CSE560 – Online Music Store Database System

Team - 04

Sathwick Kiran M S
UBID – sathwick

Kundan Anil Satkar
UBID - kundanan

I. Introduction

The music business has undergone a significant change in the digital age in the past two decades. As we are aware, online music retailers are now critical avenues for music distribution sales. To make decisions that facilitate growth, one must better understand sales behavior, customers, and the success of the business. This project will use SQL to analyze a data set of an online music store in a logical and systematic way. The primary purpose of the project is to provide insights that respond to critical business questions on sales, customer behavior, and how revenue is derived. Finding valuable insights is the full intent of this project. Ultimately, the project will hopefully make data-driven recommendations on improving customer experience satisfaction, retail process efficiency, and business profit.

Synopsis of Milestone 1

Our first milestone in the Online Music Store Database project involved building a foundation for our capabilities for managing and analysing data. We were designing and implementing a structured relational database management system, with a greater capacity for managing sites with larger datasets about music inventory, sales and user activity data. After determining the business problem i.e., the methods we were used to manage datasets like Excel were unacceptable in managing our growing online music store, we were able to justify the need for a database as opposed to traditional data management demonstrating key advantages with respect to data accuracy, scalability, analysis and security. We then identified the major user roles - customers, store managers and administrators - and documented their corresponding use cases to ensure the system would meet real-life requirements. Next we generated an E/R diagram and built eleven interrelated tables: Customer, Employee, Invoice, Invoice_Line, Track, Album, Artist, Media_Type, Genre, Playlist, and Playlist_Track. We built each table with primary keys and foreign keys to protect the integrity of the data and allow complex queries. In addition, we completed and tested some of our initial SQL queries in the following ways: querying the database for all artists, counting the total tracks by genre, rankings of total spending for all customers, determining the most 'productive' artist, and filtering playlists to only a specified number of tracks. These queries demonstrated that we could extract data from the database in a meaningful way. Overall, this phase was a good base for our milestone two implementation.

Problem Statement

If a new online music business fails to implement good management systems, tracking sales and customer information, and managing music inventories can seem unmanageable, become susceptible to mistakes, and ultimately become impossible to grow. An organized system for diligence is necessary to ensure that you can track purchases, analyze customer preferences, and accurately maintain your sales records.

A structured management database system organizes these processes by managing large volumes of data, enabling complex analysis of the data, and ensuring integrity by establishing logical connections. The technology will also help a better business decision assistance, improve sales-process effectiveness, and enhance the customer experience by allowing and advancing automation, scalability, and precision.

II. Why there is a need for a database instead of Excel?

Efficient Data Management: A database efficiently organizes and deals with large quantities of information, such as customers, their musical preferences, and sales as the shop expands without limitation.

Data Accuracy and Consistency: A database assures that information is correct, organized, and consistent by enforcing rules such as primary and foreign keys.

Powerful Data Analysis: A database generates better analytics by utilizing SQL capabilities including facilities of SELECT, JOIN, GROUP BY, aggregated functions to facilitate decisions and enhance administration.

Optimized Storage and Maintenance: Data normalization removes redundancy and is more efficient for storage and updating by not needing to repeat the same data multiple times.

Enhanced Security and Recovery: Backup and recovery mechanisms both protect against data warehouse data loss and ensure that you can restore it quickly when you delete the wrong row or the system crashes.

III. Target Users

The database system supports three main types of users:

Customers:

- Browse music by category, artist, or genre.
- Place orders, check status, and manage their profiles and collections.

Store Managers:

- Add or update music albums and songs.
- Review customer orders and maintain inventory and product details.

Administrators:

- Monitor database health and ensure consistent, accurate data.
- Fix duplicated records and manage user accounts securely.

Real-Life Scenarios:

- **Business Owner:** Siddharth, the owner of an online music store (like iTunes), tracks sales trends, customer loyalty, and revenue patterns.
- **Data Analyst:** Sathwick analyzes sales data to find best-selling genres, artists, and peak sales periods, helping optimize promotions and stock.
- **Administrator:** Kundan manages database performance, optimizes slow queries, and ensures fast, reliable operations by maintaining clean, accurate data.

IV. Sample Database Design

The sample music store database was able to be created. This allows for the testing and debugging process to happen prior to working with a much larger data set. Below are examples of SQL statements demonstrating using all aspects of database management (creating, loading, altering, updating, and destroying).

1. Creating Sample table:

```
CREATE TABLE Track (
TrackID INT PRIMARY KEY,
Name VARCHAR (255),
AlbumID INT,
MediaTypeID INT,
GenreID INT,
Milliseconds INT,
Bytes INT
);
```

2. Loading into the sample table:

```
INSERT INTO Track (TrackID, Name, AlbumID, MediaTypeID,
GenreID, Milliseconds, Bytes)
VALUES (1, 'Sample Song', 10, 1, 5, 230000, 5120000);
```

3. Altering the Table:

```
ALTER TABLE Track ADD COLUMN UnitPrice NUMERIC(5,2);
```

4. **Updating the Table:**

```
UPDATE Track SET UnitPrice = 0.99
WHERE TrackID = 1;
```

5. Deleting the table:

```
DROP TABLE IF EXISTS Track;
```

V. Tables Description

Table Name	**Description**
Artist	Stores artist information including artist name and unique ID.
Album	Contains album details linked to artists.
Genre	Stores genre names and their identifiers.
Media_Type	Lists available media formats (e.g., MP3, AAC).
Track	Contains music track information including name, genre, album, and duration.
Playlist	Stores playlist details such as playlist name and ID.
Playlist_Track	Maps tracks to playlists using many-to-many relationships.
Customer	Stores customer details including name, contact info, and support rep ID.
Employee	Maintains employee information including job title, manager, and contact info.
Invoice	Records sales transactions including billing address and customer link.
Invoice_Line	Stores individual item sales from invoices including track, quantity, and price.

