



Dremio Software

Migrate a BI Workload to the Dremio Semantic Layer

This document aims to provide a methodology for taking complex queries, often auto-generated by BI tools, and re-engineering them into semantic layer views that conform to Dremio's Semantic Layer Best Practices.

The document introduces a fictitious data model implemented inside a BI tool. It then showcases some queries generated by the BI tool and submitted to the connected data source to populate data onto the dashboard widgets.

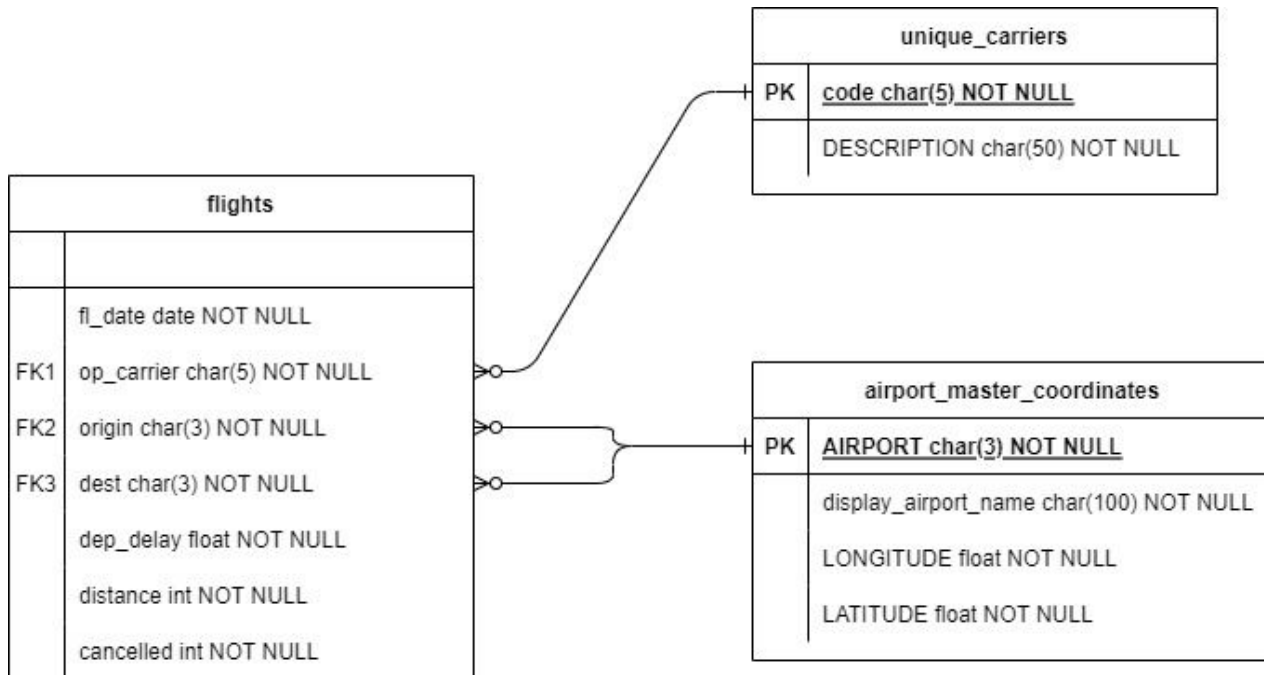
Given the sets of queries that are generated by the BI tool, the document walks through a methodology for breaking those complex queries down into views in Dremio's Semantic Layer that can act as reusable building blocks to simplify how you expose the data required for each widget, giving the same results, whilst also providing the option to leverage Dremio's acceleration techniques to yield improved performance when the BI tool retrieves the data directly from Dremio.

This ultimately means the complexity of the queries, which has manifested due to creating the data model inside the BI tool, can be extracted into easy-to-use views in Dremio. The data needed for each widget can be served by dedicated views that can be further accelerated if required.

BI Tool Data Model

This data model relates to flight information that an end user modeled in their BI Dashboard. The model contains the following datasets:

- flights - includes details of all flights made by passenger airlines over several years.
- unique_carriers - mapping of an airline's 3-letter IATA code to the full airline name.
- Airport_master_coordinates - details of the global coordinate locations of airports, their IATA code and the full airport name.



Dashboard Queries

The dashboard issues the following three queries to Dremio.

