

Tarea 02 Data Management

Master of Data Science

Profesor: Miguel Romero

25-06-2022

Contents

Antecedentes	1
Parte 1	2
Parte 2	8

Antecedentes

Alumnos

- Leonardo Rojas
- Juan Pablo Espinoza
- Sebastian Vera

Parte 1

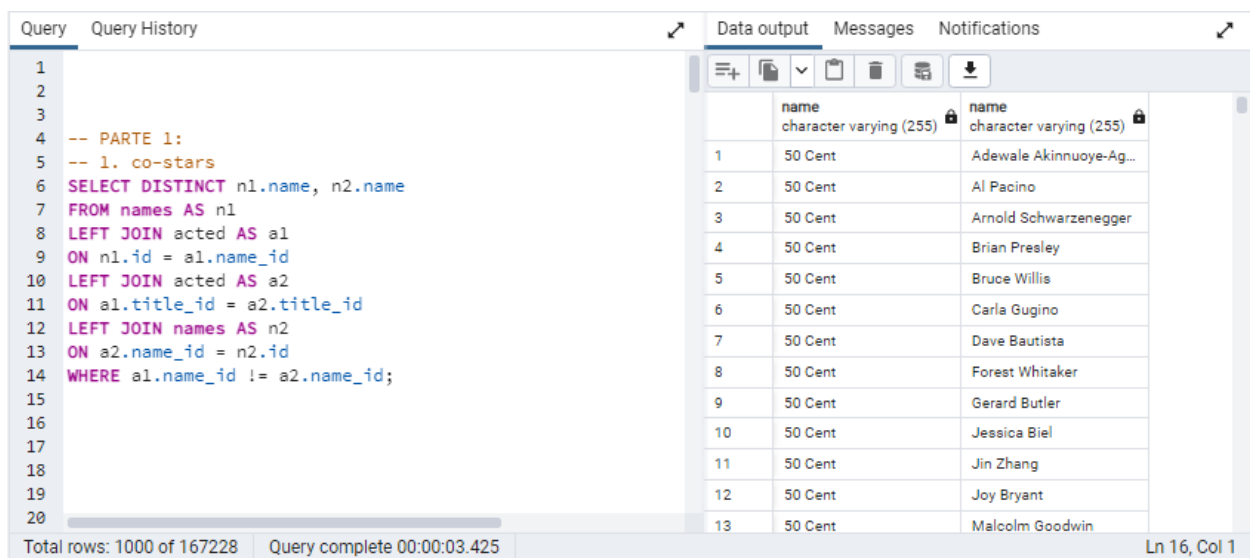
1. Co-stars

A continuación se muestra la query necesaria para construir una tabla de co-starts, parejas de actores que han actuado en una misma película:

```
-- 1. co-stars
SELECT DISTINCT n1.name, n2.name
FROM names AS n1
LEFT JOIN acted AS a1
ON n1.id = a1.name_id
LEFT JOIN acted AS a2
ON a1.title_id = a2.title_id
LEFT JOIN names AS n2
ON a2.name_id = n2.id
WHERE a1.name_id != a2.name_id;
```

La lógica de esta es comenzar por la tabla de nombres de actores (**names**) y cruzar los datos de las películas en las cuales ha actuado (**acted**). A partir de id de la película que ha actuado, se vuelve a cruzar los datos de la tabla **acted** para obtener una pareja de id de actores que actuaron en la misma película. Finalmente se cruza la información de nombre del actor secundario para proyectar sobre los nombres del actor primario y secundario.

El resultado de esta consulta se muestra a continuación:



The screenshot shows a database query interface with a query editor on the left and a data output window on the right. The query editor contains the following SQL code:

```
1
2
3
4 -- PARTE 1:
5 -- 1. co-stars
6 SELECT DISTINCT n1.name, n2.name
7 FROM names AS n1
8 LEFT JOIN acted AS a1
9 ON n1.id = a1.name_id
10 LEFT JOIN acted AS a2
11 ON a1.title_id = a2.title_id
12 LEFT JOIN names AS n2
13 ON a2.name_id = n2.id
14 WHERE a1.name_id != a2.name_id;
15
16
17
18
19
20
```

The data output window displays the results of the query. It shows a table with two columns: 'name' (character varying (255)) and 'name' (character varying (255)). The results are as follows:

	name	name
1	50 Cent	Adewale Akinnuoye-Ag...
2	50 Cent	Al Pacino
3	50 Cent	Arnold Schwarzenegger
4	50 Cent	Brian Presley
5	50 Cent	Bruce Willis
6	50 Cent	Carla Gugino
7	50 Cent	Dave Bautista
8	50 Cent	Forest Whitaker
9	50 Cent	Gerard Butler
10	50 Cent	Jessica Biel
11	50 Cent	Jin Zhang
12	50 Cent	Joy Bryant
13	50 Cent	Malcolm Goodwin

The status bar at the bottom indicates: Total rows: 1000 of 167228, Query complete 00:00:03.425, Ln 16, Col 1.

