MIF04 – Gestion de données pour le Web
TD – Map/Reduce en MongoDB

Algorithm 1 Word count (algorithmique)

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)
5:         end for
6:     end method
7: end class
1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         end for
7:         Emit(term t, count sum)
8:     end method
9: end class
```

Algorithm 2 Word count `javascript` pour MongoDB

```
var map = function() {
  var content =  this.content.toLowerCase().split(/[\s,!-.;?_:*]+/);
  // tokenization using js String.split() with a regex
  // \s matches a single white space character, including space, tab, form feed
      , line feed.
  // see https://developer.mozilla.org/en/docs/Web/JavaScript/Guide/
      Regular_Expressions
  for (var i = 0, n = content.length; i < n; i++) {
    if (content[i])
      emit(content[i], 1);
  }
};

var reduce = function(key, values) {
  var sum = 0;
  for (var i = 0, n = values.length; i < n; i++) {
    sum = sum + values[i];
  }
  return sum;
};
```

Exercice 1 : analyse et extension du problème *word count* (*[2, Chapitre 3]*)

L'algorithme 1 définit en pseudo-code les classes MAPPER et REDUCER d'HADOOP pour le calcul du nombre d'occurrences de mots dans une collection de documents. L'algorithme 2 donne une implémentation équivalente en `javascript` pour MONGODB. Pour l'exemple, on considère la collection de paires clef/valeur suivante :

Doc1 « *appreciate the fold* »

Doc2 « *cata equals fold* »

Doc3 « *fold the fold* »

1. En supposant que les documents **Doc1** et **Doc2** se situent sur un premier nœud et **Doc3** sur un autre, donner les résultats intermédiaires obtenus par chaque MAPPER.
2. En supposant que l'espace des mots soit divisé en deux, les mots inférieurs à « *f* » sur un nœud et ceux après « *f* » sur un autre, donner les résultats intermédiaires obtenus des REDUCER qui regroupent les clefs puis les résultats finaux des REDUCER. Discuter la pertinence de cette règle de répartition des clefs dans ce cas d'application.
3. On considère maintenant une agrégation locale des données : chaque MAPPER va calculer le nombre d'occurrence de chaque mot dans sa collection de documents. Donner le pseudo-code ou le code `javascript` en utilisant un *tableau associatif* (appelé aussi *dictionnaire*) pour stocker les calculs intermédiaires.
4. Reprendre la première question avec cette nouvelle version.
5. La classe MAPPER d'Hadoop dispose également des méthodes :
   — INITIALIZE qui est appelée *avant* le parcours de la collection
   — CLOSE qui est appelée *après* le parcours de la collection
   Proposez une variante du MAPPER avec tableau associatif qui utilise ces deux méthodes pour effectuer un maximum d'agrégations dans le MAPPER.
6. Reprendre la première question avec cette nouvelle version.
7. On souhaite construire un index inversé, qui à chaque mot associe la *liste des identifiants de documents où le mot apparaît*. Donner le code `javascript` pour construire l'index inversé.

Exercice 2 : calcul de moyenne (*[2, Chapitre 3]*)

1. Donner le pseudo-code ou le code `javascript` qui calcule la moyenne des valeurs associées à une clef, e.g., $\text{REDUCE}(w, [1, 2, 3, 4]) = (w, 2.5)$ dans le cas où le `reduce` est appelé *exactement une fois sur chaque clef*, comme c'est le cas dans Hadoop.
2. Proposer une variante qui fonctionne en MONGODB où il y a des *re-reduce*. Il faudra utiliser la fonction `finalize` de MONGODB qui a le même prototype que `reduce` mais est appelée une fois tous les REDUCER terminés.

Exercice 3 : algèbre relationnelle en Map/Reduce (*[3, Chapitre 2.3]*)

Dans cet exercice on s'intéresse à implémenter les principaux opérateurs de SQL avec des tâches Map/Reduce dans MONGODB. Pour les exemples, on considère deux relations $R(A, B)$ et $S(X, Y)$. On considère qu'un tuple $R(a, b)$ est représenté dans MONGODB par un objet `json` de la forme $\{\_id : i, value : \{rel : R, A : a, B : b\}\}$ où i est un identifiant automatiquement généré par MONGODB. Donner les codes `javascript` des fonctions `map` et `reduce` des requêtes suivantes.

1. ```
   SELECT DISTINCT A,B
   FROM R
   ```

