Part1:

SQL Queries:

1.Total transactions per product category:

```
SELECT product_category, COUNT(*) AS total_transactions
FROM transactions
GROUP BY product_category;
```

2.Top 5 accounts by total transaction value:

```
SELECT customer_id, SUM(transaction_amount)
FROM transactions
GROUP BY customer_id
ORDER BY SUM(transaction_amount) DESC
LIMIT 5;
```

3.Monthly spend trends over the past year:

```
SELECT DATE_FORMAT(transaction_date, '%Y-%m') AS month, SUM(transaction_amount)
FROM transactions
WHERE transaction_date >= DATE_SUB(NOW(), INTERVAL 1 YEAR)
GROUP BY month
ORDER BY month;
```

Part2:

Question 1: How do you scale this solution to 10x or 100x, taking into account different potential latencies needed?

Scaling Considerations:

To scale the solution to handle 10x or 100x more data, we need to address both performance and reliability. Key factors would include:

Database Optimization:

Indexing: Proper indexing on columns like transaction_date, customer_id, and transaction_amount will help speed up queries.

Partitioning: For large datasets, partitioning the database by transaction_date (e.g., yearly or monthly partitions) could help. This will allow the system to only query relevant data without scanning the entire database.

Sharding: Distributing the database across multiple servers based on customer ID or geographic location could distribute the load and reduce latencies.

Caching: Caching frequently accessed data or aggregation results using solutions like Redis can reduce the need for repeated computations.

Parallelization:

API Requests: Use concurrent requests when fetching large amounts of data from the API to reduce the wait time. This could involve increasing the number of threads in ThreadPoolExecutor to fetch data in parallel.

Batch Processing: Instead of loading data in small chunks, you can use larger batch sizes for inserting data into the database. Be mindful of transaction limits, but larger batches will reduce the overhead.

Microservices:

A microservices architecture could be used to isolate different parts of the process (e.g., data fetching, transformation, and loading) and run them independently across multiple instances. You could scale each service individually depending on its load.

Asynchronous Processing:

Use asynchronous techniques for non-blocking operations. For example, using Celery for background task processing to handle long-running ETL tasks can help scale the solution without blocking the main workflow.

Monitoring & Logging:

To ensure the solution can scale efficiently, it's essential to monitor the performance of both the database and the API, track latency, and be able to handle failures or retries automatically.

Handling Latency:

API Latency: Make sure to handle the network latency from the API calls gracefully. You can use retries with exponential backoff to prevent constant failure under high latency conditions.

Database Latency: With heavy data, database writes might introduce latency. Batch writing and async operations can mitigate some of the delays.

Question 2: How to handle changes within the source data, assuming some transactions would be potentially modified or backdated.

Approach to Handle Data Changes:

1. **Data Update/Correction**:
   - **Tracking Changes**: Add a last modified or status field to track the state of each transaction (e.g., pending, approved, declined, reversed). This will help identify if a transaction has changed after the initial insertion.
   - **Update Mechanism**: For transactions that are declined or reversed, you can periodically run a process that checks for transaction status changes and updates them in the database.
2. **Handling Backdated Transactions**:
   - **Time-Based Sorting**: If a transaction arrives late but should have been posted at a prior date, the system should support backdating by updating the transaction_date field accordingly.
   - **Historical Integrity**: Instead of updating records, it may be better to insert a new record with the correct date and transaction amount. This helps in keeping a history of all changes and ensures that the database reflects the true transactional history.
3. **Transactional Integrity**:
   - For any transaction that has been reversed, insert a corresponding reverse entry with a negative amount. This ensures that the original transaction is not lost, but is marked as canceled or reversed in subsequent reporting.
   - Use transaction_id as a unique identifier for transactions to track and update records efficiently.
4. **Database Constraints**:

    - **Unique Keys**: Ensure transaction_id is unique, and the database enforces this. Additionally, use foreign keys to ensure referential integrity between tables, especially for customer data.

5. **Audit Logs**:
    - Maintain an audit log that records all changes made to a transaction (e.g., status change, transaction reversal) along with a timestamp and user or process responsible for the change.

Question 3: Assuming the company is getting this data for the first time and has never had it before in a usable format for analytics or automations - what would be the single most important thing to build from it or take from it to deliver value?

Most Important Data Insight for Value Delivery:

The most important thing to build for immediate value is a **Customer Lifetime Value (CLV) model**.

- **Why CLV?**
    - CLV is a critical metric for any business because it helps estimate how much a customer is worth over the long term, allowing businesses to focus on high-value customers, optimize marketing spend, and improve retention strategies.
    - By analyzing the total spend per customer, frequency of transactions, and transaction history, you can predict future behavior and provide tailored offerings to customers, improving profitability and loyalty.

Building the CLV Model:

1. **Customer Segmentation**:
    - Use transaction amounts and frequency to categorize customers (e.g., high spenders, frequent customers). This segmentation will provide actionable insights into customer behavior and help in designing targeted promotions or loyalty programs.
2. **Revenue Forecasting**:
    - Based on the CLV and transaction history, predict the future revenue from a customer, which can then be used for operational planning (e.g., marketing budget, customer support).
3. **Automation**:
    - Automate marketing campaigns based on CLV predictions to ensure that the right customers are targeted with the right offers at the right time.
4. **Optimization of Customer Support**:
    - The CLV model can help prioritize high-value customers, ensuring they receive the most attention from customer support teams, potentially improving satisfaction and retention.

Deliverables for the Theoretical Questions:

1. **Scaling Solution**: As explained above, implementing database optimizations (indexes, partitioning), using parallel processing (threads for API fetching and batch inserts), and leveraging microservices will help scale the solution.
2. **Handling Data Changes**: The strategy of tracking transaction status, using a last_modified field, inserting reverse transactions for declined or backdated transactions, and using an audit log will ensure that the system can handle data changes.
3. **Most Important Insight (CLV)**: By building a CLV model, the company can derive valuable insights into customer behavior, optimize marketing spend, and improve customer retention.

Write-Up: Approach and Optimizations(Bonus section)

Approach:

The ETL pipeline was designed to fetch transactional data from an API, process and clean the data using Python, and load it into a MySQL database. The data was validated and transformed to handle missing values, incorrect formats, and duplicates. Additional categorization was implemented for transaction amounts and product categories to enable insightful analysis.

Optimizations and Trade-offs:

Parallel data insertion using Python's `concurrent.futures.ThreadPoolExecutor` was employed to speed up the database loading process. This trade-off increased complexity but significantly reduced the execution time for larger datasets. Error handling and logging mechanisms were incorporated to identify issues during data fetching and transformation.

To ensure compatibility, a fallback option was added to process sample data locally in case of API access issues. One limitation is that the current pipeline relies on a specific API key and assumes a MySQL database setup, which might require extra effort for a reviewer to replicate.