Part 2:

Introduction:

The Data Vault modeling approach is a methodology used for designing data warehouses that prioritize flexibility, scalability, and auditability. Here's a breakdown of its key concepts:

Hub Tables:

Hub tables represent the core business entities or "hubs" in the data warehouse.

Each hub table stores unique business keys (natural keys) for a specific entity.

Hub tables are typically simple and contain only one column, which serves as the primary key.

Examples of hub tables include Customer Hub, Product Hub, or Supplier Hub.

Satellite Tables:

Satellite tables capture the detailed attributes and historical changes associated with hub entities.

Each satellite table is linked to a corresponding hub table and contains descriptive attributes related to the hub entity.

Satellite tables often include additional columns for tracking metadata such as load dates, source system identifiers, and record status indicators.

Satellite tables support various types of slowly changing dimensions (SCDs) to manage historical changes to attribute values over time.

Link Tables:

Link tables establish relationships between hub entities or between hubs and satellite tables.

They represent the connections or associations between different business entities.

Link tables typically contain foreign keys referencing the primary keys of related hub tables.

Examples of link tables include Transaction Link, which associates sales transactions with customers and products.

Surrogate Keys:

Surrogate keys are artificially generated unique identifiers assigned to records in hub tables.

They serve as primary keys in hub tables and provide a consistent reference point for data integration and querying.

Surrogate keys simplify the process of tracking historical changes and integrating data from multiple source systems.

These keys are often numeric and have no business meaning, allowing for efficient indexing and data management.

Slowly Changing Dimensions (SCDs):

SCDs are techniques used to manage changes to dimension attributes over time.

Data Vault supports different types of SCDs, including Type 1 (overwrite), Type 2 (add new row), and Type 3 (add new column).

Type 1 SCDs overwrite existing attribute values with new values, losing historical information.

Type 2 SCDs add new rows to represent changes, preserving historical data by maintaining multiple versions of the same entity.

Type 3 SCDs add new columns to store both current and historical attribute values, allowing for limited historical analysis.

Overall, the Data Vault modeling approach emphasizes scalability, flexibility, and auditability by structuring data into hubs, satellites, and links, using surrogate keys to maintain integrity and supporting various SCD techniques to manage historical changes effectively.

Tasks to complete Part2:

1. **Design SQL Scripts**:
   * Create SQL scripts to define the structure of the data mart using the Data Vault modeling approach.
   * Design hub tables for customers and products, satellite tables for tracking historical changes to customer and product attributes, and a link table to associate sales transactions with customers and products.
   * Include primary and foreign key constraints where necessary to maintain data integrity.
2. **Data Mart Population**:
   * Populate the data mart with sample data to test the effectiveness of the created SQL scripts.
3. **Incremental Loading**:
   * Modify the scripts to support incremental loading, ensuring that only new records are added to the data mart during each update.
4. **Surrogate Keys**:
   * Implement surrogate keys for hub tables to uniquely identify each record and simplify data integration.
5. **Slowly Changing Dimensions (SCDs)**:
   * Extend the satellite tables to support various types of slowly changing dimensions (e.g., Type 1, Type 2), enabling tracking of historical changes to customer and product attributes.
6. **Partitioning and Compression**:
   * Optimize table storage by incorporating partitioning and compression techniques, improving query performance, and reducing storage requirements.
7. **Error Handling and Logging**:
   * Implement error handling mechanisms in the SQL scripts to handle data validation errors and ensure comprehensive logging of data loading activities.
8. **GitHub Integration**:
   * Utilize GitHub for version control, creating a repository to manage the development of the SQL scripts.
   * Include branches for feature development, issue tracking, and collaborative code review.

