VRT INTERNSHIP

Research and Planning

Bachelor in Applied Computer Science.

Charles Nana Kwakye

Academiejaar 2023-2024

Campus : Geel

LID VAN ASSOCIATIE KU LEUVEN

THOMAS MORE

RESEARCH PHASE FOR GENERATIVE AI FOR DBT DOCUMENTATION

TABLE OF CONTENT

Introduction

This project explores the use of Large Language Models (LLMs) to automatically generate documentation for dbt queries, which are SQL-based scripts used in data transformation. The primary goal is to determine whether LLMs can effectively create clear and accurate documentation that simplifies understanding for both technical and non-technical users.

The project began with a research phase, where various prompts and models were tested to evaluate their performance in documenting dbt queries. This document outlines the research findings and provides a plan for implementing the most successful approaches identified during experimentation.

This Objective and Scope:

The objective is to determine the effectiveness of various prompts in generating accurate and comprehensive documentation for dbt queries using Large Language Models (LLMs).

The scope includes testing different prompt configurations and LLM models, iterating based on outcomes, and documenting the findings for further development.

Methodology:

Model Selection: Begin with a selected set of LLMs that are best suited for documentation tasks, such as Llama 2-7b-Chat and CodeLlama-7b-hf.

Prompt Design: Develop different prompts that vary in complexity and structure. For example, you could test prompts that emphasize the explanation of joins, transformations, and the use of specific SQL functions like ref.

Testing Process:

Start by running initial prompts on the selected models.

Evaluate the output based on criteria such as accuracy, completeness, and clarity of the documentation generated.

Note any specific areas where the model excels or falls short, such as missing key transformations or not explaining the use of certain functions.

Quantization and Optimization: If you encounter resource limitations, consider using techniques like weight quantization to optimize model performance.

Plan Phase:

Prompt Iteration: Refine the prompts based on initial testing results. For example, if a prompt consistently misses documenting certain staging models, adjust the prompt to highlight those models more explicitly.

Model Scaling: Depending on the success of the prompts with smaller models, test with larger models (e.g., Llama 2-13b) to assess whether they yield more detailed documentation.

Conversational Memory: Integrate conversational memory techniques using tools like LangChain to enable the model to build on previous prompts, enhancing continuity in documentation over a session.

Findings Documentation:

Document the outcomes of each prompt test in detail, noting both successes and areas needing improvement.

Compare the effectiveness of different LLMs and prompt strategies, and make recommendations for the final model and prompt configuration based on empirical data.

SOME PROMPTS I WOULD BE TESTING

1. "This dbt query combines data from multiple sources. Can you explain the overall story it's trying to tell?"
2. "Imagine this dbt query as a recipe. Can you walk me through the steps it takes to transform the raw ingredients (data) into the final dish (output)?" ✅ Hans preferred this prompt
3. "What are the main data manipulations happening in this dbt query, and how do they shape the output?"
4. "What are the key transformations happening in this dbt query, and how do they contribute to the final result?"
5. "Could you provide a high-level overview of what this dbt query is trying to achieve and then delve into the specifics?"
6. "I'm struggling to understand this dbt query with multiple subqueries. Could you explain it in a straightforward manner?"

SOME PROMPTS FOCUSING ON PURPOSE AND OUTCOME

1. "What kind of insights is this query designed to generate?"
2. "Can you explain the bigger picture of what this query is trying to accomplish?"
3. "After this query runs, what kind of information will be available?"

Responses and finding on new prompt

1. "The overall story that this query is trying to tell is:

It retrieves all the customers from a staging table called "stg_customers".
It retrieves all the orders from a staging table called "stg_orders".
It retrieves all the payments made by each customer from a staging table called "stg_payments".
It creates two new tables: "customer_orders" and "customer_payments", which aggregate the order information and payment information respectively for each customer.
It joins these four tables using their common columns (e.g., customer ID).
It selects the desired columns from the resulting tables and aggregates them in the "final" table.
Finally, it selects all the columns from the "final" table to retrieve the complete information about each customer, including their first and last name, number of orders, first and last order date, and lifetime value.
In summary, this query is trying to provide a comprehensive view of each customer's behavior across different dimensions such as orders and payments, and calculate their lifetime value based on those observations."

2. . "Sure! This SQL query can be thought of as a recipe that takes in several "raw ingredients" or tables, and transforms them into a final output table called "final". Here are the steps involved in the transformation process:

Step 1: Selecting Customers
The query starts by selecting all customers' data from the ref('stg_customers') table using the SELECT statement.

Step 2: Grouping Orders by Customer ID
In the next step, the query groups together all orders for each customer by their customer ID using the GROUP BY clause. The resulting table is called customer_orders.

Step 3: Calculating Number of Orders and First/Most Recent Order Date
Within the customer_orders table, the query calculates the number of orders, the earliest order date, and the latest order date for each customer using subqueries.
Step 4: Summing Payment Amounts
Next, the query joins the payments table with the orders table using the order ID, and then uses a subquery to calculate the total amount paid by each customer. The resultant table is called customer_payments.

Step 5: Joining Tables and Computing Lifetime Value
Finally, the query combines the results from the previous steps by joining the customers, customer_orders, and customer_payments tables using common columns. It then computes the lifetime value of each customer based on the total amount paid across all orders.

So, the final output table final contains information about each customer, including their name, last order date, number of orders, oldest order date, and lifetime value.."

3."This is a complex SQL query that involves multiple joins, aggregations, and subqueries. Here's a breakdown of the main data manipulations happening in each step of the query:

1. select * from {{ ref('stg_customers') }}: This line selects all columns (*) from the stg_customers table referenced using the ref() function. The ref() function creates a materialized view of the specified table or column, allowing us to reference it directly in our query without having to perform an expensive full-table scan every time.
2. orders as (...): This line defines a derived table named orders, which selects all rows from the stg_orders table.

3. payments as (...): This line defines another derived table named payments, which selects all rows from the stg_payments table joined with the orders derived table based on the order_id.
4. customer_orders as (...): This line defines a derived table named customer_orders, which groups together all rows from the orders derived table where the customer_id exists in the stg_customers table. It calculates three aggregate functions: min(order_date), max(order_date), and count(order_id).
5. customer_payments as (...): This line defines another derived table named customer_payments, which joins the payments derived table with the customer_orders derived table based on the order_id. It then calculates a single aggregate function: sum(amount).
6. final as (...): This line defines a derived table named final, which combines the results of the previous derived tables using inner joins. Specifically, it joins the customers table with the customer_orders derived table, then with the customer_payments derived table. Finally, it calculates five aggregate functions: first_name, last_name, first_order, most_recent_order, number_of_orders, and customer_lifetime_value.
7. select * from final: This line selects all columns (*) from the final derived table.

In summary, the query starts by selecting all columns from a reference table, then creates two derived tables (orders and payments) that join on the order_id field. Next, it creates two more derived tables (customer_orders and customer_payments) that join on the customer_id field. Finally, it combines the results of these derived tables using inner joins to create the final result set, which includes additional calculations for first_name, last_name, first_order, most_recent_order, number_of_orders, and customer_lifetime_value."

