

A Relational Database for Digital Music Store Management

DUSHYANTH SAI CHOWDARY NARRAVULA, University at Buffalo, 50596138

CHARAN KONDURU, University at Buffalo, 50596081

1 Introduction

Music has evolved from physical records like CD's and tapes to vast digital libraries, where millions of songs are accessible instantly. Digital music stores have transformed the way users explore, purchase, and stream music, offering seamless access to extensive collections. However, managing these platforms is complex, requiring the efficient handling of large volumes of data, including user interactions, transactions, and track metadata. Without a well-structured system, these stores can suffer from slow performance, data redundancy, and inconsistencies, ultimately impacting both users and administrators. A relational database serves as the foundation for managing this vast and dynamic ecosystem. While it does not solve all challenges on its own, it provides a structured framework that ensures data integrity, efficient retrieval, and scalability. By designing a well-optimized relational database, we lay the groundwork for building a reliable and high-performing digital music store, enabling seamless operations, better user experiences, and future scalability.

2 Problem Statement

In the digital age, managing a music store that offers millions of songs, albums, and playlists requires a highly efficient and scalable system to handle vast amounts of data. This project aims to design an efficient relational database for digital music store management, addressing the challenges of storing, organizing, and retrieving data related to songs, user transactions, playlists, and recommendations. As digital music platforms grow, the need for a structured database becomes crucial to ensure fast query responses, data consistency, and the ability to scale as the store's data expands.

While tools like Excel might seem like a simple solution for data management, they are not suited for handling large volumes of complex, relational data. Excel lacks the capability to efficiently process real-time transactions and is not designed to manage millions of records effectively. Additionally, it does not have a query language that can quickly retrieve and filter data based on specific criteria, which is essential for digital music store operations. A relational database provides a robust query language, enabling efficient data retrieval, while also maintaining data integrity and supporting the complex relationships between users, purchases, and songs. This makes it a far more effective solution than Excel for managing the dynamic and growing data of a digital music store.

3 Real Life Scenario

A relational database for digital music store management can be applied in a real-world scenario where an online platform sells, streams, and distributes music. This platform, similar to services like Spotify or Apple Music but focused on independent artists and small labels, allows users to purchase, stream, and download music. Additionally, artists and music labels can upload and manage their content while the platform ensures smooth transactions, licensing, and digital rights management. The database plays a crucial role in

storing and organizing information related to music tracks, albums, artists, purchases, subscriptions, user reviews, and royalty payments, ensuring efficient data retrieval and management.

The target users of this system include general customers who wish to explore and purchase music, independent artists and record labels who need a platform to distribute their work, and store administrators responsible for managing content, user accounts, and financial transactions. Customers will primarily interact with the system to search, stream, and buy music, while artists and labels will upload tracks, monitor their sales, and receive royalties. Store administrators oversee the platform's operations, approve content, handle customer issues, and ensure that the business runs smoothly.

The database administrators (DBAs) in this scenario are the technical experts responsible for managing the database infrastructure. They ensure database security, performance optimization, backup and recovery, and overall system maintenance. The DBA team typically includes lead database administrators who design and maintain the database schema, database security specialists who manage user permissions and prevent unauthorized access, and performance analysts who optimize queries and indexing for faster data retrieval. Additionally, system administrators may collaborate with DBAs to handle server maintenance and cloud storage solutions, ensuring high availability and scalability of the platform.

4 Milestone 1 Summary

In Milestone 1, we have created a draft ER diagram which includes the following tables such as Artist Table, Album Table, Track Table, Genre Table, MediaType Table, Invoice Table, Invoice Line Table, Customer Table, Employee Table. And we have created the fake data using python faker package and then we used python scripts to create and insert the database. And we ran few sql queries to showcase our database.

5 Entity Relationship Diagram

The ER diagram represents the relational structure of a digital music store management system, ensuring efficient organization and retrieval of data related to artists, albums, tracks, playlists, customers, employees, and purchases. The schema is designed with a normalized approach, ensuring minimal data redundancy while maintaining referential integrity through well-defined primary keys and foreign keys. Relationships between entities are carefully structured using ON DELETE constraints, ensuring that dependent records are either removed or updated appropriately when referenced entities are modified. This robust design facilitates seamless operations such as music purchases, playlist management, customer interactions, and employee support, making it a scalable and maintainable solution for managing a digital music store. Our Database Schema Includes following tables.

The **Artist** table serves as the foundation for storing information about artists in the digital music store. The **primary key** for this

table is **ArtistId**, which uniquely identifies each artist. This key ensures that no two artists have the same identifier, maintaining the integrity of the data. The table consists of two attributes: **ArtistId**, which is of type **INTEGER**, and **Name**, which is of type **TEXT**. The **Name** attribute stores the artist's name, allowing the system to associate albums and tracks with the respective artist. There is no foreign key in this table since it is a standalone entity. Since **ArtistId** is primary key so it cannot be and Name of the artist is also not Null.

The **Album** table represents music albums available in the store. The primary key for this table is **AlbumId**, which is of type **SERIAL** (**INTEGER**) and uniquely identifies each album. This key ensures that each album has a distinct identifier, maintaining the integrity of the dataset. The table also contains a foreign key, **ArtistId**, which references **Artist(ArtistId)** in the **Artist** table. This relationship establishes a one-to-many association, meaning one artist can have multiple albums, but each album belongs to only one artist. The **Title** attribute, of type **TEXT**, stores the album's title and is marked as **NOT NULL**, ensuring that every album must have a name. The foreign key constraint uses **ON DELETE CASCADE**, meaning if an artist record is deleted, all associated albums are automatically removed, preventing orphaned album records. Since **ArtistId** is not explicitly declared as **NOT NULL**, it can be **NULL** if an album is not linked to any artist, allowing flexibility for orphan albums or compilations.

The **Track** table stores information about individual songs or audio tracks available on the platform. The primary key is **TrackId**, a **SERIAL** type that uniquely identifies each track. This table includes multiple foreign keys: **AlbumId**, which references the **Album** table with an **ON DELETE CASCADE** action to ensure that all tracks associated with a deleted album are also removed; and **MediaTypeId** and **GenreId**, which reference the **MediaType** and **Genre** tables respectively, both with an **ON DELETE SET NULL** constraint. This means that if a media type or genre is deleted, the corresponding field in the **Track** table is set to **NULL**, preserving the track record. Additional attributes in this table include **Name** (**TEXT**) for the track title, **Composer** (**TEXT**) for the composer's name, **Milliseconds** (**INTEGER**) for the duration of the track, and **Bytes** (**INTEGER**) for the file size. None of these attributes have default values, and except for the primary key, all are optional unless otherwise enforced through application logic. The **Track** table is central to the schema, acting as a link between albums, genres, and media types for each audio entry. The **Playlist** table stores playlists created by users.