2. ```
   SELECT A, count(*)
   FROM R
   GROUP BY A
   ```

3. ```
   SELECT *
   FROM R
   UNION
   SELECT X as A, Y as B
   FROM S
   ```

4. ```
   SELECT *
   FROM R
   INTERSECT
   SELECT X as A, Y as B
   FROM S
   ```

5. ```
   SELECT *
   FROM R
   WHERE A < 1;
   ```

6. ```
   SELECT R.A, R.B, S.Y
   FROM R INNER JOIN S on R.A = S.X
   ```

7. On aimerait calculer une jointure sur des collections différentes en MongoDB, malheureusement, ce n'est pas possible. Proposer une solution alternative.

Exercice 4 : définitions fonctionnelles des fonctions Map et Reduce ([1])

On s'intéresse à un cas particulier de tâches Map/Reduce qui sont représentables par la composition suivante, où $g : (K_1 \times V_1) \to [(K_2 \times V_2)]$ est la fonction qui est « mappée », $(\oplus) : V_2 \to V_2 \to V_2$ est la fonction d'agrégation, $\text{grp} : [(K_2 \times V_2)] \to [(K_2 \times [V_2])]$ la fonction de regroupement et $+\!\!+ : [A] \to [A] \to [A]$ la concaténation.

$$\text{mapreduce}(g)(\oplus) = \text{map}((x, ys) \mapsto (x, \text{red}(\oplus)(ys))) \circ \text{grp} \circ \text{red}(+\!\!+) \circ \text{map}(g)$$

On donne les définitions inductives de $\text{red}(\oplus)$ et de $\text{map}(g)$ sur des listes non-vides :

$$\begin{aligned}
\text{red}(\oplus)[x] &= x \\
\text{red}(\oplus)(xs +\!\!+ ys) &= (\text{red}(\oplus)(xs)) \oplus (\text{red}(\oplus)(ys)) \\
\text{map}(g)[x] &= [g(x)] \\
\text{map}(g)(xs +\!\!+ ys) &= \text{map}(g)(xs) +\!\!+ \text{map}(g)(ys)
\end{aligned}$$

1. En utilisant le même formalisme, donner les types des fonctions $(+\!\!+)$, map et red.
2. Donner les types des résultats intermédiaires des compositions de $\text{mapreduce}(g)(\oplus)$ et vérifier que la définition de $\text{mapreduce}(g)(\oplus)$ a bien pour type $[(K_1 \times V_1)] \to [(K_2 \times V_2)]$.
3. En utilisant uniquement des définitions fonctionnelles, donner les définitions de g et $\oplus$ pour que $\text{mapreduce}(g)(\oplus)$ calcule :
   — le nombre de valeurs associées à chaque clef ;
   — le nombre d'occurrence de chaque valeur ;
   — le nombre total de valeurs ;
   — le nombre total de valeurs *différentes* ;
4. Montrer que la loi $\oplus$ doit être associative, on utilisera pour cela la définition de $\text{red}(\oplus)$ et l'associativité de $+\!\!+$.
5. À partir de la preuve précédente pour les listes, donner l'argument pour prouver que $\oplus$ doit être commutatif dans le cas des multi-ensembles et idempotent dans le cas des ensembles.
6. Montrer que si $g(x \oplus y) = g(x) \otimes g(y)$ alors $g \circ \text{red}(\oplus) = \text{red}(\otimes) \circ \text{map}(g)$ par induction sur les listes.

Références

[1] R. Lämmel. Google's mapreduce programming model – revisited. *Science of Computer Programming*, 70(1) :1 – 30, 2008.

[2] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010. http://lintool.github.io/MapReduceAlgorithms/.

[3] A. Rajaraman and J. D. Ullman. *Mining of massive datasets*. Cambridge University Press, Cambridge, 2012.

A Jeu de données pour l'exercice 3

Ne contient pas les champs `_id` qui seront générés automatiquement à l'importation. À importer avec la commande suivante.

