Université de Montréal

Rapport du travail 2

Par
Geneviève Paul-Hus (20037331)
Abderrahim Tabta (20133680)

Faculté art et sciences

Travail présenté à Alain Tapp
Dans le cadre du cours IFT3700

Décembre 2021

IFT3700 Devoir 2

Question 1 (30%)

Imaginez que vous devez traiter des données de nature astronomique. Il s'agit de données sous forme de table concernant 7300 milliards d'étoiles.

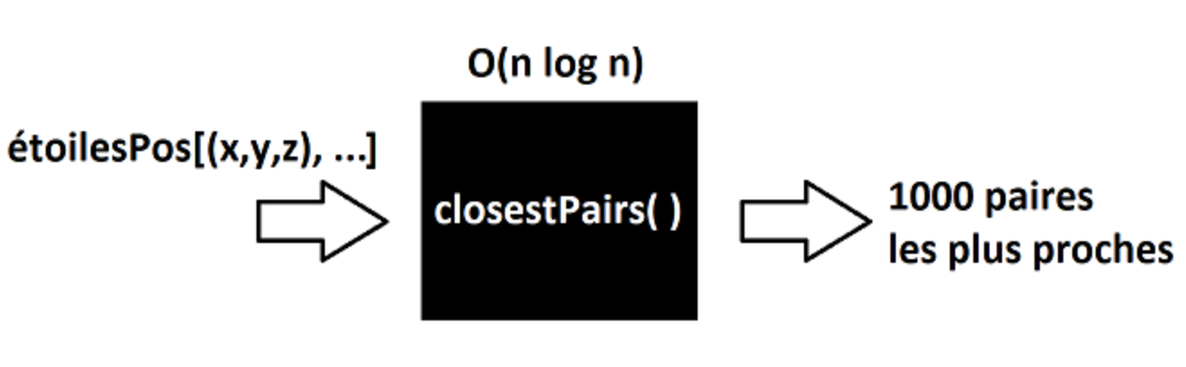
Pour chaque étoile la table contient la position dans l'espace 3D (3 FLOAT64), la luminosité apparente de l'étoile (1 FLOAT32) et sa catégorie représentée par un entier entre 1 et 10 (1 INT8) et un vecteurs de 23 caractéristiques physiques (23 FLOAT32). La taille des données qui sont stockées de façon efficace dépasse le 900 Téraoctets.

Les données sont réparties de façon balancée sur 1200 serveurs avec un processeur rapide, 128 Gigaoctets de mémoire vive et utilisant chacun un disque SSD de capacité 4 Téraoctets et de vitesse de lecture et écriture de 3 Gigaoctets/sec. La communication entre les serveurs s'effectue à une vitesse de 1 Gigaoctets/sec.

Proposez une approche distribuée qui permet de répondre aux questions suivantes et expliquez en détail toute la démarche permettant leur résolution.

A. Trouvez les 1000 paires d'étoiles les plus proches en termes de distance euclidienne.Recherche des deux points les plus rapprochés

Nous savons qu'un algorithme qui prend en argument une table d'étoiles (pour être plus précis les positions des étoiles en x,y,z) retourne les 1000 paires les plus proches d'étoiles en O(n log n).



Nous utiliserons cet algorithme comme une boîte noire.

Puisque les étoiles dans nos serveurs ne sont pas triées , il est possible qu'il existe une étoile a dans un serveur A , et une étoile b dans un serveur B , tel que la distance de la paire(a,b) est plus petite que les meilleures paires trouvées par l'algorithme dans le serveur A et le serveur B séparément. En effet, le seul moyen de trouver cette paire(a,b) et d'utiliser l'algorithme sur A+B.

Le défi étant ici qu'il existe pas deux serveurs, mais 1200. Et il faut comparer tous ces serveurs là un à un pour être sûr de manquer aucune paire. Une autre formulation serait de dire qu'il faut que chaque serveur i soit comparé à tous les autres : {Algo(Serv_i) , Algo(Serv_i+ Serv_1), ...,Algo(Serv_i, + Serv_1200)}. Évidemment, nous sommes limités par la puissance de nos serveurs (Capacité, vitesse de lecture etc).

La question est alors : Quelle est la méthode la plus rapide pour comparer tous nos serveurs un à un tout en respectant la capacité de nos serveurs ? Nous avons penser à un algorithme simple que nous alons prendre le temps d'expliquer.

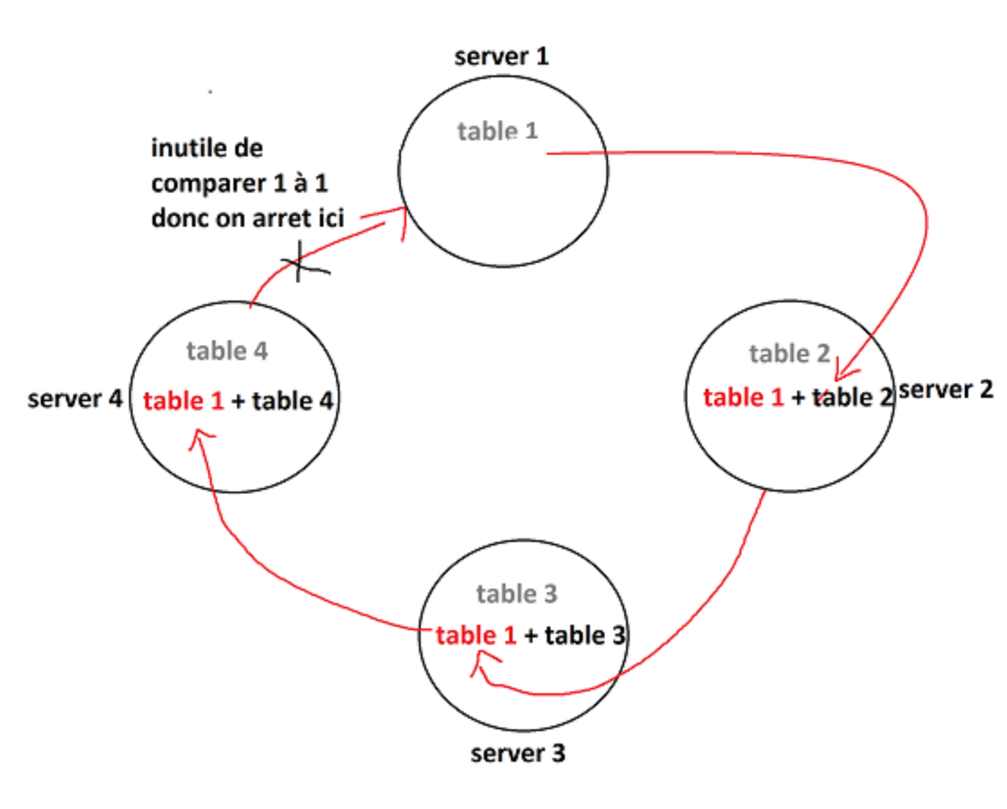
Mise en contexte

Bon, avant de rentrer dans les détails , expliquons l'essence de notre idée. Ademttons que nous avons un certains nombre de serveurs. pour illustrer disons 4. Donc, nous avons les serveurs {1,2,3,4} et chacuns de nos serveur contient une table d'étoiles. Biensûre en parallèle , tous les serveurs peuvent appliquer closestPairs() sur leurs propres tables.

Cela dit , comme nous avions expliqué, ce n'est pas suffisant, il faut "comparer" les serveurs un à un aussi. Donc, pour le moment concentrons nous sur le serveur 1. Donc, nous voulons rouler closestPairs(1) , mais aussi closestPairs(1+2), closestPairs(1+3) et enfin closestPairs(1+4).

