SeekConnect

Help Documentation

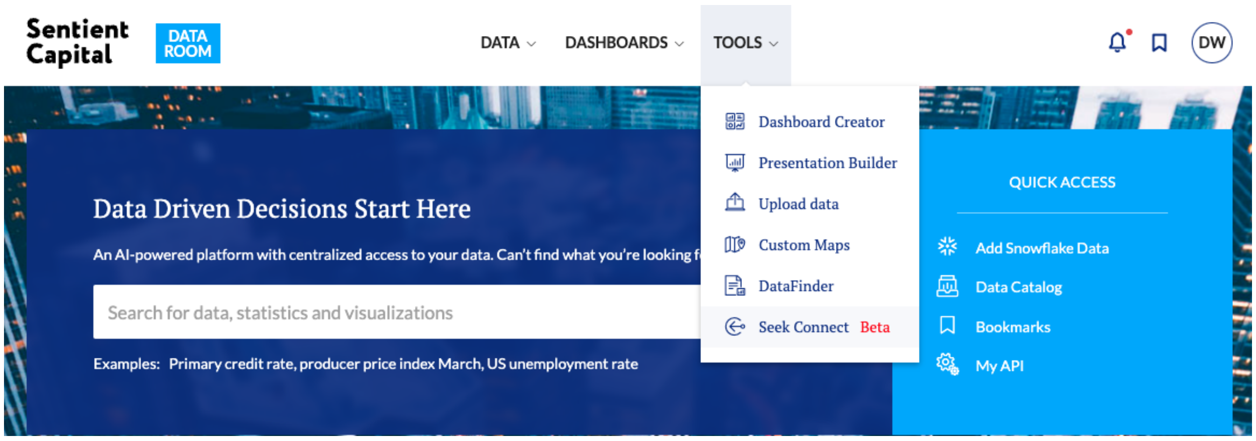
What is SeekConnect?

SeekConnect is a self-service data onboarding tool that will be made available for select pilot use near the end of Q1 2023.

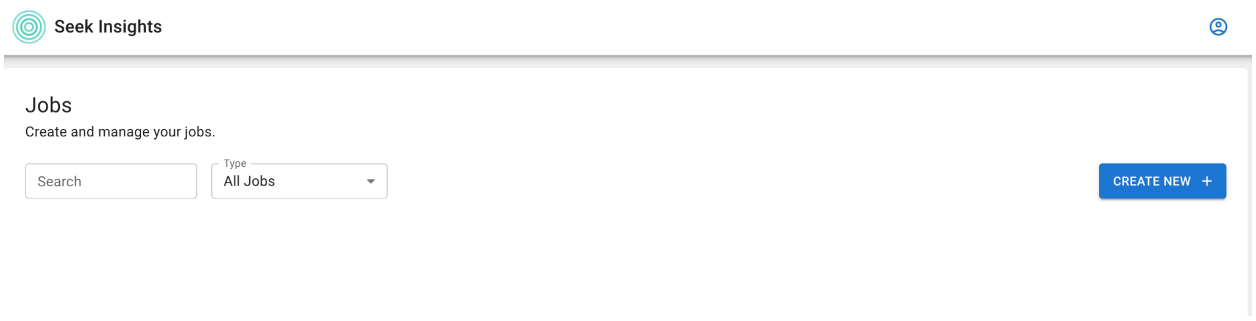
The initial version of SeekConnect supports the ingestion of CSV via S3 and SFTP, additional connectors will be added in the months to follow to enable users to extract data from other sources.

How do I access SeekConnect?

You can access SeekConnect directly in your DataHub in the tools dropdown menu, depicted below:


Sentient Capital
DATA ROOM
DATA
DASHBOARDS
TOOLS
DW
Dashboard Creator
Presentation Builder
Upload data
Custom Maps
DataFinder
Seek Connect Beta
QUICK ACCESS
Add Snowflake Data
Data Catalog
Bookmarks
My API
Data Driven Decisions Start Here
An AI-powered platform with centralized access to your data. Can't find what you're looking f
Search for data, statistics and visualizations
Examples: Primary credit rate, producer price index March, US unemployment rate

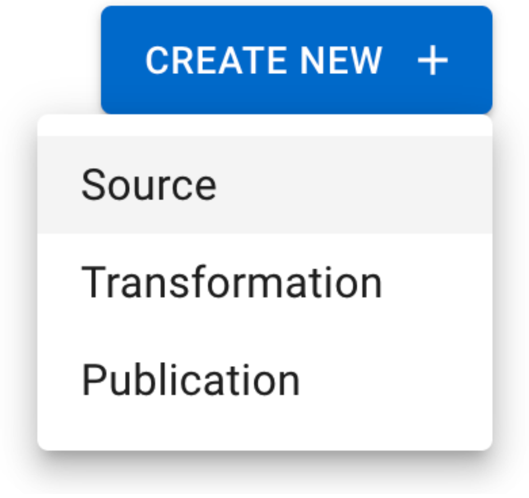
Once you select the Seek Connect tool it will navigate you to a new page:



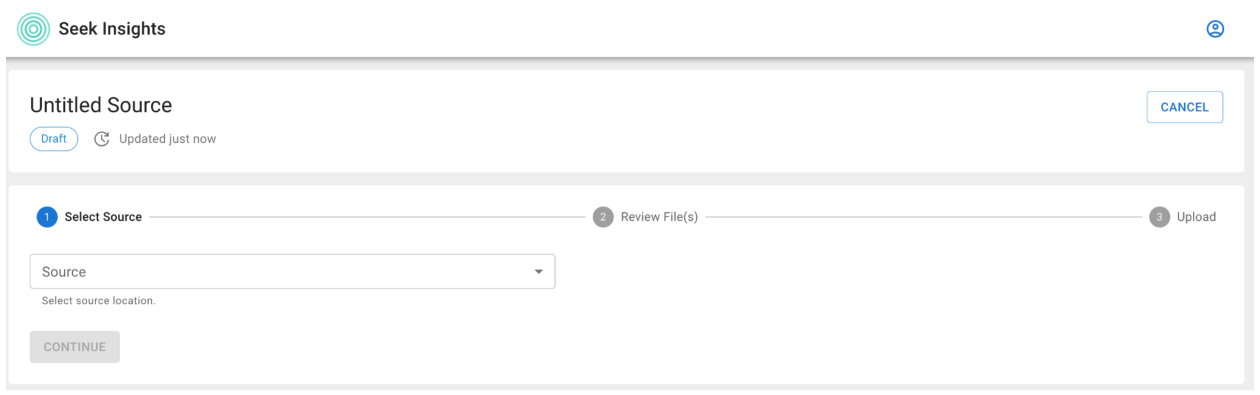
This is your SeekConnect **Jobs** page where you can manage all source connections, data transformations, and publications. To create a new **Job** simply click on the **Create New +** button on the far right. You can decide to create a new source, transformation, or publication. We will go through each of these in this document.

Step 1: How do I create a new data pipeline?

First, we will cover the process of creating a new source:

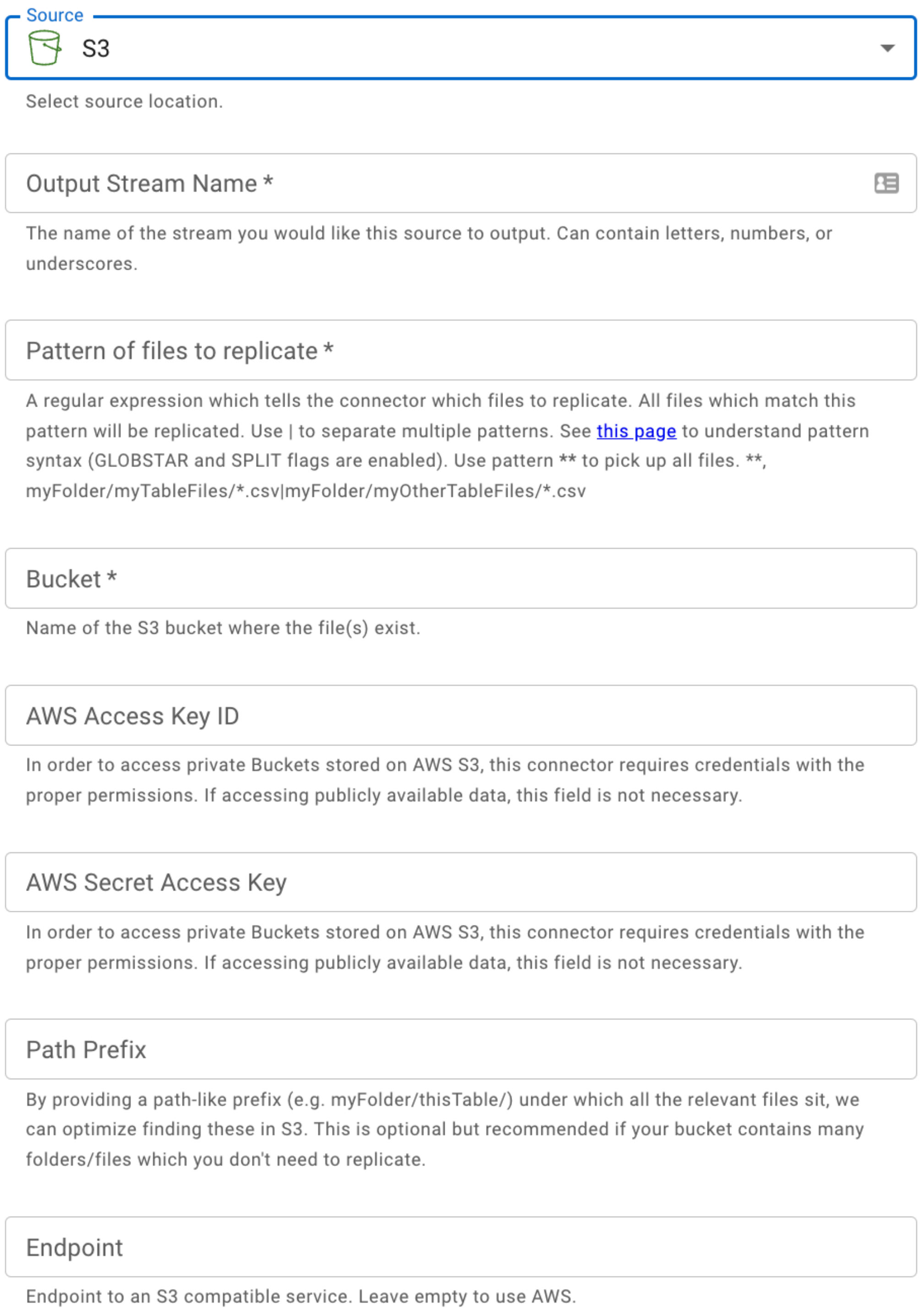


Once you select the **Source** option, you will be navigated to the **Source Job** creation page:



To name your dataset, click on the **Untitled Source** box, type the new name you want the dataset to have, and press Enter.

To set up your file source, choose the connector you want from the dropdown menu and follow the onscreen instructions to fill out the fields required to set up the connection to the source data.



Source
S3
Select source location.

Output Stream Name *
The name of the stream you would like this source to output. Can contain letters, numbers, or underscores.

Pattern of files to replicate *
A regular expression which tells the connector which files to replicate. All files which match this pattern will be replicated. Use | to separate multiple patterns. See this page to understand pattern syntax (GLOBSTAR and SPLIT flags are enabled). Use pattern ** to pick up all files. **, myFolder/myTableFiles/*.csv|myFolder/myOtherTableFiles/*.csv

Bucket *
Name of the S3 bucket where the file(s) exist.

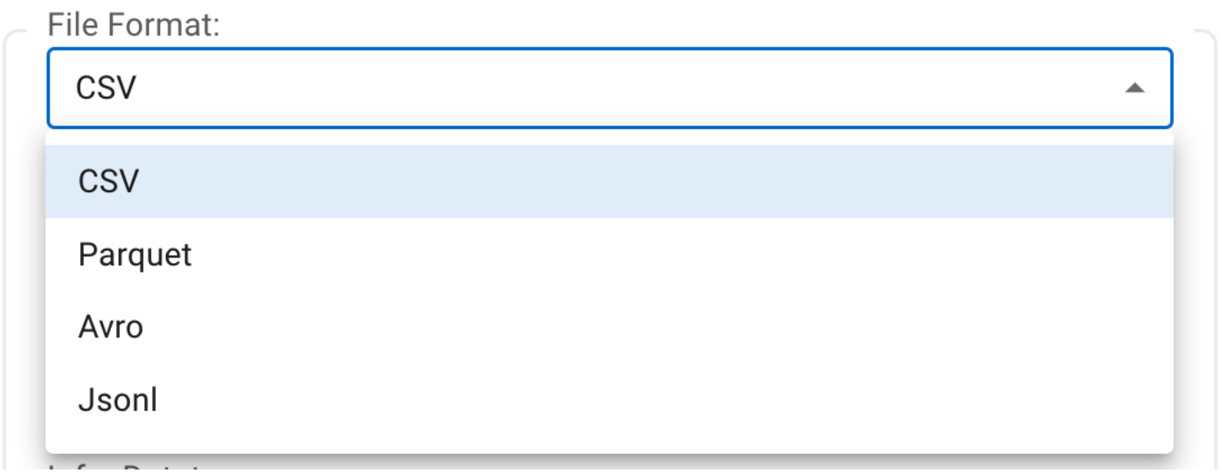
AWS Access Key ID
In order to access private Buckets stored on AWS S3, this connector requires credentials with the proper permissions. If accessing publicly available data, this field is not necessary.

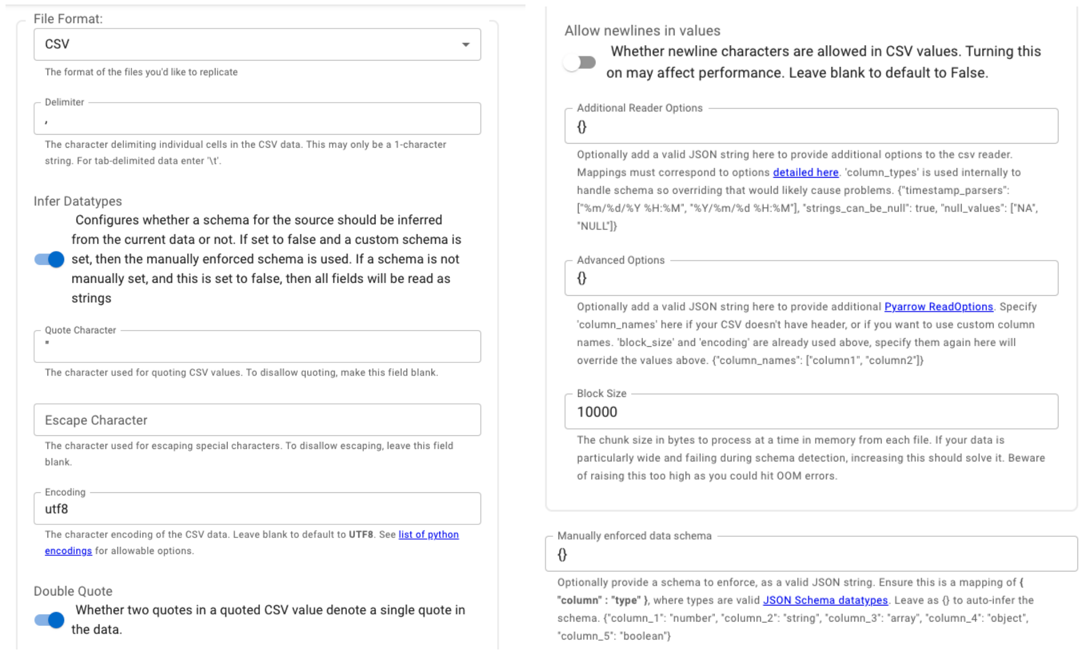
AWS Secret Access Key
In order to access private Buckets stored on AWS S3, this connector requires credentials with the proper permissions. If accessing publicly available data, this field is not necessary.

Path Prefix
By providing a path-like prefix (e.g. myFolder/thisTable/) under which all the relevant files sit, we can optimize finding these in S3. This is optional but recommended if your bucket contains many folders/files which you don't need to replicate.

Endpoint
Endpoint to an S3 compatible service. Leave empty to use AWS.

Once you enter the correct information for your S3 bucket, you have the option to adjust the file format:





File Format:

CSV

The format of the files you'd like to replicate

Delimiter

,

The character delimiting individual cells in the CSV data. This may only be a 1-character string. For tab-delimited data enter '\t'.

Infer Datatypes

Configures whether a schema for the source should be inferred from the current data or not. If set to false and a custom schema is set, then the manually enforced schema is used. If a schema is not manually set, and this is set to false, then all fields will be read as strings

Quote Character

"

The character used for quoting CSV values. To disallow quoting, make this field blank.

Escape Character

The character used for escaping special characters. To disallow escaping, leave this field blank.

Encoding

utf8

The character encoding of the CSV data. Leave blank to default to **UTF8**. See list of python encodings for allowable options.

Double Quote

Whether two quotes in a quoted CSV value denote a single quote in the data.

Allow newlines in values

Whether newline characters are allowed in CSV values. Turning this on may affect performance. Leave blank to default to False.

Additional Reader Options

{}

Optionally add a valid JSON string here to provide additional options to the csv reader. Mappings must correspond to options detailed here. 'column_types' is used internally to handle schema so overriding that would likely cause problems. {"timestamp_parsers": ["%m/%d/%Y %H:%M", "%Y/%m/%d %H:%M"], "strings_can_be_null": true, "null_values": ["NA", "NULL"]}

Advanced Options

{}

Optionally add a valid JSON string here to provide additional Pyarrow ReadOptions. Specify 'column_names' here if your CSV doesn't have header, or if you want to use custom column names. 'block_size' and 'encoding' are already used above, specify them again here will override the values above. {"column_names": ["column1", "column2"]}

Block Size

10000

The chunk size in bytes to process at a time in memory from each file. If your data is particularly wide and failing during schema detection, increasing this should solve it. Beware of raising this too high as you could hit OOM errors.

Manually enforced data schema

{}

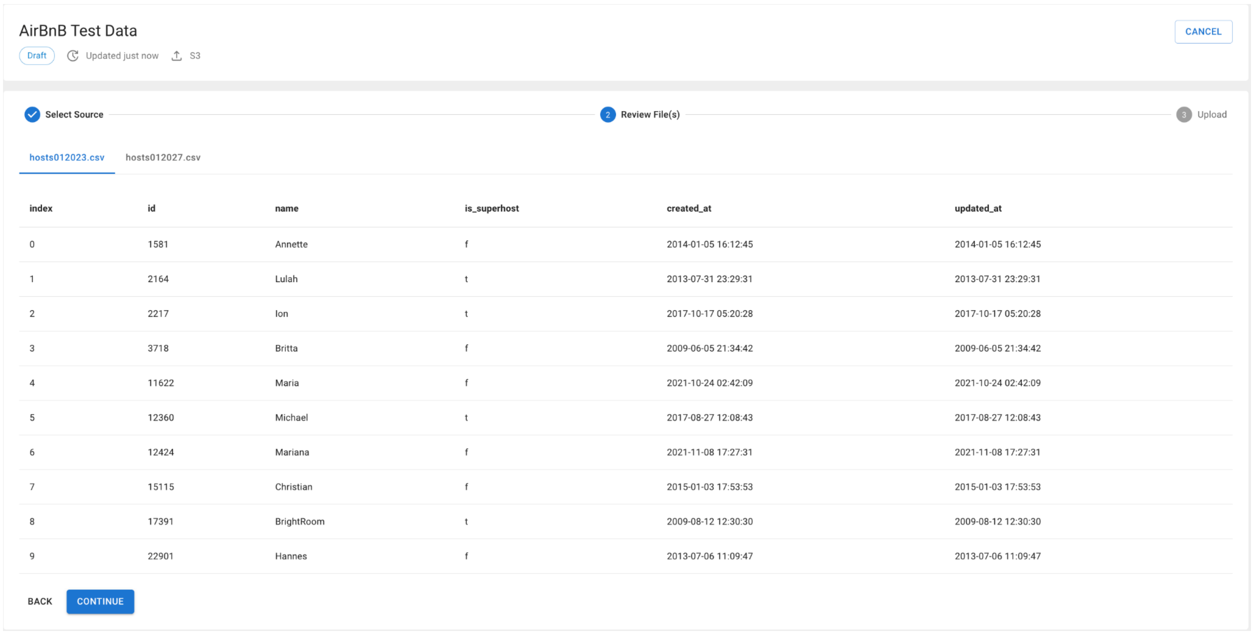
Optionally provide a schema to enforce, as a valid JSON string. Ensure this is a mapping of { **"column" : "type"** }, where types are valid JSON Schema datatypes. Leave as {} to auto-infer the schema. {"column_1": "number", "column_2": "string", "column_3": "array", "column_4": "object", "column_5": "boolean"}

Select the correct format for your data (CSV, Parquet, Avro, JSON) and adjust any parameters you would like. Next, click on the continue button at the bottom of the page to create your source.

Step 2: How do I review a sample of the source data before data upload?

WIP - ETA Q2

Once you have completed the **Select Source** step and established a connection to the source, Seek Connect will redirect you to the **Review files** screen.



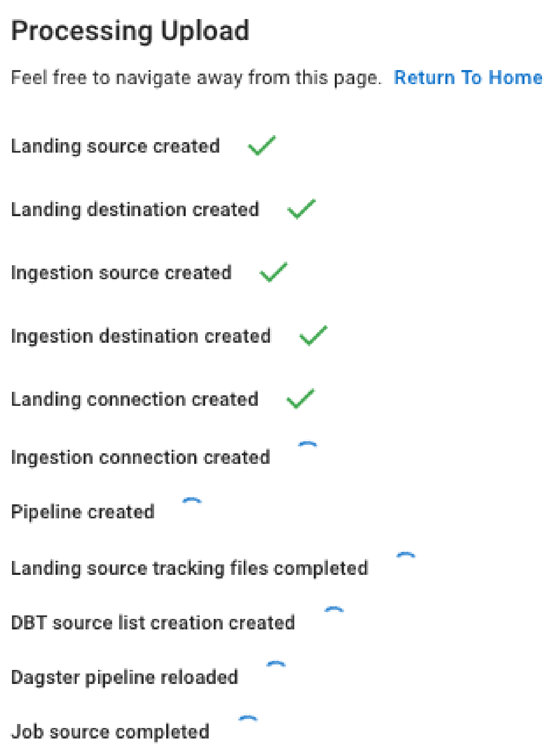
index	id	name	is_superhost	created_at	updated_at
0	1581	Annette	f	2014-01-05 16:12:45	2014-01-05 16:12:45
1	2164	Lulah	t	2013-07-31 23:29:31	2013-07-31 23:29:31
2	2217	Ion	t	2017-10-17 05:20:28	2017-10-17 05:20:28
3	3718	Britta	f	2009-06-05 21:34:42	2009-06-05 21:34:42
4	11622	Maria	f	2021-10-24 02:42:09	2021-10-24 02:42:09
5	12360	Michael	t	2017-08-27 12:08:43	2017-08-27 12:08:43
6	12424	Mariana	f	2021-11-08 17:27:31	2021-11-08 17:27:31
7	15115	Christian	f	2015-01-03 17:53:53	2015-01-03 17:53:53
8	17391	BrightRoom	t	2009-08-12 12:30:30	2009-08-12 12:30:30
9	22901	Hannes	f	2013-07-06 11:09:47	2013-07-06 11:09:47

On this screen, you can select a source file asset and preview a sample of the source data.

Click on the **Back** button to return to the **Create dataset** screen and modify your source setup.

Click on **Confirm** to proceed with the data upload process.

Seek Connect automatically creates the necessary landing and ingestion sources and destinations to generate the pipeline to your S3 bucket. The **Upload** page will confirm the Job source has been completed.



Processing Upload

Feel free to navigate away from this page. Return To Home

Landing source created ✓

Landing destination created ✓

Ingestion source created ✓

Ingestion destination created ✓

Landing connection created ✓

Ingestion connection created

Pipeline created

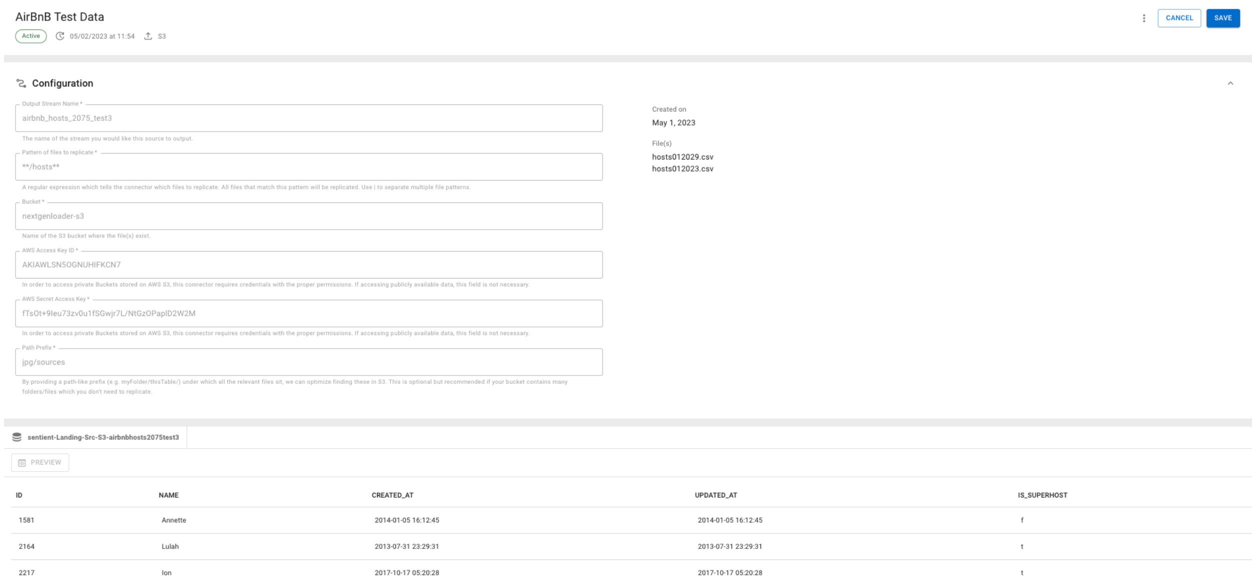
Landing source tracking files completed

DBT source list creation created

Dagster pipeline reloaded

Job source completed

After each field above is completed, you will be navigated to a preview page where you can save this Source Job to be used in Transformation and Publication Jobs.



AirBnB Test Data

Active 05/02/2023 at 11:54 S3

Configuration

Output Stream Name *	airbnb_hosts_2075_test3
Pattern of files to replicate *	**/hosts**
Bucket *	nextgenloader-s3
AWS Access Key ID *	AKIAWLSN5OGNUHIFKCN7
AWS Secret Access Key *	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
Path Prefix *	jpg/sources

The name of the stream you would like this source to output.

A regular expression which tells the connector which files to replicate. All files that match this pattern will be replicated. Use | to separate multiple file patterns.

Name of the S3 bucket where the file(s) exist.

In order to access private Buckets stored on AWS S3, this connector requires credentials with the proper permissions. If accessing publicly available data, this field is not necessary.

By providing a path-like prefix (e.g. myFolder/thisTable/) under which all the relevant files sit, we can optimize finding these in S3. This is optional but recommended if your bucket contains many folders/files which you don't need to replicate.

Created on
May 1, 2023

File(s)
hosts012029.csv
hosts012023.csv

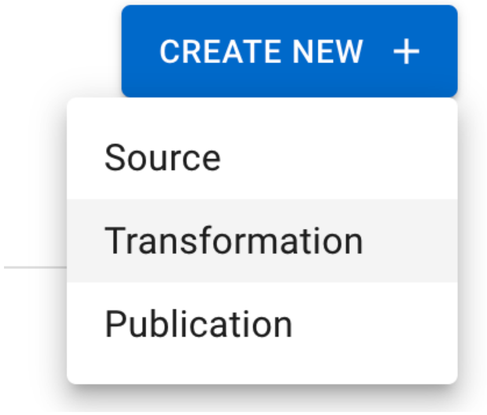
sentient-Landing-Src-S3-airbnbhosts2075test3

ID	NAME	CREATED_AT	UPDATED_AT	IS_SUPERHOST
1581	Annette	2014-01-05 16:12:45	2014-01-05 16:12:45	f
2164	Lulah	2013-07-31 23:29:31	2013-07-31 23:29:31	t
2217	Ion	2017-10-17 05:20:28	2017-10-17 05:20:28	t

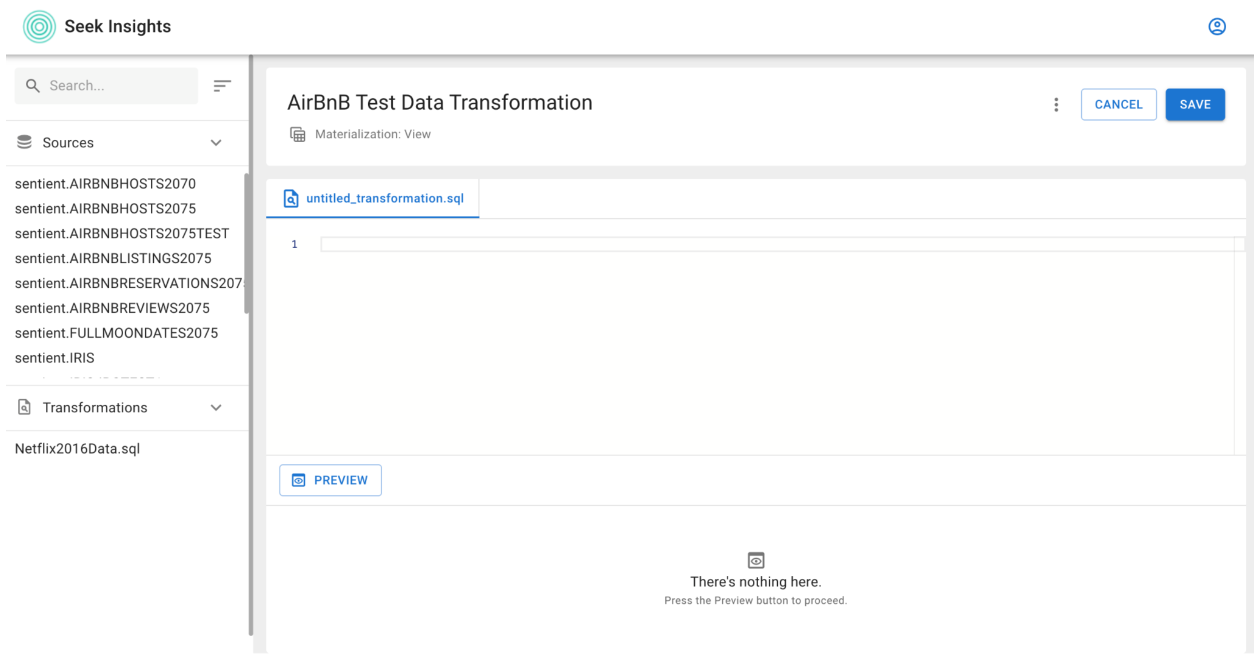
Once the connection is Active, you can click **Save** to return to the home page to create a new **Data Transformation Job.**

Step 3: How to perform a Data Transformation?

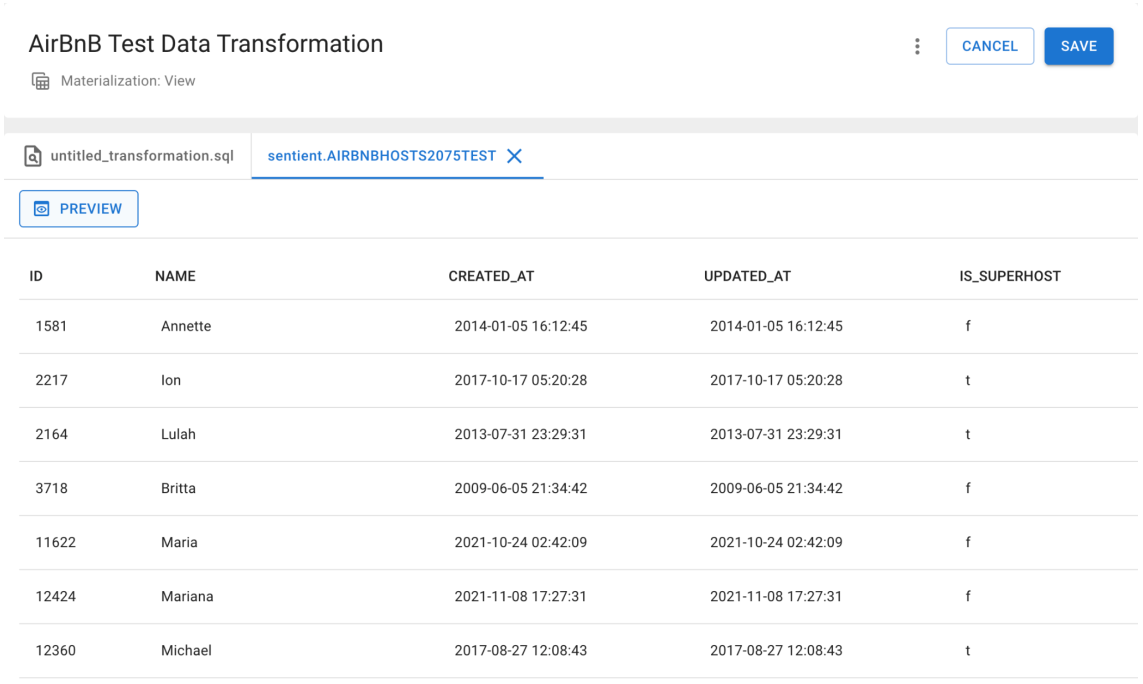
To begin your Data Transformation Job, navigate to the **Create New +** button and select the Transformation option from the drop-down menu.



The next page will allow you to name your Data Transformation Job and select the source(s) you would like to perform the transformation on.

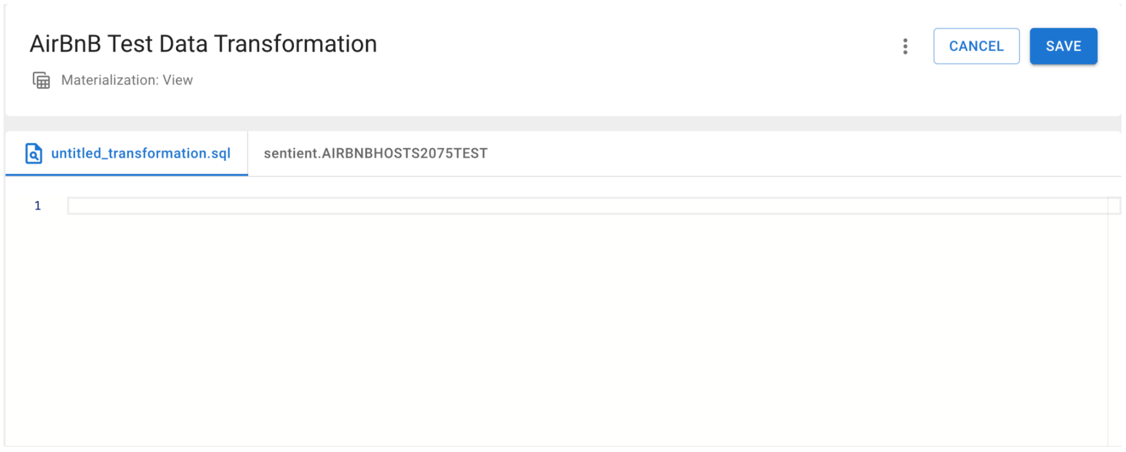

Seek Insights
Search...
Sources
sentient.AIRBNBHOSTS2070
sentient.AIRBNBHOSTS2075
sentient.AIRBNBHOSTS2075TEST
sentient.AIRBNBLISTINGS2075
sentient.AIRBNBRESERVATIONS207
sentient.AIRBNBREVIEWS2075
sentient.FULLMOONDATES2075
sentient.IRIS
Transformations
Netflix2016Data.sql
AirBnB Test Data Transformation
Materialization: View
CANCEL
SAVE
untitled_transformation.sql
PREVIEW
There's nothing here.
Press the Preview button to proceed.

Select your source from the left-hand panel. You can click the **preview** button to see a sample of the data to help inform your SQL transformation.

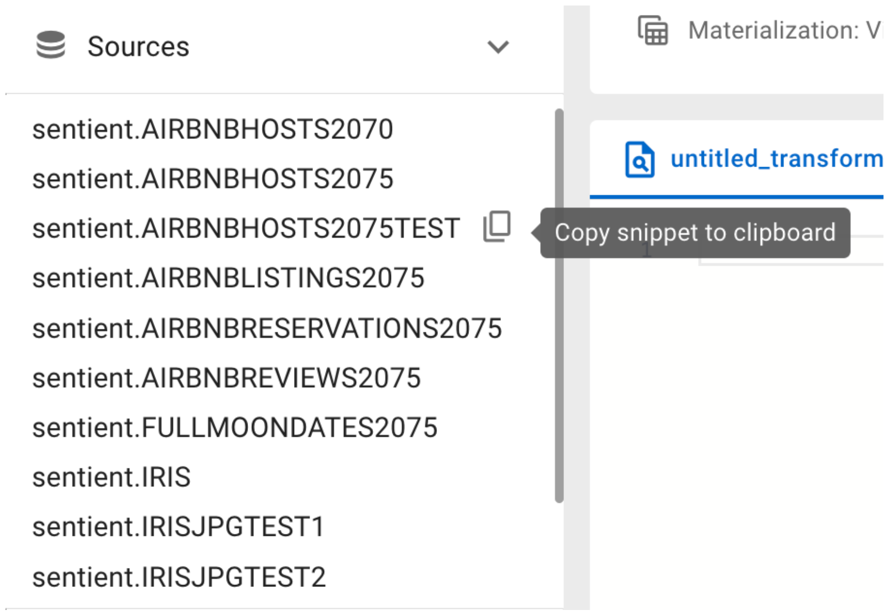

AirBnB Test Data Transformation
Materialization: View
CANCEL
SAVE
untitled_transformation.sql
sentient.AIRBNBHOSTS2075TEST
PREVIEW

ID	NAME	CREATED_AT	UPDATED_AT	IS_SUPERHOST
1581	Annette	2014-01-05 16:12:45	2014-01-05 16:12:45	f
2217	Ion	2017-10-17 05:20:28	2017-10-17 05:20:28	t
2164	Lulah	2013-07-31 23:29:31	2013-07-31 23:29:31	t
3718	Britta	2009-06-05 21:34:42	2009-06-05 21:34:42	f
11622	Maria	2021-10-24 02:42:09	2021-10-24 02:42:09	f
12424	Mariana	2021-11-08 17:27:31	2021-11-08 17:27:31	f
12360	Michael	2017-08-27 12:08:43	2017-08-27 12:08:43	t

To transform this dataset before publishing, you need to navigate to the untitled_transformation.sql tab. Here, you can write SQL code to select or subset the data you want.

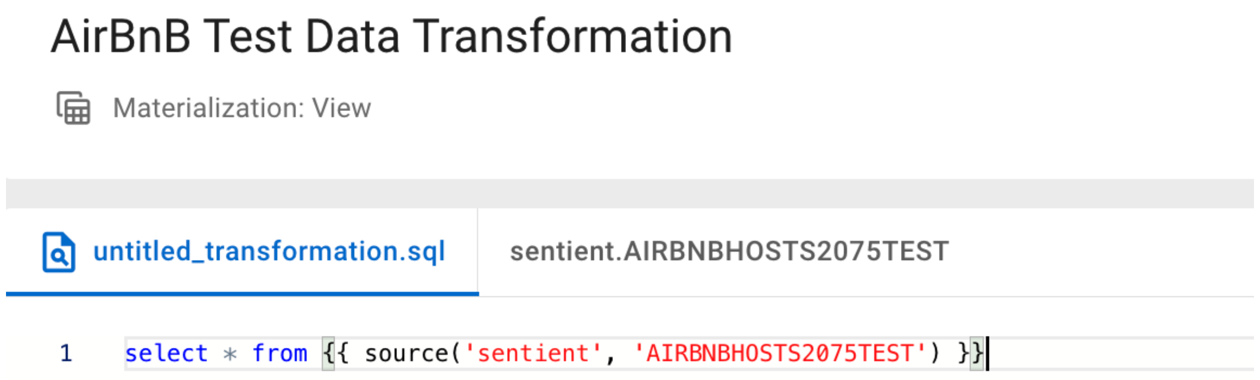

AirBnB Test Data Transformation
Materialization: View
CANCEL
SAVE
untitled_transformation.sql
sentient.AIRBNBHOSTS2075TEST

In order to help with your SQL query, we have included a copy feature for each data source you have. Simply, hover your cursor over your desired source and click the copy button:



sentient.AIRBNBHOSTS2070
sentient.AIRBNBHOSTS2075
sentient.AIRBNBHOSTS2075TEST
sentient.AIRBNBLISTINGS2075
sentient.AIRBNBRESERVATIONS2075
sentient.AIRBNBREVIEWS2075
sentient.FULLMOONDATES2075
sentient.IRIS
sentient.IRISJPGTEST1
sentient.IRISJPGTEST2

Paste this snippet into the untitled_transformations.sql file to begin writing your transformation.

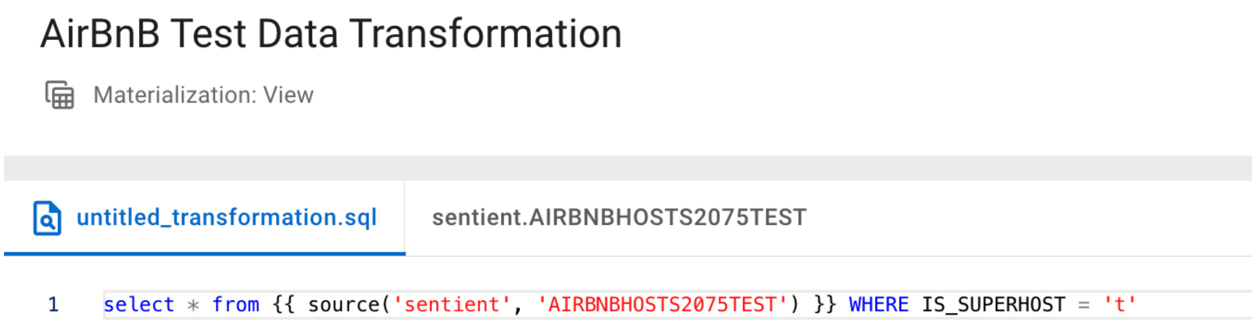

AirBnB Test Data Transformation
Materialization: View
untitled_transformation.sql
sentient.AIRBNBHOSTS2075TEST

```
select * from {{ source('sentient', 'AIRBNBHOSTS2075TEST') }}
```

Now, you can add any additional SQL code to transform your dataset.

For this example, we are going to apply a simple transformation to subset the table on the **IS_SUPERHOST** column so that we only have Superhosts in the dataset.

To subset this table to only include Superhosts, we need to add **WHERE IS_SUPERHOST = 't'** to the code snippet



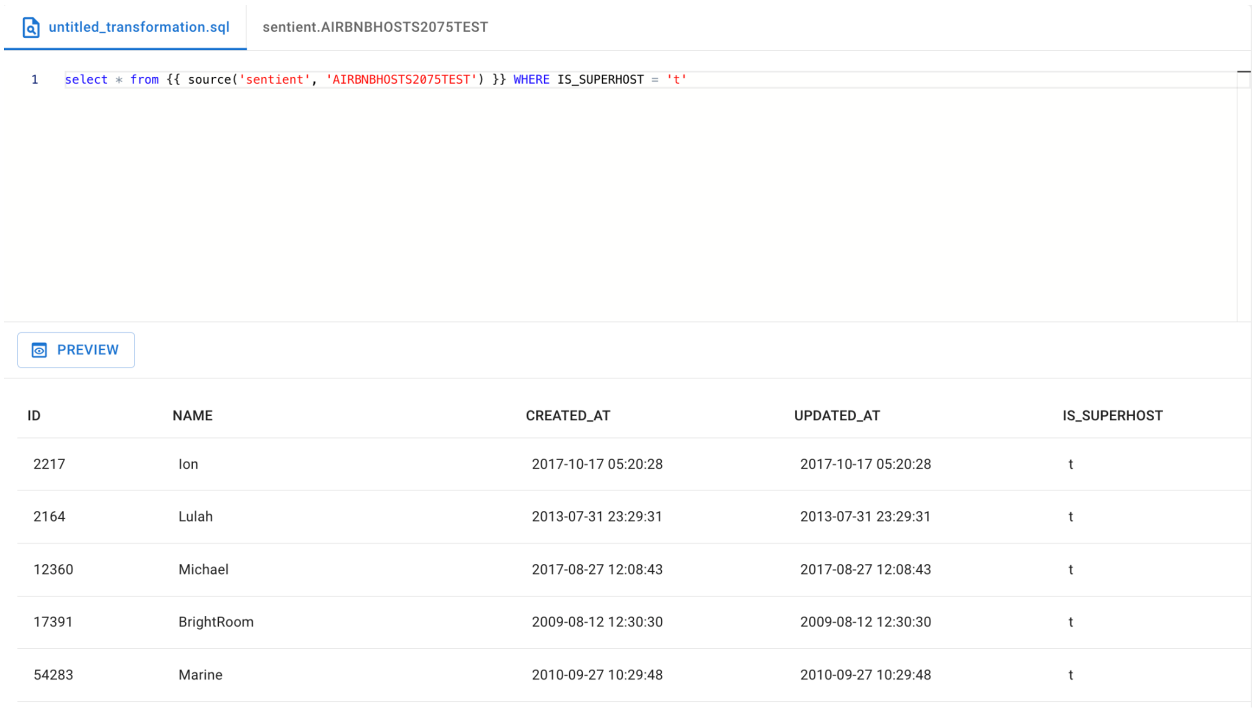
AirBnB Test Data Transformation

Materialization: View

untitled_transformation.sql | sentient.AIRBNBHOSTS2075TEST

```
select * from {{ source('sentient', 'AIRBNBHOSTS2075TEST') }} WHERE IS_SUPERHOST = 't'
```

You can preview the results of this query by selecting the **Preview** button below



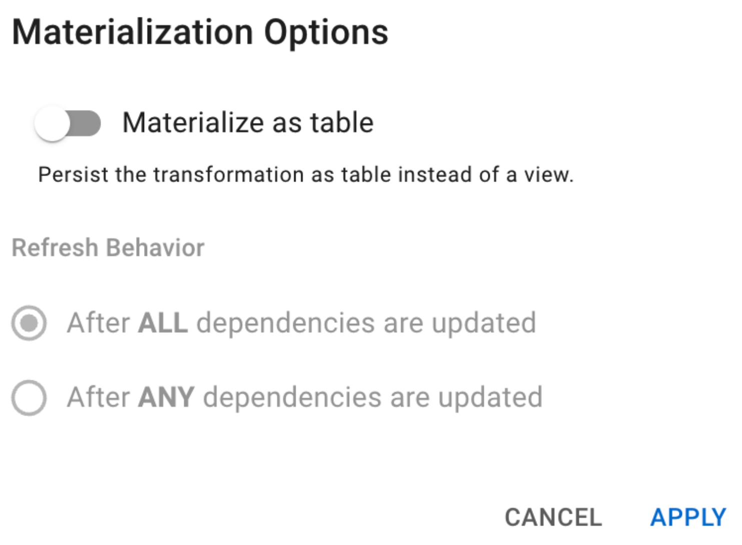
untitled_transformation.sql | sentient.AIRBNBHOSTS2075TEST

```
select * from {{ source('sentient', 'AIRBNBHOSTS2075TEST') }} WHERE IS_SUPERHOST = 't'
```

ID	NAME	CREATED_AT	UPDATED_AT	IS_SUPERHOST
2217	Ion	2017-10-17 05:20:28	2017-10-17 05:20:28	t
2164	Lulah	2013-07-31 23:29:31	2013-07-31 23:29:31	t
12360	Michael	2017-08-27 12:08:43	2017-08-27 12:08:43	t
17391	BrightRoom	2009-08-12 12:30:30	2009-08-12 12:30:30	t
54283	Marine	2010-09-27 10:29:48	2010-09-27 10:29:48	t

We can see from the above preview that the only rows included are Superhosts, so we are seeing the expected result from our above SQL query.

Before saving this Data Transformation Job, you can edit the Title, Rename the File, or choose to materialize this dataset as a table instead of a view:


Materialization Options

Materialize as table

Persist the transformation as table instead of a view.

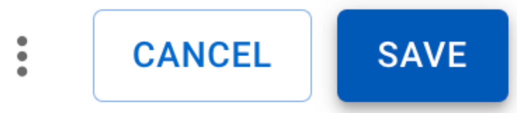
Refresh Behavior

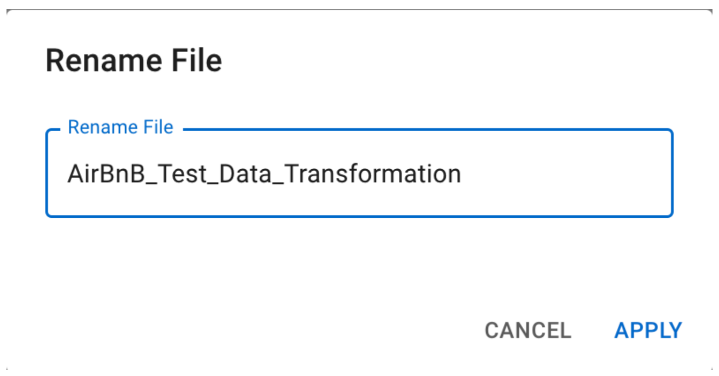
(●) After **ALL** dependencies are updated

( ) After **ANY** dependencies are updated

Materializing the data as a table will refresh/update the data regularly, based on upstream source data updates. Materializing as a view will generate results each time it is queried by a user. Each method brings compute and performance implications to consider. Views are usually the more cost-effective choice, but not always.

Once you have finished making edits to the Data Transformation Job, click **Save** in the upper right corner. This will prompt you to name your transformation. Make sure to give this a unique name as this will be used in the **Publication Job**.



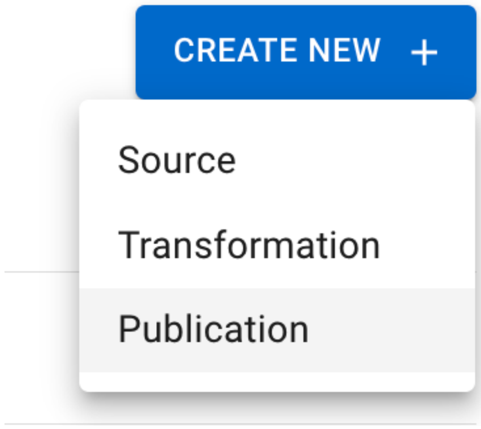

Rename File

Rename File: AirBnB_Test_Data_Transformation

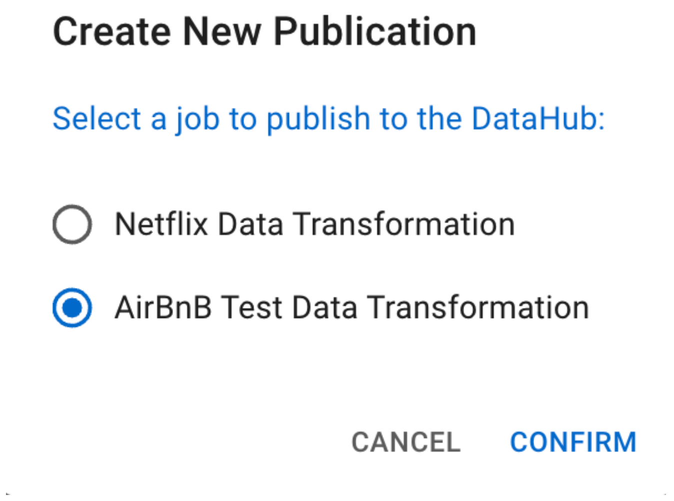
After you have saved your Transformation Job, navigate back to the home page to Publish this dataset to your Data Hub.

Step 4: How do I Publish my new dataset to my Data Hub?

