

Universidad del Valle de Guatemala  
Bases de Datos - CC3057  
Sección 10

# **MODELACIÓN BASE DE DATOS: PROYECTO 2**

## **Autores:**

Julio Roberto Herrera Sabán	19402
Martín Eduardo España Rivera	19258
Fernando José Garavito Ovando	18071

## Diccionario de entidades

### Diccionario de tablas y campos

1. Account
  - a. username (varchar 20)
  - b. password (varchar 20) (not null)
  - c. first\_name (text) (not null)
  - d. last\_name (text)
  - e. email (varchar 50) (not null) (unique)
  - f. active (boolean) (not null) (default TRUE)
2. Subscription
  - a. id (serial)
  - b. start\_date (date) (not null) (default current\_date)
  - c. renewal\_date (date) (not null)
  - d. username (varchar 20) (foreign key) (not null) (unique)
3. Manager
  - a. id (serial)
  - b. username (varchar 20) (foreign key) (not null) (unique)
4. Artist
  - a. id (varchar 50)
  - b. artistic\_name (varchar 50) (not null)
  - c. username (varchar 20) (foreign key) (not null)
  - d. active (boolean) (not null) (default TRUE)
5. Album
  - a. id (varchar 50)
  - b. name (varchar 50) (not null)
  - c. preview\_url (text)
  - d. launch\_date (date) (not null) (default current\_date)
  - e. active (boolean) (not null) (default TRUE)
6. Album\_Artist
  - a. album\_id (varchar 50) (foreign key) (not null)
  - b. artist\_id (varchar 50) (foreign key) (not null)
  - c. primary key (album\_id, artist\_id)
7. Song
  - a. id (varchar 50)
  - b. name (text) (not null)
  - c. duration\_ms (int) (not null)
  - d. active (boolean) (not null) (default TRUE)
  - e. preview\_url (text)
  - f. album\_id (varchar 50) (foreign key)
8. Genre
  - a. id (serial)
  - b. name (text) (not null) (unique)

9. Song\_Genre

- a. song\_id (varchar 50) (foreign key) (not null)
- b. genre\_id (int) (foreign key) (not null)
- c. primary key (song\_id, genre\_id)

10.Reproduction

- a. id (serial)
- b. rep\_date (date) (not null) (default current\_date)
- c. song\_id (varchar 50) (foreign key) (not null)
- d. username (varchar 20) (foreign key) (not null)

11.Song\_Artist

- a. song\_id (varchar 50) (foreign key) (not null)
- b. artist\_id (varchar 50) (foreign key) (not null)
- c. primary key (song\_id, artist\_id)

12.Playlist (públicas)

- a. id (serial)
- b. name (text) (not null)
- c. description (text)
- d. date\_creation (date) (not null) (default current\_date)
- e. username (varchar 20) (foreign key) (not null)

13.Playlist\_Song

- a. playlist\_id (int) (foreign key) (not null)
- b. song\_id (varchar 50) (foreign key) (not null)
- c. primary key (playlist\_id, song\_id)

14.Playlist\_Account

- a. playlist\_id (int) (foreign key) (not null)
- b. username (varchar 20) (foreign key) (not null)
- c. primary key (playlist\_id, username)

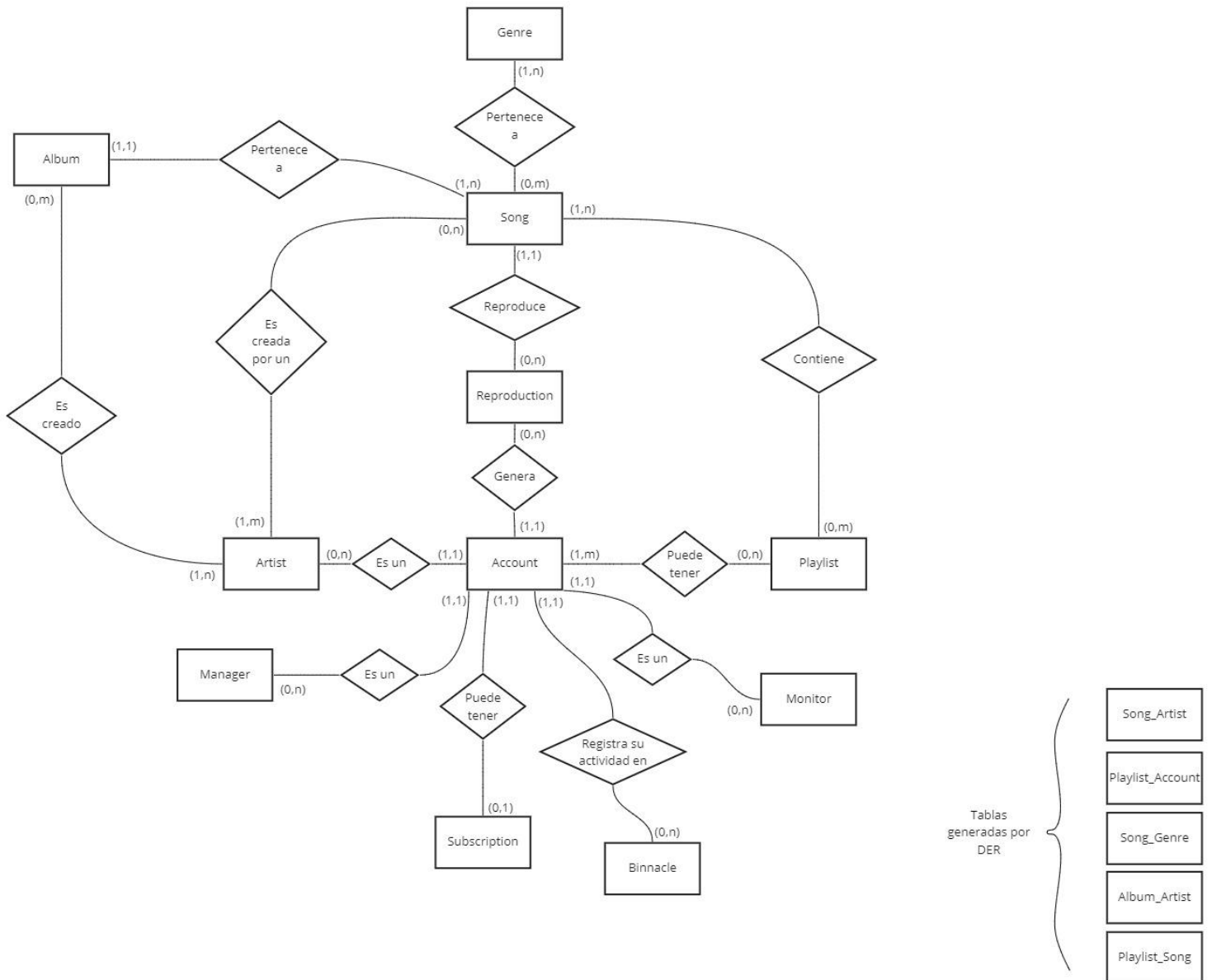
15.Monitor

- a. id (serial)
- b. username (varchar 20) (not null) (foreign key) (unique)
- c. type (varchar 20) (not null)

