Apache Spark용 NetApp 스토리지 솔루션

NetApp Solutions

NetApp
April 20, 2024

This PDF was generated from https://docs.netapp.com/ko-kr/netapp-solutions/data-analytics/apache-spark-solution-overview.html on April 20, 2024. Always check docs.netapp.com for the latest.

목차

Apache Spark용 NetApp 스토리지 솔루션 . . . 1
TR-4570: Apache Spark용 NetApp 스토리지 솔루션: 아키텍처, 사용 사례 및 성능 결과 . . . 1
대상 . . . 5
솔루션 기술 . . . 6
NetApp Spark 솔루션 개요 . . . 7
사용 사례 요약 . . . 9
주요 AI, ML 및 DL 사용 사례 및 아키텍처 . . . 11
테스트 결과 . . . 15
하이브리드 클라우드 솔루션 . . . 26
주요 활용 사례별로 Python 스크립트 . . . 27
결론 . . . 45
추가 정보를 찾을 수 있는 위치 . . . 46

Apache Spark용 NetApp 스토리지 솔루션

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

Rick Huang, Karthikeyan Nagalingam, NetApp

이 문서에서는 Apache Spark 아키텍처, 고객 사용 사례 및 빅데이터 분석 및 인공 지능(AI)과 관련된 NetApp 스토리지 포트폴리오에 대해 중점적으로 소개합니다. 또한 적절한 Spark 솔루션을 선택할 수 있도록 업계 표준 AI, 머신 러닝(ML) 및 딥 러닝(DL) 툴을 일반적인 Hadoop 시스템과 비교하여 다양한 테스트 결과를 제공합니다. 시작하려면 Spark 아키텍처, 적절한 구성 요소 및 두 가지 배포 모드(클러스터 및 클라이언트)가 필요합니다.

또한 이 문서는 구성 문제를 해결하기 위한 고객 사용 사례를 제공하며 빅데이터 분석 및 AI, ML, Spark를 지원하는 DL과 관련된 NetApp 스토리지 포트폴리오의 개요를 설명합니다. 그런 다음 Spark 관련 사용 사례 및 NetApp Spark 솔루션 포트폴리오의 테스트 결과를 마무리합니다.

고객의 당면 과제

이 섹션에서는 소매, 디지털 마케팅, 은행, 이산 제조, 프로세스 제조 등 데이터 성장 산업에서 빅데이터 분석 및 AI/ML/DL과 관련된 고객 과제에 대해 정부 및 전문 서비스.

성능을 예측할 수 없습니다

기존 Hadoop 구축에는 일반적으로 일반 하드웨어가 사용됩니다. 성능을 향상시키려면 네트워크, 운영 체제, Hadoop 클러스터, Spark와 같은 에코시스템 구성 요소 및 하드웨어를 조정해야 합니다. 각 계층을 튜닝하더라도 Hadoop은 사용자 환경에서 고성능을 발휘하도록 설계되지 않은 상용 하드웨어에서 실행되기 때문에 원하는 성능 수준을 달성하는 것이 어려울 수 있습니다.

미디어 및 노드 장애

정상적인 조건에서 상용 하드웨어는 장애가 발생하기 쉽습니다. 데이터 노드의 한 디스크에 장애가 발생하면 기본적으로 Hadoop 마스터는 해당 노드가 정상 상태가 아닌 것으로 간주합니다. 그런 다음 네트워크를 통해 해당 노드의 특정 데이터를 복제본에서 정상 노드로 복제합니다. 이 프로세스는 모든 Hadoop 작업에 대한 네트워크 패킷의 속도를 늦춥니다. 그런 다음, 정상적인 상태가 아닌 노드가 정상 상태가 될 때 클러스터가 데이터를 다시 복제하고 초과 복제된 데이터를 제거해야 합니다.

Hadoop 공급업체에 종속

Hadoop 총판은 자체 버전 관리 기능을 통해 Hadoop을 직접 배포하며, 고객은 이 버전을 통해 해당 배포판에 종속됩니다. 그러나 많은 고객들은 특정 Hadoop 배포와 고객을 연계하지 않는 인메모리 분석에 대한 지원을 필요로 합니다. 따라서 고객이 원하는 대로 배포를 변경하고 분석을 실행할 수 있어야 합니다.

둘 이상의 언어를 지원하지 않습니다

고객은 업무를 실행하기 위해 MapReduce Java 프로그램 외에 여러 언어에 대한 지원을 필요로 하는 경우가 많습니다. SQL 및 스크립트와 같은 옵션을 사용하면 답변을 보다 유연하게 얻고, 데이터를 구성 및 검색하는 더 많은 옵션을 사용할 수 있으며, 데이터를 분석 프레임워크로 신속하게 이동할 수 있습니다.

사용의 어려움

Hadoop을 사용하기 어렵다고 불만을 토로하는 사람들이 있습니다. 새로운 버전이 출시될 때마다 Hadoop이 더욱 단순해지고 강력해지기는 했지만 이러한 비판은 지속되었습니다. Hadoop을 사용하려면 Java 및 MapReduce 프로그래밍 패턴을 이해해야 합니다. 이는 데이터베이스 관리자와 기존 스크립팅 기술을 사용하는 사람들에게 어려운 과제입니다.

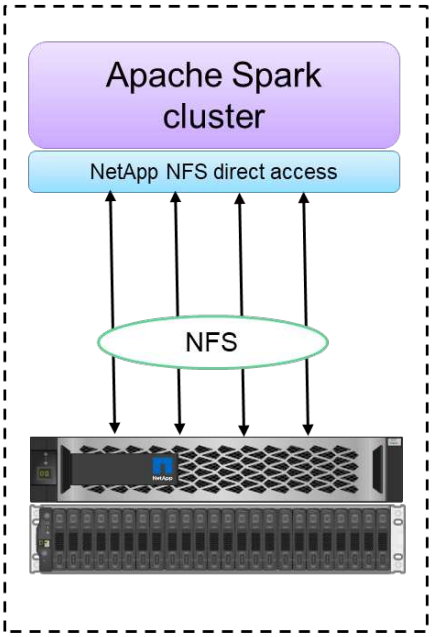
복잡한 프레임워크 및 도구

엔터프라이즈 AI 팀은 여러 가지 과제에 직면합니다. 전문 데이터 과학 지식을 갖추고 있더라도 다양한 구축 에코시스템과 애플리케이션을 위한 툴과 프레임워크에서 다른 에코시스템으로 변환되지 않을 수 있습니다. 데이터 과학 플랫폼은 Spark 기반의 해당 빅 데이터 플랫폼과 원활하게 통합되어야 하며, 간편한 데이터 이동, 재사용 가능 모델, 즉시 사용 가능한 코드, 프로토타입 제작, 검증, 버전 관리, 공유, 재사용, 운영 환경에 모델을 빠르게 구축할 수 있습니다.

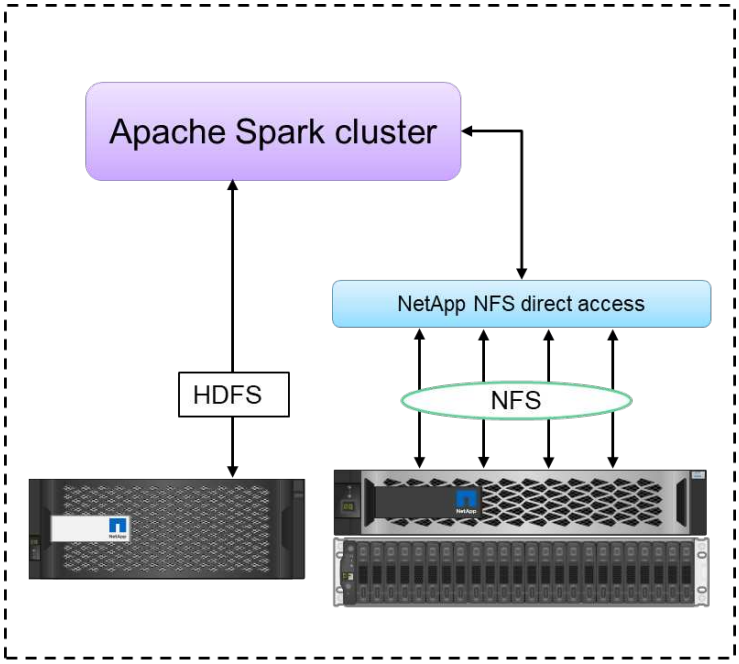
NetApp을 선택해야 하는 이유

NetApp은 다음과 같은 방법으로 Spark 경험을 개선할 수 있습니다.

- NetApp NFS 직접 액세스(아래 그림에 표시)를 사용하면 데이터를 이동하거나 복사하지 않고도 기존 또는 새로운 NFSv3 또는 NFSv4 데이터에 대해 빅데이터 분석 작업을 실행할 수 있습니다. 여러 데이터 복제본을 방지하고 소스와 데이터를 동기화할 필요가 없습니다.
- 스토리지 효율성 향상 및 서버 복제 감소 예를 들어, NetApp E-Series Hadoop 솔루션은 3개의 데이터 복제본이 아닌 2개의 복제본을 필요로 하며 FAS Hadoop 솔루션은 데이터 소스만 필요로 하지만 데이터의 복제 또는 복제본은 필요로 하지 않습니다. NetApp 스토리지 솔루션은 또한 서버 간 트래픽을 줄여 줍니다.
- 드라이브 및 노드 장애 시 Hadoop 작업 및 클러스터 동작이 향상됩니다.
- 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 부문에서 데이터를 한 위치에서 다른 위치로 이동하는 것은 쉬운 일이 아닌 법적 의무를 준수해야 합니다. 이 시나리오에서는 NetApp NFS 직접 액세스가 원래의 위치에서 재무 및 의료 데이터를 분석합니다. 또 다른 주요 이점은 NetApp NFS 직접 액세스를 통해 기본 Hadoop 명령을 사용하여 Hadoop 데이터를 간편하게

보호하고 NetApp의 강력한 데이터 관리 포트폴리오를 통해 데이터 보호 워크플로우를 구축할 수 있다는 것입니다.

NetApp NFS 직접 액세스는 Hadoop/Spark 클러스터에 대한 두 가지 유형의 구축 옵션을 제공합니다.

- 기본적으로 Hadoop 또는 Spark 클러스터는 데이터 스토리지와 기본 파일 시스템에 HDFS(Hadoop Distributed File System)를 사용합니다. NetApp NFS 직접 액세스는 기본 HDFS를 NFS 스토리지로 대체하여 NFS 데이터에 대한 직접 분석을 지원합니다.
- 다른 구축 옵션에서 NetApp NFS 직접 액세스는 NFS를 단일 Hadoop 또는 Spark 클러스터의 HDFS와 함께 추가 스토리지로 구성할 수 있도록 지원합니다. 이 경우 고객은 NFS 내보내기를 통해 데이터를 공유하고 HDFS 데이터와 함께 동일한 클러스터에서 데이터를 액세스할 수 있습니다.

NetApp NFS 직접 액세스를 사용할 때의 주요 이점은 다음과 같습니다.

- 현재 위치에서 데이터를 분석하면 분석 데이터를 HDFS와 같은 Hadoop 인프라스트럭처로 이동하는 데 시간과 성능 소모가 큰 작업이 발생하지 않습니다.
- 복제본 수를 3개부터 1개로 축소
- 사용자가 컴퓨팅과 스토리지를 분리하여 독립적으로 확장할 수 있도록 지원
- ONTAP의 강력한 데이터 관리 기능을 활용하여 엔터프라이즈 데이터 보호 제공
- Hortonworks 데이터 플랫폼을 사용한 인증
- 하이브리드 데이터 분석을 구축할 수 있습니다.
- 동적 다중 스레드 기능을 활용하여 백업 시간 단축

을 참조하십시오 "TR-4657: NetApp 하이브리드 클라우드 데이터 솔루션 - 고객 사용 사례를 기반으로 Spark 및 Hadoop" Hadoop 데이터, 클라우드에서 사내로 백업 및 재해 복구, 기존 Hadoop 데이터에 대한 DevTest, 데이터 보호 및 멀티 클라우드 연결, 분석 워크로드 가속화 등을 지원합니다.

다음 섹션에서는 Spark 고객에게 중요한 스토리지 기능에 대해 설명합니다.

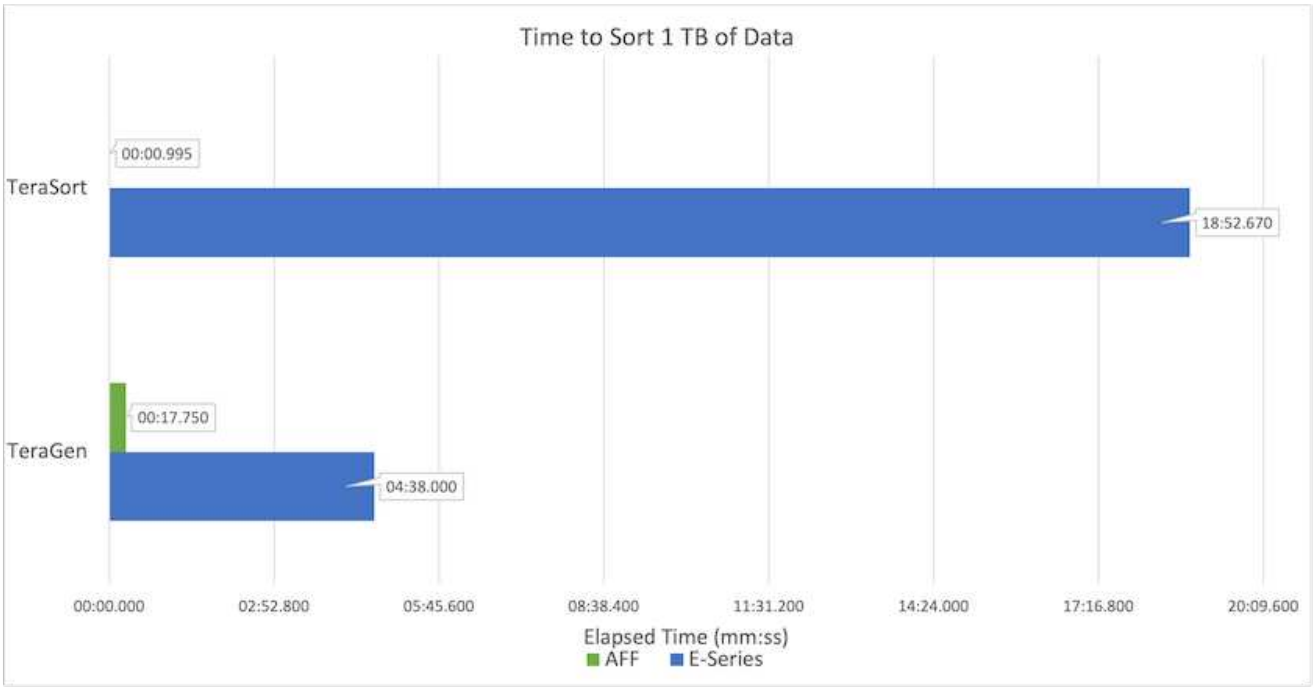
스토리지 계층화

Hadoop 스토리지 계층화를 사용하면 스토리지 정책에 따라 다양한 스토리지 유형의 파일을 저장할 수 있습니다. 스토리지 유형으로는 핫, 콜드, 웜, 올 SSD, 원 SSD 등이 있습니다. 그리고 게으른 유영이 있습니다.

