OTIMIZAÇÃO DE CONSULTAS - SQL

Sergio Mergen

Otimização de consultas SQL

- Muitas otimizações de consulta já são feitas pelo próprio SGBD
- No entanto, o desenvolvedor ainda deve tomar alguns cuidados ao criar consultas SQL
- A seguir, algumas dicas que podem melhorar o desempenho das consultas

Tópicos

- Remover colunas desnecessárias
- Usar COUNT(*)
- Explorar índices
- Evitar uso de DISTINCT
- Evitar uso de UNION
- Usar LIMIT
- Forçar ordem das junções
- Isolar trechos independentes de uma consulta

Remover colunas desnecessárias

- O uso de SELECT * faz com que todas as colunas pertencentes às tabelas de uma consulta sejam recuperadas

- As colunas a serem recuperadas representam um custo
  - Ocupam espaço em memória
    - Durante o processamento da consulta
  - Ocupam largura de banda
    - Em casos de envio da resposta pela rede

- Além disso, o SELECT * também exige que o arquivo contendo todas as colunas seja acessado, em vez de usar um caminho mais eficiente.

Remover colunas desnecessárias

- Em muitos casos não são necessárias todas as colunas das tabelas pertencentes a uma consulta
  - Apenas poucas colunas serão de fato consumidas pela aplicação que acessa o banco de dados
- Nesses casos pode-se otimizar a consulta mantendo apenas as colunas que realmente sejam de interesse

Remover colunas desnecessárias

- Ex. A consulta abaixo traz todas as colunas da tabela projeto

```
SELECT * FROM proj
```

- O DBA sabe que essa consulta foi usada para a impressão de um relatório contendo o nome e o tempo de duração de cada projeto
- Nesse caso a seguinte otimização é possível

```
SELECT nome, duração FROM proj
```

Remover colunas desnecessárias

```
SELECT m.*, mc.cast_order
FROM movie m NATURAL JOIN movie_cast2 mc
```

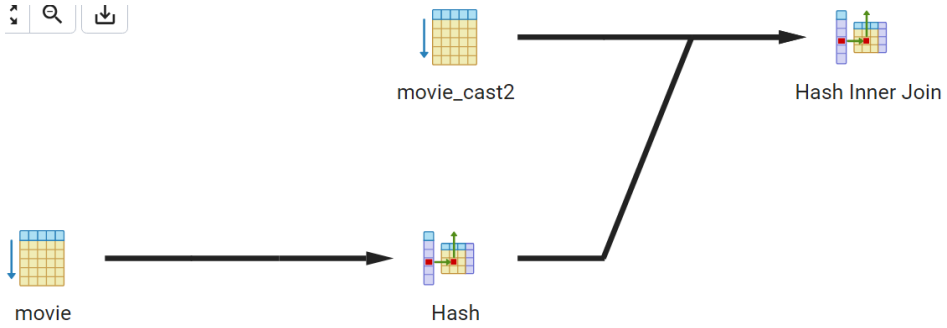
Todas as colunas de movie são requisitadas (m.*).

O PostgreSQL usou Hash Join, deixando movie do lado interno.

O consumo da tabela hash é de 336 kb.

PostgreSQL

```
Hash Join
  Hash Cond: (mc.movie_id = m.movie_id)
  ->  Seq Scan on movie_cast2 mc
  ->  Hash
        Buckets: 8192  Batches: 1  Memory Usage: 336kB
        ->  Seq Scan on movie m
```



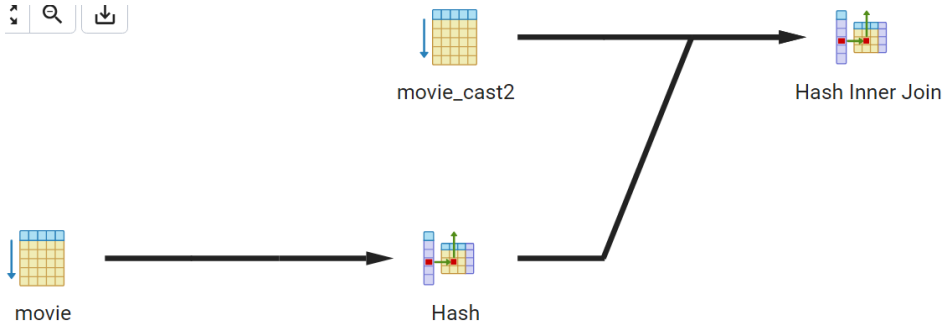
Remover colunas desnecessárias

```
SELECT m.title, mc.cast_order
FROM movie m NATURAL JOIN movie_cast2 mc
```

Quando apenas a coluna title é requisitada, o consumo foi menor (317 kb)

PostgreSQL

```
Hash Join
  Hash Cond: (mc.movie_id = m.movie_id)
  ->  Seq Scan on movie_cast2 mc
  ->  Hash
        Buckets: 8192  Batches: 1  Memory Usage: 317kB
        ->  Seq Scan on movie m
```



Remover colunas desnecessárias

```
SELECT m.*, mc.cast_order
FROM movie m NATURAL JOIN movie_cast mc
WHERE release_year > 2000
```

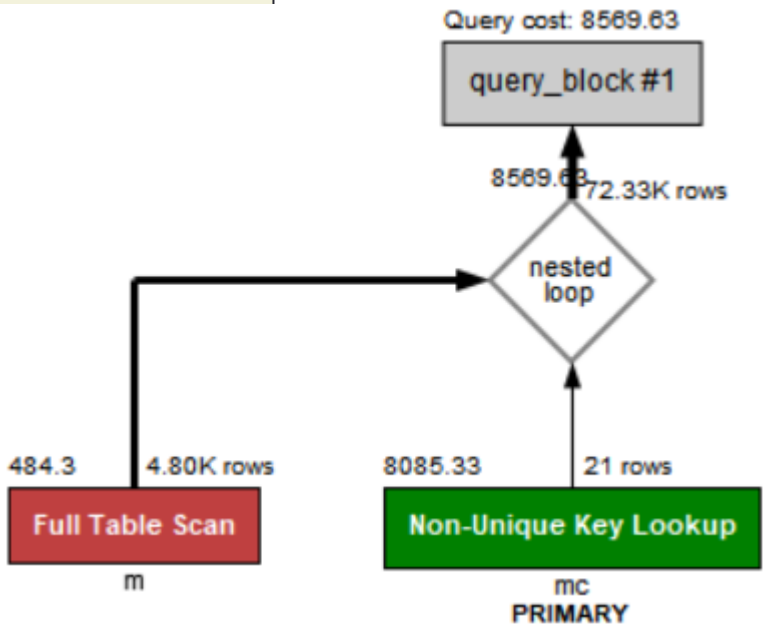
Todas as colunas de movie são requisitadas (m.*).

O MySQL usou Nested Loop, acessando a tabela movie no lado externo

O tempo total foi de 66.7 ms

MySQL

```
-> Nested loop inner join
(actual time=0.0582..66.7 rows=73985 loops=1)
    -> Filter: (m.release_year > 2000)
    (actual time=0.0423..4.93 rows=3328 loops=1)
        -> Table scan on m
        (actual time=0.04..4 rows=4803 loops=1)
    -> Index lookup on mc using PRIMARY (movie_id=m.movie_id)
    (actual time=0.00754..0.0162 rows=22.2 loops=3328)
```



