Section 1: Introduction

Data is nothing but raw and unprocessed facts and statistics stored or free-flowing over a network. Data becomes **information** when it is processed, turning it into something meaningful. Collecting and storing data for analysis is a human activity and we have been doing it for thousands of years. In order to effectively work with huge amounts of data in an organized and efficient manner, databases were invented.

> *A **Database** is a collection of related data/information:*
>
> - ***organized** in a way that data can be easily accessed, managed, and updated.*
> - ***captures** information about an organization or organizational process*
> - ***supports** the enterprise by continually updating the database*

In database management, there are two main types of databases: operational databases and analytical databases.

Operational databases are primarily used in **online transactional processing (OLTP)** scenarios, where there is a need to collect, modify, and maintain data daily. This type of database stores dynamic data that changes constantly, reflecting up-to-the-minute information.

On the other hand, analytical databases are used in **online analytical processing (OLAP)** scenarios to store and track historical and time-dependent data. They are valuable for tracking trends, analyzing statistical data over time, and making business projections. Analytical databases store static data that is rarely modified, reflecting a point-in-time snapshot. Although analytical databases often use data from operational databases as their main source, they serve specific data processing needs and require different design methodologies.

	OLTP	OLAP
Characteristics	Handles a large number of small transactions with simple queries	Handles large volumes of data with complex queries
Operations	Based on INSERT, UPDATE, and DELETE commands	Based on SELECT commands to aggregate data for reporting
Response time	Milliseconds	Seconds, minutes, or hours depending on the amount of data to process
Source		Aggregated data from transactions **Multiple** data sources Dimension tables can be connected from SQL Database, CSV files, and more for analysis purposes

	OLTP	OLAP
Purpose	Control and run essential business operations in real-time	Plan, solve problems, support decisions, discover hidden insights
Data updates	Short, fast updates initiated by the user	Data periodically refreshed with scheduled, long-running batch jobs
Space requirements	Generally small if historical data is archived	Generally large due to aggregating large datasets
Backup and recovery	Regular backups are required to ensure business continuity	Lost data can be reloaded from the OLTP database as needed
Data view	Lists day-to-day business transactions	Multi-dimensional view of enterprise data
User examples	Customer-facing personnel, clerks, online shoppers	Knowledge workers such as data analysts, business analysts, and executives
Database design	**Normalization** concept applies and architecture is generally **SNOWFLAKE** schema	Strict adherence to normalization is not followed, **STAR** schema is followed (i.e. one or more dimension tables from SQL Database can be merged and/or appended to get a single dimension table in SQL Data Warehouse)

Section 2: Intro to Data Warehousing

What is a Data Warehouse?

A data warehouse is a large, centralized repository of data that is used to support data-driven decision-making and business intelligence (BI) activities. It is designed to provide a single, comprehensive view of all the data in an organization, and to allow users to easily analyze and report on that data.

Data warehouses are typically used to store historical data and are optimized for fast query and analysis performance. They often contain data from multiple sources and may include both structured and unstructured data. Data in a data warehouse is imported from operational systems and external sources, rather than being created within the warehouse itself. Importantly, data is **copied** into the warehouse, not **moved**, so it remains in the source systems as well.

Data warehouses follow a set of rules proposed by Bill Inmon in 1990. These rules are:

1. Integrated: They combine data from different source systems into a unified environment.
2. Subject-oriented: Data is reorganized by subjects, making it easier to analyze specific topics or areas of interest.
3. Time-variant: They store historical data, not just current data, allowing for trend analysis and tracking changes over time.
4. Non-volatile: Data warehouses remain stable between refreshes, with new and updated data loaded periodically in batches. This ensures that the data does not change during analysis, allowing for consistent strategic planning and decision-making.

As data is imported into the data warehouse, it is often restructured and reorganized to make it more useful for analysis. This process helps to optimize the data for querying and reporting, making it easier for users to extract valuable insights from the data.

Why do we need a Data Warehouse?

Primary reasons for investing time, resources, and money into building a data warehouse:

1. Data-driven decision-making: Data warehouses enable organizations to make decisions based on data, rather than solely relying on experience, intuition, or hunches.
2. One-stop shopping: A data warehouse consolidates data from various transactional and operational applications into a single location, making it easier to access and analyze the data.

Data warehouses provide a comprehensive view of an organization's past, present, and potential future/forecast data. They also offer insights into unknown patterns or trends through advanced analytics and Business Intelligence (BI). In conclusion, Business Intelligence and data warehousing are closely related disciplines that provide immense value to organizations by facilitating data-driven decision-making and offering a centralized data repository for analysis.

Data Warehouse vs Data Lake

Let's discuss the similarities and differences between data warehouses and data lakes, two valuable tools in data management.

A data warehouse is often built on top of a relational database, such as Microsoft SQL Server, Oracle, or IBM DB2. These databases are used for both transactional systems and data warehousing, making them versatile data management tools. Sometimes, data warehouses are built on multidimensional databases called "cubes," which are specialized databases.

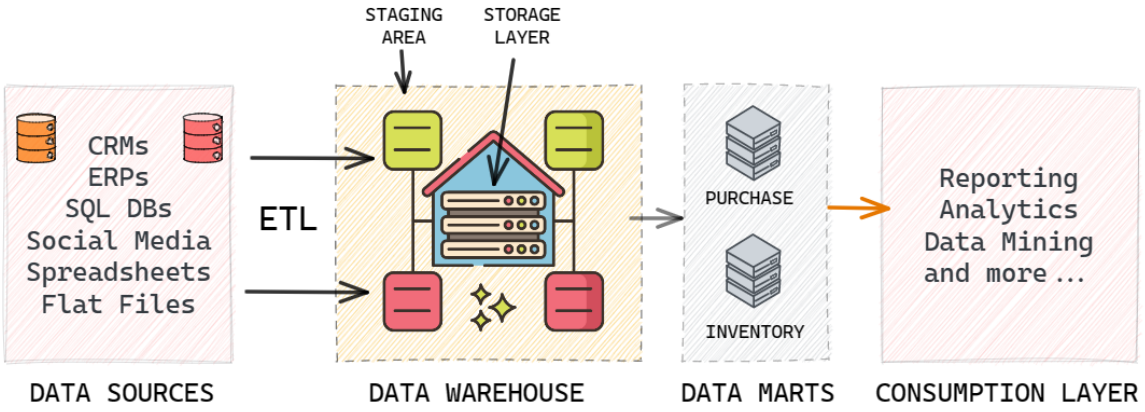
In contrast, a data lake is built on top of a big data environment rather than a traditional relational database. Big data environments allow for the management of extremely large volumes of data, rapid intake of new and changing data, and support a variety of data types (structured, semi-structured, and unstructured).

The lines between the two are increasingly blurred, as SQL, the standard relational database language, can be used on both data warehouses and data lakes. From a user perspective, traditional Business Intelligence (BI) can be performed against either a data warehouse or a data lake.

Parameters	Data Lake	Data Warehouse
Storage	All data is kept irrespective of the source and its structure in its raw form.	Processed data is stored that is extracted, cleaned, and transformed from transactional systems
History	Big data technologies used in data lakes	A data warehouse is often built on top of a relational database
Users	A data lake is ideal for users who indulge in deep analysis. Such users include data scientists who need advanced analytical tools with capabilities such as predictive modelling and statistical analysis.	The data warehouse is ideal for operational users because of being well structured, easy to use, and understandable.
Storage Costs	Data storing in big data technologies are relatively inexpensive than storing data in a data warehouse.	Storing data in a Data warehouse is costlier and more time-consuming.
Task	Data lakes can contain all data and data types; it empowers users to access data prior to the process of transforming and cleansing.	Data warehouses can provide insights into pre-defined questions for pre-defined data types.
Position of Schema	Typically, the schema is defined after data is stored. This offers high agility and ease of data capture but requires work at the end of the process	Typically schema is defined before data is stored. Requires work at the start of the process, but offers performance, security, and integration.
Data processing	Data Lakes use the ELT (Extract Load Transform) process.	Data warehouse uses a traditional ETL (Extract Transform Load) process.

Simple End-to-End Data Warehouse Environment

A simple end-to-end data warehousing environment consists of data sources, a data warehouse, and sometimes, smaller environments called data marts. The process connecting data sources to the data warehouse is known as ETL (Extract, Transform, and Load), a critical aspect of data warehousing.



An analogy to understand this relationship is to think of data sources as suppliers, the data warehouse as a wholesaler that collects data from various suppliers, and data marts as data retailers. Data marts store specific subsets of data tailored for different user groups or business functions. Users typically access data from these data marts for their data-driven decision-making processes.

Section 3: Data Warehouse Architecture

Introduction to DW Architecture Components

In data warehousing, there are various architectural options to plan and execute initiatives. These options include:

1. **Staging layer**: This is the segment within the data warehouse where data is initially loaded before it is transformed and fully integrated for reporting and analytical purposes. There are two types of staging layers: persistent and non-persistent.
2. **Data marts**: These are smaller-scale or more narrowly focused data warehouses.
3. **Cube**: A specialized type of database that can be part of the data warehousing environment.
4. **Centralized data warehouse:** This approach uses a single database to store data for business intelligence and analytics.
5. **Component-based data warehousing**: This approach consists of multiple components, such as data warehouses and data marts, that operate together to create an overall data warehousing environment.

Staging layer in a Data Warehouse

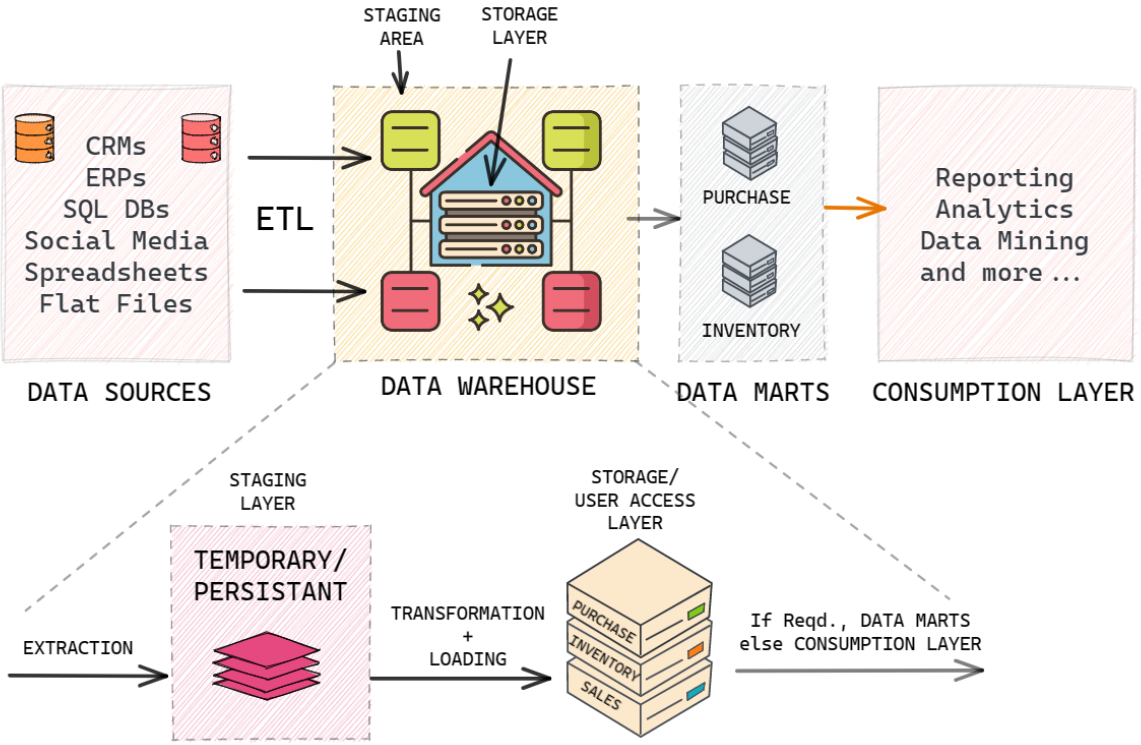
A data warehouse consists of two side-by-side layers: the staging layer and the user access layer. The staging layer serves as a landing zone for incoming data from source applications. The main objective is to quickly and non-intrusively copy the needed data from source applications into the data warehousing environment.