**Lets go through them one by one:**

**1- Design SQL Scripts for Data Mart**:

* **Objective**: Create SQL scripts to define the structure of the data mart using the Data Vault modeling approach.
* **Approach**:
  + Utilize SQL to define the tables required for the data mart, adhering to the principles of the Data Vault modeling approach.
  + Design hub tables for customers and products, satellite tables for tracking historical changes to customer and product attributes, and a link table to associate sales transactions with customers and products.
* **Components**:
  + **Hub Tables**: Create hub tables to store unique lists of customers and products.
  + **Satellite Tables**: Develop satellite tables to track historical changes to customer and product attributes, implementing Type 1 or Type 2 slowly changing dimensions (SCDs) as required.
  + **Link Table**: Design a link table to establish associations between sales transactions, customers, and products.
* **Key Features**:
  + **Data Vault Modeling**: Ensure adherence to the principles of Data Vault modeling for scalability, flexibility, and auditability.
  + **Primary and Foreign Key Constraints**: Include primary and foreign key constraints where necessary to maintain data integrity and enforce relationships between tables.
* **Script Development**:
  + Write SQL scripts to create each table, specifying column names, data types, and constraints.
  + Incorporate comments in the scripts to document the purpose of each table and its columns.
* **Testing and Validation**:
  + Validate the correctness of the SQL scripts by executing them against a test database environment.
  + Verify that tables are created as expected and that constraints are enforced properly.
* **Documentation**:
  + Document the SQL scripts, providing an overview of the data mart structure, table definitions, and rationale for design decisions.
  + Include diagrams or visual representations to illustrate the relationships between hub, satellite, and link tables.

**2- Data Mart Population**:

* **Objective**: Populate the data mart with sample data to test the effectiveness of the created SQL scripts.
* **Approach**:
  + Generate sample data representing realistic sales transactions, customers, and products to populate the data mart.
  + Ensure diversity and variability in the generated data to cover different scenarios and edge cases.
* **Components**:
  + **Mock Data Generation**: Develop a script or utility to generate mock data resembling actual sales transactions, including customer information, product details, and transactional attributes.
  + **Data Population Script**: Write SQL scripts or utilities to insert the generated sample data into the corresponding tables in the data mart.
* **Key Features**:
  + **Realistic Data Generation**: Use techniques such as randomization and variation to generate realistic sample data that reflects typical sales scenarios.
  + **Scalability**: Ensure the data generation and population process can handle large volumes of data efficiently.
  + **Data Consistency**: Implement checks to ensure the consistency and integrity of the populated data, including validation of foreign key relationships and adherence to data constraints.
* **Script Development**:
  + Generate sample data using Python scripts, libraries like Faker, or custom data generation tools tailored to the specific requirements of the data mart.
  + Write SQL scripts or utilities to insert the generated data into the corresponding tables, following best practices for data loading and bulk insertion.
* **Testing and Validation**:
  + Execute the data population scripts against a test database environment to verify successful data insertion.
  + Validate that the populated data matches the expected structure and contains the required attributes.
* **Documentation**:
  + Document the data generation process, including details of the tools or scripts used, parameters for data generation, and considerations for scalability and performance.
  + Provide instructions for executing the data population scripts and guidelines for validating the populated data.

**3- Incremental Loading**:

* **Objective**: Modify the scripts to support incremental loading, ensuring that only new records are added to the data mart during each update.
* **Approach**:
  + Implement a mechanism to identify new records since the last update and selectively load them into the data mart.
  + Utilize timestamps or incremental identifiers to track the latest updates and determine the subset of data to be loaded.
* **Components**:
  + **Change Detection**: Develop logic to identify new or updated records in the source data since the last load operation.
  + **Incremental Loading Script**: Modify the existing SQL scripts or utilities to incorporate incremental loading logic, focusing on efficient data extraction and loading mechanisms.
* **Key Features**:
  + **Efficiency**: Ensure that the incremental loading process is optimized for performance and can handle large volumes of data without significant overhead.
  + **Data Consistency**: Implement mechanisms to maintain data consistency and integrity during incremental updates, including handling of conflicts and duplicate records.
* **Script Development**:
  + Enhance the existing data extraction and loading scripts to include logic for detecting and processing incremental changes.
  + Implement query optimizations and indexing strategies to improve the efficiency of change detection and data loading operations.
* **Testing and Validation**:
  + Conduct rigorous testing to verify the accuracy and reliability of the incremental loading process under various scenarios, including scenarios with overlapping data updates.
  + Validate the consistency of the loaded data and ensure that only new or updated records are added to the data mart during incremental updates.
* **Documentation**:
  + Document the incremental loading process, including details of the change detection mechanisms, strategies for handling incremental updates, and considerations for scalability and performance.
  + Provide instructions for configuring and executing the incremental loading scripts, along with guidelines for monitoring and troubleshooting incremental loading processes.