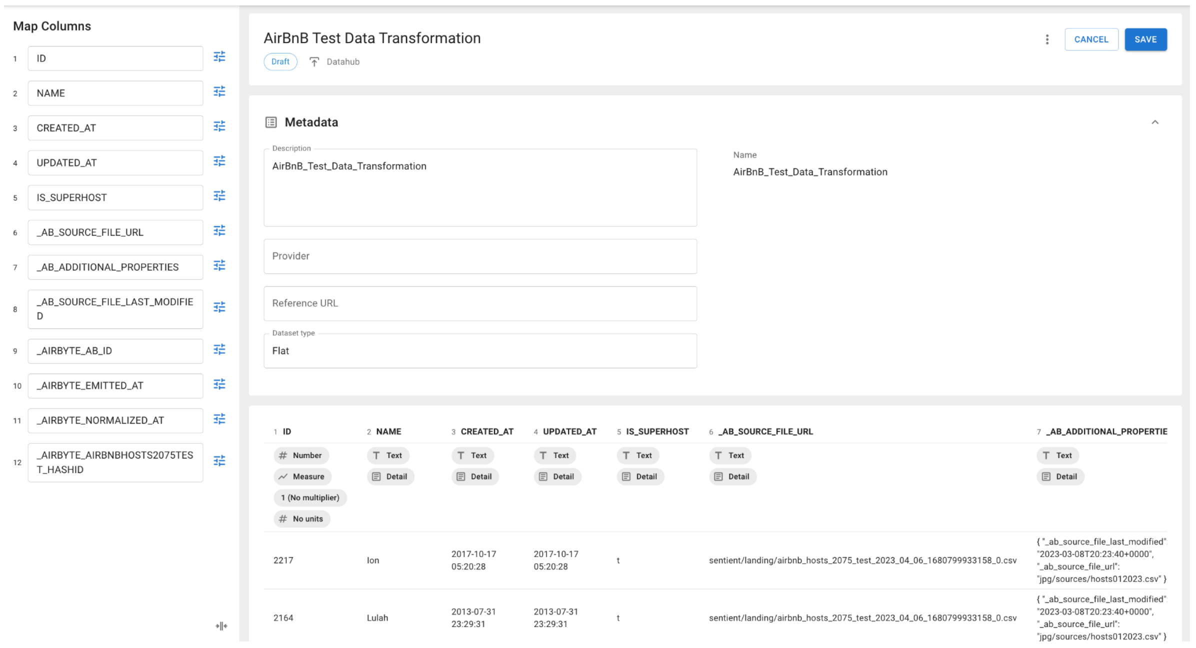
To begin your **Publication Job**, navigate to the **Create New +** button and select the Publication option from the drop-down menu.



A pop-up window will appear asking you to select which Data Transformation Job you would like to Publish to your Data hub. Select your desired transformation and click **Confirm** to continue.


Create New Publication
Select a job to publish to the DataHub:
Netflix Data Transformation
AirBnB Test Data Transformation
CANCEL
CONFIRM

Once you confirm which data transformation you would like to publish, you will be navigated to the next page to map your data


Map Columns

1 ID
2 NAME
3 CREATED_AT
4 UPDATED_AT
5 IS_SUPERHOST
6 _AB_SOURCE_FILE_URL
7 _AB_ADDITIONAL_PROPERTIES
8 _AB_SOURCE_FILE_LAST_MODIFIED
9 _AIRBYTE_AB_ID
10 _AIRBYTE_EMITTED_AT
11 _AIRBYTE_NORMALIZED_AT
12 _AIRBYTE_AIRBNBHOSTS2075TEST_HASHID

AirBnB Test Data Transformation

Draft · Datahub

CANCEL · SAVE

Metadata

Description: AirBnB_Test_Data_Transformation

Name: AirBnB_Test_Data_Transformation

Provider

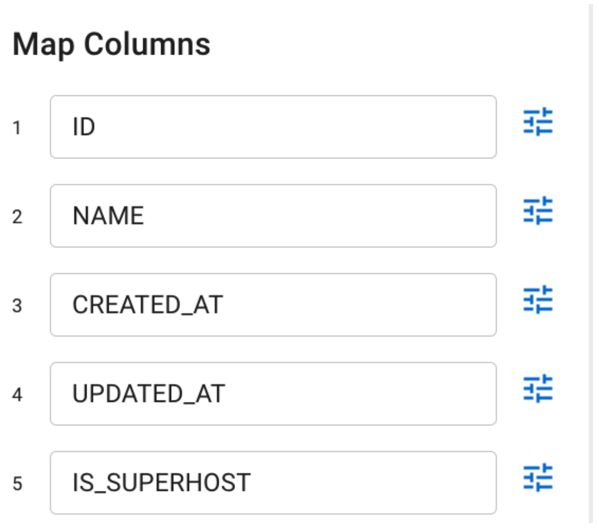
Reference URL

Dataset type: Flat

1 ID	2 NAME	3 CREATED_AT	4 UPDATED_AT	5 IS_SUPERHOST	6 _AB_SOURCE_FILE_URL	7 _AB_ADDITIONAL_PROPERTIE
Number Measure 1 (No multiplier) No units	Text Detail	Text Detail	Text Detail	Text Detail	Text Detail	Text Detail
2217	Ion	2017-10-17 05:20:28	2017-10-17 05:20:28	t	sentient/landing/airbnb_hosts_2075_test_2023_04_06_1680799933158_0.csv	{ "_ab_source_file_last_modified": "2023-03-08T20:23:40+0000", "_ab_source_file_url": "jpg/sources/hosts012023.csv" }
2164	Lulah	2013-07-31 23:29:31	2013-07-31 23:29:31	t	sentient/landing/airbnb_hosts_2075_test_2023_04_06_1680799933158_0.csv	{ "_ab_source_file_last_modified": "2023-03-08T20:23:40+0000", "_ab_source_file_url": "jpg/sources/hosts012023.csv" }

Step 5: How do I map my data?

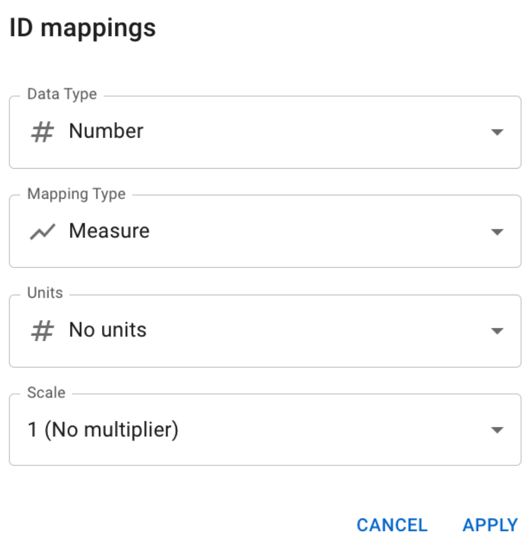
On the left-hand side of the page, you can view and edit the names of your columns. You can also edit the mapping parameters by clicking on the blue icon next to each column. Easily adjust your Data Type, Mapping Type, Units, and Scale.


Map Columns

1 ID
2 NAME
3 CREATED_AT
4 UPDATED_AT
5 IS_SUPERHOST

To rename a column, find the column you want to rename in the left panel, click on its name, type the new name you want the column to have, and press Enter.

To assign mapping for a column, click on the blue filter icon next to its name, and select the appropriate formatting, mapping type, and additional refinement options. A pop-up window will open.


ID mappings

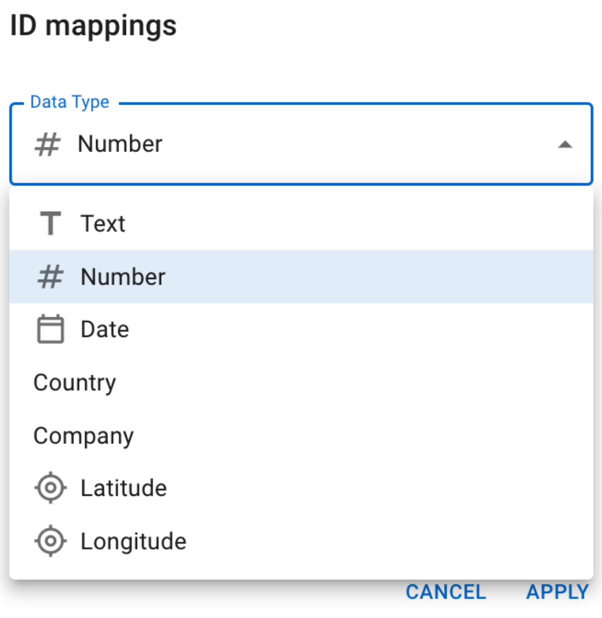
Data Type: Number

Mapping Type: Measure

Units: No units

Scale: 1 (No multiplier)

First, select the appropriate **Data Type** for that column:

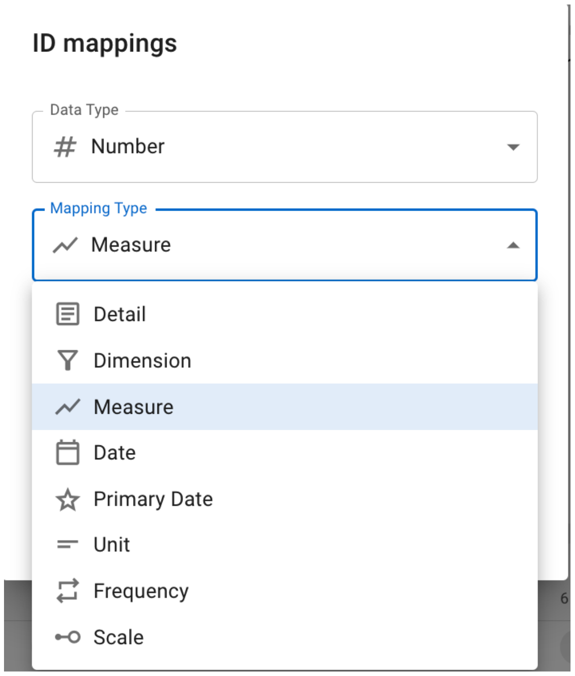

ID mappings

Data Type: Number

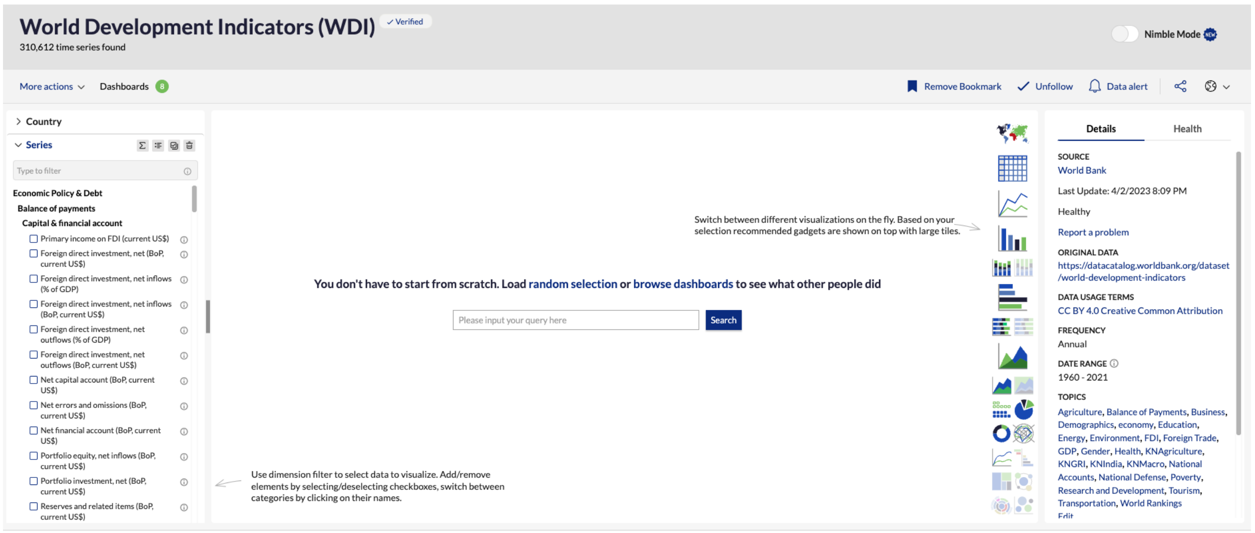
- **Text** format is used for a textual type of data.
- **Number** format is used for numeric types of data.

- **Date** field is the calendar dimension of the resulting dataset. It stores the dates of various data points and is used to build time series from your data. Our system supports the following statistical date formats for different frequencies:
  - Years: 2009, 2010, 2011
  - Half years: 2009H1, 2013H2
  - Quarters: 2009Q1, 2010Q3, 2012Q4
  - Months: 2009M2, 2011M7
- **Country** format is used for columns that contain country information, which could be represented by country names or country ISO codes
- **Company** format is used for columns that contain company information.
- **Latitude** and **Longitude** fields can be used to read the latitude or longitude attribute information for location-based data.

Next, select the appropriate **Mapping Type** for that column:



- **Dimension** is used to specify a data dimension for the resulting dataset. A data dimension is also represented in the form of a dropdown menu option within the dataset for the selection of data categories to visualize in the DataViewer. Only data rows in your source data that are complete—no data gaps—may be set as a Dimension.
  - In order to visualize data in Knoema's Dataset Viewer, you will need to choose at least one column to be a Dimension. This will be the column where you select your data. For example, the WDI dataset below from World Bank has two Dimensions (Country and Series)



World Development Indicators (WDI) ✓ Verified
310,612 time series found

Nimble Mode NEW

More actions · Dashboards 8 · Remove Bookmark · Unfollow · Data alert

Country

Series

Type to filter

Economic Policy & Debt
Balance of payments
Capital & financial account
- Primary income on FDI (current US$)
- Foreign direct investment, net (BoP, current US$)
- Foreign direct investment, net inflows (% of GDP)
- Foreign direct investment, net inflows (BoP, current US$)
- Foreign direct investment, net outflows (% of GDP)
- Foreign direct investment, net outflows (BoP, current US$)
- Net capital account (BoP, current US$)
- Net errors and omissions (BoP, current US$)
- Net financial account (BoP, current US$)
- Portfolio equity, net inflows (BoP, current US$)
- Portfolio investment, net (BoP, current US$)
- Reserves and related items (BoP, current US$)

Switch between different visualizations on the fly. Based on your selection recommended gadgets are shown on top with large tiles.

You don't have to start from scratch. Load random selection or browse dashboards to see what other people did

Please input your query here · Search

Use dimension filter to select data to visualize. Add/remove elements by selecting/deselecting checkboxes, switch between categories by clicking on their names.

Details · Health

SOURCE
World Bank

Last Update: 4/2/2023 8:09 PM

Healthy

Report a problem

ORIGINAL DATA
https://datacatalog.worldbank.org/dataset/world-development-indicators

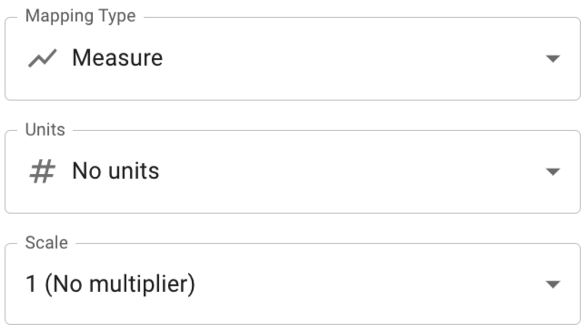
DATA USAGE TERMS
CC BY 4.0 Creative Common Attribution

FREQUENCY
Annual

DATE RANGE
1960 - 2021

TOPICS
Agriculture, Balance of Payments, Business, Demographics, economy, Education, Energy, Environment, FDI, Foreign Trade, GDP, Gender, Health, KNAgriculture, KNGRI, KNIndia, KNMacro, National Accounts, National Defense, Poverty, Research and Development, Tourism, Transportation, World Rankings
Edit

- **Measure** can be used for numerical data that should be calculated or aggregated. Any number, currency, or numerical value can be used as a Measure. For Measure columns, you can further refine the mapping by choosing its:
    - **Unit** is used to specify the units of measurement.
    - **Scale** is used to specify the scale of measurements.
    - These can be specified for a Measure in two additional drop-downs:



Mapping Type: Measure

Units: No units

Scale: 1 (No multiplier)

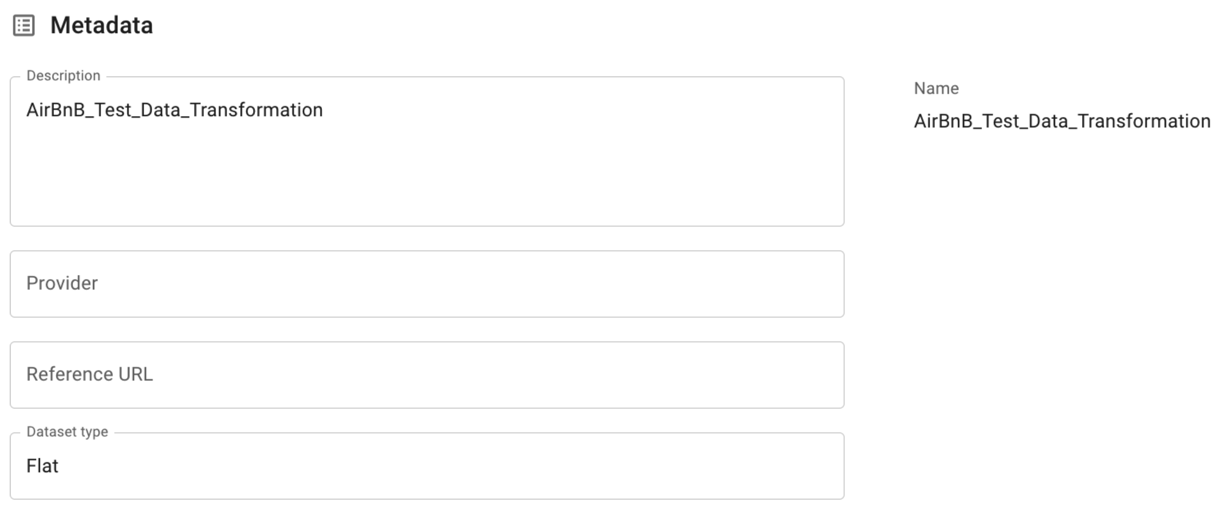
- **Date** field is the calendar dimension of the resulting dataset. It stores the dates of various data points and is used to build time series from your data. Our system supports the following statistical date formats for different frequencies:
    - Years: 2009, 2010, 2011
    - Half years: 2009H1, 2013H2
    - Quarters: 2009Q1, 2010Q3, 2012Q4
    - Months: 2009M2, 2011M7
    - Days: MM/DD/YYYY, DD.MM.YYYY

- **Primary Date** is a Knoema platform concept that signifies the column is for x-axis charting in standard Knoema time series visualizations and calculations.
- **Detail** is any additional information attached to a record. It is typically used for information that would work with the grid visualization tool. For Detail columns, you can further refine the mapping by choosing:
    - **Unit** is used to specify the units of measurement.
    - **Frequency** is used to specify the data frequency. The following frequencies are supported:
        - A - annual
        - H - semi-annual
        - Q - quarterly
        - M - monthly
        - D - daily
    - **Scale** is used to specify the scale of measurements.

Click the **Apply** button in the dropdown menu to apply your changes.

Step 6: How do I edit the Metadata?

Next, you can edit the Metadata associated with your dataset. Add Descriptions, Provider details, Reference URLs, and Dataset type fields here.


Metadata

Description: AirBnB_Test_Data_Transformation

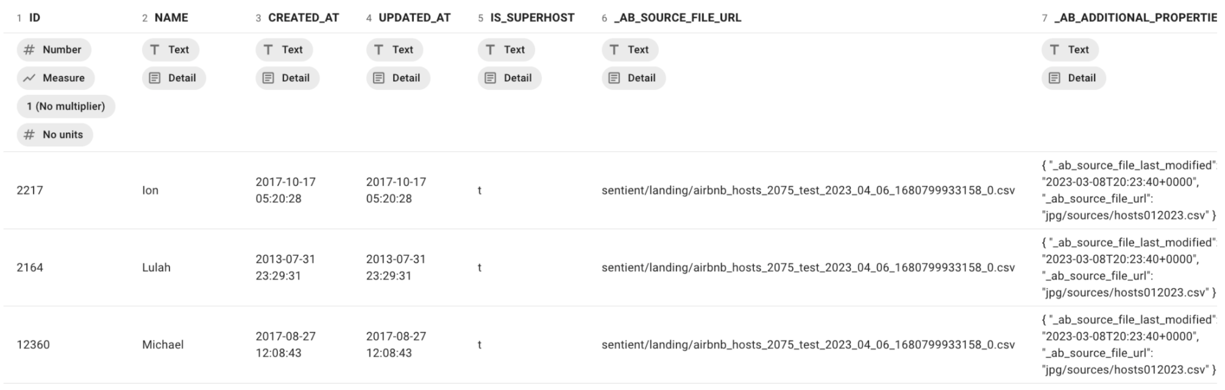
Name: AirBnB_Test_Data_Transformation

Provider

Reference URL

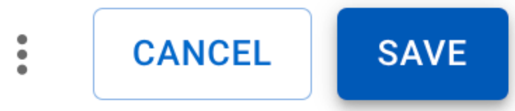
Dataset type: Flat

Scroll down to see the table with all of the mapping information displayed. If you notice any issues with the mappings you can click on them and adjust with ease.

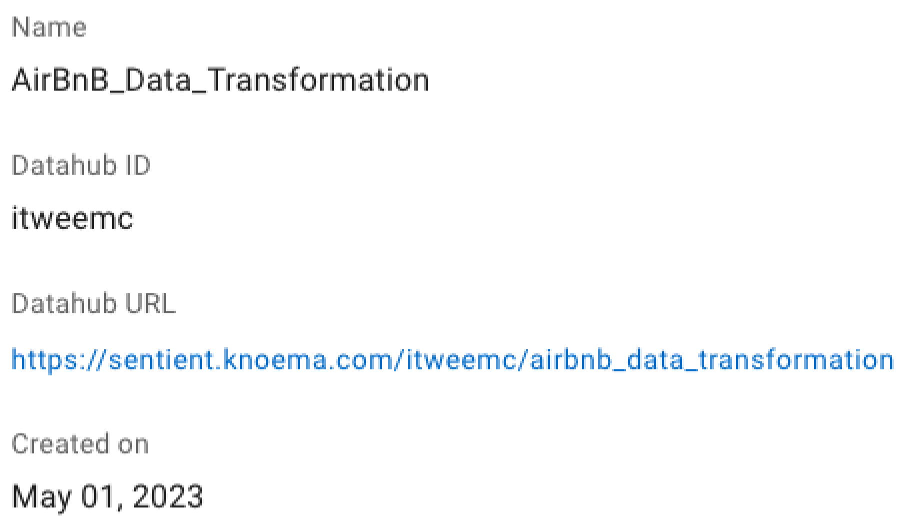


1 ID	2 NAME	3 CREATED_AT	4 UPDATED_AT	5 IS_SUPERHOST	6 _AB_SOURCE_FILE_URL	7 _AB_ADDITIONAL_PROPERTIE
Number Measure 1 (No multiplier) No units	Text Detail	Text Detail	Text Detail	Text Detail	Text Detail	Text Detail
2217	Ion	2017-10-17 05:20:28	2017-10-17 05:20:28	t	sentient/landing/airbnb_hosts_2075_test_2023_04_06_1680799933158_0.csv	{ "_ab_source_file_last_modified": "2023-03-08T20:23:40+0000", "_ab_source_file_url": "jpg/sources/hosts012023.csv" }
2164	Lulah	2013-07-31 23:29:31	2013-07-31 23:29:31	t	sentient/landing/airbnb_hosts_2075_test_2023_04_06_1680799933158_0.csv	{ "_ab_source_file_last_modified": "2023-03-08T20:23:40+0000", "_ab_source_file_url": "jpg/sources/hosts012023.csv" }
12360	Michael	2017-08-27 12:08:43	2017-08-27 12:08:43	t	sentient/landing/airbnb_hosts_2075_test_2023_04_06_1680799933158_0.csv	{ "_ab_source_file_last_modified": "2023-03-08T20:23:40+0000", "_ab_source_file_url": "jpg/sources/hosts012023.csv" }

Once you finish editing the mapping fields and metadata, click on the Save button in the upper right corner to upload this dataset to your Data Hub.



After saving this **Publication Job** the page will reload with a new link to this dataset in your Data Hub:



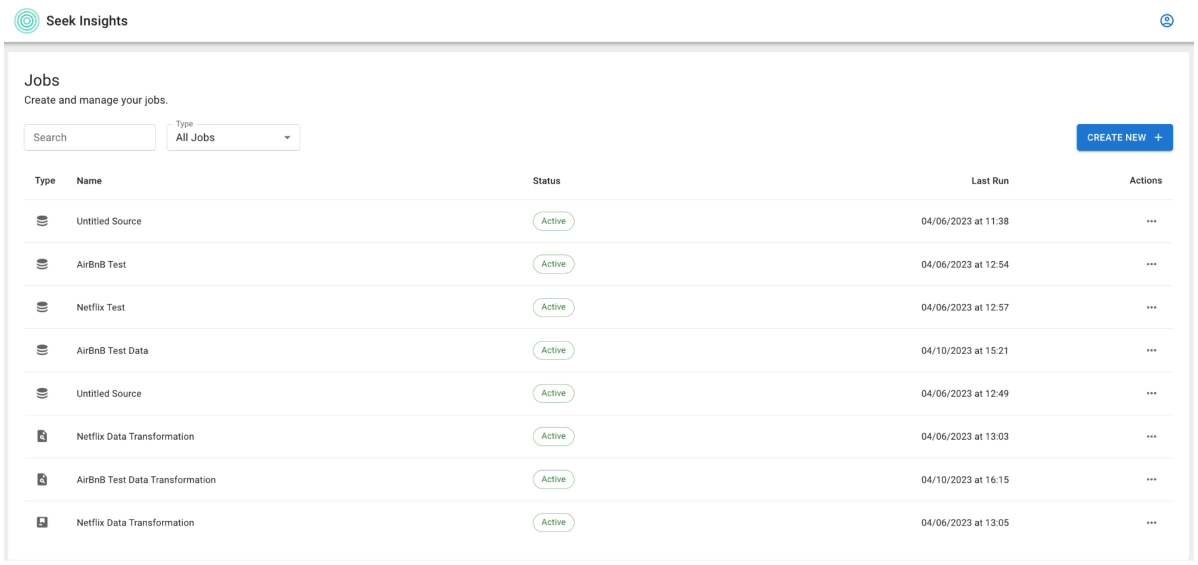
Name
AirBnB_Data_Transformation

Datahub ID
itweemc

Datahub URL
https://sentient.knoema.com/itweemc/airbnb_data_transformation

Created on
May 01, 2023

The Source Job, Transformation Job, and Publication Job will all remain active, creating a live connection to your raw data source. These Jobs will remain on your Jobs homepage, where you can easily reference them and make any edits in the future.



Jobs

Create and manage your jobs.

Type	Name	Status	Last Run	Actions
	Untitled Source	Active	04/06/2023 at 11:38	···
	AirBnB Test	Active	04/06/2023 at 12:54	···
	Netflix Test	Active	04/06/2023 at 12:57	···
	AirBnB Test Data	Active	04/10/2023 at 15:21	···
	Untitled Source	Active	04/06/2023 at 12:49	···
	Netflix Data Transformation	Active	04/06/2023 at 13:03	···
	AirBnB Test Data Transformation	Active	04/10/2023 at 16:15	···
	Netflix Data Transformation	Active	04/06/2023 at 13:05	···

Seek Connect Integration Documentation

Connector Catalog - Databases

MySQL

Features

Feature	Supported	Notes
Full Refresh Sync	Yes	
Incremental - Append Sync	Yes	
Replicate Incremental Deletes	Yes	
CDC	Yes	
SSL Support	Yes	
SSH Tunnel Connection	Yes	
Namespaces	Yes	Enabled by default
Arrays	Yes	Byte arrays are not supported yet

The MySQL source does not alter the schema present in your database. Depending on the destination connected to this source, however, the schema may be altered. See the destination's documentation for more details.

Troubleshooting

There may be problems with mapping values in MySQL's datetime field to other relational data stores. MySQL permits zero values for date/time instead of NULL which may not be accepted by other data stores. To work around this problem, you can pass the following key value pair in the JDBC connector of the source setting `zerodatetimebehavior=Converttonull`.

Some users reported that they could not connect to Amazon RDS MySQL or MariaDB. This can be diagnosed with the error message: `Cannot create a PoolableConnectionFactory`. To solve this issue add `enabledTLSProtocols=TLSv1.2` in the JDBC parameters.

Another error that users have reported when trying to connect to Amazon RDS MySQL is `Error: HikariPool-1 - Connection is not available, request timed out after 30001ms.`. Many times this is can be due to the VPC not allowing public traffic, however, we recommend going through this AWS troubleshooting checklist to the correct permissions/settings have been granted to allow connection to your database.

Getting Started (Airbyte Cloud)

On Airbyte Cloud, only TLS connections to your MySQL instance are supported. Other than that, you can proceed with the open-source instructions below.

Getting Started (Airbyte Open-Source)

Requirements

1. MySQL Server `8.0`, `5.7`, or `5.6`.
2. Create a dedicated read-only Airbyte user with access to all tables needed for replication.

1. Make sure your database is accessible from the machine running Airbyte

This is dependent on your networking setup. The easiest way to verify if Airbyte is able to connect to your MySQL instance is via the check connection tool in the UI.

2. Create a dedicated read-only user with access to the relevant tables (Recommended but optional)

This step is optional but highly recommended to allow for better permission control and auditing. Alternatively, you can use Airbyte with an existing user in your database.

To create a dedicated database user, run the following commands against your database:

```
Unset

CREATE USER 'airbyte'@'%' IDENTIFIED BY
'your_password_here';
```

The right set of permissions differ between the `STANDARD` and `CDC` replication method. For `STANDARD` replication method, only `SELECT` permission is required.

```
Unset

GRANT SELECT ON <database name>.* TO 'airbyte'@'%';
```

For `CDC` replication method, `SELECT, RELOAD, SHOW DATABASES, REPLICATION SLAVE, REPLICATION CLIENT` permissions are required.

```
Unset

GRANT SELECT, RELOAD, SHOW DATABASES, REPLICATION SLAVE,
REPLICATION CLIENT ON *.* TO 'airbyte'@'%';
```

Your database user should now be ready for use with Airbyte.

3. Set up CDC

For `STANDARD` replication method this is not applicable. If you select the `CDC` replication method then only this is required. Please read the section on CDC below for more information.

4. That's it!

Your database user should now be ready for use with Airbyte.

Change Data Capture (CDC)

- If you need a record of deletions and can accept the limitations posted below, you should be able to use CDC for MySQL.
- If your data set is small, and you just want snapshot of your table in the destination, consider using Full Refresh replication for your table instead of CDC.
- If the limitations prevent you from using CDC and your goal is to maintain a snapshot of your table in the destination, consider using non-CDC incremental and occasionally reset the data and re-sync.
- If your table has a primary key but doesn't have a reasonable cursor field for incremental syncing (i.e. `updated_at`), CDC allows you to sync your table incrementally.

CDC Limitations

- Make sure to read our CDC docs to see limitations that impact all databases using CDC replication.
- Our CDC implementation uses at least once delivery for all change records.

1. Enable binary logging

You must enable binary logging for MySQL replication. The binary logs record transaction updates for replication tools to propagate changes. You can configure your MySQL server configuration file with the following properties, which are described in below:

```
Unset


server-id                  = 223344
log_bin                    = mysql-bin
binlog_format              = ROW
binlog_row_image           = FULL
binlog_expire_log_seconds  = 864000
```

- server-id : The value for the server-id must be unique for each server and replication client in the MySQL cluster. The `server-id` should be a non-zero value. If the `server-id` is already set to a non-zero value, you don't need to make any change. You can set the `server-id` to any value between 1 and 4294967295. For more information refer mysql doc
- log_bin : The value of log_bin is the base name of the sequence of binlog files. If the `log_bin` is already set, you don't need to make any change. For more information refer mysql doc
- binlog_format : The `binlog_format` must be set to `ROW`. For more information refer mysql doc
- binlog_row_image : The `binlog_row_image` must be set to `FULL`. It determines how row images are written to the binary log. For more information refer mysql doc
- binlog_expire_log_seconds : This is the number of seconds for automatic binlog file removal. We recommend 864000 seconds (10 days) so that in case of a failure in sync or if the sync is paused, we still have some bandwidth to start from the last point in incremental sync. We also recommend setting frequent syncs for CDC.

2. Enable GTIDs (Optional)

Global transaction identifiers (GTIDs) uniquely identify transactions that occur on a server within a cluster. Though not required for a Airbyte MySQL connector, using GTIDs simplifies replication and enables you to more easily confirm if primary and replica servers are consistent. For more information refer mysql doc

- Enable gtid_mode : Boolean that specifies whether GTID mode of the MySQL server is enabled or not. Enable it via `mysql> gtid_mode=ON`
- Enable enforce_gtid_consistency : Boolean that specifies whether the server enforces GTID consistency by allowing the execution of statements that can be logged in a transactionally safe manner. Required when using GTIDs. Enable it via `mysql> enforce_gtid_consistency=ON`

3. Set up initial waiting time(Optional)
DANGER

This is an advanced feature. Use it if absolutely necessary.

The MySQl connector may need some time to start processing the data in the CDC mode in the following scenarios:

- When the connection is set up for the first time and a snapshot is needed
- When the connector has a lot of change logs to process

The connector waits for the default initial wait time of 5 minutes (300 seconds). Setting the parameter to a longer duration will result in slower syncs, while setting it to a shorter duration may cause the connector to not have enough time to create the initial snapshot or read through the change logs. The valid range is 300 seconds to 1200 seconds.

If you know there are database changes to be synced, but the connector cannot read those changes, the root cause may be insufficient waiting time. In that case, you can increase the waiting time (example: set to 600 seconds) to test if it is indeed the root cause. On the other hand, if you know there are no database changes, you can decrease the wait time to speed up the zero record syncs.

4. Set up server timezone(Optional)
DANGER

This is an advanced feature. Use it if absolutely necessary.

In CDC mode, the MySQl connector may need a timezone configured if the existing MySQL database been set up with a system timezone that is not recognized by the IANA Timezone Database.

In this case, you can configure the server timezone to the equivalent IANA timezone compliant timezone. (e.g. CEST -> Europe/Berlin).

Note

When a sync runs for the first time using CDC, Airbyte performs an initial consistent snapshot of your database. Airbyte doesn't acquire any table locks (for tables defined with MyISAM engine, the tables would still be locked) while creating the snapshot to allow writes by other database clients. But in order for the sync to work without any error/unexpected behaviour, it is assumed that no schema changes are happening while the snapshot is running.

If seeing `EventDataDeserializationException` errors intermittently with root cause `EOFException` or `SocketException`, you may need to extend the following *MySql server* timeout values by running:

```
Unset


set global slave_net_timeout = 120;

set global thread_pool_idle_timeout = 120;
```

Connection via SSH Tunnel

Airbyte has the ability to connect to a MySQl instance via an SSH Tunnel. The reason you might want to do this because it is not possible (or against security policy) to connect to the database directly (e.g. it does not have a public IP address).

When using an SSH tunnel, you are configuring Airbyte to connect to an intermediate server (a.k.a. a bastion sever) that *does* have direct access to the database. Airbyte connects to the bastion and then asks the bastion to connect directly to the server.

Using this feature requires additional configuration, when creating the source. We will talk through what each piece of configuration means.

1. Configure all fields for the source as you normally would, except `SSH Tunnel Method`.
2. `SSH Tunnel Method` defaults to `No Tunnel` (meaning a direct connection). If you want to use an SSH Tunnel choose `SSH Key Authentication` or `Password Authentication`.
    i. Choose `Key Authentication` if you will be using an RSA private key as your secret for establishing the SSH Tunnel (see below for more information on generating this key).
    ii. Choose `Password Authentication` if you will be using a password as your secret for establishing the SSH Tunnel.
3. DANGER
   Since Airbyte Cloud requires encrypted communication, select SSH Key Authentication or Password Authentication if you selected preferred as the SSL Mode; otherwise, the connection will fail.
4. `SSH Tunnel Jump Server Host` refers to the intermediate (bastion) server that Airbyte will connect to. This should be a hostname or an IP Address.

5. `SSH Connection Port` is the port on the bastion server with which to make the SSH connection. The default port for SSH connections is `22`, so unless you have explicitly changed something, go with the default.
6. `SSH Login Username` is the username that Airbyte should use when connection to the bastion server. This is NOT the MySQl username.
7. If you are using `Password Authentication`, then `SSH Login Username` should be set to the password of the User from the previous step. If you are using `SSH Key Authentication` leave this blank. Again, this is not the MySQl password, but the password for the OS-user that Airbyte is using to perform commands on the bastion.
8. If you are using `SSH Key Authentication`, then `SSH Private Key` should be set to the RSA Private Key that you are using to create the SSH connection. This should be the full contents of the key file starting with `-----BEGIN RSA PRIVATE KEY-----` and ending with `-----END RSA PRIVATE KEY-----`.

Generating an SSH Key Pair

The connector expects an RSA key in PEM format. To generate this key:

```
Unset


ssh-keygen -t rsa -m PEM -f myuser_rsa
```

This produces the private key in pem format, and the public key remains in the standard format used by the `authorized_keys` file on your bastion host. The public key should be added to your bastion host to whichever user you want to use with Airbyte. The private key is provided via copy-and-paste to the Airbyte connector configuration screen, so it may log in to the bastion.

Data Type Mapping

MySQL data types are mapped to the following data types when synchronizing data. You can check the test values examples here. If you can't find the data type you are looking for or have any problems feel free to add a new test!

MySQL Type	Resulting Type	Notes
`bit(1)`	boolean	
`bit(>1)`	base64 binary string	
`boolean`	boolean	
`tinyint(1)`	boolean	
`tinyint(>1)`	number	
`tinyint(>=1) unsigned`	number	
`smallint`	number	
`mediumint`	number	
`int`	number	
`bigint`	number	
`float`	number	
`double`	number	
`decimal`	number	
`binary`	string	
`blob`	string	

`date`	string	ISO 8601 date string. ZERO-DATE value will be converted to NULL. If column is mandatory, convert to EPOCH.
`datetime`, `timestamp`	string	ISO 8601 datetime string. ZERO-DATE value will be converted to NULL. If column is mandatory, convert to EPOCH.
`time`	string	ISO 8601 time string. Values are in range between 00:00:00 and 23:59:59.
`year`	year string	Doc
`char`, `varchar` with non-binary charset	string	
`char`, `varchar` with binary charset	base64 binary string	
`tinyblob`	base64 binary string	
`blob`	base64 binary string	
`mediumblob`	base64 binary string	
`longblob`	base64 binary string	
`binary`	base64 binary string	

`varbinary`	base64 binary string	
`tinytext`	string	
`text`	string	
`mediumtext`	string	
`longtext`	string	
`json`	serialized json string	E.g. `{"a": 10, "b": 15}`
`enum`	string	
`set`	string	E.g. `blue,green,yellow`
`geometry`	base64 binary string	

If you do not see a type in this list, assume that it is coerced into a string. We are happy to take feedback on preferred mappings.

Microsoft SQL Server (MSSQL)

Features

Feature	Supported	Notes
Full Refresh Sync	Yes	
Incremental Sync - Append	Yes	
Replicate Incremental Deletes	Yes	
CDC (Change Data Capture)	Yes	
SSL Support	Yes	
SSH Tunnel Connection	Yes	
Namespaces	Yes	Enabled by default

The MSSQL source does not alter the schema present in your database. Depending on the destination connected to this source, however, the schema may be altered. See the destination's documentation for more details.

Troubleshooting

You may run into an issue where the connector provides wrong values for some data types. See discussion on unexpected behaviour for certain datatypes.

Note: Currently hierarchyid and sql_variant are not processed in CDC migration type (not supported by debezium). For more details please check this ticket

Getting Started (Airbyte Open-Source)

Requirements

1. MSSQL Server `Azure SQL Database`, `Azure Synapse Analytics`, `Azure SQL Managed Instance`, `SQL Server 2019`, `SQL Server 2017`, `SQL Server 2016`, `SQL Server 2014`, `SQL Server 2012`, `PDW 2008R2 AU34`.
2. Create a dedicated read-only Airbyte user with access to all tables needed for replication
3. If you want to use CDC, please see the relevant section below for further setup requirements

1. Make sure your database is accessible from Seek Connect. This is dependent on your networking setup. The easiest way to verify if Airbyte is able to connect to your MSSQL instance is via the check connection tool in the UI.

2. Create a dedicated read-only user with access to the relevant tables (Recommended but optional)

This step is optional but highly recommended to allow for better permission control and auditing. Alternatively, you can use Airbyte with an existing user in your database.

Coming soon: suggestions on how to create this user.

3. Your database user should now be ready for use with Airbyte!

Change Data Capture (CDC)

We use SQL Server's change data capture feature to capture row-level `INSERT`, `UPDATE` and `DELETE` operations that occur on cdc-enabled tables.

Some extra setup requiring at least *db_owner* permissions on the database(s) you intend to sync from will be required (detailed below).

Please read the CDC docs for an overview of how Airbyte approaches CDC.

Should I use CDC for MSSQL?

- If you need a record of deletions and can accept the limitations posted below, CDC is the way to go!
- If your data set is small and/or you just want a snapshot of your table in the destination, consider using Full Refresh replication for your table instead of CDC.
- If the limitations below prevent you from using CDC and your goal is to maintain a snapshot of your table in the destination, consider using non-CDC incremental and occasionally reset the data and re-sync.
- If your table has a primary key but doesn't have a reasonable cursor field for incremental syncing (i.e. `updated_at`), CDC allows you to sync your table incrementally.

CDC Config

Parameter	Type	Default	Description
Data to Sync	Enum: `Existing and New`, `New Changes Only`	`Existing and New`	What data should be synced under the CDC. `Existing and New` will read existing data as a snapshot, and sync new changes through CDC. `New Changes Only` will skip the initial snapshot, and only sync new changes through CDC. See documentation here for details. Under the hood, this parameter sets the `snapshot.mode` in Debezium.
Snapshot Isolation Level	Enum: `Snapshot`, `Read Committed`	`Snapshot`	Mode to control which transaction isolation level is used and how long the connector locks tables that are designated for capture. If you don't know which one to choose, just use the default one. See documentation here for details. Under the hood, this parameter sets the `snapshot.isolation.mode` in Debezium.

CDC Limitations

- Make sure to read our CDC docs to see limitations that impact all databases using CDC replication.
- There are some critical issues regarding certain datatypes. Please find detailed info in this Github issue.
- CDC is only available for SQL Server 2016 Service Pack 1 (SP1) and later.
- *db_owner* (or higher) permissions are required to perform the neccessary setup for CDC.
- If you set `Initial Snapshot Isolation Level` to `Snapshot`, you must enable snapshot isolation mode on the database(s) you want to sync. This is used for retrieving an initial snapshot without locking tables.
- For SQL Server Always On read-only replica, only `Snapshot` initial snapshot isolation level is supported.
- On Linux, CDC is not supported on versions earlier than SQL Server 2017 CU18 (SQL Server 2019 is supported).
- Change data capture cannot be enabled on tables with a clustered columnstore index. (It can be enabled on tables with a *non-clustered* columnstore index).
- The SQL Server CDC feature processes changes that occur in user-created tables only. You cannot enable CDC on the SQL Server master database.
- Using variables with partition switching on databases or tables with change data capture (CDC) is not supported for the `ALTER TABLE` ... `SWITCH TO` ... `PARTITION` ... statement
- Our implementation has not been tested with managed instances, such as Azure SQL Database (we welcome any feedback from users who try this!)
    - If you do want to try this, CDC can only be enabled on Azure SQL databases tiers above Standard 3 (S3+). Basic, S0, S1 and S2 tiers are not supported for CDC.
- Our CDC implementation uses at least once delivery for all change records.
- Read more on CDC limitations in the Microsoft docs.

Setting up CDC for MSSQL

1. Enable CDC on database and tables

MS SQL Server provides some built-in stored procedures to enable CDC.

- To enable CDC, a SQL Server administrator with the necessary privileges (*db_owner* or *sysadmin*) must first run a query to enable CDC at the database level.

```
Unset
  ●  USE {database name}
  ●  GO
     EXEC sys.sp_cdc_enable_db
     GO
```

- The administrator must then enable CDC for each table that you want to capture. Here's an example:

```
Unset
  ●  USE {database name}
  ●  GO

     EXEC sys.sp_cdc_enable_table
     @source_schema = N'{schema name}',
     @source_name   = N'{table name}',
     @role_name     = N'{role name}',  [1]
     @filegroup_name = N'{fiilegroup name}', [2]
     @supports_net_changes = 0 [3]
     GO
```

  - [1] Specifies a role which will gain `SELECT` permission on the captured columns of the source table. We suggest putting a value here so you can use this role in the next step but you can also set the value of @role_name to `NULL` to allow only *sysadmin* and *db_owner* to have access. Be sure that the credentials used to connect to the source in Airbyte align with this role so that Airbyte can access the cdc tables.
  - [2] Specifies the filegroup where SQL Server places the change table. We recommend creating a separate filegroup for CDC but you can leave this parameter out to use the default filegroup.
  - [3] If 0, only the support functions to query for all changes are generated. If 1, the functions that are needed to query for net changes are also generated. If supports_net_changes is set to 1, index_name must be specified, or the source table must have a defined primary key.

- (For more details on parameters, see the Microsoft doc page for this stored procedure).
- If you have many tables to enable CDC on and would like to avoid having to run this query one-by-one for every table, this script might help!

For further detail, see the Microsoft docs on enabling and disabling CDC.

2. Enable snapshot isolation

- When a sync runs for the first time using CDC, Airbyte performs an initial consistent snapshot of your database. To avoid acquiring table locks, Airbyte uses *snapshot isolation*, allowing simultaneous writes by other database clients. This must be enabled on the database like so:

```
Unset
●
● ALTER DATABASE {database name}
●  SET ALLOW_SNAPSHOT_ISOLATION ON;
```

3. Create a user and grant appropriate permissions

- Rather than use *sysadmin* or *db_owner* credentials, we recommend creating a new user with the relevant CDC access for use with Airbyte. First let's create the login and user and add to the db_datareader role:

```
Unset
●
● USE {database name};
● CREATE LOGIN {user name}
    WITH PASSWORD = '{password}';
  CREATE USER {user name} FOR LOGIN {user name};
  EXEC sp_addrolemember 'db_datareader', '{user name}';
```

  - Add the user to the role specified earlier when enabling cdc on the table(s):

```
Unset
        ○
        ○ EXEC sp_addrolemember '{role name}', '{user
          name}';
```

○

This should be enough access, but if you run into problems, try also directly granting the user `SELECT` access on the cdc schema:

```
Unset
        ○
        ○ USE {database name};
        ○ GRANT SELECT ON SCHEMA :: [cdc] TO {user name};
```

○

If feasible, granting this user 'VIEW SERVER STATE' permissions will allow Airbyte to check whether or not the SQL Server Agent is running. This is preferred as it ensures syncs will fail if the CDC tables are not being updated by the Agent in the source database.

```
Unset
        ○
        ○ USE master;
        ○ GRANT VIEW SERVER STATE TO {user name};
```

4. Extend the retention period of CDC data

- In SQL Server, by default, only three days of data are retained in the change tables. Unless you are running very frequent syncs, we suggest increasing this retention so that in case of a failure in sync or if the sync is paused, there is still some bandwidth to start from the last point in incremental sync.
- These settings can be changed using the stored procedure sys.sp_cdc_change_job as below:

```
Unset
•
• -- we recommend 14400 minutes (10 days) as retention
  period
• EXEC sp_cdc_change_job @job_type='cleanup', @retention
  = {minutes}
```

- After making this change, a restart of the cleanup job is required:

```
Unset


 EXEC sys.sp_cdc_stop_job @job_type = 'cleanup';


 EXEC sys.sp_cdc_start_job @job_type = 'cleanup';
```

5. Ensure the SQL Server Agent is running

- MSSQL uses the SQL Server Agent
  to run the jobs necessary
  for CDC. It is therefore vital that the Agent is operational in order for to CDC to work effectively. You can check
  the status of the SQL Server Agent as follows:

```
Unset


 EXEC xp_servicecontrol 'QueryState', N'SQLServerAGENT';
```

- If you see something other than 'Running.' please follow

the [Microsoft docs](#)
to start the service.

Connection to MSSQL via an SSH Tunnel

Airbyte has the ability to connect to a MSSQL instance via an SSH Tunnel. The reason you might want to do this because it is not possible (or against security policy) to connect to the database directly (e.g. it does not have a public IP address).

When using an SSH tunnel, you are configuring Airbyte to connect to an intermediate server (a.k.a. a bastion sever) that *does* have direct access to the database. Airbyte connects to the bastion and then asks the bastion to connect directly to the server.

Using this feature requires additional configuration, when creating the source. We will talk through what each piece of configuration means.

1. Configure all fields for the source as you normally would, except `SSH Tunnel Method`.
2. `SSH Tunnel Method` defaults to `No Tunnel` (meaning a direct connection). If you want to use an
   SSH Tunnel choose `SSH Key Authentication` or `Password Authentication`.
   i. Choose `Key Authentication` if you will be using an RSA private key as your secret for
      establishing the SSH Tunnel (see below for more information on generating this key).
   ii. Choose `Password Authentication` if you will be using a password as your secret for establishing
      the SSH Tunnel.
3. `SSH Tunnel Jump Server Host` refers to the intermediate (bastion) server that Airbyte will connect to. This should
   be a hostname or an IP Address.
4. `SSH Connection Port` is the port on the bastion server with which to make the SSH connection. The default port for
   SSH connections is `22`, so unless you have explicitly changed something, go with the default.
5. `SSH Login Username` is the username that Airbyte should use when connection to the bastion server. This is NOT the
   MSSQL username.

6. If you are using `Password Authentication`, then `SSH Login Username` should be set to the
   password of the User from the previous step. If you are using `SSH Key Authentication` leave this
   blank. Again, this is not the MSSQL password, but the password for the OS-user that Airbyte is
   using to perform commands on the bastion.
7. If you are using `SSH Key Authentication`, then `SSH Private Key` should be set to the RSA
   private Key that you are using to create the SSH connection. This should be the full contents of
   the key file starting with `-----BEGIN RSA PRIVATE KEY-----` and ending with `-----END RSA PRIVATE KEY-----`.

Generating an SSH Key Pair

The connector expects an RSA key in PEM format. To generate this key:

```
Unset

ssh-keygen -t rsa -m PEM -f myuser_rsa
```

This produces the private key in pem format, and the public key remains in the standard format used by the `authorized_keys` file on your bastion host. The public key should be added to your bastion host to whichever user you want to use with Airbyte. The private key is provided via copy-and-paste to the Airbyte connector configuration screen, so it may log in to the bastion.

Data type mapping

MSSQL data types are mapped to the following data types when synchronizing data. You can check the test values examples here. If you can't find the data type you are looking for or have any problems feel free to add a new test!

MSSQL Type	Resulting Type	Notes
bigint	number	
binary	string	
bit	boolean	
char	string	
date	number	
datetime	string	
datetime2	string	
datetimeoffset	string	
decimal	number	
int	number	
float	number	
geography	string	
geometry	string	
money	number	
numeric	number	
ntext	string	
nvarchar	string	

`nvarchar(max)`	string	
`real`	number	
`smalldatetime`	string	
`smallint`	number	
`smallmoney`	number	
`sql_variant`	string	
`uniqueidentifier`	string	
`text`	string	
`time`	string	
`tinyint`	number	
`varbinary`	string	
`varchar`	string	
`varchar(max) COLLATE Latin1_General_100_CI_AI_SC_UTF8`	string	
`xml`	string	

If you do not see a type in this list, assume that it is coerced into a string. We are happy to take feedback on preferred mappings.

AlloyDB for PostgreSQL

This page contains the setup guide and reference information for the AlloyDB for PostgreSQL.

Prerequisites

- For Airbyte Open Source users, upgrade your Airbyte platform to version `v0.40.0-alpha` or newer
- For Airbyte Cloud (and optionally for Airbyte Open Source), ensure SSL is enabled in your environment

Setup guide

When to use AlloyDB with CDC

Configure AlloyDB with CDC if:

- You need a record of deletions
- Your table has a primary key but doesn't have a reasonable cursor field for incremental syncing (`updated_at`). CDC allows you to sync your table incrementally

If your goal is to maintain a snapshot of your table in the destination but the limitations prevent you from using CDC, consider using non-CDC incremental sync and occasionally reset the data and re-sync.

If your dataset is small and you just want a snapshot of your table in the destination, consider using Full Refresh replication for your table instead of CDC.

Step 1: (Optional) Create a dedicated read-only user

We recommend creating a dedicated read-only user for better permission control and auditing. Alternatively, you can use an existing AlloyDB user in your database.

To create a dedicated user, run the following command:

```
Unset

CREATE USER <user_name> PASSWORD 'your_password_here';
```

Grant access to the relevant schema:

```
Unset

GRANT USAGE ON SCHEMA <schema_name> TO <user_name>
```

NOTE

To replicate data from multiple AlloyDB schemas, re-run the command to grant access to all the relevant schemas. Note that you'll need to set up multiple Airbyte sources connecting to the same AlloyDB database on multiple schemas.

