|  |  |
| --- | --- |
| LogoEfeitrans | **Banco de Dados II Prática 2**  Profa. Vanessa Souza |

**Assunto:** Indexação

Essa prática deve ser executada no SGBD PostgreSQL.

**Questão 1:** Avaliar a relação entre o objeto criado no SGBD e os arquivos físicos.

1. Crie um novo banco de dados no servidor e avalie:

CREATE DATABASE indexacao;

show data\_directory;

* 1. Onde ele foi criado? C:\Program Files\PostgreSQL\13\data\base
  2. Qual seu nome físico? 24668
  3. Qual seu tamanho no disco? 7,71 MB (8.093.696 bytes)

1. Crie uma tabela no banco e avalie:
   1. Onde ela foi criada? C:\Program Files\PostgreSQL\13\data\base\24668
   2. Qual seu nome físico? 24674
   3. Qual seu tamanho no disco? 0 kb
2. Insira registros na tabela e avalie:
   1. Qual seu nome físico? sem
   2. A inserção alterou o tamanho da tabela no disco? Quanto? 8,00 KB (8.192 bytes)

**Questão 2:** Testando o tipo HEAP FILE

1. O comando SELECT \* retorna o dado da forma como ele está no disco

"numerodepto" "nomedepto"

1 "Recursos Humanos"

2 "Producao"

3 "Financeiro"

4 "Almoxarifado"

1. Qual o nome físico de cada registro? Dado pelo ctid:onde o primeiro indica o bloco e o segundo o local/endereço do registro

**SELECT ctid** numerodepto, nomedepto FROM departamento;

"ctid" "numerodepto" "nomedepto"

"(0,1)" 1 "Recursos Humanos"

"(0,2)" 2 "Producao"

"(0,3)" 3 "Financeiro"

"(0,4)" 4 "Almoxarifado"

1. Delete todos os registros da tabela departamento. Qual foi o efeito no arquivo físico?

DELETE FROM departamento;

1. Reinsira os registros em nova ordem (2, 1, 4, 3).

INSERT INTO departamento VALUES (2, 'Producao');

INSERT INTO departamento VALUES (1, 'Recursos Humanos');

INSERT INTO departamento VALUES (4, 'Almoxarifado');

INSERT INTO departamento VALUES (3, 'Financeiro');

1. Reavalie os comandos executados na letra a e b.

"ctid" "numerodepto" "nomedepto"

"(0,5)" 2 "Producao"

"(0,6)" 1 "Recursos Humanos"

"(0,7)" 4 "Almoxarifado"

"(0,8)" 3 "Financeiro"

Continuou a gravação em sequencial a partir da 5 posição do bloco 0. Assim infere-se que os arquivos deletados foram marcados como uma flag de remoção mas ainda continuam nos seus respctivos espaços, porem não podem mais ser acessados e visto pelo banco.

**Questão 3:** Testando o índice primário.

1. A tabela departamento foi criada com PK no atributo numeroDepto. Os índices primários são criados automaticamente durante a criação da tabela. Avalie a criação desse índice na tabela.

**SELECT \* FROM pg\_indexes WHERE tablename =** 'departamento';

* 1. Qual o nome do índice? "indexname" "departamento\_pkey"
  2. Qual o tipo do índice? Indice primário denso
  3. Qual estrutura de dados foi utilizada para criar o índice? A chave primaria numeroDepto

Tipos de índice no PostgreSQL

* *Índice sobre a chave primária*
* *Índice Secundário*
* *Índices Compostos*
* *Índices Parciais*

# Questão 4:

1. Criar e popular a tabela funcionário.

CREATE TYPE sexo\_t AS ENUM ('M', 'F');

CREATE TABLE funcionario(

numeroRegistro SERIAL UNIQUE NOT NULL,

nomeFunc varchar(30) not null,

sexo sexo\_t not null,

dataNasc date not null,

depto int not null,

primary key(numeroRegistro)

)

--inserir 1000000 registros com valores aleatórios na tabela funcionario

insert into funcionario ( nomeFunc, sexo, dataNasc, depto)

select

substr(md5(random()::varchar(30)), 0, 30),

(enum\_range(NULL::sexo\_t))[floor(random() \* (2-1 + 1) + 1)],

date(timestamp '1990-01-01' + random() \* (timestamp '1990-12-31' - timestamp '2000-01-01')),

floor(random()\* (4-1 + 1) + 1)

from generate\_series(1, 1000000) s(i)

