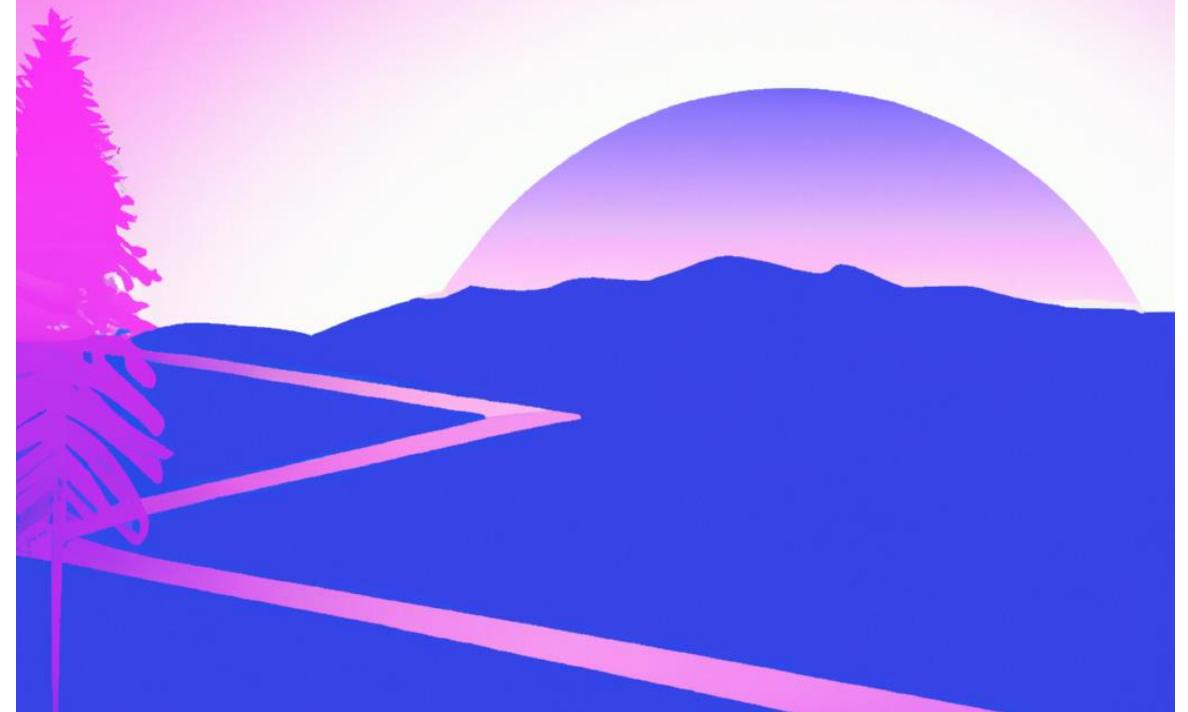


Spark

Prise en main

AVRIL 2023



Objectifs de la formation

- Comprendre Spark, son architecture et ses composants.
- Utiliser Spark Shell pour interagir avec Spark.
- Développer en production avec Spark.
- Maîtriser la lecture/écriture de données.
- Comprendre les transformations de données et les actions.
- Utiliser Spark SQL pour manipuler des données tabulaires.
- Exploiter Spark SQL pour lire/écrire, explorer, et analyser des données.
- Apprendre Spark Streaming pour le traitement de flux en temps réel.

Présentations

- Ce qui vous semble pertinent (rôle, expérience, attentes...).
- Votre rapport à Spark dans votre travail.
- Votre background (pour orienter vers certains axes le discours et les échanges).



Programme



- 01 Introduction à Spark**
- 02 Fonctionnalités essentielles de Spark**
- 03 Spark SQL**
- 04 Spark streaming et Kafka**

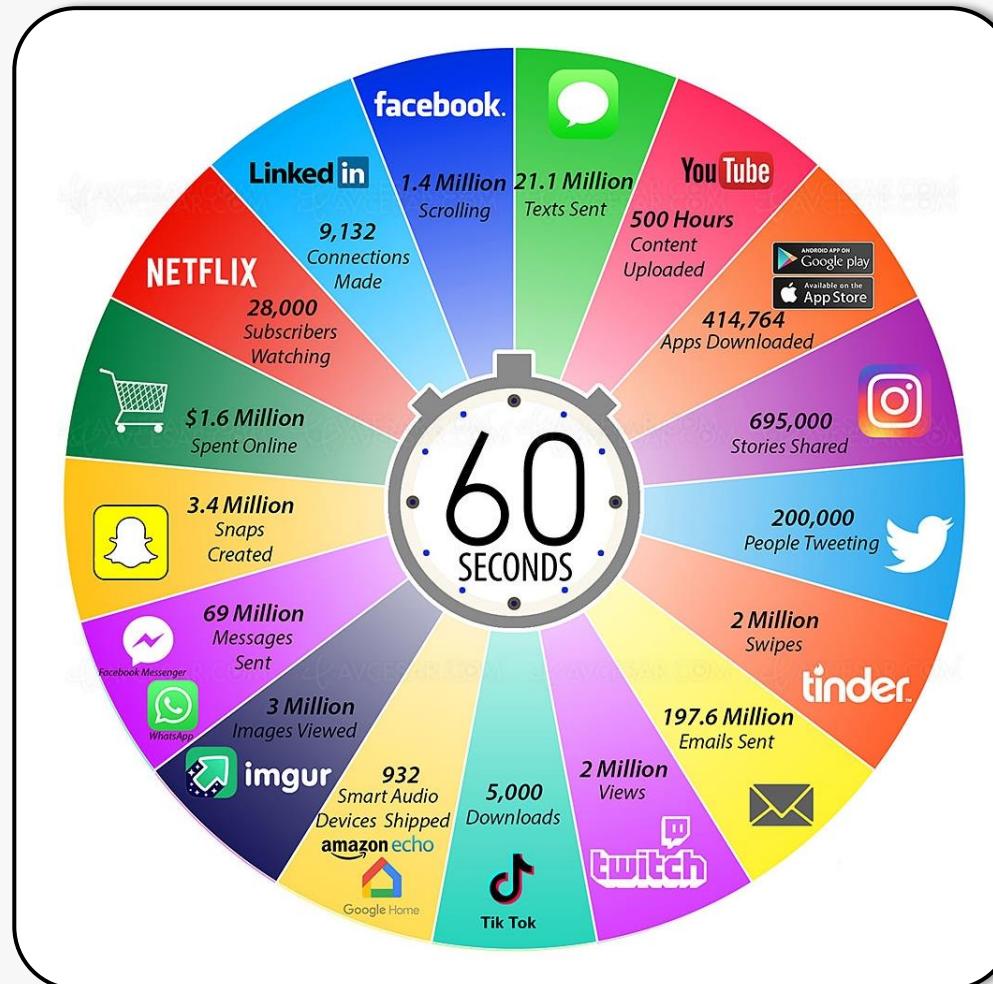
01

Introduction à Spark

Introduction à Spark

Quantité de données produites dans le monde

Que se passe-t-il sur Internet en 1 minute ? (2021)



Crédit : L. Lewis

Introduction à Spark

Ordres de grandeur

Unité	Abb.	Taille en octets	Exemple
1 octet		1	4 octets \simeq Un caractère au format utf-8
1 kilooctet	ko	10^3	20 ko \simeq Mots prononcés par une personne, par jour
1 mégaoctet	Mo	10^6	3 Mo \simeq Texte d'une édition quotidienne du New York Times
1 gigaoctet	Go	10^9	1 Go \simeq Mots prononcés par une personne durant une vie
1 téraoctet	To	10^{12}	593 To \simeq Mots prononcés par l'ensemble de la population mondiale, par jour
1 pétaoctet	Po	10^{15}	4 Po \simeq Données générées par Facebook, par jour
1 exaoctet	Eo	10^{18}	605 Eo \simeq Données générées par le télescope SKA, par jour
1 zétaoctet	Zo	10^{21}	40 Zo \simeq Données générées par le web en 2020, par an

Introduction à Spark

Une définition classique du Big Data

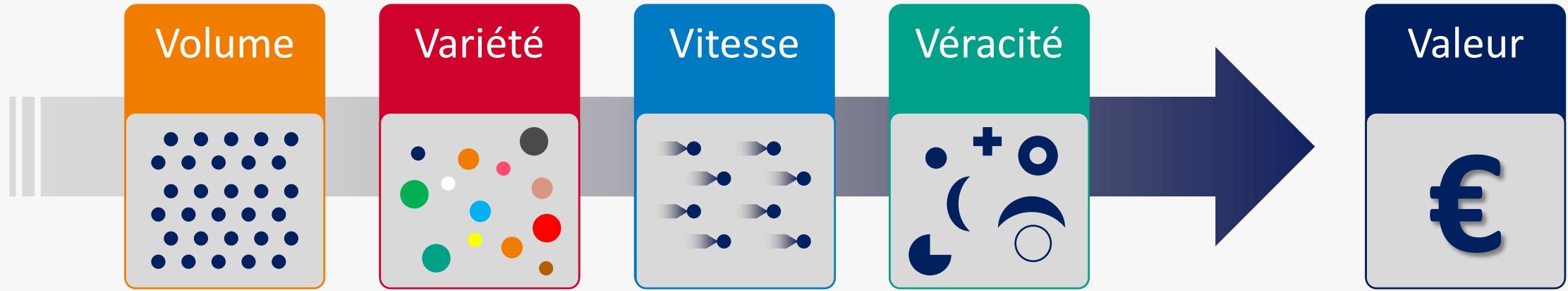


Les "big data" sont des informations de grand volume, de grande vitesse et/ou de grande variété qui exigent des formes rentables et innovantes de traitement de l'information permettant d'améliorer la compréhension, la prise de décision et l'automatisation des processus.

<https://www.gartner.com/en/information-technology/glossary/big-data>

Introduction à Spark

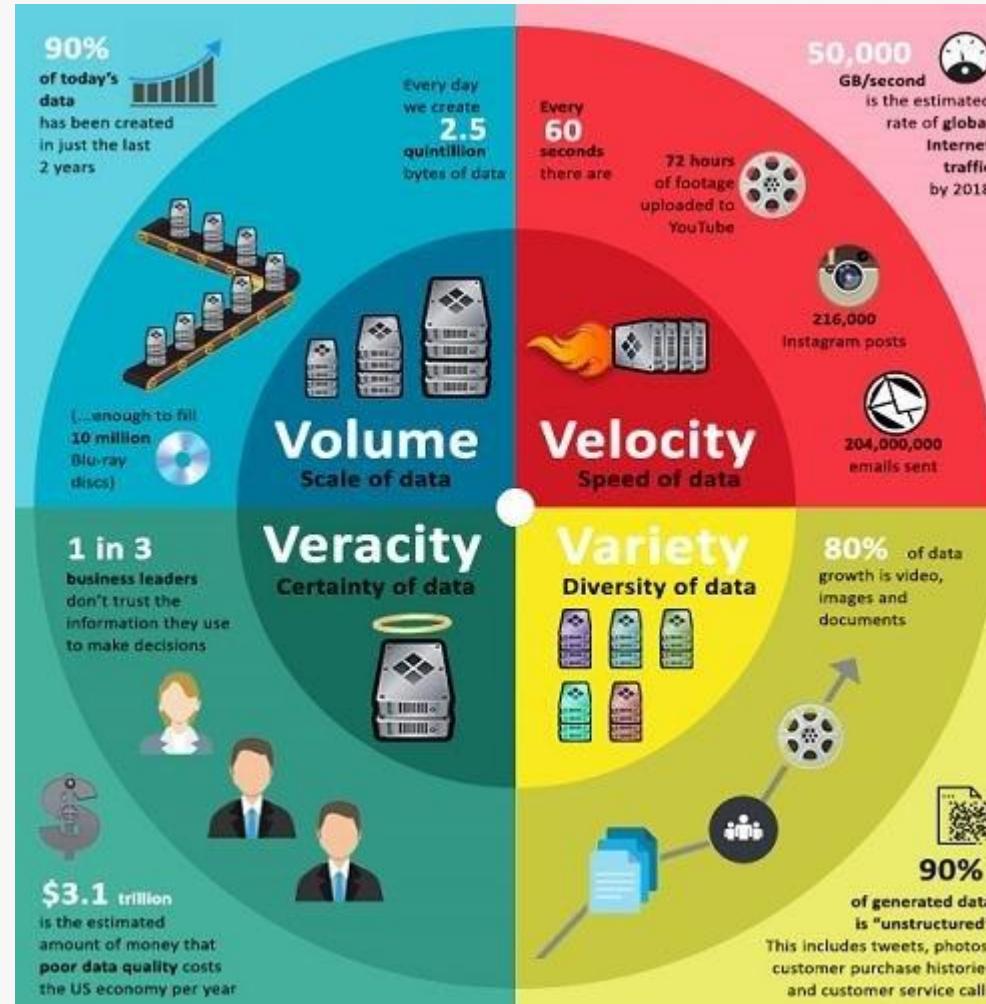
Les 3/4/5 V



- **Volume** : Croissance continue de la quantité de données à exploiter, souvent en teraoctets voire en petaoctets.
- **Variété** : Traitement des données sous forme structurée et non structurée, devant faire l'objet d'une analyse dans leur ensemble (bases de données relationnelles, textes, données de capteurs, sons, vidéos...).
- **Vélocité** : Utilisation des données en temps réel dans de multiples applications (détection de fraude...).
- **Véracité** : Gestion de la fiabilité et de la véracité des données prédictives.

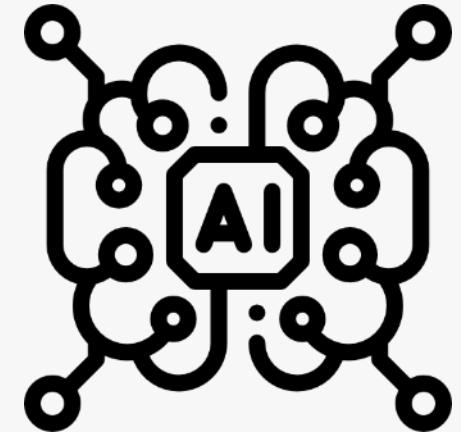
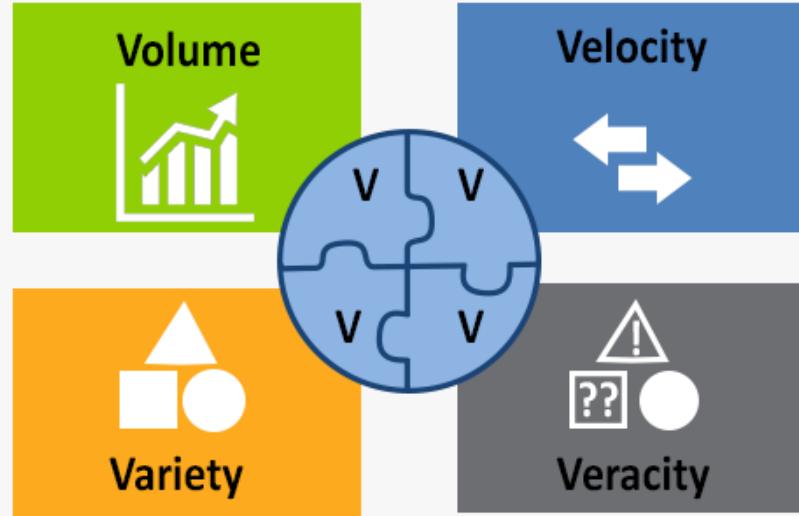
Introduction à Spark

Les 3/4/5 V en chiffres



Introduction à Spark

Inscription dans la révolution numérique



La capacité à recueillir, stocker, gérer et analyser de grandes quantités de données est devenue essentielle pour les entreprises et les organisations qui souhaitent rester compétitives et innovantes. Le Big Data alimente des avancées technologiques majeures, contribuant à des découvertes scientifiques, à l'amélioration des produits et services, et à la transformation des modèles commerciaux. En ce sens, le Big Data est plus qu'une mode, il est une réalité fondamentale de l'ère numérique.

Introduction à Spark

Spark dans tout cela

- Le développement du Web et d'Internet a engendré une explosion de la quantité de données informatiques produites.
- Cette croissance est si considérable que les ordres de grandeur associés sont difficiles à appréhender.
- La problématique du Big Data se pose souvent quand on parle de passer à l'échelle ("scale") une activité, en lien avec ce développement.
- Le stockage, le traitement et l'analyse de ces données massives nécessitent une approche spécifique.
- Cette approche implique la parallélisation du stockage et des traitements sur plusieurs machines distinctes.
- MapReduce, Hadoop MapReduce et Spark sont des moteurs de traitement adaptés aux volumes de données massifs, qui sont basés sur une approche de traitement distribué.



Introduction à Spark

Projet SETI@home



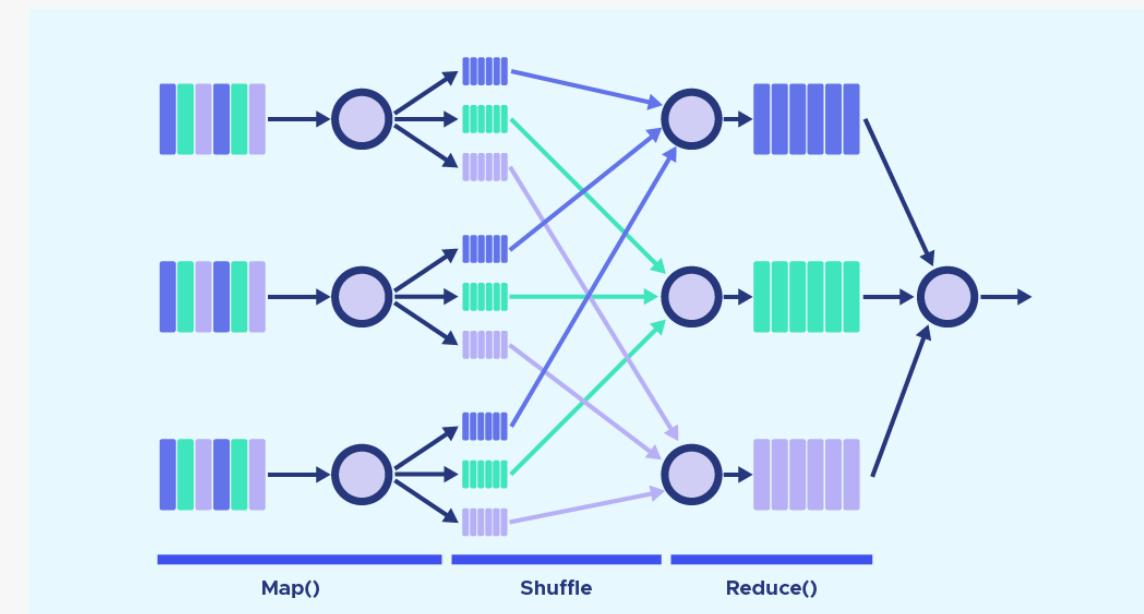
- Initiative de calcul distribué dans le domaine de la recherche en astronomie, plus spécifiquement dans la recherche de signaux extraterrestres.
 - Vise à analyser les signaux radio provenant de l'espace à la recherche de preuves d'une intelligence extraterrestre.
 - Utilise les données recueillies par le radiotélescope d'Arecibo et d'autres radiotélescopes à travers le monde.
- Ces données sont découpées en petits morceaux appelés "unités de travail".
- Les unités de travail sont ensuite distribuées à un grand nombre d'ordinateurs personnels via Internet.
- Les ordinateurs personnels analysent ces données lorsqu'ils sont inactifs, en utilisant une application logicielle fournie par le projet SETI@home.
- Une fois l'analyse terminée, les résultats sont renvoyés au projet SETI@home pour être examinés et interprétés par les chercheurs.

Introduction à Spark

De MapReduce à Spark, premières descriptions

MapReduce :

- MapReduce est un modèle de programmation et un framework de traitement de données largement utilisé pour le traitement parallèle de gros volumes de données sur des clusters de serveurs.
- Il divise les tâches de traitement de données en deux étapes principales : map et reduce.
- L'étape de map consiste à traiter et à transformer les données en paires clé-valeur.
- L'étape de reduce agrège et traite les données résultantes de l'étape de map.
- MapReduce est efficace pour le traitement de données structurées et semi-structurées, mais peut être relativement lent pour les traitements itératifs et interactifs.



Introduction à Spark

De MapReduce à Spark, premières descriptions

Hadoop MapReduce :

- Hadoop MapReduce est une implémentation du modèle MapReduce développée par la fondation Apache Hadoop.
- Il fournit une infrastructure de calcul distribué pour le traitement de données sur des clusters de serveurs.
- Hadoop MapReduce est conçu pour fonctionner sur de grands ensembles de données stockés dans le système de fichiers distribué Hadoop HDFS (Hadoop Distributed File System).
- Il est extensible, tolérant aux pannes et capable de traiter des données de manière scalable.



Introduction à Spark

De MapReduce à Spark, premières descriptions

Spark :

- Spark est un framework de traitement de données en mémoire conçu pour le traitement rapide et efficace de gros volumes de données.
- Contrairement à MapReduce, Spark conserve les données en mémoire chaque fois que cela est possible, ce qui réduit considérablement les temps d'accès aux données.
- Spark prend en charge un large éventail de charges de travail, y compris le traitement de flux de données, le traitement de graphiques, l'apprentissage automatique et le traitement interactif.
- Il fournit des APIs faciles à utiliser pour le développement d'applications de traitement de données en Python, Scala, Java et R.