Grant the user read-only access to the relevant tables:

```
Unset

GRANT SELECT ON ALL TABLES IN SCHEMA <schema_name> TO
<user_name>;
```

Allow user to see tables created in the future:

```
Unset

ALTER DEFAULT PRIVILEGES IN SCHEMA <schema_name> GRANT
SELECT ON TABLES TO <user_name>;
```

Additionally, if you plan to configure CDC for the AlloyDB source connector, grant `REPLICATION` permissions to the user:

```
Unset

ALTER USER <user_name> REPLICATION;
```

Syncing a subset of columns

Currently, there is no way to sync a subset of columns using the AlloyDB source connector:

- When setting up a connection, you can only choose which tables to sync, but not columns.
- If the user can only access a subset of columns, the connection check will pass. However, the data sync will fail with a permission denied exception.

The workaround for partial table syncing is to create a view on the specific columns, and grant the user read access to that view:

Unset

```
CREATE VIEW <view_name> as SELECT <columns> FROM <table>;
```

Unset

```
GRANT SELECT ON TABLE <view_name> IN SCHEMA <schema_name>
to <user_name>;
```

Note: The workaround works only for non-CDC setups since CDC requires data to be in tables and not views. This issue is tracked in #9771.

Step 2: Set up the AlloyDB connector in Airbyte

1. Log into your Airbyte Cloud or Airbyte Open Source account.
2. Click Sources and then click + New source.
3. On the Set up the source page, select AlloyDB from the Source type dropdown.
4. Enter a name for your source.
5. For the Host, Port, and DB Name, enter the hostname, port number, and name for your AlloyDB database.
6. List the Schemas you want to sync.
   NOTE
   The schema names are case sensitive. The 'public' schema is set by default. Multiple schemas may be used at one time. No schemas set explicitly - will sync all of existing.
7. For User and Password, enter the username and password you created in Step 1.
8. To customize the JDBC connection beyond common options, specify additional supported JDBC URL parameters as key-value pairs separated by the symbol & in the JDBC URL Parameters (Advanced) field.
   Example: key1=value1&key2=value2&key3=value3
   These parameters will be added at the end of the JDBC URL that the AirByte will use to connect to your AlloyDB database.
   The connector now supports `connectTimeout` and defaults to 60 seconds.

Setting connectTimeout to 0 seconds will set the timeout to the longest time available.
Note: Do not use the following keys in JDBC URL Params field as they will be overwritten by Airbyte: `currentSchema`, `user`, `password`, `ssl`, and `sslmode`.
DANGER
This is an advanced configuration option. Users are advised to use it with caution.

9. For Airbyte Open Source, toggle the switch to connect using SSL. Airbyte Cloud uses SSL by default.
10. For Replication Method, select Standard or Logical CDC from the dropdown. Refer to Configuring AlloyDB connector with Change Data Capture (CDC) for more information.
11. For SSH Tunnel Method, select:
    - No Tunnel for a direct connection to the database
    - SSH Key Authentication to use an RSA Private as your secret for establishing the SSH tunnel
    - Password Authentication to use a password as your secret for establishing the SSH tunnel Refer to Connect via SSH Tunnel for more information.
12. Click Set up source.

Connect via SSH Tunnel

You can connect to a AlloyDB instance via an SSH tunnel.

When using an SSH tunnel, you are configuring Airbyte to connect to an intermediate server (also called a bastion server) that has direct access to the database. Airbyte connects to the bastion and then asks the bastion to connect directly to the server.

To connect to a AlloyDB instance via an SSH tunnel:
1. While setting up the AlloyDB source connector, from the SSH tunnel dropdown, select:
    - SSH Key Authentication to use an RSA Private as your secret for establishing the SSH tunnel
    - Password Authentication to use a password as your secret for establishing the SSH Tunnel
2. For SSH Tunnel Jump Server Host, enter the hostname or IP address for the intermediate (bastion) server that Airbyte will connect to.
3. For SSH Connection Port, enter the port on the bastion server. The default port for SSH connections is 22.

4. For SSH Login Username, enter the username to use when connecting to the bastion server. Note: This is the operating system username and not the AlloyDB username.
5. For authentication:
    - If you selected SSH Key Authentication, set the SSH Private Key to the RSA Private Key that you are using to create the SSH connection.
    - If you selected Password Authentication, enter the password for the operating system user to connect to the bastion server. Note: This is the operating system password and not the AlloyDB password.

Generating an RSA Private Key

The connector expects an RSA key in PEM format. To generate this key, run:

```
Unset


ssh-keygen -t rsa -m PEM -f myuser_rsa
```

The command produces the private key in PEM format and the public key remains in the standard format used by the `authorized_keys` file on your bastion server. Add the public key to your bastion host to the user you want to use with Airbyte. The private key is provided via copy-and-paste to the Airbyte connector configuration screen to allow it to log into the bastion server.

Configuring AlloyDB connector with Change Data Capture (CDC)

Airbyte uses logical replication of the Postgres write-ahead log (WAL) to incrementally capture deletes using a replication plugin. To learn more how Airbyte implements CDC, refer to Change Data Capture (CDC)

CDC Considerations

- Incremental sync is only supported for tables with primary keys. For tables without primary keys, use Full Refresh sync.
- Data must be in tables and not views.

- The modifications you want to capture must be made using `DELETE`/`INSERT`/`UPDATE`. For example, changes made using `TRUNCATE`/`ALTER` will not appear in logs and therefore in your destination.
- Schema changes are not supported automatically for CDC sources. Reset and resync data if you make a schema change.
- The records produced by `DELETE` statements only contain primary keys. All other data fields are unset.
- Log-based replication only works for master instances of AlloyDB.
- Using logical replication increases disk space used on the database server. The additional data is stored until it is consumed.
  - Set frequent syncs for CDC to ensure that the data doesn't fill up your disk space.
  - If you stop syncing a CDC-configured AlloyDB instance with Airbyte, delete the replication slot. Otherwise, it may fill up your disk space.

Setting up CDC for AlloyDB

Airbyte requires a replication slot configured only for its use. Only one source should be configured that uses this replication slot. See Setting up CDC for AlloyDB for instructions.

Step 2: Select a replication plugin

We recommend using a pgoutput plugin (the standard logical decoding plugin in AlloyDB). If the replication table contains multiple JSON blobs and the table size exceeds 1 GB, we recommend using a wal2json instead. Note that wal2json may require additional installation for Bare Metal, VMs (EC2/GCE/etc), Docker, etc. For more information read the wal2json documentation.

Step 3: Create replication slot

To create a replication slot called `airbyte_slot` using pgoutput, run:

```
Unset


SELECT pg_create_logical_replication_slot('airbyte_slot',
'pgoutput');
```

To create a replication slot called `airbyte_slot` using wal2json, run:

```
Unset

SELECT pg_create_logical_replication_slot('airbyte_slot',
'wal2json');
```

Step 4: Create publications and replication identities for tables

For each table you want to replicate with CDC, add the replication identity (the method of distinguishing between rows) first:

To use primary keys to distinguish between rows, run:

```
Unset

ALTER TABLE tbl1 REPLICA IDENTITY DEFAULT;
```

After setting the replication identity, run:

```
Unset

CREATE PUBLICATION airbyte_publication FOR TABLE <tbl1,
tbl2, tbl3>;`
```

The publication name is customizable. Refer to the Postgres docs if you need to add or remove tables from your publication in the future.
NOTE

You must add the replication identity before creating the publication. Otherwise, `ALTER`/`UPDATE`/`DELETE` statements may fail if AlloyDB cannot determine how to uniquely identify rows. Also, the publication should include all the tables and only the tables that need to be synced. Otherwise, data from these tables may not be replicated correctly.

DANGER

The Airbyte UI currently allows selecting any tables for CDC. If a table is selected that is not part of the publication, it will not be replicated even though it is selected. If a table is part of the publication but does not have a replication identity, that replication identity will be created automatically on the first run if the Airbyte user has the necessary permissions.

Step 5: [Optional] Set up initial waiting time

DANGER

This is an advanced feature. Use it if absolutely necessary.

The AlloyDB connector may need some time to start processing the data in the CDC mode in the following scenarios:

- When the connection is set up for the first time and a snapshot is needed
- When the connector has a lot of change logs to process

The connector waits for the default initial wait time of 5 minutes (300 seconds). Setting the parameter to a longer duration will result in slower syncs, while setting it to a shorter duration may cause the connector to not have enough time to create the initial snapshot or read through the change logs. The valid range is 120 seconds to 1200 seconds.

If you know there are database changes to be synced, but the connector cannot read those changes, the root cause may be insufficient waiting time. In that case, you can increase the waiting time (example: set to 600 seconds) to test if it is indeed the root cause. On the other hand, if you know there are no database changes, you can decrease the wait time to speed up the zero record syncs.

Step 6: Set up the AlloyDB source connector

In Step 2 of the connector setup guide, enter the replication slot and publication you just created.

Supported sync modes

The AlloyDB source connector supports the following sync modes:

- Full Refresh - Overwrite
- Full Refresh - Append

- Incremental Sync - Append
- Incremental Sync - Deduped History

Supported cursors

- `TIMESTAMP`
- `TIMESTAMP_WITH_TIMEZONE`
- `TIME`
- `TIME_WITH_TIMEZONE`
- `DATE`
- `BIT`
- `BOOLEAN`
- `TINYINT/SMALLINT`
- `INTEGER`
- `BIGINT`
- `FLOAT/DOUBLE`
- `REAL`
- `NUMERIC/DECIMAL`
- `CHAR/NCHAR/NVARCHAR/VARCHAR/LONGVARCHAR`
- `BINARY/BLOB`

Data type mapping

The AlloyDb is a fully managed PostgreSQL-compatible database service.

According to Postgres documentation, Postgres data types are mapped to the following data types when synchronizing data. You can check the test values examples here. If you can't find the data type you are looking for or have any problems feel free to add a new test!

Postgres Type	Resulting Type	Notes
`bigint`	number	
`bigserial`,	number	

`serial8`		
`bit`	string	Fixed-length bit string (e.g. "0100").
`bit varying`, `varbit`	string	Variable-length bit string (e.g. "0100").
`boolean`, `bool`	boolean	
`box`	string	
`bytea`	string	Variable length binary string with hex output format prefixed with "\x" (e.g. "\x6b707a").
`character`, `char`	string	
`character varying`, `varchar`	string	
`cidr`	string	
`circle`	string	
`date`	string	Parsed as ISO8601 date time at midnight. CDC mode doesn't support era indicators. Issue: #14590
`double precision`, `float`, `float8`	number	`Infinity`, `-Infinity`, and `NaN` are not supported and converted to `null`. Issue: #8902.
`hstore`	string	
`inet`	string	
`integer`, `int`, `int4`	number	

`interval`	string	
`json`	string	
`jsonb`	string	
`line`	string	
`lseg`	string	
`macaddr`	string	
`macaddr8`	string	
`money`	number	
`numeric`, `decimal`	number	`Infinity`, `-Infinity`, and `NaN` are not supported and converted to `null`. Issue: #8902.
`path`	string	
`pg_lsn`	string	
`point`	string	
`polygon`	string	
`real`, `float4`	number	
`smallint`, `int2`	number	
`smallserial`, `serial2`	number	
`serial`, `serial4`	number	
`text`	string	

`time`	string	Parsed as a time string without a time-zone in the ISO-8601 calendar system.
`timetz`	string	Parsed as a time string with time-zone in the ISO-8601 calendar system.
`timestamp`	string	Parsed as a date-time string without a time-zone in the ISO-8601 calendar system.
`timestamptz`	string	Parsed as a date-time string with time-zone in the ISO-8601 calendar system.
`tsquery`	string	
`tsvector`	string	
`uuid`	string	
`xml`	string	
`enum`	string	
`tsrange`	string	
`array`	array	E.g. "[\"10001\",\"10002\",\"10003\",\"10004\"]".
composite type	string	

Limitations

- The AlloyDB source connector currently does not handle schemas larger than 4MB.
- The AlloyDB source connector does not alter the schema present in your database. Depending on the destination connected to this source, however, the schema may be altered. See the destination's documentation for more details.
- The following two schema evolution actions are currently supported:
    - Adding/removing tables without resetting the entire connection at the destination Caveat: In the CDC mode, adding a new table to a connection may become a temporary bottleneck. When a new table is added, the next

    sync job takes a full snapshot of the new table before it proceeds to handle any changes.
    - Resetting a single table within the connection without resetting the rest of the destination tables in that connection
- Changing a column data type or removing a column might break connections.

Postgres

This page contains the setup guide and reference information for the Postgres source connector for CDC and non-CDC workflows.

When to use Postgres with CDC

Configure Postgres with CDC if:

- You need a record of deletions
- Your table has a primary key but doesn't have a reasonable cursor field for incremental syncing (`updated_at`). CDC allows you to sync your table incrementally

If your goal is to maintain a snapshot of your table in the destination but the limitations prevent you from using CDC, consider using non-CDC incremental sync and occasionally reset the data and re-sync.

If your dataset is small and you just want a snapshot of your table in the destination, consider using Full Refresh replication for your table instead of CDC.

Prerequisites

- For Airbyte Open Source users, upgrade your Airbyte platform to version `v0.40.0-alpha` or newer
- Use Postgres v9.3.x or above for non-CDC workflows and Postgres v10 or above for CDC workflows
- For Airbyte Cloud (and optionally for Airbyte Open Source), ensure SSL is enabled in your environment

Setup guide

Step 1: (Optional) Create a dedicated read-only user

We recommend creating a dedicated read-only user for better permission control and auditing. Alternatively, you can use an existing Postgres user in your database.

To create a dedicated user, run the following command:

```
Unset


CREATE USER <user_name> PASSWORD 'your_password_here';
```

Grant access to the relevant schema:

```
Unset


GRANT USAGE ON SCHEMA <schema_name> TO <user_name>
```

NOTE

To replicate data from multiple Postgres schemas, re-run the command to grant access to all the relevant schemas. Note that you'll need to set up multiple Airbyte sources connecting to the same Postgres database on multiple schemas.

Grant the user read-only access to the relevant tables:

```
Unset


GRANT SELECT ON ALL TABLES IN SCHEMA <schema_name> TO
<user_name>;
```

Allow user to see tables created in the future:

```
Unset


ALTER DEFAULT PRIVILEGES IN SCHEMA <schema_name> GRANT
SELECT ON TABLES TO <user_name>;
```

Additionally, if you plan to configure CDC for the Postgres source connector, grant `REPLICATION` permissions to the user:

```
Unset


ALTER USER <user_name> REPLICATION;
```

Syncing a subset of columns

Currently, there is no way to sync a subset of columns using the Postgres source connector:

- When setting up a connection, you can only choose which tables to sync, but not columns.
- If the user can only access a subset of columns, the connection check will pass. However, the data sync will fail with a permission-denied exception.

The workaround for partial table syncing is to create a view on the specific columns, and grant the user read access to that view:

```
Unset

CREATE VIEW <view_name> as SELECT <columns> FROM <table>;
```

```
Unset

GRANT SELECT ON TABLE <view_name> IN SCHEMA <schema_name>
to <user_name>;
```

Note: The workaround works only for non-CDC setups since CDC requires data to be in tables and not views. This issue is tracked in #9771.

Step 2: Set up the Postgres connector in Airbyte

1. Navigate back to Connect and create a new Source.
2. On the Set up the source page, select Postgres from the Source type dropdown.
3. Enter a name for your source.
4. For the Host, Port, and DB Name, enter the hostname, port number, and name for your Postgres database.
5. List the Schemas you want to sync.
   NOTE
   The schema names are case-sensitive. The 'public' schema is set by default. Multiple schemas may be used at one time. No schemas set explicitly - will sync all of existing.
6. For User and Password, enter the username and password you created in Step 1.
7. To customize the JDBC connection beyond common options, specify additional supported JDBC URL parameters as key-value pairs separated by the symbol & in the JDBC URL Parameters (Advanced) field.
   Example: key1=value1&key2=value2&key3=value3
   These parameters will be added at the end of the JDBC URL that the AirByte will use to connect to your Postgres database.

The connector now supports `connectTimeout` and defaults to 60 seconds. Setting connectTimeout to 0 seconds will set the timeout to the longest time available.
Note: Do not use the following keys in JDBC URL Params field as they will be overwritten by Airbyte: `currentSchema`, `user`, `password`, `ssl`, and `sslmode`.
DANGER
This is an advanced configuration option. Users are advised to use it with caution.

8. For Airbyte Open Source, toggle the switch to connect using SSL. For Airbyte Cloud uses SSL by default.
9. For SSL Modes, select:
   - disable to disable encrypted communication between Airbyte and the source
   - allow to enable encrypted communication only when required by the source
   - prefer to allow unencrypted communication only when the source doesn't support encryption
   - require to always require encryption. Note: The connection will fail if the source doesn't support encryption.
   - verify-ca to always require encryption and verify that the source has a valid SSL certificate
   - verify-full to always require encryption and verify the identity of the source
10. For Replication Method, select Standard or Logical CDC from the dropdown. Refer to Configuring Postgres connector with Change Data Capture (CDC) for more information.
11. For SSH Tunnel Method, select:
    - No Tunnel for a direct connection to the database
    - SSH Key Authentication to use an RSA Private as your secret for establishing the SSH tunnel
    - Password Authentication to use a password as your secret for establishing the SSH tunnel
12. DANGER
    Since Airbyte Cloud requires encrypted communication, select SSH Key Authentication or Password Authentication if you selected disable, allow, or prefer as the SSL Mode; otherwise, the connection will fail.

Refer to Connect via SSH Tunnel for more information. 13. Click Set up source.

Connect via SSH Tunnel

You can connect to a Postgres instance via an SSH tunnel.

When using an SSH tunnel, you are configuring Airbyte to connect to an intermediate server (also called a bastion or a jump server) that has direct access to the database. Airbyte connects to the bastion and then asks the bastion to connect directly to the server.

To connect to a Postgres instance via an SSH tunnel:

1. While setting up the Postgres source connector, from the SSH tunnel dropdown, select:
   - SSH Key Authentication to use a private as your secret for establishing the SSH tunnel
   - Password Authentication to use a password as your secret for establishing the SSH Tunnel
2. For SSH Tunnel Jump Server Host, enter the hostname or IP address for the intermediate (bastion) server that Airbyte will connect to.
3. For SSH Connection Port, enter the port on the bastion server. The default port for SSH connections is 22.
4. For SSH Login Username, enter the username to use when connecting to the bastion server. Note: This is the operating system username and not the Postgres username.
5. For authentication:
   - If you selected SSH Key Authentication, set the SSH Private Key to the private Key that you are using to create the SSH connection.
   - If you selected Password Authentication, enter the password for the operating system user to connect to the bastion server. Note: This is the operating system password and not the Postgres password.

Generating a Private Key

The connector supports any SSH compatible key format such as RSA or Ed25519. To generate an RSA key, for example, run:

```
Unset


ssh-keygen -t rsa -m PEM -f myuser_rsa
```

The command produces the private key in PEM format and the public key remains in the standard format used by the `authorized_keys` file on your bastion server. Add the public key to your bastion host to the user you want to use with Airbyte. The private key is provided via copy-and-paste to the Airbyte connector configuration screen to allow it to log into the bastion server.

Configuring Postgres connector with Change Data Capture (CDC)

Airbyte uses logical replication of the Postgres write-ahead log (WAL) to incrementally capture deletes using a replication plugin. To learn more about how Airbyte implements CDC, refer to Change Data Capture (CDC)

CDC Considerations

- Incremental sync is only supported for tables with primary keys. For tables without primary keys, use Full Refresh sync.
- Data must be in tables and not views. If you require data synchronization from a view, you would need to create a new connection with `Standard` as `Replication Method`.
- The modifications you want to capture must be made using `DELETE`/`INSERT`/`UPDATE`. For example, changes made using `TRUNCATE`/`ALTER` will not appear in logs and therefore in your destination.
- Schema changes are not supported automatically for CDC sources. Reset and resync data if you make a schema change.
- The records produced by `DELETE` statements only contain primary keys. All other data fields are unset.
- Log-based replication only works for master instances of Postgres. CDC cannot be run from a read-replica of your primary database.
- Using logical replication increases disk space used on the database server. The additional data is stored until it is consumed.
  - Set frequent syncs for CDC to ensure that the data doesn't fill up your disk space.
  - If you stop syncing a CDC-configured Postgres instance with Airbyte, delete the replication slot. Otherwise, it may fill up your disk space.

CONNECTOR CONFIGURATION ARE SUPPORTED ONLY ON PRIMARY/MASTER DB HOST/SERVERS. DO NOT POINT CONNECTOR CONFIGURATION TO REPLICA DB HOSTS, IT WILL NOT WORK.. :::

Setting up CDC for Postgres

Airbyte requires a replication slot configured only for its use. Only one source should be configured that uses this replication slot. See Setting up CDC for Postgres for instructions.

Step 1: Enable logical replication

To enable logical replication on bare metal, VMs (EC2/GCE/etc), or Docker, configure the following parameters in the postgresql.conf file for your Postgres database:

Parameter	Description	Set value to
wal_level	Type of coding used within the Postgres write-ahead log	logical
max_wal_senders	The maximum number of processes used for handling WAL changes	Min: 1
max_replication_slots	The maximum number of replication slots that are allowed to stream WAL changes	1 (if Airbyte is the only service reading subscribing to WAL changes. More than 1 if other services are also reading from the WAL)

To enable logical replication on AWS Postgres RDS or Aurora:

1. Go to the Configuration tab for your DB cluster.
2. Find your cluster parameter group. Either edit the parameters for this group or create a copy of this parameter group to edit. If you create a copy, change your cluster's parameter group before restarting.
3. Within the parameter group page, search for `rds.logical_replication`. Select this row and click Edit parameters. Set this value to 1.
4. Wait for a maintenance window to automatically restart the instance or restart it manually.

To enable logical replication on Azure Database for Postgres:

Change the replication mode of your Postgres DB on Azure to `logical` using the Replication menu of your PostgreSQL instance in the Azure Portal. Alternatively, use the Azure CLI to run the following command:

Unset

```
az postgres server configuration set --resource-group group
--server-name server --name azure.replication_support
--value logical
```

Unset

```
az postgres server restart --resource-group group --name
server
```

Step 3: Create replication slot

Airbyte currently supports pgoutput plugin only. To create a replication slot called `airbyte_slot` using pgoutput, run:

Unset

```
SELECT pg_create_logical_replication_slot('airbyte_slot',
'pgoutput');
```

Step 4: Create publications and replication identities for tables

For each table you want to replicate with CDC, add the replication identity (the method of distinguishing between rows) first:

To use primary keys to distinguish between rows for tables that don't have a large amount of data per row, run:

```
Unset

ALTER TABLE tbl1 REPLICA IDENTITY DEFAULT;
```

In case your tables use data types that support TOAST and have very large field values, use:

```
Unset

ALTER TABLE tbl1 REPLICA IDENTITY FULL;
```

After setting the replication identity, run:

```
Unset

CREATE PUBLICATION airbyte_publication FOR TABLE <tbl1,
tbl2, tbl3>;`
```

The publication name is customizable. Refer to the Postgres docs if you need to add or remove tables from your publication in the future.

You must add the replication identity before creating the publication. Otherwise, `ALTER`/`UPDATE`/`DELETE` statements may fail if Postgres cannot determine how to uniquely identify rows. Also, the publication should include all the tables and only the tables that need to be synced. Otherwise, data from these tables may not be replicated correctly. :::
DANGER

The Airbyte UI currently allows selecting any tables for CDC. If a table is selected that is not part of the publication, it will not be replicated even though it is selected. If a table is part of the publication but does not have a replication identity, that replication identity will be created automatically on the first run if the Airbyte user has the necessary permissions.

Step 5: [Optional] Set up initial waiting time
DANGER

This is an advanced feature. Use it if absolutely necessary.

The Postgres connector may need some time to start processing the data in the CDC mode in the following scenarios:

- When the connection is set up for the first time and a snapshot is needed
- When the connector has a lot of change logs to process

The connector waits for the default initial wait time of 5 minutes (300 seconds). Setting the parameter to a longer duration will result in slower syncs, while setting it to a shorter duration may cause the connector to not have enough time to create the initial snapshot or read through the change logs. The valid range is 120 seconds to 1200 seconds.

If you know there are database changes to be synced, but the connector cannot read those changes, the root cause may be insufficient waiting time. In that case, you can increase the waiting time (example: set to 600 seconds) to test if it is indeed the root cause. On the other hand, if you know there are no database changes, you can decrease the wait time to speed up the zero record syncs.

Step 6: Set up the Postgres source connector

In Step 2 of the connector setup guide, enter the replication slot and publication you just created.

Supported sync modes

The Postgres source connector supports the following sync modes:

- Full Refresh - Overwrite
- Full Refresh - Append
- Incremental Sync - Append
- Incremental Sync - Deduped History

Supported cursors

- `TIMESTAMP`
- `TIMESTAMP_WITH_TIMEZONE`
- `TIME`
- `TIME_WITH_TIMEZONE`
- `DATE`
- `BIT`
- `BOOLEAN`
- `TINYINT/SMALLINT`
- `INTEGER`
- `BIGINT`
- `FLOAT/DOUBLE`
- `REAL`
- `NUMERIC/DECIMAL`
- `CHAR/NCHAR/NVARCHAR/VARCHAR/LONGVARCHAR`
- `BINARY/BLOB`

Data type mapping

According to Postgres documentation, Postgres data types are mapped to the following data types when synchronizing data. You can check the test values examples here. If you can't find the data type you are looking for or have any problems feel free to add a new test!

Postgres Type	Resulting Type	Notes
`bigint`	number	
`bigserial`, `serial8`	number	
`bit`	string	Fixed-length bit string (e.g. "0100").
`bit varying`,	string	Variable-length bit string (e.g. "0100").

`varbit`		
`boolean`, `bool`	boolean	
`box`	string	
`bytea`	string	Variable length binary string with hex output format prefixed with "\x" (e.g. "\x6b707a").
`character`, `char`	string	
`character varying`, `varchar`	string	
`cidr`	string	
`circle`	string	
`date`	string	Parsed as ISO8601 date time at midnight. CDC mode doesn't support era indicators. Issue: #14590
`double precision`, `float`, `float8`	number	`Infinity`, `-Infinity`, and `NaN` are not supported and converted to `null`. Issue: #8902.
`hstore`	string	
`inet`	string	
`integer`, `int`, `int4`	number	
`interval`	string	
`json`	string	
`jsonb`	string	

`line`	string	
`lseg`	string	
`macaddr`	string	
`macaddr8`	string	
`money`	number	
`numeric`, `decimal`	number	`Infinity`, `-Infinity`, and `NaN` are not supported and converted to `null`. Issue: [#8902](#8902).
`path`	string	
`pg_lsn`	string	
`point`	string	
`polygon`	string	
`real`, `float4`	number	
`smallint`, `int2`	number	
`smallserial`, `serial2`	number	
`serial`, `serial4`	number	
`text`	string	
`time`	string	Parsed as a time string without a time-zone in the ISO-8601 calendar system.
`timetz`	string	Parsed as a time string with time-zone in the ISO-8601 calendar system.

`timestamp`	string	Parsed as a date-time string without a time-zone in the ISO-8601 calendar system.
`timestamptz`	string	Parsed as a date-time string with time-zone in the ISO-8601 calendar system.
`tsquery`	string	
`tsvector`	string	
`uuid`	string	
`xml`	string	
`enum`	string	
`tsrange`	string	
`array`	array	E.g. "[\"10001\",\"10002\",\"10003\",\"10004\"]".
composite type	string	

Limitations

- The Postgres source connector currently does not handle schemas larger than 4MB.
- The Postgres source connector does not alter the schema present in your database. Depending on the destination connected to this source, however, the schema may be altered. See the destination's documentation for more details.
- The following two schema evolution actions are currently supported:
  - Adding/removing tables without resetting the entire connection at the destination Caveat: In the CDC mode, adding a new table to a connection may become a temporary bottleneck. When a new table is added, the next sync job takes a full snapshot of the new table before it proceeds to handle any changes.
  - Resetting a single table within the connection without resetting the rest of the destination tables in that connection
- Changing a column data type or removing a column might break connections.

Troubleshooting

Sync data from Postgres hot standby server

When the connector is reading from a Postgres replica that is configured as a Hot Standby, any update from the primary server will terminate queries on the replica after a certain amount of time, default to 30 seconds. This default waiting time is not enough to sync any meaning amount of data. See the `Handling Query Conflicts` section in the Postgres documentation for detailed explanation.

Here is the typical exception:

```
Unset


Caused by: org.postgresql.util.PSQLException: FATAL:
terminating connection due to conflict with recovery

   Detail: User query might have needed to see row versions
that must be removed.

   Hint: In a moment you should be able to reconnect to the
database and repeat your command.
```

Possible solutions include:

- [Recommended] Set `hot_standby_feedback` to `true` on the replica server. This parameter will prevent the primary server from deleting the write-ahead logs when the replica is busy serving user queries. However, the downside is that the write-ahead log will increase in size.
- [Recommended] Sync data when there is no update running in the primary server, or sync data from the primary server.
- [Not Recommended] Increase `max_standby_archive_delay` and `max_standby_streaming_delay` to be larger than the amount of time needed to complete the data sync. However, it is usually hard to tell how much time it will take to sync all the data. This approach is not very practical.

Under CDC incremental mode, there are still full refresh syncs

Normally under the CDC mode, the Postgres source will first run a full refresh sync to read the snapshot of all the existing data, and all subsequent runs will only be incremental syncs reading from the write-ahead logs (WAL). However, occasionally, you may see full refresh syncs after the initial run. When this happens, you will see the following log:
Saved offset is before Replication slot's confirmed_flush_lsn, Airbyte will trigger sync from scratch

The root causes is that the WALs needed for the incremental sync has been removed by Postgres. This can occur under the following scenarios:

- When there are lots of database updates resulting in more WAL files than allowed in the `pg_wal` directory, Postgres will purge or archive the WAL files. This scenario is preventable. Possible solutions include:
  - Sync the data source more frequently. The downside is that more computation resources will be consumed, leading to a higher Airbyte bill.
  - Set a higher `wal_keep_size`. If no unit is provided, it is in megabytes, and the default is `0`. See detailed documentation here. The downside of this approach is that more disk space will be needed.
- When the Postgres connector successfully reads the WAL and acknowledges it to Postgres, but the destination connector fails to consume the data, the Postgres connector will try to read the same WAL again, which may have been removed by Postgres, since the WAL record is already acknowledged. This scenario is rare, because it can happen, and currently there is no way to prevent it. The correct behavior is to perform a full refresh.

Kafka

This page guides you through the process of setting up the Kafka source connector.

Set up guide

Step 1: Set up Kafka

To use the Kafka source connector, you'll need:

- A Kafka cluster 1.0 or above
- Airbyte user should be allowed to read messages from topics, and these topics should be created before reading from Kafka.

Step 2: Setup the Kafka source in Airbyte

You'll need the following information to configure the Kafka source:

- Group ID - The Group ID is how you distinguish different consumer groups. (e.g. group.id)
- Protocol - The Protocol used to communicate with brokers.
- Client ID - An ID string to pass to the server when making requests. The purpose of this is to be able to track the source of requests beyond just ip/port by allowing a logical application name to be included in server-side request logging. (e.g. airbyte-consumer)
- Test Topic - The Topic to test in case the Airbyte can consume messages. (e.g. test.topic)
- Subscription Method - You can choose to manually assign a list of partitions, or subscribe to all topics matching specified pattern to get dynamically assigned partitions.
- List of topic

- Bootstrap Servers - A list of host/port pairs to use for establishing the initial connection to the Kafka cluster.
- Schema Registry - Host/port to connect schema registry server. Note: It supports for AVRO format only.

For Airbyte Open Source:

1. Go to the Airbyte UI and in the left navigation bar, click Sources. In the top-right corner, click +new source.
2. On the Set up the source page, enter the name for the Kafka connector and select Kafka from the Source type dropdown.
3. Follow the Setup the Kafka source in Airbyte

Supported sync modes

The Kafka source connector supports the following sync modes:

Feature	Supported?(Yes/No)	Notes
Full Refresh Sync	Yes	
Incremental - Append Sync	Yes	
Namespaces	No	

Supported Format

JSON - Json value messages. It does not support schema registry now.

AVRO - deserialize Using confluent API. Please refer (https://docs.confluent.io/platform/current/schema-registry/serdes-develop/serdes-avro.html)

ClickHouse

Overview

The ClickHouse source supports both Full Refresh and Incremental syncs. You can choose if this connector will copy only the new or updated data, or all rows in the tables and columns you set up for replication, every time a sync is run.

This Clickhouse source connector is built on top of the source-jdbc code base and is configured to rely on JDBC v0.3.1 standard drivers provided by ClickHouse here as described in ClickHouse documentation here.

Resulting schema

The ClickHouse source does not alter the schema present in your warehouse. Depending on the destination connected to this source, however, the schema may be altered. See the destination's documentation for more details.

Features

Feature	Supported	Notes
Full Refresh Sync	Yes	

Incremental Sync	Yes	
Replicate Incremental Deletes	Coming soon	
Logical Replication (WAL)	Coming soon	
SSL Support	Yes	
SSH Tunnel Connection	Yes	
Namespaces	Yes	Enabled by default

Getting started

Requirements

1. ClickHouse Server `21.3.10.1` or later.
2. Create a dedicated read-only Airbyte user with access to all tables needed for replication

Setup guide

1. Make sure your database is accessible from the machine running Airbyte

This is dependent on your networking setup. The easiest way to verify if Airbyte is able to connect to your ClickHouse instance is via the check connection tool in the UI.

2. Create a dedicated read-only user with access to the relevant tables (Recommended but optional)

This step is optional but highly recommended to allow for better permission control and auditing. Alternatively, you can use Airbyte with an existing user in your database.

To create a dedicated database user, run the following commands against your database:

```
Unset

CREATE USER 'airbyte'@'%' IDENTIFIED BY
'your_password_here';
```

Then give it access to the relevant schema:

```
Unset

GRANT SELECT ON <database name>.* TO 'airbyte'@'%';
```

You can limit this grant down to specific tables instead of the whole database. Note that to replicate data from multiple ClickHouse databases, you can re-run the command above to grant access to all the relevant schemas, but you'll need to set up multiple sources connecting to the same db on multiple schemas.

Your database user should now be ready for use with Airbyte.

Connection via SSH Tunnel

Airbyte has the ability to connect to a Clickhouse instance via an SSH Tunnel. The reason you might want to do this because it is not possible (or against security policy) to connect to the database directly (e.g. it does not have a public IP address).

When using an SSH tunnel, you are configuring Airbyte to connect to an intermediate server (a.k.a. a bastion sever) that *does* have direct access to the database. Airbyte connects to the bastion and then asks the bastion to connect directly to the server.

Using this feature requires additional configuration, when creating the source. We will talk through what each piece of configuration means.

1. Configure all fields for the source as you normally would, except `SSH Tunnel Method`.
2. `SSH Tunnel Method` defaults to `No Tunnel` (meaning a direct connection). If you want to use an SSH Tunnel choose `SSH Key Authentication` or `Password Authentication`.

   i. Choose `Key Authentication` if you will be using an RSA private key as your secret for establishing the SSH Tunnel (see below for more information on generating this key).
   ii. Choose `Password Authentication` if you will be using a password as your secret for establishing the SSH Tunnel.
3. `SSH Tunnel Jump Server Host` refers to the intermediate (bastion) server that Airbyte will connect to. This should be a hostname or an IP Address.
4. `SSH Connection Port` is the port on the bastion server with which to make the SSH connection. The default port for SSH connections is `22`, so unless you have explicitly changed something, go with the default.
5. `SSH Login Username` is the username that Airbyte should use when connection to the bastion server. This is NOT the Clickhouse username.
6. If you are using `Password Authentication`, then `SSH Login Username` should be set to the password of the User from the previous step. If you are using `SSH Key Authentication` leave this blank. Again, this is not the Clickhouse password, but the password for the OS-user that Airbyte is using to perform commands on the bastion.
7. If you are using `SSH Key Authentication`, then `SSH Private Key` should be set to the RSA Private Key that you are using to create the SSH connection. This should be the full contents of the key file starting with `-----BEGIN RSA PRIVATE KEY-----` and ending with `-----END RSA PRIVATE KEY-----`.

CockroachDB

Overview

The CockroachDB source supports both Full Refresh and Incremental syncs. You can choose if this connector will copy only the new or updated data, or all rows in the tables and columns you set up for replication, every time a sync is run.

Resulting schema

The CockroachDb source does not alter the schema present in your database. Depending on the destination connected to this source, however, the schema may be altered. See the destination's documentation for more details.

Data type mapping

CockroachDb data types are mapped to the following data types when synchronizing data:

CockroachDb Type	Resulting Type	Notes

`bigint`	integer	
`bit`	boolean	
`boolean`	boolean	
`character`	string	
`character varying`	string	
`date`	string	
`double precision`	string	
`enum`	number	
`inet`	string	
`int`	integer	
`json`	string	
`jsonb`	string	
`numeric`	number	
`smallint`	integer	
`text`	string	
`time with timezone`	string	may be written as a native date type depending on the destination
`time without timezone`	string	may be written as a native date type depending on the destination

`timestamp with timezone`	string	may be written as a native date type depending on the destination
`timestamp without timezone`	string	may be written as a native date type depending on the destination
`uuid`	string	

Note: arrays for all the above types as well as custom types are supported, although they may be de-nested depending on the destination.

Features

Feature	Supported	Notes
Full Refresh Sync	Yes	
Incremental Sync	Yes	
Change Data Capture	No	
SSL Support	Yes	

Getting started

Requirements

1. CockroachDb `v1.15.x` or above
2. Allow connections from Airbyte to your CockroachDb database (if they exist in separate VPCs)
3. Create a dedicated read-only Airbyte user with access to all tables needed for replication

Setup guide

1. Make sure your database is accessible from the machine running Airbyte

This is dependent on your networking setup. The easiest way to verify if Airbyte is able to connect to your CockroachDb instance is via the check connection tool in the UI.

2. Create a dedicated read-only user with access to the relevant tables (Recommended but optional)

This step is optional but highly recommended to allow for better permission control and auditing. Alternatively, you can use Airbyte with an existing user in your database.

To create a dedicated database user, run the following commands against your database:

```
Unset


CREATE USER airbyte PASSWORD 'your_password_here';
```

Then give it access to the relevant schema:

```
Unset


GRANT USAGE ON SCHEMA <schema_name> TO airbyte
```

Note that to replicate data from multiple CockroachDb schemas, you can re-run the command above to grant access to all the relevant schemas, but you'll need to set up multiple sources connecting to the same db on multiple schemas.

Next, grant the user read-only access to the relevant tables. The simplest way is to grant read access to all tables in the schema as follows:

```
Unset


GRANT SELECT ON ALL TABLES IN SCHEMA <schema_name> TO
airbyte;
```

```
# Allow airbyte user to see tables created in the future

ALTER DEFAULT PRIVILEGES IN SCHEMA <schema_name> GRANT
SELECT ON TABLES TO airbyte;
```

3. That's it!

Your database user should now be ready for use with Airbyte.

Convex

This page contains the setup guide and reference information for the Convex source connector.

Get started with Convex at the Convex website. See your data on the Convex dashboard.

Overview

The Convex source connector supports Full Refresh, Incremental Append, and Incremental Dedupe with deletes.

Output schema

This source syncs each Convex table as a separate stream. Check out the list of your tables on the Convex dashboard in the "Data" view.

Types not directly supported by JSON are encoded as described in the JSONSchema for the stream.

For example, the Javascript value `new Set(["a", "b"])` is encoded as `{"$set": ["a", "b"]}`, as described by the JSONSchema `{"type": "object", "description": "Set", "properties": {"$set": {"type": "array", "items": {"type": "string"}}}}`.

Every record includes the client-defined fields for the table, for example a `"messages"` table may contain fields for `"author"` and `"body"`. Additionally, each document has system fields:

1. `_id` uniquely identifies the document. It is not changed by `.patch` or `.replace` operations.
2. `_creationTime` records a timestamp in milliseconds when the document was initially created. It is not changed by `.patch` or `.replace` operations.
3. `_ts` records a timestamp in nanoseconds when the document was last modified. It can be used for ordering operations in Incremental Append mode, and is automatically used in Incremental Dedupe mode.
4. `_deleted` identifies whether the document was deleted. It can be used to filter deleted documents in Incremental Append mode, and is automatically used to remove documents in Incremental Dedupe mode.

Features

Feature	Supported?
Full Refresh Sync	Yes
Incremental - Append Sync	Yes
Incremental - Dedupe Sync	Yes
Replicate Incremental	Yes

Deletes	
Change Data Capture	Yes
Namespaces	No

Performance considerations

The Convex connector syncs all documents from the historical log. If you see performance issues due to syncing unnecessary old versions of documents, please reach out to Convex support.

Getting started

Requirements

- Convex Account
- Convex Project
- Deploy key

Setup guide

Airbyte integration is available to Convex teams on Professional plans.

On the Convex dashboard, navigate to the project that you want to sync. Note only "Production" deployments should be synced.

In the Data tab, you should see the tables and a sample of the data that will be synced.

1. Navigate to the Settings tab.
2. Copy the "Deployment URL" from the settings page to the `deployment_url` field in Airbyte.
3. Click "Generate a deploy key".
4. Copy the generated deploy key into the `access_key` field in Airbyte.

Elasticsearch

This page contains the setup guide and reference information for the Elasticsearch source connector.

Prerequisites

Requirements

- Elasticsearch endpoint URL
- Elasticsearch credentials (optional)

Supported sync modes

Feature	Supported?(Yes/No)	Notes

Full Refresh Sync	Yes	
Incremental Sync	No	

This source syncs data from an ElasticSearch domain.

Supported Streams

This source automatically discovers all indices in the domain and can sync any of them.

Performance Considerations

ElasticSearch calls may be rate limited by the underlying service. This is specific to each deployment.

Data type map

Elasticsearch data types:
https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping-types.html

Airbyte Data Types

In Elasticsearch, there is no dedicated array data type. Any field can contain zero or more values by default, however, all values in the array must be of the same data type. Hence, every field can be an array as well.

Integration Type	**Airbyte Type**	**Notes**
`binary`	`["string", "array"]`	

boolean	["boolean", "array"]	
keyword	["string", "array", "number", "integer"]	
constant_keyword	["string", "array", "number", "integer"]	
wildcard	["string", "array", "number", "integer"]	
long	["integer", "array"]	
unsigned_long	["integer", "array"]	
integer	["integer", "array"]	
short	["integer", "array"]	
byte	["integer", "array"]	
double	["number", "array"]	
float	["number", "array"]	
half_float	["number", "array"]	
scaled_float	["number", "array"]	
date	["string", "array"]	
date_nanos	["number", "array"]	
object	["object", "array"]	
flattened	["object", "array"]	
nested	["object", "string"]	

join	["object", "string"]	
integer_range	["object", "array"]	
float_range	["object", "array"]	
long_range	["object", "array"]	
double_range	["object", "array"]	
date_range	["object", "array"]	
ip_range	["object", "array"]	
ip	["string", "array"]	
version	["string", "array"]	
murmur3	["string", "array", "number", "integer"]	
aggregate_metric_double	["string", "array", "number", "integer"]	
histogram	["string", "array", "number", "integer"]	
text	["string", "array", "number", "integer"]	
alias	["string", "array", "number", "integer"]	
search_as_you_type	["string", "array", "number", "integer"]	
token_count	["string", "array",	

	"number", "integer"]	
dense_vector	["string", "array", "number", "integer"]	
geo_point	["string", "array", "number", "integer"]	
geo_shape	["string", "array", "number", "integer"]	
shape	["string", "array", "number", "integer"]	
point	["string", "array", "number", "integer"]	

Fauna

This page guides you through setting up a Fauna source.

Overview

The Fauna source supports the following sync modes:

- Full Sync - exports all the data from a Fauna collection.
- Incremental Sync - exports data incrementally from a Fauna collection.

You need to create a separate source per collection that you want to export.

Preliminary setup

Enter the domain of the collection's database that you are exporting. The URL can be found in the docs.

Full sync

Follow these steps if you want this connection to perform a full sync.

1. Create a role that can read the collection that you are exporting. You can create the role in the Dashboard or the fauna shell with the following query:

```
Unset



CreateRole({

 name: "airbyte-readonly",

 privileges: [

   {

     resource: Collections(),

     actions: { read: true }

   },

   {

     resource: Indexes(),

     actions: { read: true }
```

```
    },

    {

      resource: Collection("COLLECTION_NAME"),

      actions: { read: true }

    }

  ],

})
```

Replace `COLLECTION_NAME` with the name of the collection configured for this connector. If you'd like to sync multiple collections, add an entry for each additional collection you'd like to sync. For example, to sync `users` and `products`, run this query instead:

```
Unset



CreateRole({

 name: "airbyte-readonly",
privileges: [
  {
    resource: Collections(),
    actions: { read: true }
  },
  {
    resource: Indexes(),
    actions: { read: true }
  },
  {
    resource: Collection("users"),
    actions: { read: true }
  },
    {
      resource: Collection("products"),
      actions: { read: true }
    }
  ],
})
```

2. Create a key with that role. You can create a key using this query:

```
Unset

CreateKey({
 name: "airbyte-readonly",
 role: Role("airbyte-readonly"),
})
```

3. Copy the `secret` output by the `CreateKey` command and enter that as the "Fauna Secret" on the left. Important: The secret is only ever displayed once. If you lose it, you would have to create a new key.

Incremental sync

Follow these steps if you want this connection to perform incremental syncs.

1. Create the "Incremental Sync Index". This allows the connector to perform incremental syncs. You can create the index with the fauna shell or in the Dashboard with the following query:

```
Unset



CreateIndex({

 name: "INDEX_NAME",

 source: Collection("COLLECTION_NAME"),

 terms: [],

 values: [

   { "field": "ts" },

   { "field": "ref" }

 ]
})
```

Replace `COLLECTION_NAME` with the name of the collection configured for this connector. Replace `INDEX_NAME` with the name that you configured for the Incremental Sync Index.

Repeat this step for every collection you'd like to sync.

2. Create a role that can read the collection, the index, and the metadata of all indexes. It needs access to index metadata in order to validate the index settings. You can create the role with this query:

```
Unset



CreateRole({

 name: "airbyte-readonly",

 privileges: [

   {

     resource: Collections(),

     actions: { read: true }

   },
   {
     resource: Indexes(),
     actions: { read: true }
   },
   {
     resource: Collection("COLLECTION_NAME"),
     actions: { read: true }
   },
   {
     resource: Index("INDEX_NAME"),
     actions: { read: true }
   }
 ],
})
```

Replace `COLLECTION_NAME` with the name of the collection configured for this connector. Replace `INDEX_NAME` with the name that you configured for the Incremental Sync Index.

If you'd like to sync multiple collections, add an entry for every collection and index you'd like to sync. For example, to sync `users` and `products` with Incremental Sync, run the following query:

```
Unset

CreateRole({
 name: "airbyte-readonly",
 privileges: [
   {
     resource: Collections(),
     actions: { read: true }
   },
   {
     resource: Indexes(),
     actions: { read: true }
   },
   {
     resource: Collection("users"),
     actions: { read: true }
   },
   {
     resource: Index("users-ts"),
     actions: { read: true }
   },
   {
      resource: Collection("products"),

      actions: { read: true }

    },

    {

      resource: Index("products-ts"),

      actions: { read: true }

    }

  ],

})
```

3. Create a key with that role. You can create a key using this query:

```
Unset



CreateKey({

 name: "airbyte-readonly",
  role: Role("airbyte-readonly"),

})
```

4. Copy the `secret` output by the `CreateKey` command and enter that as the "Fauna Secret" on the left. Important: The secret is only ever displayed once. If you lose it, you would have to create a new key.

Export formats

This section captures export formats for all special case data stored in Fauna. This list is exhaustive.

Note that the `ref` column in the exported database contains only the document ID from each document's reference (or "ref"). Since only one collection is involved in each connector configuration, it is inferred that the document ID refers to a document within the synced collection.

Fauna Type	Format	Note
Document Ref	`{ id: "id", "collection": "collection-name", "type": "document" }`	
Other Ref	`{ id: "id", "type": "ref-type" }`	This includes all other refs, listed below.
Byte Array	base64 url formatting	
Timestamp	date-time, or an iso-format timestamp	

Query, SetRef	a string containing the wire protocol of this value	The wire protocol is not documented.

Ref types

Every ref is serialized as a JSON object with 2 or 3 fields, as listed above. The `type` field must be one of these strings:

Reference Type	`type` string
Document	`"document"`
Collection	`"collection"`
Database	`"database"`
Index	`"index"`
Function	`"function"`
Role	`"role"`
AccessProvider	`"access_provider"`
Key	`"key"`
Token	`"token"`
Credential	`"credential"`

For all other refs (for example if you stored the result of `Collections()`), the `type` must be `"unknown"`. There is a difference between a specific collection ref (retrieved with `Collection("name")`), and all the reference to all collections (retrieved with

`Collections()`). This is why the `type` is `"unknown"` for `Collections()`, but not for `Collection("name")`

To select the document ID from a ref, add `"id"` to the "Path" of the additional column. For example, if "Path" is `["data", "parent"]`, change the "Path" to `["data", "parent", "id"]`.

To select the collection name, add `"collection", "id"` to the "Path" of the additional column. For example, if "Path" is `["data", "parent"]`, change the "Path" to `["data", "parent", "collection", "id"]`. Internally, the FQL `Select` is used.

Firebase Realtime Database

Overview

The Firebase Realtime Database source supports Full Refresh sync. As the database data is stored as JSON objects and there are no records or tables, you can sync only one stream which you specifed as a JSON node path on your database at a time.

Resulting schema

As mentioned above, fetched data is just a JSON objects. The resulting records conformed of two columns `key` and `value`. The `key`'s value is keys (string) of fetched JSON object. The `value`'s value is string representation of values (string representation of any JSON object) of fetched JSON object.

If your database has data as below at path `https://{your-database-name}.firebaseio.com/store-a/users.json` ...

```
Unset

{
 "liam": {"address": "somewhere", "age": 24},
 "olivia": {"address": "somewhere", "age": 30}
}
```

and you specified a `store-a/users` as a path in configuration, you would sync records like below ...

```
Unset

{"key": "liam", "value": "{\"address\": \"somewhere\",
\"age\": 24}}"}
{"key": "olivia", "value": "{\"address\": \"somewhere\",
\"age\": 30}}"}
```

Features

Feature	Supported	Notes
Full Refresh Sync	Yes	
Incremental Sync	No	
Change Data Capture	No	
SSL Support	Yes	

Getting started

Requirements

To use the Firebase Realtime Database source, you'll need:

- A Google Cloud Project with Firebase enabled
- A Google Cloud Service Account with the "Firebase Realtime Database Viewer" roles in your Google Cloud project
- A Service Account Key to authenticate into your Service Account

See the setup guide for more information about how to create the required resources.

Service account

In order for Airbyte to sync data from Firebase Realtime Database, it needs credentials for a Service Account with the "Firebase Realtime Database Viewer" roles, which grants permissions to read from Firebase Realtime Database. We highly recommend that this Service Account is exclusive to Airbyte for ease of permissioning and auditing. However, you can use a pre-existing Service Account if you already have one with the correct permissions.

The easiest way to create a Service Account is to follow Google Cloud's guide for Creating a Service Account. Once you've created the Service Account, make sure to keep its ID handy as you will need to reference it when granting roles. Service Account IDs typically take the form `<account-name>@<project-name>.iam.gserviceaccount.com`

Then, add the service account as a Member in your Google Cloud Project with the "Firebase Realtime Database Viewer" role. To do this, follow the instructions for Granting Access in the Google documentation. The email address of the member you are adding is the same as the Service Account ID you just created.

At this point you should have a service account with the "Firebase Realtime Database" product-level permission.

Service account key

Service Account Keys are used to authenticate as Google Service Accounts. For Airbyte to leverage the permissions you granted to the Service Account in the previous step, you'll need to provide its Service Account Keys. See the Google documentation for more information about Keys.

Follow the Creating and Managing Service Account Keys guide to create a key. Airbyte currently supports JSON Keys only, so make sure you create your key in that format. As soon as you created the key, make sure to download it, as that is the only time Google will allow you to see its contents. Once you've successfully configured Firebase Realtime Database as a source in Airbyte, delete this key from your computer.

Setup the Firebase Realtime Database source in Airbyte

You should now have all the requirements needed to configure Firebase Realtime Database as a source in the UI. You'll need the following information to configure the Firebase Realtime Database source:

- Database Name
- Service Account Key JSON: the contents of your Service Account Key JSON file.
- Node Path [Optional]: node path in your database's data which you want to sync. default value is ""(root node).
- Buffer Size [Optional]: number of records to fetch at one time (buffered). default value is 10000.

Once you've configured Firebase Realtime Database as a source, delete the Service Account Key from your computer.

