# Data Warehousing: A Technical Tutorial with Real-World Implementation

## Origin Story: The Pain Points and Challenges

In the early days of data management, businesses relied on transactional databases (OLTP - Online Transaction Processing) to store and manage day-to-day operations. While these systems were efficient for handling individual transactions, they faced significant challenges when it came to **analytical processing**:

1. **Performance Issues**: OLTP systems are optimized for write heavy operations, but complex analytical queries (e.g., aggregations, joins across large datasets) slowed down the system.
2. **Data Silos**: Data was scattered across multiple systems (e.g., sales, marketing, inventory), making it difficult to get a unified view.
3. **Inconsistent Data**: Different systems often stored data in different formats, leading to inconsistencies and errors in reporting.
4. **Historical Data**: OLTP systems typically focus on current data, making it hard to analyse trends over time.
5. **Resource Contention**: Running analytical queries on operational systems impacted the performance of critical business processes.

These challenges led to the development of **Data Warehousing**; a specialized system designed for **OLAP (Online Analytical Processing)**. Data warehouses consolidate data from multiple sources, transform it into a consistent format, and optimize it for analytical queries.

## The Real-World Problem

Imagine a retail company, **Global Mart**, struggling with its data infrastructure. They have the following issues:

* **Fragmented Data**: Sales data is stored in one system, inventory in another, and customer data in yet another.
* **Slow Reporting**: Generating monthly sales reports takes hours, delaying decision-making.
* **Inconsistent Metrics**: Marketing and sales teams use different definitions for "revenue," leading to confusion.
* **No Historical Analysis**: The company cannot analyse trends over time because historical data is either deleted or inaccessible.
* **Scalability Issues**: As the company grows, the existing systems cannot handle the increasing data volume.

Global Mart hires a **Data Engineer cum Architect** (you) to design and implement a data warehousing solution.

## The Solution: Implementing a Data Warehouse

**Step 1: Understanding Data Warehousing**

A **Data Warehouse (DW)** is a centralized repository that stores integrated data from multiple sources. It is designed for:

* **Querying and Analysis**: Optimized for complex analytical queries.
* **Historical Data**: Stores large volumes of historical data for trend analysis.
* **Data Integration**: Combines data from disparate sources into a unified format.
* **Decision Support**: Provides actionable insights for business decision-making.
* Four key characteristics of a Data Warehouse need to keep in mind.
* **Subject-Oriented**: Data is grouped based on business needs (e.g., Finance, HR, Sales).
* **Integrated**: Data from various sources is cleaned and standardized.
* **Time-Variant**: Historical data is stored with timestamps to analyse trends.
* **Non-Volatile**: Data is never deleted, only appended for accurate reporting.

## Core Concepts

* **OLAP vs. OLTP vs Spark**:
  + **OLTP**: Optimized for transactional processing (e.g., inserting a new order).
  + **OLAP**: Optimized for analytical processing (e.g., calculating total sales by region).
  + **Spark**

