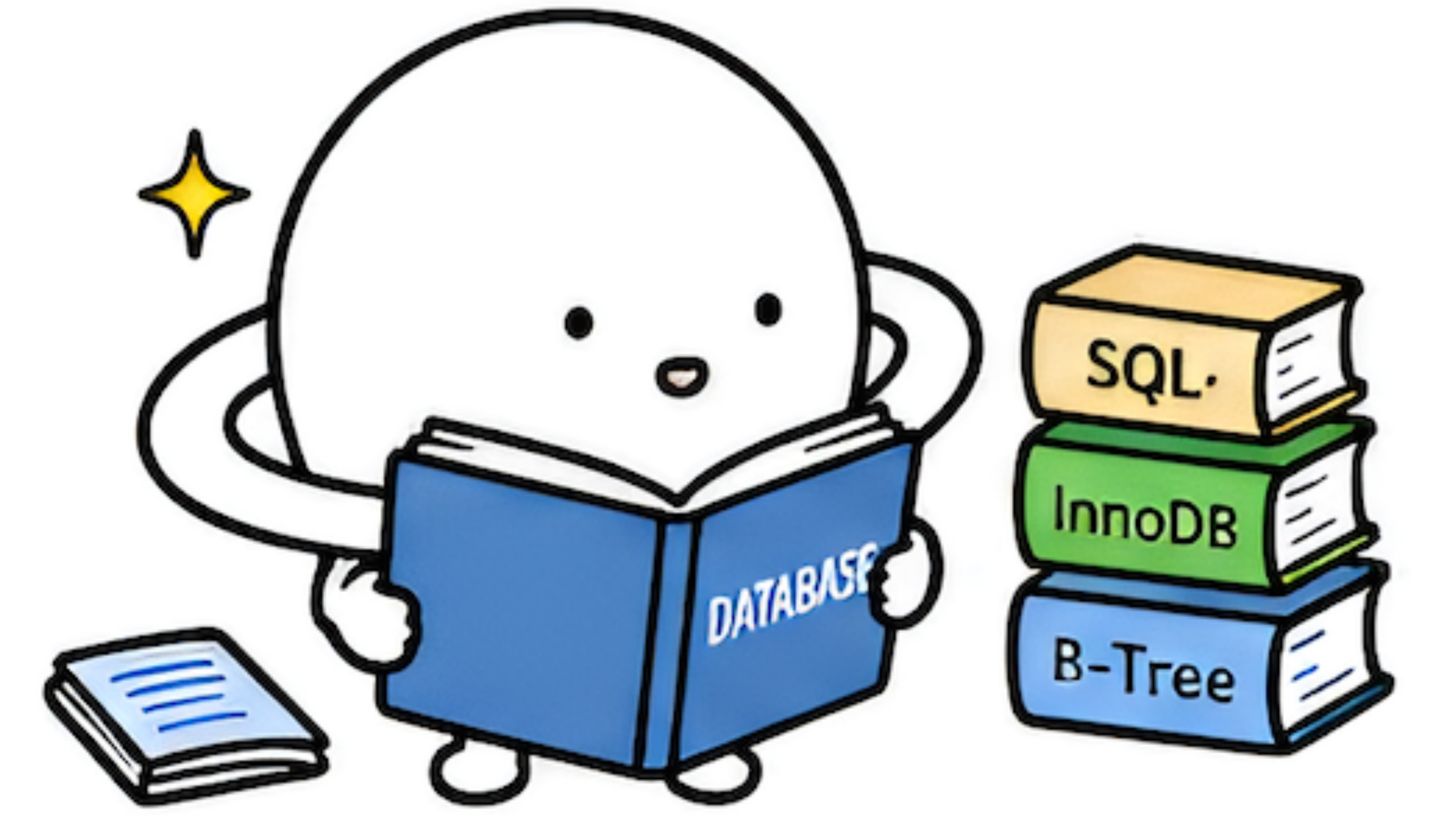


DB 인덱스는 왜 빠르고, 언제 느려질까??

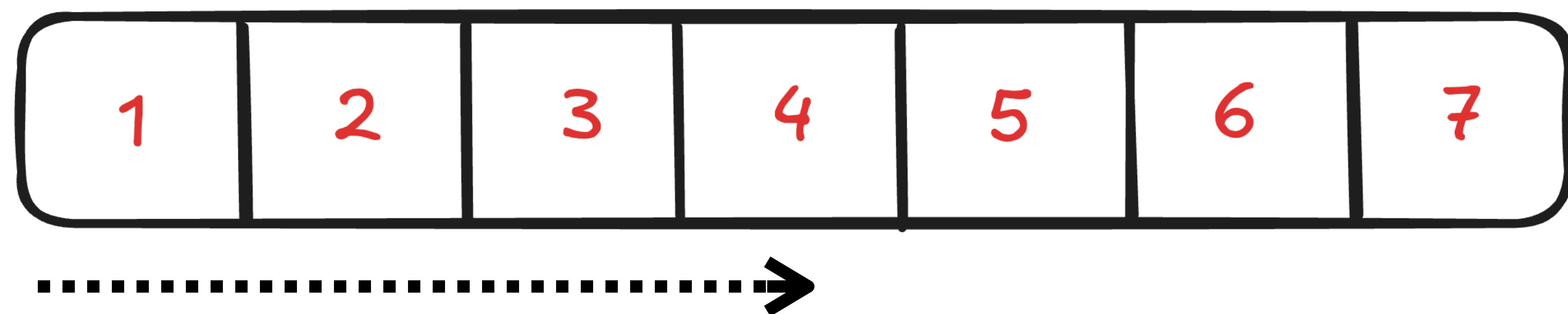
MySQL InnoDB의 B-Tree 구조와 UUID 기본키 실험으로 이해하기

