TRABALHO PRÁTICO 2

Código Fonte

- O SGBD criado para a disciplina tem suporte a algumas operações de acesso aos dados
  - Varredura de dados de uma tabela
    - TableScan
    - OrderedScan
  - Algoritmos de junção
    - NesteLoopJoin
    - MergeJoin

Interface Operation

- Todas operações precisam implementar a Interface Operation
- A interface declara as funções necessárias para a comunicação entre operações
  - funções baseadas em iteradores

```
public interface  Operation {
    public abstract void open() throws Exception;
    public abstract Tuple next() throws Exception;
    public abstract boolean hasNext() throws Exception;
    public abstract void close() throws Exception;
}
```

Interface Operation

- Possui duas extensões

- Operações que aceitam uma única operação de entrada devem implementar **UnaryOperation**

```
public interface UnaryOperation extends Operation{
    public Operation getOperation();
}
```

- Operações que aceitam duas operações de entrada devem implementar **BinaryOperation**

```
public interface BinaryOperation extends Operation{
    public Operation getLeftOperation();
    public Operation getRigthOperation();
}
```

Interface Operation

- A chamada ao método next() provoca a geração de uma tupla.
  - public abstract **Tuple** next() throws Exception;

- As operações devem retornar uma instância de Tuple

Classe Tuple

- Representa um registro contendo código e valor

```
public class Tuple {
    public long primaryKey;
    public String content;
}
```

Operação TableScan

- Operação Unária.
  - Provê acesso aos registros de uma tabela

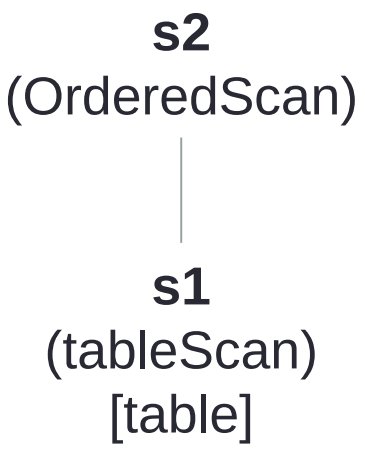
```
Operation scan = new TableScan(table);

scan.open();
while (scan.hasNext()){
      Tuple r =  scan.next();
       System.out.println(r.primaryKey);
       System.out.println(r.content);

}
```

scan
(tableScan)
[table]

Operação OrderedScan

- Operação Unária.
- Dá acesso aos dados ordenados
- Usa materialização em memória

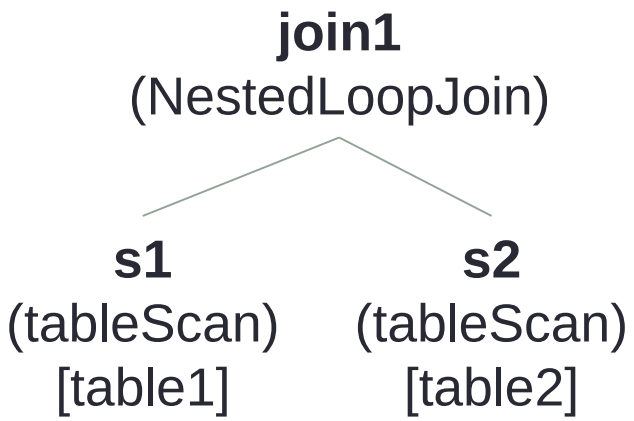
```
Operation s1= new TableScan (table);
Operation s2 = new OrderedScan(s1);

s2.open();
while (s2.hasNext()){
      …
}
```



