Université de Technologie de Compiègne

NF26: Datawarehouse et outils décisionnels

Haute volumétrie: stockage et apprentissage

Polycopié de cours

Auteurs :

- Andres Maldonado
- Emmanuelle Lejeail
- Florent Chehab
- Gaëtan Blond
- Guillaume Jorandon
- Guillaume Jounel
- Jean-Benoist Leger
- Julien Jerphanion
- Léna Schofield
- Mathis Chenuet
- Maxime Lucas
- Thomas Le Vaou

Printemps 2020

Trouver une plus belle page de garde

Table des matières

Todo list

Chapitre 1

Introduction

Le stockage de données en haute volumétrie et l'analyse d'icelles [1] se feront généralement au travers de modèles de données sortant du cadre relationnel, et de modèles de traitement sortant du cadre classique.

L'objectif de cette introduction sera de présenter les principales propriétés du stockage relationnel, et les limitations induites par ces propriétés. Puis de présenter de manière formelle les principales propriétés du modèle relationnel et des propriétés analogues en sortant de ce cadre.

1.1 Base de données classique

Cette section sera traitée au travers d'un exemple qui illustrera les différents aspects. Cette introduction n'a pas pour objectif de remplacer un cours de base de données relationnelle, elle présuppose ces notions acquises.

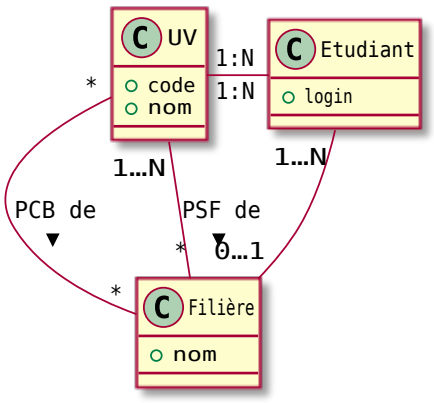

FIGURE 1.1 – Exemple de modèle conceptuel de données

Modèle conceptuel de données : Ce modèle conceptuel, caractéristique de ce qui se rencontre dans les bases de données relationnelles, contient beaucoup d'informations dont l'objectif sera de retranscrire ces informations au sein du stockage de données.

Clarifier la phrase ci-dessous

1. de celles-ci

Modèle logique de données : L'objectif du modèle logique de données est de retranscrire au mieux toutes les contraintes exprimées dans le modèle conceptuel, de telle manière à assurer la cohérence des données.

Une des manières d'atteindre cet objectif est d'utiliser la normalisation, afin de transformer un maximum de dépendances fonctionnelles, et contraintes de type clef, contrainte que gère nativement un moteur de base de données relationnelle. La réduction de la redondance des données n'est qu'un bonus, et n'est pas ce qui est recherché en premier dans le processus de normalisation de tout concepteur de base de données conscient de l'objet qu'il manipule. Il s'agira de comprendre ce que nous allons perdre en passant à de gros volumes de données.

Voici la transcription sous forme de modèle logique de données du modèle conceptuel de la figure 1.1 :

```
UV(#code, nom, filière → Filière.nom)
Étudiant(#login)
Filière(#nom)
UVÉtudiant(#uv → UV.code, #etu → Étudiant.login)
UVFilière(#fil → Filière.nom, #uv → UV.code, PCBouPSF bool)
```

Toutefois ce modèle logique ne retranscrit pas toutes les contraintes telles que développées dans le modèle conceptuel. En effet, les contraintes 1_N n'ont pas été retranscrites, uniquement des contraintes 0_N l'ont été. Pour cela, une manière de faire est d'utiliser des déclencheurs, l'implémentation est lourde.

Exemple de déclencheur (postgres) : Cette procédure est souvent utilisée pour s'assurer de l'intégrité des données dans le cas de contraintes que l'on ne peut pas directement implémenter dans le modèle logique de données.

Par exemple, pour s'assurer que toutes les UVs ont au moins un étudiant tel que décrit dans le modèle conceptuel figure 1.1, nous pouvons écrire la contrainte ensembliste :

```
Projection(UV, code) ⊆ Projection(UVÉtudiant, uv)
```

Remarquons au passage que vérifier l'égalité n'est pas nécessaire, l'inclusion dans l'autre sens résulte de la contrainte de clef étrangère, nous n'avons que ce sens à vérifier. Une manière de faire est d'implémenter tout d'abord une requête ne retournant rien si la contrainte est satisfaite :

```
-- Sélection des UVs sans étudiants présents
SELECT code FROM UV WHERE code NOT IN (SELECT uv from UVEtudiant)
```

On va maintenant créer un trigger pour s'assurer de la contrainte d'intégrité `1...N` :

```
CREATE FUNCTION funtrig_cl()
    RETURNS trigger AS
$func$
BEGIN
    IF EXISTS
        (SELECT code FROM UV
          WHERE code NOT IN
              (SELECT uv from UVEtudiant))
    THEN RAISE EXCEPTION 'Message disant que tout va mal';
    END IF;
RETURN NEW;
```

Puis on va dire au moteur de base de données de vérifier cette contrainte lors de l'insertion dans `UV` ou lors de la suppression dans `UVÉtudiant`, et seulement à la fin de la transaction :

```
CREATE CONSTRAINT TRIGGER trig_cl_UV
AFTER INSERT OR UPDATE
ON UV
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW EXECUTE PROCEDURE funtrig_cl();

CREATE CONSTRAINT TRIGGER trig_cl_UVEtudiant
AFTER DELETE
ON UVEtudiant
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW EXECUTE PROCEDURE funtrig_cl();
```

Utilisation : Imaginons que l'on rajoute une UV non présente dans la base :

```
INSERT INTO UV(code) VALUES ('NF26');
```

Une exception sera levée car la contrainte sera violée. Également réaliser l'insertion dans l'autre sens ne fonctionne pas non plus, puisque la contrainte de clef étrangère est violée :

```
INSERT INTO UVEtudiant (UV,etu) VALUES ('NF26', 'totopassoir');
```

Il est imposé que l'ensemble des requêtes d'une transaction respecte la contrainte, et ainsi les insertions peuvent être réalisées :

```
BEGIN TRANSACTION
    INSERT INTO UV(code) VALUES ('NF26');
    INSERT INTO UVEtudiant(UV, etu) VALUES ('NF26',
    ↪ 'totopassoir');
COMMIT
```

L'ensemble des contraintes exprimées dans le modèle conceptuel peuvent être implémentées, plus ou moins facilement, dans la base de données, et ainsi assurer un stockage toujours cohérent des données.

On peut néanmoins arriver dans des cas où l'on ne peut passer à l'échelle, à cause :
- Du volume des transactions ;
- Du volume des données ;
- Du volume des données annexes utilisés pour gérer les contraintes (indexes).

1.2 Bilan du relationnel

Nous pouvons réaliser le bilan suivant du relationnel :

Relationnel — Avantages :
- Intégrité des données ;
- Possibilité de faire des requêtes très complexes avec nombre de jointures en exploitant les relations ;
- Possibilité de faire des requêtes complexes, avec du calcul ;
- Standardisé (structure, requêtes...) ;
- Grande communauté d'utilisateurs et d'utilisatrices (⇒ logiciels aboutis).

Relationnel — Inconvénients :
- Gérer un tas d'indexes, ce qui induit :
- Volumes peu élevés (même en distribuant) ;
- Débit en lecture et en écriture faible (généralement le débit en lecture pose moins de problèmes, une distribution peut être mise en place, mais l'écriture est plus limitant à cause des contraintes d'intégrité et de la génération des indexes) ;
- Point unique $\Longrightarrow$ existence d'un SPOF (*Single Point Of Failure*) - un problème sur un seul point (comme un nœud coordinateur, par exemple) peut remettre en cause l'intégralité de la base de données. De plus cela impose une scalabilité verticale et non horizontale.

Dans le contexte du *Big data*, un gros volume de données est considéré et généralement incompatible avec ces inconvénients. Le seul moyen pour amoindrir ces problèmes tout en restant en relationnel est de faire du *passage à l'échelle verticale* (c'est-à-dire utiliser des serveurs plus gros...).

Dans le cadre non relationnel, on va oublier l'utilisation du relationnel mais ne pas oublier ses propriétés.

1.3 Formalisation

1.3.1 Propriétés ACID

Ces propriétés sont respectées par les moteurs de base de données relationnelle, mais c'est ce qui va nous poser problème.

- **A** : ***Atomicité***. Considérer les requêtes comme atomiques : l'intégralité d'une transaction est soit acceptée soit rejetée ; un ensemble de transactions sera accepté ou rejeté en bloc.
- **C** : ***Cohérence*** : intégrité des données insérées avec les schémas et autres données déjà présentes.
- **I** : ***Isolation***. Exécution de transactions en parallèle de façon équivalente à une exécution en série.
- **D** : ***Durabilité***. Une fois que les données sont acceptées, elles vont rester dans la base de données... *ad vitam æternam* (tant que l'utilisateur n'en décide pas autrement).

