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

Approches distribuées

Michel Crucianu

(prenom.nom@cnam.fr)

http://cedric.cnam.fr/vertigo/Cours/RCP216/

Département Informatique

Conservatoire National des Arts & Métiers, Paris, France

2 septembre 2026

Le sujet de cette séance

- La séance précédente a montré comment repousser les limites d'une architecture centralisée (réduction du volume, Polars, Dask)
- Les solutions centralisées restent contraintes par la capacité d'une seule machine : mémoire, cœurs CPU, stockage local

→ Au-delà de cette capacité, il faut distribuer données et calculs sur un ensemble d'ordinateurs interconnectés

- Cette séance présente les fondements du calcul distribué : *MapReduce* (le modèle fondateur), *Apache Spark* (l'évolution dominante), *Ray* (approche orientée apprentissage automatique distribué)

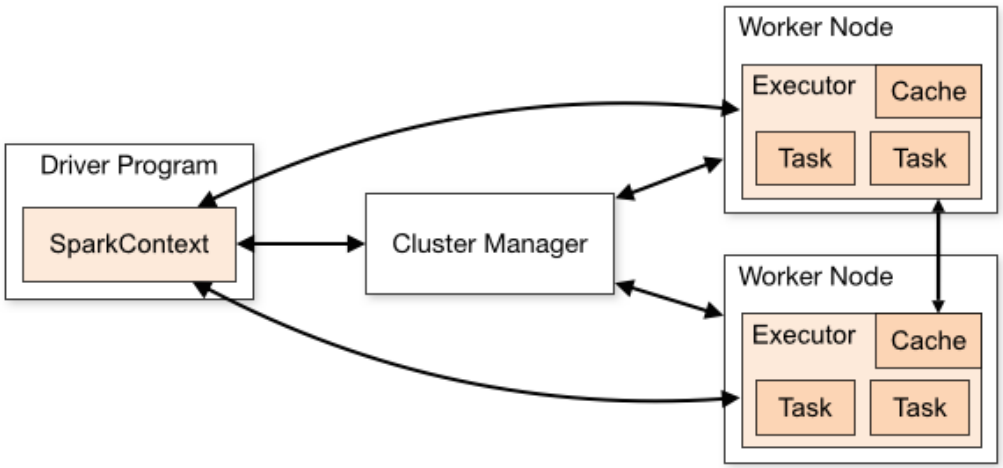
Plan du cours

Architecture d'un cluster

- Depuis les années 2010 : distribuer données et calculs sur un grand nombre d'ordinateurs standards bon marché, chacun avec processeur, mémoire vive et stockage local, interconnectés en réseau haut débit (*cluster*)
- Rapport coût/performance : un *cluster* de 1000 serveurs à 1000 € chacun offre une puissance de traitement et une capacité de stockage bien supérieures à un serveur unique à 1 000 000 €, pour un coût comparable

→ **Élasticité** : ajout ou retrait de nœuds selon les besoins

- **Localité des données** : les données sont bien plus volumineuses que le code ⇒ on envoie le code vers les données ; chaque nœud travaille sur les données stockées localement



Tolérance aux pannes

- Un grand nombre d'ordinateurs standards augmente inévitablement la probabilité de panne d'au moins un nœud
  - Sur un cluster de 1000 machines dont chacune tombe en panne en moyenne une fois tous les trois ans : environ une panne par jour

- Deux mécanismes fondamentaux
  - **Réplication** : chaque fragment de données stocké sur plusieurs nœuds (en général 3 dans HDFS) ; en cas de panne, les données restent accessibles grâce aux copies
  - **Partitionnement fin des calculs** : le travail est découpé en tâches élémentaires traçables ; en cas de panne, seules les tâches concernées sont réaffectées, sans reprendre le calcul depuis le début

