

Database Ingestion Framework

- [Getting Started](#)
 - [Access requirements](#)
 - [Prerequisite knowledge](#)
- [Configuring Inputs](#)
 - [Framework Input Schema](#)
 - [Example Framework Input File](#)
 - [Framework Input CI/CD](#)
- [Database Ingestion DAGs](#)
 - [Framework code \(for reference, feel free to ignore\)](#)
 - [Content built by these DAGs is driven by framework inputs](#)
 - [Task Flow](#)
 - [Prep for Ingestion](#)
 - [Create where clauses](#)
 - [Insert into Bronze](#)
 - [Merge into Silver](#)
 - [Create Dremio objects](#)
- [Limitations and best practices](#)

Getting Started

Access requirements

- Github repositories:
 - [avant-data-gitops](#) (contributor access)
 - [data-ingestion-framework](#) (contains framework code for those that want to reference)
- Databricks group: DatabricksDeveloper
- Airflow:
 - [Dev](#)
 - [UAT](#)
 - [Prod](#)

Prerequisite knowledge

- [Airflow 2 Data Ingestion Framework](#)
 - Work in progress: this top-level page covers general ingestion framework concepts
- [What is a Medallion Architecture?](#)
- [Medallion Architecture + dbt](#) (only the Medallion portion is relevant for understanding Avant's implementation)
- [Databricks SQL on Airflow](#)
- [What is a SQL warehouse?](#)
- [Databricks Unity Catalog: Data Governance | Databricks](#)
- [Fundamental Concepts — Airflow Documentation](#)
- [Dynamic Task Mapping — Airflow Documentation](#) (high level understanding only)
- [Pools — Airflow Documentation](#) (high level understanding only)
- [Priority Weights — Airflow Documentation](#) (high level understanding only)

Note: Users do not need to know how to write Airflow DAGs. But it is important for users to understand basic Airflow concepts, and how to operate DAGs in Airflow

The Database Ingestion Framework is designed to enable DNA users to easily configure database ingestions from any database to our Bronze and Silver catalogs. From there, the dbt transformation framework will process it into more Silver and Gold tables.

Configuring Inputs

Framework Inputs are stored in the [avant-data-gitops](#) repository, under the `avant_airflow_2/framework_inputs/ingestion/database` directory. Below that directory, there are four more:

- default: contains “default” framework inputs. These are applied across all environments, but are overridden in each environment by corresponding files in the dev/uat/prd directories.
- dev: contains dev framework inputs
- uat: contains uat framework inputs
- prd: contains prod framework inputs

Each framework input file builds one Airflow DAG. The name of the file corresponds to the DAG to be built's `dag_id`. Each DAG will ingest the tables specified in the `tables` key.

Framework Input Schema

filename = <dag_id>.yml

```
1 dag:
2   source-schema: str() # source database schema
3   source-catalog: str() # source database catalog
4   schedule: str(required=False) # cron schedule for the DAG
5   enable_dag: bool(required=False) # must be true for DAG to build
6   description: str(required=False) # brief description for the DAG
7   owner: str(required=False) # will default to avant (sends to Data - Production Support in prod, [Test Service]
8   enable_pagerduty: bool(required=False) # set to true to generate incidents on failure
9   healthchecks: list(include('healthcheck'), required=False)
10
11 tables: list(include('table'), min=1) # list of dicts that each represent a table to be ingested
12 ---
13 table:
14   name: str() # table name
15   name-suffix: str(required=False) # this will append on to the name of your tables
16   custom-where-clause: str(required=False) # override WHERE clause for operational support use cases
17   append-where-clause: str(required=False) # append the WHERE clause to the existing chunking feature
18   delta-column: str(required=False) # column that represents created_at or updated_at of a record
19   deduplication-keys: list(min=1, required=False) # list of minimum necessary keys to deduplicate
20   days: int() # number of days of data to pull per chunk (filtered on delta-column)
21   priority-weight: int() # higher priority weight tasks are run first - see above documentation on Airflow Prior
22   source-sql-warehouse: str() # name of serverless SQL warehouse used for Medallion work
23   medallion-sql-warehouse: str() # name of SQL warehouse used for source retrieval
24   include-columns: list(min=1, required=False) # list columns to select. If blank, select *. Cannot populate thi
25   exclude-columns: list(min=1, required=False) # list columns to exclude. If blank, select *. Cannot populate th
26
27 healthcheck:
28   task_id: str() # the task_id as it appears in Airflow. Do not pick a mapped_task
29   healthcheck_url: str() # the url from HClO, must be defined in advance
```

