DBMS Project Report

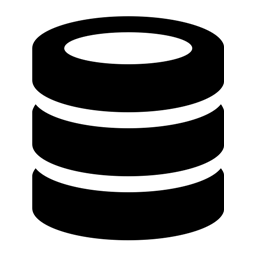
## PES University Database Management Systems

## UE18CS252

## Topic – ART GALLERY

## Submitted By

PES2201800321 Akhil Khubchandani



# **INDEX**

[Introduction 3](#_Toc41594211)

[ERD AND Schema 4](#_Toc41594212)

[FD and Normalization 6](#_Toc41594213)

[DDL SCRIPTS 7](#_Toc41594214)

[Triggers 8](#_Toc41594215)

[SQL Queries 9](#_Toc41594216)

[Conclusion 12](#_Toc41594217)

# **Introduction**

This DBMS project is about an art gallery. Complete description about the capabilities of this system is given below.

Art Gallery DBMS is meant to cater to the operations of an art gallery. the art gallery will have following entities:

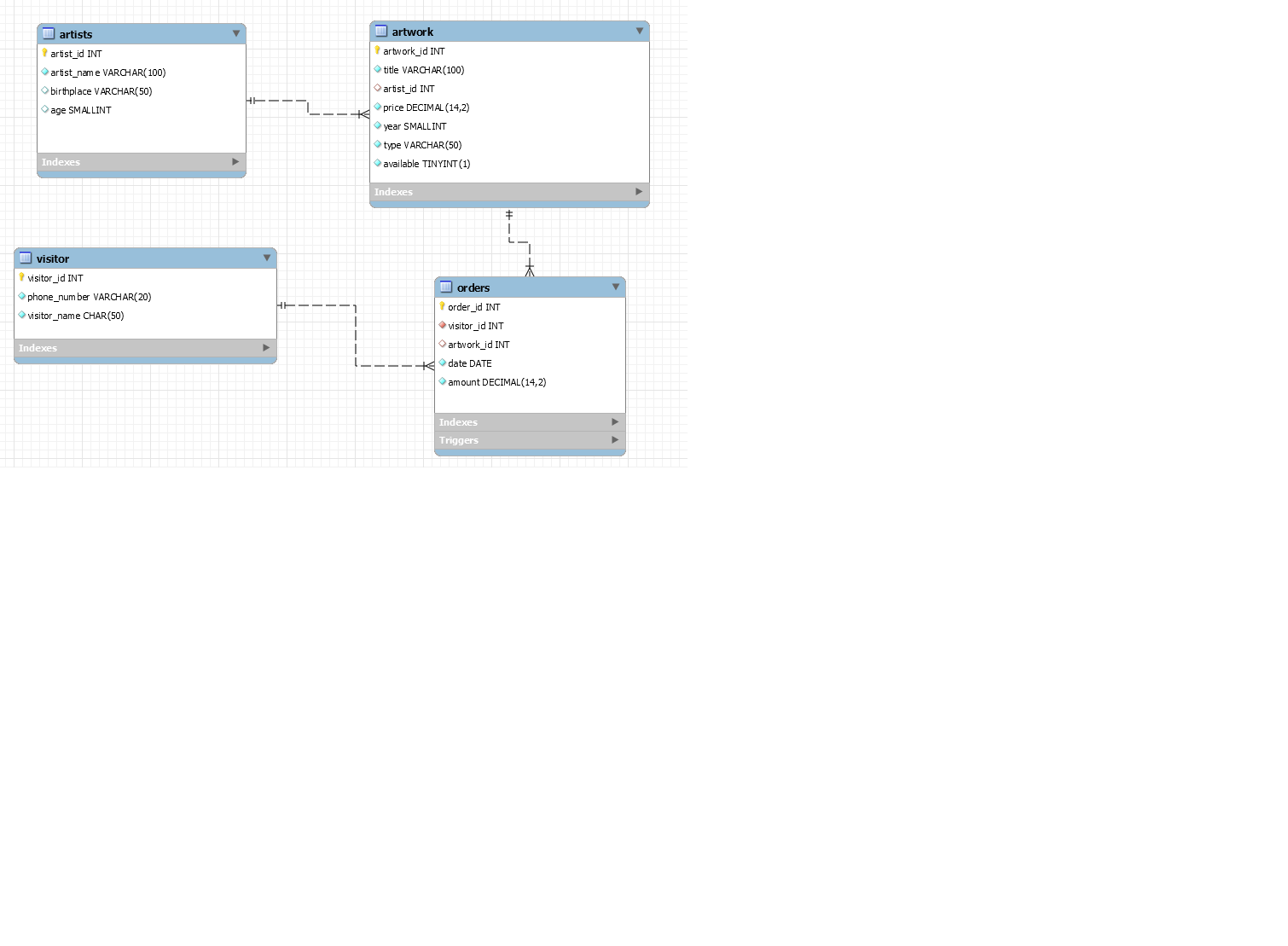
1. Artists
2. Artworks
3. Consumers
4. Orders