The **primary key** is **PlaylistId**, which is an **INTEGER** uniquely identifying each playlist. The table contains a **Name** attribute of type **TEXT**, which represents the playlist's name. There is no foreign key in this table, making it independent. The **Name** attribute does not have a default value and it must be not Null.

The **PlaylistTrack** table establishes a many-to-many relationship between playlists and tracks, as a single track can belong to multiple playlists, and a single playlist can contain multiple tracks. The primary key is a composite key consisting of **PlaylistId** and

TrackId, both of which act as foreign keys where **PlaylistId** references the **Playlist** table with **ON DELETE CASCADE**, meaning that if a playlist is deleted, all associated tracks in that playlist are also removed and **TrackId** references the **Track** table with **ON DELETE CASCADE**, meaning if a track is deleted, its reference in all playlists is also removed. Neither attribute has a default value, and both are required fields.

The **MediaType** table holds information about the different formats in which tracks are available (e.g., Audio, Video, Podcasts etc.). The primary key is **MediaTypeId**, which is an **INTEGER** uniquely identifying each media type. The table also has a **Name** attribute of type **TEXT**, which describes the media format. There is no foreign key in this table, making it independent of others. The **Name** attribute does not have a default value and cannot be set to **Null**.

The **Genre** table is used to categorize tracks into various musical genres such as Rock, Jazz, Classical, and so on. It contains two attributes: **GenreId**, which is of type **SERIAL** and serves as the **primary key**, ensuring that each genre is uniquely identified in the table. **Name**, which is of type **TEXT** and is marked as **NOT NULL**, meaning every genre record must have a name.

The **Employee** table stores information about store employees. The **primary key** is **EmployeeId**, which is an **INTEGER** uniquely identifying each employee. Attributes include **LastName**, **FirstName**, **Title** (all **TEXT**) for storing personal details, and **ReportsTo**, which is a foreign key referencing another **EmployeeId** to define hierarchical relationships among employees. If an employee who supervises others is deleted, the **ON DELETE SET NULL** constraint ensures that the **ReportsTo** field of subordinate employees is set to **NULL** instead of removing their records. Other attributes such as **BirthDate** and **HireDate** (**DATE**) store personal and employment information, while **Address**, **City**, **State**, **Country**, **PostalCode**, **Phone**, **Fax**, and **Email** (all **TEXT**) contain contact details. None of the attributes have default values, and most fields can be **NULL** except **EmployeeId**.

The **Customer** table holds details of users who purchase music from the store. The primary key is **CustomerId**, an **INTEGER** uniquely identifying each customer. Attributes include **FirstName**, **LastName**, **Company**, **Address**, **City**, **State**, **Country**, **PostalCode**, **Phone**, **Fax**, and **Email** (all **TEXT**). The **SupportRepId** is a foreign key referencing an **Employee** who assists the customer. If an employee is removed, the **ON DELETE SET NULL** constraint ensures that the associated customers remain in the database but without a support representative assigned. Most attributes can be **NULL** except **CustomerId**.

The **Invoice** table records customer purchases. The **primary key** is **InvoiceId**, an **INTEGER** uniquely identifying each invoice. The table contains a **foreign key**, **CustomerId**, referencing the **Customer** table. The **ON DELETE CASCADE** action ensures that if a customer is deleted, all related invoices are also removed. Other attributes include **InvoiceDate** (**DATE**), **BillingAddress**, **BillingCity**, **BillingState**, **BillingCountry**, **BillingPostalCode** (all **TEXT**), and **Total** (**DECIMAL(10,2)**) for the total purchase amount. None

