

O'REILLY®

# Delta Lake

## The Definitive Guide

Modern Data Lakehouse Architectures  
with Data Lakes

Early  
Release  
Raw & Unedited  
Sponsored by  
 databricks



Denny Lee,  
Prashanth Babu,  
Tristen Wentling & Scott Haines



---

# Delta Lake: 최종 가이드

데이터 레이크를 갖춘 최신 데이터 레이크하우스 아  
키텍처

초기 출시 eBook을 사용하면 가장 초기 형태의 책, 즉 저자가 집필하는 그대로 편집  
되지 않은 원본 콘텐츠를 얻을 수 있으므로 해당 타이틀이 공식 출시되기 훨씬 전에 이러한  
기술을 활용할 수 있습니다.

데니 리, 프라샨스 바부, 트리스틴 웬틀링, 스콧 해인즈

베이징·보스턴 파넘 세바스토플 도쿄

O'REILLY®

Delta Lake: 최종 가이드 - Denny Lee,  
Prashanth Babu, Tristen Wentling 및 Scott Haines

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: 그레고리 하이먼

인터리어 디자이너: David Futato

표지 디자이너: 카렌 몽고메리

일러스트레이터: 케이트 덜리아

2024년 5월: 초판

### 초기 릴리스의 개정 내역

2023-06-22: 첫 번째 릴리스

2023-10-16: 두 번째 릴리스

2023-11-08: 세 번째 릴리스

2024-02-29: 네 번째 릴리스

2024-06-05: 다섯 번째 릴리스

<http://oreilly.com/catalog/errata.csp?isbn=9781098151942>를 참조하세요. 릴리스 세부정보를 확인하세요.

O'Reilly 로고는 O'Reilly Media, Inc.의 등록 상표입니다. Delta Lake: The Definitive Guide, 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 그러한 라이선스 및/또는 권리를 준수하는지 확인하는 것은 귀하의 책임입니다.

---

# 목차

간략한 목차(아직 최종은 아님) .....	vii
<b>1. 델타 Lake 설치.....</b>	<b>9</b>
Delta Lake Docker 이미지 인터페	9
이스 선택 네이티브 Delta	10
Lake 라이브러리	16
다양한 바인딩 가능	16
설치	17
Delta Lake를 사용하는 Apache Spark	17
Apache Spark로 Delta Lake 설정	17
전제조건: Java 설정	18
대화형 셀 설정	18
PySpark 선언적 API	20
Databricks 커뮤니티 에디션	20
Databricks Runtime을 사용하여 클러스터 만들기	21
노트북 가져오기	24
노트북 연결	25
요약	26
<b>2. Delta Lake 생태계에 뛰어들어 보세요.....</b>	<b>27</b>
커넥터	28
아파치 플링크	29
플링크 데이터스트림 커넥터	29
커넥터 설치	30
델타소스 API	31
델타싱크 API	34
엔드투엔드 예시	37
카프카 델타 수집	39

커넥터 사용	40
트리노	43
시작하기	43
Trino 커넥터 구성 및 사용	47
쇼 카탈로그 사용	47
스키마 생성	48
스키마 표시	48
테이블 작업	49
테이블 작업	52
요약	55
<b>3. Delta Lake 유지 관리</b>	<b>57</b>
Delta Lake 테이블 속성 사용	58
속성이 포함된 빈 테이블 만들기	60
테이블 채우기	60
테이블 스키마 발전	62
테이블 속성 추가 또는 수정	64
테이블 속성 제거	65
델타 테이블 최적화	66
큰 테이블과 작은 파일의 문제	67
최적화를 사용하여 작은 파일 문제 해결	69
테이블 튜닝 및 관리	71
테이블 파티셔닝	71
테이블 생성 시 파티션 정의	72
분할되지 않은 테이블에서 분할된 테이블로 마이그레이션	73
테이블 데이터 복구, 복원 및 교체	74
테이블 복구 및 교체	75
데이터 삭제 및 파티션 제거	76
Delta Lake 테이블의 수명 주기	76
테이블 복원	77
청소	77
요약	79
<b>4. Delta Lake에서 스트리밍 및 스트리밍</b>	<b>81</b>
스트리밍 및 Delta Lake 스트리	82
밍 대 일괄 처리 델타를 소스 델타로 싱	82
크 델타 스트리밍 옵	88
션 입력 속도 제	89
한 업데이트 무시 또는 초기 처	91
리 위치 삭제	91
	92
	94

EventTimeOrder를 사용한 초기 스냅샷	96
Apache Spark를 사용한 고급 사용법	98
멱등성 스트림 쓰기	98
Delta Lake 성능 메트릭	103
자동 로더 및 델타 라이브 테이블	104
오토로더	104
델타 라이브 테이블	105
데이터 피드 변경	106
변경 데이터 피드 사용	107
개요	111
추가 생각	113
주요 참고자료	113
 5. 호수 별장 건축하기.....	115
레이크하우스 아키텍처 레이크하우스	116
란 무엇입니까?	116
데이터 웨어하우스에서 학습 데이터 레이크	117
에서 학습 이중 계층 데이터 아키텍처	117
처 레이크하우스 아키텍처 Delta Lake를	118
통한 기초	119
	121
개방형 생태계의 개방형 표준에 대한 오픈 소스	121
거래 지원	122
스키마 적용 및 거버넌스	124
메달리온 아키텍처	127
청동층 탐험	128
은층 탐험	131
금층 탐험	134
스트리밍 메달리온 아키텍처	136
Lakehouse 내에서 종단 간 지연 시간 줄이기	136
요약	137
 6. 성능 조정: Delta Lake를 사용하여 데이터 파이프라인 최적화.....	139
성능 목표 읽기 성능 최대화 쓰	140
기 성능 최대화	140
	142
성능 고려 사항	143
파티셔닝	144
테이블 유ти리티	146
테이블 통계	152
클러스터 기준	162
블룸 필터 지수	166

결론	168
<b>7. 성공적인 디자인 패턴.....</b>	<b>169</b>
컴퓨팅 비용 대폭 절감 고속 솔루션	170
선 스마트 장치 통합 효율적인	170
스트리밍 수집 스트리밍 수집	171
Delta Rust의 시작 복잡한 시스템 조정	177
Doordash의 운영 데이터	177
저장소 결합 결론 참고 자료	179
	183
	184
	187
	188
컴캐스트	188
스크리브	188
도어대시	188
<b>8. 레이크하우스 거버넌스 및 보안.....</b>	<b>189</b>
레이크하우스 거버넌스 레이크하우스	190
우스 거버넌스의 측면	192
데이터 거버넌스의 출현	194
데이터 제품 및 데이터 자산과의 관계	197
Lakehouse의 데이터 제품	198
데이터 자산 및 액세스	199
데이터 자산 모델	199
데이터 웨어하우스와 레이크 간의 거버넌스 통합 권한 관리 파일 시스템 권한 클라우드	202
우드 개체 저장소 액세스 제어 데이	203
터 보안 메타데이터 관리 메타데이	204
터 관리란?	205
	207
	218
	218
데이터 카탈로그 데	218
이터 흐름 및 계보 데이터 계보	222
데이터 공유 데이	222
터 수명주기 자동	226
화 감사 로깅 모니터링 및 경고 데이터	227
검색이란?	229
	230
	232
	232
요약	

---

## 간략한 목차(아직 최종본은 아님)

- 1장: Delta Lake란 무엇입니까? (없는)
- 2장: Delta Lake 설치(사용 가능)
- 3장: Delta Lake 사용(사용할 수 없음)
- 4장: 델타 공유(사용할 수 없음)
- 5장: Delta Lake 생태계 살펴보기(사용 가능)
- 6장: Delta Lake 유지 관리(사용 가능)
- 7장: 퇴적물 아래(델타 호수 내부)(사용할 수 없음)
- 8장: Delta Lake를 사용하여 네이티브 앱 빌드(사용할 수 없음)
- 9장: Delta Lake에서 스트리밍(사용 가능)
- 10장: 고급 기능(사용할 수 없음)
- 11장: 호수가 주택 설계하기(사용 가능)
- 12장: 성능 튜닝(사용 가능)
- 13장: 성공적인 디자인 패턴(사용 가능)
- 14장: 레이크하우스 거버넌스 및 보안(사용 가능)



## 제1장

# 델타 Lake 설치

초기 출시 독자를 위한 참고 사항 초기 출시 eBook을 사용

하면 가장 초기 형태의 책, 즉 저자가 집필하는 그대로의 편집되지 않은 원본 콘텐츠를 얻을 수 있으므로 이러한 타이틀이 공식 출시되기 훨씬 전에 이러한 기술을 활용할 수 있습니다.

이것은 마지막 책의 두 번째 장이 될 것입니다. GitHub 저장소는 나중에 활성화될 예정입니다.

이 책의 내용 및/또는 예제를 개선할 수 있는 방법에 대한 의견이 있거나 이 장에 누락된 자료가 있는 경우 [gobrien@oreilly.com](mailto:gobrien@oreilly.com)으로 편집자에게 문의하십시오.

이 장에서는 Delta Lake를 설정하고 첫 번째 독립 실행형 애플리케이션 작성을 시작하기 위한 간단한 단계를 안내합니다.

Delta Lake를 설치하는 방법에는 여러 가지가 있습니다. 이제 막 시작했다면 Delta Lake Docker(<https://go.delta.io/dockerhub>) 이미지가 포함된 단일 머신을 사용하는 것이 가장 좋습니다. 로컬 설치의 번거로움을 건너뛰려면 최신 버전의 Delta Lake가 포함된 Databricks Community Edition을 무료로 사용해 보세요. 이 장에서 논의되는 Delta Lake 사용을 위한 다른 옵션에는 Delta Rust Python 바인딩, Delta Lake Rust API 및 Apache SparkTM이 포함됩니다.

## 델타 Lake Docker 이미지

Delta Lake Docker는 Python, Rust, PySpark, Apache Spark 및 Jupyter Notebook을 포함하여 Delta Lake로 읽고 쓰는 데 필요한 모든 구성 요소가 포함된 독립형 이미지입니다. 기본 전제 조건은 Docker를 컴퓨터에 설치하는 것입니다.

로컬 머신; Docker 가져오기 의 단계를 따르세요 . 그런 다음 DockerHub (<https://go.delta.io/dockerhub>)에서 최신 버전의 Delta Lake Docker를 다운로드할 수 있습니다. 또는 Delta Lake Docker GitHub 리포지토리 (<https://go.delta.io/docker>)의 지침에 따라 직접 Docker를 구축할 수 있습니다 .

이는 이 책의 모든 코드 조각을 실행하는 데 선호되는 옵션입니다.

이 Docker 이미지에는 다음이 사전 설치되어 있습니다.

- **아파치 애로우:** Apache Arrow는 인메모리 분석을 위한 개발 플랫폼으로, 단순하고 계층적인 데이터를 위한 표준화되고 언어 독립적인 열형 메모리 형식과 이 형식으로 작업하기 위한 라이브러리 및 도구를 제공하는 것을 목표로 합니다. C, C++, C#, Go, Java, JavaScript, Julia, MATLAB, Python, R, Ruby 및 Rust와 같은 다양한 시스템 및 언어에서 빠른 데이터 처리 및 이동이 가능합니다. • **데이터프로젝트:** 2017년에 만들어져 2019년 Apache Arrow 프로젝트에 기증된 DataFusion은 Rust로 작성되고 Apache Arrow 인메모리 형식을 사용하는 고품질 데이터 중심 시스템을 구축하기 위한 매우 빠르고 확장 가능한 쿼리 엔진입니다.
- **ROAPI:** ROAPI(읽기 전용 API)는 Apache Arrow 및 DataFusion을 기반으로 구축된 도구이며 Delta Lake 및 기타 소스에 대한 읽기 전용 API를 자동으로 실행하는 코드가 없는 솔루션입니다.
- **녹:** Rust는 C 및 C++와 유사한 성능을 제공하지만 안전성과 메모리 관리에 중점을 둔 정적으로 유형이 지정되고 컴파일된 언어입니다. 가비지 수집기 없이 메모리 안전을 보장하는 고유한 소유권 모델로 알려져 있어 시스템 리소스에 대한 제어가 중요한 시스템 프로그래밍에 이상적입니다.



이 책에서는 macOS를 사용합니다. Windows를 실행하는 경우 git bash, WSL 또는 bash 명령으로 구성된 모든 셸을 사용할 수 있습니다.

## 인터페이스 선택

다음 각 인터페이스에 대해 자세히 설명하고 이러한 인터페이스를 사용하여 Delta Lake 테이블을 만들고 읽는 방법을 설명합니다.

- Python •
- Jupyter Lab 노트북 •
- PySpark 셸

- 스칼라 셸
- 델타 러스트 API
- 로피



### [도커 컨테이너 실행]

모든 docker 명령의 bash 진입점은 다음으로 시작합니다.  
낮춤 명령.

- Bash 셸을 엽니다.
- Bash 진입점을 사용하여 빌드 이미지에서 컨테이너를 실행합니다.  
다음 명령을 사용하여

```
docker run --name delta_quickstart --rm -it --
진입점 bash delta_quickstart
```

### Python용 델타 레이크

먼저 bash 셸을 열고 bash를 사용하여 빌드된 이미지에서 컨테이너를 실행합니다.  
진입 지점.

다음으로 Python 대화형 셸 세션 [python3] 과 다음 코드를 실행합니다.  
스나펫은 Python Pandas DataFrame을 생성하고, Delta Lake 테이블을 생성하고, 생성합니다.  
새로운 데이터, 이 테이블에 새로운 데이터를 추가하여 쓴 다음 마지막으로 읽은 다음  
이 Delta Lake 테이블의 데이터를 표시합니다.

팬더를 PD로 가져오기  
deltalake.writer에서 write\_deltalake 가져오기  
deltalake에서 DeltaTable 가져오기

```
df = pd.DataFrame(range(5))                                # 팬더 데이터프레임 생성
write_deltalake("/tmp/deltars_table", df) # 델타 레이크 테이블 쓰기
df = pd.DataFrame(range(6, 11))                           # 새로운 데이터 생성
write_deltalake("/tmp/deltars_table", \
    df, mode="추가")                                     # 새로운 데이터 추가
dt = DeltaTable("/tmp/deltars_table") dt.to_pandas()      # Delta Lake 테이블 읽기
# Delta Lake 테이블 표시
```

위 코드 조각의 출력은 다음 출력과 유사해야 합니다.

```
## 출력
0
0 0
1 1
...
8 9
9 10
```

이러한 Python 명령을 사용하여 첫 번째 Delta Lake 테이블을 만들었습니다. 이 테이블을 구성하는 기본 파일 시스템을 검토하여 이를 검증할 수 있습니다. 이렇게 하려면 Python 프로세스를 닫은 후 다음 ls 명령을 실행하여 /tmp/deltars-table에 저장한 Delta Lake 테이블 폴더 내의 콘텐츠를 나열할 수 있습니다.

```
$ ls -lsgA /tmp/deltars_table 총 12
```

```
4 -rw-r--r-- 1 NBuser 1610 4월 13일 05:48 0...-f3c05c4277a2-0.parquet 4 -rw-r--r-- 1 NBuser 1612 4월 13일 05:48
1..-674ccf40faae-0.parquet 4 drwxr-xr-x 2 NBuser 4096 4월 13일 05:48 _delta_log
```

.parquet 파일은 Delta Lake 테이블에 표시되는 데이터가 포함된 파일이고, \_delta\_log에는 Delta 의 트랜잭션 로그가 포함되어 있습니다. 이에 대해서는 이후 장에서 더 자세히 논의하겠습니다.

### JupyterLab 노트북

bash 셸을 열고 Jupyterlab 진입점이 있는 빌드된 이미지에서 컨테이너를 실행합니다.

```
docker run --name delta_quickstart --rm -it -p 8888-8889:8888-8889 delta_quick-start
```

이 명령은 JupyterLab 노트북 URL을 출력하고 해당 URL을 복사한 후 브라우저를 시작하여 노트북을 따라 각 셸을 실행합니다.

### PySpark 셸

bash 셸을 열고 bash 진입점을 사용하여 빌드된 이미지에서 컨테이너를 실행합니다.

```
docker run --name delta_quickstart --rm -it --entrypoint bash delta_quickstart
```

다음으로, PySpark 대화형 셸 세션을 시작합니다.

```
$SPARK_HOME/bin/pyspark --packages io.delta:$DELTA_PACKAGE_VERSION\ --conf
"spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension"\ --conf
"spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

셸에서 몇 가지 기본 명령을 실행해 보겠습니다.

```
# 스파크 데이터프레임 생성 data = Spark.range(0, 5)
```

```
# Delta Lake 테이블에 쓰기
```

```
(data .write .format("delta") .save("/
tmp/delta-table")
)
# Delta Lake 테이블에서 읽습니다. df = (spark
```

```

.read.format("델타").load("/tmp/delta-table").orderBy("id")

)
# Delta Lake 테이블을 표시합니다 df.show()

```

Delta Lake 테이블이 있는지 확인하려면 Delta Lake 테이블 폴더 내의 콘텐츠를 나열하면 됩니다. 예를 들어 이 전 코드에서는 /tmp/delta-table에 테이블을 저장했습니다. pyspark 프로세스를 닫은 후 Docker 셀에서 list 명령을 실행하면 아래와 비슷한 결과가 나타납니다.

```

$ ls -lsgA /tmp/delta-table 총 36 4 drwxr-xr-x 2
NBUser 4096
4월 13일 06:01 _delta_log 4 -rw-r--r-- 1 NBUser 478 4월 13일 06:01 part-00000-56a2c68a
-f90e-4764-8bf7- a29a21a04230-c000.snappy.parquet 4 -rw-r--r-- 1 NBUser 12 4월 13일 06:01 .part-00000-56a2c68a-
f90e-4764-8bf7- a29a21a04230-c000 .snappy .parquet.crc
4 -rw-r--r-- 1 NBUser 478 4월 13일 06:01 part-00001-bcbb45ab-6317-4229- a6e6-80889ee6b957-c000.snappy.parquet 4 -rw-r--r-- 1
NBUser 12 4월 13일 06:01 .part-00001-bcbb45ab-6317-4229-
a6e6-80889ee6b957-c000.snappy.parquet.crc 4 -rw-r--r-- 1 NBUser 478 4월 13일 06:01 부분 -00002-9e0efb76-
a0c9-45cf-90d6-0dba912b3c2f-c000.snappy.parquet 4 -rw-r--r-- 1 NBUser 12 4월 13일 06:01 .part-00002-9e0efb76-
a0c9-45cf-90d6-0dba912b3c2f-c000.snappy.parquet.crc 4 -rw-r--r-- 1 NBUser 486 4월 13일 06:01 part-00003-909fee02-574a-47ba-9a3b-
d531eec7f0d7-c000.snappy.parquet 4 -rw-r--r-- 1 NBUser 12 4월 13일 06:01 .part-00003-909fee02-574a-47ba-9a3b-d531eec7f0d7-
c000.snappy.parquet.crc

```

## 스칼라 셸

Bash 셸을 열고 Bash 진입점을 사용하여 빌드된 이미지에서 컨테이너를 실행합니다.

```
docker run --name delta_quickstart --rm -it --entrypoint bash delta_quickstart
```

Scala 대화형 셸 세션을 시작합니다.

```
$SPARK_HOME/bin/spark-shell --packages io.delta:${DELTA_PACKAGE_VERSION} \--conf
"spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension" \--conf "spark.sql.catalog.spark_catalog
=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

다음으로 셀에서 몇 가지 기본 명령을 실행합니다.

```
// Spark DataFrame 생성 val data =
Spark.range(0, 5)
// Delta Lake 테이블에 쓰기(데이터
    .쓰다
    .format("델타")
```

```

    .save("/tmp/델타-테이블")
)
// Delta Lake 테이블에서 읽기 val df =
(spark.read.format("delta").load("/tmp/
    delta-table").orderBy("id"))

)
// Delta Lake 테이블 표시 df.show()

```

Delta Lake 테이블을 확인하는 방법은 PySpark 셸을 참조하세요.  
부분.

### 델타 러스트 API

Bash 셸을 열고 Bash 진입점을 사용하여 빌드된 이미지에서 컨테이너를 실행합니다.

```
docker run --name delta_quickstart --rm -it --entrypoint bash delta_quickstart
```

다음으로, example/read\_delta\_table.rs를 실행하여 Delta Lake 테이블 메타데이터와 covid19\_nyt Delta Lake 테이블의 파일을 검토하세요. 이 명령은 작성된 파일 수와 절대 경로 등 유용한 출력을 나열합니다.

### CD RS

화물 실행 --예: read\_delta\_table

마지막으로 example/read\_delta\_datafusion.rs를 실행하여 DataFusion을 사용하여 covid19\_nyt Delta Lake 테이블을 쿼리합니다.

화물 실행 --예제 read\_delta\_datafusion

위 명령을 실행하면 covid19\_nyt Delta Lake 테이블의 스키마와 데이터 행 5개가 나열됩니다.

### 로피

Delta Lake 주변의 풍부한 개방형 생태계는 많은 새로운 유ти리티를 가능하게 합니다. 이러한 유ти리티 중 하나가 빠른 시작 컨테이너에 포함되어 있습니다: ROAPI(읽기 전용 API). ROAPI를 사용하면 한 줄의 코드도 필요 없이 정적 Delta Lake 데이터 세트에 대한 읽기 전용 API를 실행할 수 있습니다. 이 Docker에도 사전 설치된 ROAPI를 사용하여 Apache Arrow 및 DataFusion으로 Delta Lake 테이블을 쿼리할 수 있습니다.

Bash 셸을 열고 Bash 진입점을 사용하여 빌드된 이미지에서 컨테이너를 실행합니다.

```
docker run --name delta_quickstart --rm -it -p 8080:8080 --entrypoint bash
delta_quickstart
```

다음 nohup 명령을 사용하여 roapi API를 시작합니다. API 호출은 nohup.out 파일로 푸시됩니다.

컨테이너에 deltars\_table을 생성하지 않은 경우 다음을 통해 생성하세요.  
 위의 Python 옵션에 대한 deltalake입니다. 또는 다음을 생략할 수 있습니다.  
 명령에서: --table 'deltars\_table=/tmp/deltars\_table/,format=del  
 ta' 및 deltars\_table을 호출하는 모든 단계.

```
nohup roapi --addr=http 0.0.0.0:8080 --table 'deltars_table=/tmp/del-tars_table/,format=delta'  

--table 'covid19_nyt=/opt/spark/work-dir/rs/data/  

코로나19_NYT,format=delta' &
```

다른 셀을 열고 동일한 Docker 이미지에 연결합니다.

```
docker exec -it delta_quickstart /bin/bash
```



이전 단계에서 시작한 셀에서 아래 단계를 실행합니다.

두 Delta Lake 테이블의 스키마 확인

```
curl localhost:8080/api/스키마
```

위 명령의 출력은 다음 줄을 따라야 합니다.

```
{
  "covid19_nyt": {"필드": [{"이름": "날짜", "data_type": "Utf8", "nullable": true, "dict_id": 0, "dict_is_ordered": false}, {"name": "county", "data_type": "Utf8", "nullable": true, "dict_id": 0, "dict_is_ordered": false}, {"name": "state", "data_type": "Utf8", "nullable": true, "dict_id": 0, "dict_is_ordered": false}, {"name": "fips", "data_type": "Int32", "nullable": true, "dict_id": 0, "dict_is_ordered": false}, {"name": "cases", "data_type": "Int32", "nullable": true, "dict_id": 0, "dict_is_ordered": false}, {"name": "deaths", "data_type": "Int32", "nullable": true, "dict_id": 0, "dict_is_ordered": false}], "deltars_table": {"필드": [{"name": "0", "data_type": "Int64", "nullable": true, "dict_id": 0, "dict_is_ordered": false}]}}
```

deltars\_table을 쿼리합니다.

```
curl -X POST -d "SELECT * FROM deltars_table" localhost:8080/api/sql
```

위 명령의 출력은 다음 줄을 따라야 합니다.

```
[{"0":0}, {"0":1}, {"0":2}, {"0":3}, {"0":4}, {"0":6}, {"0":7}, {"0":8}, {"0":9}, {"0":10}]
```

covid19\_nyt Delta Lake 테이블을 쿼리합니다 .

컬 -X POST -d "COvid19\_nyt에서 사례, 카운티, 날짜를 선택하세요. ORDER BY 사례 DESC 제한 5" 로컬 호스트:8080/api/sql

위 명령의 출력은 다음 줄을 따라야 합니다.

```
[{"cases":1208672,"county":"로스앤젤레스","date":"2021-03-11"}, {"cases":1207361,"county":"로스앤젤레스","date":"2021-03-10"}, {"cases":1205924,"county":"로스앤젤레스","date":"2021-03-09"}, {"cases":1204665,"county":"Los Angeles","날짜":"2021-03-08"}, {"케이스":1203799,"카운티":"로스앤젤레스","날짜":"2021-03-07"}]
```

## 네이티브 델타 레이크 라이브러리

Rust의 Delta Lake 구현은 원래 **Scribd**에서 개발되었습니다. 더 빠르고 저렴한 스트리밍 데이터 수집 파이프라인을 구축합니다. Scribd는 개방형 프로토콜과 생태계 때문에 Delta Lake를 선택했지만 Apache Kafka에서 간단한 스트리밍 데이터를 수집하기에는 Apache Spark가 너무 무겁다는 사실을 발견했습니다. 변환이나 집계가 전혀 필요하지 않은 워크로드는 Rust 구현에 매우 적합했습니다. 라이브러리의 초기 버전은 **kafka-delta-ingest** 와 함께 공개적으로 개발되었습니다. 주로 QP Hou, Christian Williams 및 Mykhailo Osypov가 응용 프로그램을 작성했습니다. 최소한의 변경으로 Delta Lake 구현을 Python 생태계에 노출하는 Python 바인딩을 도입한 후 프로젝트가 극적으로 성장할 수 있었기 때문에 Rust의 선택은 매우 중요한 선택이었습니다. **Delta-rs**는 2020년 봄 창립 이후 프로젝트에는 거의 모든 대륙에서 거의 100명의 기여자가 참여했으며 Delta Lake를 크고 작은 수 많은 프로젝트에 도입하는 데 도움을 주었습니다.

## 다양한 바인딩 사용 가능 Rust 라이브

러리는 Delta Lake로 빌드할 수 있는 JVM 기반이 아닌 다른 라이브러리를 위한 강력한 기반을 제공합니다. 이러한 바인딩 중 가장 인기 있고 눈에 띄는 것은 DeltaTable 클래스를 노출하고 선택적으로 Pandas 또는 PyArrow와 원활하게 통합되는 Python 바인딩입니다. 이 글을 쓰는 시점에서 "deltalake" Python 패키지는 Python 버전 3.7 이상에서 구축 및 테스트되었으며 대부분의 주요 운영 체제 및 아키텍처에 쉽게 설치할 수 있도록 사전 구축된 많은 "휠"을 제공합니다.

Rust 라이브러리를 기반으로 여러 커뮤니티 바인딩이 개발되어 Delta Lake를 Ruby, Node 또는 기타 C 기반 커넥터에 노출합니다. 현재 Python 패키지에서 볼 수 있는 성숙도에 도달한 사람은 없습니다. 부분적으로는 다른 언어 생태계 중 Python 커뮤니티와 같은 데이터 도구에 대한 투자 수준을 본 적이 없기 때문입니다. Pandas, Polars, PyArrow, Dask 등은 개발자가 Delta 테이블을 읽고 쓸 수 있는 매우 풍부한 도구 세트를 제공합니다.

최근에는 추상화되는 커넥터를 위한 기본 Delta 라이브러리 인터페이스를 제공하는 것을 목표로 하는 소위 "Delta Kernel"에서 실험적인 작업이 있었습니다.

델타 프로토콜을 한 곳으로 통합합니다. 이 작업은 아직 초기 단계이지만 네이티브(예: C/C++) 및 상위 수준 엔진(예: Python, Node)에 대한 지원을 통합하여 모든 사람이 간단히 업그레이드하여 삭제 벡터와 같은 고급 기능의 혜택을 누릴 수 있을 것으로 예상됩니다. 기본 델타 커널 버전.

## 설치

Delta Lake는 [delta-rs](#)를 기반으로 네이티브 Python 바인딩을 제공합니다. [Pandas](#) 와 함께하는 프로젝트 완성. 이 Python 패키지는 다음 명령을 사용하여 쉽게 설치할 수 있습니다.

`pip`로 델타레이크 설치

설치 후 Python용 Delta Lake 섹션과 동일한 단계를 수행하고 해당 섹션의 코드 조각을 실행할 수 있습니다.

## Delta Lake를 사용하는 Apache Spark

Apache Spark는 대규모 데이터 세트의 처리 및 분석을 위해 설계된 강력한 오픈 소스 엔진입니다. 빠르고 다양한 분석을 배치 및 실시간으로 관리할 수 있도록 설계되었습니다. Spark는 포괄적인 클러스터 프로그래밍을 위한 인터페이스를 제공하여 암시적 데이터 병렬성과 내결함성을 제공합니다. 이는 인메모리 계산을 활용하여 MapReduce 작업에 대한 속도와 데이터 처리를 향상시킵니다.

Spark의 차별화된 기능 중 하나는 다국어 지원으로, 다양한 사용자에게 접근성을 제공합니다. 이를 통해 개발자는 Java, Scala, Python, R 및 SQL을 포함한 여러 언어로 애플리케이션을 구축할 수 있습니다. 또한 Spark에는 기계 학습, 스트림 처리 및 그래프 분석을 포함하여 광범위한 데이터 분석 작업을 가능하게 하는 수많은 라이브러리가 통합되어 있습니다. 이러한 특성은 Apache Spark를 대용량 데이터를 고속으로 효율적으로 처리하기 위한 기본 솔루션으로 자리매김합니다.

Spark는 주로 Scala로 작성되었지만 해당 API는 Scala, Python, Java 및 R에서 사용할 수 있습니다. 또한 Spark SQL을 통해 사용자는 SQL 또는 HiveQL 쿼리를 작성하고 실행할 수 있습니다. 신규 사용자의 경우 Python API 또는 SQL 쿼리를 탐색하여 Apache Spark를 시작하는 것이 좋습니다. Databricks에서 게시한 데이터에 따르면 SQL과 Python은 다양한 워크로드에 대한 고성능 시작점을 제공하므로 지난 몇 년 동안 사용이 급격히 증가했습니다.

Spark에 대한 자세한 소개는 [Spark 학습](#)을 확인하세요. 또는 [Spark: 최종 가이드](#).

## Apache Spark로 Delta Lake 설정 Apache Spark로

Delta Lake를 설정하려면 다음 지침을 따르십시오. 이 섹션의 단계는 다음 두 가지 방법 중 하나로 로컬 컴퓨터에서 실행할 수 있습니다.

### 대화형 실행

Delta Lake를 사용하여 Spark 셸(Scala 언어의 경우 `spark-shell` 또는 Python의 경우 `pyspark`)을 시작하고 셸에서 대화형으로 코드 조각을 실행합니다.

### 프로젝트로 실행 코드 조

각 대신 여러 파일에 코드가 있는 경우 모든 소스 파일과 함께 Delta Lake를 사용하여 Maven 또는 SBT 프로젝트(Scala 또는 Java)를 설정하고 프로젝트를 실행할 수 있습니다. [Github](#) 저장소에 제공된 예제를 사용할 수도 있습니다.



다음 모든 지침에 대해 Delta Lake 2.3.0과 호환되는 올바른 버전의 Spark 또는 PySpark를 설치했는지 확인하세요. [릴리스 호환성 매트릭스](#)를 확인하세요 자세한 내용은.

전체 조건: Java 설정 [여기](#) 공식 Apache

Spark 설치 지침에 언급된대로, 유효한 Java 버전(8, 11 또는 17)이 설치되어 있고 시스템 PATH 또는 JAVA\_HOME 환경 변수를 사용하여 시스템에 Java가 올바르게 구성되어 있는지 확인하십시오.

Windows 사용자는 이 [블로그](#)의 지침을 따라야 합니다. Delta Lake 2.3.0 이상과 호환되는 올바른 버전의 Apache SparkTM을 사용해야 합니다.

### 대화형 셸 설정 Spark SQL, Scala 또는

Python 셸 내에서 Delta Lake를 대화형으로 사용하려면 Apache Spark를 로컬에 설치해야 합니다. SQL, Python 또는 Scala 사용 여부에 따라 각각 SQL, PySpark 또는 Spark 셸을 설정할 수 있습니다.

### Spark SQL 셸

Spark SQL 명령줄 인터페이스(CLI)라고도 하는 Spark SQL 셸은 명령줄에서 직접 SQL 쿼리를 쉽게 실행할 수 있도록 설계된 대화형 명령줄 도구입니다.

[Spark 다운로드](#)의 지침에 따라 호환되는 Apache Spark 버전을 다운로드하세요. pip를 사용하거나 아카이브를 다운로드 및 추출하고 추출된 디렉토리에서 `Spark-sql`을 실행합니다.

```
bin/spark-sql --packages io.delta:delta-core_2.12:2.3.0 --conf \
"spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension" --conf \
"spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

Spark SQL 셸 프롬프트에서 다음을 복사하여 붙여넣으세요.

```
CREATE TABLE delta.`/tmp/delta-table` USING DELTA AS SELECT col1 ID FROM VALUES 0,1,2,3,4;
```

SQL 쿼리는 Spark SQL을 사용하여 첫 번째 Delta Lake 테이블 생성을 완료합니다.

위의 테이블에 기록된 데이터는 다른 간단한 방법으로 간단히 다시 읽을 수 있습니다.  
아래와 같은 SQL 쿼리:

```
SELECT * FROM delta.`/tmp/delta-table`;
```

## PySpark 셸

PySpark 명령줄 인터페이스라고도 알려진 PySpark 셸은 Python 프로그래밍 언어를 사용하여 Spark API 사용을 촉진하는 대화형 환경입니다. PySpark 예제를 학습하고, 테스트하고, 명령줄에서 직접 데이터 분석을 수행하기 위한 플랫폼 역할을 합니다. PySpark 셸은 REPL(Read Eval Print Loop)로 작동하여 PySpark 문을 신속하게 테스트할 수 있는 편리한 환경을 제공합니다.

명령 프롬프트에서 다음을 실행하여 Delta Lake 버전과 호환되는 PySpark 버전을 설치합니다.

```
pip install pyspark==<호환성-스파크-버전>
```

Delta Lake 패키지 및 추가 구성 사용하여 PySpark를 실행합니다.

```
pyspark --packages io.delta:delta-core_2.12:2.3.0 --conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension"  
--conf "spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

PySpark 셸 프롬프트에서 다음을 복사하여 붙여넣습니다.

```
데이터 = 스파크.범위(0, 5)  
data.write.format("델타").save("/tmp/delta-table")
```

코드 조각은 PySpark를 사용하여 첫 번째 Delta Lake 테이블 생성을 마칩니다.

위 표에 기록된 데이터는 아래와 같이 간단한 Pyspark 코드 조각을 사용하여 간단히 다시 읽을 수 있습니다.

```
df = Spark.read.format("delta").load("/tmp/delta-table") df.show()
```

## Spark Scala 셸

Spark Scala 명령줄 인터페이스(CLI)라고도 하는 Spark Scala 셸은 사용자가 Scala 프로그래밍 언어를 사용하여 Spark의 API와 상호 작용할 수 있는 대화형 플랫폼입니다. 이는 데이터 분석을 위한 강력한 도구이며 API 학습을 위한 접근 가능한 매체 역할을 합니다.

**호환되는 버전을 다운로드하세요** [Spark 다운로드](#) 의 지침에 따라 Apache Spark를 설치하세요 . pip를 사용하거나 아카이브를 다운로드 및 추출하고 추출된 디렉터리에서 Spark-shell을 실행합니다.

```
bin/spark-shell --packages io.delta:delta-core_2.12:2.3.0 --conf
  "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension" --conf
  "spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

Scala 셸 프롬프트에서 다음을 복사하여 붙여넣으세요.

```
값 데이터 = Spark.range(0, 5)
data.write.format("delta").save("/tmp/delta-table")
```

이 코드 조각은 Scala 셸을 사용하여 첫 번째 Delta Lake 테이블 생성을 마칩니다. 테이블에 기록된 데이터는 아래와 같이 간단한 PySpark 코드 조각을 사용하여 다시 읽을 수 있습니다.

```
val df = Spark.read.format("delta").load("/tmp/delta-table") df.show()
```

## PySpark 선언적 API

Apache Spark와 함께 Delta Lake를 사용하기 위한 Python API가 포함된 PyPi 패키지도 사용할 수 있습니다. 이는 Python 프로젝트를 설정하는 데 매우 유용할 수 있으며 더 중요하게는 단위 테스트에도 유용할 수 있습니다. Delta Lake는 다음 명령을 사용하여 설치할 수 있습니다.

pip 설치 델타 스파크

SparkSession은 Delta Lake에서 `configure_spark_with_delta_pip()` 유틸리티 함수를 사용하여 구성할 수 있습니다 .

```
from delta import *
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"
  ))
```

## Databricks 커뮤니티 에디션

Databricks는 [Databricks Community Edition](#)을 통해 개인용 플랫폼을 제공합니다 . 이는 노트북 및 번들 Spark 버전의 도움으로 Delta Lake를 학습하는 데 충분할 수 있는 15GB 메모리 클러스터를 제공합니다.

[databricks.com/try](https://databricks.com/try)로 이동하여 Databricks Community Edition에 등록하여 시작하세요.

양식에 세부 정보를 입력하고 계속을 클릭하세요. 등록 양식의 두 번째 페이지에 있는 "Community Edition 시작하기" 링크를 클릭하여 Community Edition을 선택하십시오.

계정을 성공적으로 생성한 후 이메일 주소를 확인하기 위한 이메일을 받게 됩니다. 인증을 완료해주세요. Databricks Community Edition에 로그인하면 [그림 1-1](#)과 유사한 Databricks 작업 영역이 표시됩니다.

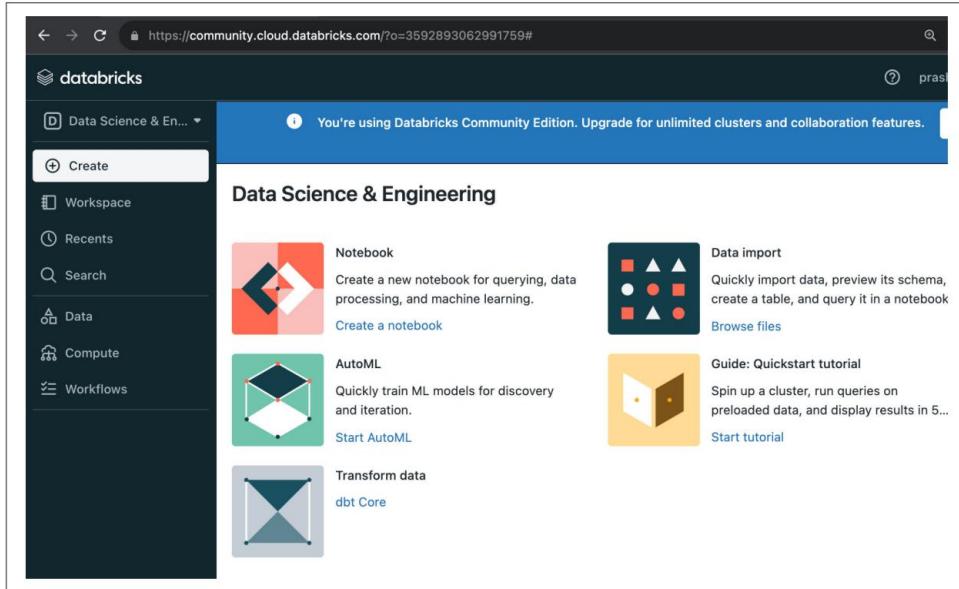


그림 1-1. 성공적으로 로그인한 후 Databricks Community Edition 방문 페이지

## Databricks Runtime을 사용하여 클러스터 만들기

왼쪽 창에서 Compute 메뉴 항목을 클릭하여 시작하세요. 생성한 모든 클러스터가 이 페이지에 나열됩니다. 그러나 이 계정에 처음 로그인하는 것이므로 이 페이지에는 [그림 1-2와 같이 아직 클러스터가 나열되지 않습니다.](#)

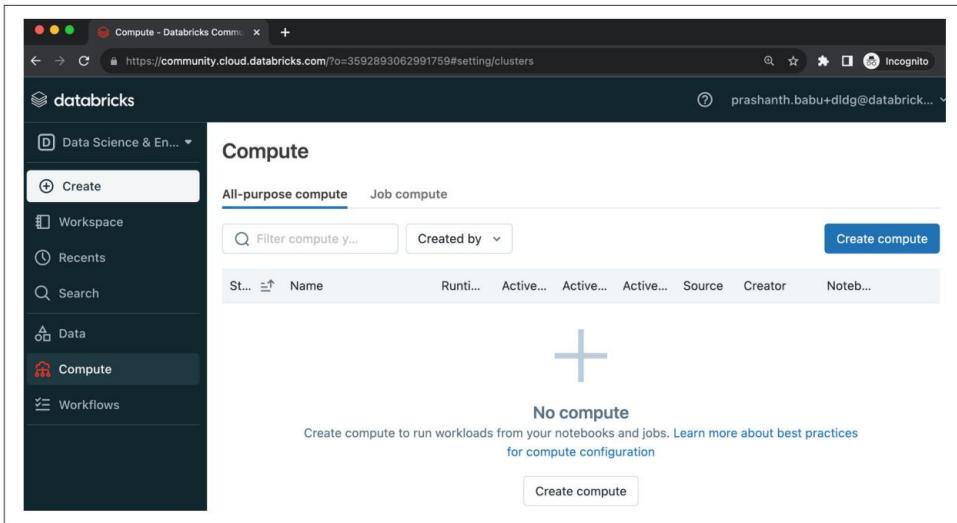


그림 1-2. Databricks Community Edition 클러스터 페이지

다음 페이지에서 Create Compute를 클릭하면 New Cluster 페이지로 이동합니다.

Databricks Runtime 13.3 LTS는 기본적으로(작성 당시) 선택되었습니다. 코드 실행을 위해 최신(LTS 권장) Databricks Runtime 중 하나를 선택할 수 있습니다.

이 경우 13.3 Databricks Runtime이 선택되었습니다 ([그림 1-3](#)). Databricks Runtime 릴리스 및 호환성 매트릭스에 대한 자세한 내용은 [Databricks 웹 사이트를 확인하세요](#). 선택한 클러스터 이름은 "Delta\_Lake\_DL0G"입니다. 원하는 이름을 선택하고 상단의 클러스터 생성 버튼을 눌러 클러스터를 시작하세요.

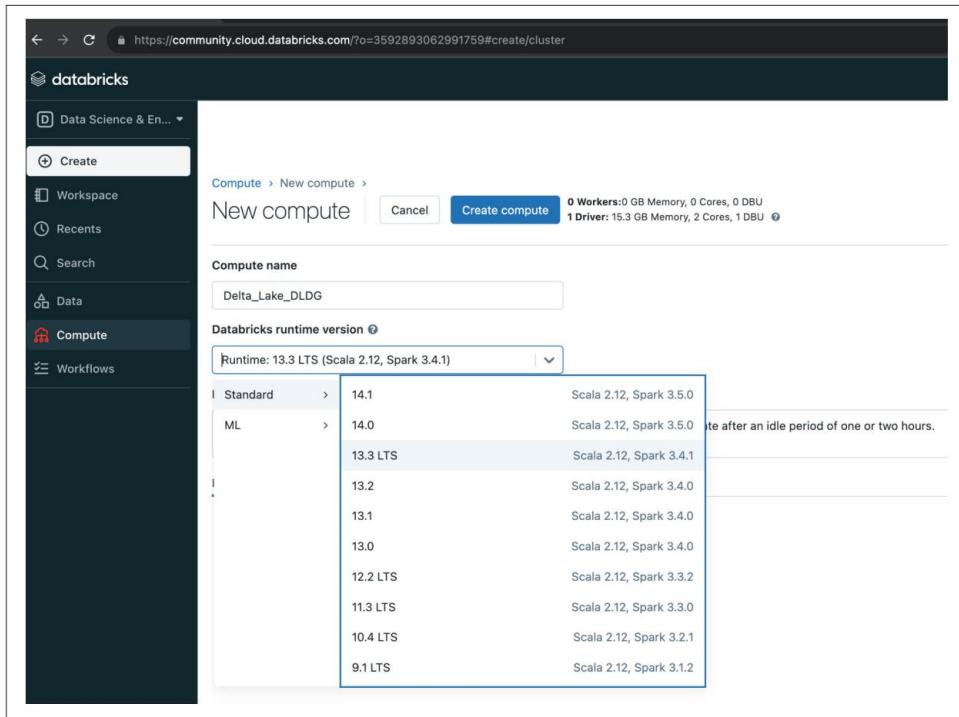


그림 1-3. Databricks Community Edition에서 클러스터에 대한 Databricks Runtime 선택



Databricks Community Edition 내에서는 한 번에 하나의 클러스터만 만들 수 있습니다. 이미 존재하는 경우 해당 항목을 사용하거나 삭제하여 새 항목을 만들어야 합니다.

그림 1-4 에 표시된 대로 클러스터가 몇 분 내에 가동되어 실행되어야 합니다 .

The screenshot shows the Databricks Compute interface. On the left, there's a sidebar with options like 'Create', 'Workspace', 'Recents', 'Search', 'Data', 'Compute' (which is selected and highlighted in red), and 'Workflows'. The main area is titled 'Compute' and shows a table for 'All-purpose compute'. The table has columns for State, Name, Runtime, Active ... (with three dots), Active ... (with three dots), Active ... (with three dots), and Source. One row is visible, showing 'Delta\_Lake\_DL DG' with a runtime of 13.3, 15 GB, 2 cores, and 1 UI.

그림 1-4. 클러스터 가동 및 실행 중



Databricks는 Databricks Runtime에 Delta Lake를 번들로 제공하므로 pip를 통해 또는 클러스터에 대한 패키지의 Maven 좌표를 사용하여 Delta Lake를 명시적으로 설치할 필요가 없습니다.

#### 노트북 가져오기 간결성과 이

해의 용이성을 위해 JupyterLab 노트북의 이전 섹션에서 본 Jupyter 노트북을 (재)사용하겠습니다. 이 노트북은 [여기](#) delta-docs GitHub 저장소에서 사용할 수 있습니다. 이 단계에서 이 노트북을 가져올 것이므로 노트북 링크를 복사하여 가까운 곳에 보관하세요.

Databricks Community Edition으로 이동하여 작업 영역, 사용자를 차례로 클릭한 다음 [그림 1-5](#)와 같이 전자 메일 옆에 있는 아래쪽 화살표를 클릭합니다.

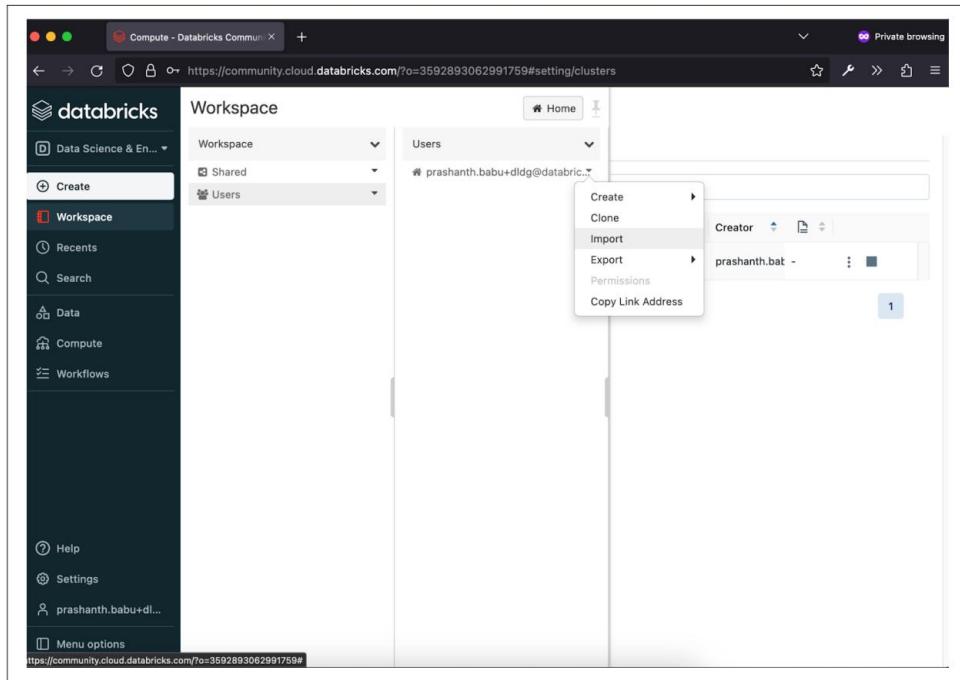


그림 1-5. Databricks Community Edition에서 노트북 가져오기

대화 상자에서 URL 라디오 버튼을 클릭하고 노트북 URL을 붙여넣은 후 가져오기를 클릭합니다. 그러면 Databricks Community Edition에서 Jupyter Notebook이 렌더링됩니다.

#### 노트북 연결 이제 이전에 생성

한 클러스터를 선택하여 이 노트북을 실행하세요. 이 경우에는 [그림 1-6](#)과 같이 “Delta\_Lake\_Rocks”이다.

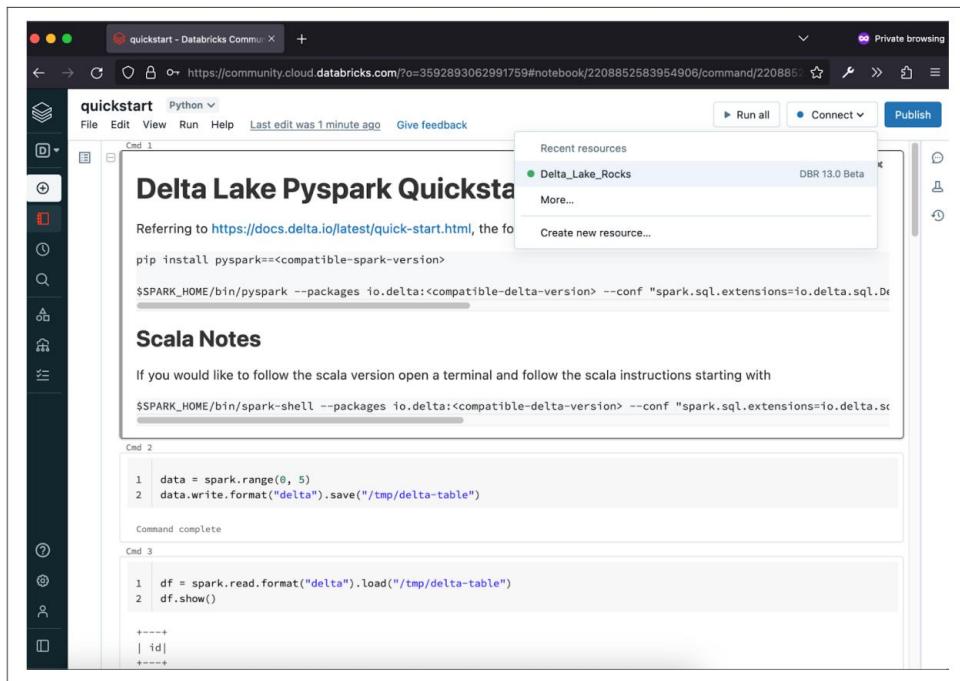


그림 1-6. 노트북을 연결하려는 클러스터를 선택하세요.

이제 노트북의 각 셀을 실행하고 키보드에서 **Ctrl + Enter**를 눌러 셀을 실행할 수 있습니다. Spark 작업이 실행 중이면 Databricks는 노트북에 직접 더 자세한 세부 정보를 표시합니다. 여기에서 Spark UI로 이동할 수도 있습니다.

이 노트북 내의 Delta Lake 테이블에 쓰고 읽을 수 있습니다.

## 요약

이 장에서는 Delta Lake를 시작하기 위해 취할 수 있는 다양한 접근 방식(Delta Docker, Delta Lake for Python, Apache SparkTM with Delta Lake, PySpark Declarative API, 마지막으로 Databricks Community Edition)을 다루었습니다. 이를 통해 간단한 노트북이나 명령 셀을 쉽게 실행하여 Delta Lake 테이블에 쓰고 읽을 수 있는 방법을 알 수 있습니다.

마지막으로, 매우 짧은 예를 통해 위의 접근 방식 중 하나를 사용할 수 있는 방법, Delta Lake를 설치하는 것이 얼마나 쉬운지 또는 Delta Lake를 사용할 수 있는 다양한 방법을 보여주었습니다. Delta Lake 테이블에 액세스 하기 위해 API를 통해 SQL, Python, Scala, Java 및 Rust 프로그래밍 언어를 사용할 수 있다는 것을 확인했습니다. 이는 다음 장인 Delta Lake 사용으로 이동하여 읽기, 쓰기에 대한 다양한 API를 더 자세히 검토합니다. 그리고 다른 많은 명령도 사용할 수 있습니다.

## 제 2 장

# Delta Lake 생태계 살펴보기

초기 출시 독자를 위한 참고 사항 초기 출시 eBook을 사용

하면 가장 초기 형태의 책, 즉 저자가 집필하는 그대로의 편집되지 않은 원본 콘텐츠를 얻을 수 있으므로 이러한 타이틀이 공식 출시되기 훨씬 전에 이러한 기술을 활용할 수 있습니다.

이것이 마지막 책의 다섯 번째 장이 될 것이다. GitHub 저장소는 나중에 활성화될 예정입니다.

이 책의 내용 및/또는 예제를 개선할 수 있는 방법에 대한 의견이 있거나 이 장에 누락된 자료가 있는 경우 [gobrien@oreilly.com](mailto:gobrien@oreilly.com)으로 편집자에게 문의하십시오.

지난 몇 장에 걸쳐 우리는 Spark 생태계의 편안함 속에서 Delta Lake를 살펴보았습니다. 그러나 Delta 프로토콜은 기본 테이블 형식뿐만 아니라 컴퓨팅 환경 내에서도 풍부한 상호 운용성을 제공합니다. 이는 테이블 진실의 단일 소스를 사용하여 레이크ハウス 애플리케이션을 구동할 수 있는 광대한 가능성의 세계로의 문을 열어줍니다. 이제는 고정관념에서 벗어나 커넥터 생태계를 살펴봐야 할 때입니다.

커넥터 생태계는 Delta를 거의 모든 곳에서 활용할 수 있도록 해주는 끊임없이 확장되는 프레임워크, 서비스 및 커뮤니티 중심 통합의 집합입니다. 상호 운용성에 대한 헌신을 통해 우리는 Spark 생태계 외부의 기술에 공동으로 쓴아부은 수년을 희생하지 않고도 성장하는 오픈 소스 커뮤니티가 제공하는 노고와 노력을 최대한 활용할 수 있습니다.

이 장에서는 기존 Spark 생태계 외부에서 Delta 기반 데이터 애플리케이션을 시험하는 방법을 배우면서 더 인기 있는 Delta 커넥터 중 일부를 살펴보겠습니다. Apache Spark로 많은 작업을 수행하지 않은 사람들을 위해,

이 장은 Apache Spark 없이 Delta Lake에 대한 사랑의 노래이고 커넥터 생태계가 어떻게 작동하는지 자세히 살펴보기 때문에 운이 좋습니다.

우리는 다음 통합을 다룰 것입니다<sup>1</sup>:

- 플링크 데이터스트림 커넥터
- 델타 카프카 수집
- 트리노 커넥터

이 장의 네 가지 핵심 커넥터 외에도 Apache Pulsar, Clickhouse, Finos Legend, Hopworks, Delta Rust, PrestoDB, StarRocks 및 Delta로의 일반 SQL 가져오기에 대한 지원은 이 글을 쓰는 시점에서 모두 사용할 수 있습니다.

커넥터란 무엇입니까? 우리는 다음에 그들에 대해 모두 배울 것입니다.

## 커넥터

인간으로서 우리는 스스로 한계를 정하는 것을 좋아하지 않습니다. 우리 중 일부는 모험심이 강하고 미래의 무한한 가능성에 대해 생각하는 것을 좋아합니다. 우리 중 다른 사람들은 삶에 대해 좀 더 직선적이고 좁은 접근 방식을 취합니다. 우리의 태도와 관계없이 우리 모두를 하나로 묶는 한 가지는 모험 추구, 참신함 추구, 스스로 결정을 내리려는 열망입니다. 탈출구가 없이 갇혀 있고 갇혀 있는 것보다 더 나쁜 것은 없습니다. 데이터 실무자의 관점에서 볼 때, 오늘 우리가 의존하는 것이 계약 재협상에 대한 두려움 없이 내일도 사용될 수 있다는 것을 아는 것도 좋은 일입니다! Delta Lake는 사람이 아니지만 오픈 소스 커뮤니티는 커뮤니티 전체의 다양한 요구와 요구에 응답했으며 누구도 Apache Spark 생태계에 직접 연결될 필요가 없도록 건강한 생태계가 생겨났습니다. JVM 또는 Python, Scala, Java와 같은 전통적인 데이터 중심 프로그래밍 언어 세트도 있습니다.

커넥터 생태계의 임무는 Delta 프로토콜과의 원활한 상호 운용성을 보장하는 것입니다. 그러나 시간이 지남에 따라 현재(델타 < 3.0) 커넥터 생태계의 단편화로 인해 델타 프로토콜의 여러 독립적 구현과 현재 커넥터 간의 차이가 발생했습니다. Delta 생태계의 미래를 위한 간소화된 지원을 위해 **Delta 커널** 7장과 8장에서 자세히 살펴보는 공통 인터페이스와 기대치를 구현하기 위해 도입되었습니다.

---

<sup>1</sup> 진화하는 통합의 전체 목록은 <https://delta.io/integrations/>를 참조하세요.



커널은 커넥터 API 구현에 대한 작업의 정확성과 표현의 자유를 보장하는 완벽한 읽기 수준 API 세트를 제공하며, 이후에는 표준 쓰기 수준 API가 제공됩니다. 이는 모든 커넥터의 동작이 동일한 입력 및 출력으로 동일한 작업 세트를 활용하는 동시에 각 커넥터가 긴 리드 타임 없이 새로운 기능을 신속하게 구현할 수 있음을 의미합니다.

작업을 트리거하는 위치에 관계없이 델타 테이블 형식 및 프로토콜과의 상호 운용성을 가능하게 하는 수 많은 커넥터 및 통합이 있습니다. 상호 운용성과 통합은 Delta 프로젝트의 핵심 원칙의 일부이며 Delta 3.0의 기능이자 Delta, Iceberg 및 Hudi에 대한 교차 테이블 지원을 제공하는 UniForm을 향한 추진을 도왔습니다.

다음 섹션에서는 Apache Flink, Trino, Kafka Delta Ingest를 포함하여 가장 널리 사용되는 커넥터를 살펴보고 Delta Rust API로 마무리하겠습니다.

선호하는 프레임워크에서 Delta를 활용하는 방법을 배우는 것은 단 몇 단계만 거치면 됩니다.

## 아파치 플링크

Apache Flink은 "모든 공통 클러스터 환경에서 실행되고 메모리 내 속도와 규모에 관계없이 계산을 수행하도록 설계된 무제한 및 제한된 데이터 스트림에 대한 상태 저장 계산을 위한 프레임워크 및 분산 처리 엔진"입니다. 즉, Flink는 대규모로 확장할 수 있으며, 오류가 발생하거나 중단되는 경우에도 스트림 처리를 위해 정확히 한 번의 미리온( CheckpointingMode에 지정된 경우)을 준수하면서 분산 방식으로 증가하는 모든 로드를 처리하면서 효율적으로 계속 수행할 수 있습니다. 애플리케이션의 런타임.



이전에 Flink를 사용해 본 적이 없다면 Stream Process with Apache Flink (<https://www.oreilly.com/library/view/stream-processing-with/9781491974285/>)라는 훌륭한 책이 있습니다. Fabian Hueske와 Vasiliki Kalavri가 제작한 이 책을 통해 곧바로 최신 정보를 얻을 수 있습니다.

여기서 앞으로의 가정은 우리가 애플리케이션을 컴파일할 만큼 Flink에 대해 충분히 이해하고 있거나 진행하면서 기꺼이 따라가고 배울 것이라는 것입니다. 그렇다면 Flink 애플리케이션에 Delta-flink 커넥터를 추가하는 방법을 살펴보겠습니다.

### 플링크 데이터스트림 커넥터

**플링크/델타 커넥터는 Delta Standalone 위에 구축됩니다.** 라이브러리를 제공하며 DataStream 및 Table API와 같은 Flink 기본 요소를 사용하여 Delta 테이블을 읽고 쓰기 위한 원활한 추상화를 제공합니다. 실제로 Delta Lake는 Parquet를 다음과 같이 사용하기 때문에

공통 데이터 형식이므로 Delta Standalone 라이브러리에 도입된 기능 외에 Delta 테이블 작업에 대해 특별히 고려해야 할 사항은 없습니다.

독립 실행형 라이브러리는 지정된 테이블의 전체 현재 버전을 읽거나, 특정 버전에서 읽기를 시작하거나, 제공된 ISO를 기반으로 테이블의 대략적인 버전을 찾기 위해 DeltaLog에서 제공하는 메타데이터를 읽는 데 필요한 필수 API를 제공 합니다. 8601 타임스탬프. 다음 섹션에서는 DeltaSource 및 DeltaSink 사용 방법을 배우면서 독립 실행형 라이브러리의 기본 기능을 다룰 것입니다 .



다음 섹션에서 참조되는 전체 Java 애플리케이션은 책의 /ch05/applications/flink/dldg-flink-delta-app 아래 git 저장소에 있습니다 .

호기심 많은 독자를 위한 후속 조치로 애플리케이션에 대한 단위 테스트를 통해 Delta 독립 실행형 API를 사용하는 방법을 엿볼 수 있습니다.

#### 커넥터 설치 모든 것은 커넥터에서

시작됩니다. Maven을 사용하여 Delta-flink 커넥터를 추가하기만 하면 됩니다 . **그래들**, 또는 **SBT** 귀하의 데이터 응용 프로그램에. 다음 예에서는 **delta-flink**를 포함하는 방법을 보여줍니다. Maven 프로젝트의 커넥터 종속성.

```
<종속성>
<groupId>io.delta</groupId>
<artifactId>delta-flink</artifactId> <version>$
{delta-connectors-version}</version> </dependent>
```



delta-connectors-version 속성 값은 새 버전이 출시되면 변경됩니다. 이 글을 쓰는 시점에는 소스 코드 위치 변경을 고려하여 버전이 0.6.0에서 3.0.0rc1로 올라갔습니다. Delta 3.0 릴리스의 경우 이제 모든 커넥터가 기본 Delta 저장소에 공식적으로 포함됩니다.



Apache Flink가 공식적으로 Scala 프로그래밍 언어에 대한 지원을 중단한다는 점은 주목할 가치가 있습니다. 이 장의 내용은 공식적으로 더 이상 Scala API를 개시하지 않는 Flink 1.17.1을 사용하여 작성되었습니다. Flink에서 Scala를 계속 사용할 수 있지만 Flink 2.0 릴리스로 이동하면 Java와 Python만 지원됩니다. 따라서 책의 GitHub에 있는 모든 예제와 애플리케이션 코드는 Java로 작성되었습니다.

커넥터는 Delta Lake에 대한 읽기 및 쓰기를 위한 클래스와 함께 제공됩니다. 읽기는 DeltaSource API에 의해 처리 되고 쓰기는 DeltaSink API에 의해 처리됩니다 . 잘

DeltaSource API로 시작하여 DeltaSink API로 이동한 다음 엔드투엔드 애플리케이션을 살펴보세요.

## 델타소스 API

DeltaSource API 는 제한된 또는 연속적인 데이터 흐름을 위한 소스를 쉽게 구성할 수 있는 정적 빌더를 제공합니다. 소스의 두 가지 변형 간의 가장 큰 차이점은 소스 델타 테이블의 제한된(일괄) 작업 또는 제한되지 않은(스트리밍) 작업과 관련됩니다. 이 두 처리 모드 간의 동작은 다르지만 구성 매개변수는 약간만 다릅니다. 제한된 소스부터 살펴보고 연속 소스로 마무리하겠습니다. 더 많은 구성 옵션이 포함되어 있기 때문입니다.

### 경계 모드

DeltaSource 개체를 생성하기 위해 DeltaSource 클래스의 정적 forBoundedRowData 메서드를 사용합니다. 이 빌더는 예제 2-1에 표시된 것처럼 Delta 테이블에 대한 경로와 애플리케이션의 hadoop 구성 인스턴스를 사용합니다.

### 실시예 2-1. DeltaSource Bounded Builder 만들기

```
% 경로 소스 테이블 = 새 경로("s3://bucket/delta/table_name")
구성 hadoopConf = 새 구성() var 빌더:
RowDataBoundedDeltaSourceBuilder = DeltaSource.forBoundedRowData(
    소스테이블
    hadoopConf);
```

예제 2-1에서 반환된 개체는 빌더입니다. 빌더의 다양한 옵션을 사용하여 읽기 속도를 늦추고 읽은 열 집합을 필터링하는 등의 옵션을 포함하여 델타 테이블에서 읽는 방법을 지정합니다.

빌더 옵션. 다음 옵션은 빌더에 직접 적용할 수 있습니다.

#### 열 이름(문자열 ...)

이 옵션을 사용하면 읽고 싶은 테이블의 열 이름만 지정하고 나머지는 무시할 수 있습니다. 이 기능은 열이 많은 넓은 테이블에서 특히 유용하며, 사용되지 않는 열에 대한 메모리 압박을 완화하는 데 도움이 될 수 있습니다.

```
% builder.columnNames("event_time", "event_type", "brand", "price"); builder.columnNames(
    Arrays.asList("event_time", "event_type", "brand", "price"));
```

#### 시작 버전(긴)

이 옵션은 우리에게 정확한 버전을 지정하는 기능을 제공합니다.  
숫자 형태로 읽기 시작하는 델타 테이블의 거래 내역

긴. 이 옵션은 StartingTimestamp 옵션과 상호 배타적입니다. 둘 다 델타 테이블에 커서(또는 트랜잭션 시작 지점)를 제공하는 수단을 제공하기 때문입니다.

```
% builder.startingVersion(100L);
```

#### 시작타임스탬프(문자열)

이 옵션은 ISO-8601 문자열 형식으로 읽기를 시작할 대략적인 타임스탬프를 지정하는 기능을 제공합니다.

이 옵션은 지정된 타임스탬프 또는 그 이후에 생성된 테이블의 일치하는 버전을 찾는 델타 트랜잭션 기록 스캔을 트리거합니다. 전체 테이블이 제공된 타임스탬프보다 최신인 경우 테이블을 완전히 읽습니다.

```
% builder.startingTimestamp("2023-09-10T09:55:00.001Z");
```

타임스탬프 문자열은 "2023-09-10" 과 같은 간단한 날짜만큼 정밀도가 낮거나 위의 예와 같이 밀리초 정밀도로 시간을 나타낼 수 있습니다. 두 경우 모두 작업을 수행하면 테이블 시간의 특정 지점에서 델타 테이블을 읽게 됩니다.

#### parquetBatchSize(정수)

내부 배치당 반환할 행 수 또는 Flink 엔진 내에서 생성된 분할을 제어하는 정수를 사용합니다.

```
% builder.option("parquetBatchSize", 5000);
```

제한된 소스 생성. 빌더에 옵션 제공을 마치면 빌드를 호출하여 DeltaSource 인스턴스를 생성합니다.

```
% 최종 DeltaSource<RowData> 소스 = builder.build();
```

제한된 소스가 구축되었으므로 이제 테이블에서 Delta Lake 레코드의 배치를 읽을 수 있습니다. 하지만 새 레코드가 도착할 때 지속적으로 처리하려면 어떻게 해야 할까요? 그런 경우에는 연속 모드 빌더를 사용하면 됩니다!

#### 연속 모드

DeltaSource 개체의 이러한 변형을 만들기 위해 DeltaSource 클래스의 ContinuousRowData 메서드에 대한 정적 메서드를 사용합니다. 빌더는 다음과 같습니다.

**예제 2-2** 및 forBoundedRowData 빌더와 마찬가지로 배치에서 스트리밍으로 매우 간단하게 전환할 수 있는 동일한 기본 매개변수를 제공할 수 있습니다.

#### 실시예 2-2. DeltaSource Continuous Builder 생성

```
% var builder = DeltaSource.forContinuousRowData(sourceTable,
    hadoopConf);
```

위에서 반환된 객체는 RowDataContinuousDeltaSource Builder 의 인스턴스이며 제한된 변형과 마찬가지로 StartingVersion 또는 StartingTimestamp를 기반으로 Delta 테이블 내의 초기 읽기 위치를 제어하는 옵션과 Flink가 테이블에서 새 항목을 확인하는 빈도입니다.

빌더 옵션. 다음 옵션은 연속 빌더에 직접 적용할 수 있으며, 추가로 제한된 빌더의 모든 옵션(columnNames, StartingVersion 및 StartingTimestamp)도 연속 빌더에 적용됩니다.

### updateCheckIntervalMillis(긴)

이 옵션은 델타 테이블의 업데이트를 확인하는 빈도를 나타내는 긴 숫자 값을 사용하며 기본값은 5000밀리초입니다.

```
% builder.updateCheckIntervalMillis(60000L);
```

스트리밍 중인 테이블이 주기적으로만 업데이트된다는 것을 안다면 본질적으로 불필요한 IO를 줄일 수 있습니다. 예를 들어, 새 데이터가 1분 간격으로만 기록된다는 것을 알고 있다면 잠시 쉬면서 매분 확인하도록 빈도를 설정할 수 있습니다. 델타 테이블의 동작에 따라 더 빠르게 또는 더 느리게 처리해야 하는 경우 언제든지 수정할 수 있습니다.

### 무시 삭제(부울)

이 옵션을 설정하면 삭제된 행을 무시할 수 있습니다. 스트리밍 애플리케이션은 과거의 데이터가 제거되었다는 사실을 전혀 알 필요가 없을 수도 있습니다. 실시간으로 데이터를 처리하는 경우 테이블의 데이터 피드를 추가 전용으로 간주하면 테이블의 헤드에 집중하고 데이터가 오래됨에 따라 테일 변경 사항을 안전하게 무시할 수 있습니다.

### 변경 무시(부울)

이 옵션을 설정하면 삭제된 행을 포함하여 업스트림에서 발생하는 테이블 변경 사항과 물리적 테이블 데이터 또는 논리적 테이블 메타데이터에 대한 기타 수정 사항을 무시할 수 있습니다. 테이블을 새 스키마로 덮어쓰지 않는 한, 테이블 구조에 대한 수정 사항을 무시하고 계속 처리할 수 있습니다.

연속 소스 생성. 빌더 구성이 완료되면 빌드를 호출하여 DeltaSource 인스턴스를 생성합니다.