Remover colunas desnecessárias

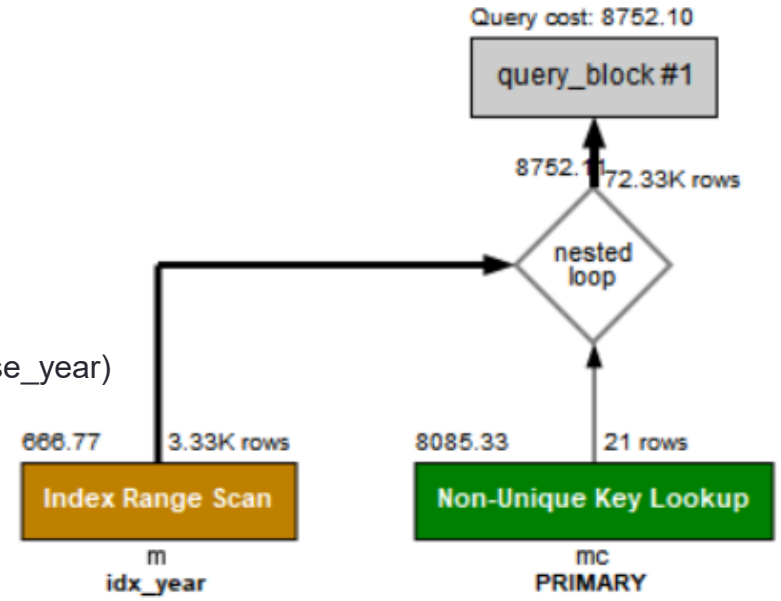
```
SELECT m.release_year, mc.cast_order
FROM movie m NATURAL JOIN movie_cast mc
WHERE release_year > 2000
```

Quando apenas a coluna year foi acessada de movie, o MySQL trocou o table scan por um index range scan no lado externo

O tempo caiu para 56.8 ms

MySQL

-> Nested loop inner join
(actual time=0.0493..56.8 rows=73985 loops=1)
    -> Filter: (m.release_year > 2000)
    (actual time=0.0193..3.12 rows=3328 loops=1)
        -> Covering index range scan on m using idx_year over (2000 < release_year)
        (actual time=0.018..2.31 rows=3328 loops=1)
    -> Index lookup on mc using PRIMARY (movie_id=m.movie_id)
    (actual time=0.00683..0.014 rows=22.2 loops=3328)



Tópicos

- Remover colunas desnecessárias
- Usar COUNT(*)
- Explorar índices
- Evitar uso de DISTINCT
- Evitar uso de UNION
- Usar LIMIT
- Forçar ordem das junções
- Isolar trechos independentes de uma consulta

Usar COUNT(*)

- Duas alternativas para a cláusula count
  - Count(*): retorna quantidade de registros
  - Count(col): retorna a quantidade de valores não nulos para a coluna col

- Encontrar quantidade de registros é mais rápido do que encontrar quantidade de valores não nulos

Usar COUNT(*)

- Ex. A consulta abaixo encontra a quantidade de projetos com custo maior do que 100.000

```
SELECT COUNT(nomeProj) FROM proj WHERE custo > 100.000
```

- Caso nomeProj seja uma coluna opcional, o banco precisa passar por todos os valores de nomeProj para contar os não nulos
- Nesse caso, a consulta abaixo é mais otimizada

```
SELECT COUNT(*) FROM proj WHERE custo > 100.000
```

Usar COUNT(*)

- Caso seja realmente necessário contar os não nulos, deixe isso claro na consulta

- Ex. Para encontrar quantos projetos têm custo definido
  - Em vez de

```
SELECT COUNT(custo) FROM proj
```

  - Use

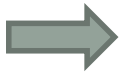
```
SELECT COUNT(*) FROM proj WHERE custo IS NOT NULL
```

Usar COUNT(*)

- A contagem de colunas é importante quando se deseja eliminar valores duplicados
  - Nesse caso, deve-se usar o count distinct
- Ex.
  - Encontrar o número de funcionários alocados em projetos.

Alocação		
idProj	**idFunc**	**função**
1	1	Coordenador
1	3	Analista
2	1	Coordenador

```
SELECT COUNT (DISTINCT idFunc) AS alocados
FROM alocacao
```



Alocados
2

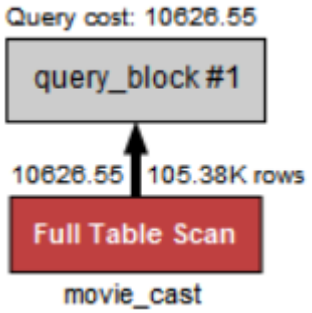
Usar COUNT(*)

SELECT COUNT(**character_name**) FROM **movie_cast**
WHERE **cast_order > 100**

Com count(character_name), o MySQL usou uma função de agregação que analisa a presença de nulos

Isso levou a um tempo total de 78.8 ms

MySQL
-> Aggregate: count(movie_cast.character_name)
(actual time=78.8..78.8 rows=1 loops=1)
    -> Filter: (movie_cast.cast_order > 100)
    (actual time=0.7..78.6 rows=1653 loops=1)
        -> Table scan on movie_cast
        (actual time=0.678..69.2 rows=106084 loops=1)



Usar COUNT(*)

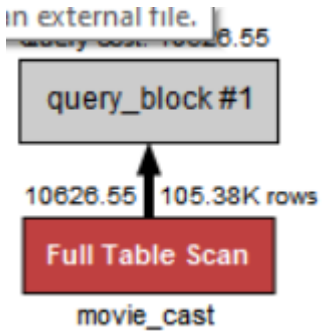
```
SELECT COUNT(*) FROM movie_cast
WHERE cast_order > 100
```

Com count(*), o MySQL usou uma função de agregação mais simples

Isso reduziu o custo a 48.8 ms

MySQL

```
-> Aggregate: count(0)
   (actual time=48.8..48.8 rows=1 loops=1)
        -> Filter: (movie_cast.cast_order > 100)
        (actual time=0.0794..48.7 rows=1653 loops=1)
             -> Table scan on movie_cast
             (actual time=0.0654..39.6 rows=106084 loops=1)
```



Tópicos

- Remover colunas desnecessárias
- Usar COUNT(*)
- Explorar índices
- Evitar uso de DISTINCT
- Evitar uso de UNION
- Usar LIMIT
- Forçar ordem das junções
- Isolar trechos independentes de uma consulta

Explorar índices

- Os índices são estruturas muito importantes para acelerar consultas

- É possível sugerir que o SGBD use um índice
  - FORCE INDEX

- A seguir, veremos que existem outras formas de encorajar o uso de um índice
  - através de adequações nas consultas que vão além do uso da cláusula FORCE INDEX

Explorar índices na cláusula group by

- A cláusula group by pode se valer da existência de índices sobre as colunas de agrupamento para formar os grupos de modo mais eficiente

- Assim, sempre que possível, use essas colunas para estabelecer os critérios de agrupamento

Explorar índices na cláusula group by

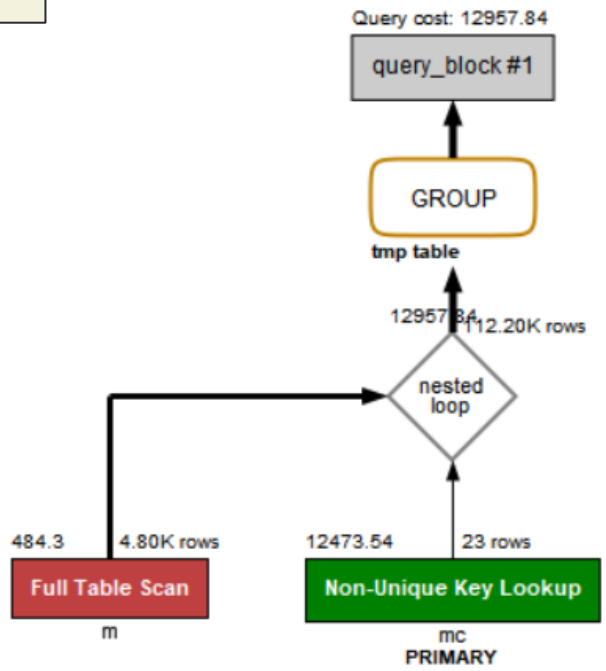
```
SELECT title, COUNT(*)
FROM movie m NATURAL JOIN movie_cast mc
GROUP BY m.title
```

Agrupando por title, o MySQL precisou usar uma tabela temporária para criar os agrupamentos.

