

Z511 DB Design Final Project Report

Bike Store Management System



Pranay Reddy Gundala
Geethika Elaprolu
Yatharth Kapadia

Company Name: PGK BikeStore Corp

Scenario and Database Requirements:

Nature of Business: PGK BikeStore Corp operates a thriving chain of bicycle retail outlets across multiple cities. The company specializes in selling a wide range of bicycles, including road bikes, mountain bikes, hybrids, and children's bikes, along with cycling accessories and gear. As a prominent player in the bicycle retail sector, PGK BikeStore Corp caters to diverse customer needs, from casual weekend riders to competitive cyclists. As the company expands, the need for a comprehensive, integrated database system has become increasingly evident. The implementation of this system is poised to revolutionize the way PGK manages its operations by harnessing data-driven strategies to enhance inventory management, customer relations, and overall business efficiency.

Purpose of the Database:

The primary goal of the PGK BikeStore Corp database is to streamline and optimize all facets of the company's operations. This encompasses enhancing transaction processing speeds, maintaining accurate and accessible inventory records, facilitating swift order processing, and ensuring effective customer relationship management. The database is designed to be the foundational system that supports the store's logistics, sales, and customer service teams by providing them with real-time data access.

The intended users of the PGK BikeStore database encompass various roles within the company, each with distinct responsibilities and specific needs from the system. Each type of user interaction with the database is tailored to ensure efficiency, security, and accessibility tailored to their roles, enhancing the overall operational workflow and customer service experience.

Store managers are pivotal in the operational hierarchy of PGK BikeStore Corp. They require comprehensive access to the database to effectively oversee daily store operations. Their primary needs include monitoring inventory levels, analyzing sales data, and managing staff performance. The database provides them with tools to generate reports that help in making informed decisions about stock replenishment, promotional strategies, and personnel management. This access helps managers maintain optimal operational flow within the stores and ensures that inventory is aligned with customer demand patterns. Sales associates are on the front lines of customer interaction and are crucial in driving the retail sales of the stores. The database system

is designed to provide them with real-time access to product information, availability, and pricing. This capability is essential for providing excellent customer service, as it allows associates to answer customer queries efficiently and process transactions quickly. Furthermore, the system facilitates the handling of customer returns and exchanges, ensuring that these transactions are reflected immediately in the inventory and sales records. Inventory managers focus on the logistical aspects of the store operations, particularly managing stock levels across multiple locations. They rely on the database for detailed insights into inventory status, which includes tracking stock movements, anticipating stock-outs, and coordinating with suppliers for timely deliveries. The system enables them to plan and execute inventory strategies that minimize overstock and understock situations, ensuring that the stores operate efficiently without unnecessary capital tied up in excess inventory. Corporate executives of PGK BikeStore Corp utilize the database for strategic oversight and long-term planning. Their interaction with the database is typically at a higher level, focusing on aggregated data and trends rather than individual transactions. They require access to comprehensive reports and dashboards that provide insights into sales performance, revenue generation, and market trends. This information is crucial for making strategic decisions such as expanding into new markets, introducing new product lines, and formulating marketing strategies.

PGK BikeStore Corp's database is envisioned as a robust system capable of handling the detailed elements of a multi-faceted retail operation. The database will capture and manage several key entities, each critical to the business's smooth functioning:

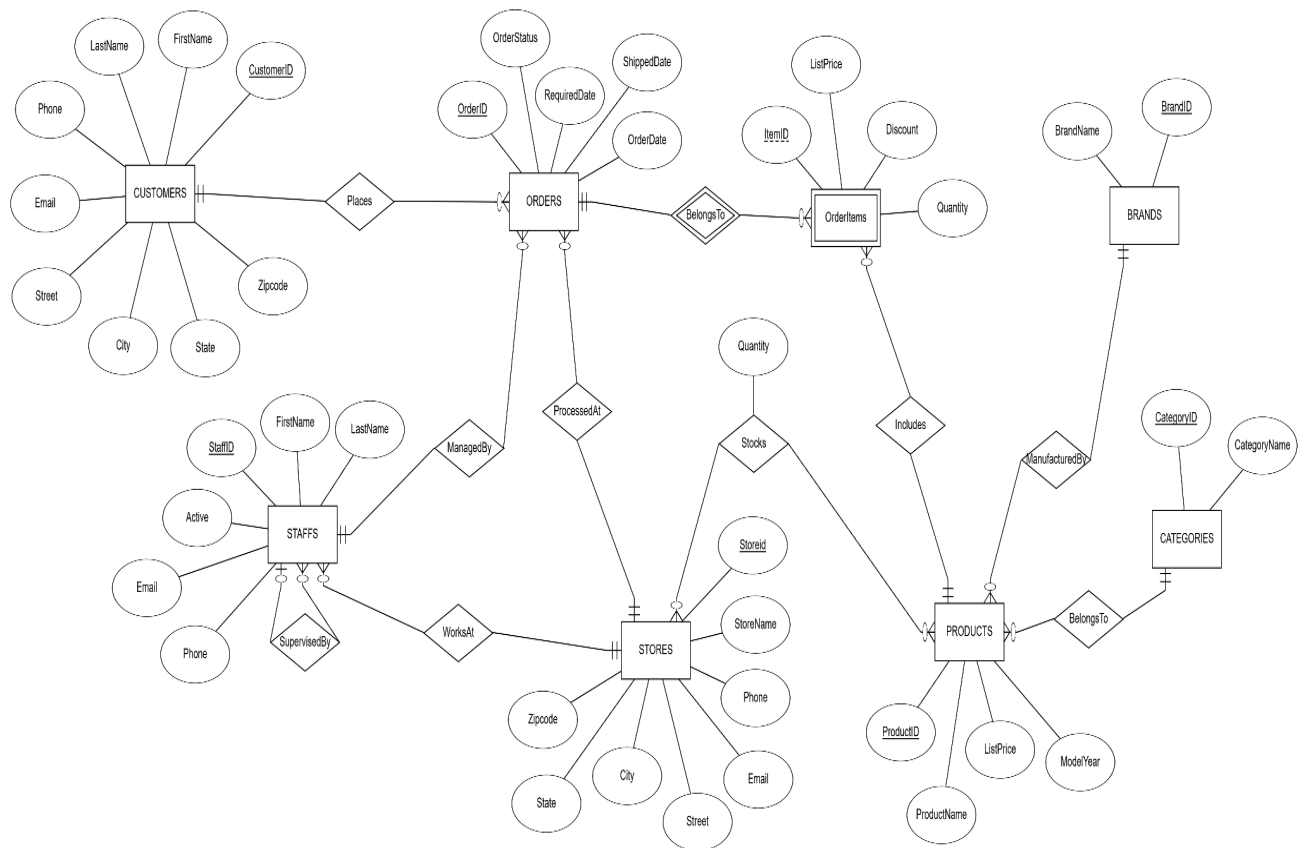
Database Requirements:

1. The database will keep track of products, orders, and customer details, as well as information regarding the stores and staff involved in the transaction processes.
2. For each customer, we will track the unique customer ID, customer name (composed of the customer's first and last name), their contact information including phone and email, and address details such as street, city, state, and zip code.
3. Each store in the database has a unique store ID, store name, and location details including street, city, state, and zip code. Each store can employ multiple staff members.
4. Each product in the database has a unique product ID, product name, model year, and list price. Products are linked to specific categories and brands.
5. Products are grouped into categories, with each category having a unique category ID and name. Each category can include multiple products.

6. Each brand is tracked with a unique brand ID and brand name. A brand can manufacture multiple products.
7. Orders are recorded with a unique order ID, order status, order date, required date, and shipped date. Each order is associated with a specific customer, staff member, and store.
8. Order items are part of orders, with each order item detailing the product ID, quantity, list price, and discount. An order can contain multiple order items, but each order item is linked to one specific order.
9. For each staff member, the database tracks the unique staff ID, first name, last name, email, phone, whether they are currently active, and the store they are affiliated with. Staff members have a manager, and this relationship is also tracked.
10. Each store stocks various products, and this relationship is managed through the stocks entity, which records what products are available at each store and in what quantity.
11. For each stock item, the database tracks the store ID, product ID, and the quantity of the product available at the store.
12. Each order is processed at a store, managed by a staff member, and consists of multiple products.
13. Staff members are assigned to manage specific orders, and they can manage multiple orders. Each staff member is assigned to work in one store.
14. Staff members can also have a supervisory role. Each supervisor can oversee multiple staff members, but each staff member has only one supervisor.
15. The database will also record the hierarchical structure within the staff, capturing who supervises whom and the chain of command within each store.
16. Each staff member works at a specific store but can assist in various tasks across multiple stores if needed.
17. Every order is linked to a customer, processed by a staff member, and occurs at a specific store, thus interlinking customers, staff, and stores.
18. Each product is categorized into a specific category and manufactured by a specific brand, establishing clear links between products, their categories, and the brands that manufacture them.
19. Each store's product stock levels are dynamically updated as sales occur and deliveries are received, ensuring inventory accuracy.
20. The database is designed to provide comprehensive reports on sales trends, staff performance, and inventory levels to assist in strategic decision-making and operational planning.

ERD

The ER diagram for the PGK BikeStore Corp provides a comprehensive visualization of the relationships and entities within the database. This diagram is essential for understanding how different components of the store's operations are interconnected through the database.