| **Feature** | **OLTP (Operational Databases)** | **Data Warehouse (OLAP)** | **Spark + Data Lake** |
| --- | --- | --- | --- |
| **Primary Purpose** | Fast transactions, CRUD operations | Analytical processing, historical data analysis | Large-scale data processing, transformations, and AI/ML |
| **Data Type** | Highly structured (Relational Tables) | Structured (SQL Tables) | Structured, Semi-Structured, Unstructured |
| **Schema** | **Schema-on-write** (Strict) | **Schema-on-write** (Enforced, pre-defined) | **Schema-on-read** (Flexible, applied at query time) |
| **Processing** | **Row-based operations**, small transactions (INSERT, UPDATE, DELETE) | **Columnar-based operations**, Aggregations, Complex Joins, Reporting | **Batch Processing, Streaming, Machine Learning** |
| **Storage Format** | **Row-based storage (B-Trees, Heaps, Hash Indexes)** | **Columnar storage (Parquet, ORC)** | **Mixed (JSON, Parquet, CSV, Avro, Images, Video, Delta Lake)** |
| **Performance Focus** | **Optimized for high-speed transactions** (millisecond latency) | **Optimized for aggregations & queries** (seconds to minutes latency) | **Optimized for parallel processing** (ETL, AI/ML workloads) |
| **Latency** | **Very low** (milliseconds) | **Moderate** (seconds to minutes) | **Higher for batch, lower for streaming** |
| **Concurrency** | Supports **thousands** of concurrent transactions | Supports **multiple concurrent query users** | Supports **large-scale distributed parallel processing** |
| **Normalization** | **Highly normalized (3NF, BCNF)** to reduce redundancy | **Denormalized** (Star, Snowflake Schema) for faster queries | Flexible schema (may store raw, normalized, or denormalized data) |
| **Scalability** | **Vertical scaling** (limited horizontal scaling with replication/sharding) | **Horizontal scaling** (MPP systems) | **Massive horizontal scaling** (Distributed processing across clusters) |
| **Streaming** | **Not built for streaming** | **Not built for real-time ingestion** | **Supports real-time streaming (Kafka, Spark Streaming, Flink)** |
| **ACID Compliance** | **Strict ACID transactions** for data consistency | Supports **ACID transactions** (but less stringent than OLTP) | **Eventual consistency** (with Delta Lake, Iceberg, ACID can be enforced) |
| **Parallel Processing** | **Not optimized** for parallel query execution | **Optimized for parallel query execution (MPP, indexing)** | **Massively parallel processing (Spark RDDs, DAG execution)** |
| **Cost Efficiency** | **Expensive** due to transaction-heavy workloads | **Moderate to expensive**, depends on query complexity | **Low-cost storage, high compute cost** (pay-as-you-go model in cloud) |
| **Typical Use Cases** | Banking, E-commerce orders, User logins | Business intelligence, Reporting, Data Analytics | Big Data processing, ETL, Machine Learning, Data Science |
| **Examples** | **MySQL, PostgreSQL, Oracle, SQL Server** | **Snowflake, Redshift, BigQuery, Teradata** | **Databricks, AWS Glue, Apache Iceberg, Delta Lake** |

**Key Takeaways:**

* **OLTP** is for **high-speed transactions** (INSERT, UPDATE, DELETE) and is highly **normalized**.
* **Data Warehouses (OLAP)** are for **analytical queries** on structured data and use **denormalized** schemas for performance.
* **Spark + Data Lakes** handle **unstructured & semi-structured data**, support **real-time and batch processing**, and are designed for **Big Data & AI workloads**.
* **Star Schema vs. Snowflake Schema**

**Scenario: Retail Company (Global Mart)**

**Let’s assume Global Mart has the following data:**

* **Sales Data: Transactional data (e.g., sale ID, product ID, customer ID, date, quantity, revenue).**
* **Product Data: Product details (e.g., product ID, product name, category, price).**
* **Customer Data: Customer details (e.g., customer ID, name, address, city, state).**
* **Time Data: Date details (e.g., date, month, year, quarter).**

**Star Schema**

**In a star schema, the data is organized into:**

* **One Fact Table: Central table containing transactional data.**
* **Multiple Dimension Tables: Surrounding tables containing descriptive attributes.**

**Fact Table: Sales\_Fact**

| **Sale\_ID** | **Product\_ID** | **Customer\_ID** | **Date\_ID** | **Quantity** | **Revenue** |
| --- | --- | --- | --- | --- | --- |
| **1** | **101** | **201** | **20230101** | **2** | **100** |
| **2** | **102** | **202** | **20230102** | **1** | **50** |

**Dimension Tables**

1. **Product\_Dim**

| **Product\_ID** | **Product\_Name** | **Category** | **Price** |
| --- | --- | --- | --- |
| **101** | **Laptop** | **Electronics** | **1000** |
| **102** | **Smartphone** | **Electronics** | **500** |

1. **Customer\_Dim**

| **Customer\_ID** | **Name** | **City** | **State** |
| --- | --- | --- | --- |
| **201** | **John** | **New York** | **NY** |
| **202** | **Alice** | **Los Angeles** | **CA** |

1. **Time\_Dim**

| **Date\_ID** | **Month** | **Year** | **Quarter** |
| --- | --- | --- | --- |
| **20230101** | **Jan** | **2023** | **Q1** |
| **20230102** | **Jan** | **2023** | **Q1** |