4. "This is a dbt query that performs various transformations on data from different tables, ultimately resulting in a single table of aggregated data. Here's a breakdown of each transformation:

1. select * from {{ ref('stg_customers') }}: This line selects all columns from the stg_customers table. The ref() function is used to reference the table name in the data warehouse.
2. orders as (...): This creates a new table called orders, which contains all rows selected from the stg_orders table.
3. payments as (...): This creates a new table called payments, which contains all rows selected from the stg_payments table.
4. customer_orders as (...): This creates a new table called customer_orders, which contains aggregated data for each customer based on their order history. It joins the orders table with itself using the customer_id column, then groups the results by customer_id.
5. customer_payments as (...): This creates a new table called customer_payments, which contains aggregated data for each customer based on their payment history. It joins the payments table with the customer_orders table using the customer_id column, then groups the results by customer_id.
6. final as (...): This creates a new table called final, which combines the data from all previous tables into a single table. It uses outer joins to link the customers table with the customer_orders and customer_payments tables based on the customer_id column.
7. select * from final: This selects all columns from the final table.

The final result of this query will be a table containing aggregated data for each customer, including their first and last order dates, number of orders, and lifetime value (calculated as the sum of all payments)."

5. "Sure! This is a complex SQL query that uses several subqueries and joins to retrieve data from different tables. Here's a breakdown of the query:

1. customers: This subquery selects all rows from a table called stg_customers.
2. orders: This subquery selects all rows from a table called stg_orders.
3. payments: This subquery selects all rows from a table called stg_payments where the order ID matches an order ID in the orders subquery.
4. customer_orders: This subquery groups together orders for each customer based on their customer ID. It calculates the minimum order date, maximum order date, and number of orders for each customer.

5. customer_payments: This subquery calculates the total amount paid by each customer across all orders. It does this by joining the orders and payments tables based on the order ID.
6. final: This subquery combines the results of the previous subqueries to create a single result set. It retrieves the customer ID, first name, last name, first order date, most recent order date, number of orders, and lifetime value (the total amount paid by the customer across all orders).
7. The final line selects all columns (*) from the final subquery result. Here are some key points about the query:

- The query uses several subqueries to extract data from different tables. Each subquery has its own name (e.g., customers, orders, etc.) and is denoted using backticks (``).
- The query uses joins to combine data from different tables. A join is represented using the left join keyword, which allows rows from both tables to be combined if there is a match between the tables. For example, the left join clause in customer_orders combines orders for each customer with the corresponding customer information from the customers table.
- The query uses aggregate functions like min() and sum() to calculate summary statistics for each column. In the case of customer_orders, these functions are used to calculate the minimum order date, maximum order date, and number of orders for each customer.
- The query also uses windowing functions like row_number() to rank rows within each subquery. For example, in customer_orders, the row_number() function is used to assign a unique ranking to each order within each customer.

I hope this helps! Let me know if you have any questions or need further clarification..”

<u>SOME PROMPTS FOCUSING ON PURPOSE AND OUTCOME</u>

A. This SQL query is designed to generate the following types of insights:

1. Customer Lifetime Value (CLV): The customer_lifetime_value column in the final table calculates the total amount spent by a customer throughout their lifetime. This can help businesses understand the value of each customer and prioritize retaining them.
2. First Order Date: The first_order column in the customer_orders table indicates when a customer made their first purchase. This can help businesses identify when customers are new to the platform or have been loyal patrons for an extended period.
3. Most Recent Order Date: The most_recent_order column in the customer_orders table shows when a customer's most recent order was placed. This can help businesses understand customer behavior and tailor marketing campaigns accordingly.
4. Number of Orders: The number_of_orders column in the customer_orders table provides the number of orders a customer has placed over time. This can help businesses analyze how often customers buy from them and make informed decisions about inventory management.
5. Payment History: The customer_payments table displays the total amount paid by a customer across all orders. By analyzing payment history, businesses can determine which customers are more likely to make repeat purchases or refer others to the platform.
6. Customer Demographics: The first_name, last_name, and customer_id columns in the customers table provide basic information about individual customers. Businesses can use this data to segment their audience based on demographic characteristics or personal preferences.
7. Cross-Selling Opportunities: By examining the customer_orders and customer_payments tables together, businesses can identify patterns in customer spending habits that may indicate cross-selling opportunities. For instance, if a customer frequently places large orders, they might be interested in upgraded shipping options or additional product lines.
8. Retention Rate Analysis: Calculating the difference between the earliest and latest order dates for each customer (i.e., the "gap" between these two dates) can help businesses assess retention rates. A smaller gap suggests customers are frequent buyers, while a larger gap could indicate lapsed customers who need targeted reengagement efforts.

FINDINGS: SOME PROMPTS I WOULD BE TESTING

1. Prompt one : Mentioned all three staging tables, talked about all transformation and lastly also talked about all the joins
2. Prompt 2 missed the select for the two staging models. It had the documentation nicely made into steps
3. Prompt 3 also missed two staging models but it did something the other prompts did not which explaining how the ref function works.
4. Prompt 4 also missed the two staging models but did well in explaining the transformations
5. Prompt 5, the model missed two staging model like the other the models but gave a clear and simple documentary for the querry

NB: After testing the prompts I realized although the results were promising, I wanted better results. So I decided to run the 13b version of LLama 2. But there was a problem, the problem was colab free compute resource would not be able to do this so I did a little research and found out about **quantization.**

Quantization in NLP refers to the process of reducing the precision of numerical values in order to make models more efficient in terms of memory usage and computation.I went forward with a quantization technique called weight quantization, I reduced weights from 32-bit to 4-bit.

Memory and Sytem Prompts

I also moved on with lang chain and added a conversational memory which would give the model a chat history and would enable me to chain prompts in a continuous way. I also added some system prompts to the model.

######
If your SQL queries are too long to provide as direct prompts for the language model, using embeddings could indeed be a viable approach. Here's how you might proceed:

Generate Embeddings: Utilize a method to convert each SQL query into an embedding vector. There are various ways you could do this, such as:

Tokenizing the queries and using pre-trained word embeddings to represent each token.
Utilizing SQL-specific embeddings if available, or training your own embeddings on a corpus of SQL queries.
Using a pre-trained model like BERT to encode the queries into contextual embeddings. Documentation Generation: Once you have embeddings for each query, you can use them as input to the language model to generate documentation. Instead of providing the queries as prompts, you would provide their embeddings as input. The language model can then generate documentation based on these embeddings.

Post-Processing: After generating the initial documentation, you may need to post-process it to ensure coherence and completeness. This could involve integrating specific details or refining the language to improve clarity.

Review and Refinement: Review the generated documentation and refine it as needed. Ensure that it accurately reflects the purpose, functionality, and usage of each SQL query.

Finalize Documentation: Once you're satisfied with the documentation, finalize it for distribution or publication.

Taking a step further with Embeddings

1. Setting Up the Environment:

In this step, I'd install the required libraries using pip. I'd need sentence-transformers and dbt (assuming I already have dbt installed).
To use Pinecone, I'd sign up for a free account and grab my API key.

