**1. Seasonality**

* **Issue**: "Detecting seasonality is tough because the data has incomplete timestamps or outliers that distort periodic trends."
* **Impact**: "Queries to group sales by month or quarter return inconsistent results, making it hard to plan warehouse stock."
* **Example**: "A random bulk order in June skewed the summer trend for electronics."

**2. Interpurchase Interval**

* **Issue**: "Calculating interpurchase intervals is difficult since the data lacks unique customer IDs, or transactions aren’t tied to individuals."
* **Impact**: "I can’t accurately measure time gaps between purchases, which limits insights into restocking frequency.”
* **Example**: "Two purchases of detergent a week apart could be one customer or two—I can’t tell."

**3. Affinity of Size 2, 3, and Above**

* **Issue**: "Finding itemset affinities (e.g., pairs or triplets) in BigQuery is slow and resource-heavy due to the large number of combinations, and many results are irrelevant."
* **Impact**: "Queries take too long or exceed BigQuery’s limits, and the output includes impractical pairs like 'pens and milk.'"
* **Example**: "Joining the transaction table with itself to find size-2 affinities generated millions of rows."

**4. Data Quality Issues**

* **Issue**: "The warehouse data has missing values (e.g., null product IDs), inconsistent naming, or duplicates that affect all analyses."
* **Impact**: "This leads to unreliable seasonality trends, incorrect intervals, and fragmented affinities."
* **Example**: "A product listed as 'SKU123' and '123SKU' splits its sales data."

**5. SQL/BigQuery Limitations**

* **Issue**: "Writing complex SQL queries for these analyses is challenging, and I’m hitting performance or cost issues in BigQuery."
* **Impact**: "Long-running queries increase costs, and I struggle to optimize them for efficiency."
* **Example**: "A query for size-3 affinities timed out after scanning terabytes of data."

**How to Propose Solutions (Using BigQuery and SQL)**

Offer practical SQL-based solutions that leverage BigQuery’s strengths (e.g., scalability, window functions) while addressing these issues.

**1. Solution for Seasonality**

* **Approach**: "Clean timestamps and use aggregation with smoothing to identify seasonal patterns."
* **How**: "Filter out outliers and group sales by time periods (e.g., month) using DATE\_TRUNC. Use a moving average to smooth noise."
* **SQL Example**:

SELECT

DATE\_TRUNC(transaction\_date, MONTH) AS month,

SUM(sales\_amount) AS total\_sales,

AVG(SUM(sales\_amount)) OVER (ORDER BY DATE\_TRUNC(transaction\_date, MONTH) ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS smoothed\_sales

FROM `project.dataset.transactions`

WHERE transaction\_date IS NOT NULL

GROUP BY month

ORDER BY month;

* **Outcome**: "This highlights trends like December peaks, filtering out random spikes for better stock planning."

**2. Solution for Interpurchase Interval**

* **Approach**: "Approximate intervals using transaction timestamps and proxies like store ID if customer IDs are missing."
* **How**: "Use LAG to calculate time differences between consecutive purchases of the same product or category."
* **SQL Example**:

WITH PurchaseGaps AS (

SELECT

product\_id,

transaction\_date,

LAG(transaction\_date) OVER (PARTITION BY product\_id ORDER BY transaction\_date) AS prev\_purchase\_date

FROM `project.dataset.transactions`

WHERE transaction\_date IS NOT NULL

)

SELECT

product\_id,

AVG(DATE\_DIFF(transaction\_date, prev\_purchase\_date, DAY)) AS avg\_interpurchase\_days

FROM PurchaseGaps

WHERE prev\_purchase\_date IS NOT NULL

GROUP BY product\_id;

* **Outcome**: "Even without customer IDs, I can estimate that detergent is repurchased every 14 days on average."

**3. Solution for Affinity of Size 2, 3, and Above**

* **Approach**: "Optimize affinity queries with self-joins and filtering to focus on frequent, meaningful itemsets."
* **How**: "Use JOIN to find pairs (size 2) and extend to triplets (size 3), adding a minimum transaction threshold to reduce noise."
* **SQL Example (Size 2)**:

WITH Pairs AS (

SELECT

t1.product\_id AS product\_a,

t2.product\_id AS product\_b,

COUNT(DISTINCT t1.transaction\_id) AS co\_occurrence

FROM `project.dataset.transactions` t1

JOIN `project.dataset.transactions` t2

ON t1.transaction\_id = t2.transaction\_id

AND t1.product\_id < t2.product\_id *-- Avoid duplicates (e.g., A-B vs B-A)*

GROUP BY t1.product\_id, t2.product\_id

HAVING co\_occurrence >= 50 *-- Minimum support threshold*

)

SELECT \* FROM Pairs ORDER BY co\_occurrence DESC;

* **For Size 3**: Add a third join and adjust the threshold.
* **Outcome**: "This efficiently finds pairs like 'diapers and wipes' with high co-occurrence, actionable for warehouse bundling."