Introduction à Spark

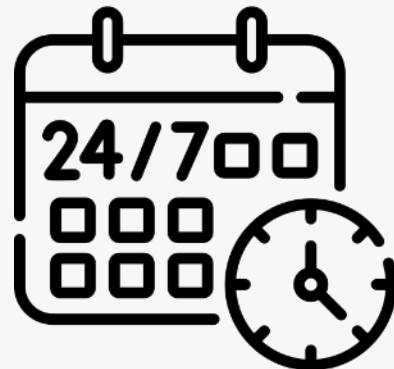
Systèmes distribués : notions importantes



Evolutivité (Scalability)



Latence (Latency)



Disponibilité (Availability)



Tolérance aux pannes (Fault Tolerance)

(+ Consistance, Partitionnement & Communication)

Introduction à Spark

Evolutivité

- Capacité d'un système à gérer efficacement une **charge de travail croissante** en ajoutant des ressources matérielles ou en élargissant son infrastructure sans compromettre ses **performances**.

Enjeux :

Répondre à la croissance des besoins

L'évolutivité vise à permettre au système de répondre à la demande croissante des utilisateurs et à s'adapter à l'augmentation du volume de données ou de trafic.

Maintenir la performance

L'évolutivité doit permettre au système de maintenir des performances acceptables même lorsque la charge de travail augmente, en garantissant des temps de réponse rapides et une disponibilité élevée.

Optimiser l'utilisation des ressources

L'évolutivité cherche à utiliser efficacement les ressources matérielles disponibles, en répartissant équitablement la charge de travail sur les différentes unités de traitement.

Introduction à Spark

Evolutivité

- **Caractéristiques de l'évolutivité dans les systèmes distribués :**

Scalabilité et performance linéaire

Idéalement, un système évolutif devrait présenter une amélioration linéaire de ses performances lorsque des ressources supplémentaires sont ajoutées. Cela signifie que le système devrait devenir proportionnellement plus rapide avec l'ajout de ressources.

Flexibilité géographique

L'évolutivité doit permettre la mise en place de centres de données supplémentaires dans différentes régions géographiques afin de réduire la latence et d'améliorer la disponibilité pour les utilisateurs répartis sur différentes zones géographiques.

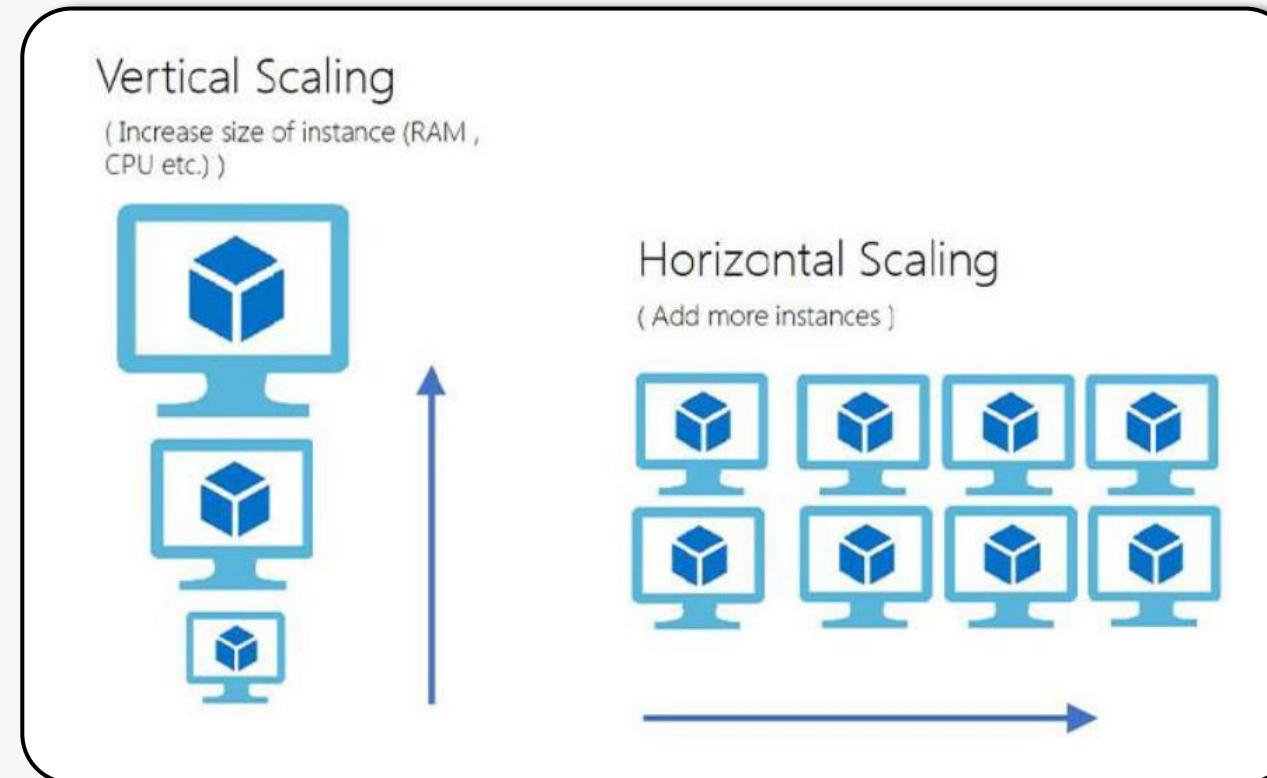
Faible coût administratif

L'évolutivité devrait être réalisée de manière à minimiser les coûts administratifs associés à la gestion et à la maintenance du système distribué dans son ensemble.

Introduction à Spark

Scaling vertical et horizontal

L'évolutivité peut être réalisée à travers la mise à niveau des ressources matérielles d'une seule unité (scaling vertical) ou en ajoutant des unités de traitement supplémentaires (scaling horizontal).



Introduction à Spark

Scaling vertical et horizontal

Caractéristique	Scaling Vertical (Up)	Scaling Horizontal (Out)
Évolutivité	Extension de la capacité d'une seule machine	Extension de la capacité par ajout de machines
Coût matériel	Élevé	Relativement plus bas
Limites	Peut atteindre des limites en termes de performances	Moins de limites en termes de performances
Exemples	Ajout de mémoire, augmentation de la puissance CPU	Ajout de serveurs, clusters
Flexibilité	Moins flexible en termes d'ajustement dynamique	Plus flexible, permet une croissance plus dynamique

Introduction à Spark

Latence

- Représente le délai ou le temps écoulé entre l'envoi d'une requête et la réception de la réponse associée.



- Il s'agit d'un aspect critique de la performance des systèmes distribués, car elle influence directement l'expérience utilisateur et la réactivité globale du système.
- Exemple : application de commerce électronique qui doit enregistrer des informations sur les produits et récupérer ces informations ultérieurement pour afficher les détails des produits aux utilisateurs. Quand l'écriture devient-elle disponible à la lecture ?

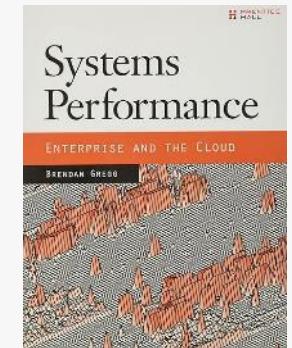
Introduction à Spark

Latence



Table 2.2 Example Time Scale of System Latencies

Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 µs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–317 years
OS virtualization system reboot	4 s	423 years
SCSI command time-out	30 s	3 millennia
Hardware (HW) virtualization system reboot	40 s	4 millennia
Physical system reboot	5 m	32 millennia



[Brendan Gregg, 2013](#)

Introduction à Spark

Disponibilité

- Capacité d'un système à rester opérationnel et accessible aux utilisateurs (même en cas de défaillance d'une partie du système, lié à la tolérance aux pannes, c.f. plus tard).
- La disponibilité mesure la proportion de temps pendant lequel un système est en état de fonctionnement et prêt à répondre aux demandes des utilisateurs.

	Par jour	Par mois	Par an
99 %	14,5 min	7,5 h	3,6 j
99,9 %	1,5 min	43,9 min	9 h
99,99 %	8 s	4,5 min	53 min
99,999 %	0,4 s	26 s	5 min

Introduction à Spark

Tolérance aux pannes

« La plupart des systèmes tombent un jour en panne »

- Capacité d'un système à maintenir un comportement défini malgré la survenue d'une panne.
- Un système fiable demeure opérationnel même en cas de dysfonctionnement de certains de ses composants.



Introduction à Spark

Paradigmes d'architecture dans les systèmes distribués

- **Shared-memory**

Plusieurs nœuds ou processus partagent l'accès à un pool commun de mémoire.

Cela permet une communication facile et un partage de données entre les processus.

Cependant, cela peut également entraîner des problèmes tels que la cohérence des données et des limitations d'évolutivité.

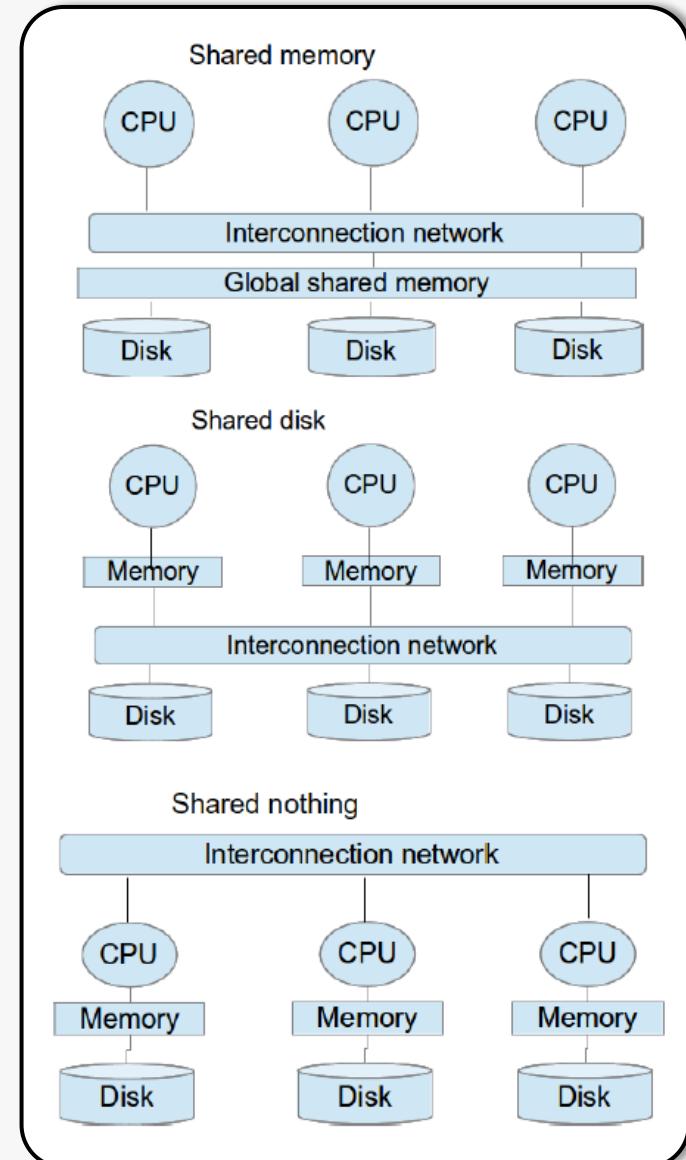
- **Shared-disk**

Plusieurs nœuds ont accès à un système de stockage partagé (disque) où les données sont stockées. Cela permet à plusieurs nœuds de lire et d'écrire sur le même magasin de données simultanément.

Couramment utilisé dans les systèmes de fichiers et les systèmes de bases de données distribués.

Offre une haute disponibilité et des capacités de partage de données, mais peut souffrir de goulets d'étranglement de performance dus à la contention pour les ressources disque.

Coût intéressant pour un cluster de petite à moyenne taille (comprenant généralement jusqu'à une dizaine de machines).



Introduction à Spark

Paradigmes d'architecture dans les systèmes distribués

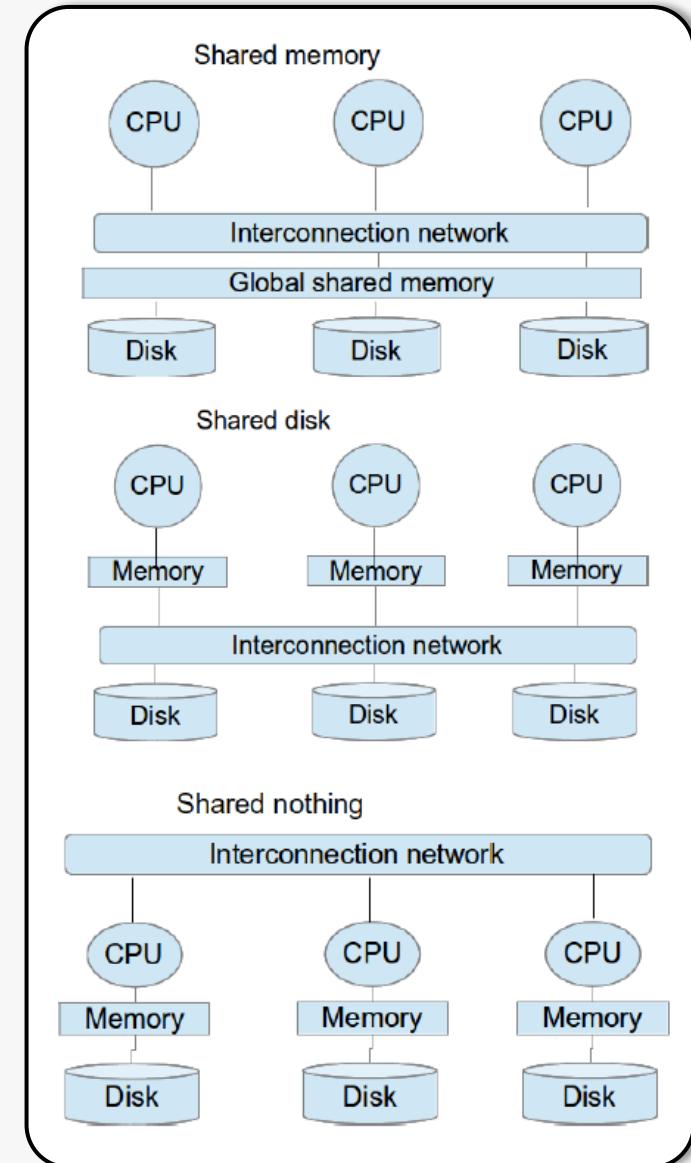
- **Shared-nothing**

Chaque nœud du système fonctionne de manière indépendante et ne partage pas de mémoire ou de stockage disque avec les autres nœuds.

Les nœuds communiquent en se passant des messages via un réseau.

Cette architecture offre une évolutivité et une tolérance aux pannes car la défaillance d'un nœud n'affecte pas les autres.

MapReduce et Hadoop sont des exemples de systèmes shared-nothing.



Introduction à Spark

Eclosion

- L'idée centrale derrière MapReduce était le traitement parallèle de données sur un grand nombre de machines tout en abstrayant la complexité associée à la gestion de ces traitements distribués
- MapReduce a inspiré (entre autres) la création des moteurs Apache Hadoop, et par la suite, Apache Spark.
- Il peut être intéressant de comprendre l'évolution de l'écosystème de traitement des mégadonnées de Google, de l'innovation interne de Google à l'émergence d'un écosystème global de traitement des données open source dynamique et en évolution constante.



Introduction à Spark

GFS

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power sup-

<https://static.googleusercontent.com/media/research.google.com/fr//archive/gfs-sosp2003.pdf>

Introduction à Spark

MapReduce

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical “record” in our input in order to compute a set of intermediate key/value pairs, and then

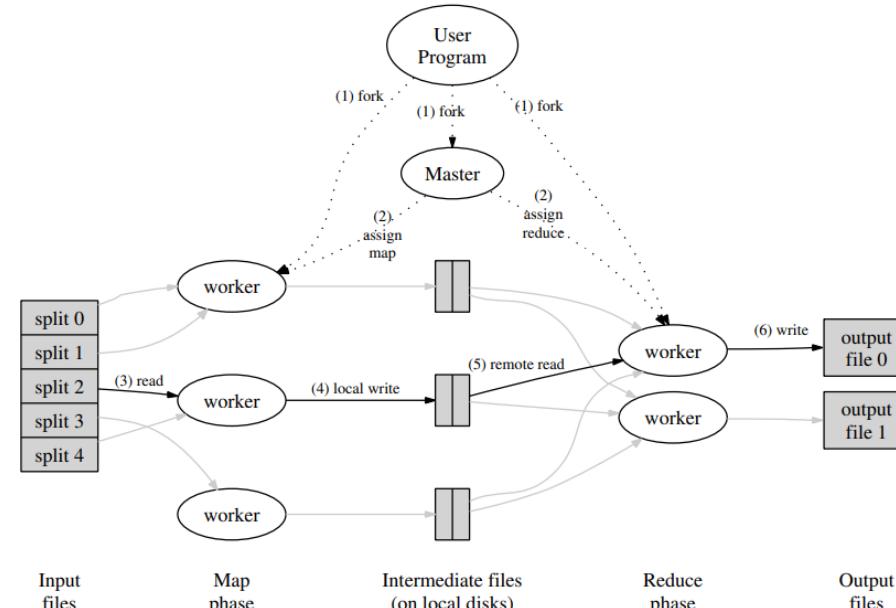


Figure 1: Execution overview

Introduction à Spark

TP MapReduce



Comprendre la fonction map
Comprendre la fonction reduce
MapReduce sur des paires clefs-valeurs
Wordcount (le Hello World de MapReduce)

Introduction à Spark

Sawzall

Interpreting the Data: Parallel Analysis with Sawzall

Rob Pike, Sean Dorward, Robert Griesemer, Sean Quinlan
Google, Inc.

Abstract

Very large data sets often have a flat but regular structure and span multiple disks and machines. Examples include telephone call records, network logs, and web document repositories. These large data sets are not amenable to study using traditional database techniques, if only because they can be too large to fit in a single relational database. On the other hand, many of the analyses done on them can be expressed using simple, easily distributed computations: filtering, aggregation, extraction of statistics, and so on.

We present a system for automating such analyses. A filtering phase, in which a query is expressed using a new procedural programming language, emits data to an aggregation phase. Both phases are distributed over hundreds or even thousands of computers. The results are then collated and saved to a file. The design—including the separation into two phases, the form of the programming language, and the properties of the aggregators—exploits the parallelism inherent in having data and computation distributed across many machines.

1 Introduction

Many data sets are too large, too dynamic, or just too unwieldy to be housed productively in a relational database. One common scenario is a set of many plain files—sometimes amounting to

<https://static.googleusercontent.com/media/research.google.com/fr//archive/sawzall-sciprog.pdf>

Introduction à Spark

Chubby

The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, *Google Inc.*

Abstract

We describe our experiences with the Chubby lock service, which is intended to provide coarse-grained locking as well as reliable (though low-volume) storage for a loosely-coupled distributed system. Chubby provides an interface much like a distributed file system with advisory locks, but the design emphasis is on availability and reliability, as opposed to high performance. Many instances of the service have been used for over a year, with several of them each handling a few tens of thousands of clients concurrently. The paper describes the initial design and expected use, compares it with actual use, and explains how the design had to be modified to accommodate the differences.

1 Introduction

This paper describes a *lock service* called Chubby. It is intended for use within a loosely-coupled distributed sys-

example, the Google File System [7] uses a Chubby lock to appoint a GFS master server, and Bigtable [3] uses Chubby in several ways: to elect a master, to allow the master to discover the servers it controls, and to permit clients to find the master. In addition, both GFS and Bigtable use Chubby as a well-known and available location to store a small amount of meta-data; in effect they use Chubby as the root of their distributed data structures. Some services use locks to partition work (at a coarse grain) between several servers.

Before Chubby was deployed, most distributed systems at Google used *ad hoc* methods for primary election (when work could be duplicated without harm), or required operator intervention (when correctness was essential). In the former case, Chubby allowed a small saving in computing effort. In the latter case, it achieved a significant improvement in availability in systems that no longer required human intervention on failure.

Readers familiar with distributed computing will recognize the election of a primary among peers as an in-

<https://static.googleusercontent.com/media/research.google.com/fr//archive/chubby-osdi06.pdf>

Introduction à Spark

BigTable

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach

Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

1 Introduction

Over the last two and a half years we have designed,

achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Sec-

<https://static.googleusercontent.com/media/research.google.com/fr//archive/bigtable-osdi06.pdf>

Introduction à Spark

Hadoop

- Désormais nous avons à disposition les principes du modèle de programmation MapReduce.
- Pour passer à l'échelle les traitements sur de gros volumes de données. il faut pour cela qu'il soit associé à une infrastructure logicielle dédiée qui permette d'exécuter le schéma MapReduce de manière massivement distribuée sur un cluster de machines tout en prenant à sa charge les enjeux du calcul distribué :
 - ➡ l'optimisation des transferts disques et réseau en limitant les déplacements de données (data locality),
 - ➡ la scalabilité pour permettre d'adapter la puissance au besoin (scalability),
 - ➡ la tolérance aux pannes (embracing failure).

