

Performance optimization and transaction processing for the Swansea art store database system.

Name:Yijun Kuang

Abstract : This report creates a database system for the Swansea art store, proposing strategies for performance optimization and transaction processing. A relatively comprehensive Swansea art store database system is established, with the addition of secondary indexing to optimize database performance, and the selection of appropriate transaction isolation methods to enhance the efficiency of the system.

Table of Contents

1.Database design	1
1.1 Entity Attributes	1
1.2Relationships.....	2
2. Database performance tuning	3
2.1 Index Optimization	4
2.2 Query Optimization	4
3. Transaction processing	5
3.1 Rollback and Commit	7
3.2 Triggers	8
3.3 Functions.....	8
4. Database Design and Implementation Assessment.....	9
4.1 Database Structure Design Assessment.....	9
4.2 Performance Optimization Assessment	9
4.3 Transaction Processing Assessment	9
5.Conclusion	10
Reference	12

1.Database design

The following text, as per the requirements, creates a comprehensive and suitable database for the Swansea Art Store. It involves triggers, functions, and indexes. Furthermore, this document experimentally utilizes the PostgreSQL database and pgAdmin 4 to accomplish this task. The drawing of the ER diagram was completed using the online tool Draw.io.

The following will introduce the basic entities of the database along with their respective attributes and relationships:

According to the requirements, the database is designed with nine categories, namely Artists, Artwork, ArtworkCategories, Categories, Customers, Feedback, Orders, Orderdetails, and Auditlog, the ER diagram is as shown in Figure 1.

1.1 Entity Attributes

Artists: This table stores information about artists in the Swansea Art Store. It includes the following attributes:

Primary Key: ArtistID

Artist Name: Name

Artist Biography: Biography

Artwork: This table contains information about artworks. Its attributes include:

Primary Key: ArtworkID

Artwork Title: Title

Artwork Description: Description

Artwork Price: Price

Artwork Average Rating: AverageRating

Foreign Key: ArtistID (relating to Artists)

ArtworkCategories: This is a junction table for categorizing artworks and has the following attributes:

Foreign Key: ArtworkID (relating to Artwork)

Foreign Key: CategoryID (relating to Categories)

Categories: This table is used for categorizing artworks and has the following attributes:

Primary Key: CategoryID

Category Name: Name

Customers: This table stores information about customers who purchase artworks. It includes the following attributes:

Primary Key: CustomerID

Customer Name: Name

Customer Address: Address

Customer Email: Email

Feedback: This table is for customer feedback on artworks and includes the following attributes:

Primary Key: FeedbackID

Rating: Rating

Comment: Comment

Foreign Key: CustomerID (relating to Customers)

Foreign Key: ArtworkID (relating to Artwork)

Orders: This table records customer orders and includes the following attributes:

Primary Key: OrderID

Order Date: OrderDate

Total Amount: TotalAmount

Foreign Key: CustomerID (relating to Customers)

OrderDetails: This table supplements order information and includes the following attributes:

Primary Key: OrderID

Quantity: Quantity

Foreign Key: ArtworkID (relating to Artwork)

Auditlog: This table logs changes made to information in the database and includes the following attributes:

Primary Key: AuditID

Table Name: TableName

Operation Type: OperationType

Operation Timestamp: OperationTimestamp

1.2 Relationship

The relationships between the tables are as follows:

1. Artists: each artist can have multiple artworks, One-to-Many relationship with Artwork.
2. Artwork: each artwork belongs to one artist, Many-to-One relationship with Artists. An artwork can belong to multiple categories, Many-to-Many relationship with Categories through ArtworkCategories; each artwork can have multiple feedback entries, One-to-Many relationship with Feedback; each artwork can be included in multiple order details One-to-Many relationship with OrderDetails.
3. Categories: a category can be associated with multiple artworks, Many-to-Many relationship with Artwork through ArtworkCategories .
4. Customers: each customer can provide multiple feedback entries, One-to-Many relationship with Feedback; each customer can place multiple orders, One-to-Many relationship with Orders.

5.Feedback: each feedback entry is given by one customer, Many-to-One relationship with Customers; each feedback entry is related to one artwork, Many-to-One relationship with Artwork.

6.Orders: each order is placed by one customer, Many-to-One relationship with Customers; each order can have multiple order details, One-to-Many relationship with OrderDetails.

7.OrderDetails: each order detail is related to one artwork, Many-to-One relationship with Artwork; each order detail belongs to one order, Many-to-One relationship with Orders.

8.Auditlog: no direct relationships, but it records changes made to other tables in the database.

