PostgreSQL
Explain y Explain Analyze

Para conocer el porque 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.

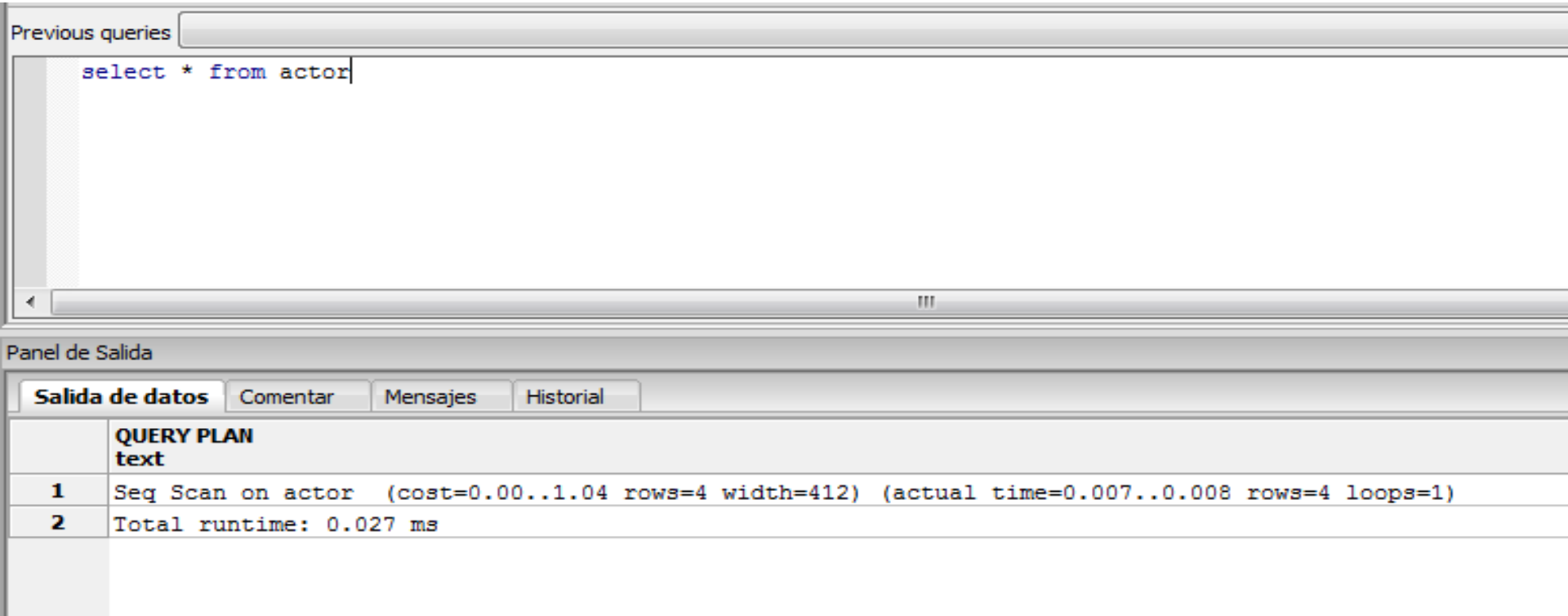
Estructura del plan de la consulta

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

Ejemplo

EXPLAIN ANALYZE SELECT * FROM actor;

