CanvasCart: Buy-and-Sell platform for Artworks

Vindhya Jain, Avanti Mittal, Khushi Bhardwaj

Introduction

CanvasCart is a buy-and-sell platform for artworks. This platforms allows artists to list their artworks for sale and buyers to browse, purchase, and manage their orders. Through the course of this assignment, we have designed and developed the some backend and frontend components of this platform. Link to Colab Notebook.

Objectives & Plan

- ER Diagram and Database Design: Create an ER diagram to visualize the platform's structure and implement the database with Python, ensuring tables meet at least 2NF.
- Data Handling and Indexing: Insert dummy records, apply clustering and secondary indexing on the Artwork Listings table, and compare their performance in terms of storage and execution time.
- Hash Function Design: Design a custom hash function for the Artwork Listings table to optimize data storage and retrieval.
- Study on MySQL Indexing: Analyze and document MySQL's hashing and indexing schemes, and apply learned concepts to optimize the platform's performance.
- SQL Queries: Write SQL queries for various operations, including adding new artists, generating reports on artwork listings, and removing specific purchases.

ER Diagram and Database Design

The ER diagram represents the database design for the CanvasCart platform with the following tables:

User Table:

- Attributes: UserID (Primary Key), First Name, Last Name, Email.
- Purpose: Stores basic information about users. Users can be either artists or buyers.

Authentication System Table:

- Attributes: LoginID (Primary Key, Foreign Key (references UserID from User table)), Password.
- Purpose: Manages login credentials for user authentication.

Artist Table:

- Attributes: ArtistID (Primary Key, Foreign Key (references UserID from User table)), PortfolioLink, Biography.
- Purpose: Stores additional details specific to artists, linked to the User table via UserID.

Buyer Table:

- Attributes: BuyerID (Primary Key, Foreign Key (references UserID from User table)), Membership, Shipping Address.
- Purpose: Stores buyer-specific information, linked to the User table via UserID.

Artwork Table:

- Attributes: ArtworkID (Primary Key), ArtistID (Foreign Key (references ArtistID from Artist table)), Style, Availability, Price, ImageUrl, Title, Medium, Description, DateAdded.
- Purpose: Manages details of artworks listed for sale, linked to the Artist table via ArtistID.

Item Table:

- Attributes: ArtworkID, OrderID (Primary Key, Foreign Key (references ArtworkID from Artwork table, references OrderID from Order Table)), Amount, Quantity.
- Purpose: Represents items added to the shopping cart by the buyer, linked to the Artwork table via ArtworkID.

Shopping Cart Table:

- Attributes: CartID (Primary Key, Foreign Key (references BuyerID from Buyer table)), LastUpdated (Date, Time).
- Purpose: Stores the current state of the buyer's cart, containing items selected for purchase.

Orders Table:

- Attributes: OrderID (Primary Key), BuyerID (Foreign Key (references BuyerID from Buyer table), Order Time, Order Date, Total Amount, Payment Status, Shipping Status.
- Purpose: Tracks orders placed by buyers, including order details and status, linked to the Buyer table via BuyerID.

Payment Table:

- Attributes: PaymentID (Primary Key), OrderID (Foreign Key (references OrderID from Orders table)), Payment Amount, Payment Method.
- Purpose: Handles payment transactions for orders, linked to the Order table via OrderID.

Review Table:

- Attributes: ReviewID (Primary Key), BuyerID (Foreign Key (reference BuyerID from Buyer table)), ArtworkID (Foreign Key (reference ArtworkID from Artwork table)), ReviewText, Rating.
- Purpose: Manages reviews left by buyers for artworks, linked to both Buyer and Artwork tables.

Each table in the diagram is interconnected to ensure a cohesive structure that allows for efficient data management within the platform. Link to ER Diagram.