SELECT \* FROM funcionario;

* 1. Onde ela foi criada? C:\Program Files\PostgreSQL\13\data\base\24668
  2. Qual seu nome físico? 24708
  3. Qual seu tamanho no disco? 73 MB (76.562.432 bytes)
  4. Endereço dos registros

--PARA LER : https://www.postgresql.org/docs/8.4/storage-fsm.html

SELECT ctid, \* FROM funcionario where numeroregistro = 1000000;

"ctid"= "(9345 blocos, posição 85 do bloco)"

* 1. Índice primário:

**SELECT \* FROM pg\_indexes WHERE tablename =** 'funcionario';

"indexname"= "funcionario\_pkey"

# Questão 5:

1. EXPLAIN ANALYSE x EXPLAIN

**EXPLAIN ANALYSE SELECT \* FROM** funcionario WHERE numeroregistro >10000;

Apresenta mais informações deve ser analisada de baixo pra cima. Tempo de execução da busca (não inclui tempo de apresentação), tempo de planejamento de qual algotimo de busca utilizar, numero de linha removidas, o filtro utilizado e o PLANO DE CONSULTA: com o custo(tempo e linhas) planejado e o real.

**EXPLAIN SELECT \* FROM** funcionario

Apenas apresenta o custo planejado pela busca

# Questão 6:

1. Avaliar a execução e o custo da consulta “SELECT \* FROM funcionario WHERE

depto = 2”

* Método de escaneamento da tabela
* Tempo de execução da consulta

**EXPLAIN ANALYSE SELECT \* FROM funcionario WHERE depto = 2;**

"**Seq Scan on funcionario** (cost=0.00..21846.00 rows=248800 width=46)

(actual time=0.020..86.956 rows=249665 loops=1)"

" Filter: (depto = 2)"

" Rows Removed by Filter: 750335"

**" Planning Time: 0.104 ms"**

" **Execution Time: 92.006 ms"**

* **Realiza a busca filtrando de forma sequencial;**

1. Criar um índice na tabela funcionários sobre o atributo departamento

# Estruturas de Dados Possíveis

* + - B-tree (*default*)
    - Hash
    - GiST - Utilizado em dados espaciais
    - GIN – Utilizado para Índices invertidos e Arrays

# LEITURA OBRIGATÓRIA :

<https://www.postgresql.org/docs/12/indexes.html>

**CREATE INDEX** func\_depto **ON** funcionario **USING btree** (depto);

1. Repetir a análise da questão 6ª

EXPLAIN ANALYSE SELECT \* FROM funcionario WHERE depto = 2;

"**Bitmap Heap Scan on funcionario** (cost=2776.63..15232.63 rows=248800 width=46) (actual time=8.416..39.852 rows=249665 loops=1)"

" Recheck Cond: (depto = 2)"

" **Heap Blocks**: exact=9346"

" -> **Bitmap Index Scan on func\_depto** (cost=0.00..2714.43 rows=248800 width=0)

(actual time=7.437..7.437 rows=249665 loops=1)"

" Index Cond: (depto = 2)"

**" Planning Time: 0.436 ms" diminui**

**" Execution Time: 44.493 ms" diminuiu**

* Melhora o desempenho utilizando o indice secundário btree func\_depto.
* Realiza a busca utilizando indice de departarmento criado onde ao varrer todos os registros encontrados via indice aqueles que forem pertencentes ao departamento serão marcados pelo bitmap. A lista de registros remarcados é repassada para ser bucada sequencialemente pelo SGBD no banco que refaz a reverificão da condição do departamento pertencente e, por fim, traz os dados requeridos pela na busca.

# Questão 7:

1. Avaliar a execução e o custo da consulta “SELECT \* FROM funcionario WHERE sexo = M AND depto = 3”
   * Método de escaneamento da tabela
   * Tempo de execução da consulta

**EXPLAIN ANALYSE SELECT \* FROM funcionario WHERE sexo = 'M' AND depto = 3**

"**Bitmap Heap Scan on funcionario** (cost=2774.11..15892.12 rows=126731 width=46)

(actual time=8.355..47.046 rows=124701 loops=1)"

" Recheck Cond: (depto = 3)"

" Filter: (sexo = 'M'::sexo\_t)"

" Rows Removed by Filter: 125382"

" **Heap Blocks:** exact=9346"

" -> **Bitmap Index Scan on func\_depto** (cost=0.00..2742.43 rows=251467 width=0) (actual time=7.430..7.431 rows=250083 loops=1)"

" Index Cond: (depto = 3)"

**" Planning Time: 0.190 ms"**

**" Execution Time: 49.583 ms"**

Utiliza o indice com o bitmap para selecionar apenas os do departemento 3 que em seguida serão buscado sequencialmente, mas reverificados em relação ao pertecimento do depertamento 3 e ainda filtrando somente aqueles que são do sexo masculino.