L'idée que nous avons eu était de faire circuler la table du serveur 1 aux autres serveurs de manière circulaire. Donc :

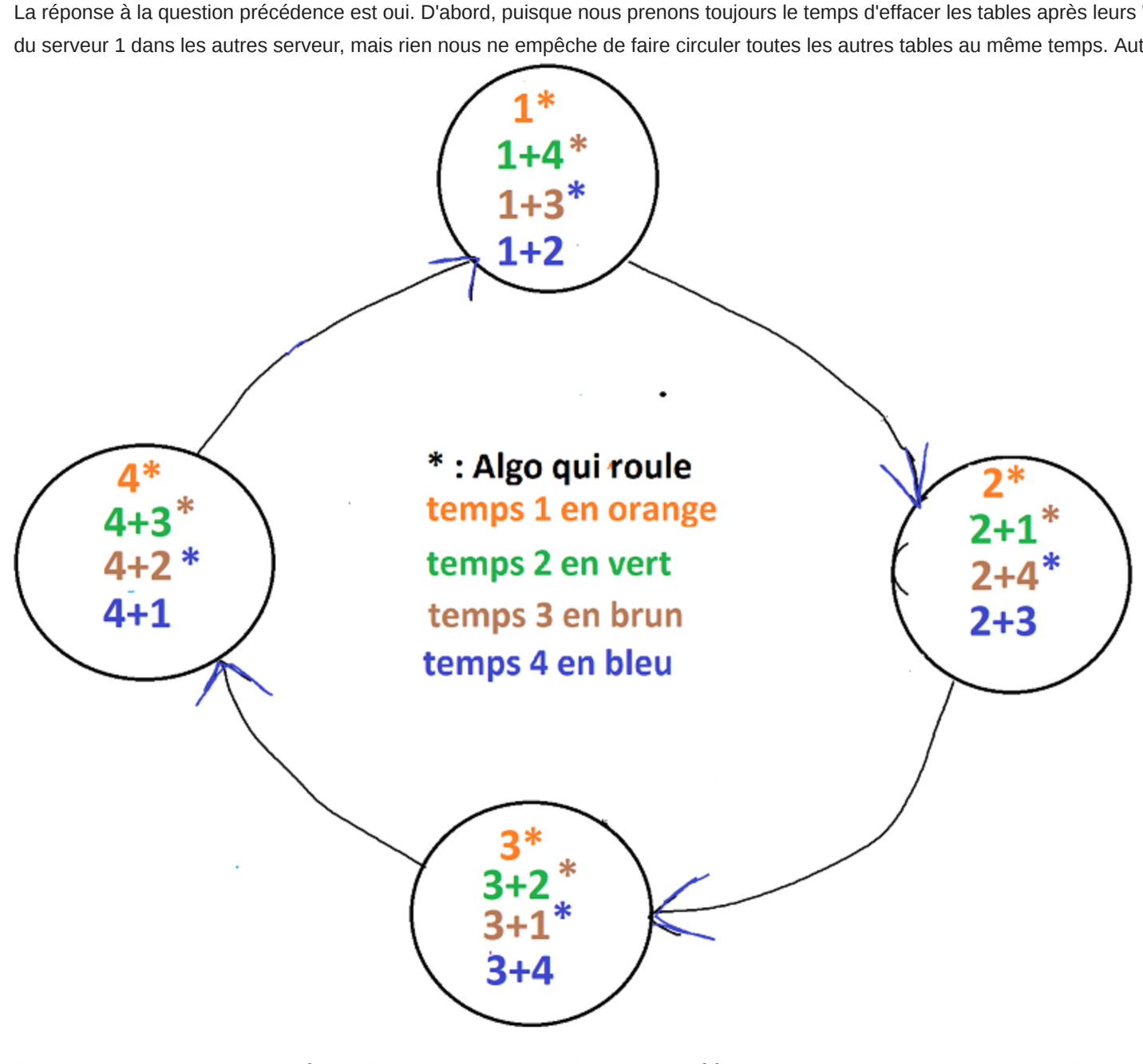
```
1. Premier  temps : closestPairs() sur la table i de chaque serveur i en parallèle ( où i ∈ {1,2,3,4} )
2. Deuxième temps : Envoyer la table 1 dans le serveur 2
3. Troisième temps : rouler closestPairs(table1 + table2) et biensûre on met à jour les meilleures paires
3. Troisième temps : On commence à envoyer la table 1 au serveur 3
4. Quatrième temps : On supprime la table 1 du serveur 2 lorsqu'elle a fini d'être copié au serveur 3
5. On recommence ce qu'on a fait plus haut ( soit closestPairs(table 1 + table3) et ainsi de suite ... )
```



Bref, chaque serveur aura réussi à comparer sa propre table à la table 1! Mais ce n'est pas suffisant, on veut que chaque serveur compare sa table à toutes les autres tables. Pouvons-nous généraliser ce qu'on a fait plus haut?

Généraliser cercle

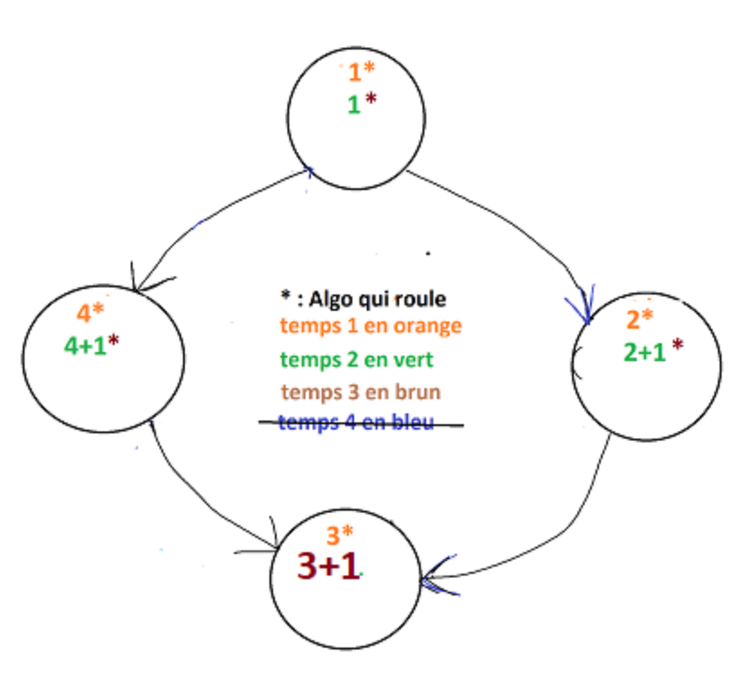
La réponse à la question précédence est oui. D'abord, puisque nous prenons toujours le temps d'effacer les tables après leurs "passages", on est sûr de ne jamais saturer la capacité (espace) de nos serveurs en mettant plus de tables en ciruculation. Ensuite, là on a fait circuler la table du serveur 1 dans les autres serveur, mais rien nous ne empêche de faire circuler toutes les autres tables au même temps. Autrement dit , lorsque le serveur 1 envoie sa table au serveur 2, parallèlement le serveur 2 peut envoyer sa table au serveur 3 et ainsi de suite.