Figure 1: Screen 1.1: Tabla co_star

Se genera una tabla con 167.228 parejas de actores que han actuado en una misma película.

2. Co starts indirectos Kevin Bacon

Ahora construimos una consulta recursiva de co-stars indirectos del actor *Kevin Bacon*.

Para esto construimos una tabla temporal `co_star(name1, name2)` mediante la clausula `WITH` usando la lógica desarrollada en el punto (1).

Luego se genera una tabla recursiva a partir de `co_star` filtrando que el primer nombre sea *Kevin Bacon*, a esta tabla inicial se le une recursivamente la tabla de `co_star` cruzando por el co-star indirecto:

```
WITH RECURSIVE co_star(name1, name2) AS (  
    SELECT DISTINCT n1.name, n2.name  
    FROM names AS n1  
    LEFT JOIN acted AS a1  
    ON n1.id = a1.name_id  
    LEFT JOIN acted AS a2  
    ON a1.title_id = a2.title_id  
    LEFT JOIN names AS n2  
    ON a2.name_id = n2.id  
    WHERE a1.name_id != a2.name_id  
) ,  
indirect_co_star_bacon(name2) AS (  
    SELECT name2  
    FROM co_star  
    WHERE name1 = 'Kevin Bacon'  
    UNION  
    SELECT c2.name2  
    FROM indirect_co_star_bacon AS c1  
    LEFT JOIN co_star AS c2  
    ON c1.name2 = c2.name1  
) SELECT DISTINCT * FROM indirect_co_star_bacon;
```

Al ejecutar esta consulta obtenemos:

The screenshot shows a database query interface with a query editor on the left and a data output window on the right. The query editor contains a recursive SQL query to find indirect co-stars of Kevin Bacon. The data output window displays a table with 14 rows of results, showing the names of the actors found.

name2
Kylli Königs
David Kross
Jean Topart
Scott Glenn
Davi Santos
Riccardo Zinna
Elina Löwensohn
Steffen Zacharias
Max Pomeranc
Aydemir Akbas
Brian Van Holt
Roger Jackson
Edith Scob
Xzibit

Total rows: 1000 of 18752 Query complete 00:00:02.444 Ln 31, Col 53

Figure 2: Screen 1.2: Co-star indirecto de Kevin Bacon

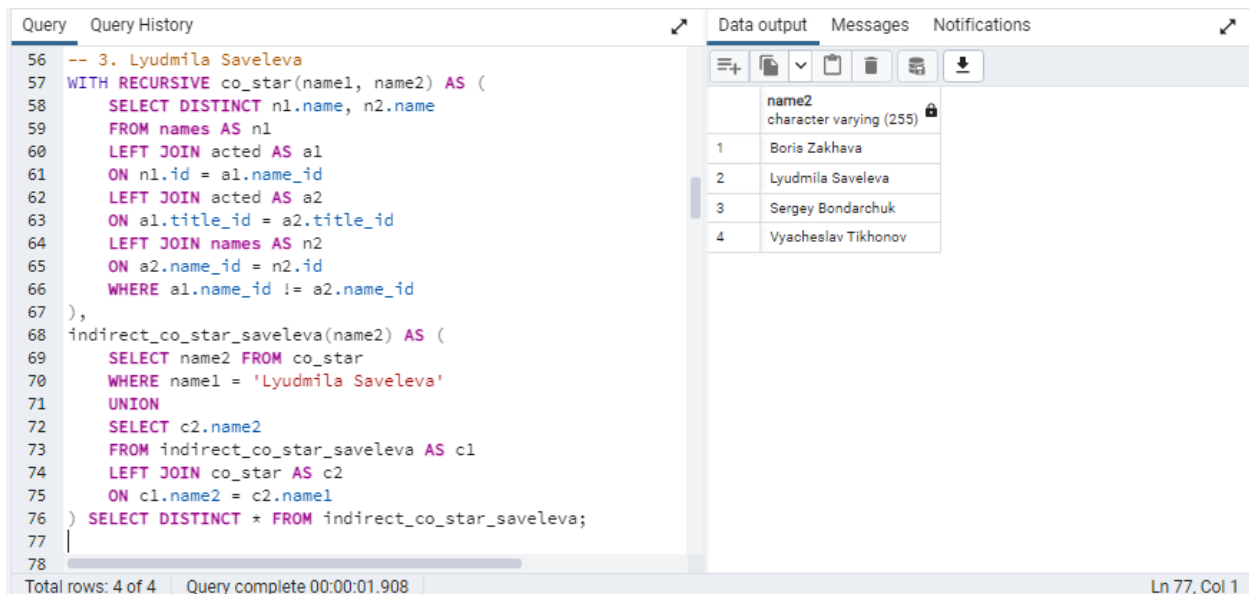
Se aprecia que se alcanzan un 18.752 actores mediante una cadena de co-stars indirectos.

