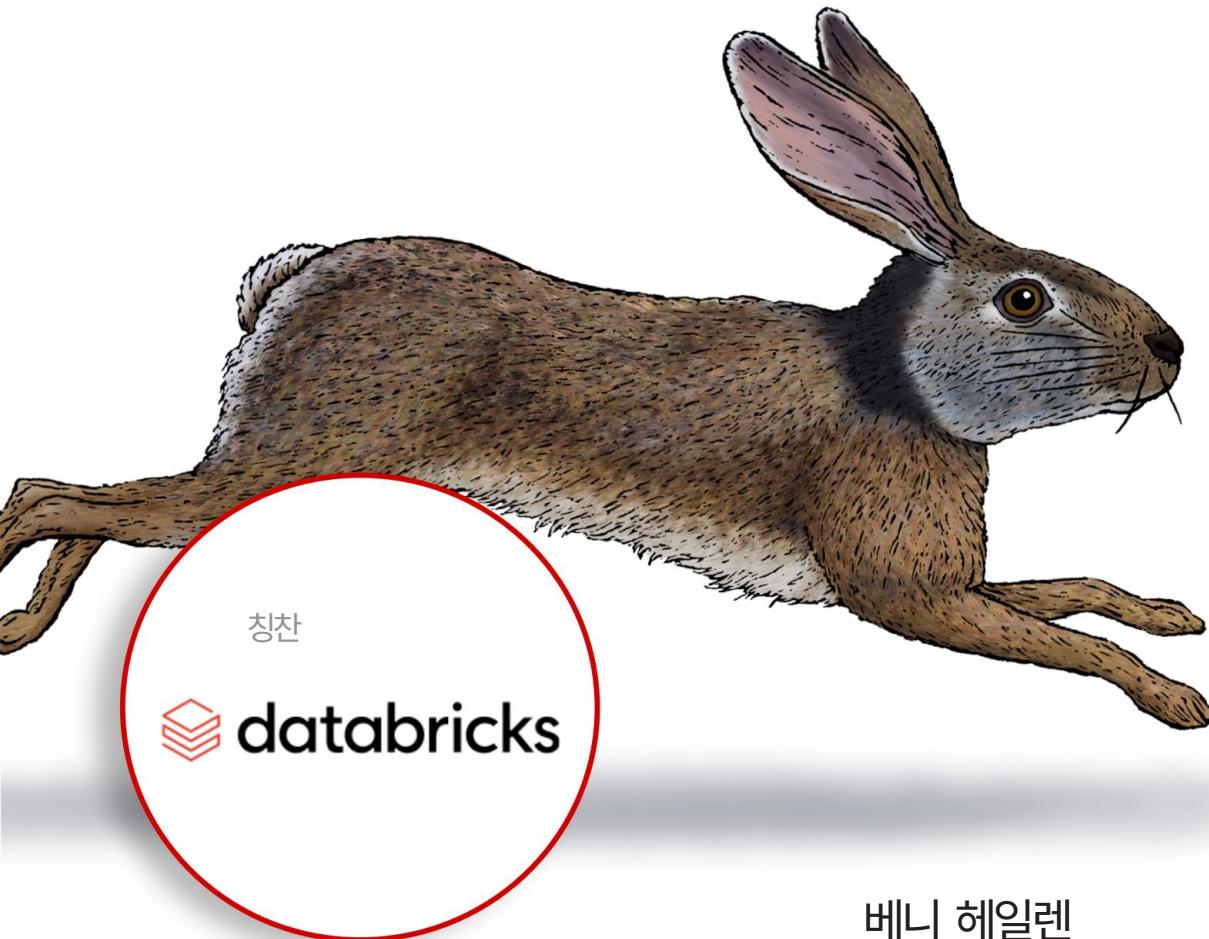


O'REILLY®

델타 레이크 가동 및 실행

Delta Lake를 사용한 최신 데이터 레이크하우스 아키텍처



베니 헤일렌
& 댄 데이비스



Delta Lake: 실행 중

빅데이터와 AI의 급증으로 조직은 데이터 제품을 빠르게 만들 수 있습니다. 그러나 분석 및 기계 학습 모델의 효율성은 데이터 품질에 따라 달라집니다. Delta Lake의 오픈 소스 형식은 Amazon S3, ADLS 및 GCS와 같은 플랫폼을 통해 강력한 레이크하우스 프레임워크를 제공합니다.

이 실용적인 책에서는 데이터 엔지니어, 데이터 과학자, 데이터 분석가에게 Delta Lake와 해당 기능을 시작하고 실행하는 방법을 보여줍니다. 데이터 파이프라인 및 애플리케이션 구축의 궁극적인 목표는 데이터에서 통찰력을 얻는 것입니다. 스토리지 솔루션 선택이 원시 데이터부터 통찰력까지 데이터 파이프라인의 결고성과 성능을 결정하는 방법을 이해하게 됩니다.

다음 방법을 배우게 됩니다.

- 최신 데이터 관리 및 데이터 엔지니어링 기술을 사용합니다.
- ACID 트랜잭션이 어떻게 대규모 데이터 레이크에 안정성을 제공하는지 이해합니다.
- 스트리밍 및 일괄 작업을 실행합니다.
데이터 레이크를 동시에
- 데이터 레이크에 대해 업데이트, 삭제, 병합 명령 실행
- 시간 여행을 통해 롤백하고 조사합니다.
이전 데이터 버전
- 다음과 같은 스트리밍 데이터 품질 파이프라인을 구축합니다.
메달리온 아키텍처

데이터

US \$65.99 캔 \$82.99

ISBN: 978-1-098-13972-8



9 781098 139735
9 781098 139728

56599

Bennie Haelen은 Microsoft와 Databricks 파트너인 Insight Digital Innovation의 수석 설계자입니다. 그는 상업용 클라우드 플랫폼의 최신 데이터 웨어하우징, 기계 학습, 생성 AI 및 IoT에 중점을 두고 있습니다.

Dan Davis는 데이터에서 분석 통찰력과 비즈니스 가치를 제공하는 10년의 경험을 보유한 클라우드 데이터 설계자입니다. 그는 데이터 통합 및 분석을 지원하는 데이터 플랫폼, 프레임워크 및 프로세스를 설계합니다.

트위터: @oreillymedia
[linkedin.com/company/oreilly-media](https://www.linkedin.com/company/oreilly-media)
[youtube.com/oreillymedia](https://www.youtube.com/oreillymedia)



One unified platform for data and AI

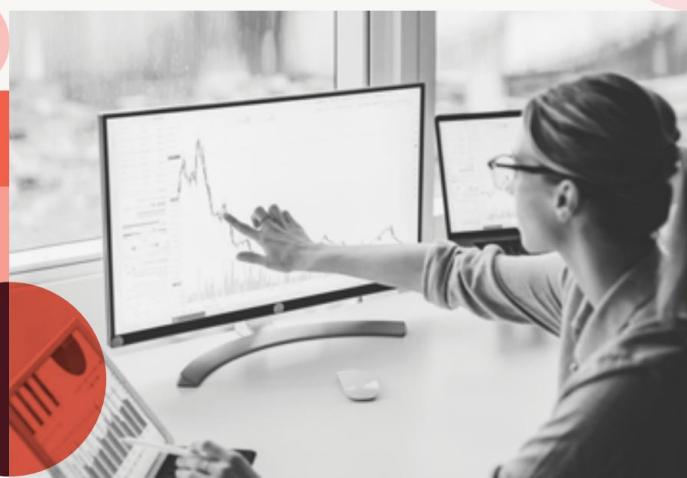
Combine data warehouse performance
with data lake flexibility

데이터 레이크와 데이터 웨어하우스를 모두 유지 관리하는 복잡성으로 인해 데이터 사일로
가 발생하고 비용이 증가하며 의사 결정 속도가 느려집니다.

레이크하우스 아키텍처를 기반으로 구축된 Databricks 플랫폼은 개방적이고 유연한 데이
터 레이크에 데이터 웨어하우스 품질과 안정성을 제공합니다.

이 단순화된 아키텍처는 분석, 스트리밍 데이터, 데이터 과학 및 기계 학습을 위한 단일 환
경을 제공하여 데이터를 최대한 활용하는 데 도움을 줍니다.

databricks.com/lakehouse에서 자세히 알아보세요.



Delta Lake: 실행 중

Delta Lake를 사용한 최신 데이터 레이크하우스 아
키텍처

베니 헤일런과 댄 데이비스

베이징·보스턴 파넘 세바스토플 도쿄·

O'REILLY®

Delta Lake: Bennie Haelen
및 Dan Davis의 Up and Running

저작권 © 2024 O'Reilly Media, Inc. 모든 권리 보유.

미국에서 인쇄되었습니다.

출판사: O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly 도서는 교육, 비즈니스 또는 판매 홍보용으로 구입할 수 있습니다. 대부분의 타이틀에 대해 온라인 버전도 제공됩니다 (<http://oreilly.com>). 자세한 내용은 기업/기관 영업부(800-998-9938 또는 Corporate@oreilly.com)에 문의하세요.

인수 편집자: Aaron Black

개발 편집자: Gary O'Brien

프로덕션 편집자: Ashley Stussy

편집자: Charles Roumeliotis

교정자: 소니아 사루바

인덱서: nSight, Inc.

인테리어 디자이너: David Futato 표지 디

자이너: Karen Montgomery 일러스트레이터:

Kate Dullea

2023년 10월: 초판

초판 개정 내역

2023-10-16: 첫 번째 출시

<http://oreilly.com/catalog/errata.csp?isbn=9781098139728>을 참조하세요. 릴리스 세부정보를 확인하세요.

O'Reilly 로고는 O'Reilly Media, Inc.의 등록 상표입니다. Delta Lake: Up and Running, 표지 이미지 및 관련 상품 외장은 O'Reilly Media, Inc.의 상표입니다.

본 저작물에 표현된 견해는 저자의 견해이며 출판사의 견해를 대변하지 않습니다.

출판사와 저자는 본 저작물에 포함된 정보와 지침이 정확한지 확인하기 위해 선의의 노력을 기울였지만, 출판사와 저자는 본 저작물의 사용으로 인해 발생하는 손해에 대한 책임을 포함하되 이에 국한되지 않고 오류나 누락에 대한 모든 책임을 부인합니다. 이 작업에 의존합니다. 이 저작물에 포함된 정보와 지침을 사용하는 데 따른 위험은 전적으로 귀하의 책임입니다. 본 저작물에 포함되거나 설명된 코드 샘플 또는 기타 기술이 오픈 소스 라이선스 또는 타인의 지적 재산권의 적용을 받는 경우, 이를 사용하는 것이 그리한 라이선스 및/또는 권리를 준수하는지 확인하는 것은 귀하의 책임입니다.

이 작업은 O'Reilly와 Databricks 간의 협력의 일부입니다. [편집 독립성 선언문](#)을 참조하세요 .

978-1-0981-3973-5

[LSI]

목차

머리말	ix
1. 데이터 아키텍처의 진화.....	1
관계형 데이터베이스의 간략한 역사	2
데이터 웨어하우스	삼
데이터 웨어하우스 아키텍처	삼
차원 모델링	7
데이터 웨어하우스의 이점과 과제	8
데이터 레이크 소개	10
데이터 레이크하우스	14
데이터 레이크하우스의 이점	15
레이크하우스 구현	16
델타 레이크	18
메달리온 아키텍처	21
델타 생태계	22
델타 레이크 스토리지	22
델타 공유	23
델타 커넥터	23
결론	24
2. Delta Lake 시작하기.....	25
표준 Spark 이미지 가져오기	26
PySpark와 함께 Delta Lake 사용	26
Spark Scala 셀에서 Delta Lake 실행	27
Databricks에서 Delta Lake 실행	28
Spark 프로그램 생성 및 실행: helloDeltaLake	29
델타 레이크 형식	30
쪽모이 세공 파일	31

델타 테이블 작성	34
Delta Lake 트랜잭션 로그	36
트랜잭션 로그가 원자성을 구현하는 방법	36
트랜잭션을 원자적 커밋으로 나누기	36
파일 수준의 트랜잭션 로그	37
대규모 메타데이터 확장	44
결론	48
3. 델타 테이블의 기본 작업	49
델타 테이블 생성	50
SQL DDL을 사용하여 델타 테이블 생성	50
DESCRIBE 문	53
DataFrameWriter API를 사용하여 델타 테이블 생성	54
DeltaTableBuilder API를 사용하여 델타 테이블 생성	57
생성된 열	58
델타 테이블 읽기	60
SQL을 사용하여 델타 테이블 읽기	60
PySpark로 테이블 읽기	63
델타 테이블에 쓰기	64
YellowTaxis 테이블 청소	65
SQL INSERT를 사용하여 데이터 삽입	65
테이블에 DataFrame 추가	66
델타 테이블에 쓸 때 덮어쓰기 모드 사용	68
SQL COPY INTO 명령을 사용하여 데이터 삽입	68
파티션	70
사용자 정의 메타데이터	76
SparkSession을 사용하여 사용자 정의 메타데이터 설정	77
DataFrameWriter를 사용하여 사용자 정의 메타데이터 설정	78
결론	79
4. 테이블 삭제, 업데이트 및 병합	81
델타 테이블에서 데이터 삭제 테이블 생성 및	81
DESCRIBE HISTORY DELETE 작업 수행 DELETE 성능	82
튜닝 팁 테이블의 데이터 업데이트 사용 사례 설명	84
테이블의 데이터 업데이트 UPDATE 성능 튜닝 팁	86
MERGE 작업을 사용하여 데이터 업	87
서트	87
88	88
90	90
90	90
사용 사례 설명	90
MERGE 데이터세트	91

MERGE 문	92
DESCRIBE HISTORY를 사용하여 MERGE 작업 분석	97
MERGE 작업의 내부 작동	98
결론	98
 5. 성능 튜닝.....	99
데이터 건너뛰기	99
파티셔닝	102
파티셔닝 경고 및 고려 사항	108
컴팩트 파일	109
압축	109
최적화	110
ZORDER BY	113
ZORDER BY 고려사항	117
액체 클러스터링	118
액체 클러스터링 활성화	119
클러스터된 열에 대한 작업	120
액체 클러스터링 경고 및 고려 사항	122
결론	123
 6. 시간여행을 이용한다.....	125
델타 레이크 시간 여행	126
타임스탬프 시간 이동을	128
통한 테이블 복원 내부적으로	129
RESTORE 고려 사항 및 경고 이전 버전	129
의 테이블 쿼리 데이터 보존 데이터 파일 보존 로그 파일 보	131
존 설정 파일 보존 기간 예제 데이터 보관 VACUUM	132
VACUUM 구문 및 예	134
제 VACUUM 및 기타를 열	134
마나 자주 실행해야 합니	136
까? 유지 관리 작업?	136
결론	137
	138
	139
	140
VACUUM 경고 및 고려 사항	141
데이터 피드 변경	143
CDF 활성화	144
CDF 보기	146
CDF 경고 및 고려 사항	149
결론	150

7. 스키마 처리.....	151
스키마 검증	152
트랜잭션 로그 항목에서 스키마 보기	152
쓰기 스키마	153
스키마 적용 예	154
스키마 진화	157
열 추가	158
소스 DataFrame의 데이터 열이 누락되었습니다.	160
열 데이터 유형 변경	162
NullType 열 추가	164
명시적 스키마 업데이트	165
테이블에 열 추가	166
열에 설명 추가	167
열 순서 변경	168
Delta Lake 열 매핑	169
열 이름 바꾸기	171
테이블 열 교체	172
열 삭제	174
REORG TABLE 명령	177
열 데이터 유형 또는 이름 변경	179
결론	181
8. 스트리밍 데이터에 대한 작업.....	183
스트리밍 개요	184
Spark 구조적 스트리밍	184
Delta Lake 및 구조적 스트리밍	184
스트리밍 예시	185
안녕하세요 스트리밍 월드	185
지금 스트리밍 가능	195
소스 레코드 업데이트	197
변경 데이터 피드에서 스트림 읽기	201
결론	204
9. 델타 공유.....	205
기존의 데이터 공유 방법 205	
레거시 및 자체 개발 솔루션 독점 공급업체 솔	206
루션 클라우드 개체 스토리지 오픈 소	207
스 델타 공유 델타 공유 목	209
표 델타 공유 내부 데이터 제공자 및	210
수신자	210
	211
	211

디자인의 이점	212
델타 공유 저장소	213
1단계: Python 커넥터 설치	213
2단계: 프로필 파일 설치	213
3단계: 공유 테이블 읽기	214
결론	215
 10. Delta Lake에 Lakehouse 건설.....	217
스토리지 계층 데	218
이터 레이크란 무엇입니까?	218
데이터 유형 클라	218
우드의 주요 이점 데이터 레이크 데이터 관리	219
SQL 분석 Spark SQL	222
을 통한 SQL 분석	225
기타 Delta Lake 통합을 통한 SQL	225
분석	227
데이터 과학 및 기계 학습을 위한 데이터	229
기존 기계 학습의 과제	230
기계 학습을 지원하는 Delta Lake 기능	231
함께 모아서	233
메달리온 아키텍처	234
청동총(원시 데이터)	236
실버 레이어	237
골드 레이어	237
더 컴플리트 레이크하우스	238
결론	240
 색인.....	241

머리말

이 책의 목표는 데이터 실무자에게 Delta Lake를 설정하고 고유한 기능을 사용하는 방법에 대한 실용적인 지침을 제공하는 것입니다. 이 책은 다음 프로필 중 하나에 해당하는 독자를 위해 설계되었습니다.

- Spark 배경을 가진 데이터 실무자 • Delta Lake에 익숙하

지 않거나 처음 접하는 데이터 실무자로서 기술, 이 기술이 해결하는 문제, 주요 기능 및 용어, 사용을 시작하는 방법에 대한 소개가 필요합니다.

- 최신 레이크하우스 아키텍처의 기능과 이점에 대해 알아보려는 데이터 실무자

이 책과 논의된 기능은 [Delta Lake 오픈 소스 프레임워크](#)에 적용된다는 점에 유의하는 것이 중요합니다. (델타 레이크 OSS). Delta Lake 주변에서 일부 회사가 제공하는 독점 기능 및 최적화는 이 책의 범위를 벗어나는 것으로 간주됩니다.

먼저 Delta Lake가 최신 엔터프라이즈 데이터 플랫폼, 데이터 과학 및 AI 솔루션을 구축하는 데 중요한 도구인 이유를 논의한 다음 Spark로 Delta Lake를 설정하는 방법에 대한 지침을 설명합니다. 각 후속 장에서는 단계별 지침과 실제 사례를 사용하여 Delta Lake의 기본 기능과 작동을 안내합니다.

이 책의 코드 예제는 PySpark 셀에서 사용할 수 있는 코드 조각부터 완전한 엔드투엔드 노트북에서 실행되도록 설계된 코드 조각까지 다양합니다. 이 책의 모든 코드 조각은 Python, SQL 및 필요한 경우 셀 명령에 포함됩니다.

GitHub [저장소](#) 독자들이 책 전체를 따라가는 데 도움이 되도록 제공됩니다. 데이터 세트, 파일 및 코드 샘플은 리포지토리에 제공되며 책 전체에서 참조됩니다. 다음은 GitHub 저장소 사용에 관해 참고해야 할 몇 가지 중요한 사항입니다.

코드 샘플 코드

샘플은 리포지토리에 장별로 구성되어 있으며 대부분의 장에서 장 초기화 스크립트는 특정 장과 관련된 코드를 실행하기 전에 실행되도록 되어 있습니다. 논의 중인 주제를 가장 잘 보여주기 위해 적절한 델타 테이블과 데이터세트를 설정하려면 코드를 실행하기 전에 이 장의 초기화 코드가 필요합니다. 이러한 장 초기화 스크립트는 주어진 장에 대한 첫 번째 샘플 코드 세트를 실행하기 전에 책의 텍스트에서 명시적으로 호출됩니다.

코드 샘플 데이터 파일 제공

된 코드 샘플을 실행하는 데 필요한 데이터 파일은 GitHub 저장소에 있습니다. GitHub 리포지토리의 데터 파일은 인기 있는 [NYC Yellow 및 Green 택시 여행 기록에서 가져온 것입니다](#). 이 파일은 이 책 전반에 걸쳐 효과적인 시연을 위해 다운로드되어 선별되었습니다.

이 책에서 Delta Lake를 실행하는 방법

이 책의 목적과 제공된 GitHub 리포지토리의 코드를 위해 Delta Lake를 실행하는 방법은 [Databricks Community Edition입니다](#). Databricks Community Edition은 무료이고 Spark 및 Delta Lake 설정을 단순화하며 자체 클라우드 계정이 필요하지 않거나 클라우드 컴퓨팅 또는 스토리지 리소스를 제공할 필요가 없기 때문에 코드 샘플을 개발하고 실행하기 위해 선택되었습니다.

이 책과 GitHub 리포지토리에 사용된 델타 테이블, 데이터 세트 및 코드 샘플은 Azure Data Lake Storage Gen2를 기본 스토리지 계층으로 사용하고 Databricks Runtime 12.2 LTS를 사용하여 Azure에서 호스팅되는 Databricks Community Edition에서 개발 및 테스트되었습니다. Databricks 외부(예: 로컬 컴퓨터)의 Spark 및 Delta Lake에서 코드 샘플을 실행하는 경우 독자가 고려해야 할 추가 설정, 구성 및 잠재적 편집기 구문 옵션이 있습니다.

노트북

학기 노트북도 볼 수 있습니다. 노트북은 [Databricks 노트북을 의미하며](#), 책 전반에 걸쳐 코드를 개발하고 결과를 제시하는 기본 도구입니다.

코드 언어 Delta

Lake는 다양한 기능을 위해 여러 언어(Scala, Java, Python 및 SQL)를 지원합니다. 이 책은 주로 Python과 SQL에 중점을 둘 것입니다. 코드 샘플은 논의 중인 주제에 가장 적합하다고 간주되는 언어로 코드를 제공합니다. 다른 언어의 유사한 기능에 대한 대안이 항상 제공되는 것은 아닙니다. [Delta Lake 설명서를 참조하세요](#). 대체 언어로 유사한 기능을 보려면

이 책 전체에 사용된 코드 조각의 경우 기본 언어는 Python입니다.

코드 조각에서 Python 이외의 언어 사용을 나타내기 위해 [언어 매직 명령이 표시됩니다](#). 즉, %<언어> (예: %sql)입니다. 언어 마법 명령이 없는 코드 조각이 Python을 사용하고 있다고 가정할 수 있습니다.

저희에게 연락하는 방법

이 책에 관한 의견과 질문은 출판사에 문의하십시오.

오라일리 미디어, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969(미국 또는 캐나다) 707-829-7019(국제 또는 현지)
707-829-0104(팩스) support@oreilly.com
<https://www.oreilly.com/about/contact.html>

이 책의 웹페이지에는 정오프, 예제 및 추가 정보가 나열되어 있습니다. 이 페이지는 <https://oreil.ly/delta-lake-up-and-running-1e>에서 액세스할 수 있습니다.

도서 및 강좌에 대한 뉴스와 정보를 보려면 <https://oreilly.com>을 방문하세요.

LinkedIn에서 우리를 찾아보세요: <https://linkedin.com/company/oreilly-media>.

트위터에서 우리를 팔로우하세요: <https://twitter.com/oreillymedia>.

YouTube에서 시청하세요: <https://youtube.com/oreillymedia>.

0 | 책에 사용된 규칙

이 책에서는 다음과 같은 표기 규칙을 사용합니다.

기울임

꼴 새 용어, URL, 이메일 주소, 파일 이름 및 파일 확장자를 나타냅니다.

상수 너비 프로그램 목록뿐

만 아니라 단락 내에서 변수 또는 함수 이름, 데이터베이스, 데이터 유형, 환경 변수, 명령문 및 키워드와 같은 프로그램 요소를 참조하는 데 사용됩니다.

일정한 너비의 굵게 사용자가 문

자 그대로 입력해야 하는 명령이나 기타 텍스트를 표시합니다.

일정한 너비 기울임꼴 사용자가 제공

한 값이나 상황에 따라 결정된 값으로 대체되어야 하는 텍스트를 표시합니다.



이 요소는 팁이나 제안을 나타냅니다.



이 요소는 일반적인 참고 사항을 나타냅니다.



이 요소는 경고 또는 주의를 나타냅니다.

코드 예제 사용

보충 자료(코드 예제, 연습 등)는 <https://github.com/benniehaelen/delta-lake-up-and-running>에서 다운로드할 수 있습니다.

기술적인 질문이 있거나 코드 예제를 사용하는 데 문제가 있는 경우 support@oreilly.com으로 이메일을 보내주세요.

이 책은 당신이 일을 완수하는 데 도움을 주기 위해 여기에 있습니다. 일반적으로 이 책에 예제 코드가 제공되면 이를 프로그램과 문서에 사용할 수 있습니다. 코드의 상당 부분을 복제하는 경우가 아니면 허가를 받기 위해 당사에 연락할 필요가 없습니다. 예를 들어, 이 책에 있는 여러 코드 링어리를 사용하는 프로그램을 작성하는 데에는 허가가 필요하지 않습니다. O'Reilly 도서의 예제를 판매하거나 배포하려면 허가가 필요합니다. 이 책을 인용하고 예제 코드를 인용하여 질문에 답하는 데에는 허가가 필요하지 않습니다. 이 책의 상당량의 예제 코드를 제품 문서에 통합하려면 허가가 필요합니다.

우리는 저작자 표시를 높이 평가하지만 일반적으로 요구하지는 않습니다. 속성에는 일반적으로 제목, 저자, 출판사 및 ISBN이 포함됩니다. 예: “Delta Lake: Bennie Haelen 및 Dan Davis(O'Reilly)의 Up and Running. 저작권 2024 O'Reilly Media, Inc., 978-1-098-13972-8.”

코드 예제 사용이 공정 사용 또는 위에 제공된 권한에 어긋난다고 생각되는 경우, 허가@oreilly.com으로 언제든지 문의해 주십시오.

오라일리 온라인 학습

O'REILLY® 40년이 넘는 세월 동안 **오라일리 미디어(O'Reilly Media)**는 기업의 성공에 도움이 되는 기술과 비즈니스 교육, 지식, 통찰력을 제공했습니다.

전문가와 혁신가로 구성된 당사의 고유한 네트워크는 책, 기사 및 온라인 학습 플랫폼을 통해 지식과 전문 지식을 공유합니다. O'Reilly의 온라인 학습 플랫폼을 사용하면 실시간 교육 과정, 심층 학습 경로, 대화형 코딩 환경, O'Reilly 및 200개 이상의 기타 출판사가 제공하는 방대한 텍스트 및 비디오 컬렉션에 대한 주문형 액세스를 제공합니다. 자세한 내용은 <https://oreilly.com>을 방문하세요.

승인

기술 검토자인 Adam Breindel, Andrei Ionescu 및 Jobenish Purushothaman에게 감사의 말씀을 전하고 싶습니다. 세부 사항에 대한 그들의 관심, 피드백, 사려 깊은 제안은 이 책의 내용을 구체화하는 동시에 정확성을 보장하는 데 중추적인 역할을 했습니다. 그들의 의견은 의심할 바 없이 이 책이 독자들에게 귀중한 자원이 될 더 나은 품질의 책을 만드는데 도움이 되었습니다.

기술 검토자 외에도 책을 집필하는 과정 전반에 걸쳐 다른 기고자들로부터 귀중한 피드백을 받았습니다. Alex Ott, Anthony Krinsky, Artem Sheiko, Bilal Obeidat, Carlos Morillo, Eli Swanson, Guillermo G. Schiava D'Albano, Jitesh Soni, Joe Widen, Kyle Hale, Marco Scagliola, Nick에게 감사의 말씀을 전하고 싶습니다. 카르포프, 누란 유니스, 오리 조하르, 시루이 선, 수잔 피어스, 유세프 므리니. 귀하의 의견이 없었다면 이 책은 가치가 없을 것입니다.

자원이다.

마지막으로 오픈 소스 커뮤니티에 감사의 말씀을 전하고 싶습니다. 커뮤니티의 기여와 집단적 노력이 없었다면 Delta Lake는 오늘날의 뛰어난 기능을 갖추지 못했을 것입니다. 혁신에 대한 커뮤니티의 헌신은 Delta Lake의 발전과 영향력을 촉진하는데 도움이 되며, 우리는 다른 사람들과 함께 감사와 감사를 충분히 표현할 수 없습니다.

베니 헤일렌

나의 훌륭한 아내 Jenny에게 감사의 말씀을 전하고 싶습니다. 이 책을 집필하는 내내 당신은 항상 저를 격려하고 동기를 부여해 주었습니다. 당신은 내 인생의 큰 영감입니다. 내 인생의 어려운 시기를 함께 해준 공동 저자 Dan에게 감사드립니다. 댄, 당신 앞에는 훌륭한 경력이 있습니다. 하루 중 언제라도 어려운 질문으로 항상 연락할 수 있는 친구와 동료들에게 감사드립니다.

댄 데이비스

가족들에게 감사 인사를 전하고 싶습니다. 여러분의 지속적인 격려와 지지가 제가 오늘의 이 책을 쓰기까지의 여정의 밑거름이 되었습니다. 항상 끊임없는 동기부여가 되어주셔서 감사합니다. 또한, 그동안 저에게 많은 것을 배우고 지속적으로 지원해 준 모든 친구와 동료들에게도 감사의 말씀을 전하고 싶습니다. 공동 저자인 Bennie에게 아무리 감사해도 부족함이 없습니다. 멘토가 되어주시고, 저에게 도움을 주시고, 좋은 기회를 주셔서 감사합니다. 그리고 마지막으로, 즐겁든 싫든 항상 내 곁에 있어주는 사랑하는 반려견 리버에게 감사 인사를 전하고 싶습니다.

제1장

데이터 아키텍처의 진화

데이터 엔지니어는 최첨단 성능을 제공하는 대규모 데이터, 기계 학습, 데이터 과학 및 AI 솔루션을 구축하고 싶어합니다. 대량의 소스 데이터를 수집한 다음 데이터를 정리, 정규화 및 결합하고 궁극적으로 사용하기 쉬운 데이터 모델을 통해 이 데이터를 다운스트림 애플리케이션에 제공함으로써 이러한 솔루션을 구축합니다.

수집하고 처리해야 하는 데이터의 양이 계속 증가함에 따라 스토리지를 수평적으로 확장할 수 있는 기능이 필요합니다. 또한 처리 및 소비 급증을 해결하려면 컴퓨팅 리소스를 동적으로 확장하는 기능이 필요합니다. 데이터 소스를 하나의 데이터 모델로 결합하므로 테이블에 데이터를 추가해야 할 뿐만 아니라 복잡한 비즈니스 논리를 기반으로 레코드를 삽입, 업데이트 또는 삭제(예: MERGE 또는 UPSERT)해야 하는 경우도 많습니다. 트랜잭션 보장을 통해 대용량 데이터 파일을 지속적으로 다시 작성할 필요 없이 이러한 작업을 수행할 수 있기를 원합니다.

과거에는 이전 요구 사항 집합이 두 가지 도구 집합으로 해결되었습니다.

클라우드 기반 데이터 레이크는 수평적 확장성과 스토리지와 컴퓨팅의 분리를 제공했고, 관계형 데이터 웨어하우스는 트랜잭션 보장을 제공했습니다. 그러나 기존 데이터 웨어하우스는 스토리지와 컴퓨팅을 오프레미스 어플라이언스에 긴밀하게 결합했으며 데이터 레이크와 관련된 수평 확장성이 없었습니다.

Delta Lake는 동적 수평 확장성과 데이터 레이크의 스토리지 및 컴퓨팅 분리를 유지하면서 트랜잭션 안정성, UPSERT 및 MERGE 지원과 같은 기능을 데이터 레이크에 제공합니다. Delta Lake는 최고의 데이터 웨어하우스와 데이터 레이크를 결합한 개방형 데이터 아키텍처인 데이터 레이크하우스를 구축하기 위한 하나의 솔루션입니다.

이 소개에서는 관계형 데이터베이스와 이것이 데이터 웨어하우스로 어떻게 발전했는지 간략하게 살펴보겠습니다. 다음으로, 데이터 레이크 출현의 주요 동인을 살펴보겠습니다. 각 아키텍처의 장점과 단점을 설명하고 마지막으로 Delta Lake 스토리지 계층이 각 아키텍처의 장점을 결합하여 데이터 레이크하우스 솔루션을 생성하는 방법을 보여줍니다.

관계형 데이터베이스의 간략한 역사

1970년의 역사적인 논문에서¹ EF Codd는 데이터를 물리적 데이터 저장소와는 별개로 논리적 관계로 보는 개념을 소개했습니다. 데이터 엔터티 간의 이러한 논리적 관계는 데이터베이스 모델 또는 스키마로 알려졌습니다. Codd의 저술은 관계형 데이터베이스의 탄생을 가져왔습니다. 최초의 관계형 데이터베이스 시스템은 1970년대 중반 IBM과 UBC에 의해 출시되었습니다.

관계형 데이터베이스와 그 기본 SQL 언어는 1980년대와 1990년대에 엔터프라이즈 애플리케이션을 위한 표준 스토리지 기술이 되었습니다. 이러한 인기의 주된 이유 중 하나는 관계형 데이터베이스가 트랜잭션이라는 개념을 제공하기 때문입니다. 데이터베이스 트랜잭션은 일반적으로 약어 ACID로 참조되는 원자성, 일관성, 격리성 및 내구성이라는 네 가지 속성을 충족하는 데이터베이스에 대한 일련의 작업입니다.

원자성은 데이터베이스에 대한 모든 변경 사항이 단일 작업으로 실행되도록 보장합니다. 이는 모든 변경이 성공적으로 수행된 경우에만 트랜잭션이 성공한다는 의미입니다. 예를 들어 온라인뱅킹 시스템을 사용하여 저축에서 당좌예금으로 돈을 이체하는 경우 원자성 속성은 해당 돈이 내 저축 계좌에서 공제되고 당좌 예금 계좌에 추가되어야만 작업이 성공하도록 보장합니다. 전체 작업은 전체 단위로 성공하거나 실패합니다.

일관성 속성은 데이터베이스가 트랜잭션 시작 시 하나의 일관된 상태에서 트랜잭션이 끝날 때 다른 일관된 상태로 전환되도록 보장합니다. 이전 예에서 자금 이체는 저축 계좌에 충분한 자금이 있는 경우에만 발생했습니다. 그렇지 않으면 거래가 실패하고 잔액은 원래의 일관된 상태로 유지됩니다.

격리는 데이터베이스 내에서 발생하는 동시 작업이 서로 영향을 미치지 않도록 보장합니다. 이 속성은 여러 트랜잭션이 동시에 실행될 때 해당 작업이 서로 간섭하지 않도록 보장합니다.

¹ 코드, EF(1970). 관계형 데이터베이스: 생산성을 위한 실용적인 기반. 산호세: 산호세 연구 실험실.

내구성은 커밋된 트랜잭션의 지속성을 나타냅니다. 트랜잭션이 성공적으로 완료되면 시스템 오류가 발생하더라도 영구적인 상태가 유지되도록 보장합니다. 송금 예에서 내구성은 저축 계좌와 당좌 계좌 모두에 대한 업데이트가 지속적이고 잠재적인 시스템 오류에서 살아남을 수 있도록 보장합니다.

데이터베이스 시스템은 1990년대 내내 계속해서 발전했고, 1990년대 중반 인터넷의 출현으로 데이터가 폭발적으로 증가하고 이 데이터를 저장해야 할 필요성이 커졌습니다. 엔터프라이즈 애플리케이션은 관계형 데이터베이스 관리 시스템(RDBMS) 기술을 매우 효과적으로 사용하고 있습니다. SAP 및 Salesforce와 같은 주력 제품은 엄청난 양의 데이터를 수집하고 유지합니다.

그러나 이러한 개발에는 단점이 없던 것은 아닙니다. 엔터프라이즈 애플리케이션은 데이터를 자체 독점 형식으로 저장하므로 데이터 사일로가 증가합니다. 이러한 데이터 사일로는 하나의 부서 또는 사업 단위에서 소유하고 관리했습니다.

시간이 지남에 따라 조직은 이러한 다양한 데이터 사일로에 대한 엔터프라이즈 뷰를 개발해야 할 필요성을 인식하여 데이터 웨어하우스의 등장을 가져왔습니다.

데이터 웨어하우스

각 엔터프라이즈 애플리케이션에는 특정 유형의 보고 기능이 내장되어 있지만 조직 전체에 대한 포괄적인 보기가 부족하여 비즈니스 기회를 놓쳤습니다. 동시에 조직은 장기간에 걸쳐 데이터를 분석하는 것의 가치를 인식했습니다. 또한 그들은 고객, 제품, 기타 사업체 등 여러 교차 주제에 대한 데이터를 분석할 수 있기를 원했습니다.

이로 인해 기업의 모든 관점을 포괄하는 통합 스키마를 통해 비즈니스에 대한 단일 통합 기록 보기 를 제공하는 여러 데이터 소스의 통합 기록 데이터의 중앙 관계형 저장소인 데이터 웨어하우스가 도입되었습니다.

데이터 웨어하우스 아키텍처

일반적인 데이터 웨어하우스 아키텍처의 간단한 표현이 [그림 1-1](#)에 나와 있습니다.

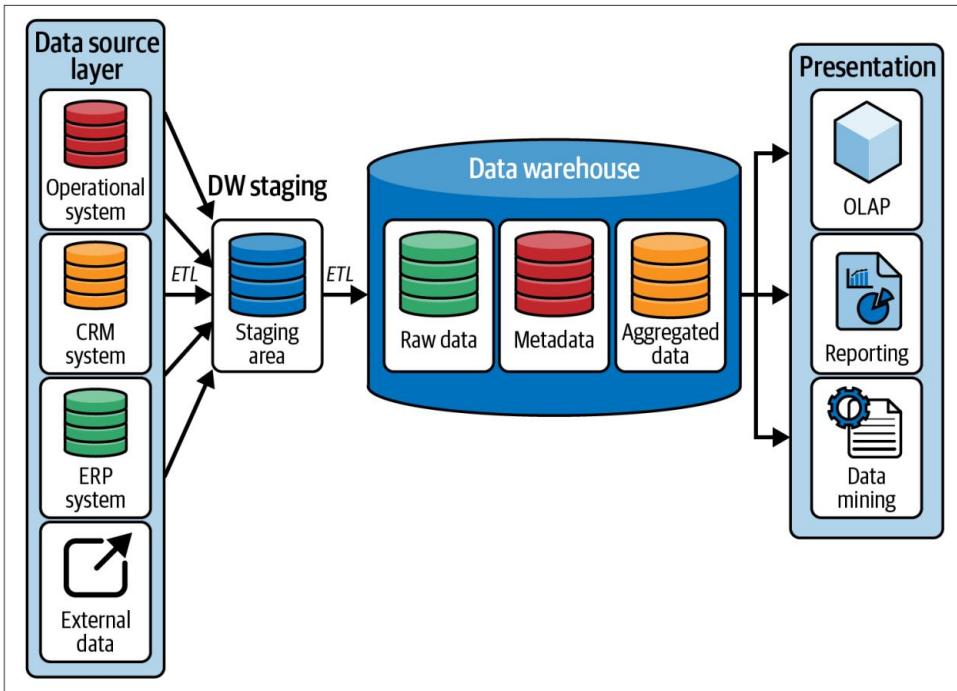


그림 1-1. 데이터 웨어하우스 아키텍처

그림 1-1 의 다이어그램을 보면 원쪽의 데이터 소스 계층부터 시작합니다. 조직은 이기종 데이터 소스 세트에서 데이터를 수집해야 합니다.

조직의 전사적 자원 관리(ERP) 시스템의 데이터가 조직 모델의 중추를 형성하는 동안, 우리는 인적 자원과 같은 일상적인 작업을 실행하는 운영 시스템의 데이터로 이 데이터를 보강해야 합니다. (HR) 시스템 및 워크플로우 관리 소프트웨어. 또한 조직에서는 CRM(고객 관계 관리) 및 POS(판매 시점) 시스템에서 다루는 고객 상호 작용 데이터를 활용하기를 원할 수 있습니다. 여기에 나열된 핵심 데이터 소스 외에도 스프레드시트, CSV 파일 등과 같은 다양한 형식의 다양한 외부 데이터 소스에서 데이터를 수집해야 합니다.

이러한 서로 다른 소스 시스템에는 각각 고유한 데이터 형식이 있을 수 있습니다. 따라서 데이터 웨어하우스에는 다양한 소스의 데이터를 하나의 공통 형식으로 결합할 수 있는 준비 영역이 포함되어 있습니다. 이를 위해 시스템은 원본 데이터 소스에서 데이터를 수집해야 합니다. 실제 수집 프로세스는 데이터 원본 유형에 따라 다릅니다. 일부 시스템은 직접 데이터베이스 액세스를 허용하고 다른 시스템은 API를 통해 데이터를 수집하도록 허용하는 반면, 많은 데이터 소스는 여전히 파일 추출에 의존합니다.

다음으로, 데이터 웨어하우스는 데이터를 표준화된 형식으로 변환하여 다운스트림 프로세스가 데이터에 쉽게 액세스할 수 있도록 해야 합니다. 마지막으로, 변형된

데이터가 준비 영역에 로드됩니다. 관계형 데이터 웨어하우스에서 이 준비 영역은 일반적으로 기본 키나 외래 키 또는 단순 데이터 유형이 없는 단순 관계형 준비 테이블 세트입니다.

데이터를 추출하고 이를 표준 형식으로 변환하여 데이터 웨어하우스에 로드하는 프로세스를 일반적으로 ETL(추출, 변환 및 로드)이라고 합니다. ETL 도구는 데이터를 최종적으로 데이터 웨어하우스에 로드하기 전에 수집된 데이터에 대해 여러 가지 다른 작업을 수행할 수 있습니다. 이러한 작업에는 중복 기록 제거가 포함됩니다. 데이터 웨어하우스는 단일 정보 소스가 되므로 동일한 데이터의 여러 복사본을 포함하는 것을 원하지 않습니다. 또한 중복 레코드는 각 레코드에 대한 고유 키 생성을 방지합니다.

ETL 도구를 사용하면 여러 데이터 소스의 데이터를 결합할 수도 있습니다. 예를 들어 고객에 대한 한 가지 관점은 CRM 시스템에서 캡처되고 다른 속성은 ERP 시스템에서 찾을 수 있습니다. 조직은 이러한 다양한 측면을 고객에 대한 하나의 포괄적인 보기로 결합해야 합니다. 여기에서 데이터 웨어하우스에 스키마를 도입하기 시작합니다. 고객의 예에서 스키마는 고객 테이블에 대한 다양한 열, 필요한 열, 각 열의 데이터 유형 및 제약 조건 등을 정의합니다.

날짜 및 시간과 같은 표준적이고 표준화된 열 표현을 갖는 것이 중요합니다. ETL 도구는 데이터 웨어하우스 전체에서 동일한 표준을 사용하여 모든 임시 열의 형식을 지정하도록 할 수 있습니다.

마지막으로 조직은 데이터 가버넌스 표준에 따라 데이터에 대한 품질 검사를 수행하기를 원합니다. 여기에는 이 최소 표준을 충족하지 않는 품질이 낮은 데이터 행을 삭제하는 것이 포함될 수 있습니다.

데이터 웨어하우스는 메모리, 컴퓨팅 및 스토리지를 결합한 단일 대형 노드로 구성된 모놀리식 물리적 아키텍처에서 물리적으로 구현됩니다. 이 모놀리식 아키텍처로 인해 조직은 인프라를 수직으로 확장해야 하므로 비용이 많이 들고 종종 과대포장된 인프라가 발생하여 최대 사용자 로드에 맞춰 프로비저닝되고 다른 시간에는 거의 유휴 상태가 됩니다.

데이터 웨어하우스에는 일반적으로 다음과 같이 분류할 수 있는 데이터가 포함되어 있습니다.

메타데이터

데이터에 대한 상황별 정보입니다. 이 데이터는 종종 데이터 카탈로그에 저장됩니다.

이를 통해 데이터 분석가는 데이터 웨어하우스에 저장된 데이터를 설명, 분류 및 쉽게 찾을 수 있습니다.

원시 데이터

는 아무런 처리 없이 원본 형식으로 유지됩니다. 원시 데이터에 액세스하면 로드 오류가 발생할 경우 데이터 웨어하우스 시스템에서 데이터를 다시 처리할 수 있습니다.

요약 데이터 기

본 데이터 관리 시스템에 의해 자동으로 생성됩니다. 새 데이터가 웨어하우스에 로드되면 요약 데이터가 자동으로 업데이트됩니다. 여기에는 일치하는 여러 차원에 대한 집계가 포함되어 있습니다. 요약 데이터의 주요 목적은 쿼리 성능을 가속화하는 것입니다.

웨어하우스의 데이터는 프리젠테이션 계층에서 소비됩니다. 이곳은 소비자가 창고에 저장된 데이터와 상호 작용할 수 있는 곳입니다. 우리는 크게 두 가지 소비자 그룹을 식별할 수 있습니다.

인간 소비자 이들은 웨어하우

스의 데이터를 소비해야 하는 조직 내 사람들입니다. 이러한 소비자는 업무의 필수적인 부분으로 데이터에 액세스해야 하는 지식 근로자 부터 일반적으로 대시보드 및 핵심성과지표(KPI) 형식으로 고도로 요약된 데이터를 소비하는 임원까지 다양합니다.

내부 또는 외부 시스템

데이터 웨어하우스의 데이터는 다양한 내부 또는 외부 시스템에서 사용될 수 있습니다. 여기에는 기계 학습 및 AI 도구 세트 또는 웨어하우스 데이터를 소비해야 하는 내부 애플리케이션이 포함될 수 있습니다. 일부 시스템은 데이터에 직접 액세스할 수 있고, 다른 시스템은 데이터 추출로 작업할 수 있으며, 다른 시스템은 게시-구독 모델에서 데이터를 직접 사용할 수 있습니다.

인간 소비자는 다양한 분석 도구와 기술을 활용하여 다음을 포함하여 데이터에 대한 실행 가능한 통찰력을 창출합니다.

보고 도구 이러

한 도구를 사용하면 사용자는 표 형식 보고서 및 다양한 그래픽 표현과 같은 시각화를 통해 데이터에 대한 통찰력을 개발할 수 있습니다.

온라인 분석 처리(OLAP) 도구

소비자는 다양한 방법으로 데이터를 분석해야 합니다. OLAP 도구는 데이터를 다차원 형식으로 표시하므로 다양한 관점에서 쿼리할 수 있습니다. 이들은 종종 메모리에 저장되는 사전 저장된 집계를 활용하여 빠른 성능으로 데이터를 제공합니다.

데이터 마이닝

이러한 도구를 사용하면 데이터 분석가는 수학적 상관 관계 및 분류를 통해 데이터에서 패턴을 찾을 수 있습니다. 이는 분석자가 이전에 다양한 데이터 소스 간에 숨겨진 관계를 인식하는 데 도움을 줍니다. 어떤 면에서 데이터 마이닝 도구는 현대 데이터 과학 도구의 전조로 볼 수 있습니다.

차원 모델링 데이터 웨어하우스

우스는 기업의 다양한 주제 영역을 포괄하는 포괄적인 데이터 모델의 필요성을 소개했습니다. 이러한 모델을 만드는 데 사용된 기술은 차원 모델링으로 알려졌습니다.

Bill Inmon 및 Ralph Kimball과 같은 선지자들의 저술 및 아이디어에 힘입어 차원 모델링은 Kimball의 저서 *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*에서 처음 소개되었습니다.² Kimball은 상향식 접근 방식에 초점을 맞춘 방법론을 정의하여 팀이 데이터 웨어하우스를 통해 가능한 한 빨리 실제 가치를 제공하도록 보장합니다.

차원 모델은 별 모양 스키마로 설명됩니다. 스타 스키마는 특정 비즈니스 프로세스(예: 판매)에 대한 데이터를 쉽게 분석할 수 있는 구조로 구성합니다. 두 가지 유형의 테이블로 구성됩니다.

- 스키마의 기본 또는 중앙 테이블인 사실 테이블. 팩트 테이블은 비즈니스 프로세스의 기본 측정, 메트릭 또는 "팩트"를 캡처합니다.
판매 비즈니스 프로세스를 예로 들면, 판매 팩트 테이블에는 판매 수량과 판매 금액이 포함됩니다.
 - 팩트 테이블에는 잘 정의된 그레인이 있습니다. 그레인은 표에 표시된 차원(열)의 조합에 따라 결정됩니다. 판매 팩트 테이블은 단지 연간 매출 룰업인 경우 낮은 세분성을 가질 수 있고, 날짜, 매장 및 고객 식별자별 매출을 포함하는 경우 높은 세분성을 가질 수 있습니다.
 - 팩트 테이블과 관련된 다중 차원 테이블. 차원은 선택한 비즈니스 프로세스를 둘러싼 컨텍스트를 제공합니다. 판매 시나리오 예에서 차원 목록에는 제품, 고객, 영업사원 및 기타 항목이 포함될 수 있습니다.
- 가게.

차원 테이블은 팩트 테이블을 "주위"하므로 이러한 유형의 스키마를 "스타 스키마"라고 합니다. 스타 스키마는 기본 및 외래 키 관계를 통해 연관된 차원 테이블에 연결된 사실 테이블로 구성됩니다. 판매 주제 영역에 대한 스타 스키마는 [그림 1-2에 나와 있습니다.](#)

2 킴볼, R. (1996). *데이터 웨어하우스 툴킷: 차원 모델링에 대한 전체 가이드*. 더 킴볼 그룹.

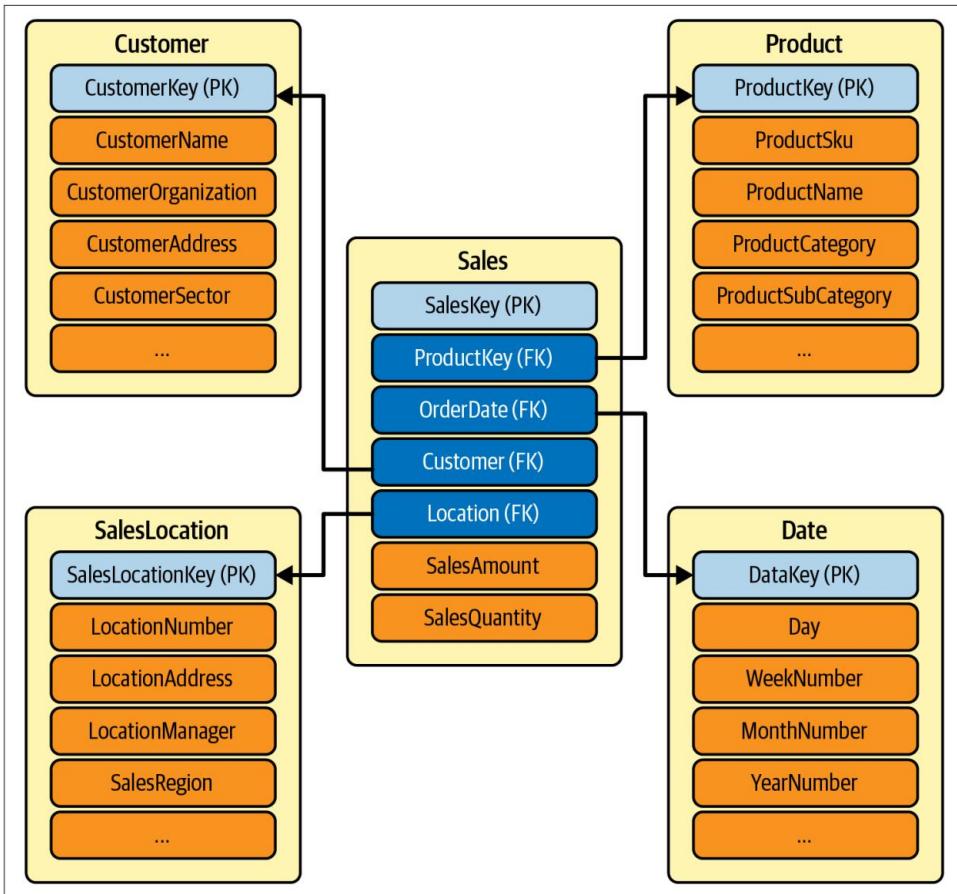


그림 1-2. 판매 차원 모델

데이터 웨어하우스의 이점과 과제

데이터 웨어하우스는 비즈니스 커뮤니티에 도움이 되는 고유한 장점을 가지고 있습니다. 다양한 데이터 소스에서 공통 형식으로 정제되고 정규화된 고품질 데이터를 제공합니다. 서로 다른 부서의 데이터가 공통된 형식으로 제공되므로 각 부서는 다른 부서와 연계하여 결과를 검토합니다. 시기적절하고 정확한 데이터를 확보하는 것은 강력한 비즈니스 결정의 기초입니다.

- 대량의 과거 데이터를 저장하므로 과거 통찰력을 제공하여 사용자가 다양한 기간과 추세를 분석할 수 있습니다. • 데이터 웨어하우스는 기본 관계형 기반을 기반으로 매우 안정적인 경향이 있습니다. ACID 트랜잭션을 실행하는 데이터베이스 기술입니다.

- 창고는 표준 스타 스키마 모델링 기법으로 모델링되어 팩트 테이블과 차원을 생성합니다. 영업, CRM 등 다양한 주제 영역에 사용할 수 있는 사전 구축된 템플릿 모델이 점점 더 많아지면서 이러한 모델 개발이 더욱 가속화되었습니다.

- 데이터 웨어하우스는 기본적으로 "무슨 일이 일어났는가?"라는 질문을 해결하는 비즈니스 인텔리전스 및 보고에 이상적으로 적합합니다. 데이터 성숙도 곡선에 대한 질문입니다.
비즈니스 인텔리전스(BI) 도구와 결합된 데이터 웨어하우스는 마케팅, 재무, 운영 및 영업에 대한 실행 가능한 통찰력을 생성할 수 있습니다.

인터넷과 소셜 미디어의 급속한 성장과 스마트폰과 같은 멀티미디어 장치의 가용성은 전통적인 데이터 환경을 파괴하면서 빅데이터라는 용어를 탄생시켰습니다. 빅 데이터는 더 많은 양, 더 빠른 속도, 더 다양한 형식으로 더 높은 진실성을 갖고 도착하는 데이터로 정의됩니다. 이는 데이터의 4가지 V로 알려져 있습니다.

볼륨 전 세

계적으로 생성, 캡처, 복사 및 소비되는 데이터의 양이 빠르게 증가하고 있습니다. Statista에 설명된 대로, 향후 2년 동안 글로벌 데이터 생성량은 200제타바이트 이상으로 증가할 것으로 예상됩니다(제타바이트는 2의 70승 바이트 수).

속도 오늘

날의 현대 비즈니스 환경에서는 시기적절한 결정이 매우 중요합니다. 이러한 결정을 내리기 위해 조직은 정보가 빠르게, 이상적으로는 최대한 실시간에 가깝게 전달되어야 합니다. 예를 들어, 주식 거래 애플리케이션은 고급 거래 알고리즘이 밀리초 단위의 결정을 내릴 수 있도록 거의 실시간 데이터에 액세스할 수 있어야 하며 이러한 결정을 이해관계자에게 전달해야 합니다. 적시에 데이터에 액세스하면 조직은 경쟁 우위를 확보할 수 있습니다.

다양성 다

양성은 현재 이용 가능한 데이터의 다양한 "유형"의 수를 나타냅니다.
전통적인 데이터 유형은 모두 구조화되어 있으며 일반적으로 관계형 데이터베이스 또는 그 추출로 제공됩니다. 빅데이터의 등장으로 이제 데이터는 새로운 비정형 유형으로 제공됩니다. 사물 인터넷(IoT) 장치 메시지, 텍스트, 오디오, 비디오와 같은 비정형 및 반정형 데이터 유형은 비즈니스 의미를 도출하기 위해 추가 전처리가 필요합니다. 다양성은 다양한 유형의 섭취를 통해 표현됩니다. 일부 데이터 원본은 배치 모드에서 가장 잘 수집되는 반면 다른 데이터 원본은 증분 수집 또는 IoT 데이터 스트림과 같은 실시간 이벤트 기반 수집에 적합합니다.

Veracity

Veracity는 데이터의 신뢰성을 정의합니다. 여기서 우리는 데이터가 정확하고 고품질인지 확인하고 싶습니다. 데이터는 여러 소스에서 수집될 수 있습니다. 데이터의 관리 연속성을 이해하는 것이 중요합니다.

우리는 풍부한 메타데이터를 보유하고 있으며 데이터가 수집된 맥락을 이해합니다. 또한 누락된 구성 요소나 늦게 도착한 사실 없이 데이터에 대한 완전한 뷰를 보장하고자 합니다.

데이터 웨어하우스는 이 네 가지 V를 해결하는 데 어려움을 겪습니다.

기존 데이터 웨어하우스 아키텍처는 기하급수적으로 증가하는 데이터 볼륨을 처리하는 데 어려움을 겪고 있습니다. 스토리지와 확장성 문제로 어려움을 겪고 있습니다. 볼륨이 페타바이트에 도달하면 많은 비용을 들이지 않고 스토리지 기능을 확장하기가 어려워집니다. 기존 데이터 웨어하우스 아키텍처는 인메모리 및 병렬 처리 기술을 사용하지 않으므로 데이터 웨어하우스를 수직적으로 확장할 수 없습니다.

데이터 웨어하우스 아키텍처는 빅 데이터의 속도를 처리하는 데에도 적합하지 않습니다. 데이터 웨어하우스는 거의 실시간 데이터를 지원하는 데 필요한 스트리밍 아키텍처 유형을 지원하지 않습니다. ETL 데이터 로드 창은 인프라가 불안정해질 때까지만 단축될 수 있습니다.

데이터 웨어하우스는 정형 데이터를 저장하는 데는 매우 적합하지만 다양한 반정형 또는 비정형 데이터를 저장하고 쿼리하는 데는 적합하지 않습니다.

데이터 웨어하우스에는 데이터의 신뢰성을 추적하는 기본 지원 기능이 없습니다. 데이터 웨어하우스 메타데이터는 주로 스키마에 초점을 맞추고 계보, 데이터 품질 및 기타 진실성 변수에는 덜 중점을 둡니다.

또한 데이터 웨어하우스는 폐쇄형 독점 형식을 기반으로 하며 일반적으로 SQL 기반 쿼리 도구만 지원합니다. 독점 형식으로 인해 데이터 웨어하우스는 데이터 과학 및 기계 학습 도구에 대한 적절한 지원을 제공하지 않습니다.

이러한 제한으로 인해 데이터 웨어하우스를 구축하는 데 비용이 많이 듭니다. 결과적으로, 프로젝트는 실행되기 전에 실패하는 경 우가 많으며, 실행되는 프로젝트는 현대 비즈니스 환경과 4가지 V의 끊임없이 변화하는 요구 사항을 따라가는 데 어려움을 겪습니다.

기존 데이터 웨어하우스 아키텍처의 한계로 인해 데이터 레이크 개념을 기반으로 하는 보다 현대적인 아키텍처가 탄생했습니다.

데이터 레이크 소개

데이터 레이크는 모든 규모의 정형, 반정형 또는 비정형 데이터를 파일 및 Blob 형태로 저장하는 비용 효율적인 종 앙 저장소입니다. "데이터 레이크"라는 용어는 물을 보유하고 있는 실제 강이나 호수, 또는 이 경우 실시간으로 호수에 물(일명 "데이터")을 흐르게 하는 여러 지류가 있는 데이터를 비유한 것에서 유래되었습니다. 일반적인 데이터 레이크의 표준 표현은 [그림 1-3에 나와 있습니다.](#)

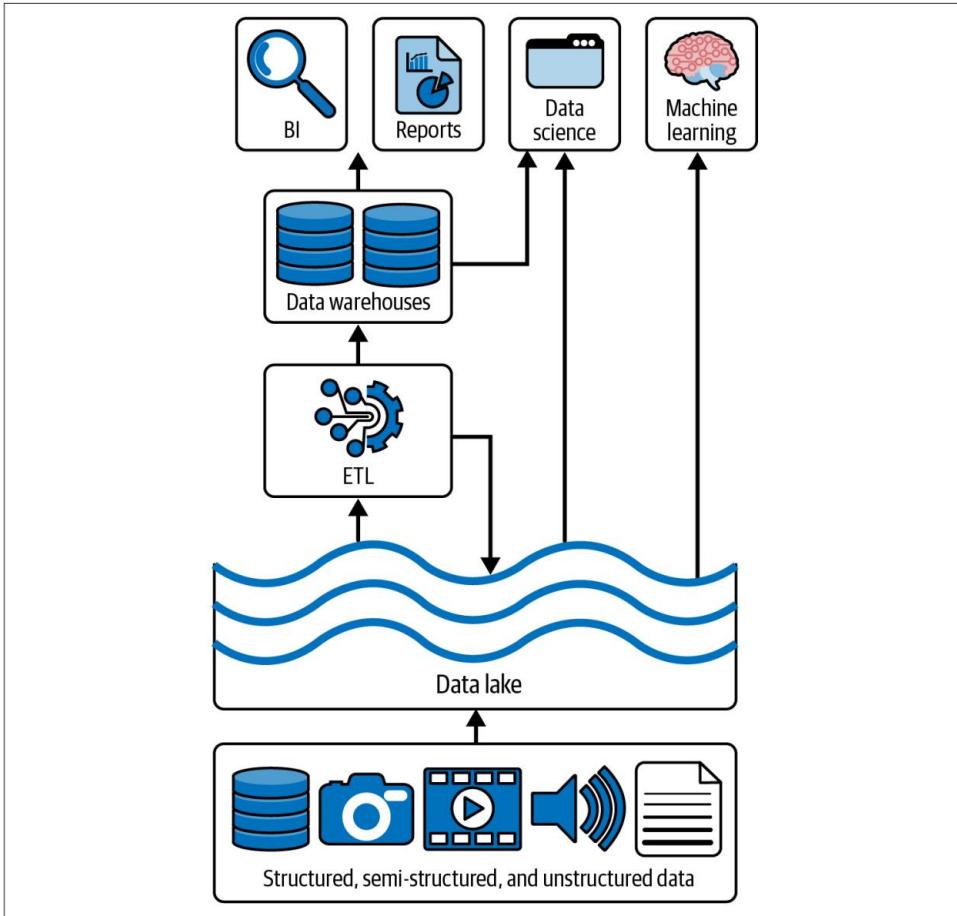


그림 1-3. 정식 데이터 레이크

초기 데이터 레이크와 빅 데이터 솔루션은 Apache Hadoop 오픈 소스 프레임워크 세트를 기반으로 하는 온프레미스 클러스터로 구축되었습니다. Hadoop은 기가바이트에서 페타바이트에 이르는 대규모 데이터 세트를 효율적으로 저장하고 처리하는 데 사용되었습니다. 하나의 대형 컴퓨터를 사용하여 데이터를 저장하고 처리하는 대신 Hadoop은 여러 상용 컴퓨팅 노드의 클러스터링을 활용하여 대량의 데이터 세트를 더 빠르게 병렬로 분석했습니다.

Hadoop은 MapReduce 프레임워크를 활용하여 여러 컴퓨팅 노드에 걸쳐 컴퓨팅 작업을 병렬화합니다. HDFS(Hadoop 분산 파일 시스템)는 표준 또는 저가형 하드웨어에서 실행되도록 설계된 파일 시스템이었습니다. HDFS는 내결함성이 매우 뛰어나고 대규모 데이터 세트를 지원했습니다.

2015년부터 Amazon Simple Storage Service(Amazon S3), Azure Data Lake Storage Gen 2(ADLS), Google Cloud Storage(GCS) 등 클라우드 데이터 레이크가

HDFS 교체를 시작했습니다. 이러한 클라우드 기반 스토리지 시스템은 뛰어난 서비스 수준 계약(SLA)(종종 10999999 이상)을 갖추고 지리적 복제를 제공하며, 가장 중요한 것은 보관을 위해 훨씬 저렴한 콜드 스토리지를 활용할 수 있는 옵션으로 매우 저렴한 비용을 제공한다는 것입니다. 목적.

가장 낮은 수준에서 데이터 레이크의 저장 단위는 데이터 덩어리입니다. Blob은 본질적으로 구조화되지 않았으므로 대용량 오디오 및 비디오 파일과 같은 반구조적 및 구조화되지 않은 데이터를 저장할 수 있습니다. 더 높은 수준에서 클라우드 스토리지 시스템은 Blob 스토리지 위에 파일 의미 체계와 파일 수준 보안을 제공하여 고도로 구조화된 데이터를 저장할 수 있도록 합니다. 높은 대역폭의 수신 및 송신 채널로 인해 데이터 레이크는 대량의 IoT 데이터 또는 스트리밍 미디어의 지속적인 수집과 같은 스트리밍 사용 사례도 가능하게 합니다.

컴퓨팅 엔진을 사용하면 ETL과 같은 방식으로 대량의 데이터를 처리하고 기존 데이터 웨어하우스, 기계 학습, AI 도구 세트와 같은 소비자에게 전달할 수 있습니다. 스트리밍 데이터는 실시간 데이터베이스에 저장될 수 있으며, 기존 BI 및 보고 도구를 사용하여 보고서를 생성할 수 있습니다.

데이터 레이크는 다양한 구성 요소를 통해 활성화됩니다.

스토리지

데이터 레이크에는 일반적으로 클라우드 환경에서 제공되는 것과 같은 매우 크고 확장 가능한 스토리지 시스템이 필요합니다. 스토리지는 내구성과 확장성이 있어야 하며 다양한 타사 도구, 라이브러리 및 드라이버와의 상호 운용성을 제공해야 합니다. 데이터 레이크는 스토리지와 컴퓨팅의 개념을 분리하여 둘 다 독립적으로 확장할 수 있습니다. 스토리지와 컴퓨팅을 독립적으로 확장하면 필요에 따라 탄력적으로 리소스를 미세 조정할 수 있으므로 솔루션 아키텍처가 더욱 유연해집니다. 스토리지 시스템에 대한 수신 및 송신 채널은 높은 대역폭을 지원하여 대규모 배치 볼륨의 수집 또는 소비 또는 IoT 및 스트리밍 미디어와 같은 대용량 스트리밍 데이터의 지속적인 흐름을 가능하게 해야 합니다.

컴퓨팅 스토리

지 계층에 저장된 대량의 데이터를 처리하려면 많은 양의 컴퓨팅 성능이 필요합니다. 다양한 클라우드 플랫폼에서 여러 컴퓨팅 엔진을 사용할 수 있습니다. 데이터 레이크에 가장 적합한 컴퓨팅 엔진은 Apache Spark입니다. Spark는 Databricks 또는 기타 클라우드 공급자가 개발한 솔루션과 같은 다양한 솔루션을 통해 배포할 수 있는 오픈 소스 통합 분석 엔진입니다. 빅 데이터 컴퓨팅 엔진은 컴퓨팅 클러스터를 활용합니다. 컴퓨팅 클러스터는 전체 데이터 수집 및 처리 작업을 처리하기 위해 컴퓨팅 노드를 풀링합니다.

형식 디스크

의 데이터 모양에 따라 형식이 정의됩니다. 다양한 저장 형식을 사용할 수 있습니다. 데이터 레이크는 주로 Parquet, Avro JSON 또는 CSV 와 같은 표준화된 오픈 소스 형식을 사용합니다.

메타데이터 층

신 클라우드 기반 스토리지 시스템은 메타데이터(즉, 데이터에 대한 상황별 정보)를 유지합니다. 여기에는 데이터가 작성되거나 액세스된 시기를 설명하는 다양한 타임스탬프, 데이터 스키마, 데이터 사용 및 소유자에 대한 정보가 포함된 다양한 태그 가 포함됩니다.

데이터 레이크에는 몇 가지 매우 강력한 이점이 있습니다. 데이터 레이크 아키텍처를 사용하면 조직의 데이터 자산을 하나의 중앙 위치로 통합할 수 있습니다. 데이터 레이크는 형식에 구애받지 않으며 Parquet 및 Avro와 같은 오픈 소스 형식을 사용합니다. 이러한 형식은 다양한 도구, 드라이버 및 라이브러리에서 잘 이해되므로 원활한 상호 운용이 가능합니다.

데이터 레이크는 성숙한 클라우드 스토리지 하위 시스템에 배포되므로 이러한 시스템과 관련된 확장성, 모니터링, 배포 용이성 및 낮은 스토리지 비용의 이점을 누릴 수 있습니다. Terraform과 같은 자동화된 DevOps 도구에는 잘 확립된 드라이버가 있어 자동화된 배포 및 유지 관리가 가능합니다.

데이터 웨어하우스와 달리 데이터 레이크는 반정형 및 비정형 데이터를 포함한 모든 데이터 유형을 지원하여 미디어 처리와 같은 워크로드를 지원합니다. 높은 처리량의 수신 채널로 인해 IoT 센서 데이터 수집, 미디어 스트리밍 또는 웹 클릭스트림과 같은 스트리밍 사용 사례에 매우 적합합니다.

그러나 데이터 레이크가 점점 대중화되고 널리 사용됨에 따라 조직은 기존 데이터 레이크의 몇 가지 과제를 인식하기 시작했습니다. 기본 클라우드 스토리지는 상대적으로 저렴하지만 효과적인 데이터 레이크를 구축하고 유지하려면 전문 기술이 필요하므로 고급 인력 배치 또는 컨설팅 서비스 증가가 필요합니다.

소송 비용.

원시 형식으로 데이터를 수집하는 것은 쉽지만 비즈니스 가치를 제공할 수 있는 형식으로 데이터를 변환하는 데는 비용이 많이 들 수 있습니다. 기존 데이터 레이크는 대기 시간 쿼리 성능이 낮기 때문에 대화형 쿼리에 사용할 수 없습니다. 결과적으로 조직의 데이터 팀은 데이터를 데이터 웨어하우스와 같은 것으로 변환하고 로드해야 하므로 가치 창출 시간이 연장됩니다. 그 결과 데이터 레이크 + 웨어하우스 아키텍처가 탄생했습니다. 이 아키텍처는 꽤 오랫동안 업계를 지배했지만(우리는 개인적으로 이러한 유형의 시스템 을 수십 개 구현했습니다) 이제는 레이크하우스의 증가로 인해 감소하고 있습니다.

데이터 레이크는 일반적으로 스키마 적용 없이 모든 형식으로 데이터를 수집할 수 있는 "읽기 시 스키마" 전략을 사용합니다. 데이터를 읽을 때만 일부 유형의 스키마를 적용할 수 있습니다. 이러한 스키마 적용 부족으로 인해 데이터 품질 문제가 발생하여 원시 데이터 레이크가 "데이터 늪"이 될 수 있습니다.

데이터 레이크는 어떠한 종류의 거래 보장도 제공하지 않습니다. 데이터 파일은 추가만 가능하므로 간단한 업데이트를 위해 이전에 작성된 데이터를 다시 작성하는 데 비용이 많이 듭니다. 이로 인해 단일 엔터티에 대해 여러 개의 작은 파일이 생성되는 "작은 파일 문제"라는 문제가 발생합니다. 이 문제를 잘 관리하지 않으면 이러한 작은 파일로 인해 전체 데이터 레이크의 읽기 성능이 저하되어 데이터가 부실해지고 스토리지가 낭비됩니다. 데이터 레이크 관리자는 반복적인 작업을 실행하여 이러한 작은 파일을 효율적인 읽기 작업에 최적화된 큰 파일로 통합해야 합니다.

이제 데이터 웨어하우스와 데이터 레이크의 장단점을 논의했으니, 두 기술의 장점을 결합하고 약점을 해결하는 데이터 레이크하우스를 소개하겠습니다.

데이터 레이크하우스

Armbrust, Ghodsi, Xin, Zaharia는 2021년에 데이터 레이크하우스 개념을 처음 도입했습니다. 저자는 레이크하우스를 "ACID 트랜잭션, 데이터 버전 관리, 감사, 인덱싱, 캐싱 및 쿼리 최적화와 같은 분석 DBMS 관리 및 성능 기능도 제공하는 저비용의 직접 액세스 가능한 스토리지를 기반으로 하는 데이터 관리 시스템"으로 정의합니다.

이 설명을 풀면 레이크하우스를 데이터 레이크의 유연성, 저비용 및 규모를 데이터 웨어하우스의 데이터 관리 및 ACID 트랜잭션과 병합하여 두 가지의 한계를 해결하는 시스템으로 정의할 수 있습니다. 데이터 레이크와 마찬가지로 레이크하우스 아키텍처는 해당 시스템의 고유한 유연성과 수평 확장성을 갖춘 저비용 클라우드 스토리지 시스템을 활용합니다. 레이크하우스의 목표는 Parquet와 같은 기존 고성능 데이터 형식을 사용하는 동시에 ACID 트랜잭션(및 기타 기능)을 활성화하는 것입니다. 이러한 기능을 추가하기 위해 레이크하우스는 ACID 트랜잭션, 레코드 수준 작업, 인덱싱 및 주요 메타데이터와 같은 기능을 기존 데이터 형식에 추가하는 개방형 테이블 형식을 사용합니다. 이를 통해 저비용 스토리지 시스템에 저장된 데이터 자산은 RDBMS 도메인에만 국한되었던 것과 동일한 신뢰성을 가질 수 있습니다. Delta Lake는 이러한 유형의 기능을 지원하는 개방형 테이블 형식의 예입니다.

레이크하우스는 별도의 컴퓨팅 및 스토리지 리소스를 갖춘 대부분의 클라우드 환경에 특히 적합합니다. 동일한 스토리지 데이터에 직접 액세스하면서 Spark 클러스터와 같이 완전히 별도의 컴퓨팅 노드에서 다양한 컴퓨팅 애플리케이션을 요청 시 실행할 수 있습니다. 그러나 앞서 언급한 HDFS와 같은 온프레미스 스토리지 시스템을 통해 레이크하우스를 구현할 수도 있습니다.

데이터 레이크하우스의 이점

레이크하우스 아키텍처의 개요는 **그림 1-4에 나와 있습니다.**

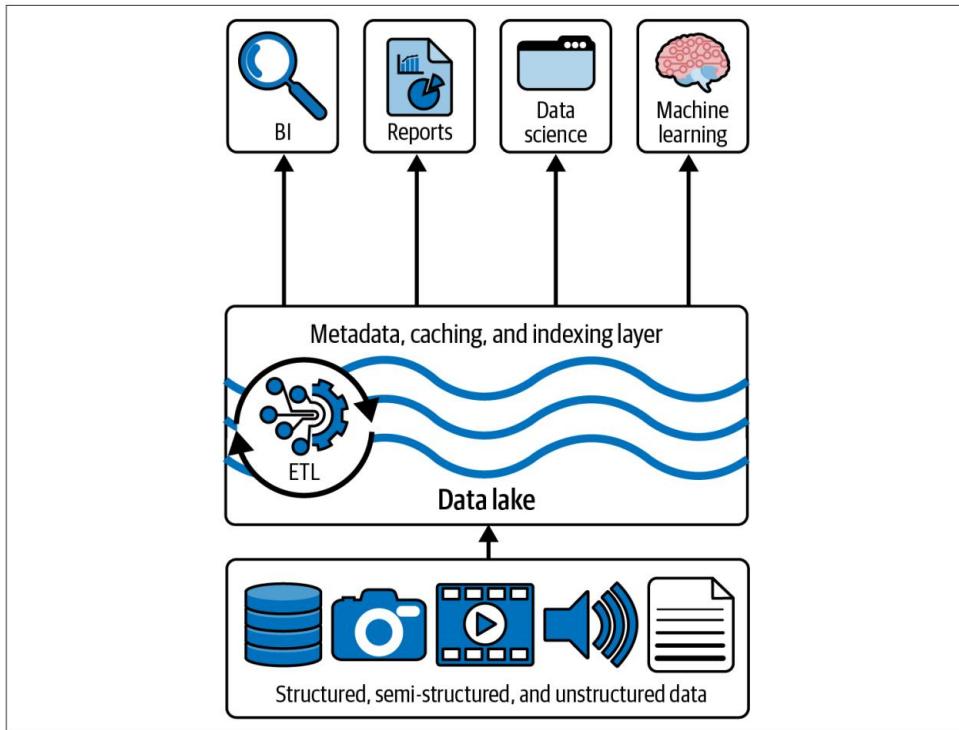


그림 1-4. 레이크하우스 아키텍처 개요

레이크하우스 아키텍처를 사용하면 더 이상 데이터 레이크에 데이터 사본을 보관하고 일부 유형의 데이터 웨어하우스 스토리지에 또 다른 사본을 보관할 필요가 없습니다. 실제로 기존 데이터 웨어하우스와 비슷한 성능을 갖춘 Delta Lake 스토리지 형식 및 프로토콜을 통해 데이터 레이크에서 데이터를 소싱할 수 있습니다.

저렴한 클라우드 기반 스토리지 기술을 계속 활용할 수 있고 더 이상 데이터 레이크에서 데이터 웨어하우스로 데이터를 복사할 필요가 없으므로 인프라와 직원 및 컨설팅 오버헤드 측면에서 상당한 비용 절감을 실현할 수 있습니다.

데이터 이동이 줄어들고 ETL이 단순화되므로 데이터 품질 문제가 발생할 가능성이 크게 줄어들고, 마지막으로 레이크하우스에 대용량 데이터 저장 기능과 정제된 차원 모델이 결합되어 개발 주기가 단축되고 가치 창출 시간이 단축됩니다. 대폭 감소되었습니다.

데이터 웨어하우스에서 데이터 레이크, 레이크하우스 아키텍처로의 진화가 [그림 1-5에](#) 나와 있습니다.

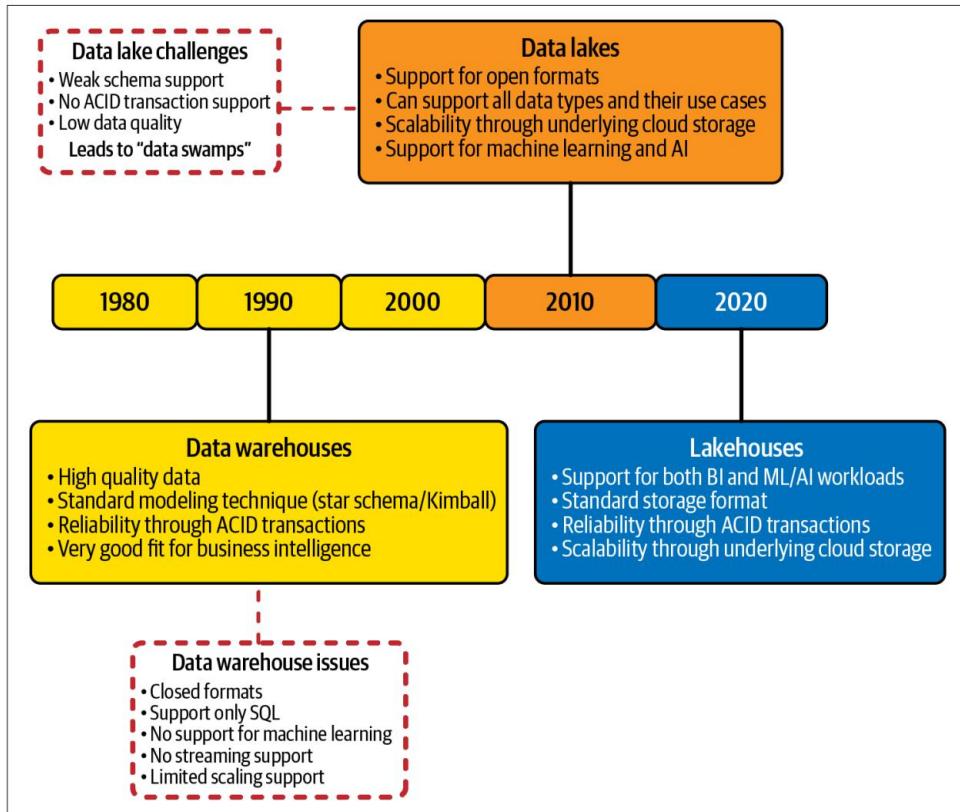


그림 1-5. 데이터 아키텍처의 진화

앞서 언급했듯이 레이크 하우스는

Amazon S3, ADLS 또는 GCS와 같은 저비용 객체 저장소를 활용하여 Apache Parquet와 같은 오픈 소스 테이블 형식으로 데이터를 저장합니다. 그러나 레이크하우스 구현은 이 데이터에 대해 ACID 트랜잭션을 실행하므로 클라우드 스토리지 위에 트랜잭션 메타데이터 계층이 필요하여 테이블 버전의 일부인 객체를 정의합니다.

이를 통해 레이크하우스는 해당 메타데이터 계층 내에서 ACID 트랜잭션 및 버전 관리와 같은 기능을 구현하는 동시에 대량의 데이터를 저비용 객체 스토리지에 보관할 수 있습니다. 레이크하우스 클라이언트는 이미 익숙한 오픈 소스 형식의 데이터를 계속 사용할 수 있습니다.

다음으로 시스템 성능을 다루어야 합니다. 앞서 언급했듯이 레이크하우스 구현이 효과적이려면 뛰어난 SQL 성능을 달성해야 합니다. 데이터 웨어하우스는 폐쇄형 스토리지 형식과 잘 알려진 스키마로 작업하기 때문에 성능 최적화에 매우 능숙했습니다. 이를 통해 통계를 유지하고, 클러스터형 인덱스를 구축하고, 핫 데이터를 빠른 SSD 장치로 이동하는 등의 작업을 수행할 수 있었습니다.

오픈 소스 표준 형식을 기반으로 하는 레이크하우스에서는 저장 형식을 변경할 수 없기 때문에 그러한 여유가 없습니다. 그러나 레이크하우스는 데이터 파일을 변경하지 않고 그대로 유지하는 다양한 다른 최적화를 활용할 수 있습니다. 여기에는 캐싱, 인덱스 및 통계와 같은 보조 데이터 구조, 데이터 레이아웃 최적화가 포함됩니다.

분석 워크로드 속도를 높일 수 있는 마지막 도구는 표준 DataFrame API의 개발입니다. TensorFlow 및 Spark MLlib와 같이 널리 사용되는 대부분의 ML 도구는 DataFrames를 지원합니다. DataFrame은 R과 pandas에 의해 처음 도입되었으며 대부분 관계형 대수에서 비롯된 다양한 변환 작업을 통해 데이터의 간단한 테이블 추상화를 제공합니다.

