Sr. Analytics Engineer Interview Tasks

Task 1 - Sessionization and Incrementality

Task 1 a - Add session id to events table.

Please use the following link to access the query: https://github.com/agvelazquez/analytics_engineer_task/blob/main/src/sessions.sql

The query has four parts.

1. Land example as a cte
2. Calculate as an additional column the timestamp of the previous event for each user id.
3. Filter out sessions by using the 30 min. threshold. The session id is the first event id of the session.
4. Calculate the timestamp of the next session.
5. Join back the sessions logic with the events table using the user id, the session start and session finish timestamps.

Task 1 - Sessionization and Incrementality

Follow up questions - How would you add incrementality to this model? What challenges might you encounter and how would you address them?

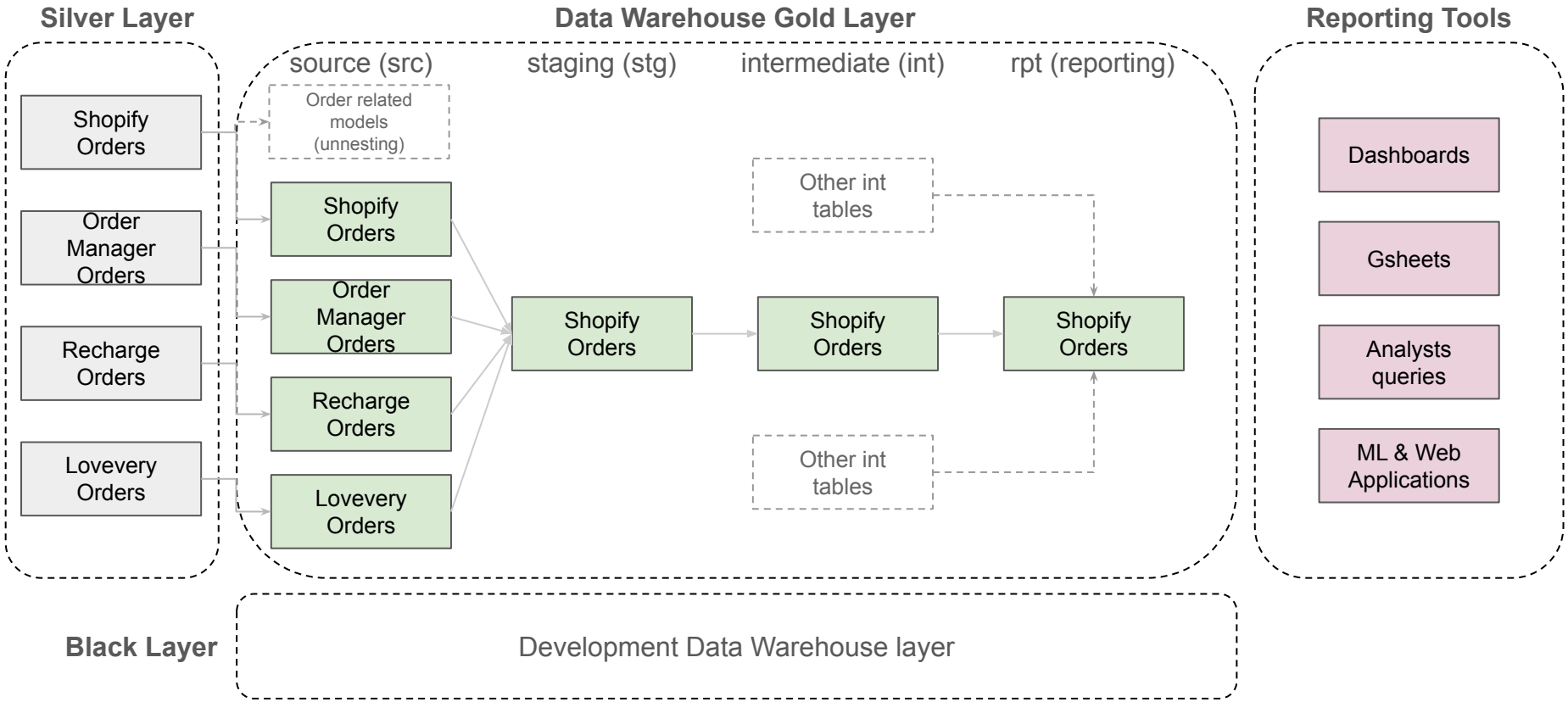
Most common issues working with digital events, sessionization and incrementality:

- **Late arriving data.** Events that arrive after initial processing. It may be worked out by setting up a current date and current date -1 partition refresh. Add data observability to validate session activity is within normal activity.
- **Data Volume and processing costs.** Setting up an incremental model with an insert - overwrite incremental model that don’t need to scan a key can reduce costs.
- **Sessions spanning multiple days (session continuity).** Ensure events are labeled correctly for those sessions that span in more than one partition and can seamlessly continue sessions across partition boundaries.
- **Open Sessions at partition refresh.** Sessions that are open at the moment of the refresh that then can be closed on subsequent refresh changing the analytics/aggregations for the session.

