

Postgres

Explain y Explain Analyze

Para conocer el porqué de la velocidad de ejecución de una consulta se utiliza el comando EXPLAIN antes de la consulta lo cual muestra el resultado esperado de dicha consulta o el plan de la consulta.

Si se utiliza EXPLAIN ANALYZE se obtiene la estimación describiendo lo que espera el planificador de PostgreSQL y además lo que realmente sucede después de ejecutar la consulta.

Por ejemplo, si se ejecuta:

```
EXPLAIN ANALYZE DELETE * FROM actor;
```

No solo se va a obtener el plan sino efectivamente se va a eliminar todo el contenido de la tabla actor.

Estructura del plan de la consulta

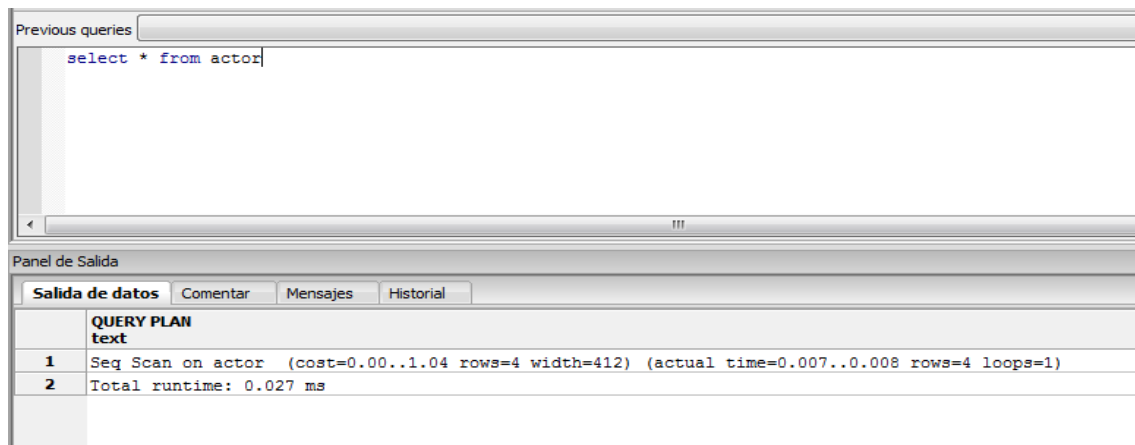
La salida del comando EXPLAIN está organizada en una serie de nodos. Cada línea de la salida es un nodo. Al más bajo nivel están los nodos que analizan tablas e índices. Los nodos de más alto nivel toman la salida de los nodos de bajo nivel y operan sobre esta.

Por ejemplo, si se ejecuta:

```
EXPLAIN ANALYZE SELECT * FROM actor;
```

La salida obtenida es la siguiente:

```
-----  
Seq Scan on actor ( cost=0.00..1.04 rows=4 width=412)  
(actual time=0.007..0.008 rows=4 loops=1)  
Total runtime: 0.027 ms  
(2 filas)
```



The screenshot shows a PostgreSQL query tool interface. At the top, there's a 'Previous queries' tab with the query 'select * from actor' entered. Below this is the 'Panel de Salida' (Output Panel) which has tabs for 'Salida de datos', 'Comentar', 'Mensajes', and 'Historial'. The 'Salida de datos' tab is selected, showing the query plan output. The output is structured as follows:

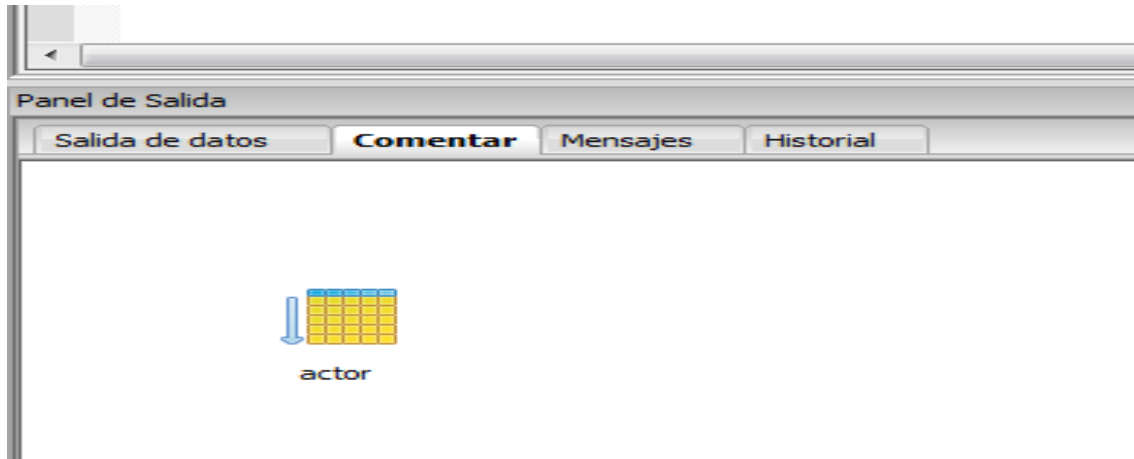
	QUERY PLAN text
1	Seq Scan on actor (cost=0.00..1.04 rows=4 width=412) (actual time=0.007..0.008 rows=4 loops=1)
2	Total runtime: 0.027 ms

Por ejemplo, si se ejecuta con SHIFT F7:

```
select * from actor;
```

SHIFT F7

La salida obtenida es la siguiente: Visualizamos un solo NODO



Este plan tiene un nodo llamado Seq Scan

cost=0.00..1.04: El primer costo (0.00) es el costo de iniciar el nodo, esto significa cuanto trabajo es estimado antes de que este nodo produzca su primera fila de salida, en este caso es 0.00 ya que una búsqueda secuencial devuelve resultados de forma inmediata.

El segundo número (1.04) es el costo estimado de ejecutar todo el nodo hasta que termine.

Rows = 4: es el número de filas que este nodo espera como salida.

Width = 412: es el número esperado de bytes de cada fila de la salida.

Los números actuales muestran como realmente se llevó a cabo la consulta:

actual time =0.007..0.008 : Costo actual de inicio no fue cero, se necesitó una pequeña fracción de tiempo para empezar. Una vez que empezó necesitó 0.008 segundos para terminar.

Rows = 4: tal como se esperaba fueron 4 filas.

Loops = 1: algunos nodos, como los que hacen joins, se ejecutan más de una vez. En ese caso el valor de loops va a ser más de uno y el tiempo y la cantidad de filas van a ser referidas a cada loop.

Costo de computación

La labor del optimizador es generar planes que puedan utilizarse para ejecutar una consulta y escoger el plan que tenga el menor costo de ejecución.

seq page cost: Cuanto demora leer una sola página desde el disco en forma secuencial.

random page cost: El costo de lectura cuando los registros están dispersos por varias páginas.

cpu tuple cost: Cuanto cuesta procesar un solo registro.

cpu index tuple cost: El costo de procesar una sola entrada de un índice. El valor por defecto es 0.005, menor de lo que cuesta procesar un registro debido a que los registros tienen mucha más información de cabecera (como xmin y xmax).

cpu operator cost: El costo esperado de procesar una función o un operador (por ejemplo la suma de dos números) y por defecto es 0.0025.

Todos estos valores por defecto se pueden encontrar en postgresql.conf. como muestra la siguiente imagen.

```
#-----
# QUERY TUNING
#-----

# - Planner Method Configuration -
#enable_bitmapscan = on
#enable_hashagg = on
#enable_hashjoin = on
#enable_indexscan = on
#enable_indexonlyscan = on
#enable_material = on
#enable_mergejoin = on
#enable_nestloop = on
#enable_seqscan = on
#enable_sort = on
#enable_tidscan = on

# - Planner Cost Constants -
#seq_page_cost = 1.0                # measured on an arbitrary scale
#random_page_cost = 4.0            # same scale as above
#cpu_tuple_cost = 0.01              # same scale as above
#cpu_index_tuple_cost = 0.005       # same scale as above
#cpu_operator_cost = 0.0025         # same scale as above
#effective_cache_size = 128MB

# - Genetic Query Optimizer -
#geqo = on
#geqo_threshold = 12
#geqo_effort = 5                    # range 1-10
#geqo_pool_size = 0                 # selects default based on effort
#geqo_generations = 0               # selects default based on effort
#geqo_selection_bias = 2.0          # range 1.5-2.0
#geqo_seed = 0.0                   # range 0.0-1.0

# - Other Planner Options -
#default_statistics_target = 100    # range 1-10000
#constraint_exclusion = partition   # on, off, or partition
```

