Ejercicio Integrador de Optimización de Consultas

En la terminal de micros de Retiro se cuenta con un sistema con el siguiente esquema de base de datos:

Micro(numMicro, marca, numEmpresa)
Empresa(numEmpresa, nombreEmpresa, fechaCreacion, direccion)
Viaje(numMicro,fechaViaje,destino)

- Micro.numEmpresa no admite nulos y es FK de Empresa.
- Viaje.numMicro es FK de Micro.

Se desea optimizar la siguiente consulta:

```
SELECT
        numMicro, numEmpresa, nombreEmpresa, fechaCreacion, direccion,destino, fechaViaje
FROM
        Viaje V, Empresa E, Micro M
WHERE
                M.numMicro = V.numMicro
AND             M.numEmpresa = E.numEmpresa
AND             E.fechaCreacion >= '1/1/2000'
AND             V.fechaViaje >= '1/1/2007'
AND             V.fechaViaje < '1/2/2007'
AND             M.marca = 'Mercedes Benz'
```

Se cuenta con las siguientes estructuras adicionales:

- $\mathbf{I_1}$: Índice Hash sobre *fechaCreacion* en la tabla *Empresa*.
- $\mathbf{I_2}$: Índice B+ unclustered sobre *numEmpresa* en *Empresa* (altura 3).
- $\mathbf{I_3}$: Índice B+ clustered sobre <*fechaViaje,numMicro*> en *Viaje* (altura 5).
- Las tablas *Micro* y *Empresa* se guardan como heap file.

La longitud de los campos es de 128 bytes cada uno.

Se tiene como datos:

- Tuplas Micro: 200.000.
- Tuplas Empresa: 3.000.
- Tuplas Viaje: 2.500.000.
- Micros con marca 'Mercedes Benz': 50.000.
- Viajes de enero del 2007: 100.000.
- Empresas creadas desde el año 2000: 2.000.

Además:

- Se cuenta con 5 bloques de memoria.
- El tamaño de bloque es 2048 bytes.
- Se asume distribución uniforme.

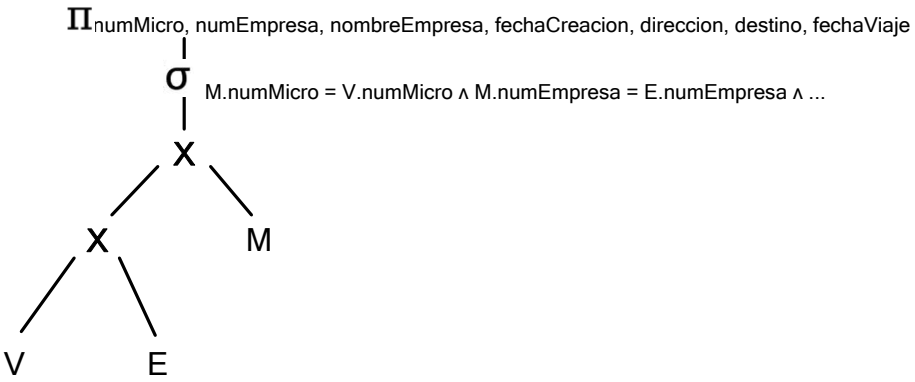
a) Armar el árbol canónico de la consulta.
b) Obtener un árbol optimizado, indicando su plan de ejecución, y calcular los costos.
c) Describir brevemente qué estructura agregaría para mejorar la performance de la consulta (sin calcular costos).

Resolución

Primero armemos la consulta en AR (no es un paso obligatorio pero puede ayudar a armar el árbol canónico):

$\pi_{numMicro,\ nombreEmpresa,\ numEmpresa,\ fechaCreacion,\ direccion,\ destino,\ fechaViaje}\ (\sigma_{M.numMicro\ =\ V.numMicro\ \ldots}\ (V \times E \times M))$

