Implementação do Algoritmo Eclat em PySpark

Leandro Alvarez de Lima

Inteligência na Web e Big Data

Universidade Federal do ABC

1. Introdução

Entre os vários tipos de algoritmos para mineração de dados voltados para Big Data, há os algoritmos de mineração de itens frequentes (frequent itemset). Esse método de mineração é um dos mais populares atualmente[1]. A mineração de frequent itemset tem o objetivo de encontrar grupos frequentes de itens em um banco de dados que contenha transações [2], com o intuito de, por exemplo, determinar a relação entre os itens de compras em supermercados [3] ou entre os filmes assistidos em uma plataforma de vídeos sob demanda.

Os principais algoritmos desse tipo são o Apriori, o FpGrowth e o Eclat.[4] Dentre eles, o algoritmo que será abordado nesse artigo será o Eclat. Ao longo desse trabalho será explicado o seu funcionamento e será realizada a sua implementaccão utilizando programacão paralela em PySpark.

2. Funcionamento do Algoritmo Eclat

Uma das principais características do Eclat é utilizar um formato de dados vertical ao invés de horizontal[5] que é mais usual, como faz o Apriori[4]. Então o primeiro passo do funcionamento desse algoritmo é transformar a base de dados para o layout vertical, como na figura a seguir.

– horizontal vs vertical data layout

Horizontal Data Layout

TID	Items
1	A,B,E
2	B,C,D
3	C,E
4	A,C,D
5	A,B,C,D
6	A,E
7	A,B
8	A,B,C
9	A,C,D
10	B

Vertical Data Layout

A	B	C	D	E
1	1	2	2	1
4	2	3	4	3
5	5	4	5	6
6	7	8	9	
7	8	9		
8	10			
9				

Com os dados já em layout vertical, os seguintes passos do algoritmo são realizados:[5]

- A base de dados é escaneada para detectar os itemsets mais simples, formados por apenas um item, chamados de frequent-1-itemsets;
- Os frequent-1-itemsets passam por interseções entre eles para gerar candidatos a itemsets de 2 itens. Esses candidatos passam pela verificação do suporte mínimo, que é o parâmetro que indica o número mínimo de vezes que o conjunto de itens deve aparecer no conjunto de transações para ser considerado frequente, e são descartados ou classificados como frequent-2-itemsets;
- Os frequent-2-itemsets passam por nova interseção e verificação do suporte mínimo para serem formados os itemsets de 3 itens(ou frequent-3-itemsets);
- Os passos são repetidos até que todos os candidatos a itemset sejam gerados e verificados para que possam ser classificados como frequent itemsets.

3. Implementação do Algoritmo Eclat em PySpark

Seguindo a proposta da disciplina, foi realizada a implementção do algoritmo estudado de forma paralelizada utilizando o PySpark, que é o framework Spark, construído para processamentos de Big Data e com suporte a paralelização e MapReduce, em conjunto com a linguagem de programação Python. O banco de dados utilizado para a implementação é um banco de dados fictício, porém baseado em alunos e as disciplinas escolhidas por eles durante um curso. O intuito final dessa implementação é apresentar os grupos de disciplinas escolhidas em conjunto com maior frequência. A seguir um modelo de testes da base de dados utilizada, onde o primeiro número é o ID do aluno e os restantes os ID's das disciplinas escolhidas por eles:

```
1,105,109,110
2,101,104,106,108,110
3,101,110,115
4,101,103,108,110
5,109,112,113
6,102,104,105,106,109
7,103,105,106,107,110,111
8,104,106,107,109,110,115
9,106,108,110
10,101,105,107,108,109
```

Após criado o RDD no PySpark com os dados das transações e setados os números de partições na paralelização e do suporte mínimo, o primeiro passo é criar uma tupla de cada linha do registro para que o primeiro elemento seja o ID correspondente ao aluno:

```
tuplesRDD = arquivoRDD.map(lambda x : (x[0].split(",")[0],  x[0].split(",")[1:]));
```

Com as tuplas no formato (ID do aluno, [ID's das disciplinas]), é necessário realizar a verticalização do layout dos dados:

```
vertical = tuplesRDD.flatMapValues(lambda x : x)
verticalTuplesRDD = vertical.map(lambda x : (x[1], x[0])) #inverte disciplina e aluno
aggregateByKeyRDD = verticalTuplesRDD.groupByKey() \
                                     .map(lambda x : (x[0], list(x[1]))) \
                                     .sortByKey();
```

O próximo passo é aplicar o filtro do suporte mínimo para eliminar as disciplinas com menos escolhas de alunos que o valor parametrizado como suporte mínimo:

```
filterValuesRDD = aggregateByKeyRDD.filter(lambda x : len(x[1]) > minSupport) \
                                   .map(lambda x : ([x[0]],x[1]));
```

A seguir, as duas funções criadas para realizarem as interseções para geração dos candidatos a frequent-itemsets. A primeira função é utilizada para geração dos primeiros itemsets e a segunda para todos os itemsets seguintes:

```
def funcao(disc, rdd_collect):
    lista = [];
    for x in rdd_collect:
        if (not disc[0] == x[0]) & (disc[0] < x[0]):
            disciplinas = unir(disc[0],x[0])
            ids = intersect(disc[1],x[1])
            tupla = tuple(sorted(disciplinas)),tuple(sorted(ids))
            if not tupla in lista: lista += [tupla]
            #tupla = tuple(sorted(disciplinas)),(sorted(ids))
            #if not tupla in list: list += [tupla]
    return tuple(lista)
```

```
def funcao2(disc, rdd_collect):
    lista = []
    for x in rdd_collect:
        if disc[0] != x[0]:
            disciplinas = unir(disc[0],x[0])
            ids = intersect(disc[1],x[1])
            if (len(disciplinas)) == (len(disc[0]) + 1):
                tupla = tuple(sorted(disciplinas)),tuple(sorted(ids))
                #tupla = (sorted(disciplinas), sorted(ids))
                if not(tupla in lista):lista += [tupla]
    return tuple(lista)
```