Operação NestedLoopJoin

- Operação binária.
- Realiza a junção com base em igualdade de chave primária

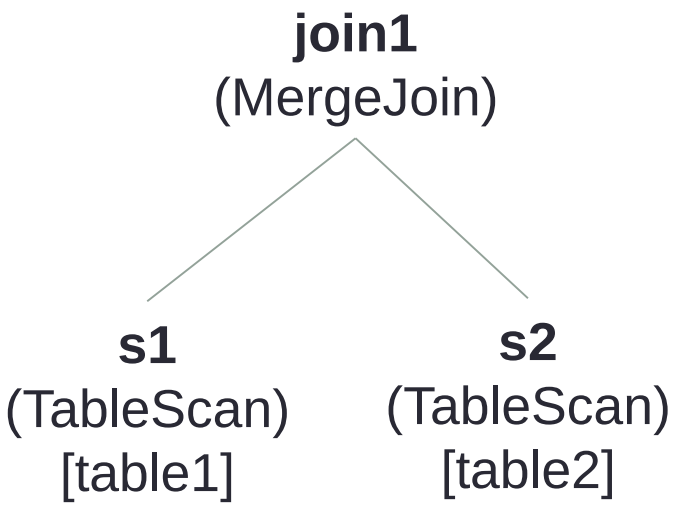
```
Operation s1 = new TableScan(table1);
Operation s2 = new TableScan(table2);

Operation join1 = new
            NestedLoopJoin(s1, s2);

join1.open();
while (join1.hasNext()){
    …
}
```



Operação MergeJoin

- Operação binária
- Os dados já precisam estar ordenados
- Pode ser usado diretamente via **TableScan** se as tabelas já estiverem ordenadas

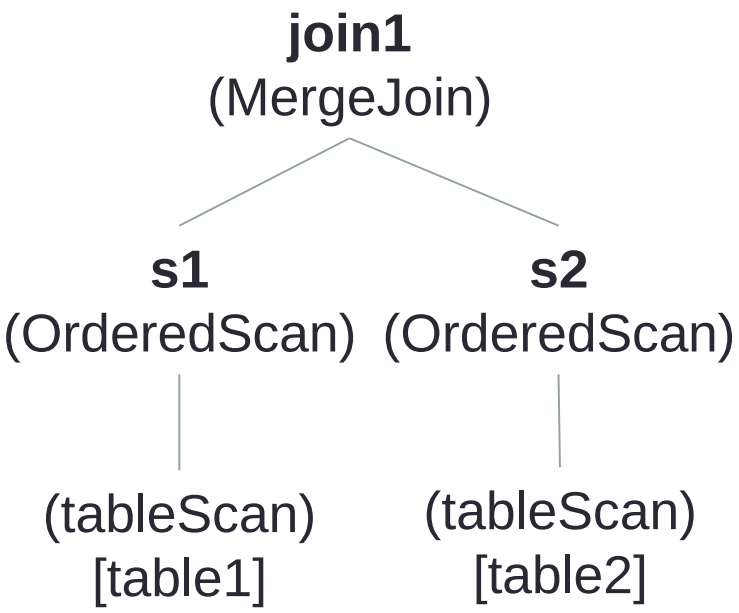
```
Operation s1 = new TableScan(table1);

Operation s2 = new TableScan(table2);

Operation join1 = new
            MergeJoin(s1, s2);
```