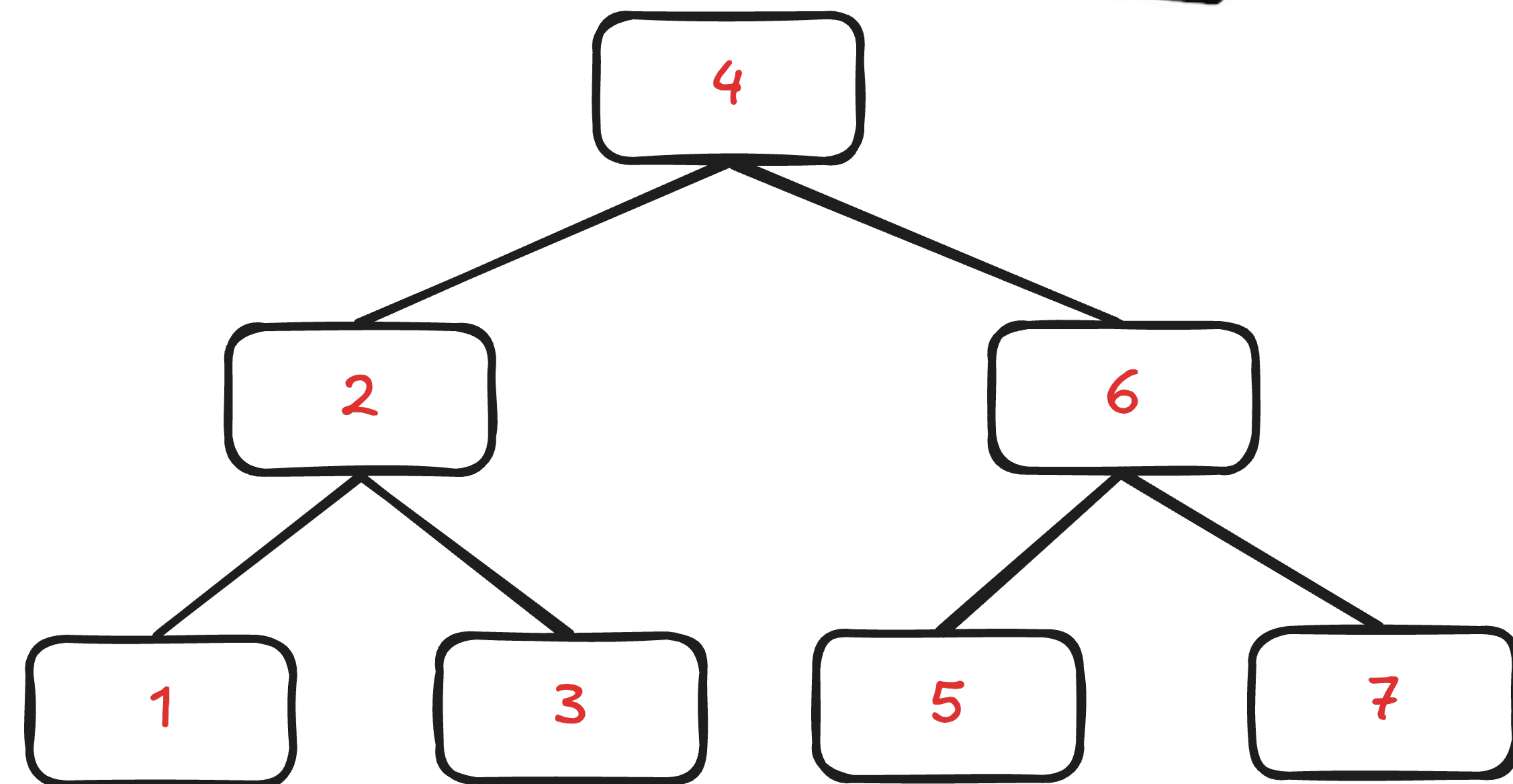
8기 BE 스타크

인덱스란??

원하는 데이터를 빠르게 찾기 위해
미리 만들어둔 별도의 자료구조

인덱스는 왜 빠를까?