Spark에서 DataFrame API는 선언적이며 지연 평가를 사용하여 실행 DAG(방향성 비순환 그래프)를 구축합니다. 그러면 어떤 작업이 DataFrame의 기본 데이터를 사용하기 전에 이 그래프를 최적화할 수 있습니다. 이를 통해 레이크하우스에는 캐싱 및 보조 데이터와 같은 몇 가지 새로운 최적화 기능이 제공됩니다. [그림 1-6은](#) 이러한 요구 사항이 전체 레이크하우스 시스템 설계에 어떻게 적용되는지 보여줍니다.

Delta Lake가 이 책의 초점이므로 Delta Lake가 레이크하우스 구현 요구 사항을 어떻게 촉진하는지 설명하겠습니다.

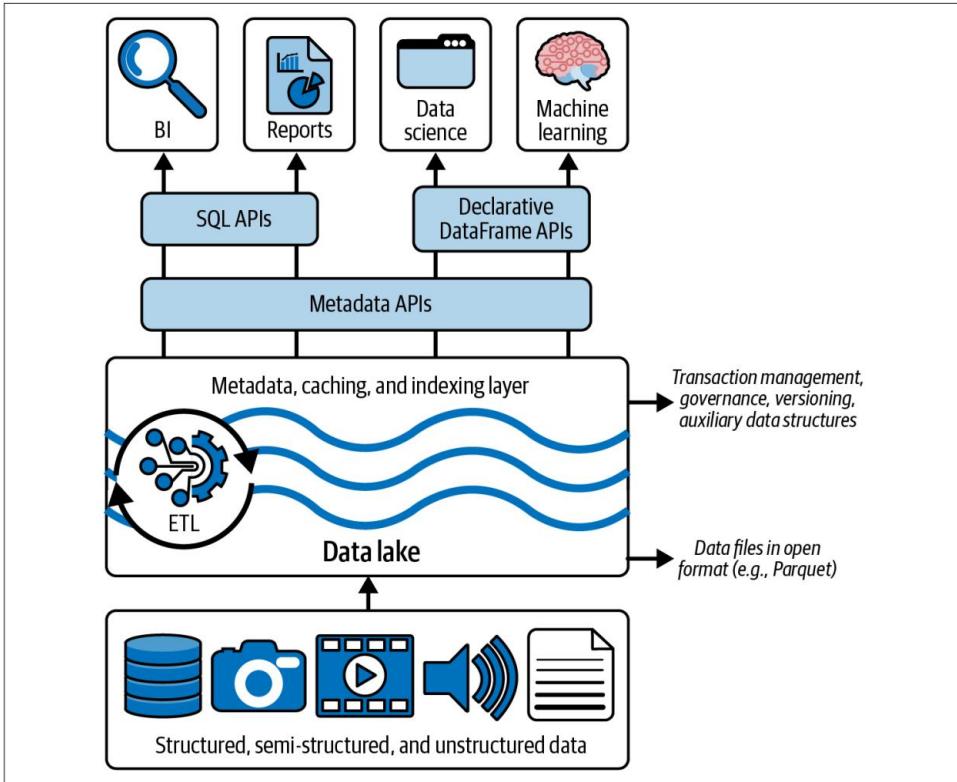


그림 1-6. 레이크하우스 구현

델타 레이크

이전 섹션에서 언급했듯이 **Delta Lake** 위에 가능한 레이크하우스 솔루션을 구축할 수 있습니다. Delta Lake는 메타데이터, 캐싱 및 인덱싱을 데이터 레이크 스토리지 형식과 결합한 개방형 테이블 형식입니다. 이는 ACID 트랜잭션 및 기타 관리 기능을 제공하기 위한 추상화 수준을 제공합니다.

Delta Lake 오픈 테이블 형식, 오픈 소스 메타데이터 레이어는 궁극적으로 레이크하우스 구현을 가능하게 합니다. Delta Lake는 ACID 트랜잭션, 확장 가능한 메타데이터 처리, 배치 및 스트리밍을 포함하는 통합 프로세스 모델, 전체 감사 기록, SQL 데이터 조작 언어(DML) 문 지원을 제공합니다. 기존 데이터 레이크에서 실행될 수 있으며 Apache Spark를 포함한 여러 처리 엔진과 완벽하게 호환됩니다.

Delta Lake는 오픈 소스 프레임워크이며, 사양은 <https://delta.io>에서 확인할 수 있습니다. **Armbrust** 등 의 연구 . Delta Lake가 ACID 트랜잭션을 제공하는 방법에 대한 자세한 설명을 제공합니다.

Delta Lake는 다음 기능을 제공합니다.

트랜잭션 ACID는 Delta Lake가 Spark

또는 기타 처리 엔진을 사용하는 모든 데이터 레이크 트랜잭션이 내구성을 위해 커밋되고 원자 방식으로 다른 판독기에 노출되도록 보장합니다. 이는 Delta 트랜잭션 로그를 통해 가능합니다. 2 장 에서는 트랜잭션 로그에 대해 자세히 다루겠습니다.

전체 DML 지원 기준 데

이터 레이크는 데이터에 대한 트랜잭션 원자적 업데이트를 지원하지 않습니다.

Delta Lake는 삭제 및 업데이트, 복잡한 데이터 병합 또는 upsert 시나리오를 포함한 모든 DML 작업을 완벽하게 지원합니다. 이 지원은 최신 데이터 웨어하우스(MDW)를 구축할 때 차원 및 팩트 테이블 생성을 크게 단순화합니다.

감사 기록 Delta

Lake 트랜잭션 로그는 데이터에 대한 모든 변경 사항을 변경된 순서대로 기록합니다. 따라서 트랜잭션 로그는 데이터 변경 사항에 대한 전체 감사 추적이 됩니다. 이를 통해 관리자와 개발자는 실수로 데이터를 삭제하거나 편집한 후 이전 버전의 데이터로 롤백할 수 있습니다. 이 기능을 시간 여행이라고 하며 6 장 에서 자세히 설명합니다.

배치 및 스트리밍을 하나의 처리 모델로 통합 Delta Lake는 배치 및 스트리밍

싱크 또는 소스와 함께 작동할 수 있습니다. IoT 데이터를 장치 참조 데이터와 병합할 때 일반적인 요구 사항인 데이터 스트림에서 MERGE를 수행할 수 있습니다. 또한 CDC를 수신하는 사용 사례도 가능합니다.

외부 데이터 소스의 데이터. 스트리밍에 대해서는 8 장 에서 자세히 다룰 것입니다.

스키마 적용 및 발전 Delta Lake는 레이크에

서 데이터를 쓰거나 읽을 때 스키마를 적용합니다.

그러나 데이터 엔터티에 대해 명시적으로 활성화되면 안전한 스키마 진화가 가능해 데이터가 진화해야 하는 사용 사례가 가능해집니다. 스키마 시행과 발전은 7 장 에서 다룹니다.

풍부한 메타데이터 지원 및 확장성 대용량 데

이터를 지원하는 능력은 훌륭하지만 메타데이터가 그에 맞게 확장되지 않으면 솔루션이 부족합니다. Delta Lake는 Spark 또는 기타 컴퓨팅 엔진을 활용하여 모든 메타데이터 처리 작업을 확장하므로 페타바이트 규모의 데이터에 대한 메타데이터를 효율적으로 처리할 수 있습니다.

레이크하우스 아키텍처는 그림 1-7 과 같이 세 개의 레이어로 구성됩니다. 레이크하우스 스토리지 계층은 ADLS, GCS 또는 Amazon S3 스토리지와 같은 표준 클라우드 스토리지 기술을 기반으로 구축되었습니다. 이는 레이크하우스에 확장성이 뛰어나고 저렴한 스토리지 레이어를 제공합니다.

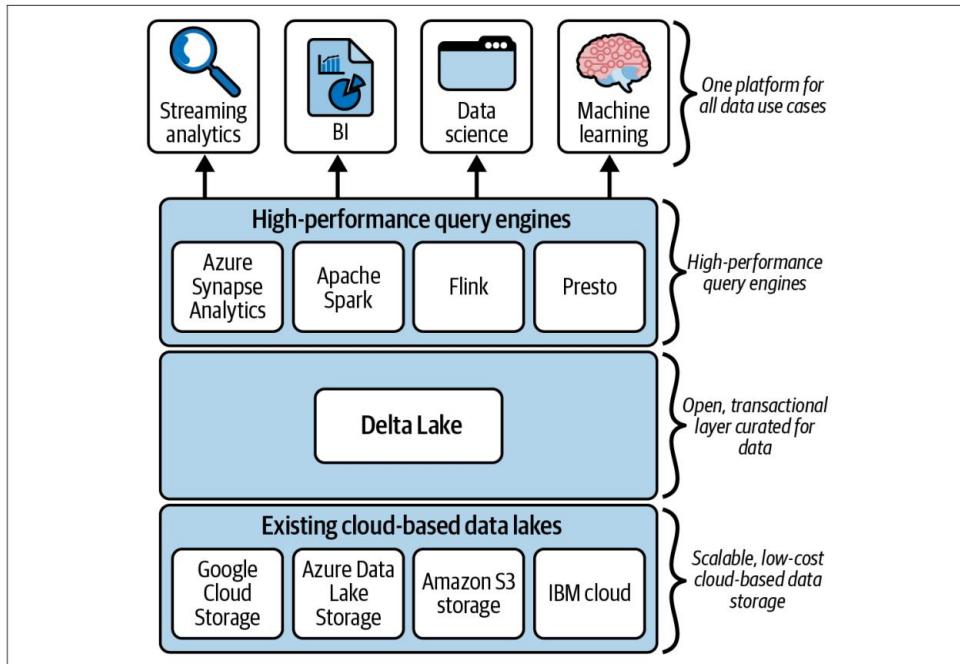


그림 1-7. 레이크하우스 계층형 아키텍처

레이크하우스의 트랜잭션 계층은 Delta Lake에서 제공됩니다. 이를 통해 레이크하우스에 ACID 보장이 제공되어 원시 데이터를 선별되고 실행 가능한 데이터로 효율적으로 변환할 수 있습니다. ACID 지원 외에도 Delta Lake는 DML 지원, 확장 가능한 메타데이터 처리, 스트리밍 지원 및 풍부한 검사 로그와 같은 다양한 추가 기능 세트를 제공합니다. 레이크 하우스 스택의 최상위 계층은 기본 클라우드 컴퓨팅 리소스를 활용하는 고성능 쿼리 및 처리 엔진으로 구성됩니다. 지원되는 쿼리 엔진은 다음과 같습니다.

- 아파치 스파크 • 아
- 파치 하이브
- 프레스토
- 트리노

Delta Lake 웹사이트를 참조하세요. 지원되는 컴퓨팅 엔진의 전체 목록을 확인하세요.

메달리온 아키텍처

Delta Lake 기반 레이크하우스 아키텍처의 예가 [그림 1-8에 나와 있습니다.](#)

브론즈, 실버, 골드 레이어로 구성된 이 아키텍처 패턴을 흔히 메달리온 아키텍처라고 합니다. 이는 많은 레이크하우스 아키텍처 패턴 중 하나일 뿐이지만 최신 데이터 웨어하우스, 데이터 마트 및 기타 분석 솔루션에 매우 적합합니다.

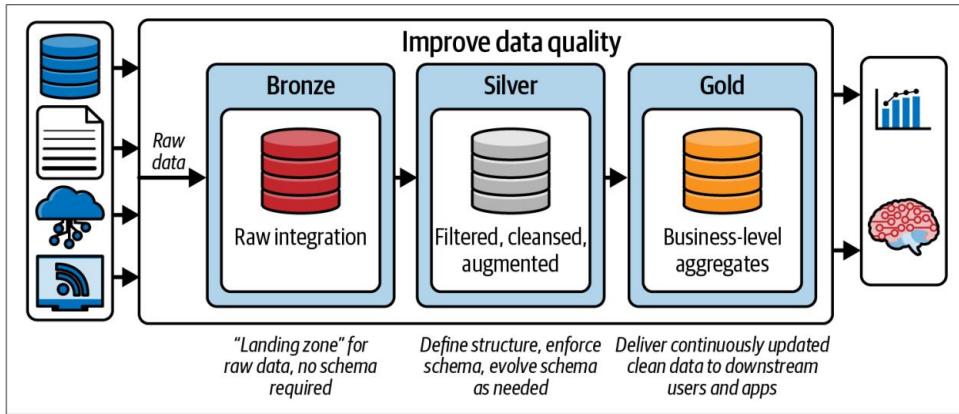


그림 1-8. 데이터 레이크하우스 솔루션 아키텍처

가장 높은 수준에서 이 솔루션에는 세 가지 구성 요소가 있습니다. 왼쪽에는 다양한 데이터 소스가 있습니다. 데이터 소스는 다양한 형태를 취할 수 있습니다. 여기에 몇 가지 예가 나와 있습니다.

- 외부 데이터 레이크에 있는 CSV 또는 TXT 파일 세트
- Oracle 또는 SQL Server와 같은 온프레미스 데이터베이스 • Kafka 또는 Azure Event Hubs와 같은 스트리밍 데이터 소스 • Salesforce 또는 ADP와 같은 SAS 서비스의 REST API

중앙 구성 요소는 메달리온 아키텍처를 구현합니다. 메달리온 아키텍처는 레이크하우스의 데이터를 브론즈, 실버, 골드 레이어를 통해 논리적으로 구성하는 데 사용되는 데이터 디자인 패턴입니다. 브론즈 계층은 소스 시스템에서 수집된 데이터를 왼쪽에 배치하는 곳입니다. 브론즈 영역의 데이터는 일반적으로 "있는 그대로" 제공되지만 로드 날짜 및 시간, 처리 식별자 등과 같은 추가 메타데이터로 확장될 수 있습니다.

Silver 레이어에서는 Bronze 레이어의 데이터가 정리, 정규화, 병합 및 일치됩니다. 다양한 주제 영역의 데이터에 대한 기업의 관점이 점차 통합되는 곳입니다.

Gold 레이어의 데이터는 "소비 준비" 데이터입니다. 이 데이터는 차원과 팩트 테이블을 포함하는 클래식 스타 스키마 형식일 수도 있고 소비 사용 사례에 적합한 모든 데이터 모델일 수도 있습니다.

메달리온 아키텍처의 목표는 데이터가 아키텍처의 각 계층을 통해 흐르면서 각 계층에 고유한 목적이 있으므로 데이터의 구조와 품질을 점진적이고 점진적으로 개선하는 것입니다. 이 데이터 디자인 패턴은 [10 장](#)에서 훨씬 더 깊이 다루겠지만 Delta Lake와 함께 레이크하우스가 어떻게 안정적이고 성능이 뛰어난 데이터 디자인 패턴 또는 멀티홀 아키텍처를 지원할 수 있는지 이해하는 것이 중요합니다. 메달리온 아키텍처와 같은 디자인 패턴은 여러 사용 사례(예: 배치 데이터, 스트리밍 데이터 및 기계 학습)를 지원하기 위해 레이크하우스에서 데이터 파이프라인을 통합하기 위한 일부 아키텍처 기반을 제공합니다.

델타 생태계

이 장에서 설명한 대로 Delta Lake를 사용하면 데이터 웨어하우징 및 기계 학습/AI 애플리케이션을 데이터 레이크에서 직접 호스팅할 수 있는 데이터 레이크하우스 아키텍처를 구축할 수 있습니다. 현재 Delta Lake는 가장 널리 사용되는 레이크하우스 형식으로, 현재 7,000개 이상의 조직에서 사용되며 하루에 엑사바이트의 데이터를 처리합니다.

그러나 데이터 웨어하우스와 기계 학습 애플리케이션이 Delta Lake의 유일한 애플리케이션 대상은 아닙니다. 데이터 레이크에 안정성을 제공하는 핵심 트랜잭션 ACID 지원 외에도 Delta Lake를 사용하면 하나님의 솔루션 아키텍처로 스트리밍 및 배치 데이터를 모두 원활하게 수집하고 사용할 수 있습니다.

Delta Lake의 또 다른 중요한 구성 요소는 기업이 안전한 방식으로 데이터 세트를 공유할 수 있게 해주는 Delta Sharing입니다. 이제 Delta Lake 3.0은 독립 실행형 판독기와 기록기를 지원하므로 모든 클라이언트(Python, Ruby 또는 Rust)가 Spark 또는 Flink와 같은 빅 데이터 엔진 없이도 Delta Lake에 직접 데이터를 쓸 수 있습니다. Delta Lake는 Presto, Flink 및 Trino를 포함한 확장된 오픈 소스 커넥터 세트와 함께 제공됩니다. Delta Lake 스토리지 계층은 이제 ADLS, Amazon S3 및 GCS를 포함한 많은 플랫폼에서 광범위하게 사용됩니다. Delta Lake 2.0의 모든 구성 요소는 Databricks에서 오픈 소스로 제공됩니다.

Delta Lake와 레이크하우스의 성공으로 Delta 기술을 기반으로 구축된 완전히 새로운 생태계가 탄생했습니다. 이 생태계는 Delta Lake 스토리지 형식, Delta Sharing 및 Delta Connector를 포함한 다양한 개별 구성 요소로 구성됩니다.

Delta Lake 스토리지

Delta Lake 스토리지 형식은 클라우드 기반 데이터 레이크 위에서 실행되는 오픈 소스 스토리지 계층입니다. 데이터 레이크 파일 및 테이블에 트랜잭션 기능을 추가합니다.

데이터 웨어하우스와 유사한 기능을 표준 데이터 레이크에 도입합니다. Delta Lake 스토리지는 다른 모든 구성 요소가 이 계층에 의존하기 때문에 에코시스템의 핵심 구성 요소입니다.

델타 공유 데이터 공유

는 비즈니스에서 일반적인 사용 사례입니다. 예를 들어, 광산 회사는 예방적 유지 관리 및 진단 목적으로 대규모 광산 트럭 엔진의 IoT 정보를 제조업체와 안전하게 공유하기를 원할 수 있습니다. 온도 조절기 제조업체는 사용량이 많은 날 전력망 부하를 최적화하기 위해 HVAC 데이터를 공공 유틸리티와 안전하게 공유하기를 원할 수 있습니다. 그러나 과거에는 안전하고 안정적인 데이터 공유 솔루션을 구현하는 것이 매우 어려웠고 비용이 많이 드는 맞춤형 개발이 필요했습니다.

Delta Sharing은 Delta Lake 데이터의 대규모 데이터 세트를 안전하게 공유하기 위한 오픈 소스 프로토콜입니다. 이를 통해 사용자는 Amazon S3, ADLS 또는 GCS에 저장된 데이터를 안전하게 공유할 수 있습니다.

Delta Sharing을 사용하면 사용자는 추가 구성 요소를 배포할 필요 없이 Spark, Rust, Power BI 등과 같이 선호하는 도구 세트를 사용하여 공유 데이터에 직접 연결할 수 있습니다. 맞춤 개발 없이 클라우드 제공업체 간에 데이터를 공유할 수 있습니다.

Delta Sharing은 다음과 같은 사용 사례를 가능하게 합니다.

- ADLS에 저장된 데이터는 AWS의 Spark 엔진에서 처리될 수 있습니다. • Amazon S3에 저장된 데이터는 GCP의 Rust로 처리할 수 있습니다.

델타 공유에 대한 자세한 내용은 [9장](#)을 참조하십시오 .

델타 커넥터

Delta Connectors^{3,4}의 주요 목표는 Delta Lake를 Apache Spark 외부의 다른 빅 데이터 엔진에 적용하는 것입니다. Delta 커넥터는 Delta Lake에 직접 연결되는 오픈 소스 커넥터입니다. 프레임워크에는 Apache Spark 클러스터 없이도 Delta Lake 테이블을 직접 읽고 쓸 수 있는 Java 기본 라이브러리인 Delta Standalone이 포함되어 있습니다. 소비 애플리케이션은 Delta Standalone을 사용하여 빅 데이터 인프라에서 작성된 Delta 테이블에 직접 연결할 수 있습니다. 이렇게 하면 데이터를 사용하기 전에 다른 형식으로 데이터를 복제할 필요가 없습니다.

³개의 Delta Lake 통합

델타 Lake 커넥터 4개

다음과 같은 경우 다른 기본 라이브러리를 사용할 수 있습니다.

하이브

Hive 커넥터는 Apache Hive에서 직접 델타 테이블을 읽습니다.

Flink

Flink/Delta 커넥터는 Apache Flink 애플리케이션에서 Delta 테이블을 읽고 씁니다. 커넥터에는 Apache Flink에서 Delta 테이블에 쓰기 위한 싱크와 Flink를 사용하여 Delta 테이블을 읽기 위한 소스가 포함되어 있습니다.

sql-delta-import 이

커넥터를 사용하면 JDBC 데이터 소스의 데이터를 직접 Delta 테이블로 가져올 수 있습니다.

파워 BI

Power BI 커넥터는 Power BI 커넥터를 허용하는 사용자 지정 파워 쿼리 기능입니다.

Microsoft에서 지원하는 모든 파일 기반 데이터 소스에서 델타 테이블을 읽는 BI
파워 BI.

Delta Connectors는 빠르게 성장하는 생태계로, 정기적으로 더 많은 커넥터를 사용할 수 있게 됩니다. 실제로 최근 발표된 Delta Lake 3.0 릴리스에는 [Delta Kernel이 포함되어 있습니다](#). Delta 커널과 단순화된 라이브러리를 사용하면 Delta 프로토콜 세부 정보를 이해할 필요가 없으므로 커넥터를 구축하고 유지 관리하는 것이 훨씬 쉬워집니다.

결론

데이터의 양, 속도, 다양성 및 진실성을 고려할 때 데이터 웨어하우스와 데이터 레이크의 한계와 과제는 데이터 아키텍처의 새로운 패러다임을 주도했습니다. Delta Lake와 같은 개방형 테이블 형식의 발전을 통해 제시된 레이크하우스 아키텍처는 이전 버전의 최고의 요소를 활용하여 데이터 플랫폼에 대한 통합 접근 방식을 제공하는 현대적인 데이터 아키텍처를 제공합니다.

[서문](#)에서 언급했듯이 이 책은 표면적인 내용을 다루는 것 이상의 역할을 할 것입니다. 이 장에서 이미 다룬 Delta Lake의 핵심 기능 중 일부를 자세히 살펴보겠습니다. Delta Lake를 가장 잘 설정하는 방법, 다양한 기능의 사용 사례를 식별하는 방법, 모범 사례 및 고려해야 할 다양한 사항 등에 대해 알아봅니다.

이는 데이터 실무자에게 이 개방형 테이블 형식이 레이크하우스 아키텍처 형태의 최신 데이터 플랫폼을 어떻게 지원하는지에 대한 맥락과 증거를 지속적으로 제공할 것입니다. 이 책이 끝나면 Delta Lake를 시작 및 실행하고 최신 데이터 레이크하우스 아키텍처를 구축하는 데 자신감을 갖게 될 것입니다.

제 2 장

Delta Lake 시작하기

이장에서는 Delta Lake를 소개하고 이것이 기존 데이터 레이크에 트랜잭션 보장, DML 지원, 감사, 통합 스트리밍 및 배치 모델, 스카마 적용, 확장 가능한 메타데이터 모델을 추가하는 방법을 살펴보겠습니다.

이장에서는 Delta Lake를 직접 살펴보겠습니다. 먼저 Spark가 설치된 로컬 머신에 Delta Lake를 설정하겠습니다. 두 개의 대화형 셀에서 Delta Lake 샘플을 실행합니다.

1. 먼저 Delta Lake 패키지를 사용하여 PySpark 대화형 셀을 실행합니다. 이를 통해 델타 테이블을 생성하는 간단한 두 줄 Python 프로그램을 입력하고 실행할 수 있습니다.
2. 다음으로 Spark Scala 셀을 사용하여 유사한 프로그램을 실행하겠습니다. 이 책에서 Scala 언어를 광범위하게 다루지는 않지만 Spark 셀과 Scala가 모두 Delta Lake의 옵션임을 보여주고 싶습니다.

다음으로, 즐겨 사용하는 편집기 내에서 Python으로 helloDeltaLake 시작 프로그램을 만들고 PySpark 셀에서 대화형으로 프로그램을 실행합니다. 이 장에서 설정한 환경과 helloDeltaLake 프로그램은 이 책에서 만드는 대부분의 다른 프로그램의 기초가 될 것입니다.

환경이 준비되어 실행되면 델타 테이블 형식을 더 자세히 살펴볼 준비가 됩니다. Delta Lake는 Parquet를 기본 저장 매체로 사용하므로 먼저 Parquet 형식을 간략하게 살펴보겠습니다. 나중에 트랜잭션 로그를 연구할 때 파티션과 파티션 파일이 중요한 역할을 하기 때문에 자동 및 수동 파티셔닝 메커니즘을 모두 연구하겠습니다. 다음으로, 델타 테이블로 이동하여 델타 테이블이 _delta_log 디렉터리에 트랜잭션 로그를 추가하는 방법을 조사합니다.

이 장의 나머지 부분은 트랜잭션 로그에 대해 다룹니다. 트랜잭션 로그 항목의 세부 사항을 조사하기 위해 여러 Python 프로그램을 만들고 실행합니다.

어떤 종류의 작업이 기록되며, 어떤 Parquet 데이터 파일이 트랜잭션 로그 항목과 언제, 어떻게 관련되는지 기록됩니다. 보다 복잡한 업데이트 예와 이것이 트랜잭션 로그에 미치는 영향을 살펴보겠습니다. 마지막으로 검사점 파일의 개념과 이것이 Delta Lake가 확장 가능한 메타데이터 시스템을 구현하는 데 어떻게 도움이 되는지 소개하겠습니다.

표준 Spark 이미지 가져오기

로컬 컴퓨터에 Spark를 설정하는 것은 어려울 수 있습니다. 다양한 설정, 업데이트 패키지 등을 조정해야 합니다. 따라서 우리는 Docker 컨테이너를 사용하기로 결정했습니다. Docker가 설치되어 있지 않은 경우 해당 [웹 사이트에서 무료로 다운로드할 수 있습니다](#). 우리가 사용한 특정 컨테이너는 표준 [Apache Spark 이미지였습니다](#).

이미지를 다운로드하려면 다음 명령을 사용할 수 있습니다.

도커 풀 아파치/스파크

이미지를 풀다운한 후에는 다음 명령을 사용하여 컨테이너를 시작할 수 있습니다.

```
docker run -it apache/spark /bin/sh
```

Spark 설치는 /opt/spark 디렉터리에 있습니다. PySpark, Spark-sql 및 기타 모든 도구는 /opt/spark/bin 디렉터리에 있습니다. 우리는 책의 [GitHub 저장소](#)의 추가 정보에 컨테이너 작업 방법에 대한 몇 가지 지침을 포함했습니다.

PySpark와 함께 Delta Lake 사용

앞서 언급했듯이 Delta Lake는 기존 스토리지 위에서 실행되며 기존 Apache Spark API와 완벽하게 호환됩니다. 즉, Spark가 이미 설치되어 있거나 이전 섹션에 지정된 컨테이너가 있는 경우 Delta Lake를 쉽게 시작할 수 있습니다.

Spark가 설치되면 delta-spark 2.4.0 패키지를 설치할 수 있습니다. PySpark 디렉터리에서 delta-spark 패키지를 찾을 수 있습니다. 명령 셀에 다음 명령을 입력합니다.

pip 설치 델타 스파크

Delta-spark를 설치하면 다음과 같이 Python 셀을 대화형으로 실행할 수 있습니다.

```
pyspark --packages io.delta:<delta_version>
--conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension" --conf
"spark.sql.catalog.spark_catalog=
org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

그리면 대화형으로 명령을 실행할 수 있는 PySpark 셀이 제공됩니다.

에 오신 것을 환영합니다

```
/__\____/ /_\V_\
`/_'/_/_.~\_,/_/_
\_ 버전 3.2.2
/_/
```

Python 버전 3.9.13 사용(tags/v3.9.13:6de2ca5, 2022년 5월 17일 16:36:42)

Spark 컨텍스트 웹 UI는 <http://host.docker.internal:4040>에서 사용할 수 있습니다. Spark 컨텍스트는 'sc'(마스터 = local[*], 앱 ID = local-1665944381326)로 사용할 수 있습니다.

SparkSession은 'spark'로 사용 가능합니다.

이제 셀 내에서 대화형 PySpark 명령을 실행할 수 있습니다. 우리는 항상 Spark를 사용하여 range()를 생성하여 Delta Lake 형식으로 저장할 수 있는 DataFrame을 생성함으로써 빠른 테스트를 수행합니다 (자세한 내용은 29 페이지의 "Spark 프로그램 생성 및 실행: helloDeltaLake" 참조).

전체 코드는 여기에 제공됩니다:

```
데이터 = 스파크.범위(0, 10)
data.write.format("delta").mode("overwrite").save("/book/testShell")
```

다음은 전체 실행입니다.

```
>>> 데이터 = 스파크.범위(0, 10) >>>
data.write.format("delta").mode("overwrite").save("/book/testShell")
>>>
```

여기서는 range()를 생성하는 명령문과 write 명령문이 뒤따르는 것을 볼 수 있습니다 . Spark Executor가 실행되는 것을 볼 수 있습니다. 출력 디렉터리를 열면 생성된 델타 테이블을 찾을 수 있습니다(델타 테이블 형식에 대한 자세한 내용은 다음 섹션 참조).

Spark Scala 셀에서 Delta Lake 실행

대화형 Spark Scala 셀에서 Delta Lake를 실행할 수도 있습니다. [Delta Lake 빠른 시작](#)에 지정된 대로 다음과 같이 Delta Lake 패키지를 사용하여 Scala 셀을 시작할 수 있습니다.

```
Spark-shell --packages io.delta:<delta_version>
--conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension" --conf
"spark.sql.catalog.spark_catalog=
org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

그리면 대화형 Scala 셀이 시작됩니다.

Spark 컨텍스트 웹 UI는 <http://host.docker.internal:4040>에서 사용할 수 있습니다. Spark 컨텍스트는 'sc'(마스터 = local[*], 앱 ID = local-1665950762666)로 사용할 수 있습니다.

다.

Spark 세션은 'spark'로 사용 가능합니다.

에 오신 것을 환영합니다

```

 /__/
 /_/_/____/ /_/\V_\
 `/_/`/_/_.~\_,/_/~/\
 버전 3.2.2
 /_

```

Scala 버전 2.12.15(Java HotSpot(TM) 64비트 서버 VM, Java 1.8.0_311) 사용
 평가할 표현식을 입력하세요.
 자세한 내용을 보려면 :help를 입력하세요.

스칼라>

이제 셀 내에서 대화형 Scala 명령을 실행할 수 있습니다. PySpark 셀에서 했던 것처럼 Scala에서도 유사한 테스트를 수행해 보겠습니다.

```
val 데이터 = Spark.range(0, 10)
데이터.write.format("delta").mode("overwrite").save("/book/testShell")
```

전체 실행은 다음과 같습니다.

```
scala> val 데이터 = Spark.range(0, 10) 데이터:
org.apache.spark.sql.Dataset[Long] = [id: bigint]

scala> 데이터.write.format("delta").mode("overwrite").save("/book/testShell")
```

다시 한번 출력을 확인하면 생성된 델타 테이블을 찾을 수 있습니다.

Databricks에서 Delta Lake 실행

이 책의 뒷부분에 나오는 예제의 경우 **Databricks Community Edition** Delta Lake를 운영하도록 선택되었습니다. 이는 무료이고 Spark 및 Delta Lake 설정을 단순화하며 자체 클라우드 계정이 필요하지 않거나 클라우드 컴퓨팅 또는 스토리지 리소스를 제공할 필요가 없기 때문에 코드 샘플을 개발하고 실행하기 위해 선택되었습니다. Databricks Community Edition을 사용하면 사용자는 이 플랫폼에 설치된 Delta Lake를 사용하여 완전한 노트북 환경과 최신 런타임을 갖춘 클러스터에 액세스할 수 있습니다.

로컬 머신에서 Spark 및 Delta Lake를 실행하지 않으려는 경우 Azure, AWS 또는 Google Cloud와 같은 클라우드 플랫폼의 Databricks에서 Delta Lake를 실행할 수도 있습니다. 이러한 환경에서는 설치된 런타임에 이미 Delta Lake 버전이 설치되어 있으므로 Delta Lake를 쉽게 시작할 수 있습니다.

클라우드의 또 다른 이점은 임의 크기의 실제 Spark 클러스터를 생성할 수 있으며, 잠재적으로 수백 개의 노드에 걸쳐 최대 수천 개의 코어를 생성하여 테라바이트 또는 페타바이트 규모의 데이터를 처리할 수 있다는 것입니다.

클라우드에서 Databricks를 사용하는 경우 두 가지 옵션이 있습니다. 인기 있는 노트북을 사용하거나 선호하는 개발 환경을 dbx를 사용하여 클라우드 기반 **Databricks 클러스터에 연결할 수 있습니다.** Databricks 연구소의 dbx는 편집 환경에서 Databricks 클러스터에 연결할 수 있는 오픈 소스 도구입니다.

Spark 프로그램 생성 및 실행: helloDeltaLake

delta-spark 패키지가 설치되면 첫 번째 PySpark 프로그램을 만드는 것은 매우 간단합니다. PySpark 프로그램을 만들려면 다음 단계를 따르세요.

새 파일을 만듭니다(우리 이름은 helloDeltaLake.py입니다). 필요한 가져오기를 추가합니다. 최소한 PySpark 및 Delta Lake를 가져와야 합니다.

```
델타 가져오기에서 pyspark
가져오기 *
```

다음으로, 다음과 같이 Delta Lake 확장을 로드하는 SparkSession 빌더를 만듭니다.

```
# 델타 확장을 사용하여 빌더를 만듭니다. builder =
pyspark.sql.SparkSession.builder.appName("MyApp")
    .config("spark.sql.extensions",
            "io.delta.sql.DeltaSparkSessionExtension")
    .config("spark.sql.catalog.spark_catalog",
            "org.apache.spark.sql.delta.catalog.DeltaCatalog")
```

다음으로 SparkSession 개체 자체를 만들 수 있습니다. SparkSession 개체를 생성하고 해당 버전을 인쇄하여 개체가 유효한지 확인합니다.

```
# 빌더를 사용하여 Spark 인스턴스 생성 # 결과적으로 이제 델타 테이블을 읽고 쓸
수 있습니다. Spark =configure_spark_with_delta_pip(builder).getOrCreate() print(f"Hello,
Spark version: {spark.version}")
```

Delta Lake 확장이 올바르게 작동하는지 확인하기 위해 범위를 만들고 Delta Lake 형식으로 작성합니다.

```
# 범위를 만들고 Delta Lake 형식으로 저장하여 # Delta Lake 확장이 실제로 작동하는지 확인합니다. df =
Spark.range(0, 10).df.write.format("delta").mode("overwrite").save("/book/
chapter02/helloDeltaLake")
```

이로써 시작 프로그램의 코드가 완성되었습니다. 책 코드 저장소의 /chapter02/helloDeltaLake.py 위치에서 전체 코드 파일을 찾을 수 있습니다. 이 코드는 자신만의 코드를 작성하려는 경우 시작하기 좋은 곳입니다.

프로그램을 실행하려면 Windows에서 명령 프롬프트를 시작하거나 MacOS에서 터미널을 시작하고 코드가 있는 폴더로 이동하면 됩니다. 프로그램을 입력으로 사용하여 PySpark를 시작하기만 하면 됩니다.

```
pyspark < helloDeltaLake.py
```

프로그램을 실행하면 Spark 버전 출력이 표시됩니다(표시되는 버전은 리더가 설치한 Spark 버전에 따라 다름).

안녕하세요, 스파크 버전: 3.4.1

그리고 출력을 보면 유효한 델타 테이블이 작성되었음을 알 수 있습니다.
Delta Lake 형식에 대한 자세한 내용은 다음 섹션에서 다룹니다.

이 시점에서 PySpark와 Delta Lake가 성공적으로 설치되었으며 Delta Lake 확장 기능을 사용하여 본격적인 PySpark 프로그램을 코딩하고 실행할 수 있습니다. 이제 자체 프로그램을 실행할 수 있으므로 다음 섹션에서 Delta Lake 형식을 자세히 살펴볼 준비가 되었습니다.

델타 레이크 형식

이 섹션에서는 Delta Lake 개방형 테이블 형식에 대해 자세히 살펴보겠습니다. 이 형식을 사용하여 파일을 저장하면 추가 메타데이터가 포함된 표준 Parquet 파일이 생성됩니다. 이 추가 메타데이터는 Delta Lake의 핵심 기능을 활성화하고 다양한 작업 중에서 INSERT, UPDATE, DELETE 와 같은 기존 RDBMS에서 일반적으로 볼 수 있는 DML 작업을 수행하기 위한 기반입니다 .

Delta Lake는 데이터를 Parquet 파일로 작성하므로 Parquet 파일 형식에 대해 좀 더 자세히 살펴보겠습니다. 먼저 간단한 Parquet 파일을 작성하고 Spark에서 작성한 아티팩트를 자세히 살펴보겠습니다. 이를 통해 이 책 전체에서 활용하게 될 파일을 잘 이해할 수 있습니다.

다음으로, Delta Lake 형식으로 파일을 작성하여 트랜잭션 로그가 포함된 _delta_log 디렉토리 생성을 트리거하는 방법을 알아봅니다. 이 트랜잭션 로그를 자세히 살펴보고 이것이 단일 정보 소스를 생성하는 데 어떻게 사용되는지 살펴보겠습니다. 트랜잭션 로그가 1 장에서 언급한 ACID 원자성 속성을 구현하는 방법을 살펴보겠습니다 .

Delta Lake가 트랜잭션을 개별 원자적 커밋 작업으로 분류하는 방법과 트랜잭션 로그에 이러한 작업을 정렬된 원자 단위로 기록하는 방법을 살펴보겠습니다. 마지막으로 여러 사용 사례를 살펴보고 어떤 Parquet 데이터 파일과 트랜잭션 로그 항목이 기록되고 이러한 항목에 무엇이 저장되는지 조사하겠습니다.

트랜잭션 로그 항목은 모든 트랜잭션에 대해 기록되므로 여러 개의 작은 파일이 생길 수 있습니다. 이 접근 방식의 확장성을 유지하기 위해 Delta Lake는 전체 트랜잭션 상태를 사용하여 트랜잭션 10개마다(작성 당시) 검사점 파일을 생성합니다. 이런 방식으로 Delta Lake 리더는 검사점 파일과 나중에 작성된 몇 가지 트랜잭션 항목을 간단히 처리할 수 있습니다. 그 결과 빠르고 확장 가능한 메타데이터 시스템이 탄생했습니다.

Parquet 파일

Apache Parquet 파일 형식은 지난 20년 동안 가장 인기 있는 빅 데이터 형식 중 하나였습니다. Parquet은 오픈 소스이므로 Apache Hadoop 라이센스에 따라 무료로 사용할 수 있으며 대부분의 Hadoop 데이터 처리 프레임워크와 호환됩니다.

CSV 또는 Avro와 같은 행 기반 형식과 달리 Parquet는 열 기반 형식입니다. 즉, 각 열/필드의 값은 각 레코드가 아닌 서로 옆에 저장됩니다. **그림 2-1에서는** 행 기반 레이아웃과 열 기반 레이아웃의 차이점과 이것이 논리적 테이블에 표시되는 방식을 보여줍니다.

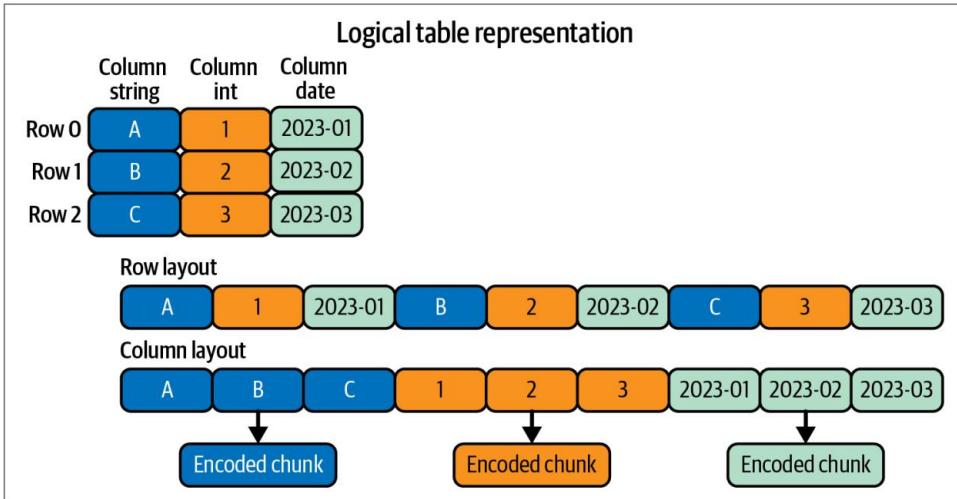


그림 2-1. 행 기반 레이아웃과 열 기반 레이아웃의 차이점

그림 2-1 에서는 행 레이아웃처럼 행 값을 순차적으로 저장하는 대신 열 레이아웃이 열 값을 순차적으로 저장하는 것을 보여줍니다. 이 열 형식은 열 단위로 압축하는 데 도움이 됩니다. 또한 이 형식은 각 데이터 유형에 대해 유연한 압축 옵션과 확장 가능한 인코딩 스키마를 지원하도록 구축되었습니다. 즉, 정수 데이터 유형과 문자열 데이터 유형을 압축하는 데 서로 다른 인코딩을 사용할 수 있습니다.

Parquet 파일도 행 그룹과 메타데이터로 구성됩니다. 행 그룹에는 동일한 열의 데이터가 포함되므로 각 열은 행 그룹에 함께 저장됩니다.

Parquet 파일의 메타데이터에는 이러한 행 그룹에 대한 정보뿐만 아니라 열에 대한 정보(예: 최소/최대 값, 값 수) 및 데이터 스키마도 포함되어 있습니다. 이를 통해 Parquet는 추가 메타데이터가 포함된 자체 설명 파일이 됩니다. 더 나은 데이터 건너뛰기.

그림 2-2에서는 Parquet 파일이 행 그룹과 메타데이터로 구성되는 방식을 보여줍니다. 각 행 그룹은 데이터세트의 각 열에 대한 열 청크로 구성되며, 각 열 청크는 열 데이터가 포함된 하나 이상의 페이지로 구성됩니다. 더 자세히 알아보려면

설명서를 읽고 Parquet 파일 형식에 대해 자세히 알아보려면 Apache Parquet 웹 사이트와 설명서를 참조하세요.

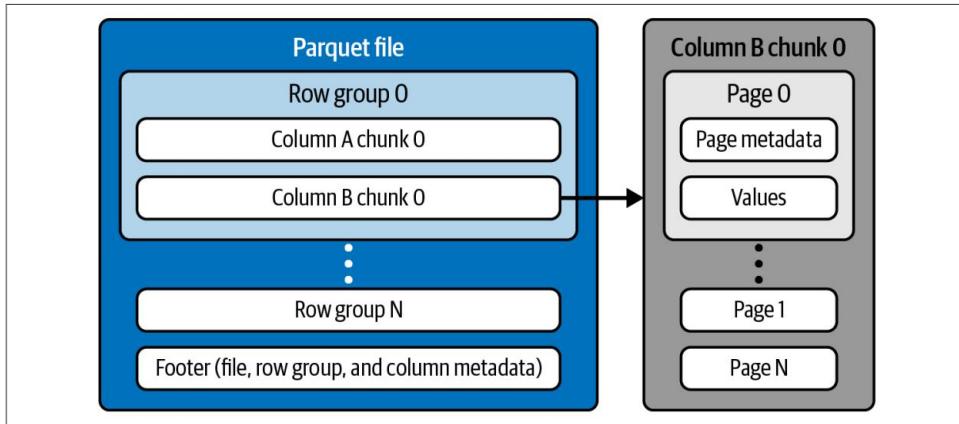


그림 2-2. 쪽모이 세공 파일 구성

Parquet 파일의 장점

열 기반 형식, 스토리지 레이아웃, 메타데이터 및 오랜 인기로 인해 Parquet 파일은 분석 워크로드 및 빅 데이터 작업 시 다음과 같은 몇 가지 강력한 이점을 제공합니다.

고성능 Parquet 파일

일은 열 중심 형식이기 때문에 이러한 알고리즘이 각 열에 저장된 유사한 값과 데이터 유형을 활용할 수 있기 때문에 더 나은 압축 및 인코딩이 가능합니다. I/O 집약적인 작업의 경우 이 압축된 데이터는 성능을 크게 향상시킬 수 있습니다.

Parquet 파일의 경우 열 값이 함께 저장되면 쿼리는 행 기반 형식의 경우 모든 열을 읽어야 하는 것과 달리 해당 쿼리에 필요한 열만 읽으면 됩니다. 이는 열 형식이 작업을 위해 읽어야 하는 데이터의 양을 줄여 성능을 향상시킬 수 있음을 의미합니다.

Parquet 파일에 포함된 메타데이터는 데이터의 일부 기능을 설명합니다. 여기에는 행 그룹, 데이터 스키마, 그리고 가장 중요한 열에 대한 정보가 포함됩니다. 열 메타데이터에는 최소/최대 값, 값 개수 등의 정보가 포함됩니다. 이 메타데이터는 함께 각 작업(즉, 데이터 건너뛰기)에 대해 읽어야 하는 데이터의 양을 줄이고 쿼리 성능을 훨씬 향상시킵니다.

비용 효율성

Parquet 파일은 더 나은 압축 및 인코딩을 활용할 수 있으므로 본질적으로 데이터 저장 비용이 더 효율적입니다. 본질적으로 압축된 데이터는 파일을 저장할 때 디스크 공간을 덜 차지하므로 필연적으로 저장 공간이 줄어들고 저장 비용이 절감됩니다.

상호 운용성

Parquet 파일은 지난 20년 동안 특히 레거시 빅 데이터 처리 프레임워크 및 도구(예: Hadoop)에서 매우 인기가 있었기 때문에 다양한 도구와 엔진에서 매우 광범위하게 지원되며 뛰어난 상호 운용성을 제공합니다.

Parquet 파일 작성

책 저장소에서 /chapter02/writeParquetFile Python 프로그램은 Spark DataFrame을 메모리에 저장하고 표준 PySpark API를 사용하여 Parquet 형식으로 /parquetData 폴더에 씁니다.

```
데이터 = 스파크.범위(0, 100)
data.write.format("parquet") .mode("덮어쓰기") .save('/book/chapter02/parquetData')
```

우리의 경우 디스크에 기록된 내용을 살펴보면 다음과 같은 내용을 볼 수 있습니다(로컬 시스템에 따라 다른 결과가 나타날 수 있음).

```
C:\book\chapter02\parquetData 디렉터리 2022년 10월 17일 2022
년 10월 17일
511 part-00000-a3885270...-c000.snappy.parquet 2022년 10월 17일 513 part-00001-a3885270...-c000.snappy.parquet 2022년 10월 17일 517 파트-00002-a3885270...-c000.snappy.parquet 2022년 10월 17일 513 파트-00003-a3885270...-c000.snappy.parquet 10/17/2022 513 파트-00004-a3885270...-c000.snappy.parquet 10/17/2022 517 파트-00005-a3885270...-c000.snappy.parquet 10/17/2022 513 파트-00006-a3885270...-c000.snappy.parquet 2022년 10월 17일 513 파트-00007-a3885270...-c000.snappy.parquet 2022년 10월 17일 517 파트-00008-a3885270...-c000.snappy.parquet 2022년 10월 17일 513 파트-00010-a3885270...-c000.snappy.parquet 10/17/2022 517 파트-00011-a3885270...-c000.snappy.parquet
```

빅 데이터 세계를 처음 접하는 개발자는 이 시점에서 약간 충격을 받을 수 있습니다. 우리는 100개의 숫자를 한 번만 썼는데 어떻게 12개의 Parquet 파일이 생겼습니까? 약간의 설명이 필요할 수 있습니다.

첫째, 쓰기에서 지정한 파일 이름은 실제로 파일이 아닌 디렉터리 이름입니다. 보시다시피 /parquetData 디렉터리에는 12개의 Parquet 파일이 포함되어 있습니다.

.parquet 파일을 보면 12개의 파일이 있는 것을 볼 수 있습니다. Spark는 시스템이 Spark 클러스터의 각 CPU 코어를 바쁜 상태로 유지하려고 시도하는 고도의 병렬 계산 환경입니다. 우리의 경우에는 로컬 머신에서 실행 중입니다.

이는 클러스터에 머신이 하나 있다는 의미입니다. 우리 시스템에 대한 정보를 보면 코어가 12개라는 것을 알 수 있습니다.

작성된 .parquet 파일 수를 살펴보면 클러스터의 코어 수와 동일한 12개의 파일이 있음을 알 수 있습니다. 이것이 이 시나리오에서 Spark의 기본 동작입니다. 파일 수는 사용 가능한 코어 수와 동일합니다. 코드에 다음 명령문을 추가한다고 가정합니다.

```
데이터 = 스파크.범위(0, 100)
data.write.format("parquet") .mode("덮어쓰기") .save('/book/
chapter02/parquetData')
print(f"파티션 수: {data.rdd.getNumPartitions()}"
```

출력에서 실제로 12개의 파일이 있음을 확인할 수 있습니다.

파티션 수: 12

100개의 숫자만 쓰는 시나리오에서는 이것이 과도한 것처럼 보일 수 있지만 매우 큰 파일을 읽거나 쓰는 시나리오를 상상할 수 있습니다. 파일을 분할하고 병렬로 처리하는 기능이 있으면 성능이 크게 향상될 수 있습니다.

출력에 표시되는 .crc 파일은 순환 중복 검사 파일입니다. Spark는 이를 사용하여 데이터가 손상되지 않았는지 확인합니다. 이러한 파일은 항상 매우 작기 때문에 제공되는 유ти리티에 비해 오버헤드가 매우 적습니다. 이러한 파일 생성을 해제하는 방법이 있지만 이점이 오버헤드보다 훨씬 크기 때문에 그렇게 하지 않는 것이 좋습니다.

출력의 최종 파일은 _SUCCESS 및 _SUCCESS.crc 파일입니다. Spark는 이러한 파일을 사용하여 모든 파티션이 올바르게 작성되었는지 확인하는 방법을 제공합니다.

델타 테이블 작성 지금까지 우리는

Parquet 파일을 사용해 작업했습니다. 이제 이전 섹션의 첫 번째 예를 가져와서 Parquet(코드: /chapter02/writeDeltaFile.py) 대신 Delta Lake 형식으로 저장해 보겠습니다. 우리가 해야 할 일은 코드에 표시된 대로 Parquet 형식을 Delta 형식으로 바꾸는 것뿐입니다.

```
data = Spark.range(0, 100) data.write
\ .format("delta") \ .mode("overwrite")
\ .save('/book/chapter02/deltaData')
print(f"파일 수는:
{data.rdd.getNumPartitions()}"
```

동일한 수의 파티션을 얻습니다.

파일 수: 12

출력을 보면 _delta_log 파일이 추가된 것을 볼 수 있습니다.

```
C:\book\chapter02\deltaaData 디렉터리 2022년 10월 17일
16 .part-00000-...-c000.snappy.parquet.crc 2022년 10월 17일 16 .part-00001-...
c000.snappy .parquet.crc 2022년 10월 17일 16 .part-00002-...-c000.snappy.parquet.crc 2022
년 10월 17일 16 .part-00003-...-c000.snappy.parquet.crc 10/ 2022년 17일 16 .part-00004-...
c000.snappy.parquet.crc 2022년 10월 17일 16 .part-00005-...-c000.snappy.parquet.crc 2022
년 10월 17일 16 .part-00006-...-c000.snappy.parquet.crc 2022년 10월 17일 16 .part-00007-...
c000.snappy.parquet.crc 2022년 10월 17일 16 .part-00008-... -c000.snappy.parquet.crc 2022
년 10월 17일 16 .part-00009-...-c000.snappy.parquet.crc 2022년 10월 17일 16 .part-00010-...
c000.snappy.parquet .crc 2022년 10월 17일 16 .part-00011-...-c000.snappy.parquet.crc 2022
년 10월 17일 524 part-00000-...-c000.snappy.parquet 2022년 10월 17일 519 부분 -00001-...
c000.snappy.parquet 2022년 10월 17일 523 파트-00002-...-c000.snappy.parquet 2022년 10월
17일 519 파트-00003-...-c000.snappy.parquet 2022년 10월 17일 519 파트-00004-...
c000.snappy.parquet 2022년 10월 17일 522 파트-00005-...-c000.snappy.parquet 2022년 10월
17일 519 파트-00006-...-c000.snappy.parquet 2022년 10월 17일 519 part-00007-...
c000.snappy.parquet 2022년 10월 17일 523 part-00008-...-c000.snappy.parquet 2022
년 10월 17일 519 파트-00009-...-c000.snappy.parquet 2022년 10월 17일 519 파
트-00010-...-c000.snappy.parquet 2022년 10월 17일 523 파트-00011-...
c000.snappy .parquet 2022/10/17 <DIR> _delta_log 24 파일 6,440바이트
```

_delta_log 파일에는 데이터에 대해 수행된 모든 단일 작업이 포함된 트랜잭션 로그가 포함되어 있습니다.



Delta Lake 3.0에는 UniForm("Universal Format"의 약어)이 포함되어 있습니다. UniForm을 활성화하면 Delta 테이블을 Iceberg와 같은 다른 개방형 테이블 형식인 것처럼 읽을 수 있습니다. 이를 통해 테이블 형식 호환성에 대해 걱정하지 않고 더 광범위한 도구를 사용할 수 있습니다.

```
%sql
테이블 T 생성
```

```
TBLPROPERTIES( 'delta.columnMapping.mode' = 'name',
    'delta.universalFormat.enabledFormats' = 'iceberg')
AS
SELECT * FROM source_table;
```

UniForm은 기본 Parquet 데이터 사본 하나 위에 Delta 메타데이터와 함께 Apache Iceberg 메타데이터를 자동으로 생성합니다.

Iceberg에 대한 메타데이터는 테이블 생성 시 자동으로 생성되며 테이블이 업데이트될 때 마다 업데이트됩니다.

Delta Lake 트랜잭션 로그

Delta Lake 트랜잭션 로그(DeltaLog라고도 함)는 Delta Lake 테이블 생성 이후 수행된 모든 트랜잭션의 순차적 레코드입니다. 이는 ACID 트랜잭션, 확장 가능한 메타데이터 처리, 시간 여행 등 중요한 기능의 핵심이기 때문에 Delta Lake 기능의 핵심입니다.

트랜잭션 로그의 주요 목표는 여러 판독기와 기록기가 특정 버전의 데이터 세트 파일에서 동시에 작업할 수 있도록 하고, 보다 효율적인 작업을 위해 데이터 건너뛰기 인덱스와 같은 추가 정보를 실행 엔진에 제공하는 것입니다. Delta Lake 트랜잭션 로그는 항상 사용자에게 데이터에 대한 일관된 보기를 보여주고 단일 정보 소스 역할을 합니다. 이는 사용자가 Delta 테이블에 적용한 모든 변경 사항을 추적하는 중앙 저장소입니다.

델타 테이블 판독기가 처음으로 델타 테이블을 읽거나 마지막으로 읽은 이후 수정된 열린 파일에 대해 새 쿼리를 실행하면 Delta Lake는 트랜잭션 로그를 확인하여 테이블의 최신 버전을 가져옵니다. 이렇게 하면 사용자의 파일 버전이 가장 최근 쿼리 기준으로 항상 마스터 레코드와 동기화되고 사용자가 파일에 대해 서로 다르고 충돌하는 변경을 수행할 수 없습니다.

트랜잭션 로그가 원자성을 구현하는 방법 1 장에서 우리는 원자성이 파일에 대해 수행되는

모든 작업(예: INSERT, UPDATE, DELETE 또는 MERGE)이 전체적으로 성공하거나 전혀 성공하지 않도록 보장한다는 것을 배웠습니다. 원자성이 없으면 하드웨어 오류나 소프트웨어 버그로 인해 데이터 파일이 부분적으로 기록되어 데이터가 손상되거나 최소한 유효하지 않은 데이터가 발생할 수 있습니다.

트랜잭션 로그는 Delta Lake가 원자성 보장을 제공할 수 있는 메커니즘입니다. 트랜잭션 로그는 또한 메타데이터, 시간 여행 및 대규모 테이블 형식 데이터 세트에 대한 상당히 빠른 메타데이터 작업을 담당합니다.

트랜잭션 로그는 델타 테이블이 생성된 이후 해당 테이블에 대해 수행된 모든 트랜잭션에 대한 정렬된 레코드입니다. 이는 단일 정보 소스 역할을 하며 테이블에 대한 모든 변경 사항을 추적합니다. 트랜잭션 로그를 통해 사용자는 자신의 데이터에 대해 추론하고 데이터의 완전성과 품질을 신뢰할 수 있습니다. 간단한 규칙은 작업이 트랜잭션 로그에 기록되지 않으면 결코 발생하지 않는다는 것입니다. 다음 섹션에서는 몇 가지 예를 통해 이러한 원칙을 설명합니다.

트랜잭션을 원자성 커밋으로 분할 테이블 또는 스토리지 파일을 수정하기 위해 일련의 작업

(예: INSERT, UPDATE, DELETE 또는 MERGE)을 수행할 때마다 Delta Lake는 해당 작업을 다음으로 구성된 일련의 원자성, 개별 단계로 분할합니다. 표 2-1에 표시된 작업 중 하나 이상

표 2-1. 트랜잭션 로그 항목에서 가능한 작업 목록

행동	설명
파일 추가	추가합니다.
파일을 지우다	파일을 제거합니다.
메타데이터 업데이트	테이블의 메타데이터를 업데이트합니다(예: 테이블이나 파일의 이름, 스키마 또는 파티셔닝 변경). 테이블이나 파일의 첫 번째 트랜잭션 로그 항목에는 항상 스키마, 파티션 열 및 기타 정보가 포함된 업데이트 메타데이터 작업이 포함됩니다.
거래 설정	구조화된 스트리밍 작업이 지정된 스트림 ID로 마이크로 배치를 커밋했음을 기록합니다. 자세한 내용은 8 장을 참조하세요.
프로토콜 변경	Delta Lake 트랜잭션 로그를 최신 소프트웨어 프로토콜로 전환하여 새로운 기능을 활성화합니다.
커밋 정보	커밋에 대한 정보, 어떤 작업이 수행되었는지, 어디서, 언제 수행되었는지에 대한 정보가 포함됩니다. 모든 트랜잭션 로그 항목에는 커밋 정보 작업이 포함됩니다.

이러한 작업은 커밋이라는 순서가 지정된 원자 단위로 트랜잭션 로그 항목(*.json)에 기록됩니다. 이는 Git 소스 제어 시스템이 원자성 커밋으로 변경 사항을 추적하는 방법과 유사합니다. 이는 또한 트랜잭션 로그의 각 커밋을 재생하여 파일의 현재 상태를 얻을 수 있음을 의미합니다.

예를 들어 사용자가 테이블에 새 열을 추가하기 위해 트랜잭션을 생성한 다음 데이터를 추가하는 경우 Delta Lake는 이 트랜잭션을 구성 요소 작업 부분으로 나누고 트랜잭션이 완료되면 다음과 같이 트랜잭션 로그에 추가합니다. 다음 커밋:

1. 메타데이터 업데이트: 새 열을 포함하도록 스키마 변경
2. 파일 추가: 추가된 새 파일
마다

파일 수준의 트랜잭션 로그 델타 테이블을 작성하면 해당

파일의 트랜잭션 로그가 자동으로 _delta_log 하위 디렉터리에 생성됩니다. 계속해서 Delta 테이블을 변경하면 이러한 변경 사항은 트랜잭션 로그에 순서가 지정된 원자성 커밋으로 자동 기록됩니다. 각 커밋은 00000000000000000000.json으로 시작하는 JSON 파일로 작성됩니다.

파일을 추가로 변경하면 Delta Lake는 숫자 오름차순으로 추가 JSON 파일을 생성하므로 다음 커밋은 00000000000000000001.json으로 기록되고 다음 커밋은 00000000000000000002.json으로 기록됩니다.

곧.

이 장의 나머지 부분에서는 가독성을 위해 트랜잭션 로그 항목에 대해 축약된 형식을 사용합니다. 최대 19자리까지 표시하는 대신 최대 5자리의 축약된 형식을 사용합니다(따라서 더 긴 표기법 대신 00001.json을 사용하게 됩니다).

또한 Parquet 파일의 이름도 단축될 예정입니다. 이러한 이름은 일반적으로 다음과 같습니다.

부품-00007-71c70d7f-c7a8-4a8c-8c29-57300cf929b-c000.snappy.parquet

데모 및 설명을 위해 GUID와 snappy.parquet 부분을 제외하고 이와 같은 이름을 part-00007.parquet로 축약하겠습니다.

시각화 예시에서는 영향을 받은 작업 이름과 데이터 파일 이름을 사용하여 각 트랜잭션 항목을 시각화합니다. 예를 들어, [그림 2-3](#)에는 단일 트랜잭션 파일에 제거(파일) 작업과 다른 추가(파일) 작업이 있습니다.

Action	Part name
Remove	part-00001
Add	part-00004

00002.json

그림 2-3. 트랜잭션 로그 항목에 대한 표기법

동일한 파일에 여러 쓰기 쓰기 이 섹션 전

체에서 각 코드 단계를 자세히 설명하는 그림 세트를 사용합니다. 각 단계에 대해 다음 정보가 표시됩니다.

- 실제 코드 조각은 두 번째 열에 표시됩니다. • 코드 조각 옆에는 코드 조각 실행의 결과로 작성된 Parquet 데이터 파일이 표시됩니다.
- 마지막 열에는 트랜잭션 로그 항목의 JSON 파일이 표시됩니다. 각 트랜잭션 로그 항목에 대한 작업과 영향을 받는 Parquet 데이터 파일 이름을 표시합니다.

이 첫 번째 예에서는 책 저장소의 Chapter02/MultipleWriteOperations.py를 사용하여 동일한 파일에 대한 여러 쓰기를 표시합니다.

다음은 [그림 2-4](#) 의 다양한 단계에 대한 단계별 설명입니다 .

1. 먼저 새로운 Delta 테이블이 경로에 기록됩니다. 하나의 Parquet 파일이 출력 경로 (part-00000.parquet)에 기록되었습니다. 첫 번째 트랜잭션 로그 항목(00000.json)이 _delta_log 디렉터리에 생성되었습니다. 이는 파일에 대한 첫 번째 트랜잭션 로그 항목이므로 메타데이터 작업 및 파일 추가 작업이 기록되어 단일 파티션 파일이 추가되었음을 나타냅니다.
2. 다음으로 테이블에 데이터를 추가합니다. 새로운 Parquet 파일(part-00001.parquet)이 작성되었으며 ,

거래 로그. 첫 번째 단계와 마찬가지로 항목에는 새 파일을 추가했기 때문에 파일 추가 작업이 포함됩니다.

- 더 많은 데이터를 추가합니다. 이번에도 새 데이터 파일(part-00002.parquet)이 기록되고 파일 추가 작업을 통해 새 트랜잭션 로그 파일(00002.json)이 트랜잭션 로그에 추가됩니다.

Step	Code	Parquet files written	JSON files in _delta_log	
			Action	Part name
1	<pre>df .coalesce(1) .write .format("delta") .save(DATALAKE_PATH)</pre>	part-00000.parquet	Metadata	N/A
			Add	<i>part-00000</i>
				00000.json
2	<pre>df .coalesce(1) .write .format("delta") .mode ("append") .save(DATALAKE_PATH)</pre>	part-00001.parquet	Action	Part name
			Add	<i>part-00001</i>
				00001.json
3	<pre>df .coalesce(1) .write .format("delta") .mode ("append") .save(DATALAKE_PATH)</pre>	part-00002.parquet	Action	Part name
			Add	<i>part-00002</i>
				00002.json

그림 2-4. 동일한 파일에 여러 번 쓰기

각 트랜잭션 로그 항목에는 트랜잭션에 대한 감사 정보가 포함된 커밋 정보 작업도 있습니다. 가독성을 위해 그림에서 커밋 정보 로그 항목을 생략했습니다.

쓰기 작업 순서는 매우 중요합니다. 각 쓰기 작업마다 항상 데이터 파일이 먼저 기록되고 해당 작업이 성공한 경우에만 트랜잭션 로그 파일이 _delta_log 폴더에 추가됩니다. 트랜잭션 로그 항목이 성공적으로 기록된 경우에만 트랜잭션이 완료된 것으로 간주됩니다.

최신 버전의 델타 테이블 읽기 시스템이 델타 테이

블을 읽을 때 트랜잭션 로그를 반복하여 테이블의 현재 상태를 "컴파일"합니다. 파일을 읽을 때 이벤트의 순서는 다음과 같습니다.

1. 트랜잭션 로그 파일을 먼저 읽습니다.
2. 로그 파일을 기반으로 데이터 파일을 읽습니다.

다음으로 이전 예제(multipleWriteOperations.py)에서 작성된 Delta 테이블에 대한 해당 시퀀스를 설명합니다. Delta는 모든 로그 파일(00000.json, 00001.json 및 00002.json)을 읽습니다. 그림 2-5 와 같이 로그 정보를 기반으로 3개의 데이터 파일을 읽습니다.

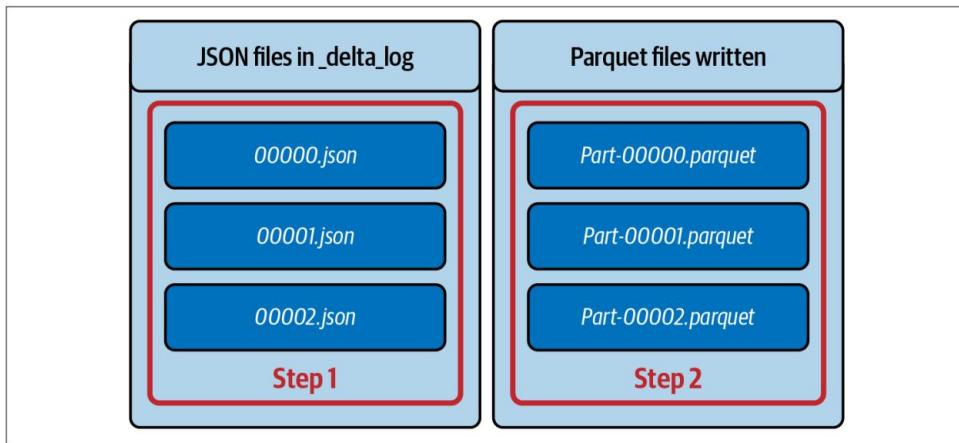


그림 2-5. 읽기 작업

또한 작업 순서는 트랜잭션 로그에서 더 이상 참조되지 않는 데이터 파일이 있을 수 있음을 의미합니다. 실제로 이는 UPDATE 또는 DELETE 시나리오에서 흔히 발생합니다. Delta Lake는 사용자가 Delta Lake의 시간 여행 기능 (6 장에서 설명)을 사용하는 경우 다시 필요할 수 있으므로 이러한 데이터 파일을 삭제하지 않습니다. VACUUM 명령을 사용하여 오래되고 사용되지 않는 데이터 파일을 제거할 수 있습니다 (6 장에서도 설명).

쓰기 작업 실패 시나리오 다음으로 쓰기

작업이 실패하면 어떤 일이 발생하는지 살펴보겠습니다. 이전 쓰기 시나리오에서는 그림 2-4 의 3단계에 있는 쓰기 작업이 중간에 실패했다고 가정해 보겠습니다. Parquet 파일의 일부가 기록되었을 수 있지만 트랜잭션 로그 항목 00002.json은 기록되지 않았습니다. 그림 2-6 과 같은 시나리오가 있을 것입니다.

Step	Code	Parquet files written	JSON files in_delta_log						
1	<pre>df .coalesce(1) .write .format("delta") .save(DATALAKE_PATH)</pre>	part-00000.parquet	<table border="1"> <thead> <tr> <th>Action</th><th>Part name</th></tr> </thead> <tbody> <tr> <td>Metadata</td><td>N/A</td></tr> <tr> <td>Add</td><td>part-00000 00000.json</td></tr> </tbody> </table>	Action	Part name	Metadata	N/A	Add	part-00000 00000.json
Action	Part name								
Metadata	N/A								
Add	part-00000 00000.json								
2	<pre>df .coalesce(1) .write .format("delta") .mode ("append") .save(DATALAKE_PATH)</pre>	part-00001.parquet	<table border="1"> <thead> <tr> <th>Action</th><th>Part name</th></tr> </thead> <tbody> <tr> <td>Add</td><td>part-00001 00001.json</td></tr> </tbody> </table>	Action	Part name	Add	part-00001 00001.json		
Action	Part name								
Add	part-00001 00001.json								
3	<pre>df .coalesce(1) .write .format("delta") .mode ("append") .save(DATALAKE_PATH)</pre>	part-00002.parquet							

그림 2-6. 마지막 쓰기 작업 중 오류가 발생했습니다.

그림 2-6에서 볼 수 있듯이 마지막 트랜잭션 파일이 누락되었습니다. 앞에서 지정한 읽기 순서에 따라 Delta Lake는 첫 번째 및 두 번째 JSON 트랜잭션 파일과 해당 part-00000 및 part-00001 Parquet 파일을 읽습니다. Delta Lake 리더는 일관되지 않은 데이터를 읽지 않습니다. 처음 두 개의 트랜잭션 로그 파일을 통해 일관된 보기를 읽습니다.

업데이트 시나리

오 마지막 시나리오는 Chapter02/UpdateOperation.py 코드 저장소에 포함되어 있습니다. 일을 단순하게 유지하기 위해 환자 정보가 포함된 작은 델타 테이블이 있습니다. 우리는 각 환자의 PatientId 와 PatientName 만 추적하고 있습니다. 이 사용 사례에서는 각 파일에 2명씩 총 4명의 환자가 포함된 Delta 테이블을 만듭니다. 다음으로 두 명의 환자로부터 데이터를 추가합니다. 마지막으로 첫 번째 환자의 이름을 업데이트합니다. 보시다시피 이번 업데이트는 예상보다 더 큰 영향을 미칩니다. 전체 업데이트 시나리오는 그림 2-7에 나와 있습니다.

Step	Code	Parquet files written		JSON files in _delta_log																		
<ul style="list-style-type: none"> • Read 00.json • Include part-0 • Include part-1 <ul style="list-style-type: none"> • Read 01.json • Include part-2 <ul style="list-style-type: none"> • Read 02.json • Remove part-0 • Include part-3 <p>Final result:</p> <ul style="list-style-type: none"> • part-1 • part-2 • part-3 <p>are included in latest data</p>	<pre>df .coalesce(2) .write.format("delta") .save(DATALAKE_PATH)</pre>	<table border="1"> <thead> <tr> <th>patientID</th> <th>Name</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>P1</td> </tr> <tr> <td>2</td> <td>P2</td> </tr> </tbody> </table> <p>part-0.parquet</p>	patientID	Name	1	P1	2	P2	<table border="1"> <thead> <tr> <th>patientID</th> <th>Name</th> </tr> </thead> <tbody> <tr> <td>3</td> <td>P3</td> </tr> <tr> <td>4</td> <td>P4</td> </tr> </tbody> </table> <p>part-1.parquet</p>	patientID	Name	3	P3	4	P4	<table border="1"> <thead> <tr> <th>Operation</th> <th>Filename</th> </tr> </thead> <tbody> <tr> <td>Add</td> <td>part0</td> </tr> <tr> <td>Add</td> <td>part1</td> </tr> </tbody> </table> <p>00000.json</p>	Operation	Filename	Add	part0	Add	part1
patientID	Name																					
1	P1																					
2	P2																					
patientID	Name																					
3	P3																					
4	P4																					
Operation	Filename																					
Add	part0																					
Add	part1																					
	<pre>df .coalesce(1) .write.format("delta") .mode("append") .save(DATALAKE_PATH)</pre>	<table border="1"> <thead> <tr> <th>patientID</th> <th>Name</th> </tr> </thead> <tbody> <tr> <td>5</td> <td>P5</td> </tr> <tr> <td>6</td> <td>P6</td> </tr> </tbody> </table> <p>part-2.parquet</p>	patientID	Name	5	P5	6	P6		<table border="1"> <thead> <tr> <th>Operation</th> <th>Filename</th> </tr> </thead> <tbody> <tr> <td>Add</td> <td>part2</td> </tr> </tbody> </table> <p>00001.json</p>	Operation	Filename	Add	part2								
patientID	Name																					
5	P5																					
6	P6																					
Operation	Filename																					
Add	part2																					
	<pre>deltaTable = DeltaTable \ .forPath(spark, DATALAKE_PATH) deltaTable.update(condition = col("patientId") == 1, set = {'name': lit("p11")})</pre>	<table border="1"> <thead> <tr> <th>patientID</th> <th>Name</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>P11</td> </tr> <tr> <td>2</td> <td>P2</td> </tr> </tbody> </table> <p>part-3.parquet</p>	patientID	Name	1	P11	2	P2		<table border="1"> <thead> <tr> <th>Action</th> <th>Part name</th> </tr> </thead> <tbody> <tr> <td>Remove</td> <td>part0</td> </tr> <tr> <td>Add</td> <td>part3</td> </tr> </tbody> </table> <p>00002.json</p>	Action	Part name	Remove	part0	Add	part3						
patientID	Name																					
1	P11																					
2	P2																					
Action	Part name																					
Remove	part0																					
Add	part3																					

그림 2-7. 업데이트 및 트랜잭션 로그

이 예에서는 다음 단계를 실행합니다.

1. 첫 번째 코드 조각은 환자 4명의 환자 ID 와 이름을 사용하여 Spark DataFrame을 생성합니다 . DataFrame을 Delta 테이블에 작성하여 .coalesce (2)를 사용하여 데이터를 두 개의 파일로 강제로 만듭니다. 결과적으로 우리는 두 개의 파일을 작성합니다. part-00000.parquet 및 part-00001.parquet 파일이 작성되면 트랜잭션 로그 항목(00000.json)이 생성됩니다. 트랜잭션 로그 항목에는 part-00000.parquet 및 part-00001.parquet 파일이 추가되었음을 나타내는 두 개의 파일 추가 작업이 포함되어 있습니다.
2. 다음 코드 조각은 두 명의 추가 환자(P5 및 P6)에 대한 데이터를 추가합니다. 그러면 part-00002.parquet 파일이 생성됩니다. 다시 한번 파일이 작성되면 트랜잭션 로그 항목(00001.json)이 작성되고 트랜잭션이 완료됩니다. 마찬가지로 트랜잭션 로그 파일에는 파일(part-00002.parquet)이 추가되었음을 나타내는 하나의 파일 추가 작업이 있습니다.

3. 코드가 업데이트를 수행합니다. 이 경우 환자 이름을 P1에서 P11까지 PatientId 1로 업데이트하려고 합니다. 현재 PatientId 1에 대한 기록은 part-0에 있습니다. 업데이트를 수행하기 위해 part-0을 읽고 맵 연산자를 사용하여 P1에서 P11까지 PatientId 1과 일치하는 모든 레코드를 업데이트합니다. part-3으로 새 파일이 작성됩니다. 마지막으로 Delta Lake는 트랜잭션 로그 항목(00002.json)을 작성합니다. part-0 파일이 제거되었음을 알리는 파일 제거 조치와 part-3 파일이 추가되었음을 알리는 추가 조치를 작성합니다. 이는 part-0의 데이터가 part-3으로 다시 작성되었고 수정된 모든 행(수정되지 않은 행과 함께)이 part-3에 추가되어 part-0 파일이 더 이상 사용되지 않게 되었기 때문입니다.

사용자가 시간 여행을 통해 과거로 돌아가고 싶어할 수 있으므로 Delta Lake는 part-0 파일을 삭제하지 않습니다. 이 경우 파일이 필요합니다. VACUUM 명령은 이와 같이 사용하지 않는 파일을 정리할 수 있습니다 ([6장](#)에서는 시간 여행 및 사용하지 않는 파일 정리에 대해 자세히 다룹니다).

이제 업데이트 중에 데이터가 어떻게 기록되는지 살펴보았으므로 [그림 2-8](#)에 설명된 대로 읽기가 읽을 내용을 어떻게 결정하는지 살펴보겠습니다.

Final data read:	patientID	Name
	3	P3
	4	P4
	5	P5
	6	P6
	1	P11
	2	P2

그림 2-8. 업데이트 후 읽기

읽기는 다음과 같이 진행됩니다.

- 첫 번째 트랜잭션 로그 항목을 읽습니다(00000.json). 이 항목은 Delta Lake에 part-0 및 part-1 파일을 포함하도록 지시합니다.
- 다음 항목(00001.json)을 읽어 Delta Lake에 part-2 파일을 포함하라고 지시합니다.
- 마지막 항목(00002.json)을 읽어서 독자에게 part-0 파일을 제거하고 part-3을 포함하라고 알립니다.

결과적으로 리더는 결국 파트 1, 파트 2, 파트 3을 읽게 되고 [그림 2-8에 표시된 올바른 데이터가 생성됩니다.](#)

대규모 메타데이터 확장 이제 트랜잭션

션 로그가 각 작업을 기록하는 방법을 살펴보았으므로 단일 Parquet 파일에 대해 수천 개의 트랜잭션 로그 항목이 포함된 매우 큰 파일을 많이 가질 수 있습니다. Delta Lake는 Spark의 읽기 성능에 부정적인 영향을 미칠 수 있는 수천 개의 작은 파일을 읽을 필요 없이 메타데이터 처리를 어떻게 확장합니까?

Spark는 대용량 파일을 읽을 때 가장 효과적인 경향이 있는데, 이 문제를 어떻게 해결합니까?

Delta Lake 작성자가 트랜잭션 로그에 커밋을 수행하면 _delta_log 폴더에 Parquet 형식으로 검사점 파일이 저장됩니다. Delta Lake 작성자는 커밋 10개마다 새 검사점을 계속 생성합니다.

체크포인트 파일은 특정 시점의 테이블 전체 상태를 저장합니다. "상태"는 파일의 실제 내용이 아니라 다양한 작업을 나타냅니다. 따라서 여기에는 모든 컨텍스트 정보와 함께 파일 추가, 파일 제거, 메타데이터 업데이트, 정보 커밋 등의 작업이 포함됩니다. 이 목록은 기본 Parquet 형식으로 저장됩니다. 이렇게 하면 Spark가 체크포인트를 빠르게 읽을 수 있습니다. 이는 Spark 리더에게 테이블 상태를 완전히 재현하고 비효율적일 수 있는 수천 개의 작은 JSON 파일을 재처리하는 것을 방지할 수 있는 "바로 가기"를 제공합니다.

체크포인트 파일 예제 다음은 여

러 커밋을 실행하고 그 결과 체크포인트 파일이 생성되는 예제([그림 2-9](#) 참조)입니다. 이 예에서는 책 저장소의 코드 파일 chap02/TransactionLogCheckPointExample.py를 사용합니다.

Step	Code	Parquet files written	JSON files in _delta_log												
1	<pre>df .coalesce(1) .write .format("delta") .save(DATALAKE_PATH)</pre>	part-00000.parquet	<table border="1"> <tr> <th>Action</th><th>Part name</th></tr> <tr> <td>Add</td><td>part-00000</td></tr> </table> 00000.json	Action	Part name	Add	part-00000								
Action	Part name														
Add	part-00000														
2	<pre># Loop from 0..9 for index in range(9): # Create a patient tuple patientID = 10 + index t = (patientID, f"Patient {patientID}", "Phoenix") # Create and write the dataframe df = spark.createDataFrame([t], columns) df .write .format("delta") .mode("append") .save(DATALAKE_PATH)</pre>	part-00001.parquet part-00002.parquet ... part-00009.parquet	<table border="1"> <tr> <th>Action</th><th>Part name</th></tr> <tr> <td>Add</td><td>part-00001</td></tr> </table> 00001.json <table border="1"> <tr> <th>Action</th><th>Part name</th></tr> <tr> <td>Add</td><td>part-00002</td></tr> </table> 00002.json ... <table border="1"> <tr> <th>Action</th><th>Part name</th></tr> <tr> <td>Add</td><td>part-00009</td></tr> </table> 00009.json	Action	Part name	Add	part-00001	Action	Part name	Add	part-00002	Action	Part name	Add	part-00009
Action	Part name														
Add	part-00001														
Action	Part name														
Add	part-00002														
Action	Part name														
Add	part-00009														
3	<pre>patientID = 100 t = (patientID, f"Patient {patientID}", "Phoenix") df = spark.createDataFrame([t], columns) df .write .format("delta") .mode("append") .save(DATALAKE_PATH)</pre>	part-00010.parquet	<table border="1"> <tr> <th>Action</th><th>Part name</th></tr> <tr> <td>Add</td><td>part-00010</td></tr> </table> 00010.json 00010.checkpoint.parquet	Action	Part name	Add	part-00010								
Action	Part name														
Add	part-00010														
4	<pre>for index in range(2): patientID = 200 + index t = (patientID, f"Patient {patientID}", "Phoenix") df = spark.createDataFrame([t], columns) df .write .format("delta") .mode("append") .save(DATALAKE_PATH)</pre>	part-00012.parquet part-00013.parquet	<table border="1"> <tr> <th>Action</th><th>Part name</th></tr> <tr> <td>Add</td><td>part-00012</td></tr> </table> 00001.json <table border="1"> <tr> <th>Action</th><th>Part name</th></tr> <tr> <td>Add</td><td>part-00013</td></tr> </table> 00002.json	Action	Part name	Add	part-00012	Action	Part name	Add	part-00013				
Action	Part name														
Add	part-00012														
Action	Part name														
Add	part-00013														

그림 2-9. 체크포인트 파일 예시

이 예시에는 다음 단계가 있습니다.

1. 첫 번째 코드 조각은 여러 환자가 포함된 표준 Spark DataFrame을 생성합니다.

DataFrame에 coalesce(1) 트랜잭션을 적용하여 해당 데이터를 하나의 파티션에 강제로 적용합니다.

