

(CS 425-02) DATABASE ORGANISATION

Homework 1.1

Group Members Details

Name	CWID	EMAIL
DEVESH PATEL	A20548274	dpatel219@hawk.iit.edu
PURVIT ASHESH PATEL	A20551053	ppatel180@hawk.iit.edu
VISHWASHREE CHANNAA REDDY	A20556543	vishwashreech@hawk.iit.edu

Deliverable 2

- 1) CREATE TABLE subscription (
SubscriptionID int AUTO_INCREMENT,
SubscriptionType varchar(255) NOT NULL,
Price DECIMAL(10,2) NOT NULL,
PRIMARY KEY (SubscriptionID)
);

The screenshot shows a database management tool interface. The top section contains SQL code for creating and querying the 'subscription' table. The bottom section displays a 'Result Grid' with 10 rows of data.

```
2 • USE musicsystem;  
3 • drop database musicsystem;  
4  
5 • DROP TABLE IF EXISTS subscription;  
6 • CREATE TABLE subscription (  
7     SubscriptionID int AUTO_INCREMENT,  
8     SubscriptionType varchar(255) NOT NULL,  
9     Price DECIMAL(10,2) NOT NULL,  
10    PRIMARY KEY (SubscriptionID)  
11 );  
12 • SELECT * FROM Subscription;
```

SubscriptionID	SubscriptionType	Price
1	Basic	5.99
2	Standard	9.99
3	Premium	14.99
4	Family	19.99
5	Student	4.99
6	Duo	12.99
7	Annual Basic	59.99
8	Annual Standard	99.99
9	Annual Premium	149.99
10	Lifetime	299.99

- 2) CREATE TABLE artist (
ArtistID int AUTO_INCREMENT,
ArtistName varchar(255) NOT NULL,
PRIMARY KEY (ArtistID)
);

The screenshot shows a database management tool interface. The top section contains SQL code for creating and querying the 'artist' table. The bottom section displays a 'Result Grid' with 15 rows of data.

```
14 • DROP TABLE IF EXISTS artist;  
15 • CREATE TABLE artist (  
16     ArtistID int AUTO_INCREMENT,  
17     ArtistName varchar(255) NOT NULL,  
18     PRIMARY KEY (ArtistID)  
19 );  
20 • SELECT * FROM artist;
```

ArtistID	ArtistName
1	Taylor Swift
2	The Beatles
3	Drake
4	Billie Eilish
5	Ed Sheeran
6	Ariana Grande
7	Elton John
8	Beyoncé
9	Bob Dylan
10	Kanye West
11	Adele
12	Bruno Mars
13	Lady Gaga
14	The Rolling St...
15	Rihanna

3) CREATE TABLE genre (
GenreID int AUTO_INCREMENT,
GenreName varchar(255) NOT NULL,
PRIMARY KEY (GenreID)
);

The screenshot shows a database management tool interface. The top section displays SQL code for creating a table named 'genre'. The code is as follows:

```
22 • DROP TABLE IF EXISTS genre;  
23 • CREATE TABLE genre (  
24     GenreID int AUTO_INCREMENT,  
25     GenreName varchar(255) NOT NULL,  
26     PRIMARY KEY (GenreID)  
27 );  
28 • SELECT * FROM genre;  
29
```

Below the code editor, the 'Result Grid' is visible, showing the data returned by the SELECT query. The grid has two columns: 'GenreID' and 'GenreName'. The data is as follows:

GenreID	GenreName
1	Pop
2	Rock
3	Hip-Hop
4	Indie
5	Electronic
6	Country
7	R&B
8	Jazz
9	Classical
10	Metal
11	Folk
12	Blues
13	Reggae
14	Soul
15	Punk
NULL	NULL

The interface also includes a toolbar with various icons for editing and viewing data, and a sidebar on the right with options like 'Result Grid', 'Form Editor', 'Field Types', and 'Query Stats'.

4) CREATE TABLE user (
 UserID int AUTO_INCREMENT,
 Username varchar(255) NOT NULL,
 Email varchar(255) NOT NULL,
 Password varchar(255) NOT NULL,
 SubscriptionID int,
 Subscription_Duration varchar(255) not null,
 PRIMARY KEY (UserID),
 FOREIGN KEY (SubscriptionID) REFERENCES subscription(SubscriptionID)
);

The screenshot shows a database management interface. The top pane contains SQL code for creating a table and querying it. The bottom pane shows a result grid with 15 rows of data.

SQL Code:

```

31 CREATE TABLE user (
32   UserID int AUTO_INCREMENT,
33   Username varchar(255) NOT NULL,
34   Email varchar(255) NOT NULL,
35   Password varchar(255) NOT NULL,
36   SubscriptionID int,
37   Subscription_Duration varchar(255) not null,
38   PRIMARY KEY (UserID),
39   FOREIGN KEY (SubscriptionID) REFERENCES subscription(SubscriptionID)
40 );
41 SELECT * FROM user;
42

```

Result Grid:

	UserID	Username	Email	Password	SubscriptionID	Subscription_Duration
1	1	AlexSmith	alexsmith@example.com	pass123	3	1 month
2	2	Jamie_F	jamief@example.com	pass124	2	6 month
3	3	ChrisP	chrisp@example.com	pass125	5	12 month
4	4	JordanK	jordank@example.com	pass126	10	1 month
5	5	SamT	samt@example.com	pass127	8	3 month
6	6	PatW	patw@example.com	pass128	7	6 month
7	7	TaylorR	taylorr@example.com	pass129	3	12 month
8	8	MorganC	morganc@example.com	pass130	6	3 month
9	9	CaseyB	caseyb@example.com	pass131	1	3 month
10	10	DrewA	drewa@example.com	pass132	4	1 month
11	11	JordanM	jordanm@example.com	pass133	4	12 month
12	12	CharlieG	charlieg@example.com	pass134	9	6 month
13	13	SydneyJ	sydneyj@example.com	pass135	8	1 month
14	14	AlexisT	alexis@example.com	pass136	6	6 month
15	15	TaylorM	taylorm@example.com	pass137	1	6 month

5) CREATE TABLE album (
 AlbumID int AUTO_INCREMENT,
 AlbumTitle varchar(255) NOT NULL,
 ReleaseYear int, -- Assuming only the year is stored
 PRIMARY KEY (AlbumID)
);

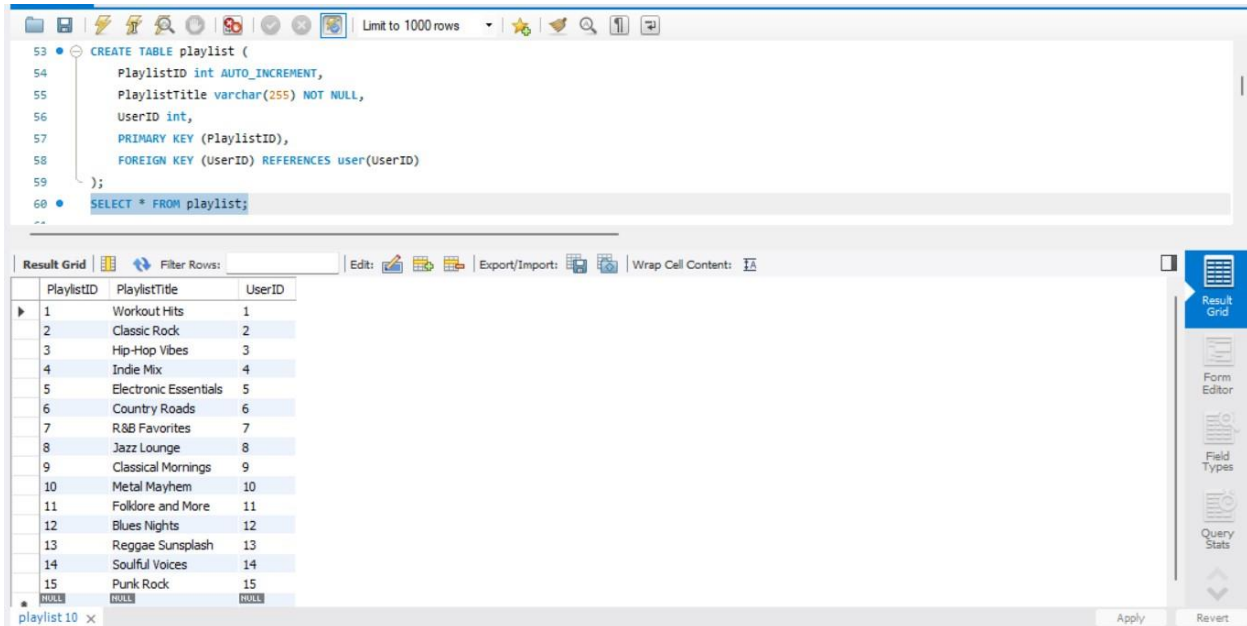
The screenshot shows a database management tool interface. The top pane displays SQL code for creating and querying a table named 'album'. The bottom pane shows the 'Result Grid' with 15 rows of data. The columns are 'AlbumID', 'AlbumTitle', and 'ReleaseYear'. The data includes albums like 'Abbey Road', 'Scorpion', 'When We All Fall Asleep, Where Do We Go?', 'Divide', 'Thank U, Next', 'Goodbye Yellow Brick Road', 'Lemonade', 'Highway 61 Revisited', 'My Beautiful Dark Twisted Fantasy', '21', '24K Magic', 'Chromatica', 'Exile On Main St.', and 'Anti'. A final row shows 'NULL' values for all three columns. The interface also includes a toolbar at the top, a 'Filter Rows' section, and a sidebar on the right with options like 'Form Editor', 'Field Types', and 'Query Stats'.

```
43 • DROP TABLE IF EXISTS album;  
44 • CREATE TABLE album (  
45     AlbumID int AUTO_INCREMENT,  
46     AlbumTitle varchar(255) NOT NULL,  
47     ReleaseYear int, -- Assuming only the year is stored  
48     PRIMARY KEY (AlbumID)  
49 );  
50 • SELECT * FROM album;
```