```
% 최종 DeltaSource<RowData> 소스 = builder.build();
```

DeltaSource 객체를 구축하는 방법과 커넥터 구성 옵션을 살펴봤습니다. 하지만 테이블 스키마나 파티션 열 검색은 어떻습니까? 다행히 둘 다 테이블 메타데이터를 사용하여 자동으로 검색되므로 너무 자세히 설명할 필요가 없습니다.

## 테이블 스키마 검색

Flink 커넥터는 Delta 테이블 메타데이터를 사용하여 모든 열과 해당 유형을 확인합니다. 예를 들어 소스 정의에 열을 지정하지 않으면 기본 델타 테이블의 모든 열이 읽혀집니다. 그러나 Delta 소스 작성기 메서드(columnNames)를 사용하여 열 이름 컬렉션을 지정하면 기본 Delta 테이블에서 해당 열 하위 집합만 읽어집니다. 두 경우 모두 소스 커넥터는 델타 테이블 열 유형을 검색하여 해당 Flink 유형으로 변환합니다. 내부 델타 테이블 데이터(마루 행)에서 외부 데이터 표현(Java 유형)으로의 이러한 변환 프로세스는 데이터 세트를 사용하는 원활한 방법을 제공합니다.

## DeltaSource 사용하기

DeltaSource 객체(제한적 또는 연속적)를 구축한 후 이제 StreamExecutionEnvironment의 인스턴스를 사용하여 DataStream의 스트리밍 그래프에 소스를 추가할 수 있습니다.

**예제 2-3에서는** 간단한 실행 환경 인스턴스를 생성하고 fromSource를 사용하여 스트림 소스(DeltaSource)를 추가합니다.

### 실시예 2-3. DeltaSource에 대한 StreamExecutionEnvironment 생성

```
% 최종 StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setRuntimeMode(RuntimeExecutionMode.AUTOMATIC);
env.enableCheckpointing(2000, CheckpointingMode.EXACTLY_ONCE);

DeltaSource<RowData> 소스 =
...
env.fromSource(source, WatermarkStrategy.noWatermarks(), "델타 테이블 소스")
```

이제 Delta를 지원하는 Flink 작업에 대한 라이브 데이터 소스가 있습니다. 추가 소스를 추가하고, 데이터를 결합 및 변환하고, 변환 결과를 DeltaSink를 사용하여 Delta에 다시 기록하거나 애플리케이션에서 요구하는 다른 곳으로 쓸 수도 있습니다. 다음으로 DeltaSink 사용을 살펴보고 전체 엔드투엔드 예제로 점들을 연결해 보겠습니다.

## 델타싱크 API

DeltaSink API는 Delta Lake로 쉽게 송신할 수 있는 정적 빌더를 제공합니다. DeltaSource API와 동일한 패턴에 따라 DeltaSink API는 빌더 클래스를 제공합니다.

빌더를 구성하는 방법은 [예제 2-4에 나와 있습니다.](#)

## 실시예 2-4. DeltaSink 빌더 만들기

```
% 경로 deltaTable = 새 경로("s3://bucket/delta/table_name")
구성 hadoopConf = 새 구성()
행 유형 행 유형 = ...
RowDataDeltaSinkBuilderinkBuilder = DeltaSink.forRowData(sourceTable,
    hadoopConf,
    rowType);
```

Delta-flink 커넥터의 빌더 패턴은 이 시점에서 이미 친숙하게 느껴질 것입니다. 이 빌더 제작의 유일한 차이점은 RowType 참조를 추가한다는 것입니다.

### RowType

Spark의 StructType과 유사하게 RowType은 주어진 논리적 행 내의 필드에 대한 논리적 유형 정보를 저장합니다. 더 높은 수준에서 이를 간단한 DataFrame의 관점에서 생각할 수 있습니다. 이는 동적 데이터 작업을 더 간단하게 만드는 추상화입니다.

보다 실질적으로 DataStream 의 DeltaSink 이전에 발생한 소스 또는 변환에 대한 참조가 있는 경우 간단한 트릭을 사용하여 RowType을 동적으로 제공할 수 있습니다. 몇 가지 캐스팅 트릭을 통해 예제 2-5 에서 볼 수 있듯이 TypeInformation<T> 와 RowData<T> 간의 변환을 적용할 수 있습니다 .

## 실시예 2-5. TypeInformation을 통해 RowType 추출

```
% 공공 RowType getRowType(TypeInformation<RowData> typeInfo) {
    InternalTypeInfo<RowData> sourceType = (InternalTypeInfo<RowData>) typeInfo; return (RowType)
        sourceType.toLogicalType();
}
```

getRowType 메소드는 제공된 typeInfo 객체를 InternalTypeInfo 로 변환하고 RowType 으로 다시 캐스팅할 수 있는 toLogicalType 을 사용합니다 . Flink의 RowData의 성능을 이해하기 위해 다음 예제 2-6 에서 이 메서드를 사용하는 방법을 살펴보겠습니다 .

## 실시예 2-6. DeltaSource에서 RowType 추출

```
% DeltaSource<RowData> 소스 = ...
TypeInformation<RowData> typeInfo = source.getProducedType(); RowType
rowTypeForSink = getRowType(typeInfo);
```

간단한 스트리밍 애플리케이션이 있다면 POJO를 수동으로 작성하고 직렬 변환기 및 역직렬 변환기로 작업하는 데 많은 시간을 소비하지 않고 한동안 잘 지낼 수 있었을 가능성이 있습니다. 아니면 대체 메커니즘을 사용하기로 결정했을 수도 있습니다.

Avro 또는 프로토콜 버퍼와 같은 데이터 개체를 생성하기 위한 Nism입니다. 또한 기존 데이터베이스 테이블을 외부의 데이터로 작업할 적이 없을 수도 있습니다. 사용 사례가 무엇이든 열 형식 데이터로 작업한다는 것은 SQL 쿼리에서와 동일한 방식으로 원하는 열을 간단히 읽을 수 있다는 의미입니다.

다음 SQL 문을 살펴보세요.

% 사용자로부터 이름, 나이, 국가를 선택합니다.

`select *`를 사용하여 테이블의 모든 열을 읽을 수 있지만 테이블에서 필요한 것만 가져오는 것이 항상 더 좋습니다. 이것이 바로 열 기반 데이터의 장점입니다. 데이터 애플리케이션에 모든 것이 필요하지 않을 가능성이 높다는 점을 고려하여 계산 주기와 메모리 오버헤드를 절약하고 읽는 데이터 소스 간에 깔끔한 인터페이스를 제공합니다.

Delta Lake 테이블을 통해 특정 열(SQL 프로젝션이라고 함)을 동적으로 읽고 선택하는 기능은 데이터 레이크에 있는 모든 데이터에 대해 항상 말할 수 있는 것이 아닌 테이블의 스키마를 신뢰할 수 있음을 의미합니다. 테이블 스키마는 시간이 지남에 따라 변경될 수 있고 변경될 예정이지만 소스 테이블을 나타내기 위해 별도의 POJO를 유지 관리할 필요는 없습니다. 이것은 큰 리프트차림 보이지 않을 수도 있지만 움직이는 부분의 수가 적을수록 데이터 애플리케이션을 작성, 릴리스 및 유지 관리하는 것이 더 간단해집니다. 우리가 읽는 델타 테이블이 이전 버전과 호환되는 스키마 진화를 사용한다는 것을 신뢰할 수 있는 한, 유연한 데이터 처리 애플리케이션을 만드는 능력을 가속화하기 위해 우리가 가질 것으로 예상되는 열만 표현하면 됩니다. 스키마 진화에 대한 자세한 내용은 6장을 참조하세요.

## 빌더 옵션 다

음 옵션은 빌더에 직접 적용할 수 있습니다.

`withPartitionColumns(문자열 ...)`

이 빌더 옵션은 열의 하위 집합을 나타내는 문자열 배열을 사용합니다.

열은 스트림에 물리적으로 존재해야 합니다.

`withMergeSchema(부울)`

자동 스키마 진화를 선택하려면 이 빌더 옵션을 `true`로 설정해야 합니다.

빌더 옵션 외에도 delta-flink 커넥터를 사용하여 정확히 한 번만 쓰기의 의미를 다루는 것이 좋습니다.

## 정확히 한 번 보장 DeltaSink

는 Delta 테이블에 즉시 쓰지 않습니다. 오히려 행은 `flink.streaming.sink.filesystem.DeltaPendingFile`에 추가됩니다 (델타 Lake와 혼동하지 마세요). 이러한 파일은 커밋할 수 있는 일련의 누적 변경 사항으로 파일 시스템에 대한 쓰기(델타)를 버퍼링하는 메커니즘을 제공하기 때문입니다.

함께. 보류 중인 파일은 체크포인트 간격이 충족될 때까지 쓰기 위해 열린 상태로 유지되며 (예 2-7은 Flink 애플리케이션에 대한 체크포인트 간격을 설정하는 방법을 보여줍니다) 보류 중인 파일은 롤오버됩니다. DeltaLog에 커밋되어야 합니다. 데이터 스트림에서 검사점을 활성화할 때 제공된 간격을 사용하여 Delta Lake에 대한 쓰기 빈도를 지정합니다.

### 실시예 2-7. 체크포인트 간격 및 모드 설정

```
% StreamExecutionEnvironment
```

```
.getExecutionEnvironment().enableCheckpointing(2000, CheckpointingMode.EXACTLY_ONCE);
```

위의 체크포인트 구성을 사용하면 최대 2초마다 새 트랜잭션을 생성하며, 이 시점에서 DeltaSink는 Flink 애플리케이션 appId 및 보류 중인 파일과 연결된 checkpointId를 사용합니다. 이는 멱등성 쓰기에 txappId 및 txnVersion을 사용하는 것과 유사하며 향후 통합될 가능성이 높습니다.

### 엔드투엔드 예제 Flink

DataStream API를 사용하여 Kafka에서 읽고 Delta Lake에 쓰는 엔드투엔드 예제를 살펴보겠습니다. 애플리케이션 소스 코드 및 도커 호환 환경은 책의 ch05 아래 저장소에 제공됩니다. 여기에는 ecomm.v1.clickstream Kafka 주제를 초기화하고, Flink 애플리케이션에서 사용할 코드를 작성 (생성)하고, 궁극적으로 해당 코드를 Delta에 작성하는 단계가 포함됩니다.. 애플리케이션 실행 결과는 그림 2-1과 같으며, Flink UI와 애플리케이션의 최종 상태를 나타냅니다.

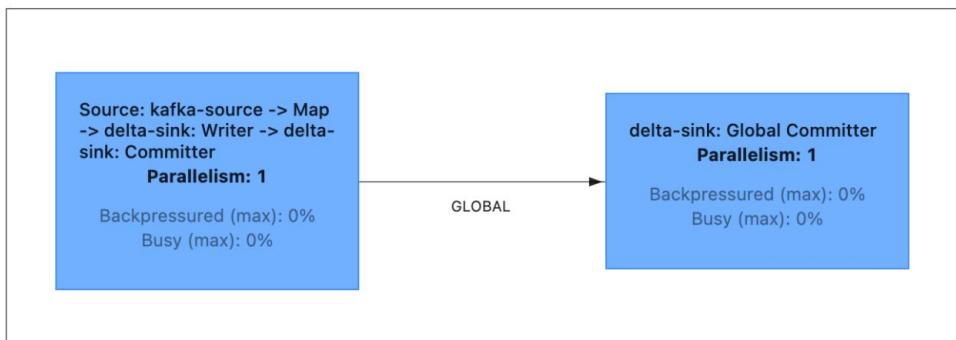


그림 2-1. Delta Sink에 Kafka 소스 쓰기

예제 2-8 의 범위 내에서 이 섹션 앞부분의 KafkaSource 커넥터와 DeltaSink를 사용하여 DataStream을 정의해 보겠습니다 .

### 실시예 2-8. Kafka에서 DeltaSink DataStream으로

```
% 공공 DataStreamSink<RowData> createDataStream(
    StreamExecutionEnvironment env)에서 IOException이 발생합니다.

final KafkaSource<Ecommerce> source = this.getKafkaSource(); final DeltaSink<RowData> 싱크 =
this.getDeltaSink(Ecommerce.ECOMMERCE_ROW_TYPE);

최종 DataStreamSource<Ecommerce> 스트림 = env
    .fromSource(소스, WatermarkStrategy.noWatermarks(), "kafka-소스");

반환 스트림
    .map((MapFunction<Ecommerce, RowData>) 전자 상거래::convertToRowData) .setParallelism(1) .sinkTo(sink) .name("delta-
sink") .setDescription("Delta
Lake에 쓰
기!") .setParallelism(1) ;

}
```

이 예에서는 전자상거래를 JSON 형식으로 나타내는 Kafka의 이진 데이터를 사용합니다. 뒤에서는 json 데이터를 전자상거래 행으로 역직렬화한 다음 JVM 개체에서 델타 테이블에 쓰는 데 필요한 일부 RowData 표현으로 변환합니다. 그런 다음 데이터 스트림에 대한 터미널 지점을 제공하기 위해 DeltaSink 인스턴스를 사용합니다.

다음으로, 예제 2-9에서 볼 수 있듯이 결과 DataStreamSink에 몇 가지 추가 설명 메타데이터를 추가한 후 간단히 실행을 호출합니다.

### 실시예 2-9. 엔드투엔드 예제 실행

```
% public void run()이 예외를 발생시킵니다.
{ StreamExecutionEnvironment env = this.getExecutionEnvironment(); DataStreamSink<RowData> 싱크 =
createDataStream(env); sing .name("delta-
sing") .setParallelism(NUM_SINKS) .setDescription("Delta Lake에 씁니다");

env.execute("kafka-to-delta-sink-job"); }
```

Delta Lake용 Flink 커넥터를 사용하는 방법에 대해 표면적으로만 살펴봤지만 Flink 생태계 외부에서 탐색할 내용이 더 많이 있습니다.



전체 작업 예제를 살펴보려면 ch05/README.md 아래의 단계별 개요를 따르거나 ch05/applications/flink를 살펴보세요.

Flink를 사용한 엔드투엔드 예제와 비슷한 맥락에서 다음에는 Kafka에서 동일한 전자상거래 데이터를 수집하는 방법을 탐색할 것입니다. 하지만 이번에는 Rust 기반 kafka-delta-ingest 라이브러리를 사용할 것입니다.

## 카프카 델타 수집

커넥터 이름은 이 작고 강력한 라이브러리가 수행하는 작업을 정확하게 요약합니다. Kafka 주제에서 레코드 스트림을 읽은 다음 선택적으로 각 레코드(데이터 스트림)를 원시 바이트에서 역직렬화된 json 또는 avro 페이로드로 변환하고 마지막으로 데이터를 Delta 테이블에 씁니다. 그 이면에는 최소한의 사용자 제공 구성이 각 특정 사용 사례를 충족하도록 커넥터를 구성하는 데 도움이 됩니다. kafka-delta-ingest 클라이언트의 단순성으로 인해 데이터 엔지니어링 수명 주기의 가장 중요한 단계 중 하나인 Delta Lake를 통해 Lakehouse로 초기 데이터를 수집하는 데 필요한 노력 수준을 줄입니다.

Kafka는 2011년부터 오픈 소스 커뮤니티에 있었지만 수집 라이브러리에 들어가기 전에 기본 사항을 언급하는 것이 좋습니다. 기본 Kafka 구성 요소 및 아키텍처에 이미 익숙하고 커넥터를 작동시키는 방법을 이해하고 싶다면 다음 섹션 "커넥터 사용"으로 건너뛰시기 바랍니다.

### 간단히 말해서 Kafka의 Apache

Kafka는 실시간 데이터 피드를 처리하기 위해 처리량이 높고 대기 시간이 짧은 통합 플랫폼을 제공하는 분산 이벤트 저장소 및 스트림 처리 프레임워크입니다.

Kafka 아키텍처는 테이블로 구성되보다는 주제 개념을 기반으로 구축되었습니다. 델타 테이블과 유사한 방식으로, 각 항목에는 (저장 공간 및 클러스터 활용 비용을 희생하면서) 무제한으로 확장할 수 있는 기능이 있습니다.

각 Kafka 주제는 클러스터 내의 여러 브로커로 분할되며, 각 클러스터는 포함된 구성 주제의 요구 사항을 충족하도록 확장될 수 있습니다.

분산 케이크의 진정한 장점은 Kafka가 ISR(동기화 복제본)이라는 것을 사용하여 고가용성과 내결 함성 주제를 가능하게 하는 간단한 구성을 통해 안정성이 매우 높다는 것입니다. 각 복제본은 각각의 고유한 Kafka 주제 내에 하나 이상의 파티션의 전체 복사본을 저장하므로 브로커가 삭제되는 경우(오프라인으로 전환되거나 네트워크 파티셔닝을 통해 사용할 수 없게 되는 경우) Kafka 주제는 다른 브로커를 대신하도록 위임할 수 있습니다. 클러스터를 리드하고 새로운 브로커가 해당 주제의 추가 사본을 받기 위해 한 발 더 나아갈 수 있습니다. 이런 식으로 다음을 수행할 수 있습니다.

전체 클러스터에서 치명적인 오류가 발생하지 않는 한 특정 주제를 통해 흐르는 데이터가 손실되지 않도록 보장합니다. 그런 일이 발생하면 데이터 위험을 완화하기 위해 좋은 재해 복구(DR) 계획이 설정되기를 바랄 뿐입니다. 손실).

마지막으로 Kafka를 특히 시계열 데이터에 매우 유용하게 만드는 몇 가지 불변성이 있습니다. 각 Kafka 토픽에는 토픽이 모든 파티션에서 삽입 순서를 조정할 필요 없이 각 토픽 파티션 내에서 동기 삽입을 보장하는 기능이 있습니다. 이는 클러스터가 정상 상태에서 실행될 때 스트림 처리 복잡성을 줄이는 이벤트 순서를 신뢰할 수 있음을 의미합니다. 이는 말할 것도 없이 당연한 일이지만, 시계열 데이터로 작업할 때 비용이 많이 드는 다시 읽기 및 정렬이 필요하지 않으므로 Kafka 소스를 통해 델타 테이블에 제공된 데이터로 작업할 때 안심하고 분석할 수 있습니다. 이제 kafka-delta-ingest 커넥터로 돌아갑니다.

### **커넥터 사용 커넥터 Kafka 데이**

터를 Delta Lake 테이블로 스트리밍하는 일반적인 단계를 단순화하는 데몬을 제공합니다. 다음 네 가지 간단한 단계를 통해 시작할 수도 있습니다.

- 러스트 설치
- 프로젝트 구축 • 델타 테이블 생성 • 수집 흐름 실행

### **러스트 설치**

이는 Rustup 툴체인을 사용하여 수행할 수 있습니다.

```
% 월 --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | 쉬
```

Rustup이 설치 되면 Rustup 업데이트를 실행하면 최신 안정 버전의 Rust를 사용할 수 있습니다.

### **프로젝트 빌드 이 단**

계에서는 소스 코드에 액세스할 수 있는지 확인합니다.

### **git 명령줄을 사용하**

여 프로젝트를 복제합니다. 커넥터를 복제하기만 하면 됩니다.

```
% git clone git@github.com:delta-io/kafka-delta-ingest.git \ && cd kafka-delta-ingest
```

### **로컬 환경 설정 프로젝트 디렉토리의 루**

트에서 docker 설정 유틸리티를 실행합니다.

### % docker 구성 설정

설정 흐름이 완료되면 localstack (로컬 aws), kafka (redpandas), 합류 스키마 레지스트리 및 로컬 Azure 스토리지용 (azurite)이 생성됩니다. 클라우드 기반 워크플로우를 로컬에서 실행하기 위해 액세스하면 애플리케이션의 설계 단계에서 프로덕션으로 이동하는 데 따른 어려움이 크게 줄어듭니다.

### 커넥터 구축 Rust는 의존성 관

리와 프로젝트 구축을 위해 카고를 사용합니다. 화물 유필리티는 Rustup 툴체인을 통해 설치됩니다. 프로젝트 루트에서 다음 명령을 실행합니다.

### % 화물 빌드

이 시점에서 우리는 커넥터를 구축하고 Rust 종속성을 설치하며 예제를 실행하거나 자체 Kafka 브로커에 연결하여 시작할 수 있습니다. kafka-delta-ingest 사용에 대한 마지막 섹션에서는 엔드투엔드 수집 실행을 다룹니다.



전체 수집 흐름 애플리케이션은 ch05/rust/kafka-delta-ingest에서 사용할 수 있습니다.

### 수집 흐름 실행 수집 애플리케이션이

작동하려면 소스 Kafka 주제와 대상 Delta 테이블이라는 두 가지가 필요합니다. 특히 Apache Spark 기반 Delta 워크플로에 의숙한 경우 Delta 테이블 생성 시 주의 사항이 있습니다. 주의할 점은 수집 흐름을 성공적으로 실행하려면 먼저 대상 델타 테이블을 만들어야 한다는 것입니다.

kafka-delta-ingest 애플리케이션을 수정할 수 있는 몇 가지 변수가 있습니다.

**표 2-1**의 기본 환경 변수 둘러보기로 시작한 다음 **표 2-2에서는** 이 커넥터를 사용할 때 사용할 수 있는 일부 런타임 변수(args)를 제 공합니다.

### 표 2-1. 환경 변수 사용

환경 변수	설명	기본
KAFKA_BROKERS	로커 문자열입니다. 이는 로컬 테스트나 분류 및 복구 애플리케이션을 위해 브로커 위치를 딛어쓰는 데 사용될 수 있습니다.	로컬호스트:9022
AWS_ENDPOINT_URL	LocalStack을 통해 로컬 테스트를 실행하는 데 사용됩니다.	없음
AWS_ACCESS_KEY_ID	애플리케이션 ID를 제공하는 데 사용됩니다.	시험
AWS_SECRET_ACCESS_KEY	애플리케이션 ID를 인증하는 데 사용됩니다.	시험

환경 변수	설명	기본
AWS_DEFAULT_REGION	LocalStack을 실행하거나 별도의 s3 버킷을 부트스트랩하는 데 유용할 수 있습니다. 없음 위치.	

## 표 2-2. 명령줄 인수 사용

인수	설명	예
allowed_latency	버퍼를 채우고 새 데이터를 기다리는 시간 처리	--allowed_latency 60
app_id	LocalStack을 통해 로컬 테스트를 실행하는 데 사용됩니다.	--app_id 수집-앱
auto_offset_reset	가장 빠르거나 가장 늦을 수 있습니다. 다음 내용을 읽는 경우 영향을 받습니다. Kafka 주제의 꼬리 또는 머리.	--auto_offset_reset 가장 이른
검문소	처리된 각 수집에 대한 Kafka 메타데이터를 기록합니다. 일괄. 이를 통해 애플리케이션을 쉽게 중지하고 데이터 손실 없이 다시 시작하세요. (Kafka가 삭제하지 않는 한 실행 사이의 데이터 - 삭제 시 확인할 수 있음 주제에 대한 정책입니다.)	--체크포인트
소비자_그룹_ID	Kafka 브로커에 대한 고유 소비자 이름을 제공합니다. 브로커는 group_id를 사용하여 처리를 분산할 수 있습니다. 여러 소비자 애플리케이션 간의 대규모 주제 중복 없이.	--consumer_group_id ecomm-ingest-app
max_messages_per_batch	당 메시지 수를 조절하려면 이 옵션을 사용하십시오. 애플리케이션 텍(루프). 이는 귀하의 애플리케이션에 도움이 될 수 있습니다. 예기치 않게 메모리가 증가하면 메모리가 부족해집니다. 주제에 기록되는 레코드의 양.	--max_messages_per_batch 1600
min_bytes_per_file	이 옵션을 사용하여 기본 델타 테이블이 작은 파일로 가득 차 있지 않습니다.	--min_bytes_per_file 64000000
카프카	Kafka 브로커 문자열을 수집 애플리케이션에 전달하는 데 사용됩니다. --kafka 127.0.0.1:29092	

이제 수집 애플리케이션을 실행하는 일만 남았습니다. 우리가 실행하는 경우 환경 변수를 사용하여 애플리케이션을 적용한다면 가장 간단한 명령은 Kafka 주제와 Delta 테이블 위치를 제공합니다. 명령 서명은 다음과 같습니다

다음과 같습니다.

```
% 화물 실행 수집 <topic> <delta_table_location>
```

다음으로 전체 예제를 살펴보겠습니다.

```
% 화물 운송 \
    ecomm.v1.clickstream 파일:///dldg/ecomm-ingest/ \ 수집
    --allowed_latency 120 \
    --app_id clickstream_ecomm \
    --auto_offset_reset 가장 빠른 \
    --체크포인트 \
    --kafka 'localhost:9092' \
    --max_messages_per_batch 2000 \
    --transform '날짜: substr(meta.producer.timestamp, `0`, `10`)' \
    --transform 'meta.kafka.offset: kafka.offset' \
```

```
--transform 'meta.kafka.partition: kafka.partition' \
--transform
'meta.kafka.topic: kafka.topic'
```

우리가 함께 살펴본 간단한 단계를 통해 이제 Kafka 주제에서 데이터를 쉽게 수집할 수 있습니다. 우리는 우리 데이터를 소비하는 사람들이 높은 수준의 신뢰성을 갖도록 보장함으로써 성공을 준비했습니다. 더 많이 자동화할수록 사람의 실수로 인해 사고가 발생하거나 무서운 데이터 손실이 발생할 가능성이 낮아집니다.

다음으로 트리노(Trino)를 탐험해보겠습니다. 두 가지 이전 예제는 보다 전통적인 SQL 도구를 통해 분석할 수 있는 견고한 테이블을 작성하기 전에 데이터를 수집하고 변환하는 노력 수준을 줄이므로 Trino 생태계와 함께 훌륭하게 작동합니다.

## 트리노

Trino는 수많은 데이터 소스에 원활하게 연결하고 상호 운용하도록 설계된 분산 SQL 쿼리 엔진입니다. Delta Lake를 기본적으로 지원하는 커넥터 생태계를 제공합니다.



Trino는 Presto 프로젝트의 커뮤니티 지원 포크이며 처음에는 Facebook에서 자체적으로 설계 및 개발되었습니다. Trino는 2020년에 이름을 얻기 전까지 PrestoSQL로 알려졌습니다.

Trino에 대해 자세히 알아보려면 Trino: 최종 가이드 (<https://learning.oreilly.com/library/view/trino-the-definitive/9781098137229/>)를 확인하세요.

### 시작하기 Trino 및

Delta Lake를 시작하는 데 필요한 것은 버전 373보다 최신 버전의 Trino뿐입니다. 이 글을 쓰는 시점에서 Trino는 현재 버전 427입니다.

### 커넥터 요구 사항

Delta 커넥터는 기본적으로 Trino 배포판에 포함되어 있지만 마찰 없는 경험을 보장하기 위해 고려해야 할 추가 사항이 여전히 있습니다.

OSS 또는 Databricks Delta Lake에 연결:

- Databricks Runtime 7.3 LTS, 9.1 LTS, 10.4 LTS, 11.3 LTS 및 12.2 LTS에서 작성된 테이블이 지원됩니다.
- AWS, HDFS, Azure Storage 및 Google Cloud Storage(GCS)를 사용한 배포가 완벽하게 지원됩니다.
- 코디네이터 및 작업자에서 Delta Lake 스토리지로의 네트워크 액세스. • Delta Lake의 Hive 메타스토어 서비스(HMS) 또는 별도의 HMS에 액세스합니다.

- 코디네이터와 작업자의 HMS에 대한 네트워크 액세스. 포트 9083은 HMS에서 사용하는 Thrift 프로토콜의 기본 포트입니다.

Docker를 사용하여 로컬로 작업:

- Trino 이미지 •

HMS(Hive Metastore Service)(독립형) • HMS 테이블

속성, 열, 데이터베이스 및 기타 구성을 저장하기 위한 Postgres 또는 지원되는 RDBMS(단순화를 위해 RDS와 같은 관리형 RDBMS를 가리킬 수 있음)

- Amazon S3 또는 MinIO(관리형 데이터 웨어하우스를 위한 객체 스토리지용)

**예제 2-10** 의 docker compose 구성은 로컬 테스트를 위해 간단한 Trino 컨테이너를 구성하는 방법을 보여줍니다.

#### 예제 2-10. 기본 Trino Docker Compose

```
서비스: trinodb:
  이미지: trinodb/
    trino:426-arm64 플랫폼: linux/arm64 호스트 이름:
    trinodb 컨테이너 이름: trinodb 볼륨: -
      $PWD/etc/catalog/
    delta.properties:/etc/trino/catalog/
    delta.properties - $ PWD/conf/etc/hadoop/conf/ 포트: - 대상: 8080 게시: 9090 프로토콜: tcp
```

모드: 호스트  
 환경:  
 - AWS\_ACCESS\_KEY\_ID=\$AWS\_ACCESS\_KEY\_ID  
 - AWS\_SECRET\_ACCESS\_KEY=\$AWS\_SECRET\_ACCESS\_KEY  
 - AWS\_DEFAULT\_REGION=\${AWS\_DEFAULT\_REGION:-us-west-1}  
 네트워크:  
 - DLDG

다음 예에서는 다음과 같은 리소스를 사용할 수 있다고 가정합니다.

- Amazon S3 또는 MinIO: (읽기, 쓰기 및 삭제 액세스를 허용하도록 사용자 및 역할이 설정된 버킷 프로비저닝). 로컬 MinIO를 사용하여 S3를 모의하는 것은 초기 비용 없이 모든 것을 시험해 볼 수 있는 간단한 방법입니다. 5장 아래의 github 책에서 docker-compose를 참조하세요.

- MySQL 또는 PostgreSQL: 로컬에서 실행하거나 선호하는 클라우드 공급자에 설정할 수 있습니다. 예를 들어 AWS RDS는 시작하는 간단한 방법입니다.
- Hive Metastore(HMS) 또는 Amazon Glue 데이터 카탈로그

다음으로 Trino에서 Delta 카탈로그를 만들 수 있도록 Delta Lake 커넥터를 구성하는 방법을 알아봅니다. hive-site.xml 구성 방법, s3에 필요한 jar 포함 방법, HMS 실행 방법 등 HMS(Hive Metastore) 사용에 대해 자세히 알아보려면 사이드바 텍스트를 읽어보세요. 그렇지 않으면 Trino 커넥터 구성 및 사용으로 건너뛰세요.

### Hive 메타스토어 실행 이미 신

회할 수 있는 메타스토어 인스턴스 설정이 있는 경우 연결 속성을 수정하여 대신 사용할 수 있습니다. 로컬 설정을 찾고 있다면 hive-site.xml 생성부터 시작할 수 있습니다. 다음은 MySQL과 Amazon S3에 연결하는 데 필요한 모든 것입니다.

#### 예제 2-11. HMS용 hive-site.xml

```
<configuration>
  <property>
    <name>hive.metastore.version</name> <value>3.1.0</
    value></property> <property>

    <name>javax.jdo.option.ConnectionURL</name> <값>jdbc:mysql://
    RDBMS_REMOTE_HOSTNAME:3306/metastore</value> </property> <property>

    <name>javax.jdo.option.ConnectionDriverName</name>
      <value>com.mysql.cj.jdbc.Driver </value> </property>
    <property>

    <name>javax.jdo.option.ConnectionUserName</name>
      <value>RDBMS_USERNAME</value> </
    property>
    <property>
      <name>javax.jdo.option.ConnectionPassword </name>
      <value>RDBMS_PASSWORD</value> </
    property>
    <property>
      <name>hive.metastore.warehouse.dir</name> <value>s3a://
      dlbgv2/delta/</value> </name> 속성 <속성> <이름
    >fs.s3a.access.key</
    name> <값
    >S3_ACCESS_KEY</값> </property> <속성>
```

```

<name>fs.s3a.secret.key</name>
<value>S3_SECRET_KEY</value></property>
<property>
<name>fs.s3.path-
style-access</name> <value>true</value></property>
<property> <name>fs.s3a.impl</
name>

<value>org.apache.hadoop.fs.s3a.S3AFileSystem</
value></property></configuration>

```

구성은 JDBC 연결 URL, 사용자 이름 및 비밀번호 속성을 사용하여 메타데이터 데이터베이스에 액세스하는 데 필요한 기본 사항과 `hive.metastore.warehouse.dir` 및 `fs.s3a`를 사용하여 데이터 웨어하우스를 제공합니다. \* 속성.

다음으로, 예제 2-12에서 수행한 대로 메타스토어를 실행하기 위해 docker compose 파일을 생성해야 합니다.

#### 예제 2-12. Hive Metastore용 Docker Compose

버전: "3.7"

서비스:

메타스토어:

이미지: `apache/hive:3.1.3` 플랫폼: linux/

amd64

호스트 이름: 메타스토어

컨테이너 이름: 메타스토어 볼륨: - \${PWD}/jars/

`hadoop-`

`aws-3.2.0.jar:/opt/hive/lib/ - ${PWD}/jars/mysql-connector-java-8.0.23.jar:/opt/hive/`

`lib/ - ${PWD}/jars/aws-java-sdk-bundle-1.11.375.jar:/opt/hive/lib/ - ${PWD}/conf:/opt/hive/conf 환경:`

- SERVICE\_NAME=메타스토어

- DB\_DRIVER=mysql

- IS\_RESUME="참"

노출: - 9083

포트: - 대

상: 9083 게시됨: 9083 프로

ток콜: tcp 모드: 호스트 네트워

크: - dldg

Metastore가 실행되면서 이제 Delta Lake용 Trino 커넥터를 활용하는 방법을 이해하기 위해 운전석에 섰습니다.

Trino 커넥터 구성 및 사용 Trino는 카탈로그라는 구성 파일을 사용

합니다. 카탈로그 유형(delta\_lake, hive 등)을 설명하고 특정 카탈로그를 조정하여 읽기 및 쓰기를 최적화하고 추가 커넥터 구성을 관리하는 데 사용됩니다.

Delta 커넥터의 최소 구성에는 주소 지정이 가능한 Hive 메타스토어 위치 thrift:hostname:port (Hive 메타스토어를 사용하는 경우)가 필요합니다. 지원되는 다른 카탈로그는 [Amazon Glue](#)입니다.

**예제 2-13**의 다음 구성은 하이브 메타스토어를 가리키는 커넥터를 구성합니다.

### 예제 2-13. Delta Lake 커넥터 속성

```
Connector.name=delta_lake
hive.metastore=thrift
hive.metastore.uri=thrift://metastore:9083 delta.hive-catalog-
name=metastore delta.compression-codec=SNAPPY
delta.enable-non-concurrent-writes=true
delta.target-max-file-size=512MB delta.unique-table-
location=true delta.vacuum.min-retention=7d
```



여러 작성자가 테이블을 비원자적으로 변경할 가능성이 있는 경우 delta.enable-non-concurrent-writes 속성을 true로 설정해야 합니다. 이는 Amazon s3의 경우에 가장 자주 발생하며 테이블의 일관성을 유지합니다.

위의 속성 파일은 delta.properties로 저장할 수 있습니다. 파일이 Trino 카탈로그 디렉토리 (/etc/trino/catalog/)에 복사되는 한 기본 hive.metastore.warehouse.dir에서 읽고, 쓰고, 삭제할 수 있으며 많은 작업을 수행할 수 있습니다. 더.

,

무엇이 가능한지 살펴보겠습니다.

### 카탈로그 표시 사용 카탈로그 표

시를 사용하는 것은 델타 커넥터가 올바르게 구성되었는지 확인하고 리소스로 표시되는지 확인하는 간단한 첫 번째 단계입니다.

```
trino> 카탈로그 표시;
목록
```

-----  
 델타  
 ...  
 (6열)

목록에 델타가 표시되는 한 스키마 생성으로 넘어갈 수 있습니다. 이는 카탈로그가 올바르게 구성되었음을 확인합니다.

### 스키마 생성 스키마라는 개념

은 약간 오버로드되어 있습니다. 테이블의 열을 설명하는 구조화된 데이터를 나타내는 스키마가 있지만 기존 데 이터베이스를 나타내는 스키마도 있습니다. 스키마 생성을 사용하면 데이터 웨어하우스 내에 액세스 및 거버넌스의 경계 역할을 할 수 있는 관리 위치를 생성할 수 있을 뿐만 아니라 브론즈, 실버, 골든 테이블 간에 물리적 테이블 데이터를 분리할 수 있습니다. 메달리온 아키텍처에 대한 자세한 내용은 11장에서 알아보겠습니다. 지금은 일부 원시 테이블을 저장하기 위해 bronze\_schema를 생성하겠습니다.

```
trino> 스키마 delta.bronze_schema 생성; 스키마 생성
```



CREATE SCHEMA가 반환되지 않고 예외가 발생했다면 물리적 웨어하우스에 쓰는 권한 문제 때문일 가능성이 높습니다. 다음은 예입니다.

```
쿼리 20231001_182856_00004_zjwqg 실패: 예외 발생:  

java.nio.file.AccessDeniedException s3a://com.new-front.dldgv2/delta/  

bronze_schema.db: s3a://com.newfront.dldgv2/delta/bronze_schema  

의 getFileStatus. db: com.amazonaws.services.s3.model.AmazonS3Exception:  

For-bidden(서비스: Amazon S3; 상태 코드: 403);
```

IAM 권한을 수정하거나 올바른 IAM 역할이나 액세스 키, 보안 액세스 키 쌍을 사용하고 있는지 확인하여 문제를 해결할 수 있습니다.

### 스키마 표시

사용 가능한 스키마를 보기 위해 카탈로그를 쿼리할 수 있습니다.

```
trino> 델타의 스키마를 표시합니다.  

개요  

-----  

기본 정보_  

키마 청동_스키마(3행)
```

우리가 찾고 있는 스키마가 존재한다면, 이제 일부 테이블을 생성할 준비가 된 것입니다.

## 테이블 작업

Trino와 Delta 생태계 간의 테이블 호환성을 위해서는 다음 사항을 준수해야 합니다.  
 몇 가지 지침. 데이터 유형 상호 운용성을 살펴본 다음 테이블을 만들고 추가합니다.  
 일부 행을 확인하고 거래 내역을 포함한 델타 메타데이터도 볼 수 있습니다.  
 변경 데이터 피드(CDF)에 대한 변경 내용을 추적합니다. 표를 보면서 마무리하겠습니다  
 최적화 및 진공 청소.

### 데이터 유형

특히 **입력** 과 관련하여 Trino를 사용하여 테이블을 생성할 때 몇 가지 주의 사항이 있습니다.  
**매핑** Trino와 Delta Lake의 차이점 다음 표를 사용할 수 있습니다.  
 적절한 유형이 사용되는지 확인하고, 다음과 같은 경우 비호환성을 방지하기 위해  
 목표는 상호 운용성입니다.

표 2-3. Delta-Trino 유형 매핑

델타 데이터 유형	트리노 데이터 유형
부울	부울
정수	정수
바이트	TINYINT
짧은	스몰린트
긴	빅트
뜨다	진짜
더블	더블
십진수(p,s)	십진수(p,s)
끈	VARCHAR
바이너리	바른바이너리
날짜	날짜
TIMESTAMPNTZ(TIMESTAMP_NTZ) TIMESTAMP(6)	
타임스탬프	시간대가 있는 타임스탬프(3)
정렬	정렬
지도	지도
구조체(...)	열(...)

### 테이블 옵션 생성

지원되는 테이블 옵션은 WITH 절을 사용하여 테이블에 적용할 수 있습니다.  
 CREATE TABLE 구문.

속성 이름	설명	기본
위치	의 파일 시스템 위치 URI입니다. 이 옵션 더 이상 사용되지 않습니다. 경고 또는 방법을 참조하세요. 아래에서 활성화	매핑된 관리되는 테이블을 사용합니다. hive.metastore.warehouse.dir의 위치 또는 접착제 카탈로그와 동등한 것.

속성 이름	설명	기본
partitioned_by	테이블을 분할할 열	OSS의 경우
checkpoint_interval	10번마다, Databricks의 경우 100번마다 Delta Lake에 변경 사항을 커밋하는 빈도는 파티션이 없습니다. (DBR)	
change_data_feed_enabled	다음에서 사용하기 위해 테이블에 대한 변경 사항을 추적합니다.	거짓
	CDC/CDF 애플리케	
컬럼_매핑_모드	이전에서 기본 쪽모이 세공 열을 매핑 하는 방법: 옵션(id, 이름, 없음)	없음

## 테이블 생성 긴

형식 <catalog>.<schema>.<table> 구문을 사용하거나 use delta.<schema>를 호출한 후 짧은 형식 구문 <table>을 사용하여 테이블을 생성할 수 있습니다. 다음 예제 2-14에서는 짧은 형식의 create를 사용하는 예제를 제공합니다.

### 예제 2-14. Trino를 사용하여 델타 테이블 만들기

```
trino> delta.bronze_schema를 사용합니다.
CREATE TABLE ecomm_v1_clickstream
  (event_date DATE,
   event_time VARCHAR(255),
   event_type VARCHAR(255),
   product_id INTEGER,
   Category_id BIGINT,
   Category_code VARCHAR(255), 브랜
   드 VARCHAR(255), 가격
   DECIMAL(5,2), user_id
   INTEGER, user_session
   VARCHAR (255)
  )
WITH
  (partitioned_by = ARRAY['event_date'],
   checkpoint_interval =
   30,change_data_feed_enabled = false,
   column_mapping_mode = 'name'
  );
```

예제 2-14 의 DDL 문을 사용하여 생성된 테이블은 데이터 웨어하우스에 매일 분할되는 관리되는 테이블을 생성합니다. 테이블 구조는 이 장 앞부분의 Flink 섹션에 있는 전자상거래 데이터를 나타냅니다.



기존 테이블과 함께 CREATE TABLE을 사용하는 것은 더 이상 사용되지 않습니다.  
 대신 system.register\_table 프로시저를 사용하세요. 그만큼  
 delta.legacy-create- ... WITH (location=...) 구문은 다음과 같습니다.  
 table-with-existing-location.enabled 카탈로그 구성 속성을 사용하여 CREATE TABLE을 일시적으로 다시 활성화했  
 습니다.  
 또는 Legacy\_create\_table\_with\_existing\_location\_enabled 고양이 -  
 alog 세션 속성.

## 테이블 나열

show tables를 사용하면 주어진 스키마 내의 테이블 컬렉션을 볼 수 있습니다.  
 델타 카탈로그에 있습니다.

```
trino:bronze_schema> 테이블 표시;
테이블
-----
      ecomm_v1_clickstream
(1줄)
```

## 설명을 사용하여 테이블 검사

특정 테이블의 소유자가 아닌 경우 설명을 사용하여 테이블에 대해 알아볼 수 있습니다.  
 메타데이터를 통해.

```
trino> delta.bronze_schema."ecomm_v1_clickstream"을 설명합니다.
```

열		유형	추가	논평
간   바르샤르    바		날짜   event_date 이벤트_시		
르샤르   event_type 제품_ID   정수   카테고리_ID   빙인트				
랜드   바르샤르   가		카테고리_코드   바르샤르   브		
격   삼진수(5,2)   사용자 ID   정수   사용자 세션   바르샤르				
(10행)				

## 삽입 사용

명령줄을 사용하거나 trino를 사용하여 행을 직접 삽입할 수 있습니다.  
 고객.

```
trino> delta.bronze_schema."ecomm_v1_clickstream"에 삽입
가치
(DATE '2023-10-01', '2023-10-01T19:10:05.704396Z', 'view', 44600062,
2103807459595387724, 'health.beauty', 'nars', 35.79, 541312140, '72d76fde-8bb3-4e00-8c23-
a032dfed738c'),
(DATE('2023-10-01'), '2023-10-01T19:20:05.704396Z', '보기', 54600062,
```

2103807459595387724, 'health.beauty', 'lancome', 122.79, 541312140, '72d76fde-8bb3-4e00-8c23-a032dfed738c';  
삽입: 2행

## 델타 테이블 쿼리 선

택 연산자를 사용하면 델타 테이블을 쿼리할 수 있습니다.

```
trino> delta.bronze_schema."ecomm_v1_clickstream"에서 event_date, product_id,  
브랜드, 가격을 선택합니다.
```

이벤트_날짜	제품_ID	브랜드	가격
2023-10-01	44600062	나스	35.79
			2023-10-01   54600062   랑콤
			122.79(2열)

## 행 업데이트 표

준 업데이트 연산자를 사용할 수 있습니다.

```
trino> delta.bronze_schema."ecomm_v1_clickstream" 업데이트  
-> SET 카테고리 코드 = 'health.beauty.products' -> 여기서 카테고리 ID =  
2103807459595387724;
```

## 선택 항목을 사용하여 테이블

만들기 다른 테이블을 사용하여 테이블을 만들 수 있습니다. 이를 CREATE TABLE AS라고 하며, 다른 테이블을 참조하여 새로운 물리적 델타 테이블을 생성할 수 있습니다.

```
trino> CREATE TABLE delta.bronze_schema."ecomm_lite"  
AS 선택 event_date, product_id, 브랜드, 가격 FROM  
delta.bronze_schema."ecomm_v1_clickstream";
```

## 테이블 작업 최적의 성능을 위

해 그리고 델타 테이블이 있는 실제 파일 시스템을 정리하기 위해 고려해야 할 테이블 작업이 많이 있습니다. 6장은 일반적인 유지 관리 및 테이블 유ти리티 기능을 다루고, 다음 섹션에서는 Trino 커넥터 내에서 사용할 수 있는 기능을 설명합니다.

## 진공

진공 작업은 지정된 델타 테이블의 현재 버전에 더 이상 필요하지 않은 파일을 정리합니다. 6장에서 진공 청소가 필요한 이유와 시간 여행을 통해 테이블 복구 및 이전 버전으로 롤백을 지원하기 위해 염두에 두어야 할 주의 사항에 대해 자세히 설명합니다.

Trino와 관련하여 델타 카탈로그 속성 delta.vacuum.min-retention pro -  
임의의 진공 호출이 발생할 경우 테이블을 보호하기 위한 게이팅 메커니즘 제공  
일수나 시간이 적습니다.

```
trino> CALL delta.system.vacuum('bronze_schema', 'ecommerce_v1_clickstream', '1d');
```

지정된 보존 기간(1.00d)이 시스템에 구성된 최소 보존 기간(7.00d)보다 짧습니다. delta.vacuum.min-retention 구성 속성 또는 delta.vacuum\_min\_retention 세션 속성을 사용하여 최소 보존 기간을 변경할 수 있습니다.

그렇지 않으면 진공 작업으로 인해 더 이상 존재하지 않는 물리적 파일이 삭제됩니다.  
테이블에 필요합니다.

#### 테이블 최적화

수정하면서 생성되는 테이블 부분의 크기에 따라  
Trino를 사용하면 테이블을 나타내는 작은 파일이 너무 많이 생성될 위험이 있습니다.  
테이블. 작은 파일을 더 큰 파일로 결합하는 간단한 기술은 bin-packing입니다.  
최적화(이 내용은 6장과 성능 튜닝 심층 분석에서 다룹니다.  
11장). 압축을 트리거하려면 EXECUTE를 사용하여 ALTER TABLE을 호출하면 됩니다.

```
trino> ALTER TABLE delta.bronze_schema."ecommerce_v1_clickstream" EXECUTE 최적화;
```

또한 최적화 작업의 동작을 변경하기 위한 추가 힌트를 제공할 수도 있습니다.  
다음은 10MB보다 큰 파일을 무시합니다.

```
trino> ALTER TABLE delta.bronze_schema."ecommerce_v1_clickstream" EXECUTE optimize(file_size_threshold => '10MB')
```

다음은 파티션 내의 테이블 파일 압축만 시도합니다.  
(event\_date="2023-10-01")

```
trino> ALTER TABLE delta.bronze_schema."ecommerce_v1_clickstream" 실행 최적화  
WHERE event_date = "2023-10-01"
```

#### 메타데이터 테이블

커넥터는 다음을 포함하는 각 Delta Lake 테이블에 대해 여러 메타데이터 테이블을 노출합니다.  
내부 구조에 대한 정보. 이 테이블을 쿼리하여 자세히 알아볼 수 있습니다.  
테이블에 대해 알아보고 변경 사항과 최근 기록을 검사합니다.

#### 테이블 기록

각 거래는 <table>\$history 메타데이터 테이블에 기록됩니다.

```
trino> delta.bronze_schema."ecommerce_v1_clickstream$history"를 설명합니다.
```

열	유형	추가	논평
버전	빅인트    시간대가 있는 타임스탬프(3)    바르샤		
타임스탬프	르		
user_id			

user_name	작업	varchar				
varchar	작업_메개변수   맵(varchar, varchar)	클러스터_id				
varchar	읽기_버전   bigint	격리_수준   varchar	is_bind_append   부울			

메타데이터 테이블을 쿼리할 수 있습니다. 우리의 마지막 세 가지 거래를 살펴보겠습니다.  
ecomm\_v1\_clickstream 테이블.

```
trino> delta.bronze_schema."ecomm_v1_clickstream$history"에서 버전, 타
임스탬프, 작업을 선택합니다.
버전 | 타임스탬프 | 작업
-----+-----+-----+-----+
0 | 2023-10-01 19:47:35.618 UTC | 테이블 만들기
1 | 2023-10-01 19:48:41.212 UTC | 쓰다
2 | 2023-10-01 23:01:13.141 UTC | 최적화
(3열)
```

## 데이터 피드 변경

Trino 커넥터는 CDF(변경 데이터 피드)를 읽는 기능을 제공합니다.

두 버전의 Delta Lake 테이블 간의 행 수준 변경 사항을 노출하는 항목입니다. 언제

특정 Delta Lake에서 change\_data\_feed\_enabled 테이블 속성이 true로 설정되어 있습니다. 테이블에서 커넥터는 테이블의 모든 데이터 변경에 대한 변경 이벤트를 기록합니다.

```
trino> delta.bronze_schema를 사용합니다.
테이블 생성 emmm_v1_clickstream(
    ...
)
와 함께 (
    Change_data_feed_enabled = true
);
```

이제 각 거래의 각 행이 (작업 유형과 함께) 기록되어 우리를 가능하게 합니다.

테이블 상태를 다시 작성하거나 특정 지점 이후의 변경 사항을 살펴보려면 테이블로 가는 시간.

예를 들어, 테이블 버전 0 이후의 모든 변경 사항을 보려면 다음을 수행하면 됩니다. 다음을 실행합니다.

```
trino> TABLE(에서 event_date, _change_type, _commit_version, _commit_timestamp를 선택합니다.

delta.system.table_changes(
    Schema_name => 'bronze_schema',
    table_name => 'ecomm_v1_clickstream',
    이후_버전 => 0
)
);
```

그리고 변경된 내용을 확인하세요. 예제 사용 사례에서는 단순히 두 개의 행을 삽입했습니다.

이벤트_날짜   _변경_유형   _커밋_버전	_commit_timestamp
2023-10-01   2023-10-01 삽입	1   2023-10-01 19:48:41.212 UTC
삽입(2행)	1   2023-10-01 19:48:41.212 UTC

## 테이블 속성 보기

테이블과 관련된 테이블 속성을 볼 수 있으면 유용합니다. 우리는 할 수 있다  
관련 델타 tblproperties를 보려면 메타데이터 테이블 <table>\$properties를 사용하십시오.

```
trino> delta.bronze_schema."ecomm_v1_clickstream$properties"에서 *를 선택합니다.
```

열쇠	값
delta.enableChangeDataFeed   진실	
delta.columnMapping.maxColumnId   10	
delta.columnMapping.mode   이름	
delta.checkpointInterval   30	
delta.minReaderVersion   2	
delta.minWriterVersion   5	

## 테이블 속성 수정

Delta 테이블의 기본 테이블 속성을 수정하려면 다음이 필요합니다.

지원되는 테이블 속성에 대해 Delta 커넥터 별칭을 사용합니다. 예를 들어,  
change\_data\_feed\_enabled는 delta.enableChangeDataFeed 속성에 매핑됩니다 .

```
트리노> ALTER TABLE delta.bronze_schema."ecomm_v1_clickstream"  
속성 설정 "change_data_feed_enabled" = false;
```

## 테이블 삭제

DROP TABLE 작업을 사용하면 존재하지 않는 테이블을 영구적으로 제거할 수 있습니다.

더 이상 필요합니다.

```
트리노> DROP TABLE delta.bronze_schema."ecomm_lite";
```

범위를 벗어나는 Trino 커넥터를 사용하여 수행할 수 있는 작업이 훨씬 더 많습니다.

이제 우리는 Trino에게 작별 인사를 하고 이 장을 마무리하겠습니다.

## 요약

이 장에서 함께 보낸 시간 동안 우리는 그것이 얼마나 간단한지 배웠습니다.

Flink 애플리케이션의 소스 또는 싱크로 Delta 테이블을 연결하는 것입니다 .

그런 다음 Rust 기반 kafka-delta-ingest 라이브러리를 사용하는 방법을 배웠습니다 .

대부분의 데이터에 대한 기본 요소인 데이터 수집 프로세스를 단순화합니다.

높은 처리량의 스트리밍 데이터를 다루는 엔지니어. 레벨을 낮추면서

단순히 데이터 스트림을 읽고 이를 델타 테이블에 쓰는 데 필요한 노력,

우리는 인지적 부담 측면에서 훨씬 더 나은 위치에 있게 됩니다. 우리가 시작할 때

경계가 있든 없든 테이블의 관점에서 모든 데이터를 생각해 보세요. 정신 모델을 적용하면 가장 데이터 집약적인 문제도 해결할 수 있습니다. 이에 대해 우리는 Delta용 기본 Trino 커넥터를 탐색하면서 이 장을 마무리했습니다. 우리는 델타 테이블에 데이터 진실의 단일 소스를 계속 유지하면서 간단한 구성을 통해 분석과 통찰력을 얻을 수 있는 방법을 발견했습니다.

3 장

# Delta Lake 유지 관리

초기 출시 독자를 위한 참고 사항 초기 출시 eBook을 사용

하면 가장 초기 형태의 책, 즉 저자가 집필하는 그대로의 편집되지 않은 원본 콘텐츠를 얻을 수 있으므로 이러한 타이틀이 공식 출시되기 훨씬 전에 이러한 기술을 활용할 수 있습니다.

이것이 마지막 책의 여섯 번째 장이 될 것이다. GitHub 저장소는 나중에 활성화될 예정입니다.

이 책의 내용 및/또는 예제를 개선할 수 있는 방법에 대한 의견이 있거나 이 장에 누락된 자료가 있는 경우 [gobrien@oreilly.com](mailto:gobrien@oreilly.com)으로 편집자에게 문의하십시오.

Delta Lake 테이블을 시간이 지나도 효율적으로 유지하는 과정은 차량이나 기타 대체 운송 수단(자전거, 스쿠터, 롤러블레이드)에 대한 예방적 유지 관리와 유사합니다. 인생에서와 마찬가지로 우리는 상황을 평가하고 해결책을 찾기 전에 타이어가 평크날 때까지 기다리지 않습니다. 우리는 조치를 취할 것입니다. 타이어 사용 사례에서는 간단한 관찰부터 시작하여 누출을 찾아 "타이어를 패치 해야 합니까?"라고 자문합니다. 문제는 "공기를 추가하는 것"만큼 간단할 수 있습니까, 아니면 이러한 상황입니까? 타이어 전체를 교체해야 하는 경우는 더욱 심각합니다. 상황을 평가하고 해결책을 찾고 솔루션을 적용하는 프로세스는 Delta Lake 테이블에도 적용될 수 있으며 모두 Delta Lake 테이블을 유지 관리하는 일반적인 프로세스의 일부입니다. 본질적으로 청소, 튜닝, 수리, 교체라는 측면에서 생각하면 됩니다.

다음 섹션에서는 Delta Lake 유ти리티 메서드를 활용하고 관련 구성(테이블 속성이라고도 함)에 대해 알아봅니다. 청소, 조정, 수리 및 교체를 위한 몇 가지 일반적인 방법을 살펴보겠습니다.

성능을 최적화하면서 도움의 손길을 빌려주기 위해 우리 테이블  
우리 테이블의 건강을 도모하고 궁극적으로 원인에 대한 확고한 이해를 구축하고  
우리가 취하는 행동의 효과 관계.

## Delta Lake 테이블 속성 사용

Delta Lake는 일반적인 유지 관리를 지원하는 다양한 유ти리티 기능을 제공합니다.

(청소 및 튜닝), 수리, 복원, 심지어 우리의 중요한 교체까지

테이블; 이 모든 것은 모든 데이터 엔지니어에게 귀중한 기술입니다. 이번 장을 시작하겠습니다.  
일반적인 유지 관리 관련 Delta Lake 테이블 중 일부를 소개합니다.

속성과 테이블을 적용, 수정, 제거하는 방법을 보여주는 간단한 연습  
속성.

**표 3-1은** 이 장의 나머지 부분 전체에서 참조되며,  
편리한 참조가 필요합니다. 각 행에는 속성 이름, 내부 데이터 유형 및  
청소, 조정, 수리 또는 교체와 관련된 사용 사례  
델타 레이크 테이블.

테이블 정의와 함께 저장된 메타데이터에는 TBLPROPERTIES가 포함됩니다. 와 함께  
Delta Lake 이러한 속성은 유ти리티 메서드의 동작을 변경하는 데 사용됩니다.  
이를 통해 속성을 추가하거나 제거하고 동작을 제어하는 것이 매우 간단해졌습니다.  
Delta Lake 테이블의

표 3-1. Delta Lake 테이블 속성 참조

	데이터 형식	함께 사용	기본
delta.logRetentionDuration	달력간격 청소		간격 30일
delta.deletedFileRetentionDuration	달력간격 청소		간격 1주일
delta.setTransactionRetentionDuration	Calendar	간격 정리, 복구(없음)	
delta.targetFileSizea	스트링 튜닝(없음)		
delta.tuneFileSizesForRewritesa	부울	튜닝 튜	(없음)
delta.autoOptimize.optimizeWritea	부울	닝 튜	(없음)
delta.autoOptimize.autoCompacta	부울	닝 튜	(없음)
delta.dataSkippingNumIndexedCols	정수	닝 튜	32
delta.checkpoint.writeStatsAsStruct	부울	닝 튜닝	(없음)
delta.checkpoint.writeStatsAsJson	부울		진실

속성 Databricks 독점.

tblproperties 사용의 장점은 tblproperties의 메타데이터에만 영향을 미친다는 것입니다.  
테이블이며 대부분의 경우 물리적 테이블 구조를 변경할 필요가 없습니다.  
또한 옵트인 또는 옵트아웃 기능을 통해 Delta Lake의  
돌아가서 기존 파이프라인 코드를 변경할 필요 없이 동작을 수행할 수 있습니다.  
대부분의 경우 스트리밍 애플리케이션을 다시 시작하거나 재배포할 필요가 없습니다.



테이블 속성을 추가하거나 제거할 때의 일반적인 동작은 삽입, 삭제, 업데이트로 구성된 일반적인 데이터 조작 언어 연산자(DML)를 사용하는 것과 다르지 않으며, 고급 경우에는 upsert를 기반으로 행을 삽입하거나 업데이트합니다. 경기. 12장에서는 Delta를 사용한 고급 DML 패턴을 다룹니다.

모든 테이블 변경 사항은 일괄 처리의 경우 다음 트랜잭션(자동) 중에 적용되거나 스트리밍 애플리케이션에서 즉시 표시됩니다.

스트리밍 Delta Lake 애플리케이션을 사용하면 테이블 메타데이터 변경 사항을 포함한 테이블 변경 사항이 ALTER TABLE 명령처럼 처리됩니다. 유ти리티 기능 vacuum 및 최적화 와 같이 물리적 테이블 데이터를 수정하지 않는 테이블에 대한 다른 변경 사항은 지정된 스트리밍 애플리케이션의 흐름을 중단하지 않고 외부에서 업데이트할 수 있습니다.

물리적 테이블 또는 테이블 메타데이터에 대한 변경 사항은 동일하게 처리되며 델타 로그에 버전이 지정된 레코드를 생성합니다. 새 트랜잭션을 추가하면 동기화되지 않은(부실) 프로세스에 대해 DeltaSnapshot의 로컬 동기화가 발생합니다. 이는 모두 Delta Lake가 여러 동시 작성자를 지원하여 Delta Log 테이블의 중앙 동기화를 통해 분산형(분산형) 방식으로 변경이 발생할 수 있기 때문입니다.

사람의 의도적인 조치와 다운스트림 소비자에 대한 사전 안내가 필요한 유지 관리 범위에 속하는 다른 사용 사례도 있습니다.

이 장을 마무리하면서 REPLACE TABLE을 사용하여 파티션을 추가하는 방법을 살펴보겠습니다. 이 프로세스는 작업이 델타 테이블의 물리적 레이아웃을 다시 작성하므로 테이블의 활성 판독기를 종단시킬 수 있습니다.

각 테이블 속성에 의해 제어되는 프로세스에 관계없이 CREATE TABLE을 사용하여 생성된 시점의 테이블 또는 주어진 테이블과 연결된 속성을 변경할 수 있는 ALTER TABLE을 통해 생성된 시점 이후의 테이블입니다.

이 장의 나머지 부분을 따라가려면 동반 docker 환경과 함께 covid\_nyt 데이터세트(책의 GitHub 저장소에 포함됨)를 사용하게 됩니다. 시작하려면 다음을 실행하세요.

```
$ 내보내기 DLDG_DATA_DIR=~/path/to/delta-lake-definitive-guide/datasets/ $ 내보내기
DLDG_CHAPTER_DIR=~/path/to/delta-lake-definitive-guide/ch6 $ docker run --rm -it \
```

```
--name delta_quickstart \ -v
$DLDG_DATA_DIR:/opt/spark/data/datasets \ -v
$DLDG_CHAPTER_DIR:/opt/spark/work-dir/ch6 \ -p
8888-8889:8888-8889 \
delta_quickstart
```

이 명령은 JupyterLab 환경을 로컬로 실행합니다. 출력에 제공된 URL을 사용하여 jupyterlab 환경을 열고 ch6/chp6\_notebook.ipynb 를 클릭하여 따라갑니다.

속성을 사용하여 빈 테이블 만들기 이 책 전체에서 다양

한 방법으로 테이블을 만들었으므로 간단히 SQL CREATE TABLE 구문을 사용하여 빈 테이블을 생성해 보겠습니다. 아래 예제 3-1 에서는 단일 날짜 열과 하나의 기본 테이블 속성 delta.logRetentionDuration을 사용하여 새 테이블을 만듭니다. 이 속성이 이 장의 뒷부분에서 어떻게 사용되는지 다루겠습니다.

### 실시예 3-1. 기본 테이블 속성을 사용하여 델타 테이블 만들기

```
$ 스파크.sql("""
    존재하지 않는 경우 테이블 생성 default.covid_nyt ('날짜 DATE') DELTA
    TBLPROPERTIES('delta.logRetentionDuration='interval 7 days')를 사용하여; """")
```



covid\_nyt 데이터 세트에는 6개의 열이 있다는 점을 지적할 가치가 있습니다. 앞의 예에서는 다음 단계에서 전체 covid\_nyt 테이블을 가져오는 동안 전체 covid\_nyt 테이블의 스키마를 훔칠 수 있으므로 의도적으로 게으릅니다. 이는 테이블 정의에서 누락된 열을 채워 현재 테이블의 스키마를 발전시키는 방법을 알려줍니다.

테이블 채우기 이제 비어 있는 Delta

Lake 테이블이 있습니다. 이는 본질적으로 테이블의 약속이지만 현재는 /{tablename}/\_delta\_log 디렉토리와 빈 테이블의 스키마 및 메타데이터가 포함된 초기 로그 항목만 포함합니다. 간단한 테스트를 실행하여 확인하려면 다음 명령을 실행하여 테이블의 백업 파일을 표시할 수 있습니다.

```
$ 스파크.테이블("default.covid_nyt").inputFiles()
```

inputFiles 명령은 빈 목록을 반환합니다. 그건 당연한 일이지만 조금 쓸쓸한 느낌도 든다. 계속해서 일부 데이터를 추가하여 이 테이블에 즐거움을 더해 보겠습니다. covid\_nyt Parquet 데이터의 간단한 읽기 작업을 관리형 Delta Lake 테이블(이전의 빈 테이블)에 직접 실행하겠습니다.

활성 세션에서 다음 코드 블록을 실행하여 covid\_nyt 데이터세트를 빈 default.covid\_nyt 테이블에 병합합니다.



코로나19 데이터세트에는 STRING 으로 표시되는 날짜 열이 있습니다 . 이 연습에서는 날짜 열을 DATE 유형으로 설정하고 테이블의 기존 데이터 유형을 존중하기 위해 withColumn("date", to\_date("date", "yyyy-MM-dd"))를 사용합니다. .

pyspark.sql.functions에서 \$ to\_date 가져오기

```
(spark.read .format("parquet") .load("/opt/spark/work-dir/rs/data/COVID-19_NYT/
    *.parquet") .withColumn("날짜", to_date("날짜", "yyyy-MM-dd"))
    .쓰다
    .format("델
    타") .saveAsTable("default.covid_nyt")
)
```

작업이 실행되지 않는 것을 확인할 수 있습니다.

\$ pyspark.sql.utils.AnalyticException: default.covid\_nyt 테이블이 이미 존재합니다.

방금 AnalysisException이 발생했습니다. 다행스럽게도 이 예외는 올바른 이유로 우리를 차단하고 있습니다. 이전 코드 블록에서 발생한 예외는 **DataFrameWriter** 의 기본 동작으로 인해 발생했습니다. Spark에서는 기본값이 **errorIfExists**입니다.

이는 테이블이 존재하는 경우 기존 테이블을 손상시킬 수 있는 작업을 수행하기보다는 예외를 발생시킨다는 의미입니다.

이러한 과속 방지턱을 극복하려면 작업의 쓰기 모드를 추가로 변경해야 합니다. 이는 의도적으로 기존 테이블에 레코드를 추가한다는 작업 동작을 변경합니다.

계속해서 쓰기 모드를 추가로 구성하겠습니다.

```
(스파크.읽기
...
    .write.format("델
    타") .mode("추가")
...
)
```

좋아요. 한 가지 장애물을 통과했고 더 이상 "테이블이 이미 존재합니다" 예외로 인해 차단되지 않지만 또 다른 AnalysisException이 발생했습니다.

\$ pyspark.sql.utils.AnalyticException: 델타 테이블(테이블 ID: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx)에 쓸 때 스키마 불일치가 감지되었습니다.

이번에는 스키마 불일치로 인해 AnalysisException이 발생합니다. 이는 예상(커밋된) 테이블 스키마 (현재 1개의 열이 있음)와 로컬 스키마(covid\_nyt 쪽모이 세공 마루를 읽음) 사이에 불일치가 있을 때 델타 프로토콜이 우리(운영자)가 맹목적으로 변경하지 않도록 보호하는 방법입니다. 현재 커밋되지 않았으며 6개의 열이 있습니다. 이 예외는 또 다른 가드레일입니다.

스키마 적용으로 알려진 프로세스인 테이블 스키마의 실수로 인한 오염을 차단하기 위해 마련되었습니다.

## 스키마 적용 및 진화

Delta Lake는 쓰기 스키마라는 기존 데이터 웨어하우스의 기술을 활용합니다. 이는 단순히 쓰기 작업이 실행되기 전에 기존 테이블과 비교하여 기록기의 스키마를 확인하는 프로세스가 있음을 의미합니다. 이는 이전 트랜잭션을 기반으로 테이블 스키마에 대한 단일 정보 소스를 제공합니다.

### 스키마 적용 쓰기 트랜잭션이 발

생하도록 허용하기 전에 기존 스키마를 확인하고 불일치가 발생할 경우 예외를 발생시키는 제어 프로세스입니다.

### 스키마 진화 이전 버전과의

호환성을 가능하게 하는 방식으로 기존 스키마를 의도적으로 수정하는 프로세스입니다. 이는 전통적으로 쓰기 시 mergeSchema 옵션을 활성화하는 기능과 함께 Delta Lake에서도 지원되는 ALTER TABLE {t} ADD COLUMN(S)을 사용하여 수행됩니다.

## 테이블 스키마 발전

기존 테이블에 covid\_nyt 데이터를 추가하는 데 필요한 마지막 단계는 예, 테이블에 가져오는 스키마 변경 사항을 승인하고 실제 테이블 데이터와 수정 사항을 모두 커밋하겠다고 명시적으로 밝히는 것입니다. 테이블 스키마.

```
$
(spark.read .format("parquet") .load("/")
opt/spark/work-dir/rs/data/COVID-19_NYT/*.parquet") .withColumn("날짜", to_date("날짜", "yyyy-MM-
dd")) .write .format("delta") .mode("append") .option("mergeSchema",
"true") .saveAsTable("default.covid_nyt")
)
```

성공. 이제 이전 코드를 실행한 결과로 작업할 테이블이 생겼습니다. 간단히 요약하면 다음과 같은 이유로 쓰기 작업에 두 개의 수정자를 추가해야 했습니다.

- 쓰기 모드를 추가 작업으로 업데이트했습니다. 이는 별도의 트랜잭션에서 테이블을 생성했고 Delta Lake 테이블이 이미 존재할 때 기본 쓰기 모드(errorIfExists)가 작업을 단락시키기 때문에 필요했습니다.

2. nyc\_taxi 데이터를 물리적으로 추가한 동일한 트랜잭션 내에서 데이터 세트에 필요한 5개의 추가 열을 추가하여 covid\_nyt 테이블 스키마를 수정할 수 있는 mergeSchema 옵션을 포함하도록 쓰기 작업을 업데이트했습니다.

말하고 수행한 모든 작업을 통해 이제 테이블에 실제 데이터가 있으며 그 과정에서 쪽모이 세공 기반 covid\_nyt 데이터 세트에서 스키마를 발전시켰습니다.

다음 DESCRIBE 명령을 실행하여 전체 테이블 메타데이터를 살펴볼 수 있습니다 .

```
$ spark.sql("확장된 default.covid_nyt 설명").show(truncate=False)
```

DESCRIBE를 실행한 후 열(및 설명), 파티셔닝(우리의 경우 없음) 및 사용 가능한 모든 tblproperties를 포함하는 전체 테이블 메타데이터를 볼 수 있습니다 . 설명을 사용하면 테이블이나 앞으로 작업해야 할 테이블에 대해 알아가는 간단한 방법입니다.

## 자동 스키마 진화의 대안

이전 사례에서는 .option("mergeSchema", "true")을 사용하여 Delta Lake 작성자의 동작을 수정했습니다. 이 옵션은 Delta Lake 테이블 스키마를 발전시키는 방법을 단순화하지만 테이블 스키마의 변경 사항을 완전히 인식하지 못하는 대가를 치르게 됩니다. 업스트림 소스에서 알 수 없는 열이 도입되는 경우 앞으로 가져올 열이 무엇인지, 무시해도 안전한 열이 무엇인지 알고 싶을 것입니다.

Alter Table을 사용하여 의도적으로 열 추가

`default.covid\_nyt` 테이블에 5개의 누락된 열이 있다는 것을 알고 있는 경우 ALTER TABLE을 실행하여 누락된 열을 추가할 수 있습니다.

```
$ 스파크.sql("""
    ALTER TABLE default.covid_nyt ADD COLUMNS
    ( 카운티 STRING, 주
        STRING, fips INT, 사례
        INT, 사망 INT

    );
""")
```

테이블 스키마에 대한 수정 사항을 자동으로 병합하는 방법을 배웠기 때문에 이 프로세스는 번거로워 보일 수 있지만 궁극적으로 예상치 못한 변경 사항을 되감고 실행 취소하는 데 더 많은 비용이 듭니다. 약간의 사전 작업을 수행하면 자동 스키마 변경을 명시적으로 거부하는 것이 어렵지 않습니다.

```
(spark.read.format("parquet"))
```

```

.load("/opt/spark/work-dir/rs/data/COVID-19_NYT/
*.parquet") .withColumn("date", to_date("date", "yyyy-MM-dd"))
쓰다

.format("delta") .option("mergeSchema",
"false") .mode("append") .saveAsTable("default.covid_nyt"))
)

```

그리고 짜잔. 우리는 예상치 못한 일 없이 의도적으로 테이블에 예상되는 모든 변경 사항을 적용하므로 테이블을 깨끗하고 깔끔하게 유지하는 데 도움이 됩니다.

## 테이블 속성 추가 또는 수정 기존 테이

틀 속성을 추가하거나 수정하는 프로세스는 간단합니다. 속성이 이미 존재하는 경우 모든 변경 사항은 기존 속성을 맹목적으로 덮어씁니다. 새로 추가된 속성은 테이블 속성 집합에 추가됩니다.

이 동작을 확인하려면 활성 세션에서 다음 ALTER TABLE 문을 실행하세요.

```

$ 스파크.sql("""
ALTER TABLE default.covid_nyc SET
TBLPROPERTIES(
'engineering.team_name'='dldg_authors',
'engineering.slack'='delta-users.slack.com'
) """
)
```

이 작업은 테이블 메타데이터에 두 가지 속성, 즉 이 책의 저자에 대한 팀 이름 (dldg\_authors)에 대한 포인터와 Slack 조직 (delta-users.slack.com)을 추가합니다. 테이블의 메타데이터를 수정할 때마다 변경 사항이 테이블 기록에 기록됩니다. 테이블 속성에 대한 변경 사항을 포함하여 테이블에 대한 변경 사항을 보려면 DeltaTable Python 인터페이스에서 기록 메서드를 호출하면 됩니다.

```

$ from delta.tables import DeltaTable dt =
DeltaTable.forName(spark, 'default.covid_nyt')
dt.history(10).select("version", "timestamp", "operation").show()
```

그러면 테이블에 대한 변경 사항이 출력됩니다.

버전	타임스탬프	운영
2	2023-06-07 04:38:...[TBL속성 설정]	1 2023-06-07 04:14:...
쓰기	0 2023-06-07 04:13:...	테이블 만들기

이전 트랜잭션의 변경 사항을 보거나 확인하려면 covid\_nyt 테이블에서 SHOW TBLPROPERTIES 를 호출하면 됩니다.

```
$ Spark.sql("show tblproperties default.covid_nyt").show(truncate=False)
```

또는 이전부터 DeltaTable 인스턴스에 대해 Detail() 함수를 실행할 수 있습니다 .

```
$ dt.detail().select("속성").show(truncate=False)
```

모든 것을 마무리하기 위해 이제 원치 않는 테이블 속성을 제거하는 방법을 배운 다음 Delta Lake 테이블을 정리하고 최적화하는 방법을 학습하여 여정을 계속할 수 있습니다.

## 테이블 속성 제거

테이블 속성만 추가하는 것만으로는 의미가 없으므로 이 장의 시작 부분을 마무리하고 ALTER TABLE table\_name UNSET을 사용하는 방법을 살펴보겠습니다.

TBL속성.

예를 들어 실제 속성 delta.logRetentionDuration 이 아니라 delta.loRgeten tionDuratio 와 같이 속성 이름의 철자를 실수로 잘못 입력했다고 가정해 보겠습니다. 이 실수가 세상의 종말은 아니므로 그대로 유지할 이유가 없습니다.

원치 않는(또는 철자가 틀린) 속성을 제거하려면 ALTER TABLE 명령에서 UNSET TBLPRO PERTIES를 실행할 수 있습니다 .

```
$ 스파크.sql("""
    ALTER TABLE default.covid_nyt UNSET
    TBLPROPERTIES('delta.loRgetentionDuratio') """)
```

마찬가지로 원치 않는 속성이 더 이상 테이블 속성에서 공간을 차지하지 않습니다.

방금 초기 생성 시점에 기본 테이블 속성을 사용하여 델타 레이크 테이블을 생성하는 방법을 배웠고, 스키마 적용 규칙과 테이블 스키마를 의도적으로 발전시키는 방법, 속성을 추가, 수정, 제거하는 방법을 다시 배웠습니다. 다음으로 Delta Lake 테이블을 깨끗하고 깔끔하게 유지하는 방법을 살펴보겠습니다.

### (Spark 전용) 기본 테이블 속성

다양한 Delta Lake 테이블 속성의 미묘한 차이에 익숙해지면 다음 Spark 구성 접두사를 사용하여 SparkSession에 고유한 기본 속성 집합을 제공할 수 있습니다.

```
Spark.databricks.delta.properties.defaults.<conf>
```

이는 Spark 워크로드에만 작동하지만 파이프라인에 속성을 자동으로 주입하는 기능이 유용할 수 있는 많은 시나리오를 상상할 수 있습니다.

```
Spark...delta.defaults.logRetentionDuration=간격 2주
Spark...delta.defaults.deletedFileRetentionDuration=간격 28일
```

유용하다고 말하면, 테이블 속성은 테이블 소유자, 엔지니어링 팀, 통신 채널(슬랙 및 이메일)에 대한 메타데이터를 저장하는 데 사용할 수 있으며 기본적으로 설명 테이블 메타데이터의 유용성을 확장하고 데이터 검색을 단순화하고 소유자를 캡처하는 데 도움이 되는 기타 모든 항목을 저장할 수 있습니다. 데이터 세트 소유권에 대한 책임은 인간에게 있습니다. 앞서 살펴본 것처럼 테이블 메타데이터는 단순한 구성을 뛰어넘는 풍부한 정보를 저장할 수 있습니다.

**표 3-20에는** 델타 테이블을 확장하는 데 사용할 수 있는 몇 가지 예제 테이블 속성이 나열되어 있습니다. 속성은 접두사로 분류되며 기존 테이블 속성과 함께 추가 데이터 카탈로그 스타일 정보를 제공합니다.

표 3-2. 데이터 카탈로그화를 위한 테이블 속성 사용

속성	설명
탈로그.팀_이름	을 입력하고 “테이블에 대한 책임은 누구에게 있습니까?”라는 질문에 답합니다.
Catalog.engineering.comms.slack	엔지니어링 팀을 위한 Slack 채널 제공: <a href="https://delta-users.slack.com/archives/CG9LR6LN4">https://delta-users.slack.com/archives/CG9LR6LN4</a> 와 같은 영구 링크 사용 채널 이름은 시간이 지남에 따라 변경될 수 있기 때문입니다.
Catalog.engineering.comms.email	dldg_authors@gmail.com : 이 이메일은 실제 이메일이 아니지만 요점을 이해하실 수 있습니다.
catalog.table.classification	테이블 유형을 선언하는 데 사용할 수 있습니다. 예: pii,sensitive-pii,general,all-액세스 등. 이러한 값은 역할 기반 액세스에도 사용할 수 있습니다. (통합은 이 책의 범위를 벗어납니다)

## 델타 테이블 최적화

“각 행동에는 동등하고 반대되는 반응이 있다”라는 인용문을 기억하십니까? 물리 법칙과 마찬가지로 Delta Lake 테이블에서 새 데이터가 삽입(추가), 수정(업데이트), 병합(업서트) 또는 제거(삭제)될 때(작업) 변경 사항이 느껴질 수 있습니다. 각 작업을 원자성 트랜잭션(버전, 타임스탬프, 작업 등)으로 기록하여 테이블이 현재 사용 사례를 계속해서 제공할 뿐만 아니라 되돌릴 수 있을 만큼 충분한 기록(시간 이동)을 유지하도록 보장하는 것입니다.) 이전 상태(테이블 시간의 시점)로 돌아가서 테이블에 더 큰 문제가 발생한 경우 테이블을 수정(덮어쓰기)하거나 복구(교체)할 수 있습니다.

그러나 더 복잡한 유지 관리 작업을 시작하기 전에 먼저 시간이 지남에 따라 테이블에 몰래 나타날 수 있는 일반적인 문제를 살펴보겠습니다.

---

1 뉴턴의 제3법칙

이러한 문제를 작은 파일 문제라고 합니다. 이제 문제와 해결책을 살펴보겠습니다.

#### 큰 테이블과 작은 파일의 문제 작은 파일 문제에 관해 이야기할 때 실제

로는 Delta Lake에만 국한된 문제가 아니라 네트워크 IO 문제와 높은(오픈 비용) 문제에 대해 이야기하고 있습니다. 너무 많은 작은 파일로 구성된 최적화되지 않은 테이블. 작은 파일은 64kb 이하의 모든 파일로 분류될 수 있습니다.

작은 파일이 너무 많으면 어떻게 해를 끼칠 수 있습니까? 대답은 여러 가지가 있지만 모든 문제의 공통점은 시간이 지남에 따라 몰래 나타나 테이블을 캡슐화하는 실제 파일의 레이아웃을 수정해야 한다는 것입니다. 테이블 속도가 느려지고 테이블 자체의 무게로 인해 문제가 발생하는 시점을 인식하지 못하면 쿼리를 효율적으로 열고 실행하기 위해 분산 컴퓨팅에 비용이 많이 증가할 수 있습니다.

테이블이 메모리에 물리적으로 로드되기 전에 필요한 작업 단계 수 측면에서 실제 비용이 발생하며, 이는 테이블을 더 이상 효율적으로 로드할 수 없는 지점까지 시간이 지남에 따라 증가하는 경향이 있습니다.



이는 배포 단위가 작업에 묶여 있고 파일이 "블록"으로 구성되고 각 블록이 1개의 작업을 수행하는 MapReduce 및 Spark와 같은 전통적인 Hadoop 스타일 생태계에서 훨씬 더 많이 느껴집니다.

테이블에 각각 1GB이고 블록 크기가 64MB인 1백만 개의 파일이 있는 경우 전체 테이블을 읽으려면 무려 1,565만 개의 작업을 배포해야 합니다. 파일 시스템 IO 및 네트워크 IO를 줄이기 위해 테이블에 있는 실제 파일의 대상 파일 크기를 최적화하는 것이 이상적입니다. 최적화되지 않은 파일(작은 파일 문제)이 발생하면 이로 인해 테이블 성능이 크게 저하됩니다. 확실한 예를 들어, 동일한 대형 테이블(~1TB)이 있지만 테이블을 구성하는 파일이 각각 약 5kb로 균등하게 분할되었다고 가정해 보겠습니다. 이는 1GB당 200,000개의 파일이 있고 테이블을 로드하기 전에 약 2억 개의 파일을 열어야 함을 의미합니다. 대부분의 경우 테이블이 열리지 않습니다.

재미삼아 실제 작은 파일 문제를 재현한 다음 테이블을 최적화하는 방법을 알아 보겠습니다. 계속하려면 다음 예에서 covid\_nyt 데이터셋을 계속 사용할 것이므로 이 장의 앞부분의 세션으로 돌아가세요.

#### 작은 파일 문제 생성 covid\_nyt 데이

터 세트에는 백만 개가 넘는 레코드가 있습니다. 테이블의 전체 크기는 작은 데이터 세트인 8개의 파티션에 걸쳐 분할된 7MB 미만입니다.

```
$ ls -lh /opt/spark/work-dir/ch6/spark-warehouse/covid_nyt/*.parquet | 화장실 -l 8
```

문제를 뒤집어서 covid\_nyt 데이터세트를 나타내는 파일이 9,000개 또는 심지어 1백만 개라면 어떻게 될까요? 이 사용 사례는 극단적이지만 책 뒷부분(9장)에서 스트리밍 애플리케이션이 수많은 작은 파일을 생성하는 일반적인 원인이라는 사실을 배우게 됩니다!

default.nonoptimal\_covid\_nyt라는 또 다른 빈 테이블을 만들고 몇 가지 간단한 명령을 실행하여 테이블 최적화를 취소해 보겠습니다. 우선, 다음 명령을 실행하십시오.

```
$ from delta.tables import
DeltaTable(DeltaTable.createIfNotExists(spark)
    .tableName("default.nonoptimal_covid_nyt") .property("description",
    "최적화 할 테이블") .property("catalog.team_name",
    "dldg_authors") .property("catalog.engineering.comms.slack",
    "https://delta-users.slack.com/archives/
CG9LR6LN4") .property("catalog.engineering.comms.email", "dldg_authors@gmail.com") .property("catalog.table.classification",
    "모든 액세스") .addColumn("날짜", "DATE") .addColumn("county", "STRING") .addColumn("state",
    "STRING") .addColumn("fips",
    "INT") .addColumn("사례", "INT") .addColumn("죽
을", "INT") .execute()
```

이제 테이블이 있으므로 일반 default.covid\_nyt 테이블을 소스로 사용하여 너무 많은 작은 파일을 쉽게 생성할 수 있습니다. 테이블의 총 행 수는 1,111,930개입니다. 테이블을 기준 8개에서 9000개의 파티션으로 다시 분할하면 테이블이 파일당 약 5kb의 9000개 파일로 분할됩니다.

\$



물리적 테이블 파일을 보려면 다음 명령을 실행하면 됩니다.

```
$ docker exec -it delta_quickstart bash \ -c "ls -l /opt/spark/work-dir/
ch6/spark-warehouse/nonoptimal_covid_nyt/*parquet | wc -l"
```

정확히 9000개의 파일이 있는 것을 볼 수 있습니다.

이제 최적화할 수 있는 테이블이 생겼습니다. 다음으로 최적화 도구를 소개하겠습니다. 유ти리티로서 친구라고 생각하십시오. 테이블을 나타내는 많은 작은 파일을 몇 개의 큰 파일로 쉽게 통합하는 데 도움이 됩니다. 모든 것이 눈 깜짝할 사이에 이루어집니다.

Optimize를 사용하여 작은 파일 문제 해결 Optimize는 z 순서와 bin-packing의 두 가지 변형으로 제공되는 Delta 유ти리티 기능입니다. 기본값은 빈 패킹입니다.

### 최적화 빈

패킹이란 정확히 무엇입니까? 상위 수준에서 이는 임의 개수의 저장소에 걸쳐 많은 작은 파일을 더 적은 수의 큰 파일로 통합하는 데 사용되는 기술입니다. bin은 최대 파일 크기의 파일로 정의됩니다(Spark Delta Lake의 기본값은 1GB, Delta Rust는 250mb).

OPTIMIZE 명령은 여러 구성들을 혼합하여 조정할 수 있습니다.

최적화 임계값을 조정할 때 염두에 두어야 할 몇 가지 고려 사항이 있습니다.

- (spark에만 해당) Spark.databricks.delta.optimize.minFileSize (긴)는 OPTIMIZE 명령으로 더 큰 파일에 다시 쓰기 전에 임계값(바이트)보다 작은 파일을 그룹화하는 데 사용됩니다 .
- (스파크 전용) Spark.databricks.delta.optimize.maxFileSize (긴)는 다음 용도로 사용됩니다. OPTIMIZE 명령으로 생성된 대상 파일 크기를 지정합니다.
- (spark 전용) Spark.databricks.delta.optimize.repartition.enabled (bool)는 OPTIMIZE의 동작을 변경하는 데 사용되며 축소 시 coalesce(1) 대신 repartition(1)을 사용합니다. • (delta-rs 및 비 -OSS delta) 테이블 속성

delta.targetFileSize (문자열)는 delta-rs 클라이언트 와 함께 사용할 수 있지만 현재 OSS 델타 릴리스에서는 지원되지 않습니다. 예를 들면 250MB입니다.

OPTIMIZE 명령은 결정적이며 고르게 분산된 Delta Lake 테이블(또는 지정된 테이블의 특정 하위 집합)을 달성을 것을 목표로 합니다.

최적화가 실행되는 모습을 보려면 nonoptimal\_covid\_nyt 테이블에서 최적화 기능을 실행하면 됩니다. 원하는 만큼 명령을 실행해 보세요. 최적화는 새 레코드가 테이블에 추가된 경우에만 두 번째로 적용됩니다.