다음으로, Delta Lake 형식의 DataFrame을 스토리지 파일에 씁니다. 단일 part-0001.parquet 파일이 작성되었는지 확인합니다. 또한 _delta_log 디렉토리에 단일 트랜잭션 로그 항목(00000.json)이 생성된 것을 볼 수 있습니다. 이 디렉토리 항목에는 part-00001.parquet 파일에 대한 파일 추가 작업이 포함되어 있습니다.

2. 다음 단계에서는 범위(0, 9)에 대해 루프를 설정합니다. 이 루프는 9번 반복되어 새 환자 행을 생성한 다음 해당 투플에서 DataFrame을 생성하고 DataFrame을 저장소 파일에 씁니다. 9번 반복하므로 part-00001.parquet부터 part-00009.parquet까지 9개의 추가 Parquet 파일을 생성합니다. 또한 00001.json부터 00009.json까지 9개의 추가 트랜잭션 로그 항목도 표시됩니다.

3. 3단계에서는 환자 투플을 하나 더 생성하고 이를 DataFrame으로 변환한 후 Delta 테이블에 씁니다. 그러면 하나의 추가 데이터 파일(part-00010.parquet)이 생성됩니다.

트랜잭션 로그에는 part-00010.parquet 파일에 대한 파일 추가 작업이 포함된 표준 로그 항목(00010.json)이 있습니다. 그러나 흥미로운 사실은 000010.checkpoint.parquet 파일도 생성한다는 것입니다. 앞서 말씀드린 체크포인트입니다. 10개의 커밋마다 체크포인트가 생성됩니다. 이 Parquet 파일에는 기본 Parquet 형식으로 커밋 당시 테이블의 전체 상태가 포함되어 있습니다.

4. 마지막 단계에서 코드는 part-00011.parquet 및 part-00012.parquet를 생성하는 두 개의 추가 커밋과 이러한 파일을 가리키는 추가 파일 항목이 있는 두 개의 새 로그 항목을 생성합니다.

Delta Lake가 테이블 상태를 다시 생성해야 하는 경우 검사점 파일(000010.checkpoint.parquet)을 읽고 두 개의 추가 로그 항목(00011.json 및 00012.json)을 다시 적용합니다.

체크포인트 파일 표시

checkpoint.parquet 파일을 생성했으므로 /chapter02/readCheckPointFile.py Python 파일을 사용하여 해당 내용을 살펴보겠습니다.

```

# 델타 테이블의 출력 경로를 설정합니다.
DATALAKE_PATH = "/book/chapter02/transactionLogCheckPointExample"
CHECKPOINT_PATH = "/_delta_log/000000000000000010.checkpoint.parquet"
# checkpoint.parquet 파일을 읽습니다.
checkpoint_df = 스파
          \
          \
          \
          \
크.read.format("parquet").load(f"{DATALAKE_PATH}{CHECKPOINT_PATH}")

# 체크포인트 데이터프레임을 표시합니다.
checkpoint_df.show()

```

여기서는 Parquet 형식을 어떻게 수행하는지 확인하세요. 체크포인트 파일은 실제로 델타 형식이 아닌 Parquet 형식으로 저장됩니다.

checkpoint_df DataFrame의 내용은 다음과 같습니다.

전송	추가체거	메타데이터프로토콜
null [part-00000-f7d9f...] null null [part-00000-	널 널	널 널
a65e0... null null [part-00000-4c3ea...] null null	널 널	널 널
{part-00000-8eb1f...} null null [part-00000-2e143...]	널 널	널 널
널 null [part-00000-d1d13...] null null	널 널	널 널
{part-00000-650bf...} null null [part-00000-ea06e...]	널 널	널 널
널 null [part-00000-79258...] null null	널 널	널 널
{part-00000-23558...} null null 널 널 널 널	널 널	널 널
{376ce2d6-11b1-46...} null [part-00000-eb29a...] null	널 널	널 널
널	널 널	널 {1, 2}
		널
		널

보시다시피 체크포인트 파일에는 다양한 작업(추가, 제거, 메타데이터 및 프로토콜). 다양한 Parquet에 대한 파일 추가 작업이 표시됩니다. 데이터 파일, 델타 테이블을 생성했을 때의 메타데이터 업데이트 작업 및 초기 델타 테이블 쓰기로 인한 프로토콜 작업 변경.

DataFrame.show() 는 DataFrame 레코드를 순서대로 표시하지 않습니다. 그만큼 프로토콜 변경 및 메타데이터 업데이트 레코드는 항상 첫 번째 레코드입니다. 체크포인트 파일을 실행한 후 다양한 파일 추가 작업을 수행합니다.

결론

Delta Lake로의 여정을 시작하면 모든 것이 초기 설정부터 시작됩니다. 이 장에서는 로컬 컴퓨터에서 PySpark 및 Spark Scala 셀을 사용하여 Delta Lake를 설정하는 방법을 살펴보고 Delta Lake 확장을 사용하여 PySpark 프로그램을 실행하는 데 필요한 라이브러리와 패키지를 다루었습니다. Databricks와 같은 클라우드 기반 도구를 사용하여 Delta Lake와 같은 Spark 기반 애플리케이션을 개발, 실행 및 공유하면 이 설정 프로세스를 단순화할 수도 있습니다.

Delta Lake를 시작하고 실행하는 방법을 읽은 후 우리는 이 책 전체에서 논의할 핵심 기능 대부분을 필연적으로 활성화하는 Delta Lake의 기본 구성 요소에 대해 배우기 시작했습니다. 확장 가능한 메타데이터를 활성화하는 검사점 파일과 표준 Parquet 파일에 트랜잭션 로그를 추가하여 ACID 트랜잭션을 지원함으로써 Delta Lake는 안정성과 확장성을 지원하는 핵심 요소를 갖습니다.

이제 이러한 기본 구성 요소를 설정했으므로 다음 장에서 델타 테이블의 기본 작업에 대해 자세히 알아보겠습니다.

3 장

델타 테이블의 기본 작업

델타 테이블은 다양한 방법으로 생성할 수 있습니다. 테이블을 만드는 방법은 도구 세트에 대한 친숙도에 따라 크게 달라집니다. 주로 SQL 개발자라면 SQL의 CREATE TABLE을 사용하여 Delta 테이블을 생성할 수 있고, Python 사용자는 DataFrameWriter API 또는 세분화되고 사용하기 쉬운 DeltaTableBuilder API를 선호할 수 있습니다.

테이블을 생성할 때 GENERATED 열을 정의할 수 있습니다. 이 열의 값은 Delta 테이블의 다른 열에 대한 사용자 지정 함수를 기반으로 자동 생성됩니다. 일부 제한 사항이 적용되지만 생성된 열은 델타 테이블 스키마를 강화하는 강력한 방법입니다.

델타 테이블은 표준 ANSI SQL 또는 널리 사용되는 PySpark DataFrameReader API를 사용하여 읽을 수 있습니다. 기본 SQL INSERT 문을 사용하여 Delta 테이블에 쓰거나 테이블에 DataFrame을 추가할 수 있습니다. 마지막으로 SQL COPY INTO 옵션을 활용하는 것은 대량의 데이터를 신속하게 추가하는 좋은 방법입니다.

자주 사용하는 쿼리 패턴을 기반으로 델타 테이블을 분할하면 쿼리 및 DML 성능이 크게 향상될 수 있습니다. 델타 테이블을 구성하는 개별 파일은 분할 열의 값에 맞춰 하위 디렉터리에 구성됩니다.

Delta Lake를 사용하면 사용자 지정 메타데이터를 트랜잭션 로그의 커밋 항목과 연결할 수 있습니다. 이는 감사 목적으로 민감한 커밋에 태그를 지정하는 데 활용할 수 있습니다. 테이블 속성에 사용자 지정 태그를 저장할 수도 있으므로 클라우드 리소스에 대한 태그를 가질 수 있는 것처럼 이제 해당 태그를 델타 테이블과 연결할 수 있습니다. 특정 델타 기능을 수정할 수도 있습니다. 예를 들어 delta.appendonly 속성을 테이블에 연결하여 삭제 및 업데이트를 방지할 수 있습니다.

델타 테이블 생성

Delta Lake를 사용하면 다음 세 가지 방법으로 테이블을 만들 수 있습니다.

SQL 데이터 정의 언어(DDL) 명령 SQL 개발자는 이미 고전적인 CREATE

TABLE 명령에 매우 익숙하므로 이를 사용하여 몇 가지 속성만 추가하여 델타 테이블을 생성할 수 있습니다.

PySpark DataFrameWriter API 빅 데이터

Python(및 Scala) 개발자는 이미 이 API에 매우 익숙할 것이며 계속해서 Delta 테이블과 함께 사용할 수 있을 것입니다.

DeltaTableBuilder API 이는

Delta 테이블용으로 특별히 설계된 새로운 API입니다. 널리 사용되는 Builder 패턴을 사용하며 모든 Delta 테이블과 열 속성에 대해 매우 세밀한 제어를 제공합니다.

다음 섹션에서는 이러한 각 테이블 생성 방법을 직접 실습합니다.

SQL DDL을 사용하여 델타 테이블 생성 Spark 컴퓨팅 환경에서

사용되는 SQL 버전은 [Spark SQL이라고 합니다](#). 이는 Spark에서 지원하는 ANSI SQL의 변형입니다. Spark SQL은 일반적으로 ANSI 표준 SQL과 호환됩니다. [스파크 문서](#)를 참고하세요 Spark SQL 변형에 대한 추가 세부정보를 확인하세요.

앞서 언급했듯이 Spark SQL에서 표준 SQL DDL 명령을 사용하여 델타 테이블을 생성할 수 있습니다.

```
%sql
-- 델타 형식을 지정하고 -- 따옴표로 묶은 경로를 지정하여 델타 테이블을 생성합니다. CREATE TABLE IF NOT
EXISTS delta.`/mnt/datalake/book/
chapter03/rateCard` (
    rateCodeId INT,
    rateCodeDesc STRING
)
델타 사용
```

테이블 이름에 사용하는 표기법은 file_format | 'path_to_table' 표기법. 여기서 file_format은 delta이고 path_to_table은 Delta 테이블의 경로입니다.

1 GitHub 저장소 위치: /Chapter03/02 - CreateDeltaTableWithSql

실제로는 파일 경로가 상당히 길어질 수 있으므로 이 형식을 사용하면 지루해질 수 있습니다. 카탈로그를 사용하면 데이터베이스 .table_name 표기법으로 테이블을 등록할 수 있습니다. 여기서 데이터베이스는 테이블의 논리적 그룹이고 table_name은 테이블의 약어입니다. 예를 들어, 다음과 같이 Taxdb 라는 데이터베이스를 처음 생성한 경우 :

```
%sql
존재하지 않는 경우 데이터베이스 생성 Taxdb;
```

그러면 다음과 같이 위의 테이블을 생성할 수 있습니다.

```
%sql
-- Taxdb 카탈로그를 사용하여 테이블을 생성합니다. CREATE TABLE IF NOT
EXISTS Taxdb.rateCard (
    rateCodeId INT,
    rateCodeDesc STRING
)
델타 사용
위치 '/mnt/datalake/book/chapter03/rateCard'
```

이 시점부터 이 델타 테이블을 Taxdb.rateCard 로 참조할 수 있습니다. 이는 delta./mnt/datalake/book/chapter03/rate Card 또는 더 긴 경로 이름 보다 기억하고 입력하기가 더 쉽습니다. Spark 생태계에서 가장 널리 사용되는 카탈로그는 Hive 카탈로그입니다.

테이블이 생성된 데이터 레이크 위치에서 디렉터리 목록을 실행하면 테이블의 트랜잭션 로그가 포함된 _delta_log 디렉터리를 제외하고 디렉터리가 비어 있음(데이터를 로드하지 않았기 때문에)을 볼 수 있습니다.

```
%sh
ls -al /dbfs/mnt/datalake/book/chapter03/rateCard 총 12 drwxrwxrwx 2 루트 루트
4096 12월 2
일 19:02 . drwxrwxrwx 2 루트 루트 4096 12월 2일 19:02 ..
drwxrwxrwx 2 루트 루트 4096 12월 2일 16:40 _delta_log
```



Databricks Community Edition 환경에서 이를 셀 명령으로 실행하므로 ls 명령에 대한 경로 앞에 /dbfs를 붙여야 합니다.

_delta_log 디렉터리를 열면 첫 번째 트랜잭션 로그 항목이 표시됩니다.

```
%sh
ls -al /dbfs/mnt/datalake/book/chapter03/rateCard/_delta_log 총 15 drwxrwxrwx 2 루트 루트 4096 12 월 2일 19:02.
drwxrwxrwx 2 루트 루트 4096 12월 2일 19:02 ..

-rwxrwxrwx 1 루트 루트 1886 12월 2일 19:02 00000000000000000000000000000000.crc -rwxrwxrwx 1 루트 루트 939 12월 2일 19:02
00000000000000000000000000000000.json
```

2 장의 트랜잭션 로그 논의에서 트랜잭션 로그 항목에 기록될 수 있는 다양한 작업에 대해 읽었습니다. 이러한 작업 중 하나는 테이블의 스키마, 분할 열(해당되는 경우) 및 기타 정보를 설명하는 메타데이터 작업입니다. 이 메타데이터 작업은 항상 새 테이블에 대해 생성된 첫 번째 트랜잭션 로그 항목에 기록됩니다.

이 메타데이터 작업을 찾으려면 트랜잭션 항목에서 문자열 메타데이터를 검색하면 됩니다.

```
%sh
grep 메타데이터 /dbfs/mnt/datalake/book/chapter03/rateCard
/_delta_log/00000.json > /tmp/metadata.json
python -m json.tool /tmp/metadata.json
```

그러면 다음과 같은 출력이 생성됩니다.

```
{
  "metaData": { "id": "f79c4c11-a807-49bc-93f4-2bbe778e2a04", "형식": {
    "공급자": "마루", "옵션": {}

  },
  "schemaString": "{ \"type\": \"struct\", \"fields\": [{\"name\": \"rateCodeId\", \"type\": \"integer\", \"nullable\": true,
  \"metadata\": {}}, {"name": \"rateCodeDesc\", \"type\": \"string\",
  \"nullable\": true, \"metadata\": {}}], \"partitionColumns\": [],
  \"configuration\": {}, \"createdTime\": 1670007736533
}
}
```

여기서는 Delta Lake가 일부 감사 및 분할 정보와 함께 테이블의 스키마를 트랜잭션 로그 항목에 기록한 것을 볼 수 있습니다.



이전 명령에서는 먼저 트랜잭션 로그 항목에서 문자열 메타데이터를 검색하는 grep 명령을 수행했습니다.

그런 다음 그 출력을 임시 파일에 씁니다. 다음 줄에서는 임시 파일과 함께 python -m json.tool을 입력으로 사용합니다. json.tool Python 모듈은 JSON 파일의 내용을 "예쁘게 인쇄"하므로 가독성이 매우 좋습니다.

DESCRIBE 문

SQL DESCRIBE 명령을 사용하여 Parquet 파일 또는 델타 테이블에 대한 기본 메타데이터를 반환할 수 있습니다. 테이블에 대해 반환된 메타데이터에는 각 열에 대해 다음 정보가 포함된 한 줄이 포함됩니다.

- 열 이름
- 열 데이터 유형 • 열에 적용된 모든 설명

다음은 테이블 수준의 DESCRIBE 명령 예입니다 .

```
%sql
DESCRIBE TABLE Taxdb.rateCard;

+-----+-----+-----+
| 열_이름          | 데이터_유형 | 댓글 |
+-----+-----+-----+
| rateCodeId | 정수 | <날> || <날> |
| rateCodeDesc | 끈 |
+-----+-----+-----+
```

Delta Lake 관련 특성을 찾으려면 다음을 사용할 수도 있습니다.

DESCRIBE TABLE EXTENDED 명령은 다음과 같은 일반 속성을 포함하여 보다 자세한 메타데이터 정보를 제공합니다.

- 테이블이 생성된 데이터베이스의 카탈로그 이름(이 경우 하이브 메타스토어)
- Hive 데이터베이스
- 테이블 이름
- 기본 파일의 위치 • 테이블 소유자
- 테이블 속성

다음 Delta Lake 관련 속성도 포함됩니다.

delta.minReaderVersion

읽을 수 있는 리더에 필요한 최소 프로토콜 리더 버전입니다.
이 델타 테이블.

delta.minWriterVersion

여기에 쓸 수 있는 작성자에게 필요한 최소 프로토콜 작성자 버전입니다.
델타 테이블. [Delta Lake 설명서](#)를 참조하세요. 모든 항목의 전체 목록을 보려면
사용 가능한 테이블 속성

다음은 DESCRIBE TABLE EXTENDED 명령의 예입니다.

```
%sql
확장된 테이블 설명 Taxdb.rateCard;
```

다음 출력이 생성됩니다.

열_이름	데이터 형식	댓글
rateCodeId	정수 <널>	
rateCodeDesc	문자열 <널>	
##		
# 테이블 상세정보		
카탈로그 hive_metastore		
데이터베이스 택시DB		
테이블 요율표		
유형 외부		
위치 dbfs:/../chapter03/rateCard		
공급자 델타		
소유자 루트		
테이블 속성 [delta.minReaderVersion=1,		
delta.minWriterVersion=2]		

지금까지 SQL DDL을 사용하여 Delta 테이블 생성을 다루었습니다. 다음에
섹션에서는 Python으로 다시 전환하여 친숙한 코드를 어떻게 사용할 수 있는지 살펴보겠습니다.
PySpark DataFrames를 사용하여 새 델타 테이블을 생성합니다.

DataFrameWriter API를 사용하여 델타 테이블 생성

Spark DataFrame은 관계형 데이터베이스 테이블이나 Excel 스프레드시트와 유사합니다.
헤더. 데이터는 다양한 데이터 유형의 행과 열에 있습니다. 컬렉션
DataFrame을 읽고, 쓰고, 조작할 수 있는 함수는 총체적으로 알려져 있습니다.
Spark DataFrameWriter API로.

관리되는 테이블 만들기



Spark 및/또는 Delta 설명서를 읽으면 관리되는 테이블과 관리되지 않는 테이블이라는 용어를 듣게 됩니다. 위치를 사용하여 생성된 델타 테이블을 관리되지 않는 테이블이라고 합니다. 이러한 테이블의 경우 Spark는 메타데이터만 관리하며 사용자에게 테이블의 기본 데이터를 저장할 정확한 위치를 지정하거나 테이블을 생성하기 위해 데이터를 가져올 소스 디렉터리를 지정하도록 요구합니다(DataFrameWriter API 를 사용하는 경우).

위치 없이 생성된 델타 테이블을 관리형 테이블이라고 합니다. Spark는 관리되는 테이블의 메타데이터와 실제 데이터를 모두 관리합니다. 데이터는 관리형 테이블의 기본값인 /spark-warehouse 하위 폴더 (Apache Spark 사나리오의 경우) 또는 /user/hive/warehouse 폴더(Databricks에서 실행하는 경우)에 저장됩니다.

DataFrameWriter API 의 이점 중 하나 는 다음 코드 조각에 표시된 것처럼 Spark DataFrame에서 동시에 테이블을 생성하고 테이블에 데이터를 삽입할 수 있다는 것입니다.

```

INPUT_PATH = '/databricks-datasets/nyctaxi/taxizone/taxi_rate_code.csv'
DELTALAKE_PATH = \
    'dbfs:/mnt/datalake/book/chapter03/createDeltaTableWithDataFrameWriter'

# 입력 경로에서 DataFrame을 읽습니다. df_rate_codes = Spark .read
\ \
\ \
\ \
    .format("csv") .option("inferSchema",
True) .option("header",
True) .load(INPUT_PATH)

# DataFrame을 관리형 Hive 테이블로 저장합니다
df_rate_codes.write.format("delta").saveAsTable('taxidb.rateCard')

```

여기서는 먼저 Taxi_rate_code.csv 파일에서 DataFrame을 채운 다음 .format("delta") 옵션을 지정하여 DataFrame을 Delta 테이블로 저장합니다. 테이블의 스키마는 DataFrame의 스키마가 됩니다. 데이터 파일의 위치를 지정하지 않았으므로 이 테이블은 관리되는 테이블이 됩니다. SQL DESCRIBE TABLE EXTENDED 명령을 실행하여 이를 확인할 수 있습니다.

² GitHub 저장소 위치: /chapter03/04 - DataFrameWriter API

```
%sql
확장된 테이블 설명 Taxdb.rateCard;
+-----+-----+
| 열_이름 | 데이터 형식 |
+-----+-----+
| 요금코드ID | 정수 |
| 요금코드설명 | 문자열 |
|
| # 테이블 상세정보 ||
| 카탈로그 | hive_metastore |
| 데이터베이스 | 택시DB |
| 테이블 | 요율표 |
| 유형 | 관리됨 |
| 위치 | dbfs:/user/hive/warehouse/taxidb.db/ratecard |
| 공급자 | 델타 |
| 소유자 | 루트 |
| ls_managed_location | 사실 |
| 테이블 속성 | [delta.minReaderVersion=1, |
| | delta.minWriterVersion=2] |
+-----+-----+
```

테이블의 데이터가 /user/hive/warehouse 위치에 있다는 것을 알 수 있습니다.

테이블 유형은 MANAGED로 설정됩니다.

테이블에 대해 SELECT를 실행하면 실제로 데이터가 성공적으로 로드된 것을 확인할 수 있습니다.
CSV 파일에서:

```
%sql
SELECT * Taxdb.rateCard에서
+-----+-----+
| 요금코드ID | 요율 코드 설명 |
+-----+-----+
| 1 | 표준요금 |
| 2 | JFK |
| 3 | 뉴어크 |
| 4 | 나소 또는 웨스트체스터 |
| 5 | 협상 요금 |
| 6 | 그룹 라이딩 |
+-----+-----+
```

관리되지 않는 테이블 만들기

경로와 이름을 모두 지정하여 관리되지 않는 테이블을 생성할 수 있습니다.

델타 테이블. 다음 코드에서는 두 단계를 차례로 실행합니다. 먼저,

기존 테이블:

```
%sql
-- 기존 테이블 삭제
존재하는 경우 테이블 삭제 Taxdb.rateCard;
```

다음으로 테이블을 작성하고 생성합니다.

```
# 다음으로, 두 가지를 모두 지정하여 Delta 테이블을 만듭니다.
# 경로 및 Delta 테이블 N=name
df_rate_codes \
    .쓰다 \
    .format("델타") \
    .mode("덮어쓰기") \
    .option('경로', DELTALAKE_PATH) \
    .saveAsTable('taxidb.rateCard')
```

다시 간단한 SELECT를 수행하여 DataFrame의 데이터가 로드되었습니다:

```
%sql
SELECT * Taxdb.rateCard에서

+-----+-----+
| 요금코드ID | 요율 코드 설명 |
+-----+-----+
| 1 | 표준요금 |
| 2 | JFK |
| 3 | 뉴어크 |
| 4 | 나소 또는 웨스트체스터 |
| 5 | 협상 요금 |
| 6 | 그룹 라이딩 |
+-----+-----+
```

DeltaTableBuilder API를 사용하여 델타 테이블 생성

델타 테이블을 만드는 마지막 방법은 DeltaTableBuilder API를 사용하는 것입니다. 부터 Delta 테이블과 함께 작동하도록 설계되었으며 더 높은 수준의 세분화된 테이블을 제공합니다. 기존 DataFrameWriter API와 비교하여 제어합니다. 사용자가 지정하기가 더 쉽습니다. 열 설명, 테이블 속성, 생성된 정보 등의 추가 정보 열.

Builder 디자인 패턴은 소프트웨어 언어에서 널리 사용됩니다. 빌더 패턴의 목표 “복잡한 객체의 구성과 표현을 분리하여 동일한 건설 과정이 다른 표현을 만들어낼 수 있습니다.” 그것은 건설하는 데 사용됩니다 복잡한 객체를 단계별로 생성하며 마지막 단계에서는 객체를 반환합니다.

이 경우 우리가 만들고 있는 복잡한 개체는 델타 테이블입니다. 델타 테이블 지원 옵션이 너무 많아서 인수가 많은 표준 API를 설계하기가 어렵습니다. 단일 기능의 경우. 대신 DeltaTableBuilder 에는 여러 가지 작은 메소드가 있습니다. addColumn() 과 같은 ods는 모두 Builder 개체에 대한 참조를 반환합니다. 저것 addColumn()에 대한 다른 호출이나 Builder의 다른 메소드를 계속 추가할 수 있습니다. 우리가 호출하는 마지막 메소드는 받은 모든 속성을 수집하는 실행()입니다. Delta 테이블을 생성하고 테이블에 대한 참조를 호출자에게 반환합니다. 사용하려면 DeltaTableBuilder를 사용하려면 다음 가져오기가 필요합니다.

delta.tables에서 가져오기 *

이 예에서는 관리되는 테이블을 생성합니다.³

```
# 이 Create Table에서는 위치를 지정하지 않으므로
# 관리되는 테이블 생성
DeltaTable.createIfNotExists(스파크) \
    .tableName("taxidb.greenTaxis") \
    .addColumn("Rideld", "INT", comment = "기본 키") \
    .addColumn("VendorId", "INT", comment = "라이드 공급업체") \
    .addColumn("이벤트 유형", "STRING") \
    .addColumn("픽업시간", "TIMESTAMP") \
    .addColumn("PickupLocationId", "INT") \
    .addColumn("CabLicense", "STRING") \
    .addColumn("DriversLicense", "STRING") \
    .addColumn("PassengerCount", "INT") \
    .addColumn("DropTime", "TIMESTAMP") \
    .addColumn("DropLocationId", "INT") \
    .addColumn("RateCodeId", "INT", comment = "RateCard 참조") \
    .addColumn("결제 유형", "INT") \
    .addColumn("TripDistance", "DOUBLE") \
    .addColumn("TotalAmount", "DOUBLE") \
    .실행하다()
```

각 메소드는 Builder 객체에 대한 참조를 반환하므로 호출을 계속할 수 있습니다.

.addColumn()을 사용하여 각 열을 추가합니다. 마지막으로 .execute()를 호출하여 Delta를 생성합니다. 테이블.

생성된 열

Delta Lake는 특수한 유형의 열인 생성된 열을 지원합니다.

그 중 다른 것보다 사용자가 지정한 기능을 기반으로 자동 생성됩니다.

델타 테이블의 열입니다. 생성된 열이 있는 Delta 테이블에 쓰는 경우 명시적으로 값을 제공하지 않으면 Delta Lake가 자동으로 가치.

다음에는 예제를 만들어 보겠습니다. 택시 테마를 유지하기 위해 간단한 노란색 택시 테이블 버전:

```
%sql
CREATE TABLE Taxdb.YellowTaxis
(
    탑승 ID          정수      COMMENT '이것은 기본 키 열입니다.',
    공급업체 ID      정수,
    데리러 갈 시간   타임스탬프,
    픽업 연도         정수      항상 생성됨(연도(픽업 시간)),
    픽업월           정수      항상 생성됨(월(픽업 시간)),
    픽업일           정수      항상 생성됨(DAY(픽업 시간)),
    드롭타임         타임스탬프,
    택시 번호        끈       COMMENT '공식 엘로케이션 번호'
```

³ GitHub 저장소 위치: /chapter03/05 - DeltaTableBuilder API

```
) 델타 위치 "/mnt/
datalake/book/chapter03/YellowTaxis.delta" 사용
COMMENT '노란택시 데이터를 저장하는 테이블'
```

PickupTime 열에서 YEAR, MONTH 및 DAY를 추출하는 GENERATED ALWAYS AS 가 있는 열이 표시됩니다 . 레코드를 삽입하면 이러한 열의 값이 자동으로 채워집니다.

```
%sql
INSERT INTO Taxdb.YellowTaxis (RidId,
VendorId, PickupTime, DropTime, CabNumber)
 가치
(5, 101, '2021-7-1T8:43:28UTC+3', '2021-7-1T8:43:28UTC+3', '51-986')
```

레코드를 선택하면 생성된 열이 자동으로 채워지는 것을 볼 수 있습니다.

```
%sql
SELECT PickupTime, PickupYear, PickupMonth, PickupDay FROM Taxdb.YellowTaxis

+-----+-----+-----+-----+
| 데리러 갈 시간           | 픽업연도 | 픽업월 | 픽업데이 |
+-----+-----+-----+
| 2021-07-01 05:43:28+00:00 | 2021    | 7      | 1      |
+-----+-----+-----+
```

생성된 열에 대한 값을 명시적으로 제공하는 경우 같은 제약 조건 (<값> ⇌ 생성 식) IS TRUE를 충족해야 합니다 . 그렇지 않으면 오류가 발생하여 삽입이 실패합니다.

GENERATED ALWAYS AS 에서 사용하는 표현식은 동일한 인수 값이 주어지면 항상 동일한 결과를 반환하는 모든 Spark SQL 함수일 수 있습니다. 몇 가지 예외는 곧 다루겠습니다. GENERATED 열을 사용하여 다음과 같은 고유 ID가 있는 열을 생성 할 수 있다고 생각할 수도 있습니다 .

```
%sql
CREATE OR REPLACE TABLE default.dummy(
  ID 문자열은 항상 (UUID())로 생성됩니다.
  이름 STRING
) 델타 사용
```

그러나 이를 실행하려고 하면 다음과 같은 오류 메시지가 나타납니다.

uuid()를 찾았습니다. 생성된 열은 비결정적 표현식을 사용할 수 없습니다.

UUID () 함수는 호출마다 다른 값을 반환하며 이는 이전 규칙을 위반합니다. 다음 유형의 함수에는 이 규칙에 대한 몇 가지 예외가 있습니다.

- 사용자 정의 함수

- 집계 함수
- 창 기능
- 여러 행을 반환하는 함수

GENERATED ALWAYS AS 열은 나열된 함수를 사용하여 유효하며 특정 레코드 샘플의 표준 편차 계산과 같은 여러 시나리오에서 매우 유용할 수 있습니다.

델타 테이블 읽기

테이블을 읽을 때 DataFrameReader를 사용하는 SQL 및 PySpark라는 몇 가지 옵션이 있습니다. Databricks Community Edition에서 노트북을 사용할 때 노트북 내에서 SQL과 PySpark 셀을 모두 사용하는 경향이 있습니다. 빠른 SELECT 와 같은 일부 작업은 SQL에서 더 쉽고 빠르게 수행할 수 있는 반면, 복잡한 작업은 때때로 PySpark 및 DataFrameReader에서 더 쉽게 표현됩니다. 물론 이는 엔지니어의 경험과 선호도에 따라 달라집니다. 현재 해결 중인 문제에 따라 두 가지를 건전하게 혼합하여 사용하는 실용적인 접근 방식을 권장합니다.

SQL을 사용하여 델타 테이블 읽기

불을 읽으려면 간단히 SQL 셀을 열고 SQL 쿼리를 작성하면 됩니다. GitHub READ.ME 파일에 지정된 대로 환경을 설정하면 /mnt/datalake/book/chapter03/YellowTaxisDelta 폴더에 Delta 테이블이 생깁니다.

```
%sh
ls -al /dbfs/mnt/datalake/book/chapter03/YellowTaxisDelta
drwxrwxrwx 2 루트 루트 drwxrwxrwx 4096 12월 4일 18:04 .
drwxrwxrwx 2 루트 루트 drwxrwxrwx 4096 12월 2일 19:02 ..
2 루트 루트 4096 12월 4일 16:41 _delta_log
-rwxrwxrwx 1 루트 루트 134759123 12월 4일 18:04 part-00000...c000.snappy.parquet
-rwxrwxrwx 1 루트 루트 107869302 12월 4일 18:04
part-00001...c000.snappy.parquet
```

다음과 같이 메타스토어에 델타 테이블 위치를 빠르게 등록할 수 있습니다.⁴

```
%sql
CREATE TABLE Taxdb.YellowTaxis
    USING DELTA
    LOCATION "/mnt/datalake/book/chapter03/YellowTaxisDelta";
```

테이블을 생성한 후에는 레코드 수를 빠르게 계산할 수 있습니다.

```
%sql
SELECT COUNT(*)
```

⁴ GitHub 저장소 위치: /chapter03/07 - SQL로 테이블 읽기

세다(*)
에서
Taxdb.yellowtaxis

이는 다음과 같은 개수를 제공합니다.

```
+-----+
| 개수(1) |
+-----+
| 9999995 |
+-----+
```

작업할 행이 거의 천만 개에 달하는 것을 볼 수 있습니다. 우리는 다른 것을 사용할 수 있습니다

테이블의 세부 정보를 가져오는 DESCRIBE 명령 변형:

```
%sql
테이블 형식 설명 Taxdb.YellowTaxis;
```

DESCRIBE TABLE FORMATTED 는 출력 형식을 지정하여 좀 더 읽기 쉽게 만듭니다.

열_이름	데이터 형식
탑승 ID 정수	
공급업체 ID 정수	
피업시간 타임스탬프	
드롭타임 타임스탬프	
피업 위치 ID 정수	
삭제 위치 ID 정수	
택시번호 문자열	
운전연하번호 문자열	
승객수 정수	
여행거리 더블	
요금 코드 ID 정수	
결제 유형 정수	
총금액 더블	
요금 더블	
추가 더블	
Mta세 더블	
팁금액 더블	
통행료금액 더블	
개선할증료 더블	
# 테이블 상세정보	
카탈로그 hive_metastore	
데이터베이스 택시DB	
테이블 엘로우택시	
유형 외부	
위치 dbfs:/../chapter03/YellowTaxisDelta	
공급자 엘타	
소유자 루트	
테이블 속성 [delta.minReaderVersion=1,	
delta.minWriterVersion=2]	

Spark SQL은 대부분의 ANSI SQL 하위 집합을 지원하므로 모든 유형의 복잡한 쿼리를 사용할 수 있습니다. 다음은 \$50를 초과하는 가장 비싼 FareAmounts를 포함하는 CabNumbers를 반환하는 예입니다.

```
%sql
선택하다

CabNumber,
AVG(FareAmount) AS AverageFare FROM

Taxdb.yellowtaxis
그룹 기준
    택시 번호
가지고 있는
    AVG(요금) > 50
주문
    2 내림차순
제한 5

이는 우리에게 다음을 제공합니다.

+-----+
| 택시번호 | 평균운임 |
+-----+
| SIR104 | 111.5 | T628190C |
109.0 | 평화16 | 89.7 | T439972C |
89.5 | T802013C | 85.0

+-----+
```

표준 SQL을 인수로 사용하여 `Spark.sql`을 사용하여 Python에서 직접 SQL을 사용할 수도 있습니다. 다음은 이전 SQL 쿼리와 동일한 쿼리를 수행하는 간단한 Python 코드 조각입니다.

```
number_of_results = 5 sql_statement
= f"""
선택하다

택시번호,
AVG(FareAmount) AS AverageFare
에서
Taxdb.yellowtaxis GROUP BY
CabNumber

가지고 있는
    AVG(요금) > 50
주문
    2 내림차순
LIMIT {number_of_results}"""

df = 스파크.sql(sql_statement) 디스플레이(df)
```

이는 SQL과 동일한 결과를 생성합니다.

택시번호	평균운임
SIR104	111.5
109.0	평화16 89.7
T439972C	89.5
T802013C	85.0

문자열을 쉽게 확장할 수 있도록 삼중따옴표 구문을 사용하는 것이 좋습니다.
 연속 라인을 사용하지 않고 여러 라인에 걸쳐. 또한, 우리가 어떻게
 변수 `number_of_results`를 갖고 삼중따옴표 문자열을 다음으로 변환합니다.
`f`-문자열을 사용하고 {} 구문을 사용하여 제한에 대한 변수를 삽입합니다.

PySpark로 테이블 읽기

PySpark에서 동일한 테이블을 읽으려면 `DataFrameReader`를 사용할 수 있습니다. 예를 들어,
 레코드 수를 구현하려면 다음을 사용합니다.5

```
df = Spark.read.format("delta").table("taxidb.YellowTaxis")
print(f"레코드 수: {df.count():,}")
```

산출:

기록수: 9,999,995

테이블은 Delta 테이블이고 다음을 사용할 수 있으므로 Delta 형식을 지정합니다.
`.table()` 메서드를 사용하여 전체 테이블을 읽도록 지정합니다. 마지막으로 우리는
`f`-문자열, 이번에는 ":" 포맷터를 사용합니다. 이는 모든 항목에 쉼표 구분 기호를 사용합니다.
 세 자리.

다음으로, 택시 번호별로 상위 5개 평균 요금에 대한 코드를 다시 생성해 보겠습니다.
 이전에 SQL에서 그랬습니다. 파일 코드는 다음과 같습니다:

```
# 사용하려는 기능을 반드시 가져와야 합니다.
from pyspark.sql import functions as col, avg, desc

# YellowTaxis를 DataFrame으로 읽어옵니다.
df = Spark.read.format("delta").table("taxidb.YellowTaxis")

# GROUP BY, 평균(AVG), HAVING 및 등가물 기준 정렬을 수행합니다.
# pySpark에서
결과 = df.groupBy("CabNumber") \
    .agg(avg("FareAmount").alias("AverageFare")) \
    .filter(col("AverageFare") > 50) \
    .sort(col("AverageFare").desc()) \
    .take(5)
```

5 GitHub 저장소 위치: /chapter03/PySpark로 테이블 읽기

```
# 결과를 인쇄합니다. 이는 DataFrame이 아니라 목록이므로 # 단일 행에 결과를 출력하려면 목록 이해를 사용합니다.
[print(result) for result in results]
```

우리는 다음과 같은 결과를 얻게 될 것입니다:

```
행(택시번호='SIR104', AverageFare=111.5)
행(택시번호='T628190C', 평균요금=109.0)
행(택시번호='PEACE16', AverageFare=89.7)
행(택시번호='T439972C', 평균요금=89.5)
행(택시번호='T802013C', 평균요금=85.0)
```

간단히 `groupBy()` 함수를 사용하여 열별로 그룹화할 수 있습니다.



이 코드 조각에 표시된 대로 결과는 더 이상 DataFrame이 아니라 pyspark.sql.GroupedData 인스턴스입니다.

```
# groupBy를 수행하고 print(type(df.groupBy("CabNumber"))) 유형을 인쇄
합니다.
```

이것은 다음과 같이 인쇄됩니다:

```
<클래스 'pyspark.sql.group.GroupedData'>
```

종종 PySpark를 처음 접하는 개발자는 `groupBy()`가 DataFrame을 반환한다고 가정 할 수 있지만 이는 GroupedData 인스턴스를 반환하므로 `avg()` 및 `filter()` 와 같은 DataFrame 함수 대신 `agg()` 및 `filter()` 와 같은 GroupedData 메서드를 사용해야 합니다. 어디().

평균을 계산하려면 먼저 `.agg()` 메서드를 사용해야 합니다. 메서드 내에서 계산하려는 집계를 지정할 수 있습니다. 이 경우에는 `.avg()` (평균)입니다. Python에서 HAVING 조건에 해당하는 것은 `.filter()` 메서드이며, 이 메서드 내에서 필터 표현식을 사용하여 필터를 지정할 수 있습니다. 마지막으로 `.sort()` 메서드를 사용하여 데이터를 정렬 한 다음 `.take()`를 사용하여 처음 5개 결과를 추출합니다. `.take()` 함수는 Python 목록을 반환합니다. 여기에 목록이 있으므로 목록 이해를 사용하여 목록의 각 결과를 출력할 수 있습니다.

델타 테이블에 쓰기

델타 테이블에 쓰는 방법은 다양합니다. 전체 테이블을 다시 작성하거나 테이블에 추가할 수도 있습니다. 업데이트 및 병합과 같은 고급 주제는 4 장에서 논의됩니다.

먼저 YellowTaxis 테이블을 정리하여 깨끗한 상태를 만든 다음 기존 SQL INSERT 문을 사용하여 데이터를 삽입 합니다. 다음으로 더 작은 CSV 파일의 데이터를 추가하겠습니다. 덮어쓰기 모드에 대해서도 간단히 살펴보겠습니다.

Delta 테이블을 작성할 때 마지막으로 SQL COPY INTO 기능을 사용하여 큰 CSV 파일로 병합합니다.

YellowTaxis 테이블 청소

CREATE TABLE 문 을 사용하여 Delta 테이블을 다시 만들 수 있습니다 .6

```
%sql
CREATE TABLE Taxdb.YellowTaxis
(
    탑승 ID          정수,
    공급업체 ID      정수,
    데리러 갈 시간    타임스탬프,
    드롭타임          타임스탬프,
    픽업 위치 ID      정수,
    삼체 위치 ID      정수,
    택시 번호          끈,
    운전면허번호      끈,
    승객수            정수,
    여행거리          더블,
    요금 코드 ID      정수,
    결제 유형          정수,
    총금액            더블,
    운임금액          더블,
    추가의            더블,
    MtaTax             더블,
    팁금액            더블,
    통행료금액        더블,
    개선할증료        더블
) 델타 사용
위치 "/mnt/datalake/book/chapter03/YellowTaxisDelta"
```

테이블이 설정되면 데이터를 삽입할 준비가 되었습니다.

SQL INSERT를 사용하여 데이터 삽입

YellowTaxis Delta 테이블에 레코드를 삽입하려면 SQL INSERT를 사용할 수 있습니다.

명령:

```
%sql
Taxdb.yellowtaxis에 삽입하세요.
(Rideld, VendorId, PickupTime, DropTime,
PickupLocationId, DropLocationId, CabNumber,
DriverLicenseNumber, PassengerCount, TripDistance,
RatecodeId, 결제 유형, TotalAmount,
FareAmount, Extra, MtaTax, TipAmount,
통행료금액, 개선할증료)
```

6 GitHub 저장소 위치: /chapter03/10 - 델타 테이블에 쓰기

```
VALUES(9999995, 1, '2019-11-01T00:00:00.000Z',
      '2019-11-01T00:02:23.573Z', 65, 71, 'TAC304', '453987', 2, 4.5, 1, 1,
      20.34, 15.0, 0.5, 0.4, 2.0, 2.0, 1.1)
```

그리면 하나의 행이 삽입됩니다.

num_affected_rows	num_inserted_rows
1	1

SQL SELECT 문과 WHERE 절을 사용하여 데이터가 올바르게 로드되었는지 확인하세요.
삽입된 RideId의 경우:

```
%sql
SELECT 개수(RideId) AS 개수 FROM Taxdb.YellowTaxis WHERE RideId = 9999995
```

산출:

카운트
1

출력에는 모든 데이터가 올바르게 로드되었음을 표시합니다.

테이블에 DataFrame 추가 이제 테이블에 DataFrame

을 추가해 보겠습니다. 이 경우 CSV 파일에서 DataFrame을 로드합니다. 데이터를 올바르게 로드하기 위해 스키마를 추론하고 싶지 않습니다.
대신 우리는 정확하다고 알고 있는 YellowTaxis 테이블의 스키마를 사용할 것입니다.

테이블에서 DataFrame을 로드하여 스키마를 쉽게 추출할 수 있습니다.

```
df = Spark.read.format("delta").table("taxidb.YellowTaxis") yellowTaxiSchema
= df.schema print(yellowTaxiSchema)
```

이는 테이블 스키마가 다음과 같다는 것을 보여줍니다.

뿌리

- |-- RideId: 정수(null 가능 = true)
- |-- VendorId: 정수(null 가능 = true)
- |-- PickupTime: 타임스탬프(null 허용 = true)
- |-- DropTime: 타임스탬프(null 허용 = true)
- |-- PickupLocationId: 정수(null 허용 = true)
- |-- DropLocationId: 정수(null 허용 = true)
- |-- CabNumber: 문자열(null 허용 = true)
- |-- DriverLicenseNumber: 문자열(null 허용 = true)

```

|-- PassengerCount: 정수(null 가능 = true)
|-- TripDistance: double(null 가능 = true)
|-- RatecodeId: 정수(null 허용 = true)
|-- 지불 유형: 정수(null 허용 = true)
|-- TotalAmount: double(null 가능 = true)
|-- FareAmount: double(null 가능 = true)
|-- 추가: double(null 허용 = true)
|-- MtaTax: double(null 허용 = true)
|-- TipAmount: double(null 허용 = true)
|-- TollsAmount: double(null 가능 = true)
|-- ImprovementSurcharge: double(null 가능 = true)

```

이제 스키마가 있으므로 다음에서 새 DataFrame (df_for_append) 을 로드할 수 있습니다.

추가된 CSV 파일:

```

df_for_append = 스팩.읽기 \
    .option("헤더", "true") \
    .schema(yellowTaxiSchema) \
    .csv("/mnt/datalake/book/데이터 파일/YellowTaxis_append.csv")
표시(df_for_append)

```

다음 출력이 표시됩니다(부분 출력이 표시됨).

탑승 ID	공급업체 ID	데리러 갈 시간	드롭타임
9999996	1	2019-01-01T00:00:00	2022-03-01T00:13:13
9999997	1	2019-01-01T00:00:00	2022-03-01T00:09:21
9999998	1	2019-01-01T00:00:00	2022-03-01T00:09:15
9999999	1	2019-01-01T00:00:00	2022-03-01T00:10:01

이제 VendorId가 1 인 4개의 추가 행이 있습니다. 이제 이를 추가할 수 있습니다.
델타 테이블에 대한 CSV 파일:

```

df_for_append.write.mode("추
가").format("델
타").save("/mnt/
datalake/book/chapter03/YellowTaxisDelta")

```

그리면 데이터가 Delta 테이블에 직접 추가됩니다. 테이블에 하나의 행이 있었기 때문에
INSERT 문에서 4개의 추가 행을 삽입하기 전에 우리는 다음을 알고 있습니다.
이제 YellowTaxis 테이블에 5개의 행이 있습니다.

```
%sql
선택하다
세다(*)
```

Taxdb.YellowTaxis에서	
+	-----+
개수(1)	
+	-----+
5	
+	-----+

이제 5개의 행이 있습니다.

델타 테이블에 쓸 때 덮어쓰기 모드 사용

이전 예에서는 DataFrameWriter API를 사용하여 Delta 테이블에 쓸 때 .mode("append")을 사용했습니다. Delta Lake는 Delta 테이블에 쓸 때 덮어쓰기 모드도 지원합니다. 이 모드를 사용하면 테이블의 모든 데이터가 원자적으로 대체됩니다.

이전 코드 블록에서 .mode("overwrite")를 사용했다면 전체 YellowTaxis Delta 테이블을 df_for_append DataFrame 으로 덮어썼을 것입니다.

코드에서 .mode("overwrite")를 사용하더라도 이전 부품 파일은 즉시 물리적으로 삭제되지 않습니다. 시간 여행과 같은 기능을 지원하기 위해 이러한 파일은 즉시 삭제할 수 없습니다. 나중에 더 이상 필요하지 않다고 확신하는 경우 VACUUM 과 같은 명령을 사용하여 이러한 파일을 물리적으로 삭제할 수 있습니다. 시간 여행과 VACUUM 명령은 6 장에서 다룹니다.

SQL COPY INTO 명령을 사용하여 데이터 삽입 SQL COPY INTO 명령을 사

용하여 테이블에 데이터를 추가할 수 있습니다. 이 명령은 매우 많은 양의 데이터를 빠르게 추가해야 할 때 특히 유용합니다.

다음 명령을 사용하여 CSV 파일의 데이터를 추가할 수 있습니다.

```
%sql
COPY INTO Taxdb.yellowtaxis FROM (SELECT
    RideId:Int
    , 공급업체 ID:Int
    , 픽업시간::타임스탬프
    , DropTime::타임스탬프
    , PickupLocationId:Int
    , DropLocationId:Int
    , 택시번호::문자열
    , 드라이버아이덴티티번호::문자열
    , PassengerCount:Int
    , TripDistance:더블
    , RateCodeId:Int
```

```

    , 지불 유형::Int
    , 총금액::더블
    , FareAmount::더블
    , 엑스트라::더블
    , MtaTax::더블
    , TipAmount::더블
    , 통행료 금액::두 배
    , 개선할증료::더블

    '/mnt/datalake/book/DataFiles/YellowTaxisLargeAppend.csv'에서
)
파일 형식 = CSV
FORMAT_OPTIONS("헤더" = "참")

```

CSV 파일의 모든 필드는 문자열이므로 특정 유형의 스키마를 제공해야 합니다.
 데이터를 로드할 때 SQL SELECT 문을 사용합니다. 이는 각 유형을 제공합니다.
 열을 통해 올바른 스키마를 로드하고 있는지 확인합니다. FILEFORMAT 은
 이 경우 CSV가 지정됩니다. 마지막으로 파일에 헤더가 있으므로 다음을 지정해야 합니다.
 FORMAT_OPTIONS가 포함된 헤더입니다 .

이 문의 출력은 다음과 같습니다.

num_affected_rows	num_inserted_rows
9999995	9999995

단 몇 초 만에 거의 천만 개의 행을 삽입한 것을 볼 수 있습니다. 사본
 INTO 명령은 또한 이전에 로드된 파일을 추적하고 다시 로드하지 않습니다. 우리
 COPY INTO 명령을 다시 실행하여 이를 테스트할 수 있습니다 .

num_affected_rows	num_inserted_rows
0	0

보시다시피 추가 행이 로드되지 않았습니다. 마지막으로 마지막 행을 확인하면
 계산해 보면 이제 백만 개의 행이 있음을 알 수 있습니다.

```

%sql
선택하다
세다(*)
에서
Taxdb.YellowTaxis

```

산출:

```
+-----+
| 개수(1) |
+-----+
| 10000000 |
+-----+
```

파티션

델타 테이블은 표준 쿼리 패턴을 사용하여 액세스하는 경우가 많습니다. 예를 들어, IoT 시스템의 데이터는 일별, 시간별, 심지어 분 단위로 액세스되는 경향이 있습니다. 노란색 택시 데이터를 쿼리하는 분석가는 VendorId 등을 기준으로 데이터에 액세스하기를 원할 수 있습니다.

이러한 사용 사례는 파티셔닝에 적합합니다. 쿼리 패턴에 맞게 데이터를 분할하면 특히 Z 순서 지정과 같은 다른 성능 최적화와 결합할 때 쿼리 성능 속도가 크게 향상될 수 있습니다.⁷ 델타 테이블 파티션은 동일한 데이터 행을 공유하는 데이터 행의 하위 집합이 있는 폴더로 구성됩니다. 하나 이상의 열에 대한 값입니다.



이러한 유형의 온디스크 파티셔닝을 데이터 프레임을 처리할 때 Spark가 적용하는 파티셔닝과 혼동해서는 안 됩니다. Spark는 Spark 클러스터의 많은 노드에서 작업을 병렬 및 독립적으로 실행할 수 있도록 메모리 내 파티셔닝을 적용합니다.

예를 들어 노란색 택시 데이터의 경우 분할 열은 VendorId일 수 있습니다.

테이블을 분할한 후에는 각 VendorId에 대해 개별 폴더가 생성됩니다.

폴더 이름의 마지막 부분은 VendorId=XX입니다.

```
drwxrwxrwx 2 루트 루트 4096 12월 13일 15:16 VendorId=1 drwxrwxrwx 2
루트 루트 4096 12월 13일 15:16 VendorId=2 drwxrwxrwx 2 루트 루트 4096
12월 13일 15:16 VendorId=4
```

테이블이 분할되면 Spark가 올바른 파티션이 있는 폴더를 즉시 선택할 수 있으므로 파티션 열을 포함하는 조건자가 포함된 모든 쿼리가 훨씬 빠르게 실행됩니다. PARTITIONED BY 절을 지정하여 델타 테이블을 생성할 때 데이터를 분할할 수 있습니다.

⁷ Z 순서 지정은 5 장에서 다릅니다.



이 글을 쓰는 시점에서는 파티션이 권장되는 접근 방식입니다.
쿼리 성능을 높이기 위해 쿼리 패턴에 데이터를 정렬합니다.
액체 클러스터링이라는 Delta Lake의 새로운 기능이 현재
5 장에서 배우게 될 미리보기입니다.
독자가 파티션의 작동 방식과 방식을 이해하는 것이 중요합니다.
기능을 배우기 전에 수동으로 적용할 수 있습니다.
이러한 명령을 자동화하고 대체합니다. 새로운 기능, 액체
클러스터링은 가까운 시일 내에 일반적으로 제공될 예정입니다. 당신은 할 수 있습니다.
액체 클러스터링 상태에 대해 자세히 알아보고 최신 상태를 유지하세요.
Delta Lake 설명서 웹 사이트 및 이 [기능 요청](#).

단일 열로 파티션 나누기

YellowTaxis 테이블을 사용하여 다음과 같이 분할된 새 버전을 생성해 보겠습니다.

공급업체 ID 먼저 분할된 테이블을 만듭니다.8

```
%sql
CREATE TABLE Taxdb.YellowTaxisPartitioned
(
    탑승 ID          정수,
    공급업체 ID      정수,
    데리러 갈 시간   타임스탬프,
    드롭티임         타임스탬프,
    픽업 위치 ID      정수,
    삭제 위치 ID      정수,
    택시 번호         끈,
    운전면허번호     끈,
    승객수           정수,
    여행거리         더블,
    요금 코드 ID      정수,
    결제 유형         정수,
    총금액           더블,
    운임금액         더블,
    추가의           더블,
    MtaTax            더블,
    팁금액           더블,
    통행료금액       더블,
    개선할증료       더블
)
```

) 델타 사용
파티셔닝 대상(VendorId)
위치 "/mnt/datalake/book/chapter03/YellowTaxisDeltaPartitioned"

PARTITIONED BY(VendorId) 절을 확인하세요. 이제 테이블이 생겼으니,
이전 YellowTaxis 테이블에서 데이터를 로드하고 해당 데이터를 새 테이블에 씁니다.
먼저 DataFrameReader를 사용하여 데이터를 읽습니다.

8 GitHub 저장소 위치: /chapter03/11 - 파티션

```
input_df = Spark.read.format("delta").table("taxidb.YellowTaxis")
```

다음으로 DataFrameWriter를 사용하여 분할된 델타 테이블에 데이터를 씁니다.

```
\\
\\
\\
\\
input_df.write.format("델
타").mode("덮어쓰기").save("/
mnt/datalake/book/chapter03/YellowTaxisDeltaPartitioned")
```

이제 테이블의 딕렉터리를 보면 모든 VendorID에 대한 하위 딕렉터리가 표시됩니다.

```
%sh
ls -al /dbfs/mnt/datalake/book/chapter03/YellowTaxisDeltaPartitioned drwxrwxrwx 2 루트 루트 4096 12월 5일 17:39 .
drwxrwxrwx 2 루트 루트 4096 12월 2일 19:02 .. drwxrwxrwx 2 루트 루트
4096 12월 5일 16:44 VendorId=1
drwxrwxrwx 2 루트 루트 4096 12월 5일 16:44 VendorId=2 drwxrwxrwx 2 루트 루트 4096 12
월 5일 16:44 VendorId=4 drwxrwxrwx 2 루트 루트 4096 12월 5일 16:44 _delta_log
```

고유한 VendorId를 살펴보면 실제로 해당 세 가지만 있음을 알 수 있습니다.

ID:

```
%sql
선택하다
DISTINCT(공급업체 ID)
```

Taxdb.YellowTaxisPartitioned에서;

동일한 ID가 표시됩니다.

```
+-----+
| 공급업체 ID |
+-----+
| 2 |
| 4 |
|
+-----+
```

VendorId 하위 딕렉터리에는 여기에 표시된 대로 개별 Parquet 파일이 포함되어 있습니다.

공급업체 ID=4:

```
%sh
ls -al /dbfs/mnt/datalake/book/chapter03/YellowTaxisDeltaPartitioned/VendorId=4 총 3378 drwxrwxrwx 2 루트 루트 4096 12월 5일 17:41 .
drwxrwxrwx 2 루
트 루트 4096 12월 5일 17:39 ..
-rwxrwxrwx 1 루트 루트 627551 12월 5일 17:41 부분-00000...parquet -rwxrwxrwx 1 루트 루트 618844 12월 5일 17:41
부분-00001...parquet -rwxrwxrwx 1 루트 루트 616377 12월 5일 17:41 부분-00002...parquet -rwxrwxrwx 1 루트 루
트 614035 12월 5일 17:41 part-00003...parquet -rwxrwxrwx 1 루트 루트 612410 12월 5일 17:41 part-00004...parquet
-rwxrwxrwx 1 루트 루트 360432 12월 5일 17:41 part-00005..마루
```

여러 열로 분할하기 단 하나의 열로 분

할 필요는 없습니다. 여러 계층 열을 분할 열로 사용할 수 있습니다. 예를 들어 IoT 데이터의 경우 가장 일반적으로 사용되는 쿼리 패턴이기 때문에 일, 시간, 분별로 분할할 수 있습니다.

예를 들어, YellowTaxis 테이블을 VendorId 뿐만 아니라 RateCodeId로 분할 하려고 한다고 가정해 보겠습니다. 먼저 기존 YellowTaxisPartitioned 테이블과 해당 기본 파일을 삭제해야 합니다. 다음으로 테이블을 다시 만들 수 있습니다.

```
%sql
-- CREATE TABLE
Taxdb.YellowTaxisPartitioned 테이블 생성(
    탑승 ID          정수,
    ...
) 델타 사용
PARTITIONED BY(VendorId, RatecodeId) -- VendorId 및 rateCodeId를 기준으로 한 파티션
위치 "/mnt/datalake/book/chapter03/YellowTaxisDeltaPartitioned"
```

업데이트된 파티션 절인 PARTITIONED BY(VendorId, RatecodeId)를 확인하세요.

그런 다음 이전과 동일한 방식으로 테이블을 다시 로드할 수 있습니다. 테이블이 로드되면 디렉토리 구조를 다시 살펴볼 수 있습니다. 첫 번째 수준은 여전히 동일해 보입니다.

```
%sh
ls -al /dbfs/mnt/datalake/book/chapter03/YellowTaxisDeltaPartitioned drwxrwxrwx 2 루트 루트 4096 12월 13일
15:33 . drwxrwxrwx 2 루트 루트 4096 12월 2일 19:02 .. drwxrwxrwx 2 루트 루트 4096
트 루트 4096 12월 13일 15:16 VendorId=1 drwxrwxrwx 2 루트 루트 4096
12월 13일 15:16 VendorId=2 drwxrwxrwx 2 루트 루트 4096 12월 13일 :16 공급업체 ID= 4
```

```
drwxrwxrwx 2 루트 루트 4096 12월 13일 15:16 _delta_log
```

VendorId=1 디렉터리를 살펴보면 다음과 같은 파티셔닝을 볼 수 있습니다.

요금 코드 ID:

```
%sh
ls -al /dbfs/mnt/datalake/book/chapter03/YellowTaxisDeltaPartitioned/VendorId=1
drwxrwxrwx 2 루트 루트 4096 12월 13일 15:35 .
drwxrwxrwx 2 루트 루트 4096 12월 13일 15:33 .. drwxrwxrwx 2 루트 루트
4096 12월 13일 15:16 RatecodeId=1 drwxrwxrwx 2 루트 루트 4096 12월 13일 15:16
RatecodeId=2 drwxrwxrwx 2 루트 루트 4096 12월 13일 1 5:16 요금코드Id= 삼

drwxrwxrwx 2 루트 루트 4096 12월 13일 15:16 RatecodeId=
drwxrwxrwx 2 루트 루트 4096 12월 13일 15:16 RatecodeId=5 drwxrwxrwx 2 루트 루트 4096
12월 13일 15:16 RatecodeId=6 drwxrwxrwx 2 루트 루트 4096 12월 13일 15:16 RatecodeId=99
```

마지막으로 RatecodeId 수준에서 쿼리하는 경우:

```
%sh
ls -al /dbfs/.../chapter03/YellowTaxisDeltaPartitioned/VendorId=1/RatecodeId=1

해당 파티션에 대한 Parquet 파일을 볼 수 있습니다.

drwxrwxrwx 2 루트 루트 drwxrwxrwx 2          4096 12월 13일 15:35 .
루트 루트           4096 12월 13일 15:35 ..

-rwxrwxrwx 1 루트 루트 10621353 12월 13일 15:35 part-00000...parquet -rwxrwxrwx 1 루트 루트 10547673 12월 13일
15:35 part-00001...parquet -rwxrwxrwx 1 루트 루트 10566377 12월 13일 15:35 부분 -00002...parquet -rwxrwxrwx 1 루
트 루트 10597523 12월 13일 15:35 part-00003...parquet -rwxrwxrwx 1 루트 루트 10570937 12월 13일 15:35
part-00004...parquet -rwxrwxrwx 1 루트 루트 6119491 12월 13일 15:35 part-00005...parquet -rwxrwxrwx 1 루트 루트
13820133 12월 13일 15:35 part-00007...parquet -rwxrwxrwx 1 루트 루트 24076060 12월 13일 15:35 part-00008... .parquet
-rwxrwxrwx 1 루트 루트 6772609 12월 13일 15:35 part-00009...parquet
```



여러 열로 분할하는 이러한 유형의 분할이 지원되지만 몇 가지 함정을 지적하고 싶습니다. 생성된 파일 수는 두 열의 카디널리티를 곱한 값이 됩니다. 따라서 이 경우 공급업체 수와 요율표 수를 곱한 값입니다. 이로 인해 작은 Parquet 부품 파일이 많이 생성되는 "작은 파일 문제"가 발생할 수 있습니다.

때로는 Z 순서 지정과 같은 다른 솔루션이 분할보다 더 효과적일 수 있습니다. [5장](#) [에서는](#) 성능 조정과 이 주제를 더 자세히 다룹니다.

파티션이 있는지 확인하기

테이블에 특정 파티션이 포함되어 있는지 확인하려면 다음 문을 사용할 수 있습니다.

```
SELECT COUNT(*) > 0 FROM <테이블 이름> WHERE <파티션 열> = <값>
```

파티션이 존재하면 true 가 반환됩니다. 다음 SQL 문은 VendorId = 1 및 RatecodeId = 99 에 대한 파티션이 존재하는지 확인합니다.

```
%sql
선택하다

COUNT(*) > 0 AS '파티션이 존재합니다' FROM

Taxdb.YellowTaxisPartitioned
어디
VendorId = 2 AND RateCodeId = 99
```

이 파티션은 이전에 표시된 대로 존재하므로 true를 반환합니다.

교체 위치를 사용하여 델타 파티션을 선택적으로 업데이트

이전 섹션에서는 쿼리 작업 속도를 크게 높일 수 있는 방법을 살펴보았습니다.
 데이터를 분할하여. 또한 다음을 사용하여 하나 이상의 파티션을 선택적으로 업데이트할 수도 있습니다.
 대체 위치 옵션. 특정 파티션에 업데이트를 선택적으로 적용하는 것이 항상 가능한 것은 아닙니다.
 가능한; 일부 업데이트는 전체 데이터 레이크에 적용되어야 합니다. 그러나 해당되는 경우,
 이러한 선택적 업데이트로 인해 속도가 크게 향상될 수 있습니다. Delta Lake는 업데이트할 수 있습니다.
 뛰어난 성능의 파티션과 동시에 데이터를 보장합니다.

진실성.

교체 위치가 실제로 작동하는 모습을 보려면 특정 파티션을 살펴보겠습니다.

```
%sql
선택하다
  Rideld, Vendorid, 결제 유형
에서
  Taxdb.yellowtaxis파티션됨
여기서
  VendorID = 1 AND RatecodeId = 99 제한 5
```

결과에는 다양한 결제 유형이 표시됩니다.

탑승 ID	공급업체 ID	결제 유형
1137733 1	1	
1144423 1	1214030	2
1	1223028	1
1300054 1	2	

모든 PaymentType이 다음과 같은 비즈니스 이유가 있다고 가정해 보겠습니다.

VendorId = 1이고 RatecodeId = 9는 3이어야 합니다. 다음 PySpark를 사용할 수 있습니다.

해당 결과를 얻으려면 교체 위치를 사용하는 표현식을 사용하세요.

```
pyspark.sql.functions에서 가져오기 *
스파크.읽기 \
  .format("델타") \
  .load("/mnt/datalake/book/chapter03/YellowTaxisDeltaPartitioned") \
  .where((col("VendorId") == 1) & (col("RatecodeId") == 99)) \
  .withColumn("결제 유형", lit(3)) \
  .쓰다 \
  \
  .format("delta") .option("replaceWhere", "VendorId = 1 AND RateCodeId = \
99") .mode("overwrite") .save(" \
  \
  mnt/datalake/book/chapter03/YellowTaxisDeltaPartitioned")
```

이제 이 파티션에 대한 고유한 PaymentType을 찾으면 다음과 같습니다.

```
%sql
선택하다
    DISTINCT(지불 유형)
FROM
    Taxdb.yellowtaxispartitioned
어디
    VendorID = 1 AND RatecodeId = 99
```

PaymentType = 3 만 있음을 알 수 있습니다.

```
+-----+
| 결제 유형 |
+-----+
| 삼          |
+-----+
```

다른 파티션은 영향을 받지 않는지 확인할 수 있습니다.

```
%sql
선택하다
    DISTINCT(지불 유형)
에서
    Taxdb.yellowtaxis파티션됨
주문
    결제 유형
```

모든 PaymentType이 표시됩니다.

```
+-----+
| 결제 유형 |
+-----+
| 1          |
2 |
3 | 4        |
|
+-----+
```

replacementWhere는 다음과 같은 작업을 실행해야 할 때 특히 유용할 수 있습니다.

계산 비용이 많이 들지만 특정 파티션에서만 실행하면 됩니다.

노란색 택시 시나리오에서 데이터 사이언스 팀이 다음과 같이 요청했다고 가정해 보겠습니다.

YellowTaxis 테이블에서 알고리즘 중 하나를 실행합니다. 처음에는 실행할 수 있습니다.

가장 작은 파티션을 선택하고 신속하게 결과를 검색하고 승인되면

밤새 나머지 모든 파티션에 알고리즘을 적용합니다.

사용자 정의 메타데이터

감사 또는 규제 목적으로 특정 SQL에 태그를 추가할 수 있습니다.

운영. 예를 들어 우리 프로젝트에서는 INSERT를 특정 태그에 태그해야 할 수도 있습니다.

일반 데이터 보호 규정(GDPR) 태그가 있는 테이블. 일단 태그를 지정하면

이 태그를 INSERT 하면 감사 도구가 이 특정 태그를 포함하는 전체 문의 목록을 생성 할 수 있습니다.

SQL 작업으로 생성된 메타데이터 커밋에서 이러한 태그를 사용자 정의 문자열로 지정할 수 있습니다.

DataFrameWriter의 옵션 userMetadata를 사용하거나 SparkSession 구성

Spark.databricks.delta.commitInfo.userMetadata를 사용하여 이 작업을 수행할 수 있습니다 . 두 옵션을 모두 지정하면 DataFrameWriter의 옵션이 우선 적용됩니다.

SparkSession을 사용하여 사용자 정의 메타데이터 설정 먼저

SparkSession 구성은 살펴보겠습니다 . 감사 목적으로 GDPR 태그를 할당하려는 INSERT 작업이 있다고 가정합니다 . 다음은 SQL 예입니다.

```
%sql
SET Spark.databricks.delta.commitInfo.userMetadata=my-custom-metadata=
{ "GDPR": "요청 1x965383 삽입" };
```

이 태그는 표준 INSERT인 다음 작업에 적용됩니다 .

```
INSERT INTO Taxdb.yellowtaxisPartitioned(RideId, VendorId,
PickupTime, DropTime, PickupLocationId, DropLocationId,
CabNumber, DriverLicenseNumber, PassengerCount, TripDistance,
RatecodeId, PaymentType, TotalAmount, FareAmount, Extra, MtaTax, TipAmount,
TollsAmount, ImprovementSurcharge)

VALUES(10000000, 3, '2019-11-01T00:00:00.000Z',
'2019-11-01T00:02:23.573Z', 65, 71, 'TAC304', '453987', 2, 4.5, 1, 1, 20.34,
15.0, 0.5, 0.4, 2.0, 2.0, 1.1)
```

INSERT에는 특별한 것이 없습니다 . 그것은 표준 작업입니다. GDPR 태그는 트랜잭션 로그의 커밋 정보에 자동으로 적용됩니다. 최신 .json 파일에 대한 트랜잭션 로그를 검색하면 00004.json이 마지막 로그 항목임을 알 수 있습니다.

```
%sh
ls -al /dbfs/.../YellowTaxisDeltaPartitioned/_delta_log/*.json
```

산출:

```
-rwxrwxrwx 1 .../_delta_log/00000000000000000000.json -rwxrwxrwx 1 .../_delta_log/
00000000000000000001.json -rwxrwxrwx 1 .../_delta_log/00000000000000000002.json
-rwxrwxrwx 1 .../_delta_log/00000000000000000003.json -rwxrwxrwx 1 .../_delta_log/
00000000000000000004.json
```

00004.json 커밋 파일을 보면 GDPR 항목을 볼 수 있습니다.

```
%sh
grep 커밋 /dbfs/.../YellowTaxisDeltaPartitioned/_delta_log/...00004.json >
/tmp/commit.json python
-m json.tool /tmp/commit.json
```

GDPR 항목은 다음과 같습니다.

```
{
  "커밋정보": {
    ...
    "노트북": { "노트북 ID": "1106853862465676" },
    "clusterId": "0605-014705-r8puunyx", "readVersion": 3,
    "isolationLevel": "WriteSerialized", "isBlindAppend": true, "erationMetrics": {

      ...
    },
    "userMetadata": "my-custom-metadata= {\\"GDPR\\": \"INSERT 요청 1x965383\\\" }", "engineInfo": "Databricks-Runtime/10.4.x-scala2.12", "txnid": "99f2f31c-8c01-4ea0-9e23-c0cbae9eb82a"
  }
}
```



SET 문은 현재 Spark 세션 내의 후속 작업에 대해 계속 유효하므로 GDPR 메타데이터를 추가하지 않고 데이터를 계속 삽입하려면 SET를 빈 문자열로 업데이트하거나 RESET 작업을 사용해야 합니다.

RESET은 메타데이터 속성뿐만 아니라 모든 Spark 속성을 재설정한다는 점에 유의하세요 !

DataFrameWriter를 사용하여 사용자 정의 메타데이터 설정

다음과 같이 userMetadata 옵션과 함께 DataFrameWriter를 사용하여 사용자 정의 태그를 삽입할 수도 있습니다.

```
df_for_append.write \ .mode("append") \ .format("delta") \ .option("userMetadata", '{"PII": "Confidential XYZ"}')
\ .save("/mnt/datalake/책/chapter03/YellowTaxisDeltaPartitioned")
```

해당 JSON 항목을 살펴보면 commitInfo에 태그가 표시됩니다.

```
%sh
grep 커밋 /dbfs/.../YellowTaxisDeltaPartitioned/_delta_log/...00005.json >
/tmp/commit.json python
-m json.tool /tmp/commit.json

{
    "커밋정보": {
        ...
        ...
        "userMetadata": "{\"PII\": \"기밀 XYZ\"}",
        ...
    }
}
```

결론

이 장에서는 Delta 테이블의 기본 작업을 논의하여 Delta Lake 사용에 대한 기본 사항을 검토했습니다. Delta Lake는 다양한 유형의 API를 사용하여 다양한 유형의 작업을 수행하는 다양한 방법을 제공합니다. 예를 들어 SQL DDL, DataFrameWriter API 또는 DeltaTable Builder API를 사용하여 델타 테이블을 생성할 수 있습니다. 각 테이블에는 고유한 기능과 구문이 있습니다. 그리고 테이블을 생성할 때 기본 데이터를 쓰기 위한 특정 위치를 지정하여 비관리형 테이블을 생성하거나, Spark가 관리형 테이블을 생성하여 메타데이터와 기본 데이터를 모두 관리하도록 할 수 있습니다.

테이블이 생성되면 여기에 언급된 다양한 API를 사용하여 테이블을 읽고 쓸 수 있습니다. 이 장에서는 주로 SQL 또는 DataFrame API를 사용하여 데이터를 삽입, 추가 또는 덮어쓰는 다양한 방법을 다루었습니다. 더 정교한 쓰기 작업(예: MERGE)은 후속 장에서 다루게 됩니다.

또한 디스크에서 Delta Lake 파티셔닝 기능을 살펴보았습니다. 단일 열로 분할하든 여러 열로 분할하든 상관없이 Delta 테이블은 데이터 처리 개선과 효율성을 크게 향상시킬 수 있는 테이블 분할을 위한 간단한 방법을 제공합니다. 테이블을 분할할 수 있을 뿐만 아니라, 교체 위치 와 같은 강력한 기본 제공 Delta Lake 기능을 사용하면 업데이트를 더 빠르고 효율적으로 적용하기 위해 특정 파티션에 선택적으로 업데이트를 적용할 수 있습니다.

마지막으로 검색 및 발견을 돋기 위해 사용자 정의 메타데이터를 델타 테이블에 추가할 수 있다는 점을 배웠습니다. 이는 특히 감사 또는 규제 목적에 유용할 수 있습니다. 사용자 정의 메타데이터를 사용하면 특정 태그가 포함된 Delta 테이블에 대한 명령문 또는 작업 목록을 컴파일할 수 있습니다.

Delta Lake 및 Delta 테이블 기본 작업에 발을 담근 후 다음 장에서는 보다 정교한 Delta 테이블 DML 작업에 대해 자세히 알아봅니다.

제4장

테이블 삭제, 업데이트 및 병합

Delta Lake는 클래식 데이터 레이크에 트랜잭션 계층을 추가하므로 업데이트, 삭제, 병합과 같은 클래식 DML 작업을 수행할 수 있습니다. Delta 테이블에서 DELETE 작업을 수행하면 작업이 데이터 파일 수준에서 수행되어 필요에 따라 데이터 파일을 제거하고 추가합니다. 제거된 데이터 파일은 더 이상 현재 버전의 델타 테이블의 일부가 아니지만 시간 여행을 통해 이전 버전의 테이블로 되돌리고 싶을 수 있으므로 물리적으로 즉시 삭제해서는 안 됩니다(시간 여행은 6 장에서 다룹니다). UPDATE 작업을 실행할 때도 마찬가지입니다. 필요에 따라 데이터 파일이 델타 테이블에 추가되고 제거됩니다.

가장 강력한 Delta Lake DML 작업은 MERGE 작업으로, 이를 통해 델타 테이블에서 UPDATE, DELETE 및 INSERT 작업이 혼합된 "upsert" 작업을 수행할 수 있습니다. 소스와 대상 테이블을 조인하고 일치 조건을 작성한 다음 일치하거나 일치하지 않는 레코드에 어떤 일이 발생하는지 지정합니다.

델타 테이블에서 데이터 삭제

깨끗한 Taxdb.YellowTaxis 테이블 부터 시작하겠습니다. 이 테이블은 4 장의 "Chapter 초기화" 스크립트에 의해 생성됩니다.¹ 9,999,995백만 개의 행이 있습니다.

```
%sql
선택하다
개수(ID)
```

Taxdb.YellowTaxis에서

산출:

¹ GitHub 저장소 위치: /chapter04/00 - 장 초기화

```
+-----+
| 개수(1) |
+-----+
| 9999995 |
+-----+
```

테이블 생성 및 기록 설명

Taxdb.YellowTaxis Delta 테이블은 "Chapter 초기화" 스크립트에서 생성되었으며 /chapter04 폴더에 복사되었습니다. 테이블에 대한 DESCRIBE HISTORY를 살펴보겠습니다 .2

```
%sql
내역 설명 Taxdb.YellowTaxis
```

출력(관련 부분만 표시됨):

```
+-----+-----+-----+
| 운영 | 작업매개변수           | 작업메트릭스          |
+-----+-----+-----+
| 쓰기 || [(['모드', '덮어쓰기'], (...)]) | [('numFiles', '2'), ||| ('numOutputRows', ||| '9999995'), ...] |
+-----+-----+-----+
```

총 9,999,995개 행에 대해 두 개의 데이터 파일을 쓰는 WRITE 작업이 포함된 하나의 트랜잭션이 있음을 알 수 있습니다 . 두 파일 모두에 대한 세부 정보를 알아 보겠습니다.

2장 에서는 트랜잭션 로그를 사용하여 파일 추가 및 제거 작업을 확인하는 방법을 배웠습니다. _delta_log 디렉터리를 살펴보겠습니다.

```
%sh
ls /dbfs/mnt/datalake/book/chapter04/YellowTaxisDelta/_delta_log/*.json
```

예상한 대로 트랜잭션 로그 항목이 하나만 표시됩니다.

```
/dbfs/mnt/datalake/book/chapter04/YellowTaxisDelta/_delta_log/…0000.json
```

DESCRIBE HISTORY의 numfiles 항목이 2개이므로 이 로그 항목에는 2개의 파일 추가 작업이 있어야 합니다 . 이번에도 grep 명령을 사용하여 해당 섹션을 찾아보겠습니다.

```
%sh
grep '\"add\"' /dbfs/…/chapter04/YellowTaxisDelta/_delta_log/..0000.json |
  sed -n 1p > /tmp/commit.json python
-m json.tool < /tmp/commit.json
```

이전 명령의 한 변형은 이제 두 개의 항목이 있으므로 sed 명령을 사용하여 올바른 add 항목을 추출해야 한다는 것입니다.

2 GitHub 저장소 위치: /chapter04/01 - 작업 삭제



grep 명령 출력을 sed3 명령으로 파이프할 수 있습니다. sed는 입력 스트림에 서 기본 텍스트 변환을 수행하고 결과를 출력 스트림에 쓰는 스트림 편집기입니다. -n 옵션은 일반 출력을 억제하고 1p 명령은 입력의 첫 번째 줄만 인쇄합니다. 다음 추가 항목을 찾으려면 두 번째 줄을 출력하는 sed -n 2p를 사용하면 됩니다.

생성된 출력(관련 부분만 표시됨):

```
{
    "추가": { "경
        로": "part-00000....c000.snappy.parquet",
        ...
        "통계": "{\"numRecords\":5530100,...}",
        "태그": {
            ...
        }
}
```

여기서는 테이블에 대해 생성된 첫 번째 데이터 파일의 이름과 해당 파일의 레코드 수를 볼 수 있습니다. sed -n 2p 와 동일한 명령을 사용하여 두 번째 추가 작업을 수행하여 두 번째 데이터 파일을 가져올 수 있습니다.

```
{
    "추가": { "경
        로": "part-00001....c000.snappy.parquet", "...: 통계":
        "{\"numRecords\":4469895,...}",
        "태그": {
            ...
        }
}
```

이제 우리는 테이블에 다음과 같은 데이터 파일이 있다는 것을 알고 있습니다.

표 4-1. 생성된 쪽모이 세공 파일

쪽모이 세공 파일 이	레코드 수
part-00000-d39cbaa1-ea7a-4913-a416-e229aa1d5616-c000.snappy.parquet	5,530,100
part-00001-947cccf8-41ae-4212-a474-fedaa0f6623d-c000.snappy.parquet	4,469,895

이러한 파일은 디렉터리 목록과 일치하므로 트랜잭션 로그와 디렉터리 목록 보고서는 일관됩니다.

```
%sh
ls -al /dbfs/mnt/datalake/book/chapter04/YellowTaxisDelta
```

```
drwxrwxrwx 2 _delta_log -rwxrwxrwx
1 부분-00000-d39cbaa1-ea7a-4913-a416-e229aa1d5616-c000.snappy.parquet -rwxrwxrwx 1 부분-00001-947cccf8-41ae-4212-a4
74-fedaa0f6623d-c000.snappy.parquet
```

DELETE 작업 수행 단일 레코드(이 경우 Rideld =

100000) 를 삭제한다고 가정해 보겠습니다. 먼저 SQL SELECT:4를 사용하여 해당 레코드가 실제로 테이블에 여전히 있는지 확인해야 합니다.

```
%sql
-- 먼저 Rideld = 10000에 대한 데이터가 있음을 보여줍니다.
선택하다
    탑승 ID,
    공급업체 ID,
    택시번호,
    총금액
에서
    Taxdb.YellowTaxis
어디
    탑승 ID = 100000
```

산출:

탑승 ID 공급업체 ID 택시번호 총금액
100000 2 T478827C 7.56

이 행을 삭제하려면 간단한 SQL DELETE를 사용하면 됩니다. DELETE 명령을 사용하여 조건자 또는 필터링 조건을 기반으로 행을 선택적으로 삭제할 수 있습니다.

```
%sql
다음에서 삭제
    Taxdb.YellowTaxis
Rideld = 100000인 곳
```

산출:

num_affected_rows
1

실제로 하나의 행을 삭제했음을 확인할 수 있습니다. DESCRIBE를 사용할 때

테이블의 다양한 작업을 살펴보는 HISTORY 명령을 사용하면 다음과 같은 결과를 얻을 수 있습니다.

버전 1의 경우(가독성을 위해 행 출력이 피벗됨):

버전: 타임스탬프	1
프: 작업:	2022-12-14T17:50:23.000+0000
	삭제
OperationParameters: [('predicate',	
'["(spark_catalog.taxidb.YellowTaxis.Rideld = 100000)"]')]])	
작업 지표:	[('제거된 파일 수', '1'), ('numCopiedRows', '5530099'), ('numAddedChangeFiles', '0'), ('executionTimeMs', '32534'), ('numDeletedRows', '1'), ('scanTimeMs', '1524'), ('numAddedFiles', '1'), ('rewriteTimeMs', '31009')]]

작업이 DELETE이고 삭제에 사용한 조건자가 다음과 같은 것을 볼 수 있습니다.

WHERE Rideld = 100000. Delta Lake는 파일 1개 (numRemovedFiles = 1)를 제거했으며

새 파일 하나를 추가했습니다 (numAddedFiles = 1). 신뢰할 수 있는 grep 명령을 사용하여 다음을 찾으면 세부 사항을 살펴보면 다음과 같습니다.

표 4-2. DELETE 작업 결과

작업 파일 이름	레코드 수
주가해당 부품-00000-96c2f047-99cc-4a43-b2ea-0d3e0e77c4c1-c000.snappy.parquet	5,530,099
부품-00000-d39cbaa1-ea7a-4913-a416-e229aa1d5616-c000.snappy.parquet	4,469,895 제거

그림 4-1은 레코드를 삭제할 때 Delta Lake의 작업을 보여줍니다.

