Proposal of LSID

Lexicographically Sortable Unique Identifier

Ryan Donghan Kwon 2022-11-30

분산컴퓨팅 환경에서 범용 고유 식별자를 대체하기 위한 사전순 정렬 가능한 고유식별자의 제안

권동한
하나고등학교
kznm.develop@gmail.com

Proposal of Lexicographically Sortable Unique Identifier to replace Universal Unique Identifier (UUID) in Distributed Computing System

Ryan Donghan Kwon
Hana Academy Seoul

요 약

분산 컴퓨팅 환경에 현재 사용되고 있는 고유 식별자에는 UUID, ULID, Snowflake ID 등이 있으며 이들은 1) PK를 기준으로 인덱싱하는 데이터베이스 엔진에 삽입할 경우 지속적으로 Latency가 증가하며 2) 1ms 기록 단위로 정밀한 인덱싱이 불가능하거나 기록 가능한 최대 일자에 제한이 있으며 3) Node ID를 관리하는 Zookeeper 등의 중앙 관리 시스템이 필수적으로 필요한 등의 단점이 있다. 본 논문에서는 이를 개선하여 사전(시간)순 정렬 가능한, epoch+91,929년까지 사용 가능한, 기존 고유 식별자 대비 Insert Latency 증가률이 가장 낮은 22자 96-bit 고유 식별자 LSID를 제안한다.

1. 서 론

분산 컴퓨팅 환경에서 고유성이 보장되는 식별자는 빈번하게 사용된다. 온라인 사진 공유 커뮤니티 사이트 'Flickr' 등의 사용사례[1]와 같이 Sequential한 고유 식별자를 발행하는 티켓 서버를 구축 및 운용할 경우, 전체 시스템의 규모가 증가할수록 티켓 서버의 오버헤드로 인한 전체 시스템의 성능 저하가 일어날 수 있으며 단일 고장점 (Single Point of Failure, SPOF)으로서 시스템 장애를 일으킬 가능성이 높다. 이에 중앙 집중화되지 않으며 고유성을 보장할 수 있는 범용 고유 식별자 (Universally Unique Identifier, UUID), ULID, Twitter Snowflake (이하 Snowflake ID) 등의 식별자가 현재 사용되고 있는데, 분산 컴퓨팅이 대중화됨에 따라 범용 고유 식별자 사용으로 인한 성능 저하 이슈의 해결과 워크로드 최적화의 필요성 등이 제시되고 있고 이에 본 연구는 기존 시스템에서 널리 사용중인 고유 식별자들의 장단점을 분석하여 개선한 사전순 정렬 가능한 고유 식별자 (Lexicographically Sortable Unique Identifier, 이하 LSID)를 제안하고자 한다.

2. 기존 고유 식별자의 분석

2.1. Universally Unique Identifier (UUID)

UUID[2]는 시간과 공간에 걸쳐 고유성이 보장될 수 있는 16진수 36문자 (32자리 문자와 4자리 하이픈)으로 표현되는 128-bit 고유 식별자이다. 총 5가지 타입의 UUID가 국제 표준으로서 정의되어 있으며 그 중 시간 기반으로 식별자를 생성하는 UUID1과, 모든 자리를 랜덤 또는 유사-랜덤 (psuedo-random) 하게 식별자를 생성하는 UUID4가 가장 빈번하게 사용된다. 일반적으로

```
2acf084b-07a3-4568-a097-4114d00a98fd
```

와 같은 형태를 가지며, UUID1의 경우 100ns 단위 UNIX-time 60-bit Timestamp, 4-bit Version, 16-bit Clock Sequence와 48-bit Node ID (Mac Address)로 구성된다.

UUID는 MySQL 등의 기본키 (Primary Key, PK)를 기준으로 데이터를 정렬하여 저장하는 InnoDB 또는 이와 유사한 메커니즘을 사용하는 데이터베이스 엔진을 사용하는 DBMS에서 PK로 사용할 시 성능 저하 이슈가 발생한다고 알려져 있다. 또한 시간 등의 정보가 사전순으로 정렬되지 않을 수 있고 48-bit Mac Address를 모두 Node ID로 사용하기 때문에 Kubernetes 등의 컨테이너 환경에 배포되는, 물리 Mac Address가 할당되지 않는 현대 분산 컴퓨팅 시스템에서 효율성이 떨어진다고 판단된다.