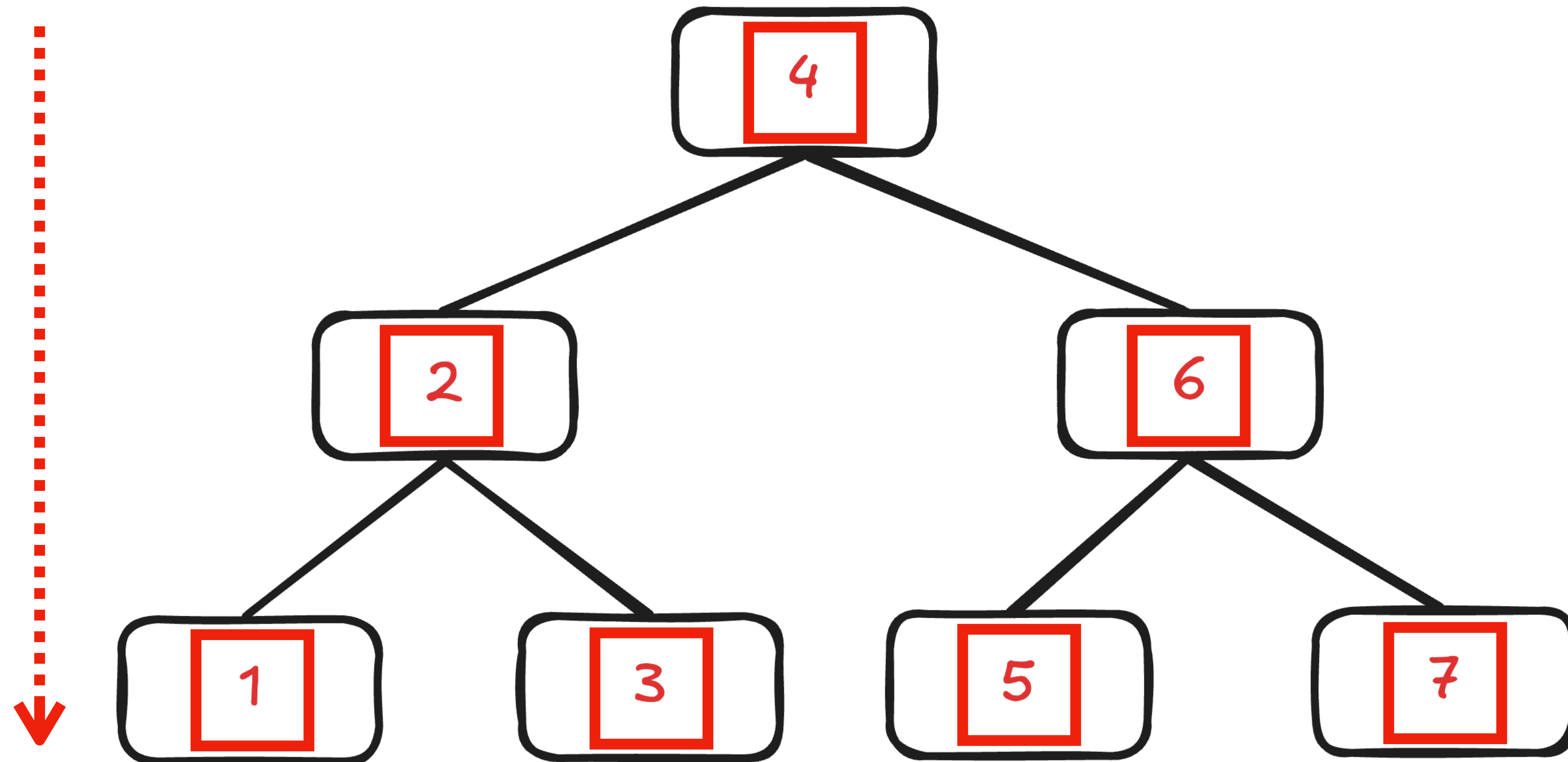




인덱스 없는 경우
처음부터 순차 탐색 O(n)

인덱스 있는 경우
절반씩 범위 제거 O(log n)

이진 탐색 트리(BST)의 문제



데이터 많을 때
노드 1억 개
→ depth 깊어짐
→ depth 약 27개

노드 당 디스크 I/O 1번
→ 디스크 I/O 27번

속도 저하!!

MySQL이 선택한 인덱스 구조

MySQL의 내부 DB 엔진, InnoDB

InnoDB가 사용하는 B-Tree 구조

MySQL이 선택한 인덱스 구조

10.3.1 How MySQL Uses Indexes

Indexes are used to find rows with specific column values quickly. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows. The larger the table, the more this costs. If the table has an index for the columns in question, MySQL can quickly determine the position to seek to in the middle of the data file without having to look at all the data. This is much faster than reading every row sequentially.

Most MySQL indexes (`PRIMARY KEY`, `UNIQUE`, `INDEX`, and `FULLTEXT`) are stored in B-trees. Exceptions: Indexes on spatial data types use R-trees; `MEMORY` tables also support hash indexes; `InnoDB` uses inverted lists for `FULLTEXT` indexes.