Example Framework Input File

batch_ancillary_bar.yml

```
1 dag:
2   source-schema: public
3   source-catalog: ext_basic_prd
4   enable_dag: true
5   owner: avant
6   enable_pagerduty: true
7   healthchecks:
8     - task_id: get_runtime_config
9       healthcheck_url: <insert appropriate url here>
10    - task_id: create_where_clause_phone_numbers_do_not_call
11      healthcheck_url: <insert appropriate url here>
12    - task_id: create_where_clause_payment_plans
13      healthcheck_url: <insert appropriate url here>
14
15 tables:
16   - name: phone_numbers
17     name-suffix: do_not_call
18     delta-column: updated_at
19     deduplication-keys:
20       - uuid
21     days: 60
22     priority-weight: 2
23     source-sql-warehouse: batch-serverless
24     medallion-sql-warehouse: batch-serverless
25     append-where-clause: WHERE do_not_call
26   - name: payment_plans
27     delta-column: updated_at
28     deduplication-keys:
29       - id
30     days: 30
31     priority-weight: 1
32     source-sql-warehouse: batch-serverless
33     medallion-sql-warehouse: batch-serverless
34
```

Framework Input CI/CD

- Whenever you add, update, or remove framework inputs, you should raise a PR to avant-data-gitops using a descriptive branch name of the form `feature/<your-descriptive-name-here>`
- Whenever you push changes to any framework input files under `avant-data-gitops/avant_airflow_2/framework_inputs/*`, input validation will be triggered via Github Actions
 - The framework input files are checked for validity, and their schema is evaluated against the example schema above. You can also view example schemas [here](#)
 - [Example execution](#)
- Upon successful YAML and schema validation, framework inputs are bundled into a single yaml file of the form:

```
1 {'dag': {<dag-metadata>},
2   'dag-id-foo': {<table-entry-for-foo>},
3   'dag-id-bar': {<table-entry-for-bar>}}
```

- The YAML file representing all database framework inputs for a given environment is written to S3, where it is read from Airflow in order to build DAGs

Database Ingestion DAGs

This section outlines the DAGs produced by the database ingestion framework.

The Data Platform Pod owns a repo called `data-ingestion-framework`, which contains all code necessary to interpret the framework inputs files and build DAGs.

Framework code (for reference, feel free to ignore)

- [DAG Builder](#)
- [DAG_INITIALIZER](#)
- [Helpers](#)

The DAG initializer initializes one DAG per entry in the environment that corresponds to a given framework input file (e.g. `avant_airflow_2/framework_inputs/ingestion/database/dev.yml` builds in Airflow 2 Dev).

At a high level, the database framework DAGs do the following. More detail in the next section.

- For each entry in `framework_inputs['tables']`:
 - Insert new data into Bronze
 - Merge new data into Silver
 - Create corresponding Dremio PDSs and VDSs

Content built by these DAGs is driven by framework inputs

- Databricks:
 - `<dag.bronze-catalog>.<dag.bronze-schema>.<table.name+table.name-suffix_t>`
 - Standard path is `<env>_bronze.basic.<table.name>`
 - `<dag.silver-catalog>.<dag.silver-schema>.<table.name+table.name-suffix>`
 - Standard path is `<env>_silver.basic.<table.name>`
 - For prod, exclude the env, e.g. `bronze.basic.<table.name+table.name-suffix>`
- Dremio:
 - ```
1 pds_path = f"avantdlds-partner01-{prod_or_nonprod}.{s3_content_key.replace('s3://', '').replace('/', '.')}"
```

```
2 vds_path = f"avant.dev.silver.staging.{table['name']}"
```

