Apache Spark를 위한 NetApp 스토리지 솔루션

NetApp artificial intelligence solutions

NetApp
August 18, 2025

목차

Apache Spark를 위한 NetApp 스토리지 솔루션

TR-4570: Apache Spark용 NetApp 스토리지 솔루션: 아키텍처, 사용 사례 및 성능 결과

Rick Huang, Karthikeyan Nagalingam, NetApp

이 문서는 Apache Spark 아키텍처, 고객 사용 사례, 빅데이터 분석 및 인공 지능(AI)과 관련된 NetApp 스토리지 포트폴리오에 중점을 둡니다. 또한 일반적인 Hadoop 시스템에 대해 업계 표준 AI, 머신 러닝(ML), 딥 러닝(DL) 도구를 사용하여 다양한 테스트 결과를 제시하여 적절한 Spark 솔루션을 선택할 수 있도록 도와줍니다. 시작하려면 Spark 아키텍처, 적절한 구성 요소, 두 가지 배포 모드(클러스터 및 클라이언트)가 필요합니다.

이 문서에서는 구성 문제를 해결하기 위한 고객 사용 사례를 제공하고 Spark를 활용한 빅데이터 분석 및 AI, ML, DL과 관련된 NetApp 스토리지 포트폴리오에 대한 개요를 설명합니다. 그런 다음 Spark 관련 사용 사례와 NetApp Spark 솔루션 포트폴리오에서 파생된 테스트 결과로 마무리합니다.

고객의 과제

이 섹션에서는 소매, 디지털 마케팅, 뱅킹, 개별 제조, 공정 제조, 정부 및 전문 서비스와 같은 데이터 증가 산업에서 빅데이터 분석 및 AI/ML/DL과 관련된 고객 과제에 중점을 둡니다.

예측할 수 없는 성능

기존 Hadoop 배포에서는 일반적으로 상용 하드웨어를 사용합니다. 성능을 향상시키려면 네트워크, 운영 체제, Hadoop 클러스터, Spark와 같은 생태계 구성 요소, 하드웨어를 조정해야 합니다. 각 계층을 조정하더라도 Hadoop은 사용자 환경에서 고성능을 위해 설계되지 않은 상용 하드웨어에서 실행되기 때문에 원하는 성능 수준을 달성하는 것이 어려울 수 있습니다.

미디어 및 노드 오류

일반적인 조건에서도 상용 하드웨어는 고장이 나기 쉽습니다. 데이터 노드의 디스크 하나에 장애가 발생하면 Hadoop 마스터는 기본적으로 해당 노드가 비정상적이라고 간주합니다. 그런 다음 네트워크를 통해 해당 노드의 특정 데이터를 복제본에서 정상 노드로 복사합니다. 이 프로세스는 Hadoop 작업에 대한 네트워크 패킷 속도를 늦춥니다. 그런 다음 클러스터는 데이터를 다시 복사하고, 건강에 해로운 노드가 건강한 상태로 돌아오면 과도하게 복제된 데이터를 제거해야 합니다.

Hadoop 공급업체 종속

Hadoop 배포업체는 자체 버전을 갖춘 자체 Hadoop 배포판을 보유하고 있어 고객이 해당 배포판에 종속됩니다. 그러나 많은 고객은 특정 Hadoop 배포판에 구애받지 않는 메모리 내 분석에 대한 지원을 요구합니다. 그들은 배포 방식을 변경하면서도 분석 결과를 그대로 적용할 수 있는 자유가 필요합니다.

두 개 이상의 언어에 대한 지원 부족

고객은 종종 작업을 실행하기 위해 MapReduce Java 프로그램 외에도 여러 언어에 대한 지원을 요구합니다. SQL 및 스크립트와 같은 옵션은 답변을 얻는 데 더 많은 유연성을 제공하고, 데이터를 구성하고 검색하는 데 더 많은 옵션을 제공하며, 분석 프레임워크로 데이터를 이동하는 더 빠른 방법을 제공합니다.

사용의 어려움

얼마 전부터 사람들은 Hadoop을 사용하기 어렵다고 불평해 왔습니다. Hadoop은 새로운 버전이 나올 때마다 더 간단해지고 강력해졌지만, 이러한 비판은 여전히 존재합니다. Hadoop을 사용하려면 Java와 MapReduce 프로그래밍 패턴을 이해해야 하는데, 이는 데이터베이스 관리자와 기존 스크립팅 기술을 보유한 사람들에게는 어려운 일입니다.

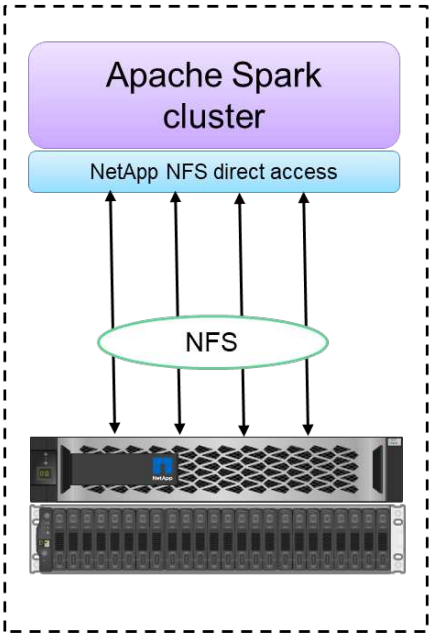
복잡한 프레임워크와 도구

기업의 AI 팀은 여러 가지 과제에 직면합니다. 전문적인 데이터 과학 지식이 있더라도, 다양한 배포 생태계와 애플리케이션에 맞는 도구와 프레임워크를 하나에서 다른 하나로 간단히 변환할 수는 없습니다. 데이터 과학 플랫폼은 Spark 기반으로 구축된 해당 빅데이터 플랫폼과 원활하게 통합되어야 하며, 데이터 이동이 쉽고, 재사용 가능한 모델, 즉시 사용 가능한 코드, 프로토타입 제작, 검증, 버전 관리, 공유, 재사용 및 모델의 신속한 프로덕션 배포를 위한 모범 사례를 지원하는 도구가 있어야 합니다.

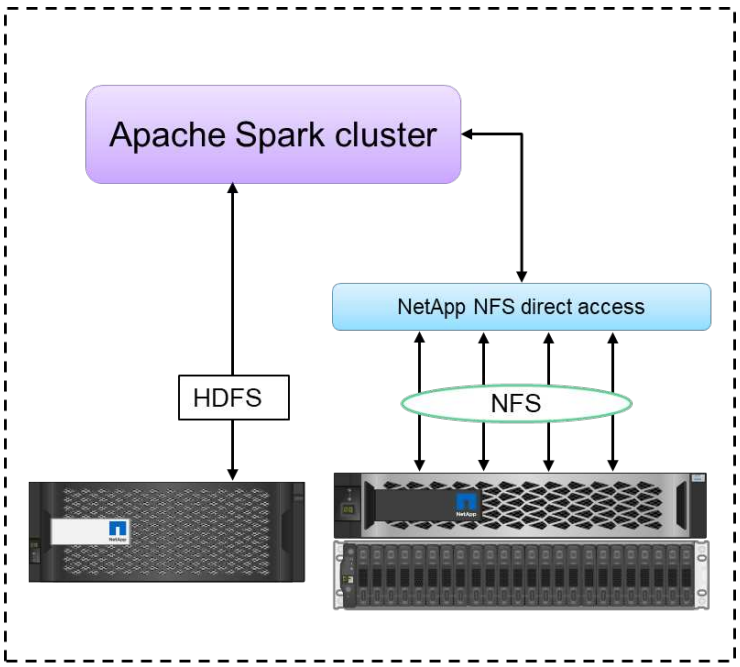
왜 **NetApp** 선택해야 하나요?

NetApp 다음과 같은 방법으로 Spark 경험을 개선할 수 있습니다.

- NetApp NFS 직접 액세스(아래 그림 참조)를 통해 고객은 데이터를 이동하거나 복사하지 않고도 기존 또는 새로운 NFSv3 또는 NFSv4 데이터에 대한 빅데이터 분석 작업을 실행할 수 있습니다. 이를 통해 여러 개의 데이터 사본이 생성되는 것을 방지하고 소스와 데이터를 동기화할 필요성이 없어집니다.
- 더 효율적인 저장과 더 적은 서버 복제. 예를 들어, NetApp E-Series Hadoop 솔루션은 3개가 아닌 2개의 데이터 복제본이 필요하고, FAS Hadoop 솔루션은 데이터 소스는 필요하지만 데이터 복제나 사본은 필요하지 않습니다. NetApp 스토리지 솔루션은 서버 간 트래픽도 줄어듭니다.
- 드라이브 및 노드 장애 발생 시 Hadoop 작업 및 클러스터 동작이 개선되었습니다.
- 더 나은 데이터 수집 성능.



Configuration 1: NFS as primary storage



Configuration 2: HDFS and NFS in single Spark cluster

예를 들어, 금융 및 의료 분야에서는 데이터를 한 장소에서 다른 장소로 이동하려면 법적 의무를 충족해야 하는데, 이는 쉬운 일이 아닙니다. 이 시나리오에서 NetApp NFS 직접 액세스는 원래 위치에서 재무 및 의료 데이터를 분석합니다. 또 다른 주요 이점은 NetApp NFS 직접 액세스를 사용하면 기본 Hadoop 명령을 사용하고 NetApp 의 풍부한 데이터 관리

포트폴리오를 통해 데이터 보호 워크플로를 활성화하여 Hadoop 데이터를 보호하는 것이 간소화된다는 것입니다.

NetApp NFS 직접 액세스는 Hadoop/Spark 클러스터에 대해 두 가지 종류의 배포 옵션을 제공합니다.

- 기본적으로 Hadoop 또는 Spark 클러스터는 데이터 저장을 위해 Hadoop 분산 파일 시스템(HDFS)과 기본 파일 시스템을 사용합니다. NetApp NFS 직접 액세스를 통해 기본 HDFS를 NFS 스토리지로 대체하여 기본 파일 시스템으로 사용할 수 있으므로 NFS 데이터에 대한 직접 분석이 가능합니다.
- 또 다른 배포 옵션으로, NetApp NFS 직접 액세스는 단일 Hadoop 또는 Spark 클러스터에서 HDFS와 함께 추가 스토리지로 NFS를 구성하는 것을 지원합니다. 이 경우, 고객은 NFS 내보내기를 통해 데이터를 공유하고 HDFS 데이터와 함께 동일한 클러스터에서 해당 데이터에 액세스할 수 있습니다.

NetApp NFS 직접 액세스를 사용하면 다음과 같은 주요 이점이 있습니다.

- 현재 위치에서 데이터를 분석하면 HDFS와 같은 Hadoop 인프라로 분석 데이터를 이동하는 데 드는 시간과 성능이 많이 소요되는 작업을 방지할 수 있습니다.
- 복제본 수를 3개에서 1개로 줄입니다.
- 사용자가 컴퓨팅과 스토리지를 분리하여 독립적으로 확장할 수 있도록 지원합니다.
- ONTAP 의 풍부한 데이터 관리 기능을 활용하여 기업 데이터 보호를 제공합니다.
- Hortonworks 데이터 플랫폼 인증.
- 하이브리드 데이터 분석 배포를 활성화합니다.
- 동적 멀티스레드 기능을 활용하여 백업 시간을 줄입니다.

보다"TR-4657: NetApp 하이브리드 클라우드 데이터 솔루션 - 고객 사용 사례 기반 Spark 및 Hadoop" Hadoop 데이터 백업, 클라우드에서 온프레미스로의 백업 및 재해 복구, 기존 Hadoop 데이터에 대한 DevTest 지원, 데이터 보호 및 멀티클라우드 연결, 분석 워크로드 가속화를 지원합니다.

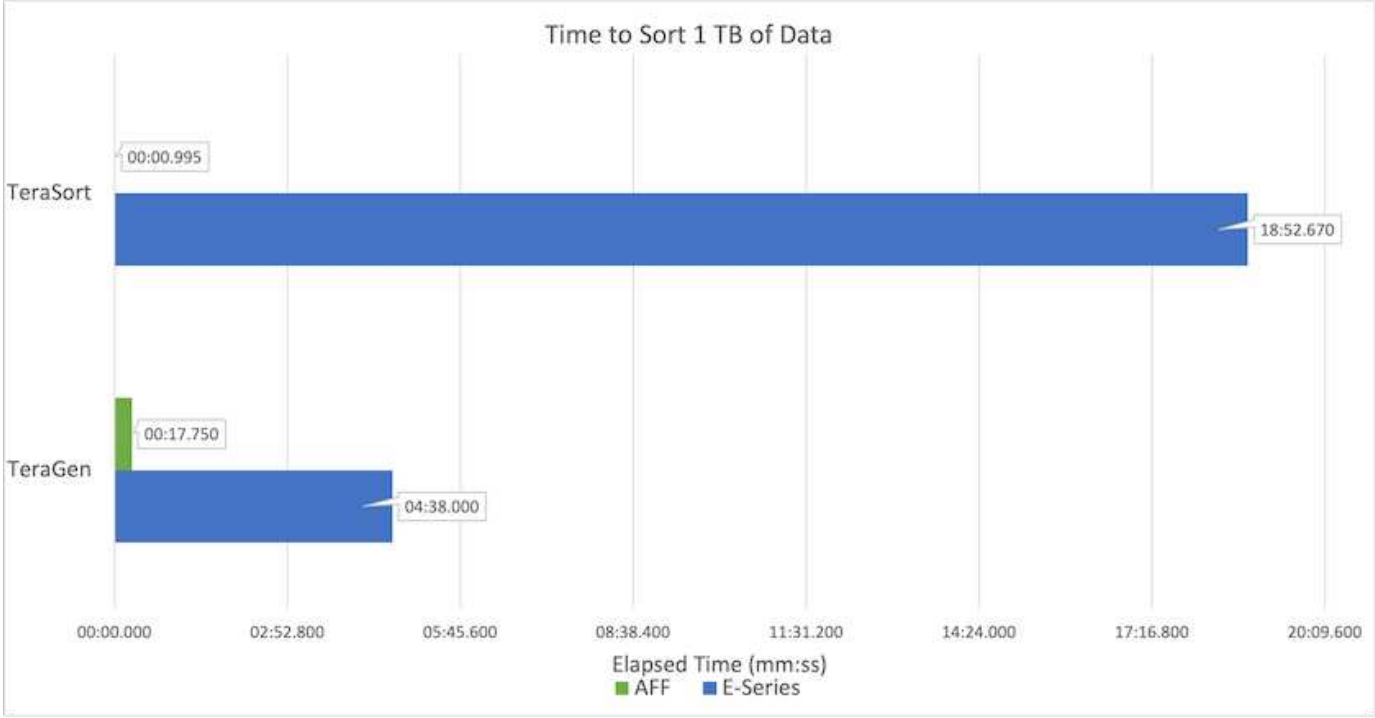
다음 섹션에서는 Spark 고객에게 중요한 저장 기능에 대해 설명합니다.