B-Tree 구조 사용!!

MySQL이 선택한 인덱스 구조

B-tree

A tree data structure that is popular for use in database indexes. The structure is kept sorted at all times, enabling fast lookup for exact matches (equals operator) and ranges (for example, greater than, less than, and `BETWEEN` operators). This type of index is available for most storage engines, such as `InnoDB` and `MyISAM`.

Because B-tree nodes can have many children, a B-tree is not the same as a binary tree, which is limited to 2 children per node.

Contrast with ***hash index***, which is only available in the `MEMORY` storage engine. The `MEMORY` storage engine can also use B-tree indexes, and you should choose B-tree indexes for `MEMORY` tables if some queries use range operators.

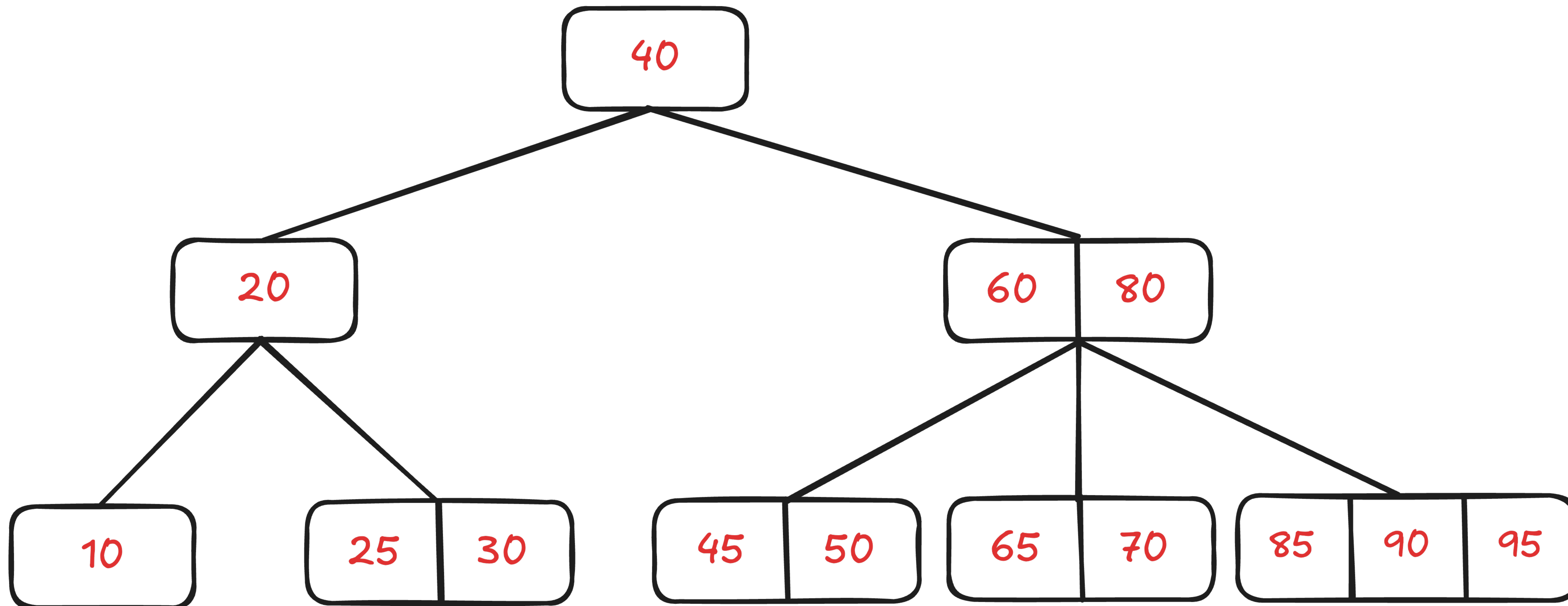
The use of the term B-tree is intended as a reference to the general class of index design. B-tree structures used by MySQL storage engines may be regarded as variants due to sophistications not present in a classic B-tree design. For related information, refer to the `InnoDB` Page Structure Fil Header section of the MySQL Internals Manual.

See Also hash index.

전통적인 B-Tree 구조와는 다르다

B+Tree 구조를 사용

B-Tree



데이터 1억 건

→ depth 약 3~4개

→ 디스크 I/O 3~4번

정렬된 균형 트리, 모든 노드에 키와 값이 담길 수 있음

노드 하나에 여러개(수백 ~ 수천)의 키, 여러개의 자식 노드

B+Tree

40

20

60 | 80

5 | 10 → 20 | 25 | 30 → 40 | 45 | 50 → 60 | 65 | 70 → 80 | 85 | 90

리프 노드에만 값이 존재함.

