

5IN852 BDLE

Stockage à l'échelle du web

HUBERT NAACKE

NOVEMBRE 2024

Plan

- ▶ Contexte et problématique
- ▶ HDFS et cas d'usage
- ▶ Stockage : Format et organisation
- ▶ Stockage en mémoire
- ▶ Perspectives et Conclusion

Contexte

- ▶ Analyse de données à large échelle
 - ▶ Analyse offline/online
- ▶ Type d'application
 - ▶ Chaine de traitements (workflow)
 - ▶ Transformation élémentaire : entrée → traitement → sortie
 - ▶ Enchaîner les transformations
 - ▶ Données intermédiaires **volumineuses**
 - ▶ Un traitement peut imposer que les entrées et sorties soient des fichiers
 - Besoin de stocker les données intermédiaires dans des fichiers
- ▶ Conception incrémentale d'un workflow
 - Réutiliser les fichiers résultats des analyses précédentes

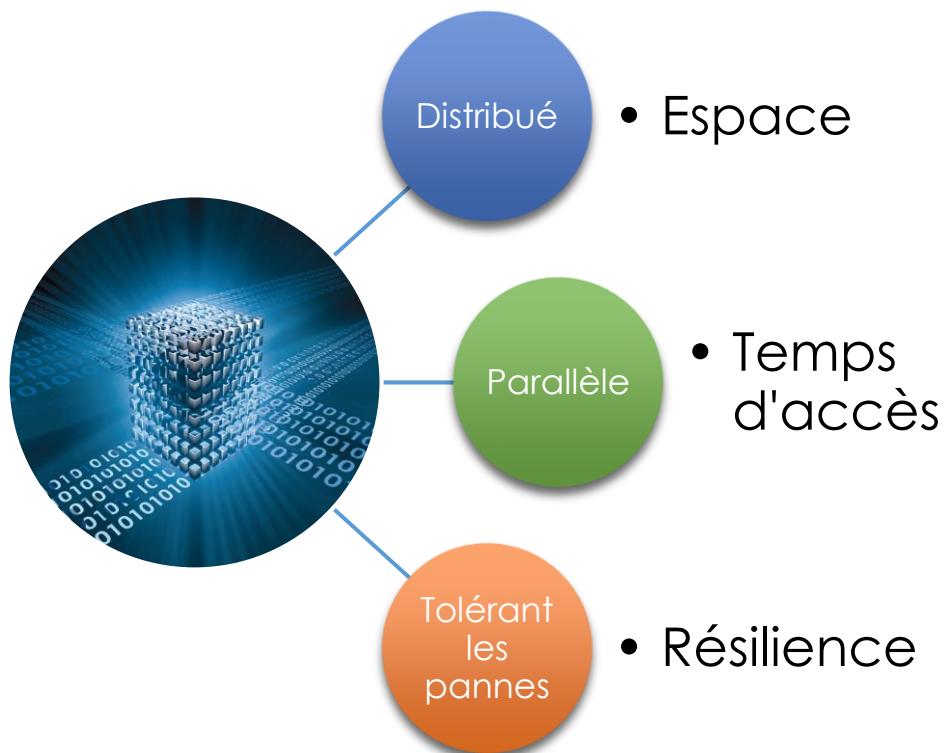
Contexte : Fichier

- ▶ Format d'un fichier
 - ▶ Textuel
 - ▶ ou Binaire
- ▶ Structure des données d'un fichier
 - ▶ Non structuré : image, audio, vidéo
 - ▶ document semi structuré
 - ▶ ou collection d'éléments de même type
- ▶ Description d'un fichier
 - ▶ Nom unique : emplacement + nom du fichier
 - ▶ Taille
 - ▶ Dates : création, dernière modification, dernière lecture, ...
 - ▶ Droits d'accès : propriétaire

Contexte : Système de fichiers

- ▶ Sert à organiser les fichiers en les classant
 - ▶ Dossier : contient des fichiers et des sous-dossiers
 - ▶ Organisation hiérarchique
- ▶ Accès en lecture à un fichier
 - ▶ Rechercher un fichier : explorateur de fichiers
 - ▶ Lire un dossier pour atteindre des sous-dossiers
 - ▶ Lire un fichier
- ▶ Accès en écriture
 - ▶ Créer un nouveau fichier
 - ▶ Modifier un fichier existant
 - ▶ Supprimer un fichier existant

Problématique du stockage à large échelle



Problématique du stockage à large échelle

- ▶ Grand nombre de fichiers à gérer (plusieurs millions de fichiers)
 - ▶ Grande taille d'un fichier (1 Go à plusieurs To par fichier)
 - ▶ Capacité de stockage > celle d'un disque
 - ▶ Une gestion et architecture centralisée ne convient pas
- Besoin d'une solution **distribuée** pouvant utiliser plusieurs disques
- ▶ Durée trop longue pour lire un fichier
 - ▶ Parcours séquentiel trop lent pour un gros fichier
- Besoin d'une solution **parallèle** pour lire le contenu d'un fichier

Les systèmes distribués mais non parallèles (NFS) ne conviennent pas

HDFS

ARCHITECTURE

ACCÈS

PANNES

PRATIQUE

CAS D'USAGE ET LOCALITÉ

Stockage sur un cluster de machines

- ▶ But : cumuler les capacités de **stockage** et de **traitement** de plusieurs machines
- ▶ Cluster (grappe) de machines
 - ▶ Une **machine** = unité de stockage et de traitement
- ▶ Avantages
 - ▶ Faible prix, bon rapport capacité/prix
 - ▶ Nombreuses machines → Parallélisme élevé (ex. 60 000 machines chez Critéo)
 - ▶ Extensible
- ▶ Inconvénient
 - ▶ Pannes fréquentes, occurrence proportionnelle au nombre de machines
 - ▶ 10 machines : 1 panne par an
 - ▶ $10 * 365$ machines : 1 panne par jour

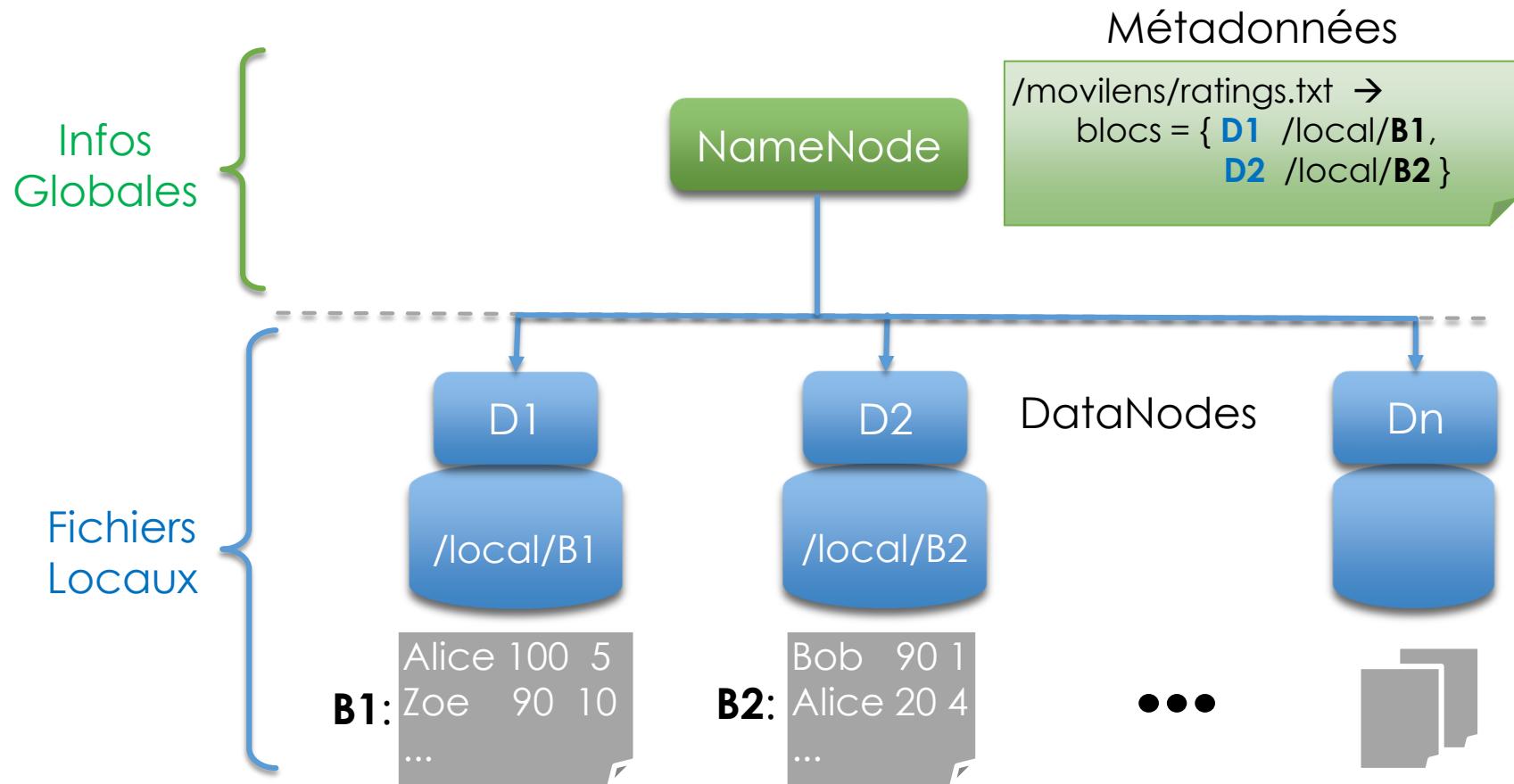
Localité données/traitement dans un cluster

- ▶ L'interconnexion des machines est en 2 niveaux
 - ▶ Une **machine** dans un rack
 - ▶ Un rack dans un cluster
- ▶ Localité donnée/traitement : décroît avec l'éloignement
 - ▶ Lire un fichier local d'une machine : **très rapide**
 - ▶ Lire un fichier d'une autre machine (du même rack) : rapide
 - ▶ Lire un fichier d'un autre rack : **lent**
 - ▶ Lire un fichier d'un autre cluster : **trop lent**
- ▶ Bénéficier de la localité
 - ▶ Déplacer les traitements vers les données
 - ▶ Favoriser les traitements locaux

