NetApp Storage-Lösungen für Apache Spark

NetApp Solutions

NetApp
April 24, 2024

Inhalt

NetApp Storage-Lösungen für Apache Spark

TR-4570: NetApp Storage-Lösungen für Apache Spark: Architektur, Anwendungsfälle und Performance-Ergebnisse

Rick Huang, Karthikeyan Nagalingam, NetApp

In diesem Dokument geht es um die Apache Spark-Architektur, Anwendungsfälle von Kunden und das NetApp Storage-Portfolio in Bezug auf Big Data Analytics und künstliche Intelligenz (KI). Er enthält zudem diverse Testergebnisse mit branchenüblichen Tools KI, Machine Learning (ML) und Deep Learning (DL) gegen ein typisches Hadoop System, sodass Sie die entsprechende Spark-Lösung wählen können. Zunächst benötigen Sie eine Spark-Architektur, geeignete Komponenten und zwei Implementierungsmodi (Cluster und Client).

Dieses Dokument enthält auch Anwendungsfälle, um Konfigurationsprobleme zu lösen, und es enthält eine Übersicht über das NetApp Storage-Portfolio, das für Big Data Analytics und KI, ML und DL mit Spark relevant ist. Abschließend werden die Testergebnisse aus Spark-spezifischen Anwendungsfällen und dem NetApp Spark-Lösungsportfolio erstellt.

Herausforderungen für Kunden

Dieser Abschnitt befasst sich mit den Herausforderungen von Kunden mit Big Data-Analysen und KI/ML/DL in wachsenden Datenbranchen wie Einzelhandel, digitales Marketing, Banking, diskrete Fertigung, Prozessfertigung Öffentlicher Sektor und Professional Services.

Unvorhersehbare Performance

Herkömmliche Hadoop Implementierungen nutzen in der Regel Standard-Hardware. Zur Verbesserung der Performance müssen Netzwerk, Betriebssystem, Hadoop Cluster, Ecosystem-Komponenten wie Spark und Hardware abgestimmt werden. Selbst bei einer Feinanpassung aller Ebenen kann das gewünschte Performance-Niveau nur schwer erreicht werden, da Hadoop auf Standard-Hardware ausgeführt wird, die nicht für eine hohe Performance in der Umgebung konzipiert wurde.

Medien- und Node-Ausfälle

Selbst unter normalen Bedingungen ist Standard-Hardware fehleranfällig. Wenn eine Festplatte in einem Daten-Node ausfällt, ist dieser Node standardmäßig von Hadoop Master als fehlerhaft erachtet. Anschließend werden bestimmte Daten von diesem Node über das Netzwerk von Replikaten auf einen gesunden Node kopiert. Dieser Prozess verlangsamt die Netzwerkpakete für alle Hadoop Jobs. Anschließend muss das Cluster die Daten wieder kopieren und die über- replizierten Daten entfernen, wenn der ungesunde Node in einen ordnungsgemäßen Zustand zurückkehrt.

Hadoop Anbieterbindung

Hadoop Distributoren verfügen über eigene Hadoop Distributionen, welche sich dem Kunden an diese Distributionen anschloss. Viele Kunden benötigen jedoch Unterstützung für in-Memory-Analysen, die den Kunden nicht an bestimmte Hadoop Distributionen binden. Sie brauchen die Freiheit, Distributionen zu ändern und trotzdem ihre Analysen mit ihnen zu bringen.

Fehlende Unterstützung für mehr als eine Sprache

Kunden benötigen für ihre Aufgaben oft Unterstützung für mehrere Sprachen neben MapReduce Java-Programmen. Optionen wie SQL und Skripte bieten mehr Flexibilität, um Antworten zu erhalten, mehr Optionen für die Organisation und den Abruf von Daten und eine schnellere Möglichkeit zum Verschieben von Daten in ein Analyse-Framework.

Nutzschwierigkeiten

Seit einiger Zeit haben sich die Leute darüber beschwert, dass Hadoop nur schwer zu bedienen ist. Auch wenn Hadoop mit jeder neuen Version einfacher und leistungsstärker geworden ist, bleibt diese Kritik bestehen. Hadoop erfordert, dass Sie Java- und MapReduce-Programmiermuster kennen. Dies ist eine Herausforderung für Datenbankadministratoren und Mitarbeiter mit klassischen Scripting-Kenntnissen.

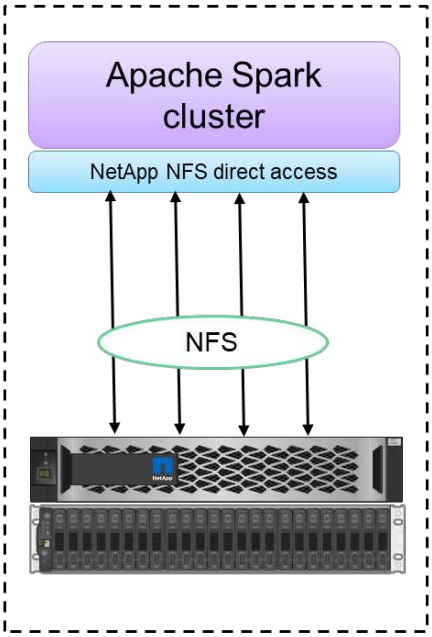
Komplizierte Frameworks und Tools

KI-Teams von Unternehmen stehen vor mehreren Herausforderungen. Selbst wenn Experten im Bereich Data Science wissen, Tools und Frameworks für verschiedene Implementierungökosysteme und -Applikationen nicht einfach zwischen Lösungen übersetzen können. Eine Data-Science-Plattform sollte sich nahtlos in die entsprechenden Big-Data-Plattformen integrieren lassen, die auf Spark basieren, mit einfacher Datenverschiebung, wiederverwendbaren Modellen, sofort verwendbarem Code und Tools, die Best Practices für Prototyping, Validierung, Versionierung, Freigabe, Wiederverwendung, Und schnelle Implementierung von Modellen in der Produktion.

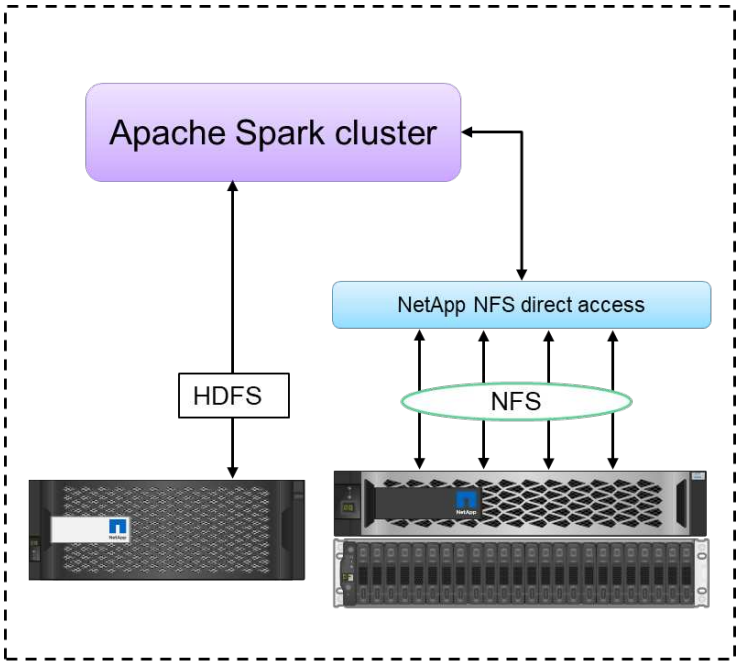
Warum NetApp?

Mit NetApp können Sie Ihre Spark-Erfahrung auf folgende Weise verbessern:

- Durch den direkten Zugriff mit NetApp NFS (in der Abbildung unten dargestellt) können Kunden Big-Data-Analytics-Jobs auf ihren vorhandenen oder neuen NFSv3- oder NFSv4-Daten ausführen, ohne die Daten zu verschieben oder zu kopieren. Es verhindert mehrere Datenkopien und macht die Synchronisierung der Daten mit einer Quelle überflüssig.
- Effizienterer Storage und weniger Server-Replizierung. Beispielsweise erfordert die NetApp E-Series Hadoop Lösung zwei statt drei Datenreplikate. Zur FAS Hadoop Lösung ist eine Datenquelle erforderlich, jedoch keine Replizierung oder Kopie der Daten. NetApp Storage-Lösungen produzieren zudem weniger Datenverkehr zwischen Servern.
- Bessere Hadoop Jobs und Clusterverhalten bei Laufwerks- und Node-Ausfällen
- Bessere Performance bei der Datenaufnahme:



Configuration 1: NFS as primary storage



Configuration 2: HDFS and NFS in single Spark cluster

So muss zum Beispiel im Finanz- und Gesundheitswesen das Verschieben von Daten von einem Ort zum anderen den gesetzlichen Verpflichtungen entsprechen, was keine einfache Aufgabe ist. In diesem Szenario analysiert NetApp NFS Direct Access die Finanz- und Gesundheitsdaten vom ursprünglichen Standort aus. Ein weiterer entscheidender Vorteil besteht darin, dass der NetApp NFS Direct Access die Sicherung von Hadoop Daten durch native Hadoop Befehle vereinfacht und Workflows zur Datensicherung mit dem umfassenden Datenmanagement-Portfolio von NetApp ermöglicht.

NetApp NFS Direct Access bietet zwei Arten von Implementierungsoptionen für Hadoop/Spark Cluster:

- Hadoop oder Spark-Cluster verwenden standardmäßig das Hadoop Distributed File System (HDFS) für die Datenspeicherung sowie das Standard-Filesystem. Der direkte NetApp NFS-Zugriff ersetzt das Standard-HDFS durch NFS-Storage als Standarddateisystem und ermöglicht so direkte Analysen für NFS-Daten.
- Bei einer anderen Implementierungsoption unterstützt der direkte NetApp NFS-Zugriff die Konfiguration von NFS als zusätzlichen Storage zusammen mit HDFS in einem einzelnen Hadoop oder Spark-Cluster. In diesem Fall kann der Kunde Daten über NFS-Exporte teilen und gemeinsam mit HDFS-Daten vom selben Cluster aus darauf zugreifen.

Zu den wichtigsten Vorteilen des direkten NetApp NFS-Zugriffs gehören:

- Analyse der Daten vom aktuellen Standort, was das verlagern Zeit- und Performance-verschlingt, Analysedaten in eine Hadoop Infrastruktur wie HDFS zu verschieben
- Reduzierung der Anzahl der Replikate von drei auf eins.
- Benutzer können Computing und Storage entkoppeln, um sie unabhängig voneinander zu skalieren.
- Datensicherung der Enterprise-Klasse durch Nutzung der umfassenden Datenmanagementfunktionen von ONTAP
- Zertifizierung mit der Hortonworks Datenplattform.
- Hybrid-Datenanalyselösungen:
- Kürzere Backup-Zeiten durch Nutzung von dynamischen Multithread-Kapazitäten

Siehe "TR-4657: NetApp Hybrid-Cloud-Datenlösungen – Spark und Hadoop auf Basis von Kundenanwendungsfällen" Für Backups von Hadoop Daten, Backup und Disaster Recovery von der Cloud bis hin zum On-Premises-System; DevTest für vorhandene Hadoop Daten, Datensicherung und Multi-Cloud-Konnektivität sowie beschleunigte Analyse-Workloads

In den folgenden Abschnitten werden Storage-Funktionen beschrieben, die für Spark Kunden wichtig sind.

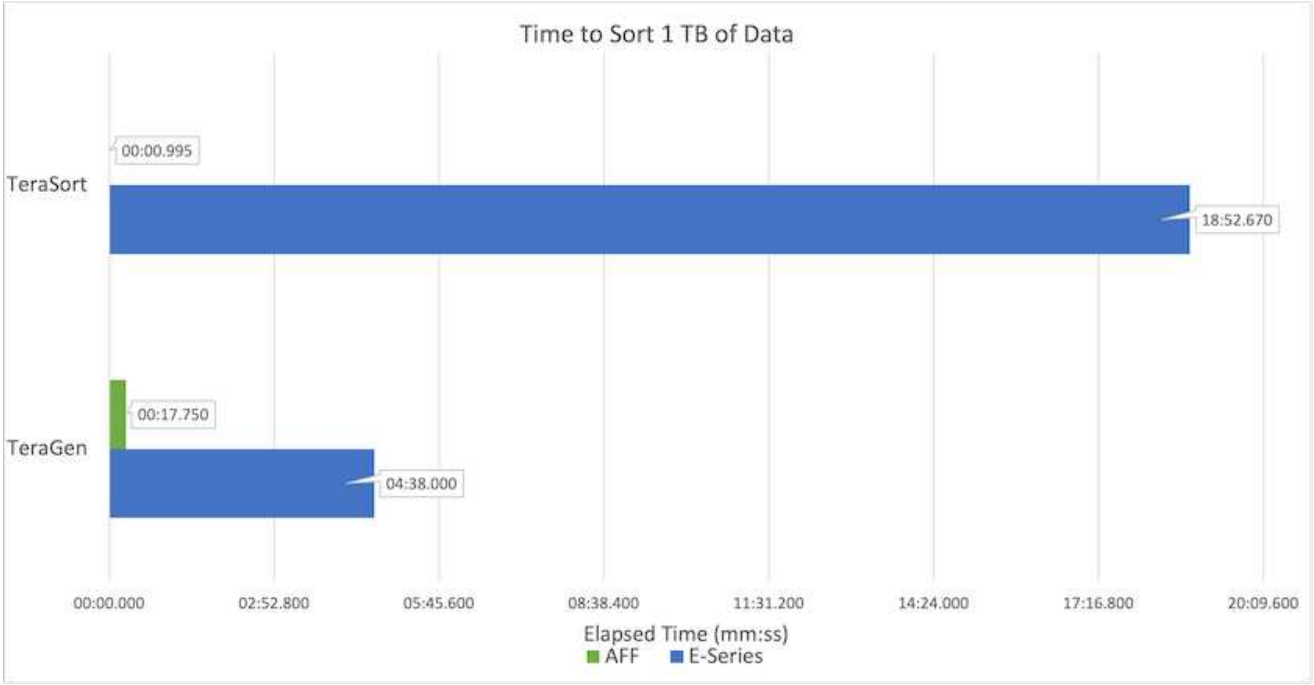
Storage Tiering

Mit Hadoop Storage Tiering können Sie Dateien mit unterschiedlichen Storage-Typen gemäß einer Storage Policy speichern. Storage-Typen sind enthalten `hot`, `cold`, `warm`, `all_ssd`, `one_ssd`, und `lazy_persist`.