Fig 1 – Base Table Description

VI. Foreign Key Action

Table Name	Foreign Key	Referenced Table	On Delete Action
Album	artist_id	Artist	CASCADE
Track	album_id	Album	CASCADE
Track	genre_id	Genre	SET NULL
Track	media_type_id	Media_Type	SET NULL
Playlist_Track	playlist_id	Playlist	CASCADE
Playlist_Track	track_id	Track	CASCADE
Invoice	customer_id	Customer	CASCADE
Invoice	support_rep_id	Employee	SET NULL
Invoice_Line	invoice_id	Invoice	CASCADE
Invoice_Line	track_id	Track	CASCADE
Customer	support_rep_id	Employee	SET NULL

Fig 2 – Foreign Keys Action

VII. Normalization Process & BCNF Validation

This tables explains the relation and their attributes before BCNF validation is done.
Now, when checking which tables violated BCNF and which tables satisfied this, we found that five tables that is.,

- Invoice
- Invoice_line
- Customer
- Employee
- Track

These tables were not in BCNF form, and we had to decompose this to satisfy this condition.
Decomposition:

1. **Invoice-**

Initial Schema –

```
CREATE TABLE public.invoice
(invoice_id character varying(30) NOT NULL,
```

customer_id character varying(30),
invoice_date timestamp without time zone,
billing_address character varying(120),
billing_city character varying(30),
billing_state character varying(30),
billing_country character varying(30),
billing_postal character varying(30),
total double precision,
PRIMARY KEY (invoice_id));

Violation –

- invoice_line_id → invoice_id, track_id, unit_price, quantity,
- track_id → unit_price which **violates BCNF** as track_id is not a superkey.

Decomposed -

CREATE TABLE customer_billing (
customer_id VARCHAR(30) PRIMARY KEY REFERENCES customer(customer_id),
billing_address VARCHAR(120), billing_city VARCHAR(30), billing_state VARCHAR(30), billing_country VARCHAR(30), billing_postal VARCHAR(30));
INSERT INTO customer_billing
SELECT DISTINCT customer_id, billing_address, billing_city, billing_state, billing_country, billing_postal
FROM invoice WHERE customer_id IS NOT NULL;
ALTER TABLE invoice DROP COLUMN billing_address,
DROP COLUMN billing_city,
DROP COLUMN billing_state, DROP COLUMN billing_country, DROP COLUMN billing_postal;

2. Track-

Initial Schema-

CREATE TABLE public.track
(track_id character varying(30) NOT NULL,
name character varying(250),
album_id character varying(30),
media_type_id character varying(30),
genre_id character varying(30),
composer character varying(250),
milliseconds bigint,
bytes integer,
unit_price numeric,
PRIMARY KEY (track_id))

Violation-

- invoice_line_id → invoice_id, track_id, unit_price, quantity
- track_id → unit_price which **violates BCNF** as track_id is not a superkey.

Decomposed-

CREATE TABLE media_type_pricing (
media_type_id VARCHAR(30), unit_price NUMERIC,
PRIMARY KEY(media_type_id, unit_price));
INSERT INTO media_type_pricing
SELECT DISTINCT media_type_id, unit_price FROM track;
ALTER TABLE track DROP COLUMN unit_price;

3. Invoice_Line-

Initial Schema-

CREATE TABLE public.invoice_line
(invoice_line_id character varying(30) NOT NULL,
invoice_id character varying(30),
track_id character varying(30),
unit_price numeric,
quantity integer,
PRIMARY KEY (invoice_line_id))

Violation-

- invoice_line_id → invoice_id, track_id, unit_price, quantity
- track_id → unit_price *which* **violates BCNF** as track_id is not a superkey.

Decompostion-

CREATE TABLE track_price (
track_id VARCHAR(30) PRIMARY KEY REFERENCES track(track_id),
unit_price NUMERIC);
INSERT INTO track_price
SELECT DISTINCT track_id, unit_price FROM invoice_line;
ALTER TABLE invoice_line DROP COLUMN unit_price;

4. Employee-

Initial Schema-

CREATE TABLE public.employee
(
employee_id character varying(30) NOT NULL,
last_name character(50),
first_name character(50),
title character varying(250),
reports_to character varying(30),
levels character varying(10),
birth_date timestamp without time zone,
hire_date timestamp without time zone,
address character varying(120),
city character varying(50),
state character varying(50),
country character varying(30),
postal_code character varying(30),
phone character varying(30),
fax character varying(30),
email character varying(30),
PRIMARY KEY (employee_id))

Violation-

- track_id → all other attributes
- media_type_id → unit_price, genre_id → unit_price which **violates BCNF**.

Decompositon-

```
CREATE TABLE location (postal_code VARCHAR(30)
PRIMARY KEY, city VARCHAR(50), state VARCHAR(30),
country VARCHAR(30));
INSERT INTO location
SELECT DISTINCT postal_code, city, state, country FROM
employee WHERE postal_code IS NOT NULL;
ALTER TABLE employee DROP COLUMN city, DROP
COLUMN state, DROP COLUMN country;
```

5. **Customer-**

Initial Schema-

```
CREATE TABLE public.customer
(
   customer_id character varying(30) NOT NULL,
   first_name character(30),
   last_name character(30),
   company character varying(150),
   address character varying(250),
   city character varying(30),
   state character varying(30),
   country character varying(30),
   postal_code character varying(30),
   phone character varying(30),
   fax character varying(30),
   email character varying(30),
   support_rep_id character varying(30),
   PRIMARY KEY (customer_id));
```

Violation-

- customer_id → city, state, country
- postal_code → city, state, country which **violates BCNF** because postal_code is not a superkey in customer.

Decomposition-

```
CREATE TABLE customer_location (
  postal_code VARCHAR(30) PRIMARY KEY, city
VARCHAR(30), state VARCHAR(30), country VARCHAR(30)
);
INSERT INTO customer_location
SELECT DISTINCT postal_code, city, state, country FROM
customer WHERE postal_code IS NOT NULL;
ALTER TABLE customer DROP COLUMN city, DROP
COLUMN state, DROP COLUMN country;
```

6. **Album-**

- Functional Dependency:
- album_id → title, artist_id
- BCNF Status:
- album_id is a superkey; no non-trivial FD violates BCNF.
- Justification-

The album table is already in BCNF because every album_id uniquely determines its title and artist_id such that its not possible to create a redundancy, partial dependency, and so no decomposition was needed.

7. **Artist-**

- Functional Dependency:
- artist_id → name
- BCNF Status:
- artist_id is a primary key directly determining the name.
- Justification

The artist table is in BCNF because the primary key artist_id can completely and uniquely determine the artist's name. No anomalies and no redundancies exist.

8. **Genre-**

- Functional Dependency:
- genre_id → name
- BCNF Status:
- Single simple key; no partial or transitive dependencies.
- Jusstification:

The genres table is entirely normalized because genre_id uniquely identifies the genre name and therefore does not contain evidence of redundancy, so this schema remains clean and simple.

9. **Media Type-**

- Functional Dependency:
- media_type_id → name
- BCNF Status:
- media_type_id is a primary key.
- Justification:

The media_types table is consistent with BCNF because each media_type_id directly corresponds to a single media type name, meaning there can be no duplication of data.

