Ingénierie de la fouille et de la visualisation de données massives (RCP216)

Introduction

Michel Crucianu
(prenom.nom@cnam.fr)
http ://cedric.cnam.fr/vertigo/Cours/RCP216/

Département Informatique
Conservatoire National des Arts & Métiers, Paris, France

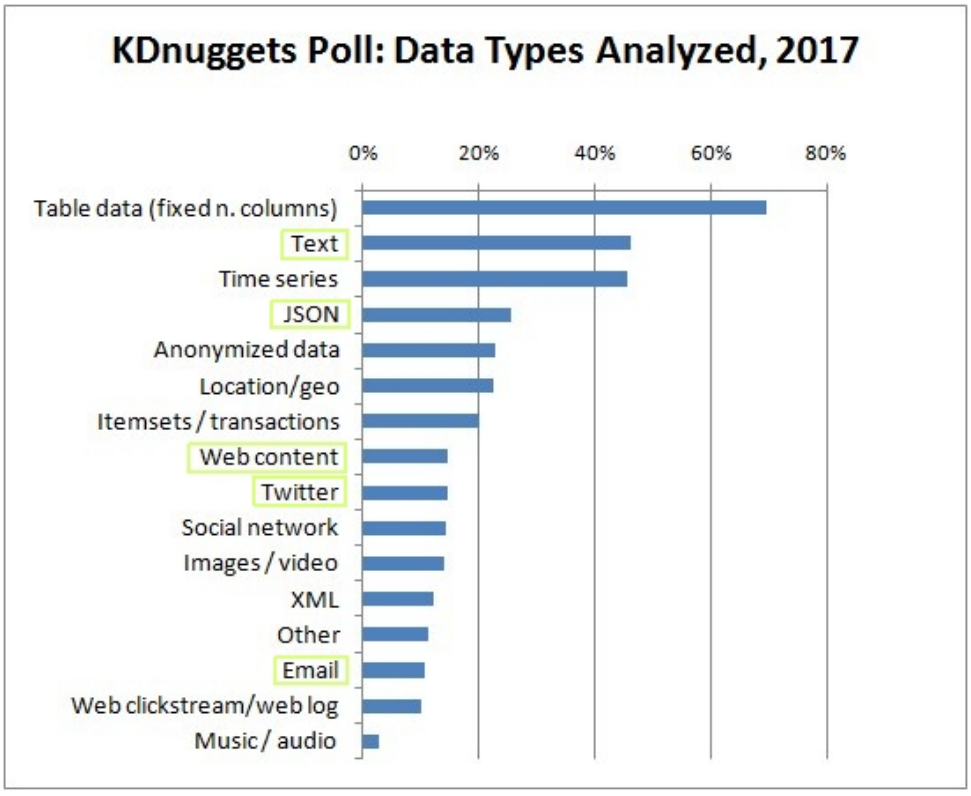
8 février 2022

Plan du cours

Le sujet

- Fouille de données : "*Process of nontrivial extraction of implicit, previously unknown and potentially useful information (such as knowledge rules, constraints, regularities) from data in databases*" (G. Piatetski-Shapiro)
- Données massives : volume peu compatible avec un traitement centralisé
  - Autres caractéristiques : variété (données peu structurées, hétérogènes), vélocité (flux à traiter en temps réel)
  - En général à « faible densité en information »
- Si l'échelle est souvent le problème dominant, les autres caractéristiques posent aussi des difficultés significatives
- Échelle ⇒ nécessité de faire « passer à l'échelle » les méthodes de fouille de données
- Faible densité en information ⇒ rôle renforcé de la visualisation, fouille interactive

Types de données traitées (source http ://www.kdnuggets.com 2017)



Plan du cours

Objectifs de l'enseignement

- Fouille de données massives :
  - Faire comprendre les approches de passage à l'échelle
  - Présenter les méthodes de résolution de certains problèmes fréquents
  - Familiariser avec des logiciels permettant l'exécution distribuée
  - Faire appliquer ces méthodes sur des données réelles
- Visualisation et interaction :
  - Faire comprendre les principes et les enjeux
  - Présenter des techniques usuelles et leurs utilisations
  - Familiariser avec la programmation de ces techniques

