Soluzioni di storage NetApp per Apache Spark

NetApp artificial intelligence solutions

NetApp
August 18, 2025

Sommario

Soluzioni di storage NetApp per Apache Spark

TR-4570: Soluzioni di storage NetApp per Apache Spark: architettura, casi d'uso e risultati delle prestazioni

Rick Huang, Karthikeyan Nagalingam, NetApp

Questo documento si concentra sull'architettura di Apache Spark, sui casi d'uso dei clienti e sul portafoglio di storage NetApp correlato all'analisi dei big data e all'intelligenza artificiale (IA). Presenta inoltre vari risultati di test effettuati utilizzando strumenti di intelligenza artificiale, apprendimento automatico (ML) e apprendimento profondo (DL) standard del settore su un tipico sistema Hadoop, in modo da poter scegliere la soluzione Spark più adatta. Per iniziare, hai bisogno di un'architettura Spark, di componenti appropriati e di due modalità di distribuzione (cluster e client).

Questo documento fornisce inoltre casi d'uso dei clienti per risolvere problemi di configurazione e fornisce una panoramica del portafoglio di storage NetApp rilevante per l'analisi dei big data e l'intelligenza artificiale, l'apprendimento automatico e il apprendimento automatico con Spark. Concludiamo quindi con i risultati dei test derivati dai casi d'uso specifici di Spark e dal portafoglio di soluzioni NetApp Spark.

Sfide dei clienti

Questa sezione si concentra sulle sfide dei clienti relative all'analisi dei big data e all'intelligenza artificiale, al machine learning e al digital learning nei settori in cui i dati crescono, come la vendita al dettaglio, il marketing digitale, il settore bancario, la produzione discreta, la produzione di processo, la pubblica amministrazione e i servizi professionali.

Prestazioni imprevedibili

Le distribuzioni Hadoop tradizionali utilizzano in genere hardware di base. Per migliorare le prestazioni, è necessario ottimizzare la rete, il sistema operativo, il cluster Hadoop, i componenti dell'ecosistema come Spark e l'hardware. Anche se si ottimizza ogni livello, può essere difficile raggiungere i livelli di prestazioni desiderati perché Hadoop viene eseguito su hardware di base che non è stato progettato per prestazioni elevate nel tuo ambiente.

Guasti dei media e dei nodi

Anche in condizioni normali, l'hardware di base è soggetto a guasti. Se un disco su un nodo dati si guasta, per impostazione predefinita il master Hadoop considera quel nodo non funzionante. Quindi copia dati specifici da quel nodo attraverso la rete dalle repliche a un nodo sano. Questo processo rallenta i pacchetti di rete per qualsiasi attività Hadoop. Il cluster deve quindi copiare nuovamente i dati e rimuovere i dati sovrareplicati quando il nodo non funzionante torna a uno stato funzionante.

Blocco del fornitore Hadoop

I distributori Hadoop dispongono di una propria distribuzione Hadoop con il proprio controllo delle versioni, che vincola il cliente a tali distribuzioni. Tuttavia, molti clienti necessitano di supporto per l'analisi in memoria che non vincoli il cliente a specifiche distribuzioni Hadoop. Hanno bisogno della libertà di modificare le distribuzioni e continuare a portare con sé le proprie analisi.

Mancanza di supporto per più di una lingua

Oltre ai programmi MapReduce Java, spesso i clienti necessitano del supporto per più lingue per eseguire i loro lavori. Opzioni come SQL e script offrono maggiore flessibilità per ottenere risposte, più possibilità per organizzare e recuperare i dati e modi più rapidi per spostare i dati in un framework di analisi.

Difficoltà d'uso

Da un po' di tempo le persone si lamentano della difficoltà di utilizzo di Hadoop. Sebbene Hadoop sia diventato più semplice e potente con ogni nuova versione, questa critica persiste. Hadoop richiede la conoscenza dei modelli di programmazione Java e MapReduce, una sfida per gli amministratori di database e per chi ha competenze di scripting tradizionali.

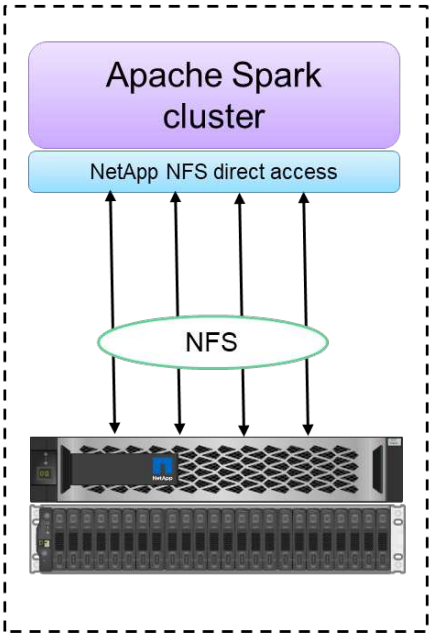
Framework e strumenti complicati

I team di intelligenza artificiale delle aziende devono affrontare molteplici sfide. Anche con conoscenze specialistiche in materia di data science, gli strumenti e i framework per diversi ecosistemi di distribuzione e applicazioni potrebbero non essere facilmente traducibili dall'uno all'altro. Una piattaforma di data science dovrebbe integrarsi perfettamente con le corrispondenti piattaforme big data basate su Spark, con facilità di spostamento dei dati, modelli riutilizzabili, codice pronto all'uso e strumenti che supportano le migliori pratiche per la prototipazione, la convalida, il controllo delle versioni, la condivisione, il riutilizzo e la rapida distribuzione dei modelli in produzione.

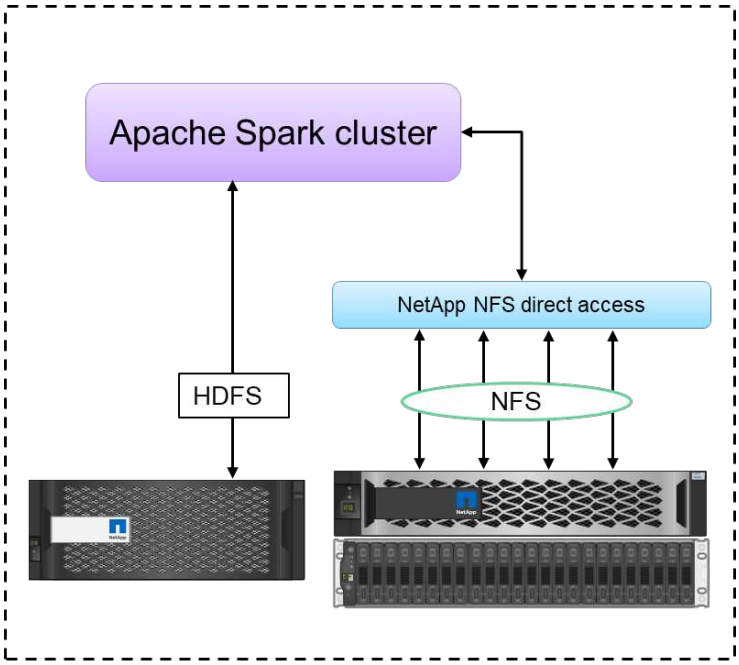
Perché scegliere NetApp?

NetApp può migliorare la tua esperienza Spark nei seguenti modi:

- L'accesso diretto NFS NetApp (mostrato nella figura sottostante) consente ai clienti di eseguire attività di analisi di big data sui propri dati NFSv3 o NFSv4 nuovi o esistenti senza dover spostare o copiare i dati. Impedisce la creazione di copie multiple dei dati ed elimina la necessità di sincronizzare i dati con una fonte.
- Archiviazione più efficiente e minore replicazione del server. Ad esempio, la soluzione NetApp E-Series Hadoop richiede due anziché tre repliche dei dati, mentre la soluzione FAS Hadoop richiede un'origine dati ma nessuna replica o copia dei dati. Le soluzioni di storage NetApp generano inoltre meno traffico tra server.
- Miglioramento del lavoro Hadoop e del comportamento del cluster in caso di guasti di unità e nodi.
- Migliori prestazioni di acquisizione dei dati.



Configuration 1: NFS as primary storage



Configuration 2: HDFS and NFS in single Spark cluster

Ad esempio, nel settore finanziario e sanitario, lo spostamento dei dati da un luogo all'altro deve rispettare obblighi di legge, il che non è un compito facile. In questo scenario, l'accesso diretto NFS NetApp analizza i dati finanziari e sanitari dalla loro posizione originale. Un altro vantaggio fondamentale è che l'utilizzo dell'accesso diretto NFS NetApp semplifica la protezione dei dati Hadoop mediante l'utilizzo di comandi Hadoop nativi e l'abilitazione di flussi di lavoro di protezione dei dati con il ricco portafoglio di gestione dei dati di NetApp.

L'accesso diretto NFS NetApp offre due tipi di opzioni di distribuzione per i cluster Hadoop/Spark:

- Per impostazione predefinita, i cluster Hadoop o Spark utilizzano Hadoop Distributed File System (HDFS) per l'archiviazione dei dati e il file system predefinito. L'accesso diretto a NFS NetApp può sostituire l'HDFS predefinito con l'archiviazione NFS come file system predefinito, consentendo l'analisi diretta sui dati NFS.
- In un'altra opzione di distribuzione, l'accesso diretto a NetApp NFS supporta la configurazione di NFS come storage aggiuntivo insieme a HDFS in un singolo cluster Hadoop o Spark. In questo caso, il cliente può condividere i dati tramite esportazioni NFS e accedervi dallo stesso cluster insieme ai dati HDFS.

I principali vantaggi dell'utilizzo dell'accesso diretto NFS NetApp includono quanto segue:

- Analizzando i dati dalla loro posizione attuale, si evita il compito, dispendioso in termini di tempo e prestazioni, di spostare i dati analitici su un'infrastruttura Hadoop come HDFS.
- Riduzione del numero di repliche da tre a una.
- Consentire agli utenti di disaccoppiare elaborazione e archiviazione per scalarli in modo indipendente.
- Garantire la protezione dei dati aziendali sfruttando le avanzate funzionalità di gestione dei dati di ONTAP.
- Certificazione con la piattaforma dati Hortonworks.
- Abilitazione di implementazioni di analisi dei dati ibride.
- Riduzione dei tempi di backup sfruttando la capacità multithread dinamica.

Vedere"TR-4657: Soluzioni dati cloud ibride NetApp - Spark e Hadoop basate sui casi d'uso dei clienti" per il backup dei dati Hadoop, il backup e il disaster recovery dal cloud all'ambiente locale, abilitando DevTest sui

dati Hadoop esistenti, la protezione dei dati e la connettività multicloud e accelerando i carichi di lavoro di analisi.

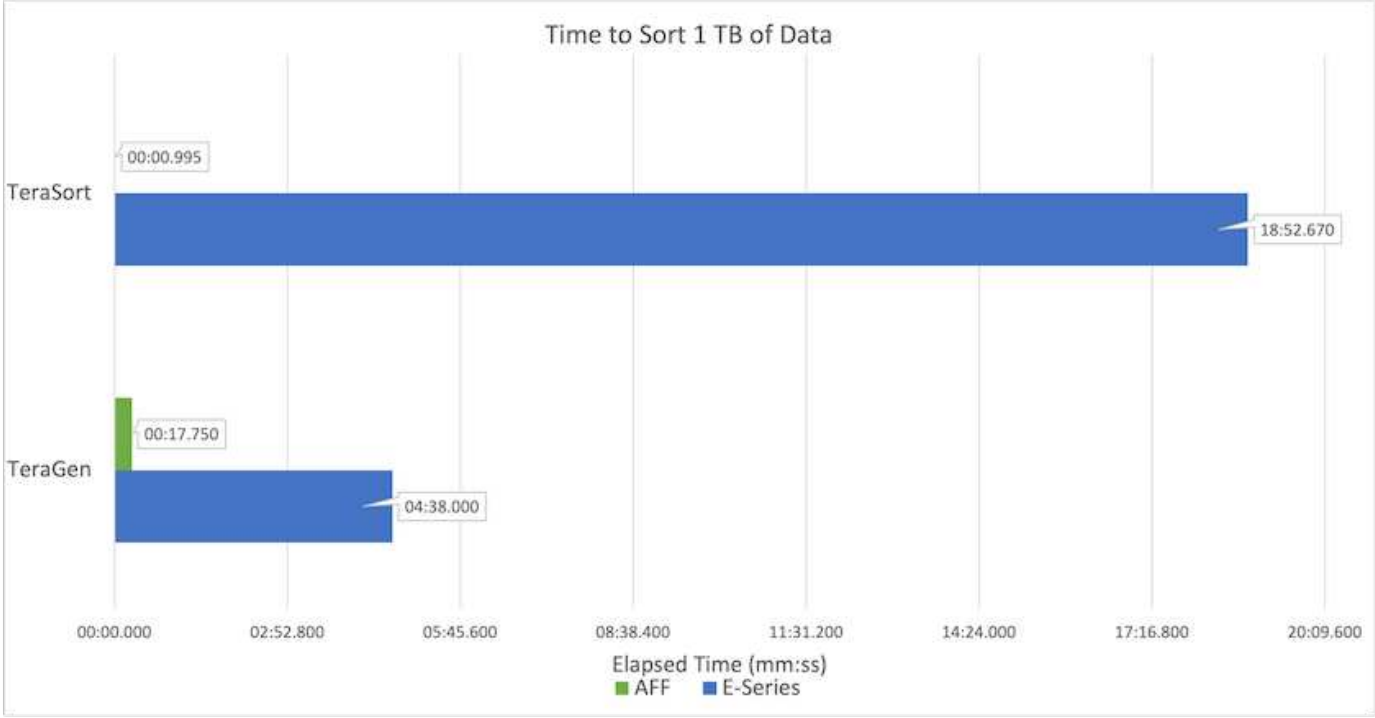
Le sezioni seguenti descrivono le capacità di archiviazione importanti per i clienti Spark.