Além do consumo de memória, isso levou a um tempo de execução de 172 ms

MySQL

```
-> Table scan on <temporary>
(actual time=171..172 rows=4758 loops=1)
    -> Aggregate using temporary table
    (actual time=171..171 rows=4758 loops=1)
        -> Nested loop inner join (actual time=0.158..78.8 rows=106084 loops=1)
            -> Table scan on m
            (actual time=0.118..2.89 rows=4803 loops=1)
            -> Covering index lookup on mc using PRIMARY (movie_id=m.movie_id)
            (actual time=0.00684..0.0139 rows=22.1 loops=4803)
```



Explorar índices na cláusula group by

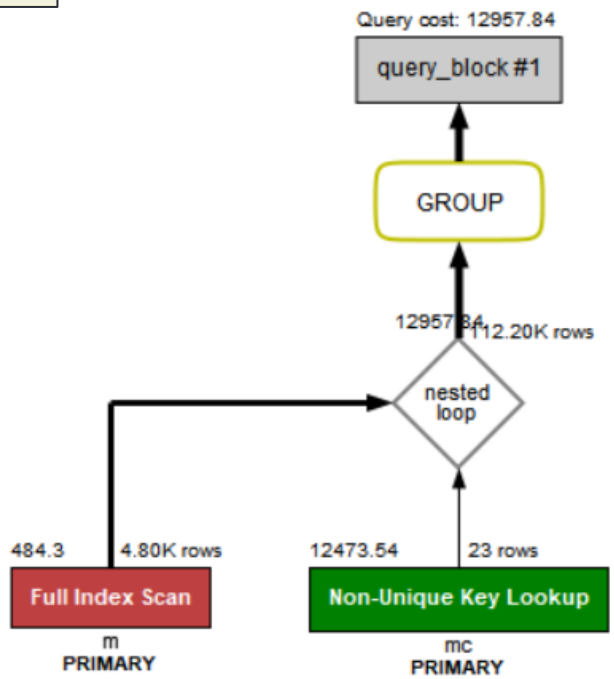
```
SELECT title, COUNT(*)
FROM movie m NATURAL JOIN movie_cast mc
GROUP BY m.movie_id
```

Agrupando por movie_id, não foi necessário recorrer a uma tabela temporária.

Além disso, o tempo total foi reduzido para 115 ms

MySQL

```
-> Group aggregate: count(0)
(actual time=0.179..115 rows=4760 loops=1)
    -> Nested loop inner join
    (actual time=0.119..100 rows=106084 loops=1)
      -> Index scan on m using PRIMARY
      (actual time=0.0867..3.71 rows=4803 loops=1)
          -> Covering index lookup on mc using PRIMARY (movie_id=m.movie_id)
          (actual time=0.0102..0.0178 rows=22.1 loops=4803)
```



Explorar índices na varredura da tabela

- Na consulta abaixo, o MySQL acaba varrendo a tabela inteira para encontrar a resposta

```
SELECT MIN(idFunc) FROM func
WHERE salario = 2000
```

- O formato abaixo se vale da informação de que o idFunc está ordenado no índice primário
  - Assim que for encontrado o primeiro registro com salario = 2000, a busca pode ser interrompida

```
SELECT idFunc FROM func USE INDEX (PRIMARY)
WHERE salario = 2000
LIMIT 1
```

Explorar índices na varredura da tabela

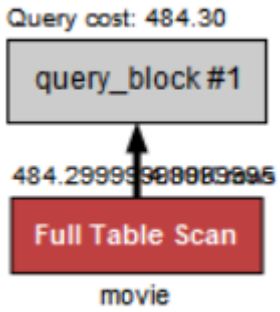
```
SELECT MIN(movie_id) FROM movie
WHERE release_year = 2000
```

Usando a função MIN, o MySQL percorre toda a tabela, faz o filtro e atualiza o menor valor de movie_id, se necessário.

O tempo total foi de 4.7 ms

MySQL

```
-> Aggregate: min(movie.movie_id)
(actual time=4.7..4.7 rows=1 loops=1)
    -> Filter: (movie.release_year = 2000)
    (actual time=0.318..4.65 rows=166 loops=1)
        -> Table scan on movie
        (actual time=0.312..4.1 rows=4803 loops=1)
```



Explorar índices na varredura da tabela

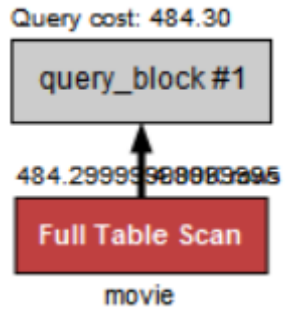
```
SELECT movie_id FROM movie USE INDEX (PRIMARY)
WHERE release_year = 2000
LIMIT 1
```

Explorando o índice de forma mais direta, não é mais necessário percorrer toda a tabela.

Agora o tempo total é de 0.0656 ms

MySQL

```
-> Limit: 1 row(s)
(actual time=0.0655..0.0656 rows=1 loops=1)
    -> Filter: (movie.release_year = 2000)
    (actual time=0.0644..0.0644 rows=1 loops=1)
        -> Table scan on movie
        (actual time=0.0597..0.0619 rows=6 loops=1)
```



Tópicos

- Remover colunas desnecessárias
- Usar COUNT(*)
- Explorar índices
- Evitar uso de DISTINCT
- Evitar uso de UNION
- Usar LIMIT
- Forçar ordem das junções
- Isolar trechos independentes de uma consulta

Evitar uso do DISTINCT

- O uso do DISTINCT indica ao SGBD que o resultado da consulta não pode trazer duplicatas
- Uma das estratégias que o SGBD emprega para garantir isso é a ordenação seguida da remoção dos registros consequentes idênticos
- O problema é que a ordenação pode ser uma tarefa custosa

Evitar uso do DISTINCT

- Em alguns casos, o uso de DISTINCT é desnecessário
- Isso ocorre quando se sabe que a resposta não poderá conter duplicatas pela característica das colunas retornadas
- Nesse caso, a remoção do DISTINCT torna a consulta mais rápida
- Pode-se inclusive usar a cláusula ALL para deixar claro que não é preciso remover duplicatas

Evitar uso do DISTINCT

- Ex. a consulta abaixo traz o nome e a duração de projetos
  - O resultado não deve conter duplicatas

```
SELECT DISTINCT idProj, duracao
FROM proj
```

- O DBA sabe que não existem projetos com mesmo id na base
  - Desse modo, a seguinte otimização é possível

```
SELECT ALL idProj, duracao
FROM proj
```

Evitar uso do DISTINCT

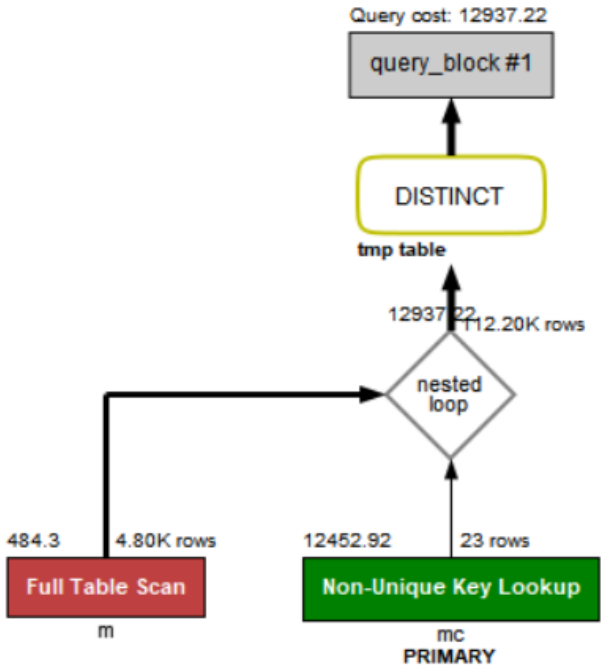
```
SELECT DISTINCT  m.movie_id, m.title, mc.person_id, mc.character_name
FROM movie m NATURAL JOIN movie_cast mc
```

Usando DISTINCT, o MySQL precisou recorrer a uma tabela temporária para realizar a remoção de duplicatas

Além do consumo de memória, isso levou a um tempo total de 249 ms

MySQL

```
-> Table scan on <temporary>
(actual time=235..249 rows=106084 loops=1)
    -> Temporary table with deduplication
    (actual time=235..235 rows=106084 loops=1)
        -> Nested loop inner join
        (actual time=0.057..80.6 rows=106084 loops=1)
            -> Table scan on m
            (actual time=0.0372..2.76 rows=4803 loops=1)
            -> Index lookup on mc using PRIMARY (movie_id=m.movie_id)
             (actual time=0.00722..0.0146 rows=22.1 loops=4803)
```



Evitar uso do DISTINCT

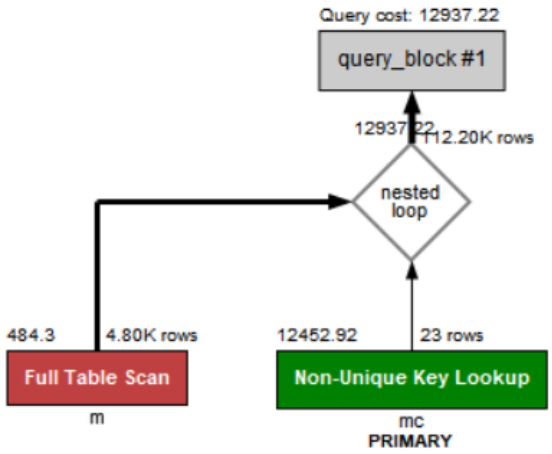
```
SELECT ALL  m.movie_id, m.title, mc.person_id, mc.character_name
FROM movie m NATURAL JOIN movie_cast mc
```

Usando ALL, a remoção de duplicatas foi dispensada.