1. Apague o índice criado na questão 6B

**DROP INDEX func\_depto;**

1. Repetir a análise da questão 7ª

EXPLAIN ANALYSE SELECT \* FROM funcionario WHERE sexo = 'M' AND depto = 3

"**Seq Scan on funcionario (**cost=0.00..24346.00 rows=126731 width=46)

(actual time=0.012..75.585 rows=124701 loops=1)"

" **Filter**: ((sexo = 'M'::sexo\_t) AND (depto = 3))"

" Rows Removed by Filter: 875299"

" **Planning Time: 0.260 ms**" aumentou

" **Execution Time: 77.776 ms**" aumentou

* Melhora o desempenho utilizando o indice secundário btree func\_depto.
* Como não ha uma busca sequencial somente dos registros do departamento 3 é realizado uma busca sequecial completa de todos os registros da tabela funcionario, onde sera filtrado apenas aqueles que são do sexo masculino e que tambem são do departamento 3. No anterior era descartado aqueles que nao eram do departamento 3, agilizando assim a busca.

1. Criar um **índice composto** com os atributos sexo e depto.

**CREATE INDEX** func\_sexo\_depto **ON** funcionario **USING btree** (depto, sexo);

1. Repetir a análise da questão 7ª

EXPLAIN ANALYSE SELECT \* FROM funcionario WHERE sexo = 'M' AND depto = 3

"**Bitmap Heap Scan** on funcionario (cost=1731.42..12978.38 rows=126731 width=46)

(actual time=4.897..30.571 rows=124701 loops=1)"

" **Recheck Cond**: ((depto = 3) AND (sexo = 'M'::sexo\_t))"

" **Heap Blocks:** exact=9346"

" **-> Bitmap Index Scan on func\_sexo\_depto** (cost=0.00..1699.73 rows=126731 width=0)

(actual time=3.917..3.917 rows=124701 loops=1)"

" Index Cond: ((depto = 3) AND (sexo = 'M'::sexo\_t))"

**" Planning Time: 0.074 ms**" diminuiu o plano

" **Execution Time: 26.331 ms**" diminuiu a execução

* Melhorou mais o desempenho utilizando o indice composto btree func\_sexo\_depto.
* Agora é utilizado o indice composto por departamento e sexo o bitmap ira pre-selicionar apenas aqueles registros que são do departamente 3 e também do sexo masculino, que posteriormente terão estes requisitos reverificados, mas buscado de forma sequencial, resultando assim em mais rapidez na busca dos dados requeridos.

Questão 8:

1. Criar um **índice parcial** para os funcionários nascidos no ano de 1987

**CREATE INDEX** func\_parcial\_nascimento **ON** funcionario **USING btree** (dataNasc)

**WHERE EXTRACT**(YEAR FROM dataNasc) = 1987

EXPLAIN ANALYSE SELECT \* FROM funcionario WHERE EXTRACT(YEAR FROM dataNasc) = 1987;

"**Bitmap Heap Scan on funcionario** (cost=46.54..8224.88 rows=5000 width=46)

(actual time=5.458..24.222 rows=110673 loops=1)"

" Recheck Cond: (date\_part('year'::text, (datanasc)::timestamp without time zone) = '1987'::double precision)"

" **Heap Blocks:** exact=9346"

" -> **Bitmap Index Scan** on func\_parcial\_nascimento (cost=0.00..45.29 rows=5000

width=0) (actual time=4.532..4.532 rows=110673 loops=1)"

**"Planning Time: 0.107 ms"**

**"Execution Time: 26.641 ms"**

Utiliza o bitmap para pre-selecionar os registros do ano 1987 que posterioremnte são de fato selecionados ao serem reverificados o ano.