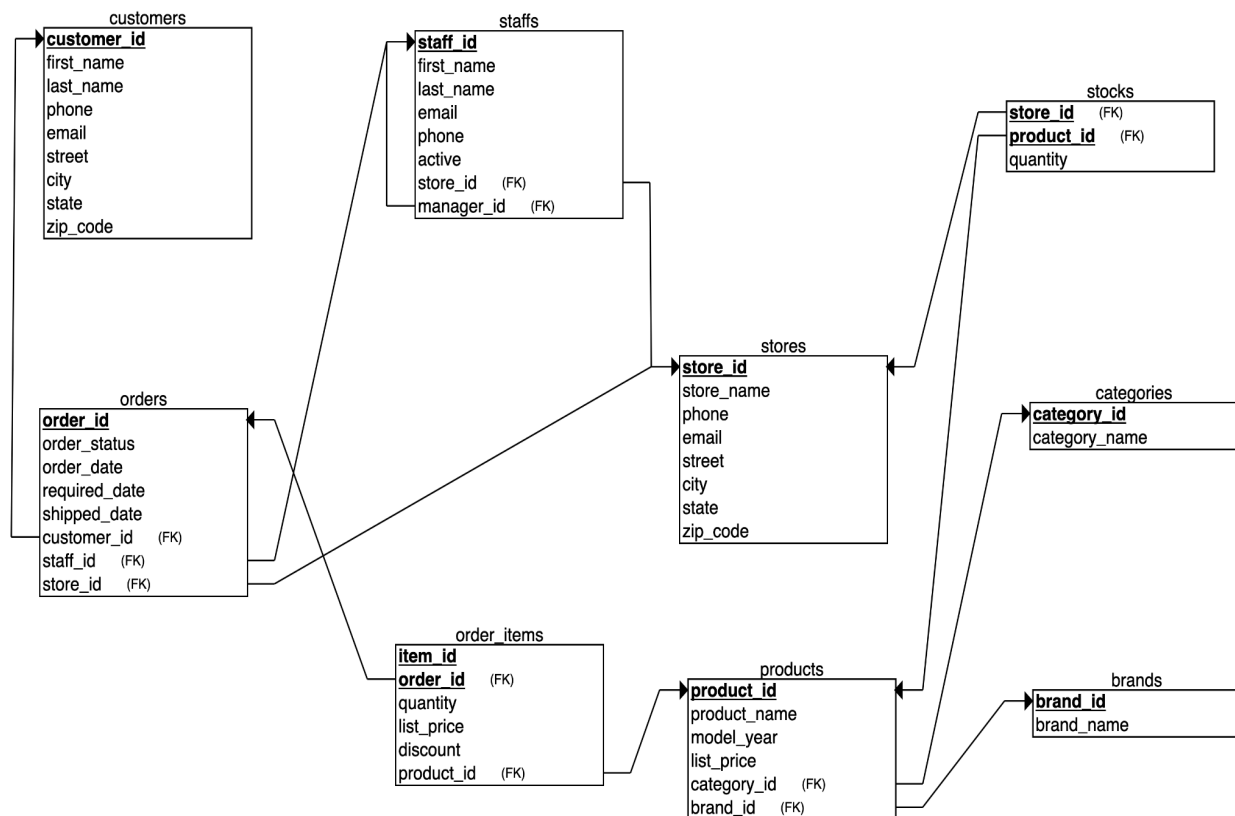
Many of the entity and connection choices in the database define the relationships that shape the system. Products and Stores are connected through a many-to-many relationship mediated by Stocks, allowing the database to track how many of each product are available at different store locations. This setup highlights the distribution of products across various stores.

Staff members are linked by a recursive 'SupervisedBy' relationship, crucial for establishing and managing the hierarchical structure within each store. This relationship allows managers to oversee the tasks of their subordinates, ensuring proper management and accountability.

Order Items are considered a weak entity because they rely on the existence of an Orders entity to which they belong. They are connected to Orders through a 'BelongsTo' relationship and to Products through an 'Includes' relationship. The composite key of OrderID and ItemID in Order Items captures detailed transaction data, emphasizing its dependency on the Orders entity for its existence and the specific details of the products included in each order.

While the current database design does not include a superclass and subclasses, they could be incorporated to enhance flexibility and scalability in certain scenarios. For example, the Staff entity could serve as a superclass with subclasses such as Manager, SalesAssociate, and Technician. This design would allow the database to separate attributes common to all staff members, like StaffID, FirstName, and Email, from role-specific attributes such as ManagerialLevel for managers, SalesTarget for sales associates, and Specialization for technicians. By implementing this generalization-specialization hierarchy, the database would better support future expansions, such as adding new staff roles, while maintaining efficient data organization and enabling more precise queries for role-specific insights.

Relational Schema



Normalization/Denormalization:

For the PGK BikeStore database, we prioritized normalization by adhering to Third Normal Form (3NF) to enhance data consistency, minimize redundancy, and streamline maintenance. Each table was carefully structured to store unique and relevant information, ensuring that updates to one attribute, such as brand or category names, only needed to be made in a single location. For instance, brand and category details were stored in their respective tables rather than being repeated across product records. This design not only prevents errors but also allows for robust queries, like analyzing sales by category or tracking inventory levels across stores, without the risk of inconsistencies. Furthermore, normalization aligns with the database's core objectives of supporting efficient transaction processing, accurate reporting, and real-time updates, all of which are critical for managing operations across a growing retail network.

We chose not to denormalize any tables because its trade-offs, such as increased data redundancy and potential inconsistencies, were not justified for this project. The dataset's manageable size and the database's ability to handle JOIN operations efficiently meant that query performance remained optimal without compromising data integrity. While denormalization could simplify certain queries by reducing the need for table joins, it would come at the cost of duplicating data, which could lead to challenges in maintenance and accuracy as the system evolves. Additionally, maintaining a normalized schema supports future scalability, allowing for the seamless addition of new stores, products, or features without requiring major design changes. By choosing normalization, we ensured a balance between performance, accuracy, and flexibility, making it a sustainable and practical solution for PGK BikeStore's operational needs.