$\Rightarrow$ Capacité à mettre œuvre, sur des données massives, des techniques de fouille de données et de visualisation interactive

Contenu du cours (15 séances)

- Approches du passage à l'échelle : réduction de la complexité, distribution
- Analyse de données et réduction de dimension - ACP, AFD, ACM
- Recherche par similarité, systèmes de recommandation
- Classification automatique
- Fouille de séquences et de flux de données
- Fouille de textes : filtrage, représentation, analyse
- Apprentissage statistique à large échelle
- Aspects éthiques dans la fouille de données massives
- Introduction aux graphes et réseaux sociaux
- Graphes : centralités, modèles, détection de communautés, dynamique
- Introduction à la visualisation d'information
- Visualisation de graphes

Contenu des travaux pratiques

- Introduction à Spark et Scala
- Manipulation de données numériques
- Échantillonnage
- Analyse en composantes principales
- Classification automatique
- Traitement de flux de données
- Fouille de données textuelles
- SVM linéaires
- Fouille de graphes et réseaux sociaux
- Introduction à la visualisation
- Visualisation de graphes

Plan du cours

Prérequis

- En mathématiques : bonnes connaissances mathématiques et statistiques générales, maîtrise de méthodes statistiques pour la fouille de données
- En informatique : connaissances élémentaires en programmation, connaissance de techniques de gestion de données massives faiblement structurées, de passage à l'échelle par distribution
- Questionnaire d'auto-évaluation anonyme : `http://cedric.cnam.fr/vertigo/Cours/RCP216/questionnaire.html`

Dans le cadre du certificat de spécialisation « Analyste de données massives », ces prérequis se traduisent, en partie, par l'exigence d'avoir suivi NFE204 et STA211 avant de commencer RCP216

Organisation

- Enseignants : Michel Crucianu, Raphaël Fournier-S'niehotta, Nicolas Audebert
- Horaires S1 (http ://emploi-du-temps.cnam.fr)
  - Cours : mardi 17h30-19h30
  - Travaux pratiques : mardi 19h45-21h45
- Horaires S2 (http ://emploi-du-temps.cnam.fr)
  - Cours : mardi 17h30-19h30
  - Travaux pratiques : mardi 19h45-21h45
- Supports : http ://cedric.cnam.fr/vertigo/Cours/RCP216/
  - Cours : transparents en ligne pour tout le cours, explications détaillées en HTML pour la partie fouille, vidéos sur Moodle (http ://lecnam.net) pour la partie visualisation
  - Travaux pratiques : contenu détaillé en HTML, corrigés mis en ligne env. 2 semaines après la séance
- Forum informel sur Moodle (http ://lecnam.net, entrer dans RCP216)

Evaluation

- Examen sur table
  - S1 : examen en février avec rattrapage en avril
  - S2 : examen en juin avec rattrapage en septembre
  - Planification des examens : http ://www.cnam-paris.fr/suivre-ma-scolarite/ rubrique Examens
  - les notes **ne sont pas** transmises par téléphone ou courriel (instructions Cnam)
- Projet : appliquer des techniques de fouille (**avec Spark**) et visualisation sur des données réelles
  - sujets proposés par auditeurs ou enseignants, env. 2 mois après le démarrage du cours
  - 2 dates limite chaque semestre, synchronisées avec l'examen principal et respectivement l'examen de rattrapage
- Note finale = moyenne entre note de projet et note d'examen

Plan du cours

Références bibliographiques

- Fouille de données massives
  - J. Leskovec, A. Rajaraman, J. Ullman. *Mining of Massive Datasets*. Cambridge University Press. http ://www.mmds.org
  - Karau, H., A. Konwinski, P. Wendell et M. Zaharia. *Learning Spark - Lightning-Fast Big Data Analysis*, O'Reilly, 2015.
  - Ryza, S., U. Laserson, S. Owen and J. Wills. *Advanced Analytics with Spark - Patterns for Learning from Data at Scale*, O'Reilly, 2015.
- Visualisation des données
  - B. Fry. *Visualizing data*. O'Reilly, 2008.
  - R. Spence. *Information Visualization, design for interaction*. Prentice Hall, 2007.
- sans oublier, entre autres
  - C. O'Neil, *Weapons of Math Destruction*. Crown, 2016.
  - Solon Barocas, Moritz Hardt, and Arvind Narayanan. Fairness and Machine Learning Limitations and Opportunities. https ://fairmlbook.org

