**Architecture d'un Cluster Apache Spark via Docker Compose**

Ce chapitre a pour objectif de documenter et d'analyser l'architecture d'un cluster Apache Spark complet, déployé localement à l'aide de Docker Compose. Conçu spécifiquement pour le développement, l'expérimentation et le monitoring de solutions de data science et de machine learning, ce cluster constitue un écosystème intégré et performant.

Il est important de noter que l'environnement hôte est un Mac Pro M4 doté de 14 cœurs et de 48 Go de RAM. Ce contexte matériel a directement influencé et justifié les décisions prises en matière d'allocation des ressources au sein du cluster, permettant un dimensionnement généreux et réaliste pour des charges de travail complexes.

**1. Architecture Générale et Composants**

**1.1. Vue d'Ensemble des Services**

Le cluster est défini par un ensemble de services interconnectés via un fichier docker-compose.yml, où chaque conteneur remplit un rôle spécifique et essentiel au sein de l'écosystème global. Cette approche modulaire garantit la clarté, la maintenabilité et la reproductibilité de l'infrastructure.

|  |
| --- |
| **Une image contenant diagramme, capture d’écran, texte, ligne  Le contenu généré par l’IA peut être incorrect.** |

Le tableau ci-dessous liste chaque service déployé et son rôle principal au sein de l'architecture.

|  |  |
| --- | --- |
| Service | Rôle Principal |
| spark-master | Orchestration du cluster, gestion des ressources et planification des applications. |
| spark-worker-1 à 4 | Exécution des tâches (tasks) des applications Spark. Ce sont les nœuds de calcul. |
| spark-submit | Conteneur utilitaire pour soumettre des applications Spark batch au cluster. |
| jupyter | Environnement de développement interactif (JupyterLab) pour le code Python et Scala. |
| mlflow | Serveur de suivi centralisé pour les expérimentations de machine learning. |
| spark-history-server | Interface web pour l'analyse post-mortem des applications Spark terminées. |
| prometheus | Collecte et stockage de métriques de performance sous forme de séries temporelles. |
| grafana | Plateforme de visualisation pour créer des tableaux de bord à partir des métriques de Prometheus. |
| cadvisor | Collecte des métriques de bas niveau des conteneurs Docker (CPU, mémoire, réseau). |

**1.2. Stratégie de Réseau et de Persistance des Données**

Das ce cluster docker nous adoptons une stratégie de gestion du réseau et des données à la fois simple et robuste, adaptée à un environnement de développement.

|  |
| --- |
| Une image contenant texte, capture d’écran, diagramme, Plan  Le contenu généré par l’IA peut être incorrect. |

* **Réseau :** Tous les conteneurs communiquent sur un réseau unique de type bridge nommé spark-network. Cette configuration assure une isolation complète du cluster par rapport au réseau de l'hôte tout en permettant une communication fluide et une résolution de noms simplifiée entre les services (par exemple, spark-worker-1 peut joindre spark-master en utilisant son nom de service).
* **Persistance des Données :** La persistance des données, du code et des configurations est assurée par l'utilisation de volumes Docker. Cette approche mappe des répertoires du système de fichiers de l'hôte vers des répertoires à l'intérieur des conteneurs. Les principaux volumes mappés incluent :
  + /notebooks, /data, /apps: Assurent la persistance du code, des jeux de données et des applications compilées, les rendant accessibles à travers les redémarrages et partagés entre les services pertinents (Jupyter, Master, Workers).
  + /spark-events: Répertoire critique où les applications Spark écrivent leurs logs d'événements, qui sont ensuite lus par le Spark History Server.
  + /mlflow: Contient la base de données SQLite et les artefacts (modèles, graphiques) des expérimentations suivies par MLFlow, garantissant leur pérennité.

Cette approche unifiée de la gestion du réseau et de la persistance des données simplifie non seulement la découverte des services, mais établit également un modèle fondamental pour la réplication de l'environnement et la reprise après sinistre, garantissant que l'ensemble de la pile de développement peut être reconstitué de manière fiable. L'analyse qui suit se concentrera sur les composants Spark, cœur de cette architecture.