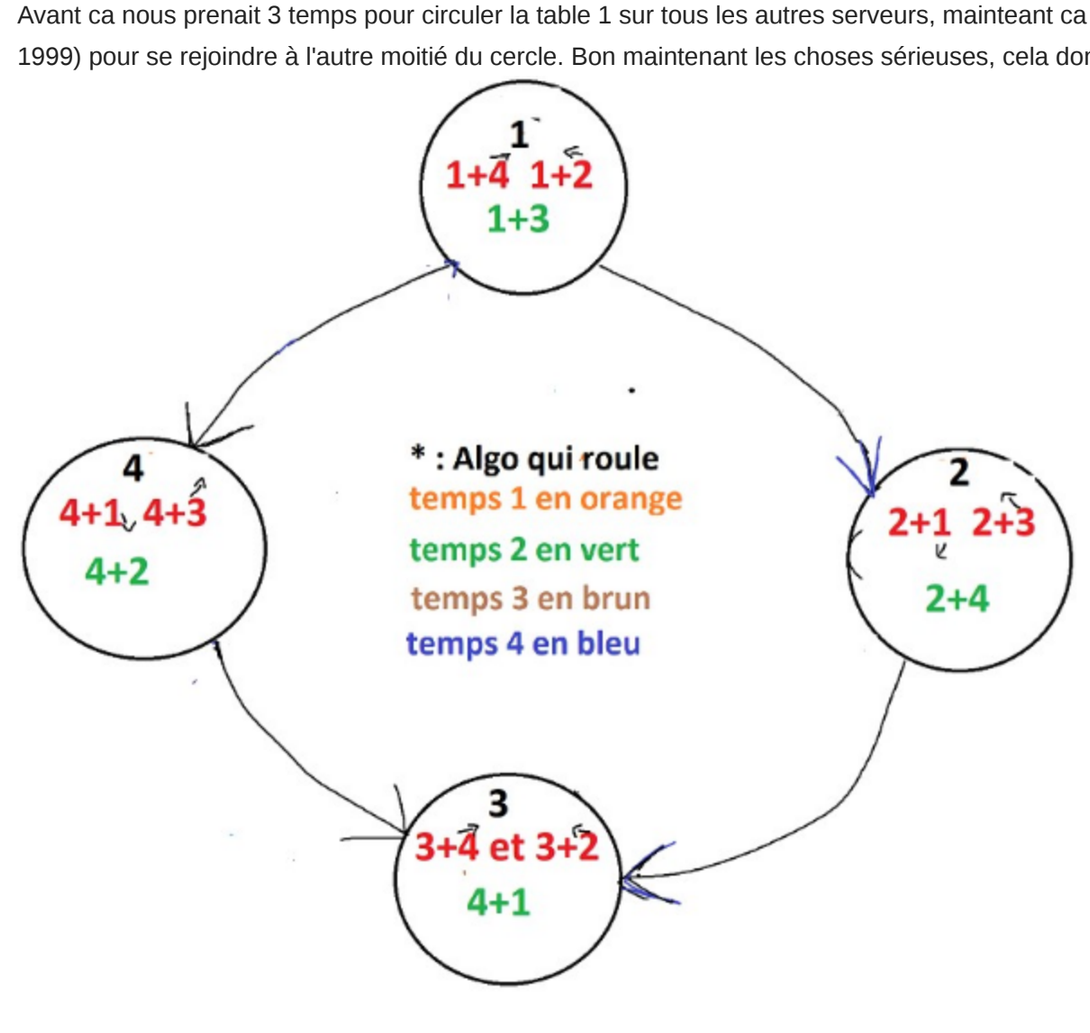
On remarque que **nous avons réussi! Chaque table (serveur) est comparé à toutes les autres tables (serveurs)**. On remarque aussi que ca nous prend 3 temps pour faire circuler chaque table d'un bout à l'autre. Ce qui est logique puisque il y a 4 serveurs et donc 4-1 arcs à traverser si on veut faire le tour sans revenir à la case départ. Là nous avons seulement 3 arcs, mais si nous avions 1200 serveurs, alors chaque table doit traverser 1999 arcs. Peut-t-on faire mieux?

Deux sens

Et si plutôt que chaque server envoie sa table dans la même direction du cercle, chaque serveur envoie sa table dans les deux directions opposées. Illustrons d'abord avec un seul serveur:



Avant ca nous prenait 3 temps pour circuler la table 1 sur tous les autres serveurs, mainteant ca nous prend seulement 2 temps, ce qui est beaucoup mieux. En effet, selon ce raisonnement, si nous avions 1200 serveurs (tables), chaque table doit seulement traverser 600 arcs (plutôt que 1999) pour se rejoindre à l'autre moitié du cercle. Bon maintenant les choses sérieuses, cela donne quoi si on déplace tout en parallèle?



J'ai tout calculer visuellement sur paint et j'ai pu généraliser. En effet, s'il existe k serveurs. Alors avec cette méthode, ca prend k/2 temps pour faire circuler tous les serveurs d'un bout à l'autre. Si k est impair alors on prend le plancher de k/2 et tout fonctionne à merveil.

Bon maintenant, il reste une chose importante. Si on dit pas à nos serveurs quand arrêter de faire circuler une table, alors cela risque de tourner à l'infini ? Oui. Donc on propose d'en plus de passer la table en question , on passe aussi une variable *pass = int (k/2)* et à chaque fois avant de passer la table on fait -1. Et quand *pass == 0* alors on arrêt de faire circuler la table. Ainsi, on est sûr que notre table ne va pas circuler plus que nécessaire.

Les détails techniques

Pour illustrer en "temps" abstraits et avec des images c'était simple. Mais combien de temps ca prend réellement ? Et surtout comment fonctionne l'algorithme concrètement?

D'abord , nous savons que nous avons 1200 serveurs. Dans chaque serveur i (où i compris entre 1 et 1200) , il y'a une table Ti. Cette table, comme nous avons conclus précédemment, doit circuler int(1200/2) fois, soit *pass = 600* dans chaque serveur.

1. La première étape est de rouler notre algorithme une première fois sur chacun de nos serveur en parallèle closestPairs(Ti)

Biensûre, pour appliquer l'algorithme, il faut d'abord read(Ti), ce qui prend 750Go */ 3Go/s = 250 secondes. Et donc vu que nous devons calculer O(n log n), 250 secondes* log(250) = 600 secondes. Donc 10 minutes. Ps : 750Go est la taille de la table dans chaque serveur : *((900 Tera* 1000) GO / 1200 Serveurs)*

En appliquant l'algorithme, on écrit une array de 1000 paires d'étoiles , soit une array de 1200 (double x , double y, double z) ou de (int index, int index2, double distance). Donc , 28 800 bytes (ou le double) dans le pire cas. Évidemment, notre vitesse d'écriture est très puissante et écrit (ou met à jour) cette table de paires instantanément.

1. Aussi au même temps qu'on lit notre table, chaque serveur envoie sa table à ses deux serveurs adjacents (qui eux aussi sont occupés à envoyer leurs tables puisque tout est en parallèle).

D'abord on envoie pas seulement la table, mais un tuple *(pass--, Table i)* pour savoir combien de fois chaque table est passée. Aussi, puisque la vitesse de communcation entre les serveurs est de 1Go par seconde, cette vitesse doit être dévisée par 4 puisque chaque serveur envoie sa table à deux serveurs et chaque serveur recoit une nouvelle table de deux autres serveurs. Donc , la vitesse est de 250Mo/s pour chacunes de ces 4 opérations. Puisque nous avons 750Go à déplacer, cela nous prend 3000 secondes au total, soit 50 minutes.

1. Après chaque fin de communication, on read les nouvelles tables pour appliquer closestPairs(Ti + T_reçues) . Donc chaque serveur doit appliquer l'algo sur sa propre table + la table reçue du serveur à gauche. Aussi sur sa propre table + la table reçue de la droite.

Biensûre, puisque nous devons read(750Go *2) deux fois, cela va prendre 3000Go/ 3Go/s = 1000 secondes. Donc en O(n log n), 1000* log (1000) = 3000 secondes au total. Soit 50 minutes. Au même temps, l'algorithme met à jour instantanément nos 1000 meilleures paires.

