DBT Style Guide & Best Practices

Overarching Principles

While there are some hard-and-fast rules, a guide like this can never cover every scenario – nor should it. When in doubt, weigh your code against the following principles. By doing so, you and the team will feel more confident in the quality of your code.

Readability

Readable code means:

- Someone unfamiliar with the code can easily understand what is intended
- Points of complexity are obvious
- Code has flow and structure to it
- Formatting is consistent

Maintainability

Maintainable code means:

- Code adheres to design principles such as
    - Open Closed Principle
    - Single-responsibility Principle

Testability

Testable code means:

- Code does not have hidden / latent bugs or join fan-outs
- Code has fewer bugs that can ripple through the entire BI stack
- The team can make incremental changes to legacy code with confidence
- New people can join a project and feel confident and productive out of the gate

Tested code sets the example for everyone else. Be a leader 💯

Best Practices

Model configuration

- Model-specific attributes (like `sort`/`dist`/`partition` keys) should be specified in the model
- If a particular configuration applies to all models in a directory, it should be specified in the `dbt_project.yml` file
- In-model configurations should be specified like this:

```
{{
  config(
    materialized = 'table',
    sort = 'id',
    dist = 'id'
  )
}}
```

DBT conventions

- Only `bronze` models should select from `source`.
- All other models should only `ref` other models.

Testing

- Schema definitions should live in a `schemas` subfolder
- At a minimum, unique and not_null tests should be applied to the primary key of each model. All models should have a unique group of columns to test.

SQL Guidelines

Hard Rules 🥇

(Violating these will result in a rejected PR Example.)

- Testability issues
- Maintainability issues
- Failing builds

Soft Rules 🥈

(One or two should not hold up a PR, but gross violations will need to be fixed.)

- Readability issues

- Formatting issues

Unimportant 🥉

(Things that should never hold up a PR. The guide points this out simply to note that they are not of significant importance.)

- Mixing upper/lower case on SQL statements

SQL Language and Features 🥇

- Use of `select distinct` is not allowed 👎
    - Exceptions require architect approval
- Ordering and grouping by a number (eg. `group by` 1, 2) is preferred
    - Note that if you are grouping by more than a few columns, it may be worth revisiting your model design 🥈
- Prefer `union all` to `union` *
    - Understand the difference 💯

Column Naming Conventions 🥈

Dates and Timestamps

- Timestamps and dates should be named explicitly
- Timestamps should be named ending in `_ts`
- Dates should be named ending in `_date`
- Any raw source which does not conform to this should be renamed in bronze

E.g.:

> Amazon Marketplace Transaction data contains a column *posted_date* of type TIMESTAMP_NTZ. This should be renamed to *posted_ts* if we want to keep the timestamp type, or cast to DATE if we only want the date component.

Timezone Conversions

All timestamps should be normalized into UTC in bronze. This requires identifying what the source timestamp timezones are in. If they are already in UTC, no conversion is necessary.

All BRDs should also specify whether they want to define a primary reporting timezone. Some BI tools do this at query time. In this case, no further effort is necessary beyond normalizing all timestamps to UTC.

If a BRD requests a reporting timezone, then in `bronze`, create a mirror column for every timestamp in which the UTC value is converted to the reporting timezone. This column would end in the *_localtz* prefix.

Boolean values

Booleans should be prefixed with is_ or has_.

Database objects 🥈

- Schema, table, and column names should be in `snake_case`
- Table names should be plural, e.g. `accounts`
- Use names based on the *business* terminology, rather than the source terminology
- Price/revenue fields should be in decimal currency (e.g. `19.99` for $19.99; many app databases store prices as integers in cents)
    - If non-decimal currency is used, indicate this with suffix, e.g. `price_in_cents`
- Avoid reserved words as column names
- Consistency is key! Use the same field names across models where possible, e.g. a key to the `customers` table should be named `customer_id` rather than `user_id`
- Field names and function names should all be lowercase

Formatting and Style 🥈

DO NOT OPTIMIZE FOR A SMALLER NUMBER OF LINES OF CODE. NEWLINES ARE CHEAP, BRAIN TIME IS EXPENSIVE.

- Indents should be four spaces
- Long lines should be broken up over multiple lines if it improves readability (`case`)
- The `as` keyword should be used when aliasing a field or table
- Fields should be stated before aggregates / window functions
    - I.e. `group by` columns are always listed first.
- If joining two or more tables, *always* prefix your column names with the table alias
    - If only selecting from one table, prefixes are not needed
    - This makes it easier to understand which table the columns are referencing, i.e. if it is on the right or left table in a `join`

- Final select should always explicitly list columns – no `s.*`
    - This improves readability and usability when building models which consume core models
    - You shouldn't have to spend minutes digging around for the right columns
- Any clause with more than one item should be listed on newlines and indented
- Single items can be inline, e.g. `where foo = bar`
- `case` statements should begin and end with `case / end`
    - The rest should be indented
- Multiple Boolean conditions should be on different lines

```
...
case
    when something
        and another
        and even_more = 1
    then result
end as my_col,
...
```

- `or` conditions should be enclosed in parenthesis `()`, and extra care must be taken to ensure and and or statements do not get mixed up

```
where
    col1 = 1
    and col2 = 2
    and (
        col3 = 4
        or col4 = 4
    )
```