<<<<<<< HEAD Wir haben eine Validierung des Hadoop Storage Tiering auf einem NetApp AFF Storage Controller und einem E-Series Storage Controller mit SSD- und SAS-Laufwerken mit unterschiedlichen Storage-Richtlinien durchgeführt. Das Spark-Cluster mit AFF A800 verfügt über vier Computing-Worker-Nodes, während das Cluster mit E-Series acht Nodes hat. Hauptsächlich wurde ein Vergleich der Performance von Solid State-Laufwerken (SSDs) und Festplatten (HDDs) durchgeführt.

Wir haben die Validierung des Hadoop Storage Tiering auf einem NetApp AFF Storage Controller und einem E-Series Storage Controller mit SSD- und SAS-Laufwerken mit unterschiedlichen Storage-Richtlinien durchgeführt. Das Spark-Cluster mit AFF A800 verfügt über vier Computing-Worker-Nodes, während das Cluster mit E-Series acht Nodes hat. Wir haben dies in erster Linie getan, um die Performance von Solid-State-Laufwerken mit Festplatten zu vergleichen. >>>>>>> A51c9ddf73ca69e1120ce05edc7b09607b96eae

Die folgende Abbildung zeigt die Performance der NetApp Lösungen für eine Hadoop SSD.



- In der NL-SAS-Basiskonfiguration wurden acht Computing-Nodes und 96 NL-SAS-Laufwerke verwendet. Durch diese Konfiguration wurden 1 TB Daten in 4 Minuten und 38 Sekunden erzeugt. Siehe "TR-3969 NetApp E-Series Lösung für Hadoop" Finden Sie Details zur Cluster- und Storage-Konfiguration.
- Mit TeraGen generierte die SSD-Konfiguration 1 TB an Daten 15,66-mal schneller als die NL-SAS-Konfiguration. Darüber hinaus verwendete die SSD-Konfiguration die Hälfte der Computing-Nodes und die Hälfte der Festplattenlaufwerke (insgesamt 24 SSD-Laufwerke). Basierend auf der Abschlusszeit war der Job fast doppelt so schnell wie die NL-SAS-Konfiguration.
- Mit TeraSort sortiert die SSD-Konfiguration 1 TB an Daten 1138.36-mal schneller als die NL-SAS-Konfiguration. Darüber hinaus verwendete die SSD-Konfiguration die Hälfte der Computing-Nodes und die Hälfte der Festplattenlaufwerke (insgesamt 24 SSD-Laufwerke). Daher war es pro Laufwerk ca. dreimal schneller als die NL-SAS-Konfiguration. <<<<<<< KOPF
- Die Schlussfolgerung lautet: Der Wechsel von rotierenden Festplatten zu All-Flash-Systemen verbessert die Performance. Die Anzahl der Computing-Nodes war nicht der Engpass. Mit dem All-Flash-Storage von NetApp lässt sich die Runtime Performance gut skalieren.
- Mit NFS entsprach der Summe der Daten, die je nach Workload die Anzahl der Computing-Nodes reduzieren konnte. Die Apache Spark-Cluster-Benutzer müssen Daten nicht manuell neu verteilen, wenn sich die Anzahl der Computing-Nodes ändert.

- Zusammenfassend lässt sich sagen, dass die Umstellung von rotierenden Festplatten auf All-Flash-Systeme die Performance steigert. Die Anzahl der Computing-Nodes war nicht der Engpass. Mit NetApp

All-Flash-Storage lässt sich die Runtime Performance gut skalieren.

- Mit NFS waren die Daten in funktioneller Hinsicht gleichbedeutend mit den gemeinsamen Pools, wodurch sich die Anzahl der Computing-Nodes je nach Workload reduzieren ließ. Apache Spark-Cluster-Benutzer müssen die Daten nicht manuell neu verteilen, wenn sie die Anzahl der Computing-Nodes ändern. >>>>>>> A51c9ddf73ca69e1120ce05edc7b09607b96eae

Performance-Skalierung: Horizontale Skalierung

Wenn von einem Hadoop Cluster in einer AFF Lösung mehr Rechenleistung benötigt wird, können Daten-Nodes mit einer entsprechenden Anzahl Storage Controller hinzugefügt werden. NetApp empfiehlt, mit vier Daten-Nodes pro Storage Controller Array zu beginnen und die Anzahl je nach Workload-Merkmalen auf acht Daten-Nodes pro Storage Controller zu erhöhen.

AFF und FAS sind ideal für in-Place-Analysen. Auf Basis von Berechnungsanforderungen können Node-Manager hinzugefügt werden und unterbrechungsfreier Betrieb ermöglichen es Ihnen, einen Storage-Controller nach Bedarf ohne Ausfallzeit hinzuzufügen. AFF und FAS bieten umfangreiche Funktionen, darunter NVME-Media-Unterstützung, garantierte Effizienz, Datenreduzierung, QOS, prädiktive Analysen, Cloud-Tiering, Replizierung, Cloud-Implementierung und Sicherheit. Um Kunden dabei zu unterstützen, die Anforderungen zu erfüllen, bietet NetApp Funktionen wie Filesystem-Analysen, Kontingente und integrierten Lastausgleich ohne zusätzliche Lizenzkosten. NetApp bietet eine bessere Performance bei gleichzeitigen Aufgaben, niedrigerer Latenz, einfacheren Abläufen und einem höheren Durchsatz von mehreren Gigabyte pro Sekunde als unsere Mitbewerber. Darüber hinaus wird NetApp Cloud Volumes ONTAP bei allen drei großen Cloud-Providern ausgeführt.

Performance-Skalierung – vertikale Skalierung

Vertikale Skalierungsfunktionen ermöglichen es, bei Bedarf zusätzliche Storage-Kapazität Festplattenlaufwerke zu AFF, FAS und E-Series Systemen hinzuzufügen. Mit Cloud Volumes ONTAP besteht die Skalierung von Storage auf PB-Ebene aus zwei Faktoren: das tiering selten genutzter Daten aus Block-Storage in Objektspeicher und das Stapeln von Cloud Volumes ONTAP Lizenzen ohne zusätzliche Rechenleistung.

Mehrere Protokolle

NetApp Systeme unterstützen die meisten Protokolle in Hadoop Implementierungen, einschließlich SAS, iSCSI, FCP, InfiniBand Und NFS.

Betriebliche und unterstützte Lösungen

Die in diesem Dokument beschriebenen Hadoop Lösungen werden von NetApp unterstützt. Diese Lösungen sind auch für größere Hadoop Distributoren zertifiziert. Weitere Informationen finden Sie im "MapR" Standort, die "Hortonworks" Standort und Cloudera zur Verfügung "Zertifizierung" Und "Partner" Standorte.

Zielgruppe

Die Welt der Analyse und Data Science hat unterschiedliche Disziplinen in DER IT und in den Bereichen Business zu berühren:

- Der Data Scientist benötigt die Flexibilität, ihre Tools und Bibliotheken einzusetzen.
- Der Data Engineer muss wissen, wie die Daten fließen und wo sie sich befinden.
- DevOps-Engineers benötigen die Tools, um neue KI- und ML-Applikationen in ihre CI- und CD-Pipelines zu integrieren.

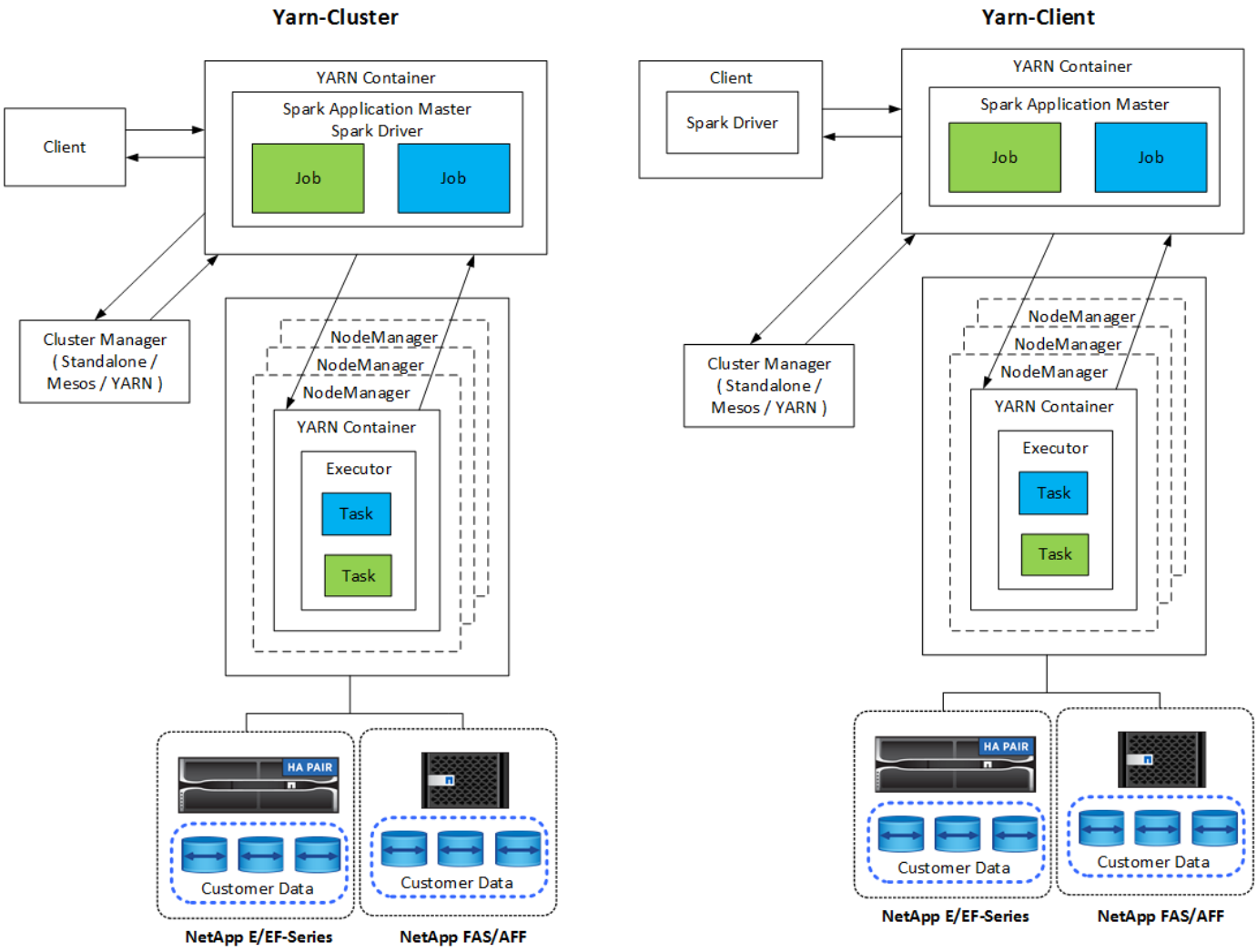
- Cloud-Administratoren und -Architekten müssen in der Lage sein, Hybrid-Cloud-Ressourcen einzurichten und zu managen.
- Unternehmensbenutzer möchten Zugriff auf Analytics-, KI-, ML- und DL-Applikationen haben.

In diesem technischen Bericht werden, wie NetApp AFF, E-Series, StorageGRID, NFS Direct Access, Apache Spark, Horovod und Keras helfen dabei, jede dieser Rollen dem Unternehmen einen Mehrwert zu bieten.

Lösungstechnologie

Apache Spark ist ein beliebtes Programmierungs-Framework für Hadoop Applikationen, das direkt mit Hadoop Distributed File System (HDFS) zusammenarbeitet. Spark ist produktionsbereit, unterstützt die Verarbeitung von Streaming-Daten und ist schneller als MapReduce. Spark verfügt über ein in-Memory-Daten-Caching und sorgt so für eine effiziente Iteration. Die Spark Shell ist interaktiv für das Lernen und die Erforschung von Daten. Mit Spark können Sie Anwendungen in Python, Scala oder Java erstellen. Funkenanwendungen bestehen aus einem oder mehreren Jobs, die eine oder mehrere Aufgaben haben.

Jede Spark-Anwendung verfügt über einen Spark-Treiber. Im YARN-Client-Modus wird der Treiber lokal auf dem Client ausgeführt. Im YARN-Cluster-Modus wird der Treiber im Cluster auf dem Anwendungsmaster ausgeführt. Im Cluster-Modus wird die Applikation weiterhin ausgeführt, auch wenn die Verbindung des Clients getrennt wird.



Es gibt drei Cluster Manager:

- **Standalone.** dieser Manager ist ein Teil von Spark, was es einfach macht, einen Cluster einzurichten.
- **Apache Mesos.** Dies ist ein allgemeiner Clustermanager, der auch MapReduce und andere Anwendungen ausführt.
- **Hadoop YARN.** Dies ist ein Resource Manager in Hadoop 3.

