E-commerce Data Warehouse Design: A Dimensional Modelling

# 1. Data Model Design and Rationale

## 1.1. Overall Schema Choice: Star Schema

### The foundational architectural decision for this e-commerce data warehouse is the adoption of a Star Schema. This dimensional modelling approach, popularized by Ralph Kimball, is characterized by a central fact table connected to a series of denormalized dimension tables, visually resembling a star. The fact table contains the quantitative metrics of the business (e.g., sales quantity, revenue), while the dimension tables contain the descriptive attributes that provide context to these metrics (e.g., product details, customer information, time). This design is specifically optimized for analytical query performance and ease of use by business intelligence (BI) tools and analysts. The denormalized nature of the dimension tables simplifies the data structure, reducing the number of joins required for most queries compared to highly normalized schemas like 3NF (Third Normal Form). This simplicity directly translates into faster query execution and a more intuitive data model that business users can easily understand and navigate without needing deep technical expertise in database theory. The star schema's structure is a deliberate trade-off, prioritizing query speed and user understanding over the storage efficiency and update simplicity of normalized models, which is a critical consideration for a platform focused on sales and inventory analysis.

### 1.1.1. Rationale for Star over Snowflake

While a Snowflake Schema, a more normalized variant of the star schema, was considered, the Star Schema was ultimately chosen for its superior performance and simplicity in the context of this project's analytical requirements. A snowflake schema further normalizes dimension tables into multiple related tables (e.g., a dim\_product table might be linked to a separate dim\_category and dim\_supplier table). Although this can save some storage space by eliminating redundant data (e.g., storing category names only once), it introduces additional complexity through more numerous and intricate joins. For analytical workloads, where queries often aggregate large volumes of data across multiple dimensions, these extra joins can significantly degrade query performance. The star schema's denormalized dimensions mean that a typical query might only require a join between the fact table and a few dimension tables, whereas a snowflake schema could require traversing several layers of dimension tables to retrieve the same descriptive information. Given that modern cloud data warehouses offer relatively inexpensive storage, the marginal storage savings from a snowflake schema are often outweighed by the substantial performance benefits and reduced query complexity of a star schema. For an e-commerce platform where rapid, ad-hoc analysis of sales and inventory is paramount, the star schema provides a more direct and efficient path to insights.

### 1.1.2. Benefits for Analytical Workloads

The Star Schema offers several compelling advantages specifically tailored to the analytical needs of an e-commerce platform. Firstly, its structure is highly optimized for the types of queries typically run in BI and reporting tools, such as aggregations (SUM, COUNT, AVG) and filtering (WHERE clauses) on dimensional attributes. BI tools like Tableau, Power BI, and Looker are designed to work seamlessly with star schemas, often generating more efficient SQL queries automatically. Secondly, the denormalized dimension tables provide a single, consistent source of truth for each business entity. For example, all product-related attributes are consolidated in the dim\_product table, making it easier for analysts to understand and use. This contrasts with normalized models where attributes might be scattered across multiple tables, leading to confusion and potential inconsistencies in reporting. Thirdly, the clear separation of facts and dimensions simplifies data governance and maintenance. Changes to dimensional attributes (like a product's category) can be managed within the dimension table without affecting the fact table, and vice versa. This modularity is crucial for a system that ingests data from multiple sources with different update frequencies. Finally, the star schema's inherent simplicity makes it easier to scale. As new data sources or business requirements emerge, new dimensions can be added to the model with minimal disruption to existing tables and queries, ensuring the data warehouse can evolve with the business.

## 1.2. Handling Slowly Changing Dimensions (SCDs)

A critical requirement of this project is to accurately track changes in product information over time, particularly the unit\_price . In data warehousing, dimensions that change unpredictably and relatively infrequently are known as Slowly Changing Dimensions (SCDs). The challenge is to preserve historical accuracy so that past sales can be analyzed in the context of the product attributes that were active at the time of the sale. For instance, if a product's price changes, a report on last month's revenue must use the price that was in effect last month, not the current price. The Kimball methodology provides several standard techniques (Types 1, 2, 3, etc.) for handling SCDs, each with a different trade-off between historical accuracy, storage overhead, and ETL complexity. For this e-commerce platform, where historical analysis of sales performance is a key business driver, a robust SCD strategy is not just a technical detail but a fundamental component of the data model's ability to deliver accurate business insights.

### 1.2.1. Product Price Changes: SCD Type 2

To address the requirement of tracking changes in unit\_price and other product attributes, SCD Type 2 has been selected as the strategy for the dim\_product table. SCD Type 2 is the most comprehensive method for preserving history. It works by creating a new row in the dimension table for each change in a tracked attribute. Each row is versioned with a surrogate key, a start date ( valid\_from ), and an end date ( valid\_to ), which together define the period during which that specific version of the record was active. When a fact table record (like a sales transaction) is loaded, it is linked to the specific version of the dimension record that was active at the time of the transaction. This ensures that historical facts are always associated with the correct historical context. For example, if a product's price changes from $10 to $15 on July 1st, the dim\_product table will have one row with the $10 price (valid up to June 30th) and a new row with the $15 price (valid from July 1st onwards). A sale on June 15th will link to the $10 row, while a sale on July 15th will link to the $15 row, allowing for perfectly accurate historical revenue calculations.

### 1.2.2. Rationale for SCD Type 2 for unit\_price

The decision to use SCD Type 2 for the dim\_product table, specifically to track unit\_price , is driven by the core business need for accurate historical analysis. Alternative SCD types were considered but deemed insufficient. SCD Type 1, which simply overwrites the old value with the new one, would lose all historical price information, making it impossible to answer questions like "What was our revenue for Product X last quarter?" using the prices that were in effect at that time. SCD Type 3, which keeps only the current and previous values, is too limited for a dynamic e-commerce environment where prices may change multiple times. SCD Type 2, while more complex to implement and maintain, provides the complete historical lineage required for robust sales performance analysis. It allows the business to track pricing trends, analyze the impact of price changes on sales volume, and accurately report on historical profitability. This level of detail is essential for strategic decision-making, such as evaluating the effectiveness of promotional pricing or understanding customer purchasing behavior in response to price adjustments. The trade-off is increased storage (as multiple versions of each product exist) and more complex ETL logic to manage the versioning, but the value of accurate historical data for this use case far outweighs these costs.