The user access layer is where users interact with the data warehouse or data mart. This layer deals with design, engineering, and dimensional modelling, such as star schema, snowflake schema, fact tables, and dimension tables.

There are two types of staging layers in a data warehouse: non-persistent staging layers and persistent staging layers. Both serve as landing zones for incoming data, which is then transformed, integrated, and loaded into the user access layer.

In a non-persistent staging layer, the data is temporary. After being loaded into the user access layer, the staging layer is emptied. This requires less storage space but makes it more difficult to rebuild the user layer or perform data quality assurance without accessing the original source systems.

In contrast, a persistent staging layer retains data even after it has been loaded into the user access layer. This enables easy rebuilding of the user layer and data quality assurance without accessing the source systems. However, it requires more storage space and can lead to ungoverned access by power users.

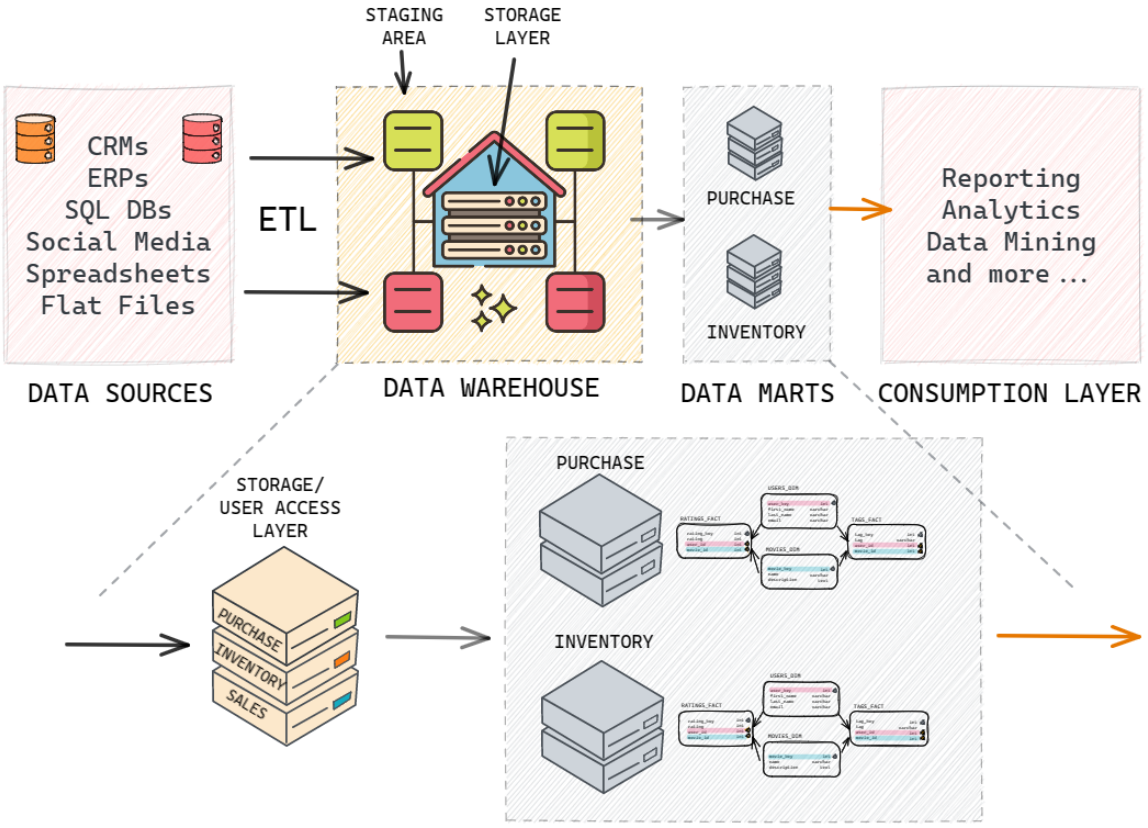


Data Warehouse vs Data Mart

Data marts are additional environments in the data warehousing process, often thought of as data retailers. There are two types of data marts: dependent and independent.

Dependent data marts rely on the existence of a data warehouse to be supplied with data. Independent data marts, on the other hand, draw data directly from one or more source applications and do not require a data warehouse. Independent data marts can be thought of as small-scale data warehouses with data organized internally.

The difference between a data warehouse and an independent data mart lies in the number of data sources and business requirements. Data warehouses typically have many sources (~10-50), while independent data marts have much fewer data sources. If a business requirement dictates the creation of well-defined business units due to the huge size of a data warehouse, data marts can be added to the architecture to create units specific to business requirements such as purchase-specific data mart, inventory-specific data mart, etc. Other than that, the properties of a data warehouse and an independent data mart are quite similar.

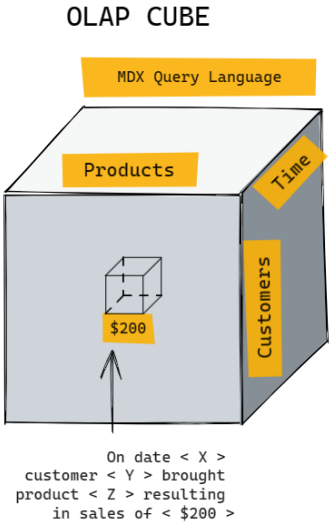


Cubes in Data Warehouse environment

Unlike relational database management systems (RDBMS), cubes are specialized databases with an inherent awareness of the dimensionality of their data contents. Cubes offer some advantages, such as fast query response times and suitability for modest data volumes (around 100 GB or less).

However, they are less flexible compared to RDBMS, as they have a rigid organizational structure for data. Structural changes to the data can be complex and time-consuming. Additionally, there is more vendor variation with cubes than with RDBMS.

In modern data warehousing environments, it is common to see a combination of relational databases and multi-dimensional databases. This mix-and-match approach can provide a powerful combination for organizations.



Figuring out the appropriate approach to DW

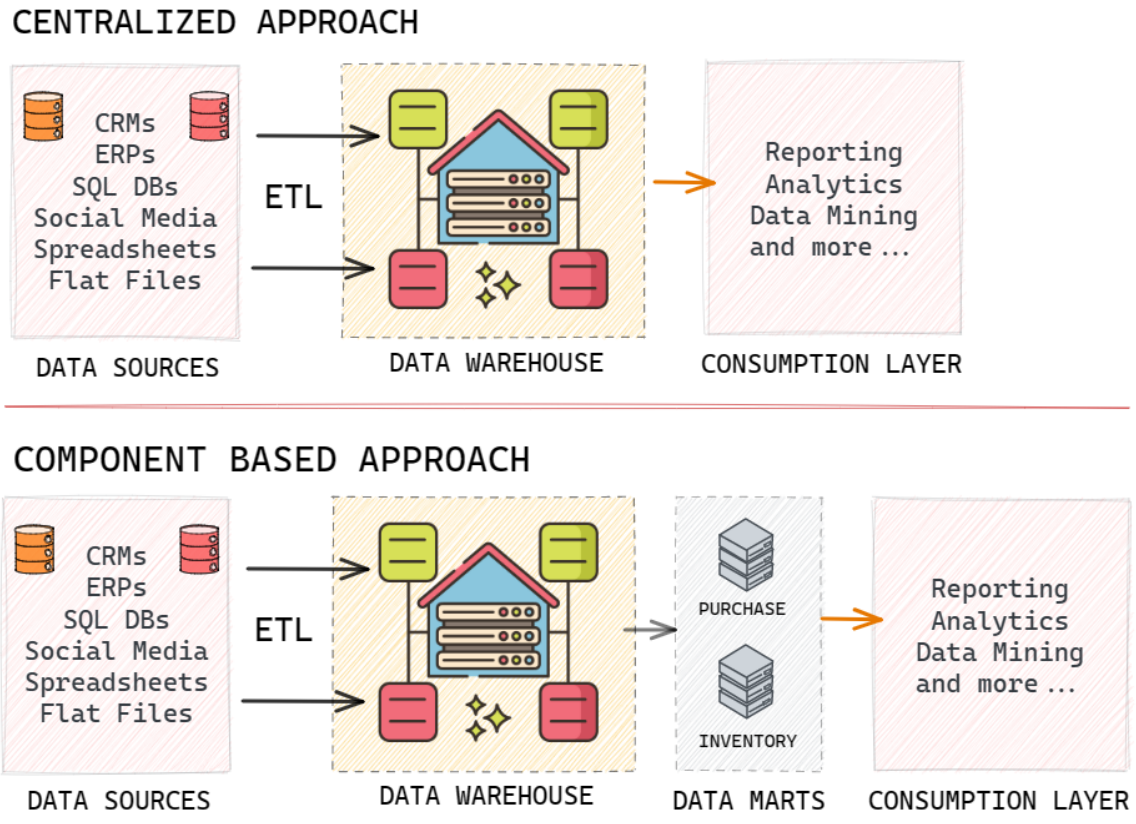
In summary, there are two main paths to choose from when deciding on a data warehouse environment: a centralized approach or a component-based approach.

A centralized approach, such as an Enterprise Data Warehouse (EDW) or a Data Lake, offers a single, unified environment for data storage and analysis. This enables one-stop shopping for all data needs but requires a high degree of cross-

organizational cooperation and strong data governance. It also carries the risk of ripple effects when changes are made to the environment.

A component-based approach, on the other hand, divides the data environment into multiple components such as data warehouses and data marts. This offers benefits like isolation from changes in other parts of the environment and the ability to mix and match technology. However, it can lead to inconsistent data across components and difficulties in cross-integration.

The choice of data warehouse environment ultimately depends on the specific needs and realities of each organization, and the decision-making process required.



Section 4: Data Warehousing Design: Building

The primary objective of a data warehouse is to enable data-driven decisions, which can involve past, present, future, and unknown aspects through different types of business intelligence and analytics. **Dimensional modelling** is particularly suited for basic reporting and online analytical processing (OLAP). However, if the data warehouse is primarily supporting predictive or exploratory analytics, different data models may be required. In such cases, only some aspects of the data may be structured dimensionally, while others will be built into forms suitable for those types of analytics.

The Basic Principles of Dimensionality

Dimensionality in data warehousing refers to the **organization of data in a structured way** that facilitates efficient querying and analysis. The two main components of dimensionality are **facts/measurements** and **dimensional** context, which are essential for understanding the data and making data-driven decisions.

The main benefits of using a dimensional approach in data warehousing include:

- **Improved query performance**: By organizing data in a structured manner, queries can be executed more efficiently, enabling faster data retrieval and analysis.
- **Simplified data analysis**: Dimensional models make it easier for analysts to understand the relationships between data elements and perform complex analyses.
- **Enhanced data visualization**: Organized data enables better visualization of patterns, trends, and relationships, which in turn helps in making informed decisions.

Facts/Measurements:

These are the quantifiable aspects of the data, such as sales amounts, user counts, or product quantities. They represent the actual values or metrics that are being analyzed. It is important to differentiate between a data warehousing fact and a logical fact, as the former is a numeric value while the latter is a statement that can be evaluated as true or false.

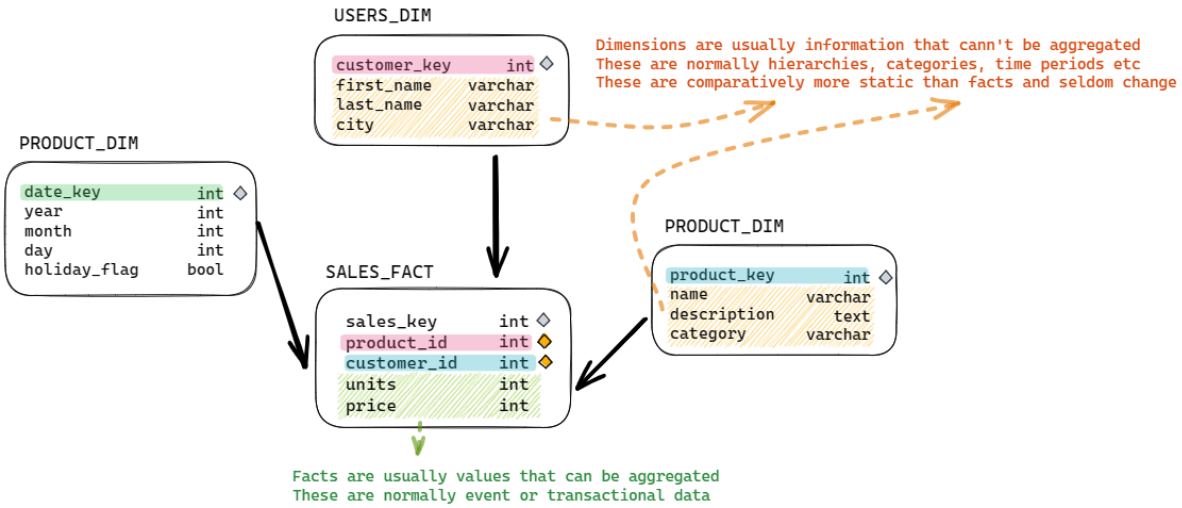
Fact tables store facts in a relational database used for a data warehouse. They are distinct from facts themselves, and there are specific rules for determining which facts can be stored in the same fact table. Fact tables can be of different types and can be connected with dimension tables to link measurements and context in the data warehouse.

