BUC

Olione Lonie

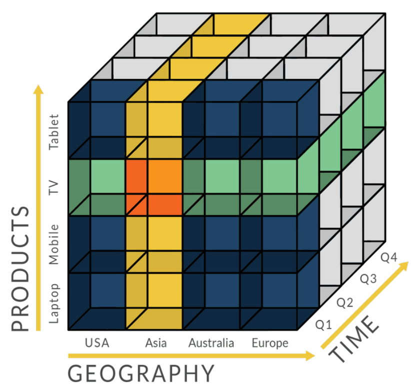
December 2022

1 Introduction

Les data-cubes sont utilisés pour présenter les données massives et les traiter plus facilement .

Un cube se partage en sous-cube selon n dimensions , ces dimensions sont constituées par un ensemble d'éléments , par exemple nous pourrions avoir la dimension produite qui est constitués de TV , brosse à dents , etc .

Dans ce rapport, nous étudierons la méthode **BUC** qui fut proposée par Kevin Beyer et Raghu Ramaskrishnan dans : Bottom-Up Computation of Sparse and Iceberg CUBES



1.1 Intérêts :

Les entreprises ont aujourd'hui grand besoin d'avoir des méthodes rapides et efficaces pour répondre aux besoins de leur client .
En effet il a été montré dans le Rapport Universel de l'Oreal 2021 que les clients demandes à leur marque de proposer des produits qui leur sont adaptés .

De plus d'après ce rapport les tendances des produits mais aussi les ralentissements de la chaîne de production pour chaque gamme doivent être déduite des reportings de chaque site . Les méthodes classiques d'extraction de connaissance (**ETL**) étant trop coûteuses pour traiter une si grande quantité de donnée de nouvelle méthode doivent être développé .

C'est dans cet environnement que se place **l'Algorithme BUC** .

1.2 Principe :

Fonctions SQL :

Classiquement, les données sont découpées (*slicing*) grâce à la fonction *GROUP BY* . *GROUP BY* permet de regrouper les éléments partageant une même valeur d'attribut .
Pour éviter d'avoir à processer l'entièreté de la base de données, on peut adjoindre une condition à ce slicing grâce au mot-clef *HAVING* .

Ceci jette les bases de l'approche iceberg-CUBE : plutôt que de créer l'ensemble du CUBE (agréger toutes les données pour l'ensemble des dimensions) on agrégera les données si et seulement si elle vérifie une condition fixée à l'avance , par exemple un *treshold* .

La Treshold Condition :

Un *treshold* est une borne qui délimite les informations pertinentes des informations peu utiles .
Imaginons qu'un commerce souhaite savoir quels sont les produits tendance pour chacune des saisons , alors il doit additionner(agréger) l'ensemble de ses ventes pour chaque produit (*GROUP BY* product) et chaque saison (*GROUP BY* season) puis de calculer l'ensemble des indicateurs qu'il souhaite calculer (*SELECT* $\{SUM, MAX, AVG, ..\}$) .

La méthode **BUC** assume qu'il est inutile de calculer les indicateurs pour les produits "peu vendus ", car ils n'influencent pas la tendance globale , ceci justifie l'emploi d'un *treshold* :

$$\textit{HAVING}\ \text{SUM(*)} > \textit{treshold}$$

Ce constat permet de diminuer drastiquement le temps de calcul des indicateurs car seules les catégories/produits les plus importants seront calculés .

2 Concept :

2.1 Outils utilisés :

Comme nous l'avons vue la méthode **BUC** s'appuie en grande partie sur le "*treshold*ing" il sera appelé support minimal (abrégé en $min - sup$) .

Nous utiliserons la mère de la fonction *GROUP BY* : la fonction *CUBE* , elle calcule l'ensemble des *GROUP BY* sur les n dimensions de la données .
Par exemple , un enregistrement avec les dimensions produit , saison et localisation se verrait agréger par *CUBE* sur les dimensions (produit) , (produit ,saison) , (produit , saison, localisation) , (saison) , (localisation) et (localisation , saison) .
Ceci correspond à l'ensemble des permutations de (produit , saison , localisation) .