16.Binnacle

- a. id (serial)(primary key)(not null)
- b. author (varchar 20) (foreign key) (not null)
- c. table\_affected (varchar 20) (not null)
- d. action (text) (not null)
- e. record\_date (timestamp with timezone) (not null) (default current\_timestamp)

## Diagrama Entidad-Relación



## Esquema fundamentado de cálculo de comisiones de artista

Para calcular las comisiones de cada artista, se utilizó como base el modelo de retribución de Spotify, en el que se pagan entre \$0.003 y \$0.005 por reproducción. Es decir, alrededor de 250 reproducciones para obtener 1 dólar. Por lo tanto, en esta solución se planteó que cada reproducción generaría \$0.009 de los cuales se les paga a los artistas \$0.003 y a la aplicación \$0.006 (Jacob, 2021).

### Selección de índices

Primero se observan los índices actuales con los que cuenta la base de datos para tener una idea de las tablas y columnas que están haciendo uso de índices. Con la instrucción:

```
SELECT tablename, indexname, indexdef
FROM pg_indexes WHERE schemaname = 'public';
```

	tablename name	indexname name	indexdef text
1	account	account_pkey	CREATE UNIQUE INDEX account_pkey ON public.account USING btree (username)
2	account	account_email_key	CREATE UNIQUE INDEX account_email_key ON public.account USING btree (email)
3	subscription	subscription_pkey	CREATE UNIQUE INDEX subscription_pkey ON public.subscription USING btree (id)
4	subscription	subscription_username_key	CREATE UNIQUE INDEX subscription_username_key ON public.subscription USING btree (username)
5	manager	manager_pkey	CREATE UNIQUE INDEX manager_pkey ON public.manager USING btree (id)
6	manager	manager_username_key	CREATE UNIQUE INDEX manager_username_key ON public.manager USING btree (username)
7	artist	artist_pkey	CREATE UNIQUE INDEX artist_pkey ON public.artist USING btree (id)
8	album	album_pkey	CREATE UNIQUE INDEX album_pkey ON public.album USING btree (id)
9	album_artist	album_artist_pkey	CREATE UNIQUE INDEX album_artist_pkey ON public.album_artist USING btree (album_id, artist_id)
10	song	song_pkey	CREATE UNIQUE INDEX song_pkey ON public.song USING btree (id)
11	genre	genre_pkey	CREATE UNIQUE INDEX genre_pkey ON public.genre USING btree (id)
12	genre	genre_name_key	CREATE UNIQUE INDEX genre_name_key ON public.genre USING btree (name)
13	song_genre	song_genre_pkey	CREATE UNIQUE INDEX song_genre_pkey ON public.song_genre USING btree (song_id, genre_id)
14	reproduction	reproduction_pkey	CREATE UNIQUE INDEX reproduction_pkey ON public.reproduction USING btree (id)
15	song_artist	song_artist_pkey	CREATE UNIQUE INDEX song_artist_pkey ON public.song_artist USING btree (song_id, artist_id)
16	playlist	playlist_pkey	CREATE UNIQUE INDEX playlist_pkey ON public.playlist USING btree (id)
17	playlist_song	playlist_song_pkey	CREATE UNIQUE INDEX playlist_song_pkey ON public.playlist_song USING btree (playlist_id, song_id)
18	playlist_account	playlist_account_pkey	CREATE UNIQUE INDEX playlist_account_pkey ON public.playlist_account USING btree (playlist_id, username)
19	monitor	monitor_pkey	CREATE UNIQUE INDEX monitor_pkey ON public.monitor USING btree (id)
20	binnacle	binnacle_pkey	CREATE UNIQUE INDEX binnacle_pkey ON public.binnacle USING btree (id)

### Índice de bitácora

La ejecución del query utilizado por el API para mostrar la bitácora, muestra un tiempo aproximado de 55 msec en su ejecución.

```
spofity/postgres@PostgreSQL 13
Query Editor
1  --INDICES--
2  SELECT * FROM Binnacle ORDER BY record_date DESC

Messages
Successfully run. Total query runtime: 55 msec.
104 rows affected.
```

Y muestra un escaneo secuencial

```
spofity/postgres@PostgreSQL 13
Query Editor
1  --INDICES--
2  EXPLAIN ANALYZE SELECT * FROM Binnacle ORDER BY record_date DESC

Messages
Successfully run. Total query runtime: 187 msec.
6 rows affected.
```

	QUERY PLAN
1	Sort (cost=6.52..6.78 rows=104 width=60) (actual time=0.065..0.067 rows=104 loops=1)
2	Sort Key: record_date DESC
3	Sort Method: quicksort Memory: 39kB
4	-> Seq Scan on binnacle (cost=0.00..3.04 rows=104 width=60) (actual time=0.023..0.029 rows=104 loops=1)
5	Planning Time: 0.082 ms
6	Execution Time: 13.913 ms

Para mejorar el desempeño de este query, se crea un índice para la tabla *Binnacle*, en la columna *record\_date* que es la columna que se ve involucrada en la ejecución de este query.

```
CREATE INDEX idx_binnacle_record_date ON Binnacle(record_date DESC);
```

Ahora volvemos a comprobar, sin embargo observamos que el análisis sigue mostrando que se ejecuta un escaneo secuencial.