<<<<<<< head는 NetApp AFF 스토리지 컨트롤러에서 Hadoop 스토리지 계층화를 검증하고 SSD 및 SAS 드라이브를 사용하는 E-Series 스토리지 컨트롤러를 다양한 스토리지 정책으로 수행했습니다. AFF-A800의 Spark 클러스터에는 4개의 컴퓨팅 작업자 노드가 있는 반면 E-Series를 사용하는 클러스터는 8개의 노드를 가지고 있습니다. 주로 SSD(Solid-State Drive)와 HDD(하드 드라이브 디스크)의 성능을 비교합니다.

NetApp AFF 스토리지 컨트롤러에서 Hadoop 스토리지 계층화를 검증하고 SSD 및 SAS 드라이브를 사용하는 E-Series 스토리지 컨트롤러를 다양한 스토리지 정책으로 수행했습니다. AFF-A800의 Spark 클러스터에는 4개의 컴퓨팅 작업자 노드가 있는 반면 E-Series를 사용하는 클러스터는 8개의 노드를 가지고 있습니다. 주로 SSD와 하드 드라이브 디스크의 성능을 비교하기 위해 이 작업을 수행하였습니다. >>>>>> a51c9ddf73ca69e1120ce05edc7b0b9607b96eae

다음 그림은 Hadoop SSD를 위한 NetApp 솔루션의 성능을 보여줍니다.



- 기본 NL-SAS 구성에는 8개의 컴퓨팅 노드와 96개의 NL-SAS 드라이브가 사용되었습니다. 이 구성에서는 4분 38초 내에 1TB의 데이터가 생성되었습니다. 을 참조하십시오 "TR-3969 Hadoop용 NetApp E-Series 솔루션" 클러스터 및 스토리지 구성에 대한 자세한 내용은 를 참조하십시오.
- TeraGen을 사용하면 SSD 구성에서 NL-SAS 구성보다 1TB의 데이터가 15.66x 더 빠르게 생성됩니다. 또한 SSD 구성에서는 컴퓨팅 노드 수의 절반과 디스크 드라이브 수의 절반을 사용했습니다(총 24개의 SSD 드라이브). 작업 완료 시간을 기준으로 할 때 NL-SAS 구성의 속도는 약 2배였습니다.
- TeraSort를 사용하면 SSD 구성에서 NL-SAS 구성에 비해 1TB의 데이터를 1138.36배 더 빠르게 정렬할 수 있습니다. 또한 SSD 구성에서는 컴퓨팅 노드 수의 절반과 디스크 드라이브 수의 절반을 사용했습니다(총 24개의 SSD 드라이브). 따라서 드라이브당 NL-SAS 구성보다 약 3배 빠른 속도를 제공합니다. <<<<<<< 머리
- 여기서 주목할 점은 회전식 디스크에서 All-Flash로 전환하여 성능을 향상할 수 있다는 것입니다. 컴퓨팅 노드의 수는 병목 현상이 아니었습니다. NetApp의 All-Flash 스토리지를 사용하면 런타임 성능이 원활하게 확장됩니다.
- NFS를 사용하면 데이터가 모두 함께 풀링되는 것과 기능적으로 동일하므로 워크로드에 따라 컴퓨팅 노드의 수를 줄일 수 있습니다. Apache Spark 클러스터 사용자는 컴퓨팅 노드의 수를 변경할 때 데이터를 수동으로 재조정할 필요가 없습니다.

- 즉, 회전식 디스크에서 All-Flash로 전환하면 성능이 향상됩니다. 컴퓨팅 노드의 수는 병목 현상이 아니었습니다. NetApp All-Flash 스토리지를 사용하면 런타임 성능이 원활하게 확장됩니다.
- NFS를 사용하면 데이터가 모두 함께 풀링되는 것과 기능적으로 동일하므로 워크로드에 따라 컴퓨팅 노드의 수를

줄일 수 있습니다. Apache Spark 클러스터 사용자는 컴퓨팅 노드 수를 변경할 때 데이터를 수동으로 재조정할 필요가 없습니다. >>>>>> a51c9ddf73ca69e1120ce05edc7b0b9607b96eae

성능 확장 - 스케일아웃

AFF 솔루션에서 Hadoop 클러스터의 컴퓨팅 성능이 더 많이 필요한 경우 적절한 수의 스토리지 컨트롤러를 사용하여 데이터 노드를 추가할 수 있습니다. 스토리지 컨트롤러 어레이당 4개의 데이터 노드로 시작하고 워크로드 특성에 따라 스토리지 컨트롤러당 8개의 데이터 노드로 숫자를 늘리는 것이 좋습니다.

AFF와 FAS는 데이터 이동 없는 분석에 적합합니다. 계산 요구사항에 따라 노드 관리자를 추가할 수 있으며 무중단 운영을 통해 다운타임 없이 스토리지 컨트롤러를 온디맨드 방식으로 추가할 수 있습니다. NetApp은 AFF NVMe 미디어 지원, 효율성 보장, 데이터 축소, QoS, 예측 분석, FAS 클라우드 계층화, 복제, 클라우드 구축, 보안 고객의 요구사항을 충족할 수 있도록 NetApp에서는 추가 라이센스 비용 없이 파일 시스템 분석, 할당량, 온박스 로드 밸런싱과 같은 기능을 제공합니다. NetApp은 동시 작업 수, 낮은 지연 시간, 단순한 운영, 경쟁업체보다 더 높은 초당 처리 성능 등의 이점을 제공합니다. 또한 NetApp Cloud Volumes ONTAP은 3가지 주요 클라우드 공급자 모두에서 실행됩니다.

성능 확장 - 스케일업

스케일업 기능을 사용하면 스토리지 용량이 더 필요할 때 AFF, FAS, E-Series 시스템에 디스크 드라이브를 추가할 수 있습니다. Cloud Volumes ONTAP를 사용하면 자주 사용되지 않는 데이터를 블록 스토리지의 오브젝트 스토리지로 계층화하고, 추가 컴퓨팅 없이 Cloud Volumes ONTAP 라이센스를 스태킹하는 두 가지 요소의 조합으로 스토리지를 PB 수준으로 확장할 수 있습니다.

다중 프로토콜

NetApp 시스템은 SAS, iSCSI, FCP, InfiniBand를 비롯한 대부분의 Hadoop 구현 프로토콜을 및 NFS 로 이동합니다.

운영 및 지원 솔루션

이 문서에 설명된 Hadoop 솔루션은 NetApp에서 지원됩니다. 또한 이러한 솔루션은 주요 Hadoop 총판에서도 인증되었습니다. 자세한 내용은 를 참조하십시오 "MapR" 사이트 "Hortonworks의" 사이트 및 Cloudera "인증" 및 "파트너" 있습니다.

대상

분석 및 데이터 과학의 세계는 IT와 비즈니스의 여러 분야를 아우릅니다.

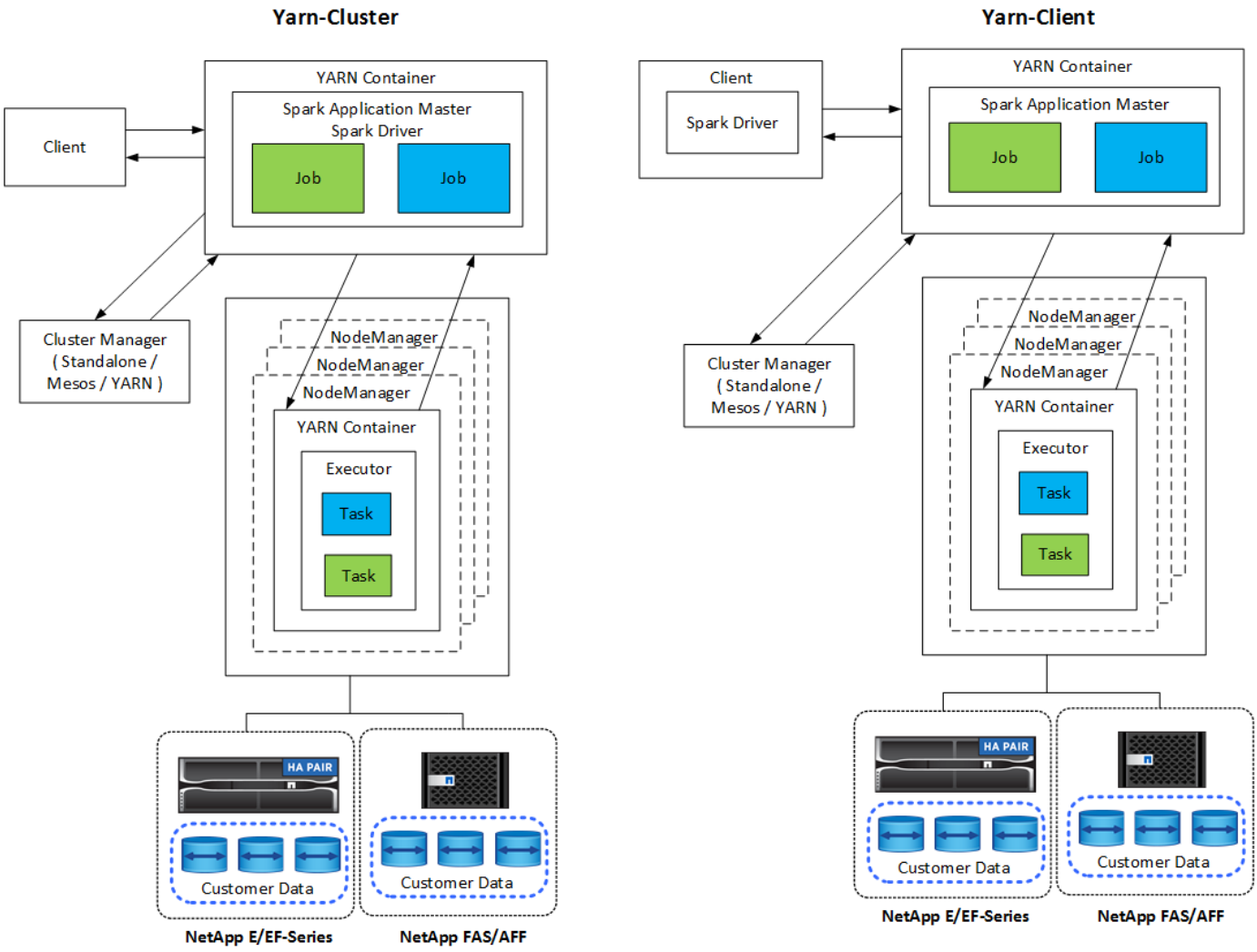
- 데이터 과학자는 자신이 선택한 도구와 라이브러리를 사용할 수 있는 유연성이 필요합니다.
- 데이터 엔지니어는 데이터 흐름과 데이터 위치를 알아야 합니다.
- DevOps 엔지니어는 새로운 AI 및 ML 애플리케이션을 CI 및 CD 파이프라인에 통합하는 툴을 필요로 합니다.
- 클라우드 관리자와 설계자는 하이브리드 클라우드 리소스를 설정하고 관리할 수 있어야 합니다.
- 비즈니스 사용자는 분석, AI, ML 및 DL 애플리케이션에 액세스해야 합니다.

이 기술 보고서에서는 NetApp AFF, E-Series, StorageGRID, NFS 직접 액세스, Apache Spark가 어떤 식으로 Horovod와 Keras는 이러한 각 역할이 비즈니스에 가치를 제공하는 데 도움을 줍니다.

솔루션 기술

Apache Spark는 HDFS(Hadoop Distributed File System)와 직접 연동되는 Hadoop 애플리케이션을 작성하기 위한 인기 있는 프로그래밍 프레임워크입니다. Spark는 프로덕션을 지원하며 스트리밍 데이터의 처리를 지원하며 MapReduce보다 빠릅니다. Spark는 효율적인 반복을 위해 메모리 내 데이터 캐싱을 구성할 수 있으며 Spark 셸은 데이터를 학습하고 탐색하는 대화형 기능을 제공합니다. Spark를 사용하면 Python, Scala 또는 Java로 애플리케이션을 만들 수 있습니다. SPARK 응용 프로그램은 하나 이상의 작업이 있는 하나 이상의 작업으로 구성됩니다.

모든 Spark 응용 프로그램에는 Spark 드라이버가 있습니다. YARN-Client 모드에서 드라이버는 클라이언트에서 로컬로 실행됩니다. YARN-Cluster 모드에서는 드라이버가 애플리케이션 마스터의 클러스터에서 실행됩니다. 클러스터 모드에서는 클라이언트의 연결이 끊기더라도 애플리케이션이 계속 실행됩니다.



세 가지 클러스터 관리자가 있습니다.

- * 독립 실행형. * 이 매니저는 Spark의 일부이며, 클러스터를 쉽게 설정할 수 있습니다.
- * Apache Mesos. * MapReduce 및 기타 애플리케이션도 실행하는 일반 클러스터 관리자입니다.
- * Hadoop YARN. * Hadoop 3의 리소스 관리자입니다.