Livelli di archiviazione

Grazie alla suddivisione in livelli di archiviazione di Hadoop, è possibile archiviare i file con diversi tipi di archiviazione in base a una policy di archiviazione. I tipi di archiviazione includono `hot` , `cold` , `warm` , `all_ssd` , `one_ssd` , E `lazy_persist` .

Abbiamo eseguito la convalida della suddivisione in livelli di archiviazione Hadoop su un controller di archiviazione NetApp AFF e un controller di archiviazione E-Series con unità SSD e SAS con diverse policy di archiviazione. Il cluster Spark con AFF-A800 ha quattro nodi di elaborazione, mentre il cluster con E-Series ne ha otto. Lo scopo principale è confrontare le prestazioni delle unità a stato solido (SSD) rispetto ai dischi rigidi (HDD).

La figura seguente mostra le prestazioni delle soluzioni NetApp per un SSD Hadoop.



- La configurazione NL-SAS di base utilizzava otto nodi di elaborazione e 96 unità NL-SAS. Questa configurazione ha generato 1 TB di dati in 4 minuti e 38 secondi. Vedere "TR-3969 Soluzione NetApp E-Series per Hadoop" per i dettagli sulla configurazione del cluster e dello storage.
- Utilizzando TeraGen, la configurazione SSD ha generato 1 TB di dati 15,66 volte più velocemente rispetto alla configurazione NL-SAS. Inoltre, la configurazione SSD utilizzava la metà del numero di nodi di elaborazione e la metà del numero di unità disco (24 unità SSD in totale). In base al tempo di completamento del lavoro, è stato quasi il doppio più veloce della configurazione NL-SAS.
- Utilizzando TeraSort, la configurazione SSD ha ordinato 1 TB di dati 1138,36 volte più velocemente rispetto alla configurazione NL-SAS. Inoltre, la configurazione SSD utilizzava la metà del numero di nodi di elaborazione e la metà del numero di unità disco (24 unità SSD in totale). Pertanto, per unità, era circa tre volte più veloce della configurazione NL-SAS.
- La conclusione è che il passaggio dai dischi rotanti a quelli completamente flash migliora le prestazioni. Il

collo di bottiglia non era il numero di nodi di elaborazione. Grazie allo storage all-flash di NetApp, le prestazioni di runtime sono ben scalabili.

- Con NFS, i dati erano funzionalmente equivalenti a essere raggruppati tutti insieme, il che può ridurre il numero di nodi di elaborazione a seconda del carico di lavoro. Gli utenti del cluster Apache Spark non devono ribilanciare manualmente i dati quando cambiano il numero di nodi di elaborazione.

Scalabilità delle prestazioni - Scalabilità orizzontale

Quando è necessaria una maggiore potenza di calcolo da un cluster Hadoop in una soluzione AFF , è possibile aggiungere nodi dati con un numero appropriato di controller di archiviazione. NetApp consiglia di iniziare con quattro nodi dati per array di controller di storage e di aumentare il numero a otto nodi dati per controller di storage, a seconda delle caratteristiche del carico di lavoro.

AFF e FAS sono perfetti per l'analisi in loco. In base ai requisiti di calcolo, è possibile aggiungere gestori di nodi e le operazioni non disruptive consentono di aggiungere un controller di storage su richiesta senza tempi di inattività. Offriamo funzionalità avanzate con AFF e FAS, come supporto multimediale NVME, efficienza garantita, riduzione dei dati, QOS, analisi predittiva, cloud tiering, replicazione, distribuzione cloud e sicurezza. Per aiutare i clienti a soddisfare le loro esigenze, NetApp offre funzionalità quali analisi del file system, quote e bilanciamento del carico integrato, senza costi di licenza aggiuntivi. NetApp offre prestazioni migliori in termini di numero di processi contemporanei, latenza inferiore, operazioni più semplici e throughput di gigabyte al secondo più elevato rispetto ai nostri concorrenti. Inoltre, NetApp Cloud Volumes ONTAP è compatibile con tutti e tre i principali provider cloud.

Scalabilità delle prestazioni - Scalabilità verticale

Le funzionalità di scalabilità consentono di aggiungere unità disco ai sistemi AFF, FAS ed E-Series quando è necessaria ulteriore capacità di archiviazione. Con Cloud Volumes ONTAP, il ridimensionamento dello storage al livello PB è una combinazione di due fattori: il livellamento dei dati utilizzati raramente nello storage di oggetti dallo storage a blocchi e l'accumulo di licenze Cloud Volumes ONTAP senza elaborazione aggiuntiva.

Protocolli multipli

I sistemi NetApp supportano la maggior parte dei protocolli per le distribuzioni Hadoop, tra cui SAS, iSCSI, FCP, InfiniBand e NFS.

Soluzioni operative e supportate

Le soluzioni Hadoop descritte in questo documento sono supportate da NetApp. Queste soluzioni sono inoltre certificate dai principali distributori Hadoop. Per informazioni, vedere il "Hortonworks" sito e Cloudera "certificazione" E "partner" siti.

Pubblico di destinazione

Il mondo dell'analisi e della scienza dei dati tocca molteplici discipline dell'IT e del business:

- Lo scienziato dei dati ha bisogno della flessibilità necessaria per utilizzare gli strumenti e le librerie che preferisce.
- L'ingegnere dei dati deve sapere come fluiscono i dati e dove risiedono.
- Un ingegnere DevOps ha bisogno degli strumenti per integrare nuove applicazioni di intelligenza artificiale e apprendimento automatico nelle proprie pipeline di CI e CD.

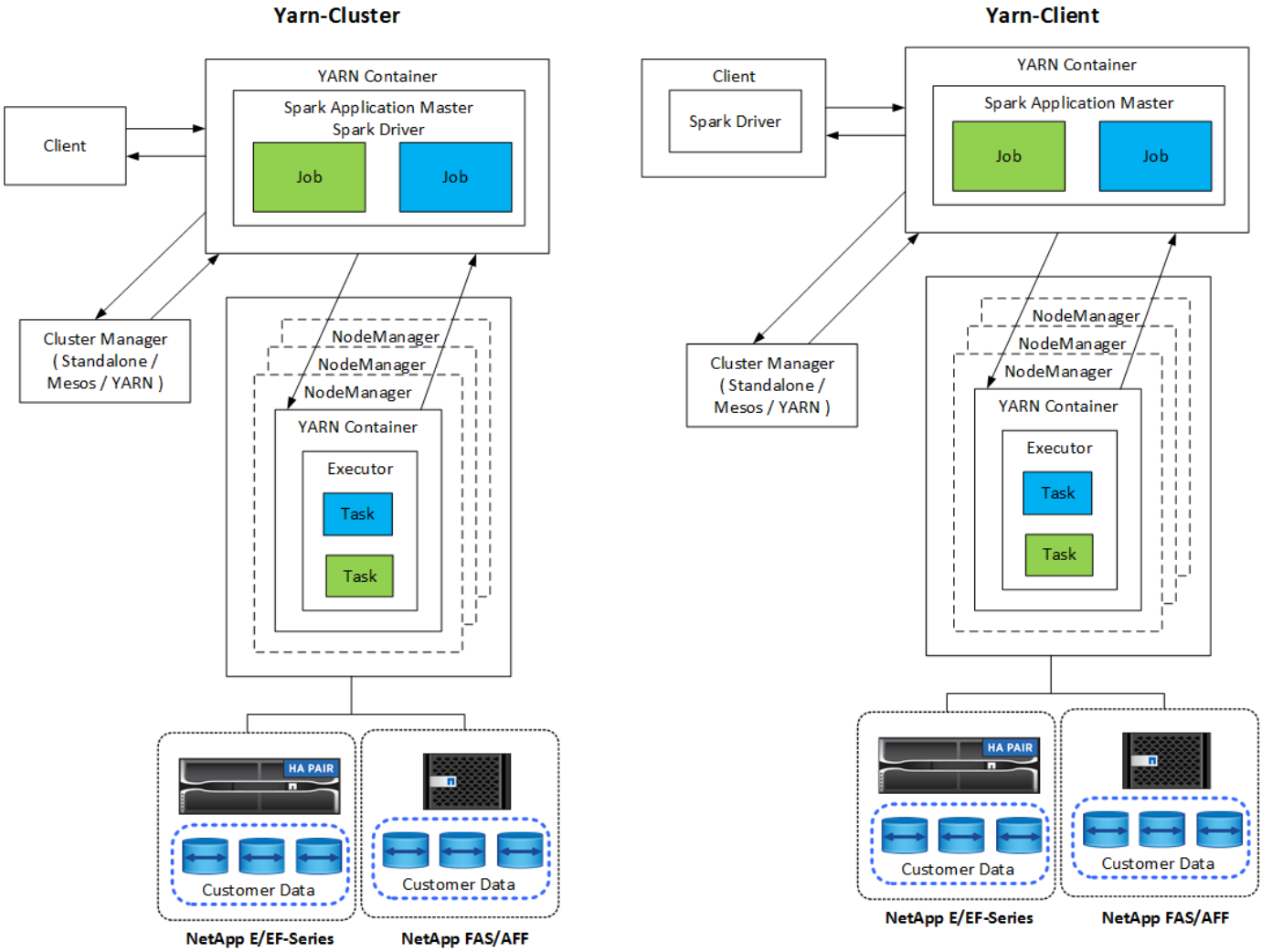
- Gli amministratori e gli architetti cloud devono essere in grado di configurare e gestire le risorse cloud ibride.
- Gli utenti aziendali desiderano avere accesso ad applicazioni di analisi, intelligenza artificiale, apprendimento automatico e apprendimento automatico.

In questo report tecnico descriviamo come NetApp AFF, E-Series, StorageGRID, NFS direct access, Apache Spark, Horovod e Keras aiutano ciascuno di questi ruoli a portare valore al business.

Tecnologia delle soluzioni

Apache Spark è un framework di programmazione diffuso per la scrittura di applicazioni Hadoop che funziona direttamente con Hadoop Distributed File System (HDFS). Spark è pronto per la produzione, supporta l'elaborazione di dati in streaming ed è più veloce di MapReduce. Spark dispone di una cache dei dati in memoria configurabile per un'iterazione efficiente e la shell Spark è interattiva per l'apprendimento e l'esplorazione dei dati. Con Spark puoi creare applicazioni in Python, Scala o Java. Le applicazioni Spark sono costituite da uno o più lavori che hanno una o più attività.

Ogni applicazione Spark ha un driver Spark. In modalità YARN-Client, il driver viene eseguito localmente sul client. In modalità YARN-Cluster, il driver viene eseguito nel cluster sul master dell'applicazione. Nella modalità cluster, l'applicazione continua a funzionare anche se il client si disconnette.



Esistono tre gestori di cluster:

- **Autonomo.** Questo gestore fa parte di Spark e semplifica la configurazione di un cluster.
- **Apache Mesos.** Si tratta di un gestore di cluster generale che esegue anche MapReduce e altre applicazioni.
- **FILATO Hadoop.** Questo è un gestore di risorse in Hadoop 3.

Il set di dati distribuito resiliente (RDD) è il componente principale di Spark. RDD ricrea i dati persi e mancanti dai dati archiviati nella memoria del cluster e memorizza i dati iniziali che provengono da un file o sono creati a livello di programmazione. Gli RDD vengono creati da file, dati in memoria o un altro RDD. La programmazione Spark esegue due operazioni: trasformazione e azioni. La trasformazione crea un nuovo RDD basato su uno esistente. Le azioni restituiscono un valore da un RDD.