**4. Solution for Data Quality Issues**

* **Approach**: "Preprocess data in BigQuery to standardize and clean it."
* **How**: "Create a cleaned table with deduplicated rows and normalized product names."
* **SQL Example**:

CREATE TABLE `project.dataset.transactions\_cleaned` AS

SELECT DISTINCT

transaction\_id,

CASE

WHEN product\_id LIKE '%SKU%' THEN REGEXP\_REPLACE(product\_id, 'SKU', '')

ELSE product\_id

END AS product\_id\_cleaned,

COALESCE(transaction\_date, '2025-01-01') AS transaction\_date

FROM `project.dataset.transactions`;

* **Outcome**: "A unified dataset ensures consistent results across all analyses."

**5. Solution for SQL/BigQuery Limitations**

* **Approach**: "Optimize queries for cost and performance using BigQuery features."
* **How**: "Partition tables by date, use clustering on product\_id, and limit scanned data with WHERE clauses. Test queries on small samples first."
* **SQL Tip**: Add LIMIT or sample with TABLESAMPLE SYSTEM (10 PERCENT) during development.
* **Outcome**: "Queries run faster and cheaper, letting me analyze all three aspects efficiently."

**Final project explaination:**

"In my BigQuery project on Reward and Recommendation for Retail Data, I faced issues analyzing seasonality, interpurchase intervals, and affinities of size 2, 3, and above. For seasonality, incomplete timestamps and outliers like June bulk sales hid trends. For interpurchase intervals, missing customer IDs prevented tracking repurchase gaps. For affinities, self-joins for itemsets were slow and produced irrelevant results like 'onion and milk.' Data quality issues (e.g., duplicate SKUs) and unoptimized SQL further complicated things, driving up costs.

To fix this, I cleaned the data in BigQuery, standardizing product IDs and filling null dates. For seasonality, I used DATE\_TRUNC and moving averages to reveal monthly patterns. For interpurchase intervals, I approximated gaps with LAG over product timestamps, estimating 14-day cycles. For affinities, I optimized self-joins with thresholds (e.g., 50 co-occurrences) to find pairs like 'Bread and Butter’ or ‘Milk and Tea Powder’. I also partitioned tables and sampled data to cut costs. These SQL solutions delivered reliable, actionable insights for warehouse management."

**Context**: FreshMart operates 20 stores across a region, selling groceries, household goods, and seasonal items. They collect transactional data via a POS (Point of Sale) system, which feeds into a BigQuery project for analytics to optimize inventory, promotions, and warehouse management. The dataset includes sales transactions, customer loyalty records, and product metadata, but it’s messy due to inconsistent data entry, system upgrades, and human error.

**Sample Data Overview**

* **Table: Transactions**
  + Columns: transaction\_id, store\_id, timestamp, customer\_id (loyalty card, often null), sku (product ID), quantity, unit\_price, total\_amount
  + Size: ~10M rows/month
  + Issues: Timestamps sometimes missing or in inconsistent formats (e.g., "2023-12-15" vs. "12/15/23 14:32"), duplicate transactions from cashier retries, bulk sales skewing trends (e.g., a store manager bulk-buying 500 turkeys for Thanksgiving).
* **Table: Products**
  + Columns: sku, product\_name, category, supplier\_id, price
  + Issues: Duplicate SKUs (e.g., "SKU123" for both "Organic Milk 1L" and "Milk 1L" due to a supplier mix-up), missing category tags.
* **Table: Customers**
  + Columns: customer\_id, join\_date, store\_id\_preferred
  + Issues: ~60% of transactions lack customer\_id because most shoppers don’t use loyalty cards.

