Traîtement distribué des données

Master 1 ISD

Données massives

MapReduce

Kim.Nguyen@lri.fr

Plan

SQL

Le standard SQL spécifie un grand nombre d'aspects des bases de données relationnelles :

- Un langage de requête (`SELECT …`)
- Un langage de modification de données (`CREATE …`, `UPDATE …`)
- Un langage de schéma et de contraintes
- Un langage de programmation (SQL/PSM)
- Un modèle de concurrence (transactions, `BEGIN … END`, `COMMIT`, `ROLLBACK`)

Garanties

Le modèle relationnel et les bases de données SQL offrent des garanties très fortes :

- Atomicity (Atomicité) : chaque transaction est du « tout ou rien ». Il est impossible pour quelqu'un en dehors de la transaction d'observer une étape intermédiaire.
- Consistency (Cohérence) : chaque transaction fait passer la base d'un état cohérent à un nouvel état cohérent. Toute donnée sauvegardée dans la base doit être valide vis à vis d'un ensemble de règles (types, contraintes, triggers, ...).
- Isolation : l'exécution concurrente de deux transactions $\mathbf{T_1}$ et $\mathbf{T_2}$ doit être la même que leur exécution séquentielle.
- Durability (Durabilité) : une transaction confirmée (`COMMIT;`) le reste même en cas de problèmes (perte de courant, erreur interne, ...).

Quel impact ?

Maintenir un modèle ACID a un impact énorme sur les performances (création de *snapshots*, mise en place de verrous coûteux, attente de confirmation par le disque que les données sont sauvées, ...).

Comment augmenter les performances et le volume des données ?

- Prendre un serveur plus puissant (ex. Oracle Exadata de 40 000 à 1 300 000 USD + souscription mensuelle, entretien, ...)
- Mettre plusieurs serveurs ? Très difficile, pourquoi ?

Distribution et modèle ACID

Il existe deux modèles de distribution possible :

- Plusieurs serveurs contenant chacun une copie complète de la base :
  - Performances accrues pour les requêtes
  - Diversité géographique possible (plus grande résistance aux évènement extérieurs)
  - Baisse des performances pour les mises à jour
  - Ne permet pas de stocker plus de données que sur un seul serveur
- Partitionnement des données. Une relation $\mathbf{R}$ peut être découpés en morceaux $\mathbf{R}_1$, ..., $\mathbf{R}_1$, disposés sur des machines différentes. Le découpage peut se faire par colonnes ou par lignes.
  - Augmentation de la capacité de stockage globale
  - Jointures très coûteuses si deux colonnes sont sur deux machines différentes
  - Agrégats coûteux si les lignes sont sur deux machines différentes
  - Une machine en panne ou inaccessible compromet tout le système

Le modèle ACID est-il toujours adapté ?

Le modèle ACID est parfois trop contraignant :

- Système de réservation de billets d'avion ? Modèle ACID nécessaire
- Calcul du nombre de messages envoyé par utilisateurs de chaque tranche d'âge, sur Facebook ? Modèle ACID inutile

Y a t-il un modèle alternatif ?

Théorème de Brewer

Le théorème (énoncé comme une conjecture en 1999 par E. Brewer et prouvé en 2004) dit que dans un système distribué partageant des données, au plus 2 des 3 propriétés suivantes peuvent être optimale :

- Consistency (cohérence) : tous les nœuds de calcul voient les même données au même moment
- Availability (disponibilité) : chaque requête reçoit une réponse de succès ou d'échec
- Partition tolerence (tolérance aux pannes) : le système continue de fonctionner correctement malgré un découpage arbitraire lié à des pannes réseau

Le théorème en lui-même est souvent mal interprété. Il a surtout été énoncé (d'après Brewer) pour inciter les gens à explorer d'autres modèles/compromis.

Modèle BASE

Le modèle BASE est régit par les propriétés suivantes :

- **B**asically **A**vailable (toujours disponible) : le système s'efforce de répondre à une requête, quel que soit son état
- **S**oft state (état « mou ») : l'état interne du système peut changer sans intervention d'un utilisateur
- **E**ventually consistent (cohérent au bout d'un temps fini) : si on laisse passer suffisamment de temps entre deux mises à jour ou pannes, le système converge vers un état cohérent

Exemple concret

On considère un système avec deux nœuds $\mathbf{N_1}$ et $\mathbf{N_2}$:

- Mise à jour $\mathbf{M}$ suivi de deux fois la même requête $\mathbf{Q}$:
  - ACID : si $\mathbf{M}$ est dans une transaction, alors $\mathbf{Q}$ s'effectue strictement après $\mathbf{M}$ et les deux instances de $\mathbf{Q}$ renvoient le même résultat
  - BASE : les deux instances de $\mathbf{Q}$ peuvent renvoyer des résultats différents, $\mathbf{M}$ peut s'exécuter en même temps que $\mathbf{Q}$ et ne même pas être fini.
- $\mathbf{N_2}$ tombe en panne :
  - ACID : le système entier est en panne
  - BASE : $\mathbf{N_1}$ continue de répondre

Que peut on faire avec BASE ?