Introduction à Spark

Hadoop

- En 2002, Doug Cutting et Mike Cafarella, deux ingénieurs, décident de s'attaquer au passage à l'échelle de Lucene, le moteur de recherche open source. L'objectif était de le rendre capable d'indexer et de rechercher dans des collections de la taille du Web. C'est le projet Nutch.
- Pour cela, ils s'inspirent de deux articles de recherche publiés par les Google Labs que nous avons vus : GFS et MapReduce.
- L'architecture de Nutch, qui repose donc sur un système de fichiers distribué et sur MapReduce, est relativement générique et donnera lieu au projet Hadoop, initié en 2006.
- Il rejoint la fondation Apache en 2008.
- La version stable actuelle est la version 2.7.



Introduction à Spark

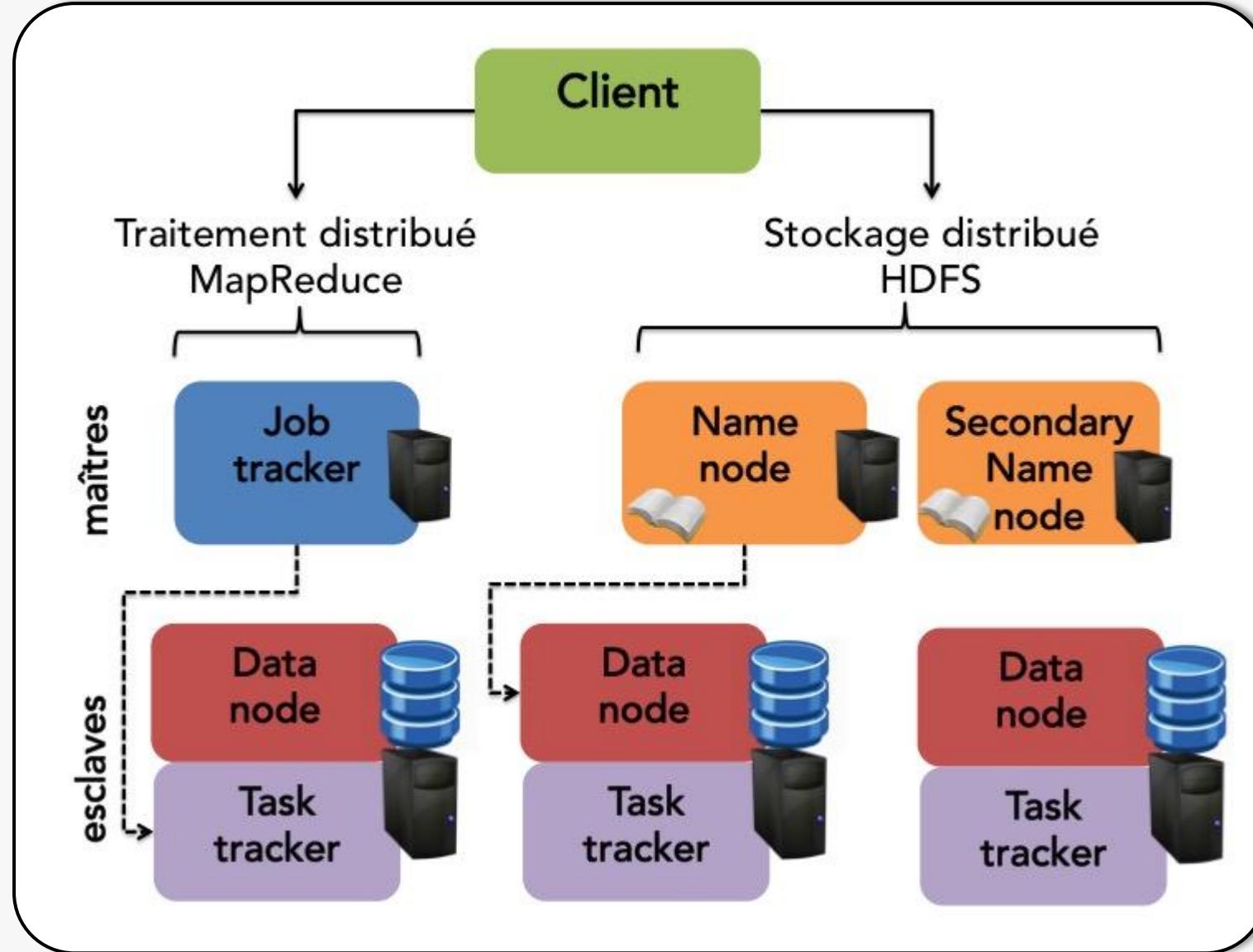
Hadoop

Le socle technique d'Hadoop est composé :

- De toute l'architecture support nécessaire pour l'orchestration de MapReduce, c'est-à-dire :
 - ➡ l'ordonnancement des traitements,
 - ➡ la localisation des fichiers,
 - ➡ la distribution de l'exécution.
- D'un système de fichiers **HDFS** qui est :
 - ➡ Distribué : les données sont réparties sur les machines du cluster.
 - ➡ Répliqué : en cas de panne, aucune donnée n'est perdue.
 - ➡ Optimisé pour la colocalisation des données et des traitements.

Introduction à Spark

Hadoop



Introduction à Spark

HDFS

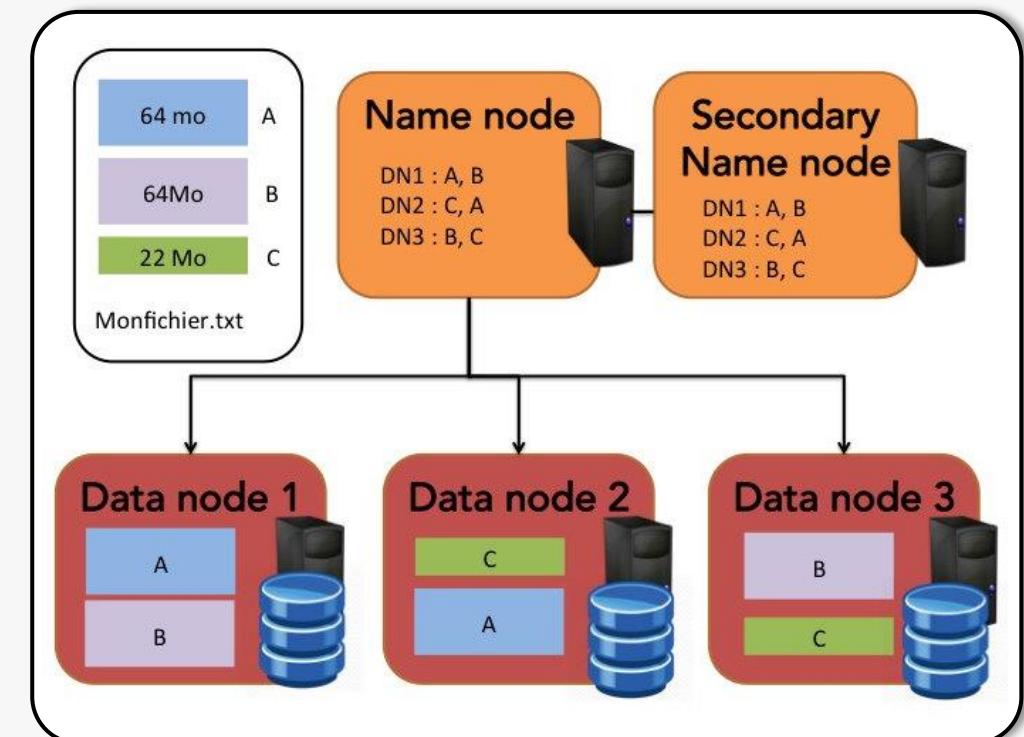
- HDFS (*Hadoop Distributed File System*) : système de fichiers distribué et couche native de stockage et d'accès à des données d'Hadoop.
- Il a été conçu pour stocker des fichiers de très grande taille et, comme son nom l'indique, dans un cadre distribué.
 - ➡ les fichiers sont physiquement **découpés en blocs d'octets de grande taille** (par défaut 64 Mo) pour optimiser les temps de transfert et d'accès ;
 - ➡ ces blocs sont ensuite **répartis** sur plusieurs machines, permettant ainsi de traiter un même fichier en parallèle. Cela permet aussi de ne pas être limité par la capacité de stockage d'une seule machine pour au contraire tirer parti de tout l'espace disponible du cluster de machines ;
 - ➡ pour garantir une tolérance aux pannes, les blocs de chaque fichier sont **repliqués**, de manière intelligente, sur plusieurs machines.

Introduction à Spark

HDFS

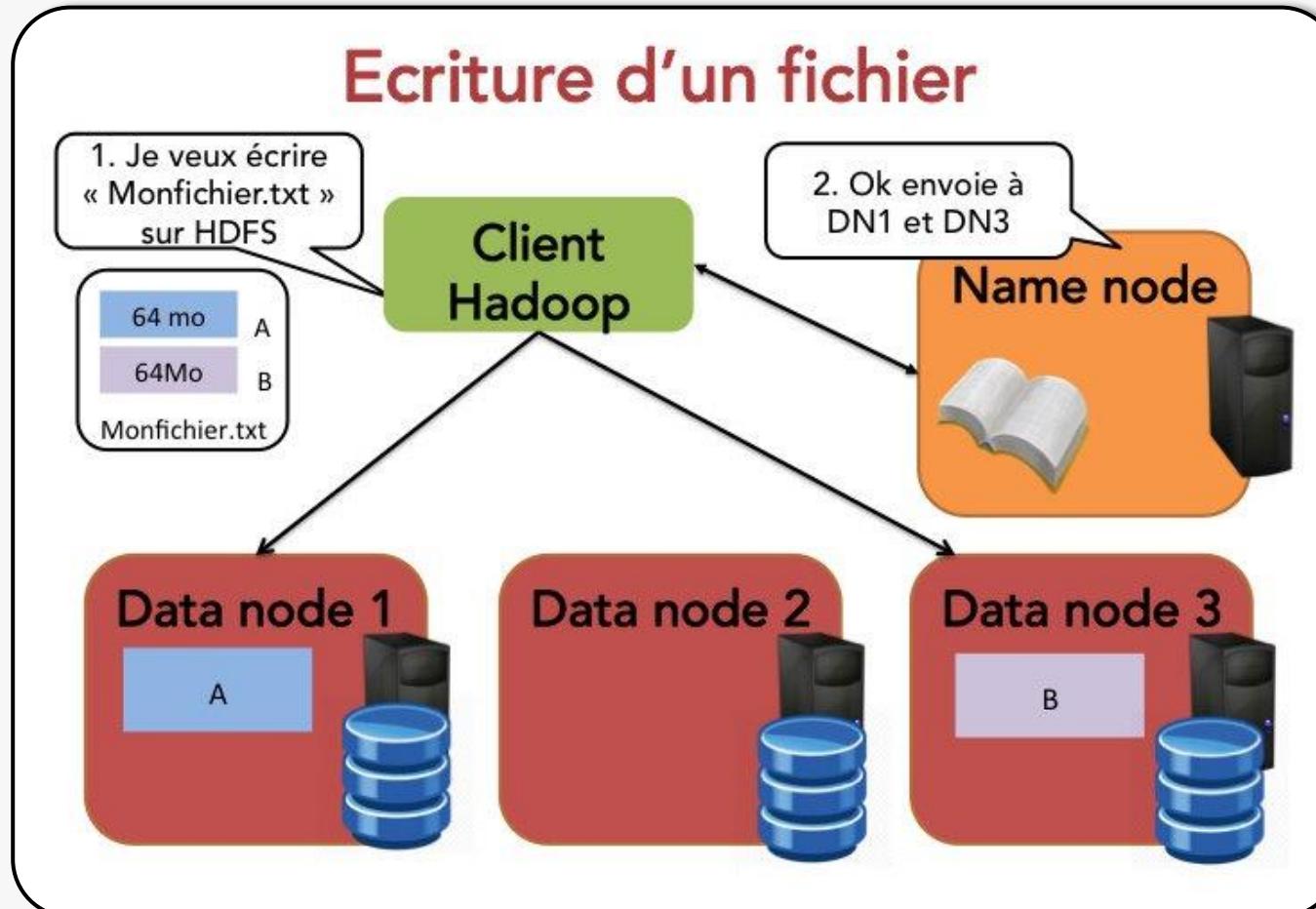
Dans Hadoop, l'architecture de stockage est une architecture maître-esclave.

- Le nœud maître appelé name node contient et stocke tous les noms et blocs des fichiers ainsi que leur localisation dans le cluster. On peut donc le voir comme un gros annuaire.
- Une autre machine, appelée secondary name node sert de name node de secours en cas de défaillance du nœud maître et il a donc pour rôle de faire des sauvegardes régulières de l'annuaire.
- Les autres nœuds, les esclaves, sont les nœuds de stockage en tant que tels. Ce sont les data nodes qui ont pour rôle la gestion des opérations de stockage locales (création, suppression et réPLICATION de blocs) sur instruction du name node.



Introduction à Spark

HDFS

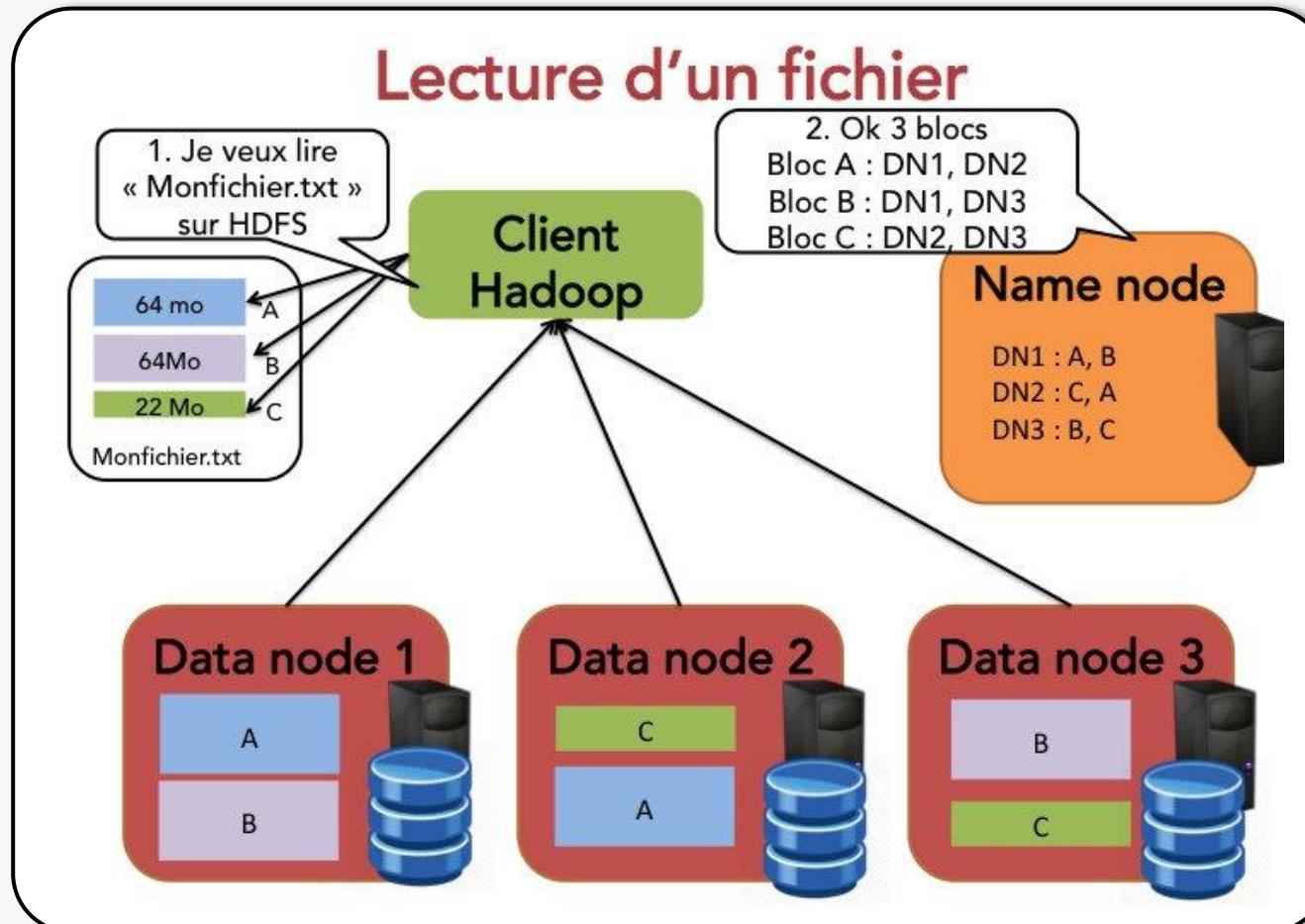


Si on souhaite écrire un fichier dans HDFS, on utilise un client Hadoop :

1. Le client indique au name node qu'il souhaite écrire un bloc.
2. Le name node indique le data node à contacter.
3. Le client envoie le bloc au data node.
4. Les data nodes répliquent les blocs entre eux.
5. Le cycle se répète pour le bloc suivant.

Introduction à Spark

HDFS



Si on souhaire lire un fichier dans HDFS :

1. Le client indique au name node qu'il souhaite lire un fichier.
2. Le name node indique sa taille ainsi que les différents data nodes contenant les blocs.
3. Le client récupère chacun des blocs sur l'un des data nodes.
4. Si le data node est indisponible, le client en contacte un autre.

Introduction à Spark

Hadoop MapReduce

- Comment ordonner les traitements ?
- Comment distribuer l'exécution sur les différents nœuds du cluster ?
- Comment connaître l'emplacement des fichiers à traiter ?

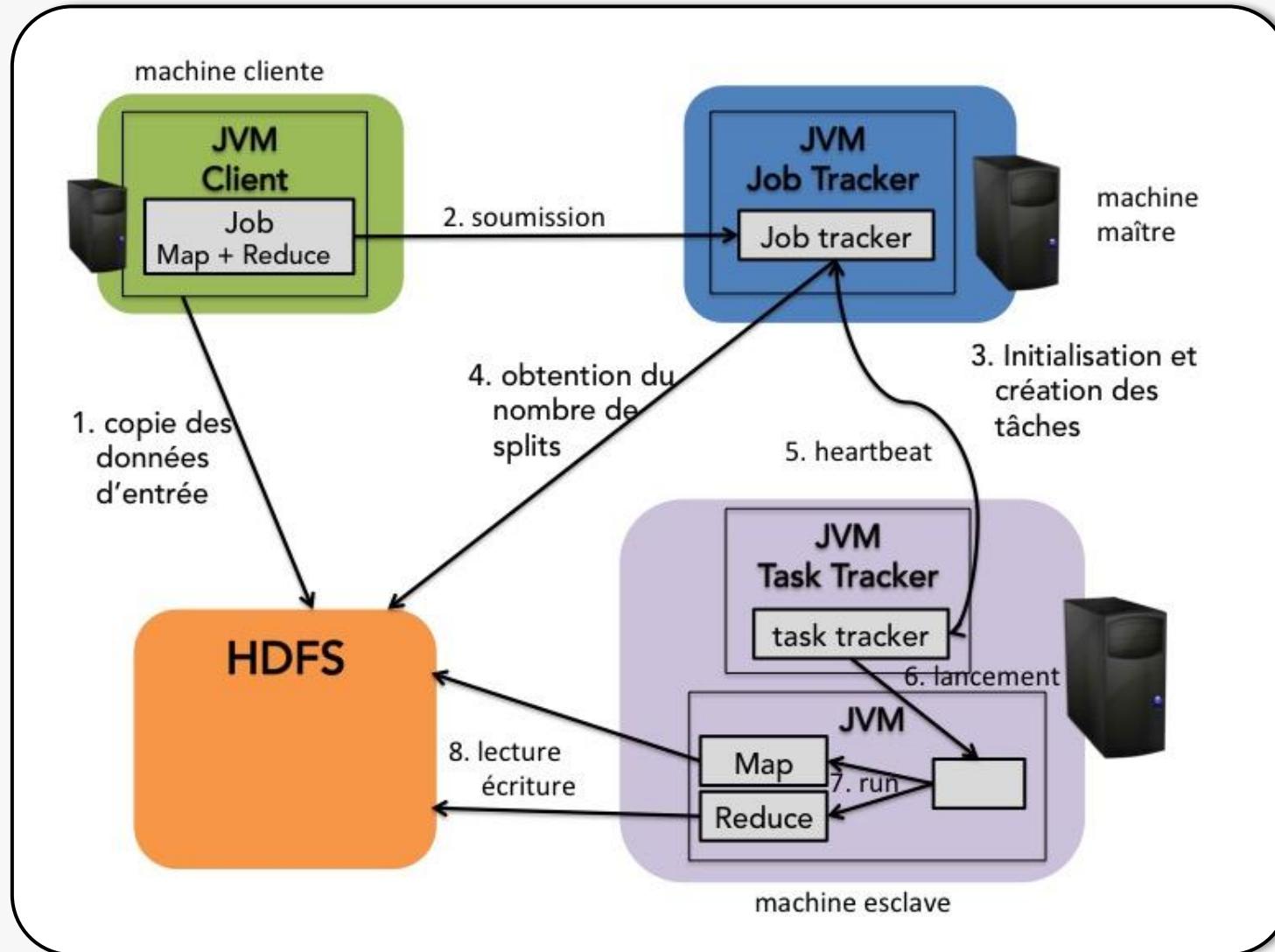
Hadoop peut s'occuper de tout cela, à nouveau avec une architecture de type **maître-esclave** :

Le **job tracker** est un processus maître qui va se charger de l'ordonnancement des traitements et de la gestion de l'ensemble des ressources du système. Il reçoit (du client) la ou les tâches MapReduce à exécuter (un .jar Java) ainsi que les données d'entrée et le répertoire où stocker les données de sorties. Il est pour cela en communication avec le **name node** d'HDFS. Le job tracker est en charge de planifier l'exécution des tâches et de les distribuer sur des **task trackers**. Comme il sait où sont situés les blocs de données, il peut optimiser la colocalisation traitements/données.

