24 November 2021

RHYTHM MUSIC STORE

DATABASE DESIGN AND MAINTENANCE

Submitted By:

Akhil Manual

Athira Tomy

Binali Patel

Praveen Babu

Reshma Nair

**INTRODUCTION**

A structured and organized data storage is crucial for any organization. A database can store huge volumes of data efficiently making it easy to retrieve, add, edit, and delete records. A music store is an example where large numbers of records can be effectively stored in a database thereby helping in its smooth operation.

**Rhythm Music Store** database holds all data related to the storage of music products, customers, and orders. The data is broken down to its smallest entities for quick analysis and easy retrieval of data. Each table holds records specific to a single entity thereby ensuring better readability and efficiency. This also makes it easy to update the product details according to latest addition of albums/products, make effective business analysis based on previous order patterns, update customer details etc. All the tables and their relationships are well-defined and helps in handling any scenario related to the music store business.

**GENERAL SCENARIO**

Rhythm Music Store database is designed to handle different scenarios that the music store might come across in its day-to-day operations. Below are the general scenarios and solutions covered in this design:

1. Display a list of clients that spent more than the average spent by clients in the past month.

**SELECT DISTINCT c.customer\_id, c.first\_name, c.last\_name FROM**

**customers c INNER JOIN orders o ON c.customer\_id = o.customer\_id**

**WHERE payment\_total > (SELECT AVG(payment\_total) FROM orders**

**WHERE order\_date<= NOW() AND order\_date > NOW() - INTERVAL 1 MONTH)**

**ORDER BY customer\_id;**

We have used subquery and inner join for this scenario. A subquery is a SQL query nested inside a larger query. It may occur in: A SELECT clause, A FROM clause and A WHERE clause. A subquery is usually added within the WHERE Clause of another SQL SELECT statement. You can use the comparison operators, such as >, <, or =. The comparison operator can also be a multiple-row operator, such as IN, ANY, or ALL. The inner query executes first before its parent query so that the results of an inner query can be passed to the outer query.

In the above query we have created inner query which will return the average of the payment total of all the clients in the past month. The outer query is displaying the customer id, customer first name and last name who are having payment total greater than the results of an inner query.

DISTINCT is used with customer id to avoid duplication.

1. The top sold products and least sold products over a week.

The below view query creates a view for the sum of order quantity for each album over a week.

**CREATE VIEW sum\_of\_order\_qty\_view AS**

**SELECT a.album\_name AS album\_name, sum(o.order\_qty) AS sum1**

**FROM order\_items o**

**INNER JOIN albums a**

**ON o.album\_id = a.album\_id**

**WHERE o.order\_id IN ( SELECT order\_id FROM orders WHERE order\_date <= NOW() AND order\_date > NOW() - INTERVAL 1 WEEK)**

**GROUP BY a.album\_id;**

The below query creates a view which has the maximum and minimum order quantity.

**CREATE VIEW max\_min\_product\_view AS**

**(SELECT album\_name, sum1 FROM sum\_of\_order\_qty\_view**

**WHERE sum1 = (SELECT MAX(sum1) MaxProductFROMsum\_of\_order\_qty\_view)**

**UNION**

**SELECT album\_name, sum1 FROM sum\_of\_order\_qty\_view**

**WHERE sum1 = (SELECT MIN(sum1) MinProductFROMsum\_of\_order\_qty\_view));**

This query will display the album name and quantity of the maximum and minimum albums ordered by the customers over a week

**SELECT album\_name AS 'Album Name', sum1 AS 'Total Products' FROM max\_min\_product\_view;**

1. The maximum price of products in the same genre (for example, rock, pop, country, hip-hop). Use GROUP BY to list all the genres and their maximum price.

**SELECT MAX(album\_price), genre\_name**

**FROM albums a JOIN genre g ON a.genre\_id = g.genre\_id**

**GROUP BY g.genre\_id;**

This query performs an inner join on the tables genre and albums, groups the records based on the genre and calculates the albums with the maximum price corresponding to each genre.