Der Resilient Distributed Dataset (RDD) ist die primäre Komponente von Spark. RDD erstellt die verlorenen und fehlenden Daten aus dem Speicher des Clusters neu und speichert die ursprünglichen Daten, die aus einer Datei stammen oder programmgesteuert erstellt werden. RDDs werden aus Dateien, Daten im Speicher oder einem anderen RDD erstellt. Die Funkenprogrammierung führt zwei Vorgänge durch: Transformation und Aktionen. Durch Transformation wird ein neues RDD auf Basis eines vorhandenen erstellt. Aktionen geben einen Wert aus einem RDD zurück.

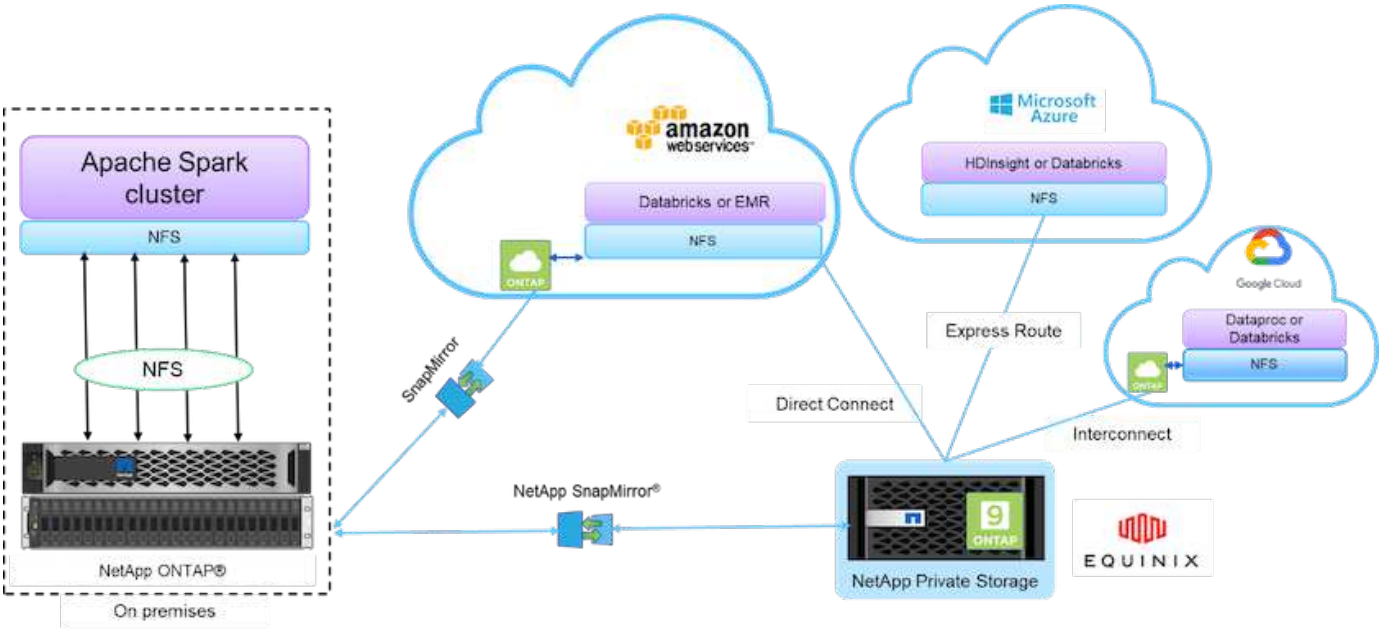
Transformationen und Aktionen gelten auch für Spark DataFrames und DataFrames. Ein Datensatz ist eine verteilte Sammlung von Daten, die die Vorteile von RDDs (starke Eingabe, Verwendung von Lambda-Funktionen) mit den Vorteilen der optimierten Ausführung von Spark SQL bietet. Ein Datensatz kann aus JVM-Objekten erstellt und anschließend mithilfe von funktionalen Transformationen (Map, FlatMap, Filter usw.) manipuliert werden. Ein DataFrame ist ein Datensatz, der in benannte Spalten organisiert ist. Es ist konzeptionell gleichbedeutend mit einer Tabelle in einer relationalen Datenbank oder einem Datenrahmen in R/Python. Datenframes können aus einer Vielzahl von Quellen wie strukturierten Datendateien, Tabellen in Hive/HBase, externen Datenbanken vor Ort oder in der Cloud oder vorhandenen RDDs erstellt werden.

Funkenanwendungen umfassen einen oder mehrere Spark-Jobs. Die Jobs führen Aufgaben in Ausführenden aus, und die Ausführenden werden in YARN-Containern ausgeführt. Jeder Ausführende wird in einem einzigen Container ausgeführt und Ausführende existieren während des gesamten Lebenszyklus einer Applikation. Ein Executor wird nach dem Start der Anwendung repariert und YARN ändert nicht die Größe des bereits zugewiesenen Containers. Ein Ausführender kann Aufgaben gleichzeitig auf Speicherdaten ausführen.

NetApp Spark-Lösungen im Überblick

NetApp verfügt über drei Storage-Portfolios: FAS/AFF, E-Series und Cloud Volumes ONTAP. Wir haben AFF und die E-Series mit ONTAP Storage-System für Hadoop Lösungen mit Apache Spark validiert.

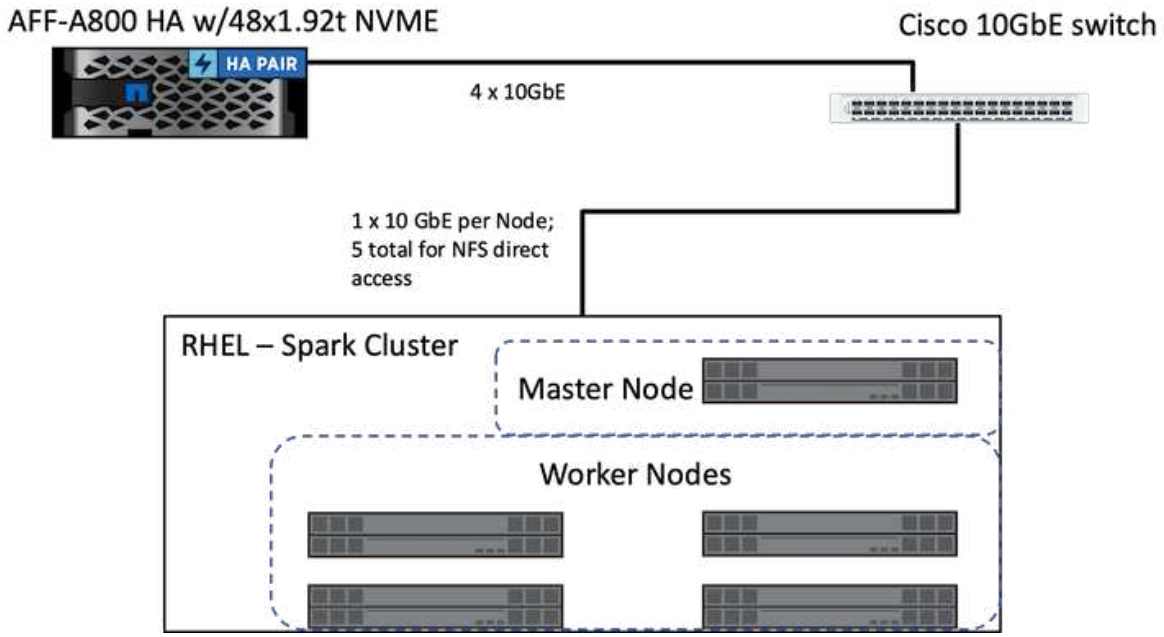
Die Data Fabric von NetApp integriert Datenmanagement-Services und -Applikationen (Bausteine) für Datenzugriff, Kontrolle, Sicherung und Sicherheit, wie in der Abbildung unten dargestellt.



Die Abbildung oben beinhaltet folgende Bausteine:

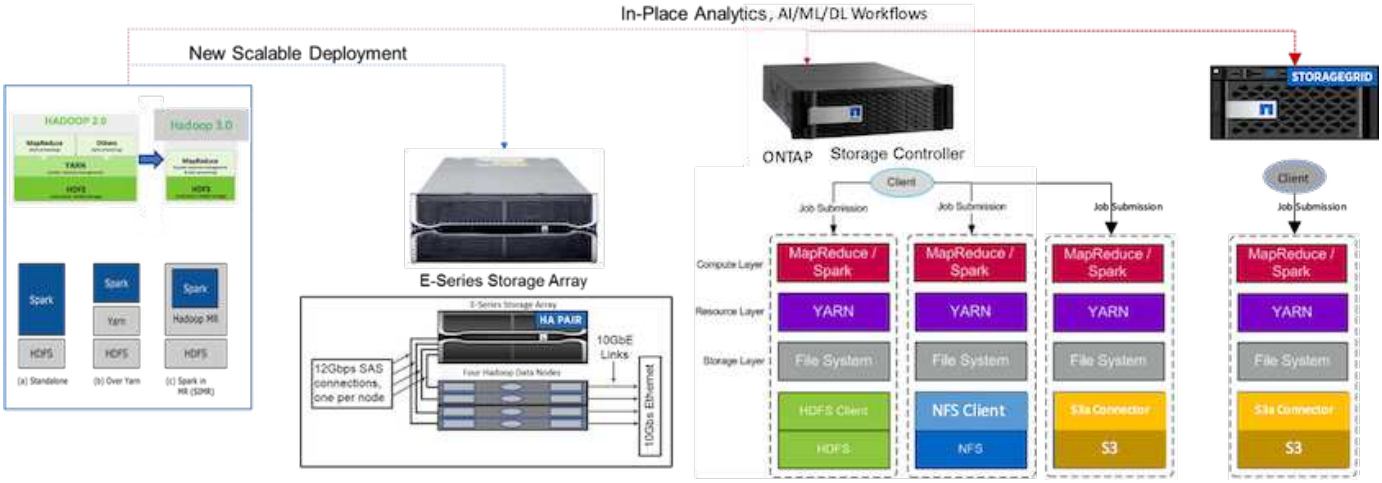
- **NetApp NFS Direct Access.** bietet die neuesten Hadoop und Spark Cluster mit direktem Zugriff auf NetApp NFS Volumes ohne zusätzliche Software- oder Treiberanforderungen.
- **NetApp Cloud Volumes ONTAP und Cloud-Volume-Services.** softwaredefinierter vernetzter Storage auf Basis von ONTAP, der in Amazon Web Services (AWS) oder Azure NetApp Files (ANF) in Microsoft Azure Cloud-Services ausgeführt wird.
- **NetApp SnapMirror Technologie.** bietet Datensicherungsfunktionen zwischen On-Premises-Umgebungen und ONTAP Cloud oder NPS Instanzen.
- **Cloud-Service-Provider.** zu diesen Anbietern gehören AWS, Microsoft Azure, Google Cloud und IBM Cloud.
- **PaaS.** Cloud-basierte Analyseservices wie Amazon Elastic MapReduce (EMR) und Databricks in AWS sowie Microsoft Azure HDInsight und Azure Databricks.

In der folgenden Abbildung ist die Spark-Lösung mit NetApp Storage dargestellt.

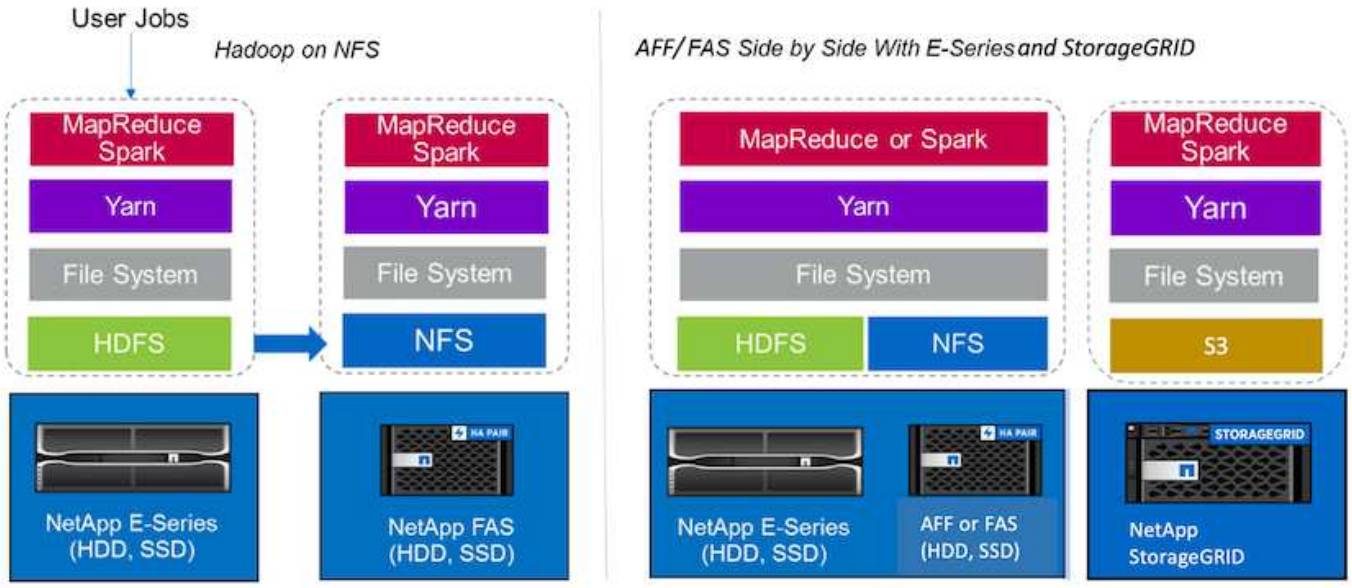


Die ONTAP Spark Lösung verwendet das NetApp NFS Direct-Access-Protokoll für in-Place-Analysen sowie KI-, ML- und DL-Workflows, wobei auf vorhandene Produktionsdaten zugegriffen wird. Produktionsdaten, die Hadoop-Nodes zur Verfügung stehen, werden exportiert, um in-Place-Analysen und KI-, ML- und DL-Jobs auszuführen. Die Daten können in Hadoop Nodes entweder mit direkt oder ohne NetApp NFS verarbeitet werden. In Spark mit dem Standalone oder `yarn` Cluster Manager, Sie können ein NFS-Volume mithilfe von konfigurieren `file:///<target_volume`. Wir haben drei Anwendungsfälle mit unterschiedlichen Datensätzen validiert. Die Details dieser Validierungen finden Sie im Abschnitt „Testergebnisse“. (xref)

Abbildung: Die Positionierung von NetApp Apache Spark/Hadoop Storage



Es wurden die einzigartigen Funktionen der E-Series Spark-Lösung, die All Flash FAS/FAS ONTAP Spark-Lösung und die StorageGRID Spark-Lösung identifiziert und detaillierte Validierungen und Tests durchgeführt. Basierend auf unseren Beobachtungen empfiehlt NetApp die E-Series Lösung für Greenfield-Installationen und neue skalierbare Implementierungen. Die All Flash FAS/FAS Lösung für in-Place-Analysen, KI-, ML- und DL-Workloads nutzt dabei vorhandene NFS-Daten sowie StorageGRID für AI, ML und DL sowie moderne Datenanalysen, wenn Objekt-Storage benötigt wird.