3. Co-stars indirectos Lyudmila Saveleva

Repetiremos el ejercicio de buscar una cadena de co-stars indirectos para la actriz Lyudmila Saveleva, rusa, conocida por ganar el oscar en 1969 por la película Guerra y Paz (1965). Su historia es interesante, ya que al volver a Moscú, le quitaron el oscar. La lógica de la query es similar al punto anterior:

```
WITH RECURSIVE co_star(name1, name2) AS (  
    SELECT DISTINCT n1.name, n2.name  
    FROM names AS n1  
    LEFT JOIN acted AS a1  
    ON n1.id = a1.name_id  
    LEFT JOIN acted AS a2  
    ON a1.title_id = a2.title_id  
    LEFT JOIN names AS n2  
    ON a2.name_id = n2.id  
    WHERE a1.name_id != a2.name_id  
) ,  
indirect_co_star_saveleva(name2) AS (  
    SELECT name2 FROM co_star  
    WHERE name1 = 'Lyudmila Saveleva'  
    UNION  
    SELECT c2.name2  
    FROM indirect_co_star_saveleva AS c1  
    LEFT JOIN co_star AS c2  
    ON c1.name2 = c2.name1  
) SELECT DISTINCT * FROM indirect_co_star_saveleva;
```

Al ejecutar esta consulta obtenemos:



The screenshot shows a database query interface with two main panels: 'Query' and 'Data output'.

Query Panel: Displays the SQL query being executed. The query is a recursive query that finds indirect co-stars of Lyudmila Saveleva. It starts with a CTE named 'co_star' that finds direct co-stars. Then, it uses a recursive query 'indirect_co_star_saveleva' to find indirect co-stars by joining the 'co_star' CTE with itself.

Data output Panel: Shows the results of the query. The results are displayed in a table with two columns: 'name2' and 'character varying (255)'. The table contains four rows of data, representing the indirect co-stars of Lyudmila Saveleva.

name2	character varying (255)
1	Boris Zakhava
2	Lyudmila Saveleva
3	Sergey Bondarchuk
4	Vyacheslav Tikhonov

At the bottom of the interface, it shows 'Total rows: 4 of 4' and 'Query complete 00:00:01.908'.

Figure 3: Screen 1.3: Co-star indirecto Lyudmila Saveleva

En este caso, es una cadena muy pequeña de 4 actores rusos, incluyendo a Lyudmila. Esto se puede explicar porque el resto de películas donde actúa Lyudmila (cine soviético y ruso) pueden no alcanzar el mínimo de 5 mil votos en su rating y solo se considera la película por la que ganó el oscar.

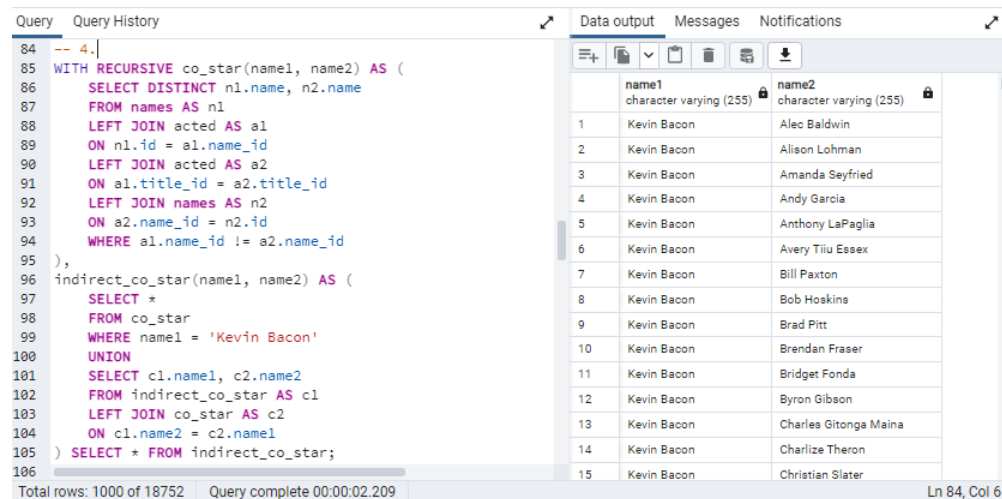
4. Misma consulta punto (2) pero indirect_co_star almacena parejas de actores conectadas por co-star:

Tenemos dos maneras de realizar esta consulta, con un filtro al inicio o al final. Comenzamos por un filtro inicial:

En este caso, al construir la consulta recursiva sobre co_star, realizamos el filtro por *Kevin Bacon* al comienzo del proceso, y retornamos tando el actor “de origen” como el co-star indirecto:

```
WITH RECURSIVE co_star(name1, name2) AS (  
    SELECT DISTINCT n1.name, n2.name  
    FROM names AS n1  
    LEFT JOIN acted AS a1  
    ON n1.id = a1.name_id  
    LEFT JOIN acted AS a2  
    ON a1.title_id = a2.title_id  
    LEFT JOIN names AS n2  
    ON a2.name_id = n2.id  
    WHERE a1.name_id != a2.name_id  
) ,  
indirect_co_star(name1, name2) AS (  
    SELECT *  
    FROM co_star  
    WHERE name1 = 'Kevin Bacon' -- filtro inicial  
    UNION  
    SELECT c1.name1, c2.name2  
    FROM indirect_co_star AS c1  
    LEFT JOIN co_star AS c2  
    ON c1.name2 = c2.name1  
) SELECT * FROM indirect_co_star;
```