Data Output	Query History	Explain	Notifications
<b>QUERY PLAN</b> text			
1	Sort (cost=6.52..6.78 rows=104 width=60) (actual time=0.038..0.041 rows=104 loops=1)		
2	Sort Key: record_date DESC		
3	Sort Method: quicksort Memory: 39kB		
4	-> Seq Scan on binnacle (cost=0.00..3.04 rows=104 width=60) (actual time=0.013..0.020 rows=104 loops=1)		
5	Planning Time: 0.063 ms		
6	Execution Time: 0.055 ms		

Esto se debe a que PostgreSQL sigue utilizando *sort*, ya que el *query planner* considera que es el mejor método para ejecutar este query. Para utilizar sí o sí nuestro índice podemos desactivar el uso de *sort* con (PostgreSQL, Sin Fecha):  
SET enable\_sort=on

Ahora analizamos nuevamente el query.

QUERY PLAN

text

1

Index Scan using idx\_binnacle\_record\_date on binnacle (cost=0.14..14.71 rows=104 width=60) (actual time=0.010..0.021 rows=104 loops=1)

2

Planning Time: 0.060 ms

3

Execution Time: 0.033 ms

4

SELECT \* FROM Binnacle ORDER BY record\_date DESC

5

Messages

Successfully run. Total query runtime: 51 msec.  
104 rows affected.

Data Output

Query History

Explain

Notifications

	id [PK] integer	author character varying (20)	table_affected character varying (20)	action text	record_date timestamp with time zone
1	108	newprueba	Artist	Actualizac...	2021-05-12 06:13:28.403616+00
2	106	newprueba2	Subscription	Eliminació...	2021-05-12 06:12:25.444683+00
3	107	newprueba2	Subscription	Creación ...	2021-05-12 06:12:25.444683+00
4	105	newprueba	Artist	Actualizac...	2021-05-12 05:07:00.538423+00
5	104	newprueba	Artist	Actualizac...	2021-05-12 05:06:20.806536+00

Vemos que el nuevo tiempo de ejecución es levemente menor, sin embargo el costo estimado por el análisis del *query plan* utilizando el índice es mayor, por algo el *query planner* en un inicio seguía utilizando el método *sort*, sin embargo la diferencia entre el uso del índice podría ser notable al haber gran cantidad de registros en un futuro, por lo que se dejará esta configuración.

## Índice de reportes

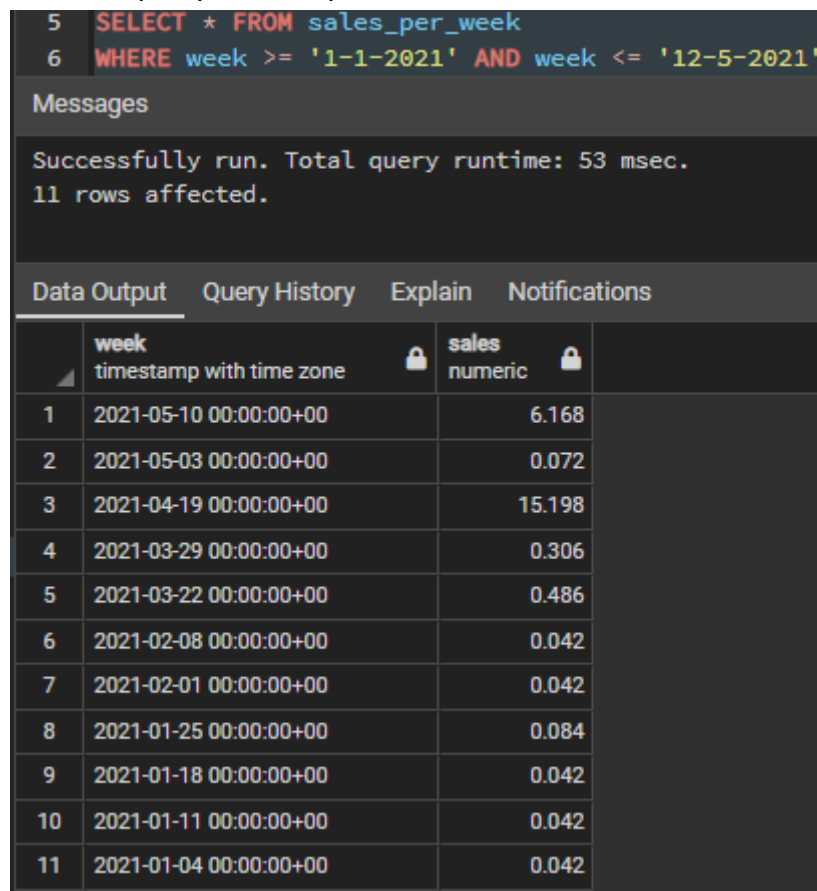
### Total de ventas por semana dado un rango de fechas

El query que se utiliza para obtener esta estadística hace uso de la vista *sales\_per\_week*.

```
SELECT * FROM sales_per_week
WHERE week >= '${dates.startDate}' AND week <= '${dates.endDate}'
```

Donde *dates.startDate* y *dates.endDate* son fechas en formato DD-MM-YYYY.

La ejecución de este query tarda aproximadamente 53 msec.



The screenshot shows a database query execution interface. At the top, the query is displayed: `SELECT * FROM sales_per_week WHERE week >= '1-1-2021' AND week <= '12-5-2021'`. Below the query, a 'Messages' section indicates 'Successfully run. Total query runtime: 53 msec. 11 rows affected.' At the bottom, a 'Data Output' table shows the results of the query.

	week timestamp with time zone	sales numeric
1	2021-05-10 00:00:00+00	6.168
2	2021-05-03 00:00:00+00	0.072
3	2021-04-19 00:00:00+00	15.198
4	2021-03-29 00:00:00+00	0.306
5	2021-03-22 00:00:00+00	0.486
6	2021-02-08 00:00:00+00	0.042
7	2021-02-01 00:00:00+00	0.042
8	2021-01-25 00:00:00+00	0.084
9	2021-01-18 00:00:00+00	0.042
10	2021-01-11 00:00:00+00	0.042
11	2021-01-04 00:00:00+00	0.042

Y el análisis del *query plan* indica que se están utilizando dos escaneos secuenciales, uno en la tabla *Subscription* y otro en la tabla *Reproduction*.