각 리프 노드들이 연결되어 있음

B+Tree의 장점

범위 기반 탐색에 매우 유리

예시로 살펴보기 - 단건 조회

단건 조회 id = 65

B-Tree 구조

방문 노드 개수 : 3

예시로 살펴보기 - 단건 조회

단건 조회 id = 65

B+Tree 구조

방문 노드 개수 : 3

예시로 살펴보기 - 범위 조회

범위 조회 25 <= id <= 65

B-Tree 구조

방문 노드 개수 : 12

40 ← Root Node

20 | 60 80 ← Branch Node

5 10 | 25 30 | 45 50 | 65 70 | 85 90 ← Leaf Node

예시로 살펴보기 - 범위 조회

범위 조회 25 <= id <= 65

B+Tree 구조

방문 노드 개수 : 5

미션에서 생각해보았던 기본키 전략

자동 증가 기본키로는
영속화 전 객체의 동등성 비교가 불가능하다

미션에서 생각해보았던 기본키 전략

그렇다면 객체 생성 시에 랜덤 UUID와 같은 것을 통해 고유한 기본키를 생성해주면 안될까?

기본키 전략에 따른 인덱스 삽입의 차이

자동 증가 기본키

vs

랜덤 UUID 기본키

자동 증가 기본키 사용

Auto Increment 방식 기본키

랜덤 UUID 기본키 사용

랜덤 UUID 기본키 사용 – 페이지 분할

- 기존 페이지의 일부 데이터를 새 페이지로 옮긴다.
- 부모 노드의 key도 갱신될 수 있다.
- 인덱스 구조를 다시 정리해야 한다.
- 랜덤한 위치에 삽입되기 때문에 캐시 효율도 떨어질 수 있다.

새로운 키 삽입 시 자동 증가 기본 키보다 더 많은 작업이 일어난다.

실제 성능 차이

실제로는 얼마나 성능 차이가 날까???

실험 구성

기본키 전략	설명
BIGINT AUTO_INCREMENT	DB가 순차적으로 증가시키는 숫자 기본키
CHAR(36) UUID_v4	랜덤 UUID를 문자열로 저장
BINARY(16) UUID_v4	랜덤 UUID를 16바이트 이진 값으로 저장
CHAR(36) UUID_v7	시간 정보를 포함한 UUID를 문자열로 저장
BINARY(16) UUID_v7	시간 정보를 포함한 UUID를 16바이트 이진 값으로 저장
CHAR(26) ULID	시간 정보와 랜덤 값을 조합한 ULID를 문자열로 저장

실험 구성

사용 테이블

```
CREATE TABLE reservation (
    id          ...          NOT NULL,
    name        VARCHAR(64)  NOT NULL,
    created_at  DATETIME(6)  NOT NULL,
    PRIMARY KEY (id),
    INDEX idx_created_at (created_at)
) ENGINE = InnoDB;
```

측정 항목

항목	의미
insert	빈 테이블에 1,000,000건 삽입하는 시간
insertLarge	이미 1,000,000건이 들어 있는 테이블에 100,000건을 추가 삽입하는 시간
lookup	기본키로 10,000번 단건 조회하는 시간
range_createdAt	created_at 기준으로 1,000건 범위 조회를 100번 수행하는 시간
range_PK	PRIMARY_KEY 기준으로 1,000건 범위 조회를 100번 수행하는 시간
size	테이블과 인덱스가 사용하는 저장 공간

1. insert - 데이터 최초 삽입 1,000,000 건

항목	평균 시간
BIGINT AUTO_INCREMENT	16,747 ms
CHAR(36) UUID_v4	23,974 ms
BINARY(16) UUID_v4	22.940 ms
CHAR(36) UUID_v7	21,260 ms
BINARY(16) UUID_v7	19,611 ms
CHAR(26) ULID	20,550 ms

2. insertLarge - 백만건 삽입 후 100,000 건 삽입

항목	평균 시간
BIGINT AUTO_INCREMENT	1,679 ms
CHAR(36) UUID_v4	6,161 ms
BINARY(16) UUID_v4	4,796 ms
CHAR(36) UUID_v7	2,283 ms
BINARY(16) UUID_v7	1,961 ms
CHAR(26) ULID	2,020 ms

자동 증가 기본키와 랜덤 기본키 삽입 시간 비교

실험 조건	삽입 건수	AUTO_INCREMENT	UUID_v4	차이
빈 테이블에 최초 삽입 (insert)	1,000,000 건	0.0167 ms / 건	0.0240 ms / 건	약 1.43배
백만건 존재, 추가 삽입 (insertLarge)	100,000 건	0.0168 ms / 건	0.0616 ms / 건	약 3.67배

