

## AT7300 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.

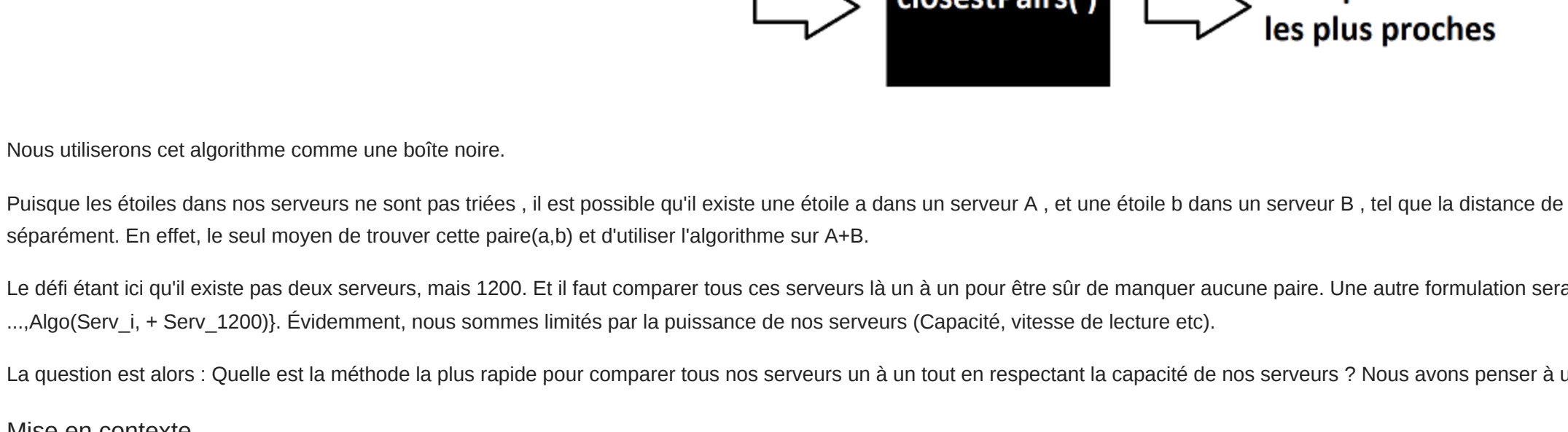
Pour chaque étoile la table contient la position dans l'espace 3D (3 FLOAT64), la luminosité apparente de l'étoile (1 FLOAT32) et sa catégorie représentée par un entier entre 1 et 10 (1 INT8) et un vecteurs de 23 caractéristiques physiques (23 FLOAT32). La taille des données qui sont stockées de façon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

Les données sont réparties de façon équilibrée sur 1200 serveurs avec un processeur rapide, 128 Gigaoctets de mémoire vive et utilisant chacun un disque SSD de capacité 4 Téracoctets 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. Recherchez 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 exsist pas deux serveurs, mais 1200. Et il faut comparer tous ces serveurs l'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\_j + Serv\_j), ...Algo(Serv\_i + Serv\_j + 1200). Évidemment, nous sommes tentés 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 aurons penser à un algorithme simple que nous allons prendre le temps d'expliquer.