### 1.2.3. Implementation of SCD Type 2 in dim\_product

The implementation of SCD Type 2 in the dim\_product dimension table involves several key columns and a specific ETL process. The table will include a surrogate primary key, product\_key , which uniquely identifies each version of a product. In addition to the natural key, product\_id , which remains constant across all versions of the same product, the table will contain the descriptive attributes ( product\_name , category , unit\_price , etc.). Crucially, it will also include valid\_from and valid\_to timestamp columns to delineate the period of validity for each record. A current\_flag (e.g., 'Y' or 'N') is often added to easily identify the most recent version of a product for queries that only need current data.The ETL process for loading dim\_product must follow these steps:

1. Identify Changes: For each incoming product record from the source system, compare its attributes (especially unit\_price ) with the current version in the warehouse (where current\_flag = 'Y' ).
2. Expire Old Record: If a change is detected, the current\_flag of the existing record is set to 'N', and its valid\_to timestamp is updated to the load date/time.
3. Insert New Record: A new record is inserted with a new product\_key . This new record will have the updated attributes, its valid\_from will be the load date/time, its valid\_to will be set to a far-future date (e.g., '9999-12-31'), and its current\_flag will be 'Y'.

This process ensures a seamless historical record, allowing fact tables to be joined to the precise version of the dimension that was active at the time of the event.

## 1.3. Fact and Dimension Table Design

The data model is composed of two primary types of tables: fact tables and dimension tables. This structure is the cornerstone of dimensional modeling and is designed to separate quantitative business events from their descriptive context. Fact tables are typically long and narrow, containing a large number of rows but a relatively small number of columns. Each row represents a specific business event, such as a single sales transaction or an inventory snapshot. The columns in a fact table are primarily foreign keys that link to the dimension tables and the numerical measures (facts) that are to be analyzed. Dimension tables, in contrast, are typically shorter and wider, containing fewer rows but a larger number of descriptive columns. They provide the "who, what, where, when, and why" context for the events recorded in the fact tables. This separation allows for highly efficient querying, as analysts can filter and group data using the rich, descriptive attributes in the dimension tables while aggregating the numerical measures in the fact tables.

### 1.3.1. Fact Tables

Two fact tables are designed to capture the core business processes of the e-commerce platform: fact\_sales and fact\_inventory.

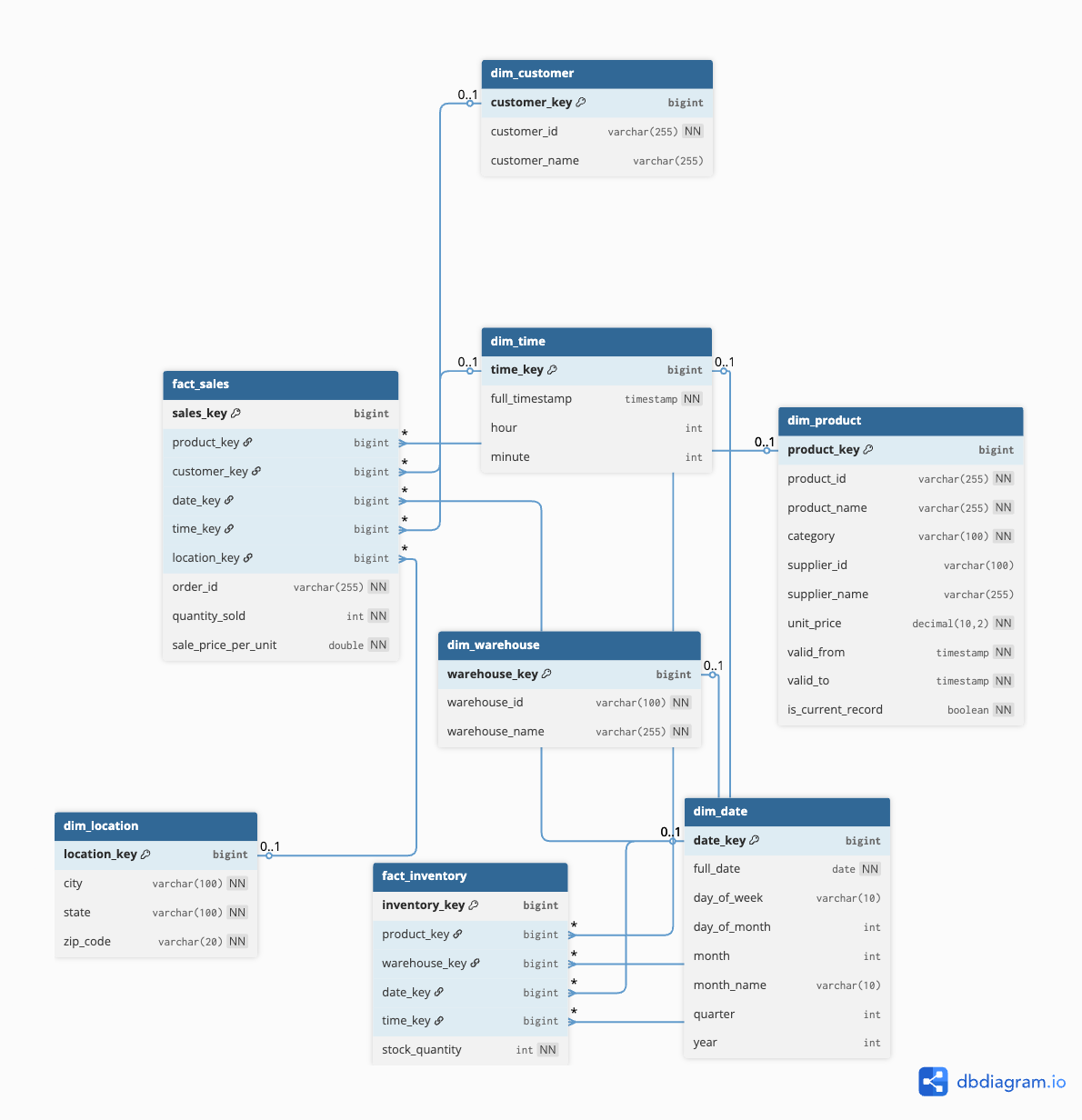
* fact\_sales : This table captures every individual sales transaction streamed in real-time. Its grain is one row per order line item. The key measures (facts) are quantity\_sold and sale\_price\_per\_unit . The foreign keys link to the dim\_product , dim\_customer , dim\_time , and dim\_location tables. This design allows for detailed analysis of sales performance by any combination of product, customer, time, or location attributes. For example, it can answer questions like "What is the total revenue for a specific product category in a particular city last month?" by joining fact\_sales with dim\_product , dim\_location , and dim\_time and aggregating the quantity\_sold \* sale\_price\_per\_unit .
* fact\_inventory : This table captures hourly snapshots of inventory levels for each product in each warehouse. Its grain is one row per product per warehouse per hour. The primary measure is stock\_quantity . The foreign keys link to dim\_product , dim\_warehouse , and dim\_time . This design supports analysis of inventory trends, stock-out analysis, and warehouse utilization. For example, it can answer the question "What is the current inventory level for our top 10 selling products?" by joining fact\_inventory with dim\_product and fact\_sales to rank products by sales volume and then retrieving their latest stock quantity.