Database Creation and Population:

Data Sources

The data for this project is collected from a sample relational database representing real-world operations of a bike store, sourced from [SQL Server Sample Database](#)

Data Type Selection

To optimize the database design, the data types were selected carefully based on the attributes' characteristics and their intended usage:

1. **Primary Keys:**

- Attributes such as `customer_id`, `product_id`, and `order_id` were assigned the `INT` data type with `AUTO_INCREMENT`. This ensures that each row has a unique identifier and supports indexing for efficient querying.

2. **Textual Data:**

- Attributes like `category_name`, `brand_name`, and `product_name` were defined as `VARCHAR(255)`. This data type is well-suited for variable-length text while imposing a reasonable limit to save storage space.

3. **Numerical Data:**

- Attributes like `list_price` were assigned the `DECIMAL(10, 2)` data type to store monetary values with high precision, avoiding errors associated with floating-point arithmetic.
- The quantity attribute was defined as `INT`, which is suitable for counting items since fractional or negative values are not valid.

4. **Date Attributes:**

- Attributes such as `order_date` and `shipped_date` were assigned the `DATE` data type to store only date information without time components, aligning with the requirements of the business logic.

5. **Boolean-like Data:**

- The active attribute in the `staffs` table was defined as `TINYINT`. This is an efficient choice for binary values, where 1 indicates active and 0 indicates inactive.

6. **Foreign Keys:**

- Attributes like `category_id` and `brand_id` were defined as `INT` to match the data type of the primary keys they reference. This ensures consistency and facilitates relational integrity between tables.

Foreign Key Constraints

Foreign key constraints were implemented to maintain referential integrity between related tables. For instance:

- In the `products` table, `category_id` is a foreign key referencing the `categories` table, and `brand_id` is a foreign key referencing the `brands` table.
- Similarly, in the `orders` table, `customer_id` references the `customers` table, ensuring that every order is linked to an existing customer.

The constraints include:

- **ON DELETE CASCADE:** Ensures that when a parent record (e.g., a category) is deleted, all related child records (e.g., products in that category) are

automatically deleted to prevent orphaned records.

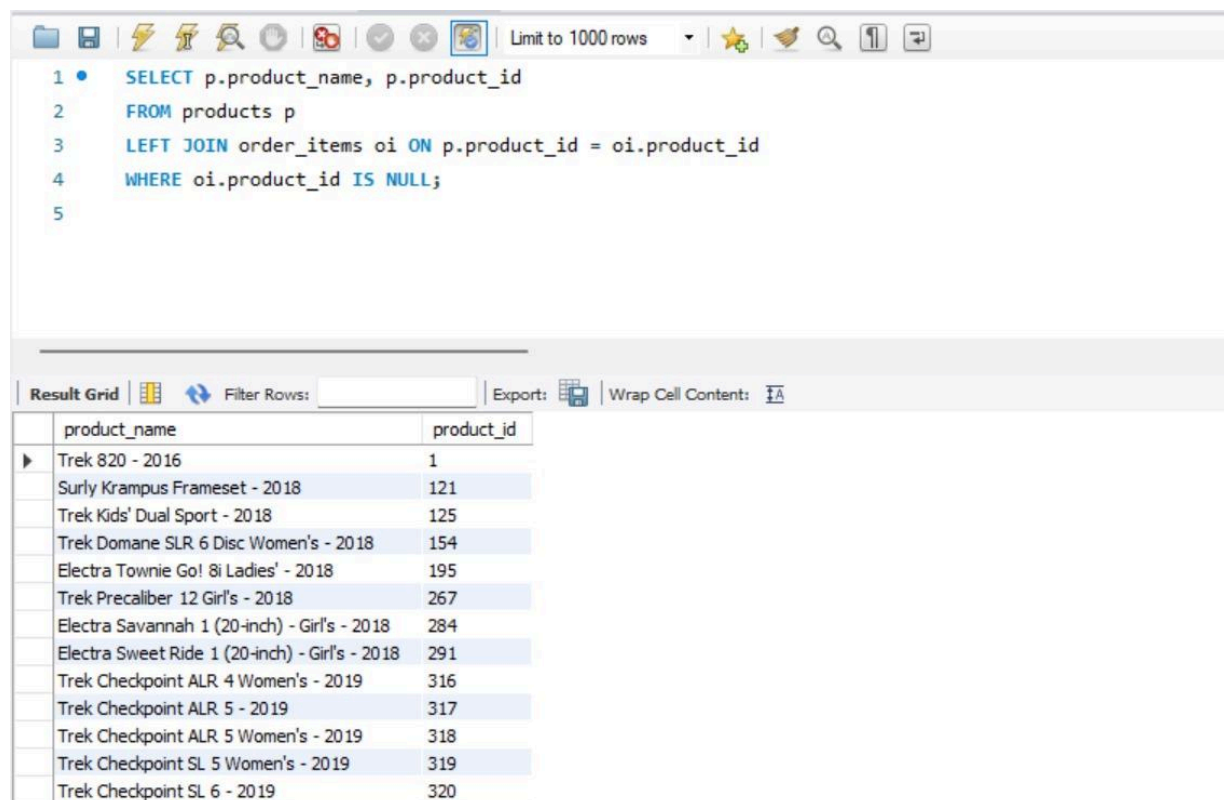
- **ON UPDATE CASCADE:** Ensures that changes to a primary key in a parent table are automatically propagated to related foreign keys in child tables.