**2. Analyse Détaillée des Composants Spark**

**2.1. Le Nœud Maître (spark-master)**

Le service spark-master est le cerveau du cluster. Il agit en tant que coordinateur central, responsable de l'allocation des ressources, de la communication avec les nœuds travailleurs (workers) et de la planification des applications soumises. Sa configuration est optimisée pour un environnement de développement local.

* **Image utilisée :** bitnamilegacy/spark:3.5.3, une image stable et largement utilisée pour les déploiements Spark.
* **Configuration mémoire :** SPARK\_DAEMON\_MEMORY=2g alloue 2 Go de mémoire au processus démon du master, une valeur suffisante pour gérer l'état d'un cluster de cette taille.
* **Variables d'environnement clés :**
  + SPARK\_MODE=master configure le conteneur pour qu'il agisse en tant que nœud maître.
  + Les options SPARK\_RPC\_... et SPARK\_SSL\_... sont explicitement désactivées (no). Cette décision simplifie considérablement la configuration pour un usage local, en éliminant la complexité liée à la gestion des certificats et de l'authentification, qui ne sont pas nécessaires dans ce contexte.
* **Exposition des ports :**
  + 8080: Expose l'interface utilisateur web (UI) du maître Spark, permettant de visualiser l'état du cluster, les workers, et les applications en cours d'exécution.
  + 7077: Le port principal pour la communication RPC. C'est l'URL (spark://spark-master:7077) que les workers et les applications clientes utilisent pour se connecter au maître.
  + 9999: Expose les métriques JMX via un agent JMX Exporter, les rendant disponibles pour être collectées par Prometheus.

**2.2. Les Nœuds Travailleurs (spark-worker)**

Les spark-worker sont les piliers du cluster. Ce sont eux qui fournissent la puissance de calcul en exécutant les *tasks* (tâches) distribuées par le maître. L'architecture déploie quatre de ces nœuds, dont la configuration est homogène.

* **Nombre de workers et ressources totales :** Le cluster est composé de 4 workers. En agrégeant leurs ressources, le cluster dispose d'une capacité de calcul totale de **12 cœurs CPU** (4 workers × 3 cœurs) et de **40 Go de mémoire RAM** (4 workers × 10 Go).
* **Configuration individuelle :** Chaque worker est configuré comme suit :
  + **Mémoire et Cœurs :** SPARK\_WORKER\_MEMORY=10G et SPARK\_WORKER\_CORES=3 définissent les ressources que chaque worker met à la disposition du cluster pour les exécuteurs Spark.
  + **Connexion au Maître :** La variable SPARK\_MASTER\_URL est configurée avec spark://spark-master:7077 pour que chaque worker puisse s'enregistrer auprès du coordinateur.
  + **Configuration JMX :** La variable SPARK\_WORKER\_OPTS est utilisée pour attacher l'agent JMX Exporter. Cela permet à Prometheus de collecter des métriques détaillées sur la performance de la JVM et des exécuteurs tournant sur chaque worker.
* **Image Docker personnalisée :** Les workers utilisent l'image custom-spark-mlflow:3.5.3, qui n'est pas une image standard. Cette image est construite localement pour inclure des dépendances spécifiques, comme détaillé ci-après.

**2.3. Les Images Docker Customiser**

L'utilisation d'images Docker personnalisées est une bonne pratique qui permet de créer des environnements d'exécution reproductibles et d'intégrer des bibliothèques ou des outils spécifiques non présents dans les images de base.

**2.3.1. Image custom-spark-mlflow**

Cette image, construite à partir du fichier Dockerfile.spark-mlflow, est conçue pour l'exécution des tâches de machine learning.