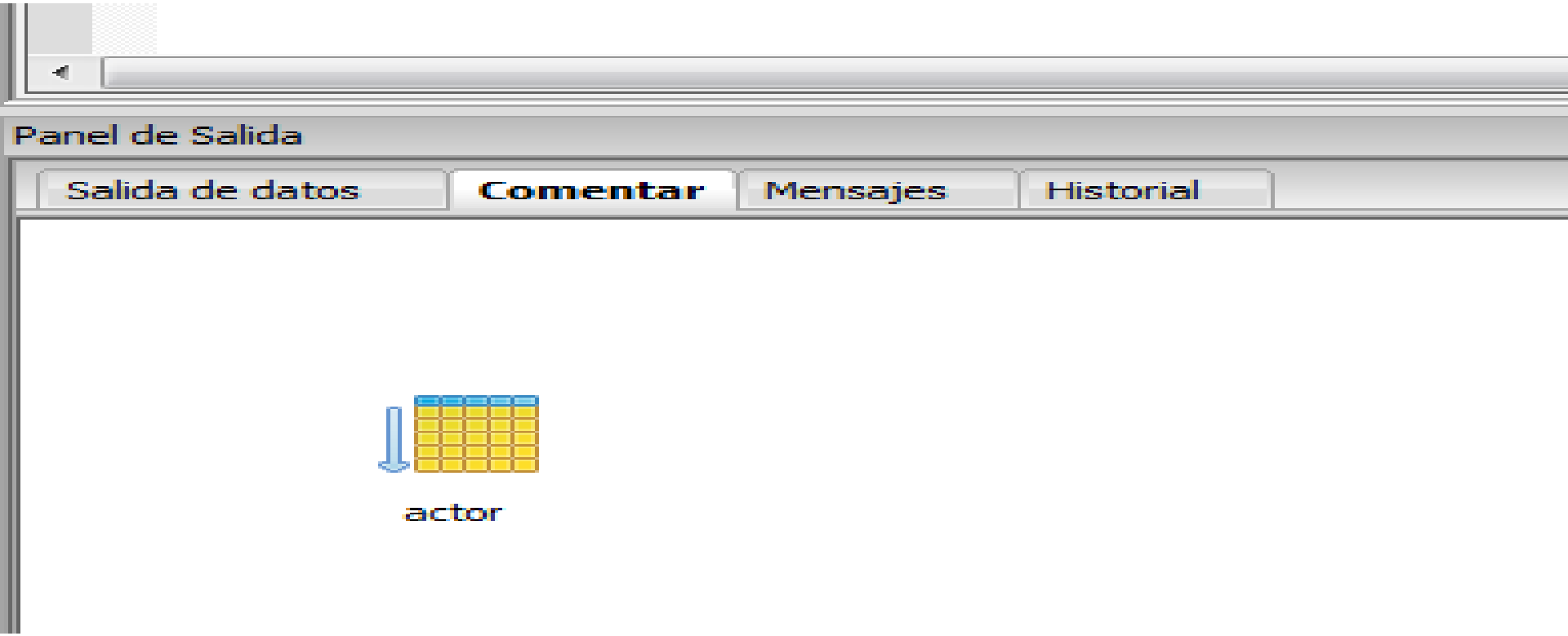
Seq Scan on actor (cost=0.00..1.04 rows=4 width=412)
(actual time=0.007..0.008 rows=4 loops=1)
Total runtime: 0.027 ms
(2 filas)



select * from actor;

SHIFT F7

Nodo



Al ejecutarse el query con EXPLAIN ANALYZE la duración aumenta varias veces. En casos de querys mas complejos la sobrecarga producida por el análisis es menor, aun así no se debe confiar completamente en los tiempos producidos por EXPLAIN ANALYZE.

Este plan tiene un nodo Seq Scan

cost=0.00..1.04: El primer costo es el costo de iniciar el nodo. Esto es 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 es el costo estimado de ejecutar todo el nodo hasta que termine.

Rows = 4: Número de filas que este nodo espera como salida.

Width = 412 : Número esperado de bytes de cada fila de la salida.