Dimensional context:

This is the descriptive information that provides context to the measurements, helping to understand how the data is organized and what it represents. Dimensions often include categories, hierarchies, or time periods, which enable data analysts to slice and dice the data in various ways. Common dimensions include geography (e.g., country, region, city), time (e.g., year, quarter, month), and product categories (e.g., electronics, clothing, food).

In a relational database, dimensions are stored in dimension tables. However, the terms "dimension" and "dimension table" are sometimes used interchangeably, although they are technically different.

This interchangeability of terms is due to the differences between star and snowflake schemas. In a star schema, all information about multiple levels of a hierarchy (e.g., product, product family, and product category) is stored in a single-dimension table, which is often called the **product dimension table**. In a snowflake schema, each level of the hierarchy is stored in a separate table, resulting in **three dimensions and three dimension tables**.



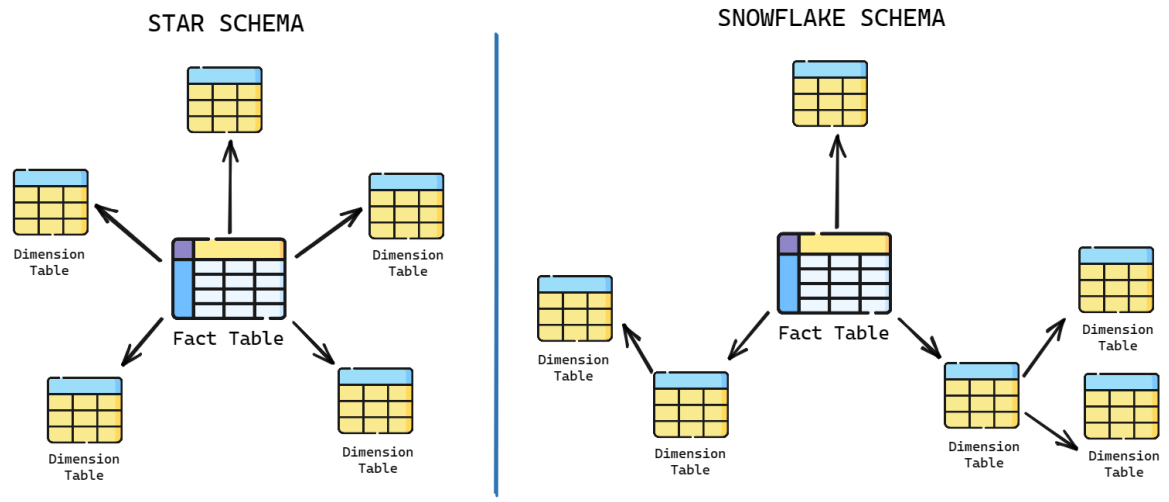
Star schema and snowflake schema are two popular approaches to dimensional modelling. Both involve organizing data into fact tables (which store measurements) and dimension tables (which store contextual information).

Star Schema vs Snowflake Schema

Star schema and snowflake schema are two different structural approaches to building dimensional models in a data warehouse. Both are used to represent fact tables and dimension tables, which are essential for dimensional analysis in OLAP

and business intelligence.

Star Schema	Snowflake Schema
All dimensions along a given hierarchy in one dimension table	Each dimension/dimensional table in its own table
One level away from the fact table thus requiring straightforward relationships	One or more levels away from the fact table thus requiring complex relationships
Fewer database joins at the cost of more data storage requirements	More database joins while consuming much less storage



Both star and snowflake schemas have the same number of dimensions, but their table representations differ. The choice between star and snowflake schema depends on factors such as the complexity of the database relationships, storage requirements, and the number of joins needed for data analysis.

Database Keys for Data Warehousing

Database keys play a crucial role in data warehousing. They help maintain relationships, ensure data integrity, and improve query performance.

1. **Primary Keys:** A primary key is a unique identifier for each row in a database table. It can be a single column or a combination of columns that guarantee the uniqueness of each record. Primary keys are essential for maintaining data integrity, preventing duplicate entries, and providing a unique reference point for other tables.
2. **Foreign Keys:** A foreign key is a column or set of columns that reference the primary key of another table. The purpose of foreign keys is to maintain the relationships between tables and ensure referential integrity. This means that a foreign key value must always match an existing primary key value in the related table or be NULL. Foreign keys help with data consistency, reduce redundancy, and make it easier to enforce constraints and retrieve related information.

Natural and Surrogate Keys

1. **Natural Keys:** Natural keys (also known as business keys or domain keys) are derived from the source system data and are used to identify a record based on its inherent attributes. They can be cryptic or easily understandable, depending on the nature of the data. Natural keys have their limitations, as they can be subject to change, and sometimes they may not guarantee uniqueness. However, keeping natural keys in dimension tables as secondary keys can be beneficial for traceability, data validation, and troubleshooting purposes.

   For example, in a table storing employee details, the employee's social security number could serve as a natural key. Similarly, a customer's email address could be a natural key in a customers table (assuming each customer has a unique email address)

2. **Surrogate Keys:** Surrogate keys are system-generated, unique identifiers that have no business meaning. They are used as primary keys in data warehousing to ensure data integrity and simplify the relationships between tables. Surrogate keys are usually auto-incremented numbers. Surrogate keys are preferred over natural keys for several reasons, including the ability to handle changes in the source data, improved performance, and the support for slowly changing dimensions.

   For example, consider a table storing customer details. Even though each customer might have a unique email address, we might choose to use a surrogate key, such as CustomerID, to uniquely identify customers. So, a customer might be

assigned an ID like 1, 2, 3, and so on, with no relation to the actual data about the customer.

Comparison:

- **Meaning**: Natural keys have a business meaning, while surrogate keys do not.
- **Change**: Natural keys can change (for example, a person might change their email address), while surrogate keys are static and do not change once assigned.
- **Simplicity**: Natural keys can be complex if they're composed of multiple attributes, while surrogate keys are simple (usually just a number).
- **Performance**: Surrogate keys can improve query performance because they're usually indexed and simpler to manage.
- **Data Anonymity**: Surrogate keys provide a level of abstraction and data protection, particularly useful when sharing data without exposing sensitive information.

When designing a data warehouse, it is crucial to make the right choices regarding key usage. As a general guideline, use surrogate keys as primary and foreign keys for better data integrity and handling of data changes. Keep natural keys in dimension tables as secondary keys to maintain traceability and ease troubleshooting. Finally, discard or do not use natural keys in fact tables, as they can lead to redundancy and complexity in managing the data warehouse.

Section 5: Design Facts, Fact Tables, Dimensions, and Dimension Tables

Different Forms of Additivity in Facts

Additivity is an important concept in data warehousing that pertains to the rules for storing facts in fact tables. Facts can be additive, non-additive, or semi-additive.

Additive facts can be added under all circumstances. For example, faculty members' salaries or students' credit hours can be added together to find a total salary or total credit hours completed. These are valid uses of addition in the context of additive facts.

Non-additive facts, on the other hand, cannot be added together to produce a valid result. Examples include grade point averages (GPA), ratios, or percentages. To handle non-additive facts, it's best to store the underlying components in fact tables rather than the ratio or average itself. You can store the non-additive fact at an individual row level for easy access but should prevent users from adding them up. Instead, you can calculate aggregate averages or ratios from the totals of the underlying components.

Semi-additive facts fall somewhere between additive and non-additive facts, as they can sometimes be added together while at other times they cannot. Semi-additive facts are often used in periodic snapshot fact tables, which will be covered in more detail in future discussions.

NULLs in Facts

A **NULL** represents a **missing** or **unknown** value; **Note** that a Null **does not** represent a zero, a character string of one or more blank spaces, or a “zero-length” character string.

The major drawback of Nulls is their adverse effect on mathematical operations. Any operation involving a Null evaluates to Null. This is logically reasonable—if a number is unknown, then the result of the operation is necessarily unknown

```
SELECT 5 + NULL; -- Result: NULL
SELECT 10 - NULL; -- Result: NULL
SELECT 3 * NULL; -- Result: NULL
SELECT 15 / NULL; -- Result: NULL
```

```
SELECT 8 % NULL; -- Result: NULL

-- Assume a table 'sales' with a column 'revenue' containing the following values:
--(10, NULL, 15)
SELECT SUM(revenue) FROM sales; -- Result: 25 (ignores NULL values)
SELECT AVG(revenue) FROM sales; -- Result: 12.5 (ignores NULL values)


SELECT COUNT(revenue) FROM sales; -- Result: 2 (ignores NULL values)
SELECT COUNT(*) FROM sales; -- Result: 3 (includes NULL values)
```

To handle NULL values in mathematical operations in MySQL, you can use functions like COALESCE or IFNULL to replace NULL values with a default value. Here's an example:

```
-- Replace NULL values with 0 and perform addition
SELECT 5 + COALESCE(NULL, 0); -- Result: 5
SELECT 5 + IFNULL(NULL, 0); -- Result: 5
```

NULLs also should be avoided in the foreign key column of fact tables. Instead of using NULLs, we can assign default values to columns in the fact table. we can choose a value that represents the absence of data or a not applicable scenario. However, even though this approach can simplify querying and analysis but it may introduce ambiguity if the default value can be mistaken for actual data.

The Four Main Types of Data Warehousing Fact Tables

In data warehousing, there are four main types of fact tables used to store different types of facts or measurements.

1. **Transaction Fact Table:** This type of fact table records facts or measurements from transactions occurring in source systems. It manages these transactions at an appropriate level of detail in the data warehouse.
2. **Periodic Snapshot Fact Table:** This table tracks specific measurements at regular intervals. It focuses on periodic readings rather than lower-level transactions.
3. **Accumulating Snapshot Fact Table:** Similar to the periodic snapshot fact table, this table shows a snapshot at a point in time. However, it is specifically used to track the progress of a well-defined business process through various stages.
4. **Factless Fact Table:** This type of fact table has two main uses. Firstly, it records the occurrence of a transaction even if there are no measurements to record. Secondly, it is used to officially document coverage or eligibility relationships, even if nothing occurred as part of that particular relationship.

The Role of Transaction Fact Tables

Transactional fact tables are the most common type of fact table used in a data warehouse. These fact tables store transaction-level data, meaning that each row in the table represents a single event or transaction. Transactional fact tables are useful for capturing and analyzing detailed, granular data about the individual events that occur in a business. When dealing with transaction fact tables, we can store multiple facts together in the same table if they meet the following rules:

1. Facts are available at the same grain or level of detail.
2. Facts occur simultaneously.

In a transactional fact table, each row is associated with one or more dimension tables. These dimension tables help provide context and additional information about the transaction. Facts within the transactional fact table are usually numeric and additive, such as sales amount, quantity sold, or total revenue.

Example:

Consider a retail store that wants to track sales transactions. In this example, the transactional fact table could be called 'sales_fact'. For each sale transaction, the following information might be stored:

- Date of sale (Date Dimension)
- Store location (Store Dimension)
- Product sold (Product Dimension)
- Customer information (Customer Dimension)
- Payment method (Payment Method Dimension)
- Quantity sold (Fact)

- Sales amount (Fact)
- Discount applied (Fact)

Here is a simplified version of what the 'sales_fact' table might look like:

transaction_id (PK)	date_key (FK)	store_key (FK)	product_key (FK)	customer_key (FK)	payment_key (FK)	quantity (fact)	sales_amount (fact)
1	1	1	1	1	1	2	50
2	1	2	2	2	2	1	25
3	1	1	3	1	1	3	75

In this example, the transactional fact table contains three fact columns (quantity, sales_amount, and discount) and five foreign key columns that connect to the respective dimension tables (date_key, store_key, product_key, customer_key, and payment_key).