Objectifs et choix de conception

- ▶ Tolérance aux pannes
 - ▶ Rester en service pendant une panne
 - ▶ Détection rapide, reprise automatique
- ▶ Conçu pour un scénario d'accès spécifique
 - ▶ Créer un fichier puis le lire **n** fois (avec $n >> 1$)
 - ▶ Ajout de données dans un fichier existant. Pas de modification du contenu
- ▶ Performance
 - ▶ Débit élevé pour créer un fichier et le lire
 - ▶ Paralléliser l'accès à un fichier
- Séparer : fichiers globaux / fichiers locaux contenant les données
 - ▶ **Métadonnées** : description *logique* globale des fichiers et dossiers
 - ▶ **Données** : un fichier *logique* composé de plusieurs fichiers locaux
 - ▶ Permet d'optimiser l'accès au contenu tout en préservant l'organisation globale

Architecture (1)



Architecture (2)

- ▶ Type Master/Slave
 - ▶ Un master, le **NameNode**, contrôle le système
 - ▶ Plusieurs slaves, un **DataNode** par machine
- ▶ **NameNode**
 - ▶ Il gère les **métadonnées** : description des dossiers et fichiers
 - ▶ Point d'entrée pour une demande d'accès
 - ▶ Connait l'état des Datanodes
 - ▶ Les données lues/écrites ne passent **pas** par le NameNode
- ▶ **DataNode**
 - ▶ Service : accès au système de fichiers (local FS)
 - ▶ Stocke le **contenu** d'une partie des fichiers

Fragmentation et réPLICATION des données

- ▶ Un fichier est découpé en blocs successifs de 64M
 - ▶ $F = \{B_1, B_2, \dots\}$
 - ▶ Taille d'un bloc $B_i = 64M$ par défaut (ajustable) << taille d'un fichier
- ▶ Chaque bloc est répliqué
 - ▶ Degré de réPLICATION = 3 (ajustable, par fichier ou par dossier)
 - ▶ Le Namenode connaît l'emplacement des répliques de chaque bloc
 - ▶ Fichier = {nom, liste de blocs}, bloc = liste de répliques, réplique = {D, B}
- ▶ Avantages
 - ▶ Tolérance aux pannes
 - ▶ Parallélisme pour les gros fichiers (< nombre de blocs)

Distribution et placement des données

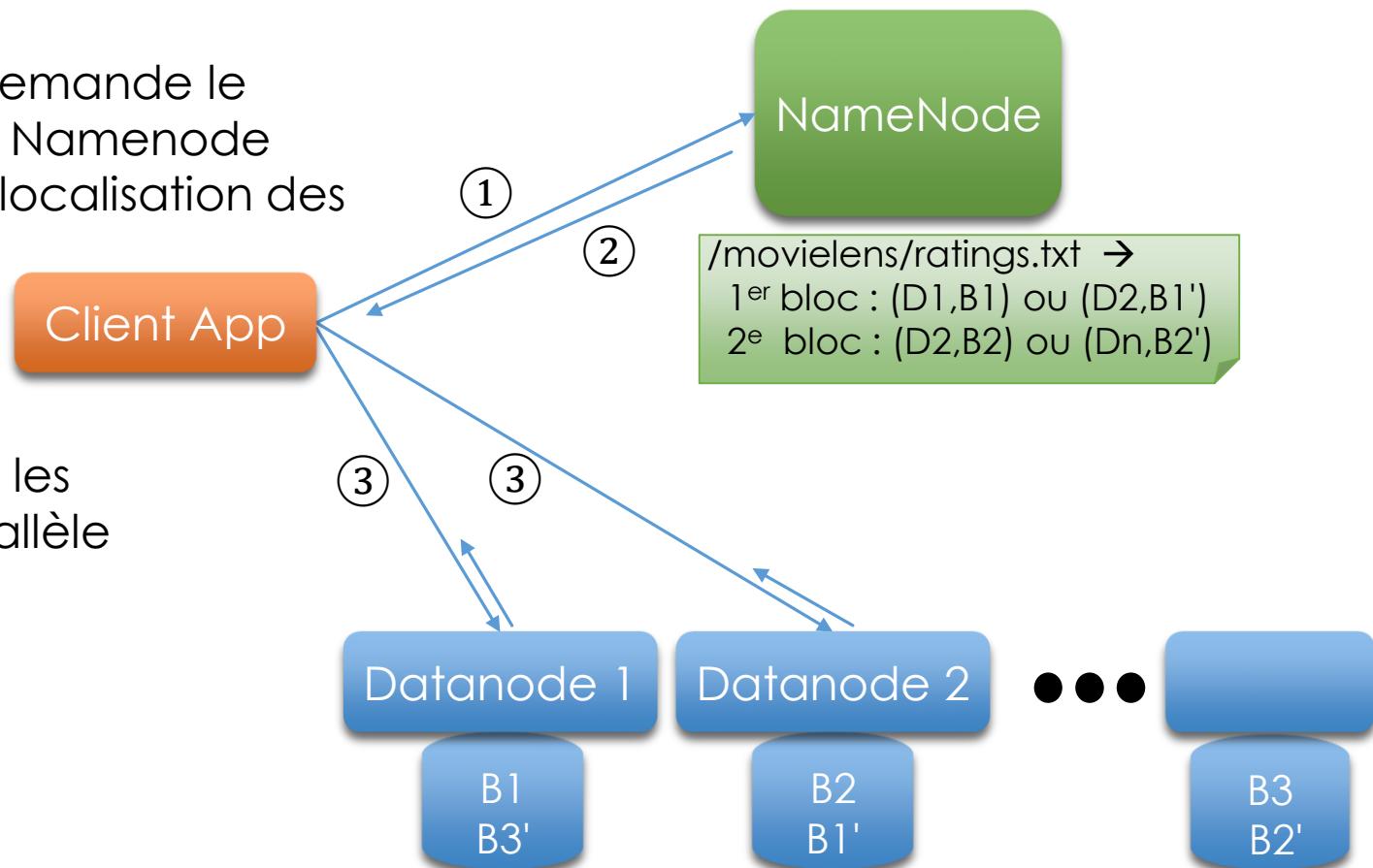
- ▶ Equilibrer le stockage
 - ▶ Remplir d'abord les datanodes avec un faible taux d'occupation
 - ▶ Possibilité de déplacer un bloc pour forcer l'équilibre
- ▶ Equilibrer la charge en lecture
 - ▶ Hypothèse : les blocs récents sont lus plus fréquemment que les autres
 - ▶ Distribuer les blocs récemment créés sur plusieurs datanodes
- ▶ Accès en parallèle aux différents blocs d'un fichier :
 - Placer les blocs d'un fichier dans le même rack
- ▶ Accès en cas de panne
 - Placer les répliques des racks différents de celui de la donnée "principale"

Lecture d'un fichier (1)

- 1) Le client demande au NameNode de lire le fichier nommé F
 - 2) Le NameNode
 - ▶ Lit les métadonnées : F → Liste de blocs : B1, B2, ... + Datanode Di des blocs
 - ▶ Réponse au client : liste de couples (Bi, {Di})
 - 3) Le client lit les blocs sur les Di
 - ▶ Pour chaque bloc Bi, choisit un DataNode Di stockant le bloc
 - ▶ Di envoie le contenu de Bi au client
-
- ▶ Accès **décentralisé** aux fichiers
 - ▶ Le client **cache** les métadonnées (Bi, {Di}) de F
 - ▶ Il ne recontacte pas le NameNode pour relire F
 - ▶ Améliore la scalabilité : évite la congestion du NameNode
 - ▶ Le client se charge de vérifier la validité du cache

Lecture d'un fichier (2)

- 1) Le client demande le ratings.txt au Namenode
Réponse : la localisation des blocs

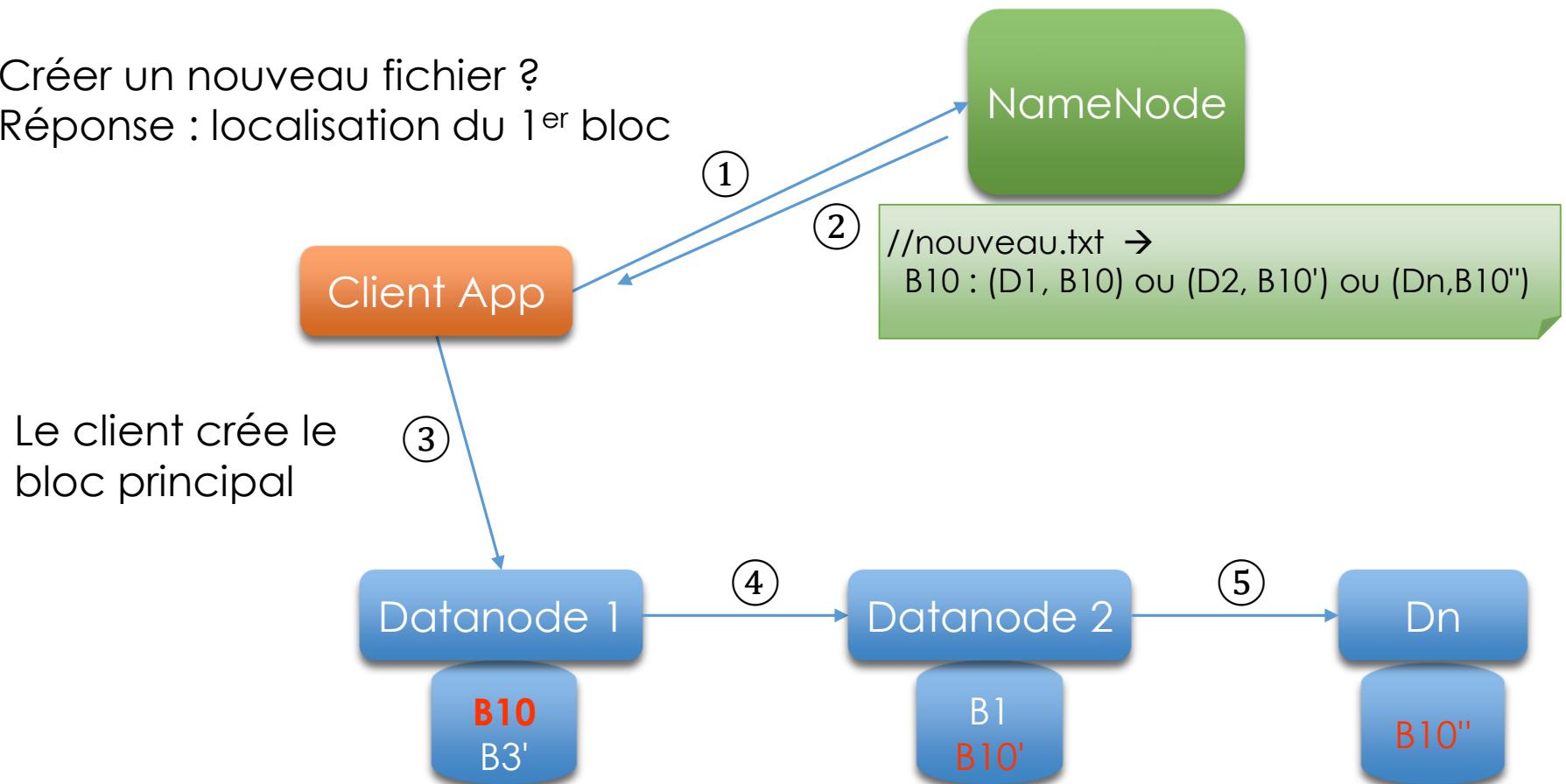


Ecriture d'un nouveau fichier (1)