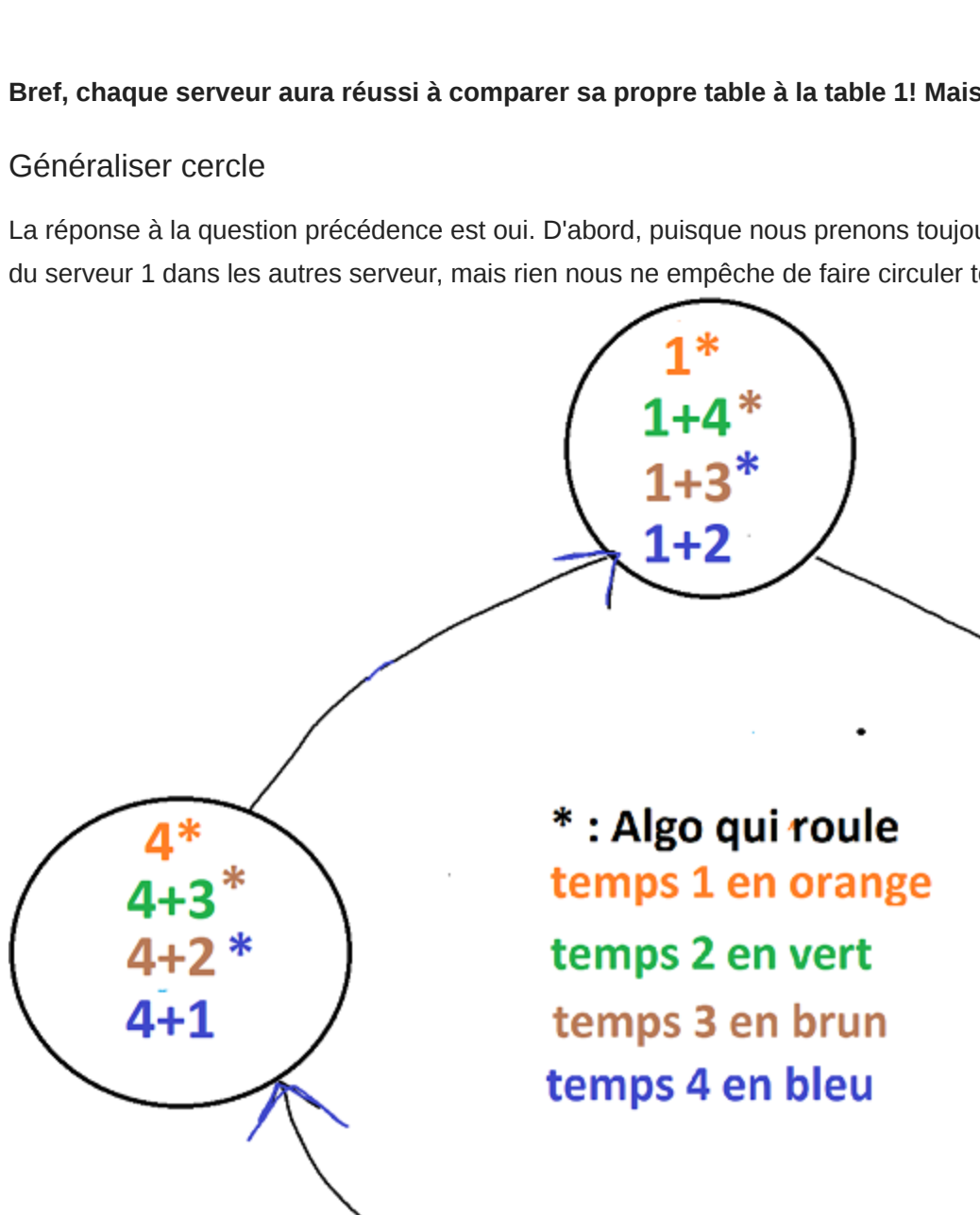
Mise en contexte

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

Cela dit, comme nous nous avons 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 router closestPairs(), 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 :

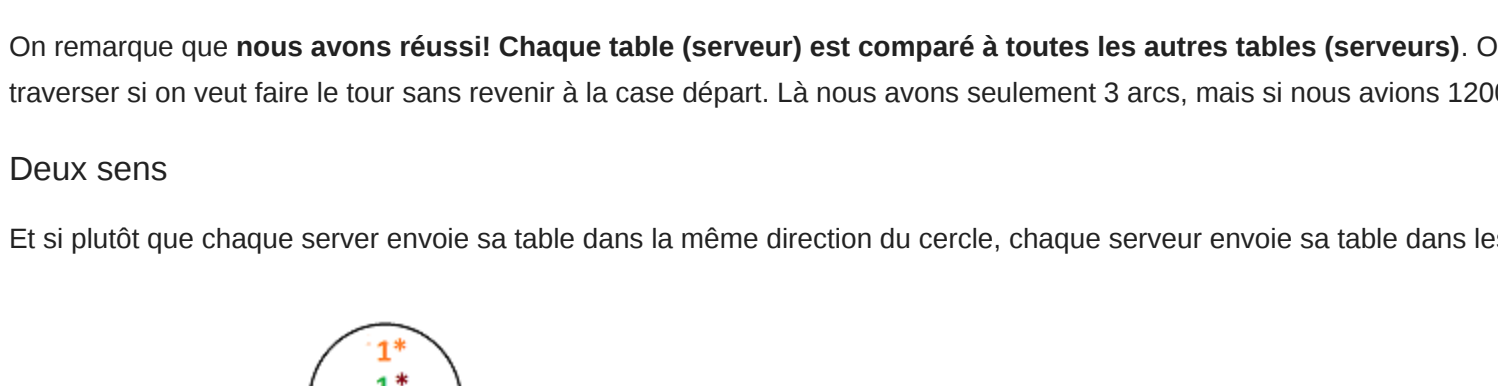
1. Premier temps : closestPairs() sur la table 1 de chaque serveur 1 en parallèle ( où  $i \in \{1,2,3,4\}$  )
2. Deuxième temps : Envoyer la table 1 dans le serveur 2
3. Troisième temps : rouler closestPairs(table1 + table2) et bien sûr on met à jour les meilleures paires
3. Troisième temps : On commence 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 copiée au serveur 3
5. On recommence ce qu'on a fait plus haut ( soit closestPairs(table1 + 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

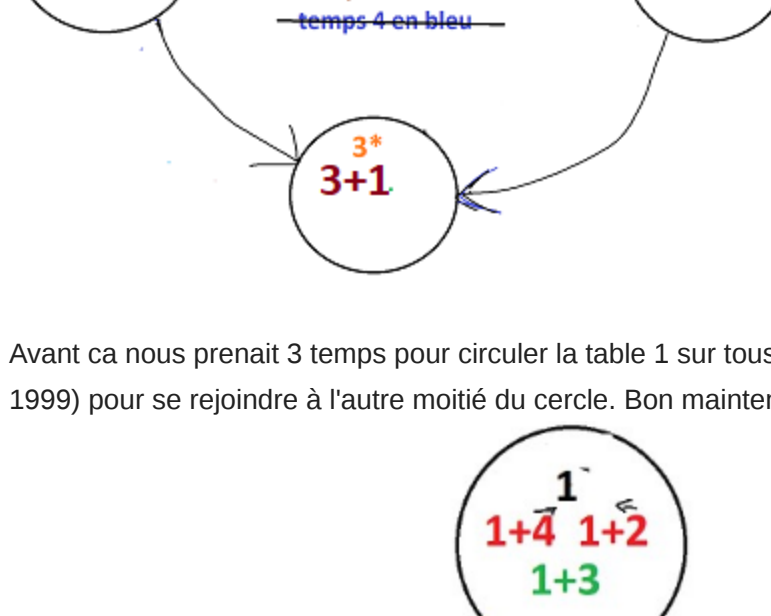
La réponse à la question précédente est oui. D'abord, puisque nous prenons toujours le temps d'effacer les tables après leurs "passages", on est sûr de ne jamais saturer la capacité (espace) de nos serveurs en mettant plus de tables en circulation. Ensuite, là on a fait circuler la table 1s serveur 1 dans les autres serveurs mais nous ne sommes pas obligés 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.