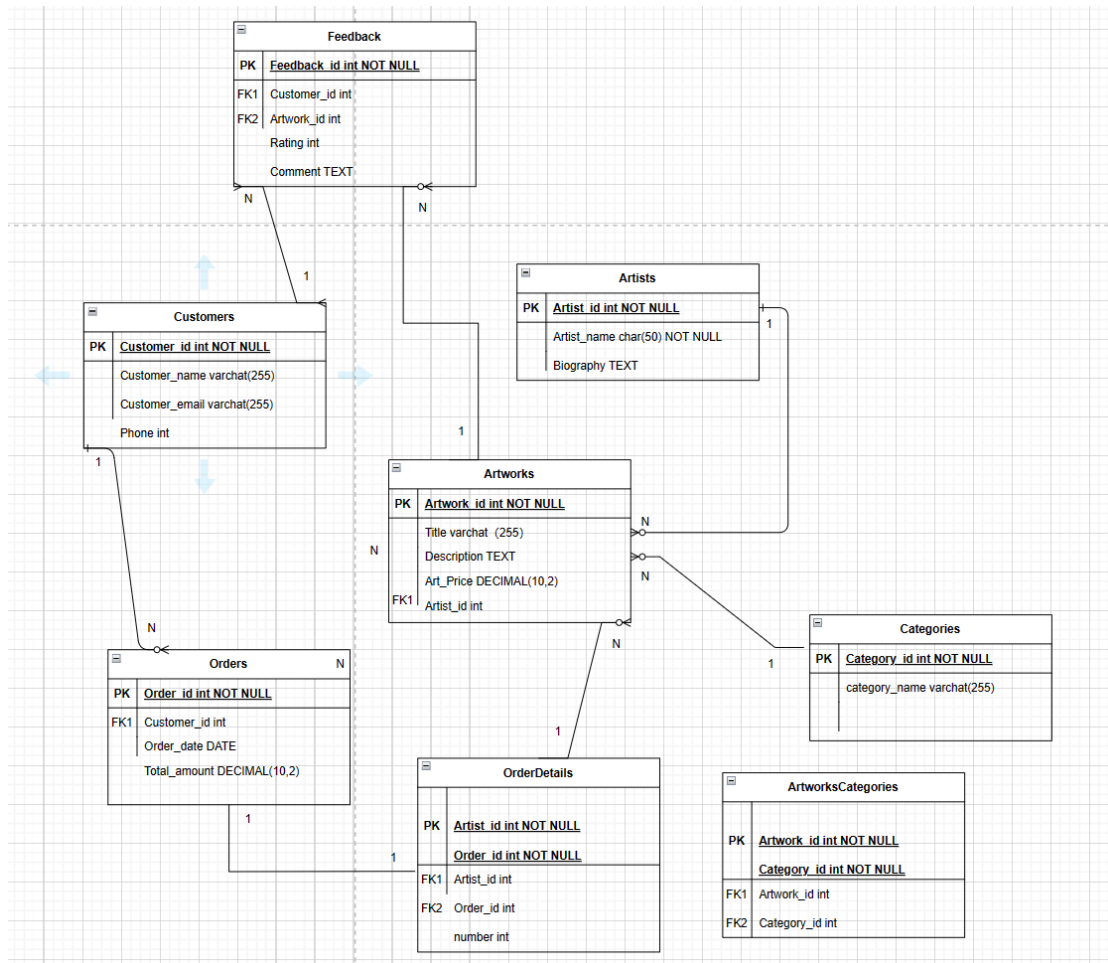


Fig.1 Swansea Art Gallery ER Diagram.

2. Database performance tuning

Database performance tuning is a process of enhancing its performance by optimizing the configuration, structure, and queries of the database system. The primary goal is to ensure the database processes data information more efficiently to meet system requirements. Performance tuning includes query optimization, index optimization, database optimization, hardware and server optimization, caching mechanisms, resource management, and data cleaning and maintenance. In this database, index optimization and

query optimization are mainly used.

2.1 Index Optimization

1. Index optimization is one of the most effective methods to improve database query performance. It involves creating and maintaining database indexes to speed up data retrieval. An index is a data structure in a database that helps users quickly find and retrieve the required information in a table without scanning all the data in the table repeatedly. This greatly improves query efficiency and reduces server load. In this system, customers need to search for information about the art they want to buy, artist information, and art ratings. Therefore, it is necessary to create indexes for the necessary fields of the tables. Five secondary indexes were mainly created to optimize the entire database.

2. Indexing artist names: Based on the artist table, indexing the artist name field. In this system, some customers have their favorite artists, and they search for corresponding artworks based on the artist's name. Therefore, indexing artist names can greatly improve system efficiency and reduce server pressure.

3. Indexing artwork titles: Searching the title field in the artwork table, customers can quickly find the corresponding artwork information based on the keywords of the artwork they want.

4. Indexing artwork prices: Searching the price field in the artwork table. When customers search for desired artworks, the database contains many similar artworks, which is not conducive to customer searches. Therefore, searching for artworks in the customer's desired price range can greatly enhance system speed.

5. Indexing feedback ratings: Searching the rating field in the feedback table. When customers need to find high-rated artworks, this index can facilitate their use and enhance the query function.

6. Indexing order dates: Searching the date field in the order table. When customers or artists need to query the date of orders, this index can improve database performance.

2.2 Query Optimization

Query optimization is an important aspect of database performance optimization. It involves designing, modifying, and improving the way database queries are made to enhance the accuracy and speed of queries. The goal is to reduce the time and resource consumption in the database during query execution, ensuring the completeness and accuracy of query results. The use of indexes mentioned above can improve the efficiency and accuracy of queries. In addition, query optimization can be achieved by optimizing joins and streamlining query methods.