1.3.2 Théorème du CAP ou plutôt Conjecture de Brewer

Certains l'appellent *Théorème de Brewer* mais celui-ci ne l'a jamais démontré, on parlera donc plutôt de *Conjecture de Brewer*.

Théorème 1.1 (Théorème du CAP)**.** *L'acronyme « CAP » correspond à :*
- ***Consistency : Cohérence****. Si lecture, on obtient l'enregistrement le plus récent ou une erreur : pas question de lire une donnée absente ou périmée.*
- ***Availability : Disponibilité****. Chaque lecture retourne un résultat, le plus récent ou non.*

— ***Partition tolerance : Tolérance au partitionnement.*** *Le système continue d'être opérationnel même si une de ses sous-parties est absente.*

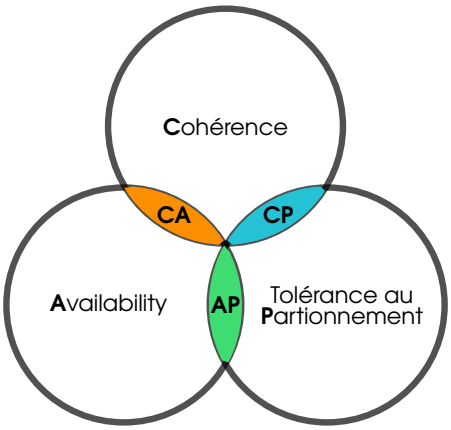

Figure 1.2 – Illustration du théorème du CAP

Théorème/Conjecture : Le respect de ces trois propriétés simultanément est impossible : on ne peut pas construire un système qui respecte ces trois propriétés en même temps.

En relationnel, on est entre C et A (*i.e.* le relationnel est tout sauf tolérant au partitionnement) là ou en non relationnel, on est soit entre C et P, soit entre A et P.

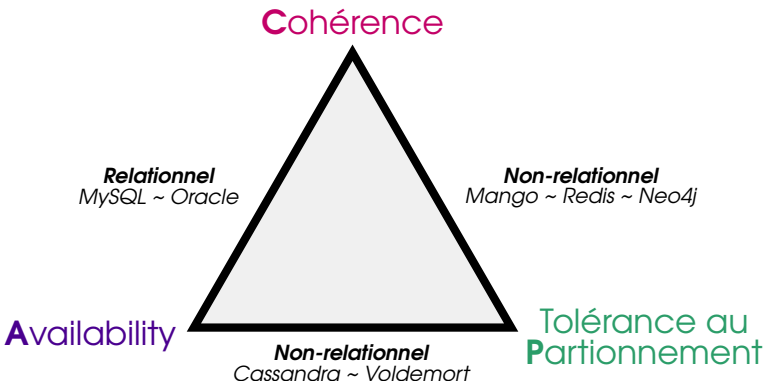

Figure 1.3 – Technologies et théorème du CAP

Le théorème a finalement été démontré par Gilbert et Lynch en 2002, *par un formalisme assez pédestre* (voir [4] pour une explication plus simple. Pour plus détails, on peut se référer au papier original [1]).

Preuve du théorème CAP :
On suppose l'existence d'un système respectant CAP, avec N_1 et N_2 isolés.

1. C écrit v_1 sur N_1

2. N_1 répond ACK car le système respectant P, il doit fonctionner même isolé.
3. C lit N_2, on a alors 3 possibilités :
   — ne répond pas $\implies$ viole P
   — répond une erreur $\implies$ viole A
   — répond v_0 $\implies$ viole C
   Ce qui constitue une contradiction, et conclue le raisonnement par l'absurde.

1.3.3 Propriétés BASE

Les moteurs de bases de données non relationnelles ne pourront pas respecter les propriétés ACID, ainsi, d'autres propriétés ont été introduites, il s'agit des propriétés BASE.

— ***Basic Availability*** : Disponibilité la plupart du temps. On sait prévoir quand la base de données sera à nouveau disponible.
— ***Soft state*** : Pas de cohérence globale au niveau des écritures, mais localement oui.
— ***Eventual consistency*** : La cohérence sera atteinte à un moment donné. À partir d'un temps fini, la dernière donnée est disponible.

1.4 Notion de passage à l'échelle

Pour améliorer les performances d'une base de données :
— En relationnel, on parle de *scalabilité* (mise à l'échelle) *verticale* en augmentant les performances de la machine hébergeant la BDD.
— En non-relationnel, on parle de *scalabilité horizontale* en distribuant les données sur plusieurs machines.

Il existe des systèmes hybrides (comme MySQL) qui proposent à l'utilisateur du relationnel mais qui délèguent une partie du stockage à une BDD non-relationnelle. Le passage à l'échelle n'est cependant pas aussi efficace que du non-relationnel pur.

Chapitre 2

Stockage clef/valeur

Il s'agit de la forme la plus simple de stockage, qui est utilisée dans de nombreux contextes : soit directement, soit indirectement par d'autres moteurs de stockage. La majorité des concepts introduits ici ont une portée plus générale dans les systèmes de base de données.

2.1 Introduction

La structure de données clef/valeur est disponible dans de nombreux langages même si le nom est susceptible de varier selon la technologie.

- `Python` : dictionnaires
- `Perl` : tables de hashage
- `C++` : `map, unordered_map` (STL)
- `LISP` : `car` et `cdr` des listes
- `Scala` : `Map`
- `JS` : `object, Map`

Définition : SK super-clef : $(k_i) \in \mathbb{K}, \quad \forall i,j \quad k_i = k_j \iff i = j$

Définition : K clef
Soit $\mathbb{K}$ un sous-espace de SK. $\mathbb{K}$ est clef ssi $\forall \mathbb{K}_1, \mathbb{K}_2$ tq $\mathbb{K}_1 \times \mathbb{K}_2 = \mathbb{K}, \quad \mathbb{K}_1$ et $\mathbb{K}_2$ ne sont pas SK. Autrement dit une clef est une super-clef minimale.

Définition : La structure clef–valeur est un ensemble de couples $\{(k_i, v_i)\}_i \subset \mathbb{K} \times \mathbb{V}$ où i est le numéro d'enregistrement, $\mathbb{K}$ un espace de super-clefs, $\mathbb{V}$ un espace.

Le plus souvent, les clefs sont uniques : $\forall i,j, \quad i \neq j \Rightarrow k_i \neq k_j$
Si ce n'est pas le cas, on peut associer à une clef le n-upplet de valeurs correspondantes. Donc faire l'hypothèse d'unicité des clefs est **WLOG** (sans perte de généralité – *Without Loss Of Generality*).

2.2 Exemple

Cet exemple se base sur l'exemple du chapitre précédent. Une manière de mettre en forme les données pour les stocker sous forme d'associations clefs/valeurs est la suivante :

```
UV(#code, (nom, filière))
Étudiant(#login, ∅)
UVÉtudiant(#(UV, etu), ∅)
```

2.3 Opérations

Avec un système clef/valeur on peut définir les 4 opérations **CRUD** (qui permettent de *tout* faire – y compris du *k-means*) : **Create**, **Read**, **Update** et **Delete**.

2.4 Technologies

virer les liens et les mettre dans le bib

Des exemples de technologies sont :

— Voldemort (utilisé par exemple par Twitter)
— Redis (utilisé par Mastodon)
— BDB (Berkley DB) et LMDB (même API), dans la même idée que `sqlite`, pas besoin de serveur

Bilan :

— **Pour :** c'est simple (et rapide et puissant)
— **Contre :** c'est simple

2.5 Distribution des données

La distribution des données au sein des stockages de types clefs/valeurs est un point crucial de nombre de systèmes distribués, et les concepts introduits ici ont une portée plus générale, dans d'autres types de modèles de données.

2.5.1 Calcul de condensat (*hashage*)

Introduction

Technique souvent utilisée : utilisation d'un condensat (*hash*) parce que les clés ne sont pas toujours comparables (par exemple faisant partie de différents ensembles). Nous allons considérer un *hash* comme étant une fonction qui respecte $\forall i,\ h(k_i) \in [\![0, N[\![$

Dans l'idéal, on voudrait que *la fonction de hashage soit injective*, *i.e.* :

$$\forall i, j \quad k_i \neq k_j \Rightarrow h(k_i) \neq h(k_j)$$

Dans les faits, c'est impossible (quand ça arrive, on parle de *collision*) : on a plus de N éléments k_i à hasher, donc nécessairement certains auront le même hash dans $[\![0, N[\![$. [1]

C'est le cas de la fonction de hachage MD5, mais aussi de SHA-1, utilisée dans `git`. Dans certains cas, cela ne remet pas en question leur utilisation — de ce point de vue, SHA-1 continue d'être utilisée pour `git` car les cas de collisions sont extrêmement rares et difficilement constructibles.

Dans un contexte de distribution de données, on demande juste que $\{h(k_i)\}_i$ soit à peu près uniforme.