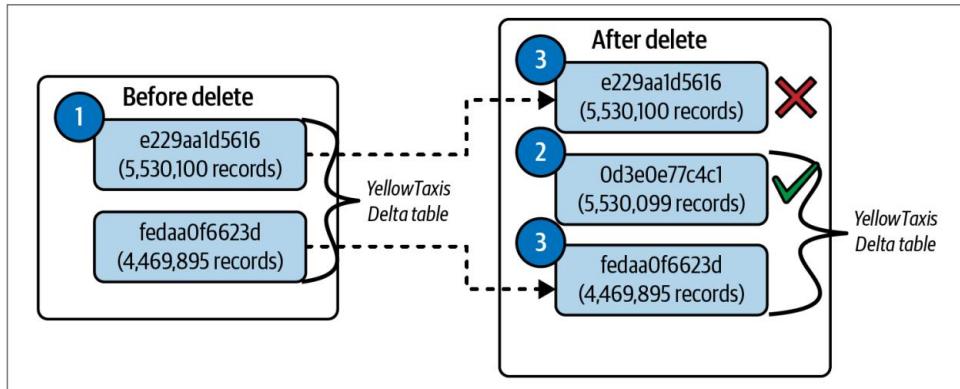


그림 4-1. DELETE 작업 전후의 YellowTaxis Delta 테이블

Delta Lake는 DELETE 작업의 일부로 다음 작업을 수행합니다.

1. Delta Lake는 조건자 조건과 일치하는 행이 포함된 파일을 식별하기 위해 데이터의 첫 번째 스캔을 수행했습니다. 이 경우 파일은 e229aa1d5616 데이터 파일입니다. 여기에는 RidId = 100000 인 레코드가 포함되어 있습니다.
2. 두 번째 검색에서는 Delta Lake가 일치하는 데이터 파일을 메모리로 읽어옵니다. 이 시점에서 Delta Lake는 새로운 정리 데이터 파일을 스토리지에 쓰기 전에 문제의 행을 삭제합니다. 이 새 데이터 파일은 0d3e0e77c4c1 데이터 파일입니다. Delta Lake가 하나의 레코드를 삭제했으므로 이 새 데이터 파일에는 5,530,099개의 레코드(5,530,100 - 1)가 포함됩니다.
3. Delta Lake가 DELETE 작업을 완료하면 데이터 파일 e229aa1d5616 이 더 이상 Delta 테이블의 일부가 아니므로 이제 Delta 트랜잭션 로그에서 제거됩니다. 이 프로세스를 "삭제"라고 합니다. 그러나 테이블의 이전 버전으로 시간을 이동하려면 여전히 필요할 수 있으므로 이 이전 데이터 파일은 삭제되지 않는다는 점에 유의하는 것이 중요합니다. VACUUM 명령을 사용하여 특정 기간보다 오래된 파일을 삭제할 수 있습니다. 시간 여행과 VACUUM 명령은 6 장에서 자세히 다룹니다.
4. 데이터 파일 fedaa0f6623d 는 변경 사항이 없으므로 델타 테이블의 일부로 유지됩니다.
그것에 적용됩니다.

디렉터리 목록에서 디렉터리에 추가된 하나의 데이터 파일 (0d3e0e77c4c1) 을 볼 수 있습니다.

```
%sh
ls -al /dbfs/mnt/datalake/book/chapter04/YellowTaxisDelta/
drwxrwxrwx _delta_log
-rwxrwxrwx 부분-00000-96c2f047-99cc-4a43-b2ea-0d3e0e77c4c1-c000.snappy.parquet -rwxrwxrwx 부분-00000-d39cbaa1-ea7a-4913-a416-e2 29aa1d5616-c000.snappy.parquet -rwxrwxrwx 부분-00001-947cccf8-41ae-4212-a474-fedaa0f6623d-c000.snappy.parquet
```

데이터 파일 e229aa1d5616 은 물리적으로 삭제되지 않았습니다.

여기서 알아야 할 가장 중요한 메시지는 삭제 트랜잭션이 데이터 파일 수준에서 발생한다는 것입니다. Delta Lake는 필요에 따라 새 파티션을 만들고 새 추가 파일을 삽입하고 트랜잭션 로그에 파일 작업을 제거합니다. 성능 조정에 관한 6장 에서는 VACUUM 명령과 더 이상 필요하지 않은 삭제 표시가 있는 데이터 파일을 정리하는 다른 방법을 다룹니다.

DELETE 성능 조정 팁

Delta Lake에서 DELETE 작업의 성능을 향상시키는 주요 방법은 검색 범위를 좁히기 위해 더 많은 조건자를 추가하는 것입니다. 예를 들어, 분할된 데이터가 있고 삭제할 레코드가 속한 파티션을 알고 있는 경우 해당 파티션 절을 DELETE 조건자에 추가할 수 있습니다.

Delta Lake는 데이터 건너뛰기, z 순서 최적화 등 다양한 기타 최적화 조건도 제공합니다. Z 순서 지정은 효율성을 극대화하기 위해 유사한 열 값이 전략적으로 서로 가까이 배치되도록 각 데이터 파일의 레이아웃을 재구성합니다. 자세한 내용은 [5장을](#) 참조하세요 .

테이블의 데이터 업데이트

YellowTaxis 테이블에 대한 DELETE 작업의 영향을 살펴보았으므로 이제 UPDATE 작업을 간단히 살펴보겠습니다 . UPDATE 작업을 사용하면 필터링 조건(조건자라고도 함)과 일치하는 행을 선택적으로 업데이트할 수 있습니다.

사용 사례 설명 Rideld = 9999994

인 레코드에 대한 DropLocationId에 오류가 있다고 가정해 보겠습니다. 먼저 다음과 같이 이 레코드가 테이블에 있는지 확인하겠습니다.

선택하다:

```
선택하다
    INPUT_FILE_NAME(),
    택승 ID,
    공급업체 ID,
    삭제 위치 ID
에서
    Taxdb.YellowTaxis
어디
    택승 ID = 9999994
```

Spark SQL INPUT_FILE_NAME() 함수는 레코드가 있는 파일 이름을 제공하는 편리한 함수입니다.

입력_파일_이름()	택승 ID 공급업체 ID 삭제 위치 ID
.../part-00001...마루 9999994 1	243

INPUT_FILE_NAME 함수는 우리 레코드가 fedaa0f6623d 데이터 파일에 있음을 보여줍니다. 이는 마지막 레코드 중 하나이므로 논리적으로 마지막으로 생성된 데이터 파일에 위치하므로 의미가 있습니다. 기존 DropLocationId가 현재 243인 것을 확인할 수 있습니다 . 이 필드를 업데이트하여 값이 250이라고 가정해 보겠습니다. 다음으로 실제 DELETE 작업을 살펴보겠습니다 .

테이블의 데이터 업데이트

이제 다음과 같이 UPDATE SQL 문을 작성할 수 있습니다.

```
%sql
업데이트
Taxdb.YellowTaxis
세트
DropLocationId = 250
어디
탑승 ID = 9999994
```

단일 행을 업데이트한 것을 볼 수 있습니다.

num_affected_rows	
1	

먼저 테이블이 성공적으로 업데이트되었는지 확인해 보겠습니다.

```
%sql
선택하다
Rideld,
DropLocationId FROM
Taxdb.YellowTaxis
어디
탑승 ID = 9999994

+-----+---+
| 탑승 ID | 삭제 위치 ID |
+-----+---+
| 9999994 | 250 |
```

출력에는 레코드가 성공적으로 업데이트되었음을 표시합니다. 테이블에서 DESCRIBE HISTORY 명령을 사용하면 테이블 버전 3에서 UPDATE 작업이 표시됩니다(명확성을 위해 출력이 피벗됨).

```
버전: 타임스탬프
프: 작업: 3 2022-12-23 17:20:45+00:00
업데이트
OperationParameters: [ ('predicate', '(Rideld = 9999994)'), ('numRemovedFiles', '1'),
'4469894'), OperationMetrics: ('numCopiedRows',
('numAddedChangeFiles', '0'), ('executionTimeMs', '25426'), ('scanTimeMs',
'129'), ('numAddedFiles', '1'),
```

```
('numUpdatedRows', '1'),
('rewriteTimeMs', '25293'))]
```

파일 1개가 제거되고 ('numRemovedFiles', '1') 1개가 추가되었습니다 ('numAdded Files', '1'). UPDATE 조건자 [('predicate', '(RidId = 9999994)')] 도 볼 수 있습니다 . 세부 정보를 찾기 위해 grep 명령을 사용하면 상황은 다음과 같습니다.

표 4-3. UPDATE 작업의 결과로 수행되는 작업

작업 파일 이름	레코드 수
추가하다 part-00000-da1ef656-46e-4de5-a189-50807db851f6-c000.snappy.parquet	
4,469,895 제거 part-00001-947cccf8-41ae-4212-a474-fedaa0f6623d-c000.snappy.parquet	4,469,895

그림 4-2는 레코드를 삭제할 때 Delta Lake가 수행한 작업을 보여줍니다.

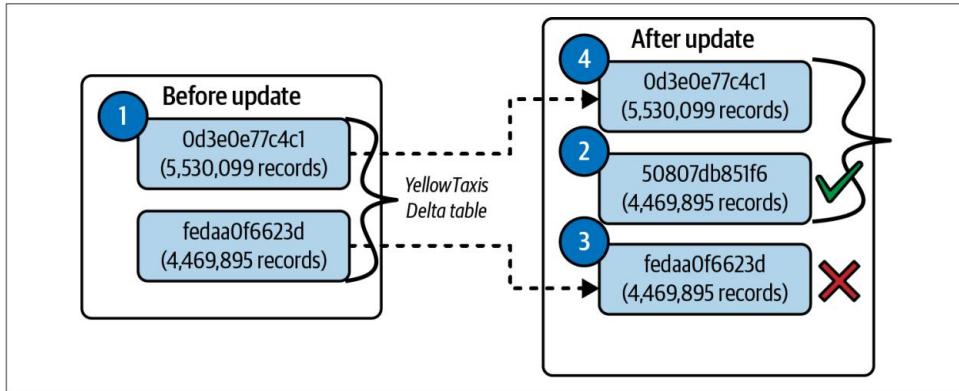


그림 4-2. UPDATE 작업 전과 후

Delta Lake는 두 단계로 테이블에 대한 업데이트를 수행합니다.

1. 조건자와 일치하여 업데이트가 필요한 데이터 파일을 찾아서 선택합니다. Delta Lake는 가능할 때마다 데이터 건너뛰기를 사용하여 이 프로세스의 속도를 높입니다. 이 경우에는 fedaa0f6623d 데이터 파일입니다. INPUT_FILE_NAME() SQL 함수를 사용하여 이를 확인할 수도 있습니다 .
2. 그런 다음 Delta Lake는 일치하는 각 파일을 메모리로 읽고, 관련 행을 업데이트하고, 결과를 새 데이터 파일에 기록합니다. 이 경우 새 데이터 파일은 50807db851f6 파일입니다. 이제 여기에는 fedaa0f6623d 파티션의 모든 레코드가 포함되어 있지만 업데이트가 적용되었습니다. 이 경우 RidId = 9999994에 대한 업데이트입니다. 이 데이터 파일은 50807db851f6입니다. 이 데이터 파일은 계속해서 4,469,895개의 기록을 보유하고 있습니다. Delta Lake가 업데이트를 성공적으로 실행하면 새 데이터 파일에 대한 파일 추가 작업을 추가합니다.

더 이상 필요하지 않으므로 데이터 파일 fedaa0f6623d는 트랜잭션 로그의 커밋 제거 작업을 통해 델타 테이블에서 제거됩니다. 그러나 DELETE 작업과 마찬가지로 시간 이동을 통해 이전 버전의 테이블을 확인하려는 경우를 대비하여 파일이 물리적으로 삭제되지 않습니다.

데이터 파일 0d3e0e77c4c1은 업데이트의 영향을 받지 않았으므로 델타 테이블의 일부로 남아 있으며 계속해서 5,530,099개의 레코드를 보유합니다.

UPDATE 성능 조정 팁 DELETE 작업 과 마찬가지로

Delta Lake에서 UPDATE 명령의 성능을 향상시키는 주요 방법은 더 많은 조건자를 추가하여 검색 범위를 좁히는 것입니다. 검색이 구체적일수록 Delta Lake에서 검색 및/또는 수정해야 하는 파일 수가 줄어듭니다.

이전 섹션에서 언급한 것처럼 Z 순서 지정과 같은 다른 Delta Lake 기능을 사용하여 UPDATE 작업 속도를 더욱 높일 수 있습니다. Delta Lake 최적화에 대한 자세한 내용은 [5장을](#) 참조하세요 .

MERGE 작업을 사용하여 데이터 Upsert

Delta Lake MERGE 명령을 사용하면 데이터에 대한 upsert를 수행할 수 있습니다. upsert는 UPDATE 와 INSERT 명령이 혼합된 것입니다. upsert를 이해하기 위해 기존 테이블(대상 테이블)과 새 레코드와 기존 레코드에 대한 업데이트가 혼합되어 있는 원본 테이블이 있다고 가정해 보겠습니다. upsert가 실제로 작동하는 방식은 다음과 같습니다.

1. 원본 테이블의 레코드가 대상 테이블의 기존 레코드와 일치하면 Delta Lake가 레코드를 업데이트합니다.
2. 일치하는 항목이 없으면 Delta Lake는 새 레코드를 삽입합니다.

사용 사례 설명 YellowTaxis 테이블

이를 예 MERGE 작업을 적용해 보겠습니다 . YellowTaxis 테이블의 개수를 계산해 보겠습니다 .

```
%sql
선택하다
세다(*)
에서
Taxdb.YellowTaxis
```

9,999,994개의 레코드가 있음을 알 수 있습니다.

개수(1)

9999994

이 장의 DELETE 섹션에서 삭제한 Rideld = 100000 인 레코드를 다시 삽입하려고 합니다 . 따라서 소스 데이터에는 Rideld가 다음으로 설정된 레코드 하나가 필요합니다.
100000.

이 예에서는 VendorId 가 잘못 삽입되었기 때문에 Rideld = 999991 로 레코드를 업데이트하려고 하며 이러한 5 개의 레코드에 대해 1 (VendorId = 1) 로 업데이트해야 한다고 가정해 보겠습니다. 마지막으로 레코드 수를 10,000,000개로 늘리고 싶으므로 Rideld 범위 가 999996 에서 10000000인 5개의 레코드가 더 있습니다.

MERGE 데이터세트

책과 함께 제공되는 소스 데이터 파일에는 이러한 기록이 포함된 YellowTaxis - MergeData.csv라는 파일이 있습니다. 스키마를 제공해야 하므로 먼저 기존 테이블에서 스키마를 읽습니다.

```
df = Spark.read.format("delta").table("taxidb.YellowTaxis") yellowTaxiSchema = df.schema
print(yellowTaxiSchema)
```

스키마를 로드한 후에는 병합 데이터 CSV 파일을 로드할 수 있습니다.

```
yellowTaxisMergeDataFrame = 스파크 \.read \.option("header",
    "true") \.schema(yellowTaxiSchema) \.csv("/
mnt/datalake/book/chapter04/
YellowTaxisMergeData.csv") .sort(col("Rideld
"))
```

```
yellowTaxisMergeDataFrame.show()
```

부분 출력은 다음과 같습니다.

탑승 ID	공급업체 ID	데리러 갈 시간
100000	2	9999991 2022-03-01T00:00:00.000+0000
1	9999992	1 2022-04-04T20:54:04.000+0000
9999993	1 9999994	2022-04-04T20:54:04.000+0000
1 9999995	1	2022-04-04T20:54:04.000+0000
9999996	3 9999997	2022-04-04T20:54:04.000+0000
3 9999998	3	2022-04-04T20:54:04.000+0000
9999999	3 10000000	2022-03-01T00:00:00.000+0000
삼		2022-03-01T00:00:00.000+0000
		2022-03-01T00:00:00.000+0000
		2022-03-01T00:00:00.000+0000
		2022-03-01T00:00:00.000+0000

Rideld = 100000, 5개의 레코드 (9999991 ~) 로 기록을 볼 수 있습니다.

9999995), 새 VendorId가 1이고 9999996에서 시작하는 5개의 새 레코드가 있습니다.

우리는 MERGE 문을 SQL로 작성하고 싶기 때문에 DataFrame이 필요합니다.

SQL에서 사용 가능합니다. DataFrame 클래스에는 createOrReplace 라는 편리한 메서드가 있습니다.

TempView 는 정확히 다음을 수행합니다.

```
# DataFrame 위에 임시 뷔를 생성하여 이를 만듭니다.  
# 아래 SQL MERGE 문에 액세스 가능  
yellowTaxisMergeDataFrame.createOrReplaceTempView("YellowTaxiMergeData")
```

이제 SQL에서 뷔 이름을 사용할 수 있습니다.

```
%sql  
선택하다  
*  
에서  
노란색택시병합데이터
```

이는 디스플레이() 메서드에 표시된 것과 정확히 동일한 출력을 보여줍니다.
데이터프레임.

MERGE 문

이제 다음과 같이 MERGE 문을 작성할 수 있습니다 .

```
%sql  
MERGE INTO Taxdb.YellowTaxis AS 대상  
    YellowTaxiMergeData AS 소스 사용  
    ON target.Rideld = 소스.Rideld  
  
-- 기록이 있는 경우 VendorId를 업데이트해야 합니다.  
-- 일치하는 경우  
그 다음에
```

-- 모든 열을 업데이트하려면 -- "SET *"이라고 말하면 됩니다.

```
UPDATE SET target.VendorId = source.VendorId WHEN NOT MATCHED THEN -- 모든
열이 일치하는 경우 "INSERT *"를
수행할
수도 있습니다.
INSERT(RideId, VendorId, PickupTime, DropTime, PickupLocationId,
DropLocationId, CabNumber, DriverLicenseNumber, PassengerCount,
TripDistance, RateCodeId, PaymentType, TotalAmount, FareAmount,
Extra, MtaTax, TipAmount, TollsAmount, ImprovementSurCharge)
VALUES(RideId, VendorId, PickupTime, DropTime, PickupLocationId,
DropLocationId, CabNumber, DriverLicenseNumber, PassengerCount,
TripDistance, RateCodeId, PaymentType, TotalAmount, FareAmount,
Extra, MtaTax, TipAmount, TollsAmount, ImprovementSurCharge)
```

이 진술을 분석해 보겠습니다.

1. YellowTaxis Delta 테이블을 병합할 예정입니다. 테이블에 소스 별칭을 제공한다는 점에 유의하세요.
2. USING 절을 사용하여 소스 데이터셋을 지정합니다. 이 경우에는 YellowTaxiMergeData를 보고 소스 별칭을 지정합니다.
3. 소스 데이터셋과 타겟 데이터셋 간의 조인 조건을 정의합니다. 우리의 경우에는 단순히 VendorId에 참여하기를 원합니다. 분할된 데이터가 있고 파티션을 대상으로 지정하려는 경우 여기에 AND를 사용하여 해당 조건을 추가할 수 있습니다. 설명.
4. 소스와 타겟 간에 RideId가 일치할 때 수행할 작업을 지정합니다.
이 사용 사례에서는 1로 설정된 소스의 VendorId로 소스를 업데이트하려고 합니다. 여기서는 하나의 열만 업데이트하지만 필요한 경우 쉼표로 구분된 열 목록을 제공할 수 있습니다. 모든 열을 업데이트하려면 UPDATE SET *라고 말하면 됩니다.
5. 레코드가 소스에는 있지만 대상에는 없는 경우의 작업을 정의합니다. WHEN NOT MATCHED에는 추가 조건이 없지만 사용 사례에서 요구하는 경우 추가 절을 추가할 수 있습니다. 대부분의 경우 INSERT 문을 작업으로 제공합니다. 소스 및 대상 열 이름이 동일하므로 간단한 INSERT *를 사용할 수도 있습니다.

이 MERGE 문을 실행하면 다음과 같은 출력을 얻습니다.

num_affected_rows	num_updated_rows	num_deleted_rows	num_inserted_rows
11	5	0	6

이 출력은 정확히 예상한 것과 같습니다.

- 5개 행 (VendorIds 9999991 ~ 9999995) 을 업데이트합니다 .
- 6개의 행을 삽입합니다 .
 - Rideld 가 100000 인 행 1개
 - 끝에 있는 5개 행 (9999996 ~ 10000000)

처음 5개 행에서 업데이트를 볼 수 있습니다.

```
%sql
-- Rideld가 9999991에서 9999995 사이인 레코드에 대해 VendorId가 업데이트되었
는지 확인하세요.
```

```
선택하다
RideId,
VendorId
FROM
Taxdb.YellowTaxis
RideId가 9999991에서 9999995 사이인 곳

+-----+-----+
| 탑승 ID | 공급업체 ID |
+-----+-----+
| 9999991 | 1 |
9999992 | 1 |
9999993 | 1 |
9999994 | 1 |
9999995 | 1
+-----+-----+
```

이제 모든 행의 소스 VendorId 는 1입니다.

Rideld = 100000 으로 삽입된 레코드를 볼 수 있습니다 .

```
%sql
--VendorId = 100000인 레코드가 있는지 확인하세요.
선택하다
*
```

```
에서
Taxdb.YellowTaxis Rideld =
100000
인 경우
```

출력(부분 출력 표시):

탑승 ID	공급업체 ID	데리러 갈 시간	드롭타임
100000		2022-03-01 00:00:00+00:00	2022-03-01 00:12:01+00:00

마지막으로 Ridel > 9999995 인 새 행을 볼 수 있습니다 .

```
%sql
선택하다
*
```

Taxdb.YellowTaxis에서

어디

탑승 ID > 9999995

탑승 ID	공급업체 ID	데리러 갈 시간	
9999996 3 9999997		2022-03-01 00:00:00+00:00 2022-03-01	
3 9999998 3		00:00:00+00:00 2022-03-01 00:00:00+00:00	
9999999 3 10000000		2022-03-01 00:00:00+00:00 2022-03-01	
삼		00:00:00+00:00	

그리고 총 1,000만 개의 레코드가 있습니다.

```
%sql
선택하다
COUNT(탑승 ID)
```

Taxdb.YellowTaxis에서

개수(1)
10000000

MERGE를 사용하여 일치하지 않는 행 수정 Delta

Lake MERGE 작업에 추가된 중요한 기능은 최근 릴리스된 WHEN NOT MATCHED BY SOURCE 절입니다. 이 절은 원본 테이블에 해당 레코드가 없는 대상 테이블의 레코드를 UPDATE 또는 DELETE하는 데 사용할 수 있습니다 . 이는 소스 테이블에 더 이

상 존재하지 않는 대상 테이블의 레코드를 삭제하거나, 대상 테이블에 레코드를 계속 유지하면서 데이터가 삭제 또는 비활성으로 간주되어야 함을 나타내는 레코드에 플래그를 지정하는 데 유용한 작업일 수 있습니다(즉, 소프트 삭제).



WHEN NOT MATCHED BY SOURCE 절은 Delta 2.3 이상의 Scala, Python 및 Java Delta Lake API에서 지원됩니다. SQL은 Delta 2.4 이상에서 지원됩니다.

원본 테이블에는 있고 대상 테이블에는 없는 레코드를 삭제하려면(예: 영구 삭제) 다음 코드 예와 같이 WHEN NOT MATCHED BY SOURCE 절을 사용합니다.



WHEN NOT MATCHED BY SOURCE 코드는 데모 목적으로만 사용되며 이전 코드 예 제와 순서대로 실행되지 않습니다. WHEN NOT MATCHED BY SOURCE 코드 예제를 실행하면 이 장의 나머지 코드 출력이 이 장의 예제 및 예상 출력과 일치하지 않는다는 점에 유의하십시오.

```
%sql
MERGE INTO Taxidb.YellowTaxis AS 대상 YellowTaxiMergeData AS 소스
ON target.Rideld = source.Rideld 사용 일치 시
```

업데이트 세트 *

일치하지 않을 때

 삽입 *

 -- 소스와 일치하지 않는 대상의 레코드를 삭제합니다.

 소스와 일치하지 않는 경우

 삭제

특정 조건을 충족하는 원본 테이블에 더 이상 존재하지 않는 대상 테이블의 레코드(예: 일시 삭제)에 플래그를 지정하려면 MERGE 조건과 UPDATE를 지정할 수 있습니다.

```
%sql
MERGE INTO Taxidb.YellowTaxis AS 대상 YellowTaxiMergeData AS 소스
사용 target.Rideld = source.Rideld

일치하는 경우
    업데이트 세트 *
    일치하지 않을 때
        삽입 *
        -- 대상 테이블에 레코드가 있는 경우 target.status = 'inactive'로 설정 -- 소스 테이블에 레코드가 없고 소스와 일치하지 않을 때 조건이 충족되는 경우 target.PickupTime >= (current_date() - INTERVAL '5' DAY)
```

그 다음에

 업데이트 설정 target.status = '비활성'

WHEN NOT MATCHED BY SOURCE 절을 UPDATE 또는 DELETE 대상 행에 추가할 때 선택적 MERGE 조건을 추가하는 것이 가장 좋습니다. 이는 지정된 MERGE 조건이 없으면 많은 수의 대상 행이 수정될 수 있기 때문입니다. 따라서 최상의 성능을 위해서는 WHEN NOT MATCHED BY SOURCE 절에 MERGE 조건을 적용하여 (예: 이전 코드 예에서 target.PickupTime >= (current_date() - INTERVAL '5' DAY) 대상 행 수를 제한하세요).

업데이트되거나 삭제되는 중입니다. 왜냐하면 해당 행에 대해 해당 조건이 true인 경우에만 대상 행이 수정되기 때문입니다.

MERGE 작업에 WHEN NOT MATCHED BY SOURCE 절을 여러 개 추가할 수도 있습니다. 여러 절이 있는 경우 지정된 순서대로 평가되며 마지막 절을 제외한 모든 WHEN NOT MATCHED BY SOURCE 절에는 조건이 있어야 합니다.

DESCRIBE HISTORY를 사용하여 MERGE 작업 분석

출력의 OperationsParameters 섹션에 있는 YellowTaxis 테이블에서 DESCRIBE HISTORY를 실행하면 MERGE 조건자를 볼 수 있습니다.

작업: 병합

```
[('predicate',
  '(target.RideId = source.RideId)'),
  ('matchedPredicates', '[{"actionType": "update"}]'),
  ('notMatchedPredicates', '[{"actionType": "끼워 넣다"}]')]
```

조인 조건 (target.RideId = source.RideId), 업데이트를 지정하는 matchPredicate, 삽입을 지정하는 notMatchedPredicate를 볼 수 있습니다.

OperationMetrics 출력 섹션에는 다양한 작업의 세부정보가 표시됩니다.

```
[('numTargetRowsCopied', '4469890'),
 ('numTargetRowsDeleted', '0'),
 ('numTargetFilesAdded', '4'),
 ('executionTimeMs', '91902'),
 ('numTargetRowsInserted', '6'),
 ('scanTimeMs', '8452'),
 ('numTargetRowsUpdated', '5'),
 ('numOutputRows', '4469901'),
 ('numTargetChangeFilesAdded', '0'),
 ('numSourceRows', '11'),
 ('numTargetFilesRemoved', '1'),
 ('rewriteTimeMs', '16782')]
```

여기서도 6개의 행이 삽입되고 (numTargetRowsInserted) 5개의 행이 업데이트된 (numTargetRowsUpdated) 것을 확인할 수 있습니다. 4개의 새로운 데이터 파일이 델타 테이블에 추가되었고 1개의 데이터 파일이 제거되었습니다.

MERGE 작업의 내부 작동

내부적으로 Delta Lake는 다음 두 단계로 다음과 같은 MERGE 작업을 완료합니다.

1. 먼저 대상 테이블과 원본 테이블 간의 내부 조인을 수행하여 일치하는 항목이 포함된 모든 데이터 파일을 선택합니다. 이렇게 하면 작업에서 안전하게 무시할 수 있는 데이터를 불필요하게 쇄는 것을 방지할 수 있습니다.
2. 그런 다음 대상 테이블과 원본 테이블에서 선택한 파일 사이에 외부 조인을 수행하고 사용자가 지정한 대로 적절한 INSERT, DELETE 또는 UPDATE 절을 적용합니다.

MERGE가 내부적으로 UPDATE 또는 DELETE 와 다른 주요 방법은 Delta Lake가 조인을 사용하여 MERGE를 완료한다는 것입니다. 이를 통해 성능을 향상시키려고 할 때 몇 가지 독특한 전략을 사용할 수 있습니다.

결론

DELETE, UPDATE, MERGE 와 같은 DML 작업은 모든 테이블 형식 및 ETL 작업에 필수적인 작업이며 모두 트랜잭션 로그를 통해 활성화됩니다.

이러한 작업을 활용하면 데이터 플랫폼에서 데이터 변경 사항을 효율적으로 처리하고 데이터 무결성을 유지할 수 있습니다.

기존 RDBMS의 테이블과 마찬가지로 이 장에서는 델타 테이블을 사용하여 DELETE, UPDATE 및 MERGE 작업을 수행할 수 있지만 SQL 또는 DataFrame API를 사용하여 이러한 작업을 적용할 수도 있다는 내용을 읽었습니다. 더 중요한 것은 Delta 테이블 디렉터리의 기본 파일을 사용하여 Delta Lake의 내부에서 어떤 일이 발생하는지, 그리고 트랜잭션 로그가 이러한 다양한 유형의 항목을 기록하고 추적하는 방법을 배웠다는 것입니다. DESCRIBE HISTORY 명령을 사용하면 테이블 트랜잭션의 출력력에 대한 세부 정보를 볼 수 있습니다. 이러한 각 작업은 조건자를 활용하여 검색된 파일 수를 줄이고 성능을 향상시킬 수도 있습니다. 작업 중 조건자를 사용하는 것 외에도 다음 장에서 배우게 될 델타 테이블에 적용할 수 있는 다른 성능 조정 기술이 있습니다.

제5장

성능 튜닝

기존 RDBMS 또는 Delta 테이블을 사용하여 데이터를 저장하고 검색할 때마다 기본 스토리지 형식으로 데이터를 구성하는 방법은 테이블 작업 및 쿼리를 수행하는 데 걸리는 시간에 큰 영향을 미칠 수 있습니다. 일반적으로 성능 튜닝은 시스템 성능을 최적화하는 프로세스를 의미하며, 델타 테이블의 맥락에서 여기에는 데이터 저장 및 검색 방법 최적화가 포함됩니다. 역사적으로 데이터 검색은 더 빠른 처리를 위해 RAM 또는 CPU를 늘리거나 관련 없는 데이터를 건너뛰어 읽어야 하는 데이터 양을 줄이는 방식으로 수행되었습니다. Delta Lake는 작업 중에 읽어야 하는 파일 및 데이터의 양을 효율적으로 줄여 데이터 검색을 가속화하기 위해 결합할 수 있는 다양한 기술을 제공합니다.

Apache Spark 및 Delta Lake에서 읽기 속도 저하와 비효율적인 처리에 기여할 수 있는 또 다른 문제는 1 장에서 간략하게 언급한 작은 파일 문제입니다. 작은 파일 문제는 기본 데이터 파일이 수많은 작은 파일로 분할될 때 발생할 수 있는 문제입니다. 더 크고 효율적인 파일과 반대되는 파일입니다.

주로 빈번한 쓰기로 인해 여러 가지 이유로 발생할 수 있지만 작은 파일을 더 큰 파일로 압축하는 것을 포함하는 Delta Lake의 다양한 기술을 통해 해결할 수 있습니다.

좋은 성능 조정 전략을 활용하여 작은 파일 문제의 영향을 줄이고 델타 테이블에서 데이터 건너뛰기를 더 효과적으로 활성화하면 특히 대규모 테이블이나 리소스 집약적인 데이터 레이크 작업 및 쿼리를 처리할 때 실행 시간 성능을 크게 향상시킬 수 있습니다.

데이터 건너뛰기

관련 없는 데이터를 건너뛰는 것은 읽어야 하는 데이터 양을 줄이는 것을 목표로 하기 때문에 궁극적으로 대부분의 성능 튜닝 기능의 기초입니다. 이 기능은,

데이터 건너뛰기라고 하는 기능은 Delta Lake의 다양한 기술을 통해 향상될 수 있습니다.

Delta Lake는 파일의 최대 32개 필드에 대한 최소값과 최대값을 자동으로 유지하고 해당 값을 메타데이터의 일부로 저장합니다. Delta Lake는 이러한 최소 및 최대 범위를 사용하여 쿼리 필드 값 범위를 벗어난 파일을 건너뜁니다. 이는 데이터 건너뛰기 통계를 통해 데이터 건너뛰기를 가능하게 하는 핵심 측면입니다.

이 기능은 Delta Lake에서 적용 가능할 때마다 활성화되므로 데이터 건너뛰기 및 데이터 통계를 구성하거나 지정 할 필요가 없지만 효율성은 데이터 레이아웃에 따라 크게 달라집니다. 데이터 건너뛰기의 효율성을 최대화하기 위해 OPTIMIZE 및 ZORDER BY와 같은 명령을 사용하여 데이터를 통합, 클러스터링 및 같은 위치에 배치할 수 있습니다. 이에 대해서는 후속 섹션에서 자세히 설명하므로 최소 및 최대 범위가 좁고, 이상적으로는 겹치지 않는 것입니다.

Delta Lake는 각 데이터 파일에 대해 다음과 같은 데이터 건너뛰기 통계를 수집합니다.

- 레코드 수
- 처음 32개 열 각각의 최소값
- 처음 32개 열 각각의 최대값
- 처음 32개 열 각각에 대한 Null 값 수

Delta Lake는 테이블 스키마에 정의된 처음 32개 열에서 이러한 통계를 수집합니다. 중첩된 열(예: StructType1) 내의 각 필드는 열로 계산됩니다. 스키마의 열 순서를 변경하여 특정 열에 대한 통계 수집을 구성하거나 delta.dataSkippingNumIndexedCols를 사용하여 통계를 수집할 열 수를 늘릴 수 있습니다. 그러나 추가 열을 추가하면 쓰기 성능에 부정적인 영향을 미칠 수 있는 추가 오버헤드도 추가됩니다. 일반적으로 필터, WHERE 절, 조인 및 집계를 수행하는 경향이 있는 열에 일반적으로 사용되는 열에 대한 데이터 건너뛰기 통계를 수집하려고 합니다.

반대로 긴 문자열에 대한 데이터 건너뛰기 통계는 데이터 건너뛰기 목적에 비해 훨씬 덜 효율적이므로 수집하지 마세요.

그림 5-1에서는 기본적으로 처음 32개 열에 대한 통계만 테이블에 수집되는 것을 보여줍니다. 그리고 통계 수집을 위해 중첩 열 내의 각 필드는 개별 열로 간주됩니다.

1 Apache Spark 데이터 유형

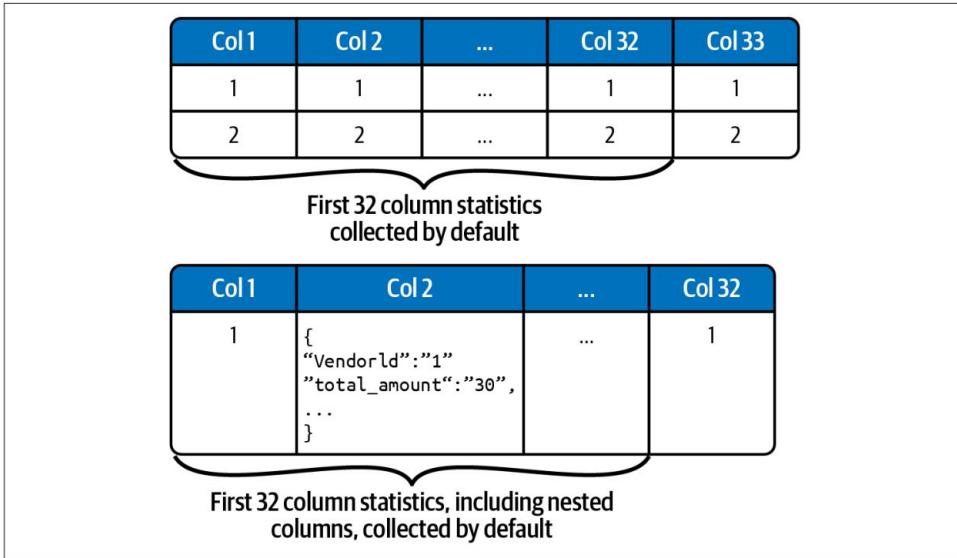


그림 5-1. 처음 32개 열에서 수집된 데이터 건너뛰기 통계

다음 예제는 지정된 위치에 델타 테이블을 생성하는 5 장의 "Chapter 초기화" 노트북을 실행한 후 실행해야 하는 "데이터 건너뛰기" 노트북에서 찾을 수 있습니다. "데이터 건너뛰기" 노트북의 스크립트는 셀 명령을 사용하여 생성된 델타 테이블의 트랜잭션 로그에서 마지막 파일 추가 작업을 살펴봅니다. 이는 마지막 트랜잭션 항목에서 수집된 데이터 건너뛰기 통계의 예를 보여줍니다.

```
%sh
# 델타 테이블에 대한 경로 정의
delta_table_path='mnt/datalake/book/chapter05/YellowTaxisDelta/'

# 마지막 트랜잭션 항목을 찾고 "add"를 검색합니다. # 출력에는 json에 저장된 파일 통계가 표시됩니다.
# 추가된 마지막 파일에 대한 트랜잭션 항목 grep "\\"add"\\" \"$(_ls -1rt /dbfs/$delta_table_path/_delta_log/
*.json | tail -n1)" | sed -n 1p > /tmp/commit.json
python -m json.tool < /tmp/commit.json
```

그러면 다음과 같은 출력이 생성됩니다(관련 부분만 표시됨).

```
통계: {"numRecords":12177114,"minValues":{"VendorID":1,
"tpep_pickup_datetime":"2022-01-01","maxValues":{"VendorID":6,"tpep_pickup_datetime":
"2022-11-01"}, "nullCount":{"VendorID":0,"tpep_pickup_datetime":0}}
```



읽을 수 있는 형식과 프로그래밍 방식으로 JSON 파일의 적절한 항목을 표시하기 위해 셀 명령을 사용하여 트랜잭션 로그에 파일 추가 명령을 표시하고 있습니다. 이 파일은 멜타 테이블이 저장된 스토리지 위치에 기록되므로 해당 위치로 이동하여 트랜잭션 로그에서 적절한 JSON 파일을 열어 정보를 볼 수도 있습니다.

이 출력에서 최소값과 최대값이 Null 수 또는 Null 개수와 함께 마지막 파일에 캡처되었음을 확인할 수 있습니다. 32개 미만의 열이 포함되어 있으므로 테이블의 모든 열에 대해 통계가 수집되었습니다. 이 메타데이터는 작업 중에 추가된 모든 파일에 대해 수집됩니다.

테이블에 32개가 넘는 열이 포함된 경우 테이블 속성 delta.dataSkippingNumIndexedCols를 사용하여 통계가 수집되는 열 수를 변경할 수도 있습니다.

```
%sql
ALTER TABLE 테이
    블_이름
세트
TBLPROPERTIES('delta.dataSkippingNumIndexedCols' = '<값>');
```



delta.dataSkippingNumIndexed Cols 와 같은 Delta Lake 속성은 Spark 구성 설정을 사용하여 설정할 수도 있습니다.

문자열이나 바이너리와 같은 긴 값에 대한 통계를 수집하는 것은 비용이 많이 드는 작업일 수 있으므로 일부 열에서는 최소값과 최대값을 수집하는 것이 효과적이지 않을 수 있습니다. 긴 값이 포함된 열을 방지하도록 테이블 속성 delta.dataSkippingNumIndexedCols를 구성하거나 2를 사용하여 긴 값이 포함된 열을 delta.dataSkippingNumIndexedCols 보다 큰 열로 이동할 수 있습니다. [7장에서는](#) 테이블 스키마 업데이트에 대해 설명하고 테이블 변경 열 변경. 순서를 더 자세히 변경합니다.

파티셔닝

작업(예: 데이터 건너뛰기) 중에 읽어야 하는 데이터 양을 더욱 줄이고 대규모 테이블의 성능을 높이기 위한 노력의 일환으로 Delta Lake 파티셔닝을 사용하면 데이터를 파티션이라는 더 작은 청크로 나누어 멜타 테이블을 구성할 수 있습니다..

² [ALTER TABLE Delta Lake 설명서](#)



이 섹션에 설명된 분할은 DataFrame을 처리할 때 Spark가 적용하는 분할을 설명하지 않습니다.

오히려 이 장의 파티셔닝은 연도=2023과 같은 키 값 쌍을 포함하는 경로로 데이터가 구성되는 온디스크 또는 Hive 스타일 파티셔닝을 참조합니다.

INSERT, UPDATE, MERGE 및 DELETE와 같은 데이터 조작 명령뿐만 아니라 테이블에 대한 쿼리 속도를 높일 수 있는 테이블의 하나 이상의 열(가장 일반적인 날짜)에 있는 값을 기반으로 파티션을 생성할 수 있습니다.



이 글을 쓰는 시점에서는 데이터 레이아웃과 관련하여 데이터 건너뛰기를 활성화하는 데 파티션이 권장되는 접근 방식입니다. 이 장의 마지막 섹션에서 배우게 될 액체 클러스터링이라는 Delta Lake의 새로운 기능은 현재 미리 보기 상태이며 파티션과 호환되지 않습니다. 이는 데이터 레이아웃과 관련하여 쿼리 성능을 최적화하기 위해 권장되는 접근 방식으로 파티션을 대체합니다. 우리는 이러한 명령을 자동화하고 대체하는 기능을 배우기 전에 파티션의 작동하는 방식과 이를 수동으로 적용하는 방법을 이해하는 것이 중요하다고 느꼈습니다. 새로운 기능인 액체 클러스터링은 가까운 시일 내에 일반 출시될 예정입니다. [Delta Lake 문서 웹 사이트](#)를 검토하면 액체 클러스터링 상태에 대해 자세히 알아보고 최신 상태를 유지할 수 있습니다. 그리고 이 [기능 요청](#).

테이블을 분할하면 기본 데이터 세트가 각 파티션에 대해 서로 다른 디렉터리와 하위 디렉터리로 구성됩니다 ([그림 5-2](#)).

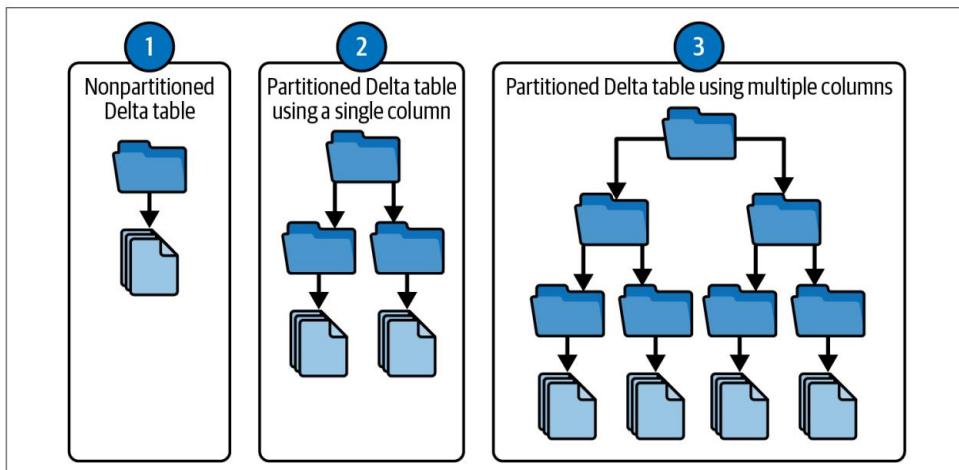


그림 5-2. 다양한 디렉터리와 하위 디렉터리로 구성된 델타 테이블의 기본 데이터 파일

그림 5-2 의 번호가 매겨진 단계는 다음을 보여줍니다.

1. 파티션이 없는 Delta 테이블은 단일 디렉터리로 구성됩니다.
2. 단일 열로 분할된 델타 테이블에는 각 파티션 값에 대해 생성된 디렉터리가 있습니다.
3. 여러 열로 분할된 델타 테이블에는 각 파티션 값에 대해 생성된 디렉터리가 있으며, 그런 다음 파티션에 정의된 각 추가 열에 대해 하위 디렉터리가 생성됩니다.



Delta Lake에서 파티션은 델타 테이블을 분할하지 않는 것과 달리 많은 경우 성능이 저하될 위험이 있습니다. 이는 파티션이 이 책의 앞부분과 이 장의 뒷부분에서 논의한 작은 파일 문제를 일으킬 수 있기 때문입니다. 특히 여러 열로 파티션을 나눌 때 더욱 그렇습니다. 파티셔닝은 거의 권장되지 않습니다. 델타 테이블에 파티션을 적용하기 전에 [108페이지의 "파티셔닝 경고 및 고려 사항"](#)을 참조하십시오.

데이터 세트의 모든 파일을 검색하는 대신 파티션을 선택적으로 쿼리할 수 있는 경우 Delta Lake는 적절한 디렉터리(또는 디렉터리) 또는 파티션을 신속하게 검색하여 작업을 수행하므로 작업이 더 빨라집니다. Delta Lake는 테이블에 있는 파티션 집합을 자동으로 추적하고 데이터가 추가되거나 제거될 때 목록을 업데이트하므로 새 파티션을 설명하기 위해 ALTER TABLE을 실행할 필요가 없습니다.

분할된 테이블을 생성하려면 SQL을 사용하여 테이블 정의에서 PARTITIONED BY 절을 사용할 수 있습니다.

```
%sql
--SQL을 사용하여 분할된 테이블 생성
CREATE TABLE tripData(PickupMonth INTEGER,
                      공급업체 ID INTEGER,
                      총 금액 DOUBLE)
분할 기준(피업월)

--PARTITION 사양을 사용하여 테이블에 INSERT
tripData에 삽입
PARTITION(PickupMonth= '12') (VendorId, TotalAmount)
12월TripData에서 VendorId, TotalAmount를 선택합니다.

-- 파티션 삭제
ALTER TABLE 학생 DROP PARTITION(PickupMonth = '12');
```

다음 스크립트3 는 "02 - Partitioning" 노트북에서 찾을 수 있습니다.
 Parquet 파일에서 분할된 델타 테이블을 작성하는 방법과 추가하는 방법을 설명합니다.
 파티션을 나눌 열:

```
# 가져오기 모듈
pyspark.sql.functions 가져오기(월, to_date)에서

## 원하는 경로로 Destination_path 업데이트 ##
# 델타 테이블 대상 경로를 정의합니다.
Destination_path = '/mnt/datalake/book/chapter05/YellowTaxisPartitionedDelta/'

# Delta 테이블을 읽고, 파티션에 열을 추가하고, # 파티션을 사용하여 씁니다.

# 테이블이 이미 존재하는 경우 # 파티션을 추가하므로 기존 스키마를 덮어써야 합니다.

Spark.table('taxidb.tripData') .withColumn('PickupMonth',
Month('PickupDate')) .withColumn('PickupDate',
to_date('PickupDate')) .write .partitionBy('PickupMonth') .format( "델
타") .option("overwriteSchema",
"true") .mode("덮어쓰
기") .save(destination_path)
```

```
# Hive에 테이블 등록
Spark.sql(f"""존재하지 않는 경우 테이블 생성 Taxdb.tripDataPartitioned
델타 위치 '{destination_path}' """ 사용 )
```



GitHub 리포지토리를 사용하여 팔로우하는 경우 다음 사항에 유의하세요.
 파일 위치는 다음의 파일 위치와 다를 수 있습니다.
 저장소에 있는 노트북입니다. 그에 따라 업데이트하십시오.

Delta 테이블의 파티션을 보려면 SHOW PARTITIONS 명령을 사용할 수 있습니다.

```
%sql
--테이블의 모든 파티션을 나열합니다.
파티션 표시 Taxdb.tripDataPartitioned
```

3 GitHub 저장소 위치: /chapter05/01 - 압축

그러면 다음과 같은 출력이 생성됩니다.

```
+-----+
| 학업월 |
+-----+
| 1 |
2
| ... | 12
|
+-----+
```

이 출력에서는 Delta 테이블이 PickUpMonth 로 분할되어 있고 매월 분할이 있는 것을 볼 수 있습니다.



스키마를 덮어쓰거나 기존 델타 테이블의 분할을 변경하려면 .option("overwriteSchema", "true")를 설정하세요.

기본 파일 시스템에서 파티션이 어떻게 구성되어 있는지 보려면 Delta 테이블이 있는 곳에 생성된 딕터리를 볼 수도 있습니다. 실제 파일 시스템을 살펴보기 때문에 오래되었거나 존재하지 않는 파티션이 표시될 수도 있다는 점을 명심하세요. 이 테이블은 해당 장의 스크립트를 사용하여 생성되었으므로 관련 파티션만 있어야 합니다.

```
# OS 모듈 가져오기 OS
가져오기
```

```
# 다음 bash 스크립트에서 이 변수를 사용할 수 있도록 환경 변수를 만듭니다. # os.environ['destination_path'] = '/dbfs' +
Destination_path
```

```
# 딕터리에 있는 파일과 딕터리를 나열합니다.
print(os.listdir(os.getenv('destination_path')))
```

그러면 다음과 같은 출력이 생성됩니다.

```
['학업월=1', '학업월=10', '학업월=11', '학업월=12',
'학업월=2', '학업월=3', '학업월=4', '학업월=5',
'학업월=6', '학업월=7', '학업월=8', '학업월=9', '_delta_log']
```

이 출력에서 Delta 테이블에는 트랜잭션 로그 딕터리인 _delta_log뿐만 아니라 파티션의 각 값(이 경우 1~12개월)에 대한 딕터리도 포함되어 있습니다.

각 파티션의 딕터리 값은 각 파일 추가 작업의 일부인 메타데이터 항목으로 트랜잭션 로그에도 포함됩니다. 이 메타데이터 항목은 다음을 수행할 수 있습니다.

트랜잭션 로그에서 파일 추가 작업을 보고 partitionValues를 보면 현재 테이블에 표시됩니다.

```
%ssh
# 마지막 트랜잭션 항목을 찾고 "add"를 검색하여 추가된 파일을 찾습니다. # 출력에 partitionValues grep "\"add\" \"$!(ls -1rt $destination_path/
_delta_log/*.json | tail -n1)" |
sed -n 1p > /tmp/commit.json | sed -n 1p > /tmp/commit.json
python -m json.tool < /tmp/commit.json
```

그리면 다음과 같은 출력이 생성됩니다(관련 부분만 표시).

```
{
  "추가": {
    "경
    로": "PickupMonth=12/part-00000....c000.snappy.parquet", "partitionValues":
    { "PickupMonth": "12"
  }
}
```

이 메타데이터로 인해 파티셔닝은 본질적으로 데이터 건너뛰기와 동일합니다. 그러나 이 장의 뒷부분에서 자세히 알아볼 주제인 데이터 통계를 기반으로 데이터 건너뛰기를 수행하는 대신, 데이터 건너뛰기는 파일 필터링에 도움이 되는 파티션 값인 문자열의 정확한 일치를 기반으로 합니다.

Delta Lake를 사용하면 3 장에서 배운 교체 위치를 사용하여 지정된 파티션을 쉽게 업데이트할 수 있습니다. 12월에 결제 유형이 4인 경우 다음과 같은 비즈니스 요구 사항이 있다고 가정해 보겠습니다. 5로 업데이트되었습니다. 다음 PySpark 표현식을 교체 위치와 함께 사용하여 해당 결과를 얻을 수 있습니다.

```
# SQL 함수에서 월 가져오기 from
pyspark.sql.functions import lit from pyspark.sql.types
import LongType

# 지정된 파티션을 업데이트하려면 교체 위치를 사용하십시오.

Spark.read .format("delta") .load(destination_path) .where("PickupMonth == '12'
and PaymentType == '3' ") .withColumn("PaymentType", lit(4).캐스트(종타입()))
.쓰다
.format("델
타") .option("replaceWhere", "PickupMonth = '12'") .mode("덮어
쓰기") .save(destination_path) \\
```



교체 위치를 사용하는 경우 델타 테이블 스키마를 덮어쓸 수 없습니다.

이전 명령에서 WHERE 절을 사용하여 단일 파티션만 로드했다는 점에 유의하세요. 파티션을 직접 읽을 필요는 없지만 WHERE 절(Spark SQL) 또는 .where() 함수(DataFrame API)를 사용하면 다음과 같은 데이터 건너뛰기가 가능합니다.

```
# Delta 테이블의 파티션을 DataFrame으로 읽습니다. df =  
spark.read.table("<delta_table_path>").where("PickupMonth = '12'"")
```

데이터를 읽는 데 .where()를 사용하는 것이 매우 효과적일 수 있지만 압축, OPTIMIZE 및 ZORDER BY와 같은 성능 조정 명령과 함께 .where()를 사용하여 지정된 파티션에서만 해당 작업을 수행할 수도 있습니다. (에스). 이는 특정 파티션에 새 데이터를 쓸 때(예: 당월 데이터 삽입) 특히 유용할 수 있습니다. WHERE 절이나 .where() 함수를 사용하지 않으면 기본적으로 전체 테이블을 검색합니다.

예를 들어 단일 파티션에서 압축을 수행할 수 있습니다.

```
# Delta 테이블에서 파티션을 읽고 다시 파티션을 나눕니다.
```

```
\  
\  
\  
\  
\  
\  
\  
\  
\  
\
```

```
Spark.read.format("delta").load(destination_path).where("PickupMonth = '12' ").repartition(5).write.option("dataChange", "false").format("델타").mode("덮어쓰기")
```

다음을 사용하여 지정된 파티션에서 OPTIMIZE 및 ZORDER BY를 쉽게 수행할 수도 있습니다.
SQL:

```
%sql  
OPTIMIZE Taxdb.tripData WHERE PickupMonth = 12 ZORDER BY tpep_pickup_datetime
```

파티셔닝 경고 및 고려 사항

파티션은 특히 매우 큰 테이블의 경우 매우 유용할 수 있지만 테이블을 파티셔닝할 때 고려해야 할 몇 가지 사항이 있습니다.

- 파티션 열을 신중하게 선택하십시오. 열의 카디널리티가 매우 높으면 해당 열을 분할에 사용하지 마십시오. 예를 들어, 백만 개의 개별 타임스탬프를 가질 수 있는 열 타임스탬프를 기준으로 분할하는 것은 잘못된 분할 전략입니다. 높은 카디널리티 열은 Z 순서 지정에는 적합하지만 분할에는 적합하지 않습니다. 왜냐하면 이 장의 시작 부분에서 설명한 것과 동일한 작은 파일 문제가 발생할 수 있기 때문입니다. 이것이 이전 예에서 날짜 열을 추가한 이유입니다. 이는 적절한 분할 열 역할을 하는 데 도움이 됩니다.

가장 일반적으로 사용되는 파티션 열은 일반적으로 날짜입니다.

- 해당 파티션의 데이터가 1GB 이상일 것으로 예상되는 경우 열을 기준으로 파티션을 나눌 수 있습니다. 더 적고 큰 파티션이 있는 테이블은 더 작은 파티션이 많은 테이블보다 성능이 뛰어난 경향이 있습니다. 그렇지 않으면 작은 파일 문제가 발생합니다.
- 파티셔닝에 사용되는 컬럼은 테이블 생성 시 컬럼 사양(각 컬럼의 이름과 데이터 타입)에서 파티션 컬럼을 명시적으로 정의하지 않는 한 항상 테이블의 끝으로 이동한다.
- 파티션이 있는 테이블을 생성하면 쿼리 패턴이나 파티션 요구 사항이 변경되더라도 해당 파티션을 변경할 수 없습니다. 파티션은 고정된 데이터 레이아웃으로 간주되며 파티션 진화를 지원하지 않습니다.
- 분할 전략에 대한 마법의 비법은 없습니다. 단지 고려해야 할 지침일 뿐입니다. 데이터, 세분성, 수집 및 업데이트 패턴 등에 따라 다릅니다.

컴팩트 파일

델타 테이블에서 DML 작업을 수행할 때 파티션 전체에 걸쳐 많은 작은 파일에 새 데이터가 기록되는 경우가 많습니다. 파일 메타데이터의 추가 볼륨과 읽어야 하는 데이터 파일의 총 개수로 인해 쿼리 및 작업 속도가 느려질 수 있습니다. 중요한 점은 앞서 언급한 작은 파일 문제입니다.

이 문제를 방지하려면 다수의 작은 파일을 16MB보다 큰 소수의 큰 파일로 다시 작성해야 합니다. Delta Lake는 작은 파일을 더 큰 파일로 통합하는 다양한 방법을 통해 스토리지의 데이터 레이아웃을 최적화하는 기능을 지원합니다.

압축 파일 통합

을 압축 또는 빙 패킹이라고 합니다. 예를 들어 Delta 테이블을 압축할 파일 수를 지정하는 등 자체 사양을 사용하여 압축을 수행하려면 `dataChange = false`인 DataFrame 기록기를 사용할 수 있습니다. 이는 작업이 데이터를 변경하지 않음을 나타냅니다. 단순히 데이터 레이아웃을 다시 정렬합니다.



데이터가 압축되면 Delta Lake는 기본적으로 `dataChange = true`를 설정합니다. 이로 인해 테이블이 스트리밍 소스로 사용될 때 다운스트림 스트리밍 소비자와 같은 테이블의 동시 작업이 중단될 수 있습니다.

반대로 `dataChange = false`를 사용하는 경우 데이터를 변경하는 작업으로 인해 테이블의 기본 데이터가 손상될 수 있습니다. 데이터 변경 사항이 없는 경우에만 `dataChange = false`를 사용하는 것이 가장 좋습니다. 이 옵션을 사용하면 모든 다운스트림 소비자에게 작업이 데이터만 다시 정렬한다는 사실을 알 수 있으므로 해당 소비자는 트랜잭션 로그의 이벤트를 무시할 수 있습니다.

다음 예는 "03 - 압축, 최적화 및 ZOrder" 노트북의 1단계에서 찾을 수 있습니다. 노트북의 스크립트는 Spark DataFrame의 파티션 수를 늘리거나 줄이는 데 사용되는 방법인 재파티션 과 함께 DataFrame 작성기를 사용하는 방법과 자체 알고리즘을 사용하여 데이터를 5개 파일로 압축하는 dataChange = False 옵션을 보여줍니다.

```
# 다시 분할할 파일의 경로와 수를 정의합니다. path = "/mnt/datalake/book/chapter05/
YellowTaxisDelta" numberOffiles = 5
```

```
# 델타 테이블을 알고 다시 파티션을 나눕니다. Spark.read \ .format("delta")
\ .load(path) \ .repartition(numberOffiles) \ .write
\ .option("dataChange", "false") \ .format("델타")
\ .mode("덮어쓰기") \ .save(경로)
```

최적화

압축을 사용하면 작은 파일을 큰 파일로 통합하는 방법을 지정할 수 있습니다.

Delta Lake에서 이 압축을 트리거하고 Delta Lake가 원하는 대용량 파일의 최적 수를 결정하도록 하는 보다 최적의 방법은 OPTIMIZE 명령을 사용하는 것입니다.

OPTIMIZE 명령은 트랜잭션 로그에서 불필요한 파일을 제거하는 동시에 파일 크기 측면에서 균등하게 균형 잡힌 데이터 파일을 생성하는 것을 목표로 합니다. 더 작은 파일은 최대 1GB의 더 큰 새 파일로 압축됩니다.

그림 5-3은 OPTIMIZE가 작은 파일을 큰 파일로 통합하는 방법을 보여줍니다. OPTIMIZE는 파일 내에서 데이터가 구성되는 방식을 고려하지 않습니다. 파일을 다시 정렬하고 통합하기만 합니다. 다음 섹션에서는 파일 내에서 데이터를 구성하는 방법에 대해 자세히 알아봅니다.

그림 5-3에서 볼 수 있듯이 :

1. Delta 테이블은 특별한 정보가 없는 데이터를 포함하는 작은 파일로 구성됩니다. 주문하다. 이 경우 각각 2개의 행이 있는 4개의 파일이 있습니다.
2. OPTIMIZE를 실행하여 실행 중에 읽어야 하는 파일 수를 줄입니다. 운영.
3. 델타 테이블의 작은 파일은 최대 1GB의 더 큰 새 파일로 압축됩니다. 이 경우 각각 4개의 행이 있는 두 개의 파일이 있습니다.

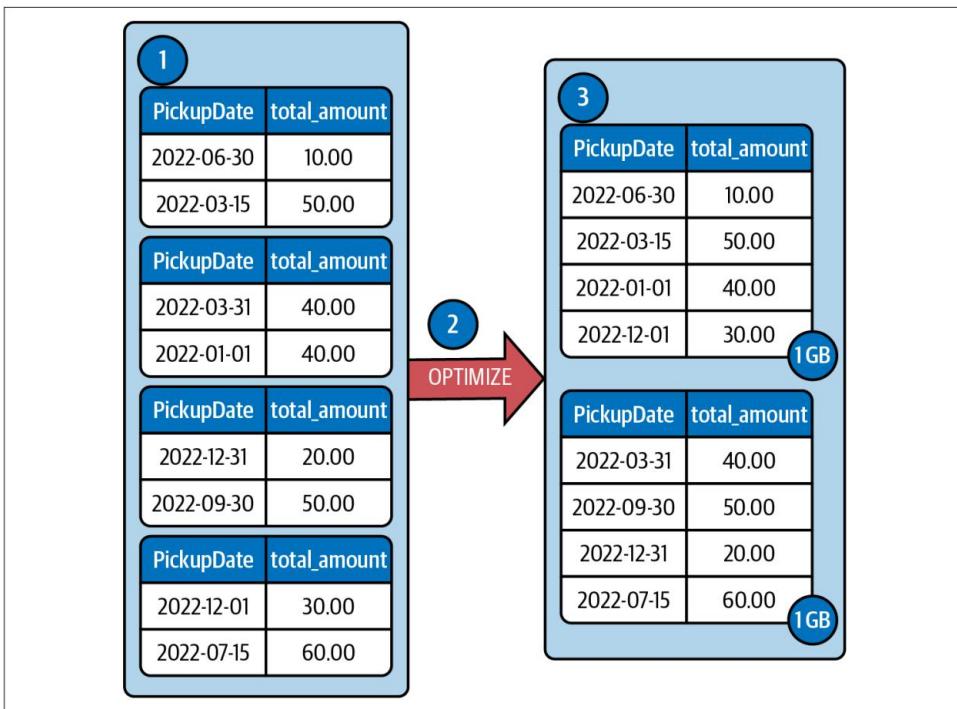


그림 5-3. OPTIMIZE 전후의 데이터 파일



재분할 방법을 통해 달성되는 압축과 달리 dataChange 옵션을 지정할 필요가 없습니다. OPTIMIZE는 명령을 수행할 때 스냅샷 격리를 사용하므로 동시 작업과 다운스트림 스트리밍 소비자가 중단 없이 유지됩니다.

OPTIMIZE 의 예를 살펴보겠습니다 . 노트북 “03 - Compaction, Optimize and ZOrder”를 사용하여 2단계(1단계는 압축 섹션에서 실행됨)를 실행하여 기존 테이블을 1,000개의 파일로 다시 분할하여 테이블에 일관되게 데이터를 삽입하는 시나리오를 시뮬레이션합니다.4 노트북의 3단계에서 OPTIMIZE 명령을 실행합니다. 출력은 작업의 메트릭을 제공합니다.

```
%sql
OPTIMIZE Taxdb.YellowTaxis
```

4 GitHub 저장소 위치: /chapter05/03 - 압축, 최적화 및 ZOrder

출력(관련 부분만 표시됨):

```
+-----+
| 측정항목 |
+-----+
| {"numFilesAdded": 9, "numFilesRemoved": 1000 | | "filesAdded":{...} "totalFiles": 9, | | "totalSize": 2096274374... | | "filesRemoved":{...} "totalFiles": 1000, "totalSize": 2317072851 |
```

테이블에서 OPTIMIZE 명령을 실행하면 1,000개의 파일이 제거되고 9개의 파일이 추가된 것을 확인할 수 있습니다.



제거된 파일의 전체 크기와 추가된 파일의 전체 크기를 비교해 보면, 파일 크기가 약간 늘어나더라도 비교적 동일하게 유지되는 것을 알 수 있습니다. 우리가 사용하는 NYC Taxi 데이터 세트는 주로 정수이므로 데이터 구성 시 압축이 거의 발생하지 않습니다. 데이터에 많은 문자열 값이 포함되어 있는 경우 OPTIMIZE를 실행한 후 훨씬 더 나은 압축을 볼 수 있습니다.

제거된 1,000개의 파일은 기본 스토리지에서 물리적으로 제거되지 않았다는 점에 유의하는 것이 중요합니다. 오히려 트랜잭션으로 그에서는 논리적으로만 제거되었습니다. 이러한 파일은 다음에 VACUUM을 실행할 때 기본 저장소에서 물리적으로 제거됩니다. 이에 대해서는 [6 장](#)에서 자세히 설명합니다.

OPTIMIZE를 사용한 최적화 도 면밀적입니다. 즉, 동일한 테이블이나 데이터 하위 집합에서 두 번 실행되는 경우 두 번째 실행은 아무런 영향을 미치지 않습니다. Taxdb.YellowTaxis 테이블에서 동일한 명령을 다시 실행하면 이 장의 뒷부분에서 자세히 알아볼 데이터 건너뛰기 통계에 0개의 파일이 추가되고 0개의 파일이 제거되었음을 나타냅니다.

```
%sql
OPTIMIZE Taxdb.YellowTaxis
```

출력(관련 부분만 표시됨):

```
+-----+
| 측정항목 |
+-----+
| {"numFilesAdded": 0, "numFilesRemoved": 0 "filesAdded": | | {"...} "totalFiles": 0, "totalSize": 0... |
```

전체 테이블을 최적화하는 대신 데이터의 특정 하위 집합을 최적화할 수도 있습니다. 이는 특정 파티션에 대해서만 DML 작업을 수행하고(이 장의 뒷부분에서 파티션에 대해 자세히 알아볼 예정임) 해당 파티션만 최적화해야 하는 경우에 유용할 수 있습니다. WHERE 절을 사용하여 선택적 파티션 조건지를 지정할 수 있습니다. 정기적으로 데이터를 추가하고 업데이트한다고 가정해 보겠습니다.

이번 달의 파티션; 이 경우에는 12월입니다. 파티션 조건자에 12를 추가한 후 다음 명령을 실행하면 지정된 파티션에 17개의 파일만 제거되고 4개의 파일이 추가되었음을 알 수 있습니다.

```
%sql
OPTIMIZE Taxdb.YellowTaxis WHERE PickupMonth = 12
```

출력(관련 부분만 표시됨):

```
+-----+
| 측정항목
+-----+
| {"numFilesAdded": 4, "numFilesRemoved": 17 "filesAdded": | | {"totalFiles": 4,
"totalSize":1020557526 |
+-----+
```

최적화 고려사항

OPTIMIZE는 쿼리 속도를 향상시키는 데 도움이 되지만 효율성을 보장하기 위해 모든 테이블에서 이 명령을 실행하기 전에 고려해야 할 몇 가지 사항이 있습니다.

- OPTIMIZE 명령은 작성하는 테이블 또는 테이블 파티션에 효과적입니다.
데이터가 계속해서 대량의 작은 파일을 포함하게 됩니다.
- OPTIMIZE 명령은 정적 데이터가 있는 테이블이나 데이터가 거의 업데이트되지 않는 테이블에는 효과적이지 않습니다. 더 큰 파일로 병합할 작은 파일이 거의 없기 때문입니다. • OPTIMIZE 명령은 실행하는 데 시간이 걸리는 리소스 집약적인 작업일 수 있습니다. 작업을 수행하기 위해 컴퓨팅 엔진을 실행하는 동안 클라우드 공급자로부터 비용이 발생할 수 있습니다. 이러한 리소스 집약적인 작업과 테이블에 대한 이상적인 쿼리 성능의 균형을 맞추는 것이 중요합니다.

ZORDER BY

OPTIMIZE는 파일 통합을 목표로 하지만 Z 순서 지정을 사용하면 데이터 레이아웃을 최적화하여 해당 파일의 데이터를 보다 효율적으로 읽을 수 있습니다. ZORDER BY는 명령의 매개변수이며 해당 값을 기준으로 파일에서 데이터가 정렬되는 방식을 나타냅니다.

특히 이 기술은 관련 정보를 동일한 파일 세트에 클러스터링하고 같은 위치에 배치하여 더 빠른 데이터 검색을 가능하게 합니다. 이 동일 지역성은 Delta Lake의 데이터 건너뛰기 알고리즘에서 자동으로 사용됩니다. 이에 대해서는 이 장의 다음 섹션에서 자세히 알아보세요.

Z 순서 인덱스는 지정된 Z 순서 열을 필터링하는 쿼리의 성능을 향상시킬 수 있습니다. 쿼리를 통해 관련 행을 보다 효율적으로 찾을 수 있고, 조인을 통해 일치하는 값이 있는 행을 보다 효율적으로 찾을 수 있으므로 성능이 향상됩니다. 이러한 효율성은 궁극적으로 쿼리 중에 읽어야 하는 데이터 양이 줄어든 덕분이라고 할 수 있습니다.



파티션과 마찬가지로 Z 순서 인덱스는 데이터 레이아웃을 단순화하고 쿼리 성능을 최적화하기 위해 선호되는 기술인 새로운 Delta Lake 기능인 유동 클러스터링으로 곧 대체될 예정입니다. Delta Lake 액체 클러스터링은 사용자 지정 테이블 파티션 또는 Z 순서와 호환되지 않습니다. 이 기능은 현재 미리 보기 상태이며 가까운 시일 내에 일반 공급될 예정입니다. [Delta Lake 문서 웹 사이트를](#) 검토하면 액체 클러스터링 상태에 대해 자세히 알아보고 최신 상태를 유지 할 수 있습니다. 그리고 이 [기능 요청](#).

Z 순서 지정과 결합된 OPTIMIZE를 시연하기 위해 6단계를 실행하여 기존 테이블을 1,000개의 작은 파일로 다시 분할하여 이전 장의 Taxdb.YellowTaxis에서 수행한 최적화를 재설정하고 지웁니다.

```
# 다시 분할할 파일의 경로와 수를 정의합니다. path = "/mnt/datalake/book/chapter05/
YellowTaxisDelta" numberOfFile = 1000
```

```
# Delta 테이블을 읽고 다시 파티션을 나눕니다.
Spark.read.format("delta").load(path).repartition(numberOfFile)
    .쓰다
    .option("dataChange", "false") .format("델
타") .mode("덮어쓰
기") .save(경로)
```

초기 쿼리에 대한 기준을 얻으려면 스크립트에서 기준 쿼리를 실행합니다.

```
%sql
-- 기준 쿼리 -- 결과를 반환하는
데 걸리는 시간을 기록해 둡니다.
선택하다
    COUNT(*)를 개수로,
    SUM(total_amount)을 totalAmount로,
    퍽업 날짜
    에서
        Taxdb.tripData 퍽업 날짜가
        '2022-01-01'과 '2022-03-31' 사이인 경우
        그룹 기준
        퍽업 날짜
```

이 쿼리는 기본 델타 테이블에 작은 파일이 많고 데이터가 특정 순서로 구성되지 않은 경우 실행에 걸리는 시간에 대한 일반적인 기준을 제공합니다. OPTIMIZE 명령 과 함께 ZORDER BY를 적용하여 파일을 통합하고 해당 파일의 데이터를 효과적으로 정렬할 수 있습니다. 결과적으로 데이터를 찾기가 더 쉽기 때문에 쿼리 결과를 가져오는 데 걸리는 시간이 크게 줄어듭니다.

이는 일반적으로 카디널리티가 높은 열과 쿼리 조건자에서 자주 사용되는 열에 사용될 때 가장 효과적입니다. 즉, Z 순서를 적용하는 열이 데이터 검색 성능에 영향을 미칩니다.

```
%sql  
OPTIMIZE Taxdb.tripData ZORDER BY PickupDate
```

이제 Z 순서를 추가했으므로 전략 이름, 입력 큐브 파일 및 ZORDER BY 작업에 대한 기타 통계를 포함하는 출력에서 강조 표시된 자세한 zOrderStats를 볼 수 있습니다.

OPTIMIZE 및 ZORDER BY 명령 이전에 실행된 동일한 기본 쿼리를 실행하면 쿼리 결과를 검색하는 데 걸리는 시간이 크게 증가한 것을 확인할 수 있습니다. 결과를 검색하는 데 걸리는 시간은 클러스터 구성에 따라 다르지만 최적화로 인해 쿼리 결과를 반환하는 데 걸리는 시간이 약 70% 감소한 것을 지속적으로 확인했습니다.

이 경우 Z 순서를 추가하면 쿼리 엔진의 데이터 읽기 효율성이 향상되었으며 OPTIMIZE는 작은 파일을 더 큰 파일로 통합했습니다. 대규모 데이터 세트를 사용하면 이를 표시하기 어려울 수 있지만 그림 5-4에서는 Taxdb.YellowTaxis 테이블을 사용하여 통합 및 정렬이 발생하는 방법을 보여줍니다.

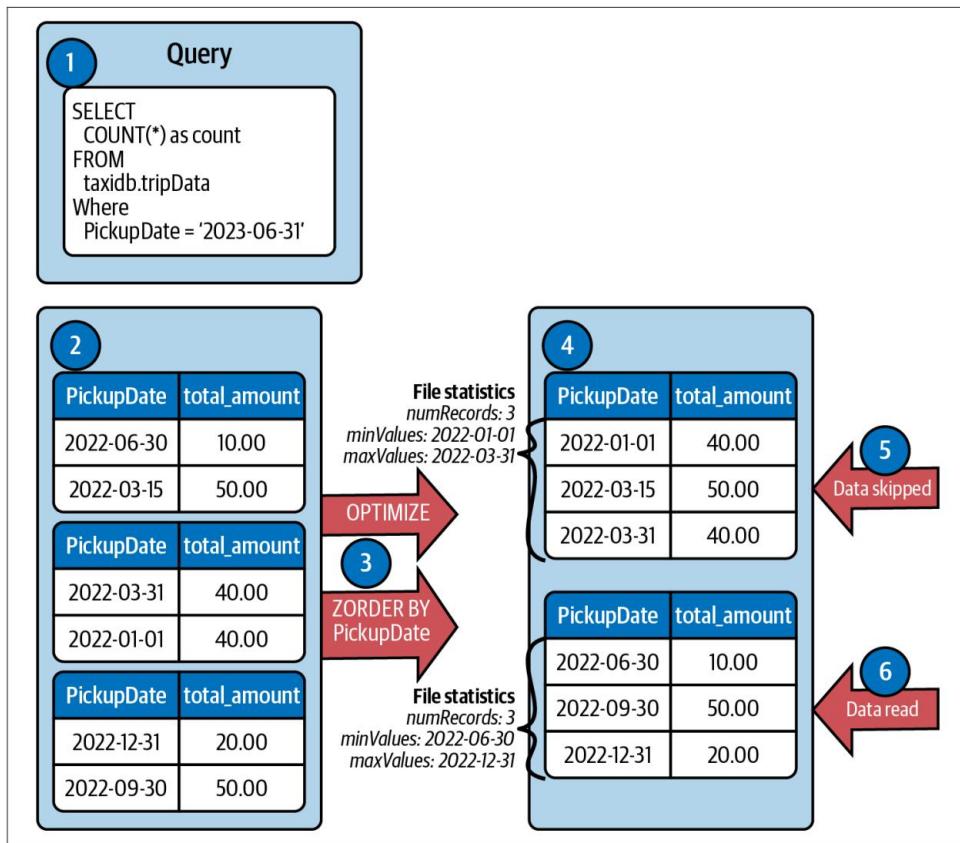


그림 5-4. Taxdb.tripData OPTIMIZE 및 ZORDER BY 전후의 델타 테이블 파일과 데이터 건너뛰기를 사용하여 데이터를 검색하는 방법

그림 5-4 의 번호가 매겨진 단계는 다음을 보여줍니다.

1. TaxDb.YellowTaxis라는 Delta 테이블을 쿼리하고 PickupDate = '2022-06-30'을 기록합니다.
2. Delta 테이블은 특별한 정보가 없는 데이터를 포함하는 작은 파일로 구성됩니다. 주문하다.
3. 쿼리 중 더 나은 성능을 위해 ZORDER BY 와 함께 OPTIMIZE를 실행합니다. 차형.
4. 작은 파일은 더 큰 파일로 합쳐지고 데이터는 Z 순서로 정렬됩니다. PickupDate 열입니다.

5. 쿼리 조건자 Pickup Date = '2022-06-30'을 찾고 있으므로 데이터 건너뛰기가 활용됩니다 . 첫 번째 파일을 건너뛰는 이유는 쿼리 조건자가 데이터 건너뛰기 통계의 최소값 및 최대값 범위를 벗어나기 때문에 쿼리 조건자가 이 파일에 포함되어 있지 않다는 것을 Delta Lake가 알고 있기 때문입니다.

6. 검색 조건자가 최소 및 최대 값 범위에 속하므로 Delta Lake가 이 파일을 검색하는 것을 알고 있으므로 두 번째 파일에서 데이터를 빠르게 읽습니다.

데이터에서 최적화를 실행하기 전에 데이터가 데이터 순서 없이 더 작은 파일로 구성되었음을 알 수 있습니다. 기준 쿼리를 실행할 때 쿼리 엔진은 '2022-01-01' AND '2022-03-31' 사이의 PickupDate가 있는 쿼리 조건자를 찾기 위해 모든 Delta Lake 파일을 검색해야 합니다. ZORDER BY를 사용 하여 OPTIMIZE를 적용하면 데이터가 더 큰 파일로 통합되었으며 데이터는 PickupDate 열을 기준으로 오름차순으로 정렬되었습니다. 이를 통해 쿼리 엔진은 쿼리 조건자를 기반으로 첫 번째 파일을 읽고 두 번째 파일을 무시하거나 건너뛰어 결과를 수집할 수 있었습니다.

ZORDER BY 고려사항

명령에서 ZORDER BY에 대한 여러 열을 쉼표로 구분된 목록으로 지정할 수 있습니다 . 그러나 열이 추가될 때마다 지역성의 효율성이 떨어집니다.

```
%sql
OPTIMIZE Taxdb.tripData ZORDER BY PickupDate, VendorId
```

OPTIMIZE 와 유사하게 Z 순서를 전체 테이블에 적용하는 대신 파티션과 같은 특정 데이터 하위 집합에 적용할 수 있습니다.

```
%sql
OPTIMIZE Taxdb.tripData ZORDER BY PickupDate, VendorId WHERE
PickupMonth = 2022
```



분할에 사용되는 필드에는 ZORDER BY를 사용할 수 없습니다 . ZORDER BY가 OPTIMIZE 명령 과 함께 작동한다는 것을 배웠습니다 . 그러나 파티션 경계를 넘어 파일을 결합할 수 없으므로 Z 순서 클러스터링은 파티션 내에서만 발생할 수 있습니다. 분할되지 않은 테이블의 경우 전체 테이블에서 파일을 결합할 수 있습니다.

쿼리 조건자에서 일반적으로 사용되는 열이 예상되고 해당 열의 카디널리티가 높은 경우(즉, 고유 값이 많은 경우) ZORDER BY를 사용하세요.

OPTIMIZE 와 달리 Z 순서 지정은 멱등성이 아니지만 증분 작업을 목표로 합니다. Z 순서 지정에 소요되는 시간은 여러 번 실행하는 동안 단축된다는 보장이 없습니다.

그러나 Z 순서만 지정된 파티션에 새 데이터가 추가되지 않은 경우 해당 파티션의 다른 Z 순서는 아무런 영향을 미치지 않습니다.⁵

액체 클러스터링

액체 클러스터링은 현재 이 글을 쓰는 시점에 미리 보기로 제공되는 Delta Lake의 새로운 기능입니다. 이 기능은 가까운 시일 내에 일반 안정화 버전으로 제공될 예정입니다. [Delta Lake 문서 웹 사이트](#)를 검토하면 액체 클러스터링 상태에 대해 자세히 알아보고 최신 상태를 유지할 수 있습니다. 그리고 이 [기능 요청](#).

이 장 전체에서 언급된 성능 조정 기술 중 일부는 데이터 레이아웃을 최적화하여 읽기 및 쓰기 성능을 향상시키는 것을 목표로 하지만 몇 가지 단점이 있습니다.

파티셔닝 파티

션은 데이터가 여러 개의 작은 파일에 저장되어 필연적으로 성능이 저하되는 작은 파일 문제를 일으킬 위험이 있습니다.

그리고 테이블이 분할되면 이 분할은 변경할 수 없으며 새로운 사용 사례 또는 새로운 쿼리 패턴에 대한 문제가 발생할 수 있습니다. Delta Lake는 분할을 지원하지만 분할은 고정된 데이터 레이아웃으로 간주되므로 파티션 발전에 문제가 있습니다.

ZORDER BY

테이블에 데이터가 삽입, 업데이트 또는 삭제될 때마다 최적화를 위해 OPTIMIZE ZORDER BY를 다시 실행해야 합니다. 그리고 다시 ZORDER BY를 적용할 때 사용자는 표현식에 사용된 열을 기억해야 합니다. 이는 ZORDER BY 에 사용된 열이 지속되지 않아 다시 적용하려고 시도할 때 오류나 문제가 발생할 수 있기 때문입니다. OPTIMIZE ZORDER BY는 멱등성이 아니므로 실행 시 데이터가 다시 클러스터링됩니다.

분할 및 Z 순서 지정의 많은 단점은 Delta Lake의 액체 클러스터링 기능을 통해 해결될 수 있습니다. 델타 테이블에 대한 다음 시나리오는 유동 클러스터링을 통해 큰 이점을 얻습니다.

- 테이블은 카디널리티가 높은 열로 필터링되는 경우가 많습니다.
- 데이터 분포에 상당한 편향이 있는 테이블
- 많은 양의 튜닝 및 유지 관리가 필요한 테이블

⁵ Delta Lake 설명서 의 "최적화"를 참조하세요 .

- 동시 쓰기 요구 사항이 있는 테이블 • 시간에 따라
변하는 파티션 패턴이 있는 테이블

Delta Lake의 유동 클러스터링 기능은 파티셔닝 및 Z 순서 지정에서 발견된 제한 사항을 해결하고 보다 동적인 데이터 레이아웃을 통해 읽기 및 쓰기 성능을 모두 개선하는 것을 목표로 합니다. 궁극적으로 유동 클러스터링은 효율적인 쿼리 액세스를 지원하는 동시에 성능 조정 오버헤드를 줄이는 데 도움이 됩니다.