AlbumID	AlbumTitle	ReleaseYear
2	Abbey Road	1969
3	Scorpion	2018
4	When We All Fall Asleep, Where Do We Go?	2019
5	Divide	2017
6	Thank U, Next	2019
7	Goodbye Yellow Brick Road	1973
8	Lemonade	2016
9	Highway 61 Revisited	1965
10	My Beautiful Dark Twisted Fantasy	2010
11	21	2011
12	24K Magic	2016
13	Chromatica	2020
14	Exile On Main St.	1972
15	Anti	2016
• NULL	NULL	NULL

album 7 x Apply Revert

6) CREATE TABLE playlist (
 PlaylistID int AUTO_INCREMENT,
 PlaylistTitle varchar(255) NOT NULL,
 UserID int,
 PRIMARY KEY (PlaylistID),
 FOREIGN KEY (UserID) REFERENCES user(UserID)
);



The screenshot shows a database management interface. The top section contains a SQL editor with the following code:

```

53 CREATE TABLE playlist (
54     PlaylistID int AUTO_INCREMENT,
55     PlaylistTitle varchar(255) NOT NULL,
56     UserID int,
57     PRIMARY KEY (PlaylistID),
58     FOREIGN KEY (UserID) REFERENCES user(UserID)
59 );
60 SELECT * FROM playlist;
  
```

Below the editor is a "Result Grid" showing 15 rows of data. The columns are PlaylistID, PlaylistTitle, and UserID. The data is as follows:

PlaylistID	PlaylistTitle	UserID
1	Workout Hits	1
2	Classic Rock	2
3	Hip-Hop Vibes	3
4	Indie Mix	4
5	Electronic Essentials	5
6	Country Roads	6
7	R&B Favorites	7
8	Jazz Lounge	8
9	Classical Mornings	9
10	Metal Mayhem	10
11	Folklore and More	11
12	Blues Nights	12
13	Reggae Sunsplash	13
14	Soulful Voices	14
15	Punk Rock	15

At the bottom of the result grid, there is a row with three "NULL" values. The interface also includes a toolbar with various icons and a sidebar on the right with options like "Form Editor", "Field Types", and "Query Stats".

```

7) CREATE TABLE songs (
    SongID int AUTO_INCREMENT,
    SongTitle varchar(255) NOT NULL,
    Duration int, -- Assuming duration in seconds
    ReleaseYear int, -- Assuming only the year is stored
    PRIMARY KEY (SongID)
);

```

The screenshot shows a database management interface. The top section displays a SQL script that has been executed. The script creates a table named 'songs' with the following specifications: SongID is an auto-incrementing integer and the primary key; SongTitle is a variable-length string of up to 255 characters, not null; Duration is an integer representing seconds; and ReleaseYear is an integer representing the year. Below the script, a 'SELECT * FROM songs;' query has been executed, resulting in a table with 15 rows of data.

	SongID	SongTitle	Duration	ReleaseYear
1	1	Cardigan	3:53	2020
2	2	Come Together	3:50	1969
3	3	God's Plan	5:23	2018
4	4	Bad Guy	5:50	2019
5	5	Shape of You	3:41	2017
6	6	7 rings	4:19	2019
7	7	Tiny Dancer	3:59	1973
8	8	Halo	3:34	2016
9	9	Like a Rolling Stone	4:45	1965
10	10	Power	4:32	2010
11	11	Rolling in the Deep	5:24	2011
12	12	That's What I ...	3:24	2016
13	13	Stupid Love	4:23	2020
14	14	Paint It Black	5:11	1972
15	15	Work	3:21	2016

The interface also includes a toolbar at the top with various icons for file operations and a sidebar on the right with options like 'Result Grid', 'Form Editor', 'Field Types', and 'Query Stats'. The bottom status bar indicates the table 'songs' has 12 rows and is in 'Read Only' mode.

8) CREATE TABLE SongPlaylist (

 SongID int NOT NULL,

 PlaylistID int NOT NULL,

 PRIMARY KEY (SongID, PlaylistID),

 FOREIGN KEY (SongID) REFERENCES songs(SongID),

 FOREIGN KEY (PlaylistID) REFERENCES playlist(PlaylistID)

);

The screenshot shows a database management interface. The top pane displays the SQL code for creating the SongPlaylist table. The bottom pane shows the result grid with 15 rows of data.

```
73 CREATE TABLE SongPlaylist (  
74     SongID int NOT NULL,  
75     PlaylistID int NOT NULL,  
76     PRIMARY KEY (SongID, PlaylistID),  
77     FOREIGN KEY (SongID) REFERENCES songs(SongID),  
78     FOREIGN KEY (PlaylistID) REFERENCES playlist(PlaylistID)  
79 );  
80 SELECT * FROM SongPlaylist;
```

	SongID	PlaylistID
1	1	1
2	1	1
3	1	1
4	2	2
5	2	2
6	2	2
7	3	3
8	3	3
9	3	3
10	4	4
11	4	4
12	4	4
13	5	5
14	5	5
15	5	5
16	NULL	NULL

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content: |

SongPlaylist 15 x | Apply | Revert


```

9) CREATE TABLE SongArtist (

    SongID int NOT NULL,

    ArtistID int NOT NULL,

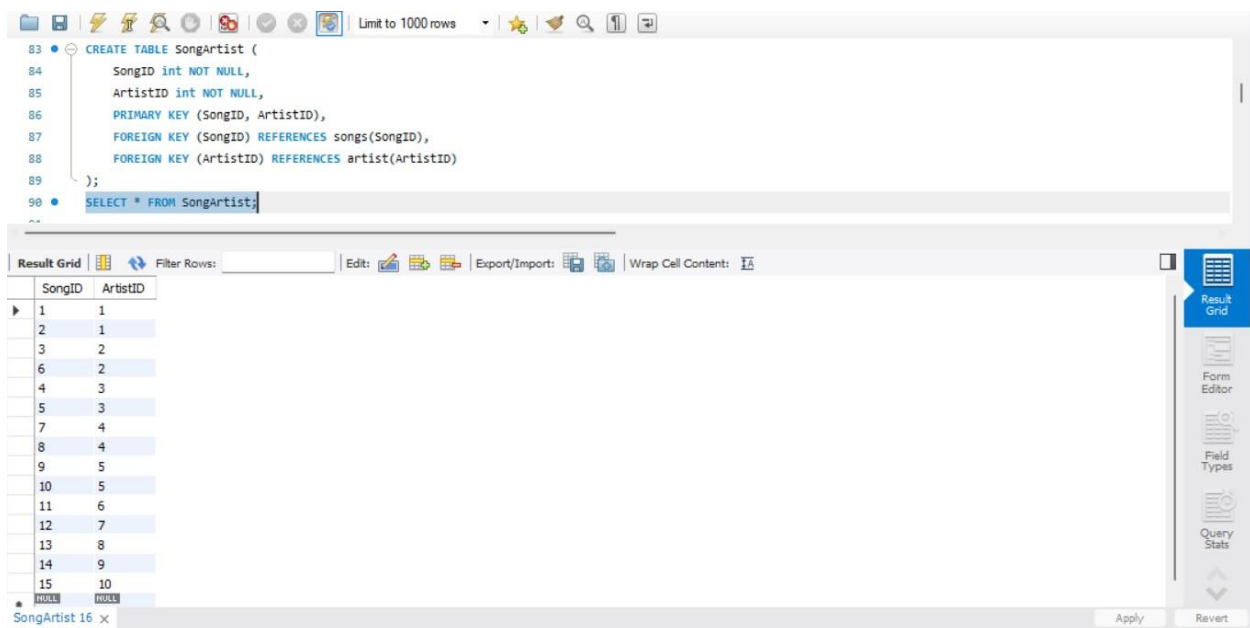
    PRIMARY KEY (SongID, ArtistID),

    FOREIGN KEY (SongID) REFERENCES songs(SongID),

    FOREIGN KEY (ArtistID) REFERENCES artist(ArtistID)

);

