Solutions de stockage NetApp pour Apache Spark

NetApp artificial intelligence solutions

NetApp
August 18, 2025

Sommaire

Solutions de stockage NetApp pour Apache Spark

TR-4570 : Solutions de stockage NetApp pour Apache Spark : architecture, cas d'utilisation et résultats de performance

Rick Huang, Karthikeyan Nagalingam, NetApp

Ce document se concentre sur l'architecture Apache Spark, les cas d'utilisation des clients et le portefeuille de stockage NetApp liés à l'analyse du Big Data et à l'intelligence artificielle (IA). Il présente également divers résultats de tests utilisant des outils d'IA, d'apprentissage automatique (ML) et d'apprentissage profond (DL) standard de l'industrie par rapport à un système Hadoop typique afin que vous puissiez choisir la solution Spark appropriée. Pour commencer, vous avez besoin d'une architecture Spark, de composants appropriés et de deux modes de déploiement (cluster et client).

Ce document fournit également des cas d'utilisation client pour résoudre les problèmes de configuration et présente un aperçu du portefeuille de stockage NetApp pertinent pour l'analyse du Big Data et l'IA, le ML et le DL avec Spark. Nous terminons ensuite avec les résultats des tests dérivés des cas d'utilisation spécifiques à Spark et du portefeuille de solutions NetApp Spark.

Les défis des clients

Cette section se concentre sur les défis des clients en matière d'analyse de Big Data et d'IA/ML/DL dans les secteurs de croissance des données tels que la vente au détail, le marketing numérique, la banque, la fabrication discrète, la fabrication de processus, le gouvernement et les services professionnels.

Des performances imprévisibles

Les déploiements Hadoop traditionnels utilisent généralement du matériel standard. Pour améliorer les performances, vous devez ajuster le réseau, le système d'exploitation, le cluster Hadoop, les composants de l'écosystème tels que Spark et le matériel. Même si vous ajustez chaque couche, il peut être difficile d'atteindre les niveaux de performances souhaités, car Hadoop s'exécute sur du matériel standard qui n'a pas été conçu pour des performances élevées dans votre environnement.

Pannes de médias et de nœuds

Même dans des conditions normales, le matériel informatique de base est sujet à des pannes. Si un disque sur un nœud de données tombe en panne, le maître Hadoop considère par défaut que ce nœud est défectueux. Il copie ensuite des données spécifiques de ce nœud sur le réseau à partir de répliques vers un nœud sain. Ce processus ralentit les paquets réseau pour tous les travaux Hadoop. Le cluster doit ensuite recopier les données et supprimer les données sur-répliquées lorsque le nœud défectueux revient à un état sain.

Verrouillage du fournisseur Hadoop

Les distributeurs Hadoop ont leur propre distribution Hadoop avec leur propre versionnage, ce qui verrouille le client sur ces distributions. Cependant, de nombreux clients ont besoin d'une prise en charge des analyses en mémoire qui ne lie pas le client à des distributions Hadoop spécifiques. Ils ont besoin de la liberté de modifier

les distributions tout en conservant leurs analyses avec eux.

Manque de support pour plus d'une langue

Les clients ont souvent besoin de la prise en charge de plusieurs langues en plus des programmes Java MapReduce pour exécuter leurs tâches. Des options telles que SQL et les scripts offrent plus de flexibilité pour obtenir des réponses, plus d'options pour organiser et récupérer des données, et des moyens plus rapides de déplacer des données dans un cadre d'analyse.

Difficulté d'utilisation

Depuis un certain temps, les gens se plaignent de la difficulté d'utilisation d'Hadoop. Même si Hadoop est devenu plus simple et plus puissant à chaque nouvelle version, cette critique a persisté. Hadoop exige que vous compreniez les modèles de programmation Java et MapReduce, un défi pour les administrateurs de bases de données et les personnes possédant des compétences de script traditionnelles.

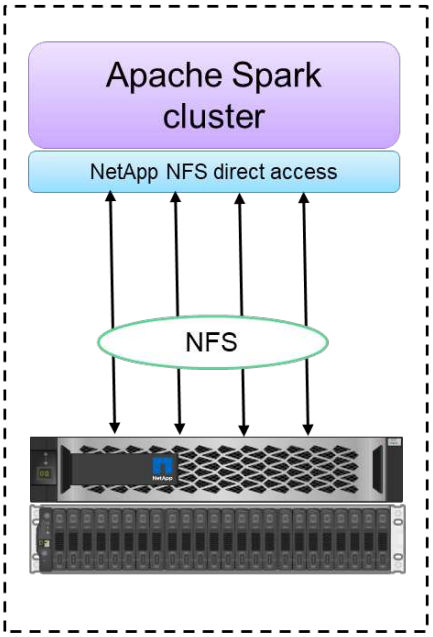
Cadres et outils complexes

Les équipes d'IA des entreprises sont confrontées à de multiples défis. Même avec des connaissances spécialisées en science des données, les outils et les cadres pour différents écosystèmes de déploiement et applications peuvent ne pas être facilement transposables de l'un à l'autre. Une plateforme de science des données doit s'intégrer de manière transparente aux plateformes Big Data correspondantes construites sur Spark avec une facilité de déplacement des données, des modèles réutilisables, du code prêt à l'emploi et des outils qui prennent en charge les meilleures pratiques de prototypage, de validation, de versionnage, de partage, de réutilisation et de déploiement rapide des modèles en production.

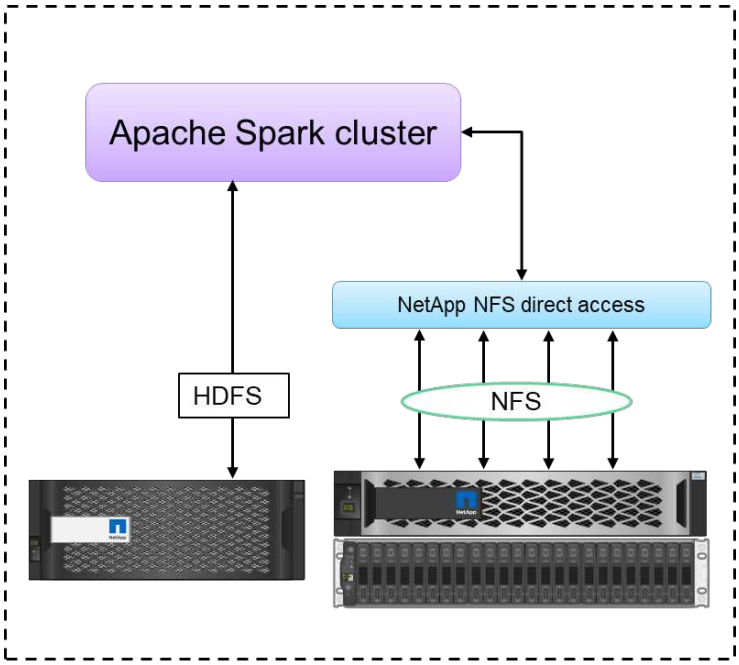
Pourquoi choisir NetApp?

NetApp peut améliorer votre expérience Spark des manières suivantes :

- L'accès direct NetApp NFS (illustré dans la figure ci-dessous) permet aux clients d'exécuter des tâches d'analyse de Big Data sur leurs données NFSv3 ou NFSv4 existantes ou nouvelles sans déplacer ni copier les données. Il empêche les copies multiples de données et élimine le besoin de synchroniser les données avec une source.
- Stockage plus efficace et moins de réplication de serveur. Par exemple, la solution Hadoop NetApp E-Series nécessite deux répliques de données au lieu de trois, et la solution Hadoop FAS nécessite une source de données mais aucune réplication ni copie de données. Les solutions de stockage NetApp génèrent également moins de trafic de serveur à serveur.
- Meilleur comportement des tâches et des clusters Hadoop en cas de panne de lecteur et de nœud.
- Meilleures performances d'ingestion de données.



Configuration 1: NFS as primary storage



Configuration 2: HDFS and NFS in single Spark cluster

Par exemple, dans le secteur financier et de la santé, le déplacement de données d'un endroit à un autre doit répondre à des obligations légales, ce qui n'est pas une tâche facile. Dans ce scénario, l'accès direct NetApp NFS analyse les données financières et de santé à partir de leur emplacement d'origine. Un autre avantage clé est que l'utilisation de l'accès direct NetApp NFS simplifie la protection des données Hadoop en utilisant des commandes Hadoop natives et en activant les flux de travail de protection des données avec le riche portefeuille de gestion des données de NetApp.

L'accès direct NetApp NFS offre deux types d'options de déploiement pour les clusters Hadoop/Spark :

- Par défaut, les clusters Hadoop ou Spark utilisent le système de fichiers distribué Hadoop (HDFS) pour le stockage des données et le système de fichiers par défaut. L'accès direct NetApp NFS peut remplacer le HDFS par défaut par le stockage NFS comme système de fichiers par défaut, permettant ainsi des analyses directes sur les données NFS.
- Dans une autre option de déploiement, l'accès direct NetApp NFS prend en charge la configuration de NFS comme stockage supplémentaire avec HDFS dans un seul cluster Hadoop ou Spark. Dans ce cas, le client peut partager des données via des exportations NFS et y accéder à partir du même cluster avec les données HDFS.

Les principaux avantages de l'utilisation de l'accès direct NetApp NFS sont les suivants :

- Analyse des données à partir de leur emplacement actuel, ce qui évite la tâche fastidieuse et coûteuse en performances consistant à déplacer les données d'analyse vers une infrastructure Hadoop telle que HDFS.
- Réduction du nombre de répliques de trois à une.
- Permettre aux utilisateurs de découpler le calcul et le stockage pour les faire évoluer indépendamment.
- Assurer la protection des données d'entreprise en exploitant les riches capacités de gestion des données d' ONTAP.
- Certification avec la plateforme de données Hortonworks.
- Permettre des déploiements d'analyse de données hybrides.
- Réduction du temps de sauvegarde en exploitant la capacité multithread dynamique.

Voir"TR-4657 : Solutions de données cloud hybrides NetApp - Spark et Hadoop basées sur des cas d'utilisation client" pour la sauvegarde des données Hadoop, la sauvegarde et la reprise après sinistre du cloud vers les locaux, permettant DevTest sur les données Hadoop existantes, la protection des données et la connectivité multicloud, et l'accélération des charges de travail d'analyse.

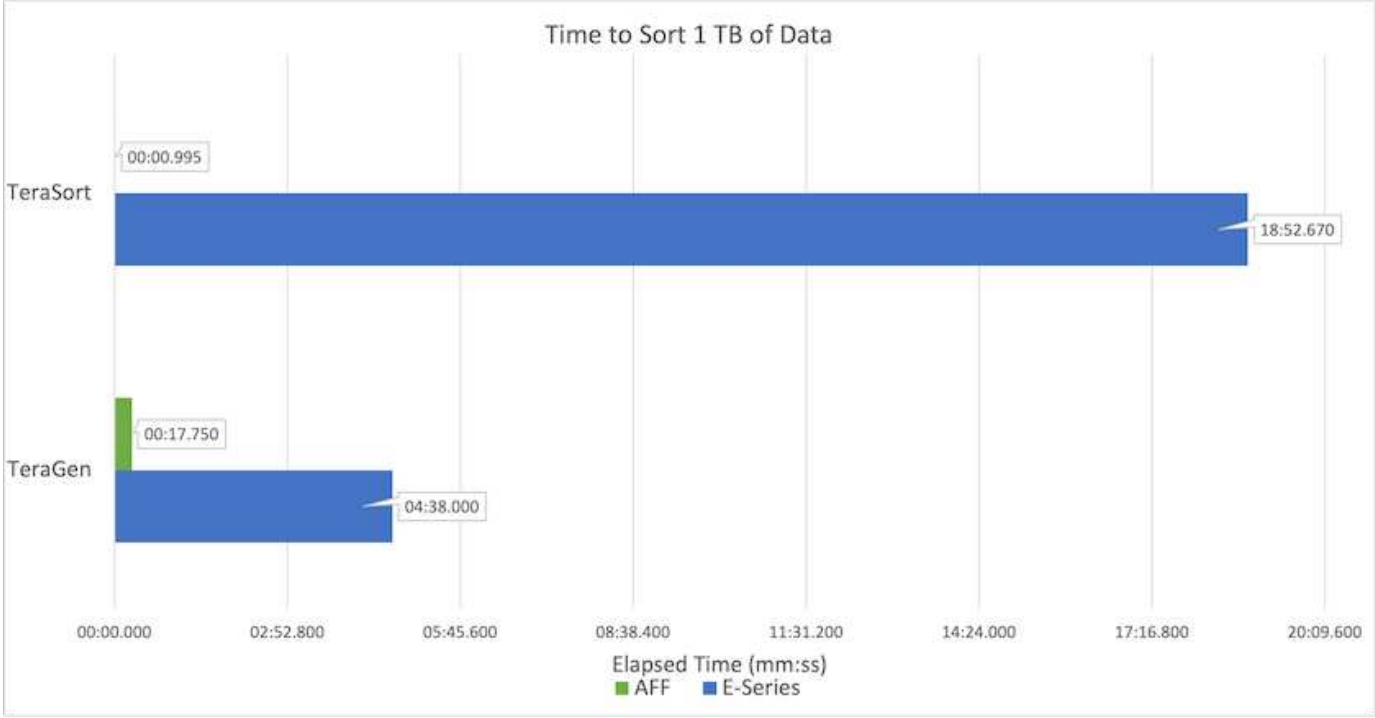
Les sections suivantes décrivent les capacités de stockage importantes pour les clients Spark.

hiérarchisation du stockage

Avec la hiérarchisation du stockage Hadoop, vous pouvez stocker des fichiers avec différents types de stockage conformément à une politique de stockage. Les types de stockage incluent `hot` , `cold` , `warm` , `all_ssd` , `one_ssd` , et `lazy_persist` .

Nous avons effectué la validation de la hiérarchisation du stockage Hadoop sur un contrôleur de stockage NetApp AFF et un contrôleur de stockage E-Series avec des disques SSD et SAS avec différentes politiques de stockage. Le cluster Spark avec AFF-A800 dispose de quatre nœuds de calcul, tandis que le cluster avec la série E en a huit. Il s'agit principalement de comparer les performances des disques SSD (Solid State Drives) par rapport aux disques durs (HDD).

La figure suivante montre les performances des solutions NetApp pour un SSD Hadoop.



- La configuration NL-SAS de base utilisait huit nœuds de calcul et 96 lecteurs NL-SAS. Cette configuration a généré 1 To de données en 4 minutes et 38 secondes. Voir "TR-3969 Solution NetApp E-Series pour Hadoop" pour plus de détails sur la configuration du cluster et du stockage.
- Grâce à TeraGen, la configuration SSD a généré 1 To de données 15,66 fois plus rapidement que la configuration NL-SAS. De plus, la configuration SSD utilisait la moitié du nombre de nœuds de calcul et la moitié du nombre de lecteurs de disque (24 lecteurs SSD au total). En fonction du temps d'exécution du travail, il était presque deux fois plus rapide que la configuration NL-SAS.
- Grâce à TeraSort, la configuration SSD a trié 1 To de données 1 138,36 fois plus rapidement que la configuration NL-SAS. De plus, la configuration SSD utilisait la moitié du nombre de nœuds de calcul et la moitié du nombre de lecteurs de disque (24 lecteurs SSD au total). Par conséquent, par lecteur, il était environ trois fois plus rapide que la configuration NL-SAS.

- Le point à retenir est que la transition des disques rotatifs vers le tout flash améliore les performances. Le nombre de nœuds de calcul n'était pas le goulot d'étranglement. Avec le stockage entièrement flash de NetApp, les performances d'exécution évoluent bien.
- Avec NFS, les données étaient fonctionnellement équivalentes à un regroupement complet, ce qui peut réduire le nombre de nœuds de calcul en fonction de votre charge de travail. Les utilisateurs du cluster Apache Spark n'ont pas besoin de rééquilibrer manuellement les données lors du changement du nombre de nœuds de calcul.

Mise à l'échelle des performances - Évolution horizontale

Lorsque vous avez besoin de plus de puissance de calcul à partir d'un cluster Hadoop dans une solution AFF , vous pouvez ajouter des nœuds de données avec un nombre approprié de contrôleurs de stockage. NetApp recommande de commencer avec quatre nœuds de données par baie de contrôleurs de stockage et d'augmenter le nombre à huit nœuds de données par contrôleur de stockage, en fonction des caractéristiques de la charge de travail.

AFF et FAS sont parfaits pour les analyses sur place. En fonction des exigences de calcul, vous pouvez ajouter des gestionnaires de nœuds et des opérations non perturbatrices vous permettent d'ajouter un contrôleur de stockage à la demande sans temps d'arrêt. Nous proposons des fonctionnalités riches avec AFF et FAS, telles que la prise en charge des médias NVME, l'efficacité garantie, la réduction des données, la qualité de service, l'analyse prédictive, la hiérarchisation du cloud, la réplication, le déploiement du cloud et la sécurité. Pour aider les clients à répondre à leurs besoins, NetApp propose des fonctionnalités telles que l'analyse du système de fichiers, les quotas et l'équilibrage de charge sur site sans frais de licence supplémentaires. NetApp offre de meilleures performances en termes de nombre de tâches simultanées, de latence plus faible, d'opérations plus simples et de débit en gigaoctets par seconde plus élevé que nos concurrents. De plus, NetApp Cloud Volumes ONTAP fonctionne sur les trois principaux fournisseurs de cloud.

Mise à l'échelle des performances - Mise à l'échelle

Les fonctionnalités de mise à l'échelle vous permettent d'ajouter des lecteurs de disque aux systèmes AFF, FAS et E-Series lorsque vous avez besoin d'une capacité de stockage supplémentaire. Avec Cloud Volumes ONTAP, la mise à l'échelle du stockage au niveau Po est une combinaison de deux facteurs : la hiérarchisation des données rarement utilisées vers le stockage d'objets à partir du stockage en blocs et l'empilement de licences Cloud Volumes ONTAP sans calcul supplémentaire.