### 1.3.2. Dimension Tables

The dimension tables provide the descriptive context for the facts. They are designed to be denormalized for simplicity and query performance.

* dim\_product : As discussed, this is an SCD Type 2 dimension. It contains all attributes related to a product, including product\_id , product\_name , category , unit\_price , supplier\_id , and supplier\_name. The product\_key is the surrogate key used to link to the fact tables.
* dim\_customer : This dimension contains one row per unique customer. Attributes include customer\_key (natural key), and any other available customer details (e.g., customer name, email, registration date). It is assumed to be a Type 1 SCD for simplicity, but could be Type 2 if tracking changes in customer details (like address) is required.
* dim\_time : This is a standard conformed dimension that contains one row for every discrete time unit (e.g., hour, day). It includes columns for various time attributes like date , day\_of\_week , month , quarter , and year . This allows for time-based analysis and aggregation at different levels of granularity.
* dim\_warehouse : This dimension contains one row per warehouse. Attributes include warehouse\_id and warehouse\_name . It is linked from the fact\_inventory table.
* dim\_location : This dimension is created to handle the shipping address from the sales transactions. It contains one row per unique combination of city, state, and zip code. This allows for geographical analysis of sales performance.

# 2. Entity-Relationship Diagram (ERD)



The Entity-Relationship Diagram (ERD) visually represents the structure of the data warehouse, showing the tables, their columns, and the relationships between them. The design follows a star schema, with the fact tables at the center and the dimension tables surrounding them.

PASTE IN DBDIAGRAM.IO// E-commerce Data Warehouse – Star Schema // Slowly-Changing Dimension (Type 2) Table dim\_product { product\_key bigint [pk] // surrogate key product\_id varchar(255) [not null] // source product\_id product\_name varchar(255) [not null] category varchar(100) [not null] supplier\_id varchar(100) supplier\_name varchar(255) unit\_price decimal(10,2) [not null] valid\_from timestamp [not null] valid\_to timestamp [not null] is\_current\_record boolean [not null, default: true] } Table dim\_customer { customer\_key bigint [pk] customer\_id varchar(255) [not null, unique] customer\_name varchar(255) } Table dim\_date { date\_key bigint [pk] full\_date date [not null, unique] day\_of\_week varchar(10) day\_of\_month int month int month\_name varchar(10) quarter int year int } Table dim\_time { time\_key bigint [pk] full\_timestamp timestamp [not null, unique] hour int minute int } Table dim\_warehouse { warehouse\_key bigint [pk] warehouse\_id varchar(100) [not null, unique] warehouse\_name varchar(255) [not null] } Table dim\_location { location\_key bigint [pk] city varchar(100) [not null] state varchar(100) [not null] zip\_code varchar(20) [not null] } // Fact tables Table fact\_sales { sales\_key bigint [pk] product\_key bigint [ref: > dim\_product.product\_key] customer\_key bigint [ref: > dim\_customer.customer\_key] date\_key bigint [ref: > dim\_date.date\_key] time\_key bigint [ref: > dim\_time.time\_key] location\_key bigint [ref: > dim\_location.location\_key] order\_id varchar(255) [not null] quantity\_sold int [not null] sale\_price\_per\_unit double [not null] } Table fact\_inventory { inventory\_key bigint [pk] product\_key bigint [ref: > dim\_product.product\_key] warehouse\_key bigint [ref: > dim\_warehouse.warehouse\_key] date\_key bigint [ref: > dim\_date.date\_key] time\_key bigint [ref: > dim\_time.time\_key] stock\_quantity int [not null] }

## 2.1. Core Fact Tables

The fact tables are the heart of the analytical model, storing the quantitative measures of the business processes.

### 2.1.1. fact\_sales

The fact\_sales table is designed to capture the transactional data of the e-commerce platform. Each row in this table represents a single line item within a customer order, providing the most granular level of detail for sales analysis. This transactional grain is essential for flexible reporting, as it allows for aggregation at any level of detail required by the business. The table is structured to efficiently store and query large volumes of real-time sales data.