**4- Surrogate Keys**:

* **Objective**: Implement surrogate keys for hub tables to uniquely identify each record and simplify data integration.
* **Approach**:
  + Introduce surrogate keys as primary identifiers for hub tables to ensure uniqueness and facilitate data integration and referencing.
  + Use auto-incrementing or UUID (Universally Unique Identifier) values as surrogate keys to assign a unique identifier to each record.
* **Components**:
  + **Surrogate Key Generation**: Develop logic to generate surrogate keys for hub tables during data loading or initialization.
  + **Integration with Data Model**: Modify the table definitions to include surrogate key columns and establish relationships with satellite and link tables.
* **Key Features**:
  + **Uniqueness**: Ensure that each surrogate key is unique across the hub table to prevent conflicts and maintain data integrity.
  + **Simplicity**: Simplify data integration and referencing by using surrogate keys as primary identifiers for hub tables.
* **Script Development**:
  + Update the SQL scripts to include definitions for surrogate key columns in hub tables, specifying appropriate data types and constraints.
  + Incorporate logic to generate surrogate key values during data loading processes, ensuring uniqueness and consistency.
* **Testing and Validation**:
  + Validate the generation and usage of surrogate keys by verifying the uniqueness of surrogate key values and their proper association with data records.
  + Conduct integration testing to ensure that surrogate keys are effectively utilized in referencing relationships between hub, satellite, and link tables.
* **Documentation**:
  + Document the implementation of surrogate keys, including details of the surrogate key generation process, considerations for data integration, and guidelines for using surrogate keys in queries and data manipulation operations.
  + Provide instructions for configuring and executing the SQL scripts to initialize or update hub tables with surrogate key columns.

**5- Slowly Changing Dimensions (SCDs)**:

* **Objective**: Extend the satellite tables to support various types of slowly changing dimensions (e.g., Type 1, Type 2), enabling tracking of historical changes to customer and product attributes.
* **Approach**:
  + Implement different strategies for handling slowly changing dimensions based on the type of change (Type 1 or Type 2) and business requirements.
  + Utilize satellite tables to maintain historical records of attribute changes while preserving data integrity and query performance.
* **Components**:
  + **SCD Types**: Define strategies for handling Type 1 and Type 2 slowly changing dimensions based on business requirements.
  + **Satellite Table Design**: Enhance the design of satellite tables to accommodate historical attribute tracking, including effective start and end dates for each record.
* **Key Features**:
  + **Type 1 SCD**: Overwrite existing attribute values with new values without preserving historical data, suitable for attributes that do not require historical tracking.
  + **Type 2 SCD**: Maintain historical records of attribute changes by introducing new rows for each change, allowing analysis of data changes over time.
* **Script Development**:
  + Update the SQL scripts to incorporate logic for handling Type 1 and Type 2 SCDs in satellite tables, including insertion, updating, and expiry of records.
  + Implement triggers or stored procedures to automate the process of capturing and managing historical changes in satellite tables.
* **Testing and Validation**:
  + Test the implementation of SCD strategies by simulating attribute changes and verifying the behavior of the ETL process in capturing and preserving historical data.
  + Validate query results to ensure accurate retrieval of historical records from satellite tables based on specified criteria.
* **Documentation**:
  + Document the SCD strategies employed (Type 1, Type 2) and their impact on data storage, query performance, and analytical capabilities.
  + Provide guidelines for configuring and using SCD functionality in SQL scripts, including considerations for data consistency and maintenance.

**6- Partitioning and Compression**:

* **Objective**: Optimize table storage by incorporating partitioning and compression techniques, improving query performance, and reducing storage requirements.
* **Approach**:
  + Implement partitioning to divide large tables into smaller, more manageable partitions based on predefined criteria such as date ranges or key values.
  + Utilize compression to reduce the size of data stored in tables, leading to decreased storage requirements and improved I/O performance.
* **Components**:
  + **Partitioning**: Define partitioning strategies based on the characteristics of the data and query patterns, such as range partitioning or list partitioning.
  + **Compression**: Select appropriate compression algorithms (e.g., gzip, lz4) supported by the database system and apply them to tables or specific columns.
* **Key Features**:
  + **Partitioning Benefits**: Improve query performance by limiting the data scanned for each query, enabling partition pruning and reducing I/O overhead.
  + **Compression Benefits**: Reduce storage costs and optimize disk space utilization by compressing data stored in tables, indexes, or individual columns.
* **Script Development**:
  + Modify the SQL scripts to include partitioning clauses in table definitions, specifying partition keys and ranges.
  + Integrate compression options into table creation scripts or alter existing tables to apply compression to specified columns or partitions.
* **Testing and Validation**:
  + Evaluate the impact of partitioning and compression on query performance, storage utilization, and resource consumption using benchmarking and profiling techniques.
  + Verify the correctness of partitioning and compression configurations by executing test queries and analyzing execution plans.
* **Documentation**:
  + Document the partitioning and compression strategies implemented, including rationale, benefits, and considerations for maintenance and performance tuning.
  + Provide guidelines for monitoring and managing partitioned tables and compressed data, including best practices for ongoing optimization.