With the help of queries, following functionalities and transactions of the art gallery can be accomplished:

1. Adding of artists, their artwork
2. Adding of orders and customer information
3. View category wise sale amount and count of artwork sold
4. View total sales for the month and year
5. Get to know which artwork has been sold at highest price
6. View all artwork available in gallery – this can help in guiding the customer, as well understanding the stock at gallery
7. Update price of artwork in case it is entered incorrectly or artist wants to change it
8. View those artists who have not sold any artwork – this will help gallery to decide a strategy about such artists
9. Able to delete orders in case wrong order has been punched in
10. Art Gallery mandates that data of all visitors is recorded
11. Customers are visitors who have purchased artwork
12. View customers – can be used by sales for any event propagation or marketing strategy
13. View customers who buy frequently (more than 5 times)– can be used by sales for any event propagation or marketing strategy

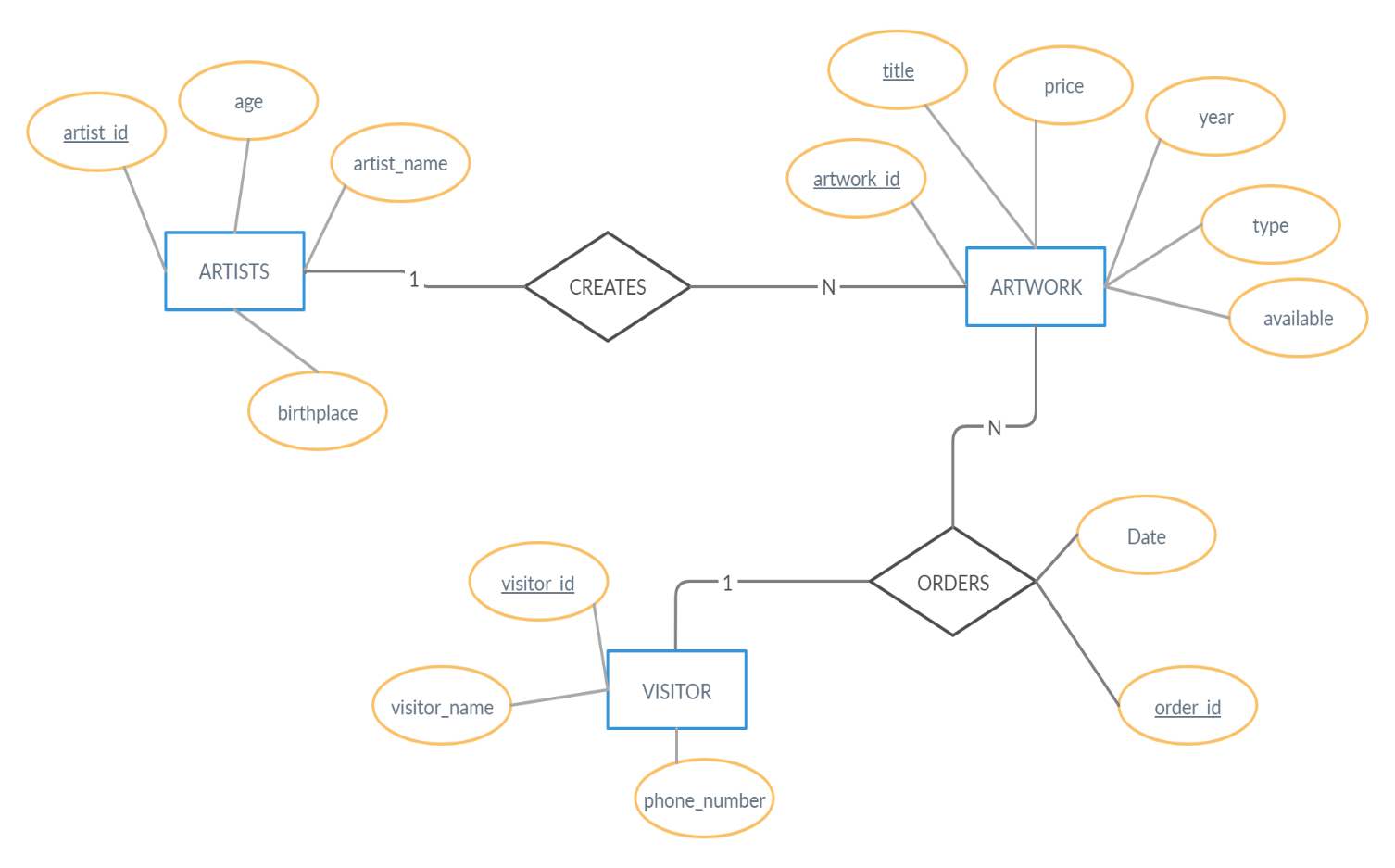
# 

# **ERD AND Schema**

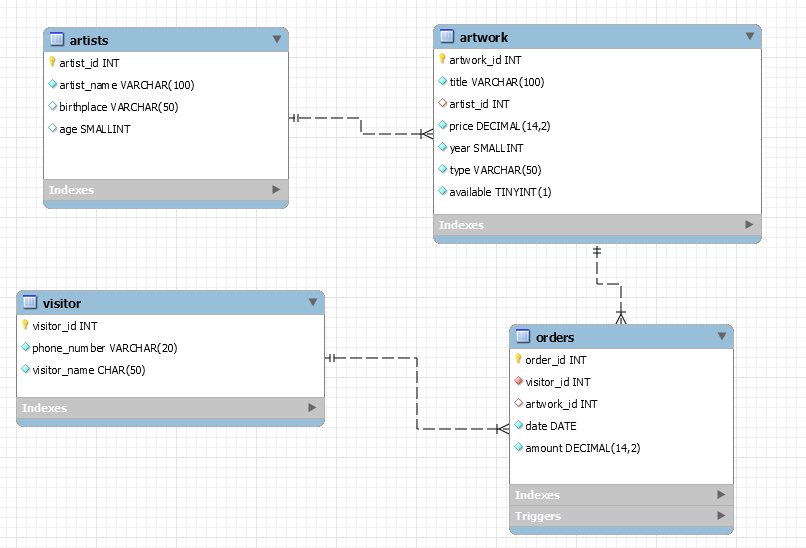


COMMENTS

1. The relation between artist and artwork has total participation on both sides
2. The orders relation has partial participation on both sides
3. The orders relation is converted into its own table in the relational mapping stage



# **SCHEMA**



# **FD and Normalization**

**Functional Dependencies and normal forms of each relation**

1. ARTIST (3NF and BCNF)
   1. artist\_id –
   2. -----> (artist\_name, birthplace, age)
2. ARTWORK (3NF and BCNF)
   1. artwork\_id(PK) -----> (title, artist\_id, price, year)
   2. Title ----------> (artwork\_id, artist\_id, price, year)