Le modèle BASE permet de créer des systèmes composés de plusieurs nœuds, capable de stocker des données indépendemment et de répondre rapidement. Comment le mettre à profit ?

Plan

Programmation fonctionnelle, le retour

Un peu de code OCaml pour se détendre

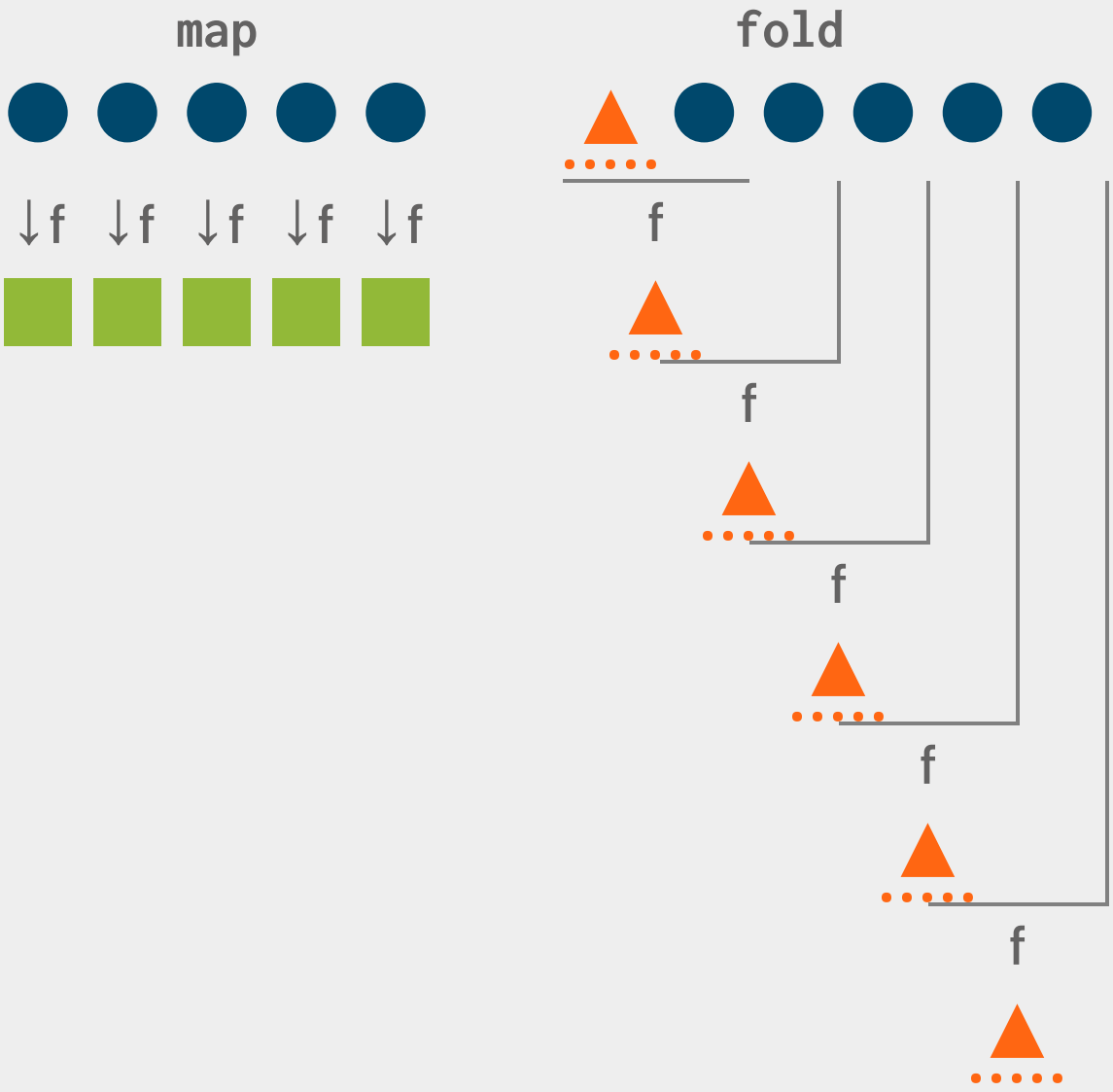
```
let words = [ "It'"; "s";  "a"; "beautifull"; "day" ]

(* On veut compter la longueur totale des chaînes de caractères
   dans words *)

let lengths = List.map (fun s -> String.length s) words

(* lengths = [ 3; 1; 1; 10; 3 ] *)

let total = List.fold_left (fun acc i -> i + acc) 0 lengths

(* total = 18 *)
```