Esta consulta es similar a la realizada en el punto (2) y entrega los mismos resultados:



	name1 character varying (255)	name2 character varying (255)
1	Kevin Bacon	Alec Baldwin
2	Kevin Bacon	Alison Lohman
3	Kevin Bacon	Amanda Seyfried
4	Kevin Bacon	Andy Garcia
5	Kevin Bacon	Anthony LaPaglia
6	Kevin Bacon	Avery Tiiu Essex
7	Kevin Bacon	Bill Paxton
8	Kevin Bacon	Bob Hoskins
9	Kevin Bacon	Brad Pitt
10	Kevin Bacon	Brendan Fraser
11	Kevin Bacon	Bridget Fonda
12	Kevin Bacon	Byron Gibson
13	Kevin Bacon	Charles Gitonga Maina
14	Kevin Bacon	Charlize Theron
15	Kevin Bacon	Christian Slater

Total rows: 1000 of 18752 Query complete 00:00:02.209 Ln 84, Col 6

Figure 4: Screen 1.4.1: Tabla indirect_co_star filtro en origen

Sin embargo, la segunda alternativa para plantear este proceso es realizar el filtro de actor de origen al final del proceso. Es decir, construir todas las relaciones de co-star indirectos, y usando el hecho de que la consulta entrega el nombre de origen y destino, realizar el filtro al final:

```
WITH RECURSIVE co_star(name1, name2) AS (
    SELECT DISTINCT n1.name, n2.name
    FROM names AS n1
    LEFT JOIN acted AS a1
    ON n1.id = a1.name_id
    LEFT JOIN acted AS a2
    ON a1.title_id = a2.title_id
    LEFT JOIN names AS n2
    ON a2.name_id = n2.id
    WHERE a1.name_id != a2.name_id
),
indirect_co_star(name1, name2) AS (
    SELECT *
    FROM co_star -- sin filtro inicial
    UNION
    SELECT c1.name1, c2.name2
    FROM indirect_co_star AS c1
    LEFT JOIN co_star AS c2
    ON c1.name2 = c2.name1
) SELECT * FROM indirect_co_star WHERE name1 = 'Kevin Bacon'; -- filtro final
```

En este caso el proceso se demora mucho más y entrega los siguientes resultados:

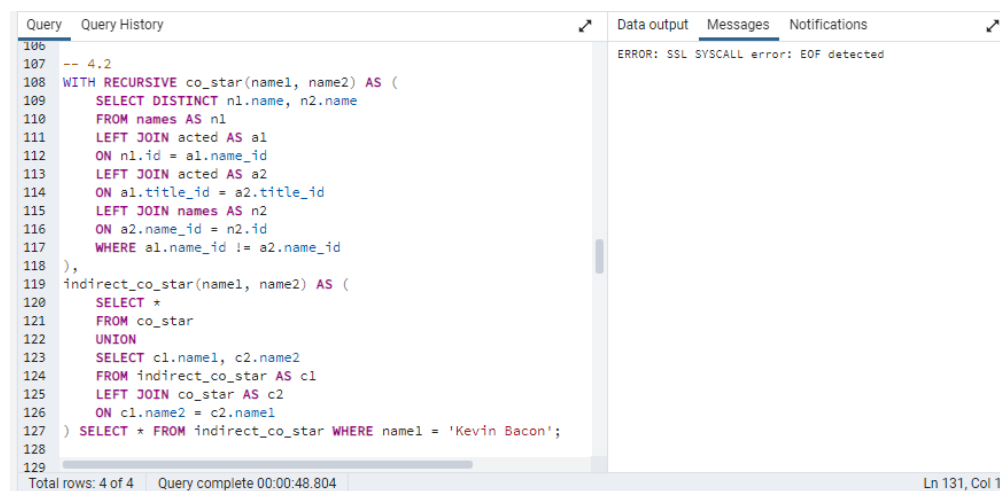


Figure 5: Screen 1.4.2: Tabla indirect_co_star filtro a la salida

Esto se debe a que en este caso construye la cadena completa de relaciones indirectas desde *todos* los actores y no solo desde un conjunto de actores de interés. El efecto de esto es que **no se cuenta con suficiente memoria para trabajar la tabla** que es lo que indica el mensaje de error encontrado en la consola.

