

Manual de Optimización de Rendimiento - Proyecto Vibesia

Proyecto: Vibesia

Versión: 1.0

Fecha: 18/06/2025

Equipo: Ad Astra

Elaborado por:

- Oscar Alejandro Prasca Chacón
- Carlos Julio Vergel Wilches
- Karen Silvana Duque Leal
- Duvan Arley Ramírez Duran
- Juan David Jaimes Rojas

Manual de Optimización de Rendimiento - Proyecto Vibesia	1
1. Introducción	2
2. Objetivo	2
3. Tipos de Índices Implementados	2
3.1 Índices Funcionales	2
3.2 Índices Compuestos.....	2
3.3 Índices Parciales	3
3.4 Índices sobre Claves Foráneas Frecuentes	3
4. Validación con EXPLAIN ANALYZE.....	3
5. Recomendaciones Finales	3
6. Ejemplos:	3
6.1 Query 1 antes de los indices:	4
6.1.1 Query 1 despues de los indices:	4
6.2 Query 2 antes de los indices:	4
6.2.1 Query 2 despues de los indices:	5

6.3 Query 3 antes de los indices:	5
6.3.1 Query 3 despues de los indices:	5
6.4 Query 4 antes de los indices:	6
6.4.1 Query 3 despues de los indices:	6

1. Introducción

Este documento presenta las técnicas de optimización implementadas sobre la base de datos PostgreSQL del proyecto Vibesia. El objetivo es mejorar el rendimiento general y reducir el tiempo de respuesta de las consultas SQL ejecutadas por el backend y durante análisis complejos. Se hace especial énfasis en el uso de índices funcionales, compuestos y parciales sobre columnas clave.

2. Objetivo

Aplicar una estrategia avanzada de optimización basada en índices especializados, para mejorar el rendimiento de las operaciones típicas del sistema (autenticación, búsqueda, análisis por fecha) y consultas analíticas utilizadas en los informes avanzados de Vibesia.

3. Tipos de Índices Implementados

3.1 Índices Funcionales

Permiten búsquedas eficientes sin distinción entre mayúsculas y minúsculas. Apoyan la búsqueda en campos como nombre de usuario, nombre de artistas, títulos y géneros.

- `idx_users_username_lower`
- `idx_users_email_lower`
- `idx_artists_name_search`
- `idx_albums_title_search`
- `idx_songs_title_search`
- `idx_genres_name_lower`
- `idx_playlists_name_lower`

3.2 Índices Compuestos

Agrupar columnas comúnmente filtradas o agrupadas juntas en las consultas. Se aplican principalmente en ``playback_history`` y otras tablas relacionadas a reproducciones, artistas y playlists.

- idx_playback_history_user_date
- idx_playback_history_song_date
- idx_playback_history_device_date
- idx_playback_history_q1_filters
- idx_q1_grouping_helper
- idx_artists_country_name_perf
- idx_playlist_songs_playlist_date_added

3.3 Índices Parciales

Aplican únicamente a subconjuntos de datos con condiciones específicas. Mejoran el rendimiento sin indexar innecesariamente toda la tabla.

- idx_users_only_active (usuarios con is_active = `TRUE`)

3.4 Índices sobre Claves Foráneas Frecuentes

Índices creados sobre claves foráneas de uso común en joins complejos, para asegurar que las operaciones no se ralenticen al escalar la base de datos.

- idx_songs_album_id_fk
- idx_albums_artist_id_fk

4. Validación con EXPLAIN ANALYZE

Para validar que los índices están siendo usados correctamente, se puede ejecutar `EXPLAIN ANALYZE` antes de cualquier consulta crítica. PostgreSQL indicará si el índice fue utilizado en la estrategia de ejecución.

