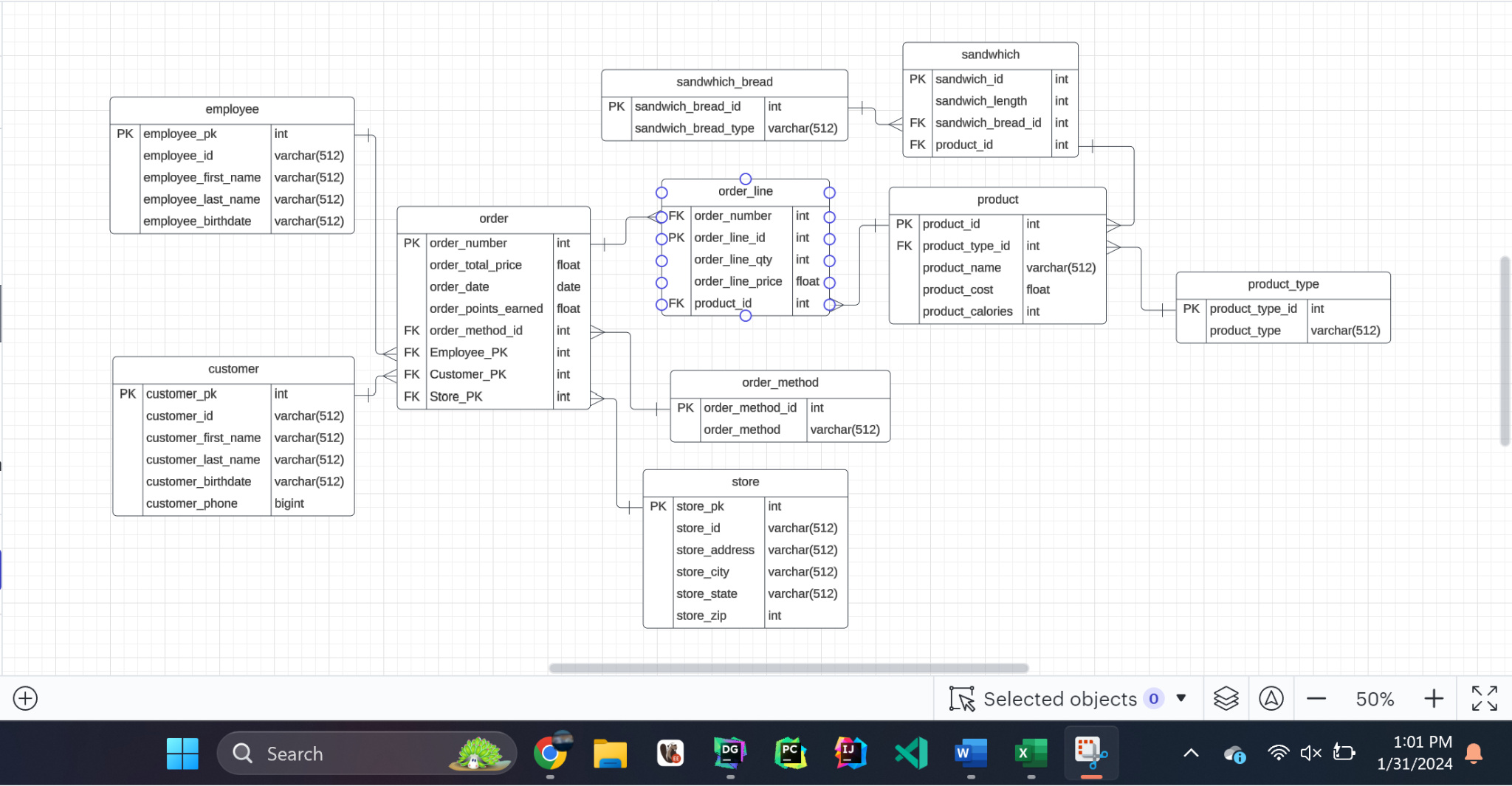
**Sam’s Subs Design Guide**

**Normalization and Database Creation**

To begin our creation of a better storage and analytics solution with Sam’s Sub

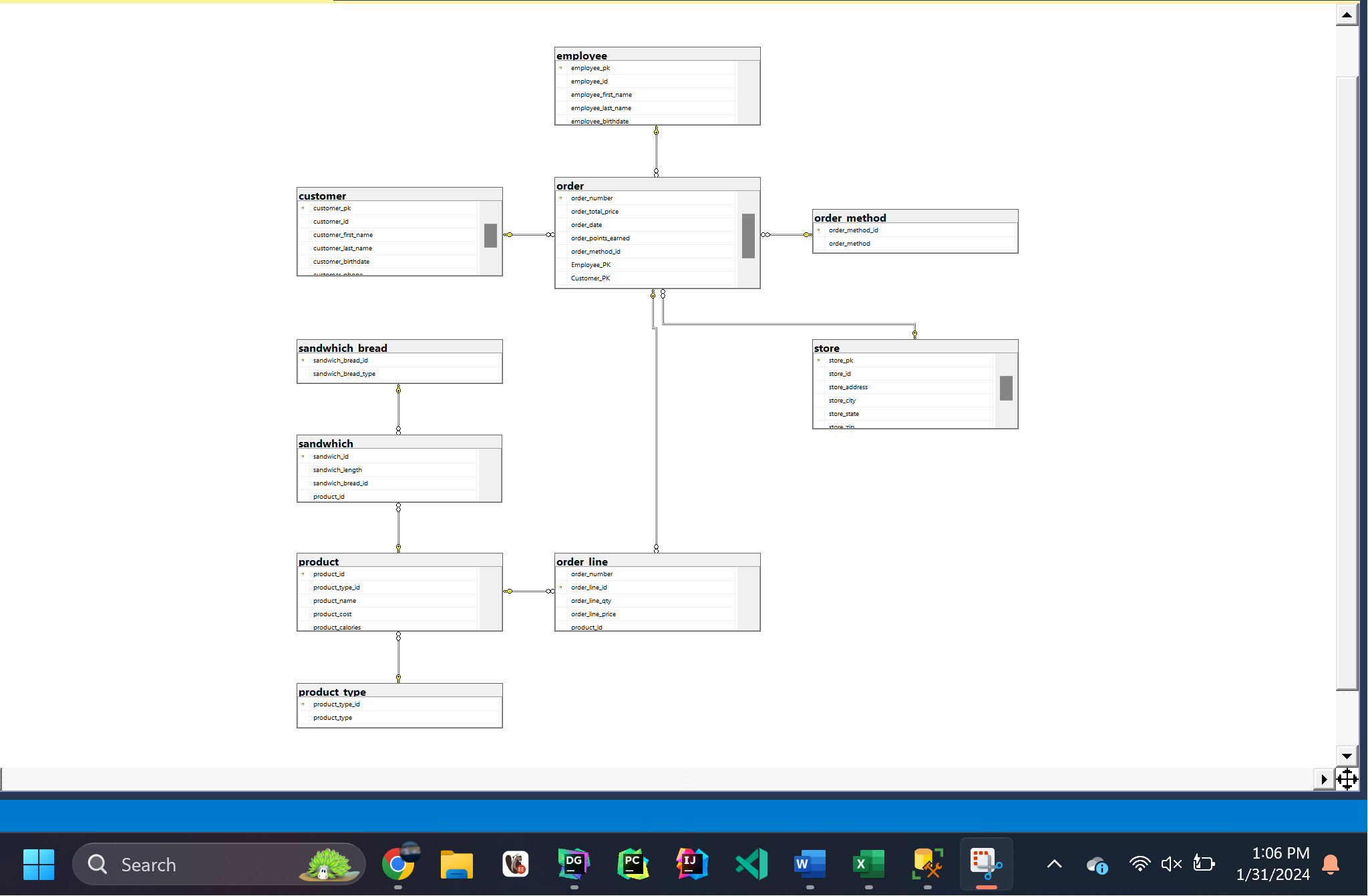
s we started with a focus on developing an entity relationship diagram (ERD) and normalizing the data. Our ERD served as a roadmap for how to organize our data and how to normalize it into third normal form. We created our ERD using Lucidchart, the diagram is pictured below.



After creating the diagram we began the normalization process, separating our data into the correct tables from the original Excel file. While normalizing the main data quality problem we ran into was duplicate records. When we identified duplicates in the data we handled them in two different ways:

1. If an order contained multiple lines with the same customer name and employee name and amounts we split the order number into two different orders, for example, order 983 was split into order 983 and order 1501, while order 984 got split into order 984 and order 1502.
2. In the event there were singular duplicate records that differed in only location and employee\_PK these records were deleted.

At the end of normalizing our data, we had an Excel document with our data organized by their intended tables. To move this to a database we created several tables in Microsoft SQL Server Management Studio, using LucidChart and tableconvert.com to generate our DDL and DML to transform from the Excel doc to a queryable database.



To validate that our process had loaded correctly we answered the following three business questions:

1. A manager is reporting seeing duplicate values in the order table, write a query to identify that there are only unique records in the table.