유동 클러스터링 활성화 테이블에서 유

동 클러스터링을 활성화하려면 테이블을 생성할 때 CLUSTER BY 명령을 지정할 수 있습니다. 테이블을 생성할 때 CLUSTER BY 명령을 사용하여 유동 클러스터링을 지정해야 합니다. 유동 클러스터링이 활성화되지 않은 기존 테이블(예: ALTER TABLE 사용)에는 클러스터링을 추가할 수 없습니다.



Databricks를 사용하여 이 책을 따라하고 GitHub 리포지토리에
서 노트북을 실행하는 경우 액체 클러스터링과 관련된 코드를 실행하
려면 Databricks Runtime 13.2 이상이 필요합니다.

Liquid Clustering이 활성화된 테이블을 생성하는 방법을 보여주기 위해 노트북 "04 - Liquid Clustering"과 다음 명령을 사용할 수 있습니다.

```
%sql
CREATE EXTERNAL TABLE Taxdb.tripDataClustered CLUSTER BY(VendorId)
  위치 '/mnt/datalake/book/chapter05/YellowTaxisLiquidClusteringDelta'
  AS SELECT * FROM TaxDb.tripData LIMIT 1000;
```

앞의 명령은 유동 클러스터링이 활성화되고 VendorId로 클러스터링되며 이전에 생성된 TaxiDb.tripData 테이블의 데이터로 채워진 외부 테이블을 생성합니다.

클러스터링을 트리거하려면 새로 생성된 테이블에서 OPTIMIZE 명령을 실행합니다.

```
%sql
OPTIMIZE Taxdb.tripDataClustered;
```

출력(관련 부분만 표시됨):

```
+-----+
| 측정항목
+-----+
| {"sizeOfTableInBytesBeforeLazyClustering": 43427, "isNewMetadataCreated": || true... "numFilesClassifiedToLeafNodes": 1, || "sizeOfFilesClassifiedToLeafNodesInBytes": 43427, || "logicalSizeOfFilesClassifiedToLeafNodesInBytes": 43427, || "numClusteringTasksPlanned": 0, "numCompactionTasksPlanned": 0, || "numOptimizeBatchesPlanned": 0, "numLeafNodesExpanded": 0, |
```

```

| "numLeafNodesClustered": 0, "numLeafNodesCompacted": 0, |
| "numIntermediateNodesCompacted": 0, "totalSizeOfDataToCompactInBytes": 0, |||
"totalLogicalSizeOfDataToCompactInBytes": 0, || "numIntermediateNodesClustered": 0, "numFilesSkippedAfterExpansion": 0, |
| "totalSizeOfFilesSkippedAfterExpansionInBytes": 0, |
| "totalLogicalSizeOfFilesSkippedAfterExpansionInBytes": 0, |
| "totalSizeOfDataToRewriteInBytes": 0, || "totalLogicalSizeOfDataToRewriteInBytes": 0… |
+-----+

```

이 장의 앞부분에서는 OPTIMIZE를 실행한 후 표시되는 측정항목을 확인했습니다.
 테이블에 명령을 내립니다. 액체가 있는 테이블에 대한 OPTIMIZE 명령의 출력
 클러스터링이 활성화되면 이제 ClusterMetrics가 측정항목에 포함된 것을 볼 수 있습니다.
 출력. 이러한 ClusterMetrics는 기본 클러스터에 대한 자세한 정보를 표시합니다.
 데이터 파일(예: 크기 및 수), 압축 세부 정보 및 클러스터 노드 정보
 클러스터링 결과를 볼 수 있습니다.

쓰기 시 데이터를 자동으로 클러스터링하는 작업은 몇 가지 작업뿐이라는 점에 유의하는 것이 중요합니다.
 액체 클러스터링을 사용하여 테이블에 데이터를 쓸 때. 다음 작업이 지원됩니다.
 삽입되는 데이터의 크기에 따라 쓰기 시 자동으로 데이터를 클러스터링합니다.
 512GB를 초과하지 않습니다:

- 예 집어 넣다
- CREATE TABLE AS SELECT (CTAS) 문
- 복사 대상
- Spark.write.format("delta").mode("append") 와 같은 쓰기 추가

이러한 특정 작업만 쓰기 시 데이터 클러스터링을 지원하므로 다음을 수행해야 합니다.
 OPTIMIZE를 실행하여 정기적으로 클러스터링을 트리거합니다. 이 명령 실행
 데이터가 적절하게 클러스터링되었는지 확인하는 경우가 많습니다.

다음에 의해 트리거될 때 액체 클러스터링이 증분된다는 점도 주목할 가치가 있습니다.
 OPTIMIZE, 즉 데이터를 수용하기 위해 필요한 데이터만 다시 작성됨을 의미합니다.
 클러스터링해야 하는 것입니다. 모든 쓰기 작업이 자동으로 데이터를 클러스터링하는 것은 아니기 때문에
 OPTIMIZE는 증분 작업 이므로 정기적으로 수행하는 것이 좋습니다.
 특히 이 증분 프로세스가 도움이 되므로 클러스터 데이터에 대한 OPTIMIZE 작업을 예약합니다.
 이러한 작업은 빠르게 실행됩니다.

클러스터된 열에 대한 작업

유동 클러스터링을 활성화하여 테이블에 어떤 열을 지정하는 방법을 배웠습니다.
 CLUSTER BY 명령을 사용하여 클러스터링됩니다. 테이블이 클러스터링되면
 특정 열의 경우 클러스터링된 컬럼을 활용하여 데이터를 보다 효율적으로 읽을 수 있습니다.
 열을 보고, 변경하고, 삭제할 수도 있습니다.

클러스터링된 열 변경

테이블을 처음 생성할 때 테이블이 클러스터링되는 방식을 지정해야 하지만 다음을 수행할 수 있습니다.
 ALTER TABLE을 사용하여 테이블에서 클러스터링에 사용되는 열을 변경하고
 클러스터별로. 앞서 생성한 테이블의 클러스터 열을 다음과 같이 변경하려면
 VendorId 및 RateCodeId 모두에 클러스터링된 경우 다음 명령을 실행합니다.

```
%sql
ALTER TABLE Taxdb.tripDataClustered CLUSTER BY (VendorId, RateCodeId);
```



클러스터링 키로 최대 4개의 열을 지정할 수 있습니다.

클러스터링된 열을 변경할 때 유동적 클러스터링에는 전체 테이블이 필요하지 않습니다.
 다시 작성됩니다. 이러한 클러스터링 발전은 동적 데이터 레이아웃 기능으로 인해 발생합니다.
 액체 클러스터링은 언급된 파티션 기능에 비해 상당한 이점을 제공합니다.
 장 일부분에서. 전통적인 파티셔닝은 고정된 데이터 레이아웃이며 그렇지 않습니다.
 전체를 다시 작성할 필요 없이 테이블 분할 방식 변경 지원
 테이블. 테이블에 대한 쿼리 패턴이 종종 발생할 수 있으므로 이러한 클러스터링 발전은 필수적일 수 있습니다.
 시간이 지남에 따라 변경되며 이를 통해 새로운 쿼리 패턴에 동적으로 적응할 수 있습니다.
 상당한 오버헤드나 어려움 없이 말이죠.

클러스터링된 열 보기

이제 테이블이 클러스터링되는 방식을 변경했으므로 테이블 메타데이터를 볼 수 있습니다.
 DESCRIBE TABLE을 사용하여 이러한 변경 사항을 확인하고 클러스터링된 열을 확인합니다.

```
%sql
테이블 설명 Taxdb.tripDataClustered;
```

출력(관련 부분만 표시됨):

열_이름	데이터_유형	댓글
# 클러스터링 정보		
열_이름	데이터_유형	댓글
공급업체 ID	큰	없는
요율 코드 ID	더블	없는

DESCRIBE TABLE 명령은 테이블에 대한 기본 메타데이터 정보를 반환합니다. 이는 클러스터 정보와 테이블이 이제 두 위치 모두에서 클러스터링되었음을 보여줍니다.

VendorId 및 RateCodeId.

클러스터링된 테이블에서 데이터 읽기 이제

클러스터링된 열을 확인했으므로 쿼리 필터(예: WHERE 절)에서 클러스터링된 열을 지정하여 최상의(즉, 가장 빠른) 쿼리 결과를 얻을 수 있습니다. 예를 들어, 최상의 쿼리 결과를 얻으려면 Taxdb.tripDataClustered 테이블의 WHERE 절에 VendorId 및 RateCodeId를 추가합니다.

```
%sql
SELECT * FROM Taxdb.tripDataClustered WHERE VendorId = 1 및 RateCodeId = 1
```

클러스터링된 열 제거

테이블이 클러스터링된 열을 제거하기로 선택한 경우 간단히 지정할 수 있습니다.

클러스터 없음:

```
%sql
ALTER TABLE Taxdb.tripDataClustered CLUSTER BY NONE;
```

액체 클러스터링 경고 및 고려 사항

현재 글을 쓰는 시점에 액체 클러스터링이 미리 보기 상태라는 점을 고려하면 액체 클러스터링을 활성화하고 활용하기 전에 고려해야 할 몇 가지 요소가 있습니다.

- 환경 런타임을 확인하여 델타 테이블에서 OPTIMIZE를 지원하는지 확인하세요.
액체 클러스터링이 활성화되었습니다.

Databricks를 사용하여 이 책을 따르고 GitHub 리포지토리에서 노트북을 실행하는 경우 Databricks Runtime 13.2 이상이 필요합니다. • 유동 클러스터링을 활성화하여 생성된 테이블에는 생성

시 다양한 델타 테이블 기능이 활성화되어 있으며 델타 버전 7 및 리더 버전 3을 사용합니다. 테이블 프로토콜 버전은 다운그레이드할 수 없으며 클러스터링이 활성화된 테이블은 지원하지 않는 Delta Lake 클라이언트에서 읽을 수 없습니다. 모든 활성화된 델타 리더 프로토콜 테이블 가능.

- 테이블을 처음 생성할 때 Delta Lake 액체 클러스터링을 활성화해야 합니다. 테이블이 처음 생성될 때 클러스터링을 활성화하지 않으면 클러스터링을 추가하기 위해 기존 테이블을 변경할 수 없습니다.
- 클러스터링된 열에 대해 수집된 통계가 있는 열만 지정할 수 있습니다.
기본적으로 델타 테이블의 처음 32개 열에만 통계가 수집된다는 점을 기억하세요.

- 구조적 스트리밍 워크로드는 쓰기 시 클러스터링을 지원하지 않습니다. • OPTIMIZE 를 자주 실행하여 새 데이터가 클러스터링되었는지 확인합니다.



Liquid 클러스터링은 파티셔닝 또는 ZORDER BY 와 호환되지 않습니다 .

결론

이 장에서는 물리적, 동적으로 데이터를 저장하고 구성하는 다양한 기술과 이 기술이 작업 중에 데이터를 읽고 검색하는 방법에 미칠 수 있는 중요한 영향에 대해 배웠습니다. 캡처된 데이터 포인트의 유형이 데이터의 양과 함께 계속해서 증가함에 따라 테이블도 점점 더 커질 것입니다. 대규모 데이터 세트에 대한 성능 조정은 좋은 전략이자 모범 사례로 간주되어 왔으며 앞으로도 그럴 것입니다. 이를 가능하게 하는 Delta Lake 기능을 이해하면 오버헤드를 크게 줄이는 데 도움이 됩니다.

우리는 작은 데이터 파일 문제, 그것이 성능에 미칠 수 있는 영향, 그리고 OPTIMIZE를 사용한 최적의 파일 통합을 포함하여 압축 전략을 사용하여 문제를 해결할 수 있는 방법에 대해 논의했습니다. 테이블 파일을 최적화한 후 ZORDER BY를 사용하여 해당 파일 내의 값을 정렬할 수 있습니다. 이를 통해 데이터 건너뛰기 통계를 통해 데이터 건너뛰기를 보다 효과적으로 활용할 수 있습니다. 델타 테이블을 분할하고 데이터를 고유한 부분으로 나누어 읽어야 하는 데이터 양을 더욱 줄임으로써 데이터 건너뛰기를 한 단계 더 발전시킬 수 있습니다.

그런 다음 분할 및 Z 순서 지정이 여전히 가질 수 있는 몇 가지 문제를 해결하는 새로운 Delta Lake 기능을 살펴보았습니다. Liquid 클러스터링은 시간이 지남에 따라 쿼리 패턴이 발전함에 따라 전체 테이블을 다시 작성할 필요가 없는 동적 데이터 레이아웃을 통해 클러스터링 발전을 제공합니다. 이렇게 크게 자동화된 기능은 파티셔닝 및 Z 순서 지정과 호환되지 않지만 다른 성능 최적화 기능에 비해 튜닝 작업이 덜 필요하며 테이블의 읽기 및 쓰기 성능을 크게 향상시킵니다.

이 장에 언급된 Delta Lake 기능을 사용하면 읽어야 하는 관련 없는 데이터의 양을 줄이고 특히 델타 테이블의 데이터 파일 수가 증가함에 따라 성능을 향상할 수 있습니다. 다음 장에서는 Delta Lake가 오래된 데이터 파일을 활용하여 데이터 버전을 지정하고 특정 시점으로 돌아갈 수 있는 방법을 알아봅니다.

제6장

시간여행 이용하기

이전에 데이터베이스와 테이블을 다루면서 WHERE 절을 잊어버리고 실수로 전체 테이블에 대해 DELETE 또는 UPDATE 문을 실행했을 때 즉시 패닉 상태에 빠졌을 가능성이 높습니다. 우리는 모두 거기에 있었습니다. 또는 감사, 추적 또는 분석 목적으로 특정 시점에 데이터나 스키마가 어떻게 생겼는지 궁금할 수도 있습니다.

데이터가 끊임없이 변화하는 방식을 고려할 때 다음 시나리오는 역사적으로 해결하거나 대답하기 어려웠던 일반적인 발생입니다.

규제 감사 및

규제 준수를 위해서는 데이터를 수년 동안 저장하고 검색해야 하거나 데이터에 대한 특정 변경 사항(예: GDPR)을 추적해야 할 수 있습니다.

실험 및 보고서 재현 데이터 과학자나 분석

가가 특정 시점의 특정 데이터 세트를 바탕으로 보고서나 기계 학습 실험 및 모델 출력을 다시 생성해야 하는 경우가 많습니다.

롤백 INSERT,

UPDATE, DELETE, MERGE 등 데이터에 대한 우발적이거나 잘못된 DML 작업을 수정하고 이전 상태로 롤백해야 할 수 있습니다.

시계열 분석 보고 요구

사항에 따라 시간이 지남에 따라 데이터를 되돌아보거나 분석해야 할 수 있습니다(예: 한 달 동안 추가된 신규 고객 수).

디버깅 ETL

파이프라인, 데이터 품질 문제 또는 특정 원인을 기록 상태에서만 관찰할 수 있는 손상된 프로세스 문제를 해결합니다.

특정 지점에서 다양한 버전의 데이터를 쉽게 탐색할 수 있는 기능
 시간 여행은 Delta Lake 시간 여행이라는 Delta Lake의 주요 기능입니다. 당신이 배운
2장 에서는 트랜잭션 로그가 자동으로 델타 테이블의 데이터 버전을 저장합니다.
 이 버전 관리를 통해 해당 데이터의 기록 버전에 액세스할 수 있습니다. 을 통해
 버전 관리 및 데이터 보존을 통해 이러한 강력한 Delta Lake를 사용하는 방법을 배우게 됩니다.
 기능을 활용하는 동시에 데이터 관리 및 스토리지 최적화도 활용합니다.

델타 레이크 시간 여행

Delta Lake 시간 여행을 통해 이전 버전의 데이터에 액세스하고 되돌릴 수 있습니다.
 Delta Lake에 저장되어 버전 제어를 위한 강력한 메커니즘을 쉽게 제공합니다.
 감사 및 데이터 관리. 그런 다음 시간 경과에 따른 변경 사항을 추적하고 롤백할 수 있습니다.
 필요한 경우 이전 버전으로.

예제를 살펴보겠습니다. 먼저 “챕터 초기화” 노트북을 실행합니다.
61 장 에서는 Taxdb.tripData Delta 테이블을 생성합니다 . 다음으로 "01 - Time"을 엽니다.
 Travel' 6 장의 노트북입니다. VendorId를 1에서 업데이트해야 한다고 가정해 보겠습니다 .
 10으로. 그런 다음 VendorId = 2인 모든 항목을 삭제해야 합니다 . 스크립트 사용
 “01 - Time Travel” 노트에서 다음 명령을 실행하여 해당 내용을 적용합니다.
 변경사항:

```
%sql
-- UPDATE Taxdb.tripData 테이블의
-- 레코드 업데이트
SET 공급업체 ID = 10
VendorId = 1;

--taxdb.tripData 테이블 DELETE FROM에서
--레코드 삭제
VendorId = 2인 경우;

--테이블 기록 설명
역사 설명 Taxdb.tripData;
```

다음 출력이 표시됩니다(관련 부분만 표시됨).

버전	작업	설명
2	삭제	{"슬러": "[{"spark_catalog.taxidb.tripData.VendorId = 2L}]"}
1	업데이트	{"predicate": "(VendorId#5081L = 1)"}
0	쓰다	{"모드": "덮어쓰기", "partitionBy": "[]")}

1 GitHub 저장소 위치: /chapter06/00 - 장 초기화

OperationMetrics를 표시하도록 출력이 계속되고 수정되었습니다 (관련 부분만 표시됨):

+-----+-----+		
버전 운영 작업메트릭스		
+-----+-----+		
2	삭제	{"numRemovedFiles": "10", "numCopiedRows": "9643805",
		"numAddedChangeFiles": "0",
		..."numDeletedRows": "23360027"..., "numAddedFiles": "10"...)
+-----+-----+		
1	업데이트	{"numRemovedFiles": "10", "numCopiedRows": "23414817",
		"numAddedChangeFiles": "0", ..."numAddedFiles": "10".}
+-----+-----+		
0	쓰다	{"numFiles": "10", "numOutputRows": "33003832" ...}
+-----+-----+		

출력을 보면 총 세 가지 버전의 테이블이 있음을 알 수 있습니다.

각 커밋마다 하나씩, 버전 2가 가장 최근 변경 사항입니다.

- 버전 0: 덮어쓰기를 사용하고 파티션 없이 초기 델타 테이블을 작성했습니다.
- 버전 1: 조건자 VendorId = 1을 사용하여 델타 테이블을 업데이트했습니다.
- 버전 2: VendorId = 2 조건자를 사용하여 Delta 테이블에서 데이터를 삭제했습니다.

DESCRIBE HISTORY 명령은 버전에 대한 세부 정보를 표시합니다.

트랜잭션 타임스탬프, 작업, 작업 매개변수 및 작업 지표. 그만큼

출력의 OperationMetrics에는 변경된 행 및 파일 수도 표시됩니다.

수술부터.

그림 6-1은 테이블의 다양한 버전과 기본 버전의 예를 보여줍니다.

데이터를 보내고 있습니다.

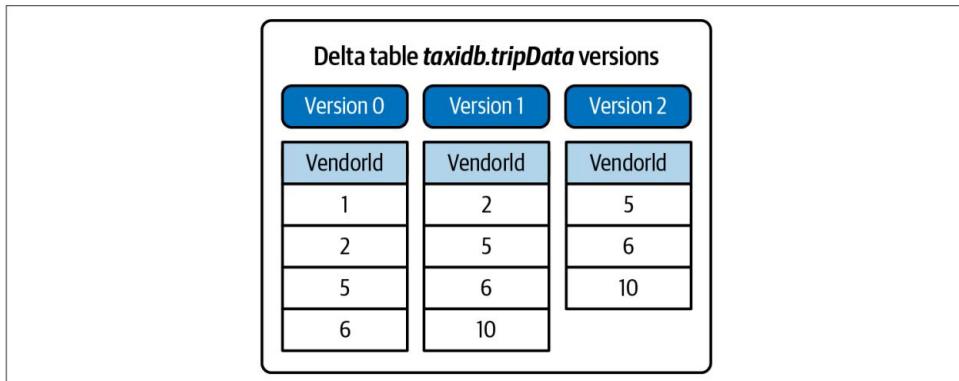


그림 6-1. Taxdb.tripData 버전 기록

테이블 복원

이제 이전 UPDATE 및 DELETE 작업을 룰백하고 싶다고 가정해 보겠습니다.
Taxdb.tripData에서 수행하고 이를 원래 상태(예: 버전)로 다시 복원합니다.
0). RESTORE 명령을 사용하여 테이블을 원하는 버전으로 룰백할 수 있습니다.

```
%sql
--테이블을 이전 버전으로 복원
테이블 Taxdb.tripData를 0 버전으로 복원합니다.

--테이블 기록 설명
역사 설명 Taxdb.tripData;
```

출력(관련 부분만 표시됨):

버전	운영	작업내개변수
삼	복원	{"버전": "0", "타임스탬프": null}
2	삭제	{"슬어": [{"\\"(spark_catalog.taxidb.tripData.VendorId IN 5,6L \")"}]}
1	업데이트	{"predicate": "(VendorId#5081L = 1)"}
0	쓰다	{"모드": "덮어쓰기", "partitionBy": "[]”}

테이블을 버전 0으로 복원하고 DESCRIBE HISTORY 명령을 실행한 후,
이제 테이블의 추가 버전인 버전 3이 있음을 알 수 있습니다.

RESTORE 작업을 캡처합니다. 그림 6-2에서는 다양한 버전의
기본 데이터의 테이블과 예.

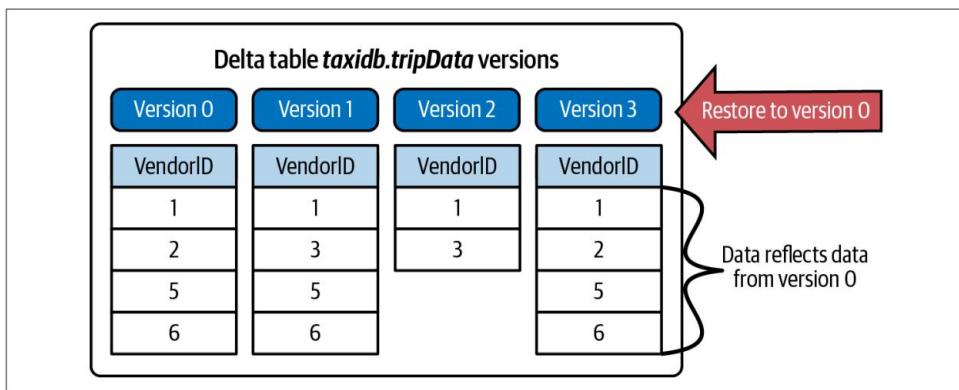


그림 6-2. 버전 0으로 다시 복원한 후의 Taxdb.tripData 버전

이제 데이터와 [그림 6-2](#)에서 테이블의 최신 버전이 버전 0의 데이터를 반영하는 것을 볼 수 있습니다.



이미 복원된 테이블을 복원할 수 있습니다. 이전에 복원된 버전으로 테이블을 복원 할 수도 있습니다.

데이터 파일이 수동으로 또는 VACUUM (이 정의 뒷부분에서 VACUUM에 대해 자세히 알아보기)을 통해 삭제되면 해당 데이터 파일을 참조하는 버전으로 테이블을 복원하는 데 실패합니다. 사용

Spark 구성 `Spark.sql.files.ignoreMissingFiles = True`는 테이블을 부분적으로 복원합니다. 이름에서 알 수 있듯이 이 Spark 구성은 데이터를 읽을 때 누락된 파일을 무시합니다.

타임스탬프를 통해 복원 이전 예에서는 테

이들을 특정 버전으로 복원했지만 테이블을 특정 타임스탬프로 복원할 수도 있습니다. 이전 상태로 복원하기 위한 타임스탬프 형식은 `yyyy-MM-dd HH:mm:ss`입니다. 날짜 (`yyyy-MM-dd`) 문자열만 제공하는 것도 지원됩니다.

```
%sql
--특정 타임스탬프로 테이블 복원 RESTORE TABLE
Taxdb.tripData TO TIMESTAMP AS OF '2023-01-01 00:00:00';

-- 테이블을 1시간 전 상태로 복원 RESTORE TABLE Taxdb.tripData TO
TIMESTAMP AS OF current_timestamp()
- INTERVAL '1' HOUR;
```

또한 테이블 복원을 위해 PySpark 및 DataFrame API를 사용하기 위해 델타 모듈을 가져올 수도 있습니다. 이전에 했던 것처럼 특정 버전으로 복원하기 위해 `RestoreToVersion(version: int)` 메소드를 사용할 수도 있고, 지정된 타임스탬프로 복원하기 위해 `RestoreToTimestamp(timestamp: str)` 메소드를 사용할 수도 있습니다:

```
--delta.tables에서 델타 모듈 가
져오기 import *

--PySpark를 사용하여 특정 타임스탬프로 테이블을 복원합니다. deltaTable =
DeltaTable.forName(spark, "taxidb.tripData")
deltaTable.restoreToTimestamp("2023-01-01")
```

내부 시간 여행

트랜잭션 로그는 테이블에서 작업을 수행할 때 읽어야 할 파일과 읽어서는 안 되는 파일을 추적하므로 버전 기록을 델타 테이블에 보관할 수 있습니다. DESCRIBE HISTORY 명령이 실행 되면 작업 중에 추가 및 제거된 파일 수를 알려주는 OperationMetrics도 반환됩니다. 테이블에서 UPDATE, DELETE 또는 MERGE를 수행할 때 해당 데이터는 기본 스토리지에서 물리적으로 제거되지 않습니다. 오히려 이러한 작업은

읽어야 할 파일과 읽지 말아야 할 파일을 나타내는 트랜잭션 로그입니다. 마찬가지로, 테이블을 이전 버전으로 복원할 때 물리적으로 데이터를 추가하거나 제거하지 않습니다. 읽을 파일을 알려주기 위해 트랜잭션 로그의 메타데이터만 업데이트합니다.

2장에서는 _delta_log 디렉터리 내의 JSON 파일과 체크포인트 파일에 대해 배웠습니다. 체크포인트 파일은 특정 시점의 전체 테이블 상태를 저장하며, JSON 커밋을 Parquet 파일에 결합하여 읽기 성능을 유지하기 위해 자동으로 생성됩니다. 그런 다음 체크포인트 파일과 후속 커밋을 읽어 테이블의 현재 상태와 시간 이동의 경우 이전 상태를 가져올 수 있으므로 모든 커밋을 나열하고 재처리할 필요가 없습니다.

트랜잭션 로그는 검사점 파일을 커밋하며, 데이터 파일이 물리적으로 제거되는 것이 아니라 논리적으로만 제거된다는 사실은 Delta Lake가 델타 테이블에서 시간 여행을 쉽게 활성화하는 방법의 기초입니다. **그림 6-3은** 다양한 트랜잭션 및 버전 전반에 걸쳐 Taxdb.tripData 테이블의 각 작업에 대한 트랜잭션 로그 항목을 보여줍니다.

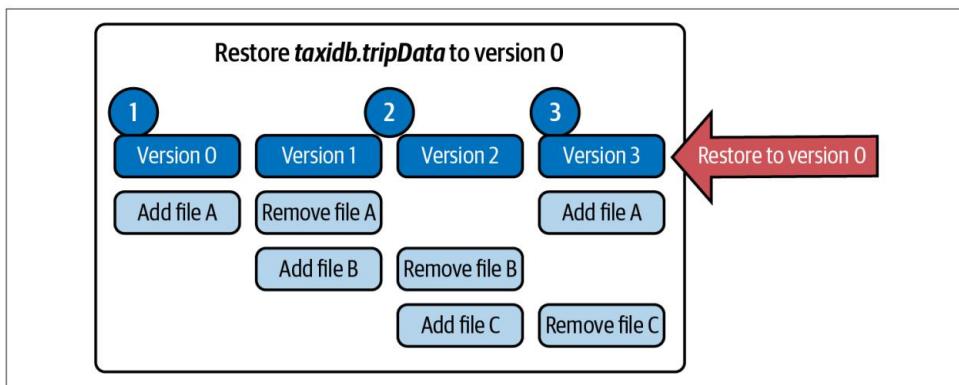


그림 6-3. Taxdb.tripData 트랜잭션 로그 파일

그림 6-3의 번호가 매겨진 단계는 다음을 보여줍니다.

- 버전 0: 초기 테이블이 생성되고 트랜잭션 로그에 파일이 추가됩니다.
- 버전 1 및 2: 파일 추가 및 파일 제거는 Delta Lake가 각 버전에 대해 읽어야 할 파일을 결정하는 데 사용하는 메타데이터 항목 역할을 합니다. 파일 제거는 데이터를 물리적으로 삭제하지 않습니다. 단지 논리적으로 테이블에서 파일을 제거할 뿐입니다.
- 버전 3: 이 버전은 버전 0으로 다시 복원되었으므로 트랜잭션 로그는 버전 0에 추가된 원본 파일인 A 파일을 복원하고 C 파일을 논리적으로 제거합니다.

이전에 버전 0으로 복원한 Taxdb.tripData 테이블의 테이블 기록을 살펴보면 OperationMetrics에서 캡처된 복원, 추가 및 제거된 파일 수를 확인할 수 있습니다.

```
%sql
--테이블 기록 설명
역사 설명 Taxdb.tripData
```

출력(관련 부분만 표시됨):

버전 운영 작업메트릭스		
3 복원 {"numRestoredFiles": "10"..."numRemovedFiles":		
	"10"..."복구 후 파일 수": "10"	
2 삭제 {"numRemovedFiles": "10"..."numAddedChangeFiles":		
	"0"..."numAddedFiles": "10"	
1 업데이트 {"numRemovedFiles": "10"..."numAddedChangeFiles":		
	"0"..."numAddedFiles": "10"	
0 쓰기 {"numFiles": "10", "numOutputRows": "33003832",		
	"numOutputBytes": "715810450"}	

나중에 이전 버전의 데이터를 유지하고 제거하는 방법에 대해 자세히 알아봅니다.
이 장의 섹션.

RESTORE 고려 사항 및 경고

RESTORE는 데이터 변경 작업, 즉 데이터를 의미한다는 점에 유의하는 것이 중요합니다.

변경 = 사실입니다. 이는 잠재적으로 Struc과 같은 다운스트림 작업에 영향을 미칠 수 있음을 의미합니다.

8 장에서 자세히 알아보게 됩니다.

스트리밍 쿼리가 델타에 대한 업데이트만 처리하는 상황을 생각해 보세요.

테이블. 테이블을 이전 버전으로 복원 하면 테이블에 대한 이전 업데이트가 수행됩니다.

트랜잭션 로그가 복원되었으므로 스트리밍 작업에서 다시 처리될 수 있습니다.

dataChange = true 인 파일 추가 작업을 사용하여 이전 버전의 데이터 . 그만큼

스트리밍 작업은 해당 레코드를 새로운 데이터로 인식합니다.

표 6-1. RESTORE 로 인한 작업

테이블 버전	작전 델타 로그 업데이트	데이터 변경 로그 업데이트 기록
0	INSERT AddFile(/path/to/file-1, 데이터변경 = 참)	(VendorId = 1, 승객_수 = 2, (VendorId = 2, 승객_수 = 3)
1	INSERT AddFile(/path/to/file-2, 데이터변경 = 참)	(VendorId = 2, 승객_수 = 4)
2	최저화 AddFile(/path/to/file-3, dataChange = false), 제거 파일(/path/to/file-1), 제거 파일(/경로/대상/파일-2)	(OPTIMIZE 중에는 기록이 변경되지 않습니다.)

테이 블 버전	작전 델타 로그 업데이트	데이터 변경 로그 업데이트 기록
상	복원 제거파일(/path/to/file-3), AddFile(/path/to/file-1, dataChange = true), 추가 파일(/경로/to/file-2, 데이터 변경 = 사실)	(VendorId = 1, 승객_수 = 2, (VendorId = 2, 승객_수 = 3), (VendorId = 2, 승객_수 = 4)

표 6-1에서 OPTIMIZE 작업은 버전 1 및 2와 관련된 파일을 제거하고 버전 3에 대한 파일을 추가했음을 알 수 있습니다. RESTORE 명령을 실행한 후 작업은 해당 버전에 대한 파일 1과 파일 2를 다시 추적했습니다. dataChange 작업으로 간주됩니다.

이전 버전의 테이블 쿼리 기본적으로 델타 테이블을 쿼

리할 때마다 항상 테이블의 최신 버전을 쿼리하게 됩니다. 그러나 Delta Lake 시간 이동을 사용하면 테이블을 복원할 필요 없이 테이블의 이전 버전에 대한 읽기 작업을 수행할 수도 있습니다. 데이터 자체는 기본 스토리지에서 물리적으로 삭제되지 않고 논리적으로만 제거된다는 점을 기억하세요.

물리적 삭제가 아닌 논리적 제거는 시간 이동을 통해 테이블을 특정 시점으로 복원할 수 있을 뿐만 아니라 복원하지 않고도 이전 버전의 테이블을 직접 쉽게 쿼리할 수 있음을 의미합니다.

다음 두 가지 방법으로 이전 버전의 데이터에 액세스할 수 있습니다.

1. 타임스탬프 사용 2. 버

전 번호 사용

버전 번호를 이용하여 테이블을 이전 버전으로 복원한 것과 유사하게, 버전 번호를 이용하여 특정 시점의 테이블을 쿼리할 수도 있습니다.

이전 예에서는 테이블을 이전 상태로 다시 복원했지만 WHERE VendorId = 2 조건자를 사용하여 버전 2에서 레코드를 삭제했습니다. 시간 여행을 사용하여 테이블 버전 2에서 해당 VendorId 레코드 수를 검색할 수 있습니다.:

```
%sql
--count 버전 번호를 사용하여 VendorId = 1인 레코드 SELECT COUNT(*) AS count
FROM Taxdb.tripData VERSION AS OF 2 WHERE VendorId = 1;

--count 작업 타임스탬프를 사용하여 VendorId = 1인 레코드 SELECT COUNT(*) AS count
FROM Taxdb.tripData VERSION AS OF '2023-01-01 00:00:00'
WHERE VendorId = 1;

--count 작업 타임스탬프 및 @ 구문을 사용하여 VendorId = 1인 레코드 --timestamp는 yyyyMMddHHmmssSSS 형식이어야 합니다. SELECT COUNT(*) AS count FROM
Taxidb.YellowTaxi@202301010000000000 WHERE VendorId = 1;
```

```
--count 버전 번호와 @ 구문을 사용하여 VendorId = 1인 레코드 SELECT COUNT(*) AS count FROM Taxib.tripData@v2 WHERE
VendorId = 1;
```

산출:

카운트	
+-----+	
0	
+-----+	

이전 예제에서 볼 수 있듯이 다양한 유형의 구문을 사용하여 다양한 버전의 데이터에 액세스할 수 있습니다. VERSION AS OF를 지정하거나 테이블 이름 뒤에 "@" 을 추가하는 구문으로 타임스탬프나 버전 번호를 사용할 수 있습니다.

시간 여행은 SQL을 통해 액세스할 수 있을 뿐만 아니라 .option() 메서드를 사용하여 DataFrame API를 통해 시간 여행을 할 수도 있습니다.

```
# 버전 번호를 사용하여 VendorId = 1인 레코드 수를 계산합니다. Spark.read.option("versionAsOf",
"0").table("taxidb.tripData").filter( "VendorId = 1" ).count()
```

```
# 타임스탬프를 사용하여 VendorId = 1인 레코드 수를 계산합니다.
Spark.read.option("timestampAsOf", "0").table("taxidb.tripData").filter( "VendorId = 1" ).count()
```

타임스탬프를 기준으로 쿼리하면 동일한 테이블의 데이터를 서로 다른 두 시점의 데이터 자체와 비교할 수 있으므로 시계열 분석을 쉽게 수행할 수 있습니다. 기록 데이터를 캡처하고 시계열 분석(예: 천천히 변화하는 차원 및 변경 데이터 피드)을 활성화하기 위해 따를 수 있는 다른 ETL 패턴이 있지만, 시간 이동은 그렇지 않은 테이블에 대해 임시 분석을 수행하는 빠르고 쉬운 방법을 제공합니다. 이러한 ETL 패턴을 마련해야 합니다. 예를 들어, Taxidb.tripData 의 버전 기록을 사용하여 지난 주와 비교하여 이번 주에 픽업된 승객 수를 빠르게 확인하려면 다음 쿼리를 실행할 수 있습니다.

```
%sql
--7일 전 신규 승객 수 SELECT sum(passenger_count) - ( SELECT
sum(passenger_count)

Taxdb.tripData TIMESTAMP AS OF date_sub(current_date(), 7)
)
Taxdb.tripData에서
```



설명된 것처럼 시간 여행을 통해 시계열 분석이 가능하지만 유사한 작업을 수행하는 더 효율적인 방법이 있습니다. 이러한 시계열 예제는 시간 여행의 기능을 보여주기 위한 목적으로 설명됩니다. 이 장의 뒷부분에서는 효율성으로 인해 시계열 분석을 수행하기 위한 권장 접근 방식을 지원하는 Delta Lake의 변경 데이터 피드에 대해 알아봅니다.

데이터 보존

델타 테이블을 지원하는 데이터 파일은 자동으로 삭제되지 않지만 검사점이 작성된 후 로그 파일은 자동으로 정리됩니다. 궁극적으로 특정 테이블 버전으로의 시간 이동을 가능하게 하는 것은 해당 테이블 버전에 대한 데이터와 로그 파일을 모두 보존하는 것입니다. 기본적으로 델타 테이블은 커밋 기록 또는 로그 파일을 30일 동안 유지합니다. 따라서 데이터나 로그 파일을 수정하지 않는 한 최대 30일 동안 델타 테이블의 시간 여행에 액세스할 수 있습니다.

이 섹션과 다음 섹션에서는 기간 보존 임계값을 확인할 수 있습니다. 보존 임계값은 파일이 저장소에서 물리적으로 제거되기 전에 파일을 보관해야 하는 간격(예: 일수)을 나타냅니다. 예를 들어 테이블의 보존 임계값이 7일인 경우 파일이 제거되기 전에 현재 테이블 버전보다 최소 7일 이전이어야 합니다. 다음 섹션에서는 이 책에서 설명하는 두 가지 유형의 보존, 즉 데이터 및 로그 파일 보존에 대해 설명합니다.

데이터 파일 보존

데이터 파일 보존은 데이터 파일이 델타 테이블에 보존되는 기간을 나타냅니다. 데이터 파일을 물리적으로 삭제하는 데 사용되는 명령인 VACUUM 으로 제거할 후보 파일의 기본 보존 기간은 7일입니다. 간단히 말해서 VACUUM 은 더 이상 Delta 테이블에서 참조되지 않고 보존 기간보다 오래된 데이터 파일을 제거합니다.

수동으로 제거하지 않는 한 데이터 파일은 VACUUM을 실행할 때만 제거됩니다. 이 명령은 델타 로그 파일을 삭제하지 않고 데이터 파일만 삭제합니다. 이 장의 뒷부분에서 VACUUM 명령과 그 작동 방식에 대해 자세히 알아볼 것입니다.

일반적으로 데이터 파일은 기본 보존 기간보다 오랫동안 보존되어야 합니다. 테이블 속성 delta.deletedFileRetentionDuration = "interval <interval>" 은 VACUUM 의 후보가 되기 전에 파일을 삭제해야 하는 기간을 제어합니다.

VACUUM을 실행하더라도 일정 기간 동안 데이터 파일을 보관하려면 테이블 속성 delta.deletedFileRetentionDuration = "interval <interval>" 을 사용하세요. 이는 후보가 되기 전에 파일을 삭제해야 하는 기간을 제어합니다.

진공. 예를 들어, 1년 동안 기록 데이터를 보관하고 액세스해야 하는 경우
delta.deletedFileRetentionDuration = "365일 간격".



데이터 파일을 제거하면 다음 장소로 시간 여행을 할 수 없습니다.
해당 데이터 파일을 사용한 테이블 버전.

그러나 초과 데이터 파일을 보관하면 시간이 지남에 따라 클라우드 스토리지 비용이 증가할 수 있습니다.
메타데이터 처리 성능에 대한 잠재적인 영향도 있습니다.

시간이 지남에 따라 데이터 파일이 어떻게 커질 수 있는지 보여주기 위해 DESCRIBE HISTORY를 사용합니다.

TaxDb.tripData 테이블에서 numFiles 및 numAddedFiles 측정항목을 사용할 수 있습니다.

각 작업 중에 추가된 파일 수를 표시하는 OperationMetrics :

```
%sql
--테이블 기록 설명
역사 설명 Taxdb.tripData
```

출력(관련 부분만 표시됨):

+-----+-----+		
버전	운영	작업메트릭스
3	복원	{"numRestoredFiles": "10"..."numRemovedFiles":
		"10"..."복구 후 파일 수": "10"
+-----+-----+		
2	삭제	{"numRemovedFiles": "10"..."numAddedChangeFiles":
		"0"..."numAddedFiles": "10"
+-----+-----+		
1	업데이트	{"numRemovedFiles": "10"..."numAddedChangeFiles":
		"0"..."numAddedFiles": "10"
+-----+-----+		
0	쓰기	{"numFiles": "10", "numOutputRows": "33003832",
		"numOutputBytes": "715810450"}
+-----+-----+		

numFiles 및 numAddedFiles 측정항목을 기반으로 30개의 파일이 있음을 확인할 수 있습니다.

이 테이블에 추가되었습니다. 매일 실행되고 수행되는 ETL 프로세스가 있는 경우

단일 테이블에 대한 INSERT, UPDATE 또는 DELETE를 수행한 후 1년 후에는

10,950(30 x 365) 파일! 이는 단일 테이블에 대한 파일 수일 뿐입니다. 상상하다

전체 데이터 플랫폼에 걸쳐 보유할 수 있는 파일 수. 개수

각 작업 중에 추가된 파일은 분명히 수행된 작업에 따라 달라집니다.

각 작업 중에 포함된 행 수 및 기타 변수가 있지만 이는 도움이 됩니다.

시간이 지남에 따라 데이터 파일이 어떻게 증가할 수 있는지 보여줍니다.

다행히 클라우드 데이터 레이크는 데이터 저장 측면에서 매우 비용 효율적입니다.

그러나 이러한 비용은 데이터 파일이 증가함에 따라 증가할 수 있습니다. 그렇기 때문에 여전히 중요한 것은

장기간 데이터 파일을 보관할 때 경제적이며, 보관 기간을 설정할 때 비용과 비즈니스 요구 사항을 고려하세요.

로그 파일 보존 로그 파일 보존

은 로그 파일이 델타 테이블에 보존되는 기간을 나타냅니다. 기본 보존 기간은 30일입니다. 테이블 속성 `delta.logRetentionDuration`을 사용하여 파일이 보존되는 기간을 변경할 수 있습니다. 예를 들어 테이블에 커밋 기록을 1년 동안 보관해야 하는 경우 `delta.logRetentionDuration = "interval 365일"`을 설정합니다.

2장 에서는 커밋 10개마다 체크포인트가 작성된다는 점을 배웠습니다(작성 당시에는 Delta Lake의 향후 버전에서 변경될 수 있음). Delta Lake는 새 검사점이 생성될 때마다 보존 간격에 따라 로그 파일을 자동으로 정리합니다.



특정 버전의 테이블로 시간 여행을 하기 위해서는 새로운 체크포인트가 기록될 때까지의 연속 로그 항목이 모두 필요합니다. 체크포인트는 10번의 커밋마다 기록됩니다. 즉, 테이블의 버전 0-9로 시간 이동을 하려면 모든 버전 0, 1, 2, ..., 9에 대한 로그 항목이 있어야 합니다. Delta가 보존 간격보다 오래된 로그 항목을 자동으로 정리하므로 버전 0에 대한 로그가 제거된 경우 테이블의 버전 0-9로 시간 여행을 할 수 있습니다.

로그 파일은 테이블의 읽기/쓰기 성능에 영향을 주지 않기 때문에 로그 파일을 유지하는 데에는 최소한의 단점이 있습니다. 테이블 기록을 활용하는 작업의 성능에만 영향을 미칩니다. 파일을 보관할 때는 항상 저장소 비용을 고려해야 하지만 로그 파일은 일반적으로 크기가 작습니다.



`delta.deletedFileRetentionDuration` 및 `delta.logRetentionDuration` 과 같은 Delta Lake 속성은 Spark 구성 속성을 사용하여 설정할 수도 있습니다.

파일 보존 기간 설정 예 Taxdb.tripData 테이블을 사용하는 경우 ,

예를 들어 시계열 분석이나 규제 목적으로 테이블의 전체 기록을 1년 동안 유지해야 한다는 요구 사항이 있다고 가정해 보겠습니다. 작년의 어느 시점에서는 이 테이블로 시간 여행을 할 수 있도록 다음 테이블 속성을 설정할 수 있습니다.

```
%sql
--로그 보존 기간을 365일로 설정
```

```

ALTER TABLE Taxdb.tripData SET
TBLPROPERTIES(delta.logRetentionDuration = "365일 간격");

--데이터 파일 보존 기간을 365일로 설정 ALTER TABLE
Taxdb.tripData SET
TBLPROPERTIES(delta.deletedFileRetentionDuration = "interval 365 days");

--데이터 및 로그 파일 보존을 확인하기 위한 테이블 속성을 표시합니다. SHOW TBLPROPERTIES
Taxdb.tripData;

```

출력(관련 부분만 표시됨):

열쇠	값
delta.deletedFileRetentionDuration 간격 365일	
delta.logRetentionDuration 간격 365일	

delta.deletedFileRetentionDuration 및 delta.logRetentionDuration 은 테이블 속성 이므로 테이블을 처음 생성할 때 이러한 속성을 설정하거나 테이블이 생성된 후 테이블의 속성을 변경할 수 있습니다.

앞의 예에서 테이블의 속성을 변경한 후 SHOW TBLPROPERTIES 명령을 실행하면 Taxdb.tripData 에 대한 삭제된 파일 및 로그 파일의 보존 간격이 반환되는 것을 볼 수 있습니다. 두 간격을 모두 365일로 설정하면 비즈니스 요구 사항과 규제 요구 사항을 모두 충족하기 위해 작년 중 언제든지 이 테이블로 시간 여행을 할 수 있습니다.

데이터 보관 특정 기간

동안 데이터를 보관해야 하는 규제 또는 보관 목적의 경우 시간 이동 및 파일 보관을 사용하여 이 데이터를 저장하면 스토리지 비용으로 인해 비용이 많이 들 수 있습니다. 비용을 최소화하기 위한 대체 솔루션은 CREATE TABLE AS SELECT 패턴을 사용하여 새 테이블을 생성하여 매일, 매주 또는 매월 방식으로 데이터를 보관하는 것입니다.

```

%sql --
CREATE OR REPLACE TABLE archive.tripData를 생성하거나 교체하여 테
이를 보관 DELTA AS SELECT 사용
★

```

Taxdb.tripData에서

이러한 방식으로 생성된 테이블은 원본 테이블과 비교하여 독립적인 기록을 갖게 됩니다. 따라서 원본 테이블과 새 테이블에 대한 시간 이동 쿼리는 보관 빈도에 따라 다른 결과를 반환할 수 있습니다.

진공

이전 섹션에서는 보존 임계값을 설정하고 논리적으로 삭제되어 더 이상 델타 테이블에서 참조되지 않는 데이터 파일을 제거 할 수 있다는 것을 배웠습니다. 이는 VACUUM 명령이 실행 되지 않는 한 이러한 데이터 파일은 스토리지에서 자동으로 물리적으로 삭제되지 않는다는 점을 상기시켜 줍니다. VACUUM은 사용자가 더 이상 필요하지 않은 이전 버전의 데이터 파일 및 디렉터리를 물리적으로 삭제하는 동시에 테이블의 보존 임계값을 고려하도록 설계되었습니다.

VACUUM을 사용하여 이전 버전의 데이터 파일을 물리적으로 삭제하는 것은 주로 다음과 같은 경우에 중요합니다.
두 가지 이유:

비용

오래되고 사용되지 않는 데이터 파일을 저장하면 특히 자주 변경되는 데이터의 경우 클라우드 스토리지 비용이 기하급수적으로 증가할 수 있습니다. 사용하지 않는 데이터 파일을 제거하여 이러한 비용을 최소화하십시오.

규제 감사 및

규제 준수(예: GDPR)에서는 일부 기록을 영구적으로 제거하여 더 이상 사용할 수 없도록 요구할 수 있습니다. 이러한 기록이 포함된 파일을 물리적으로 삭제하면 해당 규제 요구 사항을 충족하는 데 도움이 될 수 있습니다.

그림 6-4는 VACUUM 의 효과를 보여주기 위해 서로 다른 버전 간의 델타 테이블에 있는 로그 및 데이터 파일의 압축된 버전을 보여줍니다 .

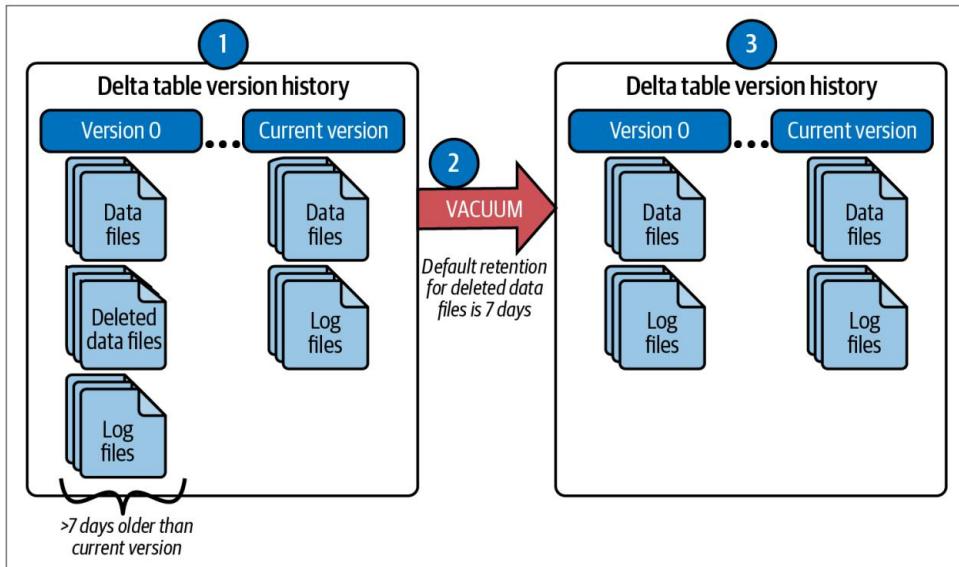


그림 6-4. Delta 테이블에서 VACUUM 명령을 실행한 결과

그림에서 번호가 매겨진 단계는 다음과 같습니다.

1. 테이블의 버전 0이 보존 임계값(7일 이상)을 초과했습니다. 이 버전의 테이블에는 로그 파일, 테이블의 현재 버전에서 사용되는 데이터 파일, 현재 버전의 테이블에서 더 이상 사용되지 않는 삭제된 데이터 파일이 포함되어 있습니다.

2. VACUUM 명령이 테이블에서 실행됩니다.
 - 삭제된 데이터 파일의 기본 보존 기간은 7일입니다.
3. VACUUM 명령을 실행한 후 버전 0에서 논리적으로 삭제된 데이터 파일은 기본 삭제 파일 보존 기간인 7일보다 길기 때문에 스토리지에서 물리적으로 제거됩니다. • 로그 파일은 삭제되지 않고 데이터 파일만 삭제되었습니다. • 현재 버전의 테이블에서 여전히 사용되는 데이터 파일은 제거되지 않았습니다.

VACUUM 구문 및 예 테이블을 Vacuum할 때 매

개변수 없이 VACUUM을 지정하면 기본 보존 기간보다 이전 버전에서 필요하지 않은 파일을 Vacuum할 수 있습니다. RETAIN num HOURS 매개변수를 사용하여 매개변수에 지정된 시간보다 큰 버전에 필요하지 않은 파일을 정리할 수도 있습니다.

테이블을 정리하려면 테이블 이름이나 파일 경로를 지정한 다음 추가 매개변수를 추가하세요.

```
%sql
--기본 보존 기간보다 오래된 버전에는 진공 파일이 필요하지 않습니다. VACUUM Taxdb.tripData;
```

```
VACUUM './chapter06/YellowTaxisDelta/'; --경로 기본 테이블의 진공 파일
```

```
VACUUM delta.`./chapter06/YellowTaxisDelta/`;
```

```
--100시간이 지난 버전에서는 진공 파일이 필요하지 않습니다. VACUUM delta.`./chapter06/YellowTaxisDelta/` RETAIN 100 HOURS;
```

테이블을 진공 청소기로 청소하기 전에 매개변수를 사용하여 VACUUM을 실행할 수도 있습니다.

삭제하기 전에 삭제할 파일 목록을 보려면 DRY RUN을 수행하세요.

```
%sql
VACUUM Taxb.tripData DRY RUN --삭제할 파일 목록을 가져오는 dry run
```

출력(관련 부분만 표시됨):

```
+-----+-----+
| 길
+-----+-----+
| dbfs:/xxx/chapter06/YellowTaxisDelta/part-xxxx.c000.snappy.parquet |
| dbfs:/xxx/chapter06/YellowTaxisDelta/part-xxxx.c000.snappy.parquet |
| dbfs:/xxx/chapter06/YellowTaxisDelta/part-xxxx.c000.snappy.parquet |
+-----+-----+
```

Delta 테이블에서 삭제될 출력의 파일 목록을 볼 수 있습니다.
VACUUM을 실행하려는 경우 Taxdb.tripData .

VACUUM은 또한 Delta 트랜잭션 로그에 커밋됩니다. 즉,
DESCRIBE HISTORY를 사용하는 이전 VACUUM 커밋 및 OperationMetrics :

```
%sql
내역 설명 Taxdb.tripData --이전 진공 커밋 보기
```

출력(관련 부분만 표시):

```
+-----+-----+-----+
| [비전] 운영 | 작업매개변수 | 작업메트릭스
+-----+-----+-----+
| 예스 | 진공 종료 [{"상태": "완료"}] | {"numDeletedFiles": "100" |
| || "numVacuumedDirectories": |
| || "1"} |
+-----+-----+-----+
| 예스 | VACUUM START["retentionCheckEnabled":|| |
| || {"numFilesToDelete": "100"} |
+-----+-----+-----+
```

출력에서 retentionCheckEnabled 인 경우 OperationParameters가 표시되는지 확인합니다.
참 또는 거짓입니다. 또한 OperationMetrics 에
삭제된 파일과 정리된 디렉터리 수.

VACUUM 및 기타 유지 관리 작업을 얼마나 자주 실행해야 합니까?

메인 테이블 외부의 모든 테이블에서 정기적으로 VACUUM을 실행하는 것이 좋습니다.
과도한 클라우드 데이터 스토리지 비용을 줄이기 위한 ETL 워크플로우입니다. 정확한 과학은 없습니다
이는 VACUUM을 얼마나 자주 실행해야 하는지를 정확하게 나타냅니다. 오히려 당신의 결정은
빈도는 주로 예산에 따른 스토리지 비용과 비즈니스에 따라 결정되어야 합니다.
규제 요구 사항. 정기적으로 발생하는 유지 관리 작업을 VACUUM 으로 예약
이러한 요소를 적절하게 충족하려면 테이블을 사용하는 것이 좋습니다.

이 유지 관리 작업에는 다음과 같은 다른 파일 정리 작업도 포함될 수 있습니다.
OPTIMIZE는 기본 ETL 워크플로 외부에서 별도의 워크플로로 실행되어야 합니다.
여러 가지 이유로:

리소스 활용 파일 정리 작

업은 리소스 집약적일 수 있으며 기본 워크플로와 리소스를 놓고 경쟁할 수 있어 전반적인 성능이 저하될 수 있습니다. 따라서 사용량이 적은 시간 이외의 유지 관리 기간을 지정해야 합니다. 이러한 유형의 작업에는 일반적으로 고정 클러스터 크기를 사용하는 일반 워크플로와 달리 자동 크기 조정과 같은 다양한 클러스터 크기 권장 사항이 필요합니다. 이 장의 끝 부분에서 클러스터 크기 권장 사항에 대해 자세히 알아볼 것입니다.

격리 잠재적

인 충돌을 피하기 위해 파일 정리 및 통합을 수행하는 프로세스를 격리하여 델타 테이블에 단독으로 액세스할 수 있도록 하는 것이 가장 좋습니다.

모니터링 이러

한 프로세스를 분리하면 튜닝을 위한 진행 상황과 리소스 소비를 추적할 수 있도록 성능을 모니터링하는 것이 훨씬 쉬워집니다. 또한 격리된 프로세스를 사용하면 프로세스가 병렬로 실행될 때 디버깅 복잡성이 줄어들고 병목 현상을 더 쉽게 식별할 수 있습니다.

VACUUM 과 같은 유지 관리 작업에 대해 별도의 워크플로를 예약하면 작업 및 워크플로에 대한 리소스 관리, 격리, 모니터링 및 전반적인 제어 기능을 강화할 수 있습니다.

유지 관리 작업 빈도와 관련하여 기억해야 할 중요한 설정은 기본 보존 기간입니다. VACUUM 의 기본 보존 임계값은 7일입니다. 필요에 따라 언제든지 델타 테이블의 보존 임계값을 늘릴 수 있습니다. 보존 임계값을 줄이는 것은 권장되지 않습니다. 모든 테이블에서 정기적으로 VACUUM을 실행하더라도 테이블의 보존 설정에 따라 제거할 수 있는 데이터 파일만 제거됩니다. 더 높은 임계값을 설정하면 더 많은 테이블 기록에 액세스할 수 있지만 저장되는 데이터 파일 수가 늘어나 클라우드 공급자의 스토리지 비용이 더 커집니다. 따라서 보존 임계값과 요구 사항 및 예산의 균형을 맞추는 것이 항상 중요합니다.

VACUUM 경고 및 고려 사항

VACUUM은 오래되고 사용되지 않는 데이터를 스토리지에서 물리적으로 제거하므로 정상적인 데이터 작업을 중단하지 않고 수행할 수 있는 영향이 적은 작업으로 설계 되었지만 충돌이나 테이블 손상을 방지하기 위해 고려해야 할 몇 가지 사항이 있습니다.

- 보존 간격을 7일보다 짧게 설정하는 것은 권장되지 않습니다.

짧은 보존 간격으로 테이블에서 VACUUM을 실행하는 경우 판독기 또는 기록기에 필요한 커밋되지 않은 파일과 같이 아직 활성 상태인 파일이 삭제될 수 있습니다. VACUUM이 커밋되지 않은 파일을 삭제 하면 읽기 작업이 실패하거나 테이블이 손상될 수도 있습니다.

- Delta Lake에는 위험한 VACUUM 명령 실행을 방지하기 위한 안전 검사 기능이 있습니다. 이 테이블에서 지정하려는 보존 간격보다 오래 걸리는 작업이 없다고 확신하는 경우 Spark 구성 속성 Spark.databricks.delta.retentionDurationCheck.enabled =를 설정하여 이 안전 검사를 해제할 수 있습니다. 거짓.

- Spark.databricks.delta.retentionDurationCheck.enabled 를 false 로 설정하는 경우 가장 오래 실행되는 동시 트랜잭션보다 긴 간격과 모든 스트림이 테이블에 대한 최신 업데이트보다 지연될 수 있는 가장 긴 기간을 선택해야 합니다.
- Spark.databricks.delta.retentionDurationCheck.enabled 를 비활성화하지 마십시오. RETAIN 0 HOURS 로 구성된 VACUUM을 실행합니다.
- VACUUM RETAIN num HOURS를 실행하는 경우 RETAIN num HOURS를 보존 기간보다 크거나 같은 간격으로 설정해야 합니다 . 그렇지 않고 Spark.databricks.delta.retentionDurationCheck.enabled = true 인 경우 오류가 발생합니다 . 이 테이블에서 INSERT/UPDATE/ DELETE/OPTIMIZE 와 같은 작업이 수행되고 있지 않다고 확신하는 경우 Spark.databricks.delta.retentionDurationCheck.enabled = false 를 설정하여 이 검사를 해제하여 다음을 방지 할 수 있습니다.

예외 오류.

- Delta 테이블에서 VACUUM을 실행하면 테이블에서 다음 파일이 제거됩니다.

기본 파일 시스템: —

_delta_log와 같이 밑줄로 시작하는 디렉터리를 무시하고 Delta Lake에서 유지 관리하지 않는 모든 데이터 파일입니다. 8 장 에서 자세히 알아볼 구조적 스트리밍 체크포인트와 같은 추가 메타데이터를 델타 테이블 디렉터리에 저장하는 경우 _checkpoints와 같은 디렉터리 이름을 사용하세요.

- 오래된 데이터 파일(델타 테이블에서 더 이상 참조되지 않는 파일)
보존 기간보다 오래되었습니다. • Vacuum

은 파일을 제거하므로 테이블 크기와 제거할 파일 수에 따라 프로세스에 다소 시간이 걸릴 수 있다는 점에 유의하는 것이 중요합니다.

OPTIMIZE를 정기적으로 실행하여 작은 파일을 제거하고 제거해야 하는 파일 수를 줄이세요. OPTIMIZE를 일반 VACUUM 실행과 결합하면 오래된 데이터 파일의 수가 최소화됩니다.

- 보존 기간보다 오래된 버전으로 시간 여행을 할 수 있는 기능이 상실되었습니다.

VACUUM을 실행한 후 .

- 시간 여행을 할 수 있는 전체 기록의 완전한 호환성을 유지하려면 삭제된 파일 보존 기간을 로그 보존 기간과 동일하게 설정하십시오.

— 즉, 기본 설정으로 VACUUM을 실행하면 VACUUM을 실행한 시점으로부터 7일 전으로만 시간 여행을 할 수 있다는 의미입니다 . • 식별하고 제거해야 하는 사용되지 않는 파일 수에 따라 VACUUM 명령을 실행하는 데 시간이 걸릴 수 있습니다. Spark 클러스터의 비용과 성능을 최적화하려면 VACUUM이 작업을 수행하기 위해 따르는 다음 단계를 기반으로 자동 크기 조정 및 구성하는 클러스터를 사용하는 것이 좋습니다 . — VACUUM 작업의 1단계는 다음을 사용하여 사용되지 않는 파일을 식별합니다 . 드라이버 노드가 유휴 상태인 동안 Spark 클러스터의 작업자 노드. 따라서 각각 최소 8개의 코어가 있는 1~4개의 작업자 노드를 사용해야 합니다.

— VACUUM 작업의 2단계에서는 Spark 클러스터의 드라이버 노드를 사용하여 식별된 파일을 삭제합니다. 따라서 메모리 부족 오류를 방지하려면 8~32개 코어가 있는 드라이버 노드를 사용해야 합니다.

데이터 피드 변경

지금까지 이 장에서는 데이터 및 파일 보존을 통해 시간 여행을 통해 특정 시점의 다양한 버전의 데이터를 탐색할 수 있다는 점을 배웠습니다. 그러나 시간 이동은 행 수준 변경 사항을 추적하지 않으며 오히려 행 수준 데이터가 여러 버전에서 삽입, 업데이트 또는 삭제되는 방식을 추적하지 않습니다. 그리고 Delta Lake는 전체 테이블 버전을 비교하는 대신 다양한 버전에서 이러한 변경 사항을 볼 수 있는 보다 효율적인 방법을 제공합니다. 여러 버전에 걸쳐 행 수준 변경 사항을 효율적으로 추적하는 것을 CDF(변경 데이터 피드)라고 합니다.

Delta 테이블에서 활성화되면 Delta Lake는 테이블에 기록된 모든 데이터에 대한 "변경 이벤트"를 기록합니다. 여기에는 지정된 행이 삽입, 삭제 또는 업데이트되었는지 여부를 나타내는 행 데이터와 막타 데이터가 포함됩니다. 다운스트림 소비자는 SQL 및 DataFrame API를 사용하는 일괄 쿼리와 .readStream을 사용하는 스트리밍 쿼리에서 변경 이벤트를 읽을 수도 있습니다 . 9 장에서 스트리밍 쿼리가 CDF를 사용하는 방법에 대해 자세히 알아볼 것입니다 .

CDF를 사용하면 델타 테이블 파일의 모든 단일 레코드를 처리하거나 테이블의 전체 버전을 쿼리하지 않고도 데이터 변경 사항을 캡처할 수 있습니다. 따라서 하나의 레코드만 변경된 경우 더 이상 파일이나 테이블의 모든 레코드를 읽을 필요가 없습니다.

CDF는 _delta_log와 함께 있고 Delta 테이블 파일의 변경 사항을 유지 관리하는 _change_data라는 별도의 디렉터리에 저장됩니다. CDF는 Delta Lake 시간 이동 및 버전 관리를 보완하는 여러 사용 사례를 지원합니다.

ETL 작업 작업

후 행 수준 변경이 필요한 레코드만 식별하고 처리하면 ETL 작업을 크게 가속화하고 단순화할 수 있습니다. ETL 작업 중 레코드를 충분적으로 로드하는 것은 효율적인 ETL 프로세스에 필수적입니다.

예를 들어, 보고에 사용되는 모든 판매 주문 정보를 포함하고 여러 업스트림 테이블의 조인을 통해 생성된 대규모 비정규화된 테이블이 있는 경우 테이블이 처리될 때마다 모든 레코드를 처리하지 않으려고 합니다.

CDF를 사용하면 업스트림 테이블의 행 수준 변경 사항을 추적하여 어떤 정보 또는 판매 주문 기록이 신규, 업데이트 또는 삭제되었는지 확인한 다음 이를 사용하여 판매 주문 정보가 포함된 테이블을 점진적으로 처리할 수 있습니다.

다운스트림 소비자를 위한 변경 사항 전송 Spark Structured

Streaming 또는 Kafka와 같은 다른 다운스트림 시스템 및 소비자는 CDF를 사용하여 데이터를 처리할 수 있습니다. 예를 들어 8 장에서 자세히 알아볼 스트리밍 쿼리는 변경 피드를 읽어 거의 실시간 분석 및 보고를 위해 데이터를 스트리밍할 수 있습니다.

이벤트 기반 애플리케이션이 있는 경우 Kafka와 같은 이벤트 스트리밍 플랫폼은 변경 피드를 읽고 다운스트림 애플리케이션 또는 플랫폼에 대한 작업을 트리거할 수 있습니다. 예를 들어 전자상거래 플랫폼이 있는 경우 Kafka는 변경 피드를 읽고 델타 테이블에 캡처된 제품 재고 변경 사항을 기반으로 플랫폼에서 거의 실시간 작업을 트리거할 수 있습니다.

감사 추적 테이블 CDF

는 시간 경과에 따른 행 수준 데이터의 변경 사항을 쿼리하여 데이터가 업데이트되거나 삭제된 시기와 시기를 쉽게 확인할 수 있어 특히 시간 이동에 비해 향상된 효율성을 제공합니다. 이를 통해 데이터에 대한 전체 감사 추적이 제공됩니다.

많은 규정 요구 사항에 따라 특정 산업에서는 이러한 행 수준 변경 사항을 추적하고 전체 감사 시험을 유지해야 할 수 있습니다. 예를 들어 의료 분야에서 HIPAA 및 감사 통제를 위해서는 전자 보호 건강 정보(ePHI)와 관련된 활동 또는 변경 사항을 추적하는 시스템이 필요합니다.² Delta Lake의 CDF는 변경 사항 추적을 위한 규제 요구 사항을 지원하는 데 도움이 됩니다.

CDF 활성화

이 Spark 구성 속성을 다음과 같이 설정하여 모든 새 테이블에 대해 CDF를 활성화할 수 있습니다.

```
%sql
```

은 Spark.databricks.delta.properties.defaults.enableChangeDataFeed = true를 설정했습니다.

환경의 모든 테이블에 대해 CDF를 활성화하지 않으려면 테이블을 생성하거나 기존 테이블을 변경할 때 테이블 속성을 사용하여 CDF를 지정할 수 있습니다. 이 장의 시작 부분에서 "장 초기화" 노트북을 실행했을 때 승객 수와 승객 수에 대한 집계 정보가 포함된 외부 델타 테이블이 있었습니다.

² “45 CFR § 164.312 - 기술적 보호 조치.” 코넬 로스쿨. 2013년 1월 25일 . <https://oreil.ly/qWFDe>.

벤더 또는 택시에 대한 요금 금액이 생성되었습니다. "02 - Change Data Feed"3에 대한 노트북에서 찾을 수 있는 다음 예에서는 새 테이블을 생성하고 CDF를 활성화합니다.

```
%sql --
변경 데이터 피드를 사용하여 새 테이블 생성 CREATE TABLE IF NOT EXISTS
Taxdb.tripAggregates (VendorId INT, PassengerCount INT, FareAmount INT)

TBLPROPERTIES(delta.enableChangeDataFeed = true);

-- 기존 테이블을 변경하여 변경 데이터 피드를 활성화합니다. ALTER TABLE myDeltaTable
SET TBLPROPERTIES (delta.enableChangeDataFeed = true);
```



CDF를 활성화한 후에 변경된 내용만 기록되므로 CDF를 활성화하기 전에 테이블에 변경한 내용은 캡처되지 않습니다.

CDF의 레코드 수정

CDF를 시연하기 위해 먼저 나중에 CDF를 볼 수 있도록 방금 생성한 Taxdb.tripAggregate 테이블의 일부 데이터를 INSERT, UPDATE 및 DELETE해 보겠습니다.

```
%sql --
테이블에 레코드 삽입 INSERT INTO
Taxdb.tripAggregates VALUES (4, 500, 1000);
```

```
-- 테이블의 레코드 업데이트 UPDATE
Taxdb.tripAggregates SET TotalAmount = 2500 WHERE VendorId = 1;

-- DELETE FROM Taxidb.tripAggregates WHERE
VendorId = 3 테이블의 레코드를 삭제합니다.
```

이제 테이블이 변경되었으므로 CDF는 행 수준 변경 사항을 캡처했습니다. Delta 테이블이 저장된 위치를 살펴보면 새로운 _change_data 디렉터리를 확인할 수 있습니다.

```
%sh
ls -al /dbfs/mnt/datalake/book/chapter06/TripAggregatesDelta/
```

출력(관련 부분만 표시됨):

```
drwxrwxrwx 2 루트 _change_data drwxrwxrwx 2 루트
트 _delta_log -rwxrwxrwx 1 루트 부분-00000-...
c000.snappy.parquet
```

3 GitHub 저장소 위치: /chapter06/02 - 데이터 피드 변경

```
-rwxrwxrwx 1 루트 부분-00000....snappy.parquet -rwxrwxrwx 1 루트 부
분-00000....c000.snappy.parquet -rwxrwxrwx 1 루트 부분-00001....snappy.parquet
```

이제 새로운 _change_data 디렉터리를 볼 수 있으므로 디렉터리에서 데이터 변경 사항이 포함된 데이터 파일을 볼 수 있습니다.

```
%sh
ls -al /dbfs/mnt/datalake/book/chapter06/TripAggregatesDelta/_change_data
```

산출:

```
-rwxrwxrwx 1 루트 cdc-00000....snappy.parquet -rwxrwxrwx 1 루트
cdc-00001....snappy.parquet
```

_change_data 디렉터리는 데이터 파일에 포함된 변경 캡처 데이터가 포함된 또 다른 메타데이터 디렉터리입니다. 앞으로 데이터를 변경할 때마다 현재 버전의 데이터 파일뿐만 아니라 _change_data 디렉터리의 파일도 업데이트 됩니다. CDF 디렉터리는 _change_data 디렉터리에 업데이트 및 삭제만 저장하는 반면, 삽입의 경우 트랜잭션 로 그에서 직접 CDF를 계산하는 것이 더 효율적이라는 점을 기억하는 것이 중요합니다. 이는 삽입물이 CDF에 캡처되지 않는다는 의미는 아닙니다. 이는 단순히 _change_data 디렉토리에 저장되지 않습니다.



_change_data 디렉터리의 데이터 파일은 테이블의 동일한 보존 정책을 준수합니다. 즉, VACUUM을 실행하면 테이블의 보존 정책을 벗어나는 CDF 데이터가 삭제됩니다.

CDF 보기 발생한 행 수준

변경 사항을 식별하는 데 도움이 되도록 CDF에는 이벤트 유형 및 커밋 정보에 레이블을 지정하는 추가 메타데이터가 포함되어 있습니다. 다음 표에는 추가 메타데이터 열의 스키마가 나와 있습니다.

표 6-2. CDF 메타데이터

열 이름	유형	가치
_change_type	끈	삽입, update_preimage, update_postimage, 삭제
_commit_version	긴	변경 사항이 포함된 테이블 버전
_commit_timestamp	타임스탬프	커밋이 발생한 타임스탬프

CDF의 추가 메타데이터 열을 사용하면 테이블의 행 수준 변경 사항을 쉽게 볼 수 있습니다. 이러한 변경 사항과 CDF 메타데이터 열을 보려면 TABLE_CHANGES(table_str, start [, end]) SQL 명령을 사용할 수 있습니다. 다음 표에서는 이 명령의 인수를 자세히 설명합니다.

표 6-3. TABLE_CHANGES 인수

인수 유형	정의
문자열 선택적으로 정규화된 테이블 이름을 나타냅니다.	
시작	BIGINT 또는 타임스탬프 반환할 변경 사항의 첫 번째 버전 또는 타임스탬프입니다.
끝	BIGINT 또는 Timestamp 반환할 마지막 버전 또는 변경 타임스탬프에 대한 선택적 인수입니다. 그렇지 않은 경우 지정하면 시작부터 현재 변경까지의 모든 변경 사항이 반환됩니다.

이제 Taxdb.tripAggregate 테이블로 돌아가면 여러 DML이 있습니다.

기존 테이블의 INSERT, UPDATE 및 DELETE 데이터에 대한 작업입니다. 표시할 수 있습니다.

시간 여행과 유사한 버전 또는 타임스탬프를 사용하여 테이블 변경 사항을 확인합니다.

TABLE_CHANGES() SQL 명령:

```
%sql
선택하다 *
FROM table_changes('taxidb.tripAggregate', 1, 4)
_commit_timestamp 기준으로 주문
```

산출:

승객수 요금 _change_type	_commit_version
1000 2000 업데이트_사전 이미지	2
1000 2500 업데이트_포스트이미지	2
7000 10000 삭제	상
500 1000 끼워 넣다	4

공급업체 ID _commit_timestamp
1 2023-07-09T19:17:54
1 2023-07-09T19:17:54
상 2023-07-09T19:17:54
4 2023-07-09T19:17:54

이 예에서 행 수준 변경 사항을 살펴보면 다음과 같은 버전을 볼 수 있습니다.

특정 레코드가 삽입, 업데이트 또는 삭제된 시기를 검색하여 확인

_commit_version에서 . _change_type은 작업 유형을 나타냅니다.

업데이트된 레코드의 경우 행 수준 데이터를 나타냅니다.

update_preimage로 표시된 대로 업데이트하고, 업데이트 이후의 행 수준 데이터는 다음과 같습니다.

update_postimage로 표시됩니다.

DataFrame API를 사용하여 동일한 테이블 변경 사항을 볼 수도 있습니다.

.option () 메서드를 사용하고 "readChangeFeed"를 "true"로 설정합니다.

```
%파이썬
# 버전을 사용하여 CDF 테이블 변경 사항 보기
Spark.read.format("델타") \
.option("readChangeFeed", "true") \
.option("startingVersion", 1) \
.option("endingVersion", 4) \
.table("taxidb.tripAggregate")

# 타임스탬프를 사용하여 CDF 테이블 변경 사항 보기
Spark.read.format("델타") \
.option("readChangeFeed", "true") \
.option("startingTimestamp", "2023-01-01 00:00:00") \
.option("endingTimestamp", "2023-01-31 00:00:00") \
.table("taxidb.tripAggregate")
```

이제 기록의 감사 추적을 보고 기록이 어떻게 변경되었는지 확인하려면

시간이 지남에 따라 간단히 CDF와 TABLE_CHANGES()를 사용하여 이 효율성을 포착 할 수 있습니다.
고대에 예를 들어, 특정 공급업체의 가치가

Taxidb.tripAggregates는 시간이 지남에 따라 변경되었습니다. VendorId = 1 인 곳에서
다음 쿼리를 사용할 수 있습니다.

```
%sql
선택하다 *
FROM table_changes('taxidb.tripAggregate', 1, 4)
VendorId = 1 AND _change_type = 'update_postimage'
_commit_timestamp 기준으로 주문
```

산출:

요금	총금액	_change_type	_커밋_버전
10000	25000	업데이트_포스트이미지	2

공급업체 ID	_commit_timestamp
1	2023-07-09T19:17:54.000+000

이는 특정 공급업체의 데이터가 어떻게 업데이트되었는지에 대한 감사 추적을 제공합니다.

시간. 이는 간단한 예이지만 매우 강력할 수 있음을 알 수 있습니다.

값이 많은 대형 테이블의 경우 시간 여행보다 훨씬 더 효율적입니다.

지속적으로 업데이트됩니다.

또는 시계열 분석을 수행하여 신규 공급업체 수를 확인한다고 가정해 보겠습니다.

추가되었습니다(이 테이블의 세분성이 VendorId라고 가정).

생성된 FareAmount 는 특정 시점 이후였습니다. 우리는 WHERE를 사용할 수 있습니다 절을 사용하여 이 정보를 지정하고 CDF를 효율적으로 읽습니다.

```
%sql
선택하다 *
FROM table_changes('taxidb.tripAggregate', '2023-01-01')
VendorId = 1 AND _change_type = '삽입'
(commit_timestamp 기준으로 주문
```

산출:

요금	_change_type	커밋_버전
500	1000	끼워 넣다
4	2023-07-09T19:17:54.000+000	

이는 특히 WHERE VendorId = 테이블 변경 사항을 볼 수 있음을 보여줍니다.

1 AND _change_type = 'insert' (이 경우 커밋 타임스탬프이므로)
2023-01-01.

CDF는 데이터의 변경 사항을 캡처할 수 있는 효율적이고 강력한 기능입니다.

시간. 이는 다른 ETL 작업과 함께 사용하여 유형 2를 천천히 쉽게 빌드할 수 있습니다.

차원을 변경하거나 MERGE 이후 행 수준 변경 사항만 처리할 수 있습니다.

ETL 작업을 가속화하고 점진적으로 로드하기 위한 UPDATE 또는 DELETE 작업

데이터 다운스트림. 앞서 언급했듯이 CDF에는 다른 사용 사례가 있습니다.

그 중 하나는 스트리밍이며, 이에 대해서는 9 장에서 자세히 알아볼 것입니다 .

CDF 경고 및 고려 사항

CDF는 강력한 기능이지만 고려해야 할 몇 가지 사항이 있습니다.

- 변경 기록은 테이블의 데이터 파일과 동일한 보존 정책을 따릅니다. 이것
이는 CDF 파일이 테이블의 보존 정책을 벗어나는 경우 해당 파일이 다음 대상이 됨을 의미합니다.
VACUUM 이며 VACUUM이 실행 되면 삭제됩니다.
- CDF는 테이블 처리에 상당한 오버헤드를 추가하지 않습니다.
DML 작업으로 인라인으로 생성됩니다. 그리고 일반적으로 다음 형식으로 작성된 데이터 파일은
_change_data 디렉토리는 전체 크기에 비해 크기가 훨씬 작습니다.
레코드에 대한 작업만 포함하므로 테이블 작업의 파일을 다시 작성했습니다.
업데이트되거나 삭제된 항목입니다. 앞서 언급한 바와 같이, 레코드에 삽입된 레코드는
테이블은 _change_data 디렉토리에 캡처되지 않습니다.

변경 데이터는 다른 작업과 함께 인라인으로 발생하므로 새 데이터가 커밋되고 테이블에서 사용 가능해지면 변경 데이터를 사용할 수 있습니다.

- CDF는 CDF 이전에 발생한 기록에 대한 변경 사항을 기록하지 않습니다.
활성화되고 있습니다.
- 테이블에 대해 CDF를 활성화하면 더 이상 다음을 사용하여 테이블에 쓸 수 없습니다.
Delta Lake 1.2.1 이하이지만 여전히 표를 읽을 수 있습니다.

결론

이 장에서는 Delta Lake가 버전 제어를 사용하는 방법과 이를 통해 특정 시점에 다양한 버전의 데이터를 탐색하는 동시에 CDF를 사용하여 시간 경과에 따른 데이터의 행 수준 변경 사항을 추적하는 방법에 대해 알아봅니다. Delta Lake의 시간 여행 및 CDF는 사용자가 시간에 따른 변경 사항을 추적할 수 있게 해주는 강력한 기능이며 다운스트림 소비자, ETL 작업, 시계열 분석, 버전 제어, 감사 및 데이터 관리를 지원하는 데 활용할 수 있습니다.

이전 버전의 테이블을 쉽게 복원하거나 쿼리하는 방법을 읽은 후 버전 번호나 타임스탬프를 사용하여 룰백, 감사 및 다양한 사용 사례를 충족할 수 있다는 것을 배웠습니다. DESCRIBE HISTORY 와 같은 명령을 사용하면 테이블의 커밋 기록을 쉽게 볼 수 있습니다. 이는 모두 트랜잭션 로그 및 파일 보존을 통해 가능합니다. Spark 구성 또는 테이블 속성을 사용하여 로그 및 데이터 파일 모두에 대한 기본 설정을 변경하려는 경우 이러한 보존 설정을 추가로 정의할 수 있습니다. 그러면 데이터 파일이 자동으로 제거되지 않으므로 VACUUM 명령을 사용하여 오래된 데이터 파일을 제거할 수 있습니다.

시간 여행을 보완하기 위해 Delta Lake는 테이블에 대한 CDF(변경 데이터 피드)도 제공합니다. 이 기능을 사용하면 변경 사항을 식별하기 위해 각 테이블 기록의 전체 버전을 비교할 필요 없이 다양한 사용 사례에 대해 효율적인 방식으로 행 수준 변경 사항을 캡처할 수 있습니다.

