요즘 우아한 AI 개발 #10

Polars로 데이터 처리를 더 빠르고 가볍게

출처: 우아한형제들 기술블로그 (https://techblog.woowahan.com/18632/)

2025년8월2일
나영재

배달시간예측서비스팀: Polars 라이브러리로 데이터 처리 성능을 혁신하다

배달의민족 앱 내의 배달 예상 시간 및 고객 전달 시간을 예측하는 시스템을 개발하는 저희 팀은, 데이터 처리 효율과 속도를 극대화하기 위해 Polars 라이브러리를 도입하여 성능을 획기적으로 개선했습니다. 이 경험을 통해 얻은 인사이트를 공유합니다.

누구를 위한 이야기인가요?

빠른 데이터 처리의 필요성

Pandas보다 메모리 효율적이고 빠른 DataFrame 라이브러리를 찾는 모든 개발자 및 데이터 분석가에게 유용합니다.

대규모 데이터 처리의 도전

GB 단위 데이터 처리에서 Spark와 같은 대안을 고민하는 데이터 과학자 및 엔지니어에게 실질적인 해결책을 제시합니다.

코드 가독성 및 효율성 개선

기존 Pandas의 가독성과 표현식에 아쉬움을 느끼며 더 나은 코드 경험을 원하는 분들에게 새로운 시야를 제공합니다.

1.도입 배경

데이터 규모의 증가

AI 및 머신러닝 프로젝트에서 **점점 더 방대한 양의 데이터**를 다루게 되었습니다.

기존 라이브러리의 한계

기존 데이터 처리 라이브러리들은 **대규모 데이터 처리**에서 성능 및 메모리 효율성 문제에 직면하고 있습니다.

1.Pandas의 문제점

- 느리고 리소스(CPU, Memory)가 많이 소모된다.
- 멀티 코어를 제대로 지원하지 않고 병렬처리가 미흡하다.
- 가독성이 떨어진다.

2.Spark의 문제점:

- 모든 데이터 과학자가 Spark로 데이터 처리 로직을 구현하고 튜닝하는 것에 익숙지 않다. (러닝 커브 존재)
- 비용 효율성이 낮다. (비싼 리소스, driver & executor 리소스 고려)
- 대용량 데이터가 아닐 때는 오히려 오버헤드 발생, slow start 문제

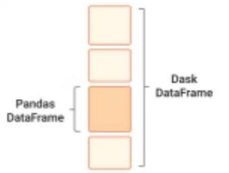
효율적인 대안의 필요성

이는 개발 워크플로우의 **병목 현상**으로 이어져, 더 빠르고 효율적인 데이터 처리를 위한 새로운 솔루션이 필요합니다.

- 로컬 환경에서도 편하게 개발 및 테스트가 가능해야 하고,
- 별다른 인프라가 필요 없고,
- 성능도 좋고,
- 러닝 커브도 적고,
- Airflow 환경 또는 컨테이너 기반으로 잘 패키징해서 운영 데이터 파이프라인에서 문제없이 돌릴 수 있고,(너무 큰 욕심일 수도...)

STUDY 진행

- Ray
  - Ray 자체는 데이터 처리보단 분산처리 프레임워크에 가깝지만, Ray Dataset API를 사용하면 Spark나 Dask처럼 병렬/분산 처리를 할 수 있지 않을까? 라는 생각에 살펴보았습니다.
  - 분산 처리에 초점이 맞춰져 있고 데이터 처리 기능이 부족했고, Spark와 비슷하게 작은 데이터엔 더 비효율적이었습니다.
  - 운영 배포 시 어느 정도 인프라 구축이 필요했습니다. 분산 학습과 튜닝 등 분산처리를 쉽게 적용할 수 있어서 좋은 발견이었으나 데이터 처리 목적에 맞지 않아서 탈락했습니다.
- Dask
  - Pandas와 문법이 비슷하면서 상대적으로 성능이 좋았습니다. 하지만 Dask의 DataFrame은 병렬화를 위해 Pandas DataFrame을 잘게 나눈 여러 파티션으로 구성되어 있어서 성능상의 한계가 존재했습니다.
  - 병렬처리뿐만 아니라 여러 노드에 분산처리까지 가능하다는 장점이 있었지만, 성능상 아쉬운 부분, 특히나 pandas DataFrame 사용 때문인지 메모리 소모 면에서 약점을 보였습니다.