```



The screenshot shows a database management interface. The top pane displays the SQL query for creating the 'SongArtist' table. The bottom pane shows the 'Result Grid' with 15 rows of data. The columns are 'SongID' and 'ArtistID'. The data is as follows:

SongID	ArtistID
1	1
2	1
3	2
6	2
4	3
5	3
7	4
8	4
9	5
10	5
11	6
12	7
13	8
14	9
15	10

At the bottom of the interface, there are buttons for 'Apply' and 'Revert'.

```
10) CREATE TABLE AlbumGenre (  
  
    AlbumID int NOT NULL,  
  
    GenreID int NOT NULL,  
  
    PRIMARY KEY (AlbumID, GenreID),  
  
    FOREIGN KEY (AlbumID) REFERENCES album(AlbumID),  
  
    FOREIGN KEY (GenreID) REFERENCES genre(GenreID)  
  
);
```

The screenshot shows a database management tool interface. The top pane displays the SQL query for creating the 'AlbumGenre' table. The bottom pane shows the 'Result Grid' with 17 rows of data. The first 15 rows contain integer values for 'AlbumID' and 'GenreID', while the last two rows contain 'NULL' values. The right sidebar includes options for 'Result Grid', 'Form Editor', 'Field Types', and 'Query Stats'.

```
93 • CREATE TABLE AlbumGenre (  
94     AlbumID int NOT NULL,  
95     GenreID int NOT NULL,  
96     PRIMARY KEY (AlbumID, GenreID),  
97     FOREIGN KEY (AlbumID) REFERENCES album(AlbumID),  
98     FOREIGN KEY (GenreID) REFERENCES genre(GenreID)  
99 );  
100 • SELECT * FROM AlbumGenre;
```

AlbumID	GenreID
1	1
6	1
11	1
2	2
7	2
12	2
3	3
8	3
13	3
4	4
9	4
14	4
5	5
10	5
15	5
NULL	NULL
AlbumGenre 17	

11) CREATE TABLE SongGenre (

SongID int NOT NULL,

GenreID int NOT NULL,

PRIMARY KEY (SongID, GenreID),

FOREIGN KEY (SongID) REFERENCES songs(SongID),

FOREIGN KEY (GenreID) REFERENCES genre(GenreID)

);

The screenshot shows a database management interface. The top panel displays the SQL code for creating the SongGenre table. The bottom panel shows the Result Grid with 15 rows of data.

```
103 CREATE TABLE SongGenre (  
104     SongID int NOT NULL,  
105     GenreID int NOT NULL,  
106     PRIMARY KEY (SongID, GenreID),  
107     FOREIGN KEY (SongID) REFERENCES songs(SongID),  
108     FOREIGN KEY (GenreID) REFERENCES genre(GenreID)  
109 );  
110 SELECT * FROM SongGenre;
```

SongID	GenreID
1	1
2	1
11	1
3	2
4	2
12	2
5	3
6	3
13	3
7	4
8	4
14	4
9	5
10	5
15	5

Result Grid: SongGenre 18 x

12) CREATE TABLE SongAlbum (

SongID int NOT NULL,

AlbumID int NOT NULL,

PRIMARY KEY (SongID, AlbumID),

FOREIGN KEY (SongID) REFERENCES songs(SongID),

FOREIGN KEY (AlbumID) REFERENCES album(AlbumID)

);

The screenshot shows a database management interface. The top section contains a SQL editor with the following code:

```
113 CREATE TABLE SongAlbum (  
114     SongID int NOT NULL,  
115     AlbumID int NOT NULL,  
116     PRIMARY KEY (SongID, AlbumID),  
117     FOREIGN KEY (SongID) REFERENCES songs(SongID),  
118     FOREIGN KEY (AlbumID) REFERENCES album(AlbumID)  
119 );  
120 SELECT * FROM SongAlbum;
```

The bottom section displays a 'Result Grid' with the following data:

SongID	AlbumID
1	1
2	1
3	2
4	2
5	3
6	3
7	4
8	4
9	5
10	5
11	6
12	6
13	7
14	7
15	8

The interface includes a toolbar at the top with icons for file operations, a 'Limit to 1000 rows' dropdown, and a 'Filter Rows' field. The bottom right corner has 'Apply' and 'Revert' buttons.

```
13) CREATE TABLE ArtistAlbum (  
  
    ArtistID int NOT NULL,  
  
    AlbumID int NOT NULL,  
  
    PRIMARY KEY (ArtistID, AlbumID),  
  
    FOREIGN KEY (ArtistID) REFERENCES artist(ArtistID),  
  
    FOREIGN KEY (AlbumID) REFERENCES album(AlbumID)  
  
);
```

The screenshot shows a database management tool interface. The top pane displays the SQL code for creating the ArtistAlbum table, which includes a primary key and two foreign key constraints. The bottom pane shows the result grid with 15 rows of data, where ArtistID and AlbumID are identical for each row. The right sidebar contains navigation options like Result Grid, Form Editor, Field Types, and Query Stats.

```
123 • CREATE TABLE ArtistAlbum (  
124     ArtistID int NOT NULL,  
125     AlbumID int NOT NULL,  
126     PRIMARY KEY (ArtistID, AlbumID),  
127     FOREIGN KEY (ArtistID) REFERENCES artist(ArtistID),  
128     FOREIGN KEY (AlbumID) REFERENCES album(AlbumID)  
129 );  
130 • SELECT * FROM ArtistAlbum;
```

	ArtistID	AlbumID
1	1	1
1	2	2
2	3	3
2	4	4
3	5	5
3	6	6
4	7	7
4	8	8
5	9	9
5	10	10
6	11	11
6	12	12
7	13	13
7	14	14
8	15	15
8	15	15

ArtistAlbum 20 x

Apply Revert

Creating Index:

-- create index for user table

```
CREATE INDEX idx_username ON user(Username);
```

```
CREATE INDEX idx_email ON user(Email);
```

```
CREATE INDEX idx_subscription_id ON user(SubscriptionID);
```

```
132 -- create index for user table
133 CREATE INDEX idx_username ON user(Username);
134 CREATE INDEX idx_email ON user(Email);
135 CREATE INDEX idx_subscription_id ON user(SubscriptionID);
```

Output

#	Time	Action	Message
✓ 1	18:57:29	CREATE INDEX idx_username ON user(Username)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
✓ 2	18:57:30	CREATE INDEX idx_email ON user(Email)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
✓ 3	18:57:30	CREATE INDEX idx_subscription_id ON user(SubscriptionID)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0

-- create index for role table

```
CREATE INDEX idx_artist_name ON artist(ArtistName);
```

```
137 -- create index for role table
138 CREATE INDEX idx_artistname ON artist(ArtistName);
```

Output

#	Time	Action	Message
✓ 1	20:27:23	CREATE INDEX idx_artistname ON artist(ArtistName)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0

-- create index for genre table

```
CREATE INDEX idx_genre_name ON genre(GenreName);
```

```
139 -- create index for genre table
140 CREATE INDEX idx_genre_name ON genre(GenreName);
```

Output

#	Time	Action	Message
✓ 1	19:00:44	CREATE INDEX idx_genre_name ON genre(GenreName)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0

-- create index for album table

CREATE INDEX idx_album_title ON album(AlbumTitle);

CREATE INDEX idx_album_release_year ON album(ReleaseYear);

```
143 -- create index for album table
144 • CREATE INDEX idx_album_title ON album(AlbumTitle);
145 • CREATE INDEX idx_album_release_year ON album(ReleaseYear);
146
147 -- create index for songs table
```

Output

#	Time	Action	Message
1	19:01:46	CREATE INDEX idx_album_title ON album(AlbumTitle)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
2	19:01:46	CREATE INDEX idx_album_release_year ON album(ReleaseYear)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0

-- create index for songs table

CREATE INDEX idx_song_title ON songs(SongTitle);

CREATE INDEX idx_duration ON songs(Duration);

CREATE INDEX idx_songs_release_year ON songs(ReleaseYear);

```
147 -- create index for songs table
148 • CREATE INDEX idx_song_title ON songs(SongTitle);
149 • CREATE INDEX idx_duration ON songs(Duration);
150 • CREATE INDEX idx_songs_release_year ON songs(ReleaseYear);
```

Output

#	Time	Action	Message
1	19:03:12	CREATE INDEX idx_song_title ON songs(SongTitle)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
2	19:03:12	CREATE INDEX idx_duration ON songs(Duration)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
3	19:03:12	CREATE INDEX idx_songs_release_year ON songs(ReleaseYear)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0

-- create index for playlist table

CREATE INDEX idx_playlist_title ON playlist(PlaylistTitle);

CREATE INDEX idx_user_id ON playlist(UserID);

```
152 -- create index for playlist table
153 • CREATE INDEX idx_playlist_title ON playlist(PlaylistTitle);
154 • CREATE INDEX idx_user_id ON playlist(UserID);
155
```