- ▶ Client → Namenode : demande la création d'un nouveau fichier
 - ▶ Réponse : identifiant du 1^{er} bloc (et de ses répliques)
- ▶ Ecriture "principale" Client → Datanode
 - ▶ Client écrit le bloc sur le Datanode principal (pas de réPLICATION)
 - ▶ Bloc plein : invoquer le Namenode pour obtenir l'identifiant du prochain bloc (et de ses répliques)
 - ▶ Le Namenode sérialise les demandes concurrentes concernant le même fichier
- ▶ RéPLICATION
 - ▶ Pour chaque bloc B, le datanode principal copie B vers la 1^{ère} réplique
 - ▶ Ainsi de suite pour les autres répliques
 - ▶ Informer le client puis le Namenode de la fin de l'écriture

Ecriture d'un nouveau fichier (2)

Créer un nouveau fichier ?
Réponse : localisation du 1^{er} bloc

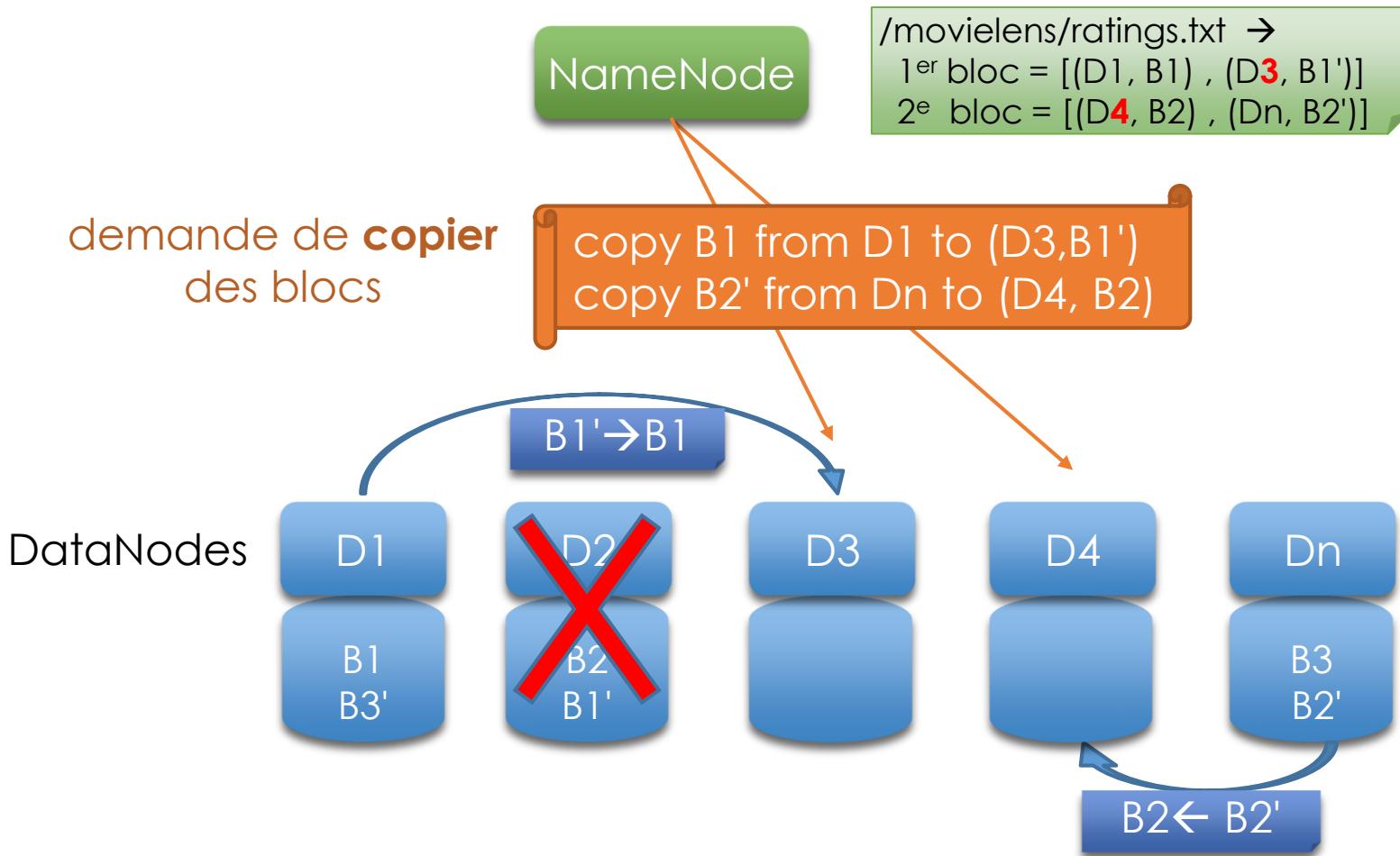


Tolérance aux pannes (1/3)

- ▶ Namenode
 - ▶ Un NameNode actif + un NameNode secondaire (backup)
 - ▶ Les opérations modifiant les métadonnées propagées sur les NameNode secondaires
 - ▶ Propagation synchrone
 - ▶ Garantit qu'un NameNode secondaire est identique au NameNode actif.
 - ▶ Permet le remplacement d'un NameNode en panne
 - ▶ Journalisation, point de reprise
- ▶ DataNode
 - ▶ Détection: signal **heartbeat** envoyé régulièrement au NameNode

Tolérance aux pannes (2/3)

Maintenir la réplication



Tolérance aux pannes (3/3)

Lecture d'un bloc corrompu

Situation: D2 ne répond pas ou D2 renvoie un bloc corrompu

Client demande **B2'** au lieu de B2

/movielens/ratings.txt →
1^{er} bloc : (D1,B1) ou (D2,B1')
2^e bloc : (**D2**,B2) ou (**Dn**,**B2'**)

Client



→ La lecture du fichier n'est **pas** interrompue

HDFS en pratique

- ▶ Plan des diapos suivantes
 - ▶ Installation
 - ▶ Utilisation
 - ▶ Administration

Installation d'HDFS

- ▶ Logiciel : fondation Apache, projet Hadoop
- ▶ Matériel : réseau de machines
- ▶ Connectivité NameNode - DataNode
 - ▶ accès ssh avec authentification par clés (sans mot de passe)
 - ▶ ssh-keygen
 - ▶ .ssh/authorized_keys sur chaque machine (NameNode et DataNode)
- ▶ Configuration
 - ▶ Rôle de chaque machine : Namenode (principal, secondaire), DataNode
- ▶ Initialisation
 - ▶ Formater le système de fichiers
 - ▶ hadoop namenode format
- ▶ Service
 - ▶ start-dfs stop-dfs

Démarrage de HDFS

- ▶ Démarrer le Namenode
 - ▶ Lire la dernière image
 - ▶ Appliquer le journal
 - ▶ Ecrire l'image courante et la répliquer
 - ▶ Attendre que les DataNodes soient prêts
- ▶ Démarrer les datanodes
 - ▶ Un dataNode transmet la description de ses blocs au NameNode
 - ▶ Description courte (8 octets) par bloc
 - ▶ Le Namenode informe les datanodes des éventuels blocs corrompus

Commandes HDFS

- ▶ Interface
 - ▶ lignes de commandes (CLI)
 - ▶ API
- ▶ Importer des données : put, appendToFile
- ▶ Exporter un fichier : get, getmerge, cat
- ▶ Explorer: ls, lsr, du, stat, count
- ▶ Organiser: mkdir, mv, cp, rm, setfattr
- ▶ Autorisations L et/ou E pour une personne, un groupe, une liste
 - ▶ chown, chmod, setfacl
- ▶ Profile : quota d'espace pour un dossier

Administration de HDFS (1/2)

- ▶ Redimensionner un cluster HDFS
 - ▶ En fonction des besoins applicatifs : volume des données, charge
 - ▶ Ajouter/supprimer un dataNode
 - ▶ `dfsadmin refreshNodes` listés dans `dfs.include` et `dfs.exclude`
- ▶ Ré-équilibrer la taille des données stockées dans les dataNodes
 - ▶ Garantie que pour tout Di , $|1 - (\text{taille } Di / \text{taille moyenne})| < N\%$
 - ▶ Déplacer des blocs Di le plus rempli → Dj le moins rempli
 - ▶ Utilitaire : `balancer`
 - ▶ Plafonner les ressources utilisées
 - ▶ ne pas ralentir le service courant
 - ▶ `dfsadmin setBalancerBandwidth`

Administration de HDFS (2/2)

- ▶ Optimiser les performances de lecture
 - ▶ ajuster le degré de réPLICATION si
 - ▶ nbre de blocs * degré < nb disques
- ▶ Cloner un cluster HDFS
 - ▶ distcp2
- ▶ Etat des dataNodes:
 - ▶ report, fsck, nombre de blocs sur/sous répliqués
- ▶ Interface d'administration : GUI
 - ▶ Navigateur web: tableau de bord, explorateur

HDFS et analyse de fichiers

- ▶ Plan des diapos suivantes
 - ▶ Cas d'usage typiques
 - ▶ HDFS + Spark : localité d'accès
 - ▶ Bilan

Analyser un fichier :

3 cas d'usage

1 : Pandas
(local)