Plan du cours

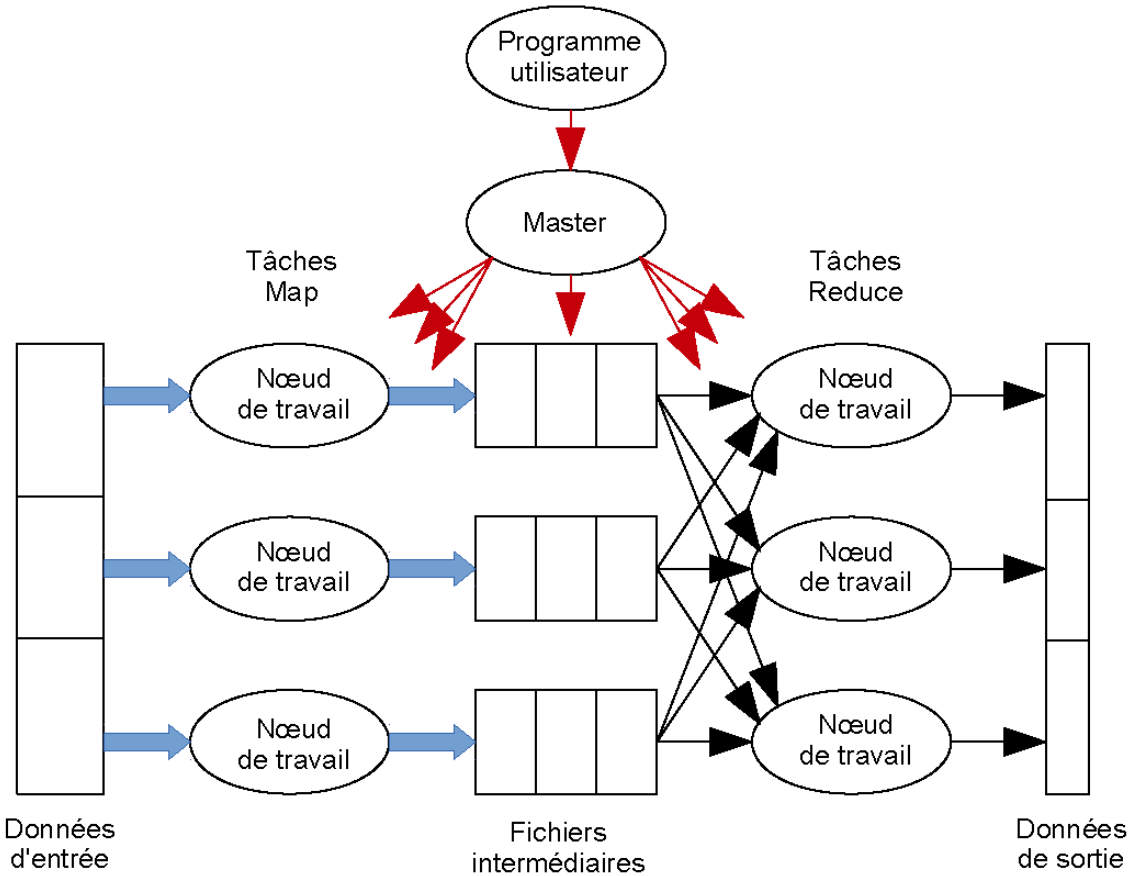
MapReduce : principe

- Modèle qui a posé les fondations du traitement distribué à grande échelle : proposé par Google en 2004, implémenté en *open source* dans Hadoop
- Reste une référence conceptuelle

- Décomposition de tout traitement en deux types de tâches
  - Tâche **Map** : reçoit un fragment de données (depuis le disque), produit une séquence de paires [clé, valeur] ; toutes les tâches Map sont indépendantes et parallélisables
  - Tâche **Reduce** : reçoit, pour une clé donnée, toutes les valeurs associées par les Map (après regroupement ou *shuffle*), les combine et stocke le résultat sur disque

- Le programmeur écrit seulement *Map* et *Reduce* ; le *framework* distribue les données, affecte les tâches, gère les pannes et collecte les résultats

