Introduction

In today's world, data is one of the most essential and powerful elements an organisation can have. Daily operations in an organisation can churn out loads of data stored in traditional data stacks. But nowadays, storing raw data will not be enough because raw data will not be able to give you actionable insights, reporting, data models, formulas, etc. You will only get these things when you analyse the raw data, and data modelling is an integral part of that.

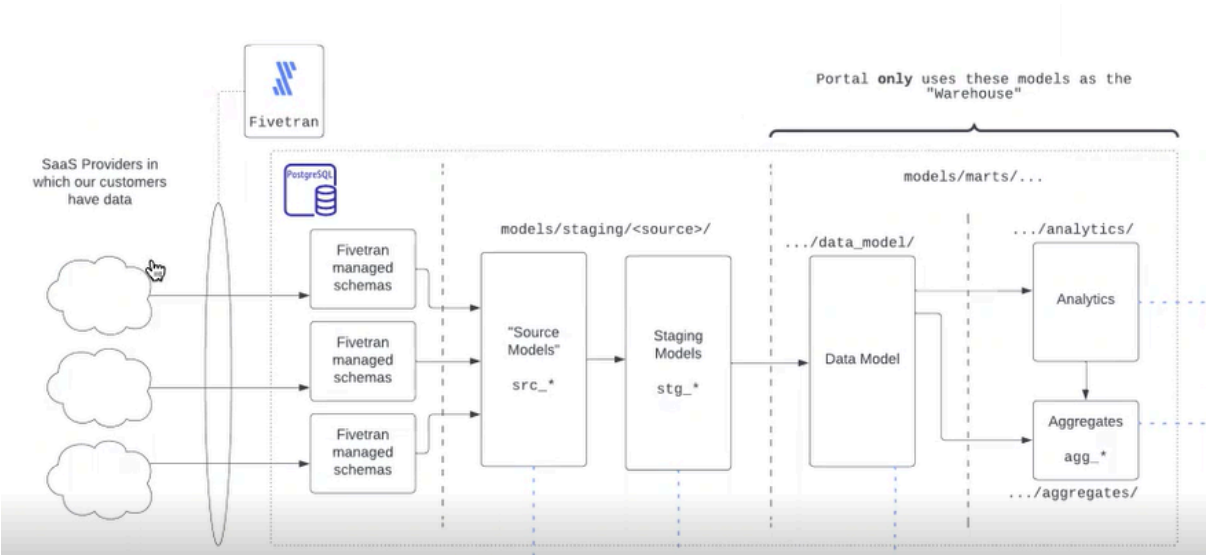
Statype helps with automated actionable insights of raw data powered by data decision models and expert formulas. This document will cover some of the subsets of Statype's data modelling, which includes the modelling pipelines, dbt general patterns, staging and media models, metrics, and much more. Let's start!

What is dbt?

Before jumping right into data modelling and pipelines, you first have to get an idea about dbt. dbt is a transformational workflow that helps get a substantial amount of work done with the simultaneous production of high-quality results. dbt helps in centralising and modularising your analytical code, which results in the collaboration on data models, and also helps with the versioning, testing, and documenting of the queries before deploying them to production. Throughout the process, you will get monitoring and visibility, which will be highly advantageous as it will raise alerts if any issue arises. In addition, the centralisation of your analytical code reduces errors whenever there is a change in logic.

A look into the data modelling structure of Statype

The data modelling structure shows how the data modelling implements the flow of data from the providers to the layer of source and staging models. You can see a schematic diagram of the modelling structure below:



The above picture shows that a load of data comes from the SaaS providers in the Fivetran in the PostgreSQL database. Fivetran acts as a connector between the data source and the PostgreSQL database. From the database, the data goes to the source models.

What are the source and staging models?

Here, source models are usual dbt-style staging models, which are used for changing the column name, and type or doing a very low-level cleanup of layers. These models are rebranded as source models because there is not always availability of one-to-one mapping between tables that exist from data sources in Fivetran. Therefore, you must do deep manipulations in the staging before entering the centralised data model.

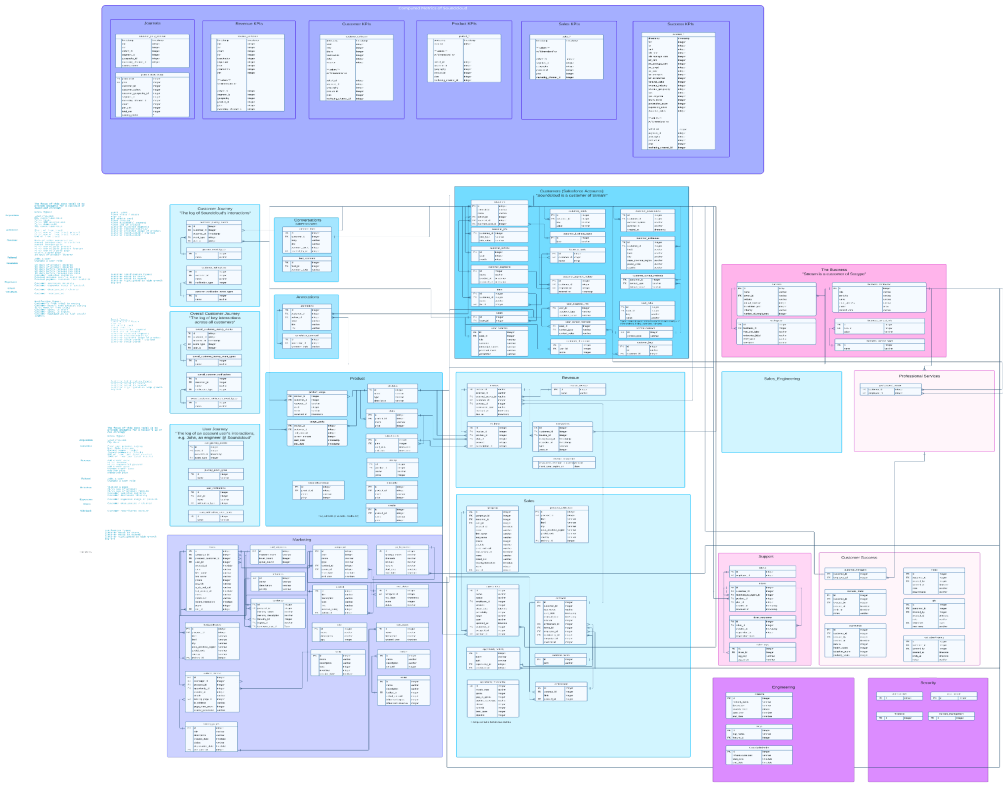
After getting low-level structural changes, the data goes to the pre-Data model layer. They are usually called source models because they can help join other models within a single source, like Salesforce or Stripe, to build a more robust model. The whole thing happens before the data enters Statype's centralised data model. The data models formed at the staging model can be merged later into the centralised data model. The centralised data model is a non-dbt and non-datawarehouse thing and is a big normalised relational data model.