Se pueden utilizar estos números para calcular el costo mostrado en el ejemplo. Una búsqueda secuencial de la tabla actor debe leer cada página en la tabla y procesar cada registro.

El siguiente query muestra los datos del catálogo relacionados con la tabla actor:

```
SELECT relpages, current_setting('seq_page_cost') AS seq_page_cost,
relpages * current_setting('seq_page_cost')::decimal AS page_cost,
reltuples, current_setting('cpu_tuple_cost') AS cpu_tuple_cost,
reltuples * current_setting('cpu_tuple_cost')::decimal AS tuple_cost
FROM pg_class WHERE relname='actor';
```

Panel de Salida						
<div>Salida de datos Comentar Mensajes Historial</div>						
	relpages integer	seq_page_cost text	page_cost numeric	reltuples real	cpu_tuple_cost text	tuple_cost double precision
1	1	1	1	4	0.01	0.04

El costo de leer la página (page cost) más el costo de procesar los registros (tuple cost) es igual a 1.04, el costo mostrado por EXPLAIN ANALYZE SELECT * FROM actor;

Para saber que columnas se están utilizando en una consulta, se puede utilizar el modo verboso:

EXPLAIN VERBOSE SELECT * FROM actor;

Panel de Salida	
<div>Salida de datos Comentar Mensajes Historial</div>	
	QUERY PLAN text
1	Seq Scan on public.actor (cost=0.00..1.04 rows=4 width=412)
2	Output: id actor, nombre, apellido

Optimización de consultas

Para optimizar las consultas se debe mejorar el rendimiento de la obtención de las filas y luego de las operaciones con sobre las filas.

Armando conjuntos de filas

Para optimizar la selección de las filas buscadas por una consulta se pueden tomar en cuenta diversas optimizaciones.

Id de las filas

Toda fila tiene un Id. Se puede utilizar en una misma transacción para referirse a una fila que se repite varias veces y así acelerar su búsqueda. También sirve para distinguir entre filas idénticas, por ejemplo al eliminar filas duplicados.

EXPLAIN SELECT id_actor FROM actor WHERE id_actor =1;

Panel de Salida	
Salida de datos	
Comentar	
Mensajes	
Historial	
	QUERY PLAN
	text
1	Seq Scan on actor (cost=0.00..1.05 rows=1 width=4)
2	Filter: (id actor = 1)

Índices

Cuando se ejecutan consultas es útil la búsqueda, mediante un campo que este indexado, ya que así el ordenamiento será más rápido.

Procesando los nodos

Una vez que se tiene un conjunto de filas, el siguiente tipo de nodo que se va a encontrar cuando se usa una sola tabla son aquellos que procesan el conjunto de varias formas. Estos nodos por lo general toman un conjunto de filas y devuelven otro.

Ordenamiento (Sort)

Nodos de ordenamiento aparecen cuando se utiliza ORDER BY en las consultas:

EXPLAIN ANALYZE SELECT id_actor FROM actor ORDER BY nombre;

Panel de Salida	
Salida de datos	
Comentar	
Mensajes	
Historial	
	QUERY PLAN
	text
1	Sort (cost=1.08..1.09 rows=4 width=208) (actual time=0.088..0.088 rows=4 loops=1)
2	Sort Key: nombre
3	Sort Method: quicksort Memory: 17kB
4	-> Seq Scan on actor (cost=0.00..1.04 rows=4 width=208) (actual time=0.006..0.007 rows=4 loops=1)
5	Total runtime: 0.117 ms

SELECT id_actor FROM actor ORDER BY nombre;

SHIFT F7

Nodos

Panel de Salida	
Salida de datos	
Comentar	
Mensajes	
Historial	



Las operaciones de ordenamiento se pueden ejecutar en memoria (más rápido) o en disco (más lento).