Le trasformazioni e le azioni si applicano anche ai dataset e ai dataframe Spark. Un set di dati è una raccolta distribuita di dati che offre i vantaggi degli RDD (tipizzazione forte, utilizzo delle funzioni lambda) con i vantaggi del motore di esecuzione ottimizzato di Spark SQL. Un set di dati può essere costruito da oggetti JVM e poi manipolato utilizzando trasformazioni funzionali (map, flatMap, filter e così via). Un DataFrame è un set di dati organizzato in colonne denominate. Concettualmente è equivalente a una tabella in un database relazionale o a un data frame in R/Python. I DataFrame possono essere creati da una vasta gamma di fonti, come file di dati strutturati, tabelle in Hive/HBase, database esterni in locale o nel cloud o RDD esistenti.

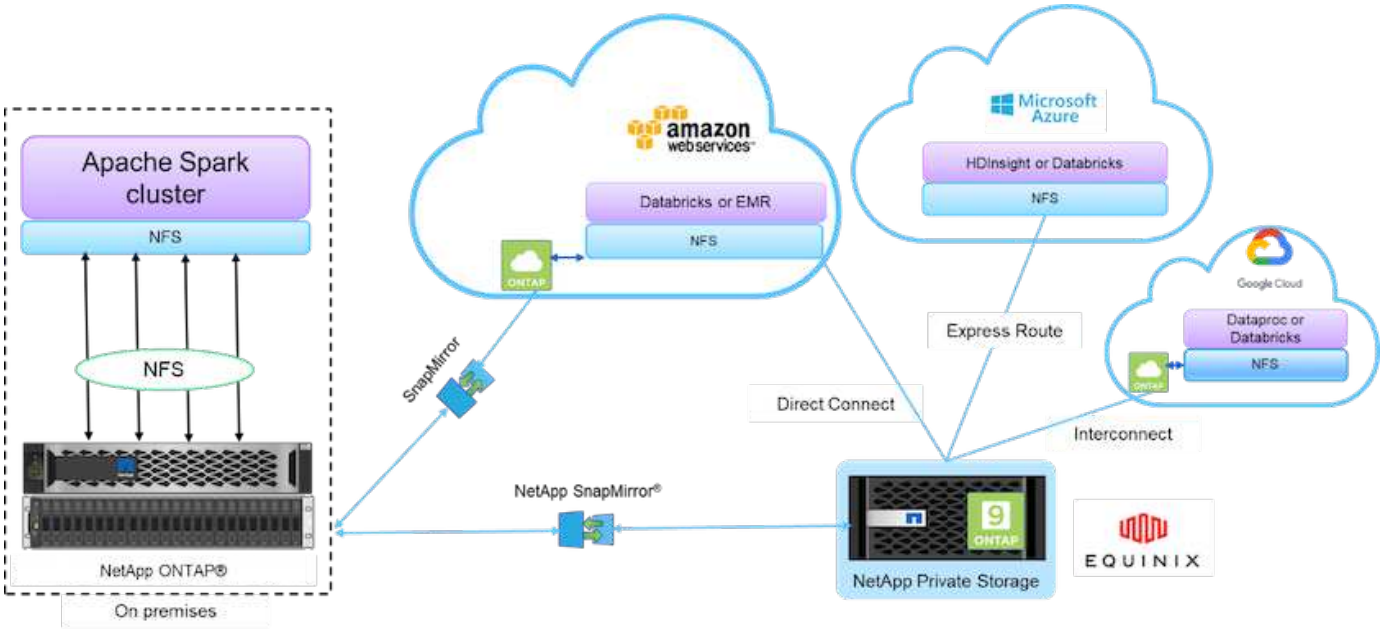
Le applicazioni Spark includono uno o più lavori Spark. I lavori eseguono le attività negli esecutori e gli

esecutori vengono eseguiti nei contenitori YARN. Ogni esecutore viene eseguito in un singolo contenitore e gli esecutori esistono per tutta la durata di un'applicazione. Un esecutore viene fissato dopo l'avvio dell'applicazione e YARN non ridimensiona il contenitore già allocato. Un esecutore può eseguire attività contemporaneamente sui dati in memoria.

Panoramica delle soluzioni NetApp Spark

NetApp dispone di tre portafogli di storage: FAS/ AFF, E-Series e Cloud Volumes ONTAP. Abbiamo convalidato AFF e la serie E con il sistema di archiviazione ONTAP per le soluzioni Hadoop con Apache Spark.

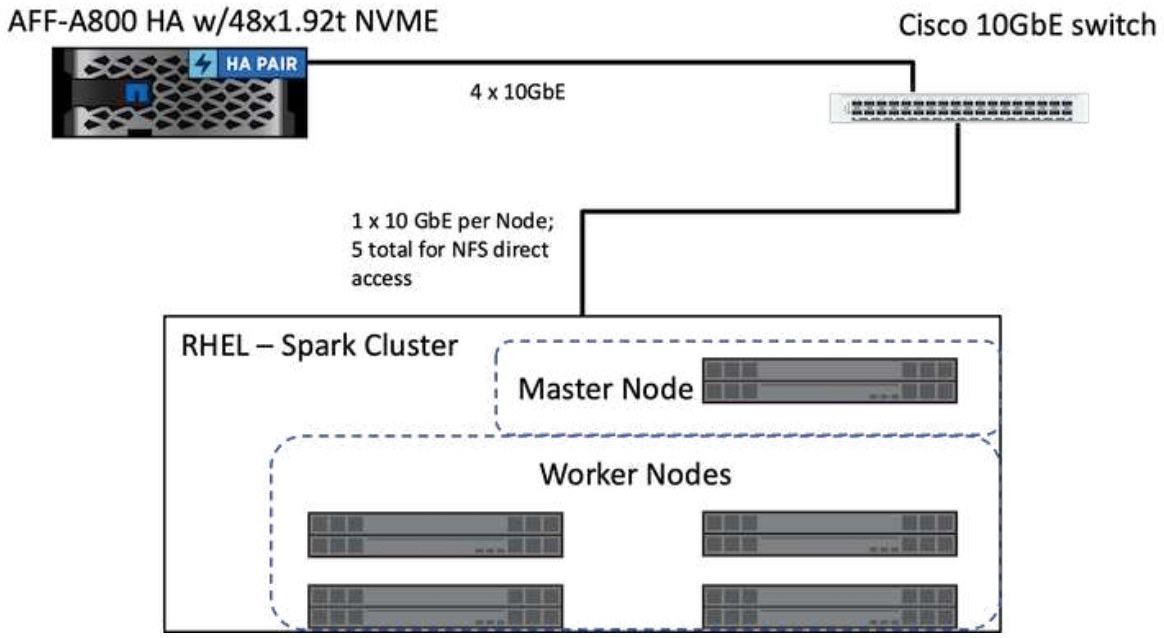
Il data fabric basato su NetApp integra servizi e applicazioni di gestione dei dati (elementi costitutivi) per l'accesso, il controllo, la protezione e la sicurezza dei dati, come mostrato nella figura seguente.



Gli elementi costitutivi nella figura sopra includono:

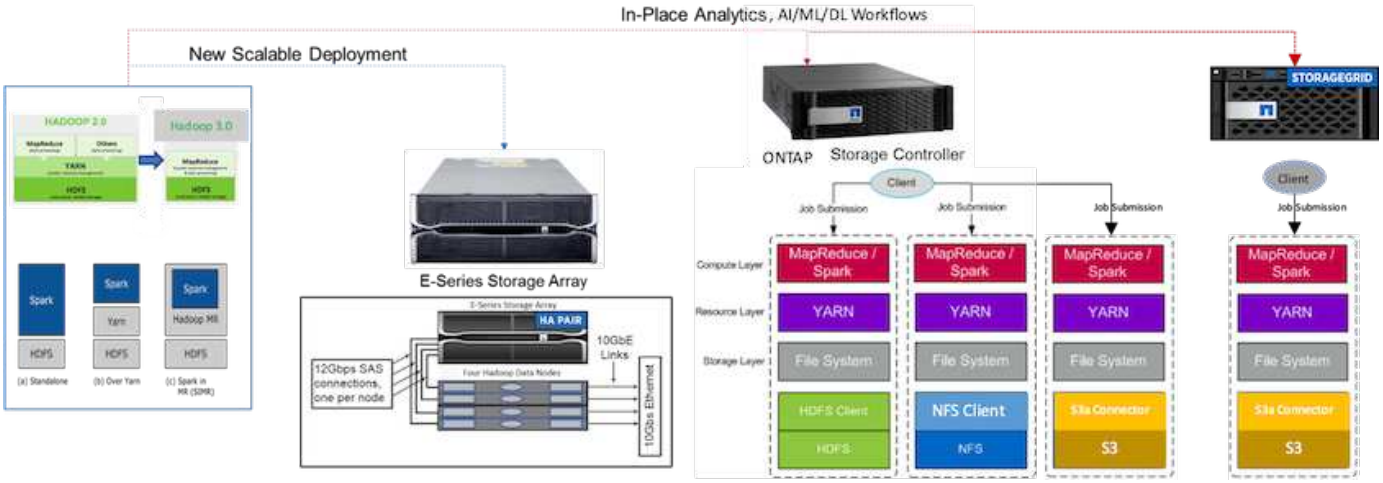
- * Accesso diretto NetApp NFS.* Fornisce i cluster Hadoop e Spark più recenti con accesso diretto ai volumi NetApp NFS senza requisiti aggiuntivi di software o driver.
- * NetApp Cloud Volumes ONTAP e Google Cloud NetApp Volumes.* Archiviazione connessa definita dal software basata su ONTAP in esecuzione in Amazon Web Services (AWS) o Azure NetApp Files (ANF) nei servizi cloud di Microsoft Azure.
- * Tecnologia NetApp SnapMirror .* Fornisce funzionalità di protezione dei dati tra istanze locali e istanze ONTAP Cloud o NPS.
- **Fornitori di servizi cloud.** Tra questi fornitori figurano AWS, Microsoft Azure, Google Cloud e IBM Cloud.
- **PaaS.** Servizi di analisi basati su cloud come Amazon Elastic MapReduce (EMR) e Databricks in AWS, nonché Microsoft Azure HDInsight e Azure Databricks.

La figura seguente illustra la soluzione Spark con storage NetApp .

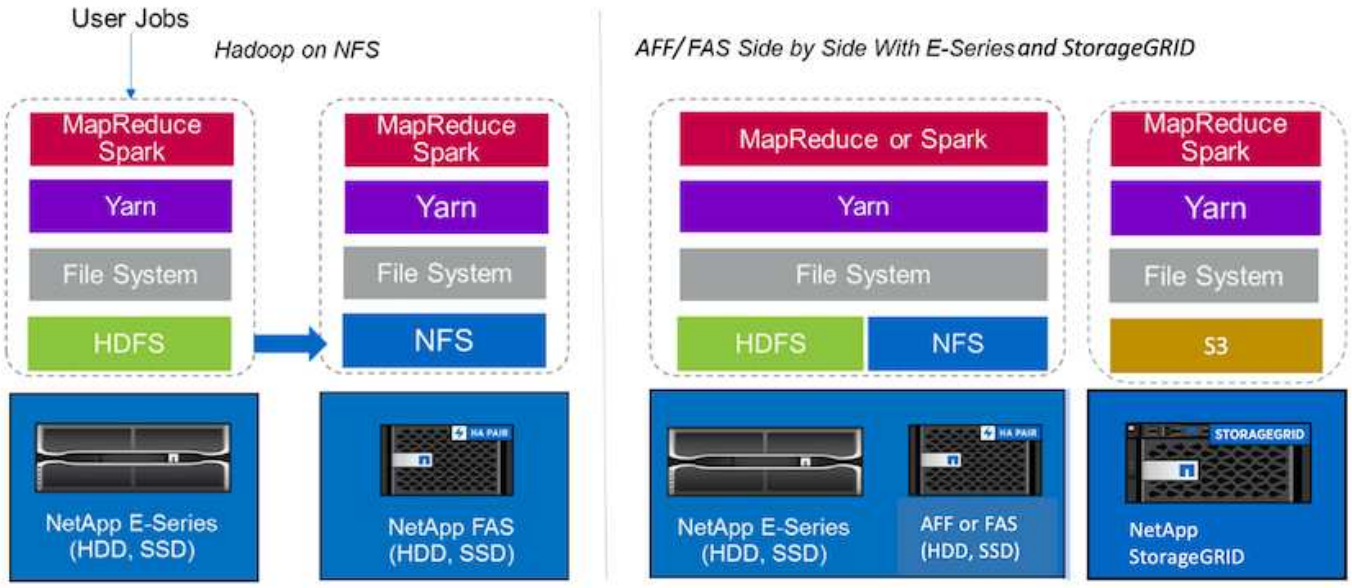


La soluzione ONTAP Spark utilizza il protocollo di accesso diretto NetApp NFS per analisi in loco e flussi di lavoro AI, ML e DL utilizzando l'accesso ai dati di produzione esistenti. I dati di produzione disponibili per i nodi Hadoop vengono esportati per eseguire analisi in loco e attività di intelligenza artificiale, apprendimento automatico e apprendimento automatico (IA), nonché attività di DL. È possibile accedere ai dati da elaborare nei nodi Hadoop con o senza accesso diretto NetApp NFS. In Spark con la versione standalone o `yarn` gestore cluster, è possibile configurare un volume NFS utilizzando `file://<target_volume>` . Abbiamo convalidato tre casi d'uso con diversi set di dati. I dettagli di queste convalide sono presentati nella sezione "Risultati dei test". (xrif)

La figura seguente illustra il posizionamento dello storage NetApp Apache Spark/Hadoop.