2.2. Universally Unique Lexicographically Sortable Identifier (ULID)

ULID[3]는 사전순 정렬 가능한 UUID의 대체제로, 1ms 단위 UNIX-time 48-bit Timestamp와 80-bit Randomness를 통하여 구성된다. UUID와는 달리 Crockford's Base32[3]를 사용하여 하이픈 없는 26자 문자열로 인코딩된다는 특징이 있다. 일반적으로

```
01ARZ3NDEKTSV4RRFFQ69G5FAV
```

와 같은 형태를 띤다. ULID는 UUID 대비 10자리의 문자열을 절약하여 데이터베이스상 정렬 및 처리 속도의 개선을 이끌었지만, UUID 대비 정밀한 시간의 표현이 불가능하며 이에 동일 밀리초 단위 Timestamp에선 Randomness로 인해 사전순 정렬이 불가능하다는 단점이 있다. ULID Python 커뮤니티 구현체의 프로덕션 사용 환경 (Python 3.9) 기준 약 200,000개의 동일한 Timestamp를 공유하는 사전순 정렬 불가능한 식별자가 생성됨을 확인하였다.

2.3. Snowflake ID

Snowflake ID는 Twitter사에서 자사의 서비스 게시글 생성을 위해 개발한 식별자 시스템으로, ms단위 Epoch 41-bit Timestamp와 10-bit Machine ID, 12-bit Machine Sequence ID로 구성된 19자리 정수 64-bit (63+1)로, signed int 범위 내에 저장이 가능하여 저장 및 정렬상에서 다른 고유 식별자 대비 강점을 가지고 있다. 일반적으로

```
1584097399939268608
```

와 같은 형태를 가지고 있고 국제표준으로 규격되어 있지 않아 Discord, Instagram 등[5]의 Well-known 글로벌 서비스에서 Snowflake ID를 변형하여 사용하고 있으며[5], 시간순으로 정렬 가능하다. 하지만 LSID와 같이 ms 단위 Timestamp를 사용하여 mission-critical service를 로깅하는 등 정밀한 이벤트 발생 시간의 관리가 필요한 경우 이러한 특징과 Zookeeper등의 Worker ID 관리 서버가 필수적으로 필요하다는 단점 등이 발생한다.

3. LSID의 제안

이 논문에서는 2.1.~2.3.에서 분석한 기존 고유 식별자들의 단점을 보완하고 장점을 최대화하고자 두가지 규격의 LSID를 제안한다. 일반적으로

```
00jtx-04fecrkm-0cgm-3n
```

와 같은 형태로 쓰이며, case-insensitive하게 설계되었으므로 정수가 아닌 문자를 대문자로 치환하여

```
00JTX-05RA0XA6-0CGM-6H
```

와도 같이 사용될 수 있다. LSID는 Crockford's Base32로 인코딩된 22문자로 구성된 96-bit 고유 식별자 규격이며 구분자 하이픈을 기준으로 Date 25-bit, 100ns 단위 Timestamp 40-bit, Worker ID 20-bit, Sequence ID 10-bit로 구성되었으며 널리 알려진 100ns 단위 Timestamp를 표기하는 고유 식별자 중 96-bit 22자로 가장 짧다. Date와, 해당 일자 0시 부터 계산되어지는 100ns 단위 timestamp를 분리하여 1 ms 단위 타임스탬프를 AD 10,889년까지 표기 가능한 ULID 등과 달리 100ns 단위 timestamp를 epoch부터 33,554,432일 (약 91,929년) 까지 별도의 수정 없이 표기 가능하다는 특징이 있다.

대부분의 경우에서 사용이 될 표준 LSID의 경우 Python 3.9로 작성된 구현체에는 인자를 전달하지 않을 경우 UTC UNIX-epoch 시간 기준으로 타임스탬프를 생성하며 Mac Address를 BLAKE2s로 해시하여 Worker ID로 사용, Python의 표준 `random.getrandombits` 구현체를 기반으로 Sequence ID를 생성한다. Worker ID와 Sequence ID를 인자로 지정하여 사용할 수 있으며, Worker ID의 경우 추가적인 인자 없이도 사용 가능하나 공간상의 이슈로 인해 해시 충돌 및 보안 이슈 발생 가능성이 있어 컨테이너 오케스트레이션 시스템에서 제공하는 노드의 고유 ID 또는 Zookeeper 등의 시스템을 사용하는것을 권장한다.

만일 해당 식별자가 외부에 노출되어 보안상 문제가 발생할 가능성이 있는 경우, Worker ID와 Sequence ID를 모두 `random.getrandombits` 표준 구현체를 사용하여 인자로 전달, 식별자를 생성하면 된다.

4. LSID의 장점 및 단점에 대한 고찰

타 고유 식별자 대비 더욱 정밀한 Timestamp의 표기가 가능하며, 커스터마이즈 가능한 Worker ID, Sequence ID 등의 부가적인 정보를 담을 수 있다는 장점이 있다. 32진 인코딩을 사용하여 96bit 22문자로 최적화하였으며, 기존 UUID 대비 지속적인 데이터베이스 Transaction 속도에서 이점을 가질 수 있다.

UUID와 ULID, LSID를 데이터베이스의 인덱싱이 공유되지 않는 서로 다른 테이블에 100,000회 삽입했을 때 그림 1~3와 같은 레이턴시를 보인다. 해당 데이터를 선형 회귀 분석한 결과는 표 1과 같다.