1. Lorsque nous sommes en train de lires les tables, nous pouvons déjà commencer à transférer les table (50 minutes).

Puisque on le fait en parallèle (heureusement) les temps ne s'additionne pas. Donc seulement 50 minutes pour l'étape 2.

1. Une fois les tables transférées , on doit les delete des anciens serveurs pour faire de la place à de nouvelles tables.

On veut supprimer 750Go * 2 , ce qui se fait en 500 secondes. Soit 8 minutes. On ne fait pas cette étape en parallèle puisque ca va revenir au même, vu qu'on va ralentir l'écriture des nouvelles tables.

Biensûre, on répète les étapes 2 et 3 jusqu'à ce que chaque table aie traversé 600 serveurs (donc tous les pass==0).

Au niveau du temps, cela donne : 50 minutes + 599 * (50 minutes + 8 minutes) = 34 782 minutes. **Donc environ 24 jours**. Ce qui est moins que 30 jours.

Une dernière note pour que tout cela fonctionne: D'abord, nous avions calculé que chaque table dans chaque disque dur occupe seulement 18,75% de l'espace (750Go/ 4 Tera). Cela a permis notre circulation de tables dans les serveurs puisque nous manquons jamais d'espace (à condition biensûr d'effacer les tables au fur et à mesure). De plus, puisque on lit toujours tout directement sur le disque dur, on ne sature jamais la mémoire vive. En effet, lorsque nous avons fais nos calculs pour l'algorithme, nous avons lu directement sur le disque dur en temps O(n log n). Sur la mémoire vive, la gestion de la mémoire aurait été très difficile vu la taille de nos données.

B. Compter combien d'étoiles il y a dans chaque catégorie.

Mise en contexte

Énoncé : Pour chaque étoile la table contient ... et sa catégorie représentée par un entier entre 1 et 10 (1 INT8). Nous avons 7300 milliards d'étoiles au total répartis à travers 1400 serveurs.

Donc, nous avons environ 5-6 milliards d'étoiles dans chaque serveur. Chaque entrée dans la table dans un serveur représente une étoile. Et pour chaque entrée la même colonne donne la catégorie de l'étoile.

Index étoile	Position	Catégorie	...
0	(1999.0, 28928.0, 3131.0)	0	...
1	(3485.0, 19211.0, 2992.0)	3	...
2	(9425.0, 39211.0, 5992.0)	9	...
...	...	...	...
5 214 285 714	(9425.0, 59211.0, 1992.0)	4	...

Nous savons d'ailleurs que les données (les tables) sont déjà assignées aux serveurs.

Étape 1

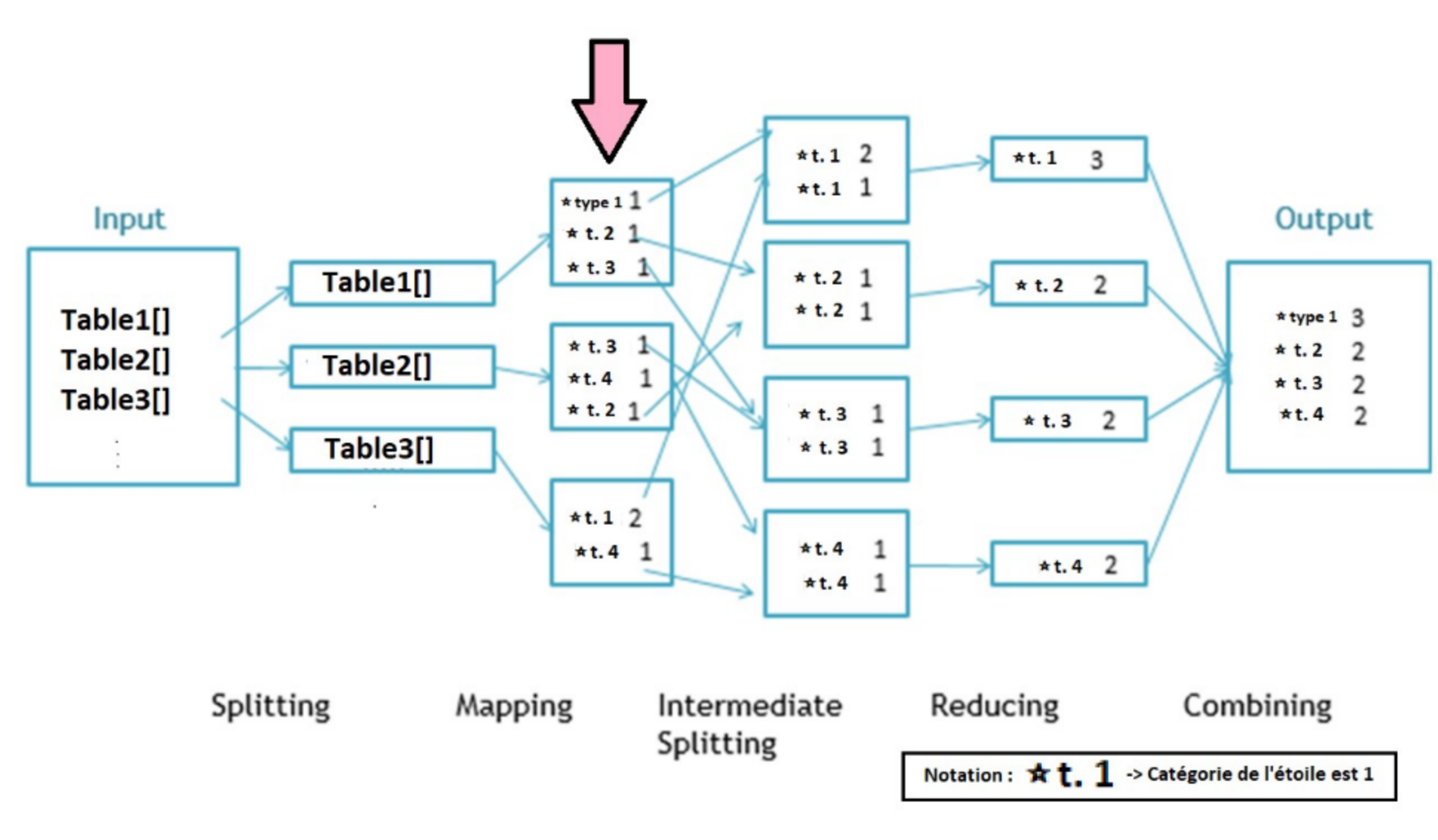
La taille de chaque table comme discuté précédemment est de **750 GO** calculé avec *((900 Tera* 1000) GO / 1200 Serveurs)*. Il suffit d'une lecture en O(n) sur un serveur pour savoir il y'a combien d'étoiles dans chacune des catégories avec un algorithme comme le suivant:

CompterCatégories(Table[étoiles]):

```
S = initializes array of length 10 filled with 0's

for étoile in Table[Étoiles]:
    S [étoile.getCatégorie-1] += 1

return S
```



Ainsi, à la fin de notre calcul en parallèle , chaque serveur aura une array d'une taille de 10 int. Chaque élément dans ce tableau peut être lu comme suit : S[0]=(étoile de catégorie 1 , nombre d'étoiles de catégorie 1) , ... , S[i]=(Étoile de Catégorie i+1, nombre d'étoiles de catégorie i+1).

le temps estimé pour cette étape est de O(n) puisque nous allons lire les n entrées de la table de tous les serveurs parallèlement. Ce qui risque de prendre le plus de temps est probablement la lecture des données. On va assumer que notre algorithme est vraiment naif et lit chaque entrée au complet au lieu de lire simplement les catégories. Donc, pour lire toutes les entrées (750GO) nous avons besoin de 750GO/ 3Go par seconde = **250 secondes**. Soit 4 minutes environ. Enfin, l'écriture se fait au même temps que la lecture et nous allons assumer que l'écriture dans le tableau S se fait instantènement dans chaque serveur puisque nous devons écrire un int dans un tableau de taille 10 à chaque fois et vu notre vitesse d'éciture ce n'est vraiment pas un problème.