1. Streamlining queries: Streamlining queries makes the query statements more accurate, with as few data tables as possible and minimal resource consumption. Therefore, avoid using select* in query statements to reduce global query statements. Instead, specify the necessary columns to reduce the amount of data transmitted, especially when dealing with large tables and joining multiple tables.

2. Optimizing joins: In this database, some tables involve joining multiple tables, such as

orders and order details, artwork, artists, and feedback tables. When queries involve joining multiple tables, arranging the order and type of joins can improve query efficiency, such as using inner join, left join, etc. Usually, the PostgreSQL database optimizer will handle some information.

Moreover, regular maintenance of the database should be conducted, such as updating statistical information, deleting and cleaning up fragmentary information, and optimizing and updating indexes and queries for tables added based on system improvement and expansion, to ensure the database is always in its best performance state.

Through the above methods, the query efficiency and overall performance of the Swansea Art Store database can be significantly improved. However, it is worth noting that performance optimization is an ongoing process, and database performance should be regularly assessed and adjusted accordingly

3.Transaction processing

Transactions in a database management system are a series of operations that involve one or more read or write operations to the database. These operations are atomic, meaning they are either fully executed after the transaction begins or completely rolled back, not executed at all. Transactions are the basic unit of data operation, ensuring the integrity and consistency of data in the database. Transactions typically have four steps: starting the transaction with 'BEGIN', executing a series of internal operations such as adding or modifying data in the database, determining whether there are errors in the transaction and if so, rolling back with 'ROLLBACK' to undo the series of operations, and finally committing the transaction with 'COMMIT' to confirm its completion.

Transactions have four main characteristics, also known as ACID properties:

1. Atomicity: All operations within a transaction must be uniform, meaning either all operations are written to the database, or all fail, with no partial operations written.
2. Consistency: Transactions must ensure the database transitions from one consistent state to another consistent state, adhering to all predetermined rules, maintaining database completeness and unity.
3. Isolation: The execution of an independent transaction is not affected by any other transactions, and the results of multiple concurrent transactions must be consistent with their sequential execution order.
4. Durability: Once a transaction is successfully executed and committed, its effects on the database are permanently saved internally, and changes are not lost even in the event of system failure.

In the Swansea Art Store's database system, three main transactions were tested: customer order submission, customer art rating, and customer information update. The customer order submission involves inserting a record into the order table and, due to the association between orders and order details, also inserting corresponding data into the order details table. As the order table contains a total amount, a function is needed to automatically calculate this total. The customer art rating transaction involves adding a record to the

feedback table. Since the art table contains an average rating field, a trigger is needed to calculate the ratings from the feedback table and reflect them in the art table. Finally, the customer information update transaction involves changing one or more pieces of information in the customer table for easy administration. the results of the three transactions are demonstrated in Figures 2 to 5.

	orderid [PK] integer	customerid integer	orderdate date	totalamount numeric (10,2)
1	1	1	2024-01-01	3000.00
2	2	2	2024-01-02	5000.00
3	3	3	2023-11-23	2000.00
4	4	4	2023-11-11	9000.00
5	5	5	2022-11-12	22000.00
6	6	7	2023-12-01	24000.00
7	7	8	2022-12-20	3000.00
8	8	9	2023-08-02	6000.00
9	9	11	2023-09-13	9000.00
10	10	12	2023-03-13	3000.00
11	11	13	2022-11-24	3000.00
12	12	1	2024-01-10	5000.00
13	14	1	2024-01-05	1000.00
14	15	1	2024-01-06	0.00
15	17	1	2024-01-06	3000.00
16	18	1	2024-01-06	3000.00
17	19	1	2024-01-07	3000.00

Fig. 2 Order diagram after executing the customer order transaction

	orderid [PK] integer	artworkid [PK] integer	quantity integer
1	1	1	1
2	2	2	1
3	3	3	1
4	4	4	1
5	5	6	1
6	6	5	2
7	7	9	3
8	8	11	3
9	9	7	1
10	10	10	1
11	11	12	1
12	15	1	1
13	17	1	1
14	18	1	1
15	19	1	1

Fig. 3 OrderDetails diagram after executing the customer order transaction.

	feedbackid [PK] integer	customerid integer	artworkid integer	rating integer	comment text
1	1	1	1	5	This is such a beautiful picture...
2	2	2	2	4	I like it.
3	3	3	3	3	I think its average.
4	4	4	4	3	It met my expectations.
5	5	5	5	5	Great. A real work of art.
6	6	6	6	3	I think its very common.
7	7	7	7	2	I dont think its worth that much.
8	8	8	8	4	Not bad.
9	9	9	9	1	This sucks. Get your money ba...
10	10	10	10	4	Very nice work.
11	11	11	11	5	I really, really like this one.
12	13	1	1	5	Excellent piece!
13	14	1	12	5	Excellent piece!
14	15	1	12	5	transaction test Excellent piece!