Remember to use surrogate keys (not natural keys) as primary and foreign keys to maintain relationships between tables. These surrogate keys serve as foreign keys that point to primary keys in the respective dimension tables, which provide the context needed for decision-making based on the data.

The Role of Periodic Snapshot Fact Tables

Periodic Snapshot fact tables are used in data warehouses to capture the state of a particular measure at specific points in time, such as daily, weekly, or monthly snapshots. These fact tables help analyze trends and changes over time, which can be valuable for understanding the business's performance and making informed decisions.

In a periodic snapshot fact table, each row represents the state of a specific measure at a particular point in time. The table typically includes one or more dimension keys that provide context for the snapshot, such as date, product, or customer information.

Example:

Consider a bank that wants to track its customers' account balances on a monthly basis. In this example, the periodic snapshot fact table could be called 'monthly_account_balance_fact'. For each customer, the following information might be stored:

1. Snapshot Date (Date Dimension)
2. Customer information (Customer Dimension)
3. Account type (Account Type Dimension)
4. Account balance (Fact)

Here is a simplified version of what the 'monthly_account_balance_fact' table might look like:

snapshot_id	date_key	customer_key	account_type_key	account_balance
1	1	1	1	1000
2	1	2	1	2500
3	1	1	2	5000
4	2	1	1	1200
5	2	2	1	2300
6	2	1	2	5100

In this example, the periodic snapshot fact table contains one fact column (account_balance) and three foreign key columns that connect to the respective dimension tables (date_key, customer_key, and account_type_key). Each row represents the state of a customer's account balance at the end of a specific month.

Using a periodic snapshot fact table, the bank can analyze its customers' account balances in various ways, such as:

1. Average account balance per month
2. Changes in account balances over time
3. Distribution of account balances by account type
4. Customer segments based on account balances

Periodic snapshot fact tables are particularly useful for tracking and analyzing data that changes over time and may not be easily captured in a transactional fact table. A complication with periodic snapshot fact tables is the presence of semi-additive facts.

Periodic Snapshots and Semi-Additive Facts

A semi-additive fact is a type of measure that can be aggregated along some (but not all) dimensions. Typically, semi-additive facts are used in periodic snapshot fact tables.

The classic example of a semi-additive fact is a bank account balance. If you have daily snapshots of account balances, you can't add the balances across the time dimension because it wouldn't make sense - adding Monday's balance to Tuesday's balance doesn't give you any meaningful information. However, you could add up these balances across the account dimension (if you were to aggregate for a household with multiple accounts, for instance) or across a geographical dimension (if you wanted to see total balances for a particular branch or region).

Here's an example:

Date	AccountID	Balance
2023-05-01	1	100
2023-05-01	2	200
2023-05-02	1	150
2023-05-02	2	250
2023-05-03	1	120
2023-05-03	2	240

If we were to add the balances across the time dimension (for example, to try to get a total balance for account 1 for the period from May 1 to May 3), we would get $370, which doesn't represent any meaningful quantity in this context. This is why the balance is a semi-additive fact - it can be added across some dimensions (like the AccountID), but not across others (like the Date).

In contrast, fully additive facts, like a bank deposit or withdrawal amount, can be summed along any dimension. Non-additive facts, like an interest rate, cannot be meaningfully summed along any dimension.

When dealing with semi-additive facts, it's important to ensure that your analysis is using the appropriate aggregation methods for each dimension. Depending on the specific requirements of your analysis, you might need to use the maximum, minimum, first, last, or average value of the semi-additive fact for a given period, rather than the sum.

The Role of Accumulating Snapshot Fact Tables

These tables are used to track the progress of a business process (e.g. order fulfillment) with formally defined stages, measuring elapsed time spent in each stage or phase. They typically include both completed phases and phases that are in progress. Accumulating snapshot fact tables also introduce the concept of multiple relationships from a fact table back to a single dimension table.

These typically include multiple date columns representing different milestones in the life cycle of the event and one or more dimension keys providing context for the snapshot, such as product, customer, or project information. The fact columns represent the measures associated with each stage or milestone.

Example:

Consider a company that wants to track its sales orders from order placement to delivery. In this example, the accumulating snapshot fact table could be called 'sales_order_fact'. For each sales order, the following information might be stored:

- Sales Order ID (Primary key)
- Customer information (Customer Dimension)
- Product Information (Product Dimension)
- Order date (Date Dimension)
- Shipping date (Date Dimension)
- Delivery date (Date Dimension)
- Order amount (Fact)
- Shipping cost (Fact)

Here is a simplified version of what the 'sales_order_fact' table might look like:

order_id	customer_key	product_key	order_date_key	shipping_date_key	delivery_date_key	order_amount	shipping_c
1	1	1	1	2	4	100	10
2	2	1	1	3	5	200	15
3	1	2	2	4	6	150	12

In this example, the accumulating snapshot fact table contains two fact columns (order_amount and shipping_cost) and five foreign key columns that connect to the respective dimension tables (customer_key, product_key, order_date_key, shipping_date_key, and delivery_date_key). Each row represents the life cycle of a sales order, from order placement to delivery.

Using an accumulating snapshot fact table, the company can analyze its sales order data in various ways, such as:

1. Average time between order placement and shipping
2. Average time between shipping and delivery
3. Total shipping cost by product or customer
4. Comparison of actual delivery dates to estimated delivery dates

In the accumulating snapshot fact table, there are multiple relationships with the same dimension table, such as the date dimension. This is because the table needs to track various dates related to the business process, like the order date, shipping date, delivery date, and so on. Similarly, multiple relationships with the employee dimension may also be required to account for different employees responsible for different phases of the process.

Comparing the three main types of fact tables

Type	Transactional	Periodic Snapshot	Accumulating Snapshot
Grain	1 row = 1 transaction	1 row =1 defined period (plus other dimensions)	1 row = lifetime of process/event
Date Dimension	1 Transaction date	Snapshot date (end of period)	Multiple snapshot dates
No of Dimension	High	Low	Very High
Size	Largest (most detailed grain)	Middle (less detailed grain)	Lowest (highest aggregation)

Why a Factless Fact Table isn't a Contradiction in Terms

Fact tables are used in data warehouses to store facts or measurements that need to be tracked. A fact is different from a fact table, and there are various types of fact tables, each with a specific purpose and usage.

Factless fact tables are used in data warehousing to capture many-to-many relationships among dimensions. They are called "factless" because they have no measures or facts associated with transactions. Essentially, they contain only dimensional keys. It's the structure and use of the table, not the presence of numeric measures, that makes a table a fact table.

Factless fact tables are used in two main scenarios:

1. **Event Tracking**: In this case, the factless fact table records an event. For example, consider a table that records student attendance. The table might contain a student key, a date key, and a course key. There are no measures or facts in this table, only the keys of the students who attended classes on certain dates. The absence of facts is not a problem; simply capturing the occurrence of the event provides valuable information.
2. **Coverage or Bridge Table**: This kind of factless fact table is used to model conditions or coverage data. For example, in a health insurance data warehouse, a factless fact table could be used to track eligibility. The table might contain keys for patient, policy, and date, and it would show which patients are covered by which policies on which dates.

Here's an example of a factless fact table for an attendance system:

Student_ID	Course_ID	Date (FK)
1	101	2023-5-1
2	101	2023-5-1
1	102	2023-5-2
3	103	2023-5-2

This table does not contain any measures or facts. However, it provides valuable information about which student attended which course on which date. We can count the rows to know the number of attendances, join this with other tables to get more details, or even use this for many other analyses. The presence of a row in the factless fact table signifies that an event occurred.

Dimension Tables

Dimension tables in a data warehouse contain the descriptive attributes of the data, and they are used in conjunction with fact tables to provide a more complete view of the data. These tables are typically designed to have a denormalized structure for simplicity of data retrieval and efficiency in handling user queries, which is often essential in a business intelligence context.

Here are some key characteristics and components of dimension tables:

1. Descriptive Attributes: These are the core of a dimension table. These attributes provide context and descriptive characteristics of the dimensions. For example, in a "Customer" dimension table, attributes might include customer name, address, phone number, email, etc.

2. Dimension Keys: These are unique identifiers for each record in the dimension table. These keys are used to link the fact table to the corresponding dimension table. They can be either natural keys (e.g., a customer's email address) or surrogate keys (e.g., a system-generated customer ID).

3. Hierarchies: These are often present within dimensions, providing a way to aggregate data at different levels. For example, a "Time" dimension might have a hierarchy like Year > Quarter > Month > Day.

Example of a Dimension Table: Customer

CustomerID (PK)	CustomerName	Gender	DateOfBirth	City	State	Country
1	John Doe	Male	1980-01-20	NY	NY	USA
2	Jane Smith	Female	1990-07-15	LA	CA	USA
3	Tom Brown	Male	1985-10-30	SF	CA	USA
4	Emma Davis	Female	1995-05-25	TX	TX	USA

In this Customer dimension table, `CustomerID` is the surrogate key which uniquely identifies each customer. Other fields like `CustomerName`, `Gender`, `DateOfBirth`, `City`, `State`, and `Country` provide descriptive attributes for the customers.

The "Sales" fact table in this data warehouse might then have a foreign key that links to the CustomerID in the dimension table. This allows users to analyze sales data not just in terms of raw numbers, but also in terms of the customers.

In summary, dimension tables provide the "who, what, where, when, why, and how" context that surrounds the numerical metrics stored in the fact tables of a data warehouse. They are essential for enabling users to perform meaningful analysis of the data.

Date Dimension Table

The Date dimension table is a special type of dimension table that is usually found in most data warehouses. This table is used to represent all the possible dates that can be used in the other tables of the data warehouse, and it typically includes several attributes related to the date, such as the day, month, year, quarter, and even weekday/weekend flag, fiscal periods, etc.

This is useful because it allows us to avoid repeating this information in our fact tables, and it also makes it easier to perform time-based analyses and aggregations.

Here is an example of a date dimension table:

DateKey	FullDate	DayOfWeek	DayOfMonth	Month	Year	Quarter	IsWeekend
1	2023-01-01	Sunday	1	January	2023	Q1	True
2	2023-01-02	Monday	2	January	2023	Q1	False
3	2023-01-03	Tuesday	3	January	2023	Q1	False
...	...	...	...	...	...	...	...
365	2023-12-31	Sunday	31	December	2023	Q4	True

In this table, `DateKey` is a surrogate key, `FullDate` is the actual date, `DayOfWeek`, `DayOfMonth`, `Month`, `Year`, `Quarter`, and `IsWeekend` are attributes derived from the full date.

This Date dimension table would be linked to fact tables in the data warehouse using the `DateKey` field. This allows for easy filtering, grouping, and other date-based analysis of the data in the fact tables. For example, you might want to compare sales data by quarter or analyze trends on weekdays versus weekends. The Date dimension table makes these types of analyses possible.