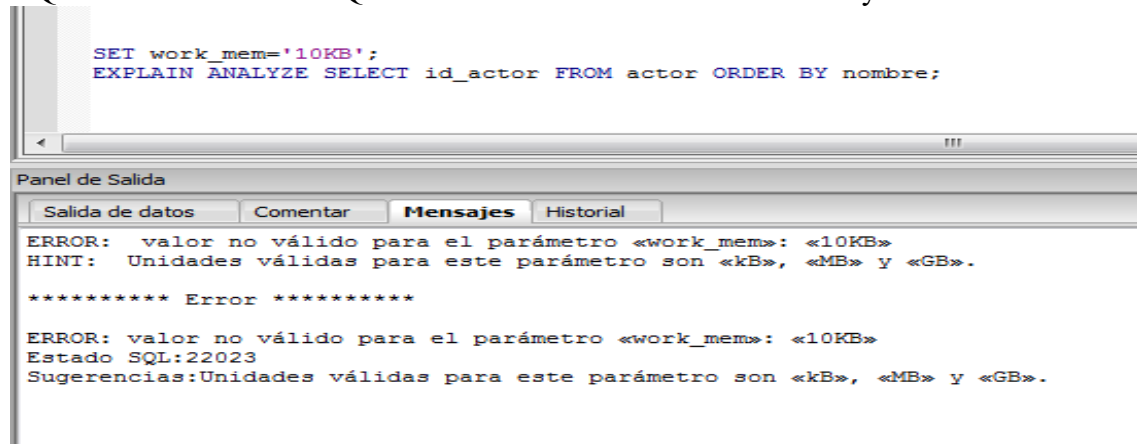
En el ejemplo podemos ver que esta ha sido calculada en memoria.

Sort method: quicksort Memory: 17kB

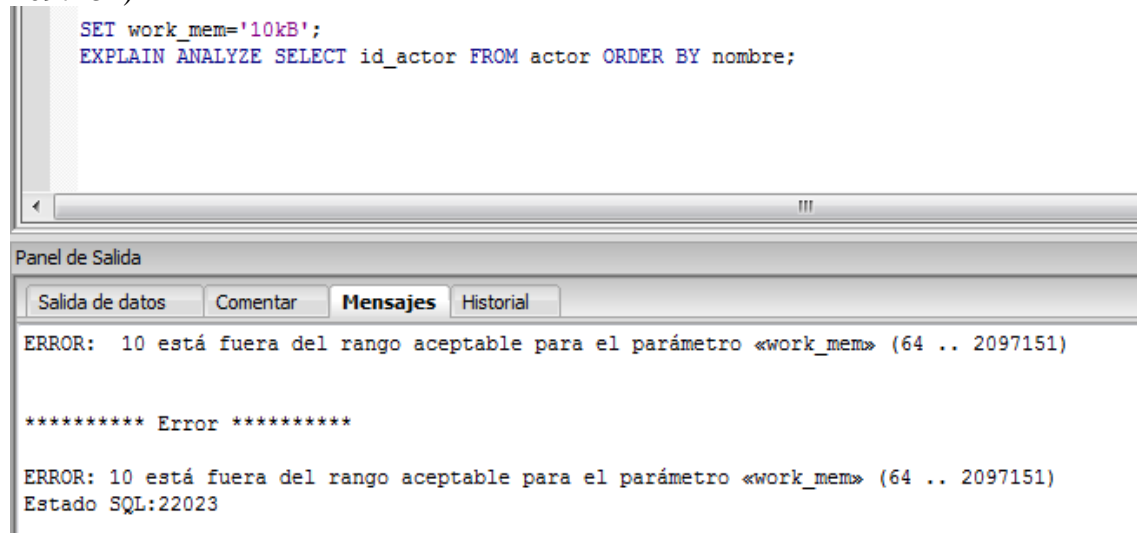
Esto depende del valor por defecto de work mem (1MB).