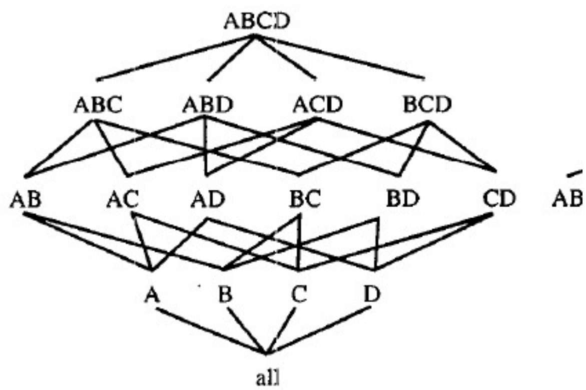
Un *CUBE* soumis à un *treshold*ing sera appelé un iceberg-CUBE car seule la partie émergé (importante) des données se fera analysé en détail .

2.2 Methode :

2.2.1 Arbre :

Même avec ce *tresholding* l'opération *CUBE* reste très gourmande , c'est pourquoi une autre de nos préoccupations sera de la partitionner en plusieurs sous-problème résolvable par des machines indépendantes (pipeline cf cloud computing) .

Ce partitionnement va s'appuyer sur le principe de l'algèbre relationnel , rappelez vous un *CUBE* se divise en n dimension par exemple : **A , B , C** et **D** .



Si on part de la racine **ABCD** alors le graphe ce lit comme suit , pour former **ABCD** j'ai besoin de **ABC , ABD , ACD** et **BCD** en d'autres termes, nous pourrions dire **ABC**∪ **ABD** ∪ ... ∪ **BCD** = **ABCD** .

Pour former **AB** j'ai besoin de **ABC** et **ABD** .
Pour former **A** j'ai besoin de **AB , AC** et **AD** .
Pour former all j'ai besoin de **A , B , C** et **D** .

Si on part de la feuille le graphe se lit **all** est constitué de **A , B , C** et **D** etc en d'autres termes, nous pourrions dire que **all** ∩ **A** = **A** (ou **all** *GROUP BY* **A** donne **A**) .

Dans l'approche **BUC** autrement dit "Bottom up Computation" c'est la dernière formulation que nous allons utiliser .

2.3 Exemple :

Reprenons l'exemple précédemment évoqué , avec **A** : le nom du produit , B : la saison et **C** : la localisation . Pour chacune de ces 3 dimensions nous nous restreindrons aux valeurs les plus fréquentes par exemple : *écran-télé-type-X* , *écran-télé-type-Y* et *écran-ordinateur-type-X* , *saison-hiver* ,*saison-été* et *saison-printemps* , *USA* , *Europe* et *Asie* .
Nous constatons que les saisons et les localisations sont toutes présentes en effet le *treshold* n'a pas exclu ces valeurs car elles sont toutes significatives pour l'analyse du marché en revanche les produits ne sont pas tous tendances (significatives) .

2.4 Implémentation :

Le **BUC** comme nous l'avons dit commence par l'entité la plus agrégée vers l'entité la moins agrégée. (du plus grand vers le plus petit/ du moins de *GROUP BY* vers le plus de *GRP BY*)

```
Procedure BottomUpCube(input, dim)
Inputs:
   input: The relation to aggregate.
   dim: The starting dimension for this iteration.
Globals:
   constant numDims: The total number of dimensions.
   constant cardinality[numDims]: The cardinality of
         each dimension.
   constant minsup: The minimum number of tuples in a
         partition for it to be output.
   outputRec: The current output record.
   dataCount[numDims]: Stores the size of each partition.
         dataCount[i] is a list of integers of size
         cardinality[i].
Outputs:
   One record that is the aggregation of input.
   Recursively, outputs CUBE(dim, ..., numDims) on
         input (with minimum support).
Method:
   Aggregate(input); // Places result in outputRec
   if input.count() == 1 then // Optimization
         WriteAncestors(input[0], dim); return;
   write outputRec;
   for d = dim ; d < numDims ; d++ do
      let C = cardinality[d];
      Partition(input, d, C, dataCount[d]);
      let k = 0;
      for i = 0 ; i < C ; i++ do // For each partition
         let c = dataCount[d][i]
         if  c >= minsup  then // The BUC stops here
            outputRec.dim[d] = input[k].dim[d];
            BottomUpCube(input[k ... k+c], d+1);
         end if
         k += c;
      end for
      outputRec.dim[d] = ALL;
   end for
```