▼ SQL Code for creating data dimension tables

```
DROP TABLE date_dim;

CREATE TABLE date_dim
(
  date_key              INT NOT NULL,
  date                    DATE NOT NULL,
  weekday               VARCHAR(9) NOT NULL,
  weekday_num           INT NOT NULL,
```

```
  day_month             INT NOT NULL,
  day_of_year           INT NOT NULL,
  week_of_year          INT NOT NULL,
  iso_week                CHAR(10) NOT NULL,
  month_num             INT NOT NULL,
  month_name            VARCHAR(9) NOT NULL,
  month_name_short        CHAR(3) NOT NULL,
  quarter                   INT NOT NULL,
  year                    INT NOT NULL,
  first_day_of_month    DATE NOT NULL,
  last_day_of_month     DATE NOT NULL,
  yyyymm                CHAR(7) NOT NULL,
  weekend_indr          CHAR(10) NOT NULL
);

ALTER TABLE public.date_dim ADD CONSTRAINT date_dim_pk PRIMARY KEY (date_key);

CREATE INDEX d_date_date_actual_idx
  ON date_dim(date);



INSERT INTO date_dim
SELECT TO_CHAR(datum, 'yyyymmdd')::INT AS date_key,
       datum AS date,
       TO_CHAR(datum, 'TMDay') AS weekday,
       EXTRACT(ISODOW FROM datum) AS weekday_num,
       EXTRACT(DAY FROM datum) AS day_month,
       EXTRACT(DOY FROM datum) AS day_of_year,
       EXTRACT(WEEK FROM datum) AS week_of_year,
       EXTRACT(ISOYEAR FROM datum) || TO_CHAR(datum, '"-W"IW-') || EXTRACT(ISODOW FROM datu
m) AS iso_week,
       EXTRACT(MONTH FROM datum) AS month,
       TO_CHAR(datum, 'TMMonth') AS month_name,
       TO_CHAR(datum, 'Mon') AS month_name_short,
       EXTRACT(QUARTER FROM datum) AS quarter,
       EXTRACT(YEAR FROM datum) AS year,
       datum + (1 - EXTRACT(DAY FROM datum))::INT AS first_day_of_month,
       (DATE_TRUNC('MONTH', datum) + INTERVAL '1 MONTH - 1 day')::DATE AS last_day_of_month,
       CONCAT(TO_CHAR(datum, 'yyyy'),'-',TO_CHAR(datum, 'mm')) AS mmyyyy,
       CASE
           WHEN EXTRACT(ISODOW FROM datum) IN (6, 7) THEN 'weekend'
           ELSE 'weekday'
           END AS weekend_indr
FROM (SELECT '2010-01-01'::DATE + SEQUENCE.DAY AS datum
      FROM GENERATE_SERIES(0, 7300) AS SEQUENCE (DAY)
      GROUP BY SEQUENCE.DAY) DQ
ORDER BY 1;


SELECT * FROM DATE_DIM
```

NULLs in Dimensions

The presence of NULLs in dimension tables can have several effects, some of which are:

1. **Loss of Information:** When a dimension attribute is NULL, it means that some information is missing. Depending on the attribute, this could result in significant losses of information. For example, if a Customer dimension has NULL values for the 'City' attribute, this could impact any analysis related to the geographical locations of customers.
2. **Complicates Query Writing:** NULL values can make query writing more complex. SQL treats NULLs in a special way. For instance, NULL is not equal to any value, not even to another NULL. Therefore, in order to properly handle NULLs, you

often need to use special SQL constructs like IS NULL or IS NOT NULL, or functions like COALESCE().

3. **Effects Join Operations:** If Foreign Keys with NULL values are used in a JOIN condition, it may lead to fewer results than expected. This is because a NULL does not equal anything, including another NULL. As a result, rows with NULL in the joining column of either table will not be matched.

Hierarchies in Dimension Tables

Hierarchies in dimensions are a way of organizing data that reflects defined levels of granularity and relationships among different levels. For instance, a date dimension might include a hierarchy that goes from year to quarter to month to day. An organization dimension might have a hierarchy that goes from company to division to department to team.

Modelling Hierarchies in Dimensions:

There are two main ways to model hierarchies in dimension tables:

1. Single Table Hierarchies (also known as "flat" hierarchies/ Denormalized Dimension):

In this approach, all levels of the hierarchy are stored in a single table, with each row containing all the hierarchy levels for a particular leaf-level member.

For instance, consider a simple Product dimension with a hierarchy that goes Category → Subcategory → Product. In a single table hierarchy, you might have a table that looks like this:

ProductID	Category	Subcategory	Product
1	Food	Fruit	Apple
2	Food	Fruit	Banana
3	Food	Vegetable	Broccoli
4	Beverage	Soda	Cola
5	Beverage	Juice	AppleJuice

This approach is simple and easy to query, but it can result in a lot of redundant data, especially for large hierarchies. This is normally the preferred approach when creating star-schemas

2. Multiple Table Hierarchies (also known as "snowflake" hierarchies/ Normalized Dimension):

In this approach, each level of the hierarchy is stored in a separate table, with each table having a foreign key that links to the level above it.

Using the same Product dimension example, in a multiple-table hierarchy, you might have three separate tables that look like this:

Category Table:

CategoryID	Category
1	Food
2	Beverage

Subcategory Table:

SubcategoryID	CategoryID	Subcategory
1	1	Fruit
2	1	Vegetable
3	2	Soda
4	2	Juice

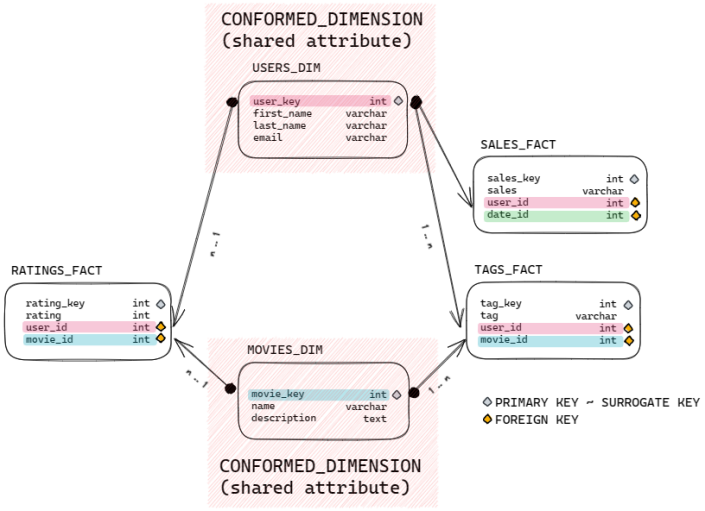
Product Table:

ProductID	SubcategoryID	Product
1	1	Apple
2	1	Banana
3	2	Broccoli
4	3	Cola
5	4	AppleJuice

This approach is more normalized and reduces data redundancy, but it can make querying more complex since it requires joining multiple tables.

Conformed Dimensions

A conformed dimension is a dimension that is shared by multiple fact tables/stars. Essentially, a conformed dimension is a dimension that has the same meaning to every fact table to which it is joined.



Characteristics of Conformed Dimensions:

1. **Consistent definition and understanding:** A conformed dimension has the same meaning and content when being referred to from different fact tables.
2. **Common keys:** Conformed dimensions **can** have the same keys available in each fact table. This enables joining the fact tables on these conformed dimensions, enabling cross-filtering.
3. **Same level of granularity:** Conformed dimensions are at the same level of granularity or detail across the fact tables.

Degenerate Dimension

A type of dimension that is derived from the fact table and does not have its own dimension table, because all the interesting attributes about them are contained in the fact table itself.

Typically, degenerate dimensions are identifiers or numbers that are used for operational or control purposes. For example, an order number, invoice number, or transaction id in a sales fact table. While these identifiers do not have any descriptive attributes of their own (which is why they don't need a separate dimension table), they are extremely useful for tracking and summarizing information.

Example:

Let's consider a retail company's sales fact table. It records data about each sales transaction, including the transaction id, product id, date id, store id, quantity sold, and sales amount.

Sales Fact Table:

TransactionID	ProductID	DateID	StoreID	QuantitySold	SalesAmount
1001	1	101	501	2	100
1002	2	102	502	1	50
1003	3	103	503	5	250
1004	1	104	504	3	150
1005	2	105	505	4	200

Here, the TransactionID is a degenerate dimension. It doesn't have a separate dimension table because there aren't any additional attributes about the transaction that we need to store. However, the TransactionID is important for tracking individual sales transactions and can be used for summarizing data, such as calculating the total sales for each transaction.

Junk Dimension

A junk dimension is a single table composed of a combination of unrelated attributes (flags, indicators, statuses, etc.) to avoid having a large number of foreign keys in the fact table. By grouping these low-cardinality attributes (attributes with very few unique values) into one dimension, we can simplify our model and improve query performance.

For example, let's say we have a retail sales fact table, and we're tracking several binary attributes for each sale:

1. Was the item on sale (Yes/No)?
2. Was a coupon used (Yes/No)?

3. Was the purchase made online (Yes/No)?
4. Was the item returned (Yes/No)?

Instead of adding four separate foreign key columns to our fact table (each corresponding to a dimension table with two rows: "Yes" and "No"), we can combine these attributes into a single junk dimension.

The junk dimension table would look something like this:

JunkKey	OnSale	CouponUsed	OnlinePurchase	ItemReturned
1	Yes	Yes	Yes	Yes
2	Yes	Yes	Yes	No
3	Yes	Yes	No	Yes
4	Yes	Yes	No	No
5	Yes	No	Yes	Yes
...	...	...	...	...

And so on, until all 16 (2^4) possible combinations of these four attributes are represented.

Then, in our fact table, we just include a single foreign key to the Junk Dimension:

TransactionID	ProductID	DateID	JunkKey	QuantitySold	SalesAmount
1001	1	101	2	2	100
1002	2	102	1	1	50
1003	3	103	3	5	250
1004	1	104	4	3	150
1005	2	105	5	4	200

By using this junk dimension, we're able to keep our fact table simpler, and our queries can become faster and more efficient.

Role Playing Dimension

A Role-Playing Dimension is a concept where a single dimension can be used in multiple roles in the same fact table. This happens when a dimension table has multiple relationships with the fact table. Each relationship represents a logically distinct role that the dimension plays in the fact table.

For example, let's take a date dimension. A single date dimension can be used for "Order Date", "Shipping Date", "Delivery Date", etc., in a sales fact table. Each of these dates could reference the same date dimension table but would represent different business concepts.

Here's how it could look:

Date Dimension Table

DateKey	Date	Day	Month	Year
1	2023-01-01	1	1	2023
2	2023-01-02	2	1	2023
3	2023-01-03	3	1	2023
...	...	...	...	...

Sales Fact Table

SalesID	ProductID	OrderDateKey	ShippingDateKey	DeliveryDateKey	Quantity	TotalPrice
1	1	1	2	3	2	100
2	2	2	3	4	1	50
3	1	3	4	5	3	150
4	2	4	5	6	2	100
5	1	5	6	7	1	50

In the above example, the DateKey is role-playing as OrderDateKey, ShippingDateKey, and DeliveryDateKey in the Sales fact table. Each of these roles represents a different business concept, but they all use the same Date dimension table for their values. This is an efficient way to reuse the same dimension in different contexts, avoiding redundancy and making the data model more manageable.

For analysis in SQL, we can create additional *views* for each role

Designing Fact and Dimension Tables

Creating a fact table in a data warehouse involves a series of steps. The exact process might vary slightly depending on the specifics of your project and the tools you're using, but the following steps provide a general outline.

1. **Identify the Business Process**: The first step is to identify the business process that you want to analyze. This could be sales transactions, inventory levels, or customer service calls, for example. The business process will guide the design of the fact table.
2. **Identify the Granularity**: Granularity refers to the level of detail or depth of a fact information. Granularity is the lowest level of information that is stored in a table. For example, if you are building a fact table to analyze sales, the granularity could be at the transaction level (each row represents a sale) or at the item level (each row represents an item within a sale).
3. **Identify the Facts**: Facts are typically numeric values that you wish to analyze. In a sales business process, examples of facts could be Sales Amount, Quantity Sold, Profit, etc. A fact table would typically store these facts along with keys to related dimension tables. The facts are usually the result of some business process event and are what you will be analyzing.
4. **Identify the Dimensions**: Dimensions are descriptive attributes related to the facts. They provide the context for the facts and are often what you group by when analyzing the facts. Typical dimensions include time, location, product, and customer. Each dimension would typically have its own table, which would include a primary key and additional attributes related to the dimension
5. **Design the Fact Table**: The fact table is usually designed next, with a column for each fact identified in Step 3 and a foreign key for each dimension identified in Step 4. The primary key of a fact table is usually a composite key composed of all its foreign keys.
6. **Create Dimension Tables**: Each dimension table should include a primary key that uniquely identifies each row and other descriptive attributes of the dimension. It should also contain other attributes that provide descriptive characteristics related to the dimension.
7. **Create Relationships**: Establish relationships between the fact table and the dimension tables. This is usually done by creating foreign keys in the fact table that correspond to the primary keys in the dimension tables.
8. **Load the Data**: Once the fact table and dimension tables have been created, the next step is to load the data. This typically involves extracting data from source systems, transforming it into the format of the fact and dimension tables (a process known as ETL), and then loading it into the data warehouse.
9. **Test and Validate**: After the data has been loaded, test and validate the fact table to ensure that it accurately represents the business process and supports the desired analysis. This might involve running sample queries, checking for data quality issues, and verifying that the relationships between the fact table and the dimension tables are working correctly.