SELECT DISTINCT(order\_number)

from [order]

where order\_number IN (select order\_number

from [order]

group by order\_number

having count(order\_number) > 1)

order by order\_number

;

1. Identify the most common sandwich sold by store

SELECT T1.store\_id, T1.product\_name, T1.store\_state, T1.QtySold

FROM (SELECT s.store\_id,

p.product\_name,

s.store\_state,

COUNT(ol.order\_line\_id)

AS QtySold,

ROW\_NUMBER() OVER (PARTITION BY s.store\_id ORDER BY

COUNT(ol.order\_line\_id) DESC) AS Rank

FROM [order] o

INNER JOIN store s ON o.Store\_PK = s.store\_pk

INNER JOIN order\_line ol ON o.order\_number = ol.order\_number

INNER JOIN product p ON ol.product\_id = p.product\_id

RIGHT JOIN sandwhich sw ON p.product\_id = sw.product\_id

GROUP BY s.store\_id,

p.product\_name,

s.store\_state) T1

WHERE T1.RANK = 1

;

1. A manager wants to explore which states can grow their online ordering the most, identify the top 3 states with the largest difference to the state with the greatest online ordering.

SELECT TOP 3 om.order\_method, s.store\_state, COUNT(o.order\_number) AS [Qty of Orders],

CONCAT((COUNT(o.order\_number) \* 100 / MAX(COUNT(o.order\_number)) OVER()), '%') [StatesQty/TopStatesQty]

FROM [order] o

INNER JOIN order\_method om ON (o.order\_method\_id = om.order\_method\_id AND om.order\_method = 'Online')

INNER JOIN store s ON o.Store\_PK = s.store\_pk

GROUP BY om.order\_method,

s.store\_state

ORDER BY [Qty of Orders]

**Dimensional Modeling**

After review we decided to condense our model in the following ways:

* The sandwich bread table became a single column in our sandwich table
* Product\_Type table was moved to be a column in Product Dimension
* Order\_Method was moved to be part of Orders

With those changes made we turned our focus to designing and implementing a data warehouse that improved Sam’s Subs reporting and analytics capabilities. To do this we planned to transition our relational database to a dimensional model. We were given guidance that Sam’s Subs priority business processes were 1) Customers placing an order and 2) Tracking inventory.

To evaluate how these processes would intersect in a functional data warehouse we created the following Bus Diagram:

| Business process | Sandwich | Product | Customer | Employees | Store | Date | Inventory |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Orders | x | x | x | x | x | x |  |
| Inventory |  |  |  |  | x | x | x |

After clarifying which dimensions we felt would be shared we started the process of designing a Star Schema. We focused primarily on the process of customers placing an order as we were told that was the priority. We followed a four step process described below to guide our creation of a star schema centered around an Order level Fact table.

1. Select the business process (Orders)

The business process we are tracking is customers making meal orders.

1. Declare the grain (Orders)

Our grain is at the order line item level, allowing us to track each item purchased.

1. Identify Dimensions (Orders)

Our Dimensions of choice would be: Order Number, Date, Product ID, Customer ID,

Store ID, Order Line ID.

1. Identify Facts (Orders)

Our facts of interest for our Orders process would be: Total Cost, Rewards Total, Order

Method, Order Line ID, Order Line Quantity

We repeated the process with less intensity for the Inventory process as well.

1. Select the business process (Inventory)

Our second business process is keeping Inventory of items used to make meal orders.

1. Declare the grain (Inventory)

The grain would be a daily snapshot at the level of Ingredients.