**7- Error Handling and Logging**:

* **Objective**: Implement error handling mechanisms in the SQL scripts to manage data validation errors effectively and ensure comprehensive logging of data loading activities.
* **Approach**:
  + Develop robust error handling logic to detect and handle exceptions, ensuring graceful recovery from failures during data loading.
  + Utilize logging mechanisms to record relevant information about errors, warnings, and status updates for monitoring and troubleshooting purposes.
* **Components**:
  + **Error Handling**: Implement try-catch blocks or exception handling routines to capture and manage errors encountered during SQL script execution.
  + **Logging**: Utilize logging facilities provided by the database system to write log messages to designated log files or tables.
* **Key Features**:
  + **Exception Handling**: Handle common error scenarios such as data type mismatches, constraint violations, and connectivity issues with external systems.
  + **Comprehensive Logging**: Log detailed information about the execution flow, including timestamps, error codes, affected rows, and contextual metadata.
* **Script Development**:
  + Integrate error handling logic into SQL scripts using appropriate error handling constructs supported by the database system (e.g., TRY...CATCH in SQL Server).
  + Configure logging settings to define log levels, output destinations, and log rotation policies based on project requirements.
* **Testing and Validation**:
  + Validate error handling mechanisms by deliberately inducing errors in test environments and verifying that they are detected and handled correctly.
  + Analyze log files or tables to ensure that log messages are generated as expected and contain relevant information for diagnosing issues.
* **Documentation**:
  + Document the error handling strategy adopted, including types of errors handled, error codes, and recovery procedures.
  + Provide guidance on interpreting log messages and troubleshooting common issues encountered during data loading operations.

**8- GitHub Integration**:

* **Objective**: Utilize GitHub for version control, enabling collaborative development, change tracking, and codebase management for the SQL scripts.
* **Approach**:
  + Establish a GitHub repository to host the SQL scripts and associated documentation.
  + Adopt version control best practices, including branching strategies, pull requests, and issue tracking, to facilitate collaborative development and code review.
* **Repository Setup**:
  + Create a new repository on GitHub dedicated to the data mart project, ensuring appropriate access controls and repository settings.
  + Initialize the repository with an appropriate README file outlining the project overview, objectives, and instructions for contributors.
* **Branching Strategy**:
  + Define a branching strategy that aligns with the project's development workflow, such as GitFlow or a simplified feature branching model.
  + Create branches for feature development, bug fixes, and experimentation, ensuring isolation of changes and logical grouping of related commits.
* **Pull Requests and Code Review**:
  + Encourage contributors to create pull requests (PRs) for proposing changes to the main branch, providing a mechanism for peer review and feedback.
  + Review PRs thoroughly, examining code changes, ensuring adherence to coding standards, and verifying the impact on project objectives.
* **Issue Tracking**:
  + Utilize GitHub's issue tracking features to manage tasks, bugs, and feature requests effectively.
  + Create and prioritize issues based on project requirements, assigning tasks to team members and tracking their progress throughout the development lifecycle.
* **Collaborative Development**:
  + Foster collaboration among team members by encouraging discussions, sharing knowledge, and leveraging GitHub's collaboration features such as comments and mentions.
  + Coordinate development efforts through the use of project boards, milestones, and labels to organize and track tasks.
* **Documentation**:
  + Maintain up-to-date documentation within the repository, including README files, contribution guidelines, and setup instructions to onboard new contributors.
  + Document decisions, discussions, and resolutions within GitHub issues and PRs to provide context and aid in knowledge sharing.