Let's walk through an example scenario for creating a fact table in a retail business setting:

1. Identify the Business Process:
Let's say our business process is "Sales" - the selling of items to customers.

2. Identify the Granularity:
In this case, we decide that each row in the fact table will represent an individual item sold in a transaction.

3. Identify the Facts:
The facts we want to analyze could include the quantity of items sold, the price of each item, and the total amount for each item sold (quantity * price).

4. Identify the Dimensions:
Possible dimensions for this process include Time (when was the item sold), Product (which product was sold), Store (in which store the sale happened), and Customer (who purchased the item).

5. Design the Fact Table:
Our fact table could look something like this:

Sale_ID (PK)	Time_ID (FK)	Product_ID (FK)	Store_ID (FK)	Customer_ID (FK)	Quantity_Sold	Item_Price	Total_Amount
1	20230501	123	001	456	2	50	100

6. Create Dimension Tables:
For each of the dimensions identified, we'll create a separate table. For instance, the Product dimension table might look like this:

Product_ID (PK)	Product_Name	Product_Category	Product_Price
123	Shirt	Clothing	50

7. Create Relationships:
The relationships between the fact table and dimension tables would be established using the foreign keys in the fact table. For example, the Product_ID in the fact table would link to the Product_ID in the Product dimension table.

8. Load the Data:
We would then extract data from our various sources (maybe point of sale systems, CRM, etc.), transform the data to match the structure of our fact and dimension tables and load the data into these tables.

9. Test and Validate:
Finally, we would need to run queries to ensure our fact table works correctly. For example, we might want to find out the total sales for a specific product in a specific store. We would use our fact and dimension tables to answer this question and verify that the result is as expected.

Remember, this is a simplified example, and real-world scenarios can be more complex, but it should give you a general idea of how the process works.

SQL for Dimension and Fact Tables

Fact tables and dimension tables can be created in star and snowflake schemas using SQL in a data warehouse. The create table statements are used to define the structure of the tables, including columns, data types, primary keys, and foreign keys.

For a star schema dimension table, the primary key is typically a single column, even if the table has multiple hierarchical levels. In a snowflake schema, the non-terminal dimension tables have both primary keys and foreign keys. The terminal dimension table in a snowflake schema has only a primary key, as there is no higher level to reference.

For a transaction-grained fact table, the primary key is a combination of all foreign keys related to the dimension tables. Each foreign key explicitly points to a specific table and column within that table. The SQL syntax for periodic snapshot fact tables is almost identical to that of transaction-grained fact tables.

Regardless of the type of fact table being created, the SQL model follows the same structure: identify keys, combine necessary keys as the primary key, and have a foreign key clause for each key pointing back to the respective dimension tables. The only complication arises when there are multiple relationships with the same dimension, as in accumulating snapshot fact tables or some types of factless fact tables. In such cases, multiple foreign key clauses for each date reference the same date dimension and its key.

Section 6: Managing Data Warehouse History Through Slowly Changing Dimensions

Slowly changing dimensions (SCD) are tables in a dimensional model that handle changes to dimension values over time and not on a set schedule. SCDs present a challenge because they can alter historical data and, in the process, affect the outcome of current analyses.

Over time, it is possible that certain product name changes or maybe a customer changes phone number. This will lead to the case where we will have to change the dimension table to reflect these changes. There are various strategies to tackle the different cases. There are three main types of SCDs: Type 1, Type 2, and Type 3.

Type 1 (Overwrite)

A Type 1 SCD always reflects the latest values, and when changes in source data are detected, the dimension table data is overwritten. No history is kept.

E.g. When a customer's email address or phone number changes, the dimension table updates the customer row with the new values.

CustomerID	FirstName	LastName	EmailAddress	CompanyName	InsertedDate	ModifiedDate
2	Keith	Harris	keith0@aw.com	Progressive Sports	2021-03-20	2021-03-20
3	Donna	Carreras	donna0@aw.com	A Bike Store	2021-03-20	2021-03-20

CustomerID	FirstName	LastName	EmailAddress	CompanyName	InsertedDate	ModifiedDate
2	Keith	Harris	keith0@aw.com	Progressive Sports	2021-03-20	2021-03-20
3	Donna	Carreras	donna0@aw.com	Bikes, Bikes, Bikes	2021-03-20	2021-03-22

Type 2 (Add a new Row):

A Type 2 SCD supports the versioning of dimension members. It includes columns that define the date range validity of the version (for example, `StartDate` and `EndDate`) and possibly a flag column (for example, `IsCurrent`) to easily filter by current dimension members.

Current versions may define an empty end date (or 12/31/9999), which indicates that the row is the current version. The table must also define a **surrogate key** because the business key (in this instance, RepSourceID) won't be unique.

SalesRepID	RepSourceId	FirstName	LastName	Region	StartDate	EndDate	IsCurrent
1	312	Jun	Cao	Southwest	2021-03-20	9999-12-31	True
2	331	Susan	Eaton	Southcentral	2021-03-20	9999-12-31	True

SalesRepID	RepSourceId	FirstName	LastName	Region	StartDate	EndDate	IsCurrent
1	312	Jun	Cao	Southwest	2021-03-20	9999-12-31	True
2	331	Susan	Eaton	Southcentral	2021-03-20	2021-03-21	False
3	331	Susan	Eaton	Southeast	2021-03-22	9999-12-31	True

Instead of putting NULL for the End date, its better to put a future date

▼ **Code to apply type 1 and 2 logic**

```
/*Logic to implement Type 1 and Type 2 updates can be complex, and there are various techniq
ues you can use. For example, you could use a combination of UPDATE and INSERT statements as
shown in the following code example:*/

-- Insert new customers
INSERT INTO dbo.DimCustomer
SELECT stg.CustomerNo,
       stg.CustomerName,
       stg.EmailAddress,
       stg.Phone,
       stg.StreetAddress
FROM dbo.StageCustomers AS stg
WHERE NOT EXISTS
    (SELECT * FROM dbo.DimCustomer AS dim
     WHERE dim.CustomerAltKey = stg.CustomerNo);

-- Type 1 updates (name, email, phone)
UPDATE dbo.DimCustomer
SET CustomerName = stg.CustomerName,
    EmailAddress = stg.EmailAddress,
    Phone = stg.Phone
FROM dbo.StageCustomers AS stg
WHERE dbo.DimCustomer.CustomerAltKey = stg.CustomerNo;

-- Type 2 updates (geographic address)
INSERT INTO dbo.DimCustomer
SELECT stg.CustomerNo AS CustomerAltKey,
       stg.CustomerName,
       stg.EmailAddress,
       stg.Phone,
       stg.StreetAddress,
       stg.City,
       stg.PostalCode,
       stg.CountryRegion
```

```
FROM dbo.StageCustomers AS stg
JOIN dbo.DimCustomer AS dim
ON stg.CustomerNo = dim.CustomerAltKey
AND stg.StreetAddress <> dim.StreetAddress;

/*As an alternative to using multiple INSERT and UPDATE statement, you can use a single MERG
E statement to perform an "upsert" operation to insert new records and update existing ones,
as shown in the following example, which loads new product records and applies type 1 update
s to existing products*/

MERGE dbo.DimProduct AS tgt
    USING (SELECT * FROM dbo.StageProducts) AS src
    ON src.ProductID = tgt.ProductBusinessKey
WHEN MATCHED THEN
    UPDATE SET
        tgt.ProductName = src.ProductName,
        tgt.ProductCategory = src.ProductCategory
        tgt.Color = src.Color,
        tgt.Size = src.Size,
        tgt.ListPrice = src.ListPrice,
        tgt.Discontinued = src.Discontinued
WHEN NOT MATCHED THEN
    INSERT VALUES
        (src.ProductID,
         src.ProductName,
         src.ProductCategory,
         src.Color,
         src.Size,
         src.ListPrice,
         src.Discontinued);

/*Another way to load a combination of new and updated data into a dimension table is to use
a CREATE TABLE AS (CTAS) statement to create a new table that contains the existing rows fro
m the dimension table and the new and updated records from the staging table. After creating
the new table, you can delete or rename the current dimension table, and rename the new tabl
e to replace it.*/

CREATE TABLE dbo.DimProductUpsert
WITH
(
    DISTRIBUTION = REPLICATE,
    CLUSTERED COLUMNSTORE INDEX
)
AS
-- New or updated rows
SELECT  stg.ProductID AS ProductBusinessKey,
        stg.ProductName,
        stg.ProductCategory,
        stg.Color,
        stg.Size,
        stg.ListPrice,
        stg.Discontinued
FROM    dbo.StageProduct AS stg
UNION ALL
-- Existing rows
SELECT  dim.ProductBusinessKey,
        dim.ProductName,
        dim.ProductCategory,
        dim.Color,
        dim.Size,
        dim.ListPrice,
```

```
        dim.Discontinued
FROM    dbo.DimProduct AS dim
WHERE NOT EXISTS
(   SELECT  *
    FROM dbo.StageProduct AS stg
    WHERE stg.ProductId = dim.ProductBusinessKey
);

RENAME OBJECT dbo.DimProduct TO DimProductArchive;
RENAME OBJECT dbo.DimProductUpsert TO DimProduct;
```

Type 3 (Add a new Column/Attribute):

A Type 3 SCD supports storing two versions of a dimension member as separate columns.

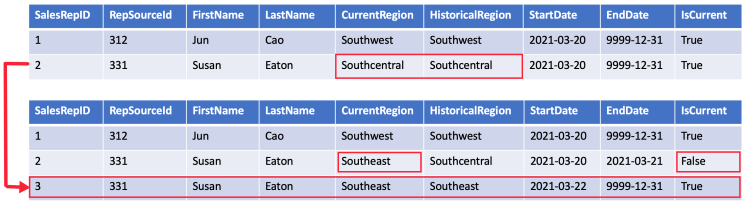
Here instead of having multiple rows to signify changes, we have multiple columns. We do have an effective/modified date column to show when the change took place.

CustomerID	FirstName	LastName	CurrentEmail	OriginalEmail	CompanyName	InsertedDate	ModifiedDate
2	Keith	Harris	keith0@aw.com	keith0@aw.com	Progressive Sports	2021-03-20	2021-03-20
3	Donna	Carreras	donna0@aw.com	donna0@aw.com	A Bike Store	2021-03-20	2021-03-20

CustomerID	FirstName	LastName	CurrentEmail	OriginalEmail	CompanyName	InsertedDate	ModifiedDate
2	Keith	Harris	keith0@aw.com	keith0@aw.com	Progressive Sports	2021-03-20	2021-03-20
3	Donna	Carreras	dc3@aw.com	donna0@aw.com	A Bike Store	2021-03-20	2021-03-22

Type 6:

A Type 6 SCD combines Type 1, 2, and 3. In Type 6 design we also store the current value in all versions of that entity so you can easily report the current value or the historical value.



SalesRepID	RepSourceId	FirstName	LastName	CurrentRegion	HistoricalRegion	StartDate	EndDate	IsCurrent
1	312	Jun	Cao	Southwest	Southwest	2021-03-20	9999-12-31	True
2	331	Susan	Eaton	Southcentral	Southcentral	2021-03-20	9999-12-31	True

SalesRepID	RepSourceId	FirstName	LastName	CurrentRegion	HistoricalRegion	StartDate	EndDate	IsCurrent
1	312	Jun	Cao	Southwest	Southwest	2021-03-20	9999-12-31	True
2	331	Susan	Eaton	Southeast	Southcentral	2021-03-20	2021-03-21	False
3	331	Susan	Eaton	Southeast	Southeast	2021-03-22	9999-12-31	True

---

Section 7: Designing Your ETL

Compare ETL to ELT

ETL and ELT are methodologies for handling the flow of data from source systems to a data warehouse or data lake for analytical purposes.

ETL (Extract, Transform, Load):

1. **Extract**: Data is extracted from various source systems (such as databases, applications, or external data sources) in raw form, often with errors.

2. **Transform**: The data is transformed into a consistent and uniform format. This may involve cleaning, validating, and converting data types, as well as aggregating, filtering, and sorting data to make it suitable for the target data warehouse or data mart.
3. **Load**: The transformed data is loaded into the user access layer (such as a data warehouse or data mart) where it becomes available for business intelligence (BI) and analytics.