1. Identify Dimensions (Inventory)

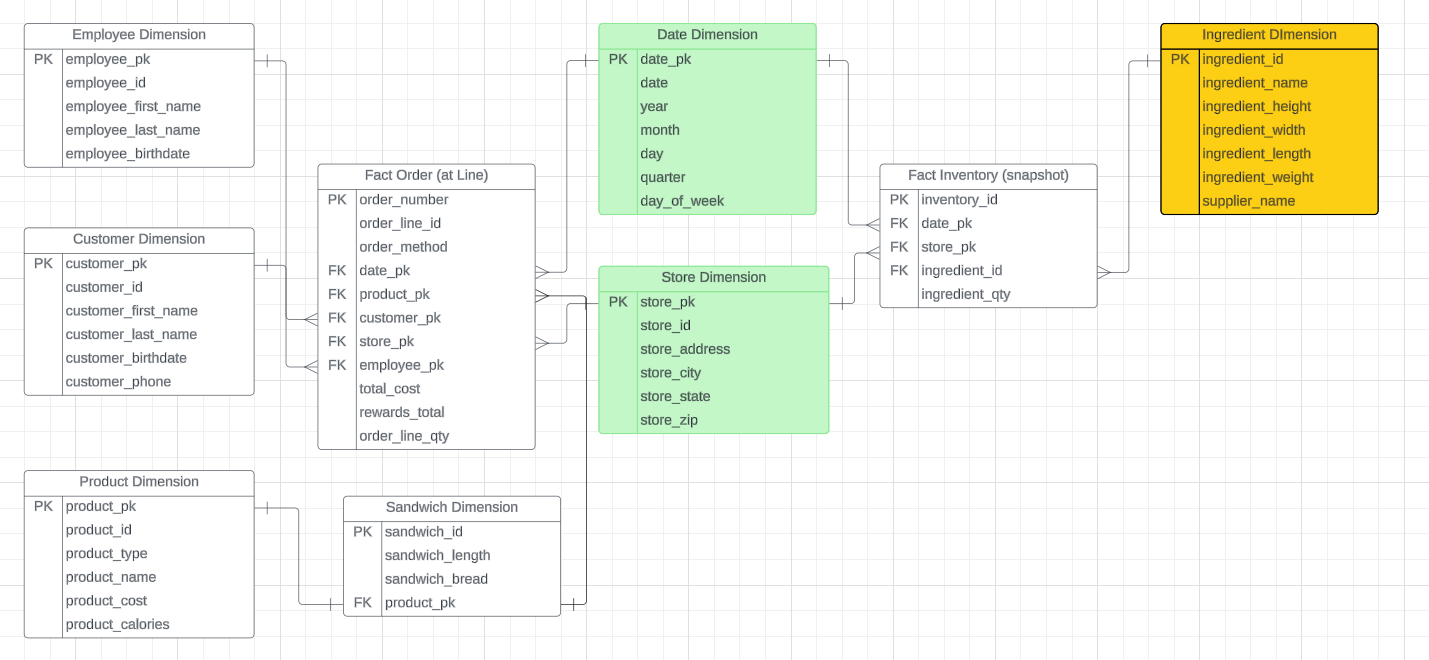
To best track Inventory we would note the following dimensions: Inventory ID, Date ID,

Store ID, Ingredient ID.

1. Identify Facts (Inventory)

Our fact to track for the Inventory table would be the Ingredient Quantity.

After creating those two outlines for star schemas we combined them into one enterprise database:



As can be seen highlighted in green, our star schemas share the conformed dimensions of Date and Store. This allows us to keep this information consistent across use cases. The additional dimension table (excluding the Inventory Fact) is the ingredients table. This is denoted in Yellow. The inventory is currently a standalone table. This is to provide ease of use and to keep the data from becoming too

normalized. As a result, we left inventory as a daily snapshot instead of a line\_item. The line\_item level would make it more of a transactional database in which PO Line information and Order Line information would in real time automate the inventory.

**ETL Implementation**

Based on our discussions with Sam’s Subs and on feedback received we chose to create surrogate

keys for our dimensional model, using the “\_Key” syntax. We also used hashed keys for those surrogates, taking the “\_ID” fields and any other relevant fields (customer\_last\_name, employee\_last\_name, etc.) We felt that this would allow us the greatest security and dependability for our data. We did move order\_line\_id and order\_number to the orderline\_dim table for greater clarity. We also changed the sandwich table by moving the information stored there into the product dim table. We elected to keep Supplier Name in the Ingredient dimension to allow us to most effectively track which ingredients came primarily from which supplier. With expanded data availability this or greater business need tables leading to our Inventrory\_Fact could be expanded. We kept order method in the order\_fact table and order table due to only having a couple values and seeing more benefit in avoiding a join.

After these changes we were prepared to implement an Extract, Load, and Transform (ELT) process to transition data from our SQL Server database to the enterprise data warehouse hosted in Snowflake.

I will include a brief conceptual overview before providing the code required to perform this process. We first used Airbyte to move data from our SQL Server setup up to Snowflake. We also used Docker in this process. VSCode and dbt allowed us to populate the dimensional model. To help ensure the transfer was complete and accurate we answered the following potential business questions:  
  
1. How many different products does each store sell per day?

SELECT s.store\_id,

d.date\_day,

COUNT(DISTINCT p.product\_id) AS num\_products\_sold

FROM DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_fact\_order AS o

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_orderline AS ol ON o.order\_number =

ol.order\_number

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_customer AS c ON o.customer\_key =

c.customer\_key

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_employee AS e ON o.employee\_key =

e.employee\_key

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_date AS d ON o.date\_key = d.date\_day

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_store AS s ON o.store\_key =

s.store\_key

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_product AS p ON ol.product\_key =

p.product\_key

GROUP BY s.store\_id,

d.date\_day;