Explanation of directory structure:

* **sql\_scripts/**: Contains SQL scripts for creating tables, implementing incremental loading, and any other database operations.
* **data/**: Stores sample data files (e.g., CSV files) used for testing and populating the database.
* **documentation/**: Holds project documentation, including data modeling design, data dictionary, and any other relevant documents.
* **tests/**: Houses unit tests and integration tests to ensure the correctness of the ETL pipeline.
* **src/**: Contains Python source code for various components of the ETL pipeline, such as data loading, transformation, incremental loading, database connection, error handling, etc.
* **requirements.txt**: Lists project dependencies, making it easier to recreate the project environment.
* **README.md**: Provides an overview of the project, its purpose, installation instructions, and usage guidelines.
* **.gitignore**: Specifies files and directories to be ignored by Git version control system.

**1- Design SQL Scripts**:

**Customers Hub Table:**

-- Create customers\_hub table with hash key

CREATE TABLE customers\_hub (

    customer\_hash\_key CHAR(32) PRIMARY KEY DEFAULT MD5(customer\_id::TEXT),

    customer\_id SERIAL

);

**Explanation:**

* **customer\_hash\_key**: Primary key representing the MD5 hash of the **customer\_id**.
* **customer\_id**: Unique identifier for customers (auto-incrementing serial).

**Products Hub Table:**

-- Create products\_hub table with hash key

CREATE TABLE products\_hub (

    product\_hash\_key CHAR(32) PRIMARY KEY DEFAULT MD5(product\_id::TEXT),

    product\_id SERIAL

);

**Explanation:**

* **product\_hash\_key**: Primary key representing the MD5 hash of the **product\_id**.
* **product\_id**: Unique identifier for products (auto-incrementing serial).

**Customers Satellite Table:**

-- Create customers\_satellite table with hash difference and related fields

CREATE TABLE customers\_satellite (

    customer\_hash\_key CHAR(32) REFERENCES customers\_hub(customer\_hash\_key),

    hash\_diff CHAR(32) DEFAULT MD5(customer\_name || customer\_email || new.customer\_address) GENERATED ALWAYS AS (MD5(customer\_name || customer\_email || customer\_address)) STORED,

    customer\_name VARCHAR(100),

    customer\_email VARCHAR(100),

    customer\_address VARCHAR(255),

    start\_date DATE,

    end\_date DATE,

    load\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

    source VARCHAR(50)

);

**Explanation:**

* **customer\_hash\_key**: Foreign key referencing the **customer\_hash\_key** in the **customers\_hub** table.
* **customer\_name**: Name of the customer.
* **customer\_email**: Email address of the customer.
* **customer\_address**: Address of the customer.
* **start\_date**: Start date indicating when the attribute values became effective.
* **end\_date**: End date indicating when the attribute values ceased to be effective.
* **load\_date**: Timestamp indicating when the record was loaded into the satellite.
* **source**: Source system providing the data.
* **hash\_diff**: Hash difference representing changes in customer attributes.

**Products Satellite Table:**

-- Create products\_satellite table with hash difference and related fields

CREATE TABLE products\_satellite (

    product\_hash\_key CHAR(32) REFERENCES products\_hub(product\_hash\_key),

    hash\_diff CHAR(32) DEFAULT MD5(product\_name || product\_category || product\_brand) GENERATED ALWAYS AS (MD5(product\_name || product\_category || new.product\_brand)) STORED,

    product\_name VARCHAR(100),

    product\_category VARCHAR(50),

    product\_brand VARCHAR(50),

    start\_date DATE,

    end\_date DATE,

    load\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

    source VARCHAR(50)

);

**Explanation:**

* **product\_hash\_key**: Foreign key referencing the **product\_hash\_key** in the **products\_hub** table.
* **product\_name**: Name of the product.
* **product\_category**: Category of the product.
* **product\_brand**: Brand of the product.
* **start\_date**: Start date indicating when the attribute values became effective.
* **end\_date**: End date indicating when the attribute values ceased to be effective.
* **load\_date**: Timestamp indicating when the record was loaded into the satellite.
* **source**: Source system providing the data.
* **hash\_diff**: Hash difference representing changes in product attributes.