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

actual time =0.007..0.008 : Costo actual de inicio no fue cero, se necesito una pequeña fracción de tiempo para empezar. Una vez que empezó necesito 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 mas de una vez. En ese caso el valor de loops va a ser mas 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 mas 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**.

```
#------------------------------------------------------------------------------
# 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 pagina en la tabla y procesar cada registro.

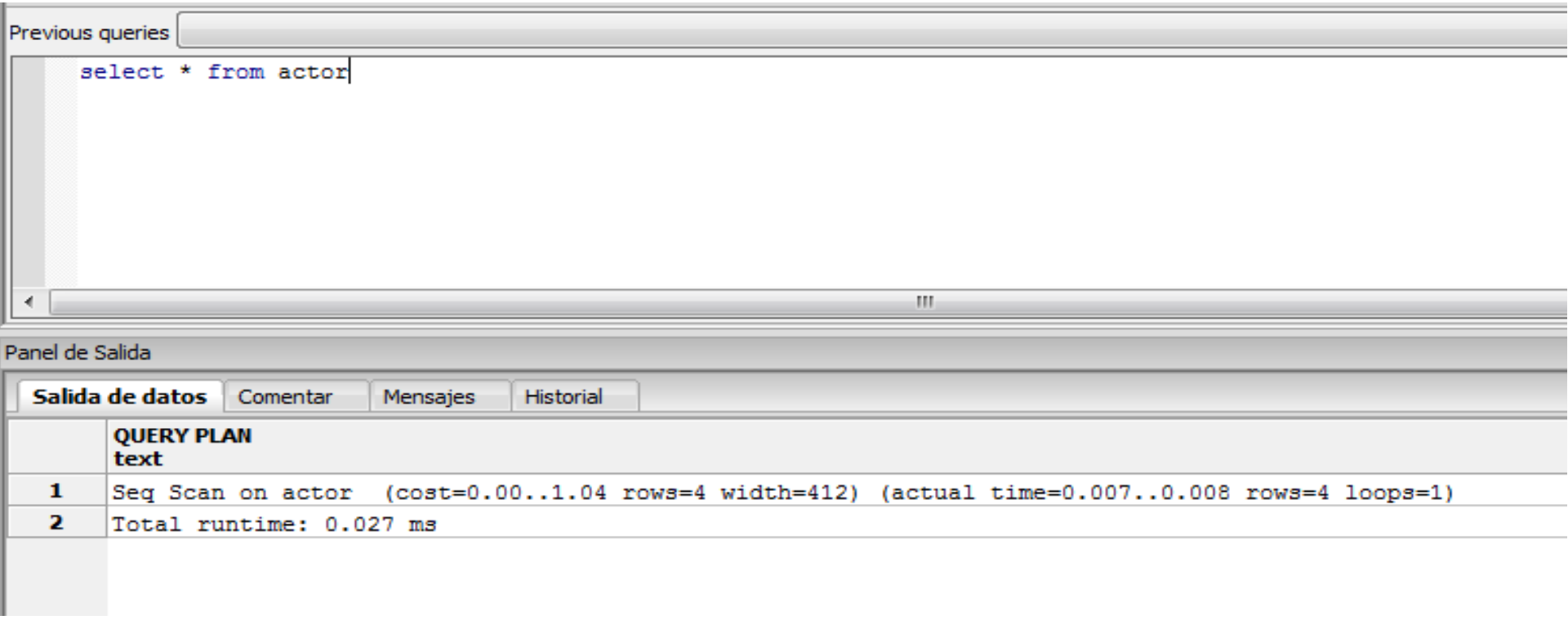
```
SELECT relpages, current_setting('seq_page_cost') AS seq_page_cost,
relpages * current_setting('seq_page_cost')::decimal AS page_cost,
reltuples, current_setting('cpu_tuple_cost') AS cpu_tuple_cost,
reltuples * current_setting('cpu_tuple_cost')::decimal AS tuple_cost
FROM pg_class WHERE relname =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

Panel de Salida

Salida de datos | Comentar | Mensajes | Historial

	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.



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

```
EXPLAIN VERBOSE SELECT * FROM actor;
```

Panel de Salida

Salida de datos | Comentar | Mensajes | Historial

	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 los registros y luego de las operaciones con los registros.

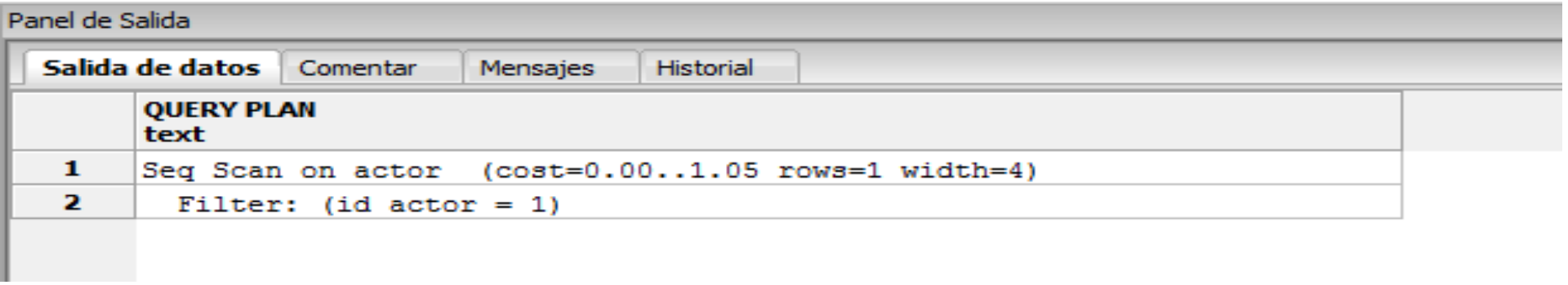
Armando conjuntos de registros

Para optimizar la selección de los registros buscados por una consulta se pueden tomar en cuenta diversas optimizaciones.