Fig. 4 Feedback table after executing the customer feedback transaction

	customerid [PK] integer	name character varying (255)	email character varying (255)
1	1	new customer	test email
2	2	Emma Smith	emma.smith@example.com
3	3	Michael Johnson	michael.johnson@example.com
4	4	Sarah Brown	sarah.brown@example.com
5	5	Olivia Davis	olivia.davis@example.com
6	6	William Martinez	william.martinez@example.com
7	7	Sophia Garcia	sophia.garcia@example.com

Fig. 5 Customer table after executing the transaction to modify customer information

3.1 Rollback and Commit

Transaction rollback and commit are fundamental elements of transactions, playing an important role in maintaining the consistency and integrity of data. Committing means, after the completion of a transaction, submitting the changed data to be stored in the database. This ensures data persistence, maintains data consistency, is a key step in concurrent control, and ensures that each transaction is accurately processed when multiple users or processes access the database simultaneously, preventing data conflicts and inconsistencies. It can also trigger constraint checks and trigger execution. Moreover, only committed transactions can be recovered and re-executed in the event of a system failure, as part of the recovery strategy. Rollback refers to the process of reverting the database to its state before the transaction started when encountering errors or needing to undo certain operations within a transaction. This guarantees data consistency and integrity, supports transaction atomicity and isolation levels, and in some isolation levels, requires rollback when transactions read inconsistent or unsatisfactory data to maintain data consistency. Finally, it allows for error recovery, as rollback operations can undo interrupted transactions in the event of system errors or logical errors, thus avoiding the recording of erroneous data.

3.2 Triggers

A trigger is a special stored procedure that automatically executes in response to specific events and conditions, typically represented by insert, update, or delete operations. Its importance lies in the following aspects:

1. **Maintaining Data Integrity and Consistency:** Triggers automatically execute rules and policies to ensure data operations do not violate database integrity constraints. For example, in this database, the average score data trigger is activated when new data is inserted into the feedback table, automatically calculating and writing the average score to the corresponding field in the art table, ensuring data accuracy.
2. **Automating Tasks:** Triggers can automatically execute tasks that need to occur after specific database operations. In this database, creating an audit log table, where each operation triggers an automatic entry.
3. **Security:** Triggers can enhance the security of data access and modification, blocking unauthorized data modification or automatically auditing sensitive operations.
4. **Real-Time Response:** Triggers can react immediately when data changes, maintaining database integrity, which is very important when real-time monitoring of the database is needed.
5. **Supporting Complex Logic:** Triggers can implement and handle logic that is complex or cannot be achieved with SQL statements.

Test the audit table trigger as shown in Figure 6:

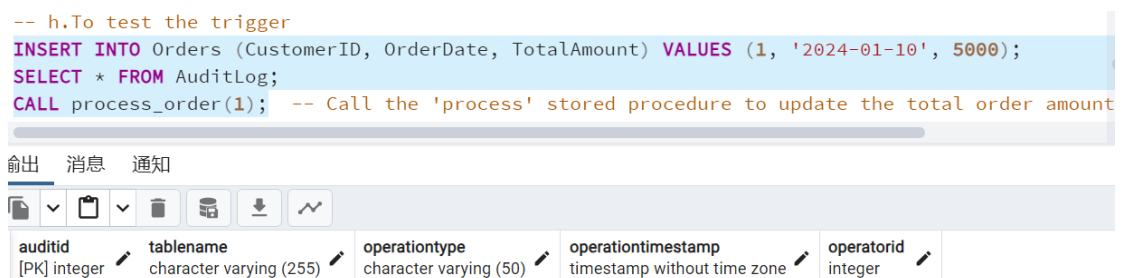


Fig. 6 Test the audit table trigger

3.3 Functions

Functions in the database are programs stored within the database that can perform specific tasks. Their importance includes:

1. **Data Processing and Transformation:** Functions can perform various data processing tasks, such as formatting into strings, calculating dates, etc., simplifying the data processing workflow, and facilitating system management and expansion.
2. **Business Logic Encapsulation:** Complex business logic can be encapsulated by user-defined functions, allowing for reuse in multiple applications and operations, improving code maintainability and expandability.
3. **Data Validation:** Functions can check the validity of data, ensuring data adheres to specific rules and standards before being inserted or updated in the database.
4. **Data Aggregation:** There are various data aggregation functions, such as SUM and COUNT, to quickly calculate totals and averages. In this system, the total amount field in

the order uses functions like SUM to calculate the total amount, ensuring database completeness.

5. Security: Functions can implement complex security logic, ensuring that only users with certain permissions can access and modify data in the database.

4.Database Design and Implementation Assessment

4.1 Database Structure Design Assessment

1. Entities and Entity Relationships: In this database, entities include all the requirements of the problem, but the attributes of entities can be expanded. For example, adding more detailed data to the customer entity, such as geographic location, age, and preferences, makes the system more complete and realistic. The relationships between entities, such as the many-to-one relationship between artworks and artists, the many-to-one relationship between orders and customers, and the many-to-one relationship between artworks and categories, are clearly articulated, and the connections between different entities are properly handled.

2. Normalization: The database follows normalization principles, i.e., 3NF (Third Normal Form). By separating entities and using join tables to manage one-to-many and many-to-many relationships, data redundancy is reduced, and data consistency is supported.

3. Data Types and Constraint Assessment: Appropriate data types, such as int and char, are selected for all entity attributes. Constraints like foreign keys and primary keys are helpful in maintaining the integrity and accuracy of the database.