**Snowflake Schema**

**In a snowflake schema, the dimension tables are normalized, meaning they are split into sub-dimensions.**

**Fact Table: Sales\_Fact (Same as Star Schema)**

**Dimension Tables**

1. **Product\_Dim**

| **Product\_ID** | **Product\_Name** | **Category\_ID** |
| --- | --- | --- |
| **101** | **Laptop** | **1** |
| **102** | **Smartphone** | **1** |

1. **Category\_Dim**

| **Category\_ID** | **Category\_Name** |
| --- | --- |
| **1** | **Electronics** |

1. **Customer\_Dim**

| **Customer\_ID** | **Name** | **City\_ID** |
| --- | --- | --- |
| **201** | **John** | **1** |
| **202** | **Alice** | **2** |

1. **City\_Dim**

| **City\_ID** | **City** | **State\_ID** |
| --- | --- | --- |
| **1** | **New York** | **1** |
| **2** | **Los Angeles** | **2** |

1. **State\_Dim**

| **State\_ID** | **State** |
| --- | --- |
| **1** | **NY** |
| **2** | **CA** |

1. **Time\_Dim (Same as Star Schema)**

**Key Differences**

* **Star Schema: Simple, denormalized, faster queries.**
* **Snowflake Schema: Normalized, reduces redundancy, more complex queries.**

**SCD Types**

* **SCD 1 (Overwrite Data, No History): Always updates the record with the latest value. No history is maintained.**

| **ID** | **Name** | **City** |
| --- | --- | --- |
| **1** | **UserA** | **D** |

**Explanation:**

* **Every time UserA moves to a new city, we simply overwrite the existing record.**
* **At the end of the journey (A → B → C → D), only D is present in the table.**
* **No way to track where UserA lived before.**
* **SCD 2 (Store Full History with Versioning): A new row is inserted each time a change happens. A flag or effective date tracks the latest record.**

| **ID** | **Name** | **City** | **Start\_Date** | **End\_Date** | **Is\_Current** |
| --- | --- | --- | --- | --- | --- |
| **1** | **UserA** | **A** | **2024-01-01** | **2024-06-01** | **N** |
| **2** | **UserA** | **B** | **2024-06-02** | **2024-12-01** | **N** |
| **3** | **UserA** | **C** | **2024-12-02** | **2025-02-01** | **N** |
| **4** | **UserA** | **D** | **2025-02-02** | **NULL** | **Y** |

**Explanation:**

* **Each time a change happens, a new row is inserted.**
* **Is\_Current helps in identifying the active record (D in this case).**
* **Start\_Date and End\_Date define when each record was valid.**
* **Querying history is straightforward.**
* **SCD 6: Keeps both full history (like SCD 2) and limited history (like SCD 3) within the same table.**

| **ID** | **Name** | **City** | **Previous\_City** | **Start\_Date** | **End\_Date** | **Is\_Current** |
| --- | --- | --- | --- | --- | --- | --- |
| **1** | **UserA** | **A** | **NULL** | **2024-01-01** | **2024-06-01** | **N** |
| **2** | **UserA** | **B** | **A** | **2024-06-02** | **2024-12-01** | **N** |
| **3** | **UserA** | **C** | **B** | **2024-12-02** | **2025-02-01** | **N** |
| **4** | **UserA** | **D** | **C** | **2025-02-02** | **NULL** | **Y** |

**Explanation:**

* **Like SCD 2, it keeps full history in multiple rows.**
* **Like SCD 3, it tracks a Previous\_City column for easy reference.**
* **It allows both historical queries (like SCD 2) and quick comparisons (like SCD 3).**
  + **Comparing All Four Types**

| **SCD Type** | **Historical Data?** | **Current Data?** | **Tracking Method** |
| --- | --- | --- | --- |
| **SCD 1** | **❌ No history** | **✅ Latest only** | **Overwrites previous data** |
| **SCD 2** | **✅ Full history** | **✅ Latest** | **New row per change, with date tracking** |
| **SCD 4** | **✅ Full history (separate table)** | **✅ Latest only** | **Historical table stores changes separately** |
| **SCD 6** | **✅ Full history** | **✅ Latest** | **Mix of SCD 2 (full history) + SCD 3 (limited history)** |