Petit
fichier

Rapide

2 : Pyspark
local

Grand
fichier

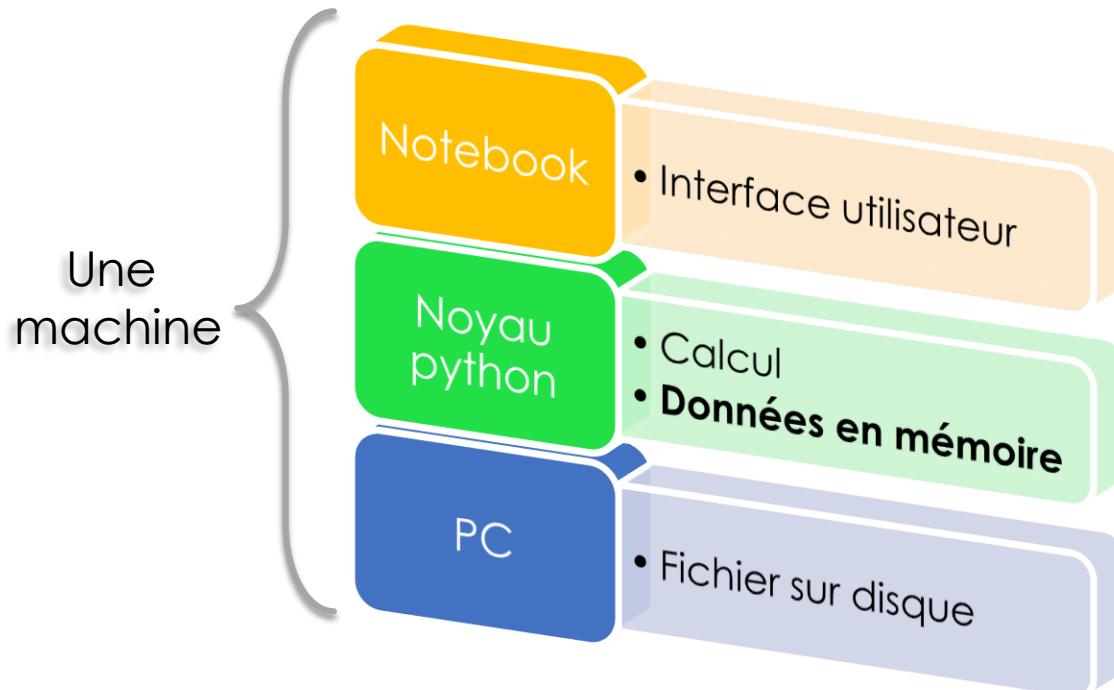
Lent

3 : Pyspark
cluster

Grand
fichier

Rapide

Analyse avec pandas



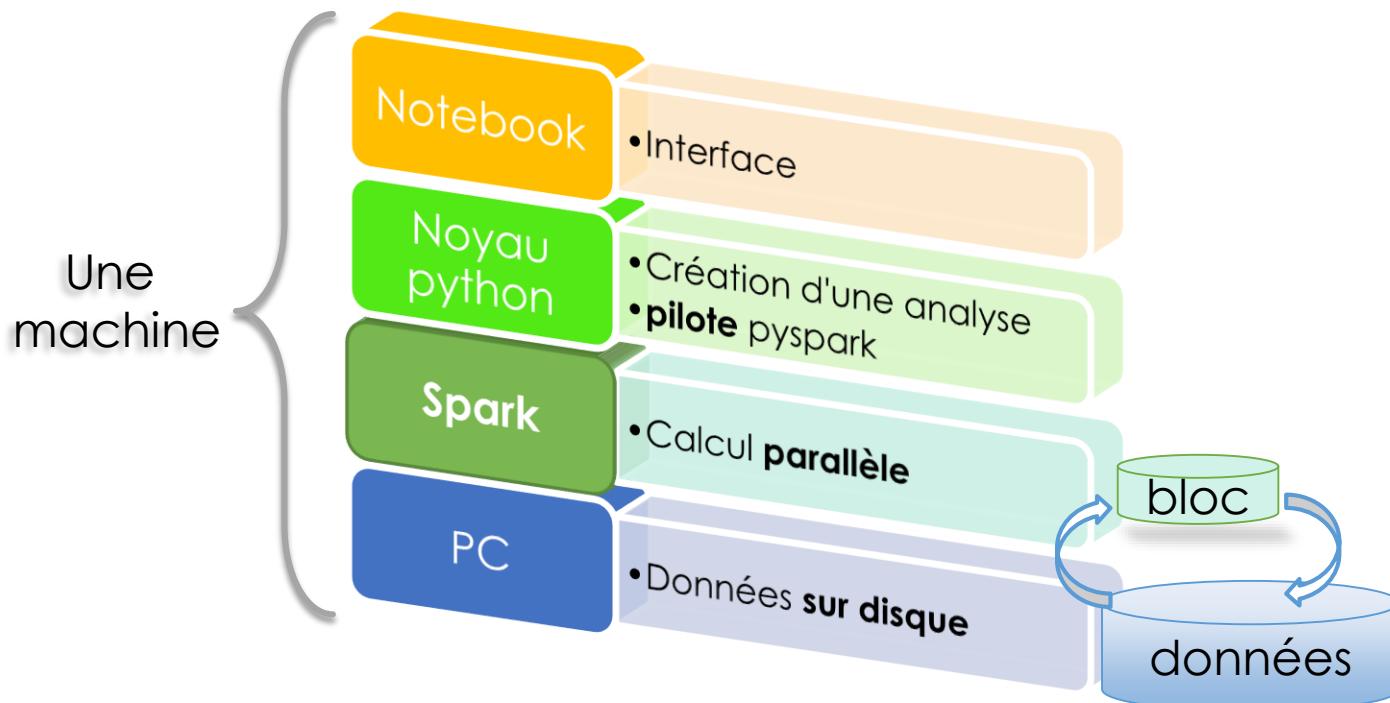
Analyse avec pandas



- ▶ Bibliothèque pandas ou similaire
 - ▶ Charger le fichier dans la mémoire du noyau python
- ▶ Avantages
 - ▶ Accès mémoire rapide
 - ▶ Calcul rapide possible
- ▶ Inconvénients
 - ▶ Limité aux petits fichiers
 - ▶ Ne fonctionne pas si le fichier dépasse la mémoire disponible
 - ▶ Besoin de librairies spécifiques de calcul parallèle

Analyse avec pyspark local

"on ajoute la couche Spark"



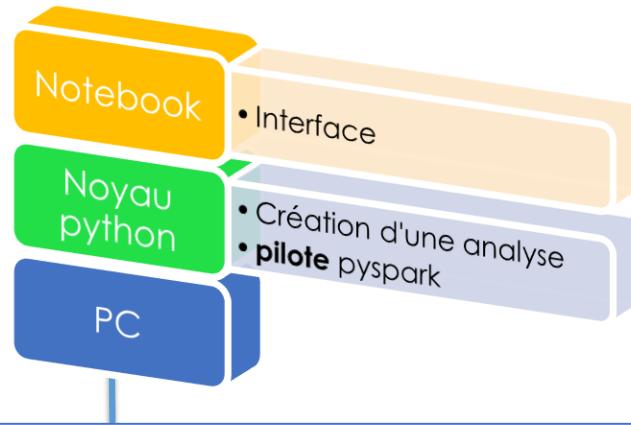
Analyse avec pyspark local



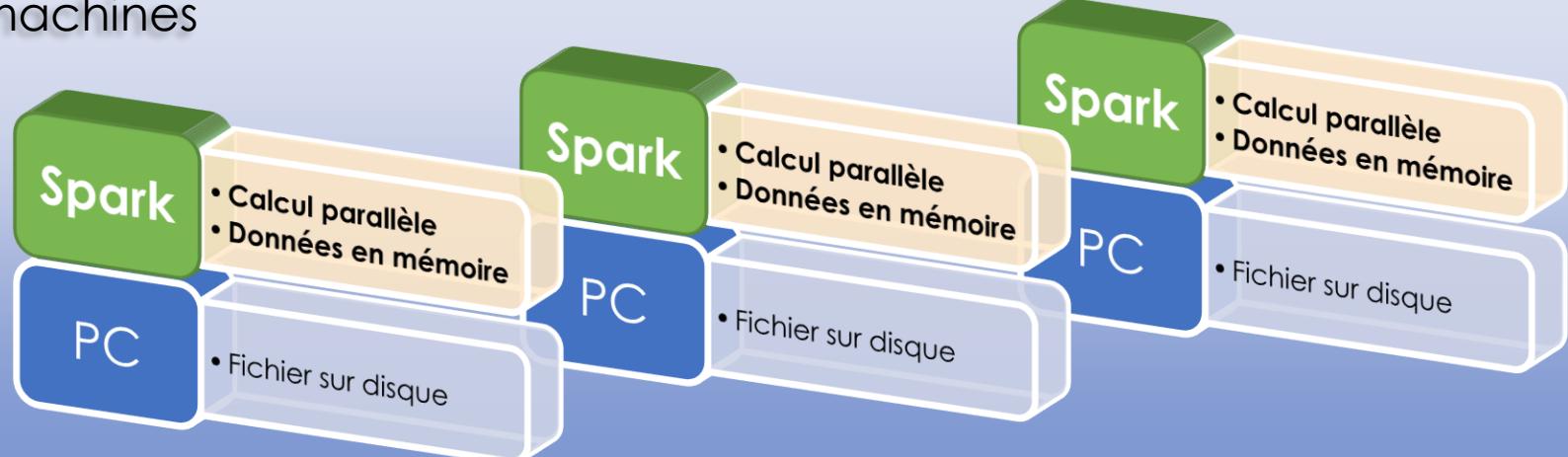
- ▶ Bibliothèque pyspark
 - ▶ Le fichier n'est **pas** chargé en entier dans le noyau python
- ▶ Avantages
 - ▶ Lire des grands fichiers qui ne tiennent pas en mémoire
 - ▶ Par itération : Lire → traiter → écrire
 - ▶ Calcul parallèle natif si plusieurs coeurs
- ▶ Inconvénients
 - ▶ Calcul lent : **un** seul PC

Analyse avec pyspark cluster

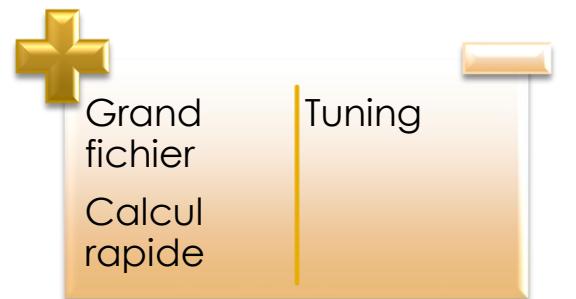
"on déploie Spark sur un cluster"