Plan du cours

Le passage à l'échelle

- Vise notamment la capacité à traiter de très gros volumes de données
- Faible densité d'information ⇒ il n'est pas toujours souhaitable d'exclure *a priori* certaines composantes des données
  - Travailler sur un échantillon trop petit : les « régularités » recherchées ne se manifestent pas suffisamment pour être détectables...
  - Travailler sur un nombre de variables trop faible : les « régularités » sont incomplètes, les capacités prédictives sont insuffisantes...
- Peut concerner d'autres aspects : hétérogénéité, nombre de variables, etc.

Le passage à l'échelle : approches complémentaires

- Réduction de la complexité (N : volume des données)
  - Sans changer l'ordre de complexité : réduction de dimension, calculs sur échantillon
  - Réduction de l'ordre de complexité : $O(N) \rightarrow O(\log(N))$ ou $O(cst\ .)$, $O(N^2) \rightarrow O(N \log(N))$
  - Méthodes exactes *ou approximatives* !
- Parallélisation des calculs
  - Architectures parallèles dédiées
    - Rapport coût / performance très élevé (architectures et technologies dédiées)
    - Goulot d'étranglement entre stockage de masse et unités centrales
  - Distribution des calculs sur un grand nombre d'ordinateurs standard (coût bas), avec
    - Minimisation des échanges avec le stockage de masse, tout en assurant la fiabilité aux pannes
    - Minimisation des échanges de données entre ordinateurs

Problématique

- Comment faciliter l'implémentation d'une grande diversité d'algorithmes sur une plateforme distribuée ?
  → Mécanisme d'exécution composé d'opérations simples et génériques
- Comment réduire la fragilité aux pannes, inévitables pour des calculs longs réalisés sur des plateformes comportant de grands nombres d'ordinateurs ?
  → Mécanisme de réplication pour le stockage fiable des données
  → Partitionnement des calculs en « grains » traçables qui peuvent être (ré)affectés à d'autres ordinateurs

MapReduce

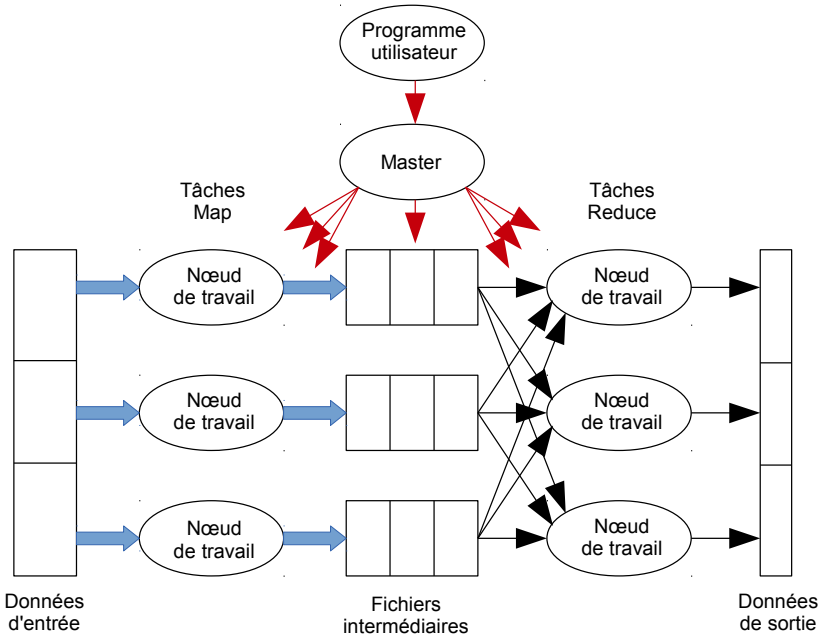
- Mécanisme d'exécution simple
  - Permettant un parallélisme massif
  - Tolérant aux pannes
- Principe : décomposer le calcul en deux types de tâches élémentaires et uniformes, *Map* (exécutées en premier) et *Reduce* (exécutées ensuite), qui peuvent être facilement assignées à des nœuds de calcul
- Autre idée de MapReduce : si les données sont (beaucoup) plus volumineuses que les programmes, stocker localement les données et transférer plutôt les programmes pour les rapprocher des données