Query 1 - Longest flights per airline

```
SELECT
  a.DESCRPTION as "airline_name",
  CONCAT(f.origin, '-', f.dest) as "route",
  origins.display_airport_name as origin_airport_name,
  destinations.display_airport_name as dest_airport_name,
  AVG(f."distance") as "distance"
FROM "flights" f
INNER JOIN unique_carriers a ON f.op_carrier = a.code
INNER JOIN "airport_master_coordinates" origins on f.origin = origins.AIRPORT
INNER JOIN "airport_master_coordinates" destinations on f.dest = destinations.AIRPORT
WHERE TO_DATE(f.fl_date, 'YYYY-MM-DD') BETWEEN '2023-01-01' AND '2023-12-31'
AND f.cancelled = 0
GROUP BY "airline_name", "route", origin_airport_name, dest_airport_name
ORDER BY "distance" DESC
```

Query 2 - Total flights by airline and origin

```
SELECT
    a.DESCRPTION as airline_name,
    f.origin,
    COUNT(f.origin) as "total_flights"
FROM "flights" f
INNER JOIN unique_carriers a ON f.op_carrier = a.code
INNER JOIN C"airport_master_coordinates" origins on f.origin = origins.AIRPORT
INNER JOIN "airport_master_coordinates" destinations on f.dest = destinations.AIRPORT
WHERE TO_DATE(f.fl_date, 'YYYY-MM-DD') BETWEEN '2023-01-01' AND '2023-12-31'
AND f.cancelled = 0
GROUP BY "airline_name", "origin"
ORDER BY "total_flights" DESC
```

Query 3 - Average departure delays on a specific route by airline

```
SELECT
    a.DESCRPTION as airline_name,
    origins.display_airport_name as origin_airport_name,
    destinations.display_airport_name as dest_airport_name,
    AVG(f.dep_delay) as avg_departure_delay,
    COUNT(f.origin) as "num_flights"
FROM "flights" f
INNER JOIN unique_carriers a ON f.op_carrier = a.code
INNER JOIN "airport_master_coordinates" origins on f.origin = origins.AIRPORT
INNER JOIN "airport_master_coordinates" destinations on f.dest = destinations.AIRPORT
WHERE TO_DATE(f.fl_date, 'YYYY-MM-DD') BETWEEN '2023-01-01' AND '2023-12-31'
AND f.cancelled = 0
AND origins.AIRPORT = ?
AND destinations.AIRPORT = ?
GROUP BY "airline_name", origin_airport_name, dest_airport_name
ORDER BY "avg_departure_delay" DESC
```

Methodology

The following high-level methodology steps assume you have already connected to the same data source in Dremio that the BI tool is also connecting to and that you have set up Preparation, Business and Application layer folders within a space in our Semantic Layer. Each step is broken down further in the subsections.

1. Create a view in the Preparation layer for each raw data table in the data model, apply relevant casts and filters, and create any required derived columns in the Preparation layer views.
2. Create a view in the business layer for each view in the Preparation layer
3. Create canonical/business model views from the base views in the Business layer
4. Create a view in the application layer for each view in the Business layer

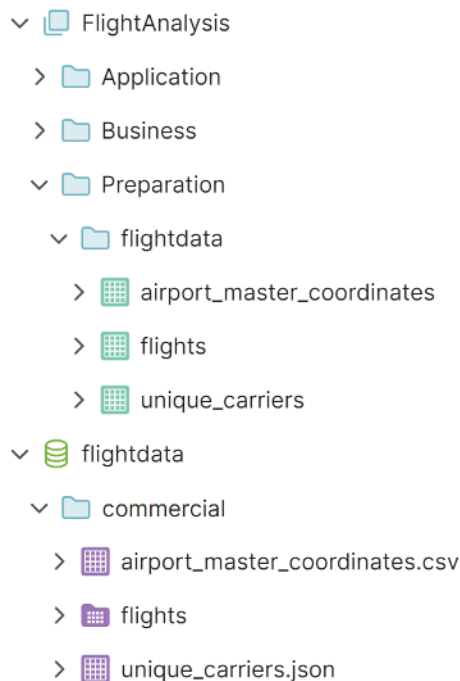
5. Create Application layer views that query the canonical model views from the Application layer and refine them by adding any further required joins/aggregations needed to expose the data in the format that can be consumed by the individual widgets in the BI tool.
6. Update the widgets in the BI tool to get their data from the relevant views exposed in the Application layer.
7. Perform end-to-end testing of the solution, initially without applying reflections.

Create Preparation Layer views

The assumption has already been made that you have connected to the same data source in Dremio that the BI tool is connected to. However, confirm that every table referenced in your complex query is also exposed as a table in the data source in Dremio.