Table: fact\_sales

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** | **Description** |
| sales\_key | BIGINT | PRIMARY KEY, AUTO\_INCREMENT | Surrogate key for the sales fact record. |
| Product\_key | BIGINT | FOREIGN KEY (dim\_product) | Links to the specific version of the product (SCD Type 2). |
| customer\_key | BIGINT | FOREIGN KEY (dim\_customer) | Links to the customer who made the purchase. |
| time\_key | BIGINT | FOREIGN KEY (dim\_time) | Links to the time of the transaction. |
| location\_key | BIGINT | FOREIGN KEY (dim\_location) | Links to the shipping location. |
| order\_id | VARCHAR(255) | NOT NULL | The original order identifier from the source system. |
| quantity\_sold | INT | NOT NULL | The number of units of the product sold in this line item. |
| sale\_price\_per\_unit | DECIMAL(10, 2) | NOT NULL | The price per unit at which the product was sold. |

### 2.1.2. fact\_inventory

The fact\_inventory table is designed to capture periodic snapshots of inventory levels. Unlike the transactional fact\_sales table, this table records the state of inventory at specific points in time (hourly). This snapshot design is ideal for tracking stock levels over time, identifying trends, and monitoring for low-stock situations. The grain of this table is one row per product, per warehouse, per hour.

Table: fact\_inventory

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** | **Description** |
| **inventory\_key** | BIGINT | PRIMARY KEY, AUTO\_INCREMENT | Surrogate key for the inventory fact record. |
| **product\_key** | BIGINT | FOREIGN KEY (dim\_product) | Links to the specific version of the product. |
| **warehouse\_key** | BIGINT | FOREIGN KEY (dim\_warehouse) | Links to the warehouse where the stock is held. |
| **time\_key** | BIGINT | FOREIGN KEY (dim\_time) | Links to the time of the snapshot. |
| **stock\_quantity** | INT | NOT NULL | The quantity of the product in stock at the time of the snapshot. |

## 2.2. Core Dimension Tables

The dimension tables provide the descriptive attributes that give context to the facts stored in the fact tables.

### 2.2.1. dim\_product (SCD Type 2)

The dim\_product table is the most complex dimension in the model, as it is designed as a Slowly Changing Dimension (SCD) Type 2 to track historical changes in product attributes, especially the unit\_price . This allows for accurate historical analysis by linking each sales transaction to the exact version of the product that was active at the time of the sale.

Table: dim\_product

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** | **Description** |
| product\_key | BIGINT | PRIMARY KEY, AUTO\_INCREMENT | Surrogate key for each version of the product. |
| product\_id | VARCHAR(255) | NOT NULL | The natural key from the source system. |
| product\_name | VARCHAR(255) | NOT NULL | The name of the product. |
| category | VARCHAR(100) | NOT NULL | The product category (e.g., Electronics, Apparel). |
| supplier\_id | VARCHAR(100) |  | The identifier of the supplier. |
| supplier\_name | VARCHAR(255) |  | The name of the supplier. |
| unit\_price | DECIMAL(10, 2) | NOT NULL | The current or historical price of the product. |
| valid\_from | TIMESTAMP | NOT NULL | The start date/time when this version became active. |
| valid\_to | TIMESTAMP | NOT NULL | The end date/time when this version expired. |
| current\_flag | CHAR(1) | NOT NULL | 'Y' if this is the current version, 'N' otherwise. |

### 2.2.2. dim\_customer

The dim\_customer table stores information about each unique customer. For this model, it is treated as a Type 1 SCD, meaning that if a customer's details change, the existing record is simply updated. This is a simplification that assumes historical changes in customer attributes are not critical for the primary analytical use cases. If this requirement changes, the dimension can be converted to a Type 2 SCD.

Table: dim\_customer

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** | **Description** |
| customer\_key | BIGINT | PRIMARY KEY | Surrogate key for the customer. |
| customer\_id | VARCHAR(255) | UNIQUE, NOT NULL | The natural key from the source system. |
| customer\_name | VARCHAR(255) |  | The name of the customer. |

### 2.2.3. dim\_time

The dim\_time table is a conformed dimension that provides a rich set of time-based attributes. It is pre-populated with rows for every hour, allowing both sales transactions and inventory snapshots to link to it. This enables powerful time-series analysis and aggregation.

Table: dim\_time

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** | **Description** |
| time\_key | BIGINT | PRIMARY KEY | Surrogate key for the time record. |
| full\_timestamp | TIMESTAMP | UNIQUE, NOT NULL | The full date and time. |
| date | DATE | NOT NULL | The calendar date. |
| hour | INT | NOT NULL | The hour of the day (0-23). |
| day\_of\_week | VARCHAR(10) | NOT NULL | The name of the day (e.g., Monday). |
| month | VARCHAR(10) | NOT NULL | The name of the month (e.g., January). |
| quarter | INT | NOT NULL | The quarter of the year (1-4). |
| year | INT | NOT NULL | The calendar year. |

### 2.2.4. dim\_warehouse

The dim\_warehouse table is a simple dimension that stores information about each warehouse where inventory is held. It is linked to the fact\_inventory table to provide context for stock levels.

Table: dim\_warehouse

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** | **Description** |
| warehouse\_key | BIGINT | PRIMARY KEY | Surrogate key for the warehouse. |
| warehouse\_id | VARCHAR(100) | UNIQUE, NOT NULL | The natural key from the source system. |
| warehouse\_name | VARCHAR(255) | NOT NULL | The name or location of the warehouse. |

### 2.2.5. dim\_location

The dim\_location table is created to handle the geographical information from the sales transactions. It stores unique combinations of city, state, and zip code, allowing for geographical analysis of sales performance.