Tiene sentido que ocurra un error de memoria porque estamos multiplicando la cantidad de actores en cada iteración. Siguiendo una analogía epidemiológica, el análisis es manejable cuando hacemos un filtro interno, porque es como partir por un subconjunto de agentes que se infecta iterativamente por una relación de participar en conjunto en una película. Dependiendo de la estructura de relaciones derivada de las películas tenemos un conjunto grande (Kevin Bacon) o uno muy pequeño (Lyudmila).

Si pretendemos comenzar por todos los actores, para construir la estructura de relaciones de participación

por películas iterativamente, estamos multiplicando la tabla actores en cada iteración.

Parte 2

1. Analizar query actores que han actuado como 'Batman'

Comenzamos por analizar el plan de ejecución de la consulta cuando se usa el operador JOIN:

```
EXPLAIN ANALYZE
SELECT DISTINCT names.name
FROM names
JOIN acted on names.id = acted.name_id
WHERE acted.role = 'Batman';
```

El resultado de esto nos entrega:

```
103 -----
104 -- PARTE 2:
105 -----
106 -- 1.
107 -- todos los actores que han hecho rol de batman con JOIN
108 EXPLAIN ANALYZE
109 SELECT DISTINCT names.name
110 FROM names
111 JOIN acted on names.id = acted.name_id
112 WHERE acted.role = 'Batman';
113
114 -- Unique (cost=1140.21..1140.22 rows=2 width=14) (actual time=4.217..4.224 rows=13 loops=1)
115 -- -> Sort (cost=1140.21..1140.21 rows=2 width=14) (actual time=4.217..4.219 rows=23 loops=1)
116 --      Sort Key: names.name
117 --      Sort Method: quicksort Memory: 26kB
118 --      -> Nested Loop (cost=0.29..1140.20 rows=2 width=14) (actual time=0.378..4.200 rows=23 loops=1)
119 --            -> Seq Scan on acted (cost=0.00..1123.59 rows=2 width=4) (actual time=0.368..4.137 rows=23 loops=1)
120 --                  Filter: ((role)::text = 'Batman'::text)
121 --                  Rows Removed by Filter: 60024
122 --            -> Index Scan using names_pkey on names (cost=0.29..8.30 rows=1 width=18) (actual time=0.002..0.002 rows=1 loops=23)
123 --                  Index Cond: (id = acted.name_id)
124 -- Planning Time: 0.174 ms
125 -- Execution Time: 4.247 ms
126
127
```

QUERY PLAN
1 Unique (cost=1140.21..1140.22 rows=2 width=14) (actual time=6.137..6.144 rows=13 loops=1)
2 -> Sort (cost=1140.21..1140.21 rows=2 width=14) (actual time=6.137..6.139 rows=23 loops=1)
3 Sort Key: names.name
4 Sort Method: quicksort Memory: 26kB
5 -> Nested Loop (cost=0.29..1140.20 rows=2 width=14) (actual time=0.373..6.116 rows=23 loops=1)
6 -> Seq Scan on acted (cost=0.00..1123.59 rows=2 width=4) (actual time=0.367..6.018 rows=23 loops=1)
7 Filter: ((role)::text = 'Batman'::text)
8 Rows Removed by Filter: 60024
9 -> Index Scan using names_pkey on names (cost=0.29..8.30 rows=1 width=18) (actual time=0.003..0.003 rows=1 loops=23)
10 Index Cond: (id = acted.name_id)
11 Planning Time: 0.175 ms
12 Execution Time: 6.171 ms

Total rows: 12 of 12 Query complete 00:00:00.515 Ln 124, Col 27

Figure 6: Screen 2.1.1: Plan de Trabajo Actores con rol Batman, caso JOIN

Se aprecia como el plan de ejecución comienza por un **sort** usando las claves de **names** mediante **quicksort**. Luego genera un join por **nested loop** finalmente filtra por el rol Batman.