2. How many different stores does each customer visit by store?

SELECT c.customer\_pk,

COUNT(DISTINCT s.store\_id) AS num\_stores\_visited

FROM DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_fact\_order AS o

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_customer AS c ON o.customer\_key =

c.customer\_key

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_store AS s ON o.store\_key =

s.store\_key

GROUP BY c.customer\_pk;

3. On which days are the most products sold, separated by store? The least amount of Products?

--Query to find which products are most sold

SELECT s.store\_id,

d.date\_day AS most\_products\_sold\_day,

COUNT(ol.order\_line\_id) AS num\_products\_sold

FROM DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_fact\_order AS o

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_orderline AS ol ON o.order\_number =

ol.order\_number

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_date AS d ON o.date\_key = d.date\_day

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_store AS s ON o.store\_key =

s.store\_key

GROUP BY s.store\_id,

d.date\_day

HAVING COUNT(ol.order\_line\_id) = (SELECT MAX(product\_count)

FROM (SELECT s.store\_id,

d.date\_day,

COUNT(ol.order\_line\_id) AS product\_count

FROM DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_fact\_order AS

o

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_orderline AS

ol

ON o.order\_number = ol.order\_number

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_date AS d ON

o.date\_key = d.date\_day

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_store AS s

ON o.store\_key = s.store\_key

GROUP BY s.store\_id,

d.date\_day) AS counts\_per\_day

WHERE counts\_per\_day.store\_id = s.store\_id);

-- Query to find the days with the least products sold by store

SELECT s.store\_id,

d.date\_day AS least\_products\_sold\_day,

COUNT(ol.order\_line\_id) AS num\_products\_sold

FROM DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_fact\_order AS o

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_orderline AS ol ON o.order\_number =

ol.order\_number

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_date AS d ON o.date\_key = d.date\_day

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_store AS s ON o.store\_key =

s.store\_key

GROUP BY s.store\_id,

d.date\_day

HAVING COUNT(ol.order\_line\_id) = (SELECT MIN(product\_count)

FROM (SELECT s.store\_id,

d.date\_day,

COUNT(ol.order\_line\_id) AS product\_count

FROM DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_fact\_order AS

o

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_orderline AS

ol

ON o.order\_number = ol.order\_number

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_date AS d ON

o.date\_key = d.date\_day

LEFT JOIN

DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_store AS s

ON o.store\_key = s.store\_key

GROUP BY s.store\_id,

d.date\_day) AS counts\_per\_day

WHERE counts\_per\_day.store\_id = s.store\_id);

We also created this visualization from our data to show a potential use case for our new warehouse

///Begin Technical details///

1. Use Airbyte and place this data in some temp tables in a schema

in group{#]project database in Snowflake. You need to create this database

in Snowflake.

# Airbyte Instructions #

- Open the Docker application

- Start Airbyte by opening a terminal and running the following (you may be able to just

click the local host link below instead of running the following):

``` cd airbyte ```

``` ./run-ab-platform.sh ```

- Open up a browser and go to http://localhost:8000. It can take a while for the Airbyte

service to start, so don't be surprised if it takes ~10 minutes.

- Username: airbyte

- Password: password

- Click `Set up a new source`

- When defining a source, select `Microsoft SQL Server (MSSQL)`

- Host: `stairway.usu.edu`

- Port: `1433`

- Database: `dw\_group3`

- Username: `5360\_student`

- Password: `Supersquids94!` (you'll need to click the dropdown for optional fields)

- Select `Scan Changes with User Defined Cursor`

- Click `Set up source`

- Airbyte will run a connection test on the source to make sure it is set up properly

- Create a schema in your group3\_project database named `SamsSubsSandwhich` and

ensure you have a data warehouse named `lastname\_wh`

- Once Airbyte has run the connection test successfully, you will pick a destination,

select `Pick a destination`.

- Find and click on `Snowflake`

- Host: `https://rbb67081.snowflakecomputing.com`

- Role: `TRAINING\_ROLE`

- Warehouse: `CABOT\_WH `

- Database: `group3\_project`

- Schema: `SamsSubsSandwhich`

- Username:

- Authorization Method: `Username and Password`

- Password:

- Click `Set up destination`

- Once the connection test passes, it will pull up the new connection window

- Change schedule type to `Manual`

- Under `Activate the streams you want to sync`, click the button next to each table.

- Click Set up connection

- Click `Sync now`

- Once it's done, go to Snowflake and verify that you see data in the landing database

3. Then, use dbt to create and populate your dimensional model. Your

documentation must include the steps you took to do this (all code, etc.). You

may want to create a markdown file (similar to the dbt exercise). Your

documentation should be detailed enough that the process can be easily

reproduced.