Les itérateurs `map` et `fold`

`map :` de type $(\alpha \to \beta) \to \alpha$ `list` $\to \beta$ `list` , prend en argument une fonction et une liste d'éléments et applique la fonction à chaque élément. Elle renvoie la liste des éléments transformés (dans le même ordre).

`fold :` de type $(\alpha \to \beta \to \alpha) \to \alpha \to \beta$ `list` $\to \alpha$, prend une fonction d'agrégat, un accumulateur initial et une liste d'élément qui sont passés tour à tour à la fonction. La valeur finale de l'accumulateur est renvoyée.

Visuellement



Propriétés de `map` et `fold` ?

- Les collections de départ sont inchangées
- les applications de la transformation par `map` sont indépendantes, leur ordre d'application n'est pas important
- pour `fold`, l'ordre n'est pas important si l'agrégateur est associatif et commutatif

Ces observations sont à la base de MapReduce

MapReduce

Popularisé par Jeffrey Dean et Sanjay Ghemawat dans l'article :
MapReduce: Simplified Data Processing on Large Clusters (OSDI, 2004)

Remarque: le paradigme n'est pas nouveau (BD distribuées, langages fonctionnels), mais l'article l'a popularisé et a permis l'arrivée d'implémentations Open-Source robustes.

Cadre : on dispose d'un grand nombre de machines, dont chacune dispose localement d'un ensemble de données (distinct des autres). Un nœud particulier joue le rôle d'orchestrateur les autres sont des travailleurs (workers).

Une transformation MapReduce se décompose en trois phases :

1. Phase Map
2. Phase Shuffle
3. Phase Reduce (fold)

Préambule

Le programmeur fournit deux transformations :

```
map(InputKey k, InputValue v) → (OutputKey * IntermediateValue list)

reduce(OutputKey, IntermediateValue list) → OutputValue list
```

Phase Map

Lors de la phase Map, l'orchestrateur exécute une copie de la transformation map sur chaque worker. Chacun ne transforme que les données qu'il possède localement.

La fonction map reçoit ses données d'entrées sous la forme d'une paire clé, valeur (par exemple (nom de fichier, fichier) ou (id, ligne correspondant à l'id dans une table)).

La fonction renvoie comme résultat une liste de valeur transformées associé à une clé de groupe

Phase Shuffle

Lorsque la phase Map est terminée sur tous les nœuds, les données sont échangées entre nœuds et groupées selon la clé de groupe

Cette opération est une barrière, elle ne peut se produire qu'après la fin de la phase Map. De plus elle nécessite l'échange de données sur le réseau (coûteux).

Phase Reduce

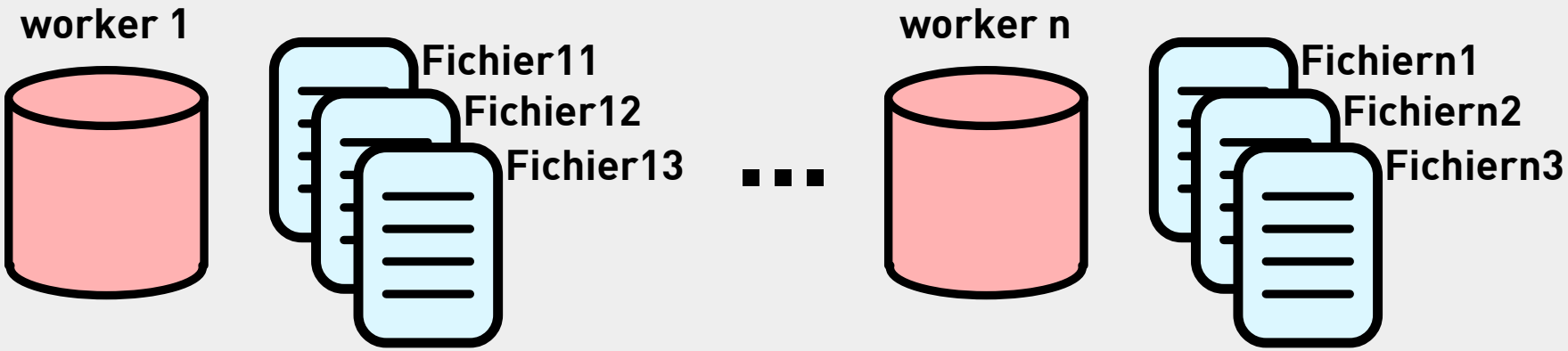
L'orchestrateur exécute une copie de la fonction reduce sur chaque nœud. Cette fonction reçoit en argument une clé de groupe et la liste de toutes les valeurs intermédiaires associées, et produit un résultat final par clé.