1. List how many customers the system has by location (Country, Province, andCity), and then sort them.

**SELECT COUNT(customer\_id) AS no\_of\_customers, country\_name, province\_name, city\_name**

**FROM customers c JOIN addresses a ON c.address\_id = a.address\_id**

**JOIN cities ci ON a.city\_id = ci.city\_id**

**JOIN provinces p ON p.province\_id = ci.province\_id**

**JOIN countries co ON co.country\_id = p.country\_id**

**GROUP BY co.country\_id, p.province\_id, ci.city\_id**

**ORDER BY country\_name, province\_name, city\_name;**

In this scenario, we join the customers, addresses, cities, provinces and countries tables to get the location corresponding to each customer and then calculate the total number of customers in a certain location. The result is then sorted in the ascending order.

1. List how many products the store has sold for a particular month.

**SELECT MONTHNAME(o.order\_date) AS Month,SUM(oi.order\_qty)**

**AS  Total\_No\_of\_Sold\_Product**

**FROM orders o**

**LEFT JOIN order\_items oi**

**ON o.order\_id = oi.order\_id**

**GROUP BY month**

**ORDER BY o.order\_date;**

In this query we have used Left Join to join two tables’ values such as Order date from orders table and order quantity from order\_items table. Then, we used MONTHNAME function to display name of the month in corresponding date and SUM function used to find out the total number of quantities in particular month. Hence, GROUP BY clause used for group together each month and orders in that month. Finally, ORDER BY clause is used to sort the data according to ascending order of date. By default, ORDER BY clause sorts the data in ascending order.

1. List how many distinct albums each singer has.

**SELECT CONCAT(art.first\_name,' ',art.last\_name) AS Singer,**

**COUNT(album\_id) AS Number\_of\_songs**

**FROM albums JOIN artists art**

**USING(artist\_id)**

**GROUP BY artist\_id**

**ORDER BY Singer;**

The above query is used to find the total number of different albums in each singer has.

In this query we have used CONCAT function and COUNT function. The CONCAT function is used to merge the first name and last name of the singer. The number of distinct albums find out by the help of COUNT function. Here, values of albums and artists tables joined by the use of left join and distinct albums of each singer is calculated by the use of artist\_id .’Using’ clause is used for this purpose and GROUP BY clause groups all singers together then sort the data by ORDER BY clause. It is in ascending order of singer.

1. List how many copies of an album are available of a particular singer.

**SELECT a.artist\_id, SUM(a.album\_qty), ar.first\_name, ar.last\_name**

**FROM albums as a INNER JOIN artists AS ar**

**ON a.artist\_id = ar.artist\_id**

**WHERE a.artist\_id = '1';**

Here in this query, we join two tables i.e artists and albums to get the total number of copies published by a particular artist with id and name. We are displaying in the output artist id, artist name and the sum of the total copies he or she has published.

**SPECIFIC SCENARIOS**

1. Display which customer has ordered which album and the total payment given by the customer.

**SELECT c.customer\_id, c.first\_name, c.last\_name, o.order\_id, a.album\_id, a.album\_name, o.payment\_total**

**from customers as c**

**INNER JOIN orders as o on c.customer\_id = o.customer\_id**

**INNER JOIN order\_items as oi on o.order\_id = oi.order\_id**

**INNER JOIN albums as a on a.album\_id = oi.album\_id**

**GROUP BY a.album\_price > 10;**

Here in this query, we are joining tables that are customers, orders, order\_items and albums. We are implementing this query to show that which customer has ordered which album. So, using the SELECT query we are first writing what we need as output getting displayed and then we are applying inner joins within the tables so that we can get the name and id of customer and what album he or she has ordered.

1. List all customers and show their city name and check customers from Toronto city using the IF function.

**SELECT  first\_name, last\_name, city\_name,**

**IF(city\_name = 'Toronto', 'Yes', 'No') AS Is\_city\_Toronto**

**FROM customers**

**JOIN addresses**

**ON customers.customer\_id = addresses.address\_id**

**JOIN cities**

**ON addresses.address\_id = cities.city\_id;**

The above query uses the IF function to test a condition and return one value if the condition is true or another value if the condition is false. In the above example, the query uses the IF function to return a string value of “Yes” if the customer’s city\_name column is equal to a value of “Toronto”. Otherwise, the IF function returns a value of “No”.