### Transform (dbt) ###

- Open VSCode

- File > Open > Select your project (lastname\_DW)

- On the top bar of the application, select Terminal > New Terminal

- This will open a terminal in the directory of your project within VSCode

- Right click on the models directory and create a new folder inside of it. (Be careful not

to create it inside of the example directory.)

- Call this new folder `samssubssandwhichs`

- Right click on the folder samssubssandwhichs and create a new file. Name this file

`\_src\_samssubssandwhichs.yml`

- Populate the following code within \_src\_samssubssandwhichs.yml

```

version: 2

sources:

- name: SAMSSUBSSANDWHICH

database: GROUP3\_PROJECT

schema: SAMSSUBSSANDWHICH

tables:

- name: employee

- name: customer

- name: product

- name: date

- name: store

- name: ingredient

- name: inventory

- name: orders

- name: order\_line

```

- If you need to make any changes to your Snowflake information in your dbt project you

can change it by going to your dbt profile.yml file. You may need to change the schema.

- On a mac, this is located under your user directory. You have to click Shift +

command + . in order to see hidden folders. The .dbt folder will appear and inside is

profiles.yml

- On Windows, it's just in the user directory under the .dbt folder and the profiles.yml

is inside.

- Once you have found the profiles.yml file you can open in a text editor, change the

database to GROUP3\_PROJECT or your target database

#### dim employee####

- Create a new file inside of the samssubssandwhichs directory called

`ss\_dim\_employee.sql`

- Populate the following code within ss\_dim\_employee.sql

```

{{ config(

materialized = 'table',

schema = 'dw\_samssubsandwhichs'

)

}}

SELECT

{{ dbt\_utils.generate\_surrogate\_key(['employee\_id', 'employee\_first\_name',

'employee\_last\_name']) }} as employee\_key,

employee\_pk,

employee\_id,

employee\_first\_name,

employee\_last\_name,

employee\_birthdate

FROM {{ source('SAMSSUBSSANDWHICH', 'employee') }}

```

- Now run the following

```

dbt run -m ss\_dim\_employee

```

#### dim customer####

- Create a new file inside of the samssubssandwhichs directory called

`ss\_dim\_customer.sql`

- Populate the following code within ss\_dim\_customer.sql

```

{{ config(

materialized = 'table',

schema = 'dw\_samssubsandwhichs'

)

}}

select

{{ dbt\_utils.generate\_surrogate\_key(['customer\_id', 'customer\_first\_name',

'customer\_last\_name']) }} as customer\_key,

customer\_pk,

customer\_phone,

customer\_birthdate,

customer\_last\_name,

customer\_first\_name

FROM {{ source('SAMSSUBSSANDWHICH', 'customer') }}

```

- Now run the following

```

dbt run -m ss\_dim\_customer

```

#### dim date####

- Create a new file inside of the samssubssandwhichs directory called `ss\_dim\_date.sql`

- Populate the following code within ss\_dim\_date.sql

```

{{ config(

materialized = 'table',

schema = 'dw\_samssubsandwhichs'

)

}}

with cte\_date as (

{{ dbt\_date.get\_date\_dimension("1990-01-01", "2050-12-31") }}

)

SELECT

date\_day as date\_key,

date\_day,

day\_of\_week,

month\_of\_year,

month\_name,

quarter\_of\_year,

year\_number

from cte\_date

```

- Now run the following

```

dbt run -m ss\_dim\_date

```

### dim ingredient ###

- Create a new file inside of the samssubssandwhichs directory called

`ss\_dim\_ingredient.sql`

- Populate the following code within ss\_dim\_ingredient.sql

```

{{ config(

materialized = 'table',

schema = 'dw\_samssubsandwhichs'

) }}

SELECT

CAST(Null as INT) as ingredient\_id,

CAST(Null as VARCHAR(255)) as ingredient\_name,

CAST(Null as DECIMAL) as ingredient\_height,

CAST(Null as DECIMAL) as ingredient\_width,

CAST(Null as DECIMAL) as ingredient\_length,

CAST(Null as DECIMAL) as ingredient\_weight,

CAST(Null as VARCHAR(255)) as supplier\_name

WHERE FALSE

```

- Now run the following

```

dbt run -m ss\_dim\_ingredient

```

### dim orderline ###

- Create a new file inside of the samssubssandwhichs directory called

`ss\_dim\_orderline.sql`

- Populate the following code within ss\_dim\_orderline.sql

```

{{ config(

materialized = 'table',

schema = 'dw\_samssubsandwhichs'

)

}}

select

{{ dbt\_utils.generate\_surrogate\_key(['order\_line\_id','order\_number']) }} as

order\_line\_key,

order\_line\_qty,

order\_line\_id,

order\_number,

p.product\_key,

order\_line\_price

FROM {{ source('SAMSSUBSSANDWHICH', 'order\_line') }} ol

INNER JOIN {{ref('ss\_dim\_product')}} p on p.product\_id = ol.product\_id

