

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 collectionProposez 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., $REDUCE(w, [1, 2, 3, 4]) = (w, 2.5)$ 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 correspondant aux requêtes suivantes. Pour la recherche dans un tableau json, on utilisera la fonction `Array.indexOf(x)` qui renvoie le premier index où x est dans `Array` et `-1` sinon.

1.

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

2. `SELECT DISTINCT A, B
FROM R`

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

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

et

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

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

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 $g : (K_1 \times V_1) \rightarrow [(K_2 \times V_2)]$ et $(\oplus) : V_2 \times V_2 \rightarrow V_2$ avec $grp : [(K_2 \times V_2)] \rightarrow [(K_2 \times [V_2])]$ la fonction de regroupement. Le pipeline complet est définissable ainsi :

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

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 . Vérifier que la définition de $\text{mapreduce}(g)(\oplus)$ est correctement typée et que son type est $[(K_1 \times V_1)] \rightarrow [(K_2 \times V_2)]$.
2. 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* ;

3. Montrer que la loi \oplus doit être associative, on utilisera pour cela la définition de $\text{red}(\oplus)$ et l'associativité de $\#$.
4. 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.
5. 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.

Corrections

Solution de l'exercice 1

- **Doc1** et **Doc2** : [(*appreciate*, 1), (*the*, 1), (*fold*, 1), (*cata*, 1), (*equals*, 1), (*fold*, 1)]
— **Doc3** : [(*fold*, 1), (*the*, 1), (*fold*, 1)]
- 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 :
 - 1^{er} REDUCER
 - entrée [(*appreciate*, 1), (*cata*, 1), (*equals*, 1)]
 - sortie [(*appreciate*, 1), (*cata*, 1), (*equals*, 1)]
 - 2^e 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.

- 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 ← new ASSOCIATIVEARRAY
    for all term t ∈ doc d do
      H{t} ← H{t} + 1
    end for
    for all term t ∈ 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]);
  }
};
```

- 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)]
- On obtient

```

1: class MAPPER
2:   method INITIALIZE
3:      $H \leftarrow \text{new ASSOCIATIVEARRAY}$ 
4:   end method
5:   method MAP(docid  $a$ , doc  $d$ )
6:     for all term  $t \in \text{doc } d$  do
7:        $H\{t\} \leftarrow H\{t\} + 1$ 
8:     end for
9:   end method
10:  method CLOSE
11:    for all term  $t \in H$  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 [r1, r2, ...])
3:     sum ← 0
4:     cnt ← 0
5:     for all integer r ∈ integers [r1, r2, ...] do
6:       sum ← sum + r
7:       cnt ← cnt + 1
8:     end for
9:     ravg ← sum/cnt
10:    EMIT(string t, integer ravg)
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;
};

```

- 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

- 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
};
```

2. Là, on va utiliser le regroupement sur les clefs pour éliminer les doublons : la clef est simplement la paire (A, B) .

```
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;
};
```

3. 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) apparait 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 mapb = 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 reduceb = function(key, values) {
  //si on est bien dans R ET dans S
  var r = {R : false, S : false};
  for (var i = 0, n = values.length; i < n; i++){
    r.R = r.R || values[i].R;
    r.S = r.S || values[i].S;
  };

  return (r);
};
```

```
//db.sql.mapReduce(mapb, reduceb, {out : { inline:1}}).results.filter(
  function(x){return (x.value.R && x.value.S)});
```

4. 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 2, voir le code ci-après. Pour le cas avec le ALL, on fait plutôt comme pour la question 1.

```
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;
};
```

5. Pas grand chose à signaler ici, c'est un cas d'application paradigmatique de Map/Reduce.

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

6. Avec un filtrage final comme pour l'intersection. On sépare le groupement dans le reduce du produit cartésien dans le finalize à cause des re-reduce. Même problème qu'une question précédente : 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 avec `results.filter(function(x) { return (x.value.valJ.length > 0)});`.

```
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) {
  //on agrege les valeurs de R et de S separement dans la "relation" J
  var res = {rel: "J", valR : [], valS : []};

  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
  // - un 'J' avec une liste dans valR et une autre dans valS
  // on ne peut avoir que :
  // - 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.
    for (var i = 0, n = value.valR.length; i < n ; i++){
      var o = value.valR[i];
      for (var j = 0, p = value.valS.length; j < p ; j++){
        o.Y = value.valS[j].Y;
        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. On peut mettre les identifiants, voire même copier les valeurs.

Solution de l'exercice 4

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

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

2. 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 $(\oplus) = (+)$
 - $g = (k_1, v_1) \mapsto (*, 1)$ et $(\oplus) = (+)$ où $*$ est un symbole arbitraire
 - on va utiliser deux jobs : un premier qui assure l'unicité et ensuite un second qui compte :
 - $g_1 = (k_1, v_1) \mapsto (v_1, *)$ et $(\oplus_1) = (*, *) \mapsto *$;
 - $g_2 = (v_1, *) \mapsto (*, 1)$ et $(\oplus_2) = (+)$
3. Il faut rappeler que la concaténation des chaînes est associative, tout le reste est basé sur la

définition de $\text{red}(\oplus)$:

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

4. 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 $++$
5. 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 :

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

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.