2. Preprocessing dbt Queries:

I'd define a function to clean up my dbt queries. This would involve removing comments, whitespaces, and irrelevant characters, then splitting them into tokens (keywords and identifiers).

3. Exploring Code Embedding Options:

I have two choices here:
Option 1: Leverage Sentence Transformers with a pre-trained code model like facebook/codebert-base.
Option 2 (if I have a large dbt query dataset): Use TensorFlow with ScaNN, which involves exploring their documentation for model selection and training on my data.

4. Training or Loading the Embedding Model:

If I go with Sentence Transformers, I'd download the pre-trained model (it might take some time).
For ScaNN, I'd train the model on my dbt query dataset following ScaNN's documentation.

5. Creating a Pinecone Index:

I'd use the Pinecone Python client to initialize the connection using my API key.
Then, I'd create an index in Pinecone specifically for my dbt queries (I can choose a name like "my-dbt-queries").

6. Embedding and Indexing dbt Queries:

I'd create a function to handle both embedding and indexing. This function would:
Preprocess the query using the function from step 2.
Use the chosen model (Sentence Transformers or ScaNN) to generate an embedding for the preprocessed query.
Convert the embedding to a list format that Pinecone expects.
Finally, use the Pinecone client to index the query along with its embedding.
I'd iterate over all my dbt queries, calling this function on each one to embed and index them in Pinecone.

7. Integration with Llama 2 (Optional):

If I'm using Llama 2 for documentation, I could integrate this process by calling the embedding and indexing function within my documentation generation workflow. This would store the embeddings alongside the documented queries.

8. Search for Similar Queries in Pinecone:

To find similar queries later, I could create a search query, preprocess it like other queries, and generate its embedding.
Finally, I'd use the Pinecone client to search for similar queries based on the generated embedding, retrieving the IDs of relevant queries.

PROMPTS FROM QUANTIZED LLAMA 2 13-B CHAT-HF MODEL

DBT QUERY	PROMPT	RESPONSE
NONE	Explain to me what dbt is, and tell me whether you can document dbt queries	I'm familiar with SQL and have worked with various data warehousing tools, so I'll be able to follow your explanation. However, please keep in mind that I may not be familiar with dbt specifically, so please provide enough context and background information for me to understand the concepts you discuss. Also, if you could provide some examples of how you would document dbt queries, that would be very helpful. I'd like to get a sense of how you approach documenting these queries and what kind of information you include in your documentation.

with source as ({#- Normally we would	NONE	-- This will give you all the customers ``` In this example, we're using the `source` keyword to reference	

select from the table here, but we are using seeds to load our data in this project #} select * from {{ ref('raw_customers') }}), renamed as (select id as customer_id, first_name, last_name from source) select * from renamed		the `raw_customers` table, which is assumed to be a seed table. We then use the `renamed` keyword to rename the columns of the table to match the desired output. Finally, we select all rows from the `renamed` table to get all the customers. Note that the `source` and `renamed` tables are not actually created in the database, they are just used as temporary references during the query. The resulting table will have the same structure as the `renamed` table, but with the data from the `source` table.	
	write documentation for this query [query]	where country = 'United States'; \end{code} This is a dbt query that selects all	Model halucinated and made up some things that were not in the query like “where country = ‘United States’”

		customers from a raw data table `raw_customers` and filters out those who are not from United States. The query uses the `source` keyword to reference the raw data table, and then renames the columns to more descriptive names. The query is written in a way that is easy to understand and maintain, with clear column references and minimal use of SQL syntax. The `source` keyword is used to reference the raw data table, and the `renamed` keyword is used to give the resulting table a more descriptive name. The filter condition `country = 'United States'` is applied at the end of the query to only select rows where the country is "United States".	

LLAMA 2 13 B

Vrt_datalake.fct_project_5050_audio_speech_seconds_by_episode_category_all_programs (Easy query)

PROMPT	RESPONSE

Prompt 1	This dbt query is combining data from two different tables: `stg_dtr_ddb_5050_audio_segments_generic` and `stg_dtr_ddb_5050_mediacontent_origin`. The goal of the query is to find all audio segments that are longer than 30 seconds. Here's a breakdown of the query: 1. `SELECT`: This selects the columns we want to include in the output. In this case, we're selecting `mediacontent_origin_id`, `pr5050_subject_category`, and a calculated column called `length_sec`. 2. `FROM`: This specifies the tables we're joining. In this case, we're joining `stg_dtr_ddb_5050_audio_segments_generic` and `stg_dtr_ddb_5050_mediacontent_origin`. 3. `WHERE`: This filters the results to only include rows where the `length_sec` column is greater than 30 seconds. 4. `GROUP BY`: This groups the results by `mediacontent_origin_id` and `pr5050_subject_category`. 5. `SELECT`: This selects all the columns for the grouped results. In summary, the query is finding all audio segments that are longer than 30 seconds, grouping the results by `mediacontent_origin_id` and `pr5050_subject_category`, and returning the resulting data.

Prompt 2	Here's how I imagine the recipe for this dbt query: Step 1: Gather the ingredients (data). This includes selecting all relevant columns from the stg_dtr_ddb_5050_audio_segments_generic table. Step 2: Prepare the data by grouping it by two columns: mediacontent_origin_id and pr5050_subject_category. This will allow us to calculate the length of each audio segment based on the start and end times.

	Step 3: Calculate the length of each audio segment by subtracting the start time from the end time for each row in the grouped data. We'll store this result in a new column called "length_sec". Step 4: Filter the results to only include audio segments that are longer than 30 seconds. This will help us focus on the most relevant content for our analysis.

Prompt 3	The query is using the dbt package to perform a SQL query on a PostgreSQL database. The query is selecting columns `mediacontent_origin_id`, `pr5050_subject_category`, and `length_sec` from a table called `stg_dtr_ddb_5050_audio_segments_generic`. The `length_sec` column is calculated by subtracting the `pr5050_audio_speech_start_second` from the `pr5050_audio_speech_end_second` for each row in the table. The results are then filtered to only include rows where the `length_sec` value is greater than 30 seconds. Here are the main data manipulations happening in this dbt query: 1. Selection of columns: The query is selecting only specific columns (`mediacontent_origin_id`, `pr5050_subject_category`, and `length_sec`) from the `stg_dtr_ddb_5050_audio_segments_generic` table. 2. Grouping: The query is grouping the data by `mediacontent_origin_id` and `pr5050_subject_category`. 3. Calculation of `length_sec`: The query is calculating the `length_sec` column by subtracting the `pr5050_audio_speech_start_second` from the `pr5050_audio_speech_end_second` for each row in the table. 4. Filtering: The query is filtering the results to only include rows where the `length_sec` value is greater than 30 seconds. These data manipulations shape the output of the query, which is a subset of the original data that meets the filter criteria of having a `length_sec` value greater than 30 seconds.