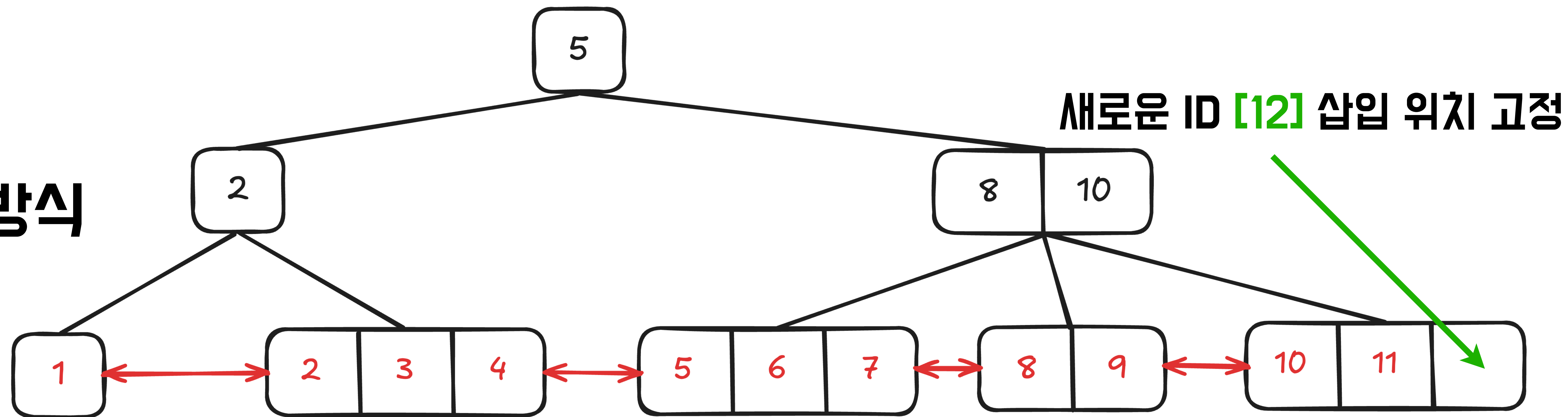
자동 증가 기본키는
두개 실험 간 차이가 없다.

랜덤 기본키는
두개 실험 간 차이가 크다.

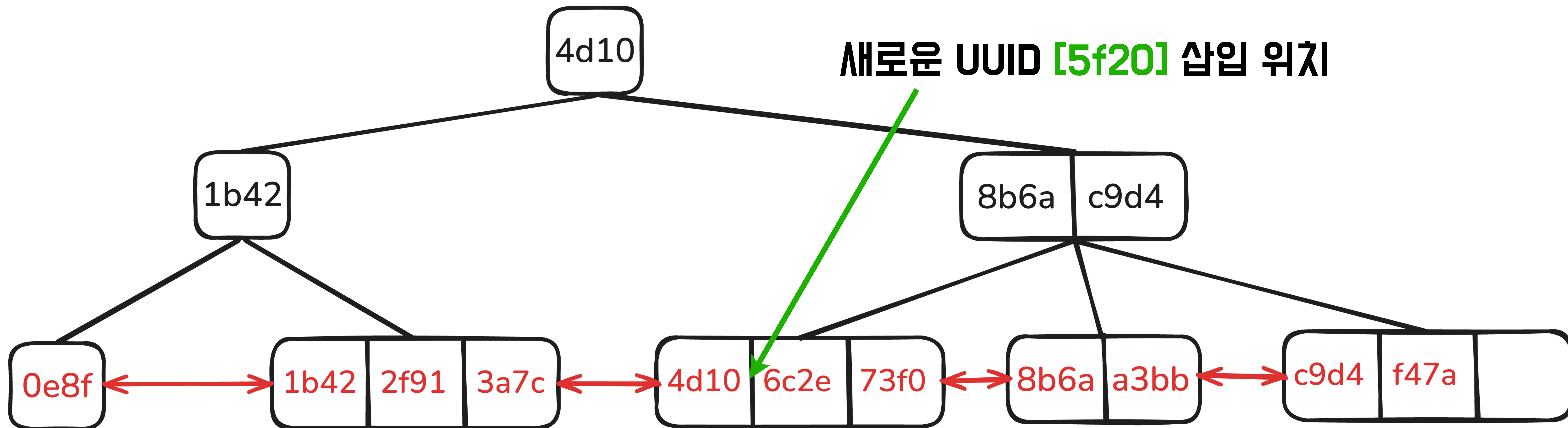
데이터가 쌓인 뒤에는 UUID v4의 랜덤 삽입 비용이 더 크게 드러난다.

자동 증가 기본키 vs 랜덤 기본키

Auto Increment 방식



UUIDv4 방식



3. lookup - 기본키 단건 조회 10,000건

항목	평균 시간
BIGINT AUTO_INCREMENT	2,027 ms
CHAR(36) UUID_v4	2,394 ms
BINARY(16) UUID_v4	2,213 ms
CHAR(36) UUID_v7	2,136 ms
BINARY(16) UUID_v7	2,214 ms
CHAR(26) ULID	2,027 ms

또 너냐…?

