7장 : 분산 시스템을 위한 유일 ID 생성기 설계

난이도	⭐⭐⭐
학습날짜	@2025년 12월 17일

분산 시스템 ❌ 유일 ID 생성
    1. 데이터베이스 자동 증가(Auto-Increment)
분산 시스템 유일 ID 생성기
    1. 다중 마스터 복제 (Multi-master Replication)
    2. UUID
    3. 티켓 서버
    4. 트위터 스노플레이크 (Snowflake)

분산 시스템 ❌ 유일 ID 생성

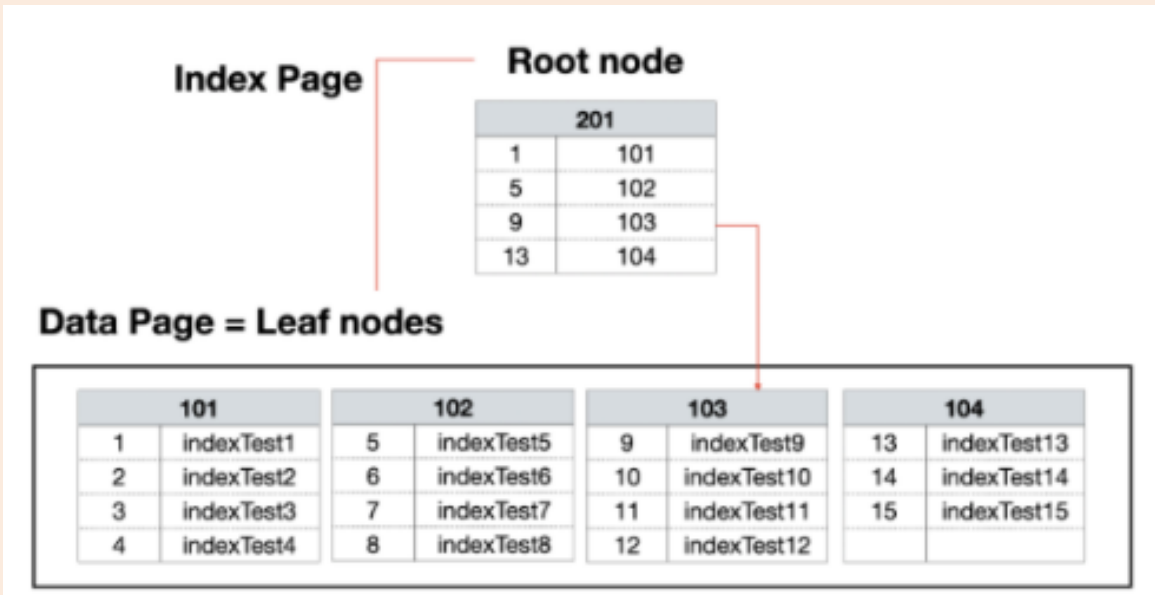
1. 데이터베이스 자동 증가(Auto-Increment)

보통 관계형 데이터베이스(RDBMS)의 기능을 직접 이용한다.

- DB 테이블의 특정 컬럼을 `AUTO_INCREMENT`(MySQL)나 `SERIAL`(PostgreSQL)로 설정
    - 새 레코드가 삽입될 때마다 DB가 값을 1씩 증가시키며 유일성을 보장한다.
- **장점:** 구현이 매우 쉽고, ID가 생성 순서에 따라 정렬되어 있어 인덱싱 성능이 좋습니다.

🤔 AUTO_INCREMENT가 인덱스 성능에 좋은 이유

(사전지식) PK - Clustered Index의 구조 [B+ 트리]



인덱스 구조 사진

Clustered Index: **테이블의 레코드를 지정된 컬럼에 의해 물리적으로 재배열하는 인덱스**를 의미한다.

- 테이블 당 1개만 존재할 수 있다
- PK 제약조건을 지정하는 컬럼에 대해 자동으로 Clustered Index가 생성된다.

Clustered Index를 구성하기 위해, 레코드를 해당 컬럼으로 정렬한 후에, 루트 페이지를 만들게 된다.

Clustered Index는 루트 페이지와 리프 페이지로 구성되며,

- **리프 페이지는 데이터 그 자체**이다.
- Index Page는 **키 값과 데이터 페이지 번호로 구성**하고, 검색하고자하는 **데이터의 키 값으로 페이지 번호를 검색**하여 데이터를 찾는다.