- Modin
  - Pandas와의 호환성을 완벽하게 지원하여 러닝 커브나 코드 수정 없이 기존 데이터 처리 성능을 개선할 수 있다는 큰 장점이 있었습니다.
  - 하지만 성능 향상에 한계가 있었습니다.
- vaex
  - 문법 구조가 다르고 기능이 부족했으며, 사용 사례가 매우 적어서 탈락했습니다.
- Numba
  - 일부 연산을 병렬화 및 고속화할 수 있지만 범용성은 떨어졌습니다.

C vs Rust

⚙️ 성능 및 제어력

항목	C	Rust
퍼포먼스	매우 빠름	C와 동급 성능 가능
저수준 접근	직접적인 포인터 연산 가능	`unsafe` 블록에서만 허용
인라인 어셈블리	자유롭게 가능	제한적 지원 (`asm!`)
크로스 컴파일	좋음	매우 활발 (embedded, WebAssembly 등 포함)

🧰 생태계와 도구

항목	C	Rust
빌드 시스템	`Makefile`, `cmake` 등 복잡	`cargo` 단일 툴로 관리
패키지 관리	외부 도구 필요 (`vcpkg`, `conan`)	`crates.io`와 통합
테스트 지원	없음 (외부 프레임워크 필요)	내장 테스트 프레임워크 제공 (`cargo test`)
Fuzzing, Linting	수동	내장 (`clippy`, `miri`, `fuzzing`) 도구 풍부

C는 속도와 제어력의 왕이지만, Rust는 "속도 + 안전 + 병렬성 + 개발 생산성"을 함께 갖춘 차세대 시스템 언어입니다.

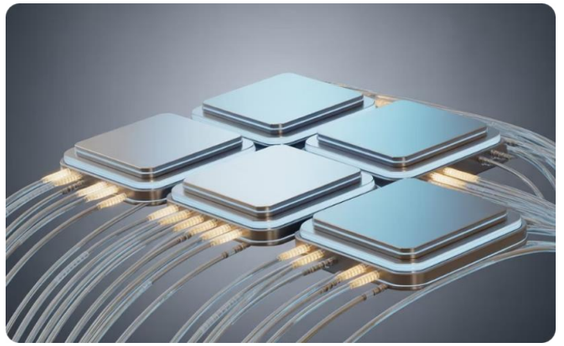
2. Polars란?

Rust로 개발되어 탁월한 성능을 자랑하는 데이터프레임 라이브러리입니다.

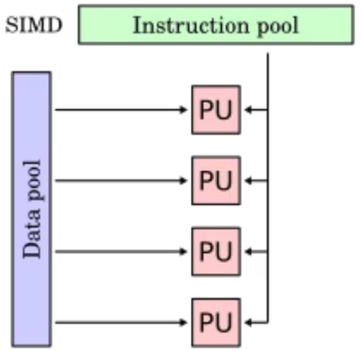
- 러스트의 소유권 모델 덕분에 메모리 관리에 대한 오버헤드가 없으며, 안전한 동시성과 병렬 처리가 가능합니다.
- 메모리 캐싱과 재사용성 또한 높습니다.
- 이러한 특징이 데이터 처리 성능을 극대화하는 데에 크게 기여합니다

Apache Arrow

- 열(columnar) 기반 인메모리 포맷
- 다양한 언어 간의 **제로 복사(zero-copy) 데이터 교환**을 목표로 함
- 고속 분석을 위한 벡터화 처리 및 SIMD 최적화 가능
- Pandas, Spark, Polars, DuckDB, Snowflake 등 여러 시스템에서 사용

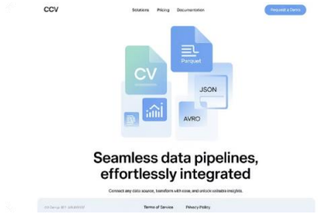


멀티스레딩과 SIMD(Single Instruction, Multiple Data) 최적화를 통해 병렬 처리 능력을 극대화합니다.