```
mongoimport --host localhost --db mif04 --collection sql --drop sql_dataset.json
```

```
{ "value" : { "rel" : "R", "A" : 0, "B" : 1 } }
{ "value" : { "rel" : "R", "A" : 0, "B" : 1 } }
{ "value" : { "rel" : "R", "A" : 0, "B" : 2 } }
{ "value" : { "rel" : "R", "A" : 1, "B" : 1 } }
{ "value" : { "rel" : "S", "X" : 0, "Y" : 0 } }
{ "value" : { "rel" : "S", "X" : 1, "Y" : 1 } }
{ "value" : { "rel" : "S", "X" : 2, "Y" : 1 } }
```

Corrections

Solution de l'exercice 1

1. — **Doc1** et **Doc2** : $[(appreciate, 1), (the, 1), (fold, 1), (cata, 1), (equals, 1), (fold, 1)]$
   — **Doc3** : $[(fold, 1), (the, 1), (fold, 1)]$
2. En cours on présente l'étape Sort/Group/Shuffle à la place Reducer : c'est simplement la fonction qui regroupe par clef les valeurs du Mapper avant de les transmettre à la fonction Reduce :
   — 1er Reducer
   — entrée $[(appreciate, 1), (cata, 1), (equals, 1)$
   — sortie $[(appreciate, 1), (cata, 1), (equals, 1)]$
   — 2e Reducer
   — entrée $[(fold, [1, 1, 1, 1]), (the, [1, 1])]$
   — sortie $[(fold, 4), (the, 2)]$

   Les occurrences des mots ne sont pas équilibrées dans une langue naturelle, la division du travail entre n Reducer basée sur l'ordre alphabétique conduira à des charges inégales. Pour s'assurer une répartition plus équilibrée, on pourrait par exemple répartir non pas selon les mots w mais par exemple selon $h(w)$ où h est une fonction de hashage cryptographique.
3. Ici, on souhaite renvoyer Emit(`fold`, 2) dès le Mapper plutôt que de laisser le Reducer agréger puis envoyer la liste de valeurs $[1, 1]$ au Reducer.

   class Mapper
   method Map(docid a, doc d)
   $H \leftarrow$ new AssociativeArray
   for all term $t \in$ doc d **do**
   $H\{t\} \leftarrow H\{t\} + 1$
   end for
   for all term $t \in H$ **do**
   Emit(term t, count $H\{t\}$)
   end for
   end method
   end class

   L'équivalent en `javascript`. On remarque qu'on se sert d'un objet *json* comme d'un dictionnaire (ce qui est apparemment une pratique usuelle).

```
var map = function() {
  var local  = {};
  var content =  this.content.toLowerCase().split(/[\s,!-.;?_:*]+/);
  for (var i = 0, n = content.length; i < n; i++) {
    if (local[content[i]])
      local[content[i]]++ ;
    else
      local[content[i]]=1 ;

  }
  for (var key in local) {
    emit(key,local[key]);
  }
};
```

4. On va directement agréger les occurrences multiples *dans un document* :
   — **Doc1** et **Doc2** : $[(appreciate, 1), (the, 1), (fold, 1), (cata, 1), (equals, 1), (fold, 1)]$
   — **Doc3** : $[(fold, 2), (the, 1)]$
5. On obtient

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     end method
5:     method Map(docid a, doc d)
6:         for all term t ∈ doc d do
7:             H{t} ← H{t} + 1
8:         end for
9:     end method
10:    method Close
11:        for all term t ∈ H do
12:            Emit(term t, count H{t})
13:        end for
14:    end method
15: end class
```

Par contre, on a un problème pour traduire cet algorithme dans MongoDB car il n'y a pas d'équivalent de la méthode Close.