Output

#	Time	Action	Message
1	19:06:01	CREATE INDEX idx_playlist_title ON playlist(PlaylistTitle)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
2	19:08:20	CREATE INDEX idx_user_id ON playlist(UserID)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0

-- create index for subscription table

CREATE INDEX idx_subscription_type ON subscription(SubscriptionType);

```
156 -- create index for subscription table
157 CREATE INDEX idx_subscription_type ON subscription(SubscriptionType);
158
```

Output

#	Time	Action	Message
1	19:09:34	CREATE INDEX idx_subscription_type ON subscription(SubscriptionType)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0

-- create index for SongArtist table

CREATE INDEX idx_songartist_songid ON SongArtist(SongID);

CREATE INDEX idx_songartist_artistid ON SongArtist(ArtistID);

```
161 -- create index for SongArtist table
162 CREATE INDEX idx_songartist_songid ON SongArtist(SongID);
163 CREATE INDEX idx_songartist_artistid ON SongArtist(ArtistID);
164
165 -- create index for AlbumGenre table
```

Output

#	Time	Action	Message
1	20:28:47	CREATE INDEX idx_songartist_songid ON SongArtist(SongID)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
2	20:28:47	CREATE INDEX idx_songartist_artistid ON SongArtist(ArtistID)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0

-- create index for AlbumGenre table

CREATE INDEX idx_albumgenre_albumid ON AlbumGenre(AlbumID);

CREATE INDEX idx_albumgenre_genreid ON AlbumGenre(GenreID);

```
163 -- create index for AlbumGenre table
164 CREATE INDEX idx_albumgenre_albumid ON AlbumGenre(AlbumID);
165 CREATE INDEX idx_albumgenre_genreid ON AlbumGenre(GenreID);
```

Output

#	Time	Action	Message
1	19:11:16	CREATE INDEX idx_albumgenre_albumid ON AlbumGenre(AlbumID)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
2	19:11:27	CREATE INDEX idx_albumgenre_genreid ON AlbumGenre(GenreID)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0

-- create index for SongGenre table

CREATE INDEX idx_songgenre_songid ON SongGenre(SongID);

CREATE INDEX idx_songgenre_genreid ON SongGenre(GenreID);

```
166
167 -- create index for SongGenre table
168 • CREATE INDEX idx_songgenre_songid ON SongGenre(SongID);
169 • CREATE INDEX idx_songgenre_genreid ON SongGenre(GenreID);
```

Output

Action Output

#	Time	Action	Message
✓ 1	19:12:25	CREATE INDEX idx_songgenre_genreid ON SongGenre(GenreID)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
✓ 2	19:12:33	CREATE INDEX idx_songgenre_songid ON SongGenre(SongID)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0

-- create index for SongAlbum table

CREATE INDEX idx_songalbum_songid ON SongAlbum(SongID);

CREATE INDEX idx_songalbum_albumid ON SongAlbum(AlbumID);

```
171 -- create index for SongAlbum table
172 • CREATE INDEX idx_songalbum_songid ON SongAlbum(SongID);
173 • CREATE INDEX idx_songalbum_albumid ON SongAlbum(AlbumID);
174
```

Output

Action Output

#	Time	Action	Message
✓ 1	19:13:25	CREATE INDEX idx_songalbum_songid ON SongAlbum(SongID)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
✓ 2	19:13:33	CREATE INDEX idx_songalbum_albumid ON SongAlbum(AlbumID)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0

-- create index for ArtistAlbum table

CREATE INDEX idx_artistalbum_artistid ON ArtistAlbum(ArtistID);

CREATE INDEX idx_artistalbum_albumid ON ArtistAlbum(AlbumID);

```
174
175 -- create index for ArtistAlbum table
176 • CREATE INDEX idx_artistalbum_artistid ON ArtistAlbum(ArtistID);
177 • CREATE INDEX idx_artistalbum_albumid ON ArtistAlbum(AlbumID);
```

Output

Action Output

#	Time	Action	Message
✓ 1	19:14:31	CREATE INDEX idx_artistalbum_artistid ON ArtistAlbum(ArtistID)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
✓ 2	19:14:31	CREATE INDEX idx_artistalbum_albumid ON ArtistAlbum(AlbumID)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0

Create View:

1)-- create view to all songs with artist names

CREATE VIEW ViewSongsWithArtists AS

SELECT

s.SongID,

s.SongTitle,

a.ArtistName

FROM

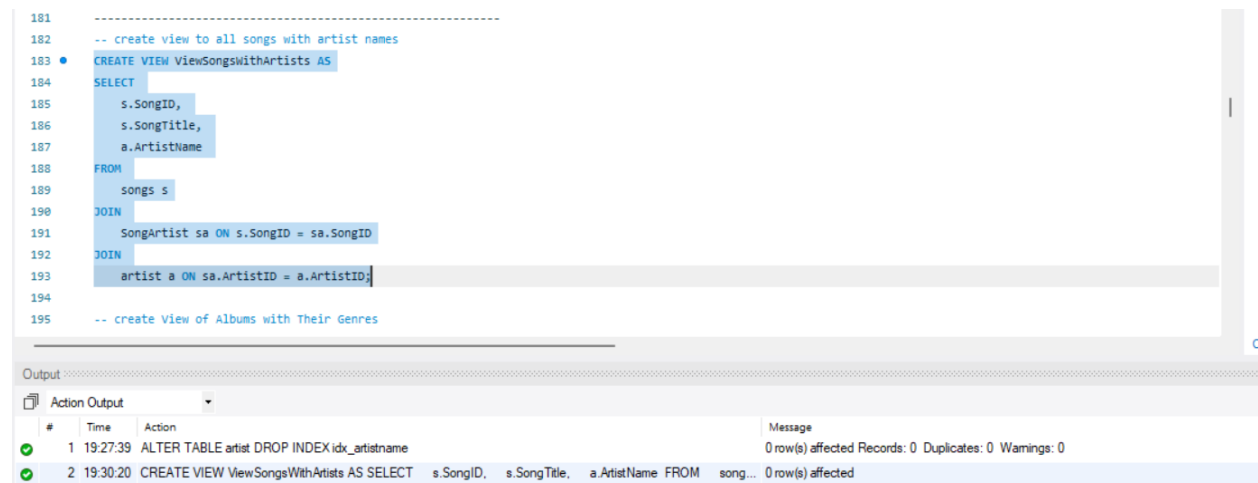
songs s

JOIN

SongArtist sa ON s.SongID = sa.SongID

JOIN

artist a ON sa.ArtistID = a.ArtistID;



The screenshot displays a SQL script editor with the following code:

```
181 -----
182 -- create view to all songs with artist names
183 • CREATE VIEW ViewSongsWithArtists AS
184 SELECT
185     s.SongID,
186     s.SongTitle,
187     a.ArtistName
188 FROM
189     songs s
190 JOIN
191     SongArtist sa ON s.SongID = sa.SongID
192 JOIN
193     artist a ON sa.ArtistID = a.ArtistID;
194
195 -- create View of Albums with Their Genres
```

Below the script editor is the 'Output' window, which shows the execution results:

#	Time	Action	Message
1	19:27:39	ALTER TABLE artist DROP INDEX idx_artistname	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0
2	19:30:20	CREATE VIEW ViewSongsWithArtists AS SELECT s.SongID, s.SongTitle, a.ArtistName FROM songs...	0 row(s) affected

```

192 JOIN
193     artist a ON sa.ArtistID = a.ArtistID;
194 • SELECT * FROM ViewSongsWithArtists; -- for extention to view
195

```

	SongID	SongTitle	ArtistName
▶	6	7 rings	The Beatles
	4	Bad Guy	Drake
	1	Cardigan	Taylor Swift
	2	Come Together	Taylor Swift
	3	God's Plan	The Beatles
	8	Halo	Billie Eilish
	9	Like a Rolling Stone	Ed Sheeran
	14	Paint It Black	Bob Dylan
	10	Power	Ed Sheeran
	11	Rolling in the Deep	Ariana Grande
	5	Shape of You	Drake
	13	Stupid Love	Beyoncé
	12	That's What I ...	Elton John
	7	Tiny Dancer	Billie Eilish
	15	Work	Kanye West

ViewSongsWithArtists22 x

Read Only

2) -- create View of Albums with Their Genres

CREATE VIEW ViewAlbumsWithGenres AS

SELECT

al.AlbumID,

al.AlbumTitle,

g.GenreName

FROM

album al

JOIN

AlbumGenre ag ON al.AlbumID = ag.AlbumID

JOIN

genre g ON ag.GenreID = g.GenreID;

```

194
195 -- create View of Albums with Their Genres
196 • CREATE VIEW ViewAlbumsWithGenres AS
197 SELECT
198     al.AlbumID,
199     al.AlbumTitle,
200     g.GenreName
201 FROM
202     album al
203 JOIN
204     AlbumGenre ag ON al.AlbumID = ag.AlbumID
205 JOIN
206     genre g ON ag.GenreID = g.GenreID;
207