O resultado gerado é o mesmo, porém o tempo total foi menor: 73.9 ms

MySQL

-> Nested loop inner join
(actual time=0.112..73.9 rows=106084 loops=1)
    -> Table scan on m
    (actual time=0.0751..2.3 rows=4803 loops=1)
    -> Index lookup on mc using PRIMARY (movie_id=m.movie_id)
    (actual time=0.00645..0.0133 rows=22.1 loops=4803)



Evitar uso do DISTINCT

- Analise se pode-se eliminar o **DISTINCT** no caso abaixo

- Solicitação: Exibir o nome dos funcionários.

```
SELECT DISTINCT nome
FROM func
```

Obs. Uma regra de negócio impede
- o registro de funcionários com o mesmo nome
- o registro de projetos com o mesmo nome

Evitar uso do DISTINCT

- Analise se pode-se eliminar o **DISTINCT** no caso abaixo

- Solicitação: Exibir o nome dos funcionários alocados em projetos com duração de 3 anos.

```
SELECT DISTINCT func.nome
FROM func NATURAL JOIN aloc NATURAL JOIN proj
WHERE proj.duracao > 3
```

Obs. Uma regra de negócio impede
- o registro de funcionários com o mesmo nome
- o registro de projetos com o mesmo nome

Evitar uso do DISTINCT

- Analise se pode-se eliminar o **DISTINCT** no caso abaixo

- Solicitação: Exibir o nome dos funcionários alocados no projeto chamado “ACME”

```
SELECT DISTINCT func.nome
FROM func NATURAL JOIN aloc NATURAL JOIN proj
WHERE proj.nome = ‘ACME’
```

Obs. Uma regra de negócio impede
- o registro de funcionários com o mesmo nome
- o registro de projetos com o mesmo nome

Tópicos

- Remover colunas desnecessárias
- Usar COUNT(*)
- Explorar índices
- Evitar uso de DISTINCT
- Evitar uso de UNION
- Usar LIMIT
- Forçar ordem das junções
- Isolar trechos independentes de uma consulta

Evitar uso do UNION

- O uso do UNION indica ao SGBD que o resultado da consulta não pode trazer duplicatas
  - Caso não seja necessário eliminar duplicatas deve-se usar UNION ALL
- Como vimos antes, uma das estratégias que o SGBD emprega para garantir que duplicatas são eliminadas é a ordenação seguida da remoção dos registros consequentes idênticos
- O problema é que a ordenação pode ser uma tarefa custosa

Evitar uso do UNION

- Em alguns casos o uso de UNION é desnecessário
- Isso ocorre quando se sabe que a resposta não poderá conter duplicatas pela característica das colunas retornadas
- Nesse caso, o uso de UNION ALL torna a consulta mais rápida

Evitar uso do UNION

- Ex. a consulta abaixo traz funcionários e clientes de uma empresa

```
SELECT nome FROM func
UNION
SELECT nome FROM cliente
```

- O DBA sabe que nenhum funcionário é também cliente
- Nesse caso, a seguinte otimização é possível

```
SELECT nome FROM func
UNION ALL
SELECT nome FROM cliente
```

Evitar uso do UNION

- A consulta abaixo exibe os nomes e salários de funcionários do departamento 3 ou que tenham trabalhado em projetos vinculados ao departamento 3

```
SELECT nome, salario
FROM func
WHERE id_depto = 3

UNION

SELECT nome, salario
FROM func
NATURAL JOIN aloc
NATURAL JOIN proj
WHERE proj.id_depto = 3
```

Usando UNION

Evitar uso do UNION

- A consulta abaixo exibe os nomes e salários de funcionários do departamento 3 ou que tenham trabalhado em projetos vinculados ao departamento 3

```
SELECT nome, salario
FROM func
WHERE id_depto = 3

UNION ALL

SELECT nome, salario
FROM func
NATURAL JOIN aloc
NATURAL JOIN proj
WHERE proj.id_depto = 3
AND func.id_depto <>3
```

Usando UNION ALL

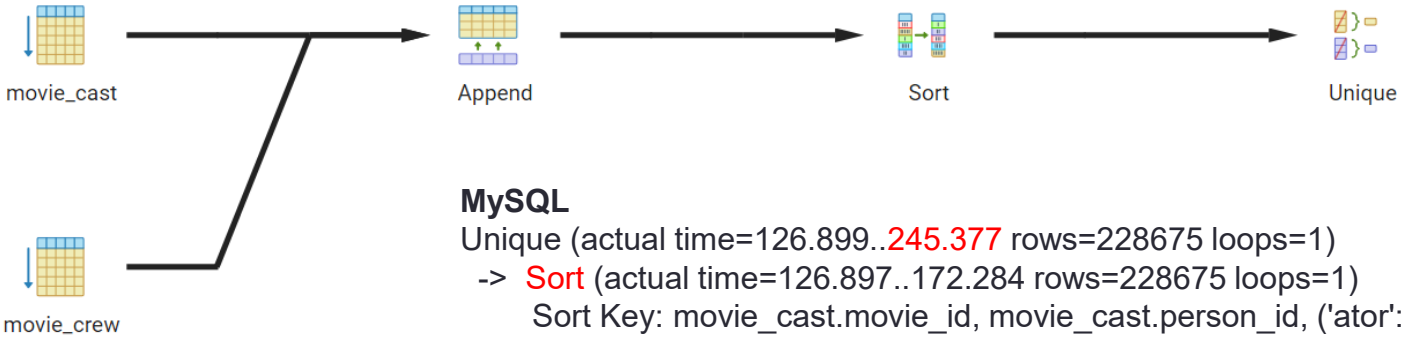
Uma mudança simples garante a ausência de duplicatas

Evitar uso do UNION

```
SELECT movie_id, person_id, 'ator' FROM movie_cast
UNION
SELECT movie_id, person_id, job FROM movie_crew
```

O Union exige remoção de duplicatas.

O PostgreSQL resolve isso com uma ordenação (que consome memória) seguida de uma remoção de entradas iguais consecutivas (unique). O tempo total foi de 245 ms



MySQL

```
Unique (actual time=126.899..245.377 rows=228675 loops=1)
  ->  Sort (actual time=126.897..172.284 rows=228675 loops=1)
        Sort Key: movie_cast.movie_id, movie_cast.person_id, ('ator'::character varying)
        Sort Method: external merge  Disk: 6408kB
        ->  Append
        (actual time=0.021..51.190 rows=228675 loops=1)
              ->  Seq Scan on movie_cast
              (actual time=0.020..18.408 rows=106084 loops=1)
              ->  Seq Scan on movie_crew
              (actual time=0.036..18.208 rows=122591 loops=1)
```

Evitar uso do UNION

```
SELECT movie_id, person_id, 'ator' FROM movie_cast
UNION ALL
SELECT movie_id, person_id, job FROM movie_crew
```

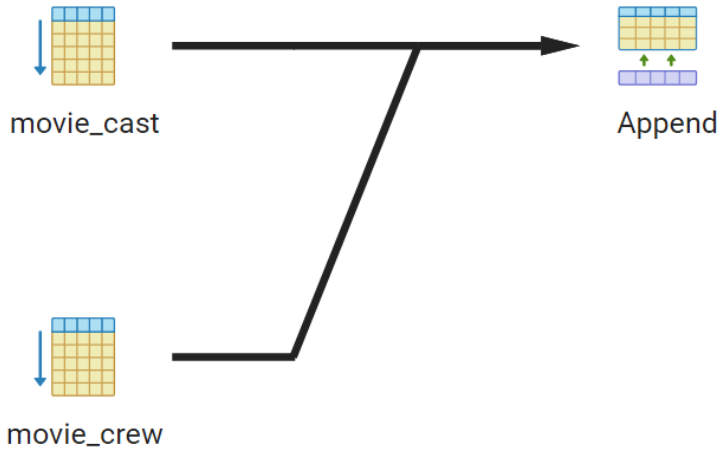
Com UNION ALL, não ocorre a materialização. O append simplesmente junta as tuplas que chegam até ele.