InnoDB 인덱스 구조

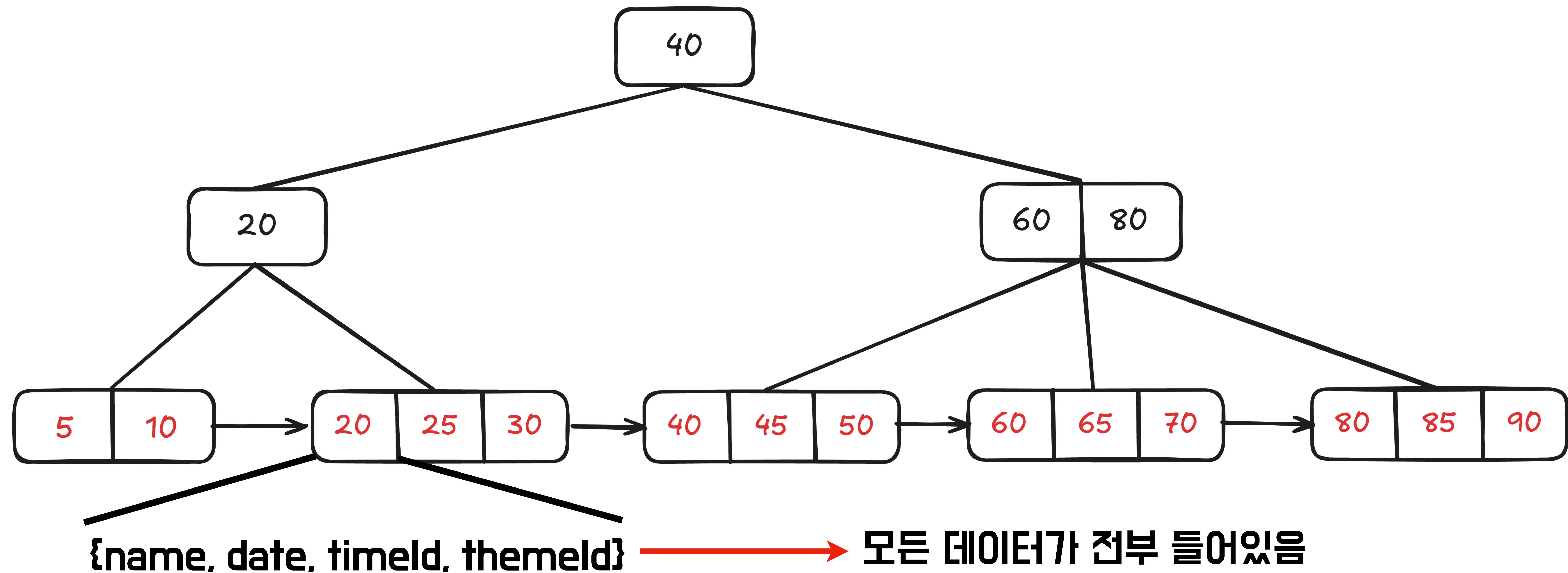
Clustered Index (클러스터드 인덱스)
= 기본키 인덱스

Secondary Index
= 기본키가 아닌 다른 속성에 대한 인덱스

InnoDB 인덱스 구조 - 클러스터드 인덱스

클러스터드 인덱스는 인덱스 내부에 전체 행의 정보가 전부 들어가 있다.