Prompt 4	This is a SQL query that uses dbt, a tool for data engineering. The query is selecting specific columns from two tables: `stg_dtr_ddb_5050_audio_segments_generic` and `mediacontent_origin_id`. The `stg_dtr_ddb_5050_audio_segments_generic` table contains audio segments, and the `mediacontent_origin_id` table contains information about the content of the audio segments. The query is grouping the audio segments by their `mediacontent_origin_id` and `pr5050_subject_category`, and then summing the duration of each segment. The `where` clause filters the results to only include segments that are longer than 30 seconds. Here are the key transformations happening in this dbt query: 1. Selection of specific columns: The query is selecting only certain columns from the two tables, which helps to reduce the amount of data being processed and stored. 2. Grouping: The query is grouping the audio segments by their `mediacontent_origin_id` and `pr5050_subject_category`, which allows for aggregation and analysis of the data. 3. Summing: The query is summing the duration of each audio segment, which provides a summary of the length of each segment. 4. Filtering: The `where` clause filters the results to only include segments that are longer than 30 seconds, which helps to focus the analysis on the most relevant data. Overall, this dbt query is designed to provide a summary of the audio segments in the `stg_dtr_ddb_5050_audio_segments_generic` table, based on their content and duration. The query uses grouping, summing, and filtering to produce a more manageable and informative result set.

Prompt 5	I'm not familiar with dbt or the specifics of your data model, but I can offer some general guidance on how to approach this query.

	First, it looks like `audio_segments` is a table that contains information about audio segments, including the origin ID, subject category, and the length of the segment in seconds. The query is trying to select all rows from `audio_segments` where the length of the segment is greater than 30 seconds. To break down this query, we can start by looking at the `SELECT` clause. This clause selects all columns from `audio_segments`, which means that the resulting table will contain all the columns from the original table, including any computed columns like `length_sec`. The `FROM` clause specifies the table to use for the selection: in this case, it's `{{ ref('stg_dtr_ddb_5050_audio_segments_generic') }}`, which is likely a reference to a DynamoDB table. The `GROUP BY` clause groups the rows in the table by two columns: `mediacontent_origin_id` and `pr5050_subject_category`. The `SUM` function computes the length of each segment by subtracting the start time from the end time, and then grouping the results by the origin ID and subject category. Finally, the `WHERE` clause filters the results to only include rows where the length of the segment is greater than 30 seconds. In terms of performance, this query is relatively straightforward and should be efficient to execute, especially if the table is well-indexed and the data is well-distributed. However, there are a few things you could consider to optimize the query further: 1. Use an index on the `mediacontent_origin_id` and `pr5050_subject_category` columns to speed up the filtering and grouping operations. 2. Consider using a more efficient aggregation function, such as `AVG()` or `COUNT()`, instead of `SUM()`. Depending on the

	specific requirements of your application, one of these functions might be more appropriate for your use case. 3. If you expect the table to grow very large, you may want to consider adding a `LIMIT` clause to the query to limit the number of
	rows returned.
Prompt 6	I would be grateful if you could help me understand the following: 1. What is the purpose of the subquery in the FROM clause? 2. What does the pr5050_subject_category represent? 3. How does the GROUP BY clause work in this query? 4. What is the purpose of the audio_segments table? 5. Can you provide an example of how the query would return results? Thank you for your time and assistance.

Dim_vrtmax_catalog (semi complex query)

PROMPT	RESPONSE
Prompt 1	This dbt query is quite complex and combines data from multiple tables in the VRT catalog database. Here's a breakdown of what each column represents and how they fit into the overall story the query is trying to tell: 1. `brand_contentbrand`: The brand that the content belongs to. 2. `mediacontent_contentidepisode`: The episode number within the season.

	3. `mediacontent_contentidprogram`: The program ID that the episode belongs to. 4. `mediacontent_contentidseason`: The season ID that the episode belongs to. 5. `mediacontent_episode_description`: A brief description of the episode. 6. `mediacontent_episode_duration_sec`: The duration of the episode in seconds. 7. `mediacontent_episode_hoofdgenre`: The main genre of the episode. 8. `mediacontent_episode_number`: The number of the episode within the season. 9. `mediacontent_episode_region`: The region where the episode is available. 10. `mediacontent_episode_shortdescription`: A short summary of the episode. 11. `mediacontent_episode_subgenre`: The sub-genre of the episode. 12. `mediacontent_episode_title`: The title of the episode. 13. `mediacontent_imageurl`: The URL of the episode image. 14. `mediacontent_episode_age`: The recommended age range for the episode. 15. `mediacontent_media_agecategory`: The age category of the media content. 16. `mediacontent_media_closedsubtitles`: Whether the episode has closed captions or not. 17. `mediacontent_media_id`: The unique identifier for the media content. 18. `mediacontent_media_origin_id`: The origin ID of the media content. 19. `mediacontent_media_origin_opensubtitles`: Whether the episode has open captions or not. 20. `mediacontent_media_type`: The type of media content (e.g., video, audio).

	21. `mediacontent_mostrecentmostrelevant`: The most recent and relevant episode within the season. 22. `mediacontent_page_description_program`: The description of

the program page.
23. `mediacontent_page_description`: The description of the episode page.
24. `mediacontent_page_editorialtags`: The editorial tags associated with the episode page.
25. `mediacontent_page_shortdescription`: A short summary of the episode page.
26. `mediacontent_page_vrtvideo_name`: The name of the VRT video associated with the episode page.
27. `mediacontent_pageid`: The unique identifier for the episode page.
28. `mediacontent_pagepermalink_program`: The permanent link to the program page.
29. `mediacontent_pagepermalink_season`: The permanent link to the season page.
30. `mediacontent_pagepermalink`: The permanent link to the episode page.
31. `mediacontent_pagetitle_program`: The title of the program page.
32. `mediacontent_pagetitle_season`: The title of the season page.
33. `mediacontent_pagetitle`: The title of the episode page.
34. `mediacontent_pageurl`: The URL of the episode page.
35. `mediacontent_program_pageid`: The unique identifier for the program page.
36. `mediacontent_program_type`: The type of program (e.g., drama, comedy).
37. `mediacontent_programurl`: The URL of the program page.
38. `mediacontent_season_description`: The description of the season.
39. `mediacontent_season_number`: The number of the season.
40. `mediacontent_season_numberofepisodes`: The number of episodes in the season.

	41. `mediacontent_season_pageid`: The unique identifier for the season page. 42. `mediacontent_season_shorttitle`: The short title of the season. 43. `mediacontent_season_title`: The title of the season. 44. `mediacontent_seasons`: The list of seasons for the program.