다양한 IO 지원

로컬 파일, 클라우드 스토리지, 데이터베이스 등 다양한 데이터 스토리지 계층을 지원하고 성능 또한 매우 우수합니다. 기본적으로 CSV, JSON, Parquet, Avro 등 다양한 포맷에 대한 읽기/쓰기를 지원하고 파일을 읽을 때도 별표(asterisk, *)와 같은 Globs Pattern을 활용해서 여러 파일을 읽어올 수 있어서 매우 편리합니다. 실무에서는 데이터베이스나 Trino와 같은 쿼리 엔진에 쿼리를 제출하고 그 결과로 polars.DataFrame을 반환하는 read_database() 기능도 자주 사용하고 있습니다.




```
import polars as pl

# data-1.parquet, data-2.parquet 등과 같은 파일 한 번에 읽기
df = pl.read_parquet("docs/data/data-*.parquet")
df.write_parquet("docs/data/total_data.parquet")

# Cloud Storage에서 데이터 읽어오기
df = pl.read_parquet("s3://bucket/*.parquet")

# read from DB
df = pl.read_database_uri(
      query="SELECT * FROM foo",
      uri="postgresql://username:password@server:port/database“
)

df = pl.read_database(
    query="SELECT * FROM test_data",
    connection=user_conn,
    schema_overrides={"normalised_score": pl.UInt8},
)
```

read_* 와 같은 함수 대신 scan_* 이라는 함수를 사용하게 되면 Lazy API를 위한 LazyFrame으로 반환되어 이를 활용하여 바로 Lazy 연산을 수행할 수 있습니다. 이렇게 되면 즉각적으로 모든 데이터를 메모리에 올리는 것이 아닌 최적화를 한 후 실제 연산을 수행하여 더 효율적으로 처리할 수 있습니다.

테스트해 본 IO 성능 실험을 간단하게 공유해드리면 아래와. 같습니다

710MB parquet file (7,373,092 rows * 64 columns) 읽기 실험
Polars의 read_parquet: 4s 554ms
Pandas의 read_parquet: 33s 916ms

Lazy API와 쿼리 최적화

Polars의 Lazy API는 즉시 연산을 수행하지 않고, Query Plan이라고 하는 연산 계획을 수립한 후 최적의 시점에 연산을 실행하는 지연 평가(Lazy Evaluation) 방식입니다. 이는 불필요한 중간 연산을 줄여주고 필터링과 pushdown 등의 최적화 기술을 사용하여 필요한 데이터만 읽어와서 처리하기 때문에 메모리 소모와 연산 복잡도를 줄여줍니다. Polars에서는 polars.DataFrame 말고 polars.LazyFrame이 있는데 이는 즉각적으로 연산을 하는 게 아니라 쿼리 플랜만 담아두고 있다가 값이 필요할 때, 즉 구체화(materialize)할 때 collect() 함수를 호출하여 연산하는 방식입니다.

아래 코드와 실험 결과를 통해 즉시 연산과 지연 연산의 차이를 보여드리겠습니다. 아래 코드는 점심/저녁 주문 피크 시간대(11시 ~ 14시, 18시 ~ 21시)에 배달 거리가 2KM 이상이고, 지역별로 배달예상시간을 평균 내는 코드입니다.

```
# Eager API
df = pl.read_parquet("predicted_data-1.parquet")
result = (df
     .filter(pl.col("distance") >= 2 & pl.col("created_hour")
     .is_in([11, 12, 13, 18, 19, 20]))
     .group_by("pickup_zone_id")
     .agg(    pl.mean("delivery_time").alias("avg_delivery_time")))

# Lazy API (scan_parquet 함수 사용과 마지막 collect 함수 호출이 유일한 차이)
df = pl.scan_parquet("predicted_data-1.parquet")
result = (df
     .filter(pl.col("distance") >= 2 & pl.col("created_hour")
     .is_in([11, 12, 13, 18, 19, 20]))
     .group_by("pickup_zone_id").agg(
         pl.mean("delivery_time").alias("avg_delivery_time")
)).collect()
```