MapReduce : fonctionnement général

1. L'ensemble de données à traiter est découpé en *fragments* (*chunks*)
2. Chaque tâche *Map* est assignée à un nœud de calcul qui reçoit un ou plusieurs fragments que la tâche *Map* transforme en une séquence de paires (clé, valeur)
3. Chaque tâche *Reduce* est associée à une ou plusieurs clés et est assignée à un nœud de calcul
4. Les paires (clé, valeur) produites par les *Map* sont groupées par clés et stockées sur les nœuds de calcul qui exécuteront les tâches *Reduce* respectives (étape *shuffle*)
5. Chaque tâche *Reduce* combine, pour chaque clé qui lui est associée, les valeurs des paires (clé, valeur) avec cette clé ; les résultats sont stockés et constituent le résultat du traitement

- Le programmeur écrit les fonctions *Map* et *Reduce*, le *framework* se charge du reste
- Hadoop : implémentation *open source* de *MapReduce*

MapReduce : exécution d'un programme



MapReduce : un exemple classique

Tâche à réaliser : compter le nombre d'occurrences de chaque mot dans une collection de documents textuels

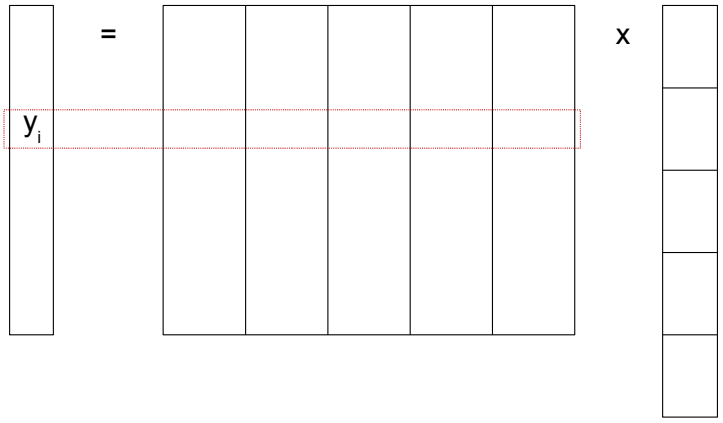
1. On considère, pour simplicité, que chaque document constitue un fragment
2. Chaque tâche *Map* reçoit un fragment, le découpe en mots et, pour chaque occurrence d'un mot, génère une paire [mot, 1] (clé := mot, valeur := 1)
3. Les paires [mot, 1] sont groupées par mots et stockées
4. Chaque tâche *Reduce* reçoit les paires correspondant à un mot, fait l'addition des valeurs qui lui sont associées et stocke les résultats

Remarque : l'addition étant associative et commutative, il est possible de transférer dans les *Map* une partie des calculs des *Reduce* ; ainsi, une tâche *Map* peut aussi additionner le nombre d'occurrences de chaque mot dans le fragment qu'elle reçoit et générer plutôt des paires [mot, nombre d'occurrences du mot dans le fragment]. Cela permet de minimiser les échanges de données.

MapReduce : un autre exemple classique

Tâche à réaliser : multiplication (grande) matrice × vecteur

$$\mathbf{y} = \mathbf{M} \times \mathbf{x} \quad \text{ou} \quad y_i = \sum_{j=1}^{n} m_{ij} x_j$$