Un **task tracker** est une unité de calcul du cluster. Il assure, en lançant une nouvelle machine virtuelle java (JVM), l'exécution et le suivi des tâches MAP ou REDUCE s'exécutant sur son nœud et qu'il reçoit du **job tracker**. Il dispose d'un nombre limité de slots d'exécution et donc un nombre limité de tâches MAP, REDUCE ou SHUFFLE pouvant s'exécuter simultanément sur le nœud. Il est aussi en communication constante avec le **job tracker** pour l'informer de l'état d'avancement des tâches (*heartbeat call*). En cas de défaillance, le **job tracker**, informé ou non par le task tracker, doit pouvoir ordonner la réexécution de la tâche.

Introduction à Spark

Hadoop MapReduce



Introduction à Spark

Hadoop 2 et Yarn

Nous avons vu que l'algorithme MapReduce permet d'implémenter de nombreux types de traitement en vue de leur parallélisation.

→ Tous les problèmes rentrent-ils dans le moule MapReduce ?

Introduction à Spark

Hadoop 2 et Yarn

Nous avons vu que l'algorithme MapReduce permet d'implémenter de nombreux types de traitement en vue de leur parallélisation.

→ Tous les problèmes rentrent-ils dans le moule MapReduce ?

Non pas forcément et très souvent, si c'est le cas, cela demande beaucoup d'efforts de transformer un algorithme en MapReduce.

Introduction à Spark

Hadoop 2 et Yarn

Nous avons vu que l'algorithme MapReduce permet d'implémenter de nombreux types de traitement en vue de leur parallélisation.

→ Tous les problèmes rentrent-ils dans le moule MapReduce ?

Non pas forcément et très souvent, si c'est le cas, cela demande beaucoup d'efforts de transformer un algorithme en MapReduce.

Pour traiter des problèmes complexes, les deux étapes MAP et REDUCE ne suffisent pas, il est très souvent nécessaire d'enchaîner des séquences de MapReduce ce qui est très coûteux car cela nécessite de démarrer un job MapReduce à chaque fois.

Introduction à Spark

Hadoop 2 et Yarn

Nous avons vu que l'algorithme MapReduce permet d'implémenter de nombreux types de traitement en vue de leur parallélisation.

→ Tous les problèmes rentrent-ils dans le moule MapReduce ?

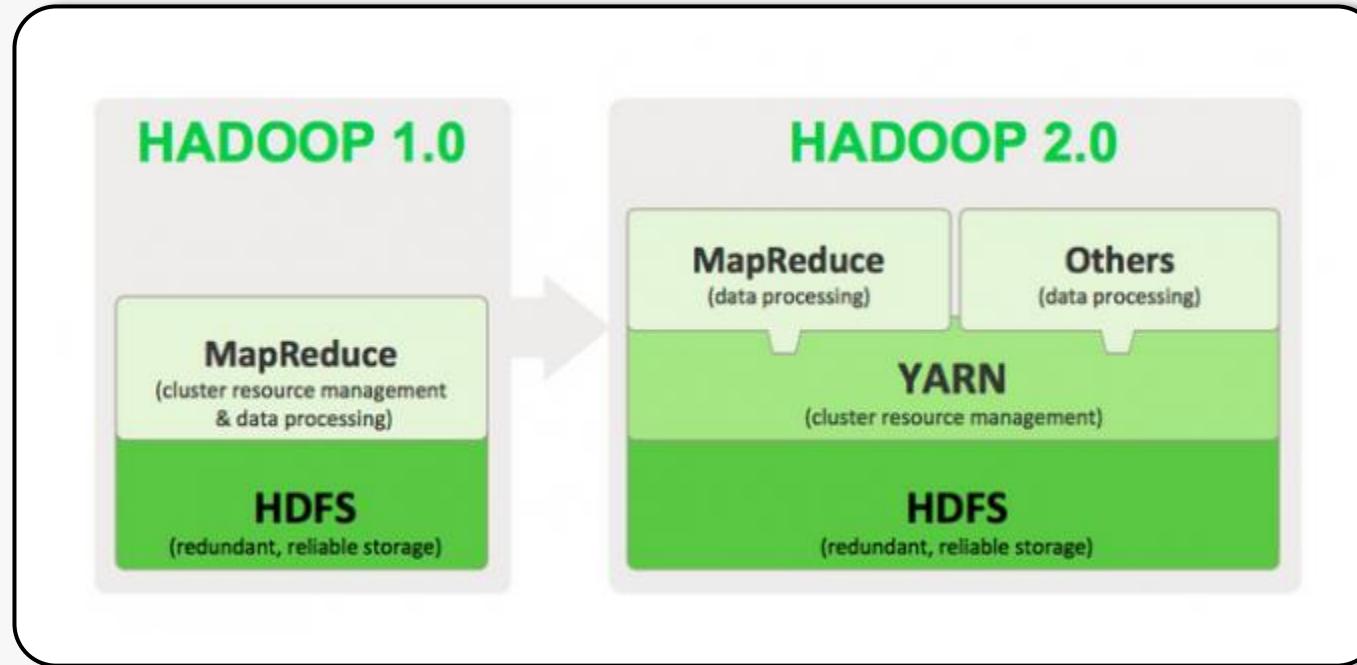
Non pas forcément et très souvent, si c'est le cas, cela demande beaucoup d'efforts de transformer un algorithme en MapReduce.

Pour traiter des problèmes complexes, les deux étapes MAP et REDUCE ne suffisent pas, il est très souvent nécessaire d'enchaîner des séquences de MapReduce ce qui est très coûteux car cela nécessite de démarrer un job MapReduce à chaque fois.

Le job tracker a une double responsabilité : gérer les ressources du cluster et ordonner les jobs. C'est un point critique en cas de défaillance.

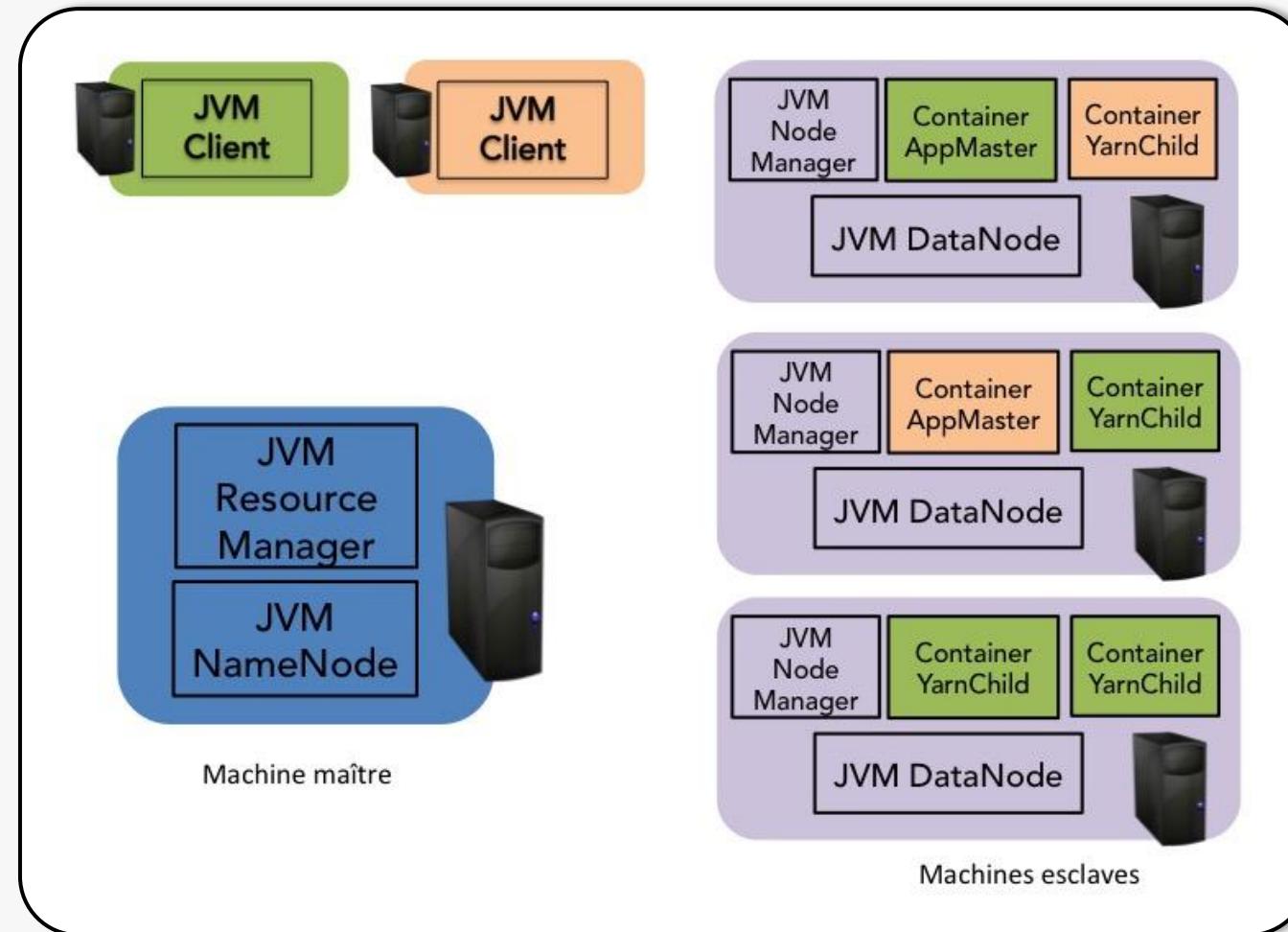
Introduction à Spark

Hadoop 2 et Yarn



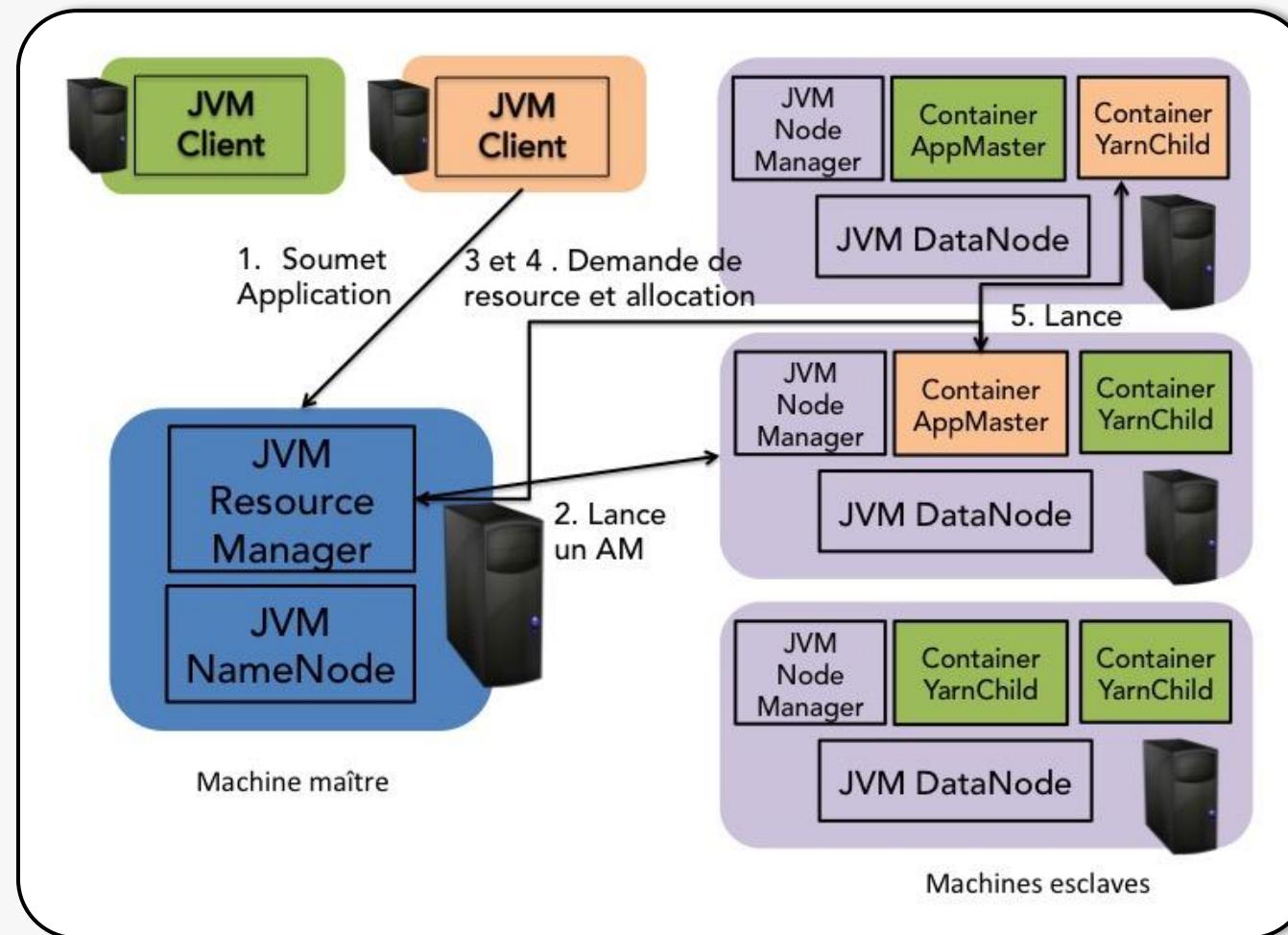
Introduction à Spark

Hadoop 2 et Yarn



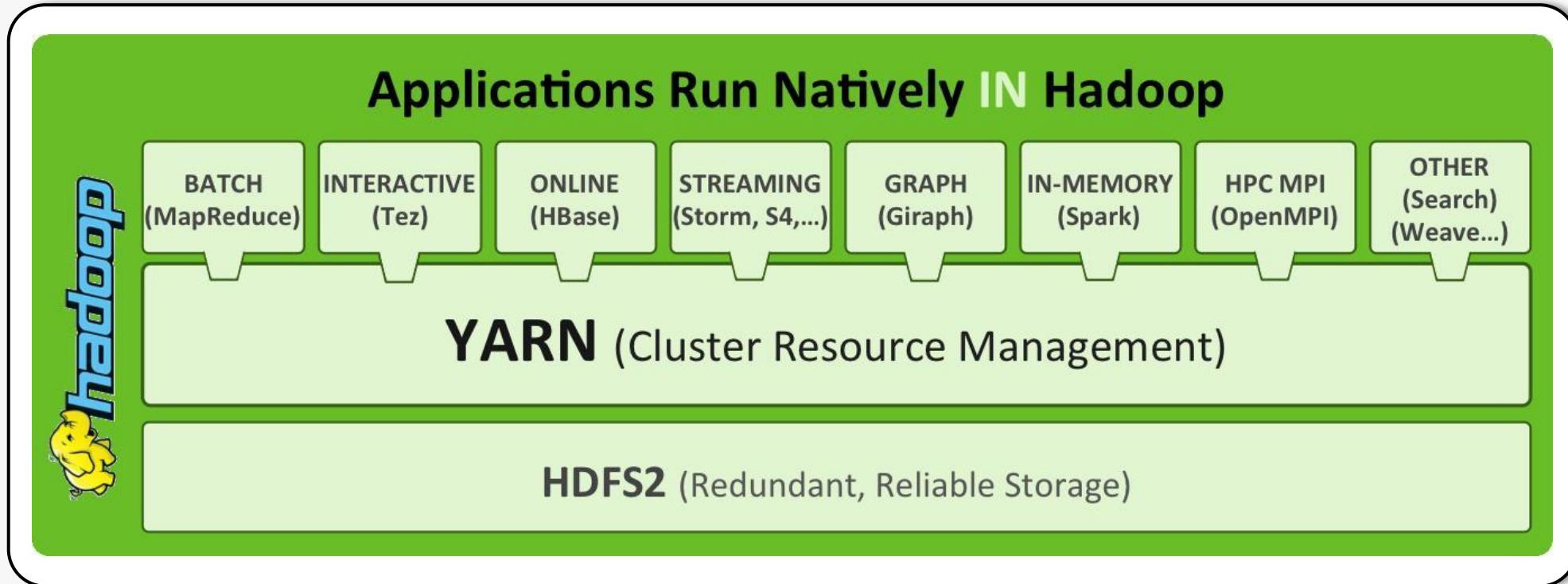
Introduction à Spark

Hadoop 2 et Yarn



Introduction à Spark

Hadoop 2 et Yarn



Introduction à Spark

Spark, le retour

Spark :

- Spark est un framework de traitement de données en mémoire conçu pour le traitement rapide et efficace de gros volumes de données.
- Contrairement à MapReduce, Spark conserve les données en mémoire chaque fois que cela est possible, ce qui réduit considérablement les temps d'accès aux données.
- Spark prend en charge un large éventail de charges de travail, y compris le traitement de flux de données, le traitement de graphiques, l'apprentissage automatique et le traitement interactif.
- Il fournit des APIs faciles à utiliser pour le développement d'applications de traitement de données en Python, Scala, Java et R.



Question 1

Volumes de données

La BNF (Bibliothèque Nationale de France) contient (à la louche) 30 millions de livres. En admettant que, en moyenne, chaque livre contienne 300 pages et que chaque page contienne 2000 caractères, combien de données non compressées cela représente-t-il ?

- Moins de 1 Go
- Entre 1 Go et 1 To
- Entre 1 To et 1 Po
- Entre 1 Po et 1 Eo

Question 2

Passage à l'échelle

Quand on ajoute des machines à un cluster pour augmenter sa capacité, il s'agit d'un passage à l'échelle :

- Horizontal
- Transversal
- Vertical

Question 3

Hadoop

Hadoop est :

- un framework théorique de calcul particulièrement adapté aux calculs distribués
- un outil qui permet l'implémentation de calculs distribués

Introduction à Spark

Définition des composants

Driver

Processus principal de l'application Spark. Il contient la fonction **main** du programme et est responsable de la création du contexte Spark, de la création des RDDs, de la planification des opérations, et de la coordination des différentes étapes de l'exécution.

Executor

Processus qui s'exécute sur les nœuds du cluster et est responsable de l'exécution réelle des tâches (tasks) de l'application Spark. Chaque application Spark dispose de ses propres executors qui sont lancés sur les nœuds du cluster par le driver.

Job

Unité logique de travail que le driver envoie aux executors pour être exécutée. Généralement créé lorsqu'une action est appelée sur un RDD, ce qui déclenche une séquence d'opérations (transformations) sur les données, qui sont ensuite regroupées en plusieurs stages et exécutées par les executors.

Stage

Unité d'opérations qui peut être exécutée en parallèle sur les données. Les stages sont divisés en fonction des opérations de transformation et des partitions des données. Un job peut contenir plusieurs stages, et chaque stage peut contenir plusieurs tasks qui sont exécutées en parallèle sur les Executors.

Task

La plus petite unité d'exécution dans Spark. Chaque task est responsable du traitement d'une partition de données spécifique. Les tasks sont exécutées sur les executors et effectuent les calculs nécessaires pour transformer ou traiter les données selon les opérations spécifiées par l'application Spark.

Introduction à Spark

Closure

L'une des difficultés principales de Spark est de comprendre la portée et le cycle de vie des variables et des méthodes lors de l'exécution du code à travers un cluster.

Les opérations sur les RDD qui modifient des variables en dehors de leur portée sont une source fréquente de confusion.

```
counter = 0
rdd = sc.parallelize(data)

def increment_counter(x):
    global counter
    counter += x

rdd.foreach(increment_counter)
print("Counter value: ", counter)
```

Introduction à Spark

Closure

- Le code précédent peut ne pas fonctionner comme prévu.
- Pour exécuter des jobs, Spark divise le traitement des opérations sur les RDD en tâches, chacune étant exécutée par un exécuteur. Avant l'exécution, Spark calcule la « closure » de la tâche.
- La closure comprend **les variables et les méthodes qui doivent être visibles pour l'exécuteur** afin qu'il puisse effectuer ses calculs sur le RDD (dans ce cas, `foreach()`). Cette closure est sérialisée et envoyée à chaque exécuteur.
- Les variables dans la fermeture envoyée à chaque exécuteur sont alors des copies et, par conséquent, lorsque le compteur est référencé dans la fonction `foreach()`, ce n'est plus le compteur sur le nœud du pilote. Il y a toujours un compteur dans la mémoire du nœud du pilote, mais celui-ci n'est plus visible pour les exécuteurs ! Les exécuteurs ne voient que la copie de la closure sérialisée.
 - ➡ Ainsi, la valeur finale du compteur sera toujours zéro puisque toutes les opérations sur le compteur référaient la valeur dans la fermeture sérialisée.