Protocoles multiples

Les systèmes NetApp prennent en charge la plupart des protocoles pour les déploiements Hadoop, notamment SAS, iSCSI, FCP, InfiniBand et NFS.

Solutions opérationnelles et supportées

Les solutions Hadoop décrites dans ce document sont prises en charge par NetApp. Ces solutions sont également certifiées auprès des principaux distributeurs Hadoop. Pour plus d'informations, consultez le "Hortonworks" site et Cloudera "certification" et "partenaire" sites.

Public cible

Le monde de l'analyse et de la science des données touche de multiples disciplines de l'informatique et des affaires :

- Le data scientist a besoin de la flexibilité nécessaire pour utiliser les outils et les bibliothèques de son choix.

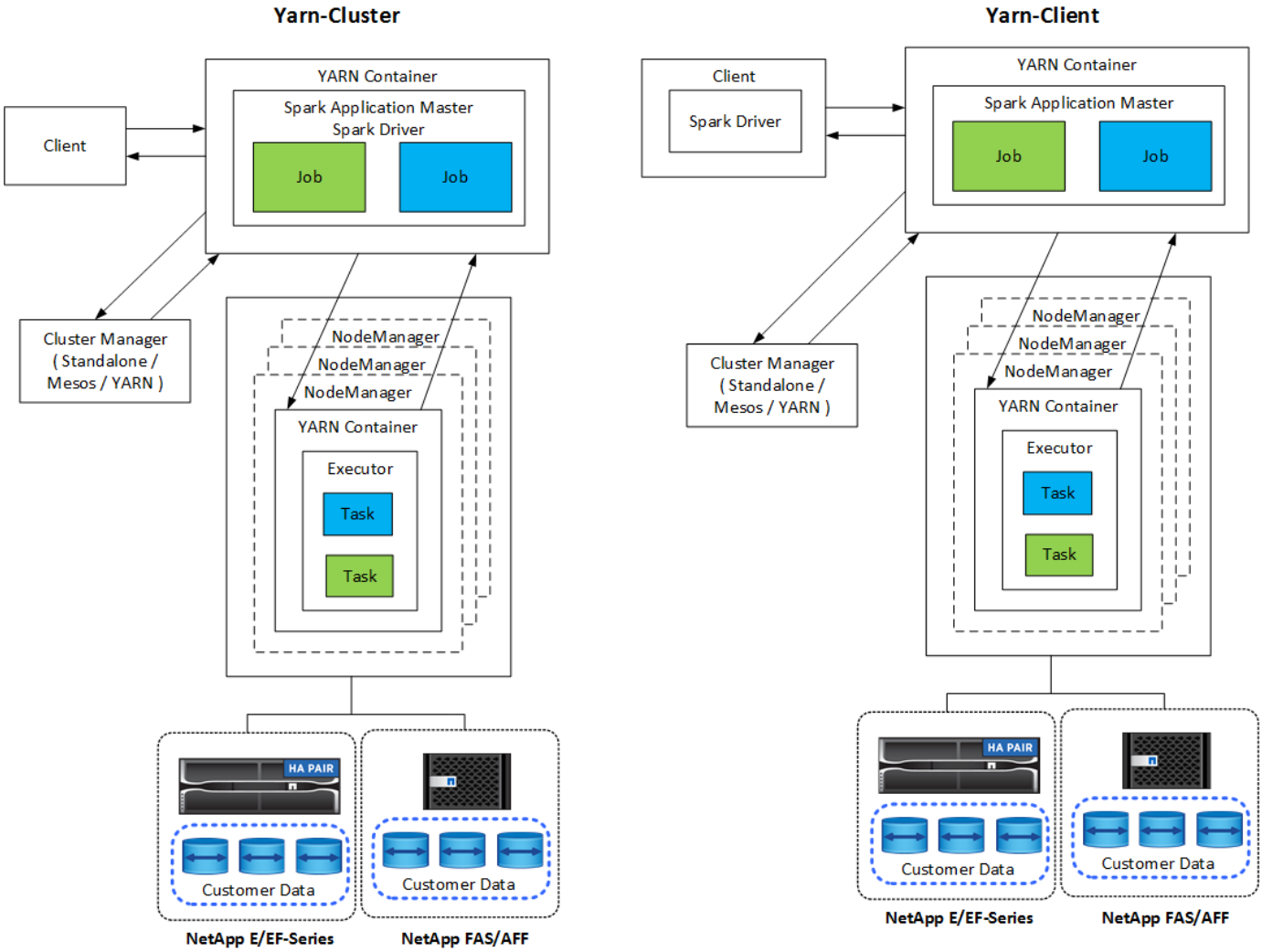
- L’ingénieur de données doit savoir comment les données circulent et où elles résident.
- Un ingénieur DevOps a besoin d’outils pour intégrer de nouvelles applications d’IA et de ML dans ses pipelines CI et CD.
- Les administrateurs et architectes cloud doivent être capables de configurer et de gérer les ressources cloud hybrides.
- Les utilisateurs professionnels souhaitent avoir accès aux applications d’analyse, d’IA, de ML et de DL.

Dans ce rapport technique, nous décrivons comment NetApp AFF, E-Series, StorageGRID, l’accès direct NFS, Apache Spark, Horovod et Keras aident chacun de ces rôles à apporter de la valeur à l’entreprise.

Technologie des solutions

Apache Spark est un framework de programmation populaire pour l’écriture d’applications Hadoop qui fonctionne directement avec le système de fichiers distribué Hadoop (HDFS). Spark est prêt pour la production, prend en charge le traitement des données en streaming et est plus rapide que MapReduce. Spark dispose d’une mise en cache de données en mémoire configurable pour une itération efficace, et le shell Spark est interactif pour l’apprentissage et l’exploration des données. Avec Spark, vous pouvez créer des applications en Python, Scala ou Java. Les applications Spark se composent d’un ou plusieurs travaux comportant une ou plusieurs tâches.

Chaque application Spark dispose d’un pilote Spark. En mode YARN-Client, le pilote s’exécute localement sur le client. En mode YARN-Cluster, le pilote s’exécute dans le cluster sur le maître d’application. En mode cluster, l’application continue de s’exécuter même si le client se déconnecte.



Il existe trois gestionnaires de clusters :

- **Autonome.** Ce gestionnaire fait partie de Spark, ce qui facilite la configuration d'un cluster.
- **Apache Mesos.** Il s'agit d'un gestionnaire de cluster général qui exécute également MapReduce et d'autres applications.
- **FIL Hadoop.** Il s'agit d'un gestionnaire de ressources dans Hadoop 3.

L'ensemble de données distribué résilient (RDD) est le composant principal de Spark. RDD recrée les données perdues et manquantes à partir des données stockées en mémoire dans le cluster et stocke les données initiales provenant d'un fichier ou créées par programmation. Les RDD sont créés à partir de fichiers, de données en mémoire ou d'un autre RDD. La programmation Spark effectue deux opérations : la transformation et les actions. La transformation crée un nouveau RDD basé sur un RDD existant. Les actions renvoient une valeur à partir d'un RDD.

Les transformations et les actions s'appliquent également aux ensembles de données et aux DataFrames Spark. Un ensemble de données est une collection distribuée de données qui offre les avantages des RDD (typage fort, utilisation de fonctions lambda) avec les avantages du moteur d'exécution optimisé de Spark SQL. Un ensemble de données peut être construit à partir d'objets JVM, puis manipulé à l'aide de transformations fonctionnelles (map, flatMap, filter, etc.). Un DataFrame est un ensemble de données organisé en colonnes nommées. Il est conceptuellement équivalent à une table dans une base de données relationnelle ou à un cadre de données dans R/Python. Les DataFrames peuvent être construits à partir d'un large éventail de sources telles que des fichiers de données structurés, des tables dans Hive/HBase, des bases de données

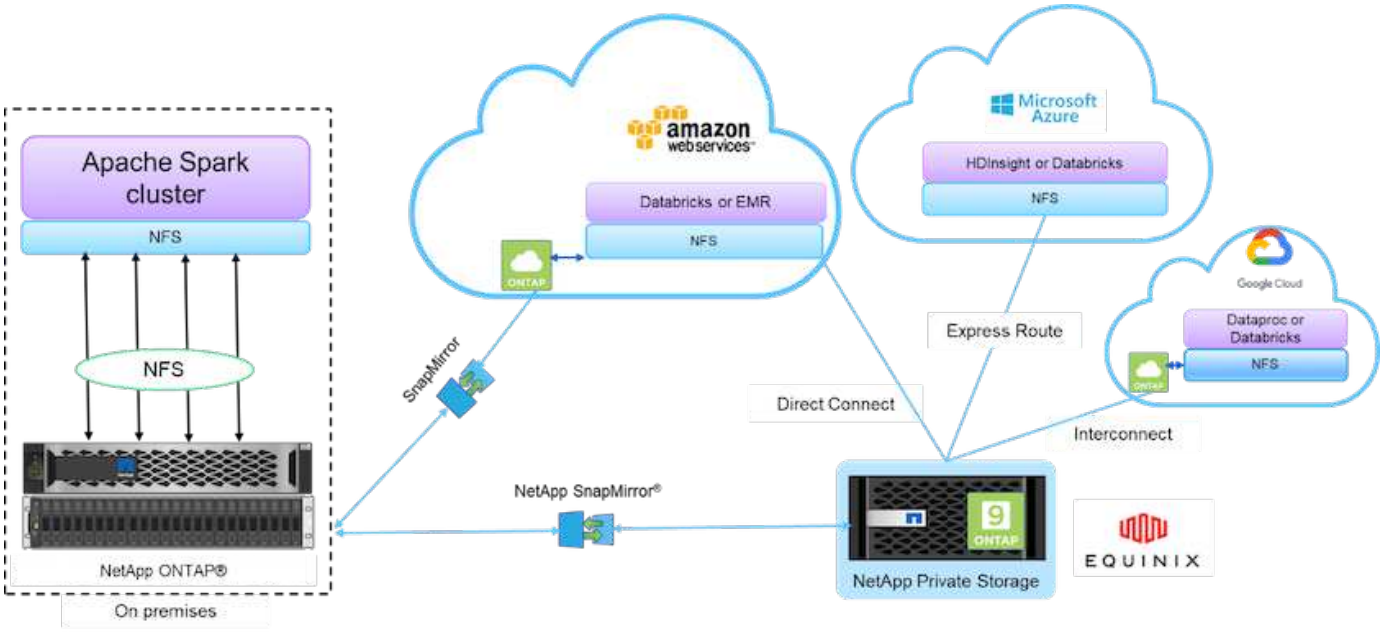
externes sur site ou dans le cloud, ou des RDD existants.

Les applications Spark incluent une ou plusieurs tâches Spark. Les tâches exécutent des tâches dans des exécuteurs, et les exécuteurs s'exécutent dans des conteneurs YARN. Chaque exécuteur s'exécute dans un seul conteneur et les exécuteurs existent tout au long de la vie d'une application. Un exécuteur est corrigé après le démarrage de l'application et YARN ne redimensionne pas le conteneur déjà alloué. Un exécuteur peut exécuter des tâches simultanément sur des données en mémoire.

Présentation des solutions NetApp Spark

NetApp dispose de trois portefeuilles de stockage : FAS/ AFF, E-Series et Cloud Volumes ONTAP. Nous avons validé AFF et la série E avec le système de stockage ONTAP pour les solutions Hadoop avec Apache Spark.

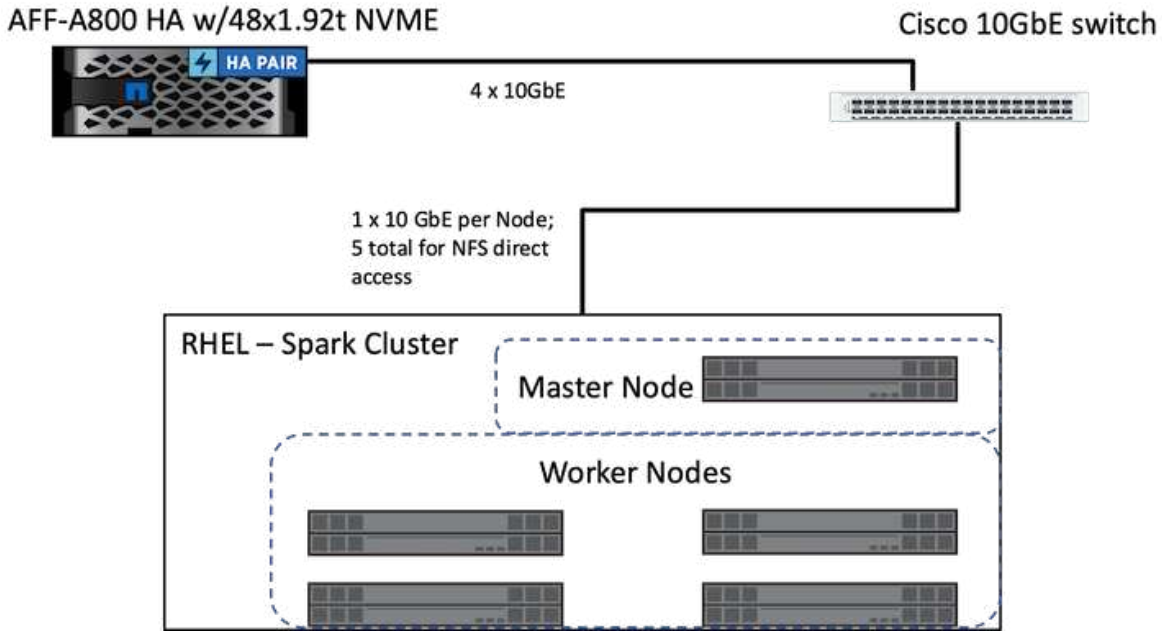
La structure de données optimisée par NetApp intègre des services et des applications de gestion des données (blocs de construction) pour l'accès, le contrôle, la protection et la sécurité des données, comme illustré dans la figure ci-dessous.



Les éléments constitutifs de la figure ci-dessus comprennent :

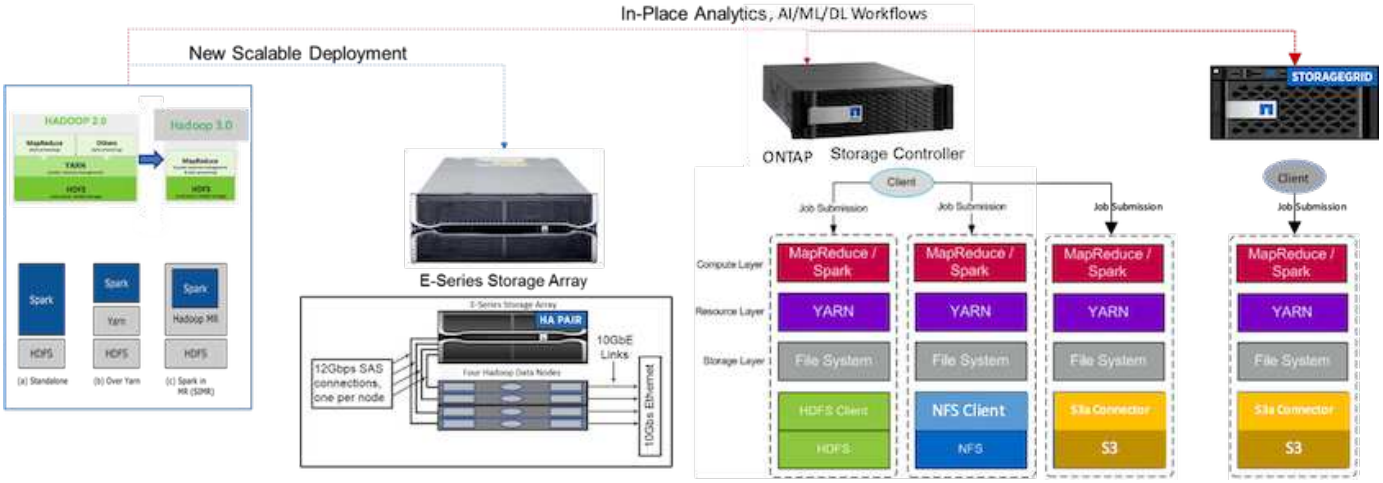
- * Accès direct NetApp NFS.* Fournit aux derniers clusters Hadoop et Spark un accès direct aux volumes NetApp NFS sans exigences de logiciel ou de pilote supplémentaires.
- * NetApp Cloud Volumes ONTAP et Google Cloud NetApp Volumes.* Stockage connecté défini par logiciel basé sur ONTAP exécuté dans Amazon Web Services (AWS) ou Azure NetApp Files (ANF) dans les services cloud Microsoft Azure.
- * Technologie NetApp SnapMirror .* Fournit des capacités de protection des données entre les instances locales et ONTAP Cloud ou NPS.
- **Fournisseurs de services cloud.** Ces fournisseurs incluent AWS, Microsoft Azure, Google Cloud et IBM Cloud.
- **PaaS.** Services d'analyse basés sur le cloud tels qu'Amazon Elastic MapReduce (EMR) et Databricks dans AWS ainsi que Microsoft Azure HDInsight et Azure Databricks.

La figure suivante illustre la solution Spark avec le stockage NetApp .