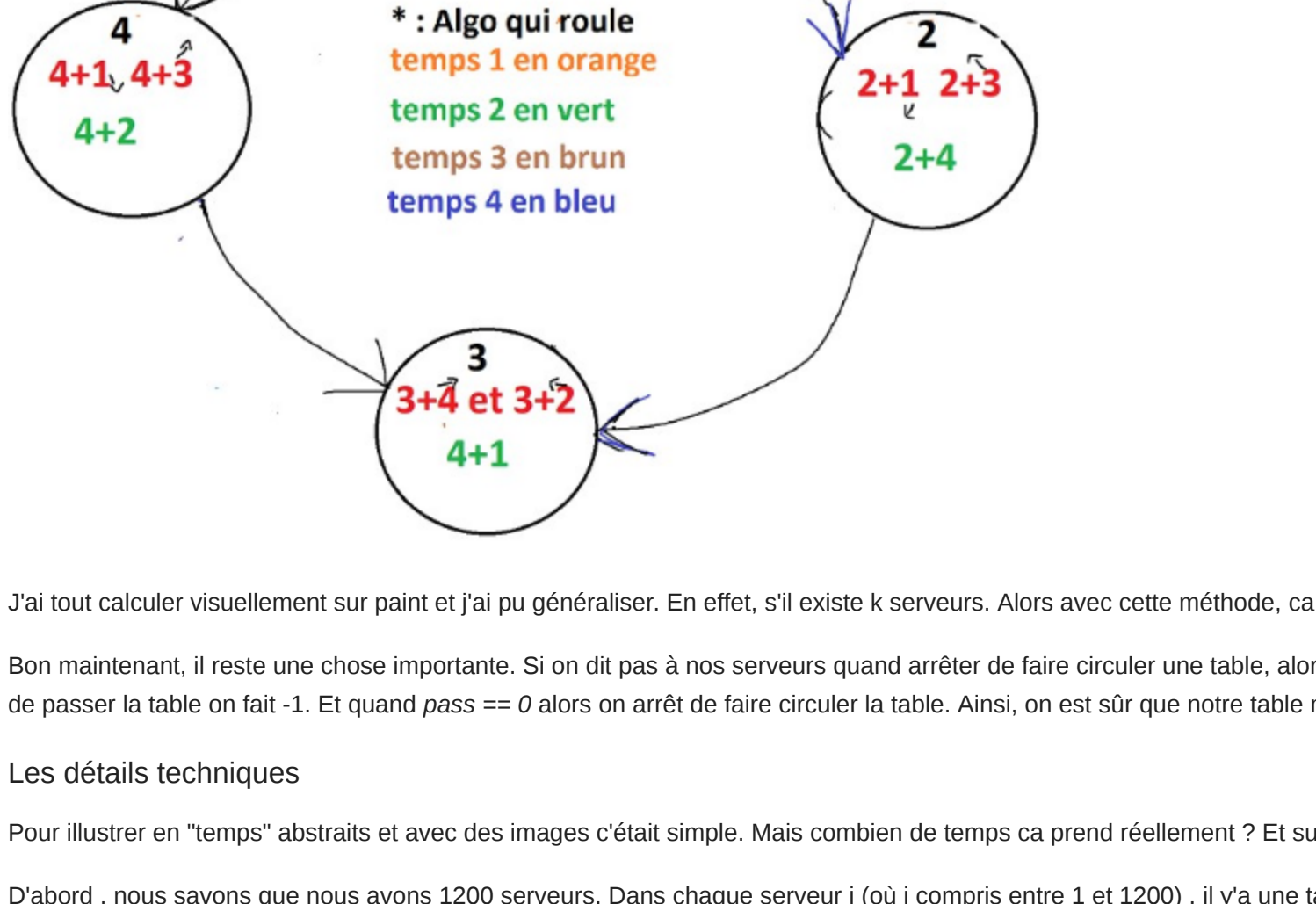
On remarque que nous avons réussi à Chaque table (serveur) est comparé à toutes les autres tables (serveurs). On remarque aussi que ca nous prend 3 temps pour faire circuler chaque table d'un bout à l'autre. Ce qui est logique puisque il y a 4 serveurs et donc 4-1 arcs à traverser si on veut faire le tour sans revenir à la case départ. Là nous avons seulement 3 arcs, mais si nous avons 1200 serveurs, alors chaque table doit traverser 1999 arcs. Peut-on faire mieux?

Deux sens

Et si plutôt que chaque serveur 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 que nous prenons 3 temps pour circuler la table 1 sur tous les autres serveurs, maintenant ca nous prend seulement 2 temps, ce qui est beaucoup mieux. En effet, selon ce raisonnement, si nous avons 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 paper 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 merveille.

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 arrête 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?

Ordon, nous savons que nous avons 1200 serveurs. Dans chaque serveur i (où i compris entre 1 et 1200), il y a une table T\_i. Cette table, comme nous avons conclu précédemment, doit circuler int(1200/2) fois, soit pass = 600 dans chaque serveur.