Introduction à Spark

Closure

- En mode local, dans certaines circonstances, la fonction `foreach()` s'exécutera en fait dans le même JVM que le pilote et fera référence au même compteur d'origine, et pourra même le mettre à jour.
- Pour garantir un comportement bien défini dans ces types de scénarios, il convient d'utiliser un accumulateur. Les accumulateurs dans Spark sont utilisés spécifiquement pour fournir un mécanisme de mise à jour sécurisée d'une variable lorsque l'exécution est répartie sur des nœuds de travail dans un cluster. On y reviendra plus tard.
- En général, les fermetures - des constructions comme des boucles ou des méthodes définies localement, ne doivent pas être utilisées pour muter un état global. Spark ne définit pas ou ne garantit pas le comportement des mutations sur des objets référencés en dehors des fermetures. Certain code qui le fait peut fonctionner en mode local, mais ce n'est que par accident et un tel code ne se comportera pas comme prévu en mode distribué. Utilisez un accumulateur si une agrégation globale est nécessaire.

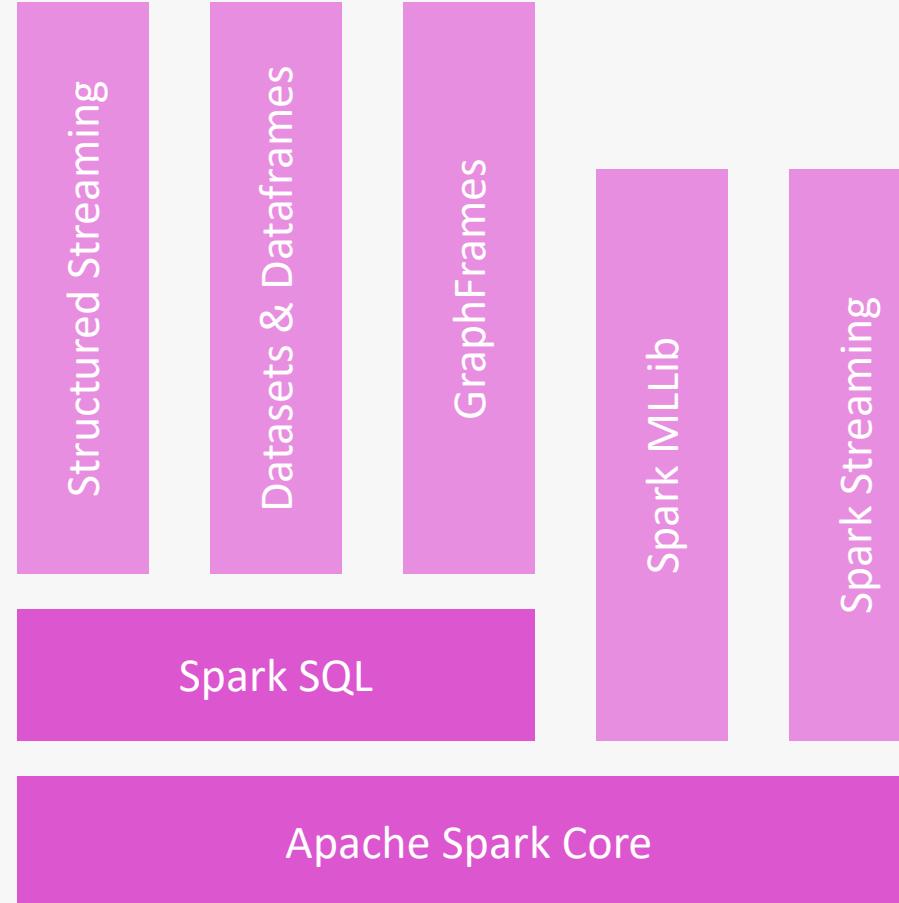
Introduction à Spark

Closure

- **Impression des éléments d'un RDD**
- Une autre pratique courante est d'essayer d'afficher les éléments d'un RDD en utilisant `rdd.foreach(println)` ou `rdd.map(println)`.
- Sur une seule machine, cela générera la sortie attendue et affichera tous les éléments du RDD.
- Cependant, en mode cluster, la sortie `stdout` appelée par les exécuteurs écrit maintenant dans `stdout` de l'exécuteur, et non dans celui du pilote, donc `stdout` du pilote ne les montrera pas !
- Pour afficher tous les éléments sur le pilote, on peut utiliser la méthode `collect()` pour d'abord amener le RDD au nœud du pilote comme suit : `rdd.collect().foreach(println)`. Cela peut amener le pilote à manquer de mémoire, cependant, car `collect()` récupère l'ensemble du RDD sur une seule machine ; si vous avez seulement besoin d'afficher quelques éléments du RDD, une approche plus sûre est d'utiliser `take()` : `rdd.take(100).foreach(println)`.

Introduction à Spark

Les principaux composants



Introduction à Spark

TP



**Installation de Spark sur Gitpod
Prise en main de Databricks
Premières manipulations de RDD**

Introduction à Spark

Installation sur Gitpod

Les énoncés sont disponibles sur Github à l'adresse suivante :

<https://github.com/BEEESPE/spark-prise-en-main>

Vous pouvez cloner ce repo et créer un workspace Gitpod à partir de celui-ci.

Introduction à Spark

Installation sur Gitpod

Étape 1 - Télécharger l'image de base

Si jamais Docker n'est pas installé (il l'est sur Gitpod), il faut commencer par installer Docker (<https://docs.docker.com/install/>).

Nous utiliserons Ubuntu comme environnement cible pour notre conteneur Docker. Télécharger l'image Ubuntu à partir de Docker Hub, avec la commande suivante :

```
docker pull ubuntu
```

Créer un conteneur à partir de cette image :

```
docker run -itd -p 8080:8080 --name spark --hostname spark ubuntu
```

Introduction à Spark

Installation sur Gitpod

Un nouveau conteneur intitulé spark a été lancé à partir de la machine ubuntu, en exposant sur le localhost son port 8080, pour pouvoir accéder à sa WebURL. Vérifier que le conteneur est bien démarré en utilisant :

```
docker ps
```

Lancer la commande suivante pour ouvrir le bash dans le conteneur :

```
docker exec -it spark bash
```

A noter que ces étapes sont faites une seule fois, à la première création du conteneur. Pour relancer un conteneur arrêté, utiliser la commande docker start

Introduction à Spark

Installation sur Gitpod

Étape 2 – Installer Java

Afin d'installer Java sur la machine, commencer par mettre à jour les packages systèmes de Ubuntu :

```
apt update  
apt -y upgrade
```

Installer la dernière version de Java :

```
apt install default-jdk
```

Vous pouvez vérifier la version installée avec :

```
java -version
```

Introduction à Spark

Installation sur Gitpod

Étape 3 – Installer Scala

```
apt install scala
```

Introduction à Spark

Installation sur Gitpod

Étape 4 – Télécharger Spark

Au jour de notre formation, la version la plus récente peut être installée à partir de ce lien :

```
apt install curl  
curl -O https://archive.apache.org/dist/spark/spark-3.5.1/spark-3.5.1-bin-hadoop3.tgz
```

Si vous vous reportez à ces slides dans le futur, vérifiez s'il n'existe pas de release plus récente.

Extraire ensuite l'archive tgz :

```
tar xvf spark-3.5.1-bin-hadoop3.tgz
```

Déplacer le dossier obtenu vers le répertoire /opt :

```
mv spark-2.4.5-bin-hadoop2.7 /opt/spark  
rm spark-2.4.5-bin-hadoop2.7.tgz
```

Introduction à Spark

Installation sur Gitpod

Étape 5 – Mettre en place l'environnement Spark

Nous devons à ce stade mettre en place certains paramètres d'environnement.

Installer vim :

```
apt install vim
```

Ouvrir le fichier de configuration bashrc :

```
vim ~/.bashrc
```

Ajouter les lignes suivantes à la fin du fichier (taper G pour aller à la fin du fichier, puis o pour insérer une nouvelle ligne et passer en mode édition) :

```
export SPARK_HOME=/opt/spark
```

```
export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin
```

Quitter l'éditeur en tapant :wq

Activer les changements réalisés :

```
source ~/.bashrc
```

Introduction à Spark

Installation sur Gitpod

Étape 6 – Démarrer un serveur master en standalone

Il est désormais possible de démarrer un serveur en standalone, en utilisant la commande suivante :

```
start-master.sh
```

Vous pourrez ensuite vérifier que votre serveur est bien démarré en tapant jps

Se rendre dans l'onglet PORTS et ouvrir le lien correspondant au port 8080. L'interface Web de Spark devrait s'afficher.

Introduction à Spark

Installation sur Gitpod

Étape 7 – Démarrer un processus worker

Pour lancer un processus Worker, utiliser la commande :

```
start-slave.sh spark://spark:7077
```

Un nouveau processus sera lancé et on pourra également le voir avec jps

Vous pouvez maintenant lancer le shell Spark pour executer des jobs Spark :

```
spark-shell
```

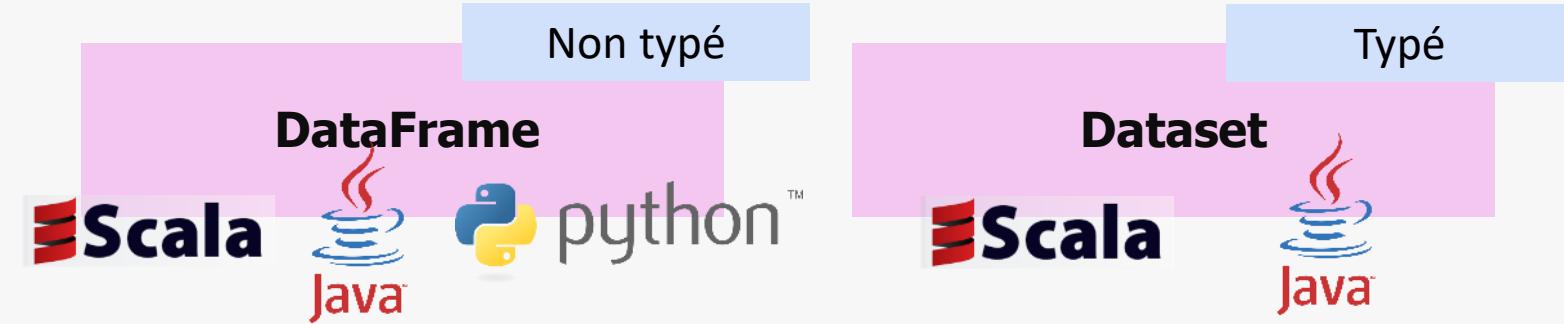
02

Fonctionnalités essentielles de Spark

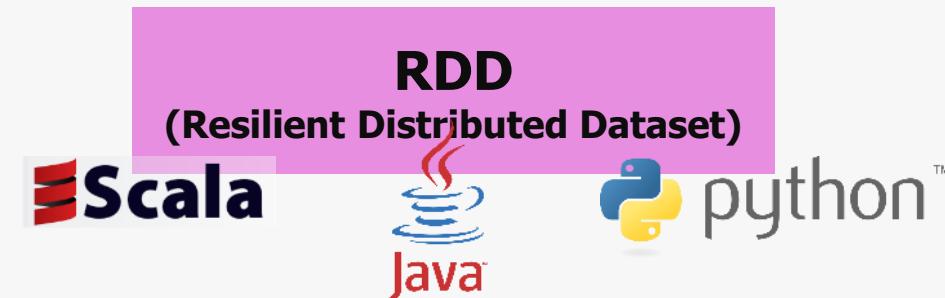
Fonctionnalités essentielles de Spark

Abstractions de Spark

Haut niveau



Bas niveau

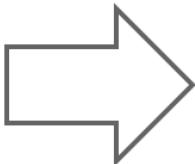


Fonctionnalités essentielles de Spark

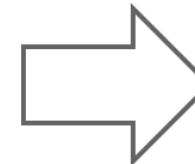
Abstractions de Spark

History of Spark APIs

RDD
(2011)



DataFrame
(2013)



DataSet
(2015)

Distribute collection
of JVM objects

Functional Operators (map,
filter, etc.)

Distribute collection
of Row objects

Expression-based operations
and UDFs

Internally rows, externally
JVM objects

Almost the “Best of both
worlds”: type safe + fast

Logical plans and optimizer

Fast/efficient internal
representations

But slower than DF
Not as good for interactive
analysis, especially Python

Fonctionnalités essentielles de Spark

RDDs

- RDD : Resilient Distributed Dataset.
Abstraction de base dans Spark. Abstraction de mémoire distribuée.
Collection immuable et distribuée d'éléments qui peuvent être traités en parallèle.
- Résilients car tolérants aux pannes : en cas de défaillance d'un nœud, les données peuvent être reconstruites à partir des opérations de transformation appliquées à partir de l'ensemble d'origine.
- Les RDD peuvent être créés à partir de fichiers stockés dans HDFS (Hadoop Distributed File System), de données en mémoire ou de tout autre stockage pris en charge par Spark.
- Les RDD peuvent être transformés à l'aide d'opérations telles que map, filter et reduce, et peuvent être actionnés pour déclencher des calculs.

Fonctionnalités essentielles de Spark

RDDs

Dans Spark, les RDD (Resilient Distributed Datasets) peuvent être de deux types principaux :

- génériques,
- key-value.

RDD générique Ensemble de données réparties sur plusieurs nœuds du cluster Spark.
Chaque élément dans un RDD générique est un enregistrement indépendant sans structure spécifique.
Les opérations sur les RDD génériques sont appliquées à chaque élément individuel du RDD.
Ces RDD sont utilisés pour des opérations simples où la structure des données n'est pas importante.

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
```

Fonctionnalités essentielles de Spark

RDDs

RDD key-value

Ensemble de paires clé-valeur réparties sur plusieurs nœuds du cluster Spark.
Chaque élément dans un RDD key-value est une paire (clé, valeur).
Les opérations sur les RDD key-value sont souvent effectuées en fonction des clés.
Ces RDD sont utilisés pour des opérations qui impliquent des agrégations par clé, des jointures et d'autres opérations de traitement par clé.

```
rdd = sc.parallelize([(1, 'a'), (2, 'b'), (3, 'c')])
```

Les RDD key-value offrent des transformations et des actions spécifiques adaptées à ce type de données, telles que `reduceByKey`, `groupByKey`, `sortByKey`, `join`, etc.



Les RDD génériques sont utilisés pour traiter des enregistrements indépendants, tandis que les RDD key-value sont utilisés pour le traitement par clé et les opérations qui impliquent des relations clé-valeur.

Fonctionnalités essentielles de Spark

Quand utiliser des RDD ?

- ➡ Vous avez besoin de fonctionnalités spécifiques uniquement disponibles dans les RDDs, distinctes des DataFrames et des Datasets.
 - ➡ Vous souhaitez maintenir un code préexistant écrit avec des RDDs.
 - ➡ Vous devez manipuler des variables partagées.
- Dans ces situations, l'utilisation des RDDs est justifiée.
- Cependant, pour la plupart des cas d'utilisation, il est recommandé d'opter pour des abstractions de plus haut niveau telles que les DataFrames ou les Datasets.
- Tout code Spark est finalement compilé en RDD, mais ces abstractions offrent une meilleure expressivité et des optimisations intégrées pour de nombreuses tâches courantes.

Fonctionnalités essentielles de Spark

Quelques caractéristiques supplémentaires des RDD

- **Liste de partitions**

Les RDD sont divisés en partitions, qui sont des morceaux de données stockées sur différents nœuds du cluster. Les partitions sont la base du parallélisme dans Spark, car les opérations peuvent être exécutées sur différentes partitions en même temps.

- **Fonction pour calculer chaque partition (en parallèle)**

Chaque partition d'un RDD est traitée de manière indépendante, ce qui signifie que la fonction de transformation ou d'action est appliquée à chaque partition en parallèle.

- **Liste de dépendances sur d'autres RDD**

Un RDD peut être dérivé d'un ou plusieurs autres RDDs. Cela permet de construire des pipelines de traitement de données complexes dans Spark.

- **En option, un Partitioner pour les RDD clé-valeur**

Pour les RDD clé-valeur, un Partitioner peut être spécifié pour contrôler comment les paires clé-valeur sont distribuées sur les partitions. Cela peut être utile pour optimiser les opérations de jointure et de regroupement.

Fonctionnalités essentielles de Spark

Tolérance aux pannes des RDD

Tolérance aux pannes des RDD = élément crucial pour garantir la fiabilité des traitements de données distribués.

- Lorsqu'une panne survient, les séquences de transformations peuvent être recalculées en cas de perte de données.
- Les RDD conservent les informations de lignée nécessaires pour reconstruire les partitions perdues, ce qui permet de ne rec算culer que les partitions affectées.
- Pour renforcer la résilience du système, il peut être parfois utile de pointer certains RDD vers un stockage stable, ce qui réduit le temps nécessaire pour reconstruire les partitions perdues, surtout pour les RDD impliqués dans des chaînes de transformations complexes. Cette fonctionnalité, unique aux RDD, est similaire à la mise en cache, mais elle stocke les données sur le disque plutôt que dans la mémoire.
- En utilisant `sc.setCheckpointDir()` pour définir un répertoire de vérification et `anRDD.checkpoint()` pour marquer un RDD pour la vérification, Spark est capable de gérer efficacement les pannes en réexécutant les étapes à partir du début avec une reconstitution judicieuse des données.

Fonctionnalités essentielles de Spark

Spark Context

- Spark Context : le point d'entrée pour créer des RDD.
- Plus précisément, le SparkContext représente la connexion à un cluster Spark et peut être utilisé pour créer des RDD, appliquer des transformations et des actions sur les données, configurer des paramètres d'application, et gérer les ressources du cluster.
- En Python, accès au contexte Spark via la variable sc (natif sur les clouders, convention dans les scripts que vous développez).

```
from pyspark import SparkContext

sc = SparkContext("local", "Exemple RDD")