If one doesn't already exist, create a folder beneath the Preparation folder named the same as the data source. Inside the folder that is named the same as the data source, for each table in the data source create a view with the same name as the table in the folder.

Example:



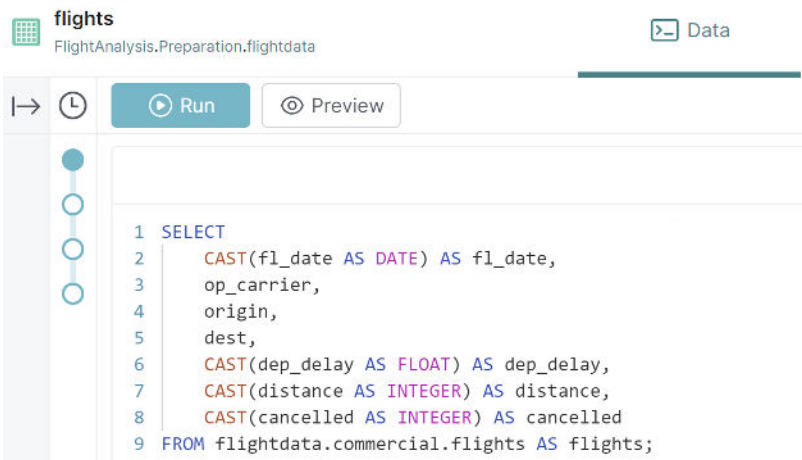
For each of the three views in our fictitious data model created in the source folder of the Preparation layer, you should do several further manipulations in the Preparation layer to conform to our Semantic Layer Best Practices, highlighted in the subsections below.

Cast data types

Where applicable, for example when you have promoted a CSV file as a table in a data source or if the data types exposed by the source for specific columns are not representative of the data in the column, ensure the data type associated with each column is accurate by adding the necessary CASTs in the Preparation layer view to achieve this.

There are several examples of this in the flights table of our data model. The physical flights table has columns entirely defined as strings, as shown below, but the data model has several columns defined as other types, including date, float and integer. Therefore, we can apply the following cast statements in our flights view in the Preparation layer:

```
CAST(f1_date AS DATE) AS f1_date,
CAST(dep_delay AS FLOAT) AS dep_delay,
CAST(distance AS INTEGER) AS distance,
CAST(cancelled AS INTEGER) AS cancelled
```



The screenshot shows the Dremio interface for a view named 'flights'. The breadcrumb path is 'FlightAnalysis.Preparation.flightdata'. The query editor contains the following SQL code:

```
1 SELECT
2   CAST(f1_date AS DATE) AS f1_date,
3   op_carrier,
4   origin,
5   dest,
6   CAST(dep_delay AS FLOAT) AS dep_delay,
7   CAST(distance AS INTEGER) AS distance,
8   CAST(cancelled AS INTEGER) AS cancelled
9 FROM flightdata.commercial.flights AS flights;
```

Create derived columns

Identify any non-aggregate derived columns in the queries comprising calculations/functions over fields in the same table and create that derived column into the view.

An example of this can be taken from query 1, where the following derived column can be added to the flights view in the Preparation layer:

```
CONCAT(origin, '-', dest) as "route"
```



Apply guaranteed filters for data that will NEVER be needed

⚠ IMPORTANT NOTE

If you cannot guarantee that the filtered data will never be needed in any downstream views, do not add the filter in the Preparation Layer.

If there are any common filters that you can GUARANTEE will always be applied to ALL views derived from your Preparation layer view, then add that filter into the view at the Preparation layer.

An example of this exists in the data model presented in this document whereby all of the example queries contain a common filter on the `flights` view, so if you can guarantee that ALL queries that ever hit the `flights` view will leverage that filter then it can be placed in a view in the Preparation layer. The view in our scenario will contain this permanent filter:

```

WHERE TO_DATE(fl_date, 'YYYY-MM-DD') BETWEEN '2023-01-01' AND '2023-12-31'
AND cancelled = 0

```

The screenshot shows the Dremio interface for a query named 'flights'. The query is as follows:

```

1 SELECT *
2 FROM (
3     SELECT
4         CAST(fl_date AS DATE) AS fl_date,
5         op_carrier,
6         origin,
7         dest,
8         CONCAT(origin, '-', dest) as "route",
9         CAST(dep_delay AS FLOAT) AS dep_delay,
10        CAST(distance AS INTEGER) AS distance,
11        CAST(cancelled AS INTEGER) AS cancelled
12    FROM flightdata."dremio-tech-challenge".flights AS flights) a
13 WHERE TO_DATE(fl_date, 'YYYY-MM-DD') BETWEEN '2023-01-01' AND '2023-12-31'
14 AND cancelled = 0