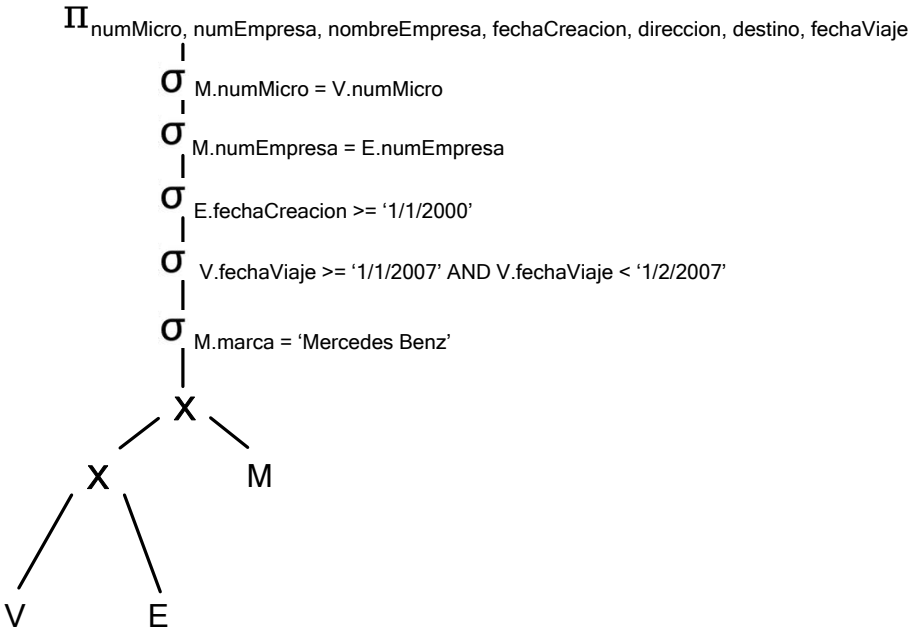
A partir de esto, armemos el árbol canónico:



Una vez hecho esto, tratemos de utilizar heurísticas para llegar a un árbol optimizado (no necesariamente EL óptimo).

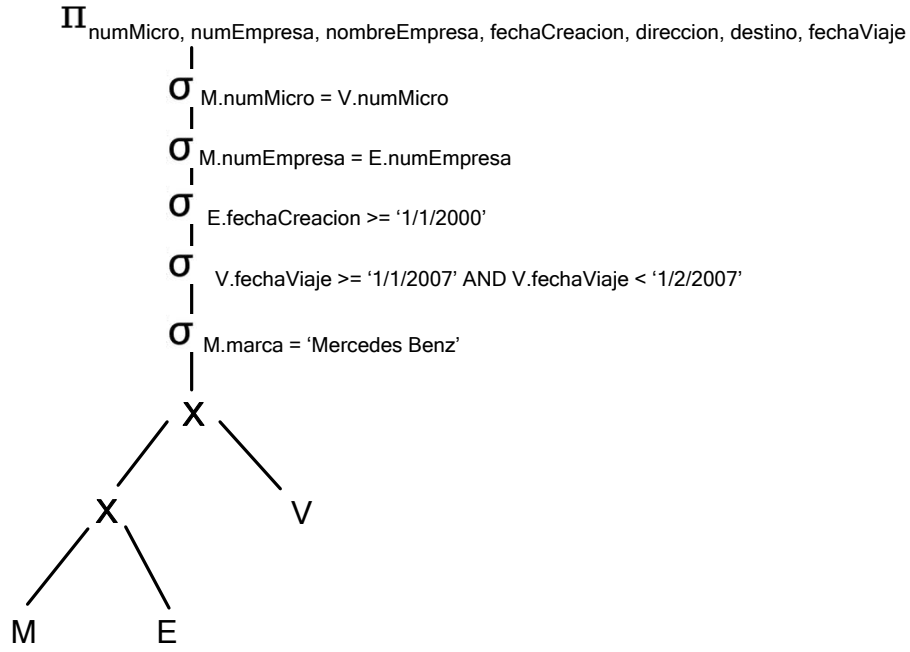
Paso 1

Separar la selección en varias selecciones.