Ein Data Lake ist ein Storage-Repository für große Datensätze in nativer Form, das für Analytics-, KI-, ML- und DL-Jobs verwendet werden kann. Wir haben ein Data-Lake-Repository für die E-Series, All Flash FAS/FAS und StorageGRID SG6060 Spark Lösungen erstellt. Das E-Series System bietet HDFS Zugriff auf das Hadoop Spark-Cluster, während auf vorhandene Produktionsdaten über das NFS-Direktzugriffsprotokoll auf den Hadoop-Cluster zugegriffen wird. Für Datensätze, die sich im Objekt-Storage befinden, bietet NetApp StorageGRID sicheren Zugriff über S3 und S3A.

Zusammenfassung des Anwendungsfalls

Auf dieser Seite werden die verschiedenen Bereiche beschrieben, in denen diese Lösung verwendet werden kann.

Streaming von Daten

Apache Spark kann Streamingdaten verarbeiten, die für ETL-Prozesse (Extrahieren, Transformieren, Laden), Anreicherung, Auslösen von Ereigniserkennung und komplexe Sitzungsanalysen verwendet werden:

- **Streaming ETL.** Daten werden kontinuierlich gereinigt und aggregiert, bevor sie in Datenspeicher geschoben werden. Netflix nutzt Kafka- und Spark-Streaming, um eine Online-Filmempfehlungen und eine Lösung für das Daten-Monitoring in Echtzeit zu erstellen, mit der täglich Milliarden von Ereignissen aus unterschiedlichen Datenquellen verarbeitet werden können. Herkömmliche ETL für die Batch-Verarbeitung wird jedoch unterschiedlich behandelt. Diese Daten werden zuerst gelesen und dann in ein Datenbankformat konvertiert, bevor sie in die Datenbank geschrieben werden.
- **Datenanreicherung.** Spark Streaming bereichert die Live-Daten mit statischen Daten, um eine Echtzeitdatenanalyse zu ermöglichen. So können Online-Werbetreibende beispielsweise personalisierte, gezielte Werbeanzeigen liefern, die von Informationen über das Kundenverhalten geleitet werden.
- **Trigger Event Detection.** mit Spark Streaming können Sie ungewöhnliche Verhaltensweisen erkennen und schnell darauf reagieren, die möglicherweise schwerwiegende Probleme darstellen könnten. So verwenden Finanzinstitute beispielsweise Auslöser zum Erkennen und Stoppen von Betrugstransaktionen, und Krankenhäuser verwenden Auslöser, um gefährliche Gesundheitsänderungen zu erkennen, die in den Vitalparamalen eines Patienten nachgewiesen wurden.
- **Komplexe Sitzungsanalyse.** Spark Streaming erfasst Ereignisse wie Benutzeraktivitäten nach der Anmeldung an einer Website oder Anwendung, die dann gruppiert und analysiert werden. Beispielsweise

setzt Netflix diese Funktionalität ein, um Filmempfehlungen in Echtzeit bereitzustellen.

Weitere Informationen zur Konfiguration von Streaming-Daten, Confluent Kafka Verification und Performance-Tests finden Sie unter "TR-4912: Best Practice Guidelines für Conflient Kafka Tiered Storage mit NetApp".

Machine Learning

Das in Spark integrierte Framework ermöglicht es Ihnen, mithilfe der Machine Learning Library (MLlib) wiederholte Abfragen zu Datensätzen durchzuführen. MLlib wird in Bereichen wie Clustering, Klassifizierung und Größenreduzierung für einige gängige Big Data-Funktionen wie Predictive Intelligence, Kundensegmentierung zu Marketingzwecken und Sentiment-Analyse verwendet. MLlib wird zur Netzwerksicherheit verwendet, um Echtzeit-Inspektionen von Datenpaketen auf Anzeichen schädlicher Aktivität durchzuführen. Sicherheitsanbieter lernen neue Bedrohungen kennen und halten sich vor Hackern, während sie ihre Kunden in Echtzeit schützen.

Deep Learning

TensorFlow ist ein beliebtes Deep-Learning-Framework, das in der Branche verwendet wird. TensorFlow unterstützt das verteilte Training auf einem CPU- oder GPU-Cluster. Dieses Distributed Training ermöglicht die Ausführung der IT-Abteilung bei großen Datenmengen mit vielen tiefgreifenden Schichten.

Bis ziemlich lange Zeit, wenn wir TensorFlow mit Apache Spark verwenden wollten, mussten wir alle erforderlichen ETL für TensorFlow in PySpark ausführen und dann Daten auf Intermediate Storage schreiben. Diese Daten werden dann für den tatsächlichen Trainingsprozess auf den TensorFlow Cluster geladen. Bei diesem Workflow musste der Benutzer zwei verschiedene Cluster verwalten, eines für ETL und eines für das verteilte Training von TensorFlow. Das Ausführen und Warten mehrerer Cluster war normalerweise langwierig und zeitaufwendig.

DataFrames und RDD in früheren Spark-Versionen waren nicht gut für Deep Learning geeignet, da der zufällige Zugriff nur begrenzt verfügbar war. In Spark 3.0 mit Projekt Wasserstoff wird native Unterstützung für die Deep-Learning-Frameworks hinzugefügt. Dieser Ansatz ermöglicht nicht-MapReduce-basierte Planung auf dem Spark-Cluster.

Interaktive Analyse

Apache Spark ist schnell genug, um explorative Abfragen ohne Sampling mit anderen Entwicklungssprachen als Spark, einschließlich SQL, R und Python, durchzuführen. Spark setzt Visualisierungstools ein, um komplexe Daten zu verarbeiten und interaktiv zu visualisieren. Spark mit strukturiertem Streaming führt interaktive Abfragen gegen Live-Daten in Web-Analytics durch, die es Ihnen ermöglichen, interaktive Abfragen gegen die aktuelle Sitzung eines Webbesuchers durchzuführen.

Empfehlungssystem

Im Laufe der Jahre haben Empfehlungssysteme zu erheblichen Veränderungen in unserem Leben geführt, da Unternehmen und Verbraucher auf dramatische Veränderungen im Online-Shopping, Online-Entertainment und vielen anderen Branchen reagiert haben. Diese Systeme gehören tatsächlich zu den offensichtlichsten Erfolgen von KI in der Produktion. In vielen praktischen Anwendungsfällen werden Empfehlungssysteme mit konversationaler KI oder Chatbots kombiniert, die mit einem NLP-Backend interfasiert werden, um relevante Informationen zu erhalten und nützliche Rückschlüsse zu gewinnen.

Heute setzen viele Einzelhändler neue Geschäftsmodelle ein, wie Online-Kauf und Abholung im Geschäft, Abholung von Bordsteinseiten, Selbstauszahlungnahme, Scannen und Los und vieles mehr. Diese Modelle haben sich während der COVID-19 Pandemie durch die Verbesserung des Einkaufs sicherer und bequemer für die Verbraucher. KI ist entscheidend für die wachsenden digitalen Trends, die vom Verbraucherverhalten

beeinflusst werden und umgekehrt. Um die steigenden Anforderungen der Verbraucher zu erfüllen, die Kundenzufriedenheit zu erhöhen, die betriebliche Effizienz zu verbessern und den Umsatz zu steigern, unterstützt NetApp seine Enterprise-Kunden und Unternehmen dabei, Machine-Learning- und Deep-Learning-Algorithmen zu nutzen, um schneller und genauer Empfehlungen zu entwickeln.

Es gibt verschiedene gängige Techniken zur Bereitstellung von Empfehlungen, wie kollaborative Filterung, Content-basierte Systeme, das Deep Learning Recomender Modell (DLRM) und hybride Techniken. Kunden nutzten bereits PySpark, um kollaborative Filterfunktionen für die Erstellung von Empfehlungssystemen zu implementieren. Spark MLlib implementiert alternierende geringste Quadrate (als) für kollaborative Filterung, ein sehr beliebter Algorithmus bei Unternehmen vor dem Aufstieg von DLRM.

Natürliche Sprachverarbeitung

Conversational AI, ermöglicht durch natürliche Sprachverarbeitung (NLP), ist der Zweig der KI, die Computer helfen, mit Menschen zu kommunizieren. NLP ist in allen Branchen vertikale und viele Anwendungsfälle verbreitet, von intelligenten Assistenten und Chatbots bis hin zu Google-Suche und Predictive Text. Laut A "Gartner" Prognose: Bis 2022 werden 70 % der Menschen täglich mit umgangssprachlichen KI-Plattformen interagieren. Für ein qualitativ hochwertiges Gespräch zwischen Mensch und Maschine müssen schnelle, intelligente und natürliche Reaktionen erfolgen.

Kunden benötigen eine große Menge an Daten, um ihre NLP- und ASR-Modelle (Automatic Speech Recognition) zu verarbeiten und zu trainieren. Darüber hinaus müssen sie Daten zwischen dem Edge-Bereich, dem Core-Bereich und der Cloud verschieben. Dazu müssen sie in wenigen Millisekunden Inferenz durchführen, um eine natürliche Kommunikation mit dem Menschen zu gewährleisten. NetApp AI und Apache Spark sind eine ideale Kombination für Computing, Storage, Datenverarbeitung, Modelltraining, Feintuning, Und -Einsatz,

Die Sentimentanalyse ist ein Untersuchungsfeld innerhalb des NLP, in dem positive, negative oder neutrale Gefühle aus dem Text extrahiert werden. Die Sentiment-Analyse hat verschiedene Anwendungsfälle: Von der Ermittlung der Mitarbeiterleistung im Support Center in Gesprächen mit Anrufern bis hin zur Bereitstellung geeigneter automatisierter Chatbot-Antworten. Es wurde auch verwendet, um den Aktienkurs eines Unternehmens auf der Grundlage der Wechselwirkungen zwischen Firmenvertretern und dem Publikum bei vierteljährlichen Ertragsaufrufen vorherzusagen. Darüber hinaus kann die Sentiment-Analyse verwendet werden, um die Sicht eines Kunden auf die Produkte, Dienstleistungen oder Unterstützung der Marke zu bestimmen.

Wir haben das verwendet "Funke NLP" Bibliothek von "John Snow Labs" So laden Sie vortrainierte Pipelines und bidirektionale Encoder-Darstellungen von Transformatoren (BERT) Modellen einschließlich "Stimmung in den Finanznachrichten" Und "FinBERT", Durchführung der Tokenisierung, Named Entity Recognition, Modelltraining, Anpassung und Sentiment Analyse nach Maß. Spark NLP ist die einzige Open-Source-NLP-Bibliothek in Produktion, die hochmoderne Transformatoren wie BERT, ALBERT, ELECTRA, XLNet, DistilBERT, Roberta, Deberta, XLM- Roberta, Longformer, ELMO, Universal Sentence Encoder, Google T5, MarianMT und GPT2. Die Bibliothek funktioniert nicht nur in Python und R, sondern auch im JVM Ökosystem (Java, Scala und Kotlin) im großen Maßstab, indem sie Apache Spark nativ erweitert.

Anwendungsfälle und Architekturen für KI, ML und DL

Größere Anwendungsfälle und Methoden für KI, ML und DL können in die folgenden Abschnitte unterteilt werden:

Spark NLP-Pipelines und TensorFlow Distributed Inferenzierung

Die folgende Liste enthält die beliebtesten Open-Source-NLP-Bibliotheken, die von der Data Science-

Community unter verschiedenen Entwicklungsstufen übernommen wurden:

- "Natural Language Toolkit (NLTK)". Das komplette Toolkit für alle NLP-Techniken. Es wurde seit den frühen 2000er Jahren erhalten.
- "TextBlob". Eine einfach zu bedienende NLP-Tools Python-API, die auf NLTK und Pattern aufgebaut ist.
- "Stanford Core NLP". NLP-Dienste und -Pakete in Java, entwickelt von der Stanford NLP Group.
- "Gensim". Thema Modellierung für Menschen begann als Sammlung von Python-Skripten für das Projekt der Tschechischen Digital Mathematics Library.
- "Spacy". Durchgängige industrielle NLP-Workflows mit Python und Cython mit GPU-Beschleunigung für Transformatoren.
- "Fasttext". Eine kostenlose, leichte, Open-Source-NLP-Bibliothek für das Lernen von Wortschaltungen und Satzklassifizierung, die im AI Research Lab (FAIR) von Facebook erstellt wurde.

