Dremio Software

Semantic Layer CI/CD with Dremio and dbt

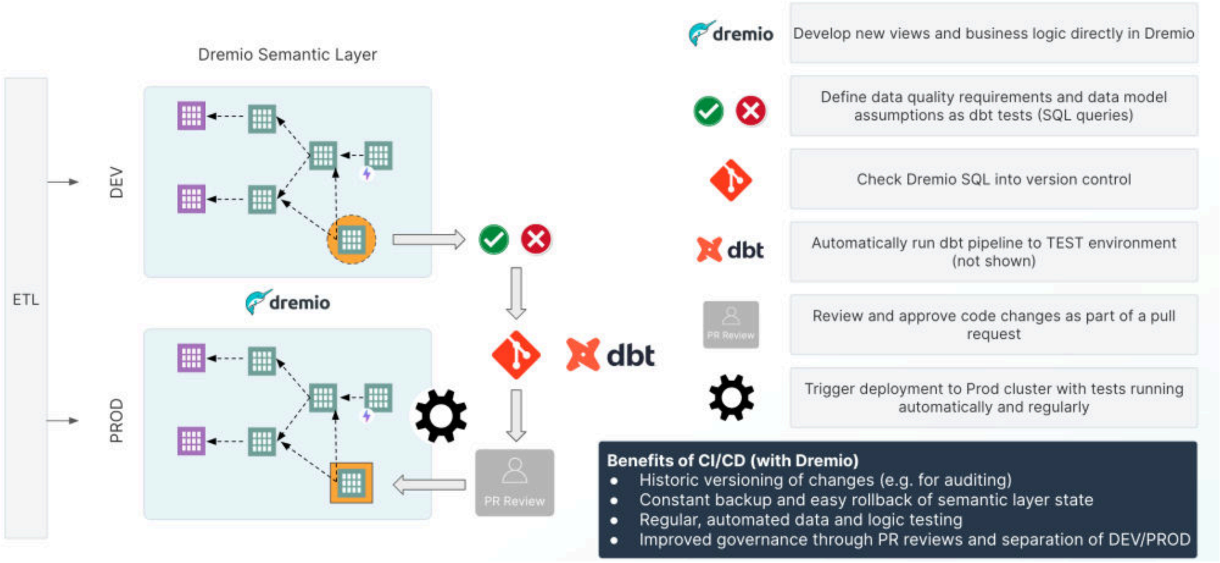
Introduction

The purpose of this document is to provide both an overview and a step-by-step guide on how to define a Dremio semantic layer in a versioned repository (like git), which makes it easy to deploy it to different environments using dbt (data build tool). It also provides a guide on how to set up materialization pipelines (that leverage Iceberg DML) in Dremio using dbt, as well as automated data quality testing.

For a starter guide on how to get your first dbt model set up and running (on Dremio Cloud), please refer to our official blog post.

Example CI/CD workflow using Dremio and dbt

The motivation for this tutorial is to showcase how you can use your SQL code repository for dataset definitions while using dbt as the workflow execution for your automated Dremio deployment pipeline, like the following example:



Develop new views and business logic directly in Dremio

Define data quality requirements and data model assumptions as dbt tests (SQL queries)

Check Dremio SQL into version control

Automatically run dbt pipeline to TEST environment (not shown)

Review and approve code changes as part of a pull request

Trigger deployment to Prod cluster with tests running automatically and regularly

Benefits of CI/CD (with Dremio)

- Historic versioning of changes (e.g. for auditing)
- Constant backup and easy rollback of semantic layer state
- Regular, automated data and logic testing
- Improved governance through PR reviews and separation of DEV/PROD

Important resources

The following docs pages are referenced in this paper and provide additional information:

- Dremio Docs - dbt
- Dremio-dbt connector Docs - Using Materializations with Dremio
- Dremio-dbt connector - Code repository
- dbt Docs
- “The Ultimate Guide to dbt” - Medium article with a comprehensive canvas board

Defining Dremio objects in dbt

Most Dremio objects can be defined using SQL. Since dbt follows a “SQL-first” approach, these object types automatically become compatible with dbt as long as they support an idempotent creation syntax. As of January 2024, a small subset of Dremio objects require using the Dremio dbt connector’s built-in features; some will need to be released later.