스토리지 계층화

Hadoop 스토리지 계층화를 사용하면 스토리지 정책에 따라 다양한 스토리지 유형에 파일을 저장할 수 있습니다. 저장 유형에는 다음이 포함됩니다. `hot` , `cold` , `warm` , `all_ssd` , `one_ssd` , 그리고 `lazy_persist` .

우리는 SSD와 SAS 드라이브가 서로 다른 스토리지 정책을 사용하는 NetApp AFF 스토리지 컨트롤러와 E-Series 스토리지 컨트롤러에서 Hadoop 스토리지 계층화의 검증을 수행했습니다. AFF-A800이 있는 Spark 클러스터에는 4개의 컴퓨팅 워커 노드가 있는 반면, E-Series가 있는 클러스터에는 8개가 있습니다. 이는 주로 솔리드 스테이트 드라이브(SSD)와 하드 드라이브 디스크(HDD)의 성능을 비교하기 위한 것입니다.

다음 그림은 Hadoop SSD에 대한 NetApp 솔루션의 성능을 보여줍니다.



- 기본 NL-SAS 구성에서는 8개의 컴퓨팅 노드와 96개의 NL-SAS 드라이브를 사용했습니다. 이 구성에서는 4분 38초 만에 1TB의 데이터가 생성되었습니다. 보다 "Hadoop을 위한 TR-3969 NetApp E-Series 솔루션" 클러스터 및 스토리지 구성에 대한 자세한 내용은 다음을 참조하세요.
- TeraGen을 사용하면 SSD 구성이 NL-SAS 구성보다 15.66배 빠르게 1TB의 데이터를 생성할 수 있습니다. 게다가 SSD 구성은 컴퓨팅 노드 수와 디스크 드라이브 수를 절반으로 줄였습니다(총 24개의 SSD 드라이브). 작업 완료 시간을 기준으로 볼 때 NL-SAS 구성보다 거의 두 배나 빨랐습니다.
- TeraSort를 사용하면 SSD 구성이 NL-SAS 구성보다 1TB의 데이터를 1138.36배 더 빠르게 정렬할 수 있습니다. 게다가 SSD 구성은 컴퓨팅 노드 수와 디스크 드라이브 수를 절반으로 줄였습니다(총 24개의 SSD 드라이브). 따라서 드라이브당으로 따지면 NL-SAS 구성보다 약 3배 더 빠릅니다.
- 결론은 회전 디스크에서 올플래시로 전환하면 성능이 향상된다는 것입니다. 병목 현상은 컴퓨팅 노드의 수가 아니었습니다. NetApp의 올플래시 스토리지를 사용하면 런타임 성능이 원활하게 확장됩니다.
- NFS를 사용하면 데이터가 모두 풀링된 것과 기능적으로 동일하므로 작업 부하에 따라 컴퓨팅 노드 수를 줄일 수 있습니다. Apache Spark 클러스터 사용자는 컴퓨팅 노드 수를 변경할 때 수동으로 데이터를 재조정할 필요가 없습니다.

성능 확장 - 확장

AFF 솔루션에서 Hadoop 클러스터로부터 더 많은 컴퓨팅 성능이 필요한 경우 적절한 수의 스토리지 컨트롤러가 있는 데이터 노드를 추가할 수 있습니다. NetApp 워크로드 특성에 따라 스토리지 컨트롤러 어레이당 4개의 데이터 노드로 시작하여 스토리지 컨트롤러당 8개의 데이터 노드로 늘릴 것을 권장합니다.

AFF 와 FAS 현장 분석에 적합합니다. 컴퓨팅 요구 사항에 따라 노드 관리자를 추가할 수 있으며, 중단 없는 운영을 통해 가동 중지 없이 필요에 따라 스토리지 컨트롤러를 추가할 수 있습니다. AFF 및 FAS 에는 NVME 미디어 지원, 효율성 보장, 데이터 감소, QoS, 예측 분석, 클라우드 계층화, 복제, 클라우드 배포 및 보안과 같은 풍부한 기능이 제공됩니다. 고객의 요구 사항을 충족할 수 있도록 NetApp 추가 라이선스 비용 없이 파일 시스템 분석, 할당량, 온박스 로드 밸런싱과 같은 기능을 제공합니다. NetApp 경쟁사보다 동시 작업 수, 대기 시간, 작업이 더 간단하고 초당 기가바이트 처리량이 더 높은 성능을 제공합니다. 또한 NetApp Cloud Volumes ONTAP 3대 주요 클라우드 공급업체 모두에서 실행됩니다.

성능 확장 - 확장

확장 기능을 사용하면 추가 저장 용량이 필요할 때 AFF, FAS 및 E-시리즈 시스템에 디스크 드라이브를 추가할 수 있습니다. Cloud Volumes ONTAP 사용하면 스토리지를 PB 수준으로 확장하는 데 두 가지 요소가 결합됩니다. 블록 스토리지에서 개체 스토리지로 자주 사용되지 않는 데이터를 계층화하고 추가 컴퓨팅 없이 Cloud Volumes ONTAP 라이선스를 스태킹합니다.

다중 프로토콜

NetApp 시스템은 SAS, iSCSI, FCP, InfiniBand, NFS를 포함하여 Hadoop 배포를 위한 대부분의 프로토콜을 지원합니다.

운영 및 지원 솔루션

이 문서에 설명된 Hadoop 솔루션은 NetApp 에서 지원됩니다. 이러한 솔루션은 주요 Hadoop 유통업체로부터도 인증을 받았습니다. 자세한 내용은 다음을 참조하세요. "호튼웍스" 사이트 및 Cloudera "인증" 그리고 "파트너" 사이트.

타겟 고객층

분석 및 데이터 과학의 세계는 IT와 비즈니스의 여러 분야에 영향을 미칩니다.

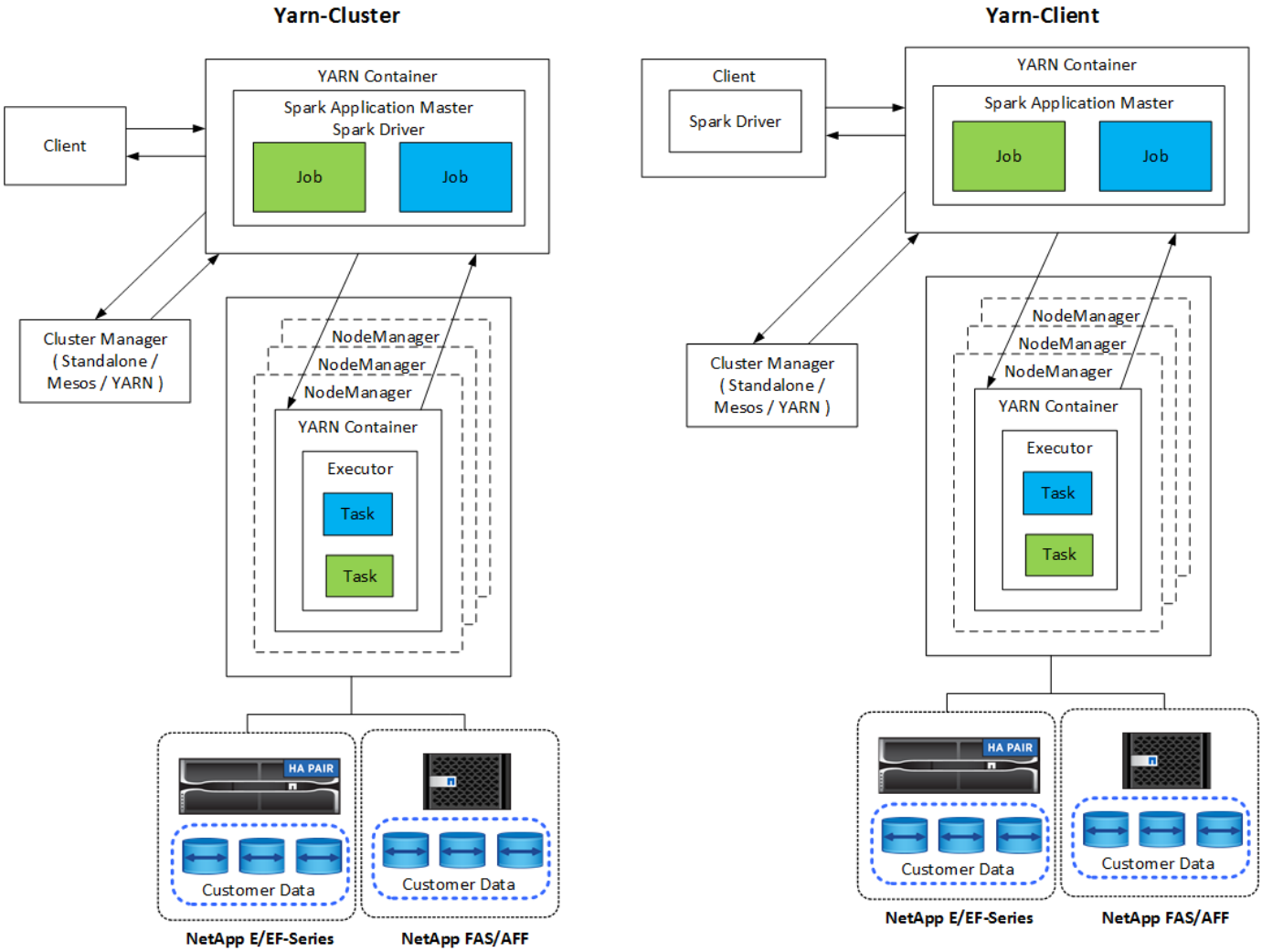
- 데이터 과학자는 자신이 선택한 도구와 라이브러리를 사용할 수 있는 유연성이 필요합니다.
- 데이터 엔지니어는 데이터가 어떻게 흐르는지, 어디에 있는지 알아야 합니다.
- DevOps 엔지니어에게는 새로운 AI 및 ML 애플리케이션을 CI 및 CD 파이프라인에 통합할 수 있는 도구가 필요합니다.
- 클라우드 관리자와 설계자는 하이브리드 클라우드 리소스를 설정하고 관리할 수 있어야 합니다.
- 비즈니스 사용자는 분석, AI, ML, DL 애플리케이션에 액세스할 수 있기를 원합니다.

이 기술 보고서에서는 NetApp AFF, E-Series, StorageGRID, NFS 직접 액세스, Apache Spark, Horovod 및 Keras가 이러한 각 역할이 비즈니스에 가치를 제공하는 데 어떻게 도움이 되는지 설명합니다.

솔루션 기술

Apache Spark는 Hadoop 분산 파일 시스템(HDFS)과 직접 작동하는 Hadoop 애플리케이션을 작성하기 위한 인기 있는 프로그래밍 프레임워크입니다. Spark는 프로덕션에 바로 사용할 수 있으며, 스트리밍 데이터 처리를 지원하며 MapReduce보다 빠릅니다. Spark는 효율적인 반복을 위해 구성 가능한 메모리 내 데이터 캐싱 기능을 갖추고 있으며, Spark 셸은 데이터를 학습하고 탐색하기 위해 대화형으로 작동합니다. Spark를 사용하면 Python, Scala 또는 Java로 애플리케이션을 만들 수 있습니다. Spark 애플리케이션은 하나 이상의 작업을 포함하는 하나 이상의 작업으로 구성됩니다.

모든 Spark 애플리케이션에는 Spark 드라이버가 있습니다. YARN-Client 모드에서는 드라이버가 클라이언트에서 로컬로 실행됩니다. YARN-클러스터 모드에서 드라이버는 애플리케이션 마스터의 클러스터에서 실행됩니다. 클러스터 모드에서는 클라이언트가 연결을 끊더라도 애플리케이션은 계속 실행됩니다.



클러스터 관리자는 세 가지가 있습니다.

- 독립형**.** 이 관리자는 Spark의 일부로, 클러스터를 쉽게 설정할 수 있도록 해줍니다.
- 아파치 메소스**.** 이는 MapReduce 및 기타 애플리케이션을 실행하는 일반 클러스터 관리자입니다.
- 하둡 **YARN.** 이는 Hadoop 3의 리소스 관리자입니다.