These foreign key constraints enforce data integrity, maintain relationships between tables, and simplify database management. This design ensures that the database adheres to the principles of normalization and relational integrity.

SQL Statements:

Query 1: Unsold Products across all stores

This query is designed for store managers, inventory managers, and marketing teams, who are responsible for optimizing stock levels and enhancing product turnover. This query identifies products that have not been sold at any store, aiding in decisions on discounting or discontinuing unsold stock. It utilizes a LEFT JOIN to connect 'products' to 'order_items', ensuring that products with no sales are highlighted. By targeting unsold items, the query supports strategic decisions aimed at reducing inventory costs and aligning stock more closely with consumer demand, crucial for maintaining efficient operations and profitability.



The screenshot shows a SQL query editor with the following query:

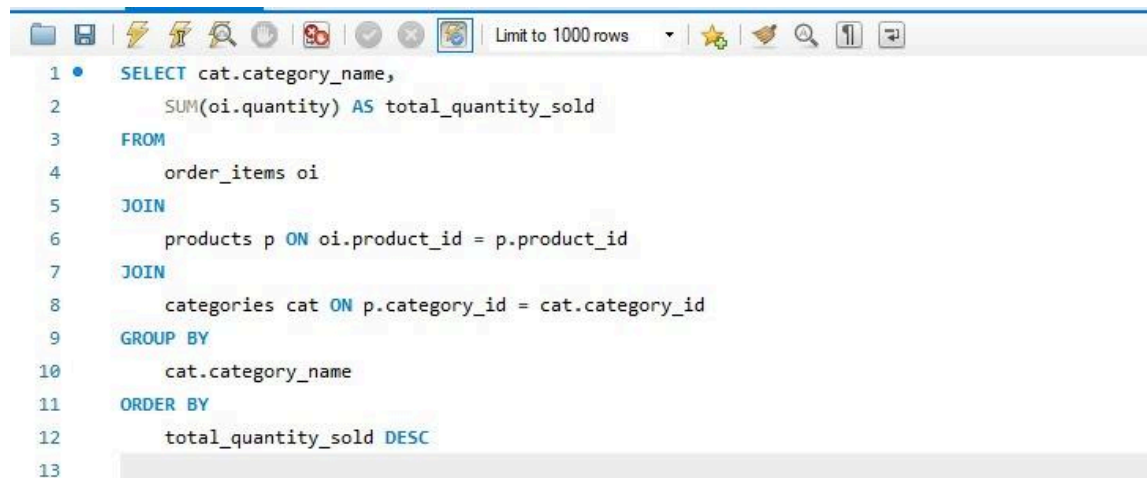
```
1 • SELECT p.product_name, p.product_id
2 FROM products p
3 LEFT JOIN order_items oi ON p.product_id = oi.product_id
4 WHERE oi.product_id IS NULL;
5
```

Below the query editor is a result grid showing the output of the query. The grid has two columns: product_name and product_id. The results are as follows:

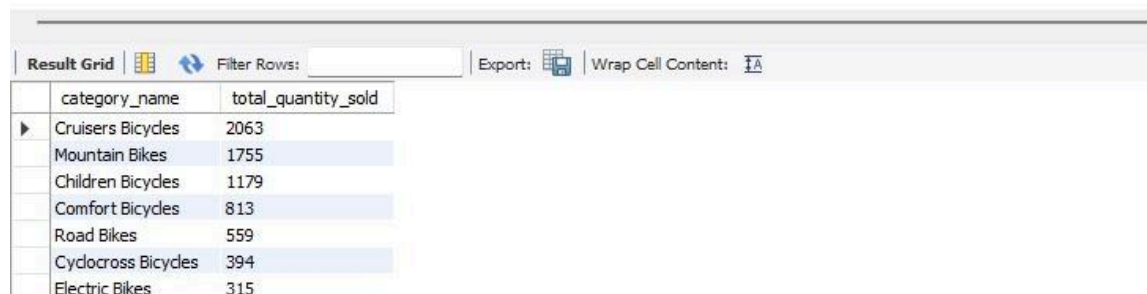
product_name	product_id
Trek 820 - 2016	1
Surly Krampus Frameset - 2018	121
Trek Kids' Dual Sport - 2018	125
Trek Domane SLR 6 Disc Women's - 2018	154
Electra Townie Go! 8i Ladies' - 2018	195
Trek Precaliber 12 Girl's - 2018	267
Electra Savannah 1 (20-inch) - Girl's - 2018	284
Electra Sweet Ride 1 (20-inch) - Girl's - 2018	291
Trek Checkpoint ALR 4 Women's - 2019	316
Trek Checkpoint ALR 5 - 2019	317
Trek Checkpoint ALR 5 Women's - 2019	318
Trek Checkpoint SL 5 Women's - 2019	319
Trek Checkpoint SL 6 - 2019	320