Objects that can be created and updated idempotently using SQL via dbt:

- Views
- Tables (Promoting existing datasets)
- Iceberg Tables (Creating or updating datasets)
- Spaces & Folders
- User-defined Functions
- Row- & Column-level Security (using UDFs)
- Users & Roles ***(not recommended)***
- RBAC privileges (grants)

Objects that can be created idempotently using the Dremio dbt connector:

- Reflections

Objects that cannot be created via dbt:

- Sources
- Wikis & Tags

In the following paragraphs, we will show how to define each object in dbt, followed by a comprehensive example that you can run on your cluster.

Required privileges for running dbt against Dremio

To run the underlying REST API calls of dbt against Dremio, the (service) user account requires the following privileges in Dremio:

- Read access to Dremio’s INFORMATION_SCHEMA tables
- Read access to the reflection system table (e.g. GRANT SELECT ON TABLE sys.reflections TO USER <XYZ>)
- Read access to the required data source (SELECT & ALTER)
- Read and write access to the target data source (SELECT, ALTER, CREATE TABLE, DROP, INSERT)
- Read and write access to the target Dremio Space/Folders (SELECT, ALTER, VIEW REFLECTION, ALTER REFLECTION)

Views

Creating views of data is at the heart of the Dremio semantic layer. In dbt, these views are defined as SQL text files that contain the corresponding SELECT statement. This assumes that the default materialization for the dbt model is set to `+materialized: view` in the `dbt_project.yml` configuration. The `dbt_project.yml` is how dbt knows a directory is a dbt project. It also contains essential configuration that tells dbt how to operate your project.

For example, we may have set the variable `dremio_space` to `dbt_demo_space` (e.g. in the `dbt_project.yml`) and we have a file called `nyc_taxi_trips.sql`, which contains the following statement:

```
    SELECT *
    FROM Samples."samples.dremio.com"."NYC-taxi-trips"
```

Executing the dbt model would then generate and send the following query to Dremio:

```
    CREATE OR REPLACE VIEW dbt_demo_space.nyc_taxi_trips AS
    SELECT *
    FROM Samples."samples.dremio.com"."NYC-taxi-trips"
```

Spaces & Folders

As you saw in the previous example, Dremio's dbt connector derives the fully qualified path of a view based on the folder structure it is placed in. Dbt will create the required spaces and folders (via separate REST calls) before the SQL for view creation is sent.

Alternatively, these implicitly derived, fully qualified paths can be overridden using dbt configurations.
For example, the previous `nyc_taxi_trips.sql` could instead be written to a completely different path with a different view name in Dremio by including the following config parameters in double curly braces:

```
{{ config(
    database="dremio_space",
    schema="dremio_folder.subfolder",
    alias="taxis"
) }}
    SELECT *
    FROM Samples."samples.dremio.com"."NYC-taxi-trips"
```

The generated query to Dremio would look as follows:

```
    CREATE OR REPLACE VIEW dremio_space.dremio_folder.subfolder.taxis AS
    SELECT *
    FROM Samples."samples.dremio.com"."NYC-taxi-trips"
```

Tables

> ℹ️ NOTE
> Tables can take on different behavior in Dremio depending on their properties and context. Historically, Dremio considered all physical data in tables as immutable and read-only. With the addition of extensive Iceberg table DML capabilities for object storage sources, Dremio can now insert, update, and delete data, which opens the door for traditional ETL use cases and workflows with dbt.

Promoting existing datasets

In cases where the data is already transformed and materialized in the data lake, Dremio’s semantic layer mainly uses read operations for virtual data transformations. For Dremio to be able to read file-based tables from object storage sources, it has to collect metadata in a process called “dataset promotion”. Depending on the file type, data formatting may require additional configuration properties to succeed (see Dremio Docs).

As part of Dremio’s semantic layer best practices, we recommend having a 1-to-1 mapping between physical tables and virtual datasets in the initial “Preparation” (or “Staging”) layer. In the context of dbt, this means that we can combine the physical dataset promotion (the act of metadata formatting) and the creation of the basic view of the data (as part of the “Preparation” layer) into one dbt file.

For example, we can include the Dremio SQL command for dataset promotion as a dbt pre-hook, which means that the promotion is guaranteed to run before the main body:

```
{{ config(
    pre_hook=’ALTER TABLE Samples."samples.dremio.com"."NYC-taxi-trips"
              REFRESH METADATA AUTO PROMOTION‘
) }}
      SELECT *
      FROM Samples."samples.dremio.com"."NYC-taxi-trips"
```

Executing the dbt model would now generate and send two queries to Dremio:

```
      ALTER TABLE Samples."samples.dremio.com"."NYC-taxi-trips"
      REFRESH METADATA AUTO PROMOTION
```

and

```
      CREATE OR REPLACE VIEW dbt_demo_space.nyc_taxi_trips AS
      SELECT *
      FROM Samples."samples.dremio.com"."NYC-taxi-trips"
```

Writing Iceberg tables

Dbt supports several types of materializations. We cover the materialization types “view” and “materialized view” (or Dremio’s equivalent “reflection”) in other sections of this document. This part will discuss “table” and “incremental” materialization strategies.