```

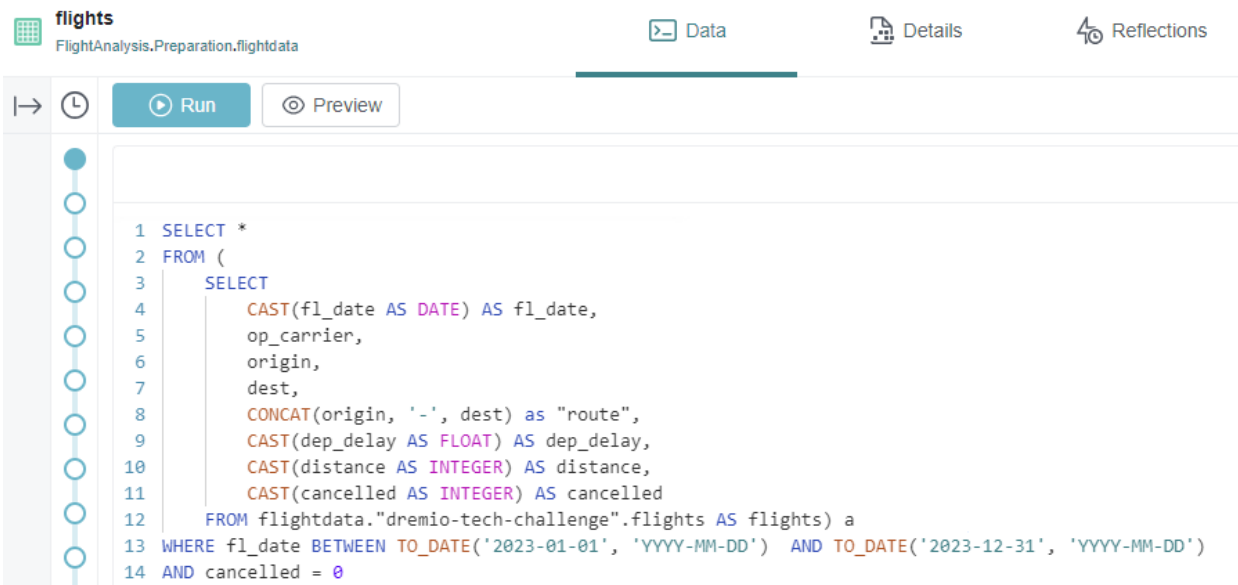
Apply transformations to scalar expressions, not columns

Using the filter in the previous section as an example, this filter is inefficient. When comparing a column value (be it a partition key, primary key or any column value) to a literal as above, if a transformation is required before a comparison, then be sure to apply the transformation to the static/literal side and not to the column value(s). Applying the transformation to the column side means the function has to be applied to every record to check for equality, whereas applying it to the static value means the transformation only needs to happen once and the result can be compared against the raw column value. With that rule in mind, the filter from the previous section can be re-written as follows to make it more efficient:

```

WHERE fl_date BETWEEN TO_DATE('2023-01-01', 'YYYY-MM-DD') AND
TO_DATE('2023-12-31', 'YYYY-MM-DD')

```



```

1 SELECT *
2 FROM (
3     SELECT
4         CAST(fl_date AS DATE) AS fl_date,
5         op_carrier,
6         origin,
7         dest,
8         CONCAT(origin, '-', dest) as "route",
9         CAST(dep_delay AS FLOAT) AS dep_delay,
10        CAST(distance AS INTEGER) AS distance,
11        CAST(cancelled AS INTEGER) AS cancelled
12    FROM flightdata."dremio-tech-challenge".flights AS flights) a
13 WHERE fl_date BETWEEN TO_DATE('2023-01-01', 'YYYY-MM-DD') AND TO_DATE('2023-12-31', 'YYYY-MM-DD')
14 AND cancelled = 0

```

The general rule is: If transformation is needed, transform the scalar side of the expression, not the column side (i.e. `WHERE partition_key = fn('some_value')`, not `WHERE fn(partition_key) = 'some_value'`)

This ensures that the transform will perform optimally whether Dremio performs the transform or whether it is pushed to the underlying data source.

Additional Preparation Layer Manipulations

As well as the manipulations described above, which directly relate to the scenario presented in this document, several other possible manipulations would also fit into the Preparation layer if these were relevant to your situation, for example leveraging a reflection to repartition a dataset.