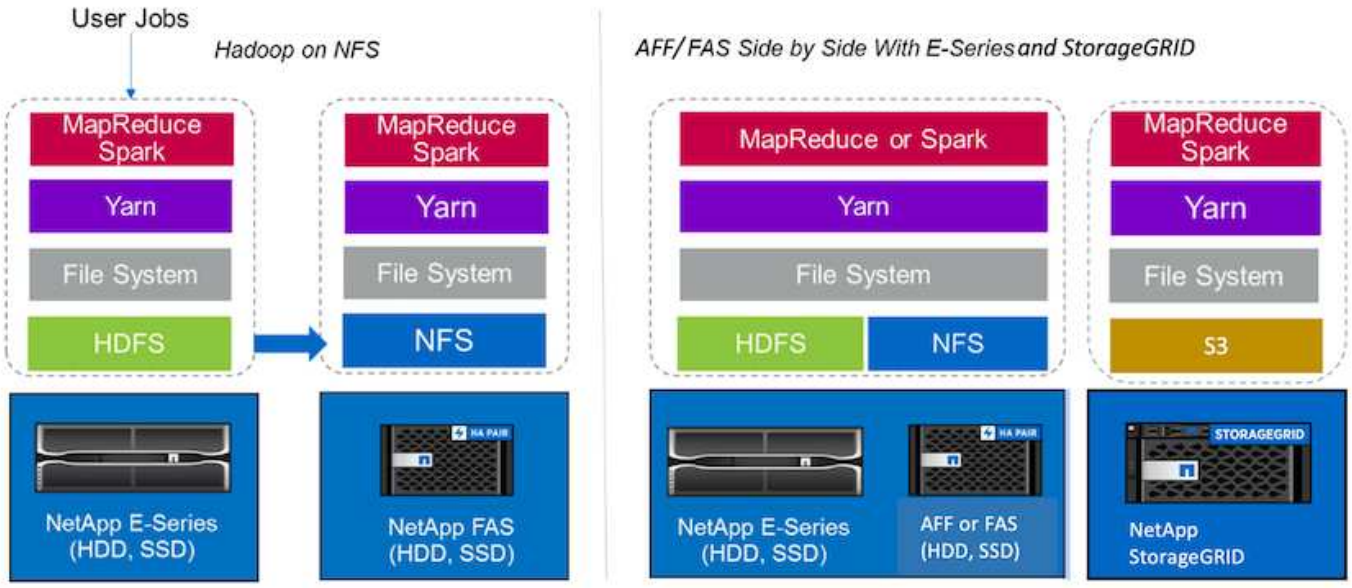


La solution ONTAP Spark utilise le protocole d'accès direct NetApp NFS pour les analyses sur place et les flux de travail IA, ML et DL utilisant l'accès aux données de production existantes. Les données de production disponibles pour les nœuds Hadoop sont exportées pour effectuer des tâches d'analyse et d'IA, de ML et de DL sur place. Vous pouvez accéder aux données à traiter dans les nœuds Hadoop avec ou sans accès direct NetApp NFS. Dans Spark avec le standalone ou `yarn` gestionnaire de cluster, vous pouvez configurer un volume NFS en utilisant `file://<target_volume>` . Nous avons validé trois cas d'utilisation avec différents ensembles de données. Les détails de ces validations sont présentés dans la section « Résultats des tests ». (xref)

La figure suivante illustre le positionnement du stockage NetApp Apache Spark/Hadoop.



Nous avons identifié les caractéristiques uniques de la solution E-Series Spark, de la solution AFF/ FAS ONTAP Spark et de la solution StorageGRID Spark, et avons effectué une validation et des tests détaillés. Sur la base de nos observations, NetApp recommande la solution E-Series pour les installations greenfield et les nouveaux déploiements évolutifs, ainsi que la solution AFF/ FAS pour les charges de travail d'analyse sur place, d'IA, de ML et de DL utilisant les données NFS existantes, et StorageGRID pour l'IA, le ML et le DL et les analyses de données modernes lorsque le stockage d'objets est requis.



Un lac de données est un référentiel de stockage pour de grands ensembles de données sous forme native qui peuvent être utilisés pour les tâches d'analyse, d'IA, de ML et de DL. Nous avons créé un référentiel de lac de données pour les solutions E-Series, AFF/ FAS et StorageGRID SG6060 Spark. Le système E-Series fournit un accès HDFS au cluster Hadoop Spark, tandis que les données de production existantes sont accessibles via le protocole d'accès direct NFS au cluster Hadoop. Pour les ensembles de données résidant dans le stockage d'objets, NetApp StorageGRID fournit un accès sécurisé S3 et S3a.

Résumé du cas d'utilisation

Cette page décrit les différents domaines dans lesquels cette solution peut être utilisée.

Données en streaming

Apache Spark peut traiter des données en streaming, qui sont utilisées pour les processus d'extraction, de transformation et de chargement (ETL) en streaming ; l'enrichissement des données ; la détection d'événements déclencheurs ; et l'analyse de sessions complexes :

- **Streaming ETL.** Les données sont continuellement nettoyées et agrégées avant d'être transférées dans les magasins de données. Netflix utilise Kafka et Spark Streaming pour créer une solution de recommandation de films en ligne et de surveillance des données en temps réel capable de traiter des milliards d'événements par jour à partir de différentes sources de données. L'ETL traditionnel pour le traitement par lots est toutefois traité différemment. Ces données sont d'abord lues, puis converties dans un format de base de données avant d'être écrites dans la base de données.
- **Enrichissement des données.** Le streaming Spark enrichit les données en direct avec des données statiques pour permettre une analyse des données en temps réel. Par exemple, les annonceurs en ligne peuvent diffuser des publicités personnalisées et ciblées en fonction des informations sur le comportement des clients.
- **Détection d'événement déclencheur.** Le streaming Spark vous permet de détecter et de réagir rapidement à un comportement inhabituel qui pourrait indiquer des problèmes potentiellement graves. Par exemple, les institutions financières utilisent des déclencheurs pour détecter et arrêter les transactions frauduleuses, et les hôpitaux utilisent des déclencheurs pour détecter les changements de santé dangereux détectés dans les signes vitaux d'un patient.
- **Analyse de session complexe.** Spark Streaming collecte des événements tels que l'activité de

l'utilisateur après la connexion à un site Web ou à une application, qui sont ensuite regroupés et analysés. Par exemple, Netflix utilise cette fonctionnalité pour fournir des recommandations de films en temps réel.

Pour plus de configuration de données en streaming, de vérification de Confluent Kafka et de tests de performances, consultez"TR-4912 : Recommandations de bonnes pratiques pour le stockage hiérarchisé Confluent Kafka avec NetApp" .

Apprentissage automatique

Le framework intégré Spark vous aide à exécuter des requêtes répétées sur des ensembles de données à l'aide de la bibliothèque d'apprentissage automatique (MLlib). MLlib est utilisé dans des domaines tels que le clustering, la classification et la réduction de dimensionnalité pour certaines fonctions courantes du Big Data telles que l'intelligence prédictive, la segmentation des clients à des fins de marketing et l'analyse des sentiments. MLlib est utilisé dans la sécurité des réseaux pour effectuer des inspections en temps réel des paquets de données à la recherche d'indications d'activité malveillante. Il aide les fournisseurs de sécurité à se renseigner sur les nouvelles menaces et à garder une longueur d'avance sur les pirates tout en protégeant leurs clients en temps réel.

Apprentissage profond

TensorFlow est un framework d'apprentissage profond populaire utilisé dans l'ensemble du secteur. TensorFlow prend en charge la formation distribuée sur un cluster CPU ou GPU. Cette formation distribuée permet aux utilisateurs de l'exécuter sur une grande quantité de données avec de nombreuses couches profondes.

Jusqu'à récemment, si nous voulions utiliser TensorFlow avec Apache Spark, nous devions effectuer tous les ETL nécessaires pour TensorFlow dans PySpark, puis écrire les données dans le stockage intermédiaire. Ces données seraient ensuite chargées sur le cluster TensorFlow pour le processus de formation réel. Ce flux de travail nécessitait que l'utilisateur maintienne deux clusters différents, un pour l'ETL et un pour la formation distribuée de TensorFlow. L'exécution et la maintenance de plusieurs clusters étaient généralement fastidieuses et prenaient beaucoup de temps.

Les DataFrames et RDD des versions antérieures de Spark n'étaient pas bien adaptés à l'apprentissage en profondeur car l'accès aléatoire était limité. Dans Spark 3.0 avec le projet Hydrogen, la prise en charge native des frameworks d'apprentissage en profondeur est ajoutée. Cette approche permet une planification non basée sur MapReduce sur le cluster Spark.

Analyse interactive

Apache Spark est suffisamment rapide pour effectuer des requêtes exploratoires sans échantillonnage avec des langages de développement autres que Spark, notamment SQL, R et Python. Spark utilise des outils de visualisation pour traiter des données complexes et les visualiser de manière interactive. Spark avec streaming structuré exécute des requêtes interactives sur des données en direct dans les analyses Web qui vous permettent d'exécuter des requêtes interactives sur la session actuelle d'un visiteur Web.

Système de recommandation

Au fil des ans, les systèmes de recommandation ont apporté d'énormes changements à nos vies, car les entreprises et les consommateurs ont réagi aux changements spectaculaires dans les achats en ligne, le divertissement en ligne et de nombreux autres secteurs. En effet, ces systèmes comptent parmi les réussites les plus évidentes de l'IA en production. Dans de nombreux cas d'utilisation pratiques, les systèmes de recommandation sont combinés à une IA conversationnelle ou à des chatbots interfacés avec un backend NLP pour obtenir des informations pertinentes et produire des inférences utiles.

Aujourd'hui, de nombreux détaillants adoptent de nouveaux modèles commerciaux tels que l'achat en ligne et le retrait en magasin, le retrait en bordure de rue, le paiement en libre-service, le scan-and-go, etc. Ces modèles sont devenus importants pendant la pandémie de COVID-19 en rendant les achats plus sûrs et plus pratiques pour les consommateurs. L'IA est essentielle à ces tendances numériques croissantes, qui sont influencées par le comportement des consommateurs et vice versa. Pour répondre aux demandes croissantes des consommateurs, améliorer l'expérience client, améliorer l'efficacité opérationnelle et augmenter les revenus, NetApp aide ses clients et entreprises à utiliser des algorithmes d'apprentissage automatique et d'apprentissage profond pour concevoir des systèmes de recommandation plus rapides et plus précis.

Il existe plusieurs techniques populaires utilisées pour fournir des recommandations, notamment le filtrage collaboratif, les systèmes basés sur le contenu, le modèle de recommandation d'apprentissage profond (DLRM) et les techniques hybrides. Les clients utilisaient auparavant PySpark pour mettre en œuvre un filtrage collaboratif afin de créer des systèmes de recommandation. Spark MLlib implémente les moindres carrés alternés (ALS) pour le filtrage collaboratif, un algorithme très populaire parmi les entreprises avant l'essor du DLRM.

Traitement du langage naturel

L'IA conversationnelle, rendue possible par le traitement du langage naturel (TALN), est la branche de l'IA qui aide les ordinateurs à communiquer avec les humains. La PNL est répandue dans tous les secteurs d'activité et dans de nombreux cas d'utilisation, des assistants intelligents et des chatbots à la recherche Google et au texte prédictif. Selon un "Gartner" Selon les prévisions, d'ici 2022, 70 % des personnes interagiront quotidiennement avec des plateformes d'IA conversationnelles. Pour une conversation de haute qualité entre un humain et une machine, les réponses doivent être rapides, intelligentes et naturelles.

Les clients ont besoin d'une grande quantité de données pour traiter et former leurs modèles de PNL et de reconnaissance automatique de la parole (ASR). Ils doivent également déplacer des données à travers la périphérie, le cœur et le cloud, et ils ont besoin de la puissance nécessaire pour effectuer des inférences en quelques millisecondes afin d'établir une communication naturelle avec les humains. NetApp AI et Apache Spark constituent une combinaison idéale pour le calcul, le stockage, le traitement des données, la formation des modèles, le réglage fin et le déploiement.

L'analyse des sentiments est un domaine d'étude de la PNL dans lequel des sentiments positifs, négatifs ou neutres sont extraits du texte. L'analyse des sentiments a une variété de cas d'utilisation, allant de la détermination des performances des employés du centre d'assistance dans les conversations avec les appelants à la fourniture de réponses de chatbot automatisées appropriées. Il a également été utilisé pour prédire le cours de l'action d'une entreprise en fonction des interactions entre les représentants de l'entreprise et le public lors des conférences téléphoniques trimestrielles sur les résultats. De plus, l'analyse des sentiments peut être utilisée pour déterminer l'opinion d'un client sur les produits, les services ou l'assistance fournis par la marque.

Nous avons utilisé le "Spark PNL" bibliothèque de "Laboratoires John Snow" pour charger des pipelines pré-entraînés et des représentations d'encodeurs bidirectionnels à partir de modèles de transformateurs (BERT), y compris "sentiment des nouvelles financières" et "FinBERT" , effectuant la tokenisation, la reconnaissance d'entités nommées, la formation de modèles, l'ajustement et l'analyse des sentiments à grande échelle. Spark NLP est la seule bibliothèque NLP open source en production qui propose des transformateurs de pointe tels que BERT, ALBERT, ELECTRA, XLNet, DistilBERT, RoBERTa, DeBERTa, XLM-RoBERTa, Longformer, ELMO, Universal Sentence Encoder, Google T5, MarianMT et GPT2. La bibliothèque fonctionne non seulement en Python et R, mais également dans l'écosystème JVM (Java, Scala et Kotlin) à grande échelle en étendant Apache Spark de manière native.

Principaux cas d'utilisation et architectures de l'IA, du ML et du DL

Les principaux cas d'utilisation et méthodologies de l'IA, du ML et du DL peuvent être divisés dans les sections suivantes :

Pipelines Spark NLP et inférence distribuée TensorFlow

La liste suivante contient les bibliothèques NLP open source les plus populaires qui ont été adoptées par la communauté des sciences des données à différents niveaux de développement :

- "Boîte à outils en langage naturel (NLTK)" . La boîte à outils complète pour toutes les techniques PNL. Il est maintenu depuis le début des années 2000.
- "TextBlob" . Une API Python d'outils NLP facile à utiliser construite sur NLTK et Pattern.
- "Stanford Core PNL" . Services et packages NLP en Java développés par le Stanford NLP Group.
- "Gensim" . Topic Modelling for Humans a commencé comme une collection de scripts Python pour le projet de bibliothèque numérique de mathématiques tchèque.
- "SpaCy" . Workflows NLP industriels de bout en bout avec Python et Cython avec accélération GPU pour les transformateurs.
- "Texte rapide" . Une bibliothèque NLP gratuite, légère et open source pour l'apprentissage des intégrations de mots et la classification des phrases créée par le laboratoire de recherche en IA (FAIR) de Facebook.

Spark NLP est une solution unique et unifiée pour toutes les tâches et exigences NLP qui permet de disposer de logiciels NLP évolutifs, hautes performances et haute précision pour des cas d'utilisation de production réels. Il s'appuie sur l'apprentissage par transfert et met en œuvre les algorithmes et modèles de pointe les plus récents dans la recherche et dans tous les secteurs. En raison du manque de support complet par Spark pour les bibliothèques ci-dessus, Spark NLP a été construit sur "Spark ML" pour tirer parti du moteur de traitement de données distribué en mémoire à usage général de Spark en tant que bibliothèque NLP de niveau entreprise pour les flux de travail de production critiques. Ses annotateurs utilisent des algorithmes basés sur des règles, l'apprentissage automatique et TensorFlow pour alimenter les implémentations d'apprentissage en profondeur. Cela couvre les tâches courantes de PNL, y compris, mais sans s'y limiter, la tokenisation, la lemmatisation, la dérivation, l'étiquetage des parties du discours, la reconnaissance d'entités nommées, la vérification orthographique et l'analyse des sentiments.

Les représentations d'encodeurs bidirectionnels à partir de transformateurs (BERT) sont une technique d'apprentissage automatique basée sur des transformateurs pour le PNL. Il a popularisé le concept de pré-entraînement et de réglage fin. L'architecture du transformateur dans BERT est issue de la traduction automatique, qui modélise mieux les dépendances à long terme que les modèles de langage basés sur le réseau neuronal récurrent (RNN). Il a également introduit la tâche de modélisation du langage masqué (MLM), où 15 % aléatoires de tous les jetons sont masqués et le modèle les prédit, permettant une véritable bidirectionnalité.

L'analyse du sentiment financier est difficile en raison du langage spécialisé et du manque de données étiquetées dans ce domaine. FinBERT, un modèle de langage basé sur BERT pré-entraîné, a été adapté au domaine "Reuters TRC2" , un corpus financier, et affiné avec des données étiquetées ("Banque de phrases financières") pour la classification du sentiment financier. Les chercheurs ont extrait 4 500 phrases d'articles de presse contenant des termes financiers. Ensuite, 16 experts et étudiants en master ayant une formation en finance ont étiqueté les phrases comme positives, neutres et négatives. Nous avons créé un flux de travail Spark de bout en bout pour analyser le sentiment des 10 premières sociétés du NASDAQ dans leurs transcriptions d'appels sur les résultats de 2016 à 2020, en utilisant FinBERT et deux autres pipelines pré-entraînés. "Expliquer le document DL") de Spark NLP.

Le moteur d'apprentissage profond sous-jacent de Spark NLP est TensorFlow, une plate-forme open source de bout en bout pour l'apprentissage automatique qui permet une création de modèles facile, une production ML robuste n'importe où et une expérimentation puissante pour la recherche. Par conséquent, lors de l'exécution de nos pipelines dans Spark `yarn cluster` En mode, nous exécutions essentiellement TensorFlow distribué avec parallélisation des données et des modèles sur un maître et plusieurs nœuds de travail, ainsi qu'un stockage en réseau monté sur le cluster.

Formation distribuée Horovod

La validation Hadoop principale pour les performances liées à MapReduce est effectuée avec TeraGen, TeraSort, TeraValidate et DFSIO (lecture et écriture). Les résultats de validation de TeraGen et TeraSort sont présentés dans "Solution NetApp E-Series pour Hadoop" et dans la section « Storage Tiering » pour AFF.

Sur la base des demandes des clients, nous considérons la formation distribuée avec Spark comme l'un des cas d'utilisation les plus importants. Dans ce document, nous avons utilisé le "Hovorod sur Spark" pour valider les performances de Spark avec les solutions NetApp sur site, cloud natives et cloud hybrides à l'aide des contrôleurs de stockage NetApp All Flash FAS (AFF), Azure NetApp Files et StorageGRID.