	QUERY PLAN
1	Sort (cost=10000000040.37..10000000040.38 rows=5 width=40) (actual time=0.349..0.350 rows=11 loops=1)
2	Sort Key: (date_trunc('week':text, (reproduction.rep_date):timestamp with time zone)) DESC
3	Sort Method: quicksort Memory: 25kB
4	-> HashAggregate (cost=40.25..40.31 rows=5 width=40) (actual time=0.340..0.343 rows=11 loops=1)
5	Group Key: (date_trunc('week':text, (reproduction.rep_date):timestamp with time zone))
6	Batches: 1 Memory Usage: 24kB
7	-> HashAggregate (cost=40.13..40.17 rows=5 width=40) (actual time=0.332..0.335 rows=13 loops=1)
8	Group Key: (date_trunc('week':text, (reproduction.rep_date):timestamp with time zone)), (((count(*)):numeric * 0.006))
9	Batches: 1 Memory Usage: 24kB
10	-> Append (cost=13.35..40.10 rows=5 width=40) (actual time=0.306..0.328 rows=13 loops=1)
11	-> HashAggregate (cost=13.35..13.39 rows=2 width=40) (actual time=0.306..0.309 rows=11 loops=1)
12	Group Key: date_trunc('week':text, (reproduction.rep_date):timestamp with time zone)
13	Batches: 1 Memory Usage: 24kB
14	-> Seq Scan on reproduction (cost=0.00..13.34 rows=2 width=8) (actual time=0.019..0.268 rows=254 loops=1)
15	Filter: ((date_trunc('week':text, (rep_date):timestamp with time zone) >= '2021-01-01 00:00:00':timestamp with time zone) AND (date_trunc('week':text, (rep_date):timestamp with time zone) <= '2021-05-12 00:00:00':timestamp with time zone))
16	Rows Removed by Filter: 162
17	-> Subquery Scan on "SELECT* 2" (cost=26.53..26.65 rows=3 width=40) (actual time=0.017..0.017 rows=2 loops=1)
18	-> HashAggregate (cost=26.53..26.58 rows=3 width=16) (actual time=0.017..0.017 rows=2 loops=1)
19	Group Key: date_trunc('week':text, (subscription.start_date):timestamp with time zone)
20	Batches: 1 Memory Usage: 24kB
21	-> Seq Scan on subscription (cost=0.00..26.52 rows=3 width=8) (actual time=0.009..0.014 rows=7 loops=1)
22	Filter: ((date_trunc('week':text, (start_date):timestamp with time zone) >= '2021-01-01 00:00:00':timestamp with time zone) AND (date_trunc('week':text, (start_date):timestamp with time zone) <= '2021-05-12 00:00:00':timestamp with time zone))
23	Planning Time: 0.223 ms
24	Execution Time: 0.413 ms

Para reemplazar estos escaneos secuenciales, se crearán los siguientes índices:

```
CREATE INDEX idx_subscription_start_date ON
```

```
Subscription(start_date);
```

```
CREATE INDEX idx_reproduction_rep_date ON Reproduction(rep_date);
```

Sin embargo, de nuevo se volvió a notar que el *query planner* no hacía uso de estos índices, por lo que se desactivo la configuración de utilizar escaneo secuencial, con (PostgreSQL, Sin Fecha):

```
SET enable_seqscan=off
```

Por lo que el resultado del *query plan* haciendo uso de los índices es:

	QUERY PLAN
1	Sort (cost=10000000037.29..10000000037.29 rows=3 width=40) (actual time=0.588..0.591 rows=11 loops=1)
2	Sort Key: (date_trunc('week':text, (reproduction.rep_date):timestamp with time zone)) DESC
3	Sort Method: quicksort Memory: 25kB
4	-> HashAggregate (cost=37.23..37.26 rows=3 width=40) (actual time=0.575..0.580 rows=11 loops=1)
5	Group Key: (date_trunc('week':text, (reproduction.rep_date):timestamp with time zone))
6	Batches: 1 Memory Usage: 24kB
7	-> HashAggregate (cost=37.15..37.18 rows=3 width=40) (actual time=0.558..0.562 rows=13 loops=1)
8	Group Key: (date_trunc('week':text, (reproduction.rep_date):timestamp with time zone)), (((count(*)):numeric * 0.006))
9	Batches: 1 Memory Usage: 24kB
10	-> Append (cost=24.65..37.14 rows=3 width=40) (actual time=0.517..0.530 rows=13 loops=1)
11	-> HashAggregate (cost=24.65..24.69 rows=2 width=40) (actual time=0.517..0.522 rows=11 loops=1)
12	Group Key: date_trunc('week':text, (reproduction.rep_date):timestamp with time zone)
13	Batches: 1 Memory Usage: 24kB
14	-> Bitmap Heap Scan on reproduction (cost=10.23..24.64 rows=2 width=8) (actual time=0.041..0.430 rows=254 loops=1)
15	Filter: ((date_trunc('week':text, (rep_date):timestamp with time zone) >= '2021-01-01 00:00:00':timestamp with time zone) AND (date_trunc('week':text, (rep_date):timestamp with time zone) <= '2021-05-12 00:00:00':timestamp with time zone))
16	Rows Removed by Filter: 162
17	Heap Blocks: exact=4
18	-> Bitmap Index Scan on idx_reproduction_rep_date (cost=0.00..10.23 rows=416 width=0) (actual time=0.026..0.026 rows=416 loops=1)
19	-> Subquery Scan on "SELECT* 2" (cost=12.35..12.39 rows=1 width=40) (actual time=0.025..0.026 rows=2 loops=1)
20	-> HashAggregate (cost=12.35..12.37 rows=1 width=16) (actual time=0.024..0.025 rows=2 loops=1)
21	Group Key: date_trunc('week':text, (subscription.start_date):timestamp with time zone)
22	Batches: 1 Memory Usage: 24kB
23	-> Index Only Scan using idx_subscription_start_date on subscription (cost=0.13..12.35 rows=1 width=8) (actual time=0.011..0.020 rows=7 loops=1)
24	Filter: ((date_trunc('week':text, (start_date):timestamp with time zone) >= '2021-01-01 00:00:00':timestamp with time zone) AND (date_trunc('week':text, (start_date):timestamp with time zone) <= '2021-05-12 00:00:00':timestamp with time zone))
25	Heap Fetches: 7
26	Planning Time: 0.275 ms
27	Execution Time: 0.683 ms

Query Editor

```
6 SELECT * FROM sales_per_week
7 WHERE week >= '1-1-2021' AND week <= '12-5-2021'
8
9 EXPLAIN ANALYZE SELECT * FROM sales_per_week
```

Messages

Successfully run. Total query runtime: 48 msec.  
11 rows affected.