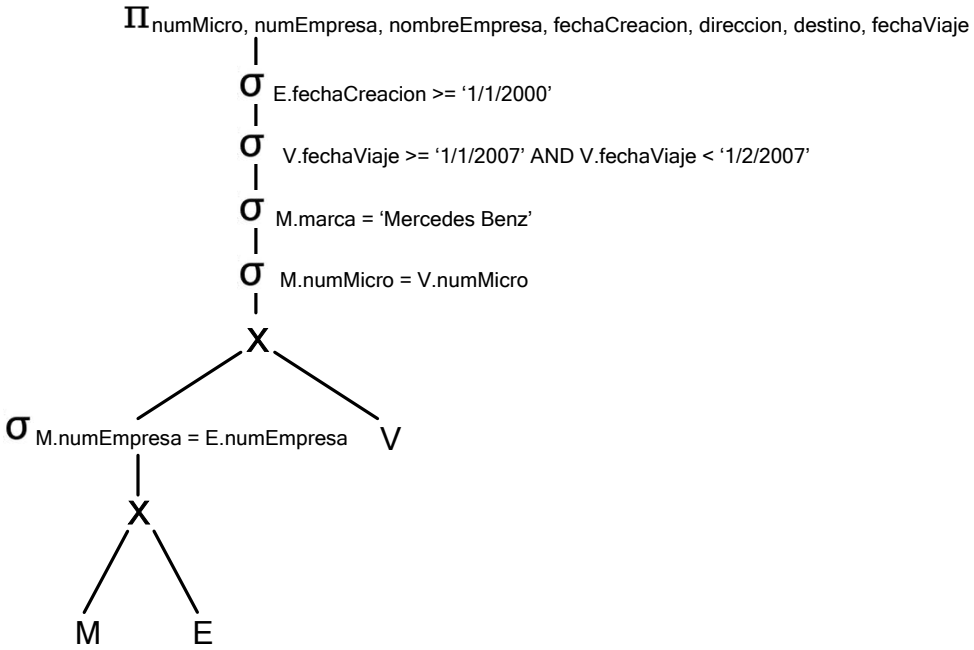
Paso 2

Permutar las hojas para evitar juntar relaciones sin atributos en común. En este caso, permutamos V y M.



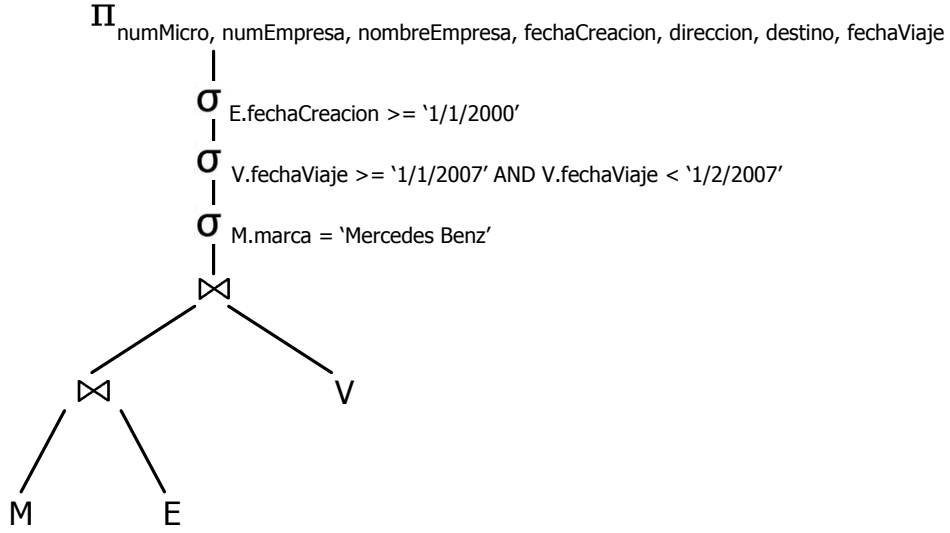
Paso 3

Bajar las selecciones que son condiciones de junta para poder cambiar productos cartesianos por *natural joins*.