Table: dim\_location

|  |  |  |  |
| --- | --- | --- | --- |
| **Column Name** | **Data Type** | **Constraints** | **Description** |
| location\_key | BIGINT | PRIMARY KEY | Surrogate key for the location. |
| city | VARCHAR(100) | NOT NULL | The city of the shipping address. |
| state | VARCHAR(100) | NOT NULL | The state of the shipping address. |
| zip\_code | VARCHAR(20) | NOT NULL | The zip code of the shipping address. |
| **Constraint** |  | UNIQUE (city, state, zip\_code) | Ensures each location is stored only once. |

# 3. SQL DDL Statements

The following SQL Data Definition Language (DDL) statements are provided to create the tables, keys, and constraints for the e-commerce data warehouse. These statements are written for a standard SQL-compliant database and may require minor adjustments for specific database systems.

## 3.1. Dimension Table Creation

This section contains the CREATE TABLE statements for all dimension tables in the star schema.

### 3.1.1. CREATE TABLE dim\_product

This statement creates the dim\_product table, implementing the SCD Type 2 logic with valid\_from , valid\_to , and current\_flag columns.

1. CREATE TABLE dim\_product (

2. product\_key BIGINT PRIMARY KEY AUTO\_INCREMENT,

3. product\_id VARCHAR(255) NOT NULL,

4. product\_name VARCHAR(255) NOT NULL,

5. category VARCHAR(100) NOT NULL,

6. supplier\_id VARCHAR(100),

7. supplier\_name VARCHAR(255),

8. unit\_price DECIMAL(10, 2) NOT NULL,

9. valid\_from TIMESTAMP NOT NULL,

10. valid\_to TIMESTAMP NOT NULL,

11. current\_flag CHAR(1) NOT NULL CHECK (current\_flag IN ('Y', 'N')),

12. INDEX idx\_product\_natural\_key (product\_id),

13. INDEX idx\_product\_current (product\_id, current\_flag),

14. INDEX idx\_product\_validity (valid\_from, valid\_to)

15. );

16.

### 3.1.2. CREATE TABLE dim\_customer

This statement creates the dim\_customer table, which stores unique customer records.

1. CREATE TABLE dim\_customer (

2. customer\_key BIGINT PRIMARY KEY AUTO\_INCREMENT,

3. customer\_id VARCHAR(255) UNIQUE NOT NULL,

4. customer\_name VARCHAR(255)

5. );

6.

### 3.1.3. CREATE TABLE dim\_time

This statement creates the dim\_time table, a conformed dimension for time-based analysis.

1. CREATE TABLE dim\_time (

2. time\_key BIGINT PRIMARY KEY AUTO\_INCREMENT,

3. full\_timestamp TIMESTAMP UNIQUE NOT NULL,

4. date DATE NOT NULL,

5. hour INT NOT NULL,

6. day\_of\_week VARCHAR(10) NOT NULL,

7. month VARCHAR(10) NOT NULL,

8. quarter INT NOT NULL,

9. year INT NOT NULL,

10. INDEX idx\_time\_date (date),

11. INDEX idx\_time\_month (year, month),

12. INDEX idx\_time\_quarter (year, quarter)

13. );

14.

### 3.1.4. CREATE TABLE dim\_warehouse

This statement creates the dim\_warehouse table, storing information about inventory locations.

1. CREATE TABLE dim\_warehouse (

2. warehouse\_id BIGINT PRIMARY KEY AUTO\_INCREMENT,

3. warehouse\_id VARCHAR(100) UNIQUE NOT NULL,

4. warehouse\_name VARCHAR(255) NOT NULL

5. );

6.

### 3.1.5. CREATE TABLE dim\_location

This statement creates the dim\_location table to store unique geographical locations from shipping addresses.

1. CREATE TABLE dim\_location (

2. location\_id BIGINT PRIMARY KEY AUTO\_INCREMENT,

3. city VARCHAR(100) NOT NULL,

4. state VARCHAR(100) NOT NULL,

5. zip\_code VARCHAR(20) NOT NULL,

6. UNIQUE (city, state, zip\_code)

7. );

8.

### 3.2. Fact Table Creation

This section contains the CREATE TABLE statements for the fact tables.

### 3.2.1. CREATE TABLE fact\_sales

This statement creates the fact\_sales table, which stores transactional sales data.

1. CREATE TABLE fact\_sales (

2. sales\_key BIGINT PRIMARY KEY AUTO\_INCREMENT,

3. product\_key BIGINT NOT NULL,

4. customer\_key BIGINT NOT NULL,

5. time\_key BIGINT NOT NULL,

6. location\_key BIGINT NOT NULL,

7. order\_id VARCHAR(255) NOT NULL,

8. quantity\_sold INT NOT NULL,

9. sale\_price\_per\_unit DECIMAL(10, 2) NOT NULL,

10. INDEX idx\_sales\_product (product\_key),

11. INDEX idx\_sales\_customer (customer\_key),

12. INDEX idx\_sales\_time (time\_key),

13. INDEX idx\_sales\_location (location\_key)

14. );

15.

### 3.2.2. CREATE TABLE fact\_inventory

This statement creates the fact\_inventory table, which stores hourly inventory snapshots.

1. CREATE TABLE fact\_inventory (

2. inventory\_key BIGINT PRIMARY KEY AUTO\_INCREMENT,

3. product\_key BIGINT NOT NULL,

4. warehouse\_key BIGINT NOT NULL,

5. time\_key BIGINT NOT NULL,

6. stock\_quantity INT NOT NULL,

7. INDEX idx\_inventory\_product (product\_key),

8. INDEX idx\_inventory\_warehouse (warehouse\_key),

9. INDEX idx\_inventory\_time (time\_key)

10. );

11.

### 3.3. Foreign Key Constraints

This section adds the foreign key constraints to the fact tables to enforce referential integrity.

