**Incremental models**

**Overview**[**​**](https://docs.getdbt.com/docs/build/incremental-models#overview)

Incremental models are built as tables in your [data warehouse](https://docs.getdbt.com/terms/data-warehouse). The first time a model is run, the [table](https://docs.getdbt.com/terms/table) is built by transforming *all* rows of source data. On subsequent runs, dbt transforms *only* the rows in your source data that you tell dbt to filter for, inserting them into the target table which is the table that has already been built.

Often, the rows you filter for on an incremental run will be the rows in your source data that have been created or updated since the last time dbt ran. As such, on each dbt run, your model gets built incrementally.

Using an incremental model **limits the amount of data that needs to be transformed, vastly reducing the runtime of your transformations. This improves warehouse performance and reduces compute costs.**

Defining Incremental materialization in config:

{{

    config(

        materialized='incremental',

        unique\_key= 'id'

    )

}}

Incremental model need two things:

To use incremental models, you also need to tell dbt:

* How to filter the rows on an incremental run
* The unique key of the model (if any)

Filtering rows on an incremental run

To tell dbt which rows it should transform on an incremental run, wrap valid SQL that filters for these rows in the is\_incremental() macro.

Using timestamp to find the updated or new data:

Often, you'll want to filter for "new" rows, as in, rows that have been created since the last time dbt ran this model. The best way to find the timestamp of the most recent run of this model is by checking the most recent timestamp in your target table. dbt makes it easy to query your target table by using the "[{{ this }}](https://docs.getdbt.com/reference/dbt-jinja-functions/this)" variable.

Using Unique key to avoid duplication in the target table:

Also common is wanting to capture both new and updated records. For updated records, you'll need to [define a unique key](https://docs.getdbt.com/docs/build/incremental-models#defining-a-unique-key-optional) to ensure you don't bring in modified records as duplicates. Your is\_incremental() code will check for rows created *or modified* since the last time dbt ran this mode

Example:

{{

    config(

        materialized='incremental',

        unique\_key= 'id'

    )

}}

**with incremental\_sales as (**

**select \* from demo.practice.sales**

**{% if is\_incremental() %}**

**where current\_time >= (select max(current\_time) from {{ this }})**

**{% endif %}**

**)**

select \* from incremental\_sales

Logic

we have given a condition that add those columns which have the time greater than the time of previous entries. Like new data would have time greater than the old data so automatically it become new data if the time Is greater and only that data will be added to the target table whose time Is greater than the most recent of data in those entries.

OPTIMIZING YOUR INCREMENTAL MODEL

*For more complex incremental models that make use of Common Table Expressions (CTEs), you should consider the impact of the position of the is\_incremental() macro on query performance.* ***In some warehouses, filtering your records early can vastly improve the run time of your query!***

Defining a unique key (optional)[​](https://docs.getdbt.com/docs/build/incremental-models#defining-a-unique-key-optional)

A unique\_key enables updating existing rows instead of just appending new rows. If new information arrives for an existing unique\_key, that new information can replace the current information instead of being appended to the table. If a duplicate row arrives, it can be ignored. Refer to [strategy specific configs](https://docs.getdbt.com/docs/build/incremental-models#strategy-specific-configs) for more options on managing this update behavior, like choosing only specific columns to update.

Not specifying a unique\_key will result in append-only behavior, which means dbt inserts all rows returned by the model's SQL into the preexisting target table without regard for whether the rows represent duplicates.

you can define unique\_key in a configuration block at the top of your model, and it can be a single column name or a list of column names.   In cases where you need multiple columns in combination to uniquely identify each row, we recommend you pass these columns as a **list (unique\_key = ['user\_id', 'session\_number']),** rather than a string **expression (unique\_key = 'concat(user\_id, session\_number)').**

for example:

**['col1', 'col2', …])**

Columns used in this way should not contain any nulls, or the incremental model run may fail.

