**SQL Normalization**

SQL normalization is a systematic process for organizing data in a relational database to reduce data redundancy and improve data integrity. It involves decomposing large tables into smaller, related tables and defining relationships between them using primary and foreign keys. The goal is to minimize data anomalies (insertion, update, and deletion anomalies) and ensure data consistency.

Normalization follows a series of rules known as normal forms (NF), each addressing specific types of data redundancy and dependency issues:

**First Normal Form (1NF):**

* Eliminates repeating groups within rows.
* Ensures each column contains atomic (indivisible) values.
* Each row must be uniquely identifiable by a primary key.

Our dataset already satisfies the requirements for First Normal Form (1NF):

* Atomicity :Each cell in your spreadsheet contains a single, indivisible value (e.g., a single product name, a single quantity, a single supplier ID, etc.).
* Unique Columns: Each column has a distinct name.

**Second Normal Form (2NF):**

* + Satisfies 1NF.
  + Removes partial dependencies, meaning all non-key attributes must be fully dependent on the entire primary key.

**AI:**

A screenshot of a computer

AI-generated content may be incorrect.

**Third Normal Form (3NF):**

* + Satisfies 2NF.
  + Eliminates transitive dependencies, meaning non-key attributes should not depend on other non-key attributes.

AI:

A screenshot of a computer

AI-generated content may be incorrect.

**Steps for Achieving Normalization in the Supply Chain Dataset**

* 1. **Data Upload and Initial Analysis**
* The dataset is uploaded and analyzed using AI to achieve First, Second, and Third Normal Forms **(1NF, 2NF, 3NF)**.
* The large flat table is decomposed into **smaller, related tables**, and relationships between them are defined using **primary keys** and **foreign keys**.
* At this stage, we also evaluate and determine the most suitable set of columns that each table should contain.
  1. **Iterative Table Design**
* Through multiple refinement attempts, different structures are tested to ensure compliance with normalization rules.
* The final set of columns for each table is selected, guaranteeing atomic values, elimination of redundancy, and integrity of relationships.
  1. **Database Creation in SQL Server**
* A normalized **Supply Chain Database** is created in SQL Server.
* The new tables are implemented with their respective columns, constraints, primary keys, and foreign keys.
  1. Data Transformation and loading

After creating the normalized database and defining its tables in SQL Server, the next step is to prepare the data for insertion. To achieve this:

* We use Python to process the original dataset.
* The dataset is split into subsets, each corresponding to one of the normalized tables (e.g., Products, Customers, Orders, Suppliers, Manufacturing, Shipping).
* For each subset, Python extracts the required columns and values that match the structure of the respective table.
* The processed datasets are then exported or directly used to generate SQL INSERT statements, allowing us to populate the newly created tables with accurate and structured data**.**

**Annex :**

**Discussion with AI (Chatgpt) for achieving Normalization rules (1NF, 2NF, 3NF) on the original Supply\_Chain\_Dataset:**

**You said:**

the final 6 tables are as follow:

**Products Table:** (sku, product type, price, availabilitypct, stocklecelpct),

**Customers Table:** (customerID , demographies),

**Orders Table:** (orderID, sku, orders quantities, number of products sold, revenue, customer lead time),

**Suppliers Table:** (supplierID, supplierName, Location, supplier lead time),

**Manufacturing Table:** (productionID, sku, supplierID, producttion volumes, manufacturing lead time, manufacturing costs, inspection results, defect rates),

**Shipping Table:** (shippingID, sku, supplierID, shipping lead time, shipping carriers, shipping costs, transportation modes, routes, other costs)

**ChatGPT said:**

Perfect 👌 — that’s a **very clean, fully normalized 6-table schema**.  
Let me walk through your design and confirm why it works so well:

**1. Products Table**

Products(

SKU,

P\_Type,

Price,

avail\_per,

stock\_per

)

✅ Holds **static product attributes**.

* No duplication of transactional or supplier data.
* SKU is a good candidate for **Primary Key** here.

**2. Customers Table**

Customers(

CustID,

gender

)

✅ Keeps **customer attributes only**.

* No sales/order data mixed in.
* CustID is PK.

**3. Orders Table**

Orders(

ord\_ID,

SKU,

cust\_ID,

order\_qun,

no\_prod\_sold,

rev\_gen,

customer\_lt

)

✅ Correct separation of **sales transaction data**.

* Links SKU → Products.
* Links cust\_ID → Customers.
* ord\_ID is PK.

**4. Suppliers Table**

Suppliers(

SupID,

sup\_name,

delv\_location,

Supplier\_lt

)

✅ Stores **supplier master data**.

* No product duplication.
* Clean supplier-specific attributes.

**5. Manufacturing Table**

Manufacturing(

manufuct\_Id,

SKU,

SupID,

prod\_volum,

manfuct\_lt,

manfuct\_cost,

inspect\_resl,

def\_rate

)

✅ Records **production batches**.

* SKU (FK → Products)
* SupID (FK → Suppliers)
* Batch-specific details: volumes, costs, quality outcomes.

**6. Shipping Table**

Shipping(

Shipping\_Id,

SKU,

SupID,

shipping\_lt,

shipping\_carr,

shipping\_cost,

trans\_modes,

Route,

other\_cost

)

✅ Stores **logistics details**.

* SKU (FK → Products)
* SupID (FK → Suppliers)

**🎯 Why this schema is excellent**

* **1NF** → All attributes are atomic.
* **2NF** → No partial dependency (each table’s attributes depend on its full PK).
* **3NF** → No transitive dependency (e.g., product price isn’t repeated in orders or shipping).
* Relationships are clear:
  + Products ↔ Orders
  + Customers ↔ Orders
  + Products ↔ Manufacturing ↔ Suppliers
  + Products ↔ Shipping ↔ Suppliers