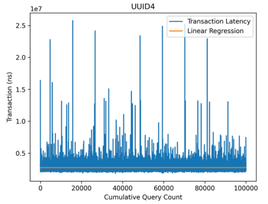


그림 1. UUID의 Transaction Latency

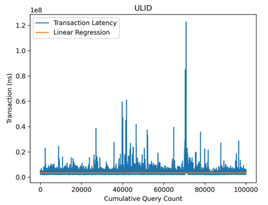


그림 2. ULID의 Transaction Latency

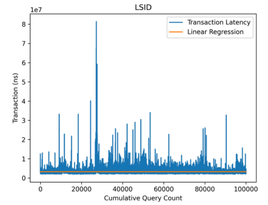


그림 3. 제안된 LSID의 Transacion Latency

	기울기	절편
UUID	.13634740839764645	3339880.9344822564
ULID	.185529453792534	3348060.9900751
LSID	.09454195695944488	3273360.7194230063

표 1. UUID, ULID, LSID의 100,000회 Insert Transaction Latency에 대한 선형 회귀 분석 결과

지속적으로 Insert Transaction을 발생시킨 결과, 초기 Latency는 3000000ns 전후로 세 고유 식별자가 거의 동일하지만, 지속될수록 ULID가 가장 높은 Latency 증가율을 보였으며 본 논문에서 제안된 LSID가 가장 낮은 증가율을 보였다.

LSID의 실 사용시 가장 우려되는 단점으로는 48-bit Mac Address를 20-bit로 hash할 시 충돌 발생 가능성이 높으며 기본적으로 Random하게 생성되는 Sequence ID의 폭이 2^{10}로 크기 않다는 부분이 있다. 분산 컴퓨팅 환경에서 각 노드에 제공되는 Node ID를 LSID의 Node ID로 사용하고, 예커니즘 상 연속해서 식별자를 생성하지 않는 방식 등으로 해당 단점을 극복할 수 있는 부분이 있다.

참 고 문 헌

[1] Kay L., "Ticket Servers: Distributed Unique Primary Keys on the Cheap," Available at https://code.flickr.net, Retrived 2022. 10. 28., 2010.
[2] P. Leach et al., "A Universally Unique IDentifier (UUID) URN Namespace," RFC 4122, July 2005.
[3] Alizain Ferrasta, "Universally Unique Lexicographically Sortable Identifier," Available at https://github.com/ulid/javascript, Retrived 2022.10.28.
[4] Ryan King, "Announcing Snowflake," Available at https://blog.twitter.com/engineering, Retrived 2022. 10. 28., 2010.
[5] Unknown, "Sharding & IDs at Instagram," Instagram Engineering, December 2012.

분산컴퓨팅 환경에서 범용 고유 식별자를 대체하기 위한 사전순 정렬 가능한 고유식별자의 제안

권동한
하나고등학교
kznm.develop@gmail.com

Proposal of Lexicographically Sortable Unique Identifier to replace Universal Unique Identifier (UUID) in Distributed Computing System

Ryan Donghan Kwon
Hana Academy Seoul

요 약

분산 컴퓨팅 환경에 현재 사용되고 있는 고유 식별자에는 UUID, ULID, Snowflake ID 등이 있으며 이들은 1) PK를 기준으로 인덱싱하는 데이터베이스 엔진에 삽입할 경우 지속적으로 Latency가 증가하며 2) 1ms 기록 단위로 정밀한 인덱싱이 불가능하거나 기록 가능한 최대 일자에 제한이 있으며 3) Node ID를 관리하는 Zookeeper 등의 중앙 관리 시스템이 필수적으로 필요한 등의 단점이 있다. 본 논문에서는 이를 개선하여 사전(시간)순 정렬 가능한, epoch+91,929년까지 사용 가능한, 기존 고유 식별자 대비 Insert Latency 증가률이 가장 낮은 22자 96-bit 고유 식별자 LSID를 제안한다.

1. 서 론

분산 컴퓨팅 환경에서 고유성이 보장되는 식별자는 빈번하게 사용된다. 온라인 사진 공유 커뮤니티 사이트 'Flickr' 등의 사용사례[1]와 같이 Sequential한 고유

타입의 UUID가 국제 표준으로서 정의되어 있으며 그 중 시간 기반으로 식별자를 생성하는 UUID1과, 모든 자리를 랜덤 또는 유사-랜덤 (psuedo-random) 하게 식별자를 생성하는 UUID4가 가장 빈번하게 사용된다. 일반적으로

고유 식별자

Unique Identifier

- 고유 식별자(unique identifier, UID)는 객체 집합 참조에서 특정 목적을 위해 객체에 대해 모든 식별자 가운데 고유성을 보장하는 식별자이다.
- UUID, ULID, Snowflake ID 등이 대표적이다.