Le package Horovod sur Spark fournit un wrapper pratique autour de Horovod qui simplifie l'exécution de charges de travail de formation distribuées dans les clusters Spark, permettant une boucle de conception de modèle étroite dans laquelle le traitement des données, la formation du modèle et l'évaluation du modèle sont tous effectués dans Spark où résident les données de formation et d'inférence.

Il existe deux API pour exécuter Horovod sur Spark : une API Estimator de haut niveau et une API Run de niveau inférieur. Bien que les deux utilisent le même mécanisme sous-jacent pour lancer Horovod sur les exécuteurs Spark, l'API Estimator fait abstraction du traitement des données, de la boucle de formation du modèle, du point de contrôle du modèle, de la collecte de métriques et de la formation distribuée. Nous avons utilisé Horovod Spark Estimators, TensorFlow et Keras pour une préparation de données de bout en bout et un flux de travail de formation distribué basé sur le "Ventes en magasin Kaggle Rossmann" concours.

Le scénario `keras_spark_horovod_rossmann_estimator.py` peut être trouvé dans la section"Scripts Python pour chaque cas d'utilisation majeur." Il contient trois parties :

- La première partie effectue diverses étapes de prétraitement des données sur un ensemble initial de fichiers CSV fournis par Kaggle et collectés par la communauté. Les données d'entrée sont séparées dans un ensemble d'apprentissage avec un `Validation` sous-ensemble et un ensemble de données de test.
- La deuxième partie définit un modèle Keras Deep Neural Network (DNN) avec une fonction d'activation sigmoïde logarithmique et un optimiseur Adam, et effectue une formation distribuée du modèle à l'aide d'Horovod sur Spark.
- La troisième partie effectue une prédiction sur l'ensemble de données de test en utilisant le meilleur modèle qui minimise l'erreur absolue moyenne globale de l'ensemble de validation. Il crée ensuite un fichier CSV de sortie.

Voir la section"Apprentissage automatique" pour divers résultats de comparaison d'exécution.

Apprentissage profond multi-travailleurs utilisant Keras pour la prédiction du CTR

Avec les avancées récentes des plateformes et applications ML, une grande attention est désormais portée à l'apprentissage à grande échelle. Le taux de clics (CTR) est défini comme le nombre moyen de clics pour cent impressions d'annonces en ligne (exprimé en pourcentage). Il est largement adopté comme indicateur clé dans divers secteurs d'activité et cas d'utilisation, notamment le marketing numérique, la vente au détail, le commerce électronique et les fournisseurs de services. Pour plus de détails sur les applications du CTR et les résultats de performance de formation distribuée, consultez le"Modèles d'apprentissage profond pour les

performances de prédiction du CTR" section.

Dans ce rapport technique, nous avons utilisé une variante de la "Ensemble de données Criteo Terabyte Click Logs" (voir TR-4904) pour l'apprentissage en profondeur distribué multi-travailleurs utilisant Keras pour créer un flux de travail Spark avec des modèles Deep and Cross Network (DCN), en comparant ses performances en termes de fonction d'erreur de perte de journal avec un modèle de régression logistique Spark ML de base. DCN capture efficacement les interactions de caractéristiques efficaces de degrés limités, apprend les interactions hautement non linéaires, ne nécessite aucune ingénierie de caractéristiques manuelle ni recherche exhaustive et présente un faible coût de calcul.

Les données des systèmes de recommandation à l'échelle du Web sont pour la plupart discrètes et catégoriques, ce qui conduit à un espace de fonctionnalités vaste et clairsemé qui rend difficile l'exploration des fonctionnalités. Cela a limité la plupart des systèmes à grande échelle à des modèles linéaires tels que la régression logistique. Cependant, identifier fréquemment des caractéristiques prédictives et explorer simultanément des caractéristiques croisées invisibles ou rares est la clé pour faire de bonnes prédictions. Les modèles linéaires sont simples, interprétables et faciles à mettre à l'échelle, mais leur pouvoir d'expression est limité.

Les caractéristiques croisées, en revanche, se sont avérées significatives pour améliorer l'expressivité des modèles. Malheureusement, il faut souvent procéder à une ingénierie manuelle des fonctionnalités ou à une recherche exhaustive pour identifier ces fonctionnalités. Il est souvent difficile de généraliser aux interactions entre fonctionnalités invisibles. L'utilisation d'un réseau neuronal croisé comme DCN évite l'ingénierie des fonctionnalités spécifiques à la tâche en appliquant explicitement le croisement des fonctionnalités de manière automatique. Le réseau croisé est constitué de plusieurs couches, où le degré d'interaction le plus élevé est déterminé de manière prouvable par la profondeur de la couche. Chaque couche produit des interactions d'ordre supérieur basées sur celles existantes et conserve les interactions des couches précédentes.

Un réseau neuronal profond (DNN) promet de capturer des interactions très complexes entre les fonctionnalités. Cependant, comparé au DCN, il nécessite près d'un ordre de grandeur de paramètres supplémentaires, est incapable de former explicitement des fonctionnalités croisées et peut ne pas réussir à apprendre efficacement certains types d'interactions de fonctionnalités. Le réseau croisé est efficace en termes de mémoire et facile à mettre en œuvre. L'entraînement conjoint des composants croisés et DNN capture efficacement les interactions des fonctionnalités prédictives et offre des performances de pointe sur l'ensemble de données Criteo CTR.

Un modèle DCN commence par une couche d'intégration et d'empilement, suivie d'un réseau croisé et d'un réseau profond en parallèle. Celles-ci sont à leur tour suivies d'une couche de combinaison finale qui combine les sorties des deux réseaux. Vos données d'entrée peuvent être un vecteur avec des caractéristiques clairsemées et denses. Dans Spark, les bibliothèques contiennent le type `SparseVector` . Il est donc important pour les utilisateurs de faire la distinction entre les deux et d'être attentifs lorsqu'ils appellent leurs fonctions et méthodes respectives. Dans les systèmes de recommandation à l'échelle du Web tels que la prédiction du CTR, les entrées sont principalement des caractéristiques catégorielles, par exemple `'country=usa'` . Ces caractéristiques sont souvent codées sous forme de vecteurs one-hot, par exemple, `'[0,1,0, …]'` . Encodage à chaud (OHE) avec `SparseVector` est utile lorsqu'il s'agit de traiter des ensembles de données du monde réel avec des vocabulaires en constante évolution et en croissance. Nous avons modifié des exemples dans "DeepCTR" pour traiter de grands vocabulaires, en créant des vecteurs d'intégration dans la couche d'intégration et d'empilement de notre DCN.

Le "Ensemble de données Criteo Display Ads" prédit le taux de clics des publicités. Il comporte 13 caractéristiques entières et 26 caractéristiques catégorielles dans lesquelles chaque catégorie a une cardinalité élevée. Pour cet ensemble de données, une amélioration de 0,001 de la perte logarithmique est pratiquement significative en raison de la grande taille d'entrée. Une petite amélioration de la précision des prédictions pour une large base d'utilisateurs peut potentiellement conduire à une augmentation importante des revenus d'une entreprise. L'ensemble de données contient 11 Go de journaux d'utilisateurs sur une période de 7 jours, ce qui équivaut à environ 41 millions d'enregistrements. Nous avons utilisé Spark

`dataFrame.randomSplit()function` pour diviser aléatoirement les données pour la formation (80 %), la validation croisée (10 %) et les 10 % restants pour les tests.

DCN a été implémenté sur TensorFlow avec Keras. La mise en œuvre du processus de formation du modèle avec DCN comporte quatre composants principaux :

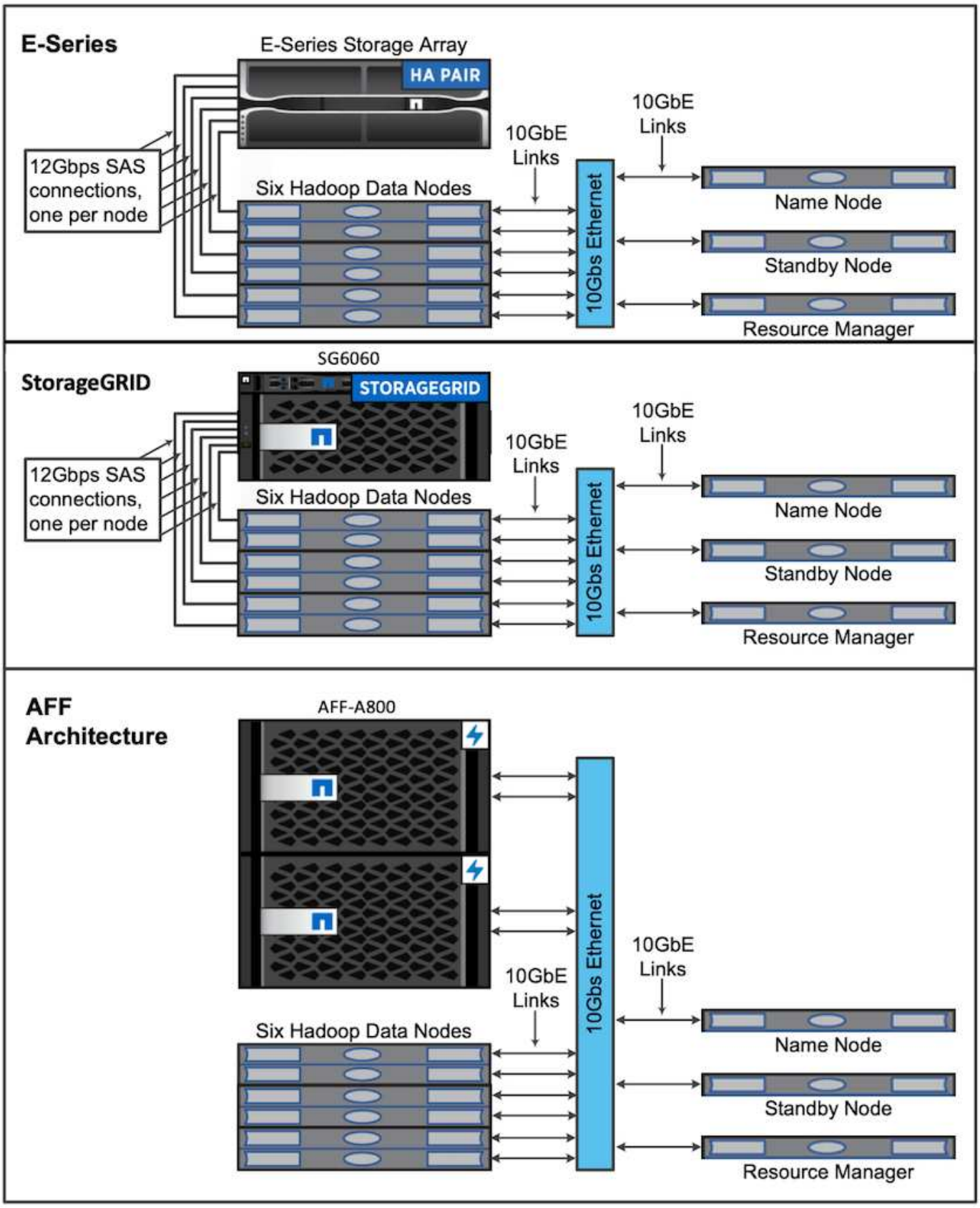
- **Traitement et intégration des données.** Les fonctionnalités à valeur réelle sont normalisées en appliquant une transformation logarithmique. Pour les fonctionnalités catégorielles, nous intégrons les fonctionnalités dans des vecteurs denses de dimension 6×(cardinalité de catégorie)1/4. La concaténation de tous les plongements donne un vecteur de dimension 1026.
- **Optimisation.** Nous avons appliqué l'optimisation stochastique par mini-lots avec l'optimiseur Adam. La taille du lot a été définie sur 512. La normalisation par lots a été appliquée au réseau profond et la norme de clip de gradient a été fixée à 100.
- **Régularisation.** Nous avons utilisé l'arrêt précoce, car la régularisation ou l'abandon de L2 ne s'est pas avéré efficace.
- **Hyperparamètres.** Nous rapportons les résultats basés sur une recherche de grille sur le nombre de couches cachées, la taille de la couche cachée, le taux d'apprentissage initial et le nombre de couches croisées. Le nombre de couches cachées variait de 2 à 5, avec des tailles de couches cachées allant de 32 à 1024. Pour le DCN, le nombre de couches croisées était de 1 à 6. Le taux d'apprentissage initial a été réglé de 0,0001 à 0,001 avec des incréments de 0,0001. Toutes les expériences ont appliqué un arrêt précoce à l'étape d'entraînement 150 000, au-delà de laquelle un surapprentissage a commencé à se produire.

En plus du DCN, nous avons également testé d'autres modèles d'apprentissage profond populaires pour la prédiction du CTR, notamment "DeepFM" , "AutoInt" , et "DCN v2" .

Architectures utilisées pour la validation

Pour cette validation, nous avons utilisé quatre nœuds de travail et un nœud maître avec une paire AFF-A800 HA. Tous les membres du cluster étaient connectés via des commutateurs réseau 10 GbE.

Pour cette validation de solution NetApp Spark, nous avons utilisé trois contrôleurs de stockage différents : le E5760, le E5724 et le AFF-A800. Les contrôleurs de stockage de la série E étaient connectés à cinq nœuds de données avec des connexions SAS 12 Gbit/s. Le contrôleur de stockage AFF HA-pair fournit des volumes NFS exportés via des connexions 10 GbE aux nœuds de travail Hadoop. Les membres du cluster Hadoop étaient connectés via des connexions 10 GbE dans les solutions Hadoop E-Series, AFF et StorageGRID .



Résultats des tests

Nous avons utilisé les scripts TeraSort et TeraValidate dans l'outil d'analyse comparative

TeraGen pour mesurer la validation des performances de Spark avec les configurations E5760, E5724 et AFF-A800. De plus, trois cas d'utilisation majeurs ont été testés : les pipelines Spark NLP et la formation distribuée TensorFlow, la formation distribuée Horovod et l'apprentissage profond multi-travailleurs utilisant Keras pour la prédiction CTR avec DeepFM.

Pour la validation des séries E et StorageGRID , nous avons utilisé le facteur de réplication Hadoop 2. Pour la validation AFF , nous n'avons utilisé qu'une seule source de données.

Le tableau suivant répertorie la configuration matérielle pour la validation des performances de Spark.

Type	Nœuds de travail Hadoop	Type de lecteur	Lecteurs par nœud	Contrôleur de stockage
SG6060	4	SAS	12	Paire unique à haute disponibilité (HA)
E5760	4	SAS	60	Paire HA unique
E5724	4	SAS	24	Paire HA unique
AFF800	4	SSD	6	Paire HA unique

Le tableau suivant répertorie les exigences logicielles.

Logiciels	Version
RHEL	7,9
Environnement d'exécution OpenJDK	1.8.0
Machine virtuelle serveur OpenJDK 64 bits	25,302
Git	2.24.1
GCC/G++	11.2.1
Étincelle	3.2.1
PySpark	3.1.2
SparkNLP	3.4.2
TensorFlow	2.9.0
Keras	2.9.0
Horovod	0.24.3

Analyse du sentiment financier

Nous avons publié"TR-4910 : Analyse des sentiments à partir des communications clients avec NetApp AI" , dans lequel un pipeline d'IA conversationnelle de bout en bout a été construit en utilisant le "Boîte à outils NetApp DataOps" , Stockage AFF et système NVIDIA DGX. Le pipeline effectue le traitement du signal audio par lots, la reconnaissance automatique de la parole (ASR), l'apprentissage par transfert et l'analyse des sentiments en exploitant la boîte à outils DataOps, "Kit de développement logiciel NVIDIA Riva" , et le "Cadre Tao" . En étendant le cas d'utilisation de l'analyse des sentiments au secteur des services financiers, nous avons créé un flux de travail SparkNLP, chargé trois modèles BERT pour diverses tâches NLP, telles que la reconnaissance d'entités nommées, et obtenu un sentiment au niveau des phrases pour les appels de

résultats trimestriels des 10 premières entreprises du NASDAQ.

Le script suivant `sentiment_analysis_spark. py` utilise le modèle FinBERT pour traiter les transcriptions dans HDFS et produire des décomptes de sentiments positifs, neutres et négatifs, comme indiqué dans le tableau suivant :

```
-bash-4.2$ time ~/anaconda3/bin/spark-submit
--packages com.johnsnowlabs.nlp:spark-nlp_2.12:3.4.3
--master yarn
--executor-memory 5g
--executor-cores 1
--num-executors 160
--conf spark.driver.extraJavaOptions="-Xss10m -XX:MaxPermSize=1024M"
--conf spark.executor.extraJavaOptions="-Xss10m -XX:MaxPermSize=512M"
/sparkusecase/tr-4570-nlp/sentiment_analysis_spark.py
hdfs:///data1/Transcripts/
> ./sentiment_analysis_hdfs.log 2>&1
real13m14.300s
user557m11.319s
sys4m47.676s
```

Le tableau suivant répertorie l'analyse des sentiments au niveau des phrases et des appels aux résultats pour les 10 premières sociétés du NASDAQ de 2016 à 2020.

Nombre et pourcentage de sentiments	**Les 10 entreprises**	**AAPL**	**DMLA**	**AMZN**	**CSCO**	**GOOGL**	**INTC**	**MSFT**	**NVDA**
Comptes positifs	7447	1567	743	290	682	826	824	904	417
Comptes neutres	64067	6856	7596	5086	6650	5914	6099	5715	6189
Comptes négatifs	1787	253	213	84	189	97	282	202	89
Comptes non classés	196	0	0	76	0	0	0	1	0
(nombre total de comptes)	73497	8676	8552	5536	7521	6837	7205	6822	6695

En termes de pourcentages, la plupart des phrases prononcées par les PDG et les directeurs financiers sont factuelles et véhiculent donc un sentiment neutre. Lors d'une conférence téléphonique sur les résultats, les analystes posent des questions qui peuvent transmettre un sentiment positif ou négatif. Il vaut la peine d'étudier plus en détail quantitativement la manière dont le sentiment négatif ou positif affecte les cours des

actions le jour même ou le jour suivant la négociation.