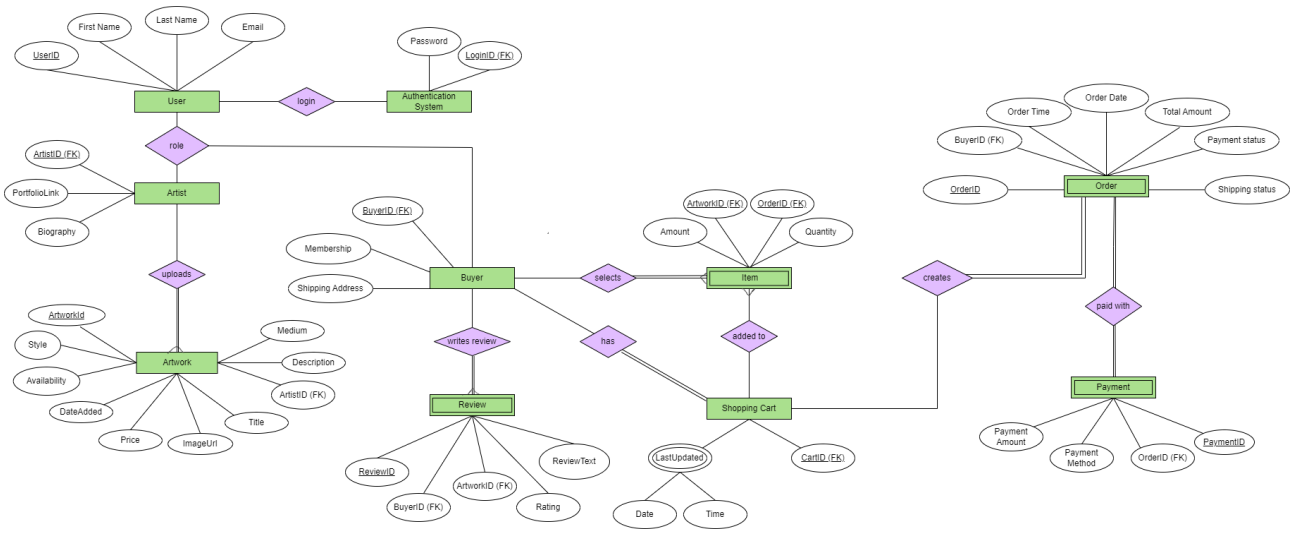

Figure 1: ER Diagram

Table Creation, Inserting Dummy Records

Following the structure of the ER diagram as shown above, a database with all these table is created using `sqlite3` module in python. Some dummy records have also been inserted using the `faker` module in python.

UserID	FirstName	LastName	Email	Role
1	Amy	Garcia	tmontoya@example.net	Artist
2	Nancy	Ramirez	nberger@example.org	Artist
3	Jeffrey	Wilson	wellsmichael@example.net	Artist
4	Melissa	Barry	ecampos@example.com	Artist
5	Olivia	Moore	smithwesley@example.net	Artist
6	Judith	Fuentes	fostertonya@example.org	Artist
7	Sarah	Aguilar	ramirezcynthia@example.net	Artist
8	Eddie	Mitchell	tammy24@example.com	Artist
9	Charles	Neal	ipearson@example.net	Artist
10	Casey	Richards	matthew29@example.com	Artist
11	Steve	Gonzalez	ujordan@example.com	Buyer
12	Nicole	Hicks	andersonamanda@example.net	Buyer
13	David	Hawkins	andersontravis@example.org	Buyer
14	Rebecca	Obrien	kaguilar@example.net	Buyer
15	Christopher	Brown	watersjennifer@example.com	Buyer