Étape 2

Puisque nous avons 1200 serveurs, donc 1200 arrays S produites dans chaque serveur après l'étape 1. Il faut faire la "somme" de toutes ces arrays pour avoir le nombre d'étoiles pour chaque catégories dans tous les serveurs réunis. Nous allons proposer une approche simple à comprendre et qui sera toute de même efficace vu la petite taille de données qu'on va manipuler dans notre cas.

Nous allons utiliser 10 serveurs arbitraires. Vu que nous avons 1200 serveurs et 10 catégories d'étoiles, faisons simple et utilisons les 10 premiers serveurs. Soit le serveur 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

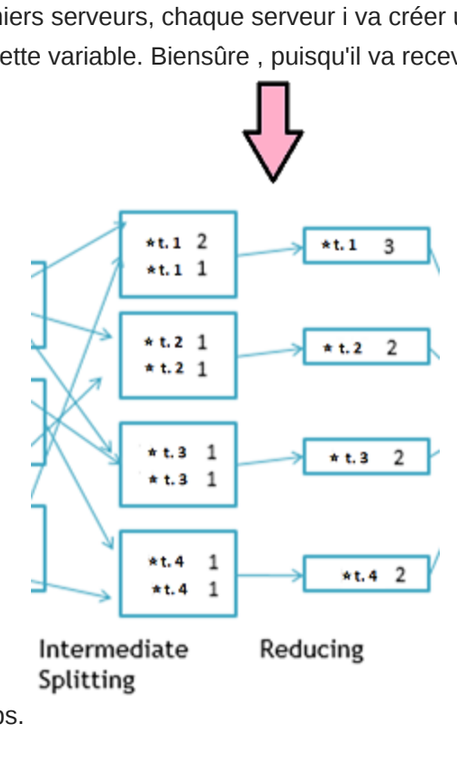
Le serveur 1 va recevoir l'information dans S[0] (soit le nombre d'étoiles dans la catégorie 1) de tous les autres serveurs. ... Le serveur 10 va recevoir l'information dans S[9] (soit le nombre d'étoiles dans la catégorie 10) de tous les autres serveurs.

Maintenant, il est intéressant de se demander combien de temps cette étape va prendre ? Nous savons que les serveurs peuvent communiquer entre eux à une vitesse de 1Go/sec. Nous allons assumer que nous avons pour chaque serveur une vitesse de 1Go/s de download et 1Go/s de upload. Puisque les 10 premiers serveurs vont recevoir à la fois des informations de tous les autres serveurs , leurs vitesse de download sera dévisées.

Le serveur 1 va recevoir de l'information de 1200 serveur (on va l'inclure lui-même , cela ne change pas grand chose) au même temps. Il ne peut que télécharger 1Go à la fois. Cela dit, quand on y'pense objectivement, le serveur 1 va recevoir 1200 valeurs de type int. En informatique, nous savons qu'un int peut être représenté avec 4 bytes. Donc nous avons 4 * 1200 bytes. Soit 4800 bytes. Donc, en réalité le serveur 1 va devoir télécharger 4,8 Kilobyte. Vu notre vitesse de téléchargement , nous allons assumer que c'est instantané.

Qu'en-t-il du côté de l'upload ? Chaque serveur doit envoyer 10 int (son tableau S) aux 10 premiers serveurs. Encore une fois, 10 int c'est seulement 40 bytes. Nous allons encore assumer que c'est instantané. Nos serveurs sont beaucoup trop puissants.

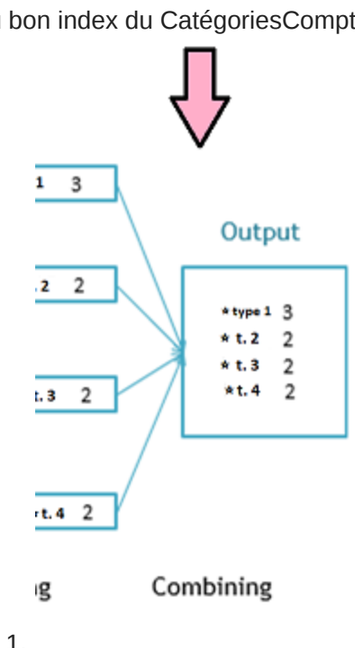
Donc on résume, dans les 10 premiers serveurs, chaque serveur i va créer une nouvelle variable *Catégorie i* de type int qui sera initialisé avec sa propre valeur S[i-1] (Ex Serveur 1 : Catégorie_1 = S[0]). Ensuite , ce serveur i va recevoir de tous les autres serveurs des valeurs de type int et va simplement les aditionner à cette variable. Biensûre , puisqu'il va recevoir toutes les int au même temps, il serait plus sécuritaire de créer une file Buffer = [] , dans laquelle on ajoute toutes les int avant de les aditionner un par un à Catégorie_i. Même avec la file (non obligatoire),



cela ne va pas impacter notre temps.

Étape 3

Maintenant que chacun de nos 10 premiers serveurs contient une donnée Catégorie i (de notre tableau final) , nous voulouns les combiner dans un seul tableau. Bref, nous avons 10 données, nous voulons un tableau de taille 10. La manière la plus simple est de créer un tableau dans le serveur 1. *CatégoriesComptés = []*. Ensuite, on peut déjà mettre dedans la variable *Catégorie_1* comme suit *CatégoriesComptés.append(Catégorie_1)*. Maintenant il suffit que nos 9 autres serveurs envoie leurs données *Catégorie_i* et chacunes des données va être append au tableau comme suit *CatégoriesComptés.append(Catégorie_i)*. Ici, biensûre, il serait préférable que chaque serveur envoie aussi l'index (Serveur i = index i-1) pour que chaque catégorie soit ajoutée au bon index du CatégoriesComptés. Bref à la fin , ce tableau est notre réponse finale. Soit la



somme totale de toutes les catégories. Encore une fois tout cela prend seulement quelques secondes (instantané) puisqu'il suffit de déplacer une très petite quantitée d'informations au serveur 1.

Tout cela nous aura pris moins de 5 minutes.

Algorithme résumé en quelques lignes:

1. appeler l'algorithme CompterCatégories(Table[étoiles]) -> S[chaque catégorie comptée] sur chaque serveur en parallèle
2. Tous les serveur en parallèle envoie S[catégorie] au Serveurs.catégorie. Ex: S[1] de tous les serveurs envoyé au serveur 1.
3. Les 10 premiers serveurs font la somme de ce qu'ils recoivent des autres serveurs et store cela dans var Catégorie_i (Bref, chaque serveur server i(i compris entre 1 et 10) aura une variable Catégorie_i qui est la somme des S[i-1] des 1200serv)
4. Le premier serveur crée une table CatégoriesComptés[] de taille 10. Il append dedans tous les Catégorie_i des 10 serveurs). (En effet,il commence par append.Catégorie_1 , ensuite Catégorie_2 ... Catégorie_10. À la fin ce te tableau est notre résultat)

C. Produisez un classifieur qui, étant donné le vecteur de caractéristiques (23 nombres réels), prédit la catégorie de l'étoile. Il est important que votre classifieur utilise une technique de calcul distribué soit pendant l'entraînement et/ou lors de la prédiction.