MapReduce : exécution d'un programme



MapReduce : exemple 1 – comptage de mots (*Word Count*)

Tâche à réaliser : compter le nombre d'occurrences de chaque mot dans une grande collection de documents découpée en fragments (un fragment par nœud)

- **Map** : chaque nœud parcourt les mots de son fragment et émet une paire [mot, 1] pour chaque occurrence
- **Shuffle** : les paires sont regroupées par clé (mot) et acheminées vers les nœuds *Reduce* responsables de chaque mot
- **Reduce** : pour chaque mot, addition de toutes les valeurs 1 reçues, stockage de [mot, nombre_total]

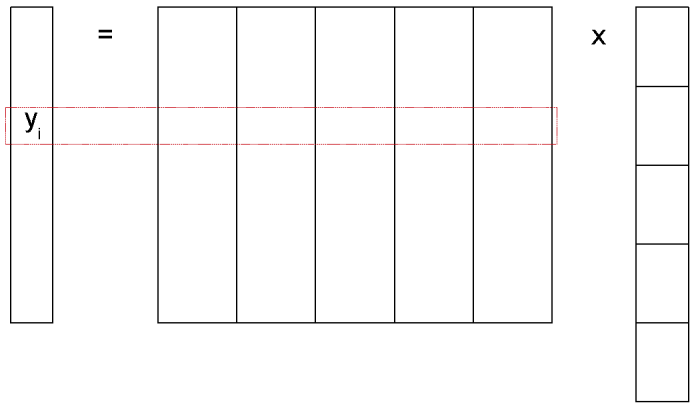
Optimisation : l'addition étant associative et commutative, chaque *Map* peut comptabiliser localement les occurrences dans son fragment et n'émettre qu'une seule paire [mot, n_local] par mot $\rightarrow$ réduit fortement le volume transféré lors du *shuffle*

MapReduce : exemple 2 – multiplication matrice × vecteur

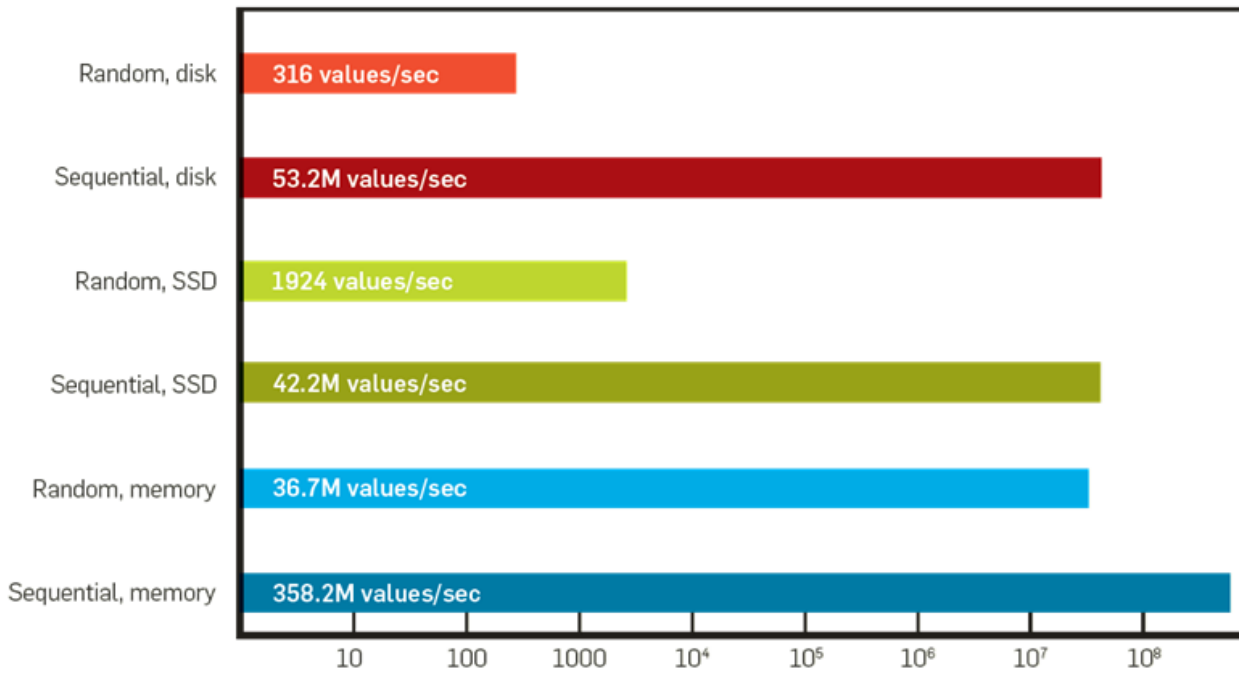
Tâche à réaliser : $\mathbf{y} = \mathbf{M} \times \mathbf{x}$, $y_i = \sum_{j=1}^{n} m_{ij}\, x_j$, avec $\mathbf{M}$ trop grande pour tenir dans la mémoire d'un seul nœud