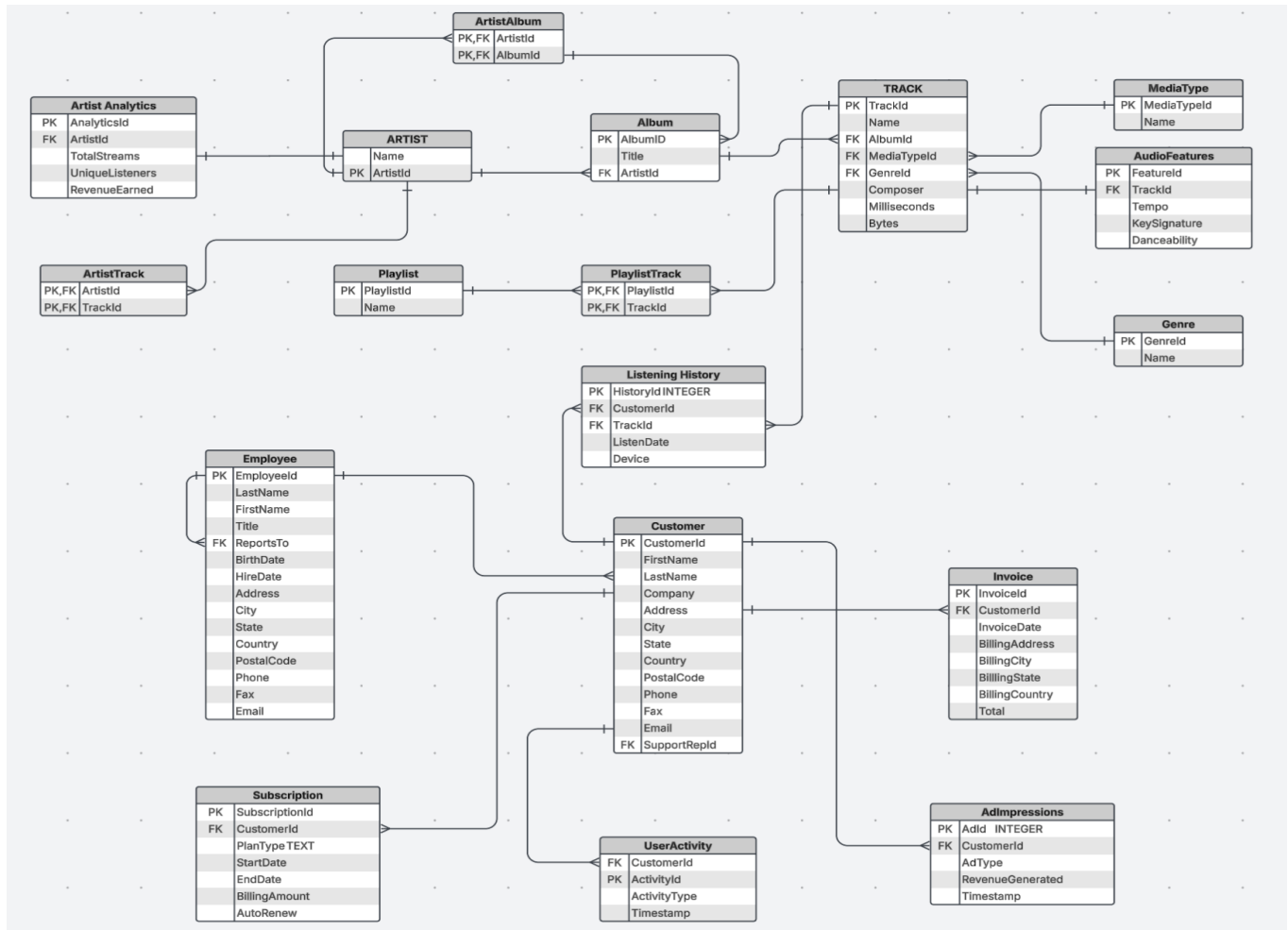


Fig. 1. ER-diagram representation of Digital Music Database

of the attributes have default values and may have Null Values. But **CustomerId** must not be Null

The **Subscription** table stores information about the subscription plans associated with customers on the platform. The primary key is **SubscriptionId**, which is of type SERIAL and uniquely identifies each subscription record. The table includes a foreign key **CustomerId**, which references the Customer table with an **ON DELETE CASCADE** constraint. This means that if a customer is deleted, all of their associated subscription records are automatically removed to maintain referential integrity. Other attributes include **PlanType** (TEXT), which specifies the type of subscription plan (such as Free or Premium), **StartDate** (DATE) and **EndDate** (DATE) to track the duration of the subscription, **BillingAmount** (DECIMAL(10,2)) indicating the recurring fee associated with the subscription, and **AutoRenew** (BOOLEAN) to denote whether the subscription is set to renew automatically.

The **UserActivity** table captures and stores the actions performed by customers on the platform, allowing for detailed tracking of user interactions. The primary key is **ActivityId**, which is of type SERIAL and uniquely identifies each recorded activity. The table includes a **foreign key CustomerId**, referencing the **Customer** table with an **ON DELETE CASCADE** constraint, meaning that if a customer is deleted, all their associated activity records are also automatically removed from the database. Additional attributes include **Activity-Type** (TEXT), which describes the type of action performed (such as Play, Pause, Skip, etc.), and **Timestamp** (TIMESTAMP), which records the exact date and time when the activity occurred.

The **ListeningHistory** table records the listening behavior of customers by tracking which tracks they have listened to and when. The primary key is **HistoryId**, which is of type SERIAL and uniquely identifies each listening record. The table includes two foreign keys: **CustomerId**, which references the **Customer** table, and **TrackId**, which references the **Track** table, both with an **ON DELETE CASCADE** constraint. This ensures that if a customer or a track is

deleted, all corresponding listening history entries are also automatically removed, maintaining referential integrity. Other attributes include **ListenDate** (TIMESTAMP), which captures the exact date and time the listening event occurred, and **Device** (TEXT), which specifies the type of device used for listening, such as Mobile, Desktop, or Tablet.

The **ArtistAnalytics** table stores performance and revenue-related metrics for artists on the platform. The primary key is **AnalyticId**, which is of type SERIAL and uniquely identifies each analytics record. The table includes a **foreign key ArtistId**, which references the **Artist** table with an **ON DELETE CASCADE** constraint. This ensures that if an artist is deleted, all corresponding analytics records are also automatically removed, maintaining the consistency of the database. Additional attributes in the table include **TotalStreams** (INTEGER), which captures the total number of times an artist's tracks have been streamed, **UniqueListeners** (INTEGER), which records the number of distinct users who have listened to the artist's tracks, and **RevenueEarned** (DECIMAL(10,2)), which reflects the total revenue generated from streaming the artist's music. None of these attributes have default values.

The **AdImpressions** table stores information about advertisements viewed by customers. The **primary key** is **AdId**, a SERIAL type that uniquely identifies each ad impression. It includes a **foreign key CustomerId**, referencing the **Customer** table with an **ON DELETE CASCADE** constraint, ensuring that when a customer is deleted, their ad impressions are also removed. Other attributes include **AdType** (TEXT) to specify the type of advertisement, **RevenueGenerated** (DECIMAL(10,2)) to capture revenue from the impression, and **Timestamp** (TIMESTAMP) to record when the ad was shown. None of these attributes have default values, and while not enforced as **NOT NULL**, they are important for tracking ad performance and customer engagement.

The **AudioFeatures** table stores technical characteristics of individual tracks. The primary key is **FeatureId**, a SERIAL type that uniquely identifies each feature record. It includes a foreign key **TrackId**, referencing the **Track** table with an **ON DELETE CASCADE** constraint, ensuring that if a track is deleted, its associated audio features are also removed. Other attributes include **Tempo** (INTEGER), representing the track's beats per minute, **KeySignature** (TEXT), indicating the musical key of the track, and **Danceability** (DECIMAL(3,2)), measuring how suitable the track is for dancing on a scale from 0.00 to 1.00.

The **ArtistAlbum** table establishes a many-to-many relationship between artists and albums. It has a composite primary key consisting of **ArtistId** and **AlbumId**, ensuring each artist-album pairing is unique. Both attributes act as foreign keys: **ArtistId** references the **Artist** table and **AlbumId** references the **Album** table, each with an **ON DELETE CASCADE** constraint. This means that if either an artist or an album is deleted, all related records in this table are also removed. The table does not contain any additional attributes or default values, and both fields are required to maintain the integrity of the association.

The **ArtistTrack** table uses a composite primary key consisting of **ArtistId** and **TrackId**, ensuring that each artist-track pairing is unique. Both attributes serve as foreign keys: **ArtistId** references the **Artist** table and **TrackId** references the **Track** table, each with an **ON DELETE CASCADE** constraint. This setup ensures that if either an artist or a track is deleted, all related associations in this table are automatically removed. The table contains no additional attributes, and both fields are required to maintain the integrity of artist-track collaborations.

6 BCNF Verification

The **Artist** table consists of attributes **ArtistId** and **Name**. The functional dependency identified here is **ArtistId** → **Name**, meaning that each artist ID uniquely determines an artist's name. Since **ArtistId** is the primary key and a superkey for this relation, the table satisfies the conditions for Boyce-Codd Normal Form (BCNF). Therefore, no decomposition is needed.