Spark NLP ist eine einzige, einheitliche Lösung für alle NLP-Aufgaben und -Anforderungen, die skalierbare, leistungsstarke und hochpräzise NLP-gestützte Software für reale Produktionsanwendungsfälle ermöglicht. Es nutzt Transfer-Learning und implementiert modernste Algorithmen und Modelle in der Forschung und Industrie. Da Spark keine volle Unterstützung für die oben genannten Bibliotheken hat, wurde Spark NLP auf der Basis aufgebaut "Funken ML" Um die Vorteile der universellen in-Memory Distributed Data Processing Engine von Spark als NLP-Bibliothek der Enterprise-Klasse für unternehmenskritische Produktions-Workflows zu nutzen. Die Kommentatoren nutzen regelbasierte Algorithmen, Machine Learning und TensorFlow, um Deep-Learning-Implementierungen zu unterstützen. Dies betrifft gängige NLP-Aufgaben, einschließlich aber nicht beschränkt auf Tokenisierung, Lemmatisierung, Stemming, Teil-of-Speech-Tagging, benannte Entity-Erkennung, Rechtschreibprüfung und Sentiment-Analyse.

Bidirektionale Encoder-Darstellungen von Transformatoren (BERT) ist eine Transformatorbasierte Machine Learning-Technik für NLP. Es popularisierte das Konzept der Vorausbildung und Feinabstimmung. Die Transformatorarchitektur in BERT entstand aus maschineller Übersetzung, die langfristige Abhängigkeiten besser modelt als rezidierende Neural Network (RNN)-basierte Sprachmodelle. Außerdem wurde die MLM-Aufgabe (Masked Language Modeling) eingeführt, bei der zufällige 15 % aller Token maskiert und das Modell sie vorhersagt, wodurch eine echte Bidirektionalität möglich ist.

Die Analyse der Finanzstimmung ist aufgrund der Fachsprache und des Mangels an gekennzeichneten Daten in diesem Bereich schwierig. "FinBERT", Ein Sprachmodell basierend auf vortrainierten BERT, wurde Domain angepasst "Reuters-TRC2", Ein Finanzkorpus, und fein abgestimmt mit beschrifteten Daten ("FinanzphraseBank") Für die Klassifizierung der Finanzstimmung. Forscher haben 4, 500 Sätze aus Nachrichten-Artikeln mit finanziellen Begriffen extrahiert. Dann 16 Experten und Master Studenten mit Finanz-Hintergrund bezeichnet die Sätze als positiv, neutral und negativ. Wir haben einen End-to-End Spark-Workflow entwickelt, um die Stimmung für die von 2016 bis 2020 am NASDAQ am Markt vertelefonisch unter den Top-10-Unternehmen zu analysieren und FinBERT sowie zwei weitere vortrainierte Pipelines ("Sentiment Analysis for Financial News", "Dokument-DL erklären") Von Spark NLP.

Die zugrunde liegende Deep-Learning-Engine für Spark NLP ist TensorFlow, eine End-to-End-Open-Source-Plattform für Machine Learning, die den einfachen Modellbau, die robuste ML-Produktion an jedem Ort und leistungsstarke Experimente ermöglicht. Deshalb bei der Ausführung unserer Pipelines in Spark `yarn cluster` Modus, wir hatten im Wesentlichen Distributed TensorFlow mit Daten und Modellparallelisierung über einen Master- und mehrere Worker-Node sowie über den auf dem Cluster angebundenen Network-Attached Storage.

Horovod Distributed Training

Die Kernvalidierung für MapReduce-bezogene Performance wird mit TeraGen, TeraSort, TeraValidate und DFSIO (Lese- und Schreibvorgänge) durchgeführt. Die Ergebnisse der TeraGen und TeraSort Validierung

werden in vorgestellt "TR-3969: NetApp Lösungen für Hadoop" Für E-Series und im Abschnitt „Storage Tiering“ (xref) für AFF.

Auf Kundenwünsche basieren verteilte Schulungen mit Spark auf einer der wichtigsten Nutzungsfälle. In diesem Dokument wurden von verwendet "Hovorod auf Spark" Spark-Performance mit NetApp On-Premises-, Cloud-nativen und Hybrid-Cloud-Lösungen mit NetApp All Flash FAS (AFF) Storage-Controllern, Azure NetApp Files und StorageGRID validieren

Das Paket Horovod on Spark bietet eine praktische Wrapper für Horovod, mit der verteilte Trainings-Workloads in Spark-Clustern einfach ausgeführt werden. Es ermöglicht einen eng aneinander auslaufenden Modelldesign-Loop, in dem Datenverarbeitung, Modelltraining und Modellevaluierung in Spark ausgeführt werden, wo sich die Trainings- und Inferenzdaten befinden.

Es gibt zwei APIs für die Ausführung von Horovod auf Spark: Eine High-Level Estimator API und eine Low-Level-Run-API. Obwohl beide den gleichen zugrunde liegenden Mechanismus verwenden, um Horovod auf Spark-Executoren zu starten, abstrahiert die Estimator-API die Datenverarbeitung, Modelltrainingschleife, Modell Checkpointing, Kennzahlenerfassung und verteilte Schulung. Wir haben Horovod Spark Estimators, TensorFlow und Keras eingesetzt, um eine End-to-End-Datenvorbereitung und einen verteilten Trainings-Workflow auf Basis der zu ermöglichen "Kaggle Rossmann Store Sales" Die Wettbewerbssituation.

Das Skript `keras_spark_horovod_rossmann_estimator.py` Finden Sie im Abschnitt "Python-Skripte für jeden größeren Anwendungsfall." Sie besteht aus drei Teilen:

- Der erste Teil führt verschiedene Schritte zur Datenvorverarbeitung über einen ersten Satz von CSV-Dateien durch, die von Kaggle bereitgestellt und von der Community gesammelt werden. Die Eingabedaten werden mit einem in ein Trainingssatz unterteilt `Validation` Teilmenge und ein Testdatensatz.
- Der zweite Teil definiert ein Keras Deep Neural Network (DNN)-Modell mit logarithmischer Sigmoid-Aktivierungsfunktion und einem Adam-Optimizer und führt Distributed Training des Modells mit Horovod on Spark durch.
- Der dritte Teil führt eine Prognose für den Testdatensatz durch. Dabei wird das beste Modell verwendet, durch das der Validierungssatz insgesamt für absolute Fehler minimiert wird. Anschließend wird eine CSV-Ausgabedatei erstellt.

Siehe Abschnitt "„Maschinelles Lernen“" Für verschiedene Laufzeitvergleichsergebnisse.

Deep Learning mit mehreren Mitarbeitern und Keras für die CTR-Vorhersage

Aufgrund der jüngsten Fortschritte BEI ML-Plattformen und -Applikationen wird jetzt viel Aufmerksamkeit auf das Lernen in großen Umgebungen gerichtet. Die Klickrate (Klickrate, CTR) ist definiert als die durchschnittliche Anzahl der Klickrate pro hundert Online-Werbeeindrücke (ausgedrückt als Prozentsatz). Es wird in verschiedenen Branchen und Anwendungsfällen, darunter digitales Marketing, Einzelhandel, E-Commerce und Service-Provider, häufig als wichtige Metrik eingeführt. Sehen Sie unsere "TR-4904: Distributed Training in Azure - Click-Through Rate Prediction" Weitere Details zu den Applikationen von CTR und einer End-to-End-Cloud-KI-Workflow-Implementierung mit Kubernetes, Distributed Data ETL und Modelltraining mit Dask und CUDA ML

In diesem technischen Bericht haben wir eine Variation der verwendet "Criteo Terabyte Klicken Sie auf Protokolldatensatz" (Siehe TR-4904) für verteiltes Deep Learning mit Keras für mehrere Mitarbeiter, um einen Spark-Workflow mit Deep and Cross Network (DCN) Modellen zu erstellen, und vergleicht seine Leistung hinsichtlich der Log-Loss-Fehlerfunktion mit einem Basismodell des Spark ML Logistic Regression. DCN erfasst effiziente Interaktionen von Funktionen in begrenzt abstufenden Graden, erlernt hochnichtlineare Interaktionen, erfordert keine manuelle Funktionstechnik oder umfassende Suche und hat geringe Rechenkosten.

Daten für webbasierte Empfehlungssysteme sind meist diskret und kategorisch. Dies führt zu einem großen und spärlichen Speicherplatz, der für die Untersuchung von Funktionen eine Herausforderung darstellt. Dies hat die meisten Großsysteme auf lineare Modelle wie logistische Regression beschränkt. Das Erkennen häufig prädiktiver Funktionen und gleichzeitig das Erforschen von unsichtbaren oder seltenen Cross-Funktionen ist jedoch der Schlüssel für gute Vorhersagen. Lineare Modelle sind einfach, auswertbar und einfach zu skalieren, aber sie sind in ihrer Ausdruckskraft begrenzt.

Kreuzstücke hingegen haben sich als bedeutsam erwiesen, um die Ausdruckskraft der Modelle zu verbessern. Leider ist oftmals eine manuelle Funktionstechnik oder eine umfassende Suche erforderlich, um solche Funktionen zu identifizieren. Die Verallgemeinerung von Interaktionen mit unsichtbaren Funktionen ist oft schwierig. Die Verwendung eines neuronalen Netzwerks wie DCN vermeidet aufgabenspezifischen Funktionengineering, indem die Funktionsübergänge explizit automatisch angewendet werden. Das Cross-Netzwerk besteht aus mehreren Ebenen, wobei der höchste Grad an Interaktionen durch die Schichttiefe bestimmt wird. Jede Ebene erzeugt Interaktionen in höherer Reihenfolge, die auf bestehenden basieren, und behält die Interaktionen von vorherigen Ebenen.

Ein Deep Neural Network (DNN) verspricht sehr komplexe Interaktionen über verschiedene Funktionen hinweg. Im Vergleich zu DCN benötigt es jedoch fast eine Größenordnung von mehr Parametern, ist nicht in der Lage, Querfunktionen explizit zu bilden und kann möglicherweise nicht effizient lernen einige Arten von Feature-Interaktionen. Das Cross Network ist speichereffizient und einfach zu implementieren. Die gemeinsame Schulung der Cross- und DNN-Komponenten ermöglicht eine effiziente Erfassung prädiktiver Feature-Interaktionen und liefert hochmoderne Performance im Criteo CTR-Datensatz.

Ein DCN-Modell beginnt mit einer Einbettung- und Stapelschicht, gefolgt von einem Cross-Netzwerk und einem tiefen Netzwerk parallel. Auf diese wiederum folgt eine endgültige Kombinationsschicht, die die Ausgänge der beiden Netzwerke miteinander kombiniert. Ihre Eingabedaten können ein Vektor mit spärlichen und dichten Funktionen sein. In Spark, beide "Ml" Und "Mllib" Bibliotheken enthalten den Typ `SparseVector`. Daher ist es wichtig, dass die Benutzer zwischen den beiden unterscheiden und beim Aufruf ihrer jeweiligen Funktionen und Methoden achtsam sind. Bei empfohlenen Web-Scale-Systemen wie der CTR-Vorhersage handelt es sich beispielsweise um kategorische Merkmale `'country=usa'`. Solche Funktionen werden oft als ein-Hot-Vektoren kodiert, z. B. `'[0,1,0, …]'`. One-Hot-Encoding (OHE) mit `SparseVector` Ist nützlich beim Umgang mit Datensätzen aus der realen Welt mit sich ständig verändernden und wachsenden Vokabularen. Wir haben Beispiele in geändert "DeepCTR" Um große Vokabularblätter zu verarbeiten, erstellen Einbettungsvektoren in der Einbettung- und Stapelschicht unseres DCN.

Der "Criteo Display Ads-Datensatz" Sagt die Durchklickrate für Werbeanzeigen aus. Es verfügt über 13 ganzzahlige Merkmale und 26 kategorische Merkmale, in denen jede Kategorie eine hohe Kardinalität hat. Bei diesem Datensatz ist eine Verbesserung von 0.001 im Logloss aufgrund der großen Eingangsgröße praktisch signifikant. Eine kleine Verbesserung der Vorhersagegenauigkeit für eine große Nutzerbasis kann möglicherweise zu einer großen Steigerung der Unternehmenseinnahmen führen. Der Datensatz enthält 11 GB Benutzerprotokolle von einem Zeitraum von 7 Tagen, was bedeutet etwa 41 Millionen Datensätzen. Wir haben Spark genutzt `dataFrame.randomSplit()function` Zur zufälligen Aufteilung der Daten für das Training (80 %), der Cross-Validierungen (10 %) und der verbleibenden 10 % für Tests

DCN wurde unter TensorFlow mit Keras implementiert. Die Implementierung des Modelltrainings mit DCN umfasst vier Hauptkomponenten:

- **Datenverarbeitung und Einbettung.** echte Funktionen werden durch Anwendung eines Logtransform normalisiert. Für kategorische Merkmale binden wir die Merkmale in dichte Vektoren der Dimension 6×(Category Cardinality)1/4 ein. Das Verketten aller Formationen ergibt einen Vektor der Dimension 1026.
- **Optimierung.** Wir haben die Mini-Batch stochastische Optimierung mit dem Adam Optimizer angewendet. Die Batch-Größe wurde auf 512 gesetzt. Die Batch-Normalisierung wurde auf das tiefe Netzwerk angewendet und die Gradient-Clip-Norm wurde auf 100 gesetzt.