10. **Playlist-**

- Functional Dependency:
- playlist_id → name
- BCNF Status:
- playlist_id is a superkey; no issues with BCNF.
- Justification:

The playlist table is in normal form, because each playlist_id identifies a playlist name uniquely which prevents all redundancy and unnecessary data proliferation.

11. **Playlist Track-**

- Functional Dependency:
- (playlist_id, track_id) → (composite PK for this join table)
- BCNF Status:
- The combination (playlist_id, track_id) forms the full primary key; no partial or transitive dependencies exist.
- Justification:

The playlist_track table is already in BCNF, because since it represents a many-to-many relationship (between a playlist and a track), the composite primary key uniquely identifies each row.

After completing decompostion like mentioned above, the decomposition is completed, and all the tables are in BCNF.

Relation	Candidate Key(s)	Functional Dependencies (FDs)	BCNF Status
album	album_id	album_id → title, artist_id	Satisfies BCNF
artist	artist_id	artist_id → name	Satisfies BCNF
customer	customer_id	customer_id → name, email, support_rep_id, postal_code	Violates BCNF → Decomposed
employee	employee_id	employee_id → name, title, etc.; postal_code → city, state, country	Violates BCNF → Decomposed
genre	genre_id	genre_id → name	Satisfies BCNF
media_type	media_type_id	media_type_id → name	Satisfies BCNF
track	track_id	track_id → name, album_id, media_type_id, genre_id, etc.; media_type_id → unit_price	Violates BCNF → Decomposed
track_price	track_id	track_id → unit_price	Satisfies BCNF
invoice	invoice_id	invoice_id → customer_id, billing_info	Violates BCNF → Decomposed
invoice_line	invoice_line_id	invoice_line_id → invoice_id, track_id, quantity; track_id → unit_price	Violates BCNF → Decomposed
customer_billing	customer_id	customer_id → billing_address, billing_city, state, country, postal_code	Satisfies BCNF
media_type_pricing	media_type_id	media_type_id → unit_price	Satisfies BCNF
playlist	playlist_id	playlist_id → name	Satisfies BCNF
playlist_track	(playlist_id, track_id)	Composite key → determines each row	Satisfies BCNF
customer_location	postal_code	postal_code → city, state, country	Satisfies BCNF
location	postal_code	postal_code → city, state, country	Satisfies BCNF

Fig 3 – BCNF Analysis

VIII. Final BCNF Validated Schema

Table Name	Keys and Attributes
album	**PK:** album_id album_id INT, title VARCHAR(150), artist_id INT (FK)
artist	**PK:** artist_id artist_id INT, name VARCHAR(100)
customer	**PK:** customer_id customer_id INT, first_name, last_name, email, phone, address, postal_code, support_rep_id INT (FK)
employee	**PK:** employee_id employee_id INT, first_name, last_name, title, reports_to, levels, birth_date, hire_date, address, postal_code, phone, fax, email
genre	**PK:** genre_id genre_id INT, name VARCHAR(50)
media_type	**PK:** media_type_id media_type_id INT, name VARCHAR(50)
track	**PK:** track_id track_id INT, name, album_id (FK), media_type_id (FK), genre_id (FK), composer, milliseconds, bytes
track_price	**PK:** track_id track_id INT (FK), unit_price NUMERIC
media_type_pricing	**PK:** media_type_id media_type_id INT (FK), unit_price NUMERIC
invoice	**PK:** invoice_id invoice_id INT, customer_id INT (FK), total NUMERIC, invoice_date DATE
customer_billing	**PK:** customer_id customer_id INT (FK), billing_address, billing_city, billing_state, billing_country, billing_postal_code
invoice_line	**PK:** invoice_line_id invoice_line_id INT, invoice_id INT (FK), track_id INT (FK), quantity INT
playlist	**PK:** playlist_id playlist_id INT, name VARCHAR(100)
playlist_track	**PK:** (playlist_id, track_id) playlist_id INT (FK), track_id INT (FK)
customer_location	**PK:** postal_code postal_code VARCHAR(30), city, state, country
location	**PK:** postal_code postal_code VARCHAR(30), city, state, country

Fig 4 – Final Table after Decomposition

This confirms that all tables are in BCNF form. Each dependency has been explained.

IX. Finalized E/R Diagram

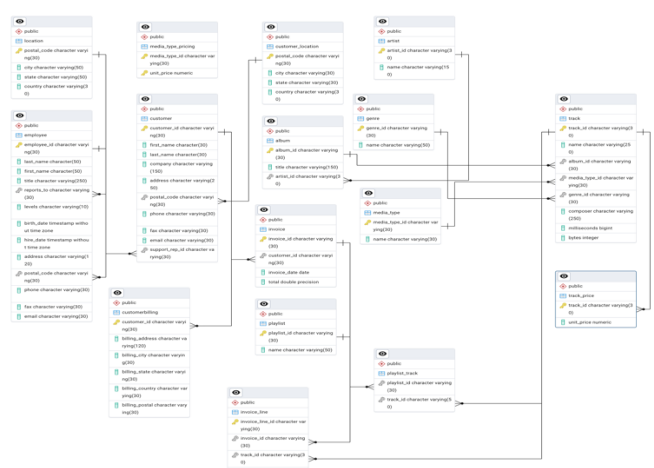


Fig 5 – Finalized E/R Diagram

The final Entity-Relationship (E/R) diagram for our Online Music Store Database System consists of 16 normalized relations, derived from our previous selection of 11 separate tables. After applying the rules for BCNF normalization, 5 additional tables (and corresponding relationships) were added to eliminate transitive dependencies and ensure all determinants are a superkey.

<u>Entity Design and Constraints</u>

The database contains its basic entities: Customer, Employee, Invoice, Invoice_Line, Track, Album, Artist, Genre, Media_Type, Playlist, and Playlist_Track. Further relations were created to ensure BCNF compliance: Location, CustomerBilling, MediaTypePricing, and TrackPrice.

Key constraints include:

Primary Keys: Every table has a unique identifier (e.g. customer_id, track_id, invoice_id).

Foreign Keys: Enforces referential integrity (e.g. Invoice.customer_id → Customer, Track.track_id → TrackPrice, postal_code → Location).

Composite Keys: Used in Playlist_Track in the case of many-to-many relations.

NOT NULL: Applied to mandatory fields to encourage completeness.

<u>BCNF and Handling of Redundancies</u>

Functional dependencies, such as postal_code → city,state, country, have been broken up into Location .

The CustomerBilling relation separates the customer address information from the Invoice .

The pricing logic is normalized into TrackPrice and MediaTypePricing.

All relations in all relations are now BCNF compliant and level of redundancy avoided ensures consistent updating.