1. Voir le principe des tiroirs pour plus de détails.

Définition : hash cryptographique
- Résistance aux collisions : il est très difficile de trouver (x, y) tq $h(x) = h(y)$
- Résistance à la $2^{\text{ème}}$ préimage : si on a x, il est très difficile de trouver y tq $h(x) = h(y)$
- Résistance à la $1^{\text{ère}}$ préimage : si on a l, il est très difficile de trouver x tq $h(x) = l$

Exemples de fonctions de condensat

- condensat non cryptographique : `CRC` (somme de redondance cyclique), `md5`
- condensat cryptographique : `sha2`, `sha3`

2.5.2 Utilisation des condensats

Cas d'usage : on a une clef k, on cherche la donnée correspondante, si elle existe. On cherche donc dans un premier temps $h(k)$. On obtient $A = \{(k_i, v_i) \mid h(k_i) = h(k)\}_i$ qui est un ensemble « *petit* ». Puis il nous reste à sélectionner dans cet ensemble les données qui nous intéressent : $\{(k_i, v_i) \in A \mid k_i = k\}$. Si le condensat est bien choisi, si la donnée existe, A est la majorité du temps un singleton ne contenant que le couple qui nous intéresse.

Exemple : On a l'ensemble de données suivant : $\{('a', \cdots), ('b', \cdots), ('c', \cdots), ('f', \cdots)\}$, avec :

$$h(a) = 113$$
$$h(b) = 8$$
$$h(c) = 127$$
$$h(f) = 113$$

On veut stocker cet ensemble de données par h trié, on obtient :

h	k	V
8	'b'	$\cdots$
113	'f'	$\cdots$
113	'a'	$\cdots$
127	'c'	$\cdots$

Question : quelle valeur il y a derrière f ? On commence par calculer $h(f) = 113$, puis on obtient $A = \{(f, \cdots), (a, \cdots)\}$. Trouver les h ne revient pas à trouver les clés. Les opérations qui sont linéaires dans la BDD peuvent être faites mais avec parcimonie. Ici, pour trouver un élément, on restreint la recherche en un temps logarithmique (pour trouver A), et après on a un temps linéaire en la taille de A (qu'on espère petite).

2.5.3 Sharding

Sharding : Partitionnement horizontal des données (*i.e.* couples {clé, valeur}). Chaque machine contient une partie des données.

Intérêts :

— Volume de stockage disponible (on peut mettre des données à plusieurs endroits) ;
— On peut répliquer les données (*point essentiel*)
— On va pouvoir répartir la charge (*point fondamental*, dans l'idéal on veut une répartition uniforme de la charge)

Remarque : il existe aussi du partitionnement vertical où une partie de la valeur est stockée sur certains serveurs et ainsi de suite.

Illustration : La division entière Soient un espace des clefs $[\![0, N[\![$ (ou espace des condensats des clefs), p nœud, n objets de clefs k_i et de valeur v_i. Ces couples seront stockées dans le nœud $(k_i \bmod p) \in [\![0, p[\![$.

Exemple : Avec un espace des clefs $[\![0, 100[\![$, on a $p = 3$ serveurs :
Soit les données $\{(27, \cdots), (31, \cdots), (71, \cdots), (18, \cdots)\}$ Chaque valeur k va aller dans le nœud $k \bmod p$.

Ainsi, la distribution des données va être effectuée comme suit :
— 1er serveur, nœud 0 : $\{(27, \cdots), (18, \cdots)\}$
— 2e serveur, nœud 1 : $\{(31, \cdots)\}$
— 3e serveur, nœud 2 : $\{(71, \cdots)\}$

Dans ce cas, la répartition est uniforme uniquement en espérance.

Récupération de données : soit une clef k. Si elle existe, elle est dans le nœud $k \bmod p$.

Un point essentiel est que l'on n'a pas besoin de table stockant la localisation des données. L'existence d'une telle table constituerait un *SPOF*.

Exemple d'implémentation :

```
1  class SimpleShardStore:
2      def __init__(self, nb_nodes):
3          self.nb_nodes = nb_nodes
4          self.nodes = [{} for _ in range(nb_nodes)]
5
6      def create(self, k, v):
7          self.nodes[hash(k) % self.nb_nodes][k] = v
8
9      def read(self, k):
10         return self.nodes[hash(k) % self.nb_nodes][k]
11
12     def update(self, k, v):
13         self.create(k, v)
14
15     def delete(self, k):
16         del self.nodes[hash(k) % self.nb_nodes][k]
```

2.5.4 *Consistent hashing*

C'est une manière avancée de distribuer les données en utilisant leur condensat, de telle manière à être évolutif, et à gérer de nombreuses situations.

Introduction et définition

Principe :
Considérons $\{(k_i, v_i)\}$ données à stocker. On définit la **fonction de condensat** :

$$h : k \to [\![0, N[\![$$

avec p nœuds de stockage. Chaque nœud q stocke un intervalle de l'espace des condensats de la forme $[\![n_q, m_q[\![$ (cf Figure 2.1).

On peut choisir des intervalles qui se recouvrent.

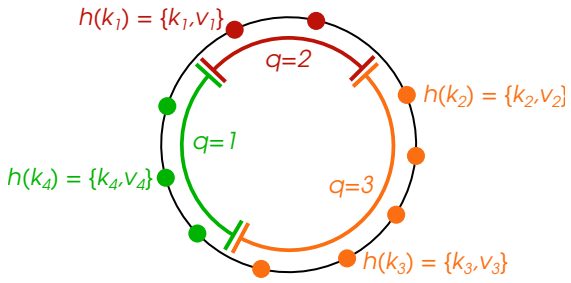

Figure 2.1 – Illustration du Consistent-hashing

Propriétés

Il faut s'assurer que toutes les données soient stockées, c'est le cas si et seulement si :

$$\bigcup_q [\![n_q, m_q[\![= [\![0, N[\![$$

On peut définir le nombre de réplications pour chaque position : pour la position $w \in [\![0, N[\![$, combien a-t-on de nœuds contenant w, *i.e.* de q tel que $w \in [\![n_q, m_q[\![$?
On note ceci $R(w) = \sum_q \mathbb{1}_{w \in [\![n_q, m_q[\![}$. On aura $R(w) \geq 2$ dans les zones de chevauchement/superposition entre deux nœuds, 0 dans les zones sans nœud, et 1 dans les zones où il n'y a qu'un seul nœud.

Le nombre de réplications à l'échelle de toutes les données est donné par

$$R = \min_{w \in [\![0,N[\![} R(w) = \min_{w \in [\![0,N[\![} \sum_{q \in [\![0,p[\![} \mathbb{1}_{[\![n_q, m_q[\![}(w)$$

Remarque : $\bigcup_q [\![n_q, m_q]\!] = [\![0, N]\!] \quad \Leftrightarrow \quad R \geq 1$
On note que cette manière de stocker permet la re-répartition des données.

Exemple d'implémentation

```python
import random

class BetterShardStore:
    def __init__(self, N):
        self.nodes = [
            [0, 10, {}],
            [5, 15, {}],
```

```
8               [12, N, {}],
9           ]
10          self.N = N
11
12      def create(self, k, v):
13          h = hash(k) % self.N
14          for min, max, store in self.nodes:
15              if min <= h < max:
16                  store[k] = v
17
18      def read(self, k):
19          h = hash(k) % self.N
20          return random.choice([store for (min, max, store) in self.nodes
            ↪  if min <= h < max])[k]
21
22      def update(self, k, v):
23          self.create(k, v)
24
25      def delete(self, k):
26          h = hash(k) % self.N
27          for min, max, store in self.nodes:
28              if min <= h < max:
29                  del store[k]
```

Illustration

blabla adapté à la figure

Re-répartition des données

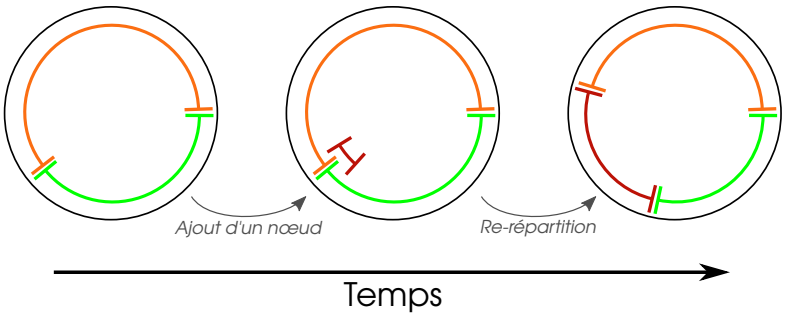

Figure 2.2 – Illustration de l'extension-sharding

Au départ, on a $R(w) = 1$ sur toute la base de données. Au fur et à mesure du temps, il récupère des données : certains nœuds *abandonnent* des données au profit du nouveau nœud.

blabla adapté à la figure

Mort d'un nœud

On considère le cas où $R = 2$ (s'il vaut 1, on se retrouve dans une situation où la donnée n'est liée qu'à un seul nœud, donc la mort de ce dernier pose problème). Exemple au tableau de base où les $R(w)$ valent 2, sauf dans une zone où il vaut 3 (cf Figure 2.3). Si le nœud bleu (à cheval entre les zones où $R(w)$ vaut 2 et 3) meurt, on perd le niveau de réplication $R = 2$ (vu qu'on se retrouve avec $R = 1$). Dans cette situation, on peut faire grossir les nœuds restants pour combler les « trous » laissés où $R = 1$ jusqu'à