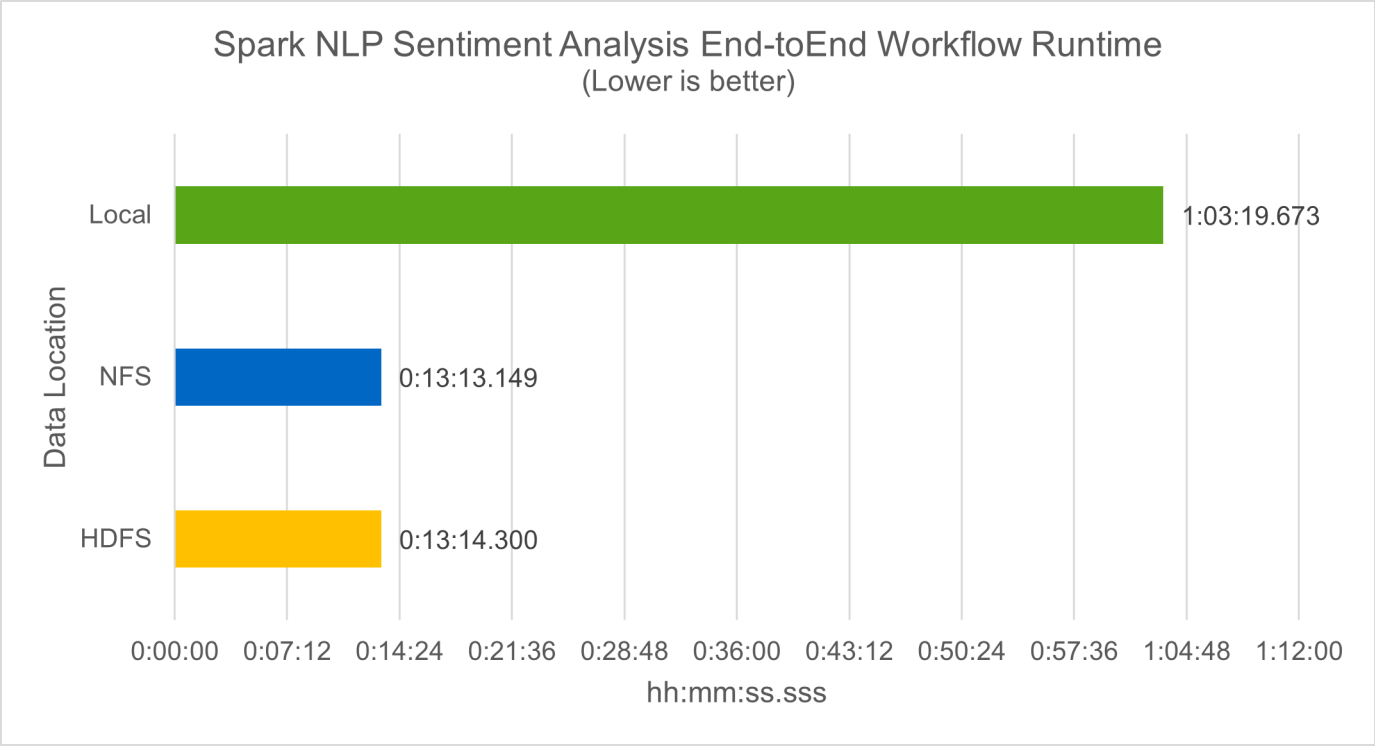
Le tableau suivant répertorie l'analyse des sentiments au niveau des phrases pour les 10 premières sociétés du NASDAQ, exprimée en pourcentage.

Pourcentage de sentiment	Les 10 entreprises	AAPL	DMLA	AMZN	CSCO	GOOGL	INTC	MSFT	NVDA
Positif	10,13%	18,06%	8,69%	5,24%	9,07%	12,08%	11,44%	13,25%	6,23%
Neutre	87,17%	79,02%	88,82%	91,87%	88,42%	86,50%	84,65%	83,77%	92,44%
Négatif	2,43%	2,92%	2,49%	1,52%	2,51%	1,42%	3,91%	2,96%	1,33%
Non classé	0,27%	0%	0%	1,37%	0%	0%	0%	0,01%	0%

En termes de temps d'exécution du flux de travail, nous avons constaté une amélioration significative de 4,78x par rapport à `local` mode vers un environnement distribué dans HDFS, et une amélioration supplémentaire de 0,14 % en exploitant NFS.

```
-bash-4.2$ time ~/anaconda3/bin/spark-submit
--packages com.johnsnowlabs.nlp:spark-nlp_2.12:3.4.3
--master yarn
--executor-memory 5g
--executor-cores 1
--num-executors 160
--conf spark.driver.extraJavaOptions="-Xss10m -XX:MaxPermSize=1024M"
--conf spark.executor.extraJavaOptions="-Xss10m -XX:MaxPermSize=512M"
/sparkusecase/tr-4570-nlp/sentiment_analysis_spark.py
file:///sparkdemo/sparknlp/Transcripts/
> ./sentiment_analysis_nfs.log 2>&1
real13m13.149s
user537m50.148s
sys4m46.173s
```

Comme le montre la figure suivante, le parallélisme des données et du modèle a amélioré le traitement des données et la vitesse d'inférence du modèle TensorFlow distribué. L'emplacement des données dans NFS a donné un temps d'exécution légèrement meilleur, car le goulot d'étranglement du flux de travail est le téléchargement de modèles pré-entraînés. Si nous augmentons la taille de l'ensemble de données de transcription, l'avantage de NFS est plus évident.



Formation distribuée avec performance Horovod

La commande suivante a produit des informations d'exécution et un fichier journal dans notre cluster Spark à l'aide d'un seul `master` nœud avec 160 exécuteurs chacun avec un cœur. La mémoire de l'exécuteur a été limitée à 5 Go pour éviter une erreur de mémoire insuffisante. Voir la section"Scripts Python pour chaque cas d'utilisation majeur" pour plus de détails concernant le traitement des données, la formation du modèle et le calcul de la précision du modèle dans `keras_spark_horovod_rossmann_estimator.py`.

```
(base) [root@n138 horovod]# time spark-submit
--master local
--executor-memory 5g
--executor-cores 1
--num-executors 160
/sparkusecase/horovod/keras_spark_horovod_rossmann_estimator.py
--epochs 10
--data-dir file:///sparkusecase/horovod
--local-submission-csv /tmp/submission_0.csv
--local-checkpoint-file /tmp/checkpoint/
> /tmp/keras_spark_horovod_rossmann_estimator_local. log 2>&1
```

Le temps d'exécution résultant avec dix époques d'entraînement était le suivant :

```
real43m34.608s
user12m22.057s
sys2m30.127s
```

Il a fallu plus de 43 minutes pour traiter les données d'entrée, former un modèle DNN, calculer la précision et produire des points de contrôle TensorFlow et un fichier CSV pour les résultats de prédiction. Nous avons limité le nombre d'époques d'entraînement à 10, qui dans la pratique est souvent fixé à 100 pour garantir une précision satisfaisante du modèle. Le temps de formation évolue généralement de manière linéaire avec le nombre d'époques.

Nous avons ensuite utilisé les quatre nœuds de travail disponibles dans le cluster et exécuté le même script dans `yarn` mode avec données dans HDFS :

```
(base) [root@n138 horovod]# time spark-submit
--master yarn
--executor-memory 5g
--executor-cores 1 --num-executors 160
/sparkusecase/horovod/keras_spark_horovod_rossmann_estimator.py
--epochs 10
--data-dir hdfs:///user/hdfs/tr-4570/experiments/horovod
--local-submission-csv /tmp/submission_1.csv
--local-checkpoint-file /tmp/checkpoint/
> /tmp/keras_spark_horovod_rossmann_estimator_yarn.log 2>&1
```

Le temps d'exécution résultant a été amélioré comme suit :

```
real8m13.728s
user7m48.421s
sys1m26.063s
```

Avec le modèle d'Horovod et le parallélisme des données dans Spark, nous avons constaté une accélération d'exécution de 5,29x `yarn` contre `local` mode avec dix époques d'entraînement. Ceci est illustré dans la figure suivante avec les légendes `HDFS` et `Local` . La formation du modèle DNN TensorFlow sous-jacent peut être encore accélérée avec des GPU s'ils sont disponibles. Nous prévoyons de réaliser ces tests et de publier les résultats dans un futur rapport technique.

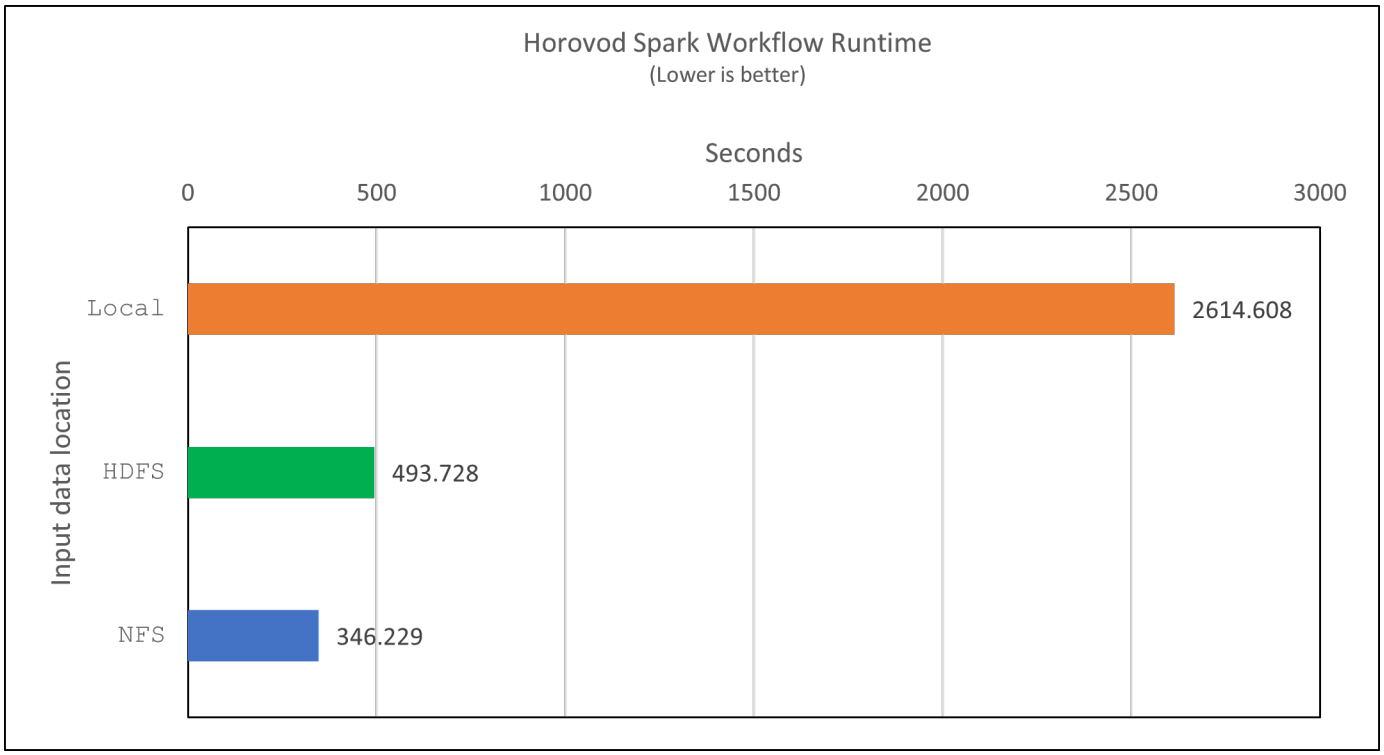
Notre prochain test a comparé les temps d'exécution avec des données d'entrée résidant dans NFS par rapport à HDFS. Le volume NFS sur l' AFF A800 a été monté sur `/sparkdemo/horovod` sur les cinq nœuds (un maître, quatre travailleurs) de notre cluster Spark. Nous avons exécuté une commande similaire à celle des tests précédents, avec le `--data- dir` paramètre pointant désormais vers le montage NFS :

```
(base) [root@n138 horovod]# time spark-submit
--master yarn
--executor-memory 5g
--executor-cores 1
--num-executors 160
/sparkusecase/horovod/keras_spark_horovod_rossmann_estimator.py
--epochs 10
--data-dir file:///sparkdemo/horovod
--local-submission-csv /tmp/submission_2.csv
--local-checkpoint-file /tmp/checkpoint/
> /tmp/keras_spark_horovod_rossmann_estimator_nfs.log 2>&1
```

Le temps d'exécution résultant avec NFS était le suivant :

```
real 5m46.229s
user 5m35.693s
sys  1m5.615s
```

Il y a eu une accélération supplémentaire de 1,43x, comme le montre la figure suivante. Par conséquent, avec un stockage entièrement flash NetApp connecté à leur cluster, les clients bénéficient des avantages d'un transfert et d'une distribution rapides des données pour les flux de travail Horovod Spark, atteignant une accélération de 7,55 fois par rapport à l'exécution sur un seul nœud.



Modèles d'apprentissage profond pour les performances de prédiction du CTR

Pour les systèmes de recommandation conçus pour maximiser le CTR, vous devez apprendre les interactions sophistiquées des fonctionnalités derrière les comportements des utilisateurs qui peuvent être calculées mathématiquement d'un ordre faible à un ordre élevé. Les interactions entre les caractéristiques d'ordre faible et d'ordre élevé doivent être tout aussi importantes pour un bon modèle d'apprentissage en profondeur, sans biaiser vers l'une ou l'autre. Deep Factorization Machine (DeepFM), un réseau neuronal basé sur une machine de factorisation, combine des machines de factorisation pour la recommandation et l'apprentissage en profondeur pour l'apprentissage des fonctionnalités dans une nouvelle architecture de réseau neuronal.

Bien que les machines de factorisation conventionnelles modélisent les interactions de caractéristiques par paires comme un produit interne de vecteurs latents entre les caractéristiques et puissent théoriquement capturer des informations d'ordre élevé, dans la pratique, les praticiens de l'apprentissage automatique n'utilisent généralement que des interactions de caractéristiques de second ordre en raison de la complexité élevée du calcul et du stockage. Variantes de réseaux neuronaux profonds comme celui de Google "Modèles larges et profonds" d'autre part, apprend les interactions de fonctionnalités sophistiquées dans une structure de réseau hybride en combinant un modèle linéaire large et un modèle profond.

Ce modèle large et profond comporte deux entrées, l'une pour le modèle large sous-jacent et l'autre pour le modèle profond, cette dernière partie nécessitant encore une ingénierie des fonctionnalités experte et rendant ainsi la technique moins généralisable à d'autres domaines. Contrairement au modèle large et profond, DeepFM peut être efficacement formé avec des fonctionnalités brutes sans aucune ingénierie de fonctionnalités, car sa partie large et sa partie profonde partagent la même entrée et le même vecteur d'intégration.

Nous avons d'abord traité le Criteo `train.txt` (11 Go) dans un fichier CSV nommé `ctr_train.csv` stocké dans un montage NFS `/sparkdemo/tr-4570-data` en utilisant `run_classification_criteo_spark.py` de la section"Scripts Python pour chaque cas d'utilisation majeur." Dans ce script, la fonction `process_input_file` exécute plusieurs méthodes de chaîne pour supprimer les tabulations et insérer `','` comme délimiteur et `'\n'` comme nouvelle ligne. Notez que vous n'avez besoin de traiter que l'original `train.txt` une fois, afin que le bloc de code soit affiché sous forme de commentaires.