4.2 Performance Optimization Assessment

The database uses indexing and query optimization to improve system performance, such as indexing artist names, artwork titles, and prices, which are key and frequent queries. However, a regular monitoring system should be established to record and maintain the effectiveness of indices and adjust them according to actual data and query patterns. In query optimization, the efficiency of common queries should be evaluated to avoid full-table scans and further optimize query statements based on actual usage and query frequency.

Performance can also be enhanced using caching for repeated and infrequent data change queries to reduce database load. Continuous monitoring of database performance metrics and maintaining database information to adjust optimization strategies based on actual usage are also beneficial.

4.3 Transaction Processing Assessment

Three transactions were tested in this database, each strictly adhering to the ACID properties to ensure database integrity and accuracy. In transaction isolation, more advanced isolation methods should be selected. Transaction isolation needs to balance

data consistency and system performance. Common levels include Read Uncommitted, where a transaction can read data from another uncommitted transaction, potentially leading to dirty reads; Read Committed, which ensures a transaction only reads committed data, avoiding dirty reads; Repeatable Read, ensuring consistent results for the same record read multiple times within the same transaction; and Serializable, completely avoiding dirty reads, non-repeatable reads, and phantom reads.

4.4 Data Integrity and Security Assessment

For integrity constraints, the database ensures that the total amount of orders cannot be negative, the price of artworks cannot be negative, and feedback ratings are between 1 to 5. These constraints ensure the system's integrity. However, the database's security measures are not yet comprehensive. Future enhancements could include user authentication, authorization, and data encryption to improve overall database security.

The database design is well-structured and normalized, with key transaction processing and performance optimizations in place. However, continuous monitoring, maintenance, and optimization are necessary to ensure long-term stability. Additionally, security within the database should be considered according to the level of the business, with attention to data security and potential expansions

5. Conclusion

This study thoroughly analyzed the requirements of the Swansea Art Store and created a comprehensive database. The database is designed to support the daily operations of an art store, encompassing multiple aspects such as artist management, art sales, customer relationship handling, and order processing. The design of the database reflects a deep understanding of business needs and exhibits an excellent balance in maintaining data consistency, ensuring operational efficiency, and supporting business scalability.

In terms of Entity-Relationship Diagram (ERD) design, this database precisely depicts the key entities and their relationships involved in business operations. Each entity has been assigned appropriate attributes, such as the artist's name and biography, the name, description, and pricing of the artworks, basic information of customers, and detailed records of orders. The design of these attributes not only exhaustively defines the characteristics of the business domain but also provides strong support for data queries and analysis. The database has been normalized to further reduce data redundancy and maintenance complexity while minimizing the risk of update anomalies and data inconsistencies. This design provides a solid foundation for future expansion of the database, making it more flexible and convenient to add new entities or attributes.

In terms of enhancing database performance, a reasonable indexing strategy has been adopted, such as indexing the titles and prices of artworks, thereby speeding up data

retrieval. Additionally, the needs of transaction processing have been considered to ensure the atomicity, consistency, isolation, and durability of database operations, especially important when handling complex order transactions.

However, as business needs evolve and technology advances, the database system may require timely adjustments and optimizations. For instance, with the growth of data volume, more complex query optimization strategies or more efficient data storage and processing mechanisms may be needed. Security and data protection are also continuous focal points, especially when dealing with customer personal information and commercially sensitive data. Moreover, the database currently lacks consideration for system security, which can be further addressed in future research.