탄력적인 분산 데이터 세트(RDD)는 Spark의 주요 구성 요소입니다. RDD는 클러스터의 메모리에 저장된 데이터에서 손실되거나 누락된 데이터를 재생성하고 파일에서 나오거나 프로그래밍 방식으로 생성된 초기 데이터를 저장합니다. RDD는 파일, 메모리에 있는 데이터 또는 다른 RDD에서 생성됩니다. SPARK 프로그래밍은 변환과 동작이라는 두 가지 작업을 수행합니다. 변환은 기존 RDD를 기반으로 새 RDD를 생성합니다. 작업은 RDD에서 값을 반환합니다.

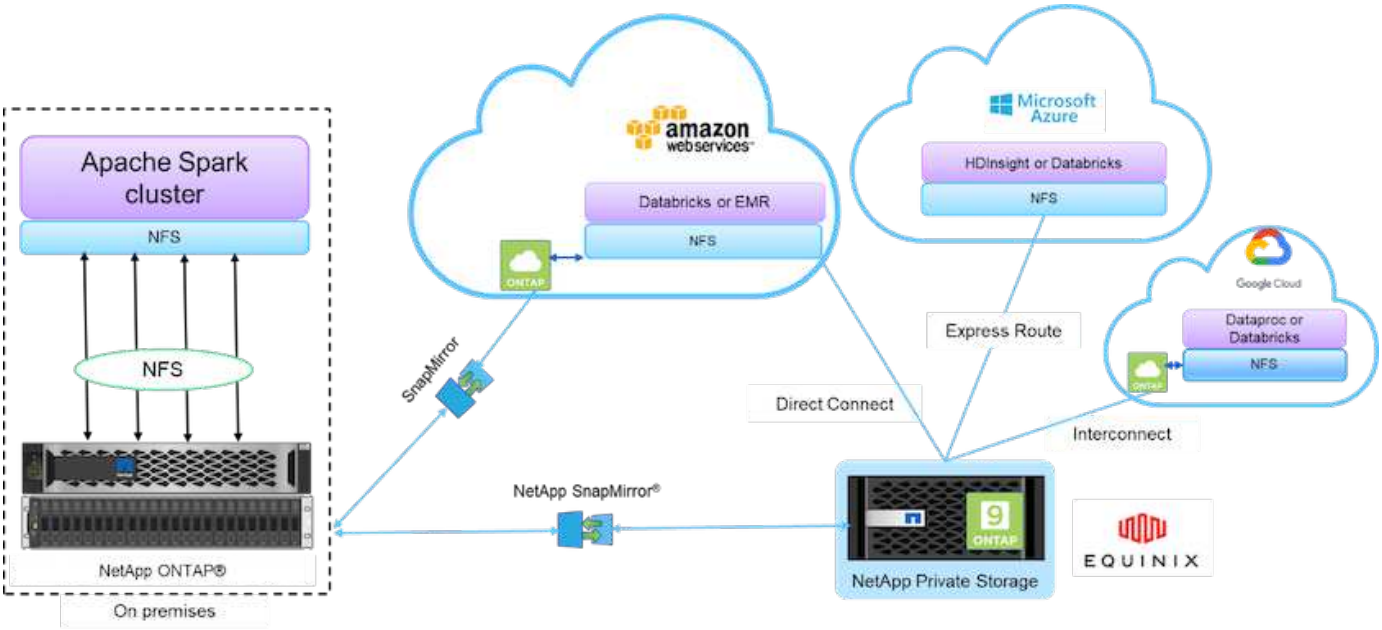
변환 및 작업은 Spark 데이터 집합 및 DataFrames에도 적용됩니다. 데이터 세트는 분산 데이터 모음으로서 Spark SQL의 최적화된 실행 엔진의 이점과 함께 RDD(강력한 타이핑, 람다 함수 사용)의 이점을 제공합니다. 데이터세트는 JVM 객체에서 생성한 다음 기능 변환(맵, 평면 맵, 필터 등)을 사용하여 조작할 수 있습니다. DataFrame 은 명명된 열로 구성된 데이터 집합입니다. 개념적으로 관계형 데이터베이스의 테이블 또는 R/Python의 데이터 프레임에 해당합니다. DataFrames는 정형 데이터 파일, Hive/HBase의 테이블, 사내 또는 클라우드의 외부 데이터베이스 또는 기존 RDD와 같은 다양한 소스에서 구성할 수 있습니다.

스파크 응용 프로그램에는 하나 이상의 스파크 작업이 포함됩니다. 작업은 실행기에서 작업을 실행하고 실행자는 YARN 컨테이너에서 실행됩니다. 각 실행자는 단일 컨테이너에서 실행되며 실행자는 응용 프로그램 수명 내내 존재합니다. 응용 프로그램이 시작된 후 실행자가 수정되고 YARN은 이미 할당된 컨테이너의 크기를 조정하지 않습니다. 집행자는 인메모리 데이터에 대해 작업을 동시에 실행할 수 있습니다.

NetApp Spark 솔루션 개요

NetApp은 FAS/AFF, E-Series, Cloud Volumes ONTAP의 3가지 스토리지 포트폴리오를 보유하고 있습니다. Apache Spark를 지원하는 Hadoop 솔루션을 위한 ONTAP 스토리지 시스템을 통해 AFF 및 E-Series를 검증했습니다.

NetApp 기반의 Data Fabric은 아래 그림과 같이 데이터 액세스, 제어, 보호 및 보안을 위한 데이터 관리 서비스와 애플리케이션(구성 요소)을 통합합니다.



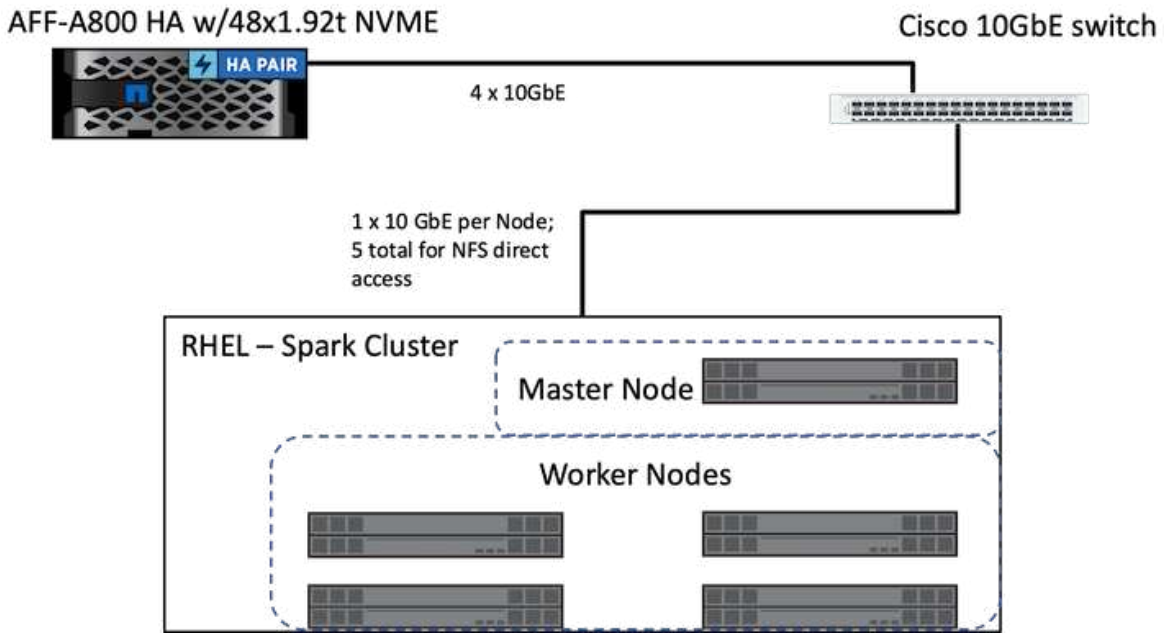
위 그림의 구성 요소는 다음과 같습니다.

- * NetApp NFS 직접 액세스 * 는 추가 소프트웨어 또는 드라이버 요구사항 없이 최신 Hadoop 및 Spark 클러스터를 NetApp NFS 볼륨에 직접 액세스할 수 있도록 지원합니다.
- * NetApp Cloud Volumes ONTAP 및 클라우드 볼륨 서비스. * Microsoft Azure 클라우드 서비스의 AWS(Amazon Web Services) 또는 ANF(Azure NetApp Files)에서 실행되는 ONTAP를 기반으로 하는

소프트웨어 정의 연결형 스토리지.

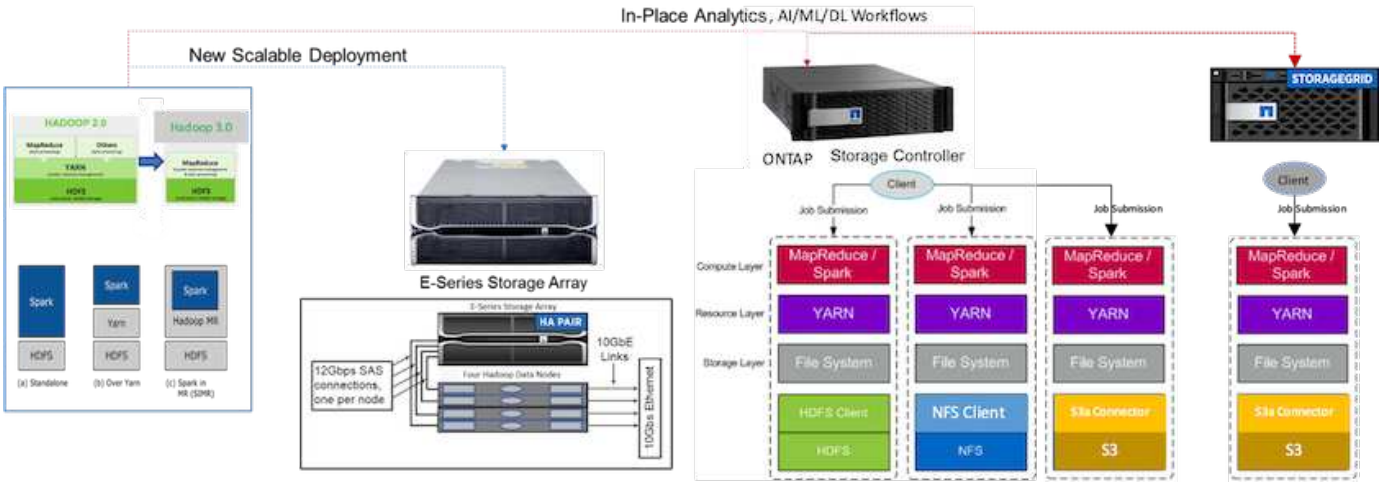
- * NetApp SnapMirror 기술. * 사내 및 ONTAP 클라우드 또는 NPS 인스턴스 간에 데이터 보호 기능을 제공합니다.
- * 클라우드 서비스 공급자 * 이러한 공급자에는 AWS, Microsoft Azure, Google Cloud 및 IBM Cloud가 포함됩니다.
- * PaaS. * AWS의 EMR(Amazon Elastic MapReduce) 및 Databricks와 같은 클라우드 기반 분석 서비스와 Microsoft Azure HDInsight 및 Azure Databricks

다음 그림은 NetApp 스토리지를 포함한 Spark 솔루션을 보여 줍니다.

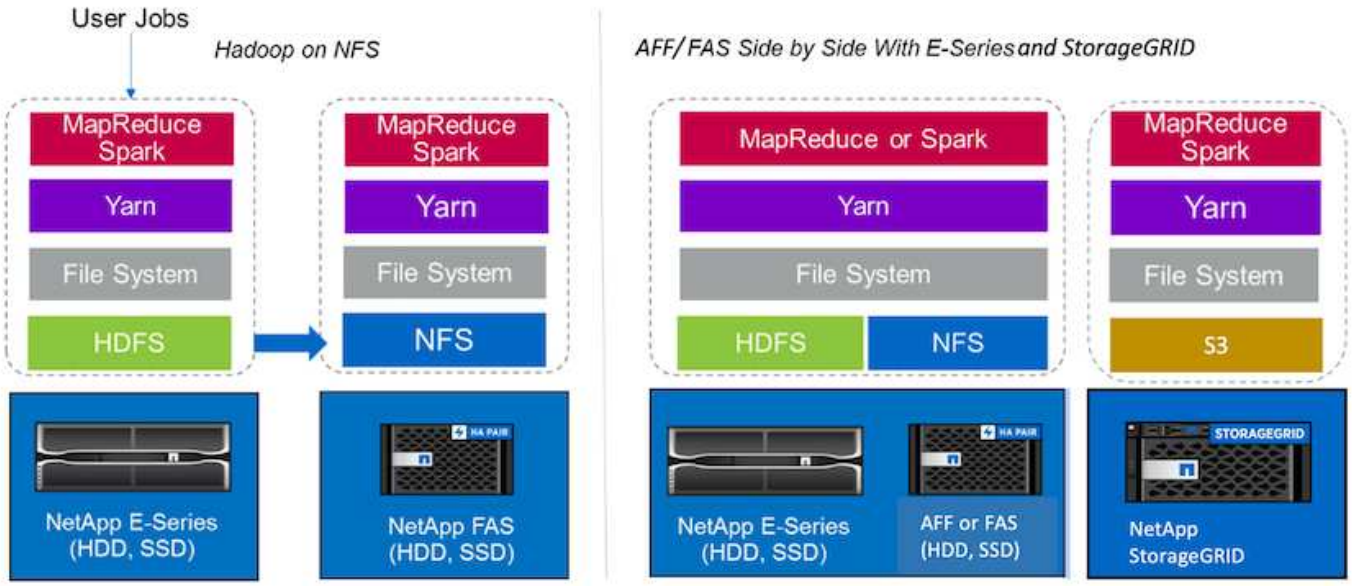


ONTAP Spark 솔루션은 NetApp NFS 직접 액세스 프로토콜을 사용하여 기존 운영 데이터에 대한 액세스를 통해 데이터 이동 없이 분석 및 AI, ML 및 DL 워크플로우를 지원합니다. Hadoop 노드에 사용 가능한 운영 데이터를 내보내 데이터 이동 없이 분석 및 AI, ML, DL 작업을 수행합니다. Hadoop 노드에서 처리하는 데이터에 NetApp NFS 직접 액세스 또는 사용하지 않고 액세스할 수 있습니다. 독립 실행형 또는 'YARN' 클러스터 관리자를 사용하여 Spark에서 ' 을 사용하여 NFS 볼륨을 구성할 수 있습니다<file:///<target_volume>`. 서로 다른 데이터 세트를 사용하여 세 가지 사용 사례를 검증했습니다. 이러한 검증에 대한 자세한 내용은 "테스트 결과" 섹션에 나와 있습니다. (Xref)

다음 그림은 NetApp Apache Spark/Hadoop 스토리지 포지셔닝을 보여줍니다.