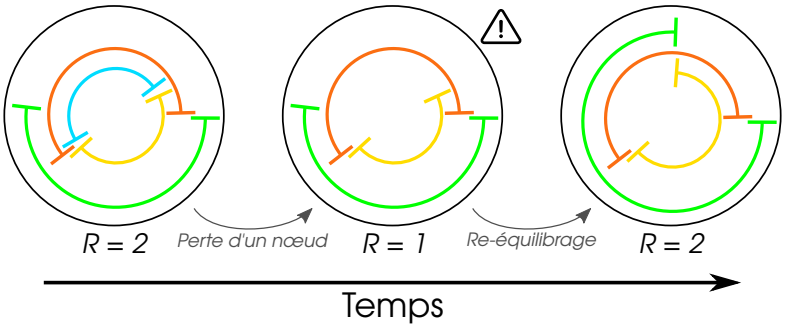

Figure 2.3 – Illustration du reduction-sharding

réobtenir $R = 2$. Cela se traduit par le transfert des données contenues dans l'intervalle où $R(w) = 1$ aux nœuds adjacents.

2.5.5 Intérêt

Absence de table de distribution : Pour retrouver une donnée, on a seulement besoin de connaître les bornes des différents intervalles (n_q, m_q). Pas besoin de table de stockage de distribution des données.

Hantise de la panne simultanée : Lors de la mort d'un disque, ses données (déjà dupliquées sur d'autres) doivent être copiées vers d'autres nœuds pour assurer à nouveau une duplication. Cela nécessite beaucoup d'accès en lecture et en écriture, ce qui peut augmenter les chances de pannes sur les autres disques.

2.5.6 Topologie et stockage (pédagogie par l'exemple)

Contexte de l'exemple : On veut pouvoir interroger le cluster depuis n'importe où. Il faut ici une stratégie de réplication qui respecte la topologie.

Exemple : des données sont présentes sur trois sites dans le monde (USA, France, Océanie) mais correspondent à la même base de données. On doit avoir $R \geq 3$ (pour que les données soient accessibles sur les trois lieux en même temps).

On veut une *non colocalisation des réplications* : Les serveurs sont sur des sites différents pour éviter les dégâts liés à des évènements exceptionnels, et pour qu'une lecture soit aussi proche que possible de n'importe quel endroit sur Terre.

insérer figure, et blabla adapté

On a $R_{global} = 3$ (superposer les trois zones pour s'en rendre compte).

Si le nœud bleu de Z_1 meurt, il faut recréer les données perdues dans la Z_1 depuis les données de Z_2 ou Z_3, il va donc falloir faire un transfert.

Lien avec le CAP theorem : tolérance au partitionnement ici.

Cassandra utilise le consistent hashing par exemple mais de nombreux autres systèmes utilisent également ce principe. Par exemple Spark permet de définir comment répartir les données sur les nœuds selon plusieurs stratégies via l'API des RDD. [3]

Chapitre 3

Bases de données orientées colonnes

3.1 Introduction

Il s'agit d'un concept relativement récent (moins de 20 ans).

Idées générales :

— Les données sont stockées par colonnes (et non par lignes comme c'est le cas pour les SGBD traditionnels). Au lieu de stocker des lignes, on stocke des couples par colonne.

Avantage :

— Cela permet de manipuler plus facilement des données incomplètes (dans un système relationnel, une valeur nulle dans une colonne relève plus de l'exception que de la règle, alors que c'est l'inverse dans une base de données orientée colonne) ;

— Le nombre de colonnes peut être potentiellement très grand ;

— La création de colonnes est gratuite en comparaison à création de ligne.

« *Les intuitions peuvent être incorrectes.* »

Exemples de technologies utilisant les bases de données orientées colonnes :

— Cassandra ;

— Big Table (de Google) ;

— HBase d'Amazon.

Dans ce cours nous allons utiliser Cassandra. Certains aspects du cours seront orientés en fonction, on utilisera des éléments propres à cette technologie comme *CQL*, le *Cassandra Query Language*.

3.2 Modèles de données

3.2.1 Introduction et définition

Si ce modèle est relativement proche du modèle relationnel, nous allons voir qu'il s'en écarte aussi dans ces mesures importantes.

mesure ?

Enregistrement : Dans ce modèle, on associe à chaque clé primaire une ou plusieurs colonnes. Ces colonnes ne sont pas forcément les mêmes à chaque fois. Ainsi, plusieurs enregistrements peuvent ne pas avoir certaines colonnes renseignées et il est possible d'avoir des enregistrements avec des colonnes qui n'existent nulle part ailleurs.

Pour chaque donnée, il y a un numéro de révision, qui est le plus souvent un timestamp. Ce modèle est illustré figure 3.1.

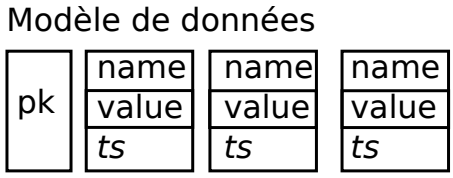

Figure 3.1 – Modèle d'une base de données non-relationnelle « orientée colonne »

illustration

Famille de colonnes ou table : Les données sont regroupées au sein d'une *table* ou *famille de colonnes*. Mais malgré la ressemblance de ce regroupement avec un modèle relationnel, les colonnes peuvent être ici toutes différentes. Même s'il se peut qu'on appelle ça une "table", ces regroupements restent différents des tables au sens relationnel du terme.

illustration

Espace de clefs : un ensemble de tables. Cet espace de clefs va gérer la réplication et le partitionnement des données. Ces propriétés peuvent être définies selon une topologie à ce niveau.

3.2.2 Exemple (avec des UV)

— Pour "NF26", certaines colonnes sont présentes ("PSF_FDD"), mais pas toutes ("PCB_GB" n'est pas présente) ;
— Pour "SY02", de même, les colonnes sont différentes de "NF26" (colonne "PCB_GI" inexistante sur "NF26").

On peut donc être amené à avoir une multitude de colonnes et donc une très forte dénormalisation. Il est fréquent que des colonnes ne soient définies que pour quelques entrées seulement. Généralement, avec de tels SGBD non relationnels, les jointures sont impossibles. Les colonnes peuvent même être toutes différentes entre les enregistrements.

3.3 Équivalence avec les magasins (k/v)

Chaque enregistrement peut avoir un nombre de valeurs (de colonnes) différent. Chaque enregistrement peut être représenté par un magasin clef/valeur $\{(c_{ik}, v_{ik})\}_k$.

— c_{ik} : nom de la k-ième colonne de l'enregistrement i.
— v_{ik} : valeur de la k-ième colonne de l'enregistrement i.

La clef d'enregistrement demeure la même (si on reprend l'exemple précédent, il s'agit de `SY02` et `NF26`) c'est-à-dire la *clef primaire*, tandis que la valeur d'enregistrement est l'ensemble des clefs/valeurs.

On représente aussi une famille de colonnes comme un *magasin* $\{k_i, \{(c_{ik}, v_{ik})\}_k\}_i$, où l'ensemble des clés-valeurs est vu comme une valeur à part entière.

Dans l'exemple :

```
1  {
2      "NF26" : {"PSF_FDD": True, "PCB_F": "..."},
3      "SY02" : {"PCB_GI": True, "PCB_GB" : True}
4  }
```

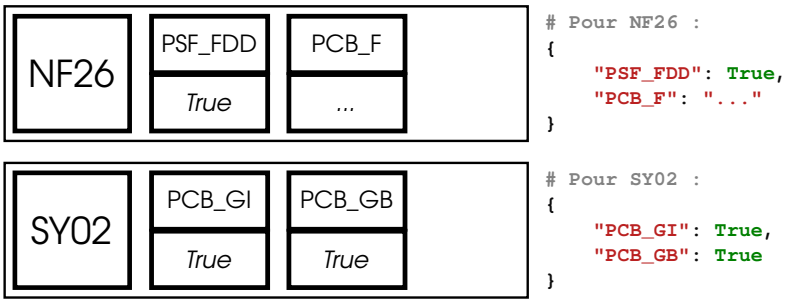

Figure 3.2 – Clefs dans le cas de l'exemple des UVs

Évidemment, chaque c_{ik} doit correspondre à une colonne existante.

3.4 Clefs de distribution et clefs de partitionnement

changer la notation avec des exposants entre parenthèses pour séparer les deux clefs, plutôt que les indices, ça prête à confusion

Utiliser du hachage,... est dangereux sur des petites quantités d'enregistrements.