> 1. PK에 인덱스가 자동으로 생성되며,
> 2. 해당 인덱스는 자동 정렬된다.
> 3. 해당 인덱스를 Clustered Index라고 부른다.

단점
이미 정렬되어 있기 때문에 조회 속도면에서는 우수한 성능을 보인다.
다만, **추가/수정/삭제** 시 매번 레코드를 재정렬해야 하기 때문에 성능이 저하되는 단점이 있다.

- PK를 어떤 값으로 설정하는지에 따라 데이터베이스의 성능이 크게 좌우된다.

🗣️ ID를 PK로 설정하고 `AUTO_INCREMENT` 옵션을 주는 이유

- Q. 기획상 UNIQUE한 `email`로 PK를 설정하면 되지 않을까?
- Q. 복합키로 PK를 설정하면 추가 데이터 없이 데이터를 저장할 수 있지 않을까?

```
create table user {
  email varchar(100) not null, primary key,
  password varchar(100) not null
}
```

앞서 언급했듯이, PK로 설정된 값을 Clustered Index가 되어 자동 정렬된다.
`email`을 PK로 설정헌다.
email은 유저의 선택에 따라 값의 변동폭이 크다.
만약, a로 시작하는 이메일인 abc@kakao.com 을 입력했다고 해보자.

그러면, 데이터베이스는

1. abc@kakao.com의 위치를 찾고
2. 아래 레코드들을 전부 아래로 내리는 작업을 수행

만약, 유저가 1억명이라면 1억명 데이터의 위치를 재수정해야 하는 것이다. → 성능에 치명적이다.

따라서, 유저가 생성될때, PK를 Id로 지정 후 `auto_increment`를 사용하여 설계하는 것이다.

- **단점:** DB 의존성이 높습니다.
  - 여러 서버로 확장하기 어렵습니다(Single Point of Failure).
  - ID를 통해 데이터의 규모나 생성 속도가 노출될 수 있어 보안상 취약할 수 있습니다.

분산 시스템 유일 ID 생성기

분산 시스템에서 AUTO_INCREMENT를 사용하여 ID를 설계하면 다음과 같은 어려움이 발생할 수 있다.

1. **SPOF (Single Point of Failure)**

   하나의 서버가 하나의 DB에 의존하여 AUTO_INCREMENT를 수행하고 id 값을 받아와야 한다면, 해당 DB가 다운되는 순간 전체 시스템의 "쓰기"작업이 중단된다.

2. **네트워크 지연**

   네트워크를 타고 다녀와야 하기 때문에 시간적 지연 발생

3. **분산 DB에서의 데이터 중복**

   서로 다른 DB에서 `AUTO_INCREMENT`를 수행할 경우, ID가 중복되는 경우가 발생할 수 있습니다.

4. **인덱스 경합**

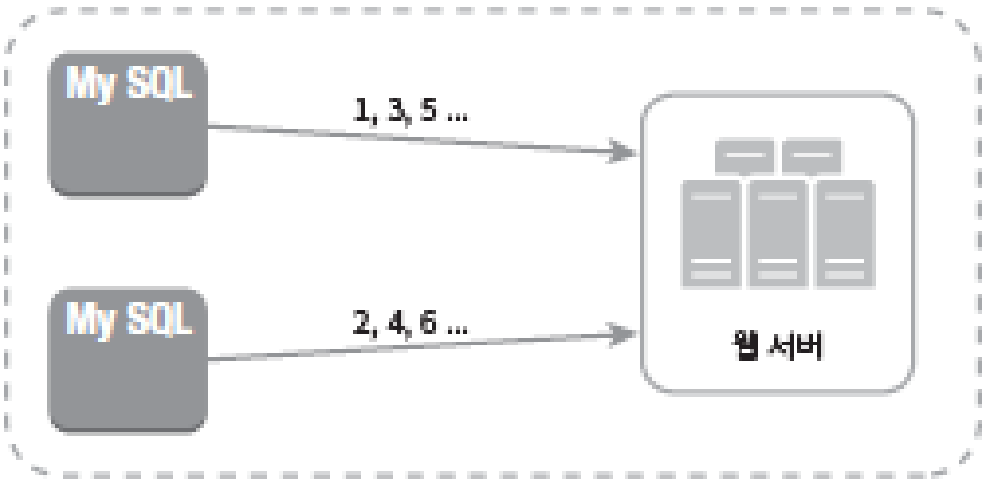
   `AUTO_INCREMENT`는 유일한 번호를 보장하기 위해 내부적으로 락(Lock)을 사용합니다.