	메모리 소모(peak)	실행 시간
Eager API	17.84 GB	9.07 초
Lazy API	5.9 GB	1.009 초

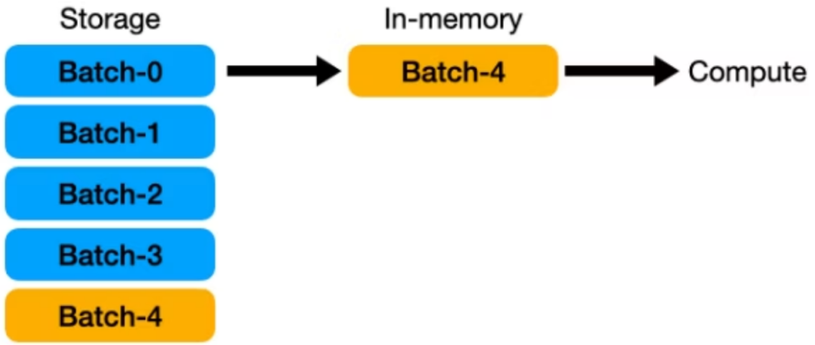
푸시다운(pushdown)이란? 쿼리 연산을 스토리지 계층으로 한 단계 내려서 데이터 로드 비용을 최소화하는 기술입니다. 푸시다운에 대한 정책과 구현은 솔루션이나 오픈소스마다 다르지만, 해당 개념은 최적화를 위해 대부분 채택하고 있습니다.

- Predicate pushdown(조건자 푸시다운): 필터를 적용해 요청한 데이터만 읽는 방식 (filter pushdown이라고도 함)
- Project pushdown: 필요한 열만 읽는 방식
- Slice pushdown: 필요한 슬라이스(일부 행)만 읽는 방식

Lazy API: 필요한 연산만 수행하여 성능을 최적화합니다.

Out of core 방식(Streaming API)

Polars에서는 streaming이라는 기능을 활용해서 out of core 방식으로 연산을 수행할 수 있습니다. 여기서 out of core 방식이란 external memory 알고리즘이라고도 하는데, 메모리에 담기 너무 큰 데이터를 처리할 때 디스크나 네트워크 등을 통해 일정 단위로 데이터를 가져와서 처리하는 방식을 말합니다. 즉, 한 번에 모든 데이터를 메모리에 올리는 것이 아니라 일정 단위로 데이터를 자르고 그 조각을 가져와서 처리하고 이를 반복하는 것이죠.



하나의 데이터셋을 배치 단위로 나누어서 작업

```
# scan_parquet에서 *를 사용하여 같은 패턴의 파일을 모두 읽어오도록 변경
df = pl.scan_parquet("predicted_data-*.parquet")
result = (df
    .filter(pl.col("distance") >= 2 & pl.col("created_hour")
    .is_in([11, 12, 13, 18, 19, 20]))
    .group_by("pickup_zone_id").agg(
        pl.mean("delivery_time").alias("avg_delivery_time")
)).collect(streaming=True)
```

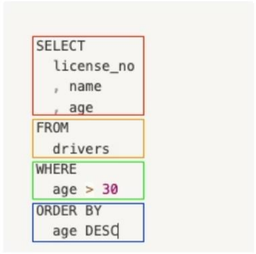
	메모리 소모(peak)	실행 시간
Lazy 연산	8.8 GB	아래와 비슷
Streaming API	3.71 GB	6.835 초

2.2. 사용성 측면에서의 장점: 왜 편리한가?