3.4.1 Distinction entre clefs de partitionnement et clefs de tri

Une clef valuée sur $\mathbb{K}_1 \times \mathbb{K}_2 = \mathbb{K}$

$$\forall k \in \mathbb{K}, \quad \exists (k_1, k_2) \in \mathbb{K}_1 \times \mathbb{K}_2 \quad | \quad k = (k_1, k_2)$$

3.4.2 Dans le cadre du k/v :

On a les données : $\{(k_i, v_i)\}_i, k_i \in \mathbb{K}_1 \times \mathbb{K}_2$, on note $k_{1i}, k_{2i} \mid k_i = (k_{1i}, k_{2i})$.
Soit $\widetilde{k}_1 \in \mathbb{K}_1$, soit le *magasin* $\mathcal{M}(\widetilde{k}_1) \hat{=} \left\{((k_{1i}, k_{2i}), v_i) \mid k_{1i} = \widetilde{k}_1\right\}_i$. Dans ce magasin, (k_{1i}, k_{2i}) est une clef, or $\forall i, k_{1i} = \widetilde{k}_1$. On peut donc définir $\mathcal{M}'(\widetilde{k}_1) = \{(k_{2i}, v_i)\}$ qui est identique à $\mathcal{M}(\widetilde{k}_1)$ à l'isomorphisme $\phi_{\widetilde{k}_1}$ près :

$$\begin{cases} \phi_{\widetilde{k}_1} : & \mathcal{M}(\widetilde{k}_1) & \longrightarrow & \mathcal{M}'(\widetilde{k}_1) \\ & ((\widetilde{k}_1, k_{2i}), v_i) & \mapsto & (k_{2i}, v_i) \end{cases}$$

les k_{2i} dans $\mathcal{M}(\widetilde{k}_1)$ sont *la clef de tri*. $\{(k_{ji}, \quad \{(k_{2i}, v_i) \mid k_{1i} = k_{ji}\}_i)\}_j$
Ainsi, on arrive la typologie suivante :

Définitions :
- les $\{(k_{2i}, v_i) | k_{1i} = k_{ji}\}_i$ sont appelées **partitions** ;
- les $(k_{1i})_i$ sont appelées **clés de partitionnement** des $\{(k_{2i}, v_i)\}_i$;
- les $(k_{2i})_i$ sont appelées **clefs de tri**.

Les clefs de partitionnement sont aussi appelées *clefs de distribution*.

3.4.3 Exemple

Nos données sont les suivantes :

```
UVS     ECTS
NF26    1
NF17    2
NF01    3
LO21    4
SY02    5
SY10    6
```

On éclate l'information des codes UV afin d'obtenir la **partition** suivante :

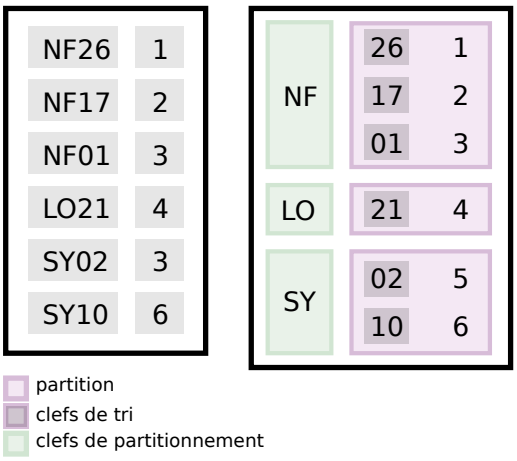

Figure 3.3 – Exemple de partitionnement

```
NF  26    1
 |  17    2
 |  01    3
LO  21    4
SY  02    5
 |  10    6
```

La ère colonne contient les **clefs de partitionnement** : (NF, LO, SY). La 2ème colonne contient les **clefs de tri** : (26,17,...,10). Voici la partition pour SY02 :

```
SY  02    5
 |  10    6
```

3.4.4 Utilisation des clefs

On utilise du *sharding* ou du *consistent hashing* pour distribuer les partitions, ce qui explique qu'on peut appeler la clef de partitionnement la **clef de distribution**.

Au sein d'une partition, les données sont souvent stockées d'une manière triée (d'où le nom de clé de tri) : c'est par exemple le cas dans LMDB qui n'utilise pas de partitionnement mais trie les clefs.

3.4.5 Retour sur les bases de données orientées colonnes

La famille de colonnes s'exprime de la façon suivante :

$$\left\{\left(k_{1j}, \underbrace{\left\{\left(k_{2i}, \overbrace{\{(c_{ik}, v_{jk})\}_k}^{\text{enregistrement}}\right)\right\}_i}_{\text{partition}}\right)\right\}_j$$

3.4.6 Illustration : Création de clefs

Exemple : cas de données géographiques.
(x_i, y_i) forme une clef, avec $x_i \in [\![0, D]\!]$ et $y_i \in [\![0, D]\!]$ tel que le choix de D conduise à décrire à l'échelle de la terre au millimètre pour enregistrement de coordonnées géographiques.

Question : comment trouver les hôtels les plus proches d'un point (x, y) donné ?
Première idée : partitionnement sur les x_i puis tri sur les y_i pour chaque x_i fixé… mais nécessite quand même beaucoup d'opérations.

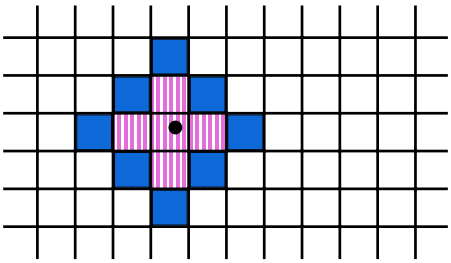

Figure 3.4 – Pavage sur les coordonnées, exemple de stratégie

Sur cet exemple, on explore en premier les 5 partitions roses (avec des hachures), on y trouve un hôtel. Il faut ensuite explorer toutes les autres partitions en bleu qui peuvent potentiellement avoir un hôtel plus proche que celui trouvé.

Autre idée : explorer des partitions connexes de points. Il faut donc ici *choisir la taille de la partition*. On a $x_i, y_i \in [\![0, 2^D + 1]\!]$. On peut écrire x_i et y_i selon leur représentation binaire, aussi appelé *développement binaire* :

$$\begin{cases} x_i = \sum_k 2^k X_{ik} \\ y_i = \sum_k 2^k Y_{ik} \end{cases}$$

On peut construire une autre représentation, $z \in [\![0, 2^{2d}[\![$. Pour cela on se base sur la représentation binaire des coordonnées qu'on intercale (ce qui revient à les espacer et les sommer).

x :		X_{i3}		X_{i2}		X_{i1}		X_{i0}
y :	Y_{i3}		Y_{i2}		Y_{i1}		Y_{i0}	
z :	Y_{i3}	X_{i3}	Y_{i2}	X_{i2}	Y_{i1}	X_{i1}	Y_{i0}	X_{i0}

Ainsi :

$$z = \sum_{k=0}^{d-1} 2^{2k} X_{ik} + \sum_{k=0}^{d-1} 2^{2k+1} Y_{ik}$$

Cela permet de trouver une représentation du pavé du point (x, y), à un niveau de précision qui sera déterminé par la troncature du développement.

3.5 Exemples

Il manque des trucs dans cet exemple

On a nécessairement besoin de stocker les données plusieurs fois (plusieurs tables).

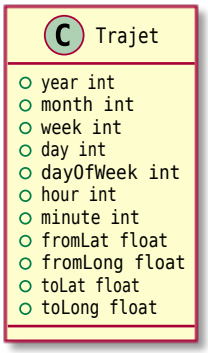

Figure 3.5 – Trajet

3.5.1 Exploitation temporelle

Cas de l'interrogation pour un jour donné, pour un jour de la semaine donnée ou pour une semaine donnée.

On veut des partitions les plus petites possibles pour pouvoir les répartir sur le cluster.
Contrainte : On a besoin de partitions dont la taille n'augmente pas avec le temps (pour conserver la scalabilité horizontale), *i.e.* des partitions de **taille bornée**, c'est mieux.

On se propose d'utiliser `(year, week, dayOfWeek)` comme clef de partitionnement.
Propriétés :

- La taille de partition correspond à une journée. En cas de flux continu, le nombre de partitions augmente mais leur taille est fixe.
- Accès aux données :
  - Pour accéder aux données par jour, il suffit d'accéder à une partition.
  - Pour accéder aux données d'un jour de la semaine sur une période de temps (ex : tous les lundis de mars 2017), on va interroger toutes les partitions qui correspondent à ce jour.
  - Pour accéder aux données par semaine, on interroge 7 partitions.

Trier les données au sein de la partition est hors de propos (dans le cadre de questions du type de celles du TD précédent). Mais, on peut le faire avec une clef artificielle ou tant qu'à faire une clef qui a du sens...

Ici, on a choisi d'utiliser une clé artificielle "réplicat" pour différencier les enregistrements, mais on aurait peut-être pu utiliser `bikeid`.

Le nombre de partitions doit être petit par rapport au nombre de données, et grand par rapport au nombre de nœuds.

3.5.2 Exploitation spatiale

Cohérence spatiale, par zone de départ.