Constraint Type	Purpose
Primary Keys	Ensure uniqueness of each record (e.g., customer_id, track_id, invoice_id)
Foreign Keys	Enforce referential integrity between tables (e.g., invoice.customer_id → customer.customer_id)
CHECK Constraints	Validate domain rules (e.g., quantity > 0 in invoice_line, rating BETWEEN 1 AND 5)
Composite Keys	Represent many-to-many relations (e.g., playlist_track(playlist_id, track_id))
Normalization Constraints	BCNF-compliant schema to avoid redundancy (e.g., TrackPrice, CustomerBilling, Location)

Fig 6 -Constraints identified and implemented

X. Challenged Faced with Dataset and Solutions

We encountered; while working with larger datasets, we encountered notable performance bottlenecks, particularly during join and filter operations across multiple tables.

One such challenge arose when executing the following query:

```
SELECT
  c.customer_id,
  c.first_name || ' ' || c.last_name AS customer_name,
  cl.city
FROM customer c
JOIN customer_location cl
  ON c.postal_code = cl.postal_code;
```

The join on postal_code between the customer and customer_location tables was significantly slower due to the absence of indexing on the join column. To address this, we created indexes on the postal_code attribute in both tables, which led to a substantial improvement in query speed.

In another query:

```
SELECT g.name, COUNT(t.track_id)
FROM genre g
JOIN track t ON g.genre_id = t.genre_id
GROUP BY g.name;
```

By indexing the genre_id and track_id columns, we enhanced the performance of aggregations and joins involving the track and genre tables. Additionally, we applied indexing to other frequently accessed fields like invoice_id and customer_id to boost overall query responsiveness. These indexing strategies were critical for maintaining performance and scalability as the dataset grew.

XI. Queries Executed

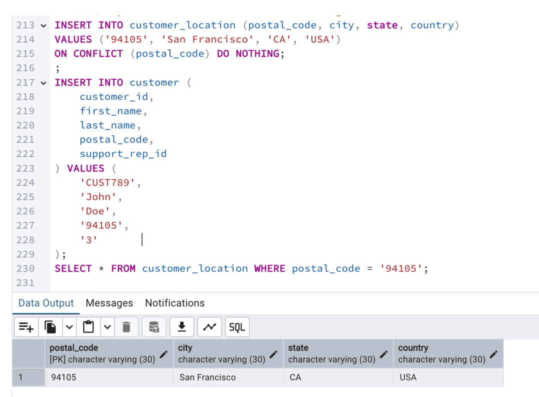


Fig 1 – Inserting into Table

This query first inserts a new location into the customer_location table, then adds a new customer linked to that postal code and finally retrieves the location details for postal code '94105'.

```
-- 3. DELETE (Remove one playlist-track mapping)
DELETE FROM playlist_track
WHERE playlist_id = '1'
  AND track_id = '3402';

-- 4. DELETE (Remove one invoice line)
DELETE FROM invoice_line
WHERE invoice_line_id = '1';

-- 5. UPDATE (Increase price for a specific track)
```

Output Messages Notifications

```
TE 1

y returned successfully in 34 msec.
```

Fig 2 – Deleting from play_list

This query removes a specific mapping between a playlist and a track from the playlist_track table using playlist_id = '1' and track_id = '3402'. It deletes one row, indicating that this track is no longer part of the specified playlist.

```
-- 4. DELETE (Remove one invoice line)
DELETE FROM invoice_line
WHERE invoice_line_id = '1';

-- 5. UPDATE (Increase price for a specific
UPDATE track_price
SET unit_price = unit_price * 1.10
WHERE track_id = '1';

-- 6. UPDATE (Adjust quantity on an invoice
UPDATE invoice_line
SET quantity = 2
```

Output Messages Notifications

```
TE 1

y returned successfully in 42 msec.
```

Fig 3 – Deleting from invoice_line

This query deletes '1' from invoice_line.

```
-- 5. UPDATE Increase price for a specific track
UPDATE track_price
SET unit_price = unit_price * 1.10
WHERE track_id = '1';
select * from track_price;
```

Data Output Messages Explain ✕ Notifications

```
UPDATE 1

Query returned successfully in 58 msec.
```

Fig 4 – Updating price

This SQL UPDATE statement increases the unit_price of the track with track_id = '1' by 10%. It multiplies the current price by 1.10 and updates the track_price table accordingly. The operation completed successfully, affecting one row.

```
-- 7. SELECT with JOIN Customers and their cities
SELECT
  c.customer_id,
  c.first_name || ' ' || c.last_name AS customer_name,
  cl.city
FROM customer c
JOIN customer_location cl
  ON c.postal_code = cl.postal_code;
```

Data Output Messages Notifications

	customer_id character varying (30)	customer_name text	city character varying (30)
1	customer_id	first_name last_name	city
2	1	Luís Gonçalves	São José dos Campos
3	2	Leonie Köhler	Stuttgart
4	3	François Tremblay	Montréal
5	4	Bjørn Hansen	Oslo
6	5	František Wichterlová	Prague
7	6	Helena Holý	Prague
8	7	Astrid Gruber	Vienne
9	8	Daan Peeters	Brussels
10	9	Kara Nielsen	Copenhagen
11	10	Eduardo Martins	São Paulo
12	11	Alexandre Rocha	São Paulo

Fig 5 – Joining customer and city

This query joins the customer and customer_location tables using postal_code to display each customer's name along with their city. It demonstrates how relational joins help in enriching customer data with geographic context.

```
-- 8. SELECT with GROUP BY Number of tracks per genre
SELECT
  g.name       AS genre_name,
  COUNT(t.track_id) AS track_count
FROM genre g
JOIN track t
  ON g.genre_id = t.genre_id
GROUP BY g.name;
```

Data Output Messages Notifications

	genre_name character varying (50)	track_count bigint
1	Heavy Metal	28
2	TV Shows	93
3	Latin	579
4	Electronica/Dance	30
5	R&B/Soul	61
6	Opera	1
7	Comedy	17
8	Classical	74
9	Pop	48
10	Easy Listening	24
11	Alternative	40
12	Drama	64
13	Sci Fi & Fantasy	26
14	Rock	1297
15	World	28
16	Reggae	58
17	Blues	81

Fig 6 – Selecting with group by

This query retrieves the number of tracks available in each genre by grouping and counting track IDs. It helps identify which genres have the most music content in the database.

