Álgebra Relacional

Inteligência na Web e Big Data

Fabricio Olivetti de França e Thiago Ferreira Covões
folivetti@ufabc.edu.br, thiago.covoes@ufabc.edu.br

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

CMCC
Centro de Matemática, Computação e Cognição
Center for Mathematics, Computation and Cognition

Álgebra Relacional

Algoritmos em MapReduce

Nas aulas anteriores vimos alguns exemplos de algoritmos implementados com MapReduce:

- Contagem de palavras
- Co-ocorrência de itens
- Medidas estatísticas
- Processamento de Textos

Algoritmos em MapReduce

Um outro tipo de algoritmo necessário ao trabalhar com dados são os de Álgebra Relacional, utilizados em consultas a banco de dados.

Algoritmos em MapReduce

Uma relação R é representada por uma tupla de valores que possuem relação entre si: $(x_1, x_2, \cdots, x_n) \in R$, também escrita na forma $R(x_1, x_2, \cdots, x_n)$.

Algoritmos em MapReduce

Computacionalmente, as relações são representadas de forma tabular como:

x_1	x_2	x_3
3	'a'	True
5	'c'	True
1	'a'	False
...	...	...

Uma relação é essa tabela com um cabeçalho que nomeia os atributos e que cada linha representa uma tupla com os valores de cada relação.

O conjunto de atributos de uma relação é chamado de esquema.

Algoritmos em MapReduce

Dessa tabela podemos responder diversas questões:

- Quais relações possuem x_3 verdadeiro?
- Quais são as relações únicas envolvendo apenas x_1, x_2?
- Quais tuplas existem em comum com outra relação utilizando o mesmo esquema?
- Como juntar duas relações que possuem um atributo em comum?
- Qual o valor estatístico de um subconjunto S de atributos agrupadas por outro subconjunto G?

Essas questões são denominadas queries na linguagem SQL e podem ser classificadas como:

- Seleção: dada uma condição C, selecione as tuplas que a satisfaçam gerando a relação $\sigma_C(R)$.
- Projeção: dado um subconjunto S dos atributos, produzir as relações $\pi_S(R)$.
- União, Interseção, Diferença: operações de conjuntos aplicadas aos conjuntos de tuplas de duas relações distintas.

Algoritmos em MapReduce

- Join: criar uma nova relação como a união das tuplas de R e S sempre que os valores dos atributos em comum concordarem. Denotado $R \bowtie S$.
- Agrupamento e Agregação: dado um conjunto de atributos G e uma operação de agregação $\theta(A)$, gerar a relação $\gamma_{G,\theta}(R)$ em que as tuplas são agrupadas pelos atributos G e o restante dos atributos calculados pela função θ.

Exemplos de Aplicação

Dada a relação (url_{from}, url_{to}) de *links* entre duas páginas da Web. Queremos saber quais páginas podem ser alcançadas partindo de u, tendo como intermediário a página v, ou seja, queremos encontrar as triplas (u, v, w).

Podemos encontrar essa relação pela junção de R com ela mesma: $R \bowtie R$.

Exemplos de Aplicação

Em uma rede social que modela as relações de amizade como $(user, friend)$, podemos querer contar quantos amigos cada usuário possui.

Para isso fazemos um agrupamento $G = \{user\}$ com a agregação $\theta = COUNT$.

Seleção

A operação de seleção pode ser implementada de duas maneiras, ou via `Mapper` ou via `Reducer`, mas não é necessário o uso de ambos.

Seleção

```
1  def mapper(key : int, value : tuple) -> (int, tuple):
2    if select(value):
3      yield (key, value)
4
5  def reducer(key : int, values : [tuple]) -> (int,
   ↪  tuple):
6    # assumindo chave unica
7    yield (key, head(values))
```

Seleção

```
1  def mapper(key : int, value : tuple) -> (int, tuple):
2    yield (key, value)
3
4  def reducer(key : int, values : [tuple]) -> (int,
   ↪  tuple):
5    if select(head(values)):
6      yield (key, head(values))
```

Projeção

A projeção pode ser implementada de forma similar a seleção, só que como pode gerar duplicação, a função `reducer` deve eliminar as tuplas repetidas.