3. ORDERS (3NF and BCNF)
   1. order\_id (PK) ------> (visitor\_id, artwork\_id, date, amount)
4. VISITOR (3NF AND BCNF)
   1. visitor\_id(PK) --------> (phone\_number, visitor\_name)

**NORMAL FORM OF OUR DATABASE**

* This database is normalized up to 3NF and also BCNF.

**VIOLATIONS THAT MAY HAPPEN**

* **Violation to 2nd Normal Form:** If we combine artist and artwork into one table along with all their attributes, (artist\_id, artwork\_id ) combined will be the primary key and the only candidate key also. Thus attributes like artist\_name will only depend on a part of the primary key (artist\_id) and the same for attributes of artwork like price will only depend on the artwork\_id. **This is a partial dependency which violates the 2NF.**
* **Violation to 3rd Normal Form:** If we add another attribute called STATE to the artist table which represents the state to which the artist belongs. All the attributes of artist table are currently dependent on artist\_id, but after adding state the birthplace which is the country where the artist is from will depend on the state alone. **This introduces a transitive dependency and violates the 3NF.**

# **DDL SCRIPTS**

create table if not exists artists (

artist\_id INT UNSIGNED auto\_increment,

artist\_name VARCHAR (100) UNIQUE NOT NULL,

birthplace VARCHAR (50),

age SMALLINT,

PRIMARY KEY(artist\_id)

);

create table if not exists artwork (

artwork\_id INT UNSIGNED auto\_increment,

title VARCHAR (100) UNIQUE NOT NULL,

artist\_id INT UNSIGNED,

price DECIMAL (14,2) NOT NULL,

year SMALLINT NOT NULL,

type VARCHAR (50) NOT NULL,

available boolean NOT NULL default true,

PRIMARY KEY(artwork\_id),

FOREIGN KEY(artist\_id)

REFERENCES artists(artist\_id)