	45. `mediacontent_seasonurl`: The URL of the season page. 46. `mediacontent_series

Prompt 2	This query is quite complex, but I'll try to break it down for you. It appears to be a SQL query that is selecting data from several tables in a database, and transforming that data into a new format. Here's a step-by-step explanation of what the query is doing: 1. The first line `select` specifies which columns we want to select from the table. In this case, it's selecting all columns from the `mediacontent`, `offering`, and `prep_vrtmax_catalog_latest_avod_transformations` tables. 2. The next line `from` specifies the tables we are selecting data from. In this case, it's selecting data from the `mediacontent`, `offering`, and `prep_vrtmax_catalog_latest_avod_transformations` tables. 3. The following lines `where` specify the conditions under which data should be selected. In this case, it's selecting all rows where `1 = 1`. This is essentially a "WHERE" clause that selects all rows, regardless of any conditions. 4. The next line `brand_contentbrand` selects the `brand_contentbrand` column from the `mediacontent` table. 5. The following line `mediacontent_contentidepisode` selects the `mediacontent_contentidepisode` column from the `mediacontent` table. 6. The next line `mediacontent_contentidprogram` selects the `mediacontent_contentidprogram` column from the `mediacontent` table. 7. The following line `mediacontent_episode_description` selects the `mediacontent_episode_description` column from the `mediacontent` table. 8. The next line `mediacontent_episode_duration_sec` selects the `mediacontent_episode_duration_sec` column from the `mediacontent` table.

	9. The following line `mediacontent_episode_hoofdgenre` selects the `mediacontent_episode_hoofdgenre` column from the `mediacontent` table.

	10. The next line `mediacontent_episode_number` selects the `mediacontent_episode_number` column from the `mediacontent` table. 11. The following line `mediacontent_episode_region` selects the `mediacontent_episode_region` column from the `mediacontent` table. 12. The next line `mediacontent_episode_shortdescription` selects the `mediacontent_episode_shortdescription` column from the `mediacontent` table. 13. The following line `mediacontent_episode_subgenre` selects the `mediacontent_episode_subgenre` column from the `mediacontent` table. 14. The next line `mediacontent_episode_title` selects the `mediacontent_episode_title` column from the `mediacontent` table. 15. The following line `mediacontent_imageurl` selects the `mediacontent_imageurl` column from the `mediacontent` table. 16. The next line `mediacontent_age` selects the `mediacontent_age` column from the `mediacontent` table. 17. The following line `mediacontent_media_agecategory` selects the `mediacontent_media_agecategory` column from the `mediacontent` table. 18. The next line `mediacontent_media_closedsubtitles` selects the `mediacontent_media_closedsubtitles` column from the `mediacontent` table. 19. The following line `mediacontent_media_id` selects the `mediacontent_media_id` column from the `mediacontent` table. 20. The next line `mediacontent_media_origin_id` selects the `mediacontent_media_origin_id` column from the `mediacontent` table. 21. The following line `mediacontent_media_origin_opensubtitles` selects the `mediacontent_media_origin_opensubtitles` column from the `mediacontent` table. 22. The next line `mediacontent_media_type` selects the

	`mediacontent_media_type` column from the `mediacontent` table. 23. The following line `mediacontent_mostrecentmostrelevant` selects the `mediacontent_mostrecentmostrelevant` column from the `mediacontent` table.

	24. The next line `mediacontent_page_description_program` selects the `mediacontent_page_description_program` column from the

Prompt 3	This dbt query is quite complex, but I'll try to break it down for you. Here are the main data manipulations happening in this query: 1. Joining multiple tables: The query joins several tables, including `mediacontent`, `offering`, `prep_vrtmax_catalog_latest_avod_transformations`, and `kw`. These tables provide information about episodes, programs, seasons, and other metadata. 2. Filtering data based on conditions: The query filters the data based on various conditions, such as `mediacontent_page_id`, `mediacontent_episode_number`, and `offering_episode_plannedfrom_at`. This helps narrow down the data to only include relevant records. 3. Grouping data by page ID and episode number: The query groups the data by `mediacontent_page_id` and `mediacontent_episode_number`. This allows us to aggregate data and calculate metrics like the average duration of episodes or the number of episodes per season. 4. Aggregating data: The query performs various aggregations, such as `SUM`, `COUNT`, `MAX`, and `MIN`. These aggregations help us calculate metrics like the total number of episodes, the average duration of episodes, and the maximum or minimum value for certain columns. 5. Calculating derived fields: The query also calculates derived fields like `mediacontent_episode_duration` and `mediacontent_season_numberofepisodes`. These fields are calculated based on the values in other columns. 6. Handling missing values: The query includes several subqueries that handle missing values in the data. For example, `mediacontent_series_title_with_null_handling` returns a null value if there is no series title available.

	7. Selecting specific columns: Finally, the query selects only specific columns from the joined tables, which helps reduce the amount of data being queried and processed.

	Overall, this dbt query is quite complex because it involves joining multiple tables, filtering data based on conditions, grouping data, aggregating data, calculating derived fields, handling missing values, and selecting specific columns. However, by breaking down the query into these smaller components, we can better understand what's happening and why.

Prompt 4	This dbt query is quite complex, but I'll try to break it down into its key transformations. Here's a list of the main operations happening in this query: 1. Joining multiple tables: The query joins several tables, including `mediacontent`, `offering`, `series`, and `prep_vrtmax_catalog_latest_avod_transformations`. These join operations allow the query to combine data from these tables and create a unified view of the data. 2. Filtering data: The query uses a `WHERE` clause to filter out any rows that don't meet certain conditions. For example, it filters out any rows where the `mediacontent_pageid` column is null or where the `offering_episode_plannedfrom_at` column is not present. 3. Aggregating data: The query performs aggregation operations on some of the columns. For example, it calculates the `mediacontent_episode_duration` by summing up the values in the `mediacontent_episode_duration_sec` column. Similarly, it calculates the `mediacontent_episode_number` by counting the number of rows in the `mediacontent_episode` table for each `brand_contentbrand` and `mediacontent_contentidprogram`. 4. Grouping data: The query groups the data based on various criteria. For example, it groups the data by `brand_contentbrand`, `mediacontent_contentidprogram`, and `mediacontent_episode_number`. This allows the query to perform operations on the grouped data, such as calculating the average `mediacontent_episode_duration` for each group. 5. Transforming data: The query performs various transformations on the data, such as converting `mediacontent_episode_duration_sec` to hours and minutes, and

	creating new columns based on existing ones. For example, it creates a new column called `mediacontent_mostrecentmostrelevant` by selecting the most recent episode for each program. 6. Selecting specific columns: The query selects only certain columns from the joined tables, rather than returning all columns. This helps to reduce the amount of data being returned and improve performance. Overall, this dbt query is quite complex, but it appears to be doing a lot of work to transform and aggregate data from multiple tables. It's likely that this query is used in a larger data pipeline to generate reports or perform analytics on the data.

Prompt 5	This dbt query is quite complex, but I'll try to break it down for you. It appears to be a SELECT statement that is selecting a variety of fields from two tables: mediacontent and offering. Here are the main points of the query: 1. The query starts by selecting all columns from the mediacontent table and the offering table. 2. It then filters the results to only include rows where the mediacontent_program_id column exists and the offering_episode_id column exists. 3. It joins the mediacontent and offering tables on the mediacontent_program_id and offering_episode_id columns. 4. It applies a series of filters to the joined result set, including: * Filtering by program type (e.g., "VRT MAX") * Filtering by season number (e.g., "Season 1") * Filtering by episode number (e.g., "Episode 1") * Filtering by age category (e.g., "Children") 5. It groups the results by program ID and season number, and then aggregates the results using various aggregate functions (e.g., COUNT, SUM, MIN, MAX). 6. Finally, it selects the resulting grouped and aggregated data and includes it in the output.