범용 고유 식별자

Universally Unique Identifier, UUID

- UUID[2]는 시간과 공간에 걸쳐 고유성이 보장될 수 있는 16진수 36문자 (32자리 문자와 4자리 하이픈) 으로 표현되는 128-bit 고유 식별자이다.

- UUID1의 경우
  - 100ns 단위 UNIX-time 60-bit Timestamp
  - 4-bit Version
  - 16-bit Clock Sequence
  - 48-bit Node ID (Mac Address)로 구성된다.

```
2acf084b-07a3-4568-a097-4114d00a98fd
```

Universally Unique Lexicographically Sortable Identifier

ULID

- 사전순 정렬 가능한 UUID의 대체제
- 1ms 단위 UNIX-time 48-bit Timestamp와 80-bit Randomness를 통하여 구성

```
01ARZ3NDEKTSV4RRFFQ69G5FAV
```

Snowflake

Twitter Snowflake ID

- Twitter사에서 자사의 서비스 게시글 생성을 위해 개발한 식별자 시스템

- ms단위 Epoch
  41-bit Timestamp와 10-bit Machine ID, 12-bit
  Machine Sequence ID로 구성된 19자리 정수 64-bit

```
1584097399939268608
```

사전순 정렬 가능한 고유 식별자의 제안
Proposal of LSID

- UUID - **너무 길다!** 데이터베이스 삽입 시 성능 저하 이슈 발생
- ULID - **너무 부실하다!** 시간을 제외하고 어느 의미있는 데이터도 담고 있지 않음
- Snowflake - **너무 복잡하다!** Ticket Server의 필요로 인해 SPoF 발생
- 짧고 알차며 복잡하지 않은 고유식별자의 필요성

```python
from time import time_ns
from datetime import datetime, timezone
from hashlib import blake2s
from random import randrange
from uuid import getnode as get_mac

base32 = "0123456789ABCDEFGHJKMNPQRSTVWXYZ"


def toBase32(n: int) -> str:
    if n < 32:
        return base32[n]
    return toBase32(n // 32) + base32[n % 32]


def LSID1(epoch: datetime = datetime(year=1970, month=1, day=1, hour=0, minute=0, second=0, microsecond=0, tzinfo=timezone.utc),
          worker_id: int = int(blake2s(key=get_mac().to_bytes(length=6, byteorder='big'), digest_size=2).hexdigest(), 16),
          sequence_id: int = randrange(0, 32 ** 2)) -> str:
    LSID: str = ""
    now: datetime = datetime.now(timezone.utc)
    now_ns = time_ns()

    # Date(5) # 5 * 5 = 25bit
    diff = now - epoch
    b32_date: str = toBase32(diff.days)
    LSID += ('0' * (5 - len(b32_date)) + b32_date + '-')

    # NS(8) # 8 * 5 = 40bit
    ns = int((now_ns % (24 * 60 * 60 * 1000 * 1000 * 1000)) / 100)
    b32_ns: str = toBase32(ns)
    LSID += ('0' * (8 - len(b32_ns)) + b32_ns + '-')

    # Worker ID(4) # 4 * 5 = 20bit
    b32_wid: str = toBase32(worker_id)
    if len(b32_wid) > 4:
        raise ValueError("Worker ID is too large (0<=int<=2^20-1)")
    LSID += ('0' * (4 - len(b32_wid)) + b32_wid + "-")

    # Random ID(2) # 2 * 5 = 10bit
    b32_sid: str = toBase32(sequence_id)
    if len(b32_sid) > 2:
        raise ValueError("Sequence ID is too large (0<=int<=2^10-1)")
    LSID += ('0' * (2 - len(b32_sid)) + b32_sid)

    return LSID
```

```python
from time import time_ns
from datetime import datetime, timezone
from hashlib import blake2s
from random import randrange
from uuid import getnode as get_mac
```

What's Hash?