Db2

Overview

The IBM Db2 source allows you to sync data from Db2. It supports both Full Refresh and Incremental syncs. You can choose if this connector will copy only the new or updated data, or all rows in the tables and columns you set up for replication, every time a sync is run.

This IBM Db2 source connector is built on top of the IBM Data Server Driver for JDBC and SQLJ. It is a pure-Java driver (Type 4) that supports the JDBC 4 specification as described in IBM Db2 documentation.

Resulting schema

The IBM Db2 source does not alter the schema present in your warehouse. Depending on the destination connected to this source, however, the result schema may be altered. See the destination's documentation for more details.

Features

Feature	Supported?(Yes/No)	Notes
Full Refresh Sync	Yes	
Incremental - Append Sync	Yes	
Namespaces	Yes	

Getting started

Requirements

1. You'll need the following information to configure the IBM Db2 source:
2. Host
3. Port
4. Database
5. Username
6. Password
7. Create a dedicated read-only Airbyte user and role with access to all schemas needed for replication.

Setup guide

1. Specify the port, host, and name of the database.

2. Create a dedicated read-only user with access to the relevant schemas (Recommended but optional)

This step is optional but highly recommended allowing for better permission control and auditing. Alternatively, you can use Airbyte with an existing user in your database.

Please create a dedicated database user and run the following commands against your database:

```
Unset




-- create Airbyte role

CREATE ROLE 'AIRBYTE_ROLE';



-- grant Airbyte database access

GRANT CONNECT ON 'DATABASE' TO ROLE 'AIRBYTE_ROLE'

GRANT ROLE 'AIRBYTE_ROLE' TO USER 'AIRBYTE_USER'
```

Your database user should now be ready for use with Airbyte.

3. Create SSL connection.

To set up an SSL connection, you need to use a client certificate. Add it to the "SSL PEM file" field and the connector will automatically add it to the secret keystore. You can also enter your own password for the keystore, but if you don't, the password will be generated automatically.

Microsoft Dataverse

Sync overview

This source can sync data for the Microsoft Dataverse API to work with Microsoft Dataverse.

This connector currently uses version v9.2 of the API

Output schema

This source will automatically discover the schema of the Entities of your Dataverse instance using the API `https://<url>/api/data/v9.2/EntityDefinitions?$expand=Attributes`

Data type mapping

Integration Type	Airbyte Type	Notes
`String`	`string`	
`UniqueIdentifier`	`string`	
`DateTime`	`timestamp with timezone`	
`Integer`	`integer`	
`BigInt`	`integer`	
`Money`	`number`	
`Boolean`	`boolean`	
`Double`	`number`	
`Decimal`	`number`	
`Status`	`integer`	
`State`	`integer`	
`Virtual`	None	We skip virtual types

Other types are defined as `string`.

Features

Feature	Supported?(Yes/No)	Notes
Full Refresh Sync	Yes	
Incremental Sync	Yes	
CDC	Yes	Not all entities support it. Deleted data only have the ID
Replicate Incremental Deletes	Yes	
SSL connection	Yes	
Namespaces	No	

Getting started

Requirements

- Application (client) ID
- Directory (tenant) ID
- Client secrets

Setup guide

The Microsoft Dataverse API uses OAuth2 for authentication. We need a 'client_credentials' type, that we usually get by using an App Registration. https://learn.microsoft.com/en-us/power-apps/developer/data-platform/authenticate-oauth

The procedure to generate the credentials and setup the necessary permissions is well described in this post from Magnetism blog: https://blog.magnetismsolutions.com/blog/paulnieuwelaar/2021/9/21/setting-up-an-application-user-in-dynamics-365

Mongo DB

The MongoDB source allows to sync data from MongoDb. Source supports Full Refresh and Incremental sync strategies.

Resulting schema

MongoDB does not have anything like table definition, thus we have to define column types from actual attributes and their values. The discover phase has two steps:

Step 1. Find all unique properties

Connector select 10k documents to collect all distinct field.

Step 2. Determine property types

For each property found, connector determines its type, if all the selected values have the same type - connector will set appropriate type to the property. In all other cases connector will fallback to `string` type.

Features

Feature	Supported
Full Refresh Sync	Yes
Incremental - Append Sync	Yes
Replicate Incremental Deletes	No
Namespaces	No

Full Refresh sync

Works as usual full refresh sync.

Incremental sync

Cursor field can not be nested. Currently, only top-level document properties are supported.

Cursor should never be blank. In case cursor is blank - the incremental sync results might be unpredictable and will totally rely on MongoDB comparison algorithm.

Only `datetime` and `number` cursor types are supported. Cursor type is determined based on the cursor field name:

- `datetime` - if cursor field name contains a string from: `time`, `date`, `_at`, `timestamp`, `ts`

- `number` - otherwise

Getting started

This guide describes in detail how you can configure MongoDB for integration with Airbyte.

Create users

Run `mongo` shell, switch to `admin` database and create a `READ_ONLY_USER`. `READ_ONLY_USER` will be used for Airbyte integration. Please make sure that user has read-only privileges.

```
Unset



mongo

use admin;

db.createUser({user: "READ_ONLY_USER", pwd:
"READ_ONLY_PASSWORD", roles: [{role: "read", db:
"TARGET_DATABASE"}]})
```

Make sure the user have appropriate access levels, a user with higher access levels may throw an exception.

Enable MongoDB authentication

Open `/etc/mongod.conf` and add/replace specific keys:

```
Unset



net:

 bindIp: 0.0.0.0



security:

 authorization: enabled
```

Binding to `0.0.0.0` will allow to connect to database from any IP address.

The last line will enable MongoDB security. Now only authenticated users will be able to access the database.

Configure firewall

Make sure that MongoDB is accessible from external servers. Specific commands will depend on the firewall you are using (UFW/iptables/AWS/etc). Please refer to appropriate documentation.

Your `READ_ONLY_USER` should now be ready for use with Airbyte.

TLS/SSL on a Connection

It is recommended to use encrypted connection. Connection with TLS/SSL security protocol for MongoDb Atlas Cluster and Replica Set instances is enabled by default. To enable TSL/SSL connection with Standalone MongoDb instance, please refer to MongoDb Documentation.

Configuration Parameters

- Database: database name
- Authentication Source: specifies the database that the supplied credentials should be validated against. Defaults to `admin`.
- User: username to use when connecting
- Password: used to authenticate the user
- Standalone MongoDb instance
  - Host: URL of the database
  - Port: Port to use for connecting to the database
  - TLS: indicates whether to create encrypted connection
- Replica Set
  - Server addresses: the members of a replica set
  - Replica Set: A replica set name
- MongoDb Atlas Cluster
  - Cluster URL: URL of a cluster to connect to

For more information regarding configuration parameters, please see MongoDb Documentation.

Oracle Netsuite

One unified business management suite, encompassing ERP/Financials, CRM and ecommerce for more than 31,000 customers.

This connector implements the SuiteTalk REST Web Services and uses REST API to fetch the customers data.

Prerequisites

- Oracle NetSuite account
- Allowed access to all Account permissions options

Airbyte OSS and Airbyte Cloud

- Realm (Account ID)
- Consumer Key
- Consumer Secret
- Token ID
- Token Secret

Setup guide

Step 1: Create NetSuite account

1. Create account on Oracle NetSuite
2. Confirm your Email

Step 2: Setup NetSuite account

Step 2.1: Obtain Realm info

1. Login into your NetSuite account
2. Go to Setup » Company » Company Information
3. Copy your Account ID (Realm). It should look like 1234567 for the `Production` env. or 1234567_SB2 - for a `Sandbox`

Step 2.2: Enable features

1. Go to Setup » Company » Enable Features
2. Click on SuiteCloud tab
3. Scroll down to SuiteScript section
4. Enable checkbox for `CLIENT SUITESCRIPT` and `SERVER SUITESCRIPT`
5. Scroll down to Manage Authentication section
6. Enable checkbox `TOKEN-BASED AUTHENTICATION`
7. Scroll down to SuiteTalk (Web Services)
8. Enable checkbox `REST WEB SERVISES`
9. Save the changes

Step 2.3: Create Integration (obtain Consumer Key and Consumer Secret)

1. Go to Setup » Integration » Manage Integrations » New
2. Fill the Name field (we recommend to put `airbyte-rest-integration` for a name)
3. Make sure the State is `enabled`
4. Enable checkbox `Token-Based Authentication` in Authentication section
5. Save changes

6. After that, Consumer Key and Consumer Secret will be showed once (copy them to the safe place)

Step 2.4: Setup Role

1. Go to Setup » Users/Roles » Manage Roles » New
2. Fill the Name field (we recommend to put `airbyte-integration-role` for a name)
3. Scroll down to Permissions tab
4. (REQUIRED) Click on `Transactions` and manually `add` all the dropdown entities with either `full` or `view` access level.
5. (REQUIRED) Click on `Reports` and manually `add` all the dropdown entities with either `full` or `view` access level.
6. (REQUIRED) Click on `Lists` and manually `add` all the dropdown entities with either `full` or `view` access level.
7. (REQUIRED) Click on `Setup` and manually `add` all the dropdown entities with either `full` or `view` access level.

- Make sure you've done all `REQUIRED` steps correctly, to avoid sync issues in the future.
- Please edit these params again when you `rename` or `customise` any `Object` in Netsuite for `airbyte-integration-role` to reflect such changes.

Step 2.5: Setup User

1. Go to Setup » Users/Roles » Manage Users
2. In column `Name` click on the user's name you want to give access to the `airbyte-integration-role`
3. Then click on Edit button under the user's name
4. Scroll down to Access tab at the bottom
5. Select from dropdown list the `airbyte-integration-role` role which you created in step 2.4
6. Save changes

Step 2.6: Create Access Token for role

1. Go to Setup » Users/Roles » Access Tokens » New
2. Select an Application Name
3. Under User select the user you assigned the `airbyte-integration-role` in the step 2.4
4. Inside Role select the one you gave to the user in the step 2.5

5. Under Token Name you can give a descriptive name to the Token you are creating (we recommend to put `airbyte-rest-integration-token` for a name)
6. Save changes
7. After that, Token ID and Token Secret will be showed once (copy them to the safe place)

Step 2.7: Summary

You have copied next parameters

- Realm (Account ID)
- Consumer Key
- Consumer Secret
- Token ID
- Token Secret Also you have properly Configured Account with Correct Permissions and Access Token for User and Role you've created early.

Step 3: Set up the source connector in Airbyte

For Airbyte Cloud:

1. Log into your Airbyte Cloud account.
2. In the left navigation bar, click Sources. In the top-right corner, click + new source.
3. On the source setup page, select NetSuite from the Source type dropdown and enter a name for this connector.
4. Add Realm
5. Add Consumer Key
6. Add Consumer Secret
7. Add Token ID
8. Add Token Secret
9. Click `Set up source`.

For Airbyte OSS:

1. Go to local Airbyte page.
2. In the left navigation bar, click Sources. In the top-right corner, click + new source.
3. On the source setup page, select NetSuite from the Source type dropdown and enter a name for this connector.

4. Add Realm
5. Add Consumer Key
6. Add Consumer Secret
7. Add Token ID
8. Add Token Secret
9. Click `Set up source`

Supported sync modes

The NetSuite source connector supports the following sync modes:

- Full Refresh
- Incremental

Supported Streams

- Streams are generated based on `ROLE` and `USER` access to them as well as `Account` settings, make sure you're using the correct role assigned in our case `airbyte-integration-role` or any other custom `ROLE` granted to the Access Token, having the access to the NetSuite objects for data sync, please refer to the Setup guide > Step 2.4 and Setup guide > Step 2.5

Performance considerations

The connector is restricted by Netsuite Concurrency Limit per Integration.

TiDB

Overview

TiDB (/'taɪdiːbiː/, "Ti" stands for Titanium) is an open-source, distributed, NewSQL database that supports Hybrid Transactional and Analytical Processing (HTAP) workloads. It is MySQL compatible and features horizontal scalability, strong consistency, and high availability. TiDB can be deployed on-premise or in-cloud.

The TiDB source supports both Full Refresh and Incremental syncs. You can choose if this connector will copy only the new or updated data, or all rows in the tables and columns you set up for replication, every time a sync is run.

Resulting schema

The TiDB source does not alter the schema present in your database. Depending on the destination connected to this source, however, the schema may be altered. See the destination's documentation for more details.

Features

Feature	Supported	Notes
Full Refresh Sync	Yes	
Incremental - Append Sync	Yes	
Replicate Incremental Deletes	Yes	
Change Data Capture	No	
SSL Support	Yes	
SSH Tunnel Connection	Yes	

Getting Started

Requirements

1. TiDB `v4.0` or above
2. Allow connections from Airbyte to your TiDB database (if they exist in separate VPCs)
3. (Optional) Create a dedicated read-only Airbyte user with access to all tables needed for replication

Note: When connecting to [TiDB Cloud]() with TLS enabled, you need to specify TLS protocol, such as `enabledTLSProtocols=TLSv1.2` or `enabledTLSProtocols=TLSv1.3` in the JDBC parameters.

Setup guide

1. Make sure your database is accessible from the machine running Airbyte

This is dependent on your networking setup. The easiest way to verify if Airbyte is able to connect to your TiDB instance is via the check connection tool in the UI.

2. Create a dedicated read-only user with access to the relevant tables (Recommended but optional)

This step is optional but highly recommended to allow for better permission control and auditing. Alternatively, you can use Airbyte with an existing user in your database.

To create a dedicated database user, run the following commands against your database:

```
Unset




CREATE USER 'airbyte'@'%' IDENTIFIED BY
'your_password_here';
```

Then give it access to the relevant database:

```
Unset




GRANT SELECT ON <database name>.* TO 'airbyte'@'%';
```

3. That's it!

Your database user should now be ready for use with Airbyte.

Connection via SSH Tunnel

Airbyte has the ability to connect to a TiDB instance via an SSH Tunnel. The reason you might want to do this because it is not possible (or against security policy) to connect to the database directly (e.g. it does not have a public IP address).

When using an SSH tunnel, you are configuring Airbyte to connect to an intermediate server (a.k.a. a bastion sever) that *does* have direct access to the database. Airbyte connects to the bastion and then asks the bastion to connect directly to the server.

Using this feature requires additional configuration, when creating the source. We will talk through what each piece of configuration means.

1. Configure all fields for the source as you normally would, except `SSH Tunnel Method`.
2. `SSH Tunnel Method` defaults to `No Tunnel` (meaning a direct connection). If you want to use an SSH Tunnel choose `SSH Key Authentication` or `Password Authentication`.
   i. Choose `Key Authentication` if you will be using an RSA private key as your secret for establishing the SSH Tunnel (see below for more information on generating this key).
   ii. Choose `Password Authentication` if you will be using a password as your secret for establishing the SSH Tunnel.
3. `SSH Tunnel Jump Server Host` refers to the intermediate (bastion) server that Airbyte will connect to. This should be a hostname or an IP Address.
4. `SSH Connection Port` is the port on the bastion server with which to make the SSH connection. The default port for SSH connections is `22`, so unless you have explicitly changed something, go with the default.
5. `SSH Login Username` is the username that Airbyte should use when connection to the bastion server. This is NOT the TiDB username.
6. If you are using `Password Authentication`, then `SSH Login Username` should be set to the password of the User from the previous step. If you are using `SSH Key Authentication` TiDB password, but the password for the OS-user that Airbyte is using to perform commands on the bastion.
7. If you are using `SSH Key Authentication`, then `SSH Private Key` should be set to the RSA Private Key that you are using to create the SSH connection. This should be the full contents of the key file starting with `-----BEGIN RSA PRIVATE KEY-----` and ending with `-----END RSA PRIVATE KEY-----`.

Data type mapping

TiDB data types are mapped to the following data types when synchronizing data:

TiDB Type	Resulting Type	Notes
`bit(1)`	boolean	
`bit(>1)`	base64 binary string	
`boolean`	boolean	
`tinyint(1)`	boolean	
`tinyint`	number	
`smallint`	number	
`mediumint`	number	
`int`	number	
`bigint`	number	
`float`	number	
`double`	number	
`decimal`	number	
`binary`	base64 binary string	
`blob`	base64	

	binary string	
`date`	string	ISO 8601 date string. ZERO-DATE value will be converted to NULL. If column is mandatory, convert to EPOCH.
`datetime`, `timestamp`	string	ISO 8601 datetime string. ZERO-DATE value will be converted to NULL. If column is mandatory, convert to EPOCH.
`time`	string	ISO 8601 time string. Values are in range between 00:00:00 and 23:59:59.
`year`	year string	Doc
`char`, `varchar` with non-binary charset	string	
`char`, `varchar` with binary charset	base64 binary string	
`tinyblob`	base64 binary string	
`blob`	base64 binary string	
`mediumblob`	base64 binary string	
`longblob`	base64 binary string	

`binary`	base64 binary string	
`varbinary`	base64 binary string	
`tinytext`	string	
`text`	string	
`mediumtext`	string	
`longtext`	string	
`json`	serialized json string	E.g. `{"a": 10, "b": 15}`
`enum`	string	
`set`	string	E.g. `blue,green,yellow`

Note: arrays for all the above types as well as custom types are supported, although they may be de-nested depending on the destination.

External resources

Now that you have set up the TiDB source connector, check out the following TiDB tutorial:

- Using Airbyte to Migrate Data from TiDB Cloud to Snowflake

Warehouses and Lakes

Redshift

Overview

The Redshift source supports Full Refresh syncs. That is, every time a sync is run, Airbyte will copy all rows in the tables and columns you set up for replication into the destination in a new table.

This Redshift source connector is built on top of the source-jdbc code base and is configured to rely on JDBC 4.2 standard drivers provided by Amazon via Mulesoft here as described in Redshift documentation here.

Sync overview

Resulting schema

The Redshift source does not alter the schema present in your warehouse. Depending on the destination connected to this source, however, the schema may be altered. See the destination's documentation for more details.

Features

Feature	Supported	Notes
Full Refresh Sync	Yes	
Incremental Sync	Coming soon	
Replicate Incremental Deletes	Coming soon	
Logical Replication (WAL)	Coming soon	
SSL Support	Yes	
SSH Tunnel Connection	Coming soon	
Namespaces	Yes	Enabled by default
Schema Selection	Yes	Multiple schemas may be used at one time. Keep empty to process all of existing schemas

Incremental Sync

Incremental sync (copying only the data that has changed) for this source is coming soon.

Getting Started

Requirements

1. Active Redshift cluster
2. Allow connections from Airbyte to your Redshift cluster (if they exist in separate VPCs)

Setup guide

1. Make sure your cluster is active and accessible from the machine running Airbyte

This is dependent on your networking setup. The easiest way to verify if Airbyte is able to connect to your Redshift cluster is via the check connection tool in the UI. You can check AWS Redshift documentation with a tutorial on how to properly configure your cluster's access here

2. Fill up connection info

Next is to provide the necessary information on how to connect to your cluster such as the `host` whcih is part of the connection string or Endpoint accessible here without the `port` and `database` name (it typically includes the cluster-id, region and end with `.redshift.amazonaws.com`).

Encryption

All Redshift connections are encrypted using SSL

Snowflake

Overview

The Snowflake source allows you to sync data from Snowflake. It supports both Full Refresh and Incremental syncs. You can choose if this connector will copy only the new or updated data, or all rows in the tables and columns you set up for replication, every time a sync is run.

This Snowflake source connector is built on top of the source-jdbc code base and is configured to rely on JDBC 3.13.22 Snowflake driver as described in Snowflake documentation.

Resulting schema

The Snowflake source does not alter the schema present in your warehouse. Depending on the destination connected to this source, however, the result schema may be altered. See the destination's documentation for more details.

Features

Feature	Supported?(Yes/No)	Notes
Full Refresh Sync	Yes	
Incremental - Append	Yes	

Sync		
Namespaces	Yes	

Getting started

Requirements

1. You'll need the following information to configure the Snowflake source:
2. Host
3. Role
4. Warehouse
5. Database
6. Schema
7. Username
8. Password
9. JDBC URL Params (Optional)
10. Create a dedicated read-only Airbyte user and role with access to all schemas needed for replication.

Setup guide

1. Additional information about Snowflake connection parameters could be found here.

2. Create a dedicated read-only user with access to the relevant schemas (Recommended but optional)

This step is optional but highly recommended to allow for better permission control and auditing. Alternatively, you can use Airbyte with an existing user in your database.

To create a dedicated database user, run the following commands against your database:

```
Unset

-- set variables (these need to be uppercase)
```

```
SET AIRBYTE_ROLE = 'AIRBYTE_ROLE';
SET AIRBYTE_USERNAME = 'AIRBYTE_USER';

-- set user password
SET AIRBYTE_PASSWORD = '-password-';

BEGIN;

-- create Airbyte role
CREATE ROLE IF NOT EXISTS $AIRBYTE_ROLE;

-- create Airbyte user
CREATE USER IF NOT EXISTS $AIRBYTE_USERNAME
PASSWORD = $AIRBYTE_PASSWORD
DEFAULT_ROLE = $AIRBYTE_ROLE
DEFAULT_WAREHOUSE= $AIRBYTE_WAREHOUSE;

-- grant Airbyte schema access
GRANT OWNERSHIP ON SCHEMA $AIRBYTE_SCHEMA TO ROLE
$AIRBYTE_ROLE;

COMMIT;
```

You can limit this grant down to specific schemas instead of the whole database. Note that to replicate data from multiple Snowflake databases, you can re-run the command above to grant access to all the relevant schemas, but you'll need to set up multiple sources connecting to the same db on multiple schemas.

Your database user should now be ready for use with Airbyte.

Authentication

There are 2 ways of OAuth supported: Login / Password and OAuth2.

Login and Password

Field	Description
Host	The host domain of the snowflake instance (must include the account, region, cloud environment, and end with snowflakecomputing.com). Example: `accountname.us-east-2.aws.snowflakecomputing.com`
Role	The role you created in Step 1 for Airbyte to access Snowflake. Example: `AIRBYTE_ROLE`
Warehouse	The warehouse you created in Step 1 for Airbyte to sync data into. Example: `AIRBYTE_WAREHOUSE`
Database	The database you created in Step 1 for Airbyte to sync data into. Example: `AIRBYTE_DATABASE`
Schema	The schema whose tables this replication is targeting. If no schema is specified, all tables with permission will be presented regardless of their schema.
Username	The username you created in Step 2 to allow Airbyte to access the database. Example: `AIRBYTE_USER`
Password	The password associated with the username.

JDBC URL Params (Optional)	Additional properties to pass to the JDBC URL string when connecting to the database formatted as `key=value` pairs separated by the symbol `&`. Example: `key1=value1&key2=value2&key3=value3`

OAuth 2.0

Field	Description
Host	The host domain of the snowflake instance (must include the account, region, cloud environment, and end with snowflakecomputing.com). Example: `accountname.us-east-2.aws.snowflakecomputing.com`
Role	The role you created in Step 1 for Airbyte to access Snowflake. Example: `AIRBYTE_ROLE`
Warehouse	The warehouse you created in Step 1 for Airbyte to sync data into. Example: `AIRBYTE_WAREHOUSE`
Database	The database you created in Step 1 for Airbyte to sync data into. Example: `AIRBYTE_DATABASE`
Schema	The schema whose tables this replication is targeting. If no schema is specified, all tables with permission will be presented regardless of their schema.
OAuth2	The Login name and password to obtain auth token.
JDBC URL Params (Optional)	Additional properties to pass to the JDBC URL string when connecting to the database formatted as `key=value` pairs separated by the symbol `&`. Example: `key1=value1&key2=value2&key3=value3`

Network policies

By default, Snowflake allows users to connect to the service from any computer or device IP address. A security administrator (i.e. users with the SECURITYADMIN role) or higher can create a network policy to allow or deny access to a single IP address or a list of addresses.

If you have any issues connecting with Airbyte Cloud please make sure that the list of IP addresses is on the allowed list

To determine whether a network policy is set on your account or for a specific user, execute the *SHOW PARAMETERS* command.

Account

```
Unset


   SHOW PARAMETERS LIKE 'network_policy' IN ACCOUNT;
```

User

```
Unset


   SHOW PARAMETERS LIKE 'network_policy' IN USER
<username>;
```

To read more please check official Snowflake documentation

Azure Table Storage

Overview

The Azure table storage supports Full Refresh and Incremental syncs. You can choose which tables you want to replicate.

Output schema

This Source have generic schema for all streams. Azure Table storage is a service that stores non-relational structured data (also known as structured NoSQL data). There is no efficient way to read schema for the given table. We use `data` property to have all the properties for any given row.

- data - This property contain all values
- additionalProperties - This property denotes that all the values are in `data` property.

```
{ "$schema": "http://json-schema.org/draft-07/schema#", "type":
"object", "properties": { "data": { "type": "object" },
"additionalProperties": { "type": "boolean" } } }
```

Data type mapping

Azure Table Storage uses different property types and Airbyte uses internally (`string`, `date-time`, `object`, `array`, `boolean`, `integer`, and `number`). We don't apply any explicit data type mappings.

Features

Feature	Supported?
Full Refresh Sync	Yes
Incremental - Append Sync	Yes
Incremental - Dedupe Sync	No
SSL connection	Yes
Namespaces	No

Performance considerations

The Azure table storage connector should not run into API limitations under normal usage. Please create an issue if you see any rate limit issues that are not automatically retried successfully.

Getting Started

Requirements

- Azure Storage Account
- Azure Storage Account Key
- Azure Storage Endpoint Suffix

Setup guide

Visit the Azure Portal. Go to your storage account, you can find :

- Azure Storage Account - under the overview tab
- Azure Storage Account Key - under the Access keys tab
- Azure Storage Endpoint Suffix - under the Enpoint tab

We recommend creating a restricted key specifically for Airbyte access. This will allow you to control which resources Airbyte should be able to access. However, shared access key authentication is not supported by this connector yet.

BigQuery

Overview

The BigQuery source supports both Full Refresh and Incremental syncs. You can choose if this connector will copy only the new or updated data, or all rows in the tables and columns you set up for replication, every time a sync is running.

Resulting schema

The BigQuery source does not alter the schema present in your database. Depending on the destination connected to this source, however, the schema may be altered. See the destination's documentation for more details.

Data type mapping

The BigQuery data types mapping:

BigQuery Type	Resulting Type	Notes
`BOOL`	Boolean	
`INT64`	Number	
`FLOAT64`	Number	
`NUMERIC`	Number	

`BIGNUMERIC`	Number	
`STRING`	String	
`BYTES`	String	
`DATE`	String	In ISO8601 format
`DATETIME`	String	In ISO8601 format
`TIMESTAMP`	String	In ISO8601 format
`TIME`	String	
`ARRAY`	Array	
`STRUCT`	Object	
`GEOGRAPHY`	String	

Features

Feature	Supported	Notes
Full Refresh Sync	Yes	
Incremental Sync	Yes	
Change Data Capture	No	
SSL Support	Yes	

Getting started

Requirements

To use the BigQuery source, you'll need:

- A Google Cloud Project with BigQuery enabled
- A Google Cloud Service Account with the "BigQuery User" and "BigQuery Data Editor" roles in your GCP project
- A Service Account Key to authenticate into your Service Account

See the setup guide for more information about how to create the required resources.

Service account

In order for Airbyte to sync data from BigQuery, it needs credentials for a Service Account with the "BigQuery User" and "BigQuery Data Editor" roles, which grants permissions to run BigQuery jobs, write to BigQuery Datasets, and read table metadata. We highly recommend that this Service Account is exclusive to Airbyte for ease of permissioning and auditing. However, you can use a pre-existing Service Account if you already have one with the correct permissions.

The easiest way to create a Service Account is to follow GCP's guide for Creating a Service Account. Once you've created the Service Account, make sure to keep its ID handy as you will need to reference it when granting roles. Service Account IDs typically take the form `<account-name>@<project-name>.iam.gserviceaccount.com`

Then, add the service account as a Member in your Google Cloud Project with the "BigQuery User" role. To do this, follow the instructions for Granting Access in the Google documentation. The email address of the member you are adding is the same as the Service Account ID you just created.

At this point you should have a service account with the "BigQuery User" project-level permission.

Service account key

Service Account Keys are used to authenticate as Google Service Accounts. For Airbyte to leverage the permissions you granted to the Service Account in the previous step, you'll need to provide its Service Account Keys. See the Google documentation for more information about Keys.

Follow the Creating and Managing Service Account Keys guide to create a key. Airbyte currently supports JSON Keys only, so make sure you create your key in that format. As soon as you created the key, make sure to download it, as that is the only time Google will allow you to see its contents. Once you've successfully configured BigQuery as a source in Airbyte, delete this key from your computer.

Setup the BigQuery source in Airbyte

You should now have all the requirements needed to configure BigQuery as a source in the UI. You'll need the following information to configure the BigQuery source:

- Project ID
- Default Dataset ID [Optional]: the schema name if only one schema is interested. Dramatically boost source discover operation.
- Credentials JSON: the contents of your Service Account Key JSON file

Once you've configured BigQuery as a source, delete the Service Account Key from your computer.

Firebolt

Overview

The Firebolt source allows you to sync your data from Firebolt. Only Full refresh is supported at the moment.

The connector is built on top of a pure Python firebolt-sdk and does not require additional dependencies.

Resulting schema

The Firebolt source does not alter schema present in your database. Depending on the destination connected to this source, however, the result schema may be altered. See the destination's documentation for more details.

Features

Feature	Supported?(Yes/No)	Notes
Full Refresh Sync	Yes	
Incremental - Append Sync	No	

Getting started

Requirements

1. An existing AWS account

Setup guide

1. Create a Firebolt account following the guide
2. Follow the getting started tutorial to setup a database
3. Load data
4. Create an Analytics (read-only) engine as described in here

You should now have the following

1. An existing Firebolt account
2. Connection parameters handy
    i. Username
    ii. Password
    iii. Account, in case of a multi-account setup (Optional)
    iv. Host (Optional)
    v. Engine (Optional), preferably Analytics/read-only
3. A running engine (if an engine is stopped or booting up you won't be able to connect to it)
4. Your data is in either Fact or Dimension tables.

You can now use the Airbyte Firebolt source.

S3

This page contains the setup guide and reference information for the Amazon S3 source connector.

Prerequisites

Define file pattern, see the Path Patterns section

Setup guide

Step 1: Set up Amazon S3

- If syncing from a private bucket, the credentials you use for the connection must have have both `read` and `list` access on the S3 bucket. `list` is required to discover files based on the provided pattern(s).

Step 2: Set up the Amazon S3 connector in Airbyte

For Airbyte Cloud:

1. Log into your Airbyte Cloud account.
2. In the left navigation bar, click <Sources/Destinations>. In the top-right corner, click +new source/destination.
3. On the Set up the source/destination page, enter the name for the `connector name` connector and select connector name from the `Source/Destination` type dropdown.
4. Set `dataset` appropriately. This will be the name of the table in the destination.
5. If your bucket contains *only* files containing data for this table, use `**` as path_pattern. See the Path Patterns section for more specific pattern matching.

6. Leave schema as `{}` to automatically infer it from the file(s). For details on providing a schema, see the User Schema section.
7. Fill in the fields within the provider box appropriately. If your bucket is not public, add credentials with sufficient permissions under `aws_access_key_id` and `aws_secret_access_key`.
8. Choose the format corresponding to the format of your files and fill in the fields as required. If unsure about values, try out the defaults and come back if needed. Find details on these settings here.

For Airbyte Open Source:

1. Create a new S3 source with a suitable name. Since each S3 source maps to just a single table, it may be worth including that in the name.
2. Set `dataset` appropriately. This will be the name of the table in the destination.
3. If your bucket contains *only* files containing data for this table, use `**` as path_pattern. See the Path Patterns section for more specific pattern matching.
4. Leave schema as `{}` to automatically infer it from the file(s). For details on providing a schema, see the User Schema section.
5. Fill in the fields within the provider box appropriately. If your bucket is not public, add credentials with sufficient permissions under `aws_access_key_id` and `aws_secret_access_key`.
6. Choose the format corresponding to the format of your files and fill in fields as required. If unsure about values, try out the defaults and come back if needed. Find details on these settings here.

Supported sync modes

The Amazon S3 source connector supports the following sync modes:

Feature	Supported?
Full Refresh Sync	Yes
Incremental Sync	Yes
Replicate Incremental Deletes	No
Replicate Multiple Files (pattern	Yes

matching)	
Replicate Multiple Streams (distinct tables)	No
Namespaces	No

File Compressions

Compression	Supported?
Gzip	Yes
Zip	No
Bzip2	Yes
Lzma	No
Xz	No
Snappy	No

Please let us know any specific compressions you'd like to see support for next!

Path Patterns

(tl;dr -> path pattern syntax using wcmatch.glob. GLOBSTAR and SPLIT flags are enabled.)

This connector can sync multiple files by using glob-style patterns, rather than requiring a specific path for every file. This enables:

- Referencing many files with just one pattern, e.g. `**` would indicate every file in the bucket.
- Referencing future files that don't exist yet (and therefore don't have a specific path).

You must provide a path pattern. You can also provide many patterns split with | for more complex directory layouts.

Each path pattern is a reference from the *root* of the bucket, so don't include the bucket name in the pattern(s).

Some example patterns:

- `**` : match everything.
- `**/*.csv` : match all files with specific extension.
- `myFolder/**/*.csv` : match all csv files anywhere under myFolder.
- `*/**` : match everything at least one folder deep.
- `*/*/*/**` : match everything at least three folders deep.
- `**/file.*|**/file` : match every file called "file" with any extension (or no extension).
- `x/*/y/*` : match all files that sit in folder x -> any folder -> folder y.
- `**/prefix*.csv` : match all csv files with specific prefix.
- `**/prefix*.parquet` : match all parquet files with specific prefix.

Let's look at a specific example, matching the following bucket layout:

Unset

```
myBucket

    -> log_files

    -> some_table_files

        -> part1.csv

        -> part2.csv

    -> images

    -> more_table_files

        -> part3.csv

    -> extras
```

```
        -> misc

            -> another_part1.csv
```

We want to pick up part1.csv, part2.csv and part3.csv (excluding another_part1.csv for now). We could do this a few different ways:

- We could pick up every csv file called "partX" with the single pattern `**/part*.csv`.
- To be a bit more robust, we could use the dual pattern `some_table_files/*.csv|more_table_files/*.csv` to pick up relevant files only from those exact folders.
- We could achieve the above in a single pattern by using the pattern `*table_files/*.csv`. This could however cause problems in the future if new unexpected folders started being created.
- We can also recursively wildcard, so adding the pattern `extras/**/*.csv` would pick up any csv files nested in folders below "extras", such as "extras/misc/another_part1.csv".

As you can probably tell, there are many ways to achieve the same goal with path patterns. We recommend using a pattern that ensures clarity and is robust against future additions to the directory structure.

User Schema

Providing a schema allows for more control over the output of this stream. Without a provided schema, columns and datatypes will be inferred from the first created file in the bucket matching your path pattern and suffix. This will probably be fine in most cases but there may be situations you want to enforce a schema instead, e.g.:

- You only care about a specific known subset of the columns. The other columns would all still be included, but packed into the `_ab_additional_properties` map.
- Your initial dataset is quite small (in terms of number of records), and you think the automatic type inference from this sample might not be representative of the data in the future.

- You want to purposely define types for every column.
- You know the names of columns that will be added to future data and want to include these in the core schema as columns rather than have them appear in the `_ab_additional_properties` map.

Or any other reason! The schema must be provided as valid JSON as a map of `{"column": "datatype"}` where each datatype is one of:

- string
- number
- integer
- object
- array
- boolean
- null

For example:

- {"id": "integer", "location": "string", "longitude": "number", "latitude": "number"}
- {"username": "string", "friends": "array", "information": "object"}

NOTE

Please note, the S3 Source connector used to infer schemas from all the available files and then merge them to create a superset schema. Starting from version 2.0.0 the schema inference works based on the first file found only. The first file we consider is the oldest one written to the prefix.

S3 Provider Settings

- `bucket` : name of the bucket your files are in
- `aws_access_key_id` : one half of the required credentials for accessing a private bucket.
- `aws_secret_access_key` : other half of the required credentials for accessing a private bucket.
- `path_prefix` : an optional string that limits the files returned by AWS when listing files to only that those starting with this prefix. This is different to path_pattern as it gets pushed down to the API call made to S3 rather than filtered in Airbyte and it does not accept pattern-style symbols (like wildcards `*`). We recommend using this if your bucket has many folders and files that are

unrelated to this stream and all the relevant files will always sit under this chosen prefix.

- Together with `path_pattern`, there are multiple ways to specify the files to sync. For example, all the following configs are equivalent:
  - `path_prefix` = `<empty>`, `path_pattern` = `path1/path2/myFolder/**/*`.
  - `path_prefix` = `path1/`, `path_pattern` = `path2/myFolder/**/*.csv`.
  - `path_prefix` = `path1/path2/` and `path_pattern` = `myFolder/**/*.csv`
  - `path_prefix` = `path1/path2/myFolder/`, `path_pattern` = `**/*.csv`. This is the most efficient one because the directories are filtered earlier in the S3 API call. However, the difference in efficiency is usually negligible.
- The rationale of having both `path_prefix` and `path_pattern` is to accommodate as many use cases as possible. If you found them confusing, feel free to ignore `path_prefix` and just set the `path_pattern`.

- `endpoint` : optional parameter that allow using of non Amazon S3 compatible services. Leave it blank for using default Amazon serivce.
- `use_ssl` : Allows using custom servers that configured to use plain http. Ignored in case of using Amazon service.
- `verify_ssl_cert` : Skip ssl validity check in case of using custom servers with self signed certificates. Ignored in case of using Amazon service.
  File Format Settings
  The Reader in charge of loading the file format is currently based on PyArrow (Apache Arrow).
  Note that all files within one stream must adhere to the same read options for every provided format.

CSV

Since CSV files are effectively plain text, providing specific reader options is often required for correct parsing of the files. These settings are applied when a CSV is created or exported so please ensure that this process happens consistently over time.

- `delimiter` : Even though CSV is an acronymn for Comma Separated Values, it is used more generally as a term for flat file data that may or may not be comma separated. The delimiter field lets you specify which character acts as the separator.
- `quote_char` : In some cases, data values may contain instances of reserved characters (like a comma, if that's the delimiter). CSVs can allow this behaviour by wrapping a value in defined quote characters so that on read it can parse it correctly.
- `escape_char` : An escape character can be used to prefix a reserved character and allow correct parsing.
- `encoding` : Some data may use a different character set (typically when different alphabets are involved). See the list of allowable encodings here.
- `double_quote` : Whether two quotes in a quoted CSV value denote a single quote in the data.
- `newlines_in_values` : Sometimes referred to as `multiline`. In most cases, newline characters signal the end of a row in a CSV, however text data may contain newline characters within it. Setting this to True allows correct parsing in this case.
- `block_size` : This is the number of bytes to process in memory at a time while reading files. The default value here is usually fine but if your table is particularly wide (lots of columns / data in fields is large) then raising this might solve failures on detecting schema. Since this defines how much data to read into memory, raising this too high could cause Out Of Memory issues so use with caution.
- `additional_reader_options` : This allows for editing the less commonly required CSV ConvertOptions. The value must be a valid JSON string, e.g.:

```
Unset
•
• {"timestamp_parsers": ["%m/%d/%Y %H:%M", "%Y/%m/%d
  %H:%M"], "strings_can_be_null": true, "null_values":
  ["NA", "NULL"]}
```

-
  `advanced_options` : This allows for editing the less commonly required CSV ReadOptions. The value must be a valid JSON string. One use case for this is when your CSV has no header, or you want to use custom column names, you can specify `column_names` using this option.

```
Unset
•
• {"column_names": ["column1", "column2", "column3"]}
```

Parquet

Apache Parquet file is a column-oriented data storage format of the Apache Hadoop ecosystem. It provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk. For now, the solution involves iterating through individual files at the abstract level thus partitioned parquet datasets are unsupported. The following settings are available:

- `buffer_size` : If positive, perform read buffering when deserializing individual column chunks. Otherwise IO calls are unbuffered.
- `columns` : If not None, only these columns will be read from the file.
- `batch_size` : Maximum number of records per batch. Batches may be smaller if there aren't enough rows in the file.

You can find details on here.

Avro

The avro parser uses fastavro. Currently, no additional options are supported.

Jsonl

The Jsonl parser uses pyarrow hence,only the line-delimited JSON format is supported.For more detailed info, please refer to the [docs] (https://arrow.apache.org/docs/python/generated/pyarrow.json.read_json.html)

Files

Google Sheets

This page guides you through the process of setting up the Google Sheets source connector.
INFO

The Google Sheets source connector pulls data from a single Google Sheets spreadsheet. To replicate multiple spreadsheets, set up multiple Google Sheets source connectors in your Airbyte instance.

Setup guide

For Airbyte Cloud:

To set up Google Sheets as a source in Airbyte Cloud:
1. Log into your Airbyte Cloud account.
2. In the left navigation bar, click Sources. In the top-right corner, click + New source.
3. On the Set up the source page, select Google Sheets from the Source type dropdown.
4. Enter a name for the Google Sheets connector.
5. Authenticate your Google account via OAuth or Service Account Key Authentication.

  - (Recommended) To authenticate your Google account via OAuth, click Sign in with Google and complete the authentication workflow.
  - To authenticate your Google account via Service Account Key Authentication, enter your Google Cloud service account key in JSON format. Make sure the Service Account has the Project Viewer permission. If your spreadsheet is viewable by anyone with its link, no further action is needed. If not, give your Service account access to your spreadsheet.
6. For Spreadsheet Link, enter the link to the Google spreadsheet. To get the link, go to the Google spreadsheet you want to sync, click Share in the top right corner, and click Copy Link.
7. For Row Batch Size, define the number of records you want the Google API to fetch at a time. The default value is 200.

For Airbyte Open Source:

To set up Google Sheets as a source in Airbyte Open Source:
1. Enable the Google Cloud Platform APIs for your personal or organization account.
   INFO
   The connector only finds the spreadsheet you want to replicate; it does not access any of your other files in Google Drive.
2. Go to the Airbyte UI and in the left navigation bar, click Sources. In the top-right corner, click + New source.
3. On the Set up the source page, select Google Sheets from the Source type dropdown.
4. Enter a name for the Google Sheets connector.
5. Authenticate your Google account via OAuth or Service Account Key Authentication:
   - To authenticate your Google account via OAuth, enter your Google application's client ID, client secret, and refresh token.
   - To authenticate your Google account via Service Account Key Authentication, enter your Google Cloud service account key in JSON format. Make sure the Service Account has the Project Viewer permission. If your spreadsheet is viewable by anyone with its link, no further action is needed. If not, give your Service account access to your spreadsheet.
6. For Spreadsheet Link, enter the link to the Google spreadsheet. To get the link, go to the Google spreadsheet you want to sync, click Share in the top right corner, and click Copy Link.

Output schema

Each sheet in the selected spreadsheet is synced as a separate stream. Each selected column in the sheet is synced as a string field.

Note: Sheet names and column headers must contain only alphanumeric characters or `_`, as specified in the Airbyte Protocol. For example, if your sheet or column header is named `the data`, rename it to `the_data`. This restriction does not apply to non-header cell values.

Airbyte only supports replicating Grid sheets.

Supported sync modes

The Google Sheets source connector supports the following sync modes:

- Full Refresh - Overwrite
- Full Refresh - Append

Data type mapping

Integration Type	Airbyte Type	Notes
any type	`string`	

Performance consideration

The Google API rate limit is 100 requests per 100 seconds per user and 500 requests per 100 seconds per project. Airbyte batches requests to the API in order to efficiently pull data and respects these rate limits. We recommended not using the same service user for more than 3 instances of the Google Sheets source connector to ensure high transfer speeds.

Files (CSV, JSON, Excel, Feather, Parquet)

This page contains the setup guide and reference information for the Files source connector.

Prerequisites

- URL to access the file
- Format
- Reader options
- Storage Providers

Setup guide

For Airbyte Cloud:

Setup through Airbyte Cloud will be exactly the same as the open-source setup, except for the fact that local files are disabled.

For Airbyte Open Source:

1. Once the File Source is selected, you should define both the storage provider along its URL and format of the file.
2. Depending on the provider choice and privacy of the data, you will have to configure more options.

Fields description

- For `Dataset Name` use the *name* of the final table to replicate this file into (should include letters, numbers dash and underscores only).
- For `File Format` use the *format* of the file which should be replicated (Warning: some formats may be experimental, please refer to the docs).
- For `Reader Options` use a *string in JSON* format. It depends on the chosen file format to provide additional options and tune its behavior. For example, `{}` for empty options, `{"sep": " "}` for set up separator to one space ' '.
- For `URL` use the *URL* path to access the file which should be replicated.
- For `Storage Provider` use the *storage Provider* or *Location* of the file(s) which should be replicated.
    - [Default] *Public Web*
        - `User-Agent` set to active if you want to add User-Agent to requests
    - *GCS: Google Cloud Storage*
        - `Service Account JSON` In order to access private Buckets stored on Google Cloud, this connector would need a service account json credentials with the proper permissions as described here. Please generate the credentials.json file and copy/paste its content to this field (expecting JSON formats). If accessing publicly available data, this field is not necessary.
    - *S3: Amazon Web Services*
        - `AWS Access Key ID` In order to access private Buckets stored on AWS S3, this connector would need credentials with the proper permissions. If accessing publicly available data, this field is not necessary.
        - `AWS Secret Access Key`In order to access private Buckets stored on AWS S3, this connector would need credentials with the proper permissions. If accessing publicly available data, this field is not necessary.
    - *AzBlob: Azure Blob Storage*
        - `Storage Account` The globally unique name of the storage account that the desired blob sits within. See here for more details.
        - `SAS Token` To access Azure Blob Storage, this connector would need credentials with the proper permissions. One option is a SAS (Shared Access Signature) token. If accessing publicly available data, this field is not necessary.

        - `Shared Key` To access Azure Blob Storage, this connector would need credentials with the proper permissions. One option is a storage account shared key (aka account key or access key). If accessing publicly available data, this field is not necessary.
    - *SSH: Secure Shell*
        - `User` use *username*.
        - `Password` use *password*.
        - `Host` use a *host*.
        - `Port` use a *port* for your host.
    - *SCP: Secure copy protocol*
        - `User` use *username*.
        - `Password` use *password*.
        - `Host` use a *host*.
        - `Port` use a *port* for your host.
    - *SFTP: Secure File Transfer Protocol*
        - `User` use *username*.
        - `Password` use *password*.
        - `Host` use a *host*.
        - `Port` use a *port* for your host.
    - *Local Filesystem (limited)*
        - `Storage` WARNING: Note that the local storage URL available for reading must start with the local mount "/local/" at the moment until we implement more advanced docker mounting options.

Provider Specific Information

- In case of Google Drive, it is necesary to use the Download URL, the format for that is `https://drive.google.com/uc?export=download&id=[DRIVE_FILE_ID]` where `[DRIVE_FILE_ID]` is the string found in the Share URL here `https://drive.google.com/file/d/[DRIVE_FILE_ID]/view?usp=sharing`
- In case of GCS, it is necessary to provide the content of the service account keyfile to access private buckets. See settings of BigQuery Destination
- In case of AWS S3, the pair of `aws_access_key_id` and `aws_secret_access_key` is necessary to access private S3 buckets.

- In case of AzBlob, it is necessary to provide the `storage_account` in which the blob you want to access resides. Either `sas_token` (info) or `shared_key` (info) is necessary to access private blobs.
- In case of a locally stored file on a Windows OS, it's necessary to change the values for `LOCAL_ROOT`, `LOCAL_DOCKER_MOUNT` and `HACK_LOCAL_ROOT_PARENT` in the `.env` file to an existing absolute path on your machine (colons in the path need to be replaced with a double forward slash, //). `LOCAL_ROOT` & `LOCAL_DOCKER_MOUNT` should be the same value, and `HACK_LOCAL_ROOT_PARENT` should be the parent directory of the other two.

Reader Options

The Reader in charge of loading the file format is currently based on Pandas IO Tools. It is possible to customize how to load the file into a Pandas DataFrame as part of this Source Connector. This is doable in the `reader_options` that should be in JSON format and depends on the chosen file format. See pandas' documentation, depending on the format:

For example, if the format `CSV` is selected, then options from the read_csv functions are available.

- It is therefore possible to customize the `delimiter` (or `sep`) to in case of tab separated files.
- Header line can be ignored with `header=0` and customized with `names`
- etc

We would therefore provide in the `reader_options` the following json:

```
Unset

{ "sep" : "\t", "header" : 0, "names": ["column1",
"column2"]}
```

In case you select `JSON` format, then options from the read_json reader are available.

For example, you can use the `{"orient" : "records"}` to change how orientation of data is loaded (if data is `[{column -> value}, … , {column -> value}]`)

If you need to read Excel Binary Workbook, please specify `excel_binary` format in `File Format` select.

```
Unset

:::warning
This connector does not support syncing unstructured data
files such as raw text, audio, or videos.
:::
```

Supported sync modes

Feature	Supported?
Full Refresh Sync	Yes
Incremental Sync	No
Replicate Incremental Deletes	No
Replicate Folders (multiple Files)	No
Replicate Glob Patterns (multiple Files)	No

```
Unset

:::info
```

```
This source produces a single table for the target file as
it replicates only one file at a time for the moment. Note
that you should provide the `dataset_name` which dictates
how the table will be identified in the destination (since
`URL` can be made of complex characters).
:::
```

File / Stream Compression

Compression	Supported?
Gzip	Yes
Zip	No
Bzip2	No
Lzma	No
Xz	No
Snappy	No

Storage Providers

Storage Providers	Supported?
HTTPS	Yes
Google Cloud Storage	Yes
Amazon Web Services	Yes

S3	
SFTP	Yes
SSH / SCP	Yes
local filesystem	Local use only (inaccessible for Airbyte Cloud)

File Formats

Format	Supported?
CSV	Yes
JSON	Yes
HTML	No
XML	No
Excel	Yes
Excel Binary Workbook	Yes
Feather	Yes
Parquet	Yes
Pickle	No
YAML	Yes

Changing data types of source columns

Normally, Airbyte tries to infer the data type from the source, but you can use `reader_options` to force specific data types. If you input `{"dtype":"string"}`, all columns will be forced to be parsed as strings. If you only want a specific column to be parsed as a string, simply use `{"dtype" : {"column name": "string"}}`.

Examples

Here are a list of examples of possible file inputs:

Dataset Name	Storage	URL	Reader Impl	Service Account	Description
epidemiology	HTTPS	https://storage.googleapis.com/covid19-open-data/v2/latest/epidemiology.csv			COVID-19 Public dataset on BigQuery
hr_and_financials	GCS	gs://airbyte-vault/financial.csv	smart_open or gcfs	{"type": "service_account", "private_key_id": "XXXXXXXX", ...}	data from a private bucket, a service account is necessary

landsat_index	GCS	gcp-public-data-landsat/index.csv.gz	smart_open		Using smart_open, we don't need to specify the compression (note the gs:// is optional too, same for other providers)

Examples with reader options:

Dataset Name	Storage	URL	Reader Impl	Reader Options	Description
landsat_index	GCS	gs://gcp-public-data-landsat/index.csv.gz	GCFS	{"compression": "gzip"}	Additional reader options to specify a compression option to `read_csv`
GDELT	S3	s3://gdelt-open-data/events/20190914.export.csv		{"sep": "\t", "header": null}	Here is TSV data separated by tabs without header row from AWS Open Data

server_logs	local	/local/logs.log		{"sep": ";"}	After making sure a local text file exists at `/tmp/airbyte_local/logs.log` with logs file from some server that are delimited by ';' delimiters

Example for SFTP:

Dataset Name	Storage	User	Password	Host	URL	Reader Options	Description
Test Rebext	SFTP	demo	password	test.rebext.net	/pub/example/readme.txt	{"sep": "\r\n", "header": null, "names": ["text"], "engine": "python"}	We use `python` engine for `read_csv` in order to handle delimiter of more than 1 character while providing our own column names.

Please see (or add) more at `airbyte-integrations/connectors/source-file/integration_tests/integration_source_test.py` for further usages examples.

Performance Considerations and Notes

In order to read large files from a remote location, this connector uses the smart_open library. However, it is possible to switch to either GCSFS or S3FS implementations as it is natively supported by the `pandas` library. This choice is made possible through the optional `reader_impl` parameter.

- Note that for local filesystem, the file probably have to be stored somewhere in the `/tmp/airbyte_local` folder with the same limitations as the CSV Destination so the `URL` should also starts with `/local/`.
- Please make sure that Docker Desktop has access to `/tmp` (and `/private` on a MacOS, as /tmp has a symlink that points to /private. It will not work otherwise). You allow it with "File sharing" in `Settings -> Resources -> File sharing -> add the one or two above folder` and hit the "Apply & restart" button.
- The JSON implementation needs to be tweaked in order to produce more complex catalog and is still in an experimental state: Simple JSON schemas should work at this point but may not be well handled when there are multiple layers of nesting.