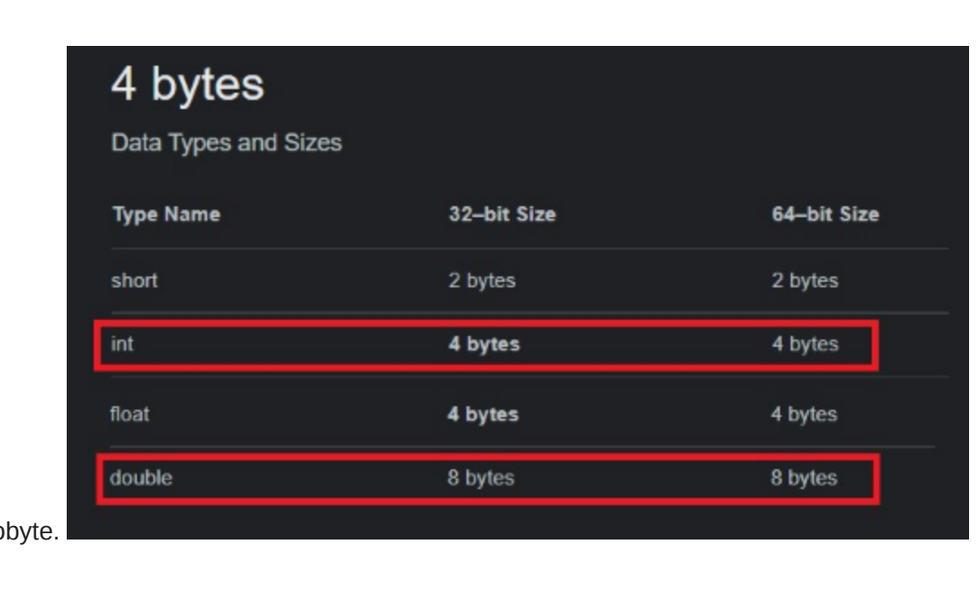
Mise en contexte

Index étoile	...	Catégorie	Caractéristique
0	...	0	(2.4, 1.1, ... 9.2)
1	...	3	(3.2, 9.9, ... 1.8)
...	...	...	...

D'abord , à quoi ressemblent nos tables ?

D'abord (dans un classifieur comme KNN), notre colonne catégorie c'est le label (y) et la colonne caractéristique sont les données sur lesquelles on se base pour classifier. (Pour faire un parallèle avec Mnist, les chiffres c'est les catégories et les vecteurs pixels sont les vecteurs caractéristiques).

Donc, si nous avons un tuple de ce type : (catégorie, vecteur de caractéristiques) , soit un (int, [double *23]) alors la taille de ce tuple est : 4 bytes + 8 bytes* 23 = 188 bytes. Soit 0,188 kilobyte.

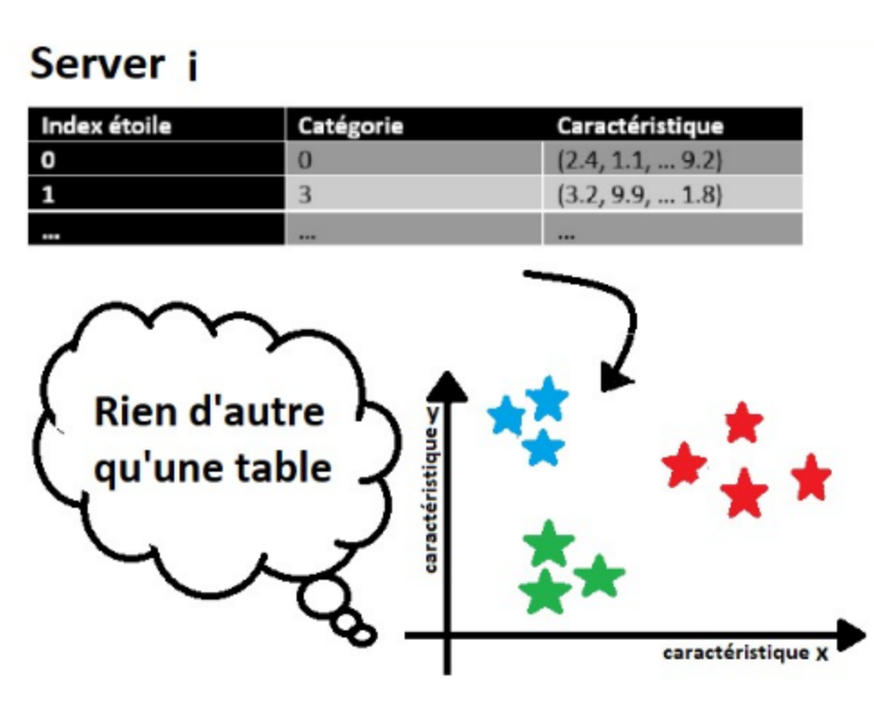


4 bytes

Data Types and Sizes

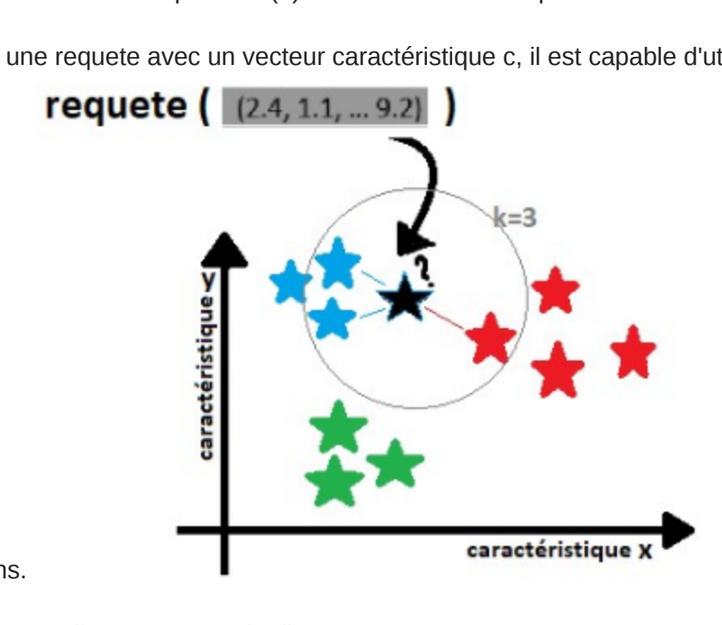
Type Name	32–bit Size	64–bit Size
short	2 bytes	2 bytes
int	4 bytes	4 bytes
float	4 bytes	4 bytes
double	8 bytes	8 bytes

KNN (Ou n'importe quel classifieur)



D'abord, il est clair que chaque serveur peut lire la table sur le disque en O(n) & 3Go/s . Pour ainsi produire la table T:

De plus, il est clair que si ensuite un serveur reçoit une requete avec un vecteur caractéristique c, il est capable d'utiliser la table T plus haut, pour prédire la catégorie de l'étoile. En effet, il calcule la distance de ce vecteur c à tous les vecteurs dont on connait la catégorie dans T et on



prédit une catégorie selon les k plus proches voisins.

Biensûre cela se fait encore une fois en O(n) en lisant T directement sur le disque.

Questionnements et pistes

Si on veut prédire la catégorie d'une étoile avec une requête(caractéristiques[doubles * 23]), comment prédire cette catégorie le plus rapidement possible? On l'envoie à un seul serveur au hasard? À plusieurs serveurs en parallèle et selectionner la catégorie la plus souvent prédite? Y'a-t-il un moyen d'accélérer le résultat ou de le rendre plus précis?

De plus, lors ou après la création de T , peut-t-on améliorer T ? Autrement dit y ajouter des vecteurs pour que T soit plus précis. Par exemple, si le serveur 1 crée T1 et le serveur 2 crée T2, est-ce une bonne idée de combiner T1+T2 ? Si oui comment procéder pour combiner un maximum de tables en parallèles? Combien de tables maximum peut-t-on combiner par serveur ?