암호학적 해시

- 해시는 임의의 데이터를 고정된 길이의 데이터로 매핑하는 단방향 함수
- 같은 입력값에 대해서는 같은 출력값이 보장
- 출력값은 가능한 한 고른 범위에 균일하게 분포하는 특성

```python
base32 = "0123456789ABCDEFGHJKMNPQRSTVWXYZ"


def toBase32(n: int) -> str:
    if n < 32:
        return base32[n]
    return toBase32(n // 32) + base32[n % 32]
```

```
def LSID1(epoch: datetime = datetime(
            year=1970, month=1, day=1, hour=0,
            minute=0, second=0, microsecond=0, tzinfo=timezone.utc),
          worker_id: int = int(blake2s(
              key=get_mac().to_bytes(length=6, byteorder='big'),
              digest_size=2).hexdigest(), 16),
          sequence_id: int = randrange(0, 32 ** 2)) -> str:
```

```
1 def 함수(변수_이름: 변수_타입 = 기본값(함수_인자)) -> 반환_타입:
2     (...)
```

```
def LSID1(epoch: datetime = datetime(
            year=1970, month=1, day=1, hour=0,
            minute=0, second=0, microsecond=0, tzinfo=timezone.utc),
          worker_id: int = int(blake2s(
              key=get_mac().to_bytes(length=6, byteorder='big'),
              digest_size=2).hexdigest(), 16),
          sequence_id: int = randrange(0, 32 ** 2)) -> str:
```

```
LSID: str = ""
    now: datetime = datetime.now(timezone.utc)
    now_ns = time_ns()
```

```
1 # Date(5) # 5 * 5 = 25bit
2 diff = now - epoch
3 b32_date: str = toBase32(diff.days)
4 LSID += ('0' * (5 - len(b32_date)) + b32_date + '-')
```

```
1 # NS(8) # 8 * 5 = 40bit
2 ns = int((now_ns % (24 * 60 * 60 * 1000 * 1000 * 1000)) / 100)
3 b32_ns: str = toBase32(ns)
4 LSID += ('0' * (8 - len(b32_ns)) + b32_ns + '-')
```

```
# Worker ID(4) # 4 * 5 = 20bit
b32_wid: str = toBase32(worker_id)
if len(b32_wid) > 4:
    raise ValueError("Worker ID is too large (0<=int<=2^20-1)")
LSID += ('0' * (4 - len(b32_wid)) + b32_wid + "-")
```

```
# Random ID(2) # 2 * 5 = 10bit
b32_sid: str = toBase32(sequence_id)
if len(b32_sid) > 2:
    raise ValueError("Sequence ID is too large (0<=int<=2^10-1)")
LSID += ('0' * (2 - len(b32_sid)) + b32_sid)
```