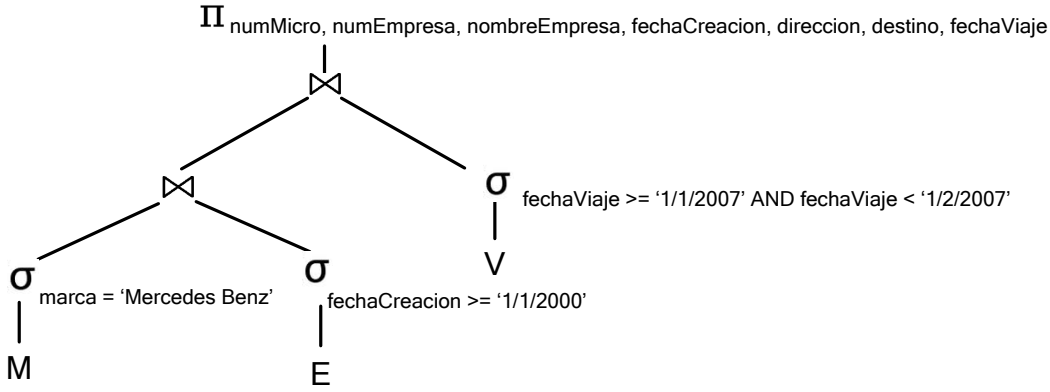
Paso 4

Cambiar productos cartesianos + condición de junta por *natural joins*.



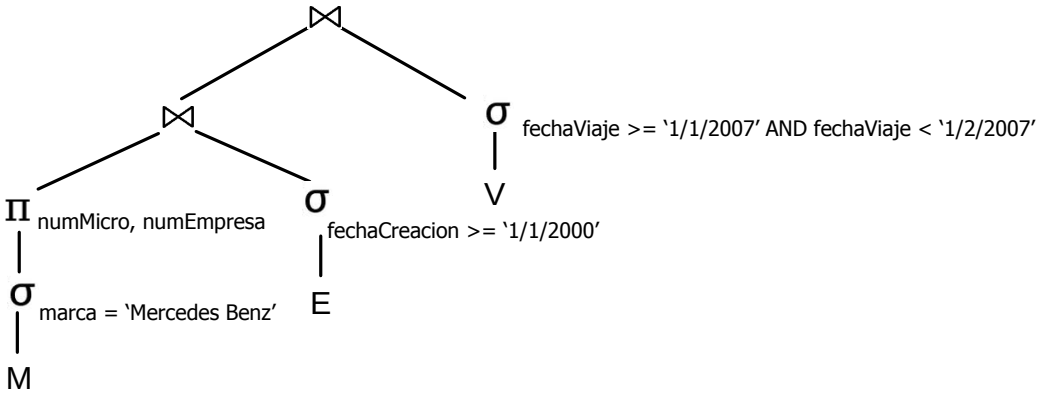
Paso 5

Bajar las selecciones hasta las hojas.



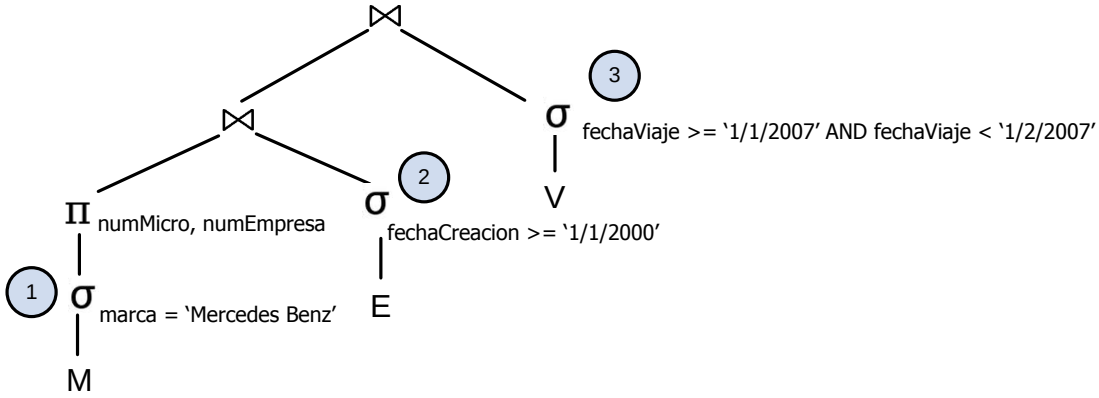
Paso 6

Proyectar solamente los atributos que se usan.



Paso 7

Analizar "utilidad" de las selecciones.



Select (1):

- Reduce considerablemente cantidad de tuplas +
- No uso índice –
- No pierdo índices +

Conclusión: Lo uso.