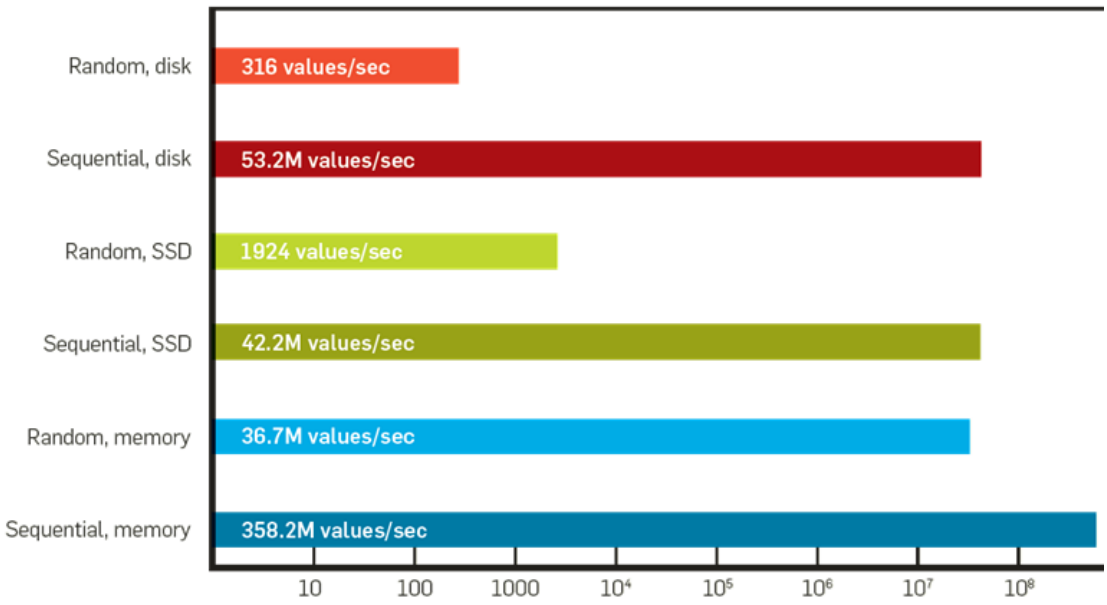


MapReduce : multiplication matrice × vecteur

1. La très grande matrice $\mathbf{M}$ est découpée en fragments qui sont des groupes de colonnes ; un fragment doit tenir dans la mémoire d'un nœud de calcul
2. Le vecteur $\mathbf{x}$ est aussi découpé en groupes de lignes de façon correspondante
3. Chaque tâche *Map* reçoit un fragment de la matrice $\mathbf{M}$ et le fragment correspondant de $\mathbf{x}$; pour chaque élément m_{ij} de son fragment, calcule $m_{ij}x_j$ et génère (i, $m_{ij}x_j$)
4. Les paires (i, $m_{ij}x_j$) sont groupées par i et stockées
5. Chaque tâche *Reduce* reçoit les paires correspondant une valeur de la clé i, fait l'addition des valeurs qui lui sont associées et stocke les résultats

Remarque : toujours grâce aux propriétés de l'addition, il est possible de transférer dans les *Map* une partie des calculs des *Reduce* pour minimiser les échanges de données ; ainsi, une tâche *Map* peut aussi additionner les $m_{ij}x_j$ pour tous les j de son groupe de colonnes.

Hiérarchie de stockage



* Disk tests were carried out on a freshly booted machine (a Windows 2003 server with 64GB RAM and eight 15,000RPM SAS disks in RAID5 configuration) to eliminate the effect of operating-system disk caching. SSD test used a latest generation Intel high-performance SATA SSD.

(figure issue de *Communications of the ACM* 2009 (8) : 36-44)

MapReduce et la fouille de données

- Un parallélisme massif serait inenvisageable sans un mécanisme robuste de reprise sur panne
- Le mécanisme de reprise sur panne de *MapReduce* impose le stockage des résultats des *Reduce*
- Pour beaucoup d'algorithmes—dont certains de fouille de données—ces résultats sont intermédiaires : l'itération suivante doit prendre ces résultats comme données d'entrée
- Le stockage des résultats intermédiaires, imposé par le mécanisme de reprise sur panne, engendre un ralentissement très significatif de ces algorithmes

La solution de Spark

- https ://spark.apache.org
- Revoir la présentation de Spark dans NFE204 (avant-dernier cours) : http ://b3d.bdpedia.fr/spark-batch.html

  (rappel MapReduce : si les données sont beaucoup plus volumineuses que les programmes, stocker localement les données et transférer plutôt les programmes)
- Idée clé de Spark : si les données intermédiaires sont volumineuses, les garder en mémoire vive ; en cas de panne *recalculer ce qui a été perdu*
- Concept important : *Resilient Distributed Dataset* (RDD) = collection de données partitionnée en fragments (*partitions*, *chunks*) distribués sur différents nœuds de calcul et conservés en mémoire