Either ensure that each column has no nulls (for example with coalesce(COLUMN\_NAME, 'VALUE\_IF\_NULL')), or define a single-column [surrogate key](https://docs.getdbt.com/terms/surrogate-key) (for example with [dbt\_utils.generate\_surrogate\_key](https://github.com/dbt-labs/dbt-utils" \l "generate_surrogate_key-source" \t "_blank)).

When you pass a list of columns in unique key, please ensure that each column does not contain any nulls, or the incremental model run may fail.

How do I rebuild an incremental model?

If your incremental model logic has changed, the transformations on your new rows of data may diverge from the historical transformations, which are stored in your target table. In this case, you should rebuild your incremental model.

To force dbt to rebuild the entire incremental model from scratch, use the --full-refresh flag on the command line. This flag will cause dbt to drop the existing target table in the database before rebuilding it for all-time.

**$ dbt run --full-refresh --select my\_incremental\_model+**

*It's also advisable to rebuild any downstream models, as indicated by the trailing +.*

Understanding incremental models

When should I use an incremental model?[​](https://docs.getdbt.com/docs/build/incremental-models#when-should-i-use-an-incremental-model)

It's often desirable to build models as tables in your data warehouse since downstream queries are more performant. While the table materialization also creates your models as tables, it rebuilds the table on each dbt run. These runs can become problematic in that they use a lot of compute when either:

* **source data tables have millions, or even billions, of rows.**
* **the transformations on the source data are computationally expensive (that is, take a long time to execute), for example, complex Regex functions, or UDFs are being used to transform data.**

Like many things in programming, ***incremental models are a trade-off between complexity and performance. While they are not as straightforward as the view and table materializations, they can lead to significantly better performance of your dbt runs***.

Understanding the is\_incremental() macro[​](https://docs.getdbt.com/docs/build/incremental-models#understanding-the-is_incremental-macro)

The is\_incremental() macro will return True if *all* of the following conditions are met:

* the destination table already exists in the database

(There should be a table in database because the incremental model focuses more on updating or appending the data rather than on creating tables. You should create table in any way before this model or with in some process in it)

* dbt is *not* running in full-refresh mode

*Full-refresh mode is a mode in dbt where all models are rebuilt, regardless of their incremental settings.*

* the running model is configured with materialized='incremental'

Note that the SQL in your model needs to be valid whether is\_incremental() evaluates to True or False.

Refresh incremental models[​](https://docs.getdbt.com/reference/commands/run#refresh-incremental-models)

If you provide the --full-refresh flag to dbt run, dbt will treat incremental models as [table](https://docs.getdbt.com/terms/table) models. This is useful when

1. The schema of an incremental model changes and you need to recreate it.
2. You want to reprocess the entirety of the incremental model because of new logic in the model code.

dbt run --full-refresh

You can also supply the flag by its short name: ***dbt run -f.***

In the dbt compilation context, this flag will be available as [flags.FULL\_REFRESH](https://docs.getdbt.com/reference/dbt-jinja-functions/flags). Further, the is\_incremental() macro will return false for *all* models in response when the --full-refresh flag is specified.

What if the columns of my incremental model change?

* Incremental models can include an optional **on\_schema\_change** parameter to handle schema changes. This parameter allows dbt to continue running incremental models even when there are changes in the columns of the target table. These options enable dbt to continue running incremental models in the presence of schema changes, resulting in fewer --full-refresh scenarios and saving query costs.
* The **on\_schema\_change** parameter can be configured in the **dbt\_project.yml** file at the project level or in individual model files.

**dbt\_project.yml**

models:

*+on\_schema\_change: "sync\_all\_columns"*

**models/staging/fct\_daily\_active\_users.sql**

{{

config(

materialized='incremental',

unique\_key='date\_day',

*on\_schema\_change='fail'*

)

}}

* The possible values for **on\_schema\_change** are:
  + **ignore** (default): Ignores schema changes and continues running the incremental model without any adjustments.
  + **fail**: Triggers an error message when there are schema differences between the source and target schemas. This helps to catch and address any inconsistencies.
  + **append\_new\_columns**: Appends new columns to the existing table without removing any columns that are not present in the new data. This allows for the addition of new columns while retaining existing data.
  + **sync\_all\_columns**: Adds new columns to the existing table and removes any columns that are no longer present. This behavior includes data type changes as well. However, note that changing column types on BigQuery requires a full table scan, so consider the performance implications when using this option.