1. $\mathbf{M}$ découpée en groupes de colonnes (un groupe tient en mémoire sur un nœud) ; $\mathbf{x}$ découpé en groupes de lignes correspondants
2. **Map** : chaque nœud reçoit un fragment de $\mathbf{M}$ et le fragment correspondant de $\mathbf{x}$; pour chaque m_{ij}, calcule $m_{ij}x_j$ et émet $(i,\, m_{ij}x_j)$
3. **Shuffle** : les paires sont regroupées par indice i
4. **Reduce** : pour chaque i, addition de toutes les valeurs $m_{ij}x_j$ reçues pour obtenir y_i

Optimisation : comme pour *Word Count*, pré-sommer les contributions partielles de chaque *Map* avant le *shuffle*



Hiérarchie de stockage et coût des accès disque



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

- La mémoire vive est plusieurs ordres de grandeur plus rapide que le disque

→ Au cœur de la différence de conception entre *MapReduce* et Spark

Limites de *MapReduce* pour les algorithmes itératifs

- Les algorithmes de fouille de données sont souvent itératifs (k-moyennes, descente de gradient, ACP par la méthode des puissances, etc.)
- **Stockage obligatoire des résultats intermédiaires** : le mécanisme de reprise sur panne exige que les résultats de chaque *Reduce* soient écrits sur disque (HDFS) avant l'itération suivante
  - Pour k itérations : k aller-retours disque, surcoût pouvant être de plusieurs ordres de grandeur supérieur au calcul effectif
  - Exemple : 1000 itérations, 1064 Mo résultats intermédiaires stockés sur disque à chaque itération $\Rightarrow$ env. 11 h de surcoût dû aux accès disque (2 accès par itération)
- **Niveau d'abstraction bas** : chaque itération doit être reformulée en *Map*/*Reduce*, puis les itérations enchaînées manuellement $\Rightarrow$ code complexe, difficile à maintenir

$\rightarrow$ Ces limitations ont motivé le développement de **Spark**

Plan du cours

Apache Spark : origine et objectif

- https://spark.apache.org – né en 2009 à l'UC Berkeley AMPLab
- **Objectif** : garder les avantages de *MapReduce* (tolérance aux pannes, parallélisme automatique, scalabilité) tout en éliminant ses limites pour les algorithmes itératifs
- Aujourd'hui, le *framework* de traitement de données distribuées dominant dans l'industrie

Spark : données en mémoire et *lineage*

- **Données en mémoire** : Spark conserve les résultats intermédiaires en mémoire vive des nœuds de calcul (autant que possible), évitant les allers-retours disque entre itérations
  - → Accélération de 10 à 100 fois par rapport à *MapReduce* classique sur les algorithmes itératifs