Projeção

```
1  def mapper(key : int, value : tuple) -> (tuple, tuple):
2    newValue = project(value)
3    yield (newValue, (key, newValue))
4
5  def reducer(key : tuple, values : [tuples]) -> (tuple,
   ↪  tuple):
6    fstVal = head(values)
7    newKey = first(fstVal)
8    newValue = second(fstVal)
9    yield (newKey, newValue)
```

União

A união entre duas relações funciona muito parecido com a projeção só que mantendo o conjunto original de atributos. Ou seja, é exatemente a mesma função anterior sendo que `projection = id`.

União

```
1  def mapper(key : int, value : tuple) -> (tuple, tuple):
2    yield (value, (key, value))
3
4  def reducer(key : tuple, values : [tuples]) -> (tuple,
   ↪  tuple):
5    fstVal = head(values)
6    newKey = first(fstVal)
7    newValue = second(fstVal)
8    yield (newKey, newValue)
```

Interseção

Para fazer a interseção, podemos partir do mesmo `mapper` da união. Se duas relações R, S possuem a mesma tupla t, então o `reducer` receberá uma lista contendo dois elementos (se a interção é entre n relações, então n elementos).

Interseção

Basta então filtrar todos os pares chave-valor que possuem apenas um elemento na lista de valores.

Interseção

```
1  def mapper(key : int, value : tuple) -> (tuple, tuple):
2    yield (value, (key, value))
3
4  def reducer(key : tuple, values : [tuples]) -> (tuple,
   ↪  tuple):
5    fstVal = head(values)
6    newKey = first(fstVal)
7    newValue = second(fstVal)
8    if len(values) >= 2):
9      yield (newKey, newValue)
```

Diferença

Para calcular a diferença entre R e S, devemos identificar de onde veio cada tupla. Para isso utilizamos uma estrutura denominada *tagged union* ou *variant*, *tipo soma*, etc.

Diferença

```
1  def mapperR(key : int, value : tuple) -> (tuple, tuple):
2    yield (value, (key, value, 'R'))
3
4  def mapperS(key : int, value : tuple) -> (tuple, tuple):
5    yield (value, (key, value, 'S'))
6
7  def reducer(key : tuple, values : [tuples]) -> (tuple,
   ↪  tuple):
8    fstVal = head(values)
9    newKey = first(fstVal)
10   newValue = second(fstVal)
11   if len(values) == 1 and third(fstVal) == 'R':
12     yield (newKey, newValue)
```

Join

Se pensarmos na relação R como os conjuntos de atributos A e B, e a relação S com os conjuntos B, C, a operação join deve criar uma relação T contendo os conjuntos de atributos A, B, C em toda tupla de R que possui os mesmos valores de B para uma tupla de S.

Join

```
1  def mapperR(key : int, value : tuple) -> (tuple, tuple):
2    a, b = splitR(value)
3    yield (b, (a, 'R'))
4
5  def mapperS(key : int, value : tuple) -> (tuple, tuple):
6    b, c = splitS(value)
7    yield (b, (c, 'S'))
8
9  def reducer(key : tuple, values : [tuples]) -> (tuple,
     ↪  tuple):
10   if len(tuples) >= 2:
11     sortedVals = sortBy(values, second)
12     yield(key, map(first, sortedVals))
```

Group e Aggregate

Vamos assumir o caso mais simples em que a base de relações será agrupada por um único atributo A e a função de agregação será aplicada em apenas um atributo B, todos os outros atributos da relação (vamos chamar de C), são descartados.

Group e Aggregate

```
1  def mapper(key : int, value : tuple) -> (int, int):
2    a, b, cs = value
3    yield (a, b)
4
5  # agg([int]) -> int
6  def reducer(key : int, values = [int]) -> (int, int):
7    yield (key, agg(values))
```

Group e Aggregate

Notem que essa implementação é genérica o suficiente para lidar com os casos em que A representa múltiplos atributos (pois uma tupla também define uma chave única) e quando B também representa múltiplos atributos.