References

- [1] C. Batini, M. Lenzerini, and S. B. Navathe. 1986. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.* 18, 4 (Dec. 1986), 323–364. <https://doi.org/10.1145/27633.27634>
- [2] Yohan Kim, Julia Ponomarenko, Zhanyang Zhu, Dorjee Tamang, Peng Wang, Jason Greenbaum, Claus Lundegaard, Alessandro Sette, Ole Lund, Philip E. Bourne, Morten Nielsen, Bjoern Peters, Immune epitope database analysis resource, *Nucleic Acids Research*, Volume 40, Issue W1, 1 July 2012, Pages W525–W530, <https://doi.org/10.1093/nar/gks438>.
- [3] Yang, D., Yu, J., He, Z. & Du, X., 2023. Applying self-powered sensor and support vector machine in load energy consumption modeling and prediction of relational database.
- [4] E. Bertino and R. Sandhu, "Database security - concepts, approaches, and challenges," in *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 1, pp. 2-19, Jan.-March 2005, doi: 10.1109/TDSC.2005.9.
- [5] Elisa Bertino, Sushil Jajodia, Pierangela Samarati, Database security: Research and practice, *Information Systems*, Volume 20, Issue 7, 1995, Pages 537-556, ISSN 0306-4379, [https://doi.org/10.1016/0306-4379\(95\)00029-4](https://doi.org/10.1016/0306-4379(95)00029-4).
- [6] Dorothy E. Denning and Peter J. Denning. 1979. Data Security. *ACM Comput. Surv.* 11, 3 (Sept. 1979), 227 – 249. <https://doi.org/10.1145/356778.356782>
- [7] D. E. Denning et al., "Views for Multilevel Database Security," in *IEEE Transactions on Software Engineering*, vol. SE-13, no. 2, pp. 129-140, Feb. 1987, doi: 10.1109/TSE.1987.232889.
- [8] Joel Coffman and Alfred C. Weaver. 2010. A framework for evaluating database keyword search strategies. In *Proceedings of the 19th ACM international conference on Information and knowledge management (CIKM '10)*. Association for Computing Machinery, New York, NY, USA, 729 – 738. <https://doi.org/10.1145/1871437.1871531>
- [9] Werner Kießling, Gerhard Köstler, Chapter 91 - Preference SQL — Design, Implementation, Experiences, Editor(s): Philip A. Bernstein, Yannis E. Ioannidis, Raghu Ramakrishnan, Dimitris Papadias, *VLDB '02: Proceedings of the 28th International Conference on Very Large Databases*, Morgan Kaufmann, 2002, Pages 990-1001, ISBN 9781558608696, <https://doi.org/10.1016/B978-155860869-6/50098-6>.
- [10] Chu S, Murphy B, Roesch J, Cheung A and Suciu D. (2018). Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *Proceedings of the VLDB Endowment*. 11:11. (1482-1495). Online publication date: 1-Jul-2018. <https://doi.org/10.14778/3236187.3236200>.
- [11] Chu S, Weitz K, Cheung A and Suciu D. (2017). HoTTSQL: proving query rewrites with univalent SQL semantics. *ACM SIGPLAN Notices*. 52:6. (510-524). Online publication date: 14-Sep-2017. <https://doi.org/10.1145/3140587.3062348>
- [12] Chu S, Weitz K, Cheung A and Suciu D. HoTTSQL: proving query rewrites with univalent SQL semantics. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. (510-524). <https://doi.org/10.1145/3062341.3062348>

Appendix 1: SQL Code.

```
-- Create Table
CREATE TABLE Artists (
    ArtistID SERIAL PRIMARY KEY,
    Name VARCHAR(255),
    Biography TEXT
);

CREATE TABLE Artworks (
    ArtworkID SERIAL PRIMARY KEY,
    Title VARCHAR(255),
    Description TEXT,
    Price DECIMAL(10, 2),
    ArtistID BIGINT UNSIGNED,
    FOREIGN KEY (ArtistID) REFERENCES Artists(ArtistID)
);

CREATE TABLE Categories (
    CategoryID SERIAL PRIMARY KEY,
    Name VARCHAR(255)
);

-- Create Artwork Category Association Table
CREATE TABLE ArtworkCategories (
    ArtworkID BIGINT UNSIGNED,
    CategoryID BIGINT UNSIGNED,
    PRIMARY KEY (ArtworkID, CategoryID),
    FOREIGN KEY (ArtworkID) REFERENCES Artworks(ArtworkID),
    FOREIGN KEY (CategoryID) REFERENCES Categories(CategoryID)
);

CREATE TABLE Customers (
    CustomerID SERIAL PRIMARY KEY,
    Name VARCHAR(255),
    Email VARCHAR(255)
);

CREATE TABLE Orders (
    OrderID SERIAL PRIMARY KEY,
    CustomerID BIGINT UNSIGNED,
    OrderDate DATE,
    TotalAmount DECIMAL(10, 2),
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

CREATE TABLE OrderDetails (
    OrderID BIGINT UNSIGNED,
    ArtworkID BIGINT UNSIGNED,
    Quantity INT,
    PRIMARY KEY (OrderID, ArtworkID),
    FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),
    FOREIGN KEY (ArtworkID) REFERENCES Artworks(ArtworkID)
);

CREATE TABLE Feedback (
    FeedbackID SERIAL PRIMARY KEY,
    CustomerID BIGINT UNSIGNED,
    ArtworkID BIGINT UNSIGNED,
    Rating INT,
    COMMENT TEXT,
    FOREIGN KEY (CustomerID) REFERENCES Customers ( CustomerID ),
    FOREIGN KEY ( ArtworkID ) REFERENCES Artworks ( ArtworkID )
);
```

```

INSERT INTO Artists (Name, Biography) VALUES
('Mack', 'Famous European Painter'),
('Bob', 'Famous American painters, specializing in oil painting'),
('James', 'Famous European Realist Painter'),
('NaZhang', 'Famous Chinese Ink Painter'),
('Susan', 'Experienced abstract painter'),
('John', 'Young Photographer'),
('Jack', 'Famous photographer, specializing in the use of light'),
('Rose', 'Famous European photographers'),
('Smith', 'Young Sculptor'),
('Fack', 'Famous European sculptors'),
('MingXiao', 'Famous Chinese Sculptors'),
('Yijun', 'Famous Architects in China'),
('David', 'Famous European architects');

-- Insert Information into the Artwork Table
INSERT INTO Artworks (Title, Description, Price, ArtistID) VALUES
('The Call of Spring', 'A painting depicting a spring scene', 3000, 1),
('starry sky', 'A painting that expresses the emotion of longing', 5000, 2),
('many stars', 'A painting depicting the night sky', 2000, 3),
('Moon and Night', 'A painting depicting the night', 9000, 3),
('Flames', 'A painting depicting the mood', 12000, 5),
('Mountains and running water', 'A landscape painting depicting a mountainous landscape', 22000, 4),
('painting of eight steeds', 'A painting of eight horses.', 9000, 4),
('Van Gogh', 'A figure painting depicting Van Gogh', 32000, 1),
('meteors', 'A picture of a shooting star', 1000, 6),
('father', 'A picture of an old man.', 3000, 6),
('youthfulness', 'A photo of a young man', 2000, 7),
('municipalities', 'A city photo', 3000, 8),
('hobgoblin', 'A ghostly sculpture.', 1000, 9),
('Zeus', 'A sculpture of Zeus', 12000, 10),
('Athena', 'An Athena sculpture', 11000, 10),
('Iron Man', 'An Iron Man sculpture.', 6000, 11),
('horse', 'Sculpture of a horse', 17000, 11),
('bridee', 'A bridee desien'. 32000. 12).

INSERT INTO Categories (Name) VALUES
('pastel'),
('abstract painting'),
('realistic painting'),
('ink painting'),
('shoot '),
('sculpture'),
('devise');

-- Insert Information into the ArtworkCategories Table
INSERT INTO ArtworkCategories (ArtworkID, CategoryID) VALUES
(1, 1),
(2, 2),
(3, 3),
(4, 3),
(5, 2),
(6, 4),
(7, 4),
(8, 5),
(9, 5),
(10, 5),
(11, 5),
(12, 5),
(13, 6),
(14, 6),
(15, 6),
(16, 6),
(17, 6),
(18, 7),
(19, 7);