Ejemplo:

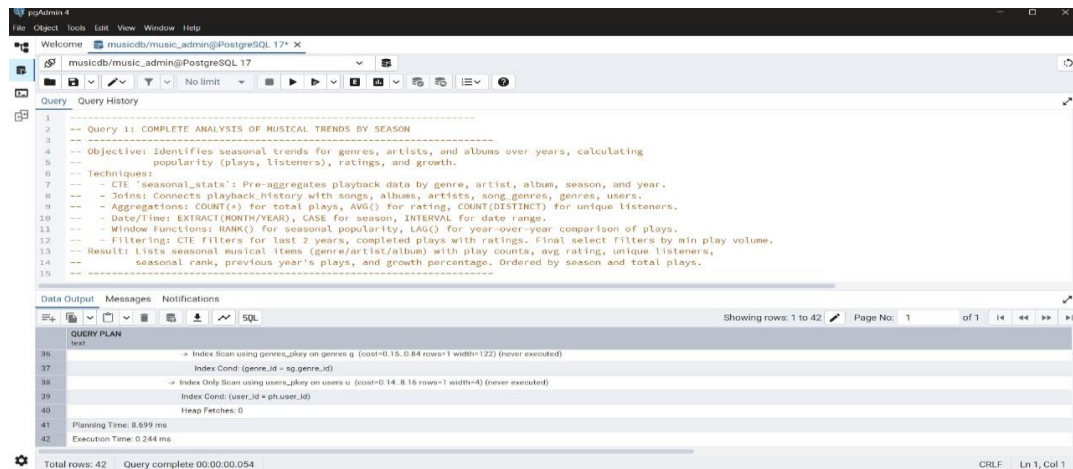
```
EXPLAIN ANALYZE SELECT * FROM vibesia_schema.playlists WHERE
LOWER(name) = 'favoritas';
```

5. Recomendaciones Finales

- Ejecutar este script luego de la creación completa del esquema y carga de datos.
- Evitar exceso de índices en tablas con muchas escrituras (puede afectar INSERT/UPDATE).
- Documentar y alinear con el backend si se usa Alembic para evitar conflictos de nombres.
- Verificar rendimiento con consultas reales y ajustar según el plan de ejecución.

6. Ejemplos:

6.1 Query 1 antes de los indices:



```
1 -- Query 1: COMPLETE ANALYSIS OF MUSICAL TRENDS BY SEASON
2 -----
3 -- Objective: Identifies seasonal trends for genres, artists, and albums over years, calculating
4 -- popularity (plays, listeners), ratings, and growth.
5 --
6 -- Techniques:
7 --   - CTE 'seasonal_stats': Pre-aggregates playback data by genre, artist, album, season, and year.
8 --   - Joins: Connects playback_history with songs, albums, artists, song_genres, genres, users.
9 --   - Aggregations: COUNT(*) for total plays, AVG() for rating, COUNT(DISTINCT) for unique listeners.
10 --   - Date/Time: EXTRACT(MONTH/YEAR), CASE for season, INTERVAL for date range.
11 --   - Window Functions: RANK() for seasonal popularity, LAG() for year-over-year comparison of plays.
12 --   - Filtering: CTE Filters for last 3 years, completed plays with ratings. Final select filters by min play volume.
13 -- Result: Lists seasonal musical items (genre/artist/album) with play counts, avg rating, unique listeners,
14 -- seasonal rank, previous year's plays, and growth percentage. Ordered by season and total plays.
15 -----
```

Showing rows: 1 to 42 | Page No: 1 of 1

QUERY PLAN

Text

Index Scan using genres_play on genres g (cost=0.15..0.84 rows=1 width=122) (never executed)

Index Cond: (genre_id = sg.genre_id)

Index Only Scan using users_play on users u (cost=0.14..8.16 rows=1 width=4) (never executed)

Index Cond: (user_id = ph.user_id)