### 3.3.1. ALTER TABLE statements for foreign keys

These ALTER TABLE statements formally link the fact tables to their corresponding

dimension tables.

Foreign keys for fact\_sales

1. ALTER TABLE fact\_sales

2. ADD CONSTRAINT fk\_sales\_product

3. FOREIGN KEY (product\_key) REFERENCES

4. dim\_product(product\_key);

5. ALTER TABLE fact\_sales

6. ADD CONSTRAINT fk\_sales\_customer

7. FOREIGN KEY (customer\_key) REFERENCES dim\_customer(customer\_key);

8. ALTER TABLE fact\_sales

9. ADD CONSTRAINT fk\_sales\_time

10. FOREIGN KEY (time\_key) REFERENCES dim\_time(time\_key);

11. ALTER TABLE fact\_sales

12. ADD CONSTRAINT fk\_sales\_location

13. FOREIGN KEY (location\_key) REFERENCES dim\_location(location\_key);

Foreign keys for fact\_inventory

1. ALTER TABLE fact\_inventory

2. ADD CONSTRAINT fk\_inventory\_product

3. FOREIGN KEY (product\_key) REFERENCES

4. dim\_product(product\_key);

5. ALTER TABLE fact\_inventory

6. ADD CONSTRAINT fk\_inventory\_warehouse

7. FOREIGN KEY (warehouse\_key) REFERENCES dim\_warehouse(warehouse\_key);

8. ALTER TABLE fact\_inventory

9. ADD CONSTRAINT fk\_inventory\_time

10. FOREIGN KEY (time\_key) REFERENCES dim\_time(time\_key);

# 4. Data Processing and Business Query Support

The designed data model is not just a static structure; it is built to actively support key business processes and analytical queries. The choices made in the design, particularly the use of SCD Type 2 for products and the separation of fact tables with different granularities, are directly aimed at enabling accurate and insightful analysis. This section details how the model handles the complexities of historical data and varying data frequencies, and how it can be used to answer specific business questions.

## 4.1. Analyzing Historical Sales with Product Prices

A core requirement of the data warehouse is the ability to analyze historical sales performance using the product prices that were active at the time of each sale. This is a classic challenge in data warehousing, and the chosen SCD Type 2 implementation for the dim\_product table is the key to solving it. Without this mechanism, any change in a product's price would retroactively alter historical revenue calculations, leading to inaccurate and misleading reports. The model ensures that each sales transaction is permanently linked to the specific state of the product dimension as it existed at that moment in time.

### 4.1.1. Linking Sales to the Correct Product Version

The mechanism for linking a sales transaction to the correct version of a product is embedded in the ETL process and the table structure. When a sales event occurs, the ETL process does not simply use the product\_id to join to the dim\_product table. Instead, it performs a more sophisticated lookup. It takes the product\_id from the sales transaction and the order\_timestamp , and it queries the dim\_product table to find the specific record where product\_id matches and the order\_timestamp falls between the valid\_from and valid\_to dates. The product\_key of this specific record is then used as the foreign key value in the fact\_sales table. This process effectively "freezes" the product's state at the time of the sale, creating an immutable link between the fact and the dimension. This ensures that even if the product's price or category changes a hundred times in the future, the historical sales record will always point to the correct version of the product, preserving the integrity of historical analysis.

### 4.1.2. Querying Revenue with Historical Pricing

With the SCD Type 2 structure in place, querying historical revenue with the correct pricing becomes straightforward. An analyst does not need to write complex queries to manually find the right price for a given date. The join between fact\_sales and dim\_product on product\_key automatically retrieves the product attributes, including the unit\_price , that were in effect at the time of the sale. For example, to calculate the total revenue for a specific product category last month, the query would join fact\_sales to dim\_product on product\_key , filter the sales by the date range from dim\_time , and filter the products by the desired category from dim\_product . The aggregation SUM(quantity\_sold \* sale\_price\_per\_unit) would then correctly use the historical prices associated with each transaction. This capability is fundamental for accurate financial reporting, trend analysis, and understanding the impact of pricing strategies over time.

### 4.2. Handling Different Data Granularities

The data warehouse ingests information from multiple sources, each with a different update frequency. Sales transactions are streamed in real-time, while inventory levels are provided as hourly snapshots. The model is designed to accommodate these different granularities by using separate fact tables, each with a grain that matches its source data. This approach avoids the complexities and potential for data loss or misrepresentation that would arise from trying to force all data into a single, uniform grain.

### 4.2.1. Real-time Sales Data (Transactional)

The fact\_sales table is designed with a transactional grain: one row per line item in an order. This matches the high-frequency, event-driven nature of the sales data source. Each row represents a single, atomic business event. This fine-grained detail is crucial for flexibility. It allows analysts to drill down to the most detailed level of a transaction if needed, while also supporting aggregation to any higher level, such as daily, weekly, or monthly sales. The real-time nature of this data means that the fact\_sales table is constantly growing as new sales are made, providing an up-to-the-minute view of the company's sales performance. The order\_timestamp from the source data is used to link each transaction to the dim\_time dimension, enabling precise time-based analysis.

### 4.2.2. Hourly Inventory Snapshots (Periodic)

In contrast to the transactional sales data, the inventory data is provided as a periodic snapshot. The fact\_inventory table has a grain of one row per product, per warehouse, per hour. This means that every hour, a new set of records is added to the table, capturing the stock quantity for every product in every warehouse at that specific moment. This design is ideal for tracking levels and states over time. It allows analysts to see how inventory levels have trended, identify when stock-outs occurred, and calculate metrics like average stock on hand. The snapshot\_timestamp from the source data is used to link to the dim\_time dimension. By separating this periodic data into its own fact table, the model avoids the issue of having to join a high-frequency transactional table with a lower-frequency snapshot table, which could lead to complex queries and incorrect results.