6. On va directement agréger les occurrences multiples *dans une collection de documents* :
   — **Doc1** et **Doc2** : $[(appreciate, 1), (the, 1), (fold, 2), (cata, 1), (equals, 1)]$
   — **Doc3** : $[(fold, 2), (the, 1)]$
7. On s'inspire du pseudo-code du *word count*, on va simplement accumuler les identifiants dans un tableau au lieu d'incrémenter le nombre d'occurrences et on va *flatten* la liste de listes dans le Reducer.
   On remarque qu'on doit encapsuler le résultat du `reduce` dans un objet `json` car MongoDB ne permet pas de retourner des listes. On remarque aussi que ce *wrapping* a lieu dès l'`emit` du `map`, car il faut que le type de retour du `map` et le type des valeurs du `reduce` soient les mêmes : c'est une des contraintes de MongoDB.

```
var map = function() {
  var local  = {};
  var content =  this.content.toLowerCase().split(/[[\s,!-.;?_:*]+]+/);
  for (var i = 0; i < content.length; i++) {
    if(content[i])
      if (local[content[i]])
        local[content[i]].push(this.source) ;
      else
        local[content[i]]=[this.source];
  }
  for (var key in local) {
    emit(key,{list : local[key]});
  }
};


var reduce = function(key, values) {
  var data = {list : []};
  values.forEach(function(v) {
    data.list = data.list.concat(v.list);
  });
  return data;
};
```

Solution de l'exercice 2

1. Le Mapper est la fonction identité.

```
1: class Reducer
2:     method Reduce(string t, integers [r_1, r_2, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r_1, r_2, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         end for
9:         r_avg ← sum/cnt
10:        Emit(string t, integer r_avg)
11:    end method
12: end class
```

En `javascript`.

```
var map = function() {
  emit(this.key, this.val);
};

// /!\ WARNING /!\ code d'exemple INCORRECT
var reduce = function(key, values) {
  var sum = 0;
  var count = 0;
  values.forEach(function(v) {
    sum += v;
    count++;
  });
  return sum/count;
};
```

2. MongoDB impose que la fonction `reduce` soit associative pour gérer les *re-reduce*. Il faut donc maintenir les sommes et les poids séparément avant de calculer les moyennes, car la moyenne n'est pas une opération associative (e.g., avec $m(x, y) = (x + y)/2$ on a $m(m(a, b), c) \neq m(a, m(b, c))$), en revanche la somme membre à membre l'est (e.g., avec $(x_0, y_0)+(x_1, y_1) = (x_0+x_1, y_0+y_1)$, on a $((x_0, y_0)+(x_1, y_1))+(x_2, y_2) = (x_0, y_0)+((x_1, y_1)+(x_2, y_2))$. Attention à bien avoir les mêmes types dans le `emit` du `map` et le paramètre `values` du `reduce`.

```
var map = function() {
  emit(this.key, {sum : this.val, count : 1});
};

var reduce = function(key, values) {
  var lsum = 0;
  var lcount = 0;
  values.forEach(function(v) {
    lsum   += v.sum;
    lcount += v.count;
  });
  return {sum : lsum, count : lcount};
};

var finalize  = function(key, val) {
  val.avg = val.sum/val.count;
  return val;
};
```

Solution de l'exercice 3

À titre préliminaire, on peut noter qu'en programmation fonctionnelle, fixer les types d'une

fonction la « rigidifie », et dans certains cas, on a même plus de choix sur la définition possible. Par exemple, les fonctions *strictes* swap : $\forall A, \forall B, A \times B \to B \times A$ ou id : $\forall A, A \to A$ n'ont qu'une seule définition possible. Ainsi, dans cet exercice, fixer le type de ce qu'on l'émet dans le MAPPER donne déjà une bonne partie de la réponse à la question.