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

Bien sûr, puisque nous devons reader 750Go 2 deux fois, cela va prendre 3000Go / 3Go/s = 2000 secondes. Donc en  $O(n \log n)$ , 1000 log(1000) = 3000 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) de (int index, int index2, double distance). Donc, 280 000 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 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).

Ordon on envoie pas seulement la table, mais un table pass--. Table i pour savoir combien de fois chaque table est passée. Aussi, puisque la vitesse de communication entre les serveurs est de 1Go par seconde, cette vitesse doit être divisée par 4 puisque chaque serveur envoie sa table à deux serveurs et chaque serveur reçoit une nouvelle table de deux autres serveurs. Donc, la vitesse est de 250Mo/s pour chacune 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 lit les nouvelles tables pour closestPairs(T\_i + T\_j\_requis). 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.

Bien sûr, puisque nous devons reader 750Go 2 deux fois, cela va prendre 3000Go / 3Go/s = 2000 secondes. Donc en  $O(n \log n)$ , 1000 log(1000) = 3000 secondes. Donc 10 minutes. Au même temps, chaque serveur met à jour instantanément nos 1000 meilleures paires.

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

Puisque on le fait en parallèle (heureusement) les temps ne s'additionnent 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 rater l'écriture des nouvelles tables.

Bien sûr, on répète les étapes 2 et 3 jusqu'à ce que chaque table ait 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 de ne pas effacer les tables au fur et à mesure). De plus, puisque on lit toujours tout directement sur le disque dur, on ne saturer jamais la mémoire vive. En effet, lorsque nous avons fini 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

Encore : 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, 299.0, 3.131.0)	0	...
1	(1485.0, 19211.0, 2992.0)	3	...
2	(9425.0, 59211.0, 9992.0)	9	...
...	...	...	...
5214 285 714	(9425.0, 59211.0, 9992.0)	4	...

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

Étape 1

L'idée 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:

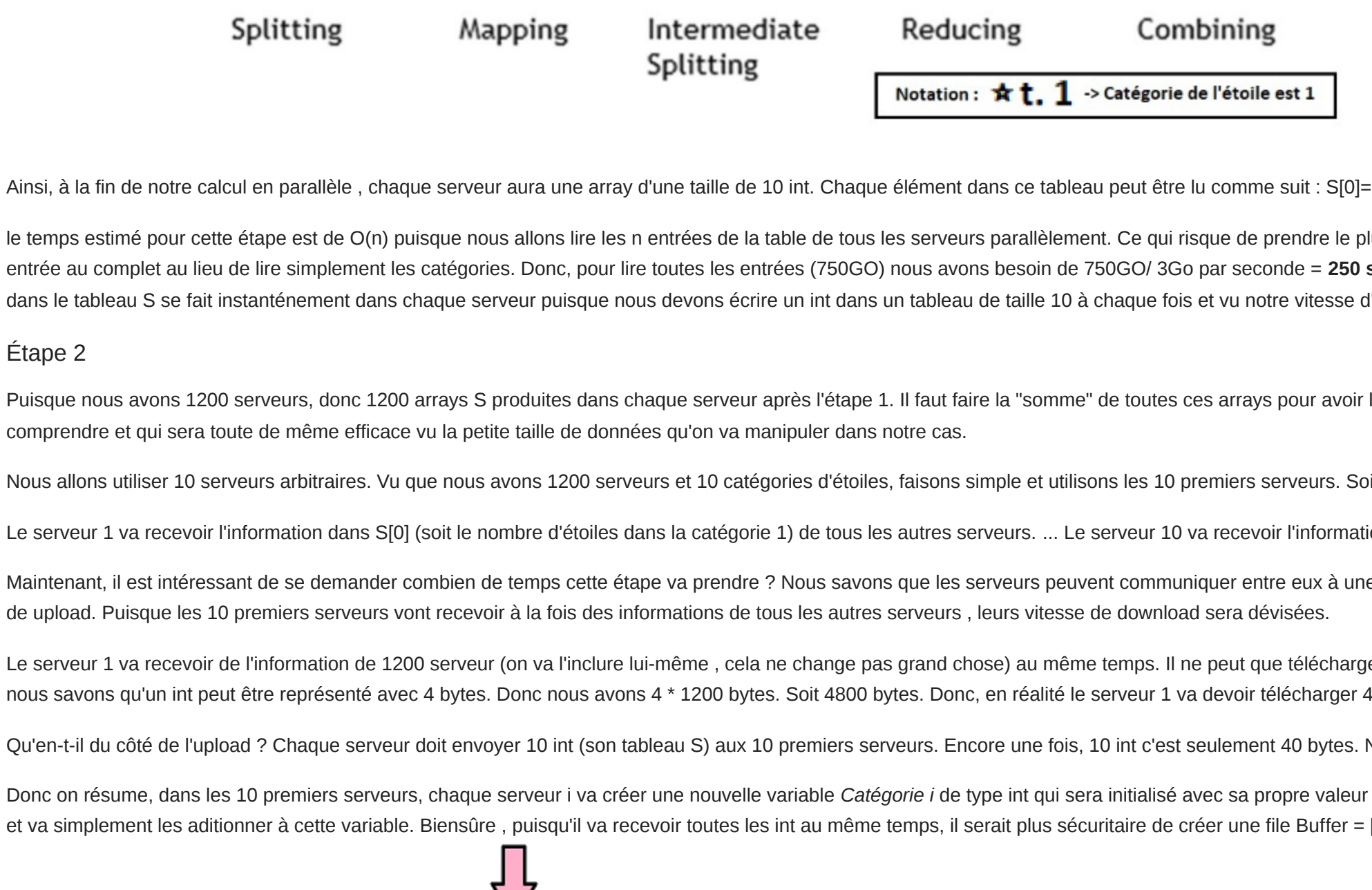
CompterCategories(Table[d'étoiles]):

```
s = initialize array of length 10 filled with 0's
```

```
for étoile in Table[d'étoiles]:
```

```
    s [etoile.categ+1] += 1
```

```
return s
```



Ainsi, à la fin de notre calcul en parallèle, chaque serveur aura une array d'un table 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[9](étoile de Catégorie +1, nombre d'étoiles de catégorie +1).