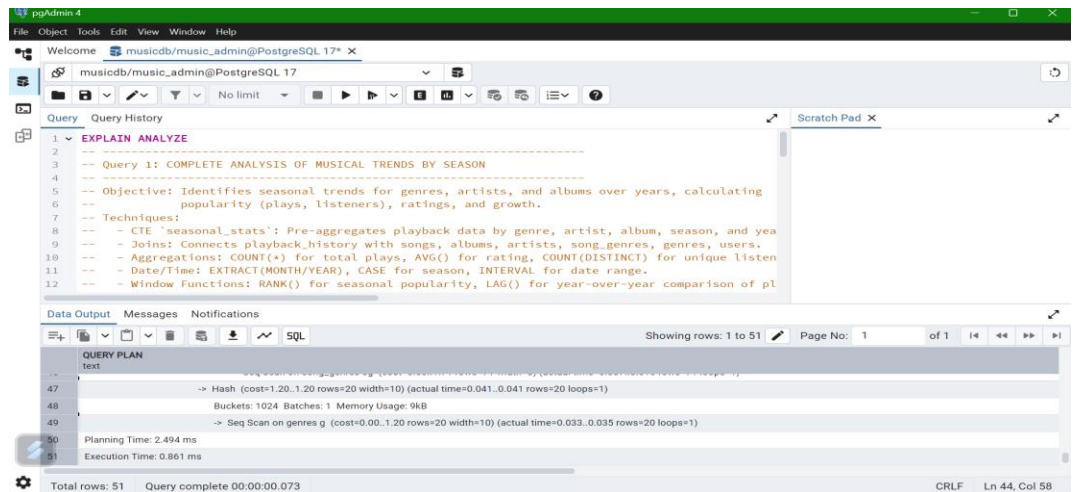
Heap Fetches: 0

Planning Time: 8.599 ms

Execution Time: 0.244 ms

Total rows: 42 | Query complete 00:00:00.054

6.1.1 Query 1 despues de los indices:



```
1 -- Query 1: COMPLETE ANALYSIS OF MUSICAL TRENDS BY SEASON
2 -----
3 -- Objective: Identifies seasonal trends for genres, artists, and albums over years, calculating
4 -- popularity (plays, listeners), ratings, and growth.
5 --
6 -- Techniques:
7 --   - CTE 'seasonal_stats': Pre-aggregates playback data by genre, artist, album, season, and year.
8 --   - Joins: Connects playback_history with songs, albums, artists, song_genres, genres, users.
9 --   - Aggregations: COUNT(*) for total plays, AVG() for rating, COUNT(DISTINCT) for unique listeners.
10 --   - Date/Time: EXTRACT(MONTH/YEAR), CASE for season, INTERVAL for date range.
11 --   - Window Functions: RANK() for seasonal popularity, LAG() for year-over-year comparison of plays.
12 -----
```

Showing rows: 1 to 51 | Page No: 1 of 1

QUERY PLAN

Text

Hash Join (cost=0.120..1.20 rows=20 width=10) (actual time=0.041..0.041 rows=20 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 9kB

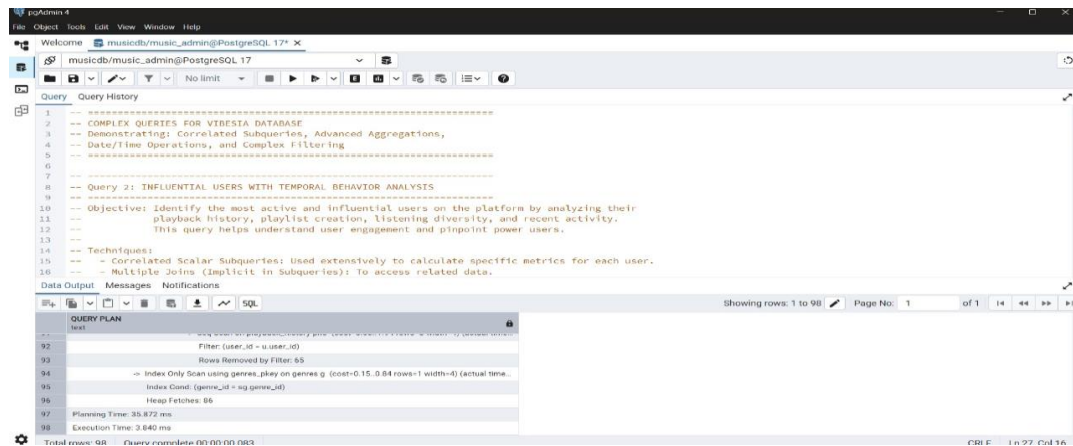
Seq Scan on genres g (cost=0.00..1.20 rows=20 width=10) (actual time=0.033..0.035 rows=20 loops=1)

Planning Time: 2.494 ms

Execution Time: 0.861 ms

Total rows: 51 | Query complete 00:00:00.073

6.2 Query 2 antes de los indices:



```
1 -- Query 2: INFLUENTIAL USERS WITH TEMPORAL BEHAVIOR ANALYSIS
2 -----
3 -- Objective: Identify the most active and influential users on the platform by analyzing their
4 -- playback history, playlist creation, listening diversity, and recent activity.
5 -- This query helps understand user engagement and pinpoint power users.
6 --
7 -- Techniques:
8 --   - Correlated Scalar Subqueries: Used extensively to calculate specific metrics for each user.
9 --   - Multiple Joins (Implicit in Subqueries): To access related data.
10 -----
```