Id de los registros

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

```
EXPLAIN SELECT id_actor FROM actor WHERE id_actor =1;
```



Índices

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

Procesando los nodos

Una vez que se tiene un conjunto de registros, 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 registros 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

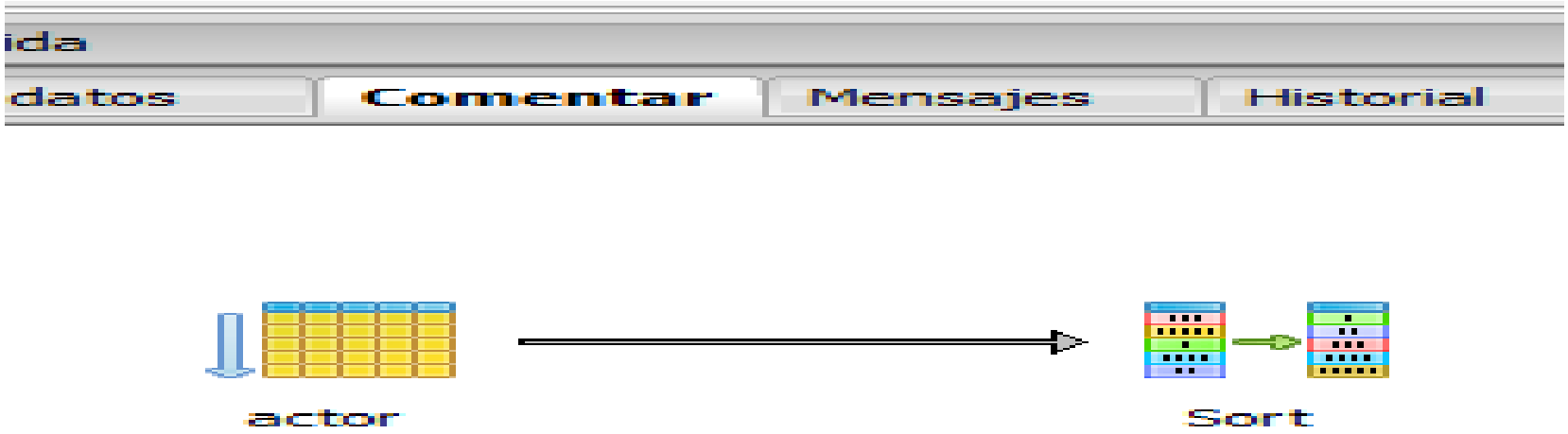
Salida de datos | Comentar | Mensajes | Historial

	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



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

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

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.

```
SET work_mem='10KB';
EXPLAIN ANALYZE SELECT id_actor FROM actor ORDER BY nombre;
```

Panel de Salida

Salida de datos | Comentar | **Mensajes** | Historial

```
ERROR:  valor no válido para el parámetro «work_mem»: «10KB»
HINT:  Unidades válidas para este parámetro son «kB», «MB» y «GB».

********** Error **********

ERROR: valor no válido para el parámetro «work_mem»: «10KB»
Estado SQL:22023
Sugerencias:Unidades válidas para este parámetro son «kB», «MB» y «GB».
```

work_mem

```
SET work_mem='10kB';
EXPLAIN ANALYZE SELECT id_actor FROM actor ORDER BY nombre;
```