```

Output

Action Output

#	Time	Action	Message
1	19:31:51	CREATE VIEW ViewAlbumsWithGenres AS SELECT al.AlbumID, al.AlbumTitle, g.GenreName FROM ...	0 row(s) affected

```

207     genre g ON ag.GenreID = g.GenreID;
208 • SELECT * FROM ViewAlbumsWithGenres;
209

```

Result Grid Filter Rows: Export: Wrap Cell Content: [T](#)

	AlbumID	AlbumTitle	GenreName
▶	11	21	Pop
	12	24K Magic	Rock
	2	Abbey Road	Rock
	15	Anti	Electronic
	13	Chromatica	Hip-Hop
	5	Divide	Electronic
	14	Exile On Main St.	Indie
	1	Folklore	Pop
	7	Goodbye Yellow Brick Road	Rock
	9	Highway 61 Revisited	Indie
	8	Lemonade	Hip-Hop
	10	My Beautiful Dark Twisted...	Electronic
	3	Scorpion	Hip-Hop
	6	Thank U, Next	Pop
	4	When We All Fall Asleep, ...	Indie

ViewAlbumsWithGenres 23 x

Read Only

3) -- create View of User Subscriptions

CREATE VIEW ViewUserSubscriptions AS

SELECT

u.UserID,

u.Username,

u.Email,

s.SubscriptionType,

s.Price

FROM

user u

JOIN

subscription s ON u.SubscriptionID = s.SubscriptionID;

```
208 -- create View of User Subscriptions
209 • CREATE VIEW ViewUserSubscriptions AS
210 SELECT
211     u.UserID,
212     u.Username,
213     u.Email,
214     s.SubscriptionType,
215     s.Price
216 FROM
217     user u
218 JOIN
219     subscription s ON u.SubscriptionID = s.SubscriptionID;
220
221 -- create View of Playlists with Song Count
222 • CREATE VIEW ViewPlaylistSongCount AS
```

Output

Action Output

#	Time	Action	Message
✓ 1	19:32:57	CREATE VIEW ViewUserSubscriptions AS SELECT u.UserID, u.Username, u.Email, s.SubscriptionT...	0 row(s) affected

```

221      subscription s ON u.SubscriptionID = s.SubscriptionID;
222 • SELECT * FROM ViewUserSubscriptions;
223

```

	UserID	Username	Email	SubscriptionType	Price
▶	9	CaseyB	caseyb@example.com	Basic	5.99
	15	TaylorM	taylorm@example.com	Basic	5.99
	2	Jamie_F	jamief@example.com	Standard	9.99
	1	AlexSmith	alexsmith@example.com	Premium	14.99
	7	TaylorR	taylorr@example.com	Premium	14.99
	10	DrewA	drewa@example.com	Family	19.99
	11	JordanM	jordanm@example.com	Family	19.99
	3	ChrisP	chrisp@example.com	Student	4.99
	8	MorganC	morganc@example.com	Duo	12.99
	14	AlexisT	alexist@example.com	Duo	12.99
	6	PatW	patw@example.com	Annual Basic	59.99
	5	SamT	samt@example.com	Annual Standard	99.99
	13	SydneyJ	sydneyj@example.com	Annual Standard	99.99
	12	CharlieG	charleg@example.com	Annual Premium	149.99
	4	JordanK	jordank@example.com	Lifetime	299.99

ViewUserSubscriptions 24 x Read Only

4) -- create View of Playlists with Song Count

CREATE VIEW ViewPlaylistSongCount AS

SELECT

p.PlaylistID,

p.PlaylistTitle,

COUNT(sp.SongID) AS NumberOfSongs

FROM

playlist p

LEFT JOIN

SongPlaylist sp ON p.PlaylistID = sp.PlaylistID

GROUP BY

p.PlaylistID,

p.PlaylistTitle;

```

221 -- create View of Playlists with Song Count
222 • CREATE VIEW ViewPlaylistSongCount AS
223 SELECT
224     p.PlaylistID,
225     p.PlaylistTitle,
226     COUNT(sp.SongID) AS NumberOfSongs
227 FROM
228     playlist p
229 LEFT JOIN
230     SongPlaylist sp ON p.PlaylistID = sp.PlaylistID
231 GROUP BY
232     p.PlaylistID,
233     p.PlaylistTitle;
234

```

Output

Action Output

#	Time	Action	Message
1	19:34:12	CREATE VIEW ViewPlaylistSongCount AS SELECT p.PlaylistID, p.PlaylistTitle, COUNT(sp.SongID) AS ...	0 row(s) affected

```

236     p.PlaylistTitle;
237 • SELECT * FROM ViewPlaylistSongCount;

```

Result Grid Filter Rows: Export: Wrap Cell Content: [I](#)

PlaylistID	PlaylistTitle	NumberOfSongs
12	Blues Nights	0
2	Classic Rock	3
9	Classical Mornings	0
6	Country Roads	0
5	Electronic Essentials	3
11	Folklore and More	0
3	Hip-Hop Vibes	3
4	Indie Mix	3
8	Jazz Lounge	0
10	Metal Mayhem	0
15	Punk Rock	0
7	R&B Favorites	0
13	Reggae Sunsplash	0
14	Soulful Voices	0
1	Workout Hits	3

ViewPlaylistSongCount25 x

Read Only

Temporary Table:

1)-- create a temporary Table for Detailed Song Information

CREATE TEMPORARY TABLE TempSongDetails AS

SELECT

s.SongID,

s.SongTitle,

s.Duration,

s.ReleaseYear,

a.ArtistName,

al.AlbumTitle,

g.GenreName

FROM

songs s

JOIN

SongArtist sa ON s.SongID = sa.SongID

JOIN

artist a ON sa.ArtistID = a.ArtistID

JOIN

SongAlbum sal ON s.SongID = sal.SongID

JOIN

album al ON sal.AlbumID = al.AlbumID

JOIN

SongGenre sg ON s.SongID = sg.SongID

JOIN

genre g ON sg.GenreID = g.GenreID;

The screenshot shows a SQL IDE window with a query editor and an output pane. The query editor contains the following SQL code:

```
241 CREATE TEMPORARY TABLE TempSongDetails AS
242 SELECT
243     s.SongID,
244     s.SongTitle,
245     s.Duration,
246     s.ReleaseYear,
247     a.ArtistName,
248     al.AlbumTitle,
249     g.GenreName
250 FROM
251     songs s
252 JOIN
253     SongArtist sa ON s.SongID = sa.SongID
254 JOIN
255     artist a ON sa.ArtistID = a.ArtistID
256 JOIN
257     SongAlbum sal ON s.SongID = sal.SongID
258 JOIN
259     album al ON sal.AlbumID = al.AlbumID
260 JOIN
261     SongGenre sg ON s.SongID = sg.SongID
262 JOIN
263     genre g ON sg.GenreID = g.GenreID;
```

The output pane shows the execution results:

#	Time	Action	Message
1	19:41:36	CREATE TEMPORARY TABLE TempSongDetails AS SELECT s.SongID, s.SongTitle, s.Duration, s.R...	15 row(s) affected Records: 15 Duplicates: 0 Warnings: 0

2)-- create a temporary Table for User Playlist Overview

CREATE TEMPORARY TABLE TempUserPlaylistOverview AS

SELECT

p.UserID,

p.PlaylistID,

p.PlaylistTitle,

COUNT(sp.SongID) AS NumberOfSongs,

SUM(s.Duration) AS TotalDuration

FROM

playlist p

JOIN

SongPlaylist sp ON p.PlaylistID = sp.PlaylistID

JOIN

songs s ON sp.SongID = s.SongID

GROUP BY

p.PlaylistID;

```
266 -- create a temporary Table for User Playlist Overview
267 CREATE TEMPORARY TABLE TempUserPlaylistOverview AS
268 SELECT
269     p.UserID,
270     p.PlaylistID,
271     p.PlaylistTitle,
272     COUNT(sp.SongID) AS NumberOfSongs,
273     SUM(TIME_TO_SEC(s.Duration)) AS TotalDurationSeconds -- Convert duration to seconds before summing
274 FROM
275     playlist p
276 JOIN
277     SongPlaylist sp ON p.PlaylistID = sp.PlaylistID
278 JOIN
279     songs s ON sp.SongID = s.SongID
280 GROUP BY
281     p.PlaylistID;
```

Output:

#	Time	Action	Message
1	19:48:50	CREATE TEMPORARY TABLE TempUserPlaylistOverview AS SELECT p.UserID, p.PlaylistID, p.PlaylistT...	5 row(s) affected Records: 5 Duplicates: 0 Warnings: 0