Leverage reflection to repartition dataset

If an underlying dataset is partitioned in a particular way, e.g. on a date column, but a significant portion of the query workload will be filtering or joining by a different column, then it can make sense in the Preparation layer to introduce a raw reflection partitioned on the different column, thus enabling partition pruning on otherwise expensive workloads to accelerate them.

Create base Business Layer views

It is always good practice to create a one-to-one view in the Business layer for every view that you create in the Preparation layer; this is so that if required, each table can essentially be exposed as-is up through each layer without being joined to any other view in Dremio to cater for situations where client tools would like to consume the raw data assets.

The screenshot shows the Dremio interface. On the left, a dark sidebar contains navigation icons for 'Datasets', 'Sources', and 'Object Storage'. The main content area is divided into two panels. The left panel, titled 'Datasets', shows a tree view with 'dremio' at the top, followed by 'Spaces (1)' containing 'FlightAnalysis' (8 items), 'Sources' (0 items), and 'Object Storage (1)' containing 'flightdata' (3 items). The right panel, titled 'FlightAnalysis.Business', shows a list of datasets: 'airport_master_coordinates', 'flights', and 'unique_carriers'. A search bar at the top reads 'Search Spaces and Datasets'.

Create canonical/business model views in the Business Layer

Using the initial Business layer views created in the previous step as a base, create the required canonical/business model views in the Business layer. In the scenario in this document, this means a single canonical model object called `flight_details`, which joins the three base views in the Business layer into a single view:

```
Space.Business."flights" f
INNER JOIN Space.Business.unique_carriers a ON f.op_carrier = a.code
INNER JOIN Space.Business."airport_master_coordinates" origins on f.origin =
origins.AIRPORT
INNER JOIN Space.Business."airport_master_coordinates" destinations on f.dest =
destinations.AIRPORT
```

In more complex scenarios, several layers of views may be created in the Business layer as you create reusable building blocks that build up to your canonical/business model. These reusable views can benefit the solution because they can be places to add reflections if you find that querying the model is slow during query execution.

Remove duplicate columns

One of the side-effects of creating a canonical model like the one described in this scenario is that in the view created by performing the joins above, you will be left with duplicate columns in the projection list, for example the columns used in the joins; `f.op_carrier` is essentially the same as `a.code`. Dremio recommends removing duplicate columns.

Rename columns if the same column name appears in the canonical model

Sometimes when joining tables together you will end up with clashes in column names between the tables. Dremio handles this by adding an incrementing number to the end of each duplicate column name it finds; for example, if two tables, let's call them a and b, were joined and each contains a column called code, then Dremio will continue to call the first column code and it will rename the second column to code0. This makes it difficult to determine which column is associated with which original table. Dremio recommends renaming the duplicate columns to reference the table they originated from, e.g. a_code and b_code.

The screenshot shows the Dremio SQL editor interface. At the top, there's a header with 'flight_details' and 'FlightAnalysis.Business'. Below that are tabs for 'Data', 'Details', and 'Reflections'. The main area contains a SQL query with line numbers 1 through 25. The query is a SELECT statement with various columns and table aliases, including joins for carriers, origins, and destinations.

```

1 SELECT
2   fl_date,
3   op_carrier,
4   origin,
5   dest,
6   route,
7   dep_delay,
8   distance,
9   cancelled,
10  a.Description AS "airline_name",
11  origins.AIRPORT AS origin_airport,
12  origins.DISPLAY_AIRPORT_NAME AS origin_display_airport_name,
13  origins.LATITUDE AS origin_latitude,
14  origins.LONGITUDE AS origin_longitude,
15  destinations.AIRPORT AS dest_airport,
16  destinations.DISPLAY_AIRPORT_NAME AS dest_display_airport_name,
17  destinations.LATITUDE AS dest_latitude,
18  destinations.LONGITUDE AS dest_longitude
19 FROM FlightAnalysis.Business."flights" f
20   INNER JOIN FlightAnalysis.Business.unique_carriers a
21   ON f.op_carrier = a.code
22   INNER JOIN FlightAnalysis.Business."airport_master_coordinates" origins
23   ON f.origin = origins.AIRPORT
24   INNER JOIN FlightAnalysis.Business."airport_master_coordinates" destinations
25   ON f.dest = destinations.AIRPORT

```

Additional Business Layer Manipulations

The aspects mentioned above apply to the scenario in this document. However, as you create your canonical model objects other factors might need your consideration; several of these are documented below.