Select (2):

- Reduce poco la cantidad de tuplas –
- Pierdo índices I_1 e I_2 –
- No uso índice –

NOTA: El índice I_1 es sobre fechaCreación pero es hash y no puede usarse para búsquedas por rango.

Conclusión: mmm...

Select (3):

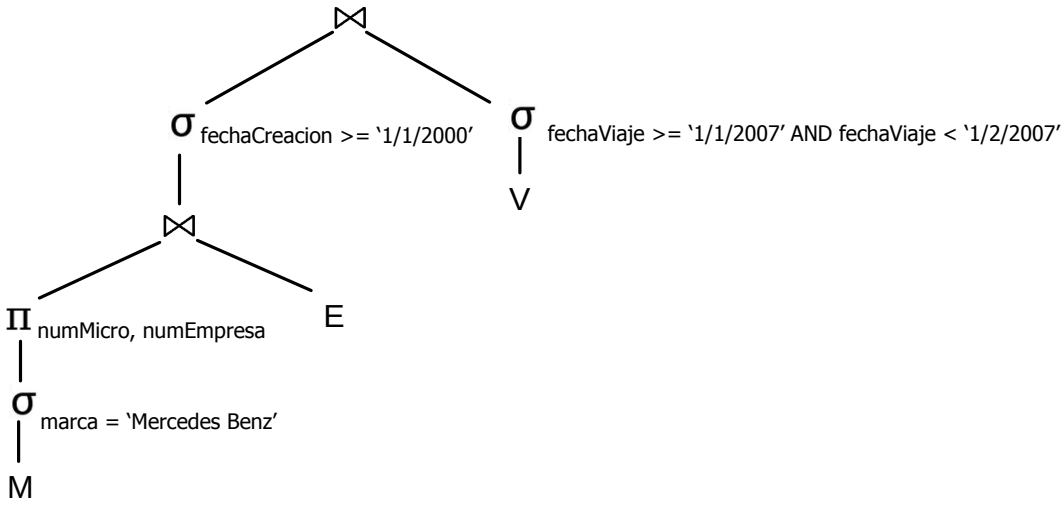
- Reduce considerablemente cantidad de tuplas +
- Pierdo índice clustered I_3 –
- Uso índice I_3 +

NOTA: El índice I_3 lo puedo usar ya que la clave de búsqueda es prefijo del índice.

Conclusión: Lo uso.

Paso 8

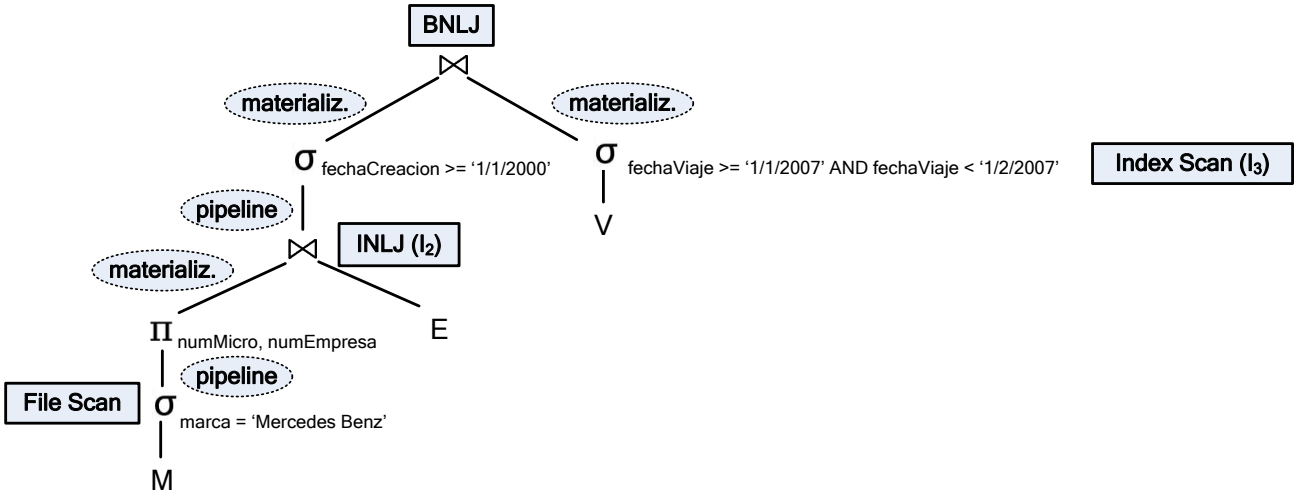
Tratar de analizar alternativas usando el árbol al que se llegó.