Who are the customers in the data model?

In the data model, you will get lots of customer data in various tables and columns. Therefore, before we explain the data model, we must know who the customers referred to in the data model are. The customers are the customers of the business that is paying Statype. Businesses are customers of Statype who are paying Statype to get different insights on their data. So, in simple terms, the customers in the data model are the customers of Statype's customers.

Data model layer

The data model layer is highly normalised, like a transactional and analytical database. It encapsulates entities and complicated analytics insights that underlie when the data arrive. You can have a look at what a data model layer looks like in the below image:



The image shows how transactional data are encapsulated into non-datawarehousy denormalised narrow and wide columns according to their weights. From the data model, the data goes to the analytics and aggregates models.

What are aggregates and analytics models?

Aggregates are models where we can get all of the relevant information from the view of a single customer within the application. For example, it aggregates a bunch of tables containing information such as contact information and others into one place so that the web application called “Portal” only takes one query. By reducing the number of tables, aggregates reduce the complexities of queries.

Based on the principle of the normalising database, a notable thing about a super highly normalised database is that every piece of data in one place is stored exactly once. It is the exact opposite of using data warehouse modelling. The downside of using such modelling is the difficulty for the product engineering team to render customers as they have to ask for different data separately.

Furthermore, they often use joins, which hinders the performance, and also it becomes challenging to write as well. The intention is that the product engineering team is able to query either analytics or the data model, meaning that querying one query at a time to get anything needed essentially. A normalised database also helps to keep the business logic in one place, and in the case of Statype, it helps the “Portal” web app to function seamlessly by doing all the required SQL work on the backend.

Analytics models are derived from the data model and are useful for providing insights to Statype’s customers. They tend to consist of the time-series which backs all of the graphs in

the application. In addition, the time-series contains a column called the period, which is of the type called hit. The highest resolution grain for Statype so far is the day, and as a result, it is in the date type. Statype uses dates with the time zones because time zones are one of the most complex problems that will keep your sign along with concurrency. Hence, the dates are ingested, and the time zones are ignored completely.

Special models

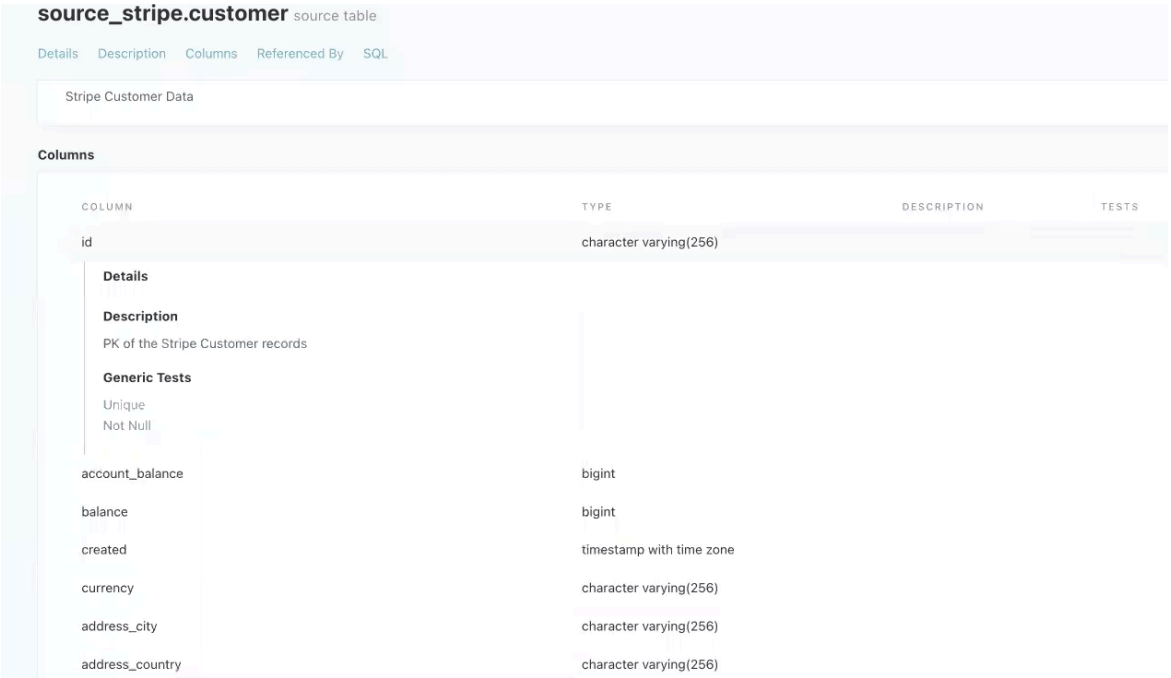
Now that we have discussed aggregates and analytics models, we will discuss a special model in the modelling structure. There is a join model called source_customers that allows the connection of customers from display sources that don't share an attribute. In addition, it contains separate records for all Statype's customers, which helps facilitate their customer records containing uncertainty at a later stage.