탄력적 분산 데이터 세트(RDD)는 Spark의 주요 구성 요소입니다. RDD는 클러스터의 메모리에 저장된 데이터에서 손실되거나 누락된 데이터를 다시 생성하고 파일에서 나오거나 프로그래밍 방식으로 생성된 초기 데이터를 저장합니다. RDD는 파일, 메모리의 데이터 또는 다른 RDD로부터 생성됩니다. Spark 프로그래밍은 변환과 동작이라는 두 가지 작업을 수행합니다. 변형은 기존 RDD를 기반으로 새로운 RDD를 생성합니다. 액션은 RDD에서 값을 반환합니다.

변환과 작업은 Spark Datasets와 DataFrames에도 적용됩니다. 데이터 세트는 RDD(강력한 타이핑, 람다 함수 사용)의 이점과 Spark SQL의 최적화된 실행 엔진의 이점을 모두 제공하는 분산된 데이터 컬렉션입니다. 데이터 세트는 JVM 객체로부터 구성된 다음 함수 변환(map, flatMap, filter 등)을 사용하여 조작될 수 있습니다. DataFrame은 이름이 지정된 열로 구성된 데이터 집합입니다. 개념적으로는 관계형 데이터베이스의 테이블이나 R/Python의 데이터 프레임과 동일합니다. DataFrames는 구조화된 데이터 파일, Hive/HBase의 테이블, 온프레미스 또는 클라우드의 외부 데이터베이스, 기존 RDD 등 다양한 소스에서 구성할 수 있습니다.

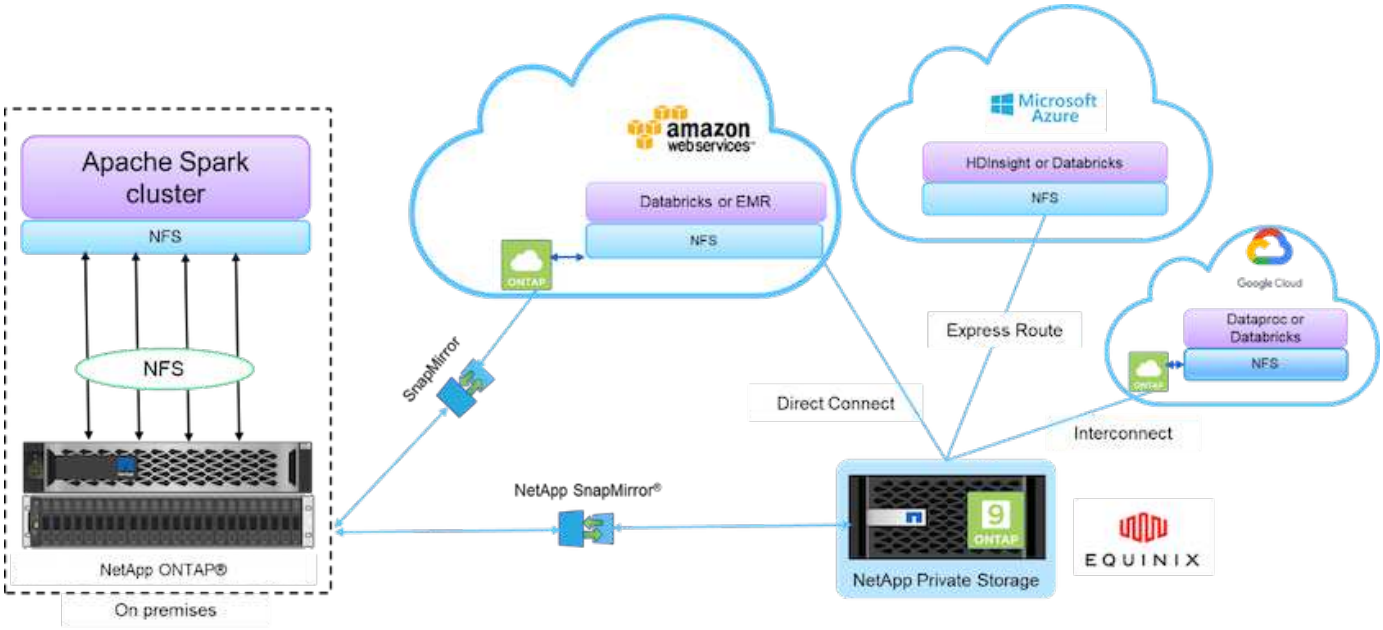
Spark 애플리케이션에는 하나 이상의 Spark 작업이 포함됩니다. 작업은 실행자에서 작업을 실행하고 실행자는 YARN 컨테이너에서 실행됩니다. 각 실행자는 단일 컨테이너에서 실행되며 실행자는 애플리케이션의 수명 동안 존재합니다. 실행자는 애플리케이션이 시작된 후에 고정되고, YARN은 이미 할당된 컨테이너의 크기를 조정하지 않습니다. 실행자는

메모리 내 데이터에서 동시에 작업을 실행할 수 있습니다.

NetApp Spark 솔루션 개요

NetApp FAS/ AFF, E-Series, Cloud Volumes ONTAP 세 가지 스토리지 포트폴리오가 있습니다. 우리는 Apache Spark를 기반으로 Hadoop 솔루션을 위한 ONTAP 스토리지 시스템을 갖춘 AFF 와 E-시리즈를 검증했습니다.

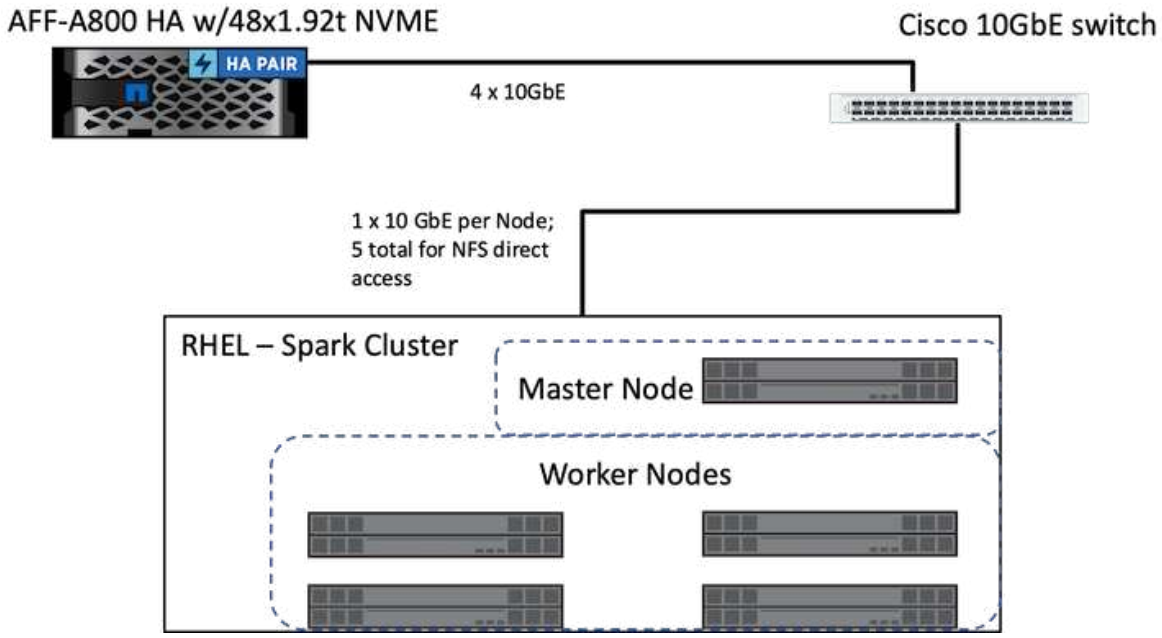
NetApp 이 제공하는 데이터 패브릭은 아래 그림에서 볼 수 있듯이 데이터 액세스, 제어, 보호 및 보안을 위한 데이터 관리 서비스와 애플리케이션(빌딩 블록)을 통합합니다.



위 그림의 구성 요소는 다음과 같습니다.

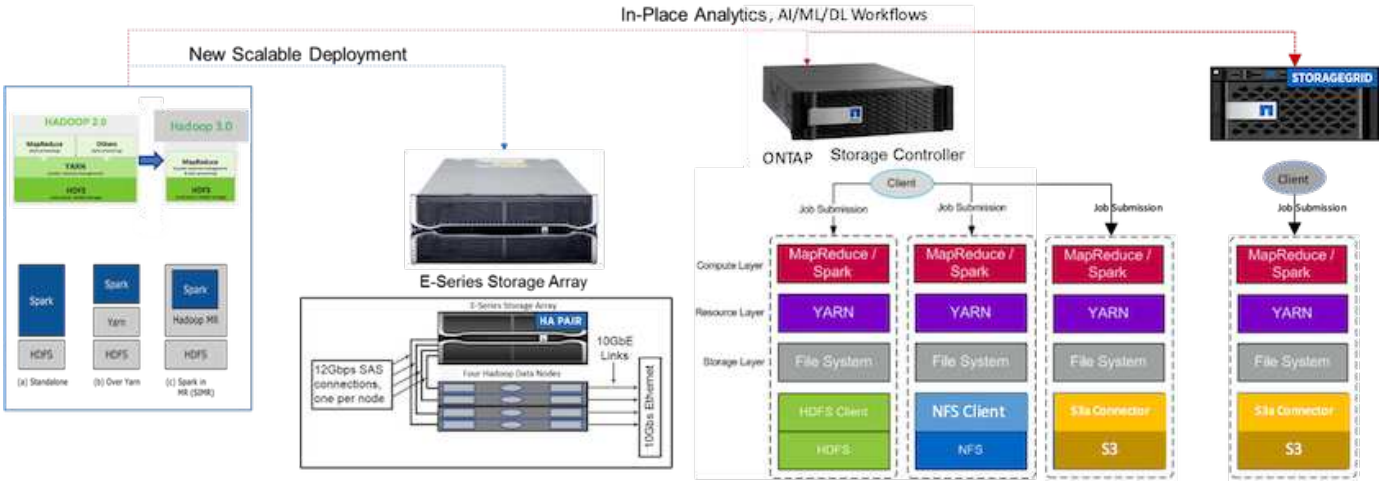
- * NetApp NFS 직접 액세스.* 추가 소프트웨어나 드라이버가 필요하지 않고 최신 Hadoop 및 Spark 클러스터에서 NetApp NFS 볼륨에 직접 액세스할 수 있습니다.
- * NetApp Cloud Volumes ONTAP 및 Google Cloud NetApp Volumes.* Amazon Web Services(AWS)에서 실행되는 ONTAP 또는 Microsoft Azure 클라우드 서비스의 Azure NetApp Files (ANF) 기반의 소프트웨어 정의 연결 스토리지입니다.
- * NetApp SnapMirror 기술.* 온프레미스와 ONTAP Cloud 또는 NPS 인스턴스 간의 데이터 보호 기능을 제공합니다.
- 클라우드 서비스 제공자**.** 이러한 공급업체로는 AWS, Microsoft Azure, Google Cloud, IBM Cloud가 있습니다.
- **PaaS.** AWS의 Amazon Elastic MapReduce(EMR) 및 Databricks, Microsoft Azure HDInsight 및 Azure Databricks와 같은 클라우드 기반 분석 서비스입니다.

다음 그림은 NetApp 스토리지를 사용한 Spark 솔루션을 보여줍니다.

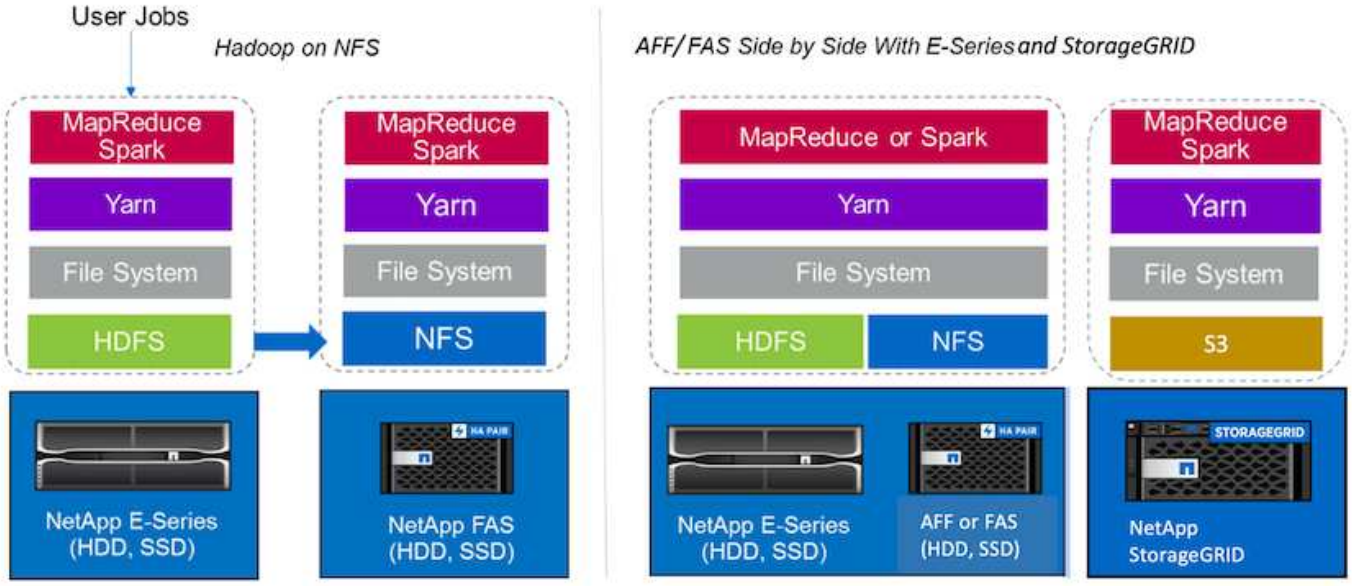


ONTAP Spark 솔루션은 기존 프로덕션 데이터에 대한 액세스를 활용하여 현장 분석 및 AI, ML, DL 워크플로를 위한 NetApp NFS 직접 액세스 프로토콜을 사용합니다. Hadoop 노드에서 사용할 수 있는 프로덕션 데이터는 현장 분석 및 AI, ML, DL 작업을 수행하는 데 내보내집니다. NetApp NFS 직접 액세스를 사용하거나 사용하지 않고도 Hadoop 노드에서 처리할 데이터에 액세스할 수 있습니다. Spark에서 독립형 또는 `yarn` 클러스터 관리자를 사용하여 NFS 볼륨을 구성할 수 있습니다. `file://<target_volume>` . 우리는 서로 다른 데이터 세트를 사용하여 세 가지 사용 사례를 검증했습니다. 이러한 검증에 대한 세부 사항은 "테스트 결과" 섹션에 나와 있습니다. (xref)

다음 그림은 NetApp Apache Spark/Hadoop 스토리지 포지셔닝을 보여줍니다.