Pour les tests suivants de différents modèles DL, nous avons utilisé `ctr_train.csv` comme fichier d'entrée. Lors des tests ultérieurs, le fichier CSV d'entrée a été lu dans un Spark DataFrame avec un schéma contenant un champ de `'label'` , caractéristiques denses en nombres entiers `['I1', 'I2', 'I3', …, 'I13']` , et des fonctionnalités éparses `['C1', 'C2', 'C3', …, 'C26']` . Ce qui suit `spark-submit` la commande prend un fichier CSV d'entrée, entraîne les modèles DeepFM avec une répartition de 20 % pour la validation croisée et sélectionne le meilleur modèle après dix époques d'entraînement pour calculer la précision de prédiction sur l'ensemble de test :

```
(base) [root@n138 ~]# time spark-submit --master yarn --executor-memory 5g
--executor-cores 1 --num-executors 160
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py --data
-dir file:///sparkdemo/tr-4570-data >
/tmp/run_classification_criteo_spark_local.log 2>&1
```

Notez que puisque le fichier de données `ctr_train.csv` est supérieur à 11 Go, vous devez définir une valeur suffisante `spark.driver.maxResultSize` supérieur à la taille de l'ensemble de données pour éviter les erreurs.

```
 spark = SparkSession.builder \
    .master("yarn") \
    .appName("deep_ctr_classification") \
    .config("spark.jars.packages", "io.github.ravwojdyla:spark-schema-
utils_2.12:0.1.0") \
    .config("spark.executor.cores", "1") \
    .config('spark.executor.memory', '5gb') \
    .config('spark.executor.memoryOverhead', '1500') \
    .config('spark.driver.memoryOverhead', '1500') \
    .config("spark.sql.shuffle.partitions", "480") \
    .config("spark.sql.execution.arrow.enabled", "true") \
    .config("spark.driver.maxResultSize", "50gb") \
    .getOrCreate()
```

Dans ce qui précède `SparkSession.builder` configuration que nous avons également activée "Apache Arrow" , qui convertit un Spark DataFrame en Pandas DataFrame avec le `df.toPandas()` méthode.

```
22/06/17 15:56:21 INFO scheduler.DAGScheduler: Job 2 finished: toPandas at
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py:96, took
627.126487 s
Obtained Spark DF and transformed to Pandas DF using Arrow.
```

Après une division aléatoire, il y a plus de 36 millions de lignes dans l'ensemble de données d'entraînement et 9 millions d'échantillons dans l'ensemble de test :

```
Training dataset size =  36672493
Testing dataset size =  9168124
```

Étant donné que ce rapport technique se concentre sur les tests de processeur sans utiliser de GPU, il est impératif de créer TensorFlow avec les indicateurs de compilateur appropriés. Cette étape évite d'appeler des bibliothèques accélérées par GPU et tire pleinement parti des instructions AVX (Advanced Vector Extensions) et AVX2 de TensorFlow. Ces fonctionnalités sont conçues pour les calculs algébriques linéaires tels que l'addition vectorisée, les multiplications matricielles dans un entraînement DNN à propagation directe ou à rétropropagation. L'instruction Fused Multiply Add (FMA) disponible avec AVX2 utilisant des registres à virgule flottante (FP) 256 bits est idéale pour le code entier et les types de données, ce qui permet une accélération jusqu'à 2x. Pour le code FP et les types de données, AVX2 atteint une accélération de 8 % par rapport à AVX.

```
2022-06-18 07:19:20.101478: I
tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow binary
is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the
following CPU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the
appropriate compiler flags.
```

Pour créer TensorFlow à partir de la source, NetApp recommande d'utiliser "Bazel" . Pour notre environnement, nous avons exécuté les commandes suivantes dans l'invite du shell pour installer `dnf` , `dnf-plugins` , et Bazel.

```
yum install dnf
dnf install 'dnf-command(copr)'
dnf copr enable vbatts/bazel
dnf install bazel5
```

Vous devez activer GCC 5 ou une version plus récente pour utiliser les fonctionnalités C++17 pendant le processus de génération, qui est fourni par RHEL avec Software Collections Library (SCL). Les commandes suivantes installent `devtoolset` et GCC 11.2.1 sur notre cluster RHEL 7.9 :

```
subscription-manager repos --enable rhel-server-rhscl-7-rpms
yum install devtoolset-11-toolchain
yum install devtoolset-11-gcc-c++
yum update
scl enable devtoolset-11 bash
. /opt/rh/devtoolset-11/enable
```

Notez que les deux dernières commandes permettent `devtoolset-11` , qui utilise `/opt/rh/devtoolset-11/root/usr/bin/gcc` (CCG 11.2.1). Assurez-vous également que votre `git` la version est supérieure à 1.8.3 (celle-ci est fournie avec RHEL 7.9). Se référer à ceci "article" pour la mise à jour `git` à 2.24.1.

Nous supposons que vous avez déjà cloné le dernier référentiel maître TensorFlow. Créez ensuite un `workspace` répertoire avec un `WORKSPACE` fichier pour créer TensorFlow à partir de la source avec AVX, AVX2 et FMA. Exécutez le `configure` fichier et spécifiez l'emplacement binaire Python correct. "CUDA" est désactivé pour nos tests car nous n'avons pas utilisé de GPU. UN `.bazelrc` le fichier est généré en fonction de vos paramètres. De plus, nous avons édité le fichier et défini `build --define=no_hdfs_support=false` pour activer la prise en charge HDFS. Se référer à `.bazelrc` dans la section"Scripts Python pour chaque cas d'utilisation majeur," pour une liste complète des paramètres et des indicateurs.

```
./configure
bazel build -c opt --copt=-mavx --copt=-mavx2 --copt=-mfma --copt=
-mfpmath=both -k //tensorflow/tools/pip_package:build_pip_package
```

Après avoir créé TensorFlow avec les indicateurs appropriés, exécutez le script suivant pour traiter l'ensemble de données Criteo Display Ads, entraîner un modèle DeepFM et calculer l'aire sous la courbe caractéristique de fonctionnement du récepteur (ROC AUC) à partir des scores de prédiction.

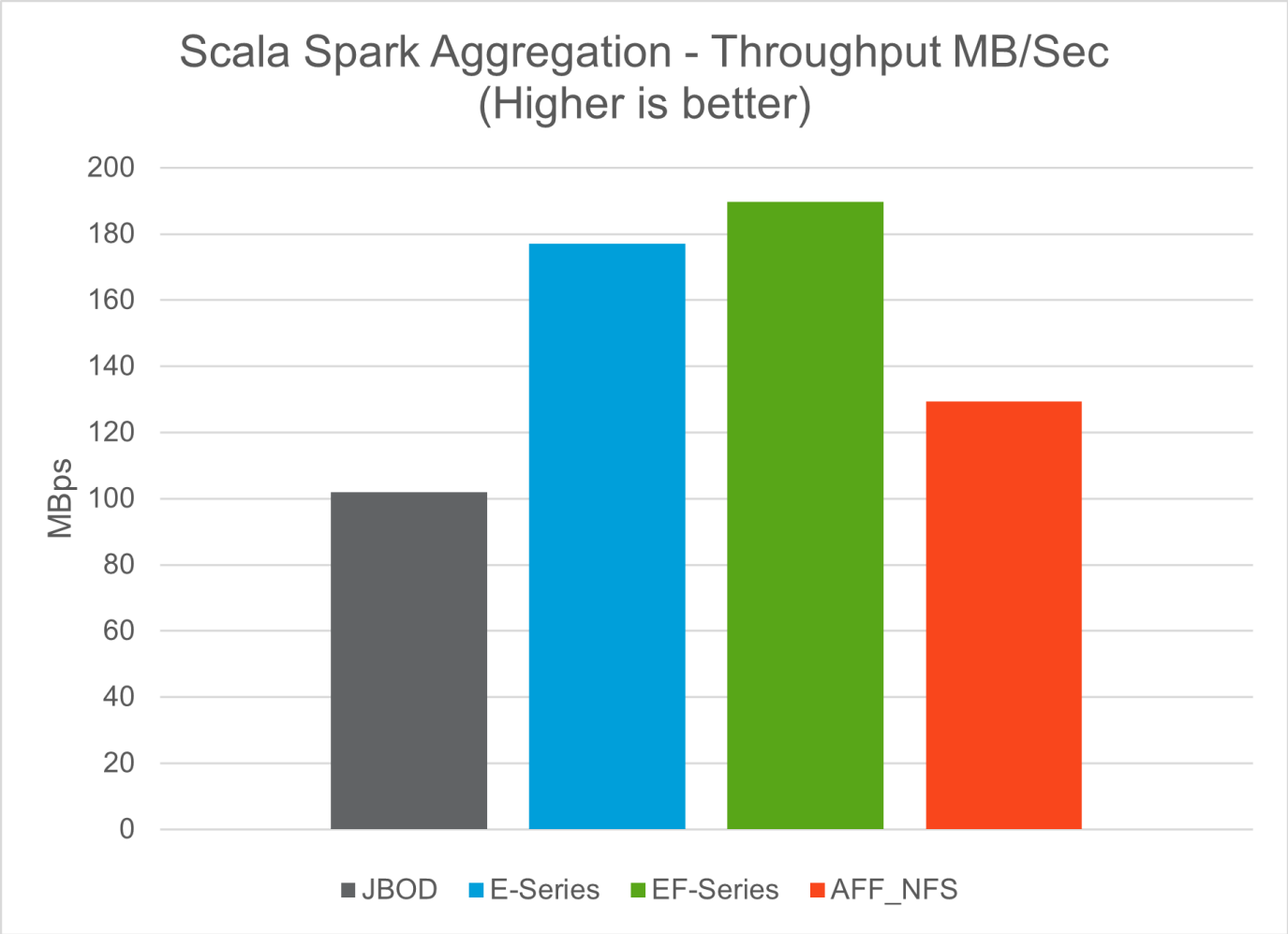
```
(base) [root@n138 examples]# ~/anaconda3/bin/spark-submit
--master yarn
--executor-memory 15g
--executor-cores 1
--num-executors 160
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py
--data-dir file:///sparkdemo/tr-4570-data
> . /run_classification_criteo_spark_nfs.log 2>&1
```

Après dix périodes d'entraînement, nous avons obtenu le score AUC sur l'ensemble de données de test :

```
Epoch 1/10
125/125 - 7s - loss: 0.4976 - binary_crossentropy: 0.4974 - val_loss:
0.4629 - val_binary_crossentropy: 0.4624
Epoch 2/10
125/125 - 1s - loss: 0.3281 - binary_crossentropy: 0.3271 - val_loss:
0.5146 - val_binary_crossentropy: 0.5130
Epoch 3/10
125/125 - 1s - loss: 0.1948 - binary_crossentropy: 0.1928 - val_loss:
0.6166 - val_binary_crossentropy: 0.6144
Epoch 4/10
125/125 - 1s - loss: 0.1408 - binary_crossentropy: 0.1383 - val_loss:
0.7261 - val_binary_crossentropy: 0.7235
Epoch 5/10
125/125 - 1s - loss: 0.1129 - binary_crossentropy: 0.1102 - val_loss:
0.7961 - val_binary_crossentropy: 0.7934
Epoch 6/10
125/125 - 1s - loss: 0.0949 - binary_crossentropy: 0.0921 - val_loss:
0.9502 - val_binary_crossentropy: 0.9474
Epoch 7/10
125/125 - 1s - loss: 0.0778 - binary_crossentropy: 0.0750 - val_loss:
1.1329 - val_binary_crossentropy: 1.1301
Epoch 8/10
125/125 - 1s - loss: 0.0651 - binary_crossentropy: 0.0622 - val_loss:
1.3794 - val_binary_crossentropy: 1.3766
Epoch 9/10
125/125 - 1s - loss: 0.0555 - binary_crossentropy: 0.0527 - val_loss:
1.6115 - val_binary_crossentropy: 1.6087
Epoch 10/10
125/125 - 1s - loss: 0.0470 - binary_crossentropy: 0.0442 - val_loss:
1.6768 - val_binary_crossentropy: 1.6740
test AUC 0.6337
```

D'une manière similaire aux cas d'utilisation précédents, nous avons comparé l'exécution du workflow Spark

avec des données résidant dans différents emplacements. La figure suivante montre une comparaison de la prédiction CTR d'apprentissage profond pour un environnement d'exécution de workflows Spark.



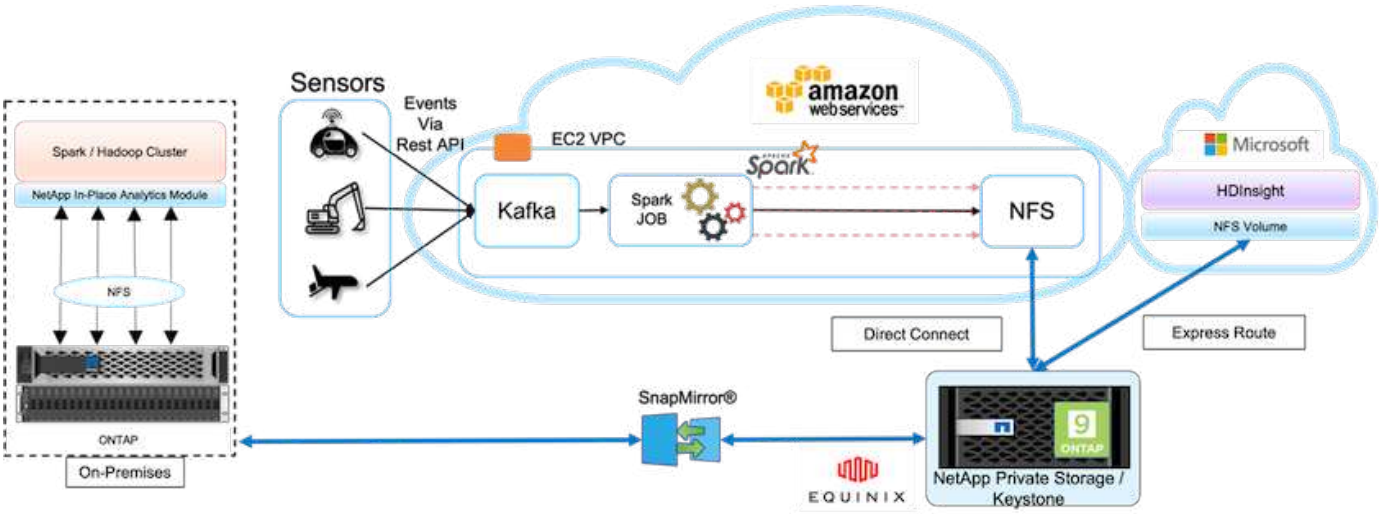
Solution de cloud hybride

Un centre de données d'entreprise moderne est un cloud hybride qui connecte plusieurs environnements d'infrastructure distribués via un plan de gestion des données continu avec un modèle d'exploitation cohérent, sur site et/ou dans plusieurs clouds publics. Pour tirer le meilleur parti d'un cloud hybride, vous devez être en mesure de déplacer de manière transparente des données entre vos environnements sur site et multicloud sans avoir besoin de conversions de données ou de refactorisation d'applications.

Les clients ont indiqué qu'ils commençaient leur parcours vers le cloud hybride soit en déplaçant le stockage secondaire vers le cloud pour des cas d'utilisation tels que la protection des données, soit en déplaçant des charges de travail moins critiques pour l'entreprise, telles que le développement d'applications et DevOps vers le cloud. Ils passent ensuite à des charges de travail plus critiques. L'hébergement Web et de contenu, le développement DevOps et d'applications, les bases de données, les analyses et les applications conteneurisées font partie des charges de travail de cloud hybride les plus populaires. La complexité, le coût et les risques des projets d'IA d'entreprise ont historiquement entravé l'adoption de l'IA du stade expérimental à la production.

Avec une solution de cloud hybride NetApp , les clients bénéficient d'outils intégrés de sécurité, de

gouvernance des données et de conformité avec un panneau de contrôle unique pour la gestion des données et des flux de travail dans des environnements distribués, tout en optimisant le coût total de possession en fonction de leur consommation. La figure suivante est un exemple de solution d'un partenaire de service cloud chargé de fournir une connectivité multicloud pour les données d'analyse de Big Data des clients.



Dans ce scénario, les données IoT reçues dans AWS à partir de différentes sources sont stockées dans un emplacement central dans NetApp Private Storage (NPS). Le stockage NPS est connecté aux clusters Spark ou Hadoop situés dans AWS et Azure, permettant aux applications d'analyse de Big Data de s'exécuter dans plusieurs clouds et d'accéder aux mêmes données. Les principales exigences et défis pour ce cas d'utilisation sont les suivants :

- Les clients souhaitent exécuter des tâches d'analyse sur les mêmes données à l'aide de plusieurs clouds.
- Les données doivent être reçues de différentes sources telles que des environnements sur site et dans le cloud via différents capteurs et hubs.
- La solution doit être efficace et rentable.
- Le principal défi est de créer une solution rentable et efficace qui fournit des services d'analyse hybrides entre différents environnements sur site et cloud.

Notre solution de protection des données et de connectivité multicloud résout le problème des applications d'analyse cloud sur plusieurs hyperscalers. Comme le montre la figure ci-dessus, les données des capteurs sont diffusées et ingérées dans le cluster AWS Spark via Kafka. Les données sont stockées dans un partage NFS résidant dans NPS, qui est situé en dehors du fournisseur de cloud dans un centre de données Equinix.

Étant donné que NetApp NPS est connecté à Amazon AWS et Microsoft Azure via des connexions Direct Connect et Express Route respectivement, les clients peuvent exploiter le module d'analyse sur place pour accéder aux données des clusters d'analyse Amazon et AWS. Par conséquent, étant donné que le stockage sur site et le stockage NPS exécutent tous deux le logiciel ONTAP , "SnapMirror" peut refléter les données NPS dans le cluster sur site, fournissant des analyses de cloud hybride sur site et sur plusieurs clouds.

Pour des performances optimales, NetApp recommande généralement d'utiliser plusieurs interfaces réseau et une connexion directe ou des routes express pour accéder aux données à partir d'instances cloud. Nous avons d'autres solutions de transfert de données, notamment "XCP" et "Copie et synchronisation BlueXP" pour aider les clients à créer des clusters Spark hybrides cloud compatibles avec les applications, sécurisés et rentables.