With this, you now have a basic understanding of the pipeline of the data modelling structure of Statype. Next, we will have a look at the source integrations.

Data sources and their implementations

As of today, Statype has integrated sources from Stripe and Salesforce. Sources are databases that contain a lot of information (commonly known as data). For instance, the Stripe source (written as source_stripe) is a PostgreSQL warehouse database having two tables and a schema. The two tables are the customer and invoice tables containing the customer and invoice data, respectively.

Now, if you look at one of the tables by clicking on them, you will find columns containing the type of data, their characters, their description, and the mode of tests associated with them. The below image shows what a table containing data columns looks like:



Now, you can expand a row by clicking on it, which will give better descriptive information about the data column. For example, in the above image, you can see that the row containing “id” is expanded. Upon expansion, the description of the data column is visible, along with a better overview of the performed test. Here, as you can see from the above image, “id” is the primary key of the stripe customer records, and upon performing generic tests, it gives unique and not null values. The table also contains the associated SQL query in the model.

The above description of “id” comes from using dbt constraints. The dbt constraints package has added the primary key attribute to the data column. Hence, a question might arise in your head regarding what a dbt constraint is, and we will have a look at it.

What are dbt constraints?

dbt constraints can be defined as a package that allows us to add primary keys, foreign keys, and other constraints based on the test result of a dbt project. The constraints are implemented by running a post-run hook within the dbt. The implementation of the package can only be done with the test data. If you run a test on your dbt project, it will check for constraints, and then only you will be able to enforce them.

So, let’s take an example with a yaml file. Below is a yaml file from Statype’s data model:

```yaml
version: 2

models:
  - name: contacts
    description: Statype's record of a Business' Contacts
    columns:
      - name: contact_id
        description: PK for Contact Records
        tests:
          - dbt_constraints.primary_key
      - name: name
        description: Contact Name
      - name: is_key_contact
        description: Whether or not Statype has identified this as a key contact
```

The above image shows a data model of an inner layer of contacts that will store contact information for businesses. It will contain the contact name, key contacts, contact ID, and other relevant information concerning a business’ contacts. In the model, a primary key constraint has been added to the contact ID, which helps in quick querying and guarantee uniqueness. The primary key for contact records will be run as a dbt test, and if there are any duplicate data, the test will fail as it will fail to maintain uniqueness.

Hence, it is evident that tests are an essential part when it comes to applying constraints, and in the next section, we will take a look at how testing works with dbt; and understand it from Statype’s perspective.

Testing with data models

Tests are assertions that you make about your models and other resources which help you get an assessment of the passing or failing of a project. Tests are needed to identify errors, monitor response times, etc., of your model that helps improve the project's quality. Since we are using dbt as our workflow, we need to know the basics of dbt testing.

Tests in dbt

In dbt, tests are SQL queries. You can use dbt tests to improve the integrity of your SQL queries based on the assertions you will make on the results generated from the tests. dbt tests help you identify whether a specified column in your model has non-null values, unique values, values from specific lists, or values that have a corresponding value in another model. Hence, you can turn any assertions made in the form of a select query into a test.

If you assert that every column in your model is unique and has non-null values, the test query selects for duplicates and will seek null values respectively. After this, if the test returns zero failing rows, it indicates that the test has been passed, validating your assertion. By running the command `dbt test`, you will get to know whether each test in your project has passed or failed. There are several modes of testing in dbt. Here, we will focus on the dbt unit tests.

What is unit testing in dbt?

A unit test in dbt can be considered a test in which mock inputs are provided to check the results against the asserted model. It is based on the software development practice of TDD or Test-Driven Development. In this practice, before you write a single line of code, you write the test that will exercise that code. As a result, the test will be read immediately and fail. After that, you will implement the code and the resources and wait for the test to pass.

dbt unit testing is highly advantageous and useful for SQL code using complicated business logic because this type of testing will help you provide mock inputs for all kinds of test cases, including edge cases. We will understand unit testing with the help of a test query.

dbt unit testing with Statype’s data model

To have a deeper understanding of the dbt unit testing, we will take a sample query from the bunch of unit test queries in the analytics package that were written based on our data model. The query with less granularity will be easy to observe and understand. Below is the image of a unit test query that is used for testing Statype’s cohorts_by_year_mrr_arr_monthly model:

```
-- depends_on: {{ ref('int_business_first_mrr') }}
{{
    config(
        tags=['unit-test'],
        now='2022-03-28'
    )
}}

{% call dbt_unit_testing.test('cohorts_by_year_mrr_arr_monthly', 'calculate the revenue per cohort as a function of cohort member MRR/ARR by retrieving the end
  {% call dbt_unit_testing.mock_source ('source_stripe', 'customer', {"input_format": "csv"}) %}
    id,name,email,created::timestamp
    'cus_1','Moisey Uretsky','moisey@statype.com','2021-11-23'
    'cus_4','David Worth','dave@statype.com','2021-11-23'

    'cus_5','Tammy Butow','tammy@statype.com','2022-01-01'
    'cus_6','Stafford Brooke','stafford@statype.com','2022-01-07'
  {% endcall %}

  {% call dbt_unit_testing.mock_source ('source_stripe', 'invoice', {"input_format": "csv"}) %}
    id,amount_due::int,created::timestamp,customer_id
    {# invoices grouped by month #}
    'in_01',5000,'2021-12-23','cus_1'
    'in_02',2000,'2021-12-23','cus_4'

    'in_05',6000,'2022-01-23','cus_1'
    'in_06',8000,'2022-01-23','cus_4'

    'in_08',4500,'2022-02-01','cus_5'
    'in_09',6600,'2022-02-07','cus_6'
    'in_10',3500,'2022-02-23','cus_1'
    'in_11',9000,'2022-02-23','cus_4'

    'in_15',2000,'2022-03-01','cus_5'
    'in_16',3300,'2022-03-07','cus_6'
    'in_18',3600,'2022-03-23','cus_1'
    'in_19',9100,'2022-03-23','cus_4'
  {% endcall %}
```