**Challenges and Examples**

1. **Seasonality Analysis**
   * **Problem**: FreshMart wants to detect monthly sales patterns to stock seasonal items like eggnog or sunscreen. However, incomplete timestamps (e.g., 5% of rows have null or "1970-01-01" defaults) and outliers (e.g., a June 2024 bulk order of 1,000 watermelons for a local festival) obscure trends.
   * **Real-Life Data Example**:
     + June 2024 sales spike to 10x normal volume due to the watermelon order, masking a subtler uptick in ice cream sales from warm weather.
     + December 2023 data shows eggnog sales doubling, but missing timestamps from a system outage on Dec 20-22 hide the peak.
   * **Solution**: Clean data by filtering out bulk orders (e.g., quantity > 100) and imputing null timestamps with transaction batch dates. Use DATE\_TRUNC(timestamp, MONTH) and a 3-month moving average (AVG(total\_amount) OVER (ORDER BY DATE\_TRUNC(timestamp, MONTH) ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)) to reveal a clear December eggnog surge and June ice cream trend.
2. **Interpurchase Intervals**
   * **Problem**: FreshMart wants to measure how often customers repurchase staples like milk or bread to optimize restocking. Missing customer\_id in 60% of transactions and inconsistent SKUs (e.g., "Bread White" vs. "White Bread Loaf") make it hard to track individual repurchase gaps.
   * **Real-Life Data Example**:
     + A loyal customer (ID: "CUST123") buys "SKU456" (milk) on Jan 5, Jan 19, and Feb 2, suggesting a ~14-day cycle. But most milk purchases lack customer\_id, and "SKU456" sometimes appears as "SKU457" due to a barcode error.
     + Bulk buyers (e.g., a café buying 50 loaves of bread weekly) inflate apparent repurchase rates.
   * **Solution**: Approximate gaps using LAG(timestamp) OVER (PARTITION BY sku ORDER BY timestamp) across all transactions for a given product, filtering out bulk purchases (quantity > 10). This estimates a 7-day milk repurchase cycle and 10-day bread cycle at the store level, despite missing customer IDs.
3. **Affinities (Market Basket Analysis)**
   * **Problem**: FreshMart wants to identify product pairs or triplets (e.g., "Milk and Cereal") for cross-promotions, but self-joins on the Transactions table are slow and yield noisy results like "Toilet Paper and Bananas" due to weak co-occurrence thresholds. Duplicate SKUs and low-support itemsets waste compute resources.
   * **Real-Life Data Example**:
     + A self-join finds "SKU123 (Milk)" and "SKU789 (Cereal)" co-occur in 200 transactions/month across stores, but "SKU123" also pairs with "SKU999 (Dog Food)" in 150 transactions—likely noise.
     + A triplet like "Milk, Cereal, Bananas" appears in only 10 transactions, below a meaningful threshold.
   * **Solution**: Standardize SKUs in the Products table (e.g., merge "SKU123" and "SKU457" into one "Milk" entry). Optimize SQL with a co-occurrence threshold (e.g., COUNT(\*) > 50) and limit to high-frequency categories (e.g., category IN ('Dairy', 'Bakery')). Results show "Milk and Cereal" (support: 200), "Bread and Butter" (support: 180), and "Pasta and Sauce" (support: 120) as actionable pairs. Partition Transactions by store\_id and sample 10% of data to reduce BigQuery costs.

**Data Quality and Cost Issues**

* **Duplicates**: A cashier double-scanning items creates ~2% duplicate transaction\_id rows, inflating sales figures.
* **Unoptimized SQL**: Early queries scanned the full 10M-row table without partitioning, costing $50/day in BigQuery fees.
* **Solution**: Deduplicate transactions using ROW\_NUMBER() OVER (PARTITION BY transaction\_id, sku ORDER BY timestamp) and keep only the first entry. Partition Transactions by DATE(timestamp) and cluster by store\_id to cut query costs to ~$10/day.

**Actionable Insights for Warehouse Management**

* **Seasonality**: Stock 20% more eggnog in December and 15% more ice cream in June across all stores.
* **Interpurchase Intervals**: Restock milk every 5-7 days and bread every 7-10 days per store, adjusting for local demand.
* **Affinities**: Bundle "Milk and Cereal" in endcap displays and place "Bread and Butter" near each other in aisles.

**How This Reflects Real Life**

This FreshMart example mirrors real retail data challenges: inconsistent POS systems, incomplete customer tracking, and the need to balance insight with compute costs. It’s grounded in plausible data quirks (e.g., bulk orders, duplicate SKUs) and SQL techniques (e.g., DATE\_TRUNC, LAG, partitioning) that align with your BigQuery approach, while offering a vivid, store-level narrative.

**What is co\_occurrence?**

* **Definition**: co\_occurrence is the count of distinct transactions where a specific set of items (e.g., product pairs or triplets) appears together.
* **Purpose**: It quantifies the strength of association between items, helping you identify patterns like "diapers and wipes are often bought together."
* **Relevance to Your Project**: For affinities of size 2, 3, and above, co\_occurrence is the foundation for filtering frequent itemsets (e.g., with HAVING co\_occurrence >= 50).

**How to Calculate co\_occurrence in BigQuery SQL**

You calculate co\_occurrence by:

1. Joining the transaction table with itself to find items in the same transaction.
2. Grouping by the itemset (e.g., pairs or triplets).
3. Counting the distinct transactions where those items co-occur.

Let’s break it down with examples assuming your table (project.dataset.transactions) has columns transaction\_id (unique per purchase) and product\_id (unique per item).

**1. For Size 2 (Pairs)**

To calculate co\_occurrence for product pairs:

* Self-join the table on transaction\_id to find all pairs of products in each transaction.
* Use COUNT(DISTINCT transaction\_id) to count unique transactions where both products appear.