**Sem o indice parcial:**

"Gather (cost=1000.00..18137.67 rows=5000 width=46) (actual time=0.794..4316.893 rows=111366 loops=1)"

" Workers Planned: 2"

" Workers Launched: 2"

" -> Parallel Seq Scan on funcionario (cost=0.00..16637.67 rows=2083 width=46)

(actual time=2.704..4149.959 rows=37122 loops=3)"

" Filter: (date\_part('year'::text, (datanasc)::timestamp without time zone) = '1987'::double precision)"

" Rows Removed by Filter: 296211"

"Planning Time: 195.777 ms"

"Execution Time: 4321.752 ms"

**Em uma varredura sequencial paralela, os blocos da tabela serão divididos em intervalos e compartilhados entre os processos cooperantes. Cada processo de trabalho concluirá a varredura de seu intervalo de blocos antes de solicitar um intervalo adicional de blocos.**

1. Criar um índice parcial para a primeira letra do nome dos funcionários

**CREATE INDEX** func\_parcial\_nome **ON** funcionario (left(nomeFunc, 1));

EXPLAIN ANALYSE SELECT nomeFunc FROM funcionario WHERE nomeFunc like 'a%'

" **Gather** (cost=1000.00..15554.43 rows=1 width=30)

(actual time=97.287..102.555 rows=0 loops=1)"

" Workers Planned: 2"

" Workers Launched: 2"

" -> **Parallel Seq Scan** on funcionario (cost=0.00..14554.33 rows=1 width=30) (actual time=41.251..41.251 rows=0 loops=3)"

" Filter: ((nomefunc)::text = 'a%'::text)"

" Rows Removed by Filter: 333333"

" Planning Time: 0.072 ms"

" Execution Time: 63.756 ms"

**Sem ou com o indice o resultado é o mesmo, não ha melhoramento e nem piora da busca, pois o SGBD não utiliza o indice criado, ele sempre realiza o mesmo plano de busca.**

<https://pganalyze-com.translate.goog/docs/explain/other-nodes/materialize?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt-BR&_x_tr_pto=sc&_x_tr_hist=true>

**Monitorando os planos do Postgres EXPLAIN:**

Entender o comportamento e o desempenho de nós de planos individuais (e quando o Postgres os escolhe para um plano) é fundamental para entender o planejamento geral da consulta.

Os tipos de nós podem ser amplamente considerados em três categorias:

* [Nós de verificação](https://pganalyze.com/docs/explain/scan-nodes) : produz linhas a partir de dados da tabela subjacente
* [Nós de junção](https://pganalyze.com/docs/explain/join-nodes) : combine linhas de nós filhos
* [Outros nós](https://pganalyze.com/docs/explain/other-nodes) : Ampla variedade de funcionalidades (por exemplo, agregação, limitação, agrupamento, etc.)

**Os fundamentos do planejamento de consultas do Postgres:** O mecanismo de execução de consulta do PostgreSQL é bastante intrincado, mas é importante entender bem para obter o máximo do seu banco de dados. As consultas SQL são principalmente declarativas: você descreve quais dados gostaria de recuperar, o Postgres descobre um plano de como obtê-lo para você e, em seguida, executa esse plano. Esse processo de planejamento é semelhante a como você planeja uma viagem: quais pontos turísticos você deseja ver? Quando eles estão abertos? Quais voos, trens e carros de aluguel estão disponíveis e quanto custam? Planejamento de consulta é a mesma coisa, mas para seus dados.

A execução da consulta é bastante direta. Vamos examinar em detalhes nós, executores de consulta, planejamento e muito mais.

* **estrutura do plano:** Primeiro, um plano de consulta é uma árvore de nós. Cada nó tem um determinado tipo e pode ter um ou mais nós filhos (cujos nós filhos dependem do tipo de nó). Diferentes tipos de nós se comportam de maneira diferente, mas o mecanismo geral é o mesmo: um nó pai extrai dados de seus filhos linha por linha. Os filhos produzem dados diretamente (por exemplo, lendo-os em tabelas) ou extraem de seus filhos. O próprio executor da consulta extrai as linhas do nó raiz e as retorna como resultado da consulta. Existem várias categorias diferentes de nós que executam essencialmente a mesma função, mas têm diferentes compensações (por exemplo, diferentes tipos de nós de varredura ou nós de junção).
* **Planejamento de consultas:** O planejamento é determinar a melhor disposição dos nós do plano para executar sua consulta e depende de vários fatores.Primeiro - e obviamente o mais importante - é a semântica de sua consulta: o plano precisa retornar exatamente as linhas solicitadas por sua consulta (embora, se houver resultados diferentes que atendam a essa semântica, você poderá obter linhas diferentes todas as vezes: é por isso que você não pode depender em um SELECT \* FROM my\_table LIMIT 1sem um ORDER BY).