La liste de ces résultats (pour chaque clé) est renvoyée au programmeur ou stockée sur les nœuds pour être réutilisée dans un nouveau cycle `Map/Shuffle/Reduce`

Exemple complet : `word count`

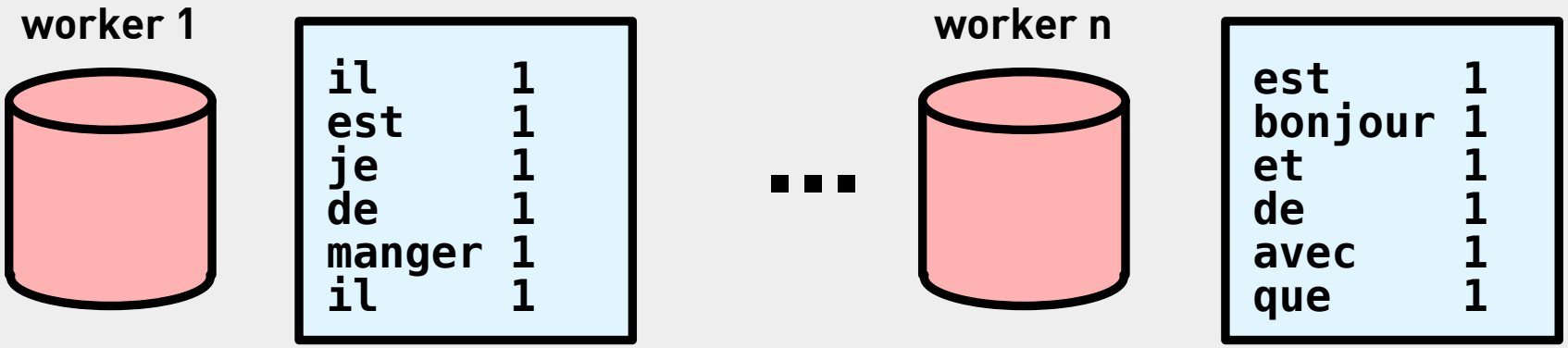
On suppose stocké sur les nœuds des ensembles de fichiers, auxquels on peut accéder sous forme de paires :