2.2.1. SQL과 비슷한 구조

Polars는 SQL과 비슷한 표현식과 직관적인 문법을 가지고 있기 때문에 배우기 쉽고 가독성도 좋은 편입니다.

```
result_df =(
    drivers
    .select(pl.col("license_no", "name", "age"))
    .filter(pl.col("age") > 30)
    .sort(by=pl.col("age"), descending=True)
)
```



```
SELECT
  license_no
  , name
  , age
FROM
  drivers
WHERE
  age > 30
ORDER BY
  age DESC
```

2.2.2. 칼럼 선택 시 편리함

데이터를 분석하거나 피처 엔지니어링을 할 때 원하는 칼럼을 선택해야 하는 경우가 많습니다. 그럴 때 칼럼 이름을 리스트로 받아오거나 타입을 받아와서 필터링하기도 하는데 Polars에서는 타입에 따라 선택할 수도 있고 정규식을 활용할 수도 있습니다. 학습이나 추론을 할 때 인코딩된 칼럼을 굉장히 자주 다루는데 아래 코드에서처럼 정규식으로 처리할 때 무척 편리했었습니다. 테이블 생성을 위해 데이터 타입을 일괄 변경할 때 정수형이나 숫자형 열을 일괄 선택하는 기능도 자주 사용했었습니다.

```
import polars as pl

# 모든 칼럼 선택
selected_df = pl_df.select(pl.col("*"))
selected_df = pl_df.select(pl.all())
pl_df.select(pl.col(pl.Int64)) # 데이터 타입과 타입의 크기에 따라 선택 가능

import polars.selectors as cs

# int64, int32인지 구분하지 않고 정수형 칼럼을 선택하고 싶을 때
pl_out_with_int = pl_df.select(cs.integer())

# int64, int16, float32인지 구분하지 않고 실수형 칼럼과 문자열 칼럼을 선택하고 싶을 때
pl_out_with_num = pl_df.select(cs.numeric(), cs.string())

# 칼럼 이름에 정규식을 적용해서 선택(_encoded로 끝나는 칼럼)
pl_df.select(pl.col("^.*_encoded$"))

# 일부 열만 제외
pl_df.select(pl.exclude("name", "private_number"))
```

2.2.3. 시계열 연산

Polars는 시계열 지원이 좋은 편인데 시간과 관련된 여러 타입을 지원하며, 리샘플링이나 시간 윈도우 기반 그룹화 등의 기능도 제공합니다. 그리고 일치하는 키가 존재하지 않을 때 가장 가까운 값을 기준으로 join하는 기능인 asof join도 지원해서 실무에서 자주 사용하고 있습니다.

30분 단위로 데이터를 생성하고, 이를 15분 간격으로 업샘플링하기 위해 중간 보간법을 사용하는 코드입니다. 업샘플링을 했을 때 빈 값을 fill_null로 채우게 됩

```
import polars as pl from datetime
import datetime df = pl.DataFrame(
    { "time": pl.datetime_range(
          start=datetime(2024, 7, 23),
          end=datetime(2024, 7, 23, 2),
          interval="30m",
          eager=True,
        ),
      "name": ["a", "a", "a", "b", "b"],
      "value": [1.0, 2.0, 3.0, 4.0, 5.0],
    })
print(df)
result = (
    df.upsample(time_column="time", every="15m")
    .interpolate()
    .fill_null(strategy="forward")
)
print(result)
```

```
shape: (5, 3)
```

time	name	value
datetime[μs]	str	f64
2024-07-23 00:00:00	a	1.0
2024-07-23 00:30:00	a	2.0
2024-07-23 01:00:00	a	3.0
2024-07-23 01:30:00	b	4.0
2024-07-23 02:00:00	b	5.0

```
shape: (9, 3)
```

time	name	value
datetime[μs]	str	f64
2024-07-23 00:00:00	a	1.0
2024-07-23 00:15:00	a	1.5
2024-07-23 00:30:00	a	2.0
2024-07-23 00:45:00	a	2.5
2024-07-23 01:00:00	a	3.0
2024-07-23 01:15:00	a	3.5
2024-07-23 01:30:00	b	4.0
2024-07-23 01:45:00	b	4.5
2024-07-23 02:00:00	b	5.0