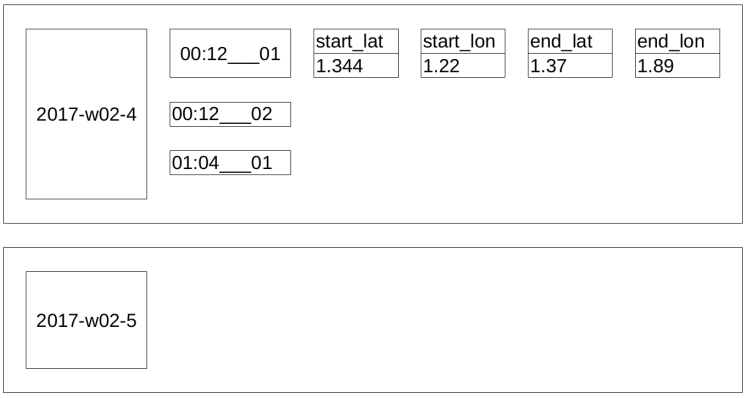

Figure 3.6 – expart2

— Première proposition : une partition par pavé géographique.
  — Perte de la scalabilité horizontale. Dans un contexte d'alimentation de BDD, on a un nombre de partitions borné par la surface de la ville, mais une taille de partition linéaire en fonction du volume total.
— Seconde proposition : une partition par pavé géographique et par an.
  — Nombre de partitions linéaire en fonction du temps : $temps \times surface$;
  — Taille des partitions bornée.

Interrogation :

— Proposition : une partition par pavé, interrogation de tous de la zone.
— Proposition alternative : une partition par pavé et par an, interrogation de toutes les années.

Calcul de l'`id` d'un pavé : $(x, y) \in [0, 1[^2$. Soit $N \in \mathbb{N}$; on a alors $Nx \in [0, N[$ et donc $\lfloor Nx \rfloor \in [\![0, N[\![$

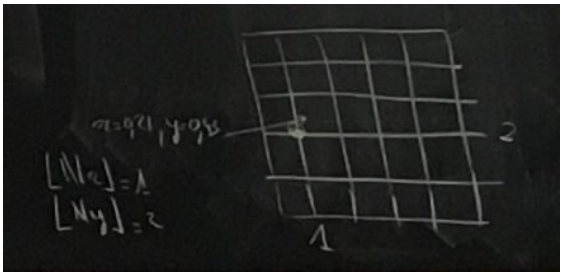

Figure 3.7 – Vive les rasters

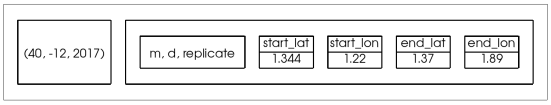

Figure 3.8 – expart

Chapitre 4

Distribution des calculs

4.1 Introduction et exemple

Exemple : Vous devez sommer les identifiants de n étudiants : comment faire ?
Solution 1 : une personne fait le tour des participants. Cette méthode demande $n-1$ opérations (et donc $n-1$ temps d'opération).
Solution 2 : on décompose le problème en deux sommes sur deux groupes, puis on fait la somme des deux groupes. Et ainsi de suite sur les sous-groupes récursivement jusqu'à ce qu'il n'y ait que deux éléments dans le sous-groupe, auquel cas on fait la somme des deux éléments. Cette solution demande toujours $n-1$ opérations, mais diminue le temps de calcul à $\lceil \log_2 n \rceil$ si les opérations sur des groupes différents peuvent s'effectuer en parallèle.

Ceci est valide car l'opération employée est associative et commutative.

Rappel de l'associativité : $(a+b)+c = a+(b+c)$
Rappel de la commutativité : $a+b = b+a$
La commutativité permet de ne pas ralentir les temps de calcul globaux, même quand un nœud met du temps à fournir une information.

4.2 Réduction

Définition : Soit $\perp$ un opérateur associatif et commutatif. Soit $(a_i)_{i\in[1,n]}$ un ensemble de valeurs. Soit b_i défini par récurrence comme étant :

$$\begin{cases} \forall i \geq 1, b_i, b_i & = b_{i-1} \perp a_i \\ b_0 & = a_0 \end{cases}$$

On nomme b_n la réduction de $(a_i)_i$ par $\perp$: $b_n = (...(((a_0 \perp a_1) \perp a_2) \perp a_3)...\perp a_n)$ et donc par associativité $b_n = a_0 \perp a_i \ldots \perp a_n$, que l'on peut noter $b_n = \perp_{i=0}^{n} a_i$.

Parenthèse sur les réductions infinies : $\frac{1}{2} - \frac{1}{3} + \frac{1}{4} - \frac{1}{5} + \frac{1}{6} - \frac{1}{7} + \frac{1}{8} \ldots$ et $\frac{1}{2} + \frac{1}{4} - \frac{1}{3} + \frac{1}{6} + \frac{1}{8} - \frac{1}{5} - \frac{1}{7} \ldots$ ne convergent pas vers la même valeur !

En effet $\sum_{i\in\mathbb{N}^\star} \frac{(-1)^i}{i}$ est un abus de notation, ce n'est pas univoque. Une réduction s'applique sur un ensemble **fini**, à la différence d'une série.

Exemple : Nous avons un ensemble de données $a_{ii\in I}$ fini, avec $\forall i, a_i \in \mathbb{R}$. On peut calculer la moyenne et l'écart-type, en utilisant la réduction, de la manière suivante :

$$\bar{x} = \frac{\sum_{i\in I} x_i}{\sum_{i\in I} 1} \qquad s^2 = \frac{\sum_{i\in I}(x_i - \bar{x})^2}{\sum_{i\in I} 1}$$

Cependant pour paralléliser toutes les réductions nous devrions plutôt utiliser :

$$s^2 = \frac{\sum_{i\in I} x_i^2}{\sum_{i\in I} 1} - \left(\frac{\sum_{i\in I} x_i}{\sum_{i\in I} 1}\right)^2$$

Sauf que ce n'est pas numériquement stable ! Il faut donc faire :

$$s^2 = \frac{\sum_{i\in I}(x_i - k)^2}{\sum_{i\in I} 1} - \left(\frac{\sum_{i\in I} x_i}{\sum_{i\in I} 1} - k\right)^2$$

On choisi k comme l'ordre de grandeur des données.

Éléments neutres

$\perp$ a un élément neutre $e \iff \forall a, a\perp e = a$ et $e\perp a = a$.
Exemples :

	sur $\mathbb{R}$	sur $\bar{\mathbb{R}}$
$+$	0	0
$\times$	1	1
min		$+\infty$
max		$-\infty$

4.3 Map Reduce : concept et technologie

Association de mapping et réduction.

4.3.1 Map

Transformation de la donnée en couple k/v :

$$m : \quad d \mapsto (k_i, v_i) = (m_k(d_i), m_v(d_i))$$

4.3.2 Reduction

Soit $\mathbb{K} = \{m_k(d_i)\}_i$. Pour $k \in \mathbb{K}$, $k : \underset{i \mid n_k(d_i)=k}{\perp}$ (on fait une réduction par valeur de la clef.)

4.3.3 Exemple, comptage de mots

Le comptage de mots est en quelque sorte le « Hello World » de l'analyse de données (exemple facile à comprendre vérifiant le bon fonctionnement de la techno). Nous analyserons le texte suivant : *'ô rage, ô désespoir, ô vieillesse...'*
Nous appliquons le mapping à chaque mot : mot $\mapsto$ (mot, 1). Réduction : $+$

$$\{('\hat{o}', 1),\ ('rage', 1),\ ('\hat{o}', 1),\ ('désespoir', 1),\ ('\hat{o}', 1),\ ('vieillesse', 1),\ \ldots\}$$
$$\text{avec } \mathbb{K} = \{'\hat{o}',\ 'rage',\ 'désespoir',\ 'vieillesse',\ \ldots\}$$

Résultat : $\{'\hat{o} : 3, 'rage' : 1, 'désespoir' : 1, ...\}$

4.3.4 Dans les technologies

Cela est emprunté aux langages fonctionnel. Des langages propose de telle API sur leur collections, c'est le cas de *Scala* par exemple sur lequel se base *Spark*

```scala
// Soit deux vecteurs
val vec1: List[Int] = List(1,2,4,3,5,6,1)
val vec2: List[Int] = List(3,0,3,1,5,9,1)

// Pour faire leur produit scalaire on peut faire
// un `map` sur leur association (via `zip`) suivi d'un `reduce` :
val prodScal: Int = vec1.zip(vec1,vec2)
                        .map((elem1,elem2) => elem1 * elem2)
                        .reduce((compo1,compo2) => compo1 +
                        ↪  compo2)

// ou de manière plus concise :
val prodScalPrime: Int = vec1.zip(vec1,vec2)
                        .map(_ * _)
                        .reduce(_ + _) // voir .sum simplement
```

Voir la doc des API de la `List` en Scala pour se faire une idée : `https://www.scala-lang.org/api/current/scala/collection/immutable/List.html`

Ce sont les mêmes API qui l'on retrouve avec *Spark* sur des collections distribuées appelées RDD (*Resilient Distributed Dataset*). Il y a plein d'autres *transformations* et *opérations* possibles comme `fold`, `aggregate`, `foldLeft`, `count`, `flatMap` et d'autres.[1]
L'API Python propose la même chose pour Spark via *PySpark*.[2]

4.4 Illustration de différents exemples de mapping-réduction.

Rappel sur les données : $\{(x_i, z_i)\}_i\ \forall i,\ x_i \in \mathbb{R},\ z_i \in [\![0, K[\![$

1. Voir la doc de l'API des RDD pour Scala : `https://spark.apache.org/docs/2.3.0/api/scala/index.html#org.apache.spark.rdd.RDD`
2. Voir la doc de l'API des RDD pour Python : `https://spark.apache.org/docs/2.3.0/api/python/pyspark.html#pyspark.RDD`

4.4.1 Médiane

Calculer directement la médiane de manière naïve est coûteux, par exemple en triant toutes les données. Cependant on peut savoir facilement si un nombre P_M (proposition de médiane) est inférieur ou supérieur à la médiane.
Objectif : calculer

$$\frac{\sum_i \mathbb{1}_{truc_i \leq P_M}}{\sum_i 1} = F_k(P_M)$$

F_k fonction de répartition.
On a $F_k(P_M) < \frac{1}{2} \implies P_M < \text{med}$.