Figure 2: User Table

ArtworkID	ArtistID	Title	Description	ImageURL	Medium	Style	Price	Availability
OIL_01	1	Hour provide.	Discuss answer history...	https://www.morrow-mcintosh.org/	Oil	Impressionism	7315.93	13
OIL_02	1	Should establish research according.	Simple company language...	http://williams.com/	Oil	Abstract	7482.24	12
OIL_03	1	Letter bad respond.	Seem watch especially...	https://www.ruiz.org/	Oil	Modern	7120.08	18
WAT_01	1	Food growth.	News arrive close...	http://www.sanchez-daniels.com/	Watercolor	Abstract	4723.26	13
ACR_01	2	Hour mind which when.	Data PM Mr...	https://www.alexander.com/	Acrylic	Realism	6653.56	13
DIG_01	2	Relationship try research.	Benefit available building...	https://www.garcia-stephenson.com/	Digital	Realism	6732.13	13
DIG_02	3	Dark several can.	Knowledge what strong...	http://hinton-clark.com/	Digital	Realism	6461.86	11
WAT_02	3	My.	Near image loss...	http://nash.com/	Watercolor	Impressionism	744.99	15
OIL_04	3	Order describe.	Financial certainly what...	http://townsend.org/	Oil	Impressionism	9525.57	7
OIL_05	3	Special both friend.	Certainly organization dream...	http://www.mcintosh.com/	Oil	Impressionism	7190.12	16
WAT_03	4	Fund watch least.	Far effort street...	http://www.gross.com/	Watercolor	Impressionism	9509.74	11
DIG_03	4	Name cup medical.	Direction north grow...	https://silva.biz/	Digital	Modern	3639.82	14
OIL_06	4	News baby.	Floor its try...	http://lucas.org/	Oil	Impressionism	3145.70	18
ACR_02	4	Smile color.	When particularly probably...	http://www.hatfield.org/	Acrylic	Realism	8562.07	3
OIL_07	5	Home make community.	Short financial common...	https://schwartz.biz/	Oil	Impressionism	1672.70	9
DIG_04	6	Theory.	Recently arm guess...	https://www.wheeler.info/	Digital	Abstract	8770.91	20
OIL_08	7	Tell wide.	Avoid behavior office...	https://www.nelson.biz/	Oil	Impressionism	6899.08	14

Figure 3: Artwork Table

Tables in 2NF and Higher

1. All records are atomic, meaning that each attribute contains indivisible values, and there are no repeating groups or arrays within the tables. (1NF)

2. Each table contains only attributes that are directly related to the primary key, ensuring there are no partial dependencies. For example, in the Artist table, attributes like PortfolioLink and Biography are directly related to ArtistID, which is the primary key. (2NF)

3. No transitive dependencies exist within the tables. This means that non-key attributes do not depend on other non-key attributes. For instance, in the Artwork table, attributes such as Style, Availability, and Price are dependent solely on ArtworkID, with no other non-key attributes affecting them. (3NF)

4. Each non-key attribute is fully functionally dependent on the primary key, meaning every attribute in a table is dependent only on the primary key and not on any other non-key attributes. (BCNF)

Comparing 2NF with 3NF/ BCNF: Trade-Offs and Benefits

When deciding between normalizing tables up to Third Normal Form (3NF) or Boyce-Codd Normal Form (BCNF) versus leaving them at Second Normal Form (2NF), there are several trade-offs to consider:

Data Redundancy and Anomalies: 3NF/BCNF addresses a broader range of anomalies by ensuring that non-key attributes are only dependent on the primary key, and all functional dependencies have a superkey on the left side. Provides a more rigorous normalization that can lead to more efficient database operations in certain cases.
Query Performance: 3NF/BCNF will result in more complex schema design with increased number of tables and joins, which might negatively impact query performance.
Maintenance: 3NF/BCNF provides better data integrity and consistency, with fewer redundancy issues, leading to easier data maintenance and updates.

Our database schema involves some scenarios where retrieving all reviews for a particular artwork requires joining tables. Although we considered denormalizing the schema to reduce the need for joins, this approach would result in tables that are not in First Normal Form (1NF). For example, while retrieving reviews for an artwork, due to the nature of multiple reviews associated with each artwork, the records are not atomic. On normalising to 1NF by adding columns (review1, review2, ...), this would lead to the inclusion of many null values. After evaluating the trade-offs, we have decided that maintaining the current schema, with its higher normal form, is the most effective solution. This approach ensures data integrity and avoids the complications of null values and redundant data, despite the need for joins.