SFTP

This page contains the setup guide and reference information for the SFTP source connector.

Prerequisites

- The Server with SFTP connection type support
- The Server host
- The Server port
- Username-Password/Public Key Access Rights

Setup guide

Step 1: Set up SFTP

1. Use your username/password credential to connect the server.
2. Alternatively generate Public Key Access

The following simple steps are required to set up public key authentication:

Key pair is created (typically by the user). This is typically done with ssh-keygen. Private key stays with the user (and only there), while the public key is sent to the server. Typically with the ssh-copy-id utility. Server stores the public key (and "marks" it as authorized). Server will now allow access to anyone who can prove they have the corresponding private key.

Step 2: Set up the SFTP connector in Airbyte

For Airbyte Cloud:

1. Log into your Airbyte Cloud account.
2. In the left navigation bar, click `Sources`. In the top-right corner, click +new source.
3. On the Set up the source page, enter the name for the SFTP connector and select SFTP from the Source type dropdown.
4. Enter your `User Name`, `Host Address`, `Port`
5. Choose the `Authentication` type `Password Authentication` or `Key Authentication`
6. Type `File type` (temporary comma separated)
7. Enter `Folder Path` (Optional) to specify server folder for sync
8. Enter `File Pattern` (Optional). e.g. `log-([0-9]{4})([0-9]{2})([0-9]{2})`. Write your own regex
9. Click on `Check Connection` to finish configuring the Amplitude source.

Supported sync modes

The SFTP source connector supports the following sync modes:

Feature	Support	Notes
Full Refresh - Overwrite	✅	Warning: this mode deletes all previously synced data in the configured bucket path.
Full Refresh - Append Sync	❌	
Incremental - Append	❌	

Incremental - Deduped History	❌	
Namespaces	❌	

Supported Streams

This source provides a single stream per file with a dynamic schema. The current supported type file: `.csv` and `.json` More formats (e.g. Apache Avro) will be supported in the future.

Smartsheets

This page guides you through the process of setting up the Smartsheets source connector.

Prerequisites

To configure the Smartsheet Source for syncs, you'll need the following:

- A Smartsheets API access token - generated by a Smartsheets user with at least read access
- The ID of the spreadsheet you'd like to sync

Step 1: Set up Smartsheets

Obtain a Smartsheets API access token

You can generate an API key for your account from a session of your Smartsheet webapp by clicking:

- Account (top-right icon)
- Apps & Integrations
- API Access
- Generate new access token

For questions on advanced authorization flows, refer to this.

Prepare the spreadsheet ID of your Smartsheet

You'll also need the ID of the Spreadsheet you'd like to sync. Unlike Google Sheets, this ID is not found in the URL. You can find the required spreadsheet ID from your Smartsheet app session by going to:

- File
- Properties

Step 2: Set up the Smartsheets connector in Airbyte

For Airbyte Cloud:

1. Log into your Airbyte Cloud account.
2. In the left navigation bar, click Sources. In the top-right corner, click +new source.
3. On the Set up the source page, enter the name for the Smartsheets connector and select Smartsheets from the Source type dropdown.
4. Authenticate via OAuth2.0 using the API access token from Prerequisites
5. Enter the start date and the ID of the spreadsheet you want to sync
6. Submit the form

For Airbyte Open Source:

1. Navigate to the Airbyte Open Source dashboard
2. Set the name for your source
3. Enter the API access token from Prerequisites
4. Enter the ID of the spreadsheet you want to sync
5. Enter a start sync date
6. Click Set up source

Supported sync modes

The Smartsheets source connector supports the following sync modes:

- Full Refresh | Overwrite
- Full Refresh | Append
- Incremental | Append
- Incremental | Deduped

Performance considerations

At the time of writing, the Smartsheets API rate limit is 300 requests per minute per API access token.

Supported streams

This source provides a single stream per spreadsheet with a dynamic schema, depending on your spreadsheet structure. For example, having a spreadsheet `Customers`, the connector would introduce a stream with the same name and properties typed according to Data type map (see below).

Important highlights

The Smartsheet Source is written to pull data from a single Smartsheet spreadsheet. Unlike Google Sheets, Smartsheets only allows one sheet per Smartsheet - so a given Airbyte connector instance can sync only one sheet at a time. To replicate multiple spreadsheets, you can create multiple instances of the Smartsheet Source in Airbyte, reusing the API token for all your sheets that you need to sync.

Note: Column headers must contain only alphanumeric characters or `_` , as specified in the Airbyte Protocol.

Data type map

The data type mapping adopted by this connector is based on the Smartsheet documentation.

NOTE: For any column datatypes interpreted by Smartsheets beside `DATE` and `DATETIME`, this connector's source schema generation assumes a `string` type, in which case the `format` field is not required by Airbyte.

Integration Type	Airbyte Type	Airbyte Format
`TEXT_NUMBER`	`string`	
`DATE`	`string`	`format: date`
`DATETIME`	`string`	`format: date-time`
`anything else`	`string`	

The remaining column datatypes supported by Smartsheets are more complex types (e.g. Predecessor, Dropdown List) and are not supported by this connector beyond its `string` representation.

Appendix

Change Data Capture (CDC)

What is log-based incremental replication?

Many common databases support writing all record changes to log files for the purpose of replication. A consumer of these log files (such as Airbyte) can read these logs while keeping track of the current position within the logs in order to read all record changes coming from `DELETE`/`INSERT`/`UPDATE` statements.

Syncing

The orchestration for syncing is similar to non-CDC database sources. After selecting a sync interval, syncs are launched regularly. We read data from the log up to the time that the sync was started. We do not treat CDC sources as infinite streaming sources.

You should ensure that your schedule for running these syncs is frequent enough to consume the logs that are generated. The first time the sync is run, a snapshot of the current state of the data will be taken. This is done using `SELECT` statements and is effectively a Full Refresh. Subsequent syncs will use the logs to determine which changes took place since the last sync and update those. Airbyte keeps track of the current log position between syncs.

A single sync might have some tables configured for Full Refresh replication and others for Incremental. If CDC is configured at the source level, all tables with Incremental selected will use CDC. All Full Refresh tables will replicate using the same process as non-CDC sources. However, these tables will still include CDC metadata columns by default.

The Airbyte Protocol outputs records from sources. Records from `UPDATE` statements appear the same way as records from `INSERT` statements. We support different options for how to sync this data into destinations using primary keys, so you can choose to append this data, delete in place, etc.

We add some metadata columns for CDC sources:

- `ab_cdc_lsn` (postgres and sql server sources) is the point in the log where the record was retrieved
- `ab_cdc_log_file` & `ab_cdc_log_pos` (specific to mysql source) is the file name and position in the file where the record was retrieved
- `ab_cdc_updated_at` is the timestamp for the database transaction that resulted in this record change and is present for records from `DELETE`/`INSERT`/`UPDATE` statements
- `ab_cdc_deleted_at` is the timestamp for the database transaction that resulted in this record change and is only present for records from `DELETE` statements

Limitations

- CDC incremental is only supported for tables with primary keys. A CDC source can still choose to replicate tables without primary keys as Full Refresh or a non-CDC source can be configured for the same database to replicate the tables without primary keys using standard incremental replication.
- Data must be in tables, not views.

- The modifications you are trying to capture must be made using `DELETE`/`INSERT`/`UPDATE`. For example, changes made from `TRUNCATE`/`ALTER` won't appear in logs and therefore in your destination.
- We do not support schema changes automatically for CDC sources. We recommend resetting and resyncing data if you make a schema change.
- There are database-specific limitations. See the documentation pages for individual connectors for more information.
- The records produced by `DELETE` statements only contain primary keys. All other data fields are unset.

Current Support

- Postgres (For a quick video overview of CDC on Postgres, click here)
- MySQL
- Microsoft SQL Server / MSSQL

Coming Soon

- Oracle DB
- Please create a ticket if you need CDC support on another database!

Configuring the Airbyte Database

Airbyte uses different objects to store internal state and metadata. This data is stored and manipulated by the various Airbyte components, but you have the ability to manage the deployment of this database in the following two ways:

- Using the default Postgres database that Airbyte spins-up as part of the Docker service described in the `docker-compose.yml` file: `airbyte/db`.
- Through a dedicated custom Postgres instance (the `airbyte/db` is in this case unused, and can therefore be removed or de-activated from the `docker-compose.yml` file). It's not a good practice to deploy mission-critical databases on Docker or Kubernetes. Using a dedicated instance will provide more reliability to your Airbyte deployment. Moreover, using a Cloud-managed Postgres instance (such as AWS RDS our GCP Cloud SQL), you will benefit from automatic backup and fine-grained sizing. You can start with a pretty small

instance, but according to your Airbyte usage, the job database might grow and require more storage if you are not truncating the job history.

The various entities are persisted in two internal databases:

- Job database
    - Data about executions of Airbyte Jobs and various runtime metadata.
    - Data about the internal orchestrator used by Airbyte, Temporal.io (Tasks, Workflow data, Events, and visibility data).
- Config database
    - Connectors, Sync Connections and various Airbyte configuration objects.

Note that no actual data from the source (or destination) connectors ever transits or is retained in this internal database.

If you need to interact with it, for example, to make back-ups or perform some clean-up maintenances, you can also gain access to the Export and Import functionalities of this database via the API or the UI (in the Admin page, in the Configuration Tab).

Connecting to an External Postgres database

Let's walk through what is required to use a Postgres instance that is not managed by Airbyte. First, for the sake of the tutorial, we will run a new instance of Postgres in its own docker container with the command below. If you already have Postgres running elsewhere, you can skip this step and use the credentials for that in future steps.

```
Unset


docker run --rm --name airbyte-postgres -e
POSTGRES_PASSWORD=password -p 3000:5432 -d postgres
```

In order to configure Airbyte services with this new database, we need to edit the following environment variables declared in the `.env` file (used by the docker-compose command afterward):

```
Unset
```

```
DATABASE_USER=postgres

DATABASE_PASSWORD=password

DATABASE_HOST=host.docker.internal # refers to localhost of
host

DATABASE_PORT=3000

DATABASE_DB=postgres
```

By default, the Config Database and the Job Database use the same database instance based on the above setting. It is possible, however, to separate the former from the latter by specifying a separate parameters. For example:

```
Unset


CONFIG_DATABASE_USER=airbyte_config_db_user

CONFIG_DATABASE_PASSWORD=password
```

Additionally, you must redefine the JDBC URL constructed in the environment variable `DATABASE_URL` to include the correct host, port, and database. If you need to provide extra arguments to the JDBC driver (for example, to handle SSL) you should add it here as well:

```
Unset


DATABASE_URL=jdbc:postgresql://host.docker.internal:3000/po
stgres?ssl=true&sslmode=require
```

Same for the config database if it is separate from the job database:

```
Unset

CONFIG_DATABASE_URL=jdbc:postgresql://<host>:<port>/<databa
se>?<extra-parameters>
```

Initializing the database

INFO

This step is only required when you setup Airbyte with a custom database for the first time.

If you provide an empty database to Airbyte and start Airbyte up for the first time, the server will automatically create the relevant tables in your database, and copy the data. Please make sure:

- The database exists in the server.
- The user has both read and write permissions to the database.
- The database is empty.
  - If the database is not empty, and has a table that shares the same name as one of the Airbyte tables, the server will assume that the database has been initialized, and will not copy the data over, resulting in server failure. If you run into this issue, just wipe out the database, and launch the server again.

Accessing the default database located in docker airbyte-db

In extraordinary circumstances while using the default `airbyte-db` Postgres database, if a developer wants to access the data that tracks jobs, they can do so with the following instructions.

As we've seen previously, the credentials for the database are specified in the `.env` file that is used to run Airbyte. By default, the values are:

```
Unset
```

```
DATABASE_USER=docker

DATABASE_PASSWORD=docker

DATABASE_DB=airbyte
```

If you have overridden these defaults, you will need to substitute them in the instructions below.

The following command will allow you to access the database instance using `psql`.

```
Unset


docker exec -ti airbyte-db psql -U docker -d airbyte
```

Following tables are created

1. `workspace` : Contains workspace information such as name, notification configuration, etc.
2. `actor_definition` : Contains the source and destination connector definitions.
3. `actor` : Contains source and destination connectors information.
4. `actor_oauth_parameter` : Contains source and destination oauth parameters.
5. `operation` : Contains dbt and custom normalization operations.
6. `connection` : Contains connection configuration such as catalog details, source, destination, etc.
7. `connection_operation` : Contains the operations configured for a given connection.
8. `state`. Contains the last saved state for a connection.

Data Types in Records

AirbyteRecords are required to conform to the Airbyte type system. This means that all sources must produce schemas and records within these types, and all destinations must handle records that conform to this type system.

Because Airbyte's interfaces are JSON-based, this type system is realized using JSON schemas. In order to work around some limitations of JSON schemas, we add an additional `airbyte_type` parameter to define more narrow types.

This type system does not constrain values. However, destinations may not fully support all values - for example, Avro-based destinations may reject numeric values outside the standard 64-bit representations, or databases may reject timestamps in the BC era.

The types

This table summarizes the available types. See the Specific Types section for an explanation of optional parameters.

Airbyte type	JSON Schema	Examples
String	`{"type": "string""}`	`"foo bar"`
Boolean	`{"type": "boolean"}`	`true` or `false`
Date	`{"type": "string", "format": "date"}`	`"2021-01-23"`, `"2021-01-23 BC"`
Timestamp with timezone	`{"type": "string", "format": "date-time", "airbyte_type": "timestamp_with_timezone"}`	`"2022-11-22T01:23:45.123456+05:00"`, `"2022-11-22T01:23:45Z BC"`
Timestamp without timezone	`{"type": "string", "format": "date-time", "airbyte_type": "timestamp_without_timezone"}`	`"2022-11-22T01:23:45"`, `"2022-11-22T01:23:45.123456 BC"`

Time without timezone	`{"type": "string", "airbyte_type": "time_with_timezone"}`	`"01:23:45.123456"`, `"01:23:45"`
Time with timezone	`{"type": "string", "airbyte_type": "time_without_timezone"}`	`"01:23:45.123456+05:00"`, `"01:23:45Z"`
Integer	`{"type": "integer"}` or `{"type": "number", "airbyte_type": "integer"}`	`42`
Number	`{"type": "number"}`	`1234.56`
Array	`{"type": "array"}`; optionally `items`	`[1, 2, 3]`
Object	`{"type": "object"}`; optionally `properties`	`{"foo": "bar"}`
Union	`{"oneOf": [...]}`	

Record structure

As a reminder, sources expose a `discover` command, which returns a list of `AirbyteStreams`, and a `read` method, which emits a series of `AirbyteRecordMessages`. The type system determines what a valid `json_schema` is for an `AirbyteStream`, which in turn dictates what messages `read` is allowed to emit.

For example, a source could produce this `AirbyteStream` (remember that the `json_schema` must declare `"type": "object"` at the top level):

Unset

```
{
 "name": "users",
 "json_schema": {
   "type": "object",
   "properties": {
     "username": {
       "type": "string"
     },
     "age": {
       "type": "integer"
     },
     "appointments": {
       "type": "array",
       "items": {
         "type": "string",
         "format": "date-time",
         "airbyte_type": "timestampt_with_timezone"
       }
     }
   }
 }
}
```

Along with this `AirbyteRecordMessage` (observe that the `data` field conforms to the `json_schema` from the stream):

```
Unset


{
 "stream": "users",
 "data": {
   "username": "someone42",
   "age": 84,
   "appointments": ["2021-11-22T01:23:45+00:00",
"2022-01-22T14:00:00+00:00"]
 },
 "emitted_at": 1623861660
}
```

The top-level `object` must conform to the type system. This means that all of the fields must also conform to the type system.

Nulls

Many sources cannot guarantee that all fields are present on all records. In these cases, sources should simply not list them as `required` fields. In most cases, sources do not need to list fields as required; by default, all fields are treated as nullable.

Unsupported types

Destinations must have handling for all types, but they are free to cast types to a convenient representation. For example, let's say a source discovers a stream with this schema:

```
Unset

{
 "type": "object",
 "properties": {
   "appointments": {
     "type": "array",
     "items": {
       "type": "string",
       "format": "date-time",
       "airbyte_type": "timestamp_with_timezone"
     }
   }
 }
}
```

Along with records which contain data that looks like this:

```
Unset

{"appointments": ["2021-11-22T01:23:45+00:00",
"2022-01-22T14:00:00+00:00"]}
```

The user then connects this source to a destination that cannot natively handle `array` fields. The destination connector is free to simply JSON-serialize the array back to a string when pushing data into the end platform. In other words, the destination connector could behave as though the source declared this schema:

Unset

```
{
 "type": "object",
 "properties": {
   "appointments": {
     "type": "string"
   }
 }
}
```

And emitted this record:

Unset

```
{"appointments": "[\"2021-11-22T01:23:45+00:00\",
\"2022-01-22T14:00:00+00:00\"]"}
```

Of course, destinations are free to choose the most convenient/reasonable representation for any given value. JSON serialization is just one possible strategy. For example, many SQL destinations will fall back to a native JSON type (e.g. Postgres' JSONB type, or Snowflake's VARIANT).

Specific types

These sections explain how each specific type should be used.

Boolean

Boolean values are represented as native JSON booleans (i.e. `true` or `false`, case-sensitive). Note that "truthy" and "falsy" values are *not* acceptable: `"true"`, `"false"`, `1`, and `0` are not valid booleans.

Dates and timestamps

Airbyte has five temporal types: `date`, `timestamp_with_timezone`, `timestamp_without_timezone`, `time_with_timezone`, and `time_without_timezone`. These are represented as strings with specific `format` (either `date` or `date-time`).

However, JSON schema does not have a built-in way to indicate whether a field includes timezone information. For example, given this JsonSchema:

```
Unset

{
 "type": "object",
 "properties": {
   "created_at": {
     "type": "string",
     "format": "date-time"
   }
 }
}
```

Both `{"created_at": "2021-11-22T01:23:45+00:00"}` and `{"created_at": "2021-11-22T01:23:45"}` are valid records.

The `airbyte_type` field resolves this ambiguity; sources producing timestamp-ish fields should choose either `timestamp_with_timezone` or `timestamp_without_timezone` (or time with/without timezone).

Many sources (which were written before this system was formalized) do not specify the timezone-ness of their fields. Destinations should default to using the `with_timezone` variant in these cases.

All of these must be represented as RFC 3339§5.6 strings, extended with BC era support. See the type definition descriptions for specifics.

Numeric values

The number and integer types can accept any value, without constraint on range. However, this is still subject to compatibility with the destination: the destination (or normalization) *may* throw an error if it attempts to write a value outside the range supported by the destination warehouse / storage medium.

Airbyte does not currently support infinity/NaN values.

Arrays

Arrays contain 0 or more items, which must have a defined type. These types should also conform to the type system. Arrays may require that all of their elements be the same type (`"items": {whatever type...}`). They may instead require each element to conform to one of a list of types (`"items": [{first type...}, {second type...}, ... , {Nth type...}]`).

Note that Airbyte's usage of the `items` field is slightly different than JSON schema's usage, in which an `"items": [...]` actually constrains the element correpsonding to the index of that item (AKA tuple-typing). This is becase destinations may have a difficult time supporting tuple-typed arrays without very specific handling, and as such are permitted to somewhat loosen their requirements.

Objects

As with arrays, objects may declare `properties`, each of which should have a type which conforms to the type system.

Unions

Sources may want to mix different types in a single field, e.g. `"type": ["string", "object"]`. Destinations must handle this case, either using a native union type, or by finding a native type that can accept all of the source's types (this frequently will be `string` or `json`).

In some cases, sources may want to use multiple types for the same field. For example, a user might have a property which holds one of two object schemas. This is supported with JSON schema's `oneOf` type. Note that many destinations do not currently support these types, and may not behave as expected.

Connection sync modes

A sync mode governs how Airbyte reads from a source and writes to a destination. Airbyte provides different sync modes to account for various use cases.

- Full Refresh | Overwrite: Sync all records from the source and replace data in destination by overwriting it.
- Full Refresh | Append: Sync all records from the source and add them to the destination without deleting any data.
- Incremental Sync | Append: Sync new records from the source and add them to the destination without deleting any data.

- Incremental Sync | Deduped History: Sync new records from the source and add them to the destination. Also provides a de-duplicated view mirroring the state of the stream in the source.

Connections and Sync Modes

A connection is a configuration for syncing data between a source and a destination. To setup a connection, a user must configure things such as:

- Sync schedule: when to trigger a sync of the data.
- Destination Namespace and stream names: where the data will end up being written.
- A catalog selection: which streams and fields to replicate from the source
- Sync mode: how streams should be replicated (read and write):
- Optional transformations: how to convert Airbyte protocol messages (raw JSON blob) data into some other data representations.

Sync schedules

Sync schedules are explained below. For information about catalog selections, see AirbyteCatalog & ConfiguredAirbyteCatalog.

Syncs will be triggered by either:

- A manual request (i.e: clicking the "Sync Now" button in the UI)
- A schedule

When a scheduled connection is first created, a sync is executed as soon as possible. After that, a sync is run once the time since the last sync (whether it was triggered manually or due to a schedule) has exceeded the schedule interval. For example, consider the following illustrative scenario:

- October 1st, 2pm, a user sets up a connection to sync data every 24 hours.
- October 1st, 2:01pm: sync job runs
- October 2nd, 2:01pm: 24 hours have passed since the last sync, so a sync is triggered.
- October 2nd, 5pm: The user manually triggers a sync from the UI
- October 3rd, 2:01pm: since the last sync was less than 24 hours ago, no sync is run

- October 3rd, 5:01pm: It has been more than 24 hours since the last sync, so a sync is run

Destination namespace

The location of where a connection replication will store data is referenced as the destination namespace. The destination connectors should create and write records (for both raw and normalized tables) in the specified namespace which should be configurable in the UI via the Namespace Configuration field (or NamespaceDefinition in the API). You can read more about configuring namespaces here.

Destination stream name

Prefix stream name

Stream names refer to table names in a typical RDBMS. But it can also be the name of an API endpoint, etc. Similarly to the namespace, stream names can be configured to diverge from their names in the source with a "prefix" field. The prefix is prepended to the source stream name in the destination.

Stream-specific customization

All the customization of namespace and stream names described above will be equally applied to all streams selected for replication in a catalog per connection. If you need more granular customization, stream by stream, for example, or with different logic rules, then you could follow the tutorial on customizing transformations with dbt.

Sync modes

A sync mode governs how Airbyte reads from a source and writes to a destination. Airbyte provides different sync modes to account for various use cases. To minimize confusion, a mode's behavior is reflected in its name. The easiest way to understand Airbyte's sync modes is to understand how the modes are named.

1. The first part of the name denotes how the source connector reads data from the source:
    i. Incremental: Read records added to the source since the last sync job. (The first sync using Incremental is equivalent to a Full Refresh)

        - Method 1: Using a cursor. Generally supported by all connectors whose data source allows extracting records incrementally.
        - Method 2: Using change data capture. Only supported by some sources. See CDC for more info.
    ii. Full Refresh: Read everything in the source.
2. The second part of the sync mode name denotes how the destination connector writes data. This is not affected by how the source connector produced the data:
    i. Overwrite: Overwrite by first deleting existing data in the destination.
    ii. Append: Write by adding data to existing tables in the destination.
    iii. Deduped History: Write by first adding data to existing tables in the destination to keep a history of changes. The final table is produced by de-duplicating the intermediate ones using a primary key.

A sync mode is therefore, a combination of a source and destination mode together. The UI exposes the following options, whenever both source and destination connectors are capable to support it for the corresponding stream:

- Full Refresh Overwrite: Sync the whole stream and replace data in destination by overwriting it.
- Full Refresh Append: Sync the whole stream and append data in destination.
- Incremental Append: Sync new records from stream and append data in destination.
- Incremental Deduped History: Sync new records from stream and append data in destination, also provides a de-duplicated view mirroring the state of the stream in the source.

Optional operations

Airbyte basic normalization

As described by the Airbyte Protocol from the Airbyte Specifications, replication is composed of source connectors that are transmitting data in a JSON format. It is then written as such by the destination connectors.

On top of this replication, Airbyte provides the option to enable or disable an additional transformation step at the end of the sync called basic normalization. This operation is:

- Only available for destinations that support dbt execution
- Automatically generates a pipeline or DAG of dbt transformation models to convert JSON blob objects into normalized tables

- Runs and applies these dbt models to the data written in the destination

NOTE

Normalizing data may cause an increase in your destination's compute cost. This cost will vary depending on the amount of data that is normalized and is not related to Airbyte credit usage.

Custom sync operations

Further operations can be included in a sync on top of Airbyte basic normalization (or even to replace it completely). See operations for more details.

Full Refresh - Overwrite

Overview

The Full Refresh modes are the simplest methods that Airbyte uses to sync data, as they always retrieve all available information requested from the source, regardless of whether it has been synced before. This contrasts with Incremental sync, which does not sync data that has already been synced before.

In the Overwrite variant, new syncs will destroy all data in the existing destination table and then pull the new data in. Therefore, data that has been removed from the source after an old sync will be deleted in the destination table.

Example Behavior

On the nth sync of a full refresh connection:

Replace existing data with new data. The connection does not create any new tables.

data in the destination *before* the sync:

Languages

Python
Java

new data:

Languages
Python
Java
Ruby

data in the destination *after* the sync:

Languages
Python
Java
Ruby

Note: This is how Singer target-bigquery does it.

In the future

We will consider making other flavors of full refresh configurable as first-class citizens in Airbyte. e.g. On new data, copy old data to a new table with a timestamp, and then replace the original table with the new data. As always, we will focus on adding these options in such a way that the behavior of each connector is both well documented and predictable.

Full Refresh - Append

Overview

The Full Refresh modes are the simplest methods that Airbyte uses to sync data, as they always retrieve all available data requested from the source, regardless of whether it has been synced before. This contrasts with Incremental sync, which does not sync data that has already been synced before.

In the Append variant, new syncs will take all data from the sync and append it to the destination table. Therefore, if syncing similar information multiple times, every sync will create duplicates of already existing data.

Example Behavior

On the nth sync of a full refresh connection:

Add new data to the same table. Do not touch existing data.

data in the destination *before* the nth sync:

Languages
Python
Java

new data:

Languages
Python
Java

Ruby

data in the destination *after* the nth sync:

Languages
Python
Java
Python
Java
Ruby

This could be useful when we are interested to know about deletion of data in the source. This is possible if we also consider the date, or the batch id from which the data was written to the destination:

new data at the n+1th sync:

Languages
Python
Ruby

data in the destination *after* the n+1th sync:

Languages	batch id
Python	1
Java	1
Python	2

Java	2
Ruby	2
Python	3
Ruby	3

In the future

We will consider making a better detection of deletions in the source, especially with `Incremental`, and `Change Data Capture` based sync modes for example.

Incremental Sync - Append

Overview

Airbyte supports syncing data in Incremental Append mode i.e: syncing only replicate *new* or *modified* data. This prevents re-fetching data that you have already replicated from a source. If the sync is running for the first time, it is equivalent to a Full Refresh since all data will be considered as *new*.

In this flavor of incremental, records in the warehouse destination will never be deleted or mutated. A copy of each new or updated record is *appended* to the data in the warehouse. This means you can find multiple copies of the same record in the destination warehouse. We provide an "at least once" guarantee of replicating each record that is present when the sync runs.

Definitions

A `cursor` is the value used to track whether a record should be replicated in an incremental sync. A common example of a `cursor` would be a timestamp from an `updated_at` column in a database table.

A `cursor field` is the *field* or *column* in the data where that cursor can be found. Extending the above example, the `updated_at` column in the database would be the `cursor field`, while the `cursor` is the actual timestamp *value* used to determine if a record should be replicated.

We will refer to the set of records that the source identifies as being new or updated as a `delta`.

Rules

As mentioned above, the delta from a sync will be *appended* to the existing data in the data warehouse. Incremental will never delete or mutate existing records. Let's walk through a few examples.

Newly Created Record

Assume that `updated_at` is our `cursor_field`. Let's say the following data already exists into our data warehouse.

name	deceased	updated_at
Louis XVI	false	1754
Marie Antoinette	false	1755

In the next sync, the delta contains the following record:

name	deceased	updated_at
Louis XVII	false	1785

At the end of this incremental sync, the data warehouse would now contain:

name	deceased	updated_at
Louis XVI	false	1754
Marie Antoinette	false	1755
Louis XVII	false	1785

Updating a Record

Let's assume that our warehouse contains all the data that it did at the end of the previous section. Now, unfortunately the king and queen lose their heads. Let's see that delta:

name	deceased	updated_at
Louis XVI	true	1793
Marie Antoinette	true	1793

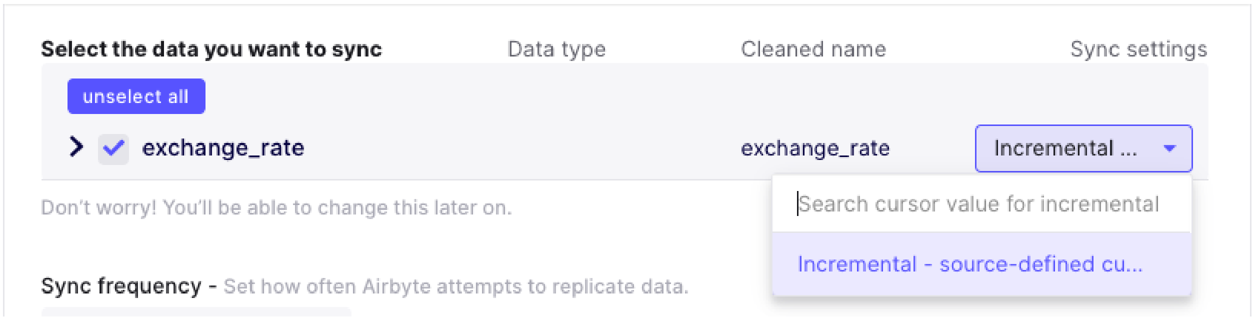
The output we expect to see in the warehouse is as follows:

name	deceased	updated_at
Louis XVI	false	1754
Marie Antoinette	false	1755
Louis XVII	false	1785
Louis XVI	true	1793

Marie Antoinette	true	1793

Source-Defined Cursor

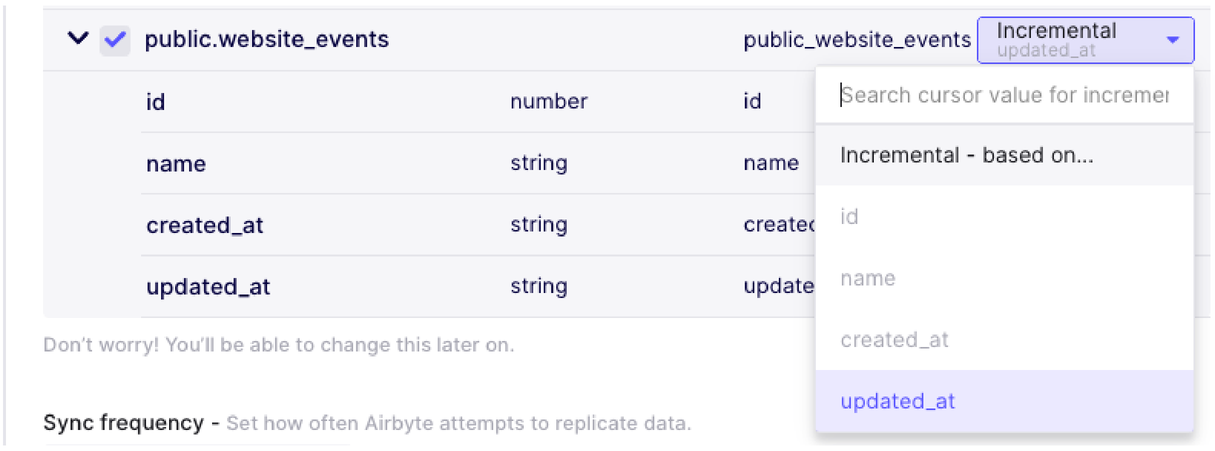
Some sources are able to determine the cursor that they use without any user input. For example, in the exchange rates source, the source knows that the date field should be used to determine the last record that was synced. In these cases, simply select the incremental option in the UI.


Select the data you want to sync
Data type
Cleaned name
Sync settings
unselect all
exchange_rate
exchange_rate
Incremental ...
Search cursor value for incremental
Incremental - source-defined cu...
Don't worry! You'll be able to change this later on.
Sync frequency - Set how often Airbyte attempts to replicate data.

(You can find a more technical details about the configuration data model here).

User-Defined Cursor

Some sources cannot define the cursor without user input. For example, in the postgres source, the user needs to choose which column in a database table they want to use as the `cursor field`. In these cases, select the column in the sync settings dropdown that should be used as the `cursor field`.



public.website_events		public_website_events	Incremental updated_at
id	number	id	
name	string	name	
created_at	string	created_at	
updated_at	string	updated_at	

Don't worry! You'll be able to change this later on.

Sync frequency - Set how often Airbyte attempts to replicate data.

(You can find a more technical details about the configuration data model here).

Getting the Latest Snapshot of data

As demonstrated in the examples above, with Incremental Append, a record which was updated in the source will be appended to the destination rather than updated in-place. This means that if data in the source uses a primary key (e.g: `user_id` in the `users` table), then the destination will end up having multiple records with the same primary key value.

However, some use cases require only the latest snapshot of the data. This is available by using other flavors of sync modes such as Incremental - Deduped History instead.

Note that in Incremental Append, the size of the data in your warehouse increases monotonically since an updated record in the source is appended to the destination rather than updated in-place.

If you only care about having the latest snapshot of your data, you may want to look at other sync modes that will keep smaller copies of the replicated data or you can periodically run cleanup jobs which retain only the latest instance of each record.

Inclusive Cursors

When replicating data incrementally, Airbyte provides an at-least-once delivery guarantee. This means that it is acceptable for sources to re-send some data when ran incrementally. One case where this is particularly relevant is when a source's cursor is not very granular. For example, if a cursor field has the granularity of a day (but not hours, seconds, etc), then if that source is run twice in the same day, there is no way for the source to know which records that are that date were already replicated earlier that day. By convention, sources should prefer resending data if the cursor field is ambiguous.

Additionally, you may run into behavior where you see the same row being emitted during each sync. This will occur if your data has not changed and you attempt to run additional syncs, as the cursor field will always be greater than or equal to itself, causing it to pull the latest row multiple times until there is new data at the source.

Known Limitations

Due to the use of a cursor column, if modifications to the underlying records are made without properly updating the cursor field, then the updated records won't be picked up by the Incremental sync as expected since the source connectors extract delta rows using a SQL query looking like:

```
Unset


SELECT * FROM table WHERE cursor_field >=
'last_sync_max_cursor_field_value'
```

Let's say the following data already exists into our data warehouse.

name	deceased	updated_at
Louis XVI	false	1754
Marie Antoinette	false	1755

At the start of the next sync, the source data contains the following new record:

name	deceased	updated_at
Louis XVI	true	1754

At the end of the second incremental sync, the data warehouse would still contain data from the first sync because the delta record did not provide a valid value for the cursor field (the cursor field is not greater than last sync's max value, `1754 < 1755`), so it is not emitted by the source as a new or modified record.

name	deceased	updated_at
Louis XVI	false	1754
Marie Antoinette	false	1755

Similarly, if multiple modifications are made during the same day to the same records. If the frequency of the sync is not granular enough (for example, set for every 24h), then intermediate modifications to the data are not going to be detected and emitted. Only the state of data at the time the sync runs will be reflected in the destination.

Those concerns could be solved by using a different incremental approach based on binary logs, Write-Ahead-Logs (WAL), or also called Change Data Capture (CDC).

The current behavior of Incremental is not able to handle source schema changes yet, for example, when a column is added, renamed or deleted from an existing table etc. It is recommended to trigger a Full refresh - Overwrite to correctly replicate the data to the destination with the new schema changes.

If you are not satisfied with how transformations are applied on top of the appended data, you can find more relevant SQL transformations you might need to do on your data in the Connecting EL with T using SQL (part 1/2)

Incremental Sync - Deduped History

High-Level Context

This connector syncs data incrementally, which means that only new or modified data will be synced. In contrast with the Incremental Append mode, this mode updates rows that have been modified instead of adding a new version of the row with the updated data. Simply put, if you've synced a row before and it has since been updated, this mode will combine the two rows in the destination and use the updated data. On the other hand, the Incremental Append mode would just add a new row with the updated data.

Overview

Airbyte supports syncing data in Incremental Deduped History mode i.e:
1. Incremental means syncing only replicate *new* or *modified* data. This prevents re-fetching data that you have already replicated from a source. If the sync is running for the first time, it is equivalent to a Full Refresh since all data will be considered as *new*.
2. Deduped means that data in the final table will be unique per primary key (unlike Append modes). This is determined by sorting the data using the cursor field and keeping only the latest de-duplicated data row. In dimensional data warehouse jargon defined by Ralph Kimball, this is referred as a Slowly Changing Dimension (SCD) table of type 1.
3. History means that an additional intermediate table is created in which data is being continuously appended to (with duplicates exactly like Append modes). With the use of primary key fields, it is identifying effective `start` and `end` dates of each row of a record. In dimensional data warehouse jargon, this is referred as a Slowly Changing Dimension (SCD) table of type 2.

In this flavor of incremental, records in the warehouse destination will never be deleted in the history tables (named with a `_scd` suffix), but might not exist in the final table. A copy of each new or updated record is *appended* to the history data in the warehouse. Only the `end` date column is mutated when a new version of the same record is inserted to denote effective date ranges of a row. This means you can find multiple copies of the same record in the destination warehouse. We provide an "at least once" guarantee of replicating each record that is present when the sync runs.

On the other hand, records in the final destination can potentially be deleted as they are de-duplicated. You should not find multiple copies of the same primary key as these should be unique in that table.

Definitions

A `cursor` is the value used to track whether a record should be replicated in an incremental sync. A common example of a `cursor` would be a timestamp from an `updated_at` column in a database table.

A `cursor field` is the *field* or *column* in the data where that cursor can be found. Extending the above example, the `updated_at` column in the database would be the `cursor field`, while the `cursor` is the actual timestamp *value* used to determine if a record should be replicated.

We will refer to the set of records that the source identifies as being new or updated as a `delta`.

A `primary key` is one or multiple (called `composite primary keys`) *fields* or *columns* that is used to identify the unique entities of a table. Only one row per primary key value is permitted in a database table. In the data warehouse, just like in incremental - Append, multiple rows for the same primary key can be found in the history table. The unique records per primary key behavior is mirrored in the final table with incremental deduped sync mode. The primary key is then used to refer to the entity which values should be updated.

Rules

As mentioned above, the delta from a sync will be *appended* to the existing history data in the data warehouse. In addition, it will update the associated record in the final table. Let's walk through a few examples.

Newly Created Record

Assume that `updated_at` is our `cursor_field` and `name` is the `primary_key`. Let's say the following data already exists into our data warehouse.

name	deceased	updated_at
Louis XVI	false	1754
Marie Antoinette	false	1755

In the next sync, the delta contains the following record:

name	deceased	updated_at
Louis XVII	false	1785

At the end of this incremental sync, the data warehouse would now contain:

name	deceased	updated_at
Louis XVI	false	1754
Marie Antoinette	false	1755
Louis XVII	false	1785

Updating a Record

Let's assume that our warehouse contains all the data that it did at the end of the previous section. Now, unfortunately the king and queen lose their heads. Let's see that delta:

name	deceased	updated_at
Louis XVI	true	1793
Marie Antoinette	true	1793

The output we expect to see in the warehouse is as follows:

In the history table:

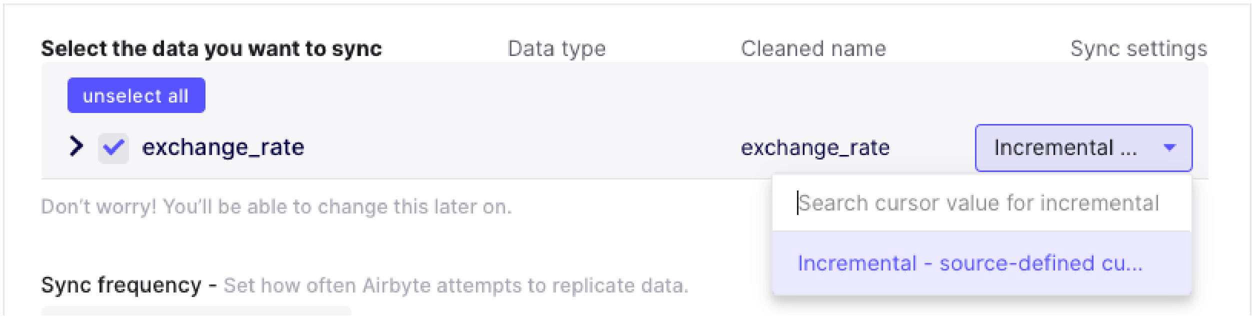
name	deceased	updated_at	start_at	end_at
Louis XVI	false	1754	1754	1793
Louis XVI	true	1793	1793	NULL
Louis XVII	false	1785	1785	NULL
Marie Antoinette	false	1755	1755	1793
Marie Antoinette	true	1793	1793	NULL

In the final de-duplicated table:

name	deceased	updated_at
Louis XVI	true	1793
Louis XVII	false	1785
Marie Antoinette	true	1793

Source-Defined Cursor

Some sources are able to determine the cursor that they use without any user input. For example, in the exchange rates source, the source knows that the date field should be used to determine the last record that was synced. In these cases, simply select the incremental option in the UI.

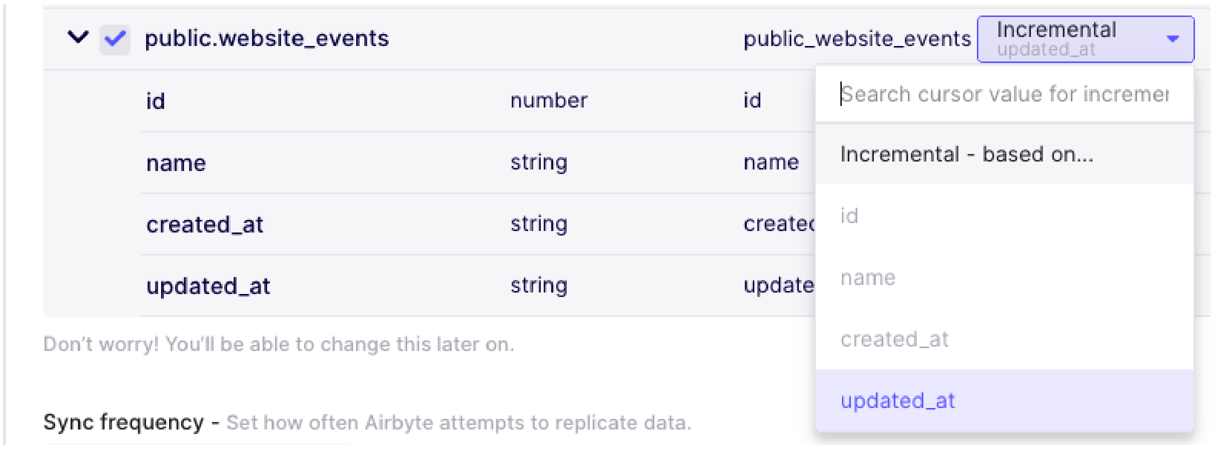


Select the data you want to sync
Data type
Cleaned name
Sync settings
unselect all
exchange_rate
exchange_rate
Incremental ...
Search cursor value for incremental
Incremental - source-defined cu...
Don't worry! You'll be able to change this later on.
Sync frequency - Set how often Airbyte attempts to replicate data.

(You can find a more technical details about the configuration data model here).

User-Defined Cursor

Some sources cannot define the cursor without user input. For example, in the postgres source, the user needs to choose which column in a database table they want to use as the `cursor field`. In these cases, select the column in the sync settings dropdown that should be used as the `cursor field`.



public.website_events
public_website_events
Incremental
updated_at
id
number
id
name
string
name
created_at
string
createc
updated_at
string
update
Search cursor value for incremer
Incremental - based on...
id
name
created_at
updated_at
Don't worry! You'll be able to change this later on.
Sync frequency - Set how often Airbyte attempts to replicate data.

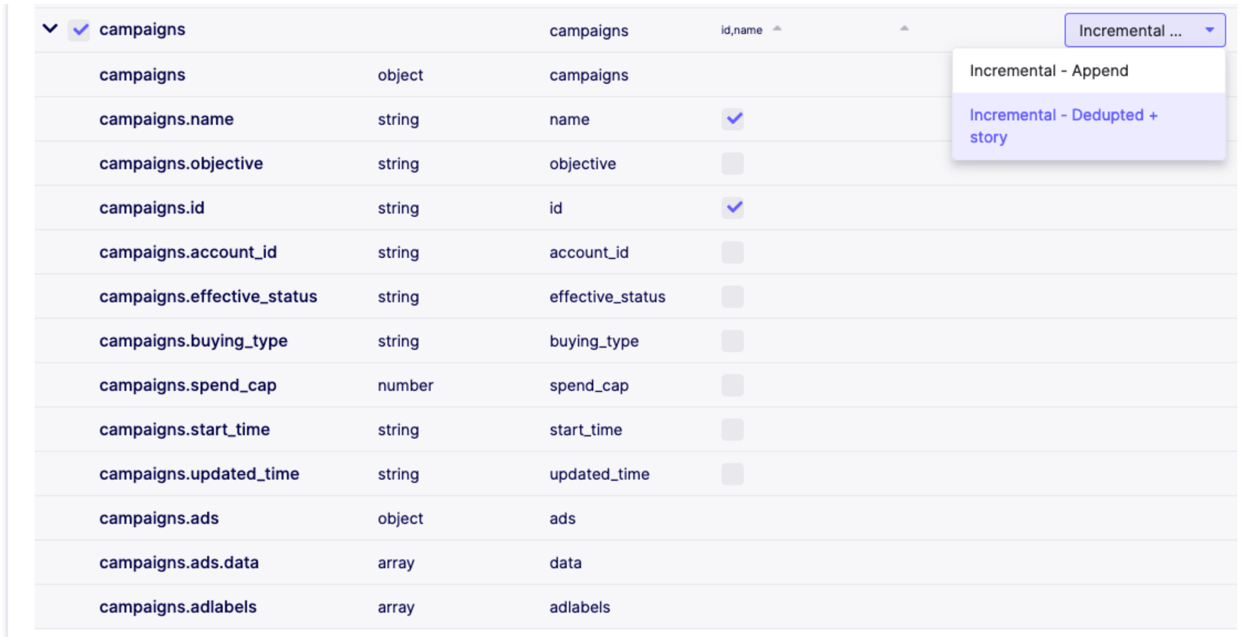
(You can find a more technical details about the configuration data model here).

Source-Defined Primary key

Some sources are able to determine the primary key that they use without any user input. For example, in the (JDBC) Database sources, primary key can be defined in the table's metadata.

User-Defined Primary key

Some sources cannot define the cursor without user input or the user may want to specify their own primary key on the destination that is different from the source definitions. In these cases, select the column in the sync settings dropdown that should be used as the `primary key` or `composite primary keys`.



✓ campaigns		campaigns	id,name	Incremental ...
campaigns	object	campaigns		
campaigns.name	string	name	✓	
campaigns.objective	string	objective		
campaigns.id	string	id	✓	
campaigns.account_id	string	account_id		
campaigns.effective_status	string	effective_status		
campaigns.buying_type	string	buying_type		
campaigns.spend_cap	number	spend_cap		
campaigns.start_time	string	start_time		
campaigns.updated_time	string	updated_time		
campaigns.ads	object	ads		
campaigns.ads.data	array	data		
campaigns.adlabels	array	adlabels		

In this example, we selected both the `campaigns.id` and `campaigns.name` as the composite primary key of our `campaigns` table.

Note that in Incremental Deduped History, the size of the data in your warehouse increases monotonically since an updated record in the source is appended to the destination history table rather than updated in-place as it is done with the final table. If you only care about having the latest snapshot of your data, you may want to periodically run cleanup jobs which retain only the latest instance of each record in the history tables.

Inclusive Cursors

When replicating data incrementally, Airbyte provides an at-least-once delivery guarantee. This means that it is acceptable for sources to re-send some data when ran incrementally. One case where this is particularly relevant is when a source's cursor is not very granular. For example, if a cursor field has the granularity of a day (but not hours, seconds, etc), then if that source is run twice in the same day, there is no way for the source to know which records that are that date were already replicated earlier that day. By convention, sources should prefer resending data if the cursor field is ambiguous.

Known Limitations

Due to the use of a cursor column, if modifications to the underlying records are made without properly updating the cursor field, then the updated records won't be picked up by the Incremental sync as expected since the source connectors extract delta rows using a SQL query looking like:

```
Unset


select * from table where cursor_field >
'last_sync_max_cursor_field_value'
```

Let's say the following data already exists into our data warehouse.

name	deceased	updated_at
Louis XVI	false	1754
Marie Antoinette	false	1755

At the start of the next sync, the source data contains the following new record:

name	deceased	updated_at
Louis XVI	true	1754

At the end of the second incremental sync, the data warehouse would still contain data from the first sync because the delta record did not provide a valid value for the cursor field (the cursor field is not greater than last sync's max value, `1754 < 1755`), so it is not emitted by the source as a new or modified record.

name	deceased	updated_at
Louis XVI	false	1754
Marie Antoinette	false	1755

Similarly, if multiple modifications are made during the same day to the same records. If the frequency of the sync is not granular enough (for example, set for every 24h), then intermediate modifications to the data are not going to be detected and emitted. Only the state of data at the time the sync runs will be reflected in the destination.

Those concerns could be solved by using a different incremental approach based on binary logs, Write-Ahead-Logs (WAL), or also called Change Data Capture (CDC).

The current behavior of Incremental is not able to handle source schema changes yet, for example, when a column is added, renamed or deleted from an existing table etc. It is recommended to trigger a Full refresh - Overwrite to correctly replicate the data to the destination with the new schema changes.

Additionally, this sync mode is only supported for destinations where dbt/normalization is possible for the moment. The de-duplicating logic is indeed implemented as dbt models as part of a sequence of transformations applied after the Extract and Load activities (thus, an ELT approach). Nevertheless, it is theoretically possible that destinations can handle directly this logic (maybe in the future) before actually writing records to the destination (as in traditional ETL manner), but that's not the way it is implemented at this time.

If you are not satisfied with how transformations are applied on top of the appended data, you can find more relevant SQL transformations you might need to do on your data in the Connecting EL with T using SQL (part 1/2)

Exchange Rates API

Overview

The exchange rates integration is a toy integration to demonstrate how Airbyte works with a very simple source.

It pulls all its data from https://exchangeratesapi.io

Output schema

It contains one stream: `exchange_rates`

Each record in the stream contains many fields:

- The date of the record
- One field for every supported currency which contain the value of that currency on that date.

Data type mapping

Currencies are `number` and the date is a `string`.

Features

Feature	Supported?
Full Refresh Sync	Yes
Incremental - Append Sync	Yes
Namespaces	No

Getting started

Requirements

- API Access Key

Setup guide

In order to get an `API Access Key` please go to this page and enter needed info. After registration and login you will see your `API Access Key`, also you may find it here.

If you have `free` subscription plan (you may check it here) this means that you will have 2 limitations:

1. 1000 API calls per month.
2. You won't be able to specify the `base` parameter, meaning that you will be dealing only with default base value which is EUR.

A Beginner's Guide to the AirbyteCatalog

Overview

The goal of this article is to make the `AirbyteCatalog` approachable to someone contributing to Airbyte for the first time. If you are looking to get deeper into the details of the catalog, you can read our technical specification on it here.

The goal of the `AirbyteCatalog` is to describe *what* data is available in a source. The goal of the `ConfiguredAirbyteCatalog` is to, based on an `AirbyteCatalog`, specify *how* data from the source is replicated.

Contents

This article will illustrate how to use `AirbyteCatalog` via a series of examples. We recommend reading the Database Example first. The other examples, will refer to knowledge described in that section. After that, jump around to whichever example is most pertinent to your inquiry.

In order to understand in depth how to configure incremental data replication, head over to the incremental replication docs.