Abbiamo identificato le caratteristiche uniche della soluzione E-Series Spark, della soluzione AFF/ FAS ONTAP Spark e della soluzione StorageGRID Spark, e abbiamo eseguito convalide e test dettagliati. Sulla base delle nostre osservazioni, NetApp consiglia la soluzione E-Series per installazioni greenfield e nuove distribuzioni scalabili e la soluzione AFF/ FAS per analisi in loco, carichi di lavoro AI, ML e DL utilizzando dati NFS esistenti e StorageGRID per AI, ML e DL e analisi dei dati moderne quando è richiesto l'archiviazione di oggetti.



Un data lake è un repository di archiviazione per grandi set di dati in formato nativo che può essere utilizzato per attività di analisi, intelligenza artificiale, apprendimento automatico e DL. Abbiamo creato un repository di data lake per le soluzioni E-Series, AFF/ FAS e StorageGRID SG6060 Spark. Il sistema E-Series fornisce l'accesso HDFS al cluster Hadoop Spark, mentre l'accesso ai dati di produzione esistenti avviene tramite il protocollo di accesso diretto NFS al cluster Hadoop. Per i set di dati che risiedono nell'archiviazione di oggetti, NetApp StorageGRID fornisce accesso sicuro S3 e S3a.

Riepilogo del caso d'uso

In questa pagina vengono descritti i diversi ambiti in cui è possibile utilizzare questa soluzione.

dati in streaming

Apache Spark può elaborare dati in streaming, utilizzati per i processi di estrazione, trasformazione e caricamento (ETL), arricchimento dei dati, attivazione del rilevamento di eventi e analisi di sessioni complesse:

- **Streaming ETL.** I dati vengono costantemente puliti e aggregati prima di essere inseriti negli archivi dati. Netflix utilizza Kafka e Spark Streaming per creare una soluzione di monitoraggio dei dati e di raccomandazione di film online in tempo reale, in grado di elaborare miliardi di eventi al giorno provenienti da diverse fonti di dati. Tuttavia, l'ETL tradizionale per l'elaborazione batch viene trattato in modo diverso. Questi dati vengono prima letti e poi convertiti in un formato di database prima di essere scritti nel database.
- **Arricchimento dei dati.** Lo streaming Spark arricchisce i dati live con dati statici per consentire un'analisi dei dati più accurata in tempo reale. Ad esempio, gli inserzionisti online possono fornire annunci personalizzati e mirati, basati sulle informazioni relative al comportamento dei clienti.
- **Rilevamento dell'evento trigger.** Lo streaming Spark consente di rilevare e reagire rapidamente a comportamenti insoliti che potrebbero indicare problemi potenzialmente gravi. Ad esempio, gli istituti finanziari utilizzano i trigger per rilevare e bloccare le transazioni fraudolente, mentre gli ospedali li utilizzano per rilevare pericolosi cambiamenti di salute rilevati nei parametri vitali di un paziente.
- **Analisi di sessioni complesse.** Spark Streaming raccoglie eventi quali l'attività dell'utente dopo l'accesso a un sito web o a un'applicazione, che vengono poi raggruppati e analizzati. Ad esempio, Netflix utilizza questa funzionalità per fornire consigli sui film in tempo reale.

Per ulteriori configurazioni di dati in streaming, verifica di Confluent Kafka e test delle prestazioni, vedere"TR-4912: Linee guida sulle best practice per l'archiviazione a livelli Confluent Kafka con NetApp" .

Apprendimento automatico

Il framework integrato Spark consente di eseguire query ripetute sui set di dati utilizzando la libreria di apprendimento automatico (MLlib). MLlib viene utilizzato in ambiti quali il clustering, la classificazione e la riduzione della dimensionalità per alcune comuni funzioni di big data, come l'intelligenza predittiva, la segmentazione dei clienti per scopi di marketing e l'analisi del sentiment. MLlib viene utilizzato nella sicurezza di rete per condurre ispezioni in tempo reale dei pacchetti di dati alla ricerca di indicazioni di attività dannose. Aiuta i fornitori di servizi di sicurezza a conoscere le nuove minacce e a restare un passo avanti agli hacker, proteggendo al contempo i propri clienti in tempo reale.

Apprendimento profondo

TensorFlow è un framework di deep learning molto diffuso e utilizzato in tutto il settore. TensorFlow supporta la formazione distribuita su un cluster CPU o GPU. Questa formazione distribuita consente agli utenti di eseguirla su una grande quantità di dati con molti livelli profondi.

Fino a poco tempo fa, se volevamo utilizzare TensorFlow con Apache Spark, dovevamo eseguire tutti gli ETL necessari per TensorFlow in PySpark e poi scrivere i dati su un archivio intermedio. Tali dati verrebbero poi caricati sul cluster TensorFlow per il processo di addestramento vero e proprio. Questo flusso di lavoro richiedeva all'utente di gestire due cluster diversi, uno per ETL e uno per la formazione distribuita di TensorFlow. L'esecuzione e la manutenzione di più cluster erano solitamente operazioni noiose e dispendiose in termini di tempo.

I DataFrame e gli RDD nelle versioni precedenti di Spark non erano adatti al deep learning perché l'accesso casuale era limitato. In Spark 3.0 con Project Hydrogen, è stato aggiunto il supporto nativo per i framework di deep learning. Questo approccio consente una pianificazione non basata su MapReduce sul cluster Spark.

Analisi interattiva

Apache Spark è sufficientemente veloce da eseguire query esplorative senza campionamento con linguaggi di sviluppo diversi da Spark, tra cui SQL, R e Python. Spark utilizza strumenti di visualizzazione per elaborare dati complessi e visualizzarli in modo interattivo. Spark con streaming strutturato esegue query interattive su dati in tempo reale nell'analisi web, consentendo di eseguire query interattive sulla sessione corrente di un visitatore web.

Sistema di raccomandazione

Nel corso degli anni, i sistemi di raccomandazione hanno apportato enormi cambiamenti alle nostre vite, poiché aziende e consumatori hanno reagito ai radicali cambiamenti nello shopping online, nell'intrattenimento online e in molti altri settori. Questi sistemi rappresentano infatti tra i casi di successo più evidenti dell'intelligenza artificiale nella produzione. In molti casi d'uso pratici, i sistemi di raccomandazione vengono combinati con l'intelligenza artificiale conversazionale o con chatbot interfacciati con un backend NLP per ottenere informazioni rilevanti e produrre inferenze utili.

Oggigiorno molti rivenditori stanno adottando nuovi modelli di business, come l'acquisto online e il ritiro in negozio, il ritiro sul marciapiede, il self-checkout, lo scan-and-go e molto altro. Questi modelli sono diventati importanti durante la pandemia di COVID-19, rendendo gli acquisti più sicuri e comodi per i consumatori. L'intelligenza artificiale è fondamentale per queste tendenze digitali in crescita, che sono influenzate dal comportamento dei consumatori e viceversa. Per soddisfare le crescenti esigenze dei consumatori, ampliare l'esperienza del cliente, migliorare l'efficienza operativa e aumentare i ricavi, NetApp aiuta i suoi clienti aziendali e le aziende a utilizzare algoritmi di apprendimento automatico e profondo per progettare sistemi di

raccomandazione più rapidi e accurati.

Esistono diverse tecniche diffuse per fornire raccomandazioni, tra cui il filtraggio collaborativo, i sistemi basati sui contenuti, il modello di raccomandazione basato sull'apprendimento profondo (DLRM) e le tecniche ibride. In precedenza i clienti utilizzavano PySpark per implementare il filtraggio collaborativo per la creazione di sistemi di raccomandazione. Spark MLlib implementa i minimi quadrati alternati (ALS) per il filtraggio collaborativo, un algoritmo molto diffuso tra le aziende prima dell'avvento del DLRM.

Elaborazione del linguaggio naturale

L'intelligenza artificiale conversazionale, resa possibile dall'elaborazione del linguaggio naturale (NLP), è il ramo dell'intelligenza artificiale che aiuta i computer a comunicare con gli esseri umani. L'NLP è diffuso in ogni settore verticale e in molti casi d'uso, dagli assistenti intelligenti e chatbot alla ricerca Google e al testo predittivo. Secondo un "Gartner" Si prevede che entro il 2022 il 70% delle persone interagirà quotidianamente con piattaforme di intelligenza artificiale conversazionale. Per una conversazione di alta qualità tra un essere umano e una macchina, le risposte devono essere rapide, intelligenti e naturali.

I clienti necessitano di una grande quantità di dati per elaborare e addestrare i loro modelli NLP e di riconoscimento automatico del parlato (ASR). Devono inoltre spostare i dati attraverso l'edge, il core e il cloud e hanno bisogno della potenza necessaria per eseguire inferenze in millisecondi per stabilire una comunicazione naturale con gli esseri umani. NetApp AI e Apache Spark rappresentano una combinazione ideale per elaborazione, archiviazione, elaborazione dati, addestramento di modelli, messa a punto e distribuzione.

L'analisi del sentimento è un campo di studio della PNL in cui vengono estratti sentimenti positivi, negativi o neutri da un testo. L'analisi del sentiment ha una varietà di casi d'uso, dalla determinazione delle prestazioni dei dipendenti del centro di supporto nelle conversazioni con i chiamanti alla fornitura di risposte automatizzate appropriate tramite chatbot. È stato utilizzato anche per prevedere il prezzo delle azioni di un'azienda in base alle interazioni tra i rappresentanti dell'azienda e il pubblico durante le conference call trimestrali sui risultati. Inoltre, l'analisi del sentiment può essere utilizzata per determinare l'opinione di un cliente sui prodotti, sui servizi o sul supporto forniti dal marchio.

Abbiamo usato il "Spark NLP" biblioteca da "John Snow Labs" per caricare pipeline pre-addestrate e modelli di rappresentazioni di encoder bidirezionali da trasformatori (BERT) inclusi "sentimento delle notizie finanziarie" E "FinBERT" , eseguendo la tokenizzazione, il riconoscimento di entità denominate, l'addestramento del modello, l'adattamento e l'analisi del sentiment su larga scala. Spark NLP è l'unica libreria NLP open source in produzione che offre trasformatori all'avanguardia come BERT, ALBERT, ELECTRA, XLNet, DistilBERT, RoBERTa, DeBERTa, XLM-RoBERTa, Longformer, ELMO, Universal Sentence Encoder, Google T5, MarianMT e GPT2. La libreria funziona non solo in Python e R, ma anche nell'ecosistema JVM (Java, Scala e Kotlin) su larga scala, estendendo Apache Spark in modo nativo.

Principali casi d'uso e architetture di intelligenza artificiale, apprendimento automatico e apprendimento automatico (DL)

I principali casi d'uso e la metodologia di intelligenza artificiale, apprendimento automatico e apprendimento digitale possono essere suddivisi nelle seguenti sezioni:

Pipeline Spark NLP e inferenza distribuita TensorFlow

L'elenco seguente contiene le librerie NLP open source più diffuse, adottate dalla comunità della scienza dei dati a diversi livelli di sviluppo:

- "Kit di strumenti per il linguaggio naturale (NLTK)" . Il kit completo per tutte le tecniche di PNL. È stato mantenuto fin dai primi anni del 2000.
- "TextBlob" . Un'API Python di strumenti NLP di facile utilizzo, basata su NLTK e Pattern.
- "Stanford Core NLP" . Servizi e pacchetti NLP in Java sviluppati dallo Stanford NLP Group.
- "Gensim" . Topic Modelling for Humans è nato come una raccolta di script Python per il progetto Czech Digital Mathematics Library.
- "SpaCy" . Flussi di lavoro NLP industriali end-to-end con Python e Cython con accelerazione GPU per trasformatori.
- "Testo veloce" . Una libreria NLP gratuita, leggera e open source per l'apprendimento di incorporamenti di parole e la classificazione di frasi, creata dal laboratorio AI Research (FAIR) di Facebook.