Hashing and Indexing schemes underlying MySQL

Hashing: In MySQL, hashing is primarily used with in-memory storage engines like the MEMORY engine. Hashing involves creating a hash table where keys are mapped to specific locations through a hash function. This function transforms a key into an index that directly points to a row in the table, allowing for extremely fast lookups. Hashing excels in scenarios requiring quick, exact-match queries due to its constant time complexity for retrieval. However, it has limitations, including inefficient support for range queries and potential hash collisions, which can affect performance. These trade-offs make hashing most suitable for applications with predictable query patterns and where range queries are not required.

Indexing: Indexing in MySQL improves query performance by allowing the database to quickly locate rows without scanning the entire table. MySQL supports several indexing methods:

- B-Tree Indexes: Used by default in most storage engines (e.g., InnoDB, MyISAM). B-Tree indexes are efficient for equality and range queries, as they maintain a balanced tree structure that allows logarithmic time complexity for search operations.
- Full-Text Indexes: Useful for text-based searches in columns with large text data. Full-text indexes are employed to perform natural language searches, allowing for complex queries involving word relevance and proximity.
- Spatial Indexes: Applied to spatial data types (e.g., geometries). Spatial indexes help optimize spatial queries and are based on R-Tree structures, which efficiently handle multi-dimensional data.
- Hash Indexes: Available in the MEMORY storage engine. Hash indexes use a hash table to map keys directly to rows, which provides constant time complexity for lookups but does not support range queries efficiently.

MySQL also allows the creation of **composite indexes**, which index multiple columns in a single index. Composite indexes are beneficial for queries involving multiple columns, as they can significantly reduce the number of rows scanned and improve query performance.

In summary, MySQL employs various hashing and indexing techniques to enhance query performance and data retrieval efficiency. Understanding and optimizing these schemes are crucial for maintaining a high-performance database system.

Hash function for 'Artwork' table

The hash function (h) implemented for the Artwork table uses the ArtworkID as input to generate a hash value, which is then used to distribute entries across a specified number of buckets. The ArtworkID has two parts - prefix and suffix. Prefix is a character string such as 'OIL' or 'WAT' and it is derived from the medium of the artwork. The suffix is a numerical serial number. For example: *OIL_01*. The hash function for ArtworkID calculates a hash value by **summing weighted ASCII values** of the identifier's characters, **added a constant based on the presence of 'A', 'I', 'E', and 'S'**. The result is then taken **modulo the number of buckets**. Using such a method, we have designed a hash function that takes into common alphabets in our roll numbers as well (R).

$$h(\text{ArtworkID}) = \left(\sum_{\text{for } p_i \text{ in prefix}} p_i \cdot (i+1) \quad + \sum_{\text{for } s_i \text{in suffix}} s_i \cdot (i+1) \quad + \quad R \right) \mod N$$

where:

$$R = \text{ASCII(A)} + \text{ASCII(I)} + \text{ASCII(E)} + \text{ASCII(S)}$$

Example for ArtworkID **OIL_01**
Prefix = OIL, Suffix = 01
h(OIL_01) = ((79*1 + 73*2 + 76*3) + (48*1 + 49*2)) mod N = 599 mod N
(where N is number of buckets)

Benefits and strengths of h

- Provides deterministic output and consistent hash values for the same ArtworkID, which is crucial for reliable storage and retrieval.
- Uses straightforward operations—ASCII value manipulation and modular arithmetic—which are computationally efficient.
- The weighted ASCII sum of the prefix and suffix introduces a level of differentiation between identifiers that start with different prefixes. For example between WAT and TAW.

Potential drawbacks

- The function is highly sensitive to small changes in the ArtworkID. A minor alteration in either the prefix or suffix could significantly change the hash value, which could be both an advantage and a disadvantage. While this sensitivity reduces the risk of collision for small changes, it also makes it difficult to predict how similar IDs are hashed.
- Relying on ASCII values for numeric part addition might not always result in well-distributed hash values, especially if numeric parts are short or consist of repeated digits.