   수많은 서버가 동시에 락을 위해 큐에 들어가게 되면 성능이 급격히 저하될 수 있습니다.

이와 같은 문제로, 분산 시스템에서는 유일 ID를 생성할 수 있는 시스템을 도입하여 ID를 도입한다.

---

1. 다중 마스터 복제 (Multi-master Replication)



다중 마스터 복제 방식으로, AUTO_INCREMENT를 활용하는 방식이다.

다만, 기존에는 1씩 증가시켰다면 이제는 현재 사용 중인 데이터베이스 서버의 수인 K씩 증가시키는 방법이다.

- K : 데이터베이스의 개수

해당 방식을 통해, 중복된 ID를 방지할 수 있다.

단점

- 여러 데이터 센터에 걸쳐 규모를 늘리기가 어려워진다.
- ID값 전체를 두고 보았을 때는, 시간 흐름에 맞춰서 증가하였다고 보장할 수 없다
- 서버를 추가하거나 / 삭제하였을 때도 정상 동작하도록 만들기 어렵다

2. UUID

UUID는 컴퓨터 시스템에 저장되는 정보를 유일하게 식별하기 위한 128비트 수이다.

- UUID는 설계 로직이 충돌 가능성이 0에 수렴하도록 설계되었다.

단점

- ID가 128비트로 길다. DB 용량을 많이 차지한다.
- ID를 시간순으로 정렬할 수 없다.
- ID에 숫자가 아닌 값이 포함될 수 있다.

🤔 UUID가 랜덤으로 값을 만들면, **인덱스 정렬 (Clustered Index)**에서 치명적 성능을 보이지 않을까?
→ ❌ UUID v7 부터는 아니다.
UUID는 버전이 존재하고 버전별로 많이 바뀌었었다.

버전	생성 원리
v4	순수 랜덤
v6	timestamp + MAC
v7	**timestamp + random (정렬 최적화)**

UUID v7은 **가장 앞 비트에 시간 정보를 둔다.**

[v7 비트 배치 (RFC 9562)]

- **상위 48비트 = Unix timestamp (milliseconds)**
- 그 뒤에 랜덤성

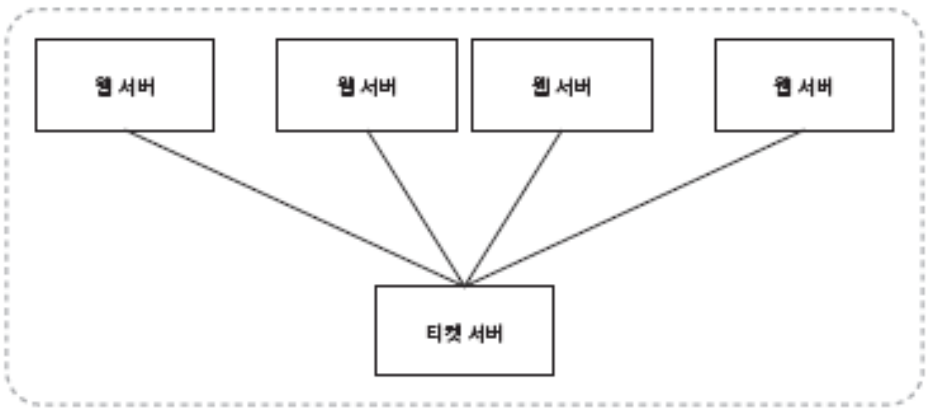
```
| 48 bits | 4 bits | 12 bits | 62 bits |
|---------|--------|---------|---------|
| unix ms | version| counter | random  |
```

단, "완전히 정렬된다고 말할 수는 없다"

- 같은 millisecond 내에서는 random 영역

3. 티켓 서버

티켓 서버는 AUTO_INCREMENT 기능을 갖춘 중앙 데이터베이스 서버, 즉 티켓 서버를 사용하는 방식이다.