1. Là, on va utiliser le regroupement sur les clefs pour éliminer les doublons : la clef est simplement la paire (A, B). En fait, on peut remarquer avec la requêtes ci-après que faire une `GROUP BY` sur tous les attributs du `SELECT` revient à faire un `DISTINCT` : c'est finalement plutôt ça qui est fait par MONGODB.

   ```
   SELECT A,B
   FROM R
   GROUP BY A,B
   ```

   Soit en `javascript` :

   ```
   var map = function() {
     if(this.value.rel === 'R')
       emit({A : this.value.A, B : this.value.B}, null);
   };

   var reduce = function(key, values) {
     return null;
     //la valeur n'a ici aucune importance
   };
   ```

2. Pas grand chose à signaler ici, sauf que c'est un cas d'application paradigmatique.

   ```
   var map = function() {
     if(this.value.rel === 'R')
       emit(this.value.A, 1);
   };

   var reduce = function(key, values) {
     return Array.sum(values);
   };
   ```

3. On va simplement effectuer un renommage des champs des objets qui représentent les tuples et tout accumuler. Pour le cas du `UNION` sans `ALL`, les champs du `SELECT` sont dans la clef comme dans la question avant, voir le code ci-après. Pour le cas avec le `ALL`, on fera plutôt comme pour `SELECT * FROM R` plus loin.

   ```
   var map = function() {
     switch(this.value.rel) {
       case 'R':
           emit({A : this.value.A, B : this.value.B}, null);
           break;
       case 'S':
           emit({A : this.value.X, B : this.value.Y}, null);
           break;
       default:
           break;
     }
   };

   var reduce = function(key, values) {
     return null;
     //la valeur n'a ici aucune importance
   };
   ```

4. Pour l'intersection, on va reprendre la technique de la question précédente en ajoutant l'identifiant de relation dans le value du `emit`. Dans le `reduce` on va vérifier que la paire (A, B) apparaît bien dans R *et* dans S. Comme *toutes les paires* sont émises par `emit`, toutes les

clefs seront présentes dans les appels à `reduce` *qui ne peut pas supprimer de clefs* (`reduce` peut seulement agréger les valeurs associées à une clef). Il faut donc faire un filtrage à l'extérieur du `reduce` et du `finalize`, par exemple `monResultat.filter(function(x){return (x.value.R && x.value.S)})`.

```
var map = function() {
  switch(this.value.rel) {
    case 'R':
        emit({A : this.value.A, B : this.value.B}, {R : true, S : false});
        break;
    case 'S':
        emit({A : this.value.X, B : this.value.Y}, {R : false, S : true});
        break;
    default:
        break;
  }
};

var reduce = function(key, values) {
  //si on est bien dans R ET dans S
  return values.reduce(
    (acc,x) => ({R:acc.R || x.R, S:acc.S || x.S}),
    {R : false, S : false}
    );
};
```

5. Les `_id` des tuples étant uniques, on est sûr qu'il y a exactement une seule valeur dans le paramètre `values` de `reduce`. On retourne les objets tels qu'ils sont stockés dans la base originale, mais on pourrait faire autrement.

```
var map = function() {
  if (this.value.rel === 'R' && this.value.A < 1)
    emit(this._id, this.value);
};

var reduce = function(key, values) {
  return values[0];
  // nécessaire pour assurer que le type du return est bien le même que le
       emit du map
  // toutefois comme on émet la clef on est sur que reduce ne sera jamais
      appellé car |values| = 1 et on peut donc tout aussi bien écrire "
      return undefined ;"
};
```