Figure 5: Algorithm BottomUpCube (BUC)

2.4.1 Paramètres :

Paramètre d'entrée :

Dans le pseudo code ci-dessus nous avons deux paramètres d'entrés :

Input:
L'input peut pour l'instant être vue comme un paramètre composite : un dictionnaire qui associe des clés les identifiants des colonnes aux valeurs de chaque tuple pour chacune de ces colonnes .
Ainsi nous avons un input avec par exemple une ligne/tuple :

$$[A : val\_a1\ ,\ B : val\_b2\ ,\ C : val\_c2\ ,\ D : val\_d1\ ]$$

Dim :
Dim quant à lui décrit l'index de la dimension actuellement processer , en d'autres termes la dimension pour laquelle on parcourt l'ensemble de ses valeurs .

Paramètre de sorties :
Le résultat de l'opération *CUBE* affiché petit à petit

2.4.2 Commentaire sur l'Algorithme :

Premièrement, on agrège l'input (à la manière d'un *GROUP BY*) on confond les entrées/lignes qui partagent la même valeur sur la dimension concernée ($index_dim$) .

Ensuite on parcoure l'ensemble des valeurs distinctes sur laquelle on a aggrégé la dimension d'indexe $index_dim$ pour chacune de ces valeurs on partitionne les lignes sur la dimension suivante .

On parcourt l'ensemble de ces partitions (i de 0 à n) pour chacune on teste si sa cardinalité (le nombre de ligne) est supérieure à la valeur *treshold* si oui on rappelle **BUC** sur *partition[i]* et la dimension suivante : *k+1* .

Ce processus est itéré pour chacune des permutations du mot **ABCD** (représentant les 4 dimensions de notre exemple) .

2.4.3 Nota Bene :

Les mots **ABCD , ACDB , BCDA** ect sont équivalents pour preuve :

GRP BY **A,B,C,D** $\leftrightarrow$ *GRP BY* **B,C,D,A** (cf. l'opération *CUBE*).

Ainsi nous avons : $\frac{n(n+1)}{2}$ computation possibles .

2.5 Implémentation de l'Algorithme :

BUC-implémentation https://github.com/LonyI175/BUC-Iceberg-Cube forked from AsaadMe repository .

Pour une première implémentation de l'algorithme il est pratique de le pensé comme une chaine de chiffre binaire 0,1,* (avec * représentant le caractère joker) .

Tout d'abord, nous devons binarizer les entrées pour ce fait nous partagerons chacune des dimensions en 2 parties :

- $\mathbf{A}[\mathrm{i}] \leq sup\_a \implies 1$
- $\mathbf{A}[\mathrm{i}] > sup\_a \implies 0$

Une fois ceci fait pour chacune des dimensions :

```
def pre_process(input_file_name: str) -> list[dict]:
    """Convert records to dict: {"A": 0 or 1, ...}"""

    out_list = []

    with open(input_file_name, "r") as dataset:

        for line in dataset:
            out_line = {}

            if re.search(r"\b[0-3]\b", line):

                out_line["A"] = 1
            else:
                out_line["A"] = 0
```

Nous allons ensuite merger ces partitions sur le nouvel indexe de dimension (nous regroupons les entrées qui ont la même valeur sur la dimension identifiée par l'indexe : *index_dim*) .

```
for row in input:
            if row.get(cur_dim) == 1:
                aggr_list_1.append(row)
            else:
                aggr_list_0.append(row)
```

Ensuite, nous calculons le nombre de ligne dans chacune de ces partitions si ce nombre est supérieur aux *treshold* alors cette partition est significative et elle fait l'office d'un nouvel appel de l'algorithme **BUC** sinon elle est discardé :

```python
datacount = [len(aggr_list_0), len(aggr_list_1)]

        if datacount[1] >= min_sup:

            cell_copy = cell.copy()
            cell_copy[cur_dim] = "1"
            print("counter : "+str(my_counter)+ f" {cell_repr(cell_copy)}: {
   datacount[1]}")

            if (index:= dims.index(cur_dim) + 1) < len(dims):
                BUC( aggr_list_1 , index, cell_copy ,my_counter)
```

Note :
Seul le traitement de la liste *aggr_list_1* est présenté , mais il en va de même pour la liste concernant le second intervalle : *aggr_list_0* .