업샘플링 결과: 15분 단위로 값이 채워진 결과

group_by_dynamic 예시

group_by_dynamic을 사용하게 되면 고정 윈도우 연산이나 롤링 윈도우 연산도 가능합니다.
아래 코드는 불규칙한 시간 단위로 생성된 데이터를 10분 윈도우로 평균과 합을 구하는 코드입니다.

```
import polars as pl

# 데이터프레임 생성
df = pl.DataFrame(
    { "timestamp": [
          "2024-07-23 00:00:00",
          "2024-07-23 00:01:00",
          "2024-07-23 00:02:00",
          "2024-07-23 00:10:00",
          "2024-07-23 00:12:00",
          "2024-07-23 00:20:00" ],
     "value": [10, 20, 30, 40, 50, 60]
} )

# timestamp 열을 날짜시간 형식으로 변환
df = df.with_columns(pl.col("timestamp").str
     .strptime(pl.Datetime, "%Y-%m-%d %H:%M:%S"))
print(df)

# 데이터프레임을 timestamp 열 기준으로 정렬
df = df.sort("timestamp")

# 10분 윈도우로 데이터프레임을 그룹화하고 각 그룹별로 평균과 합을 계산
result = (df
       .group_by_dynamic(index_column="timestamp", every="10m")
       .agg([
            pl.col("value").mean().alias("mean_values"),
            pl.col("value").sum().alias("sum_values"),
        ]
))
print(result)
```

shape: (6, 2)

timestamp	value
datetime[μs]	i64
2024-07-23 00:00:00	10
2024-07-23 00:01:00	20
2024-07-23 00:02:00	30
2024-07-23 00:10:00	40
2024-07-23 00:12:00	50
2024-07-23 00:20:00	60

shape: (3, 3)

timestamp	mean_values	sum_values
datetime[μs]	f64	i64
2024-07-23 00:00:00	20.0	60
2024-07-23 00:10:00	45.0	90
2024-07-23 00:20:00	60.0	60

10분 윈도우로 그룹화하여 연산

2.2.4. SQL 직접 사용

최근 DuckDB와 같은 도구에서 로컬 및 클라우드 스토리지 내 파일이나 DataFrame 객체에 직접 SQL 쿼리를 할 수 있도록 지원하는데, Polars 역시 비슷한 기능을 제공합니다. Python 기반으로 연산하는게 익숙치 않다면 SQL을 사용할 수도 있고, 이 기능을 사용하면 쿼리 최적화가 수행되기 때문에 성능을 높일 수도 있습니다.

```
pl_df = pl.DataFrame({
       "VendorID": ["A", "A", "B", "B", "B"],
      "passenger_count": [1, 1, 2, 2, 1],
      "trip_distance": [0.95, 1.2, 2.51, 2.9, 1.53],
      "payment_type": ["card", "card", "cash", "cash", "card"],
      "total_amount": [14.3, 16.9, 34.6, 27.8, 15.2]
})

# 위에서 만든 DataFrame에 my_table이라는 이름을 붙여준다.
ctx = pl.SQLContext(my_table=pl_df, eager_execution=True)

# eager_execution은 lazy evaluation을 끄고 즉시 결과를 보겠다는 것
result = ctx.execute("SELECT * FROM my_table WHERE trip_distance > 2")
print(result)
```

3. 실무 적용 사례

3.1. 배달예상시간 학습 파이프라인 개선

수천만 행의 학습 데이터(parquet 파일)를 가져와서 읽고 이를 전처리하는 부분이 학습 파이프라인에 포함되어 있었습니다. 기존에도 성능을 고려하여 Dask를 적용해서 처리했었는데 이 부분을 Polars로 변경했고, Dask 대비 실행시간은 80% 수준으로 단축하고 메모리는 40% 수준으로 감소시켰습니다.