00jtx-04fecrkm-0cgm-3n

Date: 25-bit

Timestamp(ns): 40-bit

Worker id: 20-bit

Sequence id: 10-bit

00jtx-04fecrkm-0cgm-3n

Date: 25-bit

Timestamp(ns): 40-bit

Worker id: 20-bit

Sequence id: 10-bit

00jtx-**04fecrkm**-0cgm-3n

Date: 25-bit

Timestamp(ns): 40-bit

Worker id: 20-bit

Sequence id: 10-bit

00jtx-04fecrkm-**0cgm**-3n

Date: 25-bit

Timestamp(ns): 40-bit

Worker id: 20-bit

Sequence id: 10-bit

00jtx-04fecrkm-0cgm-**3n**

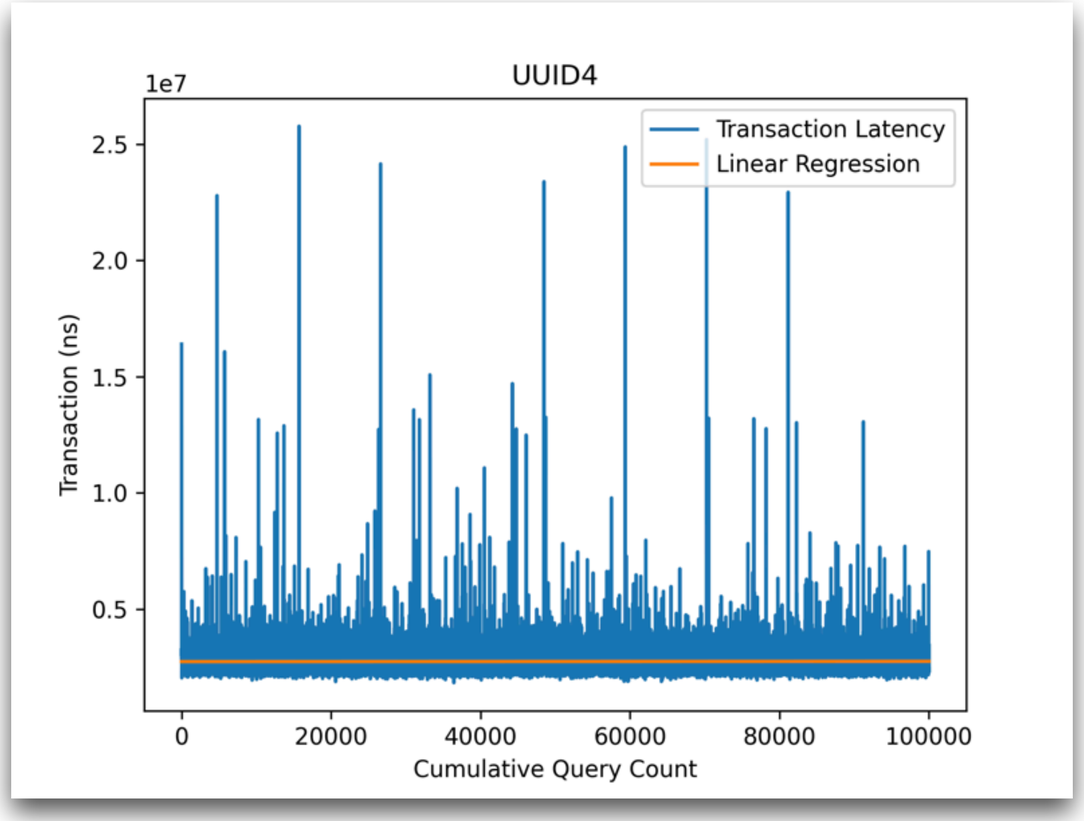
Date: 25-bit

Timestamp(ns): 40-bit

Worker id: 20-bit

Sequence id: 10-bit

```python
import uuid
from time import time_ns

import pandas as pd
import pymysql

conn = pymysql.connect(host='localhost', user='root', password='p5ssw0rd', database='benchmark')
cursor = conn.cursor()

x = []
y = []

for i in range(100000):
    start = time_ns()
    cursor.execute(f"INSERT INTO uuid4(id) VALUES('{uuid.uuid4()}');")
    conn.commit()
    end = time_ns()

    x.append(i)
    y.append((end - start))

    if i % 1000 == 0:
        print(i)

df = pd.DataFrame(x, columns=['id'])
df['insert_ns'] = y

df.to_csv('./csv_2/uuid4_2.csv', index=False)
```

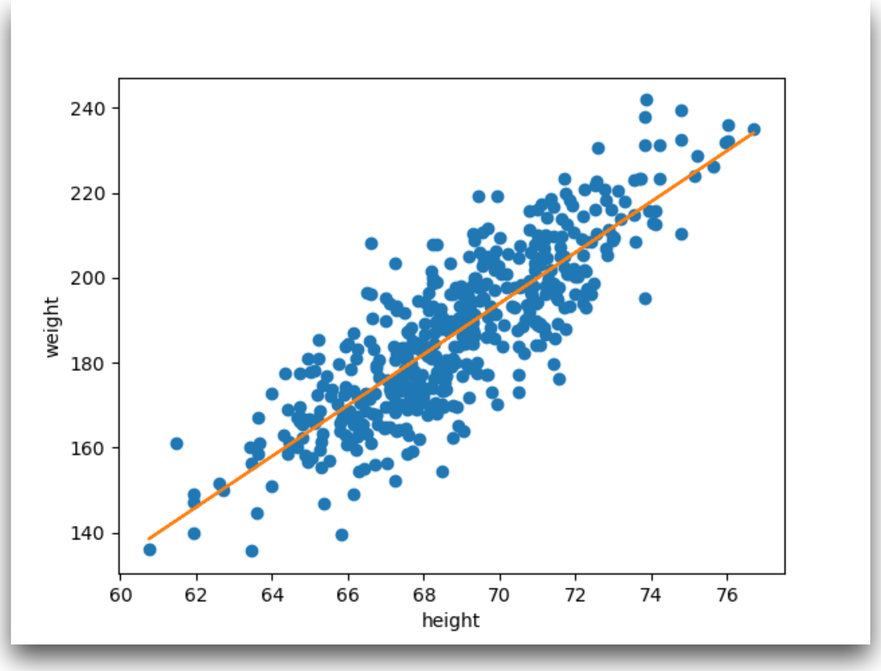

UUID4
1e7
Transaction Latency
Linear Regression
Transaction (ns)
2.5
2.0
1.5
1.0
0.5
0
20000
40000
60000
80000
100000
Cumulative Query Count