M machines



Analyse avec pyspark cluster



- ▶ Bibliothèque pyspark
 - ▶ Usage identique au mode local
- ▶ Avantages
 - ▶ Grands fichiers chargés en mémoire
 - ▶ Utilise les mémoires de toutes les machines
 - ▶ Calcul parallèle très performant
- ▶ Inconvénients ?
 - ▶ Réglage (*tuning*) plus complexe

pyspark cluster + HDFS

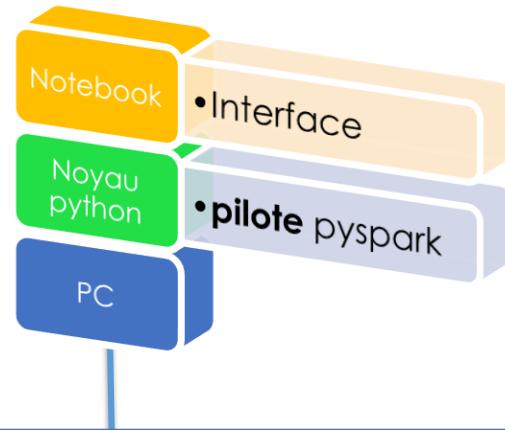
- ▶ Cas d'usage le plus approprié au Big Data
- ▶ Fichier = collection d'éléments (exple, collection d'images)
 - ▶ Lire un fichier → appliquer un traitement sur chaque élément du fichier
- ▶ Bénéficie de l'architecture HDFS
 - ▶ Le fichier est distribué sur un cluster de M machines
 - ▶ Chaque machine a C cœurs
 - ▶ Traitement indépendant par élément et **N tâches parallèles** (degré N)
 - ▶ degré max pour lire un fichier: $N_{max} = \min(\text{nombre de blocs du fichier}, M * C)$
- ▶ Traitement tolérant aux pannes
 - ▶ Si échec du traitement d'un bloc alors
 - ▶ Reprendre **seulement** le traitement du bloc

Localité (1/4)

pyspark cluster + HDFS

Une machine a deux rôles :

- ▶ Stockage HDFS : DataNode
- ▶ Calcul et cache Spark : Worker Node

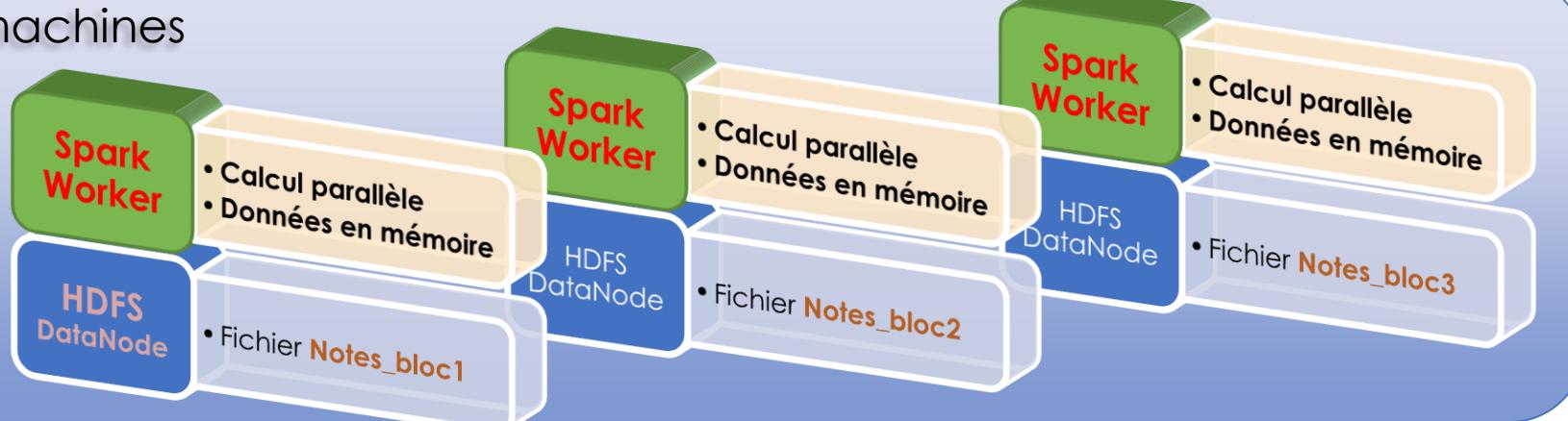


Appli :

```
N = spark.read.  
json ("hdfs://Notes") .  
selectExpr (....)
```

"N réside dans le cluster,
pas dans l'appli"

M machines



Localité (2/4)

Export de données

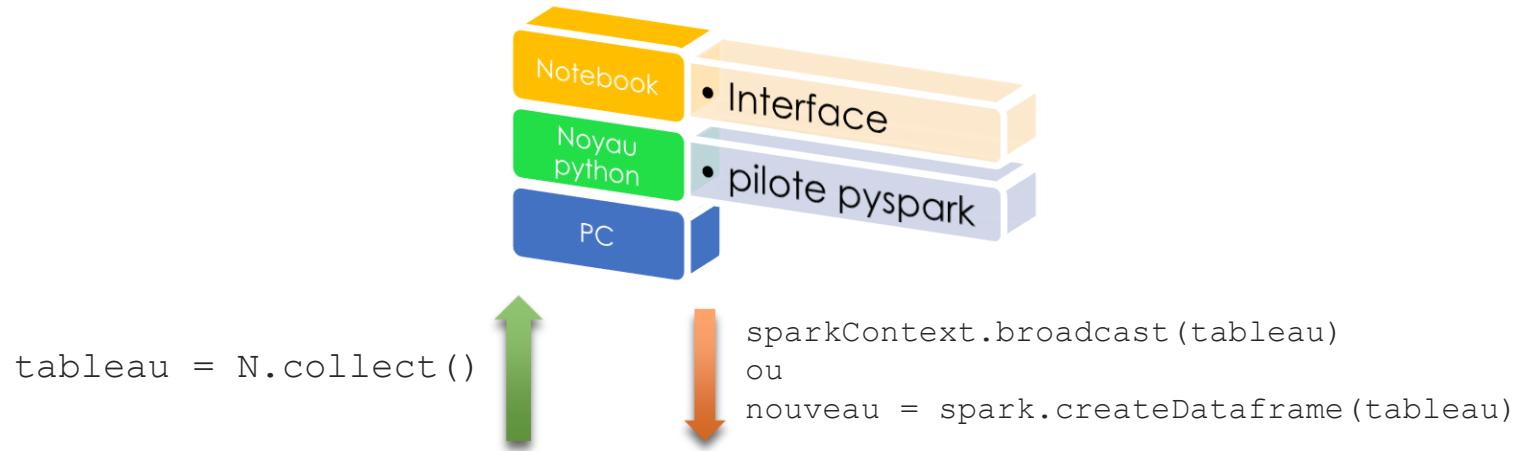
- ▶ Favoriser un traitement **local** dans Spark **sans** transférer les données
 - ▶ Déplacer le programme vers les données et non l'inverse
- ▶ Maîtriser la localité et les déplacement des données ?
- ▶ Exporter vers l'application des données gérées par Spark
 - ▶ d.collect() ou d.toPandas()
 - ▶ les données "remontent" vers l'application
 - ▶ Long car **transfert** puis conversion de données spark → python
 - ▶ Export limité aux « petites » quantités de données
 - ▶ les données exportées doivent tenir **en mémoire** dans l'application

Localité (3/4)

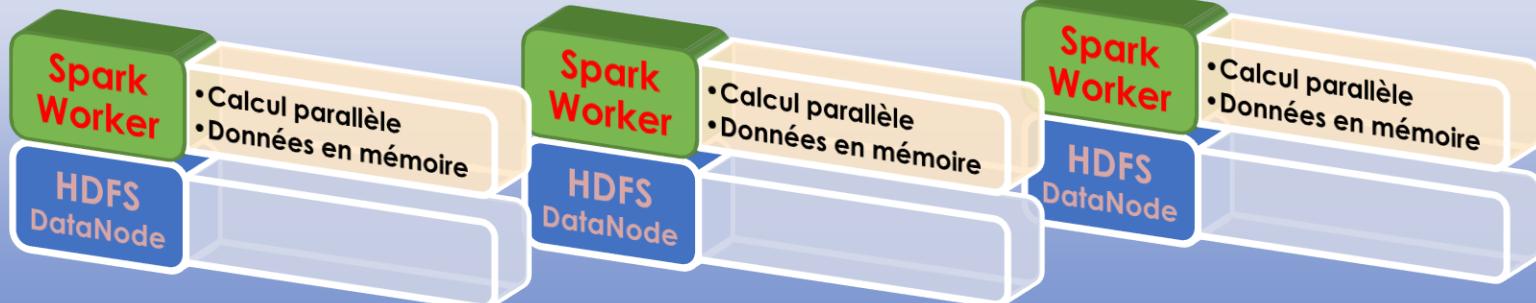
Import de données

- ▶ Importer des données applicatives dans Spark ?
 - ▶ Broadcast : les données "descendent" vers toutes les machines
 - ▶ transfert \simeq taille tableau * nombre de machines
 - ▶ Créer nouveau dataframe
 - ▶ les données sont distribuées entre les machines
 - ▶ transfert \simeq taille tableau
 - ▶ Données python (ex. tableau numpy) dans une opération map avec UDF
 - ▶ Simple à écrire mais risque de transfert très grand. (anti pattern!)
 - ▶ transfert \simeq taille tableau * nombre de tâches
 - ▶ Milliers de tâches quand on traite un grand fichier (1 tâche par bloc de 64MB)

Localité (4/4) Illustration



M machines



Bilan des cas d'usage

Fonction	pandas	pyspark local	pyspark cluster
Petit fichier en mémoire	✓	✓	✓
Grand fichier : itération		✓	✓
Grand fichier : en mémoire			✓
Calcul parallèle	n cœurs	n cœurs	n cœurs * M machines
Localité Stockage HDFS /Traitement			✓
tolère les pannes			✓

Formats de stockage

Formats de stockage

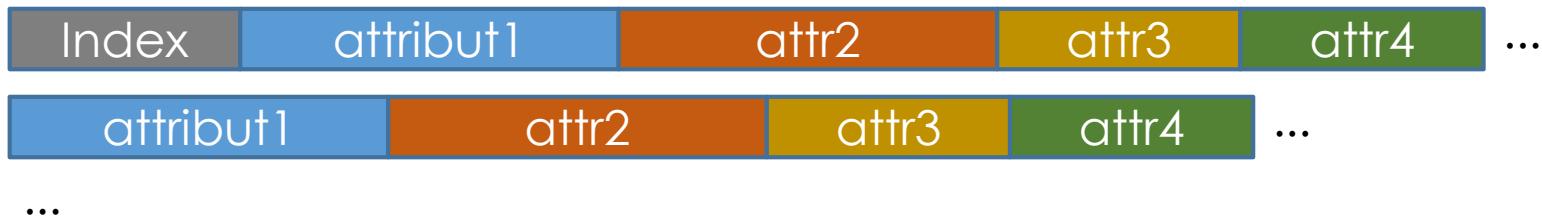
- ▶ Format texte
 - ▶ CSV, JSON
 - ▶ Avantage : lisible
- ▶ Format binaire
 - ▶ Parquet
 - ▶ RC **ORC**
 - ▶ K-store
 - ▶ Avantages : compact (= coût moindre), accès rapide
- ▶ Stockage avec partitionnement des données