### 4.3. Enabling Key Business Queries

The ultimate test of a data warehouse model is its ability to answer real-world business questions efficiently and accurately. The proposed star schema is specifically designed to support the key analytical queries outlined in the project requirements. The clear separation of facts and dimensions, combined with the SCD Type 2 handling of product history, provides a robust foundation for these analyses.

### 4.3.1. Query: "Total revenue by product category last month"

This query requires aggregating sales data, filtering by a time period, and grouping by a product attribute. The star schema makes this query intuitive and efficient.

SQL Query:

**SELECT**  
 dp.category,  
 **SUM**(fs.quantity\_sold \* fs.sale\_price\_per\_unit) **AS** total\_revenue  
**FROM**  
 fact\_sales fs  
**JOIN**  
 dim\_product dp **ON** fs.product\_key = dp.product\_key  
**JOIN**  
 dim\_time dt **ON** fs.time\_key = dt.time\_key  
**WHERE**  
 dt.year = **YEAR**(**CURRENT\_DATE** - INTERVAL 1 **MONTH**)  
 **AND** dt.month = MONTHNAME(**CURRENT\_DATE** - INTERVAL 1 **MONTH**)  
**GROUP** **BY**  
 dp.category  
**ORDER** **BY**  
 total\_revenue **DESC**;

How the model supports this query:

* fact\_sales : Provides the core transactional data ( quantity\_sold , sale\_price\_per\_unit ).
* dim\_product : Provides the category attribute for grouping. The SCD Type 2 structure ensures that the revenue is calculated using the historical price associated with each transaction.
* dim\_time : Provides the year and month attributes for filtering the data to "last month".
* Performance: The query involves simple joins on indexed surrogate keys, which is highly efficient in a star schema.

### 4.3.2. Query: "Current inventory level for our top 10 selling products"

This query is more complex, as it requires identifying the top-selling products from the sales fact table and then retrieving their current inventory levels from the inventory fact table.

SQL Query:

**WITH** top\_selling\_products **AS** (  
 **SELECT**  
 fs.product\_key,  
 **SUM**(fs.quantity\_sold) **AS** total\_quantity\_sold  
 **FROM**  
 fact\_sales fs  
 **GROUP** **BY**  
 fs.product\_key  
 **ORDER** **BY**  
 total\_quantity\_sold **DESC**  
 **LIMIT** 10  
),  
latest\_inventory **AS** (  
 **SELECT**  
 fi.product\_key,  
 fi.warehouse\_id,  
 fi.stock\_quantity,  
 **RANK**() **OVER** (**PARTITION** **BY** fi.product\_key, fi.warehouse\_id   
**ORDER** **BY** fi.time\_key **DESC**) **as** rnk  
 **FROM**  
 fact\_inventory fi  
)  
**SELECT**  
 dp.product\_name,  
 dw.warehouse\_name,  
 li.stock\_quantity **AS** current\_inventory  
**FROM**  
 top\_selling\_products tsp  
**JOIN**  
 latest\_inventory li **ON** tsp.product\_key = li.product\_key  
**JOIN**  
 dim\_product dp **ON** tsp.product\_key = dp.product\_key **AND**   
dp.current\_flag = 'Y'  
**JOIN**  
 dim\_warehouse dw **ON** li.warehouse\_id = dw.warehouse\_id  
**WHERE**  
 li.rnk = 1  
  
**ORDER** **BY**  
 tsp.total\_quantity\_sold **DESC**;

How the model supports this query:

* fact\_sales : Used in the top\_selling\_products CTE to aggregate total sales quantity per product.
* fact\_inventory : Used in the latest\_inventory CTE to find the most recent inventory snapshot for each product/warehouse combination using a window function ( RANK() ).
* dim\_product : Joined on product\_key to get the product\_name . The current\_flag = 'Y' filter is used to ensure we get the current name of the product.
* dim\_warehouse : Joined to get the warehouse\_name for context.
* Complexity Handling: The use of Common Table Expressions (CTEs) breaks down the complex logic into manageable steps, first identifying the top products and then finding their latest inventory levels before joining the results together.

# 5. Design Trade-offs and Indexing Strategy

The design of any data warehouse involves a series of trade-offs between competing goals such as query performance, storage efficiency, ETL complexity, and data accuracy. The choices made in this model reflect a deliberate balance to best serve the analytical needs of the e-commerce platform. An effective indexing strategy is also crucial to realizing the performance benefits of the star schema.

## 5.1. Modeling Trade-offs Considered

Several key trade-offs were evaluated during the design process. The final decisions were guided by the primary objective of creating a system that is optimized for fast, accurate, and intuitive business intelligence.

|  |  |  |  |
| --- | --- | --- | --- |
| **Design Decision** | **Option A (Chosen)** | **Option B (Alternative)** | **Rationale for Choice** |
| **SCD Type** | **SCD Type 2** (Full History) | SCD Type 1 (Overwrite) | **Historical Accuracy** is paramount for analyzing sales performance and pricing strategies. The complexity of managing versions is a necessary trade-off for accurate reporting. |
| **History Management** | **Integrated SCD Type 2** | Separate History/Audit Tables | **Simplicity and Query Performance**. Encapsulating history within the dimension table provides a cleaner model and avoids complex multi-table joins for historical analysis. |
| **Schema Type** | **Star Schema** (Denormalized) | Snowflake Schema (Normalized) | **Query Performance and Usability**. Fewer joins lead to faster queries. The slight increase in storage is a worthwhile trade-off for speed and a model that is easier for analysts to understand. |
| **Core Trade-off** | **Complexity for Accuracy** | Simplicity for Inaccuracy | The primary goal is a **reliable source of truth for historical analysis**. A more complex backend is an acceptable price for delivering accurate and trustworthy business insights. |

### 5.1.1. SCD Type 2 vs. SCD Type 1

The most significant trade-off was in the handling of the dim\_product table. The choice was between SCD Type 2 (track full history by creating new rows) and SCD Type 1 (overwrite old values with new ones).