`(nom de fichier, contenu du fichier)`

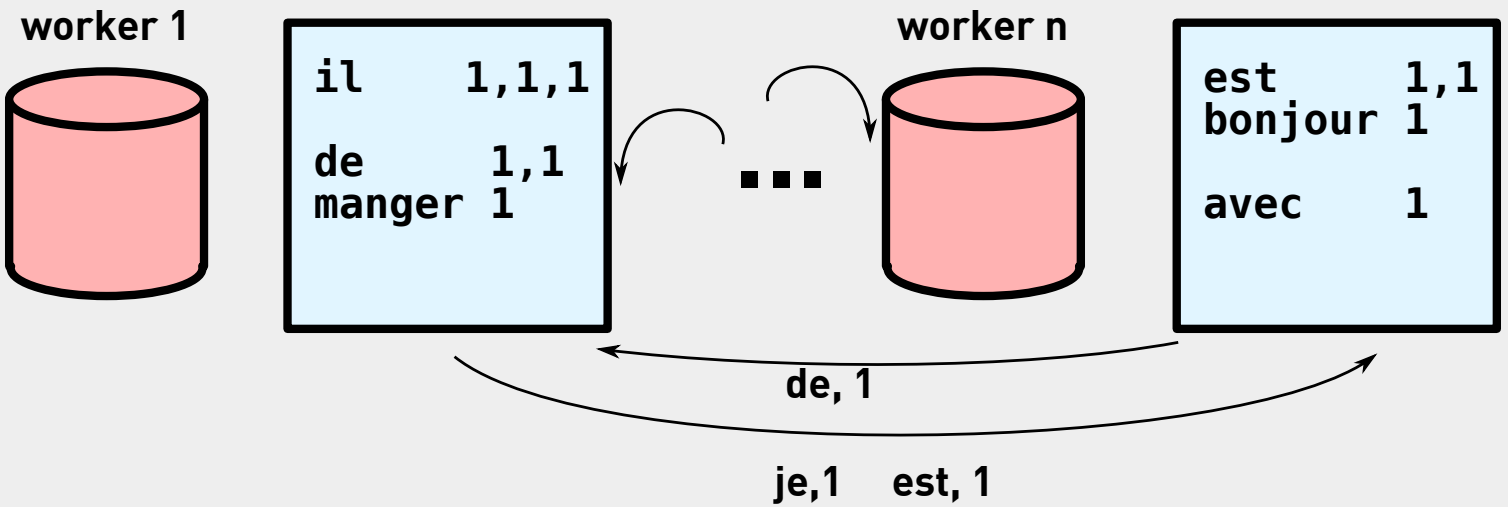


Exemple complet : word count (Map)

```
map(InputKey file, InputValue content) {

for each word in content {
          Output(word, 1);
 }
}
```

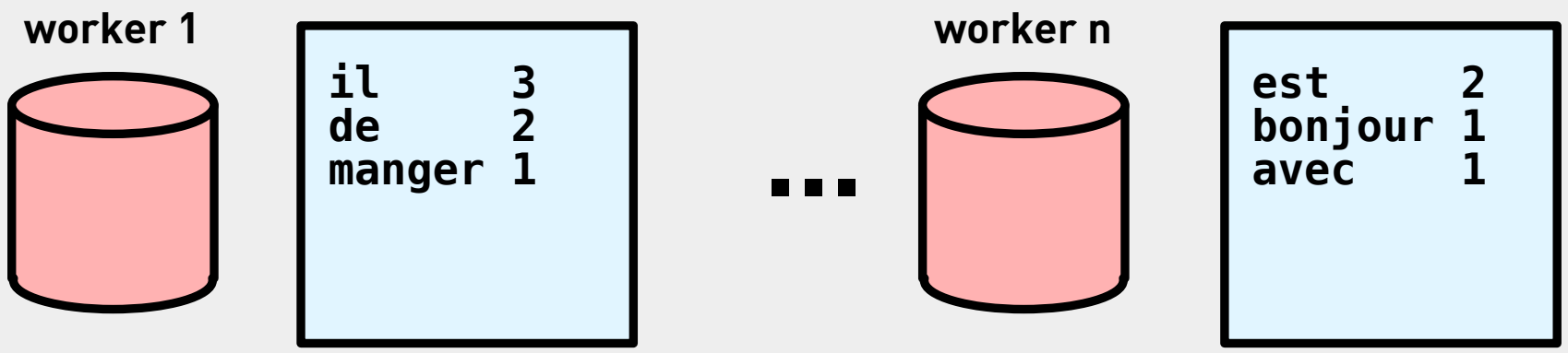


Exemple complet : `word count` (Shuffle)



Exemple complet : word count (Map)

```
reduce(OutputKey word, InputermediateValue list ones) {

        int total = 0;
        for i in ones {
          total += i;
        }
        return total;
    }
```



Parallélisme et modèle de coût

Chaque nœud travaille en parallèle lors des phases `Map` et `Reduce`.

Un nœud plus lent retarde tout le monde (en pratique, le même calcul peut être lancé de manière redondante sur plusieurs nœuds et le résultat du premier qui termine est gardé)

La phase de `Shuffle` est coûteuse en accès réseaux, le modèle de coût est différent des BD relationnelles (on va chercher à minimiser les échanges réseaux)

Optimisation : si les fonctions de réduction est associative et commutative, on peut appliquer `Reduce` localement avant la phase de `Shuffle` et échanger des paquets de réduits.

Résistance aux pannes

- Les données sont répliquées (par défaut en 3 exemplaires)
- Si un nœud est défaillant, l'orchestrateur relance le calcul sur un autre nœud qui possède les données
- Des mécanismes de redondance existe pour remplacer l'orchestrateur si ce dernier devient défaillant (coûteux, implique une re-synchronisation de tout le monde)

Passage à l'échelle

Le paradigme MapReduce est destiné aux très gros volumes de données. Une BD relationnelle sera toujours plus rapide, qu'un seul nœud. Pour tirer partie du parallélisme il faut un jeu de données inexploitable sur un machine unique (de l'ordre du tera-octet ou peta-octet).

Expressivité (digression en SQL)

On suppose que l'on dispose d'une table `Text` possédant une unique colonne `VARCHAR(100) word`.
Donner une requête SQL qui simule `WordCount` :

```
SELECT word, SUM(num) FROM
  (SELECT word, 1 AS num
      FROM WORD) AS T
GROUP BY word;
```

Quelles autres fonctionnalité de SQL peut on facilement supporter ?