```
$ results_df =
(DeltaTable.forName(spark,
"default.nonoptimal_covid_nyt").optimize().executeCompaction())
```

최적화 작업 실행 결과는 데이터 프레임 (results\_df) 에 로컬로 반환되며 테이블 기록을 통해서도 사용할 수 있습니다. OPTIMIZE 통계를 보려면 DeltaTable 인스턴스에서 기록 메서드를 사용할 수 있습니다.

pyspark.sql.functions에서 \$ 가져오기 열

```
(DeltaTable.forName(spark, "default.nonoptimal_covid_nyt") .history(10) .where(col("작업") == "OPTIMIZE") .select("version", "timestamp", "작업", "operationMetrics.numRemovedFiles", "operationMetrics.numAddedFiles")

.show(잘림=거짓))
```

결과 출력은 다음 표를 생성합니다.

버전	타임스탬프	작업	numRemovedFiles	numAddedFiles
2	2023-06-07 06:47:28.488	최적화	9000	1

작업에 대한 중요한 열은 9000개의 파일 (numRemovedFiles) 을 제거하고 압축된 파일 1개 (numAddedFiles)를 생성했음을 보여줍니다.



델타 스트리밍 및 스트리밍 최적화에 대해서는 9장으로 넘어가세요.

## Z-Order 최적화 Z-

순서는 **기술**입니다. 동일한 파일 세트에 관련 정보를 함께 배치합니다.

관련 정보는 테이블 열에 있는 데이터입니다. covid\_nyt 데이터 세트를 고려해보세요. 시간 경과에 따른 주별 사망률을 신속하게 계산하려는 경우 Z-ORDER BY를 활용하면 쿼리와 관련된 정보가 포함되지 않은 테이블에서 파일 열기를 건너뛸 수 있습니다. 이 공동 지역성은 Delta Lake 데이터 건너뛰기 알고리즘에서 자동으로 사용됩니다. 이 동작은 읽어야 하는 데이터의 양을 크게 줄입니다.

## Z-ORDER BY를 조정하려면:

- delta.dataSkippingNumIndexedCols (int)는 테이블 메타데이터에 저장되는 통계 열 수를 줄이는 역할을 하는 테이블 속성입니다. 기본값은 32개 열입니다.
- delta.checkpoint.writeStatsAsStruct(bool)는 열 형식 통계(트랜잭션별)를 쪽모아 제공 데이터로 쓸 수 있도록 하는 테이블 속성입니다. 기본값

모든 공급업체 기반 Delta Lake 솔루션이 구조체 기반 통계 읽기를 지원하는 것은 아니므로 값은 false입니다.



12장에서는 성능 튜닝에 대해 더 자세히 다룰 것이므로 지금부터 본격적으로 살펴보고 일반적인 유지 관리 고려 사항을 다루겠습니다.

## 테이블 튜닝 및 관리

방금 OPTIMIZE 명령을 사용하여 테이블을 최적화하는 방법을 다루었습니다. 대부분의 경우 테이블이 1GB보다 작으면 OPTIMIZE를 사용하는 것이 전혀 문제가 되지 않습니다. 그러나 시간이 지남에 따라 테이블이 커지는 것 이 일반적이므로 결국에는 다음 단계로 테이블 파티셔닝을 고려해야 합니다. 유지 보수를 위해.

### 테이블 파티션 나누기 테이블 파티션은

당신에게 도움이 될 수도 있고, 이상하게도 당신에게 불리할 수도 있습니다. 작은 파일 문제에서 관찰한 동작과 유사합니다. 파티션이 너무 많으면 비슷한 문제가 발생할 수 있지만 대신 디렉터리 수준 격리를 통해 발생합니다. 다른 행스럽게도 파티션을 효과적으로 관리하는 데 도움이 되거나 적어도 때가 되면 따라야 할 패턴을 제공하는 몇 가지 일반적인 지침과 규칙이 있습니다.

### 테이블 분할 규칙 다음 규칙

은 파티션을 도입할 시기를 이해하는 데 도움이 됩니다.

1. 테이블이 1TB보다 작은 경우. 파티션을 추가하지 마십시오. 최적화를 사용하여 파일 수를 줄이세요. bin-packing 최적화가 필요한 성능 향상을 제공하지 못하는 경우 다운스트림 데이터 고객과 대화하여 그들이 테이블에 일반적으로 쿼리하는 방법을 알아보고 z 순서 최적화를 사용하여 데이터 쿼리 속도를 높일 수 있습니다. 공동 위치.
2. 삭제 방법을 최적화해야 한다면? GDPR 및 기타 데이터 거버넌스 규칙은 테이블 데이터가 변경될 수 있음을 의미합니다. 대개 데이터 거버넌스 규칙을 준수한다는 것은 테이블에서 레코드를 삭제하는 방법을 최적화하거나 법적 보존의 경우처럼 테이블을 유지해야 함을 의미합니다.

간단한 사용 사례 중 하나는 N일 삭제(예: 30일 보존)입니다. Delta Lake 테이블의 크기에 따라 최적은 아니지만 일일 파티션을 사용하면 특정 시점보다 오래된 데이터와 같은 일반적인 삭제 패턴을 단순화하는 데 사용할 수 있습니다. 30일 삭제의 경우 datetime 열로 분할된 테이블이 있으면 `datetime < current\_timestamp() - 간격 30일인 {table}`를 호출하는 간단한 작업을 실행할 수 있습니다.

을바른 파티션 열을 선택하십시오. 다음

조언은 분할 시 사용할 을바른 열을 선택하는 데 도움이 됩니다. 가장 일반적으로 사용되는 파티션 열은 날짜입니다. 분할 기준으로 사용할 열을 결정하려면 다음 두 가지 경험 법칙을 따르세요.

1. 컬럼의 카디널리티가 매우 높은가요? 파티셔닝에 해당 열을 사용하지 마십시오. 예를 들어, userId 열을 기준으로 분할하고 100만 개 이상의 개별 사용자 ID가 있을 수 있다면 이는 잘못된 분할 전략입니다.
2. 각 파티션에는 얼마나 많은 데이터가 존재합니까? 해당 파티션의 데이터가 1GB 이상일 것으로 예상되는 경우 열을 기준으로 파티션을 나눌 수 있습니다.

을바른 파티셔닝 전략이 즉시 나타나지 않을 수도 있지만 괜찮습니다. 을바른 사용 사례(및 데이터)가 앞에 있을 때 까지는 최적화할 필요가 없습니다.

너.

방금 설정한 규칙을 바탕으로 테이블 생성 시 파티션 정의, 기존 테이블에 파티션 추가, 파티션 제거(삭제) 등의 사용 사례를 살펴보겠습니다. 이 프로세스는 파티셔닝 사용에 대한 확실한 이해를 제공할 것이며 결국 이는 Delta Lake 테이블의 장기적인 예방 유지 관리에 필요합니다.

테이블 생성 시 파티션 정의 날짜 열을 사용하여 개입 없이 테

이블에 새 파티션을 자동으로 추가하는 default.covid\_nyt\_by\_day라는 새 테이블을 생성해 보겠습니다 .

```
$ from pyspark.sql.types import DateType from
  delta.tables import
  DeltaTable(DeltaTable.createIfNotExists(spark)
    .tableName("default.covid_nyt_by_date")
    ...
    .addColumn("날짜", DateType(), nullable=False) .partitionedBy("날
    짜") .addColumn("county",
      "STRING") .addColumn("state",
      "STRING") .addColumn("fips",
      "INT") .addColumn("cases",
      "INT") .addColumn("죽음",
      "INT") .execute()
```

생성 논리에서 진행되는 작업은 마지막 몇 가지 예와 거의 동일합니다. 차이점은 DeltaTable 빌더에 partitionBy("date") 를 도입했다는 것입니다. 날짜 열이 항상 존재하도록 하기 위해 DDL에는 분할에 열이 필요 하므로 null을 허용하지 않는 플래그가 포함됩니다.

파티셔닝을 위해서는 테이블을 나타내는 실제 파일이 파티션당 고유한 디렉터리를 사용하여 배치되어야 합니다. 이는 모든 물리적 테이블 데이터가

파티션 규칙을 준수하기 위해 이동해야 합니다. 분할되지 않은 테이블에서 분할된 테이블로 마이그레이션하는 것은 어렵지 않지만 라이브 다운스트림 고객을 지원하는 것은 약간 까다로울 수 있습니다.

일반적으로 현재 테이블에 잠재적인 주요 변경 사항을 도입하는 것보다 기존 데이터 고객을 새 테이블(이 예에서는 분할된 새 테이블)로 마이그레이션하는 계획을 세우는 것이 항상 더 좋습니다. 모든 활동적인 독자를 위해.

현재 모범 사례를 바탕으로 다음 단계에서 이를 수행하는 방법을 알아보겠습니다.

분할되지 않은 테이블에서 분할된 테이블로 마이그레이션 분할된 테이블에 대한 테이블 정의가 있으면 분할되지 않은 테이블에서 모든 데이터를 읽고 새로 생성된 테이블에 행을 쓰는 것이 간단해집니다. 더욱 쉬운 점은 파티션 전략이 테이블 메타데이터에 이미 존재하므로 파티션 분할 방법을 지정할 필요조차 없다는 것입니다.

```
$ (
    스파
```

```
    .table("default.covid_nyt") .write .format("delta") .mode("append") .option("mergeSchema", "false") .saveAsTable("default.covid_nyt_by_date")
```

이 과정에서 도로에 갈림길이 생깁니다. 현재 우리는 이전 버전의 테이블(파티션되지 않은)과 새(파티션된) 테이블을 갖고 있으며 이는 복사본이 있음을 의미합니다. 일반적인 컷오프 중에는 일반적으로 고객이 완전히 마이그레이션할 준비가 되었다고 알릴 때까지 이중 쓰기를 계속해야 합니다. 9장에서는 보다 자동적인 충분 병합을 수행하는데 유용한 몇 가지 요령을 제공하며 이전 테이블의 두 버전을 동기화 상태로 유지하려면 병합 및 충분 처리를 사용하는 것이 좋습니다.

## 파티션 메타데이터 관리 Delta Lake는

새 데이터가 삽입되고 이전 데이터가 삭제될 때 자동으로 테이블 파티션을 생성하고 관리하므로 ALTER TABLE table\_name [ADD | DROP PARTITION] (열=값). 즉, 테이블 메타데이터를 테이블 자체 상태와 동기화하기 위해 수동으로 작업하는 대신 다른 곳에 시간을 집중할 수 있습니다.

## 파티션 메타데이터 보기 파티션

정보와 기타 테이블 메타데이터를 보려면 테이블에 대한 새 DeltaTable 인스턴스를 생성하고 Detail 메소드를 호출하면 됩니다. 이 반환됩니다.

전체를 보거나 보려는 열까지 필터링할 수 있는 DataFrame입니다.

```
$ (DeltaTable.forName(spark,"default.covid_nyt_by_date") .detail() .toJSON() .collect()
[0]

)
```

위 명령은 결과 DataFrame을 JSON 객체로 변환한 다음 이를 목록( collect() 사용)으로 변환하므로 JSON 데이터에 직접 액세스할 수 있습니다.

```
{
  "형식": "델타", "id": "8c57bc67-369f-4c84-a63e-38b8ac19bdf2", "이름": "default.covid_nyt_by_date",
  "위치": "파일:/opt/spark/work-dir/ch6/spark-warehouse/
covid_nyt_by_date", "createdAt": "2023-06-08T05:35:00.072Z", "lastModified": "2023-06-08T05:50:45.241Z", "partitionColumns": ["date "], "numFiles": 423, "sizeInBytes": 17660304, "properties": {
    "description": "기본 파티션이 있는 테이블", "catalog.table.classification": "all-access", "catalog.engineering.comms.email": "dldg_authors@gmail.com",
    "catalog.team_name": "dldg_authors",
    "catalog.engineering.comms.slack": "https://delta-users.slack.com/archives/
CG9LR6LN4" }, "minReaderVersion": 1, "minWriterVersion": 2, "tableFeatures": ["appendOnly", "invariants"]
}

}
```

파티셔닝 도입이 완료되었으므로 이제 Delta Lake 테이블 수명 주기 및 유지 관리의 두 가지 중요한 기술인 테이블 수리 및 교체에 집중할 때입니다.

## 테이블 데이터 복구, 복원 및 교체

현실을 직시하자. 최선의 의도가 있더라도 우리는 모두 인간이고 실수를 합니다. 데이터 엔지니어로서의 경력에서 배워야 할 것 중 하나는 데이터 복구 기술입니다. 데이터를 복구할 때 우리가 취하는 작업은 시계를 이전 시점으로 롤백하거나 되감는 것이므로 이 프로세스를 일반적으로 재생이라고 합니다. 이를 통해 테이블에서 문제가 있는 변경 사항을 제거하고 잘못된 데이터를 "수정된" 데이터로 바꿀 수 있습니다.

## 테이블 복구 및 교체 테이블을 복구할 수 있

을 때 중요한 점은 현재 테이블보다 더 나은 상태에 있는 사용 가능한 데이터 소스가 있어야 한다는 것입니다. 11장에서는 원시(브론즈), 정제된(실버) 및 선별된(골드) 데이터 세트 간의 명확한 품질 경계를 정의하는 데 사용되는 메달리온 아키텍처에 대해 알아봅니다.

이 장에서는 실버 데이터베이스 테이블에서 손상된 데이터를 대체하는 데 사용할 수 있는 원시 데이터가 브론즈 데이터베이스 테이블에 있다고 가정합니다.

예를 들어 2021년 2월 17일 테이블에서 데이터가 실수로 삭제되었다고 가정해 보겠습니다.

실수로 삭제된 데이터를 복원하는 다른 방법(다음에 배우겠습니다)이 있지만 데이터가 영구적으로 삭제된 경우 당황할 이유가 없으며 복구 데이터를 가져와 조건부 덮어쓰기를 사용할 수 있습니다.

```
$ Recovery_table = 스파크.테이블("bronze.covid_nyt_by_date")
partition_col = "날짜" partition_to_fix
"2021-02-17" table_to_fix =
"silver.covid_nyt_by_date"
```

```
(복구_테이블
    .where(col(partition_col) == partition_to_fix) .write .format("delta") .mode("덮어쓰기")
    .option("replaceWhere",
    f'{partition_col} =='
    {partition_to_fix}).saveAsTable("silver.covid_nyt_by_date")
```

)

이전 코드는 누락된 데이터를 대체하거나 테이블에서 조건에 따라 기존 데이터를 덮어쓸 수 있는 대체 덮어쓰기 패턴을 보여줍니다. 이 옵션을 사용하면 손상되었을 수 있는 테이블을 수정하거나 데이터가 누락되어 사용할 수 있게 된 문제를 해결할 수 있습니다. 삽입 덮어쓰기가 포함된 교체 위치는 파티션 열에만 바인딩되지 않으며 테이블의 데이터를 조건부로 바꾸는 데 사용할 수 있습니다.



교체 위치 조건이 복구 테이블의 where 절과 일치하는지 확인하는 것이 중요합니다. 그렇지 않으면 더 큰 문제가 발생하고 수정 중인 테이블이 더욱 손상될 수 있습니다. 가능하면 사람의 실수가 발생할 가능성을 제거하는 것이 좋습니다. 따라서 테이블의 데이터를 자주 복구(교체 또는 복구)하는 경우 테이블의 무결성을 보호하기 위해 몇 가지 가드레일을 만드는 것이 좋습니다.

다음으로 전체 파티션을 조건부로 제거하는 방법을 살펴보겠습니다.

데이터 삭제 및 파티션 제거 특정 요청을 이행하기 위해 Delta Lake 테이블에서 특정 파티션을 제거하는 것이 일반적입니다. 예를 들어 특정 시점보다 오래된 데이터를 삭제하고, 비정상적인 데이터를 제거하고, 일반적으로 테이블을 정리할 때입니다.

어떤 경우이든, 주어진 파티션을 단순히 지우려는 의도라면 파티션 열에 대한 조건부 삭제를 사용하여 그렇게 할 수 있습니다. 다음은 2023년 1월 1일보다 오래된 파티션(`tpep_dropoff_date`)을 조건부로 삭제합니다.

```
(  
    델타 테이블  
    .forName(spark, 'default.covid_nyt_by_date') .delete(col("date")  
    < "2023-01-01"))
```

데이터 제거 또는 전체 파티션 삭제는 모두 조건부 삭제를 사용하여 관리할 수 있습니다. 파티션 열을 기준으로 삭제하는 경우 이는 물리적 테이블 데이터를 메모리에 로드하는 처리 오버헤드 없이 데이터를 삭제하는 효율적인 방법이며, 대신 테이블 메타데이터에 포함된 정보를 사용하여 조건자를 기준으로 파티션을 정리합니다. 분할되지 않은 열을 기반으로 삭제하는 경우 부분 또는 전체 테이블 스캔이 발생할 수 있으므로 비용이 더 높습니다. 그러나 추가 보너스로 전체 파티션을 제거하든 각 테이블의 하위 집합을 조건부로 제거하든 상관 없습니다. 마음을 바꿔야 하는 경우 시간 여행을 사용하여 작업을 "실행 취소"할 수 있습니다. 다음에는 테이블을 이전 시점으로 복원하는 방법을 알아보겠습니다.



델타 레이크 작업 컨텍스트 외부에서 델타 레이크 테이블 데이터(파일)를 제거하지 마십시오.  
오. 이로 인해 테이블이 손상되고 두통이 발생할 수 있습니다.

## Delta Lake 테이블의 수명 주기

시간이 지남에 따라 각 델타 테이블이 수정되면 테이블 복원을 지원하거나 테이블 시간(시간 이동)의 이전 지점을 확인하고 스트리밍 작업에 대한 깔끔한 환경을 제공하기 위해 테이블의 이전 버전이 디스크에 남아 있습니다. 테이블의 다양한 지점(다른 시점 또는 테이블 전체의 기록과 관련됨)에서 읽을 수 있습니다. 그렇기 때문에 `delta.logRetentionDuration`에 대해 충분히 긴 전환 확인 기간을 확보하는 것이 중요합니다. 그러면 테이블에서 진공 청소기를 실행할 때 방금 사라진 데이터 스트림의 페이지가 즉시 넘치거나 고객이 만족하지 않게 됩니다.

## 테이블 복원 트랜잭션이 발생

한 경우(예: 데이터 사본이 있는 경우) 데이터를 다시 로드하는 대신 테이블에서 잘못된 삭제(인생 문제로 인해)가 발생한 경우 다음을 수행할 수 있습니다. 테이블을 되감고 이전 버전으로 복원합니다. 이는 데이터의 유일한 복사본이 실제로 방금 삭제된 데이터인 경우 문제가 발생할 수 있다는 점을 고려하면 특히 중요한 기능입니다. 데이터를 복구할 곳이 더 이상 남아 있지 않은 경우 테이블의 이전 버전으로 시간 여행을 할 수 있습니다.

테이블을 복원하는 데 필요한 것은 몇 가지 추가 정보입니다. 우리는 테이블 기록에서 이 모든 것을 얻을 수 있습니다.

```
$ dt = DeltaTable.forName(spark, "silver.covid_nyt_by_date")
(dt.역사(10)
.select("버전", "타임스탬프", "작업").show())
```

이전 코드는 Delta Lake 테이블의 마지막 10개 작업을 표시합니다. 이전 버전으로 되돌리려면 DELETE를 찾으세요.

버전	타임스탬프	운영
1 2023-06-09 19:11:... 삭제  0 2023-06-09 19:04:... S로 테이블 생성...		

버전 1에서 DELETE 트랜잭션이 발생한 것을 볼 수 있으므로 테이블을 다시 버전 0으로 복원하겠습니다.

```
$ dt.restoreToVersion(0)
```

테이블을 복원하는 데 필요한 것은 제거하려는 작업에 대한 지식뿐입니다. 우리의 경우 DELETE 트랜잭션을 제거했습니다. Delta Lake 삭제 작업은 테이블 메타데이터에서 발생하므로 VACUUM이라는 프로세스를 실행하지 않는 한 테이블의 이전 버전으로 안전하게 돌아갈 수 있습니다.

## 청소

Delta Lake 테이블에서 데이터를 삭제할 때 이 작업은 즉시 이루어지지 않습니다. 실제로 작업 자체는 단순히 Delta Lake 테이블 스냅샷에서 참조를 제거하므로 이제 데이터가 보이지 않는 것과 같습니다. 이 작업은 데이터가 실수로 삭제된 경우 "실행 취소"할 수 있음을 의미합니다. "vacuuming"이라는 프로세스를 사용하여 아티팩트와 삭제된 파일을 정리하고 델타 레이크 테이블에서 완전히 제거할 수 있습니다.

## 진공

Vacuum 명령은 삭제된 파일이나 더 이상 최신이 아닌 테이블 버전을 정리합니다. 이는 테이블에서 덮어쓰기 방법을 사용할 때 발생할 수 있습니다. 테이블을 덮어쓰는 경우 실제로 수행하는 작업은 테이블 메타데이터에서 참조하는 새 파일에 대한 새 포인터를 만드는 것뿐입니다. 따라서 테이블을 자주 덮어쓰면 디스크의 테이블 크기가 기하급수적으로 늘어납니다. 다행히 시간이 지남에 따라 변경 사항이 발생할 때 테이블의 동작을 제어하는 데 도움이 되는 몇 가지 테이블 속성이 있습니다. 이러한 규칙은 진공 프로세스에 적용됩니다.

- `delta.logRetentionDuration`의 기본값은 30일 간격이며 테이블 기록을 추적합니다. 더 많은 작업이 발생할 수록 더 많은 기록이 유지됩니다. 시간 이동 작업을 사용하지 않을 경우 기록 일수를 일주일로 줄여 볼 수 있습니다.
- `delta.deletedFileRetentionDuration`의 기본값은 1주 간격이며 삭제 작업이 실행 취소되지 않을 것으로 예상되는 경우 변경할 수 있습니다. 마음의 평화를 위해 삭제된 파일을 보관하려면 최소 1일을 유지하는 것이 좋습니다.

테이블에 테이블 속성이 설정되어 있으면 Vacuum 명령이 대부분의 작업을 수행합니다. 다음 코드 예제에서는 진공 작업을 실행하는 방법을 보여줍니다.

```
$ (DeltaTable.forName(spark, "default.nonoptimal_covid_nyt")
    .진공())
```

테이블에서 Vacuum을 실행하면 이전 버전의 테이블에서 삭제된 파일을 포함하여 테이블 스냅샷에서 더 이상 참조하지 않는 모든 파일이 제거됩니다. Vacuuming은 특정 테이블의 이전 버전을 유지 관리하는 비용을 줄이기 위해 필요한 프로세스이지만, 초기 버전을 읽어야 하는 경우 다운스트림 데이터 소비자(소비자)가 실수로 높은 상태로 남을 수 있는 부작용이 있습니다. 당신의 테이블 Delta Lake 테이블 안팎으로 스트리밍 데이터를 처리할 때 9장에서 다룬 문제가 발생할 수 있습니다.



진공 명령은 자체적으로 실행되지 않습니다. 테이블을 프로덕션 환경으로 가져올 계획이고 테이블을 깔끔하게 유지하는 프로세스를 자동화하려는 경우 크론 작업을 설정하여 일반적인 주기(매일, 매주)로 Vacuum을 호출할 수 있습니다. 또한 Vacuum은 파일이 디스크에 기록될 때 파일의 타임스탬프에 의존한다는 점을 지적할 가치가 있습니다. 따라서 전체 테이블을 가져온 경우 Vacuum 명령은 보존 임계값에 도달할 때까지 아무 작업도 수행하지 않습니다.

## 테이블 삭제 테이

블 삭제는 실행 취소할 수 없는 작업입니다. `{테이블}`에서 삭제`를 실행하면 본질적으로 테이블이 잘 리고 시간 여행을 계속 활용할 수 있습니다(작업 실행 취소). 하지만 테이블의 모든 흔적을 실제로 제거하려면 다음 경고를 읽어보세요. 상자를 열고 테이블 복사본(또는 **CLONE**)을 생성하여 미리 계획하는 것을 잊지 마세요. 회복 전략을 원한다면.



테이블 삭제는 실행 취소할 수 없는 작업입니다. `{테이블}`에서 삭제`를 실행하면 기본적으로 테이블이 잘 리고 시간 이동을 활용하여 변경을 취소할 수 있습니다. 정말로 테이블의 모든 흔적을 제거하고 싶다면 이 장을 마무리하고 그 방법을 보여줄 것입니다.

### Delta Lake 테이블의 모든 추적 제거 영구 삭제를 수행

하고 관리되는 Delta Lake 테이블의 모든 추적을 제거하고 수행 중인 작업과 관련된 위험을 이해하고 실제로 다음 가능성을 포기하려는 경우 테이블을 복구한 다음 SQL `DROP TABLE` 구문을 사용하여 테이블을 삭제할 수 있습니다.

```
$ 스파크.sql(f"drop silver.covid_nyt_by_date")
```

Delta Lake 테이블의 파일을 나열해 보면 테이블이 사라진 것을 확인할 수 있습니다.

```
$ docker exec \ -it
    delta_quickstart bash \ -c "ls -l /opt/
    spark/work-dir/ch6/spark-warehouse/sil-
    ver.db/covid_nyt_by_date/
```

결과는 다음과 같습니다. 이는 테이블이 실제로 더 이상 디스크에 존재하지 않음을 나타냅니다.

```
ls: './spark-warehouse/sil-ver.db/covid_nyt_by_date/'에
액세스할 수 없습니다. 해당 파일이나 디렉터리가 없습니다.
```

## 요약

이 장에서는 Delta Lake 프로젝트 내에서 제공되는 일반적인 유틸리티 기능을 소개했습니다. 우리는 테이블 속성으로 작업하는 방법을 배웠고, 가장 흔히 접할 수 있는 보다 일반적인 테이블 속성을 탐색했으며, 작은 파일 문제를 해결하기 위해 테이블을 최적화하는 방법을 배웠습니다. 이를 통해 테이블 내의 데이터를 분할하고 복원하고 교체하는 방법을 배울 수 있었습니다. 우리는 시간 여행을 사용하여 테이블을 복원하는 방법을 살펴보고, 우리 자신을 정리하고 마지막으로 더 이상 필요하지 않은 테이블을 영구적으로 삭제하는 방법에 대해 알아보며 장을 마무리했습니다.

모든 사용 사례가 책에 딱 들어맞을 수는 없지만 이제 Delta Lake 테이블을 유지 관리하고 시간이 지나도 원활하게 실행되도록 유지하는 데 필요한 일반적인 문제와 솔루션에 대한 훌륭한 참고 자료가 있습니다.



## 제4장

# Delta Lake에서 스트리밍 및 스트리밍

초기 출시 독자를 위한 참고 사항 초기 출시 eBook을 사용

하면 가장 초기 형태의 책, 즉 저자가 집필하는 그대로의 편집되지 않은 원본 콘텐츠를 얻을 수 있으므로 이러한 타이틀이 공식 출시되기 훨씬 전에 이러한 기술을 활용할 수 있습니다.

이것이 마지막 책의 아홉 번째 장이 될 것이다. GitHub 저장소는 나중에 활성화될 예정입니다.

이 책의 내용 및/또는 예제를 개선할 수 있는 방법에 대한 의견이 있거나 이 장에 누락된 자료가 있는 경우 [gobrien@oreilly.com](mailto:gobrien@oreilly.com)으로 편집자에게 문의하십시오.

이제 전 세계는 그 어느 때보다 실시간 데이터 소스로 가득 차 있습니다. 전자상거래, 소셜 네트워크 피드, 항공 비행 데이터부터 네트워크 보안 및 IoT 장치에 이르기까지 데이터 소스의 양은 증가하는 반면 데이터를 사용할 수 있는 빈도는 급격히 감소하고 있습니다. 이에 대한 한 가지 문제는 일부 이벤트 수준 작업이 의미가 있지만 우리가 의존하는 많은 정보가 해당 정보의 집계에 존재한다는 것입니다. 따라서 우리는 a.) 통찰력을 얻는 시간을 최대한 줄이는 것과 b.) 집계에서 충분히 의미 있고 실행 가능한 정보를 캡처하는 두 가지 우선 순위 사이에 갇혀 있습니다. 수년 동안 우리는 처리 기술이 이 방향으로 이동하는 것을 보았고 이것이 바로 Delta Lake가 탄생한 환경이었습니다. 우리가 Delta Lake에서 얻은 것은 대부분의 분산 데이터 저장소에 일반적으로 없는 ACID 트랜잭션 및 확장 가능한 메타데이터 처리와 같은 필수 기능을 제공하면서 여러 배치 및 스트림 프로세스의 원활한 통합을 지원하는 개방형 레이크하우스 형식이었습니다. 이를 염두에 두고 Delta Lake를 사용한 스트림 처리에 대한 세부 정보, 즉 스트리밍 프로세스의 핵심 기능, 구성 옵션,

특정 사용 방법 및 Delta Lake와 Databricks의 Delta Live Tables의 관계.

## 스트리밍 및 Delta Lake

진행하면서 몇 가지 기본 개념을 다룬 다음 실제로 스트림 처리에 Delta Lake를 사용하는 방법에 대해 더 자세히 알아보고 싶습니다. 개념과 몇 가지 용어에 대한 개요부터 시작한 후 Delta Lake와 함께 사용할 수 있는 몇 가지 스트림 처리 프레임워크를 살펴보겠습니다. 스트림 처리에 대한 자세한 소개는 Learning Spark 책을 참조하세요.

그런 다음 핵심 기능, 사용 가능한 일부 옵션, Apache Spark의 몇 가지 일반적인 고급 사례를 살펴보겠습니다. 그런 다음 마무리하기 위해 Delta Live Tables와 같은 Databricks에서 사용되는 몇 가지 관련 기능과 이것이 Delta Lake와 어떻게 관련되는지 다루고 마지막으로 Delta Lake에서 사용할 수 있는 변경 데이터 피드 기능을 사용하는 방법을 검토합니다.

### 스트리밍 대 일괄 처리 개념으로서의 데이터 처리

는 우리에게 의미가 있습니다. 수명 주기 동안 데이터를 수신하고, 이에 대해 다양한 작업을 수행한 다음, 저장하거나 전달합니다. 그렇다면 일괄 데이터 프로세스와 스트리밍 데이터 프로세스의 주요 차이점은 무엇입니까? 자연 시간. 무엇보다도 대기 시간이 주요 동인입니다. 이러한 프로세스는 설계 이면의 비즈니스 논리가 다르지 않고 대신 메시지/파일 크기 및 처리 속도에 중점을 두는 경향이 있기 때문입니다. 어떤 방법을 사용할지는 일반적으로 프로젝트 시작 시 수집되는 요구 사항의 일부가 되어야 하는 시간 요구 사항이나 서비스 수준/전달 계약에 따라 선택됩니다. 요구 사항은 또한 데이터에서 실행 가능한 통찰력을 얻는 데 필요한 시간을 고려해야 하며 처리 방법에 대한 결정을 내리는 데 도움이 됩니다. 우리가 선호하는 추가 설계 선택 중 하나는 처리 논리의 차이가 거의 없기 때문에 통합 배치 및 스트리밍 API가 있는 프레임워크를 사용하고 요구 사항이 변형될 경우 유연성을 제공하는 것입니다.

시간이 지남에 따라.

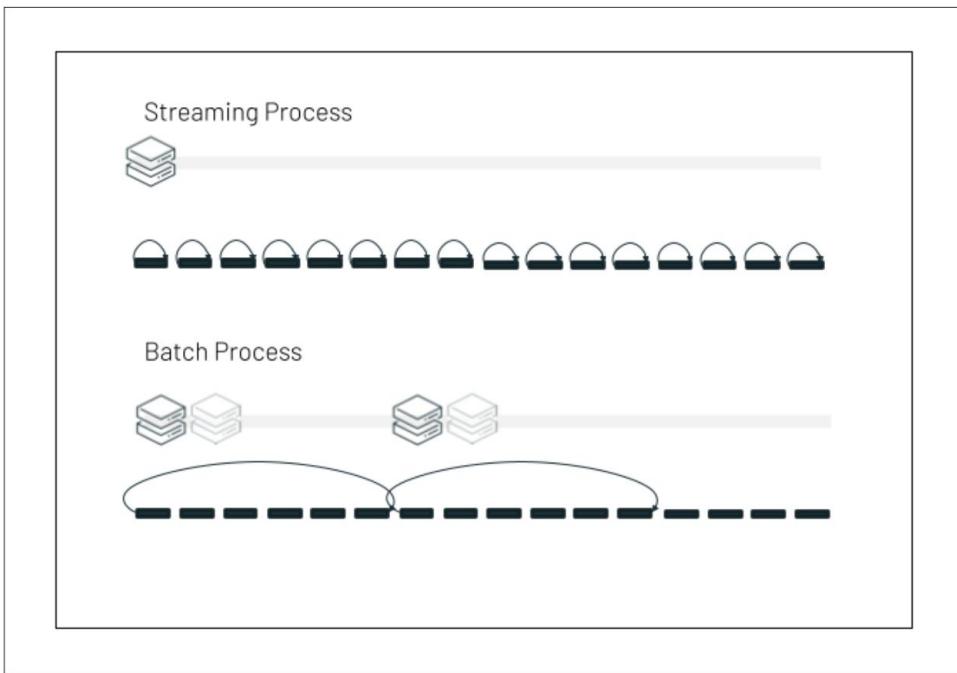


그림 4-1. 일괄 처리와 스트림 처리의 가장 큰 차이점은 대기 시간입니다. 각 개별 파일이나 메시지가 사용 가능해지면 처리하거나 그룹으로 처리할 수 있습니다.

배치 프로세스는 시작점과 끝점을 정의합니다. 즉, 시간과 형식 측면에서 경계가 설정됩니다. 당사는 일괄 처리로 "파일" 또는 "파일 세트"를 처리할 수 있습니다. 스트림 처리에서는 데이터를 약간 다르게 보고 대신 데이터를 무한하고 연속적인 것으로 취급합니다. 저장소에 도착하는 파일의 경우에도 지속적으로 도착하는 파일 스트림(예: 로그 데이터)을 생각할 수 있습니다. 결국 이 무한성을 소스를 데이터 스트림으로 만드는 데 필요한 전부입니다.

**그림 4-1**에서 일괄 처리는 스트림 프로세스가 항상 실행되고 사용 가능한 각 파일을 처리하는 예약된 각 실행에 대한 6개 파일의 처리 그룹과 동일합니다.

Delta Lake를 사용할 수 있는 일부 프레임워크를 비교할 때 곧 살펴보겠지만 Apache Flink 또는 Apache Spark와 같은 스트림 처리 엔진은 데이터 스트림의 시작점 또는 종료 대상으로 Delta Lake와 함께 작동할 수 있습니다. 이러한 다중 역할은 Delta Lake가 다양한 종류의 스트리밍 워크로드의 여러 단계에서 사용될 수 있음을 의미합니다. 종종 우리는 두 종류의 작업이 모두 발생하는 더 복잡한 데이터 파이프라인의 여러 단계에 대해 스토리지 계층과 처리 엔진이 존재하는 것을 볼 수 있습니다. 대부분의 스트림 처리 엔진의 공통된 특징 중 하나는 단지 처리 엔진이라는 것입니다. 스토리지와 컴퓨팅을 분리한 후에는 각각을 고려하고 선택해야 하지만 둘 다 독립적으로 작동할 수는 없습니다.

실용적인 관점에서 처리 시간 및 테이블 유지 관리와 같은 다른 관련 개념에 대해 생각하는 방식은 배치 또는 스트리밍 중 하나를 선택하는 것에 영향을 받습니다. 배치 프로세스가 특정 시간에 실행되도록 예약된 경우 프로세스 실행 시간, 처리된 데이터 양을 쉽게 측정하고 이를 추가 프로세스와 연결하여 테이블 유지 관리 작업을 처리할 수 있습니다.

스트림 프로세스를 측정하고 유지 관리할 때 약간 다르게 생각해야 하지만 자동 압축 및 최적화된 쓰기와 같이 이미 살펴본 많은 기능은 실제로 두 영역 모두에서 작동할 수 있습니다. [그림 4-2](#)에서 우리는 최신 시스템을 통해 배치와 스트리밍이 어떻게 서로 수렴할 수 있는지 확인할 수 있으며 대신 기존 프레임워크에서 벗어나 자연 시간의 균형에 집중할 수 있습니다. 배치 및 스트리밍 사용 사례 모두에 대한 프로그래밍 차이를 최소화하고 유지 관리 작업을 단순화하고 두 처리 방법 중 하나를 제공하는 Delta Lake와 같은 스토리지 형식 위에서 실행하는 합리적으로 통합된 API가 있는 프레임워크를 선택함으로써 우리는 마무리합니다. 모든 데이터 처리 작업을 처리하고 여러 도구의 균형을 유지해야 하는 필요성을 최소화하며 여러 시스템을 실행하는 데 필요한 기타 합병증을 피할 수 있는 더욱 강력하면서도 유연한 시스템입니다. 따라서 Delta Lake는 스트리밍 워크로드를 위한 이상적인 스토리지 솔루션입니다. 다음으로 스트림 처리 애플리케이션에 대한 몇 가지 특정 용어를 고려하고 Delta Lake에서 사용할 수 있는 몇 가지 다양한 프레임워크 통합을 검토하겠습니다.

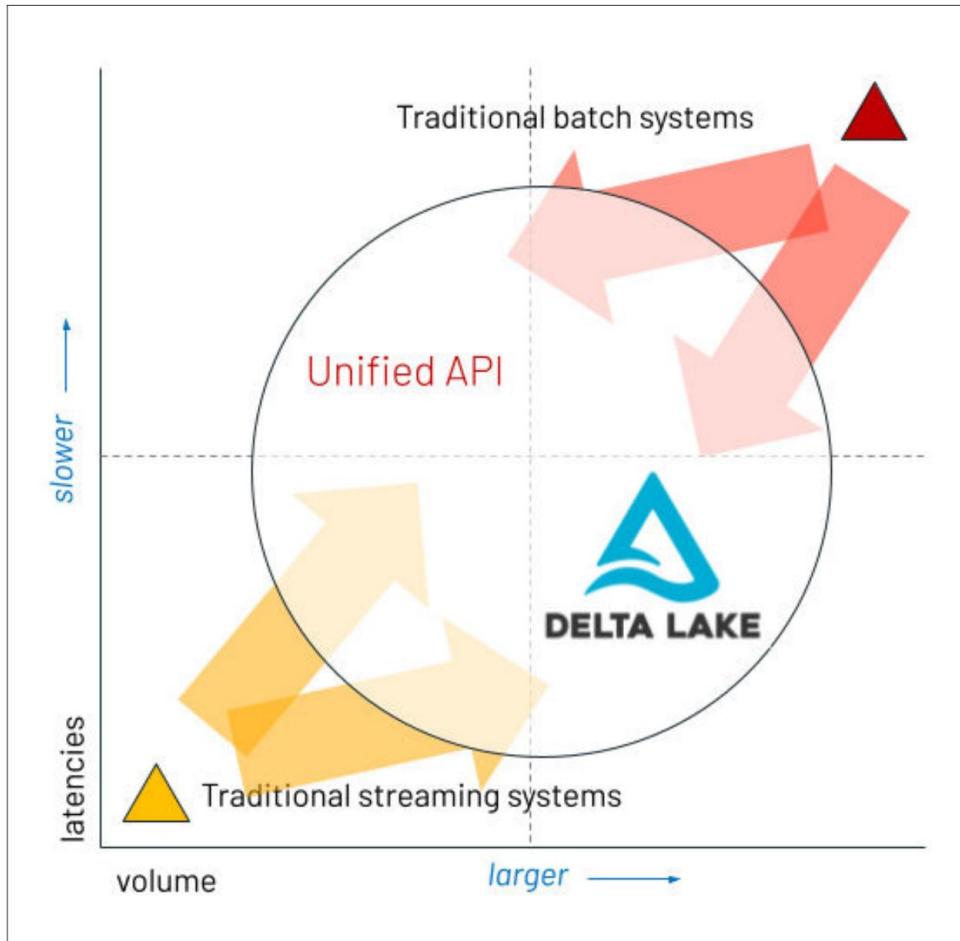


그림 4-2. 스트리밍 및 배치 프로세스는 최신 시스템에서 겹칩니다.

#### 스트리밍 용어 여러 면에

서 스트리밍 프로세스는 대부분 대기 시간과 흐름의 차이점을 제외하면 배치 프로세스와 상당히 동일합니다. 그러나 이것이 스트리밍 프로세스에 자체 용어가 포함되어 있지 않다는 의미는 아닙니다. 일부 용어는 소스 및 싱크와 같이 배치 사용과 약간만 다르지만 체크포인트 및 워터마크와 같은 다른 용어는 실제로 배치에 적용되지 않습니다. 이러한 용어에 대해 어느 정도 익숙해지면 유용하지만 Apache Flink를 사용한 스트림 처리 또는 Learning Spark에서 더 깊이 있게 알아볼 수 있습니다.

원천. 스트림 처리 소스는 무제한 데이터 세트로 처리될 수 있는 다양한 데이터 소스 중 하나입니다. 데이터 스트림 처리의 소스는 다양하며 궁극적으로 염두에 두고 있는 처리 작업의 성격에 따라 달라집니다. 숫자가 있습니다 -

Spark 및 Flink 생태계 전반에 걸쳐 데이터 소스로 사용되는 다양한 메시지 대기열 및 게시-구독 커넥터. 여기에는 Apache Kafka, Amazon Kinesis, ActiveMQ, RabbitMQ, Azure Event Hubs, Google Pub/Sub 등 널리 사용되는 다양한 기능이 포함됩니다. 두 시스템 모두 예를 들어 새 파일에 대한 클라우드 저장소 위치를 모니터링하여 파일에서 스트림을 생성할 수도 있습니다. Delta Lake가 스트리밍 데이터 원본으로 어떻게 적합한지 곧 살펴보겠습니다.

싱크대. 스트림 데이터 처리 싱크도 마찬가지로 모양과 형태가 다양합니다.

우리는 종종 동일한 메시지 대기열과 게시-구독 시스템을 많이 볼 수 있지만 특히 싱크 측에서는 키-값 저장소, RDBMS 또는 AWS S3 또는 Azure ADLS와 같은 클라우드 스토리지와 같은 일부 구체화 계층을 자주 발견합니다. 일반적으로 최종 목적지는 일반적으로 후자 범주 중 하나이며 출발지에서 목적지까지 중간에 몇 가지 방법이 혼합된 것을 볼 수 있습니다. Delta Lake는 특히 대용량, 높은 처리량의 스트리밍 수집 프로세스를 관리하기 위한 싱크 역할을 매우 잘 수행합니다.

검문소. 체크포인트는 일반적으로 스트리밍 프로세스에서 구현되었는지 확인하는 중요한 작업입니다. 체크포인트는 처리 작업의 진행 상황을 추적하고 매번 처음부터 처리를 다시 시작하지 않고도 오류 복구를 가능하게 해줍니다. 이는 스트림 오프셋 및 관련 상태 정보에 대한 일부 추적 기록을 유지함으로써 수행됩니다. Flink 및 Spark와 같은 일부 처리 엔진에는 검사점 작업을 더 쉽게 사용할 수 있는 메커니즘이 내장되어 있습니다. 자세한 사용법은 해당 문서를 참조하세요.

Spark의 예를 살펴보겠습니다. 스트림 쓰기 프로세스를 시작하고 적절한 체크포인트 위치를 정의하면 백그라운드에서 대상 위치에 몇 개의 디렉토리가 생성됩니다. 이 예에서는 'gold'라는 프로세스에서 작성된 체크포인트를 발견하고 디렉토리 이름도 비슷하게 지정했습니다.

트리 -L 1 /…/ckpt/gold/

```
/…/ckpt/gold/
├── __tmp_path_dir
├── 커밋
├── 메타데이터
└── 오프셋
└── 상태
```

메타데이터 디렉토리에는 스트리밍 쿼리에 대한 일부 정보가 포함되고 상태 디렉토리에는 쿼리와 관련된 상태 정보(있는 경우)의 스냅샷이 포함됩니다. 오프셋 및 커밋 디렉토리는 소스에서 스트리밍 진행 상황을 마이크로 배치 수준에서 추적하고 Delta Lake의 경우 입력 또는 출력 파일을 각각 추적하는데 필요한 프로세스에 대해 씁니다.

양수표. 워터마킹은 처리되는 기록에 상대적인 시간 개념입니다. 주제와 사용법은 논의하기에는 다소 복잡하므로 사용법에 대한 적절한 문서를 검토하는 것이 좋습니다. 제한된 목적을 위해 작업 정의를 사용할 수 있습니다. 기본적으로 워터마크는 처리 중에 데이터가 얼마나 늦게 받아들여질 수 있는지에 대한 제한입니다. 특히 윈도우 집계 작업과 함께 사용됩니다.<sup>1</sup>

### Apache Flink

Apache Flink은 제한된 데이터 조작과 무제한 데이터 조작을 모두 지원하는 주요 분산형 메모리 내 처리 엔진 중 하나입니다. Flink는 사전 정의되고 내장된 다양한 데이터 스트림 소스 및 싱크 커넥터를 지원합니다.<sup>2</sup> 데이터 소스 측면에서는 RabbitMQ, Apache Pulsar 및 Apache Kafka와 같이 지원되는 많은 메시지 대기열 및 개시-구독 커넥터를 볼 수 있습니다(문서 참조). 더 자세한 스트리밍 커넥터 정보를 보려면 Kafka와 같은 일부는 출력 대상으로 지원되지만 파일 저장소나 Elasticsearch에 쓰기 또는 데이터베이스에 대한 JDBC 연결과 같은 것을 목표로 보는 것이 가장 일반적일 것입니다. Flink 커넥터에 대한 자세한 내용은 해당 설명서에서 확인할 수 있습니다.

Delta Lake를 통해 우리는 Flink의 또 다른 소스와 대상을 얻게 되지만 이는 다중 도구 하이브리드 생태계에서 중요하거나 논리적 처리 전환을 단순화할 수 있습니다. 예를 들어 Flink를 사용하면 이벤트 스트림 처리에 집중한 다음 Spark에서 후속 처리를 위해 액세스 할 수 있는 클라우드 스토리지의 델타 테이블에 직접 쓸 수 있습니다. 또는 이 상황을 완전히 반전시키고 Delta Lake의 레코드에서 메시지 큐를 공급할 수 있습니다. 구현 및 아키텍처 세부 사항을 모두 포함한 커넥터에 대한 보다 심층적인 검토는 [delta.io 웹 사이트의 블로그 게시물](#)에서 확인할 수 있습니다.

### Apache Spark

Apache Spark는 유사하게 다양한 입력 소스와 싱크를 지원합니다.<sup>3</sup> Apache Spark는 대규모 수집 및 ETL 측면에서 더 많은 위치를 차지하는 경향이 있기 때문에 사용 가능한 입력 소스의 방향이 약간 편향되는 것을 볼 수 있습니다. 이벤트 처리 중심의 Flink 시스템보다 파일 기반 소스 외에도 Spark의 Kafka와의 강력한 기본 통합은 물론 여러 가지 개별적인 통합도 있습니다.

<sup>1</sup> 워터마크를 더 자세히 살펴보려면 Spark의 "이벤트 시간 및 상태 저장 처리" 섹션을 참조하세요. 확실한 가이드.

<sup>2</sup> 우리는 많은 독자들이 Apache Spark에 더 익숙하다는 것을 알고 있습니다. Apache Flink에 보다 구체적인 개념을 소개하려면 [Learn Flink](#)를 제안합니다. 문서의 페이지.

<sup>3</sup> Apache Spark 소스 및 싱크 문서는 일반적 [으로](#) Spark를 사용한 모든 스트리밍에 대한 소스로 간주되는 "구조적 스트리밍 프로그래밍 가이드"에서 찾을 수 있습니다: <https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>

Azure Event Hubs 와 같은 유지 관리 커넥터 라이브러리 , Google 게시/구독 라이트 , 그리고 아파치 펄서.

여전히 사용 가능한 출력 싱크가 여러 개 있지만 Delta Lake는 Spark를 사용하는 가장 큰 규모의 데이터 대상 중 하나입니다. 앞서 언급했듯이 Delta Lake는 기본적으로 쪽모이 세공 파일 형식의 제한으로 인해 대규모 스트림 수집 문제를 해결하도록 설계되었습니다. 부분적으로 Delta Lake의 기원과 Apache Spark의 오랜 역사를 통해 여기에서 다른 세부 사항의 대부분은 Spark 중심이지만 설명된 많은 개념에는 다른 프레임워크와도 결과가 있다는 점에 유의해야 합니다.

## 델타-rs

Rust 생태계에는 **delta-rs**라는 구현 덕분에 추가 처리 엔진과 자체 라이브러리도 있습니다. Delta Lake에서 실행 할 수 있는 추가 처리 옵션이 제공됩니다. 이 지역은 새로운 지역 중 하나이며 최근 몇 년 동안 집중적인 증축이 이루어졌습니다. **극지방 및 데이터퓨전** 이는 스트리밍 데이터 처리에 사용할 수 있는 몇 가지 추가 방법일 뿐이며 이 두 가지 모두 델타와 합리적으로 잘 결합됩니다. 이는 빠르게 발전하고 있는 분야로 앞으로 더욱 많은 성장이 기대됩니다.

Delta-rs 구현의 또 다른 이점은 데이터 스트림 처리 작업에 대한 추가 가능성을 열어주는 직접적인 Python 통합이 있다는 것입니다. 이는 소규모 작업의 경우 불필요한 오버헤드를 유발하는 상호 작용을 위해 대규모 프레임워크가 필요한 서비스에 Python API(예: AWS boto3)를 사용할 수 있음을 의미합니다. 스트리밍 작업을 보다 자연스럽게 지원하는 프레임워크의 일부 기능을 활용하지 못할 수도 있지만 인프라 요구 사항이 크게 감소하는 이점을 누리면서도 여전히 매우 빠른 성능을 얻을 수 있습니다.

Delta-rs 구현의 최종 결과는 Delta Lake가 추가 RDBMS에 의존하지 않고도 여러 처리 프레임워크와 엔진을 동시에 사용하고 더 많은 Java 중심 스택 외부에서 계속 작동할 수 있는 형식을 제공한다는 것입니다. 이는 서로 다른 시스템에서 작업하더라도 Delta Lake를 통해 얻을 수 있는 기본 제공 이점을 희생하지 않고도 자신 있게 데이터 애플리케이션을 구축할 수 있음을 의미합니다.

## 소스로서의 델타

Delta Lake 디자인의 원래 의도 중 대부분은 이전에 실제로 누락되었던 기능과 안정성을 추가하는 스트리밍 싱크였습니다. 특히 Delta Lake는 소규모 트랜잭션과 파일이 많은 경향이 있는 프로세스의 유지 관리를 단순화하고 ACID 트랜잭션 보장을 제공합니다. 하지만 그 측면을 더 자세히 살펴보기 전에 Delta Lake를 스트리밍 스스로 생각해 보겠습니다. 우리가 거래에서 본 이미 증분적인 성격을 통해

log에는 ID 값이 잘 정렬된 json 파일의 간단한 소스가 있습니다. 이는 모든 엔진이 추가 작업 중에 추가된 파일의 전체 트랜잭션 레코드와 함께 스트리밍 메시지의 오프셋으로 파일 ID 값을 사용하고 어떤 새 파일이 있는지 확인할 수 있음을 의미합니다. 트랜잭션 로그에 플러그인 dataChange를 포함하면 새 파일을 생성하지만 다운스트림 소비자에게 보낼 필요가 없는 압축 또는 기타 테이블 유지 관리 이벤트를 분리하는 데 도움이 됩니다. ID가 단조롭기 때문에 오프셋 추적이 더 간단해지기 때문에 다운스트림 소비자에 대해 정확히 한번 의미론이 여전히 가능합니다.

이 모든 것의 실질적인 장점은 Spark Structured Streaming을 사용하면 `readStream` 형식을 "델타"로 정의할 수 있으며 대상 테이블 또는 파일에서 이전에 사용 가능한 모든 데이터를 처리한 다음 추가될 때 증분 업데이트를 추가한다는 것입니다. 이를 통해 이전에 보았고 나중에 더 자세히 논의할 메달리온 아키텍처와 같은 많은 처리 아키텍처를 크게 단순화할 수 있지만, 지금은 추가 데이터 정제 레이어를 만드는 것이 간접성을 크게 줄이면서 자연스러운 작업이 된다고 가정해야 합니다.

Spark를 사용하면 `readStream` 자체가 작업 모드를 정의하고 "델타"는 형식일 뿐이며 작업은 백그라운드에서 발생하는 많은 작업과 함께 평소와 같이 진행됩니다. Flink에서는 접근 방식이 다소 바뀌었습니다. 대신 데이터 스트림 클래스의 델타 소스 개체를 구축한 다음 `forContinuousRowData API`를 사용하여 증분 처리를 시작합니다.

```
## Python
StreamingDeltaDf =
    ( Spark.readStream.format("delta").option("ignoreDeletes",
        "true").load("/files/delta/user_events") )
```

### 델타를 싱크로 사용

스트리밍 싱크에 필요한 많은 기능(비동기 압축 작업 등)은 현대적인 대용량 스트리밍 수집을 지원할 수 있는 방식으로 사용할 수 없거나 확장할 수 없었습니다. 사용자 활동과 장치의 가용성과 연결성 증가, 사물인터넷(IoT)의 급속한 성장으로 인해 대규모 스트리밍 데이터 소스의 성장이 빠르게 가속화되었습니다. 가장 중요한 문제 중 하나는 "어떻게 하면 모든 데이터를 효율적이고 안정적으로 캡처할 수 있습니까?"라는 질문에 대답하는 것입니다.

Delta Lake의 많은 기능은 특히 이 문제에 답하기 위해 제공됩니다.

예를 들어 작업이 트랜잭션 로그에 커밋되는 방식은 소스에 대한 스트리밍 진행 상황을 추적하고 완료된 트랜잭션만 확인하는 스트리밍 처리 엔진의 맥락에 자연스럽게 들어맞습니다.

손상된 파일이 없는 동안 로그에 커밋하면 안정성을 보장하면서 실제로 모든 소스 데이터를 캡처하고 있는지 확인할 수 있습니다. 생성되어 델타 로그에 내보내진 측정항목은 각 트랜잭션 중에 추가된 행 및 파일 수를 사용하여 스트림 프로세스의 일관성(또는 변동성)을 분석하는 데 도움이 됩니다.

대부분의 대규모 스트림 처리는 "마이크로 배치"에서 발생합니다. 이는 본질적으로 유사한 대규모 배치 프로세스의 소규모 트랜잭션입니다. 그 결과 스트림 처리 엔진이 전송 중인 데이터를 캡처할 때 많은 쓰기 작업이 발생하는 것을 볼 수 있습니다. 이러한 처리가 "상시 실행" 스트리밍 프로세스에서 발생하면 유지 관리 작업 실행, 채우기 또는 기록 데이터 수정과 같은 데이터 상태계의 다른 측면을 관리하기가 어려워질 수 있습니다.

최적화 와 같은 테이블 유ти리티 명령 과 환경의 여러 프로세스에서 델타 로그와 상호 작용하는 기능은 한편으로는 이 중 많은 부분이 사전에 고려되었으며 증분 특성으로 인해 이러한 프로세스를 보다 쉽게 중단할 수 있음을 의미합니다. 예측 가능한 방법. 반면에 우리는 이러한 작업의 어떤 종류의 조합이 때때로 우리가 피하고 싶은 충돌을 일으킬 수 있는지에 대해 좀 더 자주 생각해야 할 수도 있습니다. 자세한 내용은 7장의 동시성 제어 섹션을 참조하세요.

특히 11장에서 자세히 다룬 Delta Lake와 Apache Spark를 사용한 메달리온 아키텍처는 Delta Lake를 스트리밍 싱크와 동시에 작동하는 스트리밍 소스로 보는 중간 지점이 됩니다.

이는 실제로 많은 경우 추가 인프라의 필요성을 제거하고 전체 아키텍처를 단순화하는 동시에 깨끗한 데이터 엔지니어링 관행을 유지하면서 낮은 대기 시간, 높은 처리량의 스트림 처리를 위한 메커니즘을 제공합니다.

ces.

스트리밍 DataFrame 개체를 Delta Lake에 작성하는 것은 간단하며 writeStream 메서드 를 통해 형식 사양 과 디렉터리 위치만 필요합니다 .

```
## Python (스
트리밍DeltaDf
.writeStream

.format("delta") .outputMode("append") .start("<delta_path>/" )
```

마찬가지로 readStream 정의(비슷한 형식)를 writeStream 정의와 함께 연결하여 전체 입력-변환-출력 흐름을 설정할 수 있습니다(여기서는 간결성을 위해 변환 코드가 생략됨).

```
## 파이썬 (스파
크 .readStream
```

```

.format("델타") .load("/")
files/delta/user_events")
...
<기타 변환 논리>
...
.writingStream .format("delta") .outputMode("append") .start("/<delta_path>/"))

```

## 델타 스트리밍 옵션

이제 Delta Lake에서 스트리밍이 개념적으로 어떻게 작동하는지 논의했으므로 실제로 사용할 옵션의 좀 더 기술적인 측면과 원하는 인스턴스에 대한 약간의 배경 지식을 살펴보겠습니다. 수정하세요. 먼저 입력 속도를 제한할 수 있는 방법, 특히 이를 Apache Spark에서 제공하는 일부 기능과 함께 활용하는 방법을 살펴보겠습니다. 그런 다음 일부 트랜잭션을 건너뛰고 싶을 수 있는 몇 가지 경우를 살펴보겠습니다. 마지막으로 시간과 처리 작업 간의 관계에 대한 몇 가지 측면을 고려하여 후속 조치를 취하겠습니다.

입력 속도 제한 스트리밍 처리에 관해 이

야기할 때 일반적으로 정확성, 대기 시간, 비용이라는 세 가지 요소 사이의 균형을 찾아야 합니다. 우리는 일반적으로 정확성 측면에서 어떤 것도 포기하고 싶지 않으므로 이는 일반적으로 대기 시간과 비용 간의 균형으로 귀결됩니다. 즉, 더 높은 비용을 수용하고 리소스를 확장하여 가능한 한 빨리 데이터를 처리하거나 데이터 처리에 있어 크기를 제한하고 더 긴 처리 시간을 허용할 수 있습니다. 종종 이는 스트리밍 처리 엔진의 제어하에 있지만 Delta Lake에는 마이크로 배치의 크기를 추가로 제어할 수 있는 두 가지 추가 옵션이 있습니다.

maxFilesPerTrigger 이는

모든 마이크로 배치에서 고려되는 새 파일 수의 제한을 설정합니다. 기본값은 1000입니다.

maxBytesPerTrigger 각 마

이크로 배치에서 처리되는 데이터 양에 대한 대략적인 제한을 설정합니다.

이 옵션은 "소프트 최대"를 설정합니다. 즉, 마이크로 배치는 대량의 데이터를 처리하지만 가장 작은 입력 단위가 이 제한보다 클 때 더 많은 데이터를 처리할 수 있음을 의미합니다. 즉, 이 크기 설정은 하나의 파일이든 여러 파일이든 초과해야 하는 임계값과 유사하게 작동하지만 이 임계값을 초과하는 데 필요한 파일 수는 그만큼 많은 파일을 사용하게 됩니다.

대략적인 크기를 사용하는 마이크로 배치의 파일 수에 대한 동적 설정과 같습니다.

이 두 가지 설정은 [트리거를](#) 사용하여 균형을 맞출 수 있습니다. 구조적 스트리밍에서는 각 마이크로배치에서 처리되는 데이터의 양을 늘리거나 줄입니다.

예를 들어 이러한 설정을 사용하여 처리에 필요한 컴퓨팅 크기를 낮추거나 작업할 예상 파일 크기에 맞게 작업을 조정할 수 있습니다.

스트리밍에 Trigger.Once를 사용하는 경우 이 옵션은 무시됩니다. 이는 기본적으로 설정되어 있지 않습니다. 실제로 동일한 스트리밍 쿼리에 maxBytesPerTrigger와 maxFilesPerTrigger를 모두 사용할 수 있습니다. 그러면 마이크로 배치는 어느 한도에 도달할 때까지 실행됩니다.



여기서는 정리가 발생하는 경우 오래된 트랜잭션을 건너뛸 수 있는 방식으로 더 긴 트리거 또는 작업 예약 간격으로 더 짧은 logRetention - Duration을 설정할 수 있다는 점에 주목하고 싶습니다. 사전 처리는 무엇이 있는지 알 수 없기 때문에 로그에서 사용 가능한 가장 빠른 트랜잭션에서 시작됩니다. 이는 처리에서 데이터를 건너뛸 수 있음을 의미합니다. 이런 일이 발생할 수 있는 간단한 예는 logRetentionDuration이 하루 또는 이틀로 설정되어 있지만 충분 변경 사항을 선택하려는 처리 작업이 매우 실행되는 경우입니다. 중간 기간 동안의 진공 작업으로 인해 파일의 이전 버전 중 일부가 제거되므로 해당 변경 사항이 다음 버전으로 전파되지 않습니다.

달리다.

### 업데이트 또는 삭제 무시 지금까지

Delta Lake를 사용한 스트리밍에 대해 이야기하면서 실제로 논의해야 할 사항은 아니지만 실제로 논의한 내용이 있습니다. 이전 장에서는 Delta Lake의 일부 기능이 CRUD 작업 수행의 용이성을 어떻게 향상시키는지, 특히 업데이트 및 삭제 작업을 살펴보았습니다. 여기서 언급해야 할 점은 기본적으로 Delta Lake에서 스트리밍할 때 추가 전용 유형의 소스에서 스트리밍한다고 가정한다는 것입니다. 즉, 발생하는 충분 변경 사항은 새 파일의 추가일 뿐입니다. 그러면 "스트림 소스에서 업데이트 또는 삭제 작업이 있으면 어떻게 되나요?"라는 질문이 생깁니다.

간단히 말해서 최소한 기본 설정에서는 Spark readStream 작업이 실패합니다. 이는 스트림 소스로서 새 파일만 수신할 것으로 예상하고 변경 또는 삭제로 인해 발생하는 파일을 처리하는 방법을 지정해야 하기 때문입니다. 이는 일반적으로 대규모 수집 테이블이나 CDC(변경 데이터 캡처) 레코드 수신에 적합합니다. 일반적으로 다른 유형의 작업이 적용되지 않기 때문입니다. 이러한 상황을 처리할 수 있는 방법에는 두 가지가 있습니다. 어려운 방법은 출력과 체크포인트를 삭제하고 스트림을 처음부터 다시 시작하는 것입니다. 더 쉬운 방법은 ignoreDeletes 또는 ignoreChanges 옵션을 활용하는 것입니다. 이 두 옵션은 다소 다릅니다.

이름의 유사성에도 불구하고 동작. 가장 큰 주의 사항은 두 설정 중 하나를 사용할 때 곧 설명하겠지만 다운스트림을 수동으로 추적하고 변경해야 한다는 것입니다.

#### ignoreDeletes 설정

ignoreDeletes 설정은 말 그대로 작동합니다. 즉, 새 파일이 생성되지 않은 경우 삭제 작업을 무시합니다. 이것이 중요한 이유는 업스트림 파일을 삭제하면 해당 변경 사항이 다운스트림 대상으로 전파되지 않지만 이 설정을 사용하여 스트림 처리 작업의 실패를 방지하고 예를 들어 일반과 같은 중요한 삭제 작업을 계속 지원할 수 있기 때문입니다. 개인 사용자 데이터를 삭제해야 하는 경우 GDPR(데이터 보호 규정) 준수를 잊어버릴 권리가 있습니다. 중요한 점은 삭제 작업을 위해 필터링한 것과 동일한 값으로 데이터를 분할해야 새 파일을 생성하는 잔여 항목이 없다는 것입니다. 이는 잠재적으로 여러 테이블에서 동일한 삭제 작업을 실행해야 하지만 스트림 프로세스에서 이러한 작은 삭제 작업을 무시하고 다운스트림 삭제 작업을 별도의 프로세스에 남겨두고 정상적으로 계속할 수 있음을 의미합니다.

#### ignoreChanges 설정

ignoreChanges 설정은 실제로 ignoreDeletes 와 약간 다르게 동작합니다. 파일을 제거하기만 하는 작업을 건너뛰는 대신, ignoreChanges는 변경으로 인해 발생하는 새 파일이 마치 새 파일인 것처럼 허용합니다. 즉, 특정 파일 내의 일부 레코드를 업데이트하거나 파일의 새 버전이 생성되도록 파일에서 몇 개의 레코드만 삭제하면 파일의 새 버전이 전파될 때 이제 새 파일로 해석됩니다. - 개울. 이는 최신 버전의 데이터를 사용하는 데 도움이 되지만, 데이터 중복을 방지하려면 이것이 미치는 영향을 이해하는 것이 중요합니다.

이러한 경우에 필요한 것은 병합 논리를 통해 또는 추가 시간 유지 정보(즉, `version_as_of` 타임스탬프 또는 이와 유사한 버전 추가)를 포함하여 데이터를 차별화하여 중복 레코드를 처리할 수 있는지 확인하는 것입니다.

다양한 유형의 변경 작업에서 대부분의 레코드가 변경 없이 재처리되므로 병합이나 중복 제거가 일반적으로 선호되는 경로라는 것을 확인했습니다.

#### 예 예를 생

각해 보겠습니다. `date`, `user_email` 및 `action` 열이 있는 `user_events`라는 Delta Lake 테이블이 있고 날짜 열로 분할되어 있다고 가정해 보겠습니다.

또한 `user_events` 테이블을 대규모 파이프라인 프로세스의 한 단계에 대한 스트리밍 소스로 사용하고 있으며 GDPR 관련 요청으로 인해 해당 테이블에서 데이터를 삭제해야 한다고 가정해 보겠습니다.

파티션 경계에서 삭제하면(즉, 쿼리의 WHERE 절이 파티션 열의 데이터를 필터링함) 파일이 이미 해당 값을 기반으로 하는 디렉토리에 있으므로 삭제 시 테이블 메타데이터에서 해당 파일이 삭제됩니다.

따라서 특정 날짜에 맞춰 일부 전체 파티션에서 데이터를 삭제하려는 경우 다음과 같이 읽기 스트림에 ignoreDeletes 옵션을 추가할 수 있습니다.

```
## Python
StreamingDeltaDf =

        ( Spark.readStream.format("delta").option("ignoreDeletes",
        "true").load("/files/delta/user_events") )
```

대신 user\_email과 같은 파티션이 아닌 열을 기반으로 데이터를 삭제하려면 대신 다음과 같이 ignoreChanges 옵션을 사용해야 합니다.

```
## Python 스트리밍
StreamingDeltaDf = ( Spark

        .readStream.format("delta").option("ignoreChanges",
        "true").load("/files/delta/user_events") )
```

비슷한 방식으로 user\_email과 같은 파티션이 아닌 열에 대해 레코드를 업데이트하면 변경된 레코드와 변경되지 않은 원본 파일의 다른 레코드를 포함하는 새 파일이 생성됩니다. ignoreChanges가 설정되면 이 파일은 readStream 쿼리에 의해 표시되므로 이 프로세스의 출력에 중복 데이터가 들어가는 것을 방지하려면 이 스트림에 대한 추가 논리를 포함해야 합니다.

### 초기 처리 위치 Delta Lake 소스를

사용하여 스트리밍 프로세스를 시작할 때 기본 동작은 테이블의 가장 초기 버전으로 시작한 다음 가장 최신 버전까지 점진적으로 처리하는 것입니다. 물론 스트리밍 프로세스에 대한 체크포인트를 삭제하고 중간 지점 또는 가장 최근 지점에서 다시 시작해야 하는 경우와 같이 실제로 가장 초기 버전으로 시작하고 싶지 않은 경우도 있을 것입니다. 사용 가능. 트랜잭션 로그 덕분에 실제로 이 시작점을 지정하여 체크포인트를 통해 스트림이 특정 지점에서 복구되는 방식과 유사하게 로그 시작 부분부터 모든 것을 다시 처리할 필요가 없도록 할 수 있습니다.

여기서 할 수 있는 일은 처리를 시작할 초기 위치를 정의하는 것이며 두 가지 방법 중 하나로 이를 수행할 수 있습니다.<sup>4</sup> 첫 번째는 처리를 시작하려는 특정 버전을 지정하는 것이고 두 번째는 처리를 시작하려는 시간을 지정하는 것입니다. 처리를 시작하고 싶습니다. 이러한 옵션은 StartingVersion 및 StartingTimestamp를 통해 사용할 수 있습니다.

StartingVersion을 지정하면 예상할 수 있는 것과 거의 같은 작업이 수행됩니다. 트랜잭션 로그의 특정 버전이 주어지면 해당 버전에 커밋된 파일이 처리를 시작하는 첫 번째 데이터가 되며 계속해서 처리됩니다. 이러한 방식으로 이 버전(포함)부터 시작하는 모든 테이블 변경 사항은 스트리밍 소스에서 읽혀집니다. 트랜잭션 로그에서 버전 매개변수를 검토하여 필요한 특정 버전을 식별하거나 "최신"을 지정하여 최신 변경 사항만 가져올 수도 있습니다.



Apache Spark를 사용할 때 SQL 컨텍스트에서 DESCRIBE HISTORY 명령 출력의 버전 열에서 커밋 버전을 확인하면 가장 쉽습니다.

마찬가지로 좀 더 시간적인 접근 방식을 위해 StartingTimestamp 옵션을 지정할 수 있습니다.

타임스탬프 옵션을 사용하면 실제로 약간씩 다른 몇 가지 동작을 얻을 수 있습니다.

주어진 타임스탬프가 커밋과 정확히 일치하는 경우 처리를 위해 해당 파일이 포함됩니다. 그렇지 않으면 해당 시점 이후에 발생하는 버전의 파일만 처리하는 동작입니다. 여기서 특히 유용한 기능 중 하나는 완전히 형식화된 타임스탬프 문자열을 엄격하게 요구하지 않고 해석할 수 있는 유사한 날짜 문자열을 사용할 수도 있다는 것입니다. 이는 startTimestamp 매개변수가 다음 중 하나와 같아야 함을 의미합니다.

- 타임스탬프 문자열(예: “2023-03-23T00:00:00.000Z”)
- 날짜 문자열(예: “2023-03-23”).

다른 설정과 달리 여기서는 두 옵션을 동시에 사용할 수 없습니다.

둘 중 하나를 선택해야 합니다. 체크포인트가 이미 정의된 기존 스트리밍 쿼리에 이 설정을 추가하면 새 쿼리를 시작할 때만 적용되므로 둘 다 무시됩니다.

주목해야 할 또 다른 사항은 이러한 옵션을 사용하여 소스의 지정된 위치에서 시작할 수 있더라도 스키마는 사용 가능한 최신 버전을 반영한다는 것입니다. 이는 잘못된 값이 있거나 오류가 발생할 수 있음을 의미합니다.

지정된 시작점과 현재 사이의 호환되지 않는 스키마 변경 버전.

---

<sup>4</sup> <https://docs.delta.io/latest/delta-streaming.html#specify-initial-position>

user\_events 데이터세트를 다시 생각해 보면 버전 5 이후에 발생한 변경 사항을 읽고 싶다고 가정해 보겠습니다. 그런 다음 다음과 같이 작성합니다.

```
## Python
```

```
(spark.readStream.format("delta").option("startingVersion",
"5").load("/files/delta/user_events"))
```

또는 날짜를 기준으로 변경 사항을 읽으려면( 2023-04-18 이후 발생) 대신 다음과 같은 것을 사용하세요.

```
## Python
```

```
(spark.readStream.format("delta").option("startingTimestamp",
"2023-04-18").load("/files/delta/user_events"))
```

EventTimeOrder를 사용한 초기 스냅샷 Delta Lake를

스트리밍 소스로 사용할 때의 기본 순서는 파일 수정 날짜를 기준으로 합니다. 또한 쿼리를 처음 실행할 때 테이블의 현재 상태에 도달할 때까지 쿼리가 자연스럽게 실행된다는 것도 확인했습니다. 우리는 이 버전의 테이블을 호출합니다. 이 버전은 시작점부터 현재 상태까지, 스트리밍 쿼리 시작 부분의 초기 스냅샷을 포함합니다. Databricks에는 이 초기 스냅샷의 시간을 해석하기 위한 추가 옵션이 있습니다.

데이터 세트의 경우 수정 시간을 기반으로 한 기본 순서가 올바른지 또는 데이터 순서를 단순화할 수 있는 데이터 세트에서 활용할 수 있는 이벤트 시간 필드가 있는지 고려할 수 있습니다.

레코드가 마지막으로 수정된(표시된) 시간과 관련된 타임스탬프는 반드시 이벤트가 발생한 시간과 일치하지 않습니다. 다양한 간격으로 버스트적으로 전달되는 IoT 장치 데이터를 생각해 볼 수 있습니다. 즉, last\_modified 타임스탬프 열이나 그와 유사한 열을 사용하는 경우 레코드가 순서대로 처리될 수 있으며 이로 인해 워터마크에 의해 레코드가 지연 이벤트로 삭제될 수 있습니다. 수정 시간보다 이벤트 시간을 선호하는 EventTimeOrder 옵션을 활성화하면 이 데이터 삭제 문제를 방지할 수 있습니다. 이것은 event\_time 열에 연관된 워터마크 옵션을 사용하여 readStream에 옵션을 설정하는 예입니다.

```
## Python
```

```
(spark.readStream.format("delta").option("withEventTimeOrder", "true"))
```

```
.load("/files/delta/  
user_events") .withWatermark("event_time", "10  
초"))
```

옵션이 활성화되면 초기 스냅샷을 분석하여 전체 시간 범위를 얻은 다음 버킷으로 나누어 각 버킷이 차례로 마이크로배치로 처리되어 일부 셜플 작업이 추가될 수 있습니다. maxFilesPerTrigger 또는 maxBytesPerTrigger 옵션을 사용하여 처리 속도를 조절할 수 있습니다.

이와 관련하여 귀하가 알고 있어야 할 몇 가지 설명이 있습니다.

상황:

- 데이터 삭제 문제는 Stateful의 초기 델타 스냅샷이 생성될 때만 발생합니다.  
스트리밍 쿼리는 기본 순서로 처리됩니다.
- withEventTimeOrder는 스트리밍 쿼리 시작 시에만 적용되는 또 다른 설정이므로 쿼리가 시작되고 초기 스냅샷이 계속 처리되는 후에는 변경할 수 없습니다. withEventTimeOrder 설정을 수정하려면 체크포인트를 삭제하고 초기 처리 위치 옵션을 활용하여 진행해야 합니다.
- 이 옵션은 Delta Lake 1.2.1에서 사용할 수 있게 되었습니다. withEventTimeOrder가 활성화된 상태에서 스트림 쿼리를 실행하는 경우 초기 스냅샷 처리가 완료될 때까지 이 기능을 지원하지 않는 버전으로 다운그레이드할 수 없습니다. 버전을 다운그레이드해야 하는 경우 초기 스냅샷이 완료될 때까지 기다리거나 체크포인트를 삭제하고 쿼리를 다시 시작할 수 있습니다.
- withEventTimeOrder를 사용할 수 없는 몇 가지 드문 시나리오가 있습니다 .
  - 이벤트 시간 열이 생성된 열이고 델타 소스와 워터마크 사이에 비투영 변환이 있는 경우.
  - 스트림 쿼리에 여러 델타 소스가 포함된 워터마크가 있습니다. • 셜플 작업이 증가할 가능성이 있으므로 초기 스냅샷 처리 성능이 영향을 받을 수 있습니다.

이벤트 시간 순서를 사용하면 초기 스냅샷 스캔이 트리거되어 각 마이크로 배치에 해당하는 이벤트 시간 범위를 찾습니다. 이는 더 나은 성능을 위해 이벤트 시간 열이 통계를 수집하는 열 중 하나인지 확인하고 싶다는 것을 의미합니다. 이렇게 하면 쿼리에서 데이터 건너뛰기를 활용할 수 있고 더 빠른 필터 작업을 수행할 수 있습니다. 이벤트 시간 열을 기준으로 데이터를 분할하는 것이 적합한 경우 처리 성능을 높일 수 있습니다.

성능 지표는 각 마이크로 배치에서 참조되는 파일 수를 나타내야 합니다.



Spark.databricks.delta.withEventTimeOrder.enabled true를 설정하면 클러스터 수준 Spark 구성으로 설정할 수 있지만 이렇게 하면 클러스터에서 실행되는 모든 스트리밍 쿼리에 적용됩니다.

## Apache Spark를 사용한 고급 사용법

지금까지 다룬 많은 기능은 앞서 나열된 프레임워크 중 하나 이상에서 적용될 수 있습니다. 여기서는 특별히 Apache Spark를 사용하는 동안 직면했던 몇 가지 일반적인 사례에 주목합니다. 이는 프레임워크의 기능을 활용하면 Delta Lake에 내장된 일부 기능을 직접 사용하지 못할 수 있는 경우입니다.

### 멱등성 스트림 쓰기 이전 논의의 대부분

은 단일 소스에서 단일 대상으로 처리 작업을 실행한다는 아이디어에 중점을 두었습니다. 그러나 현실 세계에서는 이와 같이 깔끔하고 단순한 파이프라인이 항상 존재하는 것은 아니며, 대신 여러 소스를 사용하여 여러 대상에 쓰는 파이프라인을 구축하여 결국 중복될 수도 있습니다. 트랜잭션 로그 및 원자성 커밋 동작을 통해 우리는 이미 고려한 기능적 관점에서 단일 Delta Lake 대상에 대한 여러 작성자를 지원할 수 있습니다. 그런데 이것을 스트림 처리 파이프라인에 어떻게 적용할 수 있을까요?

Apache Spark에는 구조화된 스트리밍 DataFrame에서 사용할 수 있는 `foreachBatch` 메서드가 있습니다. 이를 통해 각 스트림 마이크로 배치에 대해 보다 맞춤화된 논리를 정의할 수 있습니다. 이는 단일 스트림 소스를 여러 대상에 쓰는 것을 지원하는 데 사용하는 일반적인 방법입니다. 그러면 우리가 직면하는 문제는 예를 들어 두 개의 서로 다른 대상이 있고 두 번째 대상에 대한 쓰기에 트랜잭션이 실패하는 경우 각 대상의 처리 상태가 동기화되지 않는 다소 문제가 있는 시나리오가 있다는 것입니다. 보다 구체적으로 말하면 첫 번째 쓰기가 완료되고 두 번째 쓰기가 실패했기 때문에 스트림 처리 작업이 다시 시작되면 성공적으로 완료되지 않았으므로 마지막 실행과 동일한 오프셋을 고려합니다.

`sourceDf DataFrame`이 있고 이를 두 개의 서로 다른 대상으로 일괄 처리하려는 이 예를 생각해 보세요. 입력 DataFrame을 사용하고 일반적인 Spark 작업을 사용하여 각 마이크로배치를 작성하는 함수를 정의합니다.

그런 다음 `writeStream` 메서드에서 사용할 수 있는 `foreachBatch` 메서드를 사용하여 해당 함수를 적용할 수 있습니다.