Join considerations

When creating the joins, consider the following aspects to ensure you maximize the ability to do fast columnar scans:

1. If joining a fact to a dimension table on column(s) where a function must be applied to one or the other column to get the columns to match, apply the function on the smaller side of the join
2. If joining to subselects, particularly ones containing aggregations (select max(), select distinct, etc.), consider creating a view with a reflection on the subselect.
3. Attempt to factor out repeated subselects to a common set of views, which can be reflected and reused across many joins.

Scalar subqueries

If you have a query that contains a scalar subquery (defined as a subquery that returns a single value - one column and one row) that is filtering the query, e.g.:

```
SELECT name, cost
FROM product
WHERE id=(SELECT product_id
          FROM sale
           WHERE price=5000
           AND product_id=product.id
          );
```

This is inefficient because the inner query needs to be executed for each record in the outer query. Instead, you should create a view with an INNER JOIN to make things more efficient, e.g.:

```
SELECT p.name, p.cost
FROM product p
INNER JOIN sale s ON p.id=s.product_id
WHERE s.price=5000;
```

Subqueries in an IN clause

If you have a query that contains a subquery in an IN clause, e.g.:

```
SELECT name, cost
FROM product
WHERE id IN (SELECT product_id FROM sale);
```

This is not very efficient because again the inner query needs to be executed for each record in the outer query. Instead, you should create a view with an INNER JOIN to make things more efficient:

```
SELECT DISTINCT p.name, p.cost
FROM product p
INNER JOIN sale s ON s.product_id=p.id;
```

Note using DISTINCT in the query above to remove duplicates; this is often necessary if you transform queries with an IN or a NOT IN clause into an INNER JOIN.

Subqueries in a NOT IN clause

If you have a query that contains a subquery in a NOT IN clause, e.g.:

```
SELECT name, cost
FROM product
WHERE id NOT IN (SELECT product_id FROM sale);
```

This could be more efficient. Instead, you should create a view with a LEFT OUTER JOIN and a WHERE clause:

```
SELECT DISTINCT p.name, p.cost
FROM product p
LEFT OUTER JOIN sale s ON s.product_id=p.id
WHERE s.product_id IS NULL;
```

Correlated Subqueries in EXISTS and NOT EXISTS clauses

Subqueries in an EXISTS or a NOT EXISTS clause are easy to rewrite with LEFT OUTER JOINS. Let's say you have the following query that is used to obtain details of products NOT sold in the year 2023:

```
SELECT name, cost, city
FROM product
WHERE NOT EXISTS ( SELECT id
                   FROM sale WHERE year=2023 AND product_id=product.id );
```

This can be rewritten in a view with a LEFT OUTER JOIN for improved efficiency as follows:

```
SELECT p.name, p.cost, p.city FROM product p
LEFT OUTER JOIN sale s ON s.product_id=p.id
WHERE s.year!=2023 OR s.year IS NULL;
```

If the query were an EXISTS clause, the WHERE clause in the JOIN version of the query above would simply be `WHERE s.year=2023` instead.

Correlated Subqueries with aggregates in the projection column

If a query contains a subquery to derive an aggregate value in the list of projected columns, e.g.:

```
SELECT
  main.project_name,
  main.project_budget,
  main.project_start_ym,
  (SELECT SUM(sub.project_budget)
   FROM projects sub
   WHERE sub.project_start_ym <= main.project_start_ym) AS total_budget
FROM
  projects main
ORDER BY main.project_start_ym ASC
```

Then you can re-write this in a view using a window function as follows:

```
SELECT
  project_name,
  project_budget,
  project_start_fiscal_ym,
  SUM(main.project_budget) OVER (ORDER BY project_start_fiscal_ym) AS total_budget
from projects
```

Common Table Expressions (CTEs)

If the query contains a Common Table Expression (CTE) (e.g. there is a SELECT query in a WITH clause), create a view of the CTE and use that view when joining it back into the main query. Avoid using WITH clauses in the Semantic layer because the queries in the WITH clauses can end up being executed multiple times.

Create base Application Layer views

Much like we said about the base Business layer views, it is always good practice to create a one-to-one view in the Application layer for every base view that you create in the Business layer; this is so that if required, each table can essentially be exposed as-is up through each layer without being joined to any other view in Dremio to cater for situations where client tools would like to consume the raw data assets. This is often useful for self-service reporting use cases.