우리는 E-Series Spark 솔루션, AFF/ FAS ONTAP Spark 솔루션, StorageGRID Spark 솔루션의 고유한 기능을 파악하고 자세한 검증 및 테스트를 수행했습니다. 당사의 관찰에 따르면 NetApp 그린필드 설치 및 새로운 확장형 배포의 경우 E-Series 솔루션을, 기존 NFS 데이터를 사용하는 현장 분석, AI, ML, DL 워크로드의 경우 AFF/ FAS 솔루션을, 객체 스토리지가 필요한 경우 AI, ML, DL 및 최신 데이터 분석의 경우 StorageGRID 권장합니다.



데이터 레이크는 분석, AI, ML, DL 작업에 사용할 수 있는 기본 형태의 대규모 데이터 세트를 저장하는 저장소입니다. 우리는 E-Series, AFF/ FAS, StorageGRID SG6060 Spark 솔루션을 위한 데이터 레이크 저장소를 구축했습니다. E-시리즈 시스템은 Hadoop Spark 클러스터에 HDFS 액세스를 제공하는 반면, 기존 프로덕션 데이터는 Hadoop 클러스터에 대한 NFS 직접 액세스 프로토콜을 통해 액세스됩니다. 개체 스토리지에 있는 데이터 세트의 경우 NetApp StorageGRID S3 및 S3a 보안 액세스를 제공합니다.

사용 사례 요약

이 페이지에서는 이 솔루션을 사용할 수 있는 다양한 영역을 설명합니다.

스트리밍 데이터

Apache Spark는 스트리밍 데이터를 처리할 수 있으며, 이 데이터는 스트리밍 추출, 변환 및 로드(ETL) 프로세스, 데이터 강화, 이벤트 감지 트리거, 복잡한 세션 분석에 사용됩니다.

- 스트리밍 **ETL.** 데이터는 데이터 저장소에 푸시되기 전에 지속적으로 정리되고 집계됩니다. Netflix는 Kafka와 Spark 스트리밍을 사용하여 다양한 데이터 소스에서 하루에 수십억 개의 이벤트를 처리할 수 있는 실시간 온라인 영화 추천 및 데이터 모니터링 솔루션을 구축합니다. 하지만 일괄 처리를 위한 기존 ETL은 다르게 처리됩니다. 이 데이터는 먼저 읽혀지고, 그런 다음 데이터베이스에 기록되기 전에 데이터베이스 형식으로 변환됩니다.
- 데이터 강화**.** Spark 스트리밍은 실시간 데이터에 정적 데이터를 추가하여 보다 실시간적인 데이터 분석을 가능하게 합니다. 예를 들어, 온라인 광고주는 고객 행동에 대한 정보를 바탕으로 개인화되고 타겟이 지정된 광고를 게재할 수 있습니다.
- 트리거 이벤트 감지**.** Spark 스트리밍을 사용하면 잠재적으로 심각한 문제를 나타낼 수 있는 비정상적인 동작을 감지하고 신속하게 대응할 수 있습니다. 예를 들어, 금융 기관은 트리거를 사용하여 사기 거래를 감지하고 중단하고, 병원에서는 트리거를 사용하여 환자의 생체 신호에서 발견된 위험한 건강 변화를 감지합니다.
- 복잡한 세션 분석**.** Spark 스트리밍은 웹사이트나 애플리케이션에 로그인한 후의 사용자 활동과 같은 이벤트를 수집한 다음 이를 그룹화하여 분석합니다. 예를 들어, Netflix는 이 기능을 사용하여 실시간으로 영화를 추천해 줍니다.

스트리밍 데이터 구성, Confluent Kafka 검증 및 성능 테스트에 대한 자세한 내용은 다음을 참조하세요."TR-4912: NetApp 사용한 Confluent Kafka 계층형 스토리지에 대한 모범 사례 가이드라인" .

머신러닝

Spark 통합 프레임워크는 머신 러닝 라이브러리(MLlib)를 사용하여 데이터 세트에 대한 반복 쿼리를 실행하는 데 도움이 됩니다. MLlib은 예측 인텔리전스, 마케팅 목적을 위한 고객 세분화, 감정 분석 등 일반적인 빅데이터 기능에 대한 클러스터링, 분류, 차원 축소와 같은 분야에서 사용됩니다. MLlib은 네트워크 보안에서 악의적인 활동을 나타내는 데이터 패킷의 실시간 검사를 수행하는 데 사용됩니다. 보안 서비스 제공업체가 새로운 위협에 대해 알아내고 해커보다 앞서 나가는 동시에 실시간으로 고객을 보호하는 데 도움이 됩니다.

딥러닝

TensorFlow는 업계 전반에서 사용되는 인기 있는 딥 러닝 프레임워크입니다. TensorFlow는 CPU 또는 GPU 클러스터에서의 분산 학습을 지원합니다. 이 분산형 학습을 통해 사용자는 많은 심층 레이어가 있는 대량의 데이터에서 학습을 실행할 수 있습니다.

얼마 전까지만 해도 Apache Spark에서 TensorFlow를 사용하려면 PySpark에서 TensorFlow에 대한 모든 필수 ETL을 수행한 다음 중간 저장소에 데이터를 써야 했습니다. 그런 다음 해당 데이터는 실제 학습 과정을 위해 TensorFlow 클러스터에 로드됩니다. 이 워크플로를 사용하려면 사용자가 두 개의 서로 다른 클러스터를 유지해야 합니다. 하나는 ETL용이고 다른 하나는 TensorFlow의 분산 학습용입니다. 여러 개의 클러스터를 실행하고 유지 관리하는 일은 일반적으로 지루하고 시간이 많이 걸렸습니다.

이전 Spark 버전의 DataFrames와 RDD는 무작위 접근이 제한되어 있어 딥러닝에 적합하지 않았습니다. Spark 3.0과 Project Hydrogen에서는 딥러닝 프레임워크에 대한 기본 지원이 추가되었습니다. 이 접근 방식을 사용하면 Spark 클러스터에서 MapReduce 기반이 아닌 스케줄링이 가능합니다.

대화형 분석

Apache Spark는 SQL, R, Python 등 Spark 이외의 개발 언어로 샘플링하지 않고도 탐색적 쿼리를 수행할 만큼 빠릅니다. Spark는 시각화 도구를 사용하여 복잡한 데이터를 처리하고 대화형으로 시각화합니다. 구조화된 스트리밍을 탑재한 Spark는 웹 분석에서 라이브 데이터에 대한 대화형 쿼리를 수행하여 웹 방문자의 현재 세션에 대한 대화형 쿼리를 실행할 수 있습니다.

추천 시스템

수년에 걸쳐 추천 시스템은 우리 삶에 엄청난 변화를 가져왔습니다. 기업과 소비자는 온라인 쇼핑, 온라인 엔터테인먼트 등 여러 산업의 극적인 변화에 대응해 왔습니다. 실제로 이러한 시스템은 생산 현장에서 AI를 활용한 가장 확실한 성공 사례 중 하나입니다. 많은 실제 사용 사례에서 추천 시스템은 NLP 백엔드와 인터페이스된 대화형 AI 또는 챗봇과 결합되어 관련 정보를 얻고 유용한 추론을 생성합니다.

오늘날 많은 소매업체가 온라인에서 구매하고 매장에서 픽업하는 방식, 연석 픽업, 셀프 체크아웃, 스캔 앤 고 등 새로운 비즈니스 모델을 도입하고 있습니다. 이러한 모델은 소비자의 쇼핑을 더 안전하고 편리하게 만들어줌으로써 COVID-19 팬데믹 기간 동안 두각을 나타냈습니다. 소비자 행동에 영향을 받는 이러한 성장하는 디지털 트렌드에 AI는 매우 중요합니다. 소비자의 증가하는 요구를 충족하고, 고객 경험을 강화하고, 운영 효율성을 개선하고, 수익을 늘리기 위해 NetApp 기업 고객과 사업체가 머신 러닝과 딥 러닝 알고리즘을 사용하여 더 빠르고 정확한 추천 시스템을 설계할 수 있도록 지원합니다.

추천을 제공하는 데 사용되는 여러 가지 인기 있는 기술이 있는데, 여기에는 협업 필터링, 콘텐츠 기반 시스템, 딥러닝 추천 모델(DLRM), 하이브리드 기술이 포함됩니다. 이전에 고객들은 PySpark를 활용해 추천 시스템을 만드는 협업 필터링을 구현했습니다. Spark MLlib은 DLRM이 등장하기 전에 기업에서 매우 인기 있는 알고리즘인 협업 필터링을 위한 교대 최소 제곱법(ALS)을 구현합니다.

자연어 처리

자연어 처리(NLP)를 통해 가능해진 대화형 AI는 컴퓨터가 인간과 소통하는 데 도움이 되는 AI의 한 분야입니다. NLP는 모든 산업 분야에서 널리 사용되고 있으며, 스마트 어시스턴트와 챗봇부터 Google 검색과 예측 텍스트에 이르기까지 다양한 사용 사례가 있습니다. 에 따르면 "가트너" 예측에 따르면 2022년까지 70%의 사람들이 매일 대화형 AI 플랫폼과 상호작용하게 될 것입니다. 인간과 기계가 고품질 대화를 하려면 응답이 빠르고, 지능적이며, 자연스럽게 들려야 합니다.

고객은 NLP 및 자동 음성 인식(ASR) 모델을 처리하고 훈련하기 위해 대량의 데이터가 필요합니다. 또한 엣지, 코어, 클라우드 전반에서 데이터를 이동해야 하며, 인간과 자연스러운 소통을 구축하기 위해 밀리초 단위로 추론을 수행할 수 있는 능력이 필요합니다. NetApp AI와 Apache Spark는 컴퓨팅, 스토리지, 데이터 처리, 모델 학습, 미세 조정 및 배포에 이상적인 조합입니다.

감정 분석은 NLP 내의 연구 분야로, 텍스트에서 긍정적, 부정적 또는 중립적 감정을 추출합니다. 감정 분석은 통화자와의 대화에서 지원 센터 직원의 성과를 파악하는 것부터 적절한 자동화된 챗봇 응답을 제공하는 것까지 다양한 사용 사례에 적용됩니다. 또한 분기별 실적 발표에서 회사 대표와 청중 간의 상호 작용을 기반으로 회사 주가를 예측하는 데 사용되었습니다. 더욱이, 감정 분석은 브랜드가 제공하는 제품, 서비스 또는 지원에 대한 고객의 견해를 파악하는 데 사용될 수 있습니다.

우리는 사용했다 "스파크 NLP" 도서관에서 "존 스노우 랩스" BERT(Transformers) 모델을 포함하여 사전 학습된 파이프라인 및 양방향 인코더 표현을 로드하려면 "금융 뉴스 감정" 그리고 "핀버트" 대규모로 토큰화, 명명된 엔터티 인식, 모델 학습, 피팅 및 감정 분석을 수행합니다. Spark NLP는 BERT, ALBERT, ELECTRA, XLNet, DistilBERT, RoBERTa, DeBERTa, XLM- RoBERTa, Longformer, ELMO, Universal Sentence Encoder, Google T5, MarianMT, GPT2와 같은 최첨단 변환기를 제공하는 유일한 오픈 소스 NLP 라이브러리입니다. 이 라이브러리는 Python과 R에서 작동할 뿐만 아니라 Apache Spark를 기본적으로 확장하여 JVM 생태계(Java, Scala, Kotlin)에서도 대규모로 작동합니다.

주요 **AI, ML** 및 **DL** 사용 사례 및 아키텍처

주요 AI, ML, DL 사용 사례와 방법론은 다음 섹션으로 나눌 수 있습니다.

Spark NLP 파이프라인 및 **TensorFlow** 분산 추론

다음 목록은 다양한 개발 수준에서 데이터 과학 커뮤니티에서 채택된 가장 인기 있는 오픈소스 NLP 라이브러리를 포함합니다.

- "자연어 툴킷(NLTK)" . 모든 NLP 기술을 위한 완벽한 툴킷입니다. 2000년대 초반부터 유지되어 왔습니다.
- "텍스트블롭" . NLTK와 Pattern을 기반으로 구축된 사용하기 쉬운 NLP 도구 Python API입니다.
- "스탠포드 코어 NLP" . Stanford NLP Group에서 개발한 Java로 작성된 NLP 서비스와 패키지입니다.
- "젠심" . 인간을 위한 주제 모델링은 체코 디지털 수학 도서관 프로젝트를 위한 Python 스크립트 모음으로 시작되었습니다.
- "스파시" . GPU 가속을 탑재한 Python 및 Cython을 사용한 엔드투엔드 산업용 NLP 워크플로우로 변압기를 구현합니다.
- "패스트텍스트" . Facebook의 AI 연구(FAIR) 랩에서 만든 단어 임베딩 학습과 문장 분류를 위한 무료, 가벼운 오픈 소스 NLP 라이브러리입니다.