The **Album** table includes **AlbumId**, **Title**, and **ArtistId**. The functional dependency is **AlbumId** → **Title**, **ArtistId**, where **AlbumId** uniquely identifies an album and its associated artist. As **AlbumId** is a candidate key and is the determinant for all attributes, the **Album** relation is already in BCNF.

The **Genre** table contains **GenreId** and **Name**. The functional dependency is **GenreId** → **Name**. Since **GenreId** is the primary key and uniquely identifies the genre name, the table adheres to BCNF without any need for decomposition. The **MediaType** table includes **MediaTypeId** and **Name**. The functional dependency here is **MediaTypeId** → **Name**. Given that **MediaTypeId** is a primary key and determines the other attribute, the table is already in BCNF.

The **Employee** table contains attributes such as **EmployeeId**, **LastName**, **FirstName**, **Title**, **ReportsTo**, **BirthDate**, **HireDate**, **Address**, **City**, **State**, **Country**, **PostalCode**, **Phone**, **Fax**, and **Email**. The functional dependency identified is **EmployeeId** → (**LastName**, **FirstName**, **Title**, **ReportsTo**, **BirthDate**, **HireDate**, **Address**, **City**, **State**, **Country**, **PostalCode**, **Phone**, **Fax**, and **Email**). Since **EmployeeId** is a superkey, the relation satisfies BCNF and does not require decomposition.

The **Track** table consists of **TrackId**, **Name**, **AlbumId**, **MediaTypeId**, **GenreId**, **Composer**, **Milliseconds**, and **Bytes**. The dependency **TrackId** → (**Name**, **AlbumId**, **MediaTypeId**, **GenreId**, **Composer**, **Milliseconds**, **Bytes**) ensures that each track is uniquely identified by its **TrackId**. Since **TrackId** is a superkey, the **Track** table is already in BCNF. The **Playlist** table has attributes **PlaylistId** and **Name**. The functional dependency **PlaylistId** → **Name** holds, meaning each playlist is uniquely determined by its ID. Since **PlaylistId** is the primary key, the table satisfies BCNF.

The **Customer** table consists of **CustomerId**, **FirstName**, **LastName**, **Company**, **Address**, **City**, **State**, **Country**, **PostalCode**, **Phone**, **Fax**, **Email**, and **SupportRepId**. The functional dependency **CustomerId** → (**CustomerId**, **FirstName**, **LastName**, **Company**, **Address**, **City**, **State**, **Country**, **PostalCode**, **Phone**, **Fax**, **Email**, and **SupportRepId**) shows that each customer ID uniquely determines all

associated customer details. Since `customerId` is superkey therefore, the table is already in BCNF.

The `PlaylistTrack` table has a composite primary key consisting of `PlaylistId` and `TrackId`. There are no additional attributes dependent on this composite key. Since the combination of `PlaylistId` and `TrackId` serves as the superkey, this relation is in BCNF. The `ArtistTrack` table also has a composite primary key consisting of `ArtistId` and `TrackId`. No partial dependencies exist, and the composite key uniquely identifies each entry. Thus, the table satisfies BCNF. The `ArtistAlbum` table contains a composite key composed of `ArtistId` and `AlbumId`. Since there are no other attributes in the table so `ArtistId`, `AlbumId` combininly determies each row in a table. Also the entire composite key serves as the superkey, the table is already in BCNF.

The `Invoice` table includes `InvoiceId`, `CustomerId`, `InvoiceDate`, `BillingAddress`, `BillingCity`, `BillingState`, `BillingCountry`, `BillingPostalCode`, and `Total`. The functional dependency is **`InvoiceId` → (`CustomerId`, `InvoiceDate`, `BillingAddress`, `BillingCity`, `BillingState`, `BillingCountry`, `BillingPostalCode`, `Total`)**. As `InvoiceId` uniquely identifies each invoice and determines all other fields and it is the superkey, the table is already in BCNF.

The `Subscription` table consists of `SubscriptionId`, `CustomerId`, `PlanType`, `StartDate`, `EndDate`, `BillingAmount`, and `AutoRenew`. The dependency **`SubscriptionId` → (`CustomerId`, `PlanType`, `StartDate`, `EndDate`, `BillingAmount`, `AutoRenew`)** implies that each subscription ID uniquely determines all other details. Hence, the table is already in BCNF.

The `UserActivity` table contains `ActivityId`, `CustomerId`, `ActivityType`, and `Timestamp`. The functional dependency **`ActivityId` → (`CustomerId`, `ActivityType`, `Timestamp`)** ensures uniqueness through the activity ID. As a result, the table is already in BCNF. The `ListeningHistory` table comprises `HistoryId`, `CustomerId`, `TrackId`, `ListenDate`, and `Device`. The dependency **`HistoryId` → (`CustomerId`, `TrackId`, `ListenDate`, `Device`)** ensures that each history ID uniquely determines listening details. Consequently, the table is in BCNF.

The `AdImpressions` table includes `AdId`, `CustomerId`, `AdType`, `RevenueGenerated`, and `Timestamp`. The functional dependency is **`AdId` → (`CustomerId`, `AdType`, `RevenueGenerated`, `Timestamp`)**. Since `AdId` is a superkey, this relation satisfies BCNF. The `AudioFeatures` table contains `FeatureId`, `TrackId`, `Tempo`, `KeySignature`, and `Danceability`. The functional dependency **`FeatureId` → (`TrackId`, `Tempo`, `KeySignature`, `Danceability`)** guarantees uniqueness. Since `FeatureId` is SuperKey, then the table is already in BCNF.

The `ArtistAnalytics` table consists of `AnalyticsId`, `ArtistId`, `TotalStreams`, `UniqueListeners`, and `RevenueEarned`. The dependency **`AnalyticsId` → (`ArtistId`, `TotalStreams`, `UniqueListeners`, `RevenueEarned`)** shows that each analytics record is uniquely determined. Since `AnalyticsId` is superkey then the `ArtistAnalytics` table satisfies BCNF.

7 SQL Queries

Here we have run 4 sample SQL Queries on our database, with `SELECT` clause, such as `GROUP BY`, sub-queries, and `JOIN` etc.