En el paso anterior se pudo ver que tal vez bajar el segundo *select* no era tan conveniente. Además, ¿por qué no usar el índice sobre E para el primer *join*? Porque al hacer primero el *select*, ese índice se perdió... Entonces, no hagamos ese *select* hasta después del primer *join*.

Paso 9

Indicar operadores físicos y dónde se usa *pipelining* o *materialización*. De esta forma, queda definido un plan de ejecución.

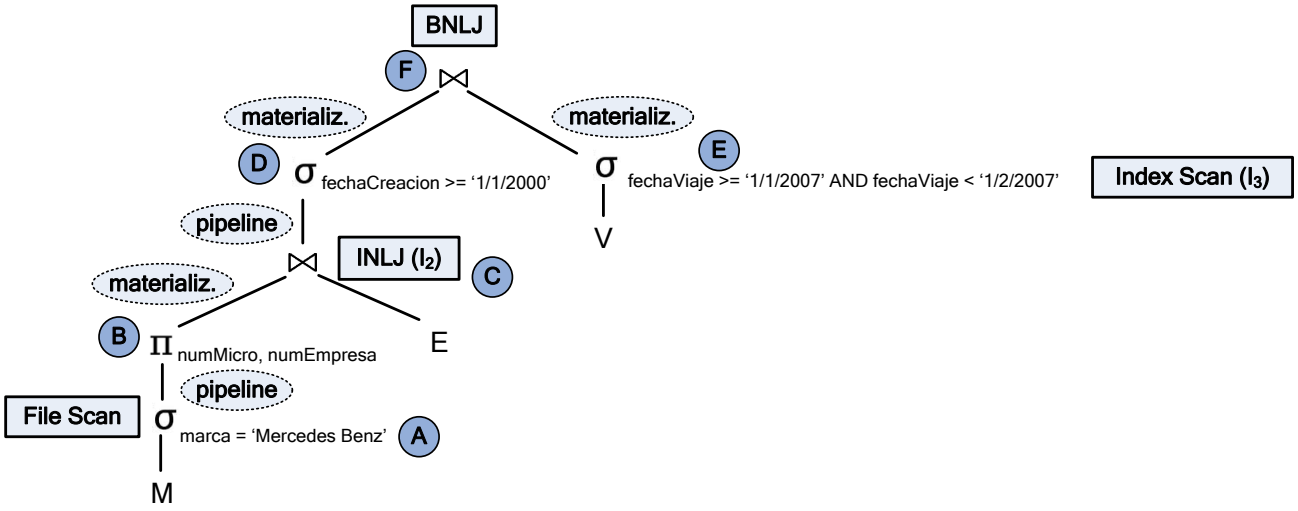


Posibles errores a tener en cuenta:

- No usar índices que podrían utilizarse.
- Perder índices y usarlos después.
- Usar índices hash para consultas de rango.

Paso 10

Calcular los costos:



Iremos calculando A, B, C, D, E y F hasta obtener el costo total.

A)

Calculemos datos del input:

$L_M = 3*128 = 384$

$FB_M = [2048 / 384] = 5$

$T_M = 200.000$

$B_M = 200.000/5 = 40.000$

CI_A (costo input) = B_M = 40.000 (recorrer todo M)

$CI_A = 40.000$

Al devolver el resultado por pipeline, su costo de output es 0.

$CO_A = 0$

B)

Como lee a través del pipeline, su costo de input es nulo.

$CI_B = 0$

Llamemos M' al resultado del select y la proyección:

$L_{M'} = 2*128 = 256$

$FB_{M'} = 2048/256 = 8$

$T_{M'} = 50.000$ (dato)

$B_{M'} = 50.000/8 = 6.250$

CO_B (costo output) = $B_{M'} = 6.250$

$CO_B = 6.250$

C)

Usamos Index Nested Loops Join:

$CI_C = B_{M'} + T_{M'}$*("costo índice E" + 1)

donde "costo índice E" significa el costo de llegar al nodo hoja del índice clustered de E, que es igual a 3 (dato). Además, por cada tupla de M' se va a encontrar una sola tupla de E debido a que el campo del join es clave de E.