시간 이동, 보존, 유지 관리 및 CDF를 위한 기본 제공 Delta Lake 기능을 사용하면 귀중한 시간과 리소스를 절약하고 규제 및 감사 요구 사항을 충족할 수 있습니다. 이러한 기능은 _change_data 디렉터리와 더 중요하게는 트랜잭션 로그를 통해 가능해지며, 다음 장에서는 트랜잭션 로그가 테이블의 스키마를 저장하고 이를 사용하여 테이블 스키마를 업데이트하고 수정하는 방법에 대해 자세히 알아볼 것입니다.

제7장

스키마 처리

전통적으로 데이터 레이크는 항상 읽기 시 스키마 원칙에 따라 운영되었지만 쓰기 시 스키마를 적용하는 데는 항상 어려움이 있었습니다. 즉, 데이터가 스토리지에 기록될 때 미리 정의된 스키마가 없으며 스키마는 데이터가 처리될 때만 적용됩니다. 분석 및 데이터 플랫폼의 경우 테이블 형식이 쓰기 시 스키마를 적용하여 변경을 방해하는 프로세스를 도입하는 것을 방지하고 적절한 데이터 품질과 무결성을 유지하는 것이 필수적입니다.

쓰기 시 스키마를 준수하는 것이 중요하지만, 오늘날 빠르게 변화하는 비즈니스 환경과 진화하는 데이터 관리 환경, 데이터 소스, 분석, 그리고 단순히 데이터와 그 전체 구조가 끊임없이 변화하고 있다는 점도 인정해야 합니다. 이러한 변경 사항은 새롭고 변화하는 정보를 포착하기 위해 시간이 지남에 따라 발전할 수 있을 만큼 유연한 스키마로 설명되어야 합니다.

기존 데이터 레이크에서 흔히 볼 수 있는 도식적 문제는 스토리지 계층에 관계없이 모든 데이터 플랫폼과 테이블 형식이 지원해야 하는 두 가지 주요 스키마 처리 기능으로 더 분류될 수 있습니다.

스키마 적용 이는 테이블에 추

가되는 모든 데이터가 해당 특정 스키마를 준수하는지 확인하는 프로세스입니다. 여기서 스키마는 열 이름 목록, 해당 데이터 유형 및 선택적 제약 조건을 기준으로 테이블 구조를 정의합니다. 정의된 스키마의 구조에 맞게 데이터를 적용하면 스키마를 준수하지 않는 테이블에 대한 모든 쓰기가 거부되므로 데이터의 품질과 일관성을 유지하는 데 도움이 됩니다. 결과적으로 이는 테이블의 데이터가 정확하고 일관성이 있는지 확인하기 어려울 수 있는 다양한 형식의 데이터로 인해 발생할 수 있는 데이터 품질 문제를 방지하는 데 도움이 됩니다.

스키마 진화 이를 통해 데

이터 레이크에 저장된 데이터는 변화하는 비즈니스 요구 사항과 데이터 환경에 맞게 유연하고 적응할 수 있습니다. 스키마 진화는 매우 의식적이고, 통제되고, 조직화된 방식으로 수행되어야 하며, 대부분 스키마에 열을 추가하는 것으로 제한됩니다.

다행히 Delta Lake에는 유연한 스키마 진화와 엄격한 적용을 모두 허용하는 탁월한 스키마 처리 기능이 있습니다. 이 장에서는 Delta Lake가 스키마 진화 시나리오와 함께 유효성 검사 및 적용을 수행하는 방법과 Delta Lake가 이를 처리할 수 있는 방법을 보여줍니다.

스키마 검증

Apache Spark에서 생성하는 모든 DataFrame에는 스키마가 있습니다. 스키마를 보는 가장 좋은 방법은 데이터의 형태를 정의하는 청사진 또는 구조로 보는 것입니다.

여기에는 각 열의 이름, 해당 데이터 유형, 열이 NULL일 수 있는지 여부, 각 열과 연결된 모든 메타데이터가 포함됩니다.

Delta Lake는 트랜잭션 로그 항목의 MetaData 작업에 Delta 테이블의 스키마를 SchemaString으로 저장합니다. 이 섹션에서는 이러한 항목을 살펴보겠습니다.

다음으로 쓰기 작업 시 스키마 중에 Delta Lake가 적용하는 유효성 검사 규칙을 살펴봅니다. 각 스키마 유효성 검사 규칙에 대한 사용 사례로 이 섹션을 마무리하겠습니다.

코드를 따라가려면 먼저 “[00 - Chapter 초기화](#)” 노트북을 실행하세요. [7장](#)에서는 TaxiRateCode Delta 테이블을 생성합니다.

트랜잭션 로그 항목에서 스키마 보기 Delta Lake는 트랜잭션 로그 내에 JSON 형식으로 스 키마를 저장합니다. 예를 들어 초기화 노트북은 다음과 같은 델타 테이블을 작성합니다.

```
# Delta Lake 형식으로 쓰기 df.write.format("delta")
\ .mode("overwrite") \ .save("/mnt/datalake/
book/chapter07/TaxiRateCode")
```

스키마가 어떻게 저장되는지 살펴보려면 “[01 - Schema Enforcement](#)” 노트북을 열어보세요. 이 노트북에서는 트랜잭션 로 그 파일을 볼 때 테이블의 스키마가 트랜잭션 로그 내부에 JSON 형식으로 저장되어 있음을 알 수 있습니다.

```
%sh
# 스키마 문자열은 트랜잭션 로그 항목의 메타데이터 작업의 일부입니다. # 스키마 문자열에는 로그 항목이 기록된 시점의 # 델타 테이블 파일의 전체 스키마가 포함되어 있습니다. grep "metadata" /dbfs/mnt/datalake/.../TaxiRateCode.delta/_delta_log/...000.json > /tmp/
commit.json python -m json.tool < /tmp/commit.json
```

다음 출력이 표시됩니다.

```
{
  "metaData":
    { "id": "8f348474-0288-440a-a76e-2358ccf45a96", "형식": {
        "공급자": "마루", "옵션": {}

      },
      "schemaString": "{\"type\":\"struct\",\"fields\":[{\"name\":\\\"RateCodeId\\\", \"type\":\\\"정수\\\", \"nullable\":true,\"metadata\":{}},{\"name\":\\\"RateCodeDesc\\\", \"type\":\\\"유형\\\", \"nullable\":true,\"metadata\":{}},\"partitionColumns\":[], \"configuration\":{}, \"createdTime\":1681161987269
    }
  }
}
```

스키마는 열을 나타내는 필드 목록이 포함된 구조(구조체)입니다. 각 필드에는 이름, 유형 및 필드가 필수인지 여부를 알려주는 null 허용 표시기가 있습니다.

각 열에는 메타데이터 필드도 포함되어 있습니다. 메타 데이터 필드는 수행 중인 트랜잭션에 따라 다양한 유형의 정보를 포함할 수 있는 JSON 문자열입니다. 예를 들면 다음과 같습니다.

- 트랜잭션을 실행한 사람의 사용자 이름 • 트랜잭션의 타임스탬프 • 사용된 Delta Lake 버전

- 스키마 파티션 열 • 관련될 수 있는 추가 애플리케이션별 메타데이터
거래

쓰기|스키마

스키마 유효성 검사는 테이블의 스키마와 일치하지 않는 테이블에 대한 쓰기를 거부합니다. Delta Lake는 쓰기 시 스키마 유효성 검사를 수행하므로 테이블에 기록되는 데이터의 스키마를 확인합니다. 스키마가 호환되면 유효성 검사가 통과되고 쓰기가 성공합니다. 데이터의 스키마가 호환되지 않으면 Delta Lake는 트랜잭션을 취소하고 데이터가 기록되지 않습니다.

이 작업은 항상 원자적이므로 데이터의 일부만 테이블에 기록되는 조건이 발생하지 않습니다. 트랜잭션이 성공하면 모든 소스 데이터가 기록되고, 검증에 실패하면 소스 데이터가 기록되지 않습니다.

스키마 유효성 검사가 실패하면 Delta Lake는 사용자에게 불일치에 대해 알리기 위해 예외를 발생시킵니다.

테이블에 대한 쓰기가 호환되는지 확인하기 위해 Delta Lake는 다음 규칙을 사용합니다.

작성할 소스 DataFrame:

존재하지 않는 열은 대상 테이블의 스키마에 포함될 수 없습니다.

누락된 열이 대상 테이블의 스키마에서 Null 하용으로 표시되어 있는 한 새 데이터에 테이블의 모든 열이 포함되지는 않아도 됩니다. 누락된 열이 대상 스키마에서 Null 하용으로 표시되지 않은 경우 트랜잭션이 실패합니다.

대상 테이블의 열 데이터 유형과 다른 열 데이터 유형을 가질 수 없습니다. 예를 들어 대상 테이블의 열에 StringType 데이터가 포함되어 있지만 해당 소스 열에 IntegerType 데이터가 포함되어 있는 경우 스키마 적용은 예외를 발생시키고 쓰기 작업이 수행되지 않도록 합니다. 장소.

대소문자만 다른 열 이름을 포함할 수 없습니다. 예를 들어 원본 데이터에 Foo라는 열이 있고 원본 데이터에 foo라는 열이 있으면 트랜잭션이 실패합니다. 이 특정 규칙 뒤에는 약간의 역사가 있습니다.

- Spark는 대소문자 구분 또는 대소문자 구분 안 함(기본값) 모드에서 사용할 수 있습니다. • 반면에 Parquet는 저장 및 반환 시 대소문자를 구분합니다.
열 정보.

- Delta Lake는 대소문자를 보존하지만 스키마를 저장할 때 구분하지 않습니다.

이전 규칙을 결합하면 다소 복잡해집니다. 따라서 잠재적인 실수, 데이터 손상 또는 손실 문제를 방지하기 위해 Delta Lake에서는 대소문자만 다른 열 이름을 허용하지 않습니다.

스키마 적용 예 스키마 적용에 대한 세부 내용을 살펴보

겠습니다. 일치하는 스키마가 있는 DataFrame을 추가하는 것부터 시작하겠습니다. 그러면 문제 없이 성공할 것입니다. 다음으로 DataFrame에 추가 열을 추가하고 이를 Delta 테이블에 추가해 보겠습니다. 이로 인해 예외가 발생하고 데이터가 기록되지 않았는지 확인합니다.

일치하는 스키마

스키마 적용을 설명하기 위해 먼저 "01 - Schema Enforcement" 노트북의 2단계에 표시된 대로 올바른 스키마가 포함된 DataFrame을 TaxiRateCode 테이블에 추가합니다.

```

# DataFrame의 스키마를 정의합니다.
# 열이 테이블 스키마와 일치한다는 점에 유의하세요.schema = StructType([
    StructField("RateCodeId", IntegerType(), True),
    StructField("RateCodeDesc", StringType(), True)
])

# DataFrame에 대한 행 목록을 만듭니다.
data = [(10, "요금 코드 10"), (11, "요금 코드 11"), (12, "요금 코드 12")]

# 데이터 행과 스키마를 전달하여 DataFrame을 생성합니다. df =
Spark.createDataFrame(data,
Schema)

# 쓰기를 수행합니다. 이 쓰기는 # 문제 없이 성공합니다 df.write \ .format("delta") \ .mode("append")
\ .save("/mnt/
datalake/book/chapter07/
TaxiRateCode")

```

소스와 대상 스키마가 정렬되므로 DataFrame이 테이블에 성공적으로 추가됩니다.

추가 열이 있는 스키마

노트북의 3단계에서는 소스 스키마에 열을 하나 더 추가하려고 합니다.

```

# DataFrame의 스키마를 정의합니다.
# 추가 열을 추가했음을 주목하세요.schema = StructType([
    StructField("RateCodeId", IntegerType(), True),
    StructField("RateCodeDesc", StringType(), True),
    StructField("RateCodeName", StringType(), True)
])

# DataFrame 데이터에 대한 행 목록을 생성합니다 = [
    (15, "요금 코드 15", "C15"),
    (16, "요금 코드 16", "C16"),
    (17, "요금 코드 17", "C17")]
]

# 행 목록과 스키마에서 DataFrame을 생성합니다. df = Spark.createDataFrame(data, Schema)

# 테이블에 DataFrame을 추가하려고 시도합니다. df.write \ .format("delta")
\ .mode("append") \ .save("/mnt/
datalake/book/chapter07/
TaxiRateCode")

```

이 코드는 다음 예외로 인해 실패합니다.

AnalysisException: 델타 테이블(테이블 ID: 8f348474-0288-440a-a76e-2358ccf45a96)에 쓸 때 스키마 불일치가 감지되었습니다.

아래로 스크롤하면 Delta Lake에서 발생한 상황에 대한 자세한 설명을 제공한 것을 볼 수 있습니다.

DataFrameWriter 또는 DataStreamWriter를 사용하여 스키마 마이그레이션을 활성화하려면 '.option("mergeSchema", "true")'를 설정하십시오.
다른 작업의 경우 세션 구성 Spark.databricks.delta.schema.autoMerge.enabled를
"true"로 설정합니다. 자세한 내용은 해당 작업과 관련된 설명서를 참조하세요.

테이블 스키마:

뿌리
-- RateCodeId: 정수(null 허용 = true)
-- RateCodeDesc: 문자열(null 허용 = true)

데이터 스키마:

뿌리
-- RateCodeId: 정수(null 허용 = true)
-- RateCodeDesc: 문자열(null 허용 = true)
-- RateCodeName: 문자열(null 허용 = true)

Delta Lake는 mergeSchema 옵션을 true 로 설정하여 스키마를 발전시킬 수 있음을 알려줍니다. 이에 대해서는 다음 섹션에서 살펴보겠습니다. 그런 다음 디버깅에 매우 유용한 테이블과 소스 데이터 스키마를 보여줍니다.

트랜잭션 로그 항목을 살펴보면 다음과 같은 내용을 볼 수 있습니다.

```
# 모든 트랜잭션 로그 항목의 목록을 만듭니다.
# 항목이 2개만 있음을 알 수 있습니다.
# 첫 번째 항목은 테이블 생성을 나타냅니다. # 두 번째 항목은 유효한 데이터 프레임의 추가입니다. # 예
외가 발생한 이후 위 코드에 대한 항목이 없어 # 트랜잭션이 끝납니다. ls -al /dbfs/mnt/datalake/
book/chapter07/TaxiRateCode/_delta_log/*.json
```

```
-rwxrwxrwx 4월 10일 1일 21:26 /dbfs/.../TaxiRateCode.delta/_delta_log/...000.json -rwxrwxrwx 4월 10일 1일 21:27 /dbfs/...
TaxiRateCode.delta/_delta_log/..001.json
```

첫 번째 항목 (0000…0.json)은 테이블 생성을 나타내고 두 번째 항목 (0000…1.json)은 유효한 DataFrame을 추가합니다. 스키마 불일치 예외가 발생하고 데이터가 전혀 작성되지 않았기 때문에 이전 코드에 대한 항목이 없습니다. 이는 Delta Lake 트랜잭션의 원자적 동작을 보여줍니다.

테이블에 대해 DESCRIBE HISTORY 명령을 실행하면 다음이 확인됩니다.

```
%sql
- 델타 테이블 DESCRIBE HISTORY delta.` /mnt/datalake/book/
chapter07/TaxiRateCode `에 대한 기록을 확인합니다.
```

출력(관련 데이터만 표시됨):

버전 작업	작업매개변수
1	쓰다 {"모드":"추가","partitionBy":[]} {"모드":"덮어쓰기","partitionBy":[]}
0	쓰다 {"모드":"추가","partitionBy":[]} {"모드":"덮어쓰기","partitionBy":[]}

이 섹션에서는 작업 중인 스키마 적용을 살펴보았습니다. 테이블의 스키마는 변경을 선택하지 않는 한 변경되지 않으므로 안심할 수 있습니다. 스키마 적용은 Delta Lake 테이블의 데이터 품질과 일관성을 보장하고 개발자를 정직하게 유지하고 테이블을 깨끗하게 유지합니다.

그러나 비즈니스를 용이하게 하기 위해 테이블에 추가 열이 정말로 필요하다고 의식적으로 결정한 경우 다음 섹션에서 다룰 스키마 진화를 활용할 수 있습니다.

스키마 진화

Delta Lake의 스키마 발전은 테이블의 기존 데이터를 유지하면서 시간이 지남에 따라 델타 테이블의 스키마를 발전시키는 기능을 의미합니다. 즉, 스키마 발전을 통해 데이터 손실이나 테이블에 의존하는 다운스트림 작업 중단 없이 기존 Delta 테이블의 열을 추가, 제거 또는 수정할 수 있습니다. 시간이 지남에 따라 데이터 및 비즈니스 요구 사항이 변하고 새로운 사용 사례를 지원하기 위해 테이블에 새 열을 추가하거나 기존 열을 수정해야 할 수 있으므로 이는 중요합니다.

쓰기 작업 중에 .option("mergeSchema", "true")를 사용하여 테이블 수준에서 스키마 진화를 활성화합니다. Spark.databricks.delta.schema.auto Merge.enabled 를 true 로 설정하여 전체 Spark 클러스터에 대해 스키마 발전을 활성화할 수도 있습니다. 기본적으로 이 설정은 false로 설정됩니다.

스키마 진화가 활성화되면 다음 규칙이 적용됩니다.

- 작성 중인 소스 DataFrame에는 열이 있지만 Delta 테이블에는 열이 없는 경우 이름과 데이터 유형이 동일한 새 열이 Delta 테이블에 추가됩니다. 모든 기존 행은 새 열에 대해 Null 값을 갖게 됩니다.
- 열이 Delta 테이블에 있지만 작성 중인 소스 DataFrame에는 없으면 열은 변경되지 않고 기존 값을 유지합니다. 새 레코드는 소스 DataFrame의 누락된 열에 대해 null 값을 갖게 됩니다.

- 이름은 같지만 데이터 유형이 다른 열이 Delta 테이블에 존재하는 경우 Delta Lake는 데이터를 새 데이터 유형으로 변환하려고 시도합니다. 변환에 실패하면 오류가 발생합니다.
- Delta 테이블에 NullType 열이 추가되면 기존 행은 모두 null로 설정됩니다. 해당 열에 대해.

가장 일반적인 것부터 시작하여 다양한 스키마 발전 시나리오를 살펴보겠습니다. 테이블에 열을 추가하는 것입니다.

열 추가 스키마 적용 예제에

서는 스키마 진화를 사용하여 이전에 스키마 불일치로 인해 거부된 스키마에 RateCodeName 열을 추가할 수 있습니다. 규칙에 따르면 다음과 같습니다.

작성 중인 DataFrame에는 열이 있지만 Delta 테이블에는 열이 없으면 이름과 데이터 유형이 동일한 새 열이 Delta 테이블에 추가됩니다. 모든 기존 행은 새 열에 대해 Null 값을 갖게 됩니다.

“02 - Schema Evolution” 노트북의 코드를 따라갈 수 있습니다. 노트북의 2단계에서는 .write Spark 명령에 .option("mergeSchema", "true")를 추가하여 스키마 진화가 활성화됩니다.

```
# DataFrame의 스키마를 정의합니다.
# 대상 테이블 스키마의 일부가 아닌 # 추가 RateCodeName 열을 확인하세요.
schema = StructType([
    StructField("RateCodeId", IntegerType(), True),
    StructField("RateCodeDesc", StringType(), True),
    StructField("RateCodeName", StringType(), True)
])

# DataFrame 데이터에 대한 행 목록을 생성합니다 =
[
    (20, "요금 코드 20", "C20"),
    (21, "요금 코드 21", "C21"),
    (22, "요금 코드 22", "C22")
]

# 행 목록과 스키마에서 DataFrame을 생성합니다. df = Spark.createDataFrame(data, Schema)

# DataFrame을 멀타 테이블에 추가합니다 df.write
\ .format("delta") \ .option("mergeSchema", "true")
\ .mode("append") \ .save("/mnt/datalake/book/
chapter07/TaxiRateCode")
```

```
# 스키마 인쇄
```

```
df.printSchema()
```

새로운 스키마가 표시됩니다.

뿌리

```
|-- RateCodeId: 정수(null 허용 = true)
|-- RateCodeDesc: 문자열(null 허용 = true)
|-- RateCodeName: 문자열(null 허용 = true)
```

이제 쓰기 작업이 성공적으로 완료되고 데이터가
델타 테이블:

```
%sql
선택하다
*
에서
      delta.`/mnt/datalake/book/chapter07/TaxiRateCode`'
주문
    요율 코드 ID
```

우리는 다음과 같은 결과를 얻습니다:

RateCodeId	요율 코드 자정	요금코드이름
1	표준요금 없는	
2	JFK 없는	
3	뉴어크 없는	
4	나소 또는 웨스트체스터 없는	
5	협상 요금 없는	
6	그룹 라이딩 없는	
20	요금 코드 20	C20
21	요금 코드 21	C21
22	요금 코드 22	C22

새 데이터가 추가되었으며 기존 행의 RateCodeName이 변경되었습니다.

예상되는 null로 설정됩니다. 해당 트랜잭션 로그를 보면

항목을 보면 새 메타데이터 항목이 업데이트된 내용으로 작성되었음을 확인할 수 있습니다.
개요:

```
{
  "메타데이터": {
    "id": "ac676ac9-8805-4aca-9db7-4856a3c3a55b",
    "형식": {
      "공급자": "마루",
      "옵션": {}
    },
    "schemaString": "{"유형": "구조체", "필드": [
      {"name": "RateCodeId", "type": "integer", "nullable": true, "메타데이터": {}},
      {"name": "RateCodeDesc", "type": "string", "nullable": true, "메타데이터": {}}
    ]}
```

```
{
    "name": "RateCodeName",
    "type": "string",
    "nullable": true,
    "metaData": {},
    "partitionColumns": [],
    "config": {},
    "createdTime": 1680650616156
}
}
```

이는 열 추가에 대한 스키마 발전 규칙의 유효성을 검사합니다.

소스 DataFrame의 데이터 열이 누락되었습니다.

다음으로 열 제거가 미치는 영향을 살펴보겠습니다. 규칙에 따르면 다음과 같습니다.

Delta 테이블에는 열이 있지만 작성 중인 DataFrame에는 열이 없는 경우
열은 변경되지 않고 기존 값을 유지합니다. 새 레코드에는 null이 있습니다.
소스 DataFrame에서 누락된 열의 값입니다.

3단계의 "02 - Schema Evolution" 노트북에는 다음과 같은 코드 예제가 있습니다.
DataFrame에서 RateCodeDesc 열을 그대로 두었습니다.

```
# DataFrame의 스키마를 정의합니다.
스키마 = StructType([
    StructField("RateCodeId", IntegerType(), True),
    StructField("RateCodeName", StringType(), True)
])

# DataFrame에 대한 행 목록을 만듭니다.
데이터 = [(30, "C30"), (31, "C31"), (32, "C32")]

# 행 목록과 스키마에서 DataFrame을 만듭니다.
df = Spark.createDataFrame(데이터, 스키마)

# 테이블에 DataFrame을 추가합니다.
df.쓰기 \
    .format("델타") \
    .option("mergeSchema", "true") \
    .mode("추가") \
    .save("/mnt/datalake/book/chapter07/TaxiRateCode")
```

이제 Delta 테이블의 데이터를 살펴보면 다음과 같은 내용을 볼 수 있습니다.

	요율 코드 지정		요금 코드 이름
1	표준요금 없는		
2	JFK 없는		
3	뉴어크 없는		
4	소 또는 웨스트체스터 없는		
5	협상 요금 없는		
6	그룹 라이딩 없는		
20	요금 코드 20	C20	
21	요금 코드 21	C21	

22	요금 코드 22 null	C22	
30	null	C30	
31	null	C31	
32		C32	

다음 동작을 관찰하세요.

- Delta 테이블의 스키마는 변경되지 않습니다.
- 기존 행의 RateCodeDesc 열 값은 변경되지 않습니다.
- 새 DataFrame의 RateCodeDesc 열 값은 NULL로 설정됩니다.
DataFrame에 존재하지 않기 때문입니다.

해당 트랜잭션 로그 항목을 살펴보면

commitInfo 및 3개의 추가 섹션(새 소스 레코드마다 하나씩), 새 항목은 없음
SchemaString은 스키마가 변경되지 않았음을 의미합니다.

```
{
    "커밋정보": {
        ...
    }
}
{
    "추가하다": {
        ...
        "stats": "{\"numRecords\":1,\"minValues\":{\"RateCodeId\":30,
            \"RateCodeName\":\"C30\"},\"maxValues\":
            {\"RateCodeId\":30,\"RateCodeName\":\"C30\"},\"nullCount\":
            {\"RateCodeId\":0,\"RateCodeDesc\":1,
            \"RateCodeName\":0}}",
        ...
    }
}
{
    "추가하다": {
        ...
        "stats": "{\"numRecords\":1,\"minValues\":{\"RateCodeId\":31,
            \"RateCodeName\":\"C31\"},\"maxValues\":
            {\"RateCodeId\":31,\"RateCodeName\":\"C31\"},\"nullCount\":
            {\"RateCodeId\":0,\"RateCodeDesc\":1,
            \"RateCodeName\":0}}",
        "태그": {
            ...
        }
}
{
    "추가하다": {
        ...
    }
}
```

```

"stats": "{\"numRecords\":1,\"minValues\":{\"RateCodeId\":32,
  \"RateCodeName\":\"C32\"},\"maxValues\":
  {\"RateCodeId\":32,\"RateCodeName\":\"C32\"},\"nullCount\"
  :{\"RateCodeId\":0,\"RateCodeDesc\":1,
  \"RateCodeName\":0}}",
  "태그": [
    ...
  ]
}
}

```

이는 소개에서 설명한 대로 열 제거에 대한 규칙의 유효성을 검사합니다.

열 데이터 유형 변경 다음으로 열의 데이터 유형

변경이 미치는 영향을 살펴보겠습니다. 규칙에 따르면 다음과 같습니다.

이름은 같지만 데이터 형식이 다른 열이 Delta 테이블에 있는 경우 Delta Lake는 데이터를 새 데이터 형식으로 변환하려고 시도합니다. 변환에 실패하면 오류가 발생합니다.

“02 -Schema Evolution” 노트북의 4단계에서는 먼저 디렉터리를 제거하여 테이블을 재설정합니다.

```
dbutils.fs.rm("dbfs:/mnt/datalake/book/chapter07/TaxiRateCode", recurse=True)
```

그런 다음 테이블을 삭제할 수 있습니다.

```
%sql
DROP TABLE Taxdb.taxiratecode;
```

다음으로 테이블을 다시 생성하지만 이번에는 RateCodeId에 대해 짧은 데이터 유형을 사용합니다.

```
# CSV 데이터를 읽고, RateCodeId의 데이터 유형을 # short로 변경합니다. df =
spark.read.format("csv") .option("header",
"true") .load("/mnt/datalake /book/chapter07/
TaxiRateCode.csv") df =
df.withColumn("RateCodeId", df["RateCodeId"].cast(ShortType()))

# Delta Lake 형식으로 쓰기 df.write.format("delta")
\ .mode("overwrite") \ .save("/mnt/datalake/
book/chapter07/TaxiRateCode")

# 스키마 인쇄 df.printSchema()
```

새 스키마를 확인하고 이제 RateCodeId가 실제로 짧은 데이터 유형인지 확인할 수 있습니다.

뿌리

```
|-- RateCodeId: 짧음(null 가능 = true)
|-- RateCodeDesc: 문자열(null 허용 = true)
```

다음으로, RateCodeId 열의 데이터 유형을 다음과 같이 변경해 보겠습니다.

ShortType을 IntegerType 으로 변환합니다 . 이는 스키마 발전을 위해 지원되는 변환 중 하나입니다.

```
# DataFrame의 스��마를 정의합니다.
# 이제 RateCodeId를 다음과 같이 정의합니다.
# 정수 유형 스��마 =
StructType([
    StructField("RateCodeId", IntegerType(), True),
    StructField("RateCodeDesc", StringType(), True)
])

# DataFrame 데이터에 대한 행 목록을 생성합니다 = [(20, "Rate Code 20"),
(21, "Rate Code 21"), (22, "Rate Code 22")]

# 행 목록과 스��마에서 DataFrame을 생성합니다. df = Spark.createDataFrame(data, Schema)

# 스키마 진화를 사용하여 DataFrame 작성
df.write \.format("delta") \.option("mergeSchema",
"true") \.mode("append") \.save("/mnt/datalake/
book/chapter07/TaxiRateCode")

# 스키마 인쇄 df.printSchema()
```

이 코드는 다음 스��마를 성공적으로 실행하고 인쇄합니다.

뿌리

```
|-- RateCodeId: 정수(null 허용 = true)
|-- RateCodeName: 문자열(null 허용 = true)
```

새 SchemaString은 해당 트랜잭션 로그 항목에 다음과 같이 기록됩니다.

정수 유형:

```
{
  "metaData": { "id": "7af3c5b8-0742-431f-b2d5-5634aa316e94", "형식": {
    "공급자": "마루", "옵션": {}
  },
  "schemaString": "{\"유형\": \"구조체\", \"필드\": [
    {"name": \"RateCodeId\", \"type\": \"integer\", \"nullable\": true, \"metadata\": {}}, {"name": \"RateCodeDesc\", \"type\": \"string\",
    \"nullable\": true, \"metadata\": {}}]}",
```

```

    "partitionColumns": [], "configuration":  

    {}, "createdTime": 1680658999999  

}  

}

```

현재 Delta Lake는 제한된 수의 변환만 지원합니다.

- NullType 을 다른 유형으로 변환할 수 있습니다 . • ByteType 에서 ShortType 으로 업캐스트할 수 있습니다 . • ShortType 에서 IntegerType 으로 업캐스트할 수 있습니다 (이는 더 일찍).

NullType 열 추가

Delta Lake에서 NullType() 유형은 “02 - Schema”의 5단계에 표시된 것처럼 Null 값을 포함할 수 있는 열을 나타내는 데 사용되는 유효한 데이터 유형입니다.

진화” 노트북:

```

# DataFrame 스키마에 대한 스키마 정의 = StructType([  

    StructField("RateCodeId", IntegerType(), True),  

    StructField("RateCodeDesc", StringType(), True),  

    StructField("RateCodeExp", NullType(), True)  

])  

# DataFrame에 대한 행 목록을 만듭니다.  

데이터 = [  

    (50, "요금 코드 50", 없음),  

    (51, "요금 코드 51", 없음),  

    (52, "요금 코드 52", 없음)]  

# 행 목록과 스키마에서 DataFrame을 생성합니다. df = Spark.createDataFrame(data, Schema)

```

```

df.write \.format("delta") \.option("mergeSchema",  

    "true") \.mode("append") \.save("/mnt/datalake/  

    book/chapter07/TaxiRateCode")

```

```

# 스키마 인쇄  

df.printSchema()

```

이 DataFrame의 스키마는 다음과 같습니다.

```

뿌리  

|-- RateCodeId: 정수(null 허용 = true)  

|-- RateCodeDesc: 문자열(null 허용 = true)  

|-- RateCodeExp: void(null 가능 = true)

```

해당 트랜잭션 로그 항목에 대한 메타데이터 항목을 살펴보면 null 허용 유형이 반영된 것을 볼 수 있습니다.

```
"schemaString": "{\"유형\": \"구조체\", \"필드\": [
  {\"name\": \"RateCodeId\", \"type\": \"integer\", \"nullable\": true, \"metadata\": {}},
  {\"name\": \"RateCodeDesc\",
    \"type\": \"string\", \"nullable\": true, \"metadata\": {}}, {\"name\": \"RateCodeExp\",
    \"type\": \"void\", \"nullable\": true,
    \"metadata\": {}}]}",
```

데이터 유형이 void로 반영된 것을 볼 수 있습니다. SELECT * 를 사용하여 이 테이블을 쿼리하려고 하면 오류가 발생합니다.

```
%sql
선택하다
*
FROM
delta.`/mnt/datalake/book/chapter07/TaxiRateCode`
```

다음과 같은 예외가 발생합니다.

```
java.lang.IllegalStateException: [RateCodeId#26344,RateCodeDesc#26345]에서 RateCodeExp#26346
을 찾을 수 없습니다.
```

이 오류가 발생하는 이유는 Delta Lake의 NullType 열에 정의된 스키마가 없어 Spark가 열의 데이터 형식을 유추할 수 없기 때문입니다. 따라서 쿼리 시 Spark는 NullType 열을 매핑할 수 없으며 SELECT * 특정 데이터 형식을 실행하려고 시도 하지만 쿼리가 실패합니다.

테이블을 쿼리하려면
NullType 열:

```
%sql
선택하다
요율 코드 ID,
요율 코드 설명
에서
delta.`/mnt/datalake/book/chapter07/TaxiRateCode`
```

아무 문제 없이 성공할 것입니다.

명시적 스키마 업데이트

지금까지 우리는 스키마 진화를 활용하여 스키마가 여러 규칙에 따라 진화할 수 있도록 했습니다. 델타 테이블의 스키마를 명시적으로 조작하는 방법을 살펴보겠습니다. 먼저 SQL ALTER TABLE 및 ADD COLUMN 명령을 모두 사용하여 Delta 테이블에 열을 추가합니다. 다음으로, SQL ALTER COLUMN 문을 사용하여 테이블 열에 주석을 추가하겠습니다. 다음으로 ALTER TABLE 의 변형을 사용하겠습니다.

테이블의 열 순서를 변경하는 명령입니다. Delta Lake를 검토하겠습니다.
열 매핑은 다음에 필요하기 때문입니다.

테이블에 열 추가

"03 - 명시적 스키마 업데이트" 노트북의 3단계에는 다음과 같은 예가 있습니다.

SQL ALTER TABLE...ADD COLUMN 명령을 사용하여 델타 테이블에 열을 추가하려면:

```
%sql
ALTER TABLE delta.`/mnt/datalake/book/chapter07/TaxiRateCode`  

RateCodeId 뒤에 RateCodeTaxPercent INT 열을 추가하세요.
```

AFTER 키워드를 사용했기 때문에 열은 다음 뒤에 추가됩니다.

표준 관행과 같이 열 목록의 끝이 아닌 RateCodeId 필드

AFTER 키워드 없이 . 마찬가지로 FIRST 키워드를 사용하여 새 항목을 추가 할 수 있습니다.
열 목록의 첫 번째 위치에 있는 열입니다.

DESCRIBE 명령 으로 스키마를 살펴보면 새 열이 다음과 같은 것을 알 수 있습니다.

실제로 RateCodeId 열 뒤에 삽입되었습니다.

col_name	data_type	코멘트
RateCodeId	int	null
RateCodeTaxPercent	int	null
RateCodeDesc	문자열	null

기본적으로 Null 허용 여부는 true 로 설정되어 있으므로 새로 추가된 열의 모든 값은 null로 설정됩니다.

RateCodeId	RateCodeTaxPercent	RateCodeDesc
1	널 널	표준요금
2	널 널	JFK
3	널	뉴어크
4		나소 또는 웨스트체스터
5	널 널	협상 요금
6		그룹 라이딩

ADD COLUMN 작업에 대한 트랜잭션 로그 항목을 보면 다음이 표시됩니다.

- ADD COLUMN 연산자를 사용한 commitInfo 작업 .
- 새로운 SchemaString을 사용한 MetaData 작업 . SchemaString에서 우리는 새로운 RateTaxCodePercent 열:

```

{
    "커밋정보": {
        ...
        "작업": "열 추가", "작업 매개변수": {

            "열": "[{\\"열\":{\\\"이름\\\":\"RateCodeTaxPercent\\\",\\\"유형\\\":\\\"정수\\\",\\\"nullable\\\":true, \\\"메타데이터\\\":\\\"{}\\\",\\\"위치\\\":\"RateCodeId 이후\\\"}]}"

        },
        ...
    }
}

{
    "메타데이터": {
        ...
        "schemaString": "{\\\"유형\\\":\"구조체\\\",\\\"필드\\\":[
            {\\\"name\\\":\"RateCodeId\\\",\\\"type\\\":\"integer\\\",\\\"nullable\\\":true,\\\"metadata\\\":{}}, {\\\"name\\\":\\\"RateCodeTaxPercent\\\",\\\"type\\\":\\\"integer\\\",\\\"nullable\\\":true,\\\"metadata\\\":{}}, {\\\"name\\\":\"RateCodeDesc\\\",\\\"type\\\":\"string\\\",\\\"nullable\\\":true,\\\"metadata\\\":{}}
        ]\\\",\\\"partitionColumns\\\":[], \\\"configuration\\\":{}, \\\"createdTime\\\":1681168745910
    }
}
}

```

추가 또는 제거 작업이 없으므로 ADD COLUMN이 성공하기 위해 데이터를 다시 작성할 필요가 없습니다 . Delta Lake가 수행해야 했던 유일한 작업은 MetaData 트랜잭션 로그 작업에서 SchemaString을 업데이트하는 것입니다.

열에 주석 추가 "명시적 스키마 업데이트" 노트북의

3단계에서는 ALTER COLUMN 문과 함께 SQL을 사용하여 델타 테이블에 주석을 추가하는 방법을 살펴봅니다 . 예를 들어 표준 Taxidb.TaxiRateCode 테이블이 있는 경우 열에 설명을 추가할 수 있습니다.

```
%sql
--
-- RateCodeId 열에 설명을 추가합니다.
--
ALTER TABLE Taxidb.TaxiRateCode ALTER
COLUMN RateCodeId COMMENT '이것은 탑승 ID입니다.'
```

해당 트랜잭션 로그 항목에서 CHANGE 가 포함된 commitInfo 항목을 볼 수 있습니다 . COLUMN 연산 및 주석 추가:

```
{
    "커밋정보": {
        ...
        "userName": "bennie.haelen@insight.com", "작업": "열 변경", "작업 메개변수": {
            "열": "\\"name\\\"RateCodeId\\",
            "type": "\\"정수\\", "nullable": true, "메타데이터": {
                "comment": "\\"이것은 라이드의 ID입니다\\"
            }
        }
    },
    ...
}
}
```

메타데이터 항목에는 해당 열에 대해 업데이트된 메타데이터가 표시됩니다.

```
"schemaString": "{"유형": "구조체", "필드": [
    {"name": "RateCodeId", "type": "integer", "nullable": true, "metadata": {"comment": "\\"이것은 라이드의 ID\\"}},
    {"name": "RateCodeDesc", "type": "string", "nullable": true, "metadata": {}}]
}
```

DESCRIBE HISTORY 명령을 사용하여 열 변경 사항을 볼 수도 있습니다.

내역 설명 Taxidb.TaxiRateCode

열 순서 변경 기본적으로 Delta Lake

는 처음 32개 열에 대해서만 통계를 수집합니다. 따라서 통계에 포함시키고 싶은 특정 열이 있는 경우 해당 열을 열 순서로 이동하고 싶을 수도 있습니다. “03 - 명시적 스키마 업데이트” 노트북의 4단계에서는 ALTER TABLE 및 ALTER COLUMN을 사용하여 테이블 순서를 변경하는 방법을 확인할 수 있습니다. 현재 테이블은 다음과 같습니다.

```
%sql
설명 Taxidb.TaxiRateCode

+-----+-----+-----+
|col_name          | data_type|코멘트          |
+-----+-----+-----+
|RateCodeId        |int         |라이드의 아이디입니다| null || null |
|RateCodeTaxPercent|int         |
|RateCodeDesc      |문자열      |
+-----+-----+-----+
```

RateCodeDesc 열을 위로 이동하여 RateCodeId 뒤에 나타나도록 한다고 가정해 보겠습니다. ALTER COLUMN 구문을 사용할 수 있습니다.

```
%sql
ALTER TABLE Taxidb.TaxiRateCode ALTER COLUMN RateCodeDesc AFTER RateCodeId
```

이 명령문을 실행한 후 스키마는 다음과 같습니다.

col_name	data_type	코멘트
RateCodeId	int	라이드의 아이디입니다
RateCodeDesc	문자열	null
RateCodeTaxPercent	int	null

단일 ALTER 내에서 열 순서 지정과 설명 추가를 결합할 수 있습니다.

COLUMN 문입니다. 이 작업은 테이블의 모든 데이터를 보존합니다.

Delta Lake 열 매핑

열 매핑을 통해 Delta Lake 테이블과 기본 Parquet 파일 열을 사용할 수 있습니다.

다른 이름을 사용하려면 이를 통해 RENAME 과 같은 Delta Lake 스키마 발전이 가능해집니다.

다시 작성할 필요 없이 Delta Lake 테이블의 COLUMN 및 DROP COLUMN

기본 Parquet 파일.



이 글을 쓰는 시점에서 Delta Lake 열 매핑이 실험 중입니다.
정신적 지원 모드이지만 이는 중요하고 강력한 기능입니다.

많은 일반적인 시나리오를 지원하는 방법에 대해 논의합니다. 더 많은 것을 찾을 수 있습니다.

[Delta Lake 문서](#) 의 열 매핑에 대한 정보

[웹사이트](#).

Delta Lake는 Delta Lake 테이블에 대한 열 매핑을 지원합니다. 이를 통해 메타데이터만 변경하면 데이터 파일을 다시 쓰지 않고도 열을 삭제하거나 이름이 바뀐 것으로 표시할 수 있습니다. 그것 또한 사용자는 하용되지 않는 문자를 사용하여 델타 테이블 열의 이름을 지정할 수 있습니다. 사용자가 CSV 또는 JSON 데이터를 직접 수집할 수 있도록 공백과 같은 Parquet을 사용합니다. 이전 문자로 인해 열 이름을 바꿀 필요 없이 Delta Lake로 제약.

열 매핑에는 다음 Delta Lake 프로토콜이 필요합니다.

- 리더 버전 2 이상
- 라이터 버전 5 이상

델타 테이블에 필요한 프로토콜 버전이 있으면 열 맵을 활성화할 수 있습니다.

`delta.columnmapping.mode`를 이름으로 설정하여 평합니다.

“03 - Explicit Schema Updates” 노트북의 4단계에서 이를 확인하는 것을 볼 수 있습니다.

테이블의 리더 및 라이터 프로토콜 버전을 사용하려면 DESCRIBE EXTENDED를 사용할 수 있습니다.
명령:

```
%sql
확장된 설명 Taxdb.TaxiRateCode
+-----+
|col_name          |data_type
+-----+
|RateCodeId |int | | ..... || 테이블 속성 [delta.minReaderVersion=1,delta.minWriterVersion=2] |
+-----+
```

테이블이 열 매핑에 필요한 프로토콜 버전이 아닌 것을 알 수 있습니다.

다음 SQL 문을 사용하여 버전과 delta.columnmapping.mode를 모두 업데이트할 수 있습니다.

```
%sql
ALTER TABLE Taxidb.TaxiRateCode SET TBLPROPERTIES
  ('delta.minReaderVersion' = '2',
   'delta.minWriterVersion' = '5',
   'delta.columnMapping.mode' = 'name'
  )
```

SET TBLPROPERTIES 문에 해당하는 로그 항목을 살펴보면 꽤 많은 변경 사항이 있음을 알 수 있습니다.

먼저 SET TBLPROPERTIES 항목이 포함된 commitInfo 작업을 확인합니다.

```
{
  "커밋정보": {
    ...
    "작업": "SET TBLPROPERTIES", "작업 매개변수": {
      "속성": "{\"delta.minReaderVersion\":\"2\", \"delta.minWriterVersion\":"
              '\"5\", \"delta.columnMapping.mode\":"
              '\"name\""}",
      ...
    }
  }
}
```

다음으로 minReader 및 minWriter 버전이 업데이트되었음을 알리는 프로토콜 작업이 표시됩니다.

```
{
  "프로토콜": {
    {"minReaderVersion": 2,
     "minWriterVersion": 5
    }
  }
}
```

마지막으로 SchemaString 이 포함된 MetaData 항목을 볼 수 있습니다 . 그러나 이제 열 매핑이 SchemaString 에 추가되었습니다 .

```
{
    "메타데이터": {
        ...
        "schemaString": "{\"유형\": \"구조체\", \"필드\": [
            {"name": "RateCodeId", "type": "정수", "nullable": true, "메타데이터": {"comment": "이것은
                라이드의 ID", "delta.columnMapping.id": 1, "delta.columnMapping.physicalName": "
                \"RateCodeId\"}},
            {"name": "RateCodeDesc", "type": "string", "nullable": true,
                "메타데이터": {"delta.columnMapping.id": 2,
                    "delta.columnMapping.physicalName": "\"RateCodeDesc\""}},
            {"name": "RateCodeTaxPercent", "type": "integer", "nullable": true, "metadata": {
                "delta.columnMapping.id": 3, "delta.columnMapping.physicalName": "
                \"RateCodeTaxPercent\"}}]}
        ...
        "configuration": {
            "delta.columnMapping.mode": "name",
            "delta.columnMapping.maxColumnId": "3"
        },
        ...
    }
}
```

각 열에 대해 다음이 있습니다.

- 공식 Delta Lake 열 이름인 이름(예: RateCodeId). • delta.columnMapping.id: 열의 ID입니다.
이 아이디는 그대로 유지됩니다
안정적인.
- delta.columnMapping.physicalName: Parquet의 물리적 이름입니다.
파일.

열 이름 바꾸기 ALTER

TABLE...RENAME COLUMN을 사용하면 열의 기존 데이터를 다시 쓰지 않고 열 이름을 바꿀 수 있습니다. 이를 활성화하려면 열 매핑이 준비되어 있어야 합니다. RateCodeDesc 열의 이름을 more로 바꾸고 싶다고 가정합니다.

설명적인 RateCodeDescription:

```
%sql-
열 이름 변경을 수행합니다. ALTER TABLE
Taxidb.taxiratecode RENAME COLUMN RateCodeDesc를 RateCodeDescription으로 변경합니다.
```

해당 로그 항목을 보면 이름 변경이 반영된 것을 볼 수 있습니다.

스키마문자열:

```
"schemaString": "{\"유형\": \"구조체\", \"필드\": [
  ...
  {"\"name\": \"RateCodeDescription\", \"type\": \"string\", \"nullable\": \"전설\",
   \"메타데이터\": {\"delta.columnMapping.id\": 2, \"delta.columnMapping.physicalName\": \"RateCodeDesc\"}},
  ...
]}
```

Delta Lake 열 이름이 RateCodeDescription 으로 변경된 것을 확인할 수 있습니다 .

그러나 PhysicalName은 Parquet 파일에서 여전히 RateCodeDesc 입니다. 이것이 바로 델타 레이크입니다.

RENAME COLUMN 과 같은 복잡한 DDL 작업을 수행할 수 있습니다 .

간단한 메타데이터 작업으로 모든 파일을 다시 작성합니다.

테이블 열 교체

Delta Lake에서는 ALTER TABLE REPLACE COLUMNS 명령을 사용하여
새로운 열 집합이 포함된 기존 Delta 테이블의 모든 열. 참고하세요
이렇게 하려면 다음 항목에 설명된 대로 Delta Lake 열 매핑을 활성화해야 합니다.
이전 섹션.

열 매핑이 활성화되면 REPLACE COLUMNS 명령을 사용할 수 있습니다.

```
%sql
ALTER TABLE Taxidb.TaxiRateCode
    열 교체(
        Rate_Code_Identifier INT COMMENT '코드를 식별합니다',
        Rate_Code_Description STRING COMMENT '코드 설명',
        Rate_Code_Percentage INT COMMENT '세율 적용'
    )
```

스키마를 보면 다음이 표시됩니다.

```
%sql
확장된 설명 Taxidb.TaxiRateCode

+-----+-----+
|col_name          |data_type
+-----+-----+
|Rate_Code_Identifier | int |
|Rate_Code_Description| 문자열 |
|Rate_Code_Percentage| 정수 | .....
||      .....
|테이블 속성 ||      [[delta.columnMapping.maxColumnId=6,
|                  | delta.columnMapping.mode=이름,
|                  | delta.minReaderVersion=2,delta.minWriterVersion=5] |
+-----+-----+
```

DESCRIBE 출력에서 새로운 스키마를 볼 수 있으며, 최소 리더 및 라이터 버전도 볼 수 있습니다.

해당 트랜잭션 로그 항목을 살펴보면 REPLACE COLUMNS 작업이 포함된 commitInfo를 볼 수 있습니다.

```
"커밋정보": {
  ...
  "작업": "열 교체", "작업 매개변수": {
    "columns":
      "[ {"name":"Rate_Code_Identifier","type":"integer","nullable":true,
        \"메타데이터\":{\"설명\": \"코드를 식별합니다\"}, {"name":"
          "Rate_Code_Percentage","type":"integer","nullable":true, \"메타데이터"
          \"\":{\"설명\": \"세율 적용\"}}]"
    },
    ...
  }
}
```

MetaData 섹션에서 몇 가지 흥미로운 정보가 포함된 새로운 SchemaString을 볼 수 있습니다. 새로운 Delta Lake 열은 이제 새 ID(4로 시작)를 사용하여 가이드 기반 열 이름에 매핑됩니다.

```
{
  "메타데이터": {
    ...
    "schemaString": "{\"유형\": \"구조체\", \"필드\": [
      {"name": "Rate_Code_Identifier", "type": "integer", "nullable": true,
        "metadata": {
          "comment": "식별합니다. 코드", "delta.columnMapping.id": 4,
          "delta.columnMapping.physicalName": "col-72397feb-3cb0-4613-baad-aa78ffff64a40"}},
      {"name": "Rate_Code_Description", "type": "string",
        "nullable": true, "metadata": {
          "comment": "설명 코드", "delta.columnMapping.id": 5,
          "delta.columnMapping.physicalName": "col-67d47d0c-5d25-45d8-8d0e-c9b13f5f2c6e"}, ...
        "구성": { "delta.columnMapping.mode": "이름",
          "delta.columnMapping.maxColumnId": 6}
    ]}}
```

```
},
...
}
}
```

데이터를 보면 6개 행이 모두 표시되지만 모든 열은 null로 설정되어 있습니다.

Rate_Code_Identifier	Rate_Code_Description	Rate_Code_Percentage
null	null	null
null	null	null
null	null	null
null		null
null 없		null 없
는		

REPLACE COLUMNS 작업은 모든 열 값을 null로 설정합니다.

스키마는 스키마와 데이터 유형이 다르거나 열 순서가 다를 수 있습니다.

오래된 스키마. 결과적으로 테이블의 기존 데이터가 새 스키마에 맞지 않을 수 있습니다.

따라서 Delta Lake는 모든 열의 값을 null로 설정하여 새 항목이

스키마는 테이블의 모든 레코드에 일관되게 적용됩니다.



REPLACE COLUMNS 작업은
Delta의 전체 스키마를 대체하므로 파괴적인 작업
테이블을 만들고 새 스키마에 데이터를 다시 씁니다. 그러므로 당신은
주의해서 사용해야 하며 데이터를 백업해야 합니다.
이 작업을 적용하기 전에.

열 삭제

Delta Lake는 이제 메타데이터 전용 작업으로 열 삭제를 지원합니다.

모든 데이터 파일을 다시 작성합니다. 이를 위해서는 열 매팅을 활성화해야 합니다.

작업.

메타데이터에서 열을 삭제해도 해당 열이 삭제되지는 않는다는 점에 유의하는 것이 중요합니다.

파일의 열에 대한 기본 데이터입니다. 삭제된 열 데이터를 제거하려면

REORG TABLE을 사용하여 파일을 다시 쓸 수 있습니다. 그런 다음 VACUUM 명령을 사용하여 다음을 수행할 수 있습니다.

삭제된 열 데이터가 포함된 파일을 물리적으로 삭제합니다.

Taxidb.TaxiRateCode 테이블의 표준 스키마부터 시작하겠습니다.

뿌리

```
-- RateCodeId: 정수(null 허용 = true)
-- RateCodeDesc: 문자열(null 허용 = true)
```

RateCodeDesc 열 을 삭제한다고 가정해 보겠습니다 . ALTER를 사용할 수 있습니다

이를 수행하려면 DROP COLUMN SQL 명령을 사용하여 TABLE을 사용하세요 .

```
%sql
-- ALTER TABLE... DROP COLUMN 명령을 사용합니다.
-- RateCodeDesc 열을 삭제합니다.
ALTER TABLE Taxidb.TaxiRateCode DROP COLUMN RateCodeDesc
```

DESCRIBE 명령을 사용하여 스키마를 보면

왼쪽의 RateCodeId 열 :

col_name	data_type	댓글
RateCodeId	int	null

테이블을 확인해 보면 삭제된 항목을 제외하고 데이터가 그대로 남아 있음을 알 수 있습니다.

열:

```
%sql
-- 나머지 열 선택
SELECT * FROM Taxidb.TaxiRateCode
```

RateCodeId
1
2
3 4
5
6

해당 트랜잭션 로그 항목에는 다음 섹션이 표시됩니다.

- DROP COLUMNS 작업을 지정하는 commitInfo 작업 :

```
{
    "커밋정보": {
        ...
        "작업": "열 삭제",
        "작업 매개변수": {
            "columns": "[\"RateCodeDesc\"]"
        },
        ...
    }
}
```

- 메타데이터 섹션의 열 매핑을 포함하여 새 스키마를 지정하는 MetaData 작업:

```
{
  "메타데이터": {
    ...
    "schemaString": "{\"type\":\"struct\",\"fields\": [{\"name\": \"RateCodeId\",\"type\":\"integer\",\"nullable\":true,\"메타데이\
    터\": {\"delta.columnMapping.id\":1,\"delta.columnMapping.physicalName\": \"RateCodeId\"}}], \
    ...
    "구성": {
      "delta.columnMapping.mode": "이름",
      "delta.columnMapping.maxColumnId": "2"
    }
  ...
}
}
```

RateCodeDesc 열은 "소프트 삭제"만 되었습니다. 앞서 트랜잭션 로그 항목을 살펴보면 가장 눈에 띄는 것은 있던 것이 아니라 없었던 것이었다. 데이터 파일에 대한 제거 및 추가 작업이 없었으므로 부품 파일이 다시 작성되지 않았으며 이전 부품 파일이 여전히 RateCodeId 및 RateCodeDesc 열과 함께 남아 있습니다.

부품 파일을 보면 하나의 부품 파일이 보입니다.

```
%sh
# 데이터 파일을 표시합니다.
# 전혀 손대지 않은 한 부분의 파일만 있는 것을 볼 수 있습니다. # ls -al /dbfs/mnt/datalake/book/chapter07/
TaxiRateCode.delta
```

```
drwxrwxrwx 2 루트 루트 4096 4월 6일 00:00 _delta_log -rwxrwxrwx 1 루트 루트 980 4월
6일 00:00 part-00000...c000.snappy.parquet
```

Parquet 뷰어¹를 사용하여 파일을 다운로드하고 보면 두 열이 여전히 존재하는 것을 볼 수 있습니다(표 7-1 참조).

¹ 예를 들어, [Parquet 뷰어](#)

표 7-1. DROP COLUMN 이후 Parquet 파일 보기

RateCodeId	RateCodeDesc
1	표준요금
2	JFK
3	뉴어크
4	나소 또는 웨스트체스터
5	협상 요금
6	그룹 탑승

DROP COLUMN은 메타데이터만 업데이트하며, 부품 파일을 추가하거나 제거하지 않습니다. 대용량 파일로 작업할 때 이러한 "일시 삭제된" 데이터가 있으면 작은 파일 문제가 발생할 수 있습니다. 따라서 다음 섹션에서는 REORG TABLE 명령을 사용하여 삭제된 컬럼의 공간을 회수하도록 하겠습니다.

REORG TABLE 명령

REORG TABLE 명령은 ALTER TABLE DROP COLUMN 명령으로 열을 삭제한 이전 섹션에서 만든 일시 삭제된 데이터를 제거하기 위해 파일을 다시 작성하여 Delta Lake 테이블을 재구성합니다.

삭제한 RateCodeDesc 열이 차지하는 공간을 회수하려면 다음 명령을 실행할 수 있습니다.

```
%sql
-- RateCodeDesc 열을 포함하는 부분 파일을 제거하고 -- RateCodeId 열만 포함된 새 부분 파일을 추가하여 테이블을 재
구성합니다. REORG TABLE Taxidb.TaxiRateCode APPLY (PURGE)
```

이 명령을 실행한 후 Delta Lake는 명령을 실행하는 데 사용된 경로를 표시합니다. 이 경우에는 dbfs:/mnt/datalake/book/chapter07/TaxiRate- Code.delta입니다. 또한 추가 및 제거된 파일 수가 포함된 측정항목도 표시됩니다.

```
{
    "numFilesAdded": 1,
    "numFilesRemoved": 1,
    "filesAdded": { "min":
        665, "max": 665,
        "avg": 665,
        "totalFiles": 1,
        "totalSize": 665

    },
    "filesRemoved": { "min":
        980, "max": 980,
        "avg": 980,
        "totalFiles": 1,
        "totalSize": 980

    },
    "partitionsOptimized": 0,
    ...
}
```

한 파일(두 열이 모두 포함된 부품 파일)이 제거되고 다른 파일(RateCodeId 열만 포함된 부품 파일)이 추가되었습니다.

해당 트랜잭션 로그 항목을 살펴보면 다음과 같은 추가 및 제거 작업이 표시됩니다.

```
{
    "제거하다": {
        "경로": "part-00000-...-c000.snappy.parquet",
        ...
    }
}

}{

    "추가": {
        "경
        로": "9g/part-00000-...-c000.snappy.parquet", "partitionValues":
        {},
        ...
        "stats": "{\"numRecords\":6,\"minValues\":{\"RateCodeId\":1}, \"maxValues\":
        {"RateCodeId\":6},\"nullCount\":{\"RateCodeId\":0}}",
        ...
    }
}
}
```

제거 작업에서는 두 열을 모두 포함하는 원본 Parquet 파일을 제거하고 하위 디렉터리에 파일을 추가합니다. 해당 위치를 보면 Parquet 파일이 표시됩니다.

```
%sh
# 이는 RateCodeId 열만 포함하는 새 부품 파일입니다. ls -al /dbfs/mnt/datalake/book/chapter07/TaxiRateCode.delta/9g
```

-rwxrwxrwx 1 루트 루트 665 4월 6일 01:45 part-00000....snappy.parquet

이 파일을 다운로드하여 확인해 보면 **표 7-2** 와 같이 RateCodeId 컬럼만 존재하는 것을 확인할 수 있다.

표 7-2. REORG TABLE 명령 이후에 새로 추가된 부품 파일

요율 코드 ID
1
2
3
4
5
6

열 데이터 유형 또는 이름 변경 테이블을 수동으로 다시 작성

하여 열의 데이터 유형이나 이름을 변경하거나 열을 삭제할 수 있습니다. 이를 위해 `overwriteSchema` 옵션을 사용할 수 있습니다. 표준 스키마부터 시작해 보겠습니다.

```
뿌리
|-- RateCodeId: 정수(null 허용 = true)
|-- RateCodeDesc: 문자열(null 허용 = true)
```

다음으로, RateCodeId 열의 데이터 유형을 정수에서 short로 변경합니다. 테이블을 다시 작성할 수 있습니다. 먼저 테이블을 읽고 `.withColumn` PySpark 함수를 사용하여 RateCodeId 열의 데이터 유형을 변경한 다음 `overwriteSchema` 옵션을 True로 설정하여 테이블을 다시 작성합니다.

```
# 
# overwriteSchema 설정으로 테이블을 다시 작성합니다. # .withColumn을 사용하여 RateCodeId
열의 데이터 유형을 변경합니다. #

Spark.read.table('taxidb.TaxiRateCode') \ .withColumn("RateCodeId", col("RateCodeId").cast("short")) \ .write
\ .mode("덮어쓰기") \ .option("overwriteSchema", "true") \ .saveAsTable('taxidb.TaxiRateCode')
```

DESCRIBE를 사용하여 테이블의 스키마를 확인하면 해당 테이블의 데이터 유형이 변경되는 것을 볼 수 있습니다.
RateCodeId 테이블:

```
%sql
설명 Taxidb.TaxiRateCode

+-----+-----+
|col_name      |데이터_유형| 댓글 |
+-----+-----+
|RateCodeId |smallint |null
|RateCodeDesc|문자열 |null
+-----+-----+  
||
```

이 작업에 대한 트랜잭션 로그 항목을 확인하면 다음 네 가지 항목이 표시됩니다.

1. CREATE OR REPLACE TABLE AS SELECT 작업을 사용한 commitInfo :

```
{
  "commitInfo": {... "작
    업": "SELECT로 테이블 생성 또는 교체",
    ...
  }
}
```

2. SchemaString을 사용한 MetaData 작업 :

```
{
  "메타데이터": {
    ...
    "schemaString": "{\"type\":\"struct\",\"fields\":[{\"name\":
      \"RateCodeId\",\"type\":\"short\",\"null 가능\": true,\"메타데이터\":{}},
      ...
      {"name\":\"RateCodeDesc\",\"type\":\"string\",\"nullable\": true,\"metadata\":
      {}}],\"...\"}
    ...
  }
}
```

3. 테이블에서 이전 부품 파일을 제거하는 제거 작업:

```
{
  "제거하다": {
    "경로": "part-00000-....snappy.parquet",
    ...
  }
}
```

4. 6개의 레코드가 포함된 부품 파일을 추가하는 추가 작업:

```
{
  "추가": { "경
    로": "part-00000-....snappy.parquet", "partitionValues":
    {}, ...
    "stats": "{\"numRecords\":6,\"minValues\":{\"RateCodeId\"
    \":1,\"RateCodeDesc\":\"그룹 라이드\"},
```

```
        \\"maxValues\":[\\"RateCodeId\\":6,\\\"RateCodeDesc\\":\\"\\\"\\\", \\\"nullCount\\\":\n        {\\\"RateCodeId\\":0,\\\"RateCodeDesc\\\":0}}},\n\n        ...\n    }\n}
```

여기서 우리는 PySpark를 사용하여 열의 데이터 유형을 변경할 수 있음을 보여주었습니다. 비록 델타 테이블을 완전히 다시 작성하는 비용이 들지만 말입니다. 동일한 접근 방식을 사용하여 열을 삭제하거나 열 이름을 변경할 수 있습니다.

결론

분석을 위해 ETL을 활용하는 최신 데이터 플랫폼은 다양한 데이터 소스에서 데이터를 수집하므로 항상 데이터 소비자가 됩니다. 그리고 조직이 점점 더 많은 데이터 소스에서 데이터를 계속 수집, 처리 및 분석함에 따라 스키마 진화 및 데이터 검증을 신속하게 처리하는 능력은 모든 데이터 플랫폼의 중요한 측면입니다. 이 장에서는 Delta Lake가 동적 및 명시적 스키마 업데이트를 통해 테이블의 스키마를 발전시키는 동시에 스키마 유효성 검사를 시행할 수 있는 유연성을 어떻게 제공하는지 살펴보았습니다.

Delta Lake는 트랜잭션 로그 항목을 사용하여 MetaData 작업에 Delta 테이블의 스키마를 저장합니다. 열 이름과 데이터 유형이 포함된 이 스키마는 스키마 유효성 검사를 지원하고 시도된 작업에 대한 스키마 불일치를 보고하는 데 사용됩니다. 이 스키마 유효성 검사는 본질적으로 델타 테이블에 대한 작업의 원자성입니다. 이는 트랜잭션 로그 항목에 표시되거나 오히려 스키마 위반에 대한 트랜잭션 로그 항목이 생략될 수 있습니다.

그리고 Delta Lake가 스키마 유효성 검사를 지원한다는 사실을 알게 된 동시에 기존 Delta 테이블의 열을 추가, 제거 또는 수정하기 위한 동적 스키마 진화도 지원한다는 사실도 알게 되었습니다. mergeSchema 옵션을 사용하여 테이블의 스키마를 발전시키거나, 스키마를 명시적으로 업데이트하여 열이나 데이터 유형을 추가, 제거 또는 이름을 바꿀 수 있습니다. 또한 SQL 또는 SQL을 사용하여 주석을 추가하거나 열 순서(데이터 건너뛰기에 중요)를 변경할 수도 있습니다. 데이터프레임 구문. 데이터 유형에 대해 지원되는 변환을 포함하여 이러한 모든 유형의 스키마 작업은 해당 명령(예: REPLACE COLUMNS) 및 트랜잭션 로그 항목과 함께 장 전반에 걸쳐 설명되어 이러한 작업을 수행하고 이러한 동작을 설명합니다.

스키마 발전은 주로 일괄 데이터 작업의 변경에 중점을 두지만 다음 장에서는 Spark Structured Streaming을 사용하여 데이터를 스트리밍하는 데 필요한 요구 사항과 작업을 살펴봅니다.

제8장

스트리밍 데이터에 대한 작업

Spark 구조적 스트리밍은 Apache Spark 2.0에서 처음 도입되었습니다. 구조적 스트리밍의 주요 목표는 Spark에서 실시간에 가까운 스트리밍 애플리케이션을 구축하는 것이었습니다.

구조적 스트리밍은 이전 Spark RDD 모델을 기반으로 한 DStreams(Discretized Streams)라는 이전 하위 수준 API를 대체했습니다. 그 이후로 구조적 스트리밍은 Delta Lake와의 통합을 포함하여 많은 최적화와 커넥터를 추가했습니다.

Delta Lake는 `readStream` 및 `writeStream`이라는 두 가지 주요 연산자를 통해 Spark Structured Streaming과 통합됩니다. 델타 테이블은 스트리밍 소스와 스트리밍 싱크로 모두 사용 할 수 있습니다. Delta Lake는 다음을 포함하여 일반적으로 스트리밍 시스템과 관련된 많은 제한 사항을 극복합니다.

- 자연 시간이 짧은 수집으로 생성된 작은 파일 통합 • 둘 이상의 스트림(또는 동시)으로 "정확히 한 번" 처리 유지
일괄 작업)
- 소스 스트림에 파일을 사용할 때 어떤 파일이 새로운 것인지 효율적으로 검색하기 위해 델타 트랜잭션 로그를 활용합니다.

Spark Structured Streaming에 대한 간략한 검토로 이 장을 시작한 다음 Delta Lake 스트리밍 및 해당 고유 기능에 대한 초기 개요를 살펴보겠습니다. 다음으로 작은 "Hello Streaming World!"를 살펴보겠습니다. Delta Lake 스트리밍 예시.

범위가 제한되어 있지만 이 예는 매우 간단한 맥락에서 Delta Lake 스트리밍 프로그래밍 모델의 세부 사항을 이해할 수 있는 기회를 제공합니다.

데이터의 증분 처리는 널리 사용되는 ETL 모델이 되었습니다. AvailableNow 스트림 트리거링 모드를 사용하면 개발자는 자체 상태 변수를 유지할 필요 없이 증분 파이프라인을 구축할 수 있으므로 더 간단하고 강력한 파이프라인을 얻을 수 있습니다.

델타 테이블에서 CDF(변경 데이터 피드)를 활성화할 수 있습니다. 클라이언트는 SQL 쿼리를 통해 이 CDF 피드를 사용하거나 이러한 변경 사항을 애플리케이션으로 스트리밍하여 감사 시도 생성, 스트리밍 분석, 규정 준수 분석 등과 같은 사용 사례를 활성화할 수 있습니다.

스트리밍 개요

이 장은 Delta Lake 스트리밍 모델에만 적용되지만 Delta Lake 구조적 스트리밍의 고유한 기능을 살펴보기 전에 Spark 구조적 스트리밍의 기본 사항을 간략하게 검토해 보겠습니다.

Spark 구조적 스트리밍 Spark 구조적

스트리밍은 Apache Spark를 기반으로 구축된 실시간에 가까운 스트림 처리 엔진입니다. 이는 지속적인 데이터 스트림의 확장 가능하고 내결함성이 있으며 대기 시간이 짧은 처리를 가능하게 합니다. Spark Streaming은 Kafka, Azure Event Hubs, Amazon S3, Google Cloud Platform의 Pub/Sub, Hadoop 분산 파일 시스템 등.

구조적 스트리밍의 핵심 아이디어는 데이터 스트림을 SQL과 같은 작업을 사용하여 쿼리하고 조작할 수 있는 무한한 테이블과 같은 구조로 처리할 수 있어 데이터를 쉽게 분석하고 조작할 수 있다는 것입니다. Spark 구조적 스트리밍의 많은 이점 중 하나는 사용 용이성과 단순성입니다. API는 익숙한 Spark SQL 구문을 기반으로 구축되었으므로 새로운 복잡한 API 세트를 배우지 않고도 SQL 및 DataFrame 작업에 대한 기존 지식을 활용하여 스트리밍 애플리케이션을 구축할 수 있습니다.

또한 구조적 스트리밍은 오류를 복구하고 각 데이터 포인트가 정확히 한 번 처리되도록 보장할 수 있는 Spark의 처리 엔진을 활용하여 내결함성과 안정성을 제공합니다. 이러한 유형의 내결함성은 짧은 대기 시간과 높은 처리량의 데이터 처리가 필요한 미션 크리티컬 애플리케이션을 구축하는 데 이상적입니다.

Delta Lake 및 구조적 스트리밍 Delta Lake를 구조

적 스트리밍과 함께 활용하면 Delta Lake의 트랜잭션 보장과 Apache Spark 구조적 스트리밍의 강력한 프로그래밍 모델을 모두 얻을 수 있습니다. Delta Lake를 사용하면 이제 Delta 테이블을 스트리밍 소스 및 싱크로 사용할 수 있으므로 스트리밍 방식으로 Raw, Bronze, Silver 및 Gold 데이터 레이크 계층을 통해 데이터를 처리하는 지속적인 처리 모델이 가능해 일괄 작업이 필요하지 않습니다., 결과적으로 솔루션 아키텍처가 단순화되었습니다. 이 장의 후반부에서는 이러한 연속 처리 아키텍처의 예를 제시하겠습니다.

7장 에서는 스키마 적용과 스키마 진화에 대해 논의했습니다. Delta Lake로 스트리밍하면 들어오는 데이터 스트림이 미리 정의된 스키마에 대해 검증되어 데이터 변칙이 데이터 레이크에 유입되는 것을 방지하는 스키마 적용이 제공됩니다. 그러나 비즈니스 요구 사항 변경으로 인해 추가 정보를 캡처해야 하는 경우 Delta Lake의 스키마 진화 기능을 활용하여 시간이 지남에 따라 스키마가 변경되도록 할 수 있습니다.

스트리밍 예시

매우 간단한 "Hello Streaming World!"를 검토하여 이 섹션을 시작하겠습니다. 델타 테이블과의 스트리밍 기본 사항을 보여주는 예입니다.

안녕하세요 스트리밍 월드

이 섹션에서는 간단한 델타 테이블 스트리밍 시나리오를 만들고 다음과 같은 스트리밍 쿼리를 설정합니다.

- 소스 Delta 테이블의 모든 변경 사항을 스트리밍 DataFrame으로 읽습니다. Delta Lake 테이블의 경우 "변경 사항 읽기"는 "트랜잭션 로그 항목 읽기"와 동일합니다. 테이블에 대한 모든 변경 사항에 대한 세부 정보가 포함되어 있기 때문입니다.
- 스트리밍 DataFrame에 대해 몇 가지 간단한 처리를 수행합니다. • 스트리밍 DataFrame을 대상 델타 테이블에 씁니다.

그림 8-1에 설명된 것처럼 소스에서 스트림을 읽고 대상에 스트림을 쓰는 조합을 종종 스트리밍 쿼리라고 합니다.

스트리밍 쿼리가 실행되면 소스 테이블에서 여러 번의 소규모 일괄 업데이트를 수행하여 데이터가 스트리밍 쿼리를 통해 대상으로 흐를 수 있도록 합니다. 쿼리를 실행하는 동안 쿼리 프로세스 로그를 쿼리하고 스트리밍 쿼리 상태를 유지하는 체크포인트 파일의 내용을 연구합니다.

이 간단한 예를 통해 더 복잡한 예로 넘어가기 전에 Delta Lake 스트리밍 모델의 기본 사항을 완전히 이해할 수 있습니다. 먼저 81 장의 "장 초기화" 노트북을 실행하여 필요한 델타 테이블을 생성합니다.

다음으로, "01 - Simple Streaming" 노트북을 엽니다.

1 GitHub 저장소 위치: /chapter08/00 - 장 초기화

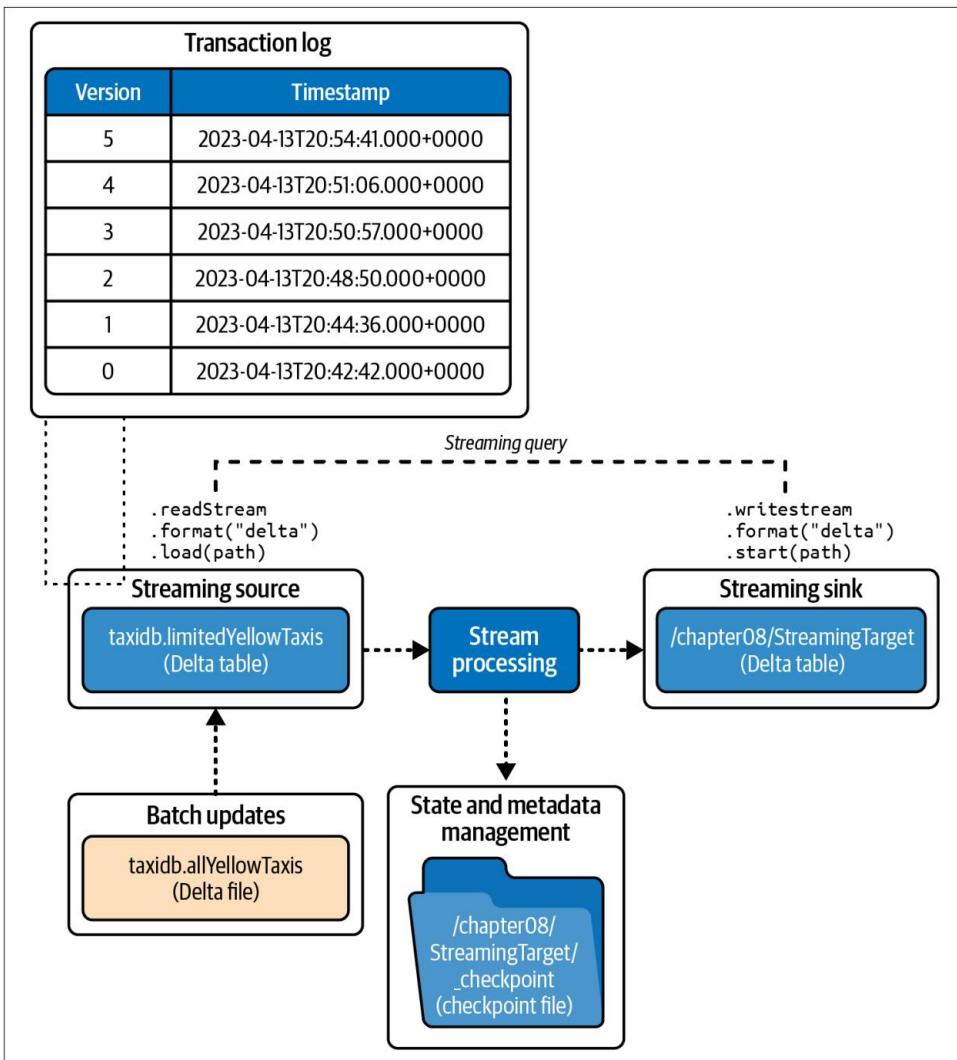


그림 8-1. 기본 스트리밍 예

여기에는 단일 Parquet 파일에 모두 포함된 10개의 노란색 택시 데이터 레코드가 포함된 Delta 테이블이 있습니다.

```
%sh
ls -al /dbfs/mnt/datalake/book/chapter08/LimitedRecords.delta
```

```
drwxrwxrwx 2 루트 루트 4096 4월 11일 19:40 _delta_log -rwxrwxrwx 1 루트 루트 6198
4월 12일 00:04 part-00000....snappy.parquet
```

```
%sql
SELECT * from delta.`/mnt/datalake/book/chapter08/LimitedRecords.delta`
```

출력(관련 부분만 표시):

탑승 ID	공급업체 ID	데리러 갈 시간	드롭타임	
1	1	2022-03-01T00:00:00.000+0000	2022-03-01T00:15:34.000+0000	
2	1	2022-03-01T00:00:00.000+0000	2022-03-01T00:10:56.000+0000	
3	1	2022-03-01T00:00:00.000+0000	2022-03-01T00:11:20.000+0000	
4	2	2022-03-01T00:00:00.000+0000	2022-03-01T00:20:01.000+0000	
5	2	2022-03-01T00:00:00.000+0000	2022-03-01T00:00:00.000+0000	
6	2	2022-03-01T00:00:00.000+0000	2022-03-01T00:00:00.000+0000	
7	2	2022-03-01T00:00:00.000+0000	2022-03-01T00:00:00.000+0000	
8	2	2022-03-01T00:00:00.000+0000	2022-03-01T00:00:00.000+0000	
9	2	2022-03-01T00:00:00.000+0000	2022-03-01T00:00:00.000+0000	
10	2	2022-03-01T00:00:01.000+0000	2022-03-01T00:11:15.000+0000	

스트리밍 쿼리 만들기

먼저, 첫 번째 간단한 스트리밍 쿼리를 생성하겠습니다. 우리는 다음을 읽는 것부터 시작합니다. 다음과 같이 소스 테이블에서 스트리밍합니다.

```
# 소스 "LimitedRecords" 테이블에서 스트리밍을 시작합니다.
# 이제 "read" 대신 "readStream"을 사용합니다.
# 나머지 부분에 대해 우리의 진술은 Delta가 읽은 다른 스파크와 같습니다
stream_df = \
    불꽃 \
    .readStream \
    .format("델타") \
    .load("/mnt/datalake/book/chapter08/LimitedRecords.delta")
```

readStream은 Stream을 제외하고 다른 표준 Delta 테이블 읽기와 같습니다.

접미사. stream_df에서 스트리밍 DataFrame을 반환합니다.



스트리밍 DataFrame은 표준 Spark Data Frame과 매우 유사합니다.
Frame을 사용하면 Spark API를 모든 메서드와 함께 사용할 수 있습니다.
이미 알고 있습니다. 그러나 몇 가지 차이점이 있습니다.
알아두세요. 첫째, 스트리밍 DataFrame은 연속적이고 바인딩되지 않은 데이터 프레임입니다.
각 데이터 조각이 새로운 것으로 처리되는 데이터 시퀀스
DataFrame의 행. 스트리밍 DataFrame은 제한이 없으므로
count() 또는 sort() 작업을 수행할 수 없습니다.

다음으로 DataFrame에서 몇 가지 조작을 수행합니다. 타임스탬프를 추가하므로
소스 테이블에서 각 레코드를 언제 읽는지 알 수 있습니다. 우리도 다 필요하지 않아
소스 DataFrame의 열이므로 필요한 열을 선택합니다.

```
# 우리가 읽은 타임스탬프와 함께 "RecordStreamTime" 열을 추가합니다.
# 스트림의 레코드
stream_df = stream_df.withColumn("RecordStreamTime", current_timestamp())
```

```

# 이것은 스트리밍에서 원하는 열 목록입니다.
# 데이터프레임
select_columns = [
    'RideId', 'VendorId', 'PickupTime', 'DropTime',
    'PickupLocationId', 'DropLocationId', 'PassengerCount',
    'TripDistance', 'TotalAmount', 'RecordStreamTime'
]

# 필요한 열을 선택합니다. # 다른 DataStream과 마찬가지로 스트림을 조작할 수 있습니다. 단, count()와 같
은 일부 작업은 # 제한되지 않은 DataFrame이므로 지원되지 않습니다. stream_df =
stream_df.select(select_columns)

```

마지막으로 DataFrame을 출력 테이블에 작성합니다.

```

# 출력 위치와 체크포인트 위치 정의 target_location = "/mnt/datalake/book/
chapter08/StreamingTarget" target_checkpoint_location = f'{target_location}/
_checkpoint'

# 출력 위치에 스트림을 쓰고 체크포인트 위치의 # 상태를 유지합니다. \ streamQuery = stream_df
\ .writeStream \ .format("delta")
\ .option("checkpointLocation", target_checkpoint_location) \ .start(target_location)

```

먼저 스트림을 작성하려는 대상 또는 출력 위치를 정의합니다. 옵션에서 체크포인트 파일 위치를 정의합니다. 이 검사점 파일은 스트리밍 쿼리의 메타데이터와 상태를 유지합니다. 검사점 파일은 내결합성을 보장하고 오류 발생 시 쿼리 복구를 활성화하는 데 필요합니다. 다른 많은 정보 중에서 스트리밍 소스의 어떤 트랜잭션 로그 항목이 이미 처리되었는지 유지하므로 아직 처리되지 않은 새 항목을 식별할 수 있습니다.

마지막으로 대상 위치로 시작 메소드를 호출합니다. 출력과 체크포인트 파일 모두에 대해 동일한 기본 디렉터리를 사용하고 있다는 점에 유의하세요. 체크포인트 하위 디렉터리에 밑줄 (_checkpoint)만 추가합니다.

트리거를 지정하지 않았기 때문에 스트리밍 쿼리는 계속 실행되므로 실행하고 새 레코드를 확인하고 처리한 후 즉시 다음 레코드 집합을 확인합니다. 다음 섹션에서는 트리거를 사용하여 이 동작을 변경할 수 있음을 살펴보겠습니다.

쿼리 프로세스 로그 스

트리밍 쿼리를 시작하면 스트림 초기화가 표시되고 QPL(쿼리 진행 로그)이 표시됩니다. QPL은 모든 단일 마이크로 배치에서 생성된 JSON 로그이며, 마이크로 배치에 대한 실행 세부 정보를 제공합니다. 표시하는 데 사용됩니다.

노트북 셀의 작은 스트리밍 대시보드. 대시보드는 스트림 애플리케이션의 성능, 처리량 및 대기 시간에 대한 다양한 지표, 통계 및 통찰력을 제공합니다. 스트림 디스플레이를 확장하면 두 개의 탭이 있는 대시보드가 표시됩니다 ([그림 8-2](#)).