Database Example

Let's jump into an example using a relational database. We will assume we have a database with the following schema:

Unset

```
CREATE TABLE "airlines" (
    "id"   INTEGER,
    "name" VARCHAR
);

CREATE TABLE "pilots" (
    "id"   INTEGER,
    "airline_id" INTEGER,
    "name" VARCHAR
);
```

AirbyteCatalog

We would represent this data in a catalog as follows:

```
Unset

{
 "streams": [
   {
     "name": "airlines",
     "supported_sync_modes": [
       "full_refresh",
       "incremental"
     ],
     "source_defined_cursor": false,
     "json_schema": {
       "type": "object",
       "properties": {
         "id": {
           "type": "number"
         },
         "name": {
           "type": "string"
         }
       }
     }
   },
   {
     "name": "pilots",
     "supported_sync_modes": [
       "full_refresh",
       "incremental"
     ],
     "source_defined_cursor": false,
     "json_schema": {
       "type": "object",
       "properties": {
         "id": {
           "type": "number"
         },
         "airline_id": {
           "type": "number"
         },
         "name": {
           "type": "string"
         }
       }
     }
   }
 ]
}
```

The catalog is structured as a list of `AirbyteStream`. In the case of a database a "stream" is analogous to a table. (For APIs the mapping can be a more creative; we will discuss it later in API Examples)

Let's walk through what each field in a stream means.

- `name` - The name of the stream.
- `supported_sync_modes` - This field lists the type of data replication that this source supports. The possible values in this array include `FULL_REFRESH` (docs) and `INCREMENTAL` (docs).

- `source_defined_cursor` - If the stream supports `INCREMENTAL` replication, then this field signals whether the source can figure out how to detect new records on its own or not.
- `json_schema` - This field is a JsonSchema object that describes the structure of the data. Notice that each key in the `properties` object corresponds to a column name in our database table.

Now we understand *what* data is available from this source. Next we will configure *how* we want to replicate that data.

ConfiguredAirbyteCatalog

Let's say that we do not care about replicating the pilot data at all. We do want to replicate the airlines data as a `FULL_REFRESH`. Here's what our `ConfiguredAirbyteCatalog` would look like.

Unset

```
{
 "streams": [
   {
     "sync_mode": "FULL_REFRESH",
     "stream": {
       "name": "airlines",
       "supported_sync_modes": [
         "full_refresh",
         "incremental"
       ],
       "source_defined_cursor": false,
       "json_schema": {
         "type": "object",
         "properties": {
           "id": {
             "type": "number"
           },
           "name": {
             "type": "string"
           }
         }
       }
     }
   }
 ]
}
```

Just as with the `AirbyteCatalog` the `ConfiguredAirbyteCatalog` contains a list. This time it is a list of `ConfiguredAirbyteStream` (instead of just `AirbyteStream`).

Let's walk through each field in the `ConfiguredAirbyteStream`:

- `sync_mode` - This field must be one of the values that was in `supported_sync_modes` in the `AirbyteStream` - Configures which sync mode will be used when data is replicated.
- `stream` - Hopefully this one looks familiar! This field contains an `AirbyteStream`. It should be *identical* to the one we saw in the `AirbyteCatalog`.

- `cursor_field` - When `sync_mode` is `INCREMENTAL` and `source_defined_cursor = false`, this field configures which field in the stream will be used to determine if a record should be replicated or not. Read more about this concept in our documentation of incremental replication.

Summary of the Postgres Example

When thinking about `AirbyteCatalog` and `ConfiguredAirbyteCatalog`, remember that the `AirbyteCatalog` describes *what* data is present in the source (and metadata around what replication configuration it can support). It is output by the `discover` method of source. It should be treated as an immutable object; if you are ever manually editing a catalog outside of a source, you've gone off the rails. The `ConfiguredAirbyteCatalog` is a mutable configuration object that specifies, for each `AirbyteStream`, *how* (and if) it should be replicated. The `ConfiguredAirbyteCatalog` does this by wrapping each `AirbyteStream` in an `AirbyteCatalog` inside a `ConfiguredAirbyteStream`.

API Examples

The `AirbyteCatalog` offers the flexibility in how to model the data for an API. In the next two examples, we will model data from the same API--a stock ticker--in two different ways. In the first, the source will return a single stream called `ticker`, and in the second, the source with return a stream for each stock symbol it is configured to retrieve data for. Each stream's name will be a stock symbol.

Static Streams Example

Let's imagine we want to create a basic Stock Ticker source. The goal of this source is to take in a single stock symbol and return a single stream. We will call the stream `ticker` and will contain the closing price of the stock. We will assume that you already have a rough understanding of the `AirbyteCatalog` and the `ConfiguredAirbyteCatalog` from the previous database example.

AirbyteCatalog

Here is what the `AirbyteCatalog` might look like.

Unset

```
{
 "streams": [
   {
     "name": "ticker",
     "supported_sync_modes": [
       "full_refresh",
       "incremental"
     ],
     "source_defined_cursor": false,
     "json_schema": {
       "type": "object",
       "properties": {
         "symbol": {
           "type": "string"
         },
         "price": {
           "type": "number"
         },
         "date": {
           "type": "string"
         }
```

```
        }
      }
    }
  ]
}
```

This catalog looks pretty similar to the `AirbyteCatalog` that we created for the Database Example. For the data we've picked here, you can think about `ticker` as a table and then each field it returns in a record as a column, so it makes sense that these look pretty similar.

ConfiguredAirbyteCatalog

The `ConfiguredAirbyteCatalog` follows the same rules as we described in the Database Example. It just wraps the `AirbyteCatalog` described above.

Dynamic Streams Example

Now let's build a stock ticker source that handles returning ticker data for *multiple* stocks. The name of each stream will be the stock symbol that it represents.

AirbyteCatalog

```
Unset

{
 "streams": [
   {
     "name": "TSLA",
     "supported_sync_modes": [
        "full_refresh",
        "incremental"
      ],
      "source_defined_cursor": false,
      "json_schema": {
        "type": "object",
        "properties": {
          "symbol": {
            "type": "string"
          },
          "price": {
            "type": "number"
          },
          "date": {
            "type": "string"
          }
        }
      }
    },
    {
      "name": "FB",
      "supported_sync_modes": [
        "full_refresh",
        "incremental"
      ],
      "source_defined_cursor": false,
      "json_schema": {
        "type": "object",
        "properties": {
          "symbol": {
            "type": "string"
          },
          "price": {
            "type": "number"
          },
          "date": {
            "type": "string"
          }
        }
      }
    }
  ]
}
```

This example provides another way of thinking about exposing data in a source. As a developer building a source, you can model the `AirbyteCatalog` for a source however makes most sense to the use case you are trying to fulfill.

Nested Schema Example

Often, a data source contains "nested" data. In other words this is data where each record contains other objects nested inside it. Cases like this cannot be easily modeled just as tables / columns. This is why Airbyte uses JsonSchema to model the schema of its streams.

Let's imagine we are modeling a flight object. A flight object might look like this:

```
Unset

{
 "airline": "alaska",
 "origin": {
   "airport_code": "SFO",
   "terminal": "2",
   "gate": "G23"
 },
 "destination": {
   "airport_code": "JFK",
   "terminal": "7",
   "gate": "1"
 }
}
```

The `AirbyteCatalog` would look like this:

```
Unset

{
 "streams": [
   {
     "name": "flights",
     "supported_sync_modes": [
       "full_refresh"
     ],
     "source_defined_cursor": false,
     "json_schema": {
       "type": "object",
       "properties": {
         "airline": {
           "type": "string"
         },
         "origin": {
           "type": "object",
           "properties": {
             "airport_code": {
               "type": "string"
             },
             "terminal": {
               "type": "string"
             },
             "gate": {
               "type": "string"
             }
           }
         },
         "destination": {
           "type": "object",
           "properties": {
             "airport_code": {
               "type": "string"
             },
             "terminal": {
               "type": "string"
             },
             "gate": {
               "type": "string"
             }
           }
         }
```

```
            }
          }
        }
      ]
    }
```

Because Airbyte uses JsonSchema to model the schema of streams, it is able to handle arbitrary nesting of data in a way that a table / column based model cannot.

Airbyte Protocol

Goals

The Airbyte Protocol describes a series of standard components and all the interactions between them in order to declare an ELT pipeline. All message passing across components is done via serialized JSON messages for inter-process communication.

This document describes the protocol as it exists in its CURRENT form. Stay tuned for an RFC on how the protocol will evolve.

This document is intended to contain ALL the rules of the Airbyte Protocol in one place. Anything not contained in this document is NOT part of the Protocol. At the time of writing, there is one known exception, which is the Supported Data Types, which contains rules on data types that are part of the Protocol. That said, there are additional articles, e.g. A Beginner's Guide to the Airbyte Catalog that repackage the information in this document for different audiences.

Key Concepts

There are 2 major components in the Airbyte Protocol: Source and Destination. These components are referred to as Actors. A source is an application that is described by a series of standard interfaces. This application extracts data from an underlying data store. A data store in this context refers to the tool where the data is actually stored. A data store includes: databases, APIs, anything that produces data, etc. For example, the Postgres Source is a Source that pulls from Postgres (which is a data store). A Destination is an application that is described by a series of standard interfaces that loads data into a data store.

The key primitives that the Protocol uses to describe data are Catalog, Configured Catalog, Stream, Configured Stream, and Field:

- Stream - A Stream describes the schema of a resource and various metadata about how a user can interact with that resource. A resource in this context might refer to a database table, a resource in a REST API, or a data stream.
- Field - A Field refers to a "column" in a Stream. In a database this would be a column; in a JSON object it is a field.
- Catalog - A Catalog is a list of Streams that describes the data in the data store that a Source represents.

An Actor can advertise information about itself with an Actor Specification. One of the main pieces of information the specification shares is what information is needed to configure an Actor.

Each of these concepts is described in greater depth in their respective section.

Actor Interface

This section describes important details about the interface over actors. It reviews parts of the interface that are the same across all actors. It also describes some invariants for all methods in actor interfaces.

Common Interface Methods

The following part of the interface is identical across all actors:

```
Unset


spec() -> ConnectorSpecification

check(Config) -> AirbyteConnectionStatus
```

These methods are described in their respective sections (spec, check).

Interface Invariants

The output of each method in actor interface is wrapped in an `AirbyteMessage`. This struct is an envelope for the return value of any message in the described interface. See the section the AirbyteMessage section below for more details. For the sake of brevity, interface diagrams will elide these `AirbyteMessage`s.

Additionally, all methods described in the protocol can emit `AirbyteLogMessage`s and `AirbyteTraceMessage`s (for more details see Logging). These messages allow an actor to emit logs and other informational metadata. All subsequent method signatures will assume that any number of messages of these types (wrapped in the `AirbyteMessage`) may be emitted.

Each method in the protocol has 3 parts:

1. Input: these are the arguments passed to the method.
2. Data Channel Egress (Output): all outputs from a method are via STDOUT. While some method signatures declare a single return value, in practice, any number of `AirbyteLogMessage`s and `AirbyteTraceMessage`s may be emitted. An actor is responsible for closing STDOUT to declare that it is done.
3. Data Channel Ingress: after a method begins running, data can be passed to it via STDIN. For example, records are passed to a Destination on STDIN so that it can load them into a data warehouse.

Sources are a special case and do not have a Data Channel Ingress.

Additional Invariants

- All arguments passed to an Actor and all messages emitted from an Actor are serialized JSON.
- All messages emitted from Actors must be wrapped in an `AirbyteMessage`(ref) envelope.
- Messages not wrapped in the `AirbyteMessage` must be dropped (e.g. not be passed from Source to Destination). However certain implementations of the Airbyte Protocol may choose to store and log unknown messages for debugging purposes.
- Each message must be on its own line. Multiple messages *cannot* be sent on the same line. The JSON objects cannot be serialized across multiple lines.

- STDERR should only be used for log messages (for errors). All other Data Channel Data moves on STDIN and STDOUT.

Common Interface

Spec

```
Unset


spec() -> ConnectorSpecification
```

The `spec` command allows an actor to broadcast information about itself and how it can be configured.

Input:

1. none.

Output:

1. `spec` - a ConnectorSpecification wrapped in an `AirbyteMessage` of type `spec`. See the Actor Specification for more details on the information in the spec.

Check

```
Unset


check(Config) -> AirbyteConnectionStatus
```

The `check` command validates that, given a configuration, that the Actor is able to connect and access all resources that it needs in order to operate. e.g. Given some Postgres credentials, it determines whether it can connect to the Postgres database. If it can, it will return a success response. If it fails (perhaps the password is incorrect), it will return a failed response and (when possible) a helpful error message. If an actor's `check` command succeeds, it is expected that all subsequent methods in the sync will also succeed.

Input:

1. config - A configuration JSON object that has been validated using ConnectorSpecification#connectionSpecification (see ActorSpecification for information on connectionSpecification).

Output:

1. connectionStatus - an AirbyteConnectionStatus Message wrapped in an AirbyteMessage of type connection_status.

Source

A Source is an application that extracts data from an underlying data store. A Source implements the following interface:

```
Unset

spec() -> ConnectorSpecification

check(Config) -> AirbyteConnectionStatus

discover(Config) -> AirbyteCatalog

read(Config, ConfiguredAirbyteCatalog, State) ->
Stream<AirbyteRecordMessage | AirbyteStateMessage>
```

spec and check are the same as the commands described in the Common Commands section.

Discover

The discover method detects and describes the *structure* of the data in the data store and which Airbyte configurations can be applied to that data. For example, given a Postges source and valid Config, discover would return a list of available tables as streams.

Input:

1. `config` - A configuration JSON object that has been validated using `ConnectorSpecification#connectionSpecification` (see ActorSpecification for information on `connectionSpecification`).

Output:

1. `catalog` - an AirbyteCatalog wrapped in an `AirbyteMessage` of type `catalog`. See the Catalog Section for details.

Read

`read` extracts data from the underlying data store and emits it as `AirbyteRecordMessage`s. It also emits `AirbyteStateMessage`s to allow checkpointing replication.

Input:

1. `config` - A configuration JSON object that has been validated using `ConnectorSpecification#connectionSpecification` (see ActorSpecification for information on `connectionSpecification`).
2. `configured catalog` - A `ConfiguredAirbyteCatalog` is built on top of the `catalog` returned by `discover`. The `ConfiguredAirbyteCatalog` specifies HOW the data in the catalog should be replicated. The catalog is documented in the Catalog Section.
3. `state` - An JSON object that represents a checkpoint in the replication. This object is only ever written or read by the source, so it is a JSON blob with whatever information is necessary to keep track of how much of the data source has already been read (learn more in the State & Checkpointing Section).

Output:

1. `message stream` - An iterator of `AirbyteRecordMessage`s and `AirbyteStateMessage`s piped to the Data Channel Egress i.e: stdout.
    - A source outputs `AirbyteStateMessages` in order to allow checkpointing data replication. State is described in more detail below in the State & Checkpointing section.
    - Only `AirbyteRecordMessage`s that contain streams that are in the catalog will be processed. Those that do not will be ignored. See Schema Mismatches for more details.

- AirbyteRecordMessages from multiple streams can be multiplexed/mixed together, and do not need to be emitted serially as a group.

Destination

A destination receives data on the Data Channel Ingress and loads it into an underlying data store (e.g. data warehouse or database).

It implements the following interface.

```
Unset

spec() -> ConnectorSpecification
check(Config) -> AirbyteConnectionStatus
write(Config, AirbyteCatalog,
Stream<AirbyteMessage>(stdin)) ->
Stream<AirbyteStateMessage>
```

For the sake of brevity, we will not re-describe `spec` and `check`. They are exactly the same as those commands described for the Source.

Write

Input:

1. `config` - A configuration JSON object that has been validated using the `ConnectorSpecification`.
2. `catalog` - An `AirbyteCatalog`. This `catalog` should be a subset of the `catalog` returned by the `discover` command. Any `AirbyteRecordMessages`s that the destination receives that do *not* match the structure described in the `catalog` will fail.
3. `message stream` - (this stream is consumed on stdin--it is not passed as an arg). It will receive a stream of JSON-serialized `AirbyteMesssage`.

Output:

1. `message stream` - A stream of `AirbyteStateMessage`s piped to stdout. The destination connector should only output state messages if they were previously received as input on stdin. Outputting a state message indicates that all records which came before it have been successfully written to the destination. Implementations of this spec will likely want to move messages filtering and validation upstream of the destination itself

- The destination should read in the `AirbyteMessages` and write any that are of type `AirbyteRecordMessage` to the underlying data store.
- The destination should ignore fields or streams that are out of sync with the `catalog`. The destination should always make its best effort to load what data is there that does match that catalog. e.g. if the User Stream has the fields first_name and last_name in the catalog, but the record has first_name and eye_color, the destination should persist first_name, even though last_name is missing. It should ignore eye_color as extraneous.

This concludes the overview of the Actor Interface. The remaining content will dive deeper into each concept covered so far.

Actor Specification

The specification allows the Actor to share information about itself.

The `connectionSpecification` is JSONSchema that describes what information needs to the actor for it operate. e.g. If using a Postgres Source, the `ConnectorSpecification` would specify that a `hostname`, `port`, and `password` are required in order for the connector to function. This JSONSchema can be used to validate that the provided inputs are valid. e.g. If `port` is one of the fields and the JsonSchema in the `connectionSpecification` specifies that this field should be a number, if a user inputs "airbyte", they will receive an error. For connection specification, Airbyte adheres to JsonSchema validation rules. The Airbyte implementation of the Protocol is able to render this JSONSchema to produce a form for users to fill in the information for an Actor.

The specification also contains information about what features the Actor supports.

- `protocol_version` describes which version of the protocol the Connector supports. The default value is "0.2.0".

- `supported_destination_sync_modes` - describes which sync modes a destination is able to support. See [Sync Modes](#).

`documentationUrl` and `changelogUrl` are optional fields that link to additional information about the connector.

The following are fields that still exist in the specification but are slated to be removed as they leak choices about how Airbyte implements the protocol as opposed to being strictly necessary part of the protocol.

- `supportsIncremental` is deprecated and can be ignored. It is vestigial from when full refresh / incremental was specified at the Actor level.
- `supportsNormalization` determines whether the Destination supports Basic Normalization
- `supportsDBT` - determines whether the Destination supports Basic Normalization
- `authSpecification` and `advanced_auth` will be removed from the protocol and as such are not documented. Information on their use can be found here.

```
Unset

ConnectorSpecification:
 description: Specification of a connector
(source/destination)
 type: object
 required:
   - connectionSpecification
 additionalProperties: true
 properties:
   # General Properties (Common to all connectors)
   protocol_version:
     description: "the Airbyte Protocol version supported
by the connector. Protocol versioning uses SemVer."
     type: string
   documentationUrl:
     type: string
     format: uri
   changelogUrl:
     type: string
     format: uri
   connectionSpecification:
     description: ConnectorDefinition specific blob. Must
be a valid JSON string.
     type: object
     existingJavaType:
com.fasterxml.jackson.databind.JsonNode
   # Connector Type Properties (Common to all connectors from
same type)
   # Source Connectors Properties
   supportsIncremental:
     description: (deprecated) If the connector supports
incremental mode or not.
     type: boolean
   # Destination Connectors Properties
   # Normalization is currently implemented using dbt, so it
requires `supportsDBT` to be true for this to be true.
   supportsNormalization:
     description: If the connector supports normalization
or not.
     type: boolean
     default: false
   supportsDBT:
     description: If the connector supports DBT or not.
     type: boolean
     default: false
   supported_destination_sync_modes:
     description: List of destination sync modes supported
by the connector
     type: array
     items:
       "$ref": "#/definitions/DestinationSyncMode"
```

Catalog

Overview

An `AirbyteCatalog` is a struct that is produced by the `discover` action of a source. It is a list of `AirbyteStream`s. Each `AirbyteStream` describes the data available to be synced from the source. After a source produces an `AirbyteCatalog` or `AirbyteStream`, they should be treated as read only. A

`ConfiguredAirbyteCatalog` is a list of `ConfiguredAirbyteStream`s. Each `ConfiguredAirbyteStream` describes how to sync an `AirbyteStream`.

Each `AirbyteStream` of these contain a `name` and `json_schema` field. The `json_schema` field accepts any valid JsonSchema and describes the structure of a stream. This data model is intentionally flexible. That can make it a little hard at first to mentally map onto your own data, so we provide some examples below: *If we are using a data source that is a traditional relational database, each table in that database would map to an `AirbyteStream`. Each column in the table would be a key in the `properties` field of the `json_schema` field.* e.g. If we have a table called `users` which had the columns `name` and `age` (the age column is optional) the `AirbyteCatalog` would look like this:

Unset

```
{
 "streams": [
   {
     "name": "users",
     "json_schema": {
       "type": "object",
       "required": ["name"],
       "properties": {
         "name": {
           "type": "string"
         },
         "age": {
           "type": "number"
```

```
          }
        }
      }
    }
  ]
}
```

If we are using a data source that wraps an API with multiple different resources (e.g. `api/customers` and `api/products`) each route would correspond to a stream. The JSON object returned by each route would be described in the `json_schema` field.

e.g. In the case where the API has two endpoints `api/customers` and `api/products` and each returns a list of JSON objects, the `AirbyteCatalog` might look like this. (Note: using the JSON schema standard for defining a stream allows us to describe nested objects. We are not constrained to a classic "table/columns" structure)

Unset

```
{
  "streams": [
    {
      "name": "customers",
      "json_schema": {
        "type": "object",
        "required": ["name"],
        "properties": {
          "name": {
            "type": "string"
          }
        }
      }
    },
    {
      "name": "products",
      "json_schema": {
        "type": "object",
        "required": ["name", "features"],
        "properties": {
          "name": {
            "type": "string"
          },
          "features": {
            "type": "array",
            "items": {
              "type": "object",
              "required": ["name", "productId"],
              "properties": {
                "name": { "type": "string" },
                "productId": { "type": "number" }
              }
            }
          }
        }
      }
    }
  ]
 }
```

Note: Stream and field names can be any UTF8 string. Destinations are responsible for cleaning these names to make them valid table and column names in their respective data stores.

Namespace

Technical systems often group their underlying data into namespaces with each namespace's data isolated from another namespace. This isolation allows for better organisation and flexibility, leading to better usability.

An example of a namespace is the RDBMS's `schema` concept. An API namespace might be used for multiple accounts (e.g. `company_a` vs `company_b`, each having a "users" and "purchases" stream). Some common use cases for schemas are enforcing permissions, segregating test and production data and general data organization.

The `AirbyteStream` represents this concept through an optional field called `namespace`. Additional documentation on Namespaces can be found here.

Cursor

- The cursor is how sources track which records are new or updated since the last sync.
- A "cursor field" is the field that is used as a comparable for making this determination.
    - If a configuration requires a cursor field, it requires an array of strings that serves as a path to the desired field. e.g. if the structure of a stream is `{ value: 2, metadata: { updated_at: 2020-11-01 } }` the `default_cursor_field` might be `["metadata", "updated_at"]`.

AirbyteStream

This section will document the meaning of each field in an `AirbyteStream`

- `json_schema` - This field contains a JsonSchema representation of the schema of the stream.
- `supported_sync_modes` - The sync modes that the stream supports. By default, all sources support `FULL_REFRESH`. Even if this array is empty, it can be assumed that a source supports `FULL_REFRESH`. The allowed sync modes are `FULL_REFRESH` and `INCREMENTAL`.
- `source_defined_cursor` - If a source supports the `INCREMENTAL` sync mode, and it sets this field to true, it is responsible for determining internally how it tracks which records in a source are new or updated since the last sync. When set to `true`, `default_cursor_field` should also be set.
- `default_cursor_field` - If a source supports the `INCREMENTAL` sync mode, it may, optionally, set this field. If this field is set, and the user does not override it with the `cursor_field` attribute in the `ConfiguredAirbyteStream` (described below), this field will be used as the cursor. It is an array of keys to a field in the schema.

Data Types

Airbyte maintains a set of types that intersects with those of JSONSchema but also includes its own. More information on supported data types can be found in Supported Data Types.

ConfiguredAirbyteStream

This section will document the meaning of each field in an `ConfiguredAirbyteStream`

```
Unset

ConfiguredAirbyteStream:
 type: object
 additionalProperties: true
 required:
   - stream
   - sync_mode
   - destination_sync_mode
 properties:
   stream:
     "$ref": "#/definitions/AirbyteStream"
   sync_mode:
     "$ref": "#/definitions/SyncMode"
     default: full_refresh
   cursor_field:
     description: Path to the field that will be used to
determine if a record is new or modified since the last
sync. This field is REQUIRED if `sync_mode` is
`incremental`. Otherwise it is ignored.
     type: array
     items:
       type: string
   destination_sync_mode:
     "$ref": "#/definitions/DestinationSyncMode"
     default: append
   primary_key:
     description: Paths to the fields that will be used as
primary key. This field is REQUIRED if
`destination_sync_mode` is `*_dedup`. Otherwise it is
ignored.
     type: array
     items:
       type: array
       items:
         type: string
SyncMode:
 type: string
 enum:
   - full_refresh
   - incremental
DestinationSyncMode:
 type: string
 enum:
   - append
   - overwrite
```

```
    - append_dedup # SCD Type 1 & 2
```

- `stream` - This field contains the `AirbyteStream` that it is configured.
- `sync_mode` - The sync mode that will be used to by the source to sync that stream. The value in this field MUST be present in the `supported_sync_modes` array for the discovered `AirbyteStream` of this stream.
- `cursor_field` - This field is an array of keys to a field in the schema that in the `INCREMENTAL` sync mode will be used to determine if a record is new or updated since the last sync.
    - If an `AirbyteStream` has `source_defined_cursor` set to `true`, then the `cursor_field` attribute in `ConfiguredAirbyteStream` will be ignored.
    - If an `AirbyteStream` defines a `default_cursor_field`, then the `cursor_field` attribute in `ConfiguredAirbyteStream` is not required, but if it is set, it will override the default value.
    - If an `AirbyteStream` does not define a `cursor_field` or a `default_cursor_field`, then `ConfiguredAirbyteStream` must define a `cursor_field`.
- `destination_sync_mode` - The sync mode that will be used the destination to sync that stream. The value in this field MUST be present in the `supported_destination_sync_modes` array in the specification for the Destination.

Source Sync Modes

- `incremental` - send all the data for the Stream since the last sync (e.g. the state message passed to the Source). This is the most common sync mode. It only sends new data.
- `full_refresh` - resend all data for the Stream on every sync. Ignores State. Should only be used in cases where data is very small, there is no way to keep a cursor into the data, or it is necessary to capture a snapshot in time of the whole dataset. Be careful using this, because misuse can lead to sending much more data than expected.

Destination Sync Modes

- `append` - add new data from the sync to the end of whatever already data already exists.
- `append_dedup` - add new data from the sync to the end of whatever already data already exists and deduplicate it on primary key. This is the most common sync mode. It does require that a primary exists in the data. This is also known as SCD Type 1 & 2.
- `overwrite` - replace whatever data exists in the destination data store with the data that arrives in this sync.

Logic for resolving the Cursor Field

This section lays out how a cursor field is determined in the case of a Stream that is doing an `incremental` sync.

- If `source_defined_cursor` in `AirbyteStream` is true, then the source determines the cursor field internally. It cannot be overridden. If it is false, continue...
- If `cursor_field` in `ConfiguredAirbyteStream` is set, then the source uses that field as the cursor. If it is not set, continue...
- If `default_cursor_field` in `AirbyteStream` is set, then the sources use that field as the cursor. If it is not set, continue...
- Illegal - If `source_defined_cursor`, `cursor_field`, and `default_cursor_field` are all false-y, this is an invalid configuration.

Schema Mismatches

Over time, it is possible for the catalog to become out of sync with the underlying data store it represents. The Protocol is design to be resilient to this. In should never fail due to a mismatch.

Scenario	Outcome
Stream exists in catalog but not in data store	When the source runs, it will not find the data for that stream. All other streams sync as usual.

Stream exists in data store but not in catalog	When the source runs, it never looks for the data in the store related to that stream and thus does not emit it.
Field exists in catalog but not in data store	If the column for a table is remove in the underlying data store the Source will not find it and will not replicate it. It should not cause a failure. The data simply will not be there.
Field exists in data store but not in catalog	When the source runs, it never looks for the field in the store. It should not emit that field. If it does, it should be ignored downstream. The existence of an unknown field should not cause a failure.

In short, if the catalog is ever out of sync with the schema of the underlying data store, it should never block replication for data that is present.

State & Checkpointing

Sources are able to emit state in order to allow checkpointing data replication. The goal is that given wherever a sync stops (whether this is due to all data available at the time being replicated or due to a failure), the next time the Source attempts to extract data it can pick up where it left off and not have to go back to the beginning.

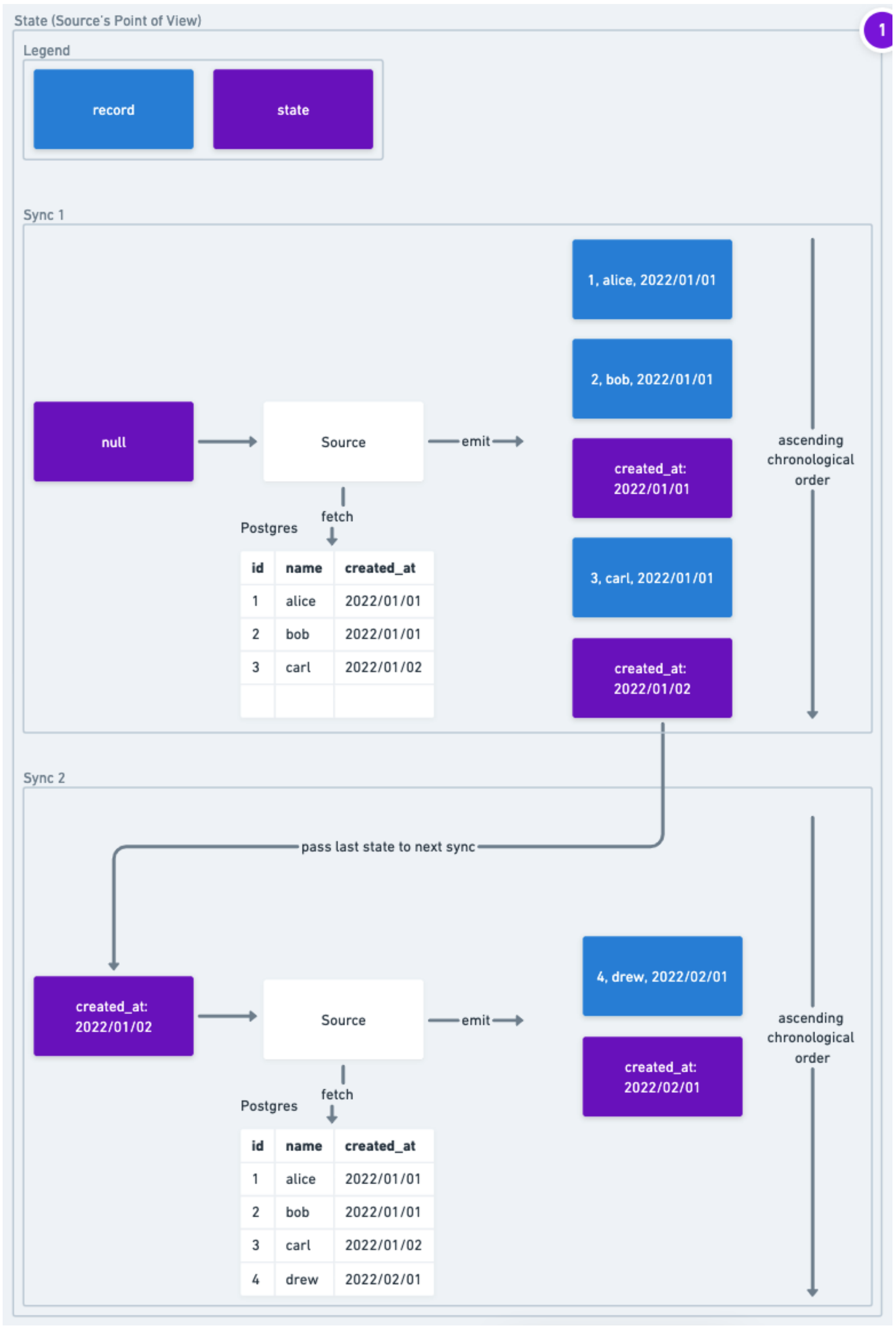
This concept enables incremental syncs--syncs that only replicate data that is new since the previous sync.

State also enables Partial Success. In the case where during a sync there is a failure before all data has been extracted and committed, if all records up to a certain state are committed, then the next time the sync happens, it can start from that state as opposed to going back to the beginning. Partial Success is powerful, because especially in the case of high data volumes and long syncs, being able to pick up from wherever the failure occurred can costly re-syncing of data that has already been replicated.

State & Source

This section will step through how state is used to allow a Source to pick up where it left off. A Source takes state as an input. A Source should be able to take that input and use it to determine where it left off the last time. The contents of the Source is a black box to the Protocol. The Protocol provides an envelope for the Source to put its state in and then passes the state back in that envelope. The Protocol never needs to know anything about the contents of the state. Thus, the Source can track state however makes most sense to it.

Here is an example of the lifecycle of state in reference to the Source.



id	name	created_at
1	alice	2022/01/01
2	bob	2022/01/01
3	carl	2022/01/02

id	name	created_at
1	alice	2022/01/01
2	bob	2022/01/01
3	carl	2022/01/02
4	drew	2022/02/01

-- link to source image

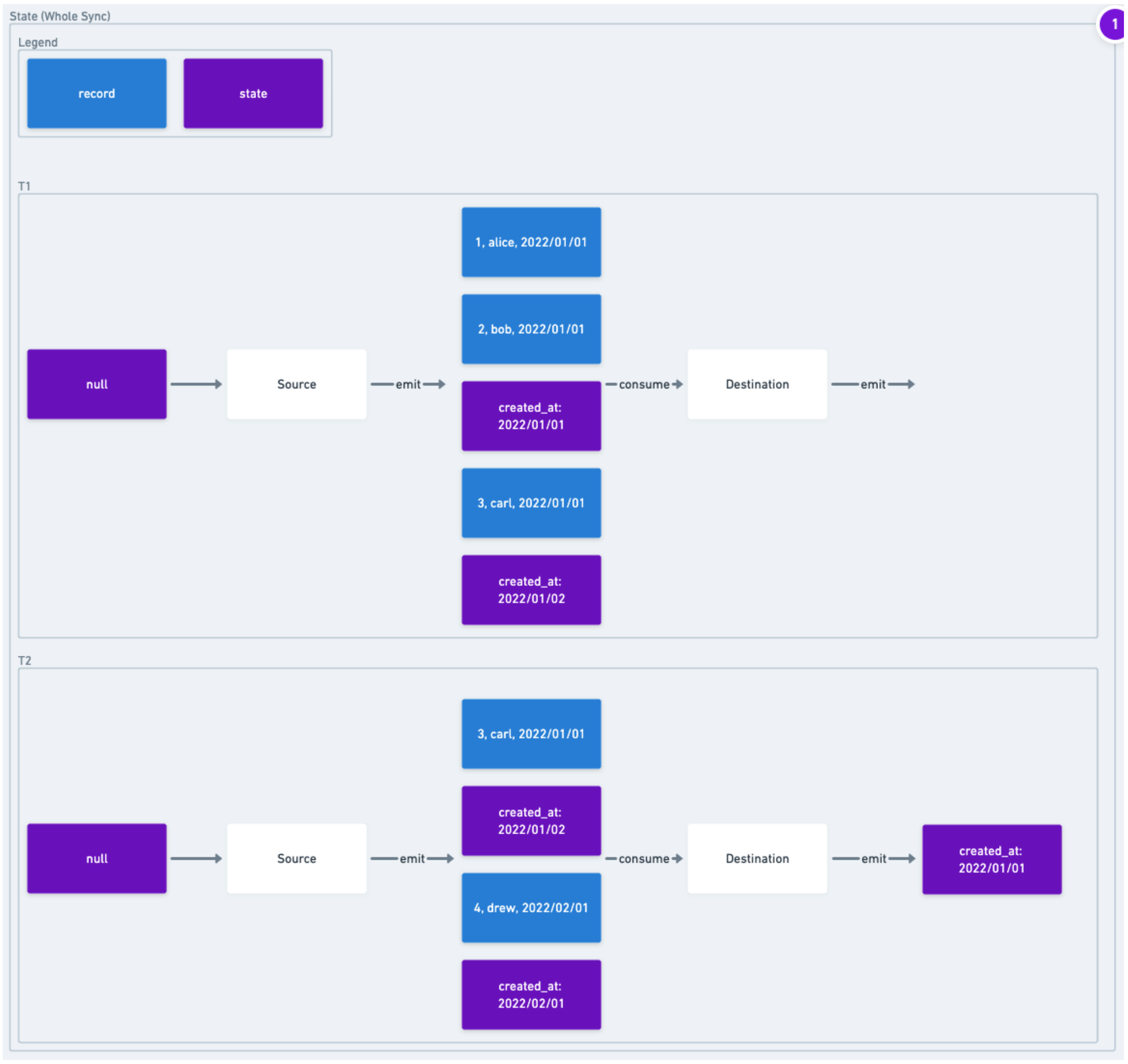
In Sync 1, the Postgres Source receives null state as an input. Thus, when it queries data from the database, it starts at the beginning and returns all the records it finds. In addition, it emits state records that show track the high watermark of what records it has replicated. The Source has broad latitude to decide how frequently it will emit state records. In this implementation it emits a state message for each new day of the created_at it processes.

In Sync 2, the last state that was emitted from Sync 1 is passed into the Source. When the Source queries the data, it knows that it has already replicated records from 2022/01/02 and before, so it does not resend them. It just emits records after that date.

While this example, demonstrates a success case, we can see how this process helps in failure cases as well. Let's say that in Sync 1 after emitting the first state message and before emitting the record for Carl, the Source lost connectivity with Postgres due to a network blip and the Source Actor crashed. When Sync 2 runs, it will get the state record with 2022/01/01 instead, so it will replicate Carl and Drew, but it skips Alice and Bob. While in this toy example this procedure only saves replicating one record, in a production use case, being able to checkpoint regularly can save having to resend huge amounts of data due to transient issues.

State & the Whole Sync

The previous section, for the sake of clarity, looked exclusively at the life cycle of state relative to the Source. In reality knowing that a record was emitted from the Source is NOT enough guarantee to know that we can skip sending the record in future syncs. For example, imagine the Source successfully emits the record, but the Destination fails. If we skip that record in the next sync, it means it never truly made it to its destination. This insight means, that a State should only ever be passed to a Source in the next run if it was both emitted from the Source and the Destination.



This image looks at two time points during an example sync. At T1 the Source has emitted 3 records and 2 state messages. If the Sync were to fail now, the next sync should start at the beginning because no records have been saved to the destination.

At T2, the Destination has received all records before the first state message and has emitted that that state message. By emitting that state message, the destination is confirming that all records in that state message have been committed. The diagram only shows the state being emitted because the destination does not emit record messages, only state messages. In addition, the Source has also emitted more records, including the record for Drew and another state message. If the sync were to fail at T2, then the next sync could start replicating records after Bob. Because the state records for Carl and Drew did not make it through to the destination, despite being emitted by the source, they have to be resent.

The normal success case (T3, not depicted) would be that all the records would move through the destination and the last state message that the Source emitted is then emitted by the Destination. The Source and Destination would both close `STDOUT` and `exit 0` signal that they have emitted all records without failure.

-- link to source image

V1

The state for an actor is emitted as a complete black box. When emitted it is wrapped in the AirbyteStateMessage. The contents of the `data` field is what is passed to the Source on start up. This gives the Source lead to decide how to track the state of each stream. That being said, a common pattern is a `Map<StreamDescriptor, StreamStateBlob>`. Nothing outside the source can make any inference about the state of the object EXCEPT, if it is null, it can be concluded that there is no state and the Source will start at the beginning.

V2 (coming soon!)

In addition to allowing a Source to checkpoint data replication, the state object is structure to allow for the ability to configure and reset streams in isolation from each other. For example, if adding or removing a stream, it is possible to do so without affecting the state of any other stream in the Source.

There are 3 types of state: Stream, Global, and Legacy.

- Stream represents Sources where there is complete isolation between stream states. In these cases, the state for each stream will be emitted in its own state message. In other words, if there are 3 streams replicated during a sync, the Source would emit at least 3 state message (1 per stream). The state of the Source is the sum of all the stream states.
- Global represents Sources where this shared state across streams. In these cases each state message contains the whole state for the connection. The `shared_state` field is where any information that is shared across streams must go. The `stream_states` field contains a list of objects that contain a

Stream Descriptor and the state information for that stream that is stream-specific. There are drawbacks to this state type, so it should only be used in cases where a shared state between streams is unavoidable.

- Legacy exists for backwards compatibility. In this state type, the state object is totally a black box. The only inference tha can be drawn from the state object is that if it is null, then there is no state for the entire Source. All current legacy cases can be ported to stream or global. Once they are, it will be removed.

This table breaks down attributes of these state types.

	Stream	**Global**	**Legacy**
Stream-Level Configuration / Reset	X	X	
Stream-Level Replication Isolation	X		
Single state message describes full state for Source		X	X

- Protocol Version simply connotes which versions of the Protocol have support for these State types. The new state message is backwards compatible with the V1 message. This allows old versions of connectors and platforms to interact with the new message.
- Stream-Level Configuration / Reset was mentioned above. The drawback of the old state struct was that it was not possible to configure or reset the state for a single stream without doing it for all of them. Thus, new state types support this, but the legacy one cannot.
- Stream-Level Replication Isolation means that a Source could be run in parallel by splitting up its streams across running instances. This is only possible for Stream state types, because they are the only state type that can update its current state completely on a per-stream basis. This is one of the main drawbacks of Sources that use Global state; it is not possible to increase their throughput through parallelization.
- Single state message describes full state for Source means that any state message contains the full state information for a Source. Stream does not meet this condition because each state message is scoped by stream. This means that in order to build a full picture of the state for the Source, the state messages for each configured stream must be gathered.

Messages

Common

For forwards compatibility all messages should allow for unknown properties (in JSONSchema parlance that is `additionalProperties: true`).

Messages are structs emitted by actors.

StreamDescriptor

A stream descriptor contains all information required to identify a Stream:

- The `name` of the stream (required). It may not be `null`.
- The `namespace` of the stream (optional). It may be `null` if the stream does not have an associated namespace, otherwise must be populated.
- Any UTF-8 string value is valid for both `name` and `namespace`, including the empty string (`""`) value.

This is the new pattern for referring to a stream. As structs are updated, they are moved ot use this pattern. Structs that have not been updated still refer to streams by having top-level fields called `stream_name` and `namespace`.

```
Unset

StreamDescriptor:
 type: object
 additionalProperties: true
 required:
   - name
 properties:
   name:
     type: string
```

```
    namespace:
      type: string
```

AirbyteMessage

The output of each method in the actor interface is wrapped in an `AirbyteMessage`. This struct is an envelope for the return value of any message in the described interface.

The envelope has a required `type` which described the type of the wrapped message. Based on the type only the field of that type will be populated. All other fields will be null.

```
Unset

AirbyteMessage:
 type: object
 additionalProperties: true
 required:
   - type
 properties:
   type:
     description: "Message type"
     type: string
     enum:
       - RECORD
       - STATE
```

```
        - LOG
        - SPEC
        - CONNECTION_STATUS
        - CATALOG
        - TRACE
    log:
      description: "log message: any kind of logging you
want the platform to know about."
      "$ref": "#/definitions/AirbyteLogMessage"
    spec:
      "$ref": "#/definitions/ConnectorSpecification"
    connectionStatus:
      "$ref": "#/definitions/AirbyteConnectionStatus"
    catalog:
      description: "catalog message: the catalog"
      "$ref": "#/definitions/AirbyteCatalog"
    record:
      description: "record message: the record"
      "$ref": "#/definitions/AirbyteRecordMessage"
    state:
      description: "schema message: the state. Must be the
last message produced. The platform uses this information"
      "$ref": "#/definitions/AirbyteStateMessage"
```

```
  trace:
    description: "trace message: a message to communicate
information about the status and performance of a
connector"
    "$ref": "#/definitions/AirbyteTraceMessage"
```

AirbyteRecordMessage

The record message contains the actual data that is being replicated.

The `namespace` and `stream` fields are used to identify which stream the data is associated with. `namespace` can be null if the stream does not have an associated namespace. If it does, it must be populated.

The `data` contains the record data and must always be populated. It is a JSON blob.

The `emitted_at` field contains when the source extracted the record. It is a required field.

Unset

```
AirbyteRecordMessage:
 type: object
 additionalProperties: true
 required:
   - stream
   - data
   - emitted_at
 properties:
   namespace:
     description: "namespace the data is associated with"
     type: string
   stream:
     description: "stream the data is associated with"
     type: string
   data:
     description: "record data"
     type: object
     existingJavaType:
com.fasterxml.jackson.databind.JsonNode
   emitted_at:
     description: "when the data was emitted from the
source. epoch in millisecond."
     type: integer
```

AirbyteStateMessage (V1)

The state message enables the Source to emit checkpoints while replicating data. These checkpoints mean that if replication fails before completion, the next sync is able to start from the last checkpoint instead of returning to the beginning of the previous sync. The details of this process are described in State & Checkpointing.

The state message is a wrapper around the state that a Source emits. The state that the Source emits is treated as a black box by the protocol--it is modeled as a JSON blob.

Unset

```
AirbyteStateMessage:
 type: object
 additionalProperties: true
 required:
   - data
 properties:
   data:
     description: "the state data"
     type: object
     existingJavaType:
com.fasterxml.jackson.databind.JsonNode
```

AirbyteStateMessage (V2 -- coming soon!)

The state message enables the Source to emit checkpoints while replicating data. These checkpoints mean that if replication fails before completion, the next sync is able to start from the last checkpoint instead of returning to the beginning of the previous sync. The details of this process are described in State & Checkpointing.

In the previous version of the protocol, the state object that the Source emitted was treated entirely as a black box. In the current version of protocol, Sources split up state by Stream. Within each Stream, the state is treated like a black box. The current version of the protocol is backwards compatible to the previous state message. The previous version is referred to as type `LEGACY` (if type is not set, it is assumed that the state message is `LEGACY`).

state_type is a required field. Only the field associated with that type we be populated. All others will be null. If the type is LEGACY and data is null, that means the state should be reset.

STREAM is the common way of constructing states and should be preferred wherever possible. In the STREAM case, the state for each stream is emitted in a separate message. This is described by the AirbyteStreamState struct. The stream_descriptor field is required to determine which stream a state is associated with. stream_state contains the black box state for a stream. If it is null, it means that the state for that stream should be reset.

In the GLOBAL case, the state for the whole Source is encapsulated in the message (see: AirbyteGlobalState). Within that message the state for individual streams is split. The GLOBAL case allows the author of a Source to share state across streams (shared_state). The contract is that if the state of the stream is set to null in stream_states then the next time the Source runs, it should treat that state as reset. This message should only be used in cases where there is a shared state across streams (e.g. CDC where the WAL log number is a global cursor), otherwise prefer STREAM.

```
Unset

AirbyteStateMessage:
 type: object
 additionalProperties: true
 properties:
   state_type:
     "$ref": "#/definitions/AirbyteStateType"
   stream:
     "$ref": "#/definitions/AirbyteStreamState"
   global:
     "$ref": "#/definitions/AirbyteGlobalState"

   data:

     description: "(Deprecated) the state data"

     type: object

     existingJavaType:
com.fasterxml.jackson.databind.JsonNode

AirbyteStateType:

 type: string

 description: >

   The type of state the other fields represent.

   Is set to LEGACY, the state data should be read from the
`data` field for backwards compatibility.

   If not set, assume the state object is type LEGACY.

   GLOBAL means that the state should be read from `global`
and means that it represents the state for all the streams.
It contains one shared

   state and individual stream states.

   PER_STREAM means that the state should be read from
`stream`. The state present in this field correspond to the
isolated state of the

   associated stream description.

 enum:

   - GLOBAL

   - STREAM
```

```
    - LEGACY
AirbyteStreamState:
  type: object
  additionalProperties: true
  required:
    - stream_descriptor
  properties:
    stream_descriptor:
      "$ref": "#/definitions/StreamDescriptor"
    stream_state:
      "$ref": "#/definitions/AirbyteStateBlob"
AirbyteGlobalState:
  type: object
  additionalProperties: true
  required:
    - stream_states
  properties:
    shared_state:
      "$ref": "#/definitions/AirbyteStateBlob"
    stream_states:
      type: array
      items:
```

```
      "$ref": "#/definitions/AirbyteStreamState"
```

AirbyteConnectionStatus Message

This message reports whether an Actor was able to connect to its underlying data store with all the permissions it needs to succeed. The goal is that if a successful stat is returned, that the user should be confident that using that Actor will succeed. The depth of the verification is not specified in the protocol. More robust verification is preferred but going to deep can create undesired performance tradeoffs

```
Unset

AirbyteConnectionStatus:
 description: Airbyte connection status
 type: object
 additionalProperties: true
 required:
   - status
 properties:
   status:
     type: string
     enum:
       - SUCCEEDED
       - FAILED
   message:
     type: string
```

ConnectorSpecification Message

This message returns the `ConnectorSpecification` struct which is described in detail in Actor Specification

AirbyteCatalog Message

This message returns the `AirbyteCatalog` struct which is described in detail in Catalog

AirbyteLogMessage

Logs are helping for debugging an Actor. In order for a log emitted from an Actor be properly parsed it should be emitted as an `AirbyteLogMessage` wrapped in an `AirbyteMessage`.

The Airbyte implementation of the protocol does attempt to parse any data emitted from an Actor as a log, even if it is not properly wrapped in an `AirbyteLogMessage`. It attempts to treat any returned line that is not JSON or that is JSON but is not an `AirbyteMessage` as a log. This an implementation choice outside the boundaries of the strict protocol. The downside of this approach is that metadata about the log that would be captured in the `AirbyteLogMessage` is lost.

```
Unset

AirbyteLogMessage:
 type: object
 additionalProperties: true
 required:
   - level
   - message
 properties:
   level:
     description: "log level"
     type: string
     enum:
       - FATAL
       - ERROR
       - WARN
       - INFO
       - DEBUG
       - TRACE
   message:
     description: "log message"
     type: string
   stack_trace:
     description: "an optional stack trace if the log
message corresponds to an exception"
     type: string
```

AirbyteTraceMessage

The trace message allows an Actor to emit metadata about the runtime of the Actor, such as errors or estimates. This message is designed to grow to handle other use cases, including additonal performance metrics.

```
Unset
```

```
AirbyteTraceMessage:
 type: object
 additionalProperties: true
 required:
   - type
   - emitted_at
 properties:
   type:
     title: "trace type" # this title is required to avoid
python codegen conflicts with the "type" parameter in
AirbyteMessage. See
https://github.com/airbytehq/airbyte/pull/12581
     description: "the type of trace message"
     type: string
     enum:
       - ERROR
       - ESTIMATE
   emitted_at:
     description: "the time in ms that the message was
emitted"
     type: number
   error:
     description: "error trace message: the error object"
```

```
      "$ref": "#/definitions/AirbyteErrorTraceMessage"
    estimate:
      description: "Estimate trace message: a guess at how
much data will be produced in this sync"
      "$ref": "#/definitions/AirbyteEstimateTraceMessage"
AirbyteErrorTraceMessage:
  type: object
  additionalProperties: true
  required:
    - message
  properties:
    message:
      description: A user-friendly message that indicates
the cause of the error
      type: string
    internal_message:
      description: The internal error that caused the
failure
      type: string
    stack_trace:
      description: The full stack trace of the error
      type: string
    failure_type:
      description: The type of error
      type: string
      enum:
        - system_error
        - config_error
 AirbyteEstimateTraceMessage:
  type: object
  additionalProperties: true
  required:
    - name
    - type
  properties:
    name:
      description: The name of the stream
      type: string
    type:
      description: The type of estimate
      type: string
      enum:
        - STREAM
        - SYNC
    namespace:
      description: The namespace of the stream

      type: string

    row_estimate:

      description: The estimated number of rows to be
emitted by this sync for this stream

      type: integer

    byte_estimate:

      description: The estimated number of bytes to be
emitted by this sync for this stream

      type: integer
```

AirbyteErrorTraceMessage

Error Trace Messages are used when a sync is about to fail and the connector can provide meaningful information to the orhcestrator or user about what to do next.

Of note, an `internal_message` might be an exception code, but an `external_message` is meant to be user-facing, e.g. "Your API Key is invalid".

Syncs can fail for multiple reasons, and therefore multiple `AirbyteErrorTraceMessage` can be sent from a connector.

AirbyteEstimateTraceMessage

Estimate Trace Messages are used by connectors to inform the orchestrator about how much data they expect to move within the sync. This ise useful to present the user with estimates of the time remaining in the sync, or percentage complete. An example of this would be for every stream about to be synced from a databse to provde a `COUNT (*) from {table_name} where updated_at > {state}` to provide an estimate of the rows to be sent in this sync.

`AirbyteEstimateTraceMessage` should be emitted early in the sync to provide an early estimate of the sync's duration. Multiple `AirbyteEstimateTraceMessage`s can be sent for the same stream, and an updated estimate will replace the previous value.