E-Series Spark 솔루션, AFF/FAS ONTAP Spark 솔루션 및 StorageGRID Spark 솔루션의 고유한 기능을 식별하고 자세한 검증 및 테스트를 수행했습니다. 관찰 결과에 따라, NetApp은 E-Series 솔루션을 그린필드 설치 및 새로운 확장 가능한 구축에 사용하고, 기존 NFS 데이터를 사용하는 데이터 이동 없는 분석, AI, ML, DL 워크로드, 오브젝트 스토리지가 필요할 경우 StorageGRID for AI, ML, DL 및 최신 데이터 분석을 지원하는 AFF/FAS 솔루션을 권장합니다.



데이터 레이크는 분석, AI, ML 및 DL 작업에 사용할 수 있는 기본 형식의 대규모 데이터 세트를 위한 스토리지 리포지토리입니다. E-Series, AFF/FAS, StorageGRID SG6060 Spark 솔루션을 위한 데이터 레이크 저장소를 구축했습니다. E-Series 시스템은 HDFS에 Hadoop Spark 클러스터에 액세스할 수 있는 반면, 기존 운영 데이터는 NFS 직접 액세스 프로토콜을 통해 Hadoop 클러스터에 액세스할 수 있습니다. 오브젝트 스토리지에 상주하는 데이터 세트의 경우 NetApp StorageGRID를 통해 S3 및 S3a의 안전한 액세스를 제공합니다.

사용 사례 요약

이 페이지에서는 이 솔루션을 사용할 수 있는 다양한 영역에 대해 설명합니다.

스트리밍 데이터

Apache Spark는 스트리밍 추출, 변환 및 로드(ETL) 프로세스, 데이터 보강, 이벤트 감지 트리거 및 복잡한 세션 분석에 사용되는 스트리밍 데이터를 처리할 수 있습니다.

- * Streaming ETL. * 데이터가 데이터 저장소로 푸시되기 전에 지속적으로 정리 및 집계됩니다. Netflix는 Kafka 및 Spark 스트리밍을 사용하여 다양한 데이터 소스에서 매일 수십억 개의 이벤트를 처리할 수 있는 실시간 온라인 영화 추천 및 데이터 모니터링 솔루션을 구축합니다. 그러나 일괄 처리를 위한 기존 ETL은 다르게 처리됩니다. 이 데이터를 먼저 읽은 다음 데이터베이스에 쓰기 전에 데이터베이스 형식으로 변환됩니다.
- * 데이터 보강. * Spark 스트리밍은 라이브 데이터를 정적 데이터로 보강하여 보다 실시간 데이터 분석을 가능하게 합니다. 예를 들어, 온라인 광고주는 고객 행동에 대한 정보를 바탕으로 맞춤화된 맞춤형 광고를 제공할 수 있습니다.
- * 이벤트 감지 트리거 * 스파크 스트리밍을 사용하면 잠재적으로 심각한 문제를 나타낼 수 있는 비정상적인 동작을 신속하게 감지하고 대응할 수 있습니다. 예를 들어, 금융 기관은 사기 거래를 탐지 및 중지하는 트리거를 사용하며 병원에서는 환자의 바이탈 사인에 감지된 위험한 건강 변화를 감지하기 위해 트리거를 사용합니다.
- * 복잡한 세션 분석. * Spark 스트리밍은 웹 사이트나 응용 프로그램에 로그인한 후 사용자 활동과 같은 이벤트를 수집하여 그룹화하고 분석합니다. 예를 들어 Netflix는 이 기능을 사용하여 실시간 영화 권장 사항을 제공합니다.

스트리밍 데이터 구성, Confluent Kafka 검증 및 성능 테스트에 대한 자세한 내용은 을 참조하십시오 "TR-4912: NetApp을 통해 Confluent Kafka 계층형 스토리지를 위한 모범 사례 지침".

머신 러닝

Spark 통합 프레임워크는 MLlib(Machine Learning Library)를 사용하여 데이터 세트에서 반복되는 쿼리를 실행할 수 있도록 도와줍니다. MLlib는 예측 인텔리전스, 마케팅 목적으로 고객 세분화, 감정 분석과 같은 일반적인 빅 데이터 기능의 클러스터링, 분류 및 차원 감소 등의 영역에서 사용됩니다. MLlib는 네트워크 보안에 사용되어 데이터 패킷의 실시간 검사를 수행하여 악의적인 활동의 징후를 파악합니다. 이 솔루션은 보안 제공업체가 새로운 위협에 대해 학습하고 해커보다 앞서나가는 동시에 클라이언트를 실시간으로 보호할 수 있도록 도와줍니다.

딥 러닝

TensorFlow는 업계 전반에 걸쳐 사용되는 인기 있는 딥 러닝 프레임워크입니다. TensorFlow는 CPU 또는 GPU 클러스터에 대한 분산 교육을 지원합니다. 이렇게 분산된 교육을 통해 사용자는 많은 양의 데이터에서도 딥 레이어가 많이 포함된 상태로 데이터를 실행할 수 있습니다.

얼마 전까지만 해도 TensorFlow에 Apache Spark를 사용하려면 PySpark에서 TensorFlow에 필요한 ETL을 수행한 다음 중간 스토리지에 데이터를 써야 했습니다. 그런 다음 실제 훈련 프로세스를 위해 TensorFlow 클러스터에 데이터를 로드합니다. 이 워크플로우에서는 사용자가 ETL용 클러스터와 TensorFlow의 분산 교육용으로 각각 하나씩, 두 개의 서로 다른 클러스터를 유지해야 했습니다. 여러 클러스터를 실행하고 유지하는 일은 일반적으로 지루하고 시간이 오래 걸립니다.

이전 Spark 버전의 DataFrames 및 RDD는 랜덤 액세스가 제한되었기 때문에 딥 러닝에 적합하지 않았습니다. 프로젝트 수소가 포함된 Spark 3.0에서는 딥 러닝 프레임워크에 대한 기본 지원이 추가됩니다. 이 접근 방식을 사용하면 Spark 클러스터에서 MapReduce를 기반으로 하지 않는 일정을 수행할 수 있습니다.

대화형 분석

Apache Spark는 SQL, R, Python 등 Spark 이외의 개발 언어를 사용하여 샘플링하지 않고 탐색 쿼리를 수행할 수 있을 만큼 빠릅니다. Spark는 시각화 도구를 사용하여 복잡한 데이터를 처리하고 대화형으로 시각화합니다. Spark with 구조화된 스트리밍은 웹 분석의 라이브 데이터에 대한 대화형 쿼리를 수행하여 웹 방문자의 현재 세션에 대해 대화형 쿼리를 실행할 수 있도록 합니다.

추천 시스템

기업과 소비자가 온라인 쇼핑, 온라인 엔터테인먼트 및 기타 다양한 산업의 급격한 변화에 대응함에 따라, 지난 몇 년 동안 추천 시스템은 우리의 삶에 엄청난 변화를 가져왔습니다. 실제로 이러한 시스템은 생산 과정에서 AI의 가장 확실한 성공 사례 중 하나입니다. 많은 실제 사용 사례에서 추천 시스템은 NLP 백엔드와 상호 작용 AI 또는 챗봇과 결합되어 관련 정보를 얻고 유용한 추론을 생성합니다.

오늘날, 많은 소매업체들은 온라인 구매 및 매장에서 픽업, 큐브사이드 픽업, 셀프 체크아웃, 스캔 및 이동 등 새로운 비즈니스 모델을 채택하고 있습니다. COVID-19가 범세계적으로 확산되고 있는 가운데, 이러한 모델은 더욱 안전하고 편리한 쇼핑을 통해 뚜렷하게 부각되고 있습니다. AI는 소비자 행동의 영향을 받는 디지털 트렌드에 있어 매우 중요합니다. 고객의 증가하는 요구사항을 충족하고, 고객 경험을 보강하고, 운영 효율성을 개선하고, 수익을 증대하기 위해 NetApp은 엔터프라이즈 고객과 기업이 머신 러닝 및 딥 러닝 알고리즘을 사용하여 더 빠르고 정확한 추천 시스템을 설계할 수 있도록 지원합니다.

협업 필터링, 콘텐츠 기반 시스템, 딥 러닝 추천 모델(DLRM), 하이브리드 기술 등 권장 사항을 제공하는 데 사용되는 몇 가지 기술이 있습니다. 고객은 이전에 PySpark를 사용하여 권장 시스템 생성을 위한 협업 필터링을 구현했습니다. Spark MLlib는 DLRM이 등장하기 전에 엔터프라이즈 사이에서 매우 널리 사용되는 알고리즘인 협업 필터링을 위해 교류 최소 사각형(ALS)을 구현합니다.

자연어 처리

자연어 처리(NLP)로 가능해진 대화형 AI는 컴퓨터가 인간과 통신할 수 있도록 지원하는 AI의 지점입니다. NLP는 스마트 비서 및 챗봇에서 Google 검색 및 예측 텍스트에 이르기까지 모든 업계 수직 및 다양한 사용 사례에서 널리 사용되고 있습니다. 에 따르면 "Gartner" 2022년까지 70%의 사용자가 매일 대화형 AI 플랫폼과 상호 작용할 것으로 예측 인간과 기계 사이의 고품질 대화를 위해서는 신속하고 지능적이며 자연스러운 대화가 이루어져야 합니다.

고객은 NLP 및 ASR(자동 음성 인식) 모델을 처리하고 교육하기 위해 대량의 데이터가 필요합니다. 또한 에지, 코어, 클라우드 전반에서 데이터를 이동해야 하며, 인류와 자연적 통신을 위해 수 밀리초 내에 추론을 수행할 수 있는 기능이 필요합니다. NetApp AI 및 Apache Spark는 컴퓨팅, 스토리지, 데이터 처리, 모델 교육, 미세 조정, 있습니다.

정서 분석은 NLP에서 텍스트에서 긍정적, 부정적 또는 중립적 감정을 추출하는 연구 분야입니다. 고객 의견 분석에는 지원 센터 직원의 성과 파악부터 적절한 자동 챗봇 응답 제공에 이르기까지 다양한 활용 사례가 있습니다. 또한 분기별 수익 통화 시 기업 담당자와 대상 간의 상호 작용을 기반으로 회사의 주가를 예측하기도 했습니다. 또한, 감정 분석을 사용하여 브랜드가 제공하는 제품, 서비스 또는 지원에 대한 고객의 관점을 결정할 수 있습니다.

을 사용했습니다 "스파크 NLP" 라이브러리 시작 "John Snow Labs를 참조하십시오" 를 포함하여 Transformers(BERT) 모델의 사전 교육 파이프라인 및 양방향 인코더 표현을 로드합니다 "금융 뉴스 정서" 및 "핀베르트"즉, 토큰화, 명명된 엔터티 인식, 모델 교육, 피팅 및 정서 분석을 대규모로 수행합니다. Spark NLP는 BERT, Albert, Electra, XLNet, DistillBERT 등의 최첨단 변압기(transformer)를 제공하는 유일한 오픈 소스 NLP 라이브러리입니다. Roberta, DeBERTa, XLM-Roberta, Longrofer, Elmo, Universal 문장 인코더, Google T5, MarianMT 및 GPT2. 이 라이브러리는 Python 및 R 뿐만 아니라 Apache Spark를 기본적으로 확장하여 JVM 에코시스템(Java, Scala, Kotlin)에서도 사용할 수 있습니다.

주요 **AI, ML** 및 **DL** 사용 사례 및 아키텍처

주요 AI, ML 및 DL 사용 사례 및 방법론을 다음 절로 나눌 수 있습니다.

NLP 파이프라인 및 **TensorFlow** 분산 추론을 스파크합니다

다음 목록에는 데이터 과학 커뮤니티가 다양한 개발 수준에서 채택한 가장 널리 사용되는 오픈 소스 NLP 라이브러리가 포함되어 있습니다.

- "자연어 도구 키트(NLTK)". 모든 NLP 기술에 대한 완벽한 도구 키트. 2000년대 초반부터 유지되었습니다.
- "텍스트 부분". NLTK 및 패턴 위에 구축된 사용하기 쉬운 NLP 도구 Python API.
- "스탠포드 코어 NLP". 스탠포드 NLP Group이 개발한 Java의 NLP 서비스 및 패키지.
- "젠심". 인간을 위한 주제 모델링은 체코 디지털 수학 라이브러리 프로젝트를 위한 Python 스크립트의 모음으로 시작되었습니다.
- "스파이". 트랜스포머용 GPU 가속 기능을 갖춘 Python 및 Cython을 사용한 엔드 투 엔드 산업용 NLP 워크플로.
- "빠른 텍스트". Facebook의 AI 연구(Fair) 연구소에서 만든 단어 인식 및 문장 분류용 무료 경량 오픈 소스 NLP 라이브러리.