Data Output

Query History

Explain

Notifications

	week timestamp with time zone	sales numeric	
1	2021-05-10 00:00:00+00	6.168	
2	2021-05-03 00:00:00+00	0.072	
3	2021-04-19 00:00:00+00	15.198	
4	2021-03-29 00:00:00+00	0.306	
5	2021-03-22 00:00:00+00	0.486	
6	2021-02-08 00:00:00+00	0.042	
7	2021-02-01 00:00:00+00	0.042	
8	2021-01-25 00:00:00+00	0.084	
9	2021-01-18 00:00:00+00	0.042	
10	2021-01-11 00:00:00+00	0.042	
11	2021-01-04 00:00:00+00	0.042	

El costo se redujo principalmente en la tabla *Subscription*, pero el tiempo de ejecución no tuvo una gran diferencia, esto puede deberse igualmente a que no se cuentan con suficientes datos para notar el potencial de uso de índices.

### Los N artistas con mayores ventas para un rango de fechas

El query utilizado en el API para obtener esta estadística es:

```
SELECT id, artistic_name, artist_id, SUM(sales) AS sales
FROM artist_sales_per_date X
WHERE rep_date >= '${dates.startDate}' AND rep_date <= '${dates.endDate}'
GROUP BY id, artistic_name, artist_id ORDER BY sales DESC LIMIT ${limit}
```

Donde dates.startDate y dates.endDate son fechas en formato DD-MM-YYYY y limit es un número entero positivo.

La ejecución de este query tarda aproximadamente 56 msec.

spotify/postgres@PostgreSQL 13 ▾

Query Editor

```

6 SELECT id, artistic_name, artist_id, SUM(sales) AS sales
7 FROM artist_sales_per_date X
8 WHERE rep_date >= '1-1-2021' AND rep_date <= '12-5-2021'
9 GROUP BY id, artistic_name, artist_id ORDER BY sales DESC LIMIT 10
10
11 EXPLAIN ANALYZE SELECT id, artistic_name, artist_id, SUM(sales) AS sales

```

Messages

Successfully run. Total query runtime: 56 msec.  
10 rows affected.

Data Output Query History Explain Notifications

	id character varying (50)	artistic_name character varying (50)	artist_id character varying (50)	sales numeric
1	7dGJo4pcD2V6oG8kP0tJRR	Eminem	7dGJo4pcD2V6oG8kP0tJRR	0.321
2	3YQKmKGau1PzVlkL1iodx	Twenty One Pilots	3YQKmKGau1PzVlkL1iodx	0.192
3	6C1ohJrd5VydigQtaGy5Wa	Joyner Lucas	6C1ohJrd5VydigQtaGy5Wa	0.156
4	2DuJi13MWHjRHrqRUwk8vH	Knife Party	2DuJi13MWHjRHrqRUwk8vH	0.054
5	0ZCO8oVkJmJ897cKgFH7fRW	Los Ángeles Azules	0ZCO8oVkJmJ897cKgFH7fRW	0.054
6	4gzpq5DPGxSnKTe4SA8HAU	Coldplay	4gzpq5DPGxSnKTe4SA8HAU	0.036
7	2ye2Wgw4gimLv2eAKyk1NB	Metallica	2ye2Wgw4gimLv2eAKyk1NB	0.030
8	4q3ewBCX7sLwd24euvV69X	Bad Bunny	4q3ewBCX7sLwd24euvV69X	0.027
9	3KedxarmBCyFBevnqQH3P	Jessie Reyez	3KedxarmBCyFBevnqQH3P	0.024
10	6bwVWdQfr6ttzrGLboVkjg	Paul Rosenberg	6bwVWdQfr6ttzrGLboVkjg	0.015

Y su *query plan* indica que ya está haciendo uso de los índices predefinidos o de los índices creados por nosotros anteriormente, sin ningún escaneo secuencial y siendo el único proceso utilizando gran cantidad de costo, el último sort, sin embargo este no es posible de solucionar utilizando índices ya que este último método está trabajando sobre funciones de agregación.

	QUERY PLAN
	text
1	Limit (cost=10000000072.95..10000000072.97 rows=10 width=90) (actual time=0.600..0.605 rows=10 loops=1)
2	-> Sort (cost=10000000072.95..10000000073.17 rows=88 width=90) (actual time=0.599..0.603 rows=10 loops=1)
3	Sort Key: (sum((((count(r.id)::numeric * 0.003)))) DESC
4	Sort Method: top-N heapsort Memory: 27kB
5	-> HashAggregate (cost=69.95..71.05 rows=88 width=90) (actual time=0.549..0.570 rows=32 loops=1)
6	Group Key: a.id, sa.artist_id
7	Batches: 1 Memory Usage: 32kB
8	-> Hash Join (cost=59.35..69.29 rows=88 width=90) (actual time=0.393..0.492 rows=147 loops=1)
9	Hash Cond: ((sa.artist_id)::text = (a.id)::text)
10	-> HashAggregate (cost=45.86..51.24 rows=359 width=59) (actual time=0.319..0.379 rows=147 loops=1)
11	Group Key: sa.artist_id, r.rep_date
12	Batches: 1 Memory Usage: 61kB
13	-> Hash Join (cost=30.20..43.17 rows=359 width=31) (actual time=0.095..0.197 rows=348 loops=1)
14	Hash Cond: ((r.song_id)::text = (sa.song_id)::text)
15	-> Bitmap Heap Scan on reproduction r (cost=10.60..18.18 rows=239 width=30) (actual time=0.023..0.046 rows=257 loops=1)
16	Recheck Cond: ((rep_date >= '2021-01-01'::date) AND (rep_date <= '2021-05-12'::date))
17	Heap Blocks: exact=4
18	-> Bitmap Index Scan on idx_reproduction_rep_date (cost=0.00..10.54 rows=239 width=0) (actual time=0.014..0.015 rows=...
19	Index Cond: ((rep_date >= '2021-01-01'::date) AND (rep_date <= '2021-05-12'::date))
20	-> Hash (cost=18.02..18.02 rows=127 width=46) (actual time=0.060..0.061 rows=131 loops=1)
21	Buckets: 1024 Batches: 1 Memory Usage: 18kB
22	-> Index Only Scan using song_artist_pkey on song_artist sa (cost=0.14..18.02 rows=127 width=46) (actual time=0.012..0.0...
23	Heap Fetches: 131
24	-> Hash (cost=12.88..12.88 rows=49 width=35) (actual time=0.063..0.063 rows=77 loops=1)
25	Buckets: 1024 Batches: 1 Memory Usage: 13kB
26	-> Index Scan using artist_pkey on artist a (cost=0.14..12.88 rows=49 width=35) (actual time=0.015..0.035 rows=77 loops=1)
27	Planning Time: 0.668 ms

## Total de ventas por género para un rango de fechas

El query utilizado en el API para obtener esta estadística es:

```
SELECT id, name, SUM(sales) AS sales
FROM genre_sales_per_date
WHERE date >= '${dates.startDate}' AND date <= '${dates.endDate}'
GROUP BY id, name ORDER BY sales DESC
```

Donde dates.startDate y dates.endDate son fechas en formato DD-MM-YYYY.