* **ETL vs. ELT vs. ELTL**

**ETL (Extract, Transform, Load)**

1. **Extract: Pull data from sources (e.g., MySQL, API).**
2. **Transform: Clean, join, and aggregate data using a transformation tool (e.g., Talend, Informatica).**
3. **Load: Load transformed data into the target system (e.g., Redshift).**

**Example:**

* **Extract sales data from MySQL.**
* **Transform data to calculate total revenue by product.**
* **Load transformed data into Redshift.**

**ELT (Extract, Load, Transform)**

1. **Extract: Pull data from sources.**
2. **Load: Load raw data into the target system (e.g., Redshift).**
3. **Transform: Transform data within the target system using SQL or tools like dbt.**

**Example:**

* **Extract sales data from MySQL.**
* **Load raw data into Redshift.**
* **Use SQL to calculate total revenue by product in Redshift.**

**ELTL (Extract, Load, Transform, Load)**

1. **Extract: Pull data from sources.**
2. **Load: Load raw data into a staging area (e.g., S3).**
3. **Transform: Transform data in the staging area using tools like Spark.**
4. **Load: Load transformed data into the target system (e.g., Redshift).**

**Example:**

* **Extract sales data from MySQL.**
* **Load raw data into S3.**
* **Use Spark to calculate total revenue by product.**
* **Load transformed data into Redshift.**

## Designing the Data Warehouse

You analyse Global Mart’s requirements and design a data warehouse with the following components:

1. **Data Sources**:
   * Sales Database (OLTP)
   * Inventory System
   * Customer Relationship Management (CRM) System
   * Marketing Platform
2. **ETL Pipeline**:
   * Extract data from all sources.
   * Transform data into a consistent format (e.g., standardize date formats, resolve duplicates).
   * Load data into the data warehouse.
3. **Data Warehouse Schema**:
   * Use a **Star Schema** for simplicity and performance.
   * **Fact Table**: Sales Fact (stores transactional data like sales amount, quantity).
   * **Dimension Tables**:
     + Product\_Dim (product details).
     + Customer\_Dim (customer details).
     + Time\_Dim (date and time details for trend analysis).
4. **Data Marts**:
   * Create separate data marts for Sales, Inventory, and Marketing teams.

## Technical Explanations

**1. ETL (Extract, Transform, Load)**

* **Extract**: Data is pulled from multiple sources (e.g., databases, APIs, flat files).
* **Transform**: Data is cleaned, standardized, and enriched (e.g., adding derived columns).
* **Load**: Data is loaded into the data warehouse for analysis.

**2. Star Schema**

* **Fact Table**: Stores measurable data (e.g., sales amount, quantity).
* **Dimension Tables**: Store descriptive data (e.g., product details, customer details).

**3. OLAP Operations**

* **Slice and Dice**: Filter data based on specific criteria.
* **Roll-Up**: Aggregate data at higher levels (e.g., daily → monthly sales).
* **Drill-Down**: Explore data at a more granular level (e.g., yearly → monthly sales).

**Best Practices**

1. **Data Quality**: Ensure data is clean, consistent, and accurate.
2. **Scalability**: Choose a data warehouse that can handle growing data volumes.
3. **Indexing**: Use indexes to speed up query performance.
4. **Partitioning**: Partition large tables by date or region for faster queries.
5. **Security**: Implement access controls and encryption to protect sensitive data.

**Structured Breakdown**

1. **Requirement Analysis**: Understand the business needs and data sources.
2. **Schema Design**: Design a star schema for the data warehouse.
3. **ETL Pipeline**: Build an ETL pipeline to extract, transform, and load data.
4. **Data Population**: Load sample data into the data warehouse.
5. **Querying**: Write analytical queries to generate insights.
6. **Optimization**: Monitor and optimize the data warehouse for performance.