Scripts Python pour chaque cas d'utilisation majeur

Les trois scripts Python suivants correspondent aux trois principaux cas d'utilisation testés. Le premier est `sentiment_analysis_sparknlp.py` .

```
# TR-4570 Refresh NLP testing by Rick Huang
from sys import argv
import os
import sparknlp
import pyspark.sql.functions as F
from sparknlp import Finisher
from pyspark.ml import Pipeline
from sparknlp.base import *
from sparknlp.annotator import *
from sparknlp.pretrained import PretrainedPipeline
from sparknlp import Finisher
# Start Spark Session with Spark NLP
spark = sparknlp.start()
print("Spark NLP version:")
print(sparknlp.version())
print("Apache Spark version:")
print(spark.version)
spark = sparknlp.SparkSession.builder \
     .master("yarn") \
     .appName("test_hdfs_read_write") \
     .config("spark.executor.cores", "1") \
     .config("spark.jars.packages", "com.johnsnowlabs.nlp:spark-
nlp_2.12:3.4.3")\
     .config('spark.executor.memory', '5gb') \
     .config('spark.executor.memoryOverhead','1000')\
     .config('spark.driver.memoryOverhead','1000')\
     .config("spark.sql.shuffle.partitions", "480")\
     .getOrCreate()
sc = spark.sparkContext
from pyspark.sql import SQLContext
sql = SQLContext(sc)
sqlContext = SQLContext(sc)
# Download pre-trained pipelines & sequence classifier
explain_pipeline_model = PretrainedPipeline('explain_document_dl',
lang='en').model#pipeline_sa =
PretrainedPipeline("classifierdl_bertwiki_finance_sentiment_pipeline",
lang="en")
# pipeline_finbert =
BertForSequenceClassification.loadSavedModel('/sparkusecase/bert_sequence_
classifier_finbert_en_3', spark)
```

```
sequenceClassifier = BertForSequenceClassification \
        .pretrained('bert_sequence_classifier_finbert', 'en') \
        .setInputCols(['token', 'document']) \
        .setOutputCol('class') \
        .setCaseSensitive(True) \
        .setMaxSentenceLength(512)
def process_sentence_df(data):
    # Pre-process: begin
    print("1. Begin DataFrame pre-processing...\n")
    print(f"\n\t2. Attaching DocumentAssembler Transformer to the
pipeline")
    documentAssembler = DocumentAssembler() \
        .setInputCol("text") \
        .setOutputCol("document") \
        .setCleanupMode("inplace_full")
        #.setCleanupMode("shrink", "inplace_full")
    doc_df = documentAssembler.transform(data)
    doc_df.printSchema()
    doc_df.show(truncate=50)
    # Pre-process: get rid of  blank lines
    clean_df = doc_df.withColumn("tmp", F.explode("document")) \
        .select("tmp.result").where("tmp.end !=
-1").withColumnRenamed("result", "text").dropna()
    print("[OK!] DataFrame after initial cleanup:\n")
    clean_df.printSchema()
    clean_df.show(truncate=80)
    # for FinBERT
    tokenizer = Tokenizer() \
        .setInputCols(['document']) \
        .setOutputCol('token')
    print(f"\n\t3. Attaching Tokenizer Annotator to the pipeline")
    pipeline_finbert = Pipeline(stages=[
        documentAssembler,
        tokenizer,
        sequenceClassifier
        ])
    # Use Finisher() & construct PySpark ML pipeline
    finisher = Finisher().setInputCols(["token", "lemma", "pos",
"entities"])
    print(f"\n\t4. Attaching Finisher Transformer to the pipeline")
    pipeline_ex = Pipeline() \
        .setStages([
           explain_pipeline_model,
           finisher
           ])
    print("\n\t\t\t ---- Pipeline Built Successfully ----")
```

```
    # Loading pipelines to annotate
    #result_ex_df = pipeline_ex.transform(clean_df)
    ex_model = pipeline_ex.fit(clean_df)
    annotations_finished_ex_df = ex_model.transform(clean_df)
    # result_sa_df = pipeline_sa.transform(clean_df)
    result_finbert_df = pipeline_finbert.fit(clean_df).transform(clean_df)
    print("\n\t\t\t ----Document Explain, Sentiment Analysis & FinBERT
Pipeline Fitted Successfully ----")
    # Check the result entities
    print("[OK!] Simple explain ML pipeline result:\n")
    annotations_finished_ex_df.printSchema()
    annotations_finished_ex_df.select('text',
'finished_entities').show(truncate=False)
    # Check the result sentiment from FinBERT
    print("[OK!] Sentiment Analysis FinBERT pipeline result:\n")
    result_finbert_df.printSchema()
    result_finbert_df.select('text', 'class.result').show(80, False)
    sentiment_stats(result_finbert_df)
    return
def sentiment_stats(finbert_df):
    result_df = finbert_df.select('text', 'class.result')
    sa_df = result_df.select('result')
    sa_df.groupBy('result').count().show()
    # total_lines = result_clean_df.count()
    # num_neutral = result_clean_df.where(result_clean_df.result ==
['neutral']).count()
    # num_positive = result_clean_df.where(result_clean_df.result ==
['positive']).count()
    # num_negative = result_clean_df.where(result_clean_df.result ==
['negative']).count()
    # print(f"\nRatio of neutral sentiment = {num_neutral/total_lines}")
    # print(f"Ratio of positive sentiment = {num_positive / total_lines}")
    # print(f"Ratio of negative sentiment = {num_negative /
total_lines}\n")
    return
def process_input_file(file_name):
    # Turn input file to Spark DataFrame
    print("START processing input file...")
    data_df = spark.read.text(file_name)
    data_df.show()
    # rename first column 'text' for sparknlp
    output_df = data_df.withColumnRenamed("value", "text").dropna()
    output_df.printSchema()
    return output_dfdef process_local_dir(directory):
    filelist = []
    for subdir, dirs, files in os.walk(directory):
```

```
        for filename in files:
            filepath = subdir + os.sep + filename
            print("[OK!] Will process the following files:")
            if filepath.endswith(".txt"):
                print(filepath)
                filelist.append(filepath)
    return filelist
def process_local_dir_or_file(dir_or_file):
    numfiles = 0
    if os.path.isfile(dir_or_file):
        input_df = process_input_file(dir_or_file)
        print("Obtained input_df.")
        process_sentence_df(input_df)
        print("Processed input_df")
        numfiles += 1
    else:
        filelist = process_local_dir(dir_or_file)
        for file in filelist:
            input_df = process_input_file(file)
            process_sentence_df(input_df)
            numfiles += 1
    return numfiles
def process_hdfs_dir(dir_name):
    # Turn input files to Spark DataFrame
    print("START processing input HDFS directory...")
    data_df = spark.read.option("recursiveFileLookup",
"true").text(dir_name)
    data_df.show()
    print("[DEBUG] total lines in data_df = ", data_df.count())
    # rename first column 'text' for sparknlp
    output_df = data_df.withColumnRenamed("value", "text").dropna()
    print("[DEBUG] output_df looks like: \n")
    output_df.show(40, False)
    print("[DEBUG] HDFS dir resulting data_df schema: \n")
    output_df.printSchema()
    process_sentence_df(output_df)
    print("Processed HDFS directory: ", dir_name)
    returnif __name__ == '__main__':
    try:
        if len(argv) == 2:
            print("Start processing input...\n")
    except:
        print("[ERROR] Please enter input text file or path to
process!\n")
        exit(1)
    # This is for local file, not hdfs:
```

```
    numfiles = process_local_dir_or_file(str(argv[1]))
    # For HDFS single file & directory:
    input_df = process_input_file(str(argv[1]))
    print("Obtained input_df.")
    process_sentence_df(input_df)
    print("Processed input_df")
    numfiles += 1
    # For HDFS directory of subdirectories of files:
    input_parse_list = str(argv[1]).split('/')
    print(input_parse_list)
    if input_parse_list[-2:-1] == ['Transcripts']:
        print("Start processing HDFS directory: ", str(argv[1]))
        process_hdfs_dir(str(argv[1]))
    print(f"[OK!] All done. Number of files processed = {numfiles}")
```

Le deuxième script est `keras_spark_horovod_rossmann_estimator.py`.

```
# Copyright 2022 NetApp, Inc.
# Authored by Rick Huang
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
==========================================================================
====
# The below code was modified from: https://www.kaggle.com/c/rossmann-
store-sales
import argparse
import datetime
import os
import sys
from distutils.version import LooseVersion
import pyspark.sql.types as T
import pyspark.sql.functions as F
from pyspark import SparkConf, Row
from pyspark.sql import SparkSession
```

```
import tensorflow as tf
import tensorflow.keras.backend as K
from tensorflow.keras.layers import Input, Embedding, Concatenate, Dense,
Flatten, Reshape, BatchNormalization, Dropout
import horovod.spark.keras as hvd
from horovod.spark.common.backend import SparkBackend
from horovod.spark.common.store import Store
from horovod.tensorflow.keras.callbacks import BestModelCheckpoint
parser = argparse.ArgumentParser(description='Horovod Keras Spark Rossmann
Estimator Example',

formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--master',
                    help='spark cluster to use for training. If set to
None, uses current default cluster. Cluster'
                         'should be set up to provide a Spark task per
multiple CPU cores, or per GPU, e.g. by'
                         'supplying `-c <NUM_GPUS>` in Spark Standalone
mode')
parser.add_argument('--num-proc', type=int,
                    help='number of worker processes for training,
default: `spark.default.parallelism`')
parser.add_argument('--learning_rate', type=float, default=0.0001,
                    help='initial learning rate')
parser.add_argument('--batch-size', type=int, default=100,
                    help='batch size')
parser.add_argument('--epochs', type=int, default=100,
                    help='number of epochs to train')
parser.add_argument('--sample-rate', type=float,
                    help='desired sampling rate. Useful to set to low
number (e.g. 0.01) to make sure that '
                         'end-to-end process works')
parser.add_argument('--data-dir', default='file://' + os.getcwd(),
                    help='location of data on local filesystem (prefixed
with file://) or on HDFS')
parser.add_argument('--local-submission-csv', default='submission.csv',
                    help='output submission predictions CSV')
parser.add_argument('--local-checkpoint-file', default='checkpoint',
                    help='model checkpoint')
parser.add_argument('--work-dir', default='/tmp',
                    help='temporary working directory to write
intermediate files (prefix with hdfs:// to use HDFS)')
if __name__ == '__main__':
    args = parser.parse_args()
    # =============== #
    # DATA PREPARATION #
```

```
    # ================ #
    print('================')
    print('Data preparation')
    print('================')
    # Create Spark session for data preparation.
    conf = SparkConf() \
        .setAppName('Keras Spark Rossmann Estimator Example') \
        .set('spark.sql.shuffle.partitions', '480') \
        .set("spark.executor.cores", "1") \
        .set('spark.executor.memory', '5gb') \
        .set('spark.executor.memoryOverhead','1000')\
        .set('spark.driver.memoryOverhead','1000')
    if args.master:
        conf.setMaster(args.master)
    elif args.num_proc:
        conf.setMaster('local[{}]'.format(args.num_proc))
    spark = SparkSession.builder.config(conf=conf).getOrCreate()
    train_csv = spark.read.csv('%s/train.csv' % args.data_dir,
header=True)
    test_csv = spark.read.csv('%s/test.csv' % args.data_dir, header=True)
    store_csv = spark.read.csv('%s/store.csv' % args.data_dir,
header=True)
    store_states_csv = spark.read.csv('%s/store_states.csv' %
args.data_dir, header=True)
    state_names_csv = spark.read.csv('%s/state_names.csv' % args.data_dir,
header=True)
    google_trend_csv = spark.read.csv('%s/googletrend.csv' %
args.data_dir, header=True)
    weather_csv = spark.read.csv('%s/weather.csv' % args.data_dir,
header=True)
    def expand_date(df):
        df = df.withColumn('Date', df.Date.cast(T.DateType()))
        return df \
            .withColumn('Year', F.year(df.Date)) \
            .withColumn('Month', F.month(df.Date)) \
            .withColumn('Week', F.weekofyear(df.Date)) \
            .withColumn('Day', F.dayofmonth(df.Date))
    def prepare_google_trend():
        # Extract week start date and state.
        google_trend_all = google_trend_csv \
            .withColumn('Date', F.regexp_extract(google_trend_csv.week,
'(.*?) -', 1)) \
            .withColumn('State', F.regexp_extract(google_trend_csv.file,
'Rossmann_DE_(.*)', 1))
        # Map state NI -> HB,NI to align with other data sources.
        google_trend_all = google_trend_all \
```

```
            .withColumn('State', F.when(google_trend_all.State == 'NI',
'HB,NI').otherwise(google_trend_all.State))
        # Expand dates.
        return expand_date(google_trend_all)
    def add_elapsed(df, cols):
        def add_elapsed_column(col, asc):
            def fn(rows):
                last_store, last_date = None, None
                for r in rows:
                    if last_store != r.Store:
                        last_store = r.Store
                        last_date = r.Date
                    if r[col]:
                        last_date = r.Date
                    fields = r.asDict().copy()
                    fields[('After' if asc else 'Before') + col] = (r.Date
- last_date).days
                    yield Row(**fields)
            return fn
        df = df.repartition(df.Store)
        for asc in [False, True]:
            sort_col = df.Date.asc() if asc else df.Date.desc()
            rdd = df.sortWithinPartitions(df.Store.asc(), sort_col).rdd
            for col in cols:
                rdd = rdd.mapPartitions(add_elapsed_column(col, asc))
            df = rdd.toDF()
        return df
    def prepare_df(df):
        num_rows = df.count()
        # Expand dates.
        df = expand_date(df)
        df = df \
            .withColumn('Open', df.Open != '0') \
            .withColumn('Promo', df.Promo != '0') \
            .withColumn('StateHoliday', df.StateHoliday != '0') \
            .withColumn('SchoolHoliday', df.SchoolHoliday != '0')
        # Merge in store information.
        store = store_csv.join(store_states_csv, 'Store')
        df = df.join(store, 'Store')
        # Merge in Google Trend information.
        google_trend_all = prepare_google_trend()
        df = df.join(google_trend_all, ['State', 'Year',
'Week']).select(df['*'], google_trend_all.trend)
        # Merge in Google Trend for whole Germany.
        google_trend_de = google_trend_all[google_trend_all.file ==
'Rossmann_DE'].withColumnRenamed('trend', 'trend_de')
```

```
        df = df.join(google_trend_de, ['Year', 'Week']).select(df['*'],
google_trend_de.trend_de)
        # Merge in weather.
        weather = weather_csv.join(state_names_csv, weather_csv.file ==
state_names_csv.StateName)
        df = df.join(weather, ['State', 'Date'])
        # Fix null values.
        df = df \
            .withColumn('CompetitionOpenSinceYear',
F.coalesce(df.CompetitionOpenSinceYear, F.lit(1900))) \
            .withColumn('CompetitionOpenSinceMonth',
F.coalesce(df.CompetitionOpenSinceMonth, F.lit(1))) \
            .withColumn('Promo2SinceYear', F.coalesce(df.Promo2SinceYear,
F.lit(1900))) \
            .withColumn('Promo2SinceWeek', F.coalesce(df.Promo2SinceWeek,
F.lit(1)))
        # Days & months competition was open, cap to 2 years.
        df = df.withColumn('CompetitionOpenSince',
                           F.to_date(F.format_string('%s-%s-15',
df.CompetitionOpenSinceYear,

df.CompetitionOpenSinceMonth)))
        df = df.withColumn('CompetitionDaysOpen',
                           F.when(df.CompetitionOpenSinceYear > 1900,
                                  F.greatest(F.lit(0), F.least(F.lit(360 *
2), F.datediff(df.Date, df.CompetitionOpenSince))))
                           .otherwise(0))
        df = df.withColumn('CompetitionMonthsOpen',
(df.CompetitionDaysOpen / 30).cast(T.IntegerType()))
        # Days & weeks of promotion, cap to 25 weeks.
        df = df.withColumn('Promo2Since',
                           F.expr('date_add(format_string("%s-01-01",
Promo2SinceYear), (cast(Promo2SinceWeek as int) - 1) * 7)'))
        df = df.withColumn('Promo2Days',
                           F.when(df.Promo2SinceYear > 1900,
                                  F.greatest(F.lit(0), F.least(F.lit(25 *
7), F.datediff(df.Date, df.Promo2Since))))
                           .otherwise(0))
        df = df.withColumn('Promo2Weeks', (df.Promo2Days /
7).cast(T.IntegerType()))
        # Check that we did not lose any rows through inner joins.
        assert num_rows == df.count(), 'lost rows in joins'
        return df
    def build_vocabulary(df, cols):
        vocab = {}
        for col in cols:
```

```
            values = [r[0] for r in df.select(col).distinct().collect()]
            col_type = type([x for x in values if x is not None][0])
            default_value = col_type()
            vocab[col] = sorted(values, key=lambda x: x or default_value)
        return vocab
    def cast_columns(df, cols):
        for col in cols:
            df = df.withColumn(col,
F.coalesce(df[col].cast(T.FloatType()), F.lit(0.0)))
        return df
    def lookup_columns(df, vocab):
        def lookup(mapping):
            def fn(v):
                return mapping.index(v)
            return F.udf(fn, returnType=T.IntegerType())
        for col, mapping in vocab.items():
            df = df.withColumn(col, lookup(mapping)(df[col]))
        return df
    if args.sample_rate:
        train_csv = train_csv.sample(withReplacement=False,
fraction=args.sample_rate)
        test_csv = test_csv.sample(withReplacement=False,
fraction=args.sample_rate)
    # Prepare data frames from CSV files.
    train_df = prepare_df(train_csv).cache()
    test_df = prepare_df(test_csv).cache()
    # Add elapsed times from holidays & promos, the data spanning training
& test datasets.
    elapsed_cols = ['Promo', 'StateHoliday', 'SchoolHoliday']
    elapsed = add_elapsed(train_df.select('Date', 'Store', *elapsed_cols)
                          .unionAll(test_df.select('Date', 'Store',
*elapsed_cols)),
                          elapsed_cols)
    # Join with elapsed times.
    train_df = train_df \
        .join(elapsed, ['Date', 'Store']) \
        .select(train_df['*'], *[prefix + col for prefix in ['Before',
'After'] for col in elapsed_cols])
    test_df = test_df \
        .join(elapsed, ['Date', 'Store']) \
        .select(test_df['*'], *[prefix + col for prefix in ['Before',
'After'] for col in elapsed_cols])
    # Filter out zero sales.
    train_df = train_df.filter(train_df.Sales > 0)
    print('==================')
    print('Prepared data frame')
```

```
    print('==================')
    train_df.show()
    categorical_cols = [
        'Store', 'State', 'DayOfWeek', 'Year', 'Month', 'Day', 'Week',
'CompetitionMonthsOpen', 'Promo2Weeks', 'StoreType',
        'Assortment', 'PromoInterval', 'CompetitionOpenSinceYear',
'Promo2SinceYear', 'Events', 'Promo',
        'StateHoliday', 'SchoolHoliday'
    ]
    continuous_cols = [
        'CompetitionDistance', 'Max_TemperatureC', 'Mean_TemperatureC',
'Min_TemperatureC', 'Max_Humidity',
        'Mean_Humidity', 'Min_Humidity', 'Max_Wind_SpeedKm_h',
'Mean_Wind_SpeedKm_h', 'CloudCover', 'trend', 'trend_de',
        'BeforePromo', 'AfterPromo', 'AfterStateHoliday',
'BeforeStateHoliday', 'BeforeSchoolHoliday', 'AfterSchoolHoliday'
    ]
    all_cols = categorical_cols + continuous_cols
    # Select features.
    train_df = train_df.select(*(all_cols + ['Sales', 'Date'])).cache()
    test_df = test_df.select(*(all_cols + ['Id', 'Date'])).cache()
    # Build vocabulary of categorical columns.
    vocab = build_vocabulary(train_df.select(*categorical_cols)

.unionAll(test_df.select(*categorical_cols)).cache(),
                             categorical_cols)
    # Cast continuous columns to float & lookup categorical columns.
    train_df = cast_columns(train_df, continuous_cols + ['Sales'])
    train_df = lookup_columns(train_df, vocab)
    test_df = cast_columns(test_df, continuous_cols)
    test_df = lookup_columns(test_df, vocab)
    # Split into training & validation.
    # Test set is in 2015, use the same period in 2014 from the training
set as a validation set.
    test_min_date = test_df.agg(F.min(test_df.Date)).collect()[0][0]
    test_max_date = test_df.agg(F.max(test_df.Date)).collect()[0][0]
    one_year = datetime.timedelta(365)
    train_df = train_df.withColumn('Validation',
                                   (train_df.Date > test_min_date -
one_year) & (train_df.Date <= test_max_date - one_year))
    # Determine max Sales number.
    max_sales = train_df.agg(F.max(train_df.Sales)).collect()[0][0]
    # Convert Sales to log domain
    train_df = train_df.withColumn('Sales', F.log(train_df.Sales))
    print('==================================')
    print('Data frame with transformed columns')
```

```
    print('====================================')
    train_df.show()
    print('================')
    print('Data frame sizes')
    print('================')
    train_rows = train_df.filter(~train_df.Validation).count()
    val_rows = train_df.filter(train_df.Validation).count()
    test_rows = test_df.count()
    print('Training: %d' % train_rows)
    print('Validation: %d' % val_rows)
    print('Test: %d' % test_rows)
    # ============== #
    # MODEL TRAINING #
    # ============== #
    print('==============')
    print('Model training')
    print('==============')
    def exp_rmspe(y_true, y_pred):
        """Competition evaluation metric, expects logarithic inputs."""
        pct = tf.square((tf.exp(y_true) - tf.exp(y_pred)) /
tf.exp(y_true))
        # Compute mean excluding stores with zero denominator.
        x = tf.reduce_sum(tf.where(y_true > 0.001, pct,
tf.zeros_like(pct)))
        y = tf.reduce_sum(tf.where(y_true > 0.001, tf.ones_like(pct),
tf.zeros_like(pct)))
        return tf.sqrt(x / y)
    def act_sigmoid_scaled(x):
        """Sigmoid scaled to logarithm of maximum sales scaled by 20%."""
        return tf.nn.sigmoid(x) * tf.math.log(max_sales) * 1.2
    CUSTOM_OBJECTS = {'exp_rmspe': exp_rmspe,
                      'act_sigmoid_scaled': act_sigmoid_scaled}
    # Disable GPUs when building the model to prevent memory leaks
    if LooseVersion(tf.__version__) >= LooseVersion('2.0.0'):
        # See https://github.com/tensorflow/tensorflow/issues/33168
        os.environ['CUDA_VISIBLE_DEVICES'] = '-1'
    else:

K.set_session(tf.Session(config=tf.ConfigProto(device_count={'GPU': 0})))
    # Build the model.
    inputs = {col: Input(shape=(1,), name=col) for col in all_cols}
    embeddings = [Embedding(len(vocab[col]), 10, input_length=1,
name='emb_' + col)(inputs[col])
                  for col in categorical_cols]
    continuous_bn = Concatenate()([Reshape((1, 1), name='reshape_' +
col)(inputs[col])
```

```
                                    for col in continuous_cols])
    continuous_bn = BatchNormalization()(continuous_bn)
    x = Concatenate()(embeddings + [continuous_bn])
    x = Flatten()(x)
    x = Dense(1000, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
    x = Dense(1000, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
    x = Dense(1000, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
    x = Dense(500, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
    x = Dropout(0.5)(x)
    output = Dense(1, activation=act_sigmoid_scaled)(x)
    model = tf.keras.Model([inputs[f] for f in all_cols], output)
    model.summary()
    opt = tf.keras.optimizers.Adam(lr=args.learning_rate, epsilon=1e-3)
    # Checkpoint callback to specify options for the returned Keras model
    ckpt_callback = BestModelCheckpoint(monitor='val_loss', mode='auto',
save_freq='epoch')
    # Horovod: run training.
    store = Store.create(args.work_dir)
    backend = SparkBackend(num_proc=args.num_proc,
                           stdout=sys.stdout, stderr=sys.stderr,
                           prefix_output_with_timestamp=True)
    keras_estimator = hvd.KerasEstimator(backend=backend,
                                         store=store,
                                         model=model,
                                         optimizer=opt,
                                         loss='mae',
                                         metrics=[exp_rmspe],
                                         custom_objects=CUSTOM_OBJECTS,
                                         feature_cols=all_cols,
                                         label_cols=['Sales'],
                                         validation='Validation',
                                         batch_size=args.batch_size,
                                         epochs=args.epochs,
                                         verbose=2,

checkpoint_callback=ckpt_callback)
    keras_model =
keras_estimator.fit(train_df).setOutputCols(['Sales_output'])
    history = keras_model.getHistory()
    best_val_rmspe = min(history['val_exp_rmspe'])
    print('Best RMSPE: %f' % best_val_rmspe)
    # Save the trained model.
```

```
    keras_model.save(args.local_checkpoint_file)
    print('Written checkpoint to %s' % args.local_checkpoint_file)
    # ================ #
    # FINAL PREDICTION #
    # ================ #
    print('================')
    print('Final prediction')
    print('================')
    pred_df=keras_model.transform(test_df)
    pred_df.printSchema()
    pred_df.show(5)
    # Convert from log domain to real Sales numbers
    pred_df=pred_df.withColumn('Sales_pred', F.exp(pred_df.Sales_output))
    submission_df = pred_df.select(pred_df.Id.cast(T.IntegerType()),
pred_df.Sales_pred).toPandas()
    submission_df.sort_values(by=['Id']).to_csv(args.local_submission_csv,
index=False)
    print('Saved predictions to %s' % args.local_submission_csv)
    spark.stop()
```