El tiempo de ejecución de este query es de aproximadamente 54 msec.

6

SELECT id, name, SUM(sales) AS sales

7

FROM genre\_sales\_per\_date

8

WHERE date >= '1-1-2021' AND date <= '12-5-2021'

9

GROUP BY id, name ORDER BY sales DESC

Messages

Successfully run. Total query runtime: 54 msec.  
23 rows affected.

Data Output

Query History

Explain

Notifications

	<div>id</div> <div>integer</div>	<div>name</div> <div>text</div>	<div>sales</div> <div>numeric</div>
1	1	Hip Hop	1.647
2	3	Rap	1.071
3	4	Trap	1.008
4	2	Hardco...	1.008
5	5	Pop	0.684
6	20	Rock	0.666
7	19	Reggae	0.576
8	18	Pop In...	0.576
9	16	Hip Ho...	0.576

Y su *query plan* indica igualmente que ya está haciendo uso de los índices existentes y no ejecuta ningún escaneo secuencial, siendo el último *sort* el método con mayor coste pero realizado sobre funciones de agregación, por lo que este query no necesita nuevos índices.

	QUERY PLAN
	text
1	Sort (cost=10000000202.20..10000000202.70 rows=200 width=68) (actual time=0.911..0.914 rows=23 loops=1)
2	Sort Key: (sum((((count(r.id)::numeric * 0.009)))) DESC
3	Sort Method: quicksort Memory: 26kB
4	-> HashAggregate (cost=192.05..194.55 rows=200 width=68) (actual time=0.889..0.897 rows=23 loops=1)
5	Group Key: g.id, g.name
6	Batches: 1 Memory Usage: 40kB
7	-> HashAggregate (cost=162.02..175.88 rows=924 width=72) (actual time=0.737..0.828 rows=278 loops=1)
8	Group Key: g.id, r.rep_date
9	Batches: 1 Memory Usage: 97kB
10	-> Hash Join (cost=135.58..155.09 rows=924 width=44) (actual time=0.333..0.534 rows=1099 loops=1)
11	Hash Cond: ((r.song_id)::text = (sg.song_id)::text)
12	-> Bitmap Heap Scan on reproduction r (cost=10.60..18.18 rows=239 width=30) (actual time=0.027..0.056 rows=257 loops=1)
13	Recheck Cond: ((rep_date >= '2021-01-01'::date) AND (rep_date <= '2021-05-12'::date))
14	Heap Blocks: exact=4
15	-> Bitmap Index Scan on idx_reproduction_rep_date (cost=0.00..10.54 rows=239 width=0) (actual time=0.019..0.019 rows=257 lo...
16	Index Cond: ((rep_date >= '2021-01-01'::date) AND (rep_date <= '2021-05-12'::date))
17	-> Hash (cost=120.95..120.95 rows=323 width=59) (actual time=0.297..0.298 rows=325 loops=1)
18	Buckets: 1024 Batches: 1 Memory Usage: 31kB
19	-> Hash Join (cost=83.35..120.95 rows=323 width=59) (actual time=0.039..0.213 rows=325 loops=1)
20	Hash Cond: (sg.genre_id = g.id)
21	-> Index Only Scan using song_genre_pkey on song_genre sg (cost=0.27..37.02 rows=323 width=27) (actual time=0.012..0.11...
22	Heap Fetches: 325
23	-> Hash (cost=67.20..67.20 rows=1270 width=36) (actual time=0.017..0.017 rows=23 loops=1)
24	Buckets: 2048 Batches: 1 Memory Usage: 18kB
25	-> Index Scan using genre_pkey on genre g (cost=0.15..67.20 rows=1270 width=36) (actual time=0.006..0.009 rows=23 loo...
26	Planning Time: 0.842 ms
27	Execution Time: 1.080 ms

## Las N canciones con más reproducciones para un artista

El query utilizado en el API para obtener esta estadística es:

```
SELECT id, name, duration_ms, active, cover, album, album_id, count,
       (SELECT ARRAY_AGG(ROW_TO_JSON(X)) FROM
        (SELECT artist_id, artistic_name FROM Song_Artist SA INNER JOIN Artist A
        ON SA.artist_id = A.id WHERE SA.song_id = B.id) X) AS artists
FROM song_reproduction_count_by_artist B
WHERE B.artist_id = '${artist_id}' ORDER BY count DESC LIMIT ${limit}
```

Donde artist\_id es el id del artista y limit es un número entero positivo.

La ejecución de este query tarda aproximadamente 57 msec.

spotify/postgres@PostgreSQL 13

Query Editor

```
6 SELECT id, name, duration_ms, active, cover, album, album_id, count,
7       (SELECT ARRAY_AGG(ROW_TO_JSON(X)) FROM
8       (SELECT artist_id, artistic_name FROM Song_Artist SA INNER JOIN Artist A
9       ON SA.artist_id = A.id WHERE SA.song_id = B.id) X) AS artists
10      FROM song_reproduction_count_by_artist B
11      WHERE B.artist_id = '7dGJo4pcD2V6oG8kP0tJRR' ORDER BY count DESC LIMIT 10
```

Messages

Successfully run. Total query runtime: 57 msec.  
10 rows affected.