Réponses

D'abord, inuitivement, plus notre "modèle" (table) est "petit", moins de temps ça prendera pour prédire la catégorie d'un vecteur caractéristiques c. En effet, si nous avons un serveur A avec une table T1 avec n d'étoiles connues, alors pour pour prédire c , nous devons le comparer à tous les n dans T1. Cela va se faire en n temps. Cela dit, si nous avons une table T1 du serveur 1 et une table T2 du serveur 2 combinées, soit T1+T2= T12 , alors prédire la catégorie de c dans T12 nous prendera 2n temps.

Aussi, il est clair que plus que notre "modèle" (table) est grand, plus on aura de précision dans la prédiction de c. Reprenons l'exemple avec deux serveurs. Si le serveur 1 produit la table T1 et le serveur 2 prododuit table T2. Alors, essayer de prédire la catégorie de c sur T1 ou sur T2 sera moins précis que de prédire c sur T1+T2 (T12).

En d'autres mots, si chaque serveur i contient sa table Ti et on envoie une requête à chaque serveur. Alors les prédictions se feront toutes plus rapidement , mais avec moins de précision. Cependant, si on envoie une requête à un seul serveur avec une grande table (plusieurs tables combinées ex Ti+..+Tj = Ti..j), la réponse sera plus lente, mais de meilleure précision.

Est-ce que nous pouvons faire un "mélange" et trouver un équilibre? Oui, on crée un certains nombre de "super" tables. Ensuite, on envoie nos requetes de prédiction à toutes ces super tables en parallèle. Lorsqu'on a nos prédiction, on garde la prédiction qui revient le plus souvent comme étant la meilleure.

Super Tables

D'abord, re définissons plus rigoureusement ce qui est une super table dans notre contexte. Pour cela rappelons qu'un serveur i produit une table T_i (dans l'algorithme knn). On se sert alors de cette table T_i (toujours dans l'algorithme knn) pour prédire la catégorie d'une étoile en la comparant à toutes les étoiles dans T_i pour trouver ses k plus proches voisins. Si nous avons m serveurs , et chaque serveur i produit une table Ti , alors une super table est une combinaison de m tables T_0..m . Ex, 10 serveurs, 10 tables : T1, T2..., T10. SuperTable_1..10 = T1+T2+..+T10.

Quelle est la manière la plus rapide pour calculer des supers tables? Nous allons utiliser des arbres binaire complet. Où tous les calculs fait à la hauteur d'un arbre sont fait en parallèle. Pour créer un arbre binaire complet, nous pouvons utiliser un algorithme comme le suivant:

Algo BinaryTree (serveurs)

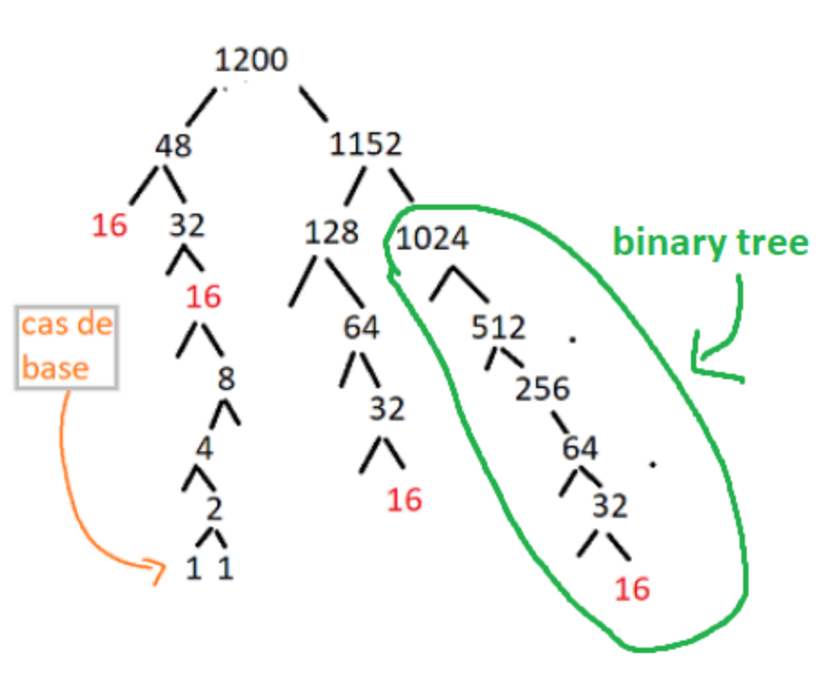
```
width <- puissance de deux <= nombre de serveur (exemple dans notre cas 1024, 512 , ...)

Branche ( serveurs[0..width] ) :

    if |serveurs | > 2

        branche gauche = Branche(serveurs[0..|serveurs|/2])
        branche droite = Branche(serveurs[|serveurs|/2 ... |serveurs|])
        return (branche gauche, branche droite)

    else
        branche gauche = serveur[0]
        branche droite = serveurs[1]
        return (branche gauche, branche droite)
```



Tant que le nombre de serveurs est une puissance de deux, on peut le deviser récursivement pour obtenir un arbre binaire complet de hauteur log_2(width). Biensûre, encore une fois, nous ne voulions pas une seule super table (un seul arbre binaire), mais plusieurs super tables. Pourquoi ?

1. Nos serveurs ont une capacité limitée. Si nous essayons d'écrire toutes nos tables dans un seul serveur, il va probablement saturer.
2. Nous avons précédemment discuté sur comment accélérer la prédiction. Si on envoie la requête (d'un vecteur caractérstique c) pour prédire une catégorie à une grosse table, cela va prendre plus de temps. Tandis que si nous envoyons notre requête à plusieurs petites tables, les calculs se feront plus vite et en parallèle.