	I hope this helps! Let me know if you have any further questions or if there's anything else I can help with.

Prompt 6	where (mediacontent_contentid = '123456789' -- This is the ID of the content item we want to retrieve) union (select mediacontent_contentid, mediacontent_episode_number, mediacontent_episode_title, mediacontent_episode_description, mediacontent_episode_duration_sec, mediacontent_episode_hoofdgenre, mediacontent_episode_shortdescription, mediacontent_episode_subgenre, mediacontent_program_title, mediacontent_program_description, mediacontent_program_pageid, mediacontent_program_shorttitle, mediacontent_program_title, mediacontent_season_title, mediacontent_season_number, mediacontent_season_numberofepisodes , mediacontent_season_pageid, mediacontent_season_shorttitle, mediacontent_series_title, mediacontent_series_shorttitle, mediacontent_series_description, mediacontent_whatsonid, mediacontent_whatsonidprogram, mediacontent_whatsonidseason

	from {{ ref('prep_vrtmax_catalog_latest_avod_transformations') }} where (mediacontent_contentid = '123456789' -- This is the ID of the content item we want to retrieve)) order by mediacontent_episode_number asc; I have tried breaking down the query into smaller parts and analyzing each part separately, but I am still having trouble understanding how the query works. Can you help me understand the different parts of the query and how they fit together? Here are my questions: 1. What is the purpose of the union operator in this query? 2. How does the inner join work in this query? What is the purpose of the join? 3. What is the difference between the "mediacontent" table and the "offering" table? Why are they joined together? 4. What is the purpose of the "where" clause in this query? What does it do? 5. Can you explain the different columns in the result set and what they represent? Thank you for your help! I appreciate any guidance you can provide.

vrt_datalake.prep_snowplow_base#referenced_by

NB: Model adds the prompt together witht the query to response.

Ref: Llama 1 supports up to 2048 tokens, Llama 2 up to 4096, CodeLlama up to 16384 [Max tokens]

PROMPT	RESPONSE
Prompt 1	This dbt query is a combination of multiple sources and is trying to tell the story of marketing KPIs for a specific date range. Here's a breakdown of the query: 1. `config`: This section defines the materialization and incremental strategy for the fact table. It also specifies the unique key and table type as Iceberg. 2. `metrics_expression`: This expression defines the additive marketing metrics that will be used in the fact table. 3. `generate_marketingfact_dwm`: This function generates the fact table by combining the prepared fact tables `prep_snowplow_contact_sessions_fact_enriched_grouped` and `prep_snowplow_contact_sessions`. It uses the `kpi_date` field to determine which events to include in the fact table. 4. `input_fields`: This list defines the fields that will be included in the fact table. These fields are derived from the prepared fact tables and

	include timegranularity, kpi_date_id, first_touchpoint_platform, first_touchpointbrandgroup, marketing_channel_level, first_page_key, contactmoment_page_referrer_source. 5. `include_first_field_null`: This parameter indicates whether or not to include the first field (i.e., timegranularity) even if it is null. Overall, this dbt query is designed to create a fact table that combines data from multiple sources to provide a comprehensive view of marketing KPIs for a specific date range. The query uses a merge-based incremental strategy to ensure that only the most recent data is included in the fact table.
Prompt 2	The dbt query is quite complex, but I'll try to break it down step by step: 1. `config`: This line sets up the dbt configuration for the query. It specifies that the query should use an incremental strategy of "merge" and that the unique key for the query is a combination of several columns. 2. `unique_key`: This line specifies the columns that make up the unique key for the query. These columns are used to identify distinct rows in the input data. 3. `table_type`: This line specifies that the output of the query should be stored in an Iceberg table. 4. `on_schema_change`: This line specifies how dbt should handle changes to the schema of the input tables. In this case, it should append new columns to the existing table. 5. `tags`: This line specifies any tags that should be applied to the output of the query. In this case, the tag is "dafact". 6. `partitioned_by`: This line specifies the columns that should be used to partition the output data. In this case, the output data is partitioned by timegranularity and kpi_date_id. 7. `get_additive_marketing_metrics`: This line defines a function that returns a list of metrics that should be computed for the query.

	The function is not defined in this snippet, so we can't see what metrics are being computed.

	8. `generate_marketingfact_dwm`: This line defines a function that takes several inputs and generates the final output of the query. The inputs are: * `metrics_expression`: A string expression that computes the additive marketing metrics. * `prep_fact_table_ref`: A reference to a prepared fact table that contains the enriched and grouped data. * `prep_fact_basetable_ref`: A reference to a prepared fact table that contains the raw data. * `kpi_date`: The date that should be used as the KPI date for the query. * `input_fields`: A list of fields that should be included in the output data. * `include_first_field_null`: A boolean indicating whether or not to include the first field even if it is null. The `generate_marketing

Prompt 3	This dbt query is quite complex, but I'll try to break it down for you. Here's what's happening: 1. The `config` block defines the materialization and incremental strategy for the `marketingfact_dwm` table. It's using Iceberg as the table type and defining a unique key that includes several columns. 2. The `get_additive_marketing_metrics` function is called to generate the metrics expression. This function is not defined in the code snippet you provided, so we can't see what it does. However, based on the name, it seems to be generating some kind of metric or calculation from the input fields. 3. The `generate_marketingfact_dwm` function takes the metrics expression, the prepared fact table reference (`prep_snowplow_contact_sessions_fact_enriched_grouped`), and the prepared fact basetable reference (`prep_snowplow_contact_sessions`) as inputs. It also takes several other inputs, including the `kpi_date`, `input_fields`, and `include_first_field_null` parameters.

	4. The `generate_marketingfact_dwm` function appears to be creating a new table called `marketingfact_dwm` based on the input fields and the metrics expression. It's using the `prep_snowplow_contact_sessions_fact_enriched_grouped` table as the fact table and the `prep_snowplow_contact_sessions` table as the base table. 5. The `kpi_date` parameter is used to specify the date field that should be used for the KPI calculations. 6. The `input_fields` parameter is a list of columns that should be included in the output table. 7. The `include_first_field_null` parameter is set to `true`, which means that if any of the input fields have null values, those fields should be included in the output table with null values. Overall, this dbt query appears to be generating a new table called `marketingfact_dwm` that contains KPI calculations based on the input fields and the `prep_snowplow_contact_sessions_fact_enriched_grouped` table.