SPARK NLP는 모든 NLP 작업 및 요구 사항을 충족하는 단일 통합 솔루션으로, 실제 생산 사용 사례에 맞게 확장 가능하고 성능이 뛰어나며 정확도가 높은 NLP 기반 소프트웨어를 지원합니다. 이 제품은 전송 학습을 활용하고 연구 및 산업 전반에 걸쳐 최신 최신 알고리즘 및 모델을 구현합니다. Spark는 위의 라이브러리에 대한 완벽한 지원이 부족하기 때문에 Spark NLP를 기반으로 구축되었습니다 "스파크 ML" Spark의 범용 인메모리 분산 데이터 처리 엔진을 미션 크리티컬 프로덕션 워크플로우를 위한 엔터프라이즈급 NLP 라이브러리로 활용하십시오. 주석 달기 는 규칙 기반 알고리즘, 머신 러닝 및 TensorFlow를 활용하여 딥 러닝 구현을 지원합니다. 여기에는 토큰화, 레몬화, 스테밍, 부분 음성 태깅, 명명된 엔터티 인식 등을 포함한 일반적인 NLP 작업이 포함됩니다. 맞춤법 검사 및 정서 분석.

Transformers(BERT)의 양방향 인코더 표현은 NLP를 위한 변압기 기반 기계 학습 기술입니다. 사전 교육과 미세 조정 개념을 대중화했습니다. BERT의 변압기 아키텍처는 재발성 신경망(RNN) 기반 언어 모델보다 장기적인 종속성을 더 잘 모델링하는 기계 번역에서 비롯되었습니다. 또한 MLM(Masked Language Modeling) 작업도 도입되었으며, 이 작업에서는 모든 토큰의 임의 15%가 마스킹되고 모델이 이를 예측하여 진정한 방향성을 가능하게 합니다.

전문 언어와 해당 영역에 레이블이 지정된 데이터가 없기 때문에 재무 감정의 분석은 어렵습니다. "핀베르트"사전 훈련된 BERT에 기반한 언어 모델은 도메인 적용에 적합합니다 "로이터 TRC2", 재정 문제, 라벨이 붙은 데이터("금융 서비스 은행")를 참조하십시오. 연구원들은 재무 용어로 뉴스 기사에서 4개, 500개 문장을 추출했습니다. 그때 16명의 전문가들과 재정 배경을 가진 대학생들은 그 문장을 긍정적이고 중립적이며 부정적으로 표시했다. 2016년부터 2020년까지 FinBERT와 사전 교육 받은 두 개의 다른 파이프라인을 사용하여 상위 10개 NASDAQ 회사 수익 통화 녹취록의 정서를 분석하기 위해 Spark 워크플로를 구축했습니다 "금융 뉴스를 위한 감정 분석", "문서 DL을 설명합니다 ") Spark NLP에서.

Spark NLP를 위한 기본 딥 러닝 엔진은 손쉬운 모델 구축, 어디서나 강력한 ML 생산, 연구를 위한 강력한 실험을 지원하는 기계 학습용 엔드 투 엔드 오픈 소스 플랫폼인 TensorFlow입니다. 따라서 Spark 'YARN 클러스터' 모드에서 파이프라인을 실행할 때 기본적으로 하나의 마스터 노드와 여러 작업자 노드 및 클러스터에 마운트된 네트워크 연결 스토리지에서 데이터와 모델 병렬화를 통해 분산된 TensorFlow를 실행하고 있었습니다.

Horovod의 분산 훈련

MapReduce 관련 성능을 위한 핵심 Hadoop 검증은 TeraGen, TeraSort, TeraValidate 및 DFSIO(읽기 및 쓰기)를 통해 수행됩니다. TeraGen 및 TeraSort 검증 결과는 에 나와 있습니다 "TR-3969: Hadoop용 NetApp 솔루션" E-Series의 경우 및 AFF의 경우 "스토리지 계층화"(xref) 섹션

고객의 요청에 따라 Spark와 함께 분산된 교육을 다양한 사용 사례 중 가장 중요한 것으로 간주합니다. 이 문서에서는 을 사용했습니다 "Spark의 Hoborod" AFF(All Flash FAS) 스토리지 컨트롤러, Azure NetApp Files 및 StorageGRID를 사용하여 NetApp 사내, 클라우드 네이티브 및 하이브리드 클라우드 솔루션을 통해 Spark 성능을 검증합니다.

Horovod on Spark 패키지는 Spark 클러스터에서 분산된 훈련 워크로드를 간단하게 실행할 수 있도록 Horovod를 둘러싸는 편리한 래퍼를 제공합니다. 따라서 훈련 및 추론 데이터가 상주하는 Spark에서 데이터 처리, 모델 훈련 및 모델 평가를 모두 수행하는 엄격한 모델 설계 루프를 사용할 수 있습니다.

Spark에서 Horovod를 실행하기 위한 두 가지 API(고급 추정기 API 및 하위 수준의 Run API)가 있습니다. 둘 다 동일한 기본 메커니즘을 사용하여 Spark 실행기에서 Horovod를 실행하지만 Estimator API는 데이터 처리, 모델 훈련 루프, 모델 체크포인트, 메트릭 수집 및 분산 교육을 추상화합니다. Horovod Spark Estimators, TensorFlow 및 Keras를 사용하여 에 기반한 엔드 투 엔드 데이터 준비 및 분산 교육 워크플로를 사용했습니다 "Kaggle Rossmann 매장 판매" 경쟁업체.

'keras_spark_horovod_rossmann_estimator.py' 스크립트는 섹션에서 찾을 수 있습니다 "주요 활용 사례별로 Python 스크립트" 여기에는 다음 세 가지 부분이 포함됩니다.

- 첫 번째 부분은 Kaggle이 제공하고 커뮤니티에서 수집한 초기 CSV 파일 세트에 대해 다양한 데이터 전처리 단계를 수행합니다. 입력 데이터는 '검증' 하위 세트와 테스트 데이터 세트가 있는 교육 세트로 구분됩니다.
- 두 번째 부분은 로그 시그마리드 활성화 기능과 ADAM 옵티마이저가 있는 Keras Deep Neural Network(DNN) 모델을 정의하고 Spark On을 사용하여 모델 분산 훈련을 수행합니다.
- 세 번째 부분은 검증 세트의 전체 평균 절대 오류를 최소화하는 최상의 모델을 사용하여 테스트 데이터 세트에 대한 예측을 수행합니다. 그런 다음 출력 CSV 파일을 만듭니다.

섹션을 참조하십시오 ""머신 러닝"" 다양한 런타임 비교 결과.

CTR 예측에 Keras를 사용한 다중 작업자 딥 러닝

최근 ML 플랫폼 및 애플리케이션의 발전으로 이제 대규모 학습에 많은 관심이 집중되고 있습니다. 클릭 비율(CTR)은 온라인 광고 노출 100회 당 평균 클릭 수(백분율로 표시)로 정의됩니다. 디지털 마케팅, 소매, 전자 상거래 및 서비스 공급자를 포함한 다양한 산업 및 사용 사례에서 핵심 메트릭으로 널리 채택되고 있습니다. 자세한 내용은 를 참조하십시오 "TR-4904: Azure에서 제공되는 분산 교육 - 클릭 비율 예측" CTR 및 Kubernetes를 통한 엔드 투 엔드 클라우드 AI 워크플로우 구현, 분산된 데이터 ETL, Dask 및 CUDA ML을 사용한 모델 교육에 대한 자세한 내용을 확인하십시오.

이 기술 보고서에서는 의 변형을 사용했습니다 "Criteo Terabyte Click Logs 데이터 세트" (TR-4904 참조) Keras를 사용하여 여러 작업자가 딥 및 크로스 네트워크(DCN) 모델로 Spark 워크플로를 구축하여 로그 손실 오류 기능과 기준 Spark ML 회귀 모델의 성능을 비교한 딥 러닝(Deep and Cross Network) 모델에 대해 알아보십시오. DCN은 경계가 지정된 수준의 효과적인 기능 상호 작용을 효율적으로 캡처하고, 고도의 비선형 상호 작용을 학습하며, 수동 기능 엔지니어링 또는 철저한 검색이 필요하지 않으며, 계산 비용이 낮습니다.

웹 스케일 추천 시스템을 위한 데이터는 대부분 불연속적이고 범주적이며, 이는 기능 탐색이 어려운 크고 부족한 기능 공간으로 이어집니다. 이는 대부분의 대규모 시스템을 물류 회귀와 같은 선형 모델로 제한합니다. 그러나 자주 예측 가능한 기능을 식별하고 보이지 않거나 희귀한 크로스 기능을 탐색하는 것은 좋은 예측을 위한 열쇠입니다. 선형 모델은 단순하고, 해석 가능하며, 확장이 쉽지만 표현 능력이 제한적입니다.

반면 크로스 기능은 모델의 표현 능력을 향상시키는 데 중요한 것으로 나타났습니다. 하지만 이러한 기능을 식별하려면 수동 기능 엔지니어링이나 철저한 검색이 필요한 경우가 많습니다. 보이지 않는 피처 상호작용에 일반화하기는 어려운 경우가 많습니다. DCN과 같은 교차 신경망(cross neural network)을 사용하면 자동 방식으로 기능적으로 명시적으로 교차하여 작업별 기능별 엔지니어링을 피할 수 있습니다. 크로스 네트워크는 다중 계층으로 구성되어 있으며, 이 경우 가장 높은 수준의 상호 작용이 레이어 깊이에 따라 결정될 수 있습니다. 각 계층은 기존 레이어를 기반으로 보다 높은 수준의 상호 작용을 생성하고 이전 레이어의 상호 작용을 유지합니다.

DNN(Deep Neural Network)은 여러 기능에서 매우 복잡한 상호 작용을 포착할 수 있는 가능성을 가지고 있습니다. 그러나 DCN과 비교할 때 거의 많은 매개변수가 필요하며, 상호 기능을 명시적으로 구성할 수 없으며, 일부 유형의 기능 상호 작용을 효율적으로 학습하지 못할 수 있습니다. 크로스 네트워크는 메모리 효율적이고 구현하기 쉽습니다. 상호 및 DNN 구성 요소를 공동으로 교육하여 예측 가능한 기능 상호 작용을 효율적으로 캡처하고 Critio CTR 데이터 세트에 대한 최첨단 성능을 제공합니다.

DCN 모델은 포매 및 스태킹 계층, 크로스 네트워크 및 딥 네트워크를 병렬로 사용하여 시작합니다. 두 네트워크의 출력을 결합하는 최종 조합 계층이 이어집니다. 입력 데이터는 희소 및 고밀도 피처를 가진 벡터가 될 수 있습니다. Spark에서는 두 가지 모두 가능합니다 "ML" 및 "mllib를 참조하십시오" 라이브러리에는 'parseVector' 형식이 포함됩니다. 따라서 사용자는 두 가지 기능을 구분할 수 있어야 하며 각 함수와 메서드를 호출할 때 유의해야 합니다. CTR 예측과 같은 웹 스케일 추천 시스템에서 입력은 주로 범주적인 기능(예: ``country=USA')입니다. 이러한 기능은 `[0,1,0,…]''와 같은 하나의 인기 벡터로 인코딩되는 경우가 많습니다. 'ParseVector'를 사용한 단일 핫 인코딩(OHE)은 끊임없이 변화하고 성장하는 어휘를 사용하여 실제 데이터세트를 처리할 때 유용합니다. 에서 예제를 수정했습니다 "DeepCTR" 대용량 어휘를 처리하기 위해 DCN의 포매 및 스태킹 계층에 포매 벡터를 만듭니다.

를 클릭합니다 "Criteo Display Ads 데이터 세트" 광고 클릭률을 예측합니다. 13개의 정수 기능과 각 범주에 높은 카디널리티가 있는 26개의 범주 기능이 있습니다. 이 데이터 세트의 경우 입력 크기가 커서 로그손실이 0.001로 향상되는 것이 실질적으로 중요합니다. 대규모 사용자 기반의 예측 정확도가 약간 개선되는 경우 회사의 수익이 크게 증가할 수 있습니다. 데이터 세트에는 7일 동안 11GB의 사용자 로그가 포함되어 있으며, 이는 약 4100만 개의 레코드에 해당합니다. Spark dataFrame.RandomSplit() 기능을 사용하여 교육(80%), 교차 검증(10%) 및 나머지 10%의 테스트 데이터를 무작위로 분할했습니다.

DCN은 Keras를 통해 TensorFlow에 구현되었습니다. DCN을 통해 모델 교육 프로세스를 구현하는 주요 구성 요소는 다음과 같습니다.

- * 데이터 처리 및 포함. * 실제 가치가 있는 기능은 로그 변환을 적용하여 정규화됩니다. 카테고리 피처의 경우, 치수 6 × (카테고리 카디널리티) 1/4의 고밀도 벡터에 피처를 포함시킵니다. 모든 임베디드 디딩을 연결하면 치수 1026의

벡터가 됩니다.