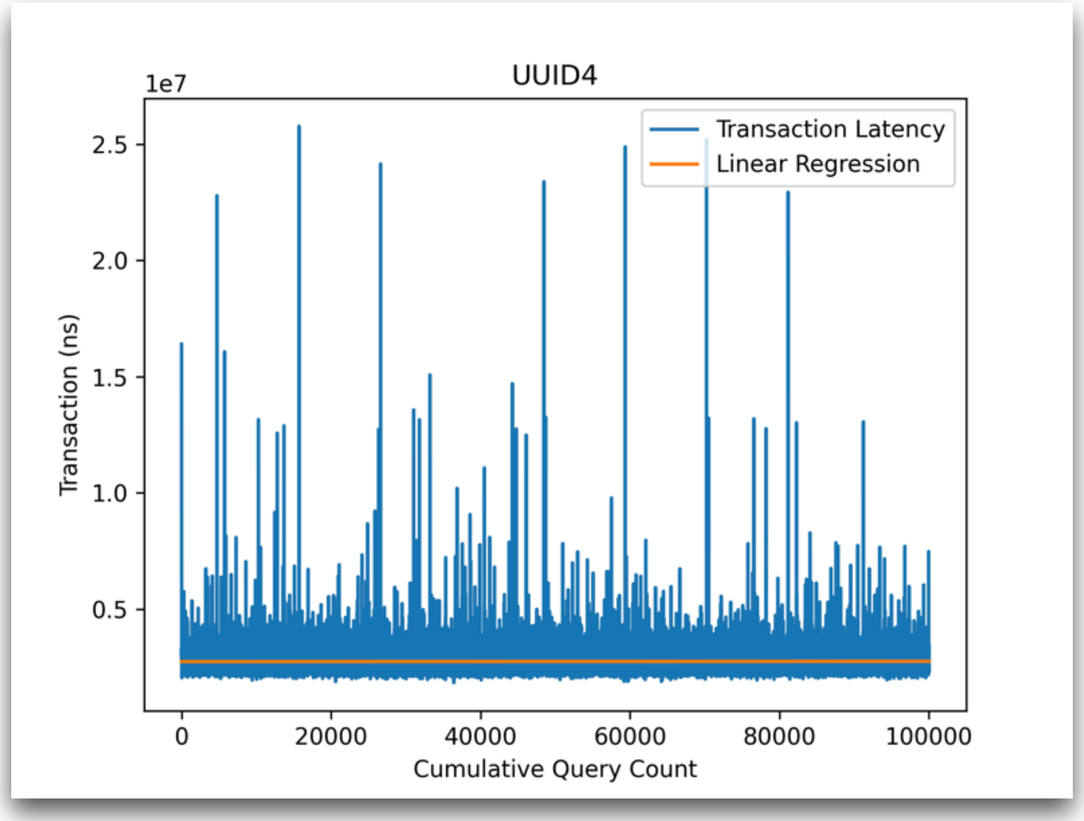
```python
import pandas as pd
from sklearn.linear_model import LinearRegression


def DOLR(df, title: str):
    X = df['id'].values.reshape(-1, 1)
    Y = df['insert_ns'].values.reshape(-1, 1)

    lr = LinearRegression()
    lr.fit(X, Y)

    print(title)
    print("coef_:", lr.coef_[0][0])
    print("intercept_", lr.intercept_[0])
    print(lr.score(X, Y))
    print()

    import matplotlib.pyplot as plt

    plt.plot(X, Y, '-', label='Transaction Latency')
    plt.plot(X, lr.predict(X), label='Linear Regression')
    plt.title(title)
    plt.xlabel('Cumulative Query Count')
    plt.ylabel('Transaction (ns)')
    plt.legend()
    plt.ylim(2000000, 4000000)
    plt.savefig(f"./plot/{title}_2.png", dpi=300)
    plt.show()


df = pd.read_csv('./csv_2/uuid4_2.csv')
DOLR(df, "UUID4")