Query 2: Total Items sold per Product Category

This Query is designed for inventory analysts and sales teams to determine the sales performance of different product categories. It employs an INNER JOIN to merge 'order_items' with 'products', and subsequently with 'categories', ensuring that each sold item is accurately linked to its respective category. By using the aggregate function SUM() on 'oi.quantity', the query calculates the total quantity of products sold within each category. The results are then organized using the GROUP BY function, which groups the data by 'cat.category_name', allowing for an easy assessment of which categories perform best. The ORDER BY clause finally arranges the categories in descending order of total quantity sold, offering clear insights into consumer preferences and guiding inventory and marketing strategies to capitalize on the most in-demand products and refining sales strategies to boost overall profitability.



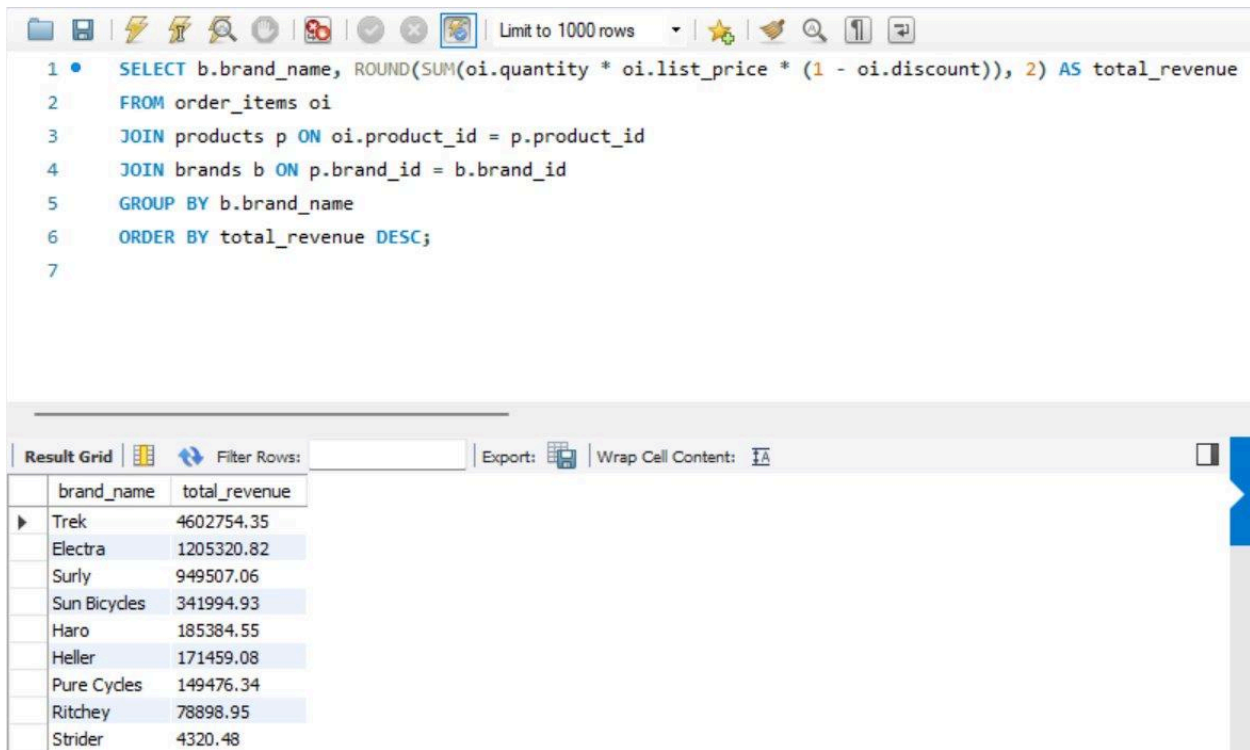
```
1 • SELECT cat.category_name,
2       SUM(oi.quantity) AS total_quantity_sold
3 FROM
4       order_items oi
5 JOIN
6       products p ON oi.product_id = p.product_id
7 JOIN
8       categories cat ON p.category_id = cat.category_id
9 GROUP BY
10      cat.category_name
11 ORDER BY
12      total_quantity_sold DESC
13
```



category_name	total_quantity_sold
Cruisers Bicycles	2063
Mountain Bikes	1755
Children Bicycles	1179
Comfort Bicycles	813
Road Bikes	559
Cyclocross Bicycles	394
Electric Bikes	315

Query 3: Total Revenue per Brand

This Query is specifically crafted for financial analysts and marketing strategists to assess the financial performance of different bicycle brands sold across their stores. By calculating the total revenue generated per brand, this query provides essential insights for strategic planning and resource allocation. The query employs a JOIN operation to correlate 'order_items' with 'products' and then with 'brands', ensuring accurate revenue computation per brand. Revenue is computed using an aggregate function SUM() on the product of quantity sold, list price, and the net effect of any discounts, grouped by brand name to consolidate sales data under each brand. This data helps the company identify top-performing brands, adjust procurement strategies, and optimize marketing efforts to enhance profitability across its product lines.