- **Lignage** (historique des opérations, *Lineage*) : Spark conserve le graphe des transformations ayant produit chaque donnée intermédiaire
  - En cas de panne, seules les données perdues sont recalculées à la demande à partir des dernières données stables, en rejouant les transformations sur un autre nœud
  - → Tolérance aux pannes sans stockage non volatil des données intermédiaires

Structures de données : RDD

- *Resilient Distributed Dataset* (RDD), introduit dans la v1.0 : structure fondamentale de Spark
  - Collection de données **partitionnée** et **distribuée** sur les nœuds, conservée autant que possible en mémoire
  - **Immuable** : toute transformation produit un nouveau RDD

- Deux catégories d'opérations
  - **Transformations** (`map`, `filter`, `flatMap`, `groupByKey`, `reduceByKey`…) : définissent une nouvelle opération, aucun calcul n'est déclenché (évaluation paresseuse ou *lazy*)
  - **Actions** (`collect`, `count`, `reduce`, `saveAsTextFile`…) : déclenchent l'exécution effective du graphe de transformations accumulées

Structures de données : Dataset / DataFrame

- Introduites v1.6 / v2.0 : abstractions de plus haut niveau que RDD, ajoutent
  - **Typage fort** et représentation binaire compacte (format *Tungsten*), plus efficace que la représentation *JavaObjects* des RDD
  - **Optimiseur d'exécution** (*Catalyst*) : analyse et optimise le plan d'exécution logique (fusion d'opérations, filtres appliqués au plus tôt, réordonnancement de jointures, etc.), génère un plan physique efficace
  - Une API **SQL** générale

- Une *DataFrame* est un *Dataset* organisé en colonnes nommées (analogue à une table relationnelle) ; en PySpark, `DataFrame` = `Dataset[Row]`

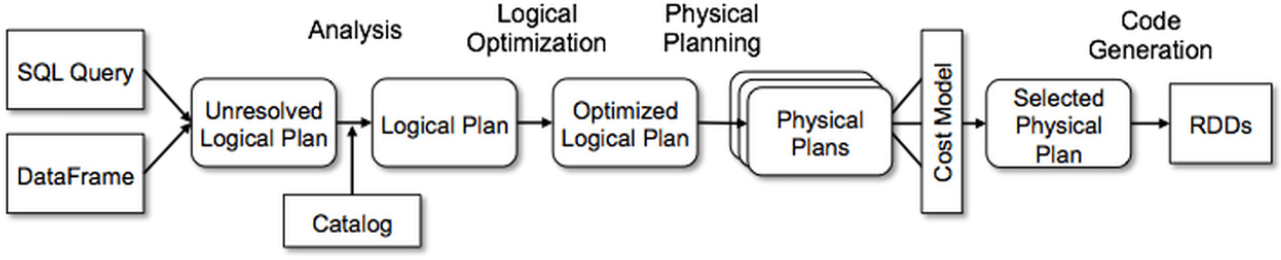


FIG. – Optimisation de l'exécution par *Catalyst* ; s'applique à tout traitement complexe de *DataFrame*, non seulement aux requêtes SQL.

Spark : recommandation pratique

→ Pour tout nouveau développement, préférer l'API **DataFrame / Dataset** à l'API RDD

- Plus expressive, bénéficie des optimisations de *Catalyst*, performances généralement supérieures

- L'API RDD reste utilisable pour des opérations de bas niveau qui ne s'expriment pas naturellement en colonnes