data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)
pairs = rdd.filter(lambda x: x % 2 == 0)
resultat = pairs.collect()
print("Nombres pairs : ", resultat)
sc.stop()
```

Fonctionnalités essentielles de Spark

Transformations

- Les transformations sont des opérations sur les RDD qui créent un nouveau RDD à partir d'un RDD existant.
- Les transformations sont **paresseuses** (lazy), ce qui signifie qu'elles ne sont pas évaluées immédiatement. Au lieu de cela, elles construisent un **plan de calcul**.
- Exemples de transformations : **map**, **filter**, **flatMap**, **reduceByKey**, **join**, **sortBy...**
- Les transformations sont généralement utilisées pour transformer les données en appliquant des opérations de filtrage, de projection, d'agrégation...



Fonctionnalités essentielles de Spark

Transformations sur un RDD

Function name	Purpose	Example	Result when applied to {1, 2, 3, 3}
map()	Apply a function to each element in the RDD and return an RDD of the result.	rdd.map(lambda x: x + 1)	{2, 3, 4, 4}
flatMap()	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	rdd.flatMap(lambda x: range(x, 4))	{1, 2, 3, 2, 3, 3, 3}
filter()	Return an RDD consisting of elements that pass the condition passed to filter().	rdd.filter(lambda x: x != 1)	{2, 3, 3}
distinct()	Remove duplicates.	rdd.distinct()	{1, 2, 3}
sample(withReplacement, fraction, [seed])	Sample an RDD, with or without replacement.	rdd.sample(False, 0.5)	Non-deterministic

Adapté de : <https://kks32-courses.gitbook.io/data-analytics/spark/rdd/common-rdd-operations>

Fonctionnalités essentielles de Spark

Transformations sur deux RDD

Function name	Purpose	Example	Result when applied to {1, 2, 3} and {3, 4, 5}
union()	Produce an RDD containing elements from both RDDs.	rdd.union(other)	{1, 2, 3, 3, 4, 5}
intersection()	Produce an RDD containing only elements found in both RDDs.	rdd.intersection(other)	{3}
subtract()	Remove the contents of one RDD (e.g., remove training data).	rdd.subtract(other)	{1, 2}
cartesian()	Cartesian product with the other RDD.	rdd.cartesian(other)	{(1, 3), (1, 4), ... (3,5)}

Adapté de : <https://kks32-courses.gitbook.io/data-analytics/spark/rdd/common-rdd-operations>

Fonctionnalités essentielles de Spark

Actions

- Les actions sont des opérations sur les RDD qui déclenchent l'évaluation des transformations et déclenchent le traitement des données.
- Contrairement aux transformations, les actions déclenchent des calculs et des opérations réels sur les données.
- Exemples d'actions : **collect, take, reduce, count, saveAsTextFile...**
- Les actions sont généralement utilisées pour obtenir des résultats à partir des données, telles que la collecte des données sur le driver, le comptage des éléments, la sauvegarde des résultats dans un fichier...



Fonctionnalités essentielles de Spark

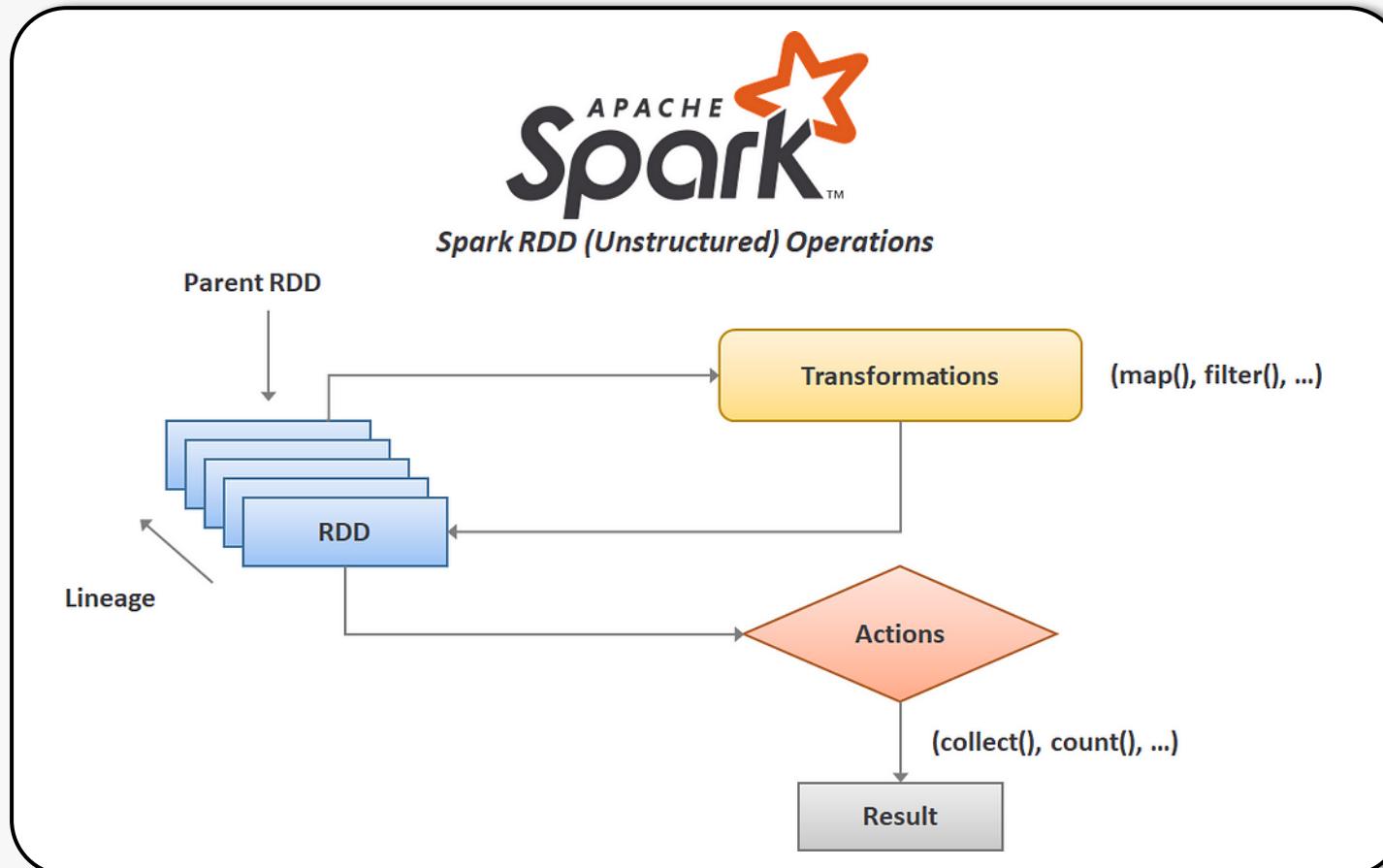
Actions sur un RDD

Function name	Purpose	Example	Result when applied to {1, 2, 3, 3}
collect()	Return all elements of RDD.	rdd.collect()	{1, 2, 3}
count()	Return the number of all elements of RDD.	rdd.count()	4
countByValue()	Number of times each element occurs in the RDD.	rdd.countByValue()	{(1, 1), (2, 1), (3, 2)}
take(num)	Return num elements from the RDD.	rdd.take(2)	{1, 2}
top(num)	Return the top num elements from the RDD.	rdd.top(2)	{3, 3}
takeSample(withReplacement, num, [seed])	Return num elements at random.	rdd.takeSample(False, 1)	Non-deterministic
reduce(func)	Combine the elements of the RDD together in parallel (e.g., sum).	rdd.reduce(add)	9
fold(zero)(func)	Same as reduce() but with the provided zero value.	rdd.fold(0)(add)	9
aggregate((0, 0), seqOp, combOp)	Similar to reduce() but used to return a different type.	rdd.aggregate((0, 0)) => (x._1 + y, x._2 + 1), (x, y) (9, 4) => (x._1 + y._1, x._2 + y._2))	
foreach(func)	Apply the provided function to each element of the RDD.	def f(x): print(x) rdd.foreach(f)	Nothing

Adapté de : <https://kks32-courses.gitbook.io/data-analytics/spark/rdd/common-rdd-operations>

Fonctionnalités essentielles de Spark

RDD, transformations et actions



Source : <https://medium.com/analytics-vidhya/spark-rdd-low-level-api-basics-using-pyspark-a9a322b58f6i>

Fonctionnalités essentielles de Spark

Mise en cache

- La mise en cache est le processus par lequel les RDD sont stockés **en mémoire** pour une réutilisation ultérieure.
- Lorsqu'un RDD est mis en cache, les données sont stockées dans la mémoire RAM des nœuds du cluster Spark. Cela permet d'accélérer les calculs en évitant de re-calculer les RDD intermédiaires à chaque action.
- Pour cela, on utilise la méthode **cache()**.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Exemple de mise en cache").getOrCreate()
df = spark.read.csv("chemin/vers/fichier.csv", header=True)
df.cache()
df.show()
df.groupBy("colonne").count().show()
spark.stop()
```



Fonctionnalités essentielles de Spark

Persistance

- La persistance est similaire à la mise en cache, mais elle offre plus de flexibilité en permettant de **spécifier le niveau de stockage des données**.
- Spark offre différents niveaux de stockage pour la persistance des RDD, tels que la mémoire, le disque ou une combinaison des deux.
- Pour cela, on utilise la méthode **persist()**.

```
from pyspark import SparkContext
sc = SparkContext("local", "Exemple de persistance")
lines = sc.textFile("chemin/vers/fichier.txt")
words = lines.flatMap(lambda line: line.split(" "))
persisted_words = words.persist()
word_counts = persisted_words.map(
    lambda word: (word, 1))
.persist()
word_counts.reduceByKey(
    lambda x, y: x + y)
word_counts.collect()
sc.stop()
```

Fonctionnalités essentielles de Spark

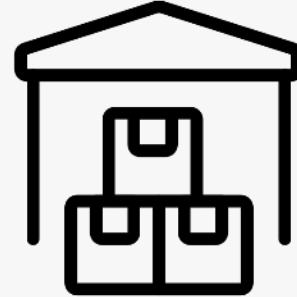
Niveaux de stockage

Storage Level	Description
MEMORY_ONLY	It stores the RDD as deserialized Java objects in the JVM. This is the default level. If the RDD doesn't fit in memory, some partitions will not be cached and recomputed each time they're needed.
MEMORY_AND_DISK	It stores the RDD as deserialized Java objects in the JVM. If the RDD doesn't fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	It stores RDD as serialized Java objects (i.e. one-byte array per partition). This is generally more space-efficient than deserialized objects.
MEMORY_AND_DISK_SER (Java and Scala)	It is similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them.
DISK_ONLY	It stores the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	It is the same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	It is similar to MEMORY_ONLY_SER, but store the data in off-heap memory. The off-heap memory must be enabled.

Source : <https://www.javatpoint.com/apache-spark-rdd-persistence>

Fonctionnalités essentielles de Spark

Niveaux de stockage



```
from pyspark import SparkContext
from pyspark import StorageLevel
sc = SparkContext("local", "Exemple de niveaux de stockage")
lines = sc.textFile("chemin/vers/fichier.txt")
lines.persist(StorageLevel.MEMORY_ONLY)
lines.count()
sc.stop()
```

Fonctionnalités essentielles de Spark

Caractéristiques impactant la pertinence de l'utilisation de cache()/persist()

- ➡ Fréquence d'utilisation de l'abstraction
- ➡ Coût de la transformation sur l'abstraction
- ➡ Volume de l'abstraction



Fonctionnalités essentielles de Spark

Les partitions : unités de base pour la répartition des données et le parallélisme

Définition

Partie de l'ensemble de données, répartie entre les différents nœuds d'un cluster Spark. Chaque partition contient un sous-ensemble des données et peut être traitée indépendamment par les nœuds du cluster.

Taille

Elle peut varier en fonction de plusieurs facteurs, tels que la taille totale des données, le nombre de nœuds dans le cluster, la configuration de Spark... Il est important de choisir une taille de partition appropriée pour optimiser les performances du traitement.

Parallélisme

Le nombre de partitions dans un RDD ou un DataFrame détermine le niveau de parallélisme dans le traitement. Plus il y a de partitions, plus le traitement peut être parallélisé, ce qui peut améliorer les performances globales.

Création

Lors de la lecture des données à partir de sources externes (fichiers CSV, bases de données...). Spark essaie généralement de répartir les données de manière équilibrée entre les partitions.

Répartition

Il est parfois nécessaire de répartir explicitement les données sur un certain nombre de partitions pour optimiser le traitement parallèle. Cela peut être fait à l'aide de transformations telles que `repartition()` ou `coalesce()` dans Spark.

Performance

Une gestion efficace des partitions peut grandement influencer les performances d'une application Spark. Une répartition équilibrée des données entre les partitions et une taille de partition appropriée peuvent aider à maximiser l'utilisation des ressources du cluster.

Fonctionnalités essentielles de Spark

Les partitions : unités de base pour la répartition des données et le parallélisme

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Configuration partitions").getOrCreate()

spark.conf.set("spark.sql.shuffle.partitions", "4")
donnees = spark.read.csv("donnees.csv")
donnees.printSchema()
spark.stop()
```

Fonctionnalités essentielles de Spark

Les partitions : unités de base pour la répartition des données et le parallélisme

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Exemple repartition coalesce").getOrCreate()
data = [("John", 25), ("Anna", 30), ("Robert", 35), ("Julia", 40)]
df = spark.createDataFrame(data, ["Name", "Age"])
print("Nombre de partitions actuelles :", df.rdd.getNumPartitions())
df_repartition = df.repartition(2)
print("Nombre de partitions après repartition() :", df_repartition.rdd.getNumPartitions())
df_coalesce = df.coalesce(1)
print("Nombre de partitions après coalesce() :", df_coalesce.rdd.getNumPartitions())
spark.stop()
```

Fonctionnalités essentielles de Spark

Partitionnement par plage ou par hachage

Partitionnement par hachage (Hash Partitioning)

Les enregistrements sont répartis dans les partitions en fonction de la valeur de hachage d'une clé.

Chaque enregistrement se voit attribuer une clé de hachage en fonction de la clé spécifiée dans la fonction de partitionnement. Ensuite, les enregistrements avec la même clé de hachage sont regroupés dans la même partition.

Efficace pour **répartir uniformément les données**, ce qui peut aider à réduire les goulets d'étranglement lors du traitement parallèle.

Exemple d'utilisation : Lorsqu'on a des clés de données aléatoires ou qu'on souhaite répartir uniformément les données entre les partitions.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Hash Partitioning Example").getOrCreate()
data = [("John", 25), ("Bob", 30), ("Alice", 35), ("Mary", 40), ("David", 45)]
df = spark.createDataFrame(data, ["Name", "Age"])
hashed_df = df.repartitionByHashPartitioning("Name", 3)
print("Number of partitions:", hashed_df.rdd.getNumPartitions())
spark.stop()
```

Fonctionnalités essentielles de Spark

Partitionnement par plage ou par hachage

Partitionnement par plage (Range Partitioning)

Les enregistrements sont répartis dans les partitions en fonction des plages de valeurs d'une clé donnée.

Les données sont triées en fonction de la clé de partitionnement spécifiée. Ensuite, ces données triées sont réparties dans les partitions en fonction des plages de valeurs spécifiées.

Utile lorsque vous avez des données **triées** et que vous souhaitez **maintenir cet ordre** dans les partitions résultantes.

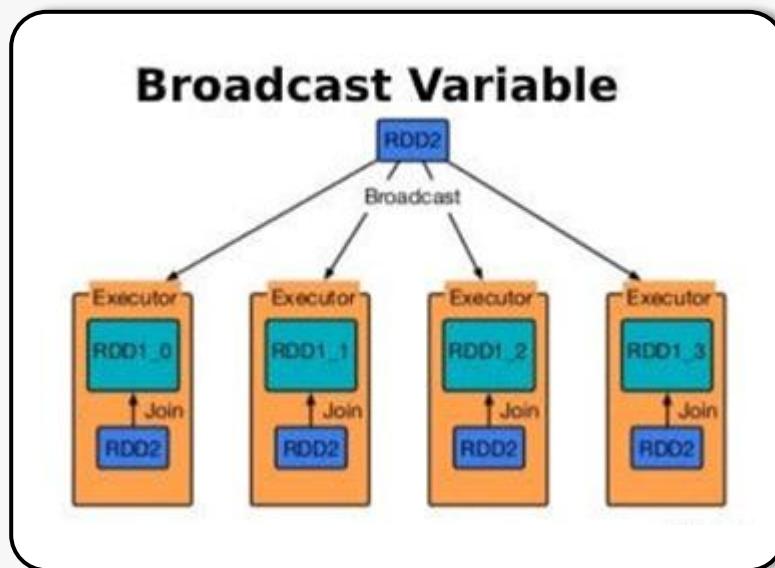
Exemple d'utilisation : Lorsqu'on a des données temporelles ou des données qui sont déjà triées selon une clé donnée.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Range Partitioning Example").getOrCreate()
data = [("John", 25), ("Bob", 30), ("Alice", 35), ("Mary", 40), ("David", 45)]
df = spark.createDataFrame(data, ["Name", "Age"])
ranged_df = df.repartitionByRangePartitioning("Age", [0, 30, 40, 50])
print("Number of partitions:", ranged_df.rdd.getNumPartitions())
spark.stop()
```

Fonctionnalités essentielles de Spark

Variables partagées : variables diffusées et accumulateurs

- Les variables partagées (shared variables) sont des fonctionnalités de Spark qui permettent de distribuer efficacement les données aux tâches exécutées sur différents nœuds du cluster.
- Accumulateur** (accumulator) : Pour agréger des données provenant de toutes les tâches dans un résultat partagé.
- Variable diffusée** (broadcast variable) : Pour diffuser une abstraction à tous les nœuds de travail et l'utiliser sans avoir à la renvoyer à travers le cluster.



Accumulators										
Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 3	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 4	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
										counter: 17

Fonctionnalités essentielles de Spark

Variables diffusées

- Permet au programmeur de garder une variable en lecture seule en cache sur chaque machine plutôt que d'en envoyer une copie avec les tâches.
- La variable n'est pas envoyée aux nœuds plus d'une fois.
- Spark utilise un algorithme de diffusion pour réduire les coûts de communication.

```
broadcast_var = sc.broadcast([1, 2, 3, 4, 5])
```

Fonctionnalités essentielles de Spark

Accumulateurs

- C'est une variable partagée aux workers et qui autorise l'écriture.
-
-

A votre avis, quelles propriétés doivent être vérifiées par l'opération qu'on cherche à appliquer à notre accumulateur ?

Fonctionnalités essentielles de Spark

Accumulateurs

- C'est une variable partagée aux workers et qui autorise l'écriture.
- Ces variables ne peuvent être ajoutées que par une opération **associative** et **commutative**.
- Les mises à jour de l'accumulateur sont effectuées uniquement par les ???.

Compléter les ???

Fonctionnalités essentielles de Spark

Accumulateurs

- C'est une variable partagée aux workers et qui autorise l'écriture.
- Ces variables ne peuvent être ajoutées que par une opération **associative** et **commutative**.
- Les mises à jour de l'accumulateur sont effectuées uniquement par les **actions**.

```
accum = sc.accumulator(0)
rdd.foreach(lambda x: accum.add(x))
```

Fonctionnalités essentielles de Spark

Note sur l'utilisation des variables partagées dans les versions récentes de Spark

NEW

À partir de Spark 2.2, Spark peut optimiser l'utilisation des variables diffusées de manière transparente dans de nombreux cas. Même si dans certains scénarios spécifiques ou pour un contrôle plus fin, il est toujours possible d'avoir besoin d'utiliser `sc.broadcast()` pour diffuser des variables, c'est beaucoup moins fréquent qu'avant !

Fonctionnalités essentielles de Spark

Soumettre des jobs Python à Spark

→ spark-submit

```
spark-submit --master <master-url> --deploy-mode <deploy-mode> --executor-memory <memory> your_script.py
```

→ PySpark Shell

Lancer le shell interactif PySpark pour interagir avec Spark en utilisant Python.
Cela est utile pour l'exploration de données et le prototypage rapide.

```
pyspark
```

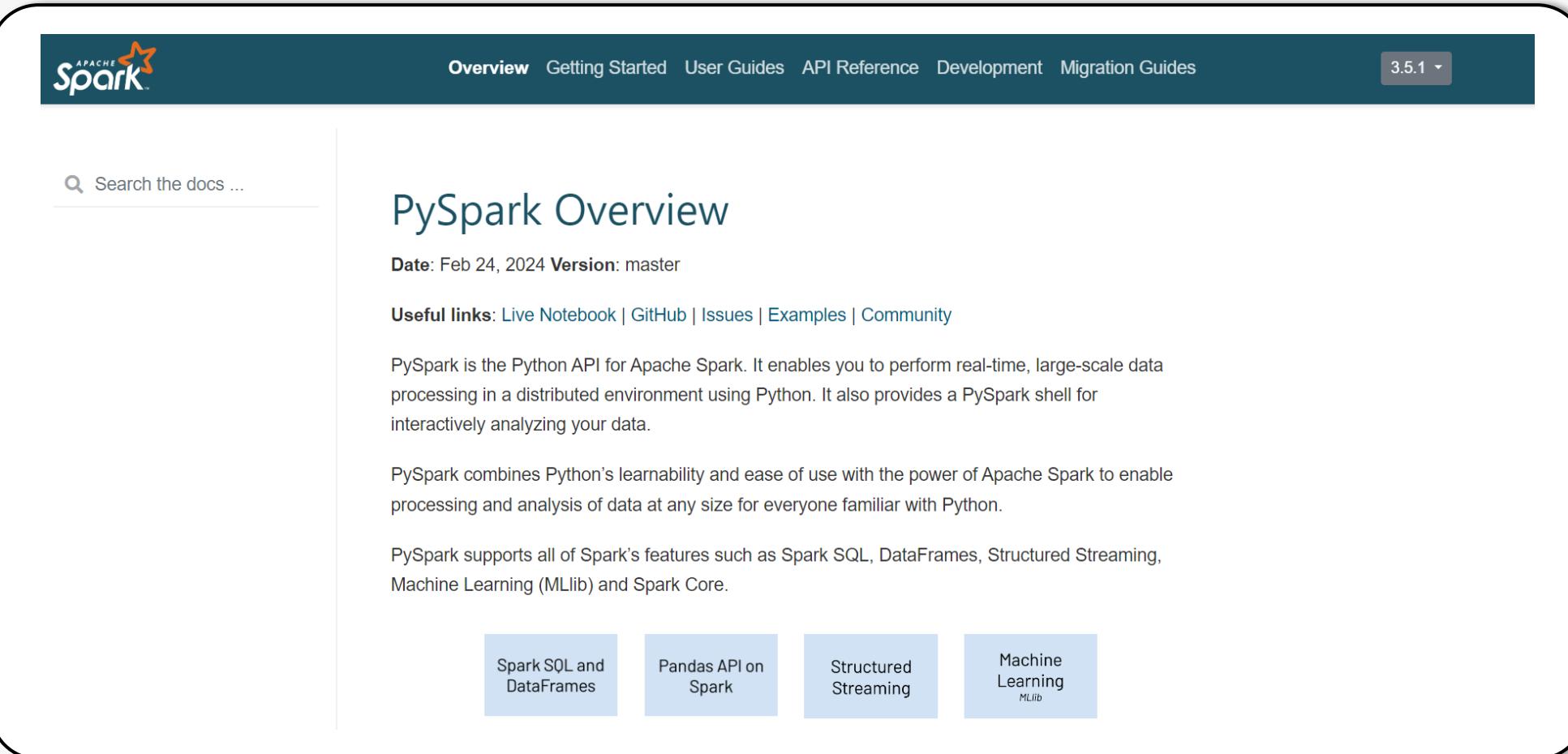
→ Notebooks Jupyter

Installer le package pyspark et configurer un kernel Jupyter pour PySpark.

```
from pyspark.sql import SparkSession  
spark = SparkSession.builder.appName("Nom de votre choix").getOrCreate()  
# Votre code Spark à suivre
```

Fonctionnalités essentielles de Spark

Pyspark



The screenshot shows the Apache Spark documentation page for PySpark. The header includes the Apache Spark logo, navigation links for Overview, Getting Started, User Guides, API Reference, Development, and Migration Guides, and a version dropdown set to 3.5.1. A search bar is present on the left. The main content features a large heading "PySpark Overview", a timestamp of "Date: Feb 24, 2024 Version: master", and useful links to Live Notebook, GitHub, Issues, Examples, and Community. It describes PySpark as the Python API for Apache Spark, enabling real-time, large-scale data processing in a distributed environment using Python. It also provides a PySpark shell for interactively analyzing your data. The text states that PySpark combines Python's learnability and ease of use with the power of Apache Spark to enable processing and analysis of data at any size for everyone familiar with Python. It supports all of Spark's features such as Spark SQL, DataFrames, Structured Streaming, Machine Learning (MLlib) and Spark Core. Below this, four blue boxes list "Spark SQL and DataFrames", "Pandas API on Spark", "Structured Streaming", and "Machine Learning MLlib".