1. Select artist name and total track count for top 10 Rock bands.

Query Query History

```
1 SELECT Artist.Name, COUNT(Track.TrackId) AS TotalTracks
2 FROM Artist
3 JOIN Album ON Artist.ArtistId = Album.ArtistId
4 JOIN Track ON Album.AlbumId = Track.AlbumId
5 JOIN Genre ON Track.GenreId = Genre.GenreId
6 WHERE Genre.Name = 'Rock'
7 GROUP BY Artist.Name
8 ORDER BY TotalTracks DESC
9 LIMIT 10;
```

Fig. 2. SQL Query for Top 10 Rock bands by tracks

Data Output			Messages	Notifications
	name text	totaltracks bigint		
1	Jason Sanchez	4		
2	Michelle Bryant	4		
3	Sarah Ramsey	4		
4	Justin Ward	3		
5	John McCormi...	3		
6	Justin Mcdowell	3		
7	Janice Nelson	3		
8	Beth Haynes	3		
9	Gregory Romero	3		
10	Aaron Meza	3		

Fig. 3. Output of Top10 rock bands Query

2. Find top 3 tracks for each genre by total listens

Query Query History

```
1 WITH GenreTrackCounts AS (
2 SELECT GenreName, TrackName, COUNT(ListeningHistory.HistoryId) AS ListenCount,
3 ROW_NUMBER() OVER (PARTITION BY GenreName ORDER BY COUNT(ListeningHistory.HistoryId) DESC) AS RowNum
4 FROM Genre
5 JOIN Track ON Genre.GenreId = Track.GenreId
6 JOIN ListeningHistory ON Track.TrackId = ListeningHistory.TrackId
7 GROUP BY GenreName, TrackName
8 )
9 SELECT GenreName, TrackName, ListenCount
10 FROM GenreTrackCounts
11 WHERE RowNum <= 3
12 ORDER BY GenreName, ListenCount DESC;
```

Fig. 4. SQL Query for Top 3 tracks per genre

	genrename text	trackname text	listencount bigint
1	Alternative	Especialy trip.	8
2	Alternative	Else.	8
3	Alternative	Picture think.	7
4	Blues	Month.	9
5	Blues	Voice.	9
6	Blues	Whole.	8
7	Classical	Word do.	8
8	Classical	Identify.	8
9	Classical	Interesting.	8
10	Electronic	Alone.	8
11	Electronic	Develop.	8
12	Electronic	Answer.	8
13	Hip Hop	Player.	7

Fig. 5. Output of Top 3 tracks per genre Query

3. List all customers and classify them as 'High Spender' or 'Low Spender' based on total purchases.

```

Query  Query History
1  SELECT Customer.FirstName, Customer.LastName,
2      CASE
3          WHEN SUM(Invoice.Total) > 100 THEN 'High Spender'
4          ELSE 'Low Spender'
5      END AS SpendingCategory
6  FROM Customer
7  JOIN Invoice ON Customer.CustomerId = Invoice.CustomerId
8  GROUP BY Customer.CustomerId, Customer.FirstName, Customer.LastName;

```

Fig. 6. SQL Query for Customer spending classification

	firstname text	lastname text	spendingcategory text
1	Randall	Ferrell	Low Spender
2	Lisa	Perry	Low Spender
3	Raymond	Peterson	High Spender
4	George	Bradley	Low Spender
5	Angela	Russell	Low Spender
6	Colleen	Brady	Low Spender
7	Michael	Harris	High Spender
8	Kelly	Dickerson	High Spender
9	Dawn	Hughes	High Spender
10	Laura	Castillo	Low Spender
11	Shannon	Wilson	Low Spender
12	Christopher	Evans	Low Spender
13	Thomas	Day	Low Spender

Fig. 7. Output of Customer spending classification Query

4. Get playlists that contain tracks from more than 5 different albums

```

Query  Query History
1  SELECT p.Name, COUNT(DISTINCT t.AlbumId) AS AlbumCount
2  FROM Playlist p
3  JOIN PlaylistTrack pt ON p.PlaylistId = pt.PlaylistId
4  JOIN Track t ON pt.TrackId = t.TrackId
5  GROUP BY p.PlaylistId, p.Name
6  HAVING COUNT(DISTINCT t.AlbumId) > 5;
7

```

Fig. 8. SQL Query to Playlists with >5 different albums

	name text	albumcount bigint
1	indeed	18
2	former	21
3	tax	21
4	peace	26
5	individual	11
6	activity	15
7	thing	18
8	success	17
9	learn	25
10	charge	19
11	plan	24
12	follow	16
13	research	28
Total rows: 500		Query complete 00:00:00.125

Fig. 9. Output of Playlists with >5 different albums Query

5. Find average song duration (milliseconds) for each genre

```

Query  Query History
1  SELECT Genre.Name, AVG(Track.Milliseconds) AS AvgDuration
2  FROM Genre
3  JOIN Track ON Genre.GenreId = Track.GenreId
4  GROUP BY Genre.Name
5  ORDER BY AvgDuration DESC;

```

Fig. 10. SQL Query to Average song duration by genre

Data Output			Messages	Notifications
	name	avgduration		
	text	numeric		
1	Blues	232948.196319018405		
2	Reggae	232543.102362204724		
3	Metal	229531.228744939271		
4	Electronic	228372.510476190476		
5	Alternative	227328.122699386503		
6	Classical	227205.887005649718		
7	Hip Hop	226621.233962264151		
8	Rock	225828.766595289079		
9	Pop	225165.985074626866		
10	Jazz	223326.337349397590		

Fig. 11. Output of Average song duration by genre Query

6. Update billing country for all customers in a particular state

Query	Query History
1	UPDATE Customer
2	SET Country = 'United States'
3	WHERE State = 'NY';
4	
5	Select * From Customer Where state = 'NY'
6	

Fig. 12. SQL Query to Update billing country for customers

Data OutputMessagesNotifications

Data Output

SQL

Showing rows: 1 to 81Page No: 1of 1

	customer [PK] integer	firstname text	lastname text	company text	address text	city text	state text	country text	postalcode text
1	37	Christopher	Camacho	Walker, Weiss and Mccarty	5363 Baker Cape Suite 455	Scottchester	NY	United States	11838
2	3214	Joseph	Moore	May, Wilson and Bishop	63966 Johnson River Suite 179	Enribury	NY	United States	73165
3	3265	Joshua	Garcia	[null]	66841 Johnson Glen Suite 058	Stevenshire	NY	United States	57092
4	3091	Diane	Martinez	Anderson-Martin	939 Brian Passage	Thomashaven	NY	United States	40373
5	46	George	King	Gonzalez Perez	372 Powell Trace	Port Austin	NY	United States	14609
6	3331	John	Bell	[null]	Unit 6306 Box 1599	West Kathrynburgh	NY	United States	41377
7	50	Matthew	Walter	[null]	9105 Daniel Junctions Suite 288	Cynohlastad	NY	United States	84761
8	58	Christina	Shea	Brown-Walsh	8355 Cox Vista Suite 821	New Davidmouth	NY	United States	41040
9	87	Anna	Carter	[null]	3896 Brenda Greens Apt. 317	East Ashlee	NY	United States	52384
10	104	Kristine	Garcia	[null]	1006 Joseph Freeway Apt. 702	Launamouth	NY	United States	74632
11	145	Shannon	Evans	[null]	957 Strickland Square	East Theresa	NY	United States	08921
12	147	Susan	Zamora	[null]	8517 Joshua Crest	South Kimberlerville	NY	United States	31652

Total rows: 81

Query complete 00:00:00.198

CRLF

Ln 5, Col 41

Fig. 13. Output of Update billing country for customers Query

Data Output	Messages	Notifications
UPDATE 81		
	Query returned successfully in 141 msec.	