→ Fort impact sur la réduction de la latence des algorithmes itératifs

- A partir de la *v1.6* introduction des *DataFrame*, collections de données distribuées avec des colonnes nommées, conceptuellement similaires aux Data Frames de R

Spark : librairies existantes

- Librairies importantes pour nous (utilisées dans les TP de RCP216) :
  - MLlib : analyse des données, fouille de données, apprentissage statistique
  - GraphX : calculs sur les graphes (*data-parallel* et *graph-parallel*)
  - Spark Streaming : traitement de flux de données (par ex. Twitter, ZeroMQ, Kinesis, *sockets* TCP)
- Spark est écrit en Scala et peut être utilisé à partir de Scala, Java, Python et R (dans une moindre mesure, via SparkR ou sparklyr)
- Les programmes en Scala tournent dans une machine virtuelle Java (JVM), des programmes en Scala peuvent appeler des librairies en Java et inversement

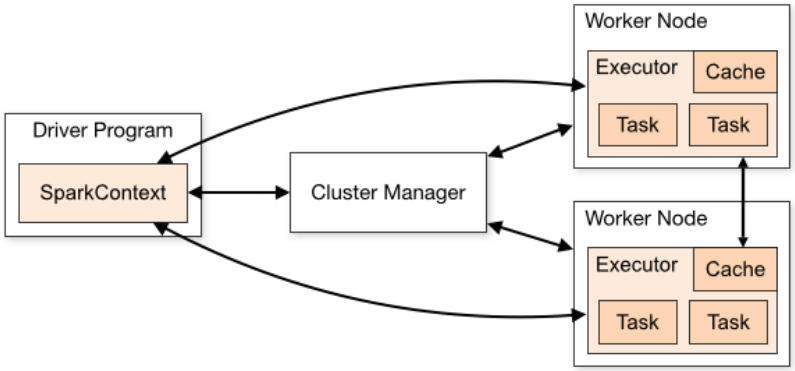
Fouille de données sur Hadoop

- Grande diversité, mutations rapides
- On peut citer, en *open source* :
  - H2O (0xdata) : librairie assez large et en développement rapide, orientée apprentissage statistique, exécution efficace, interface avec R ; support commercial de 0xdata
  - MLlib / Spark (Apache, Databricks) : librairie en développement rapide, utilisant le support efficace de Spark, interface avec R (SparkR) ; distribuée par Cloudera
  - Mahout (Apache) : librairie large mais très hétérogène, latence due à un fonctionnement MapReduce classique (stockage résultats intermédiaires), sans support commercial
- Également en *open source* :
  - Vowpal Wabbit (John Langford) : exécution efficace, inclut quelques algorithmes clé, mais sans support commercial
  - RHadoop (Revolution Analytics) : écrire des jobs MapReduce en R ; support commercial pour la licence Enterprise
  - RHIPE (Suptarshi Guha) : écrire des jobs MapReduce en R ; sans support commercial

Critères de choix

- Efficacité : suivi strict de MapReduce ou solutions pour réduire la latence ?
- Facilité d'utilisation : gestion plus ou moins explicite du parallélisme ?
- Fonctionnalités actuelles et perspectives de développement : avantage à certaines solutions *open source*
- Gestion de la distribution : explicite ou implicite, intégrable avec d'autres opérations MapReduce et non-MapReduce ? Parfait si compatible avec Hadoop YARN
- Format des données : HDFS, Bzip/Gzip, Apache Avro, HBase, Hive, Impala ?
- Support commercial : pour les solutions propriétaires mais aussi pour certaines solutions *open source*

Spark : déploiement



(figure issue de https ://spark.apache.org)

- Spécificités :
  - Gestionnaires de cluster : propre à Spark (mode *standalone*) ou Apache Mesos ou Hadoop YARN
  - Isolation entre applications Spark différentes : chaque application tourne dans une JVM et est donc isolée des autres, mais ne peut communiquer avec d'autres qu'à travers des fichiers
  - Pour des raisons d'efficacité, le *driver* devrait tourner sur un nœud du cluster