Showing rows: 1 to 98 | Page No: 1 of 1

QUERY PLAN

Text

Filter: (user_id = u.user_id)

Rows Removed by Filter: 65

Index Only Scan using genres_play on genres g (cost=0.15..0.84 rows=1 width=4) (actual time=0.033..0.035 rows=1 loops=1)

Index Cond: (genre_id = sg.genre_id)

Heap Fetches: 66

Planning Time: 35.872 ms

Execution Time: 0.640 ms

Total rows: 98 | Query complete 00:00:00.083

6.2.1 Query 2 despues de los indices:

pgAdmin 4

Welcome musicdb/music_admin@PostgreSQL 17* X

musicdb/music_admin@PostgreSQL 17

Query Query History

Scratch Pad X

```
-- Query 2: INFLUENTIAL USERS WITH TEMPORAL BEHAVIOR ANALYSIS
--
-- Objective: Identify the most active and influential users on the platform by analyzing their
-- playback history, playlist creation, listening diversity, and recent activity.
-- This query helps understand user engagement and pinpoint power users.
--
-- Techniques:
-- - Correlated Scalar Subqueries: Used extensively to calculate specific metrics for each user.
-- - Multiple Joins (Implicit in Subqueries): To access related data.
-- - Complex Aggregations: COUNT(*), COUNT(DISTINCT), AVG.
-- - Date/Time Operations: (CURRENT_DATE - u.registration_date), INTERVAL, DATE_TRUNC.
-- - Filtering:
--   - 'u.is_active = TRUE': Selects only users marked as active.
```

Data Output Messages Notifications

Showing rows: 1 to 100 Page No: 1 of 1

QUERY PLAN

text

95 -> Seq Scan on songs s (cost=0.00..1.40 rows=40 width=4) (actual time=0.008..0.0...

96 -> Hash (cost=1.20..1.20 rows=20 width=4) (actual time=0.013..0.013 rows=20 loops=1)

97 Buckets: 1024 Batches: 1 Memory Usage: 9kB

98 -> Seq Scan on genres g (cost=0.00..1.20 rows=20 width=4) (actual time=0.008..0.009 rows=...

99 Planning Time: 6.230 ms

100 Execution Time: 4.112 ms

Total rows: 100 Query complete 00:00:00.089 CRLF Ln 65, Col 50

6.3 Query 3 antes de los indices:

pgAdmin 4

Welcome musicdb/music_admin@PostgreSQL 17* X

musicdb/music_admin@PostgreSQL 17

Query Query History

Data Output Messages Notifications

Showing rows: 1 to 71 Page No: 1 of 1

QUERY PLAN

text

55 -> Seq Scan on playback_history phr (cost=0.00..1.40 rows=40 width=4) (actual time=0.008..0.0...

56 -> Hash (cost=10.88..10.88 rows=1 width=4) (actual time=0.034..0.034 rows=2 loops=...

57 Buckets: 1024 Batches: 1 Memory Usage: 9kB

58 -> Seq Scan on songs s2,1 (cost=0.00..10.88 rows=1 width=4) (actual time=0.012...

59 Filter: (album_id = al.album_id)

60 Rows Removed by Filter: 38

70 Planning Time: 1.409 ms

71 Execution Time: 10.601 ms

Total rows: 71 Query complete 00:00:00.118 CRLF Ln 18, Col 16

6.3.1 Query 3 despues de los indices:

pgAdmin 4

File Object Tools Edit View Window Help

Welcome musicdb/music_admin@PostgreSQL 17*

musicdb/music_admin@PostgreSQL 17

Query Query History

1 -- EXPLAIN ANALYZE
 2 --
 3 -- Query 3: ALBUM COHESION AND DURATION ANALYSIS WITH ADVANCED METRICS
 4 --
 5 -- Objective: Evaluates albums on structure (song count, duration), popularity (plays, listeners),
 6 -- and cohesion (song duration consistency).
 7 -- Techniques:
 8 -- - CTE 'album_metrics': Calculates per-album metrics (song count, total/avg/stddev/min/max dur
 9 -- - Joins: Combines albums, artists, songs.
 10 -- - Correlated Subqueries (in CTE): Gathers album-specific play counts, unique listeners,
 11 -- and recent plays from 'playback_history'.
 12 -- - Aggregations: COUNT, SUM, AVG, STDDEV, MIN, MAX.