Spark NLP는 모든 NLP 작업과 요구 사항을 위한 단일 통합 솔루션으로, 실제 생산 사용 사례에서 확장 가능하고 성능이 뛰어나며 정확도가 높은 NLP 기반 소프트웨어를 구현합니다. 이는 전이 학습을 활용하고 연구와 산업 전반에 걸쳐 최신 첨단 알고리즘과 모델을 구현합니다. Spark에서 위 라이브러리에 대한 완전한 지원이 부족하기 때문에 Spark NLP는 다음 라이브러리를 기반으로 구축되었습니다. "스파크 ML" 미션 크리티컬 프로덕션 워크플로를 위한

엔터프라이즈급 NLP 라이브러리로 Spark의 범용 인메모리 분산 데이터 처리 엔진을 활용합니다. 주석자는 규칙 기반 알고리즘, 머신 러닝, TensorFlow를 활용해 딥 러닝 구현을 지원합니다. 여기에는 토큰화, 레마토제닉, 어간 추출, 품사 태깅, 명명된 엔터티 인식, 철자 검사, 감정 분석 등을 포함하되 이에 국한되지 않는 일반적인 NLP 작업이 포함됩니다.

BERT(Bidirectional Encoder Representations from Transformers)는 NLP를 위한 변환기 기반 머신 러닝 기술입니다. 이를 통해 사전 학습과 미세 조정이라는 개념이 대중화되었습니다. BERT의 변환기 아키텍처는 기계 번역에서 유래되었는데, 기계 번역은 RNN(Recurrent Neural Network) 기반 언어 모델보다 장기 종속성을 더 잘 모델링합니다. 또한 모든 토큰 중 무작위로 15%를 마스크 처리하고 모델이 이를 예측하여 진정한 양방향성을 구현하는 MLM(Masked Language Modelling) 작업을 도입했습니다.

금융 감정 분석은 해당 분야의 전문 용어와 라벨이 지정된 데이터의 부족으로 인해 어렵습니다. 사전 학습된 BERT를 기반으로 하는 언어 모델인 FinBERT는 도메인에 적응되었습니다. "로이터 TRC2" , 금융 자산 및 레이블이 지정된 데이터로 미세 조정됨("금융 문구 은행") 금융 감정 분류를 위해. 연구자들은 금융 용어가 포함된 뉴스 기사에서 4,500개의 문장을 추출했습니다. 그런 다음 금융 분야 경력이 있는 16명의 전문가와 석사과정 학생들이 문장을 긍정적, 중립적, 부정적으로 분류했습니다. 우리는 FinBERT와 두 개의 다른 사전 훈련된 파이프라인을 사용하여 2016년부터 2020년까지 상위 10개 NASDAQ 회사 수익 전화 회의록에 대한 감정을 분석하기 위한 종단 간 Spark 워크플로를 구축했습니다. "문서 DL 설명") Spark NLP에서.

Spark NLP의 기반이 되는 딥 러닝 엔진은 TensorFlow입니다. TensorFlow는 머신 러닝을 위한 종단 간 오픈 소스 플랫폼으로, 쉬운 모델 구축, 어디서나 강력한 ML 생산, 연구를 위한 강력한 실험을 가능하게 합니다. 따라서 Spark에서 파이프라인을 실행할 때 `yarn cluster` 모드에서는 기본적으로 하나의 마스터 노드와 여러 개의 워커 노드, 그리고 클러스터에 마운트된 네트워크 연결 스토리지에서 데이터와 모델을 병렬화하여 분산형 TensorFlow를 실행했습니다.

Horovod 분산 훈련

MapReduce 관련 성능에 대한 핵심 Hadoop 검증은 TeraGen, TeraSort, TeraValidate 및 DFSIO(읽기 및 쓰기)를 사용하여 수행됩니다. TeraGen 및 TeraSort 검증 결과는 다음과 같습니다. "Hadoop을 위한 NetApp E-Series 솔루션" AFF 의 "스토리지 계층화" 섹션에 있습니다.

고객 요청에 따라, 우리는 Spark를 활용한 분산 학습을 다양한 사용 사례 중 가장 중요한 것 중 하나로 생각합니다. 이 문서에서는 다음을 사용했습니다. "스파크의 호보로드" NetApp All Flash FAS (AFF) 스토리지 컨트롤러, Azure NetApp Files 및 StorageGRID 사용하여 NetApp 온프레미스, 클라우드 네이티브 및 하이브리드 클라우드 솔루션으로 Spark 성능을 검증합니다.

Spark 패키지의 Horovod는 Horovod를 둘러싼 편리한 래퍼를 제공하여 Spark 클러스터에서 분산형 학습 워크로드를 간편하게 실행하고, 데이터 처리, 모델 학습, 모델 평가가 모두 학습 및 추론 데이터가 있는 Spark에서 수행되는 긴밀한 모델 설계 루프를 구현합니다.

Spark에서 Horovod를 실행하기 위한 두 가지 API가 있습니다. 상위 수준 Estimator API와 하위 수준 Run API입니다. 둘 다 Spark 실행자에서 Horovod를 실행하기 위해 동일한 기본 메커니즘을 사용하지만 Estimator API는 데이터 처리, 모델 학습 루프, 모델 검사점, 메트릭 수집 및 분산 학습을 추상화합니다. 우리는 종단 간 데이터 준비 및 분산 학습 워크플로를 위해 Horovod Spark Estimators, TensorFlow 및 Keras를 사용했습니다. "Kaggle Rossmann 매장 판매" 경쟁.

대본 `keras_spark_horovod_rossmann_estimator.py` 섹션에서 찾을 수 있습니다"주요 사용 사례별로 Python 스크립트가 제공됩니다." 이 글은 세 부분으로 구성되어 있습니다.

- 첫 번째 부분에서는 Kaggle에서 제공하고 커뮤니티에서 수집한 초기 CSV 파일 세트에 대해 다양한 데이터 전처리 단계를 수행합니다. 입력 데이터는 다음과 같은 훈련 세트로 분리됩니다. `Validation` 하위 집합과 테스트 데이터 세트.
- 두 번째 부분에서는 로그 시그모이드 활성화 함수와 Adam 옵티마이저를 갖춘 Keras 딥 신경망(DNN) 모델을 정의하고, Spark에서 Horovod를 사용하여 모델의 분산 학습을 수행합니다.

- 세 번째 부분에서는 검증 세트의 전체 평균 절대 오차를 최소화하는 최상의 모델을 사용하여 테스트 데이터 세트에 대한 예측을 수행합니다. 그런 다음 출력 CSV 파일을 생성합니다.

섹션을 참조하세요"머신 러닝" 다양한 런타임 비교 결과를 위해.

CTR 예측을 위한 Keras를 사용한 멀티 워커 딥 러닝

최근 ML 플랫폼과 애플리케이션이 발전함에 따라 대규모 학습에 많은 관심이 쏠리고 있습니다. 클릭률(CTR)은 온라인 광고 노출 100회당 평균 클릭 수(백분율로 표시)로 정의됩니다. 이는 디지털 마케팅, 소매, 전자상거래, 서비스 제공업체를 포함한 다양한 산업 분야와 사용 사례에서 핵심 지표로 널리 채택되고 있습니다. CTR 및 분산형 교육 성능 결과의 적용에 대한 자세한 내용은 다음을 참조하세요."CTR 예측 성능을 위한 딥러닝 모델" 부분.

이 기술 보고서에서는 다음 변형을 사용했습니다. "Criteo 테라바이트 클릭 로그 데이터 세트" (TR-4904 참조) Keras를 사용하여 다중 작업자 분산 딥 러닝을 통해 딥 및 교차 네트워크(DCN) 모델을 포함하는 Spark 워크플로를 구축하고, 로그 손실 오차 함수 측면에서 기준 Spark ML 로지스틱 회귀 모델과 성능을 비교합니다. DCN은 제한된 차수의 효과적인 기능 상호작용을 효율적으로 포착하고, 높은 비선형 상호작용을 학습하며, 수동 기능 엔지니어링이나 철저한 검색이 필요 없고, 계산 비용이 낮습니다.

웹 규모 추천 시스템에 사용되는 데이터는 대부분 이산적이고 범주형이어서 기능 탐색이 어려운 크고 희소한 기능 공간이 발생합니다. 이로 인해 대부분의 대규모 시스템은 로지스틱 회귀와 같은 선형 모델로 제한되었습니다. 그러나 자주 예측 가능한 특징을 식별하고 동시에 보이지 않거나 드문 교차 특징을 탐색하는 것이 좋은 예측을 하는 데 중요합니다. 선형 모델은 간단하고, 해석 가능하며, 확장하기 쉽지만, 표현력이 제한적입니다.

반면, 교차 특징은 모델의 표현력을 향상시키는 데 중요한 것으로 나타났습니다. 안타깝게도 이러한 기능을 식별하려면 수동 기능 엔지니어링이나 철저한 검색이 필요한 경우가 많습니다. 보이지 않는 특징의 상호작용을 일반화하는 것은 종종 어렵습니다. DCN과 같은 교차 신경망을 사용하면 자동으로 기능 교차를 명시적으로 적용하여 작업별 기능 엔지니어링을 피할 수 있습니다. 교차 네트워크는 여러 계층으로 구성되어 있으며, 가장 높은 수준의 상호작용은 계층 깊이에 의해 결정됩니다. 각 계층은 기존 계층의 상호작용을 기반으로 고차원 상호작용을 생성하고 이전 계층의 상호작용은 유지합니다.

심층 신경망(DNN)은 여러 기능 간의 매우 복잡한 상호작용을 포착할 수 있는 잠재력을 가지고 있습니다. 그러나 DCN과 비교하면 거의 10배 더 많은 매개변수가 필요하고, 교차 특징을 명확하게 형성할 수 없으며, 일부 유형의 특징 상호작용을 효율적으로 학습하지 못할 수 있습니다. 크로스 네트워크는 메모리 효율성이 높고 구현하기 쉽습니다. 교차 분석과 DNN 구성 요소를 함께 공동으로 훈련하면 예측 기능 상호작용을 효율적으로 포착하고 Criteo CTR 데이터 세트에서 최첨단 성능을 제공합니다.

DCN 모델은 임베딩 및 스태킹 계층으로 시작하여, 이어서 교차 네트워크와 딥 네트워크가 병렬로 이어집니다. 이어서 두 네트워크의 출력을 결합하는 최종 결합 계층이 이어집니다. 입력 데이터는 희소 특성과 밀집 특성을 모두 갖춘 벡터일 수 있습니다. Spark에서는 라이브러리에 다음 유형이 포함되어 있습니다. `SparseVector` . 따라서 사용자는 두 가지를 구별하고 각각의 함수와 메서드를 호출할 때 주의하는 것이 중요합니다. CTR 예측과 같은 웹 규모 추천 시스템에서 입력은 대부분 범주형 기능입니다. `'country=usa'` . 이러한 기능은 종종 원핫 벡터로 인코딩됩니다. 예를 들어, `'[0,1,0, …]'` . One-hot-encoding(OHE) `SparseVector` 끊임없이 변화하고 증가하는 어휘를 포함하는 실제 데이터 세트를 다룰 때 유용합니다. 우리는 예를 수정했습니다 "딥CTR" 대규모 어휘를 처리하여 DCN의 임베딩 및 스태킹 계층에서 임베딩 벡터를 생성합니다.

그만큼 "Criteo 디스플레이 광고 데이터 세트" 광고 클릭률을 예측합니다. 13개의 정수 특성과 26개의 범주형 특성이 있으며, 각 범주는 높은 카디널리티를 갖습니다. 이 데이터 세트의 경우 입력 크기가 크기 때문에 logloss가 0.001 향상되는 것은 실질적으로 의미가 있습니다. 대규모 사용자 기반의 예측 정확도가 약간만 향상되어도 회사 수익이 크게 증가할 가능성이 있습니다. 이 데이터 세트에는 7일간의 사용자 로그 11GB가 포함되어 있으며, 이는 약 4,100만 개의 레코드에 해당합니다. 우리는 Spark를 사용했습니다 `dataFrame.randomSplit()function` 데이터를 무작위로 분할하여 학습(80%), 교차 검증(10%), 나머지 10%를 테스트에 사용합니다.

DCN은 Keras를 사용하여 TensorFlow에서 구현되었습니다. DCN을 사용하여 모델 학습 프로세스를 구현하는 데는

4가지 주요 구성 요소가 있습니다.