It is important to remember that Dremio has a strict separation of tables and views unless you use Dremio’s lakehouse management service on Dremio Cloud or the Nessie catalog as a source on Dremio software. Object storage sources in Dremio (like S3 or ADLS) only allow the creation of (Iceberg) tables, while traditional Dremio spaces and folders only allow the creation of views. This means that the definition of the semantic layer objects (views) follows rules different from those used to create data materializations (tables).

> ℹ️ NOTE
> The Dremio dbt-connector allows setting rules that prevent (or enforce) having tables and views with identical paths and names inside the same Dremio environment via the `twin_strategy` setting.

Table materializations

In this example, we will create an Iceberg table in our object storage data source based on Dremio’s `sys.jobs_recent` system table, released as part of Dremio v24.3. In the following `jobs_recent_full.sql` file, our Azure object storage source is called “`azurestorage`” and it contains the folder “`table-folder`”:

```
{{ config(
    object_storage_source="azurestorage",
    object_storage_path="table-folder",
    database="dremio_space",
    materialized="table"
) }}
SELECT *
FROM sys.jobs_recent
```

When we run the dbt model, it will create the following three SQL statements against Dremio, which will make sure a new table is created and a corresponding view is created in Dremio’s semantic layer (space):

```
drop table if exists "azurestorage"."table-folder"."jobs_recent_full";
create table "azurestorage"."table-folder"."jobs_recent_full" as (
SELECT *
FROM sys.jobs_recent
);
```

```
create or replace view "dremio_space"."jobs_recent_full" as (
    select *
    from "azurestorage"."table-folder"."jobs_recent_full"
);
```

Incremental materializations

Incremental materializations follow the same approach as full table materializations. They create (or append to) the physical Iceberg table in the underlying object storage source and then create a corresponding view in Dremio's semantic layer.

Building on the example above, we can now create an incremental load on the `sys.jobs_recent` table, which will only look for new records since the last load based on the timestamp column `submitted_ts`:

```
{{ config(
    object_storage_source="azurestorage",
    object_storage_path="table-folder",
    database="dremio_space",
    materialized="incremental"
)}}
select *
from sys.jobs_recent
{% if is_incremental() %}
  -- this filter will only be applied on an incremental run
  -- (uses > to include records whose timestamp occurred since the last run of
this model)
  where "submitted_ts" > (select max("submitted_ts") from {{ this }})
{% endif %}
```

Running this model for the first time will create a new table, and every subsequent run will only retrieve and insert timestamps that are larger than the largest timestamp from the main table.

Reflections

Reflections are a critical component that allows Dremio to serve any data lake query with fast response times. Reflections can be seen as a hybrid between a traditional ETL-style materialization of data and the virtualized, view-centric approach that sits at the heart of Dremio's semantic layer philosophy.

For this reason, there are arguments for and against including reflections in defining and deploying the semantic layer. On the one hand, creating reflections goes hand in hand with the definition of the underlying views and any change to the transformation logic can have implications for materialized data downstream. On the other hand, the creation of reflections can be a very expensive job, which may be more economical to govern by a separate ETL-style batch workflow rather than having it slow down the frequent, continuous deployment pipeline. Suppose we include reflections in our main dbt model. In that case, we should avoid using raw Dremio SQL via post-hooks and instead leverage the Dremio dbt connector's built-in materialization type.

To enable reflections in dbt, we must set the following variable in the `dbt_project.yml` file:

```
vars:
  dremio:reflections_enabled: true
```

This will allow us to create a nyc_taxi_trips_refl.sql file, which builds on our previous view via the reference `{{ ref('nyc_taxi_trips') }}`:

```
{{ config(
    materialized='reflection',
    reflection_type='aggregate',
    dimensions=['passenger_count'],
    measures=['trip_distance_mi'],
    computations=['COUNT,SUM']
)}}
-- depends_on: {{ ref('nyc_taxi_trips') }}
```

A detailed guide on defining reflections in dbt can be found here.

Sources

Currently, it is impossible to create sources via dbt since Dremio sources can only be created in the UI or via REST API call. More importantly, we also do not recommend keeping the creation of data sources in the same workflow as regular tables and views since sources usually require authentication secrets, which should always be stored safely with only very selective access for administrators.