Stockage au format RC et ORC

- ▶ Record-Columnar File Format (RCFile)
 - ▶ Proposé par Facebook, utilisé par Apache Hive
- ▶ Objectif
 - ▶ Stockage plus compact par compression
 - ▶ Accès rapide aux données par n-uplet (record)
- ▶ Principe
 - ▶ Découpage **horizontal** des données
 - ▶ Compression **verticale** des valeurs d'une colonne
- ▶ Stockage optimisé : RC File → ORC File
 - ▶ <https://code.fb.com/core-data/scaling-the-facebook-data-warehouse-to-300-pb/>

Structure du format ORC

- ▶ Segmentation horizontale des **lignes** en blocs de 256MB puis en sous-groupes
- ▶ Métadonnées d'un bloc
 - ▶ Emplacement des **groupes**
 - ▶ index des **colonnes**
- ▶ Encodage compressé du contenu de chaque colonne de chaque groupe
- ▶ Stockage consécutif des colonnes compressées
 - ▶ Un fichier = juxtaposition des contenus des colonnes



ORC : Segmentation

- Une table → m **blocs** de **256 MB**

	Nom	Prénom	Ville	Age
Bloc 1				
			
			
Bloc m				

Le nombre de lignes **L** dans un bloc peut être très grand : $L = 256M/\text{taille nuplet}$

ORC : Métadonnées (1/2)

- ▶ **Index** d'un bloc → infos sur n **groupes** de **10 000** lignes

	Nom	Prénom	Ville	Age
Bloc i	Gr 1			

Bloc i	Gr n			

ORC : Métadonnées (2/2)

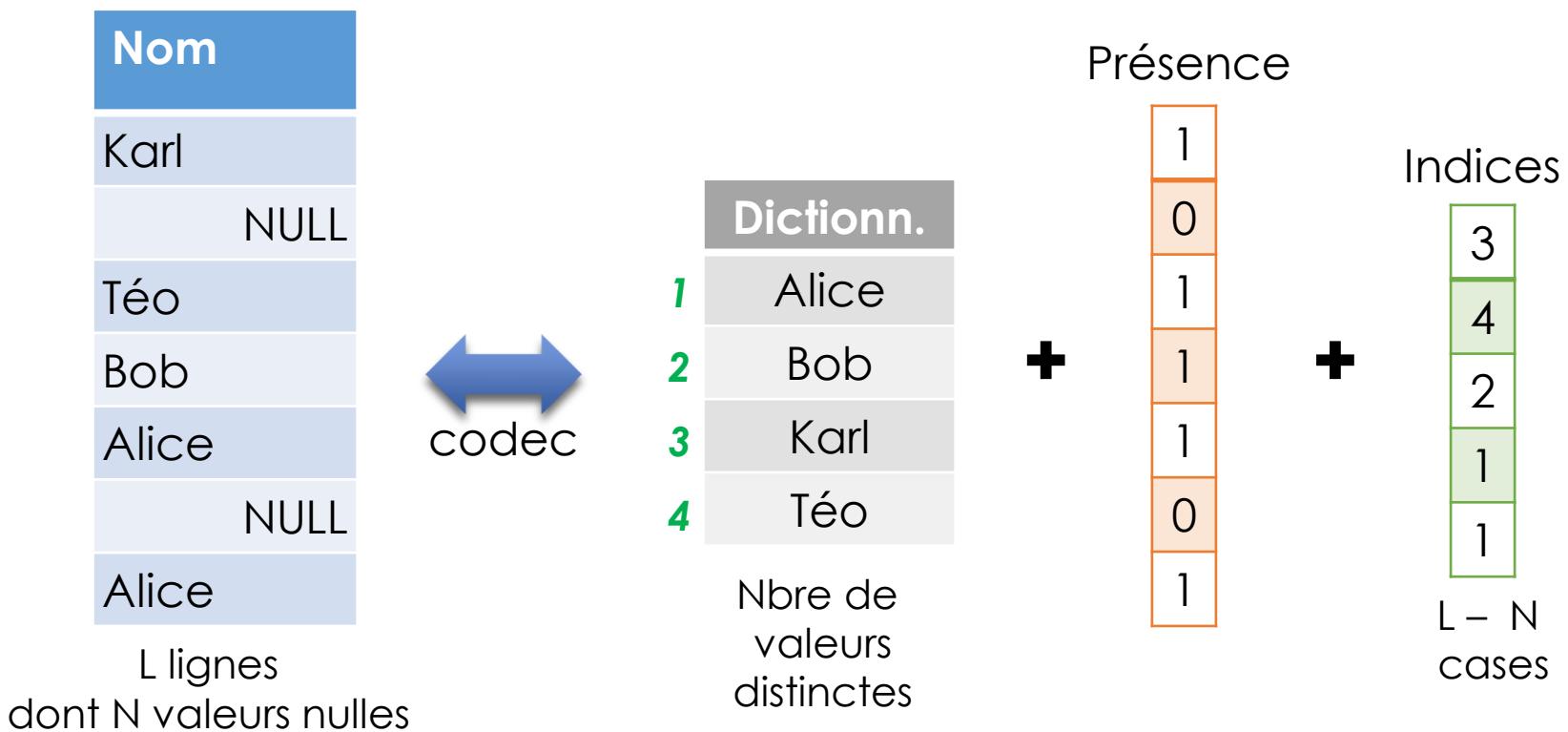
Index de colonne

- ▶ Un groupe $Gr_i \rightarrow$ position pour localiser le groupe
- ▶ Statistiques de colonne : Nbre de valeurs non nulles, min, max,
Intéressant si le domaine d'un groupe est une "petite" partie du domaine de l'attribut.
- ▶ Filtre de Bloom : présence probable d'une valeur

	Nom	Prénom	Ville	Age
	min, max	min, max	min, max	min, max
$Gr_1 :$	Alice, Téo			18, 40
$Gr_2 :$	Alan, Zoé	...		25, 60
...			...	
$Gr_n :$	Bob, Paul			5, 80

ORC : Encodage

- ▶ Dictionnaire de colonne : valeurs distinctes du domaine
- ▶ Chaque colonne d'un bloc est encodée



ORC : Compression

- ▶ Nombreux algorithmes de compression existants
 - ▶ Lempel-Ziv-Oberhumer (LZO) Zlib,
 - ▶ Run Length Encoding
 - ▶ Optimisés : Snappy, LZ4, Ozip, ...
- ▶ Compression adaptée pour un type de valeurs
 - ▶ Nombres entiers : integer packing, delta value
 - ▶ Chaînes de caractères avec préfixe commun
 - ▶ Dates
- ▶ Taux de compression élevé (4 à 10)
 - ▶ car nombreuses répétitions de valeurs dans une colonne

ORC : Performances

- ▶ Faible taille des données compressées
 - ▶ Moins de lectures consécutives
- ▶ Traitement des blocs en parallèle
 - ▶ Un bloc compressé tient dans un bloc HDFS
 - ▶ HDFS permet de lire/écrire les blocs en parallèle
- ▶ Lecture avec projection
 - ▶ Lire seulement les colonnes utiles
- ▶ Lecture avec sélection
 - ▶ utiliser [min, max] ou le filtre de Bloom pour cibler les groupes d'un bloc à lire et éviter de lire tous les groupes.
- ▶ Gain de perf en lecture > surcoût pour décoder les données

ORC : Projections et Sélections

- ▶ Select nom, ville from Fichier

Bloc k



- ▶ Select nom from Fichier where ville = 'Aix'

Bloc i



Bloc j



...

'Aix' n'est pas une ville du bloc j

Stockage de données imbriquées

- ▶ Motivation : données plus riches que les données tabulaires
- ▶ Encodage : 2 méthodes
 - ▶ Méthode de Dremel : coder les répétition et définitions
 - ▶ Méthode de ORC et Arrow : coder les longueurs et présences
- ▶ Comparaison des méthodes
 - ▶ Taille : ORC plus compacte que Dremel en moyenne
 - ▶ Lecture d'un attribut: Dremel plus rapide qu'ORC en moyenne
- ▶ Ref biblio : article « Test of time » à VLDB 2020
 - ▶ Dremel: A Decade of Interactive SQL Analysis at Web Scale
 - ▶ <http://www.vldb.org/pvldb/vol13/p3461-melnik.pdf>
 - ▶ Voir aussi les requêtes sur des données encodées :
 - ▶ [Inside Capacitor BigQuery's next-generation columnar storage format](#), Blog
 - ▶ [Storing and Querying Tree-Structured Records in Dremel](#), VLDB 2014

Données imbriquées : schéma

- ▶ Besoin de représenter des données complexes
 - ▶ Structures avec imbrication, répétition
 - ▶ Répétition de structures proches mais pas identiques
- ▶ Schéma définissant l'occurrence des structures
 - ▶ Occurrence multiple : repeated group *
 - ▶ Attribut optionnel : optional ?

```
message Document {
    required int64 DocId;
    optional group Links {
        repeated int64 Backward;
        repeated int64 Forward; }
    repeated group Name {
        repeated group Language {
            required string Code;
            optional string Country; }
        optional string Url; }}
```

r₁

```
DocId: 10
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
    Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

r₂

```
DocId: 20
Links
  Backward: 10
  Backward: 30
  Forward: 80
Name
  Url: 'http://C'
```

Données imbriquées : format avec répétition/définition

```

DocId: 10      r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
Language
  Code: 'en'
Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'

```



```

DocId: 20      r2
Links
  Backward: 10
  Backward: 30
  Forward: 80
Name
  Url: 'http://C'

```

- ▶ Encoder chaque branche
- ▶ Notion de **Répétition**
 - ▶ L'indice de l'élément répété
 - ▶ Ne pas compter les éléments optionnels ou requis
 - ▶ indice <= nombre d'éléments « repeated » sur la branche
 - ▶ Backward dans [0, 1]
 - ▶ Code dans [0, 2]
 - ▶ Country dans [0, 2]
 - ▶ Notion de **Définition**
 - ▶ indice de l'élément répété ou optionnel qui est défini sur la branche

```

message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}

```

Données imbriquées

Exemple 1

- ▶ Modèle Name.Language.country
 - ▶ Considérer **tous** les chemins Name*.Language*.Country?
 - ▶ Chemin : name.language.country = 'us'
 - ▶ rép = 0 car 1^{er} chemin de r1
 - ▶ déf = 3 car les 3 éléments répétables (ou optionnels) name, language et country sont définis
 - ▶ Chemin name.language
 - ▶ Value = NULL car le chemin n'a pas de country
 - ▶ Rép = 2 car l'élément language est « nouveau »
 - ▶ L'élément name est commun avec le chemin précédent
 - ▶ Def = 2 car les 2 éléments répétables name et language sont définis
 - ▶ Chemin name
 - ▶ Rép = 1 car l'élément name est nouveau
 - ▶ Def = 1 car name est défini
 - ▶ Chemin name.language.country = 'gb'
 - ▶ Rép = 1 car l'élément name est nouveau
 - ▶ Déf = 3 car les 3 éléments name, language et country sont définis