3.2. 사용자 정의 함수 적용시 개선

Pandas에서는 apply를 활용해 사용자 정의 함수(User Defined Function, UDF)를 행 단위로 수행합니다. Polars도 같은 기능을 지원합니다. 사용자 정의 함수를 적용하여 변환하는 부분을 Polars로 변경하고 나서 성능 개선을 체감할 수 있었습니다. 물론 Polars도 Python UDF를 적용할 때는 자체 지원 함수를 사용할 때보다 성능이 현저하게 떨어지게 되고 이에 대한 경고 또한 공식 문서에 포함되어 있습니다. 왜냐하면 UDF 실행 때문에 러스트 코어의 장점이나 벡터화 연산의 장점을 많이 잃어버리기 때문이죠. 하지만 Pandas 대비 훌륭한 실행시간 개선을 보여줬습니다.

팀에서 데이터 전처리를 할 때 위/경도 데이터를 가져와서 H3 index로 변환하는 부분이 있습니다. 이 부분을 아래와 같이 Polars 코드로 변경했더니 기존 Pandas로 24.3초 정도 걸리던 연산이 5.87초로 단축되었습니다.

중요하고 많이 사용하는 UDF의 경우 함수 자체를 러스트로 개발하고 이를 파이썬 코드에서 호출해서 사용하면 성능을 높일 수 있으니 참고해 주세요. 거의 5배 가량 빨라진 것이죠.

3.3. 준실시간 추론 파이프라인

저희는 고객에게 더 정확한 시간을 안내하기 위해 준실시간 추론을 하게 됩니다. 배달예상시간의 경우, 사용자가 지면을 조회하면서 여러 가게에 대한 시간을 확인하기 때문에 모든 케이스를 고려하여 미리 피처를 생성해 놓고 준실시간 추론 시점에 피처를 가져와 메모리에 올리고 간단한 처리를 하여 모델에 넣게 됩니다. 그리고 이 모델의 추론 결과를 비즈니스 로직과 전시를 담당하는 쪽에 이벤트로 만들어 전달해야 하죠. 이 과정에서도 Polars를 적용하여 눈에 띄는 성능 효율화를 가져올 수 있었습니다. 이때 수행한 작업은 파일 읽기, group_by 및 aggregation 작업, 이벤트 형태로 변환하기 위한 UDF 적용 작업이었습니다. 매번 실행할 때마다 24GB 정도의 메모리를 소모하던 것을 대략 40% 이하로 줄여서 10GB 정도로 감소한 것을 보실 수 있습니다. 이를 통해 k8s pod 사이즈를 줄여서 비용효율을 높일 수 있었습니다.

1. 추천 시스템 데이터 처리

배달의민족 리뷰 추천 시스템에서 방대한 데이터를 효율적으로 처리하는 데 Polars가 활용되었습니다.

2. 핵심 데이터 전처리

리뷰 필터링, 데이터 클렌징, 그리고 텍스트 벡터화와 같은 핵심 전처리 과정에서 Polars의 성능을 활용했습니다.

3. 성능 및 효율성 향상

기존 Pandas 대비 8배에서 최대 20배 빠른 데이터 처리 성능을 확인하여, 실제 업무 효율성을 크게 향상시켰습니다.

4. 결론

Pandas 한계 극복:

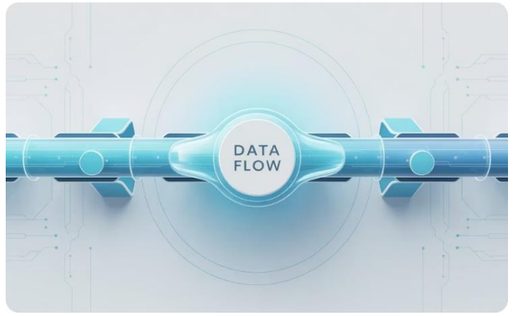
Polars는 Pandas의 한계를 뛰어넘는 고성능 데이터프레임 프레임워크입니다.

러스트 기반의 안정성:

Rust로 개발되어 탁월한 속도와 안정성을 제공합니다.

실무에 최적화된 기능:

Lazy API, Apache Arrow, 멀티스레딩 기능으로 실무 데이터 처리에 매우 적합합니다.



파이프라인 최적화:

데이터 파이프라인을 경량화하고 전반적인 성능 향상에 크게 기여합니다.