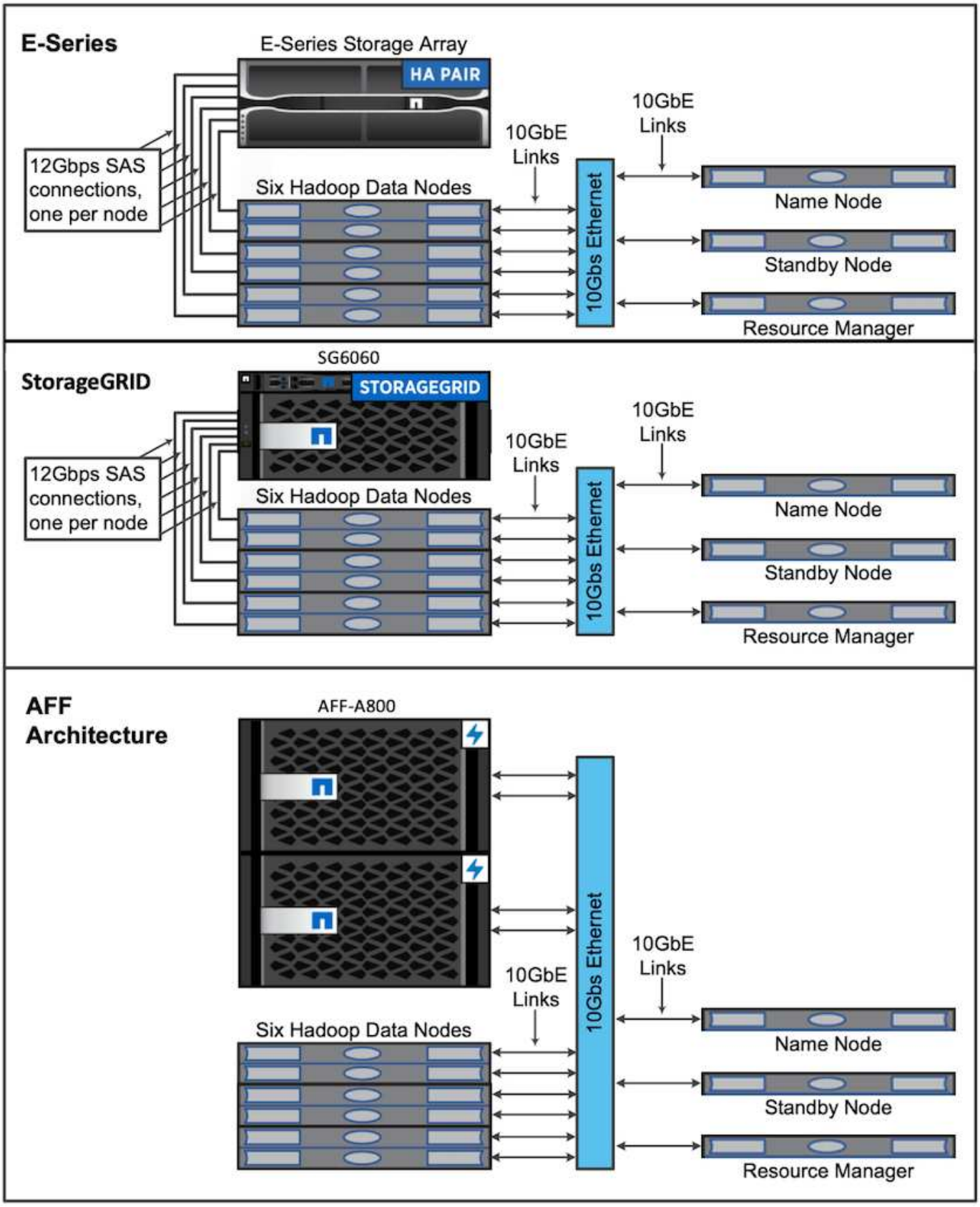
- 최적화. * ADAM 옵티마이저를 사용하여 미니 배치 확률적 최적화를 적용했습니다. 배치 크기가 512로 설정되었습니다. 배치 정규화가 딥 네트워크에 적용되고 그레이디언트 클립 표준은 100으로 설정되었습니다.
- "규칙화." 우리는 L2 규칙화 또는 중도탈락이 효과가 없는 것으로 확인됨에 따라 조기 정지에 사용했습니다.
- * Hyperparameters. * 숨겨진 계층 수, 숨겨진 계층 크기, 초기 학습 속도 및 교차 계층 수에 대한 그리드 검색을 기반으로 결과를 보고합니다. 숨겨진 레이어의 수는 2-5개까지이며 숨겨진 레이어 크기는 32-1024입니다. DCN의 경우 교차 계층 수는 1에서 6까지 있었습니다. 초기 학습 속도는 0.0001에서 0.0001로 조정되었으며 0.0001씩 증가하였습니다. 모든 실험은 훈련 단계 150,000에 일찍 멈추어 적용되었고, 그 이후에는 과다 피팅이 발생하기 시작했습니다.

DCN 외에도 CTR 예측을 위해 기타 인기 있는 딥 러닝 모델도 테스트했습니다(예 "DeepFM", "xDeepFM", "자동 내부", 및 "DCN v2".

검증에 사용된 아키텍처

이 검증을 위해 4개의 작업자 노드와 AFF-A800 HA 쌍의 1개의 마스터 노드를 사용했습니다. 모든 클러스터 구성원이 10GbE 네트워크 스위치를 통해 연결되었습니다.

이러한 NetApp Spark 솔루션 검증을 위해 E5760, E5724, AFF-A800의 3가지 스토리지 컨트롤러를 사용했습니다. E-Series 스토리지 컨트롤러는 12Gbps SAS로 연결된 5개의 데이터 노드에 연결되었습니다. AFF HA 쌍 스토리지 컨트롤러는 10GbE 연결을 통해 Hadoop 작업자 노드에 내보낸 NFS 볼륨을 제공합니다. Hadoop 클러스터 멤버는 E-Series, AFF 및 StorageGRID 하둡 솔루션에서 10GbE 연결을 통해 연결했습니다.



테스트 결과

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

AFF-A800 구성을 사용하여 Spark 성능 검증을 측정했습니다. 또한, Spark NLP 파이프라인 및 TensorFlow 배포 교육, Horovod 분산 교육, DeepFM을 통한 CTR 예측에 Keras를 사용한 다중 작업자 딥 러닝 등 3가지 주요 사용 사례를 테스트했습니다.

E-Series와 StorageGRID 검증 모두에 Hadoop 복제 계수 2를 사용했습니다. AFF 검증에서는 하나의 데이터 소스만 사용했습니다.

다음 표에는 Spark 성능 검증을 위한 하드웨어 구성이 나와 있습니다.

유형	**Hadoop** 작업자 노드	드라이브 유형입니다	노드당 드라이브 수	스토리지 컨트롤러
SG6060	4	SAS를 참조하십시오	12	단일 고가용성(HA) 쌍
E5760에서	4	SAS를 참조하십시오	60	단일 HA 쌍
E5724)를 참조하십시오	4	SAS를 참조하십시오	24	단일 HA 쌍
AFF800	4	SSD를 지원합니다	6	단일 HA 쌍

다음 표에는 소프트웨어 요구 사항이 나와 있습니다.

소프트웨어	버전
RHEL을 참조하십시오	7.9
OpenJDK 런타임 환경	1.8.0
OpenJDK 64비트 서버 VM	25.302
기트	2.24.1
GCC/G++	11.2.1
스파크	3.2.1
피스파크	3.1.2
스파르크LP	3.4.2
TensorFlow를 참조하십시오	2.9.0
Keras	2.9.0
호로브	0.24.3

재무 정서 분석

발표했습니다 "TR-4910: NetApp AI를 통한 고객 커뮤니케이션의 감정 분석"즉, 포괄적인 대화형 AI 파이프라인은 를 사용하여 구축되었습니다 "NetApp DataOps 툴킷", AFF 스토리지 및 NVIDIA DGX 시스템. 파이프라인은 DataOps 툴킷을 활용하여 배치 오디오 신호 처리, 자동 음성 인식(ASR), 전송 학습 및 정서 분석을 수행합니다. "NVIDIA Riva SDK를 참조하십시오", 및 "Tao 프레임워크". 정서 분석 활용 사례를 금융 서비스 산업으로 확대하면서 스파르크LP 워크플로를 구축하고, 명명된 엔터티 인식 등의 다양한 NLP 작업을 위한 BERT 모델을 3개 로드했으며, NASDAQ 상위 10개 기업의 분기별 수익 통화에 대해 문장 수준의 감정을 얻었습니다.

다음 스크립트의 entiment_analysis_spark. py는 다음 표와 같이 FinBERT 모델을 사용하여 HDFS에서 전사체를 처리하고 양의, 중성적이고, 부정적인 정서 수를 산출합니다.

```
-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 표에는 2016년부터 2020년까지 NASDAQ 상위 10대 기업에 대한 순익금 및 문장 수준의 정서 분석이 나와 있습니다.

정서 수와 백분율	**10**개 회사 모두	**AAPL**	**AMD**	**AMZN**	**CSCO**	**GOOGL**	아이**NTC**	**MSFT**	**NVDA**
양수입니다	7447	1567	743	290	682를 참조하십시오	826	824	904	417
중립 카운트	64067	6856)을 참조하십시오	7596	5086	6650	5914	6099	5715)를 참조하십시오	6189
음의 카운트	1787	253	213	84	189	97	282	202	89
분류되지 않은 카운트	196	0	0	76	0	0	0	1	0
(총 카운트)	73497	8676)을 참조하십시오	8552	5536	7521	6837	7205	6822	6695)를 참조하십시오

백분율의 경우 CEO와 CFO가 말하는 대부분의 문장이 사실에 근거해 중립적인 감정을 가지고 있습니다. 수익 통화 중에 분석가들은 긍정적이거나 부정적인 감정을 전달할 수 있는 질문을 합니다. 이에 따라 내당일 또는 다음 날 주가가 음성이나 양성으로 어떻게 영향을 미치는지를 정량적으로 조사할 필요가 있다.

다음 표에는 NASDAQ 상위 10대 기업에 대한 문장 수준의 감정 분석이 백분율로 나와 있습니다.

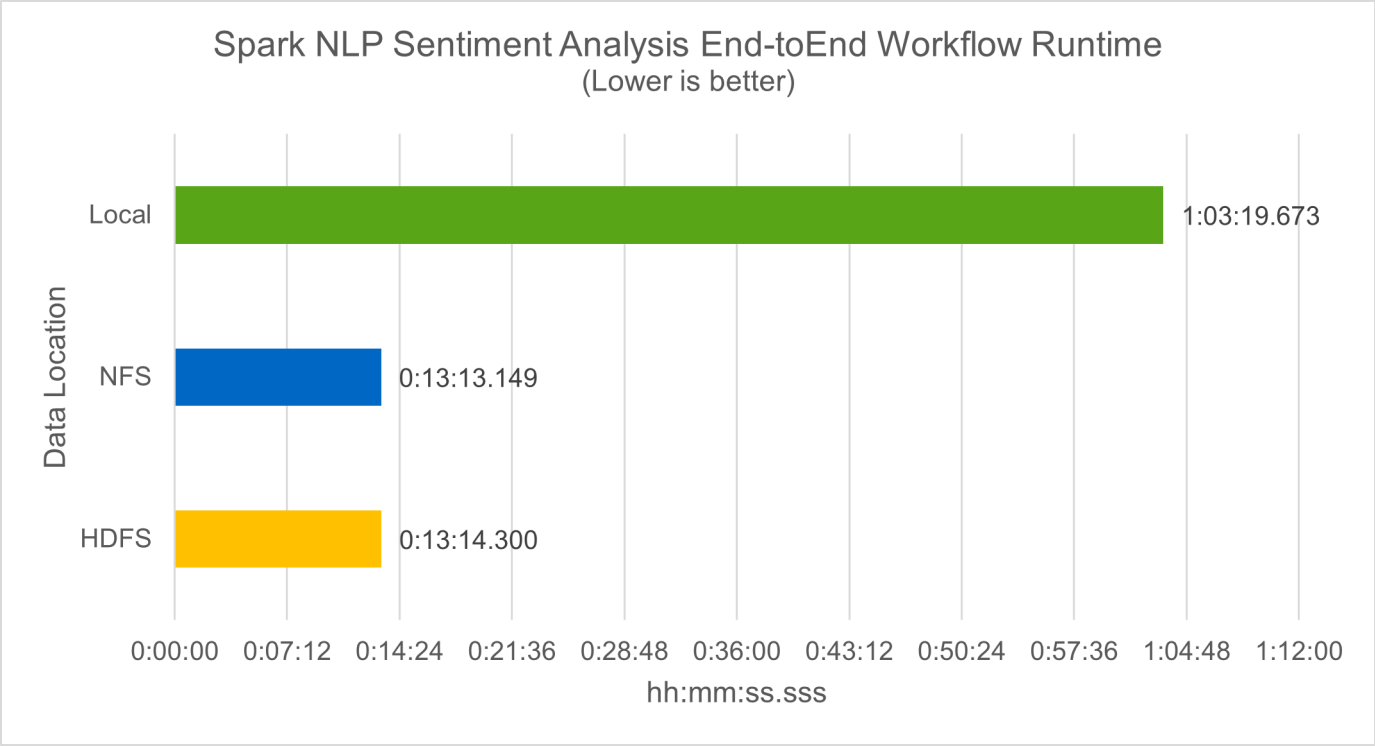
정서 백분율	**10**개 회사 모두	**AAPL**	**AMD**	**AMZN**	**CSCO**	**GOOGL**	아이**NTC**	**MSFT**	**NVDA**
긍정적	10.13%	18.06%	8.69%	5.24%	9.07%	12.08%	11.44%	13.25%	6.23%
중립	87.17%	79.02%	88.82%	91.87%	88.42%	86.50%	84.65%	83.77%	92.44%

정서 백분율	**10**개 회사 모두	**AAPL**	**AMD**	**AMZN**	**CSCO**	**GOOGL**	아이**NTC**	**MSFT**	**NVDA**
네거티브	2.43%	2.92%	2.49%	1.52%	2.51%	1.42%	3.91%	2.96%	1.33%
분류되지 않았습니다	0.27%	0%	0%	1.37%	0%	0%	0%	0.01%	0%

워크플로우 런타임 측면에서 HDFS의 로컬 모드에서 분산 환경으로 4.78배, NFS를 활용하여 0.14% 향상되었습니다.

```
-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 처리를 통해 데이터 처리 및 분산 TensorFlow 모델 추론 속도가 향상되었습니다. NFS의 데이터 위치는 워크플로우 병목 현상이 사전 교육 모델을 다운로드하는 것이기 때문에 런타임 성능이 약간 향상되었습니다. 전사체 데이터 세트 크기를 늘릴 경우 NFS의 장점은 더욱 명확합니다.



Horovod 성과를 통한 분산 훈련

다음 명령을 실행하면 코어 1개가 포함된 160개의 실행자가 있는 단일 마스터 노드를 사용하여 Spark 클러스터에서 런타임 정보와 로그 파일이 생성되었습니다. 메모리 부족 오류가 발생하지 않도록 executor 메모리가 5GB로 제한되었습니다. 섹션을 참조하십시오 ""주요 활용 사례별로 Python 스크립트"" 'keras_spark_horovod_rossmann_estimator.py'의 데이터 처리, 모델 훈련 및 모델 정확도 계산에 대한 자세한 내용

```
(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
```

10번의 교육 Epoch를 사용한 결과 런타임은 다음과 같습니다.

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

입력 데이터를 처리하고, DNN 모델을 교육하고, 정확도를 계산하고, 예측 결과를 위한 TensorFlow 체크포인트와 CSV 파일을 생성하는 데 43분 이상이 걸렸습니다. 교육 Epoch의 수를 10개로 제한했습니다. 실제로 만족스러운 모델 정확도를 보장하기 위해 대개 100으로 설정되어 있습니다. 일반적으로 교육 시간은 Epoch 수에 비례하여 확장됩니다.