AirbyteControlMessage

An `AirbyteControlMessage` is for connectors to signal to the Airbyte Platform or Orchestrator that an action with a side-effect should be taken. This means that the Orchestrator will likely be altering some stored data about the connector, connection, or sync.

```
Unset


AirbyteControlMessage:
 type: object
 additionalProperties: true
 required:
   - type
   - emitted_at
 properties:
   type:
     title: orchestrator type
     description: "the type of orchestrator message"
     type: string
     enum:
       - CONNECTOR_CONFIG
   emitted_at:
     description: "the time in ms that the message was
emitted"
     type: number
   connectorConfig:
     description: "connector config orchestrator message:
the updated config for the platform to store for this
connector"
     "$ref":
"#/definitions/AirbyteControlConnectorConfigMessage"
```

AirbyteControlConnectorConfigMessage

`AirbyteControlConnectorConfigMessage` allows a connector to update its configuration in the middle of a sync. This is valuable for connectors with short-lived or single-use credentials.

Emitting this message signals to the orchestrator process that it should update its persistence layer, replacing the connector's current configuration with the config present in the `.config` field of the message.

The config in the `AirbyteControlConnectorConfigMessage` must conform to connector's specification's schema, and the orchestrator process is expected to validate these messages. If the output config does not conform to the specification's schema, the orchestrator process should raise an exception and terminate the sync.

```
Unset

AirbyteControlConnectorConfigMessage:
 type: object
 additionalProperties: true
 required:
   - config
 properties:
   config:
     description: "the config items from this connector's
spec to update"
     type: object
     additionalProperties: true
```

For example, if the currently persisted config file is `{"api_key": 123, start_date: "01-01-2022"}` and the following `AirbyteControlConnectorConfigMessage` is output `{type: ORCHESTRATOR, connectorConfig: {"config": {"api_key": 456}, "emitted_at": <current_time>}}` then the persisted configuration is merged, and will become `{"api_key": 456, start_date: "01-01-2022"}`.

Acknowledgements

We'd like to note that we were initially inspired by Singer.io's specification and would like to acknowledge that some of their design choices helped us bootstrap our project. We've since made a lot of modernizations to our protocol and specification, but don't want to forget the tools that helped us get started.

Namespaces

High-Level Overview

INFO

The high-level overview contains all the information you need to use Namespaces when pulling from APIs. Information past that can be read for advanced or educational purposes.

When looking through our connector docs, you'll notice that some sources and destinations support "Namespaces." These allow you to organize and separate your data into groups in the destination if the destination supports it. In most cases, namespaces are schemas in the database you're replicating to. If your desired destination doesn't support it, you can ignore this feature.

Note that this is the location that both your normalized and raw data will get written to. Your raw data will show up with the prefix `_airbyte_raw_` in the namespace you define. If you don't enable basic normalization, you will only receive the raw tables.

If only your destination supports namespaces, you have two simple options. This is the most likely case, as all HTTP APIs currently don't support Namespaces.

1. Mirror Destination Settings - Replicate to the default namespace in the destination, which will differ based on your destination.
2. Custom Format - Create a "Custom Format" to rename the namespace that your data will be replicated into.

If both your desired source and destination support namespaces, you're likely using a more advanced use case with a database as a source, so continue reading.

What is a Namespace?

Technical systems often group their underlying data into namespaces with each namespace's data isolated from another namespace. This isolation allows for better organisation and flexibility, leading to better usability.

An example of a namespace is the RDMS's `schema` concept. Some common use cases for schemas are enforcing permissions, segregating test and production data and general data organisation.

Syncing

The Airbyte Protocol supports namespaces and allows Sources to define namespaces, and Destinations to write to various namespaces.

If the Source does not support namespaces, the data will be replicated into the Destination's default namespace. For databases, the default namespace is the schema provided in the destination configuration.

If the Destination does not support namespaces, the namespace field is ignored.

Destination namespace configuration

As part of the connections sync settings, it is possible to configure the namespace used by: 1. destination connectors: to store the `_airbyte_raw_*` tables. 2. basic normalization: to store the final normalized tables.

Note that custom transformation outputs are not affected by the namespace settings from Airbyte: It is up to the configuration of the custom dbt project, and how it is written to handle its custom schemas. The default target schema for dbt in this case, will always be the destination namespace.

Available options for namespace configurations are:

- Mirror source structure

Some sources (such as databases based on JDBC for example) are providing namespace information from which a stream has been extracted. Whenever a source is able to fill this field in the catalog.json file, the destination will try to reproduce exactly the same namespace when this configuration is set. For sources or streams where the source namespace is not known, the behavior will fall back to the "Destination Connector settings".

- Destination connector settings

All stream will be replicated and store in the default namespace defined on the destination settings page. In the destinations, namespace refers to:

Destination Connector	Namespace setting
BigQuery	dataset
MSSQL	schema
MySql	database
Oracle DB	schema
Postgres	schema
Redshift	schema
Snowflake	schema
S3	path prefix

- Custom format

When replicating multiple sources into the same destination, conflicts on tables being overwritten by syncs can occur.

For example, a Github source can be replicated into a "github" schema. But if we have multiple connections to different GitHub repositories (similar in multi-tenant scenarios):

- we'd probably wish to keep the same table names (to keep consistent queries downstream)
- but store them in different namespaces (to avoid mixing data from different "tenants")

To solve this, we can either:

- use a specific namespace for each connection, thus this option of custom format.
- or, use prefix to stream names as described below.

Note that we can use a template format string using variables that will be resolved during replication as follow:

- `${SOURCE_NAMESPACE}`: will be replaced by the namespace provided by the source if available

Examples

The following table summarises how this works. We assume an example of replication configurations between a Postgres Source and Snowflake Destination (with settings of schema = "my_schema"):

Namespace Configuration	Source Namespace	Source Table Name	Destination Namespace	Destination Table Name
Mirror source structure	public	my_table	public	my_table
Mirror source structure		my_table	my_schema	my_table
Destination connector settings	public	my_table	my_schema	my_table
Destination connector settings		my_table	my_schema	my_table
Custom format = "custom"	public	my_table	custom	my_table
Custom format = "${SOURCE_NAMESPACE}"	public	my_table	public	my_table
Custom format = "my_${SOURCE_NAMESPACE}_schema"	public	my_table	my_public_schema	my_table
Custom format = " "	public	my_table	my_schema	my_table

Requirements

- Both Source and Destination connectors need to support namespaces.
- Relevant Source and Destination connectors need to be at least version `0.3.0` or later.
- Airbyte version `0.21.0-alpha` or later.

Current Support

Sources

- MSSQL
- MYSQL
- Oracle DB
- Postgres
- Redshift

Destination

- BigQuery
- MSSQL
- MySql
- Oracle DB
- Postgres
- Redshift
- Snowflake
- S3

BigQuery (Destination)

Setting up the BigQuery destination connector involves setting up the data loading method (BigQuery Standard method and Google Cloud Storage bucket) and configuring the BigQuery destination connector using the Airbyte UI.

This page guides you through setting up the BigQuery destination connector.

Prerequisites

- For Airbyte Open Source users using the Postgres source connector, upgrade your Airbyte platform to version `v0.40.0-alpha` or newer and upgrade your BigQuery connector to version `1.1.14` or newer
- A Google Cloud project with BigQuery enabled

- A BigQuery dataset to sync data to.
  Note: Queries written in BigQuery can only reference datasets in the same physical location. If you plan on combining the data that Airbyte syncs with data from other datasets in your queries, create the datasets in the same location on Google Cloud. For more information, read Introduction to Datasets
- (Required for Airbyte Cloud; Optional for Airbyte Open Source) A Google Cloud Service Account with the `BigQuery User` and `BigQuery Data Editor` roles and the Service Account Key in JSON format.

Connector modes

While setting up the connector, you can configure it in the following modes:

- BigQuery: Produces a normalized output by storing the JSON blob data in `_airbyte_raw_*` tables and then transforming and normalizing the data into separate tables, potentially `exploding` nested streams into their own tables if basic normalization is configured.
- BigQuery (Denormalized): Leverages BigQuery capabilities with Structured and Repeated fields to produce a single "big" table per stream. Airbyte does not support normalization for this option at this time.

Setup guide

Step 1: Set up a data loading method

Although you can load data using BigQuery's `INSERTS`, we highly recommend using a Google Cloud Storage bucket not only for performance and cost but reliability since larger datasets are prone to more failures when using standard inserts.

(Recommended) Using a Google Cloud Storage bucket

To use a Google Cloud Storage bucket:

1. Create a Cloud Storage bucket with the Protection Tools set to `none` or `Object versioning`. Make sure the bucket does not have a retention policy.
2. Create an HMAC key and access ID.
3. Grant the `Storage Object Admin` role to the Google Cloud Service Account.

4. Make sure your Cloud Storage bucket is accessible from the machine running Airbyte. The easiest way to verify if Airbyte is able to connect to your bucket is via the check connection tool in the UI.

Your bucket must be encrypted using a Google-managed encryption key (this is the default setting when creating a new bucket). We currently do not support buckets using customer-managed encryption keys (CMEK). You can view this setting under the "Configuration" tab of your GCS bucket, in the `Encryption type` row.

Using `INSERT`

You can use BigQuery's `INSERT` statement to upload data directly from your source to BigQuery. While this is faster to set up initially, we strongly recommend not using this option for anything other than a quick demo. Due to the Google BigQuery SDK client limitations, using `INSERT` is 10x slower than using a Google Cloud Storage bucket, and you may see some failures for big datasets and slow sources (For example, if reading from a source takes more than 10-12 hours). For more details, refer to https://github.com/airbytehq/airbyte/issues/3549

Step 2: Set up the BigQuery connector

1. Log into your Airbyte Cloud or Airbyte Open Source account.
2. Click Destinations and then click + New destination.
3. On the Set up the destination page, select BigQuery or BigQuery (denormalized typed struct) from the Destination type dropdown depending on whether you want to set up the connector in BigQuery or BigQuery (Denormalized) mode.
4. Enter the name for the BigQuery connector.
5. For Project ID, enter your Google Cloud project ID.
6. For Dataset Location, select the location of your BigQuery dataset.
   DANGER
   You cannot change the location later.
7. For Default Dataset ID, enter the BigQuery Dataset ID.
8. For Loading Method, select Standard Inserts or GCS Staging.
   TIP
   We recommend using the GCS Staging option.
9. For Service Account Key JSON (Required for cloud, optional for open-source), enter the Google Cloud Service Account Key in JSON format.
10. For Transformation Query Run Type (Optional), select interactive to have BigQuery run interactive query jobs or batch to have BigQuery run batch queries.
    NOTE

Interactive queries are executed as soon as possible and count towards daily concurrent quotas and limits, while batch queries are executed as soon as idle resources are available in the BigQuery shared resource pool. If BigQuery hasn't started the query within 24 hours, BigQuery changes the job priority to interactive. Batch queries don't count towards your concurrent rate limit, making it easier to start many queries at once.

11. For Google BigQuery Client Chunk Size (Optional), use the default value of 15 MiB. Later, if you see networking or memory management problems with the sync (specifically on the destination), try decreasing the chunk size. In that case, the sync will be slower but more likely to succeed.

Supported sync modes

The BigQuery destination connector supports the following sync modes:

- Full Refresh Sync
- Incremental - Append Sync
- Incremental - Deduped History

Output schema

Airbyte outputs each stream into its own table in BigQuery. Each table contains three columns:

- `_airbyte_ab_id`: A UUID assigned by Airbyte to each event that is processed. The column type in BigQuery is `String`.
- `_airbyte_emitted_at`: A timestamp representing when the event was pulled from the data source. The column type in BigQuery is `Timestamp`.
- `_airbyte_data`: A JSON blob representing the event data. The column type in BigQuery is `String`.

The output tables in BigQuery are partitioned and clustered by the Time-unit column `_airbyte_emitted_at` at a daily granularity. Partitions boundaries are based on UTC time. This is useful to limit the number of partitions scanned when querying these partitioned tables, by using a predicate filter (a `WHERE` clause). Filters on the partitioning column are used to prune the partitions and reduce the query cost. (The parameter Require partition filter is not enabled by Airbyte, but you may toggle it by updating the produced tables.)

BigQuery Naming Conventions

Follow BigQuery Datasets Naming conventions.

Airbyte converts any invalid characters into _ characters when writing data. However, since datasets that begin with _ are hidden on the BigQuery Explorer panel, Airbyte prepends the namespace with n for converted namespaces.

Data type map

Airbyte type	BigQuery type	BigQuery denormalized type
DATE	DATE	DATE
STRING (BASE64)	STRING	STRING
NUMBER	FLOAT	NUMBER
OBJECT	STRING	RECORD
STRING	STRING	STRING
BOOLEAN	BOOLEAN	BOOLEAN
INTEGER	INTEGER	INTEGER
STRING (BIG_NUMBER)	STRING	STRING
STRING (BIG_INTEGER)	STRING	STRING
ARRAY	REPEATED	REPEATED
STRING (TIMESTAMP_WITH_TIMEZONE)	TIMESTAMP	DATETIME
STRING (TIMESTAMP_WITHOUT_TIMEZONE)	TIMESTAMP	DATETIME

Troubleshooting permission issues

The service account does not have the proper permissions.

- Make sure the BigQuery service account has `BigQuery User` and `BigQuery Data Editor` roles or equivalent permissions as those two roles.
- If the GCS staging mode is selected, ensure the BigQuery service account has the right permissions to the GCS bucket and path or the `Cloud Storage Admin` role, which includes a superset of the required permissions.

The HMAC key is wrong.

- Make sure the HMAC key is created for the BigQuery service account, and the service account has permission to access the GCS bucket and path.

Local CSV

DANGER

This destination is meant to be used on a local workstation and won't work on Kubernetes

Overview

This destination writes data to a directory on the *local* filesystem on the host running Airbyte. By default, data is written to `/tmp/airbyte_local`. To change this location, modify the `LOCAL_ROOT` environment variable for Airbyte.
CAUTION

Please make sure that Docker Desktop has access to `/tmp` (and `/private` on a MacOS, as /tmp has a symlink that points to /private. It will not work otherwise). You allow it with "File sharing" in `Settings -> Resources -> File sharing -> add the one or two above folder` and hit the "Apply & restart" button.

Sync Overview

Output schema

Each stream will be output into its own file. Each file will contain 3 columns:

- `_airbyte_ab_id`: a uuid assigned by Airbyte to each event that is processed.
- `_airbyte_emitted_at`: a timestamp representing when the event was pulled from the data source.
- `_airbyte_data`: a json blob representing with the event data.

Features

Feature	Supported	
Full Refresh Sync	Yes	
Incremental - Append Sync	Yes	
Incremental - Deduped History	No	As this connector does not support dbt, we don't support this sync mode on this destination.
Namespaces	No	

Performance considerations

This integration will be constrained by the speed at which your filesystem accepts writes.

Getting Started

The `destination_path` will always start with `/local` whether it is specified by the user or not. Any directory nesting within local will be mapped onto the local mount.

By default, the `LOCAL_ROOT` env variable in the `.env` file is set `/tmp/airbyte_local`.

The local mount is mounted by Docker onto `LOCAL_ROOT`. This means the `/local` is substituted by `/tmp/airbyte_local` by default.

Example:

- If `destination_path` is set to `/local/cars/models`
- the local mount is using the `/tmp/airbyte_local` default
- then all data will be written to `/tmp/airbyte_local/cars/models` directory.

Access Replicated Data Files

If your Airbyte instance is running on the same computer that you are navigating with, you can open your browser and enter file:///tmp/airbyte_local to look at the replicated data locally. If the first approach fails or if your Airbyte instance is running on a remote server, follow the following steps to access the replicated files:

1. Access the scheduler container using `docker exec -it airbyte-server bash`
2. Navigate to the default local mount using `cd /tmp/airbyte_local`
3. Navigate to the replicated file directory you specified when you created the destination, using `cd /{destination_path}`
4. List files containing the replicated data using `ls`
5. Execute `cat {filename}` to display the data in a particular file

You can also copy the output file to your host machine, the following command will copy the file to the current working directory you are using:

Unset

```
docker cp
airbyte-server:/tmp/airbyte_local/{destination_path}/{filen
ame}.csv .
```

Note: If you are running Airbyte on Windows with Docker backed by WSL2, you have to use a similar step as above or refer to this link for an alternative approach.

Windows - Browsing Local File Output

Overview

This tutorial will describe how to look for json and csv files in when using local destinations on Windows on a local deployment.

There can be confusion when using local destinations in Airbyte on Windows, especially if you are running WSL2 to power Docker. There are also two folders generated at the root folder of your Docker folder which will point you in the wrong direction.

Locating where your temp folder is

While running Airbyte's Docker image on Windows with WSL2, you can access your temp folder by doing the following:

1. Open File Explorer (Or any folder where you can access the address bar)
2. Type in `\\wsl$` in the address bar
3. The folders below will be displayed

docker-desktop
docker-desktop-data

4. You can start digging here, but it is recommended to start searching from here and just search for the folder name you used for your local files. The folder address should be similar to `\\wsl$\docker-desktop\tmp\docker-desktop-root\containers\services\docker\rootfs\tmp\airbyte_local`
5. You should be able to locate your local destination CSV or JSON files in this folder.

Note that there are scenarios where you may not be able to browse to the actual files in which case, use the below method to take a local copy.

Use Docker to Copy your temp folder files

Note that this method does not allow direct access to any files directly, instead it creates local, readable copies.

1. Open and standard CMD shell
2. Type the following (where `<local path>` is the path on your Windows host machine to place copies) `docker cp airbyte-server:/tmp/airbyte_local <local path>`
3. This will copy the entire `airbyte_local` folder to your host machine.

Note that if you know the specific filename or wildcard, you can add append it to the source path of the `docker cp` command.

Notes

1. Local JSON and Local CSV files do not persist between Docker restarts. This means that once you turn off your Docker image, your data is lost. This is consistent with the `tmp` nature of the folder.
2. In the root folder of your docker files, it might generate tmp and var folders that only have empty folders inside.

Transformations with SQL (Part 1/3)

Transformations with SQL (Part 1/3)

Overview

This tutorial will describe how to integrate SQL based transformations with Airbyte syncs using plain SQL queries.

This is the first part of ELT tutorial. The second part goes deeper with Transformations with dbt and then wrap-up with a third part on Transformations with Airbyte.

(Examples outputs are updated with Airbyte version 0.23.0-alpha from May 2021)

First transformation step: Normalization

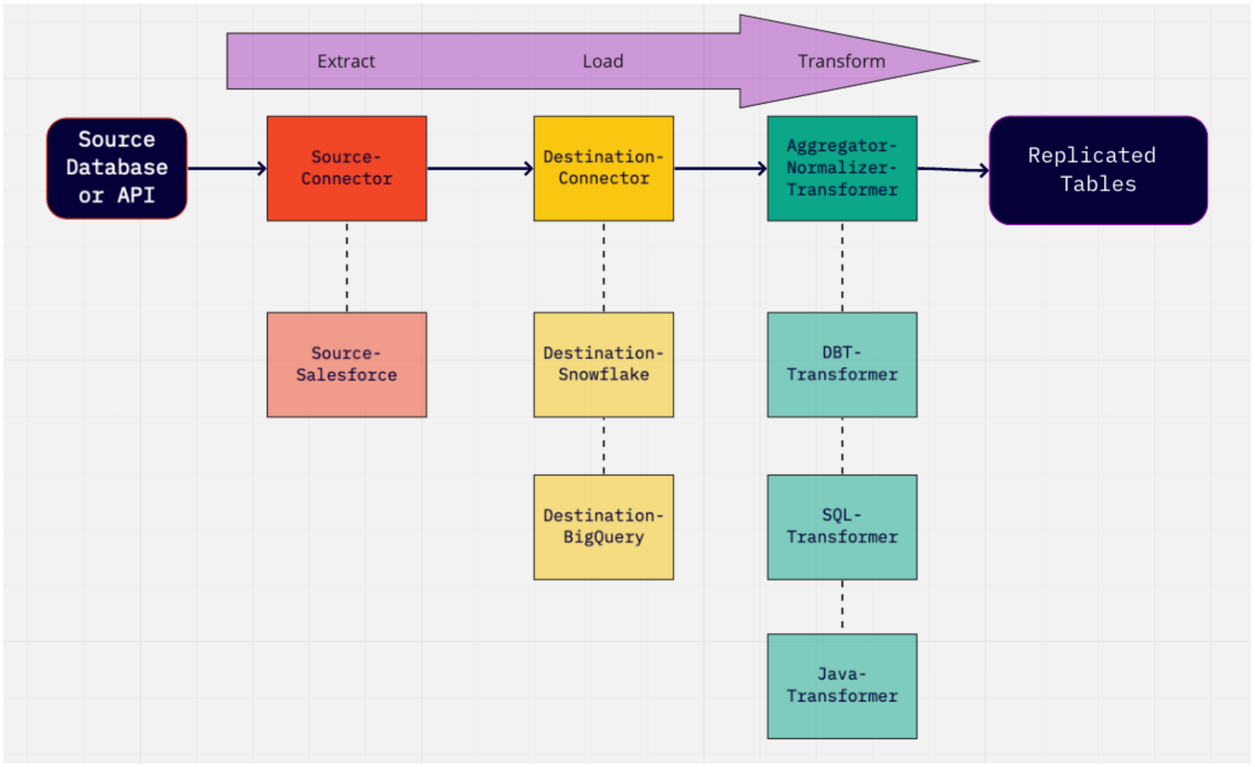
At its core, Airbyte is geared to handle the EL (Extract Load) steps of an ELT process. These steps can also be referred in Airbyte's dialect as "Source" and "Destination".

However, this is actually producing a table in the destination with a JSON blob column... For the typical analytics use case, you probably want this json blob normalized so that each field is its own column.

So, after EL, comes the T (transformation) and the first T step that Airbyte actually applies on top of the extracted data is called "Normalization". You can find more information about it here.

Airbyte runs this step before handing the final data over to other tools that will manage further transformation down the line.

To summarize, we can represent the ELT process in the diagram below. These are steps that happens between your "Source Database or API" and the final "Replicated Tables" with examples of implementation underneath:



Anyway, it is possible to short-circuit this process (no vendor lock-in) and handle it yourself by turning this option off in the destination settings page.

This could be useful if:

1. You have a use-case not related to analytics that could be handled with data in its raw JSON format.
2. You can implement your own transformer. For example, you could write them in a different language, create them in an analytics engine like Spark, or use a transformation tool such as dbt or Dataform.
3. You want to customize and change how the data is normalized with your own queries.

In order to do so, we will now describe how you can leverage the basic normalization outputs that Airbyte generates to build your own transformations if you don't want to start from scratch.

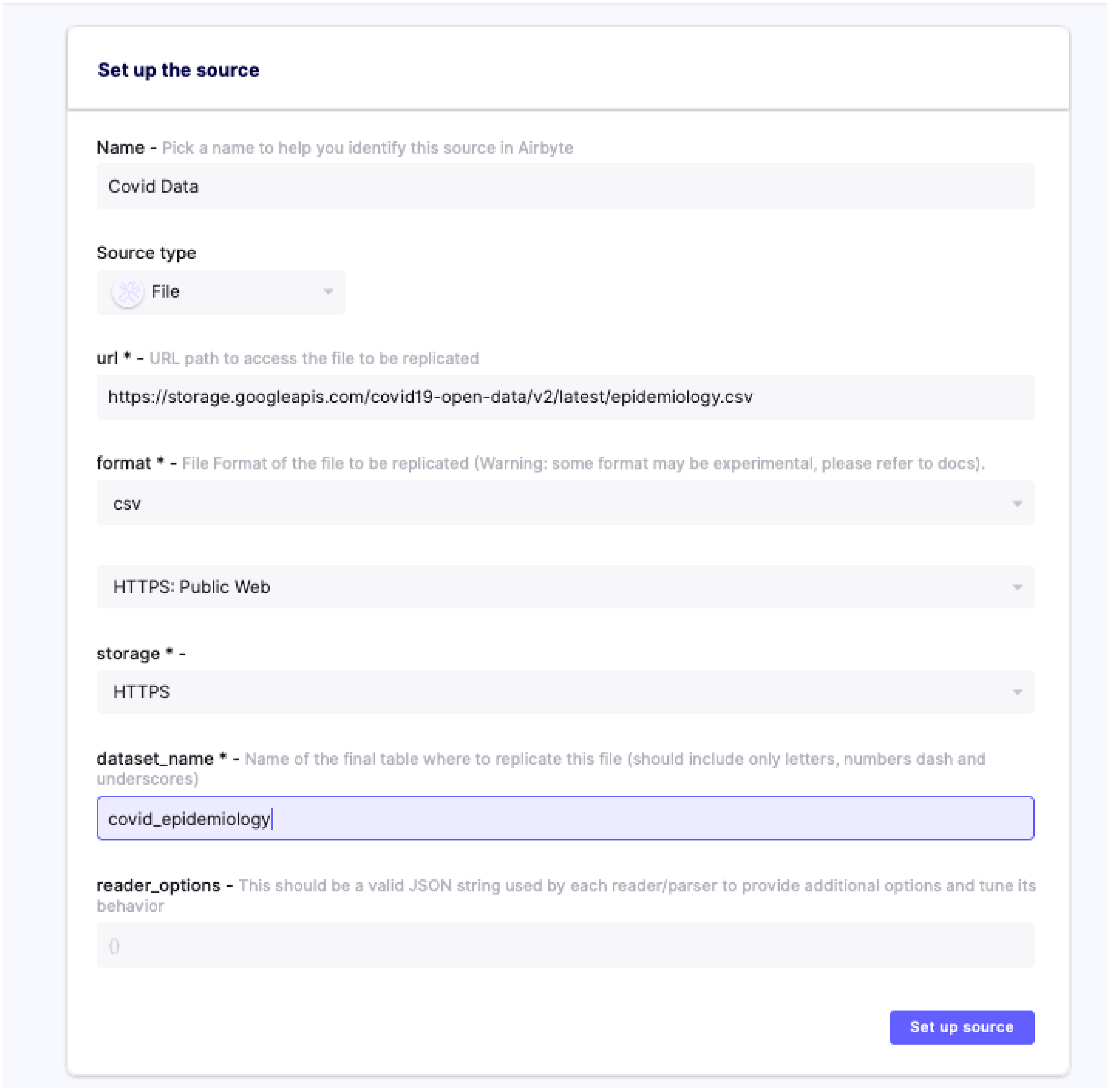
Note: We will rely on docker commands that we've gone over as part of another Tutorial on Exploring Docker Volumes.

(Optional) Configure some Covid (data) source and Postgres destinations

If you have sources and destinations already setup on your deployment, you can skip to the next section.

For the sake of this tutorial, let's create some source and destination as an example that we can refer to afterward. We'll be using a file accessible from a public API, so you can easily reproduce this setup:

```Unset
Here are some examples of public API CSV:

https://storage.googleapis.com/covid19-open-data/v2/latest/epidemiology.csv
```


Set up the source

Name - Pick a name to help you identify this source in Airbyte

Covid Data

Source type

File

url * - URL path to access the file to be replicated

https://storage.googleapis.com/covid19-open-data/v2/latest/epidemiology.csv

format * - File Format of the file to be replicated (Warning: some format may be experimental, please refer to docs).

csv

HTTPS: Public Web

storage * -

HTTPS

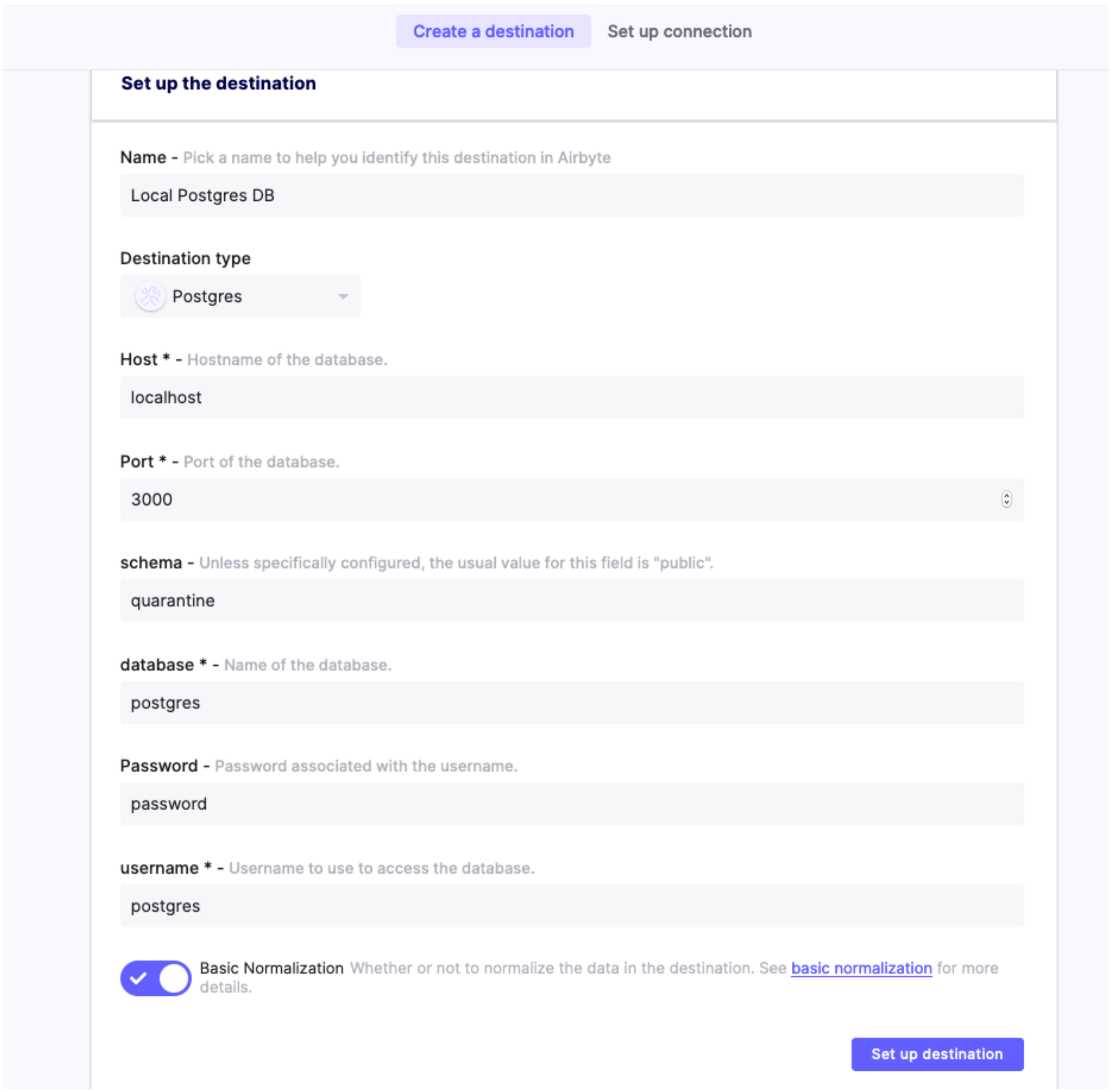
dataset_name * - Name of the final table where to replicate this file (should include only letters, numbers dash and underscores)

covid_epidemiology

reader_options - This should be a valid JSON string used by each reader/parser to provide additional options and tune its behavior

{}

And a local Postgres Database, making sure that "Basic normalization" is enabled:



Set up the destination

Name - Pick a name to help you identify this destination in Airbyte
Local Postgres DB

Destination type
Postgres

Host * - Hostname of the database.
localhost

Port * - Port of the database.
3000

schema - Unless specifically configured, the usual value for this field is "public".
quarantine

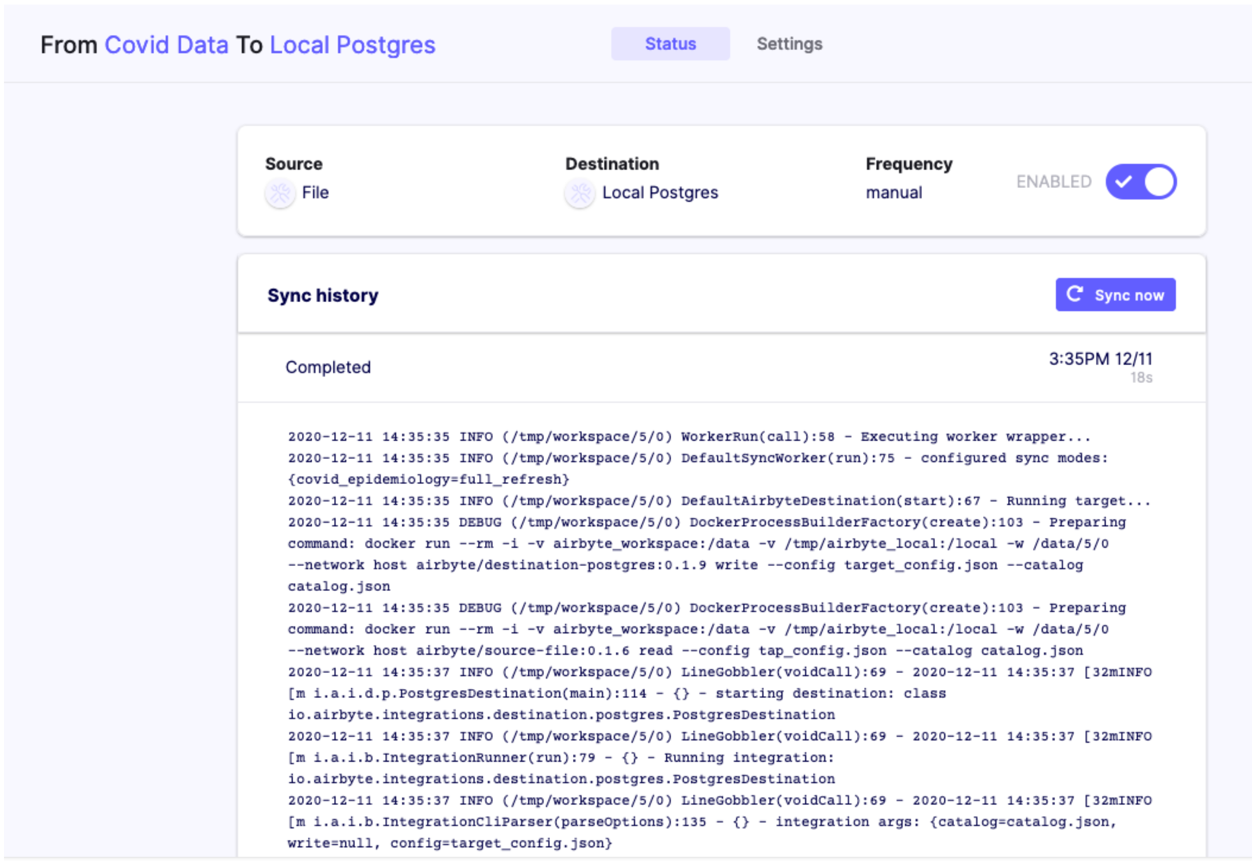
database * - Name of the database.
postgres

Password - Password associated with the username.
password

username * - Username to use to access the database.
postgres

Basic Normalization Whether or not to normalize the data in the destination. See basic normalization for more details.

After setting up the connectors, we can trigger the sync and study the logs:



```
2020-12-11 14:35:35 INFO (/tmp/workspace/5/0) WorkerRun(call):58 - Executing worker wrapper...
2020-12-11 14:35:35 INFO (/tmp/workspace/5/0) DefaultSyncWorker(run):75 - configured sync modes:
{covid_epidemiology=full_refresh}
2020-12-11 14:35:35 INFO (/tmp/workspace/5/0) DefaultAirbyteDestination(start):67 - Running target...
2020-12-11 14:35:35 DEBUG (/tmp/workspace/5/0) DockerProcessBuilderFactory(create):103 - Preparing
command: docker run --rm -i -v airbyte_workspace:/data -v /tmp/airbyte_local:/local -w /data/5/0
--network host airbyte/destination-postgres:0.1.9 write --config target_config.json --catalog
catalog.json
2020-12-11 14:35:35 DEBUG (/tmp/workspace/5/0) DockerProcessBuilderFactory(create):103 - Preparing
command: docker run --rm -i -v airbyte_workspace:/data -v /tmp/airbyte_local:/local -w /data/5/0
--network host airbyte/source-file:0.1.6 read --config tap_config.json --catalog catalog.json
2020-12-11 14:35:37 INFO (/tmp/workspace/5/0) LineGobbler(voidCall):69 - 2020-12-11 14:35:37 [32mINFO
[m i.a.i.d.p.PostgresDestination(main):114 - {} - starting destination: class
io.airbyte.integrations.destination.postgres.PostgresDestination
2020-12-11 14:35:37 INFO (/tmp/workspace/5/0) LineGobbler(voidCall):69 - 2020-12-11 14:35:37 [32mINFO
[m i.a.i.b.IntegrationRunner(run):79 - {} - Running integration:
io.airbyte.integrations.destination.postgres.PostgresDestination
2020-12-11 14:35:37 INFO (/tmp/workspace/5/0) LineGobbler(voidCall):69 - 2020-12-11 14:35:37 [32mINFO
[m i.a.i.b.IntegrationCliParser(parseOptions):135 - {} - integration args: {catalog=catalog.json,
write=null, config=target_config.json}
```

Notice that the process ran in the `/tmp/workspace/5/0` folder.

Identify Workspace ID with Normalize steps

If you went through the previous setup of source/destination section and run a sync, you were able to identify which workspace was used, let's define some environment variables to remember this:

```
Unset


NORMALIZE_WORKSPACE="5/0/"
```

Or if you want to find any folder where the normalize step was run:

```
Unset

# find automatically latest workspace where normalization was run

NORMALIZE_WORKSPACE=`docker run --rm -i -v airbyte_workspace:/data  busybox find /data -path "*normalize/models*" | sed -E "s;/data/([0-9]+/[0-9]+/)normalize/.*;\1;g" | sort | uniq | tail -n 1`
```

Export Plain SQL files

Airbyte is internally using a specialized tool for handling transformations called dbt.

The Airbyte Python module reads the `destination_catalog.json` file and generates dbt code responsible for interpreting and transforming the raw data.

The final output of dbt is producing SQL files that can be run on top of the destination that you selected.

Therefore, it is possible to extract these SQL files, modify them and run it yourself manually outside Airbyte!

You would be able to find these at the following location inside the server's docker container:

```
Unset

/tmp/workspace/${NORMALIZE_WORKSPACE}/build/run/airbyte_utils/models/generated/airbyte_tables/<schema>/<your_table_name>.sql
```

In order to extract them, you can run:

Unset

```
#!/usr/bin/env bash
docker cp
airbyte-server:/tmp/workspace/${NORMALIZE_WORKSPACE}/build/
run/airbyte_utils/models/generated/ models/


find models
```

Example Output:

Unset

```
models/airbyte_tables/quarantine/covid_epidemiology_f11.sql
```

Let's inspect the generated SQL file by running:

Unset

```
cat models/**/covid_epidemiology*.sql
```

Example Output:

Unset

```
create  table
"postgres".quarantine."covid_epidemiology_f11__dbt_tmp"
 as (
with __dbt__CTE__covid_epidemiology_ab1_558 as (


-- SQL model to parse JSON blob stored in a single column and
extract into separated field columns as described by the JSON
Schema

select

   jsonb_extract_path_text(_airbyte_data, 'key') as "key",

   jsonb_extract_path_text(_airbyte_data, 'date') as
"date",

   jsonb_extract_path_text(_airbyte_data, 'new_tested') as
new_tested,

   jsonb_extract_path_text(_airbyte_data, 'new_deceased')
as new_deceased,

   jsonb_extract_path_text(_airbyte_data, 'total_tested')
as total_tested,

   jsonb_extract_path_text(_airbyte_data, 'new_confirmed')
as new_confirmed,

   jsonb_extract_path_text(_airbyte_data, 'new_recovered')
as new_recovered,

   jsonb_extract_path_text(_airbyte_data, 'total_deceased')
as total_deceased,

   jsonb_extract_path_text(_airbyte_data,
'total_confirmed') as total_confirmed,

   jsonb_extract_path_text(_airbyte_data,
'total_recovered') as total_recovered,
   _airbyte_emitted_at
from "postgres".quarantine._airbyte_raw_covid_epidemiology
-- covid_epidemiology
),  __dbt__CTE__covid_epidemiology_ab2_558 as (

-- SQL model to cast each column to its adequate SQL type
converted from the JSON schema type
select
   cast("key" as
   varchar
) as "key",
   cast("date" as
   varchar
) as "date",
   cast(new_tested as
   float
) as new_tested,
   cast(new_deceased as
   float
) as new_deceased,
   cast(total_tested as
   float
) as total_tested,
   cast(new_confirmed as
   float
) as new_confirmed,
   cast(new_recovered as
   float
) as new_recovered,
   cast(total_deceased as
   float
) as total_deceased,
   cast(total_confirmed as
   float
) as total_confirmed,
   cast(total_recovered as
   float
) as total_recovered,
   _airbyte_emitted_at
from __dbt__CTE__covid_epidemiology_ab1_558
-- covid_epidemiology
),  __dbt__CTE__covid_epidemiology_ab3_558 as (
-- SQL model to build a hash column based on the values of
this record

select

   *,

   md5(cast(


   coalesce(cast("key" as

   varchar

), '') || '-' || coalesce(cast("date" as

   varchar

), '') || '-' || coalesce(cast(new_tested as

   varchar

), '') || '-' || coalesce(cast(new_deceased as

   varchar

), '') || '-' || coalesce(cast(total_tested as

   varchar

), '') || '-' || coalesce(cast(new_confirmed as

   varchar

), '') || '-' || coalesce(cast(new_recovered as

   varchar

), '') || '-' || coalesce(cast(total_deceased as

   varchar
), '') || '-' || coalesce(cast(total_confirmed as
   varchar
), '') || '-' || coalesce(cast(total_recovered as
   varchar
), '')

as
   varchar
)) as _airbyte_covid_epidemiology_hashid
from __dbt__CTE__covid_epidemiology_ab2_558
-- covid_epidemiology
)-- Final base SQL model
select
   "key",
   "date",
   new_tested,
   new_deceased,
   total_tested,
   new_confirmed,
   new_recovered,
   total_deceased,
   total_confirmed,
    total_recovered,
    _airbyte_emitted_at,
    _airbyte_covid_epidemiology_hashid
from __dbt__CTE__covid_epidemiology_ab3_558
-- covid_epidemiology from
"postgres".quarantine._airbyte_raw_covid_epidemiology
 );
```

Simple SQL Query

We could simplify the SQL query by removing some parts that may be unnecessary for your current usage (such as generating a md5 column; [Why exactly would I want to use that?!](#)).

It would turn into a simpler query:

```
Unset

create table "postgres"."public"."covid_epidemiology"
as (
    select
        _airbyte_emitted_at,
        (current_timestamp at time zone 'utc')::timestamp as
_airbyte_normalized_at,

        cast(jsonb_extract_path_text("_airbyte_data",'key')
as varchar) as "key",
       cast(jsonb_extract_path_text("_airbyte_data",'date')
as varchar) as "date",

cast(jsonb_extract_path_text("_airbyte_data",'new_tested')
as float) as new_tested,

cast(jsonb_extract_path_text("_airbyte_data",'new_deceased'
) as float) as new_deceased,

cast(jsonb_extract_path_text("_airbyte_data",'total_tested'
) as float) as total_tested,

cast(jsonb_extract_path_text("_airbyte_data",'new_confirmed
') as float) as new_confirmed,

cast(jsonb_extract_path_text("_airbyte_data",'new_recovered
') as float) as new_recovered,

cast(jsonb_extract_path_text("_airbyte_data",'total_decease
d') as float) as total_deceased,

cast(jsonb_extract_path_text("_airbyte_data",'total_confirm
ed') as float) as total_confirmed,

cast(jsonb_extract_path_text("_airbyte_data",'total_recover
ed') as float) as total_recovered
   from "postgres".public._airbyte_raw_covid_epidemiology
```

```
);
```

Customize SQL Query

Feel free to:

- Rename the columns as you desire
    - avoiding using keywords such as `"key"` or `"date"`
- You can tweak the column data type if the ones generated by Airbyte are not the ones you favor
    - For example, let's use `Integer` instead of `Float` for the number of Covid cases...
- Add deduplicating logic
    - if you can identify which columns to use as Primary Keys (since airbyte isn't able to detect those automatically yet...)
    - (Note: actually I am not even sure if I can tell the proper primary key in this dataset...)
- Create a View (or materialized views) instead of a Table.
- etc

```
Unset

create view "postgres"."public"."covid_epidemiology" as (

   with parse_json_cte as (

       select

           _airbyte_emitted_at,



cast(jsonb_extract_path_text("_airbyte_data",'key') as
varchar) as id,
cast(jsonb_extract_path_text("_airbyte_data",'date') as
varchar) as updated_at,

cast(jsonb_extract_path_text("_airbyte_data",'new_tested')
as float) as new_tested,

cast(jsonb_extract_path_text("_airbyte_data",'new_deceased'
) as float) as new_deceased,

cast(jsonb_extract_path_text("_airbyte_data",'total_tested'
) as float) as total_tested,

cast(jsonb_extract_path_text("_airbyte_data",'new_confirmed
') as float) as new_confirmed,

cast(jsonb_extract_path_text("_airbyte_data",'new_recovered
') as float) as new_recovered,

cast(jsonb_extract_path_text("_airbyte_data",'total_decease
d') as float) as total_deceased,

cast(jsonb_extract_path_text("_airbyte_data",'total_confirm
ed') as float) as total_confirmed,

cast(jsonb_extract_path_text("_airbyte_data",'total_recover
ed') as float) as total_recovered
```

```
        from
"postgres".public._airbyte_raw_covid_epidemiology
    ),
    cte as (
        select
            *,
            row_number() over (
                partition by id
                order by updated_at desc
            ) as row_num
        from parse_json_cte
    )
    select
        substring(id, 1, 2) as id, -- Probably not the right
way to identify the primary key in this dataset...
        updated_at,
        _airbyte_emitted_at,

        case when new_tested = 'NaN' then 0 else
cast(new_tested as integer) end as new_tested,
        case when new_deceased = 'NaN' then 0 else
cast(new_deceased as integer) end as new_deceased,
       case when total_tested = 'NaN' then 0 else
cast(total_tested as integer) end as total_tested,

       case when new_confirmed = 'NaN' then 0 else
cast(new_confirmed as integer) end as new_confirmed,

       case when new_recovered = 'NaN' then 0 else
cast(new_recovered as integer) end as new_recovered,

       case when total_deceased = 'NaN' then 0 else
cast(total_deceased as integer) end as total_deceased,

       case when total_confirmed = 'NaN' then 0 else
cast(total_confirmed as integer) end as total_confirmed,

       case when total_recovered = 'NaN' then 0 else
cast(total_recovered as integer) end as total_recovered

   from cte

   where row_num = 1

);
```

Then you can run in your preferred SQL editor or tool!

If you are familiar with dbt or want to learn more about it, you can continue with the following tutorial using dbt...

Transformations with dbt (Part 2/3)

Overview

This tutorial will describe how to integrate SQL based transformations with Airbyte syncs using specialized transformation tool: dbt.

This tutorial is the second part of the previous tutorial Transformations with SQL. Next, we'll wrap-up with a third part on submitting transformations back in Airbyte: Transformations with Airbyte.

(Example outputs are updated with Airbyte version 0.23.0-alpha from May 2021)

Transformations with dbt

The tool in charge of transformation behind the scenes is actually called dbt (Data Build Tool).

Before generating the SQL files as we've seen in the previous tutorial, Airbyte sets up a dbt Docker instance and automatically generates a dbt project for us. This is created as specified in the dbt project documentation page with the right credentials for the target destination. The dbt models are then run afterward, thanks to the dbt CLI. However, for now, let's run through working with the dbt tool.

Validate dbt project settings

Let's say we identified our workspace (as shown in the previous tutorial Transformations with SQL), and we have a workspace ID of:

```
Unset



NORMALIZE_WORKSPACE="5/0/"
```

We can verify that the dbt project is properly configured for that workspace:

```
Unset



#!/usr/bin/env bash

docker run --rm -i -v airbyte_workspace:/data -w
/data/$NORMALIZE_WORKSPACE/normalize --network host
--entrypoint /usr/local/bin/dbt airbyte/normalization debug
--profiles-dir=. --project-dir=.
```

Example Output:

```
Unset



Running with dbt=0.19.1

dbt version: 0.19.1

python version: 3.8.8

python path: /usr/local/bin/python

os info: Linux-5.10.25-linuxkit-x86_64-with-glibc2.2.5

Using profiles.yml file at ./profiles.yml

Using dbt_project.yml file at
/data/5/0/normalize/dbt_project.yml



Configuration:

 profiles.yml file [OK found and valid]
 dbt_project.yml file [OK found and valid]


Required dependencies:
- git [OK found]


Connection:
 host: localhost
 port: 3000
 user: postgres
 database: postgres
 schema: quarantine
 search_path: None
 keepalives_idle: 0
 sslmode: None
 Connection test: OK connection ok
```

Compile and build dbt normalization models

If the previous command does not show any errors or discrepancies, it is now possible to invoke the CLI from within the docker image to trigger transformation processing:

```
Unset



#!/usr/bin/env bash

docker run --rm -i -v airbyte_workspace:/data -w
/data/$NORMALIZE_WORKSPACE/normalize --network host
--entrypoint /usr/local/bin/dbt airbyte/normalization run
--profiles-dir=. --project-dir=.
```

Example Output:

```
Unset



Running with dbt=0.19.1

Found 4 models, 0 tests, 0 snapshots, 0 analyses, 364
macros, 0 operations, 0 seed files, 1 source, 0 exposures



Concurrency: 32 threads (target='prod')



1 of 1 START table model
quarantine.covid_epidemiology.............................
......................... [RUN]

1 of 1 OK created table model
quarantine.covid_epidemiology.............................
.................... [SELECT 35822 in 0.47s]
```

```
Finished running 1 table model in 0.74s.

Completed successfully

Done. PASS=1 WARN=0 ERROR=0 SKIP=0 TOTAL=1
```

Exporting dbt normalization project outside Airbyte

As seen in the tutorial on exploring workspace folder, it is possible to browse the `normalize` folder and examine further logs if an error occurs.