The screenshot shows the Dremio interface. On the left is a navigation sidebar with icons for Datasets, Spaces, Sources, and Object Storage. The main area is split into two panels. The left panel, titled 'Datasets', shows a tree view under 'dremio' with 'Spaces (1)' containing 'FlightAnalysis' (12 items) and 'Sources' containing 'Object Storage (1)' with 'flightdata' (3 items). The right panel, titled 'FlightAnalysis.Application', shows a list of views: 'airport_master_coordinates', 'flight_details', 'flights', and 'unique_carriers', each with a grid icon and a name arrow icon.

Create bespoke Application Layer views

In the Application layer, you may need to take the Business layer objects and create further joins/aggregations upon them to expose data in a format ready to be consumed by end user tools. If there are use case-specific filters that need to be applied then these are best to be placed in the Application layer.

Update widgets in the BI tool

Before testing the solution, you must update the widgets in your BI tool to gather their data from the views exposed from the Application layer in Dremio rather than the internal data model. Taking a copy of your dashboard before making these changes is recommended.

Test the solution

Once the end-to-end solution is created, test how it performs without any reflections by leveraging the same tools the end user would use or, at worst, by simulating the same query that those tools would issue to Dremio.

In our testing of the queries in this scenario, we noted these query times:

	Original query execution directly against tables via Dremio	Query execution against the Application layer view in the Dremio model
Query 1	28s / 13046 rows	8s / 13046 rows
Query 2	23s / 1967 rows	7s / 1967 rows
Query 3	2s / 5 rows	1s / 5 rows

If a query is slow, consider where it is slow and determine if there are views in the query plan where a reflection could be placed to accelerate multiple workloads, not just the specific query. In our scenario, since all three queries are accessing the same canonical model object and all that is different is the way the data is grouped and aggregated, it should be possible to create an Aggregate Reflection on the Business layer canonical model object that all three queries can make use of. Let us analyze this further below: we can see that several fields are being grouped by and several measures.

Superset of fields used in GROUP BY clauses in the three queries: "airline_name", "route", origin_airport_name, dest_airport_name

Superset of aggregated measures used in the three queries: AVG(f."distance"), COUNT(f.origin), AVG(f.dep_delay)

Therefore, we can create an Aggregate Reflection to help accelerate every query in our solution as follows:

The screenshot shows the Dremio interface for the 'flight_details' dataset. The 'Reflections' page is open, and the 'Aggregation Reflections' tab is selected. A table titled 'Aggregation Reflections' is displayed, showing the configuration for various columns. The table has columns for Dimension, Measure, Sort, and Partition. The 'Measure' column contains green checkmarks for several rows, indicating that reflections are active for those columns. The 'Footprint' is listed as 428.72 KB.

Search columns...	Dimension	Measure	Sort	Partition
fl_date	-	-	-	-
op_carrier	-	-	-	-
origin	✓	✓	-	-
dest	✓	-	-	-
route	✓	-	-	-
dep_delay	-	✓	-	-
distance	-	✓	-	-
cancelled	-	-	-	-
airline_name	✓	-	-	-

Running each of the queries in this scenario again, you can see that they all leverage that single Aggregate Reflection and they all take **less than a second** to complete:

Query 1 (<1s / 13046 rows)

The screenshot shows the Dremio interface for a specific query. The 'Jobs' page is open, and the 'Summary' tab is selected. The query status is 'COMPLETED'. The 'Submitted SQL' tab shows the query text. The 'Queried Datasets' section lists 'flight_details'. The 'Scans' section shows 'Aggregation Reflection (flight_details)'. The 'Acceleration' section shows 'Query was Accelerated'.

Summary

- Status: COMPLETED
- Total Memory: 29.61 MB
- CPU Used: <1s
- Query Type: UI (run)
- Start Time: 24/10/2023 11:12:38
- Duration: <1s
- Wait on Client: <1s
- User: dremio
- Queue: SMALL
- Input: 826.79 KB / 13K Rows
- Output: 723.23 KB / 13K Rows

Total Execution Time <1s (100%)

- Pending: 1ms (0.12%)
- Metadata Retrieval: 588ms (70.67%)
- Planning: 45ms (5.41%)
- Queued: 10ms (1.20%)
- Execution Planning: 12ms (1.44%)
- Starting: 4ms (0.48%)
- Running: 172ms (20.67%)

Submitted SQL

```

1 SELECT
2   "air_line_name",
3   "route",
4   origin,
5   dest,
6   AVG("distance") as "distance"
7 FROM FlightAnalysis.Application."flight_details"
8 GROUP BY "air_line_name", "route", origin, dest
9 ORDER BY "distance" DESC
    
```

