|  |  |  |
| --- | --- | --- |
| Customer\_id(PK) | Customer  \_name | Customer  \_email |
| 1 | Jean Doe | jsmith@lmail.com |
| 2 | Rebecca Yeboah | ryeb@lmail.com |
| 3 | Jean Doe | jdoe@lmail.com |
| 4 | Bertina Ayuure | bayuure@lmail.com |

**Order** **Customer**

|  |  |  |
| --- | --- | --- |
| Order\_id  (PK) | Order\_date | Customer\_  id(FK) |
| 001 | 2023-11-01 | 1 |
| 002 | 2023-11-02 | 2 |
| 003 | 2024-11-01 | 3 |
| 004 | 2024-11-02 | 4 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Order\_details\_id(PK) | Order\_id(FK) | Products\_id(FK) | Quantity | Total |
| 1 | 001 | 101 | 5 | 1000 |
| 2 | 002 | 102 | 4 | 500 |
| 3 | 003 | 103 | 6 | 1250 |
| 4 | 004 | 104 | 10 | 2500 |

**Order\_details**

**Product**

|  |  |  |
| --- | --- | --- |
| Product\_id(PK) | Product\_name | category |
| 101 | Laptop | Electronics |
| 102 | Phone | Electronics |
| 103 | Mouse | Electronics |
| 104 | Keyboard | Accessories |

|  |  |  |
| --- | --- | --- |
| City\_id | City\_name | Country\_id(FK) |
| C001 | Accra | F11 |
| C002 | Kigali | F31 |
| C003 | Bonn | F21 |

**Inventory City**

|  |  |  |  |
| --- | --- | --- | --- |
| Supplier\_  Id(PK) | Supplier\_  name | Stock\_quantity | Product\_id |
| A01 | CompuGhana | 120 | 101 |
| A02 | RoboTech | 50 | 102 |
| A03 | T\_Shop | 23 | 104 |

**Country**

|  |  |
| --- | --- |
| Country\_id(PK) | country |
| F11 | Ghana |
| F21 | Germany |
| F31 | Rwanda |

**1. Customers Table**

* **Columns:** customer\_id (PK), customer\_name, customer\_email
* Each customer is uniquely identified by customer\_id.

**2. Products Table**

* **Columns:** product\_id (PK), productname, supplierid (FK)
* Each product is uniquely identified by productid.

**3. Inventory Table**

* **Columns:** supplier\_id (PK), suppliername, stock\_quantity
* Each inventoryz is uniquely identified by supplier\_id.

**4. Orders Table**

* **Columns:** order\_id (PK), order\_date, customer\_id (FK)
* Each order is uniquely identified by orderid and references customer\_id.

**5. Order\_Details Table**

* **Columns:** order\_details\_id(PK), order\_id (FK), product\_id (FK), quantity, total
* This table resolves the many-to-many relationship between orders and products.

**6. City Table**

* **Columns:** city\_id(PK), city\_name, country\_id(FK)

7. **Country Table**

* **Columns:** country\_id(PK), country\_name

**How Normalization Helps Reduce Redundancy and Improve Data Integrity**

**1. Reduction of Redundancy:**

* Normalization eliminates duplicate data by dividing a large table into smaller, related tables.
* For example, separating Customers and Orders into distinct tables ensures customer information (e.g., name, address) is stored once and referenced as needed. This avoids repeating the same data across multiple rows.

**2. Improved Data Integrity:**

* Normalization ensures consistency through **referential integrity** by using **primary keys** and **foreign keys**.
  + E.g., if a customerid in the Orders table references a non-existent customerid in the Customers table, it would result in an error, ensuring data accuracy.
* **Update Anomalies:** Updating customer information occurs in one place (Customers table), reducing the risk of conflicting data across multiple rows.
* **Deletion Anomalies:** Deleting an order won’t remove critical customer data, as it is stored independently.

**3. Scalability and Efficiency:**

* Smaller, related tables enable more efficient query performance and faster indexing.
* Data normalization aligns with **ACID (Atomicity, Consistency, Isolation, Durability)** properties, critical for database transaction management.

**Trade-Offs and When Denormalization Might Be Necessary**

**1. Performance Trade-Offs:**

* **Join Overhead:** Highly normalized databases often require multiple joins to retrieve data from related tables. This can slow down query performance, especially for complex queries in high-traffic systems.
* **Read-Heavy Systems:** In systems where reads are more frequent than writes, the cost of frequent joins may outweigh the benefits of reduced redundancy.

**2. Increased Complexity:**

* Normalized databases can be harder to understand and maintain for teams unfamiliar with relational design principles.
* Writing queries requires a deeper understanding of relationships and may involve more complex SQL statements.

**When Denormalization Might Be Necessary:**

1. **Performance Optimization:**
   * Denormalization is often used in **data warehousing** and **read-heavy applications** (e.g., dashboards, reporting tools) to avoid expensive joins by storing pre-aggregated data.
2. **Faster Query Execution:**
   * For high-volume transactional systems, denormalization can help reduce query latency by duplicating some data across tables.
3. **Simplified Schema Design:**
   * Denormalized schemas like **star schema** or **snowflake schema** are popular in analytics databases to simplify querying.

**Example of Denormalization:**

* Storing both customername and customeremail in the Orders table to avoid frequent joins with the Customers table in a reporting system. This introduces redundancy but improves performance for read operations.