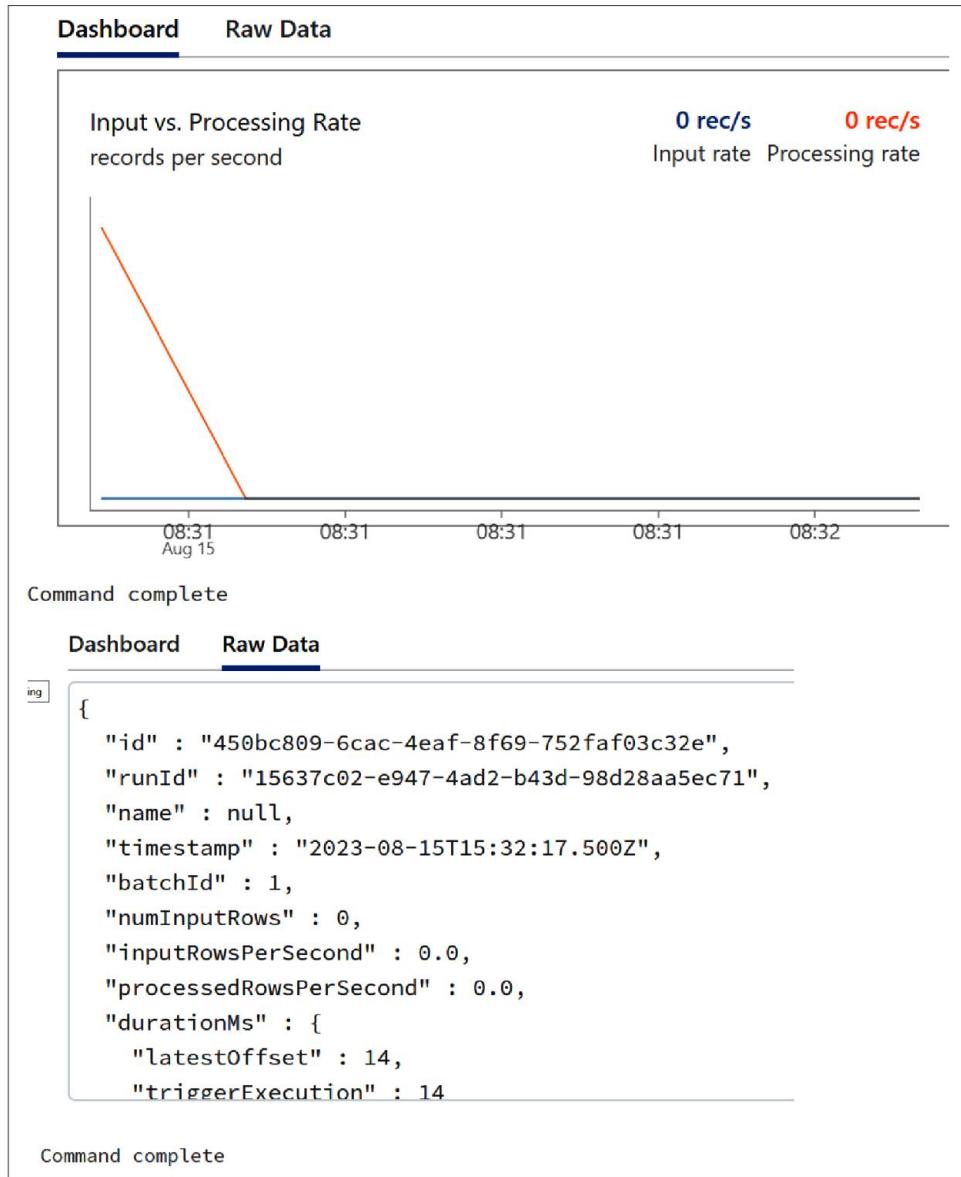


그림 8-2. 쿼리 진행 로그 표시

첫 번째 탭에는 QPL의 주요 지표 중 일부가 그래픽으로 표시되는 대시보드가 포함되어 있습니다. 원시 측정항목은 원시 데이터 탭에 표시됩니다.

쿼리 프로세스 로그의 원시 데이터 일부가 여기에 표시됩니다.

```
{
  "id": "c5eaca75-cf4d-410f-b34c-9a2128ee1944", "runId": "508283a5-9758-4cf9-8ab5-3ee71a465b67", "name": null,
  "timestamp": "2023-05-30T16:31:48.500Z", "batchId": 1,
  "numInputRows": 0, "inputRowsPerSecond": 0.0, "processedRowsPerSecond": 0.0, "durationMs": { "latestOffset": 14,
  "triggerExecution": 15 },
```

QPL의 주요 지표는 로그의 첫 번째 항목인 스트림 고유 ID입니다. 나중에 볼 수 있듯이 이 ID는 스트림을 고유하게 식별하고 체크포인트 디렉토리로 다시 매핑됩니다. 스트림 고유 ID는 스트리밍 대시보드 헤더 위에도 표시됩니다.

쿼리 로그에는 마이크로 배치 ID인 batchID 도 포함되어 있습니다. 모든 스트림에 대해 이 ID는 0으로 시작하고 처리된 모든 마이크로 배치에 대해 1씩 증가합니다. numInputRows 필드는 현재 마이크로 배치에서 수집된 행 수를 나타냅니다.

QPL의 다음 중요한 메트릭 세트는 멜타 소스 및 싱크 메트릭입니다.

- startOffset 및 endOffset 소스는 각 배치가 시작된 위치와 끝났다. 여기에는 다음 하위 필드가 포함됩니다.
 - ReservoirVersion은 현재 데이터가 저장되어 있는 Delta 테이블의 버전입니다.
 - 마이크로 배치가 작동 중입니다.
 - 색인은 처리를 시작할 부품 파일을 추적하는 데 사용됩니다.
 - ReservoirVersion이 현재 스트림이 시작된 Delta 테이블의 버전으로 설정된 경우 isStartingVersion 부울 필드가 true로 설정됩니다.
- 싱크 필드에는 스트리밍 싱크의 위치가 포함됩니다.

マイクロ 배치 1의 소스 및 싱크 측정항목을 살펴보면 다음과 같은 내용을 볼 수 있습니다.

```
"소스": [ {
  "설명": "DeltaSource[dbfs:/mnt/.../LimitedRecords.delta]", "startOffset": { "sourceVersion": 1, "reservoirId": "6c25c8cd-88c1-4b74-9c96-a61c1727c3a2", "저장소 버전": 0,
```

```

    "index" : 0,
    "isStartingVersion" : true },

    "endOffset" :
    { "sourceVersion" : 1,
      "reservoirId" : "6c25c8cd-88c1-4b74-9c96-a61c1727c3a2", "reservoirVersion" :
      0, "index" : 0, "isStartingVersion" :
      true },
      "latestOffset" : null, "numInputRows" :
      0,
      "inputRowsPerSecond" : 0.0,
      "processedRowsPerSecond" :
      0.0, "metrics" : { "numBytesOutstanding" :
      "0", "numFilesOutstanding" : "0" } }, "싱
      크" : { "설명" :
      "DeltaSink[/mnt/datalake/book/
      chapter08/StreamingTarget]",

      "numOutputRows" : -1

    }

```

numInputRows 는 0입니다 . 소스 테이블에 10개의 행이 있다는 것을 알고 있으므로 이는 다소 놀랍게 보일 수 있습니다. 그러나 .writeStream 을 시작했을 때 스트리밍 쿼리가 실행되기 시작하고 배치 0의 일부로 처음 10개 행을 즉시 처리했습니다. 또한 배치 ID가 현재 1이고 배치 ID가 0으로 시작하므로 첫 번째 배치가 이미 있음을 알 수 있습니다. 처리됨.

또한 새 레코드가 처리되지 않았기 때문에 이 배치가 아직 실행되지 않았으므로 ReservoirVersion 이 여전히 0임을 확인할 수 있습니다 . 따라서 우리는 여전히 소스 테이블의 버전 0에 있습니다. 또한 인덱스가 0에 있음을 알 수 있습니다. 이는 첫 번째 데이터 파일을 처리하고 있으며 실제로 시작 버전에 있음을 의미합니다. 소스 테이블의 버전을 표시하여 이를 확인할 수 있습니다.

```
%sql
내역 설명 delta.`/mnt/datalake/book/chapter08/LimitedRecords.delta`
```

출력(관련 부분만 표시):

버전	타임스탬프
0	2023-05-30T16:25:23.000+0000

여기에서 현재 버전이 실제로 0임을 알 수 있습니다. 출력 스트리밍 테이블을 쿼리하여 이를 확인할 수도 있습니다.

```
%sql
SELECT 개수(*) FROM delta.`/mnt/datalake/book/chapter08/StreamingTarget`
```

실제로 10개의 행이 있음을 알 수 있습니다.

```
+-----+
| 개수(1) |
+-----+
| 10      |
+-----+
```

.start를 사용하여 writeStream을 시작했고 쿼리가 얼마나 자주 실행되어야 하는지 표시하지 않았으므로 쿼리가 지속적으로 실행됩니다. writeStream이 완료 되면 다음 readStream 등을 수행합니다. 그러나 소스 테이블에 새 행이 생성되지 않으므로 실제로 아무 일도 일어나지 않으며 출력 레코드 수는 10으로 유지됩니다. 배치 ID는 스트림에서 행을 선택할 때까지 변경되지 않으므로 1로 유지됩니다.

다음으로 소스 테이블에 10개의 새 레코드를 삽입하는 다음 SQL 문을 실행합니다.

```
%sql
-- 이 쿼리를 사용하여 allYellowTaxis 테이블의 무작위 레코드 10개를 LimitedYellowTaxis 테이블에 삽입합니다.
에 집어 넣다
    Taxdb.limitedYellowTaxis
선택하다
*
에서
    Taxdb.allYellowTaxis
랜드()로 주문
제한 10
```

이 쿼리의 주석 처리를 제거하고 실행하면 이제 BatchId가 1로 설정되고 10이 표시됩니다.

새 행:

```
{
    "id": "c5eaca75-cf4d-410f-b34c-9a2128ee1944", "runId": "508283a5-9758-4cf9-8ab5-3ee71a465b67", "name": null, "timestamp": "2023-05-30T16:46:08.500Z", "batchId": 1, "numInputRows": 10, "inputRowsPerSecond": 20.04008016032064,
```

이 배치 ID가 처리되면 스트림에서 새 행이 도착하지 않으므로 배치 ID 2의 레코드 수가 다시 0으로 돌아갑니다.

스트리밍 쿼리는 소스 테이블에서 새 트랜잭션 항목을 찾고 해당 행을 스트리밍 대상에 쓰면서 영원히 계속 실행된다는 점을 기억하세요. 일반적으로 Spark 구조적 스트리밍은 마이크로 배치 기반 스트리밍 서비스로 실행됩니다. 소스에서 레코드 배치를 읽고 레코드를 처리한 후 대상에 쓴 후 즉시 다음 배치를 시작합니다.

새 레코드를 찾고 있습니다(또는 Delta Lake의 경우 새 트랜잭션 항목을 찾습니다).

소스 테이블에 대한 일괄 업데이트를 수행하는 이 모델은 실제 애플리케이션에서는 경제적이지 않습니다. 소스 테이블은 주기적으로만 업데이트되지만 스트리밍 쿼리는 지속적으로 실행되므로 클러스터를 항상 실행해야 하므로 비용이 많이 듭니다. 이 장의 뒷부분에서는 사용 사례에 더 잘 맞도록 스트리밍 쿼리를 수정하겠지만 먼저 체크포인트 파일을 간략하게 살펴보겠습니다.

체크포인트 파일 앞

서 체크포인트 파일이 스트리밍 쿼리의 메타데이터와 상태를 유지한다는 것을 확인했습니다. 체크포인트 파일은 _checkpoint 하위 디렉터리에 있습니다.

```
%sh
ls -al /dbfs/mnt/datalake/book/chapter08/StreamingTarget/_checkpoint/
drwxrwxrwx 2 루트 루트 4096 5월 1일 23:37 __tmp_path_dir drwxrwxrwx 2 루트 루트 4096 5월 1
일 23:37 커밋 -rwxrwxrwx 1 루트 루트 45 5월 2일 15:48 메타데이터 drwxrwxrwx 2 루트
루트 4096 5월 1일 23:37 오프셋
```

하나의 파일(메타데이터)과 두 개의 디렉터리(오프셋 및 커밋)가 있습니다. 각각을 살펴보겠습니다. 메타데이터 파일에는 JSON 형식의 스트림 식별자만 포함됩니다.

```
%sh
헤드 /dbfs/mnt/datalake/book/chapter08/StreamingTarget/_checkpoint/metadata {"id":"c5eaca75-cf4d-410f-
b34c-9a2128ee1944"}
```

오프셋 디렉터리를 보면 각 배치 ID에 하나씩, 두 개의 파일이 표시됩니다.

```
%sh
ls -al /dbfs/mnt/datalake/book/chapter08/StreamingTarget/_checkpoint/offsets
-rwxrwxrwx 1 루트 루트 769 5월 30일 16:28 0 -rwxrwxrwx 1 루트 루트 771
5월 30일 16:46 1
```

파일 0의 내용을 보면 다음과 같은 내용을 볼 수 있습니다.

```
v1
{
  "batchWatermarkMs": 0,
  "batchTimestampMs": 1685464087937, "conf":

    { "spark.sql.streaming.stateStore.providerClass":
      "org.apache.spark.sql.execution.streaming.state.HDFSBackedStateStoreProvider",

        ...
      }
    }
{
```

```

    "sourceVersion": 1,
    "reservoirId": "6c25c8cd-88c1-4b74-9c96-a61c1727c3a2", "reservoirVersion": 0, "index": 0, "isStartingVersion": true

}

```

첫 번째 섹션에는 Spark 스트리밍 구성 변수가 포함되어 있습니다. 두 번째 섹션에는 이전 QPL에서 본 것과 동일한 ReservoirVersion, 인덱스 및 isStartingVersion이 포함되어 있습니다. 여기에 기록되는 것은 배치가 실행되기 전의 상태이므로 버전이 0이고 파일 인덱스도 0이며 isStartingVersion 변수는 시작 버전에 있음을 나타냅니다.

파일 1을 보면 다음과 같은 내용을 볼 수 있습니다.

```

v1
{
  "batchWatermarkMs": 0,
  "batchTimestampMs": 1685465168696, "conf": {
    "spark.sql.streaming.stateStore.providerClass": "org.apache.spark.sql.execution.streaming.state.HDFSBackedStateStoreProvider",
    ...
  }
}
{
  "sourceVersion": 1,
  "reservoirId": "6c25c8cd-88c1-4b74-9c96-a61c1727c3a2", "reservoirVersion": 2, "index": -1, "isStartingVersion": false
}

}

```

이 배치에서는 10개의 추가 레코드가 처리되었으며 다음으로 처리될 수 있는 버전은 2이며, 이는 ReservoirVersion에 반영됩니다. 또한 인덱스가 -1로 설정되어 있으며 이는 현재 버전에 대해 처리할 추가 파일이 없음을 나타냅니다.

커밋 폴더에는 마이크로 배치당 하나의 파일이 포함됩니다. 우리의 경우 각 배치마다 하나씩 두 개의 커밋이 있습니다.

```

drwxrwxrwx
-rwxrwxrwx 1 루트 루트 29 6월 9일 15:55 0
-rwxrwxrwx 1 루트 루트 29 6월 9일 16:15 1

```

각 파일은 마이크로 배치의 성공적인 완료를 나타냅니다. 여기에는 단순히 워터마크가 포함되어 있습니다.

```

%sh
헤드 /dbfs/mnt/datalake/book/chapter08/StreamingTarget/_checkpoint/commits/0

```

이는 다음을 생성합니다.

```
v1 {"nextBatchWatermarkMs":0}
```

이 섹션에서는 델타 스트리밍을 처음 살펴보았습니다. 스트리밍 쿼리의 소스와 싱크로 델타 테이블을 사용하는 간단한 예를 살펴보았습니다.

다음 섹션에서는 충분 처리 모델에서 델타 스트리밍을 활용하는 방법을 살펴보겠습니다.

AvailableNow 스트리밍 Spark 구조

적 스트리밍은 가능한 다양한 트리거 모드를 제공합니다. AvailableNow 트리거 옵션은 maxBytesPerTrigger 와 같은 옵션으로 배치 크기를 구성하는 기능을 통해 사용 가능한 모든 레코드를 충분 배치로 사용합니다.

먼저 스트리밍 대시보드로 이동한 후 취소 링크를 클릭하여 “02 - Simple Streaming” 노트북에서 현재 실행 중인 스트리밍 쿼리를 취소해야 합니다. 그런 다음 취소를 확인하고 스트리밍 쿼리를 중지할 수 있습니다.

소스 테이블은 주기적으로만 업데이트되므로 스트리밍 쿼리가 지속적으로 실행되는 것을 원하지 않습니다. 대신 쿼리를 시작하고, 새 트랜잭션 항목을 선택하고, 해당 레코드를 처리하고, 싱크에 쓴 다음 중지하려고 합니다. 이것이 다음 트리거를 통해 수행할 수 있는 작업입니다. 스트리밍 쿼리에 .trigger(availableNow=True) 코드를 추가하면 노트북 “02 - AvailableNow Streaming”에 표시된 것처럼 쿼리가 한 번 실행된 다음 중지됩니다.

```
# 출력 위치에 스트림을 쓰고, 체크포인트 위치에서 # 상태를 유지합니다. streamQuery
= stream_df
    \
    \
    .writeStream
    \.format("델타") \.option("checkpointLocation", target_checkpoint_location)
    \.trigger(availableNow=True) \.start(target_location)
```

이 노트북을 실행하면 새 레코드가 발견되지 않을 때까지 스트리밍 쿼리가 실행되지만 원본 테이블에 새 레코드가 추가되지 않아 레코드가 발견되지 않고 대상 테이블에 레코드가 기록되지 않습니다. writeStream 의 원시 데이터를 보면 이를 확인할 수 있습니다.

```
{
  "id": "c5eaca75-cf4d-410f-b34c-9a2128ee1944",
  ...
  "numInputRows": 0,
```

이제 노트북의 writeStream 아래에서 SQL 쿼리를 실행하면 소스 테이블에 10개의 레코드가 추가됩니다. 그런 다음 스트리밍 쿼리를 다시 실행하면 10개의 새로운 행이 다시 표시됩니다.

```
{
  "id": "c5eaca75-cf4d-410f-b34c-9a2128ee1944", "runId": "36a31550-c2c1-48b0-9a6f-ce112572f59d", "name": null,
  "timestamp": "2023-05-30T17:48:12.079Z", "batchId": 2,
  "numInputRows": 10,
  "소스": [
    {
      "설명": "DeltaSource[dbfs:/mnt/.../LimitedRecords.delta]", "startOffset": { "sourceVersion": 1,
      "reservoirId": "6c25c8cd-88c1-4b74-9c96-a61c1727c3a2", "ReservoirVersion": 3, "index": -1, "isStartingVersion": false
    }
  ]
}
```

출력에는 현재 3으로 설정된 ReservoirVersion 변수가 있는 소스 섹션도 표시됩니다. 이 경우 ReservoirVersion은 다음으로 가능한 버전 ID를 나타냅니다. 테이블에 대해 DESCRIBE HISTORY를 수행하면 버전 2에 있음을 알 수 있으므로 다음 버전은 3이 됩니다.

```
%sql
은 기록 블록을 설명합니다. `/mnt/datalake/book/chapter08/LimitedRecords.delta`
```

출력(버전 열만 표시됨):

버전
2
1
0

다음 쿼리에서는 원본 테이블에 20개의 레코드를 더 추가합니다. 그런 다음 스트리밍 쿼리를 다시 실행하고 원시 데이터를 보면 20개의 새 레코드가 표시되고 startOffset 의 ReservoirVersion이 이제 3으로 설정되어 있는 것을 볼 수 있습니다.

```
{
  "id": "d89a5c02-052b-436c-a372-2445fb8d88d6",
  ...
  "numInputRows": 20,
  ...
  "소스": [
    {
      "설명": "DeltaSource[dbfs:/mnt/.../LimitedRecords.delta]", "startOffset": { "sourceVersion": 1,
      "reservoirId": "31611029-07d1-4bcc-8ee3-cad0d4fa8bc4", "저수지 버전": 3,
      ...
    },
    ...
  ],
  ...
}
```

이 AvailableNow 모델은 이제 "02 - AvailableNow Streaming" 노트북에 표시된 대로 스트리밍 쿼리를 하루에 한 번, 한 시간에 한 번 또는 사용 사례에서 요구하는 시간 간격에 따라 실행할 수 있음을 의미합니다. Delta Lake는 검사점 파일에 저장된 상태 덕분에 마지막 실행 이후 소스 테이블에 발생한 모든 변경 사항을 항상 선택합니다.



레거시 솔루션에서는 이러한 유형의 증분 처리가 매우 복잡했습니다. ETL 개발자는 마지막 실행 날짜를 유지한 다음 소스 데이터의 전용 날짜에서 쿼리하여 새 행 등을 검색해야 했습니다. AvailableNow 스트리밍은 이 복잡한 모든 것을 추상화하므로 이 프로그래밍 모델을 크게 단순화합니다. 논리.

AvailableNow 트리거 외에도 매우 유사하게 동작하는 RunOnce 트리거도 있습니다. 두 트리거 모두 사용 가능한 모든 데이터를 처리합니다. 그러나 RunOnce 트리거는 단일 일괄 처리의 모든 레코드를 사용하는 반면, AvailableNow 트리거는 적절한 경우 여러 일괄 처리로 데이터를 처리하므로 일반적으로 확장성이 향상됩니다.



스트리밍 쿼리로 사용 가능한 모든 데이터를 사용하려면 AvailableNow 트리거를 사용하세요. 필요할 때 여러 배치를 실행하여 더 나은 확장성을 제공하기 때문입니다.

소스 레코드 업데이트

다음으로, 다음과 같은 업데이트를 실행할 때 어떤 일이 발생하는지 살펴보겠습니다.
성명:

```
%sql
-- 스트리밍 업데이트를 보여주기 위한 쿼리 업데이트 -- 동작

업데이트
    Taxdb.limitedyellowtaxis
세트
    PickupLocationId = 100
어디
    공급업체 ID = 2
```

해당 트랜잭션 로그 항목에 대한 commitInfo 작업을 살펴보면 다음을 볼 수 있습니다.

```
"커밋정보": {
    ...
    "작업": "업데이트",
    ...
}
```

```
        },
        "공책": [
            "notebookId": "3478336043398159",
            ],
        ...
        "오피레이션메트릭스": {
            ...
            "numCopiedRows": "23",
            ...
            "numUpdatedRows": "27",
            ...
        },
        ...
    }
}
```

23개의 행이 새 데이터 파일에 복사되고 27개의 행이 업데이트된 것을 볼 수 있습니다.
총 50행.

따라서 쿼리를 다시 실행하면 배치에 정확히 50개의 행이 표시됩니다. 언제 우리가 “02 - AvailableNow Streaming” 노트북을 다시 실행하면 50개의 행이 표시됩니다.

```
{  
    ...  
    "배치 ID": 4,  
    "numInputRows": 50,
```

돌아가서 스트리밍 쿼리를 다시 실행하면 다음 오류가 표시됩니다.

스트리밍이 중지되었습니다...
com.databricks.sql.transaction.tahoe.DeltaUnsupportedOperationException: 버전 3의 소스 테이블에
서 데이터 업데이트(예: part-00000-....snappy.parquet)를 감지했습니다. 이는 현재 지원되지 않습니다. 업데이트를 무시하려면
'ignoreChanges' 옵션을 'true'로 설정하세요. 데이터 업데이트를 반영하려면 새로운 체크포인트 디렉터리로 이 쿼리를 다시
시작하세요. 소스 테이블은 dbfs:/mnt/.../LimitedRecords.delta 경로에서 찾을 수 있습니다.

여기에서 Delta Lake는 스트림의 데이터 업데이트가 현재 제공되지 않음을 알려줍니다. 지원됩니다. 우리가 정말로 새로운 기록만을 원하고 변화는 원하지 않는다는 것을 안다면, `readStream`에 `.option("ignoreChanges", "True")` 옵션을 추가합니다.

```
# 소스 "LimitedRecords" 테이블에서 스트리밍을 시작합니다.
# 이제 "read" 대신 "readStream"을 사용합니다.
# 나머지 부분에 대해 우리의 진술은 Delta가 읽은 다른 스파크와 같습니다

# 변경사항만 수신하려면 ignoreChanges 옵션의 주석을 해제하세요.
#개의 새 레코드, 업데이트된 레코드 없음
stream_df = 스파크  
\\
\\
\\
크 .readStream .option("ignoreChanges",
True).format("delta").load("/")
mnt/datalake/book/chapter08/LimitedRecords.delta")
```

이제 스트리밍 쿼리를 다시 실행하면 성공할 것입니다. 그러나 원시 데이터를 보면 여전히 50개의 입력 행이 모두 표시되는데 이는 잘못된 것 같습니다.

```
{
  "id": "d89a5c02-052b-436c-a372-2445fb8d88d6", "runId": "b1304246-4083-4275-8637-1f99768b8e03", "name": null,
  "timestamp": "2023-04-13T17:28:31.380Z", "batchId": 3,
  "numInputRows": 50, "inputRowsPerSecond": 0.0}
```

이 동작은 정상입니다. `ignoreChanges` 옵션은 여전히 Delta 테이블의 모든 다시 작성된 파일을 스트림으로 내보냅니다. 이는 일반적으로 변경된 모든 레코드의 상위 집합입니다.
그러나 실제로는 삽입된 레코드만 처리됩니다.

StreamingQuery 클래스

`streamQuery` 변수의 유형을 살펴보겠습니다.

```
# streamQuery 변수 print(type(streamQuery))
의 유형을 # 살펴보겠습니다.
```

산출:

```
<클래스 'pyspark.sql.streaming.query.StreamingQuery'>
```

유형이 `StreamingQuery`임을 알 수 있습니다. `streamQuery`의 상태 속성을 호출하면 다음을 얻습니다.

```
# 마지막 StreamingQuery print(streamQuery.status)의 상태를 출력합니다.
```

산출:

```
{'message': '중지됨', 'isDataAvailable': False, 'isTriggerActive': False}
```

쿼리가 현재 중지되었으며 사용 가능한 데이터가 없습니다. 활성화된 트리거가 없습니다.
또 다른 흥미로운 속성은 `RecentProgress`입니다. 이는 노트북의 스트리밍 출력에서 원시 데이터 섹션과 동일한 출력을 인쇄합니다. 예를 들어, 입력 행 수를 보려면 다음을 인쇄하면 됩니다.

```
print(streamQuery.recentProgress[0]["numInputRows"])
```

산출:

```
50
```

이 객체에는 몇 가지 흥미로운 메서드도 있습니다. 예를 들어 스트리밍이 종료될 때까지 기다리면 `waitTermination()` 메서드를 사용할 수 있습니다.

소스 레코드 전체 또는 일부 재처리 소

스 테이블에서 여러 배치를 처리하면서 체크포인트 파일은 이러한 모든 변경 사항을 체계적으로 구축해 왔습니다. 체크포인트 파일을 삭제하고 스트리밍 쿼리를 다시 실행하면 소스 테이블의 맨 처음부터 시작하여 모든 레코드를 가져옵니다.

```
%$st
# 체크포인트를 재설정하려면 이 줄의 주석 처리를 제거하십시오. rm -r /dbfs/mnt/datalake/
book/chapter08/StreamingTarget/_checkpoint
```

스트리밍 쿼리의 출력:

```
{
...
"numInputRows": 50,
...
"stateOperators": [], "소스": [
{
    ...
    "설명": "DeltaSource[dbfs:/mnt/.../LimitedRecords.delta]", "startOffset": null, "endOffset": null
    ...
},
    ...
    ],
    "latestOffset": null,
    "numInputRows": 50,
```

소스 테이블의 모든 행을 읽습니다. 오프셋 null에서 시작하여 저장소에서 끝났습니다.

버전 5.

변경 사항의 일부만 스트리밍할 수도 있습니다. 이를 위해 체크포인트를 다시 지운 후 readStream에 시작 버전을 지정할 수 있습니다.

```
stream_df = 스파
            \
            \
            \
            .readStream .option("ignoreChanges",
True) .option("startingVersion",
3) .format("delta") .load("/")
mnt/datalake/book/chapter08/LimitedRecords.delta")
```

원시 데이터를 보면 다음과 같은 결과를 얻습니다.

```
{
...
"batchId": 0,
"numInputRows": 70,
"inputRowAnd": 0.0,
...
"stateOperators": [], "소스": [ {
```

```

...
  "startOffset": null, "endOffset": null,
  { "sourceVersion": 1,
    "reservoirId": "32c71d93-ca81-4d6e-9928-c1a095183016", "reservoirVersion": 6, "index": -1,
    "isStartingVersion": 거짓 },

```

우리는 70개의 행을 얻습니다. 버전 3부터 시작했기 때문에 이는 정확하지 않습니다. 지금까지 작업한 버전과 작업을 요약한 [표 8-1](#)을 살펴보겠습니다.

표 8-1. 버전별 레코드 수

버전 영향을 받는 행 수 작업에서		
5	50	업데이트
4	10	끼워 넣다
상	10	끼워 넣다
총	70	

이는 스트리밍 쿼리에 대한 총 입력 행 수의 유효성을 검사합니다. StartingVersion을 설정하면 DESCRIBE HISTORY 명령과 결합할 때 많은 옵션이 제공됩니다. 기록을 보고 데이터를 로드할 시점을 결정할 수 있습니다.

변경 데이터 피드에서 스트림 읽기 [6 장](#)에서는 Delta Lake가 CDF를 통해 테이블에 기록된 모든 데이터에 대해 "변경 이벤트"를 기록하는 방법에 대해 읽었습니다. 이러한 변경 사항은 다운스트림 소비자에게 전송될 수 있습니다. 이러한 다운스트림 소비자는 `.readStream()`이 포함된 스트리밍 쿼리를 사용하여 CDF에서 캡처 및 전송된 변경 이벤트를 읽을 수 있습니다.

CDF가 활성화된 테이블을 읽는 동안 CDF에서 변경 사항을 가져오려면 `readChangeFeed` 옵션을 `true`로 설정하세요. `.readStream()`과 함께 `readChangeFeed`를 `true`로 설정하면 소스 테이블에서 다운스트림 대상 테이블로 변경 사항을 효율적으로 스트리밍할 수 있습니다. 또한 `StartingVersion` 또는 `StartingTimestamp`을 사용하여 전체 테이블을 처리하지 않고 델타 테이블 스트리밍 소스의 시작점을 지정할 수도 있습니다.

```
# 버전 5 Spark.readStream 이후 readChangeFeed로 CDF 스트림을 읽습니다.
```

```

  \.format("delta") \.option("readChangeFeed",
  "true") \.option("startingVersion", 5)
  \.table("<delta_table_name>")

```

```
# 타임스탬프 2023-01-01 00:00:00 시작 이후 CDF 스트림을 읽습니다.
```

```
스파크.readStream
    .format("델타") \
    .option("readChangeFeed", "true") \
    .option("startingTimestamp", "2023-01-01 00:00:00").table("<delta_table_name>") \
```

.option("readChangeFeed", "true")를 사용하면 CDF와 함께 테이블 변경 사항이 반환됩니다.
`_change_type`, `_commit_timestamp` 및 `_commit_version`을 제공하는 스키마

readStream이 소비할 것 입니다 . 다음은 CDF 데이터의 예입니다.

6 장):

공급업체 ID	승객수	요금	<code>_change_type</code>	<code>_commit_timestamp</code>	<code>_commit_version</code>	업데이트_사전 이미지	업데이트_포스트이미지	커밋 버전
1	1000	2000						2
1	1000	2500						2
상	7000	10000				삭제		상
4	500	1000				끼워 넣다		4

변경 피드를 읽기 위한 이전 코드 조작은

StartingVersion 또는 StartingTimestamp. 이러한 방법에 유의하는 것이 중요합니다.

선택사항이며 제공되지 않은 경우 스트림은 테이블의 최신 스냅샷을 다음 위치에서 가져옵니다.

스트리밍 시간은 INSERT로 , 향후 변경 사항은 변경 데이터로 표시됩니다.



지정된 버전에서 스트리밍 소스를 시작하는 동안 또는
 타임스탬프가 가능하며 스트리밍과 관련된 스키마
 소스는 델타 테이블의 최신 스키마를 반영합니다. 중요해요 -
 호환되지 않는 스키마 변경 사항이 없는지 확인해야 합니다.
 지정된 버전 또는 타임스탬프를 따르는 델타 테이블입니다. 실패
 그렇게 하면 스트리밍 시 부정확한 결과가 발생할 수 있습니다.
 소스가 일치하지 않는 스키마가 있는 데이터를 검색합니다.

변경 데이터를 읽을 때 지정할 수 있는 다른 옵션이 있습니다.

데이터 변경 및 속도 제한(각 마이크로 배치에서 처리되는 데이터의 양)에 관한 것입니다. [표 8-2](#)

는 스트리밍 쿼리에 사용하기 위한 추가적이고 중요한 옵션을 강조합니다.

즉, 델타 테이블을 스트림 소스로 사용하는 경우입니다.

표 8-2. 추가 스트리밍 옵션

옵션	정의
최대파일수	모든 마이크로 배치에서 고려되는 새 파일 수를 제어합니다. 기본값은 1,000입니다.
방아쇠	
maxBytesPer	각 마이크로 배치에서 처리되는 데이터의 양을 제어합니다. Trigger.Once를 사용하는 경우 이 옵션은 무시됩니다. 이 옵션은 기본적으로 설정되어 있지 않습니다.
TriggerignoreDeletes	파티션 경계에서 데이터를 삭제하는 트랜잭션을 무시합니다.
ignoreChanges	UPDATE, MERGE INTO, DELETE (파티션 내) 또는 OVERWRITE와 같은 데이터 변경 작업으로 인해 소스 테이블에서 파일을 다시 작성해야 하는 경우 업데이트를 다시 처리합니다. ignoreChanges에는 ignoreDeletes도 포함되어 있습니다.

비율 제한 옵션은 전반적인 리소스 관리 및 활용도를 보다 효율적으로 제어하는 데 유용할 수 있습니다. 예를 들어, 새로운 데이터 파일이 유입되거나 처리할 데이터 양이 많을 때 잠재적으로 처리 리소스(예: 클러스터)에 과부하가 걸리는 것을 방지할 수 있습니다. 속도 제한을 제어하면 마이크로 배치 크기를 제어하여 보다 균형 잡힌 처리 경험을 달성하는 데 도움이 될 수 있습니다. 기존 스트리밍 쿼리가 중단되지 않도록 삭제를 무시하면서 속도 제한을 효과적으로 제어하려면 스트리밍 쿼리에서 다음 옵션을 지정할 수 있습니다.

```
# readChangeFeed로 CDF 스트림을 읽고 # 시작 타임스탬프나 버전을 지정하지 마세요. 비율 제한을 지정하고 삭제를 무시합니다. 스파크.readStream
    \
    \
    .format("delta") .option("maxFilesPerTrigger",
    50) .option("maxBytesPerTrigger",
    "10MB") .option("ignoreDeletes",
    "true") .option("readChangeFeed", "true"). 테이
    블("delta_table_name")
```

이 예에서는 속도 제한 옵션을 설정하고, 삭제를 무시하고, 시작 타임스탬프 및 버전 옵션을 생략합니다. 이렇게 하면 최신 버전의 테이블을 읽고(버전이나 타임스탬프가 지정되지 않았으므로) 마이크로 배치 크기와 처리 리소스를 더 효과적으로 제어하여 스트리밍 쿼리에 대한 잠재적인 중단을 줄일 수 있습니다.

결론

Delta Lake의 주요 기능 중 하나는 일괄 처리 및 스트리밍 데이터를 단일 테이블로 통합하는 것입니다. 이 장에서는 Delta Lake가 구조적 스트리밍과 완전히 통합되는 방법과 Delta 테이블이 연속 데이터 스트림의 확장 가능하고 내결합성이 있으며 지연 시간이 짧은 처리를 지원하는 방법에 대한 세부 사항과 예를 자세히 살펴보았습니다.

readStream 및 writeStream을 통해 구조적 스트리밍과 통합된 Delta 테이블은 스트리밍 소스와 대상 모두로 사용할 수 있으며 스트리밍 DataFrame을 활용할 수 있습니다. 이 장의 예제에서는 이러한 스트리밍 DataFrame의 변경 사항을 읽고 간단한 처리를 수행하여 대상에 스트림을 쓰는 방법을 살펴보았습니다. 그런 다음 체크포인트 파일과 메타데이터, 쿼리 프로세스로 그, 스트리밍 클래스를 탐색하여 스트리밍 작동 방식을 더 잘 이해하고 내부적으로 정보를 추적했습니다. 마지막으로 readStream 과 함께 CDF를 활용하여 스트리밍 쿼리에서 행 수준 변경 사항을 전송하고 읽는 방법을 배웠습니다.

배치 데이터와 스트리밍 데이터를 단일 델타 테이블로 통합한 후 [9장에서는](#) 이 데이터를 다른 조직과 안전하게 공유하는 방법을 살펴보겠습니다.

제9장

델타 공유

오늘날 경제의 데이터 중심적 특성으로 인해 조직과 고객, 공급업체 및 파트너 간의 광범위한 데이터 교환이 필요합니다. 효율성과 즉각적인 접근성이 중요하지만 보안 문제와 충돌하는 경우가 많습니다.

조직이 디지털 경제에서 성공하려면 데이터 공유에 대한 개방적이고 안전한 접근 방식이 필요합니다.

조직 내에서 내부적으로 데이터 공유가 필요한 경우가 많습니다. 조직은 로컬 클라우드 솔루션을 사용하여 지리적으로 분산되어 있습니다. 이러한 회사는 소유권이 분산되고 데이터 관리가 분산 및 통합되는 데이터 메시 아키텍처를 구현하려고 하는 경우가 많습니다. 효율적이고 안전한 데이터 공유는 조직 전체에서 데이터 제품을 효율적으로 공유하는 데 중요한 요소입니다.

기업 전체의 다양한 비즈니스 그룹은 중요한 비즈니스 결정을 내리기 위해 데이터에 액세스해야 합니다. 데이터 팀은 솔루션을 통합하여 비즈니스에 대한 포괄적인 전사적 관점을 구축하기를 원합니다.

기존의 데이터 공유 방법

과거에는 다양한 플랫폼, 기업, 클라우드 전반에서 데이터를 공유하는 것이 항상 복잡한 과제였습니다. 조직에서는 보안 위험, 경쟁, 데이터 공유 솔루션 구현과 관련된 상당한 비용에 대한 우려로 인해 데이터 공유를 꺼려했습니다.

기존의 데이터 공유 기술은 여러 클라우드 환경과의 호환성, 개방형 형식 지원 등 최신 요구 사항을 충족하면서도 필요한 성능을 제공하는 데 어려움을 겪고 있습니다. 많은 데이터 공유 솔루션이 특정 공급업체에 묶여 있어 호환되지 않는 플랫폼에서 작동하는 데이터 공급자와 소비자에게 문제를 야기합니다.

데이터 공유 솔루션은 레거시 및 자체 개발(맞춤형) 솔루션, 최신 클라우드 개체 스토리지, 독점 상용 솔루션이라는 세 가지 형식으로 개발되었습니다. 이러한 각 접근 방식에는 장단점이 있습니다.

레거시 및 자체 개발 솔루션 조직은 [그림 9-1](#)에

표시된 것처럼 이메일, SFTP 또는 사용자 정의 API와 같은 레거시 기술을 기반으로 데이터 공유 솔루션을 구현하기 위해 자체 시스템을 구축했습니다.

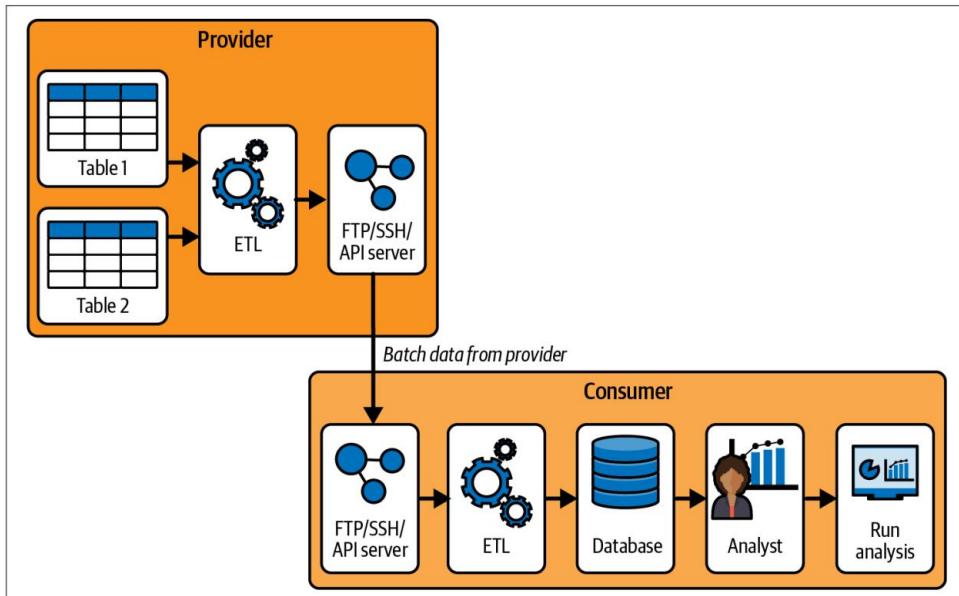


그림 9-1. 데이터 공유를 위한 자체 개발 솔루션

이러한 솔루션의 장점:

공급업체에 구애

받지 않는 FTP, 이메일 및 API는 잘 문서화된 프로토콜이므로 데이터 소비자가 다양한 클라우드 인프라를 활용하여 제공된 데이터에 액세스할 수 있습니다.

유연성 많

맞춤형 솔루션은 오픈 소스 기술을 기반으로 하므로 온프레미스와 클라우드 환경 모두에서 작동할 수 있습니다.

이러한 솔루션의 단점:

데이터 이동

클라우드 스토리지에서 데이터를 추출하고, 변환하고, 다양한 수신자를 위해 FTP 서버에서 호스팅하려면 상당한 노력이 필요합니다. 또한 이 접근 방식은 데이터 중복으로 이어져 조직이 실시간 데이터에 즉시 액세스하는 것을 방해합니다.

데이터 공유의 복잡성

맞춤형 솔루션에는 복제 및 프로비저닝으로 인해 복잡한 아키텍처가 포함되는 경우가 많습니다. 이러한 복잡성으로 인해 데이터 공유 활동에 상당한 시간이 추가되고 최종 소비자에게 오래된 데이터가 제공될 수 있습니다.

데이터 수신자를 위한 운영 오버헤드

데이터 수신자는 특정 사용 사례에 대해 데이터 추출, 변환 및 로드(ETL)를 수행해야 하므로 통찰력을 얻는 시간이 더욱 지연됩니다.

공급자가 데이터를 업데이트할 때마다 소비자는 ETL 파이프라인을 반복적으로 다시 실행해야 합니다.

보안 및 거버넌스 최신 데이터 요

구 사항이 더욱 엄격해짐에 따라 자체 기술과 레거시 기술을 보호하고 관리하는 것이 점점 더 어려워지고 있습니다.

확장성 이슈

한 솔루션을 관리하고 유지하는 데는 비용이 많이 들고 대규모 데이터 세트를 수용할 수 있는 확장성이 부족합니다.

독점 공급업체 솔루션 상용 데이터 공유 솔루션

은 사내 솔루션 구축에 대한 대안을 모색하는 기업에서 널리 선택됩니다. 이러한 솔루션은 [그림 9-2](#)에 표시된 것처럼 독점 솔루션 개발에 많은 시간과 리소스를 할당하고 싶지 않은 것과 클라우드 개체 스토리지가 제공할 수 있는 것보다 더 큰 제어를 원하는 것 사이의 균형을 제공합니다.

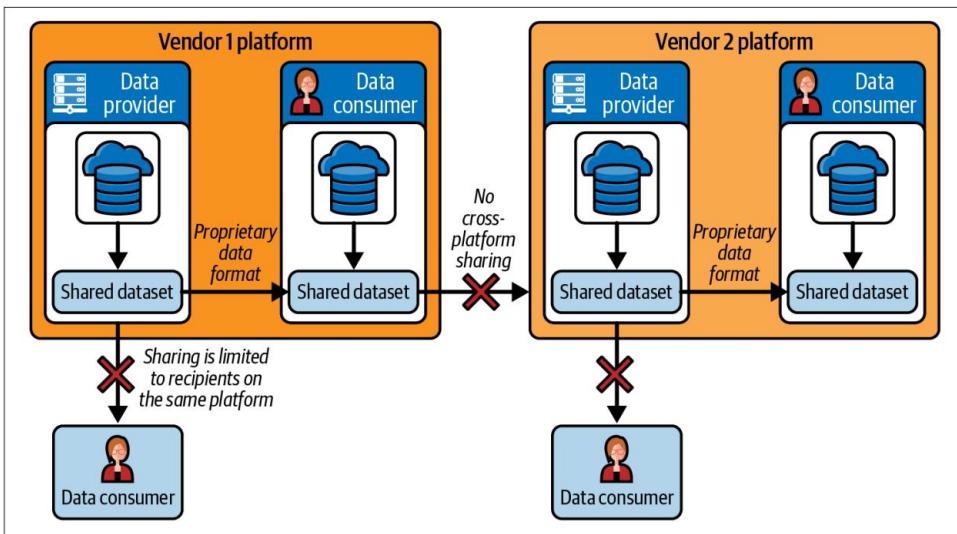


그림 9-2. 독점 벤더 데이터 공유 솔루션

이 솔루션의 장점:

단순성 상

이 솔루션은 사용자가 동일한 플랫폼에서 다른 사람들과 데이터를 공유할 수 있는 쉬운 방법을 제공합니다.

이 솔루션의 단점:

공급업체 종속

상용 솔루션은 종종 다른 플랫폼과의 상호 운용성이 부족하여 경쟁 솔루션 사용자와 데이터를 공유하기가 어렵습니다. 이러한 제한으로 인해 데이터 접근성이 떨어지고 공급업체에 종속되는 결과가 발생합니다. 또한 데이터 제공자와 수신자 간의 플랫폼 차이로 인해 데이터 공유가 복잡해집니다.

데이터 이동 데이터는 특

정 플랫폼에 로드되어야 하며, 여기에는 ETL 및 데이터 복사본 생성과 같은 추가 단계가 포함됩니다.

확장성 상용

데이터 공유 솔루션에는 공급업체가 부과하는 확장에 제한이 있을 수 있습니다.

비용

데이터 공급자는 다양한 클라우드 플랫폼에서 다양한 수신자에 대한 데이터를 복제해야 하기 때문에 앞서 언급한 문제로 인해 잠재 고객과 데이터를 공유하는 데 추가 비용이 발생합니다.

Cloud Object Storage 객

체 스토리지는 탄력적인 특성과 원활한 확장성을 갖추고 있어 방대한 양의 데이터를 처리하고 무제한적인 성장을 손쉽게 수용할 수 있어 클라우드 환경에 적합한 솔루션으로 높은 평가를 받고 있습니다. Amazon S3, ADLS(Azure Data Lake Storage), GCS(Google Cloud Storage)와 같은 주요 클라우드 제공업체는 비용 효율적인 객체 스토리지 서비스를 제공하고 탁월한 확장성과 안정성을 제공합니다.

클라우드 객체 스토리지의 주목할만한 기능 중 하나는 서명된 URL을 생성하는 기능입니다. 이러한 URL은 특정 객체를 다운로드할 수 있는 시간 제한 권한을 제공합니다. 미리 서명된 URL을 공유함으로써 이를 소유한 사람은 누구나 지정된 객체에 편리하게 접근할 수 있어 효율적인 데이터 공유가 가능해집니다.

이 솔루션의 장점:

데이터를 제자리에서 공유

객체 스토리지를 제자리에서 공유할 수 있으므로 소비자는 사용 가능한 최신 데이터에 액세스할 수 있습니다.

확장성 클

라우드 객체 스토리지는 일반적으로 온프레미스에서 달성할 수 없는 가용성 및 내구성 보장으로 이익을 얻습니다. 데이터 소비자는 클라우드 공급자로부터 직접 데이터를 검색하여 공급자의 대역폭을 절약합니다.

이 솔루션의 단점:

단일 클라우드 제공업체로 제한됨

객체에 액세스하려면 수신자가 동일한 클라우드에 있어야 합니다.

번거로운 보안 및 거버넌스

권한 할당 및 액세스 관리와 관련된 복잡성이 있습니다.

서명된 URL을 생성하려면 사용자 정의 애플리케이션 논리가 필요합니다.

복잡성 데이터 공

유를 관리하는 페르소나(데이터베이스 관리자, 분석가)는 ID 및 액세스 관리 정책과 데이터가 기본 파일에 매핑되는 방식을 이해하는 데 어려움을 겪습니다. 대용량 데이터를 보유한 기업의 경우 클라우드 스토리지를 통한 공유는 시간이 많이 걸리고 번거로우며 확장이 거의 불가능합니다.

데이터 수신자를 위한 운영 오버헤드

데이터 수신자는 최종 사용 사례를 위해 원시 파일을 사용하기 전에 원시 파일에 대한 ETL 파이프라인을 수행해야 합니다.

포괄적인 솔루션이 부족하면 데이터 제공자와 소비자가 데이터를 쉽게 공유하는 데 어려움을 겪게 됩니다. 번거롭고 불완전한 데이터 공유는 공유 데이터를 통한 비즈니스 기회 개발을 제한하기도 합니다.

오픈 소스 델타 공유

독점 솔루션과 달리 오픈 소스 데이터 공유는 불필요한 제한과 재정적 부담을 초래하는 공급업체별 기술과 관련이 없습니다. 오픈 소스 Delta Sharing은 대규모로 데이터를 공유해야 하는 모든 사람이 쉽게 사용할 수 있습니다.

델타 공유 목표

Delta Sharing은 다음과 같은 목적으로 설계된 오픈 소스 프로토콜입니다.

개방형 크로스 플랫폼 데이터 공유 Delta

Sharing은 공급업체 종속을 방지하는 오픈 소스 크로스 플랫폼 솔루션을 제공합니다. 온프레미스나 다른 클라우드 등 모든 플랫폼과 Delta Lake 및 Apache Parquet 형식의 데이터 공유가 가능합니다.

데이터 이동 없이 실시간 데이터 공유

데이터 수신자는 데이터를 복제하지 않고도 Delta Sharing에 직접 연결할 수 있습니다. 이 기능을 사용하면 불필요한 데이터 복제나 이동 없이 기존 데이터를 실시간으로 쉽게 공유할 수 있습니다.

다양한 클라이언트 지원

Delta Sharing은 Power BI, Tableau, Apache Spark, Pandas 및 Java와 같은 널리 사용되는 도구를 포함하여 다양한 클라이언트를 지원합니다. 비즈니스 인텔리전스, 머신러닝, AI 등 다양한 사용 사례에 맞게 선택한 도구를 사용하여 데이터를 소비할 수 있는 유연성을 제공합니다. 델타 공유 커넥터를 구현하는 것은 빠르고 간단합니다.

중앙 집중식 거버넌스 Delta

Sharing은 강력한 보안, 감사 및 거버넌스 기능을 제공합니다.

데이터 공급자는 데이터 액세스를 세부적으로 제어하여 전체 테이블이나 테이블의 특정 버전 또는 파티션을 공유할 수 있습니다. 공유 데이터에 대한 액세스는 단일 시점에서 관리 및 감사되므로 중앙 집중식 제어 및 규정 준수가 보장됩니다.

대규모 데이터 세트를 위한 확장성

Delta Sharing은 대규모 구조화된 데이터 세트를 처리하도록 설계되었으며 구조화되지 않은 데이터와 함께 학습 모델, 대시보드, 노트북 및 표 형식 데이터와 같은 미래 데이터 파생물의 공유를 지원합니다. Delta Sharing을 사용하면 클라우드 스토리지 시스템의 비용 효율성과 확장성을 활용하여 대규모 데이터 세트를 경제적이고 안정적으로 공유할 수 있습니다.

내부적으로 델타 공유

Delta Sharing은 클라우드 데이터 세트의 특정 부분에 대한 보안 액세스를 가능하게 하는 REST API 엔드포인트를 정의하는 개방형 프로토콜입니다. Amazon S3, ADLS 또는 GCS와 같은 최신 클라우드 스토리지 시스템의 기능을 활용하여 대규모 데이터 세트의 안정적인 전송을 보장합니다. 이 프로세스에는 그림 9-3에 설명된 대로 데이터 제공자와 수신자라는 두 가지 주요 당사자가 포함됩니다.

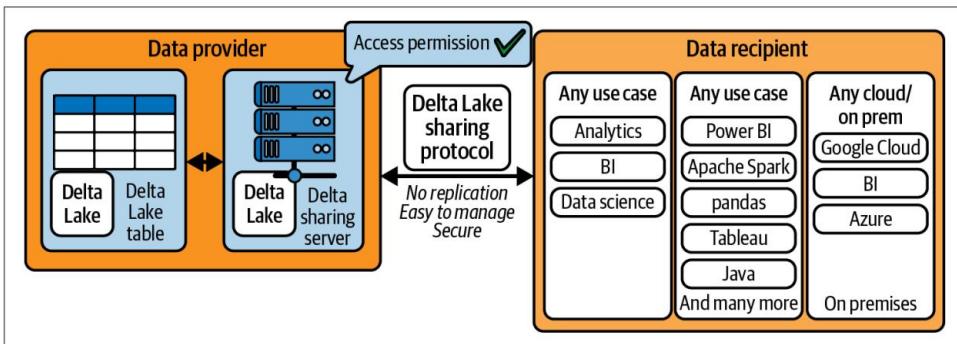


그림 9-3. 델타 공유 프로토콜 개요

데이터 공급자 및 수신자 데이터 공급자로서

Delta Sharing을 사용하면 클라우드 데이터 레이크에 Delta Lake 형식으로 저장된 기존 테이블 또는 그 일부(예: 파티션의 특정 테이블 버전)를 공유할 수 있습니다. 데이터 공급자는 공유할 데이터를 결정하고 그 앞에 델타 공유 프로토콜을 구현하고 수신자의 액세스를 관리하는 공유 서버를 실행합니다. 오픈 소스 Delta Lake에는 참조 공유 서버가 포함되어 있으며 Databricks는 해당 플랫폼에 대한 서버를 제공합니다. 다른 공급업체도 곧 뒤따를 것으로 예상됩니다.

데이터 수신자로서 프로토콜을 지원하는 많은 델타 공유 클라이언트 중 하나만 필요합니다. 오픈 소스 Delta Lake는 Pandas, Apache Spark, Rust 및 Python용 오픈 소스 커넥터를 출시했으며 더 많은 클라이언트에서 파트너와 협력하고 있습니다.

실제 거래소는 클라우드 스토리지 시스템과 Delta Lake의 기능을 활용하여 효율적이도록 세심하게 설계되었습니다. 델타 공유 프로토콜은 다음과 같이 작동합니다(그림 9-4 참조).

1. 수신자의 클라이언트는 (베어러 토큰 또는 기타 방법을 통해) 공유 서버에 인증하고 특정 테이블에 대한 쿼리를 요청합니다. 클라이언트는 데이터의 하위 집합만 읽기 위한 힌트로 데이터에 대한 필터(예: "국가 = 미국")를 제공할 수도 있습니다.
2. 서버는 클라이언트가 데이터에 액세스할 수 있는지 확인하고 요청을 기록한 다음 다시 보낼 데이터를 결정합니다. 이는 테이블을 구성하는 ALDS(Azure), S3(AWS) 또는 GCS(GCP)의 데이터 개체의 하위 집합입니다.

3. 데이터를 전송하기 위해 서버는 클라이언트가 클라우드 제공자로부터 직접 Parquet 파일을 읽을 수 있도록 하는 단기 사전 서명된 URL을 생성하므로 전송은 공유 서버를 통해 스트리밍하지 않고 대규모 대역폭을 사용하여 병렬로 발생할 수 있습니다. 모든 주요 클라우드에서 사용할 수 있는 이 강력한 기능을 사용하면 매우 큰 데이터 세트를 빠르고 저렴하며 안정적으로 공유할 수 있습니다.

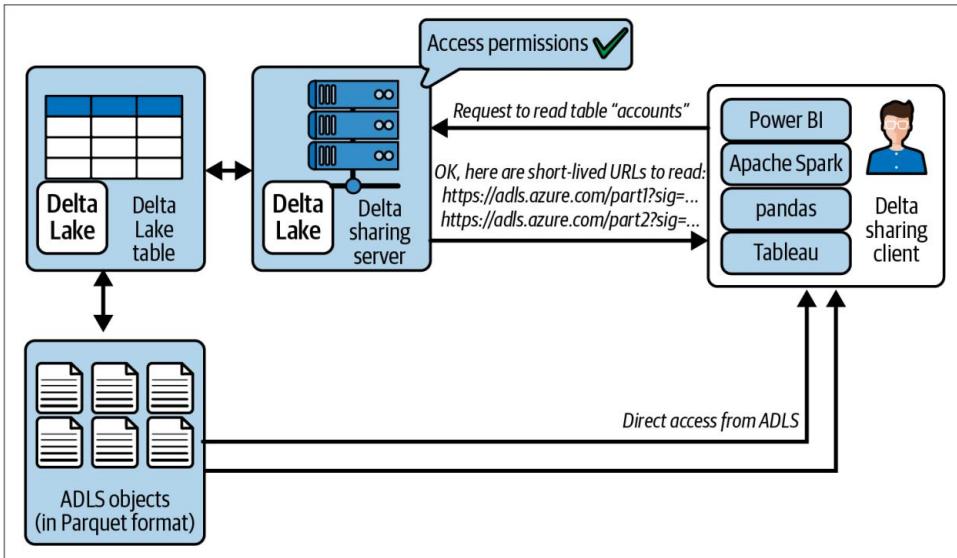


그림 9-4. 델타 공유 프로토콜 세부정보

디자인의 이점

Delta Sharing 설계는 공급자와 소비자 모두에게 다음과 같은 많은 이점을 제공합니다.

- 데이터 공급자는 전체 테이블을 쉽게 공유하거나 테이블의 파티션 버전만 공유할 수 있습니다. 클라이언트에게는 테이블에 포함된 개체의 특정 하위 집합에 대한 액세스 권한만 부여되기 때문입니다.
- 데이터 공급자는 Delta Lake의 ACID 트랜잭션을 사용하여 실시간으로 안정적으로 데이터를 업데이트할 수 있으며 수신자는 항상 일관된 보기만 볼 수 있습니다.
- 데이터 수신자는 제공자와 동일한 플랫폼에 있을 필요가 없으며 클라우드에 있을 필요도 없습니다. 클라우드 전체는 물론 클라우드에서 온프레미스까지 작업을 공유할 수 있습니다.
- 고객이 이미 Parquet를 활용하고 있다면 Delta Sharing 프로토콜을 간단하게 구현할 수 있습니다.
- 기본 클라우드 시스템을 사용한 데이터 전송은 빠르고 저렴하며 안정적이며 병렬화 가능.

델타 공유 저장소

델타 공유 GitHub 리포지토리를 온라인에서 찾을 수 있습니다. 여기에는 다음 구성 요소가 포함되어 있습니다.

- 델타 공유 프로토콜 [사양](#). • Python 커넥터. 이것은 Delta Sharing을 구현하는 Python 라이브러리입니다. 공유 테이블을 pandas 또는 PySpark DataFrames로 읽는 프로토콜입니다.
- Apache Spark 커넥터. 이 커넥터는 델타 공유 서버에서 공유 테이블을 읽기 위해 델타 공유 프로토콜을 구현합니다. 그런 다음 SQL, Python, Scala, Java 또는 R을 사용하여 테이블에 액세스할 수 있습니다.
- [참조 구현](#) 델타 공유 서버의 델타 공유 프로토콜. 사용자는 이 서버를 배포하여 Azure, AWS 또는 GCP 스토리지 시스템에서 Delta Lake 및 Parquet 형식의 기존 테이블을 공유할 수 있습니다.

다음으로 Python 커넥터를 사용하여 [delta-io](#)에서 호스팅하는 예제 Delta Sharing Server의 Delta 테이블에 액세스해 보겠습니다.

1단계: Python 커넥터 설치 Python 커넥터는 delta-

sharing이라는 PyPi 라이브러리로 제공되므로 [그림 9-5](#)와 같이 이 라이브러리를 클러스터에 추가하기만 하면 됩니다.

	Status	Name	Type
<input checked="" type="checkbox"/>		delta-sharing	PyPI

그림 9-5. 클러스터에 델타 공유 PyPi 라이브러리 설치

2단계: 프로필 파일 설치 Python 커넥터는 프로

필 파일을 기반으로 공유 테이블에 액세스합니다. [프로필 파일](#)을 다운로드할 수 있습니다. 예제 델타 공유 서버의 경우 링크를 따라가세요. 이 파일은 open-datasets.share라는 파일로 다운로드됩니다. 이는 서버에 대한 자격 증명이 포함된 간단한 JSON 파일입니다(이 예의 전달자 토큰은 난독화되어 있음).

```
{
  "shareCredentialsVersion": 1, "앤프로인트":
  "https://sharing.delta.io/delta-sharing/", "bearerToken":
  "faaixxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
}
```

dbfs를 사용하여 Databricks 파일 시스템의 dbfs:/ 위치에 공유 파일을 업로드합니다.

cp 명령:

```
C:\Users\bhael\Downloads>dbfs cp open-datasets.share dbfs:/mnt/.../delta-sharing/
```

```
C:\Users\bhael\Downloads>dbfs ls dbfs:/mnt/datalake/book/delta-sharing
```

오픈 데이터셋.share

```
C:\Users\bhael\다운로드>
```

3단계: 공유 테이블 읽기

그런 다음 "01 - 공유 예" 노트북에서 파일을 참조할 수 있습니다.

```
# 프로필 파일을 가리킵니다. 로컬에 있는 파일일 수 있습니다.  
# 파일 시스템 또는 원격 파일 시스템. 이 경우, 우리는  
# dbfs에 파일을 업로드했습니다.  
profile_path = "/dbfs/mnt/datalake/book/delta-sharing/open-datasets.share"
```



공유 테이블에 액세스하는 방법에 따라 다음을 수행해야 합니다.
다른 경로 구문을 사용하십시오. 여기에 지정된 profile_path 는
Pandas DataFrame으로 테이블에 액세스할 때 작동합니다. 만약 너라면
Spark를 사용하여 테이블에 액세스하려면 dbfs:/ 를 사용해야 합니다.
/dbfs/ 포트 대신 구문 .

다음으로 SharingClient를 생성하여 프로필 경로를 전달하고 모든 공유 항목을 나열할 수 있습니다.
델타 테이블:

```
# SharingClient를 생성하고 모든 공유 테이블을 나열합니다.  
클라이언트 = delta_sharing.SharingClient(profile_path)  
클라이언트.목록_모든_테이블()
```

그리면 다음과 같은 출력이 생성됩니다.

```
Out[22]: [테이블(이름='COVID_19_NYT', 공유='delta_sharing', 스키마='기본'), 테이블(이  
름='boston-housing', 공유='delta_sharing', 스키마='기본'), 테이블(이름='flight-asa_2008', 공유  
='delta_sharing', 스키마='기본'), 테이블(이름='lending_club', 공유='delta_sharing', 스키마='기본'), 테이블  
(이름='nyctaxi_2019', 공유='delta_sharing', 스키마='기본값'), 테이블(이름='nyctaxi_2019_part', 공유  
='delta_sharing', 스키마='기본'), 테이블(이름='owid-covid-data', 공유='delta_sharing', 스키마='기본  
값')]
```

공유 테이블에 대한 URL을 생성하려면 다음 구문을 사용합니다.

```
<프로필 파일 기본 이름>#<공유 이름>.<스키마 이름>.<테이블 이름>
```

이제 URL을 작성하고 공유 Delta 테이블의 내용을 pandas DataFrame으로 읽을 수 있습니다.

```
# 공유 테이블에 액세스하기 위한 URL을 생성합니다. # 테이블 경로는
('<share-name>.<schema_name>.<table_name>') 뒤에 오는 프로필 경로입니다.

# 여기서는 테이블을 팬더 DataFrame으로 로드합니다. table_url = profile_path +
"#delta_sharing.default.boston-housing" df = delta_sharing.load_as_pandas(table_url,limit=10) df.head()
```

출력(관련 부분만 표시):

```
+-----+-----+-----+-----+
|ID|크림|zn|indus|차|녹스|RM|
+-----+-----+-----+-----+
|1|0.00632|18|2.31|0|0.538|6.575|2|0.02731|0|7.0|0|0.469|
6.421|4|0.03237|0|2.18|0|0.458|6.998|5|0.06905|0|2.18|0|
0.458|7.147|7|0.08829|12.5|7.87|0|0.524|6.012|
+-----+
```

테이블을 표준 PySpark DataFrame으로 로드하려면 load_as_spark () 메서드를 사용할 수 있습니다.

```
# Spark를 사용하여 공유 테이블에 액세스할 수도 있습니다. 여기서는 # dbfs:/ 경로 접두사를 사용해야 합니다. profile_path_spark = "dbfs:/
mnt/datalake/book/delta-sharing/open-
datasets.share" table_url_spark = profile_path_spark + "#delta_sharing.default.boston-housing"

df_spark = delta_sharing.load_as_spark(table_url_spark) 표시(df_spark.limit(5))
```

앞에서 설명한 것처럼 URL이 약간 변경된 것을 확인하세요. 그러면 pandas 예제와 동일한 출력이 생성됩니다.

결론

오픈 소스 기술을 사용하여 데이터 교환을 활성화하면 내부 및 외부 사용 모두에 많은 이점이 제공됩니다. 첫째, 상당한 유연성을 제공하여 팀이 특정 비즈니스 사용 사례 및 요구 사항을 충족하도록 데이터 교환 프로세스를 맞춤화할 수 있습니다. 활발한 오픈 소스 커뮤니티의 지원을 통해 지속적인 개선, 버그 수정, 방대한 지식에 대한 액세스가 보장되며 팀과 비즈니스 사용자가 데이터 공유 방식의 최전선에 머물 수 있도록 지원합니다.

데이터 제공자와 데이터 수신자를 위한 델타 공유 사용의 주요 이점 중에서 가장 중요한 것은 다음과 같습니다.

- 끊임없이 증가하는 데이터 세트와 수요가 높은 사용 사례를 다루는 데이터 팀에게는 확장성이 매우 중요합니다.
- 상호 운용성은 또 다른 중요한 이점입니다. 오픈 소스 기술인 Delta Sharing은 데이터 생태계의 다른 구성 요소와 조화롭게 작동하여 원활한 통합을 촉진하도록 설계되었습니다.
- 또한 Delta Sharing 소스 코드를 검토할 수 있으므로 독점 솔루션에 비해 투명성과 보안이 향상됩니다. 이를 통해 더 강력한 보안 조치와 식별된 취약점에 대응하고 사전 대응할 수 있는 능력이 가능해집니다.

Delta Sharing을 사용하면 팀은 새로운 아키텍처에 적응하는 데 투자할 필요 없이 도구나 공급업체 간에 자유롭게 전환할 수 있으므로 공급업체 종속을 피할 수 있습니다. 오픈 소스 커뮤니티의 빠른 혁신 속도를 통해 팀은 최첨단 기능을 수용하고 데이터 관리 및 분석의 새로운 추세에 빠르게 적응할 수 있습니다.

Delta Sharing을 사용하여 데이터를 공유하는 기능은 끊임없이 변화하는 환경과 데이터 환경에서 조직의 성공을 위한 더 나은 데이터 기반 솔루션과 통찰력을 제공함으로써 보다 민첩하고 비용 효율적이며 혁신적인 데이터 생태계를 가능하게 합니다.

지금까지 배운 기본 구성 요소를 바탕으로 [10장](#) 에서는 완전한 데이터 레이크하우스를 구축하는 방법에 대해 자세히 알아보겠습니다.

제10장

델타 호수에 레이크하우스 짓기

1장에서는 전통적인 데이터 웨어하우스와 데이터 레이크의 최고의 요소를 결합한 데이터 레이크하우스의 개념을 소개했습니다. 이 책 전반에 걸쳐 레이크하우스 아키텍처를 활성화하는 데 도움이 되는 5가지 주요 기능인 스토리지 계층, 데이터 관리, SQL 분석, 데이터 과학 및 기계 학습, 메달리온 아키텍처에 대해 배웠습니다.

Delta Lake에 레이크하우스를 건설하기 전에 업계의 데이터 관리 및 분석 발전을 빠르게 검토해 보겠습니다.

데이터 웨어하우스 1970

년대부터 2000년대 초반까지 데이터 웨어하우스는 데이터를 수집하고 비즈니스 컨텍스트에 통합하여 비즈니스 인텔리전스 및 분석을 지원하도록 설계되었습니다. 데이터 양이 증가함에 따라 속도, 다양성 및 진실성도 증가했습니다. 데이터 웨어하우스는 유연하고 통합적이며 비용 효율적인 방식으로 이러한 요구 사항을 해결하는 데 어려움을 겪었습니다.

데이터 레이크

2000년대 초반 데이터 양의 증가로 인해 데이터 레이크(처음에는 Hadoop을 사용하는 온프레미스, 나중에는 클라우드를 사용)가 개발되었습니다. 이는 모든 규모의 데이터 형식을 저장할 수 있는 비용 효율적인 중앙 저장소입니다. 그러나 이점이 추가되 었음에도 불구하고 추가적인 과제가 있었습니다. 데이터 레이크는 트랜잭션 지원이 없었고 비즈니스 인텔리전스를 위해 설계되지 않았으며 제한된 데이터 거버넌스 지원을 제공했으며 데이터 생태계를 완벽하게 지원하려면 여전히 다른 기술(예: 데이터 웨어하우스)이 필요했습니다. 이로 인해 다양한 도구, 시스템, 기술 및 여러 데이터 사본이 뒤섞여 있는 환경이 지나치게 복잡해졌습니다.

공존하는 데이터 레이크와 데이터 웨어하우스의 출현에는 아직 부족한 점이 많습니다. 사용 사례에 대한 불완전한 지원과 호환되지 않는 보안 및

거버넌스 모델은 다양한 도구와 기술에 걸쳐 데이터 하위 집합을 포함하는 분리되고 중복된 데이터 사일로로 인해 복잡성이 증가했습니다.

레이크하우스

(Lakehouse) 2010년대 후반, 레이크하우스라는 개념이 등장했다. 이를 통해 데이터 레이크의 유연성을 저하시키지 않으면서 모든 이점과 기능을 제공하는 현대화된 버전의 데이터 웨어하우스가 도입되었습니다. 레이크하우스는 ACID 트랜잭션을 지원하는 개방형 테이블 형식을 갖춘 기술을 통해 데이터 신뢰성 및 일관성 보장과 결합된 저비용의 유연한 클라우드 스토리지 계층인 데이터 레이크를 활용합니다. 이러한 유연성은 단일 통합 플랫폼에서 스트리밍, 분석, 기계 학습과 같은 다양한 워크로드를 지원하는 데 도움이 되며 궁극적으로 모든 데이터 자산에 대한 단일 보안 및 거버넌스 접근 방식을 가능하게 합니다. Delta Lake와 레이크하우스의 출현으로 앤드투엔드 데이터 플랫폼의 패러다임은 이 아키텍처 패턴이 지원하는 주요 기능으로 인해 변화하기 시작했습니다.

레이크하우스의 기능을 이 책에서 배운 내용과 결합함으로써 레이크하우스 아키텍처가 제공하는 주요 기능을 활성화하고 Delta Lake를 완벽하게 가동하고 실행하는 방법을 배우게 됩니다.

스토리지 계층

잘 설계된 아키텍처의 첫 번째 단계 또는 계층은 데이터를 저장할 위치를 결정하는 것입니다. 여러 이기종 데이터 소스에서 다양한 형태와 모양으로 들어오는 데이터의 양이 증가하는 세상에서는 유연하고 비용 효율적이며 확장 가능한 방식으로 대량의 데이터를 저장할 수 있는 시스템을 갖추는 것이 필수적입니다. 이것이 바로 데이터 레이크와 같은 클라우드 개체 저장소가 레이크하우스의 기본 스토리지 계층인 이유입니다.

데이터 레이크란 무엇입니까?

이전에 1 장에서 정의한 데이터 레이크는 규모에 관계없이 구조적, 반구조적 또는 구조화되지 않은 데이터를 파일 및 Blob 형식으로 저장하는 비용 효율적인 중앙 저장소입니다. 데이터 레이크에서는 데이터 쓰기 시 스키마를 강요하지 않아 데이터를 있는 그대로 저장할 수 있기 때문에 가능합니다. 데이터 레이크는 일반적으로 스키마를 적용하면서 디렉터리와 파일이 포함된 계층 구조로 데이터를 저장하는 데이터 웨어하우스와 달리 플랫 아키텍처와 객체 스토리지를 사용하여 데이터를 저장합니다. 모든 개체에는 메타데이터와 고유 식별자가 태그로 지정되므로 애플리케이션에서 쉽게 액세스하고 검색할 수 있습니다.

데이터 유형 데이

터 레이크의 핵심 요소 중 하나는 클라우드 개체 저장소가 모든 유형의 데이터를 저장할 수 있는 무제한 확장성을 제공한다는 것입니다. 이러한 다양한 유형의 데이터가 다루어졌습니다.

하지만 데이터가 어떻게 구성되어 있고 어디서 발생하는지 보여주기 위해 데이터의 세 가지 분류를 정의하는 것이 가장 좋습니다.

구조화된 데이터

- 그것은 무엇입니까? 구조화된 데이터의 모든 데이터에는 미리 정의된 구조 또는 스키마가 있습니다. 이는 행과 열이 있는 테이블 형태로 데이터베이스에서 나오는 가장 일반적인 관계형 데이터입니다.
- 무엇이 그것을 생산하는가? 이와 같은 데이터는 일반적으로 기존 관계형 데이터베이스에서 생성되며 ERP(전사적 자원 관리), CRM(고객 관계 관리) 또는 재고 관리 시스템에서 자주 사용됩니다.

반구조화된 데이터

- 그것은 무엇입니까? 반구조화된 데이터는 구조화된 데이터와 같은 일반적인 관계형 형식을 따르지 않습니다. 오히려 키/값 쌍과 같이 데이터 요소를 분리하는 패턴이나 태그로 느슨하게 구조화되어 있습니다. 반구조화된 데이터의 예로는 Parquet, JSON, XML, CSV 파일은 물론 이메일이나 소셜 피드도 있습니다. • 무엇이 그것을 생산하는가? 이러한 유형의 데이터에 대한 일반적인 데이터 원본에는 비관계형 또는 NoSQL 데이터베이스, IoT 장치, 앱 및 웹 서비스가 포함될 수 있습니다.

구조화되지 않은 데이터

- 그것은 무엇입니까? 구조화되지 않은 데이터에는 조직화된 구조가 포함되어 있지 않습니다. 어떤 유형의 스키마나 패턴으로도 배열되지 않습니다. 사진(예: JPEG)이나 비디오 파일(예: MP4)과 같은 미디어 파일로 전달되는 경우가 많습니다. 기본 비디오 파일에는 전체적인 구조가 있을 수 있지만 비디오 자체를 구성하는 데이터는 구조화되어 있지 않습니다.
- 무엇이 그것을 생산하는가? 조직 데이터의 대부분은 비정형 데이터 형태로 제공되며 미디어 파일(예: 오디오, 비디오, 사진), Word 문서, 로그 파일 및 기타 형태의 서식 있는 텍스트에서 생성됩니다.

비정형 및 반정형 데이터는 AI 및 기계 학습 사용 사례에 중요한 경우가 많은 반면, 정형 및 반정형 데이터는 BI 사용 사례에 중요합니다.

기본적으로 세 가지 유형의 데이터 분류를 모두 지원하므로 데이터 레이크에서 이러한 다양한 워크로드를 지원하는 통합 시스템을 생성할 수 있습니다. 이러한 워크로드는 잘 설계된 처리 아키텍처에서 서로를 보완할 수 있으며, 이에 대해서는 이 장에서 자세히 알아볼 것입니다. 데이터 레이크는 데이터 볼륨, 유형 및 비용과 관련된 많은 문제를 해결하는 데 도움이 됩니다. Delta Lake는 데이터 레이크 위에서 실행되지만 클라우드 데이터 레이크에서 가장 잘 실행되도록 최적화되어 있습니다.

클라우드 데이터 레이크의 주요 이점 우리는 데이터 레이크가

데이터 웨어하우스의 일부 단점을 해결하는 데 어떻게 도움이 되는지 논의했습니다. 온프레미스 데이터 레이크나 데이터 웨어하우스와 달리 클라우드 데이터 레이크는 다음과 같은 여러 가지 이유로 레이크하우스 아키텍처를 스토리지 계층으로 가장 잘 지원합니다.

단일 스토리지 계층 레이크

하우스의 가장 중요한 기능 중 하나는 플랫폼을 통합하는 것이며, 클라우드 데이터 레이크는 데이터 사일로와 다양한 유형의 데이터를 제거하고 단일 개체 저장소로 통합하는 데 도움이 됩니다. 클라우드 데이터 레이크를 사용하면 단일 시스템에 서 모든 데이터를 처리할 수 있으므로 서로 다른 시스템 간에 이동하는 데이터의 추가 복사본을 생성할 수 없습니다. 다양한 시스템 간 데이터 이동이 줄어들면 통합 지점도 줄어들어 오류가 발생할 수 있는 장소가 줄어듭니다. 단일 스토리지 계층은 다양한 시스템을 포괄하는 여러 보안 정책의 필요성을 줄이고 시스템 간 협업으로 인한 어려움을 해결하는 데 도움이 됩니다. 또한 데이터 소비자에게 모든 데이터 소스를 찾을 수 있는 단일 장소를 제공합니다.

유연한 온디맨드 스토리지 계층 속도(스트리밍 대

배치), 블룸, 다양성(정형 대 비정형) 등 클라우드 데이터 레이크는 데이터를 저장할 수 있는 최고의 유연성을 제공합니다.

Rukmani Gopalan이 최근 출판한 저서 The Cloud Data Lake에서 1 "이러한 시스템은 어떤 속도로든 데이터 레이크에 입력되는 데이터, 즉 지속적으로 방출되는 실시간 데이터와 일괄적으로 수집되는 데이터 블룸과 함께 작동하도록 설계되었습니다. 예정대로." 데이터의 유연성뿐만 아니라 인프라의 유연성도 있습니다. 클라우드 제공업체를 사용하면 필요에 따라 인프라를 프로비저닝하고 탄력적으로 신속하게 확장하거나 축소할 수 있습니다. 이러한 수준의 유연성으로 인해 조직은 무제한 확장성을 제공하는 단일 스토리지 계층을 가질 수 있습니다.

분리된 스토리지 및 컴퓨팅 기준 데이터 웨어하

우스와 온프레미스 데이터 레이크는 전통적으로 스토리지와 컴퓨팅이 긴밀하게 결합되어 있었습니다. 스토리지는 일반적으로 저렴하지만 컴퓨팅은 그렇지 않습니다. 클라우드 데이터 레이크를 사용하면 이를 분리하고 스토리지를 독립적으로 확장하고 매우 적은 비용으로 방대한 양의 데이터를 저장할 수 있습니다.

기술 통합 데이터 레이크는 표준

화된 API를 통해 간단한 통합을 제공하므로 완전히 다른 기술, 도구 및 프로그래밍 언어(예: SQL, Python, Scala 등)를 사용하는 조직과 사용자가 다양한 분석 작업을 동시에 수행할 수 있습니다.

복제 클라우드

공급자는 데이터 레이크에 대해 다양한 지리적 위치에 대한 복제를 쉽게 설정할 수 있는 기능을 제공합니다. 복제를 활성화하면 규정 준수 요구 사항 충족, 비즈니스에 중요한 작업에 대한 장애 조치, 재해 복구 및 개체를 다른 위치에 더 가깝게 저장하여 대기 시간을 최소화하는 데 유용할 수 있습니다.

1 고팔란, 루크마니(2022). 클라우드 데이터 레이크, 1판. 캘리포니아주 세바스토폴: 오라일리.

가용성 대부분

의 클라우드 시스템은 클라우드 데이터 레이크에 대해 다양한 유형의 데이터 가용성을 제공합니다.

즉, 자주 액세스하는 "핫" 데이터와 비교하여 자주 액세스하지 않거나 보관되는 데이터의 경우 수명 주기 정책을 설정할 수 있습니다. 이러한 수명 주기 정책을 사용하면 규정 준수, 비즈니스 요구 사항 및 비용 최적화를 위해 다양한 스토리지 가용성 등급(자주 액세스하지 않는 데이터에 대한 비용 절감) 간에 데이터를 이동할 수 있습니다.

가용성은 SLA(서비스 수준 계약)를 통해 정의될 수도 있습니다. 클라우드에서 이는 리소스의 최소 서비스 수준에 대한 클라우드 공급자의 보증입니다. 대부분의 클라우드 제공업체는 이러한 비즈니스에 중요한 리소스에 대해 99.99% 이상의 가동 시간을 보장합니다.

비용

클라우드 데이터 레이크를 사용하면 일반적으로 사용한 만큼만 비용을 지불하므로 비용은 항상 데이터 볼륨에 맞춰 조정됩니다. 단일 스토리지 계층만 있고 다양한 시스템 간의 데이터 이동이 적고 가용성 설정이 있으며 스토리지와 컴퓨팅이 분리되어 있으므로 데이터 스토리지에 대한 비용을 분리하고 최소화할 수 있습니다. 더 큰 비용 할당을 위해 대부분의 클라우드 데이터 레이크는 다양한 데이터 계층(예: 원시 데이터 대 변환된 데이터)을 저장하기 위해 버킷 또는 컨테이너(파일 시스템, 애플리케이션 컨테이너와 혼동하지 말 것)를 제공합니다. 이러한 컨테이너를 사용하면 조직의 다양한 영역에 대해 보다 세밀하게 비용을 할당할 수 있습니다. 데이터 소스와 볼륨이 기하급수적으로 증가하고 있기 때문에 저장할 수 있는 데이터의 볼륨이나 다양성을 제한하지 않고 비용을 할당하고 최적화하는 것이 매우 중요합니다.

그림 10-1은 이 글을 쓰는 시점에 널리 사용되는 다양한 유형의 클라우드 기반 데이터 레이크와 여기에 저장되어 있는 다양한 데이터 유형을 보여줍니다.