```
## 파이썬 소스
스스로 Df = ...
... # 스트리밍 소스 DataFrame

# 두 대상에 쓰는 함수 정의 def writeToDeltaLakeTables(batch_df):
```

```

# 위치 1 (batch_df

쓰다

.format("delta") .save("/")
<delta_path_1>/") # 위치 2

(batch_df .write .format("delta") .save("/")
<delta_path_2>/") )

# 'foreachBatch'를 사용하여 마이크로 배치에 대해 함수를 적용합니다. (sourceDf .writeStream .format("delta") .queryName("Unclear
status

stream") .foreachBatch(writeToDeltaLakeTables) .start() )

```

이제 첫 번째 위치에 쓴 후 두 번째 위치가 완료되기 전에 오류가 발생한다고 가정합니다. 트랜잭션이 실패했기 때문에 두 번째 테이블에는 로그에 커밋된 내용이 없다는 것을 알 수 있지만 첫 번째 테이블에서는 트랜잭션이 성공했습니다. 작업을 다시 시작하면 동일한 지점에서 시작되고 해당 마이크로배치에 대한 전체 기능이 다시 실행되어 중복된 데이터가 첫 번째 테이블에 기록될 수 있습니다.

다행히 Delta Lake에는 보다 세부적인 트랜잭션 추적을 지정할 수 있도록 하여 이 경우 도움이 될 수 있는 기능이 있습니다.

#### 멱등성 쓰기 스트리밍 소스

스스로 foreachBatch를 활용하고 단 두 개의 대상에만 쓰고 있다고 가정해 보겠습니다. 우리가 하고 싶은 것은 foreachBatch 트랜잭션의 구조를 취하고 이를 멋진 Delta Lake 기능과 결합하여 일부 테이블에서 중복 트랜잭션을 발생시키지 않고 모든 테이블에 걸쳐 마이크로 배치 트랜잭션을 커밋하도록 하는 것입니다( 즉, 테이블에 멱등성 쓰기를 원합니다.) 이 상태에 도달하는 데 사용할 수 있는 두 가지 옵션이 있습니다.

#### txnid 이

는 고유한 문자열 식별자여야 하며 각 DataFrame 쓰기 작업에 대해 전달할 수 있는 애플리케이션 ID 역할을 합니다. 이는 각 쓰기의 소스를 식별합니다. 스트리밍 쿼리 ID 또는 원하는 다른 의미 있는 이름을 txnAppId로 사용할 수 있습니다.

#### txnb전

이는 트랜잭션 버전 역할을 하며 기능적으로 writeStream 쿼리에 대한 오프셋 식별자가 되는 단조롭게 증가하는 숫자입니다.

이러한 옵션을 모두 포함함으로써 여러 대상에 쓰는 `foreachBatch` 작업 내에서도 쓰기 수준에서 고유한 소스 및 오프셋 추적을 생성합니다.

이를 통해 무시할 수 있는 중복 쓰기 시도를 테이블 수준에서 감지할 수 있습니다. 즉, 여러 테이블 대상 중 하나만 처리하는 동안 쓰기가 중단된 경우 트랜잭션이 이미 성공한 테이블에 쓰기 작업을 복제하지 않고도 처리를 계속할 수 있습니다. 스트림이 체크포인트에서 다시 시작되면 동일한 마이크로 배치로 다시 시작되지만 `foreachBatch`에서는 쓰기 작업이 이제 테이블 수준의 세분화에서 확인되므로 완료할 수 없는 테이블에만れます. 이전에는 동일한 `txnAppId` 및 `txnVersion` 식별자를 갖게 되므로 성공적이었습니다.



애플리케이션 ID (`txnAppId`)는 사용자가 생성한 고유 문자열일 수 있으며 스트림 ID와 관련될 필요가 없으므로 이를 사용하여 작업을 수행하는 애플리케이션을 보다 기능적으로 설명하거나 데이터 소스를 식별할 수 있습니다. 동일한 `DataStreamWriter` 옵션을 실제로 사용하여 일괄 처리에서도 유사한 면밀성 쓰기를 달성할 수 있습니다.



소스에서 처리를 다시 시작하고 스트리밍 체크포인트를 삭제/다시 생성하려는 경우 쿼리를 다시 시작하기 전에 새 `appId`도 제공해야 합니다. 그렇지 않으면 다시 시작된 쿼리의 모든 쓰기가 동일한 `txnAppId`를 포함하고 배치 ID 값이 다시 시작되어 대상 테이블에 중복 트랜잭션으로 표시되므로 무시됩니다.

이러한 옵션을 사용하여 면밀성이 있는 여러 위치에 쓰도록 이전 예제의 함수를 업데이트하려면 다음과 같이 각 대상에 대한 옵션을 지정할 수 있습니다.

```
## 파이썬
app_id = ... # 애플리케이션 ID로 사용되는 고유 문자열입니다.

def writeToDeltaLakeIdempotent(batch_df, 배치_id):
    # 위치 1

    (batch_df .write .format("delta") .option("txnVersion",
        배치_id) .option("txnAppId", app_id) .save("/")
        <delta_path>"/") # 위치 2

    (batch_df .format("델
        타") 쓰기
```

```
.option("txnVersion", 배치
_id).option("txnApplId",
app_id).save("/<delta_path>/") )
```

## 병합

스트림 처리에 `foreachBatch`가 사용되는 경향이 있는 또 다른 일반적인 경우가 있습니다. 파이프라인을 통해 변경되지 않은 대량의 레코드를 재처리하도록 허용하거나 CDC 레코드 처리와 같은 고급 일치 및 변환 논리가 필요할 수 있는 위에서 본 몇 가지 제한 사항을 생각해 보세요. 값을 업데이트하려면 단순히 정보를 추가하는 것이 아니라 변경 사항을 기준 테이블에 병합해야 합니다. 나쁜 소식은 스트리밍 종류의 기본 동작에서는 `foreachBatch`를 활용하지 않는 한 추가 유형 동작을 사용해야 한다는 것입니다.

우리는 3장 앞부분에서 병합 작업을 살펴보았고 이를 통해 일치 기준을 사용하여 기존 레코드를 업데이트하거나 삭제하고 기준과 일치하지 않는 다른 레코드를 추가할 수 있다는 것을 확인했습니다. 즉, `upsert` 작업을 수행할 수 있습니다. `foreachBatch`를 사용하면 각 마이크로 배치를 일반 `DataFrame`처럼 처리할 수 있으므로 마이크로 배치 수준에서 실제로 Delta Lake를 사용하여 이러한 `upsert` 작업을 수행할 수 있습니다. MERGE SQL 작업이나 Scala, Java 및 Python에 대한 결과를 사용하여 소스 테이블, 뷰 또는 `DataFrame`의 데이터를 대상 델타 테이블로 `upsert`할 수 있습니다. 델타 레이크 API. 고급 사용 사례를 용이하게 하기 위해 SQL 표준을 넘어서는 확장된 구문도 지원합니다.

Delta Lake의 병합 작업에는 일반적으로 원본 데이터에 대한 두 번의 전달이 필요합니다.

소스 `DataFrame`에서 `current_timestamp` 또는 무작위와 같은 비결정적 함수를 사용하는 경우 소스 데이터에 대한 여러 전달이 행에 다른 값을 생성하여 잘못된 결과를 초래할 수 있습니다. 열에 대해 보다 구체적인 함수나 값을 사용하거나 결과를 중간 테이블에 기록하면 이를 방지할 수 있습니다. 캐시 무효화로 인해 소스 데이터가 부분적으로 또는 완전히 재처리되어 동일한 종류의 값 변경이 발생할 수 있기 때문에 소스 데이터를 캐싱하는 것이 도움이 될 수 있습니다 (예를 들어 클러스터가 축소될 때 실행기 중 일부가 손실되는 경우). 난수 생성을 기반으로 `DataFrame` 파티셔닝을 재구성하기 위해 슬트 열을 사용하는 것과 같은 작업을 시도할 때 이것이 놀라운 방식으로 실패할 수 있는 경우를 보았습니다(예: Spark는 임의의 접두사가 예상과 다르기 때문에 디스크에서 셜플 파티션을 찾을 수 없습니다). 재시도 실행. 병합 작업을 여러 번 수행하면 이러한 일이 발생할 가능성이 높아집니다.

일련의 고객에 대한 최신 일일 소매 거래 요약을 업데이트하기 위해 `foreachBatch`를 사용하여 스트림에서 병합 작업을 사용하는 예를 고려해 보겠습니다.

이 경우 고객 ID 값을 일치시키고 거래 날짜, 항목 수 및 달러 금액을 포함합니다. 실제로 여기서 `mergeBuilder` API를 사용하기 위해 수행하는 작업은 스트리밍 `DataFrame`에 대한 논리를 처리하는 함수를 구축하는 것입니다. 함수 내에서 대상 테이블에 대한 일치 기준으로 고객 ID를 제공합니다.

그리고 변경 소스를 확인한 다음 삭제 메커니즘을 허용하고 기존 고객을 업데이트하거나 새로운 고객을 추가합니다.5 함수의 작업 흐름은 일치 조건에 대한 인수를 사용하여 병합 할 항목과 일치하는 고객을 지정하는 것입니다. 레코드가 일치하는지 여부에 따라 수행하려는 작업(몇 가지 추가 조건을 추가할 수 있음)

```
##  
delta.tables 의 Python 가져오기 *  
  
def upsertToDelta(microBatchDf, 배치id):  
    Target_table = "retail_db.transactions_silver" deltaTable =  
        DeltaTable.forName(spark, target_table)  
  
    (deltaTable.alias("dt") .merge(source=microBatchDf.alias("sdf"),  
        Condition="sdf.t_id = dt.t_id  
    ") .whenMatchedDelete(condition="sdf.eration='DELETE'" ) .whenMatchedUpdate(set={ "t_id":  
        "sdf.t_id", "transaction_date":  
        "sdf.transaction_date",  
        "item_count": "sdf.item_count", "amount":  
        "sdf.amount" }) .whenNotMatchedInsert(values={ "t_id":  
        "sdf.t_id", "transaction_date":  
  
        "sdf.transaction_date", "item_count":  
        "sdf.item_count", "amount":  
        "sdf.amount" }).execute())
```

함수 본문 자체는 이미 일반 일괄 처리로 병합 논리를 지정하는 방법과 유사합니다. 이 경우 유일한 실제 차이점은 전체 소스를 한꺼번에 실행하는 것이 아니라 수신된 모든 배치에 대해 병합 작업을 실행한다는 것입니다. 이제 함수가 이미 정의되었으므로 변경 사항 스트림을 읽고 Spark의 foreachBatch를 사용하여 사용자 정의된 병합 논리를 적용한 후 다른 테이블에 다시 쓸 수 있습니다.

```
## 파이썬 변  
경 스트림 = ... # CDC 레코드로 DataFrame 스트리밍  
  
# 스트리밍 집계 쿼리의 출력을 델타 테이블에 씁니다 (changesStream .writeStream .format("delta") .queryName("Summaries  
Silver  
  
Pipeline") .foreachBatch(upsertToDelta) .outputMode("update")
```

---

5 forEachBatch에서 병합 사용에 대한 추가 세부 정보 및 예(예: SCD 유형 II 병합의 경우)는 <https://docs.delta.io/latest/delta-update.html#merge-examples>를 참조하세요 .

## [시작\(\)](#)

따라서 변경 스트림의 각 마이크로 배치에는 병합 논리가 적용되고 면등성 쓰기에 대한 예에서 했던 것처럼 대상 테이블이나 여러 테이블에 기록됩니다.

## Delta Lake 성능 메트릭

데이터 처리 파이프라인에서 흔히 간과되지만 매우 유용한 것 중 하나는 진행 중인 작업에 대한 통찰력입니다. 처리가 진행되는 속도와 규모를 이해하는 데 도움이 되는 지표가 있으면 비용 추정, 용량 계획 또는 문제 발생 시 문제 해결을 위한 귀중한 정보가 될 수 있습니다. Delta Lake로 스트리밍할 때 메트릭 정보를 수신하는 몇 가지 사례를 이미 확인했지만 여기서는 실제로 수신되는 내용을 더 주의 깊게 살펴보겠습니다.

### 측정항목

살펴본 것처럼 Delta Lake를 사용하여 처리하기 위해 시작 및 끝 경계 지점을 수동으로 설정하려는 경우가 있으며 이는 일반적으로 버전 또는 타임스탬프에 맞춰 정렬됩니다. 이러한 경계 내에서 서로 다른 수의 파일 등을 가질 수 있으며 특히 스트리밍 프로세스에서 중요한 개념 중 하나는 해당 파일을 통해 오프셋 또는 진행 상황을 추적하는 것입니다.

Spark Structured Streaming에 대해 보고된 측정항목에서 이러한 오프셋을 추적하는 몇 가지 세부 정보를 볼 수 있습니다.

Databricks에서 프로세스를 실행할 때도 배압을 추적하는 데 도움이 되는 몇 가지 추가 측정항목이 있습니다. 즉, 현재 시점에 수행해야 할 미해결 작업의 양이 얼마나 됩니까? 출력으로 표시되는 성능 지표는 numInputRows, inputRowsPerSecond 및 selectedRowsPerSecond입니다. 역압 측정항목은 numBytesOutstanding 및 numFilesOutstanding입니다. 이러한 측정 항목은 설계상 자체적으로 설명이 가능하므로 각 항목을 개별적으로 살펴보지는 않겠습니다.



inputRowsPerSecond 측정항목을 processedRowsPerSecond 측정항목과 비교하면 상대적 성능을 측정하는 데 사용할 수 있는 비율이 제공되며 작업에 더 많은 리소스를 할당해야 하는지 또는 트리거를 약간 줄여야 하는지 여부를 나타낼 수 있습니다.

### 맞춤 측정항목

Apache Flink와 Apache Spark 모두 애플리케이션에서 추적되는 지표 정보를 확장하는 데 사용할 수 있는 사용자 지정 지표 옵션도 있습니다. 이 개념을 사용하여 본 방법 중 하나는 추가 사용자 정의 측정항목 정보를 보내는 것이었습니다.

Spark의 `foreachBatch` 작업 내부에서. 이 옵션을 추구하는 데 필요한 각 처리 프레임워크에 대한 설명서를 참조하세요. 이는 최고 수준의 사용자 정의를 제공하지만 가장 많은 수작업을 필요로 합니다.

## 자동 로더 및 델타 라이브 테이블

우리의 초점은 Delta Lake 오픈 소스 프로젝트에서 무료로 사용할 수 있는 모든 것에 있지만, 언급 할 가치가 있는 Delta Lake에 의존하거나 자주 작동하는 Databricks에서만 사용할 수 있는 몇 가지 주요 주제가 있습니다. Delta Lake 및 Apache Spark의 창시자로서

### 오토로더

Databricks에는 **Auto Loader**로 알려진 다소 독특한 Spark 구조의 스트리밍 소스가 있습니다. 하지만 실제로는 `cloudFiles` 소스로 생각하는 것이 더 좋습니다. 전체적으로 `cloudFiles` 소스는 Databricks의 구조적 스트리밍의 스트리밍 소스 정의에 가깝지만 Delta Lake가 일반적으로 대상 싱크인 많은 조직에서 스트리밍을 위한 더 쉬운 진입점이 되었습니다. 이는 부분적으로 스트림 처리의 구성 요소인 오프셋 추적과 같은 일부 이점을 통합하기 위해 일괄 처리를 증분화하는 자연스러운 방법을 제공하기 때문입니다.

`cloudFiles` 소스에는 실제로 두 가지 다른 작업 방법이 있습니다. 하나는 저장소 위치에서 파일 목록 작업을 직접 실행하는 것이고, 다른 하나는 저장소 위치에 연결된 알림 대기열을 수신하는 것입니다. 어떤 방법을 사용하든 유ти리티는 진행 상황을 추적하는 데 사용하는 오프셋이 지정된 소스 디렉토리의 실제 파일 이름이므로 클라우드 저장소에서 파일을 정기적으로 수집하기 위한 확장 가능하고 효율적인 메커니즘이라는 점을 금방 알 수 있습니다. 가장 일반적인 사용법의 예는 Delta Live Tables 섹션을 참조하세요.

Auto Loader의 상당히 표준적인 응용 프로그램 중 하나는 파일을 수집하고 골드 레이어 집계 데이터 테이블까지 추가 수준의 변환, 강화 및 집계를 통해 Delta Lake 테이블에 데이터를 공급하는 프로세스를 통해 매달리온 아키텍처 설계의 일부로 이를 사용하는 것입니다. 일반적으로 이는 낮은 대기 시간, 높은 처리량, 종단 간 데이터 변환 파이프라인을 제공하는 스트리밍 프로세스의 소스이자 싱크인 Delta Lake를 사용하여 추가 데이터 계층 처리가 수행되는 경우에 수행됩니다. 이 프로세스는 파일 기반 수집에 대한 표준이 되었으며 더 복잡한 람다 아키텍처 기반 프로세스에 대한 필요성을 일부 제거했기 때문에 Databricks도 이 접근 방식을 중심으로 프레임워크를 구축했습니다.

## 델타 라이브 테이블

### 선언적 프레임워크

증분 수집, 간소화된 ETL 및 예상과 같은 자동화된 데이터 품질 프로세스를 결합한 Databricks는 **Delta Live Tables**라는 Delta Lake 위에서 실행되는 데이터 엔지니어링 파이프라인 프레임워크를 제공합니다. (DLT). 이는 cloudFiles 소스를 조사할 때 방금 설명한 것과 같은 파이프라인 구축을 단순화하는 데 도움이 됩니다. 이는 실제로 Delta Lake를 사용한 스트리밍에 대한 토론에 여기에 포함시키는 주된 이유를 설명합니다. 관리하기 쉬운 프레임워크에 이 가이드 전반에 걸쳐 언급된 주요 원칙이 포함되어 있습니다.

### 델타 라이브 테이블 사용

처리 파이프라인을 하나씩 구축하는 대신 선언적 프레임워크를 사용하면 많은 모범 사례를 자동화하여 논의한 많은 기능보다 더 적은 구문으로 일부 테이블과 뷰를 간단히 정의할 수 있습니다. 분야 전반에 걸쳐 일반적으로 사용됩니다. 사용자를 대신하여 관리하는 작업으로는 컴퓨팅 리소스, 데이터 품질 모니터링, 파이프라인 상태 처리, 최적화된 작업 조정 등이 있습니다.

DLT는 정적 테이블, 스트리밍 테이블, 뷰 및 구체화된 뷰를 제공하여 훨씬 더 복잡한 작업을 함께 연결합니다. 스트리밍 측면에서 자동 로더는 Delta Lake 지원 테이블 전체에서 다운스트림 증분 프로세스를 공급하는 눈에 띄고 일반적인 초기 소스로 간주됩니다. 다음은 설명서의 예를 기반으로 한 몇 가지 파이프라인 코드 예입니다 .

```
## 파일 가져오
기 dlt

@dlt.table def
autoloader_dlt_bronze(): return

( Spark.readStream.format("cloudFiles") .option("cloudFiles.format",
"json") .load("<데이터 경로>")
)

@dlt.table def
delta_dlt_silver(): return

(dlt.read_stream("autoloader_dlt_bronze")
...
<변환 논리>
```

```

    ...
)

@dlt.table def
live_delta_gold(): return

( dlt.read("delta_dlt_silver")
...
<집계 논라>
...
)

```

초기 소스는 스트리밍 프로세스이므로 실버 및 골드 테이블도 점진적으로 처리됩니다. 특히 스트리밍 소스에서 얻을 수 있는 이점 중 하나는 단순화입니다. 체크포인트 위치를 정의하거나 프로그래밍 방식으로 메타스토어에 테이블 항목을 생성할 필요가 없으므로 적은 노력으로 파이프라인을 구축할 수 있습니다.

간단히 말해서 DLT는 Delta Lake 위에 데이터 파이프라인을 구축하는 것과 동일한 많은 이점을 제공하지만 많은 세부 정보를 추상화하여 더 간단하고 쉽게 사용할 수 있도록 합니다.

## 데이터 피드 변경

앞서 우리는 변경 데이터 캡처(CDC) 데이터를 스트리밍 Delta Lake 파이프라인에 통합하는 것이 어떤 모습인지 살펴보았습니다. Delta Lake에는 이러한 유형의 피드를 지원하는 옵션이 있나요? 짧은 대답은 다음과 같습니다. 그렇습니다. 더 긴 답변을 얻으려면 먼저 이해 수준이 어느 정도인지 확인하겠습니다.

지금까지 우리는 Delta Lake를 사용하는 몇 가지 예를 살펴보았고 기본적으로 특정 데이터 행에 대해 레코드 삽입, 레코드 업데이트 또는 레코드 삭제라는 세 가지 주요 작업만 있다는 것을 확인했습니다. 이는 다른 데이터 시스템과 거의 유사합니다. 그렇다면 CDC는 정확히 어디에서 작동합니까?

데이터 엔지니어링 기초에서 Joe Reis와 Matt Housley가 정의한 대로 "변경 데이터 캡처(CDC)는 데이터베이스에서 발생하는 각 변경 이벤트(삽입, 업데이트, 삭제)를 추출하는 방법입니다. CDC는 거의 실시간으로 데이터베이스 간 복제를 수행하거나 다운스트림 처리를 위한 이벤트 스트림을 생성하는 데 자주 활용됩니다." 또는 더 간단히 말하면 "CDC는... 소스 데이터베이스 시스템에서 변경 사항을 수집하는 프로세스입니다."<sup>6</sup>

이를 초기 문의로 되돌리면 **변경 데이터 피드**라는 기능을 통해 Delta Lake에서 변경 내용 추적이 지원됩니다. (CDF). CDF의 기능은 Delta Lake 테이블의 변경 사항을 추적할 수 있다는 것입니다. 활성화되면 모든 것을 얻을 수 있습니다

---

<sup>6</sup> 데이터 엔지니어링의 기초: 강력한 데이터 시스템 계획 및 구축(Joe Reis 및 Matt Housley 저), p. 163, 피. 256, 오라일리, 2022.

발생하는 대로 테이블이 변경됩니다. 업데이트, 병합 및 삭제는 새로운 \_change\_data 폴더에 저장되며 추가 작업에는 이미 테이블 기록에 자체 항목이 있으므로 추가 파일이 필요하지 않습니다. 이 추적을 통해 다운스트림을 사용하기 위해 테이블의 변경 사항 피드로 결합된 작업을 읽을 수 있습니다. 변경 사항에는 변경 유형을 보여주는 일부 추가 메타데이터와 함께 필수 행 데이터가 포함됩니다.



이 기능은 Delta Lake 2.0.0 이상에서 사용할 수 있습니다. 글을 쓰는 시점에서 이 기능은 실험적 지원 모드에 있습니다.

열 매핑이 포함된 테이블에서 변경 데이터 피드 사용에 대한 지원 수준은 사용 중인 버전에 따라 다릅니다.

- 버전 <= 2.0에서는 열 매핑이 활성화된 테이블의 변경 데이터 피드에 대한 스트리밍 또는 일괄 읽기를 지원하지 않습니다. • 버전 2.1의 경우 열 매핑이 활성화된 테이블에 대해서는 일괄 읽기만 지원됩니다. 또한 이 버전에서는 추가되지 않는 스키마 변경(이름 바꾸기 또는 순서 변경 없음)이 없어야 합니다.
- 버전 2.2의 경우 비가산적 스키마 변경이 없는 한 열 매핑이 활성화된 테이블의 변경 데이터 피드에 대해 일괄 읽기와 스트리밍 읽기가 모두 지원됩니다.
- 열 매핑이 활성화된 테이블에 대한 변경 데이터 피드에 대한 버전 2.3 이상의 일괄 읽기는 이제 비부가적 스키마 변경을 지원할 수 있습니다. 사용 가능한 최신 버전의 테이블이 아닌 쿼리에 사용된 종료 버전의 스키마를 사용합니다. 지정된 버전 범위가 비추가적 스키마 변경에 걸쳐 있는 경우에도 여전히 오류가 발생할 수 있습니다.

### 변경 데이터 피드 사용 데이터 파이

프라인을 구축할 때 CDF 기능을 활용할지 여부는 궁극적으로 사용자에게 달려 있지만, 이를 잘 활용하여 일부 데이터 파이프라인을 처리하는 방식을 단순화하거나 재고할 수 있는 몇 가지 일반적인 사용 사례가 있습니다. 처리 작업.

다음은 이를 활용하는 방법에 대해 생각할 수 있는 몇 가지 예입니다.

#### 다운스트림 테이블 큐레이팅

소스 테이블에 대한 초기 작업 이후 행 수준 변경 사항만 처리하여 논리적 복잡성을 줄여 ETL 및 ELT 작업을 단순화함으로써 다운스트림 Delta Lake 테이블의 성능을 향상시킬 수 있습니다. 이는 현재 상태를 확인하기 전에 레코드가 어떻게 변경되고 있는지 이미 알고 있기 때문에 발생합니다.

### 변경 사항 전파 Kafka와 같

은 다른 스트리밍 싱크와 같은 다운스트림 시스템이나 이를 사용하여 데이터 파이프라인의 이후 단계에서 점진적으로 처리하는 데 사용할 수 있는 다른 RDBMS로 변경 데이터 피드를 보낼 수 있습니다.

### 감사 추적 만들기 변경 데이터

피드를 델타 테이블로 캡처할 수도 있습니다. 이는 삭제 발생 시기와 업데이트 내용을 포함하여 시간 경과에 따른 모든 변경 사항을 확인할 수 있는 영구 저장소와 효율적인 쿼리 기능을 제공할 수 있습니다. 이는 시간 경과에 따른 참조 테이블의 변경 사항을 추적하거나 민감한 데이터의 보안 감사에 유용할 수 있습니다.

또한 CDF를 사용해도 반드시 추가 스토리지가 추가되는 것은 아닙니다.

일단 활성화되면 우리가 실제로 발견한 것은 처리 오버헤드에 큰 영향이 없다는 것입니다. 변경 레코드의 크기는 매우 작으며 대부분의 경우 변경 작업 중에 작성된 실제 데이터 파일보다 훨씬 작습니다. 즉, 가능 활성화 시 성능에 미치는 영향이 거의 없습니다.

작업에 대한 변경 데이터는 트랜잭션 로그와 유사한 Delta 테이블 디렉터리 아래의 \_change\_data 폴더에 있습니다. 파일 추가 또는 전체 파티션 삭제와 같은 간단한 작업은 다른 유형의 변경보다 훨씬 간단합니다. 변경 사항이 이와 같이 단순한 유형인 경우 Delta Lake는 트랜잭션 로그에서 직접 변경 데이터 피드를 효율적으로 계산할 수 있으며 이러한 레코드는 폴더에서 완전히 건너뛸 수 있습니다. 이러한 작업은 종종 가장 일반적인 작업 중 하나이므로 이는 오버헤드를 줄이는 데 큰 도움이 됩니다.



현재 버전의 테이블 데이터에 포함되지 않으므로 \_change\_data 폴더의 파일은 테이블의 보존 정책을 따릅니다.

이는 보존 정책을 벗어나는 다른 트랜잭션 로그 파일과 마찬가지로 진공 작업 중에 제거될 수 있음을 의미합니다.

### 변경 피드 활성화 전반적으로

Delta Lake용 CDF를 구성하는 것까지는 별로 해야 할 일이 없습니다.<sup>7</sup> 실제로 요점은 그냥 켜는 것이지만, 이 작업은 새 피드를 생성하는지 여부에 따라 약간 다릅니다. 테이블 또는 기존 기능에 대한 기능을 구현하는 경우.

새 테이블의 경우 CREATE TABLE 명령 내에서 테이블 속성 delta.enableChangeDataFeed = true를 설정하면 됩니다.

---

<sup>7</sup> <https://docs.delta.io/latest/delta-change-data-feed.html#enable-change-data-feed>

```
## SQL
CREATE TABLE 학생 (ID INT, 이름 STRING, 연령 INT) TBLPROPERTIES (delta.enable ChangeDataFeed = true)
```

기존 테이블의 경우 ALTER TABLE 명령을 사용하여 테이블 속성을 변경하여 delta.enableChangeDataFeed = true로 설정할 수 있습니다.

```
## SQL
ALTER TABLE myDeltaTable SET TBLPROPERTIES (delta.enableChangeDataFeed = true)
```

Apache Spark를 사용하는 경우 Spark.databricks.delta.properties.defaults.enableChangeDataFeed를 true로 설정하여 이를 SparkSession 개체의 기본 동작으로 설정할 수 있습니다.

#### 변경 피드 읽기 변경 피드를 읽는 것은

Delta Lake를 사용하는 대부분의 읽기 작업과 유사합니다. 주요 차이점은 readChangeFeed를 true로 설정하여 있는 그대로의 데이터가 아니라 피드 자체를 변경하기를 원한다는 것을 읽기에서 지정해야 한다는 것입니다. 그렇지 않으면 구문은 시간 여행 또는 일반적인 스트리밍 읽기에 대한 옵션 설정과 매우 유사해 보입니다. 변경 피드를 일괄 작업으로 읽는 동작과 스트림 처리 작업으로 읽는 동작이 다르므로 각각을 차례로 살펴보겠습니다. 예제에서는 실제로 사용하지 않지만 maxFilesPerTrigger 또는 maxBytesPer Trigger를 사용한 속도 제한은 초기 스냅샷 버전이 아닌 버전에 적용될 수 있습니다. 사용하면 읽는 전체 커밋 버전이 예상대로 속도가 제한되거나 임계 값 미만일 때 전체 커밋이 반환됩니다.

일괄 처리에 대한 경계 지정. 일괄 작업은 제한된 프로세스이므로 변경 피드를 읽는 데 사용할 범위를 Delta Lake에 알려야 합니다. 버전 번호 또는 타임스탬프 문자열을 제공하여 시작 및 종료 경계를 모두 설정할 수 있습니다.<sup>8</sup> 설정한 경계는 쿼리에 포함됩니다. 즉, 최종 타임스탬프 또는 버전 번호가 커밋과 정확히 일치하면 해당 커밋의 변경 사항이 적용됩니다. 변경 피드에 포함됩니다. 특정 지점부터 사용 가능한 최신 변경 사항까지 변경 사항을 읽으려면 시작 버전이나 타임스탬프만 지정하세요.

경계 지점을 설정할 때 시간 이동 옵션을 설정하는 방법과 비슷한 방식으로 정수를 사용하여 버전을 지정하거나 타임스탬프에 대해 yyyy-MM-dd[ HH:mm:ss[.SSS]] 형식의 문자열을 사용해야 합니다. 제공한 타임스탬프나 버전이 변경 데이터 피드가 활성화된 이전 버전보다 낮거나 오래된 경우 변경 데이터 피드가 활성화되지 않았음을 알리는 오류가 발생합니다.

```
## Python #
버전(int 또는 long)
```

---

<sup>8</sup> <https://docs.delta.io/latest/delta-change-data-feed.html#read-changes-in-batch-queries>

```

(spark.read.format("delta") .option("readChangeFeed",
    "true") .option("startingVersion",
    0) .option("endingVersion",
    10) .table("myDeltaTable")
)

# 형식화된 타임스탬프로서의 타임스탬프

(spark.read.format("delta") .option("readChangeFeed",
    "true") .option("startingTimestamp", '2023-04-01
    05:45:46') .option( "endingTimestamp", '2023-04-21
    12:00:00') .table("myDeltaTable")
)

# 시작 버전/타임스탬프만 제공합니다

(spark.read.format("delta") .option("readChangeFeed",
    "true") .option("startingTimestamp", '2023-04-21 12:00:00.001') .테이블
    ("myDeltaTable")
)

```

# 파일 위치와 유사함

```

(spark.read.format("delta") .option("readChangeFeed",
    "true") .option("startingTimestamp", '2021-04-21 05:45:46') .load (/
    pathToMyDeltaTable")
)

```

스트리밍 프로세스에 대한 경계 지정. 테이블의 변경 피드에 readStream을 사용하려는 경우에도 StartingVersion 또는 StartingTimestamp를 설정할 수 있지만 마치 옵션이 제공되지 않은 것처럼 스트림이 테이블의 최신 스냅샷을 반환하는 것처럼 일괄 처리 사례보다 선택 사항입니다. 스트리밍 시에는 INSERT로, 이후의 모든 변경 사항은 변경 데이터로 저장됩니다.

스트리밍의 또 다른 차이점은 스트림이 무제한이고 종료 경계가 없기 때문에 종료 위치를 구성하지 않는다는 것입니다. 변경 데이터를 읽을 때 속도 제한 (maxFilesPerTrigger, maxBytesPerTrigger) 및 excludeRegex 와 같은 옵션도 지원되므로 그 외에는 평소대로 진행합니다.

```

## Python #
시작 버전 제공 (spark.readStream.format("delta")

    .option("readChangeFeed",
    "true") .option("startingVersion",
    0) .load("/pathToMyDeltaTable")
)

# 시작 타임스탬프 제공

```

```
(spark.readStream.format("델타")
    .option("readChangeFeed",
    "true") .option("startingTimestamp", "2021-04-21 05:35:43") .load("/
    pathToMyDeltaTable")
)

# 둘 중 하나도 제공하지 않음
(spark.readStream.format("delta")
    .option("readChangeFeed", "true") .load("/
    pathToMyDeltaTable")
)
```



지정된 시작 버전이나 타임스탬프가 테이블에 있는 최신 버전보다 높으면 timestampGreater ThanLatestCommit 오류가 발생합니다. 이 옵션을 설정하면 대신 빈 결과 집합을 수신하도록 선택하는 오류를 방지할 수 있습니다.

```
## SQL
세트 Spark.databricks.delta.changeDataFeed.timestampOutOfRange.enabled =
true; 시작 버전 또는 타임스탬프 값이
```

테이블에 있는 범위 내에 있지만 종료 버전 또는 타임스탬프가 범위를 벗어난 경우 이 기능을 활성화하면 지정된 범위 내에 있는 사용 가능한 모든 버전이 반환되는 것을 볼 수 있습니다.

## 개요

이 사점에서 변경 피드에서 수신되는 데이터가 전달될 때 정확히 어떻게 보이는지 궁금할 수 있습니다. 이전과 마찬가지로 데이터에 동일한 열이 모두 표시됩니다.

그렇지 않으면 테이블의 스키마와 일치하지 않기 때문에 이는 의미가 있습니다. 그러나 발생하는 변경 유형과 같은 사항을 이해할 수 있도록 몇 가지 추가 열을 얻습니다. 데이터를 변경 피드로 읽으면 데이터에 이러한 세 개의 새 열이 표시됩니다.

### 변경 유형

\_change\_type 열은 각 행에 대해 발생하는 변경이 삽입, update\_preimage, update\_postimage 또는 삭제 작업인지 식별하는 문자열 유형 열입니다. 이 경우 사전 이미지는 업데이트 전 일치된 값이고 사후 이미지 안에는 업데이트 후 일치된 값입니다.

### 커밋 버전

\_commit\_version 열은 변경 사항이 속한 트랜잭션 로그의 Delta Lake 파일/테이블 버전을 나타내는 긴 정수 유형 열입니다.

변경 피드를 일괄 프로세스로 읽는 경우 변경 피드는

쿼리에 대해 정의된 경계입니다. 스트림으로 읽으면 시작 버전 이상이고 시간이 지남에 따라 계속해서 증가합니다.

### 커밋 타임스탬프

\_commit\_timestamp 열은 \_com mit\_version 의 버전이 생성되어 로그에 커밋된 시간을 나타내는 타임스탬프 유형 열( yyyy-MM-dd[ HH:mm:ss[.SSS]] 형식)입니다 .

예를 들어 people10m 데이터 세트에 (가상) 불일치가 있는 다음 예가 있다고 가정해 보겠습니다. 잘못된 레코드를 업데이트할 수 있으며 변경 피드를 보면 사전 이미지로 표시된 원본 레코드 값과 사후 이미지로 표시된 업데이트된 값을 볼 수 있습니다. 잘못 입력된 이름에 대한 세트를 업데이트하고 개인의 이름과 성별을 수정하겠습니다. 그런 다음 변경 피드 기록 전후를 강조 표시하는 표의 하위 집합을 보고 어떻게 보이는지 살펴보겠습니다. 또한 커밋의 버전과 타임스탬프를 동시에 캡처한다는 점도 확인할 수 있습니다.

```
## SQL
업데이트
명10m
SET
성별 = 'F', firstName='리
아'
어디
firstName='레오' 및
lastName='Conkay';

## 파이썬 (
    스파

    크 .read.format("delta") .option("readChangeFeed",
    "true") .option("startingVersion",
    5) .option("endingVersion",
    5) .table("tristen.people10m") .select

    ( col("firstName"),
        col("lastName"),
        col("gender"),
        col("_change_type"),
        col("_commit_version")) .show()

+-----+-----+-----+-----+
|이름|성|성별|           _change_type|_commit_version|_commit_timestamp|
+-----+-----+-----+-----+
|레오|이|리|          남| update_preimage|      5|2023-04-05 13:14:40| 5|2023-04-05
||  레아| 콘케이|          F|update_postimage|      13:14:40|
+-----+-----+-----+-----+
```

## 추가 생각

여기서 우리는 이전 장에서 다룬 많은 개념을 기반으로 구축했으며 여러 가지 다른 용도에 걸쳐 적용할 수 있는 방법을 살펴보았습니다. 우리는 스트림 데이터 처리에 사용되는 몇 가지 기본 개념과 이러한 개념이 Delta Lake에서 어떻게 작동하는지 살펴보았습니다. 우리는 사용 방식의 유사성으로 인해 통합 API를 사용하여 핵심 스트리밍 기능(특히 Spark에서)이 어떻게 단순화되는지 간접적으로 확인했습니다. 그런 다음 Delta Lake를 통한 스트리밍 읽기 및 쓰기 동작을 보다 직접적으로 제어할 수 있는 몇 가지 다양한 옵션을 살펴보았습니다. 우리는 Apache Spark 또는 Databricks를 사용한 스트림 처리와 밀접하게 관련되어 있지만 Delta Lake 위에 구축된 일부 영역을 살펴보는 방식으로 이를 따라갔습니다. Delta Lake에서 사용할 수 있는 변경 데이터 피드 기능과 이를 스트리밍 또는 비스트리밍 애플리케이션에서 사용하는 방법을 검토하는 것으로 마무리했습니다. 이것이 Delta Lake 사용 영역에 대해 가질 수 있는 많은 질문이나 호기심에 대한 답변이 되기를 바랍니다. 그런 다음 Delta Lake에서 사용할 수 있는 다른 고급 기능 중 일부를 살펴보겠습니다.

## 주요 참고자료

• [스파크 최종 가이드](#) • [Apache Flink](#)

[를 사용한 스트림 처리](#) • [학습 스파크](#) • [스트리밍 시스템](#)



## 제5장

## 당신의 레이크하우스 건축하기

초기 출시 독자를 위한 참고 사항 초기 출시 eBook을 사용

하면 가장 초기 형태의 책, 즉 저자가 집필하는 그대로의 편집되지 않은 원본 콘텐츠를 얻을 수 있으므로 이러한 타이틀이 공식 출시되기 훨씬 전에 이러한 기술을 활용할 수 있습니다.

이것이 마지막 책의 11번째 장이 될 것이다. GitHub 저장소는 나중에 활성화될 예정입니다.

이 책의 내용 및/또는 예제를 개선할 수 있는 방법에 대한 의견이 있거나 이 장에 누락된 자료가 있는 경우 [gobrien@oreilly.com](mailto:gobrien@oreilly.com)으로 편집자에게 문의하십시오.

성공적인 엔지니어링 이니셔티브는 올바른 비전과 명확한 목적(우리가 수행하는 작업과 이유)은 물론 견고한 설계와 아키텍처(비전 달성을 방법)에서 시작됩니다. 사려 깊은 계획과 올바른 구성 요소(도구, 리소스, 엔지니어링 가능)를 결합하면 최종 결과가 임무를 반영하고 규모에 맞게 잘 수행되도록 할 수 있습니다. Delta Lake는 엔터프라이즈급 데이터 레이크하우스를 설계, 구성, 테스트, 배포 및 유지 관리할 수 있는 핵심 빌딩 블록을 제공합니다.

이 장의 목표는 단순히 아이디어, 패턴, 모범 사례 모음을 제공하는 것 이상으로 현장 가이드 역할을 하는 것입니다. 올바른 정보, 추론 및 정신적 모델을 제공함으로써 여기에서 배운 교훈은 자신의 데이터 레이크하우스를 설계할 때 사용할 명확한 청사진으로 통합될 수 있습니다. 레이크하우스 개념이 처음이시든, 충분 데이터 품질을 위한 메달리온 아키텍처에 익숙하지 않으시든, 스트리밍 데이터 작업을 처음 시작하시는 분이든 우리는 이 여정을 함께 할 것입니다.

우리가 배울 내용:

- 레이크하우스 아키텍처란 무엇입니까?
- Delta Lake를 Lakehouse Architecture 구현을 위한 기반으로 사용 - 사실
- 메탈리온 아키텍처
- 스트리밍 레이크하우스 아키텍처

## 레이크하우스 아키텍처

성공적인 엔지니어링 이니셔티브가 명확한 비전과 목적에서 시작되고 궁극적으로 자체 데이터 레이크하우스의 기반을 마련하는 것이 목표라면 먼저 레이크하우스가 무엇인지 정의해야 합니다.

### 레이크하우스란 무엇인가요?

“Lakehouse는 데이터 레이크의 유연성, 비용 효율성 및 규모를 기준 데이터 웨어하우스의 데이터 관리, 스키마 적용 및 ACID 트랜잭션과 결합한 개방형 데이터 관리 아키텍처입니다.” - 데이터브릭스

이 정의에서 풀어야 할 내용이 많습니다. 즉, 엔지니어링 및 데이터 관리 관점 모두에서 실제 경험이나 공유 정신 모델이 필요하다는 가정이 만들어지고 있습니다. 특히 이 정의에서는 데이터 웨어하우스와 데이터 레이크에 대해 잘 알고 있을 뿐만 아니라 사람들이 하나의 기술을 선택할 때 다른 기술을 선택할 때 고려해야 하는 장단점도 가정합니다. 다음 섹션에서는 각 선택의 장단점을 다루고 레이크하우스가 어떻게 탄생했는지 설명합니다.

데이터 웨어하우스와 데이터 레이크에서 공유되는 내역과 수많은 사용 사례는 이전에 제공 및 소비 공간에 걸쳐 역할을 수행해 본 사람이라면 누구에게나 제2의 천성이 되어야 합니다. 데이터 여정을 막 시작했거나, 데이터 웨어하우징에서 전환했거나, 데이터 레이크의 데이터로만 작업해 본 사람이라면 누구나 이 섹션을 참조하세요.

레이크하우스 아키텍처가 어디에서 진화했는지 이해하려면 다음 사항에 답할 수 있어야 합니다.

- 레이크하우스가 데이터 레이크와 데이터 웨어하우스의 장점을 결합한 하이브리드 아키텍처라면 그 부분을 합친 것보다 더 좋아야 합니다. • 유연성, 비용 효율성, 무제한 데이터 확장이 필요한 이유는 무엇입니까?

전통적인 데이터 레이크가 오늘날 우리 모두에게 중요합니까?

- 데이터 레이크의 이점이 기존 데이터 웨어하우스에서 영감을 받은 스키마 적용 및 발전, ACID 트랜잭션, 적절한 데이터 관리의 이점과 결합될 때만 진정으로 중요한 이유는 무엇입니까?

## 데이터 웨어하우스에서 배우기 데이터 웨어하우

스는 대기업 내의 데이터 파일로 문제를 해결하고 비즈니스 인텔리전스 및 분석적 의사 결정을 단순화하기 위해 등장했습니다. 데이터 웨어하우스는 특정 데이터 도메인 내의 구조화된 데이터 문제를 해결하기 위한 중앙 집중식 솔루션으로 존재하지만, 데이터 웨어하우스 아키텍처 내의 물리적 한계로 인해 비용은 웨어하우스 내 데이터의 크기와 규모에 비례하여 증가합니다. 물리적 제한의 근본 원인은 데이터가 수직 확장 아키텍처로 알려진 로컬(비분산)에 저장되었기 때문입니다.

비용은 대규모 데이터 웨어하우스의 제한 요소(수직적 확장으로 인해)이지만, 데이터 웨어하우스를 운영함으로써 얻을 수 있는 이점은 많은 독립적인 데이터 파일로 운영하는 것과 비교할 때 더 높은 비용보다 클 수 있습니다. 안전한 데이터 관리, 액세스 정책, 규칙 및 표준 시행을 염두에 두고 설계되었습니다. 데이터 웨어하우스는 일관성을 최우선으로 고려하여 구축되었습니다. 이는 이제 자체적인 데이터 품질 범위에 포함되는 데이터의 정확성을 고려할 때 많은 것을 의미합니다. 유형이 안전하고 구조화된 데이터 및 스키마 적용을 지원하는 데이터 웨어하우스는 일관된 테이블과 명확한 데이터 정의를 제공해야 하는 기본 비즈니스 인텔리전스 및 운영 데이터 시스템에 일반적으로 활용됩니다.

데이터 관리 측면에서 권한 부여라고 하는 사용자 및 역할 기반 권한을 통한 액세스 제어 지원을 통해 보안 및 규칙 기반 시스템을 통해 어떤 사용자가 읽기(선택), 쓰기(삽입), 업데이트 및 삭제를 실행할 수 있는지 제어할 수 있습니다. 웨어하우스의 후속 테이블 및 뷰 내의 데이터.

비용 외에도 데이터 웨어하우스 아키텍처가 오늘날의 요구 사항에 맞게 확장되지 못하는 문제는 데이터 과학 및 기계 학습을 포함한 다양한 종류의 워크로드를 지원하는 유연성이 부족하기 때문에 발생합니다.

오늘날 비정형(이미지), 반정형(csv, json) 및 완전 정형 데이터(parquet/orc)를 지원하는 사용자 지정 데이터 유형 및 형식이 필요한 일반적인 기계 학습 및 데이터 과학 워크플로에 대한 지원이 누락되었습니다. 반복 알고리즘을 위해 값비싼 쿼리를 여러 번 만들 필요 없이 효율적인 파일 건너뛰기, 열 정리를 통해 전체 테이블을 메모리로 쉽게 읽을 수 있습니다.

불행하게도 데이터 과학에 대한 지원이 부족하여 더 많은 데이터 복사로 인해 파일로가 발생했습니다. 데이터를 데이터 레이크에 저장해야 하는 동시에 데이터를 웨어하우스에 유지해야 하는 분석가와 비즈니스 인텔리전스 담당자를 지원해야 하기 때문입니다.

## 데이터 레이크에서 학습 데이터 레이

크는 분산 파일 시스템 내에서 다양한 형식(csv, json, orc, 텍스트, 바이너리)으로 원시(처리되지 않은) 데이터를 저장하기 위해 등장했습니다. 당시 인기 있는 선택은 HDFS(Hadoop 분산 파일 시스템)였습니다. 상용 하드웨어를 활용하면 데이터 레이크를 활용하여 분산 처리 작업을 실행할 수 있습니다(Map

축소) 또는 활용하여 데이터 웨어하우스에 로드할 데이터를 위한 준비 영역 역할을 합니다. 오늘날 많은 워크로드는 여전히 유사한 패턴을 따르며 클라우드 기반 개체 저장소 또는 기타 관리형 탄력적 스토리지 및 탄력적 컴퓨팅을 활용하여 데이터 레이크를 지원합니다. 그렇다면 이것이 레이크하우스 이야기에 어떻게 들어맞을까요?

데이터 레이크는 데이터 과학 및 기계 학습 사용 사례를 위해 직접 처리할 수 있는 원시 데이터 피드(파일)를 저장하기 위한 솔루션을 제공하며, 데이터 웨어하우스 내에서 사용할 수 없는 데이터 형식을 지원합니다. 발견된 데이터 피드

다음 섹션에서 다루는 이중 계층 데이터 아키텍처를 사용하여 데이터 웨어하우스를 동기화 상태로 유지하도록 변환되는 또 다른 용도도 있습니다.

데이터 레이크의 이점은 데이터 웨어하우스와 비교할 때 상대적으로 낮은 비용과 파일 형식 유연성에 대한 일반적인 지원과 관련이 있습니다.

파일 형식의 유연성은 양날의 검 역할도 합니다. 데이터 레이크는 스키마 없이 그대로 유지되어 파일 시스템 내에 무언가든 저장할 수 있으므로 현재 한 가지 형식으로 존재하는 것이 내일 쉽게 바뀔 수 있습니다.

좋은 점은 스토리지와 컴퓨팅이 분리되어 있다는 것은 데이터가 실행될 때까지 비용이 낮게 유지되고 최소한의 오버헤드가 필요하다는 것을 의미합니다.

안타깝게도 데이터 레이크의 스키마가 없는 특성으로 인해 오래된 데이터 세트를 스토리지에서 깨낼 때 상황이 항상 잘 진행되는 것은 아닙니다. 데이터 레이크가 '데이터 늪'이라는 이름을 갖게 된 가장 큰 이유 중 하나는 손상된 데이터입니다.

데이터 웨어하우스와 더 멀리 떨어져 있는 데이터 레이크는 트랜잭션, 운영 수준 격리를 지원하지 않으며 결과적으로 데이터 레이크에서 동일한 리소스 세트를 공유하는 여러 동시 데이터 생산자 또는 소비자에 대한 지원이 부족합니다. 일관성과 관련하여 활성 판독기와 기록기 간의 일관된 상태를 달성하거나 동일한 물리적 테이블에서 작동하는 배치 및 스트리밍 작업에서 오늘날 더 일반적인 것과 같은 다중 액세스 모드를 지원하는 것은 거의 불가능합니다.

규칙이 없는 데이터 레이크는 결국 데이터 불안정, 사용할 수 없는 데이터로 이어진다는 점과 최악의 경우 완전히 "오염"되거나 "독성"이 있는 데이터 레이크로 이어진다는 사실을 이해하면서 "두 세계의 장점을 모두 얻을 수 있다면 어떨까?"라는 급진적인 아이디어가 떠올랐습니다."

## 이중 계층 데이터 아키텍처

이중 계층 아키텍처는 데이터 레이크와 웨어하우스 간의 관계가 자연스럽게 발전한 것입니다. Airflow와 같은 오픈스택 플랫폼을 염두에 두십시오.

Airflow가 인기 있는 이유는 데이터 레이크와 데이터 웨어하우스 간의 일관성을 관리하기 어렵다는 사실에 있습니다. 둘 다 관리할 수 있는 방법이 있다면 어떨까요?

운영 데이터 시스템(사일로화된 데이터)에서 데이터 웨어하우스(공유) 또는 데이터 레이크로 단일 흐름을 갖는 대신 이 중 계층 아키텍처는 ETL(추출-변환-로드) 작업을 사용하여 일관성을 관리했습니다. 다음 작업 세트를 고려하십시오.

1. 원본 데이터베이스 A의 운영 데이터를 데이터 레이크(위치 a)에 씁니다.
2. (위치 a)의 데이터를 읽고, 정리하고, 변환하고 변경 사항을 다음에 씁니다.  
(위치 b)
3. (위치 b)에서 읽고 (위치 c)의 데이터를 결합하고 정규화하여  
착륙 구역(위치 d)
4. (위치 d)에서 데이터를 읽고 비즈니스에서 사용할 수 있도록 데이터 웨어하우스에 씁니다.

워크플로가 완료되는 한 데이터 레이크의 데이터는 웨어하우스와 동기화되며 테이블 언로드 또는 다시 로드를 지원하여 데이터 웨어하우스의 비용을 절감합니다.

이것은 들이켜 보면 의미가 있습니다.

데이터에 대한 직접 읽기 액세스를 지원하려면 기계 학습 사용 사례를 지원하기 위한 데이터 레이크가 필요하고, 비즈니스 및 분석 처리를 지원하려면 데이터 웨어하우스가 필요합니다. 그러나 추가된 복잡성으로 인해 데이터 엔지니어는 여러 진실 소스를 관리해야 하는 부담, 모든 동일한 데이터의 여러 복사본을 유지 관리하는 비용(데이터 레이크에서 한 번 이상, 데이터 웨어하우스에서 한 번) 및 오래된 데이터가 무엇인지, 어디에 있는지, 이유가 무엇인지 파악하는 데 어려움을 겪습니다.

두 가지 진실과 거짓말 게임을 해본 적이 있다면 이것은 구조적으로 동일하지만 재미있는 게임이라기보다는 위험이 훨씬 더 높습니다. 이는 결국 우리의 귀중한 운영 데이터입니다. 두 가지 진실 소스는 정의에 따라 두 시스템이 동기화되지 않아 서로의 진실 버전을 말할 수 있음을 의미합니다. 이는 또한 진실의 각 출처도 거짓말을 하고 있음을 의미합니다. 그들은 단지 인식하지 못합니다.

그래서 질문은 여전히 공중에 있습니다. 두 세계의 장점을 최대한 활용하고 데이터 레이크와 데이터 웨어하우스를 효율적으로 결합할 수 있다면 어떨까요? 바로 그곳이 데이터 레이크하우스가 탄생한 곳입니다.

## 레이크하우스 건축

레이크하우스는 최고의 데이터 웨어하우스와 최고의 데이터 레이크를 결합한 하이브리드 데이터 아키텍처입니다. 그림 5-1은 데이터 웨어하우스, 데이터 레이크, 데이터 레이크하우스 등 세 가지 데이터 아키텍처 각각에 어떤 사용 사례가 포함될 수 있는지에 대한 렌즈를 통해 간단한 개념 흐름을 제공합니다.

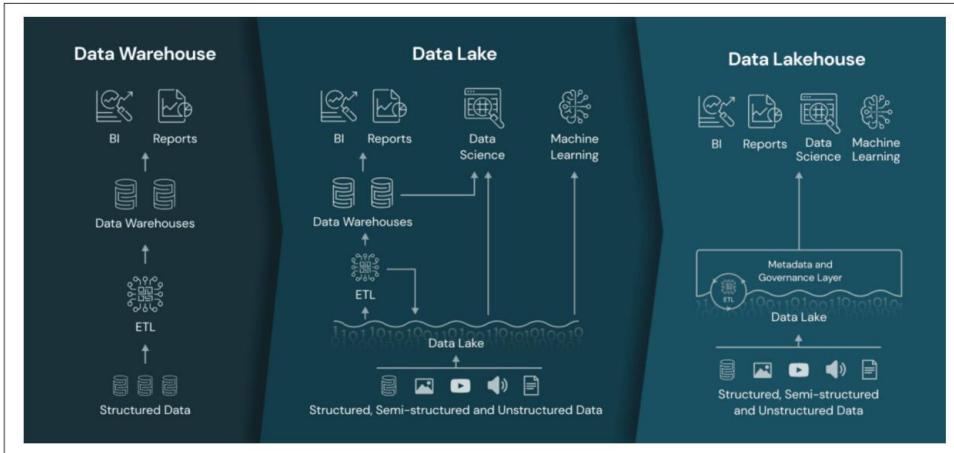


그림 5-1. Data Lakehouse는 BI 및 보고를 위한 공통 인터페이스를 제공하는 동시에 데이터 과학 및 기계 학습 워크플로가 단일 통합 방식으로 지원되도록 보장합니다.

이 새로운 아키텍처는 개방형 시스템 설계를 통해 구현됩니다. 즉, 데이터 레이크에 사용되는 일종의 저비용 스토리지에서 데이터 웨어하우스와 유사한 데이터 구조 및 데이터 관리 기능을 직접 구현하는 것입니다.

이를 단일 시스템으로 병합하면 데이터 팀이 여러 시스템에 액세스할 필요 없이 데이터를 사용할 수 있으므로 더 빠르게 움직일 수 있습니다. 이는 데이터 웨어하우스와 데이터 레이크 사이의 경계를 허무는 동시에 단일 정보 소스를 제공합니다. 이는 이중 계층 아키텍처에 비해 큰 이점이며 어느 쪽(웨어하우스 또는 레이크)에 데이터가 있는지 파악하는 문제를 방지합니다. 올바른 데이터, 동기화되지 않은 사람, 그리고 올바른 답을 찾기 위해 관련된 모든 비용이 많이 드는 작업입니다.

또한 이러한 이점을 통해 팀은 데이터 과학, 기계 학습 및 비즈니스 분석 프로젝트에 사용할 수 있는 가장 완전한 최신 데이터를 확보할 수 있습니다.

#### Data Lakehouse의 아키텍처 기둥

- 트랜잭션 지원 • 스키마 적

용 및 거버넌스 - 감사 로그 및 데이터 무결성 • SQL 및 JDBC와 같은 개방형 인터페이스를 통한 BI 지원 • 스토리지와 컴퓨팅 분리 • 개방형 표준: 개방형 API 및 개방형 데이터 형식 • 엔드 투 엔드 스트리밍 • 다양한 지원 기존 SQL에서 딥 러닝까지의 워크로드

## Delta Lake 기반

우리는 방금 Lakehouse를 단생시킨 성공적인 아이디어 결합에 대해 배웠습니다.

데이터 웨어하우스 방식에 제한이 없고 고가용성, 거의 무한한 확장성, 비용 효율적인 스토리지 분리와 데이터 레이크 컴퓨팅의 이점을 누릴 수 있는 설계입니다.

이 섹션에서는 Delta Lake를 통해 기본적으로 얻을 수 있는 이점과 이것이 Lakehouse에 전력을 공급하는 데 적합한 도구인 이유를 설명합니다.

개방형 생태계의 개방형 표준에 대한 오픈 소스 Delta Lake를 사용하여 레이크하우스를 설계하면

개방형 표준과 개방형 프로토콜, 상식 및 표준 규칙에 초점을 맞춘 개방형 생태계에 대한 약속이 함께 제공됩니다.

### 개방형 파일 형식

Apache Parquet는 델타 테이블에 저장된 데이터의 실제 파일 형식입니다.

빅 데이터 커뮤니티 내에서 널리 지원되는 Parquet는 이미 속도와 확장성 측면에서 그 가치를 입증했지만 시간이 지나도 유지 관리가 어려웠습니다. Parquet 자체는 스키마 유효성 검사나 진화를 제공하지 않습니다. 또한 열 재매핑도 지원하지 않습니다.

Delta가 테이블에 가져오는 가장 큰 차이점은 시간이 지남에 따라 연속적인 데이터 모음으로 처리될 때 표준 쪽 마루가 손상될 수 있는 스키마 변환과 시간이 지남에 따른 미묘한 변화에서 기본 쪽 마루가 살아남을 수 있도록 하는 일관성과 열 수준 보장입니다.

Parquet는 열 기반 분석 데이터의 표준 파일 형식입니다. 따라서 내부 독점 테이블 형식 및 액세스 프로토콜을 구현하는 대신 Delta 프로토콜은 커뮤니티에서 새로운 도구 및 커넥터(5장에서 살펴봤음)를 구축하는 데 무료로 사용할 수 있으며 제공된 많은 제품 내에서 기본적으로 사용할 수 있습니다. Amazon, Microsoft, Starburst 및 Databricks와 같은 주요 클라우드 서비스 공급업체가 제공합니다.

자체 설명 테이블 메타데이터 각 델타 테이블의

메타데이터는 실제 테이블 데이터와 함께 저장됩니다. 이 설계를 사용하면 주어진 테이블을 간단히 설명하기 위해 Hive Metastore와 같은 별도의 Metastore를 유지할 필요가 없습니다. 설계 결정을 통해 정적 테이블을 보다 효율적으로 복사하고 표준 파일 시스템 도구를 사용하여 이동할 수 있으며, 7장의 SHALLOW CLONE에서 본 것처럼 테이블의 메타데이터 전용 복사본도 존재할 수 있습니다.

## 오픈 테이블 사양 마지막으로

로 공급업체 종속에 대한 두려움이 없습니다. 전체 Delta Lake 프로젝트 자체는 Linux Foundation을 통해 전체 오픈 소스 커뮤니티에 무료로 제공되며 이를 중심으로 좋은 커뮤니티가 있습니다.

### 델타 범용 형식 (\*1UniForm)

UniForm은 Delta Lake 3.0에 도입된 새로운 기능입니다. UniForm을 사용하면 애플리케이션에 필요한 형식으로 Delta를 읽을 수 있어 호환성이 향상되고 생태계가 확장됩니다. Delta UniForm은 Apache Iceberg 또는 Apache Hudi에 필요한 메타데이터를 자동으로 생성하므로 사용자는 미리 선택하거나 오류가 발생하기 쉬운 형식 간에 수동 변환을 수행할 필요가 없습니다. UniForm을 사용하면 Delta는 Lakehouse에 상호 운용성을 제공하는 생태계 전반에서 작동하는 범용 형식입니다.

### 트랜잭션 지원 트랜잭션 지원은 데이터

터 정확성과 순차적 삽입 순서가 중요할 때마다 중요합니다. 아마도 이는 거의 모든 생산 사례에 필요합니다. 우리는 항상 최소한의 높은 기준을 달성하는 데 관심을 기울여야 합니다. 트랜잭션은 추가 확인과 균형이 있음을 의미하지만, 예를 들어 여러 작성자가 테이블을 변경하는 경우 항상 충돌 가능성성이 있습니다. 분산형 델타 트랜잭션 프로토콜의 동작을 이해한다는 것은 어떤 쓰기가 어떻게 승리해야 하는지 정확히 알고 읽기에 대한 데이터 삽입 순서가 정확하다는 것을 보장할 수 있다는 것을 의미합니다.

### 직렬화 가능한 쓰기

Delta는 쓰기 직렬화라는 기술을 사용하여 여러 동시 작성자를 활성화하는 동시에 트랜잭션에 대한 ACID 보장을 제공합니다. INSERT 작업과 같이 새 행이 단순히 테이블에 추가되는 경우 커밋이 발생하기 전에 테이블 메타데이터를 읽을 필요가 없습니다. 그러나 테이블이 더 복잡한 방식으로 수정되는 경우(예: 행이 삭제되거나 업데이트되는 경우) 쓰기 작업이 커밋되기 전에 테이블 메타데이터를 읽습니다. 이 프로세스는 변경 사항이 커밋되기 전에 변경 사항이 충돌하여 델타 테이블의 실제 순차 삽입 및 작업 순서가 손상될 가능성이 없도록 보장합니다.

손상 위험이 있는 대신 충돌로 인해 동시 수정 유형에 따라 발생하는 특정 예외 집합이 발생합니다.

---

1 UniForm은 이번 초기 릴리스부터 "출시 예정"입니다.

### 읽기를 위한 스냅샷 격리 특정 델타 테이

블을 읽는 프로세스는 여러 동시 기록의 복잡성으로부터 격리되며 정확한 직렬 순서로 델타 테이블의 일관된 스냅샷을 읽도록 보장됩니다.

### 증분 처리 지원 각 테이블에는 테이블의 원자 버전

에 대한 단일 직렬 기록이 포함되어 있으며 테이블의 각 버전에 대한 상태는 스냅샷에 포함되어 있습니다. 즉, 특정 버전(시점)에 Delta 테이블에서 읽는 프로세스(작업)는 로컬 테이블 스냅샷과 테이블의 현재(최신) 버전 사이의 특정 변경 사항만 직관적으로 읽을 수 있습니다.

증분 처리는 커서(마지막 오프셋, ID) 또는 더 복잡한 상태를 유지 관리하는 작업 부담을 줄여줍니다. 예제 5-1을 고려하십시오. 우리는 아마도 경력에서 이와 같은 작업을 본 적이 있거나 시작 타임스탬프, 읽고 쓰고 삭제할 레코드 수를 취하고 마지막으로 성공한 배치에서 식별된 마지막 레코드를 취하고 있다고 추측할 수 있습니다.. 일괄 작업이 체크포인트를 사용한다고 말하는 것이 더 쉽습니다. 하지만 상태를 유지하는 일에는 쉬운 일이 없습니다.

### 실시예 5-1. 상태 비저장 배치 작업에 상태 제공

```
% ./run-some-batch-job.py \
--startTime x \ --
recordsPerBatch 10000 \ --lastRecordId
z
```

Delta Lake를 사용하면 StartingVersion을 사용하여 테이블에서 읽을 특정 지점을 제공할 수 있습니다. 예제 5-2는 시작 버전과 동일한 작업을 간략하게 보여줍니다.

### 예제 5-2. 상태 비저장 배치 작업에 Delta StartingVersion 제공

```
% ./run-some-batch-job.py --startingVersion 10
```

#### 시간 여행 지원 잘못된 삽입을 기

반으로 테이블을 되감고 재설정하는 기능 외에도 트랜잭션에서 가장 큰 이점은 이 기능(시간 여행)을 활용하여 특정 지점에서 주어진 테이블의 상태를 보는 것과 같은 새로운 작업을 수행하는 기능입니다. 시간이 지나면 변경 사항을 비교할 수 있습니다. 이는 소수의 데이터 엔지니어가 필요하다는 것을 알고 있는 유리한 지점이자 각 테이블에 기록이 있고 해당 기록은 git 기록 또는 git 비난과 매우 유사하므로 MTTR(평균 해결 시간)을 대폭 줄일 수 있는 기능입니다. 익숙한 것.

## 스키마 적용 및 거버넌스

다음 컨테스트의 거버넌스는 테이블을 구성하는 열, 열 유형 및 설명 메타데이터를 관리하는 특정 테이블 정의(DDL)의 구조를 제어하는 규칙에 적용됩니다. 스키마 적용은 잘못된 콘텐츠를 테이블에 쓰려고 시도한 결과와 관련이 있습니다.

Delta Lake는 쓰기 시 스키마를 사용하여 클래식 데이터베이스에 필요한 높은 수준의 일관성을 달성하고 사람들이 데이터베이스 관리 시스템(DBMS) 내에서 의존하게 된 거버넌스를 지원합니다. 명확성을 위해 다음에는 쓰기 시 스키마와 읽기 시 스키마의 차이점을 다루겠습니다.

### 쓰기 시 스키마

Delta Lake는 쓰기 시 스키마 및 선언적 스키마 진화를 지원하므로 올바른지에 대한 책임은 지정된 Delta Lake 테이블에 대한 데이터 생산자에게 있습니다.

그러나 이것이 '데이터 생산자'라는 모자를 썼다고 해서 모든 것이 된다는 의미는 아닙니다. 데이터 레이크는 거버넌스 부족으로 인해 데이터 높이 될 뿐이라는 점을 기억하세요. Delta Lake를 사용하면 커밋된 초기 성공적인 트랜잭션 테이블 열과 유형을 식별하는 단계를 자동으로 설정합니다. 거버넌스 모자를 쓴 상태에서 우리는 이제 트랜잭션 로그에 기록된 규칙을 준수해야 합니다. 조금 무섭게 들릴 수도 있지만 안심하세요. 이는 데이터 생태계를 개선하기 위한 것입니다. 스키마 적용에 대한 명확한 규칙과 스키마 진화를 처리하기 위한 적절한 절차를 통해 테이블 구조 수정 방법을 제어하는 규칙은 궁극적으로 주어진 테이블의 소비자를 문제가 되는 상황으로부터 보호합니다.



일관된 데이터 및 품질 기대 현실 세계에서 불변성을 적용

하면 누가 무엇을, 언제, 어디서 위반했는지에 대한 대화가 줄어듭니다. Delta Lake를 사용하면 mergeSchema 옵션을 자주 사용하지 않고 사람들이 overwriteSchema를 사용하려는 경우 매우 우려한다는 의미입니다. 몇 가지 확립된 작업 방식으로 Delta Lake를 사용하는 경우 DeltaLog는 중재를 위한 진실의 소스가 되어 불필요한 회의를 효과적으로 제거합니다. 보기만 해도 문제가 발생한 경우 근본 원인을 자동으로 찾아낼 수 있기 때문입니다. DeltaTable.forName(spark, …).history(10)에서.



### 읽기 시 스키마

데이터 레이크는 본질적으로 미화된 분산 파일 시스템인 데이터 레이크 고유의 일관된 형태의 거버넌스 또는 메타데이터가 없기 때문에 읽기 시 스키마 접근 방식을 사용합니다. 읽기 스키마는 유연하지만 그 유연성은 데이터 레이크가 서부와 같이 분류되는 이유이기도 합니다. 통제되지 않고 혼란스럽고 문제가 없는 경우가 많습니다.

이것이 의미하는 바는 일부 위치(디렉터리 루트)에 일부 파일 형식(json, csv, 바이너리, 쪽 모이 세공, 텍스트 등)의 데이터가 있고 파일을 특정 위치에 기록할 수 있다는 것입니다. 무 제한으로 커지면 데이터 세트의 수명이 길어짐에 따라 문제가 커질 가능성이 높습니다.

특정 위치에 있는 데이터 레이크의 데이터 소비자로서 운이 좋으면 데이터를 추출하고 구문 분석할 수 있을 수도 있습니다. 운이 좋다면 일종의 문서가 있을 수도 있습니다. 리드 타임과 계산을 통해 작업을 수행할 수 있습니다. 그러나 적절한 거버넌스와 유형 안전성이 없으면 데이터 레이크는 저렴한 스토리지 오버헤드로 본질적으로 데이터 쓰레기인 돈을 태우는 것을 좋아하는 경우 수 테라바이트, 페타바이트로 빠르게 성장할 수 있습니다. 이는 극 단적인 진술이지만 많은 데이터 조직에서도 현실이 되고 있습니다.

### 스토리지와 컴퓨팅 Delta Lake의 분리는

스토리지와 컴퓨팅을 명확하게 분리합니다. 데이터 레이크 아키텍처의 가장 큰 이점 중 하나는 무 제한 스토리지의 유연성과 파일 시스템 확장성입니다. 레이크하우스 아키텍처는 데이터 레이크의 이점을 채택합니다. 오늘날에는 엄청난 양의 데이터를 생산하고 소비하는 것이 최신 데이터, 분석 및 기계 학습의 영역과 함께 제공되기 때문입니다.

이론적으로는 기본 파일 형식(마루)에 대한 독선적인 지원과 함께 스키마 적용, 적합성 및 진화(작성 시 스키마의 불변성과 함께 제공)에 대한 엄격한 거버넌스가 있는 한 다음과 같은 이점을 얻을 수 있습니다. 상호 운용이 가능하고 휴대성이 뛰어난 파일 형식을 사용하여 데이터 레이크하우스에 있는 데이터에 대해 거의 무한한 확장성을 제공합니다. 이식성 측면은 더욱 세분화될 수 있습니다. 트랜잭션 로그를 포함하여 모든 테이블의 무결성을 유지하면서 Delta Lake 테이블을 한 클라우드에서 다른 클라우드로 가져갈 수 있습니다(레이크하우스 전체를 정리하여 이동).



### 논리적 작업과 물리적 반응 간의 분리 Delta Lake 내의 논리적 작업

과 기본 물리적 스토리지 계층의 결과적인 물리적 작업 간에는 훨씬 더 많은 분리가 있다는 점을 지적할 가치가 있습니다. 6장에서 테이블 정리를 예로 들어보겠습니다. 주어진 테이블에서 DELETE FROM을 호출하는 것과 실제 파일이 영향을 받는(실제로 삭제되는) 시기 사이에는 구분이 있습니다. 이는 실수로 삭제한 항목을 제거할 수 있는 시간 이동 기능(되감기/실행 취소) 때문입니다. 복원 가능성 없이 데이터 무결성에 해를 끼칠 수 있는 삭제입니다. 실수로 데이터를 삭제하는 일은 모든 사람이 경력의 한 지점 또는 다른 지점에서 발생했지만 모든 사람이 그것을 인정하는 것은 아닙니다! 이것이 바로 VACUUM 및 REORG 작업이 매우 중요한 이유입니다. 실제로 파일을 삭제하려면 물리적인 반응이 있는 작업이 발생해야 합니다.

### 트랜잭션 스트리밍 지원 9장에서 Delta

의 스트리밍 기능을 소개했습니다. Delta를 사용한 특정 작업(인바운드 읽기 또는 아웃바운드 쓰기)에 관계없이 트랜잭션 테이블 전체에서 배치와 스트리밍 사이를 쉽게 전환하는 기능은 처음에는 마술처럼 들릴 수 있습니다. 많은 스트리밍 파이프라인이 외부 힘(예: 누락된 데이터를 대체하기 위한 덮어쓰기 작업)에 의한 테이블 변경으로 인해 소스 테이블에서 분산 파일이 갑자기 사라지면서 예상치 못한 종료를 맞이했지만, Delta에서는 다중 버전 동시성 제어가 완벽하게 지원됩니다. 이는 테이블에서 읽는 스트리밍 애플리케이션이 다른 작성기 작업으로 인해 중단되지 않음을 의미합니다.

Delta Lake는 속도 때문에 품질을 희생하지 않고도 완전한 엔드투엔드 스트리밍을 지원합니다. 모든 것에는 장단점이 있으며 빠르게 진행하고 맹목적으로 운영하기 쉽습니다. 현실 세계에서는 속도에 대한 필요성과 자연으로 인한 비용을 비교하고 올바른 균형을 달성하기 위해 비즈니스 또는 데이터 팀이 기꺼이 취할 트레이드오프에 대해 일반적인 합의에 도달하는 것이 더 좋습니다. 우리는 항상 케이크를 먹고 먹을 수는 있지만 시간 여행을 통해 거의 모든 것이 가능합니다.

### 분석 및 ML 워크로드를 위한 통합 액세스 결국 Delta는 광범위

한 데이터 관련 솔루션에 대한 균형 잡힌 접근 방식을 제공합니다. 데이터 분석가와 BI 엔지니어는 간단한 SQL을 사용하여 쉽게 쿼리할 수 있으며 Delta Lake 테이블을 포함하는 데이터에 대한 효율적인 직접 물리적 파일 액세스도 동시에 지원합니다. 이는 모든 열 형식에 직접 액세스하는 데이터 과학 및 ML 워크로드에 대한 올바른 운영 모델을 제공합니다. 데이터는

---

2 일반적인 추가 스타일의 경우 테이블에 씁니다. 테이블 덮어쓰기 또는 테이블 삭제와 같은 다른 작업은 스트리밍 애플리케이션에 영향을 미칠 수 있습니다.

작업 범위 내에서 반복 알고리즘을 (내부) 실행하는 기능을 포함하여 필요합니다.

### Delta Sharing Protocol

내부 및 외부 이해관계자 간에 데이터를 안전하고 안정적으로 공유하는 것은 데이터 모델링 이후 가장 어려운 문제 중 하나입니다. 예를 들어 하나의 S3 버킷에서 다른 버킷으로 데이터 레이크 외부로 데이터를 내보내는 ETL 작업을 보는 것이 일반적인 관행입니다. 기본적으로 FTP(파일 전송 프로토콜)를 사용하여 데이터를 보내고 받는 이유는 IAM(ID 및 액세스 관리) 및 상호 운용 가능한 데이터 형식에 대한 표준이 누락되었기 때문입니다. 델타 공유 프로토콜은 이 문제를 해결합니다.

**그림 5-2는** 델타 공유 프로토콜을 보여줍니다. 물리적 Delta 테이블은 단일 정보 소스로 존재하며 Delta 공유 서버의 도입으로 안전하고 안정적인 데이터 교환을 제공하는 데 필요한 누락된 액세스 제어 및 거버넌스가 추가됩니다.

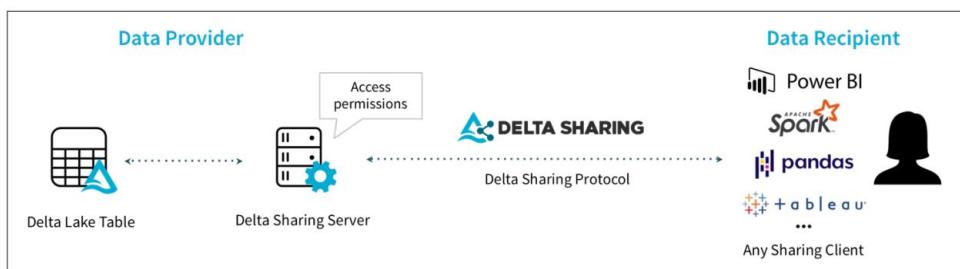


그림 5-2. 델타 공유 프로토콜 안전한 데이터 공유를 위한 업계 최초의 개방형 프로토콜로, 사용하는 컴퓨팅 플랫폼에 관계없이 다른 조직과 데이터를 간단하게 공유할 수 있습니다.

Delta 공유 프로토콜을 사용하면 내부 또는 외부 이해관계자가 Delta 테이블에 안전하게 직접 액세스할 수 있습니다. 이를 통해 데이터를 내보낼 때 발생하는 운영 비용을 제거하는 동시에 시간, 비용 및 엔지니어링 건전성을 절약하는 동시에 플랫폼에 구애받지 않는 공유 진실 소스를 제공합니다.

Delta 프로토콜이 제공하는 일반 기능은 데이터 레이크하우스에 필요한 기본 기능을 지원합니다. 이제 우리는 메달리온 아키텍처(Medallion Architecture)라는 목적 중심의 계층화된 데이터 아키텍처를 사용하여 레이크하우스 내에서 데이터 품질을 설계하는 방법을 좀 더 구체적으로 살펴보아야 할 때입니다.

## 메달리온 아키텍처

이동 중인 데이터는 형태와 크기가 다르고 정확성과 완전성이 다양하여 도착하자마자 지저분합니다. 모든 데이터가 수많은 최종 사용자 기대, 기존 데이터 계약 및 확립된 데이터 품질 검사를 준수하지는 않는다는 점을 인정하고,

정시에 도착하거나 적시에 도착하는 것이 이러한 데이터 품질 문제를 해결하는 데 핵심입니다. 이러한 과제로 인해 데이터 엔지니어링 팀은 주관적이고 객관적인 요구 사항의 역동적인 환경을 지속적으로 제공해야 한다는 높은 수준의 압력을 받게 되었으며, 이러한 집단적 노고를 바탕으로 메달리온 아키텍처가 탄생했습니다.

Medallion Architecture는 Lakehouse의 데이터를 논리적으로 구성하는 데 사용되는 데이터 디자인 패턴입니다. 이는 데이터세트를 점진적으로 개선하기 위한 프레임워크를 제공하기 위해 일련의 격려된 데이터 레이어를 사용하여 수행됩니다. 그림 5-4는 초기 수집 시점(브론즈)부터 가변 계보를 거쳐 여러 처리 및 향상 단계 또는 단계에 걸쳐 배치 또는 스트리밍 소스에서 흐르는 데이터를 포함하는 아키텍처의 상위 수준 보기입니다.

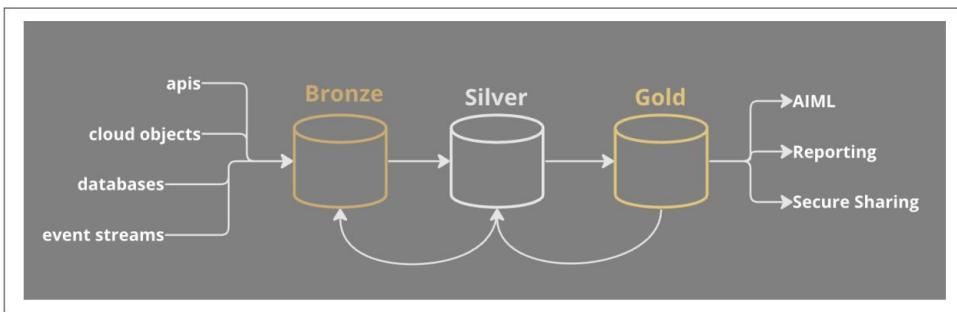


그림 5-3. 메달리온 아키텍처는 수집 시점부터 특정 목적에 맞게 선별된 데이터 제품에 이르기 까지 품질 게이트와 계층을 제공하는 절차적 프레임워크입니다.

메달리온 아키텍처는 구조화된 방식으로 데이터의 점진적인 향상을 처리하기 위한 유연한 프레임워크를 제공합니다. 일반적으로 세 가지 계층이 있지만 모든 사용 사례에 세 가지 계층(브론즈, 실버, 골드)이 필요하다는 규칙은 없다는 점을 지적할 가치가 있습니다. 보다 성숙한 데이터 실무자는 골든 테이블이 다른 골든 테이블과 결합되어 훨씬 더 많은 골든 테이블을 생성하는 2계층 시스템을 갖게 될 수도 있습니다. 그래서 은과 금, 동과 은의 구분이 모호할 때가 있습니다. 3계층 프레임워크를 사용하는 주요 이유는 문제가 발생하거나 요구 사항이 변경될 때 복구하거나 대체할 수 있는 위치를 확보할 수 있다는 것입니다.

### 브론즈 레이어 탐색 브론즈 레이어는

Lakehouse 내 데이터 계보의 초기 지점을 나타냅니다. 여기서 일반적인 관행은 데이터에 최소한의 변환(있는 경우)을 적용하는 것입니다. Delta Lake에 쓰기 위해 소스 형식을 호환 가능한 형식으로 변환하는 등 무시할 수 없는 변환이 있습니다. 최소한의 결과

변환 접근 방식은 추가 사용 사례를 지원하기 위해 이 원시 데이터를 재처리할 수 있는 옵션을 열어 두는 것을 의미합니다.  
다3 , 또는 향후 요구사항이 수정될 수 있습니다.



### 브론즈 레이어는 최소한의 확장을 위한 것입니다. 브론

즈 레이어의 가장 중요한 요구 사항은 쓰기를 위해 소스 데이터를 Delta Lake로 변환하는 것입니다. 최소한의 증강 접근 방식을 취하는 경우 이 초기 수집 단계를 단순화하고 심지어 자동화하는 방법도 모색해 볼 가치가 있습니다. DataFrame API와 상호 운용 가능한 개방형 데이터 프로토콜을 사용하면(예: Apache Avro 또는 Google 프로토콜 버퍼와 같은 유형이 안전한 바이너리 직렬화 가능 교환 형식 사용) 수집보다 더 나은 문제를 해결하는 데 더 많은 시간을 할애할 수 있습니다. 테이블 수가 적을 경우 자동화를 무시하는 것이 논란의 여지가 있지만 표면적 이 증가함에 따라 자동화를 무시하는 것은 엔지니어링 정신 건강에 좋지 않습니다.

### 최소한의 변환 및 확장 가능한 한 "원시"에 가까운 데

이터를 수집하기 때문에 제한된 스키마를 유지하고 데이터 변환을 가능한 한 적게 수행해야 한다는 점을 기억해야 합니다. 구체적인 예를 들어보겠습니다. Kafka와 같은 스트리밍 소스에서 데이터를 읽고 주제 이름, 이진 키 및 값은 물론 각 레코드의 타임스탬프를 캡처하여 Delta Lake 테이블에 쓰려고 한다고 가정해 보겠습니다. 이러한 속성은 모두 Kafka DataFrame 구조에 존재합니다(KafkaSource를 사용하는 경우). API는 Spark 포함이며 [kafka-delta-ingest](#) 로 추출할 수 있습니다. 라이브러리(5장에서 처음 살펴보았음)도 마찬가지입니다.

**예제 5-3** (ch11/notebooks/medallion\_bronze.ipynb)은 최소한의 변형과 확대의 간결한 예입니다.

실시에 5-3. Kafka에서 읽고, 최소한의 변환을 적용하고, 데이터를 Delta에 쓰는 간단한 브론즈 스타일 파이프라인을 보여줍니다.