Data Output

Query History

Explain

Notifications

	id	name	duration_ms	active	cover	album	album_id	count	artists
	character varying (50)	text	integer	boolean	text	character varying (50)	character varying (50)	bigint	json[]
1	60SdxE8apGAxMiRpbmLY0	Lucky ...	244679	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	101	{("artist_i...
2	2XTquzYQAdT1Hk78bOUwsv	Greatest	226937	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	24	{("artist_i...
3	4g32MdRoqwgKQd3NXIRhpU	Good G...	142192	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	10	{("artist_i...
4	6B3zy3LOKHndqsviCr2z15	Em Cal...	49023	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	7	{("artist_i...
5	0evt4UZbdhnHtcAnxkm6A1	Normal	222477	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	6	{("artist_i...
6	28FGV3ORH14MYORd7s5dlU	Not Ali...	288086	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	5	{("artist_i...
7	2gsNpSn7VvvJuSriDfRoYy	Kamik...	216029	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	3	{("artist_i...
8	58QhkbAklFnn7JwAnAato	Fall	262493	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	2	{("artist_i...
9	2SL6oP2YAEQbqsrk0zRG04	Venom...	269554	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	2	{("artist_i...
10	60Z57Wdm0pEvPH7d7GAX	Steppi...	309637	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	2	{("artist_i...

Y su *query plan* indica que hace uso de ciertos índices, sin embargo cuenta con un proceso de escaneo secuencial, que en realidad no estima tanto costo, pero de igual forma lo buscaremos reemplazar por un índice.

	QUERY PLAN
	text
1	Limit (cost=20000000145.70..200000000312.38 rows=10 width=257) (actual time=3.049..3.142 rows=10 loops=1)
2	-> Result (cost=20000000145.70..20000000762.42 rows=37 width=257) (actual time=3.048..3.140 rows=10 loops=1)
3	-> Sort (cost=20000000145.70..20000000145.79 rows=37 width=225) (actual time=2.997..3.001 rows=10 loops=1)
4	Sort Key: b.count DESC
5	Sort Method: quicksort Memory: 27kB
6	-> Subquery Scan on b (cost=10000000004.41..10000000144.90 rows=37 width=225) (actual time=0.953..2.983 rows=11 loops=1)
7	-> GroupAggregate (cost=10000000004.41..10000000144.53 rows=37 width=349) (actual time=0.952..2.976 rows=11 loops=1)
8	Group Key: a.id, s.id, alb.name, alb.preview_url
9	-> Incremental Sort (cost=10000000004.41..10000000143.70 rows=37 width=345) (actual time=0.941..2.884 rows=163 loops=1)
10	Sort Key: s.id, alb.name, alb.preview_url
11	Presorted Key: s.id
12	Full-sort Groups: 3 Sort Method: quicksort Average Memory: 46kB Peak Memory: 46kB
13	Pre-sorted Groups: 3 Sort Method: quicksort Average Memory: 37kB Peak Memory: 53kB
14	-> Nested Loop (cost=10000000000.57..10000000142.03 rows=37 width=345) (actual time=0.322..2.484 rows=163 loops=1)
15	-> Nested Loop (cost=10000000000.43..10000000122.87 rows=37 width=195) (actual time=0.309..1.822 rows=163 loops=1)
16	-> Nested Loop (cost=10000000000.29..10000000114.24 rows=37 width=196) (actual time=0.291..1.684 rows=163 loops=1)
17	-> Nested Loop (cost=10000000000.14..10000000101.25 rows=59 width=72) (actual time=0.276..0.893 rows=163 loops=1)
18	Join Filter: ((r.song_id)::text = (sa.song_id)::text)
19	Rows Removed by Join Filter: 4829
20	-> Index Only Scan using song_artist_pkey on song_artist sa (cost=0.14..17.17 rows=12 width=46) (actual time=0.276..0.893 rows=163 loops=1)
21	Index Cond: (artist_id = '7dGJo4pcD2V6oG8kP0tJRR')::text)
22	Heap Fetches: 12



23	-> Materialize (cost=10000000000.00..10000000010.24 rows=416 width=26) (actual time=0.003..0.032 rows=...
24	-> Seq Scan on reproduction r (cost=10000000000.00..10000000008.16 rows=416 width=26) (actual time=...
25	-> Index Scan using song_pkey on song s (cost=0.14..0.22 rows=1 width=169) (actual time=0.004..0.004 rows=1...
26	Index Cond: ((id)::text = (r.song_id)::text)
27	-> Materialize (cost=0.14..8.16 rows=1 width=22) (actual time=0.000..0.000 rows=1 loops=163)
28	-> Index Only Scan using artist_pkey on artist a (cost=0.14..8.16 rows=1 width=22) (actual time=0.015..0.017 ro...
29	Index Cond: (id = '7dGJo4pcD2V6oG8kP0tJRR'::text)
30	Heap Fetches: 1
31	-> Index Scan using album_pkey on album alb (cost=0.15..0.52 rows=1 width=268) (actual time=0.003..0.003 rows=1 lo...
32	Index Cond: ((id)::text = (s.album_id)::text)
33	SubPlan 1
34	-> Aggregate (cost=16.65..16.66 rows=1 width=32) (actual time=0.013..0.013 rows=1 loops=10)
35	-> Nested Loop (cost=0.28..16.64 rows=1 width=36) (actual time=0.008..0.009 rows=1 loops=10)
36	-> Index Only Scan using song_artist_pkey on song_artist sa_1 (cost=0.14..8.16 rows=1 width=23) (actual time=0.004..0.004 row...
37	Index Cond: (song_id = (b.id)::text)
38	Heap Fetches: 13
39	-> Index Scan using artist_pkey on artist a_1 (cost=0.14..8.16 rows=1 width=35) (actual time=0.003..0.003 rows=1 loops=13)
40	Index Cond: ((id)::text = (sa_1.artist_id)::text)
41	Planning Time: 1.647 ms
42	Execution Time: 3.337 ms

El índice creado es:

```
CREATE INDEX idx_reproduction_song_id ON Reproduction(song_id);
```


Por lo que el *query plan* ahora muestra que usa índices y ningún escaneo secuencial.