1. Create Procedure to display payment total of orders for customer\_id =1

**DELIMITER //**

**CREATE PROCEDURE GetTotalOfPayment()**

**Begin**

**DECLARE sum\_payment\_total DECIMAL(9,2);**

**SELECT SUM(payment\_total) INTO sum\_payment\_total**

**FROM orders**

**WHERE customer\_id=1;**

**SELECT CONCAT('$', sum\_payment\_total) AS 'Total of payment';**

**END //**

**DELIMITER ;**

This query creates a stored procedure to display the payment total of orders for the customer\_id = 1. In this procedure, I made a variable and used the SUM function to calculate the sum of all payment\_total for the customer\_id=1

1. Create view to update the columns in the table order\_items:

**CREATE OR REPLACE VIEW order\_data AS**

**SELECT order\_item\_id, price\_item, order\_id, album\_id FROM order\_items;**

**UPDATE order\_data SET price\_item = 200 WHERE order\_item\_id =2;**

A view is a SELECT statement that’s stored in the database as a database object. The above query creates a view of base table order\_items. Users can manipulate the data using this view. The given query is an example of performing an update in order\_items table using the view named order\_data.

1. Trigger on countries table to update the country\_name for a specific country\_id:

**DROP TRIGGER IF EXISTS countries\_before\_update;**

**DELIMITER //**

**CREATE TRIGGER countries\_before\_update**

**BEFORE UPDATE ON countries FOR EACH ROW**

**BEGIN**

**SET NEW.country\_name = upper (NEW.country\_name);**

**END //**

**UPDATE countries**

**SET country\_name = 'Germany'**

**WHERE country\_id = 1;**

**SELECT country\_id, country\_name**

**FROM countries**

**WHERE country\_id = 1;**

**CONCLUSION**

Rhythm Music Store database effectively manages to store the huge data related to the music store, its albums, orders, and customers. All entities have been converted to corresponding tables and the tables have been normalized to 3rd form. Various relationships among the entities have also been well-defined in the design. The database design provides a simple solution to handle different scenarios related to day-to-day operations of the music store and its business analysis.

Rhythm Music Store database design also includes stored procedures, views and triggers for quick calculations and data update. The solution also comes with queries to handle frequently encountered situations and analysis and a quick retrieval of data that involves combining different related entities.

**APPENDIX**

**TABLES**

**artists**

This table holds the data related to all the artists. Given below are the column definitions:

*artist\_id*: Primary Key, INT datatype, Auto Increment

*first\_name*: VARCHAR(55) datatype

*last\_name*: VARCHAR(55) datatype

**cities**

This table contains the records for all cities in a province. Column definitions:

*city\_id*: Primary Key, INT datatype, Auto Increment

*city\_name*: VARCHAR(45) datatype

*province\_id*: Foreign Key, INT datatype

Foreign Keys: fk\_cities\_provinces1 (related to the table *provinces*)

**addresses**

This table gives the address details of all the customers. It has one to many relationship with cities and customers table. It consists of following fields:

*address\_id:*This is a primary key for this table. Its datatype is integer. It cannot be null and must be unique for all the address. It auto increments its value whenever a new row is added.

*address\_line1:* This field stores the address of the customer. Its datatype is VARCHAR.

*postal\_code:*This column stores the zip code of the address. Its datatype is VARCHAR.

*city\_id:* This is the foreign key of the table. A foreign key is a column or group of columns in a relational database table that provides a link between data in two tables. Its datatype is integer. This column links addresses and cities table.

**albums**

This table contains all the details about the album which is available in the Music Store. It consists of following fields:

*album\_id*:This is a primary key for this table. Its datatype is integer. A primary key is used to ensure data in the specific column is unique. It cannot be null and must be unique for all the address. It auto increments its value whenever a new row is added.