```

- Now run the following

```

dbt run -m ss\_dim\_orderline

```

### dim product ###

- Create a new file inside of the samssubssandwhichs directory called

`ss\_dim\_product.sql`

- Populate the following code within ss\_dim\_product.sql

```

{{ config(

materialized = 'table',

schema = 'dw\_samssubsandwhichs'

)

}}

select

{{ dbt\_utils.generate\_surrogate\_key(['product\_id', 'product\_name']) }} as product\_key,

product\_id,

product\_type,

product\_name,

product\_cost,

product\_calories

FROM {{ source('SAMSSUBSSANDWHICH', 'product') }}

```

- Now run the following

```

dbt run -m ss\_dim\_product

```

### dim store ###

- Create a new file inside of the samssubssandwhichs directory called

`ss\_dim\_store.sql`

- Populate the following code within ss\_dim\_store.sql

```

{{ config(

materialized = 'table',

schema = 'dw\_samssubsandwhichs'

)

}}

select

{{ dbt\_utils.generate\_surrogate\_key(['store\_id', 'store\_state']) }} as store\_key,

store\_id,

store\_pk,

store\_address,

store\_city,

store\_state,

store\_zip

FROM {{ source('SAMSSUBSSANDWHICH', 'store') }}

```

- Now run the following

```

dbt run -m ss\_dim\_store

```

### dim inventory ###

- Create a new file inside of the samssubssandwhichs directory called

`ss\_dim\_inventory.sql`

- Populate the following code within ss\_dim\_inventory.sql

```

{{ config(

materialized = 'table',

schema = 'dw\_samssubsandwhichs'

) }}

SELECT

CAST(Null as INT) as inventory\_id,

CAST(Null as DATE) as date\_key,

CAST(Null as INT) as store\_key,

CAST(Null as INT) as ingredient\_id,

CAST(Null as INT) as ingredient\_qty,

WHERE FALSE

```

- Now run the following

```

dbt run -m ss\_fact\_inventory

```

### dim order ###

- Create a new file inside of the samssubssandwhichs directory called

`ss\_dim\_order.sql`

- Populate the following code within ss\_dim\_order.sql

```

{{ config(

materialized = 'table',

schema = 'dw\_samssubsandwhichs'

) }}

SELECT

o.order\_number,

o.order\_method,

d.date\_key,

c.customer\_key,

s.stores\_key,

e.employee\_key,

o.order\_total\_price,

o.order\_points\_earned

FROM {{ source('SAMSSUBSSANDWHICH', 'orders') }} o

INNER JOIN {{ ref('ss\_dim\_customer') }} c ON o.customer\_pk = c.customer\_pk

INNER JOIN {{ ref('ss\_dim\_employee') }} e ON o.employee\_pk = e.employee\_pk

INNER JOIN {{ ref('ss\_dim\_store') }} s ON o.store\_pk = s.store\_pk

INNER JOIN {{ ref('ss\_dim\_date') }} d ON o.order\_date = d.date\_key

```

- Now run the following

```

dbt run -m ss\_fact\_order

```

- Now create the \_schema\_samssubssandwhichs.yml file with the following code

```

version: 2

models:

- name: ss\_dim\_employee

description: "Sams Employees Dimension"

- name: ss\_dim\_customer

description: "Sams Customer Dimension"

- name: ss\_dim\_date

description: "Sams Date Dimension"

- name: ss\_dim\_store

description: "Sams Store Dimension"

- name: ss\_dim\_ingredient

description: "Sams Ingredient Dimension"

- name: ss\_dim\_orderline

description: "Sams OrderLine Dimension"

- name: ss\_fact\_order

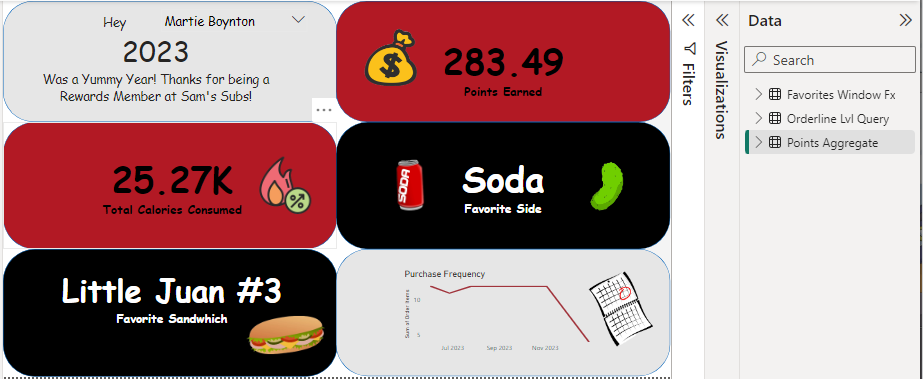
description: "Sams Order Fact"

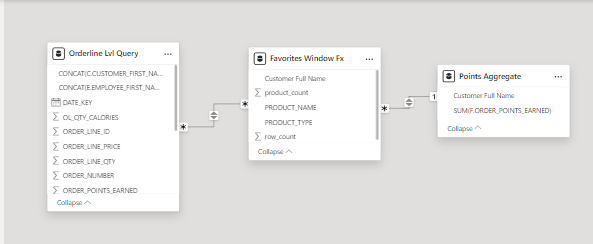
- name: ss\_fact\_inventory

description: "Sams Inventory Fact"