The screenshot displays a database query editor interface. At the top, there is a toolbar with various icons and a dropdown menu set to 'Limit to 1000 rows'. Below the toolbar, the SQL query is written in a monospaced font. The query selects the brand name and the total revenue (calculated as the sum of quantity multiplied by list price, adjusted for discounts, rounded to two decimal places) for each brand. The results are ordered by total revenue in descending order. Below the query, the 'Result Grid' tab is active, showing a table with two columns: 'brand_name' and 'total_revenue'. The table contains ten rows of data, with 'Trek' having the highest revenue and 'Strider' having the lowest.

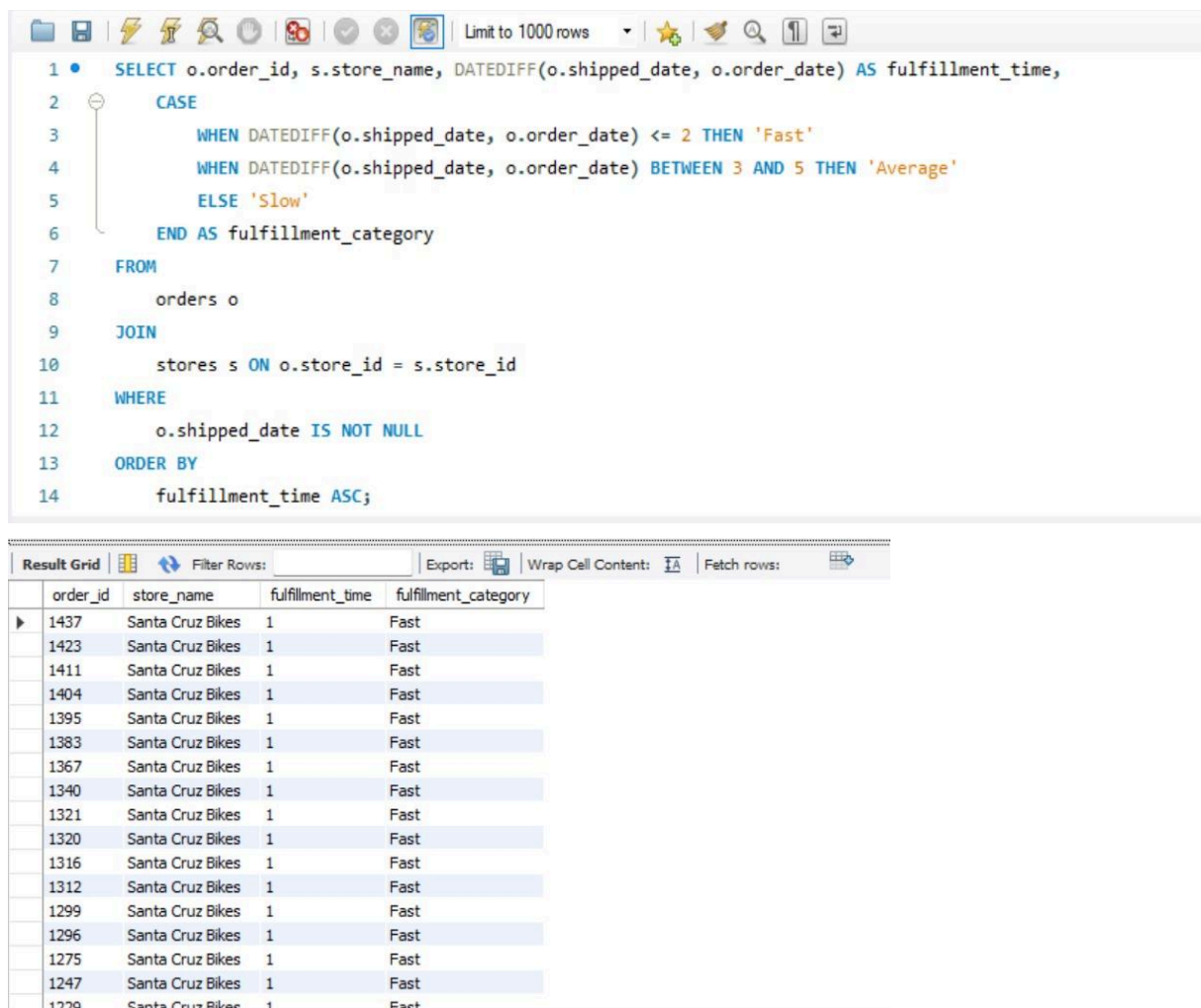
```
1 • SELECT b.brand_name, ROUND(SUM(oi.quantity * oi.list_price * (1 - oi.discount)), 2) AS total_revenue
2 FROM order_items oi
3 JOIN products p ON oi.product_id = p.product_id
4 JOIN brands b ON p.brand_id = b.brand_id
5 GROUP BY b.brand_name
6 ORDER BY total_revenue DESC;
7
```

brand_name	total_revenue
Trek	4602754.35
Electra	1205320.82
Surly	949507.06
Sun Bicycles	341994.93
Haro	185384.55
Heller	171459.08
Pure Cycles	149476.34
Ritchey	78898.95
Strider	4320.48