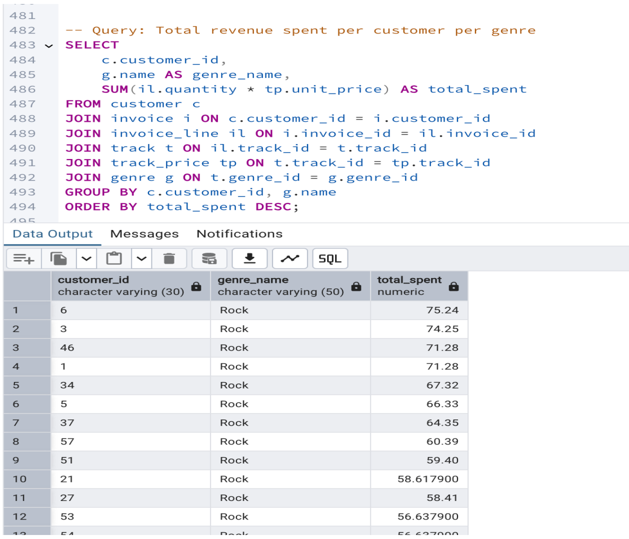
```
-- 9. SELECT with ORDER BY Top 5 longest tracks
SELECT
  track_id,
  name,
  milliseconds
FROM track
ORDER BY milliseconds DESC
LIMIT 5;
```

Data Output Messages Notifications

	track_id [PK] character varying (30)	name character varying (250)	milliseconds bigint
1	2820	Occupation / Precipice	5286953
2	3224	Through a Looking Glass	5088838
3	3244	Greetings from Earth, Pt...	2960293
4	3242	The Man With Nine Lives	2956998
5	3227	Battlestar Galactica, Pt. 2	2956081

Fig 7 – Select with Order By

This query returns the top 5 longest tracks by ordering all tracks in descending order of duration (milliseconds). It is useful for highlighting extended compositions or soundtracks in the database.



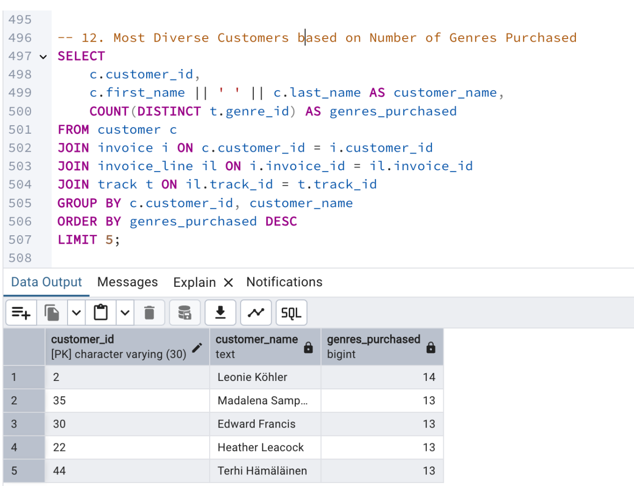
```sql
-- Query: Total revenue spent per customer per genre
SELECT
    c.customer_id,
    g.name AS genre_name,
    SUM(il.quantity * tp.unit_price) AS total_spent
FROM customer c
JOIN invoice i ON c.customer_id = i.customer_id
JOIN invoice_line il ON i.invoice_id = il.invoice_id
JOIN track t ON il.track_id = t.track_id
JOIN track_price tp ON t.track_id = tp.track_id
JOIN genre g ON t.genre_id = g.genre_id
GROUP BY c.customer_id, g.name
ORDER BY total_spent DESC;
```

Data Output Messages Notifications

	customer_id character varying (30)	genre_name character varying (50)	total_spent numeric
1	6	Rock	75.24
2	3	Rock	74.25
3	46	Rock	71.28
4	1	Rock	71.28
5	34	Rock	67.32
6	5	Rock	66.33
7	37	Rock	64.35
8	57	Rock	60.39
9	51	Rock	59.40
10	21	Rock	58.617900
11	27	Rock	58.41
12	53	Rock	56.637900

Fig 8 – Total revenue spent/customer

This query calculates the total amount each customer spent on each genre by joining invoices, invoice lines, tracks, and pricing details.
It groups results by customer and genre, showing the highest-spending customers per genre.



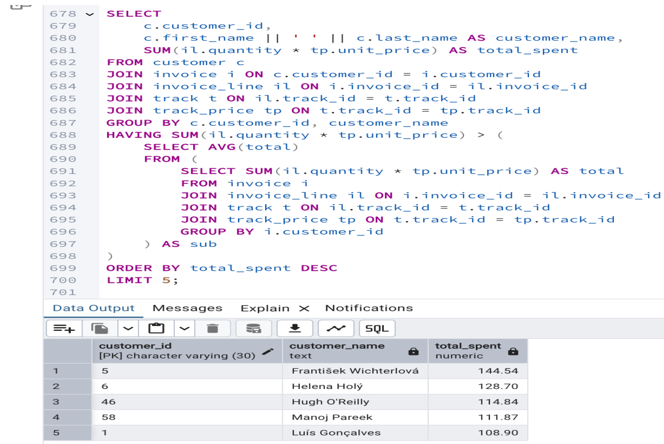
```sql
-- 12. Most Diverse Customers based on Number of Genres Purchased
SELECT
    c.customer_id,
    c.first_name || ' ' || c.last_name AS customer_name,
    COUNT(DISTINCT t.genre_id) AS genres_purchased
FROM customer c
JOIN invoice i ON c.customer_id = i.customer_id
JOIN invoice_line il ON i.invoice_id = il.invoice_id
JOIN track t ON il.track_id = t.track_id
GROUP BY c.customer_id, customer_name
ORDER BY genres_purchased DESC
LIMIT 5;
```

Data Output Messages Explain Notifications

	customer_id [PK] character varying (30)	customer_name text	genres_purchased bigint
1	2	Leonie Köhler	14
2	35	Madalena Samp...	13
3	30	Edward Francis	13
4	22	Heather Leacock	13
5	44	Terhi Hämäläinen	13

Fig 9 – Most Diverse Customers

This query identifies the most diverse customers by counting how many unique genres each has purchased from. It joins customer, invoice, invoice_line, and track tables to compute genre diversity per user. The result ranks customers based on the count of distinct genres, showing the top 5.



```sql
SELECT
    c.customer_id,
    c.first_name || ' ' || c.last_name AS customer_name,
    SUM(il.quantity * tp.unit_price) AS total_spent
FROM customer c
JOIN invoice i ON c.customer_id = i.customer_id
JOIN invoice_line il ON i.invoice_id = il.invoice_id
JOIN track t ON il.track_id = t.track_id
JOIN track_price tp ON t.track_id = tp.track_id
GROUP BY c.customer_id, customer_name
HAVING SUM(il.quantity * tp.unit_price) > (
    SELECT AVG(total)
    FROM (
        SELECT SUM(il.quantity * tp.unit_price) AS total
        FROM invoice i
        JOIN invoice_line il ON i.invoice_id = il.invoice_id
        JOIN track t ON il.track_id = t.track_id
        JOIN track_price tp ON t.track_id = tp.track_id
        GROUP BY i.customer_id
    ) AS sub
)
ORDER BY total_spent DESC
LIMIT 5;
```

Data Output Messages Explain Notifications

	customer_id [PK] character varying (30)	customer_name text	total_spent numeric
1	5	František Wichterlová	144.54
2	6	Helena Holý	128.70
3	46	Hugh O'Reilly	114.84
4	58	Manoj Pareek	111.87
5	1	Luís Gonçalves	108.90

Fig 10 – SubQuery

This query retrieves the top 5 customers whose total spending is above the average spending of all customers.