To understand the test query, firstly, you need to know the model for which the query is being written. Here, the model we need to understand is the cohorts_by_year_mrr_arr_monthly model. Cohorts can be defined as a group of customers who join in a month or a year. Cohorts_by_year means that you are opting for yearly buckets of cohorts instead of monthly. Fairly new companies opt for monthly cohorts as generating revenue within a short period becomes essential for them. The mrr_arr_monthly refers to the monthly and annual recurring revenue with a monthly grain associated. Hence, this model calculates the revenue earned per yearly cohort with the help of monthly and annual recurring revenue with a monthly grain.

In the above image, you can see that the source data and the customer data of stripe are being set up. The customer data shows the names of four employees of Statype, their email IDs, and their date of joining. Furthermore, you can also see the source data of stripe for invoices. It contains the invoice ID, amount, the date on which those invoices were created, and the customer ID. Also, the source data of stripe for customers and invoices are in CSV format. The amounts for invoice data are written in the form of pennies. Hence, the amount written in the form of 5000 denotes a 50 dollars bill.

In the test, the source data gets compared with the expected data. This is what a query of expected data looks like:

```
  {% call dbt_unit_testing.expect({"input_format": "csv"}) %}
    period::date,cohort::date,mrr::int,arr::int
    '2021-11-30','2021-01-01',0,0

    '2021-12-31','2021-01-01',7000,84000

    '2022-01-31','2021-01-01',14000,168000
    '2022-01-31','2022-01-01',0,0

    '2022-02-28','2021-01-01',12500,150000
    '2022-02-28','2022-01-01',11100,133200
  {% endcall %}
{% endcall %}
```

When you do the dbt test, the stripe's customer and invoice source data get loaded into the source tables in dbt. Then, the system runs the cohorts_by_year_mrr_arr_monthly model, and the generated results get compared with the expected results. If you look closely at the source data of stripe for customers, you will see that Moisey and David joined in the same cohort, and Tammy and Stafford joined in another cohort. So, it is expected that none of them can print money until the next month, and as a result, in the expected data, the MRR and ARR for the month of their joining remain zero. The months after that shows the expected revenue generated in the form of monthly and annual recurring revenues.

You will find the `now` tag in the config section at the start of the query. 28th of March is the date associated with that tag. This tag will make the system pretend that during the time of the test, the date is the 28th of March. The `now` function is associated with the `now` macro. Below is a query of the `now` macro:

```
{% macro now() %}
   {%- if 'unit-test' in config.get('tags') and config.get('now') -%}
      {{ "'" ~ config.get('now') ~ "'::timestamptz" }}
   {%- else -%}
      now()
   {%- endif -%}
{% endmacro %}
```

The macro shows that if Postgres's `now` function is in a unit test, it will return the value of `now` as a timestamp.

If you run the test in your terminal using the command `dbt test`, you will see that the test has been completed. The output in the terminal will look like this:

```
ustomer_id']
22:27:06  Creating foreign key: CUSTOMER_COHORTS_CUSTOMER_ID_FK referencing customers ['custom
er_id']
22:27:06  Creating foreign key: INVOICES_CUSTOMER_ID_FK referencing customers ['customer_id']
22:27:06  Creating foreign key: REVENUE_ITEMS_CUSTOMER_ID_FK referencing customers ['customer_
id']
22:27:06  Finished dbt Constraints
22:27:06  1 of 1 START hook: dbt_constraints.on-run-end.0 ............................... [RU
N]
22:27:06  1 of 1 OK hook: dbt_constraints.on-run-end.0 ................................... [OK
 in 0.00s]
22:27:06
22:27:06
22:27:06  Finished running 69 tests, 1 hook in 0 hours 0 minutes and 16.97 seconds (16.97s).
22:27:06
22:27:06  Completed successfully
22:27:06
22:27:06  Done. PASS=69 WARN=0 ERROR=0 SKIP=0 TOTAL=69
vscode ➜ /workspace/transformer (main) $
```

Completing a test doesn't necessarily mean passing the test. The following section will help you understand how to identify that a test has passed.