O resultado é o mesmo, pois a última coluna ajuda a diferenciar os registros dos dois selects.

O tempo total foi reduzido para 64 ms

MySQL

```
Append
(actual time=0.032..64.284 rows=228675 loops=1)
    ->  Seq Scan on movie_cast
    (actual time=0.031..21.348 rows=106084 loops=1)
    ->  Seq Scan on movie_crew
    (actual time=0.036..26.818 rows=122591 loops=1)
```



Tópicos

- Remover colunas desnecessárias
- Usar COUNT(*)
- Explorar índices
- Evitar uso de DISTINCT
- Evitar uso de UNION
- Usar LIMIT
- Forçar ordem das junções
- Isolar trechos independentes de uma consulta

Usar LIMIT

- Em muitos casos não são necessários todos os registros que satisfazem o critério de busca
  - Basta recuperar os N registros mais relevantes
  - A relevância geralmente é indicada por uma ordenação de valores
- Nesse caso, pode-se limitar os registros a serem recuperados
  - Através da palavra chave LIMIT
- Isso pode implicar em duas formas de otimização
  - Redução do número de registros em processamento
  - Escolha de algoritmos mais adequados para o processamento de consulta

Usar LIMIT

- Ex. A consulta abaixo traz os nomes e salários dos funcionários, ordenados pelo salário

```
SELECT nome, salario FROM func
ORDER BY salario
```

- O DBA sabe que essa consulta está sendo empregada apenas para obter os dados dos 2 funcionários mais bem pagos
  - Nesse caso, a seguinte otimização é possível

```
SELECT nome, salario FROM func
ORDER BY salario
LIMIT 2
```

Usar LIMIT

- Ex. A consulta abaixo traz os nomes e salários dos funcionários, ordenados pelo salário

```
SELECT nome, salario FROM func
ORDER BY salario
```

- O DBA sabe que essa consulta está sendo empregada apenas para obter os dados dos 2 funcionários mais bem pagos
  - Nesse caso, a seguinte otimização é possível

```
SELECT nome, salario FROM
ORDER BY salario
LIMIT 2
```

Possibilita que o SGBD use um algoritmo mais eficiente que não envolva a ordenação total dos registros

Usar LIMIT

- Ex. A consulta abaixo traz os 20 maiores valores pagos, sejam eles de salario ou custo

```
SELECT salario FROM func
UNION
SELECT custo FROM projeto
ORDER BY 1 DESC
LIMIT 20
```

- Pode-se otimizar essa consulta limitando a quantidade de registros a recuperar já dentro das subconsultas

```
SELECT salario FROM func ORDER BY 1
DESC LIMIT 20
UNION
SELECT custo FROM projeto ORDER BY 1
DESC LIMIT 20
ORDER BY 1 DESC
LIMIT 20
```

Usar LIMIT

```
SELECT *
FROM movie_cast
ORDER BY cast_order;
```

Sem Limit, a o PostgreSQL usou um método de ordenação que passa por todos os registros

Isso levou ao uso de um método de ordenação em disco, o que consumiu 4312 kb

O tempo total de execução foi de 69 ms

PstgreSQL

```
Sort
(actual time=47.069..69.394 rows=106084 loops=1)
  Sort Key: cast_order
  Sort Method: external merge  Disk: 4312kB
    ->  Seq Scan on movie_cast
    (actual time=0.019..6.281 rows=106084 loops=1)
```

Usar LIMIT

```
SELECT *
FROM movie_cast
ORDER BY cast_order
LIMIT 2;
```

Com Limit, o PostgreSQL usou um método de ordenação em memória, pois foi capaz de desprezar muitos valores irrelevantes.

Apenas 25 kb de memória foram consumidos, com um tempo total de execução de apenas 28 ms

PostgreSQL

```
Limit (actual time=28.434..28.436 rows=2 loops=1)
  ->  Sort (actual time=28.433..28.434 rows=2 loops=1)
        Sort Key: cast_order"
        Sort Method: top-N heapsort  Memory: 25kB
        ->  Seq Scan on movie_cast  (actual time=0.019..9.856 rows=106084 loops=1)
```

Tópicos

- Remover colunas desnecessárias
- Usar COUNT(*)
- Explorar índices
- Evitar uso de DISTINCT
- Evitar uso de UNION
- Usar LIMIT
- Forçar ordem das junções
- Isolar trechos independentes de uma consulta

Forçar ordem das junções

- Quando uma consulta possui junções envolvendo mais do que 2 tabelas
  - normalmente é mais eficiente realizar primeiro a junção entre as tabelas que retornarem a menor quantidade de registros
- A ordem das junções é determinada pelo SGBD
- Normalmente a ordem escolhida pelo SGBD é a mais eficiente
  - Com base em estatísticas internas
- No entanto, em alguns casos o SGBD pode errar

Forçar ordem das junções

- Alguns SGBDs permitem que o desenvolvedor especifique a ordem com que as junções devem ser executadas
  - Ex. MySQL: STRAIGHT_JOIN
- Útil quando o SGBD não possui estatísticas precisas
  - E o desenvolvedor possui estimativas mais apuradas

Forçar ordem das junções

- Ex. A consulta abaixo retorna dados de alocação

Estatísticas
- 1.000.000 alocações
- 100.000 funcionários
- 1.000 projetos

```
SELECT nomeF, nomeP
FROM func f
JOIN  aloc a ON  f.idFunc = a.idFunc
JOIN  proj  p ON  p.idProj = a.idProj
WHERE    f.salario >   20.000
AND          p.custo  > 50.000
```

Com base nas estatísticas, a melhor ordem das junções é a seguinte:
1. primeiro proj e aloc
2. depois func

Forçar ordem das junções

- Ex. A consulta abaixo retorna dados de alocação

Estatísticas
1.000.000 alocações
100.000 funcionários
1.000 projetos

```
SELECT nomeF, nomeP
FROM func f
JOIN  aloc a ON  f.idFunc = a.idFunc
JOIN  proj  p ON  p.idProj = a.idProj
WHERE    f.salario >   20.000
AND          p.custo  > 50.000
```

No entanto,
existem muito menos funcionários que recebem acima de 20.000
do que projetos com custo superior a 50.000
E o DBA sabe disso

Forçar ordem das junções

- Ex. A consulta abaixo retorna dados de alocação

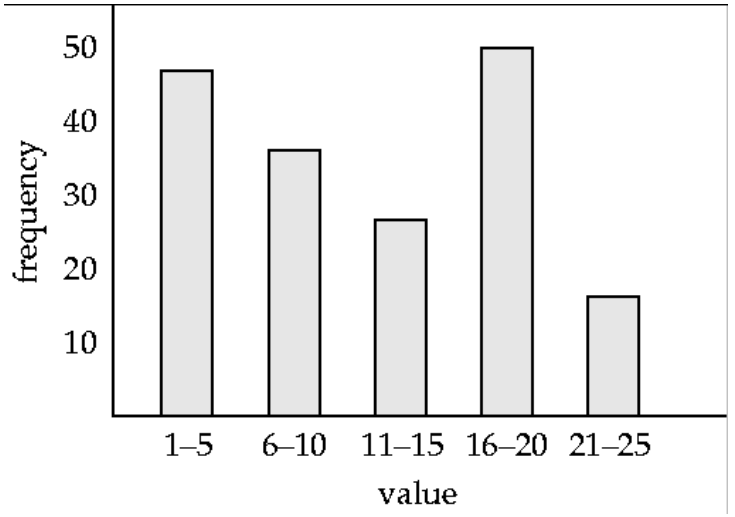
Estatísticas
- 1.000.000 alocações
- 100.000 funcionários
- 1.000 projetos

```
SELECT nomeF, nomeP
FROM func f
STRAIGHT_JOIN  aloc a ON  f.idFunc = a.idFunc
STRAIGHT_JOIN  proj  p ON  p.idProj = a.idProj
WHERE    f.salario >   20.000
AND          p.custo  > 50.000
```

Para evitar escolhas erradas pode-se forçar a ordem das junções
1. primeiro func e aloc
2. depois proj

Forçar ordem das junções

- As estatísticas que o SGBD possuem costumam ser bem completas
  - Número de registros de cada tabela
  - Distribuição dos valores para as colunas das tabelas
    - Usando histogramas
  - E por aí vai...