Entonces,
$CI_C = B_{M'} + T_{M'}$*("costo índice E" + 1) = $B_{M'} + T_{M'}$*(3+1) = 6.250 + 50.000*4 = 206.250

$CI_C = 206.250$

Al igual que antes, su costo de salida es 0.

$CO_C = 0$

D)
Como lee por el pipeline, no tiene costo de input.

$CI_D = 0$

Llamemos R al resultado del join y al posterior select:

$L_R = 5 * 128 = 640$
$FB_R = [2048 / 640] = 3$

El cálculo de la cantidad de tuplas veámoslo en detalle.
Por un lado, había 50.000 registros de micros en la relación M'. Como numEmpresa es FK (no nula), al hacer el join con E la cantidad de tuplas seguirá siendo 50.000.
Ahora nos falta ver cuántos de esos registros cumplen con la condición de que fechaCreacion >= '1/1/2000'.
Es dato que 2/3 (2.000/3.000) de las empresas fueron creadas después del '1/1/2000'. Como asumimos distribución uniforme, 2/3 de los micros que son 'Mercedez Benz' corresponderán a empresas creadas después del '1/1/2000'. Luego,
$T_R = [50.000*2/3] = 33.333$
$B_R = 33.333/3 = 11.111$

$CO_D = B_R = 11.111$

$CO_D = 11.111$

E)
Datos del input:
$L_V = 3*128 = 384$
$FB_V = [2048 / 384] = 5$
$T_V = 2.500.000$ (dato)
$B_V = 2.500.000/5 = 500.000$

CI_E = "costo índice V" + "cantidad de bloques que contienen tuplas que cumplen con la condición"

"costo índice V" es lo que cuesta llegar hasta la hoja del árbol B+, que es 5 (dato)
También es dato que cumplen 100.000 viajes. Luego, ocupan $100.000/FB_V = 100.000/5 = 20.000$ bloques.
Luego,
$CI_E = 5 + 20.000 = 20.005$

$CI_E = 20.005$

Llamemos V' al resultado del select:
$L_{V'} = 3*128 = 384$
$FB_{V'} = [2048/384] = 5$
$T_{V'} = 100.000$ (dato)
$B_{V'} = 100.000/5 = 20.000$

$CO_E = B_{V'} = 20.000$

$$\boxed{CO_E = 20.000}$$

F)
Se realiza *Block Nested Loops Join*, obteniendo un costo:

$CI_F = B_R + B_{V'} * \lceil B_R/(B-2) \rceil = 11.111 + 20.000 * \lceil 11.111/3 \rceil = 74.091.111$

$$\boxed{CI_F = 74.091.111}$$

¿Qué podemos decir de este último resultado? Sin lugar a dudas, este costo es altísimo con respecto a los valores que veníamos obteniendo. Tal vez un resultado así nos lleve a replantearnos alguna estrategia. En este caso, calculemos cuánto nos costaría hacer un *Sort Merge Join*:

CI_F = " costo ordenar R " + " costo ordenar V' " + " costo Merge"
= $(\lceil \log_{B-1} \lceil B_R/B \rceil \rceil+1)*2B_R + (\lceil \log_{B-1} \lceil B_{V'}/B \rceil \rceil+1)*2B_{V'} + B_R + B_{V'}$
= $(6 + 1)*2*11.111 + (6+1)*2*20.000 + 11.111 + 20.000$
= $155.554 + 280.000 + 31.111$
= 466.665

Con esta estrategia, el costo de este join se reduce más de ¡¡150 veces!!

El costo de output no hace falta calcularlo debido a que para cualquier plan es el mismo.

Por lo tanto, el costo total es:

$C_{Total} = CI_A + CO_A + CI_B + CO_B + CI_C + CO_C + CI_D + CO_D + CI_E + CO_E + CI_F$

Por último, una estructura que tal vez sería útil agregar sería un índice hash sobre la tabla Micro, en el campo marca, de forma de evitarnos realizar un file scan. La razón es que los índices hash son muy útiles cuando se busca por igualdad.