Por meio de um flatMap a função de interseção é aplicada para criar os primeiros candidatos a frequent-itemsets e logo a seguir um filtro elimina os candidatos inferiores ao suporte mínimo, criando as primeiras frequent-itemsets (abaixo do código o resultado do processamento fornece os primeiros frequent-2-itemsets no formato [(Disciplinas), número de alunos]:

```
disciplinas_singleton = filterValuesRDD.collect()
filterDisciplinasRDD = filterValuesRDD.flatMap(lambda x : funcao(x, disciplinas_singleton)) \
                                      .filter(lambda x : len(x[1]) > minSupport) \
                                      .sortByKey()

contagem = filterDisciplinasRDD.map(lambda x : (x[0], len(x[1])))
print(contagem.collect())
```

```
[(('129', '135'), 70992), (('129', '139'), 69900), (('129', '153'), 70512), (('129', '17'), 66984), (('13
5', '139'), 70092), (('135', '153'), 70260), (('135', '17'), 66288), (('139', '153'), 70032), (('139', '1
7'), 65916), (('147', '17'), 65736), (('153', '17'), 67080)]
```

A criação de novos candidatos com 3 itens continua a ser realizada pela função de interseção por um flatMap. Então, o reduceByKey junta os candidatos gerados em duplicidade e o filtro retira novamente os que não alcanam o suporte mínimo. O processo se repete até que não tenham mais candidatos a serem criados.

```
disciplinas2itemsets = filterDisciplinasRDD.collect()

filterDisciplinasRDD2 = filterDisciplinasRDD.flatMap(lambda x : funcao2(x, disciplinas2itemsets)) \
                                            .map(lambda x : (x,1)) \
                                            .reduceByKey(add) \
                                            .sortByKey() \
                                            .map(lambda x : x[0]) \
                                            .filter(lambda x : len(x[1]) > minSupport)

contagem2 = filterDisciplinasRDD2.map(lambda x : (x[0], len(x[1])))
print(contagem2.collect())
```

```
[(('129', '135', '139'), 69444), (('129', '135', '153'), 69684), (('129', '135', '17'), 65400), (('129',
'139', '153'), 69288), (('129', '139', '17'), 65100), (('129', '153', '17'), 65532), (('135', '139', '15
3'), 69372), (('135', '139', '17'), 65124), (('135', '153', '17'), 65160), (('139', '153', '17'), 65292)]
```

```
disciplinas3itemsets = filterDisciplinasRDD2.collect()

filterDisciplinasRDD3 = filterDisciplinasRDD2.flatMap(lambda x : funcao2(x, disciplinas3itemsets)) \
                                             .map(lambda x : (x,1)) \
                                             .reduceByKey(add) \
                                             .sortByKey() \
                                             .map(lambda x : x[0]) \
                                             .filter(lambda x : len(x[1]) > minSupport)

contagem3 = filterDisciplinasRDD3.map(lambda x : (x[0], len(x[1])))
print(contagem3.collect())
```

```
[(('129', '135', '139', '153'), 68976)]
```

```
disciplinas4itemsets = filterDisciplinasRDD3.collect()

filterDisciplinasRDD4 = filterDisciplinasRDD3.flatMap(lambda x : funcao2(x, disciplinas4itemsets)) \
                                             .map(lambda x : (x,1)) \
                                             .reduceByKey(add) \
                                             .sortByKey() \
                                             .map(lambda x : x[0]) \
                                             .filter(lambda x : len(x[1]) > minSupport)

contagem4 = filterDisciplinasRDD4.map(lambda x : (x[0], len(x[1])))
print(contagem4.collect())
```

```
[]
```

3.1. Resultados

Ao término do algoritmo todos frequent-itemsets são obtidos enquanto vão sendo gerados do menor para o maior. O algoritmo foi rodado em uma máquina quadcore, com 8gb de memória RAM e sistema operacional Ubuntu. A versão paralelizada em 4 partições apresentou ganho de cerca de 46% sobre a não paralelizada, pois rodou em 164.73467183113098 segundos contra 112.58522939682007 segundos da não paralelizada.

4. Referências

[1] C. Borgelt, Frequent item set mining, Wiley Int. Rev. Data Min. and Knowl. Disc. 2 (2012) 437–456.

[2] R. Agrawal, T. Imieliński, A. Swami, Mining association rules between sets of items in large databases, SIGMOD Rec. 22 (1993) 207–216.

[3] J. Heaton, Comparing dataset characteristics that favor the apriori, eclat or fp-growth frequent itemset mining algorithms, SoutheastCon 2016 (2016) 1–7.

[4] K. Garg, D. Kumar, Comparing the performance of frequent pattern mining algorithms, International Journal of Computer Applications 69 (2013) 21–28.

[5] Z. Ma, J. Yang, T. Zhang, F. Liu, An improved eclat algorithm for mining association rules based on increased search strategy 9 (2016) 251–266.