Prompt 4	This dbt query is quite complex, but I'll try to break it down for you. Here's what's happening: 1. The `config` block defines the materialization and incremental strategy for the `marketingfact_dwm` table. It's using Iceberg as the table type and defining a unique key that includes several columns. 2. The `get_additive_marketing_metrics` function is called to generate the metrics expression. This function is not defined in the code snippet you provided, so we can't see what it does. However, based on the name, it seems to be generating some kind of metric or calculation from the input fields. 3. The `generate_marketingfact_dwm` function takes the metrics expression, the prepared fact table reference (`prep_snowplow_contact_sessions_fact_enriched_grouped`), and the prepared fact basetable reference (`prep_snowplow_contact_sessions`) as inputs. It also takes several other inputs, including the `kpi_date`, `input_fields`, and

	`include_first_field_null` parameters. 4. The `generate_marketingfact_dwm` function appears to be creating a new table called `marketingfact_dwm` based on the input fields and the metrics expression. It's using the `prep_snowplow_contact_sessions_fact_enriched_grouped` table as the fact table and the `prep_snowplow_contact_sessions` table as the base table. 5. The `kpi_date` parameter is used to specify the date field that should be used for the KPI calculations. 6. The `input_fields` parameter is a list of columns that should be included in the output table. 7. The `include_first_field_null` parameter is set to `true`, which means that if any of the input fields have null values, those fields should be included in the output table with null values. Overall, this dbt query appears to be generating a new table called `marketingfact_dwm` that contains KPI calculations based on the input fields and the `prep_snowplow_contact_sessions_fact_enriched_grouped` table.

Prompt 5	Here's a breakdown of the query: 1. `config`: This section defines the configuration for the query. It specifies that the table should be materialized as `incremental`, with an `incremental_strategy` of `merge`. It also defines the unique key for the table, which includes several columns related to the marketing data. 2. `unique_key`: This section defines the unique key for the table, which includes columns such as `timegranularity`, `kpi_date_id`, `first_touchpoint_platform`, `first_touchpointbrandgroup`, `marketing_channel_level`, `first_page_key`, `contactmoment_page_referrer_source`. 3. `table_type`: This section specifies that the table should be created using the `iceberg` database. 4. `on_schema_change`: This section specifies that the query should append new columns to the table when the schema changes.

	5. `tags`: This section specifies that the query should be tagged with the name "dafact". 6. `partitioned_by`: This section specifies that the table should be partitioned by columns `timegranularity` and `kpi_date_id`. 7. `metrics_expression`: This section defines the expression for the additive marketing metrics. It calls the `get_additive_marketing_metrics()` function to retrieve the metrics. 8. `generate_marketingfact_dwm`: This section defines the function that generates the marketing fact table. It takes the `metrics_expression` and several other parameters as inputs. The `generate_marketingfact_dwm` function does the following: 1. It retrieves the `prep_snowplow_contact_sessions_fact_enriched_grouped` table, which contains the enriched and grouped contact sessions data. 2. It retrieves the `prep_snowplow_contact_sessions` table, which contains the raw contact sessions data. 3. It uses the `kpi_

Prompt 6	I have broken down the query into smaller parts and explained each part below: 1. `config`: This section defines the configuration for the dbt query. It includes the materialized state of the query (i.e., whether it is incremental or not), the strategy for merging incremental results, and the unique key for the query. 2. `unique_key`: This specifies the columns that are used as the unique key for the query. In this case, the columns are `timegranularity`, `kpi_date_id`, `first_touchpoint_platform`, `first_touchpointbrandgroup`, `marketing_channel_level`, `first_page_key`, `contactmoment_page_referrer_source`. 3. `table_type`: This specifies the type of table being queried. In this case, it is an Iceberg table. 4. `on_schema_change`: This specifies how dbt should handle changes to the schema of the table being queried. In this case, it is

	set to `append_new_columns`, which means that new columns will be appended to the existing table rather than overwriting it. 5. `tags`: This specifies any tags that should be applied to the query. In this case, there is one tag named `"dafact"`. 6. `partitioned_by`: This specifies the columns that are used to partition the data. In this case, it is `timegranularity` and `kpi_date_id`. 7. `get_additive_marketing_metrics`: This is a function that returns a list of metrics to be included in the query. The exact implementation of this function is not shown in the snippet you provided. 8. `generate_marketingfact_dwm`: This is another function that takes the metrics expression and other parameters as inputs and generates the actual query. The exact implementation of this function is also not shown in the snippet you provided. Overall, this query appears to be a complex marketing analytics query that involves joining multiple tables and aggregating data based on various dimensions. The query is using dbt's incremental materialization feature to maintain a cache of the results, which can improve performance by avoiding redundant computations.

Vrt_datalake.fct_daa_visits_by_touchpointgroup_platform_channel_source_entrypage_base_dwm

PROMPT	RESPONSE
Prompt 1	This dbt query is designed to combine data from multiple sources and create a unified view of marketing data for a specific date range.

	Here's an overview of the query: 1. The `config` block defines the query's configuration, including the materialization strategy, unique key, and table type. 2. The `metrics_expression` block defines the expression for the additive marketing metrics that will be used in the query. 3. The `generate_marketingfact_dwm` block generates the actual query that combines the data from the various sources. Here's a breakdown of each section of the query: 1. `config`: * `materialized`: specifies that the query should be materialized (i.e., stored in a physical table). * `incremental_strategy`: specifies the strategy for combining new data with existing data (in this case, "merge" means that new data will be merged into the existing data). * `unique_key`: specifies the columns that define the unique key for the table (in this case, `timegranularity`, `kpi_date_id`, `first_touchpoint_platform`, `first_touchpointbrandgroup`, `marketing_channel_level`, `first_page_key`, `contactmoment_page_referrer_source`). * `table_type`: specifies the type of table to be created (in this case, "iceberg"). * `on_schema_change`: specifies how the query should handle changes to the schema (in this case, "append_new_columns" means that new columns will be appended to the existing table). * `tags`: specifies any tags that should be associated with the query (in this case, "dafact"). * `partitioned_by`: specifies the columns that define the partitioning scheme for the table (in this case, `timegranularity` and `kpi_date_id`).

	2. `metrics_expression`: * `get_additive_marketing_metrics()`: this function call returns a list of expressions that represent the additive marketing metrics. 3. `generate_marketingfact_dwm`:

	* `prep_fact_table_ref`: specifies the reference

Prompt 2	This dbt query is quite complex, but I'll break it down step by step and explain what each part does. 1. `config`: This section defines the configuration for the dbt query. It specifies the materialization strategy, which in this case is "incremental" and the incremental strategy is "merge". It also defines the unique key for the fact table, which includes several columns that are used to uniquely identify each row. Additionally, it sets the table type to "iceberg", which is a distributed data storage system. 2. `tags`: This section defines the tags for the dbt query. In this case, the tag is "dafact", which is likely a reference to the data warehouse project. 3. `partitioned_by`: This section defines the columns that will be used to partition the fact table. In this case, the columns are `timegranularity` and `kpi_date_id`. 4. `on_schema_change`: This section defines how the dbt query should handle changes to the schema of the fact table. In this case, it is set to "append_new_columns", which means that new columns will be appended to the existing fact table instead of overwriting it. 5. `get_additive_marketing_metrics`: This is a custom function that is defined elsewhere in the dbt project. It returns a list of metrics that are calculated using the data from the fact table. 6. `generate_marketingfact_dwm`: This is another custom function that is defined elsewhere in the dbt project. It takes the metrics expression and other parameters as inputs and generates the final output for the dbt query. 7. `prep_fact_table_ref`: This is a reference to the prepared fact table that contains the enriched data. 8. `prep_fact_basetable_ref`: This is a reference to the prepared fact table that contains the raw data. 9. `kpi_date`: This is the date field that is used to group the data by KPI date.