Identifying the passing of a test

A simple test with your test query can help you observe whether the test has passed. You can choose a query with a low resolution where the test values are more diminutive. It will help you understand easily. For this example, we will take the previous unit test query of cohorts_by_year_mrr_arr_monthly and change the stripe source invoice data. Hence, in the first row, we will change the due amount of 5000 to 5001. The expected output will look like this:

```
{% call dbt_unit_testing.mock_source ('source_stripe', 'invoice', {"input_format": "csv"}) %}
  id,amount_due::int,created::timestamp,customer_id
  {# invoices grouped by month #}
  'in_01',5001,'2021-12-23','cus_1'
  'in_02',2000,'2021-12-23','cus_4'
```

Now, if you run the command `dbt test` in the terminal, you will see warnings while the test is being performed. The below image shows the warning as an output in the terminal indicating that the test has failed:

```
e `except` (no prefix) instead. The transformer.business_customers_count_monthly model triggered this warning.
22:30:24  Warning: the `except` macro is now provided in dbt Core. It is no longer available in dbt_utils and backwards compatibility will be removed
e `except` (no prefix) instead. The transformer.business_customers_count_monthly model triggered this warning.
22:30:24  6 of 69 PASS business_customers_count_monthly ..................................... [PASS in 4.33s]
22:30:24  20 of 69 START test customer_associations ....................................... [RUN]
22:30:24  MODEL: model_name
22:30:24  TEST:  calculate the revenue per cohort as a function of cohort member MRR/ARR by retrieving the end of month ARR/MRR
22:30:24  ERROR: Rows mismatch:
22:30:24  | diff | count |     period     |     cohort     |  mrr  |  arr  |
22:30:24  | ---- | ----- | ---------- | ---------- | ---- | ----- |
22:30:24  | +    |     1 | 2021-12-31 | 2021-01-01 | 7001 | 84012 |
22:30:24  Warning: the `except` macro is now provided in dbt Core. It is no longer available in dbt_utils and backwards compatibility will be removed
e `except` (no prefix) instead. The transformer.contacts model triggered this warning.
22:30:24  Warning: the `except` macro is now provided in dbt Core. It is no longer available in dbt_utils and backwards compatibility will be removed
e `except` (no prefix) instead. The transformer.contacts model triggered this warning.
22:30:24  | -    |     1 | 2021-12-31 | 2021-01-01 | 7000 | 84000 |
22:30:24  17 of 69 FAIL 1 cohorts_by_year_mrr_arr_monthly ................................ [FAIL 1 in 2.43s]
22:30:24  21 of 69 START test customers__coalesce_duplicate_stripe_records ............... [RUN]
```

Now, as we have changed the due amount to 5001, it has compared with an MRR of 7001. The changed MRR gave a change in the value of the ARR as well. But in the expected value, we wanted a comparison with an MRR value of 7000; as a result, the test failed. Hence, with the changed value in the source data, the system ran the model and failed to

match the expected MRR and ARR related to that data. Changing the source data again to 5000 will pass the test without any warnings.

We have discussed everything related to testing with dbt and how we can do a unit test with one of our data models. Next, we will understand the structure of the analytics column of a model in the following sections.

Structuring a data model in an analytics package

We have seen the expected data of a unit test query in action, which comes with a structure comprising the period column, cohorts column, and values column. If we pick any other query, we will see the same format. For example, if we take a query representing the Lifetime Collective Revenue (LCR) with a monthly grain, we will see a structure like this:

```
27    {% set customer_id1=customer_surrogate_key('moisey@statype.com')%}
28    {% set customer_id2=customer_surrogate_key('liv@statype.com')%}
29
30    {% call dbt_unit_testing.expect({"input_format": "csv"}) %}
31      period::date,customer_id,total::int
32      '2021-12-01','{{ customer_id1 }}',5000
33      '2022-01-01','{{ customer_id1 }}',10000
34      '2022-02-01','{{ customer_id1 }}',15000
35      '2022-03-01','{{ customer_id1 }}',18000
36      '2022-04-01','{{ customer_id1 }}',21500
37      '2022-05-01','{{ customer_id1 }}',25000
38      '2022-06-01','{{ customer_id1 }}',28500
39      '2022-06-01','{{ customer_id2 }}',2400
40      '2022-07-01','{{ customer_id1 }}',28500
41      '2022-07-01','{{ customer_id2 }}',2400
42    {% endcall %}
43  {% endcall %}
```

The above image clearly shows that the format stays the same. Here again, you can notice that the structure of the expected data comprises a table containing periods as dates, a jinja macro that defines the customer ID linked with a customer surrogate key, and a value column.

Customer surrogate key

In this query, the customer_id1 is set to a customer surrogate key of the customer. The customer surrogate key is a dbt utils function, which in the case of Postgres is known as md5. md5 gives an output of a 128-bit value which can be easily represented as an integer but is often represented as the hexadecimal string value of the 128-bit output. It is challenging to create a sizeable normalised database with an integer Primary Key with the help of dbt. The surrogate key method takes a list of values, and md5 unifies them and

places them elsewhere. It is highly performant as it helps index a column and concatenates them.

We can view the customer surrogate key as a hexadecimal string by running an `echo` command in the terminal. The following command gives you an md5 value as a string:

``echo -n “” | md5``

Within the quotation, entering the customer email will give you the customer surrogate key as an output. Here is how it looks in a terminal:

```
~ via 💎v3.0.0 on ☁ (us-west-2)
❯ echo -n "moisey@statype.com" | md5
42f580724e0de98bf4136161fec447b8
```

The customer surrogate key of Moisey is extracted as an md5 value consisting of a hexadecimal string that can be considered an integer. So, this is how you can view a customer surrogate key which plays a significant role in indexing customer IDs. Next, we will have a look at the indexing.

Indexing a model

Indexing is an essential part of structuring and comes from the Postgres utilities. Indexing data models in the data_model package is simple because it requires indexing done in a transactional database. This involves indexing all primary keys. On the other hand, In the case of data models in the analytics package segmented by `period` and `customer_id`, like the ones we used previously, we will need three indexes which will be used on `period`, `consumer_id`, and a compound index that will be used on both `period and consumer_id` respectively.