6. L'astuce est de voir qu'on va mettre dans la clef du `emti` les colonnes sur lesquelles on fait la jointure. On sépare le groupement dans le `reduce` du produit cartésien dans le `finalize` à cause des `re-reduce`. Même problème que pour l'intersection : les singletons qui ont un produit cartésien vide apparaissent car *toutes* les valeurs de la clef de jointures sont émises par `map`, il faut faire un filtrage final. Au final, cet exercice est là pour faire comprendre qu'on se retrouve presque à écrire un algorithme de jointure à la main et que donc le paradigme n'est pas vraiment fait pour ça.

```
var map = function() {
  switch(this.value.rel) {
    case 'R':
        emit(this.value.A, {rel : 'R', valR : [{A : this.value.A, B : this
            .value.B}], valS : [] } );
        break;
    case 'S':
        emit(this.value.X, {rel : 'S', valS : [{X : this.value.X, Y : this
            .value.Y}], valR : [] } );
        break;
```

```
    default:
        break;
  }
};

var reduce = function(key, values) {
  // le group automatique entre le map et le reduce va regroupper
  // tous les tuples de R selon la valeur de A et ceux de S sur la valeur
      de X
  // s'ils sont dans values, c'est que la condition R.A == S.X est
      satisfaite

  var res = {rel: 'J', valR : [], valS : []};

  // on agrege les valeurs de R et de S separement dans la "relation" J
  for (var i = 0, n = values.length; i < n ; i++){
    res.valR = res.valR.concat(values[i].valR);
    res.valS = res.valS.concat(values[i].valS);
  }

  return res;
};

var finalize = function(key, value){
  //a ce point là tous les re-reduce ont été exécuté donc dans values
  // on ne peut avoir que trois possibilités :
  // - un 'J' avec une liste dans valR et une autre dans valS
  // - un 'R' avec un unique objet dans valR
  // - un 'S' avec un unique objet dans valS

  var res = {rel : 'J', valJ : []};

  // on va éviter les 'R' et 'S' qui passent à travers reduce
  if (value.rel == 'J'){
    //on calcule le produit cartésien des tableaux des valeurs issues de R
        et celles issues de S.

    //pour tout tuple où R.A a pour valeur 'key'
    for (var i = 0, n = value.valR.length; i < n ; i++){
      // o est un objet temporaire qui stocke le tuple en question
      var o = value.valR[i];
      //pour tout tuple où R.X a pour valeur 'key', la mëme que R.A
      for (var j = 0, p = value.valS.length; j < p ; j++){
        // on ajoute à o un champs Y pour la valeur de S.Y
        o.Y = value.valS[j].Y;
        // on ajoute o au résultat
        res.valJ.push(o);
      }
    }
  }
  return res;
};
```

7. Le problème se pose quand on souhaite accéder à des collections différentes. Une solution est de le faire à l'extérieur de MongoDB, au niveau applicatif. C'est une solution acceptable si on a des sélections très filtrantes (e.g., on veut les S associés à une famille de valeurs $R.A$ très réduite). Une solution plus dans l'esprit MongoDB, consisterait à dénormaliser le schéma et à ajouter directement aux objets qui représentent les tuples de R les objets de S associés, voir ce qu'on a fait pour les arbres XML à partir d'une base relationnelle avant dans l'UE où on choisit d'associer les inscriptions entres UEs et étudiants soit dans l'entité *UE*, soit dans l'entité *étudiant*.

Solution de l'exercice 4

1. Pour les types des fonctions de base $+\!\!+ : [[A]] \to [A]$, map : $(A \to B) \to ([A] \to [B])$ et red : $(B \to B \to B) \to ([B] \to B)$. Remarques : c'est le type le plus général pour map, qu'on instancie dans mapreduce avec $A = (K_1 \times V_1)$ et $B = [(K_2 \times V_2)]$.
2. Pour le typage de l'expression mapreduce$(g)(\oplus)$ on remarque d'abord que la fonction $r = (x, ys) \mapsto (x, \text{red}(\oplus)(ys))$ a pour type $(A \times [B]) \to (A \times B)$ puis on obtient la chaîne suivante de compositions :

$$[(K_1 \times V_1)] \xrightarrow{map(g)} [[(K_2 \times V_2)]] \xrightarrow{+\!\!+} [(K_2 \times V_2)] \xrightarrow{\text{grp}} [(K_2 \times [V_2])] \xrightarrow{\text{map}(r)} [(K_2 \times V_2)]$$