Operação MergeJoin

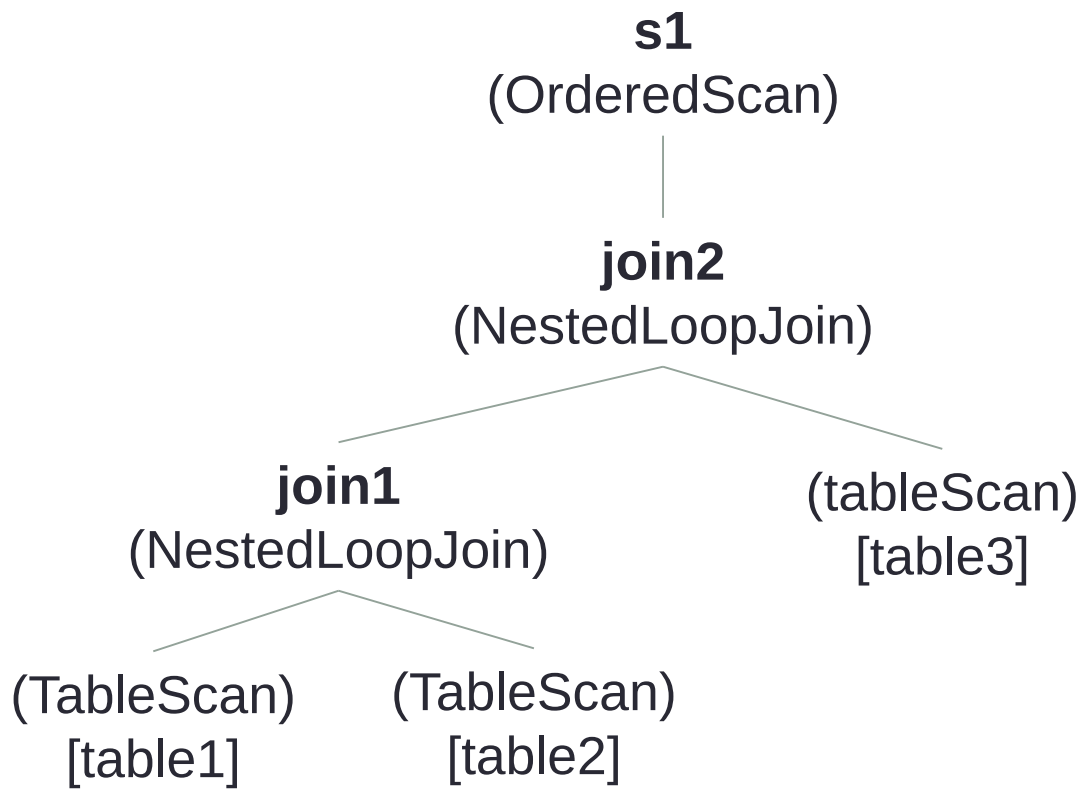
- Operação binária
- Os dados já precisam estar ordenados
- Pode ser alcançado via **OrderedScan** caso os dados estejam desordenados

```
Operation s1 = new OrderedScan (
                    new TableScan(table1));

Operation s2 = new OrderedScan (
                    new TableScan(table2));

Operation join1 = new
          MergeJoin(s1, s2);
```



Composições de Operações

- As operações implementam uma interface comum
  - Isso permite criar composições complexas