Task 2

Databricks to dbt migration.

New Data Warehouse Architecture



Architecture Diagram References & Explanation

Silver. The Data Warehouse layer for landing raw data from ETL tools (e.g., Fivetran, Airbyte) or custom pipelines (e.g., Dagster, Airflow). The orders table should be partitioned by a timestamp column (preferably the order completion timestamp) and include an `updated_at` column. This Silver layer resides in a different Dataset/Schema from the dbt layer.

Gold. The layer where dbt production tables are located, handling all intermediate transformations.

- **source (src_):** Incremental model using `updated_at` column and partitioning. This includes transformations like formatting, standardization, cleaning, and unnesting. It also contains `record_source` and `region` columns. Should have same number of rows as the raw data. The unnesting process may be used as source models for other processes.
- **staging (stg_):** Incremental models serving as a single source of truth by integrating all sources. This involves filtering, applying business rules, and further data standardization.
- **intermediate (int_):** Partitioned and clustered models enriched with additional columns from other source and staging tables. This layer includes complex business logic transformations.
- **reporting (rpt_):** Production-grade tables connected to reporting tools, ML models, and data applications, and utilized by the business. These are wide tables that combine information from multiple sources, such as sessions, identity IDs, customers, shipments, payments, etc.

Black. The Data Warehouse layer used for development and managed by dbt. Each developer has a dedicated Dataset/Schema.

Project Roadmap Example

Example of Project Roadmap delivered to PMs and leaders to asses timeline and resources.

Task	Day 1	Day 2	Day 3	Day 4	Day 5	Day 6
Kick-off. Environment. set-up ccess & Permissions, Q&A, etc.						
Map all Databricks transformations.						
Validate initial POC						
Create source layer in dbt						
Validate and test source layer.						
Create staging layer in dbt						
Validate and test staging layer.						...
Update intermediate layer and any other donwnstream model.						...
Overall validation and testing.						...
Aumomate and create dbt testing.						...
Code deployment.						...
Ongoing validation.						...
Buffer time for final updates.						...
...						

All project modifications should be done on the first two or three layers of the Data Warehouse. The output of the intermediate level should remain the same to ensure smooth transition and business continuity.

What are some ways you would ensure this transformation logic migration produces data identical to the existing logic?

There are three actions I would recommend to ensure a successful migration.

1. **Totals validations**. Perform an initial validation using key metrics such as revenue, quantity, discounts, shipping, etc, to ensure both scripts produce the same results. This validation should span from high-level aggregations down to the order line level.
   You can find an example in the following link:
   https://github.com/agvelazquez/analytics_engineer_task/blob/main/src/total_validation.sql

2. **Column wise validation.** Conduct a column-to-column comparison using the outputs from both methods. This approach ensures accuracy across various data types, including strings, categorical columns, IDs, and numerical values.
   You can find an example in the following link:
   https://github.com/agvelazquez/analytics_engineer_task/blob/main/src/column_validation.sql

3. **Cross validation dbt tests.** Once the new dbt pipelines are deployed, I would recommend that both transformation methods coexists during the initial weeks ensuring both methods produce identical results while maintaining the ability to revert to the old methodology if unexpected issues arise.
   I would recommend two libraries for testing: dbt_utils and dbt_expectations.

Other Q&A

- **What would the benefits be of running this all in dbt?**
  - **Increased speed to production.** SQL is a versatile and widely used across all data roles, it simplifies the generation of reusable code and testing
  - **Easy to debug**. Consolidating everything into the same language, architecture and repository handle with dbt enhances the analyst experience at debugging, implementing test and answering ad-hoc questions.
  - **Simplify future migrations.** Using SQL as the sole language in a unified repository makes it easier to migrate to different architectures in the future.
- **What are the cons of running this in dbt?**
  - Databricks and JVM (Java Virtual Machine) may offer better running times for some large models like sessions.
  - Python is more versatile to solve complex transformations or calculations.
- **Do you believe this structure is still the best way to architect our project in this scenario? Would you go about it differently?**
  A priori, I don't think it is necessary to separate the orders by region knowing they are going to be aggregated later on. We can simplify the architecture by consolidating the sources at early stage and adding identifiers. I would recommend to add two columns from the source to detect from where the record is coming from. Example:
  - column 1: record_source → Values (Shopify, Recharge, Order Manager, Lovevey)
  - column 2: region → Values (US, EU, AUS)