Let’s take an example of a query of a model in the analytics package where the ‘business_mrr_arr_daily` model is run to calculate the monthly and annual recurring revenue of a business with a daily grain. It is given below:

```
{{ config(
    indexes=[
      {'columns': ['period']},
    ]
) }}

with

customers_mrr_arr_daily as (
    select * from {{ ref('customers_mrr_arr_daily') }}
),

business_mrr_arr_daily as (
    select
        period,
        sum(mrr) as mrr,
        sum(arr) as arr
    from customers_mrr_arr_daily
    group by 1
),

final as (
    select * from business_mrr_arr_daily
)

select * from final
```

The business_mrr_arr_daily model denotes the revenue collected by Statype's customers over a period. In the query, only one index has been used for the period because the period and the value column containing the MRR and the ARR values will be the only column present. When this model is deployed to the customers, there will be only one business consisting of all the customers' revenues.

In the above query, we have assumed the monthly and annual recurring revenue of all customers with a daily grain for the `business_mrr_arr_daily` model. The model used for calculating the customers' revenue is the `customers_mrr_arr_daily` model. Let's have a look at its query:

```
{{ config(
    indexes=[
      {'columns': ['period']},
      {'columns': ['customer_id']},
      {'columns': ['period', 'customer_id']},
    ]
) }}

with

customer_revenue_item_days as (
    select * from {{ ref('int_customer_revenue_item_days') }}
),

customer_mrr_daily_28d_with_gaps as (
    select
        period,
        customer_id,
        sum(
            daily_sum_for_customer
        ) over (
            partition by
                customer_id
            order by period rows between 27 preceding and current row
        ) as mrr
    from customer_revenue_item_days
),
```

In the query, you will observe that we have taken an index on `period`, `customer_id`, and on both ‘period` and `customer_id`, which can be defined as range queries. So, there will be four columns containing `period` as dates, `customer_id`, MRR and ARR values, and the revenue ranging from the first to the last date of the month.

This is how indexing is done in a data model in an analytics package. It helps in generating graphs and locating data in databases quite easily and quickly. In the following segment, we will look at the lineage graphs generated through complex computations of data models.

Lineage graphs

Directed Acrylic Graphs (DAG) or lineage graphs show the transformation and consumption of data during the movement of data that exists for the data transformations in an organisation. A lineage graph helps you visualise the nodes that must come before a current model, also known as upstream dependencies. It also gives you the downstream relationships with the work that the current model has impacted.

As lineage graphs are directional, they display a defined flow of movement and form loops that are non-cyclical in nature. In addition, the graphs show the functional relationships between the data sources, models, and dashboards. Finally, the bottlenecks and inefficiencies in data work can be seen effectively with the help of a lineage graph.

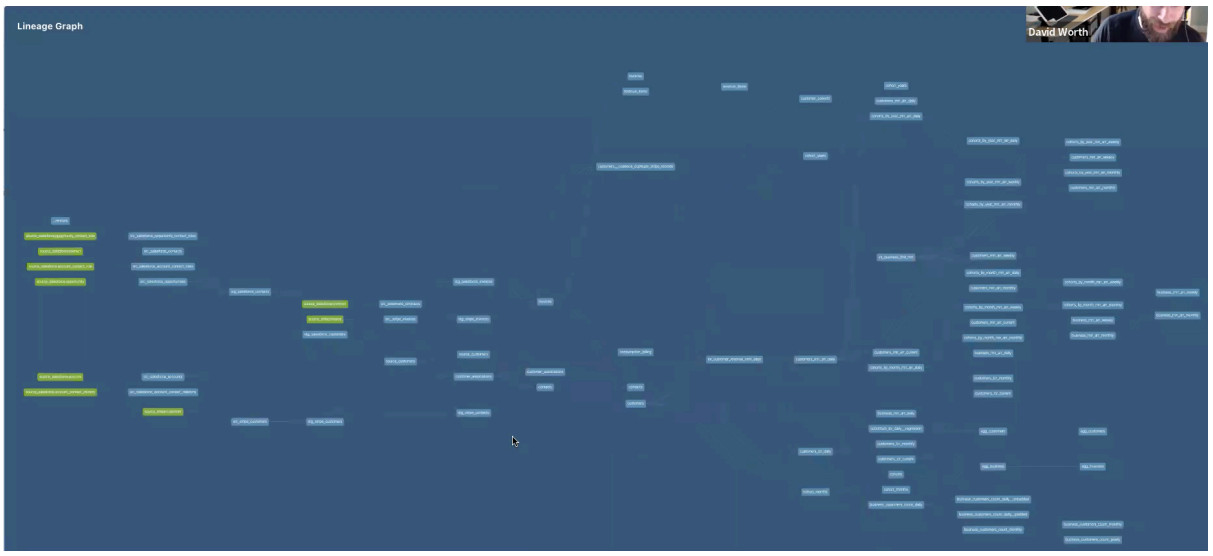
Why is data lineage important?

Data lineage is an essential aspect of analytics engineering. Through data lineage, you will get a holistic view of the data pipeline and the movement of data within an organisation, including the transformations and consumptions. It is usually represented through directional graphs and data catalogues.

The overall detailed view of the data pipeline is advantageous to the data team as it helps them build, analyse, and troubleshoot workflows more effectively. In addition, data lineage helps reduce headaches during root cause analysis, minimises unexpected downstream problems that usually occur while making upstream changes, and empowers business owners by helping them understand the origins of reporting data and providing means for data discovery.

Plotting a lineage graph with a Statype dataset

Below is a picture of a giant lineage graph plotted with the help of the data obtained from Statype’s Stripe integration:



On the left, you will observe ``_versions`` from where the directional flow of the graph starts. It is where all the build versions are stored. Build versions are useful only in the sample and production warehouses to denote the freshness of Statype's customers. The graph also shows the various sources along with the data flows.