Si el parámetro work mem es incrementado o decrementado, la operación se llevará a cabo en memoria o en disco según su necesidad.

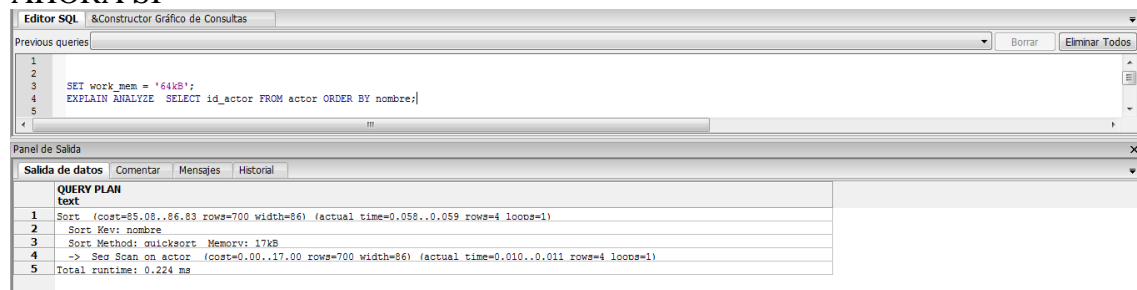
AQUÍ DIO ERROR PORQUE LA UNIDAD SE EXPRESA 'kB' y no 'KB'



AQUÍ DIO ERROR PORQUE EL RANGO DE LA UNIDAD ESTA ENTRE (64 y 2097151)



AHORA SI



Funciones de agregación

Las funciones de agregación reciben una serie de valores y producen una sola salida. Algunos ejemplos son AVG(), COUNT(), MIN(), MAX() y SUM(). Para calcular una

función de agregación, se leen todas las filas y luego se pasan por el nodo agregado para calcular el resultado:

```
EXPLAIN ANALYZE SELECT max(nombre) FROM actor;
```

Panel de Salida

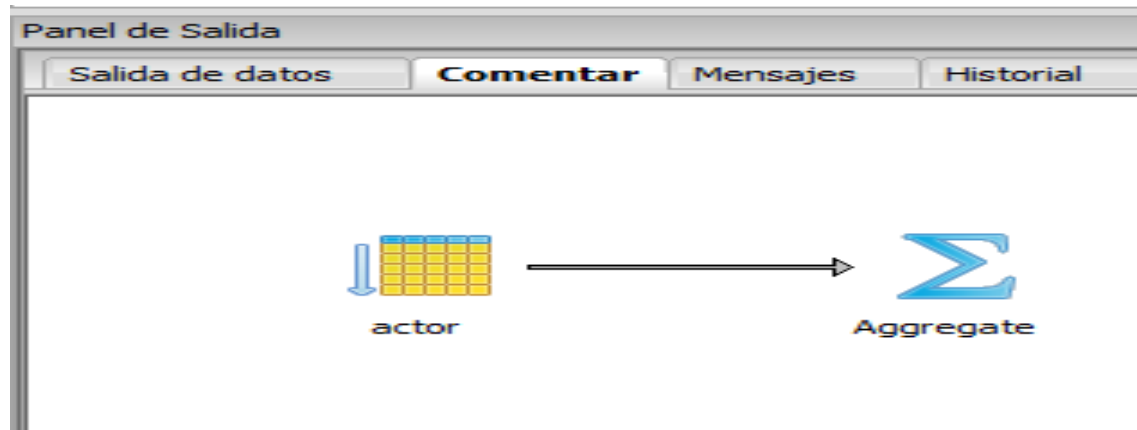
Salida de datos Comentar Mensajes Historial

	QUERY PLAN text
1	Aggregate (cost=1.05..1.06 rows=1 width=204) (actual time=0.027..0.027 rows=1 loops=1)
2	-> Seq Scan on actor (cost=0.00..1.04 rows=4 width=204) (actual time=0.004..0.005 rows=4 loops=1)
3	Total runtime: 0.051 ms

SELECT max(nombre) FROM actor;

SHIFT F7

Nodos



Producto Cartesiano

Las tareas más complejas del planificador de consultas son las que tienen que ver con unir tablas. Cada vez que se agrega una tabla al conjunto que se le aplicara join, el número de posibilidades crece. Si solo son tres tablas, el planificador considerará todo posible plan para seleccionar el óptimo, pero si son veinte tablas, la cantidad de posibilidades es muy grande para considerarlas todas.