**SQL Example**:

sql

CollapseWrapCopy

WITH Pairs AS (

SELECT

t1.product\_id AS product\_a,

t2.product\_id AS product\_b,

COUNT(DISTINCT t1.transaction\_id) AS co\_occurrence

FROM `project.dataset.transactions` t1

JOIN `project.dataset.transactions` t2

ON t1.transaction\_id = t2.transaction\_id

AND t1.product\_id < t2.product\_id *-- Avoid duplicates (e.g., A-B vs B-A)*

GROUP BY t1.product\_id, t2.product\_id

)

SELECT

product\_a,

product\_b,

co\_occurrence

FROM Pairs

WHERE co\_occurrence >= 50 *-- Optional: filter as per your earlier question*

ORDER BY co\_occurrence DESC;

* **Step-by-Step**:
  1. **Join**: t1 and t2 match rows within the same transaction\_id. If transaction T1 has products [A, B, C], the join creates pairs [A-B], [A-C], [B-C].
  2. **Condition**: t1.product\_id < t2.product\_id ensures each pair is unique (e.g., only "A-B", not "B-A").
  3. **Group**: GROUP BY product\_a, product\_b aggregates all transactions for each pair.
  4. **Count**: COUNT(DISTINCT t1.transaction\_id) counts unique transactions where the pair appears, giving co\_occurrence.
* **Example Data**:

| **transaction\_id** | **product\_id** |
| --- | --- |
| T1 | A |
| T1 | B |
| T2 | A |
| T2 | B |
| T3 | A |
| T3 | C |

* 1. Pair A-B appears in T1 and T2 → co\_occurrence = 2.
  2. Pair A-C appears in T3 → co\_occurrence = 1.

**2. For Size 3 (Triplets)**

To calculate co\_occurrence for triplets:

* Join the table three times to find three products in the same transaction.
* Count distinct transactions as before.

**SQL Example**:

sql

CollapseWrapCopy

WITH Triplets AS (

SELECT

t1.product\_id AS product\_a,

t2.product\_id AS product\_b,

t3.product\_id AS product\_c,

COUNT(DISTINCT t1.transaction\_id) AS co\_occurrence

FROM `project.dataset.transactions` t1

JOIN `project.dataset.transactions` t2

ON t1.transaction\_id = t2.transaction\_id

AND t1.product\_id < t2.product\_id

JOIN `project.dataset.transactions` t3

ON t1.transaction\_id = t3.transaction\_id

AND t2.product\_id < t3.product\_id *-- Ensure unique ordering (A < B < C)*

GROUP BY t1.product\_id, t2.product\_id, t3.product\_id

)

SELECT

product\_a,

product\_b,

product\_c,

co\_occurrence

FROM Triplets

WHERE co\_occurrence >= 50 *-- Optional filter*

ORDER BY co\_occurrence DESC;

* **Step-by-Step**:
  1. **Join**: Three-way join finds triplets like [A, B, C] in the same transaction.
  2. **Condition**: t1.product\_id < t2.product\_id < t3.product\_id ensures uniqueness (e.g., only "A-B-C", not "B-A-C").
  3. **Group**: Aggregate by the triplet.
  4. **Count**: COUNT(DISTINCT t1.transaction\_id) gives co\_occurrence.
* **Example Data**:

| **transaction\_id** | **product\_id** |
| --- | --- |
| T1 | A |
| T1 | B |
| T1 | C |
| T2 | A |
| T2 | B |
| T2 | C |

* 1. Triplet A-B-C appears in T1 and T2 → co\_occurrence = 2.

**Why COUNT(DISTINCT)?**

* **Avoid Duplicates**: If a transaction has multiple rows for the same product (e.g., quantity > 1), COUNT(\*) might overcount. COUNT(DISTINCT transaction\_id) ensures each transaction is counted once per itemset.
* **BigQuery Efficiency**: It’s optimized for large-scale data, which suits your warehouse dataset.

**Connecting to HAVING co\_occurrence >= 50**

* After calculating co\_occurrence, the HAVING clause filters out itemsets with fewer than 50 occurrences:

sql

CollapseWrapCopy

HAVING co\_occurrence >= 50

* This step comes after GROUP BY and before the final SELECT, ensuring only frequent itemsets remain.

**Optimization Tips for BigQuery**

* **Partitioning**: If your table is partitioned by transaction\_date, the joins will scan less data.
* **Clustering**: Cluster by product\_id to speed up grouping.
* **Sampling**: Test on a subset first:

sql

CollapseWrapCopy

FROM `project.dataset.transactions` TABLESAMPLE SYSTEM (10 PERCENT)

**Sample Output**

For size 2:

| **product\_a** | **product\_b** | **co\_occurrence** |
| --- | --- | --- |
| Diapers | Wipes | 120 |
| Milk | Cereal | 75 |

Filtered with HAVING co\_occurrence >= 50, only pairs like these remain.