**Sales Link Table:**

-- Create sales\_link table with hash keys referencing customers\_hub and products\_hub

CREATE TABLE sales\_link (

    transaction\_hash\_key CHAR(32) PRIMARY KEY DEFAULT MD5(CONCAT(customer\_id::TEXT, product\_id::TEXT)),

    customer\_hash\_key CHAR(32) REFERENCES customers\_hub(customer\_hash\_key),

    product\_hash\_key CHAR(32) REFERENCES products\_hub(product\_hash\_key),

    transaction\_date DATE,

    transaction\_amount NUMERIC(10, 2),

    load\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

    source VARCHAR(50)

);

**Explanation:**

* **transaction\_hash\_key**: Primary key representing the MD5 hash of the concatenated **customer\_id** and **product\_id**.
* **customer\_hash\_key**: Foreign key referencing the **customer\_hash\_key** in the **customers\_hub** table.
* **product\_hash\_key**: Foreign key referencing the **product\_hash\_key** in the **products\_hub** table.
* **transaction\_date**: Date of the sales transaction.
* **transaction\_amount**: Amount of the sales transaction.
* **load\_date**: Timestamp indicating when the record was loaded into the table.
* **source**: Source system providing the data.
* **hash\_diff**: Hash difference representing changes in transaction attributes.

**Sales Sales transactions satellite Table:**

CREATE TABLE sales\_transactions\_satellite (

    transaction\_hash\_key CHAR(32) PRIMARY KEY DEFAULT MD5(CONCAT(customer\_id::TEXT, product\_id::TEXT)),

    start\_date DATE,

    end\_date DATE,

    load\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

    source VARCHAR(50),

    hash\_diff CHAR(32) DEFAULT MD5(load\_date || source) GENERATED ALWAYS AS (MD5(load\_date || source)) STORED,

    CONSTRAINT fk\_sales\_link FOREIGN KEY (transaction\_hash\_key) REFERENCES sales\_link(transaction\_hash\_key)

);

The **sales\_transactions\_satellite** table is designed to store additional information related to sales transactions, serving as a satellite table directly connected to the **sales\_link** table. Here's a detailed explanation of why and how it's implemented:

1. **Purpose**:
   * The primary purpose of the **sales\_transactions\_satellite** table is to capture metadata and contextual information about sales transactions.
   * By directly connecting it to the **sales\_link** table, we establish a one-to-one relationship between each sales transaction and its corresponding satellite data.
2. **Structure**:
   * **transaction\_hash\_key**: This field serves as the primary key for the **sales\_transactions\_satellite** table. It's generated using the MD5 hashing algorithm applied to the concatenation of the **customer\_id** and **product\_id** fields converted to text. This approach ensures a unique identifier for each transaction.
   * **start\_date**: Indicates the start date of the validity period for the sales transaction. It represents the date from which the transaction is considered active in the system.
   * **end\_date**: Indicates the end date of the validity period for the sales transaction. It represents the date until which the transaction is considered active in the system.
   * **load\_date**: This column captures the timestamp indicating when the record was loaded into the satellite table. It defaults to the current timestamp (**CURRENT\_TIMESTAMP**).
   * **source**: This field specifies the source system or origin of the data. It provides context about where the transaction data originated from.
   * **hash\_diff**: Similar to other satellite tables, the **hash\_diff** field represents the hash difference calculated based on relevant fields (**load\_date** and **source**). It's generated using the MD5 hashing algorithm applied to the concatenation of these fields.
   * **CONSTRAINT fk\_sales\_link**: This constraint establishes a foreign key relationship between the **transaction\_hash\_key** field in the **sales\_transactions\_satellite** table and the **transaction\_hash\_key** field in the **sales\_link** table. It ensures referential integrity between the two tables.
3. **Connection to sales\_link**:
   * Directly connecting the **sales\_transactions\_satellite** table to the **sales\_link** table allows for a streamlined relationship between transaction data and its associated metadata.
   * This design choice simplifies querying and analysis, as all relevant information about a sales transaction can be retrieved from a single source.
   * Additionally, it enables efficient retrieval of historical data and tracking of changes over time, as each transaction's metadata is closely linked to its primary data record.
4. **Hashing and Hash Difference**:
   * Hashing techniques are employed to generate unique identifiers (**transaction\_hash\_key**) and hash differences (**hash\_diff**) for each record in the satellite table.
   * MD5 hashing is used for its cryptographic properties and efficiency in generating unique hash values.
   * The **hash\_diff** field provides a checksum or fingerprint of the satellite data, allowing for quick validation of data integrity and identification of changes between records.

Overall, the **sales\_transactions\_satellite** table serves as a critical component in the data model, facilitating comprehensive analysis and tracking of sales transaction data within the data mart. Its direct connection to the **sales\_link** table enhances data integrity, query performance, and analytical capabilities.