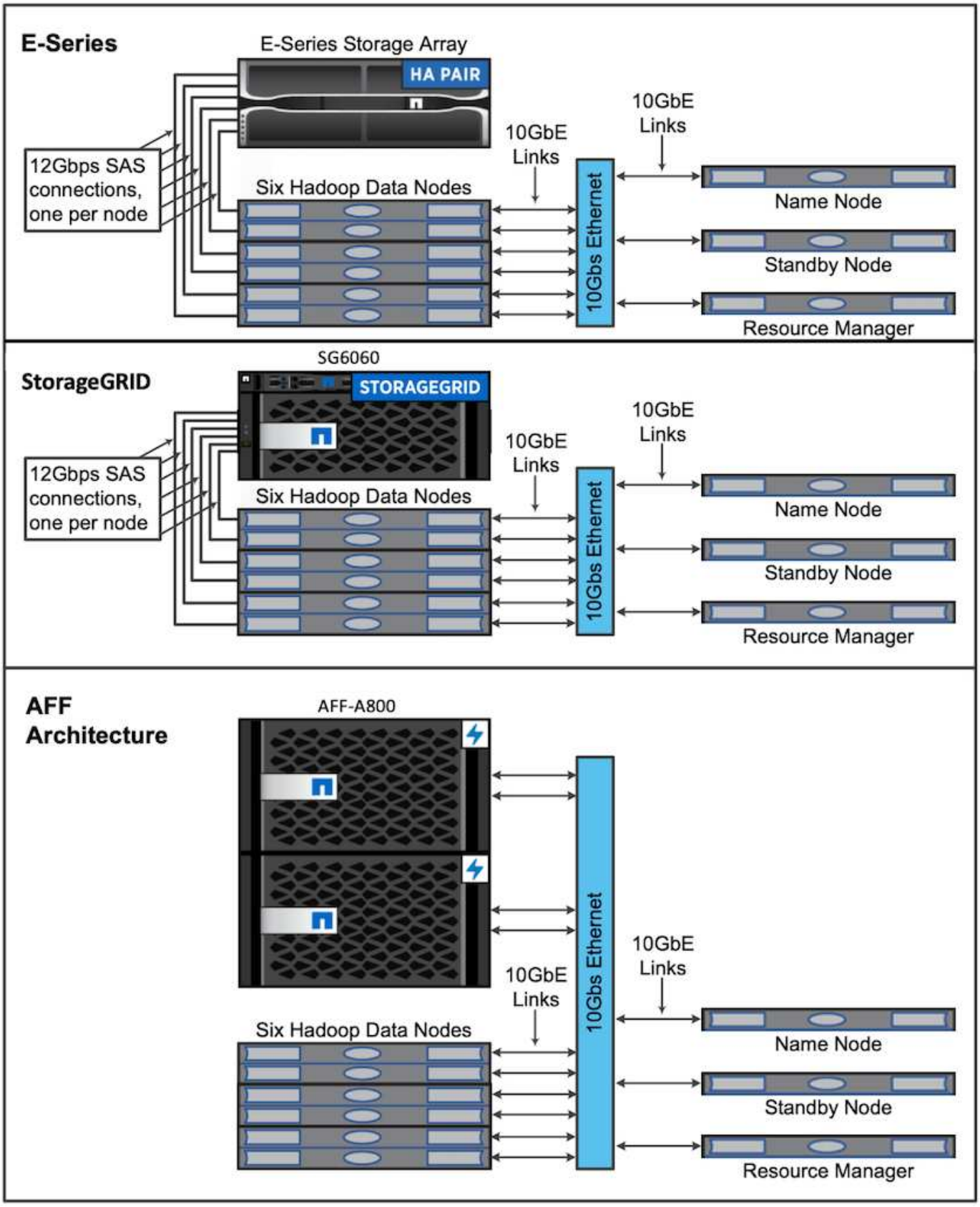
- **Regularisierung.** Wir verwendeten das frühe Stoppen, da L2-Regularisierung oder Dropout nicht als wirksam erwiesen wurde.
- **Hyperparameter.** die Ergebnisse werden anhand einer Rastersuche über die Anzahl der ausgeblendeten Schichten, die versteckte Ebenengröße, die anfängliche Lernrate und die Anzahl der Querschichten berichtet. Die Anzahl der versteckten Schichten reichte von 2 bis 5, mit versteckten Schichtgrößen von 32 bis 1024. Bei DCN betrug die Anzahl der Querschichten von 1 bis 6. Die erste Lernrate wurde von 0.0001 auf 0.001 mit Schritten von 0.0001 abgestimmt. Alle Experimente haben einen frühen Stopp bei Trainingsschritt 150,000 durchgeführt, über den die Überlastung begann.

Neben DCN haben wir auch andere gängige Deep-Learning-Modelle für die CTR-Vorhersage getestet, einschließlich "DeepFM", "XDeepFM", "AutoInt", und "DCN v2".

Zur Validierung verwendete Architekturen

Für diese Validierung haben wir vier Worker-Nodes und einen Master-Node mit einem AFF A800 HA-Paar verwendet. Die Verbindung aller Cluster-Mitglieder wurde über 10-GbE-Netzwerk-Switches hergestellt.

Für diese Validierung der NetApp Spark-Lösungen haben wir drei verschiedene Storage-Controller verwendet: E5760, E5724 und AFF-A800. Die Storage-Controller der E-Series wurden mit fünf Daten-Nodes mit SAS-Verbindungen mit 12 Gbit/s verbunden. Der AFF HA-Paar-Storage Controller liefert exportierte NFS-Volumes über 10-GbE-Verbindungen zu Hadoop Worker-Nodes. Die Hadoop Cluster-Mitglieder wurden über 10-GbE-Verbindungen in den Hadoop Lösungen der E-Series, AFF und StorageGRID Hadoop verbunden.



Testergebnisse

Wir haben mithilfe der Skripts TeraSort und TeraValidate im Benchmark-Tool TeraGen die

Spark-Performance-Validierung mit E5760, E5724 und AFF-A800 Konfigurationen gemessen. Darüber hinaus wurden drei wesentliche Anwendungsfälle getestet: Spark NLP Pipelines und TensorFlow Distributed Training, Horovod Distributed-Training und Multi-Worker Deep Learning mit Keras für CTR Prediction bei DeepFM.

Sowohl für die E-Series als auch für die StorageGRID Validierung verwendeten wir Hadoop Replizierungsfaktor 2. Für die AFF Validierung verwendeten wir nur eine Datenquelle.

In der folgenden Tabelle ist die Hardwarekonfiguration für die Spark-Performance-Validierung aufgeführt.

Typ	Hadoop Worker-Nodes	Laufwerkstyp	Laufwerke pro Node	Storage Controller
SG6060	4	SAS	12	Single High Availability (HA)-Paar
E5760	4	SAS	60	Single HA-Paar
E5724	4	SAS	24	Single HA-Paar
AFF800	4	SSD	6	Single HA-Paar

In der folgenden Tabelle sind die Softwareanforderungen aufgeführt.

Software	Version
RHEL	7.9
OpenJDK Runtime Environment	1.8.0
OpenJDK 64-Bit-Server-VM	25.302
Git	2.24.1
GCC/G++	11.2.1
Funke	3.2.1
PySpark	3.1.2
SparkNLP	3.4.2
TensorFlow	2.9.0
Keras	2.9.0
Horovod	0.24.3

Analyse der finanziellen Stimmung

Wir haben veröffentlicht "TR-4910: Sentiment Analysis from Customer Communications with NetApp AI", In der mithilfe der eine End-to-End-Conversational KI-Pipeline aufgebaut wurde "NetApp DataOps Toolkit", AFF Storage und NVIDIA DGX-System. Die Pipeline führt Stapelverarbeitung von Audiosignalen, ASR (Automatic Speech Recognition), Transfer Learning und Sentiment-Analysen durch und nutzt dabei das DataOps Toolkit, "NVIDIA Riva SDK", Und das "Tao-Rahmen". Wir haben den Anwendungsfall der Sentiment-Analyse auf die Finanzdienstleistungsbranche ausgeweitet und einen SparkNLP-Workflow entwickelt, drei BERT-Modelle für verschiedene NLP-Aufgaben wie die benannte Anerkennung von Unternehmen geladen und die Quartalsgewinne der NASDAQ Top 10-Unternehmen in Sentenniveal getitelt.

Das folgende Skript `sentiment_analysis_spark. py` Verwendet das FinBERT-Modell, um Transkripte in HDFS zu verarbeiten und positive, neutrale und negative Stimmungswerte zu erzielen, wie in der folgenden Tabelle dargestellt:

```
-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
```

Die folgende Tabelle enthält eine Analyse der Stimmung bei den Gewinnen und Satzungen der NASDAQ Top 10-Unternehmen von 2016 bis 2020.

Einschätzung zählt und Prozentsatz	**Alle 10 Unternehmen**	**AAPL**	**AMD**	**AMZN**	**CSCO**	**GOOGL**	**INTC**	**MSFT**	**NVDA**
Positive Ergebnisse	7447	1567	743	290	682	826	824	904	417
Nullwerte	64067	6856	7596	5086	6650	5914	6099	5715	6189
Negative Zählung	1787	253	213	84	189	97	282	202	89
Nicht kategorisierte Anzahl	196	0	0	76	0	0	0	1	0
(Gesamtanzahl)	73497	8676	8552	5536	7521	6837	7205	6822	6695

In Bezug auf Prozentsätze sind die meisten Sätze, die von den CEOs und CFOs gesprochen werden, faktisch und tragen daher eine neutrale Stimmung. Während eines Gewinnanrufs stellen Analysten Fragen, die eine positive oder negative Stimmung vermitteln könnten. Es lohnt sich, noch einmal quantitativ zu untersuchen, wie sich negative oder positive Stimmungen auf die Aktienkurse am gleichen oder nächsten Handelstag auswirken.

In der folgenden Tabelle ist die Sentiment-Analyse auf Satzebene für NASDAQ Top 10-Unternehmen

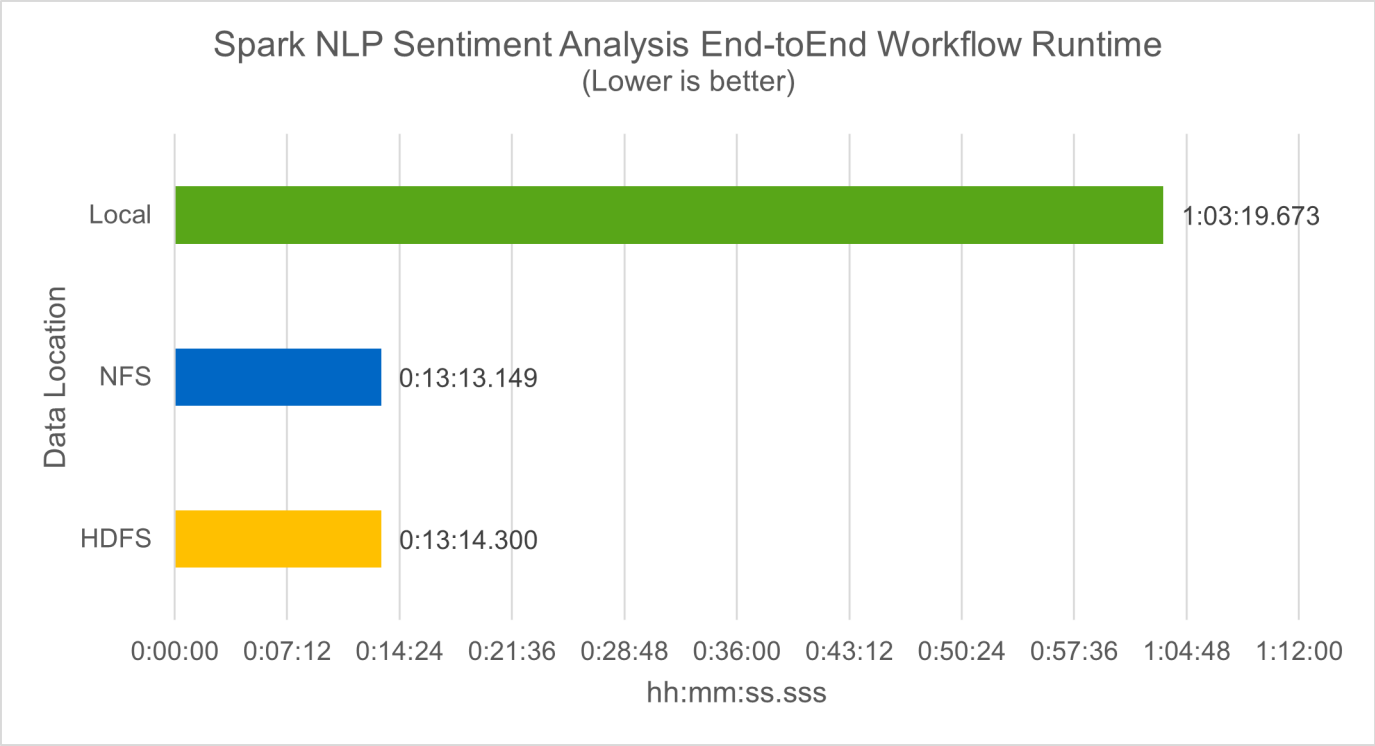
aufgeführt, in Prozent.