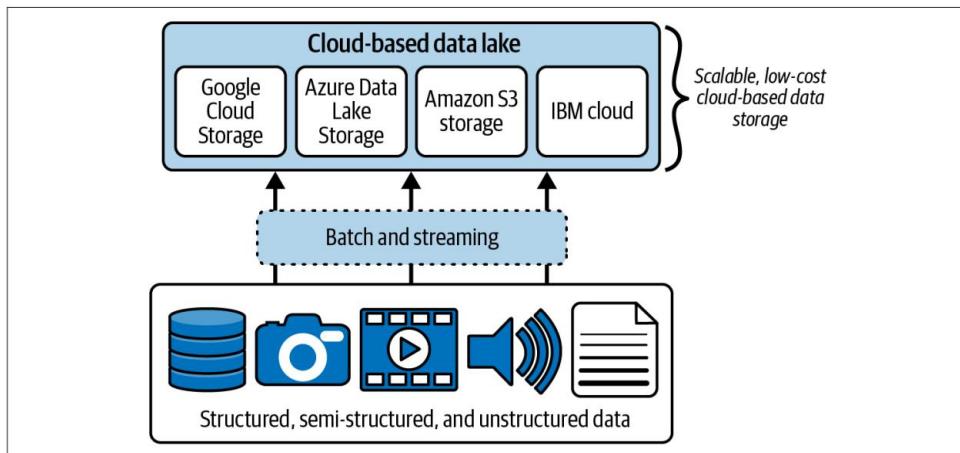


그림 10-1. 클라우드 기반 데이터 레이크 및 지원하는 데이터 유형의 예

전반적으로 스토리지 계층은 조직이 비용 효율적이고 확장 가능한 방식으로 막대한 양의 데이터를 저장하고 관리할 수 있도록 지원하므로 레이크하우스 아키텍처의 중요한 구성 요소입니다. 이제 데이터를 저장할 장소가 정의되었으므로 이를 적절하게 관리해야 합니다.

데이터 관리

클라우드 데이터 레이크를 사용하면 기본 형식으로 대규모 데이터를 탄력적으로 저장할 수 있지만 레이크하우스 기반의 다음 부분은 데이터 관리를 촉진하는 것입니다. Gartner에 따르면 데이터 관리(DM)는 기업의 데이터 소비 요구 사항을 충족하기 위해 기업의 다양한 데이터 주제 영역 및 데이터 구조 유형에 걸쳐 데이터에 대한 일관된 액세스 및 전달을 달성하기 위한 관행, 아키텍처 기술 및 도구로 구성됩니다. 모든 애플리케이션 및 비즈니스 프로세스.2 데이터 관리는 모든 조직의 핵심 기능이며 데이터 액세스 및 전달에 사용되는 도구 및 기술에 크게 좌우됩니다. 전통적으로 데이터 레이크는 데이터를 반구조화된 형식(예:

Parquet)의 "파일 묶음"으로 간단히 관리해 왔으며, 이는 지원 부족으로 인해 안정성과 같은 데이터 관리의 일부 주요 기능을 활성화하기가 어렵습니다. ACID 트랜잭션, 스키마 시행, 감사 추적, 데이터 통합 및 상호 운용성을 위한 것입니다.

데이터 레이크의 데이터 관리는 안정성을 위해 구조화된 트랜잭션 계층에서 시작됩니다. 이러한 신뢰성은 ACID 트랜잭션, 공개 테이블 형식, 일괄 데이터 처리와 스트리밍 데이터 처리 간의 통합, 확장 가능한 메타데이터 관리를 지원하는 트랜잭션 계층에서 비롯됩니다. 데이터 관리를 지원하는 레이크하우스 아키텍처의 핵심 구성 요소로 Delta Lake가 도입되는 곳입니다.

그림 10-2 에서는 데이터 레이크 위에 구축된 Delta Lake가 구조화된 트랜잭션 계층 역할을 하는 클라우드 기반 데이터 레이크가 지원하는 다양한 유형의 데이터 유형을 볼 수 있습니다.

2 “데이터 관리(DM).” Gartner, Inc, 2023년 4월 24일 액세스, <https://oreil.ly/xK4ws>.

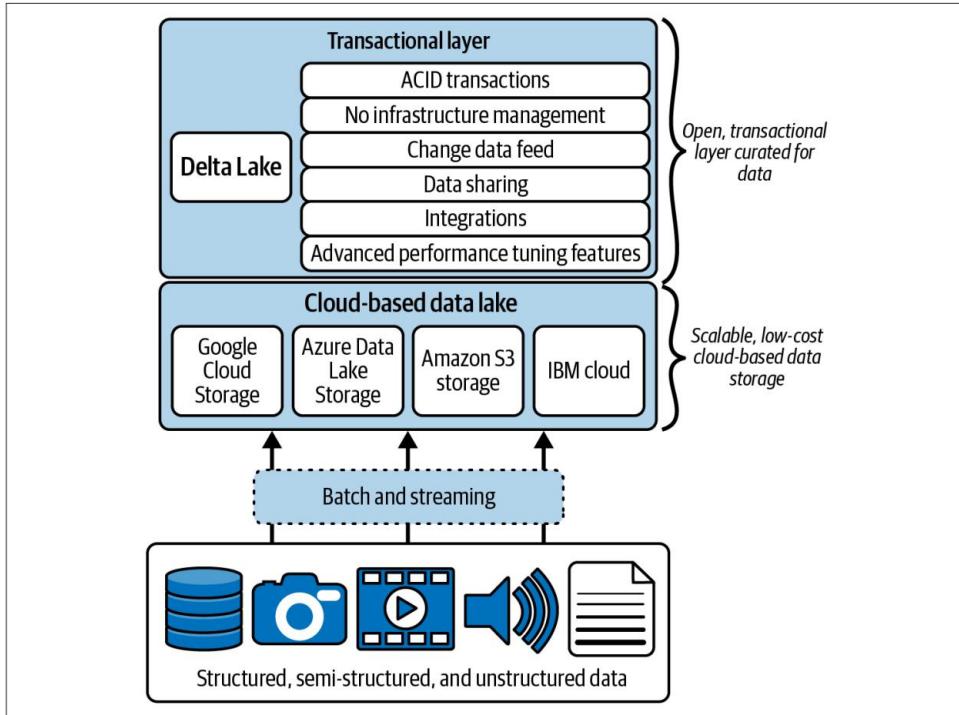


그림 10-2. Delta Lake 트랜잭션 계층

Delta Lake는 데이터 레이크에 내구성, 안정성 및 일관성을 제공합니다. 데이터 관리를 용이하게 하는 데 도움이 되는 Delta Lake의 몇 가지 핵심 요소는 다음과 같습니다.

메타데이터

Delta Lake의 핵심에는 메타데이터 계층이 있습니다. 이 계층은 Delta Lake의 핵심 기능 대부분을 활성화하는 광범위하고 확장 가능한 메타데이터 추적을 제공합니다.

이는 ACID 트랜잭션 및 기타 다양한 관리 기능을 구현하기 위한 추상화 수준을 제공합니다. 이러한 메타데이터 계층은 액세스 제어 및 감사 로깅과 같은 거버넌스 기능 활성화를 시작하는 자연스러운 장소이기도 합니다.

ACID 트랜잭션 Delta

Lake는 단일 테이블에서 동시성 트랜잭션이 활성화된 경우 데이터가 그대로 유지되도록 보장합니다. ACID 트랜잭션 지원은 데이터 레이크에 일관성과 신뢰성을 제공하며 이는 트랜잭션 로그를 통해 가능합니다. 트랜잭션 로그는 모든 커밋을 추적하고 격리 수준을 사용하여 데이터의 일관성과 정확한 보기를 보장합니다.

버전 관리 및 감사 기록

Delta Lake는 테이블의 일부인 파일에 대한 정보를 트랜잭션 로그로 저장하므로 사용자는 이전 버전의 데이터를 쿼리하고 수행할 수 있습니다.

풀백, 시간 이동이라고도 합니다. 이는 비즈니스 요구 사항이나 규제 요구 사항에 대한 데이터에 대한 전체 감사 추적을 제공하고 기계 학습 절차도 지원할 수 있습니다.

데이터 통합 이는 다양

한 소스의 데이터를 단일 장소로 통합하는 것으로 정의할 수 있습니다. Delta Lake는 테이블에서 동시 일괄 처리 및 스트리밍 작업을 처리할 수 있습니다. 사용자가 데이터를 볼 수 있는 단일 장소를 제공합니다. 뿐만 아니라 INSERT, UPDATE, DELETE 등의 DML 작업을 수행하는 기능을 통해 효과적인 데이터 통합을 수행하고 모든 테이블에서 일관성을 유지할 수 있습니다.

스키마 적용 및 진화 기준 RDBMS 시스템과 마찬가

지로 Delta Lake는 유연한 스키마 진화와 함께 스키마 적용을 제공합니다. 이는 제약 조건을 통해 깨끗하고 일관된 데이터 품질을 보장하는 동시에 다른 프로세스 및 업스트림 소스로 인해 발생하는 스키마 드리프트에 대한 유연성을 제공하는 데 도움이 됩니다.

데이터 상호 운용성 Delta

Lake는 클라우드에 구애받지 않는 오픈 소스 프레임워크이며, API 및 확장 세트를 제공하여 Apache Spark와 원활하게 상호 작용하므로 프로젝트, 기타 API 및 플랫폼 전반에 걸쳐 다양한 통합 기능이 있습니다. 이를 통해 다양한 도구와 프레임워크를 통해 기존 데이터 관리 워크플로 및 프로세스와 쉽게 통합할 수 있습니다. Delta UniForm은 Apache Iceberg 와의 형식 상호 운용성을 제공하여 통합할 수 있는 시스템 및 도구 세트를 더욱 확장합니다. Delta Lake는 조직 전체에서 데이터를 간단하고 안전하게 공유할 수 있게 해주는 Delta Sharing과 함께 공급업체 종속을 방지하고 상호 운용성을 지원합니다.

기계 학습 기계 학습(ML)

시스템은 기존 SQL에 적합하지 않은 복잡한 논리를 사용하면서 대규모 데이터 세트를 처리해야 하는 경우가 많기 때문에 이제 DataFrame API를 사용하여 Delta Lake의 데이터에 쉽게 액세스하고 처리할 수 있습니다. 이러한 API를 사용하면 ML 워크로드가 Delta Lake에서 제공하는 최적화 및 기능의 혜택을 직접 누릴 수 있습니다. 이제 모든 다양한 워크로드의 데이터 관리를 모두 Delta Lake에 통합할 수 있습니다.

데이터 관리를 처리하기 위해 더 잘 갖춰진 구조화된 트랜잭션 계층을 추가함으로써 레이크하우스는 원시 데이터, 분석용 데이터를 선별하는 ETL/ELT 프로세스 및 ML 워크로드를 동시에 지원합니다. ETL/ELT를 통한 데이터 큐레이션은 전통적으로 데이터 웨어 하우스의 맥락에서 생각되고 제시되었지만 Delta Lake에서 제공하는 데이터 관리 기능을 통해 이러한 프로세스를 한 곳으로 가져올 수 있습니다. 또한 이를 통해 ML 시스템은 Delta Lake가 제공하는 기능과 최적화의 이점을 직접 활용하여 다양한 워크로드 전반에 걸쳐 데이터 관리 및 통합을 완료할 수 있습니다. 이러한 모든 노력을 결합함으로써 구하는

데이터 레이크에 더 큰 신뢰성과 일관성을 제공하고 기업 전체에서 모든 유형의 데이터에 대한 단일 정 보 소스가 되는 레이크하우스를 만들 수 있습니다.

SQL 분석

데이터 분석, 비즈니스 인텔리전스, 데이터 웨어하우징 분야에서 SQL은 가장 일반적이고 유연한 언어 중 하나로 알려져 있습니다. 그 이유 중 하나는 진입 장벽이 낮고 접근 가능한 학습 곡선을 제공할 뿐만 아니라 수행할 수 있는 복잡한 데이터 분석 작업 때문입니다. 사용자가 데이터와 신속하게 상호 작용할 수 있도록 하는 동시에 SQL을 사용하면 모든 기술 수준의 사용자가 임시 쿼리를 작성하고, BI 도구용 데이터를 준비하고, 보고서 및 대시보드를 만들고, 다양한 데이터 분석 기능을 수행할 수 있습니다. 이러한 이유로 SQL은 데이터 엔지니어부터 비즈니스 사용자까지 모든 사람이 비즈니스 인텔리전스 및 데이터 웨어하우징을 위해 선택하는 언어가 되었습니다. 이것이 바로 레이크하우스 아키텍처가 확장성 및 성능 측면에서 뛰어난 SQL 성능을 달성하고 SQL 분석을 가능하게 하는 것이 필요한 이유입니다.

다행히 선별된 데이터의 트랜잭션 계층으로 Delta Lake를 기반으로 구축된 레이크하우스 아키텍처에는 Apache Spark 및 Spark SQL을 통해 쉽게 액세스할 수 있는 확장 가능한 메타데이터 스토리지가 있습니다.

Spark SQL을 통한 SQL 분석 Spark

SQL 구조화된 데이터 작업을 위한 Apache Spark의 모듈로, 라이브러리라고도 합니다. 이는 Spark SQL 엔진의 지원을 받는 SQL 및 DataFrame API를 사용하여 구조화된 데이터로 작업할 수 있는 프로그래밍 인터페이스를 제공합니다. 다른 SQL 엔진과 마찬가지로 Spark SQL 엔진은 효율적인 쿼리와 컴팩트 코드 생성을 담당합니다. 이 실행 계획은 런타임 시 조정됩니다.

Apache Spark 생태계는 Spark Core와 Spark SQL 엔진이 구축되는 기반인 다양한 라이브러리로 구성됩니다 ([그림 10-3](#)).

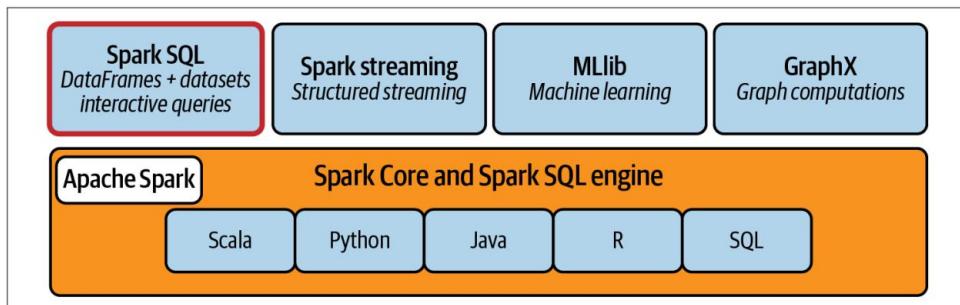


그림 10-3. Spark SQL을 포함한 Apache Spark 에코시스템

Spark SQL 라이브러리는 ANSI SQL을 지원하므로 사용자는 익숙한 SQL 구문을 사용하여 데이터를 쿼리하고 분석할 수 있습니다. 이전 장에서 살펴본 것처럼 Delta 테이블은 Spark SQL과 PySpark의 `sql()` 메서드를 사용하여 쉽게 쿼리할 수 있습니다. 예를 들면 다음과 같습니다.

```
%python
Spark.sql("taxdb.tripData에서 SELECT 개수(*)")
```

또는 이 책에 나오는 대부분의 쿼리가 작성된 방식과 유사하게 노트북이나 일부 IDE에서 매직 명령과 `%sql`을 사용하여 언어 참조를 지정하고 셀에 직접 Spark SQL을 작성할 수 있습니다.

```
%sql
SELECT 개수(*)는 Taxdb.tripData의 개수로 표시됩니다.
```

SQL뿐만 아니라 Spark SQL 라이브러리를 사용하면 DataFrame도 사용할 수 있습니다. 데이터 세트 및 테이블과 상호 작용하는 API:

```
%python
Spark.table("taxidb.tripData").count()
```

분석가, 보고서 작성자 및 기타 데이터 소비자는 일반적으로 SQL 인터페이스를 통해 데이터와 상호 작용합니다. 이 SQL 인터페이스는 사용자가 Spark SQL을 활용하여 Delta 테이블에서 단순하거나 복잡한 쿼리 및 분석을 수행하고 Delta 테이블이 제공하는 성능과 확장성을 활용하는 동시에 Spark SQL 엔진, 분산 처리, 그리고 최적화. Delta Lake는 직렬성을 보장하므로 동시에 읽기 및 쓰기가 완벽하게 지원됩니다. 이는 데이터가 다양한 ETL 워크로드를 통해 업데이트되더라도 모든 데이터 소비자가 자신 있게 데이터를 읽을 수 있음을 의미합니다. 즉, Spark SQL 엔진은 가능한 한 빨리 쿼리를 만들기 위해 Spark 클러스터에서 쿼리를 최적화하고 실행하는 데 사용되는 실행 계획을 생성합니다.

그림 10-4는 Spark SQL 라이브러리를 사용하여 SQL 쿼리를 표현하거나 DataFrame API를 사용하여 데이터 세트와 상호 작용하고 Spark SQL 엔진 및 실행 계획을 활용할 수 있음을 보여줍니다. Spark SQL을 사용하는 DataFrame API를 사용하는 Spark SQL 엔진은 클러스터에서 명령을 최적화하고 실행하는 데 사용되는 쿼리 계획을 생성합니다.

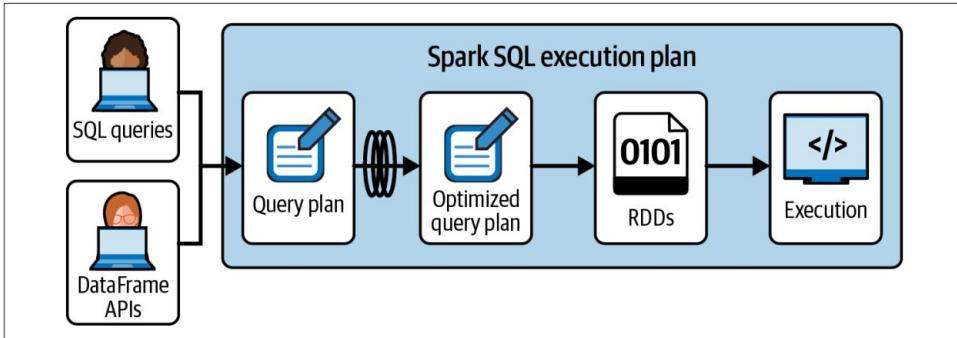


그림 10-4. Spark SQL 실행 계획



그림 10-4 에서 실행 계획의 RDD(복원력 있는 분산 데이터 세트)가 하위 수준 API를 활용하는 기존 사용자 정의 RDD를 참조하지 않는다는 점에 유의하는 것이 중요합니다. 오히려 그림에 언급된 RDD는 구조화된 데이터에 특별히 최적화되어 추가적인 오버헤드를 추가하지 않는 DataFrame 또는 데이터 세트라고도 하는 Spark SQL RDD입니다.

일반적으로 ETL 및 데이터 수집 프로세스3 또는 기계 학습 워크로드에는 DataFrame API를 사용하는 것이 권장되는 반면, 대부분의 데이터 소비자(예: 분석가, 보고서 작성기 등)는 Spark SQL을 사용하여 Delta 테이블과 상호 작용합니다. . 표준 JDBC/ODBC 데이터베이스 커넥터를 통해 또는 명령줄을 사용하여 SQL 인터페이스와 상호 작용할 수도 있습니다. JDBC/ODBC 커넥터는 Spark SQL이 분석을 위해 테이블과 상호 작용하고 사용하기 위해 Power BI, Tableau 및 기타 BI 도구와 같은 외부 도구 간의 브리지도 제공한다는 것을 의미합니다.

기타 Delta Lake 통합을 통한 SQL 분석 Delta Lake는 Spark SQL 및 나머

지 Spark 에코시스템과의 강력한 통합을 제공하는 동시에 다양한 다른 고성능 쿼리 엔진에서도 액세스할 수 있습니다. 지원되는 쿼리 엔진은 다음과 같습니다.

- 아파치 스파크 • 아
- 파치 하이브
- 프레스토
- 트리노

3 Damji, Jules S, 그 외 여러분. (2020). 스파크 학습, 2판. 캘리포니아주 세바스트폴: 오라일리.

이러한 커넥터는 Delta Lake를 Apache Spark 이외의 빅 데이터 SQL 엔진으로 가져오고 커넥터에 따라 읽고 쓸 수 있도록 도와줍니다. [Delta Lake 커넥터 리포지토리](#) 다음이 포함됩니다:

- Delta Lake 메타데이터를 읽고 쓰기 위한 기본 라이브러리인 Delta Standalone • 널리 사용되는 빅 데이터 엔진(예: Apache Hive, Presto, Apache Flink)에 대한 커넥터
Microsoft Power BI와 같은 일반적인 보고 도구도 있습니다.

다음을 포함하여 Delta Lake에서 데이터를 통합하고 읽을 수 있는 여러 관리형 서비스도 있습니다.

- Amazon Athena 및 Redshift
- Azure Synapse 및 Stream Analytics • Starburst
- 데이터브릭

[Delta Lake 웹사이트](#)를 참조하세요. 지원되는 쿼리 엔진 및 관리 서비스의 전체 목록을 확인하세요.

SQL 분석을 수행할 때는 SQL 쿼리를 해석하고 데이터 분석을 위해 델타 테이블에 대해 대규모로 실행할 수 있는 고성능 쿼리 엔진을 활용하는 것이 중요합니다. [그림 10-5](#) 에서 레이크하우스는 세 가지 서로 다른 복합 레이어와 서로 다른 레이어가 서로 통신할 수 있도록 하는 API로 구성되어 있음을 볼 수 있습니다.

스토리지 계층 정

형, 반정형 및 비정형 데이터를 위한 확장 가능하고 저렴한 클라우드 데이터 스토리지에 사용되는 데이터 레이크입니다.

Delta Lake를 통해 확

장 가능한 메타데이터가 포함된 트랜잭션 계층 ACID 규격 개방형 테이블 형식입니다.

API

SQL API를 통해 사용자는 Delta Lake에 액세스하고 읽기 및 쓰기 작업을 수행할 수 있습니다. 그런 다음 메타데이터 API는 시스템이 Delta Lake 트랜잭션 로그를 이해하여 데이터를 적절하게 읽을 수 있도록 도와줍니다.

고성능 쿼리 사용자가 Apache

Spark SQL 또는 Delta Lake와 통합되는 다른 쿼리 엔진을 통해 데이터 분석을 수행할 수 있도록 SQL을 해석하는 쿼리 엔진입니다.

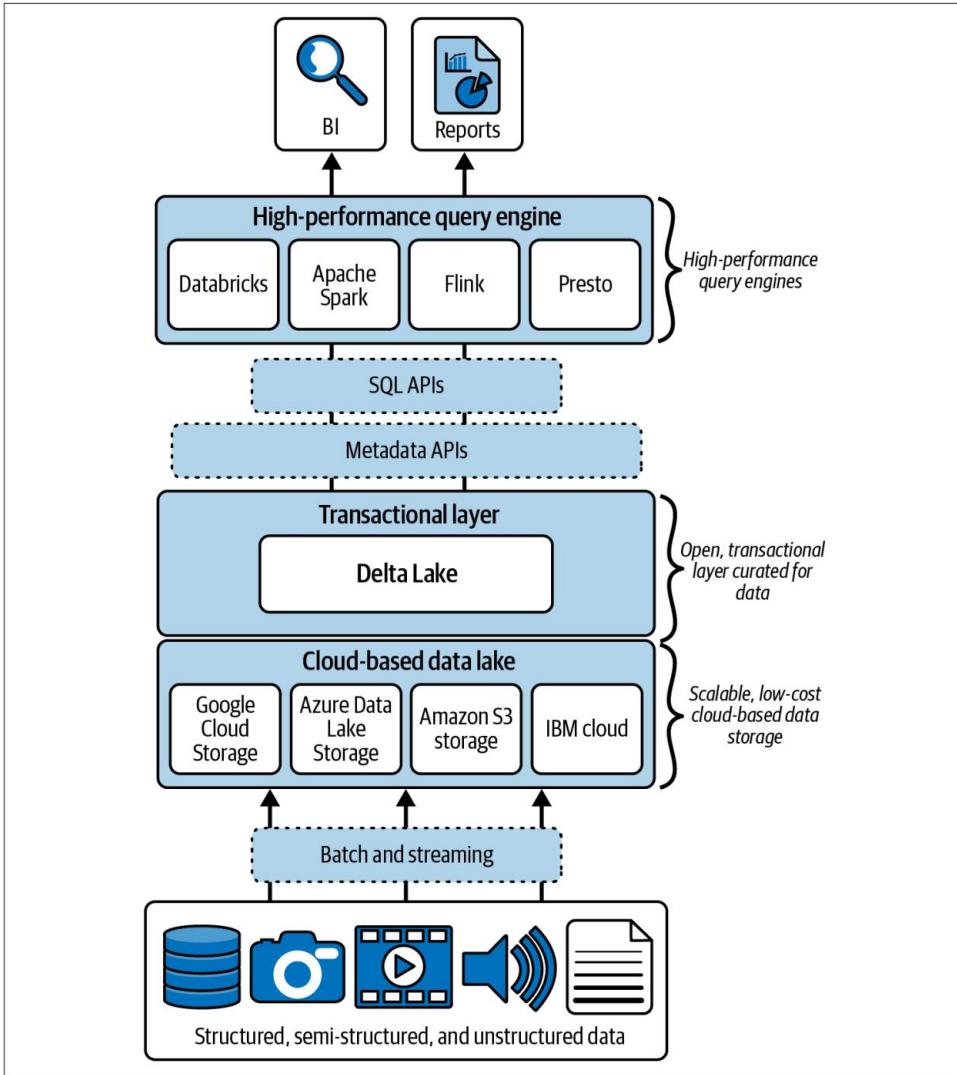


그림 10-5. BI 및 보고서에 사용되는 고성능 쿼리 엔진을 갖춘 레이크하우스 계층형 아키텍처의 예

데이터 과학 및 기계 학습을 위한 데이터

Delta Lake는 레이크하우스 전체에서 안정적이고 단순화된 데이터 관리를 제공하는 데 도움이 되므로 데이터 과학 활동 및 기계 학습에 활용되는 데이터 및 데이터 파이프라인에 다양한 이점을 제공하는 데 도움이 됩니다.

기존 기계 학습의 과제 일반적으로 MLOps(기계 학습 작업)는 엔

드 투 앤드 기계 학습 수명 주기와 관련된 일련의 사례 및 원칙입니다. MLOps에는 조직에서 기계 학습 모델을 구축하고 최종적으로 생산화하려고 할 때 대다수의 조직과 데이터 과학자가 직면하는 여러 가지 일반적인 과제가 있습니다.

데이터 사일로

데이터 엔지니어링 활동과 데이터 과학 활동 간의 격차가 커지면서 데이터 사일로가 발전하기 시작하는 경우가 많습니다. 데이터 과학자는 데이터를 정리 및 변환하고 모델의 기능으로 준비하는 별도의 ETL 및 데이터 파이프라인을 만드는 데 대부분의 시간을 소비하는 경우가 많습니다. 이러한 사일로는 일반적으로 데이터 엔지니어링에 사용되는 도구와 기술이 데이터 과학자의 동일한 활동을 지원하지 않기 때문에 발생합니다.

배치 및 스트리밍 데이터 통합

기계 학습 모델은 스트리밍 데이터에 대한 정확한 예측을 위해 기록 데이터를 사용하여 모델을 교육합니다. 문제는 기존 아키텍처가 과거 데이터와 스트리밍 데이터를 모두 안정적으로 결합하는 것을 지원하지 않기 때문에 두 유형의 데이터를 머신러닝 모델에 공급하는 데 어려움이 있다는 것입니다.

데이터 볼륨

기계 학습 시스템은 SQL에 반드시 적합하지 않은 복잡한 코드를 사용하면서 대규모 데이터 세트를 처리해야 하는 경우가 많습니다. 그리고 데이터 과학자가 데이터 엔지니어링 파이프라인을 통해 생성된 테이블에서 ODBC/JDBC 인터페이스를 통해 데이터를 사용하려는 경우 이러한 인터페이스는 매우 비효율적인 프로세스를 만들 수 있습니다.

이러한 비효율적인 프로세스는 주로 이러한 인터페이스가 SQL과 함께 작동하도록 설계되었으며 비SQL 코드 논리에 대한 제한된 지원을 제공하기 때문입니다. 이로 인해 비SQL 코드 논리에 종종 포함될 수 있는 데이터 볼륨, 데이터 변환 및 복잡한 데이터 구조로 인해 발생할 수 있는 비효율적인 비SQL 쿼리가 발생합니다.

재현성 MLOps 모범

사례에는 ML 워크플로의 모든 단계를 재현하고 검증해야 하는 필요성이 포함됩니다. 모델을 재현하는 기능은 오류 위험을 줄이고 ML 솔루션의 정확성과 견고성을 보장합니다. 일관된 데이터는 재현성에서 직면하는 가장 어려운 문제이며, ML 모델은 정확히 동일한 데이터가 사용되는 경우에만 정확히 동일한 결과를 재현합니다. 그리고 데이터는 시간에 따라 지속적으로 변경되므로 ML 재현성과 MLOps에 심각한 문제가 발생할 수 있습니다.

비표 형식 데이터 일반적

으로 데이터가 테이블에 저장되어 있다고 생각하지만 텍스트, 이미지, 오디오, 비디오 또는 PDF 파일과 같은 대규모 비표 데이터 컬렉션을 지원하기 위해 기계 학습 워크로드에 대한 사용 사례가 늘어나고 있습니다. 이 비표 데이터

테이블 형식 데이터에 필요한 것과 동일한 거버넌스, 공유, 저장 및 데이터 탐색 기능이 필요합니다. 이러한 디렉터리 모음과 표 형식이 아닌 데이터를 카탈로그화하는 특정 유형의 기능이 없으면 표 형식 데이터에 사용되는 것과 동일한 거버넌스 및 관리 기능을 제공하기가 매우 어렵습니다.

기계 학습 모델을 생산하기 위해 기존 아키텍처에서 발생하는 문제로 인해 기계 학습은 종종 매우 복잡하고 고립된 프로세스가 됩니다. 이러한 사일로화된 복잡성으로 인해 데이터 관리에 더 많은 어려움이 발생합니다.

기계 학습을 지원하는 Delta Lake 기능 다행스럽게도 Delta Lake는 전통

적으로 기계 학습 활동으로 인해 발생했던 이러한 데이터 관리 문제를 무효화하고 BI/보고 분석 및 고급 분석에 사용되는 데이터와 프로세스 간의 격차를 해소하는데 도움이 됩니다. 기계 학습 수명 주기를 지원하는 데 도움이 되는 Delta Lake의 다양한 기능은 다음과 같습니다.

최적화 및 데이터 볼륨 Delta Lake

가 제공하는 이점은 모두 Apache Spark를 기반으로 구축되었다는 사실에서 시작됩니다. 데이터 과학 활동은 Spark SQL을 통해 DataFrame API를 사용하여 Delta Lake 테이블에서 직접 데이터에 액세스할 수 있습니다. 이를 통해 기계 학습 워크로드는 Delta Lake가 제공하는 최적화 및 성능 향상의 혜택을 직접 누릴 수 있습니다.

일관성 및 안정성 Delta Lake

는 일관되고 안정적인 데이터를 보장하는 ACID 트랜잭션을 제공합니다. 이는 부정확하거나 일관되지 않은 데이터로 인한 부정적인 영향을 피하기 위해 모델 교육 및 예측에 이러한 수준의 안정성이 필요하기 때문에 기계 학습 및 데이터 과학 워크플로에 중요합니다.

배치 및 스트리밍 데이터 통합

Delta Lake 테이블은 Spark Streaming 및 Spark Structured Streaming을 통해 가능해진 기록 소스와 스트리밍 소스 모두의 지속적인 데이터 흐름을 원활하게 처리할 수 있습니다. 즉, 두 가지 유형의 데이터가 모두 단일 테이블에 통합되므로 기계 학습 모델의 데이터 흐름 프로세스를 단순화할 수 있습니다.

스키마 적용 및 발전 기본적으로 Delta

Lake 테이블에는 스키마 적용이 있습니다. 즉, 더 나은 데이터 품질을 적용할 수 있으므로 기계 학습 입력에 사용되는 데이터의 신뢰성이 높아집니다. 그러나 Delta Lake는 스키마 진화도 지원합니다. 즉, 데이터 과학자는 기존 데이터 모델을 손상시키지 않고도 기존 기계 학습 생산 테이블에 새 열을 쉽게 추가할 수 있습니다.

버전 관리 이전

장에서 Delta Lake를 사용하면 데이터 버전을 쉽게 지정하고 트랜잭션 로그를 통해 시간 여행을 수행할 수 있다는 점을 배웠습니다. 이 버전 관리를 사용하면 특정 시점의 특정 데이터 세트를 바탕으로 기계 학습 실험과 모델 출력을 쉽게 다시 생성할 수 있으므로 ML 재현성과 관련하여 흔히 볼 수 있는 문제를 완화하는 데 도움이 됩니다. 단순화된 버전 관리를 통해 ML 모델에 대한 더 나은 추적성, 재현성, 감사 및 규정 준수를 제공할 수 있으므로 MLOps 프로세스에서 볼 수 있는 일부 문제를 크게 줄이는 데 도움이 됩니다.

통합 1 장 과 이 장

에서는 Apache Spark에서 Spark SQL 라이브러리를 사용하여 Delta Lake에 액세스하는 방법에 대해 읽었습니다. 또한 8 장에서 Spark 스트리밍 및 Delta Lake와 해당 라이브러리의 통합에 대해 읽었습니다. Spark 생태계의 추가 라이브러리는 MLlib입니다. MLlib는 일반적인 학습 알고리즘, 기능화, 파이프라인, 지속성 및 유ти리티에 대한 액세스를 제공합니다. 실제로 많은 기계 학습 라이브러리(예: TensorFlow, scikit-learn)도 DataFrame API를 활용하여 Delta Lake 파일에 액세스할 수 있습니다.

Spark 생태계는 모든 유형의 데이터 및 분석 사용 사례에 대한 다기능 지원을 제공하기 위해 Spark Core 위에서 실행되는 여러 라이브러리로 구성됩니다 ([그림 10-6](#)). Spark 표준 라이브러리 외에도 Spark는 Spark MLlib 모델을 추적, 기록 및 재현할 수 있는 종단 간 기계 학습 수명주기 관리를 위한 인기 있는 오픈 소스 플랫폼인 MLflow와 같은 다른 플랫폼과도 통합되어 있습니다..

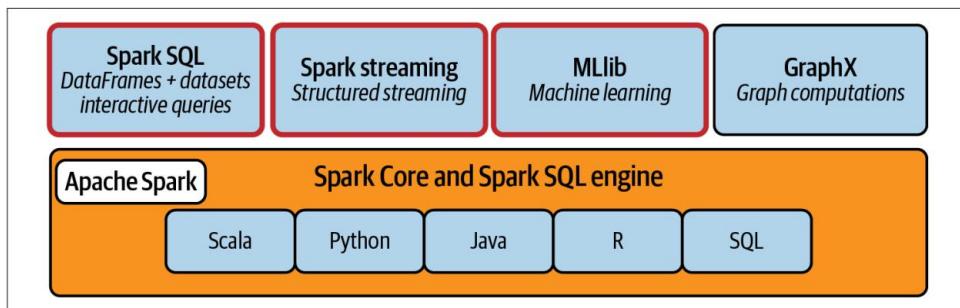


그림 10-6. Spark SQL, Spark Streaming 및 MLlib 라이브러리를 보여주는 Apache Spark 에코시스템

앞서 언급했듯이 기계 학습 모델은 일반적으로 TensorFlow, PyTorch, scikit-learn 등과 같은 라이브러리를 사용하여 구축됩니다. 그리고 Delta Lake는 모델 구축을 위한 직접 기술은 아니지만 많은 모델에 대한 가치 있고 기초적인 지원을 다루고 제공하는 데 중점을 둡니다. 머신러닝 활동이 직면하는 과제에 대해 설명합니다.

MLOps와 모델은 Delta Lake가 제공하는 데이터 품질과 무결성, 재현성, 신뢰성 및 데이터 통합에 의존합니다. Delta Lake가 제공하는 강력한 데이터 관리 및 통합 기능은 MLOps를 단순화하고 기계 학습 엔지니어와 데이터 과학자가 모델 교육 및 배포에 사용되는 데이터에 더 쉽게 액세스하고 작업할 수 있도록 해줍니다.

모든 것을 하나로 통합 Delta

Lake에서 지원하는 기능을 통해 기계 학습 및 데이터 엔지니어링 수명 주기를 통합하는 것이 훨씬 쉬워집니다. Delta Lake를 사용하면 기계 학습 모델이 기본적으로 Delta 테이블 최적화를 활용하면서 단일 위치에서 제공되는 기록(배치) 및 스트리밍 데이터로부터 학습하고 예측할 수 있습니다.

ACID 트랜잭션 및 스키마 적용은 기계 학습 모델 입력에 사용되는 테이블에 데이터 품질, 일관성 및 안정성을 제공하는 데 도움이 됩니다. 그리고 Delta Lake의 스키마 발전은 기존 프로세스에 큰 변화를 주지 않고도 시간이 지남에 따라 기계 학습 출력을 변경할 수 있도록 도와줍니다. 시간 여행을 통해 기계 학습 모델을 쉽게 감사하거나 재현할 수 있으며, Spark 에코시스템은 추가 라이브러리와 기타 기계 학습 수명 주기 도구를 제공하여 데이터 과학자를 더욱 지원합니다.

그림 10-7 에서는 완전히 구성된 레이크하우스 환경의 결과 레이어를 모두 볼 수 있습니다. Delta Lake의 기능은 사일로를 줄이고 워크로드를 통합하려는 노력의 일환으로 데이터 엔지니어와 데이터 과학자 간의 격차를 줄이는 데 도움이 됩니다.

Delta Lake와 강력한 레이크하우스 아키텍처를 함께 사용하면 조직은 더 빠르고 효율적인 방식으로 기계 학습 모델을 구축하고 관리할 수 있습니다.

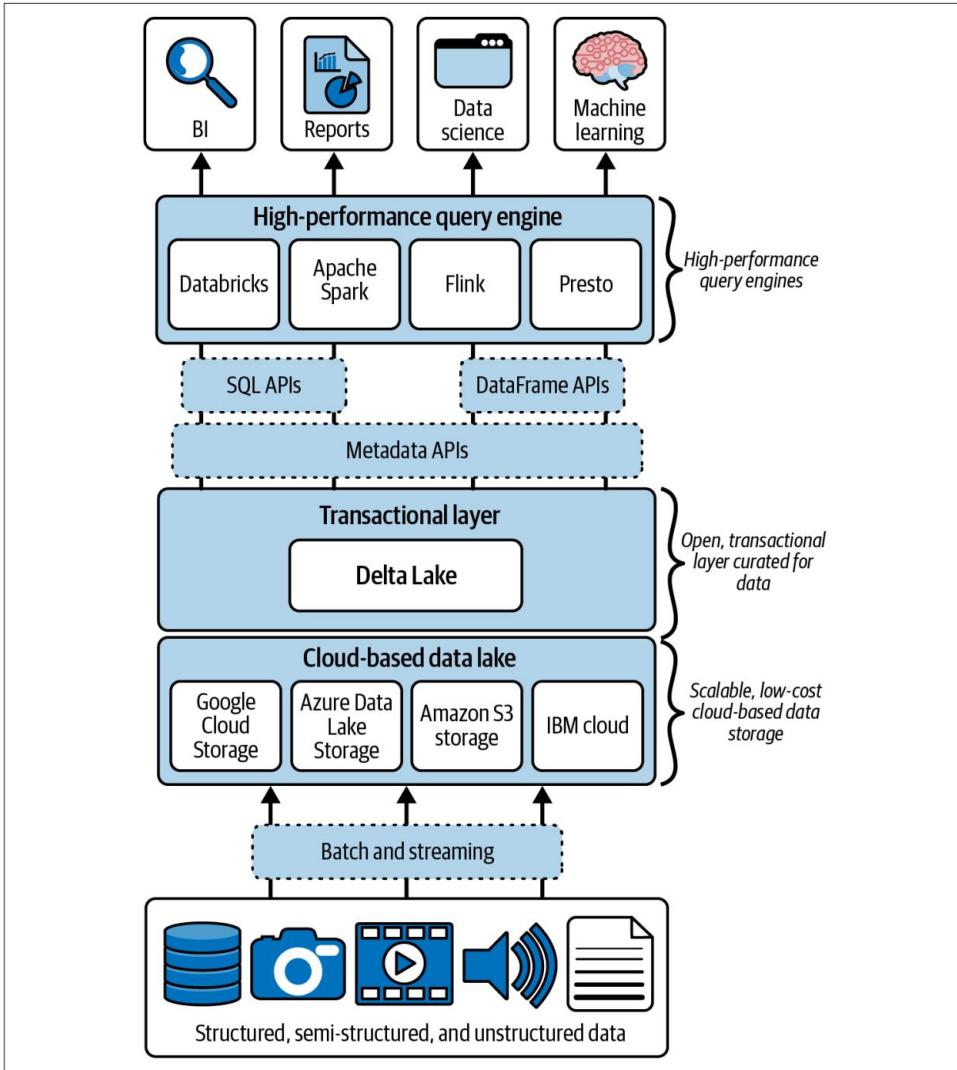


그림 10-7. 데이터 과학 및 기계 학습 워크로드가 추가된 레이크하우스 아키텍처

메달리온 아키텍처

레이크하우스는 통일이라는 아이디어를 중심으로 다양한 기술의 최고의 요소를 한곳에 결합합니다. 이는 레이크하우스 자체 내의 데이터 흐름이 이러한 데이터 통합을 지원하는 것도 중요하다는 것을 의미합니다. 모든 사용 사례를 지원하려면 이 데이터 흐름에서 배치 및 스트리밍 데이터를 단일 데이터 흐름으로 병합하여 전체 데이터 수명 주기에 걸쳐 시나리오를 지원해야 합니다.

1장에서는 궁극적으로 Delta Lake를 통해 활성화되는 브론즈, 실버 및 골드 레이어가 포함된 인기 있는 데이터 디자인 패턴인 메달리온 아키텍처의 아이디어를 소개했습니다. 이 인기 있는 패턴은 데이터 레이크 전체에서 데이터의 구조와 품질을 개선하기 위해 반복적인 방식으로 레이크하우스의 데이터를 구성하는 데 사용됩니다. 각 계층은 배치 및 스트리밍 데이터 흐름을 통합하는 동시에 분석을 위한 특정 기능과 목적을 갖습니다. 메달리온 아키텍처의 예가 그림 10-8에 나와 있습니다.

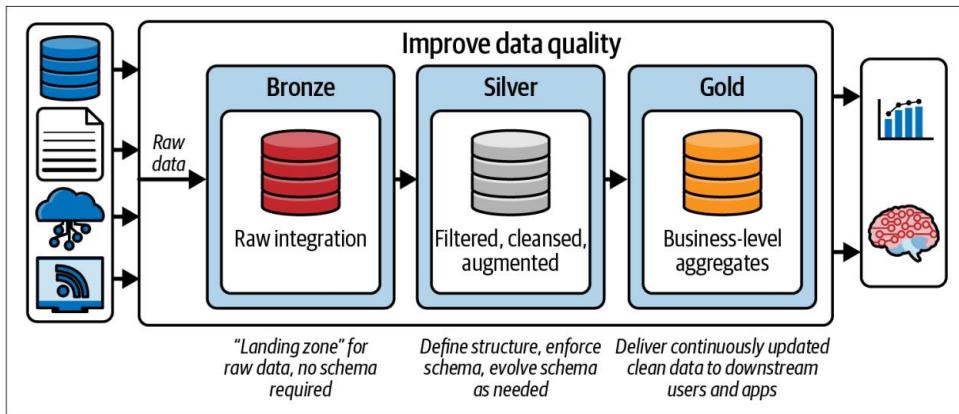


그림 10-8. 데이터 레이크하우스 솔루션 아키텍처

그림 10-8에 표시된 Bronze, Silver 및 Gold 레이어는 표 10-1에 요약되어 있습니다. 각 계층에 대해 해당 비즈니스 가치, 속성 및 구현 세부 정보(예: "완료 방법")를 볼 수 있습니다.

표 10-1. 메달리온 아키텍처 요약

	청동	은	금
사업 값	<ul style="list-style-type: none"> • 스스로부터 받은 내용을 정확하게 감사합니다. • 직접 진행하지 않고 재처리할 수 있는 기능 	<ul style="list-style-type: none"> • 비즈니스에 유용한 첫 번째 레이어 • 데이터 검색, 셀프 서비스, 임시 보고, 고급 분석 및 ML 지원 	<ul style="list-style-type: none"> • 데이터는 비즈니스 사용자가 쉽게 탐색할 수 있는 형식입니다. • 뛰어난 성능
속성	<ul style="list-style-type: none"> • 어떤 종류의 비즈니스 규칙이나 변환도 없습니다. • 이 레이어에 새 데이터를 빠르고 쉽게 가져올 수 있어야 합니다. 	<ul style="list-style-type: none"> • 시장 출시 속도 우선순위 성능을 기록합니다. 충분한 변환만 하면 됩니다. • 품질 데이터 기대 	<ul style="list-style-type: none"> • 비즈니스 사용 사례 및 사용자 경험의 우선순위 지정 • 사전 계산된 비즈니스별 변환 • 다양한 소비에 대해 별도의 데이터 보기를 가질 수 있습니다.

청동	은	금
어떻게 이루어 졌는가	<ul style="list-style-type: none"> 수령한 내용의 사본을 포함해야 합니다. 일반적으로 데이터는 수신된 날짜를 기준으로 풀더에 저장됩니다. 	<ul style="list-style-type: none"> 델타 병합 • 조명 모델링 포함 가능 (3NF, 볼팅) • 데이터 품질 점검이 포함되어야 합니다.

다음 섹션에서는 메달리온 아키텍처를 구성하는 다양한 레이어를 더 자세히 살펴보겠습니다.

청동층(원시 데이터)

데이터 소스의 원시 데이터는 변환이나 비즈니스 규칙 적용 없이 Bronze 계층으로 수집됩니다. 이 레이어는 원시 데이터의 "착륙 영역"이므로 이 레이어의 모든 테이블 구조는 소스 시스템 구조와 정확하게 일치합니다. 데이터 소스의 형식이 유지되므로 데이터 소스가 CSV 파일인 경우 Bronze에 CSV 파일로 저장되고, JSON 데이터는 JSON으로 작성되는 등의 방식이 됩니다. 데이터베이스 테이블에서 추출된 데이터는 일반적으로 Bronze에 CSV 파일로 저장됩니다. 쪽모이 세공 마루 또는 AVRO 파일.

이 시점에서는 스키마가 필요하지 않습니다. 데이터가 수집되면 데이터 소스, 전체 로드가 수행되었는지 증분 로드가 수행되었는지 여부, 필요한 경우 증분 로드를 지원하는 세부 워터마크를 포함하는 상세한 감사 기록이 유지됩니다.

브론즈 레이어에는 보관 메커니즘이 포함되어 있어 데이터를 장기간 보관할 수 있습니다. 상세한 감사 기록과 함께 이 아카이브는 메달리온 아키텍처의 다운스트림 어딘가에서 오류가 발생할 경우 데이터를 재처리하는 데 사용될 수 있습니다.

수집된 데이터는 로드 날짜 및 시간, ETL 프로세스와 같은 추가 메타데이터로 보강되는 경우가 많지만 소스 시스템 형식의 구조와 데이터 유형을 유지하면서 Bronze 계층 "소스 시스템 미러링"에 저장됩니다. 시스템 식별자. 수집 프로세스의 목표는 데이터 계보 및 재처리가 가능하도록 충분한 감사 및 메타데이터를 사용하여 소스 데이터를 Bronze 계층에 빠르고 쉽게 배치하는 것입니다.

브론즈 계층은 변경 데이터 캡처(CDC) 프로세스의 소스로 사용되는 경우가 많으므로 새로 도착하는 데이터가 실버 및 골드 계층을 통해 다운스트림으로 즉시 처리될 수 있습니다.

실버 레이어 실버 레이어

에서는 먼저 데이터를 정리하고 정규화합니다. 우리는 날짜 및 시간과 같은 구성에 표준 형식을 사용하고, 회사의 열 명명 표준을 시행하고, 데이터 중복을 제거하고, 일련의 추가 데이터 품질 검사를 수행하여 필요할 때 품질이 낮은 데이터 행을 삭제하는지 확인합니다.

다음으로 관련 데이터를 결합하고 병합합니다. Delta Lake MERGE 기능은 이러한 목적에 매우 적합합니다. 예를 들어 다양한 소스(영업, CRM, POS 시스템 등)의 고객 데이터가 단일 엔터티로 결합됩니다. 다양한 주제 영역에서 재사용되는 데이터 엔터티인 규격 데이터는 뷰 전체에서 식별되고 정규화됩니다. 이전 예에서 결합된 고객 엔터티는 이러한 일치 데이터의 예입니다.

이 시점에서 데이터에 대한 통합된 전사적 관점이 나타나기 시작합니다. 여기서는 가능한 최소한의 노력으로 충분한 세부 정보를 제공하여 메달리온 아키텍처 구축에 대한 민첩한 접근 방식을 유지하는 "직절한" 철학을 적용합니다.

이 시점에서 우리는 스키마 적용을 시작하고 스키마가 다운스트림으로 발전하도록 허용합니다. 실버 레이어에서는 GDPR 및/또는 PII/PHI 시행 규칙을 적용할 수도 있습니다.

이는 데이터 품질이 적용되고 엔터프라이즈 뷰가 생성되는 첫 번째 계층이므로 특히 셀프 서비스 분석 및 임시 보고와 같은 목적을 위해 비즈니스에 유용한 데이터 소스로 사용됩니다. 실버 레이어는 기계 학습 및 AI 사용 사례를 위한 훌륭한 데이터 소스임이 입증되었습니다. 실제로 이러한 유형의 알고리즘은 골드 레이어의 소비 형식 대신 실버 레이어의 "덜 다듬어진" 데이터에서 가장 잘 작동합니다.

골드 계층 골드 계층에

서는 비즈니스 수준 집계를 생성합니다. 이는 표준 Kimball 스타 스키마, Inmon 눈송이 스키마 차원 모델 또는 소비자 비즈니스 사용 사례에 맞는 기타 모델링 기술을 통해 수행할 수 있습니다. 데이터 변환 및 데이터 품질 규칙의 마지막 계층이 여기에 적용되어 조직에서 단일 정보 소스 역할을 할 수 있는 신뢰할 수 있는 고품질 데이터가 생성됩니다.

골드 계층은 다운스트림 사용자와 애플리케이션에 고품질의 깨끗한 데이터를 제공하여 지속적으로 비즈니스 가치를 제공합니다. 골드 계층의 데이터 모델에는 소비 사용 사례에 따라 데이터에 대한 다양한 관점이나 보기가 포함되는 경우가 많습니다. 골드 계층은 파티셔닝, 데이터 건너뛰기, Z 순서 지정과 같은 여러 Delta Lake 최적화 기술을 구현하여 성능이 뛰어난 방식으로 고품질 데이터를 제공합니다.

BI 도구, 보고, 애플리케이션 및 비즈니스 사용자의 최적 사용을 위해 선별된 이 계층은 고성능 쿼리 엔진을 사용하여 데이터를 읽는 기본 계층이 됩니다.

완전한 레이크하우스 Delta Lake 기

반 아키텍처의 일부로 메달리온 아키텍처를 구현하면 레이크하우스의 모든 이점과 확장성을 확인할 수 있습니다. 이 장 전체에서 레이크하우스를 구축하는 동안 전체 데이터 플랫폼을 통합하기 위해 다양한 계층이 어떻게 서로 보완하는지 확인했습니다.

그림 10-9는 메달리온 건축물의 모습을 포함하여 호수집 전체를 보여줍니다. 메달리온 아키텍처는 확실히 레이크하우스의 데이터 흐름에 대한 유일한 디자인 패턴은 아니지만 가장 인기 있는 패턴 중 하나이며 그럴 만한 이유가 있습니다. 궁극적으로 메달리온 아키텍처는 Delta Lake에서 지원하는 기능을 통해 단일 데이터 흐름의 데이터 통합을 지원하여 모든 퍼스она나에 대해 일괄 및 스트리밍 워크로드, 기계 학습, 비즈니스 수준 집계 및 분석을 전체적으로 지원합니다.

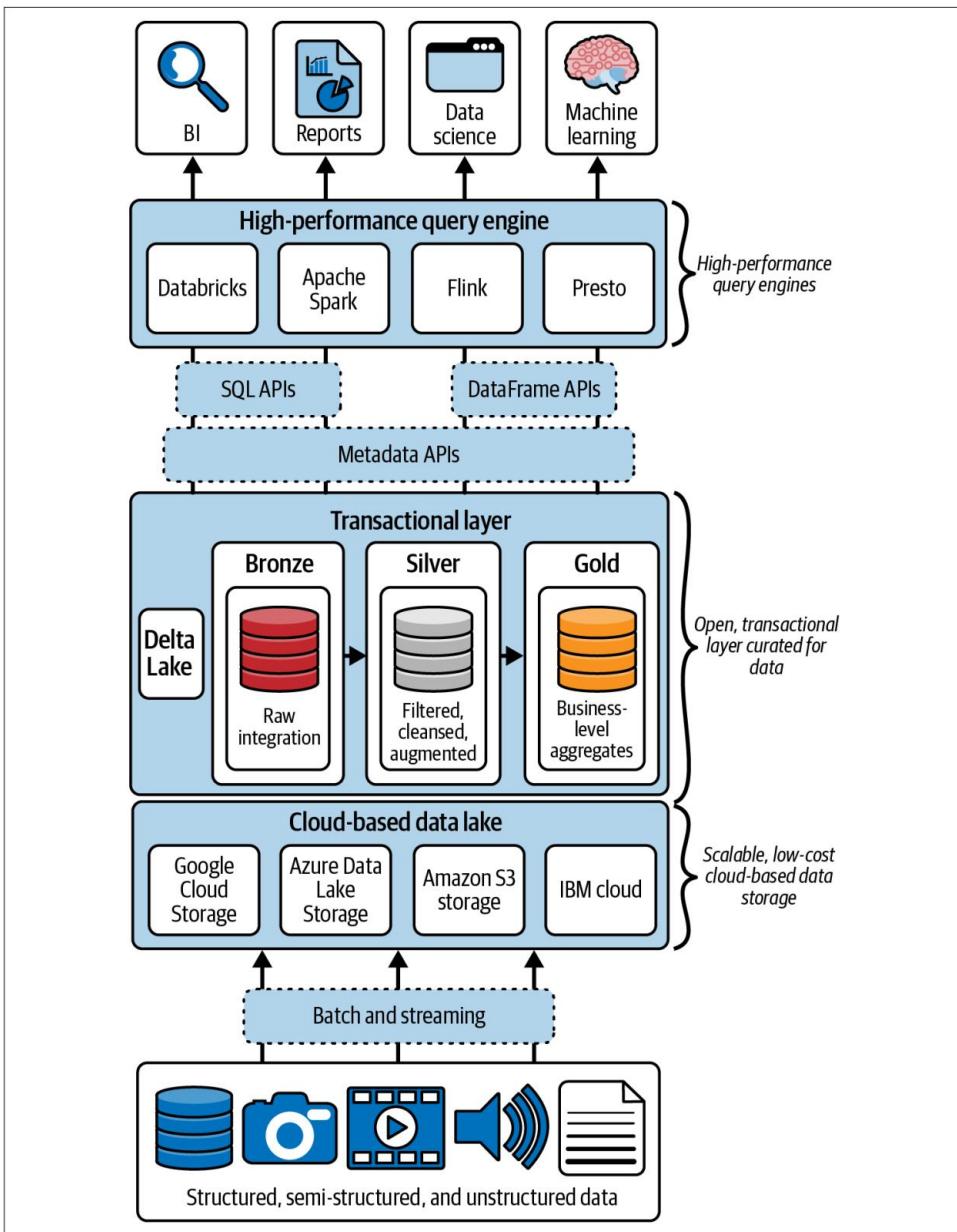


그림 10-9. 완전한 엔드투엔드 레이크하우스 아키텍처

결론

이 책 전반에 걸쳐 Delta Lake의 오픈 테이블 형식, 오픈 소스 표준의 출현으로 데이터 레이크에 대한 안정성, 확장성 및 전반적으로 향상된 데이터 관리가 어떻게 제공되었는지 배웠습니다. Delta Lake는 기존 기술의 많은 한계를 극복함으로써 기존 데이터 웨어하우스와 데이터 레이크 간의 격차를 해소하고 레이크하우스 형태의 단일 통합 빅 데이터 관리 플랫폼을 조직에 제공합니다.

Delta Lake는 조직이 데이터를 저장, 관리, 처리하는 방법을 지속적으로 변화시키고 있습니다. ACID 트랜잭션, 데이터 버전 관리, 스트리밍 및 일괄 트랜잭션 지원, 스키마 적용, 성능 튜닝 기술과 같은 강력한 기능을 통해 사실상 데이터 레이크를 위한 개방형 테이블 형식이 선택되었습니다.

이 글을 쓰는 시점에서 Delta Lake는 매달 수백만 건의 다운로드가 이루어지고 점점 늘어나는 참여자들로 구성된 강력한 커뮤니티를 통해 세계에서 가장 널리 채택되는 레이크하우스 형식입니다. Delta Lake의 기여자의 힘과 채택이 계속 증가함에 따라 Delta 생태계는 계속 확장되고 빅 데이터의 전체 분야가 계속 발전함에 따라 자연스럽게 오픈 소스 형식과 오픈 소스 커뮤니티.

Delta Lake와 레이크하우스 패러다임의 지속적인 상승은 데이터 플랫폼과 데이터 관리의 발전에 중요한 이정표를 제시합니다. Delta Lake는 오늘날의 데이터 중심 세계에서 대규모로 성공할 수 있는 기능과 기능을 제공하는 반면, Lakehouse는 이를 지원하기 위한 확장 가능한 통합 아키텍처를 제공합니다. Delta Lake와 레이크하우스는 계속해서 아키텍처를 단순화하고 성장하는 데이터 및 기술 생태계에서 혁신을 주도하는 데 중요한 역할을 할 것입니다.

색인

†

- ACID(원자성, 일관성, 격리, 내구성), 2, 223개의 레이크하우스 및 17
개
- ADLS(Azure Data Lake Storage Gen 2), 11, 209
- ALTER COLUMN 명령, 168-169
- ALTER TABLE 명령, 121, 168-169
- ALTER TABLE REPLACE COLUMNS com -
ALTER TABLE...ADD
COLUMN 명령, 166-167
- Amazon Simple Storage Service(S3), 11
- 아파치 하둡, 11
- Apache Spark(Spark 참조) 아키텍처 데이터 웨어하우스
우스, 3-6 레이크하우스, 델타 기반, 19 메달리온, 21-22 모놀리식, 5 보관 데이터, 시간 여행 및, 137 원자성, 2 트랜잭션 로그 및, 36 감사 기록, 223 감사, 시간 여행과, 125 AvailableNow 옵션, 구조적 스트리밍, 195-197
- AWS S3, 209
- Azure Data Lake Storage Gen 2(ADLS), 11, 209
- 비 BI(비즈니스 인텔리전스), 9 빅데이터, 9-10
- 씨 CDF(변경 데이터 피드), 143 감사 추적 테이블, 144 다운스트림 소비자, 144 활성화, 144-146, 184
- ETL 작업, 143 개의 스트리밍 기, 201-203 레코드, 수정, 145 보기, 146-149 경고 및 고려 사항, 149-150 체크포인트 파일, 193-195 클라우드 객체 스토리지, 데이터 공유 및, 209-210
- 변경, 121 제거, 122 보기, 121 클러스터
- 쓰기 시 데이터, 120 유동
- 클러스터링, 118 활성화, 119-120 파티션, 118 경고 및 고려 사항, 122-123 스키마 진화 및, 157 압축 파일, 109 압축, 109
- 압축, 109 컴퓨팅 성능, 112
- 일관성, 2 소비자

외부 시스템 6개, 인간 6개, 내부 시스템 6 개 COPY INTO 명령, 68-69 CREATE EXTERNAL TABLE 명령, 119 CREATE TABLE 명령, 50 CRM(고객 관계 관리), 4 크로스 플랫폼 데이터 공유, 210	메타데이터 및, 107 분할, 103 쿼리 및, 117 통계, 100 데이터 유형, 레이크 하우스 반구조화된 데이터, 219 구조화된 데이터, 219 구조화되지 않은 데이터, 219 데 이터 불통, 230 데이터 워어하 우스, 217 아키텍처, 3-6
디	
데이터	
프리젠테이션 레이어, 소비자, 6 원시 데이터, 5 요약 데이터, 6 데이터 아키텍처, 기록, 2 데이터 보관, 137 데이터 통합, 230 데이 터 피드 변경, 시간 여행 및, 143-150 데이터 파일, 물리적 삭 제, 138 데이터 통합, 224 데이터 레이크하우스(레이크하우스 참 조) 데이터 레이크, 217 컴퓨팅 성능, 12 설명, 218 형식, 13 메타데이터, 13 장단점, 13-14 읽기 스kip마, 13 저장소, 12 원 본 용어, 10 데이터 관 리(DM)(DM 참조) 데이터 마이닝, 6개 데이터 제공 자, 211-212개 데이터 수신자, 211-212개 데이터 보유	ETL(추출, 변환, 로드), 기록 통찰력 5개, 제한 사항 8개, 메타 데이터 10개, 원시 데이터 5개, 준 비 영역 5개, 스 타 스키마 4개, 요약 데이터 9개, 데 이터베이스 6개
시간 여행 및, 134 데이터 파 일 보존, 134-137 로그 파일 보존, 136 데이터 공유, 205 (델타 공유 참 조) 클라우드 객체 스토리 지, 209-210 상용 솔루션, 207-208 기준, 어려움, 205 크로스 플랫폼, 210 자체 개발 솔루션, 206, 207 레거시 솔루 션, 206, 207 라이브 데이터, 210 데이터 사 일로, 3, 230 데이터 건너뛰 기, 99	인터넷 및 관계형 데이 터베이스 3개, 기록, 2개 Databricks, 28 액 체 클러스터링 및 119 데이터프레임 스트리밍, 185, 187 데이터프레임 API, 17 데이터프레임 열, 누락, 160-162 델타 테이블, 66-67에 추가 DataFrameWriter API, 50, 54-57 압축 파일, 109 메타데이터, 사용자 정 의, 78 재파티션, 110 데이터 세 트, MERGE 작업, 91-92 DDL(데이터 정의 언어), 50-53 디버깅, 시간 여행 및 125 분 리된 스토리지 및 컴퓨팅, 220
DELETE 명령, 81, 84-86, 103 성능 조정, 86	
델타 커넥터, 23-24 델타 레이크 ACID 트랜잭션, 19개 감사, 19 개 일괄 처리, 19개 DML 지원, 19 형식, 30-35 메타데이터, 19 처리 모델, 19	

스키마, 19 스토리지, 22 스트리밍, 19 구조적 스트리밍, 184 트랜잭션 로그, 19 델타 로그, 25

델타 공유, 22, 23 종양 집중식 가버넌스, 210 클라이언트 다양성, 210 데이터 제공자, 211-212 데이터 수신자, 211-212 설계 이점, 212 폰 소스, 목표, 210 확장성, 210

델타 공유 프로토콜 Apache Spark 커넥터, 213 Python 커넥터, 213 설치, 213 프로필 파일, 213 공유 테이블, 읽기, 214-215

델타 독립형, 23

델타 테이블 CDF(변경 데이터 피드), 143 감사 추적 테이블, 144 다운스트림 소비자, 144 활성화, 144-146, 184

ETL 작업, 143 보기, 146-149 경고 및 고려 사항, 149-150 변경 이벤트, 143 열 추가, 158-160, 166-167 맷글, 167 데이터 유형 변경, 162-164, 179-181 삭제, 174-177 생성 열, 58-60 매핑, 169-171 DataFrame에서 누락, 160-162 이름 변경, 179-181 NullType, 164-165 순서 변경, 168-169 이름 바꾸기, 171-172 교체, 172-174 변환, 164 생 성

DDL(데이터 정의 언어) 명령, 50-53

DeltaTableBuilder API, 50, 57-58 DESCRIBE 명령, 53-54

PySpark DataFrameWriter API, 50, 54-57 데이터 삭제, 81-87 DataFrames, 추가, 66-67 DELETE 작업, 84-86 성능 조정, 86

역사 설명, 82-83 위치, 등록, 60 MERGE 작업, 90 데이터 세트, 91-92 프로세스, 98 사용 사례, 90

MERGE 문, 92-95 DESCRIBE HISTORY 및 일치하지 않는 행 개, 97 개, 패턴 73-74개, 검사, 74 개 바꾸기, 75-76개 단일 열, 타임스탬프별 쿼리 71-72 개, 이전 버전 133 개, PySpark로 읽기 132-134 개, 63-64 (SQL 포함), 60-63 스키마

진화, 157-165 저장, 152 시 간 여행 RESTORE 명령 고려 사항 및 경고, 131-132 복원, 128-129 타임 스탬프 및, 129 트랜잭션 로그, 130 버전 기록, 129 UPDATE 작업, 88-90 성능 조정, 90 사용 사례, 87 upsert, 90-98 쓰기, 34-35 쓰기, 64 COPY INTO 명령, 68-69 CREATE TABLE 문, 65 덮어쓰기 모 드, 68-76 델타 공유 GitHub 리포지토리, 213-215 delta-spark 2.4.0, 26 DeltaLog, 36 (트랜잭션 로그도 참조)

DeltaTableBuilder API, 50, 57-58
역사를 설명하다, 82-83, 127
 MERGE 연산 및, 97 차원 테이블, 7 차
 원 모델링, 7 스타 스키마
 차원 테이블, 7 팩트 테이블, 7

이산화된 스트림(DStream), 183
 DM(데이터 관리), 222 트랜잭션 레이어,
 222
 DML 작업, 81
 DROP COLUMN 명령, 169-171
 DStreams(이산 스트림), 183 내구성, 3

ERP(전사적 자원 관리) 시스템, 4
 ETL(추출, 변환 및 로드), 5
 CDF(변경 데이터 피드), 143개 증분 처리,
 183개 도구, 5개
 VACUUM, 140 단 한
 번만 처리, 183 실험, 시간 여행 및 125

F
 팩트 테이블, 7
 파일 보존, 134-136 파일, 물리
 적 삭제, 138 유연한 주문형 스토리지 계
 층, 220 형식, 13

G
 GCS(구글 클라우드 스토리지), 11, 209
 GDPR(일반 데이터 보호 규정),
 76
 GitHub
 델타 공유 저장소, 213-215
 골드 레이어, 메달리온 아키텍처, 237
 Google 클라우드 스토리지(GCS), 11, 209

시간
 하둡, 11
 HDFS(하둡 분산 파일 시스템), 11
 안녕하세요 스트리밍 월드
 체크포인트 파일, 193-195 쿼리

창조, 187-188
 QPL(쿼리 프로세스 로그), 188-193 읽기 변경
 사항, 185 스트리밍
 DataFrame, 185, 187 스트리밍 쿼리, 185

helloDeltaLake, 25, 29 달리
 기, 29-30 역사적 통
 찰, 8
 Hive 스타일 파티셔닝, 103
 HR(인적 자원), 4

I 증분 처리, 183 인덱스, Z 순서, 113 수
 집, 짧은 대기 시간, 183

INSERT 명령, 103 격리, 2

L
 레이크하우스, 218
 ACID 및, 14, 17 트랜잭
 션, 223 아키텍처, 델타
 기반, 19 감사 내역, 223 이점, 15-16 데이
 터 통합, 224 데이터 상호
 운용성, 224 설명, 14

청동층, 236
 금층, 237
 실버 레이어, 237 메
 타데이터 레이어, 223
 ML(머신러닝) 및 224 과제, 230-231 지원 기
 능, 231-233 스키마 시행,
 224 스키마 진화, 224

SQL 분석, 225 데이터 통
 합, 228 레이어, 228 쿼리 앤
 진 지원, 227 데
 이터 읽기, 228

Spark SQL, 225-227 스
 토리지 레이어, 218 클
 라우드 데이터 레이크 혜택, 220-222

데이터 유형, 218-219 시스

템 성능, 17 버전 관리, 223 유동 클

러스터링, 118 클래스

터 열 변경, 121 제거, 122 보
기, 121 쓰기 데이터, 120

활성화, 119-120 분

할 및, 118 경고 및

고려 사항,

122-123 로그 파일 보존,

136 낮은 대기 시간 수집,

183

N

표 형식이 아닌 데이터, ML(기계 학습), 230

NullType 열, 164-165

영향

OLAP(온라인 분석 처리) 도구, 6 온디스크 파티셔닝, 103 오픈

소스 델타 공유, 210

OPTIMIZE 명령, 110-113

Z-순서 및 114

ZORDER BY, 113-118

피

M

기계 학습(ML), 224

맵리듀스, 11

MDW(현대 데이터 웨어하우스), 19 메탈리온 아키텍

처, 21-22

레이크하우스, 234-236

청동총, 236

골드 레이어, 237

은총, 237

MERGE 명령, 81, 90, 103

MERGE 작업 데이터 세트,

91-92

기록 설명 및 97 프로세스, 98 사용 사례, 90

PARTITIONED BY 절, 104 열 기준 파티셔

닝, 108 고려 사

항, 108 데이터 건너뛰

기, 103

Hive 스타일, 103 액

체 클러스터링, 118 다중 열,

104 온디스크, 103

PARTITIONED BY 절, 104 성능 튜닝, 102

쿼리, 104

SHOW PARTITIONS, 105개 보기 파

티션, 105개 경고, 108개 파티션

MERGE 문, 92-95

일치하지 않는 행, 95-97 메타데이터, 5,

13, 19 맞춤

델타 테이블, 70 다중

열, 73-74 패턴, 확인, 74 대체 위

치, 75-76 단일 열, 71-72

성능 조정 압축 파일, 109 압

축, 109-113 데이터 건너

뛰기, 99 통제, 100-102 유동

클러스터링, 118 클래스

터링 열, 121-122 활성

화, 119-120

분할 및, 118 경고 및 고려 사항, 122-123 분할, 102 고려 사항, 108 데이터 건너뛰기 및, 103

Hive 스타일, 103 액체 클러스터링, 118 다중 열, 104 온디스크, 103 쿼리, 104

SHOW PARTITIONS, 105 보기 파티션, 105 경고, 108

Z 순서 Z ORDER BY 매개변수, 113-118 성능, DELETE 작업 및, 86

POS(판매 시점) 시스템, 4

PySpark, 25-27 DataFrameWriter API, 50, 54-57

델타 테이블, 읽기, 63-64 Python 커넥터, 213 설치, 213 프로필 파일, 213 공유 테이블, 읽기, 214-215

큐 QPL(쿼리 프로세스 로그), 188-193 타임스탬프 별 쿼리, 133 데이터 건너뛰기, 117 델타 테이블, 이전 버전, 132-134 파티션, 104 스트리밍 쿼리, 185, 187-188

R 원시 데이터, 5 RDBMS(관계형 데이터베이스 관리 시스템), 3가지 규정 준수, 시간 여행, 125 가지 관계형 데이터베이스, 이력, 2-3

RENAME COLUMN 명령, 169-172 REORG TABLE 명령, 174-179 복제, 저장, 220 보고 도구, 6 보고서, 시간 여행, 125 재현성, ML(기계 학습), 230 RESTORE 명령, 128 고려 사항 및 경고, 131-132 테이블 복원, 시간 여행, 128-129

를백, 시간 여행, 125 예스 S3(Amazon 단순 스토리지 서비스), 11 Scala 셸, 27-28 확장 성, 델타 공유, 210 스키마 열, 추가, 155-157

델타 테이블 컬럼 데이터 유형 변경, 179-181 컬럼 삭제, 174-177 컬럼 매핑, 169-171 컬럼 이름 변경, 179-181 컬럼 순서 변경, 168-169 컬럼 이름 변경, 171-172 컬럼 교체, 172-174 컬럼, 추가, 166, 167 열, 댓글, 167 시행, 19, 151, 154, 224

추가 열, 155-157 일치 스키마, 154 진화, 152, 157, 224 열 추가, 158-160 열 데이터 유형 변경, 162-164 DataFrame 누락 열, 160-162 NullType 열, 164, 165 규칙, 157

Spark 클러스터, 일치 157개, 쓰기 시 154 개, 검증 및 153-154 별 차원 테이블, 7 팩트 테이블, 7 스타 스키마, 9 저장, 152 트랜잭션 로그 항목, 152-153 업데이트, 165-181 검증, 152-157 읽기 스키마, 13 반구조화된 데이터, 레이크하우스, 219

SET 문, 78 소 파티션, 105 SHOW TBLPROPERTIES 명령, 137 실버 레이어, 메탈리온 아키텍처, 237 단일 스토리지 레이어, 220 SLA(서비스 수준 계약), 12개의 소스 기록

재처리, 200-201 구조적 스트리밍, 업데이트, 197-201

- Spark, 12개**
- 클러스터, 스키마 진화 및, 157 delta-spark 2.4.0,
 - 26개 이미지, 26 PySpark,
 - 25 Scala 셸,
 - 27-28 Spark
 - Scala 셸, 25 Spark
 - SQL, 레이크하우스, 225-227
 - Spark 구조적 스트리밍(구조적 스트리밍 참조)
 - SparkSession 빌더, 29 메타데이터, 사용자 정의, 77-78 SQL(구조적 쿼리 언어)
 - ALTER COLUMN 명령, 168-169
 - ALTER TABLE 명령, 121, 168-169
 - 테이블 변경 열 고체 명령, 172-174
 - ALTER TABLE...ADD COLUMN com - 맨드, 166-167
 - CLUSTER BY 명령, 121
 - COPY INTO 명령, 68-69
 - CREATE EXTERNAL TABLE 명령, 119
 - CREATE TABLE 명령, 50, 65
 - DDL(데이터 정의 언어) 명령
 - 델타 테이블, 50-53
 - 삭제 명령, 103
 - 델타 테이블, 읽기, 60-63
 - DESCRIBE 명령, 델타 테이블, 53-54
 - DESCRIBE HISTORY 명령, 127
 - DROP COLUMN 명령, 169-171
 - INSERT 명령, 103
 - MERGE 명령, 103
 - MERGE 문, 92-95
 - 일치하지 않는 행, 95-97
 - OPTIMIZE 명령, 110-113
 - ZORDER BY, 113-118
 - PARTITIONED BY 절, 104
 - RENAME COLUMN 명령, 169-172
 - REORG TABLE 명령, 174-179
 - RESTORE 명령, 128 고려 사항 및 경고, 131-132
 - SET 문, 78
 - 쇼 파티션, 105
 - SHOW TBLPROPERTIES 명령, 137
 - 업데이트 명령, 103
 - VACCUUM 명령, 138-143
- VACUUM 명령, 134, 174-177**
- SQL 분석 레이크하우스**
- 우스, 225 데이터 통합, 228 레이어, 228 쿼리 엔진 지원, 227 데이터 원, 228 읽기, 228
- Spark SQL, 225-227 스타일**
- 스키마, 9 팩트 테이블, 7 스토리지,
 - 12, 22 사용성, 221 비용, 221 분리된 스토리지 계층, 220 기술 통합, 220 스토리팅, 220 유연성, 온디맨드, 220 복제, 220 단일 스토리지 계층, 220 클라우드 데이터 레이크 이점, 220-222 데이터 유형 반정형 데이터, 219 정형 데이터, 219 비정형 데이터, 219 스트리밍 구성 변수, 194 정확히 한 번 처리, 183
- 구조적 스트리밍, 183 스트리밍 쿼리, 185 streamQuery 클래스, 199**
- 문자열, 사용자 정의, 77 구조적 데이터, 레이크하우스, 219**
- 구조적 스트리밍, 183**
- 사용 가능지금 옵션, 195-197
 - CDF(변경 데이터 피드), 읽기, 201-203
 - 델타 호수와, 184
 - 인녕하세요 스트리밍 월드
 - 체크포인트 파일, 193-195
 - QPL(쿼리 프로세스 로그), 188-193 변경 내용 읽기, 185 스트리밍 DataFrame, 185, 187 스트리밍 쿼리, 185 스트리밍 쿼리 생성, 187-188 소스 레코드 재처리, 200-201 업데이트, 197-201 streamQuery 클래스, 199 요약 데이터,
- 6

T

테이블

- 차원 테이블, 팩트 테이블 [7](#)
- 개, 기록 [7개](#),
- [137개](#)
- OPTIMIZE 명령, [113](#) 시간 여행 데이터
- 보관, [137](#) 데이
- 터 피드 변경, [143-150](#) 데이
- 터 보존, [134](#) 데이터 파일 보존,
- 134-137 로그 파일 보존,
- [136](#)

델타 테이블, 이전 버전 쿼리,
[132-134](#)

DESCRIBE HISTORY 명령, [127](#) 예, [126-134](#) 유지 관
리 작업, [140-141](#)

복원 명령

- 고려 사항 및 경고, [131-132](#) 시나리오, [125](#) 테이블 복
원, [128-129](#) 타임스
탬프, [129](#)
- VACUUM 명령 및, [138-139](#) 빈도, [140-141](#) 구문,
[139-140](#) 경고 및 고려 사
항, [141-143](#) 버전 기
록, [129](#) 트랜잭션 로그, [130](#) 시계열 데이터, 시간 여행
및, [125](#) 타임스탬프 쿼리,
[133](#) 복원, [129](#) 트랜잭션 로
그, [36](#) 원자 커밋, [36-37](#) 원자성 및 [36](#) 체크포인트 파
일, [30](#)

체크포인트 파일, [46-47](#)

- 델타 테이블, 최신 버전, [39](#) 실패 시나리오, [40](#) 파일,
[37-43](#) 메타데이터, 크기 조정,
[44-47](#) 스키마, 보
기, [152-153](#) SchemaString, [152](#) 업데이
트 시나리오, [41](#) 쓰기, 디중 대 단일 파일,
[38](#) _delta_log, [37](#)

U

- 구조화되지 않은 데이터
호수집, [219](#)
- 업데이트 명령, [81](#), [88-90](#), [103](#)
- UPDATE 작업 성능 튜닝, [90](#)
개 사용 사례, [87개](#)

V

- VACUUM 명령, [134](#), [138-143](#), [174-177](#) 공급업체, 데이터 공유
솔루션, [207-208](#) 버전 관리, [223](#)

여

- WHERE 절, [108](#)
OPTIMIZE 명령, [112](#)
- 쓰기 작업, [82](#)

지

- Z 순서 인덱스, [113](#)
- Z 순서
OPTIMIZE 명령 및 [114](#)
- ZORDER BY 매개변수, [113-118](#)

저자 소개

Bennie Haelen은 Microsoft와 Databricks 파트너인 Insight Digital Innovation의 수석 설계자입니다. Insight의 수석 설계자로서 Bennie의 주요 초점 분야는 다양한 상용 클라우드 플랫폼의 최신 데이터 웨어하우징, 기계 학습, AI 및 IoT입니다. Bennie는 의료, 공공 부문, 석유 및 가스, 금융 애플리케이션과 같은 다양한 애플리케이션 도메인에서 많은 데이터+AI 프로젝트를 감독했습니다. Bennie는 다양한 애플리케이션 도메인을 위해 Databricks, Spark Structured Streaming, Delta Lake 및 Microsoft Power BI를 사용하여 실시간 스트리밍 데이터 레이크하우스 애플리케이션을 설계하고 제공했습니다. Bennie는 비즈니스 인텔리전스, 데이터 과학 및 기계 학습을 지원하는 안전한 엔터프라이즈 규모 데이터 레이크하우스 기반 솔루션을 구현하는 데 있어 풍부한 실무 경험을 제공합니다. Bennie는 또한 전국 Microsoft 기술 센터에서 열리는 Databricks 이벤트에서 자주 연설했으며 Data+AI 2021 서밋에서도 연설했습니다.

Dan Davis는 데이터에서 분석 통찰력과 비즈니스 가치를 제공하는 10년의 경험을 보유한 클라우드 데이터 설계자입니다. Dan은 최신 도구와 기술을 사용하여 기업 규모의 온프레미스, 하이브리드 및 클라우드 환경에 대한 데이터 통합 및 분석을 지원하는 데이터 플랫폼, 프레임워크 및 프로세스를 설계하고 제공하는 전문 분야입니다.

출판사 마크

Delta Lake: Up and Running의 표지에 등장하는 동물은 인도토끼(*Lepus nigricollis*)입니다. 목덜미를 따라 자라는 검은 털 반점 때문에 검은 목덜미 산토끼라고도 불립니다.

목에 있는 검은 털 외에도 인도토끼에는 검은 꼬리도 있습니다. 목과 등은 대부분 갈색이고 검은 털이 약간 있으며 밑 부분은 흰색입니다. 다른 산토끼처럼 그들은 긴 귀와 털로 덮인 긴 뒷발을 가지고 있습니다. 길이는 16~28인치이고 무게는 3~15파운드까지 자랄 수 있습니다. 암컷은 수컷보다 큰 경우가 많습니다.

인도토끼는 인도 아대륙과 자바가 원산지입니다. 인도, 방글라데시, 인도네시아, 네팔, 스리랑카, 파키스탄에서 발견할 수 있으며 열대 우림, 초원, 관목지, 사막에 서식합니다. 우기에는 다양한 풀을 주로 먹으며, 건기에는 꽃이 피는 식물을 먹습니다. 그들은 또한 농경지에서 발견되며 그곳에서 농작물을 먹고 씨앗을 발아시킵니다.

인도토끼는 개체수가 풍부하며 멸종위기종 목록에서 가장 우려가 적은 종으로 간주됩니다. 이들의 존재에 대한 주요 위협은 사냥과 농업 확장으로 인한 서식지 손실입니다. O'Reilly 표지에 등장하는 많은 동물은 멸종 위기에 처해 있습니다. 그들 모두는 세상에 중요합니다.

표지 그림은 *Histoire Naturelle*의 골동품 선 조각을 바탕으로 Karen Montgomery가 그렸습니다. 표지 글꼴은 Gilroy Semibold 및 Guardian Sans입니다. 텍스트 글꼴은 Adobe Minion Pro입니다. 제목 글꼴은 Adobe Myriad Condensed입니다. 코드 글꼴은 Dalton Maag의 Ubuntu Mono입니다.