3)-- create a temporary Table for Subscription Analysis

```
CREATE TEMPORARY TABLE TempSubscriptionAnalysis AS
```

```
SELECT
```

```
    s.SubscriptionType,
```

```
    COUNT(u.UserID) AS NumberOfUsers,
```

```
    AVG(
```

```
        CASE
```

```
            WHEN u.Subscription_Duration LIKE '%month' THEN  
            CONVERT(SUBSTRING_INDEX(u.Subscription_Duration, ' ', 1), SIGNED)
```

```
        END
```

```
    ) AS AverageDurationMonths
```

```
FROM
```

```
    subscription s
```

```
JOIN
```

```
    user u ON s.SubscriptionID = u.SubscriptionID
```

```
GROUP BY
```

```
    s.SubscriptionType;
```

The screenshot displays a SQL IDE interface. The main editor shows a SQL script with line numbers 284 through 299. The script creates a temporary table named 'TempSubscriptionAnalysis' and populates it with data from a 'subscription' table joined with a 'user' table. The query groups the results by 'SubscriptionType' and calculates the average duration in months for each type, based on subscriptions that include the word 'month' in their duration. The 'Output' pane at the bottom shows the 'Action Output' for the execution, indicating that 10 rows were affected and no warnings occurred.

```
284 -- create a temporary Table for Subscription Analysis
285 CREATE TEMPORARY TABLE TempSubscriptionAnalysis AS
286 SELECT
287     s.SubscriptionType,
288     COUNT(u.UserID) AS NumberOfUsers,
289     AVG(
290         CASE
291             WHEN u.Subscription_Duration LIKE '%month' THEN CONVERT(SUBSTRING_INDEX(u.Subscription_Duration, ' ', 1), SIGNED)
292         END
293     ) AS AverageDurationMonths
294 FROM
295     subscription s
296 JOIN
297     user u ON s.SubscriptionID = u.SubscriptionID
298 GROUP BY
299     s.SubscriptionType;
```

Output: Action Output

#	Time	Action	Message
1	19:52:56	CREATE TEMPORARY TABLE TempSubscriptionAnalysis AS SELECT s.SubscriptionType, COUNT(u.UserID) AS NumberOfUsers, AVG(CASE WHEN u.Subscription_Duration LIKE '%month' THEN CONVERT(SUBSTRING_INDEX(u.Subscription_Duration, ' ', 1), SIGNED) END) AS AverageDurationMonths FROM subscription s JOIN user u ON s.SubscriptionID = u.SubscriptionID GROUP BY s.SubscriptionType;	10 row(s) affected Records: 10 Duplicates: 0 Warnings: 0

Trigger:

1)-- Trigger to Log When a New Artist is Added

DELIMITER \$\$

CREATE TRIGGER AfterArtistInsert

AFTER INSERT ON artist

FOR EACH ROW

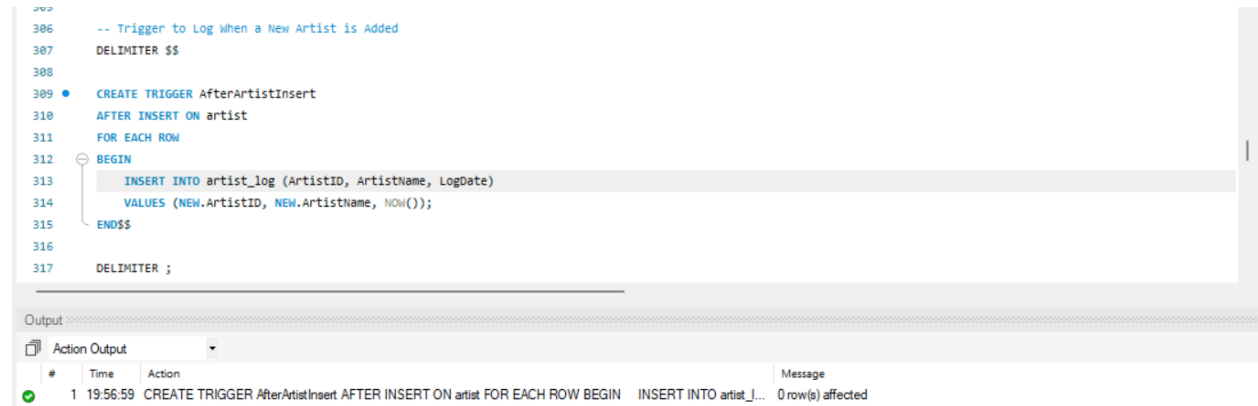
BEGIN

INSERT INTO artist_log (ArtistID, ArtistName, LogDate)

VALUES (NEW.ArtistID, NEW.ArtistName, NOW());

END\$\$

DELIMITER ;



The screenshot displays a SQL script in a text editor and its execution results in an output window. The script defines a trigger named 'AfterArtistInsert' that fires after an insert on the 'artist' table, logging the new artist's details into the 'artist_log' table. The output window shows a successful execution of the script at 19:56:59, with 0 rows affected.

```
306 -- Trigger to Log when a New Artist is Added
307 DELIMITER $$
308
309 CREATE TRIGGER AfterArtistInsert
310 AFTER INSERT ON artist
311 FOR EACH ROW
312 BEGIN
313     INSERT INTO artist_log (ArtistID, ArtistName, LogDate)
314     VALUES (NEW.ArtistID, NEW.ArtistName, NOW());
315 END$$
316
317 DELIMITER ;
```

Output

#	Time	Action	Message
1	19:56:59	CREATE TRIGGER AfterArtistInsert AFTER INSERT ON artist FOR EACH ROW BEGIN INSERT INTO artist_log (ArtistID, ArtistName, LogDate) VALUES (NEW.ArtistID, NEW.ArtistName, NOW());	0 row(s) affected

2)-- Trigger to Update User Subscription Duration

DELIMITER \$\$

CREATE TRIGGER AfterUserSubscriptionUpdate

AFTER UPDATE ON user

FOR EACH ROW

BEGIN

IF OLD.SubscriptionID != NEW.SubscriptionID THEN

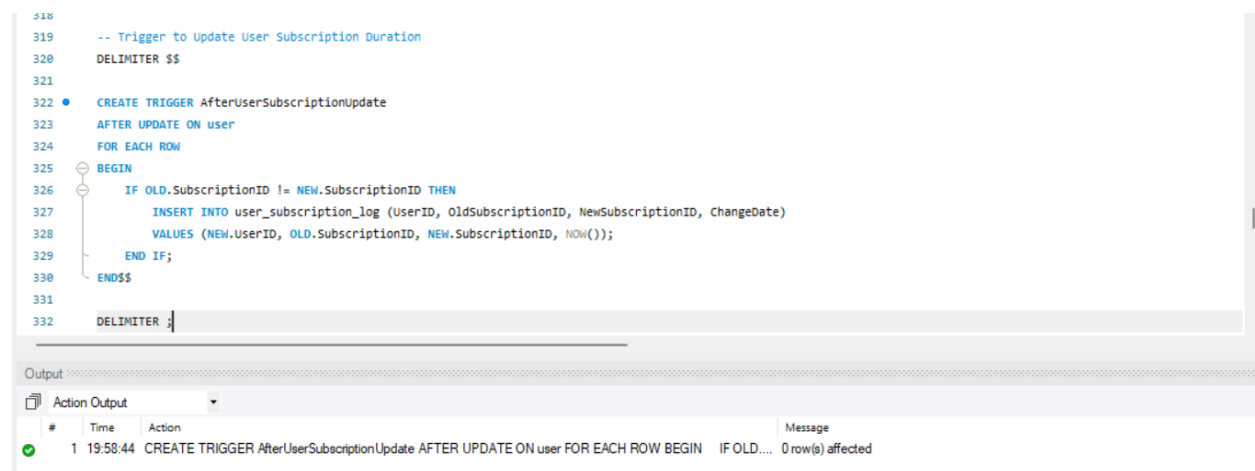
INSERT INTO user_subscription_log (UserID, OldSubscriptionID, NewSubscriptionID, ChangeDate)

VALUES (NEW.UserID, OLD.SubscriptionID, NEW.SubscriptionID, NOW());

END IF;

END\$\$

DELIMITER ;



The screenshot shows a SQL script in a text editor with line numbers 318 to 332. The script defines a trigger named 'AfterUserSubscriptionUpdate' that fires after an update on the 'user' table. The trigger body checks if the 'SubscriptionID' has changed and, if so, inserts a record into the 'user_subscription_log' table. The script uses '\$\$' as a delimiter. Below the editor, the 'Output' window is visible, showing a table with columns '#', 'Time', 'Action', and 'Message'. A single row indicates that the trigger was successfully created at 19:58:44.

```
318
319 -- Trigger to Update User Subscription Duration
320 DELIMITER $$
321
322 • CREATE TRIGGER AfterUserSubscriptionUpdate
323 AFTER UPDATE ON user
324 FOR EACH ROW
325 BEGIN
326     IF OLD.SubscriptionID != NEW.SubscriptionID THEN
327         INSERT INTO user_subscription_log (UserID, OldSubscriptionID, NewSubscriptionID, ChangeDate)
328             VALUES (NEW.UserID, OLD.SubscriptionID, NEW.SubscriptionID, NOW());
329     END IF;
330 END$$
331
332 DELIMITER ;
```