*album\_name:* This field display all the name of the albums available in the Music store. Its datatype is VARCHAR.

*album\_qty:* This column displays the quantity of albums. Its datatype is INT.

*album\_price:* This field has the price of album. Its datatype is DOUBLE.

Foreign Keys: artist\_id, genre\_id, rating\_id, language\_id, product\_type\_id these are the foreign keys of this table which links albums table to artists, genre, ratings, languages, products\_type tables.

**provinces**

This table stores the records for all province in the address. Column definitions:

*Province\_id*:It is the primary key of this table and integer datatype is used to this field.

The auto increment and not null constraints added in this field.The value of the province \_id is unique.

*province\_name* : This field is used to collect the names of each province in the address.It is varchar datatype and 45 is the size of this field.

Foreign Keys : country\_id is the only foreign key is in province table and it is integer datatype.This filed used to link this province table with country table.

**countries**

This table used to stores all the values regarding country details .The column fields are:

*country\_id*: This field is the primary key of this table .Integer datatype is used for this field and it must be not null ,unique values.It’s value incremented automatically if there is new row added.

*country\_name*: In this field we store name of the each countries and Varchar used as datatype of this field and 45 is the size of this field

**customers**

This table used to collect information regarding customers of the music store. This

Information are contact information of each customer. The column definitions are:

*customer\_id*: This field is the primary key of this table and it is integer datatype.in this field

we added not null constrains and auto increment constraints so that this field cannot be null and value will be incremented automatically when new record add.

*first\_name*: This field we used to store the first name of the customer.it is a varchar datatype and size is 45.

*last\_name*:In this field it stores last name of the customers of the music store.It is also varachar and size is 45.

Foreign Keys : address\_id this is the only foreign key is in customers table and it is integer datatype.This key helps to link this customer table with address table.

**genre**

This table gives data regarding the genre of the album. The column definitions are:

*genre\_id:* This field gives a unique id to the genre. It is the primary key of the genre table and it auto increments the id value. Moreover, the datatype is integer INT.

*genre\_name:* This field displays the name of the genre input by the user. The datatype is VARCHAR.

**languages**

This table gives data regarding the language of the album. For example the album is written in which language like French, German, Spanish etc. The column definitions are:

*language\_id:* This field gives a unique id to the language. It is the primary key of the language table and it auto increments the id value. Moreover, the datatype is integer INT.

*language\_name:* This field displays the name of the language input by the user. The datatype is VARCHAR.

**product\_type**

This table gives data regarding the type of the product. The column definitions are:

*product\_type\_id:*This field gives a unique id to the product type. It is the primary key of the product\_type table and it auto increments the id value. Moreover, the datatype is integer INT.

*product\_type:* This field displays the name of which type of product is input by the user. The datatype is VARCHAR.

**order\_items**

order\_items table store all the details about the purchased item from the Music Store . It contains the following fields:

*order\_item\_id*:  Primary Key, INT data type, Auto Increment

*order\_qty:* INT datatype

*price\_item :*DOUBLE datatype .

*order\_id* : Foreign Key, INT datatype [This foreign key links this table to orders table by 1:N relationship]

*album\_id* : Foreign Key, INT datatype [This foreign key links this table to album table by 1:N relationship]

**orders**

orders table store all the details about the orders in the Music Store. It contains the following fields:

*order \_id*:  Primary Key, INT data type, Auto Increment

*order\_date:* DATE datatype

*payment\_date:* DATE datatype

*payment\_total :*DECIMAL datatype .

*customer\_id* : Foreign Key, INT datatype [This foreign key links this table to customers table by 1:N relationship]

**ratings**

This table stores all the details about the customer ratings of albums in the Music Store. It contains the following fields:

*rating \_id*:  Primary Key, INT data type, Auto Increment

*rating\_value:* INT data type

*rating\_date:* DATE data type

*comments:*VARCHAR  data type .

*customer\_id* : Foreign Key, INT datatype [This foreign key links this table to customers table by 1:N relationship]