In particular, we can also take a look at the dbt models generated by Airbyte and export them to the local host filesystem:

```
Unset

#!/usr/bin/env bash

TUTORIAL_DIR="$(pwd)/tutorial/"
rm -rf $TUTORIAL_DIR/normalization-files
mkdir -p $TUTORIAL_DIR/normalization-files
```

```
docker cp
airbyte-server:/tmp/workspace/$NORMALIZE_WORKSPACE/normaliz
e/ $TUTORIAL_DIR/normalization-files


NORMALIZE_DIR=$TUTORIAL_DIR/normalization-files/normalize

cd $NORMALIZE_DIR

cat $NORMALIZE_DIR/models/generated/**/*.sql
```

Example Output:

```
Unset


{{ config(alias="covid_epidemiology_ab1",
schema="_airbyte_quarantine",
tags=["top-level-intermediate"]) }}

-- SQL model to parse JSON blob stored in a single column
and extract into separated field columns as described by
the JSON Schema

select

   {{ json_extract_scalar('_airbyte_data', ['key']) }} as
{{ adapter.quote('key') }},

   {{ json_extract_scalar('_airbyte_data', ['date']) }} as
{{ adapter.quote('date') }},
   {{ json_extract_scalar('_airbyte_data', ['new_tested'])
}} as new_tested,

   {{ json_extract_scalar('_airbyte_data',
['new_deceased']) }} as new_deceased,

   {{ json_extract_scalar('_airbyte_data',
['total_tested']) }} as total_tested,

   {{ json_extract_scalar('_airbyte_data',
['new_confirmed']) }} as new_confirmed,

   {{ json_extract_scalar('_airbyte_data',
['new_recovered']) }} as new_recovered,

   {{ json_extract_scalar('_airbyte_data',
['total_deceased']) }} as total_deceased,

   {{ json_extract_scalar('_airbyte_data',
['total_confirmed']) }} as total_confirmed,

   {{ json_extract_scalar('_airbyte_data',
['total_recovered']) }} as total_recovered,

   _airbyte_emitted_at

from {{ source('quarantine',
'_airbyte_raw_covid_epidemiology') }}

-- covid_epidemiology
```

```
{{ config(alias="covid_epidemiology_ab2",
schema="_airbyte_quarantine",
tags=["top-level-intermediate"]) }}

-- SQL model to cast each column to its adequate SQL type
converted from the JSON schema type

select

   cast({{ adapter.quote('key') }} as {{
dbt_utils.type_string() }}) as {{ adapter.quote('key') }},

   cast({{ adapter.quote('date') }} as {{
dbt_utils.type_string() }}) as {{ adapter.quote('date') }},

   cast(new_tested as {{ dbt_utils.type_float() }}) as
new_tested,

   cast(new_deceased as {{ dbt_utils.type_float() }}) as
new_deceased,

   cast(total_tested as {{ dbt_utils.type_float() }}) as
total_tested,

   cast(new_confirmed as {{ dbt_utils.type_float() }}) as
new_confirmed,

   cast(new_recovered as {{ dbt_utils.type_float() }}) as
new_recovered,

   cast(total_deceased as {{ dbt_utils.type_float() }}) as
total_deceased,
   cast(total_confirmed as {{ dbt_utils.type_float() }}) as
total_confirmed,

   cast(total_recovered as {{ dbt_utils.type_float() }}) as
total_recovered,

   _airbyte_emitted_at

from {{ ref('covid_epidemiology_ab1_558') }}

-- covid_epidemiology


{{ config(alias="covid_epidemiology_ab3",
schema="_airbyte_quarantine",
tags=["top-level-intermediate"]) }}

-- SQL model to build a hash column based on the values of
this record

select

   *,

   {{ dbt_utils.surrogate_key([

       adapter.quote('key'),

       adapter.quote('date'),

       'new_tested',

       'new_deceased',

       'total_tested',
          'new_confirmed',

          'new_recovered',

          'total_deceased',

          'total_confirmed',

          'total_recovered',

    ]) }} as _airbyte_covid_epidemiology_hashid

from {{ ref('covid_epidemiology_ab2_558') }}

-- covid_epidemiology


{{ config(alias="covid_epidemiology", schema="quarantine",
tags=["top-level"]) }}

-- Final base SQL model

select

   {{ adapter.quote('key') }},

   {{ adapter.quote('date') }},

   new_tested,

   new_deceased,

   total_tested,

   new_confirmed,
   new_recovered,

   total_deceased,

   total_confirmed,

   total_recovered,

   _airbyte_emitted_at,

   _airbyte_covid_epidemiology_hashid

from {{ ref('covid_epidemiology_ab3_558') }}

-- covid_epidemiology from {{ source('quarantine',
'_airbyte_raw_covid_epidemiology') }}
```

If you have dbt installed locally on your machine, you can then view, edit, version, customize, and run the dbt models in your project outside Airbyte syncs.

```
Unset

#!/usr/bin/env bash

dbt deps --profiles-dir=$NORMALIZE_DIR
--project-dir=$NORMALIZE_DIR

dbt run --profiles-dir=$NORMALIZE_DIR
--project-dir=$NORMALIZE_DIR --full-refresh
```

Example Output:

Unset

```
Running with dbt=0.19.1

Installing
https://github.com/fishtown-analytics/dbt-utils.git@0.6.4

 Installed from revision 0.6.4



Running with dbt=0.19.1

Found 4 models, 0 tests, 0 snapshots, 0 analyses, 364
macros, 0 operations, 0 seed files, 1 source, 0 exposures



Concurrency: 32 threads (target='prod')



1 of 1 START table model
quarantine.covid_epidemiology..............................
......................... [RUN]

1 of 1 OK created table model
quarantine.covid_epidemiology..............................
.................... [SELECT 35822 in 0.44s]



Finished running 1 table model in 0.63s.
```

```
Completed successfully


Done. PASS=1 WARN=0 ERROR=0 SKIP=0 TOTAL=1
```

Now, that you've exported the generated normalization models, you can edit and tweak them as necessary.

If you want to know how to push your modifications back to Airbyte and use your updated dbt project during Airbyte syncs, you can continue with the following tutorial on importing transformations into Airbyte...

Transformations with Airbyte (Part 3/3)

Overview

This tutorial will describe how to push a custom dbt transformation project back to Airbyte to use during syncs.

This guide is the last part of the tutorial series on transformations, following Transformations with SQL and connecting EL with T using dbt.

(Example outputs are updated with Airbyte version 0.23.0-alpha from May 2021)

Transformations with Airbyte

After replication of data from a source connector (Extract) to a destination connector (Load), multiple optional transformation steps can now be applied as part of an Airbyte Sync. Possible workflows are:

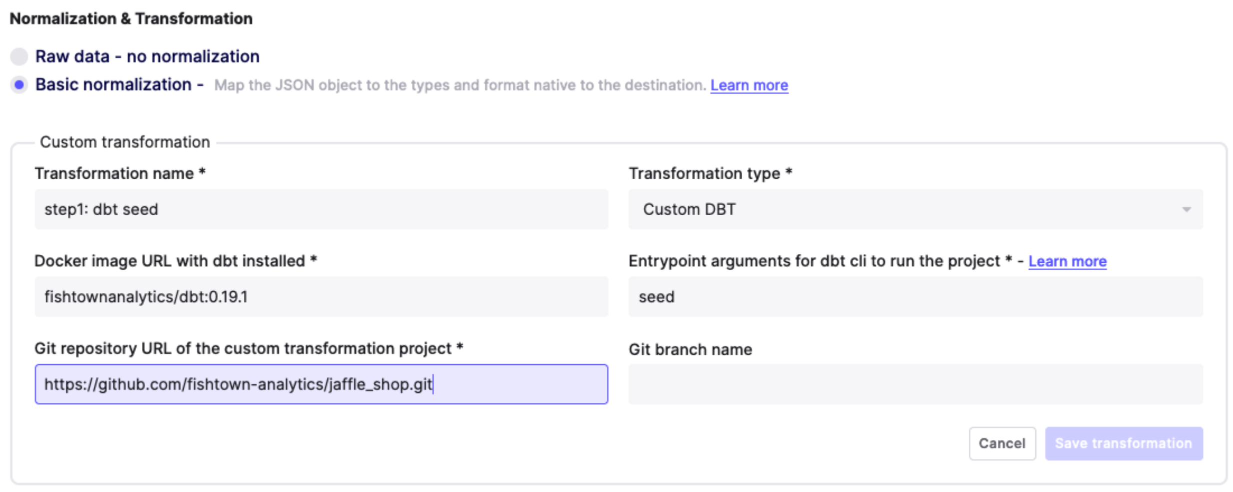
1. Basic normalization transformations as automatically generated by Airbyte dbt code generator.

2. Customized normalization transformations as edited by the user (the default generated normalization one should therefore be disabled)
3. Customized business transformations as specified by the user.

Public Git repository

In the connection settings page, I can add new Transformations steps to apply after normalization. For example, I want to run my custom dbt project jaffle_shop, whenever my sync is done replicating and normalizing my data.

You can find the jaffle shop test repository by clicking here.



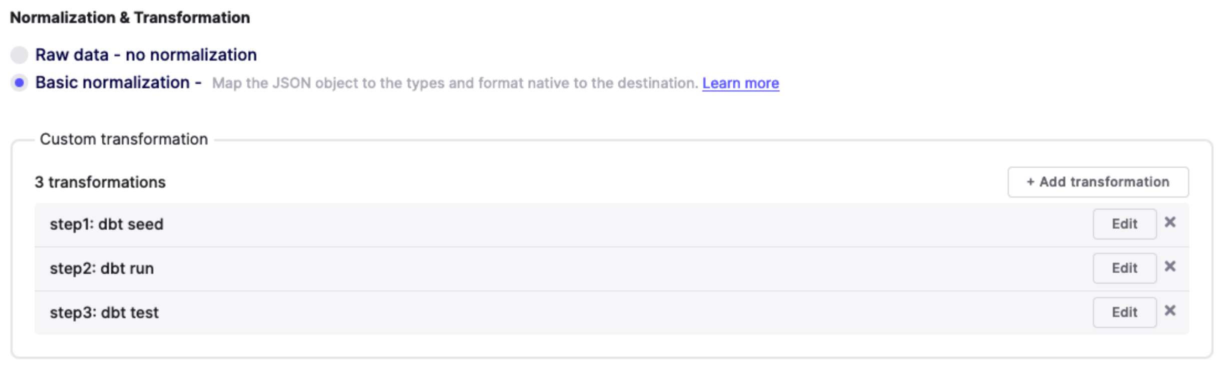
Normalization & Transformation

Raw data - no normalization

● Basic normalization - Map the JSON object to the types and format native to the destination. Learn more

Custom transformation

Transformation name *	Transformation type *
step1: dbt seed	Custom DBT
Docker image URL with dbt installed *	Entrypoint arguments for dbt cli to run the project * - Learn more
fishtownanalytics/dbt:0.19.1	seed
Git repository URL of the custom transformation project *	Git branch name
https://github.com/fishtown-analytics/jaffle_shop.git	



Normalization & Transformation

Raw data - no normalization

● Basic normalization - Map the JSON object to the types and format native to the destination. Learn more

Custom transformation

3 transformations

step1: dbt seed

step2: dbt run

step3: dbt test

Private Git repository

Now, let's connect my mono-repo Business Intelligence project stored in a private git repository to update the related tables and dashboards when my Airbyte syncs complete.

Note that if you need to connect to a private git repository, the recommended way to do so is to generate a `Personal Access Token` that can be used instead of a password. Then, you'll be able to include the credentials in the git repository url:

- GitHub - Personal Access Tokens
- Gitlab - Personal Access Tokens
- Azure DevOps - Personal Access Tokens

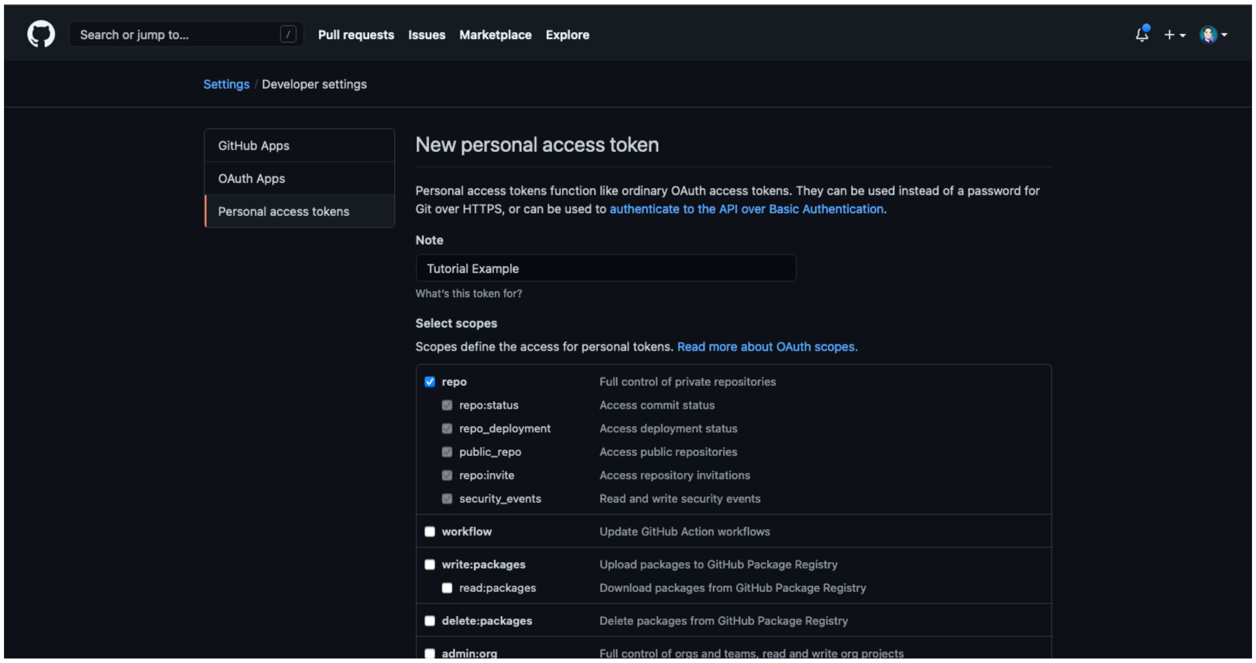
And then use it for cloning:

```
Unset



git clone https://username:token@github.com/user/repo
```

Where `https://username:token@github.com/user/repo` is the git repository url.

Example of a private git repo used as transformations

As an example, I go through my GitHub account to generate a Personal Access Token to use in Airbyte with permissions to clone my private repositories:


Search or jump to... Pull requests Issues Marketplace Explore

Settings / Developer settings

GitHub Apps
OAuth Apps
Personal access tokens

New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.

Note

Tutorial Example

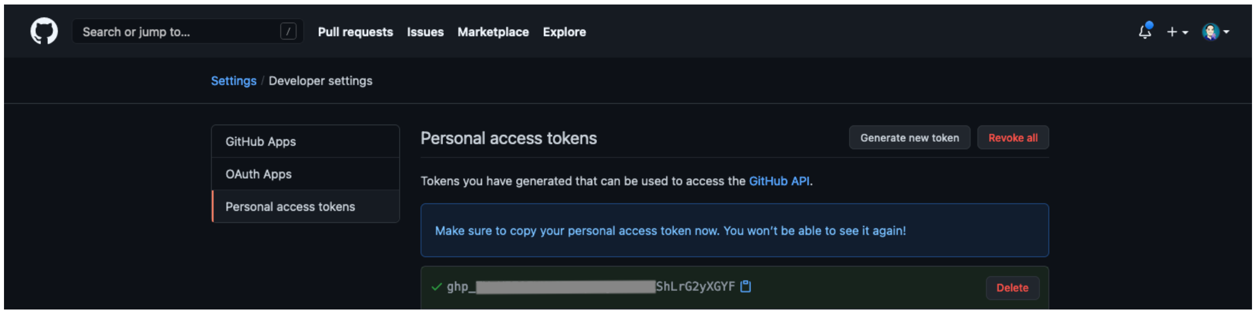
What's this token for?

Select scopes

Scopes define the access for personal tokens. Read more about OAuth scopes.

✓ repo	Full control of private repositories
✓ repo:status	Access commit status
✓ repo_deployment	Access deployment status
✓ public_repo	Access public repositories
✓ repo:invite	Access repository invitations
✓ security_events	Read and write security events
☐ workflow	Update GitHub Action workflows
☐ write:packages	Upload packages to GitHub Package Registry
☐ read:packages	Download packages from GitHub Package Registry
☐ delete:packages	Delete packages from GitHub Package Registry
☐ admin:org	Full control of orgs and teams, read and write org projects

This provides me with a token to use:


Search or jump to... Pull requests Issues Marketplace Explore

Settings / Developer settings

GitHub Apps
OAuth Apps
Personal access tokens

Personal access tokens

Generate new token | Revoke all

Tokens you have generated that can be used to access the GitHub API.

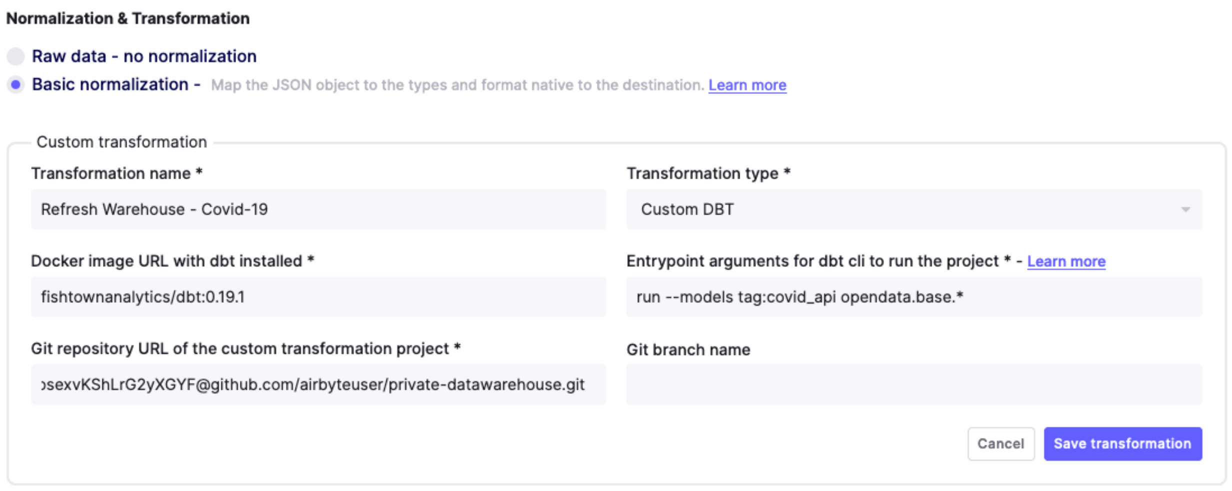
Make sure to copy your personal access token now. You won't be able to see it again!

✓ ghp_[redacted]ShLrG2yXGYF

Delete

In Airbyte, I can use the git url as:

```
https://airbyteuser:ghp_**********ShLrG2yXGYF@github.com/airbyt
euser/private-datawarehouse.git
```



Normalization & Transformation

Raw data - no normalization
[✓ Basic normalization] - Map the JSON object to the types and format native to the destination. Learn more

Custom transformation

Transformation name *	Refresh Warehouse - Covid-19
Transformation type *	Custom DBT
Docker image URL with dbt installed *	fishtownanalytics/dbt:0.19.1
Entrypoint arguments for dbt cli to run the project * - Learn more	run --models tag:covid_api opendata.base.*
Git repository URL of the custom transformation project *	›sexvKShLrG2yXGYF@github.com/airbyteuser/private-datawarehouse.git
Git branch name	

How-to use custom dbt tips

Allows "chained" dbt transformations

Since every transformation leave in his own Docker container, at this moment I can't rely on packages installed using `dbt deps` for the next transformations. According to the dbt documentation, I can configure the packages folder outside of the container:

```
Unset



# dbt_project.yml

packages-install-path: '../dbt_packages'
```

If I want to chain dbt deps and dbt run, I may use dbt build instead, which is not equivalent to the two previous commands, but will remove the need to alter the configuration of dbt.

Refresh models partially

Since I am using a mono-repo from my organization, other team members or departments may also contribute their dbt models to this centralized location. This will give us many dbt models and sources to build our complete data warehouse...

The whole warehouse is scheduled for full refresh on a different orchestration tool, or as part of the git repository CI. However, here, I want to partially refresh some small relevant tables when attaching this operation to a specific Airbyte sync, in this case, the Covid dataset.

Therefore, I can restrict the execution of models to a particular tag or folder by specifying in the dbt cli arguments, in this case whatever is related to "covid_api":

Unset

```
run --models tag:covid_api opendata.base.*
```

Now, when replications syncs are triggered by Airbyte, my custom transformations from my private git repository are also run at the end!

Using a custom run with variables

If you want to use a custom run and pass variables you need to use the follow syntax:

Unset

```
run --vars
'{"table_name":"sample","schema_name":"other_value"}'
```

This string must have no space. There is a Github issue to improve this. If you want to contribute to Airbyte, this is a good opportunity!

Basic Normalization

High-Level Overview

INFO

The high-level overview contains all the information you need to use Basic Normalization when pulling from APIs. Information past that can be read for advanced or educational purposes.

When you run your first Airbyte sync without the basic normalization, you'll notice that your data gets written to your destination as one data column with a JSON blob that contains all of your data. This is the `_airbyte_raw_` table that you may have seen before. Why do we create this table? A core tenet of ELT philosophy is that data should be untouched as it moves through the E and L stages so that the raw data is always accessible. If an unmodified version of the data exists in the destination, it can be retransformed without needing to sync data again.

If you have Basic Normalization enabled, Airbyte automatically uses this JSON blob to create a schema and tables with your data in mind, converting it to the format of your destination. This runs after your sync and may take a long time if you have a large amount of data synced. If you don't enable Basic Normalization, you'll have to transform the JSON data from that column yourself.

Example

Basic Normalization uses a fixed set of rules to map a json object from a source to the types and format that are native to the destination. For example if a source emits data that looks like this:

```
Unset

```

```
{
 "make": "alfa romeo",
 "model": "4C coupe",
 "horsepower": "247"
}
```

The destination connectors produce the following raw table in the destination database:

Unset

```
CREATE TABLE "_airbyte_raw_cars" (
   -- metadata added by airbyte
   "_airbyte_ab_id" VARCHAR, -- uuid value assigned by
connectors to each row of the data written in the
destination.
   "_airbyte_emitted_at" TIMESTAMP_WITH_TIMEZONE, -- time at
which the record was emitted.
   "_airbyte_data" JSONB -- data stored as a Json Blob.
);
```

Then, basic normalization would create the following table:

```
Unset


CREATE TABLE "cars" (
   "_airbyte_ab_id" VARCHAR,
   "_airbyte_emitted_at" TIMESTAMP_WITH_TIMEZONE,
   "_airbyte_cars_hashid" VARCHAR,
   "_airbyte_normalized_at" TIMESTAMP_WITH_TIMEZONE,

   -- data from source
   "make" VARCHAR,
   "model" VARCHAR,
   "horsepower" INTEGER
);
```

Normalization metadata columns

You'll notice that some metadata are added to keep track of important information about each record.

- Some are introduced at the destination connector level: These are propagated by the normalization process from the raw table to the final table
    - `_airbyte_ab_id`: uuid value assigned by connectors to each row of the data written in the destination.
    - `_airbyte_emitted_at`: time at which the record was emitted and recorded by destination connector.
- While other metadata columns are created at the normalization step.

- `_airbyte_<table_name>_hashid`: hash value assigned by airbyte normalization derived from a hash function of the record data.
- `_airbyte_normalized_at`: time at which the record was last normalized (useful to track when incremental transformations are performed)

Additional metadata columns can be added on some tables depending on the usage:

- On the Slowly Changing Dimension (SCD) tables:
    - `_airbyte_start_at`: equivalent to the cursor column defined on the table, denotes when the row was first seen
    - `_airbyte_end_at`: denotes until when the row was seen with these particular values. If this column is not NULL, then the record has been updated and is no longer the most up to date one. If NULL, then the row is the latest version for the record.
    - `_airbyte_active_row`: denotes if the row for the record is the latest version or not.
    - `_airbyte_unique_key_scd`: hash of primary keys + cursors used to de-duplicate the scd table.
    - On de-duplicated (and SCD) tables:
    - `_airbyte_unique_key`: hash of primary keys used to de-duplicate the final table.

The normalization rules are *not* configurable. They are designed to pick a reasonable set of defaults to hit the 80/20 rule of data normalization. We respect that normalization is a detail-oriented problem and that with a fixed set of rules, we cannot normalize your data in such a way that covers all use cases. If this feature does not meet your normalization needs, we always put the full json blob in destination as well, so that you can parse that object however best meets your use case. We will be adding more advanced normalization functionality shortly. Airbyte is focused on the EL of ELT. If you need a really featureful tool for the transformations then, we suggest trying out dbt.

Airbyte places the json blob version of your data in a table called `_airbyte_raw_<stream name>`. If basic normalization is turned on, it will place a separate copy of the data in a table called `<stream name>`. Under the hood, Airbyte is using dbt, which means that the data only ingresses into the data store one time. The normalization happens as a query within the datastore. This implementation avoids extra network time and costs.

Why does Airbyte have Basic Normalization?

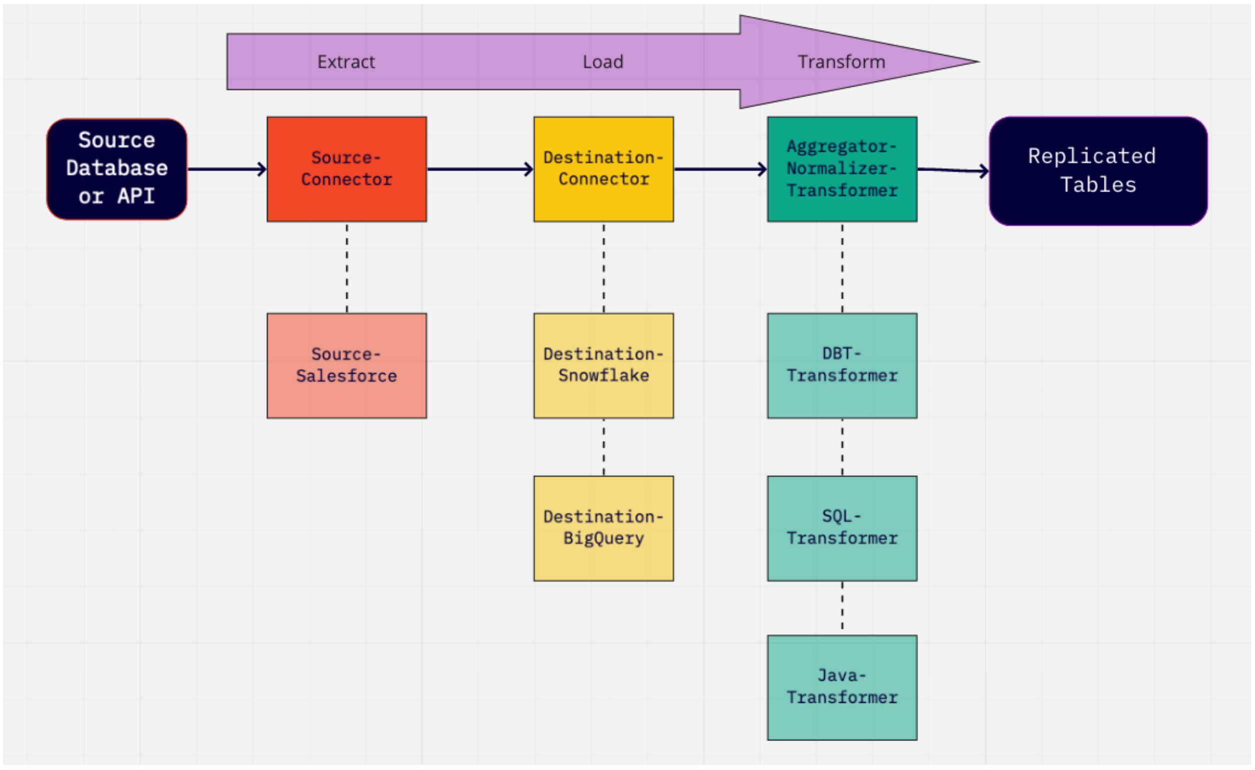
At its core, Airbyte is geared to handle the EL (Extract Load) steps of an ELT process. These steps can also be referred in Airbyte's dialect as "Source" and "Destination".

However, this is actually producing a table in the destination with a JSON blob column... For the typical analytics use case, you probably want this json blob normalized so that each field is its own column.

So, after EL, comes the T (transformation) and the first T step that Airbyte actually applies on top of the extracted data is called "Normalization".

Airbyte runs this step before handing the final data over to other tools that will manage further transformation down the line.

To summarize, we can represent the ELT process in the diagram below. These are steps that happens between your "Source Database or API" and the final "Replicated Tables" with examples of implementation underneath:



In Airbyte, the current normalization option is implemented using a dbt Transformer composed of:

- Airbyte base-normalization python package to generate dbt SQL models files
- dbt to compile and executes the models on top of the data in the destinations that supports it.

Destinations that Support Basic Normalization

- BigQuery
- MS Server SQL
- MySQL
  - The server must support the WITH keyword.
  - Require MySQL >= 8.0, or MariaDB >= 10.2.1.
- Postgres
- Redshift
- Snowflake

Basic Normalization can be configured when you're creating the connection between your Connection Setup and after in the Transformation Tab. Select the option: Normalized tabular data.

Rules

Typing

Airbyte tracks types using JsonSchema's primitive types. Here is how these types will map onto standard SQL types. Note: The names of the types may differ slightly across different destinations.

Airbyte uses the types described in the catalog to determine the correct type for each column. It does not try to use the values themselves to infer the type.

JsonSchema Type	Resulting Type	Notes
`number`	float	
`integer`	integer	

`string`	string	
`bit`	boolean	
`boolean`	boolean	
`string` with format label `date-time`	timestamp with timezone	
`array`	new table	see nesting
`object`	new table	see nesting

Nesting

Basic Normalization attempts to expand any nested arrays or objects it receives into separate tables in order to allow more ergonomic querying of your data.

Arrays

Basic Normalization expands arrays into separate tables. For example if the source provides the following data:

```
Unset


{

 "make": "alfa romeo",

 "model": "4C coupe",

 "limited_editions": [

   { "name": "4C spider", "release_year": 2013 },
   { "name" : "4C spider italia" , "release_year":  2018 }
 ]
}
```

The resulting normalized schema would be:

```
Unset


CREATE TABLE "cars" (
   "_airbyte_cars_hashid" VARCHAR,
   "_airbyte_emitted_at" TIMESTAMP_WITH_TIMEZONE,
   "_airbyte_normalized_at" TIMESTAMP_WITH_TIMEZONE,

   "make" VARCHAR,
   "model" VARCHAR
);

CREATE TABLE "limited_editions" (
   "_airbyte_limited_editions_hashid" VARCHAR,
   "_airbyte_cars_foreign_hashid" VARCHAR,
   "_airbyte_emitted_at" TIMESTAMP_WITH_TIMEZONE,

   "_airbyte_normalized_at" TIMESTAMP_WITH_TIMEZONE,


   "name" VARCHAR,

   "release_year" VARCHAR

);
```

If the nested items in the array are not objects then they are expanded into a string field of comma separated values e.g.:

```
Unset


{

  "make": "alfa romeo",

  "model": "4C coupe",

  "limited_editions": [ "4C spider", "4C spider italia"]

}
```

The resulting normalized schema would be:

```
Unset
```

```
CREATE TABLE "cars" (
   "_airbyte_cars_hashid" VARCHAR,
   "_airbyte_emitted_at" TIMESTAMP_WITH_TIMEZONE,
   "_airbyte_normalized_at" TIMESTAMP_WITH_TIMEZONE,

   "make" VARCHAR,
   "model" VARCHAR
);

CREATE TABLE "limited_editions" (
   "_airbyte_limited_editions_hashid" VARCHAR,
   "_airbyte_cars_foreign_hashid" VARCHAR,
   "_airbyte_emitted_at" TIMESTAMP_WITH_TIMEZONE,
   "_airbyte_normalized_at" TIMESTAMP_WITH_TIMEZONE,

   "data" VARCHAR
);
```

Objects

In the case of a nested object e.g.:

```
Unset


{

 "make": "alfa romeo",

 "model": "4C coupe",

 "powertrain_specs": { "horsepower": 247, "transmission":
"6-speed" }

}
```

The normalized schema would be:

```
Unset


CREATE TABLE "cars" (

   "_airbyte_cars_hashid" VARCHAR,

   "_airbyte_emitted_at" TIMESTAMP_WITH_TIMEZONE,

   "_airbyte_normalized_at" TIMESTAMP_WITH_TIMEZONE,


   "make" VARCHAR,

   "model" VARCHAR

);
```

```
CREATE TABLE "powertrain_specs" (
   "_airbyte_powertrain_hashid" VARCHAR,
   "_airbyte_cars_foreign_hashid" VARCHAR,
   "_airbyte_emitted_at" TIMESTAMP_WITH_TIMEZONE,
   "_airbyte_normalized_at" TIMESTAMP_WITH_TIMEZONE,

   "horsepower" INTEGER,
   "transmission" VARCHAR
);
```

Naming Collisions for un-nested objects

When extracting nested objects or arrays, the Basic Normalization process needs to figure out new names for the expanded tables.

For example, if we had a `cars` table with a nested column `cars` containing an object whose schema is identical to the parent table.

```
Unset

{
 "make": "alfa romeo",
 "model": "4C coupe",
 "cars": [

   { "make": "audi", "model": "A7" },

   { "make" : "lotus" , "model":  "elise" }

   { "make" : "chevrolet" , "model":  "mustang" }

 ]

}
```

The expanded table would have a conflict in terms of naming since both are named `cars`. To avoid name collisions and ensure a more consistent naming scheme, Basic Normalization chooses the expanded name as follows:

- `cars` for the original parent table
- `cars_da3_cars` for the expanded nested columns following this naming scheme in 3 parts: `<Json path>_<Hash>_<nested column name>`
- Json path: The entire json path string with '_' characters used as delimiters to reach the table that contains the nested column name.
- Hash: Hash of the entire json path to reach the nested column reduced to 3 characters. This is to make sure we have a unique name (in case part of the name gets truncated, see below)
- Nested column name: name of the column being expanded into its own table.

By following this strategy, nested columns should "never" collide with other table names. If it does, an exception will probably be thrown either by the normalization process or by dbt that runs afterward.

```
Unset



CREATE TABLE "cars" (
    "_airbyte_cars_hashid" VARCHAR,
    "_airbyte_emitted_at" TIMESTAMP_WITH_TIMEZONE,
    "_airbyte_normalized_at" TIMESTAMP_WITH_TIMEZONE,

    "make" VARCHAR,
    "model" VARCHAR
);

CREATE TABLE "cars_da3_cars" (
    "_airbyte_cars_hashid" VARCHAR,
    "_airbyte_cars_foreign_hashid" VARCHAR,
    "_airbyte_emitted_at" TIMESTAMP_WITH_TIMEZONE,
    "_airbyte_normalized_at" TIMESTAMP_WITH_TIMEZONE,

    "make" VARCHAR,
    "model" VARCHAR
);
```

Naming limitations & truncation

Note that different destinations have various naming limitations, most commonly on how long names can be. For instance, the Postgres documentation states:
The system uses no more than NAMEDATALEN-1 bytes of an identifier; longer names can be written in commands, but they will be truncated. By default, NAMEDATALEN is 64 so the maximum identifier length is 63 bytes

Most modern data warehouses have name lengths limits on the longer side, so this should not affect us that often. Basic Normalization will fallback to the following rules:

1. No Truncate if under destination's character limits

However, in the rare cases where these limits are reached:

1. Truncate only the `Json path` to fit into destination's character limits
2. Truncate the `Json path` to at least the 10 first characters, then truncate the nested column name starting in the middle to preserve prefix/suffix substrings intact (whenever a truncate in the middle is made, two '__' characters are also inserted to denote where it happened) to fit into destination's character limits

As an example from the hubspot source, we could have the following tables with nested columns:

Description	Example 1	Example 2
Original Stream Name	companies	deals
Json path to the nested column	`companies/property_engagements_last_meeting_booked_campaign`	`deals/properties/engagements_last_meeting_booked_medium`
Final table name of expanded nested column on BigQuery	companies_2e8_property_engagements_last_meeting_booked_campaign	deals_properties_6e6_engagements_last_meeting_booked_medium

Final table name of expanded nested column on Postgres	companies_2e8_property_engag__oked_campaign	deals_prop_6e6_engagements_l__booked_medium

As mentioned in the overview:

- Airbyte places the json blob version of your data in a table called `_airbyte_raw_<stream name>`.
- If basic normalization is turned on, it will place a separate copy of the data in a table called `<stream name>`.
- In certain pathological cases, basic normalization is required to generate large models with many columns and multiple intermediate transformation steps for a stream. This may break down the "ephemeral" materialization strategy and require the use of additional intermediate views or tables instead. As a result, you may notice additional temporary tables being generated in the destination to handle these checkpoints.

UI Configurations

To enable basic normalization (which is optional), you can toggle it on or disable it in the "Normalization and Transformation" section when setting up your connection:

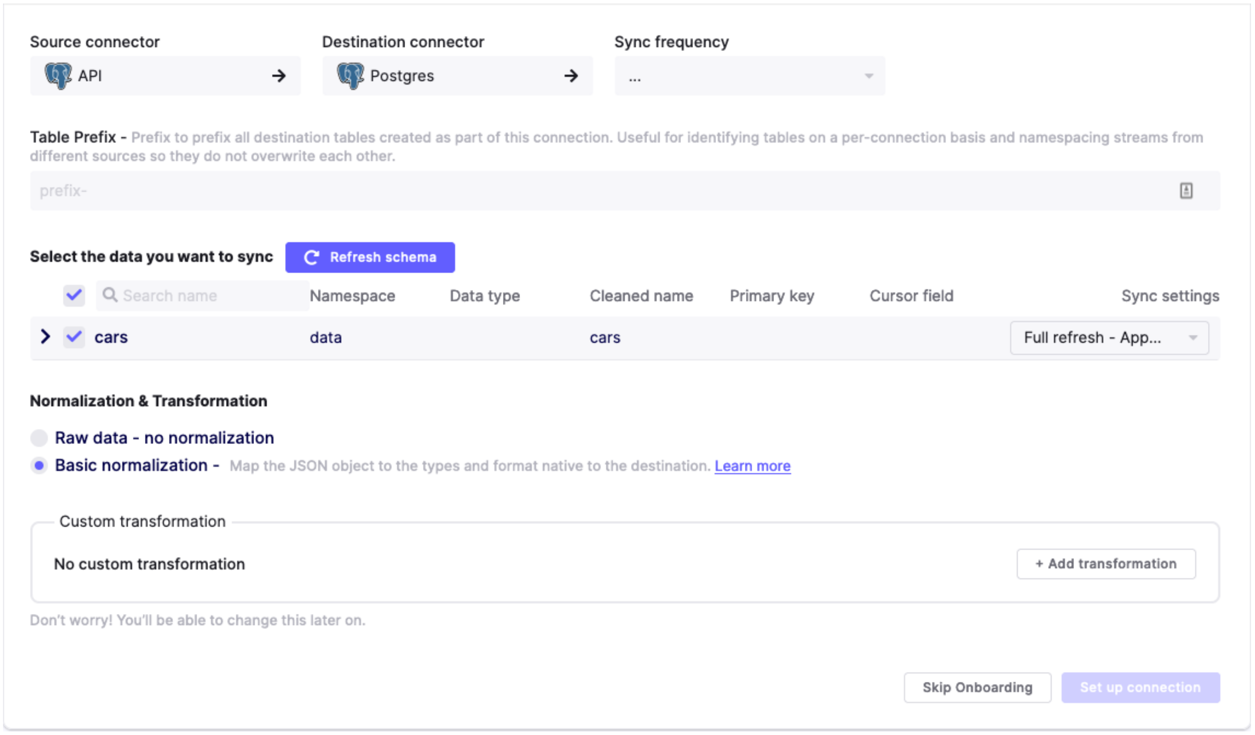

Source connector
API
Destination connector
Postgres
Sync frequency
...

Table Prefix - Prefix to prefix all destination tables created as part of this connection. Useful for identifying tables on a per-connection basis and namespacing streams from different sources so they do not overwrite each other.

prefix-

Select the data you want to sync

	Search name	Namespace	Data type	Cleaned name	Primary key	Cursor field	Sync settings
✓	cars	data		cars			Full refresh - App...

Normalization & Transformation

Raw data - no normalization [✓ Basic normalization - Map the JSON object to the types and format native to the destination. Learn more]

Custom transformation

No custom transformation

Don't worry! You'll be able to change this later on.

Incremental runs

When the source is configured with sync modes compatible with incremental transformations (using append on destination) such as (full_refresh_append, incremental append or incremental deduped history), only rows that have changed in the source are transferred over the network and written by the destination connector. Normalization will then try to build the normalized tables incrementally as the rows in the raw tables that have been created or updated since the last time dbt ran. As such, on each dbt run, the models get built incrementally. This limits the amount of data that needs to be transformed, vastly reducing the runtime of the transformations. This improves warehouse performance and reduces compute costs. Because normalization can be either run incrementally and, or, in full refresh, a technical column `_airbyte_normalized_at` can serve to track when was the last time a record has been transformed and written by normalization. This may greatly diverge from the `_airbyte_emitted_at` value as the normalized tables could be totally re-built at a latter time from the data stored in the `_airbyte_raw` tables.

Partitioning, clustering, sorting, indexing

Normalization produces tables that are partitioned, clustered, sorted or indexed depending on the destination engine and on the type of tables being built. The goal of these are to make read more performant, especially when running incremental updates.

In general, normalization needs to do lookup on the last emitted_at column to know if a record is freshly produced and need to be incrementally processed or not. But in certain models, such as SCD tables for example, we also need to retrieve older data to update their type 2 SCD end_date and active_row flags, thus a different partitioning scheme is used to optimize that use case.

On Postgres destination, an additional table suffixed with `_stg` for every stream replicated in incremental deduped history needs to be persisted (in a different staging schema) for incremental transformations to work because of a limitation.

Extending Basic Normalization

Note that all the choices made by Normalization as described in this documentation page in terms of naming (and more) could be overridden by your own custom choices. To do so, you can follow the following tutorials:

- to build a custom SQL view with your own naming conventions
- to export, edit and run custom dbt normalization yourself
- or further, you can configure the use of a custom dbt project within Airbyte by following this guide.

Operations

Airbyte connections support configuring additional transformations that execute after the sync. Useful applications could be:

- Customized normalization to better fit the requirements of your own business context.
- Business transformations from a technical data representation into a more logical and business oriented data structure. This can facilitate usage by end-users,

non-technical operators, and executives looking to generate Business Intelligence dashboards and reports.
- Data Quality, performance optimization, alerting and monitoring, etc.
- Integration with other tools from your data stack (orchestration, data visualization, etc.)

Supported Operations

dbt transformations

- git repository url:

A url to a git repository to (shallow) clone the latest dbt project code from.

The project versioned in the repository is expected to:

- be a valid dbt package with a `dbt_project.yml` file at its root.
- have a `dbt_project.yml` with a "profile" name declared as described here.

When using the dbt CLI, dbt checks your `profiles.yml` file for a profile with the same name. A profile contains all the details required to connect to your data warehouse. This file generally lives outside of your dbt project to avoid sensitive credentials being checked in to version control. Therefore, a `profiles.yml` will be generated according to the configured destination from the Airbyte UI.

Note that if you prefer to use your own `profiles.yml` stored in the git repository or in the Docker image, then you can specify an override with `--profiles-dir=<path-to-my-profiles-yml>` in the dbt CLI arguments.

- git repository branch (optional):

The name of the branch to use when cloning the git repository. If left empty, git will use the default branch of your repository.

- docker image:

A Docker image and tag to run dbt commands from. The Docker image should have `/bin/bash` and `dbt` installed for this operation type to work.

A typical value for this field would be for example: `fishtownanalytics/dbt:1.0.0` from dbt dockerhub.

This field lets you configure the version of dbt that your custom dbt project requires and the loading of additional software and packages necessary for your transformations (other than your dbt `packages.yml` file).

- dbt cli arguments

This operation type is aimed at running the dbt cli.

A typical value for this field would be "run" and the actual command invoked would as a result be: `dbt  run` in the docker container.

One thing to consider is that dbt allows for vast configuration of the run command, for example, allowing you to select a subset of models. You can find the dbt reference docs which describes this set of available commands and options.

Future Operations

- Docker/Script operations: Execute a generic script in a custom Docker container.
- Webhook operations: Trigger API or hooks from other providers.
- Airflow operations: To use a specialized orchestration tool that lets you schedule and manage more advanced/complex sequences of operations in your sync workflow.

Going Further

In the meantime, please feel free to react, comment, and share your thoughts/use cases with us. We would be glad to hear your feedback and ideas as they will help shape the next set of features and our roadmap for the future. You can head to our GitHub and participate in the corresponding issue or discussions. Thank you!

Browsing Output Logs

Overview

This tutorial will describe how to explore Airbyte Workspace folders.

This is useful if you need to browse the docker volumes where extra output files of Airbyte server and workers are stored since they may not be accessible through the UI.

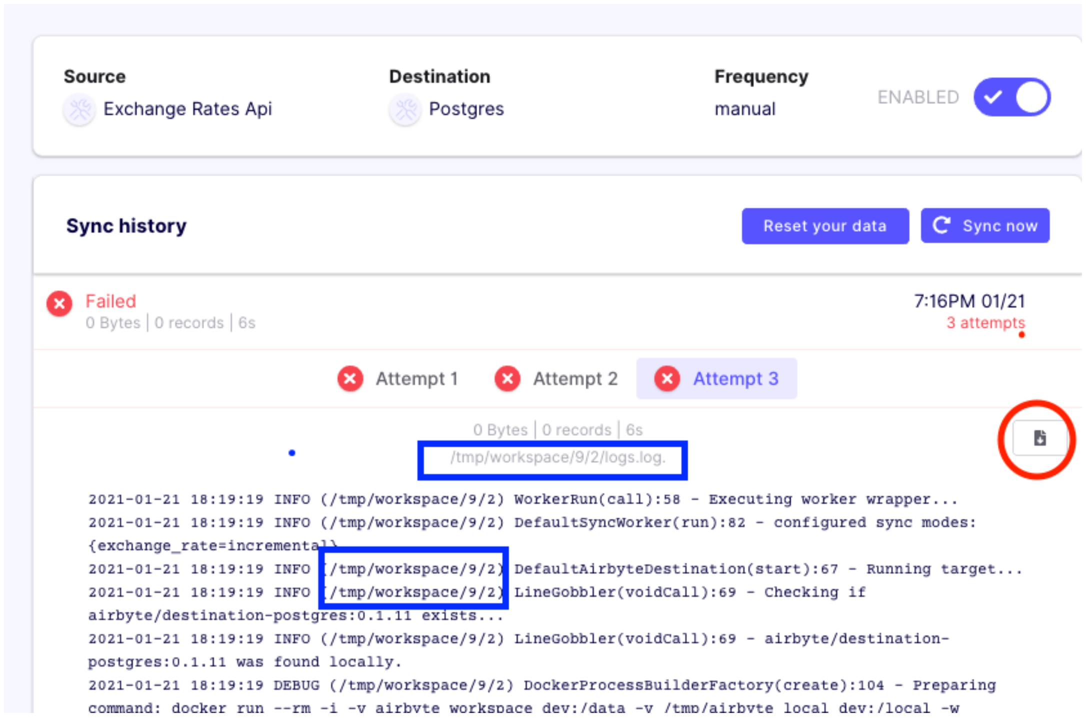
Exploring the Logs folders

When running a Sync in Airbyte, you have the option to look at the logs in the UI as shown next.

Identifying Workspace IDs

In the screenshot below, you can notice the highlighted blue boxes are showing the id numbers that were used for the selected "Attempt" for this sync job.

In this case, the job was running in `/tmp/workspace/9/2/` folder since the tab of the third attempt is being selected in the UI (first attempt would be `/tmp/workspace/9/0/`).



```
2021-01-21 18:19:19 INFO (/tmp/workspace/9/2) WorkerRun(call):58 - Executing worker wrapper...
2021-01-21 18:19:19 INFO (/tmp/workspace/9/2) DefaultSyncWorker(run):82 - configured sync modes:
{exchange_rate=incremental}
2021-01-21 18:19:19 INFO (/tmp/workspace/9/2) DefaultAirbyteDestination(start):67 - Running target...
2021-01-21 18:19:19 INFO (/tmp/workspace/9/2) LineGobbler(voidCall):69 - Checking if
airbyte/destination-postgres:0.1.11 exists...
2021-01-21 18:19:19 INFO (/tmp/workspace/9/2) LineGobbler(voidCall):69 - airbyte/destination-
postgres:0.1.11 was found locally.
2021-01-21 18:19:19 DEBUG (/tmp/workspace/9/2) DockerProcessBuilderFactory(create):104 - Preparing
command: docker run --rm -i -v airbyte_workspace_dev:/data -v /tmp/airbyte_local_dev:/local -w
```

The highlighted button in the red circle on the right would allow you to download the logs.log file.
However, there are actually more files being recorded in the same workspace folder... Thus, we might want to dive deeper to explore these folders and gain a better understanding of what is being run by Airbyte.

Understanding the Docker run commands

Scrolling down a bit more, we can also read the different docker commands being used internally are starting with:

```
Unset

docker run --rm -i -v airbyte_workspace:/data -v
/tmp/airbyte_local:/local -w /data/9/2 --network host ...
```

From there, we can observe that Airbyte is calling the `-v` option to use a docker named volume called `airbyte_workspace` that is mounted in the container at the location `/data`.

Following Docker Volume documentation, we can inspect and manipulate persisted configuration data in these volumes.

Opening a Unix shell prompt to browse the Docker volume

For example, we can run any docker container/image to browse the content of this named volume by mounting it similarly, let's use the busybox image.

```
Unset

docker run -it --rm --volume airbyte_workspace:/data
busybox
```

This will drop you into an `sh` shell inside the docker container to allow you to do what you want inside a BusyBox system from which we can browse the filesystem and accessing to log files:

```
Unset

ls /data/9/2/
```

Example Output:

```
Unset

catalog.json                          normalize
tap_config.json
logs.log                              singer_rendered_catalog.json
target_config.json
```

Browsing from the host shell

Or, if you don't want to transfer to a shell prompt inside the docker image, you can simply run Shell commands using docker commands as a proxy like this:

```
Unset

docker run -it --rm --volume airbyte_workspace:/data
busybox ls /data/9/2
```

Example Output:

```
Unset

catalog.json                         singer_rendered_catalog.json
logs.log                             tap_config.json
normalize                            target_config.json
```

Reading the content of the catalog.json file

For example, it is often useful to inspect the content of the catalog file. You could do so by running a `cat` command:

```
Unset

docker run -it --rm --volume airbyte_workspace:/data
busybox cat /data/9/2/catalog.json
```

Example Output:

```
Unset

{"streams":[{"stream":{"name":"exchange_rate","json_schema"
:{"type":"object","properties":{"CHF":{"type":"number"},"HR
K":{"type":"number"},"date":{"type":"string"},"MXN":{"type"
:"number"},"ZAR":{"type":"number"},"INR":{"type":"number"},
"CNY":{"type":"number"},"THB":{"type":"number"},"AUD":{"typ
e":"number"},"ILS":{"type":"number"},"KRW":{"type":"number"
},"JPY":{"type":"number"},"PLN":{"type":"number"},"GBP":{"t
ype":"number"},"IDR":{"type":"number"},"HUF":{"type":"numbe
r"},"PHP":{"type":"number"},"TRY":{"type":"number"},"RUB":{
"type":"number"},"HKD":{"type":"number"},"ISK":{"type":"num
ber"},"EUR":{"type":"number"},"DKK":{"type":"number"},"CAD"
:{"type":"number"},"MYR":{"type":"number"},"USD":{"type":"n
umber"},"BGN":{"type":"number"},"NOK":{"type":"number"},"RO
N":{"type":"number"},"SGD":{"type":"number"},"CZK":{"type":
"number"},"SEK":{"type":"number"},"NZD":{"type":"number"},"
BRL":{"type":"number"}}},"supported_sync_modes":["full_refr
esh"],"default_cursor_field":[]},"sync_mode":"full_refresh"
,"cursor_field":[]}]}
```

Extract catalog.json file from docker volume

Or if you want to copy it out from the docker image onto your host machine:

```
Unset


docker cp airbyte-server:/tmp/workspace/9/2/catalog.json .

cat catalog.json
```

Browsing on Kubernetes

If you are running on Kubernetes, use the following commands instead to browsing and copy the files to your local.

To browse, identify the pod you are interested in and exec into it. You will be presented with a terminal that will accept normal linux commands e.g ls.

```
Unset


kubectl exec -it <pod name> -n <namespace pod is in> -c
main bash

e.g.

kubectl exec -it destination-bigquery-worker-3607-0-chlle
-n jobs  -c main bash

root@destination-bigquery-worker-3607-0-chlle:/config# ls

FINISHED_UPLOADING  destination_catalog.json
destination_config.json
```

To copy the file on to your local in order to preserve it's contents:

```
Unset
```

```
kubectl cp <namespace pods are
in>/<normalisation-pod-name>:/config/destination_catalog.js
on ./catalog.json

e.g.

kubectl cp
jobs/normalization-worker-3605-0-sxtox:/config/destination_
catalog.json ./catalog.json

cat ./catalog.json
```

CSV or JSON local Destinations: Check local data folder

If you setup a pipeline using one of the local File based destinations (CSV or JSON), Airbyte is writing the resulting files containing the data in the special `/local/` directory in the container. By default, this volume is mounted from `/tmp/airbyte_local` on the host machine. So you need to navigate to this local folder on the filesystem of the machine running the Airbyte deployment to retrieve the local data files.
CAUTION

Please make sure that Docker Desktop has access to `/tmp` (and `/private` on a MacOS, as /tmp has a symlink that points to /private. It will not work otherwise). You allow it with "File sharing" in `Settings -> Resources -> File sharing -> add the one or two above folder` and hit the "Apply & restart" button.

Or, you can also run through docker commands as proxy:

```
Unset

#!/usr/bin/env bash
```

```
echo "In the container:"

docker run -it --rm -v /tmp/airbyte_local:/local busybox
find /local

echo ""
echo "On the host:"

find /tmp/airbyte_local
```

Example Output:

```
Unset

In the container:
/local
/local/data
/local/data/exchange_rate_raw.csv

On the host:
/tmp/airbyte_local
/tmp/airbyte_local/data
/tmp/airbyte_local/data/exchange_rate_raw.csv
```

Notes about running on macOS vs Linux

Note that Docker for Mac is not a real Docker host, now it actually runs a virtual machine behind the scenes and hides it from you to make things "simpler".

Here are some related links as references on accessing Docker Volumes:

- on macOS Using Docker containers in 2019
- official doc Use Volume

From these discussions, we've been using on macOS either:

1. any docker container/image to browse the virtual filesystem by mounting the volume in order to access them, for example with busybox
2. or extract files from the volume by copying them onto the host with Docker cp

However, as a side remark on Linux, accessing to named Docker Volume can be easier since you simply need to:

```
Unset


docker volume inspect <volume_name>
```

Then look at the `Mountpoint` value, this is where the volume is actually stored in the host filesystem and you can directly retrieve files directly from that folder.