* **Image de base :** Elle étend l'image bitnamilegacy/spark:3.5.3, garantissant une compatibilité totale avec le nœud maître.
* **Objectif :** Son but principal est d'enrichir l'environnement d'exécution Spark avec les bibliothèques Python essentielles pour le machine learning et le suivi d'expérimentations. Cela inclut mlflow, scikit-learn, pandas, matplotlib et seaborn. Crucialement, toutes les bibliothèques Python sont installées avec des versions épinglées, garantissant un environnement d'exécution déterministe et reproductible pour toutes les tâches de machine learning à travers le cluster.
* **Utilisation :** Elle est utilisée par tous les *workers* ainsi que par le conteneur spark-submit. Cette uniformité assure que le code exécuté sur le cluster trouvera les mêmes dépendances, quel que soit le nœud sur lequel une tâche est planifiée, évitant ainsi les erreurs d'environnement.

**2.3.2. Image de Développement Dockerfile.jupyter**

Cette image, définie par Dockerfile.jupyter, crée un environnement de développement complet et préconfiguré.

* **Image de base :** Elle part de quay.io/jupyter/all-spark-notebook:spark-3.5.3, une image de la communauté Jupyter déjà bien fournie.
* **Environnement Polyglotte :** Cette image crée un environnement de développement polyglotte en augmentant la base centrée sur Python avec une chaîne d'outils complète Scala et JDK 11. L'intégration de la distribution Spark complète, plutôt qu'une simple bibliothèque cliente, couplée à l'inclusion des JARs natifs de Delta Lake, permet aux développeurs de construire et de tester des pipelines de données de bout en bout qui exploitent à la fois les API PySpark et Scala sur des formats de données transactionnels modernes.
* **Configuration du Kernel :** Le fichier kernel.json du kernel Scala est modifié. Cette étape est cruciale car elle injecte un ensemble pré-configuré de propriétés Spark directement dans chaque session lancée depuis Jupyter. Cela garantit que toutes les charges de travail interactives bénéficient automatiquement d'un parallélisme optimisé (spark.sql.shuffle.partitions=128), de l'exécution adaptative des requêtes et d'une journalisation cohérente des événements vers /spark-events, ce qui est indispensable pour le Spark History Server.

Ces images personnalisées forment la colonne vertébrale de l'écosystème, qui est complété par des services de développement et de monitoring.

**3. Écosystème de Développement et de Monitoring**

**3.1. Environnement de Développement Intégré (Jupyter)**

Le service jupyter fournit une interface web JupyterLab, un environnement de développement interactif puissant pour écrire, exécuter et déboguer du code Spark en Python et Scala. Sa configuration est finement ajustée pour s'intégrer au cluster.

* **Connexion au Cluster :** La variable SPARK\_MASTER\_URL est définie pour que toutes les sessions Spark initiées depuis Jupyter se connectent automatiquement au maître du cluster.
* **Allocation des Ressources :** Les variables SPARK\_DRIVER\_MEMORY=8g et SPARK\_EXECUTOR\_MEMORY=7g sont d'une importance capitale. Elles définissent les ressources par défaut allouées au *driver* (le programme principal s'exécutant dans le conteneur Jupyter) et aux *executors* (les processus s'exécutant sur les workers) pour chaque application lancée.
* **Activation des Logs d'Événements :** La configuration --conf spark.eventLog.enabled=true passée via la variable SPARK\_OPTS est cruciale. Elle ordonne à chaque application Spark de journaliser ses événements d'exécution dans un répertoire partagé, ce qui est la condition sine qua non pour que le Spark History Server puisse reconstituer et afficher les détails de ces exécutions.

**3.2. Suivi d'Expérimentations (MLFlow)**

Le service mlflow agit comme un serveur de suivi centralisé pour les expérimentations de machine learning. Il offre une interface web et une API pour logger de manière structurée les paramètres, les métriques, et les artefacts (comme les modèles entraînés ou les graphiques de performance) de chaque exécution d'un modèle. Dans cette architecture, il s'appuie sur une base de données SQLite pour les métadonnées et sur le système de fichiers local pour le stockage des artefacts, tous deux rendus persistants grâce à des volumes Docker.