	QUERY PLAN text
1	Limit (cost=10000000059.26..10000000225.94 rows=10 width=257) (actual time=0.749..0.923 rows=10 loops=1)
2	-> Result (cost=10000000059.26..10000000675.99 rows=37 width=257) (actual time=0.748..0.919 rows=10 loops=1)
3	-> Sort (cost=10000000059.26..10000000059.36 rows=37 width=225) (actual time=0.621..0.631 rows=10 loops=1)
4	Sort Key: b.count DESC
5	Sort Method: quicksort Memory: 27kB
6	-> Subquery Scan on b (cost=57.72..58.46 rows=37 width=225) (actual time=0.597..0.610 rows=11 loops=1)
7	-> HashAggregate (cost=57.72..58.09 rows=37 width=349) (actual time=0.596..0.607 rows=11 loops=1)
8	Group Key: a.id, s.id, alb.name, alb.preview_url
9	Batches: 1 Memory Usage: 32kB
10	-> Nested Loop (cost=9.68..57.26 rows=37 width=345) (actual time=0.121..0.469 rows=163 loops=1)
11	-> Nested Loop (cost=9.53..51.72 rows=12 width=364) (actual time=0.110..0.288 rows=12 loops=1)
12	-> Nested Loop (cost=9.38..45.50 rows=12 width=214) (actual time=0.090..0.208 rows=12 loops=1)
13	-> Index Only Scan using artist_pkey on artist a (cost=0.14..8.16 rows=1 width=22) (actual time=0.019..0.021 rows=1 l...
14	Index Cond: (id = '7dGJo4pcD2V6oG8kP0tJRR'::text)
15	Heap Fetches: 1
16	-> Nested Loop (cost=9.24..37.23 rows=12 width=215) (actual time=0.045..0.156 rows=12 loops=1)
17	-> Bitmap Heap Scan on song_artist sa (cost=9.10..11.25 rows=12 width=46) (actual time=0.028..0.032 rows=12 l...
18	Recheck Cond: ((artist_id)::text = '7dGJo4pcD2V6oG8kP0tJRR'::text)
19	Heap Blocks: exact=1
20	-> Bitmap Index Scan on song_artist_pkey (cost=0.00..9.10 rows=12 width=0) (actual time=0.010..0.010 rows=...
21	Index Cond: ((artist_id)::text = '7dGJo4pcD2V6oG8kP0tJRR'::text)
22	-> Index Scan using song_pkey on song s (cost=0.14..2.16 rows=1 width=169) (actual time=0.007..0.007 rows=1 lo...



23	Index Cond: ((id)::text = (sa.song_id)::text)
24	-> Index Scan using album_pkey on album alb (cost=0.15..0.52 rows=1 width=268) (actual time=0.005..0.005 rows=1 loo...
25	Index Cond: ((id)::text = (s.album_id)::text)
26	-> Index Scan using idx_reproduction_song_id on reproduction r (cost=0.15..0.41 rows=5 width=26) (actual time=0.006..0.01...
27	Index Cond: ((song_id)::text = (s.id)::text)
28	SubPlan 1
29	-> Aggregate (cost=16.65..16.66 rows=1 width=32) (actual time=0.026..0.026 rows=1 loops=10)
30	-> Nested Loop (cost=0.28..16.64 rows=1 width=36) (actual time=0.017..0.020 rows=1 loops=10)
31	-> Index Only Scan using song_artist_pkey on song_artist sa_1 (cost=0.14..8.16 rows=1 width=23) (actual time=0.010..0.011 rows...
32	Index Cond: (song_id = (b.id)::text)
33	Heap Fetches: 13
34	-> Index Scan using artist_pkey on artist a_1 (cost=0.14..8.16 rows=1 width=35) (actual time=0.005..0.005 rows=1 loops=13)
35	Index Cond: ((id)::text = (sa_1.artist_id)::text)
36	Planning Time: 3.744 ms
37	Execution Time: 1.119 ms

Y su tiempo aproximado de ejecución es de 54 msec, el cual nuevamente, no representa una gran diferencia respecto al original, pero la reducción deseada podría darse al contar con una gran cantidad de registros en la base de datos que aprovechen de mejor manera el uso de índices.


spotify/postgres@PostgreSQL 13 ▾

Query Editor

```

5
6 SELECT id, name, duration_ms, active, cover, album, album_id, count,
7       (SELECT ARRAY_AGG(ROW_TO_JSON(X)) FROM
8       (SELECT artist_id, artistic_name FROM Song_Artist SA INNER JOIN Artist A
9       ON SA.artist_id = A.id WHERE SA.song_id = B.id) X) AS artists
10      FROM song_reproduction_count_by_artist B
11      WHERE B.artist_id = '7dGJo4pcD2V6oG8kP0tJRR' ORDER BY count DESC LIMIT 10
12

```

Messages

Successfully run. Total query runtime: 54 msec.  
10 rows affected.

Data Output

Query History

Explain

Notifications

	id	name	duration_ms	active	cover	album	album_id	count	artists
	character varying (50)	text	integer	boolean	text	character varying (50)	character varying (50)	bigint	json[]
1	60SdxE8apGAXmIRpblmLY0	Lucky ...	244679	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	101	{("artist_i...
2	2XTquzYQAdT1Hk78bOUwsv	Greatest	226937	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	24	{("artist_i...
3	4g32MdRoqwkGKQd3NXIRhpU	Good G...	142192	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	10	{("artist_i...
4	6B3zy3LOKHndqsviCr2z15	Em Cal...	49023	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	7	{("artist_i...
5	0evt4UZbdhnHtcAnxkm6A1	Normal	222477	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	6	{("artist_i...
6	28FGV3ORH14MYORd7s5dlU	Not Ali...	288086	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	5	{("artist_i...
7	2gsNpSn7VvwJuSrIdfRoYy	Kamik...	216029	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	3	{("artist_i...
8	58QhkbAklFnn7JwAnAato	Fall	262493	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	2	{("artist_i...
9	60Z57WdmOpEVPhI7d7GAX	Steppi...	309637	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	2	{("artist_i...
10	2SL6oP2YAEQbqsrkOzRG04	Venom...	269554	true	https://i...	Kamikaze	3HNnxK7NgLXbDoxRZxNWIR	2	{("artist_i...

## Bibliografía

Jacob, E. (2021, 26 de febrero). ¿Cuánto dinero paga Spotify por cada reproducción? Esto es lo que ganan los artistas con sus canciones.  
Extraído de:

<https://www.businessinsider.es/cuanto-dinero-paga-spotify-cada-reproduccion-como-ganar-818251>

The PostgreSQL Global Development Group. (Sin Fecha). PostgreSQL 13 Documentation: 19.7 Query Planning.

<https://www.postgresql.org/docs/9.1/runtime-config-query.html>