XII. Query Execution Analysis

1. Joining Customer, invoice and invoice_line table-

EXPLAIN ANALYZE
SELECT c.customer_id, c.first_name, c.last_name, SUM(il.quantity)
FROM customer c
JOIN invoice i ON c.customer_id = i.customer_id
JOIN invoice_line il ON i.invoice_id = il.invoice_id
GROUP BY c.customer_id, c.first_name, c.last_name
ORDER BY SUM(il.quantity) DESC;

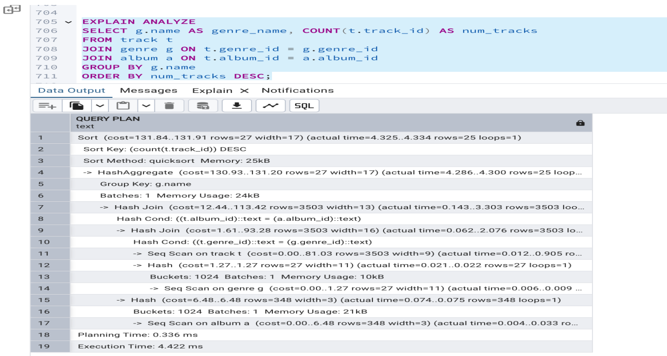
```sql
EXPLAIN ANALYZE
SELECT g.name AS genre_name, COUNT(t.track_id) AS num_tracks
FROM track t
JOIN genre g ON t.genre_id = g.genre_id
JOIN album a ON t.album_id = a.album_id
GROUP BY g.name
ORDER BY num_tracks DESC;
```

Data Output Messages Explain Notifications

	QUERY PLAN text
1	Sort (cost=131.84..131.91 rows=27 width=17) (actual time=7.293..7.299 rows=25 loops=1)
2	Sort Key: (count(t.track_id)) DESC
3	Sort Method: quicksort Memory: 25kB
4	-> HashAggregate (cost=130.93..131.20 rows=27 width=17) (actual time=7.261..7.271 rows=25 loop...
5	Group Key: g.name
6	Batches: 1 Memory Usage: 24kB
7	-> Hash Join (cost=12.44..113.42 rows=3503 width=13) (actual time=0.559..5.672 rows=3503 loo...
8	Hash Cond: ((t.album_id)::text = (a.album_id)::text)
9	-> Hash Join (cost=1.61..93.28 rows=3503 width=16) (actual time=0.061..3.294 rows=3503 lo...
10	Hash Cond: ((t.genre_id)::text = (g.genre_id)::text)
11	-> Seq Scan on track t (cost=0.00..81.03 rows=3503 width=9) (actual time=0.011..1.402 ro...
12	-> Hash (cost=1.27..1.27 rows=27 width=11) (actual time=0.040..0.041 rows=27 loops=1)
13	Buckets: 1024 Batches: 1 Memory Usage: 10kB
14	-> Seq Scan on genre g (cost=0.00..1.27 rows=27 width=11) (actual time=0.016..0.019 ...
15	-> Hash (cost=6.48..6.48 rows=348 width=3) (actual time=0.487..0.488 rows=348 loops=1)
16	Buckets: 1024 Batches: 1 Memory Usage: 21kB
17	-> Seq Scan on album a (cost=0.00..6.48 rows=348 width=3) (actual time=0.005..0.402 ro...
18	Planning Time: 0.757 ms
19	Execution Time: 7.377 ms

Fig 1 – Before Indexing

CREATE INDEX idx_invoice_customer_id ON invoice(customer_id);
CREATE INDEX idx_invoice_line_invoice_id ON invoice_line(invoice_id);



```
EXPLAIN ANALYZE
SELECT g.name AS genre_name, COUNT(t.track_id) AS num_tracks
FROM track t
JOIN genre g ON t.genre_id = g.genre_id
JOIN album a ON t.album_id = a.album_id
GROUP BY g.name
ORDER BY num_tracks DESC;
```

	QUERY PLAN text
1	Sort (cost=131.84..131.91 rows=27 width=17) (actual time=4.325..4.334 rows=25 loops=1)
2	Sort Key: (count(t.track_id)) DESC
3	Sort Method: quicksort Memory: 25kB
4	-> HashAggregate (cost=130.93..131.20 rows=27 width=17) (actual time=4.286..4.300 rows=25 loop...
5	Group Key: g.name
6	Batches: 1 Memory Usage: 24kB
7	-> Hash Join (cost=12.44..113.42 rows=3503 width=13) (actual time=0.143..3.303 rows=3503 loo...
8	Hash Cond: ((t.album_id)::text = (a.album_id)::text)
9	-> Hash Join (cost=1.61..93.28 rows=3503 width=16) (actual time=0.062..2.076 rows=3503 lo...
10	Hash Cond: ((t.genre_id)::text = (g.genre_id)::text)
11	-> Seq Scan on track t (cost=0.00..81.03 rows=3503 width=9) (actual time=0.012..0.905 ro...
12	-> Hash (cost=1.27..1.27 rows=27 width=11) (actual time=0.021..0.022 rows=27 loops=1)
13	Buckets: 1024 Batches: 1 Memory Usage: 10kB
14	-> Seq Scan on genre g (cost=0.00..1.27 rows=27 width=11) (actual time=0.006..0.009 ...
15	-> Hash (cost=6.48..6.48 rows=348 width=3) (actual time=0.074..0.075 rows=348 loops=1)
16	Buckets: 1024 Batches: 1 Memory Usage: 21kB
17	-> Seq Scan on album a (cost=0.00..6.48 rows=348 width=3) (actual time=0.004..0.033 ro...
18	Planning Time: 0.336 ms
19	Execution Time: 4.422 ms