- Toutes les fonctions d'agrégat et même plus (`COUNT`, `AVG`, ...)
- Toutes les comparaisons d'un attribut avec une constante ou un autre attribut de la même ligne

Fonctionnalités non supportées par le modèle

- Jointures

Pourquoi ?

Rappel du pseudo-code de la jointure naturelle entre deux attributs `a` et `b` de deux tables `A` et `B` :

```
for each row r in A
do
   for each row s in B
   do
        if r.a == s.b then output r ⋈ s
   done
done
```

On doit comparer chaque ligne de `A`, à toutes les lignes de `B`. Dans un contexte distribué, une grande partie des lignes de `B` n'est pas locale
$\Longrightarrow$ trop d'échanges sur le réseau !

Quelles solutions ?

- jointures calculées en mémoire par le programme principal (souvent de manière naïve, coûteux en temps et en mémoire)
- stockage de données dé-normalisées

Les jointures permettent de relier des relations entre elles (via des clés primaires/étrangères). On élimine ces jointures en stockant des données de manière dupliquées:

```
MOVIE(INTEGER mid, VARCHAR(30) title);
PEOPLE(INTEGER pid, VARCHAR(30) name);
DIRECTOR(INTEGER pid REFERENCES PEOPLE, INTEGER mid REFERENCES MOVIE);
```

devient :

```
MOVIE(INTEGER mid, VARCHAR(30) title, List of (INTEGER,VARCHAR(30)) directors)
PEOPLE(INTEGER pid, VARCHAR(30) name);
```

Note : comme on n'est pas en SQL, on n'est plus contraint par le modèle relationnel « plat ».

Conclusions sur MapReduce

- Paradigme de programmation adapté au traitement de volume de données massifs
- Structure figée en 3 phases (dont deux seulement programmables)
- Si le calcul a effectuer s'exprime sous cette forme, cela marche plutôt bien, les implémentations modernes gèrent la reprise sur panne, la réplication, ...
- Ce n'est pas la solution ultime cependant (dé-normalisation et donc mises à jour plus coûteuses)

Plan

Le *framework* Hadoop

Apache Hadoop a initialement été développé en interne à Yahoo! en 2005. C'est un projet OpenSource, comprenant :

Hadoop Common : ensemble de bibliothèques partagées par le reste de la pile d'application

Hadoop Distributed File System (HDFS) : système de fichiers distribués permettant de stocker des gros volumes de données sur des *clusters* de machines modestes (comodity servers)

Hadoop YARN : ordonnanceur de tâches, gestion de la reprise sur panne, ...

Hadoop MapReduce : bibliothèques permettant de programmer des tâches MapReduce

Utilisation

Hadoop est installable facilement, peut s'exécuter sans privilège particulier et se programme principalement en Java (mais des interfaces existent pour d'autres langages)

Les nœuds communiquent entre eux par des requêtes HTTP ou *over* ssh.

HDFS

HDFS est un système de fichiers distribués. C'est par lui que transitent les données des tâches MapReduce d'Hadoop. Ces caractéristiques sont :

- support pour de très grands fichier (peta-octet)
- support pour un très grand nombre de fichiers (pas de limite a priori)
- distribution sur un grand nombre de machines
- gestion automatique de la réplication
- optimisé pour les lectures séquentielles de fichier
- immuable !

Encore un peu d'OCaml

```
let words = [ "It'"; "s";  "a"; "beautifull"; "day" ]

let words2 = words @ [ ","; "today";  "!" ]
```

La liste words n'est pas modifié par la concaténation (à l'inverse de `Collection.addAll()` en Java qui modifie en place l'argument)

Si un programme (ou plutôt un *thread*) travaille en parallèle sur words, la concaténation ne le gène pas (en Java $\Longrightarrow$ ConcurrentAccesException)

On dit que le type liste est immuable : on ne peut pas modifier le contenu d'une liste une fois crée, on peut juste créer des copies de celle-ci.

HDFS

HDFS est un système de fichier très particulier. Il suppose que les fichiers vont être lus séquentiellement ne permet pas de les modifier (on peut par contre supprimer tout un fichier)

Les fichiers sont découpés en gros blocs (typiquement 64 ou 128 Mo), à contraster avec les blocs d'un disque dur (entre 512 et 4096 octets) et les pages d'une base de données (de 8Ko à 1Mo).

L'accès direct au milieu d'un fichier est relativement coûteux (presque autant que lire le fichier jusqu'à la position donnée)

Une tâche MapReduce a la garantie que les données persistent au moins jusqu'à la fin de la tâche.