Le temps estimé pour cette étape est de  $O(n)$  puisque nous allons lire le n-entree 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 naïf et lit chaque condition besoin d'effacer les tables au fur et à mesure). De plus, puisque on lit toujours tout directement sur le disque dur, on ne saturer jamais la mémoire vive. En effet, lorsque nous avons fini 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.

Puisque nous avons 1200 serveurs, donc 1200 array 5 produits 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 table 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.

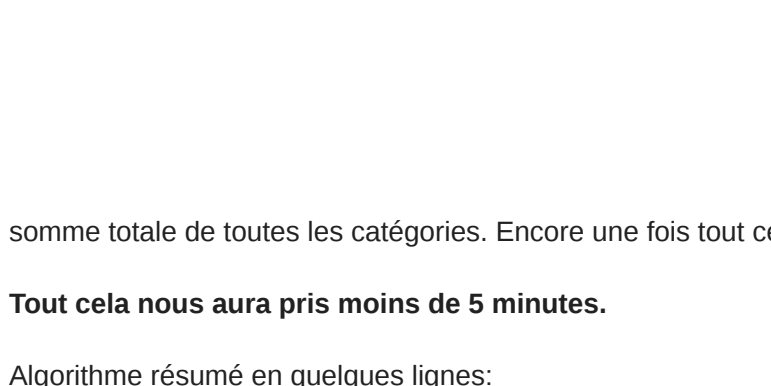
Le serveur 1 va recevoir l'information dans S[0] (soit le nombre d'étoiles dans la catégorie 1) de tous les autres serveurs... Le serveur 10 va recevoir l'information dans S[9] (soit le nombre d'étoiles dans la catégorie 10) de tous les autres serveurs.

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, leur vitesse de download sera divisée.

Le serveur 1 va recevoir de l'information de 1200 serveurs (on va l'indiquer lui-même, cela ne change pas grand chose) au même temps, il ne peut 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 il du côté de l'upload? Chaque serveur doit envoyer 10 int (pour 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.

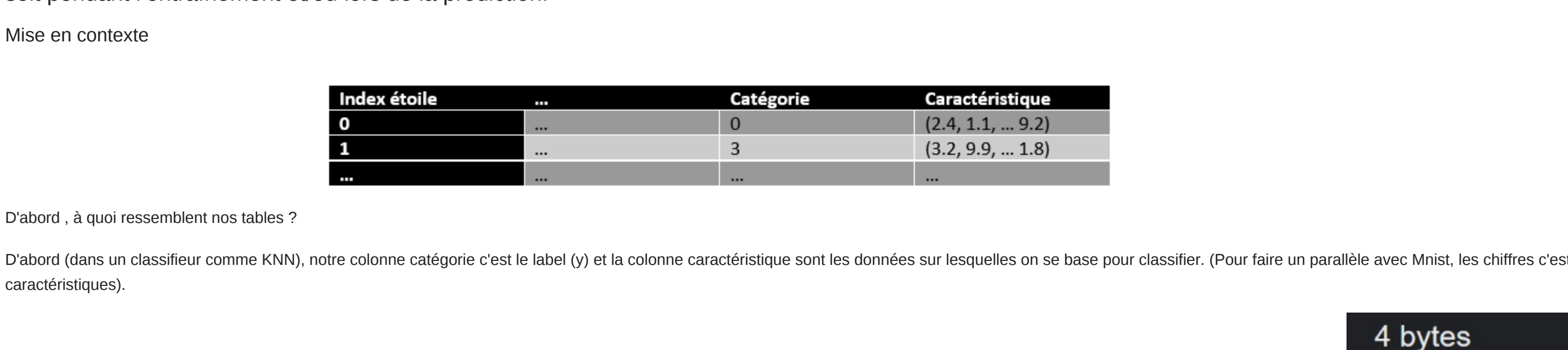
Donc en résumé, dans les 10 premiers serveurs, chaque serveur va créer une nouvelle variable Catégorie\_i de type int qui sera initialisé avec la propre valeur S[i] (Ex: Serveur 1 - Catégorie\_1 = S[0]). Ensuite, ce serveur 1 va recevoir de tous les autres serveurs des valeurs de type int et va simplement les additionner à cette variable. Bien sûr, 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 additionner 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 (le dit notre tableau final), nous voulons 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. CatégorieCompte = []. Ensuite, on peut déjà mettre dans la variable Catégorie\_i comme suit CatégorieCompte.append(Catégorie\_i). Maintenant il suffit que nos 9 autres serveurs envoient leurs données Catégorie\_i. Et chacune des données va être ajoutée au tableau comme suit CatégorieCompte.append(Catégorie\_i). Ici, bien sûr, il serait préférable que chaque serveur envoie aussi l'index (Serveur = index +1) pour que chaque catégorie soit ajoutée au bon index du CatégorieCompte. 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 quantité d'informations au serveur 1.