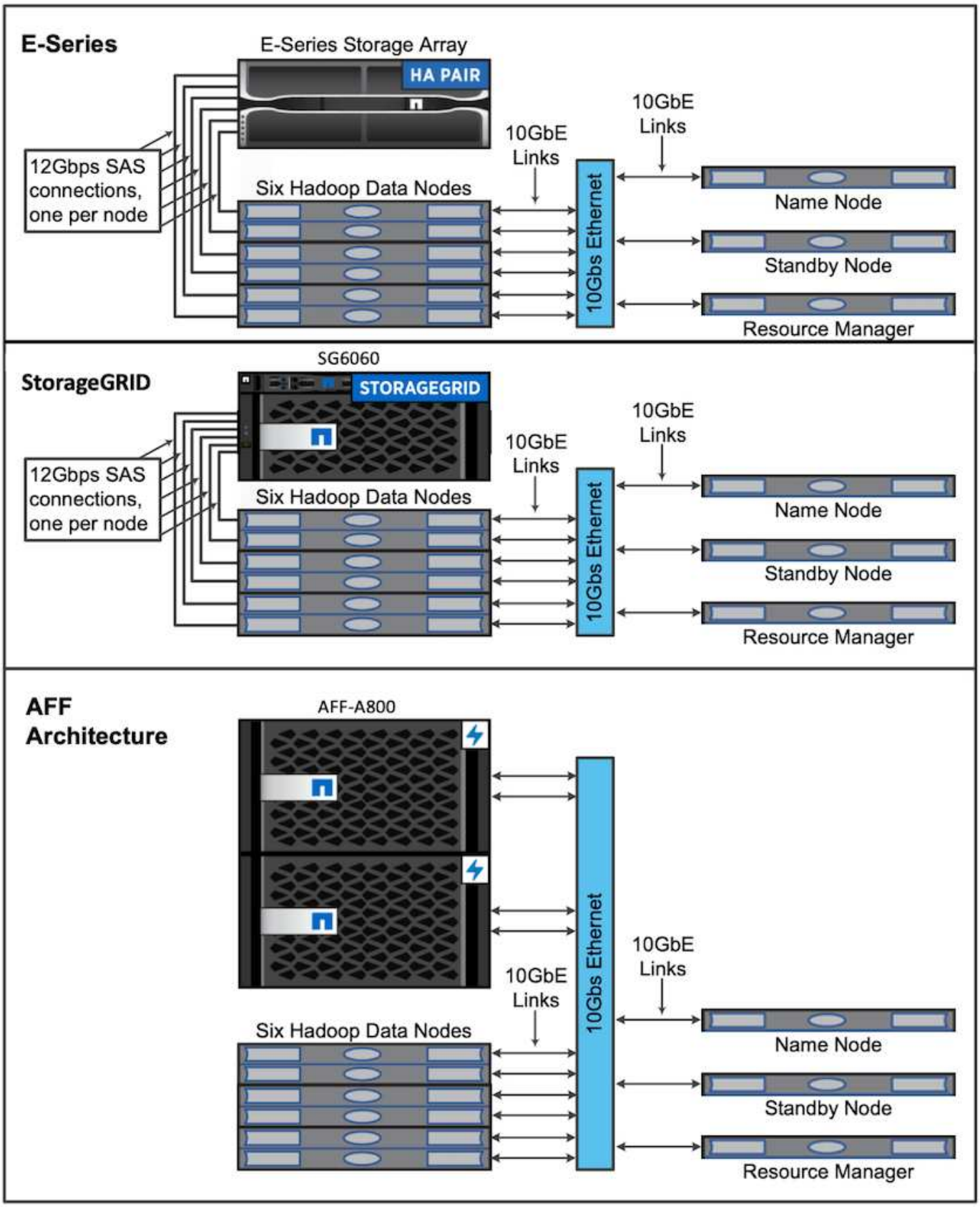
- 데이터 처리 및 임베딩. 실수 값의 특징은 로그 변환을 적용하여 정규화됩니다. 범주형 특성의 경우 차원 6×(범주 카디널리티)1/4의 밀집 벡터에 특성을 포함합니다. 모든 임베딩을 연결하면 차원이 1026인 벡터가 생성됩니다.
- 최적화. 우리는 Adam 최적화 도구를 사용하여 미니 배치 확률적 최적화를 적용했습니다. 배치 크기는 512로 설정되었습니다. 딥 네트워크에 배치 정규화를 적용하고 그래디언트 클립 노름을 100으로 설정했습니다.
- 정규화. L2 정규화나 드롭아웃이 효과적이지 않은 것으로 나타났기 때문에 조기 중단을 사용했습니다.
- 하이퍼매개변수 우리는 숨겨진 계층의 수, 숨겨진 계층의 크기, 초기 학습률, 교차 계층의 수에 대한 그리드 검색을 기반으로 결과를 보고합니다. 숨겨진 레이어의 수는 2~5개이고, 숨겨진 레이어의 크기는 32~1024개입니다. DCN의 경우 교차 레이어의 수는 1~6입니다. 초기 학습률은 0.0001에서 0.001까지 0.0001씩 증가하며 조정되었습니다. 모든 실험은 150,000번째 훈련 단계에서 조기에 중단되었으며, 그 단계를 넘어서면 과잉 맞춤이 발생하기 시작했습니다.

DCN 외에도 CTR 예측을 위한 다른 인기 있는 딥러닝 모델도 테스트했습니다. "딥FM" , "자동 Int" , 그리고 "DCN v2" .

검증에 사용되는 아키텍처

이러한 검증을 위해 우리는 AFF-A800 HA 쌍을 갖춘 4개의 워커 노드와 1개의 마스터 노드를 사용했습니다. 모든 클러스터 구성원은 10GbE 네트워크 스위치를 통해 연결되었습니다.

이 NetApp Spark 솔루션 검증을 위해 E5760, E5724, AFF-A800의 세 가지 스토리지 컨트롤러를 사용했습니다. E-시리즈 스토리지 컨트롤러는 12Gbps SAS 연결을 통해 5개의 데이터 노드에 연결되었습니다. AFF HA 쌍 스토리지 컨트롤러는 10GbE 연결을 통해 Hadoop 워커 노드에 내보낸 NFS 볼륨을 제공합니다. Hadoop 클러스터 멤버는 E-Series, AFF 및 StorageGRID Hadoop 솔루션에서 10GbE 연결을 통해 연결되었습니다.



테스트 결과

TeraGen 벤치마킹 도구의 TeraSort 및 TeraValidate 스크립트를 사용하여 E5760, E5724 및

AFF-A800 구성으로 Spark 성능 검증을 측정했습니다. 또한, Spark NLP 파이프라인과 TensorFlow 분산 학습, Horovod 분산 학습, DeepFM을 사용한 CTR 예측을 위한 Keras를 사용한 다중 워커 딥 러닝의 세 가지 주요 사용 사례가 테스트되었습니다.

E-Series와 StorageGRID 검증에는 Hadoop 복제 계수 2를 사용했습니다. AFF 검증을 위해 우리는 단 하나의 데이터 소스만 사용했습니다.

다음 표는 Spark 성능 검증을 위한 하드웨어 구성을 나열한 것입니다.

유형	**Hadoop** 워커 노드	구동 유형	노드당 드라이브	스토리지 컨트롤러
SG6060	4	SAS	12	단일 고가용성(HA) 쌍
E5760	4	SAS	60	단일 HA 쌍
E5724	4	SAS	24	단일 HA 쌍
AFF800	4	SSD	6	단일 HA 쌍

다음 표에는 소프트웨어 요구 사항이 나열되어 있습니다.

소프트웨어	버전
RHEL	7.9
OpenJDK 런타임 환경	1.8.0
OpenJDK 64비트 서버 VM	25.302
깃	2.24.1
GCC/G++	11.2.1
불꽃	3.2.1
파이스파크	3.1.2
스파크NLP	3.4.2
텐서플로우	2.9.0
케라스	2.9.0
호로보드	0.24.3

금융 심리 분석

우리는 출판했다"TR-4910: NetApp AI를 활용한 고객 커뮤니케이션의 감정 분석" , 종단 간 대화형 AI 파이프라인이 다음을 사용하여 구축되었습니다. "NetApp DataOps 툴킷" , AFF 스토리지 및 NVIDIA DGX 시스템. 파이프라인은 DataOps Toolkit을 활용하여 일괄 오디오 신호 처리, 자동 음성 인식(ASR), 전이 학습 및 감정 분석을 수행합니다. "NVIDIA 리바 SDK" , 그리고 "타오 프레임워크" . 감정 분석 사용 사례를 금융 서비스 산업으로 확장하여 SparkNLP 워크플로를 구축하고, 명명된 엔터티 인식과 같은 다양한 NLP 작업을 위해 세 개의 BERT 모델을 로드하고, NASDAQ 상위 10대 기업의 분기별 실적 전화 회의에 대한 문장 수준의 감정을 얻었습니다.

다음 스크립트 `sentiment_analysis_spark. py` 다음 표에 표시된 것처럼 FinBERT 모델을 사용하여 HDFS에서 전사본을 처리하고 긍정적, 중립적, 부정적 감정 수를 생성합니다.

```
-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
```

다음 표는 2016년부터 2020년까지 NASDAQ 상위 10개 기업의 수익 발표 및 문장 단위 감정 분석을 나열한 것입니다.

감정 수와 백분율	**10**개 회사 모두	아에이피엘	**AMD**	아마존	**CSCO**	구글	인티씨	**MSFT**	엔비디아
양성 카운트	7447	1567	743	290	682	826	824	904	417
중립 카운트	64067	6856	7596	5086	6650	5914	6099	5715	6189
음수 카운트	1787	253	213	84	189	97	282	202	89
분류되지 않은 카운트	196	0	0	76	0	0	0	1	0
(총 개수)	73497	8676	8552	5536	7521	6837	7205	6822	6695

백분율로 보면, CEO와 CFO가 말한 대부분의 문장은 사실에 기반을 두고 있어 중립적인 감정을 담고 있습니다. 수익 전화 회의에서 분석가들은 긍정적이거나 부정적인 감정을 전달할 수 있는 질문을 합니다. 부정적 또는 긍정적 감정이 거래 당일이나 다음 날의 주가에 어떤 영향을 미치는지 정량적으로 추가 조사해 볼 가치가 있습니다.

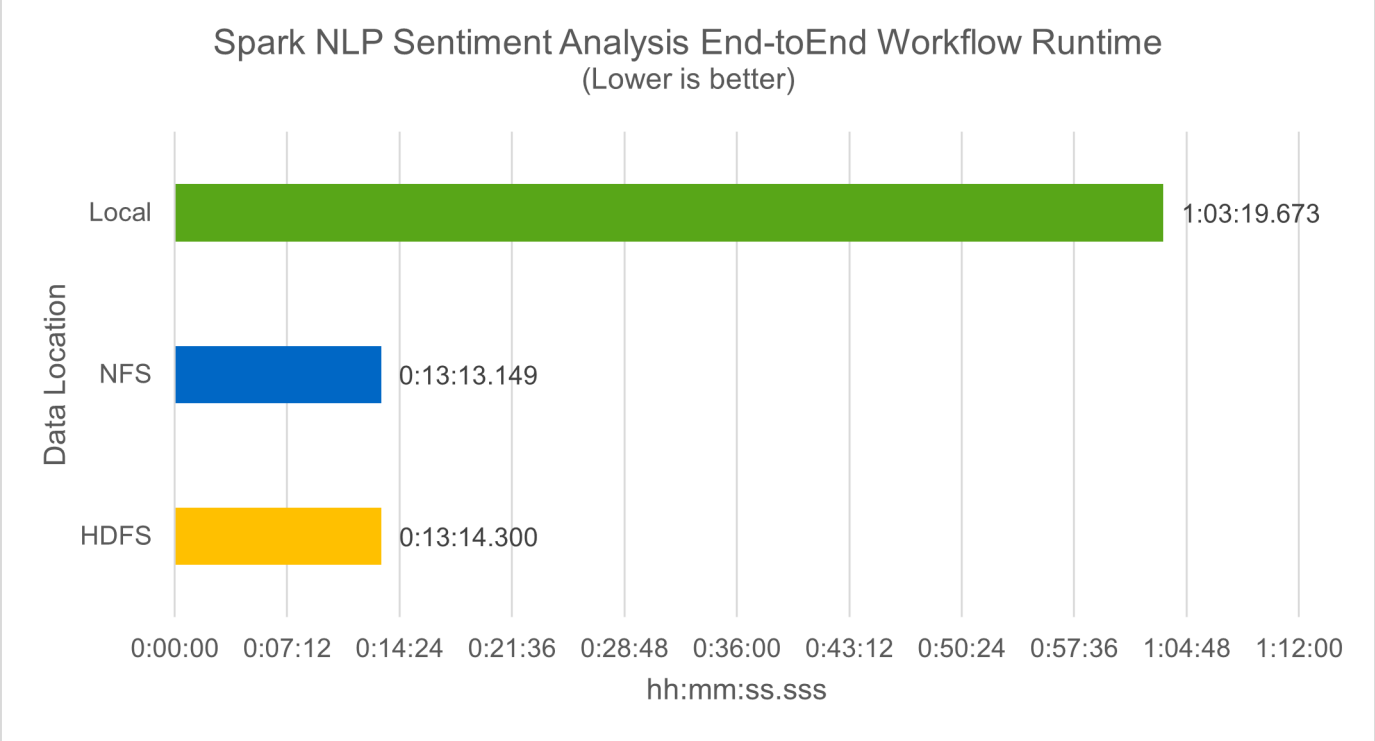
다음 표는 NASDAQ 상위 10개 기업의 문장 수준 감정 분석을 백분율로 나타낸 것입니다.