Al realizar la consulta sin JOIN tenemos:

```
EXPLAIN ANALYZE
SELECT DISTINCT names.name
FROM names, acted
WHERE names.id = acted.name_id
AND acted.role = 'Batman';
```



```
Query    Query History
128      -- todos los actores que han hecho rol de batman sin JOIN
129      EXPLAIN ANALYZE
130      SELECT DISTINCT names.name
131      FROM names, acted
132      WHERE names.id = acted.name_id
133      AND acted.role = 'Batman';
134
135      -- Unique (cost=1140.21..1140.22 rows=2 width=14) (actual time=4.543..4.549 rows=13 loops=1)
136      --   -> Sort (cost=1140.21..1140.21 rows=2 width=14) (actual time=4.542..4.544 rows=23 loops=1)
137      --       Sort Key: names.name
138      --       Sort Method: quicksort Memory: 26kB
139      --   -> Nested Loop (cost=0.29..1140.20 rows=2 width=14) (actual time=0.395..4.525 rows=23 loops=1)
140      --       -> Seq Scan on acted (cost=0.00..1123.59 rows=2 width=4) (actual time=0.380..4.443 rows=23 loops=1)
141      --           Filter: ((role)::text = 'Batman')::text
142      --           Rows Removed by Filter: 60024
143      --       -> Index Scan using names_pkey on names (cost=0.29..8.30 rows=1 width=18) (actual time=0.003..0.003 rows=1 loops=23)
144      --           Index Cond: (id = acted.name_id)
145      -- Planning Time: 0.188 ms
146      -- Execution Time: 4.572 ms
Data output  Messages  Notifications
[Icons]
QUERY PLAN
TEXT
1 Unique (cost=1140.21..1140.22 rows=2 width=14) (actual time=6.330..6.337 rows=13 loops=1)
2 -> Sort (cost=1140.21..1140.21 rows=2 width=14) (actual time=6.329..6.331 rows=23 loops=1)
3 Sort Key: names.name
4 Sort Method: quicksort Memory: 26kB
5 -> Nested Loop (cost=0.29..1140.20 rows=2 width=14) (actual time=0.415..6.306 rows=23 loops=1)
6 -> Seq Scan on acted (cost=0.00..1123.59 rows=2 width=4) (actual time=0.407..6.218 rows=23 loops=1)
7 Filter: ((role)::text = 'Batman')::text
8 Rows Removed by Filter: 60024
9 -> Index Scan using names_pkey on names (cost=0.29..8.30 rows=1 width=18) (actual time=0.003..0.003 rows=1 loops=23)
10 Index Cond: (id = acted.name_id)
11 Planning Time: 0.178 ms
12 Execution Time: 6.363 ms
Total rows: 12 of 12    Query complete 00:00:00.486
```

Se aprecia que el plan involucra realizar un quicksort por names, luego un neste loopdy finalmente un filtter por rol 'Batman'. Es decir **el plan de ejecución es el mismo en ambos casos.**

2. Comparar tiempo de consulta sin indices y usando indice *btree*:

Comenzando por la consulta anterior usando JOIN, podemos obtener un rango de tiempo de ejecución de esta:

Query	Query History	Data output	Messages	Notifications
161	-- Pregunta 2: Analizar Execution Time			
162				
163	-- Execution Time por defecto			
164	EXPLAIN ANALYZE	QUERY PLAN		
165	SELECT DISTINCT names.name	text		
166	FROM names	1 Unique (cost=1140.21..1140.22 rows=2 width=14) (actual time=4.626..4.632 rows=13 loops=		
167	JOIN acted on names.id = acted.name_id	2 -> Sort (cost=1140.21..1140.21 rows=2 width=14) (actual time=4.625..4.627 rows=23 loops=		
168	WHERE acted.role = 'Batman';	3 Sort Key: names.name		
169	-- Execution Time varia entre 3 a 6ms	4 Sort Method: quicksort Memory: 26kB		
170		5 -> Nested Loop (cost=0.29..1140.20 rows=2 width=14) (actual time=0.490..4.608 rows=23 lo		
171		6 -> Seq Scan on acted (cost=0.00..1123.59 rows=2 width=4) (actual time=0.478..4.549 rows=2		
172		7 Filter: ((role)::text = 'Batman'::text)		
173		8 Rows Removed by Filter: 60024		
174		9 -> Index Scan using names_pkey on names (cost=0.29..8.30 rows=1 width=18) (actual time=		
175		10 Index Cond: (id = acted.name_id)		
176		11 Planning Time: 0.191 ms		
177		12 Execution Time: 4.657 ms		
178				
179				
180				
181				
Total rows: 12 of 12 Query complete 00:00:00.341		Ln 174, Col 1		

Figure 8: Screen 2.2.1: Consulta Batman sin indice

En el caso sin índices la consulta tomó 4.6 ms, y el rango encontrado va entre 3 a 6 ms.

Ahora repetimos la consulta creando un índice *btree*

Query	Query History	Data output	Messages	Notifications
184				
185				
186				
187				
188	-- agregar btree acted.role	QUERY PLAN		
189	CREATE INDEX indice_btree ON acted (role);	text		
190	-- correr nuevamente	1 Unique (cost=28.64..28.65 rows=2 width=14) (actual time=0.133..0.139 rows=13 loops=1)		
191	EXPLAIN ANALYZE	2 -> Sort (cost=28.64..28.64 rows=2 width=14) (actual time=0.133..0.134 rows=23 loops=1)		
192	SELECT DISTINCT names.name	3 Sort Key: names.name		
193	FROM names	4 Sort Method: quicksort Memory: 26kB		
194	JOIN acted on names.id = acted.name_id	5 -> Nested Loop (cost=4.72..28.63 rows=2 width=14) (actual time=0.042..0.118 rows=23 lo		
195	WHERE acted.role = 'Batman';	6 -> Bitmap Heap Scan on acted (cost=4.43..12.02 rows=2 width=4) (actual time=0.022..0.04		
196	-- Execution Time varia entre 0.1 y 0.3ms	7 Recheck Cond: ((role)::text = 'Batman'::text)		
197	-- el btree mejora significativamente el rendimiento	8 Heap Blocks: exact=23		
198		9 -> Bitmap Index Scan on indice_btree (cost=0.00..4.43 rows=2 width=0) (actual time=0.017		
199		10 Index Cond: ((role)::text = 'Batman'::text)		
200		11 -> Index Scan using names_pkey on names (cost=0.29..8.30 rows=1 width=18) (actual time		
201		12 Index Cond: (id = acted.name_id)		
202		13 Planning Time: 0.263 ms		
203				
204				
Total rows: 14 of 14 Query complete 00:00:00.388		Ln 201, Col 1		

Figure 9: Screen 2.2.2: Consulta Batman indice btree

Vemos que **el desempeño mejora usando btree** pasando a un rango entre 0.1 y 0.3 ms.