Fig. 14. Update to total number of rows in the table

7. Update subscription plan type from 'Free' to 'Premium' for active customers

Query	Query History
1	UPDATE Subscription
2	SET PlanType = 'Premium'
3	WHERE AutoRenew = TRUE
4	AND EndDate > CURRENT_DATE;
5	
6	Select * From Subscription Where AutoRenew = TRUE
7	

Fig. 15. SQL Query to Upgrade subscription plan for users








Data Output		Messages	Notifications				
<div><div><div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div><div>SQL</div></div></div>							
	subscriptionid  [PK] integer	customerid  integer	plantype  text	startdate  date	enddate  date	billingamount  numeric (10,2)	autorenew  boolean
1	5	3045	Premium	2023-07-06	2025-08-14	9.99	true
2	8	1607	Premium	2023-12-04	2025-09-26	9.99	true
3	12	3388	Premium	2024-11-28	2025-11-08	9.99	true
4	13	4826	Premium	2024-10-29	2026-03-31	9.99	true
5	16	963	Premium	2025-03-11	2025-06-13	9.99	true
6	17	3399	Premium	2024-01-27	2025-12-08	9.99	true
7	20	1914	Premium	2024-02-20	2025-10-01	9.99	true
8	21	1664	Premium	2023-12-06	2025-10-26	9.99	true
9	25	1730	Premium	2025-03-30	2025-07-03	9.99	true
10	26	1578	Premium	2023-06-06	2025-11-21	9.99	true
11	27	3780	Premium	2025-04-11	2026-02-14	9.99	true
12	28	2502	Premium	2024-08-01	2025-05-27	9.99	true
13	29	2293	Premium	2024-09-11	2026-04-19	9.99	true
Total rows: 2034		Query complete 00:00:00.137					

Fig. 16. Output of Upgrade subscription plan for users Query

Data Output	Messages	Notifications
UPDATE 2034		
	Query returned successfully in 81 msec.	

Fig. 17. Update to total number of rows in the table

8. Insert a new audio feature entry for an existing track.


```
1  ✓ INSERT INTO AudioFeatures(TrackId, Tempo, KeySignature, Danceability)
2    SELECT TrackId, 120, 'C', 0.75
3    FROM Track
4   WHERE Name = 'Score.'
5  LIMIT 1;
6
7  select * from audiofeatures where keysignature = 'C' and danceability = 0.75 and tempo = 120
```


Fig. 18. SQL Query to Insert a new audio feature entry


Data Output


Messages


Notifications














SQL






	featureid  integer [PK]	trackid  integer	tempo  integer	keysignature  text	danceability  numeric (3,2)	
1	5001	2046	120	C	0.75	

Fig. 19. Output of Insert new audio feature Query

9. Inserting a New Playlist

```
Query  Query History
1  ✓  INSERT INTO Playlist (Name)
2     VALUES ('Road Trip Songs');
3
4     select * from playlist where name = 'Road Trip Songs'
```

Fig. 20. SQL Query for Inserting a new Playlist

Data Output

Messages

Notifications

≡

📄

▼

📋

▼

🗑️

🗄️

⬇️

📈

SQL

	playlistid [PK] integer	name text
1	501	Road Trip Songs

Fig. 21. Output of Inserting new Playlist Query

10. Delete ad impressions that earned revenue less than 3.

Query Query History

```
1  ✓ DELETE FROM AdImpressions
2    WHERE RevenueGenerated < 4.00;
3
4
```

Fig. 22. SQL Query to delete ad impressions that earned revenue less than 3

Data Output Messages Notifications

DELETE 1033

Query returned successfully in 85 msec.

Fig. 23. Output of delete query changes in the table

11. Delete playlists that do not contain any tracks.

Query Query History

```
1  DELETE FROM Playlist
2  WHERE PlaylistId NOT IN (
3      SELECT DISTINCT PlaylistId
4      FROM PlaylistTrack
5  );
```

Fig. 24. SQL Query to delete playlists that do not contain any tracks.

Data Output **Messages** Notifications

DELETE 1

Query returned successfully in 80 msec.

Fig. 25. Output of delete query changes in the table

8 Query Execution Analysis

1. Top 5 Genres by Total Streams

```
Query Query History
1  EXPLAIN ANALYZE
2  SELECT g.Name, COUNT(lh.TrackId) AS StreamCount
3  FROM Genre g
4  JOIN Track t ON g.GenreId = t.GenreId
5  JOIN ListeningHistory lh ON t.TrackId = lh.TrackId
6  GROUP BY g.Name
7  ORDER BY StreamCount DESC
8  LIMIT 5;
```

Fig. 26. SQL Query for top 5 Genres by total streams

	QUERY PLAN text
1	Limit (cost=488.01.488.02 rows=5 width=40) (actual time=9.852..9.855 rows=5 loops=1)
2	-> Sort (cost=488.01.488.51 rows=200 width=40) (actual time=9.849..9.851 rows=5 loops=1)
3	Sort Key: (count(lh.trackid)) DESC
4	Sort Method: quicksort Memory: 25kB
5	-> HashAggregate (cost=482.69.484.69 rows=200 width=40) (actual time=9.808..9.813 rows=10 loops=1)
6	Group Key: g.name
7	Batches: 1 Memory Usage: 40kB
8	-> Hash Join (cost=202.07.432.69 rows=10000 width=36) (actual time=1.911..7.854 rows=10000 loops=1)
9	Hash Cond: (t.genreid = g.genreid)
10	-> Hash Join (cost=163.50.367.77 rows=10000 width=8) (actual time=1.805..6.135 rows=10000 loops=1)
11	Hash Cond: (lh.trackid = t.trackid)
12	-> Seq Scan on listeninghistory lh (cost=0.00.178.00 rows=10000 width=4) (actual time=0.043..1.391 row...
13	-> Hash (cost=101.00.101.00 rows=5000 width=8) (actual time=1.705..1.706 rows=5000 loops=1)
14	Buckets: 8192 Batches: 1 Memory Usage: 260kB
15	-> Seq Scan on track t (cost=0.00.101.00 rows=5000 width=8) (actual time=0.041..1.029 rows=5000 lo...
16	-> Hash (cost=22.70.22.70 rows=1270 width=36) (actual time=0.057..0.058 rows=10 loops=1)
17	Buckets: 2048 Batches: 1 Memory Usage: 17kB
18	-> Seq Scan on genre g (cost=0.00.22.70 rows=1270 width=36) (actual time=0.044..0.046 rows=10 loops...
19	Planning Time: 1.148 ms
20	Execution Time: 10.176 ms