In the next part, we will take a closer look at one of the data models that helps calculate monthly and annual recurring revenue with a daily grain for a business.

Calculating the ARR/MRR with a daily grain of a customer

The ``customers_mrr_arr_daily`` model helps calculate annual and monthly recurring revenue with a daily grain for Statype's customers from their data. It is a complex data model that deals with many complications. To understand the concept of MRR and ARR deeply, we have to understand the backstory behind customer data. The data used in the model are the Stripe data from one of Statype's customers. Most of the customers of the business that gave Statype data pay them monthly on a usage basis. In addition, the customer sells products related to APIs, and as a result, they have a bunch of different plans. However, the plans help them have the same number of monthly API calls. Also, a few of the plans got interesting details to deal with the company's Stripe invoice data for revenue. The invoice data helps in generating MRR/ARR on the page as graphs backed by data.

What are overages?

For the above business, the majority of their customers get billed monthly on their anniversary or sign-up date for a given product. For example, if the sign-up date is on the 3rd of the month, the customer will get invoiced every third of a month for an amount. The invoice generally includes the plan's cost for that product and the overages. The overages are a one-off thing, but the real question arises whether we should add them to the monthly or annual recurring revenues. But as we have previously discussed, this business has custom plans for its customers, and as a result, they have predictable revenue. Hence, there will be no overages for most cases.

Moreover, the invoice total will be considered for the model rather than the invoice line items. For example, suppose we try to distinguish between invoice total and line items with respect to Statype's recurring revenue data model for the customer. In that case, it will be seen that the model only looks at the amount of money taken in the last month per customer. Hence, if there are any overages, they will be added to the Monthly Recurring Revenue.

Why are overages included in the MRR calculation?

Monthly Recurring Revenue answers the reasonable question of the amount of money a business gets per month from a particular customer. However, as overages are extra expenses on rare occasions, it usually gets omitted from the MRR calculation. Figuring out an overage from a regular bill is very difficult because it requires high granularity in the data. But as of the moment, Statype doesn't have that granularity level on the data; hence, we include overages in our calculation from the total invoice values. Also, it is certain that we will try to eliminate overages in the future.

What is the importance of periods in total invoice values?

The total invoice values for any given period are generated on the customer's anniversary date. So, for example, if one customer has joined the business on the 3rd of a month, the invoice values will be billed on the 3rd of every subsequent month. As a result, the monthly recurring revenue of every month is different because not all months have the same number of days, and it also does impact the annual recurring revenue in case of a leap year.

Let us assume a company with a product that charges every minute. As a customer, you joined that business on the 28th of February. In that case, you will be billed on the 28th of every month, meaning you will be charged for 28 days worth of minutes; hence, there will be months when you will get free access to that product for 2 to 3 days.

Similarly, if you join that same business on the 31st of any month, you will be charged for 31 days worth of minutes for the subsequent months, irrespective of the total number of days per month. So now, for the same company, if they have two different products and you are a customer of both products, you will be billed separately and have separate invoices. But in the MRR, both the values from the invoices will be added and calculated.

Calculating MRR/ARR daily with a 28-day rolling average

Suppose a customer gets billed every 31st for the anniversary of his joining and then gets billed every 3rd and 9th for the two other products. Here, the problem arises because the joining anniversary is at the last of the month. Hence, after paying every last of a month, the customer has to pay on two subsequent dates at the start of the following month. Sometimes, this can go against the will, and to address this problem, the MRR with a daily grain can be calculated with a 28-day rolling average. Also, if we use a rolling average of 30 days instead of 28, there is a chance of double-counting billings at both ends of the month because you will often get two billing cycles.

Due to the double-counting of billings for two different months, the revenues will be combined, and the business will see a sharp spike in their revenue for a particular segment. But it will drop once the old bills are rolled up, harming the business.

The following image will show how the 28-day rolling average has been used in the `customers_mrr_arr_daily.sql` query:

```
{{ config(
    indexes=[
      {'columns': ['period']},
      {'columns': ['customer_id']},
      {'columns': ['period', 'customer_id']},
    ]
) }}

with

customer_revenue_item_days as (
    select * from {{ ref('int_customer_revenue_item_days') }}
),

customer_mrr_daily_28d_with_gaps as (
    select
        period,
        customer_id,
        sum(
            daily_sum_for_customer
        ) over (
            partition by
                customer_id
            order by period rows between 27 preceding and current row
        ) as mrr
    from customer_revenue_item_days
),

```

In the query, we see that customer revenue item days have been pulled from the intermediate customer revenue item days. The intermediate customer revenue item days is an intermediate model denoted by the `int_customer_revenue_item_days.sql` query. This model includes the cross-joining of unique customer IDs. Furthermore, the particular table generated from this model contains the period, customer IDs, and the daily sum of all customer revenues in three columns. The code depicting the above can be seen below:

```
final as (
    select
        recognized_date::date as period,
        customer_id,
        daily_sum_for_customer
    from customer_revenue_item_days
)

select * from final
```

Hence, this model sums up all of the everyday revenue models per customer. So, for example, if you are a customer getting two invoices in a day, the value of both invoices will be summed up and put in the table. Next, in the previous image, you can see the use of the windowing function with customer IDs as identifiers to get the rolling average of 28 days. Although we are calculating the daily grain MRR with a rolling average of 28 days, we have to order between the 27 preceding and the current row according to the period or date because that's how PostgreSQL works.