Output

#	Time	Action	Message
1	19:58:44	CREATE TRIGGER AfterUserSubscriptionUpdate AFTER UPDATE ON user FOR EACH ROW BEGIN IF OLD....	0 row(s) affected

3)-- Trigger for Deleting Songs from Playlists When a Song is Deleted

DELIMITER \$\$

CREATE TRIGGER BeforeSongDelete

BEFORE DELETE ON songs

FOR EACH ROW

BEGIN

DELETE FROM SongPlaylist WHERE SongID = OLD.SongID;

END\$\$

DELIMITER ;

```
333
334 -- Trigger for Deleting Songs from Playlists When a Song is Deleted
335 DELIMITER $$
336
337 • CREATE TRIGGER BeforeSongDelete
338 BEFORE DELETE ON songs
339 FOR EACH ROW
340 BEGIN
341     DELETE FROM SongPlaylist WHERE SongID = OLD.SongID;
342 END$$
343
344 DELIMITER ;
```

Output

#	Time	Action	Message
1	19:59:31	CREATE TRIGGER BeforeSongDelete BEFORE DELETE ON songs FOR EACH ROW BEGIN DELETE FRO...	0 row(s) affected

STORED PROCEDURE:

1)-- Adding a New Artist

DELIMITER \$\$

CREATE PROCEDURE AddNewArtist(IN artistName VARCHAR(255))

BEGIN

INSERT INTO artist (ArtistName) VALUES (artistName);

END\$\$

DELIMITER ;



The screenshot shows a SQL IDE with a script editor and an output window. The script editor contains the following SQL code:

```
346 -- Adding a New Artist
347
348 DELIMITER $$
349 CREATE PROCEDURE AddNewArtist(IN artistName VARCHAR(255))
350 BEGIN
351     INSERT INTO artist (ArtistName) VALUES (artistName);
352 END$$
353 DELIMITER ;
```

The output window shows the results of the execution:

#	Time	Action	Message
1	19:59:31	CREATE TRIGGER BeforeSongDelete BEFORE DELETE ON songs FOR EACH ROW BEGIN DELETE FROM ...	0 row(s) affected
2	20:01:19	CREATE PROCEDURE AddNewArtist(IN artistName VARCHAR(255)) BEGIN INSERT INTO artist (ArtistName)...	0 row(s) affected

2) -- Adding a New Album for an Art

DELIMITER \$\$

CREATE PROCEDURE AddNewAlbum(IN albumTitle VARCHAR(255), IN releaseYear INT,
IN artistID INT)

BEGIN

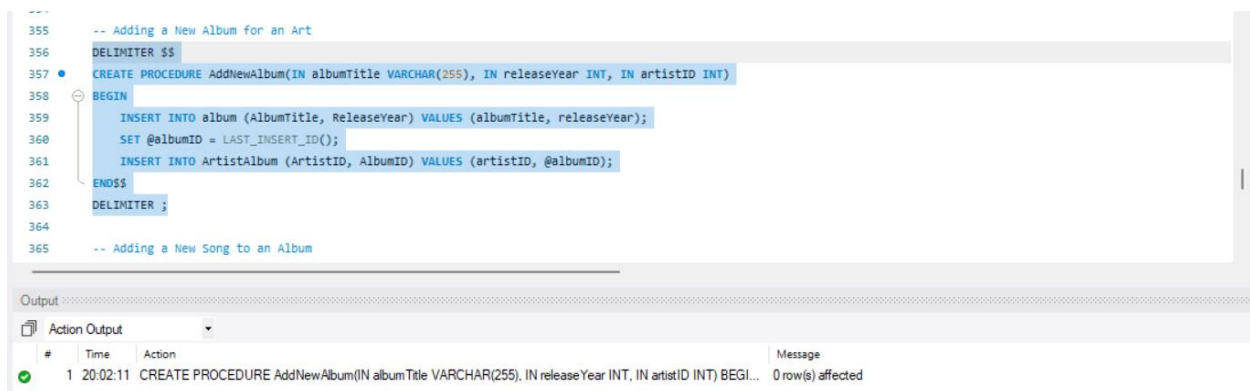
INSERT INTO album (AlbumTitle, ReleaseYear) VALUES (albumTitle, releaseYear);

SET @albumID = LAST_INSERT_ID();

INSERT INTO ArtistAlbum (ArtistID, AlbumID) VALUES (artistID, @albumID);

END\$\$

DELIMITER ;



The screenshot shows a SQL IDE with a script editor and an output window. The script editor contains the following SQL code:

```
-- Adding a New Album for an Art
355 DELIMITER $$
356 CREATE PROCEDURE AddNewAlbum(IN albumTitle VARCHAR(255), IN releaseYear INT, IN artistID INT)
357 BEGIN
358     INSERT INTO album (AlbumTitle, ReleaseYear) VALUES (albumTitle, releaseYear);
359     SET @albumID = LAST_INSERT_ID();
360     INSERT INTO ArtistAlbum (ArtistID, AlbumID) VALUES (artistID, @albumID);
361 END$$
362 DELIMITER ;
363
364
365 -- Adding a New Song to an Album
```

The output window shows the execution of the stored procedure:

#	Time	Action	Message
1	20:02:11	CREATE PROCEDURE AddNewAlbum(IN albumTitle VARCHAR(255), IN releaseYear INT, IN artistID INT) BEGI...	0 row(s) affected

3)-- Adding a New Song to an Album

DELIMITER \$\$

CREATE PROCEDURE AddNewSong(IN songTitle VARCHAR(255), IN duration INT, IN releaseYear INT, IN albumID INT)

BEGIN

INSERT INTO songs (SongTitle, Duration, ReleaseYear) VALUES (songTitle, duration, releaseYear);

SET @songID = LAST_INSERT_ID();

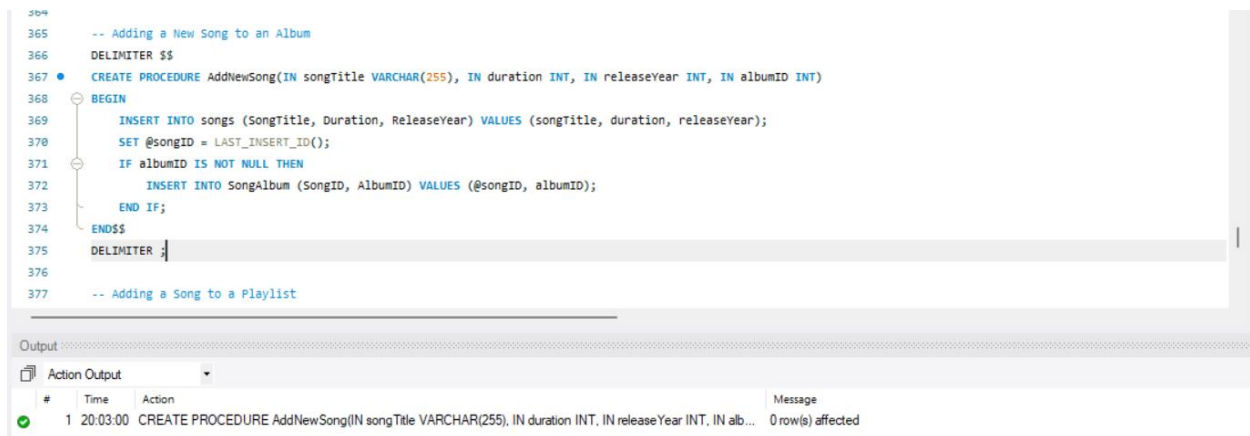
IF albumID IS NOT NULL THEN

INSERT INTO SongAlbum (SongID, AlbumID) VALUES (@songID, albumID);

END IF;

END\$\$

DELIMITER ;



The screenshot shows a SQL IDE with a script editor and an output window. The script editor contains the following SQL code:

```
364 -- Adding a New Song to an Album
365
366 DELIMITER $$
367 CREATE PROCEDURE AddNewSong(IN songTitle VARCHAR(255), IN duration INT, IN releaseYear INT, IN albumID INT)
368 BEGIN
369     INSERT INTO songs (SongTitle, Duration, ReleaseYear) VALUES (songTitle, duration, releaseYear);
370     SET @songID = LAST_INSERT_ID();
371     IF albumID IS NOT NULL THEN
372         INSERT INTO SongAlbum (SongID, AlbumID) VALUES (@songID, albumID);
373     END IF;
374 END$$
375 DELIMITER ;
376
377 -- Adding a Song to a Playlist
```

The output window shows the following message:

#	Time	Action	Message
1	20:03:00	CREATE PROCEDURE AddNewSong(IN songTitle VARCHAR(255), IN duration INT, IN releaseYear INT, IN alb...	0 row(s) affected

4)-- Adding a Song to a Playlist

DELIMITER \$\$

CREATE PROCEDURE AddSongToPlaylist(IN playlistID INT, IN songID INT)

BEGIN

INSERT INTO SongPlaylist (PlaylistID, SongID) VALUES (playlistID, songID);