다음으로 클러스터에서 사용할 수 있는 4개의 작업자 노드를 사용하고 HDFS에서 데이터와 함께 'YARN' 모드로 동일한 스크립트를 실행했습니다.

```
(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
```

결과 런타임은 다음과 같이 개선되었습니다.

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

Spark의 Horovod 모델과 데이터 병렬화를 통해 10번의 교육 Epoch로 "원사"와 "로컬" 모드의 실행 속도가 5.29배 빨라졌습니다. 이 그림은 다음 그림과 같이 전설적인 HDFS와 Local로 표시됩니다. 기본 TensorFlow DNN 모델 교육은 GPU를 사용하여 더 가속화될 수 있습니다. NetApp은 이 테스트를 수행하고 결과를 향후 기술 보고서에 게시할 계획입니다.

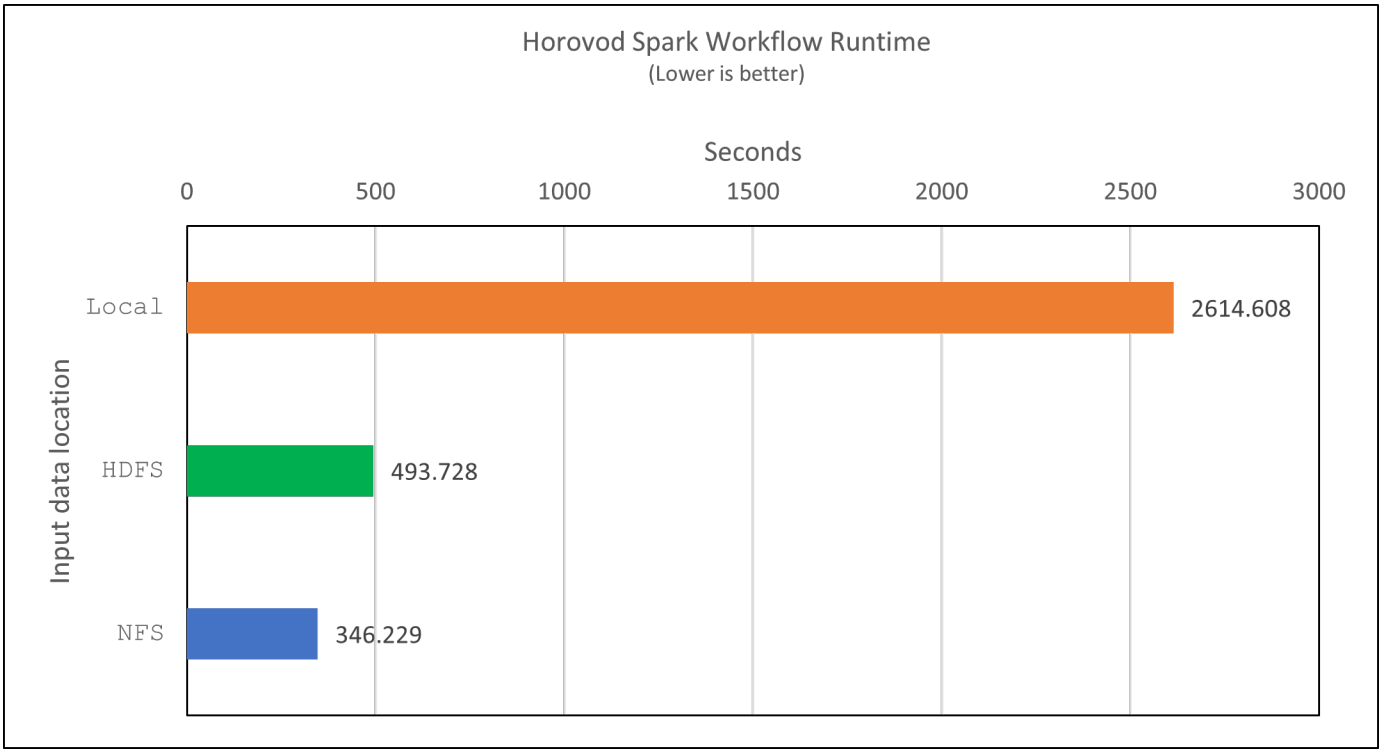
다음 테스트에서는 NFS에 상주하는 입력 데이터와 HDFS를 비교하여 실행 시간을 비교했습니다. AFF A800의 NFS 볼륨은 Spark 클러스터의 5개 노드(마스터 1개, 작업자 4명)에 걸쳐 '/Spkdemo/horovod'에 마운트되었습니다. NFS 마운트를 가리키는 '--data-dir' 매개 변수를 사용하여 이전 테스트와 비슷한 명령을 실행했습니다.

```
(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
```

NFS의 결과 런타임은 다음과 같습니다.

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

다음 그림과 같이 속도가 1.43배 더 향상되었습니다. 따라서 고객은 클러스터에 연결된 NetApp All-Flash 스토리지를 통해 Horovod Spark 워크플로우에서 빠른 데이터 전송 및 배포의 이점을 누리고 단일 노드에서 실행되는 것에 비해 7.55배 더 빠른 속도를 달성할 수 있습니다.



CTR 예측 성능을 위한 딥 러닝 모델

CTR을 최대화하도록 설계된 추천 시스템의 경우 낮은 순서에서 높은 순서로 수학적으로 계산할 수 있는 사용자 행동 뒤에 정교한 기능 상호 작용을 학습해야 합니다. 낮은 순서의 기능과 높은 순서의 기능 상호 작용은 둘 중 하나를 편향하지 않고 우수한 딥 러닝 모델에 똑같이 중요합니다. 인수 기계 기반 신경 네트워크인 DeepFM(Deep Factorization Machine)은 새로운 신경망 아키텍처에서 기능 학습을 위한 권장 사항과 딥 러닝을 위한 인수 기계(factorization Machine)를 결합합니다.

기존의 공장 인수 기계는 쌍 단위 기능 상호 작용을 기능 간의 잠재적 벡터의 내부 제품으로 모델링하고 이론적으로 높은 순서 정보를 캡처할 수 있지만, 실제로 머신 러닝 실무자는 높은 계산 및 스토리지 복잡성으로 인해 일반적으로 2차 기능 상호 작용만 사용합니다. Google과 같은 딥 뉴럴 네트워크 변형 "와이드 앰프, 딥 모델" 반면, 선형 와이드 모델과 딥 모델을 결합하여 하이브리드 네트워크 구조에서 정교한 기능 상호 작용을 학습합니다.

이 Wide & Deep Model에는 기본 와이드 모델과 딥 모델에 대한 두 가지 입력이 있으며, 그 중 후자는 여전히 전문적인 피처 엔지니어링을 필요로 하므로 다른 영역에 대해 일반화할 수 없는 기술을 렌더링합니다. 광각 및 딥 모델과 달리, DeepFM은 넓은 부분과 깊은 부분이 동일한 입력 및 포함 벡터를 공유하기 때문에 기능 엔지니어링 없이 RAW 기능으로 효율적으로 교육을 받을 수 있습니다.

먼저 Criteo '기차.txt'(11GB) 파일을 NFS 마운트에 저장된 CTR_트레인.csv라는 CSV 파일로 처리했습니다.

/spclassification_criteo_spark.py라는 섹션을 사용하여 /spkdemo/TR-4570-data로 처리했습니다 ""각 주요 활용 사례에 대한 Python 스크립트"" 이 스크립트에서 함수 "process_input_file"은 여러 문자열 메소드를 수행하여 탭을 제거하고 구분 기호로 ",", 줄 바꿈으로 "\n""를 삽입합니다. 코드 블록이 주석으로 표시되도록 원래 기차 .txt만 처리하면 됩니다.

다음 DL 모델 테스트를 위해 입력 파일로 CTR_트레인.csv를 사용했습니다. 후속 테스트 실행에서 입력 CSV 파일은 ""레이블", 정수 밀도 기능 "['I1', 'I2', 'i3',…, 'I13']" 필드가 포함된 스키마가 있는 Spark DataFrame으로 읽혀졌습니다. 그리고 스파스 피처 "['C1','C2','C3',…,'C26']". 다음 'park-submit' 명령은 입력 CSV에서 수행하고 교차 검증을 위해 20% 분할로 DeepFM 모델을 교육하고 10번의 교육 Epoch 후에 최적의 모델을 선택하여 테스트 세트의 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 'CTR_트레인.csv'가 11GB를 초과하므로 오류를 방지하려면 데이터 세트 크기보다 충분한 spark.driver.maxResultSize를 설정해야 합니다.

```
 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()
```

위 'parkSession.builder' 설정에서도 활성화했다 "Apache 화살표"이는 Df. toPandas() 메소드를 사용하여 Spark DataFrame을 Pandas DataFrame으로 변환합니다.

```
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, 교육 데이터 세트에 36M 행이 넘고 테스트 세트에 9M 샘플이 있습니다.

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

이 기술 보고서는 GPU를 사용하지 않고 CPU 테스트에 집중되므로 적절한 컴파일러 플래그를 사용하여 TensorFlow를 구축하는 것이 중요합니다. 이 단계는 GPU 가속 라이브러리를 호출하지 않고 TensorFlow의 AVX(Advanced Vector Extensions) 및 AVX2 명령을 최대한 활용합니다. 이러한 기능은 벡터화된 추가, 피드 포워드 내부의 행렬 다중화 또는 역전파 DNN 교육과 같은 선형 대수 계산에 맞게 설계되었습니다. 256비트 부동 소수점(FP) 레지스터를 사용하는 AVX2에서 사용할 수 있는 FMA(Fused Multiply Add) 명령은 정수 코드 및 데이터 형식에 적합하며 최대 2배의 속도를 제공합니다. FP 코드 및 데이터 유형의 경우 AVX2는 AVX에 비해 8% 빠른 속도를 제공합니다.

```
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 사용을 권장합니다 "Bazel". 우리는 현재 환경에 대해 셸 프롬프트에서 다음 명령을 실행하여 df, df-plugins, Bazel을 설치합니다.

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

빌드 프로세스 중에 C++ 17 기능을 사용하려면 GCC 5 이상을 활성화해야 합니다. 이 기능은 RHEL에서 SCL(Software Collections Library)과 함께 제공합니다. 다음 명령은 RHEL 7.9 클러스터에 devt툴셋 및 GCC 11.2.1을 설치합니다.

```
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 /opt/rh/dev툴셋 -11/root/usr/bin/gcc(GCC 11.2.1)를 사용하는 dev툴셋 -11을 활성화합니다. 또한 git 버전이 1.8.3 이상인지 확인하십시오(RHEL 7.9와 함께 제공됨). 이를 참조하십시오 "기사" git를 2.24.1로 업데이트하는 경우.

최신 TensorFlow 마스터 저장소 를 이미 복제했다고 가정합니다. 그런 다음 "작업 공간" 파일을 사용하여 "작업 공간" 디렉토리를 만들어 AVX, AVX2 및 FMA를 사용하여 소스에서 TensorFlow를 구축합니다. '설정' 파일을 실행하고 올바른 Python 바이너리 위치를 지정합니다. "CUDA" GPU를 사용하지 않았기 때문에 테스트에 사용할 수 없습니다. 설정에 따라 '.bazelrc' 파일이 생성됩니다. 또한 파일을 편집하고 "build—define=no_hdfs_support=false"를 설정하여

HDFS 지원을 활성화했습니다. 절의 '.bazelrc'를 참조하십시오 ""주요 활용 사례별로 Python 스크립트,"" 설정 및 플래그의 전체 목록을 표시합니다.

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

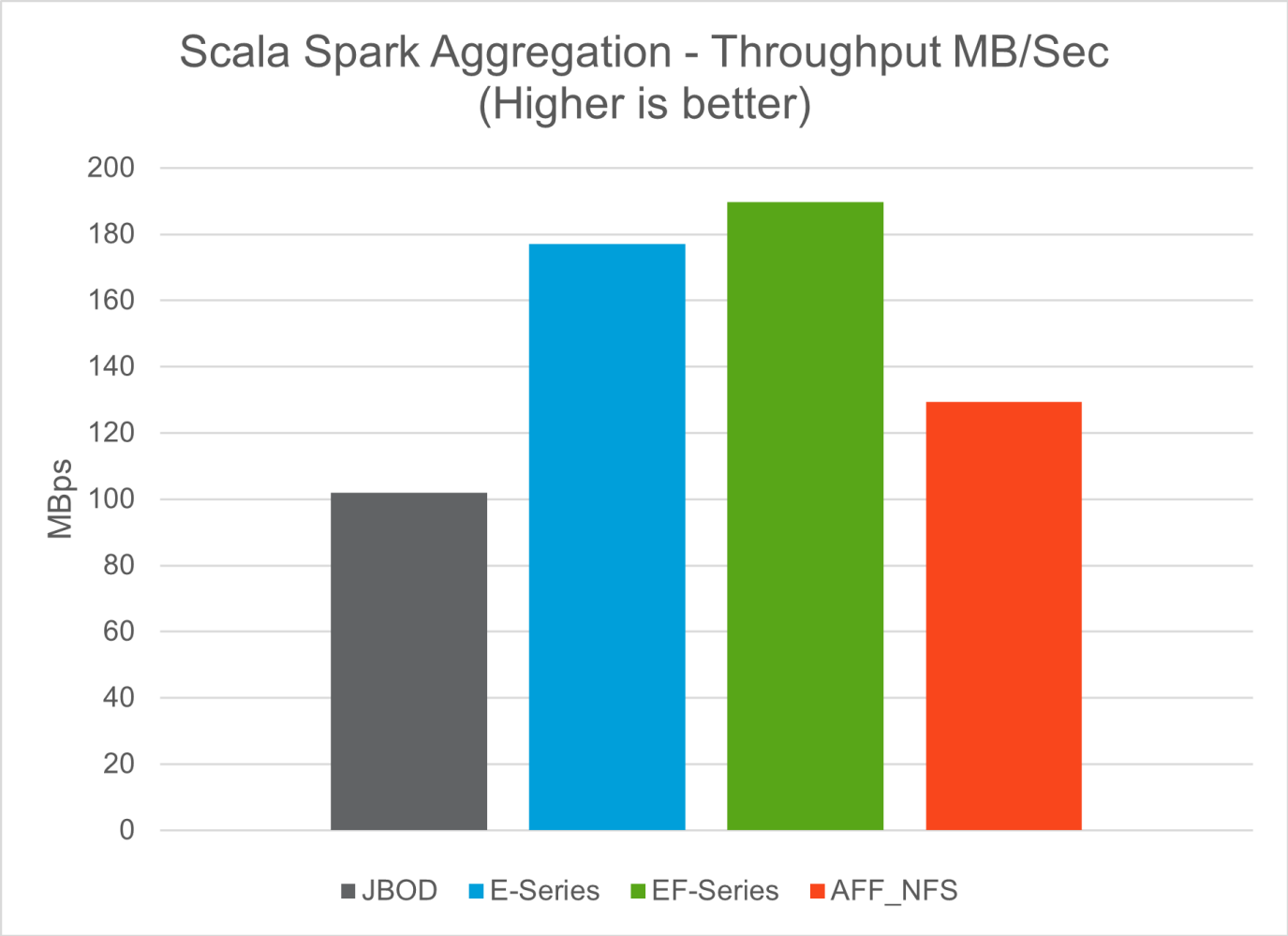
올바른 플래그를 사용하여 TensorFlow를 빌드한 후 다음 스크립트를 실행하여 Critio Display Ads 데이터 세트를 처리하고 DeepFM 모델을 교육하고 예측 점수의 Receiver Operating Characteristic Curve(ROC AUC) 아래에 있는 영역을 계산합니다.