Spark : déploiement

- **Driver** : contrôle la logique applicative, construit le graphe des transformations, le soumet au *cluster*, collecte les résultats – doit tourner sur un nœud du *cluster* pour minimiser la latence
- **Executors** (*worker nodes*) : exécutent les tâches assignées, stockent les partitions en mémoire ou sur disque local
- **Gestionnaire de cluster** : alloue les ressources
  - *Standalone* : intégré à Spark, simple, sans intégration externe
  - *Hadoop YARN* : partage du *cluster* avec d'autres jobs, standard en environnement Hadoop
  - *Apache Mesos* : généraliste, moins utilisé aujourd'hui
  - *Kubernetes* : gestion de conteneurs, mode *de facto* pour le *cloud* actuel
- Applications isolées (chacune dans sa propre JVM), communication uniquement via fichiers
- Langages : Scala (natif), Java (*bytecode* compatible), Python (PySpark), R (SparkR, sparklyr)

Spark : bibliothèques de l'écosystème

MLlib	Apprentissage statistique distribué (classification, régression, clustering, réduction de dimension, recommandation, pipelines)
Structured Streaming	Traitement de flux temps réel, même API que les DataFrames *batch*
GraphX / GraphFrames	Calculs sur graphes distribués (centralités, composantes connexes, PageRank, détection de communautés)
SparkNLP	Traitement du langage naturel distribué (tokenisation, NER, plongements lexicaux, BERT, LLM)
Spark SQL	Requêtes SQL sur DataFrames, connecteurs Hive, JDBC, Parquet

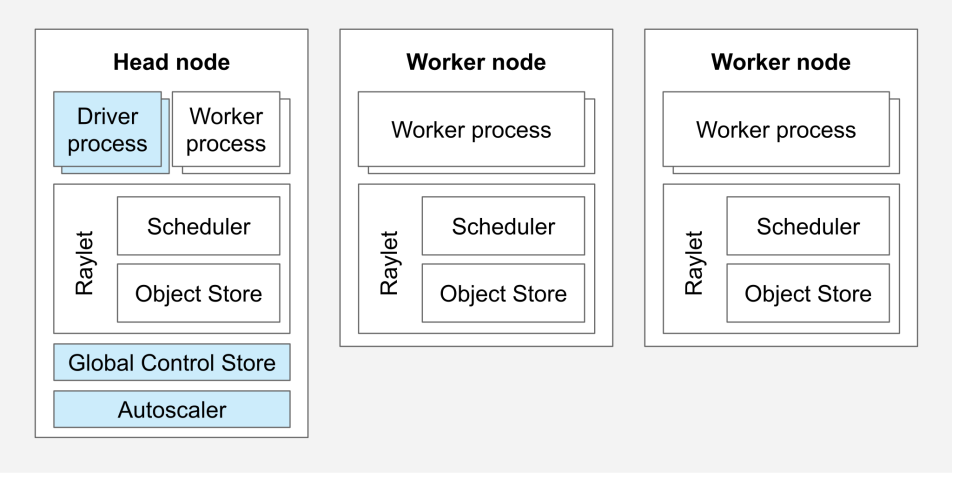
Plan du cours

Ray : motivations

- Ray : *framework* distribué développé depuis 2017 (UC Berkeley, puis Anyscale), écrit en C++ avec API Python
- Objectif différent de Spark : parallélisation d'algorithmes Python arbitraires, en particulier pour l'apprentissage profond, plutôt que le traitement de données tabulaires en *batch*
- Cas d'usage où Spark peine
  - **Entraînement distribué** de modèles profonds : communication fine entre *workers* (gradients, paramètres), ordonnancement dynamique
  - **Apprentissage par renforcement** (RL) : simulation, entraînement GPU, orchestration asynchrone
  - **Service de modèles** (*model serving*) à faible latence, état maintenu entre requêtes
  - Recherche d'**hyperparamètres** par essais parallèles massifs

Ray : architecture

- Modèle d'acteurs distribués, trois primitives fondamentales
  - **Task** (`@ray.remote` sur une fonction) : sans état (*stateless*), exécution asynchrone ; les tâches peuvent dépendre les unes des autres (graphe de calcul dynamique)
  - **Actor** (`@ray.remote` sur une classe) : avec état (*stateful*), état maintenu entre appels (ex. modèle chargé en mémoire, *buffer* de *replay* RL)
  - **Object** : données immuables dans l'*Object Store* distribué (mémoire partagée via Apache Plasma), accessibles depuis tout nœud