ETL requires significant upfront work, including business analysis, data modelling, and data structure design, to ensure that the data is ready for analytical use when it is loaded into the user access layer.

ELT (Extract, Load, Transform):

1. **Extract**: Data is extracted from source systems, just like in the ETL process.
2. **Load**: Instead of transforming the data immediately, it is loaded into a big data environment, such as Hadoop Distributed File System (HDFS) or cloud-based storage like Amazon S3, in its raw form. This is usually done for both structured and unstructured data.
3. **Transform**: When the data is needed for analytical purposes, it is transformed using the computing power of the big data environment. This approach allows for more flexible and scalable data processing, as **transformations can be performed on demand**.

ELT defers the data modelling and analysis until the data is required for analytical use, following a schema-on-read approach. This allows for more agile data handling and reduces the need for upfront data modelling and analysis.

In summary, ETL is more suitable for traditional data warehousing with predefined data structures and strict data quality requirements, while ELT is more appropriate for big data environments, data lakes, or data warehouse-data lake hybrids, where data can be stored in its raw form and transformed later when needed.

	ETL	ELT
Process Order	Data is Extracted, Transformed, and then Loaded into the target database or data warehouse.	Data is Extracted, Loaded into the target system, and then Transformed.
Data Transformation	Transformation occurs before loading, typically in an intermediate staging area.	Transformation occurs after loading, directly in the target system.
Performance	May be slower for large data volumes, as transformations are performed before loading.	May be faster for large data volumes, as raw data is loaded first and transformations are performed within the target system.
Hardware Requirement	Can be less demanding, as transformation is done before loading the data, which can be distributed across multiple servers.	Can be more demanding, especially for cloud-based data warehouses that are designed for heavy computation.
Data Storage	Only transformed data is stored in the target system, which can be more efficient in terms of storage.	Both raw and transformed data are often stored, which can require more storage space.
Flexibility	Less flexible for changes in transformation logic, as changes may require re-extracting and re-loading data.	More flexible for changes in transformation logic, as raw data is already loaded and can be re-transformed.
Complexity	Can be more complex, as transformation logic needs to be defined before loading data.	Can be less complex, as raw data is loaded first and transformations can be performed using SQL or other data manipulation languages.
Use Case	Suitable for smaller data volumes, or when transformations are complex and need to be carefully controlled.	Suitable for large data volumes, or when using modern cloud-based data warehouses that can handle complex transformations.

Design the Initial Load ETL

Next, we focus on two forms of ETL (Extract, Transform, Load) processes in data warehousing environments: initial ETL and incremental ETL.

Initial ETL:

This is a one-time process, typically performed right before the data warehouse goes live. The goal is to gather all the relevant data needed for business intelligence (BI) and analytics, transform it, and load it into the user access layer of the data

warehouse.

- **Relevance is key**: Only data that is essential or likely to be needed for BI and analytics should be included instead of importing all data from the source systems.
- Historical data may also be imported to provide a basis for trend analysis and other historical reporting.
- Initial ETL might be repeated in cases of data corruption or re-platforming, but this is not common.

Incremental/Delta ETL:

This process is used to keep the data warehouse up-to-date after the initial ETL is completed. It is done regularly to refresh and update the data warehouse with new, modified, or deleted data.

- Regularly updates the data warehouse, ensuring it remains up-to-date.
- Adds new data, modifies existing data, and handles deleted data without purging it.
- Makes the data warehouse non-volatile and static between ETL runs.

Four major incremental ETL patterns:

1. **Append pattern**: New information is added to the existing data warehouse content.
2. **In-place update**: Existing rows are updated with changes, rather than appending new data.
3. **Complete replacement**: A portion of the data warehouse is entirely overwritten, even if only a small part of the data has changed.
4. **Rolling append**: A fixed duration of history is maintained, with new updates appending data and removing the oldest equivalent data.

Modern incremental ETL primarily uses the append and in-place update patterns, which align well with dimensional data handling.

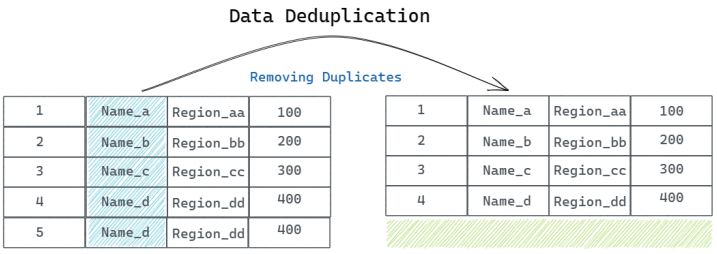
Why do we require Data Transformation?

Data transformation, a key aspect of ETL focuses on two main goals: uniformity and restructuring of data.

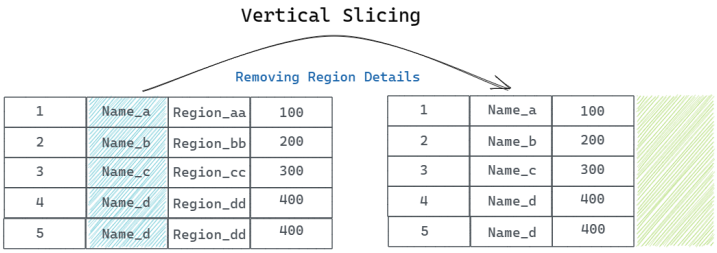
Uniformity ensures that data from different systems is transformed to allow for apples-to-apples comparisons. Restructuring involves organizing raw data from the staging layer into well-engineered data structures.

Some of the data transformation models are explained:

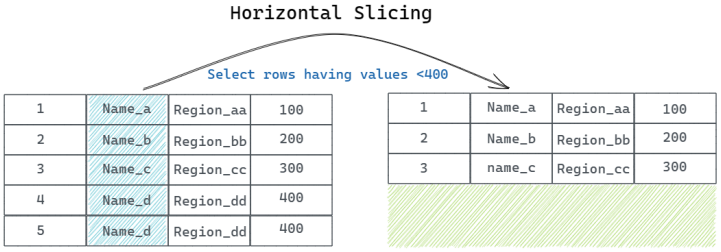
1. **Data value unification**: This involves unifying data values from different systems to present a consistent format to users. E.g. Daily sales data from two different regions being transformed to use a uniform abbreviation format in the data warehouse.
2. **Data type and size unification**: This involves unifying data types and sizes from different systems into a single representation in the data warehouse. E.g. Salesperson names from different regions have different character lengths, which need to be standardized in the data warehouse.
3. **Data deduplication**: This involves identifying and removing duplicate data to ensure accuracy in analytics and reporting. E.g. A particular salesperson's sales data is duplicated across different systems. The data warehouse should store only one copy of the information to avoid double counting.



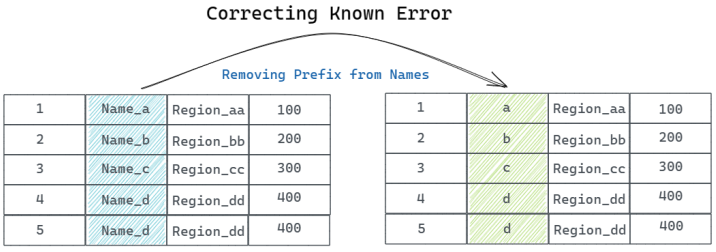
4. **Dropping columns (vertical slicing)**: This involves removing unnecessary columns from the data warehouse. E.g. Region data from a particular source system may be redundant as it has already been captured by another source. Thus, unnecessary columns need to be removed to maintain data quality in the warehouse.



5. **Row filtering based on values (horizontal slicing)**: This model filters out rows based on certain values in specific fields. E.g. Data warehouse is being built to analyze only particular sales values thus any rows not satisfying the value data need to be dropped.



6. **Correcting known errors**: This model involves fixing errors in the source data during the ETL process. E.g. names of salespersons have an unnecessary prefix that needs to be removed.



Some of the more advanced transformations may include joining, splitting (by length/position or by delimiter), aggregating (sum, count, average), deriving new values etc

Implement Mix-and-Match Incremental ETL

Let's discuss the flexibility and customization of incremental ETL (Extract, Transform, Load) processes in data warehousing. Although ETL feeds might appear similar, they can vary in terms of frequency and patterns, depending on the data and requirements.

Different ETL feeds might update the data warehouse at different frequencies, such as daily, hourly, or weekly, depending on the data volatility and criticality. Furthermore, incremental ETL patterns, like append, complete replacement, and in-place updates, can also vary across feeds.

The frequency and patterns are not solely determined by the ETL feed but can also depend on the table level within the source systems. Each source system can have a set of tables with different frequencies and ETL patterns. Thus it's important to implement a mix-and-match approach to cater to the specific needs of each table and source system.

Section 8: Selecting and Using a Data Warehouse

ETL Tools

Extract, Transform, Load (ETL) tools are used in the process of extracting data from different source systems, transforming it into a standard format, and loading it into a destination system, typically a data warehouse or data lake. There are a variety of ETL tools available, which cater to different environments and needs. Some of these tools are:

1. **Enterprise ETL Tools:**
   - **Informatica PowerCenter:** This is a widely used ETL tool that supports all the steps of the ETL process and is known for its high performance, intuitive interface, and wide support for different data sources and targets.
   - **IBM InfoSphere DataStage:** It is a part of IBM's Information Platforms Solutions suite and also its InfoSphere Platform. It uses a graphical notation to construct data integration solutions.
   - **Oracle Data Integrator (ODI):** Oracle's ETL tool provides a fully unified solution for building, deploying, and managing real-time data-centric architectures in an SOA, BI, and data warehouse environment.
   - **SAP Business Objects Data Services (BODS):** This ETL tool by SAP provides comprehensive data integration, data quality, and data processing.
   - **Alteryx, Microsoft SSIS** etc.
2. **Open Source ETL Tools:**
   - **Apache NiFi:** Apache NiFi supports powerful and scalable directed graphs of data routing, transformation, and system mediation logic.
   - **Talend Open Studio:** It's a robust open-source ETL tool that supports a wide array of source and target systems. Talend also offers a commercial version with additional features.
   - **Pentaho Data Integration:** Also known as Kettle, this tool offers data integration and transformation, including ETL capabilities.
3. **Cloud-based ETL Tools:**
   - **AWS Glue:** This is a fully managed ETL service provided by Amazon that makes it easy to prepare and load data for analytics.
   - **Google Cloud Dataflow:** This is a cloud-based data processing service for both batch and real-time data streaming applications.
   - **Azure Data Factory:** This is a cloud-based data integration service provided by Microsoft that orchestrates and automates the movement and transformation of data.
4. **Python-based ETL Tools:** For those who prefer to write their own ETL scripts, there are Python libraries such as **Pandas** and **PySpark** that can be used to create custom ETL processes.

The choice of an ETL tool depends on various factors: such as the nature of the source and target systems, the complexity of the ETL process, the required performance, the available budget, and the skills of the available staff.

1. **Data Source and Target Compatibility:** The ETL tool should be compatible with your data sources (databases, APIs, file formats, etc.) and target systems (data warehouse, data lake, etc.).
2. **Performance:** The efficiency and speed of the ETL tool are important, especially if you need to process large volumes of data.
3. **Scalability:** The ETL tool should be able to scale and handle increasing data volumes as your business grows.
4. **Ease of Use:** A tool with a user-friendly interface can lower the learning curve and increase productivity. Some ETL tools provide a graphical interface to design ETL flows, which can be easier to use than writing code.
5. **Data Transformation Capabilities:** The ETL tool should support the types of data transformations you need, such as filtering, aggregation, joining, splitting, data type conversion, etc.
6. **Data Quality Features:** Some ETL tools provide built-in features to ensure data quality, such as data profiling, data validation, and error handling capabilities.
7. **Scheduling and Automation:** The ETL tool should provide capabilities to schedule ETL jobs and automate workflows.
8. **Real-time Processing:** If you need real-time or near real-time data integration, choose an ETL tool that supports streaming data and real-time processing.
9. **Security:** The ETL tool should provide strong security features, including data encryption, user authentication, access controls, and audit logs.

10. **Cost:** The cost of ETL tools can vary widely, from free open-source tools to expensive commercial solutions. Consider your budget and the total cost of ownership, including license costs, hardware costs, support costs, and training costs.