2.5.1 Résultat :

Fichier utilisé :

Database https://github.com/LonyIl75/BUC-Iceberg-Cube/blob/master/Dataset.txt (réalisé par Mehran Asaad)

Aperçu :

```
cur_dim:A  counter : 1 parent :0
[{'A': 0, 'B': 1, 'C': 0, 'D': 0, 'E': 0}, {'A': 0, 'B': 1, 'C': 1, 'D': 0, 'E': 1}, {'A': 0, 'B': 1, 'C': 1, 'D': 1,
0 : (*, *, *, *, *): 13610
1 : (*, *, *, *, *): 86390
counter : 1 (1, *, *, *, *): 86390
cur_dim:B  counter : 2 parent :1
[{'A': 1, 'B': 0, 'C': 1, 'D': 1, 'E': 1}, {'A': 1, 'B': 0, 'C': 1, 'D': 1, 'E': 1}, {'A': 1, 'B': 0, 'C': 0, 'D': 1,
0 : (1, *, *, *, *): 15328
1 : (1, *, *, *, *): 71062
counter : 2 (1, 1, *, *, *): 71062
cur_dim:C  counter : 3 parent :2
[{'A': 1, 'B': 1, 'C': 0, 'D': 1, 'E': 1}, {'A': 1, 'B': 1, 'C': 0, 'D': 1, 'E': 1}, {'A': 1, 'B': 1, 'C': 0, 'D': 1,
0 : (1, 1, *, *, *): 8716
1 : (1, 1, *, *, *): 62346
counter : 3 (1, 1, 1, *, *): 62346
cur_dim:D  counter : 4 parent :3
[{'A': 1, 'B': 1, 'C': 1, 'D': 0, 'E': 1}, {'A': 1, 'B': 1, 'C': 1, 'D': 0, 'E': 0}, {'A': 1, 'B': 1, 'C': 1, 'D': 0,
0 : (1, 1, 1, *, *): 7129
1 : (1, 1, 1, *, *): 55217
counter : 4 (1, 1, 1, 1, *): 55217
cur_dim:E  counter : 5 parent :4
[{'A': 1, 'B': 1, 'C': 1, 'D': 1, 'E': 0}, {'A': 1, 'B': 1, 'C': 1, 'D': 1, 'E': 0}, {'A': 1, 'B': 1, 'C': 1, 'D': 1,
0 : (1, 1, 1, 1, *): 6428
1 : (1, 1, 1, 1, *): 48789
counter : 5 (1, 1, 1, 1, 1): 48789
counter : 5 (1, 1, 1, 1, 0): 6428
counter : 4 (1, 1, 1, 0, *): 7129
cur_dim:E  counter : 6 parent :4
[{'A': 1, 'B': 1, 'C': 1, 'D': 0, 'E': 0}, {'A': 1, 'B': 1, 'C': 1, 'D': 0, 'E': 0}, {'A': 1, 'B': 1, 'C': 1, 'D': 0,
0 : (1, 1, 1, 0, *): 1039
1 : (1, 1, 1, 0, *): 6090
```

References

[1] *BUC implémentation* , Mehran Asaad , 2021, https://github.com/LonyIl75/BUC-Iceberg-Cube

[2] *Bottom-Up Computation of Sparse and Iceberg CUBES* , Kevin Beyer and Raghu Ramakrishnan , 1999, https://dl.acm.org/doi/10.1145/304181.304214

[3] *Iceberg-Cubes* , Uregina , 2005, http://www2.cs.uregina.ca/~dbd/cs831/notes/dcubes/iceberg.html