Tout cela nous aura pris moins de 5 minutes.

Algorithme résumé en quelques lignes:

1. appeler l'algorithme CompterCategories(Table[d'étoiles]) > S[chaque catégorie comptée] sur chaque serveur en parallèle
2. Tous les serveur en parallèle envoient S[catégorie] au Serveurs catégorie. Ex: S[0] de tous les serveurs envoient au serveur 1
3. Les 10 premiers serveurs font la somme de ce qu'ils reçoivent des autres serveurs et store cela dans var Catégorie\_j (Bref, chaque serveur server i) compris entre 1 et 10 aura une variable Catégorie\_j, qui est la somme des S[i-1] des 1200serv)
4. Le premier serveur crée une table CatégorieCompte de plus ou moins 10 (pour tableau S) des 10 catégories. (En effet, il commence par append(Catégorie\_1), ensuite Catégorie\_2, ... Catégorie\_10. À la fin ce tableau est notre résultat)

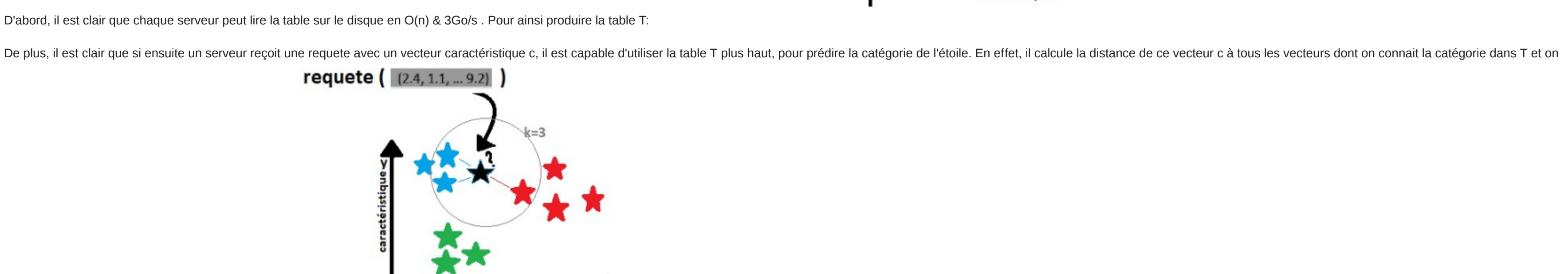
**C. Produisez un classifieur qui, étant donné le vecteur de caractéristiques (23 nombres réels), prédit la catégorie de l'étoile. Il est important que votre classifieur utilise une technique de calcul distribué soit pendant l'entraînement et/ou lors de la prédiction.**

Mise en contexte

Index étoile	...	Catégorie	Caractéristique
0	...	0	(2.5, 1.1, ..., 9.2)
1	...	3	(3.2, 9.5, ..., 1.8)
...	...	...	...

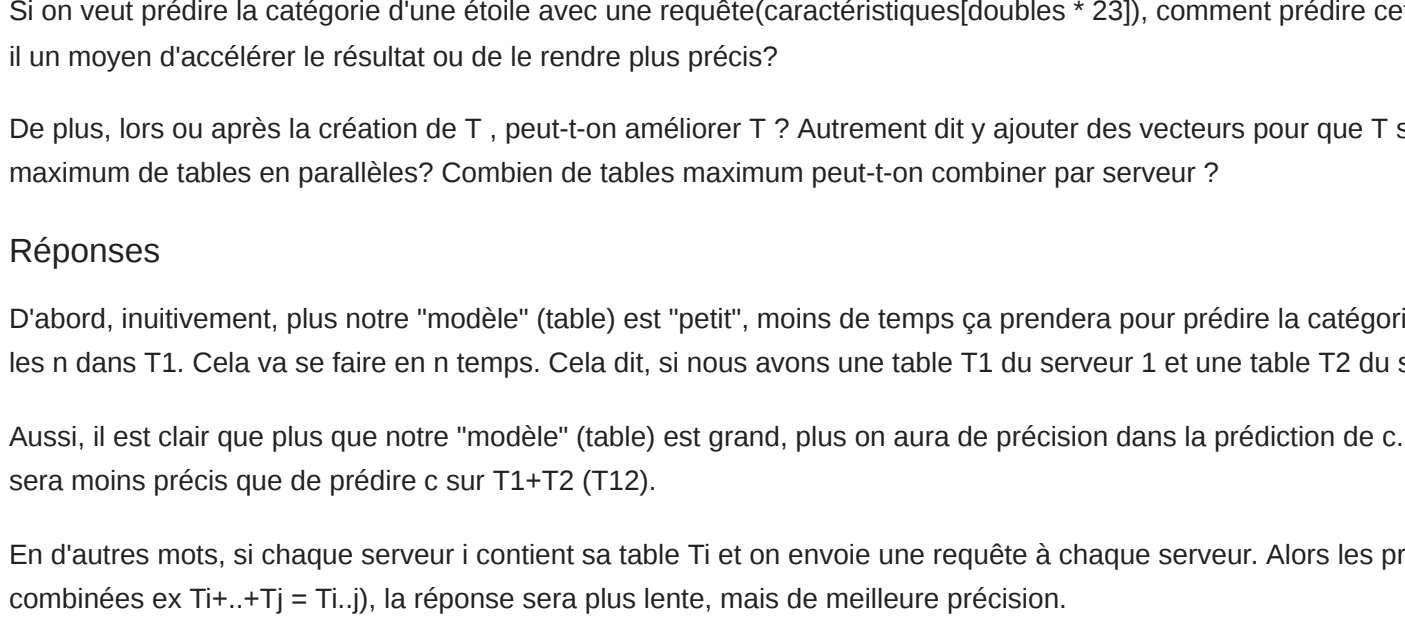
Ordon, à quoi ressemblent nos tables ?