  - Na maioria dos casos o SGBD irá acertar

Forçar ordem das junções

```
SELECT title, release_year, mc.character_name
FROM movie_cast mc NATURAL JOIN movie m
WHERE cast_order > 220;
```

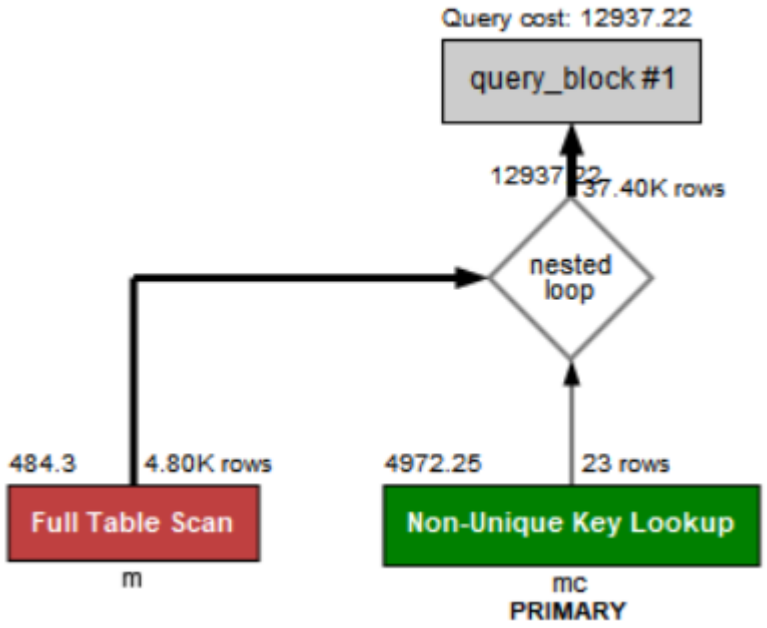
Com junção normal, o MySQL optou por manter o filtro do lado interno.

Ou seja, todos os movies são processados.

Isso levou a um tempo total de 117 ms

MySQL

```
-> Nested loop inner join
(actual time=101..117 rows=4 loops=1)
    -> Table scan on m
    (actual time=0.0927..3.52 rows=4803 loops=1)
    -> Filter: (mc.cast_order > 220)
    (actual time=0.0233..0.0233 rows=833e-6 loops=4803)
        -> Index lookup on mc using PRIMARY (movie_id=m.movie_id)
        (actual time=0.0118..0.0211 rows=22.1 loops=4803)
```

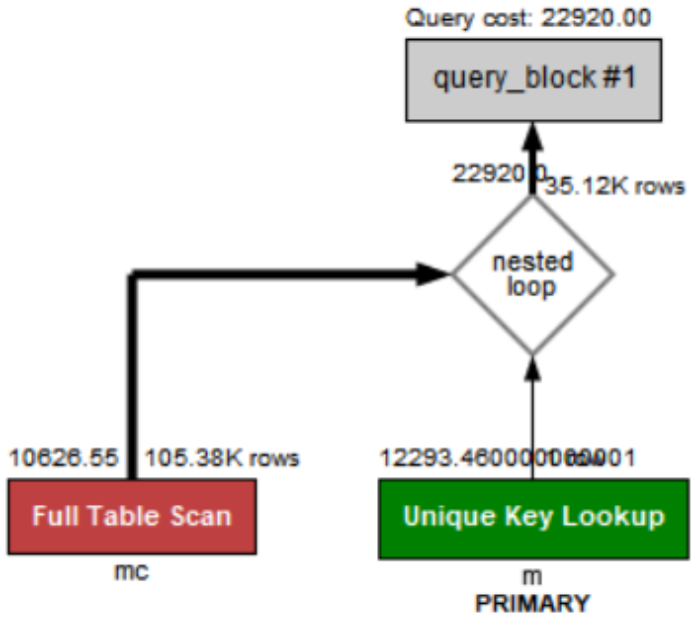


Forçar ordem das junções

```
SELECT title, release_year, mc.character_name
FROM movie_cast mc STRAIGHT_JOIN movie m USING (movie_id)
WHERE cast_order > 220;
```

Forçando a junção a começar por movie_cast, apenas 4 registros tiveram que sofrer junção.

Iso reduziu o tempo de execução para 73 ms



MySQL

```
-> Nested loop inner join
(actual time=58.6..73.6 rows=4 loops=1)
    -> Filter: (mc.cast_order > 220)
    (actual time=58.6..73.5 rows=4 loops=1)
        -> Table scan on mc
        (actual time=0.362..64 rows=106084 loops=1)
    -> Single-row index lookup on m using PRIMARY (movie_id=mc.movie_id)
    (actual time=0.0205..0.0206 rows=1 loops=4)
```

Tópicos

- Remover colunas desnecessárias
- Usar COUNT(*)
- Explorar índices
- Evitar uso de DISTINCT
- Evitar uso de UNION
- Usar LIMIT
- Forçar ordem das junções
- Isolar trechos independentes de uma consulta

Isolar trechos independentes de uma consulta

- Algumas consultas podem fazer junções entre tabelas para atender critérios de filtro que são independentes entre si.
  - Ou seja, quando o conector de disjunção é utilizado (OR)

- Essas junções são desnecessárias, e trazem as seguintes consequências
  - Mais espaço necessário para armazenar o resultado dessas junções
  - Mais processamento necessário para calcular essas junções

Isolar trechos independentes de uma consulta

- Ex.
  - Buscar nomes de projetos com mais do que 20 horas em uma única alocação ou uma única atividade

Estatísticas:
1.000 projetos
10.000 alocações – 10 por proj
50.000 atividades – 50 por proj

A solução abaixo faz junções para satisfazer critérios que são independentes

```
SELECT DISTINCT nomeP
FROM proj
      LEFT JOIN aloc ON proj.idProj = aloc.idProj
      LEFT JOIN ativ  ON proj.idProj = ativ.idProj
      WHERE aloc.horas > 20 OR
                  ativ.horas > 20
```

Isolar trechos independentes de uma consulta

- Ex.
  - Buscar nomes de projetos com mais do que 20 horas em uma única alocação ou uma única atividade

Estatísticas:
1.000 projetos
10.000 alocações – 10 por proj
50.000 atividades – 50 por proj

A presença do OR impede o uso de índices para filtrar as horas

```
SELECT DISTINCT nomeP
FROM proj
      LEFT JOIN aloc ON proj.idProj = aloc.idProj
      LEFT JOIN ativ  ON proj.idProj = ativ.idProj
      WHERE aloc.horas > 20 OR
                  ativ.horas > 20
```

Isolar trechos independentes de uma consulta

- Ex.
  - Buscar nomes de projetos com mais do que 20 horas em uma única alocação ou uma única atividade

Estatísticas:
1.000 projetos
10.000 alocações – 10 por proj
50.000 atividades – 50 por proj

O resultado é uma tabela intermediária com aproximadamente 1.000 * 10 * 50 = 500.000 registros

```
SELECT DISTINCT nomeP
FROM proj
      LEFT JOIN aloc ON proj.idProj = aloc.idProj
      LEFT JOIN ativ  ON proj.idProj = ativ.idProj
      WHERE aloc.horas > 20 OR
                  ativ.horas > 20
```