Apache Spark

Overview Getting Started User Guides API Reference Development Migration Guides 3.5.1 ▾

Search the docs ...

PySpark Overview

Date: Feb 24, 2024 Version: master

Useful links: [Live Notebook](#) | [GitHub](#) | [Issues](#) | [Examples](#) | [Community](#)

PySpark is the Python API for Apache Spark. It enables you to perform real-time, large-scale data processing in a distributed environment using Python. It also provides a PySpark shell for interactively analyzing your data.

PySpark combines Python's learnability and ease of use with the power of Apache Spark to enable processing and analysis of data at any size for everyone familiar with Python.

PySpark supports all of Spark's features such as Spark SQL, DataFrames, Structured Streaming, Machine Learning (MLlib) and Spark Core.

Spark SQL and DataFrames Pandas API on Spark Structured Streaming Machine Learning MLlib

<https://spark.apache.org/docs/latest/api/python/index.html>

Fonctionnalités essentielles de Spark

TP



**Manipulation de RDD
Transformations et actions**

Question 4

Abstractions de Spark

Quelle abstraction de Spark est utilisée pour représenter un ensemble de données distribué, immuable et partitionné ?

- RDD
- DataFrame
- Dataset
- SparkContext

Question 5

Librairie Python pour Spark

Quel module Python est généralement utilisé pour interagir avec Spark ?

- PySpark
- SparkPy
- SparkTools
- PySparkle

Question 6

spark-submit

Quelle option est utilisée pour spécifier l'URL du maître dans la commande spark-submit ?

- cluster-url
- master-url
- url
- master

03

Spark SQL

Spark SQL

Spark SQL



- Spark SQL : module Spark dédié au traitement de données structurées.
- Contrairement à l'API de base Spark RDD, les interfaces fournies par Spark SQL fournissent à Spark plus d'informations sur la structure à la fois des données et des calculs effectués.
- En interne, Spark SQL utilise ces informations supplémentaires pour effectuer des optimisations supplémentaires.
- Il existe plusieurs façons d'interagir avec Spark SQL (SQL, API Dataset...), mais, lors du calcul d'un résultat, le même moteur d'exécution est utilisé, indépendamment de l'API/langage utilisé pour exprimer le calcul.

Spark SQL

Requêtes SQL

- Spark SQL peut exécuter des requêtes SQL.
- Spark SQL peut également être utilisé pour lire des données à partir d'une installation Hive existante.
- Lorsque on exécute du SQL à partir d'un autre langage de programmation, les résultats sont renvoyés sous forme de Dataset/DataFrame.
- Il est possible d'interagir avec l'interface SQL en utilisant la ligne de commande ou via JDBC/ODBC.



Spark SQL

Datasets et DataFrames

Dataset

Collection de données distribuée.

Nouvelle interface ajoutée dans Spark 1.6 qui combine les avantages des RDD (typage fort, capacité à utiliser de puissantes fonctions lambda) avec les avantages du moteur d'exécution optimisé de Spark SQL.

Peut être construit à partir d'objets JVM puis manipulé à l'aide de transformations fonctionnelles (map, flatMap, filter, etc.).

L'API Dataset est disponible en Scala et en Java. Python ne prend pas en charge l'API Dataset (mais en raison de la nature dynamique de Python, de nombreux avantages de l'API Dataset sont déjà disponibles - par exemple, accéder naturellement au champ d'une ligne par son nom row.columnName). Idem pour R.

Spark SQL

Datasets et DataFrames

DataFrame

Dataset organisé en colonnes nommées.

Il est conceptuellement équivalent à une table dans une base de données relationnelle ou à un data frame en R/Python, mais avec des optimisations plus riches sous le capot.

Peut être construit à partir de diverses sources telles que des fichiers de données structurées, des tables dans Hive, des bases de données externes ou des RDD existants.

L'API DataFrame est disponible en Scala, en Java, en Python et en R. (En Scala et en Java, un DataFrame est représenté par un Dataset de Rows. Dans l'API Scala, DataFrame est simplement un alias de type de Dataset[Row]. Alors qu'en Java API, les utilisateurs doivent utiliser Dataset<Row> pour représenter un DataFrame.)

Etant donné que cette formation est orientée vers une utilisation de Spark avec Python, nous nous focaliserons sur les DataFrames.

Spark SQL

Fonctions PySpark classiques pour la préparation des données (1/2)

Catégorie	Fonction/méthode	Description
Chargement	<code>spark.read.csv()</code> <code>spark.read.format()</code> <code>spark.read.jdbc()</code> <code>spark.readStream()</code>	Charge les données à partir d'un fichier CSV. Charge les données à partir de différents formats de fichiers (Parquet, JSON...). Charge les données à partir d'une base de données JDBC. Charge les données en streaming à partir de différentes sources.
Exploration	<code>DataFrame.printSchema()</code> <code>DataFrame.show()</code> <code>DataFrame.describe()</code>	Affiche le schéma des données. Affiche les premières lignes du DataFrame. Affiche des statistiques descriptives pour les colonnes numériques.
Nettoyage	<code>DataFrame.dropna()</code> <code>DataFrame.fillna()</code> <code>DataFrame.dropDuplicates()</code>	Supprime les lignes contenant des valeurs manquantes. Remplace les valeurs manquantes par une valeur spécifiée. Supprime les lignes en double.
Transformation	<code>DataFrame.select()</code> <code>DataFrame.filter()</code> <code>DataFrame.groupBy()</code> <code>DataFrame.join()</code> <code>DataFrame.withColumn()</code>	Sélectionne un sous-ensemble de colonnes du DataFrame. Filtre les lignes du DataFrame en fonction d'une condition. Regroupe les données en fonction des valeurs d'une ou plusieurs colonne(s). Effectue une jointure entre deux DataFrames. Ajoute une nouvelle colonne à partir d'une transformation sur les valeurs existantes d'une colonne (similaire à <code>.map()</code> sur les RDD).

Spark SQL

Fonctions PySpark classiques pour la préparation des données (2/2)

Catégorie	Fonction/méthode	Description
Feature engineering	Ajout de colonnes avec des expressions SQL ou des fonctions UDF Utilisation de fonctions intégrées comme concat(), substring(), split()...	
Partitionnement	DataFrame.repartition() DataFrame.coalesce()	Réorganise les partitions du DataFrame. Réduit le nombre de partitions du DataFrame.
Enregistrement	DataFrame.write.format() DataFrame.write.saveAsTable() DataFrame.writeStream()	Ecrit les données dans différents formats de fichiers. Enregistre le DataFrame en tant que table dans le catalogue Spark SQL. Ecrit les données en streaming vers une destination.

Spark SQL

Spark Session (et Spark Context le retour)

Spark Context (sc)

Point d'entrée principal dans les anciennes versions de Spark (avant 2.0).
Représente la connexion au cluster Spark et gère les ressources de calcul.
Responsable de la création des RDDs (Resilient Distributed Datasets), la principale abstraction de données dans Spark.
Opérations sur les RDDs effectuées via ce Spark Context.

Spark Session (spark)



Point d'entrée principal dans Spark depuis Spark 2.0.
Regroupe les fonctionnalités de SparkContext, SQLContext et HiveContext dans une seule interface.
Fournit des fonctionnalités avancées pour travailler avec des DataFrames et des Datasets structurés.
Fournit des fonctionnalités pour interagir avec Spark SQL, comme l'exécution de requêtes SQL sur des DataFrames.
Facilite la création et la gestion de DataFrames, ainsi que l'exécution de tâches dans un environnement Spark.

Spark SQL

Spark Session : point de départ

Le point d'entrée dans toutes les fonctionnalités de Spark est la classe `SparkSession`.

```
from pyspark.sql import SparkSession
spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

Spark SQL

Créer des DataFrames

Avec une SparkSession, les applications peuvent créer des DataFrames à partir d'un RDD existant, d'une table Hive ou de sources de données Spark.

Par exemple, le code suivant crée un DataFrame basé sur le contenu d'un fichier JSON :

```
# spark is an existing SparkSession
df = spark.read.json("examples/src/main/resources/people.json")
# Displays the content of the DataFrame to stdout
df.show()
# +---+-----+
# | age| name|
# +---+-----+
# | null|Michael|
# | 30| Andy|
# | 19| Justin|
# +---+-----+
```

Spark SQL

Opérations sur les DataFrames (Datasets non typés)

- Les DataFrames fournissent un langage spécifique au domaine pour la manipulation de données structurées en Scala, Java, Python et R (parfois appelé DSL pour Domain-Specific Language).
- Rappel : Les DataFrames sont simplement des Datasets de lignes dans l'API Scala et Java. Ces opérations sont également appelées "transformations non typées" par opposition aux "transformations typées" associées aux Datasets Scala/Java.
- En Python, il est possible d'accéder aux colonnes d'un DataFrame soit par attribut (`df.age`) soit par indexation (`df['age']`). Bien que le premier soit pratique pour l'exploration interactive des données, il est fortement recommandé d'utiliser le second, qui est évolutif et ne créera pas de conflit lorsque les noms de colonnes seront également des attributs de la classe DataFrame.

Spark SQL

Opérations sur les DataFrames (Datasets non typés)

```
# Print the schema in a tree format
df.printSchema()
# root
# |-- age: Long (nullable = true)
# |-- name: string (nullable = true)

# Select only the "name" column
df.select("name").show()
# +-----+
# | name|
# +-----+
# |Michael|
# | Andy|
# | Justin|
# +-----+

# Select everybody, but increment the age by 1
df.select(df['name'], df['age'] + 1).show()
# +-----+-----+
# | name|(age + 1)
# +-----+-----+
# |Michael| null|
# | Andy| 31|
# | Justin| 20|
# +-----+-----+
```

```
# Select people older than 21
df.filter(df['age'] > 21).show()
# +----+
# |age|name|
# +----+
# | 30|Andy|
# +----+
```

```
# Count people by age
df.groupBy("age").count().show()
# +----+-----+
# | age|count|
# +----+-----+
# | 19| 1|
# | null| 1|
# | 30| 1|
# +----+-----+
```

Spark SQL

Vues temporaires globales

Les vues temporaires dans Spark SQL sont spécifiques à la session et disparaîtront si la session qui les crée se termine. Pour disposer d'une vue temporaire partagée entre toutes les sessions et conservée jusqu'à la fin de l'application Spark, il est possible de créer une vue temporaire globale. La vue temporaire globale est liée à une base de données préservée par le système, `global_temp`, et il faut utiliser le nom qualifié pour y faire référence (par exemple `SELECT * FROM global_temp.view1`).

```
# Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

# Global temporary view is tied to a system preserved database
spark.sql("SELECT * FROM global_temp.people").show()
# +-----+
# | age| name|
# +-----+
# |null|Michael|
# | 30| Andy|
# | 19| Justin|
# +-----+

# Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
# +-----+
# | age| name|
# +-----+
# |null|Michael|
# | 30| Andy|
# | 19| Justin|
# +-----+
```

Spark SQL

Interopérabilité avec les RDD

Spark SQL prend en charge deux méthodes différentes pour convertir des RDD existants en Datasets.

- ➡ La première méthode utilise la « réflexion » (reflection) pour inférer le schéma d'un RDD qui contient des types d'objets spécifiques. Cette approche basée sur la réflexion conduit à un code plus concis et fonctionne bien lorsque le schéma est déjà connu lors de l'écriture de l'application Spark.
- ➡ La deuxième méthode pour créer des Datasets consiste à utiliser une interface programmatique qui permet de construire un schéma, puis de l'appliquer à un RDD existant. Bien que cette méthode soit plus verbuse, elle permet de construire des Datasets lorsque les colonnes et leurs types ne sont pas connus avant l'exécution.

Spark SQL

Interopérabilité avec les RDD

```
from pyspark.sql import Row

sc = spark.sparkContext
# Load a text file and convert each Line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))

# Infer the schema, and register the DataFrame as a table.
schemaPeople = spark.createDataFrame(people)
schemaPeople.createOrReplaceTempView("people")

# SQL can be run over DataFrames that have been registered as a table.
teenagers = spark.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

# The results of SQL queries are Dataframe objects.
# rdd returns the content as an :class:`pyspark.RDD` of :class:`Row`.
teenNames = teenagers.rdd.map(lambda p: "Name: " + p.name).collect()
for name in teenNames:
    print(name)
# Name: Justin
```

Spark SQL

Interopérabilité avec les RDD

```
# Import data types
from pyspark.sql.types import StringType, StructType, StructField

sc = spark.sparkContext

# Load a text file and convert each Line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
# Each Line is converted to a tuple.
people = parts.map(lambda p: (p[0], p[1].strip()))

# The schema is encoded in a string.
schemaString = "name age"

fields = [StructField(field_name, StringType(), True) for field_name in schemaString.split()]
schema = StructType(fields)