```
% reader_opts: Dict[str, str] = writer_opts: Dict[str, ...]
str] = bronze_layer_stream =
    ...
    ...

(Spark.readStream.options(**reader_opts).format("kafka").load().select(col("key"), col("값"), col("주
제"), col("타임스탬프")).withColumn("event_date",
    to_date(col("timestamp"))).writeStream.format('delta')
```

---

3 사용자 데이터가 포함된 모든 것은 최종 사용자가 동의한 동의와 데이터 거버넌스 조례 및 표준에 따라 캡처되고 처리되어야 한다는 점을 기억하십시오.

```
.options(**writer_opts) .partitionBy("event_date")
) Streaming_query = bronze_layer.toTable(...)
```

**예제 5-3**에 적용된 극단적인 최소 접근 방식은 데이터를 원시 형식에 최대한 가깝게 보존하는 데 필요한 정보만 사용합니다. 이 기술은 값 열에서 데이터를 추출하고 변환하는 책임을 실버 레이어에 둡니다.

약간의 추가 작업을 수행하는 동안 이 기본 접근 방식을 사용하면 데이터 만료(데이터 손실로 이어질 수 있음)에 대한 걱정 없이 Kafka에서 가져온 원시 데이터를 재처리(다시 읽을)할 수 있는 향후 기능이 가능해집니다. Kafka의 삭제에 대한 대부분의 데이터 보존 정책은 24시간에서 7일 사이입니다.

Postgres와 같은 외부 데이터베이스에서 읽는 경우 최소 스키마는 단순히 테이블 DDL입니다. 데이터베이스의 쓰기 시 스키마 특성을 고려하면 이미 명시적인 보장과 행 전체에 대한 예상 동작이 있으므로 **예제 5-3**에 표시된 예와 비교할 때 실버 레이어에 필요한 작업을 단순화할 수 있습니다.

경험상 데이터 소스에 유형 안전 스키마(avro, protobuf)가 있거나 데이터 소스가 쓰기 시 스키마를 구현하는 경우 일반적으로 Bronze 계층에 필요한 작업이 크게 줄어듭니다. 이는 청동 레이어가 금으로 진행되는 예상치 못한 데이터 또는 손상된 데이터 행을 차단하는 첫 번째 수호자이기 때문에 맹목적으로 익에 직접 쓸 수 있다는 의미는 아닙니다. csv 또는 json 데이터에서 볼 수 있듯이 형식이 안전하지 않은 데이터를 가져오는 경우 브론즈 계층은 손상되거나 문제가 있는 데이터를 제거하는 데 매우 중요합니다.

Spark에서 허용 모드로 Bronze 레이어 보호 **예제 5-4**에서는 Spark를 사용한 허용 통과라는 기술을 보여줍니다. 이 옵션을 사용하면 사전 정의된(일관된) 스키마를 사용하여 게이팅 메커니즘을 추가하여 손상된 데이터를 차단하는 동시에 디버깅을 위해 비준수 행을 보존할 수 있습니다.

#### 예제 5-4. 허용형 패스스루로 잘못된 데이터 방지

```
% from pyspark.sql.types import StructType, StructField, StringType Known_schema:
StructType =
( StructType.fromJson(...) .add(StructField('_corrupt', StringType(), True, {
    'comment': '잘못된 행은 단순히 삭제되는 것이 아니라 _corrupt 상태가 됩니다.'
})))
happy_df =
( Spark.read.options(**{
    "inferSchema": "false",
    "columnNameOfCorruptRecord": "_corrupt", "mode":
    "PERMISSIVE",
```

```
}).schema(알려진_스키
마).json(...)
```

1. StructType.fromJson 메서드를 사용하여 알려진 스키마를 로드하는 것으로 시작합니다. StructType().add(...) 패턴을 사용하여 스키마를 수동으로 쉽게 구축할 수도 있습니다.
2. 그런 다음 \_corrupt 필드를 스키마에 추가합니다. 이는 잘못된 데이터를 보관할 컨테이너를 제공합니다. \_corrupt 열이 null이거나 나 값이 포함되어 있다고 생각하세요. 그런 다음 필터 where(col("\_corrupt").isNotNull()) 또는 역 필터를 사용하여 데이터를 읽어 좋은 것과 나쁜 것을 구분할 수 있습니다.
3. 그런 다음 판독기 옵션 inferSchema:false, mode:Permissive, 열 nNameOfCorruptRecord:\_corrupt를 적용합니다. 스키마 유추를 끄면 업데이트된 스키마를 명시적으로 제공해야만 스키마 변경을 선택할 수 있습니다. 이는 런타임에 예상치 못한 일이 발생하지 않을음을 의미합니다. 스키마 추론은 다수의 반구조화된 데이터(예: csv 또는 json) 행을 스캔(샘플링)하여 안정적인 StructType(스키마)이라고 생각되는 것을 생성하는 강력한 기술입니다. 스키마 추론의 문제점은 데이터의 기록 구조를 이해하지 못하고 초기 배치에서 제공된 내용을 기반으로 가정을 생성하는 것으로 제한된다는 것입니다.

**예제 5-4** 의 기술은 sql.functions 패키지 (pyspark.sql.functions.\* , Spark.sql.functions.\* ) 에 있는 from\_json 기본 함수를 사용하여 쉽게 스트리밍 변환에 적용할 수 있습니다 . 즉, 일괄적으로 테스트한 다음 스트리밍 파일호스를 켜서 일관되지 않은 반구조화된 데이터 세계에서도 수집 파이프라인의 정확한 동작을 이해할 수 있습니다.

## 요약 브론

즈 레이어는 범위와 책임이 제한되어 있다고 느낄 수 있지만 디버깅, 복구 및 향후 새로운 아이디어의 소스로서 매우 중요한 역할을 합니다. 브론즈 레이어 테이블의 원시적 특성으로 인해 이러한 테이블의 가용성을 널리 알리는 것도 바람직하지 않습니다. 원시 테이블의 오용으로 인해 발생하는 문제로 인해 호출을 받거나 사건에 호출되는 것보다 더 나쁜 것은 없습니다.

## 실버 레이어 탐색 메달리온 아키텍

처에서 계보의 초기 지점을 나타내는 브론즈 레이어와 함께 실버 레이어는 원시 데이터가 필터링되고, 정리되고, 정리되고, 심지어 하나 이상의 다른 테이블을 조인하여 확장되는 지점을 나타냅니다. 브론즈 레이어가 유아기의 데이터라면 실버 레이어는 10대 시절의 데이터인데, 우리 모두가 10대였던 것처럼 우리의 데이터 성장 스토리에도 우여곡절이 있습니다.

## 데이터 정리 및 필터링에 사용됨 브론

즈 레이어에 처음 도착한 데이터 소스에 따라 우리는 험난한 상황에 처할 수 있습니다. 두 사람이 완전히 똑같을 수는 없듯이, 각 데이터 소스의 일반적인 일관성과 기준 품질은 크게 다를 수 있습니다. 여기에서 초기 청소 및 필터링이 시작됩니다.

우리는 다운스트림 소비를 위해 신뢰할 수 있는 데이터의 일관된 소스를 정규화하고 제시하기 위해 데이터를 정리합니다. 우리의 다운스트림 소비자는 우리 자신일 수도 있고 조직 내 팀일 수도 있고 심지어 외부 이해관계자일 수도 있습니다. 극단적인 경우에는 Kafka와 같은 스트리밍 소스에서 생성된 이진 데이터를 추출 및 디코딩하여 avro 또는 protobuf에서 변환한 다음 결과 데이터에 추가 변환을 적용할 수 있습니다. 파이프라인의 출력으로 인해 행이 중첩되거나 평면화될 수 있습니다.

이 시점에서 일부 열을 필터링하거나 삭제하는 것도 정상입니다. 예제 5-4 에서는 \_corrupt 열이 포함된 것을 확인했습니다. 이 정보는 메달리온 아키텍처의 은색 또는 황금색 레이어에서 소비하는 데 필요하지 않습니다.

이는 브론즈 레이어의 데이터 보존 기술을 지원하고 엔지니어 간의 커뮤니케이션 형태로만 제공됩니다.

엔지니어가 간단한 정보나 보다 구체적인 구조체나 맵을 포함하는 \_corrupt 또는 \_debug 와 같은 \_\* 열을 제공하는 것은 드문 일이 아닙니다. 이 기술은 보고 목적으로 관찰 가능성 메타데이터 또는 추가 컨텍스트를 전달하는 데에도 사용할 수 있습니다.

예제 5-5 에서는 예제 5-4 에 이어 브론즈 델타 테이블에서 판독값을 수집한 다음 수신할 행을 필터링, 삭제 및 변환하여 정리된 실버 테이블로 변환하는 방법을 보여줍니다.

실시예 5-5. 필터링, 삭제 및 변환. Silver에 글을 쓰는 데 필요한 모든 것.

```
% Medallion_stream =
```

```
( delta_source.readStream.format("delta") .options(**reader_options) .load() .transform(transform_from_json) .transform(transform_
```

예제 5-5 에 표시된 파이프라인은 브론즈 델타 테이블( 예 5-3)에서 읽고, 수신된 이진 데이터(값 열에서)를 디코딩하는 동시에 예제 5-4 에서 살펴본 허용 모드를 활성화합니다. 4.

```
def 변환_from_json(input_df: DataFrame) -> DataFrame:
    return input_df.withColumn("ecommm",
        from_json( col("value").cast(StringType()),known_schema,
            options={ 'mode':
                'PERMISSIVE',
                'columnNameOfCorruptRecord':
                    '_corrupt'
            }
        ))
    ))
```

그런 다음 은색 레이어에 쓰기를 준비하면서 두 번째 변환이 필요합니다. 이는 손상된 행을 제거하고 별칭을 적용하여 이벤트 타임스탬프 및 날짜와 다를 수 있는 수집 데이터 및 타임스탬프를 선언하는 사소한 보조 변환입니다.

```
def 변환_for_silver(input_df: DataFrame) -> DataFrame:
    return
        (input_df.select( col("event_date").alias("ingest_date"),
            col("timestamp").alias("ingest_timestamp"), col("ecommm.*"))
    )
    .where(col("_corrupt").isNull()) .drop("_corrupt"))
```

변환이 처리된 후 데이터를 실버 델타 테이블에 기록합니다. 또한 mergeSchema:false를 명시적으로 설정했습니다. 이는 기본 동작이지만 예상되는 동작이 무엇인지 다른 엔지니어에게 알리고 실수로 발생한 기동이 실수로 동메달에서 은메달로 바뀌지 않도록 하기 때문에 중요한 호출입니다. 우리는 6장에서 ALTER TABLE을 사용한 자동 스키마 진화의 대안을 다루었습니다.

브론즈 데이터를 정리하고 필터링하는 이유에 관계없이 우리 노력의 결과는 이해관계자에게 다양한 사용 사례를 지원하는 보다 일관되고 신뢰할 수 있는 데이터를 제공합니다. 우리는 은총을 메달리온 구조의 첫 번째 안정적인 층으로 간주 할 수 있습니다.

#### 데이터 보강을 위한 레이어 설정 실버 테이블이 브

론즈 테이블에서 읽어야 한다는 규칙은 없습니다. 실제로 은총은 하나 이상의 은 테이블, 심지어는 금 테이블을 결합하는데 사용되는 것이 일반적입니다. 예를 들어, 브론즈 테이블 중 하나를 청소하고 필터링한 결과를 사용하여 여러 추가 사용 사례를 강화할 수 있다면 내부 팀과 외부 파트너의 노동 성과를 재사용하여 시간과 추가적인 복잡성을 모두 절약할 수 있습니다. 브론즈, 실버, 골드 사이의 계보를 시각적으로 볼 수 있으면 테이블과 뷰, 데이터 제품 및 소유자의 수가 시간이 지남에 따라 자연스럽게 증가함에 따라 추가 컨텍스트를 제공하는 데 도움이 될 수 있습니다.

데이터 확인 및 균형을 가능하게 합니다.

Delta는 단순한 스키마 적용만으로는 제공할 수 없는 기능을 향상시키기 위해 열 기반 제약 조건에 대한 기능을 제공합니다. 스키마 적용 및 진화는 6장에서 다루었습니다.

열 수준 제약 조건을 사용하면 CHECK 형식으로 조건자를 적용하여 테이블 수준에서 직접 더 복잡한 규칙을 적용할 수 있습니다.

```
ALTER TABLE <테이블 이름>
  제약조건 추가 <이름>
    CHECK <sql-출야>
```

여기서 장점은 테이블의 데이터가 제약 조건 기준을 결코 충족하지 않는다는 것을 보장할 수 있다는 것입니다. 단점은 행이 제약 조건 검사를 충족하지 않는 경우 DeltaInvariantViolationException이 발생하여 작업이 단락된다는 것입니다.

데이터 품질 프레임워크는 기본 물리적 테이블 정의에서 규칙을 분리하여 테이블 제약 조건을 단순화하는 데 도움이 될 수 있습니다.

오픈 소스 세계에서 인기 있는 프레임워크는 [Great Expectations](#)입니다. [스파크 기대](#), [델타 라이브 테이블 \(DLT\)](#) 기대 – 이는 Databricks의 유료 서비스입니다. 데이터 품질은 DataOps의 중요한 부분이며, 잘못된 데이터가 Medallion Architecture 내의 특정 계층을 떠나기 전에 차단하는 데 도움이 될 수 있습니다.

## 요약 데이

터 엔지니어로서 우리는 소유자처럼 행동하고 데이터 이해관계자에게 우수한 고객 서비스를 제공해야 한다는 점을 기억하십시오. 개선 프로세스 초기에 좋은 품질의 게이트를 설정할 수 있을수록 다운스트림 데이터 소비자가 더 행복해질 것입니다.

## 골드 레이어 탐색 골드 레이어는 메

달라온 아키텍처에서 가장 성숙한 데이터 레이어입니다. 은이 모두 성장하는 길에 있었지만 완전히는 아니었던 것처럼, 금 계층의 데이터는 여러 가지 변형을 거쳐 구체적으로 큐레이팅되었으며 데이터 세계에서 특정 위치를 차지합니다. 이는 골드 레이어의 데이터가 선별되고 명시적인 의도된 목표를 해결하기 위해 만들어졌기 때문입니다. 브론즈가 유아기의 데이터를, 실버가 10대의 데이터를 나타낸다면, 골든 테이블은 30대 후반이나 40대 초반, 혹은 구체적인 정체성이 확립된 시점의 데이터를 의미합니다.

## 높은 신뢰도와 높은 일관성 구축 삶의 다양

한 시점에 있는 사람들의 데이터에 대한 비유는 정확하지 않을 수 있지만 정신 모델 데이터로는 작동합니다. 골든 레이어의 데이터는 우리의 성격이 원하는 것과 같은 방식으로 매일 급격하게 변할 가능성이 훨씬 적습니다.

그리고 나이가 들수록 소망의 변화 속도는 느려집니다. 예제 5-6에서는 실버 레이어 (예 5-5) 의 변환을 통해 상위 N개 보고서를 생성하는 방법을 살펴봅니다 .

예제 5-6. 비즈니스 수준 소비를 위한 의도적인 테이블 생성

```
% pyspark
silver_table = Spark.read.format("delta")... top5 =
    ( silver_table .groupBy("ingest_date",
        "category_id") .agg( count(col("product_id")).alias("impressions" ),
            min(col("price")).alias("min_price"),
            avg(col("price")).alias("avg_price"),
            max(col("price")).alias("max_price"
        ") ) .orderBy(desc("impressions")) .limit(5))

    (top5 .write.format("delta") .mode("overwrite") .options(**view_options) .saveAsTable(f"gold_{topN_products_daily}"))
```

앞의 예에서는 일일 집계를 수행하는 방법을 보여줍니다. 보고 데이터는 골드 레이어에 저장되는 것이 일반적입니다. 이는 우리(및 비즈니스)가 관심을 갖는 데이터입니다.

비즈니스에 중요한 데이터의 가용성, 신뢰성 및 정확성을 보장하기 위해 특별히 제작된 테이블(또는 뷰)을 제공하는 것이 우리의 임무입니다.

기본 테이블과 실제로 비즈니스에 중요한 데이터가 있는 경우 예상치 못한 변경으로 인해 혼란스러운 보고가 발생하고 기계 학습 모델에 대한 부정확한 런타임 추론이 발생할 수 있습니다. 이는 회사에 비용보다 더 많은 비용을 초래할 수 있으며, 경쟁이 치열한 업계에서 고객 유지와 평판 사이의 차이가 될 수 있습니다.

## 요약 골드

레이어는 물리적 테이블이나 가상 테이블(뷰)을 사용하여 구현할 수 있습니다.

이를 통해 뷰를 사용하지 않을 때 전체 물리적 테이블을 생성하고 가상 테이블과 상호 작용할 때 필요한 필터, 열 별칭 또는 조인 기준을 제공하는 간단한 메타데이터를 생성하는 선별된 테이블을 최적화하는 방법을 제공합니다. 성능 요구 사항에 따라 궁극적으로 테이블과 뷰의 사용이 결정되지만, 많은 경우 뷰는 많은 골드 레이어 사용 사례의 요구 사항을 지원하기에 충분합니다.

이제 메달리온 아키텍처를 살펴보았으므로 여정의 마지막 단계는 데이터 수집 시점부터 다운스트림 이해관계자가 데이터를 사용할 수 있게 되는 시점까지 노력과 시간 요구 사항 수준을 줄이기 위한 패턴을 살펴보는 것입니다. 금 가장자리에.

## 스트리밍 메달리온 아키텍처

앞서 우리는 메달리온 아키텍처가 전송 중인 모든 데이터에서 직면하는 일반적인 데이터 문제를 해결할 수 있는 데이터 디자인 패턴이라는 것을 배웠습니다. 문제는 다음과 같습니다.

- 재생 또는 복구 부족(브론즈 레이어로 해결됨) • 손상된 열 수준 기대치(델타 프로토콜을 사용하여 mergeSchema를 끄고 최후의 수단으로 필요하지 않은 경우 overwriteSchema를 무시하여 해결됨).
- 열별 데이터 품질 및 정확성에 문제가 있습니다. 제약 조건을 사용하거나 **Spark-expectations** 와 같은 유ти리티 라이브러리를 사용하여 해결할 수 있습니다. 또는 **델타 라이브 테이블** (@dlt.expect 사용 ).

불완전성을 제거하고, 명시적으로 정의된 스키마를 준수하고, 데이터 확인 및 균형을 제공하기 위해 메달리온 아키텍처를 사용하여 데이터를 정제하는 패턴을 이미 살펴보았지만, 브론즈에서 브론즈로의 변환을 위한 원활한 흐름을 제공하는 방법은 다루지 않았습니다. 은과 은에서 금으로.

시간은 종종 방해가 되는 경향이 있습니다. 시간이 너무 적으면 정보에 입각한 결정을 내릴 수 있는 정보가 충분하지 않으며, 시간이 너무 많으면 안주하고 때로는 약간 게으르게 되는 경향이 있습니다. 특히 레이크하우스를 통과하는 데이터의 종단 간 대기 시간을 줄이는 데 관심이 있는 경우에는 시간이 훨씬 더 중요한 문제입니다. 다음 섹션에서는 엔드투엔드 스트리밍에 초점을 맞춰 메달리온 아키텍처 내 각 계층의 지연 시간을 줄이기 위한 일반적인 패턴을 살펴보겠습니다.

Lakehouse 내에서 종단 간 지연 시간 줄이기 책 전체에서 살펴본 것처럼 Delta 프

로토콜은 테이블에 대한 일괄 액세스 또는 스트리밍 액세스를 모두 지원합니다. 우리는 파이프라인을 배포하여 출력되는 데이터 세트가 품질 표준을 모두 충족하고 데이터의 업스트림 소스를 신뢰할 수 있는 기능을 제공하도록 특정 단계를 수행하여 데이터 수집으로 인한 종단 간 대기 시간을 대폭 줄일 수 있습니다. (청동)에서 (은)을 거쳐 궁극적으로 (금) 계층의 비즈니스 또는 데이터 제품 소유자의 손에 넘어갑니다.

데이터 품질 문제가 더 널리 퍼지기 전에 이를 차단하고 수정하도록 파이프라인을 제작함으로써 예제 5-3부터 예제 5-5 까지 학습한 교훈을 활용하여 엔드투엔드 스트리밍 워크플로를 통합할 수 있습니다.

**그림 5-4는** 스트리밍 작업 흐름의 예를 제공합니다. 예제 5-3에서 본 것처럼 데이터는 Kafka 주제에서 도착합니다. 그런 다음 데이터세트가 브론즈 델타 테이블(ecomm\_raw)에 추가되어 실버 애플리케이션의 증분 변경 사항을 확인할 수 있습니다. 변환을 제공하는 예는 다음과 같습니다.

**실시예 5-5.** 마지막으로 임시 뷰(또는 Databricks의 구체화된 뷰)를 생성 및 교체하거나, ecomm\_silver에서 주기적으로 데이터를 수집하여 목적에 맞는 테이블 또는 뷰를 생성하는 또 다른 골든 애플리케이션을 생성합니다. 예제 5-6에 표시된 패턴을 확장하면 직접 업스트림에서 점진적으로 수집하는 엔드투엔드 파일프라인을 함께 연결하여 초기 시작 지점(kafka)까지 변환 계보를 추적할 수 있습니다.

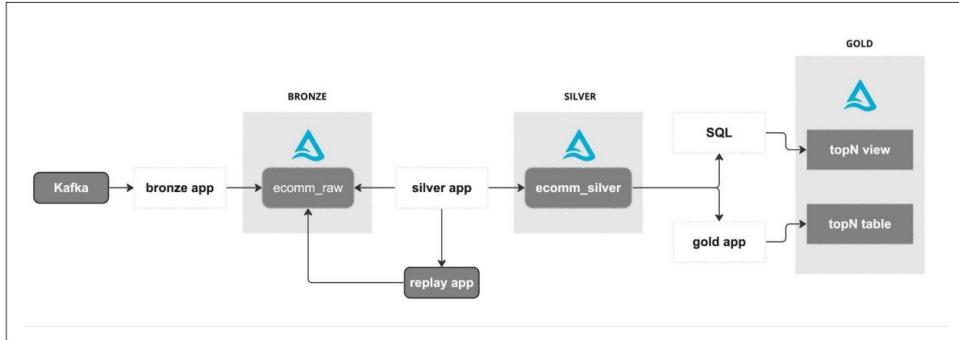


그림 5-4. 워크플로 수준에서 본 스트리밍 메달리온 아키텍처.

예약된 작업이나 Apache Airflow, Databricks Workflows 또는 Delta Live Tables와 같은 본격적인 프레임워크를 사용하여 엔드투엔드 작업 흐름을 조율하는 방법에는 여러 가지가 있습니다. 최종 결과는 에지부터 가장 중요하고 비즈니스에 중요한 골든 테이블에 이르기까지 대기 시간을 줄여줍니다.

## 요약

이 장에서는 현대 Lakehouse 아키텍처의 아키텍처 원칙을 소개하고 이 임무에 대한 기본 지원을 위해 Delta Lake를 사용할 수 있는 방법을 보여주었습니다.

개방형 프로토콜 및 형식을 갖춘 개방형 표준을 기반으로 구축되어 ACID 트랜잭션, 테이블 수준 시간 이동, UniForm과의 단순화된 상호 운용성 및 즉시 사용 가능한 데이터 공유 프로토콜을 지원하여 데이터 교환을 단순화합니다. 내부 및 외부 이해관계자 모두를 위한 것입니다. 우리는 델타 프로토콜의 표면을 훑어보고 쓰기 중 스키마 및 스키마 적용이 다운스트림 데이터 소비자를 우발적인 유출로부터 어떻게 보호하는지 살펴봄으로써 참여 규칙과 테이블 수준 보증을 제공하는 불변성에 대해 더 많이 배웠습니다. 손상되었거나 품질이 낮은 데이터.

그런 다음 메달리온 아키텍처를 사용하여 데이터 품질을 위한 표준 프레임워크를 제공하는 방법과 일반적인 브론즈-온-골드 모델에서 각 레이어가 어떻게 활용되는지 살펴보았습니다.

품질 게이팅 패턴을 통해 일관된 데이터 전략을 구축하고 브론즈(원시)부터 실버(정리 및 정규화), 골드(큐레이트 및 목적 중심)까지 점진적인 품질 모델을 기반으로 보증 및 기대치를 제공할 수 있습니다.

레이크하우스 내, 이러한 게이트 사이의 데이터 흐름 방식은 레이크하우스 내에서 더 높은 수준의 신뢰를 가능하게 하며, 레이크하우스에서 엔드투엔드 스트리밍을 활성화하여 엔드투엔드 대기 시간을 줄일 수도 있습니다.

# 성능 조정: 데이터 최적화

## Delta Lake를 사용한 파이프라인

초기 출시 독자를 위한 참고 사항 초기 출시 eBook을 사용

하면 가장 초기 형태의 책, 즉 저자가 집필하는 그대로의 편집되지 않은 원본 콘텐츠를 얻을 수 있으므로 이러한 타이틀이 공식 출시되기 훨씬 전에 이러한 기술을 활용할 수 있습니다.

이것이 마지막 책의 12번째 장이 될 것이다. GitHub 저장소는 나중에 활성화될 예정입니다.

이 책의 내용 및/또는 예제를 개선할 수 있는 방법에 대한 의견이 있거나 이 장에 누락된 자료가 있는 경우 [gobrien@oreilly.com](mailto:gobrien@oreilly.com)으로 편집자에게 문의하십시오.

지금까지 Delta Lake를 사용하는 다양한 방법을 살펴보았습니다. 지금까지 Delta Lake를 데이터 스토리지 형식으로 더욱 우수하고 안정적인 선택으로 만드는 많은 기능을 살펴보았습니다. 그러나 성능을 위해 Delta Lake 테이블을 조정하려면 6장에서 다른 테이블 유지 관리의 기본 메커니즘에 대한 확실한 이해와 일부 내부 및 고급 기능을 조작하거나 구현하는 약간의 지식과 실습이 필요합니다. 7장과 10장에서 소개되었습니다. 이제 이러한 성능 측면에 초점을 맞춰 일부 기능의 레버를 당길 때의 영향을 좀 더 자세히 설명합니다. 최근에 사용하거나 검토한 적이 없다면 6장에 설명된 주제를 검토하는 것이 좋습니다.

일반적으로 데이터 생성, 소비 및 유지 관리 작업을 수행할 수 있는 안정성과 효율성을 최대화하려는 경우가 많습니다.

데이터 처리 파이프라인에 불필요한 비용을 추가합니다. 시간을 들여 워크로드를 적절하게 최적화함으로써 이러한 작업의 오버헤드 비용과 다양한 성능 고려 사항의 균형을 맞춰 목표에 맞출 수 있습니다. 여기서 얻을 수 있는 것은 이미 본 기능 중 일부를 조정하는 것이 목표 달성을 어떻게 도움이 될 수 있는지 이해하는 것입니다.

첫째, 목표의 성격을 명확하게 하기 위한 몇 가지 배경 작업이 있습니다. 그런 다음 Delta Lake의 여러 기능과 이러한 기능이 이러한 목표에 어떤 영향을 미치는지 살펴봅니다. Delta Lake는 일반적으로 제한된 변경으로 적절하게 사용할 수 있지만 최신 데이터 스택에 대한 요구 사항을 생각하면 항상 더 잘할 수 있다는 것을 깨달아야 합니다. 결국, 성능 튜닝을 수행하려면 균형을 맞추고 절충점을 고려하여 필요한 부분에서 이점을 얻는 것이 필요합니다. 따라서 일부 매개변수 설정을 고려할 때 다른 설정이 어떤 영향을 받는지 확인하고 생각해 보는 것이 가장 좋습니다.

## 성과 목표

고려해야 할 가장 큰 요소 중 하나는 데이터 생산자 또는 소비자에게 가장 적합한 최적화를 시도하고 싶은지 여부입니다. 11장에서 설명한 대로 메달리온 아키텍처는 데이터 큐레이션 레이어를 통해 필요한 경우 읽기 및 쓰기 모두에 최적화할 수 있는 데이터 아키텍처의 한 예입니다. 이러한 프로세스 분리는 파이프라인의 다양한 지점에서 각 목표에 집중함으로써 데이터 생성 시점과 소비 시점의 프로세스를 간소화하는 데 도움이 됩니다. 먼저 튜닝 노력의 방향을 정할 수 있는 다양한 목표 중 일부를 고려해 보겠습니다.

### 읽기 성능 최대화 데이터 소비자를 위한 프로세스

최적화는 데이터 세트의 읽기 성능을 향상시키는 것으로 더 간단하게 생각할 수 있습니다. 정확한 머신 러닝 모델을 구축하기 위해 데이터 세트의 하위 집합을 반복해서 읽는 데이터 과학자가 있을 수도 있고 특정 정보를 도출하여 비즈니스 이해관계자에게 전달하려는 비즈니스 분석가가 있을 수도 있습니다. 프로세스의 설계 및 레이아웃에서는 데이터 소비자의 요구 사항을 고려해야 합니다. 이 섹션에는 요구 사항 수집 또는 ER(엔티티 관계) 다이어그램에 대한 심층적인 내용이 포함되어 있지 않지만, 큐레이션 및 거버넌스가 중앙에서 발생하든, 아니면 더 분산되어 발생하든 관계없이 적절한 데이터 모델링은 성공적인 데이터 플랫폼을 구축하기 위한 높은 가치의 전제 조건입니다. 데이터 메시 아키텍처<sup>1</sup> 여기서 주로 관심을 두는 데이터 소비자 요구 사항은 해당 데이터 소비자가 대부분의 시간 동안 데이터에 액세스하는 방법입니다. 크게 말하면,

---

1 데이터 모델링 및 ER 다이어그램에 대한 자세한 내용을 보려면 SQL 학습, 3판의 부록 A를 확인하세요. Alan Beaulieu 작성 (<https://www.oreilly.com/library/view/learning-sql-3rd/9781492057604/>) 아니면 위키피디아

쿼리는 좁은 지점 쿼리, 더 넓은 범위 쿼리, 집계라는 세 가지 패턴 유형 중 하나에 속합니다.

## 포인트 쿼리

포인트 쿼리는 데이터 소비자 또는 사용자가 데이터 세트에서 단일 레코드를 반환하기 위해 쿼리를 제출하는 쿼리입니다. 예를 들어, 사용자는 사례별로 개별 기록을 조회하기 위해 데이터베이스에 액세스할 수 있습니다. 이러한 사용자는 SQL 기반 조인 논리 또는 고급 필터링 조건과 관련된 고급 쿼리 패턴을 사용할 가능성이 적습니다. 또 다른 예는 사례별로 프로그래밍 방식 및 동적으로 결과를 검색하는 강력한 웹 서버 프로세스입니다. 이러한 쿼리는 **인지된 성능** 지표 와 관련하여 더 높은 수준의 정밀 조사를 통해 평가될 가능성이 더 높습니다. 두 경우 모두 쿼리 성능의 영향을 받는 사람이 반대편에 있기 때문에 높은 비용을 발생시키지 않고 레코드 조회가 자연되는 것을 방지 하려고 합니다. 이는 후자와 같은 일부 경우에는 대기 시간 요구 사항을 충족하기 위해 고성능 전용 트랜잭션 시스템이 필요하다는 것을 의미할 수 있지만, 그렇지 않은 경우가 많으며 여기에 표시된 조정 방법을 통해 목표를 달성 할 수 있습니다. 보조 시스템 없이도 적절하게 작동합니다.

고려해야 할 사항 중 일부는 파일 크기, 키 또는 인덱싱, 분할 전략 등이 포인트 쿼리 성능에 어떤 영향을 미칠 수 있는 것입니다. 경험상 파일 크기를 더 작게 설정하고 건초 더미가 전체 필드인 경우에도 건초 더미에서 바늘을 검색 할 때 대기 시간을 줄이는 인덱스와 같은 기능을 사용하려고 노력해야 합니다. 또한 통계 및 파일 배포가 조회 성능에 어떤 영향을 미치는지 확인할 수 있습니다.

## 범위 쿼리 범위 쿼리

리는 포인트 쿼리(경계가 좁은 특수 사례로 생각할 수 있음)와 같은 단일 레코드 결과 대신 레코드 집합을 검색합니다. 정확한 필터 일치 조건을 갖기보다는 이러한 쿼리가 경계 내의 데이터를 찾는다는 것을 알게 될 것입니다. 그러한 상황을 암시하는 몇 가지 일반적인 문구는 다음과 같습니다.

- 사이
  - 적어도
  - 이전에 • 그
- 런 식으로

다른 많은 것도 가능하지만 일반적인 생각은 많은 레코드가 그러한 조건을 충족할 수 있다는 것입니다(단 하나의 레코드로 마무리하는 것도 여전히 가능하지만). 당신은 것입니다

---

데이터 모델링 페이지 ([https://en.wikipedia.org/wiki/Data\\_modeling](https://en.wikipedia.org/wiki/Data_modeling)) 및 엔터티-관계 모델 ([https://en.wikipedia.org/wiki/Entity%E2%80%93relationship\\_model](https://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model)).

애완동물 종 및 품종 목록에서 동물 유형으로 고양이를 선택하는 것과 같이 광범위한 범주를 설명하는 정확한 일치 기준을 사용할 때 여전히 범위 쿼리가 발생합니다. 당신은 하나의 종만 가질 것이지만 많은 다른 품종을 가질 것입니다. 즉, 얻으려는 결과는 일반적으로 1보다 큽니다. 일반적으로 순서 요소를 추가하고 범위를 추가로 제한하지 않으면 특정 레코드 수를 알 수 없습니다.

### 집계 표면적으

로 집계 쿼리는 특정 레코드 집합까지 선택하는 대신 추가 논리 작업을 사용하여 각 레코드 그룹에 대해 일부 작업을 수행한다는 점을 제외하면 범위 쿼리와 유사합니다. 애완동물의 예를 빌려 종당 품종 수 또는 기타 요약 유형의 정보를 얻고 싶을 수 있습니다. 이러한 경우 카테고리별로 데이터를 분할하거나 세분화된 타임스탬프를 더 큰 기간(예: 연도별)으로 나누는 방식으로 데이터를 분할하는 경우가 종종 있습니다. 집계 쿼리는 범위 쿼리와 동일한 검색 및 필터링 작업을 많이 수행하므로 동일한 종류의 최적화를 통해 유사한 이점을 얻을 수 있습니다.

여기서 찾을 수 있는 것 중 하나는 크기 및 구성 측면에서 파일을 생성하는 방법에 대한 기본 설정이 이러한 사용 유형에 대해 일반적으로 경계를 선택하거나 그룹을 정의하는 방법에 따라 달라진다는 것입니다. 마찬가지로, 인덱싱 및 파티셔닝은 일반적으로 쿼리 패턴에 맞춰 정렬되어 더 높은 성능의 읽기를 생성해야 합니다.

포인트 쿼리, 범위 쿼리, 집계 쿼리의 유사점은 다음과 같이 요약할 수 있습니다. “최고의 성능을 제공하려면 전체 데이터 전략을 데이터가 소비되는 방식에 맞춰야 합니다.” 이는 테이블을 최적화할 때 소비 패턴 외에도 데이터 레이아웃 전략을 고려해야 함을 의미합니다.

그렇게 하려면 데이터를 유지 관리하는 방법과 최적화 또는 통계 수집과 같은 유지 관리 프로세스를 실행하는 것이 성능에 어떤 영향을 미치는지, 필요에 따라 가동 중지 시간을 예약하는 방법도 고려해야 합니다.

### 쓰기 성능 최대화 데이터 생산자를 위한 성

능 최적화는 대기 시간, 즉 레코드 수신(수집)과 이를 사용 가능한 스토리지에 기록(커밋)하는 사이의 시간을 줄이는 것 이상입니다. 일반적으로 이 시간을 최대한 최소화하여 SLA, 성능 목표 및 비용 간의 균형을 유지하려고 하지만 고려해야 할 사항이 더 있습니다. 주로 최적화 목표를 사용되는 쿼리 패턴 종류에 맞춰 데이터 아키텍처에 사용하는 전략을 데이터 소비자가 주도해야 하는 방법에 대해 몇 가지 생각하는 방법을 이미 살펴보았습니다. 또한 기억해야 할 점은 일반적으로 데이터 수신 방법을 정확히 지정할 수 있을 만큼 많은 제어권을 가질 만큼 운이 좋지 않다는 것입니다.

업스트림 데이터 생산자, 즉 데이터를 생성하는 시스템에 의해 구동되는 제약이 있습니다.

비즈니스에 필요한 데이터 자산을 제공하기 위해 다양한 데이터 소스를 함께 결합해야 할 수도 있습니다. 이는 공유 클라우드 스토리지 위치 및 레거시 RDBMS 인스턴스에 자주 업로드되지 않는 파일부터 메모리 저장소 및 대용량 메시지 버스 파이프라인에 이르기까지 다양합니다. 데이터를 수신하는 양과 빈도 등이 데이터 애플리케이션의 성능 요구 사항에 영향을 미치고 전체 데이터 전략에 더 큰 영향을 미치기 때문에 관련 시스템 유형이 많은 의사 결정을 내리게 됩니다.

## 절충안

앞서 언급했듯이 쓰기 프로세스에 대한 많은 제약 조건은 생산자 시스템에 의해 결정됩니다. 대규모 파일 기반 수집, 이벤트 또는 마이크로 배치 수준 스트림 처리를 고려하고 있다면 트랜잭션의 크기와 수가 상당히 달라질 것입니다. 마찬가지로, 단일 노드 Python 애플리케이션으로 작업하거나 대규모 분산 프레임워크를 사용하는 경우 이러한 차이가 발생합니다. 또한 처리에 필요한 시간과 속도도 고려해야 합니다.

이러한 것 중 많은 부분이 균형을 이루어야 하므로 메달리온 아키텍처는 핵심 데이터 생성 프로세스를 브론즈 수준으로 최적화하고 데이터 소비자를 위해 골드 수준으로 최적화하여 이러한 문제 중 일부를 분리할 수 있기 때문에 도움이 됩니다. 은총은 그들 사이에 일종의 다리를 형성합니다. 메달리온 아키텍처를 검토하려면 11장을 다시 참조하세요.

## 갈등 회피

쓰기 작업을 수행하는 빈도는 예를 들어 z 순서 지정을 사용하는 경우와 같이 테이블 유지 관리 작업을 실행할 수 있는 시기를 제한할 수 있습니다. Apache Spark와 함께 구조적 스트리밍을 사용하여 Delta Lake에 대한 마이크로 배치 수준 트랜잭션을 시간별로 분할된 테이블에 쓰는 경우 해당 파티션이 여전히 활성 상태인 동안 해당 파티션에 대해 다른 프로세스를 실행할 때의 영향을 고려해야 합니다(동시성에 대한 자세한 내용 참조). 부록에서 제어). 자동 압축 및 최적화된 쓰기와 같은 옵션을 선택하는 방법은 추가 유지 관리 작업을 실행해야 하는 시기나 실행 여부에도 영향을 미칩니다. 인덱스를 작성하는 데 시간이 걸리고 다른 프로세스와 충돌할 수도 있습니다. 필요할 때 충돌을 피하는 것은 모든 파일 액세스에 관련된 읽기/쓰기 잠금과 같은 작업보다 충돌을 피하는 것이 훨씬 쉽기 때문에 사용자에게 달려 있습니다.

## 성능 고려 사항

지금까지 Delta Lake와 상호 작용하는 방식에 대한 의사 결정의 기반이 되는 몇 가지 기준을 살펴보았습니다. 다양한 도구가 내장되어 있으며 이를 사용하는 방법은 일반적으로 특정 테이블의 상태에 따라 달라집니다.

상호 작용했습니다. 이제 우리의 목표는 당길 수 있는 다양한 레버를 살펴보고 위의 경우에 대해 다양한 매개변수를 설정하는 방식이 어떻게 더 나을 수 있는지 생각하는 것입니다. 이 중 일부는 데이터 생산자/소비자 균형의 맥락에서 6장에서 논의된 개념을 검토합니다.

## 분할 Delta Lake

의 가장 큰 장점 중 하나는 Hive 스타일 분할을 사용하여 데이터를 Parquet 파일처럼 분할할 수 있다는 것입니다.<sup>2</sup> 그러나 이러한 방식으로 테이블을 분할할 수 있는 것도 단점 중 하나입니다. 이 장의 액체 클러스터링). 열 또는 여러 열을 기준으로 델타 테이블을 분할할 수 있습니다.

가장 일반적으로 사용되는 파티션 열은 날짜이지만 대용량 프로세스에서는 시간 및 분 열도 사용하여 여러 수준의 파티셔닝이 있는 테이블을 찾는 것이 일반적입니다. 이는 대부분의 프로세스에 대해 다소 과도한 것이지만 기술적으로는 분할 구조를 얼마나 세밀하게 만들 수 있는지에 제한이 없지만 그렇게 할 경우 위험이 발생할 수 있습니다! 과도하게 분할된 테이블은 성능 저하 측면에서 많은 골칫 거리를 초래할 수 있습니다.

## 구조

파티셔닝이 무엇인지 생각하는 가장 쉬운 방법은 파일 세트를 파티셔닝 열에 연결된 정렬된 디렉터리로 나누는 것입니다. 다음 예와 같이 모든 고객 기록이 "유료" 멤버십 또는 "무료" 멤버십에 속하는 고객 멤버십 범주 열이 있다고 가정합니다. 이 회원 유형 열을 기준으로 분할하면 "유료" 회원 기록이 있는 모든 파일은 하나의 하위 디렉터리에 있고 "무료" 회원 기록이 있는 모든 파일은 두 번째 디렉터리에 있게 됩니다.

```
# Python
from deltalake.writer import write_deltalake import pandas
as pd

df = pd.DataFrame(data=[(1, "고객 1", "무료"), (2, "고객 2", "유료"), (3, "고객 3", "무료"), (4, "고객 4", "결제")],
columns=["id", "이름", "회원_유형"])

write_deltalake( "/tmp/delta/partitioning.example.delta", 데이터=df,
```

---

2 데이터 레이아웃의 Hive 측면에 대한 자세한 내용은 프로그래밍 Hive를 참조하세요 . <https://learning.oreilly.com/library/view/programming-hive/9781449326944/>

```
mode="overwrite",
partition_by=["membership_type"])
```

강제로 파티셔닝을 내리고 membership\_type 열을 기준으로 동시에 파티셔닝하면 쓰기 경로 디렉터리를 확인할 때 Membership\_type 열의 각 고유 값에 대한 하위 디렉터리가 표시되는 것을 볼 수 있습니다.

```
# 배수 트
리 /tmp/delta/partitioning.example.delta

/tmp/delta/partitioning.example.delta |——
 _delta_log |
 00000000000000000000.json |—— 멤버십_유형=
무료 |
   — 0-9bfd1aed-43ce-4201-9ef0-1d6b1a42db8a-0.parquet 멤버십_유형=유료
   — 0-9bfd1aed-43ce-4201-9ef0-1d6b1a42db8a-0.parquet
```

다음 섹션은 테이블을 언제 분할할지, 언제 분할하지 않을지, 그러한 결정이 다른 성능 기능에 미치는 영향을 파악하는 데 도움이 될 수 있지만 테이블을 직접 분할하도록 선택하지 않더라도 더 큰 분할 개념을 이해하는 것이 중요합니다. 파티션을 나눈 테이블의 소유권을 다른 사람으로부터 상속받을 수도 있습니다.

## 함정

Delta Lake의 분할 구조와 관련하여 여기에는 몇 가지 주의 사항이 나와 있습니다(6장의 테이블 분할 규칙을 기억하세요!).

사용해야 하는 실제 파일 크기를 결정하는 것은 테이블을 사용할 데이터 소비자의 종류에 따라 영향을 받지만 파일을 분할하는 방식도 다운스트림 결과에 영향을 미칩니다. 일반적으로 특정 파티션의 총 데이터 양이 1GB 이상인지 확인하고 총 테이블 크기가 1TB 미만인 경우 파티셔닝을 전혀 원하지 않을 것입니다. 그보다 적으면 파일 및 디렉터리 목록 작업 시 불필요한 오버헤드가 많이 발생할 수 있습니다. 특히 클라우드에서 Delta Lake를 사용하는 경우 더욱 그렇습니다.<sup>3</sup> 즉, 카디널리티 열이 높은 경우 대부분의 경우 해당 열을 사용해서는 안 됩니다. 크기 조정이 여전히 적절하지 않은 한 분할 열로 사용됩니다. 분할 구조를 수정해야 하는 경우 6장(델타 레이크 테이블 복구 및 교체)에 설명된 것과 같은 방법을 사용하여 테이블을 보다 최적화된 레이아웃으로 교체해야 합니다. 테이블을 과도하게 파티셔닝하는 것은 시간이 지남에 따라 수많은 사람들에게 성능 문제를 일으키는 것으로 알려진 문제입니다. 성능 저하를 다운스트림으로 전달하는 것보다 시간을 들여 문제를 해결하는 것이 훨씬 낫습니다.

---

<sup>3</sup> 이에 대한 자세한 내용은 Delta Lake 백서(<https://www.databricks.com/wp-content/uploads/2020/08/p975->)에서 확인하세요.  
[암브러스트.pdf](#)

## 파일 크기

과도한 파티셔닝으로 인해 발생하는 직접적인 영향 중 하나는 파일 크기가 너무 작은 경우가 많다는 것입니다. 대규모 데이터 처리를 상대적으로 쉽게 처리하려면 전체 파일 크기 약 1GB가 권장됩니다. 그러나 일반적으로 32~128MB 범위의 더 작은 파일 크기를 활용하면 읽기 작업에 대한 성능 이점을 얻을 수 있는 경우가 많습니다. 둘 중 하나를 선택하는 시기는 데이터 소비자의 특성을 고려하는 것입니다. 브론즈 레이어의 대용량, 추가 전용 테이블은 일반적으로 파일 크기가 클수록 다른 것에 거의 상관하지 않고 작업당 처리량을 최대화하므로 크기가 클수록 더 잘 작동합니다. 크기가 작을수록 포인트 쿼리와 같은 보다 세부적인 읽기 작업이나 생성된 파일 다시 쓰기 횟수가 많아 병합 작업이 많은 경우에 훨씬 더 도움이 됩니다.

결국 파일 크기는 유지 관리 작업을 적용하는 방식에 따라 결정되는 경우가 많습니다. 최적화를 실행할 때, 특히 포함된 z 순서 옵션을 사용하여 실행할 때 결과 파일 크기에 영향을 미치는 것을 볼 수 있습니다.

그러나 파일 크기를 제어하기 위한 몇 가지 기본 옵션이 있습니다.

## 테이블 유ти리티

당신은 아마도 작은 파일 문제의 일부 버전에 대해 꽤 잘 알고 있을 것입니다. 원래는 MapReduce 처리에 큰 영향을 미치는 조건이었지만 문제의 근본적인 특성은 최근의 대규모 분산 처리 시스템에도 확장되었습니다.<sup>4</sup> 6장에서는 Delta Lake 테이블과 일부 테이블을 유지 관리해야 하는 필요성을 확인했습니다. 그것을 할 수 있는 도구 예를 들어, 다른 시나리오 중 일부는 트랜잭션이 더 작은 경향이 있는 스트리밍 사용 사례의 경우 유사한 작은 파일 문제를 피하기 위해 해당 파일을 더 큰 파일로 다시 작성해야 한다는 것입니다. 여기에서는 이러한 도구를 활용하면 Delta Lake와 상호 작용하는 동안 읽기 및 쓰기 성능에 어떤 영향을 미칠 수 있는지 확인할 수 있습니다.

## 최적화 최

적화 작업 자체는 Delta Lake 테이블에 포함된 파일 수를 줄이기 위한 것입니다(6장의 탐색 내용을 떠올려 보세요). 이는 파일을 생성하는 마이크로 배치와 몇 MB 이하의 커밋을 생성하는 스트리밍 워크로드의 경우 특히 그렇습니다. 따라서 상대적으로 작은 파일이 많이 생성될 수 있습니다. 압축은 작은 파일을 함께 압축하는 프로세스를 설명하는 데 사용되는 용어이며 이 작업에 대해 말할 때 자주 사용됩니다.

압축이 성능에 미치는 가장 일반적인 영향 중 하나는 압축 실패입니다. 이러한 작은 파일에는 약간의 이점(예: 세분화된 열 통계)이 있을 수 있지만 일반적으로 많은 파일을 나열하고 여는 데 드는 비용보다 훨씬 큽니다.

---

<sup>4</sup> 익숙하지 않다면 읽어 볼 가치가 있을 것입니다: <https://blog.cloudera.com/the-small-files-problem/>

작동 방식은 최적화를 실행할 때 테이블에서 활성화된 모든 파일과 해당 크기를 나열하는 나열 작업을 시작하는 것입니다. 그러면 결합할 수 있는 모든 파일이 목표 크기인 1GB 정도의 파일로 결합됩니다. 이는 예를 들어 동일한 Delta Lake 대상에 더 작은 트랜잭션을 커밋하는 여러 동시 프로세스에서 발생할 수 있는 문제를 줄이는 데 도움이 됩니다. 즉, 최적화는 작은 파일 문제를 방지하는 데 도움이 되는 메커니즘입니다.

이 작업은 여러 파일을 읽고 이를 최종적으로 기록되는 파일로 결합해야 하므로 상당한 I/O 작업이므로 약간의 오버헤드가 있다는 점을 기억하십시오. 파일 오버헤드를 제거하는 것은 다운스트림 데이터 소비자의 읽기 시간을 개선하는 데 도움이 되는 것의 일부입니다. 9장에서 살펴본 것처럼 최적화된 테이블 다운스트림을 스트리밍 소스로 사용하는 경우 결과 파일은 데이터 변경 파일이 아니므로 무시됩니다.

최적화를 통해 일부 파일 크기 설정을 조정하여 원하는대로 성능을 조정할 수 있다는 점을 기억하는 것이 중요합니다. 이러한 설정과 해당 동작은 6장에서 자세히 다룹니다. 다음으로 z-순서에 대해 자세히 살펴볼 수 있습니다. 이는 기본 개념이 밀접하게 관련되어 있으므로 유동 클러스터링을 사용할 계획인 경우에도 유익합니다.

## Z-순서 파일

을 삽입하거나 작업 중인 데이터를 모델링하는 방식이 일종의 자연스러운 레코드 클러스터링을 제공하는 경우가 있습니다. 고객 거래 기록 등의 파일 하나를 테이블에 삽입하거나 10분마다 비디오 장치에서 재생 이벤트를 집계한다고 가정해 보겠습니다. 그런 다음 한 시간 후에 다시 돌아가 데이터에서 일부 KPI를 계산한다고 가정해 보겠습니다. 얼마나 많은 파일을 읽어야 합니까? 작업 중인 자연 시간 요소(이벤트 또는 트랜잭션 시간을 사용했다고 가정) 때문에 6이라는 것을 이미 알고 있습니다. 데이터가 자연스러운 선형 클러스터링 동작을 갖는 것으로 설명할 수 있습니다. 데이터에 자연적인 정렬 순서가 내재되어 있는 모든 경우에 동일한 설명을 적용할 수 있습니다. 또한 UUID(고유 범용 식별자)를 사용하거나 파일 삽입 시간을 사용하고 필요에 따라 순서를 변경하여 알파벳순으로 데이터 정렬 또는 분할을 인위적으로 만들 수도 있습니다.

그러나 다른 경우에는 데이터가 소비되는 방식에 적합한 기본 클러스터링이 데이터에 없을 수도 있습니다. 추가로 두 번째 범위를 기준으로 정렬하면 상황이 개선될 수 있지만 첫 번째 정렬 범위로 필터링하면 거의 항상 가장 강력한 결과가 나옵니다. 이 추세는 여전히 너무 선형적이기 때문에 추가 열이 추가됨에 따라 가치가 계속해서 감소합니다.

여러 응용 프로그램에서 사용되는 방법이 있는데, 이는 단순한 데이터 응용 프로그램 이상으로 확장되며 이 방법은 공간 채우기를 사용하여 데이터를 다시 매핑하는 방법에 의존합니다.

curve.5 (아직) 엄격한 세부 사항을 너무 많이 설명하지 않고도 이는 여러 열의 값과 같은 다차원 정보를 정렬된 범위의 클러스터 ID와 같은 보다 선형적인 정보로 매핑할 수 있는 구성을 합니다. 좀 더 구체적으로 말하면, 가장 일반적으로 사용되는 Z 순서 또는 힐베르트 곡선과 같은 지역성을 보존하고 공간을 채우는 곡선이 필요합니다.<sup>6</sup> 이를 통해 훨씬 덜 선형적인 스타일로 데이터 클러스터를 생성할 수 있습니다. 특히 세분화된 포인트 쿼리나 보다 복잡한 범위 쿼리의 경우 데이터 소비자에게 성능이 크게 향상됩니다.

즉, 이 다차원 접근 방식은 분리된 조건을 더 쉽게 필터링할 수 있음을 의미합니다. 고객 또는 장치 ID 번호 열과 위치 정보가 있는 경우를 생각해 보세요. 이러한 열에는 특별한 상관 관계가 없으므로 자연스러운 선형 클러스터링 순서가 없습니다. 공간 채우기 곡선을 사용하면 어쨌든 클러스터링 순서를 적용할 수 있습니다. 작동 방식에 대한 자세한 내용을 볼 수 있지만 실용적인 관점에서 이는 전체 데이터 세트를 읽어야 하는 번거로움 없이 결합된 클러스터로 필터링할 수 있음을 의미합니다.

데이터 생산자의 경우 이는 프로세스 속도를 늦추는 데이터 생산의 추가 단계를 나타내므로 다운스트림의 필요성을 미리 결정해야 합니다. 아무도 혜택을 받지 못한다면 그것을 적용하는 데 드는 비용이 들지 않을 것입니다. 즉, 프로세스는 대체로 증분적이며 지정되면 개별 파티션에서 실행될 수 있습니다.

`zorder by`를 사용한 최적화를 통한 압축은 면등성이 아니지만(데이터 변경 플래그가 `False`가 되는 경우 중 하나임) 실행 시 증분되도록 설계되었습니다. 즉, 파티션(또는 파티션되지 않은 테이블의 경우 테이블)에 새 데이터가 추가되지 않으면 해당 파티션이나 테이블을 다시 클러스터링하려고 시도하지 않습니다. 이 동작은 Z 순서 지정에 동일한 열 사양을 사용한다고 예상합니다. 이는 새 열 사양을 사용하려면 전체 파티션(또는 테이블)에 대한 재클러스터링이 필요하기 때문에 의미가 있습니다.

---

5 일반적인 공간 채우기 곡선에 대한 자세한 내용은 [https://en.wikipedia.org/wiki/Space-filling\\_curve](https://en.wikipedia.org/wiki/Space-filling_curve)를 참조하세요. 6 Delta Lake의 초기 구현에 대한 원본 Databricks Engineering 블로그 게시물을 참조하세요. <https://www.databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html>. Z 순서 곡선에 대한 자세한 내용은 [https://en.wikipedia.org/wiki/Z-order\\_curve](https://en.wikipedia.org/wiki/Z-order_curve)를 참조하세요. 힐베르트 곡선에 대한 자세한 내용은 [https://en.wikipedia.org/wiki/Hilbert\\_curve](https://en.wikipedia.org/wiki/Hilbert_curve)를 참조하세요.



Z 순서 지정은 메모리에 비슷한 크기의 클러스터를 생성하려고 시도하며 일반적으로 디스크 크기와 직접적으로 연관되지만 이것이 사실이 아닐 수 있는 상황이 있습니다. 이러한 경우 압축 프로세스 중에 작업 왜곡이 발생할 수 있습니다.

예를 들어 JSON 값이 포함된 문자열 열이 있고 시간이 지남에 따라 이 열의 크기가 크게 증가한 경우 날짜별로 정렬하면 나중에 처리하는 동안 작업 기간과 결과 파일 크기가 모두 왜곡될 수 있습니다.

가장 극단적인 경우를 제외하고 이는 일반적으로 하위 소비자나 프로세스에 큰 영향을 미치지 않아야 합니다.

테이블에서 파일의 z-순서를 사용하거나 사용하지 않고 실험해 보면 알 수 있는 한 가지는 파일 크기 분포가 변경된다는 것입니다. 기본값을 그대로 두면 최적화는 일반적으로 상당히 균일한 크기의 파일을 생성 하지만, 클러스터링 동작을 적용하면 파일 크기가 내장된 파일 크기 제한기(또는 사용 가능한 경우 지정된 제한기)보다 작아지거나 커질 수 있습니다. 엄격한 파일 크기 조정보다 클러스터링 동작을 선호하는 이유는 데이터가 원하는 대로 같은 위치에 배치되도록 하여 최상의 성능을 제공하기 위한 것입니다.<sup>7</sup>

### Spark의 최적화 자동화 Databricks

에서 사용할 수 있는 두 가지 설정, 특히 자동 압축과 최적화된 쓰기는 이러한 테이블 유ти리티 중 일부를 더 쉽게 사용하고 중단을 줄이는 데 도움이 됩니다(예: 스트림 처리 워크로드). 과거에는 이들의 결합된 사용을 종종 자동 최적화라고 불렀습니다. 이제는 함께 사용할 수 있을 뿐만 아니라 다양한 상황에서 필요에 따라 독립적으로 유연하게 사용할 수 있어 개별적으로 처리할 수 있어 큰 이점을 얻을 수 있습니다.

**자동 압축.** 첫 번째 설정인 `delta.autoCompact`는 몇 년 동안 Databricks 런타임에서 사용할 수 있었지만 Delta Lake 전체에서 사용할 수 있을 것으로 예상됩니다. 자동 압축의 개념은 추가 명령 없이 프로세스가 이미 실행 중인 테이블에서 최적화를 실행할 수 있다는 것입니다. 가장 큰 장점 중 하나는 예를 들어 스트림 처리 애플리케이션과 충돌할 수 있는 보조 프로세스를 실행할 필요가 없다는 것입니다. 단점은 처리 대기 시간에 상대적으로 미미한 영향이 있을 수 있다는 것입니다. 이는 파일이 커밋된 후 Spark가 동일한 프로세스의 일부로 최적화 작업을 수행하기 때문입니다.

테이블에서 사용 가능한 파일을 분석하고 필요에 따라 압축을 적용합니다.

이는 트랜잭션이 다른 많은 워크로드에서 볼 수 있는 것보다 작은 경향이 있기 때문에 메시지 버스를 기반으로 한 스트리밍 쓰기에 특히 유용할 수 있습니다.

---

<sup>7</sup> 이 장의 뒷부분에 z-순서 지정에 대한 더 자세한 예가 있지만, 급한 경우에는 다음을 참조하십시오.

훌륭하고 빠른 엔드 투 엔드 연습입니다: <https://dennyglee.com/2024/01/29/optimize-by-clustering-not-partitioning-data-with-delta-lake/>.

유형이지만 처리 시간을 지연시킬 수 있는 압축을 수행하기 위해 추가 작업을 삽입하게 되므로 절충안이 됩니다. 이는 SLA 마진이 좁은 경우에는 사용을 피하고 싶을 수도 있음을 의미합니다.

이 기능을 활성화하는 것은 스파크 구성 설정일 뿐입니다.

`delta.autoCompact.enabled` 참

압축 작업의 동작을 원하는 대로 조정할 수 있는 추가 유연성을 제공하는 몇 가지 추가 설정이 있습니다.



이 기능은 Delta Lake에서 최적화를 사용하는 방식을 개선할 수 있지만 파일에 `zorder`를 포함하는 옵션을 허용하지 않습니다. 다운스트림 데이터 소비자에게 최상의 성능을 제공하기 위해 사용되는 경우에도 추가 프로세스가 필요할 수 있습니다.

`Spark.data bricks.delta.autoCompact.maxFileSize`를 사용하여 자동 압축의 대상 출력 크기를 제어할 수 있습니다. 실제로 기본값인 128MB이면 충분하지만 주기율표 유지 관리 작업 실행 여부와 처리 중 여러 파일을 다시 쓰는 경우의 영향과 파일 크기에 대해 원하는 대상 최종 상태.

압축이 시작되기 전에 필요한 파일 수는 `Spark.databricks.delta.autoCompact.minNumFiles`를 통해 설정됩니다. 기본 숫자는 50입니다.

이렇게 하면 파일 수가 적은 작은 테이블에 대한 추가 작업이 부정적인 영향을 미치지 않도록 임계값을 낮추게 됩니다. 작지만 추가 및 삭제 작업이 많은 테이블의 경우 이 값을 낮게 설정하면 더 적은 수의 파일이 생성되지만 더 작은 크기로 인해 성능에 미치는 영향이 적기 때문에 이점을 얻을 수 있습니다. 단일 트랜잭션에서 Delta Lake에 대한 쓰기 수가 일반적으로 더 높은 대규모 프로세스에는 더 높은 설정이 도움이 될 수 있습니다. 이렇게 하면 각 트랜잭션에 추가된 운영 비용 측면에서 부담이 될 수 있는 모든 쓰기 단계에 대해 최적화 단계를 실행하는 것을 피할 수 있습니다.

최적화된 쓰기. 이 설정 역시 Delta Lake의 Databricks 관련 구현이지만 모든 버전에서 예상됩니다.<sup>8</sup> 과거에는 사용 중인 DataFrame 파티션 수가 파일 수보다 훨씬 더 크게 증가하는 시나리오가 종종 발생할 수 있었습니다. 각 파일의 크기가 너무 작아서 불필요한 오버헤드가 추가로 발생하기 때문에 쓰기를 원합니다. 이 문제를 해결하려면 일반적으로 실제 작업 이전에 `coalesce(n)` 또는 `repartition(n)`과 같은 작업을 수행합니다.

---

<sup>8</sup> <https://docs.databricks.com/delta/tune-file-size.html#optimized-writes-for-delta-lake-on-databricks>

결과를 작성 중인 n 개 파일로 압축하기 위한 쓰기 작업입니다.

최적화된 쓰기는 이러한 작업을 수행할 필요를 방지하는 방법입니다.

테이블에서 delta.optimizeWrites를 true로 설정하거나 Databricks 스파크 세션에서 이와 비슷하게 Spark.databricks.delta.optimizeWrites.enabled를 true로 설정하면 이와 다른 동작이 발생합니다. 후자의 설정은 Spark 세션에서 새로 생성된 모든 테이블에 전자의 옵션 설정을 적용합니다. 이 마법 같은 자동화가 어떻게 배후에서 적용되는지 궁금할 것입니다. 작업의 쓰기 부분이 발생하기 전에 커밋 중에 더 적은 수의 파일을 추가할 수 있도록 메모리 파티션을 결합하기 위해 필요에 따라 추가 셜플 작업을 수행하게 됩니다. 이는 분할로 인해 파일이 더욱 세분화되는 경향이 있으므로 분할된 테이블에 유용합니다. 추가된 셜플 단계는 쓰기 작업에 약간의 대기 시간을 추가할 수 있으므로 데이터 생산자 최적화 시나리오의 경우 건너뛰고 싶을 수도 있지만 쓰기 작업 이후에 발생하지 않고 쓰기 작업 전에 발생한다는 점을 제외하면 위의 autoCompact와 유사한 추가 압축을 자동으로 제공합니다. 그림 6-1은 여러 실행기에 걸쳐 데이터를 분산하면 각 파티션에 여러 파일이 기록되는 경우의 차이점과 추가된 셜플이 배열을 어떻게 개선하는지 보여줍니다.

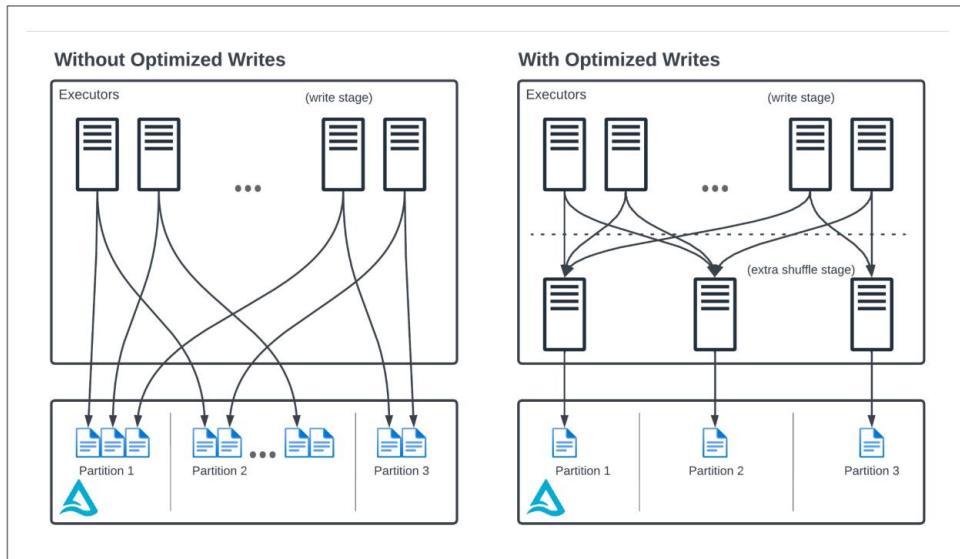


그림 6-1. 최적화된 쓰기가 파일을 쓰기 전에 셜플을 추가하는 방식을 비교합니다.

진공

실패한 쓰기와 같은 사항은 트랜잭션 로그에 커밋되지 않으므로 최적화가 실행 되지 않는 추가 전용 테이블도 진공 처리해야 합니다. 일부 클라우드 제공업체의 오류 또는 다른 이유로 인해 쓰기 오류가 때때로 발생하며 결과 스텝은 여전히 내부에 남아 있습니다.

Delta Lake 디렉터리를 약간 정리하면 됩니다. 일찍 하지 않는 것도 고통을 유발할 수 있는 또 다른 문제입니다. 우리는 프로덕션에서 계획 중에 정리가 간과되어 처리해야 할 더 크고 비용이 많이 드는 일이 되는 꽤 큰 Delta 테이블을 보았습니다. 그 시점에서 수백만 개의 파일을 제거해야 했기 때문입니다(수정하는 데 약 3일이 걸렸습니다). 한 경우에는, 관련된 불필요한 저장 비용 외에도 추가 파일이 포함된 파티션에 도달하는 일부 트랜잭션에는 선별해야 할 파일이 더 많습니다. 매일 또는 매주 정리 작업을 수행하거나 처리 파이프라인에 유지 관리 작업을 포함하는 것이 훨씬 좋습니다. 진공 청소 작업에 대한 자세한 내용은 6장에서 공유되었지만 이를 수행하지 않을 경우 발생하는 결과는 여기서 언급할 가치가 있습니다.<sup>9</sup>

#### Databricks 자동 조정

Databricks에는 활성화된 경우 delta.targetFileSize 설정을 자동으로 조정하는 몇 가지 시나리오가 포함되어 있습니다. 한 가지 사례는 워크로드 유형을 기반으로 하고 두 번째 사례는 테이블 크기를 기반으로 합니다.

DBR 8.2 이상에서는 delta.tuneFileSizesForRewrites가 true로 설정 되면 랜타임은 테이블에 대한 마지막 10개 작업 중 9개가 병합 작업인지 여부를 확인합니다. 그러한 경우 쓰기 효율성을 향상시키기 위해 대상 파일 크기가 줄어듭니다(적어도 일부 추론은 테이블 통계에서 다루게 될 통계 및 파일 건너뛰기와 관련이 있습니다).

DBR 8.4부터는 이 설정을 결정할 때 테이블 크기가 고려됩니다. 약 2.5TB 미만의 테이블의 경우 delta.targetFileSize 설정은 256MB의 낮은 값으로 설정됩니다. 테이블이 10TB보다 큰 경우 대상은 더 큰 1GB로 설정됩니다. 2.5TB에서 10TB 사이의 중간 범위에 속하는 크기의 경우 대상의 규모가 256MB에서 1GB 값까지 선형적으로 증가합니다. [문서를 참고하세요](#) 자세한 내용은 이 척도에 대한 참조 표를 참조하세요.

## 테이블 통계

지금까지는 테이블의 파일 레이아웃과 배포에 중점을 두었습니다. 그 이유는 해당 파일 내 데이터의 기본 배열과 많은 관련이 있습니다. 해당 데이터가 어떻게 보이는지 확인하는 기본 방법은 메타데이터의 파일 통계를 기반으로 합니다. 이제 통계 정보를 얻는 방법과 그것이 왜 중요한지 살펴보겠습니다. 프로세스의 모양, 통계의 모양, 성능에 어떤 영향을 미치는지 확인할 수 있습니다.

---

<sup>9</sup> 여기에 몇 가지 미묘한 차이에 대한 예와 탐색을 통해 진공청소기에 대한 더 자세한 탐색이 있습니다: <https://delta.io/blog/2023-01-03-delta-lake-vacuum-command/>

어떻게

데이터에 대한 통계는 매우 유용할 수 있습니다. 이것이 무엇을 의미하고 어떻게 보이는지에 대해 자세히 알아보겠습니다. 먼저 Delta Lake의 파일에 대한 통계가 필요한 몇 가지 이유에 대해 생각해 보겠습니다. 100개의 가능한 값 중 1개를 사용하는 'color' 필드가 있는 테이블이 있고 각 색상 값은 정확히 100개의 행에 발생한다고 가정합니다. 그러면 총 10,000개의 행이 제공됩니다. 이것이 행 전체에 무작위로 분포되어 있는 경우 모든 '녹색' 레코드를 찾으려면 전체 세트를 스캔해야 합니다. 이제 세트를 10개의 파일로 나누어 세트에 구조를 더 추가한다고 가정해 보겠습니다. 이 경우 10개의 파일 각각에 녹색 레코드가 있다고 추측할 수 있습니다. 10개의 파일을 모두 검사하지 않고 이것이 사실인지 어떻게 알 수 있습니까? 이는 파일에 대한 통계를 확보하려는 동기의 일부입니다. 즉, 파일을 작성할 때 또는 유지 관리 작업의 일부로 일부 계산 작업을 수행하면 테이블 메타데이터에서 특정 값이 발생하는지 여부를 알 수 있습니다. 파일 레코드가 정렬되면 **그림 6-2**에서 볼 수 있듯이 모든 녹색 레코드를 찾거나 50에서 150 사이의 행 번호를 찾기 위해 읽어야 하는 파일 수를 대폭 줄일 수 있기 때문에 이 영향은 더욱 커집니다. . 이 예제는 단지 개념적이지만 테이블 통계가 중요한 이유를 이해하는 데 도움이 됩니다. 하지만 보다 자세한 실제 예제를 살펴보기 전에 먼저 통계가 Delta Lake에서 작동하는 방식을 확인하세요.

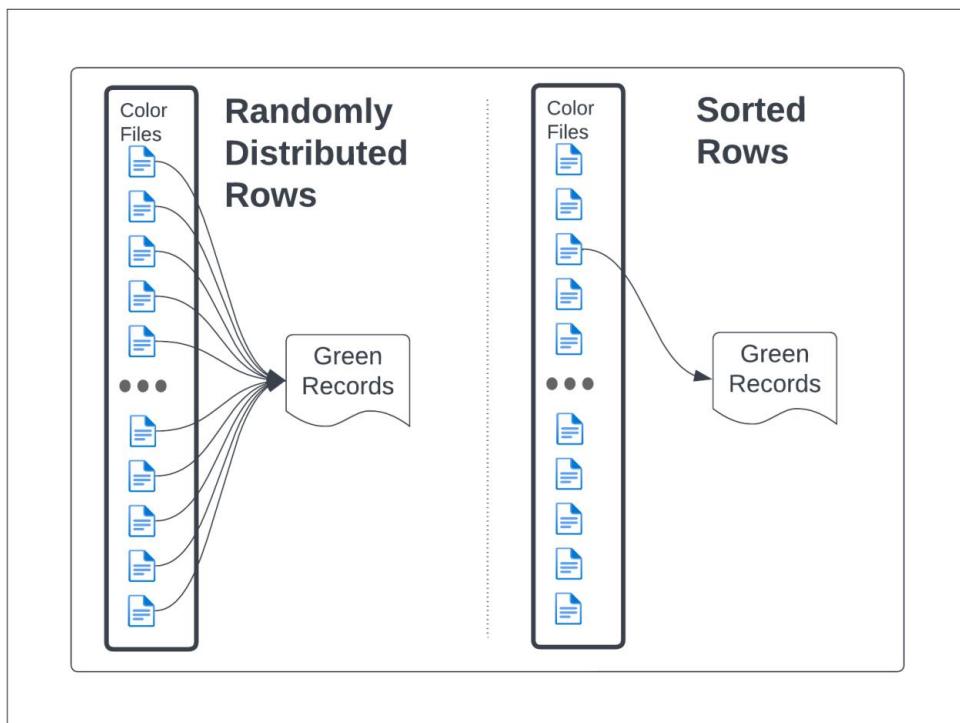


그림 6-2. 데이터 배열은 읽는 파일 수에 영향을 줄 수 있습니다.

## 파일 통계

이전에 생성한 고객 데이터 테이블로 돌아가면 델타 로그를 자세히 살펴보면 파일 생성 중에 통계가 생성되는 방식에 대한 간단한 보기를 얻을 수 있습니다. [Delta Lake 프로토콜](#)의 값이나 관련 섹션을 확인하는 것이 좋습니다. 시간이 지남에 따라 추가되는 추가 통계를 보려면 여기에서 테이블의 경로 정의를 사용한 다음 \_delta\_log 디렉터리에 테이블 생성 시 초기 JSON 레코드를 추가할 수 있습니다.

```
## Python 가
저오기 json

basepath = "/tmp/delta/partitioning.example.delta/" fname = basepath
+ "_delta_log/00000000000000000000.json" with open(fname) as f: for i in
f.readlines():

    구문 분석 = json.loads(i) 구문 분
    섹 키() 에서 '추가' 인 경우 : stats =
        json.loads(parsed['add']['stats']) print(json.dumps(stats))
```

이를 실행하면 Delta Lake 테이블에 추가된 생성된 각 파일에 대해 생성된 통계 컬렉션을 얻을 수 있습니다.