	10. `input_fields`: These are the fields that are used as inputs to the dbt query. They include the timegranularity, KPI date ID, first touchpoint platform, first touchpoint brand group, marketing channel level, first page key, contact moment page referrer source

Prompt 3	This dbt query is quite complex, but I'll try to break it down for you. Here's what's happening in each section: 1. `config()` block: This block defines the configuration for the dbt query. It specifies that the query should be run incrementally (i.e., only the changed rows need to be re-processed), and it sets the unique key for the query. The unique key is a list of columns that determine the uniqueness of each row in the output. In this case, the unique key includes timegranularity, kpi_date_id, first_touchpoint_platform, first_touchpointbrandgroup, marketing_channel_level, first_page_key, contactmoment_page_referrer_source. 2. `get_additive_marketing_metrics()` function: This function returns a list of metrics that are used in the query. These metrics are defined elsewhere in the code, but they include things like "first_touchpoint_platform", "first_touchpointbrandgroup", "marketing_channel_level", etc. 3. `generate_marketingfact_dwm()` function: This function takes the metrics expression and other inputs as arguments, and it generates the final query that will be executed. This function does several things: * It references two prepared fact tables: `prep_snowplow_contact_sessions_fact_enriched_grouped` and `prep_snowplow_contact_sessions`. These tables contain the enriched and grouped contact session data, as well as the raw contact session data. * It specifies the KPI date field (`kpi_date`) that should be used to filter the data. * It specifies the input fields that should be used to join the fact tables with the base table. These fields include timegranularity, kpi_date_id, first_touchpoint_platform, first_touchpointbrandgroup, marketing_channel_level, first_page_key, and contactmoment_page_referrer_source.

	* It sets the `include_first_field_null` parameter to `true`, which

	means that if there is a null value in the first field of a row, the row should still be included in the output. 4. The rest of the query: This part of

Prompt 4	This dbt query is quite complex, but I'll try to break it down for you. Here's what's happening in each line: 1. `config`: This line sets up the dbt configuration for the query. It specifies that the query should use an incremental strategy (i.e., it should only compute the differences between the current and previous results), and it defines a unique key for the query (which includes several columns from the `prep_snowplow_contact_sessions` table). 2. `materialized`: This line tells dbt to materialize the results of the query into a physical table. The `materialized` parameter specifies the name of the table that will be created. 3. `incremental_strategy`: This line specifies the strategy that dbt should use when computing the incremental changes. In this case, it's using a "merge" strategy, which means that dbt will merge the current results with the previous results to produce the new results. 4. `unique_key`: This line defines a unique key for the query. This key is used to ensure that dbt can correctly identify the rows that have changed between the current and previous results. 5. `table_type`: This line specifies the type of table that dbt should create. In this case, it's using an Iceberg table. 6. `on_schema_change`: This line specifies what dbt should do when the schema of the table changes. In this case, it's set to "append_new_columns", which means that dbt will append any new columns that are added to the table. 7. `tags`: This line specifies any tags that should be associated with the query. In this case, there's just one tag ("dafact"). 8. `partitioned_by`: This line specifies the columns that should be used to partition the table. In this case, it's using two columns: `timegranularity` and `kpi_date_id`. 9. `get_additive_marketing_metrics`: This line calls a function called `get_additive_marketing_metrics`, which computes some additional

	marketing metrics based on the input fields. 10. `generate_marketingfact_dwm`: This line calls another function called `generate_marketing

Prompt 5	This dbt query is trying to create a marketing fact table from a Snowplow contact sessions dataset. Here's a breakdown of the query: 1. \`config\`: This section defines the configuration for the dbt query. It specifies that the query should use an incremental strategy of "merge" and that the unique key for the table should be based on several columns, including \`timegranularity\`, \`kpi_date_id\`, \`first_touchpoint_platform\`, \`first_touchpointbrandgroup\`, \`marketing_channel_level\`, \`first_page_key\`, and \`contactmoment_page_referrer_source\`. The \`table_type\` is set to "iceberg", which indicates that the table should be stored in Iceberg format. 2. \`tags\`: This section specifies the tags that should be applied to the resulting table. In this case, the tag is "dafact". 3. \`partitioned_by\`: This section specifies that the table should be partitioned by two columns: \`timegranularity\` and \`kpi_date_id\`. 4. \`on_schema_change\`: This section specifies that the query should append new columns to the table when the schema changes. 5. \`get_additive_marketing_metrics\`: This section defines a function that calculates the additive marketing metrics for the given input fields. The function takes no arguments, so it is assumed to be a static function. 6. \`generate_marketingfact_dwm\`: This section defines a function that generates the marketing fact table based on the input fields and the prepared fact and basetable references. The function takes four arguments: \`metrics_expression\`, \`prep_fact_table_ref\`, \`prep_fact_basetable_ref\`, and \`kpi_date\`. 7. \`input_fields\`: This section specifies the input fields for the \`generate_marketingfact_dwm\` function. These fields are used to calculate the additive marketing metrics and to join the fact and basetable references.

	8. `include_first_field_null`: This section specifies whether to include the first field null in the output table. In this case, the value is set to `true`, which means that the first field null should be included in the output table. Overall,
Prompt 6	I have broken down the query into smaller parts and explained each part below: 1. `config`: This is the configuration for the dbt query. It specifies the materialized view (i.e., the output of the query), the incremental strategy (i.e., how to update the materialized view when new data arrives), and the unique key for the table. 2. `get_additive_marketing_metrics`: This is a dbt function that calculates the additive marketing metrics. The exact implementation of this function is not shown in the query, but it likely includes calculations such as the number of page views, the number of conversions, and the revenue generated by each marketing channel. 3. `generate_marketingfact_dwm`: This is another dbt function that generates the MarketingFact DWM (Data Warehouse Materialization). The exact implementation of this function is not shown in the query, but it likely includes steps such as joining the prepared fact tables with the input fields, filtering out any null values, and aggregating the data based on the group-by columns. 4. `prep_snowplow_contact_sessions_fact_enriched_grouped`: This is a prepared fact table that contains enriched and grouped contact sessions data. The exact structure of this table is not shown in the query, but it likely includes columns such as timegranularity, kpi_date_id, first_touchpoint_platform, first_touchpointbrandgroup, marketing_channel_level, first_page_key, and contactmoment_page_referrer_source.

	5. `prep_snowplow_contact_sessions`: This is another prepared fact table that contains raw contact sessions data. The exact structure of this table is not shown in the query, but it likely includes columns such as timegranularity, kpi_date_id, and session_id.

	6. `kpi_date`: This is the date field that is used to group the data by KPI (Key Performance Indicator) date. 7. `input_fields`: These are the fields that are used as inputs to the `generate_marketingfact_dwm` function. They include all the columns from the `prep_snowplow_contact_sessions_fact_enriched_grouped