```

DocId: 10      r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
Language
  Code: 'en'
Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'

DocId: 20      r2
Links
  Backward: 10
  Backward: 30
  Forward: 80
Name
  Url: 'http://C'

```

```

message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; })
}

```

Name.Language.Country		
value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

Données imbriquées

Exemple 2

```

DocId: 10      r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'

DocId: 20      r2
Links
  Backward: 10
  Backward: 30
  Forward: 80
Name
  Url: 'http://C'

```

- ▶ Modèle Name.Language.Code

- ▶ Considérer tous les chemins Name*.Language*.Code

- ▶ r₁

- ▶ Chemin : name.language.code = 'en-us'

- ▶ Rép = 0 1^{er} chemin

- ▶ Déf = 2 car les 2 éléments répétables name et language sont définis.

- ▶ Rmq: Code étant obligatoire, on ne le compte pas

- ▶ Chemin : name.language.code = 'en'

- ▶ Rép = 2 car le 2^e élément répétable language est « nouveau »

- ▶ Déf = 2

- ▶ ...

- ▶ r₂ : un seul chemin « name »

```

message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country;
      optional string Url; } }
}

```

Name.Language.Code		
value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Données imbriquées

Exemple 3

- ▶ Modèle Links.Backward
 - ▶ Tous les chemins : Links?.Backward*
- ▶ r₁
 - ▶ chemin : links (NULL,0,1)
- ▶ r₂
 - ▶ chemin : Links.Backward = 10 (10,0,2)
 - ▶ ...