```
{
  "numRecords": 2,
  "minValues": {"id": 2, "name": "고객 2"}, "maxValues": {"id": 4,
  "name": "고객 4"}, "nullCount": {"ID": 0, "이름": 0}

}
{
  "numRecords": 2,
  "minValues": {"id": 1, "name": "고객 1"}, "maxValues": {"id": 3,
  "name": "고객 3"}, "nullCount": {"id": 0, "이름": 0} }
```

이 경우 테이블에 레코드가 4개만 있고 null 값이 삽입되지 않아 해당 카운트가 0으로 반환되므로 모든 데이터 값이 표시됩니다.



분할 데모 테이블에서 가져온 예제 통계에는 각 파일에 대한 레코드 수가 있다는 점에 유의하세요.

Apache Spark는 많은 애플리케이션에서 상당한 성능 이점을 제공하는 데이터 파일을 검색하는 대신 통계를 합산하여 파티션 또는 전체 테이블에 걸쳐 있는 간단한 계산 작업을 실행할 때 실제 데이터 파일을 읽지 않도록 이 수를 활용합니다. 마찬가지로 Spark는 이러한 통계를 활용하여 다음과 같은 유사한 쿼리에 효과적으로 응답할 수 있습니다.

```
## SQL
은 example_table 에서 max(id)를 선택합니다.
```

Databricks(DBR 8.3 이상)에서는 추가로 테이블 분석 명령을 실행하여 고유 값 수, 평균 길이 및 최대 길이와 같은 추가 통계를 수집할 수 있습니다. 이러한 추가된 통계 값은 성능을 더욱 향상시킬 수 있으므로 호환되는 컴퓨팅 엔진을 사용하는 경우 이를 활용해야 합니다.

6장을 떠올려 보면 사용할 수 있는 설정 중 하나는 delta.dataSkippingNumIndexedCols입니다. 기본값은 32이며 통계를 수집할 열 수를 결정합니다. 예를 들어 브론즈에서 실버 레이어 스트림 프로세스와 같이 테이블에 대해 선택 쿼리를 실행할 가능성이 없는 상황에서는 이 값을 줄여 쓰기 작업으로 인한 추가 오버헤드를 방지할 수 있습니다. 또한 더 넓은 테이블에 대한 쿼리 동작이 zorder 기준 보다 훨씬 더 많이 변하는 경우 인덱싱된 열 수를 늘릴 수도 있습니다 (몇 개의 열을 초과하는 것은 일반적으로 그다지 유익하지 않습니다). 여기서 주목해야 할 또 다른 항목은 ALTER TABLE CHANGE COLUMN(FIRST | AFTER)을 사용하여 인덱스된 열 수 뒤에 더 큰 값의 열을 직접 배치하도록 테이블 순서를 변경할 수 있다는 것입니다.<sup>10</sup>

초기 테이블이 생성된 후 추가한 열에 대해 통계가 수집되도록 하려면 첫 번째 매개변수를 사용합니다. 예를 들어, 열 수를 줄이고 긴 텍스트 열을 타임스탬프 열과 같은 항목 뒤로 이동하여 큰 텍스트 열에 대한 통계 수집을 방지하고 여전히 타임스탬프 정보를 포함하여 필터링을 더 잘 활용할 수 있습니다.

각각의 설정은 after 인수에 명명된 열이 필요하다는 점을 제외하면 매우 간단합니다.

```
## SQL

테이블 변경
delta.`example` set
tblproperties("delta.dataSkippingNumIndexedCols"=5);
ALTER TABLE
delta.`example` CHANGE
기사날짜 먼저;
테이블 변경
delta.`example` 수정 후 textCol 변경 Timestamp;
```

파티션 정리 및 데이터 건너뛰기 그렇다면 파

티셔닝을 최적화하고 파일 수준 통계를 수집하는 실제 목표는 무엇입니까?

읽어야 할 데이터의 양을 줄이는 것이 아이디어입니다. 논리적으로 읽기를 건너뛸 수 있는 횟수가 많을수록 쿼리 결과를 더 빨리 검색할 수 있습니다. 표면적인 수준에서 통계 수집을 사용하여 다음을 찾는 방법을 이미 살펴보았습니다.

---

<sup>10</sup> 이 관행을 보여주는 명령별 클러스터를 다루는 섹션에 예제가 있습니다.

실제 파일을 읽지 않고도 열의 최대값을 계산하거나 레코드 수를 계산할 수 있습니다. 이는 해당 작업의 읽기 부분이 파일이 생성될 때 수행되었고 해당 결과를 메타데이터에 저장하면 다시 읽는 데 필요한 모든 오버헤드가 없기 때문에 캐시된 결과에서 기대할 수 있는 결과를 얻을 수 있기 때문입니다. 모든 데이터를 사용하여 결과를 계산합니다. 그러면 좋습니다. 하지만 기록을 세는 것만큼 사소하지 않은 일을 하고 있다면 어떨까요?

차선책은 결과를 검색하기 위해 가능한 한 많은 파일 읽기를 건너뛰는 것입니다. 이러한 통계는 파일별로 수집되므로 회원 자격을 확인하는 데 사용할 수 있는 경계 세트가 제공됩니다. 작은 예제 테이블에 대한 통계를 기억하시나요?

```
{
    "numRecords": 2,
    "minValues": {"id": 2, "name": "고객 2"}, "maxValues": {"id": 4, "name": "고객 4"}, "nullCount": {"ID": 0, "이름": 0}

} {
    "numRecords": 2,
    "minValues": {"id": 1, "name": "고객 1"}, "maxValues": {"id": 3, "name": "고객 3"}, "nullCount": {"ID": 0, "이름": 0}
}
```

고객 1에 대해 포함된 모든 레코드를 가져오려면 사용 가능한 두 파일 중 하나만 읽어야 한다는 것을 쉽게 알 수 있습니다. 이 간단한 경우에만 작업량이 절반으로 줄었습니다. 이는 파일 크기나 파티셔닝에 관해 내릴 수 있는 결정과 같이 이미 본 일부 사항의 영향을 강조하기 시작하고 실제로 더 큰 요점을 통합합니다.

이러한 동작이 존재한다는 것을 알고 있으면 이러한 통계를 활용하여 목표에 따라 성능을 최대화할 수 있는 파티션 레이아웃 및 열 구성을 목표로 삼아야 합니다. 쓰기 성능을 최적화하고 있지만 병합 기능을 사용하여 이전 시점으로 값을 자주 다시 채워야 하는 경우, 낭비되는 데이터를 제거하기 위해 가능한 한 많은 다른 날의 데이터 읽기를 건너뛸 수 있도록 데이터를 구성하고 싶을 것입니다. 처리 시간.

마찬가지로, 읽기 성능을 최대화하고 최종 사용자가 소비 시점에 데이터에 액세스하는 방식을 이해하고 있다면 읽기 시 파일을 건너뛸 수 있는 기회를 최대한 제공하는 대상 레이아웃을 찾을 수 있습니다.

추가 처리 오버헤드로 인해 테이블을 과도하게 분할하는 것에 대한 몇 가지 주의 사항이 있으므로 다음으로 각 파일에 포함된 통계에 대한 지식과 함께 zorder를 사용하여 다운스트림 성능에 영향을 미칠 수 있는 방법을 살펴보겠습니다.

## Z 순서 재검토

파일 건너뛰기는 다양한 종류의 쿼리에 대해 읽어야 하는 파일 수를 줄여 성능을 크게 향상시킵니다. 하지만 "zorder"에서 클러스터링 동작을 추가하면 이 프로세스에 어떤 영향을 미치나요?"라고 질문할 수도 있습니다. 이는 매우 간단합니다. Z 순서는 공간 채우기 곡선을 사용하여 레코드 클러스터를 생성한다는 점을 기억하세요. 이렇게 하면 테이블의 파일이 데이터 클러스터링에 따라 정렬된다는 의미입니다. 이는 파일에 대한 통계가 수집될 때 프로세스에서 레코드 클러스터가 분리되는 방식과 일치하는 경계 정보를 얻을 수 있음을 의미합니다. 따라서 이제 Z 순서 클러스터와 일치하는 레코드를 찾을 때 읽어야 하는 파일 수를 더욱 줄일 수 있습니다.

처음에 데이터의 클러스터가 어떻게 생성되는지 더 궁금할 수도 있습니다. 보다 간단한 경우를 위해 읽기 작업을 최적화하려는 목표를 생각해 보세요.

타임스탬프 열이 있는 데이터 세트가 있다고 가정해 보겠습니다. 명확한 경계가 있는 동일한 크기의 파일을 생성하려는 경우 간단한 대답이 나타납니다. 타임스탬프 열을 기준으로 데이터를 선형적으로 정렬한 다음 동일한 크기의 청크로 나눌 수 있습니다. 하지만 두 개 이상의 열을 사용하고 직접 수행할 수 있는 선형 정렬 대신 키에 따라 실제 클러스터를 생성하려면 어떻게 해야 합니까?

여러 열에 공간 채우기 곡선을 사용하는 고급 작업은 아이디어를 보면 이해하기 그리 나쁘지는 않지만 선형 정렬 사례만큼 간단하지는 않습니다. 적어도 아직은 그렇지 않습니다. 그것은 실제로 아이디어의 일부입니다. 여러 열에 걸쳐 데이터를 비슷한 범위로 분할할 수 있는 방법을 구축하려면 몇 가지 추가 작업을 수행해야 합니다. 이를 위해서는 선형 정렬 사례에서와 마찬가지로 분할 단계를 수행할 수 있도록 여러 차원을 단일 차원으로 변환할 수 있는 매핑 기능이 필요합니다. Delta Lake에서 사용되는 실제 구현은 맥락에 맞지 않게 이해하기가 약간 까다로울 수 있지만 [Delta Lake 리포지토리의 이 코드 조각을 고려해 보세요](#).

```
## Scala 객체
ZOrderClustering 확장 SpaceFillingCurveClustering { override protected[skipping] def
  getClusteringExpression( cols: Seq[Column], numRanges: Int): Column = {
    주장(cols.size >= 1, "0
    개 열을 기준으로 Z 순서 클러스터링을 수행할 수 없습니다. !") val rangeIdCols =
      cols.map(range_partition_id(_, numRanges)) interleave_bits(rangeIdCols: _*).cast(StringType)
  }
}
```

이는 Z 순서 수정자에 전달된 여러 열을 가져온 다음 열 비트를 대체하여 이제 정렬하고 범위로 분할할 수 있는 선형 차원을 제공하는 새 임시 열을 만듭니다. 이제 작동 방식을 알았으므로 이 접근 방식을 보여주는 보다 구체적인 예를 살펴보겠습니다.

### Lead by 예제 이 예

제를 보고 레이아웃의 차이가 z 순서 클러스터링과 관련된 읽어야 하는 파일 수에 어떤 영향을 미칠 수 있는지 확인하세요. 그림 6-3 에는 데이터 파일을 일치시키려는 2차원 배열이 있습니다. x 범위와 y 범위 모두 1부터 9까지 번호가 지정됩니다. 점은 x 값으로 분할되며 x와 y가 모두 5 또는 6인 점을 모두 찾으려고 합니다.

먼저 x=5 또는 x=6 조건과 일치하는 행을 찾습니다. 그런 다음 y=5 또는 y=6 조건과 일치하는 열을 찾습니다. 교차하는 지점이 원하는 목표 값이지만, 파일에 대해 조건이 일치하면 파일 전체를 읽어야 합니다.

따라서 읽은 파일(일치하는 조건이 포함된 파일)의 경우 데이터를 두 가지 범주, 즉 구체적으로 조건과 일치하는 데이터와 어쨌든 읽어야 하는 파일의 추가 데이터로 정렬할 수 있습니다.

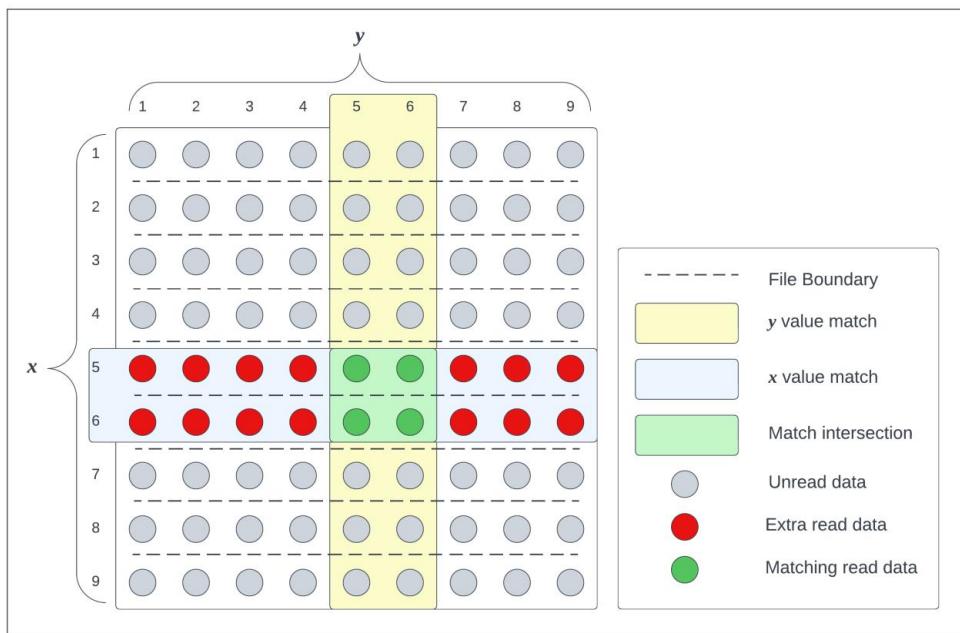


그림 6-3. 파일을 선형 방식으로 배치하면 추가 레코드를 읽게 됩니다.

보시다시피, 일치하는 y 값을 캡처하려면 x=5 또는 x=6인 파일(행) 전체를 읽어야 합니다. 이는 읽기 작업의 거의 80%가 불필요했음을 의미합니다.

이제 대신 공간을 채우는 z 순서 곡선으로 배열되도록 세트를 업데이트하세요. 두 경우 모두 총 9개의 데이터 파일이 있지만 이제 데이터 레이아웃은 메타데이터를 분석하여(파일당 최소/최대 값 확인) 추가 파일을 건너뛰고 불필요한 레코드의 큰 덩어리를 피할 수 있도록 구성되었습니다. 읽고 있는 중.

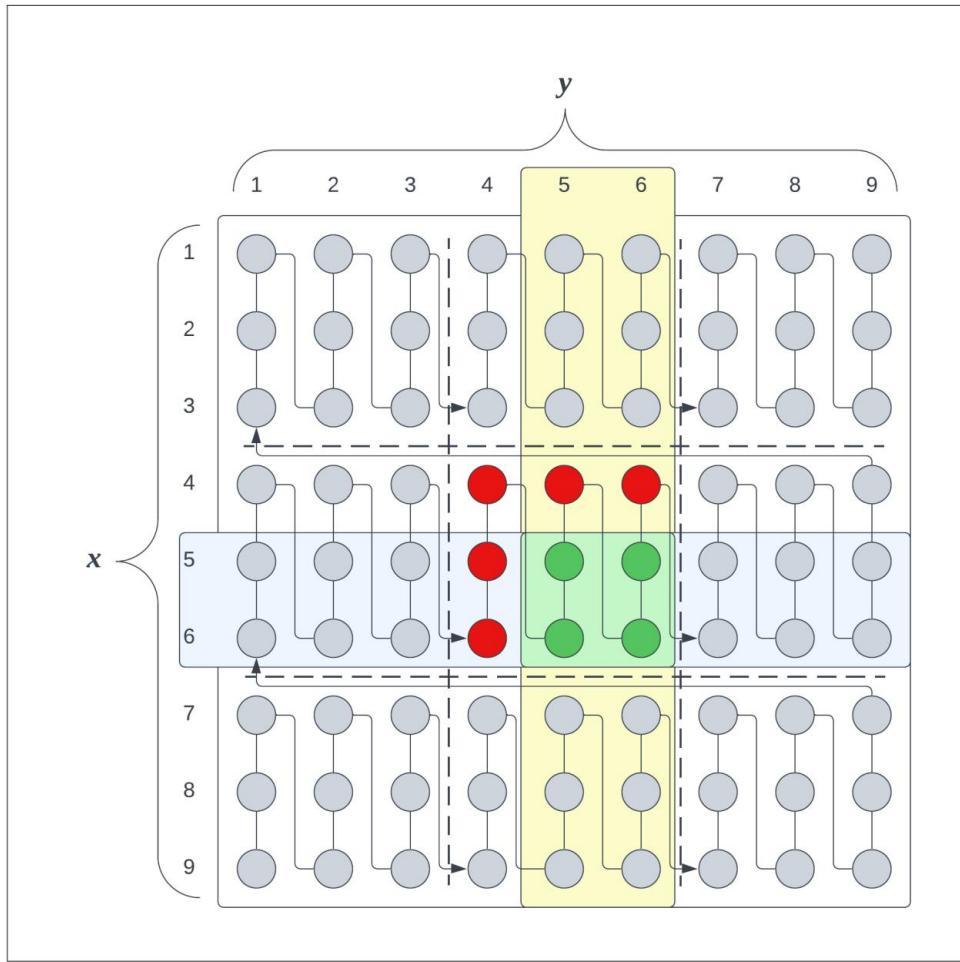


그림 6-4. Z 순서 곡선과 같은 공간 채우기 곡선을 사용하면 작업에 필요한 파일 수와 불필요한 데이터 읽기가 줄어듭니다.

예제에 클러스터링 기술을 적용한 후에는 단일 파일만 읽으면 됩니다. 이것이 부분적으로 z 순서가 최적화 작업과 함께 진행되는 이유입니다. 데이터는 클러스터에 따라 정렬되고 배열되어야 합니다. 이러한 경우에도 데이터가 효율적으로 구성되어 있으므로 데이터를 분할해야 하는지 궁금할 수 있습니다. 예를 들어 유동 클러스터링을 사용하지 않고 동시성 문제가 발생할 수 있는 경우 데이터를 분할하고 싶을 수 있으므로 짧은 대답은 "예"입니다.

데이터가 분할되면 최적화 및 zorder는 동일한 파티션 내에 이미 같은 위치에 있는 데이터만 클러스터링하고 압축합니다. 즉, 클러스터는 단일 파티션 내부의 데이터 범위 내에서만 생성되므로 zorder의 이점은 여전히 올바른 파티션 구성표 선택에 직접적으로 의존합니다.

근접성 또는 클러스터 멤버십을 결정하는 방법은 열 비트를 삽입한 다음 데이터 세트를 범위 분할하는 것에 의존합니다.<sup>11</sup> 다음 단계를 사용하여 이를 수행할 수 있습니다.

1. 좌표 위치를 정수로 포함하는 열을 만듭니다.
2. 이를 이진 값으로 매핑합니다.
3. 이진 값을 비트 단위로 인터리브합니다.
4. 결과 이진 값을 다시 정수로 매핑합니다.
5. 새로운 1차원 열을 범위 분할합니다.
6. 좌표와 빈 식별자를 기준으로 점을 표시합니다.

---

<sup>11</sup> 이 책의 12장 섹션에서 추가 탐색을 장려하기 위해 Python으로 작성된 버전이 있습니다.  
책 보관소.



그림 6-5. z 순서 클러스터를 생성하는 계산 결과를 표시합니다.

결과는 [그림 6-5에 나와 있습니다](#). 매우 깔끔하고 정돈된 [그림 6-4](#) 와 같은 동작을 보여주지는 않지만, 자체 생성 및 직접 계산된 접근 방식을 사용하더라도 데이터 집합에 대한 고유한 zordering을 만들 수 있다는 것을 분명히 보여줍니다. 수학적 관점에서 고려할 수 있는 더 많은 세부 사항과 향상된 기능이 있지만 이 알고리즘은 이미 Delta Lake에 내장되어 있으므로 우리의 건전한 정신을 위해 이것이 현재 우리의 엄격함의 한계입니다.<sup>12</sup>

12 더 자세한 기술 내용은 Mohamed F. Mokbel, Walid G. Aref 및 Ibrahim Kamel을 참조하십시오. 2002. 퍼포 -

다차원 공간 채우기 곡선의 형태. 지리정보시스템 발전에 관한 제10회 ACM 국제 심포지엄 진행 중(GIS '02). 컴퓨팅 기계 협회, 뉴욕, 뉴욕, 미국, 149–154. <https://doi.org/10.1145/585147.585179>

최근에는 z 순서 지정과 같은 아이디어의 추가 개발에 대한 제약을 줄이기 위해 테이블을 분할해야 하는지에 대한 의문이 제기되었습니다. 이는 부분적으로 정적인 프로세스 외부에서는 처음부터 올바른 분할 열을 결정하는 것이 매우 어려울 수 있기 때문입니다. 시간이 지남에 따라 요구 사항이 변경되어 테이블 구조를 업데이트하는 데 유지 관리 작업이 추가될 수도 있습니다(이 작업을 수행해야 하는 경우 예를 참조하세요). 이 분야의 한 가지 발전은 이러한 유지 관리 부담과 결정을 영원히 줄일 수 있습니다.

### Cluster By 파티셔닝

安宁이 끝나면? 그것이 바로 아이디어입니다. 데이터 건너뛰기를 활용하는 가장 최신의 최고 성능 방법은 Delta Lake 3.0에 등장했습니다. 유동 클러스터링은 테이블 생성 중에 매개변수 별 클러스터를 도입하여 전통적인 하이브 스타일 파티셔닝을 대신합니다. zorder 와 마찬가지로 클러스터링은 공간 채우기 곡선을 사용하여 최상의 데이터 레이아웃을 결정하지만 더 많은 효율성을 제공하는 다른 곡선 유형으로 변경합니다. 그림 6-6은 동일한 테이블 구조 내에서 서로 다른 파티션이 어떻게 서로 통합되거나 서로 다른 조합으로 분할될 수 있는지 보여줍니다.

	2023	2022	2021
Customer A		◆	
Customer B		◆	
Enterprise Customer	◆ ◆	◆ ◆ ◆ ◆ ◆	◆
Tiny Customer C			
Tiny Customer D		◆	◆
Customer E			

그림 6-6. 데이터 세트에 액체 클러스터링을 적용한 결과로 생성된 파일 레이아웃 예시<sup>13</sup>

13 이 예는 액체 클러스터링이 더 큰 파티션을 분할하고 더 작은 파티션을 합치기 위해 어떻게 작동하는지 강조하는 전체 연습에서 나온 것입니다. 전체 예는 <https://denny glee.com/2024/02/06/how-> 를 확인하세요. **델타-레이크-액체-클러스터링-개념적으로-작동/**

달라지기 시작하는 부분은 사용 방법에 있습니다. Liquid 클러스터링을 활성화하려면 테이블 생성 중에 선언해야 하며 파티셔닝과 호환되지 않으므로 둘 다 정의할 수 없습니다. 설정되면 테이블에 대해 액체 클러스터링이 적용되는지 확인하는데 사용할 수 있는 테이블 속성인 ClusteringColumns가 생성됩니다. 기능적으로는 어떤 열이 쿼리에 대해 가장 큰 필터링 동작을 생성할 수 있는지 파악하는 데 여전히 도움이 된다는 점에서 zorder by 와 유사하게 작동하므로 여전히 최적화 목표를 염두에 두어야 합니다.

또한 작업은 주로 압축 작업 중에 발생하므로 테이블을 독립적으로 zordering 할 수 없습니다. 언급할 가치가 있는 작은 측면 이점은 설정할 추가 매개변수가 없기 때문에 테이블 세트에 대해 최적화를 실행하는데 필요한 특정 정보를 줄여준다는 것입니다. 따라서 테이블 목록을 반복하여 일치 항목에 대한 걱정 없이 최적화를 실행할 수도 있습니다. 각 테이블에 대한 올바른 클러스터링 키. 또한 파티션 없는 테이블의 필수 기능인 행 수준 동시성을 얻을 수 있습니다. 즉, 쓰기 작업 중에 최적화 도 실행할 수 있으므로 대부분의 경우 서로 프로세스를 예약하려는 시도를 중단하고 가동 중지 시간을 줄일 수 있습니다. 발생하는 유일한 충돌은 두 작업이 동시에 동일한 행을 수정하려고 할 때입니다.

**그림 6-6**에 표시된 것과 같은 파일 클러스터링은 두 가지 다른 방식으로 압축에 적용됩니다. 일반적인 최적화 작업의 경우 레이아웃 배포에 대한 변경 사항을 확인하고 필요한 경우 조정합니다. 이 최신 클러스터링을 사용하면 쓰기 프로세스 중에 데이터를 클러스터링하는 최선의 노력을 통해 훨씬 더 안정적으로 충분 적용할 수 있습니다. 이는 압축 중에 파일을 다시 작성하는 데 필요한 작업이 줄어들어 해당 프로세스가 더욱 효율적이라는 것을 의미합니다. 이 기능을 열성 클러스터링이라고 합니다. 즉, 임계값(기본적으로 512GB) 미만의 데이터의 경우 테이블에 추가된 새 데이터가 쓰기 시 부분적으로 클러스터링됩니다(최선의 노력 부분). 경우에 따라 더 많은 양의 데이터가 누적되고 최적화가 다시 실행될 때까지 이러한 크기는 더 큰 테이블에서 달라집니다. 이는 파일 크기가 여전히 최적화 명령에 의해 결정되기 때문입니다.



인수로 클러스터를 사용하려면 최소한 **작성자 버전이** 필요합니다. 액체 클러스터링 테이블 기능이 존재하고 활성화된 Delta Lake 릴리스의 7개 중 하나입니다. 테이블만 사용하려면 **리더 버전이** 필요합니다. 즉, 환경에 다른/이전 소비자가 있는 경우 최신 버전 및 프로토콜로 마이그레이션하는 동안 워크플로가 중단될 위험이 있습니다.

### 설명 클러스터

터는 zorder 와 다른 공간 채우기 곡선을 사용 하지만 파티션이 없으면 전체 테이블에 클러스터를 생성합니다. 테이블 생성 문의 일부로 인수 별로 클러스터를 포함하기만 하면 사용이 매우 간단합니다. 생성 시 그렇게 해야 합니다. 그렇지 않으면 테이블이 액체와 호환되지 않습니다.

파티셔닝 테이블은 나중에 추가할 수 없습니다. 그러나 나중에 작업을 위해 선택한 열을 업데이트하거나 `alter table` 문을 사용하여 클러스터링에서 모든 열을 제거할 수도 있고 후자의 경우에는 클러스터링을 없음으로 할 수 있습니다(이에 대한 예는 곧 제공됩니다). 이는 클러스터링 키가 필요에 따라 변경될 수 있거나 소비 패턴이 발전함에 따라 변경될 수 있기 때문에 클러스터링 키를 통해 뛰어난 유연성을 얻을 수 있음을 의미합니다.

다운스트림 소비자 또는 쓰기 프로세스에 최적화된 테이블을 생성할 때 둘 사이에서 그러한 결정을 내릴 수 있는 영역이 제공됩니다. 다른 경우와 마찬가지로, 가장 빠른 쓰기 성능을 얻는 것이 목표라면 클러스터링을 전혀 포함하지 않거나 원하는 만큼 적게 포함하도록 선택할 수 있습니다.

다운스트림 소비자에게는 상당한 이점을 얻을 수 있습니다. 주어진 테이블을 다시 분할하는 것이 가능하기는 하지만 이것이 가장 간단한 작업은 아니라는 것을 6장에서 살펴보았습니다. 이제 클러스터링 열을 재정의하여 다운스트림 소비자 요구 사항에 보다 최적으로 적응할 수 있으며 이는 기본 파일에 레이아웃을 적용하기 위해 다음 압축 프로세스 중에 선택됩니다. 즉, 사용 패턴이 변경되거나 원본 레이아웃에서 의심스러운 가정이나 오류가 발생하더라도 더 쉽게 수정할 수 있게 됩니다. 다음 예에서는 Databricks 환경에서 유동 클러스터링을 활용하는 방법을 보여줍니다.



테이블에 대한 초기 쓰기가 10TB보다 큰 경우, 예를 들어 CTAS(Create Table As Select) 문을 사용하여 일회성 변환을 수행하는 경우 첫 번째 압축 작업에서 성능 문제가 발생하고 완료하는 데 시간이 걸릴 수 있습니다. . 클러스터링 품질도 다소 영향을 받을 수 있습니다. 결과적으로 큰 테이블의 경우 프로세스를 일괄적으로 실행하는 것이 좋지만 그렇지 않으면 100TB의 테이블에도 유동 클러스터링이 적용될 수 있습니다.

다행히도 Liquid Clustering은 적합할 때마다 Hive 스타일 파티셔닝 및 Zordering 테이블에 비해 몇 가지 이점을 제공한다는 것이 분명해졌습니다. 잘 조정된 다른 테이블과 유사한 읽기 성능으로 더 빠른 쓰기 작업을 수행할 수 있습니다. 파티셔닝 관련 문제를 방지할 수 있으며, 보다 일관된 파일 크기를 얻을 수 있어 다운스트림 프로세스가 작업 왜곡에 대한 저항력을 더 높일 수 있습니다. 모든 열은 클러스터링 열이 될 수 있으며 필요에 따라 이러한 키를 이동하는 데 훨씬 더 많은 유연성을 얻을 수 있습니다. 마지막으로 행 수준 동시성 덕분에 프로세스와의 충돌이 최소화되어 워크플로가 더욱 동적이고 적응 가능해집니다.

### 예 이 예에

서는 모든 Databricks 작업 영역에서 사용할 수 있는 `/databricks-datasets/` 디렉토리에 있는 Wikipedia 기사 데이터 세트를 볼 수 있습니다. 이 쪽모이 세공 디렉토리에는 거의 1,100개의 gzip 파일에 걸쳐 약 11GB의 데이터(디스크 크기)가 있습니다.

먼저 작업할 DataFrame을 만들고 일반 날짜 열을 세트에 추가한 다음 나중에 SQL에서 작업할 임시 보기 를 만듭니다.

```

## 파일

기사 _ 경로 = (" /databricks-
datasets/wikipedia-datasets / " + " data-001/en_wikipedia/articles-only-
parquet ")

parquetDf = ( 스파

    .read .parquet(articles_path)
)

parquetDf.createOrReplaceTempView("source_view")

```

테이블을 생성하기 위해 읽을 임시 뷰가 있으면 일반 CTAS 문에 인수 별로 클러스터를 추가하여 테이블을 정의할 수 있습니다.

```

## SQL 테
이를 생성
example.wikipages 클러스터
(id) as

( source_view에서 기
사 날짜로 *, 날짜(revisionTimestamp) 선택 )

```

이제 생각해 볼 일반적인 통계 수집 작업이 있으므로 해당 프로세스에서 실제 기사 텍스트를 제외하고 싶을 수도 있지만 클러스터링에 사용하려는 itemDate 열도 만들었습니다. 이를 위해 다음 세 단계를 추가할 수 있습니다. 통계를 수집하는 열 수를 처음 5개로 줄이고, itemDate 및 text 열을 모두 이동한 다음, 마지막으로 열 별로 새 클러스터를 정의합니다. alter table을 사용하면 이 모든 작업을 수행할 수 있습니다.

진술.

```

## SQL
ALTER TABLE example.wikipages는 tblproperties를 설정합니다 ("delta.dataSkippingNumIndexed Cols"=5); ALTER TABLE

example.wikipages CHANGE 기사날짜를 먼저 입력하세요. ALTER TABLE example.wikipages 개정
후 '텍스트' 변경 Timestamp; ALTER TABLE example.wikipages CLUSTER BY (articleDate);

```

이 단계 후에 최적화 명령을 실행하면 다른 모든 것이 자동으로 처리됩니다. 그런 다음 테스트를 위해 다음과 같은 간단한 쿼리를 사용할 수 있습니다.

```

## SQL은

PublishingYear로 연도(articleDate)를 선택하고, 월(articleDate)=3
인 example.wikipages의 기사로 개수(고유 제목)를 선택합니
다.

```

```

day
(articleDate)=4 연도 별 그룹
(articleDate)
계시 연도별 순서

```

전반적으로 프로세스는 쉬웠고 성능은 비슷했지만 zordered Delta Lake 테이블보다 약간 더 빨랐습니다. Liquid Partitioning을 위한 초기 쓰기에도 거의 같은 시간이 걸렸습니다. 배열이 여전히 기본적으로 선형이기 때문에 이러한 결과가 예상됩니다. 그러나 여기서 가치 측면에서 가장 큰 이점 중 하나는 유연성이 추가된다는 것입니다. 어떤 시점에서 원래 정의에서와 같이 id 열을 기준으로 클러스터링으로 되돌리기로 결정한 경우 다른 alter table 문을 실행한 다음 나중에 평소보다 더 큰 최적화 프로세스를 계획하면 됩니다. 유동 클러스터링을 사용하든 익숙한 z 순서를 사용하든 선택한 테이블의 쿼리 성능을 더욱 향상시킬 수 있는 추가 인덱싱 도구가 여전히 있습니다.

## 블룸 필터 지수

블룸 필터 인덱스는 값이 파일에 존재할 가능성이 있는지 또는 확실히 존재하지 않는지 여부를 식별하는 해시맵 인덱스입니다.<sup>14</sup> 해시된 값(단일 행)을 포함하는 인덱스 파일과 관련 파일과 함께 저장되므로 공간 효율적인 것으로 간주됩니다. 데이터 파일이며 색인을 생성하려는 열을 지정할 수 있습니다.

문제는 얼마나 많은 고유 값을 인덱싱해야 하는지에 대한 합리적인 아이디어를 갖고 싶다는 것입니다. 이렇게 하면 해시가 너무 작게 설정된 경우 충돌을 방지하거나 너무 크게 설정된 경우 공간 낭비를 방지하는 데 필요한 해시 길이가 결정되기 때문입니다.

블룸 필터 인덱스는 액체 클러스터링을 사용하는 경우에도 Apache Spark의 Parquet 또는 Delta Lake 테이블에서 사용할 수 있습니다. 런타임 시 Spark는 딕렉토리가 있는지 확인하고 딕렉토리가 있으면 인덱스를 사용합니다. 쿼리 시에는 지정할 필요가 없습니다.

## 더 자세히 살펴

보기 블룸 필터 인덱스는 파일 작성 시 생성되므로 옵션을 사용하려는 경우 고려해야 할 몇 가지 의미가 있습니다. 특히, 모든 데이터를 인덱싱하려면 테이블을 정의한 직후, 테이블에 데이터를 쓰기 전에 인덱스를 정의해야 합니다. 이 부분의 비결은 인덱스를 올바르게 정의하려면 인덱스하려는 모든 열의 고유 값 수를 미리 알아야 한다는 것입니다. 이를 위해서는 약간의 추가 처리 오버헤드가 필요할 수 있지만 예를 들어 count Unique 문을 추가하고 값을 다음의 일부로 가져올 수 있습니다.

---

<sup>14</sup> 블룸 필터 인덱스를 생성하는 데 사용되는 메커니즘과 계산에 대해 더 자세히 알아보고 싶다면 여기에서 시작해 보세요: [https://en.wikipedia.org/wiki/Bloom\\_filter](https://en.wikipedia.org/wiki/Bloom_filter).

메타데이터만 사용하여 이를 수행하는 프로세스입니다(또 다른 Delta Lake 이점). 예를 들어 클러스터의 동일한 테이블을 사용하되 이제 테이블 정의 문 바로 뒤에( 최적화 프로세스를 실행하기 전) 블룸 필터 생성 프로세스를 삽입합니다.

```
## 파일

pyspark.sql.functions에서 import countDistinct

cdf = Spark.table("example.wikipages") raw_items =
cdf.agg(countDistinct(cdf.id)).collect()[0][0] num_items = int(raw_items * 1.25)