Reasons for overlooking potential drawbacks

- The function's simplicity makes it easy to implement and understand. In scenarios where hash table performance is acceptable with this approach, the drawbacks may be deemed minor.
- For our dataset with well-defined ID formats, the function's assumptions (e.g., fixed prefix length) may not significantly impact performance. The benefits of straightforward implementation and customization may outweigh the potential drawbacks.

We are using linear hashing and have started with 5 buckets (with 5 max depth).

Clustering Indexing for ArtistID data in 'Artwork' table

If file records are physically ordered on a non-key field that field is called the **clustering field** and the data file is called a **clustered file**. Clustering fields aid in the retrieval of data with the same value for the clustering field, unlike the primary index.

The Artwork table has the following attributes:
Artwork <ArtworkID, ArtistID (FK), Title, Description, ImageURL, Medium, Style, Price, Availability>

Filtering artworks on the basis of the **artist** can be a very common query. Repeatedly parsing the entire table for such a query would eat up a lot of computing resources and time. It will also slow down the experience of using your application. Clustering indexing can be used in such a case because **a) the data in ArtistID is ordered**, and **b) it is not unique** (as there can be many artworks by a single artist), as can be seen in Figure 3 .

Secondary Indexing for Medium & Price data in 'Artwork' table

In addition to searching by 'artist', users might also want to view artworks based on criteria like 'medium' or 'price'. While there are many potential query combinations for which indexes could be created, having too many indexes can lead to increased overhead during updates and deletions. After careful consideration, we've decided to create indexes specifically for the 'medium' and 'price' fields, as these queries are highly likely to be used frequently.

Let us make an index for 'Medium' and check if it is being used when querying:

```
CREATE INDEX idx_medium
ON Artwork (Medium);

EXPLAIN QUERY PLAN
SELECT * FROM Artwork
WHERE Medium = "Oil";
```

(3, 0, 0, 'SEARCH Artwork USING INDEX idx_medium (Medium=?)')

Explanation of the Output:

- Operation Code (3): 3 indicates a SEARCH operation, which means it's using an index to search for data.
- Index ID (0): This is an internal identifier for the index being used. In this case, it's 0, which corresponds to the index name in the query.
- Table ID (0): This indicates the internal identifier for the table involved in the query. Again, 0 corresponds to the Artwork table.
- Detail (SEARCH Artwork USING INDEX idx_artistid (Medium=?)): This is the actual explanation of the operation. It tells that the index idx_medium is being used to perform a SEARCH operation on the Artwork table.

Storage and Performance Analysis of Indexing Schemes

Clustering Indexing (ArtistID)

Storage:

- Organization: Clustering indexing arranges the physical storage of records in the table according to the clustering field, which in this case is ArtistID. This means that all records with the same ArtistID are stored contiguously on disk.
- Space Requirement: Clustering indexes do not require a separate index file for every entry because the physical ordering of the records serves as the index. Instead, they may use block pointers or similar mechanisms, resulting in a modest additional storage overhead compared to secondary indexing.

Execution Time:

- Retrieval Efficiency: When querying by ArtistID, the execution time is minimized because the records are stored contiguously. This reduces the need for scanning or jumping between different disk blocks, leading to faster access times.
- Range Queries: Clustering indexes are particularly efficient for range queries on the ArtistID field. Since the records are physically sorted, a range scan can be performed very quickly.

- Insertion/Deletion Performance: Insertions and deletions can be more complex and time-consuming because the physical order of records must be maintained. This could involve shifting records on the disk, which can slow down performance.

Secondary Indexing (Medium & Price)

Storage:

- Organization: Secondary indexes create a separate index structure that contains pointers to the actual data records in the table. For example, an index on Medium would store each unique medium and a list of pointers to the records that match that medium.
- Space Requirement: Secondary indexes require additional storage space for the index files, which store key-pointer pairs. If both Medium and Price are indexed, each index will consume storage proportional to the number of unique values and records in the table.