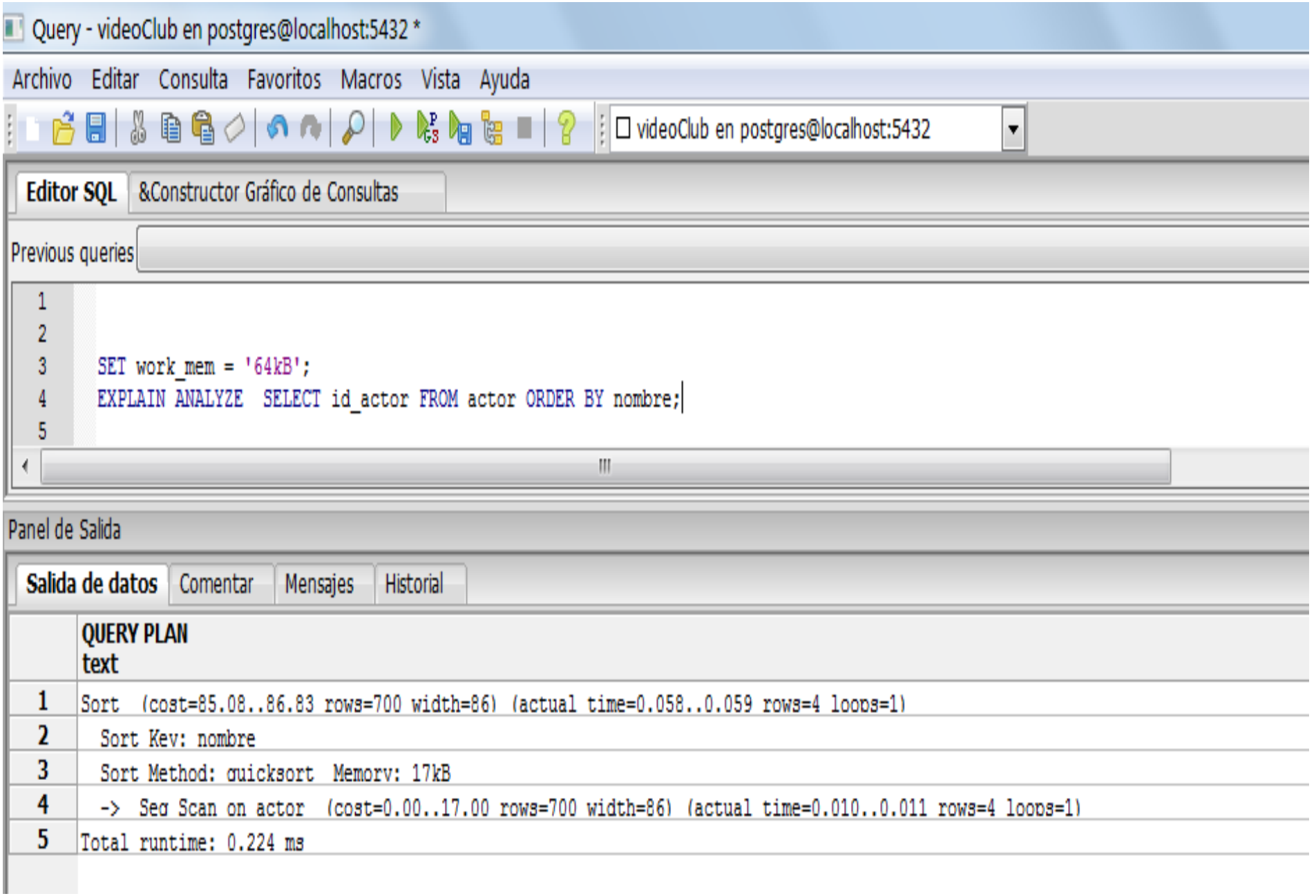
Panel de Salida

Salida de datos | Comentar | **Mensajes** | Historial

```
ERROR:  10 está fuera del rango aceptable para el parámetro «work_mem» (64 .. 2097151)


********** Error **********

ERROR: 10 está fuera del rango aceptable para el parámetro «work_mem» (64 .. 2097151)
Estado SQL:22023
```


Query - videoClub en postgres@localhost:5432 *
Archivo Editar Consulta Favoritos Macros Vista Ayuda
videoClub en postgres@localhost:5432
Editor SQL
&Constructor Gráfico de Consultas
Previous queries
1
2
3 SET work_mem = '64kB';
4 EXPLAIN ANALYZE SELECT id_actor FROM actor ORDER BY nombre;
5
Panel de Salida
Salida de datos
Comentar
Mensajes
Historial
QUERY PLAN
text
1 Sort (cost=85.08..86.83 rows=700 width=86) (actual time=0.058..0.059 rows=4 loops=1)
2 Sort Key: nombre
3 Sort Method: quicksort Memory: 17kB
4 -> Seq Scan on actor (cost=0.00..17.00 rows=700 width=86) (actual time=0.010..0.011 rows=4 loops=1)
5 Total runtime: 0.224 ms

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 todos los registros y luego se pasan por el nodo agregado para calcular el resultado:

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

Panel de Salida

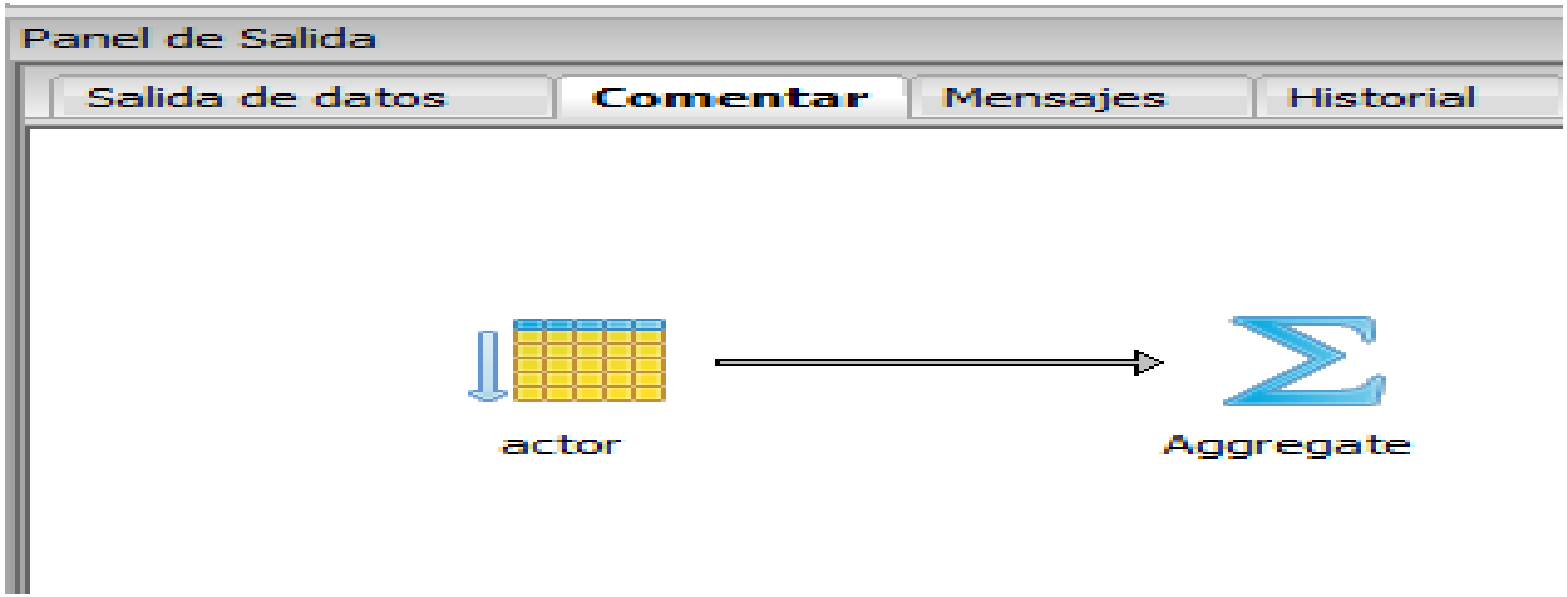
Salida de datos | Comentar | Mensajes | Historial

	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 entre si. 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

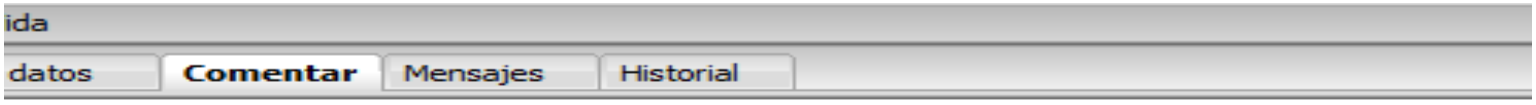
Salida de datos | Comentar | Mensajes | Historial

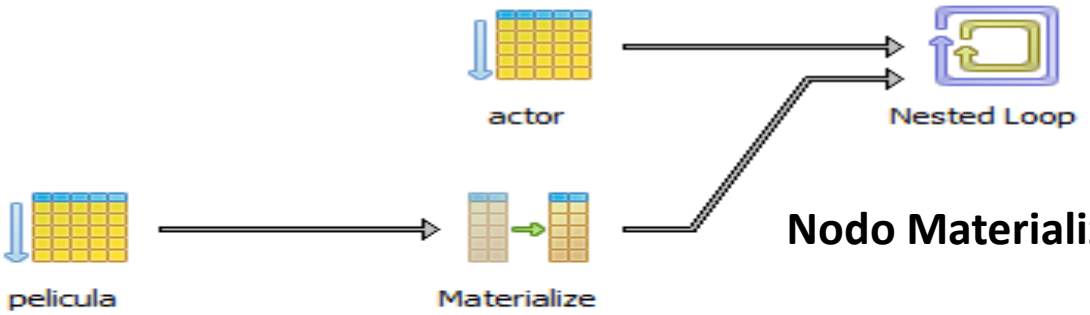
	QUERY PLAN text
1	Nested Loop (cost=0.00..2.29 rows=16 width=628) (actual time=0.071..0.080 rows=16 loops=1)
2	-> Seq Scan on actor (cost=0.00..1.04 rows=4 width=412) (actual time=0.004..0.004 rows=4 loops=1)
3	-> Materialize (cost=0.00..1.06 rows=4 width=216) (actual time=0.016..0.016 rows=4 loops=4)
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.