```

```

--Insert Information into the Customer Table
INSERT INTO Customers (Name, Email) VALUES
('John Doe', 'john.doe@example.com'),
('Emma Smith', 'emma.smith@example.com'),
('Michael Johnson', 'michael.johnson@example.com'),
('Sarah Brown', 'sarah.brown@example.com'),
('Olivia Davis', 'olivia.davis@example.com'),
('William Martinez', 'william.martinez@example.com'),
('Sophia Garcia', 'sophia.garcia@example.com'),
('James Wilson', 'james.wilson@example.com'),
('Isabella Anderson', 'isabella.anderson@example.com'),
('Lucas Miller', 'lucas.miller@example.com'),
('Mia Lee', 'mia.lee@example.com'),
('Benjamin Taylor', 'benjamin.taylor@example.com'),
('Charlotte Rodriguez', 'charlotte.rodriguez@example.com'),
('Ethan Thomas', 'ethan.thomas@example.com'),
('Amelia White', 'amelia.white@example.com');

--Insert Information into the Order Table
INSERT INTO Orders (CustomerID, OrderDate, TotalAmount) VALUES
(1, '2024-01-01', 3000),
(2, '2024-01-02', 5000),
(3, '2023-11-23', 2000),
(4, '2023-11-11', 9000),
(5, '2022-11-12', 22000),
(7, '2023-12-01', 24000),
(8, '2022-12-20', 3000),
(9, '2023-08-02', 6000),
(11, '2023-09-13', 9000),
(12, '2023-03-13', 3000),
(13, '2022-11-24', 3000);

--Insert Information into the Order Details Table
INSERT INTO OrderDetails (OrderID, ArtworkID, Quantity) VALUES
(1, 1, 1),
(2, 2, 1),
(3, 3, 1),
(4, 4, 1),
(5, 6, 1),
(6, 5, 2),
(7, 9, 3),
(8, 11, 3),
(9, 7, 1),
(10, 10, 1),
(11, 12, 1);

--Insert Information into the Feedback Table
INSERT INTO Feedback (CustomerID, ArtworkID, Rating, Comment) VALUES
(1, 1, 5, 'This is such a beautiful picture! '),
(2, 2, 4, 'I like it.'),
(3, 3, 3, 'I think its average.'),
(4, 4, 3, 'It met my expectations.'),
(5, 5, 5, 'Great. A real work of art.'),
(6, 6, 3, 'I think its very common.'),
(7, 7, 2, 'I dont think its worth that much.'),
(8, 8, 4, 'Not bad.'),
(9, 9, 1, 'This sucks. Get your money back.'),
(10, 10, 4, 'Very nice work.'),
(11, 11, 5, 'I really, really like this one.');
```

```

-- Check Constraint
-- Constraints for Price and Quantity greater than 0
-- Constraint for Rating between 1 and 5
ALTER TABLE Artworks ADD CONSTRAINT chk_price CHECK (Price > 0);
ALTER TABLE Feedback ADD CONSTRAINT chk_rating CHECK (Rating BETWEEN 1 AND 5);
ALTER TABLE OrderDetails ADD CONSTRAINT chk_quantity CHECK (Quantity > 0);
```

```

-- Add an average rating field to the Artwork table.
ALTER TABLE Artworks
ADD COLUMN AverageRating DECIMAL(3,2);
-- Add an average rating trigger
CREATE OR REPLACE FUNCTION update_artwork_rating()
RETURNS TRIGGER AS $$
BEGIN
    -- Calculate the new average rating
    UPDATE Artworks
    SET AverageRating = (
        SELECT AVG(Rating)
        FROM Feedback
        WHERE ArtworkID = NEW.ArtworkID
    )
    WHERE ArtworkID = NEW.ArtworkID;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_update_artwork_rating
AFTER INSERT ON Feedback
FOR EACH ROW EXECUTE FUNCTION update_artwork_rating();

-- Update the AverageRating field in the Artworks table.
UPDATE Artworks
SET AverageRating = subquery.avg_rating
FROM (
    SELECT ArtworkID, AVG(Rating) AS avg_rating
    FROM Feedback
    GROUP BY ArtworkID
) AS subquery
WHERE Artworks.ArtworkID = subquery.ArtworkID;

-- index
--Create indexes on Artist Name, Artwork Name, Price, Order Date, and Feedback Rating
CREATE INDEX idx_artist_name ON Artists(Name);

CREATE INDEX idx_title ON Artworks(Title);

CREATE INDEX idx_artwork_price ON Artworks(Price);

CREATE INDEX idx_feedback_rating ON Feedback(Rating);

CREATE INDEX idx_order_date ON Orders(OrderDate);

-- Create Audit Table
CREATE TABLE AuditLog (
    AuditID SERIAL PRIMARY KEY,
    TableName VARCHAR(255),
    OperationType VARCHAR(50),
    OperationTimestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    OperatorID INT
);

-- Create Audit Trigger Function
CREATE OR REPLACE FUNCTION audit_trigger_function() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO AuditLog (TableName, OperationType, OperatorID)
    VALUES (TG_TABLE_NAME, TG_OP, NEW.CustomerID);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- Apply Audit Trigger to the Order Table
CREATE TRIGGER order_audit_trigger
AFTER INSERT OR UPDATE OR DELETE ON Orders
FOR EACH ROW EXECUTE FUNCTION audit_trigger_function();