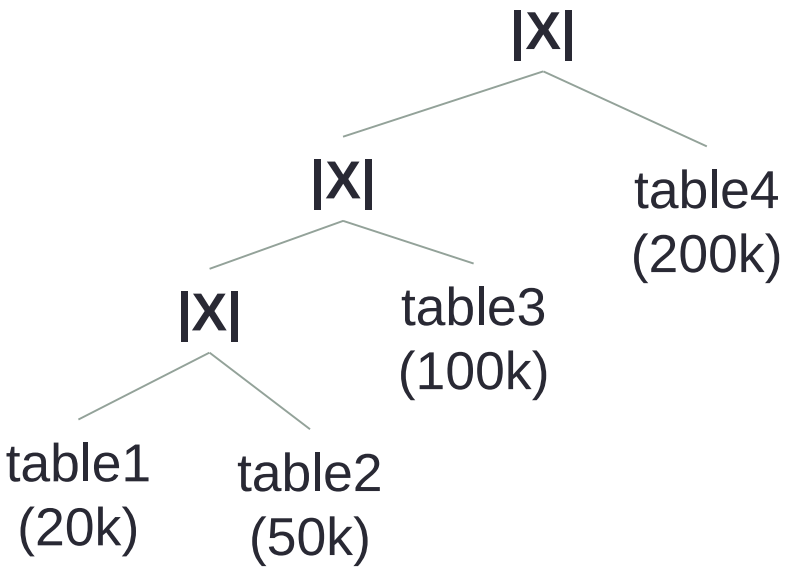


Composições de Operações

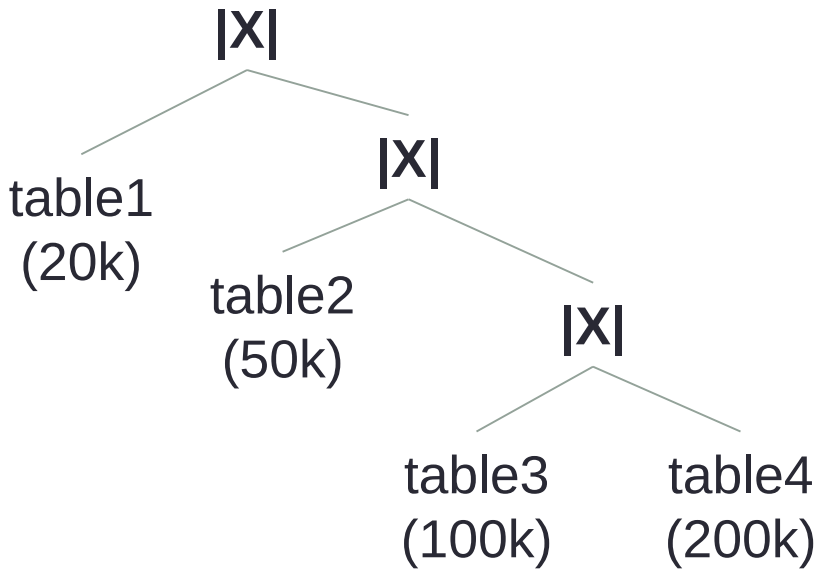
- Existem diferentes formas de combinar as operações em uma árvore de execução
- Algumas costumam ser mais eficientes do que outras.
- Ex.
  - Deixar a tabela menor do lado esquerdo da junção
  - Manter a árvore inclinada para a esquerda

Composições de Operações

- O plano da esquerda deve ser mais eficiente
- Está todo inclinado para a esquerda