## Task Flow

### Prep for Ingestion

- Process runtime argument overrides
- Generate select clause
- Create table if not exists
- Evolve schema if necessary
  - Check schema of source table vs. corresponding Bronze table
  - If a non-breaking schema change occurred, the framework will automatically accommodate it, assuming the `include-columns` key is not supplied, or the `exclude-columns` key is supplied and not exclusive of the updated schema
  - An `alter table` statement runs in this case
- Check if the Bronze table is ahead of the corresponding Silver

- This is done in terms of the `ingested_at` column created on all of these tables by these DAGs AND the records are within the bounds of the current chunk where for our mapped task.
- If it is, delete all Bronze data with `Bronze.meta_ingestion_at >= Silver.ingestion_at` for our mapped task
- Get the min and max values of `Bronze.delta-column`. We use these `min_start_time` and `min_end_time` downstream to determine what data to pull
  - If the Bronze table is empty or does not exist, it means we need to ingest the entire table
    - In this case, the min and max measurements are taken from the source table instead of bronze

## Create where clauses

- We need to append a where clause to the select statement generated in the previous task that will select all data between `min_start_time` and `max_end_time` in order to build our insert queries
- In cases where we are pulling large ranges of dates, we don't want to attempt to do that all in one query
  - In cases where the number of days between `min_start_time` and `max_end_time` is greater than `framework_inputs['table'] ['days']`, we create “chunks” that cover the number of days specified in `framework_inputs['table']['days']`
  - Example:
    - `framework_inputs['table']['days'] == 10`
    - `min_start_time == '2023-12-01:00:00:000000'`
    - `max_end_time == '2023-12-20:00:00:000000'`
    - In this case, we would generate two queries:

```
1 SELECT ... FROM ... WHERE <delta-column> > '2023-12-01:00:00:00' and <delta-column> <= '2023-12-10:99:99:999999'
2
3 # note this is cut off on the 20th, which is 9 days, because the most recent chunk is bounded by max_end_time
4 SELECT ... FROM ... WHERE <delta-column> > '2023-12-11:00:00:00' and <delta-column> <= '2023-12-19:99:99:999999'
```

- If we need to add additional filters to the existing queries, you can add the `framework_inputs['table']['append-where-clause']` to the input schema and it will merge the “chunks” clauses with the append one.
  - Example:
    - `framework_inputs['table']['days'] == 10`
    - `min_start_time == '2023-12-01:00:00:000000'`
    - `max_end_time == '2023-12-20:00:00:000000'`
    - `framework_inputs['table']['append-where-clause'] = "product_type == 'credit-card'"`
    - In this case, we would generate two queries:

```
1 SELECT ... FROM ... WHERE <delta-column> > '2023-12-01:00:00:00' and <delta-column> <= '2023-12-10:99:99:999999'
2
3 # note this is cut off on the 20th, which is 9 days, because the most recent chunk is bounded by max_end_time
4 SELECT ... FROM ... WHERE <delta-column> > '2023-12-11:00:00:00' and <delta-column> <= '2023-12-19:99:99:999999'
```

- In the vast majority of runs, we will only be pulling one date of data (because it ran yesterday), so in most cases, only one WHERE clause is created
- Another options is that if we don't need the chunking feature and only need to select certain rows from the table the `framework_inputs['table']['custom-where-clause']` is what needs to only select certain rows. Specifying this key, will override the chunking feature.
  - Example:
    - `framework_inputs['table']['custom-where-clause'] == "department == 'marketing'"`
    - In this case, we would generate only one query (ignoring the other chunking variables):

```
1 SELECT ... FROM ... WHERE department == 'marketing'
```