티켓 서버 방식

단점

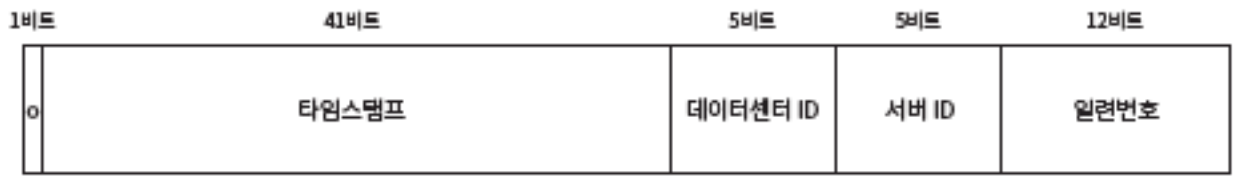
- 티켓 서버의 SPOF에 취약하다.
  - 티켓 서버를 위한 서버, 티켓 서버를 위한 서버의 서버 등 대응이 어렵다.

4. 트위터 스노플레이크 (Snowflake)

스노플레이크 전략은 트위터에서 분산 시스템에 활용될 ID의 구조를 만들어서 사용한 방식이다.

- 정렬 가능하면서도 중복되지 않는 64비트 ID를 생성하기 위한 영리한 방법.

기본적으로, 다음의 구조를 지니고 있다. (8Byte)



스노플레이크 구조

1. **사인 비트**

   음수와 양수를 구별하는데 사용

   📌 사인 비트는 **항상 0으로 설정**된다?
   찾아보니, 사인 비트는 0 또는 1로 설정하는 것이 아닌, 항상 0으로 설정한다고 한다.

   1. **"2의 보수"** 방식에서 **부호가 있는 정수 (Signed Integer)로 처리**할 때 발생할 수 있는 **오류를 차단**하기 위해서라고 한다.

   - 많은 언어에서 사인 비트를 1로 사용하게 되면, 해당 숫자를 **음수로 해석**하게 된다.
     - ID가 음수가 되면, 대소 비교시 로직이 꼬이거나, UI 화면에서 ID를 출력할 때 Integer Overflow가 계산된 음수 값이 도출될 수 있다.

   2. 생성 시간 정렬을 보장하기 위해
      타임스탬프를 사용하는 이유는 **"ID값 자체의 크기 순서 = 생성 시간 순서"**를 보장하기 위해서인데, 0과 1을 혼합해서 사용하면 해당 규칙이 깨진다.

   🤔 그럼 걍 지워버리면 안됨?

   - 64bit Fixed Width 유지
   - CPU 정렬 최적화
   - 기존 언어 타입 재활용 가능성 (Java Long (signed))

2. **타임스탬프**

   타임스탬프는 시간의 흐름에 따라 점점 큰 값을 갖게 되므로, 결국 ID는 시간 순으로 정렬 가능하게 된다.

   41비트로 표현할 수 있는 타임스탬프의 최대값은 2^41-1로, 약 69년에 해당한다.

   69년이 지나면 기원 시각을 바꾸거나, ID 체계를 다른 것으로 이전해야 한다.

   🤔 타임스탬프는 지리적으로는 동일하다 → 생성 방식 자체가 그리니치 시간대를 기준으로 하기 때문에

   - 물리적으로 다른 방법을 쓴다 → 여러 코어에서 실행될 경우 유효하지 않을 수 있다.

   🤔 분산 환경에서 타임스탬프를 신뢰하지 않는다?

   - 정합성 / 순서 결정

3. **데이터센터 ID**

   32(5bit)개의 데이터센터를 지원할 수 있다.

4. **서버 ID**

   데이터센터당 32개(5bit) 서버를 활용할 수 있다.

5. **일련번호**

   각 서버는 ID를 생성할 때마다 이 일련번호를 1만큼 증가시킨다.

   - 이 값은 1ms가 경과할 때마다 0으로 초기화(Reset)된다.

   12비트로, 4096개의 값을 가질 수 있다.

어떤 서버가 같은 밀리초 동안 하나 이상의 ID를 만들어 낸 경우에만 0보다 큰 값을 가진다.

📌 **스노플레이크가 오름차순**을 보장하는 방법

- 타임스탬프 + 일련번호

기본적으로, 타임스탬프는 매초 증가하는 값을 가지므로, 자연스럽게 증가하도록 설계된다.
다만, 동일 시간에 여러 개의 ID가 생성되는 경우에 같은 타임스탬프를 가진 여러개의 ID가 존재할 수 있다.
이를 처리하 귕해 일련번호가 등장한다.
일련번호를 통해, 같은 타임스탬프를 가졌더라도 서로 다른 ID를 지니도록 증가하는 일련번호를 설계한 것이다.