Optimizing Data Warehouse

A performance improvement strategy should be in place whenever a DWH is delivered. This ensures that DWH stays relevant and meets the business end-user requirements on time.

Indexing

Adding some search functionality or indexes can drastically improve your chances of finding your item quicker. Indexes store the value from the given column in a searchable structure, which allows the query to read fewer data to find the information. However, they have an overhead, and having too many indexes slows down the insert and update operations on your DWH. Bitmap indexes, B-tree indexes, and columnstore indexes are some of the types that can be used in a data warehouse.

1. **Bitmap Indexing**

   Bitmap indexing is a special kind of database indexing that uses bitmaps or bit arrays. It is particularly effective for queries on large tables that return a small percentage of rows, and it's very useful for low-cardinality fields, i.e., fields that have a small number of distinct values.

   For example, let's consider a table `Customer` in a database, where we have a column `Gender` with two distinct values: Male and Female.

   ```
   Customer Table:
   +----+------+--------+
   | ID | Name | Gender |
   +----+------+--------+
   | 1  | Alex | Male   |
   | 2  | Sam  | Female |
   | 3  | John | Male   |
   | 4  | Anna | Female |
   | 5  | Mike | Male   |
   +----+------+--------+
   ```

   A bitmap index on the `Gender` column would look something like this:

   ```
   Bitmap Index:
   +--------+---------+
   | Gender | Bitmap  |
   +--------+---------+
   | Male   | 10101   |
   | Female | 01010   |
   +--------+---------+
   ```

   In this bitmap representation, each bit position corresponds to a row in the table. A `1` indicates that the row has that value for the indexed column, and a `0` indicates that it does not.

   For example, the bitmap for `Male` is `10101`. This means that rows 1, 3, and 5 in the table have the `Gender` value `Male`. Similarly, the bitmap for `Female` is `01010`, which corresponds to rows 2 and 4.

   The advantage of a bitmap index is that it can be very space-efficient for low-cardinality data, and it allows for very fast Boolean operations (AND, OR, NOT) between different bitmaps. For instance, if you want to find all customers who are Male, the database engine can quickly find this information from the bitmap index without scanning the whole table. However, it performs poorly when it comes to high-cardinality data, where the number of distinct values is high, and the bitmaps become sparse and take up more space.

2. **B-tree Indexing**

   B-tree indexing is a commonly used indexing method in databases. B-tree stands for "balanced tree", and it's a sorted data structure that maintains sorted data and allows for efficient insertion, deletion, and search operations.

   B-tree indexes are beneficial when dealing with large amounts of data, as they keep data sorted and allow searches, sequential access, insertions, and deletions in logarithmic time. They are particularly useful for databases stored on disk, as they minimize disk I/O operations.

   Let's consider a simplified example of a B-tree index on an `Employee` table.

```
Employee Table:
+----+-------+
| ID | Name  |
+----+-------+
| 1  | Alex  |
| 2  | Bob   |
| 3  | Carol |
| 4  | Dave  |
| 5  | Ed    |
+----+-------+
```

We might create a B-tree index on the `ID` column. The B-tree index structure might look something like this:

```
           [3]
       /   \
    [2]     [5]
   /   \   /   \
 [1] [3] [4]  [5]
```

In this tree, each node represents a page or block of the index, and each page contains one or more keys and pointers. The keys act as separation values which divide its subtrees.

For instance, in the root node of the tree above, the key `3` separates two subtrees. The left subtree contains values less than `3`, and the right subtree contains values greater than `3`.

If we wanted to find the data associated with `ID = 4`, we would:

1. Start at the root and see the value `3`.
2. `4` is greater than `3`, so we follow the right pointer.
3. In the right child node, we find `5`. Since `4` is less than `5`, we follow the left pointer.
4. We reach the leaf node with the key `4`, where we can find the pointer to the actual data record in the table.

This efficient structure of the B-tree index allows the database to quickly find data without scanning the whole table. It's important to note that in actual databases, B-trees can have many more child nodes per parent, and the tree can be much deeper, allowing for efficient indexing of large amounts of data.

3. **Columnstore Indexing**

   Columnstore indexing is a technology used to enhance the processing speed of database queries and is especially efficient for large data warehouse queries. The key difference between a columnstore index and traditional row-based indexes (like B-tree) is that data is stored column-wise rather than row-wise. This columnar storage allows for high compression rates and significantly improves query performance.

   Columnstore indexes work best on fact tables in the data warehouse schema that are loaded through a single ETL process and used for read-only queries. Let's consider an example:

   Imagine we have a Sales table with millions of rows and the following structure:

```
Sales Table:
+--------+-------+----------+--------+
| SaleID | Item  | Quantity | Price  |
+--------+-------+----------+--------+
| 1      | Apple | 10       | 2.00   |
| 2      | Pear  | 15       | 2.50   |
| 3      | Grape | 20       | 3.00   |
| ...    | ...   | ...      | ...    |
+--------+-------+----------+--------+
```

   If we create a columnstore index on this table, the data would be stored something like this:

```
SaleID Column:     | 1 | 2 | 3 | ...
Item Column:       | Apple | Pear | Grape | ...
```

```
Quantity Column:   | 10 | 15 | 20 | ...
Price Column:      | 2.00 | 2.50 | 3.00 | ...
```

Each column is stored separately, which enables high compression rates because the redundancy within a column is typically higher than within a row. This structure also improves the performance of queries that only need a few columns from the table because only the columns needed for the query are fetched from storage.

For example, if we wanted to calculate the total revenue from all sales, we would need only the `Quantity` and `Price` columns. A database with a columnstore index could perform this operation much faster than traditional row-based storage because it only needs to read two columns instead of the entire table.

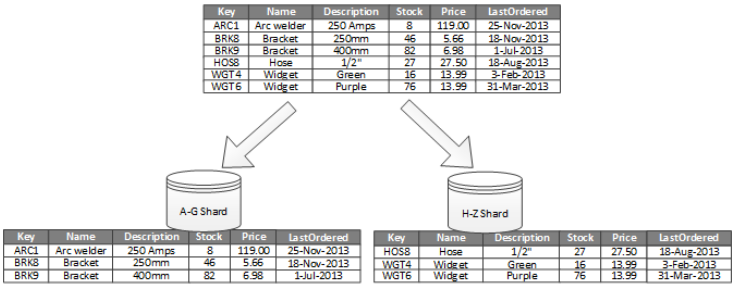
It's also worth mentioning that columnstore indexes allow for batch processing, which is another aspect that enhances their performance compared to traditional B-tree indexes.

Partitioning

In many large-scale solutions, data is divided into *partitions* that can be managed and accessed separately. Partitioning can improve scalability, reduce contention, and optimize performance. It can also provide a mechanism for dividing data by usage pattern. There are three typical strategies for partitioning data:

1. **Horizontal partitioning** (often called *sharding*)

   In this strategy, each partition is a separate data store, but all partitions have the same schema. Each partition is known as a *shard* and holds a specific subset of the data, such as all the orders for a specific set of customers.



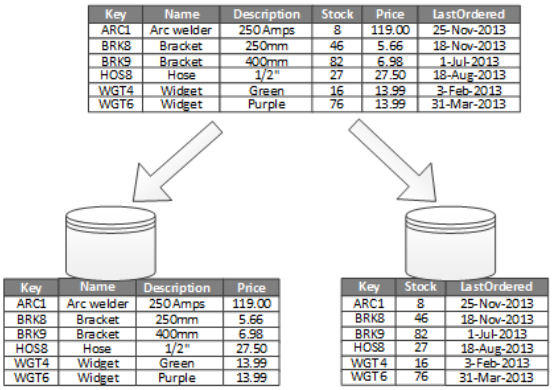
Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013

Key	Name	Description	Stock	Price	LastOrdered
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

2. **Vertical partitioning**

   In this strategy, each partition holds a subset of the fields for items in the data store. The fields are divided according to their pattern of use. For example, frequently accessed fields might be placed in one vertical partition and less frequently accessed fields in another.



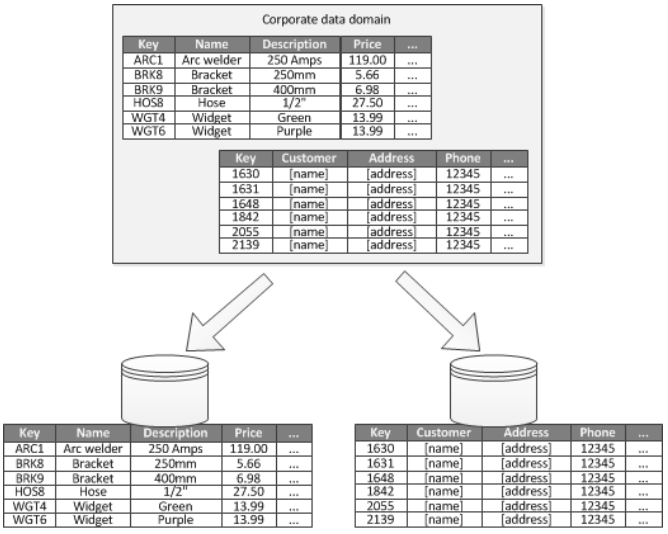
Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Key	Name	Description	Price
ARC1	Arc welder	250 Amps	119.00
BRK8	Bracket	250mm	5.66
BRK9	Bracket	400mm	6.98
HOS8	Hose	1/2"	27.50
WGT4	Widget	Green	13.99
WGT6	Widget	Purple	13.99

Key	Stock	LastOrdered
ARC1	8	25-Nov-2013
BRK8	46	18-Nov-2013
BRK9	82	1-Jul-2013
HOS8	27	18-Aug-2013
WGT4	16	3-Feb-2013
WGT6	76	31-Mar-2013

3. **Functional partitioning**

   In this strategy, data is aggregated according to how it is used by each bounded context in the system. For example, an e-commerce system might store invoice data in one partition and product inventory data in another.



Materialized Views:

Materialized views are a type of view in databases but with a twist. While a standard view is a virtual table that dynamically retrieves data from the underlying tables, a materialized view stores the result of the query physically, similar to a table. Because of this, they can significantly speed up query execution times as the database does not need to compute the result set every time the view is queried - it simply accesses the pre-computed results.

Let's consider an example where a materialized view can be beneficial. Suppose we have two tables, `Orders` and `Customers`, in a retail database.

The `Orders` table might look something like this:

```
Orders:
+---------+------------+-------+
| OrderID | CustomerID | Total |
+---------+------------+-------+
| 1       | 101        | 50.00 |
| 2       | 102        | 75.00 |
| 3       | 103        | 25.00 |
| ...     | ...        | ...   |
+---------+------------+-------+
```

And the `Customers` table might look like this:

```
Customers:
+------------+-------+
| CustomerID | Name  |
+------------+-------+
| 101        | Alice |
| 102        | Bob   |
| 103        | Carol |
| ...        | ...   |
+------------+-------+
```

Now, suppose we frequently need to calculate the total amount spent by each customer. We could create a view that aggregates this data:

```
CREATE VIEW TotalSpent AS
SELECT Customers.Name, SUM(Orders.Total) as TotalSpent
FROM Customers
JOIN Orders ON Customers.CustomerID = Orders.CustomerID
GROUP BY Customers.Name;
```

Each time we query this view, the database would need to perform the join and aggregation operation, which can be costly on large tables.

However, if we were to create a materialized view:

```
CREATE MATERIALIZED VIEW TotalSpent AS
SELECT Customers.Name, SUM(Orders.Total) as TotalSpent
FROM Customers
JOIN Orders ON Customers.CustomerID = Orders.CustomerID
GROUP BY Customers.Name;
```

The database would store the result of the query in a table-like structure. When we query the `TotalSpent` view, the database would simply return the pre-computed results, which can be significantly faster.

However, there is a tradeoff. The data in a materialized view can become stale when the underlying data changes. Depending on the database system, you may need to manually refresh the materialized view, or it might be possible to set it to refresh automatically at certain intervals or in response to certain events.

Compression

Compression reduces the amount of storage space needed and can also improve query performance, as less data needs to be read from the disk. Many modern DBMSs offer data compression features.

Parallel Processing

Many data warehouse systems support parallel processing, which can greatly improve the performance of large queries. This involves dividing a task into smaller subtasks that are executed concurrently.