Spark NLP è una soluzione unica e unificata per tutte le attività e i requisiti NLP che consente di realizzare un software NLP scalabile, ad alte prestazioni e ad alta precisione per casi d'uso di produzione reali. Sfrutta l'apprendimento per trasferimento e implementa gli algoritmi e i modelli più all'avanguardia nella ricerca e in tutti i settori. A causa della mancanza di supporto completo da parte di Spark per le librerie di cui sopra, Spark NLP è stato costruito su "Spark ML" per sfruttare il motore di elaborazione dati distribuito in memoria di Spark per uso generico come libreria NLP di livello aziendale per flussi di lavoro di produzione mission-critical. I suoi annotatori utilizzano algoritmi basati su regole, apprendimento automatico e TensorFlow per potenziare le implementazioni di deep learning. Ciò comprende attività NLP comuni, tra cui, a titolo esemplificativo ma non esaustivo, tokenizzazione, lemmatizzazione, stemming, tagging delle parti del discorso, riconoscimento di entità denominate, controllo ortografico e analisi del sentiment.

BERT (Bidirectional Encoder Representations from Transformers) è una tecnica di apprendimento automatico basata sui trasformatori per l'elaborazione del linguaggio naturale. Ha reso popolare il concetto di pre-addestramento e messa a punto. L'architettura del trasformatore in BERT ha avuto origine dalla traduzione automatica, che modella le dipendenze a lungo termine meglio dei modelli linguistici basati sulle reti neurali ricorrenti (RNN). Ha inoltre introdotto il task Masked Language Modelling (MLM), in cui un numero casuale del 15% di tutti i token viene mascherato e il modello li prevede, consentendo una vera bidirezionalità.

L'analisi del sentiment finanziario è complessa a causa del linguaggio specialistico e della mancanza di dati specifici in tale ambito. FinBERT, un modello linguistico basato su BERT preaddestrato, è stato adattato al dominio "Reuters TRC2" , un corpus finanziario, e perfezionato con dati etichettati ("Financial PhraseBank") per la classificazione del sentiment finanziario. I ricercatori hanno estratto 4.500 frasi da articoli di giornale contenenti termini finanziari. Successivamente, 16 esperti e studenti magistrali con formazione in finanza hanno etichettato le frasi come positive, neutre e negative. Abbiamo creato un flusso di lavoro Spark end-to-end per analizzare il sentiment delle trascrizioni delle call sugli utili delle 10 principali società del NASDAQ dal 2016 al 2020 utilizzando FinBERT e altre due pipeline pre-addestrate, "Spiega il documento DL") da Spark NLP.

Il motore di apprendimento profondo alla base di Spark NLP è TensorFlow, una piattaforma end-to-end open source per l'apprendimento automatico che consente una facile creazione di modelli, una solida produzione di ML ovunque e una potente sperimentazione per la ricerca. Pertanto, quando eseguiamo le nostre pipeline in Spark `yarn cluster` modalità, stavamo essenzialmente eseguendo TensorFlow distribuito con parallelizzazione di dati e modelli su un master e più nodi worker, nonché storage collegato alla rete montato sul cluster.

Formazione distribuita Horovod

La convalida principale di Hadoop per le prestazioni correlate a MapReduce viene eseguita con TeraGen, TeraSort, TeraValidate e DFSIO (lettura e scrittura). I risultati della convalida TeraGen e TeraSort sono presentati in "Soluzione NetApp E-Series per Hadoop" e nella sezione "Storage Tiering" per AFF.

In base alle richieste dei clienti, riteniamo che la formazione distribuita con Spark sia uno dei casi d'uso più importanti. In questo documento abbiamo utilizzato il "Hovorod su Spark" per convalidare le prestazioni di Spark con soluzioni NetApp on-premise, cloud-native e cloud ibride utilizzando i controller di archiviazione NetApp All Flash FAS (AFF), Azure NetApp Files e StorageGRID.

Il pacchetto Horovod su Spark fornisce un comodo wrapper attorno a Horovod che semplifica l'esecuzione di carichi di lavoro di formazione distribuiti nei cluster Spark, consentendo un ciclo di progettazione del modello rigoroso in cui l'elaborazione dei dati, la formazione del modello e la valutazione del modello vengono eseguite tutte in Spark, dove risiedono i dati di formazione e inferenza.

Sono disponibili due API per eseguire Horovod su Spark: un'API Estimator di alto livello e un'API Run di basso livello. Sebbene entrambi utilizzino lo stesso meccanismo di base per avviare Horovod sugli esecutori Spark, l'API Estimator astrae l'elaborazione dei dati, il ciclo di addestramento del modello, il checkpointing del modello, la raccolta delle metriche e l'addestramento distribuito. Abbiamo utilizzato Horovod Spark Estimators, TensorFlow e Keras per una preparazione dei dati end-to-end e un flusso di lavoro di formazione distribuito basato su "Vendite del negozio Kaggle Rossmann" concorrenza.

La sceneggiatura `keras_spark_horovod_rossmann_estimator.py` può essere trovato nella sezione"Script Python per ogni caso d'uso principale." Si compone di tre parti:

- La prima parte esegue vari passaggi di pre-elaborazione dei dati su un set iniziale di file CSV forniti da Kaggle e raccolti dalla community. I dati di input vengono separati in un set di addestramento con un `Validation` sottoinsieme e un set di dati di test.
- La seconda parte definisce un modello Keras Deep Neural Network (DNN) con funzione di attivazione sigmoide logaritmica e un ottimizzatore Adam, ed esegue l'addestramento distribuito del modello utilizzando Horovod su Spark.
- La terza parte esegue una previsione sul set di dati di test utilizzando il modello migliore che riduce al minimo l'errore assoluto medio complessivo del set di convalida. Quindi crea un file CSV di output.

Vedi la sezione"Apprendimento automatico" per vari risultati di confronto in fase di esecuzione.

Apprendimento approfondito multi-worker con Keras per la previsione del CTR

Grazie ai recenti progressi nelle piattaforme e nelle applicazioni di apprendimento automatico, molta attenzione è ora rivolta all'apprendimento su larga scala. Il tasso di clic (CTR) è definito come il numero medio di clic per cento impressioni di annunci online (espresso in percentuale). È ampiamente adottato come metrica chiave in vari settori verticali e casi d'uso, tra cui marketing digitale, vendita al dettaglio, e-commerce e fornitori di servizi. Per maggiori dettagli sulle applicazioni di CTR e sui risultati delle prestazioni di formazione distribuita, vedere"Modelli di apprendimento profondo per le prestazioni di previsione del CTR" sezione.

In questo rapporto tecnico abbiamo utilizzato una variante del "Set di dati Criteo Terabyte Click Logs" (vedere TR-4904) per l'apprendimento profondo distribuito multi-worker utilizzando Keras per creare un flusso di lavoro Spark con modelli Deep e Cross Network (DCN), confrontando le sue prestazioni in termini di funzione di errore di perdita del registro con un modello di regressione logistica Spark ML di base. DCN cattura in modo efficiente interazioni di caratteristiche efficaci di gradi limitati, apprende interazioni altamente non lineari, non richiede alcuna progettazione manuale delle caratteristiche o ricerche esaustive e ha un basso costo computazionale.

I dati per i sistemi di raccomandazione su scala web sono per lo più discreti e categoriali, il che determina uno spazio di funzionalità ampio e sparso, che risulta difficile da esplorare. Ciò ha limitato la maggior parte dei sistemi su larga scala a modelli lineari come la regressione logistica. Tuttavia, per fare buone previsioni è fondamentale identificare le caratteristiche frequentemente predittive e allo stesso tempo esplorare le caratteristiche incrociate invisibili o rare. I modelli lineari sono semplici, interpretabili e facili da scalare, ma hanno un potere espressivo limitato.

D’altro canto, è stato dimostrato che le caratteristiche incrociate sono significative nel migliorare l’espressività dei modelli. Purtroppo, spesso è necessario un’ingegneria manuale delle funzionalità o una ricerca esaustiva per identificarle. Generalizzare le interazioni tra caratteristiche invisibili è spesso difficile. Utilizzando una rete neurale incrociata come DCN si evita l’ingegneria delle caratteristiche specifiche per un’attività, applicando esplicitamente l’incrocio delle caratteristiche in modo automatico. La rete incrociata è composta da più strati, in cui il grado più elevato di interazioni è determinato in modo dimostrabile dalla profondità dello strato. Ogni livello produce interazioni di ordine superiore basate su quelle esistenti e mantiene le interazioni dei livelli precedenti.

Una rete neurale profonda (DNN) promette di catturare interazioni molto complesse tra le caratteristiche. Tuttavia, rispetto a DCN, richiede quasi un ordine di grandezza in più di parametri, non è in grado di formare esplicitamente le caratteristiche incrociate e potrebbe non riuscire ad apprendere in modo efficiente alcuni tipi di interazioni delle caratteristiche. La rete incrociata è efficiente in termini di memoria e facile da implementare. L’addestramento congiunto dei componenti cross e DNN cattura in modo efficiente le interazioni delle caratteristiche predittive e garantisce prestazioni all’avanguardia sul set di dati Criteo CTR.

Un modello DCN inizia con uno strato di incorporamento e di impilamento, seguito da una rete incrociata e da una rete profonda in parallelo. A loro volta, seguono uno strato di combinazione finale che unisce gli output delle due reti. I dati di input possono essere un vettore con caratteristiche sparse e dense. In Spark, le librerie contengono il tipo `SparseVector` . È quindi importante che gli utenti distinguano tra i due e facciano attenzione quando chiamano le rispettive funzioni e metodi. Nei sistemi di raccomandazione su scala web come la previsione CTR, gli input sono per lo più caratteristiche categoriali, ad esempio `'country=usa'` . Tali caratteristiche sono spesso codificate come vettori one-hot, ad esempio, `'[0,1,0, …]'` . Codifica one-hot (OHE) con `SparseVector` è utile quando si ha a che fare con set di dati del mondo reale con vocabolari in continua evoluzione e crescita. Abbiamo modificato gli esempi in "DeepCTR" per elaborare vocabolari di grandi dimensioni, creando vettori di incorporamento nello strato di incorporamento e impilamento del nostro DCN.

IL "Set di dati sugli annunci display di Criteo" prevede il tasso di clic degli annunci. Presenta 13 caratteristiche intere e 26 caratteristiche categoriali, in cui ogni categoria ha un’elevata cardinalità. Per questo set di dati, un miglioramento di 0,001 nel logloss è praticamente significativo a causa delle grandi dimensioni dell’input. Un piccolo miglioramento nella precisione delle previsioni per una vasta base di utenti può potenzialmente portare a un notevole aumento del fatturato di un’azienda. Il set di dati contiene 11 GB di registri utente relativi a un periodo di 7 giorni, pari a circa 41 milioni di record. Abbiamo usato Spark `dataFrame.randomSplit()function` per suddividere casualmente i dati per l’addestramento (80%), la convalida incrociata (10%) e il restante 10% per i test.