Em segundo lugar, estão os índices disponíveis: os índices podem permitir que o Postgres busque dados com muito mais eficiência, e manter os índices corretos para seu esquema e carga de trabalho de consulta requer compreensão da execução da consulta.

Em terceiro lugar, estão as definições de configuração atuais: as [constantes de custo do planejador](https://www.postgresql.org/docs/current/runtime-config-query.html" \l "RUNTIME-CONFIG-QUERY-CONSTANTS) e [as configurações de consumo de recursos](https://www.postgresql.org/docs/current/runtime-config-resource.html) , como work \_ mem (observe que muitas delas podem ser definidas dinamicamente por banco de dados, por usuário, por sessão ou até mesmo por consulta). E, finalmente, estão [as estatísticas](https://www.postgresql.org/docs/current/view-pg-stats.html) que o Postgres reuniu sobre seus dados (seja executando o comando ANALYZE manualmente ou como parte do vácuo manual ou automático). Para chegar a um plano, o Postgres inspeciona sua consulta e avalia uma ou mais alternativas dadas acima. Ele estima um custo para cada nó folha e propaga esses custos para cima na árvore para calcular os custos para os nós internos e, eventualmente, para a raiz. Ele considera vários planos e escolhe o mais barato com base nas configurações de custo, índices disponíveis e heurística com base nas estatísticas coletadas do banco de dados.