Ordon, à quoi ressemblent nos tables ? Ordon, 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 Minst, les chiffres c'est les catégories et les vecteurs pixels sont les vecteurs caractéristiques).



Ordon, il est clair que chaque serveur peut lire la table sur le disque en  $O(n)$  à 3Go/s. Pour ainsi produire la table T.

De plus, il est clair que si ensuite un serveur reçoit une requête 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 à tous les vecteurs dont on connaît la catégorie dans T et on



prédit une catégorie selon les k plus proches voisins.

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

Questionsnements 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 sélectionner la catégorie la plus souvent prédite? Y'a-t-il un moyen d'accélérer le résultat de ce dernier plus précis?

De plus, lors ou après la création de T, peut-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-on combiner par serveur ?

Réponses

Ordon, malheureusement, plus "modèle" (table) est "petit", moins de temps ça prendra pour prédire la catégorie d'un vecteur caractéristique c. En effet, si nous avons un serveur A avec une table T1 avec n d'étoiles connues, alors pour prédire c, nous devons le comparer à tous les n dans T1. Cela va se faire en n temps. Cela dit, 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 prendra 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 produit une 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).

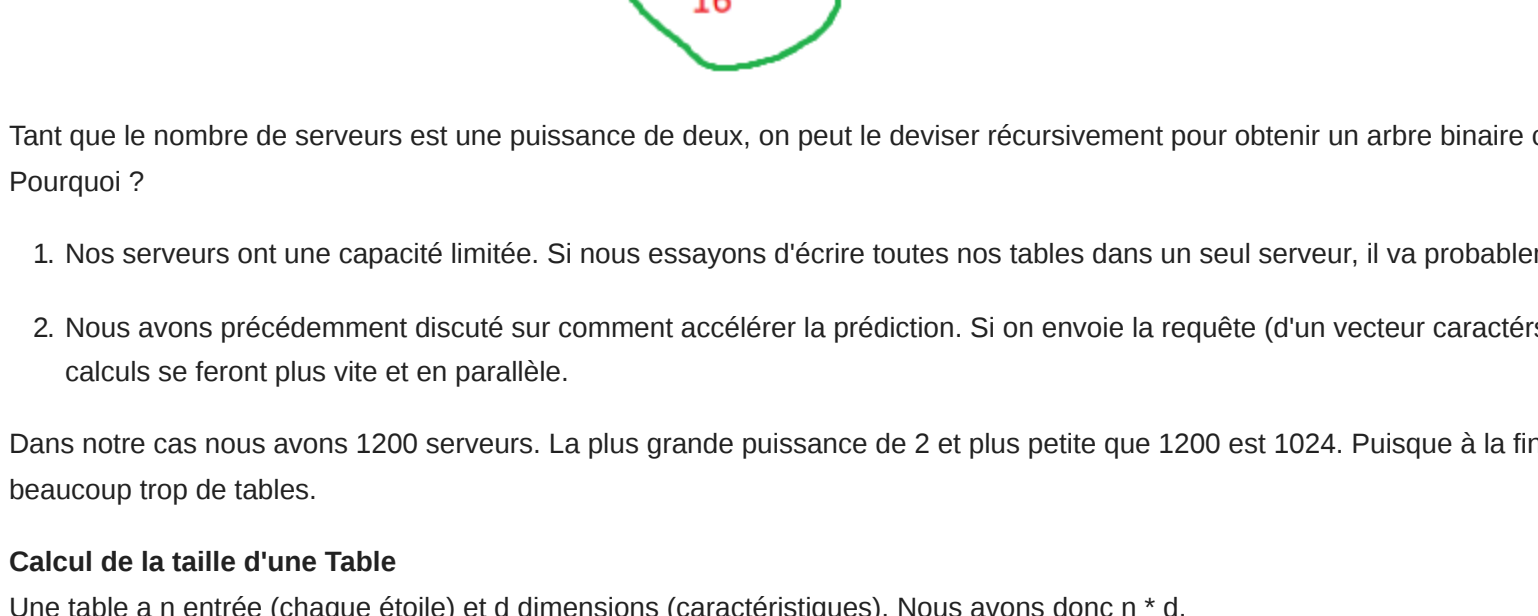
En d'autres mots, si chaque serveur i contient sa table Ti et on envoie une requête à chaque serveur. Alors les prédictions se feront toutes plus rapidement, mais avec moins de précision. Cependant, si on envoie une requête à un seul serveur avec une grande table (plusieurs tables créées donc des arbres binaires de hauteur 2 fois 200),

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

Super Tables

Donc, si on divise nos 1200 serveurs en 100 super tables, nous pouvons créer 100 super tables. Pour cela, nous allons créer un serveur i produit une table Ti (dans l'algorithme knn). On se sert alors de cette table T\_j (lorsqu'on dans l'algorithme knn) pour prédire la catégorie d'une étoile en la comparant à toutes les étoiles dans T\_j pour trouver ses k plus proches voisins. Si nous avons m serveurs, et chaque serveur i produit une table T\_i, 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 super tables? Nous allons utiliser des arbres binaires complets. 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:



Pour que le nombre de serveurs est une puissance de deux, on peut le devoir récursivement pour obtenir un arbre binaire complet de hauteur log\_2(width). Bien sûr, encore une fois, nous ne voulons pas une seule super table (un seul arbre binaire), mais plusieurs super tables. Tant que ?