DCN è stato implementato su TensorFlow con Keras. L’implementazione del processo di addestramento del modello con DCN si basa su quattro componenti principali:

- **Elaborazione e incorporamento dei dati.** Le caratteristiche a valore reale vengono normalizzate applicando una trasformata logaritmica. Per le caratteristiche categoriali, incorporiamo le caratteristiche in vettori densi di dimensione 6×(cardinalità di categoria)1/4. Concatenando tutti gli embedding si ottiene un vettore di dimensione 1026.
- **Ottimizzazione.** Abbiamo applicato l’ottimizzazione stocastica mini-batch con l’ottimizzatore Adam. La dimensione del lotto è stata impostata su 512. La normalizzazione batch è stata applicata alla rete profonda e la norma di clip del gradiente è stata impostata a 100.
- **Regolarizzazione.** Abbiamo utilizzato l’interruzione anticipata, poiché la regolarizzazione o l’abbandono della L2 non si sono rivelati efficaci.
- **Iperparametri.** Riportiamo i risultati basati su una ricerca a griglia sul numero di livelli nascosti, sulla dimensione dei livelli nascosti, sul tasso di apprendimento iniziale e sul numero di livelli incrociati. Il numero di livelli nascosti variava da 2 a 5, con dimensioni dei livelli nascosti che andavano da 32 a 1024. Per DCN, il numero di strati incrociati era compreso tra 1 e 6. Il tasso di apprendimento iniziale è stato

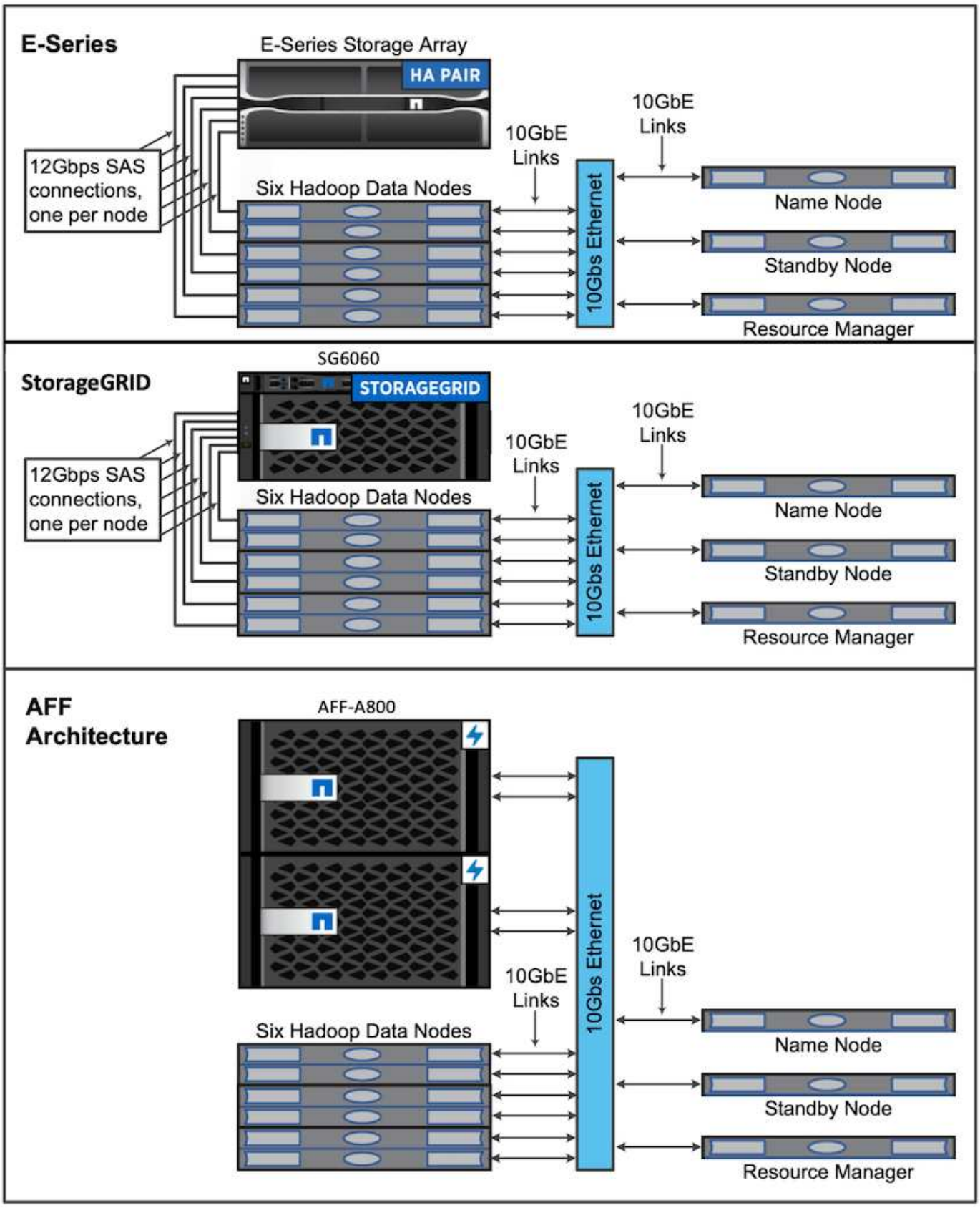
regolato da 0,0001 a 0,001 con incrementi di 0,0001. Tutti gli esperimenti sono stati applicati in anticipo, fermandosi al passo di addestramento 150.000, oltre il quale ha iniziato a verificarsi un overfitting.

Oltre a DCN, abbiamo testato anche altri modelli di deep learning popolari per la previsione del CTR, tra cui "DeepFM" , "AutoInt" , E "DCN v2" .

Architetture utilizzate per la convalida

Per questa convalida, abbiamo utilizzato quattro nodi worker e un nodo master con una coppia AFF-A800 HA. Tutti i membri del cluster erano connessi tramite switch di rete 10GbE.

Per la convalida della soluzione NetApp Spark, abbiamo utilizzato tre diversi controller di storage: E5760, E5724 e AFF-A800. I controller di archiviazione della serie E sono stati collegati a cinque nodi dati con connessioni SAS da 12 Gbps. Il controller di archiviazione AFF HA-pair fornisce volumi NFS esportati tramite connessioni 10GbE ai nodi worker Hadoop. I membri del cluster Hadoop erano connessi tramite connessioni 10GbE nelle soluzioni Hadoop E-Series, AFF e StorageGRID .



Risultati dei test

Abbiamo utilizzato gli script TeraSort e TeraValidate nello strumento di benchmarking

TeraGen per misurare la convalida delle prestazioni di Spark con le configurazioni E5760, E5724 e AFF-A800. Sono stati inoltre testati tre principali casi d'uso: pipeline Spark NLP e formazione distribuita TensorFlow, formazione distribuita Horovod e apprendimento approfondito multi-worker mediante Keras per la previsione CTR con DeepFM.

Per la convalida di E-Series e StorageGRID , abbiamo utilizzato il fattore di replicazione Hadoop 2. Per la convalida AFF , abbiamo utilizzato una sola fonte di dati.

Nella tabella seguente è elencata la configurazione hardware per la convalida delle prestazioni di Spark.

Tipo	Nodi worker Hadoop	Tipo di unità	Unità per nodo	Controllore di archiviazione
SG6060	4	SAS	12	Singola coppia ad alta disponibilità (HA)
E5760	4	SAS	60	Singola coppia HA
E5724	4	SAS	24	Singola coppia HA
AFF800	4	SSD	6	Singola coppia HA

Nella tabella seguente sono elencati i requisiti software.

Software	Versione
RHEL	7,9
Ambiente di runtime OpenJDK	1.8.0
Macchina virtuale server OpenJDK a 64 bit	25,302
Git	2.24.1
GCC/G++	11.2.1
Scintilla	3.2.1
PySpark	3.1.2
SparkNLP	3.4.2
TensorFlow	2.9.0
Keras	2.9.0
Horovod	0.24.3

Analisi del sentiment finanziario

Abbiamo pubblicato"TR-4910: Analisi del sentiment dalle comunicazioni con i clienti con NetApp AI" , in cui è stata costruita una pipeline di intelligenza artificiale conversazionale end-to-end utilizzando "Kit di strumenti NetApp DataOps" , archiviazione AFF e sistema NVIDIA DGX. La pipeline esegue l'elaborazione del segnale audio in batch, il riconoscimento vocale automatico (ASR), l'apprendimento del trasferimento e l'analisi del sentiment sfruttando il DataOps Toolkit, "NVIDIA Riva SDK" , e il "Quadro Tao" . Estendendo il caso d'uso dell'analisi del sentiment al settore dei servizi finanziari, abbiamo creato un flusso di lavoro SparkNLP, caricato tre modelli BERT per varie attività NLP, come il riconoscimento di entità denominate, e ottenuto il sentiment a livello di frase per le call sugli utili trimestrali delle 10 principali aziende del NASDAQ.

Il seguente script `sentiment_analysis_spark. py` utilizza il modello FinBERT per elaborare le trascrizioni in HDFS e produrre conteggi di sentiment positivi, neutri e negativi, come mostrato nella tabella seguente:

```
-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
```

La tabella seguente elenca l'analisi del sentiment a livello di frase, nelle conference call sui risultati finanziari, per le 10 principali società del NASDAQ dal 2016 al 2020.

Conteggi e percentuali del sentimento	**Tutte le 10 aziende**	**AAPL**	**AMD**	**AMZN**	**CSCO**	**GOOGLE**	**INTC**	**MSFT**	**NVDA**
Conteggi positivi	7447	1567	743	290	682	826	824	904	417
Conteggi neutrali	64067	6856	7596	5086	6650	5914	6099	5715	6189
Conteggi negativi	1787	253	213	84	189	97	282	202	89
Conteggi non categorizzati	196	0	0	76	0	0	0	1	0
(conteggi totali)	73497	8676	8552	5536	7521	6837	7205	6822	6695

In termini percentuali, la maggior parte delle frasi pronunciate dai CEO e dai CFO sono basate sui fatti e quindi trasmettono un sentimento neutrale. Durante una conference call sui risultati finanziari, gli analisti pongono domande che possono trasmettere un sentimento positivo o negativo. Vale la pena approfondire l'analisi quantitativa di come il sentiment negativo o positivo influenzi i prezzi delle azioni nello stesso giorno di negoziazione o in quello successivo.

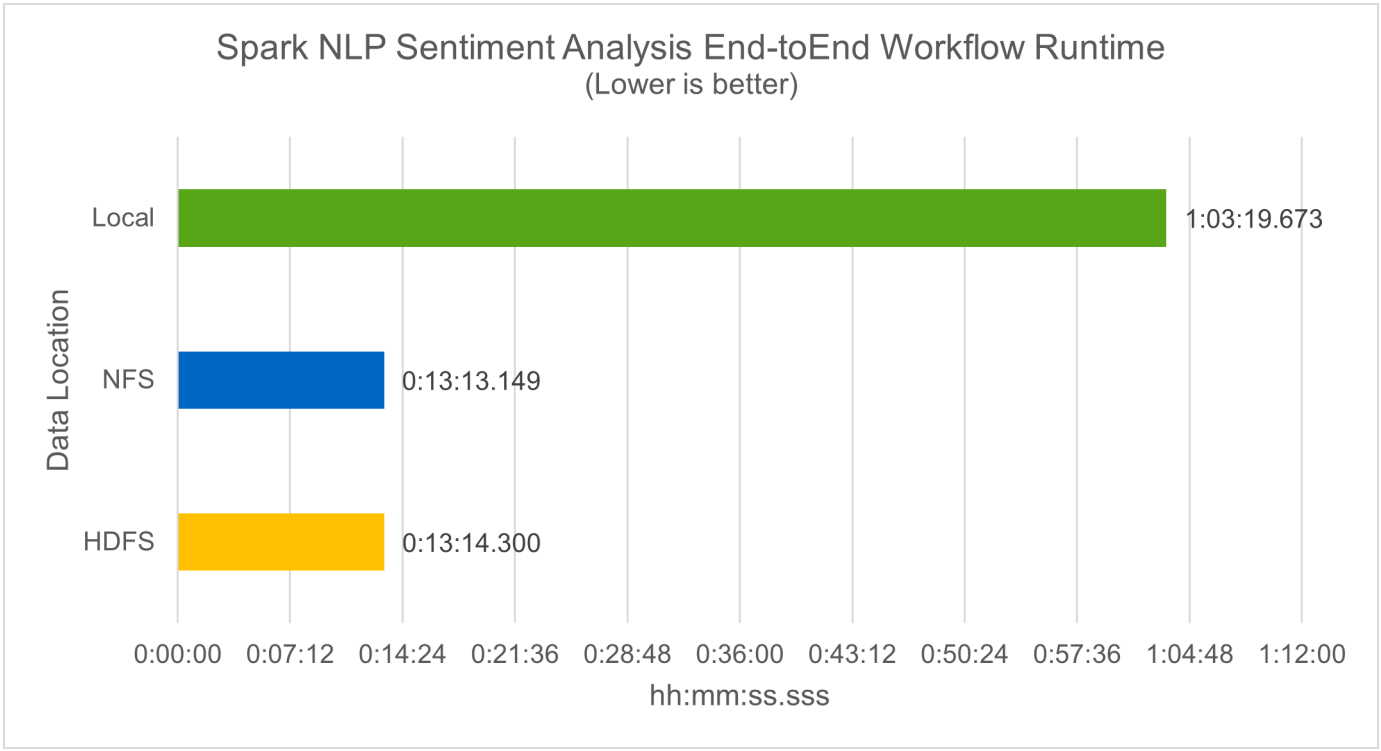
Nella tabella seguente è riportata l'analisi del sentiment a livello di frase per le prime 10 aziende del NASDAQ, espressa in percentuale.