Isolar trechos independentes de uma consulta

- Ex.
  - Buscar nomes de projetos com mais do que 20 horas em uma única alocação ou uma única atividade

Estatísticas:
1.000 projetos
10.000 alocações – 10 por proj
50.000 atividades – 50 por proj

Essa nova solução usa UNIÃO para separar as junções

```
(SELECT nomeP FROM proj NATURAL JOIN aloc
WHERE aloc.horas > 20)
UNION
(SELECT nomeP FROM proj NATURAL JOIN ativ
WHERE ativ.horas > 20)
```

Isolar trechos independentes de uma consulta

- Ex.
  - Buscar nomes de projetos com mais do que 20 horas em uma única alocação ou uma única atividade

Estatísticas:
1.000 projetos
10.000 alocações – 10 por proj
50.000 atividades – 50 por proj

Essa trecho devolverá no máximo 10.000 registros

Com a aplicação do filtro o tamanho ficará bem menor

```
(SELECT nomeP FROM proj NATURAL JOIN aloc
WHERE aloc.horas > 20)
UNION
(SELECT nomeP FROM proj NATURAL JOIN ativ
WHERE ativ.horas > 20)
```

Isolar trechos independentes de uma consulta

- Ex.
  - Buscar nomes de projetos com mais do que 20 horas em uma única alocação ou uma única atividade

Estatísticas:
1.000 projetos
10.000 alocações – 10 por proj
50.000 atividades – 50 por proj

O mesmo vale para a segunda parte.

Ela retorna no máximo 50.000 registros

```
(SELECT nomeP FROM proj NATURAL JOIN aloc
WHERE aloc.horas > 20)
UNION
(SELECT nomeP FROM proj NATURAL JOIN ativ
WHERE ativ.horas > 20)
```

Isolar trechos independentes de uma consulta

- Ex.
  - Buscar nomes de projetos com mais do que 20 horas em uma única alocação ou uma única atividade

Estatísticas:
1.000 projetos
10.000 alocações – 10 por proj
50.000 atividades – 50 por proj

Com a eliminação do OR, índices podem ser usados para resolver os filtros (caso hajam)

```
(SELECT nomeP FROM proj NATURAL JOIN aloc
WHERE aloc.horas > 20)
UNION
(SELECT nomeP FROM proj NATURAL JOIN ativ
WHERE ativ.horas > 20)
```

Isolar trechos independentes de uma consulta

- Ex.
  - Buscar nomes de projetos com mais do que 20 horas em uma única alocação ou uma única atividade

Estatísticas:
1.000 projetos
10.000 alocações – 10 por proj
50.000 atividades – 50 por proj

O UNION precisou remover duplicatas.

Por outro lado, houve uma redução considerável na tabela de resposta gerada

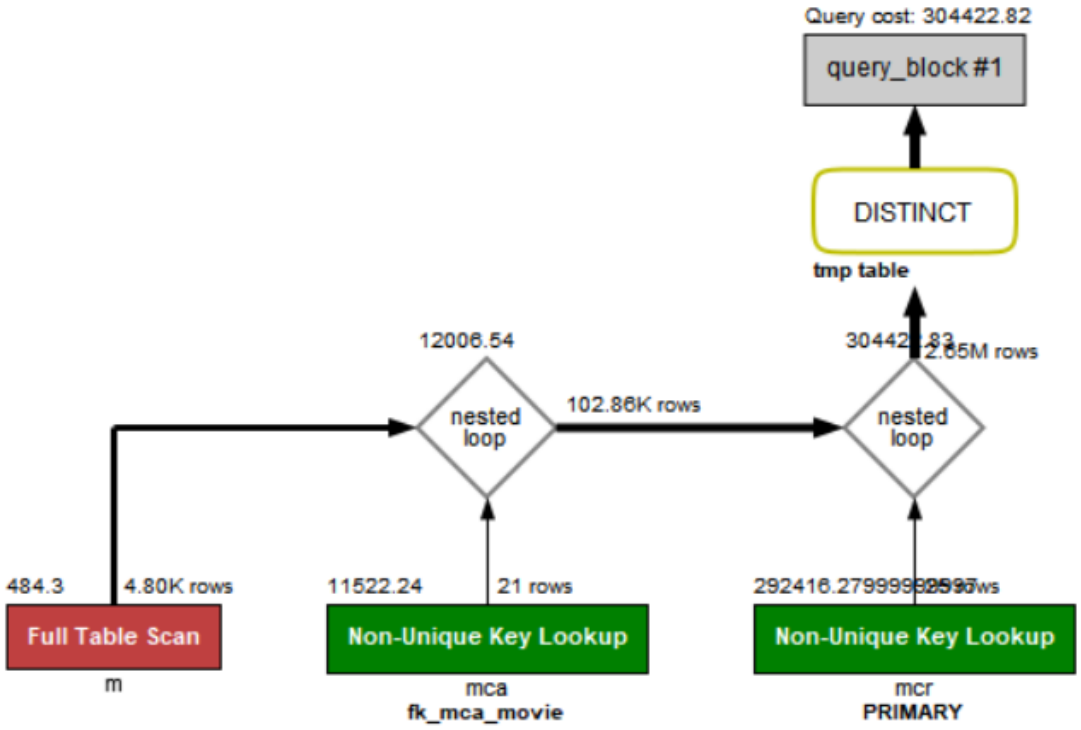
```
(SELECT nomeP FROM proj NATURAL JOIN aloc
WHERE aloc.horas > 20)
UNION
(SELECT nomeP FROM proj NATURAL JOIN ativ
WHERE ativ.horas > 20)
```

Isolar trechos independentes de uma consulta

```
SELECT DISTINCT title FROM movie m JOIN movie_cast mca USING(movie_id)
JOIN movie_crew mcr USING (movie_id)
WHERE mca.person_id = 10 OR mcr.person_id = 10
```

Nesta versão não foi possível realizar a filtragem antecipada, o que levou a 2.65 M de registros após as junções. Isso contribuiu para elevar consideravelmente o tempo total de execução