Query 4: Order Fulfillment Analysis

This Query is designed and tailored for operations managers and logistics coordinators to assess and enhance the efficiency of order fulfillment processes across their store network. The query leverages the DATEDIFF function to calculate the number of days

between the order date and the shipped date, providing a quantitative measure of fulfillment speed. Using a CASE statement, this duration is categorized into 'Fast', 'Average', or 'Slow', which allows for straightforward visual segmentation of order processing times: 'Fast' for shipments completed within 2 days, 'Average' for those taking 3 to 5 days, and 'Slow' for any exceeding 5 days. This classification helps in pinpointing inefficiencies and potential bottlenecks in the shipping process. The insights gained from this analysis guide efforts to streamline operations and enhance customer satisfaction by enabling proactive management of order processing times. Additionally, employing an INNER JOIN ensures each order is analyzed in its respective store context, providing a detailed and actionable overview of fulfillment performance across the entire retail chain.



The image shows a SQL query editor window with a toolbar at the top. The query is as follows:

```
1 • SELECT o.order_id, s.store_name, DATEDIFF(o.shipped_date, o.order_date) AS fulfillment_time,
2     CASE
3         WHEN DATEDIFF(o.shipped_date, o.order_date) <= 2 THEN 'Fast'
4         WHEN DATEDIFF(o.shipped_date, o.order_date) BETWEEN 3 AND 5 THEN 'Average'
5         ELSE 'Slow'
6     END AS fulfillment_category
7 FROM
8     orders o
9 JOIN
10    stores s ON o.store_id = s.store_id
11 WHERE
12    o.shipped_date IS NOT NULL
13 ORDER BY
14    fulfillment_time ASC;
```

Below the query editor is a 'Result Grid' window. It has a toolbar with 'Filter Rows', 'Export', 'Wrap Cell Content', and 'Fetch rows'. The grid displays the following data:

order_id	store_name	fulfillment_time	fulfillment_category
1437	Santa Cruz Bikes	1	Fast
1423	Santa Cruz Bikes	1	Fast
1411	Santa Cruz Bikes	1	Fast
1404	Santa Cruz Bikes	1	Fast
1395	Santa Cruz Bikes	1	Fast
1383	Santa Cruz Bikes	1	Fast
1367	Santa Cruz Bikes	1	Fast
1340	Santa Cruz Bikes	1	Fast
1321	Santa Cruz Bikes	1	Fast
1320	Santa Cruz Bikes	1	Fast
1316	Santa Cruz Bikes	1	Fast
1312	Santa Cruz Bikes	1	Fast
1299	Santa Cruz Bikes	1	Fast
1296	Santa Cruz Bikes	1	Fast
1275	Santa Cruz Bikes	1	Fast
1247	Santa Cruz Bikes	1	Fast
1229	Santa Cruz Bikes	1	Fast

Alternative Database Designs: MongoDB and Neo4j

If we were to use MongoDB or Neo4j instead of MySQL, the design and structure of the database would change significantly to fit the features of these systems.

Using MongoDB:

MongoDB is a document-based database, which means instead of tables, we would use collections, and data would be stored as JSON-like documents. For example, instead of having separate tables for Orders and OrderItems, all order details could be embedded within a single Orders document. This approach reduces the need for joins but might result in some data duplication, like customer details being included in every order document.

MongoDB is schema-less, so we wouldn't need to predefine a rigid structure. This flexibility is helpful if the data changes frequently, such as adding new attributes to products or orders. However, we would need to carefully design how data is embedded or referenced to ensure good performance and consistency.

Using Neo4j:

Neo4j is a graph database, which is perfect for scenarios where relationships are central. Instead of tables, we would represent entities like Customers, Orders, Staff, and Stores as nodes, and their relationships (like PLACED, MANAGES, or STOCKS) as edges.

For instance, a Customer node could have a PLACED relationship to an Order node, and an Order node could have an INCLUDES relationship to multiple Product nodes. Neo4j would make it easier to query things like "Who placed orders containing a specific product?" or "What are the hierarchical relationships between staff members?"

How the Design and Scenario Would Change:

If we used MongoDB, the database would focus on flexibility and embedding data for fast access. For example, all order-related details might be stored in one document instead of being split into multiple tables. However, this could lead to data duplication, like repeating product details in every order where the product appears.

If we used Neo4j, the focus would shift to explicitly modeling relationships, which is great for exploring connections, like identifying which staff members work in the same store or analyzing how products flow across stores. Neo4j's graph-based design would also simplify complex queries about relationships.

In both cases, the way we query the database would change. MongoDB uses a query language that works with JSON documents, while Neo4j uses Cypher, a query language designed for traversing graphs.

Overall, MongoDB would be ideal if we needed flexibility and fast reads for semi-structured data, while Neo4j would be a great fit for exploring and analyzing relationships within the data.

References:

- [1] [Bike Store Relational Database | SQL](#)
- [2] [SQL Server Sample Database](#)
- [3] https://data.world/**
- [4] <https://iu.instructure.com/courses/2252743/files/175970124?wrap=1>