Percentuale di sentimento	Tutte le 10 aziende	AAPL	AMD	AMZN	CSCO	GOOGLE	INTC	MSFT	NVDA
Positivo	10,13%	18,06%	8,69%	5,24%	9,07%	12,08%	11,44%	13,25%	6,23%
Neutro	87,17%	79,02%	88,82%	91,87%	88,42%	86,50%	84,65%	83,77%	92,44%
Negativo	2,43%	2,92%	2,49%	1,52%	2,51%	1,42%	3,91%	2,96%	1,33%
Non categorizzato	0,27%	0%	0%	1,37%	0%	0%	0%	0,01%	0%

In termini di runtime del flusso di lavoro, abbiamo visto un miglioramento significativo di 4,78 volte rispetto `local` modalità a un ambiente distribuito in HDFS e un ulteriore miglioramento dello 0,14% sfruttando 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
```

Come mostra la figura seguente, il parallelismo dei dati e dei modelli ha migliorato la velocità di elaborazione dei dati e di inferenza del modello distribuito TensorFlow. L'ubicazione dei dati in NFS ha prodotto un tempo di esecuzione leggermente migliore perché il collo di bottiglia del flusso di lavoro è il download dei modelli pre-addestrati. Se aumentiamo la dimensione del dataset delle trascrizioni, il vantaggio di NFS diventa più evidente.



Formazione distribuita con prestazioni Horovod

Il seguente comando ha prodotto informazioni di runtime e un file di registro nel nostro cluster Spark utilizzando un singolo `master` nodo con 160 esecutori, ciascuno con un core. La memoria dell'esecutore è stata limitata a 5 GB per evitare errori di memoria insufficiente. Vedi la sezione"Script Python per ogni caso d'uso principale" per maggiori dettagli riguardanti l'elaborazione dei dati, l'addestramento del modello e il calcolo dell'accuratezza del modello in `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
```

Il tempo di esecuzione risultante con dieci epoche di addestramento è stato il seguente:

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

Ci sono voluti più di 43 minuti per elaborare i dati di input, addestrare un modello DNN, calcolare l'accuratezza e produrre i checkpoint TensorFlow e un file CSV per i risultati delle previsioni. Abbiamo limitato il numero di epoche di addestramento a 10, che nella pratica è spesso impostato su 100 per garantire una precisione soddisfacente del modello. Il tempo di addestramento in genere varia in modo lineare con il numero di epoche.

Successivamente abbiamo utilizzato i quattro nodi worker disponibili nel cluster ed eseguito lo stesso script in `yarn` modalità con dati in 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
```

Il tempo di esecuzione risultante è stato migliorato come segue:

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

Con il modello di Horovod e il parallelismo dei dati in Spark, abbiamo visto un'accelerazione del runtime di 5,29 volte `yarn` contro `local` modalità con dieci epoche di allenamento. Ciò è mostrato nella figura seguente con le legende `HDFS` E `Local` . L'addestramento del modello DNN TensorFlow sottostante può essere ulteriormente accelerato con le GPU, se disponibili. Abbiamo intenzione di condurre questi test e di pubblicare i risultati in un futuro rapporto tecnico.

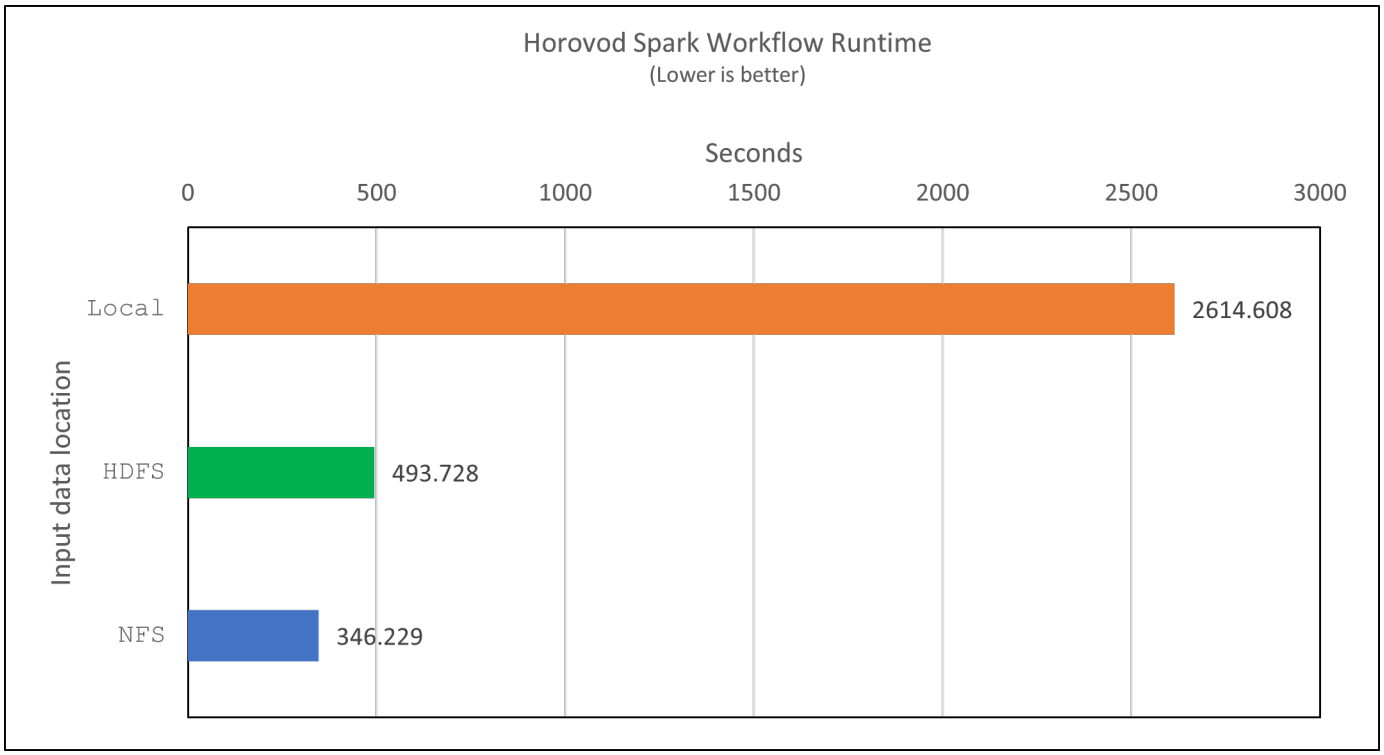
Il nostro test successivo ha confrontato i tempi di esecuzione con dati di input residenti in NFS rispetto a HDFS. Il volume NFS sull'AFF AFF A800 è stato montato su `/sparkdemo/horovod` attraverso i cinque nodi (un master, quattro worker) nel nostro cluster Spark. Abbiamo eseguito un comando simile a quello dei test precedenti, con il `--data- dir` parametro che ora punta al montaggio 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
```

Il runtime risultante con NFS era il seguente:

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

Si è verificato un ulteriore aumento di velocità di 1,43 volte, come mostrato nella figura seguente. Pertanto, con uno storage all-flash NetApp connesso al proprio cluster, i clienti possono usufruire dei vantaggi di un rapido trasferimento e distribuzione dei dati per i flussi di lavoro di Horovod Spark, ottenendo una velocità 7,55 volte superiore rispetto all'esecuzione su un singolo nodo.



Modelli di apprendimento profondo per le prestazioni di previsione del CTR

Per i sistemi di raccomandazione progettati per massimizzare il CTR, è necessario apprendere le interazioni sofisticate delle funzionalità alla base dei comportamenti degli utenti, che possono essere calcolate matematicamente dal livello più basso a quello più alto. Per un buon modello di deep learning, sia le interazioni tra le caratteristiche di ordine basso che quelle di ordine alto dovrebbero essere ugualmente importanti, senza sbilanciarsi verso l'una o l'altra. Deep Factorization Machine (DeepFM), una rete neurale basata su macchine di fattorizzazione, combina macchine di fattorizzazione per la raccomandazione e apprendimento profondo per l'apprendimento delle caratteristiche in una nuova architettura di rete neurale.

Sebbene le macchine di fattorizzazione convenzionali modellino le interazioni delle caratteristiche a coppie come un prodotto interno di vettori latenti tra caratteristiche e possano teoricamente catturare informazioni di ordine elevato, in pratica, gli esperti di apprendimento automatico solitamente utilizzano solo interazioni delle caratteristiche di secondo ordine a causa dell'elevata complessità di calcolo e di archiviazione. Varianti di reti neurali profonde come quelle di Google "Modelli larghi e profondi" d'altro canto, apprende interazioni sofisticate tra le caratteristiche in una struttura di rete ibrida combinando un modello lineare ampio e un modello profondo.

Questo modello Wide & Deep prevede due input, uno per il modello wide sottostante e l'altro per quello deep; quest'ultima parte richiede ancora un'ingegneria delle funzionalità da parte di esperti e rende quindi la tecnica meno generalizzabile ad altri domini. A differenza del modello Wide & Deep, DeepFM può essere addestrato in modo efficiente con feature grezze senza alcuna progettazione delle feature, poiché la sua parte ampia e quella profonda condividono lo stesso input e lo stesso vettore di incorporamento.

Abbiamo prima elaborato il Criteo `train.txt` (11 GB) file in un file CSV denominato `ctr_train.csv` memorizzato in un mount NFS `/sparkdemo/tr-4570-data` usando `run_classification_criteo_spark.py` dalla sezione"Script Python per ogni caso d'uso principale." All'interno di questo script, la funzione `process_input_file` esegue diversi metodi stringa per rimuovere le tabulazioni e inserirle `','` come delimitatore e `'\n'` come nuova riga. Nota che devi elaborare solo l'originale `train.txt` una volta, in modo che il blocco di codice venga visualizzato come commento.

Per i seguenti test di diversi modelli DL, abbiamo utilizzato `ctr_train.csv` come file di input. Nelle successive esecuzioni di test, il file CSV di input è stato letto in uno Spark DataFrame con schema contenente un campo di `'label'` , caratteristiche dense intere `['I1', 'I2', 'I3', …, 'I13']` e caratteristiche sparse `['C1', 'C2', 'C3', …, 'C26']` . Il seguente `spark-submit` il comando accetta un CSV di input, addestra i modelli DeepFM con una suddivisione del 20% per la convalida incrociata e sceglie il modello migliore dopo dieci epoche di addestramento per calcolare l'accuratezza della previsione sul set di 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
```

Si noti che poiché il file di dati `ctr_train.csv` è superiore a 11 GB, è necessario impostare un numero sufficiente `spark.driver.maxResultSize` maggiore della dimensione del set di dati per evitare errori.

```
 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()
```

In quanto sopra `SparkSession.builder` configurazione che abbiamo anche abilitato "Freccia Apache" , che converte uno Spark DataFrame in un Pandas DataFrame con `df.toPandas()` metodo.

```
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.
```

Dopo la suddivisione casuale, ci sono oltre 36 milioni di righe nel set di dati di addestramento e 9 milioni di campioni nel set di test:

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

Poiché questo report tecnico è incentrato sui test della CPU senza l'utilizzo di GPU, è fondamentale compilare TensorFlow con flag di compilazione appropriati. Questo passaggio evita di richiamare librerie accelerate dalla GPU e sfrutta appieno le Advanced Vector Extensions (AVX) e le istruzioni AVX2 di TensorFlow. Queste funzionalità sono progettate per calcoli algebrici lineari come l'addizione vettorizzata, le moltiplicazioni di matrici all'interno di un addestramento DNN feed-forward o back-propagation. L'istruzione Fused Multiply Add (FMA) disponibile con AVX2 che utilizza registri in virgola mobile (FP) a 256 bit è ideale per codice intero e tipi di dati, con un conseguente aumento della velocità fino a 2 volte. Per il codice FP e i tipi di dati, AVX2 raggiunge un aumento di velocità dell'8% rispetto ad 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.
```

Per creare TensorFlow dalla sorgente, NetApp consiglia di utilizzare "Bazel" . Per il nostro ambiente, abbiamo eseguito i seguenti comandi nel prompt della shell per installare `dnf` , `dnf-plugins` e Bazel.