    -

Joins 🥈

- Default to `inner join` rather than `left join`
    - Use `left join` only when the right-side table may not have matches and you still want to select everything from the left-side
    - (This is often the case, but it shouldn't be your default join.)
- `right join` is not allowed 👎
    - Rewrite to use `left join`
- Any pre-filtering on a table in a `join` should happen within a Common Table Expression (CTE) before the join
- Do not filter on the right-side of a `left` join within the `where` predicate

    - This will filter out all `null` values, which negates the purpose of a `left outer join`
    - Instead, either filter in a CTE or filter in the `join` predicate
    - `left join right ON left.id = right.id AND right.column = 'foo'`
- Any complicated filtering on a joined table should happen in a CTE before the join
- Specify join keys – do not use `using`
    - Certain warehouses have inconsistencies in `using` results (specifically Snowflake)

Common Table Expressions (CTEs) 🥈

- Where performance permits, CTEs should perform a single, logical unit of work
- CTE names should be as verbose as needed to convey what they do
- CTEs with potentially confusing logic should be commented
- CTEs that are duplicated across models should be pulled out into their own models or macros

Example SQL:

```
with

my_data as (

    select * from {{ ref('my_data') }}

),

some_cte as (

    select *
    from {{ ref('some_cte') }}
    WHERE foo = 'bar'

),

select
    my_data.field_1,
    my_data.field_2,
    my_data.field_3,

    -- use line breaks to visually separate calculations into blocks
    case
        when my_data.cancellation_date is null
            and my_data.expiration_date is not null then expiration_date
        when my_data.cancellation_date is null then my_data.start_date + 7
```

```
        else my_data.cancellation_date
    end as cancellation_date,

    -- use a line break before aggregations
    sum(some_cte.field_4),
    max(some_cte.field_5)

from
    my_data
    left join some_cte
        on my_data.id = some_cte.id

where
    my_data.field_1 = 'abc'
    and (
        my_data.field_2 = 'def' or
        my_data.field_2 = 'ghi'
    )

group by 1, 2, 3, 4
having count(*) > 1
qualify row_number() over(partition by id order by timestamp) = 1
```

Model Definition 🥇

- Always keep models in subfolders relative to its use and source
    - `models/bronze/salesforce/`
    - `models/silver/dims/`
    - `models/gold/marketing/`
- Name models with the folder structure prefixed
    - `bronze_salesforce_opportunity.sql`
    - `silver_dims_opportunity.sql`
    - `gold_marketing_opportunity_conversion_dashboard.sql`
- Alias the table name in the model config

```
#gold_marketing_opportunity_conversion_dashboard.sql
{{
    config(
        alias="opportunity_conversion_dashboard"
    )
}}
```

- Specify schemas at the folder level in the project file
- Schemas should be named after the folder structure
- Schema name should be the same as what was removed from the table alias

```
models:
```

```
client_warehouse:
  gold:
    marketing:
      schema: gold_marketing
    finance:
      schema: gold_finance
```

Testability 🥇

- Each model should have a unique key defined in its `schema.yml` file
- Consider what is necessary to make a model unique – often, this consists of several columns

```
version: 2
models:
- name: my_model_name
  description: ''
  tests:
  - unique:
      column_name: "concat(user_id, event_name, timestamp)"
```

Jinja style guide 🥉

- When using Jinja delimiters, use spaces on the inside of your delimiter, like `{{ this }}` instead of `{{this}}`
- Use newlines to visually indicate logical blocks of Jinja

Review Guide

Mentorship

- Always assume the PR requester is doing their best
- PRs should be seen as a growth opportunity by the requestor, not a failure opportunity
    - How are you helping the requester to grow?

Keepers of standards

- Code should adhere to the core principles laid out in the style guide
- Change requests should be limited to:
    - Bugs
    - Poor readability
    - Poor maintainability

    - Insufficient tests
- The need for revisions should stem from the following questions:
    - How would a new person make a change to this code?
    - How obvious is it to alter a rule/logic
    - How likely would this result in bugs?

Respecting the reviewer's time

- Reviewers are not testers or bug fixers
- Requesters should respect the reviewer’s time by ensuring all necessary tests have been performed, example SQL has been provided, and the staging build has passed
- A review should not proceed until those conditions are met
- In most cases, pull requests should be brief and have minimal code changes. A massive dump of changes will need architecture approval

Review request process

- Set two reviewers: a primary reviewer and a secondary reviewer
- Assign review to primary reviewer in JIRA ticket
- If primary reviewer is unavailable, assign JIRA ticket to secondary reviewer

Review turnaround

- Reviewers are not required to immediately drop everything
- If reviewer is unable to review within an hour, reviewer should communicate back to the requester the ETA of a first review via Slack
- While the requester should give themselves sufficient time to have the review completed, merged, and deployed, the reviewer should also take into consideration the turnaround which may be required for revisions
- If a review is requested in the AM, it's reasonable to expect it to be reviewed by the PM
- If requested in the PM, then it's reasonable to expect it to be reviewed by the following AM
- If unable to review in a timely manner, re-assign the PR to the secondary reviewer