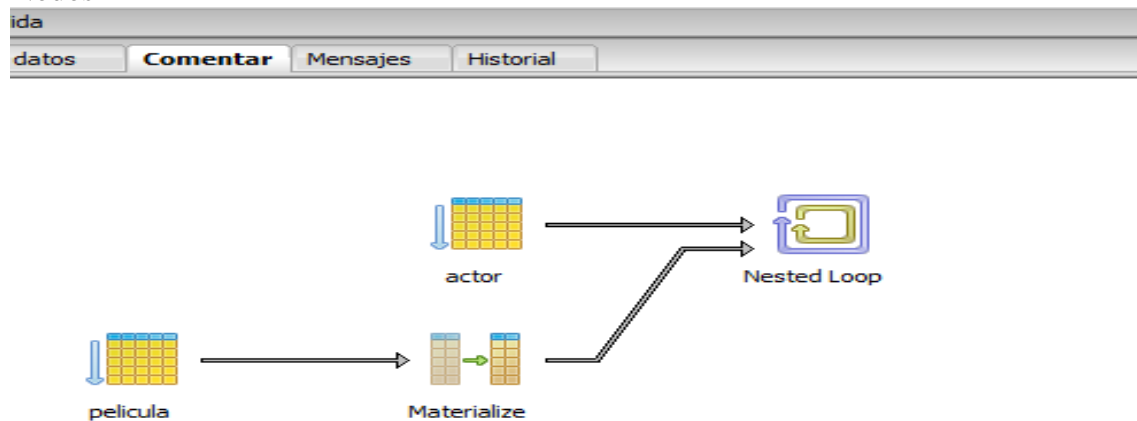
```
EXPLAIN ANALYZE SELECT * FROM actor,pelicula;
```

Panel de Salida	
Salida de datos Comentar Mensajes Historial	
	QUERY PLAN text
1	Nested Loop (cost=0.00..2.29 rows=16 width=628) (actual time=0.071..0.080 rows=16 loops=1)
2	-> Seq Scan on actor (cost=0.00..1.04 rows=4 width=412) (actual time=0.004..0.004 rows=4 loops=1)
3	-> Materialize (cost=0.00..1.06 rows=4 width=216) (actual time=0.016..0.016 rows=4 loops=4)
4	-> Seq Scan on pelicula (cost=0.00..1.04 rows=4 width=216) (actual time=0.060..0.061 rows=4 loops=1)
5	Total runtime: 0.136 ms

SELECT * FROM actor , pelicula ;

SHIFT F7

Nodos



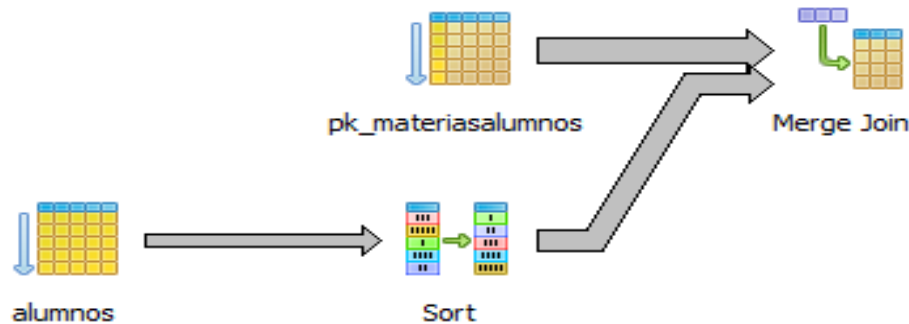
Nodo Materialize

Un nodo Materialize da como salida lo que está debajo de él en el árbol (Que puede ser una exploración, o un conjunto completo de uniones). Se materializa en la memoria antes de ejecutar el nodo superior.