El índice por arbol binario (*btree*) mejora el rendimiento ya que en vez de una búsqueda secuencial se utiliza un **Bitmap Heap Scan**, la cual genera reglas de comparación por rangos.

Es interesante tener en cuenta esto ya que el sorteo (ordenamiento) ocurre igualmente, probablemente la tabla ya estaba ordenada. Esto significa que el lugar donde mejora el rendimiento es en la búsqueda de la palabra “Batman” y no en el JOIN o su ausencia.

3. Comparar tiempos de ejecución en tres casos: sin índice, índice *btree* e índice *hash* para títulos del año 2022.

Ahora construimos una consulta para rescatar la tabla `titles` filtrando para un año en particular.

- Cuando realizamos la consulta sin índices obtenemos:

The screenshot shows a database query interface with a query editor on the left and a query plan on the right. The query editor contains the following SQL code:

```
190
191
192 -- Pregunta 3:
193
194 -- sin indice
195 EXPLAIN ANALYZE
196 SELECT *
197 FROM titles
198 WHERE release_year = 2022;
199 -- Execution Time entre 0.7 y 1.1ms
200
201
202
203
204
205
206
207
208
209
210
```

The query plan on the right shows the following steps:

Step	Operation
1	Seq Scan on titles (cost=0.00..316.27 rows=67 width=35) (actual time=0.607..1.040 rows=67)
2	Filter: (release_year = 2022)
3	Rows Removed by Filter: 14755
4	Planning Time: 0.051 ms
5	Execution Time: 1.056 ms

The status bar at the bottom indicates "Total rows: 5 of 5" and "Query complete 00:00:00.400".

Figure 10: Screen 2.3.1: Consulta Titulos 2022 sin indice

Con un desempeño cercano a 1.0 ms.

- Luego, usando un arbol binario como índice:

The screenshot shows a database query interface with a query editor on the left and a query plan on the right. The query editor contains the following SQL code:

```
201
202
203
204 -- btree en release_year
205 CREATE INDEX indice_btree ON titles (release_year);
206 EXPLAIN ANALYZE
207 SELECT *
208 FROM titles
209 WHERE release_year = 2022;
210 DROP INDEX indice_btree;
211 -- Execution Time entre 0.04 y 0.07ms
212
213
214
215
216
217
218
219
220
221
```

The query plan on the right shows the following steps:

Step	Operation
1	Index Scan using indice_btree on titles (cost=0.29..25.18 rows=67 width=35) (actual time=0.040..0.070 rows=67)
2	Index Cond: (release_year = 2022)
3	Planning Time: 0.074 ms
4	Execution Time: 0.065 ms

The status bar at the bottom indicates "Total rows: 4 of 4" and "Query complete 00:00:00.695".

Figure 11: Screen 2.3.2: Consulta Titulos 2022 indice btree

Con *btree* tenemos una mejora importante en tiempo de ejecución que llega a 0.04 y 0.07 ms.

- Finalmente, usando un índice hash:

Se aprecia una mejora también significativa, en este caso llegando al rango 0.05 y 0.09 ms.

En ambos casos, al usar índice el rendimiento mejora. Sin embargo, se alcanzó un mejor rendimiento en el margen con el índice *btree*. Esto puede parecer contra intuitivo.

The screenshot shows a database query interface with a SQL editor on the left and a query plan on the right. The SQL editor contains the following code:

```

214
215
216 -- indice hash
217 CREATE INDEX indice_hash ON titles USING HASH (release_year)
218 EXPLAIN ANALYZE
219 SELECT *
220 FROM titles
221 WHERE release_year = 2022;
222 DROP INDEX indice_hash;
223 -- Execution Time entre 0.05 y 0.09ms
224
225
226
227
228
229
230
231
232
233

```

The query plan on the right is titled "QUERY PLAN" and shows the following steps:

Step	Operation
1	Bitmap Heap Scan on titles (cost=4.52..117.35 rows=67 width=35) (actual time=0.020..0.063)
2	Recheck Cond: (release_year = 2022)
3	Heap Blocks: exact=28
4	-> Bitmap Index Scan on indice_hash (cost=0.00..4.50 rows=67 width=0) (actual time=0.011..0.011)
5	Index Cond: (release_year = 2022)
6	Planning Time: 0.060 ms
7	Execution Time: 0.083 ms

At the bottom of the interface, it shows "Total rows: 7 of 7", "Query complete 00:00:00.580", and "Ln 218, Col 1".