3. On va prendre les définitions suivantes :
   — $g = (k_1, v_1) \mapsto (k_1, 1)$ et $(\oplus) = (+)$
   — $g = (k_1, v_1) \mapsto (v_1, 1)$ et $(\oplus) = (+)$
   — $g = (k_1, v_1) \mapsto (\star, 1)$ et $(\oplus) = (+)$ où $\star$ est un symbole arbitraire
   — on va utiliser deux jobs : un premier qui assure l'unicité et ensuite un second qui compte :
      — $g_1 = (k_1, v_1) \mapsto (v_1, \star)$ et $(\oplus_1) = (\star, \star) \mapsto \star$;
      — $g_2 = (v_1, \star) \mapsto (\star, 1)$ et $(\oplus_2) = (+)$
4. Il faut rappeler que la concaténation des chaînes est associative, tout le reste est basé sur la définition de red$(\oplus)$:

$$\begin{aligned}
(x \oplus y) \oplus z &= (\text{red}(\oplus)[x] \oplus \text{red}(\oplus)[y]) \oplus \text{red}(\oplus)[z] \\
&= \text{red}(\oplus)([x] +\!\!+ [y]) \oplus \text{red}(\oplus)[z] \\
&= \text{red}(\oplus)(([x] +\!\!+ [y]) +\!\!+ [z]) \\
&= \text{red}(\oplus)([x] +\!\!+ ([y] +\!\!+ [z])) && \textit{associativité de } +\!\!+ \\
&= \text{red}(\oplus)[x] \oplus \text{red}(\oplus)([y] +\!\!+ [z]) \\
&= \text{red}(\oplus)[x] \oplus (\text{red}(\oplus)[y] \oplus \text{red}(\oplus)[z]) \\
&= x \oplus (y \oplus z)
\end{aligned}$$

5. On va avoir la même structure de réécriture via la définition de red, le seul changement sera l'argument "pivot", où l'on considérera :
   — $\uplus$ l'union commutative de multi-ensembles au lieu de $+\!\!+$
   — $\cup$ l'union idempotente d'ensembles au lieu de $+\!\!+$
6. Par induction sur la structure de liste non vide.

   Base La liste est réduite à un singleton :

$$\begin{aligned}
(g \circ \text{red}(\oplus))[x] &= g(\text{red}(\oplus)[x]) \\
&= g(x) \\
&= \text{red}(\otimes)[g(x)] \\
&= \text{red}(\otimes)(\text{map}(g)[x]) \\
&= (\text{red}(\otimes) \circ \text{map}(g))[x]
\end{aligned}$$

   Induction On suppose que c'est vrai pour la liste xs :

$$
\begin{aligned}
(g \circ \mathsf{red}(\oplus))([x] +\!\!+ xs) &= g(\mathsf{red}(\oplus)([x] +\!\!+ xs)) \\
&= g(\mathsf{red}(\oplus)[x] \oplus \mathsf{red}(\oplus)xs) \\
&= g(\mathsf{red}(\oplus)[x]) \otimes g(\mathsf{red}(\oplus)xs) && \textit{hyp. sur } g \\
&= g(x) \otimes (g \circ \mathsf{red}(\oplus))xs \\
&= \mathsf{red}(\otimes)(\mathsf{map}(g)[x]) \otimes (\mathsf{red}(\otimes) \circ \mathsf{map}(g))xs && \textit{hyp. ind.} \\
&= \mathsf{red}(\otimes)(\mathsf{map}(g)[x]) \otimes \mathsf{red}(\otimes)(\mathsf{map}(g)xs) \\
&= \mathsf{red}(\otimes)(\mathsf{map}(g)[x] +\!\!+ \mathsf{map}(g)xs) \\
&= \mathsf{red}(\otimes)(\mathsf{map}(g)([x] +\!\!+ xs)) \\
&= (\mathsf{red}(\otimes) \circ \mathsf{map}(g))([x] +\!\!+ xs)
\end{aligned}
$$