Since most use cases involve only a handful of sources, we recommend setting up a separate workflow to create and migrate those source configurations between environments, either via REST API or manually.

User-defined functions (UDFs)

User-defined functions can be defined and referenced in multiple ways inside a dbt model.

One way to define a Dremio UDF is inside of the `dbt_macros` folder, e.g. in a file called `create_udf.sql`, like this:

```
{% macro create_udf(folder_path) %}

    {% set sql %}
CREATE OR REPLACE FUNCTION {{ folder_path }}.rls_udf (passenger_count INTEGER)
RETURNS BOOLEAN
RETURN SELECT is_member('ADMIN') OR passenger_count > 1
    {% endset %}

    {% do run_query(sql) %}
    {% do log("UDF created", info=True) %}

{% endmacro %}
```

Dbt will resolve dependencies during object creation into a directed acyclic graph (DAG) to preserve the correct order. We must, therefore, guarantee that UDFs are created before the view (or views) that reference them.

One way to ensure this is to define the `on-run-start` option in the `dbt_project.yml` file to reference the `create_udf` macro from above as follows (this assumes that the destination path of the UDF, `'TEST_SPACE'`, already exists):

```
on-run-start:
  - "{{ create_udf('TEST_SPACE') }}"
```

Row-/Column-Level-Security leveraging UDFs

After we have defined the required UDFs (using the approach described in the previous paragraph), applying these UDFs as a Dremio-native Row-access or Column-Masking policy is done using dbt's post-hook functionality for sending the appropriate Dremio SQL commands. Taking the existing `nyc_taxi_trips.sql` file and the previously created `"dremio_space".rls_udf(passenger_count)` UDF, we can add the post-hook with the following syntax as part of the config:

```
{{ config(
  post_hook='ALTER VIEW {{ this }} ADD ROW ACCESS POLICY
"dremio_space".rls_udf(passenger_count)'
) }}
SELECT *
FROM Samples."samples.dremio.com"."NYC-taxi-trips"
```

Users & Roles

While it is possible to create both users and roles that are local to Dremio using Dremio SQL via dbt, it is highly recommended to instead manage users and roles through a central enterprise identity provider (IDP), such as Azure Active Directory, Okta, or LDAP, which is then synched to Dremio. For this reason, this paper will not cover the creation of users and roles via dbt.

Role-based Access Control (RBAC) privileges

Dremio supports an extensive set of privilege types and scopes. While it is possible to use dbt's built-in grants functionality to grant privileges, it is limited to granting user privileges to individual views and tables at the time of this writing. We highly recommend setting access controls for roles and on folder or space level instead because it makes setting and maintaining permissions far more manageable.

One place where RBAC can be defined in dbt is using the `on-run-end` option in the `dbt_project.yml` file. In the following example, the role `dremio_user` is an external role that Dremio can fetch from the Dremio-connected identity provider:

```
on-run-end:
  - "GRANT SELECT ON SPACE dbt_demo_space TO ROLE dremio_user"
```

Wikis & Tags

As of Dremio version 24.3, Dremio does not support the creation of wiki entries and tags via SQL. Since dbt is a “SQL-first” tool, having this feature available in dbt will either require extensions of the open-source Dremio dbt connector to support Dremio's REST API for wikis and tags or for Dremio to add SQL syntax for these two metadata fields.

Example: Defining a Dremio semantic layer in dbt

After discussing each component in a typical Dremio semantic layer, let us use them in a small dbt model.

The dbt project has the following folder and file structure:

```
dbt_cicd_demo/
      dbt_project.yml
      macros/
            create_udf.sql
      models/demo/
            nyc_taxi_trips_refl.sql
            nyc_taxi_trips_rls.sql
            nyc_taxi_trips.sql
```

dbt_project.yml

```
name: 'dbt_cicd_demo'
version: '1.0.0'
config-version: 2

profile: 'dbt_cicd_demo'

model-paths: ["models"]
macro-paths: ["macros"]

on-run-start:
  - "{{ create_udf('dremio_space') }}"
on-run-end:
  - "GRANT SELECT ON SPACE dremio_space TO ROLE dremio_user"

models:
  dbt_cicd_demo:
    demo:
      +materialized: view

vars:
  dremio:reflections_enabled: true
```