Le troisième script est `run_classification_criteo_spark.py` .

```
import tempfile, string, random, os, uuid
import argparse, datetime, sys, shutil
import csv
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
from pyspark import SparkContext
from pyspark.sql import SparkSession, SQLContext, Row, DataFrame
from pyspark.mllib import linalg as mllib_linalg
from pyspark.mllib.linalg import SparseVector as mllibSparseVector
from pyspark.mllib.linalg import VectorUDT as mllibVectorUDT
from pyspark.mllib.linalg import Vector as mllibVector, Vectors as
mllibVectors
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.classification import LogisticRegressionWithSGD
from pyspark.ml import linalg as ml_linalg
from pyspark.ml.linalg import VectorUDT as mlVectorUDT
from pyspark.ml.linalg import SparseVector as mlSparseVector
from pyspark.ml.linalg import Vector as mlVector, Vectors as mlVectors
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import OneHotEncoder
from math import log
from math import exp  # exp(-t) = e^-t
```

```
from operator import add
from pyspark.sql.functions import udf, split, lit
from pyspark.sql.functions import size, sum as sqlsum
import pyspark.sql.functions as F
import pyspark.sql.types as T
from pyspark.sql.types import ArrayType, StructType, StructField,
LongType, StringType, IntegerType, FloatType
from pyspark.sql.functions import explode, col, log, when
from collections import defaultdict
import pandas as pd
import pyspark.pandas as ps
from sklearn.metrics import log_loss, roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from deepctr.models import DeepFM
from deepctr.feature_column import SparseFeat, DenseFeat,
get_feature_names
spark = SparkSession.builder \
    .master("yarn") \
    .appName("deep_ctr_classification") \
    .config("spark.jars.packages", "io.github.ravwojdyla:spark-schema-
utils_2.12:0.1.0") \
    .config("spark.executor.cores", "1") \
    .config('spark.executor.memory', '5gb') \
    .config('spark.executor.memoryOverhead', '1500') \
    .config('spark.driver.memoryOverhead', '1500') \
    .config("spark.sql.shuffle.partitions", "480") \
    .config("spark.sql.execution.arrow.enabled", "true") \
    .config("spark.driver.maxResultSize", "50gb") \
    .getOrCreate()
# spark.conf.set("spark.sql.execution.arrow.enabled", "true") # deprecated
print("Apache Spark version:")
print(spark.version)
sc = spark.sparkContext
sqlContext = SQLContext(sc)
parser = argparse.ArgumentParser(description='Spark DCN CTR Prediction
Example',

formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--data-dir', default='file://' + os.getcwd(),
                    help='location of data on local filesystem (prefixed
with file://) or on HDFS')
def process_input_file(file_name, sparse_feat, dense_feat):
    # Need this preprocessing to turn Criteo raw file into CSV:
    print("START processing input file...")
    # only convert the file ONCE
```

```
    # sample = open(file_name)
    # sample = '\n'.join([str(x.replace('\n', '').replace('\t', ',')) for
x in sample])
    # # Add header in data file and save as CSV
    # header = ','.join(str(x) for x in (['label'] + dense_feat +
sparse_feat))
    # with open('/sparkdemo/tr-4570-data/ctr_train.csv', mode='w',
encoding="utf-8") as f:
    #     f.write(header + '\n' + sample)
    #     f.close()
    # print("Raw training file processed and saved as CSV: ", f.name)
    raw_df = sqlContext.read.option("header", True).csv(file_name)
    raw_df.show(5, False)
    raw_df.printSchema()
    # convert columns I1 to I13 from string to integers
    conv_df = raw_df.select(col('label').cast("double"),
                            *(col(i).cast("float").alias(i) for i in
raw_df.columns if i in dense_feat),
                            *(col(c) for c in raw_df.columns if c in
sparse_feat))
    print("Schema of raw_df with integer columns type changed:")
    conv_df.printSchema()
    # result_pdf = conv_df.select("*").toPandas()
    tmp_df = conv_df.na.fill(0, dense_feat)
    result_df = tmp_df.na.fill('-1', sparse_feat)
    result_df.show()
    return result_df
if __name__ == "__main__":
    args = parser.parse_args()
    # Pandas read CSV
    # data = pd.read_csv('%s/criteo_sample.txt' % args.data_dir)
    # print("Obtained Pandas df.")
    dense_features = ['I' + str(i) for i in range(1, 14)]
    sparse_features = ['C' + str(i) for i in range(1, 27)]
    # Spark read CSV
    # process_input_file('%s/train.txt' % args.data_dir, sparse_features,
dense_features) # run only ONCE
    spark_df = process_input_file('%s/data.txt' % args.data_dir,
sparse_features, dense_features) # sample data
    # spark_df = process_input_file('%s/ctr_train.csv' % args.data_dir,
sparse_features, dense_features)
    print("Obtained Spark df and filled in missing features.")
    data = spark_df
    # Pandas
    #data[sparse_features] = data[sparse_features].fillna('-1', )
    #data[dense_features] = data[dense_features].fillna(0, )
```

```
    target = ['label']
    label_npa = data.select("label").toPandas().to_numpy()
    print("label numPy array has length = ", len(label_npa)) # 45,840,617
w/ 11GB dataset
    label_npa.ravel()
    label_npa.reshape(len(label_npa), )
    # 1.Label Encoding for sparse features,and do simple Transformation
for dense features
    print("Before LabelEncoder():")
    data.printSchema()  # label: float (nullable = true)
    for feat in sparse_features:
        lbe = LabelEncoder()
        tmp_pdf = data.select(feat).toPandas().to_numpy()
        tmp_ndarray = lbe.fit_transform(tmp_pdf)
        print("After LabelEncoder(), tmp_ndarray[0] =", tmp_ndarray[0])
        # print("Data tmp PDF after lbe transformation, the output ndarray
has length = ", len(tmp_ndarray)) # 45,840,617 for 11GB dataset
        tmp_ndarray.ravel()
        tmp_ndarray.reshape(len(tmp_ndarray), )
        out_ndarray = np.column_stack([label_npa, tmp_ndarray])
        pdf = pd.DataFrame(out_ndarray, columns=['label', feat])
        s_df = spark.createDataFrame(pdf)
        s_df.printSchema() # label: double (nullable = true)
        print("Before joining data df with s_df, s_df example rows:")
        s_df.show(1, False)
        data = data.drop(feat).join(s_df, 'label').drop('label')
        print("After LabelEncoder(), data df example rows:")
        data.show(1, False)
        print("Finished processing sparse_features: ", feat)
    print("Data DF after label encoding: ")
    data.show()
    data.printSchema()
    mms = MinMaxScaler(feature_range=(0, 1))
    # data[dense_features] = mms.fit_transform(data[dense_features]) # for
Pandas df
    tmp_pdf = data.select(dense_features).toPandas().to_numpy()
    tmp_ndarray = mms.fit_transform(tmp_pdf)
    tmp_ndarray.ravel()
    tmp_ndarray.reshape(len(tmp_ndarray), len(tmp_ndarray[0]))
    out_ndarray = np.column_stack([label_npa, tmp_ndarray])
    pdf = pd.DataFrame(out_ndarray, columns=['label'] + dense_features)
    s_df = spark.createDataFrame(pdf)
    s_df.printSchema()
    data.drop(*dense_features).join(s_df, 'label').drop('label')
    print("Finished processing dense_features: ", dense_features)
    print("Data DF after MinMaxScaler: ")
```

```
    data.show()

    # 2.count #unique features for each sparse field,and record dense
feature field name
    fixlen_feature_columns = [SparseFeat(feat,
vocabulary_size=data.select(feat).distinct().count() + 1, embedding_dim=4)
                              for i, feat in enumerate(sparse_features)] +
\
                             [DenseFeat(feat, 1, ) for feat in
dense_features]
    dnn_feature_columns = fixlen_feature_columns
    linear_feature_columns = fixlen_feature_columns
    feature_names = get_feature_names(linear_feature_columns +
dnn_feature_columns)
    # 3.generate input data for model
    # train, test = train_test_split(data.toPandas(), test_size=0.2,
random_state=2020) # Pandas; might hang for 11GB data
    train, test = data.randomSplit(weights=[0.8, 0.2], seed=200)
    print("Training dataset size = ", train.count())
    print("Testing dataset size = ", test.count())
    # Pandas:
    # train_model_input = {name: train[name] for name in feature_names}
    # test_model_input = {name: test[name] for name in feature_names}
    # Spark DF:
    train_model_input = {}
    test_model_input = {}
    for name in feature_names:
        if name.startswith('I'):
            tr_pdf = train.select(name).toPandas()
            train_model_input[name] = pd.to_numeric(tr_pdf[name])
            ts_pdf = test.select(name).toPandas()
            test_model_input[name] = pd.to_numeric(ts_pdf[name])
    # 4.Define Model,train,predict and evaluate
    model = DeepFM(linear_feature_columns, dnn_feature_columns,
task='binary')
    model.compile("adam", "binary_crossentropy",
                  metrics=['binary_crossentropy'], )
    lb_pdf = train.select(target).toPandas()
    history = model.fit(train_model_input,
pd.to_numeric(lb_pdf['label']).values,
                        batch_size=256, epochs=10, verbose=2,
validation_split=0.2, )
    pred_ans = model.predict(test_model_input, batch_size=256)
    print("test LogLoss",
round(log_loss(pd.to_numeric(test.select(target).toPandas()).values,
pred_ans), 4))
```

```
    print("test AUC",
round(roc_auc_score(pd.to_numeric(test.select(target).toPandas()).values,
pred_ans), 4))
```

Conclusion

Dans ce document, nous discutons de l'architecture Apache Spark, des cas d'utilisation des clients et du portefeuille de stockage NetApp en relation avec le Big Data, l'analyse moderne et l'IA, le ML et le DL. Lors de nos tests de validation des performances basés sur des outils d'analyse comparative standard du secteur et sur la demande des clients, les solutions NetApp Spark ont démontré des performances supérieures par rapport aux systèmes Hadoop natifs. Une combinaison des cas d'utilisation client et des résultats de performances présentés dans ce rapport peut vous aider à choisir une solution Spark appropriée pour votre déploiement.

Où trouver des informations supplémentaires

Les références suivantes ont été utilisées dans ce TR :

- Architecture et composants d'Apache Spark

  "http://spark.apache.org/docs/latest/cluster-overview.html"

- Cas d'utilisation d'Apache Spark

  "https://www.qubole.com/blog/big-data/apache-spark-use-cases/"

- Spark PNL

  "https://www.johnsnowlabs.com/spark-nlp/"

- BERT

  "https://arxiv.org/abs/1810.04805"

- Réseau profond et croisé pour les prévisions de clics publicitaires

  "https://arxiv.org/abs/1708.05123"

- FlexGroup

  https://www.netapp.com/pdf.html?item=/media/7337-tr4557pdf.pdf

- ETL en continu

  "https://www.infoq.com/articles/apache-spark-streaming"

- Solutions NetApp E-Series pour Hadoop

  "https://www.netapp.com/media/16420-tr-3969.pdf"

- Solutions d’analyse de données modernes NetApp

  "Solutions d’analyse de données"

- SnapMirror

  "https://docs.netapp.com/us-en/ontap/data-protection/snapmirror-replication-concept.html"

- XCP

  https://mysupport.netapp.com/documentation/docweb/index.html?productID=63942&language=en-US

- Copie et synchronisation BlueXP

  "https://cloud.netapp.com/cloud-sync-service"

- Boîte à outils DataOps

  "https://github.com/NetApp/netapp-dataops-toolkit"

Informations sur le copyright

Informations sur les marques commerciales