Nodo Cruce de iteración anidada (Nested Loop) La relación derecha se recorre para cada fila encontrada en la relación izquierda. Esta estrategia es fácil de implementar pero puede consumir mucho tiempo.

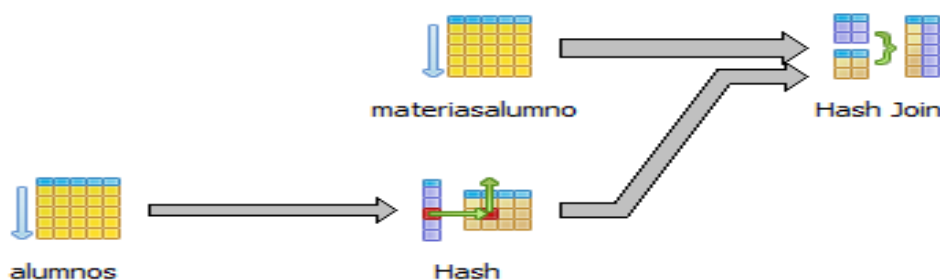
Otros ejemplos del EXPLAIN ANALYZE gráfico

```
select * from alumnos inner join materiasalumno on alumnos.matricula =
materiasalumno.matricula order by alumnos.matricula, materia
```



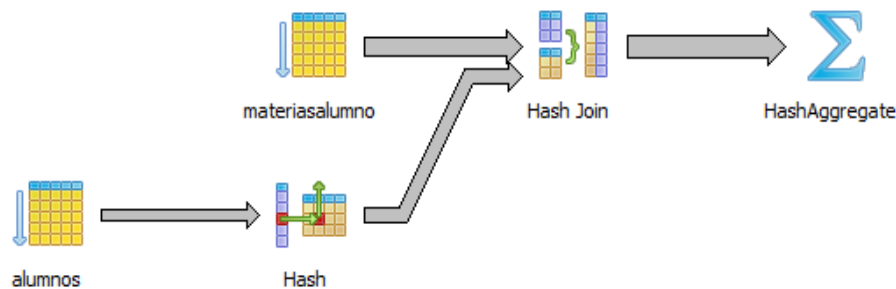
Nodo Cruce de ordenación mezclada (*Merge Join*) Cada relación es ordenada por los atributos del cruce antes de iniciar el cruce mismo. Después se mezclan las dos relaciones teniendo en cuenta que ambas relaciones están ordenadas por los atributos del cruce. Este modelo de cruce es más atractivo porque cada relación debe ser barrida sólo una vez.

```
select * from alumnos , materiasalumno where alumnos.matricula =
materiasalumno.matricula
```



Cruce indexado (*Hash Join*) La relación de la derecha se indexa primero sobre sus atributos para el cruce y se carga en una tabla *hash*. A continuación, se barre la relación izquierda, y los valores apropiados de cada fila encontrada se utilizan como clave indexada para localizar las filas de la relación derecha.

```
select apellidonombres , count(*) from alumnos inner join materiasalumno on
alumnos.matricula = materiasalumno.matricula group by apellidonombres
```



HashAggregate Utiliza una tabla hash temporal para agrupar filas. La operación HashAggregate no requiere un conjunto de datos preestablecidos, sino que utiliza grandes cantidades de memoria para materializar el resultado intermedio (no en forma de pipeline). La salida no se ordena de ninguna manera significativa.