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

È necessario abilitare GCC 5 o versioni successive per utilizzare le funzionalità C++17 durante il processo di compilazione, fornite da RHEL con Software Collections Library (SCL). I seguenti comandi installano `devtoolset` e GCC 11.2.1 sul nostro 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
```

Nota che gli ultimi due comandi abilitano `devtoolset-11` , che utilizza `/opt/rh/devtoolset-11/root/usr/bin/gcc` (GCC 11.2.1). Inoltre, assicurati che il tuo `git` la versione è successiva alla 1.8.3 (inclusa in RHEL 7.9). Fare riferimento a questo "articolo" per l'aggiornamento `git` a 2.24.1.

Supponiamo che tu abbia già clonato l'ultimo repository master di TensorFlow. Quindi crea un `workspace` directory con un `WORKSPACE` file per compilare TensorFlow dal codice sorgente con AVX, AVX2 e FMA. Esegui il `configure` file e specificare la posizione corretta del binario Python. "CUDA" è disabilitato per i nostri test perché non abbiamo utilizzato una GPU. UN `.bazelrc` il file viene generato in base alle tue impostazioni. Inoltre, abbiamo modificato il file e impostato `build --define=no_hdfs_support=false` per abilitare il supporto HDFS. Fare riferimento a `.bazelrc` nella sezione"Script Python per ogni caso d'uso principale," per un elenco completo delle impostazioni e dei flag.

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

Dopo aver creato TensorFlow con i flag corretti, esegui lo script seguente per elaborare il set di dati Criteo Display Ads, addestrare un modello DeepFM e calcolare l'area sotto la curva ROC (AUC) dai punteggi di previsione.

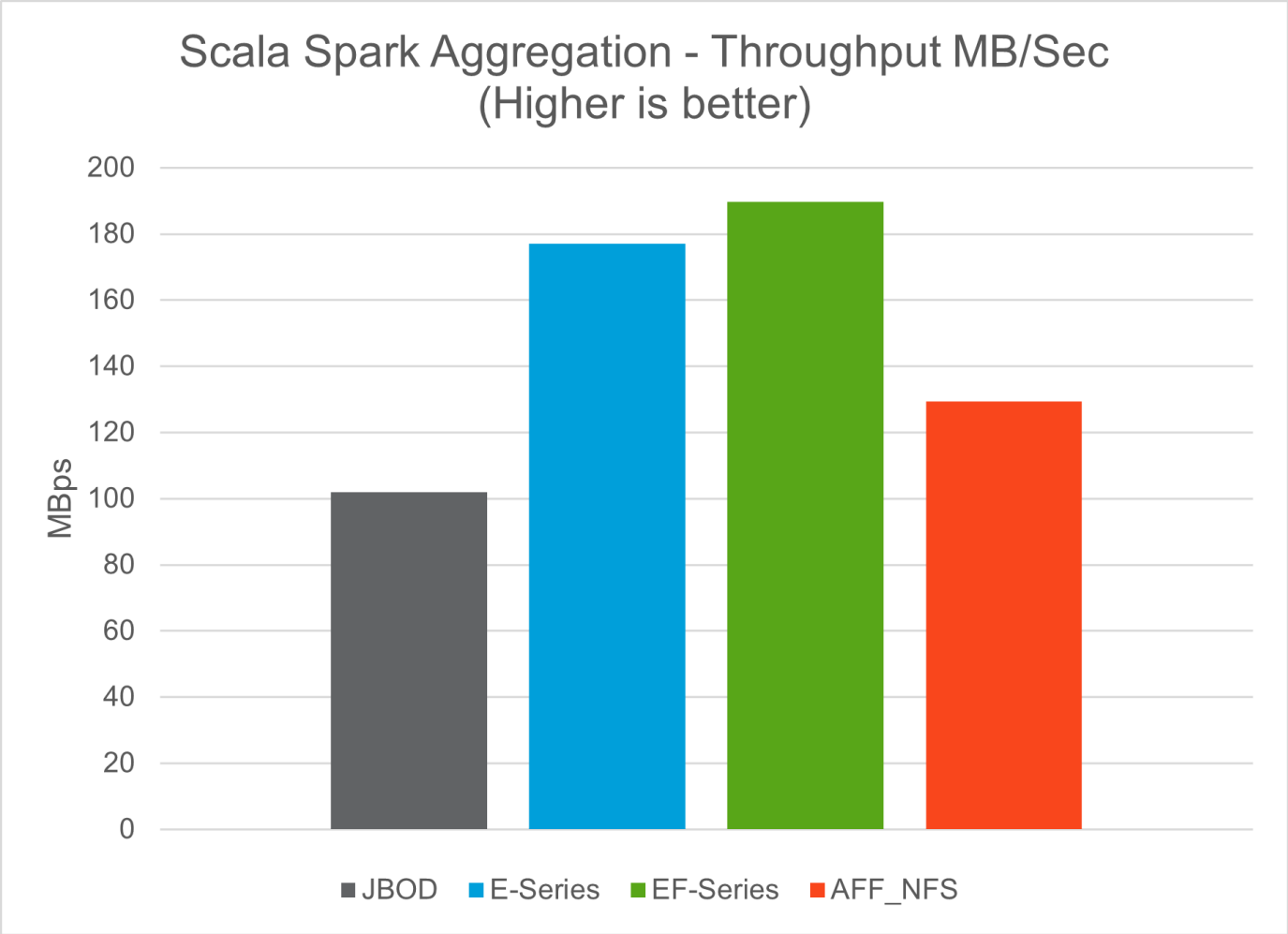
```
(base) [root@n138 examples]# ~/anaconda3/bin/spark-submit
--master yarn
--executor-memory 15g
--executor-cores 1
--num-executors 160
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py
--data-dir file:///sparkdemo/tr-4570-data
> . /run_classification_criteo_spark_nfs.log 2>&1
```

Dopo dieci epoche di addestramento, abbiamo ottenuto il punteggio AUC sul set di dati di 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
```

In modo simile ai casi d’uso precedenti, abbiamo confrontato il runtime del flusso di lavoro Spark con dati

residenti in posizioni diverse. La figura seguente mostra un confronto della previsione CTR del deep learning per un runtime di flussi di lavoro Spark.



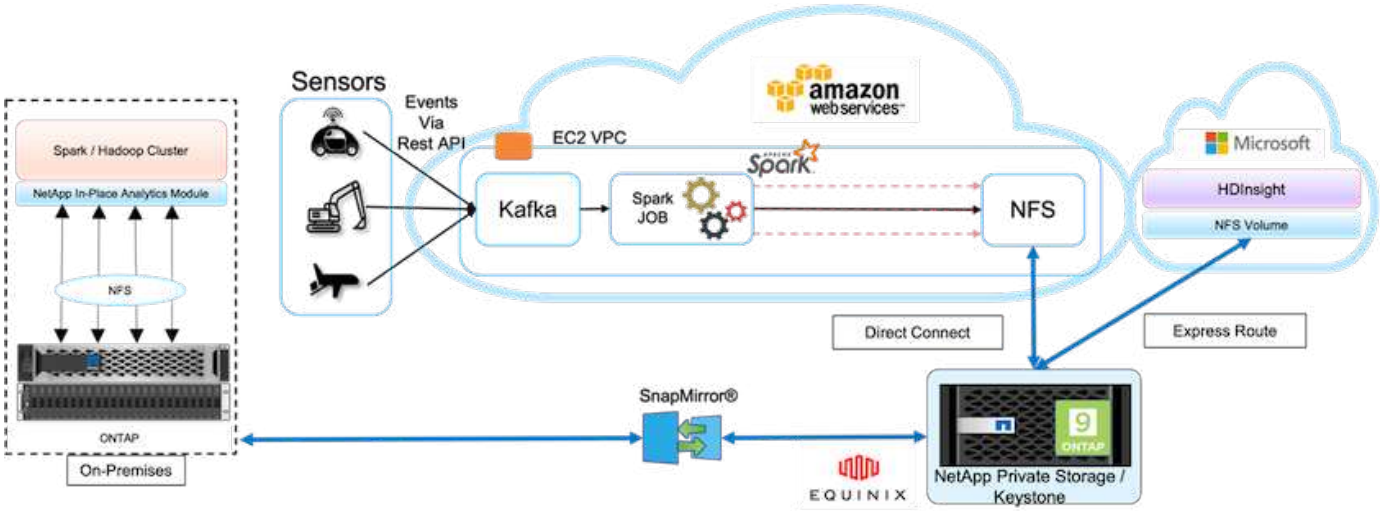
Soluzione cloud ibrida

Un moderno data center aziendale è un cloud ibrido che collega più ambienti infrastrutturali distribuiti tramite un piano di gestione dati continuo con un modello operativo coerente, in sede e/o in più cloud pubblici. Per sfruttare al meglio un cloud ibrido, è necessario essere in grado di spostare i dati senza problemi tra gli ambienti on-premise e multi-cloud, senza dover effettuare conversioni di dati o refactoring delle applicazioni.

I clienti hanno dichiarato di iniziare il loro percorso verso il cloud ibrido spostando lo storage secondario sul cloud per casi d'uso quali la protezione dei dati oppure spostando sul cloud carichi di lavoro meno critici per l'azienda, come lo sviluppo di applicazioni e DevOps. Passano poi a carichi di lavoro più critici. Tra i carichi di lavoro più diffusi nel cloud ibrido rientrano l'hosting di contenuti e web, lo sviluppo di applicazioni e DevOps, i database, l'analisi e le app containerizzate. La complessità, i costi e i rischi dei progetti di intelligenza artificiale aziendali hanno storicamente ostacolato l'adozione dell'intelligenza artificiale dalla fase sperimentale a quella produttiva.

Con una soluzione cloud ibrida NetApp , i clienti beneficiano di strumenti integrati di sicurezza, governance dei dati e conformità con un unico pannello di controllo per la gestione dei dati e dei flussi di lavoro in ambienti

distribuiti, ottimizzando al contempo il costo totale di proprietà in base al loro consumo. La figura seguente è un esempio di soluzione di un partner di servizi cloud incaricato di fornire connettività multi-cloud per i dati di analisi dei big data dei clienti.



In questo scenario, i dati IoT ricevuti in AWS da diverse fonti vengono archiviati in una posizione centrale in NetApp Private Storage (NPS). Lo storage NPS è connesso ai cluster Spark o Hadoop situati in AWS e Azure, consentendo alle applicazioni di analisi dei big data di essere eseguite in più cloud e di accedere agli stessi dati. I principali requisiti e sfide per questo caso d'uso includono quanto segue:

- I clienti desiderano eseguire attività di analisi sugli stessi dati utilizzando più cloud.
- I dati devono essere ricevuti da diverse fonti, ad esempio ambienti on-premise e cloud, tramite diversi sensori e hub.
- La soluzione deve essere efficiente e conveniente.
- La sfida principale è quella di creare una soluzione efficiente e conveniente che fornisca servizi di analisi ibridi tra diversi ambienti on-premise e cloud.

La nostra soluzione di protezione dei dati e connettività multicloud risolve il problema di dover disporre di applicazioni di analisi cloud su più hyperscaler. Come mostrato nella figura sopra, i dati provenienti dai sensori vengono trasmessi in streaming e acquisiti nel cluster AWS Spark tramite Kafka. I dati vengono archiviati in una condivisione NFS residente in NPS, che si trova all'esterno del provider cloud all'interno di un data center Equinix.

Poiché NetApp NPS è connesso ad Amazon AWS e Microsoft Azure rispettivamente tramite connessioni Direct Connect ed Express Route, i clienti possono sfruttare il modulo In-Place Analytics per accedere ai dati dai cluster di analisi di Amazon e AWS. Di conseguenza, poiché sia lo storage locale che quello NPS eseguono il software ONTAP , "SnapMirror" può rispecchiare i dati NPS nel cluster locale, fornendo analisi cloud ibride su cloud locali e multipli.

Per ottenere le migliori prestazioni, NetApp consiglia in genere di utilizzare più interfacce di rete e connessioni dirette o percorsi rapidi per accedere ai dati dalle istanze cloud. Disponiamo di altre soluzioni di spostamento dati, tra cui "XCP" E "BlueXP Copia e Sincronizza" per aiutare i clienti a creare cluster Spark cloud ibridi, sicuri, convenienti e compatibili con le applicazioni.

Script Python per ogni caso d'uso principale

I tre script Python seguenti corrispondono ai tre principali casi d'uso testati. Il primo è

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}")
```

Il secondo copione è `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()
```

Il terzo copione è `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))
```

Conclusione

In questo documento, analizziamo l'architettura di Apache Spark, i casi d'uso dei clienti e il portfolio di storage NetApp in relazione a big data, analisi moderne, intelligenza artificiale, apprendimento automatico e apprendimento automatico (ML) e apprendimento automatico (DL). Nei nostri test di convalida delle prestazioni basati su strumenti di benchmarking standard del settore e sulla domanda dei clienti, le soluzioni NetApp Spark hanno dimostrato prestazioni superiori rispetto ai sistemi Hadoop nativi. Una combinazione dei casi d'uso dei clienti e dei risultati delle prestazioni presentati in questo report può aiutarti a scegliere la soluzione Spark più adatta alla tua implementazione.

Dove trovare ulteriori informazioni

In questo TR sono stati utilizzati i seguenti riferimenti:

- Architettura e componenti di Apache Spark

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

- Casi d'uso di Apache Spark

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

- Spark NLP

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

- BERT

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

- Rete profonda e incrociata per le previsioni sui clic sugli annunci

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

- FlexGroup

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

- Streaming ETL

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

- Soluzioni NetApp E-Series per Hadoop

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

- Soluzioni di analisi dei dati moderni NetApp

  "Soluzioni di analisi dei dati"

- 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

- BlueXP Copia e Sincronizza

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

- Kit di strumenti DataOps

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

Informazioni sul copyright

Informazioni sul marchio commerciale