Problèmes d'Hadoop MapReduce + HDFS

Architecture logicielle très bas niveau :

- Les tâches Map/Reduce doîvent être programmées en Java
- Le découpage d'une opération complexe en suites de Map et Reduce n'est pas trivial, surtout si l'on veut éviter de tout faire en mémoire, côté client
- La gestion des données est rudimentaires (ajout/suppression de fichiers sous HDFS), pas de gestion de concurrence (pas de transactions)

L'API MapReduce d'Hadoop :

- Une classe `org.apache.hadoop.mapreduce.Mapper<IK, IV, OK, OV>` à étendre, en particulier la méthode `public void map(IK ik, IV iv, Context ctx)`.
- Une classe `org.apache.hadoop.mapreduce.Reducer<IK, IV, OK, OV>` à étendre, en particulier la méthode `public void reduce(IK ik, Iterable<IV> iv, Context ctx)`.
- L'objet de type `Context` permet d'accéder à la configuration (options, ...) et possède une méthode `write(OK ok, OV ov)` pour écrire les résultats.
- Des classes pour les types de base `IntWritable`, `DoubleWritable`, `Text`, ... Les objets de cette classe sont efficacement sérialisable (pas comme `java.lang.Serializable`).
- Du *boiler plate* pour configurer et lancer le tout.

Démo

On va effectuer les manipulations suivantes :

- Initialiser et formatter HDFS
- Lancer Hadoop (sur un nœud)
- Créer des sous-répertoire et importer un fichier texte dedans
- Écrire un programme comptant le nombre d'occurrences de chaque sous-chaîne de taille n dans un fichier texte donné
- Exécuter le programme

Plan

Une interface haut niveau pour Hadoop

Hadoop, HDFS et MapReduce proposent des briques de bases permettant de traîter des gros volumes de données

Ce sont des outils très bas-niveau, on perd le côté déclaratif des bases de données relationnelles (ou XML) alors qu'on voulait uniquement relâcher un peu les contraintes du modèle ACID

Hive

Apache Hive est un système d'entrepôt de données (data-warehouse system), i.e. une base de données spécialisée pour le *reporting* et l'analyse de données

Il a été développé en interne par Facebook avant d'être diffusé sous licence OpenSource

Initialement construit au dessus de Hadoop/HDFS/MapReduce, il supporte maintenant d'autres backends

Avantages de Hive

- Fournit une abstraction de haut niveau au dessus de MapReduce/HDFS
- Propose un langage de requête et mise à jour, HiveQL très proche de SQL (avec jointures et tri !)
- Support pour certains types d'indexes
- Supporte des mises à jours de ligne et colonnes (`UPDATE` et `ALTER TABLE`)
- Schémas de table et certaines contraintes
- Support (limité) de transactions de type ACID
- Possibilité d'utiliser des UDF (*User Defined Functions*) écrite directement en Java.

Gestion des mises à jour

Les tables HiveQL sont stockées comme des fichiers dans HDFS.

Que se passe-t-il quand on insère ou supprime une seule ligne ?

Un delta entre la table originale et la table modifiée est stocké (dans un nouveau fichier). Lors du chargement d'une portion de table, les delta sont appliqués pour reconstituer la table courante.

Un garbage collector compacte les tables et leur delta quand suffisamment de delta se sont accumulés.

Optimiseur de requête

Dans Hive, l'optimiseur de requête ne cherche pas le plan optimal pour les jointures mais cherche à exprimer la requête sous forme d'un arbre (ou plus exactement d'un DAG) dont les nœuds sont des MapReduce.

Le modèle de coût utilisé est complexe, et fait intervenir entre autres les coût des communications réseau

Démo

On va effectuer les manipulations suivantes :

- Démarrer Hive (Hadoop étant déjà démarré)
- Importer des fichiers représentant une base de donnée de films
- Écrire des requêtes et analyser leur plan d'exécution
- (Comparer avec un plan SQL)

Plan

Java est enfin un langage fonctionnel

Java 8 apporte des traits fonctionnels au langage. Intérêt de la programmation fonctionnelle :

- Code souvent plus compact
- Meilleure réutilisation du code
- Code potentiellement plus efficace
- Code souvent plus lisible (si on en a l'habitude)

Trois ingrédients :

Les interfaces fonctionnelles : Une manière de déclarer le type de fonctions
Les lambdas-abstraction : Une manière de définir des petites fonctions à la volée
La *Stream API* : Une bibliothèque d'itérateurs efficaces qui vient s'ajouter aux `Collections`

Interfaces fonctionnelles

Une interface fonctionnelle est simplement une `interface` contenant une seule méthode abstraite. Cette utilisation permet de définir des types de « fonctions ».