```
DocId: 10          r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
    Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
DocId: 20          r2
Links
  Backward: 10
  Backward: 30
  Forward: 80
Name
  Url: 'http://C'
```

DocId		
value	r	d
10	0	0
20	0	0

Name.Url		
value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward		
value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

Links.Backward		
value	r	d
NULL	0	1
10	0	2
30	1	2

Organisation du stockage :

stockage optimisé
pour l'analyse

PARTITIONNEMENT PAR ATTRIBUT

CACHE EN MÉMOIRE

Motivation : accès sélectif

- ▶ Cas 1 : Lire un fichier pour analyser seulement une portion
 - ▶ Lecture avec sélection selon un critère
 - ▶ Exple : Where année between 2017 and 2018
 - ▶ Lire seulement les données qui satisfont le critère
 - ▶ "*on pousse la sélection sur le disque*"
- ▶ Cas 2 : Rapprocher les données de 2 grands fichiers
 - ▶ Comparer seulement des portions stockées sur le même disque
 - ▶ "*on évite les transferts entre les machines*"

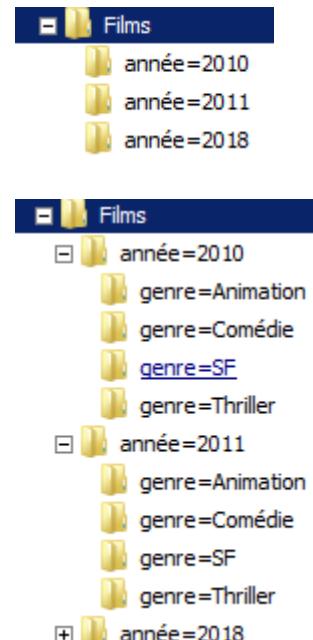
Partitionnement

Accès sélectif

Analyse plus rapide

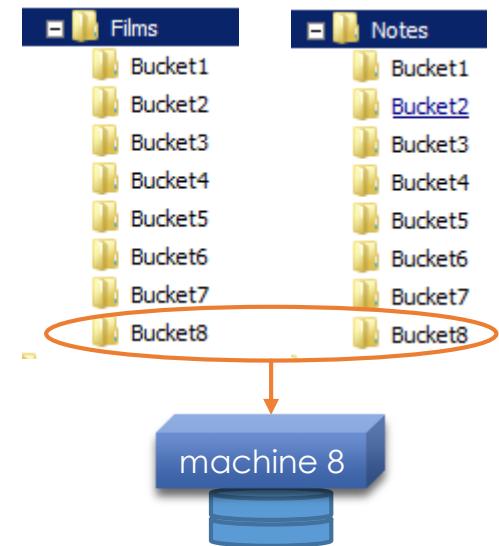
Partitionnement par attribut (1/2)

- ▶ Partition par **valeurs distinctes** d'un **attribut**
 - ▶ `data.write.partitionBy("année").json("dossier")`
- ▶ Composer **plusieurs** attributs
 - ▶ `data.write.partitionBy("année", "genre").json("dossier")`
 - ▶ Hiérarchie
- ▶ Divers **formats** de fichier supportés (csv, json, parquet)
 - ▶ `data.write.partitionBy("age").parquet("dossier")`
- ▶ Pousser une sélection
 - ▶ `where année in (2005, 2007) → Lire seulement 2 dossiers`
- ▶ Inconvénient
 - ▶ Grand nombre de dossiers (= domaine de l'attribut)



Partitionnement par attribut (2/2)

- ▶ Partitionnement par **hachage**
 - ▶ en P partitions
 - ▶ `Films.bucketBy(8, "nF").json("Films")`
 - ▶ `Notes.bucketBy(8, "nF").json("Notes")`
 - ▶ Plusieurs attributs possible
 - ▶ Utile pour attribut avec nombreuses valeurs distinctes
- ▶ Avantages
 - ▶ Pousser une sélection
 - ▶ `where nF=999` → Lire seulement le bucket 3
 - ▶ Requête Group by *nF* sans transfert
 - ▶ Jointure entre 2 fichiers sans transfert ?
 - ▶ Les deux **bucket i** de Films **ET** de Notes sont sur le disque de la même machine



Persistante disque / mémoire (1/4)

- ▶ Stockage sur disque + copie en mémoire
- ▶ Cache en mémoire volatile *persistent* ???
 - ▶ Répliqué dans la mémoire de plusieurs machines pour tolérer les pannes
- ▶ Format en mémoire
 - ▶ Encodage par colonne et compression (parquet / orc)
- ▶ Partitionnement en mémoire
 - ▶ Par attribut et hachage
- ▶ Accès et sélection très rapide avec SIMD

Persistance disque / mémoire (2/4)

- ▶ Spark Dataset ou Dataframe
 - ▶ **repartition**(nombre de partitions, attributs)
 - ▶ **persist()**
- ▶ Exemple
 - ▶ Notes = spark.read.json(".....movielens/Notes.json")
 - ▶ Note.count()
 - ▶ lire le fichier, **trop lent!**
 - ▶ NoteMémoire = Notes.**repartition**(8, Notes.année).**persist()**
 - ▶ NoteMémoire.count()
 - ▶ lire le fichier, transformer au format orc et le garder en mémoire :
 - ▶ **Lent**, mais acceptable car traité une seule fois
 - ▶ NoteMémoire.count() **Très rapide !**

Persistance disque / mémoire (3/4)

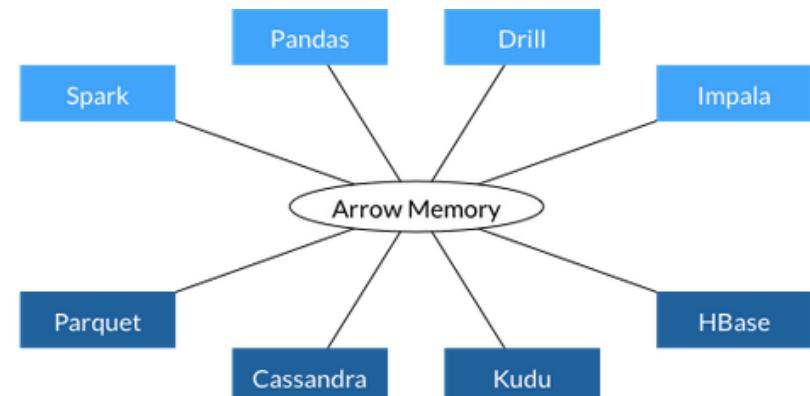
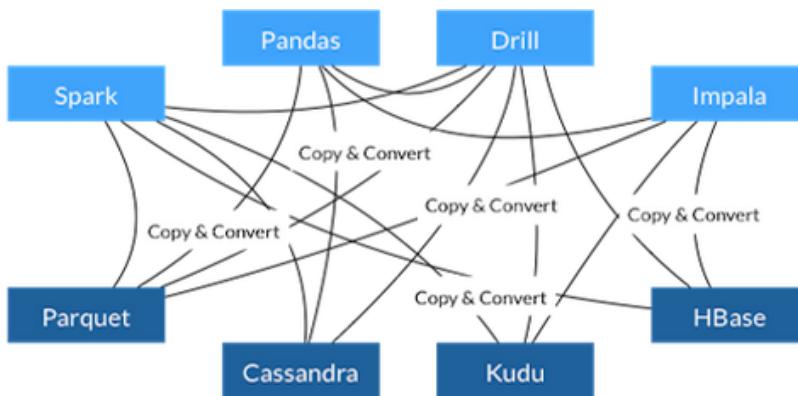
Solutions connexes

- ▶ Apache **Arrow**
- ▶ Oracle in memory
- ▶ Hana SAP
- ▶ Vertical
 - ▶ Ref : Why All Column Stores Are Not the Same Twelve Low-Level Features That Offer High Value to Analysts
 - ▶ <https://www.vertica.com/blog/column-stores-not>
- ▶ ...

Persistante disque / mémoire (4/4)

Apache Arrow

- ▶ Couche commune en mémoire
- ▶ Pour persister et partager des données
- ▶ Données **réutilisées** par plusieurs outils applicatifs sans recopie ni conversion.
- ▶ Format optimisé en colonne, cf. ORC



Oracle In Memory (1/3)

- ▶ Cas d'usage combiné : OLTP + OLAP
- ▶ Stockage disque en format ligne "classique"
- ▶ Cache en mémoire : 2 formats de données
 - ▶ Ligne (row format)
 - ▶ Colonne compressée (column format)
- ▶ Transactions : rapide car moins d'index secondaires à maintenir
- ▶ Cohérence : Niveau *read commited* pour les requêtes

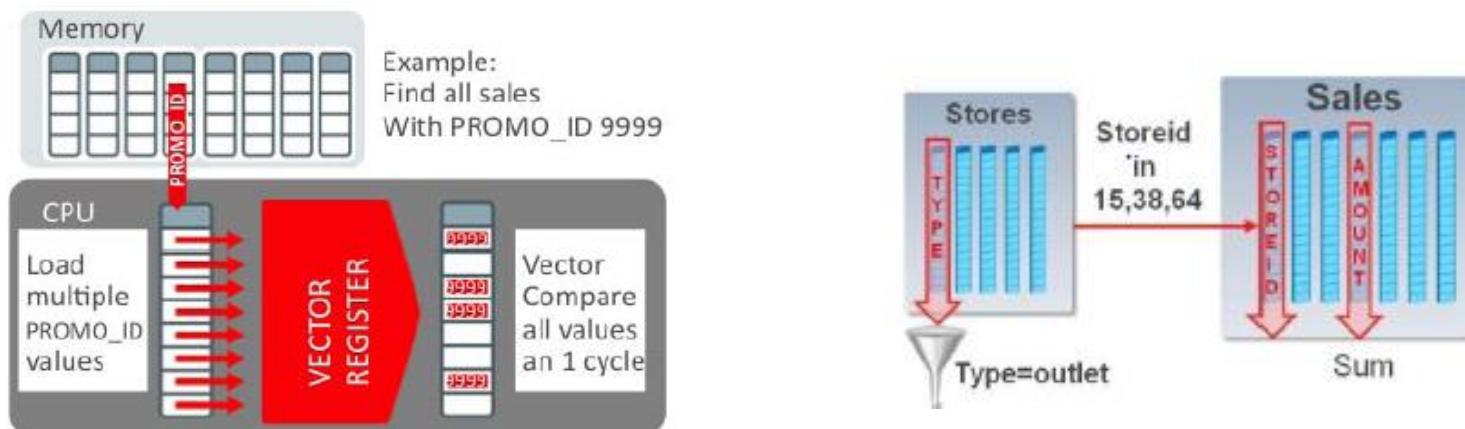


Oracle In Memory (2/3)

- ▶ Transparent et compatible avec une base relationnelle existante
- ▶ Contrôler quelles données sont mémoire
 - ▶ Une table
 - ▶ Une partition d'une table
 - ▶ Exple : les données des n derniers mois
 - ▶ Exclure certains attributs
 - ▶ Blob (image) non accédée par les requêtes analytiques
 - ▶ Recommandation basée sur les requêtes récentes
- ▶ Contrôler la mise en mémoire des données
 - ▶ Dès le début
 - ▶ Lors de la première lecture

Oracle In Memory (3/3) : Performance

- ▶ OLAP : filtrage d'une colonne très rapide
 - ▶ Select et Group By traités en parallèle
 - ▶ SIMD (single instruction on multiple data)
 - ▶ Jointure avec pré-sélection par **filtre** de Bloom



Ref : Oracle Database In-Memory: A Dual Format In-Memory Database, ICDE 2013

Perspective : HDFS / S3

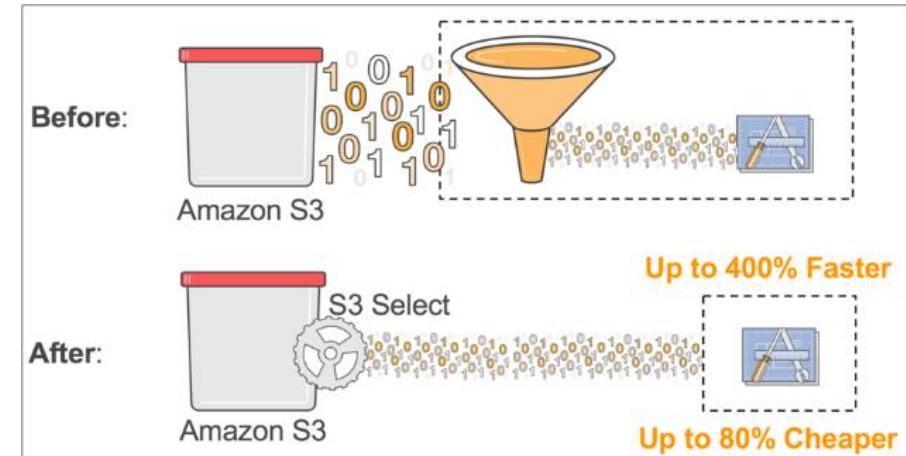
- ▶ Amazon S3 : Simple Storage Service
 - ▶ Solution cloud (Platform as a service) pour stocker des fichiers
 - ▶ Espace de stockage : bucket objet = (clé, tags, fichier)
 - ▶ Disponibilité : $d = 99.99\%$ sur une année
 - ▶ Hors service $< 1/(1-d) = 1/10^4$: inaccessible moins de 53mn par an
 - ▶ Durabilité
 - ▶ Réplication : perte de données $< 1 / 10^{11}$
 - ▶ Géo-réplication
 - ▶ Elasticité
 - ▶ Sécurité: données encryptées
 - ▶ Localité donnée/traitement au niveau d'une zone de disponibilité

S3 : coût

- ▶ Coût à l'usage: 1To = 25\$/mois
 - ▶ Plusieurs qualités de service selon les besoins
 - ▶ accès peu fréquent et disponibilité moindre (moitié prix),
 - ▶ archivage (1/5 du prix)
- ▶ Coût pour importer/exporter les données:
 - ▶ réduit si les transferts sont **internes** au cloud
- ▶ Coût inférieur à la location de machines exécutant HDFS
 - ▶ EC2 : 5\$/h avec disque 48To avec replication 3x et remplissage à 70%
 - ▶ 1To coûte 100\$/mois

S3 : select

- ▶ Principe : Filtrer des données avant de les lire pour réduire les transferts
- ▶ Concerne les données non binaires : CSV, texte
- ▶ Supporte les données compressées
- ▶ Traitement parallèle entre les objets stockés (cf. principe mapreduce)
 - ▶ Coût et temps de calcul réduit



Perspective : Stockage Delta io

- ▶ Données stockées au format parquet
- ▶ Méthode pour ajouter modifier des données
- ▶ Requêtes analytiques
- ▶ Versions « time travel »
- ▶ Facilite la gestion unifiée des données

Inconvénients

- ▶ incrément « gros grain » = 1 partition
 - ▶ ajout/suppr d'une partition
 - ▶ pas de modification d'une partition existante

Perspective : Google Big Query

- ▶ Désagrégation des couches de stockage et de calcul
 - ▶ Nécessite un réseau ultra rapide entre les couches
- ▶ Redimensionnement dynamique
 - ▶ Indépendamment pour le stockage ou le calcul
 - ▶ Meilleur rapport performance/coût

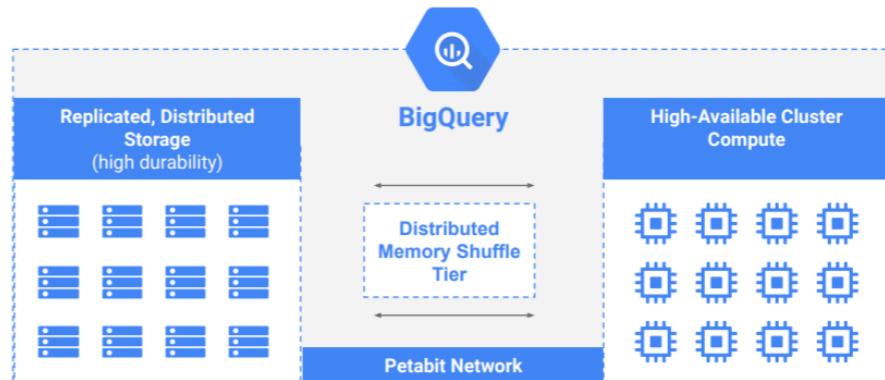


Figure 1: Disaggregated storage, memory, and compute

Perspective : Microsoft Socrates

- ▶ Solution cloud basée sur le SGBD SQL Server. Séparation en 4 services :
- ▶ Traitement des requêtes
 - ▶ Disponibilité par réPLICATION primary/secondary
- ▶ Gestion des logs (XLOG)
 - ▶ Durabilité : stockage cher, petites quantités
- ▶ Serveurs de pages de données
 - ▶ Scalabilité, performance (stockage sur SSD)
- ▶ Backup des données
 - ▶ Durabilité : stockage à faible coût (disques durs), grandes quantités

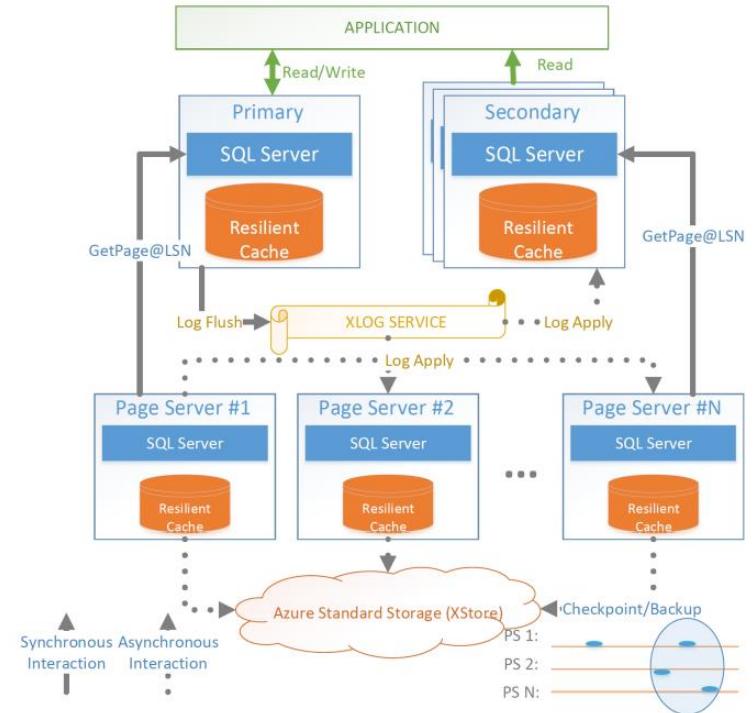


Figure 2: Socrates Architecture

Conclusion

- ▶ HDFS : solution distribuée et parallèle pour l'accès efficace à des fichiers
 - ▶ Très grands fichiers, format quelconque
 - ▶ Disponibilité : tolérance aux pannes, accès rapide
- ▶ Formats efficaces pour stocker des données
 - ▶ Encodage par colonne et compression
 - ▶ Stockage partitionné : lectures sélectives
- ▶ Cache de données en mémoire
- ▶ Applications ciblées : analyse de données à large échelle
 - ▶ Ecosystème Hadoop : SQL Hive, dataStore Hbase
 - ▶ Plateforme Spark