**3.3. Analyse Post-Mortem (Spark History Server)**

Le service spark-history-server fournit une interface web qui permet de consulter les détails d'exécution des applications Spark qui sont terminées. Il est essentiel pour le débogage et l'optimisation, car il permet d'analyser les plans d'exécution (DAGs), de consulter les logs des tâches et d'examiner les métriques de performance de manière détaillée. Son fonctionnement repose entièrement sur le volume partagé /spark-events, où les applications comme celles lancées depuis Jupyter écrivent leurs logs.

**3.4. Pile de Monitoring en Temps Réel**

Une pile de monitoring complète est intégrée pour observer la santé et les performances du cluster en temps réel. Son architecture est composée de quatre services complémentaires :

|  |
| --- |
|  |

* **cAdvisor :** Collecte des métriques de bas niveau directement depuis le démon Docker de l'hôte. Il fournit des informations sur l'utilisation des ressources (CPU, mémoire, réseau, I/O disque) pour chaque conteneur en cours d'exécution.
* **JMX Exporter :** Il fonctionne comme un agent Java au sein de chaque JVM Spark (spark-master, spark-worker-\*), récupérant les MBeans JMX internes et les traduisant dans un format compatible avec Prometheus, exposé via un endpoint HTTP.
* **Prometheus :** Agit comme le moteur central de la pile. Il interroge périodiquement (scrape) les endpoints exposés par cAdvisor et les différents JMX Exporters. Il stocke ensuite ces métriques dans une base de données de séries temporelles optimisée pour les requêtes et les alertes.
* **Grafana :** Est la couche de visualisation. Elle se connecte à Prometheus comme source de données et permet de construire des tableaux de bord interactifs et riches. Ces tableaux de bord permettent une analyse fine et visuelle de la performance du cluster, des applications Spark et des conteneurs individuels.

Cette architecture est non seulement fonctionnelle, mais elle applique également plusieurs bonnes pratiques reconnues dans l'écosystème Spark.

**4. Analyse des Bonnes Pratiques Appliquées**

**4.1. Dimensionnement des Exécuteurs et Allocation des Ressources**

Le dimensionnement correct des exécuteurs Spark est l'un des facteurs les plus critiques pour optimiser les performances d'un job. Un mauvais dimensionnement peut entraîner une sous-utilisation des ressources, des erreurs de mémoire ou une surcharge du Garbage Collector (GC).

|  |
| --- |
| Une image contenant texte, capture d’écran, Police, ligne  Le contenu généré par l’IA peut être incorrect. |

* **Nombre de Cœurs par Exécuteur :** Le choix de configurer chaque worker avec SPARK\_WORKER\_CORES=3 est judicieux. Une valeur se situant dans la fourchette de 3 à 5 cœurs par exécuteur est souvent un compromis optimal. Un nombre de cœurs trop élevé (> 5) peut entraîner une dégradation des performances due à deux facteurs distincts : une faible bande passante HDFS et une activité excessive du Garbage Collector (GC). Un nombre trop faible mène à une gestion inefficace des ressources. Avec 3 cœurs, cette configuration favorise un bon parallélisme des tâches tout en limitant les frais généraux.
* **Allocation Mémoire :** La configuration SPARK\_WORKER\_MEMORY=10G alloue une quantité de mémoire significative à chaque worker. Cette mémoire n'est pas un bloc monolithique. Elle est divisée en plusieurs régions gérées par Spark : la mémoire d'exécution (pour les shuffles, joins, sorts), la mémoire de stockage (pour le caching), la mémoire utilisateur (pour les objets et structures de données définis par l'utilisateur) et une mémoire d'overhead. L'allocation de 10 Go représente un bon équilibre entre les approches "fat executor" (beaucoup de mémoire, risque de longs cycles de GC) et "thin executor" (peu de mémoire, risque de "spill" sur disque fréquent).

**4.2. Gestion du Partitionnement et du Parallélisme**

La manière dont les données sont partitionnées influence directement le degré de parallélisme et, par conséquent, la performance des requêtes. Les paramètres définis dans le fichier kernel.json pour les sessions Jupyter reflètent une stratégie de partitionnement proactive.