`java.util.function.Function<X,Y>` : représente le type d'une fonction de `X` vers `Y`

`java.util.function.Predicate<X>` : représente le type d'une fonction de `X` vers `boolean`

`java.util.function.Consumer<X>` : représente le type d'une fonction de `X` vers `void`

Exemple

```
  class MyIncr implements Function<Integer,Integer> {
  //Doit s'appeler apply

  Integer apply(Integer x) { return x + 1; }
}


  class MyOdd implements Predicate<Integer> {
  //Doit s'appler test

  boolean test(Integer x) { return x % 2 == 1; }
}

  class MyPrint implements Consumer<Integer> {
  //Doit s'appler accept

  void accept(Integer x) { System.out.println(x); }
}
```

Stream API

Nouvelle API de Java 8. Permet de définir de manière déclarative des itérateurs sur des flux. Étant donné un objet qui implémente l'(ancienne) interface `Collection<E>` (par exemple `Vector<E>`, `List<E>`, `Set<E>`) :

```
Vector<Integer> v = …;
  …
Stream<Integer> s = v.stream();
```

Que gagne t'on ?

On va pouvoir définir des itérateurs persistants sur les flux, paresseux et parallélisable

Méthodes de l'interface `Stream<T>`

`.count()` : renvoie le nombre d'éléments dans le flux

`.skip(long n)` : renvoie un flux où `n` éléments ont été sautés

`.limit(long n)` : renvoie un flux avec uniquement les `n` premiers éléments

`.distinct()` : renvoie un flux où les doublons (au sens de `.equals(Object o)` sont retirés

`.sorted()` : renvoie un flux où les éléments (qui doivent implémenter `Comparable`) son triés

Quelle différence avec les opérateurs sur les Collections ?

Les collections sous-jacentes ne sont pas détruites!

Méthodes de l'interface `Stream<T>` (suite)

`.filter(Predicate< ? super T> p)` : renvoie le flux de tous les éléments pour lesquels le prédicat `p` renvoie vrai (`p` peut être défini sur un super-type de T)

`.allMatch(Predicate< ? super T> p)` : renvoie vrai si tous les éléments satisfont le prédicat `p`

`.anyMatch(Predicate< ? super T> p)` : renvoie vrai si un élément satisfait le prédicat `p`

`.map(Function< ? super T, ? extends R> f)` : renvoie le flux d'éléments de type `R` obtenu par application de `f` sur tous les éléments du flux de départ.

`.forEach(Consumer< ? super T> c)` : applique `c` tour à tour à tous les éléments du flux.

`.collect(…)` : transforme le flux en Collection. On peut utiliser `Collectors.toList()`, `Collectors.toSet()`, ...

Exemple

Supposons que l'on a un vecteur d'`Integer v`. On souhaite, tous les incrémenter, les trier, puis afficher ceux qui sont impairs :

```
Vector<Integer> v = … ;
v.stream().map(new MyIncr())
          .sorted()
          .filter(new MyOdd())
          .forEach(new MyPrint());
```

Exercice, faire la même chose sans passer par les flux

λ-abstraction

Il est malcommode de définir les opérations à passer aux itérateurs dans des classes (`MyIncr`, ..., dans l'exemple précédent). Java 8 fournit une syntaxe commode pour définir des fonctions :

```
Vector<Integer> v = … ;
v.stream().map(i -> i+1)
          .sorted()
          .filter(i -> i % 2 == 1)
          .forEach(i -> System.out.println(i));
```

La syntaxe générale est :

$(\text{Type}_1\ p_1, \ldots, \text{Type}_n\ p_n)$ -> *corps*

Le corps peut être soit une expression simple soit un bloc :

```
(Integer i) -> i+1
(Integer i) -> {
        System.out.println(i);
        return i+1;
}
```

λ-abstraction (suite)

Avantage des λ-abstractions ?

- Syntaxe compacte mais lisible
- Localité de la définition (on n'est pas obligé de mettre le code dans une classe ailleurs)
- Accès à l'environnement local (à l'inverse des classes anonymes)

```
PrintWriter out = …;
…
v.stream().forEach(i -> out.println(i) );
```

Que gagne t'on à utiliser la Stream API ?

- Persistance : les collections sous-jacentes ne sont pas détruites/modifiées, pas besoin de faire des copies
- Paresse : les opérations ne sont faites que si c'est nécessaire :

```
Stream<Integer> vs = v.stream().sorted();
if (do_print)
      vs.forEach(i -> System.out(i));
//le tri n'est fait que si on force (en utilisant .forEach())
```

- Les opérations peuvent se faire le plus possible en parallèle, si on utilise `.parallelStream()` au lieu de `.stream()` pour récupérer le flux
- implémente le même paradigme que Map/Reduce