Nesse segundo caso, a função $\mathtt{agg}$ deve ser capaz de lidar com os múltiplos atributos.

Group e Aggregate

Como exemplo, considere que agg deve fazer a somatória de b_1, o máximo de b_2 e o mínimo de b_3:

```
1  def agg(values : [(int, int, int)]) -> (int, int, int):
2    total, maximum, minimum = head(values)
3    for b1, b2, b3 in tail(values):
4      total += b1
5      maximum = max(maximum, b2)
6      minimum = min(minimum, b3)
7    return (total, maximum, minimum)
```

Group e Aggregate

Para outras funções de agregação, como a média, podemos utilizar o mesmo algoritmo de aulas anteriores.

Múltiplas Operações

Como se traduz a seguinte *query* SQL em *MapReduce*?

```
1 SELECT AVG(idade) FROM base
2 WHERE salario > 2000
3 GROUP BY ocupacao
```

Múltiplas Operações

```
1  def mapper(key : int, value : (int, float, str)) ->
   ↪  (str, (int, int)):
2    if salario(value) > 2000:
3      yield ocupacao(value), (idade(value), 1)
4
5  def reducer(key : str, values : [(int, int)]) -> (str,
   ↪  float):
6    total = sum(map(fst, values))
7    n = sum(map(snd, values))
8    yield key, total/n
```

Multiplicação de Matriz

Multiplicação de Matriz

A multiplicação de Matriz é uma operação recorrente em mineração de dados em diversas tarefas como Regressão, Redução de Dimensionalidade, Agrupamento, Análise de Grafos, etc.

Multiplicação de Matriz

Dada duas matrizes $P_{m \times n}$ e $Q_{n \times o}$, a multiplicação delas gera uma matriz $R_{m \times o}$, de tal forma que:

$$r_{ik} = \sum_j p_{ij} \cdot q_{jk}$$

Multiplicação de Matriz

Podemos pensar em uma matriz como uma relação contendo três atributos: linha, coluna, valor. Ou seja, a tripla (i, j, pij) para a matriz P e (j, k, qjk) para a matriz Q.

Multiplicação de Matriz

Como geralmente as matrizes que trabalhamos são esparsas, podemos omitir todas as triplas cujo valor seja igual a zero. Dessa forma a representação relacional é uma das mais adequadas para armazenar uma matriz de dados.

Multiplicação de Matriz

Observando novamente a fórmula da multiplicação matricial, o ponto central é o cálculo do produto interno entre a coluna j de P e a linha j de Q. Ou seja, um *join* através desses campos.

Multiplicação de Matriz

```
1  def mapperP(key : (int, int), value : float) -> (int,
   ↪  (int, float, str)):
2    i, j = key
3    yield (j, (i, value, 'P'))
4
5  def mapperQ(key : (int, int), value : float) -> (int,
   ↪  (int, float, str)):
6    j, k = key
7    yield (j, (k, value, 'Q'))
8
9  def reducer(key : int, values : [(int, float, str)]) ->
   ↪  ((int,int), float):
10   P, Q = splitPQ(values)
11   for i, pij in P:
12     for k, qjk in Q:
13       yield ((i,k), pij*qjk)
```

Multiplicação de Matriz

Agora basta agregarmos os valores efetuando a somatória e gerando uma base de relações de triplas, que representa a matriz resultante.

Multiplicação de Matriz

```
1  def mapper(key : (int, int), value : float) -> ((int,
    ↪  int), float):
2    yield key, value
3
4  def reducer(key : (int, int), values : [float]) ->
    ↪  ((int, int), float):
5    yield key, sum(values)
```

Comentários Finais

Comentários Finais

Vimos na aula de hoje como podemos implementar algoritmos de álgebra relacional e multiplicação de matrizes utilizando *MapReduce*.

É possível fazer a junção dessas operações para executar *queries* mais complexas em poucas passagens de *MapReduce*.

Na próxima aula aprenderemos sobre o *Apache Spark*, a evolução do *Hadoop*.