Execution Time:

- Retrieval Efficiency: Queries on Medium and Price will be faster with secondary indexing because the index allows for direct access to the records without scanning the entire table. However, unlike clustering indexes, these queries may require accessing multiple non-contiguous disk blocks, leading to potentially slower access times compared to clustering for large datasets.
- Range Queries: Secondary indexes also support range queries efficiently, but the records retrieved may not be stored contiguously, leading to higher I/O overhead compared to clustering indexing.
- Insertion/Deletion Performance: Secondary indexes do not impose the same level of overhead during insertions and deletions as clustering indexes, since the physical order of records on disk does not need to be maintained. However, the index files themselves will need to be updated, which can still slow down performance, especially if there are multiple secondary indexes.

Hence, we see that Clustering Indexing requires minimal additional storage, as it leverages the physical order of the table itself, but Secondary Indexing requires separate index files, increasing storage overhead.

Clustering Indexing also provides fast retrieval for queries on the clustering field (ArtistID), particularly for range queries, but can slow down insertions and deletions due to the need to maintain physical order. Secondary Indexing allows for fast queries on non-clustering fields (Medium and Price), with less impact on insertion and deletion times, but may have slower access times compared to clustering indexing for large datasets, especially when range queries involve accessing multiple non-contiguous records.

In conclusion, clustering indexing is highly efficient for queries that frequently filter by ArtistID but may increase complexity during data modifications. On the other hand, secondary indexing is flexible and efficient for queries on fields like Medium and Price, with a trade-off in storage overhead and slightly longer access times compared to clustering indexes.

Some more queries

Inclusion of information on 5 new contemporary artists

```
artists = [
    ('Banksy', 'Anonymous', 'banksy@example.com', 'Artist'),
    ('Yayoi Kusama', 'Kusama', 'yayoi@example.com', 'Artist'),
    ('Ai Weiwei', 'Weiwei', 'aiweiwei@example.com', 'Artist'),
    ('Damien Hirst', 'Hirst', 'damien@example.com', 'Artist'),
    ('Jeff Koons', 'Koons', 'jeffkoons@example.com', 'Artist')
]

cursor.executemany('''
    INSERT INTO User (FirstName, LastName, Email, Role)
    VALUES (?, ?, ?, ?)
''', artists)
```

Figure 4: Query to add 5 new contemporary artists

5 new entries added to the end of artwork table

51	Banksy	Anonymous	banksy@example.com	Artist
52	Yayoi Kusama	Kusama	yayoi@example.com	Artist
53	Ai Weiwei	Weiwei	aiweiwei@example.com	Artist
54	Damien Hirst	Hirst	damien@example.com	Artist
55	Jeff Koons	Koons	jeffkoons@example.com	Artist

Figure 5: Updated User table

Prepare a report on all artwork listings made in the month of August, 2024

```
query = '''
    SELECT * FROM Artwork
    WHERE DateAdded BETWEEN '2024-08-01' AND '2024-08-31'
'''
```

Figure 6: Query to find artworks in the month of August

Only 1 artwork listing found in the month of August, 2024.

ArtworkID	ArtistID	Title	Description	ImageURL	Medium	Style	Price	Availability	DateAdded
WAT_04	6	Live.	Prove tonight feel much. Huge season behavior ...	http://lee.com/	Watercolor	Realism	9746.22	13	2024-08-15

Figure 7: Fetched artwork listed in the month of August, 2024 (from Artwork table)

Remove all artwork purchases made after 7PM on August 15, 2024.

```
cutoff_date = '2024-08-15'
cutoff_time = '19:00:00'

cutoff_datetime = f'{cutoff_date} {cutoff_time}'

cursor.execute('''
    DELETE FROM Orders
    WHERE OrderDate = ? AND OrderTime > ?
''', (cutoff_date, cutoff_time))
```

Figure 8: Query to remove all artwork purchases made after 7PM on August 15

No records were found for artworks that were purchased after 7PM on August 15. Hence no change was there in the table.