```

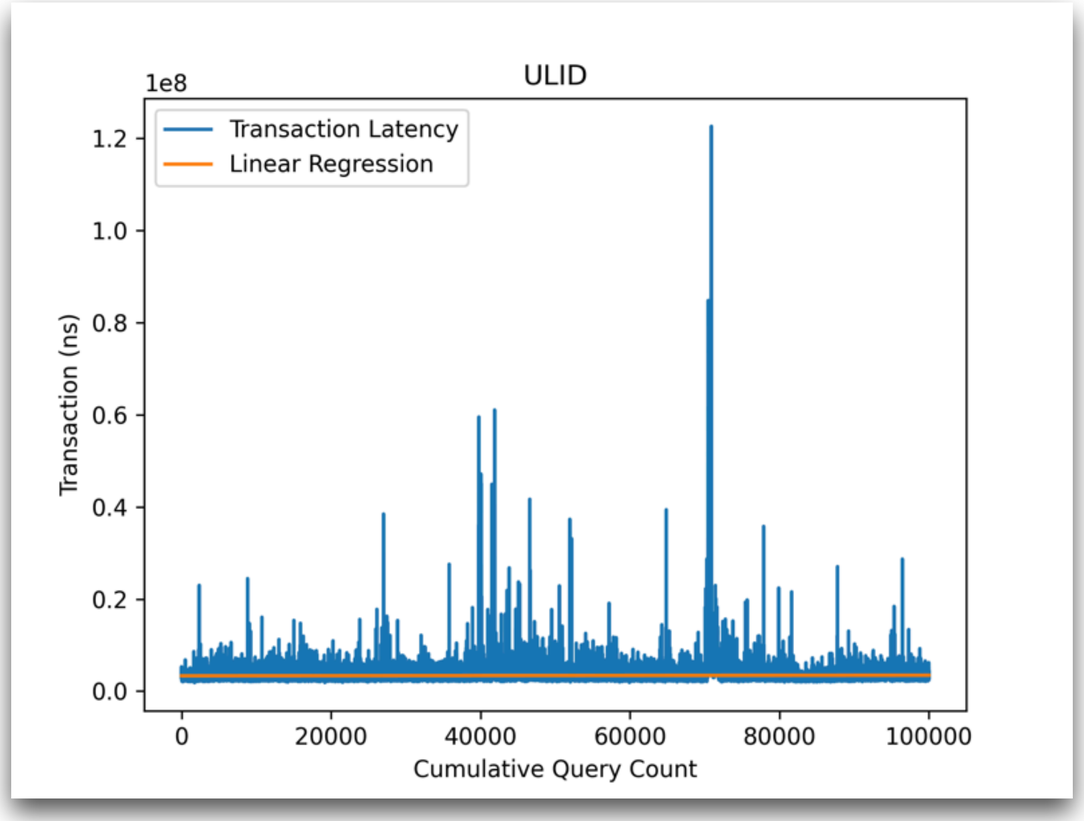
What's Linear Regression

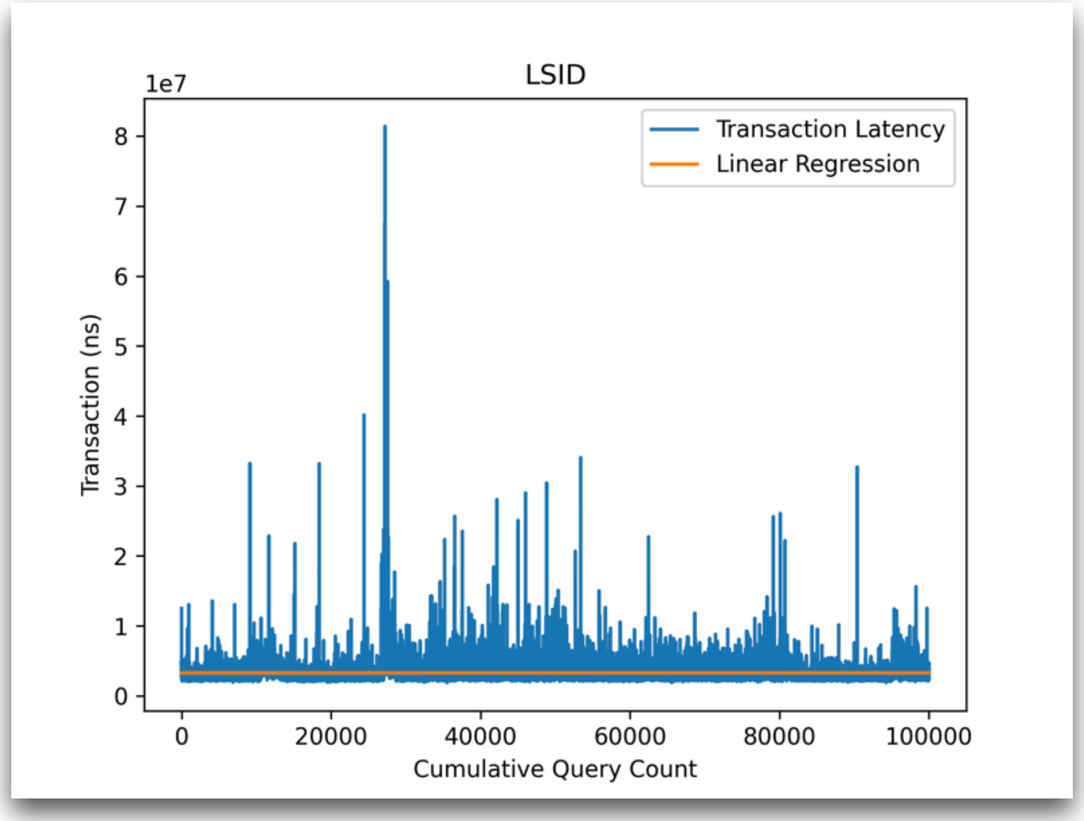
선형 회귀란 무엇인가

- 알려진 관련 데이터 값을 사용하여 알 수 없는 데이터의 값을 예측하는 데이터 분석 기법
- 종속 변수와 독립 변수 사이의 선형 관계를 찾는 분석




UUID4
1e7
Transaction Latency
Linear Regression
Transaction (ns)
2.5
2.0
1.5
1.0
0.5
0
20000
40000
60000
80000
100000
Cumulative Query Count


ULID
1e8
Transaction Latency
Linear Regression
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Transaction (ns)
0
20000
40000
60000
80000
100000
Cumulative Query Count


LSID
1e7
Transaction Latency
Linear Regression
Transaction (ns)
0
1
2
3
4
5
6
7
8
0
20000
40000
60000
80000
100000
Cumulative Query Count

	기울기	절편
UUID	0.136347408397646	3339880.93448226
ULID	0.185529453792534	3348060.9900751
LSID	0.0945419569594488	3273360.71942301

	기울기	절편
UUID	0.136347408397646	3339880.93448226
ULID	0.185529453792534	3348060.9900751
LSID	**0.0945419569594488**	**3273360.71942301**

감사합니다 :)