Sous question : calculer $F_k(x)$?
map : ? ? red : ? ? map : ? ?

Objectif : calculer une médiane.

Algorithme de base : on trie les points et on sélectionne un point de *l'ensemble médian*. En effet, choisir le point central (ou la moyenne des deux points centraux) serait trop coûteux. **Cas particulier :** Dans le cas d'un *échantillon impair*, on acceptera de prendre un point entre la médiane -1 point et la médiane $+1$ point.

Complexité de cet algorithme : $\mathcal{O}(n \log n)$

Idée : soit m_k une proposition de médiane pour la classe k.

Exemple

Deux classes ω_0, ω_1 avec propositions initiales de médiane respective m_0 et m_1.

m_0	m_1
20	50

Si on définit le *mapping* et la *réduction* suivants :

$$\begin{cases} \text{map} & : & (x,z) & \mapsto & (z, (\mathbb{1}_{x<m_z}, \mathbb{1}_{x>m_z})) \\ \text{reduction} & : & ((a_0,b_0),(a_1,b_1)) & \mapsto & (a_0+a_1, b_0+b_1) \end{cases}$$

Données et mapping

x	z	Mapping	z	(a,b)
12	0	$\to$	0	$(1,0)$
24	0	$\to$	0	$(0,1)$
14.1	0	$\to$	0	$(1,0)$
7	0	$\to$	0	$(1,0)$
8	1	$\to$	1	$(1,0)$
31.2	1	$\to$	1	$(1,0)$
53	1	$\to$	1	$(0,1)$

Après réduction :

z	$(\sum_i a_i,\ \sum_i b_i)$
0	$(3,1)$
1	$(2,1)$

Nous pouvons voir que la médiane m_0 est trop grande (trois valeurs en dessous de celle-ci pour une seule au-dessus). En revanche, elle fait partie de l'intervalle médiane ± 1 point. On pourrait raffiner par dichotomie.

Algorithm 1: procédureTesterProposition (m) :

```
1  map ← ((x, z) ↦ (z, (𝟙_{z<m_z}, 𝟙_{z>m_z}))) ;
2  red ← (x, y ↦ x + y) ;
3  res ← procédureMapReduce(data, map, red) ;
4  ret ← () ;
5  foreach (z, counts) ∈ res do
6      if |counts[0] − counts[1]| ≤ 1 then
7          ret[z] ← 0 ;
8      else if counts[0] < counts[1] then
9          ret[z] ← 1 ;
10     else
11         ret[z] ← −1 ;
12 return (ret)
```

4.4.2 Recherche du min et max

Opérations :

$$\text{map} : (x, z) \mapsto (z, (x, x))$$
$$\text{red} : (\underline{a}, \bar{a}), (\underline{b}, \bar{b}) \mapsto (\min(\underline{a}, \underline{b}), \max(\bar{a}, \bar{b}))$$

Commutativité :

$$\begin{cases} \text{red}((\underline{a}, \bar{a}), (\underline{b}, \bar{b})) &= (\min(\underline{a}, \underline{b}), \max(\bar{a}, \bar{b})) \\ \text{red}((\underline{b}, \bar{b}), (\underline{a}, \bar{a})) &= (\min(\underline{b}, \underline{a}), \max(\bar{b}, \bar{a})) \end{cases}$$

Et par commutativité de max et de min, nous avons directement le résultat.

Associativité : On a par définition :

$$\begin{cases} \text{red}(\text{red}((\underline{a}, \bar{a}), (\underline{b}, \bar{b})), (\underline{c}, \bar{c})) &= (\min(\min((\underline{a}, \underline{b}), \underline{c}), \max(\max(\bar{a}, \bar{b}), \bar{c}))) \\ \text{red}((\underline{a}, \bar{a}), (\text{red}((\underline{b}, \bar{b}), (\underline{c}, \bar{c})) &= (\min(\underline{a}, \min(\underline{b}, \underline{c})), \max(\bar{a}, \max(\bar{b}, \bar{c}))) \end{cases}$$

NB : $\bar{a}$ (resp. $\underline{a}$) est le maximum (resp. minimum) des a.
or max et min sont associatifs ; on a donc :

$$\forall (a, b, c), \quad \begin{cases} \min(a, \min(b, c)) = \min(\min(a, b), c) \\ \max(a, \max(b, c)) = \max(\max(a, b), c) \end{cases}$$

ainsi

$$\text{red}(\text{red}((\underline{a}, \bar{a}), (\underline{b}, \bar{b})), (\underline{c}, \bar{c})) = \text{red}((\underline{a}, \bar{a}), \text{red}((\underline{b}, \bar{b}), (\underline{c}, \bar{c})))$$

```
1 ProcédureMinMax()
2     map ← ...
3     red ← ...
4     res ← ProcédureMapReduce(data, map, red)
5     retourner(res)
```

Mauvais algo

Faux

Complexité des opérations

- `TrouverMinMax` en $\mathcal{O}(n)$ est complètement parallélisable ;
- `Itérer` en $\mathcal{O}(\log n)$ (algorithme de dichotomie) ;
- `ProcédureTesterProposition` en $\mathcal{O}(n)$ et complètement parallélisable.

→ Complexité complète de `ProcédureTrouverMédianes` en $\mathcal{O}(n \log n)$.

Algorithm 2: ProcédureTrouverMédianes :

```
bornes ← TrouverMinMax() ;
while 1 > 0 do
    foreach (z, (b, b̄)) ∈ bornes do
        proposition[z] ← (b + b̄)/2 ;
        res_propo ← ProcédureTesterProposition(proposition[z]) ;
        foreach (z, (b, b̄)) ∈ bornes do
            if res_propo[z] = 1 then
                bornes[z][1] ← proposition[z];
            else if res_propo[z] = −1 then
                bornes[z][0] ← proposition[z];
            else
                bornes[z][1] ← bornes[z][0] ← proposition[z];
```

4.4.3 Fractiles

Soit $Q \in \mathbb{N}\backslash\{0,1\}$. On cherche les fractiles $\left\{\frac{q}{Q}\right\}_{q\in[\![1,Q-1]\!]}$

Idée : Un map reduce avec un ensemble de propositions nous retourne la fonction de répartition empirique dans ses points.

Soit $t_{k,q}$ le q-ème fractile de la k-ème classe (par exemple le premier quartile de la $2^{\text{ème}}$ classe) avec k l'identifiant de classe, et $q \in [\![1, p-1]\!]$

$$\text{map} : (x, z) \mapsto \left(z, (1, \mathbb{1}_{x\leq t_{z,1}}, \mathbb{1}_{x\leq t_{z,2}}, \dots, \mathbb{1}_{x\leq t_{z,p-1}})\right)$$

Exemple pour $t_{0,1} = 3, t_{0,2} = 26, t_{0,3} = 57$

x	z	Mapping	z	
7	0	$\to$	0	$(1,0,1,1)$
25	0	$\to$	0	$(1,0,1,1)$
12	0	$\to$	0	$(1,0,1,1)$
53	0	$\to$	0	$(1,0,0,1)$
28	0	$\to$	1	$(1,0,0,1)$
17	0	$\to$	1	$(1,0,1,1)$
68	0	$\to$	1	$(1,0,0,0)$

Après réduction :

z	
0	$(7,0,4,6)$

$(z, \sum_{z_i=z} 1, \sum_{z_i=z} \mathbb{1}_{x_i\leq t_{z,1}}, \sum \dots)$

on peut ainsi obtenir les images par la fonction de répartition empirique de la classe z aux points $(t_{z,j})_j$ ainsi :

$$\widehat{\mathcal{F}}_z(t_{z,j}) = \frac{\sum_{i|z_i=z} \mathbb{1}_{x_i<t_{z,j}}}{\sum_{i|z_i=z} 1}$$

4.4.4 Régression linéaire

Contexte : Magasin(année, mois, jour, heure, minute, seconde, x, y, age, dist)
On a une supposition sur la distribution de données : $\text{dist}_i = f(\text{age}_i) + \epsilon_i$ avec $f(x) = \beta_0 + \beta_1 x + \beta_2 x^2$.