Tempo: 1706 ms
8 registros

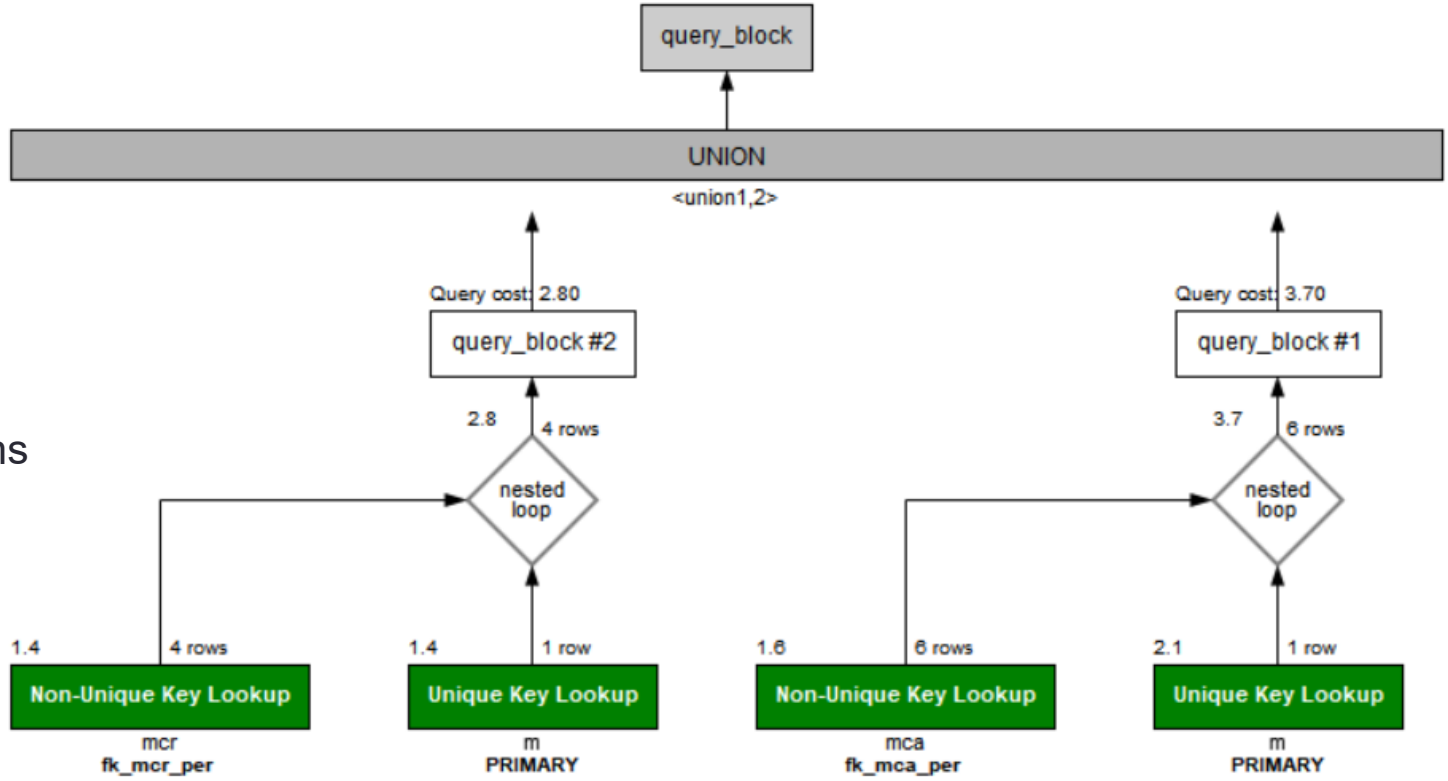


Isolar trechos independentes de uma consulta

```
SELECT title FROM movie m JOIN movie_cast mca USING(movie_id)  WHERE mca.person_id = 10
UNION
SELECT title FROM movie m  JOIN movie_crew mcr USING (movie_id) WHERE  mcr.person_id = 10
```

Separando movie_cast e movie_crew, foi possível usar dois índices. A filtragem antecipada reduziu a quantidade de registros de cada lado da junção

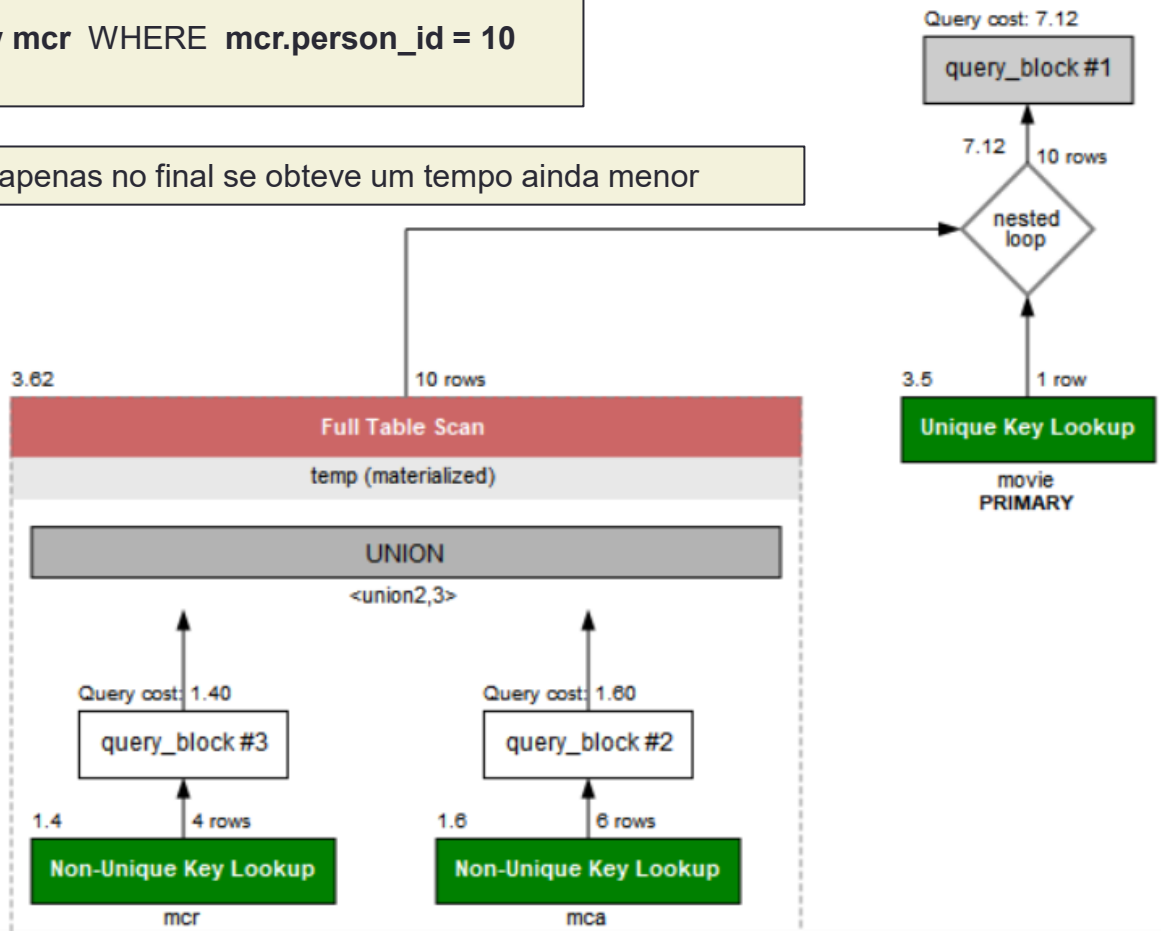
Tempo: 0.182 ms
8 registros



Isolar trechos independentes de uma consulta

```
SELECT title FROM movie JOIN
  (SELECT movie_id FROM movie_cast mca WHERE mca.person_id = 10
  UNION
  SELECT movie_id FROM movie_crew mcr  WHERE  mcr.person_id = 10
) AS temp USING (movie_id)
```

Deixando para fazer a junção com movie apenas no final se obteve um tempo ainda menor



Tempo: 0.0844 ms
8 registros

Exercícios e Atividade Individual

- Os exercícios nos slides seguintes são referentes a um esquema de banco de dados disponível no tópico da aula de hoje.

- Já a atividade individual é referente ao banco de dados de filmes disponibilizado na aula anterior

Exercício 1

- A consulta abaixo foi usada para recuperar os dados de alocação.

```
SELECT * FROM func
      NATURAL JOIN alocacao
      JOIN projeto USING(idproj)
ORDER BY salario DESC;
```

- A aplicação que acessa os dados irá consumir somente os dados de alocação dos funcionários com os dois maiores salários.

- No entanto, as junções acabam sendo feitas para todos os funcionários

- Como resolver isso?

Exercício 2

- Descubra o que a consulta abaixo faz e crie uma nova versão otimizada

```
SELECT a1.idProj,
            MAX(f1.salario - f2.salario) AS maior_diferenca
FROM alocacao a1
  JOIN alocacao a2 ON a1.idProj = a2.idProj
                      AND a1.idFunc <> a2.idFunc
  JOIN func f1 ON a1.idFunc = f1.idFunc
  JOIN func f2 ON a2.idFunc = f2.idFunc
GROUP BY a1.idProj;
```

Atividade Individual

- A consulta abaixo foi usada como exemplo de aula para mostrar que a remoção de duplicatas (DISTINCT) torna a consulta mais onerosa

```
SELECT DISTINCT  m.movie_id, m.title, mc.person_id, mc.character_name
FROM movie m NATURAL JOIN movie_cast mc
```

- O objetivo da atividade é replicar esse exemplo usando o DBest
  - Crie dois planos de execução para a consulta acima
    - Um com e outro sem a etapa de remoção de duplicatas
    - Compare os dois planos usando o recurso ‘Comparator’, presente na ferramenta
    - Envie um PDF mostrando os dois planos gerados e fale a respeito das diferenças encontradas

Atividade Individual

- Operadores necessários para fazer a atividade
  - Aba ‘Rel. Algebra Operators’
    - Join
    - Projection
  - Aba ‘Remove Operators’
    - Hash Duplicate Removal