Default behavior[​](https://docs.getdbt.com/docs/build/incremental-models#default-behavior)

This is the behavior if **on\_schema\_change:** ignore, which is set by default, and on older versions of dbt.

If you add a column to your incremental model, and execute a dbt run, this column will *not* appear in your target table.

Similarly, if you remove a column from your incremental model, and execute a dbt run, this column will *not* be removed from your target table.

Instead, whenever the logic of your incremental changes, execute a full-refresh run of both your incremental model and any downstream models.

Incremental Strategies:

On some adapters, an optional incremental\_strategy config controls the code that dbt uses to build incremental models. Different approaches may vary by effectiveness depending on the volume of data, the reliability of your unique\_key, or the availability of certain features.

Configuring incremental strategy[​](https://docs.getdbt.com/docs/build/incremental-models#configuring-incremental-strategy)

The incremental\_strategy config can either be specified in specific models, or for all models in your dbt\_project.yml file:

* **dbt\_project.yml**

models:  
 **+incremental\_strategy: "insert\_overwrite**

or:

* **models/my\_model.sql**

{{  
 config(  
 materialized='incremental',  
 unique\_key='date\_day',  
 **incremental\_strategy='delete+insert',**  
 ...  
 )  
}}  
  
select ...

We can define the incremental strategy in both of the ways

1. Defining it in dbt\_project.yml
2. Defining in the models sections where incremental strategy is defined

Strategy-specific configs**[​](https://docs.getdbt.com/docs/build/incremental-models" \l "strategy-specific-configs" \o "Direct link to Strategy-specific configs)**

***If you are using the merge strategy and have specified a unique\_key, by default, dbt will entirely overwrite matched rows with new values.***

1. **merge\_update\_columns**

On adapters which support the merge strategy (including **Snowflake, BigQuery, Apache Spark, and Databricks**), you may optionally pass a list of column names to a **merge\_update\_columns** config.

***In that case, dbt will update only the columns specified by the config, and keep the previous values of other columns.***

**models/my\_model.sql**

{{  
 config(  
 materialized = 'incremental',  
 unique\_key = 'id',  
  **merge\_update\_columns** = **['email', 'ip\_address'],**  
 ...  
 )  
}}  
  
select ...

Now only the specified columns in the merge\_update\_columns would be updated and the rest of the columns will be inserted or left as they were. Modifications are only made to the

1. **merge\_exclude\_columns**

Alternatively, you can specify a list of columns to exclude from being updated by passing a list of column names to a **merge\_exclude\_columns config.**

**models/my\_model.sql**

{{  
 config(  
 materialized = 'incremental',  
 unique\_key = 'id',  
 **merge\_exclude\_columns** **= ['created\_at'],**  
 ...  
 )  
}}

Now the specified columns in the merge\_exclude\_columns would be excluded and the rest of the columns would be updated as they do. The defined columns would be ignored.

Incremental Predicates:

Incremental predicates in dbt are an advanced feature that allows you to further optimize the processing of incremental models by limiting the data scan of the existing table. This can be particularly useful when dealing with large volumes of data.

Here's a simplified explanation of incremental predicates:

When running an incremental model, dbt needs to compare the new data with the existing data in the target table to determine what needs to be updated or inserted. By default, dbt scans the entire existing table.

However, with incremental predicates, you can specify conditions or filters that limit the scan of the existing table to a subset of data that is relevant for the incremental update. This can significantly reduce the processing time and improve performance.