Fig. 27. Before Performing Indexing

	QUERY PLAN text
1	Limit (cost=488.01..488.02 rows=5 width=40) (actual time=8.760..8.764 rows=5 loops=1)
2	-> Sort (cost=488.01..488.51 rows=200 width=40) (actual time=8.758..8.761 rows=5 loops=1)
3	Sort Key: (count(lh.trackid)) DESC
4	Sort Method: quicksort Memory: 25kB
5	-> HashAggregate (cost=482.69..484.69 rows=200 width=40) (actual time=8.730..8.736 rows=10 loops=1)
6	Group Key: g.name
7	Batches: 1 Memory Usage: 40kB
8	-> Hash Join (cost=202.07..432.69 rows=10000 width=36) (actual time=1.492..6.606 rows=10000 loops=1)
9	Hash Cond: (l.genreid = g.genreid)
10	-> Hash Join (cost=163.50..367.77 rows=10000 width=8) (actual time=1.449..4.820 rows=10000 loops=1)
11	Hash Cond: (lh.trackid = t.trackid)
12	-> Seq Scan on listeninghistory lh (cost=0.00..178.00 rows=10000 width=4) (actual time=0.009..0.749 row...
13	-> Hash (cost=101.00..101.00 rows=5000 width=8) (actual time=1.366..1.367 rows=5000 loops=1)
14	Buckets: 8192 Batches: 1 Memory Usage: 260kB
15	-> Seq Scan on track t (cost=0.00..101.00 rows=5000 width=8) (actual time=0.011..0.675 rows=5000 lo...
16	-> Hash (cost=22.70..22.70 rows=1270 width=36) (actual time=0.029..0.029 rows=10 loops=1)
17	Buckets: 2048 Batches: 1 Memory Usage: 17kB
18	-> Seq Scan on genre g (cost=0.00..22.70 rows=1270 width=36) (actual time=0.019..0.021 rows=10 loops...
19	Planning Time: 9.381 ms
20	Execution Time: 8.942 ms

Fig. 28. After performing Indexing

Before indexing, the query plan showed sequential scans on the Genre, Track, and ListeningHistory tables, with hash joins used to join them. The estimated cost was 488.01..488.02 and the execution time was approximately 10.176 ms. After creating indexes on Track(GenreId) and ListeningHistory(TrackId), the structure of the plan remained the same, still using sequential scans and hash joins, with a slightly improved execution time of 8.942 ms. This minor improvement is expected since the dataset is relatively small; however, the indexes lay the groundwork for performance gains at scale. As the number of records increases, PostgreSQL is likely to switch to index-based scans and joins, significantly reducing query cost and improving execution speed.

2. Customers Who Spent More Than \$500

```
Query    Query History
1  EXPLAIN ANALYZE
2  SELECT c.CustomerId, c.FirstName, c.LastName, SUM(i.Total) AS TotalSpent
3  FROM Customer c
4  JOIN Invoice i ON c.CustomerId = i.CustomerId
5  GROUP BY c.CustomerId, c.FirstName, c.LastName
6  HAVING SUM(i.Total) > 500;
```

Fig. 29. SQL Query for customers who spent more than 500\$

Data Output		Messages	Notifications
<div> </div>			
<div> <div>QUERY PLAN</div> <div>text</div> <div></div> </div>			
1	HashAggregate (cost=419.90..494.90 rows=1667 width=50) (actual time=20.982..20.986 rows=0 loops=1)		
2	Group Key: c.customerid		
3	Filter: (sum(i.total) > '500':numeric)		
4	Batches: 1 Memory Usage: 1489kB		
5	Rows Removed by Filter: 3176		
6	-> Hash Join (cost=251.77..394.90 rows=5000 width=24) (actual time=7.185..16.269 rows=5000 loops=1)		
7	Hash Cond: (i.customerid = c.customerid)		
8	-> Seq Scan on invoice i (cost=0.00..130.00 rows=5000 width=10) (actual time=0.284..3.528 rows=5000 loops=1)		
9	-> Hash (cost=185.23..185.23 rows=5323 width=18) (actual time=6.837..6.838 rows=5000 loops=1)		
10	Buckets: 8192 Batches: 1 Memory Usage: 309kB		
11	-> Seq Scan on customer c (cost=0.00..185.23 rows=5323 width=18) (actual time=0.032..5.806 rows=5000 loops=1)		
12	Planning Time: 8.365 ms		
13	Execution Time: 21.441 ms		

Fig. 30. Before Performing Indexing

Data Output		Messages	Notifications
<div> </div>			
	QUERY PLAN text		
1	HashAggregate (cost=419.90..494.90 rows=1667 width=50) (actual time=8.489..8.491 rows=0 loops=1)		
2	Group Key: c.customerid		
3	Filter: (sum(i.total) > '500':numeric)		
4	Batches: 1 Memory Usage: 1489kB		
5	Rows Removed by Filter: 3176		
6	-> Hash Join (cost=251.77..394.90 rows=5000 width=24) (actual time=2.619..4.913 rows=5000 loops=1)		
7	Hash Cond: (i.customerid = c.customerid)		
8	-> Seq Scan on Invoice i (cost=0.00..130.00 rows=5000 width=10) (actual time=0.026..0.368 rows=5000 loops=1)		
9	-> Hash (cost=185.23..185.23 rows=5323 width=18) (actual time=2.515..2.515 rows=5000 loops=1)		
10	Buckets: 8192 Batches: 1 Memory Usage: 309kB		
11	-> Seq Scan on customer c (cost=0.00..185.23 rows=5323 width=18) (actual time=0.025..1.280 rows=5000 loops=1)		
12	Planning Time: 4.758 ms		
13	Execution Time: 9.168 ms		

Fig. 31. After performing Indexing

Before indexing, the query to retrieve customers who spent more than \$500 used sequential scans on both the Invoice and Customer tables and relied on a hash join to connect them. The estimated cost was 419.90..494.90, and the execution time was around 21.44 ms. After creating an index on Invoice(CustomerId) and running ANALYZE, the planner still chose sequential scans but significantly improved performance, reducing the execution time to 9.17 ms, a 57% improvement. Although the execution plan structure remained the same, the lower runtime suggests better caching and improved memory usage during the join and aggregation stages. As the invoice

table grows, the index is expected to become more valuable, prompting PostgreSQL to switch to index-based join strategies, which will further improve performance.