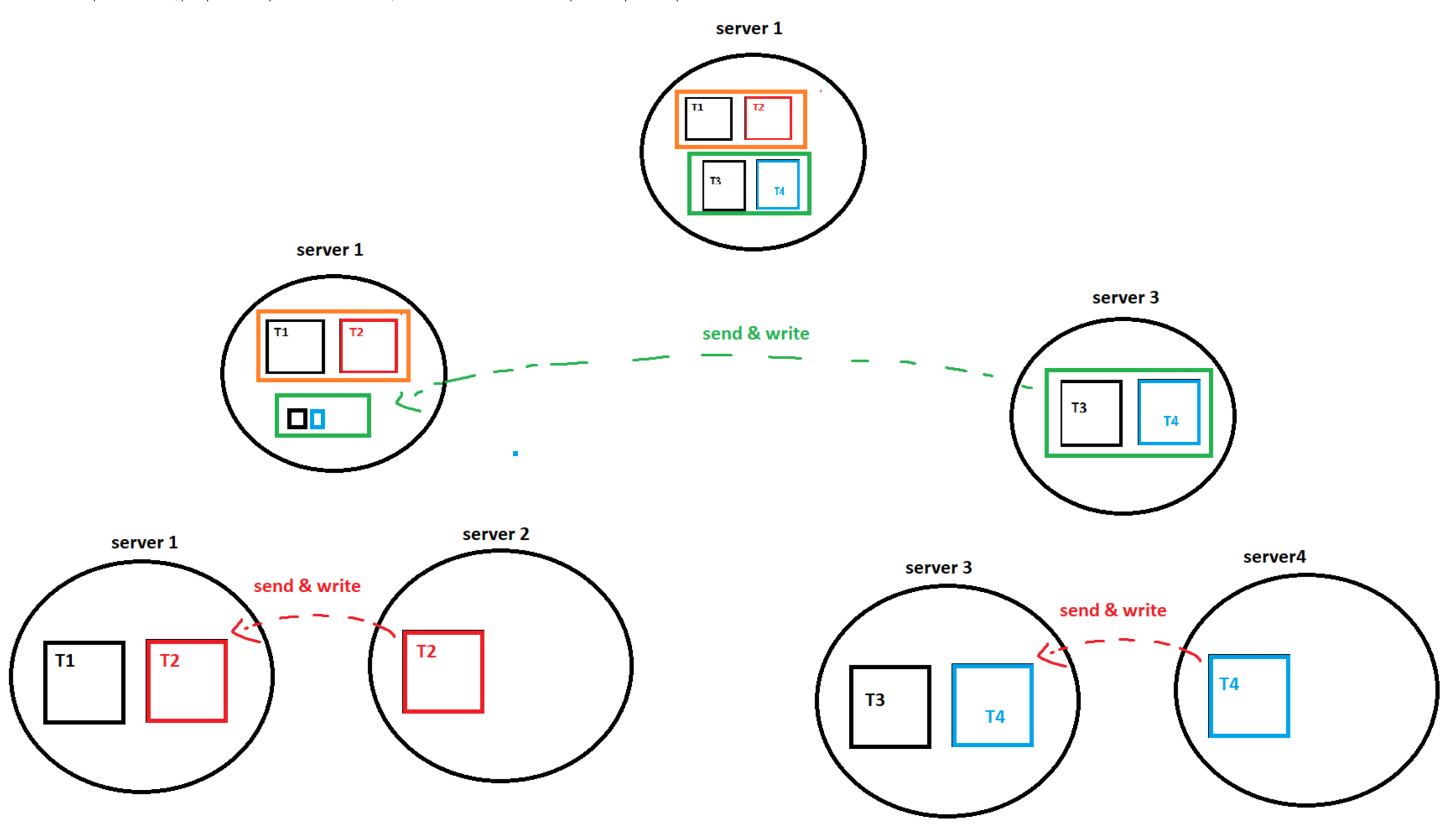
Dans notre cas nous avons 1200 serveurs. La plus grande puissance de 2 et plus petite que 1200 est 1024. Puisque à la fin des calculs de notre arbre binaire, notre super table (la combinaison de toutes tables) sera écrite sur un seul serveur. Il est donc important de ne pas combiner beaucoup trop de tables.

Calcul de la taille d'une Table
Une table a n entrée (chaque étoile) et d dimensions (caractéristiques). Nous avons donc n * d.

Nous savons que nous avons au total 7300 milliards d'étoile répartis de manière balancées dans 1200 serveurs. 7300 Milliards / 1200 serveurs = 6,08 milliards (=n) d'étoile par serveur. Et nous avons 23 caractéristiques, donc d=23. Bref , nous avons n *d = 139,91 Milliards de floats dans la table. float = 4 bytes. 4* 139,91 M = 559,67 Milliards de bytes. Si on convertis nous avons environ 560 Go.

Chaque serveur a une capacité de 4 Tera (soit 4000Go). Et 750Go sont déjà occupée (par les étoiles). Donc, nous avons en réalité 3250 Go de libre. Combien de fois pouvons nous mettre 560 dans 3250? 3250 / 460 = 5,8. Donc , la puissance de deux la plus proche est 4. Nous pouvons créer donc des arbres binaire de hauteur 2 (log_2(4)).

Biensûre dans notre cas, nous avons d = 23 , soit un vecteur de 23 caractéristiques. Nous pouvons , comme nous avons appris dans le cours, réduire la dimensionnalité. Je suis sûr que nous pouvons créer des arbres binaire de hauteur 4 ou plus. Cela fera surement des super tables avec une meilleure précision. Mais , puisque ce n'est pas le but de l'exercice , nous allons nous contenter de super tables produites par des arbres binaire de hauteurs 2.



Combien de super tables de hauteur 2 pouvons et devons nous créer ? Puisque plus on a de super tables, plus on aura de retour sur nos prédictions et donc plus on pourra améliorer notre précision tout en gardant la même rapidité (calculs fait en parallèle), on va en créer beaucoup.

Dans la figure, nous avons utiliser 4 serveurs. Prenons par exemple 1024 serveurs. Nous pouvons donc créer 256 super tables. Les serveurs {1,2..,4} , {5,..8}, {9,..12}, ..., {1021,1024}. Bref, on crée des arbres binaire de hauteurs 2 en groupant par 4 en ordre nos 1024 premiers serveurs. une fois cela fait, on commence nos calculs en parallèle.

1. Appliquer algo Knn pour produire table dans chaque serveur du niveau i
2. Faire un transfert de la table crée vers le serveur à gauche dans chaque niveau i
3. Recommencer 2 tant que nous avons pas atteint la racine de l'arbre

Pour créer une super table, on a besoin d'une lecture de 750GO , et d'une écriture de 560GO. Puisque cela se fait en parallèle, donc Max(750,560)/ 3Go/sec = 250 secondes : 4 minutes.

Ensuite , on doit transférer à gauche au total 3 tables de 560 Go (Selon image). 560Go * 3 = 1680Go. 1680Go / 3Gb/s = 560 secondes. 9 minutes.

Une super table (de hauteur 2) met environ 14 minutes à se créer. Puisque nos 256 super tables vont se créer au même temps, elles vont toutes se créer en 14 minutes.

Prédiction

C'est ici que ca devient intéressant. Ademettons que nous voulons prédire la catégorie d'un vecteur caractéristique c, ou de plusieurs vecteurs caractéristiques (venant des 1200-1024 = 176 serveurs restants). Nous avons devant nous 256 super tables sur lesquelles faire nos prédictions en parallèle!

Dans le cas où nous avons un seul vecteur caractéristique à prédire :

1. Nous envoyons ce vecteur (instanténement aux 256 serveurs {1,5,...,1021} (Super Tables)) à partir d'un serveur Envoyeur.
2. Chaque serveur Receveur applique l'algo Knn en O(n) et doit donc lire 560 * 4Go = 2240Go . En 2240/ 3Go/s = 746 secondes. Soit 12 minutes et demi.
3. Chaque serveur Receveur renvoie la catégorie prédite (int) instantanément au serveur Renvoyeur qui se retrouve avec un tableau de 256 int. Il calcule instantanément quelle catégorie revient le plus souvent et c'est la réponse qu'on cherche.

Dans le cas où nous avons k vecteurs caractéristique à prédire :

1. On devise nos 256 vecteurs par k afin de partager la tâche sur plusieurs super tables. Et chaque (ou le) serveur Envoyeur envoie son (ou ses) vecteur caractéristiques à 256/k serveurs Receveurs non-occupés.
2. Chaque serveur Receveur applique l'algo Knn en O(n) et doit donc lire 560 * 4Go = 2240Go . En 2240/ 3Go/s = 746 secondes. Soit 12 minutes et demi.
3. Chaque serveur Receveur renvoie la catégorie prédite (int) instantanément au serveur Envoyeur qui se retrouve avec un tableau de 256/k int. Il calcule instantanément quelle catégorie revient le plus souvent et c'est la réponse qu'on cherche.

Conclusion

Avec notre méthode, on cherche l'équilibre entre la performance et la précision. Faire des super tables avec des hauteurs plus que 2 n'est pas recommandé puisque nous avons remarqué que pour seulement 4 tables combinées, c'est 12 minutes "d'entrainement". Biensûre, nous pouvons faire des super tables de hauteur 1 pour réduire le temps de moitié. Ou ne simplement pas utiliser les super tables. Mais cela est au prix d'avoir moins de précision.