Figure 12: Screen 2.3.3: Consulta Titulos 2022 indice hash

El mecanismo de este opera creando un árbol de comparaciones donde se hace más rápido la búsqueda de un resultado. En cambio el índice *hash* genera un regla sobre la cual se arman distintos *buckets* (ejemplo, resto de division por 5). Se comenta que el índice *hash* permite acelerar la búsqueda de un valor en particular, pero no sirve tanto al buscar un rango.

Teniendo esto en cuenta uno esperaría que el índice *hash* generara mejores desempeños, debido a que la consulta implica un filtro por igualdad. Sin embargo, el desempeño de *btree* es comparable e incluso mejor para un filtro por igualdad en este caso particular.

4. Comparar tiempos de ejecución tres casos: sin índice, índice *btree* e índice *hash* para títulos de los años 2018 a 2020.

Ahora realizamos una consulta que involucra un *rango* por lo que esperamos que el índice *hash* tenga un peor desempeño que el índice *btree*.

Caso sin índice:

Query	Query History	Data output	Messages	Notifications
223				
224				
225	-- Pregunta 4:			
226				
227	-- sin índice			
228	EXPLAIN ANALYZE			
229	SELECT *			
230	FROM titles			
231	WHERE release_year >= 2018			
232	AND release_year <= 2020;			
233	-- ET entre 0.8 y 1.8ms			
234				
235				
236				
237				
238				
239				
240				
241				
242				
243				
Total rows: 5 of 5		Query complete 00:00:00.359		Ln 236, Col 1

QUERY PLAN	
1	Seq Scan on titles (cost=0.00..353.33 rows=1485 width=35) (actual time=0.125..1.092 rows=)
2	Filter: ((release_year >= 2018) AND (release_year <= 2020))
3	Rows Removed by Filter: 13337
4	Planning Time: 0.055 ms
5	Execution Time: 1.149 ms

Figure 13: Screen 2.4.1: Consulta Titulos 2018-2020 sin índice

En el caso base sin índice tenemos un resultado centrado en 1 ms.

Caso índice *btree*:

Query	Query History	Data output	Messages	Notifications
246				
247	-- btree			
248	CREATE INDEX indice_btree ON titles (release_year);			
249	EXPLAIN ANALYZE			
250	SELECT *			
251	FROM titles			
252	WHERE release_year >= 2018			
253	AND release_year <= 2020;			
254	DROP INDEX indice_btree;			
255	-- 0.2 - 0.55ms			
256				
257				
258				
259				
260				
261				
262				
Total rows: 4 of 4		Query complete 00:00:00.326		Ln 257, Col 1

QUERY PLAN	
1	Index Scan using indice_btree on titles (cost=0.29..104.63 rows=1487 width=35) (actual time
2	Index Cond: ((release_year >= 2018) AND (release_year <= 2020))
3	Planning Time: 0.098 ms
4	Execution Time: 0.516 ms

Figure 14: Screen 2.4.2: Consulta Titulos 2018-2020 índice btree

Para el caso *btree* tenemos una mejora relevante, llegando al rango 0.2 a 0.5 ms

Caso índice *hash*:

The screenshot shows a database query interface with a query editor on the left and a query plan on the right. The query editor contains the following SQL code:

```
265 -- hash
266 CREATE INDEX indice_hash ON titles USING HASH (release_year)
267 EXPLAIN ANALYZE
268 SELECT *
269 FROM titles
270 WHERE release_year >= 2018
271 AND release_year <= 2020;
272 DROP INDEX indice_hash;
273 -- 0.8 - 1.5ms
274
275
276
277
278
279
280
281
```

The query plan on the right is titled "QUERY PLAN" and contains the following steps:

Step	Operation
1	Seq Scan on titles (cost=0.00..353.33 rows=1485 width=35) (actual time=0.125..1.147 rows=)
2	Filter: ((release_year >= 2018) AND (release_year <= 2020))
3	Rows Removed by Filter: 13337
4	Planning Time: 0.060 ms
5	Execution Time: 1.204 ms

The status bar at the bottom indicates "Total rows: 5 of 5", "Query complete 00:00:00.435", and "Ln 279, Col 1".

Figure 15: Screen 2.4.3: Consulta Titulos 2018-2020 indice hash

Finalmente el caso de índice *hash*, tenemos que el desempeño es similar al caso sin índice, con un desempeño centrado en 1.1 ms.

Estos resultados corroboran la intuición que nos dice que el índice *hash* funciona bien para acelerar la búsqueda de valores particulares (un año en específico) pero no funciona bien para buscar un rango de valores. Por otro lado, el índice *btree* funciona bien al acelerar las búsquedas de años en particular y también para la búsqueda de un rango de valores, como este ejercicio demuestra.