3. Top 10 Most Active Customers by Unique Tracks Streamed in the Last 30 Days.

```
Query QueryHistory
1 v EXPLAIN ANALYZE
2 SELECT c.CustomerId, c.FirstName, c.LastName, COUNT(DISTINCT lh.TrackId) AS UniqueTracks
3 FROM Customer c
4 JOIN ListeningHistory lh ON c.CustomerId = lh.CustomerId
5 WHERE lh.ListenDate >= NOW() - INTERVAL '30 days'
6 GROUP BY c.CustomerId, c.FirstName, c.LastName
7 ORDER BY UniqueTracks DESC
8 LIMIT 10;
```

Fig. 32. SQL Query for top 10 most active customers by unique tracks

Data Output	Messages	Notifications
SQL		Sho
QUERY PLAN		
text		
1	Limit (cost=751.89..751.91 rows=10 width=26) (actual time=10.874..10.877 rows=10 loops=1)	
2	-> Sort (cost=751.89..758.18 rows=2516 width=26) (actual time=10.872..10.875 rows=10 loops=1)	
3	Sort Key: (count(DISTINCT lh.trackid)) DESC	
4	Sort Method: top-N heapsort Memory: 25kB	
5	-> GroupAggregate (cost=653.49..697.52 rows=2516 width=26) (actual time=9.684..10.563 rows=1975 loops=1)	
6	Group Key: c.customerid	
7	-> Sort (cost=653.49..659.78 rows=2516 width=22) (actual time=9.676..9.783 rows=2524 loops=1)	
8	Sort Key: c.customerid, lh.trackid	
9	Sort Method: quicksort Memory: 198kB	
10	-> Hash Join (cost=251.77..511.37 rows=2516 width=22) (actual time=2.228..8.501 rows=2524 loops=1)	
11	Hash Cond: (lh.customerid = c.customerid)	
12	-> Seq Scan on listeninghistory lh (cost=0.00..253.00 rows=2516 width=8) (actual time=0.035..5.107 ro...	
13	Filter: (listendate >= (now() - '30 days'::interval))	
14	Rows Removed by Filter: 7476	
15	-> Hash (cost=185.23..185.23 rows=5323 width=18) (actual time=2.145..2.146 rows=5000 loops=1)	
16	Buckets: 8192 Batches: 1 Memory Usage: 309kB	
17	-> Seq Scan on customer c (cost=0.00..185.23 rows=5323 width=18) (actual time=0.011..1.039 rows...	
18	Planning Time: 3.743 ms	
19	Execution Time: 11.001 ms	

Fig. 34. After performing Indexing

Before indexing, the query to retrieve the top 10 most active customers by unique tracks streamed in the last 30 days had a high estimated cost of 751.89 and an execution time of 16.97 ms, with sequential scans on ListeningHistory and Customer, along with expensive sort and aggregation operations due to COUNT(DISTINCT ...). After creating a composite index on (CustomerId, TrackId) and an index on ListenDate, the execution time improved significantly to 11.00 ms (35% faster), while the plan structure remained largely similar. PostgreSQL still used a hash join and a group aggregate, but the faster filtering and join performance indicate the index is reducing the overhead in processing recent activity. This improvement demonstrates how composite indexes can boost performance in multi-step analytical queries, especially as data volume scales.

9 Challenges faced while handling larger datasets.

Yes, we encountered several challenges while handling larger datasets, especially during query execution involving joins, aggregations, and time-based filters. As the volume of records increased, we observed a noticeable slowdown in performance, particularly with queries that lacked appropriate indexes. For example, the query to retrieve the top genres by stream count initially relied on full table scans and hash joins, leading to higher execution costs and longer run-times. Similarly, queries filtering by dates or calculating distinct counts became inefficient as PostgreSQL defaulted to sequential scans. To resolve these issues, we adopted indexing strategies, including creating indexes on join keys (CustomerId, TrackId), filter columns (ListenDate), and even composite indexes where necessary. After implementing and analyzing these indexes using EXPLAIN ANALYZE, we saw consistent improvements in execution time—up to 35–57% in some cases. These optimizations demonstrated the importance of aligning database indexing with query patterns for scalable performance.

Data Output	Messages	Notifications
SQL		Sho
QUERY PLAN		
text		
1	Limit (cost=751.89..751.91 rows=10 width=26) (actual time=16.049..16.053 rows=10 loops=1)	
2	-> Sort (cost=751.89..758.18 rows=2516 width=26) (actual time=16.038..16.041 rows=10 loops=1)	
3	Sort Key: (count(DISTINCT lh.trackid)) DESC	
4	Sort Method: top-N heapsort Memory: 25kB	
5	-> GroupAggregate (cost=653.49..697.52 rows=2516 width=26) (actual time=13.814..15.204 rows=1975 loops=1)	
6	Group Key: c.customerid	
7	-> Sort (cost=653.49..659.78 rows=2516 width=22) (actual time=13.793..13.966 rows=2524 loops=1)	
8	Sort Key: c.customerid, lh.trackid	
9	Sort Method: quicksort Memory: 198kB	
10	-> Hash Join (cost=251.77..511.37 rows=2516 width=22) (actual time=3.890..12.362 rows=2524 loops=1)	
11	Hash Cond: (lh.customerid = c.customerid)	
12	-> Seq Scan on listeninghistory lh (cost=0.00..253.00 rows=2516 width=8) (actual time=0.058..7.351 ro...	
13	Filter: (listendate >= (now() - '30 days'::interval))	
14	Rows Removed by Filter: 7476	
15	-> Hash (cost=185.23..185.23 rows=5323 width=18) (actual time=3.659..3.659 rows=5000 loops=1)	
16	Buckets: 8192 Batches: 1 Memory Usage: 309kB	
17	-> Seq Scan on customer c (cost=0.00..185.23 rows=5323 width=18) (actual time=0.024..1.786 rows...	
18	Planning Time: 3.699 ms	
19	Execution Time: 16.971 ms	

Fig. 33. Before Performing Indexing