These SQL scripts define the tables required for the data mart, incorporating hash keys for primary and foreign key constraints to ensure data integrity and efficient referencing between tables. Additionally, hash\_diff fields are included in satellite tables for tracking changes in attribute values, and load\_date and source fields are included for auditing and tracking purposes.

**Hub Tables:**

* **Primary Key as Hash Key**: Using the MD5 hash of the ID field (customer\_id for customers\_hub and product\_id for products\_hub) as the primary key ensures uniqueness and provides a consistent, stable identifier for each record. MD5 hashing helps maintain data integrity and avoids the exposure of raw IDs.
* **Serial ID**: While the primary key is derived from the MD5 hash of the ID, a separate auto-incrementing serial ID (customer\_id for customers\_hub and product\_id for products\_hub) is included for internal reference and ease of querying.

**Satellite Tables:**

* **Foreign Key Reference**: The foreign key references (customer\_hash\_key for customers\_satellite and product\_hash\_key for products\_satellite) link each satellite record to the corresponding hub record, ensuring referential integrity and enabling easy joins between hub and satellite tables.
* **Hash Difference Field**: The hash\_diff field captures the MD5 hash of concatenated attribute values (e.g., customer\_name, customer\_email, customer\_address for customers\_satellite) to detect changes in attribute values over time. This allows for efficient tracking of historical changes and facilitates incremental loading.

**Sales Link Table:**

* **Primary Key as Transaction Hash Key**: Using the MD5 hash of the concatenated customer\_id and product\_id as the primary key (transaction\_hash\_key) ensures uniqueness and provides a stable identifier for each sales transaction. This approach avoids the need for a separate serial ID and ensures efficient querying and referencing.
* **Foreign Key References to Hub Tables**: The customer\_hash\_key and product\_hash\_key fields in the sales\_link table reference the corresponding hash keys in the customers\_hub and products\_hub tables, maintaining referential integrity and enabling easy association of sales transactions with customers and products.
* **Hash Difference Field**: The hash\_diff field captures the MD5 hash of the concatenated customer\_hash\_key and product\_hash\_key, allowing for efficient detection of changes in the association between customers and products over time.
* **Load Date and Source Fields**: The load\_date field captures the timestamp when each record is loaded into the table, facilitating auditing and tracking of data loading activities. The source field identifies the source system providing the data, aiding in data lineage and provenance.

**Reasoning for Decisions:**

* **Data Integrity**: The use of hash keys ensures data integrity by providing unique, stable identifiers that are resistant to manipulation or exposure. MD5 hashing provides a fast and reliable method for generating hash keys.
* **Efficiency**: Hash keys and hash difference fields enable efficient detection of changes in attribute values and associations, supporting historical tracking and incremental loading without the need for complex comparison logic.
* **Referential Integrity**: Foreign key constraints maintain referential integrity between hub and satellite tables, ensuring that each satellite record is associated with a valid hub record.
* **Auditing and Tracking**: Load date and source fields allow for comprehensive auditing and tracking of data loading activities, supporting data lineage, provenance, and compliance requirements.

These design decisions aim to optimize data modeling for the data mart, balancing data integrity, efficiency, and usability considerations.

1. **Data Writing Function**:
   * **write\_to\_csv**: This function writes the generated data to CSV files. It takes two arguments: the data to be written (a list of lists) and the filename for the CSV file. It uses the **csv.writer** class to write the data to the CSV file.
2. **Data Generation Process**:
   * First, mock customer data is generated using the **generate\_customer\_data** function, creating a list of lists where each inner list represents a customer record. This data is then written to a CSV file named **customer\_data.csv**.
   * Next, mock product data is generated using the **generate\_product\_data** function, creating a list of lists where each inner list represents a product record. This data is then written to a CSV file named **product\_data.csv**.
   * Then, mock sales data is generated using the **generate\_sales\_data** function. This data includes random customer and product IDs from the previously generated data. It creates a list of lists where each inner list represents a sales transaction record. This data is then written to a CSV file named **sales\_data.csv**.

recently, the standard practice is beginning to shift towards using hash keys instead of sequence values to assign surrogate keys, particularly within the data vault 2.0 approach to data warehousing.

A hash key is the output from a hashing algorithm, where a specific input value is transformed into a distinct, unique string per input value.