Stimmungsanteil	Alle 10 Unternehmen	AAPL	AMD	AMZN	CSCO	GOOGL	INTC	MSFT	NVDA
Positiv	10.13 % erzielt	18.06 % erzielt	8.69 % erzielt	5.24 % erzielt	9.07 % erzielt	12.08 % erzielt	11.44 % erzielt	13.25 % erzielt	6.23 % erzielt
Neutral	87.17 % erzielt	79.02 % erzielt	88.82 % erzielt	91.87 % erzielt	88.42 % erzielt	86.50 % erzielt	84.65 % erzielt	83.77 % erzielt	92.44 % erzielt
Negativ	2.43 % erzielt	2.92 % erzielt	2.49 % erzielt	1.52 % erzielt	2.51 % erzielt	1.42 % erzielt	3.91 % erzielt	2.96 % erzielt	1.33 % erzielt
Ohne Kategorie	0.27 % erzielt	0 %	0 %	1.37 % erzielt	0 %	0 %	0 %	0.01 % erzielt	0 %

Hinsichtlich der Workflow-Laufzeit wurde gegenüber der Workflow-Laufzeit eine deutliche 4,78-fache Verbesserung erzielt `local` Modus zu einer verteilten Umgebung in HDFS und eine weitere Verbesserung von 0.14 % durch Nutzung von 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
```

Wie in der folgenden Abbildung dargestellt, verbesserte Daten- und Modellparallelität die Datenverarbeitung und die Inferenzgeschwindigkeit des verteilten TensorFlow-Modells. Der Datenspeicherort in NFS führte zu einer etwas besseren Laufzeit, da der Workflow-Engpass das Herunterladen von vortrainierten Modellen ist. Wenn wir die Datensatzgröße der Transkripte erhöhen, ist der Vorteil von NFS offensichtlicher.



Verteiltes Training mit Horovod Leistung

Mit dem folgenden Befehl wurden Laufzeitinformationen und eine Protokolldatei in unserem Spark-Cluster unter Verwendung einer einzigen erzeugt `master` Node mit 160 Ausführenden mit jeweils einem Kern. Der Ausführende-Speicher wurde auf 5 GB beschränkt, um einen Fehler außerhalb des Arbeitsspeichers zu vermeiden. Siehe Abschnitt "„Python-Skripte für jeden größeren Anwendungsfall“" Weitere Details zur Datenverarbeitung, Modellschulung und Berechnung der Modellgenauigkeit finden Sie 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
```

Die daraus resultierende Laufzeit mit zehn Trainingsepochen war wie folgt:

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

Es dauerte mehr als 43 Minuten, Eingabedaten zu verarbeiten, ein DNN-Modell zu trainieren, die Genauigkeit zu berechnen und TensorFlow Checkpoints und eine CSV-Datei für Vorhersageergebnisse zu erstellen. Wir limitierten die Anzahl der Trainingsepochen auf 10, die in der Praxis oft auf 100 gesetzt werden, um eine zufriedenstellende Modellgenauigkeit zu gewährleisten. Die Trainingszeit wird in der Regel linear mit der Anzahl der Epochen skaliert.

Als nächstes verwendeten wir die vier Worker Nodes, die im Cluster verfügbar sind, und führten das gleiche Skript in aus `yarn` Modus mit Daten 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
```

Die daraus resultierende Laufzeit wurde wie folgt verbessert:

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

Mit Horovods Modell und Datenparallelität in Spark haben wir eine 5,29fache Laufzeitgeschwindigkeit von gesehen `yarn` Vs `local` Modus mit zehn Trainingsepochen. Dies wird in der folgenden Abbildung mit den Legenden dargestellt `HDFS` Und `Local`. Das zugrunde liegende TensorFlow DNN-Modelltraining kann mit GPUs weiter beschleunigt werden, falls verfügbar. Wir planen diese Tests durchzuführen und die Ergebnisse in einem zukünftigen technischen Bericht zu veröffentlichen.

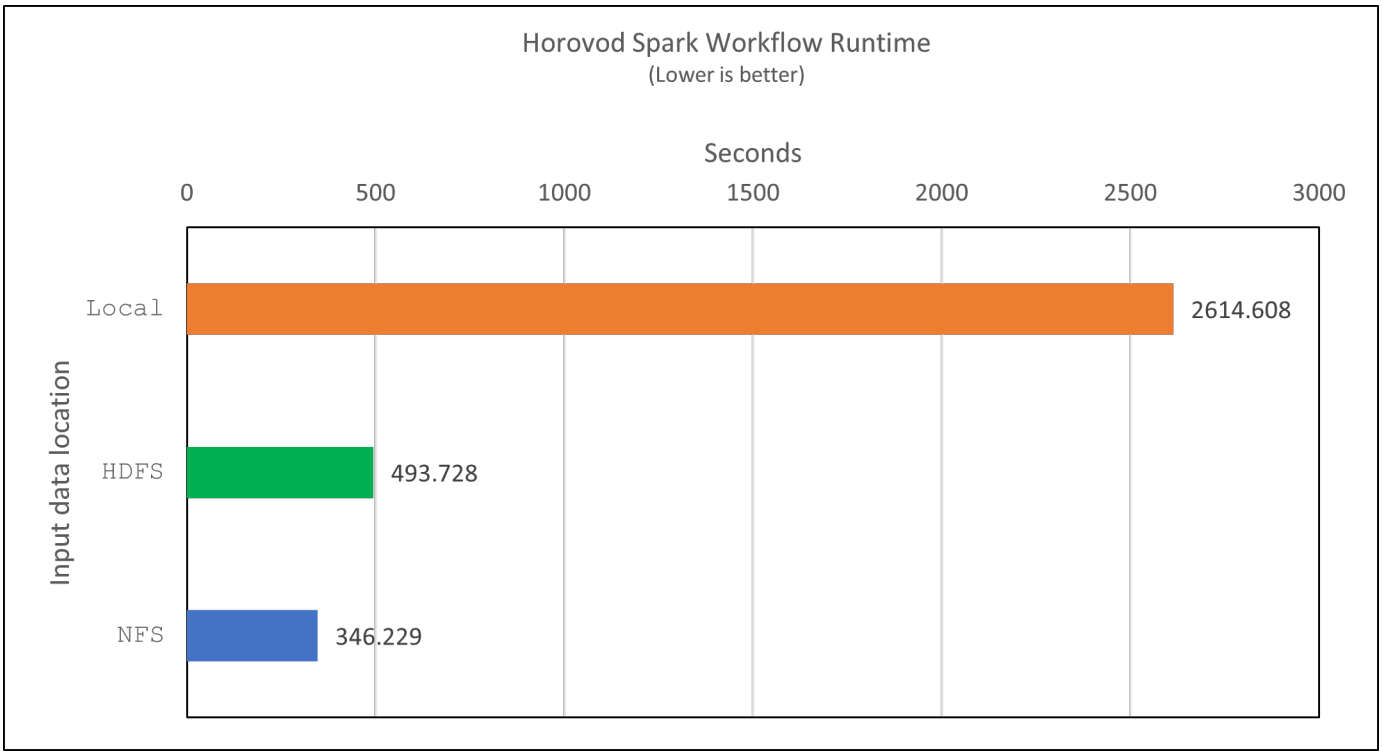
Bei unserem nächsten Test wurden die Laufzeiten mit Eingabedaten im NFS verglichen und HDFS. Das NFS-Volume auf der AFF A800 wurde angehängt `/sparkdemo/horovod` Über die fünf Nodes (ein Master, vier Mitarbeiter) in unserem Spark-Cluster verteilt Wir führten einen ähnlichen Befehl aus wie bei früheren Tests, mit `--data- dir` Parameter, der jetzt auf NFS-Mount zeigt:

```
(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
```

Die daraus resultierende Laufzeit mit NFS war wie folgt:

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

Wie in der folgenden Abbildung dargestellt, gab es eine weitere 1,43x Geschwindigkeitsnachbildung. Da ein NetApp All-Flash-Storage an seinen Cluster angeschlossen ist, können Kunden von den Vorteilen einer schnellen Datenübertragung und -Verteilung für Horovod Spark-Workflows profitieren. So wird 7.55-mal schneller als auf einem einzelnen Node ausgeführt.



Deep-Learning-Modelle für die Vorhersageleistung von CTR

Für Empfehlungssysteme, die zur Maximierung der CTR entwickelt wurden, müssen Sie lernen, anspruchsvolle Funktionsinteraktionen hinter Benutzerverhalten zu erlernen, die mathematisch von niedrig bis hoch berechnet werden können. Interaktionen zwischen Low-Order- und High-Order-Funktionen sollten für ein gutes Deep-Learning-Modell genauso wichtig sein, ohne das eine oder andere per Bicken zu tun. Deep Factorisation Machine (DeepFM), ein maschinell basiertes neuronales Netz zur Faktorisierung, kombiniert für das Feature Learning in einer neuen neuronalen Netzwerkarchitektur Factorisationsmaschinen für Empfehlung und Deep Learning.

Obwohl herkömmliche Factorisierungsmaschinen paarweise Interaktionen als inneres Produkt latenter Vektoren zwischen Features modellieren und theoretisch Informationen in hoher Reihenfolge erfassen können, verwenden maschinelle Lernende in der Regel aufgrund der hohen Rechen- und Speicherkomplexität nur Interaktionen in zweiter Reihenfolge. Deep Neural Network Varianten wie Googles "Wide Deep Modelle" Zum anderen lernt man in einer hybriden Netzwerkstruktur anspruchsvolle Feature-Interaktionen, indem man ein linear weites Modell mit einem tiefen Modell kombiniert.

Es gibt zwei Eingänge zu diesem Wide & Deep Model, einen für das zugrunde liegende breite Modell und den anderen für die Tiefe, der letzte Teil von denen noch erfordert fachkundige Feature-Engineering und macht damit die Technik weniger generierbar für andere Domains. Im Gegensatz zum Wide & Deep Model lässt sich DeepFM ohne jede Funktionstechnik effizient mit RAW-Funktionen Schulen, da sein breites Teil und das tiefe Teil denselben Eingang und den Einbettungsvektor teilen.

Wir haben den Criteo bearbeitet `train.txt` (11 GB) Datei in einer CSV-Datei namens `ctr_train.csv` In einem NFS-Mount gespeichert `/sparkdemo/tr-4570-data` Wird verwendet `run_classification_criteo_spark.py` Aus dem Abschnitt "„Python-Skripte für jeden größeren Anwendungsfall.“" Innerhalb dieses Skripts die Funktion `process_input_file` Führt mehrere String-Methoden durch, um Tabs zu entfernen und einzufügen `','` Als Trennzeichen und `'\n'` Als neue Zeile. Beachten Sie, dass Sie nur das Original verarbeiten müssen `train.txt` Einmal, damit der Code-Block als Kommentare angezeigt wird.

Für die folgenden Tests der verschiedenen DL-Modelle haben wir verwendet `ctr_train.csv` Als Eingabedatei. Bei nachfolgenden Testläufen wurde die CSV-Eingabedatei in einen Spark DataFrame mit einem Schema mit einem Feld von eingelesen `'label'`, Ganzzahlige dichte Funktionen `['I1', 'I2', 'I3', …, 'I13']`, Und spärliche Merkmale `['C1', 'C2', 'C3', …, 'C26']`. Im Folgenden `spark-submit` Befehl nimmt einen CSV-Eingang ein, trainiert DeepFM-Modelle mit 20% Teilung zur Kreuzvalidierung und wählt das beste Modell nach zehn Trainingsepochen, um Vorhersagegenauigkeit auf dem Prüfsatz zu berechnen:

```
(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
```

Beachten Sie, dass seit der Datendatei `ctr_train.csv` Ist über 11 GB, müssen Sie eine ausreichende einstellen `spark.driver.maxResultSize` Größer als die Datensatzgröße, um Fehler zu vermeiden.

```
 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 der obigen `SparkSession.builder` Konfiguration wurde ebenfalls aktiviert "Apache Arrow", Die einen Spark DataFrame in einen Pandas DataFrame mit dem konvertiert `df.toPandas()` Methode.

```
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.
```

Nach der zufälligen Aufteilung befinden sich im Trainingdatensatz über 36 M und 9 M-Muster im Testsatz:

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

Da sich dieser technische Bericht auf CPU-Tests ohne GPUs konzentriert, ist es zwingend erforderlich, dass Sie TensorFlow mit den entsprechenden Compiler-Flags erstellen. Dieser Schritt verhindert das Aufrufen von GPU-beschleunigten Bibliotheken und nutzt die Advanced Vector Extensions (AVX)- und AVX2-Anweisungen in vollem Umfang. Diese Eigenschaften sind für lineare algebraische Berechnungen wie vectorisierte Addition, Matrix-Multiplikationen innerhalb eines Vorschub-Forward oder Back-Propagation DNN-Training konzipiert. Fused Multiply Add (FMA)-Anweisung, die mit AVX2 über 256-Bit-Floating-Point-Register (FP) verfügbar ist, ist ideal für Integer-Code und Datentypen, was zu einer doppelten Geschwindigkeit führt. Für FP-Code und Datentypen erreicht AVX2 eine Beschleunigung von 8 % über 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.
```

Um TensorFlow von der Quelle zu erstellen, empfiehlt NetApp die Verwendung "Bazel". Für unsere Umgebung haben wir in der Shell-Eingabeaufforderung die folgenden Befehle zur Installation ausgeführt `dnf`, `dnf-plugins`, Und Bazel.

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

Sie müssen GCC 5 oder höher aktivieren, damit während des Builds C++17-Funktionen verwendet werden können, die von RHEL with Software Collections Library (SCL) bereitgestellt werden. Die folgenden Befehle werden installiert `devtoolset` Und GCC 11.2.1 auf unserem RHEL 7.9 Cluster:

```
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
```

Beachten Sie, dass die letzten beiden Befehle aktiviert sind `devtoolset-11`, Die verwendet `/opt/rh/devtoolset-11/root/usr/bin/gcc` (GCC 11.2.1). Stellen Sie auch sicher, dass Sie `git` Version ist größer als 1.8.3 (dies kommt mit RHEL 7.9). Weitere Informationen finden Sie hier "Artikel" Für Aktualisierung `git` Bis 2.24.1.

Wir gehen davon aus, dass Sie die neueste TensorFlow Master-Repo bereits geklont haben. Erstellen Sie dann ein `workspace` Verzeichnis mit einem `WORKSPACE` Datei zum Erstellen von TensorFlow aus der Quelle mit AVX, AVX2 und FMA Führen Sie die aus `configure` Datei und geben Sie den richtigen Python-Binärspeicherort an. "CUDA" Ist für unsere Tests deaktiviert, da wir keine GPU verwendet haben. A `.bazelrc` Die Datei wird entsprechend Ihren Einstellungen erzeugt. Außerdem haben wir die Datei bearbeitet und gesetzt `build --define=no_hdfs_support=false` Um die HDFS-Unterstützung zu aktivieren. Siehe `.bazelrc` Im Abschnitt "„Python-Skripte für jeden Hauptanwendungsfall“," Für eine vollständige Liste von Einstellungen und Flags.

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

Nachdem Sie TensorFlow mit den richtigen Flags erstellt haben, führen Sie das folgende Skript aus, um den Datensatz Criteo Display Ads zu bearbeiten, ein DeepFM-Modell zu trainieren und den Bereich unter der Receiver Operating Characteristic Curve (ROC AUC) aus den Vorhersagewerten zu berechnen.

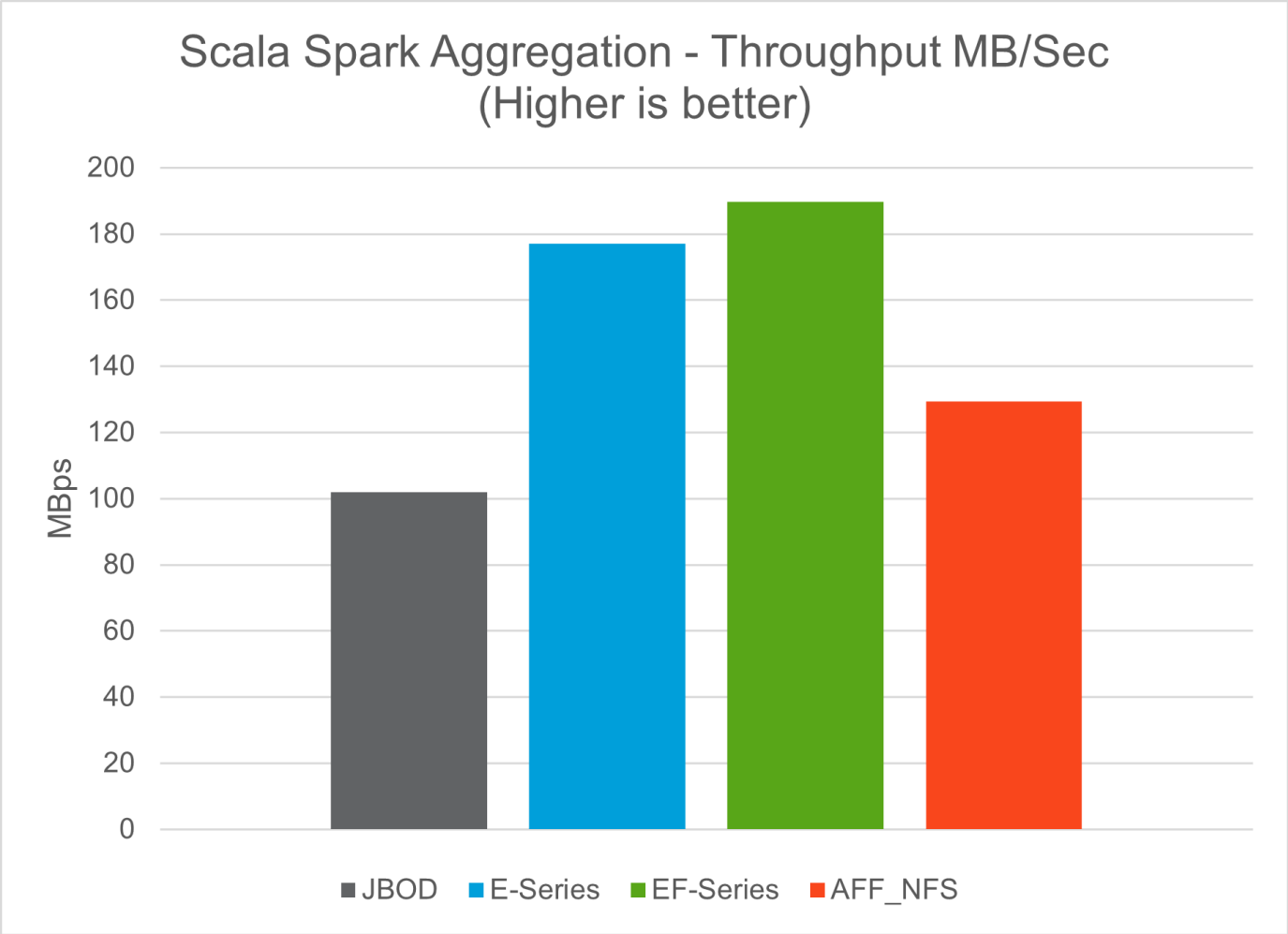
```
(base) [root@n138 examples]# ~/anaconda3/bin/spark-submit
--master yarn
--executor-memory 15g
--executor-cores 1
--num-executors 160
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py
--data-dir file:///sparkdemo/tr-4570-data
> . /run_classification_criteo_spark_nfs.log 2>&1
```

Nach zehn Epochen des Trainings erhielten wir die AUC-Punktzahl auf dem Testdatensatz:

```
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
```

Ähnlich wie bei früheren Anwendungsfällen haben wir die Spark-Workflow-Laufzeit mit Daten an

verschiedenen Standorten verglichen. Die folgende Abbildung zeigt einen Vergleich der Deep-Learning-CTR-Vorhersage für eine Spark-Workflows-Laufzeit.



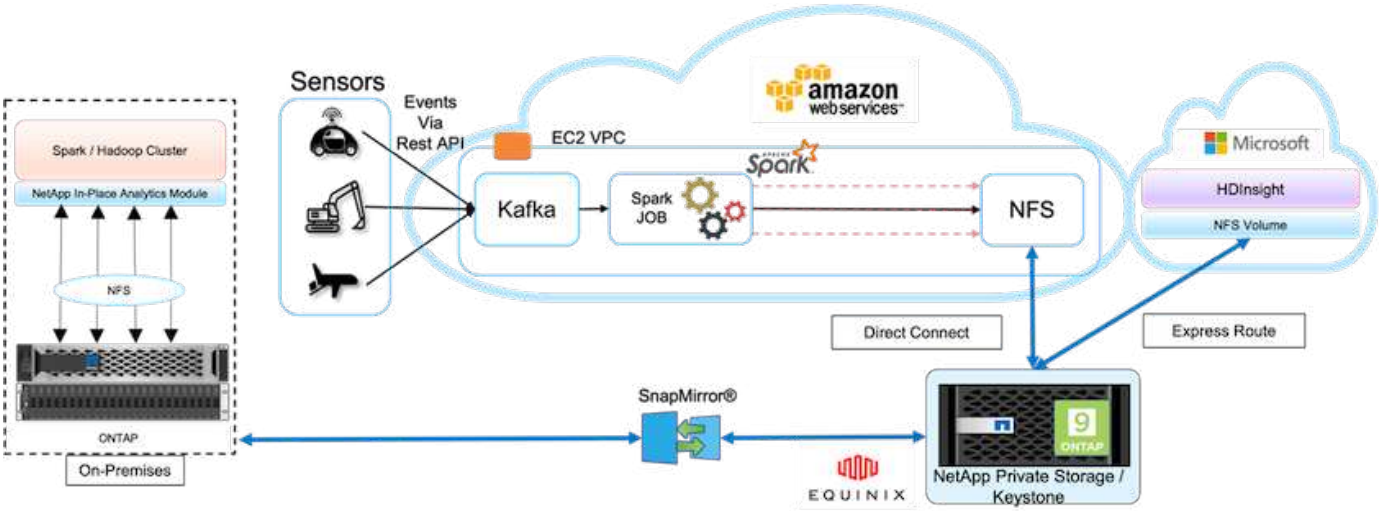
Hybrid Cloud-Lösung

Ein modernes Enterprise-Datacenter ist eine Hybrid Cloud, die diverse verteilte Infrastrukturumgebungen über eine kontinuierliche Datenmanagementebene mit einem konsistenten Betriebsmodell verbindet – vor Ort und/oder in mehreren Public Clouds. Um die Vorteile einer Hybrid Cloud optimal nutzen zu können, müssen Sie die Daten nahtlos zwischen Ihren lokalen und Multi-Cloud-Umgebungen verschieben können, ohne dass Datenkonvertierung oder Applikationsrefakturierung erforderlich ist.

Die Kunden haben angegeben, dass sie den Weg in die Hybrid Cloud entweder durch das verlagern von sekundärem Storage in die Cloud beginnen. Dies ist u. a. für Datensicherung oder die Verlagerung weniger geschäftskritischer Workloads wie Applikationsentwicklung und DevOps in die Cloud möglich. Dann wechseln sie zu geschäftskritischeren Workloads. Zu den beliebtesten Hybrid-Cloud-Workloads gehören Web- und Content-Hosting, DevOps und Applikationsentwicklung, Datenbanken, Analysen und Container-Applikationen. Die Komplexität, Kosten und Risiken von Enterprise-KI-Projekten haben bislang die Einführung der KI von der Testphase in die Produktion gehindert.

Mit einer Hybrid-Cloud-Lösung von NetApp profitieren Kunden von integrierten Tools für Sicherheit, Daten-Governance und Compliance mit einer einzigen Konsole für Daten- und Workflow-Management in verteilten

Umgebungen. Gleichzeitig können sie die TCO basierend auf ihrer Nutzung optimieren. Die folgende Abbildung zeigt eine beispielhafte Lösung eines Cloud-Service-Partners, der für die Bereitstellung von Multi-Cloud-Konnektivität für Big Data-Analysedaten des Kunden beauftragt ist.



In diesem Szenario werden die in AWS empfangenen IoT-Daten aus verschiedenen Quellen an einem zentralen Standort in NetApp Private Storage (NPS) gespeichert. Der NPS Storage ist mit Spark oder Hadoop Clustern in AWS und Azure verbunden, sodass Big-Data-Analyseapplikationen in diversen Clouds ausgeführt werden, die auf dieselben Daten zugreifen. Zu den wesentlichen Anforderungen und Herausforderungen dieses Anwendungsfalls gehören:

- Kunden möchten Analysejobs in denselben Daten mithilfe mehrerer Clouds ausführen.
- Daten müssen von unterschiedlichen Quellen wie On-Premises- und Cloud-Umgebungen über unterschiedliche Sensoren und Hubs empfangen werden.
- Die Lösung muss effizient und kosteneffektiv sein.
- Die größte Herausforderung ist der Aufbau einer kostengünstigen und effizienten Lösung, die Hybrid-Analyseservices für verschiedene On-Premises- und Cloud-Umgebungen bietet.

Unsere Datensicherungs- und Multi-Cloud-Konnektivitätslösung behebt die Probleme bei Cloud-Analyseapplikationen über diverse Hyperscaler hinweg. Wie in der Abbildung oben gezeigt, werden die Daten von Sensoren gestreamt und über Kafka in den AWS Spark-Cluster aufgenommen. Die Daten werden in einem NFS-Share in NPS gespeichert, der sich außerhalb des Cloud-Providers in einem Equinix Datacenter befindet.

Da NetApp NPS über Direct Connect und Express Route Verbindungen mit Amazon AWS und Microsoft Azure verbunden ist, können Kunden das in-Place-Analysemodul für den Zugriff auf Daten von Amazon- und AWS-Analyse-Clustern nutzen. Da sowohl On-Premises- als auch NPS-Storage auf ONTAP Software ausgeführt wird, "SnapMirror" Die NPS Daten können in das On-Premises-Cluster gespiegelt werden und dabei Hybrid-Cloud-Analysen für On-Premises-Systeme und diverse Clouds bieten.

Um die beste Performance zu erzielen, empfiehlt NetApp normalerweise, mehrere Netzwerkschnittstellen und direkte Verbindung oder Express Routen zu verwenden, um auf die Daten von Cloud-Instanzen zuzugreifen. Wir bieten unter anderem eine Data Mover-Lösung an "XCP" Und "BlueXP Copy und Sync" Damit Kunden applikationsspezifische, sichere und kostengünstige Hybrid-Cloud-Spark-Cluster erstellen können,

Python-Skripte für jeden größeren Anwendungsfall

Die folgenden drei Python-Skripte entsprechen den drei getesteten Hauptanwendungsfällen. Zunächst einmal `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}")
```

Das zweite Skript lautet `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()
```

Das dritte Skript ist `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))
```

Schlussfolgerung

In diesem Dokument befassen wir uns mit der Apache Spark Architektur, mit Kundenanwendungsfällen und dem NetApp Storage-Portfolio, bezogen auf Big Data, moderne Analysen sowie KI, ML und DL. Bei unseren Performance-Validierungstests, die auf branchenüblichen Benchmark-Tools und Kundennachfrage basieren, haben die NetApp Spark-Lösungen eine überragende Performance im Vergleich zu nativen Hadoop Systemen gezeigt. Anhand einer Kombination aus in diesem Bericht vorgestellten Kundenanwendungsfällen und Performance-Ergebnissen können Sie eine passende Spark-Lösung für Ihre Implementierung auswählen.

Wo Sie weitere Informationen finden

Folgende Referenzen wurden in dieser TR verwendet:

- Apache Spark Architektur und Komponenten

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

- Anwendungsfälle für Apache Spark

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

- Apache Herausforderungen

  "http://www.infoworld.com/article/2897287/big-data/5-reasons-to-turn-to-spark-for-big-data-analytics.html"

- Funke NLP

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

- BERT

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

- Deep and Cross Network for Ad Click Predictions

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

- FlexGroup

  "http://www.netapp.com/us/media/tr-4557.pdf"

- Streaming-ETL

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

- NetApp E-Series Lösungen für Hadoop

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

- Sentiment-Analyse aus Kundenkommunikation mit NetApp KI

  "https://docs.netapp.com/us-en/netapp-solutions/pdfs/sidebar/Sentiment_analysis_with_NetApp_AI.pdf"

- NetApp Lösungen für moderne Datenanalysen

  "https://docs.netapp.com/us-en/netapp-solutions/data-analytics/index.html"

- 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 Copy und Sync

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

- DataOps-Toolkit

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