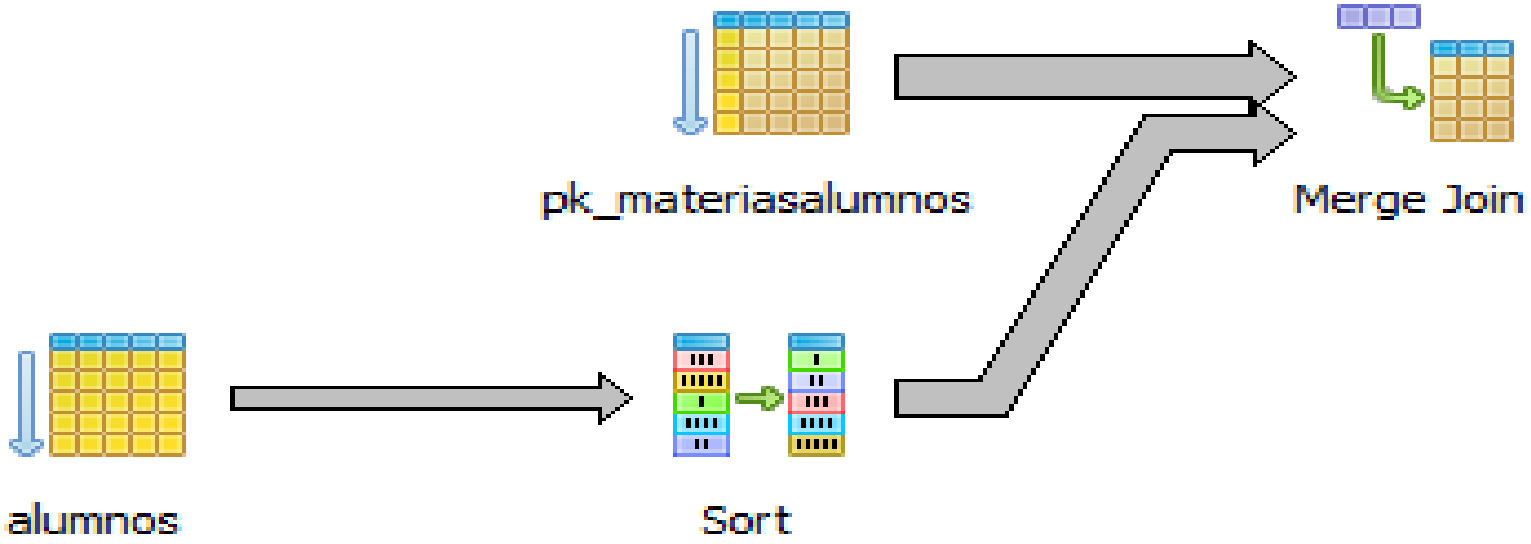
Nodo Cruce de iteración anidada (*Nested Loop*) La relación derecha se recorre para cada tupla encontrada en la relación izquierda. Esta estrategia es fácil de implementar pero puede consumir mucho tiempo.

EXPLAIN ANALYZE gráfico

El administrador gráfico de PostgreSQL, PgAdmin, posee la capacidad de explicar de forma gráfico los resultados de EXPLAIN ANALYZE. Proporciona un sustituto fácil de leer e interpretar, que puede ser utilizado en conjunto con la salida de texto de EXPLAIN ANALYZE.

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.