* Uma **varredura de heap de bitmap** pega um bitmap de localização de linha gerado por uma [varredura de índice de bitmap](https://pganalyze-com.translate.goog/docs/explain/scan-nodes/bitmap-index-scan?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt-BR&_x_tr_pto=sc) (diretamente ou por meio de uma série de operações de conjunto de bitmap por meio dos nós [BitmapAnd](https://pganalyze-com.translate.goog/docs/explain/other-nodes/bitmap-and?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt-BR&_x_tr_pto=sc) e [BitmapOr](https://pganalyze-com.translate.goog/docs/explain/other-nodes/bitmap-or?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt-BR&_x_tr_pto=sc) ) e procura os dados relevantes. Cada pedaço de um bitmap pode ser exato (apontando diretamente para linhas) ou com perdas (apontando para uma página contendo pelo menos uma linha correspondente ao predicado).
* A **varredura de índice de bitmap** realmente opera em conjunto com uma varredura de heap de bitmap: ela não busca os dados em si. Em vez de produzir as linhas diretamente, a varredura de índice de bitmap constrói um bitmap de possíveis localizações de linhas. Ele alimenta esses dados para um pai Bitmap Heap Scan, que pode decodificar o bitmap para buscar os dados subjacentes, capturando os dados página por página. Você pode pensar em uma varredura de índice de bitmap como um meio termo entre uma varredura sequencial e uma varredura de índice. Como uma varredura de índice, ele verifica um índice para determinar exatamente quais dados ele precisa buscar, mas, como uma varredura sequencial, ele tira proveito dos dados sendo mais fáceis de ler em massa.
* Uma **varredura de índice** usa um índice para localizar uma linha específica ou todas as linhas correspondentes a um predicado. Uma varredura de índice procurará uma única linha por vez (para uma consulta como WHERE id = 1234 ou como a tabela interna em um loop aninhado, procurando a linha correspondente à linha externa atual) ou examinará uma seção de a mesa em ordem. Uma varredura de índice deve primeiro procurar cada linha no índice e, em seguida, verificar os dados reais da tabela para essa entrada de índice. Os dados da tabela devem ser verificados para garantir que a linha encontrada esteja realmente visível para a transação atual e também para buscar as colunas incluídas na consulta que não estão presentes no índice. Por causa disso, uma varredura de índice na verdade tem maior sobrecarga por linha do que uma varredura sequencial: sua vantagem real é que ela permite que você leia apenas algumas das linhas em uma tabela. Se o seu predicado de consulta corresponder exatamente ao índice, a varredura recuperará apenas as linhas correspondentes. Se você tiver um predicado adicional em sua consulta, a varredura de índice pode filtrar as linhas enquanto as lê, exatamente como uma varredura sequencial.
* **Varredura sequencial:** Esta é a maneira mais simples de buscar dados de uma tabela: ela verifica todas as páginas de dados sequencialmente. Como a maioria das outras varreduras, isso pode aplicar um filtro durante a leitura dos dados, mas precisa primeiro ler os dados e depois descartá-los. Uma varredura sequencial não tem como zerar apenas os dados que você deseja: ela sempre lê tudo na tabela. Isso geralmente é ineficiente, a menos que você precise de uma grande proporção da tabela para responder à sua consulta, mas está sempre disponível e, às vezes, pode ser a única opção.
* **Varredura somente de índice:** Isso é muito semelhante a uma varredura de índice, mas os dados vêm diretamente do índice e a verificação de visibilidade é tratada de forma especial, para evitar a verificação total dos dados da tabela. Uma [varredura somente de índice](https://www.postgresql.org/docs/current/indexes-index-only-scans.html) é mais rápida, mas nem sempre está disponível como alternativa a uma varredura de índice regular. Ele tem duas restrições: o tipo de índice deve suportar varreduras somente de índice (o tipo de índice btree comum sempre oferece) e (um tanto obviamente) a consulta deve apenas projetar colunas incluídas no índice. Se você tiver uma consulta SELECT \*, mas na verdade não precisar de todas as colunas, poderá usar uma varredura somente de índice apenas alterando a lista de colunas.
* **Hash:** Lê os dados em uma tabela de hash, onde podem ser facilmente consultados pela chave de hash. Isso é usado para junções de hash e agregações de hash.
* **Aggregate:** Um nó Agg implementa agregação simples ou agrupada. Para agregação agrupada, o Postgres pode trabalhar com entradas pré-classificadas ou não classificadas; a última estratégia usa uma tabela de hash interna.
* **Gather**: Reúna dados de vários trabalhadores— semelhante ao nó Anexar.
* **Materialize:**Materialize o resultado de seu nó filho na memória (para evitar recalcular os valores).
* **BITMAP AND:** Gere um bitmap da interseção de dois bitmaps de local de linha física (ou seja, somente locais que ocorrem em ambos os bitmaps). Os bitmaps podem vir de Bitmap Index Scans ou outros nós filhos BitmapOr ou BitmapAnd. Observe que, devido a limitações internas de implementação, os nós BitmapAnd não rastreiam o número de linhas que produzem. Sua contagem de linhas sempre será listada como "Desconhecida" e não será sinalizada como estimativas incorretas.
* **BITMAP OR**: Gere um bitmap da união de dois bitmaps de local de linha física (isto é, locais que ocorrem em qualquer um dos bitmaps). Os bitmaps podem vir de Bitmap Index Scans ou outros nós filhos BitmapOr ou BitmapAnd. Observe que, devido a limitações internas de implementação, os nós BitmapOr não rastreiam o número de linhas que produzem. Sua contagem de linhas sempre será listada como "Desconhecida" e não será sinalizada como estimativas incorretas.
* **LIMIT:** Obtém dados de um nó filho e produz uma saída classificada, usando a memória, se disponível (dependendo da work\_memconfiguração) ou “derramando” para o disco. Obviamente, isso é necessário se a saída precisar ser classificada, embora às vezes a entrada possa ser examinada de uma maneira já classificada - por exemplo, se a varredura de um índice btree for compatível com a ordem de classificação desejada.
* **Sort**: Recebe dados de um nó filho e produz uma saída classificada, usando a memória, se disponível (dependendo da configuração work\_mem) ou “derramando” para o disco. Obviamente, isso é necessário se a saída precisar ser classificada, embora às vezes a entrada possa ser examinada de uma maneira já classificada - por exemplo, se a varredura de um índice btree for compatível com a ordem de classificação desejada.
* **Gather Merge**: Mescle os resultados das saídas de trabalho pré-classificadas — semelhante ao nó Merge Append.
* **Merge Append:** Mescle os resultados de subplanos pré-classificados para preservar a ordem.
* **Em uma varredura sequencial paralela, os blocos da tabela serão divididos em intervalos e compartilhados entre os processos cooperantes. Cada processo de trabalho concluirá a varredura de seu intervalo de blocos antes de solicitar um intervalo adicional de blocos.**
* **Nested Loop Join: Junção de Loop Aninhado:** Para cada linha da tabela externa, percorra todas as linhas da tabela interna e veja se elas correspondem à condição de junção. Se a relação interna puder ser verificada com um índice, isso poderá melhorar o desempenho de uma junção de loop aninhada. Geralmente, essa é uma maneira ineficiente de processar junções, mas está sempre disponível e, às vezes, pode ser a única opção.