1. Nous pouvons limiter notre capacité limitée. Si nous essayons d'ajouter toutes les données dans un seul serveur, il va probablement saturer.

2. Nous avons précédemment discuté sur comment accélérer la prédiction. Si on envoie la requête (un vecteur caractéristique c) pour prédire une catégorie à une grosse table, cela va prendre plus de temps. Tandis que si nous pouvons 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.

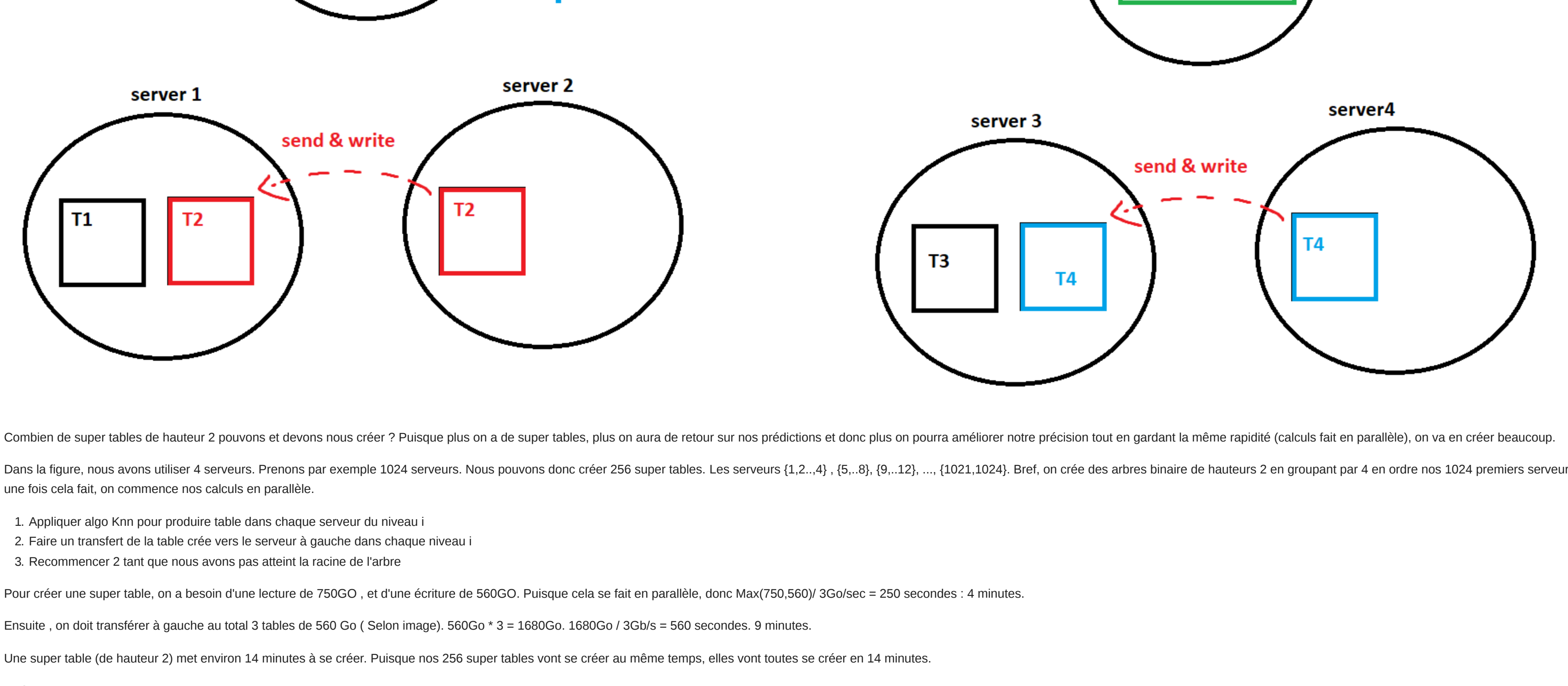
Calcul de la taille d'une Table

Une table a n-entree (chaque étoile) et d dimensions (caractéristiques). Nous avons donc n \* d.

Nous savons que nous avons au total 7300 milliards d'étoiles réparties de manière équilibrée dans 1200 serveurs. 7300 Milliards / 1200 serveurs = 6,08 milliards (m) d'étoiles par serveur. ( 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 convertit nos données nous aurons environ 560 Go.

Chaque serveur a une capacité de 4 Tera (soit 4000Go). Et 750Go sont déjà occupés (par les étoiles). Donc, nous avons en réalité 3250 Go de libre. Combien de fois pouvons nous mettre 560 dans 3250? 3250 / 560 = 4,5. Donc, la puissance de deux la plus proche est 4. Nous pouvons créer donc des arbres binaires de hauteur 2 fois 200.

Bien sûr 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 binaires 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 binaires de hauteur 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 allons utiliser 4 serveurs. Devons nous créer 1024 serveurs. Nous pouvons donc créer 256 super tables (la combinaison de toutes tables) sera écrite sur un seul serveur. Il est donc important de ne pas combiner beaucoup trop de tables.

1. Appliquer algo knn pour produire table dans chaque serveur du niveau i

2. Faire un tableau 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 ( si nous appelons 560Go / 3Go/s = 160Go. 160Go / 3Go/s = 50 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 ça devient intéressant. Admettons 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!