감정 비율	**10**개 회사 모두	아에이피엘	**AMD**	아마존	**CSCO**	구글	인티씨	**MSFT**	엔비디아
긍정적인	10.13%	18.06%	8.69%	5.24%	9.07%	12.08%	11.44%	13.25%	6.23%
중립적	87.17%	79.02%	88.82%	91.87%	88.42%	86.50%	84.65%	83.77%	92.44%
부정적인	2.43%	2.92%	2.49%	1.52%	2.51%	1.42%	3.91%	2.96%	1.33%
분류되지 않음	0.27%	0%	0%	1.37%	0%	0%	0%	0.01%	0%

워크플로 런타임 측면에서 우리는 4.78배의 상당한 개선을 보았습니다. `local` HDFS의 분산 환경으로 모드를 전환하고 NFS를 활용하여 0.14% 더 개선되었습니다.

```
-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
```

다음 그림에서 볼 수 있듯이, 데이터 및 모델 병렬 처리로 데이터 처리와 분산 TensorFlow 모델 추론 속도가 향상되었습니다. NFS에서 데이터를 배치하면 워크플로 병목 현상이 사전 학습된 모델을 다운로드하는 데서 발생하기 때문에 런타임이 약간 더 향상됩니다. 전사본 데이터 세트의 크기를 늘리면 NFS의 장점이 더욱 분명해집니다.



Horovod 성능을 활용한 분산 학습

다음 명령은 단일 명령을 사용하여 Spark 클러스터에서 런타임 정보와 로그 파일을 생성했습니다. `master` 각각 1개의 코어를 갖춘 160개의 실행자가 있는 노드입니다. 메모리 부족 오류를 방지하기 위해 실행자 메모리는 5GB로 제한되었습니다. 섹션을 참조하세요"각 주요 사용 사례에 대한 Python 스크립트" 데이터 처리, 모델 학습 및 모델 정확도

계산에 대한 자세한 내용은 `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
```

10개의 학습 에포크를 적용한 결과 런타임은 다음과 같습니다.

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

입력 데이터를 처리하고, DNN 모델을 훈련하고, 정확도를 계산하고, TensorFlow 체크포인트와 예측 결과를 위한 CSV 파일을 생성하는 데 43분 이상 걸렸습니다. 우리는 훈련 에포크의 수를 10으로 제한했는데, 실제로는 만족스러운 모델 정확도를 보장하기 위해 종종 100으로 설정합니다. 일반적으로 학습 시간은 에포크 수에 따라 선형적으로 증가합니다.

다음으로 클러스터에서 사용 가능한 4개의 작업자 노드를 사용하여 동일한 스크립트를 실행했습니다. `yarn` HDFS에 데이터가 있는 모드:

```
(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
```

그 결과 런타임은 다음과 같이 개선되었습니다.

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

Spark에서 Horovod의 모델과 데이터 병렬 처리를 통해 런타임 속도가 5.29배 향상되었습니다. `yarn` ~ 대 `local` 10개의 훈련 에포크를 가진 모드. 이는 다음 그림에서 범례와 함께 표시됩니다. `HDFS` 그리고 `Local` . 사용 가능한 경우 GPU를 사용하여 기본 TensorFlow DNN 모델 학습을 더욱 가속화할 수 있습니다. 우리는 이 테스트를 수행하고 그 결과를 향후 기술 보고서에 발표할 계획입니다.

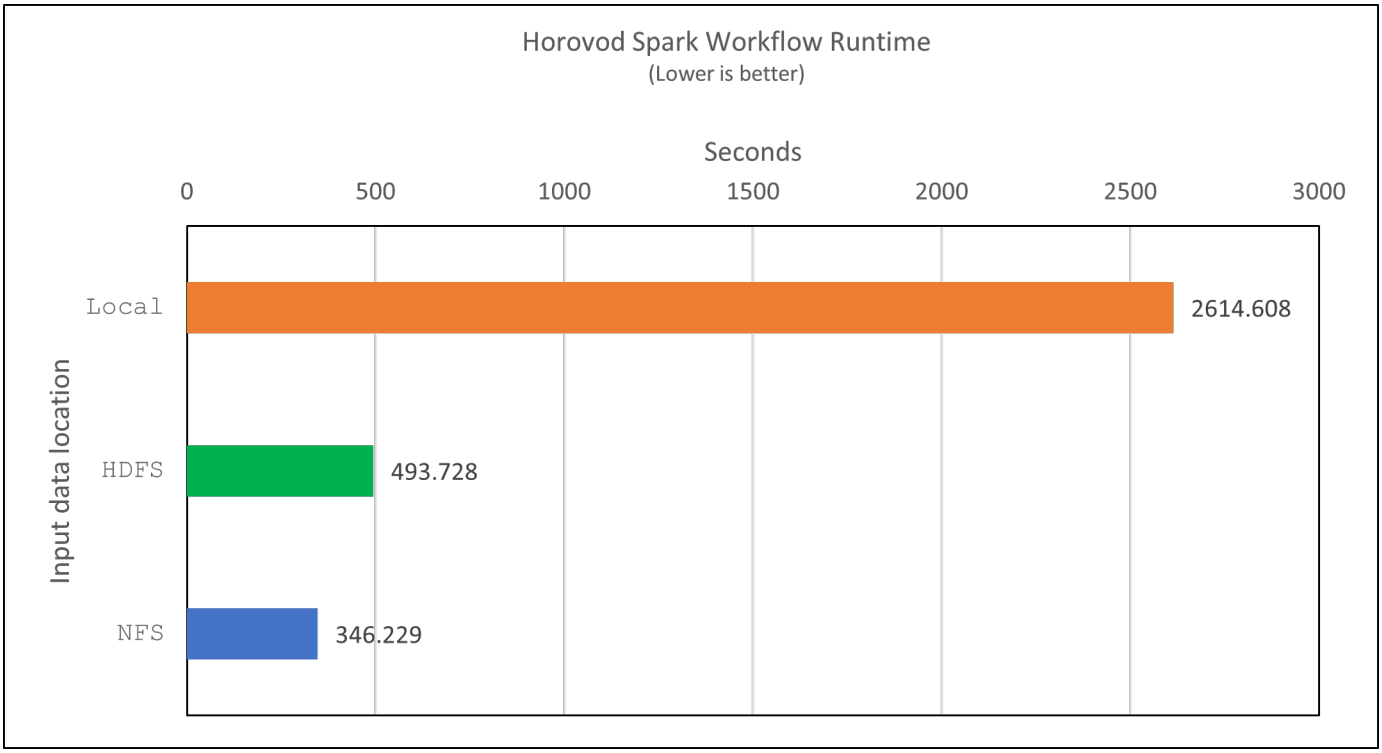
다음 테스트에서는 NFS와 HDFS에 있는 입력 데이터를 사용하여 런타임을 비교했습니다. AFF A800 의 NFS 볼륨은 다음에 마운트되었습니다. `/sparkdemo/horovod` Spark 클러스터의 5개 노드(마스터 1개, 워커 4개)에 걸쳐 있습니다. 우리는 이전 테스트와 유사한 명령을 실행했습니다. `--data- dir` 이제 NFS 마운트를 가리키는 매개변수:

```
(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
```

NFS를 사용한 결과 런타임은 다음과 같습니다.

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

다음 그림에서 볼 수 있듯이, 1.43배 더 속도가 향상되었습니다. 따라서 클러스터에 연결된 NetApp 올플래시 스토리지를 통해 고객은 Horovod Spark 워크플로우에서 빠른 데이터 전송 및 배포의 이점을 누릴 수 있으며, 단일 노드에서 실행하는 것보다 7.55배 빠른 속도를 달성할 수 있습니다.



CTR 예측 성능을 위한 딥러닝 모델

CTR을 극대화하도록 설계된 추천 시스템의 경우, 낮은 순위에서 높은 순위까지 수학적으로 계산할 수 있는 사용자 행동의 이면에 있는 정교한 기능 상호작용을 학습해야 합니다. 좋은 딥 러닝 모델에서는 낮은 순서와 높은 순서의 특징 상호작용이 둘 다 똑같이 중요해야 하며, 어느 한쪽에 치우치지 않아야 합니다. 인수분해 머신 기반 신경망인 DeepFM(Deep Factorization Machine)은 추천을 위한 인수분해 머신과 기능 학습을 위한 딥러닝을 새로운 신경망 아키텍처로 결합합니다.

기존의 인수분해 머신은 쌍별 특징 상호작용을 특징 간의 잠재 벡터의 내적으로 모델링하고 이론적으로는 고차 정보를 포착할 수 있지만, 실제로 머신 러닝 실무자는 높은 계산 및 저장 복잡성으로 인해 2차 특징 상호작용만 사용합니다. Google과 같은 딥 신경망 변형 "와이드 딥 모델" 반면에 선형 광역 모델과 심층 모델을 결합하여 하이브리드 네트워크 구조에서 정교한 기능 상호 작용을 학습합니다.

이 Wide & Deep 모델에는 두 가지 입력이 있습니다. 하나는 기본 Wide 모델을 위한 것이고 다른 하나는 Deep 모델을 위한 것입니다. Deep 모델의 후자에는 여전히 전문적인 기능 엔지니어링이 필요하므로 이 기술을 다른 도메인으로 일반화하기는 어렵습니다. Wide & Deep 모델과 달리 DeepFM은 와이드 부분과 딥 부분이 동일한 입력과 임베딩 벡터를 공유하기 때문에 기능 엔지니어링 없이 원시 기능으로 효율적으로 학습할 수 있습니다.

우리는 먼저 Criteo를 처리했습니다. `train.txt` (11GB) 파일을 CSV 파일로 변환 `ctr_train.csv` NFS 마운트에 저장됨 `/sparkdemo/tr-4570-data` 사용 중 `run_classification_criteo_spark.py` 섹션에서"주요 사용 사례별로 Python 스크립트가 제공됩니다." 이 스크립트 내에서 함수 `process_input_file` 탭을 제거하고 삽입하기 위해 여러 문자열 메서드를 수행합니다. `','` 구분 기호로 `'\n'` 줄바꿈으로. 원본만 처리하면 된다는 점에 유의하세요. `train.txt` 한 번, 코드 블록이 주석으로 표시됩니다.

다양한 DL 모델에 대한 다음 테스트를 위해 다음을 사용했습니다. `ctr_train.csv` 입력 파일로. 이후 테스트 실행에서 입력 CSV 파일은 스키마가 포함된 Spark DataFrame으로 읽혀졌습니다. `'label'` , 정수 밀집 기능 `['I1', 'I2', 'I3', …, 'I13']` , 그리고 희소한 특징 `['C1', 'C2', 'C3', …, 'C26']` . 다음 `spark-submit` 명령은 입력 CSV를 받고, 교차 검증을 위해 20% 분할로 DeepFM 모델을 학습하고, 10번의 학습 에포크 후에 가장 좋은 모델을 선택하여 테스트 세트에서 예측 정확도를 계산합니다.

```
(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
```

데이터 파일 이후에는 `ctr_train.csv` 11GB가 넘으면 충분한 용량을 설정해야 합니다. `spark.driver.maxResultSize` 오류를 피하기 위해 데이터 세트 크기보다 크게 설정합니다.

```
 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()
```

위의 `SparkSession.builder` 구성도 활성화했습니다 "아파치 애로우" Spark DataFrame을 Pandas DataFrame으로 변환하는 `df.toPandas()` 방법.

```
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.
```

무작위 분할 후, 훈련 데이터 세트에는 3,600만 개가 넘는 행이 있고 테스트 세트에는 900만 개의 샘플이 있습니다.

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

이 기술 보고서는 GPU를 사용하지 않고 CPU 테스트에 초점을 맞추고 있으므로 적절한 컴파일러 플래그로 TensorFlow를 빌드하는 것이 중요합니다. 이 단계에서는 GPU 가속 라이브러리를 호출하지 않고 TensorFlow의 AVX(Advanced Vector Extensions) 및 AVX2 명령어를 최대한 활용합니다. 이러한 기능은 벡터화된 덧셈, 피드포워드 내부의 행렬 곱셈 또는 역전파 DNN 훈련과 같은 선형 대수 계산을 위해 설계되었습니다. AVX2에서 제공하는 256비트 부동 소수점(FP) 레지스터를 사용하는 융합 곱셈 및 덧셈(FMA) 명령어는 정수 코드와 데이터 유형에 이상적이며 최대 2배의 속도 향상을 가져옵니다. FP 코드와 데이터 유형의 경우 AVX2는 AVX보다 8% 더 빠른 속도를 달성합니다.

```
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.
```

소스에서 TensorFlow를 빌드하려면 NetApp 다음을 사용하는 것이 좋습니다. "바젤" . 우리 환경에서는 셸 프롬프트에서 다음 명령을 실행하여 설치했습니다. `dnf` , `dnf-plugins` , 그리고 바젤.

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

RHEL에서 SCL(소프트웨어 컬렉션 라이브러리)을 통해 제공하는 C++17 기능을 빌드 프로세스 중에 사용하려면 GCC 5 이상 버전을 활성화해야 합니다. 다음 명령어를 설치합니다. `devtoolset` 그리고 RHEL 7.9 클러스터에 GCC 11.2.1이 있습니다.