Formule :

$$X_h = \left[1|\text{age}_i|\text{age}_i^2\right]_{i\in S_n} \in \mathcal{M}_{n,3} \qquad Y_h = [\text{dist}_i]_{i\in S_n} \in \mathcal{M}_{n,1}$$

$$Y_h = X_h\beta_n + \epsilon$$
$$\beta_h = (X_h^\top X_h)^{-1} X_h^\top Y_h$$

Nous pouvons développer le calcul de β_h :

$$X_n^\top X_n = \sum_{i\in S_n} (1, \text{age}_i, \text{age}_i^2)^\top (1, \text{age}_i, \text{age}_i^2) \tag{4.1}$$

$$= \sum_{i\in S_n} \begin{pmatrix} 1 & age & age^2 \\ & age^2 & age^3 \\ & & age^4 \end{pmatrix} \tag{4.2}$$

Ce qui nous ramène à des réductions (somme de matrices 3×3).

Exercice : écrire une régression linéaire dist = age + $b$age + $c$age^2 + ϵ avec un modèle par heure.
Mapping : $(\dots, h, \dots) \mapsto (h, ([1, age, age^2]^\top [1, age, age^2], [1, age, age^2]^\top dist)$
Réduction par clef : $((M_1, \theta_1), (M_2, \theta_2)) \mapsto (M_1 + M_2, \theta_1 + \theta_2)$
Mapping final : $[\![0, 23]\!] \times (\mathbb{R}^{3\times 3} \times \mathbb{R}^3) \to [\![0, 23]\!] \times \mathbb{R}^3$
$(h, (M, \theta)) \mapsto (h, M^{-1}\theta)$

4.4.5 Arbre de décision

Comment trouver la première frontière ? $Q(s) = f((n_{cp})_{c\in\{1,2\}, p\in\{\leq,>\}})$ Faire une fonction qui calcule la valeur du critère.
- objectif pour une variable k on veut calculer
$m^k_{1,\leq}(s)$ $m^k_{2,\leq}(s)$ $m^k_{1,>}(s)$ $m^k_{2,>}(s)$
$m^k_{1,>}(s)$: nombre d'éléments de la classe 1 dont $x^{(k)}$ est supérieur à s
Mapping : $\mathbb{D} \to \mathbb{R}^4$
$(x^{(1)}, x^{(2)}, l) \mapsto (\mathbb{1}_{l=1}\mathbb{1}_{x^{(1)}\leq s}, \mathbb{1}_{l=2}\mathbb{1}_{x^{(1)}\leq s}, \mathbb{1}_{l=1}\mathbb{1}_{x^{(1)}>s}, \mathbb{1}_{l=2}\mathbb{1}_{x^{(1)}>s})$ Réduction : +

On est capable de calculer $Q^k(s)$ par k et s connu => on choisit l'argmin (ou du moins une approximation satisfaisante). À l'itération j on a un classifieur qui à un élément de l'espace envoie un identifiant de partition.

4.4.6 Recherche de points le plus proche dans un graphe routier

Projeter le point qui nous intéresse (notre localisation, par exemple) sur une composante connexe *agréable*, pas forcément la plus proche.

Pour un nouveau jeu de points hors de composantes connexes, trouver la composante connexe la plus agréable (*i.e.* qui contient un point le plus proche d'un de ces nouveaux points).

Données :

$$\{(x_i, y_i, z_i, t_i, c_i, \dots)\}_i$$

x_i et y_i étant les coordonnées géographiques, z_i l'altitude, t_i le "stuff" (informations supplémentaires dépendant de la situation), c_i une composante connexe.

Question : Trouver par composante connexe les coordonnées du point le plus proche d'un point (a, b). *Ce genre de question est susceptible de tomber au final, mais il a choisi un autre exemple.* Pour cela, on choisit le mapping suivant :

$$\text{map} : (x, y, t, z, c, ...) \mapsto (c, (x, y, \|(x, y) - (a, b)\|^2))$$

On choisit la composante connexe c_i comme clé car on effectuera une réduction dessus.

Données et mapping

x	y	z	t	c	Mapping	c	
1	0			0	$\to$	0	$(1, 0, 4)$
0	1			1	$\to$	1	$(0, 1, 2)$
1	1			0	$\to$	0	$(1, 1, 5)$
0	0			1	$\to$	1	$(0, 0, 1)$

On peut alors poser l'opération de réduction suivante :

$$\text{red} : (x_1, y_1, d_1), (x_2, y_2, d_2) \mapsto \begin{cases} (x_1, y_1, d_1) \text{ si } d_1 < d_2 \\ (x_2, y_2, d_2) \text{ sinon} \end{cases}$$

Pour commutativité, on doit poser des hypothèse (non équidistance de points par exemple). Mais avec un peu plus de plomberie, on peut résoudre le problème.

Cette réduction retourne alors la distance minimum avec les coordonnées d'un point de la composante c.

Chapitre 5

RGPD : Règlement Général sur la Protection des Données

Règlement européen qui sert de référentiel (ce n'est pas une directive européenne, pas besoin d'être retranscrit pas les états pour avoir une application nationale $\Longrightarrow$ il s'applique déjà et dans toute l'europe).

À qui il s'adresse ? À tous, à tous les niveaux.

Sources d'information :

— texte disponible sur europa-lex

— `https://atelier-rgpd.cnil.fr/` (en licence CC-BY-NC-ND)

— documentation CLUSIF

5.1 Données personnelles

Donnée personnelle : (non sensible) donnée permettant d'identifier un individu, (sensible) contenant des informations d'origine raciale, ethnique, politique, philosophique, religieuse, syndicale, santé, sexuelle.

Exemple : données à caractère personnel :

— état-civil, identité

— vie personnelle

— vie professionnel

— info éco et financière

— donnée de connexion

— donnée de localisation

Données personnelles sensibles

— NIR (ñuméro de sécurité sociale)

— données biométriques

— données bancaires

Données sensibles au sens RGPD

— raciale, ethnique

— judiciaires

— données concernant les mineurs

A/ Ouverture de droits

— accès aux données

— rectification

— droit à l'oubli

— limitation

— portabilité des données
— d'opposition
— refus de décision automatique

A/ Violation de données personnelles

Obligation de mécanisme de détection
Obligation d'avoir préalablement un plan d'action
Obligation de signalement à l'autorité compétente (CNIL pour la France)

A/ Traitements

Respect de la finalité des données (quelques exceptions pour les statistiques, cas de force majeure). Les données personnelles ne doivent être utilisées que sous stricte nécessité. Inscrit au registre, autorisé par le responsable de traitement.

Annexe A

Retours sur le premier TD

je ne sais pas quoi faire de ça, je le garde pour l'instant

A.1 Retour sur l'indexation :

Trop de données pour que cela puisse être tracé.

Représenter les classifieurs plus que les classifications pour résumer l'information.

On pouvait utiliser une clef artificielle... mais il y a mieux. Dans le cas d'une clef artificielle ; il faut parcourir toutes les données. Le bon choix d'une clef permet d'explorer les données de manière adaptée. On peut par exemple choisir une indexation temporelle pour effectuer une classification sur des périodes ou dates précises.

On va stocker les données de manière dupliquée pour répondre à des questions de manière adaptée et rapide : même donnée mais indexation différente.

A.2 Retour sur le k-means

Le k-means consiste à itérer deux étapes (classification et apprentissage).

```
dataSet : jeu de données considéré
centroïdes ← [x_i in R^d]
while [condition d'arrêt du k-means]:
 |  next_sum_pts ← 0_R^(K\times d)
 |  next_sum_nb ← 0_R^K
 |  for x_i in dataSet:
 |   |  k ← index du centroïde le plus proche
 |   |  next_sum_pts[K] ← next_sum_pts[K] + x_i
 |   |  next_sum_nb[K] ← next_sum_nb[K] + x_i
 |
 |  for k in 0:K-1:
 |   |  centroïdes[k] ← next_sum_pts[k] / next_sum_nb[k]
 |
```

A.2.1 Exercice :

Calculer la moyenne et variance empirique d'un jeu de données de manière distribuée avec stabilité numérique.

Bibliographie

[1] Seth Gilbert and Nancy A. Lynch. An Illustrated Proof of the CAP Theorem. [en ligne] `https://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf` dernière consultation : 26 mai 2018.

[2] KB Petersen. *The Matrix Cookbook*. 2017. [en ligne] `https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf` dernière consultation : 22 mai 2018.

[3] StackOverFlow. Spark partitioning. [en ligne] `https://stackoverflow.com/questions/43644247/understanding-spark-partitioning` dernière consultation : 27 mai 2018.

[4] Michael Whittaker. An illustrated proof of the cap theorem. [en ligne] `https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/` dernière consultation : 26 mai 2018.