# Apply the schema to the RDD.
schemaPeople = spark.createDataFrame(people, schema)

# Creates a temporary view using the DataFrame
schemaPeople.createOrReplaceTempView("people")

# SQL can be run over DataFrames that have been registered as a table.
results = spark.sql("SELECT name FROM people")

results.show()
# +---+
# | name|
# +---+
# |Michael|
# | Andy |
# | Justin|
# +---+
```

Question 7

API de Spark

Quelle est la principale différence entre un DataFrame et un RDD dans Spark ?

- Un DataFrame est une abstraction de données structurées tandis qu'un RDD est une collection distribuée d'objets.
- Un DataFrame est une collection d'enregistrements tandis qu'un RDD est une collection d'objets arbitraires.
- Un DataFrame ne peut pas être utilisé avec Spark SQL.
- Un RDD est une abstraction plus récente que le DataFrame.

Question 8

Les méthodes des DataFrames de Spark

Comment peut-on créer un DataFrame à partir d'un fichier CSV dans Spark ?

- En utilisant la fonction `spark.createDataFrame()`.
- En utilisant la fonction `spark.load.csv()`.
- En utilisant la fonction `spark.read.csv()`.
- En convertissant un RDD en DataFrame.

Question 9

Les méthodes des DataFrames de Spark

Quelle opération est utilisée pour sélectionner des colonnes spécifiques dans un DataFrame ?

- `DataFrame.filter()`
- `DataFrame.select()`
- `DataFrame.groupBy()`
- `DataFrame.orderBy()`

Question 10

Les méthodes des DataFrames de Spark

Quelle opération est utilisée pour filtrer les lignes d'un DataFrame en fonction d'une condition donnée ?

- DataFrame.select()
- DataFrame.groupBy()
- DataFrame.filter()
- DataFrame.sort()

Question 11

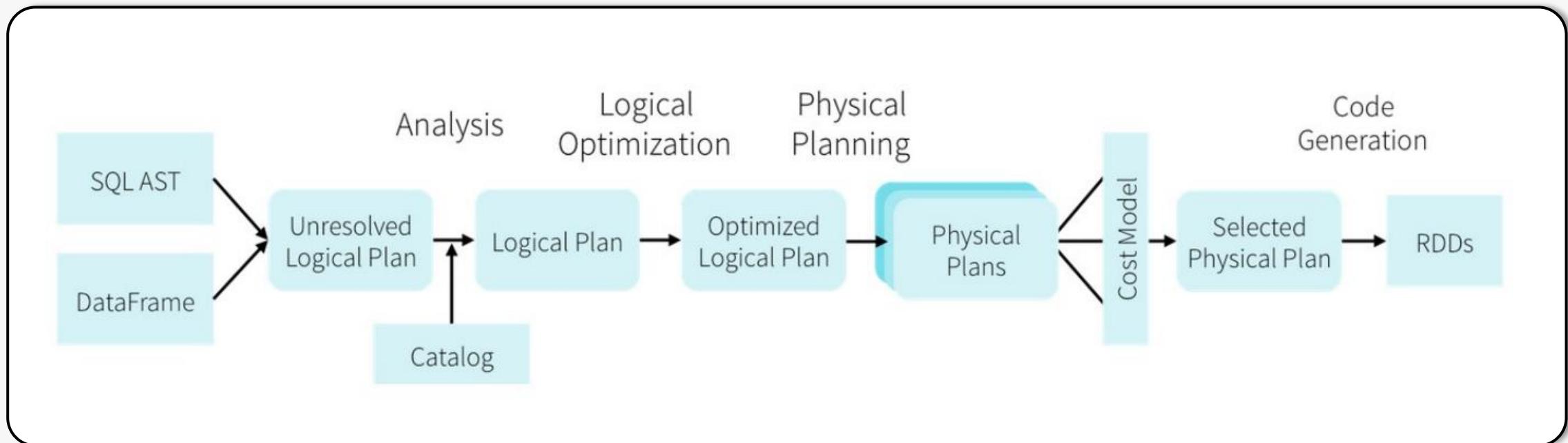
Les méthodes des DataFrames de Spark

Comment peut-on renommer une colonne dans un DataFrame ?

- En utilisant la fonction `DataFrame.renameColumn()`.
- En utilisant la fonction `DataFrame.alterColumn()`.
- En utilisant la fonction `DataFrame.withColumnRenamed()`.
- En utilisant la fonction `DataFrame.updateColumn().DataFrame.filter()`

Spark SQL

Fonctionnement de Catalyst



04

Analyser en temps réel avec Spark Streaming et Kafka

Analyser en temps réel avec Spark Streaming et Kafka

Batch processing et stream processing

Batch processing (traitement par lots)	Stream processing (traitement de flux)
Consiste à traiter un grand volume de données en une seule fois, par lots.	Consiste à traiter les données au fur et à mesure de leur arrivée, de manière continue.
Les données sont collectées, stockées et traitées en blocs à intervalles réguliers ou selon un calendrier défini.	Les données sont traitées au moment où elles sont produites, permettant une réaction en temps réel aux événements.
Convient aux tâches qui peuvent tolérer un certain délai de latence dans le traitement des données, telles que les analyses de données, les rapports générés à intervalles réguliers, etc.	Utilisé dans les cas où une analyse en temps réel des données est nécessaire, comme la surveillance des systèmes, la détection des fraudes, le traitement des événements en temps réel, etc.
 Apache Hadoop avec MapReduce Apache Spark avec les opérations sur les RDD Les travaux cron sur Unix/Linux Etc.	 Apache Kafka Streams Apache Flink Apache Storm Spark Structured Streaming Apache Samza Etc.

Analyser en temps réel avec Spark Streaming et Kafka

Stream processing : applications

- Surveillance des transactions en temps réel pour la détection de fraude.
- Analyse en temps réel du trafic réseau pour la détection des pannes et l'optimisation des performances.
- Surveillance des flux de données pour la détection des attaques et des tentatives d'intrusion.
- Surveillance des signes vitaux en temps réel pour la télémédecine et les soins à distance.
- Surveillance des capteurs IoT pour la gestion des équipements industriels et la maintenance prédictive.
- Diffusion en continu de contenu multimédia en direct.
- Suivi des médias sociaux pour la surveillance de la réputation de la marque et l'analyse des tendances.
- ETL incrémental.
- Apprentissage automatique en ligne.

Analyser en temps réel avec Spark Streaming et Kafka

Stream processing : challenges

- Latence et temps réel
- Gestion du volume des données
- Robustesse et fiabilité
- Traitement des données out-of-order
- Gestion des retardataires
- Garantie de livraison exactly-once
- Maintien d'une grande quantité d'états
- Gestion des déséquilibres de charge
- Joindre des données externes
- Mise à jour des cibles en sortie
- Écriture transactionnelle des données
- Mise à jour de la logique métier en temps réel
- Débogage et monitoring
- Synchronisation et cohérence
- Evolutivité et performance



Analyser en temps réel avec Spark Streaming et Kafka

Batch processing et stream processing

Deux composants principaux :

→ **Système de messagerie**

Sert de canal de communication entre les producteurs de données (sources) et les consommateurs de données (applications ou services).

Généralement équipé d'une capacité de stockage tampon (buffer) pour gérer les flux de données entrants et sortants.

Assure la transmission fiable et efficace des messages à travers le système de streaming.

→ **Moteur de gestion de flux**

Stream Processing Engine

Cœur du système de streaming.

Responsable du traitement, de l'analyse et de la manipulation des flux de données en temps réel.

Exécute des opérations telles que la transformation, l'agrégation, le filtrage, la jointure et d'autres opérations analytiques sur les flux de données entrants.

Prend en charge la logique métier spécifique et les algorithmes de traitement définis par l'utilisateur pour répondre aux besoins applicatifs.

Les deux composants du système de streaming sont concernés par une **sémantique de livraison des messages**.

Analyser en temps réel avec Spark Streaming et Kafka

Sémantiques de livraison

Livraison au plus une fois

At-most once

Un message est livré au consommateur au maximum une fois.

La plus simple à mettre en œuvre : privilégie la faible latence et le débit élevé.

Les messages peuvent être perdus, mais ils ne sont jamais renvoyés.

Livraison au moins une fois

At-least once

Un message est garanti d'être livré au consommateur au moins une fois, garantissant qu'aucun message n'est perdu.

Cela peut entraîner une livraison de messages en double (les messages peuvent être renvoyés plusieurs fois jusqu'à ce qu'ils soient accusés de réception par le consommateur).

Suppose généralement que les opérations de traitement soient idempotentes.

Livraison exactement une fois

Exactly once

Garantit que chaque message est livré au consommateur exactement une fois et une seule, sans perte ni duplication.

Plus complexe à atteindre, et implique généralement des mécanismes de coordination entre le producteur et le consommateur pour suivre la livraison des messages et gérer les échecs de manière efficace.

Analyser en temps réel avec Spark Streaming et Kafka

Garantie de cohérence

Strong Consistency

Chaque lecture après une écriture renvoie toujours la valeur la plus récente. Tous les nœuds du système voient les mêmes données au même moment, ce qui garantit une forte cohérence.

Cela peut entraîner des performances plus faibles et une plus grande latence, car chaque opération nécessite une coordination étroite entre les nœuds pour maintenir la cohérence.

Eventual Consistency

Permet des écarts temporaires dans la cohérence des données.

Après une écriture, il peut y avoir un délai avant que toutes les lectures ne reflètent cette écriture.

Les systèmes basés sur la consistance éventuelle tolèrent les incohérences temporaires en échange de performances plus élevées et d'une latence réduite.

Les conflits potentiels sont résolus à terme à mesure que les mises à jour se propagent à travers le système.

Weak Consistency

Entre la consistance forte et la consistance éventuelle.

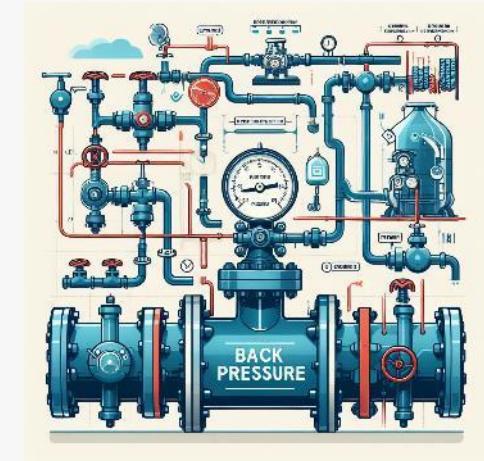
Garantit que les données sont cohérentes à l'intérieur d'une limite de temps spécifiée, mais ne garantit pas que toutes les parties du système voient les mêmes données au même moment.

Permet une certaine flexibilité dans la synchronisation des données, ce qui peut conduire à des performances optimisées tout en maintenant une certaine cohérence.

Analyser en temps réel avec Spark Streaming et Kafka

Backpressure

- Backpressure (ou pression de retour) : capacité d'un système de streaming à gérer et de réguler le flux de données entrantes lorsqu'il est confronté à un débit de données supérieur à sa capacité de traitement.
- Lorsque le système est incapable de traiter les données entrantes à la même vitesse à laquelle elles sont reçues, cela peut entraîner des problèmes tels que des goulots d'étranglement, des retards dans le traitement et éventuellement des pannes.
- Le backpressure permet au système de contrôler le flux de données entrantes en ajustant la vitesse de traitement en fonction de sa capacité actuelle. Lorsque le système est surchargé, il peut ralentir ou même suspendre temporairement le traitement des données jusqu'à ce qu'il puisse rattraper son retard.
- Crucial pour assurer la stabilité et la fiabilité des applications de streaming, en particulier dans des environnements à forte charge de travail.



Analyser en temps réel avec Spark Streaming et Kafka

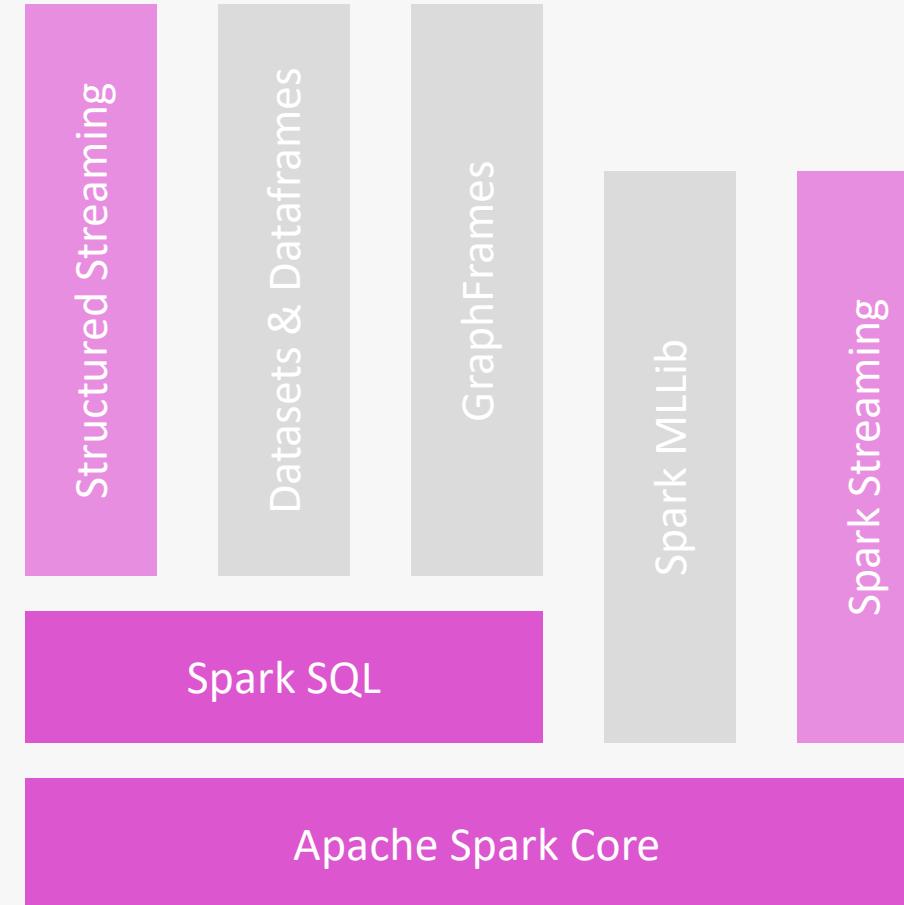
Backpressure

- Lorsque le débit de données entrantes dépasse la capacité de traitement d'un système de streaming, celui-ci utilise des **mémoires tampons** pour temporairement stocker les données en attente de traitement.
- Kafka, un système de messagerie distribué, est souvent utilisé pour gérer la transmission de flux de données entre les producteurs et les consommateurs dans les pipelines de streaming. Il offre des fonctionnalités de mise en mémoire tampon et de réPLICATION DES DONNÉES, ce qui contribue à atténuer les effets de la backpressure en permettant aux consommateurs de traiter les données à leur propre rythme, tout en régulant le flux de données entrantes.



Analyser en temps réel avec Spark Streaming et Kafka

Batch processing et stream processing



Analyser en temps réel avec Spark Streaming et Kafka

Spark Streaming



Source : <https://spark.apache.org/docs/latest/streaming-programming-guide.html>

Analyser en temps réel avec Spark Streaming et Kafka

Spark Streaming

Motivation de Spark Streaming (depuis Spark 0.7 en 2012) :

- Spark est d'abord un système orienté batch avec une extension pour la gestion de flux, utilisant le concept de micro-batches (au dessus de son moteur batch).
- Chaque micro-batch est un job Spark représenté comme un RDD, ce qui constitue les Discretized Streams (DStreams).
- Bien que livré avec des latences plus élevées que l'approche un enregistrement à la fois, Spark offre une meilleure récupération des pannes, notamment avec un traitement exactly-once.
- Spark vise à résoudre les problèmes de tolérance aux pannes et de retardataires.
- Pour la tolérance aux pannes, en général, deux solutions sont envisagées : la réPLICATION et l'upstream backup.
- Cependant, aucune de ces approches ne traite efficacement les retardataires.

Analyser en temps réel avec Spark Streaming et Kafka

Spark Streaming



Source : <https://spark.apache.org/docs/latest/streaming-programming-guide.html>

Analyser en temps réel avec Spark Streaming et Kafka

Spark Streaming : DStreams

Dstream = flux discréteisé, abstraction de base fournie par Spark Streaming

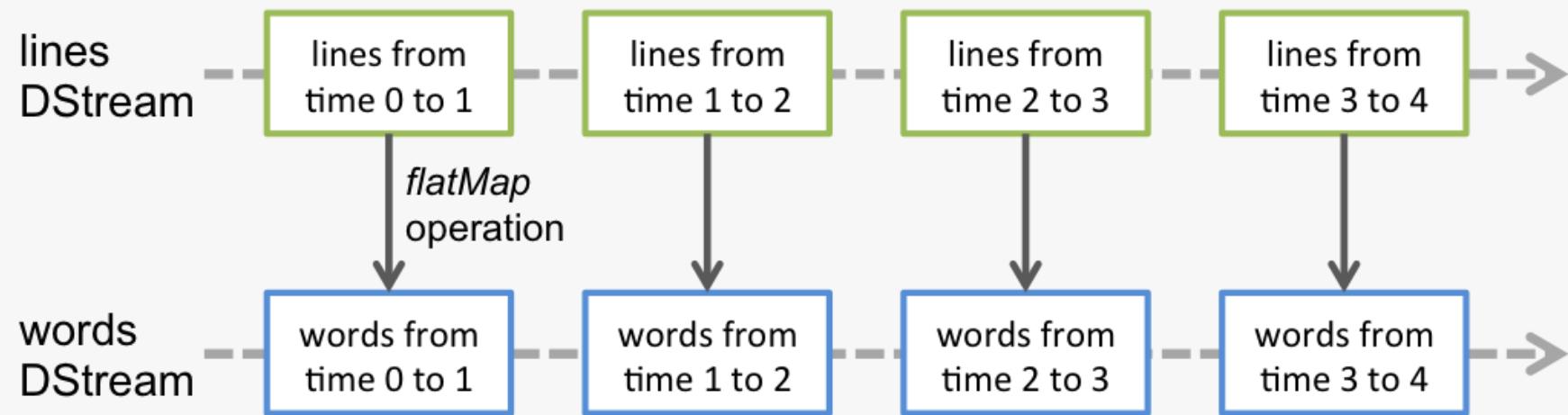
- Il représente un flux continu de données, soit le flux de données d'entrée reçu de la source, soit le flux de données traitées généré par la transformation du flux d'entrée.
- En interne, un DStream est représenté par une série continue de RDD.
- Chaque RDD dans un DStream contient des données d'un certain intervalle :



Analyser en temps réel avec Spark Streaming et Kafka

Spark Streaming : DStreams

Toute opération appliquée à un DStream se traduit par des opérations sur les RDD sous-jacents. Par exemple, dans l'exemple précédent de conversion d'un flux de lignes en mots, l'opération flatMap est appliquée à chaque RDD dans le DStream des lignes pour générer les RDD du DStream de mots.



Ces transformations sous-jacentes de RDD sont calculées par le moteur Spark. Les opérations DStream masquent la plupart de ces détails et nous fournissent une API de plus haut niveau pour plus de commodité.

Analyser en temps réel avec Spark Streaming et Kafka

Spark Streaming : exemple du Word Count

Tout d'abord, on importe StreamingContext, qui est le point d'entrée principal pour toutes les fonctionnalités de streaming. Ensuite, on crée un StreamingContext local avec deux threads d'exécution et un intervalle de lot de 1 seconde.

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Create a Local StreamingContext with two working thread and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
```

Analyser en temps réel avec Spark Streaming et Kafka

Spark Streaming : exemple du Word Count

À l'aide de ce contexte, nous pouvons créer un DStream qui représente des données en streaming à partir d'une source TCP, spécifiée par un nom d'hôte (par exemple localhost) et un port (par exemple 9999).

```
# Create a DStream that will connect to hostname:port, Like Localhost:9999  
lines = ssc.socketTextStream("localhost", 9999)
```

Ce DStream représente le flux de données qui sera reçu du serveur de données. Chaque enregistrement dans ce DStream est une ligne de texte.

Analyser en temps réel avec Spark Streaming et Kafka

Spark Streaming : exemple du Word Count

On veut diviser les lignes en mots (séparer selon les caractères espaces).

```
# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))
```

flatMap est une opération DStream one-to-many qui crée un nouveau DStream en générant plusieurs nouveaux enregistrements à partir de chaque enregistrement dans le DStream source. Dans ce cas, chaque ligne sera divisée en plusieurs mots et le flux de mots est représenté sous forme de DStream de mots.

Analyser en temps réel avec Spark Streaming et Kafka

Spark Streaming : exemple du Word Count

Ensuite, on compte ces mots.

```
# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)

# Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.pprint()
```

Le DStream des mots est ensuite transformé (transformation one-to-one) en un DStream de paires (mot, 1), qui est ensuite réduit pour obtenir la fréquence des mots dans chaque lot de données. `wordCounts.pprint()` affiche des décomptes générés chaque seconde.

Analyser en temps réel avec Spark Streaming et Kafka

Spark Streaming : exemple du Word Count

Lorsque ces lignes sont exécutées, Spark Streaming configure uniquement le calcul qu'il effectuera lorsqu'il sera démarré, et aucun traitement réel n'a encore commencé. Pour démarrer le traitement après que toutes les transformations ont été configurées, il faut exécuter :

```
ssc.start() # Start the computation  
ssc.awaitTermination() # Wait for the computation to terminate
```

Analyser en temps réel avec Spark Streaming et Kafka Structured Streaming

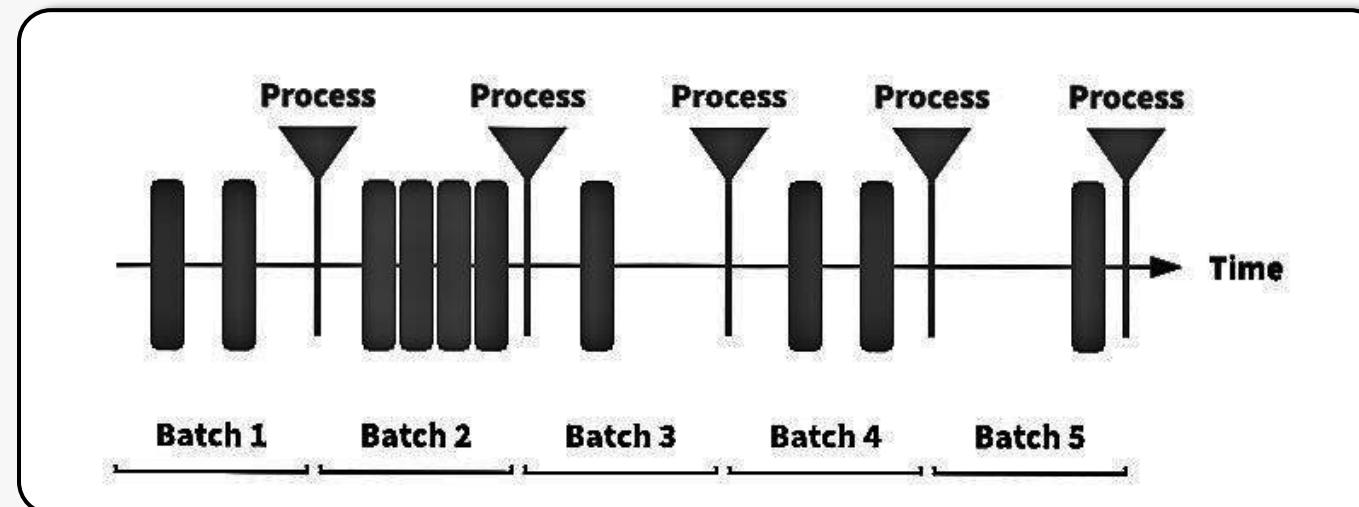
Structured Streaming = moteur de traitement de flux évolutif et tolérant aux pannes construit sur le moteur Spark SQL

- Les calculs en streaming s'expriment de la même manière que ceux pour un lot de données statiques.
- Le moteur Spark SQL se charge de les exécuter de manière incrémentielle et continue et de mettre à jour le résultat final à mesure que les données de streaming continuent d'arriver.
- L'API s'utilise en Scala, Java, Python ou R pour exprimer des agrégations de streaming, des fenêtres de temps événementielles, des jointures de flux à lot...
- Le calcul est exécuté sur le même moteur Spark SQL optimisé.
- Le système garantit la tolérance aux pannes de bout en bout exactement une fois grâce à des checkpoints et à des logs de transactions.

Structured Streaming fournit ce traitement de flux sans que l'utilisateur ait à raisonner sur le streaming.

Analyser en temps réel avec Spark Streaming et Kafka Structured Streaming

- En interne, les requêtes de Structured Streaming sont traitées (par défaut) à l'aide d'un moteur de traitement par micro-lots, qui traite les flux de données comme une série de petits travaux par lot, ce qui permet d'obtenir des latences de bout en bout de l'ordre de la centaine de millisecondes et des garanties de tolérance aux pannes exactly-once.



Source : https://www.researchgate.net/figure/Data-Stream-Processing-Flow-Micro-batching-Micro-batching-based-architecture-takes_fig2_336141201

- Spark 2.3 a introduit un nouveau mode : Continuous Processing.

Analyser en temps réel avec Spark Streaming et Kafka

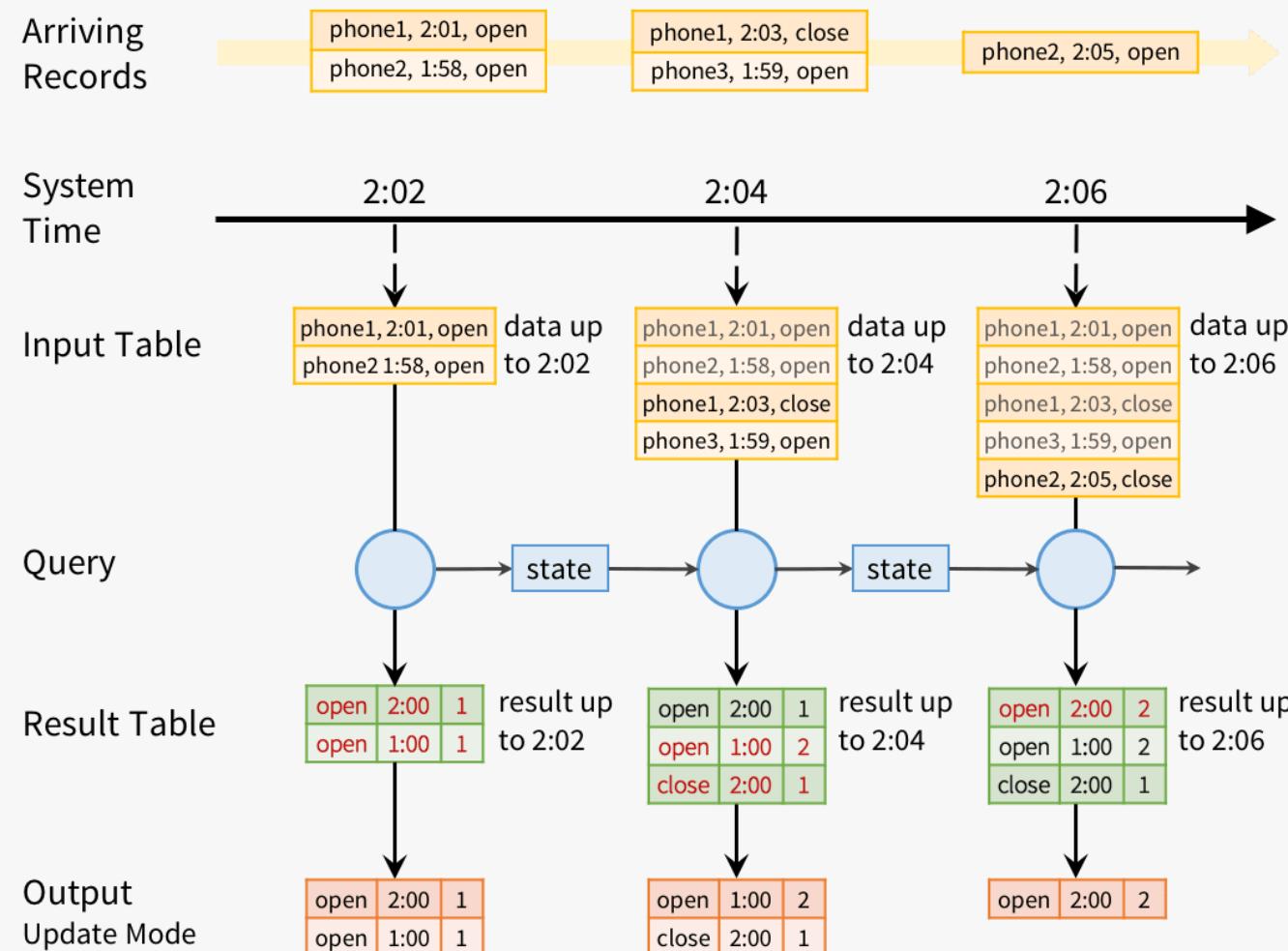
TP



Structured Streaming

Analyser en temps réel avec Spark Streaming et Kafka

Structured Streaming – schéma récapitulatif



Source : <https://www.databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html>

Question 16

Streaming

Quelle est désormais le principal composant utilisée pour le streaming avec Spark ?

- RDD
- MLlib
- Structured Streaming
- MapReduce

Question 17

Streaming

Quel est le principal inconvénient de l'approche de micro-batch utilisée par Spark Streaming ?

- Latence élevée
- Faible tolérance aux pannes
- Complexité de mise en œuvre
- Limitations sur la scalabilité

**Fin de la
première
partie**

Merci !