```
(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 교육 Epoch 후에 테스트 데이터 세트에 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 비슷한 방식으로 Spark 워크플로우 런타임을 다른 위치에 있는 데이터와 비교했습니다. 다음 그림은 Spark 워크플로 런타임에 대한 딥 러닝 CTR 예측을 비교한 것입니다.

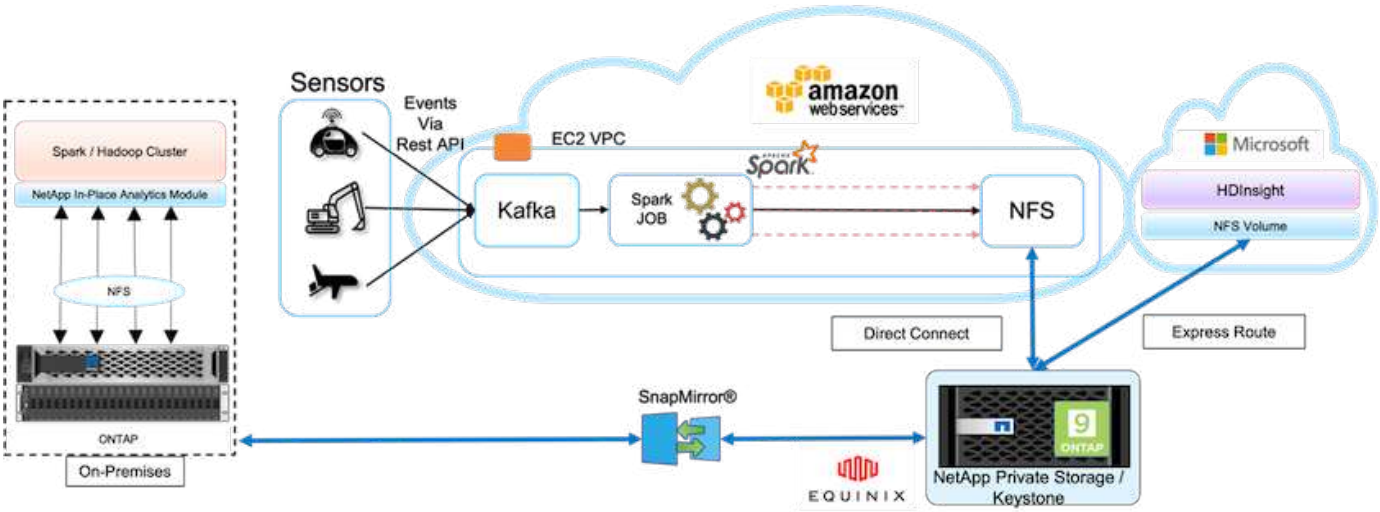


하이브리드 클라우드 솔루션

최신 엔터프라이즈 데이터 센터는 지속적인 데이터 관리 계층을 통해 여러 분산 인프라 환경을 연결하여 온프레미스 및/또는 멀티 퍼블릭 클라우드의 일관된 운영 모델을 유지하는 하이브리드 클라우드입니다. 하이브리드 클라우드를 최대한 활용하려면 데이터 변환 또는 애플리케이션 리팩토링 없이 사내 환경과 멀티 클라우드 환경 간에 데이터를 원활하게 이동할 수 있어야 합니다.

고객은 2차 스토리지를 클라우드로 이동하여 데이터 보호와 같은 사용 사례를 활용하거나 애플리케이션 개발, DevOps 등과 같은 비즈니스 크리티컬 워크로드를 클라우드로 이전함으로써 하이브리드 클라우드 여정을 시작한다고 했습니다. 그런 다음 더 중요한 워크로드로 이동합니다. 웹 및 콘텐츠 호스팅, DevOps 및 애플리케이션 개발, 데이터베이스, 분석, 컨테이너 앱 등이 가장 인기 있는 하이브리드 클라우드 워크로드 중 하나입니다. 엔터프라이즈 AI 프로젝트의 복잡성, 비용, 위험은 기존에 AI 채택의 실험 단계에서 운영 단계로 넘어왔던 것입니다.

NetApp 하이브리드 클라우드 솔루션을 사용하는 고객은 통합 보안, 데이터 거버넌스, 규정 준수 툴을 활용하여 분산 환경에서 데이터 및 워크플로우를 단일 제어판에서 관리할 수 있으며, 사용에 따른 총 소유 비용을 최적화할 수 있습니다. 다음 그림은 고객의 빅데이터 분석 데이터를 위한 멀티 클라우드 연결을 제공하는 클라우드 서비스 파트너의 예입니다.



이 시나리오에서는 여러 소스에서 AWS로 수신된 IoT 데이터가 NPS(NetApp 프라이빗 스토리지)의 중앙 위치에 저장됩니다. NPS 스토리지는 AWS와 Azure에 위치한 Spark 또는 Hadoop 클러스터에 연결되어 여러 클라우드에서 실행되는 빅데이터 분석 애플리케이션이 동일한 데이터에 액세스할 수 있도록 지원합니다. 이 활용 사례의 주요 요구사항과 과제는 다음과 같습니다.

- 고객은 여러 클라우드를 사용하여 동일한 데이터에 대한 분석 작업을 실행하려고 합니다.
- 다양한 센서와 허브를 통해 사내 환경, 클라우드 환경과 같은 다양한 소스로부터 데이터를 받아야 합니다.
- 솔루션은 효율적이고 비용 효율적이어야 합니다.
- 주된 과제는 서로 다른 사내 및 클라우드 환경 간에 하이브리드 분석 서비스를 제공하는 비용 효율적이고 효율적인 솔루션을 구축하는 것입니다.

NetApp의 데이터 보호 및 멀티 클라우드 연결 솔루션은 여러 하이퍼스케일러에 클라우드 분석 애플리케이션이 있다는 문제를 해결합니다. 위 그림과 같이 센서의 데이터가 스트리밍되어 Kafka를 통해 AWS Spark 클러스터로 수집됩니다. 이 데이터는 Equinix 데이터 센터 내의 클라우드 공급자 외부에 있는 NPS에 있는 NFS 공유에 저장됩니다.

NetApp NPS는 각각 Direct Connect와 Express Route 연결을 통해 Amazon AWS 및 Microsoft Azure에 연결되므로, 고객은 데이터 이동 없는 분석 모듈을 활용하여 Amazon 및 AWS 분석 클러스터 모두에서 데이터에 액세스할 수 있습니다. 따라서 사내 스토리지와 NPS 스토리지 모두 ONTAP 소프트웨어를 실행하기 때문에 "SnapMirror를 참조하십시오" NPS 데이터를 사내 클러스터에 미러링하여 사내 및 여러 클라우드 전반에 하이브리드 클라우드 분석 기능을 제공할 수 있습니다.

최적의 성능을 위해 일반적으로 여러 네트워크 인터페이스 및 직접 연결 또는 빠른 경로를 사용하여 클라우드 인스턴스의 데이터에 액세스하는 것이 좋습니다. NetApp은 다음을 비롯한 다른 Data Mover 솔루션을 보유하고 있습니다 "xCP" 및 "BlueXP 복사 및 동기화" 고객이 애플리케이션 인식, 보안 및 비용 효율적인 하이브리드 클라우드 스파크 클러스터를 구축할 수 있도록 지원합니다.

주요 활용 사례별로 Python 스크립트

다음 세 가지 Python 스크립트는 테스트를 거친 세 가지 주요 사용 사례에 해당합니다. 첫 번째는 'sentiment_analysis_sparklp.py'입니다.

```
# 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.py입니다.

```
# 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.py입니다.

```
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 및 빅데이터, 최신 분석, AI, ML 및 DL과 관련된 NetApp 스토리지 포트폴리오에 대해 설명합니다. 업계 표준 벤치마킹 툴 및 고객

요구사항을 기반으로 한 성능 검증 테스트에서 NetApp Spark 솔루션은 네이티브 Hadoop 시스템에 비해 뛰어난 성능을 입증했습니다. 이 보고서에 제공된 고객 사용 사례 및 성능 결과를 조합하여 배포할 적절한 Spark 솔루션을 선택할 수 있습니다.

추가 정보를 찾을 수 있는 위치

이 TR에 사용된 참조 자료는 다음과 같습니다.

- 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/"

- Apache 당면 과제

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

- 스파크 NLP

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

- 베르

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

- Deep and Cross Network for Ad Predictions를 클릭합니다

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

- FlexGroup

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

- 스트리밍 ETL

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

- Hadoop용 NetApp E-Series 솔루션

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

- NetApp AI와 고객 커뮤니케이션을 통한 감정 분석

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

- NetApp의 최신 데이터 분석 솔루션

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

- SnapMirror를 참조하십시오

"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 복사 및 동기화

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

- DataOps 툴킷

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

저작권 정보

Copyright © 2024 NetApp, Inc. All Rights Reserved. 미국에서 인쇄된 본 문서의 어떠한 부분도 저작권 소유자의 사전 서면 승인 없이는 어떠한 형식이나 수단(복사, 녹음, 녹화 또는 전자 검색 시스템에 저장하는 것을 비롯한 그래픽, 전자적 또는 기계적 방법)으로도 복제될 수 없습니다.

NetApp이 저작권을 가진 자료에 있는 소프트웨어에는 아래의 라이센스와 고지사항이 적용됩니다.

본 소프트웨어는 NetApp에 의해 '있는 그대로' 제공되며 상품성 및 특정 목적에의 적합성에 대한 명시적 또는 묵시적 보증을 포함하여(이에 제한되지 않음) 어떠한 보증도 하지 않습니다. NetApp은 대체품 또는 대체 서비스의 조달, 사용 불능, 데이터 손실, 이익 손실, 영업 중단을 포함하여(이에 국한되지 않음), 이 소프트웨어의 사용으로 인해 발생하는 모든 직접 및 간접 손해, 우발적 손해, 특별 손해, 징벌적 손해, 결과적 손해의 발생에 대하여 그 발생 이유, 책임론, 계약 여부, 엄격한 책임, 불법 행위(과실 또는 그렇지 않은 경우)와 관계없이 어떠한 책임도 지지 않으며, 이와 같은 손실의 발생 가능성이 통지되었다 하더라도 마찬가지입니다.

NetApp은 본 문서에 설명된 제품을 언제든지 예고 없이 변경할 권리를 보유합니다. NetApp은 NetApp의 명시적인 서면 동의를 받은 경우를 제외하고 본 문서에 설명된 제품을 사용하여 발생하는 어떠한 문제에도 책임을 지지 않습니다. 본 제품의 사용 또는 구매의 경우 NetApp에서는 어떠한 특허권, 상표권 또는 기타 지적 재산권이 적용되는 라이센스도 제공하지 않습니다.

본 설명서에 설명된 제품은 하나 이상의 미국 특허, 해외 특허 또는 출원 중인 특허로 보호됩니다.

제한적 권리 표시: 정부에 의한 사용, 복제 또는 공개에는 DFARS 252.227-7013(2014년 2월) 및 FAR 52.227-19(2007년 12월)의 기술 데이터-비상업적 품목에 대한 권리(Rights in Technical Data -Noncommercial Items) 조항의 하위 조항 (b)(3)에 설명된 제한사항이 적용됩니다.

여기에 포함된 데이터는 상업용 제품 및/또는 상업용 서비스(FAR 2.101에 정의)에 해당하며 NetApp, Inc.의 독점 자산입니다. 본 계약에 따라 제공되는 모든 NetApp 기술 데이터 및 컴퓨터 소프트웨어는 본질적으로 상업용이며 개인 비용만으로 개발되었습니다. 미국 정부는 데이터가 제공된 미국 계약과 관련하여 해당 계약을 지원하는 데에만 데이터에 대한 전 세계적으로 비독점적이고 양도할 수 없으며 재사용이 불가능하며 취소 불가능한 라이센스를 제한적으로 가집니다. 여기에 제공된 경우를 제외하고 NetApp, Inc.의 사전 서면 승인 없이는 이 데이터를 사용, 공개, 재생산, 수정, 수행 또는 표시할 수 없습니다. 미국 국방부에 대한 정부 라이센스는 DFARS 조항 252.227-7015(b)(2014년 2월)에 명시된 권한으로 제한됩니다.

상표 정보

NETAPP, NETAPP 로고 및 http://www.netapp.com/TM에 나열된 마크는 NetApp, Inc.의 상표입니다. 기타 회사 및 제품 이름은 해당 소유자의 상표일 수 있습니다.