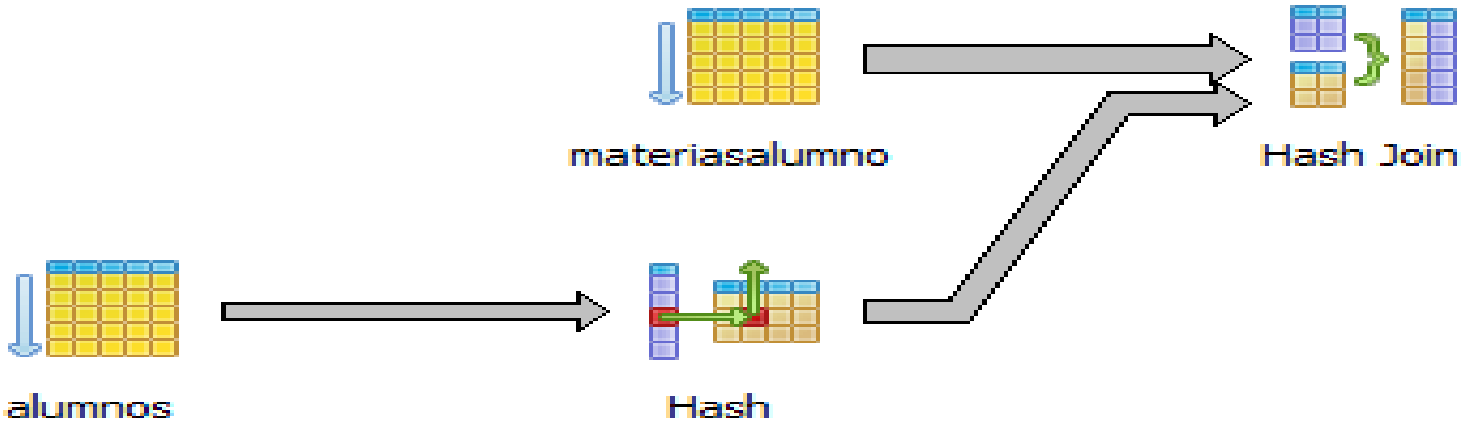
select * from alumnos inner join materiasalumno on alumnos.matricula = materiasalumno.matricula order by alumnos.matricula, materia



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 tupla encontrada se utilizan como clave indexada para localizar las tuplas de la relación derecha.

select * from alumnos inner join materiasalumno on alumnos.matricula = materiasalumno.matricula

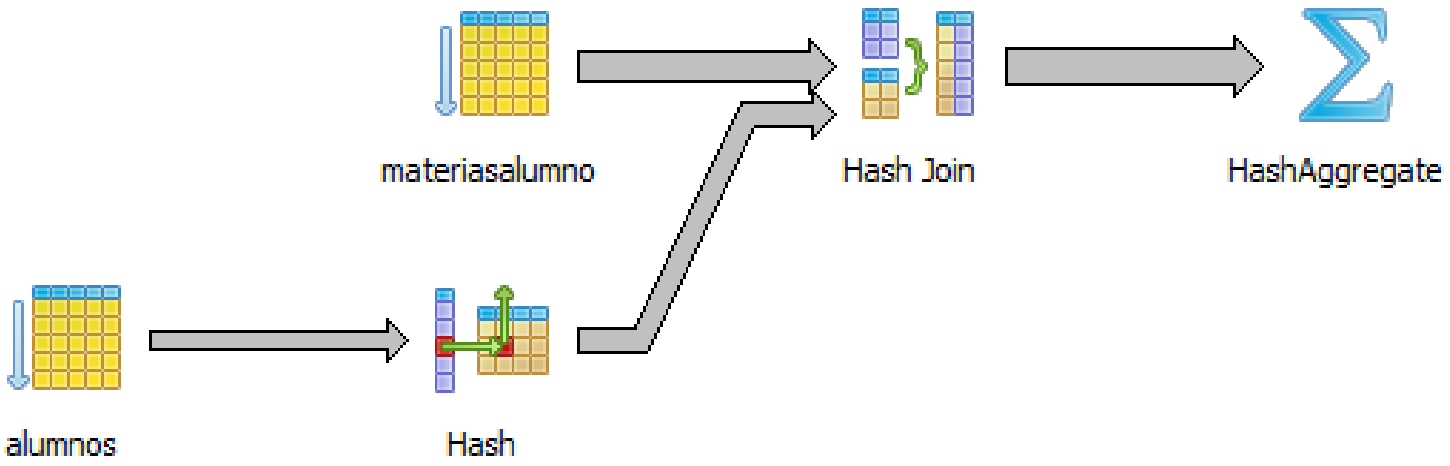
select * from alumnos , materiasalumno where alumnos.matricula = materiasalumno.matricula



HashAggregate

Utiliza una tabla hash temporal para agrupar registros. 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.

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



HashAggregate

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