Data Output Messages Notifications

Showing rows: 1 to 71 Page No: 1 of 1

QUERY PLAN

Text

66 Buckets: 1024 Batches: 1 Memory Usage: 9kB
 67 -> Seq Scan on songs s2_1 (cost=0.00..1.50 rows=2 width=4) (actual time=0.003..0.003 rows=2)

68 Filter: (album_id = al.album_id)
 69 Rows Removed by Filter: 38

70 Planning Time: 2.113 ms
 71 Execution Time: 2.839 ms

Total rows: 71 Query complete 00:00:00.076 CRLF Ln 81, Col 70

6.4 Query 4 antes de los indices:

pgAdmin 4

File Object Tools Edit View Window Help

Welcome musicdb/music_admin@PostgreSQL 17*

musicdb/music_admin@PostgreSQL 17

Query Query History

1 --
 2 -- Query 4: LISTENING PATTERNS BY DEVICE AND TIME OF DAY
 3 --
 4 -- Objective: Analyze how listening patterns vary by device type, OS, time of day, and day of the week.
 5 -- Techniques: CTE, Joins, Date/Time EXTRACT, Aggregations (COUNT, AVG, CASE), Window Functions (SUM, ROW_NUMBER).
 6 -- Result: Detailed listening statistics segmented by device, OS, hour, and day.
 7 --
 8 -- EXPLAIN ANALYZE
 9 WITH hourly_device_stats AS (
 10 SELECT
 11 d.device_type,
 12 d.operating_system,
 13 EXTRACT(HOUR FROM ph.playback_date) AS hour_of_day,
 14 EXTRACT(DOW FROM ph.playback_date) AS day_of_week, -- @=Sunday, 6=Saturday
 15 COUNT(*) AS total_plays,

Data Output Messages Notifications

Showing rows: 1 to 24 Page No: 1 of 1

QUERY PLAN

Text

17 -- Index Scan using devices_play on devices d (cost=0.15..8.17 rows=1 width=200) (never executed)
 18 Rows Removed by Filter: 75
 19 -> Index Scan using devices_play on devices d (cost=0.15..8.17 rows=1 width=200) (never executed)
 20 Index Cond: (device_id = ph.device_id)
 21 -> Index Scan using songs_play on songs s (cost=0.14..8.16 rows=1 width=8) (never executed)
 22 Index Cond: (song_id = ph.song_id)
 23 Planning Time: 5.645 ms
 24 Execution Time: 0.291 ms

Total rows: 24 Query complete 00:00:00.116 CRLF Ln 8, Col 17

6.4.1 Query 3 despues de los indices:

pgAdmin 4

File Object Tools Edit View Window Help

Welcome musicdb/music_admin@PostgreSQL 17+ X

musicdb/music_admin@PostgreSQL 17

Query Query History No limit

Scratch Pad X

```
44 ROUND(completion_rate * 100, 2) AS completion_percentage,
45 ROUND(avg_song_duration / 60.0, 2) AS avg_duration_minutes,
46 high_ratings,
47 CASE
48   WHEN total_plays > 0 THEN ROUND(high_ratings::DECIMAL / total_plays * 100, 2)
49   ELSE 0
50 END AS high_rating_percentage, -- Avoid division by zero
51 ROUND(total_plays::DECIMAL / NULLIF(SUM(total_plays) OVER (PARTITION BY device_type, operating
52   ROW_NUMBER() OVER (PARTITION BY device_type, operating_system, day_of_week ORDER BY total_play
53 FROM hourly_device_stats
54 WHERE total_plays >= 2 -- MODIFIED: Filters combinations with at least 2 plays (Adjust from 10 for
55 ORDER BY device_type, operating_system, day_of_week, total_plays DESC;
```

Data Output Messages Notifications

Showing rows: 1 to 28 Page No: 1 of 1

QUERY PLAN

text

23 Buckets: 1024 Batches: 1 Memory Usage: 8kB

24 -> Seq Scan on playback_history ph (cost=0.00..2.31 rows=1 width=25) (actual t...

25 Filter: (playback_date >= (CURRENT_DATE - '6 mons'::interval))

26 Rows Removed by Filter: 75

27 Planning Time: 1.096 ms

28 Execution Time: 0.146 ms

Total rows: 28 Query complete 00:00:00.059 CRLF Ln 55, Col 71

Fin del manual