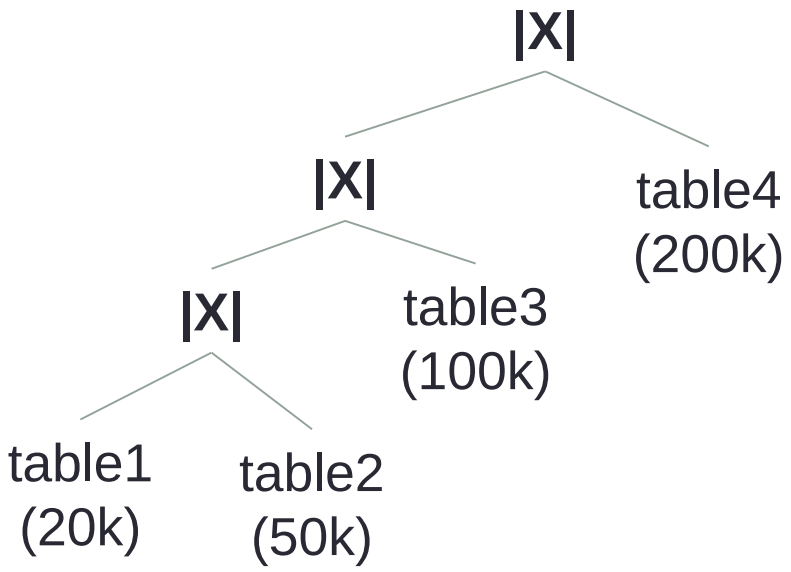
Plano mais eficiente



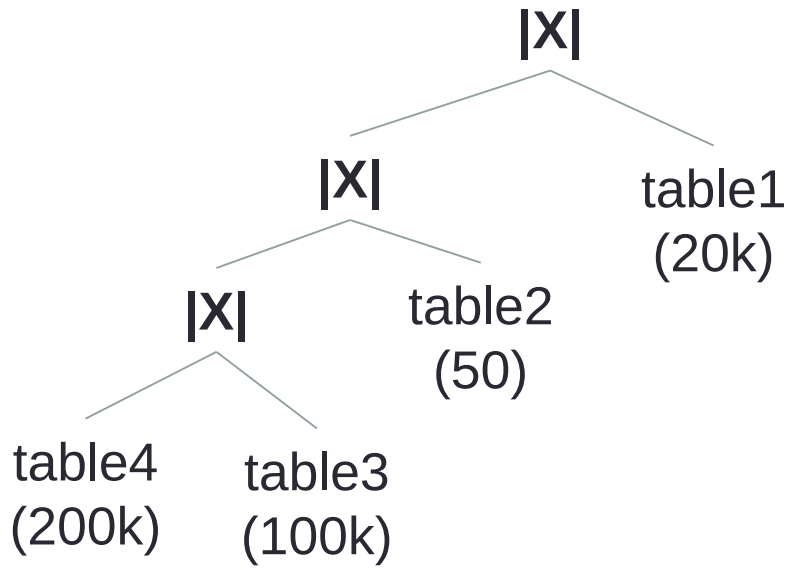
Plano menos eficiente

Composições de Operações

- O plano da esquerda deve ser mais eficiente
- Processa as tabelas menores primeiros



Plano mais eficiente



Plano menos eficiente

Objetivo do Trabalho

- O objetivo do trabalho é criar um otimizador de plano de execução
  - Criar uma classe chamda xxxQueryOptimizer, onde xxx é o nome do aluno
  - Implementar a função pública optimizeQuery
- Exemplo

```
public class SergioQueryOptimizer {

   public Operator optimizeQuery(Operator op) throws Exception{
    …
   }

}
```

Objetivo do Trabalho

- A função **optimizeQuery**() deve receber uma consulta original e a transforma em uma consulta otimizada onde
  - A estrutura dos operadores esteja inclinada para a esquerda
  - As tabelas menores sejam processadas primeiro

- A consulta é representada pelo operador de nível mais alto na árvore

Exemplo de uso

- Para reformular a árvore, o otimizador deve receber o seu operador raiz (de nível mais alto)
  - No exemplo, trata-se do operador **join2**

```
Operation scan1 = new TableScan(table1);
Operation scan2 = new TableScan(table2);
Operation scan3 = new TableScan(table3);
Operation join1 = new MergeJoin(scan1, scan2);
Operation join2 = new MergeJoin(scan3, join1);

SergioQueryOptimizer opt = new SergioQueryOptimizer ();
Operation query = opt.optimizeQuery(join2);

query.open();
while (query.hasNext()){
Tuple r = query.next();
    System.out.println(r.primaryKey + " - "+r.content);
}
```

Considerações

- A única operação unária permitida na árvore é TableScan.
  - Caso outra operação unária seja usada, deve-se retornar uma exceção
- As operações de junções a serem criadas devem ser todas do tipo NestedLoopJoin
  - Independente das operações de junção usadas na árvore original
- A consulta original pode estar em qualquer formato e tamanho

Formas de Verificação

- Verificação de consistência
  - Analise se os registros retornados pela consulta original e a consulta otimizada são os mesmos
- Verificação de desempenho
  - Faça manualmente a otimização, trocando a ordem dos operadores
  - Analise se o número de blocos carregados e salvos usando a consulta otimizada pelo algoritmo e a consulta otimizada manualmente são equivalentes
- Obs. Mantenha o tamanho do buffer de blocos com o tamanho original

Partes úteis do código

- Para fins de otimização, o tamanho de cada tabela pode ser obtido pela função
  - table1.getRecordsAmount()

- Para fins de verificação de desempenho, a quantidade de blocos carregados e salvos pode ser obtida através dos parâmetros
  - Params.BLOCKS_LOADED
  - Params.BLOCKS_SAVED

Entrega

- Prazo final de entrega, sem descontos
  - Domingo, 10 de maio às 23:55
- A cada dia de atraso, a nota é decrementada em 50%.
- O que entregar
  - O código fonte da classe de otimização (.java)