```

**Visualization**





Orderline Lvl Query:

SELECT

o.order\_number,

ol.order\_line\_id,

concat(c.customer\_first\_name, ' ', c.customer\_last\_name),

concat(e.employee\_first\_name, ' ', e.employee\_last\_name),

s.store\_address,

d.date\_key,

p.product\_id,

p.product\_name,

p.product\_calories,

p.product\_type,

p.product\_cost,

ol.order\_line\_qty,

ol.order\_line\_price,

p.product\_calories \* ol.order\_line\_qty as ol\_qty\_calories,

o.order\_total\_price,

o.order\_points\_earned

FROM

dbt\_test\_dw\_samssubsandwhichs.ss\_dim\_orderline as ol

LEFT JOIN

dbt\_test\_dw\_samssubsandwhichs.ss\_fact\_order as o on ol.order\_number = o.order\_number

LEFT JOIN

dbt\_test\_dw\_samssubsandwhichs.ss\_dim\_customer as c on o.customer\_key = c.customer\_key

LEFT JOIN

dbt\_test\_dw\_samssubsandwhichs.ss\_dim\_date as d on o.date\_key = d.date\_key

LEFT JOIN

dbt\_test\_dw\_samssubsandwhichs.ss\_dim\_product as p on ol.product\_key = p.product\_key

LEFT JOIN

dbt\_test\_dw\_samssubsandwhichs.ss\_dim\_employee as e on o.employee\_key = e.employee\_key

LEFT JOIN

dbt\_test\_dw\_samssubsandwhichs.ss\_dim\_store as s on o.store\_key = s.store\_key

Points Aggregate Query:

SELECT

c.customer\_first\_name || ' ' || c.customer\_last\_name as "Customer Full Name",

SUM(f.order\_points\_earned)

FROM dbt\_test\_dw\_samssubsandwhichs.ss\_fact\_order as f

LEFT JOIN dbt\_test\_dw\_samssubsandwhichs.ss\_dim\_customer as c on f.customer\_key = c.customer\_key

GROUP BY "Customer Full Name"

Favorites Window Function:  
select \*

from

(

Select customer\_first\_name || ' ' || customer\_last\_name as "Customer Full Name"

, count(sdol.product\_key) as "product\_count"

, sdp.product\_name

, product\_type

, row\_number() over (partition by sdc.customer\_key, product\_type order by count(sdol.product\_key) desc) as "row\_count"

from DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_orderline sdol

inner join DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_fact\_order sfo on sdol.order\_number = sfo.order\_number

inner join dbt\_test\_dw\_samssubsandwhichs.ss\_dim\_customer sdc on sfo.customer\_key = sdc.customer\_key

inner join DBT\_TEST\_DW\_SAMSSUBSANDWHICHS.ss\_dim\_product sdp on sdp.product\_key = sdol.product\_key

group by sdc.customer\_key,

sdc.customer\_first\_name, sdc.customer\_last\_name,

sdp.product\_name,

sdol.product\_key,

product\_type

) ptt

where "row\_count" = 1

In our Power BI visualization, we've structured the data processing as follows. First, we've implemented a customer-level slicer to enable filtering at the customer level, leveraging order line information and linking via a customer key to subsequent queries. The order line level query includes a slicer and computes the calorie consumption measure by multiplying the order line quantity with the product's calorie content. To determine the most purchased item, we've developed a custom Favorite Window Function. This function was separated from the main query due to limitations experienced with DAX in Power BI, ensuring accurate results. Additionally, the Points Aggregate function aggregates points earned from orders, resolving duplication issues encountered at the order line level due to DAX calculations. Lastly, the Calories Consumed Measure has been introduced, replacing the deprecated "inches eaten" feature from the sandwich table, and is now computed based on the product table's calorie content.