Ray : exemple

```
import ray
ray.init()   # démarre un cluster (ou se connecte à un cluster existant)

@ray.remote
def calculer_carre(x):
    return x * x

# Exécution asynchrone : retourne un ObjectRef, pas encore la valeur
ref = calculer_carre.remote(5)
print(ray.get(ref))   # bloque jusqu'à disponibilité -> 25

# Exécution massivement parallèle : 1000 tâches en parallèle
refs = [calculer_carre.remote(i) for i in range(1000)]
resultats = ray.get(refs)
```

Ray : bibliothèques de l'écosystème

Ray Train	Entraînement distribué (PyTorch, TensorFlow, XGBoost), *checkpoints* et reprise sur panne automatiques
Ray Tune	Recherche distribuée d'hyperparamètres (*grid/random search*, optimisation bayésienne, algorithmes évolutionnaires)
Ray Serve	Service de modèles à faible latence, scalabilité automatique, composition de *pipelines*
RLlib	Référence pour l'apprentissage par renforcement distribué (utilisée par OpenAI, DeepMind)

Ray : positionnement par rapport à Spark

- Ray et Spark ne sont pas en concurrence directe mais répondent plutôt à des besoins complémentaires
  - **Spark** : préparation de données à grande échelle (ETL, agrégations, jointures sur des To), application de modèles en *batch*, *workflows* tabulaires
  - **Ray** : développement/entraînement de modèles complexes (*deep learning*, RL), recherche d'hyperparamètres, service de modèles en production

- Combinaisons possibles : *Spark on Ray*, *Ray on Spark* $\rightarrow$ combiner les points forts des deux systèmes dans une même *pipeline*

$\rightarrow$ Dans ce cours, les TP utilisent principalement **Spark** (couvrant l'ensemble des sujets) ; Ray sera mentionné ponctuellement

Plan du cours

Comparaison de plateformes (1/2)

	Pandas	**Polars**	**Dask**	**Spark**	**Ray**	**MapReduce**
Distribution	Non	Non	Optionnelle	Oui	Oui	Oui
Multi-cœur	Non	Oui (auto)	Oui	Oui	Oui	Oui
Données $>$ RAM	Non	Partiel	Oui	Oui	Oui	Oui
Tolérance pannes	Non	Non	Partielle	Oui (*lineage*)	Oui (*lineage*)	Oui (HDFS)

Comparaison de plateformes (2/2)

	Pandas	**Polars**	**Dask**	**Spark**	**Ray**	**MapReduce**
Alg. itératifs	Oui	Oui	Oui	Très efficace	Très efficace	Lent (disque)
ML intégré	Non	Non	Partiel	MLlib (complet)	Ray Train/Tune	Non
Streaming	Non	Non	Oui (limité)	Structured Streams	Non natif	Non
API Python	Oui	Oui	Oui	PySpark	Oui (natif)	Non natif
Point fort	Référence universelle	Rapidité mono-machine	Pandas sur *cluster*	*Batch* + ML distribués	ML avancé, RL	Robustesse maturité

Critères de choix en pratique

1. Les données tiennent en RAM sur une seule machine ? Préférer **Polars** (rapidité) ou Pandas (familiarité)
2. Les données dépassent la RAM mais une seule machine suffit ? Préférer **Dask**
3. Les données sont massives et on veut distribuer des calculs NumPy/Pandas ? Préférer **Dask**
4. Les données sont massives et les calculs sont du *batch* / ML / *streaming* ? Préférer **Spark**
5. Entraînement de modèles complexes ou recherche d'hyperparamètres ? Préférer **Ray**, éventuellement en combinaison avec Spark pour la préparation des données
6. Infrastructure existante basée sur Hadoop ? **Spark** est le choix naturel