Fig 2 – After Indexing

2. Joining track, genre and album with join and group by

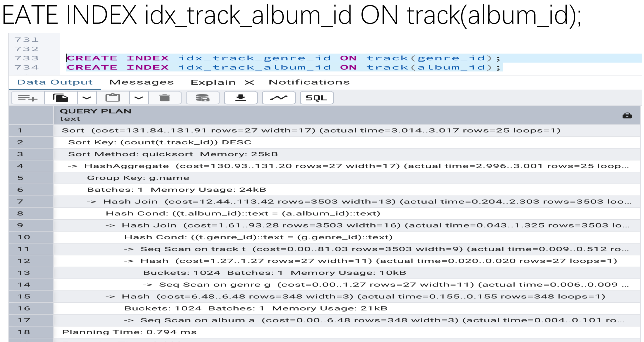
EXPLAIN ANALYZE
SELECT g.name AS genre_name, COUNT(t.track_id) AS num_tracks
FROM track t
JOIN genre g ON t.genre_id = g.genre_id
JOIN album a ON t.album_id = a.album_id
GROUP BY g.name
ORDER BY num_tracks DESC;

```
SELECT g.name AS genre_name, COUNT(t.track_id) AS num_tracks
FROM track t
JOIN genre g ON t.genre_id = g.genre_id
JOIN album a ON t.album_id = a.album_id
GROUP BY g.name
ORDER BY num_tracks DESC;
```

	QUERY PLAN text
1	Sort (cost=131.84..131.91 rows=27 width=17) (actual time=2.913..2.917 rows=25 loops=1)
2	Sort Key: (count(t.track_id)) DESC
3	Sort Method: quicksort Memory: 25kB
4	-> HashAggregate (cost=130.93..131.20 rows=27 width=17) (actual time=2.897..2.902 rows=25 loop...
5	Group Key: g.name
6	Batches: 1 Memory Usage: 24kB
7	-> Hash Join (cost=12.44..113.42 rows=3503 width=13) (actual time=0.146..2.255 rows=3503 loo...
8	Hash Cond: ((t.album_id)::text = (a.album_id)::text)
9	-> Hash Join (cost=1.61..93.28 rows=3503 width=16) (actual time=0.042..1.373 rows=3503 lo...
10	Hash Cond: ((t.genre_id)::text = (g.genre_id)::text)
11	-> Seq Scan on track t (cost=0.00..81.03 rows=3503 width=9) (actual time=0.011..0.600 ro...
12	-> Hash (cost=1.27..1.27 rows=27 width=11) (actual time=0.022..0.023 rows=27 loops=1)
13	Buckets: 1024 Batches: 1 Memory Usage: 10kB
14	-> Seq Scan on genre g (cost=0.00..1.27 rows=27 width=11) (actual time=0.007..0.010 ...
15	-> Hash (cost=6.48..6.48 rows=348 width=3) (actual time=0.093..0.093 rows=348 loops=1)
16	Buckets: 1024 Batches: 1 Memory Usage: 21kB
17	-> Seq Scan on album a (cost=0.00..6.48 rows=348 width=3) (actual time=0.003..0.042 ro...
18	Planning Time: 0.540 ms
19	Execution Time: 2.979 ms

Fig 3 - Before Indexing

CREATE INDEX idx_track_genre_id ON track(genre_id);
CREATE INDEX idx_track_album_id ON track(album_id);



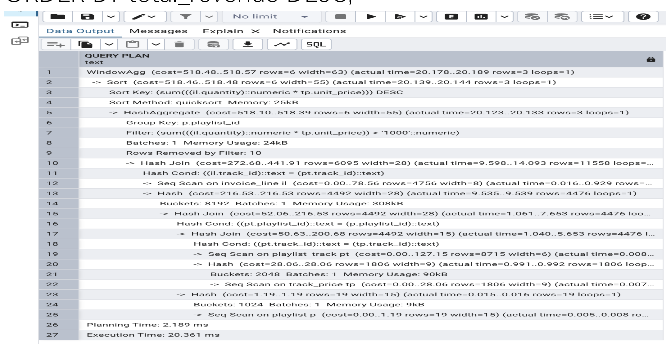
```
CREATE INDEX idx_track_genre_id ON track(genre_id);
CREATE INDEX idx_track_album_id ON track(album_id);
```

	QUERY PLAN text
1	Sort (cost=131.84..131.91 rows=27 width=17) (actual time=3.014..3.017 rows=25 loops=1)
2	Sort Key: (count(t.track_id)) DESC
3	Sort Method: quicksort Memory: 25kB
4	-> HashAggregate (cost=130.93..131.20 rows=27 width=17) (actual time=2.996..3.001 rows=25 loop...
5	Group Key: g.name
6	Batches: 1 Memory Usage: 24kB
7	-> Hash Join (cost=12.44..113.42 rows=3503 width=13) (actual time=0.204..2.303 rows=3503 loo...
8	Hash Cond: ((t.album_id)::text = (a.album_id)::text)
9	-> Hash Join (cost=1.61..93.28 rows=3503 width=16) (actual time=0.043..1.325 rows=3503 lo...
10	Hash Cond: ((t.genre_id)::text = (g.genre_id)::text)
11	-> Seq Scan on track t (cost=0.00..81.03 rows=3503 width=9) (actual time=0.009..0.512 ro...
12	-> Hash (cost=1.27..1.27 rows=27 width=11) (actual time=0.020..0.020 rows=27 loops=1)
13	Buckets: 1024 Batches: 1 Memory Usage: 10kB
14	-> Seq Scan on genre g (cost=0.00..1.27 rows=27 width=11) (actual time=0.006..0.009 ...
15	-> Hash (cost=6.48..6.48 rows=348 width=3) (actual time=0.155..0.155 rows=348 loops=1)
16	Buckets: 1024 Batches: 1 Memory Usage: 21kB
17	-> Seq Scan on album a (cost=0.00..6.48 rows=348 width=3) (actual time=0.004..0.101 ro...
18	Planning Time: 0.794 ms
19	Execution Time: 3.083 ms