After this, we will understand how the MRR and ARR can be calculated from the months having 30 or 31 days. The following code from the `customers_mrr_arr_daily` model will help us get a clear picture:

```
customer_mrr_daily_with_gap_closure as (
    select
        period,
        customer_id,
        case
            when mrr != 0 then mrr
            else coalesce(greatest(
                lag(mrr, 1) over (partition by customer_id order by period),
                lag(mrr, 2) over (partition by customer_id order by period),
                lag(mrr, 3) over (partition by customer_id order by period)
                ), 0)
        end as mrr
    from customer_mrr_daily_28d_with_gaps
),

final as (
    select
        period,
        customer_id,
        mrr,
        mrr * 12 as arr
    from customer_mrr_daily_with_gap_closure
)

select * from final
```

To get to the fact that there are 30 or 31 days months, we can see that without this `customer_mrr_daily_with_gap_closure` on non-February months, there are 2 or 3 days which represents zero revenues in the middle that differentiates between the 28-day average and 30 or 31-day average. Hence, as done in the code, we have to do the coalescence of the initial values that are non-zero before the gap and fill up the three days with them. The coalescence will take place between the greatest of the three non-zero values or nulls and zero. Also, note that if there are no prior non-zero values that can fill the gap of those three days, you have to consider them nulls.

Next, if you multiply the generated MRR by 12 from `customers_mrr_daily_with_gap_closure`, you will get the annual recurring revenue of a customer.

What are the compromises made when calculating MRR with gap closure?

The above process of calculation does have some downsides. For example, suppose a customer stops giving money to the business; then, there will be three days of non-zero revenue at the end of the churn. Hence, it will look like the customer is in the plan for an extra three days, which might turn into a month as those three days are crossing the month boundary. But it is a minor concern because it gives us a value much closer to the one we deserve.

Difficulties in calculating MRR/ARR with contractual-based payments of a business

The Statype customer that sells products related to API has customers that get billed monthly on their anniversary date. But they also have customers who pay a lump sum of money upfront, equivalent to getting a service for 6 to 12 months as part of a contract. Unfortunately, due to the lack of contract information, we do not divide that amount by 6 or 12. As a result, there are occurrences of funny spikes in their revenue, which is a problem that needs to be solved in due time.

Moreover, we don't have a deep understanding of the product. We don't know whether there will be any secondary effect as part of the spikes in the graph, or we can have completely different sets of data based on the contract that can feed into the revenue metrics. Now, if we divide the lump sum amount by 6 or 12, depending on the month, we will get a flat and uninteresting MRR/ARR until any sign of renewals. Although, the generated value will not affect any revenues of any other segments. Again, this problem can't be fixed within the `customers_mrr_arr_daily` model but can be fixed within the intermediate `int_customer_revenue_item_days` model.

Understanding the `int_customer_revenue_item_days` model

The intent of the intermediate `int_customer_revenue_item_days` model is to groove a given customer's revenue model on a daily basis. Therefore, it creates a time-series metric for every customer for the invoiced amount mounting daily from the first day of the invoice. As mentioned in a segment earlier, this model gives out a data table with three columns. It

consists of the periods, which refers to the day in which the customer gets invoiced, the customer_ids, which serve as a unique identifier for the invoice created per customer, and the daily_sum_for_customer which tells us the total of all the invoices that have been issued to a customer in a single day. Below is the query of this intermediate model that will give us a picture of how this model will work:

```
with

customer_revenue_items as (
    select
        customer_id,
        recognized_date,
        sum(amount) as daily_sum_for_customer
    from {{ ref('revenue_items') }}
    group by 1, 2
),

first_revenue_item_date as (
    select min(recognized_date) as recognized_date
    from customer_revenue_items
),

day_spine as (
    select generate_series(
        first_revenue_item_date.recognized_date,
        {{ now() }},
        interval '1 day'
    ) as recognized_date
    from first_revenue_item_date
),
```

The `customer_revenue_items` are essentially invoice items that can be separated with specific naming from Stripe in accordance with the actual data model. Next, the `first_revenue_item_date` can be considered the very first date we have a revenue model. This data is essential as it is a stepping stone for building the data model. After that comes the `day_spine`, a daily period spine often denoted as a series of every single day from the date of the very first revenue model as a date.

```
28  unique_customer_ids as (
29      select distinct
30          customer_id,
31          date_trunc('day', created_at) as created_date
32      from {{ ref('customers') }}
33  ),
34
35  -- create the cartesian product of unique customers and the day on which
36  -- the customer was created so we don't generate revenue data before they
37  -- joined.
38  customer_days as (
39      select
40          day_spine.recognized_date,
41          unique_customer_ids.customer_id
42      from day_spine
43      cross join unique_customer_ids
44      where day_spine.recognized_date >= unique_customer_ids.created_date
45  ),
46
47  customer_revenue_item_days as (
48      select
49          customer_days.recognized_date,
50          customer_days.customer_id,
51          coalesce(
52              customer_revenue_items.daily_sum_for_customer, 0
53          ) as daily_sum_for_customer
54      from customer_days
55      left outer join
56          customer_revenue_items on
57              customer_days.recognized_date
58              = customer_revenue_items.recognized_date
59              and customer_days.customer_id
60              = customer_revenue_items.customer_id
61  ),
```

`unique_customer_ids` are required to build the correct cross-join with all dates. Multiple customers can have the same email address, but they can be differentiated with unique customer IDs associated with the creation date. The unique customer IDs are linked up every single day with the day the customer is created. Hence, it will list the company's revenue every day from the date of the first revenue model and then combine the unique customer IDs after the customers have been created. So, it will not make sense for a customer joining in 2022 to have a customer ID with a date of 2017 for revenue items.

After customer IDs, you will find `customer_days`, which shows the revenue data that the customers can have every single day. And then, the joining happens with the recognition date of the customer revenue items to the day the customer actually exists. Finally, the model gives the output of a table containing the customer's dates of their first invoice, their IDs, and the daily sum of all the invoices.