END\$\$

DELIMITER ;

```
377  -- Adding a Song to a Playlist
378  DELIMITER $$
379  • CREATE PROCEDURE AddSongToPlaylist(IN playlistID INT, IN songID INT)
380  BEGIN
381      INSERT INTO SongPlaylist (PlaylistID, SongID) VALUES (playlistID, songID);
382  END$$
383  DELIMITER ;
```

Output

#	Time	Action	Message
1	20:03:50	CREATE PROCEDURE AddSongToPlaylist(IN playlistID INT, IN songID INT) BEGIN INSERT INTO SongPlaylist...	0 row(s) affected

Function:

1)-- Function to get the name of an artist based on the artist ID

DELIMITER \$\$

CREATE FUNCTION GetArtistName(artistID INT) RETURNS VARCHAR(255)

READS SQL DATA

BEGIN

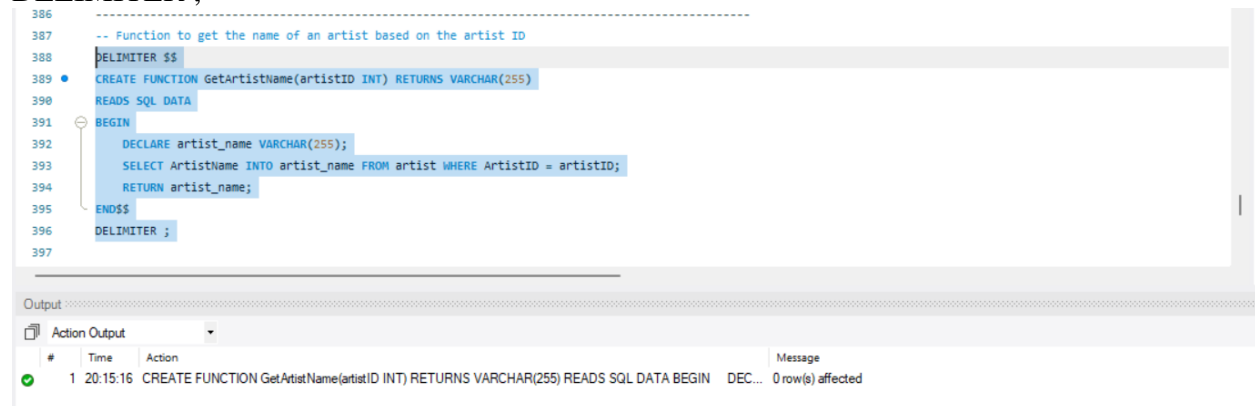
DECLARE artist_name VARCHAR(255);

SELECT ArtistName INTO artist_name FROM artist WHERE ArtistID = artistID;

RETURN artist_name;

END\$\$

DELIMITER ;



The screenshot shows a SQL IDE with a script editor and an output window. The script editor contains the following SQL code:

```
386 -----
387 -- Function to get the name of an artist based on the artist ID
388 DELIMITER $$
389 CREATE FUNCTION GetArtistName(artistID INT) RETURNS VARCHAR(255)
390 READS SQL DATA
391 BEGIN
392     DECLARE artist_name VARCHAR(255);
393     SELECT ArtistName INTO artist_name FROM artist WHERE ArtistID = artistID;
394     RETURN artist_name;
395 END$$
396 DELIMITER ;
397
```

The output window shows the execution results:

#	Time	Action	Message
1	20:15:16	CREATE FUNCTION GetArtistName(artistID INT) RETURNS VARCHAR(255) READS SQL DATA BEGIN	DEC... 0 row(s) affected

2)-- Function to Get Artist Name by Song ID

DELIMITER \$\$

CREATE FUNCTION GetArtistNameBySongID(songID INT) RETURNS VARCHAR(255)

READS SQL DATA

BEGIN

DECLARE artistName VARCHAR(255);

SELECT a.ArtistName INTO artistName

FROM artist a

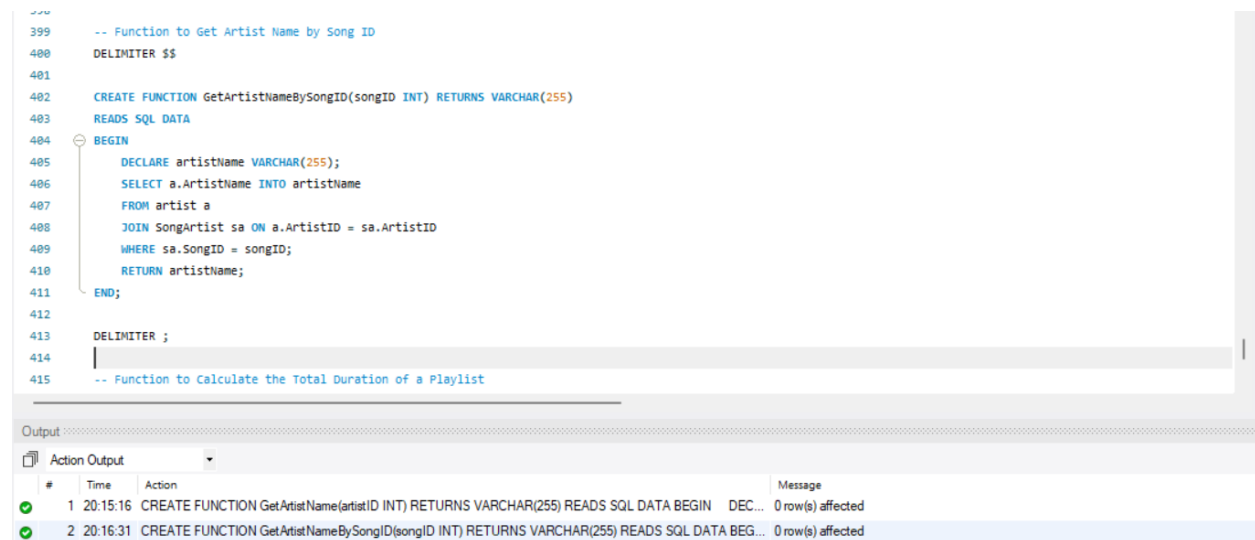
JOIN SongArtist sa ON a.ArtistID = sa.ArtistID

WHERE sa.SongID = songID;

RETURN artistName;

END;

DELIMITER ;



The screenshot shows a SQL IDE with a script editor and an output window. The script editor contains the following SQL code:

```
399 -- Function to Get Artist Name by Song ID
400 DELIMITER $$
401
402 CREATE FUNCTION GetArtistNameBySongID(songID INT) RETURNS VARCHAR(255)
403 READS SQL DATA
404 BEGIN
405     DECLARE artistName VARCHAR(255);
406     SELECT a.ArtistName INTO artistName
407     FROM artist a
408     JOIN SongArtist sa ON a.ArtistID = sa.ArtistID
409     WHERE sa.SongID = songID;
410     RETURN artistName;
411 END;
412
413 DELIMITER ;
414
415 -- Function to Calculate the Total Duration of a Playlist
```

The output window shows the execution results of the SQL script. It has a table with columns: #, Time, Action, and Message. The first two rows are highlighted with green checkmarks, indicating successful execution.

#	Time	Action	Message
1	20:15:16	CREATE FUNCTION GetArtistName(artistID INT) RETURNS VARCHAR(255) READS SQL DATA BEGIN DEC...	0 row(s) affected
2	20:16:31	CREATE FUNCTION GetArtistNameBySongID(songID INT) RETURNS VARCHAR(255) READS SQL DATA BEG...	0 row(s) affected

3)-- Function to Calculate the Total Duration of a Playlist

DELIMITER \$\$

CREATE FUNCTION GetPlaylistDuration(playlistID INT) RETURNS INT

READS SQL DATA

BEGIN

DECLARE totalDuration INT DEFAULT 0;

SELECT SUM(TIME_TO_SEC(s.Duration)) INTO totalDuration

FROM songs s

JOIN SongPlaylist sp ON s.SongID = sp.SongID

WHERE sp.PlaylistID = playlistID;

RETURN totalDuration;

END\$\$

DELIMITER ;

```
414
415 -- Function to Calculate the Total Duration of a Playlist
416 DELIMITER $$
417
418 • CREATE FUNCTION GetPlaylistDuration(playlistID INT) RETURNS INT
419 READS SQL DATA
420 BEGIN
421     DECLARE totalDuration INT DEFAULT 0;
422     SELECT SUM(TIME_TO_SEC(s.Duration)) INTO totalDuration
423     FROM songs s
424     JOIN SongPlaylist sp ON s.SongID = sp.SongID
425     WHERE sp.PlaylistID = playlistID;
426     RETURN totalDuration;
427 END$$
428
429 DELIMITER ;
430 • DROP FUNCTION IF EXISTS GetPlaylistDuration;
```

Output

#	Time	Action	Message
1	20:20:25	CREATE FUNCTION GetPlaylistDuration(playlistID INT) RETURNS INT READS SQL DATA BEGIN DECLARE...	0 row(s) affected