ON UPDATE CASCADE

ON DELETE CASCADE

);

create table if not exists visitor (

visitor\_id INT UNSIGNED auto\_increment,

phone\_number VARCHAR (20) UNIQUE NOT NULL,

visitor\_name CHAR (50) NOT NULL,

PRIMARY KEY(visitor\_id)

);

create table if not exists orders (

order\_id INT UNSIGNED AUTO\_INCREMENT,

visitor\_id INT UNSIGNED NOT NULL,

artwork\_id INT UNSIGNED,

date DATE NOT NULL,

amount DECIMAL (14,2) NOT NULL,

PRIMARY KEY(order\_id),

FOREIGN KEY(visitor\_id)

REFERENCES visitor(visitor\_id)

ON DELETE RESTRICT

ON UPDATE CASCADE,

FOREIGN KEY(artwork\_id)

REFERENCES artwork(artwork\_id)

ON DELETE RESTRICT

ON UPDATE CASCADE )

# **Triggers**

The trigger is a database object that is associated with a table. It will be activated when a defined action is executed for the table. The trigger can be executed when you run one of the following MySQL statements on the table: INSERT, UPDATE and DELETE and it can be invoked before or after the event.

Here I have used an AFTER INSERT Trigger on the **orders** table that sets the **available** attribute of the artwork table to **false** once the order for the corresponding artwork is placed and vice versa for the AFTER DELETE trigger.

DELIMITER $$

create

trigger my\_trigger after insert

on orders for each row

begin

update artwork

set available = false

where artwork\_id = NEW.artwork\_id;

end $$

DELIMITER;

DELIMITER $$

create

trigger delete\_trigger after delete

on orders for each row

begin

update artwork

set available = true

where artwork\_id = OLD.artwork\_id;

end $$

DELIMITER;

# **SQL Queries**

# Following are the main (not all) queries including nested and aggregate queries.

**-- All time highest bid**

Select max(amount) from orders;

**-- category wise sales and count of artwork**

select SUM(orders.amount) ,COUNT(\*) as num\_sold, artwork.type

from artwork,orders

where artwork.title = orders.title

group by artwork.type;

**-- sales in this month , this year.**

select \* from orders

where (date between DATE\_FORMAT(NOW() ,'%Y-%m-01') AND NOW() );

select \* from orders

WHERE (date between DATE\_FORMAT(NOW() ,'%Y-01-01') AND NOW() ) ;

**-- sales in any particular year**

select \* from orders

where ( year(date) = 2019);

**-- display all available artwork ordered by year**

select a.artwork\_id,

a.title,

artists.artist\_name,

a.price,

a.year,

a.type,

a.available

from artwork as a

inner join artists using(artist\_id)

where a.available = true

order by year;

**-- artist with maximum sales in order descending artist with max sales at top**

select x.artist\_id , artist\_name , sum(z.amount) as maxSale

from artists x , artwork y , orders

where x.artist\_id = y.artist\_id and y.artwork\_id = z.artwork\_id

group by x.artist\_id

order by maxSale desc;

**-- artists that have not sold any work (nested)**

select \* from artwork;

select artist\_id , artist\_name

from artists

where not exists ( select \* from artwork

where artists.artist\_id = artwork.artist\_id and artwork.available = false );

**-- show visitors who have bought more than 5 times in the past 1 year (nested -2)**

select visitor\_id from visitor

where visitor\_id in ( select visitor\_id from orders

WHERE (date between DATE\_FORMAT(NOW() ,'%Y-01-01') AND NOW() )

group by visitor\_id

having count(\*) > 3

);

**-- visitors who haven’t bought more than 5 times in past 1 year**

select visitor\_id from visitor

where visitor\_id not in ( select visitor\_id from orders

WHERE (date between DATE\_FORMAT (NOW (),'%Y-01-01’) AND NOW ())

group by visitor\_id

having count(\*) > 3

);

# **Conclusion**

The Art Gallery DBMS system shows the robustness and capability of SQL and DBMS abilities. Multiple functions have been achieved using SQL on the DBMS

* Proper layout of the tables and attributes, thereby avoiding redundant data, consistency errors and enabling tables to be in a normalized fashion
* Regular functions like adding, deleting, viewing and updating of data have been implemented using SQL
* Implements some complex queries such as nested and correlated queries. Also variables in SQL have been used while inserting orders for ease of use.
* Constraints and trigger have been used to maintain data integrity.

**LIMITATIONS AND IMPROVEMENTS**

* Currently does not support multiple users and permissions. Could be implemented in the future.
* Could also be combined with front end to make it into a deployable application
* Does not implement complex features such as database sharding and they could be implemented in future.
* Limited to the necessary tables currently, more tables could be added on.