explain analyze select apellidonombres , count(*) from alumnos inner join materiasalumno on alumnos.matricula = materiasalumno.matricula group by apellidonombres

de datos	Comentar	Mensajes	Historial
QUERY PLAN			
text			
HashAggregate (cost=1158.89..1175.87 rows=1698 width=132) (actual time=30.342..30.552 rows=709 loops=1)			
-> Hash Join (cost=129.25..1032.95 rows=25188 width=132) (actual time=1.124..14.920 rows=22878 loops=1)			
Hash Cond: (materiasalumno.matricula = alumnos.matricula)			
-> Seq Scan on materiasalumno (cost=0.00..525.88 rows=25188 width=17) (actual time=0.008..3.355 rows=25188 loops=1)			
-> Hash (cost=108.00..108.00 rows=1700 width=149) (actual time=1.103..1.103 rows=1700 loops=1)			
Buckets: 1024 Batches: 1 Memory Usage: 288kB			
-> Seq Scan on alumnos (cost=0.00..108.00 rows=1700 width=149) (actual time=0.004..0.482 rows=1700 loops=1)			
Total runtime: 30.724 ms			

Actividad

Ejemplo

Considérese la siguiente expresión de álgebra relacional: Hallar los nombres de todos los clientes que tengan una cuenta ubicada en cualquier sucursal en Luján.

SUCURSAL(ID_SUCURSAL, CIUDAD_SUCURSAL)

CUENTA(DNI_CLIENTE, ID_SUCURSAL, SALDO)

CLIENTE(DNI_CLIENTE, NOMBRE_CLIENTE)

Π nombre-cliente (σ ciudad-sucursal = "Luján" (sucursal \bowtie (cuenta \bowtie cliente)))

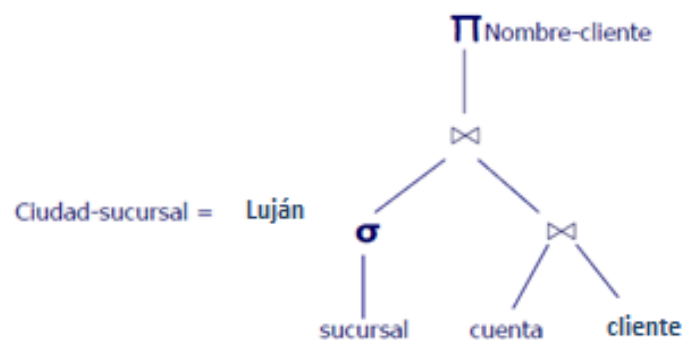
Árbol canónico de la consulta



consulta optimizada:

Π nombre-cliente ((σ ciudad-sucursal = "Luján" (sucursal)) \bowtie (cuenta \bowtie cliente))

Árbol optimizado de la consulta



Basándose en el ejemplo anterior

Ejercicio 1

Dadas las tablas siguientes:

PROYECTO(ID_PROYECTO, ID_DTO, NOMBRE_PROYECTO, LOCALIZACION)

EMPLEADO(ID_EMPLEADO, ID_CATEGORIA, APELLIDO, DIRECCION,
FECHA_NACIMIENTO)

DEPARTAMENTO(ID_DTO, NOMBRE_DTO, ID_JEFE)

1.a) Escribir el álgebra relacional (utilizando productos cartesianos) de la siguiente consulta, seleccione ID_PROYECTO, ID_DTO, APELLIDO, DIRECCION y FECHA_NACIMIENTO de los empleados jefes de departamento de los proyectos localizados en Luján.

1.b) Dibujar el árbol canónico de la consulta.

1.c) Optimizar la consulta y escribir el álgebra relacional y realizar el gráfico optimizado de la consulta.

1.d) Implementar el query de 1.a) y 1.c) y realizar la comparación de ambas consultas en Postgres usando EXPLAIN ANALYZE.

Ejercicio 2

Usando las tablas del Ejercicio 1 y agregando las tablas

CATEGORIA(ID_CATEGORIA, SUELDO)

TRABAJA(ID_PROYECTO, ID_EMPLEADO, HORAS)

Escribir los árboles canónico y optimizado correspondientes de la siguiente consulta e implementar la comparación de la consulta canónica y optimizada de cada consulta usando EXPLAIN ANALYZE de Postgres.

2.a) Mostrar el id_empleado, apellido, categoría y sueldo de los empleados que trabajan en el proyecto llamado “Felicidad”.

2.b) incrementar el sueldo en un 10% de los empleados que trabajan en el proyecto llamado “Felicidad”.