Fig 4 – After Indexing

3. Playlist revenue Ranking

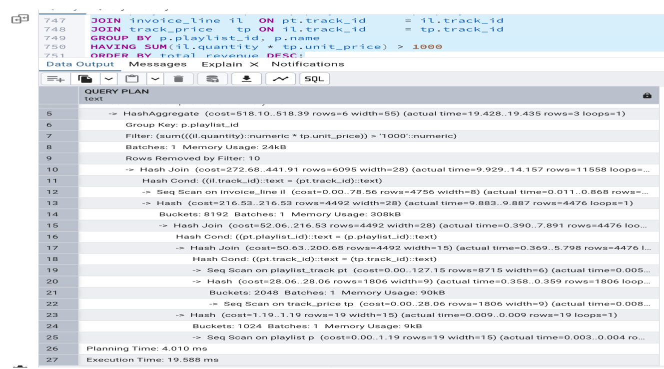
EXPLAIN ANALYZE
SELECT
  p.playlist_id,
  p.name AS playlist_name,
  COUNT(pt.track_id)   AS num_tracks,
  SUM(il.quantity * tp.unit_price)   AS total_revenue,
  RANK() OVER (ORDER BY SUM(il.quantity * tp.unit_price) DESC)
                          AS revenue_rank
FROM playlist p
JOIN playlist_track pt ON p.playlist_id = pt.playlist_id
JOIN invoice_line il ON pt.track_id   = il.track_id
JOIN track_price   tp ON il.track_id   = tp.track_id
GROUP BY p.playlist_id, p.name
HAVING SUM(il.quantity * tp.unit_price) > 1000
ORDER BY total_revenue DESC;



	QUERY PLAN text
1	WindowAgg (cost=518.48..518.57 rows=6 width=63) (actual time=20.178..20.189 rows=3 loops=1)
2	-> Sort (cost=518.46..518.48 rows=6 width=55) (actual time=20.139..20.144 rows=3 loops=1)
3	Sort Key: (sum(((il.quantity)::numeric * tp.unit_price))) DESC
4	Sort Method: quicksort Memory: 25kB
5	-> HashAggregate (cost=518.10..518.39 rows=6 width=55) (actual time=20.123..20.133 rows=3 loops=1)
6	Group Key: p.playlist_id
7	Filter: (sum(((il.quantity)::numeric * tp.unit_price)) > '1000'::numeric)
8	Batches: 1 Memory Usage: 24kB
9	Rows Removed by Filter: 10
10	-> Hash Join (cost=272.68..441.91 rows=6095 width=28) (actual time=9.598..14.093 rows=11558 loops=...
11	Hash Cond: ((il.track_id)::text = (pt.track_id)::text)
12	-> Seq Scan on invoice_line il (cost=0.00..78.56 rows=4756 width=8) (actual time=0.016..0.929 rows=...
13	-> Hash (cost=216.53..216.53 rows=4492 width=28) (actual time=9.535..9.539 rows=4476 loops=1)
14	Buckets: 8192 Batches: 1 Memory Usage: 308kB
15	-> Hash Join (cost=52.06..216.53 rows=4492 width=28) (actual time=1.061..7.653 rows=4476 loo...
16	Hash Cond: ((pt.playlist_id)::text = (p.playlist_id)::text)
17	-> Hash Join (cost=50.63..200.68 rows=4492 width=15) (actual time=1.040..5.653 rows=4476 l...
18	Hash Cond: ((pt.track_id)::text = (tp.track_id)::text)
19	-> Seq Scan on playlist_track pt (cost=0.00..127.15 rows=8715 width=6) (actual time=0.008...
20	-> Hash (cost=28.06..28.06 rows=1806 width=9) (actual time=0.991..0.992 rows=1806 loop...
21	Buckets: 2048 Batches: 1 Memory Usage: 90kB
22	-> Seq Scan on track_price tp (cost=0.00..28.06 rows=1806 width=9) (actual time=0.007...
23	-> Hash (cost=1.19..1.19 rows=19 width=15) (actual time=0.015..0.016 rows=19 loops=1)
24	Buckets: 1024 Batches: 1 Memory Usage: 9kB
25	-> Seq Scan on playlist p (cost=0.00..1.19 rows=19 width=15) (actual time=0.005..0.008 ro...
26	Planning Time: 2.189 ms
27	Execution Time: 20.361 ms

Fig 5 – Before Indexing

CREATE INDEX idx_playlist_track_playlist_id
  ON playlist_track(playlist_id);
CREATE INDEX idx_invoice_line_track_id
  ON invoice_line(track_id);
CREATE INDEX idx_track_price_track_id
  ON track_price(track_id);



```
JOIN invoice_line il  ON pt.track_id     = il.track_id
JOIN track_price   tp ON il.track_id     = tp.track_id
GROUP BY p.playlist_id, p.name
HAVING SUM(il.quantity * tp.unit_price) > 1000
ORDER BY total_revenue DESC;
```

	QUERY PLAN text
5	-> HashAggregate (cost=518.10..518.39 rows=6 width=55) (actual time=19.428..19.435 rows=3 loops=1)
6	Group Key: p.playlist_id
7	Filter: (sum(((il.quantity)::numeric * tp.unit_price)) > '1000'::numeric)
8	Batches: 1 Memory Usage: 24kB
9	Rows Removed by Filter: 10
10	-> Hash Join (cost=272.68..441.91 rows=6095 width=28) (actual time=9.929..14.157 rows=11558 loops=...
11	Hash Cond: ((il.track_id)::text = (pt.track_id)::text)
12	-> Seq Scan on invoice_line il (cost=0.00..78.56 rows=4756 width=8) (actual time=0.011..0.868 rows=...
13	-> Hash (cost=216.53..216.53 rows=4492 width=28) (actual time=9.883..9.887 rows=4476 loops=1)
14	Buckets: 8192 Batches: 1 Memory Usage: 308kB
15	-> Hash Join (cost=52.06..216.53 rows=4492 width=28) (actual time=0.390..7.891 rows=4476 loo...
16	Hash Cond: ((pt.playlist_id)::text = (p.playlist_id)::text)
17	-> Hash Join (cost=50.63..200.68 rows=4492 width=15) (actual time=0.369..5.798 rows=4476 l...
18	Hash Cond: ((pt.track_id)::text = (tp.track_id)::text)
19	-> Seq Scan on playlist_track pt (cost=0.00..127.15 rows=8715 width=6) (actual time=0.005...
20	-> Hash (cost=28.06..28.06 rows=1806 width=9) (actual time=0.358..0.359 rows=1806 loop...
21	Buckets: 2048 Batches: 1 Memory Usage: 90kB
22	-> Seq Scan on track_price tp (cost=0.00..28.06 rows=1806 width=9) (actual time=0.008...
23	-> Hash (cost=1.19..1.19 rows=19 width=15) (actual time=0.009..0.009 rows=19 loops=1)
24	Buckets: 1024 Batches: 1 Memory Usage: 9kB
25	-> Seq Scan on playlist p (cost=0.00..1.19 rows=19 width=15) (actual time=0.003..0.004 ro...
26	Planning Time: 4.010 ms
27	Execution Time: 19.588 ms

Fig 6 – After Indexing