Spark.sql(f"""
    블룸필터
    인덱스 생성
    테이블
    example.wikipages 열에 대해
    (id 옵션 (fpp=0.05, numItems={num_items})) """
)
```

여기서 이전에 생성된 테이블이 로드되고 Spark SQL 함수 countDistinct를 가져와 인덱스를 추가하려는 열의 항목 수를 가져올 수 있습니다. 이 숫자가 전체 해시 길이를 결정하므로 이를 채우는 것이 좋습니다. 예를 들어 raw\_items에 1.25를 곱한 경우 num\_items를 얻기 위해 추가로 25%가 추가되어 테이블의 일부 증가를 하용합니다(예상 요구 사항에 따라 조정). 그런 다음 SQL을 사용하여 블룸 필터 인덱스 자체를 정의합니다. 생성 문의 구문은 테이블에 대해 수행하려는 작업을 정확하게 설명하며 매우 간단합니다. 그런 다음 인덱싱할 열을 지정하고 fpp 값 (자세한 내용은 구성 섹션에 있음)과 인덱싱할 수 있는 고유 항목 수(이미 계산된 대로)를 설정합니다.

## 구성 매개변수

의 fpp 값은 거짓양성 확률의 약자입니다. 이 숫자는 읽기 중에 허용되는 잘못된 긍정 비율에 대한 제한을 설정합니다. 값이 낮을수록 인덱스의 정확도가 높아지지만 성능이 약간 저하됩니다. 이는 fpp 값이 각 요소를 저장하는 데 필요한 비트 수를 결정하므로 정확도를 높이면 인덱스 자체의 크기도 커지기 때문입니다.

털 일반적으로 사용되는 구성 옵션인 maxExpectedFpp는 기본적으로 임계값이 1.0으로 설정되어 비활성화됩니다. 간격 [0, 1]에서 다른 값을 설정하면 예상되는 최대 거짓 긍정 확률이 설정됩니다. 계산된 fpp 값이 임계값을 초과하면 필터는 이점보다 사용 비용이 더 많이 드는 것으로 간주되어 디스크에 기록되지 않습니다. 연결된 데이터 파일에 대한 읽기는 인덱스가 남아 있지 않으므로 일반 Spark 작업으로 대체됩니다.

숫자 유형, 날짜/시간 유형, 문자열 및 바이트에 대해 블룸 필터 인덱스를 정의할 수 있지만 중첩 열에서는 사용 할 수 없습니다. 이러한 열에 작동하는 필터링 작업은 `and`, `or`, `in`, `equals` 및 `equalsnullsafe`입니다. 한 가지 추가 제한 사항은 `null` 값이 프로세스에서 인덱싱되지 않으므로 `null` 값과 관련된 필터링 작업에는 여전히 메타데이터 또는 파일 검색이 필요하다는 것입니다.

## 결론

Delta Lake를 사용하여 데이터 테이블 및 파이프라인을 엔지니어링하는 방식을 개선하기 시작 하면 명확한 최적화 목표가 있거나 상충되는 목표가 있을 수 있습니다. 이 장에서는 분할 및 파일 크기가 Delta Lake 테이블에 대해 생성된 통계에 어떤 영향을 미치는지 확인했습니다. 또한 압축 및 공간 채우기 곡선이 이러한 통계에 어떤 영향을 미칠 수 있는지 확인했습니다. 어떤 경우든 Delta Lake 작업 시 사용할 수 있는 다양한 종류의 최적화 도구에 대한 지식을 잘 갖추고 있어야 합니다. 특히, 파일 통계 및 데이터 건너뛰기는 다운스트림 쿼리 성능을 향상시키는 데 가장 유용한 도구일 수 있으며 이러한 통계에 영향을 미치고 모든 상황에 맞게 최적화하는 데 사용할 수 있는 많은 수단이 있습니다. 목표가 어디에 있든 이는 Delta Lake를 사용하여 데이터 프로세스를 평가하고 설계할 때 귀중한 참고 자료가 될 것입니다.

## 제7장

# 성공적인 디자인 패턴

초기 출시 독자를 위한 참고 사항 초기 출시 eBook을 사용

하면 가장 초기 형태의 책, 즉 저자가 집필하는 그대로의 편집되지 않은 원본 콘텐츠를 얻을 수 있으므로 이러한 타이틀이 공식 출시되기 훨씬 전에 이러한 기술을 활용할 수 있습니다.

이것이 마지막 책의 13번째 장이 될 것이다. GitHub 저장소는 나중에 활성화될 예정입니다.

이 책의 내용 및/또는 예제를 개선할 수 있는 방법에 대한 의견이 있거나 이 장에 누락된 자료가 있는 경우 [gobrien@oreilly.com](mailto:gobrien@oreilly.com)으로 편집자에게 문의하십시오.

Delta Lake의 데이터 애플리케이션에 대한 유연성과 적용성을 통해 사용할 수 있는 모든 사례를 포착하려는 것은 종이의 모든 잠재적 용도를 설명하려는 것과 같습니다. 다양성은 무한하며 그 가치는 엄청납니다. 즉, 우리는 Delta Lake의 훌륭한 활용과 그 가치를 강조하는 모범적인 사례를 포착하기 위해 최선을 다합니다.

먼저 Delta Lake의 성능 최적화와 단순화된 유지 관리 작업이 Comcast가 스마트 원격 프로세스를 실행하는 데 필요한 리소스 양을 10배까지 줄이는 데 어떻게 도움이 되었는지부터 보여 드리겠습니다. 그런 다음 Scribd가 어떻게 Delta Lake 환경을 발전시키는 데 도움을 주고 동등한 구조적 스트리밍 애플리케이션보다 100배 더 저렴한 Delta Rust 구현을 만들었는지 설명합니다. 마지막으로 Delta Lake가 대용량 운영 CDC 수집을 제공하고 DoorDash의 Flink에서 실시간 워크로드를 지원하여 다양한 운영 시스템에서 단일 정보 소스 레이크하우스를 만드는 방법을 살펴봅니다. 각 섹션에는

여기에 있는 이야기를 더 자세히 탐색하기 위해 검토할 수 있는 여러 리소스가 있습니다.

## 컴퓨팅 비용 대폭 절감

이 섹션의 초점은 말 그대로 많은 청중에게 다가갑니다! 지난 몇 년 동안 스트리밍 엔터테인먼트 서비스의 수가 어느 정도 폭발적으로 증가했다는 것은 비밀이 아닙니다. 이러한 종류의 서비스를 지원하는 조직은 서비스 지원을 위해 관리해야 하는 대량의 높은 처리량 스트리밍 데이터를 보유하는 경향이 있습니다.

### 고속 솔루션 스트리밍 미디어 서비

스는 일반적으로 여러 가지 구성 요소를 포함하는 개별 최종 사용자 장치에서 데이터를 캡처합니다. 이러한 서비스를 성공적으로 실행하려면 장치 상태, 애플리케이션 상태, 재생 이벤트 정보 및 상호 작용 정보에 대한 다양한 종류의 정보가 필요할 수 있습니다. 이는 일반적으로 처리량이 높은 스트림 처리 애플리케이션 및 솔루션을 구축해야 한다는 의미입니다.

이러한 스트리밍 애플리케이션에서 가장 중요한 구성 요소 중 하나는 안정성과 효율성으로 데이터를 캡처하는 것입니다. 9장에서는 몇 가지 구현 방법과 그 이점을 통해 Delta Lake가 이러한 종류의 데이터 캡처 작업을 수행하는 데 어떻게 중요한 역할을 할 수 있는지 보여줍니다. Delta Lake는 ACID 트랜잭션 보장과 대용량 스트림 처리를 더 좋고 쉽게 만드는 최적화된 쓰기와 같은 추가 기능을 갖추고 있기 때문에 이러한 많은 수집 프로세스의 대상이 되는 경우가 많습니다.

거의 실시간으로 모든 사용자의 QoS(서비스 품질)를 모니터링한다고 가정해 보겠습니다. 이 작업을 수행하려면 일반적으로 재생 이벤트 정보뿐만 아니라 각 사용자 세션의 관련 컨텍스트, 일정 기간 동안 함께 결합된 일련의 상호 작용도 필요합니다. 세션화는 수집을 넘어 많은 다운스트림 작업의 중요한 초석이 되는 경우가 많으며 일반적으로 [그림 7-1](#)에 표시된 것처럼 대규모 데이터 프로세스의 데이터 엔지니어링 단계에 속합니다. Delta Lake의 세션 정보 및 기타 시스템 정보를 사용하면 처리 시간을 짧게 유지하면서 서비스 품질 측정 또는 인기 항목 추천과 같은 다운스트림 분석 사용 사례를 강화할 수 있습니다.

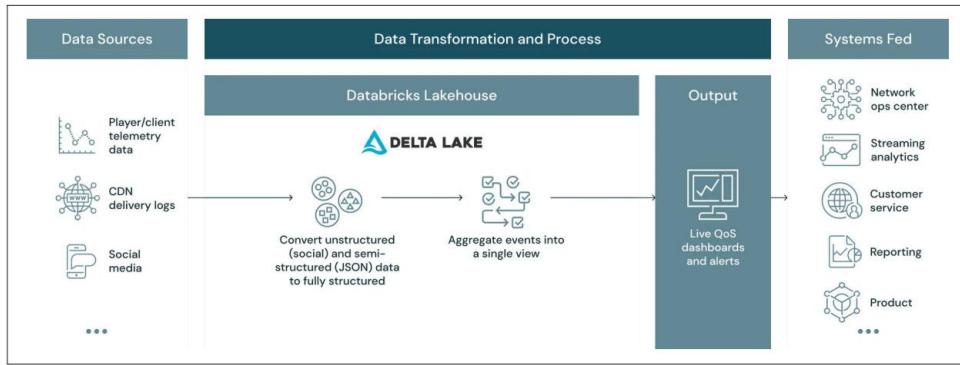


그림 7-1. Delta Lake를 사용한 서비스 품질 모니터링을 위한 참조 아키텍처1

이러한 파이프라인을 구축하는 것은 상당히 복잡한 경우가 많으며 여러 파이프라인과 프로세스의 상호 작용이 필요합니다. 핵심에서는 각 구성 요소가 여러 비즈니스 요구 사항을 충족하기 위해 강력한 데이터 처리 파이프라인을 구축해야 한다는 아이디어로 요약된다는 것을 알게 될 것입니다.

### 스마트 장치 통합 Comcast는

사람들이 TV를 시청하는 방식을 바꾸는 성공적인 스마트 원격 제어 장치를 개발했습니다. 그들이 안고 있는 데이터 문제의 핵심은 이러한 종류의 시스템에는 많은 양의 데이터 처리와 여러 가지 기술 및 조직적 과제가 필요하다는 것입니다. Delta Lake를 데이터 형식으로 사용함으로써 이러한 과제 중 많은 부분이 극복되었으며 가장 중요한 워크로드 중 하나에 대한 클라우드 인프라 요구 사항을 90%까지 줄일 수 있었습니다. 또한 이러한 데이터 프로세스와 관련된 많은 삶의 질 문제를 해결할 수 있었습니다. 여기에서 그들이 이러한 많은 문제를 어떻게 해결했는지 확인할 수 있습니다.

### Comcast의 스마트 리모컨

**컴캐스트 Comcast**는 미국 최대의 다국적 통신 및 미디어 대기업입니다. 여기에서 이 회사가 가장 중요한 워크로드를 실행하는 데 필요한 클라우드 리소스의 양을 어떻게 획기적으로 줄일 수 있었는지 확인할 수 있습니다.<sup>2</sup> Comcast는 다음을 통해 사람들이 TV와 상호 작용하는 방식을 변화시키기 위해 노력해 왔습니다. 중앙 액세스 지점 역할을 하는 보이스 리모콘입니다. 따라서 예상할 수 있듯이 다음이 있습니다.

1 엔드투엔드 QoS 솔루션에 대한 확장된 탐색을 위해 Databricks의 노트북과 함께 이 블로그를 제안합니다: <https://www.databricks.com/blog/2020/05/06/how-to-build-a-service-quality-qos-analysis-solution-streaming-video-service.html>.

2 “AI로 청중 확보: Comcast가 대규모로 민첩한 데이터 및 AI 플랫폼을 구축한 방법 | (컴캐스트)”.

Spark + AI Summit 2019. 2023년 11월 6일에 액세스함 [https://www.youtube.com/watch?v=5sDH\\_dJqoYo](https://www.youtube.com/watch?v=5sDH_dJqoYo)

엣지에 있는 장치를 중심으로 많은 중요한 데이터 워크로드가 발생합니다. 그림 7-2는 상호 작용 흐름의 상위 수준 예를 보여줍니다.

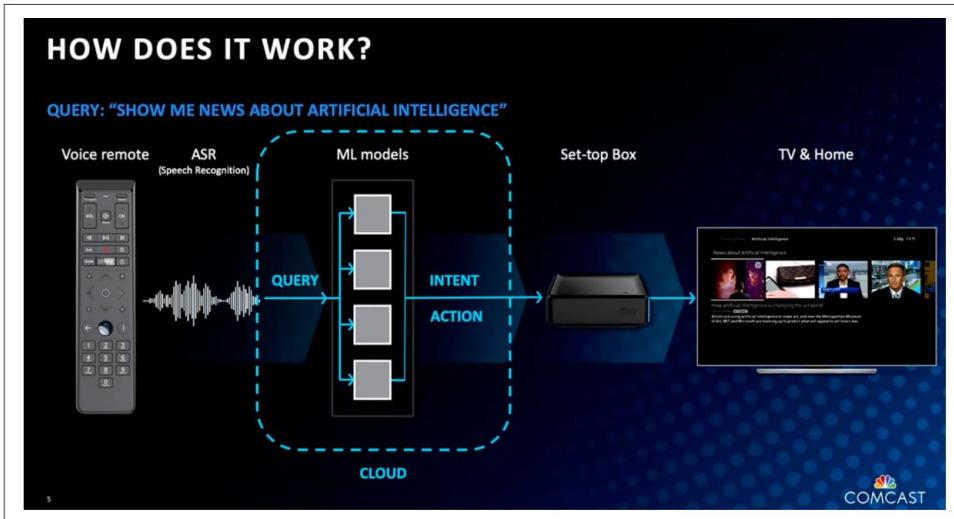


그림 7-2. Comcast의 스마트 리모컨은 엔터테인먼트를 위한 대체 인터페이스를 제공합니다.

Delta Lake에서 솔루션을 구축하는 방법을 탐색하기 전에 운영 규모에 대한 보다 구체적인 정보를 검토하는 것이 유용할 수 있습니다. Comcast는 Xfinity(R) 스마트 리모컨을 통해 상호 작용을 주도하며 고객은 2018~2019년에 이 리모컨을 140억 번 사용했습니다 (그림 7-3은 데이터 처리의 상대적 규모를 보여줍니다). 사용자는 정확한 검색, 소비에 적합한 콘텐츠를 찾을 수 있는 느낌 등 애플리케이션 사용 경험에서 많은 것을 기대합니다. 사용자 경험에는 자신만의 경험을 제공하는 개인화 요소도 있어야 합니다. 음성을 통해 원격 사용자는 전체 시스템과 상호 작용할 수 있습니다. 무엇이든 짧은 문구만 있으면 됩니다. 또한 사용자 데이터를 사용하여 개인화된 경험을 만듭니다.

이러한 서비스를 백그라운드에서 실행하는 데 필수적인 기술 구성 요소를 고려하십시오. 첫째, 음성 명령을 입력으로 받는 것(최근 인기가 폭발적으로 증가한 것)은 기술적으로 어려운 문제입니다. 음성이 디지털 신호로 변환된 다음 필요한 각 명령에 매핑되어야 합니다.

의도 수정 매핑에는 추가 구성 요소가 있는 경우가 많습니다. 누군가가 'How It's Made'라는 프로그램을 검색할 가능성이 더 높습니까? 아니면 특정 제품이 어떻게 만들어지는지에 대한 다른 프로그램에 대해 질문할 가능성이 더 높습니까? 검색 명령이라면 여전히 매칭 알고리즘을 통해 유사한 콘텐츠를 찾아야 합니다.

이 모든 것은 사용자 경험을 정확성에 대해 측정해야 하는 환경에서 단일 인터페이스 지점으로 통합되어 이에 대한 약간의 데이터를 얻습니다.

이러한 프로세스를 수행하고 분석을 통해 즉각적인 문제나 장기적인 추세를 평가하는 것도 중요합니다.

이제 우리는 임베딩 벡터("토큰"으로 의미론적 의미를 캡처하는 숫자 데이터의 벡터)와 상황별 데이터(사용자가 현재 있는 페이지 유형, 기타 최근 검색, 날짜 등)로 변환해야 하는 음성 입력을 갖게 되었습니다. -시간 매개변수 등).<sup>3</sup> 목표는 이 모든 것을 수집하고 거의 실시간으로 사용자 인터페이스(UI)를 통해 추론을 다시 제공하는 것입니다. 기능적 관점에서 보면 장치 상태, 연결 상태, 세션 데이터 보기 및 기타 유사한 문제에 대한 통찰력을 유지하기 위해 수집해야 하는 원격 측정 정보도 많습니다.

개별 장치에서 중앙 처리 플랫폼으로 이 데이터를 가져오는 문제가 해결된 후에도 여러 버전의 장치에 서로 다른 사용 가능한 정보가 있거나 사용 지역에 서로 다른 수집 법률이 있을 수 있으므로 데이터 소스를 표준화하는 방법을 결정하는 데 여전히 추가적인 과제가 있습니다. 캡처된 이벤트의 내용이 더 많거나 적습니다. 표준화의 다운스트림에서는 여전히 데이터를 구성하고 기능에 적합한 형식으로 실행 가능한 단계를 생성해야 합니다.

단일 팀에서 이 모든 일이 일어나기를 기대하려면 엄청난 노력과 오랜 시간이 필요하므로 여러 팀이 협력하여 복잡성을 해결하는 것이 꼭 필요한 것은 아니더라도 도움이 될 것입니다.

### 초기 시도 음성 원격

을 지원하기 위해 Comcast는 쿼리를 분석하고 쿼리 의도 측정과 같은 작업을 수행하기 위한 사용자 여정을 볼 수 있어야 합니다. 초당 최대 1,500만 건의 트랜잭션 속도로 Comcast는 여러 페타바이트 규모의 데이터에 대해 수십억 개의 세션에 걸쳐 세션화를 지원해야 합니다. 기본 AWS 서비스에서 실행하면 세션화에 필요한 규모에 도달하기 위해 결국 640개의 가상 머신에서 32개의 동시 작업 실행을 실행할 때까지 제한을 초과하고 사용 중인 동시성을 증가시켰습니다. 처리 흐름은 [그림 7-3에 나와 있습니다](#). 이로 인해 그들은 확장 가능하고 안정적이며 성능이 뛰어난 솔루션을 찾게 되었습니다.

---

3 임베딩에 대한 보다 강력한 처리에 대해서는 Marcos Garcia를 참조하십시오. 자연어 임베딩 처리: 의미의 벡터 표현에 대한 이론 및 발전. 전산언어학 2021; 47 (3): 699-701. 도이: [https://doi.org/10.1162/coli\\_r\\_00410](https://doi.org/10.1162/coli_r_00410)

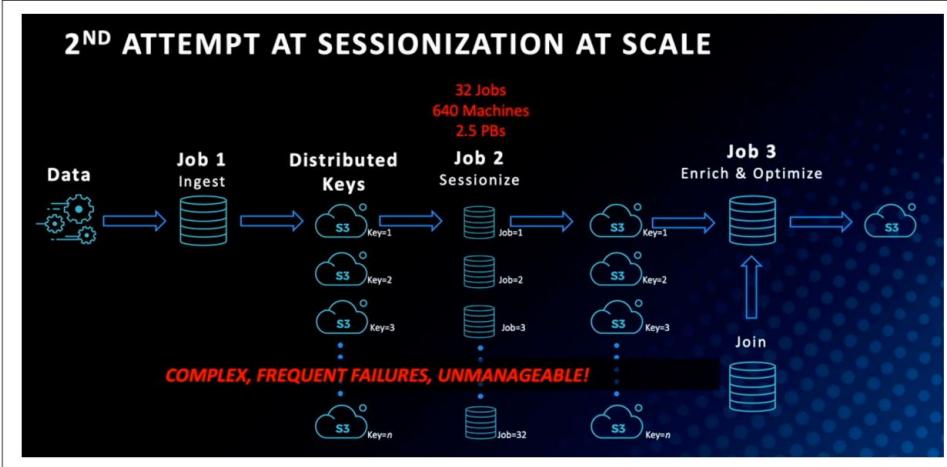


그림 7-3. 초기 데이터 수집 파이프라인을 확장하기 위해 Comcast는 동시성.

Delta Lake는 복잡성을 줄여줍니다.

Delta Lake는 이러한 종류의 문제를 정확하게 해결하는 데 도움을 주기 위해 구축되었습니다. ACID 트랜잭션과 최적화된 쓰기 및 자동 압축과 같은 기능을 갖춘 여러 작성자에 대한 지원은 각각 대규모 스트림 처리 작업과 관련된 문제를 단순화하고 극복하는 역할을 합니다. 클라우드 제공업체의 높은 트랜잭션 속도를 위해 `delta.randomFilePrefixes` 와 같은 추가 기능을 활성화하면 엔지니어로서 최적의 효율성으로 대규모 확장을 달성할 수 있습니다. 이러한 변경을 통해 Comcast는 단 64개의 가상 머신에서 단일 Spark 작업으로 동일한 수집 프로세스를 실행할 수 있었습니다. 결과 프로세스 흐름은 [그림 7-4에 나와 있습니다.](#)

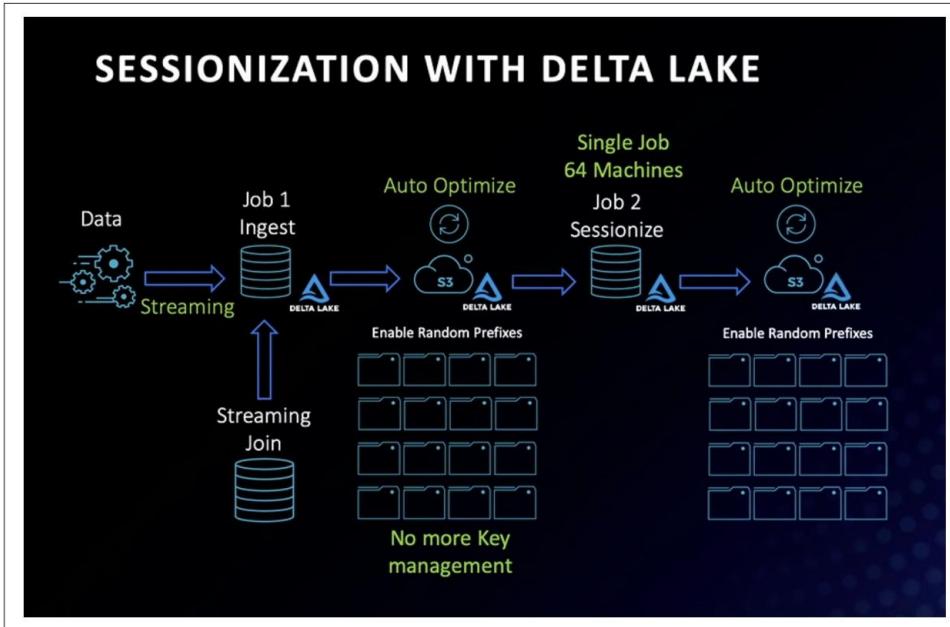


그림 7-4. Delta Lake는 최적화된 수집 및 세션화를 위한 기반을 제공합니다.  
tion.

이것이 전체 이야기라면 아마도 Delta Lake가 처리 부담을 완화하기 위해 가져올 수 있는 가치에 대해 이미 확신하셨을 것입니다. 좋은 점은 그것이 전부가 아니라는 것입니다. Databricks 환경에서 Comcast는 여러 다운스트림 목적으로 이 세션화된 데이터에 쉽게 액세스할 수 있었습니다.

이와 같은 프로세스를 구축하는 데에는 임베딩 벡터 생성이나 모델 추론과 같은 다양한 종류의 기계 학습 작업이 포함될 수 있다고 이미 언급했습니다. 특히 음성 입력을 의미 있는 행동으로 변환해야 할 필요가 있습니다. 세션화된 데이터를 캡처하고 효율적으로 저장함으로써 데이터 과학자는 모델링 파이프라인을 빠르고 쉽게 구축할 수 있습니다.



**ML플로우**, 또 다른 오픈 소스 제품은 엔드투엔드 MLOps 프로세스를 개선하기 위한 많은 기능을 제공합니다. MLflow의 주요 기능 중 일부에는 실험에서 여러 모델 버전을 추적 및 비교하는 기능, 관리용 레지스트리, 모델 개체를 보다 쉽게 배포할 수 있는 메커니즘이 포함됩니다. 여기에는 최근 추가된 기능 중 일부를 통한 LLM(대형 언어 모델)에 대한 특정 지원도 포함됩니다.

Comcast는 MLFlow를 사용하기 때문에 기계 학습 프로세스에서 Delta Lake의 추가적인 이점을 얻습니다. 다음에서 데이터 소스 추적을 사용할 수 있습니다.

프로젝트에 대한 실험 MLflow는 CSV 파일이나 기타 데이터 원본에서와 동일한 방식으로 데이터 복사본을 만들지 않고도 실험에 사용되는 Delta Lake 테이블에 대한 정보를 추적할 수 있습니다.<sup>4</sup> Delta Lake에도 시간이 있기 때문에 여행 기능을 통해 기계 학습 실험은 재현성을 향상시켜 프로덕션에서 데이터 과학 제품을 유지 관리하는 모든 사람에게 도움이 될 수 있습니다.

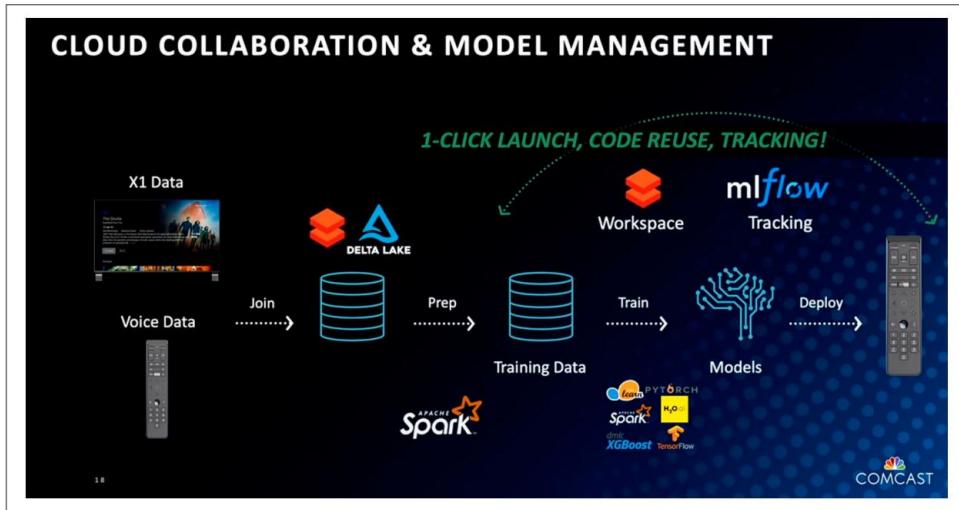


그림 7-5. Delta Lake는 안정적인 엔드투엔드 MLOps 프로세스를 지원합니다.

또 다른 중요한 목표는 QoS 또는 기타 유사한 유형의 분석 애플리케이션과 관련된 원격 측정 데이터를 모니터링할 수 있는 것입니다. Comcast의 경우 Databricks SQL을 사용하여 이전처럼 Redshift 대신 Delta Lake 테이블에서 직접 분석 워크로드를 실행했습니다. 그들은 이 접근 방식의 파일럿을 위해 성능을 평가하기 위해 최악의 쿼리 10개를 선택했다고 보고했습니다. 그들은 쿼리 런타임 대기 시간이 70% 이상 크게 감소한 것을 관찰했습니다.

<sup>4</sup> MLflow 실험에서 다양한 종류의 파일을 추적하는 전체 기능을 비교하기 위해 문서의 이 섹션을 제안하십시오: [https://mlflow.org/docs/latest/python\\_api/mlflow.data.html?highlight=delta#mlflow-data](https://mlflow.org/docs/latest/python_api/mlflow.data.html?highlight=delta#mlflow-data)

## Results

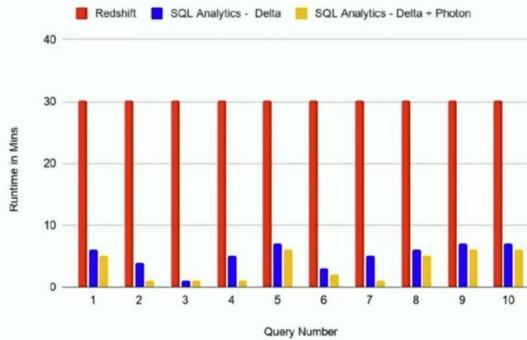


그림 7-6. Delta Lake와 Redshift의 Databricks SQL에서 쿼리 실행 시간에 대한 성능 비교 결과입니다.

결국 Comcast가 Delta Lake를 통해 혁신을 계속하는 것이 매우 유리할 것으로 보입니다. 그들은 지금까지 데이터 수집 프로세스에서 막대한 절감 효과를 경험했으며 보고 개선에 대한 전망이 밝습니다. 전반적으로 이를 통해 스마트 리모컨에 대한 최종 사용자 경험을 더욱 향상하고 전반적인 만족도를 높일 수 있습니다.

## 효율적인 스트리밍 수집

Delta Lake 환경에 공급하기 위해 Kafka 및 Databricks에서 실행되는 대규모 수집 파이프라인이 있다고 가정해 보겠습니다. 이제 Spark의 무거운 작업 기능이 필요하지 않은 소규모 스트림을 위한 솔루션을 제작하여 비용을 절감하기 위해 상당한 노력을 투자하기로 결정한 크랙 엔지니어링 팀이 있다고 가정해 보겠습니다. 또한 수집 프로세스의 모든 데이터를 다운스트림으로 통합하려고 합니다. 그렇다면 당신이 찾고 있는 것은 [Scribd](#)의 팀과 같습니다. 했다.

### 스트리밍 수집 수집 작업을 위한

스트리밍 처리 애플리케이션은 비교적 일반적입니다. 우리는 선택할 수 있는 다양한 스트리밍 프레임워크를 보유하고 있습니다. 가장 일반적인 것 중에는 오픈 소스 Apache Kafka, AWS의 Kinesis, Azure의 Event Hubs 및 Google의 Pub/Sub가 있습니다. 그 이유 중 하나는 스트리밍 데이터 애플리케이션에 대한 일반적인 추세입니다.

IoT 장치의 실시간 원격 측정 모니터링, 사기 거래 모니터링 또는 경고와 같은 흥미로운 주제를 다루는 다양한 적용 가능성이 확실히 있지만 스트리밍 처리의 가장 일반적인 사례 중 하나는 대규모입니다.

및 동적 데이터 수집.5 많은 조직의 경우 모바일 애플리케이션에서 최종 사용자의 활동에 대한 데이터나 소매업체의 POS(Point-of-Sale) 데이터를 수집하는 것은 미션 크리티컬 비즈니스 분석 애플리케이션 지원의 성공으로 직접적으로 해석됩니다. 널리 분산된 소스에서 대량의 데이터를 신속하고 정확하게 수집하면 기업이 변화하는 조건에 보다 신속하게 적응할 수 있습니다 (그림 7-7은 여러 스트리밍 소스에 걸친 통합 아키텍처를 보여줍니다).

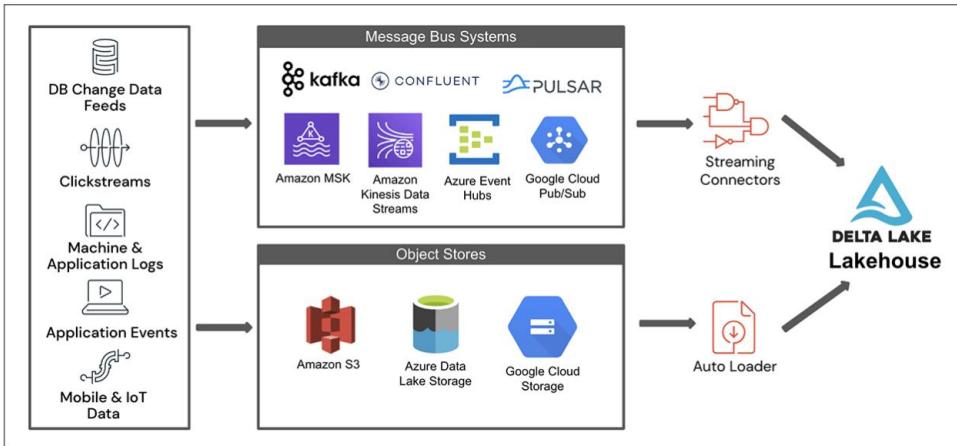


그림 7-7. Databricks의 Delta Lake 싱크를 사용한 스트림 처리 애플리케이션에 대한 참조 아키텍처 다이어그램의 예6

실시간 프로세스의 활성화와 인공 지능 애플리케이션의 사용을 통해 뛰어난 유연성은 종종 이 범주에 속하는 동적이며 탄력적인 데이터 파이프라인에 의해 촉진됩니다.7 이 모든 것에는 일반적으로 다음을 위한 인바운드 데이터를 캡처하는 요소가 있습니다. 나중에 분석 또는 평가 목적으로 사용되므로 최종적으로 일부 처리 파이프라인에 추가 구성 요소가 있을 수 있지만 이 프로세스는 대부분의 스트림 처리 애플리케이션에 적용됩니다.8

5 많은 팀이 Delta Lake에 스트리밍 데이터 소스를 저장하는 과정을 문서화합니다. 예를 들어 Michelin 팀은 Microsoft Azure 환경에서 Kafka+Avro+Spark+Delta Lake 를 구축하기 위한 단계별 구현 가이드를 캡처했습니다. <https://blogit.michelin.io/kafka-to-delta-lake-using-apache-spark-streaming-avro/>

6 아키텍처 다이어그램은 2023년 12월 7일에 액세스한 Databricks 블로그 게시물( <https://www.databricks.com/> )에서 가져온 것입니다. [블로그/2022/09/12/simplifying-streaming-data-ingestion-delta-lake.html](https://blog.databricks.com/simplifying-streaming-data-ingestion-delta-lake.html)

7 여기서 "인공 지능"이라는 용어는 고전적인 소프트웨어 개발 의미로 사용됩니다.

좁은 AI는 기계 학습 알고리즘을 적용하여 사람의 상호 작용 없이 자동화된 비즈니스 결정을 내리는 것을 의미합니다. 예를 들어 <https://hai.stanford.edu/sites/default/files/2023-03/AI-Key-Terms-Glossary-Definition.pdf> 을 참조하세요.

8 스트림 처리 애플리케이션 및 Delta Lake 구현에 대한 자세한 내용은 11장 또는 9장의 메달리온 아키텍처 섹션을 참조하세요 .

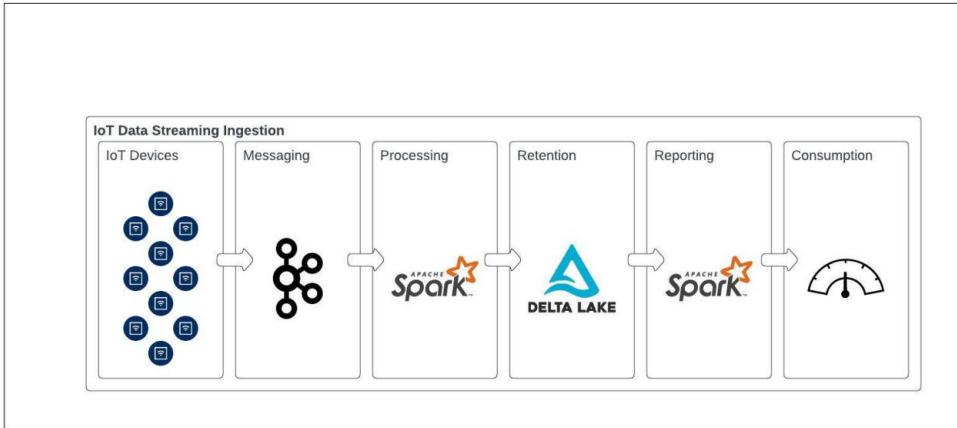


그림 7-8. Kafka 관련 IoT 장치를 위한 단순화된 스트리밍 데이터 수집 아키텍처

장치에서 들어오는 IoT 데이터의 경우를 생각해 보세요. 모든 데이터를 Kafka로 보내는 경우 Spark 애플리케이션을 구축하여 해당 스트림을 사용하고 메탈리온 아키텍처 모델에 따라 수신된 모든 원본 데이터를 캡처할 수 있습니다. 그런 다음 비즈니스 수준 보고를 생성하고 해당 결과를 다운스트림 애플리케이션에서 사용하도록 보낼 수 있습니다. 당연히 이 접근 방식에는 다양한 변형이 있지만 일반적인 파이프라인 모델은 유사합니다. Scribd에서는 이 애플리케이션이 너무 흔해서 이 프로세스 구현을 중심으로 새로운 프레임워크를 구축했습니다.

#### Delta Rust의 시작 개방형 출판 플랫

폼으로 시작했지만 이제는 **Scribd** 150개 이상의 카테고리에 걸쳐 1억 7천만 개가 넘는 문서를 보유하고 있는 디지털 문서 라이브러리입니다. 그들의 임무 중 하나는 세상이 읽는 방식을 바꾸는 것입니다. 그들은 창작자와 소비자 모두에게 공정한 가격으로 다양한 독서 자료를 제공하고, 창작자를 위한 지적 재산권 보호를 제공하며, 비용을 낮게 유지하고, 광고보다는 커뮤니티에 브랜드 구축을 선호함으로써 그렇게 하는 것을 목표로 합니다.

Scribd는 디지털 도서관이라는 존재 자체로 웹사이트와 모바일 애플리케이션을 운영하고 있습니다. 사용자는 Scribd의 웹사이트와 모바일 애플리케이션을 사용하여 수백만 개의 프레젠테이션, 연구 논문, 템플릿 및 기타 다양한 종류의 문서가 포함된 디지털 라이브러리를 탐색할 수 있습니다. 라이브러리의 모든 문서는 작성자, 작성자 및 편집자가 .pdf, .txt, .doc, .ppt, .xls 및 .docx와 같은 여러 공통 문서 형식을 사용하여 업로드합니다. 구독제도도 있습니다. 이러한 다양한 시스템 구성 요소는 모두 그에 따라 수집하고 처리해야 하는 이벤트로 변환됩니다. Scribd에서는 Kafka를 통해 상당히 많은 수의 이벤트 스트림을 사용하여 이를 수행합니다.

스트리밍 수집 파이프라인을 구축하려면 일반적으로 여러 구성 요소가 필요합니다.

이를 즉각적인 맥락에 적용하면 Kafka에서 나오는 각 주제 스트림에 대한 스트림 처리 애플리케이션을 구축하는 것이 간단한 설계 접근 방식이 될 것입니다. Scribd의 경우 작성자 업로드, 읽기 이벤트, 시스템 로그인 또는 인증 이벤트, 구독 이벤트, 웹 트래픽 이벤트, 검색, 항목 북마크 또는 저장 이벤트 등 몇 가지 가능한 이벤트 주제 스트림 목록을 쉽게 구축할 수 있습니다. 그리고 아이템 공유 이벤트도 진행됩니다. 이는 다양한 스트림 처리 애플리케이션이 관련되어 일반적으로 모든 애플리케이션에서 개발 및 유지 관리 오버헤드를 줄이기 위한 일종의 프레임워크 개발로 이어진다는 것을 의미합니다.

많은 이벤트 스트림에 대한 스트림 처리 프레임워크를 유지 관리하는 것은 매우 복잡한 작업일 수 있으며 신중한 계획이 없으면 비용도 많이 듭니다. 다음은 [kafka-delta-ingest](#) 생성으로 이어지는 Scribd의 스트림 처리 프레임워크의 진화에 대한 이야기입니다. 라이브러리와 수집 비용을 95% 절감한 방법을 알아보세요.

### 수집의 진화 Scribd의 스

트림 처리 플랫폼은 몇 차례에 걸쳐 개편되었습니다. 초기에는 모든 처리가 상당히 표준적인 스트림 처리 방식이었던 Kafka와 Hadoop에서 수행되었습니다. 이 버전의 플랫폼은 나중에 Spark Structured Streaming 및 Delta Lake를 사용하여 Kafka 및 Databricks로 이동하여 포함되었습니다. 이는 부분적으로 최적화 및 진공 유 틸리티, ACID 트랜잭션 추가와 같은 Delta Lake의 기능 때문에 Scribd에게 유리한 움직임이었습니다.

그러나 Scribd의 경우 토픽 스트림이 많았고 그 중 작은 편도 많았습니다. 이로 인해 급증하는 수집 비용을 줄이려는 시도가 이루어졌습니다.

자연스러운 접근 방식 중 하나는 동일한 클러스터에 여러 스트림 처리 애플리케이션을 스택하는 것입니다. 이를 통해 클러스터 리소스를 보다 최적으로 활용할 수 있습니다. Scribd에서는 그렇게 하기 위해 "낭비적이지 않을 때", 즉 클러스터 리소스를 효율적으로 활용하는 대규모 작업이 있을 때 더 큰 전용 클러스터가 여전히 사용되었습니다. 대신 많은 작은 스트림이 스택되어(동일한 클러스터에서 동시에 실행) 유사한 수준의 효율적인 리소스 활용을 생성하고 따라서 전체 처리 비용을 줄입니다. 하지만 이를 수행하는 데에는 여전히 몇 가지 과제가 있습니다. 주제를 논리적으로 그룹화하는 방법을 결정하는 것은 실망스러울 수 있습니다. 처리 작업 중 하나가 실패하여 해당 클러스터에 쌓인 모든 스트림이 연속적으로 실패할 가능성은 항상 존재합니다. 이는 수집 프로세스에서 유지 관리 작업을 수용하려는 이미 약간 어려운 작업에 추가됩니다.

Scribd 팀은 상황을 개선하고자 하는 몇 가지 소망을 갖고 있었습니다.

- 가능한 경우 비용 절감 • 수집 프로세스에 대한 다 양한 관찰 가능성 • 작업 실패 처리 개선

- 이벤트 스트림의 처리량 크기 변화에 대한 보다 유연한 조정

이는 또한 그들이 문제에 어떻게 접근할 수 있는지에 대한 사려 깊은 성찰로 이어졌습니다. Spark 없이 이 작업을 수행할 수 있습니까? 아니면 좀 더 최소한의 오버헤드 방법을 찾을 수 있습니까? 관리 업무가 훨씬 쉬워졌는데 Delta Lake에서 표준화를 어떻게 계속 유지할까요?

당시 Scribd 팀에게는 약간의 노력을 기울이면 문제에 접근하는 다른 방법이 있을 수 있을 것 같았습니다. 필터, 조인 또는 집계가 없는 추가 전용 작업인 비교적 간단한 수집 프로세스를 가지며 Delta Lake 기능의 하위 집합만 사용하므로 대안 개발이 단순화되는 것으로 입증되었습니다.

Scribd의 시나리오는 현재 더 큰 Delta Lake 생태계의 일부로 잘 지원되고 수용되는 두 가지 프로젝트 개발에 대한 투자로 이어졌습니다. 첫 번째 프로젝트는 **Delta-rs**이고, **Delta Lake 프로토콜**의 Rust 기반 구현 5장에서 자세히 살펴보았습니다. 두 번째 프로젝트는 **kafka-delta-ingest**입니다. Kafka 주제 스트림의 데이터를 Delta Lake 테이블로 빠르고 쉽게 수집하도록 설계된 경량 동반 프레임워크입니다. 이들은 함께 효율적인 작동 쌍을 형성합니다 (**그림 7-9는** 단순화된 데이터 흐름을 보여 줍니다).

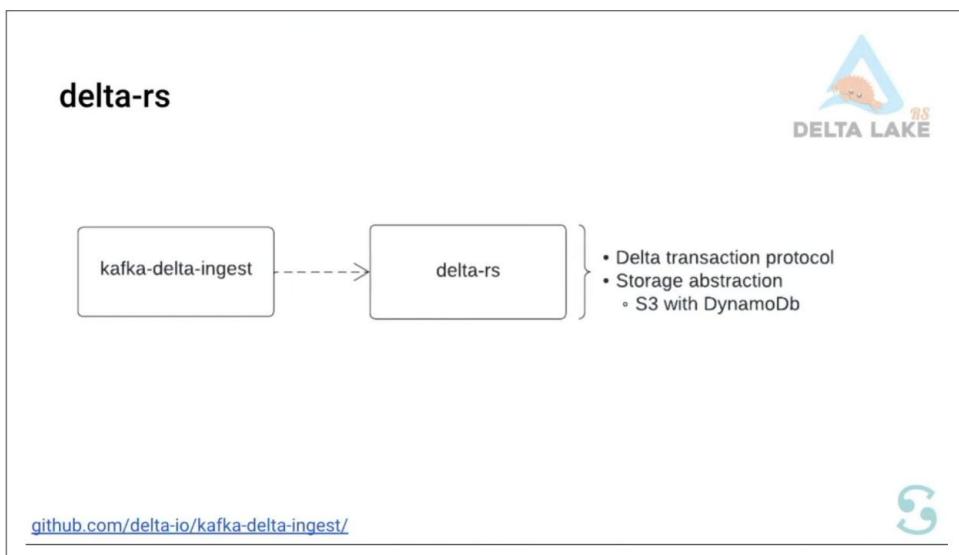


그림 7-9. 효율적인 수집을 위해 Delta-rs 와 함께 Scribd의 kafka-delta-ingest를 사용합니다 .

그러한 노력을 수행하는 데 위험이 있거나 잠재적으로 방해가 되는 문제가 없지는 않았습니다. 중복되거나 삭제된 레코드를 피하기 위해 Kafka에서 오프셋 추적을 수동으로 제어해야 하는 것과 마찬가지로 델타 로그가 손상될 위험이 한 가지 문제를 야기했습니다. 그들은 또한 필요합니다

여러 작성자를 지원하기 위해 AWS S3의 일부 제한 사항에는 특정 처리가 필요합니다(예: S3 잠금 조정).<sup>9</sup>

Scribd는 프로덕션 환경에서 이러한 kafka-delta-ingest 및 delta-rs 파이프라인 중 70-90개를 실행합니다. AWS Fargate를 통해 이러한 파이프라인의 서비스 컴퓨팅을 실행합니다. Datadog의 모든 것을 모니터링합니다. 모니터링하는 항목에는 메시지 역직렬화 로그와 변환 수, 실패, 메모리의 화살표 배치 수, 작성된 쪽모이 세공 데이터 파일 크기, Kafka 스트림의 현재 시간 지연 등 여러 측정항목이 포함됩니다.


  
**DELTA LAKE**

## Cost/Infra - Examples

**Based on Public Pricing Information**

Example 1	Example 2	Example 3
Spark	Spark	Spark
<ul style="list-style-type: none"> <li>• Driver: r5.large</li> <li>• Workers: r5.large x3</li> <li>• ~\$5,200 annually</li> </ul>	<ul style="list-style-type: none"> <li>• Driver: m5.large</li> <li>• Workers: r5.large x2</li> <li>• ~\$3,900 annually</li> </ul>	<ul style="list-style-type: none"> <li>• Driver: r5.large</li> <li>• Workers: r5.large x5</li> <li>• ~\$23,170 annually</li> </ul>
Rust	Rust	Rust
<ul style="list-style-type: none"> <li>• 1 vcpu x1</li> <li>• 4gb</li> <li>• \$400 annually</li> </ul>	<ul style="list-style-type: none"> <li>• 1/4 vcpu x1</li> <li>• 2gb</li> <li>• \$100 annually</li> </ul>	<ul style="list-style-type: none"> <li>• 2 vcpu x3</li> <li>• 16gb</li> <li>• \$2200 annually</li> </ul>

[github.com/delta-io/kafka-delta-ingest/](https://github.com/delta-io/kafka-delta-ingest/)

그림 7-10. Data+AI Summit 2022에서 Scribd가 공유한 비용 절감 사례 중 일부는 원래 Spark에서 프로세스를 실행한 다음 유사하게 Delta-rs를 사용하여 프로세스를 실행하는 비용을 비교했습니다. Rust 리소스는 vCPU와 메모리 할당을 표시하는 반면 Spark 클러스터는 전체 EC2 인스턴스를 사용하며, 표시된 대로 r5.large 인스턴스에는 vCPU 2개와 16GB RAM이 있습니다.<sup>10</sup>

이 모든 것은 Scribd 팀이 구축한 도구를 사용하여 일부 스트림 처리 애플리케이션을 실행하는 데 드는 비용이 100배 정도 낮아짐에 따라 수집 처리에서 상당한 비용 절감으로 이어졌습니다. 이 환상적인 성과를 마무리하는 또 다른 특징은 수집된 데이터를 즉시 분석에 사용할 수 있는 방식으로(Delta Lake에서 표준화된 상태를 유지함으로써) 달성된다는 것입니다.

<sup>9</sup> 이러한 S3 문제 중 일부는 D3L2 웹 시리즈 에피소드 "The Inception of Delta Rust"에서 논의됩니다.

유튜브: <https://www.youtube.com/watch?v=2JgfpJD5D6U>.

10개의 AWS r5 유형 지표는 여기에서 찾을 수 있습니다: <https://aws.amazon.com/ec2/instance-types/r5/>

IC 및 기계 학습 프로세스 또는 Databricks 환경의 다른 배치 프로세스와의 추가 통합을 통해 쿼리 가능성을 유지합니다.

## 복잡한 시스템 조정

스마트 장치 및 엔터테인먼트부터 보안 및 디지털 결제 시스템에 이르기까지 대용량 데이터 소스가 부족하지 않습니다. Scribd를 사용하면 kafka-delta-ingest가 실행 가능한 솔루션인 운영 체제에 대한 스트레스를 줄이면서 간단한 이벤트 캡처에 중점을 두었습니다. 이제 외부 세계와의 상호 작용 가장자리가 덜 간단하고 더 많은 서비스가 필요한 경우를 고려해 보겠습니다. 더 자저분하고 더 인간적입니다. 지속적으로 발전하는 복잡한 애플리케이션에는 시간이 지남에 따라 조화를 유지해야 하는 더 많은 통합 운영 구성 요소가 있는 경향이 있습니다. 또는 원하는 대로 새로운 요구 사항, 소스 또는 프로세스에 대해 생각하는 대신 기존 데이터를 관리하는 데 더 많은 시간을 소비하게 될 수도 있습니다.

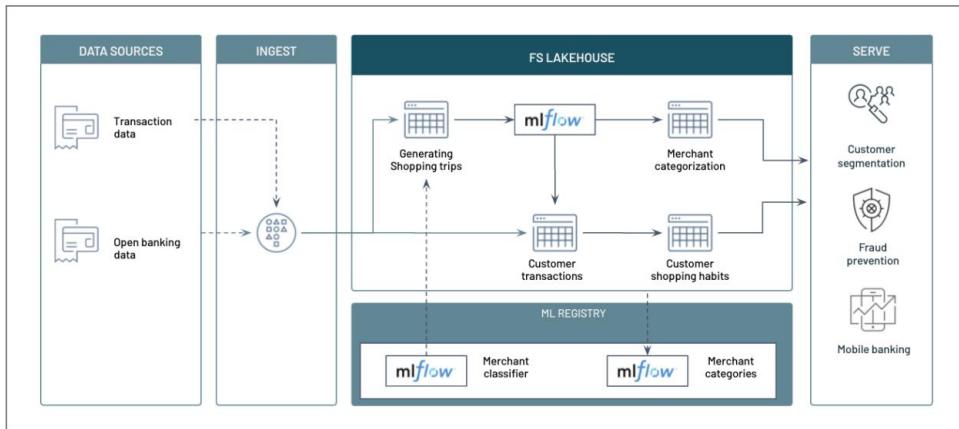


그림 7-11. 소매 상인 신용 거래는 복잡한 시스템 상호 작용을 볼 수 있는 영역 중 하나일 뿐입니다.

여러 개의 실시간 운영 데이터베이스가 포함되고 비즈니스 가치 창출에 대한 요구가 있다는 것은 분석 및 기계 학습 애플리케이션 개발을 위해 이러한 데이터베이스의 정보를 통합된 위치로 수집해야 한다는 것을 의미하는 경우가 많습니다. 다른 시스템에는 운영 데이터베이스가 없지만 이벤트 중심 시스템에 의존 할 수 있습니다. 종종 이 데이터는 공개 시장에서 사용할 수 있는 악명화된 추세 데이터가 포함된 고객 거래 데이터와 같이 상대적으로 복잡한 데이터 생태계를 생성하는 다른 시스템의 데이터와 함께 필요할 수 있습니다.

**그림 7-11은** 이와 같은 데이터 소스를 결합하여 여러 다운스트림 애플리케이션을 지원하는 방법을 보여 줍니다. 다양한 시스템을 위한 광범위한 커넥터 배열을 갖춘 Delta Lake와 같은 레이크하우스 형식을 사용하면 이러한 복잡성이 줄어들고 분석 및 인공 지능 기반 애플리케이션이 가능해집니다.

DoorDash에서 운영 데이터 저장소 결합 많은 사람들은 누군가 가 식사, 식료품, 전자 제품 또는 기타 거의 모든 것을 픽업하는 데 도움을 주고 여행을 떠날 수 있다면 편리할 상황에 처해 있습니다. DoorDash는 배송 서비스를 통해 유연성과 편의성을 제공함으로써 모든 종류의 요구를 충족하는 데 도움을 줍니다. 대부분은 "공연" 기반 운영 방법에 익숙하지만 고려해야 할 몇 가지 특정 사항을 알아 두는 것이 도움이 될 수 있습니다.

DoorDash를 통한 구매 프로세스에는 여러 당사자가 참여합니다.

일반적으로 요청자, 배달하는 사람, 주문을 준비하거나 제품을 제공하는 레스토랑이나 판매자가 있습니다. DoorDash 조직의 더 큰 IT 생태계에 들어가지 않고도 이미 대규모의 저지연 데이터 파이프라인, 즉 스트리밍 데이터 애플리케이션에 대한 분명한 필요성이 있습니다. 왜냐하면 각 "이벤트" 자체가 많은 이벤트의 조합이기 때문입니다. 프로세스를 단계별로 진행합니다.

DoorDash는 두 가지 방법으로 Delta Lake를 데이터 생태계의 일부로 활용하고 있습니다. 첫 번째는 대규모 변경 데이터 캡처(CDC) 관리와 분석을 위한 데이터 다운스트림 노출을 단순화하는 것입니다. 두 번째는 Flink에서 실시간 워크로드를 지원하는 것입니다. 두 가지 모두 아키텍처 설계에 Delta Lake를 활용함으로써 얻을 수 있는 이점 중 일부를 포착합니다.

#### 변경 데이터 캡처 변경 데

이터 캡처(CDC)는 다양한 이유로 지원이 필요한 일반적인 애플리케이션 패턴입니다.<sup>11</sup> DoorDash에서는 여러 서비스를 지원하는 운영 데이터베이스를 분석 환경으로 복제하기 위해 CDC를 사용합니다.

이는 "어제 DoorDash가 얼마나 많은 주문을 했습니까?"라는 질문에 대답할 수 있어야 하는 역사적 필요성에 의해 주도됩니다. 이전에는 데이터베이스 복사본을 만들고 복사본에 대한 쿼리를 사용하여 분석 질문에 대답하거나 데이터 과학 작업을 수행함으로써 질문에 대한 대답을 해결할 수 있었기 때문에 더 쉬운 작업이었습니다.

DoorDash가 성장함에 따라 서비스 아키텍처가 발전하여 **CockroachDB** 와 같은 다양한 버전으로 제공되는 여러 운영 데이터베이스가 있는 환경으로 이어졌습니다. **포스트그레SQL**, 그리고 **아파치 카assandra**. 가장 간단한 방법으로 이러한 데이터베이스에서 데이터를 얻으려고 처음에는 데이터베이스에서 스냅샷을 가져와 매일 가져왔습니다. 이 접근 방식은 효과가 있었지만 특히 데이터 버전 관리를 추적하고 데이터 프로세스를 증분화하기 위해 스냅샷을 필터링해야 하는 등의 문제를 야기했습니다.

---

<sup>11</sup> 논리적 로그 복제라고도 알려진 CDC를 탐색하는 데 더 많은 시간을 할애하고 싶다면 다음을 제안합니다. Martin Kleppmann(O'Reilly)의 데이터 집약적 애플리케이션.

효율적으로 환경에서 다양한 변화를 시도한 후 팀은 결국 보다 강력한 시스템을 개발하기 시작했습니다.

우리 목적을 위한 주요 시스템 요구 사항은 다음과 같습니다.

- 데이터 지연 시간이 하루 미만 •

Lakehouse 설계 패턴 사용 • 스키마

발전 지원 • 데이터 백필 허용 • 분

서브 워크로드 활성화 • 한 번 쓰고

여러 번 읽기 • 늦게 도착하는 데이

터 방지 • 오픈 소스 소프트웨어로

구축

이러한 요구 사항에서 나온 설계는 광범위한 쿼리 인터페이스에서 다운스트림 통합을 지원하는 Delta Lake에 구축된 통합 정보 소스에 변경 피드를 복제하는 Spark Structured Streaming을 기반으로 구축된 스트리밍 CDC 프레임워크입니다. 병합 지원 및 ACID 트랜잭션과 같은 기능은 Delta Lake를 설계의 중요한 구성 요소로 만드는 데 도움이 되었습니다.

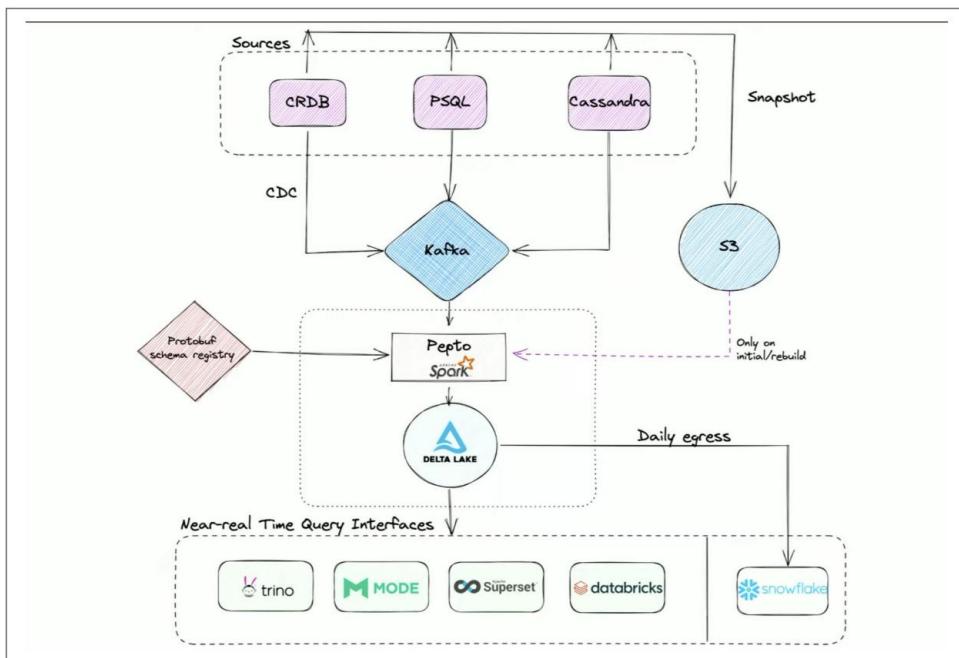


그림 7-12. DoorDash의 CDC 지원 Lakehouse 아키텍처 설계.

이 디자인의 성공은 여러 가지 방법으로 측정할 수 있지만 팀이 강조하는 몇 가지 사항이 있습니다. 이 시스템은 1000개가 넘는 EC2 노드에서 연중무휴로 실행되는 450개의 스트림(테이블과 일대일)을 지원합니다. 이는 Kafka에서 매일 약 800GB를 수집하고 일일 총 처리량은 약 80TB에 해당합니다. 설계는 초기 요구 사항을 훨씬 초과했으며 30분 미만의 데이터 최신 상태를 달성했습니다. 이를 통해 환경 내 데이터 사용자를 위한 셀프 서비스 테이블 생성이 가능해졌고, 이는 1시간 이내에 사용 가능해졌습니다.

### Delta와 Flink의 조화 실시

간 이벤트가 DoorDash의 핵심이므로 Kafka를 많이 사용하는 것은 놀라운 일이 아닙니다. Apache Spark는 많은 스트림 처리 애플리케이션에 자연스러운 선택입니다. 그러나 이것이 유일한 선택은 아닙니다. DoorDash의 일부 팀은 많은 실시간 프로세스에 Apache Flink을 사용하므로 쉽게 지원할 수 있어야 합니다. 5장에서는 Flink/Delta 커넥터가 어떻게 작동하는지 살펴보았지만 여기서는 이를 더 큰 데이터 생태계로 끌어와 유연성과 안정성을 모두 제공할 수 있는 방법을 알아보는 것이 유용할 수 있습니다.

DoorDash의 실시간 플랫폼 팀은 매일 페타바이트 규모의 중요한 고객 이벤트를 관리하고 있으며 데이터 사용자와 애플리케이션이 이 정보를 캡처, 생성 또는 액세스할 수 있도록 하는 플랫폼을 제공해야 합니다. Flink/Delta 커넥터를 추가하면 Flink의 빠른 작동 특성과 이러한 종류의 워크로드를 정확하게 처리하도록 구축된 스토리지 형식을 결합하고 전체 데이터에서 사용할 수 있는 공통 형식을 제공하는 Delta Lake와 사용자 및 애플리케이션이 상호 작용할 수 있는 방법이 확장됩니다. 다양한 팀이 다양한 애플리케이션 처리 프레임워크를 활용하기로 선택하는 경우에도 마찬가지입니다.

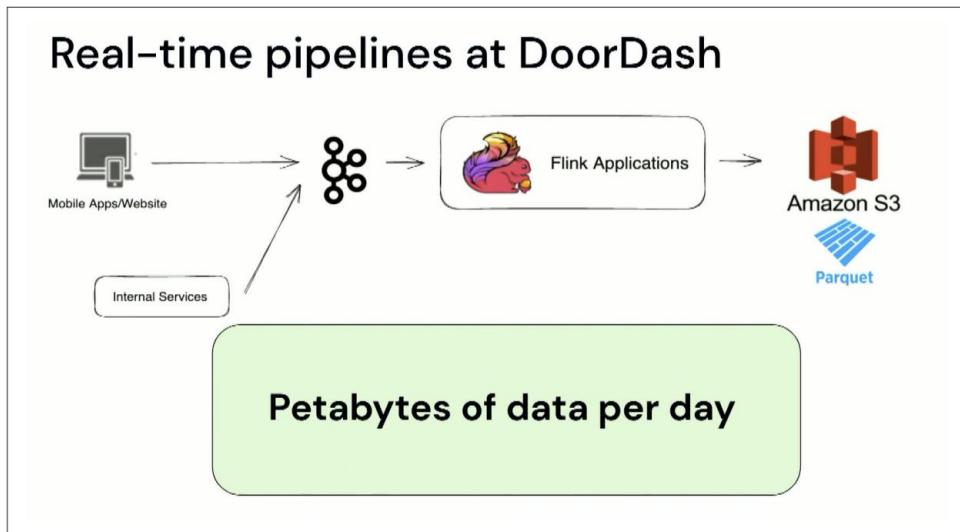


그림 7-13. Delta Lake로 이동하기 전 DoorDash의 프로세스 시작 상태입니다.

이것이 바로 DoorDash의 이러한 변경으로 가능해진 것이며, 대규모 ACID 보장을 추가하여 현재 도구와 쉽게 통합할 수 있습니다. 이전에는 이 프로세스가 쓰기 잠금 및 기타 문제의 형태로 추가적인 복잡성을 추가하는 일반 Parquet 파일에서 수행되었습니다. 또한 사용하기 쉬운 압축 작업을 통해 얻은 삶의 질 향상과 스트림 처리 애플리케이션이 계속 실행되는 동안 이러한 작업을 수행할 수 있는 기능은 매우 중요합니다. 데이터에 z 순서 클러스터를 포함하여 효율적으로 쿼리 가능한 상태도 마찬가지입니다.

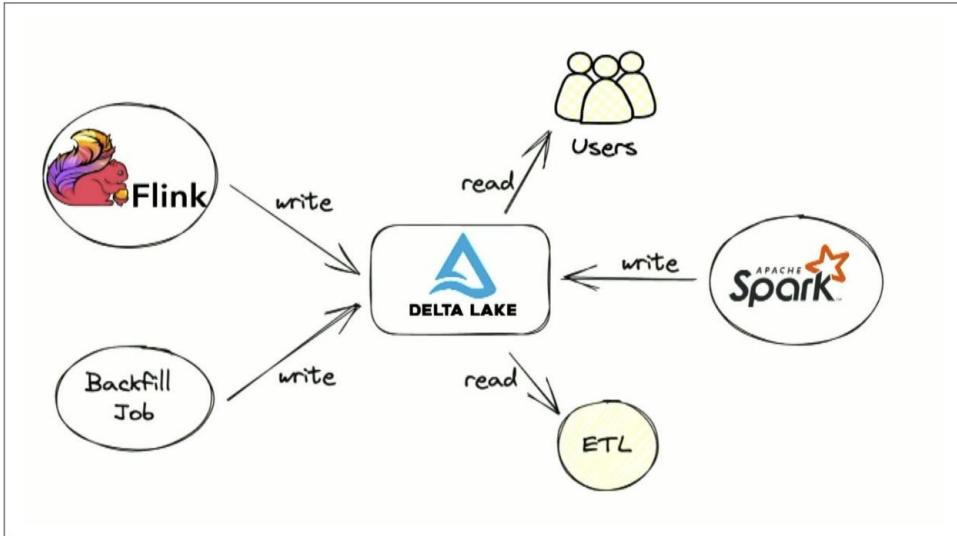


그림 7-14. Delta Lake로 이동한 후 DoorDash의 데이터 생태계 결과 상태입니다.

Delta Lake를 채택하기로 한 DoorDash 결정의 교훈은 다음과 같습니다. 대규모로 작동하는 여러 유형의 도구가 있는 데이터 시스템의 경우에도 실시간 이벤트 스트림에서 데이터를 효율적으로 캡처하거나 운영을 통해 발생하는 변경 사항을 지원해야 한다는 것입니다. 데이터베이스 Delta Lake는 신뢰성과 유용성을 제공하여 성공적인 선택입니다.

## 결론

데이터 애플리케이션은 다양한 형태와 형식으로 제공됩니다. 이러한 데이터 애플리케이션을 작성하는 것은 복잡하고 고통스러울 수 있습니다. 여기에서는 Delta Lake의 다양한 이점을 통해 이러한 고통을 완화할 수 있는 몇 가지 방법을 살펴보았습니다. 특히 Delta Lake의 기능은 광범위한 도구 선택을 지원하고 비용을 절감하며 개발자로서의 삶의 질을 향상시키는 강력한 데이터 환경을 만드는 데 도움이 됩니다.

## 참고자료

### 컴캐스트

- 음성, 데이터, AI를 통해 홈 엔터테인먼트 혁신 • 기업 메타데이터 및 보안을 위한 Comcast 아키텍처에 대해 알아보세요. • AI로 청중 확보: Comcast가 민첩한 데이터와 AI를 구축한 방법 대규모 플랫폼 | (컴캐스트)
  - Comcast의 원격 측정 분석을 지원하는 SQL 분석 • Comcast는 음성, 데이터 및 통신을 통해 누구나 홈 엔터테인먼트에 액세스할 수 있도록 해줍니다.
- 일체 포함
- Comcast의 원격 측정 분석을 지원하는 SQL 분석

### 스크리브

- Kafka에서 Delta Lake로 최대한 빠르게
- Rust 및 Kafka를 사용하여 Delta Lake로 데이터 스트리밍

### 도어대시

- Apache Flink에서 Delta Lake에 쓰기 • Delta Lake 테이블용 Apache Flink 소스 커넥터 • Kafka 및 Flink를 사용하여 확장 가능한 실시간 이벤트 처리 구축 • Flink + Delta: DoorDash에서 실시간 파이프라인 구동 • CDC, Apache Spark™ 스트리밍 및 Delta Lake를 통해 거의 실시간 데이터 복제 잠금 해제

## 레이크하우스 거버넌스 및 보안

초기 출시 독자를 위한 참고 사항 초기 출시 eBook을 사용

하면 가장 초기 형태의 책, 즉 저자가 집필하는 그대로의 편집되지 않은 원본 콘텐츠를 얻을 수 있으므로 이러한 타이틀이 공식 출시되기 훨씬 전에 이러한 기술을 활용할 수 있습니다.

이것이 마지막 책의 14번째 장이 될 것이다. GitHub 저장소는 나중에 활성화될 예정입니다.

이 책의 내용 및/또는 예제를 개선할 수 있는 방법에 대한 의견이 있거나 이 장에 누락된 자료가 있는 경우 [gobrien@oreilly.com](mailto:gobrien@oreilly.com)으로 편집자에게 문의하십시오.

우리는 매일 의식적으로 생각하지 않고 많은 일을 합니다. 이러한 기계적인 행동, 즉 자동적인 행동은 우리의 일상생활과 시간이 지남에 따라 우리가 신뢰하게 된 정보를 기반으로 합니다. 우리의 루틴은 작업이 서로 다른 논리적 버킷으로 그룹화되고 분류되어 간단할 수도 있고 복잡할 수도 있습니다. 예를 들어, 하루를 떠나기 전에 문을 잠그는 일상을 생각해 보십시오. 이는 위험을 완화하기 위한 일반적인 행동입니다. 모든 사람이 최선의 이익을 염두에 두고 있다고 믿을 수는 없기 때문입니다. 이 위험 완화를 간단한 이야기로 생각하면 물리적 위치(엔티티: 집, 자동차, 사무실)에 대한 무단 접근을 방지하기 위해 접근 제어(잠금 메커니즘)가 도입되어 물리적 공간(자원)을 확보하고 승인된 서비스를 제공합니다. 신뢰가 확인될 수 있는 경우에만 입장 가능(키).

가장 단순한 의미에서 침입을 방지하는 유일한 방법은 열쇠입니다. 키는 주어진 물리적 공간에 대한 액세스를 허용하지만, 주어진 키의 소유자는 보호되는 리소스의 물리적 위치도 알아야 합니다. 그렇지 않으면 키가 아무 소용이 없습니다. 이는 사이트 보안의 한 예로서 멘탈 모델로서 사이트 보안에 대한 계획을 수립할 때 유용합니다.

레이크하우스에 포함된 리소스에 대한 계층화된 거버넌스 및 보안 모델입니다.

결국, 호숫가는 우리가 그 안에 포함된 자원을 집단적으로 관리한다면 우리가 가까이에 있고 소중한 것을 보호하는 안전한 공간일 뿐입니다.

하지만 데이터 리소스의 거버넌스는 정확히 무엇이며, 거버넌스 환경에 다양한 측면이 있을 때 어떻게 시작해야 할까요?

이 장은 레이크하우스에 포함된 데이터 자산(리소스)에 대한 확장 가능한 데이터 거버넌스 전략을 설계하기 위한 기초를 제공합니다. 계층화된 보안의 기본 사항, 제안된 사용자 페르소나 및 역할(그룹)은 물론 액세스 및 권한 부여, 감사를 단순화하기 위해 구현할 수 있는 청사진을 포함하여 다중 테넌트 환경에서 보안, 개인 정보 보호 및 거버넌스와 관련된 문제를 해결하기 위한 패턴을 다룹니다. 로깅, 그리고 훨씬 더. 여기서는 가능한 한 많은 표면적을 다루는 것을 목표로 하고 있지만, 이 장을 레이크하우스 데이터 거버넌스의 수많은 측면을 표면적으로만 살펴보는 참조 장으로 간주하십시오.

## 레이크하우스 거버넌스

레이크하우스 거버넌스에 대해 자세히 알아보기 전에 거버넌스 우산의 다양한 측면 또는 구성 요소를 소개하는 것이 중요합니다. 전체 이야기에는 최소한 8가지 측면이 있습니다. 이를 통해 기본 액세스 제어를 뛰어넘어 많은 시스템과 서비스를 함께 결합하여 레이크하우스로 들어오고, 사이에서, 밖으로 흐르는 데이터의 동적 네트워크에 대한 포괄적인 보기를 제공할 수 있습니다. 시작하기 위해 [그림 8-1](#)의 다이어그램은 이 장에서 자세히 살펴볼 거버넌스 구성 요소에 대한 조감도를 제공합니다.

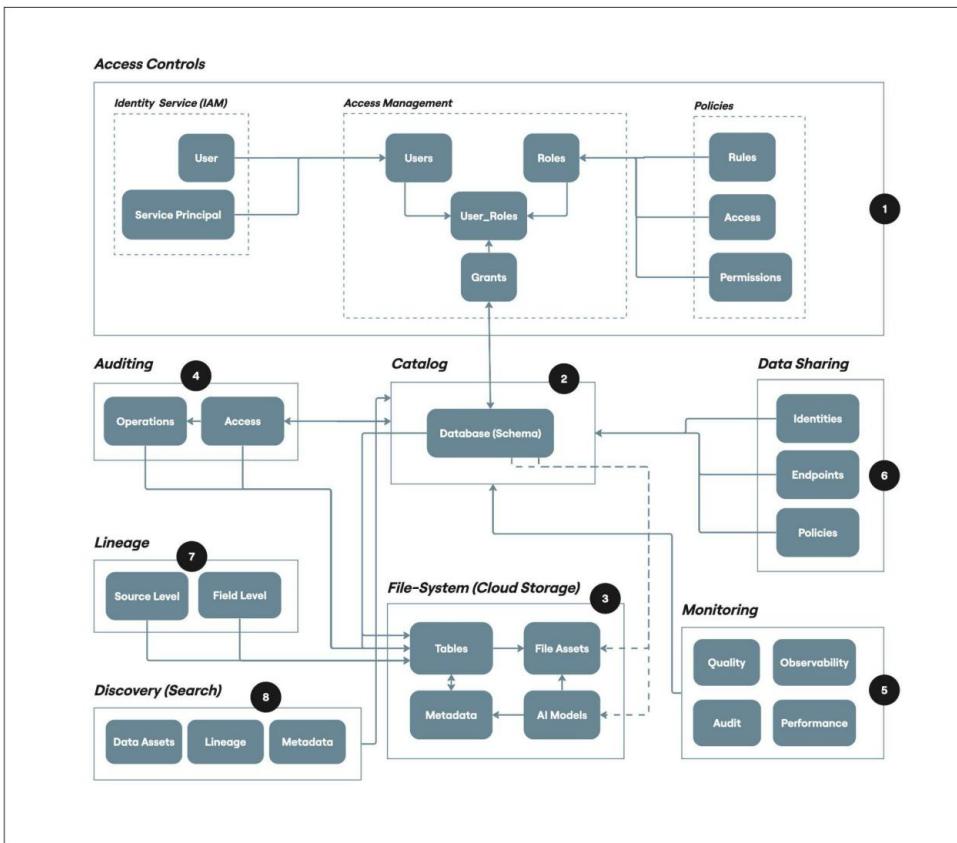


그림 8-1. 레이크하우스를 관리하는 것은 기본적인 파일 시스템 액세스 제어 이상의 것입니다.

그림 8-1에 표시된 것처럼 레이크하우스 거버넌스에는 액세스 제어(1), 데이터 카탈로그(2) 및 기본 클라우드에서 제공하는 탄력적인 데이터 관리에 제시된 기본 구성 요소 및 핵심 테넌트를 포함하여 여려 측면이 있습니다. 객체 저장소 또는 온프레미스 분산 파일 시스템(3).

또한 레이크하우스에 대한 최신 거버넌스에는 (4) 작업별로 일반적인 데이터 관리 작업에 대한 강력한 감사, (5) 상황이 올바르지 않은 것으로 보일 때 이를 알리는 통합 모니터링(감사), 목표 관찰 및 기록도 포함됩니다. 미션 크리티컬 테이블의 데이터 품질과 데이터 파이프라인의 성능은 소유 비용에 대한 통찰력을 높이는 데 도움이 될 수 있습니다. 점을 연결하려면 누가 무엇을, 어디서, 어떻게 데이터에 액세스하고, 변환하고, 달리 "사용"하는지를 포함하여 (7) 데이터 자산 계보(적어도 카탈로그->스키마->테이블)가 필요합니다. 마지막으로 중요한 것은 (8) 우리의 모든 노력의 결정체는 "데이터에 대해 우리가 알고 있는 모든 것"을 함께 묶어 강력한 데이터 검색을 제공하는 능력이기 때문입니다. 이는 틀림없이 주어진 가장 널리 퍼진 문제 중 하나입니다. 점점 더 많은 사일로, 플랫폼, 시스템 및 서비스 전반에 걸쳐 데이터가 확산되고 있습니다. 잘

지금 모든 것에 대한 간략한 개요를 확인한 다음 레이크하우스 거버넌스의 각 측면에 대해 자세히 알아보세요.

## Lakehouse 거버넌스의 측면

### 액세스 제어 (1)

누수 방지 테이블 추상화(기본 스토리지에 직접 액세스할 수 없음)를 통해 레이크하우스 내의 데이터 자산을 보호하고 관리하기 위한 기본 구성 요소를 제공합니다. 사용자나 서비스를 식별할 수 없으면 액세스를 승인 또는 거부할 방법이 없으며 생성, 읽기(보기), 쓰기(삽입), 업데이트(또는 upsert) 또는 삭제 권한을 부여할 수 없습니다. 그것은 거친 서부가 될 것입니다.

### 데이터 카탈로그 (2)

데이터베이스 및 테이블을 나열하고 데이터베이스, 테이블 또는 열 수준에서 권한을 제어하는 기능을 활성화합니다. 카탈로그는 각 테이블에 대한 중요한 메타데이터를 제공합니다. 즉, 테이블이 데이터 레이크에 있는 위치(탄력적 파일 시스템의 델타 테이블 경로)를 정의하고 각 테이블의 소유자, 열, 제약 조건, tblproperties(6장에서 살펴봤습니다)를 정의합니다. 테이블 세트를 포함하는 데이터베이스(dbproperties)와 관련된 메타데이터도 포함됩니다.

따라서 액세스 제어와 데이터 카탈로그는 서로 밀접하게 연관되어 있습니다. 둘 중 하나가 없으면 가질 수 없기 때문입니다.

### 탄력적 데이터 관리(3)

레이크하우스 아키텍처의 마지막 핵심 구성 요소는 데이터 레이크입니다. 이제 우리는 Delta 프로토콜이 스키마 시행 및 발전 기능을 제공하는 데 도움이 되며 테이블 수준에 불변성을 가짐으로써 데이터 관리의 복잡성을 줄일 수 있다는 것을 알고 있습니다. 데이터 레이크는 데이터 카탈로그에 포함된 데이터베이스와 테이블에 탄력적인 확장성을 제공하며, 곧 배우게 되겠지만 ID 및 액세스 관리는 데이터 자산을 안전하게 관리하는 데 중요한 역할을 합니다.

함께, 강력한 기본 모델을 구축하여 레이크하우스를 강화하고 감사 서비스(4), 액세스에 대한 포괄적인 모니터링(5), 데이터 운영 및 작업에 대한 통찰력 생성을 포함한 추가 중요 기능을 위한 기반을 마련할 수 있습니다.

### 감사 (4)

누가 또는 무엇을 리소스(카탈로그, 데이터베이스, 테이블)에 대해 특정 작업(생성 또는 삭제 등)을 실행할 수 있는지 또는 변경 테이블과 같은 중요한 변경이 발생할 때와 같은 레이크하우스 동작의 변경 사항을 캡처합니다. 테이블에 대한 작업(작업의 테이블 버전 기록) 또는 예를 들어 테이블이 삭제(및 삭제)되는 경우입니다. 또한 단순 감사는 특정 데이터 자산에 대한 액세스 빈도를 추적하고, 무언가가 발생한 시기를 추적합니다.

인기가 있거나, 자주 액세스되지 않거나, 읽기나 쓰기를 위해 액세스한 적이 없거나, 둘 다를 수행합니다.  
이 장의 뒷부분에서 감사 추적 및 측정 항목을 다를 것입니다.

#### 모니터링 (5)

감사 렌즈(보안 목적) 및 테이블 수준(엔지니어, 분석가 및 과학자용)을 통해 레이크하우스의 동작을 캡처하면 단순히 메트릭 또는 이벤트의 기록(시계열)이 제공됩니다. 모니터링은 지표를 가져와 이를 변환하여 핵심 성과 지표(KPI)를 생성하고 이벤트(감사 이벤트)를 지표(KPI)로 변환하여 통찰력을 생성하는 데 필요합니다. 각 KPI는 레이크하우스 내 또는 특정 데이터 자산에 대한 추세를 이해하는 데 사용할 수 있는 측정값을 제공하며 정보를 올리거나(경고 또는 페이지를 통해) 팀에 중앙 통신 채널을 제공하는 데 중요합니다.

통합 데이터 품질 지표, 액세스 및 권한 내역, 시스템 전반의 이벤트 추적이 함께 결합되어 데이터 자산과 관련된 변경 사항을 관찰하는 비행 기록 장치처럼 작동합니다. 즉, 중요한 역사적 순간을 여러 시스템 및 서비스의 상태와 함께 연결합니다. 거버넌스 스택. 적절한 모니터링(5) 및 감사 로깅(4)이 없으면 읽기 전용 데이터 공유 또는 무복제 공유(6) 및 데이터 계보 기록(7)과 같은 고급 기능은 그다지 강력하지 않습니다.

#### 데이터 공유 및 무복제 공유(6)

우리는 4장에서 데이터 공유에 대해 살펴보고 11장에서도 다시 살펴보았습니다. 데이터 공유는 신뢰도가 높은 데이터 생태계를 먼저 구축하려면 운영 성숙도가 필요하기 때문에 레이크하우스 거버넌스의 복잡한 구성 요소입니다. 통제할 수 없는 사용자 및 서비스와 데이터를 공유하면 비용이 적게 들고 데이터 관리 오버헤드도 줄어듭니다. 단, 레이크하우스 외부로 데이터를 내보낼 필요 없이 데이터를 읽을 수 있는 경우에만 해당됩니다. 따라서 델타 공유 프로토콜에는 기반(1, 2, 3, 4, 5)이 마련되어야 합니다. 공유 및 수신자 관리 추가는 내부 액세스 관리 패러다임의 확장일 뿐이기 때문입니다.

#### 데이터 계보 (7)

정적이거나 동적일 수 있습니다. 각 데이터 애플리케이션(파이프라인, 워크플로우, 스트리밍 또는 배치)이 하나 이상의 소스에서 데이터를 가져와(읽기를 통해) 데이터를 변환하고 (쓰기를 통해) 레이크하우스 내 또는 레이크하우스 외부의 다른 위치(테이블)로 전송한다는 점을 고려하면, Apache Kafka 또는 Pulsar와 같은 스트리밍 프로세서도 있습니다.

DLAG(방향성 계보 그래프)는 각 데이터 애플리케이션에 대한 메타데이터를 사용하여 동적으로 구성할 수 있습니다.

나중에 계보를 더 자세히 살펴보겠지만, 데이터가 어떻게 생성되고 소비되는지에 대한 귀중한 통찰력을 제공하고 데이터 조직의 규모에 따라 데이터 흐름에 대한 귀중한 눈과 귀를 제공합니다. 다른 방식으로 생성되었습니다.

우리는 발생한 일의 기록을 생성하기 위해 중요한 데이터 자산에 대한 관찰 가능한 상태 변경, 운영 및 작업에 대한 데이터를 캡처합니다. 이 정보는 규정 준수 감사(gdpr, ccpa 등)를 관리하고, 위험을 식별하고, 시간이 지남에 따라 데이터 품질을 추적하고, 기내치가 다를 경우 조치를 취하고, 데이터 중단을 정확히 찾아내는 알림을 자동화하는 데에도 유용합니다.

마지막으로 중요한 것은 데이터 검색(8)의 추가입니다. 다른 모든 시스템과 서비스가 연결되어 함께 작동하면 관리되는 데이터 자산의 메타데이터를 색인화하고 모든 데이터 검색 엔진을 강화하는 지능형 검색을 제공하는 것이 훨씬 쉬워집니다.

#### 데이터 발견 (8)

레이크하우스 내부의 데이터 자산(데이터베이스, 테이블, \*쿼리, \*대시보드, \*모니터 및 \*경고)의 경우 데이터 검색은 다양한 인물(엔지니어, 분석가, 과학자)이 최상의 시작점을 신속하게 식별할 수 있도록 하는 필수 구성 요소가 됩니다. 긴 일련의 회의를 만들거나 복잡하게 조정된 노력을 할 필요 없이 업무를 수행할 수 있습니다. 특정 테이블에 대한 액세스 빈도(감사(4))에 대한 통찰력을 재사용함으로써 인기도 점수를 정의하여 사용량별로 데이터 자산을 표시하는 데 도움을 줄 수도 있습니다.

데이터 엔지니어라면 누구나 알 수 있듯이 런타임에 모든 종류의 문제가 발생할 수 있습니다. 예를 들어 데이터 자산에 대한 액세스가 (옳든 그르든) 취소되어 데이터 파이프라인이 실패하거나 성능이 저하될 수 있습니다. 테이블은 더 이상 사용되지 않거나 더 이상 업데이트되지 않는 읽기 전용 모드로 전환되거나 실수로 삭제될 수도 있습니다(적절한 거버넌스 확인 및 균형 없이). 권한 변경, 테이블 상태에 대한 명확한 기록이 없거나 상태 변경이나 성능 저하를 데이터 이해관계자에게 전달하기 위한 확립된 패턴이 없으면 신뢰도가 저하됩니다.

명확한 의사소통 수단이 없으면 신뢰는 쉽게 깨집니다. 데이터 거버넌스는 복잡한 문제를 해결하기 위해 노력하는 서로 다른 데이터 팀을 연결하는 데 도움이 되는 보안 및 규정 준수를 뛰어넘는 안정적인 도구와 서비스를 통해 높은 신뢰도의 환경을 유지하는 한 가지 방법입니다.



이 장에서는 지역 간 데이터 액세스 관리를 위한 설계 패턴뿐만 아니라 지역 데이터 거버넌스 및 규정 준수 규정을 건너뜁니다. 이러한 주제는 이 책의 범위를 벗어나지만 전체 거버넌스 솔루션의 중요한 핵심 원칙으로 남아 있습니다.

## 데이터 거버넌스의 출현

데이터 거버넌스는 "전체 수명주기 동안 조직의 데이터 자산을 관리하기 위한 다양한 원칙, 관행, 도구 및 워크플로를 통합하는 우산"으로 정의됩니다. 데이터의 수명주기는 모든 변환과 액세스를 포함하여 생성부터 삭제까지 전체 앤드투엔드 여정을 캡슐화합니다.

그리고 그 과정에서 (데이터가 존재하는 범위 내에서) 어느 시점에서든 데이터를 활용합니다.

델타 테이블의 관점에서 데이터의 수명 주기를 고려해보세요.

- 데이터의 통로는 테이블 자체입니다.
- 각 테이블은 테이블에 누가, 무엇을, 어디서, 어떻게 변경했는지에 대한 트랜잭션 로그와 함께 시간이 지남에 따라 제한되거나 제한되지 않은 데이터 집합을 저장하는 컨테이너를 제공합니다.
- 테이블은 단순히 존재했다가 사라지는 것이 아닙니다. 전체 여정을 완료하려면 먼저 각 테이블을 생성하고, 행을 삽입, 읽기, 수정 또는 삭제해야 하며, 테이블도 삭제(삭제)해야 합니다.

이 수명 주기는 리소스 수준에서 발생하는 작업 및 작업(타임라인)의 전체 기록을 캡슐화합니다. 이러한 관찰 가능한 순간은 데이터 거버넌스의 목적뿐만 아니라 엔지니어링 관점에서 테이블의 유지 관리 및 유용성을 위해 매우 중요합니다. 각 테이블은 데이터 자산이라고 하는 관리 가능한 리소스입니다.



여기서 자산이라는 용어의 사용을 고려하는 것이 중요합니다. 데이터 자산(테이블)은 조직을 대표하는 사람(또는 팀)이 직접 소유, 관리, 관리합니다. 그러면 조직은 데이터 자산을 관리하고 엔지니어링, 제품, 보안, 개인 정보 보호 및 거버넌스 전반에 걸쳐 책임 당사자에게 비용을 지불하기 위한 자금을 제공합니다.

경험상 데이터 자산은 가치를 제공하는 동안에만 유지되어야 합니다. 데이터 수명주기 관리는 데이터가 더 이상 유용하지 않을 때까지만 존재하는 것으로 생각할 때 더 의미가 있습니다.

우리는 11장에서 데이터 품질을 위한 메달리온 아키텍처에 대해 배웠습니다. 이 새로운 디자인 패턴은 브론즈에서 실버, 골드까지 데이터 정체를 위한 3계층 접근 방식을 도입했습니다. 이 아키텍처는 시간이 지남에 따라 데이터 자산의 수명 주기를 관리하고 특정 계층에서 데이터를 얼마나 오래 보관할지 고려할 때 실용적인 역할을 합니다. 그림 8-2 의 다이어그램을 활용하면 시간이 지남에 따라 브론즈, 실버, 골드로 표시되는 논리적 데이터 품질 경계를 넘어 데이터 자산의 가치가 개선되는 방식을 시각화할 수 있습니다.

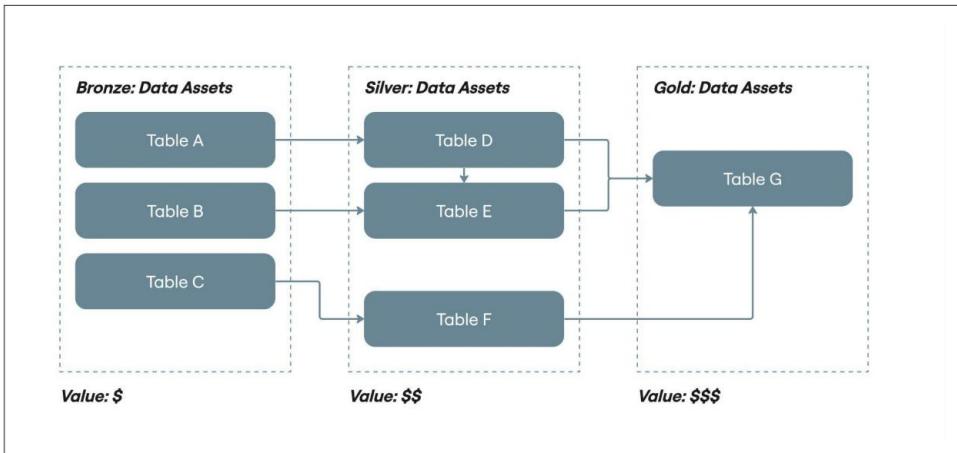


그림 8-2. 우리 데이터 자산의 가치는 브론즈에서 실버로, 실버에서 골드로 정제될수록 증가합니다. 메달리온 아키텍처는 계보의 어느 시점(브론즈에서 골드까지)에 대한 데이터를 얼마나 오래 보관할지, 더 구체적으로 어떤 테이블을 고려할 때 유용한 도구입니다.

위의 다이어그램은 (표 g)라는 선별된 데이터 제품에 대한 소스 테이블과 변환 계보를 보여줍니다. 골드 데이터 자산에서 거꾸로 작업하면 실버 계층(df)을 통해 계보를 다시 추적하여 브론즈 데이터 자산(ac)으로 결론을 내리면서 테이블(개별적으로, af)의 가치가 감소하는 것을 볼 수 있습니다. 단일 테이블이 이전 6개의 테이블 모음보다 개념적으로 더 가치가 있는 이유는 무엇입니까?

간단히 말해서, 테이블 g에 대한 데이터 자산 종속성 컬렉션을 구축, 관리, 모니터링 및 유지하는 복잡성은 개별 부품보다 더 높은 비용을 나타냅니다.

브론즈 데이터 자산(ac)으로 표시되는 원시 데이터는 직접적인 다운스트림 데이터 소비자(df)가 액세스하고 추가로 정제, 결합 또는 일반적으로 활용하기 위해 필요한 동안만 생존할 것으로 예상되며, 골드 계층에서도 실버 계층 데이터 자산에 대해 동일한 기대가 이루어집니다. 즉, 필요한 동안에만 존재해야 하며 계보 체인 아래로 내려갈수록 데이터 품질이 단순화되고 전반적으로 향상된다는 것입니다..

엔드투엔드 계보에 대해 생각하는 유용한 방법은 데이터 제품의 렌즈를 이용하는 것입니다.

데이터 제품 및 데이터 자산과의 관계 데이터 제품1이라는 용어는 코드, 데이터 및 메타데이터는 물론 특정 선별된 데이터 제품을 구축, 생산 및 관리하는 데 필요한 논리적 인프라를 나타냅니다.  
아래 그림 8-3은 코드, 데이터, 해당 데이터에 대한 데이터(메타데이터) 간의 교차점과 데이터 제품을 실행하고 제공하는 인프라를 자세히 보여줍니다.

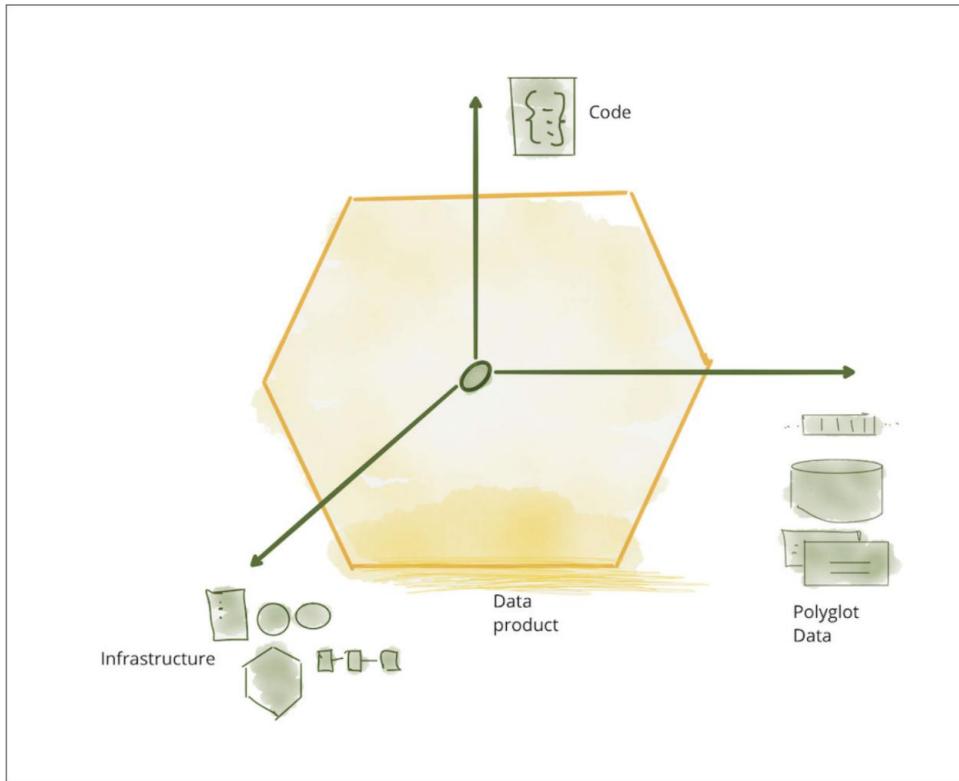


그림 8-3. 데이터 제품은 모든 부분의 합입니다.

데이터 제품에 대한 새로운 아이디어는 Zhamak Dehghani가 그녀의 [아키텍처 패러다임인 데이터 메시와 함께 소개했습니다](#). 그녀는 모든 선별된 데이터 제품이 특정 목적에 맞게 구축되어야 하고 다른 테이블에 대한 추가 조인 없이 사용할 수 있어야 한다는 규칙을 제안했습니다. 기본적으로 데이터 제품을 생산하는 데 드는 비용과 노력을 소비자를 대신하여 전액 지불되어야 합니다. 데이터 제품의

1 데이터 메시: 대규모로 데이터 기반 가치 제공 (<https://www.oreilly.com/library/view/data-mesh/9781492092384/>)

그 자체. 이 규칙은 또한 데이터 제품이 서비스에 연결되어 있고 서비스는 유용하고 목적에 맞는 데이터의 생산이라는 단순한 사실을 연결하는 데 도움이 됩니다.

논리적으로, 하나 이상의 데이터 자산 없이는 데이터 제품이 존재할 수 없다고 가정하는 것도 안전합니다. 따라서 데이터 자산과 데이터 제품에 관해 이야기할 때 궁극적으로 우리는 생성하는 데 필요한 애플리케이션과 워크플로우를 설계, 구축, 테스트, 출시, 모니터링 및 유지 관리하는 조직에 충분히 가치 있는 데이터에 대해 이야기하고 있습니다. 특정 데이터 제품을 kapsuh화하는 귀중한 데이터 자산 세트입니다. 이러한 수준의 엄격함과 운영에 대한 협신이 기존 소프트웨어 프로젝트의 종소리를 울리고 있다면 그것은 맞습니다.

고품질 데이터를 생성하려면 엔지니어가 표준 소프트웨어 개발 수명 주기(SDLC)를 따라야 합니다. 기본적으로 데이터 제품 수명 주기 동안 런타임 시 "놀라움이 없도록" 설계합니다.

## Lakehouse의 데이터 제품

점점 복잡해지는 종속성 그래프와 함께 대규모 데이터 수집 네트워크를 통해 점점 더 많은 양의 데이터를 생성하는 조직의 경향을 고려할 때, 레이크하우스가 매우 가치 있는 데이터 자산(스트리밍 또는 정적)의 수명 주기를 추적하기 위한 일반적인 기능을 제공하는 것이 매우 중요합니다.

이는 업스트림 종속성은 물론 다운스트림 데이터 자산 종속성을 포함한 데이터 자산의 메타데이터를 추적할 수 있음을 의미합니다. 이는 특히 데이터 블룸의 변화, 특정 소스 또는 테이블의 스키마 및 구조 수정, 생성되는 데이터 흐름에 대한 기타 고려 사항 및 기대 사항을 이해하고 대응해야 하는 다운스트림 소비자에게 중요합니다.

## 높은 신뢰도 유지

데이터 제품에 대한 암묵적인 신뢰를 유지하기 위해 각 데이터 제품과 관련된 데이터 계보의 통합, 데이터 품질 및 데이터 관측 가능성을 포함한 중요한 추가 데이터는 데이터 소비자가 계속 자신감을 갖고 신뢰할 수 있는 올바른 정보를 갖도록 보장하는 데 도움이 됩니다. 높은 신뢰 환경을 유지합니다.

한 걸음 물러서서 데이터 계보의 한 지점을 나타내는 델타 테이블에 포함된 데이터를 관리하는 데 필요한 도구, 워크플로(레이크하우스 오케스트레이션), 메타데이터, 프로세스, 아키텍처 원칙 및 엔지니어링 모범 사례를 고려해 보면 시스템과 서비스, 사용자와 페르소나, 데이터 분류 및 액세스 정책, 최고의 데이터 관리를 나타내는 선별된 데이터 제품 전반에 걸쳐 수집에서 삭제까지의 여정을 통해 우리는 현대 데이터의 우산인 크기와 규모를 빠르게 깨닫기 시작합니다. 통치.

## 데이터 자산 및 액세스

전통적인 데이터베이스 관리 초기에는 오늘날 데이터 거버넌스 조직에서 볼 수 있듯이 조직이 데이터를 효율적으로 수집, 수집, 변환, 카탈로그화, 태그 지정, 액세스 및 폐기하는 방법을 관리하는 방법을 전담하는 대규모 팀이 없었습니다. 데이터베이스에 대한 액세스는 데이터베이스 관리자의 손에 달려 있습니다. 그들은 사용자에게 권한을 부여하고, 비용이 많이 드는 쿼리를 실행하고, 데이터베이스가 계속 작동하도록 하는 "책임"을 맡았습니다.

사용자 또는 그룹이 실행할 수 있는 작업에 대한 거버넌스는 데이터 제어 언어(DCL), 데이터 정의 언어(DDL) 및 데이터 조작 언어(DML)와 같은 SQL 구문 그룹을 사용하여 권한을 통해 관리됩니다.

## 데이터 자산 모델

레이크하우스와 관련된 리소스의 거버넌스는 일반적으로 데이터 자산으로 알려진 관리 가능한 개체에 대한 정책 간의 관계를 설명합니다. 가장 단순한 전통적인 의미에서 데이터 자산은 TABLE 또는 VIEW이고 정책은 GRANT 권한입니다. 테이블 리소스가 포함된 데이터베이스 또는 스키마도 데이터 자산입니다. 정책은 보안 주체라고 하는 사용자에게 데이터 자산(데이터베이스, 테이블 또는 뷰)에 대한 작업(표시 또는 선택)을 실행할 수 있는 액세스 권한을 부여하기 때문입니다.). 보안 주체가 리소스에 대한 작업을 실행하려면 먼저 데이터 자산을 생성해야 합니다.

이 데이터 자산 모델은 [그림 8-4](#)에 나와 있으며 공통 SQL 권한을 통해 액세스 및 사용 제어가 필요한 모든 보안 개체와 관련될 수 있습니다.

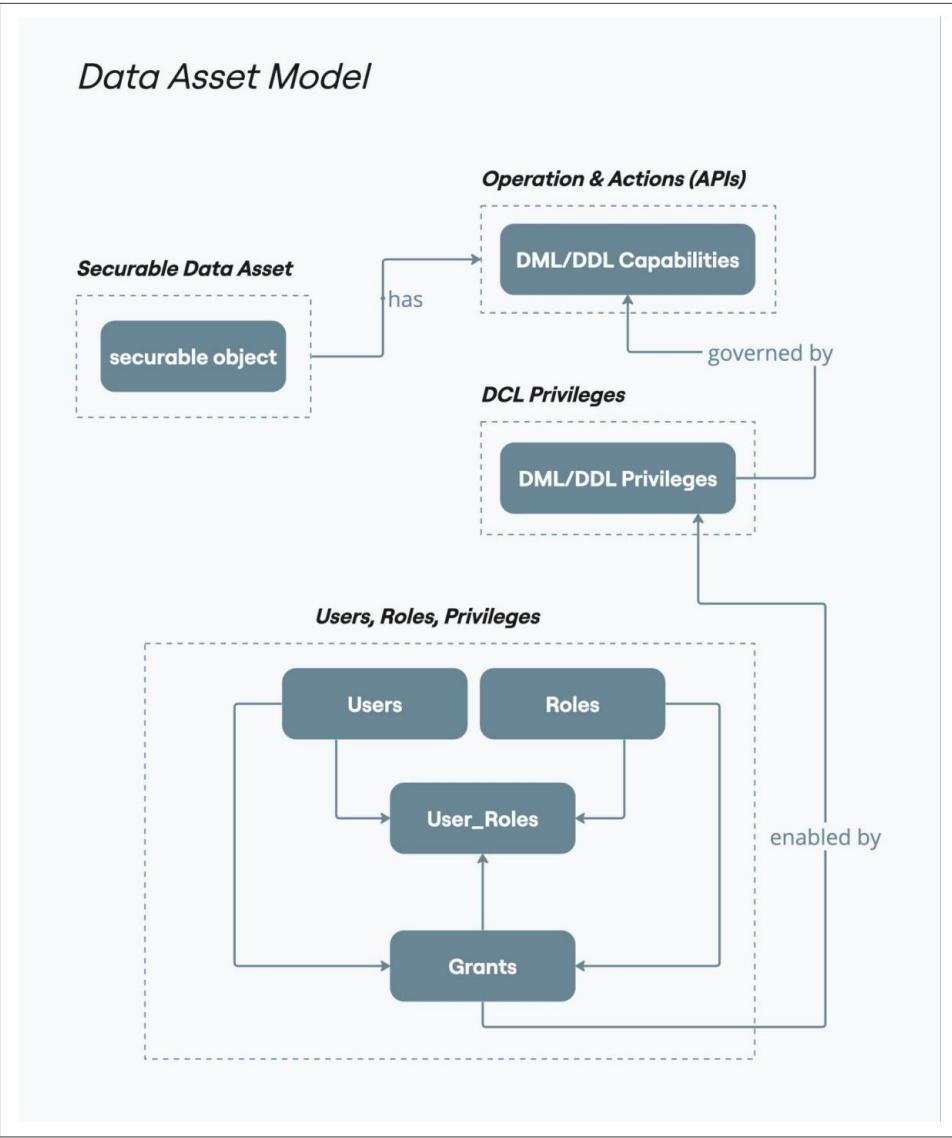


그림 8-4. 데이터 자산은 일반적으로 액세스 및 일반 사용을 승인하는 하나 이상의 권한 집합이 필요한 보안 개체로 정의될 수 있습니다.

주체가 데이터 자산에 대해 실행할 수 있는 일련의 작업 및 작업은 다음을 포함하는 데이터 정의 언어(DDL)에 포함됩니다.

CREATE, ALTER, DROP 작업과 INSERT 및 UPDATE 작업을 가능하게 하는 데이터 조작 언어(DML)를 통해 다음 중 하나를 실행하는 기능

더 많은 작업과 작업은 GRANT 및 REVOKE 문을 통해 DCL(데이터 제어 언어)을 사용하여 관리됩니다.

오늘날 데이터 자산은 상호 작용 방법을 관리하는 액세스 및 사용 제어(권한 부여) 정책이 필요한 다른 리소스(예: 대시보드, 쿼리(대시보드를 구동함), 노트북, 기계 학습)도 캡슐화하도록 발전했습니다. 모델 등.

### SQL 권한 부여를 통한 데이터 액세스 관리 SQL과 유

사한 시스템 내의 작업 관리는 DCL, DDL 및 DML의 문법을 통해 처리됩니다. 다음은 이러한 기능에 대한 간략한 정보입니다.

#### 데이터 제어 언어(DCL)

이 특수 구문은 SQL 유사 시스템 내에서 액세스 관리에 사용됩니다. GRANT 및 REVOKE 작업을 사용하면 승인된 작업(권한) 집합이 USER 또는 GROUP 집합과 연결되어 DDL 및 DML로 정의된 작업을 실행하거나 이전에 부여된 하나 이상의 권한을 제거할 수 있습니다.

GRANT 권한 구문은 데이터베이스 유형에 따라 다르지만 일반적으로 공통 ANSI-SQL 표준 구문을 지원합니다.

```
% GRANT priv_type [(열 목록)]
  ON [객체_유형]
  사용자_또는_역할, [사용자_또는_역할]
```

사용자나 그룹이 수행할 수 있는 작업을 제어하는 것은 단순히 추가되는 것이 아닙니다. 대부분의 경우 권한은 제한된 시간 동안만 부여된 후 다시 제거됩니다. 임시 권한 부여 요구 사항을 충족하기 위해 REVOKE 구문을 통해 권한 제거 기능이 활성화됩니다.

```
% REVOKE priv_type [(열 목록)]
  ON [객체_유형]
  사용자_또는_역할, [사용자_또는_역할]
```

#### 데이터 정의 언어(DDL)

이 구문은 CREATE, ALTER, DROP, COMMENT 및 RENAME과 같은 표준 작업을 제공합니다. 우리는 Delta scala, Python 및 Rust 컴파니언 라이브러리를 사용하여 간접적으로나 직접적으로 DDL을 실제로 사용했습니다. 6장에서 우리는 테이블을 생성 및 변경하고, 열에 대한 주석을 수정하고, 작업이 끝나면 테이블을 삭제하는 방법도 배웠습니다.

#### CREATE 구문

```
% CREATE [OR REPLACE] TABLE [IF NOT EXISTS] 테이블_이름( [열_이름, 유
  형, ...]
  ) DELTA
  TBLPROPERTIES 사용('키'='값')
  클러스터 기준(...)
```

## ALTER 구문

```
% ALTER TABLE 테이블_이름
    TBLPROPERTIES 설정('키'='값')

% ALTER TABLE 테이블_이름 ADD
    COLUMNS ( [열_
        이름, 유형, ...],_
    )
```

### 데이터 조작 언어(DML)

이 구문은 사용자 또는 그룹이 다음 표준 작업 SELECT, INSERT, UPDATE 및 DELETE를 사용하여 리소스에 대해 실행할 수 있는 작업을 제어하는 권한을 제공합니다.

% [테이블 또는 내부 선택]에서 [열]을 선택합니다.  
 [여기서], [그룹화], [갖고], [순서 기준], [제한]

Together DCL을 사용하면 DDL을 사용하여 생성된 리소스와 DML을 통해 활성화된 작업이 관리하는 작업의 일부 또는 전체를 실행할 수 있도록 사용자 또는 그룹에 권한을 할당할 수 있습니다.

데이터 작업의 규모와 규모가 전 세계적으로 계속해서 증가하고 있지만, 간단한 GRANT 및 REVOKE 권한을 사용하여 데이터 자산의 액세스와 권한 부여를 모두 제어하는 패러다임은 여전히 통합 거버넌스 전략을 채택하는 가장 간단한 경로입니다. 단순히 SQL을 사용하지 않는 시스템 및 서비스와의 상호 운용성을 고려하기 시작하면 거의 즉시 문제가 발생합니다.

## 데이터 웨어하우스와 데이터 웨어하우스 간의 거버넌스 통합 호수

이전 섹션에서 우리는 GRANT 및 REVOKE 문을 사용하여 특정 리소스에 대한 액세스를 허용하거나 거부하는 기능을 활성화하는 SQL 데이터베이스용 DCL 구문을 추가하면서 기존 데이터 거버넌스 기능이 시작되었다는 사실을 발견했습니다.

권한 부여를 함께 사용하면 사용자 또는 역할과 관련된 특정 권한을 부여하여 보안 리소스(데이터 자산)에 대한 일련의 작업을 실행할 수 있습니다. DCL을 사용한 액세스에 대한 거버넌스는 MySQL 및 Postgres와 같은 기존의 사일로화된 데이터베이스(RDBMS)뿐만 아니라 Databricks, AWS 및 Snowflake와 같은 공급업체를 통한 대부분의 최신 데이터 웨어하우스에서도 작동합니다.

그러나 우리 레이크하우스의 거버넌스에 기존 SQL 보조금을 사용하는 데에는 문제가 있습니다. 문제는 모든 시스템과 서비스가 SQL을 이해하는 것은 아니라는 점입니다. 설상가상으로 우리는 단순히 하나의 거버넌스 모델을 사용하여 모든 데이터 자산을 보호할 수 있는 능력이 없습니다.

호수가에는 여전히 표면 바로 아래에 전통적인 데이터 레이크가 있다는 단순한 사실을 생각해 보십시오. 이는 레이크ハウス에 대한 거버넌스를 통합하기 위해 SQL과 유사한 인터페이스를 제공하기 위해 기본 데이터에 대한 권한 및 액세스 모델을 해결해야 함을 의미합니다.

권한 관리 호수집 표면 바로 아래에는 데이

터 호수가 있습니다. 지금까지 우리 모두 알고 있듯이 데이터 레이크는 기존 파일 시스템의 기본 요소를 사용하여 원시 데이터 구성을 지원하는 데이터 관리 패러다임입니다. 대부분의 경우 클라우드 개체 저장소가 사용되며 이러한 탄력적 파일 시스템의 루트는 버킷입니다.

버킷은 클라우드 객체 저장소 내에서 표준 파일 시스템과 유사한 구조를 나타내는 리소스 루트 "/"를 캡슐화합니다. 그림 8-5는 버킷을 구성 부분으로 분류한 것을 보여줍니다. 예를 들어 루트 바로 옆에는 최상위 디렉토리(경로 및 파티션)와 해당 기본 파일이 있습니다.

**Take the following example of a layout of a cloud storage container:**

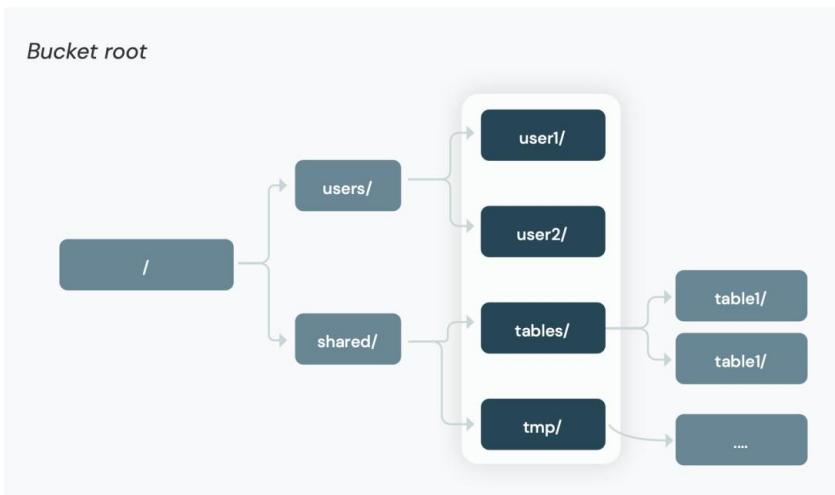


그림 8-5. 데이터 레이크는 일반적으로 클라우드 개체 저장소를 사용하여 구축됩니다. 이러한 컬렉션의 기본 요소는 파일 시스템의 루트 또는 버킷에서 시작하여 디렉토리와 파일의 하위 컬렉션 또는 추가 디렉터리에 걸쳐 순서대로 내려갑니다.

각 디렉터리에는 일반적으로 로그 파일, 이미지, 비디오 또는 구성(속성)과 같은 공유 가능한 자산에서 볼 수 있는 구조화되지 않은(원시) 데이터 모음이 포함되어 있습니다.

파일, yaml, json 등) 및 라이브러리(jar, whl's, egg)뿐만 아니라 구조화되었지만 처리되지 않은 데이터도 포함됩니다.



구조화된 데이터의 경우 apache **avro** 와 같이 잘 알려진 행 기반 형식을 사용하는 것이 좋습니다. 또는 Google의 **protobuf**, 또는 Apache **Parquet** 와 같은 열 기반 형식, Delta 유ти리티를 사용하여 Delta의 테이블 형식으로 간단하게 변환할 수 있습니다<sup>2</sup>

, 아래 그림과 같이.

```
% from delta.tables import * deltaTable
    =DeltaTable.convertToDelta(spark, "par-
quet.`<테이블 경로>`")
```

다른 모든 유형의 비정형 및 정형 데이터 외에도 데이터 레이크는 관리형(또는 비관리형) 델타 테이블을 저장합니다. 따라서 우리는 레이크하우스 표면 바로 아래에 있는 데이터 레이크에 숨겨진 다양한 종류의 파일을 저장하고 있습니다.

무단 액세스로부터 기본 파일 시스템을 보호하는 방법을 이해하는 것은 Lakehouse 거버넌스에 매우 중요하며 운 좋게도 SQL과 유사한 권한은 기존 운영 체제(OS) 파일 시스템 권한과 유사한 데이터 관리 패러다임을 공유합니다. 사용자, 그룹(역할과 유사) 및 읽기, 쓰기 및 실행 작업을 부여하는 권한을 사용하여 제어됩니다.

### 파일 시스템 권한 노트북을 실행

하는 OS 또는 우리가 프로비저닝한 서버에서 원격으로 유사한 액세스 및 위임 패턴을 공유합니다. 예를 들어, OS는 단기 및 장기 프로세스(작업) 사이에서 유한 리소스(컴퓨팅, RAM, 스토리지)의 배포를 감독합니다. 각 프로세스는 그 자체로 명령(작업) 실행의 결과이며 실행은 사용자, 그룹 및 권한 집합과 연결됩니다. 이 모델을 사용하여 OS는 간단한 거버넌스 규칙을 구성할 수 있습니다.

실제적인 예로 ls 명령을 살펴보겠습니다 .

```
% ls -lah /lakehouse/bronze/
```

명령의 출력은 파일 시스템 리소스(파일, 디렉터리) 및 해당 메타데이터 목록입니다. 메타데이터에는 리소스 유형(파일 또는 디렉터리), 액세스 모드(권한), 참조(이 리소스에 의존하는 리소스), 소유권(사용자), 그룹 연결은 물론 파일 크기, 마지막 수정 날짜 및 파일이 포함됩니다. 또는 디렉토리 이름.

### 파일 유

형 단일 문자로 표시됩니다. 파일은 - 로 표시되고 디렉터리는 d로 표시됩니다 .

---

<sup>2</sup> <https://docs.delta.io/latest/delta-utility.html#convert-a-parquet-table-to-a-delta-table>

## 권한

읽기(r), 쓰기(w), 실행(x)을 포함합니다. 권한은 리소스 소유자, 특정 액세스 그룹 및 마지막으로 다른 사람(기타)에 대해 별도로 관리됩니다.

### 참조 리소스

에 연결된 다른 리소스 수를 추적합니다.

### 소유자

각 리소스에는 소유자가 있습니다. 소유자는 OS에서 알려진 사용자입니다. 리소스 소유자는 그룹 수준 권한 할당을 통해 다른 사용자와 프로세스가 상호 작용하는 방식을 완전히 제어할 수 있습니다.

### 그룹 사용

자는 하나 이상의 그룹과 연결됩니다. 그룹을 사용하면 여러 사용자와 프로세스가 특정 권한을 제한하면서 함께 작업할 수 있습니다. 운영 체제 컨텍스트 내의 그룹은 데이터 웨어하우스 컨텍스트 내의 역할과 유사합니다. 각 리소스에 대해 특정 그룹에 권한(리소스 소유자 외부)을 부여할 수 있으며, 알 수 없는 그룹 구성원의 경우 기본 권한도 적용할 수 있습니다.

모든 것이 합쳐지면 파일 시스템 권한 간의 연관성과 이것이 레이크하우스의 거버넌스에도 적용될 수 있는 방법을 확인할 수 있습니다. 다음 예를 보면 출력에서 ecomm\_aggs\_table에 대해 무엇을 알 수 있습니까?