## Insert into Bronze

- The previous task generated a list of WHERE clauses (it might only contain one element!) that are used in conjunction with Airflow's Dynamic Task Mapping feature to create a Task Instance for each WHERE clause, which will all run under one Task
  - [Dynamic Task Mapping — Airflow Documentation](#)
- The Bronze queries are executed - they look like this:

```
1 bronze_query = f"""
2 INSERT INTO {bronze_path}.{table['name']}_t\n
3 SELECT {select_columns}, {get_metadata_fields(ingested_at, ingestion_process_id)}\n
4 FROM {table['source-catalog']}.{table['source-schema']}.{table['name']}\n
5 {where_clause}
6 """
```


- `ingested_at` is used to associate a given ingestion run (DAG run) with the records it writes. The value it uses is the Airflow `dagrun.execution_date` (poorly named, it's a timestamp). This value is set once per DAG run, and is completely independent of task instances
- `ingestion_process_id` is the ID associated with the DAG run
- After all the insert queries run, a check is made to ensure that Bronze doesn't contain duplicate records for the current ingestion date. **If the bronze task inserts data, and then is re-run within the same dagrun, it will fail this validation and the task itself will fail.**
- The duplicate query looks like this:

```
1 bronze_duplicate_query = f"""
2 SELECT {dedupe_keys}, COUNT(*) as cnt\n
3 FROM {bronze_path}.{table['name']}_t\n
4 WHERE ingestion_at = '{dag_execution_time_utc}'\n
5 GROUP BY {dedupe_keys}
6 HAVING cnt > 1
7 """
```

## Merge into Silver

- The previous task inserted records into Bronze. Now, we need to get records into Silver. In order to ensure Silver only contains the most recent versions of each record, we use a MERGE INTO query:
  - [MERGE INTO](#)
- The logic that builds the MERGE INTO query can be found below:

```
1 match_query = ''
2 for count, key in enumerate(table['deduplication-keys']):
3 match_query = match_query + f"target.{key} = source.{key}\n"
4 if count < len(table['deduplication-keys']) - 1:
5 match_query = match_query + "AND\n"
6
7 logger.info('Executing silver merge query')
8 silver_merge_query = f"""MERGE INTO {silver_path}.{table['name']} AS target\n
9 USING (SELECT {select_columns}, ingestion_at\n
10 FROM {bronze_path}.{table['name']}_t\n
11 {silver_where_clause}) AS source\n
12 ON {match_query}
13 WHEN MATCHED THEN\n
14 UPDATE SET *\n
15 WHEN NOT MATCHED THEN\n
16 INSERT *
17 """
```

- After this query executes, Silver will be up-to-date and “deduplicated”
- As a final step, we run a VACUUM query on the Silver delta table. We’re now ready to build Dremio content
  -  **VACUUM**

### Create Dremio objects

- The final task creates necessary objects to make this content usable in our legacy Dremio environment
- Each execution will build (or refresh if already exists) a PDS.
  - The PDS path is set to the S3 storage location assigned by the unity catalog.
  - The paths are not human readable, and are retrieved directly from a query to the Unity Catalog
  - The PDS path calculated by:

```
1 s3_content_key = query_helpers.get_table_path_from_catalog(table['name'], table['sql-serverless'], silver_path, d
2 pds_path = f"avantdlds-partner01-{prod_or_nonprod}.{s3_content_key.replace('s3://', '').replace('/', '.')}"
```

- After the PDS is built or refreshed, we determine whether or not the corresponding VDS needs to be built or re-built
- If the VDS does not exist, or the schema has evolved since the last run, then the VDS will be built. Otherwise, we don’t have to do anything because it’s a view.

### Limitations and best practices

- Do not put more than 50 tables into one DAG, doing so will overload Airflow’s Scheduler and cause unexpected behavior and/or errors
- This framework does not provide a solution to malformed source data. Any number of issues can occur when querying bad source data
- Be careful making the `table.days` chunk size value too large. Doing so can cause errors backpopping. 30 days is generally a good starting point
- Re-running a task has meaningful financial and output implications. There are a lot of issues in the legacy environment that are fixed with re-runs. Generally speaking, errors that arise from executions of this pipeline will not be resolved by simply re-running tasks, and in a lot of cases, that will create new problems.