Queried Datasets

- flight_details (FlightAnalysis.Application.flight_details)

Scans

- Aggregation Reflection (flight_details)

Acceleration

- Query was Accelerated

Reflections Used

- Aggregation Reflection (FlightAnalysis.Application.flight_details) (Age: 14 minutes)

Reflections Not Used

Query 2 (<1s / 1967 rows)

Jobs » 1ac86764-a54a-b7e9-fab7-e6d89f29d500
Overview SQL Raw Profile

Summary

Status: COMPLETED

Total Memory: 36.16 MB

CPU Used: <1s

Query Type: UT (run)

Start Time: 24/10/2023 11:12:42

Duration: <1s

Wait on Client: <1s

User: dremio

Queue: SMALL

Input: 490.71 KB / 13K Rows

Output: 72.70 KB / 2K Rows

Total Execution Time <1s (100%)

Pending	1ms (0.13%)
Metadata Retrieval	49ms (66.71%)
Planning	48ms (6.42%)
Queued	16ms (2.14%)
Execution Planning	16ms (2.14%)
Starting	5ms (0.67%)
Running	163ms (21.79%)

Submitted SQL

```

1 SELECT
2   airline_name,
3   origin,
4   COUNT(origin) as "total_flights"
5 FROM FlightAnalysis.Application."flight_details"
6 GROUP BY "airline_name", "origin"
7 ORDER BY "total_flights" DESC
        
```

Queried Datasets

flight_details
FlightAnalysis.Application.flight_details

Scans

> ⚡ Aggregation Reflection (flight_details)

Acceleration ⚡ Query was Accelerated

Reflections Used

⚡ Aggregation Reflection
FlightAnalysis.Application.flight_details Age: 15 minutes

Reflections Not Used

Query 3 (<1s / 5 rows)

Jobs » 1ac86762-6afb-59da-80fd-8d45a388de00
Overview SQL Raw Profile

Summary

Status: COMPLETED

Total Memory: 35.71 MB

CPU Used: <1s

Query Type: UT (run)

Start Time: 24/10/2023 11:12:45

Duration: <1s

Wait on Client: <1s

User: dremio

Queue: SMALL

Input: 355 B / 5 Rows

Output: 278 B / 5 Rows

Total Execution Time <1s (100%)

Pending	3ms (0.41%)
Metadata Retrieval	528ms (71.84%)
Planning	53ms (7.19%)
Queued	9ms (1.22%)
Execution Planning	23ms (3.12%)
Starting	5ms (0.68%)
Running	116ms (15.74%)

Submitted SQL

```

1 SELECT
2   airline_name,
3   origin,
4   dest,
5   AVG(dep_delay) as avg_departure_delay,
6   COUNT(origin) as "num_flights"
7 FROM FlightAnalysis.Application."flight_details"
8 WHERE origin = 'JFK'
9 AND dest = 'LAX'
10 GROUP BY "airline_name", origin, dest
        
```

Queried Datasets

flight_details
FlightAnalysis.Application.flight_details

Scans

> ⚡ Aggregation Reflection (flight_details)

Acceleration ⚡ Query was Accelerated

Reflections Used

⚡ Aggregation Reflection
FlightAnalysis.Application.flight_details Age: 16 minutes

Reflections Not Used

Additional Application Layer Considerations

As you create your solutions, other aspects might need your consideration; several of these are documented below.

Aggregate Reflections and count(distinct)

⚠ WARNING

Do not create an Aggregate reflection on measures that you might later want to use in a count(distinct) query (perhaps from a BI tool)

This is because you will get incorrect results. You need to know what the individual values are at the time you deduce your distinct values, which could change depending on the filters/ranges you apply to your data. In this situation, you must use a raw reflection.

Reflections and row-level / column-level filtering

To ensure data integrity and to ensure users are only able to see data that they are permitted to see, the use of row-level security and column-level security must occur ABOVE any use of reflections; this is to avoid the risk of a specific user's filtered records being placed into the reflection and hence giving other users access to data they shouldn't have.

Conclusion

This document has demonstrated a methodology and things to consider when migrating workloads generated by BI tools into Dremio's semantic layer, which leverages both Dremio [Semantic Layer best practices](#) and general SQL rewrite best practices to achieve performance gains whilst at the same time maintaining organized structure of your resources and promoting their reuse. Even in our relatively simple scenario, we have demonstrated how reorganizing seemingly distinct queries into a reusable semantic layer, coupled with strategically placed aggregate reflections, can vastly improve the performance of our BI queries.