* spark.sql.shuffle.partitions=128 et spark.default.parallelism=128 : Ces paramètres contrôlent le nombre de partitions créées après une opération de *shuffle* (comme un groupBy ou un join). Ce nombre définit le degré de parallélisme pour les étapes suivantes du job. Fixer cette valeur à 128 est un point de départ raisonnable pour un cluster local disposant de 12 cœurs au total. Cela assure qu'il y aura suffisamment de tâches à distribuer pour maintenir tous les cœurs occupés sans pour autant créer une surcharge due à la gestion d'un trop grand nombre de petites partitions.
* spark.sql.files.maxPartitionBytes=64m : Ce paramètre, mentionné dans "How Partitioning Works", contrôle la taille maximale d'une partition lors de la lecture de fichiers depuis une source de données. En fixant cette limite à 64 Mo, on s'assure que Spark ne créera pas de partitions initiales trop grandes. Des partitions trop volumineuses peuvent entraîner des erreurs de mémoire sur des tâches individuelles avant même que le traitement ne commence réellement. Cette valeur est une bonne pratique pour équilibrer la taille des fichiers et le nombre de tâches initiales.

**4.3. Optimisation Adaptative des Requêtes (AQE)**

L'Adaptive Query Execution (AQE), introduite dans Spark 3.x, est l'une des optimisations les plus significatives de ces dernières années. Elle permet à Spark d'ajuster le plan d'exécution de la requête *en cours de vol* (mid-flight), en utilisant des statistiques en temps réel issues des étapes terminées, ce qui constitue une rupture significative avec la planification statique des versions antérieures.

Les paramètres spark.sql.adaptive.enabled=true et spark.sql.adaptive.coalescePartitions.enabled=true dans le kernel.json activent cette fonctionnalité clé. Cette configuration apporte des bénéfices majeurs :

1. **Fusion de Partitions :** AQE peut détecter lorsque des partitions deviennent trop petites après une opération de filtrage ou de shuffle. Grâce à coalescePartitions, il les fusionne automatiquement en partitions de taille plus raisonnable. Cela réduit drastiquement la surcharge liée à la planification et à la gestion de milliers de petites tâches et de petits fichiers, un problème courant dans les jobs Spark.
2. **Gestion de l'Asymétrie des Données (Data Skew) :** Le *data skew* se produit lorsqu'une ou plusieurs partitions sont beaucoup plus volumineuses que les autres, créant des goulots d'étranglement. AQE peut détecter ces partitions asymétriques au moment du shuffle et les diviser dynamiquement en sous-partitions plus petites. Cette technique assure une meilleure répartition de la charge de travail entre les exécuteurs et peut réduire de manière spectaculaire le temps d'exécution des jobs affectés par ce problème.

Cette configuration proactive démontre une compréhension approfondie des mécanismes d'optimisation modernes de Spark.

**6. Conclusion**

En synthèse, l'architecture de ce cluster Apache Spark déployé via Docker Compose constitue un écosystème de développement local complet, puissant et remarquablement bien configuré. Sa force ne réside pas seulement dans le déploiement des composants Spark de base, mais dans l'intégration réfléchie d'un ensemble d'outils qui couvrent l'ensemble du cycle de vie d'un projet de data science.

L'inclusion de services pour le développement interactif (Jupyter), le suivi rigoureux d'expérimentations (MLFlow), et le monitoring en temps réel de la performance (Prometheus/Grafana) crée un environnement de travail intégré et efficace. De plus, la configuration du cluster démontre une application méticuleuse des meilleures pratiques d'optimisation de Spark, notamment en ce qui concerne le dimensionnement des exécuteurs, la gestion proactive du partitionnement et l'activation de l'optimisation adaptative des requêtes (AQE). Cette combinaison de fonctionnalités et d'optimisations en fait une plateforme exemplaire, idéale pour prototyper, développer et affiner des solutions de traitement de données et de machine learning complexes.