macros/create_udf.sql

```
{% macro create_udf(folder_path) %}

  {% set sql %}
CREATE OR REPLACE FUNCTION {{ folder_path }}.rls_udf (passenger_count INTEGER)
RETURNS BOOLEAN
RETURN SELECT is_member('ADMIN') OR passenger_count > 1
  {% endset %}

  {% do run_query(sql) %}
  {% do log("UDF created: {{ folder_path }}.rls_udf (passenger_count INTEGER)", info=True) %}

{% endmacro %}
```

models/demo/nyc_taxi_trips.sql

```
{{ config(
    database="dremio_space",
    schema="dremio_folder.subfolder",
    alias="taxis",
    pre_hook='ALTER TABLE Samples."samples.dremio.com"."NYC-taxi-trips" REFRESH METADATA AUTO
PROMOTION',
) }}
SELECT *
FROM Samples."samples.dremio.com"."NYC-taxi-trips"
```

models/demo/nyc_taxi_trips_refl.sql

```
{{ config(
    materialized='reflection',
    reflection_type='aggregate',
    dimensions=['passenger_count'],
    measures=['trip_distance_mi'],
    computations=['COUNT,SUM']
)}}
-- depends_on: {{ ref('nyc_taxi_trips') }}
```

models/demo/nyc_taxi_trips_rls.sql

```
{{ config(
    database="dremio_space",
    schema="dremio_folder.subfolder",
    alias="taxis_rls",
    post_hook='ALTER VIEW {{ this }} ADD ROW ACCESS POLICY
"dremio_space".rls_udf(passenger_count)'
) }}
SELECT *
FROM {{ ref('nyc_taxi_trips') }}
```

Running this model via the `dbt run` command will send the following sequence to Dremio:

```
10:04:09 Running with dbt=1.5.2
10:04:09 Registered adapter: dremio=1.5.0
10:04:09 Found 5 models, 0 tests, 350 macros, 2 operations[,..]
10:04:09
10:04:15 Running 1 on-run-start hook
10:04:16 UDF created: {{ folder_path }}.rls_udf (passenger_count INTEGER)
10:04:16 1 of 1 START hook: dbt_cicd_demo.on-run-start.0 ........... [RUN]
10:04:16 1 of 1 OK hook: dbt_cicd_demo.on-run-start.0 .............. [OK in 0.00s]
10:04:16
10:04:16 Concurrency: 1 threads (target='dev')
10:04:16
10:04:16 1 of 3 START sql view model dremio_space.[..].taxis .........[RUN]
10:04:33 1 of 3 OK created sql view model dremio_space.[..].taxis .... [OK in 16.86s]
10:04:33 2 of 3 START sql reflection model nyc_taxi_trips_refl .......[RUN]
10:04:34 2 of 3 OK created sql reflection model nyc_taxi_trips_refl ...[OK in 1.00s]
10:04:34 3 of 3 START sql view model dremio_space.[..].taxis_rls ..... [RUN]
10:04:37 3 of 3 OK created sql view model dremio_space.[..].taxis_rls .[OK in 2.90s]
10:04:37
10:04:37 Running 1 on-run-end hook
10:04:37 1 of 1 START hook: dbt_cicd_demo.on-run-end.0 ............. [RUN]
10:04:38 1 of 1 OK hook: dbt_cicd_demo.on-run-end.0 ................ [OK in 0.98s]
10:04:38
10:04:38 Finished running 2 view models, 1 reflection model, 2 hooks in 28.63s
10:04:38
10:04:38 Completed successfully
10:04:38
10:04:38 Done. PASS=3 WARN=0 ERROR=0 SKIP=0 TOTAL=3
```

Using dbt tests to validate data quality

Dbt also offers convenient ways to define, run, and report tests on datasets. These tests can be standard tests defined in YAML files (like verifying the uniqueness of column values) or custom tests written in SQL files (like checking that timestamps are within a specific date range).
For examples of how to define tests in dbt, please see the official documentation: https://docs.getdbt.com/docs/build/data-tests

Object storage sources do not provide the same dataset constraints and guarantees as relational databases (e.g. via primary keys), so it is essential to verify the underlying data's assumed properties regularly. Dremio is ideally suited for running these analytical-style data quality queries against large datasets in the data lake.

After deploying changes to your semantic layer, it is recommended that you run automated tests on your datasets to ensure that your analyses and dashboards are still correct and consistent.

Summary

This document provided an overview of how to define your entire Dremio semantic layer, including physical transformations, view definitions, and automated testing, in a versioned repository. With the power and broad support of dbt, deploying all these elements is automated, coordinated and reported out of the box, making administering and deploying changes to your Dremio production cluster a better experience.