# Data Mart

## Understanding

* A **subset** of a Data Warehouse that is designed for a specific business function (Finance, Sales, HR, etc.).
* It allows **faster querying** and is optimized for a particular use case.
* Can be **independent** (built directly from the source) or **dependent** (built from the Data Warehouse).
* **Redshift Hierarchy and Where Data Marts Fit**

📌 **Redshift (Data Warehouse System)** → **Database** → **Schema** → **Tables**

**✅ Where Does a Data Mart Exist?**

🔹 **A Data Mart is NOT a physical object but rather a logical concept**.

* It can be a **set of tables inside a schema** in the same database.
* It can be a **separate schema dedicated to a specific domain (e.g., finance, marketing, etc.).**
* It can also be a **separate Redshift database altogether**.

**🔵 Independent vs. Dependent Data Marts in Redshift**

| **Type** | **Definition** | **How is Data Pulled?** |
| --- | --- | --- |
| **Independent Data Mart** | Created directly from the data lake or transactional system without relying on a central Data Warehouse. | Data is extracted from **S3 (Parquet/ORC/CSV), RDS, DynamoDB**, etc. into a dedicated schema. |
| **Dependent Data Mart** | Created from a Data Warehouse by selecting relevant data for a department. | Data is derived from **main Data Warehouse tables** using transformations (ETL or ELT). |

**How to Structure Dependent Data Marts in Redshift?**

Since a **Data Mart is a subset of a Data Warehouse**, the question arises:

**How do we create Data Marts from existing Redshift tables?**

**✅ Options to Implement a Data Mart in Redshift**

| **Method** | **Description** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **1. Regular Table (ETL)** | Create a physical table with selected data using INSERT INTO datamart\_table SELECT ... | Faster queries, indexing possible | Data duplication, needs ETL to refresh |
| **2. Materialized View (ELT)** | Use CREATE MATERIALIZED VIEW AS SELECT ... to store precomputed results | Faster than regular views, auto-refresh possible | Requires manual refresh if not automated |
| **3. Regular View (ELT)** | Use CREATE VIEW AS SELECT ... (virtual table, no storage) | No storage overhead | Queries run in real-time, can be slow |
| **4. Schema Separation** | Create a separate **schema** for different Data Marts | Logical separation, better organization | Requires access control and management |

**How Does a Data Mart Get Updated?**

* **For Regular Tables**: Needs a scheduled ETL job (e.g., using **AWS Glue, Airflow, dbt**)
* **For Materialized Views**: Use REFRESH MATERIALIZED VIEW to update periodically
* **For Regular Views**: Auto-updates when the base table changes

## **When to Use a Data Mart?**

| **Scenario** | **Need for Data Mart?** | **Why?** |
| --- | --- | --- |
| 🔍 Queries on only a **small subset** of Data Warehouse | ✅ Yes | Avoid scanning full Data Warehouse for faster queries |
| 🏎️ Slow query performance in Data Warehouse | ✅ Yes | Optimize performance by storing pre-processed data |
| 🔐 Restricted access to sensitive data | ✅ Yes | Enforce **security & compliance** by isolating data |
| 🏢 Different departments need **specific views** | ✅ Yes | Avoid giving full access to the central Data Warehouse |
| 💰 Reducing **Redshift/Snowflake costs** | ✅ Yes | Minimize storage and query costs by pre-aggregating data |
| 📊 Business teams need **fast reporting** | ✅ Yes | Store KPIs in a precomputed form |
| 🛠️ Business-specific transformations needed | ✅ Yes | Apply domain-specific rules for each department |

## **🔹 How Data Marts Fit into Data Architecture?**

### **🔥 Centralized Data Warehouse Model (Traditional)**

🔹 **Single Data Warehouse** → All departments query the same database.  
🔹 Pros: Centralized governance, data consistency  
🔹 Cons: Performance issues, complex queries, access control challenges

### **🔥 Data Warehouse + Dependent Data Marts**

🔹 **Data Warehouse** → **Finance Data Mart**, **Marketing Data Mart**, **HR Data Mart**  
🔹 Pros: Faster queries, reduced load on Data Warehouse, controlled access  
🔹 Cons: Need ETL jobs to maintain Data Marts

### **🔥 Independent Data Marts (Decentralized)**

🔹 **Independent Data Marts** without a central Data Warehouse  
🔹 Pros: Each department controls its own data  
🔹 Cons: Possible data duplication, harder governance