```
% ls -lh /lakehouse/bronze/ecomm_aggs_table/
drwxr-x---@ 338 dataeng eng_analysts 11K Oct 23 12:53 _delta_log
drwxr-x---@ 130 dataeng eng_analysts 4.1K Oct 23 12:34 날짜=2019-10-01
drwxr-x---@ 130 dataeng eng_analysts 4.1K 10월 23일 12:34 날짜=2019-10-02
```

먼저 \_delta\_log 디렉터리는 우리가 델타 테이블을 보고 있음을 알려줍니다. 이는 전체 읽기-쓰기-실행 권한(rwx)을 가진 dataeng이라는 사용자가 소유합니다.

또한 eng\_analysts 그룹은 테이블을 읽고 실행할 수 있지만 읽기 전용 액세스 권한이 부여된 경우 테이블을 수정할 수는 없습니다. OS의 다른 사용자의 경우 이 경로에서 파일과 상호 작용을 시도하는 동안 예외(승인되지 않음)가 발생합니다.

유사한 권한 모델을 클라우드 개체 저장소에도 적용할 수 있습니다. 주요 차이점은 사용자를 식별하고 그룹을 관리하는 방식입니다.

### 클라우드 개체 저장소 액세스 제어 데이터 레이크의 저장과

계산을 분리하면 데이터 자산의 위치와 계산 프로세스를 실행하는 서버 사이의 물리적 경계가 보장됩니다. 우려 사항의 분리를 더 자세히 살펴보면 기존 OS 수준 사용자 권한 모델에서도 추가로 단절된다는 사실도 발견할 수 있습니다.

로컬 컴퓨팅 프로세스에 바인딩된 사용자(ID)는 실제로 특정 작업을 실행할 수 있다는 원격 프로세스에 신호를 보내는 키(또는 토큰)를 추가하지 않고는 개체 저장소에 직접 알려지지 않습니다. 이 키는 요청을 식별하고 원격 실행(읽기, 쓰기 또는 삭제) 형태로 요청된 작업을 허용하거나 거부하는 간단한 규칙 세트를 승인하는 데 도움이 됩니다.

공유 운영 체제가 없는 경우, 우리는 ID를 활용하여 데이터를 처리하는 위치(컴퓨팅)와 데이터를 저장하는 위치 사이에 신뢰 관계를 구축합니다.

신원은 다음 질문에 답하는 데 도움이 됩니다.

- 특정 런타임 프로세스의 ID(사용자)는 무엇이며 이는 기존 사용자 권한 모델에 어떻게 적용됩니까?
- 하나 이상의 클라우드 기반 리소스에 대한 액세스를 활성화하려면 어떻게 해야 합니까?
- 식별된 후에는 어떤 방법으로 특정 작업 및 운영을 승인할 수 있습니까?  
특정 사용자에게 발생합니까?

패러다임은 기존 파일 시스템 권한(사용자, 그룹, 권한)에서 ID 및 액세스 관리, 줄여서 IAM이라는 보다 유연한 시스템으로 전환됩니다.

신원 및 액세스 관리 문을 두드리는 소리가 들

리는 경우. 당신은 대답합니까? 낯선 사람을 들여보내 주시겠습니까? IAM이 존재하는 전체 이유는 알려지지 않은 엔터티(그들이 누구라고 말할 수 있음)와 내부 시스템 사이에 상호 신뢰 기반 관계가 있는지 확인하는 것입니다. 그렇다면 동적 클라우드 기반 세계에서 사용자, 시스템 또는 서비스를 어떻게 식별합니까?

신원. 각 ID는 사용자(사람) 또는 서비스(API, 파일라인 작업, 태스크 등)를 나타냅니다. ID는 개별 사용자뿐만 아니라 농담으로 헤드레스 사용자라고 불리는 서비스 주체를 모두 캡슐화합니다. 이들은 인간은 아니지만 여전히 사용자를 대신하여 작업을 수행하는 시스템을 나타내기 때문입니다. 신원은 여권과 같은 역할을 하여 사용자의 적법성을 인증합니다. 또한 ID는 정책을 통해 사용자를 일련의 권한에 연결하는 데 사용됩니다.

개별 사용자와 장기 토큰 모두에 대해 발급된 액세스 토큰과 서비스 주체에 대해 발급되는 인증서(인증서)를 보는 것이 일반적입니다.

입증. 신원은 합법적일 수 있지만 인증의 핵심은 테스트를 통해 절대적으로 확실하다는 것입니다. 대부분의 시스템은 특정 기간 동안만 키나 토큰을 발행(생성)합니다. 이렇게 하면 신원을 수시로 재인증하여 여전히 합법적이라는 것을 증명해야 합니다. 악의적인 행위자(해커, 스푸피)가 불법적인 목적으로 끌어올린 토큰을 재사용하려는 경우 낮은 TTL

токен에서는 도난당한 신원의 잠재적인 영향을 제한합니다. 일반적으로 시스템의 보안이 강화될수록 토큰의 TTL은 낮아집니다.

권한 부여. 신원 및 인증 메커니즘이 함께 결합되어 사용자가 단순한 사기꾼이 아니라는 것을 보장합니다. 이 두 가지 개념은 인증 프로세스와 긴밀하게 연결되어 있습니다. 승인은 GRANT 권한과 유사합니다. 우리는 사용자가 우리 리소스의 물리적 위치에 접근할 수 있었기 때문에(키, 인증서, 또는 레이크하우스의 데이터 자산에 액세스하기 위한 토큰). 인증 프로세스는 사용자가 특정 시스템 내에서 수행할 수 있는 작업을 설명하는 정책 파일 집합과 사용자 사이를 연결하는 다리입니다.

액세스 관리. 간단히 말해서 액세스 관리는 데이터에 대한 액세스를 제어하고 보안 점검 및 균형을 시행하는 방법을 제공하는 것이며 거버넌스의 초석입니다. 액세스 제어는 특정 리소스(데이터 자산, 파일, 디렉토리, ML 모델)에서 어떤 종류의 작업과 작업을 실행할 수 있는지 식별하는 수단을 제공하고 정책에 따라 승인 또는 거부할 수 있는 기능을 제공합니다.

사용자(ID) 생성, 자격 증명(토큰) 발급, 리소스에 대한 액세스 인증 및 권한 부여의 전체 프로세스는 실제로 GRANT 메커니즘과 다르지 않습니다. 그 반대는 REVOKE이며 활성 자격 증명을 무효화합니다.

전체 액세스 수명주기를 완료하는 ID를 제거하는 기능이 없으면 프로세스가 완료되지 않습니다.

IAM은 ID 및 액세스 정책을 사용하여 레이크하우스에 GRANT와 유사한 권한 관리를 구현할 수 있는 누락된 기능을 제공합니다.

다음 섹션에서는 액세스 정책을 살펴보고, 역할 기반 액세스 제어가 페르소나(또는 행위자)를 사용하여 데이터 액세스 관리를 단순화하는 데 어떻게 도움이 되는지 알아보고, 정책을 생성하고 사용하는 방법에 대해 알아봅니다. 임호.

### 데이터 보안 거버넌스

스토리에는 여러 측면이 있으며, 솔루션을 효과적으로 확장하려면 먼저 설정하거나 기존 솔루션에 신중하게 통합해야 하는 중요한 규칙과 작업 방식이 있습니다.

예를 들어, “오리처럼 보이고, 오리처럼 헤엄치고, 오리처럼 팩팩거린다면 아마도 오리일 것입니다”라는 오리 테스트에 익숙할 것입니다. 이는 익숙하지 않은 것에 대해 추론하고 이를 “우리에게 알려진” 범주로 분류하는 능력을 의미합니다. 레이크하우스 내에서 활동하는 다양한 페르소나 또는 행위자와 관련하여 수정된 오리 테스트를 사용하여 경로의 첫 번째 단계로 누가 어떤 데이터 자산에 대한 액세스 수준을 가지고 있는지 식별하는 제한된 수의 역할을 만들 수 있습니다. 인증을 위한 보다 복잡한 정책 생성.

### 역할 기반 액세스 제어

RBAC(역할 기반 액세스 제어)는 시스템 액세스를 승인 또는 거부하고 조직 내 역할을 사용하여 특정 인물의 임무를 수행하는 데 필요한 리소스에 대한 권한 하위 집합을 승인하는 데 사용됩니다.

레이크하우스의 경우 일상 업무(엔지니어, 과학자, 분석가, 비즈니스 기능)에서 수행하는 역할, 우리가 속한 팀 및 조직, 비즈니스의 논리적 구분선(도움이 될 수 있음)을 고려하십시오. 데이터 도메인 설정, 시스템, 서비스 및 데이터 제품(개발, 단계, 프로덕션)을 위한 런타임 환경, 마지막으로 관리하고 액세스하는 데이터 분류(모든 액세스, 제한, 민감, 매우 민감). 역할이 의인화하는 경계가 때로는 약간 모호할 수 있으므로 RBAC의 R은 자원을 의미할 수도 있습니다.

이야기를 가지고 RBAC에 접근해 봅시다. 이야기를 위해 우리 모두는 Complete Foods라는 지역 유기농 농산물을 판매하는 글로벌 식료품 체인에 고용되어 있습니다. Complete Foods는 배송 목적으로 실제 매장은 물론 온라인에서도 제품을 판매합니다.

각 매장은 특정 지역에서 운영되며 쇼핑 트렌드는 지역적, 지역적, 계절적 요인에 따라 다릅니다. 이는 모든 매장이 동일한 회사 내에서 운영되고 동일한 공통 제품의 대부분을 공유하지만 지역별 재고 차이, 공급업체 관계, 판매 및 고객은 매장이 위치한 세계 위치에 따라 달라질 수 있음을 의미합니다.

그러나 레이크하우스 데이터에 대한 액세스가 필요한 직원의 역할과 책임은 기본적으로 동일하게 유지되며 액세스는 필요와 이유(사용 사례)에 따라 관리되고 데이터 개인 정보 보호, 거버넌스 및 주권에 대한 필수 교육을 받은 후에도 관리됩니다. 기밀 데이터에 대한 액세스 또는 마케팅 및 광고 캠페인에 필요한 고객 데이터에 액세스할 때.

역할은 사람이 아닙니다. 각 역할은 사람이나 서비스가 맡을 수 있습니다. 간단하게 시작하여 누가, 무엇을, 어디서, 어떻게 분류하는 것이 중요합니다.

### 페르소나에 대한 역할 설정 조직 내 공

통 페르소나와 관련된 역할을 추상화하면 누가, 무엇을, 어디서, 어떻게, 왜를 이해하는 것이 단순화됩니다. 일반적인 조직 내에서 페르소나에 대한 몇 가지 일반적인 구분선을 살펴보겠습니다.

### 엔지니어링 역할 허

드리는 사용자를 포함하여 하드웨어, 소프트웨어, 보안, 플랫폼, 데이터 및 기계 학습 전반에 걸쳐 모든 개발자 역할에 적용될 수 있습니다. 책임에는 POS(매장 및 온라인) 시스템 유지 관리, 모바일 애플리케이션, 이벤트 데이터 정의, 데이터 캡처 및 수집 네트워크 구축, 신용 카드 및 사용자 집 주소와 같은 개인 데이터 처리 및 학습이 포함됩니다. 고객의 쇼핑 습관을 바탕으로 연중 적절한 시기에 적절한 지역에서 적절한 제품을 구매할 수 있도록 보장합니다.

액세스 패턴: 모두(읽기, 읽기-쓰기, 관리자)

역할 이름: role/developerRole

#### 분석가 역할 비즈

니스 분석가 또는 전문가를 포함합니다. 중요한 비즈니스 운영을 정확하게 포착하기 위해 올바른 데이터를 사용할 수 있도록 비즈니스 및 엔지니어링과 협력하고 호박 항신료 제품을 선반에 다시 옮겨놓을 때와 같은 통찰력 생성을 통해 의사 결정 프로세스를 지원하는 일을 담당합니다. 그리고 어떤 지역에서 어떤 종류의 비유제품 우유를 계속 제공할지에 대해 알아보세요.

액세스 패턴: 주로 데이터에 대한 읽기 전용입니다. 쿼리를 생성 및 공유하고, 대시보드를 구축하고, 기록 데이터 또는 새로운 추세를 분석하는 기능을 갖추고 있습니다.

역할 이름: role/analystRole

#### 과학자 역할

데이터 및 행동 과학자를 포함합니다. 엔지니어 및 분석가와 협력하여 권장 사항 및 기타 추론 모델을 강화하기 위해 적절한 데이터가 적시에 적절한 장소로 흐르도록 하는 일을 담당합니다.

액세스 패턴: 주로 읽기 전용이며, 모델 교육을 강화하고 테스트 및 실험을 위한 결과를 캡처하는 테이블을 생성하는 기능이 있습니다.

역할 이름: role/scientistRole

#### 비즈니스 역할

관리자, 이사, 인사, 심지어 리더십까지 포함됩니다. Complete Foods 브랜드를 구축하고 유지하는 데 책임이 있습니다. 지역 매장 관리자나 구매자뿐만 아니라 현지 및 글로벌 책임도 있으며, 모든 사람은 판매 수치, 예측, 지원과 관련된 데이터 하위 집합 또는 사업 부문 외부의 우려 사항에 액세스해야 합니다. 또한 엔지니어링 리더십에는 엔지니어링 관리자 및 이사와는 다른 접근 권한과 역량이 필요합니다.

액세스 패턴: 대부분 읽기 전용입니다. HR에서는 직원을 생성, 수정, 삭제해야 할 수도 있습니다.

역할 이름: role/businessRole

레이크하우스의 특정 데이터 자산(리소스)에 대한 액세스 권한을 부여하는 프로세스는 다음 요소를 통합하여 결정할 수 있습니다.

- 사용자의 역할 및 책임(누구 및 무엇)
- 리소스 위치(버킷 및 접두사)(어디)
- 사용자가 운영하는 환경(어디)(개발, 단계, 프로덕션)

- 데이터 분류(일반적으로 사용 가능, 제한, 민감 등)  
(무엇)
- 작업(작업): 읽기(보기), 수정(읽기, 쓰기) 또는 관리(읽기, 쓰기,  
생성 또는 삭제)를 (방법)으로

그러므로 우리는 누가, 무엇을, 어디서, 어떻게, 그리고 만약을 염두에 두어야 한다는 것을 항상 기억해야 합니다.

“주어진 ID(누구)에 대한 액세스 권한을 부여한다면, 주어진 작업 세트를 수행하기 위해 (무엇) 작업이 필  
요한지(어떻게), 그리고 어떤 환경에서(어디에서) 사용자 수준 액세스가 필요한지 vs. 헤드리스 사용자 액세  
스? 마지막으로, 읽기 수준과 읽기-쓰기 수준 액세스 권한을 부여하는 데에는 어떤 잠재적인 위험이 있습니  
까?”

또한 액세스 권한 부여 여부에 대한 고려 사항 외에도 항상 염두에 두어야 할 또 다른 질문은 ID가 테이블  
에 있는 모든 데이터를 볼(읽을) "허용" 되는지 여부입니다. 데이터 액세스에 대한 보안 및 개인 정보 보호  
고려 사항을 기반으로 데이터를 그룹으로 나누는 것이 일반적입니다.

다음에는 데이터 분류 패턴을 살펴보겠습니다.

#### 데이터 분류

다음 분류자는 데이터 레이크의 특정 위치에 있는 리소스 내에 어떤 종류의 데이터가 저장되어 있는지 식별  
하는 유용한 방법입니다.

간단한 추상화로 신호등 패턴(녹색, 노란색, 빨간색) 측면에서 데이터 분류를 생각해 보겠습니다. 정지 신  
호등은 운전자에게 계속(녹색), 감속(노란색), 정지(빨간색)하라는 신호를 보냅니다. 비유하자면, 데이터  
자산에 대한 액세스 관리를 고려할 때 신호등 패턴은 녹색, 노란색 또는 빨간색일 수 있는 데이터에 태그를  
지정(또는 레이블 지정)(식별)하는 간단한 정신적 모델을 제공합니다.



#### 분류가 의심스러울 때

모든 조직은 다양한 종류의 데이터를 처리합니다. 의심스러울 경우 특정 데이  
터 세트(테이블, 원시 데이터 등)가 대중에게 유출될 경우 회사에 미치는 피해  
를 생각해 보세요. 개인 사용자 데이터에 적용되는 엄격한 법률을 고려하면 일  
부 항목은 자동으로 노란색 또는 빨간색으로 분류됩니다. 다양한 데이터세트  
로 더 많이 작업할수록 무엇이 적절한지 직관하기가 더 쉬워집니다.

예를 들어, 리소스가 민감한 데이터를 유출하지 않도록 적절한 검사가 이루어지면 "그린"으로 분류된 데이  
터에 대한 액세스를 자동화할 수 있습니다. "친환경"에 대한 실제적인 예는 USGS(미국 지질 조사국)에서  
일반적으로 제공되는 지진 및 위험 데이터입니다.

"노란색" 또는 "빨간색"으로 분류된 데이터에 액세스하려면 피부여자는 액세스 권한이 있는 사람, 액세스가 필요한 이유, 액세스가 필요한 기간, 액세스가 조직에 이익이 되거나 해를 끼칠 수 있는 방법을 고려해야 합니다. 확실하지 않은 경우 항상 if를 고려하십시오. 이 데이터에 대한 액세스 권한을 부여하면 수혜자가 올바른 일을 할 것이라고 신뢰합니까?

규칙과 일반적인 작업 방식을 설정하면 데이터를 공통된 방식으로 분류하여 의사 결정 프로세스를 과학적인 프로세스로 줄이는 데 도움이 될 수 있습니다.

일반 액세스. 분류에서는 데이터가 일반 청중에게 제공된다고 가정합니다. 예를 들어, Complete Foods가 Instacart, UberEats, Doordash와 같은 서비스를 통해 재고 데이터에 액세스할 수 있게 함으로써 더 많은 식료품을 판매할 수 있다고 믿고 있다고 가정해 보겠습니다. 등록하고, 토큰을 받고, 델타 공유 엔드포인트에 접속하는 등 개방형 액세스를 활성화함으로써 모든 외부 조직이 읽기 전용으로 제한된 일반 액세스 역할과 연결된 특정 테이블에 액세스할 수 있도록 할 수 있습니다.

신호등 패턴: 녹색 수준 접근.

제한된 액세스. 분류에서는 데이터가 알 필요(사용) 기준에 따라 승인된 읽기 전용이라고 가정합니다. 이전의 완전 식품 사례를 계속하면, 재고 데이터에 대한 외부 액세스(일반 액세스 분류를 통해)는 상호 이익이 되는 관계를 통해 식료품 사업과 브랜드의 범위를 확장할 수 있지만, 남아 있어야 하는 경쟁 우위를 나타내는 데이터가 있습니다. 내부 전용이거나 외부 도메인으로 제한됩니다.

예를 들어, 공개된 제품당 가격(매장 및 파트너 서비스를 통해)이 있지만 특정 제품을 구입하는 데 드는 실제 실제 비용을 나타내는 내부 가격도 있다고 가정해 보겠습니다. 대부분의 경우 마진(상품 구입 비용과 판매 당시 가격 사이의 델타)은 우리가 광고하고 싶은 것이 아니며 경쟁 우위를 나타낼 뿐만 아니라 실제 비용이 많이 드는 가격 협상을 나타냅니다..

신호등 패턴: 녹색, 노란색 또는 빨간색 수준 접근.

민감한 액세스. 분류는 민감한 데이터에 적용됩니다. 민감한 데이터는 유출되면 조직에 해를 끼칠 수 있지만 신용 카드 번호, 주민등록번호, 의료 또는 건강 정보(HIPPA 데이터 규정 준수 문제를 일으킬 수 있음)와 같은 중요한 정보를 포함하지 않습니다. 민감한 데이터에는 사용자의 이름과 성, 주소, 이메일 주소, 생일, 공급업체에 대한 정보(예: 농산물을 구매하는 농장) 및 비즈니스 운영과 관련된 기타 데이터와 같은 개인 식별 정보(PII)가 포함될 수 있습니다. Sensitive-Access는 데이터 자산에 신용 카드 번호, 사회 보장 제도 또는 급여 정보가 포함되어 있지 않음을 인정한다는 점을 명시하는 것이 중요합니다.

민감한 데이터에는 사용자 이름, 주소 또는 기타 PII를 노출하지 않고 소비자 행동 데이터와 같은 정보가 포함될 수도 있습니다. 일별 집계 데이터의 경우

예를 들어, 분기별 수치가 모두 일정 수준인 경우 누출되면 피해를 입게 됩니다.  
추세가 실제 대비 백분율로 표시되는 경우에도 하향 추세입니다.

신호등 패턴: 노란색 또는 빨간색 레벨 접근.

매우 민감한 액세스. 이 분류는 가장 매우 민감한 데이터에 적용됩니다.  
조직과 사용자가 사용할 수 있습니다. 여기에는 직원 급여 정보가 포함됩니다.  
회사 재무 기록, 사용자 신용 카드 데이터 및 의료 데이터,  
집 주소 등. 이러한 데이터 자산에 액세스하려면 일반적으로 내부  
교육을 완료하고 액세스와 관련된 전체 감사 추적을 완료해야 합니다. 많은  
이 데이터는 전통적으로 인사(HR) 및 급여용으로 예약되어 있습니다.  
비즈니스 내의 특정 행위자.

신호등 패턴: 적색 레벨 접근.

이제 데이터 액세스(개발)와 관련된 기본 페르소나와 역할을 식별했습니다.  
perRole, analyzeRole, ScientistRole, businessRole) 및 일반 분류자  
당사의 데이터(일반 액세스, 제한된 액세스, 민감한 액세스, 매우 민감한 액세스),  
IAM과 데이터 액세스 정책 간의 점 연결을 마치고 마무리하겠습니다.  
코드형 정책에 대한 간략한 소개와 함께

## 조직의 성공을 위해 접두사 패턴 사용

S3 버킷 및 정책과 관련하여 우리가 할 수 있는 가장 유용한 것 중 하나는

우리가 해야 할 일은 데이터 레이크를 구성하는 데 "미리" 시간을 들여 데이터 레이크를 구성하는 방법을 단순화하는 것입니다.  
일반적으로 S3 버킷을 설정하고

창고 디렉토리를 확인하고 팀이 올바른 일을 하기를 바랍니다. 오히려 사전작업을  
원활한 런타임 실행에 필요한 주요 패턴 설정을 포함해야 합니다.

환경, 최상위 카탈로그, 다양한 데이터베이스(스키마), 기본  
테이블은 물론 데이터 애플리케이션과 해당 메타데이터, 라이브러리를 위한 전용 공간,  
구성.

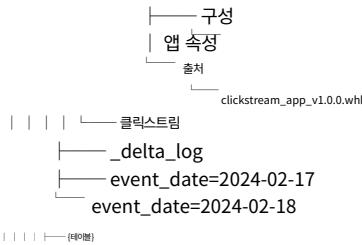
**예제 8-1**에서 다음 호수집 구조를 살펴보겠습니다.

실시예 8-1. Lakehouse 네임스페이스 패턴 탐색

```

└── s3://com.common_foods.{dev|prod}
    └── common_foods
        ├── 소비자
        │   └── _apps
        │       └── 클릭스트림
        │           └── app.yaml
        │               └── v1.0.0
        │                   └── _체크포인트
        │                       └── 커밋
        │                           └── 메타데이터
        │                               └── 오프셋
        └── 상태

```



Lakehouse 네임스페이스 패턴을 사용하면 데이터 애플리케이션을 같은 위치에 배치할 수 있습니다. 그들이 생산하는 물리적 델타 테이블을 측면에 배치합니다. 이렇게 하면 정책 수가 줄어듭니다. 팀 기반 액세스, LOB(Line of Business) 수준 데이터와 같은 기본 사항을 관리하는 데 필요합니다. 관리 및 액세스를 제공할 환경과 같은 기타 문제가 있습니다. 언제 일이 올바르게 수행되면 개발 환경이 시험장 역할을 할 수 있습니다. 익명화된 생산 데이터를 사용하여 구축된 모의 데이터로 준비된 새로운 아이디어를 위해(거기 여기에서는 위험이 더 높으므로 누가, 무엇을, 어디서, 어떻게, 왜, 규칙을 기억하세요) 두 개의 환경을 물리적 버킷으로 분리하면 따라가기가 더 쉽습니다. 개발 및 스테이징 이후 정지등 패턴은 전통적으로 모든 액세스가 가능하지만 프로덕션 환경은 거의 항상 최소한 노란색 또는 빨간색 수준 액세스가 정당합니다. "개인" 데이터의 경우.

이 장의 앞부분에서 설명한 파일 시스템 소유권 패턴을 다시 생각해 보면, 우리는 지정된 리소스(파일, 애플리케이션, 디렉토리), 승인된 ID에 대한 그룹 수준 액세스 권한은 다음과 같습니다. "읽기 전용"이고 "다른 모든 사람"에 대한 액세스는 단순히 차단됩니다.

데이터 도메인을 담당하는 엔지니어의 "기본" 그룹 멤버십 패턴 특정 도메인의 미션 크리티컬 데이터를 지원하는 데이터 애플리케이션 세트 새로운 팀원이 합류한 후 온보딩 프로세스에 참여해야 합니다. 조직의 업무 방식에 대한 교육을 받고 이를 빠르게 익힐 수 있습니다.

**예제 8-1**에 제공된 레이크하우스 레이아웃에서 데이터 애플리케이션과 소비자 우산 아래에 있는 클릭스트림 데이터 애플리케이션을 위한 테이블 리소스 common\_foods 카탈로그에 있습니다. 디렉토리에는 다음이 포함됩니다.

- 메타데이터: (app.yaml). 리소스 구성 및 기타 중요한 내용을 포함할 수 있습니다. 소유 팀, Pagerduty 또는 Slack 채널을 포함한 애플리케이션 메타데이터 정보. 또한 app.yaml에는 모든 런타임 요구 사항이 포함될 수 있습니다. CPU 코어의 형태, 램, 실행기의 최소 및 최대 수, 액세스 정책, 원하는대로 말만 해.
- 소스 라이브러리: (\*.whl, \*.jar, \*.py 등). 이 라이브러리는 직접 게시될 수 있습니다. s3로 또는 대안으로 컨테이너화된 데이터 애플리케이션으로 작업하는 경우 따라서 애플리케이션에 필요한 모든 것을 컨테이너 파일에 기록할 수 있습니다. 시스템 레이어.
- 구성: (app.properties, Spark.conf). 애플리케이션 구성은 다음과 같습니다. Kubernetes용 ConfigMap을 사용하여 애플리케이션에 Spark.conf 또는

Spark.properties는 기존 Spark 애플리케이션용이거나 데이터 애플리케이션 내에서 지원하는 모든 유형의 구성입니다. 중요한 점은 각 버전(예에서는 v1.0.0)에 대해 모든 리소스가 자체 포함되어 있다는 것입니다. 이 패턴을 사용하면 실수가 발생한 경우(우리 모두는 실수를 합니다) 체크포인트를 손상시키지 않고(작동 중이던) "마지막" 버전으로 쉽게 뒤집을 수 있습니다. • 스트리밍 체크포인트: (\_checkpoints). 이 컬렉션에는 상태 저장 애플리케이션(구조적 스트리밍 또는 기타)에 대한 메타데이터가 포함되어 있습니다. 예를 들어, 업스트림이 또 다른 Delta 테이블인 경우 \_checkpoints에는 처리된 Delta(저장소 버전)의 마지막 읽기 버전과 마지막으로 "관찰된" 커밋 버전을 포함한 싱크 정보가 포함됩니다.

- 테이블 메타데이터 및 실제 파일: Delta 테이블은 팀이 레이크하우스 내에서 운영하는 데 필요한 정책 파일 및 역할의 수를 최소화하기 위해 데이터 제품의 범위 내에 포함됩니다.

모든 애플리케이션 리소스는 s3 접두사({catalog}/{database\_or\_schema}/\_apps/{app\_name}/\*)의 간단한 네임스페이스 패턴을 사용하여 위치합니다. 버전이 지정된 모든 리소스와 자산이 **의미 체계 버전** 내에 포함되어 있습니다. 릴리스(v1.0.0). 지속적인 통합 및 전달(ci/cd)을 데이터 애플리케이션이 포함된 github 저장소와 연결하면 git 릴리스의 git 태그와 함께 애플리케이션 버전을 연결하는 것이 간단해집니다. 이는 또한 현재 릴리스 - 1을 확인하여 오류가 발생한 경우 자동 롤백을 활성화합니다.

이제 실제 델타 테이블을 살펴보겠습니다. 데이터 애플리케이션의 출력 테이블은 데이터 애플리케이션 자체와 동일한 상대 경로 아래에 존재합니다. 데이터 애플리케이션과 테이블 모두의 공통 조상은 특정 카탈로그에 포함된 데이터베이스(또는 스키마)입니다. 이 패턴이 항상 가능한 것은 아닐 수도 있습니다. 특히 데이터 애플리케이션이 여러 브론즈 테이블에서 읽어 실버 기반 출력 테이블을 생성하는 경우에는 더욱 그렇습니다.

애플리케이션 구성의 경우 Spark를 예로 사용하여 다음 구성 속성 Spark.sql.warehouse.dir=s3://com.common\_foods.prod/com\_mon\_foods를 설정할 수 있습니다. 애플리케이션이 common\_foods 카탈로그에 포함된 테이블을 읽거나 쓸 수 있도록 합니다.

## 데이터 자산 및 코드형 정책

일반적인 액세스 패턴을 사용하여 Lakehouse의 보안 및 거버넌스를 단순화할 수 있습니다. 예를 들어 Amazon S3 액세스 권한 부여의 도입을 예로 들 수 있습니다. 스타일은 기존 S3 버킷에 걸쳐 권한을 부여합니다.



다음 섹션에서는 **Amazon S3 액세스 권한 부여** 사용을 살펴봅니다. 높은 수준에서. 또한 이 섹션에서는 Amazon S3에 대한 사전 경험과 정책 관리 작동 방식을 가정합니다.

S3 버킷을 생성합니다. s3 버킷은 프로덕션 레이크하우스를 캡슐화하는 컨테이너 역할을 합니다. Amazon cli를 사용하여 버킷을 설정하고 이를 Production.v1이라고 부릅니다.

### 예제 8-2. Lakehouse를 위한 버킷 설정

```
aws s3api create-bucket \ --bucket
    com.dldgv2.production.v1 \ --region us-west-1 \ --create-
    bucket-configuration
    LocationConstraint=us-west-1
```

버킷 설정이 성공적으로 완료되면 버킷 위치가 반환됩니다. 이는 고유한 arn(Amazon 리소스 이름)이 있음을 의미합니다. 예를 들어 arn:aws:s3:::com.dldgv2.production.v1입니다.

S3 액세스 권한 부여 인스턴스를 생성합니다. s3 액세스 권한 부여 인스턴스는 누가 S3 데이터에 대해 어떤 수준의 액세스 권한을 갖고 있는지 정의하는 논리적으로 그룹화된 개별 권한을 포함합니다. 단일 AWS 계정 내에는 AWS 리전당 하나의 인스턴스가 있습니다. 즉, s3 버킷에 대한 글로벌 액세스가 가능한 경우에도 지역 데이터 액세스 제어가 적용됩니다.

### 예제 8-3. 액세스 권한 부여 인스턴스 생성

```
% 내보내기 ACCOUNT_ID="123456789012"; aws s3control create-access-grants-instance -- 계정 ID $ACCOUNT_ID
```

새 부여 인스턴스를 만든 결과입니다.

```
{
  "CreatedAt": "2024-01-15T22:54:18.587000+00:00", "AccessGrantsInstanceId":
  "default", "AccessGrantsInstanceArn": "arn:aws:s3:us-
  west-1:123456789012:access-grants / 기본"}
```

이제 s3 버킷 설정과 액세스 권한 부여 인스턴스 설정(둘 다 us-west-1에 있음)이 있으므로 s3 액세스 권한 부여에 사용할 IAM 역할과 신뢰 정책을 생성할 수 있습니다.

신뢰 정책을 만들습니다. AWS 서비스(access-grants.s3.amazonaws.com) 권한이 S3 리소스에 대한 GetDataAccess 작업을 사용하여 임시 IAM 자격 증명을 생성할 수 있도록 허용하는 신뢰 정책을 생성해야 합니다.

#### 예제 8-4. trust-policy.json 파일 만들기

```
{
    "버전": "2012-10-17",
    "성명": [{

        "효과": "허용", "주체": { "서
        비스": "access-
        grants.s3.amazonaws.com"
    },
    "작업":
        [ "sts:AssumeRole",
        "sts:SetSourceIdentity", "sts:SetContext"

    ]
}
]
}
```

이제 다음을 실행하십시오.

```
% aws iam create-role --role-name s3ag-location-role \ --assume-role-policy-
document file://trust-policy.json
```

액세스 권한 설정을 완료하는 마지막 단계는 S3 버킷 접두사에 대한 읽기 및 읽기-쓰기 기능을 활성화하는 정책을 생성하는 것입니다.

S3 데이터 액세스 정책을 생성합니다. 마지막 단계는 s3 버킷에 대한 일반 읽기 및 쓰기 권한을 연결하는 것입니다.

```
% aws iam put-role-policy --role-name s3ag-location-role --policy-name s3ag-location-role --policy-
document file://iam-policy.json
```

iam-policy.json 파일은 책의 14장에 대한 github 자료에 포함되어 있습니다.

이제 s3 액세스 권한을 설정했으므로 읽기 및 읽기-쓰기 권한 또는 레이크하우스의 리소스에 대한 관리자 수준 권한을 관리하는 방법을 단순화할 수 있습니다.

### 역할 수준에서 정책 적용

읽다. 리소스에 대한 읽기 전용 기능 또는 테이블 속성, 소유권, 계보 및 기타 관련 데이터를 포함하여 특정 데이터 자산에 대한 메타데이터를 볼 수 있는 기능을 승인합니다. 이 기능은 테이블 내의 행 수준 데이터를 보거나, 버킷 접두사(파일 시스템 경로)에 포함된 리소스를 나열하거나, 테이블 수준 메타데이터를 읽는 데 필요합니다.

### 예제 8-5. Amazon S3 액세스 권한 부여 읽기 정책 적용

```
$ 내보내기 ACCOUNT_ID="123456789012"
```

```
aws s3control create-access-grant \ --account-id
$ACCOUNT_ID \ --access-grants-location-id
default \ --access-grants-location-configuration
S3SubPrefix="warehouse/gold/analytic/**" \ -- 권한 READ \ --grantee GranteeType=IAM,GranteeIdentifier=arn:aws:iam:$ACCOUNT_ID:role/
analy-stRole
```

위의 예에서는 액세스 권한 부여를 사용하여 Amazon S3에 대한 권한을 부여하는 간단한 방법을 보여줍니다.

읽기쓰기. 읽기에서 제공되는 작업 외에도 쓰기 기능에는 행위자가 새 데이터를 삽입(쓰기)하고, 테이블 메타데이터를 업데이트하고, 테이블에서 행을 삭제할 수 있는 수정 기능이 추가됩니다.

<% 코드 />

관리자. 읽기/쓰기로 관리되는 기능 외에도 관리자 역할은 행위자에게 특정 위치에 있는 데이터 자산을 생성하거나 삭제할 수 있는 권한을 부여합니다. 예를 들어 파괴적인 기능을 서비스 주체로만 제한하는 것이 일반적입니다. 마찬가지로 리소스를 생성한다는 것은 리소스를 관리하고 모니터링하기 위한 추가 조정을 의미하는 경우도 많습니다. 헤드레스 사용자는 사용자를 대신해서만 작업할 수 있으므로 워크플로, 명령만 실행하고 이미 존재하는 작업과 작업을 실행할 수 있습니다. 즉, 서비스 주체는 일부 외부 이벤트를 기반으로 특정 작업을 트리거하여 실수로 "죄송한" 표면 영역을 줄일 수 있습니다.

<% code /> 조직의 특정 서비스 주체 및 특정 행위자

### RBAC의 한계

물론 단순히 역할만 사용하여 액세스를 관리하는 경우에는 제한이 있습니다. 주로 역할이 폭발적으로 증가하는 경향이 있습니다. 이것을 "무분별한 확장"으로 생각하세요. 이는 예상치 못한 성공의 부작용입니다. 솔직히 말해서, 비즈니스 라인이 4개만 있고 지원 역할이 4개(개발자, 분석가, 과학자, 비즈니스)인 경우 최대  $4^4 = 256$ (n은 내부 테이블 수)를 보고 있습니다. 회사 전체의 일반 관리 요구 사항을 처리하기 위해 액세스를 관리하는 특별한 규칙이 필요한 사업 부문). 사업 분야가 4개에서 20개로 늘어나면 어떻게 될까요? 50은 어떻습니까? "다음에 무엇을 해야 할지"를 정의하는 것은 가정입니다. 운이 좋아서 회사가 성공하고 좋은 인재를 채용하고 강력한 엔지니어링 규율과 관행을 유지할 수 있었다면 기술적으로 속성 기반 액세스 제어(ABAC)로 전환하기 시작할 수 있습니다. 이는 태그 기반 정책이라고도 하며 세분화된 액세스 제어의 범위 내에서 실행될 수도 있습니다. 사이드바에서 이 내용을 요약하거나 건너뛰겠습니다.

IAM부터 데이터 카탈로그까지, 이는 틀림없이 Lakehouse 내의 두뇌처럼 작동합니다.

## 메타데이터 관리

숲속에서 길을 잃은 적이 있나요? 아니면 GPS나 구식 지도 없이 새로운 장소에서 운전하시나요? 길을 잃는 것은 우리 모두가 공통적으로 느끼는 현상이며, 존재해야 한다고 알고 있는 테이블에 접근하려고 노력하는 데이터 팀에서도 동일한 느낌을 표현할 수 있습니다. 그런데 어디에 있나요? 메타데이터 관리 시스템은 길을 잃는 것과 방향을 갖는 것 사이에서 누락된 구성요소를 제공합니다. 우리의 경우, 우리가 도달하려는 위치는 경유지나 최종 목적지가 아닌 테이블입니다. 그러나 두 가지 문제(올바른 목적지에 도달하는 것)를 해결하는 데 필요한 메타데이터(데이터에 대한 데이터)는 행동을 취할 만큼 유사합니다. 좋은 정신 모델로.

## 메타데이터 관리란 무엇입니까?

데이터 관리와 마찬가지로 메타데이터의 수명 주기는 우리가 가까이에 보유하고 있는 데이터 자산을 추적(및 메모, 설명, 설명 유지)하는 방법을 제공합니다. 일반적으로 데이터 카탈로그의 구성 요소인 중앙 집중식 메타데이터 계층은 조직의 정보 아키텍처를 표현합니다. 여기에는 예 14-1 조직의 성공을 위해 접두사 패턴 사용에서 본 것처럼 카탈로그, 데이터베이스(또는 스키마) 및 데이터베이스(또는 스키마) 아래에 저장된 테이블로 표시되는 계층이 포함됩니다. 메타데이터 계층의 역할은 사용 가능하고 사용 중인 모든 데이터 자산의 현재 상태와 관련하여 기업 전체에 걸쳐 거시적인 보기를 제공하는 것입니다.



운영 메타데이터 계층을 언급할 때 데이터 카탈로그 또는 메타스토어라는 용어를 사용하는 것이 일반적입니다. 메타스토어와 데이터 카탈로그라는 용어는 같은 의미로 사용되지만, 두 용어 모두 API를 통해 액세스할 수 있는 데이터에 대한 데이터를 저장하는 서비스를 설명합니다.

### 데이터 카탈로그 조직 내

위치에 따라 "데이터 카탈로그란 무엇인가"에 대한 다양한 해석이 있을 수 있습니다. 본질적으로 가장 기본적인 형태의 데이터 카탈로그는 "사용자"가 작업을 완료하는 데 필요한 데이터를 "찾을" 수 있게 해주는 도구입니다. "사물 찾기" 문제를 해결하는 방법에는 여러 가지가 있으며, 주어진 문제를 해결하는 방법에는 여러 가지가 있는 것처럼 우리가 해결하려는 문제와 문제의 정의는 실행 가능해야 하며 실제 사례를 기반으로 해야 합니다.

### 고객 사용 사례.

예를 들어 모든 테이블과 해당 소유자에 대한 수동 목록을 만들고 "누군가"가 항상 메타데이터를 최신 상태로 유지하도록 최선을 다할 수 있습니다. 해결책

간단한 공유 스프레드시트일 수 있습니다. 공유 스프레드시트의 알려진 제한 사항은 최신 상태를 유지해야 하는 "누군가"입니다. 이 책은 문제 해결에 관한 것으로 앞의 예는 하지 말아야 할 일에 더 가깝지만 조직의 규모에 따라 가장 간단한 솔루션일 수도 있습니다.

정적 데이터 카탈로그를 유지하기 위해 필요한 수동적 인간 노력을 기반으로 구축된 프로세스의 문제는 귀하 또는 다른 사람이 신뢰할 수 있는 정보가 필요할 때 필연적으로 항상 최신 상태가 아니라는 단순한 사실입니다.

수동 동기화는 확장되지 않기 때문에 업계 추세는 자동 카탈로그 작성 및 동적 데이터 검색 쪽으로 바뀌었습니다. 누가, 무엇을, 어디서, 왜, 언제, 어떻게를 "유지"하는 문제는 모두 데이터 제품 소유자와 Lakehouse 거버넌스 솔루션이 제공하는 기능에 오프로드됩니다(그림 8-1 참조). 여기에는 데이터 생산자, 데이터 제품의 수명 주기, SLA, 테이블 스키마, 델타 테이블이 어떻게 발전할지에 대한 신뢰 측면에서 확립된 약속이 포함됩니다. 메타스토어를 사용하여 달성되는 기능입니다.

## Metastore가 중요한 이유

Lakehouse를 논의할 때 Hive Metastore를 무시하는 것은 거의 불가능합니다. 이는 Hive 메타스토어가 파일 기반 데이터 레이크 테이블을 기준 SQL 테이블처럼 쿼리할 수 있는 구조로 변환하는 기능을 제공하기 때문입니다. Apache Spark SQL 이전에는 Hadoop 클러스터 내에서 MapReduce 작업을 실행하는 Hive SQL을 사용하여 데이터 레이크 내부의 테이블을 쿼리하는 기능을 구현했습니다. Spark SQL이 더욱 널리 보급됨에 따라 Hive 메타스토어는 계속 유지 관리되었지만 시간이 지남에 따라 업계에서는 더 이상 완전한 Hive 배포가 필요하지 않으며 Hive 메타스토어 만으로도 Spark 작업이 Hive 데이터를 Spark 테이블 자체로 변환할 수 있도록 누락된 부분을 제공했습니다..

Hive 메타스토어는 메타스토어 자체가 기존 관계형 데이터베이스(예: Postgres 또는 MySQL)에 상주하는 경우 데이터(데이터베이스 및 테이블) 검색에 활용할 수 있는 일련의 기본 기능을 제공합니다. 즉, Hive 메타스토어에 대한 읽기 액세스 권한이 있는 사용자는 show 명령을 실행하여 데이터베이스와 그 안에 포함된 테이블을 나열하고 어떤 테이블이 존재하는지 검색하거나 tblproperties, dbproperties 또는 기타 검색 테이블을 쿼리할 수 있습니다.

물리적 메타스토어와 물리적 델타 테이블 사이의 관심이 분리되어 있기 때문에 IAM은 파일 시스템 관리를 제공할 수 있으며 SQL 권한은 노출 영역(사용자가 메타스토어 내에서 볼 수 있는 데이터베이스(스키마), 테이블 또는 열)을 제한합니다. 그림 8-6의 다이어그램은 관계형 데이터베이스에 저장된 메타데이터(왼쪽)와 클라우드 개체 저장소 또는 분산 파일 시스템 내에 있는 데이터베이스 및 테이블에 대한 참조(오른쪽)와 관련된 모델을 보여줍니다.

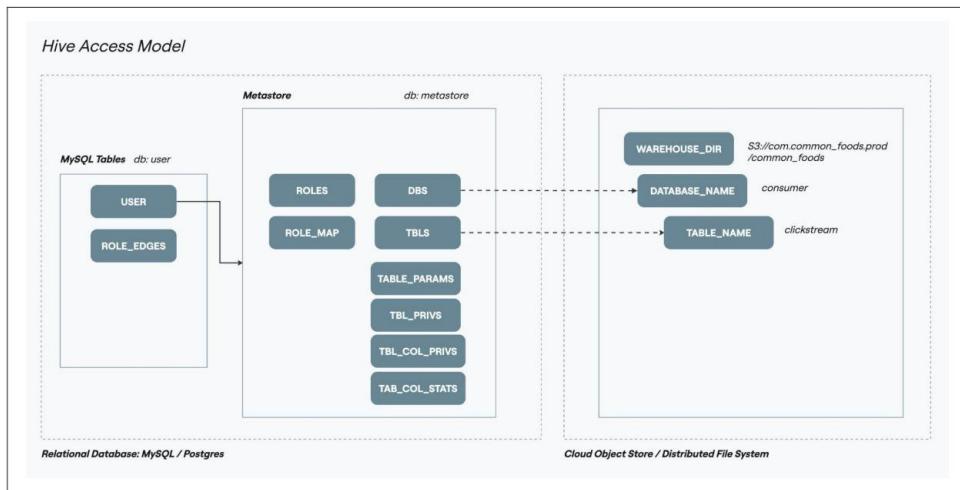


그림 8-6. Hive 액세스 모델은 데이터베이스 및 테이블 메타데이터에 대한 액세스와 스토리지 계층 내의 접두사에 있는 델타 테이블을 나타내는 실제 파일에 대한 액세스 간의 문제를 분리합니다.

그림 8-6 의 다이어그램은 Hive 메타스토어에 대한 높은 수준의 개요를 제공합니다. 메타스토어 자체는 테이블의 위치와 내용에 대한 카탈로그를 제공하는 방법을 가능하게 하는 테이블 세트(>70)입니다. 그러나 클라우드 스토리지의 경로와 관련하여 이러한 데이터 자산의 위치를 포함하여 참조 데이터베이스 및 테이블 데이터를 저장하는 역할만 담당합니다. 메타데이터에는 테이블 유형, 파티션, 열 및 스키마도 포함됩니다.

테이블에 대한 기본 정보는 있으면 좋지만 레이크하우스를 대규모로 운영하는 데 실제로 필요한 상당한 양의 정보가 누락되어 있습니다. 말할 것도 없이 이 책은 모든 테이블 유형이 아닌 Delta에 관한 책입니다. 또는 지원되는 프로토콜이므로 각 델타 테이블에 자체 메타데이터에 대한 참조가 포함되어 있는 경우 하이브 메타스토어가 제공하는 대부분의 내용을 안전하게 무시할 수 있습니다.

Hive 메타스토어가 레이크하우스를 위해 Delta에 제공하는 기능은 객체 트리를 수동으로 나열할 필요 없이(비용이 많이 드는 작업일 수 있음) 지정된 클라우드 스토리지 접두사에 포함된 데이터베이스(스키마) 및 테이블을 식별하는 기능입니다. 델타 로그(테이블 기록 기록)와 테이블의 격리된 스냅샷을 가져오는 기능(시간 이동을 사용하거나 '테이블의 현재 버전'에 대해서만)이 있다는 점을 고려하면 메타스토어의 알려진 인스턴스 내에 있는 카탈로그, 스키마 및 테이블의 일반 "목록" 외부에 있는 Hive 메타스토어입니다.

## 제한사항

Delta-spark 라이브러리를 사용하는 경우 Hive 메타스토어의 변형(또는 대부분의 Hive 메타스토어 API와 호환되는 AWS Glue와 같은 변형)을 사용하고 있을 가능성이 높습니다. Hive 메타스토어를 사용할 때 자주 겪는 문제 중 하나는 다음과 같은 제한 사항입니다. 특정 데이터 애플리케이션의 경우 세션 당 하나의 카탈로그에만 연결할 수 있습니다. 여러 "카탈로그"에 포함된 테이블을 조인해야 하는 요구 사항이 있는 경우 이는 당황스러운 일이 될 수 있습니다. 이 제한은 Spark.sql.warehouse.dir 뿐만 아니라 전역 Spark.sql.catalog.spark\_catalog 설정으로 인해 발생합니다.

이러한 제한 요소는 서로 다른 물리적 버킷 간에 테이블 복사본을 생성하여 해결할 수 있지만(각 카탈로그에 대해 버킷을 사용하는 경우) 이로 인해 단일 소스의 데이터 진실을 달성하는 능력이 저하됩니다.

물리적 버킷 간 및 분산형 데이터 패브릭 전반의 액세스 제어 문제에 대한 솔루션은 Databricks에서 제공됩니다. 유니티 카탈로그라고 합니다. 다음 사이드바는 솔루션에 대한 간략한 개요입니다.

### Unity Catalog

**Databricks Unity Catalog**를 사용하는 경우 이는 레이크하우스, 모든 기업 지역, 심지어 클라우드 전반에 걸쳐 데이터 자산을 분류하고 공유하는 기능을 제공하는 메타스토어라고 하는 중앙 집중식 메타데이터 계층을 제공합니다. Unity 카탈로그 내의 데이터 자산에는 카탈로그, 데이터베이스(스키마), 테이블은 물론 노트북, 워크플로, 쿼리, 대시보드는 물론 파일 시스템 볼륨, ML 모델 등이 포함됩니다.

Unity Catalog는 작업 공간 메타스토어를 물리적으로 분리하고 조직의 모든 데이터 자산에 대한 조감도를 제공하는 Unity Catalog용 계정 수준 API를 제공함으로써 기본적으로는 불가능한 통합 보기 기능을 지원합니다. 현재 오픈 소스 데이터 카탈로그. 단일 메타스토어가 각 작업 영역에 연결되어 있고 Databricks의 각 작업 영역이 us-east 또는 us-west와 같은 물리적 지역에 연결되어 있기 때문에 이는 각 물리적 지역 내에서 생성되고 소비되는 데이터가 안전하게 유지될 수 있음을 의미합니다. 기본적으로 해당 지역입니다. 이러한 우려의 분리는 지역의 데이터 주권 규칙에 따라 특정 관할권에 구속된 현지 규칙 및 규정을 단순화하는데 도움이 됩니다.

플랫폼 측면에서 데이터 거버넌스 엔지니어링은 Databricks 내에서 사용할 수 있는 SQL과 유사한 액세스 제어에 바인딩된 데이터 액세스 규칙을 엔지니어링할 수 있으며, 동일한 플랫폼에서 여러 시스템 간에 이동하지 않고도 감사 정보를 쉽게 검색할 수 있습니다. 이를 통해 조직의 위험을 줄이고 규정 준수 표준을 충족하며 조직 내 데이터 거버넌스의 역할을 단순화합니다.

또한 Unity Catalog는 Databricks 작업 영역 내의 모든 구성 요소에 대한 액세스 권한을 부여하고 취소하는 공통 운영 모델을 제공합니다. 여기에는 그림 8-1에 소개된 데이터 거버넌스 우산의 모든 측면이 포함됩니다.

이 장의 나머지 부분은 오픈 소스에 관한 것입니다. 자체 Lakehouse 거버넌스 플랫폼을 구축하면서 공백을 메우려는 분들에게는 Unity 카탈로그의 한 페이지를 가져가는 것이 매우 도움이 됩니다.

## 데이터 흐름 및 계보

데이터가 Lakehouse로 유입되는 다양한 방식은 가장자리 또는 표면에서 보기를 제공합니다. 표면적으로는 테이블이 어딘가에서 시작되어야 하며 현재 특정 테이블을 구동하는 데이터 소스가 다른 시점에 변경될 가능성이 높다는 점을 이해하고 있습니다. Lakehouse 외부의 데이터 소스는 영구적입니다.

예를 들어, 이메일이나 푸시 알림이 성공적으로 전송될 때마다 공급업체로부터 데이터를 받는다고 가정해 보겠습니다. 우리가 이러한 공급업체로부터 수집하는 데이터는 API 및 내부 데이터 모델에 매우 구체적이며 "그" 시점에 우리가 체결한 모든 계약과도 연결되어 있습니다. 우리가 특정 공급업체를 사용하고 있다는 사실은 소비자 행동에 대한 통찰력, 이메일 공개율 증가 또는 마케팅 성공과 관련된 일부 전환율 지표에 중점을 두는 데이터 소비자와의 관심사가 아닙니다. 운동.

외부 데이터 도메인 내에서 통찰력을 생성하기 위해 결국 다른 팀이 사용할 수 있는 공통 데이터 형식으로 공급업체별 데이터를 변환하는 것은 소비자 데이터 도메인(위의 예)에서 작업하는 데이터 엔지니어링 팀의 임무입니다(이는 메탈리온 아키텍처의 골드 레이어입니다). 공통 형식을 제공함으로써 업무상 중요한 데이터 자산(테이블, 보고서 등)으로의 데이터 흐름을 중단하지 않고 한 공급업체에서 다른 공급업체로 전환할 수 있습니다.

그렇다면 데이터 계보는 이 모델에 어떻게 적합할까요?

### 데이터 계보 데이터

계보의 목적은 레이크하우스 내 초기 수집 지점(데이터 시작)부터 최종 목적지까지(인사이트의 형태를 취할 수 있음) 데이터 여정에 따른 이동, 변환 및 개선을 기록하는 것입니다. 및 기타 BI 기능을 제공하거나 미션 크리티컬 ML 모델을 위한 견고한 기반을 제공합니다. 데이터 계보는 데이터 품질, 일관성 및 전반적인 규정 준수에 대한 척도를 제공하는 데 사용되는 목적으로 중요한 데이터 자산 전체에서 중요한 순간을 포착하는 비행 기록 장치로 간주됩니다.



Andy Petrella는 계보를 "계통 + 연령"3의 교차점으로 설명합니다 . 데이터 소스 간의 직접적인 연결과 연결을 공유한 기간을 나타냅니다.

또한 당사의 데이터 제품 계보는 런타임 파이프라인 작업에 대한 관찰 가능한한 렌즈를 제공하여 데이터 팀이 런타임에 필연적으로 문제가 발생했을 때 언제, 어디서, 왜 문제가 발생했는지 이해할 수 있도록 도와줍니다. 데이터 계보를 활용하여 데이터 흐름을 확인하고 "변경된 사항"을 신속하게 시각화하거나 "더 이상 예상대로 작동하지 않는 사항"을 확인합니다.

### 데이터 계보의 일반적인 용도

- 카탈로그, 데이터베이스, 테이블 및 열 형식 스키마 기반 연결을 제공하여 레이크하우스 전체에서 표 형식 데이터에 액세스하고 사용하는 방법과 공통 접두사를 이해합니다. • Medallion 아키텍처 내에서 중요한 전환 지점을 식별하고 데이터 도메인 내의 어떤 데이터 계층(내부 또는 외부)이 데이터 문제를 해결하는 데 적합한 수준의 개선을 제공하는지 이해합니다. • 감사 인식을 위한 빈도 그래프 또는 기타 테이블을 구축하고 활성 데이터 고객을 이해하기 위해 업스트림 및 다운스트림 종속성을 해결합니다.
- 기계 학습 모델을 교육하는 데 사용되는 테이블이나 대시보드를 구성하는 데 사용되는 특정 테이블 또는 보기 를 포함하기 위해 데이터 자산 유형 전반에 걸쳐 계보를 지정합니다.
- 모니터링을 강화하기 위한 레이크하우스 전체 액세스 및 감사 수준 통찰력에 대한 통찰력을 얻고 감사와 관련하여 중앙 집중식 데이터 거버넌스 팀에 답변을 제공합니다(특정 주체(사용자 또는 그룹)가 작업(읽기, 쓰기)을 실행할 수 있고 "해야" 합니다). 특정 데이터 자산(파일, 테이블, 대시보드 등)

### 기타 사용 사례:

- 규정 준수 및 감사 • 영향 분석  
(문제가 발생한 경우) • 데이터 변경 관리(특정 데이터 자산에 대해 v1에서 v2로 마이그레이션) • 데이터 품질 보증 • 디버깅 및 진단

---

3 데이터 관찰성의 기초, 2장(44페이지).

사용 사례: OpenLineage를 사용하여 데이터 계보 자동화 OpenLineage4

는 데이터 계보 수집 및 분석을 위한 오픈 소스 프레임워크입니다. 확장 가능하며 이를 둘러싼 커뮤니티가 성장하고 있습니다. 프레임워크 설계는 특정 실행 내에서 작업에 대한 메타데이터를 기록하도록 설계된 계보 메타데이터에 대한 개방형 표준을 제공합니다.

**그림 8-7** 의 다이어그램은 데이터 세트, 작업 및 실행 엔터티로 구성된 일반 운영 모델을 보여줍니다. 각 핵심 엔터티(데이터 세트, 작업 및 실행)에는 Facet 키워드로 식별되는 확장 개체가 있습니다. 이는 엔터티 강화를 가능하게 하는 사용자 정의 메타데이터를 캡슐화합니다.

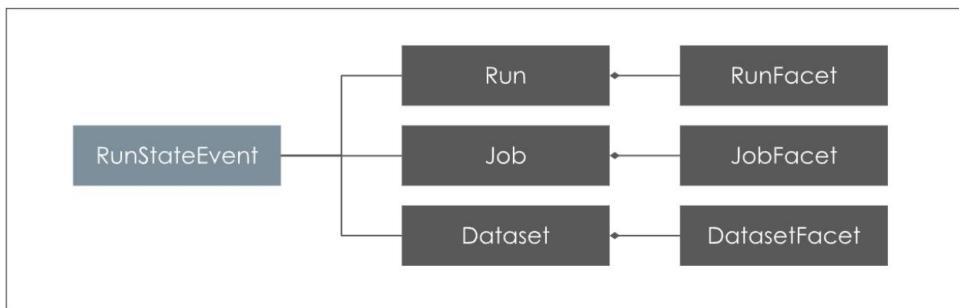


그림 8-7. OpenLineage는 데이터 세트, 작업 및 실행을 캡슐화하는 간단한 엔터티 위에 구축됩니다.

데이터가 단순히 Lakehouse에 존재하는 것이 아니라 초기 테이블(Dataset)을 수집하거나 하나 이상의 업스트림 테이블(Dataset's)을 수정하려면 프로세스(Job)를 실행(Run)해야 한다는 점을 고려하면 새로운 테이블 또는 테이블 세트를 생성하기 위해 이 운영 모델은 기본적으로 모든 데이터 파이프라인 또는 간단한 데이터 흐름의 운영 동작을 추적합니다.

OpenLineage 시작하기 [Java](#) 가 있습니다. 그

리고 [파이썬](#) 클라이언트 API를 사용할 수 있습니다(작성 당시). 다음 예에서는 Python 클라이언트 API를 사용합니다. 전체 예제를 살펴보고 싶다면 책의 github 콘텐츠에서 확인할 수 있습니다.

다음 예에서는 OpenLineageClient 인스턴스를 생성하고 데이터 생산자, 업스트림 데이터 세트, 명명된 작업, 네임스페이스 실행 인스턴스를 할당하기 위한 메타데이터와 시작 및 완료 이벤트를 내보내는 간단한 함수를 설정하는 방법을 보여줍니다.

4 OpenLineage 문서: <https://openlineage.io/docs/>

### 예제 8-6. 시작 및 완료 이벤트를 보내도록 OpenLineage 클라이언트 설정

```

클라이언트 = OpenLineageClient.from_environment() 생산자 =
'common_foods.consumer.clickstream' job_name =
'consumer.clickstream.orders'

데이터 세트 =
{ 'clickstream': Dataset(네임스페이스='consumer', name='consumer.clickstream')
}

cs_job: Job = Job(네임스페이스='소비자', 이름=job_name)

# Run 인스턴스를 생성합니다. run: Run =
Run(f'{job_name}:{str(uuid4())}')

def 방출_시작(클라이언트, 실행, 작업, 생산자): run_event =
    RunEvent(RunState.START, datetime.now().isoformat(), 실행, 작업, 생산자) client.emit(run_event)

def aim_complete(클라이언트, 실행, 작업, 생산자, 입력: 목록[데이터 세트], 출력: 목록[데이터 세트]): run_event = (RunEvent( RunState.COMPLETE,
datetime.now().isoformat(),
실행, 작업, 생산자 , 입력, 출력,
)

클라이언트.방출(run_event)

// 데이터 파이프라인 코드 삽입 app = DataApplication(config) //
애플리케이션을 시작하기 전에 aim_start(client, run,
job, producer)

// 데이터 애플리케이션을 시작합니다. app.run()

// 프로세스를 종료하기 전(app.status() ==
'complete'인 경우:
    Emit_complete(클라이언트, 실행, 작업, 생산자, app.inputs, app.outputs) else: 방출_실패(클라이언트, 실행, 작업, 생산자,
app.Exception())
)

```

**예제 8-6**의 예제 코드에서는 데이터 생산자, 데이터세트를 식별하고 데이터세트, 작업, 실행 및 RunEvent 식별자를 구성하는 규칙을 처리하기 위해 문자열 이름과 명명 규칙을 구성하는 데 약간의 수동 작업이 필요합니다.

엔지니어로서 우리는 무언가가 어떻게 작동하는지 이해한 후에는 항상 단순화를 모색할 것입니다. 데이터 계보의 경우 단순히 단순화하는 것만으로는 충분하지 않고 명명 규칙도 간소화하여 데이터 계보를 제공하려는 새로운 팀이 파이프라인에는 "참"을 제공하기 위해 올바른 이름이 포함됩니다.



Python 데코레이터에 익숙하다면 데이터 애플리케이션의 실행 또는 실행 메서드를 래핑하는 함수를 간단히 제공할 수 있습니다. 데이터 계보를 사용하여 이동 중인 데이터의 현재 상태를 관찰할 수도 있으므로 "실패"를 캡처하는 방법을 제공해야 합니다. Scala 또는 Java 애플리케이션을 작성하는 경우 데이터 계보 아키텍처에 일관된 후크를 "제공"하는 데 사용할 수 있는 간단한 특성 또는 추상 기본 클래스를 제공하십시오.

데이터 애플리케이션과 궁극적으로 데이터 제품을 공급하는 서비스 사이에서 Lakehouse를 통해 데이터가 흐르는 방식은 동적입니다. 물처럼. 썰물과 흐름이 있고 항상 피드가 고갈되는 영역이 있습니다. 그러나 데이터 제품이 종료되거나 이전 데이터 진실 소스의 버전이 관리되면 항상 새로운 소스와 연결하는 새로운 방법이 있을 것입니다. 데이터 도트. 이것이 데이터 계보 캡처의 장점입니다. 올바르게 완료되면 결과 정보는 무엇을, 언제, 어떻게에 대한 실시간 또는 마지막 "활성" 상태를 제공합니다. 그런 다음 이 추가 메타데이터를 델타 테이블 메타데이터와 결합하여 연결 그래프에 관한 귀중한 정보를 제공하고 모니터링, 경고를 위한 정보를 제공하거나 데이터 검색 서비스와 결합할 때 알리는 데 사용할 수 있습니다. 이 정보는 "테이블에 대한 마지막 업데이트가 언제였습니까?", "이전 소스가 더 이상 사용되지 않으므로 이제 어떤 데이터 소스를 사용할 수 있습니까?" 등과 같은 질문에 답할 수 있습니다.

## 데이터 공유 데이

터를 공유한다는 것은 무엇을 의미합니까? 아니면 데이터 자산인가요? 가장 간단한 방법으로, 알려진 신원(이해관계자, 고객, 시스템 또는 서비스)이 데이터를 읽고 사용할 수 있는 기능을 제공합니다. 델타 테이블의 경우 이는 알려진 ID에 델타 트랜잭션 로그를 읽고 테이블의 스냅샷을 생성하여 테이블 읽기를 실행할 수 있는 기능을 제공하는 것을 의미합니다.



4장에서는 Delta Sharing 프로토콜을 사용한 공유를 다루며, 이는 Lakehouse 내에서 공유를 활성화하는 가장 간단한 방법입니다.

우리가 우리의 데이터를 다른 사람들에게 제공하려는 데에는 여러 가지 이유가 있습니다. 우리는 우리의 데이터로 수익을 창출하여 다른 회사에서는 얻을 수 없는 통찰력을 제공할 수 있습니다(데이터 사용 법률을 준수하고 소름끼치지 않는 한). 소매업에서 자주 볼 수 있는 경우로서 파트너나 공급업체에 데이터를 제공해야 할 수도 있습니다.

회사 외부로 유출되지 않는 데이터의 경우, 내부 비즈니스 라인 간에 데이터를 공유하는 것은 모든 사람이 동일한 데이터 진실 소스를 참조하도록 하는 데 중요합니다.

### 데이터 수명주기 자동화 이 장의 앞부분에서

데이터와 데이터 자산은 필요한 동안만 생존할 것으로 예상된다는 개념을 소개했습니다. 데이터의 자연스러운 수명 주기에 있어서 선택의 여지가 있는 경우도 있고 법적 및 지역적 요구 사항을 준수해야 하는 경우도 있습니다. 어느 쪽이든 데이터에는 만료일이 있습니다. 일부 데이터는 우유와 비슷합니다. 사용해야 합니다. 그렇지 않으면 빨리 상할 것입니다. 다른 곳에서는 데이터가 꿀처럼 작동하여 초과 근무를 통해 결정화되지만 약간의 노력으로 쉽게 완벽하게 건강한 상태로 돌아갑니다. 그렇다면 이러한 데이터 수명주기를 어떻게 자동화할 수 있을까요?

### 테이블 속성을 사용하여 데이터 수명주기 관리 우리는 6장

에서 Delta 테이블에 속성을 적용하는 방법을 배웠습니다. Delta 프로토콜이 속성을 사용하여 테이블의 반복적인 유지 관리를 쉽게 하기 위해 유ти리티 기반 기능을 제어하는 것과 같은 방식으로 우리도 다음과 같은 방법을 활용할 수 있습니다. 반복적인 작업을 처리하는 방식을 통합하는 테이블입니다.

**예제 8-7** 에 표시된 다음 예제에서는 델타 테이블에서 데이터를 삭제하는 간단한 방법을 만들기 위해 INTERVAL 유형을 사용하는 방법을 소개합니다. 세 가지 새로운 테이블 속성이 소개되며 책에 사용된 명명 규칙은 모든 Lakehouse에 설정된 접두사 패턴에 맞게 조정될 수 있습니다.

델타 테이블에 보존 정책을 추가합니다. 속성 접두사 `cata log.table.gov.retention.*`을 사용하면 보존 관련 용도에 맞는 네임스페이스가 제공됩니다.

사례.

#### 예제 8-7. 테이블 속성 추가

```
% 스파크.sql(f"""
ALTER TABLE delta.`{table_path}` SET
TBLPROPERTIES
('catalog.table.gov.retention.enabled='true',
'catalog.table.gov.retention.date_col='event_date',
'catalog.table.gov.retention.policy='간격 28일'
)""")
```

Lakehouse에 새로운 거버넌스 동작을 추가할 때마다 특정 기능을 선택하거나 선택 해제할 수 있는 방법을 제공하는 것이 좋습니다. 이 경우, 카탈로그 `log.table.gov.retention.enabled` 부울로 기능을 커울 수 있습니다. 아디 -

속성이 테이블에 존재하지 않는 한 기본 상태가 false인 경우 선택하고 다른 항목을 무시하는 것이 훨씬 쉽습니다.

다음으로, 예제 8-8 에 표시된 코드는 Interval 값(28일)을 IntervalType 이 포함된 Column 객체로 변환하는 함수를 도입합니다 .

#### 예제 8-8. 문자열을 간격으로 변환

```
% python
def Convert_to_interval(간격: str): target = str.lower(interval).lstrip()
    target = target.replace("interval", "").lstrip() if
    target.startswith("inter-val") else 대상 번호, 간격_유형 = re.split("\s+", 대상) 금액 = int(번호)

dt_interval = [없음, 없음, 없음, 없음] 간격_유형 == "일"인 경우:

    dt_interval[0] = lit(364 if amount > 365 else amount) elif Interval_type == "시간":
        dt_interval[1] = lit(23 if amount > 24 else amount)
        elif Interval_type == "mins": dt_interval[2] = lit(59 if amount > 60 else amount) elif
        Interval_type == "초": dt_interval[3] = lit(59 if
            amount > 60 else amount) else: raise RuntimeException(f"알 수 없는 간격_유형
            {interval_type}")

make_dt_interval(일=dt_interval[0], 시
    긴=dt_interval[1],
    mins=dt_interval[2],
    secs=dt_interval[3]을 반환합니다.

)
```

이제 예제 8-8 의 python 함수를 사용하여 델타 테이블에서 간격 형식으로 catalog.table.gov.retention.policy 규칙을 추출할 수 있습니다 .

다음으로, 새로운 Convert\_to\_interval 함수를 사용하여 델타 테이블을 가져와서 유지할 수 있는 가장 빠른 날짜를 반환합니다. 이는 테이블에서 오래된 데이터를 자동으로 "삭제"하거나 심지어 "테이블"을 규정을 준수하지 않는 것으로 표시하는 데 사용할 수 있습니다. 최종 흐름은 예제 8-9에 나와 있습니다.

#### 예제 8-9. 표준을 통한 규정 준수 보장

```
% python
table_path = "..." dt =
DeltaTable.forPath(spark, table_path) props = dt.detail().first()
['properties'] table_retention_enabled =
bool(props.get('catalog.table.gov.보존.활성화'),
```

```
'false'))
table_retention_policy = props.get('catalog.table.gov.retention.policy', '간격 90일')

간격 = Convert_to_interval(table_retention_policy)

규칙 =
( Spark.sql("지금으로 current_timestamp() 선
택") .withColumn("retention_interval", 간
격) .withColumn("retain_after", to_date((col("now")-col("retention_interval")))) )
)

rule.show(truncate=False)
```

우리는 DeltaTable 유ти리티 함수를 사용하여 테이블 속성에 접근하는 간단한 방법을 제공합니다. 테이블 속성에서 보존 관련 구성은 추출합니다. 여기에는 기본값이 false인 부울(기능 플래그)과 기본값이 90일인 보존 정책이 포함됩니다. IntervalType 열인 간격 변수를 사용하면 현재 시간(이 표현식을 실행할 때)을 Convert\_to\_interval의 결과와 함께 가져 와서 간격을 빼고 이를 keep\_after 열의 DateType으로 캐스팅할 수 있습니다. DataFrame 규칙을 살펴보면 다음을 볼 수 있습니다.

지금	보유_간격	retain_after
2024-03-24 20:11:27.759222	간격 '28 00:00:00' 일부터 초까지 2024-02-25	

그래서 3월 24일부터 28일을 되돌아보면 2월 25일이 됩니다. 윤년으로 인해.

14-5에서 시작하여 14-7로 끝나는 예는 델타 테이블에 자동 수명 주기 정책 제어를 제공하는 한 가지 방법을 보여줍니다. 이 패턴을 사용할 수 있는 곳은 많습니다. 데이터 삭제 이외의 영역으로 확장하기로 결정하거나 이 예를 사용하여 오래된 데이터 삭제 방법을 제어할 수 있습니다. 6장에 제시된 삭제 조건을 기억하시나요?

Catalog.table.gov.retention.date\_col에 제공된 열 ID를 사용하여 keep\_after Date 보다 오래된 데이터를 삭제할 수 있습니다.

### 감사 로깅 감사는

Lakehouse 내의 또 다른 렌즈로 볼 수 있습니다. 데이터 계보를 소개한 것처럼 각 데이터 자산에는 특정 규칙(정책) 및 자격 세트가 있습니다. 어떤 작업을 기록해야 하는지 생각하면서 Lakehouse 내에서 비행 기록 장치와 같은 특정 작업을 사용할 수 있습니다. 데이터 수명주기 및 데이터가 데이터 네트워크를 통해 어떻게 흐르는지 측면에서 과정을 추적하는 대신 액세스가 알려지고 관리 가능하도록 보장합니다. 추가적으로,

이동 중인 데이터에 대한 추적 작업은 위험을 완화하고 위험을 식별하는 데 도움이 될 수 있는 변칙을 식별하는 데 도움이 되는 데이터 소스(메트릭)를 제공할 수 있으며, 악의적인 행위자가 하점 보안을 활용할 가능성 을 식별합니다. 사용자 또는 그룹은 (언제든지) 모든 데이터 자산(리소스)

에 액세스할 수 있습니다. 또한 우리는 리소스, 소유자 및 수행할 수 있었던 사람을 이해하기 위해 수행된 작업(작업)에 대해 어떤 ID(사용자 또는 그룹)가 주어진 작업(작업) 또는 그 반대를 수행했는지 알고 싶습니다. 주어진 행동.

보안 및 거버넌스 페르소나에게 시기적절한 정보를 제공하고 시스템 전반에 걸쳐 안심할 수 있도록 하려면 데이터를 수집하고 Lakehouse 내에서 사용할 수 있도록 하여 감사 이벤트 수집을 간소화해야 합니다. 감사 추적을 간소화하는 것은 이 책의 범위를 벗어납니다. 그러나 모든 데이터 자산에는 "책임 있는 소유자가 있어야 하며" 각 작업에는 IAM 권한 및 역할 기반 정책을 통해 처리되는 "액세스 제어"가 필요하다는 점을 고려하면 다음과 같이 할 수 있습니다. 특정 미션 크리티컬 데이터 제품이 소유한 리소스에 대한 IAM의 변경 사항을 간단히 캡처하여 소규모로 시작하세요.

이는 간단하고 소박한 시작을 제공하고 테이블 및 해당 계보에 대한 집합적 메타데이터를 사용하여 효율적인 감사 기능을 제공하고 이를 기반으로 테이블에 액세스하고, 새로 고치고, 삭제하거나 심지어는 테이블이 삭제되는 빈도에 대한 추가 데이터를 구축할 수 있습니다. 자동 데이터 수명주기 관리를 위해 [예제 8-8](#)에 소개된 기술을 사용하여 어떤 테이블이 규정을 준수하지 않는지 추적합니다.

## 모니터링 및 경고 Lakehouse의 성

공을 위해서는 모니터링 및 경고 기능을 제공하는 것이 필수적입니다. 이는 데이터 거버넌스 및 보안 기능의 목적으로만 사용할 수도 있고 각 데이터 제품이 적절한 운영 관찰, 모니터링 및 경고 기능을 갖도록 확장할 수도 있습니다.

### 일반 규정 준수 모니터링 보존 자동화 사용 사

예 ([예 8-7](#))로 돌아가서 테이블이 규정을 준수하지 않는지 확인하는 데 보존 기간을 사용할 수 있다는 사실을 논의했습니다. 예를 들어 거버넌스 조직에서 Catalog.table.gov.retention.\* 속성을 활성화하기 위해 모든 테이블 기반 데이터 자산이 필요하다고 가정해 보겠습니다.

거버넌스 엔지니어는 "데이터 카탈로그"의 도움을 받아 메타데이터 읽기 전용 통합을 쉽게 설정하여 "테이블" 소유자가 규칙을 따르고 테이블에 보존 정책 구성을 "활성화"했는지 확인할 수 있었습니다. 스캔은 매 일 수행되어 어떤 "테이블"이 일반 규정을 벗어나는지 기록하고 자동으로 Catalog.engineering.comms. [email|slack] 속성(6장에서 소개됨)을 사용할 수 있습니다.

팀에 자동화된 커뮤니케이션을 보내거나 엔지니어링 조직의 책임자에게 에스컬레이션합니다. 이 경우 경고는 PagerDuty 경보가 아니지만 팀이 규정을 준수하도록 호출하기 위해 통합될 수 있습니다.

데이터 품질 및 파이프라인 저하 메달리온 아키텍처를

논의할 때 데이터 품질에 대해 다루었습니다. 각 테이블 기반 데이터 자산(고객 포함)에 대해 파이프라인이 실패하거나 한때 중요한 데 이터가 포함된 열이 비어 있는 경우 다운스트림 소비자(데이터 고객)가 다운될 수 있습니다. 데이터가 얼마나 자주 도착할 것으로 예상되는지(테이블 새로 고침 주기 또는 최신 상태)를 전달하기 위해 각 델타 테이블에 테이블 속성이 도입된 경우, 이러한 속성을 사용하여 문제가 발생했음을 데이터 생성 팀에 자동으로 경고할 수 있습니다.

실제 시나리오의 경우 예제 8-10에 소개된 테이블 속성은 많은 강력한 정보를 제공하는 네 가지 간단한 속성을 보여줍니다.

#### 실시예 8-10. 각 델타 테이블의 의도 선언

```
% 스파크.sql(f"""
ALTER TABLE delta.`{table_path}` SET
TBLPROPERTIES
('catalog.table.deprecated='false',
'catalog.table.expectations.sla.refresh.주파수='간격 1시간', 'catalog.table.expectations .checks.주파
수='간격 15분', 'catalog.table.expectations.checks.alert_after_num_failed='3'
)
""")
```

예제 8-7 부터 예제 8-9 까지 소개된 것과 동일한 테이블 스캐닝 프로세스와 기술을 사용하여 간단한 패턴을 활용하여 주어진 테이블에 대한 검사를 자동으로 실행할 수 있습니다. 여기서 이론은 테이블이 더 이상 사용되지 않는 한 알려진 서비스 수준 계약(SLA)이 있어야 한다는 것입니다. 데이터 자산(특히 테이블)과 관련하여 다운스트림 소비자는 모두 데이터가 사용 가능하게 되는 빈도 또는 새로 고쳐지는 빈도를 알고 싶어하는 경향이 있습니다.

배치 처리 또는 마이크로 배치 처리를 사용할 시기를 기준으로 결정을 내릴 때 이는 일반적으로 하나 이상의 업스트림 데이터 소스의 기대치에 따라 결정됩니다. 모든 소스가 일반적으로 15분 이내에 새로 고쳐지지만 하나만 업데이트되고 특정 데이터 답변을 제공하기 위해 모든 데이터가 필요한 경우 항상 일괄 처리 모드에 머물게 됩니다. 회의 없이 테이블에서 업스트림 SLA 정보를 이해하는 것이 더 쉬울수록 의사 결정이 훨씬 쉬워집니다. 그런 다음 문제가 발생하거나 업스트림의 "새 데이터 없음"으로 인해 파이프라인이 정지되는 경우 테이블에 대해 선언된 SLA를 확인하여 무슨 일이 일어났는지 이해할 수 있습니다. 데이터 계보 테이블과 약간의 창의적인 에너지를 활용하여 간단한 UI를 구축하여 다음에 대한 "최신" 정보를 제공할 수도 있습니다.

Lakehouse 내의 데이터 흐름 및 경로의 어떤 테이블이 규정을 준수하는지, 예상보다 느리게 실행되는지 또는 회의를 줄이기 위해 자동화할 수 있는 모든 사용 사례.

## 데이터 검색이란 무엇입니까?

데이터 검색을 통해 사용자는 간단한 인터페이스를 통해 레이크하우스 전체의 데이터 자산(자원)을 검색할 수 있습니다. 이는 모니터링 및 경고에 대해 설명한 것과 동일한 기술을 사용하여 메타데이터 관리에 필요한 메타데이터에 의존함으로써 달성을 할 수 있지만 복잡한 검색 기능을 제공할 수 있는 방식으로 수행됩니다. ElasticSearch에 익숙하시거나, Google, ChatGPT를 사용해보신 분. 귀하의 질문에 대한 답변은 거의 즉시 제공됩니다. 이는 인덱스를 사용하기 때문입니다.

데이터 검색의 경우 문제에 대한 해결책은 테이블 메타데이터(소유권, 규칙, 즉각적인 업스트림 및 다운스트림)를 ElasticSearch 인덱스에 "추가"하는 것만큼 간단할 수 있습니다. 그런 다음 "카탈로그", "데이터베이스/스키마" 또는 기타 데이터 자산 유형 등 검색 엔진에 추가 유형을 추가하려면 인덱싱된 메타데이터만 수정하면 됩니다. 유지 관리되는 자산의 크기와 수에 따라 솔루션을 적절하게 확장할 수 있지만 데이터 자산이 100만 개 미만인 경우 간단한 ElasticSearch 인덱스를 사용하면 매우 먼 길을 갈 수 있습니다.

Lakehouse 고객이 성공하려면 어떤 종류의 것들이 필요한지 생각해 보십시오. 어떤 경우에는 검증된 "고품질" 테이블이나 "검증된" 소유자가 유용할 수 있습니다. 특정 "태그 또는 배지"를 얻기 위한 프로세스가 통제된 프로세스인 한(누구도 '자신의 태그를 추가'할 수 없음을 의미) 고객은 프로세스를 조작할 수 없다고 신뢰할 것입니다. 다른 것이 없다면 데이터 검색 솔루션의 이동 부분 측면에서 복잡성의 균형을 맞추는 방법에 대해 생각해 보십시오. 색인을 생성해야 하는 메타데이터 소스는 몇 개입니까? 얼마나 자주? 상황이 바뀔 때 알림을 받을 수 있는 간단한 방법이 있나요? 프로세스를 자동화할 수 있나요?

데이터 검색을 위한 좋은 솔루션을 보유하면 셀 수 없이 많은 시간을 절약할 수 있으며 데이터 조직의 기준을 실제로 높일 수 있습니다. 속도와 정확성의 균형을 맞추는 것을 잊지 마세요. 잘못된 데이터에 대한 빠른 검색 결과는 회사 리소스를 낭비하고 Lakehouse에 대한 신뢰도를 낮출 수 있습니다.

## 요약

Lakehouse 내부의 귀중한 자산을 관리, 보호 및 저장하는 방법은 복잡하거나 복잡하거나 단순할 수 있습니다. 이는 모두 크기와 규모(또는 테이블 및 기타 데이터 자산의 수)와 우리가 실현하는 시점에 따라 다릅니다. 보다 완벽한 거버넌스 솔루션이 필요합니다. 여정의 어느 시점에 있는 작게 시작하십시오. 버킷 수준에서 데이터 카탈로그를 분리하여 "모든 액세스" 데이터를 "매우 민감한" 데이터에서 분리하는 것부터 시작하십시오. 데이터에서 "사람"이 필요로 하는 것과 "시스템 및 서비스"가 무엇인지 동기화하는 솔루션 방식을 계층화하세요.

동일한 데이터가 필요하며 이를 누가, 무엇을, 언제 전략에 적용할지 결정합니다. 상황이 더욱 복잡해짐에 따라 데이터 계보는 "데이터 애플리케이션"이 데이터에 연결하고 데이터를 사용하여 새로운 데이터 자산을 생성하는 방법과 위치를 "확인"하는 데 도움이 된다는 점을 기억하십시오. 그런 다음 미래에 필요한 것이 무엇인지 염두에 두고 계속해서 거버넌스 기반을 구축하세요.