```
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
```

마지막 두 명령은 다음을 활성화합니다. `devtoolset-11` , 사용하는 `/opt/rh/devtoolset-11/root/usr/bin/gcc` (GCC 11.2.1). 또한 다음을 확인하세요. `git` 버전이 1.8.3보다 높습니다(RHEL 7.9에 포함됨). 이것을 참조하세요 "기사" 업데이트용 `git` 2.24.1로.

귀하가 이미 최신 TensorFlow 마스터 저장소를 복제했다고 가정합니다. 그런 다음 생성하세요 `workspace` 디렉토리 `WORKSPACE` AVX, AVX2, FMA를 사용하여 소스에서 TensorFlow를 빌드하는 파일입니다. 실행하다 `configure` 파일을 만들고 올바른 Python 바이너리 위치를 지정합니다. "쿠다" 우리는 GPU를 사용하지 않았기 때문에 테스트를 위해 비활성화되었습니다. 에이 `.bazelrc` 파일은 귀하의 설정에 따라 생성됩니다. 또한, 우리는 파일을 편집하고 설정했습니다. `build --define=no_hdfs_support=false` HDFS 지원을 활성화합니다. 참조하다 `.bazelrc` 섹션에서"각 주요 사용 사례에 대한 Python 스크립트" 설정 및 플래그의 전체 목록을 확인하세요.

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

올바른 플래그로 TensorFlow를 빌드한 후 다음 스크립트를 실행하여 Criteo Display Ads 데이터 세트를 처리하고 DeepFM 모델을 학습시키고 예측 점수에서 수신자 조작 특성 곡선 아래의 면적(ROC AUC)을 계산합니다.

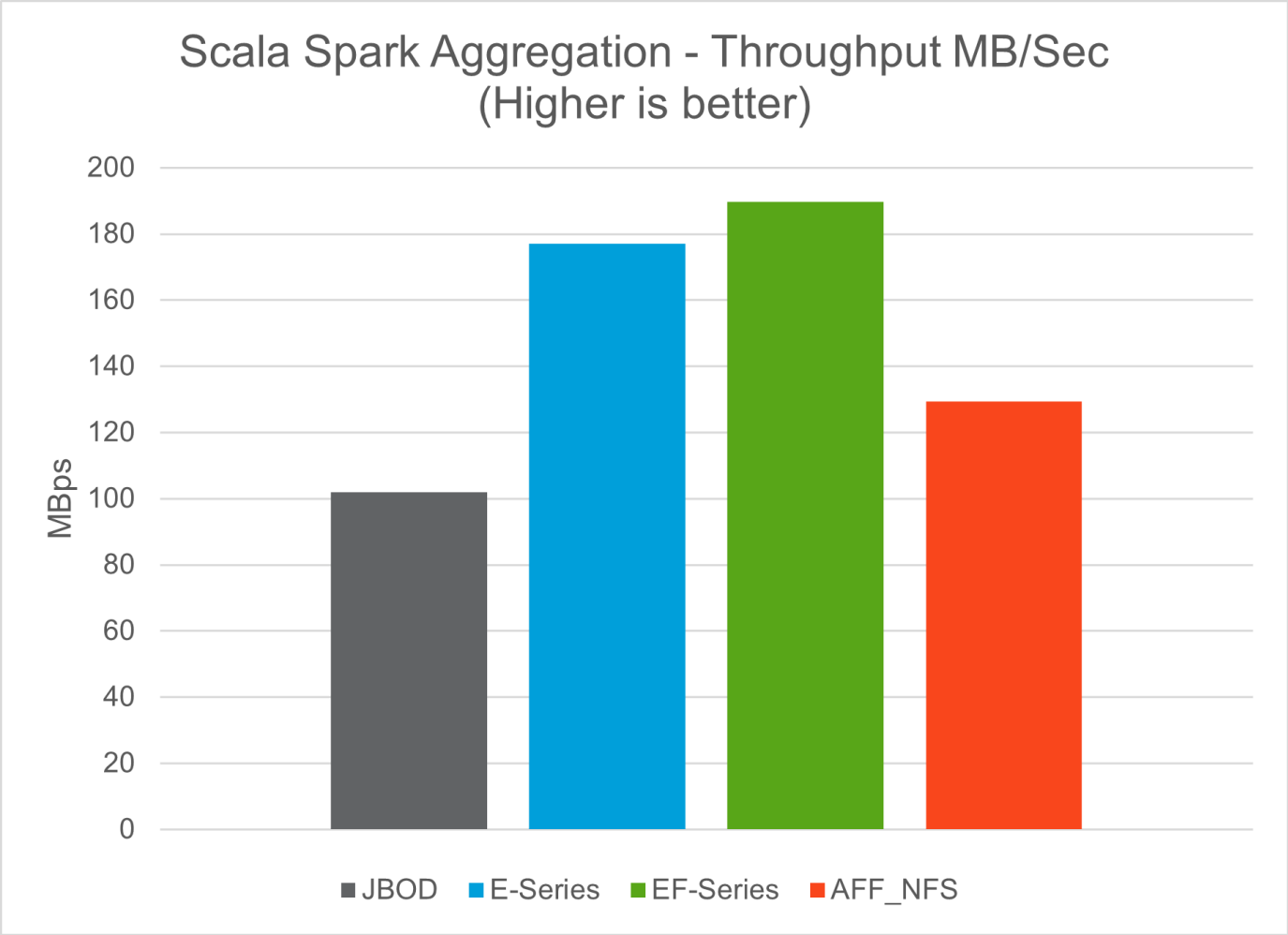
```
(base) [root@n138 examples]# ~/anaconda3/bin/spark-submit
--master yarn
--executor-memory 15g
--executor-cores 1
--num-executors 160
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py
--data-dir file:///sparkdemo/tr-4570-data
> . /run_classification_criteo_spark_nfs.log 2>&1
```

10번의 훈련 에포크 후에 우리는 테스트 데이터 세트에 대한 AUC 점수를 얻었습니다.

```
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
```

이전 사용 사례와 유사한 방식으로 Spark 워크플로 런타임을 다양한 위치에 있는 데이터와 비교했습니다. 다음 그림은

Spark 워크플로 런타임에 대한 딥 러닝 CTR 예측을 비교한 것입니다.

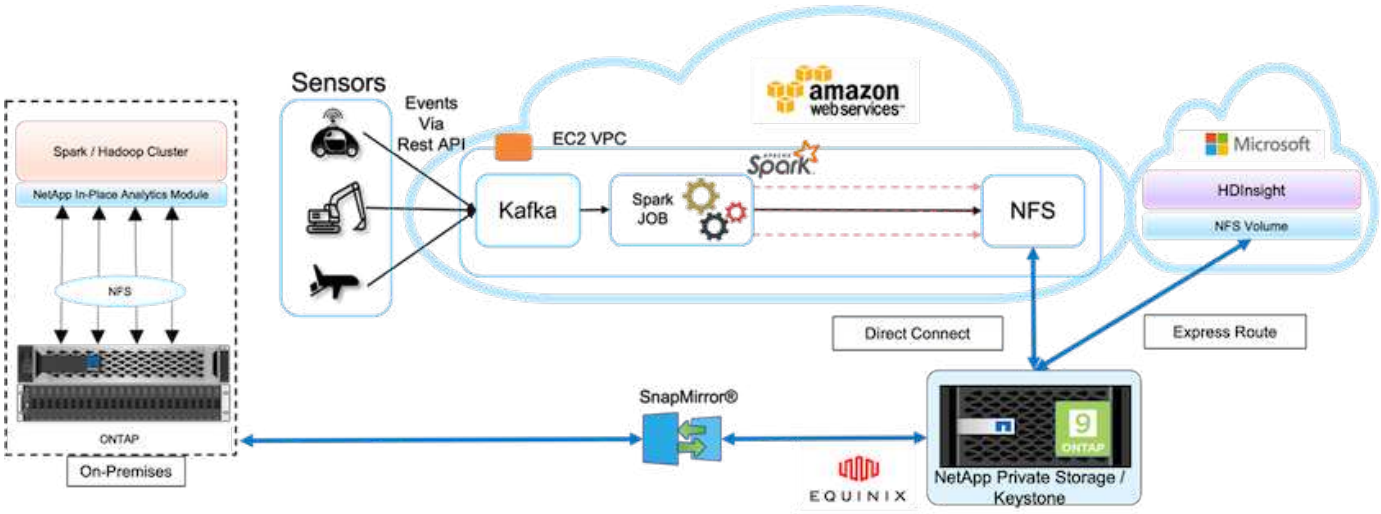


하이브리드 클라우드 솔루션

최신 기업 데이터 센터는 일관된 운영 모델을 갖춘 지속적인 데이터 관리 평면을 통해 여러 분산 인프라 환경을 온프레미스 및/또는 여러 퍼블릭 클라우드에 연결하는 하이브리드 클라우드입니다. 하이브리드 클라우드를 최대한 활용하려면 데이터 변환이나 애플리케이션 리팩토링이 필요 없이 온프레미스와 멀티클라우드 환경 간에 데이터를 원활하게 이동할 수 있어야 합니다.

고객들은 데이터 보호와 같은 사용 사례를 위해 보조 스토리지를 클라우드로 옮기거나 애플리케이션 개발 및 DevOps와 같은 비즈니스에 덜 중요한 워크로드를 클라우드로 옮기는 것으로 하이브리드 클라우드 여정을 시작한다고 밝혔습니다. 그런 다음 더 중요한 작업으로 넘어갑니다. 웹 및 콘텐츠 호스팅, DevOps 및 애플리케이션 개발, 데이터베이스, 분석, 컨테이너화된 앱은 가장 인기 있는 하이브리드 클라우드 워크로드에 속합니다. 기업 AI 프로젝트의 복잡성, 비용, 위험은 역사적으로 실험 단계에서 생산 단계로 AI 도입을 방해해 왔습니다.

NetApp 하이브리드 클라우드 솔루션을 통해 고객은 분산 환경 전반에서 데이터 및 워크플로 관리를 위한 단일 제어판을 통해 통합 보안, 데이터 거버넌스 및 규정 준수 도구의 이점을 누릴 수 있으며, 소비량에 따라 총 소유 비용을 최적화할 수 있습니다. 다음 그림은 고객의 빅데이터 분석 데이터에 대한 멀티클라우드 연결을 제공하는 업무를 맡은 클라우드 서비스 파트너의 솔루션 사례입니다.



이 시나리오에서는 다양한 소스에서 AWS로 수신된 IoT 데이터가 NetApp Private Storage(NPS)의 중앙 위치에 저장됩니다. NPS 스토리지는 AWS와 Azure에 있는 Spark 또는 Hadoop 클러스터에 연결되어 여러 클라우드에서 실행되는 빅데이터 분석 애플리케이션이 동일한 데이터에 액세스할 수 있도록 합니다. 이 사용 사례에 대한 주요 요구 사항과 과제는 다음과 같습니다.

- 고객은 여러 클라우드를 사용하여 동일한 데이터에 대한 분석 작업을 실행하려고 합니다.
- 데이터는 다양한 센서와 허브를 통해 온프레미스 및 클라우드 환경 등 다양한 소스에서 수신되어야 합니다.
- 해결책은 효율적이고 비용 효율적이어야 합니다.
- 가장 큰 과제는 온프레미스와 클라우드 환경 간에 하이브리드 분석 서비스를 제공하는 비용 효율적이고 효과적인 솔루션을 구축하는 것입니다.

당사의 데이터 보호 및 멀티클라우드 연결 솔루션은 여러 하이퍼스케일러에 걸쳐 클라우드 분석 애플리케이션을 운영하는 데 따르는 문제점을 해결합니다. 위 그림에서 볼 수 있듯이, 센서의 데이터는 Kafka를 통해 AWS Spark 클러스터로 스트리밍되고 수집됩니다. 데이터는 클라우드 제공자 외부의 Equinix 데이터 센터 내에 있는 NPS에 있는 NFS 공유에 저장됩니다.

NetApp NPS는 Direct Connect 및 Express Route 연결을 통해 각각 Amazon AWS 및 Microsoft Azure에 연결되어 있으므로 고객은 In-Place Analytics Module을 활용하여 Amazon 및 AWS 분석 클러스터의 데이터에 액세스할 수 있습니다. 따라서 온프레미스와 NPS 스토리지 모두 ONTAP 소프트웨어를 실행하기 때문에 "SnapMirror" NPS 데이터를 온프레미스 클러스터로 미러링하여 온프레미스와 여러 클라우드에서 하이브리드 클라우드 분석을 제공할 수 있습니다.

최상의 성능을 위해 NetApp 일반적으로 여러 네트워크 인터페이스와 직접 연결 또는 고속 경로를 사용하여 클라우드 인스턴스의 데이터에 액세스할 것을 권장합니다. 다음을 포함한 다른 데이터 이동 솔루션이 있습니다. "엑스피" 그리고 "BlueXP 복사 및 동기화" 고객이 애플리케이션 인식, 보안 및 비용 효율적인 하이브리드 클라우드 Spark 클러스터를 구축할 수 있도록 지원합니다.

각 주요 사용 사례에 대한 **Python** 스크립트

다음 세 가지 Python 스크립트는 테스트된 세 가지 주요 사용 사례에 해당합니다. 첫 번째는 `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}")
```

두 번째 스크립트는 `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()
```

세 번째 스크립트는 `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))
```

결론

이 문서에서는 Apache Spark 아키텍처, 고객 사용 사례, 그리고 빅데이터, 최신 분석, AI, ML,

DL과 관련된 NetApp 스토리지 포트폴리오에 대해 설명합니다. 업계 표준 벤치마킹 도구와 고객 수요에 따른 성능 검증 테스트에서 NetApp Spark 솔루션은 기본 Hadoop 시스템에 비해 뛰어난 성능을 보였습니다. 이 보고서에 제시된 고객 사용 사례와 성능 결과를 조합하면 배포에 적합한 Spark 솔루션을 선택하는 데 도움이 될 수 있습니다.

추가 정보를 찾을 수 있는 곳

이 TR에서는 다음 참고문헌이 사용되었습니다.

- Apache Spark 아키텍처 및 구성 요소

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

- Apache Spark 사용 사례

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

- 스파크 NLP

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

- 버트

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

- 광고 클릭 예측을 위한 심층 및 교차 네트워크

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

- FlexGroup

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

- 스트리밍 ETL

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

- Hadoop을 위한 NetApp E-Series 솔루션

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

- NetApp 최신 데이터 분석 솔루션

  "데이터 분석 솔루션"

- SnapMirror

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

- 엑스피

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

- BlueXP 복사 및 동기화

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

- DataOps 툴킷

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