```

```

CREATE OR REPLACE FUNCTION calculate_order_total(order_id INT) RETURNS DECIMAL AS $$
DECLARE
    total DECIMAL;
BEGIN
    SELECT SUM(Price * Quantity) INTO total
    FROM OrderDetails
    JOIN Artworks ON OrderDetails.ArtworkID = Artworks.ArtworkID
    WHERE OrderDetails.OrderID = order_id;
    IF total IS NULL THEN
        total := 0;
    END IF;
    RETURN total;
END;
$$ LANGUAGE plpgsql;

-- Create Stored Procedure for Handling Orders
CREATE OR REPLACE PROCEDURE process_order(order_id INT) AS $$
BEGIN
    UPDATE Orders SET TotalAmount = calculate_order_total(order_id)
    WHERE OrderID = order_id;
END;
$$ LANGUAGE plpgsql;

-- f.Try to insert an artwork record with a negative price to test the chk_price constraint.
INSERT INTO Artworks (Title, Description, Price, ArtistID) VALUES ('Test Artwork', 'Test Description', -100, 1);
-- f.To attempt inserting a feedback record with a rating beyond the range to test the chk_rating constraint.
INSERT INTO Feedback (CustomerID, ArtworkID, Rating, Comment) VALUES (1, 1, 6, 'Test Comment');
-- g.To begin a transaction for testing rollback
START TRANSACTION;
INSERT INTO Orders (CustomerID, OrderDate, TotalAmount) VALUES (1, CURRENT_DATE, 1000);
ROLLBACK;
-- g.To test transaction commit, insert an order, and then commit
START TRANSACTION;
INSERT INTO Orders (CustomerID, OrderDate, TotalAmount) VALUES (1, CURRENT_DATE, 1000);
COMMIT;

-- h.To test the trigger
INSERT INTO Orders (CustomerID, OrderDate, TotalAmount) VALUES (1, '2024-01-10', 5000);
SELECT * FROM AuditLog;
CALL process_order(1); -- Call the 'process' stored procedure to update the total order amount

-- Example transaction, customer purchasing artwork transaction.
-- Begin transaction
BEGIN;
-- Insert the order using the WITH statement and capture the newly inserted order's ID
WITH inserted_order AS (
    INSERT INTO Orders (CustomerID, OrderDate, TotalAmount)
    VALUES (1, CURRENT_DATE, 0)
    RETURNING OrderID
)
-- Insert order details while referencing the OrderID from the inserted_order CTE.
INSERT INTO OrderDetails (OrderID, ArtworkID, Quantity)
SELECT OrderID, 1, 1 FROM inserted_order; -- ArtworkID and Quantity are 1 and 1.
-- Update the total order amount.
UPDATE Orders
SET TotalAmount = calculate_order_total((SELECT MAX(OrderID) FROM Orders))
WHERE OrderID = (SELECT MAX(OrderID) FROM Orders);
-- Commit the transaction.
COMMIT;

-- Test if the average rating trigger works correctly.
-- Add a customer feedback transaction.
BEGIN;
-- Insert artwork rating.
INSERT INTO Feedback (CustomerID, ArtworkID, Rating, Comment)
VALUES (1, 12, 5, 'transaction test Excellent piece!');
COMMIT;

-- Begin a transaction for updating customer information
START TRANSACTION;
-- Update customer information
UPDATE Customers
SET Name = 'new customer', Email = 'test email'
WHERE CustomerID = 1;
-- Commit the transaction.
COMMIT;

```