* SCD Type 1 Trade-off: Using SCD Type 1 would have dramatically simplified the ETL process. There would be no need to manage version numbers, validity dates, or current flags. The ETL would simply be an "upsert" (update or insert) operation. However, the cost of this simplicity is the complete loss of historical data. Any change in unit\_price would overwrite the previous price, making it impossible to accurately calculate historical revenue or analyze the impact of price changes. For an e-commerce business where pricing is dynamic and historical analysis is critical, this trade-off was deemed unacceptable.
* SCD Type 2 Decision: SCD Type 2 was chosen because the value of preserving historical accuracy far outweighs the increased complexity in the ETL pipeline and the additional storage required for multiple versions of product records. This decision ensures that the data warehouse can be a reliable source of truth for historical performance analysis.

### 5.1.2. SCD Type 2 vs. Separate History/Audit Tables

An alternative to SCD Type 2 is to keep the main dimension table as Type 1 (always current) and maintain a separate, fully normalized audit or history table that records every change as a new row.

* Separate History Table Trade-off: This approach would keep the main dim\_product table small and simple, which could be beneficial for queries that only need the current state of products. The historical analysis would then require joining the fact table to the main dimension table and then to the separate history table to find the correct state at a point in time. This adds complexity to the queries and can negatively impact performance.
* SCD Type 2 Decision: The integrated SCD Type 2 approach was preferred because it encapsulates the historical logic within the dimension table itself. This provides a cleaner and more intuitive model for analysts. The logic for retrieving the correct historical version is handled by a simple join on the surrogate key, which is more efficient than the multi-table joins required by the separate history table approach. This aligns with the Kimball principle of prioritizing ease of use and query performance for analytical workloads.

### 5.1.3. Star Schema vs. Snowflake Schema

As discussed earlier, the choice between a star and a snowflake schema was a key architectural decision.

* Snowflake Schema Trade-off: A snowflake schema would reduce data redundancy by normalizing dimension tables (e.g., creating a separate dim\_category table linked from dim\_product ). This can lead to minor storage savings. However, it introduces more joins into analytical queries, which can significantly slow down performance and make the model harder for business users to understand.
* Star Schema Decision: The star schema was chosen to optimize for query speed and simplicity. The denormalized dimensions mean fewer joins, which is a critical factor for performance in large-scale analytical databases. The slight increase in storage is a worthwhile trade-off for the significant gains in query efficiency and user-friendliness, especially given the low cost of modern cloud storage.

### 5.1.4. Complexity vs. Historical Accuracy

Across all these decisions, the overarching trade-off is between system complexity and historical accuracy.

* Simpler, Less Accurate Model: A model using SCD Type 1 and a snowflake schema would be simpler to build and maintain from an ETL perspective. However, it would fail to deliver accurate historical insights, which is a primary goal of the data warehouse.
* More Complex, More Accurate Model: The chosen model, with its SCD Type 2 dimensions and star schema, is more complex to build. The ETL logic for handling SCD Type 2 is more involved, and the denormalized dimensions require more storage. However, this complexity is a necessary investment to achieve the core business objective of providing a reliable and accurate platform for historical analysis. The model prioritizes the accuracy and integrity of the analytical results over the simplicity of the backend implementation.

## 5.2. Recommended Indexing Strategy

A well-designed indexing strategy is essential to ensure that the data warehouse can deliver fast query performance. The strategy should focus on supporting the most common and critical query patterns, which in a star schema typically involve joining fact tables to dimension tables and filtering on dimensional attributes.

### 5.2.1. Indexing on Fact Tables

The primary goal of indexing on fact tables is to speed up the joins to dimension tables and to support filtering on foreign keys.

* Foreign Key Indexes: Every foreign key column in the fact tables ( fact\_sales and fact\_inventory ) should have a non-clustered index. This is the most critical index for star schema performance, as it accelerates the joins between the large fact tables and the smaller dimension tables.
  + fact\_sales : Indexes on product\_key , customer\_key , time\_key , and location\_id .
  + fact\_inventory : Indexes on product\_key , warehouse\_id , and time\_key .
* Composite Indexes: For queries that frequently filter on multiple dimensions, composite indexes can be beneficial. For example, an index on (time\_key, product\_key) in fact\_sales could speed up queries that analyze sales for a specific product over a period of time.

### 5.2.2. Indexing on Dimension Tables

Indexing on dimension tables is primarily focused on supporting filtering and grouping operations on their descriptive attributes.

* Surrogate Key: The primary key of each dimension table (e.g., product\_key in dim\_product ) is already indexed by virtue of being a primary key.
* Natural Key: The natural key from the source system (e.g., product\_id in dim\_product ) should have a unique index. This is important for the ETL process to efficiently look up existing records.
* Filtering Attributes: Columns that are frequently used in WHERE clauses or GROUP BY clauses should be indexed. For example, in dim\_product , an index on category would speed up queries that filter or group by product category. In dim\_time , indexes on date , month , and year are essential for time-based filtering.

### 5.2.3. Indexing for SCD Type 2 Queries

The SCD Type 2 structure of dim\_product requires special consideration for indexing to support efficient historical lookups.

* Current Record Lookup: Queries that need the current state of a product will often filter on current\_flag = 'Y' . A composite index on (product\_id, current\_flag) can make this lookup very efficient.
* Historical Record Lookup: Queries that need to find the version of a product that was active at a specific point in time will filter on product\_id and a date range between valid\_from and valid\_to . A composite index on (product\_id, valid\_from, valid\_to) is crucial for the performance of these types of queries. This allows the database to quickly locate the specific version of a product that was valid at any given timestamp.