To use incremental predicates, you need to configure the **incremental\_predicates** option in your model configuration. This option accepts a list of valid SQL expressions that define the conditions for limiting the scan. The expressions can reference the columns in the target table.

Examples:

let's say you have an incremental model that updates daily and you only want to scan the last 7 days of data in the existing table. Your **incremental\_predicates** configuration could be:

{{ config

( materialized = 'incremental',

**incremental\_predicates** = **["date\_column > dateadd(day, -7, current\_date)"]** )

}}

*In this case, dbt will generate a merge statement that includes the additional condition* ***date\_column > dateadd(day, -7, current\_date)*** *in the* ***ON*** *clause. This ensures that only the data from the last 7 days will be considered for the incremental update*.

By using incremental predicates, you can optimize the performance of your incremental models by reducing the amount of data that needs to be processed during each update.

Example 2

This an example of a model configuration in a yml file we might expect to see on Snowflake:

**In dbt\_project.yml**  
models:  
 - name: my\_incremental\_model  
 config:  
 materialized: incremental  
 unique\_key: id  
 *# this will affect how the data is stored on disk, and indexed to limit scans*  
 cluster\_by: ['session\_start']   
 **incremental\_strategy: merge**  
 *# this limits the scan of the existing table to the last 7 days of data*  
 **incremental\_predicates:** **["DBT\_INTERNAL\_DEST.session\_start > datediff(day, -7, current\_date)"]**

*# `incremental\_predicates` accepts a list of SQL statements.*   
 *# `DBT\_INTERNAL\_DEST` and `DBT\_INTERNAL\_SOURCE` are the standard aliases for the target table and temporary table, respectively, during an incremental run using the merge strategy.*

Alternatively, here are the same same configurations configured within a model file:

**In separate Models**

*-- in models/my\_incremental\_model.sql*  
{{  
 config(  
 materialized = 'incremental',  
 unique\_key = 'id',  
 cluster\_by = ['session\_start'],   
 **incremental\_strategy = 'merge',**  
 **incremental\_predicates** = **[  
 "DBT\_INTERNAL\_DEST.session\_start > datediff(day, -7, current\_date)"  
 ]**  
 )  
}}  
  
...

Incremental Strategies (Optional)

1. Merge: This is the default strategy used by many adapters, including Snowflake, BigQuery, Databricks, and Spark. It involves merging the new data with the existing data in the target table based on a unique key. The merge operation matches the rows based on the unique key and updates or inserts the data accordingly.

Example configuration for using the merge strategy:

sqlCopy code

{{ config

( materialized = 'incremental', unique\_key = 'id',

incremental\_strategy = 'merge' ) }}

1. Insert Overwrite: This strategy is an alternative to the merge strategy and is supported by adapters like BigQuery and Spark. Instead of performing a merge operation, it simply inserts the new data into the target table, overwriting any existing data with the same unique key.

Example configuration for using the insert overwrite strategy:

sqlCopy code

{{ config(

materialized = 'incremental', unique\_key = 'id',

incremental\_strategy = 'insert\_overwrite' ) }}

1. Append: This strategy involves appending the new data to the existing table without making any updates or replacements. It is commonly used when you want to maintain a historical record of all the changes over time.

Example configuration for using the append strategy:

sqlCopy code

{{ config(

materialized = 'incremental', unique\_key = 'id'

, incremental\_strategy = 'append' ) }}

1. Delete + Insert: This strategy, specific to Snowflake, involves deleting the existing rows with matching unique keys and then inserting the new data. It is an alternative to the merge strategy and can be useful in certain scenarios.

Example configuration for using the delete + insert strategy:

sqlCopy code

{{ config(

materialized = 'incremental', unique\_key = 'id',

incremental\_strategy = 'delete+insert' ) }}

The choice of incremental strategy depends on factors such as the volume of data, reliability of the unique key, performance considerations, and the capabilities of the underlying database or data warehouse. By selecting the appropriate incremental strategy, you can optimize the update process and ensure the incremental models reflect the latest changes in your data.