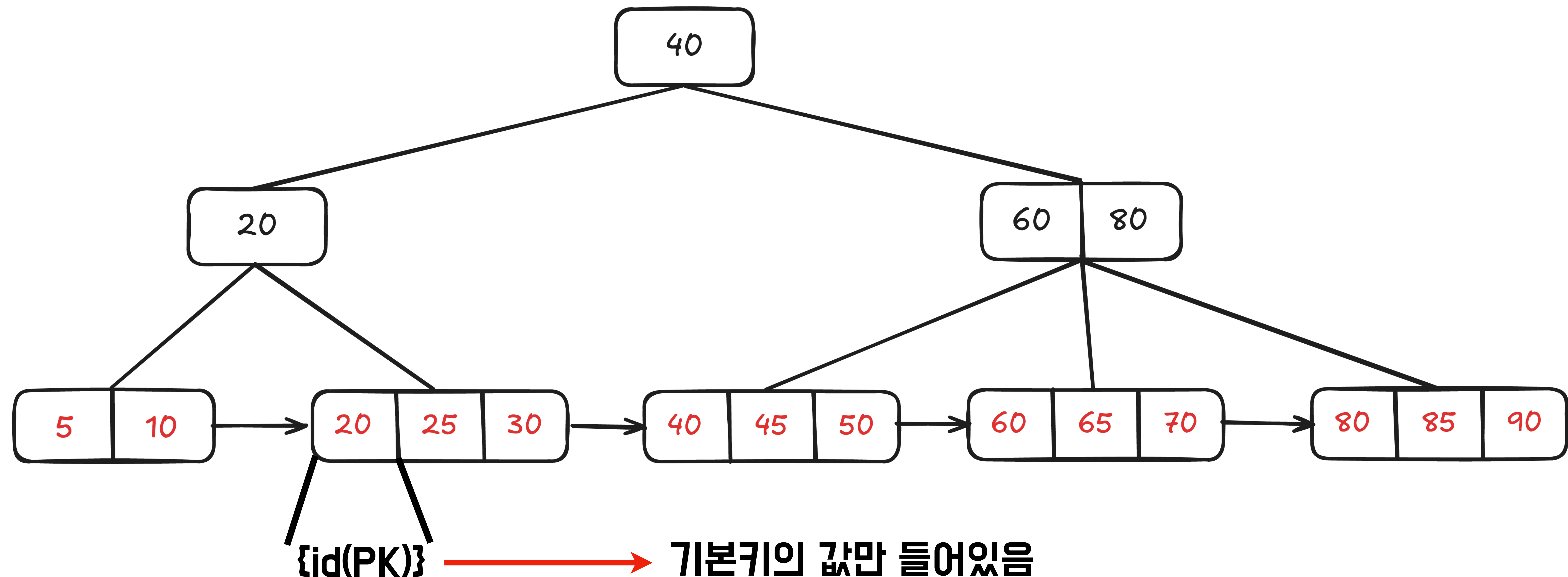
리프 노드를 조회하면 그걸로 끝



InnoDB 인덱스 구조 - 세컨더리 인덱스

세컨더리 인덱스는 인덱스 내부에 기본키의 값만 들어가 있다.

리프 노드를 조회해서 얻은 기본키의 값으로 다시 클러스터드 인덱스 조회(더블 룩업)



4. range_createdAt - 세컨더리 인덱스

항목	평균 시간
BIGINT AUTO_INCREMENT	369 ms
CHAR(36) UUID_v4	3,722 ms
BINARY(16) UUID_v4	1,275 ms
CHAR(36) UUID_v7	730 ms
BINARY(16) UUID_v7	473 ms
CHAR(26) ULID	622 ms

세컨더리 인덱스에서는
시간 차이가 큰 모습

5. range_PK - 클러스터드 인덱스

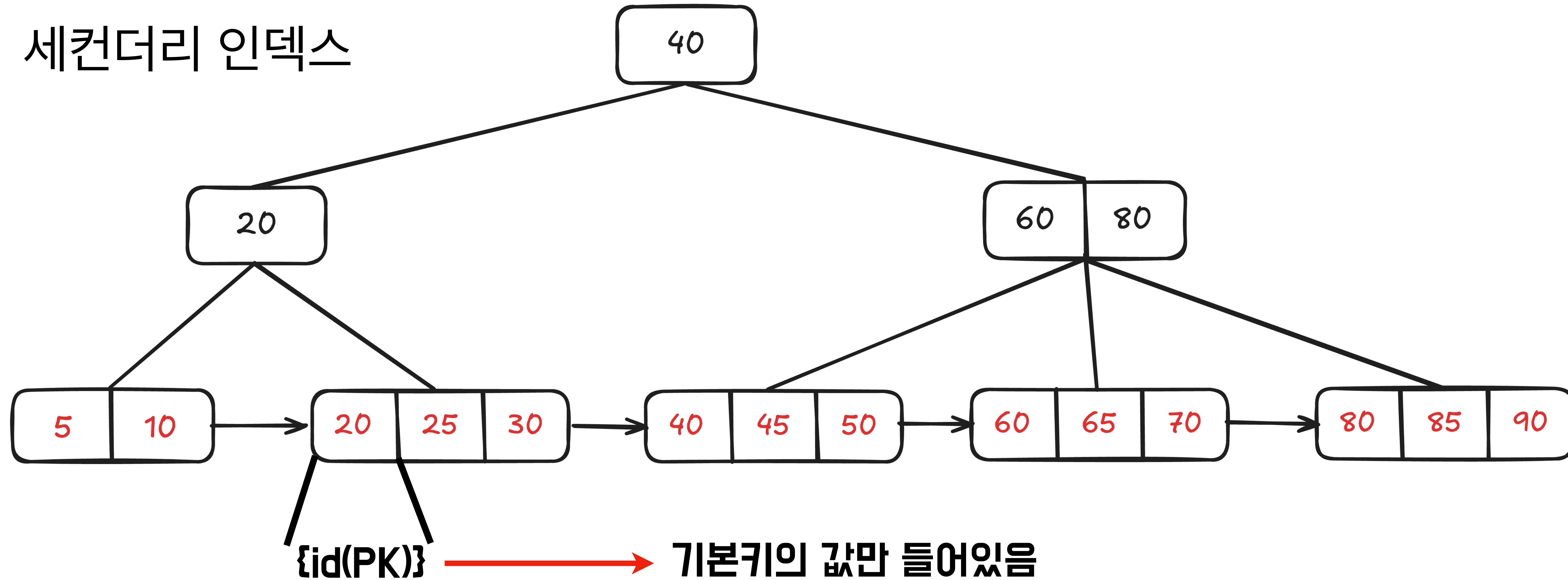
항목	평균 시간
BIGINT AUTO_INCREMENT	361 ms
CHAR(36) UUID_v4	308 ms
BINARY(16) UUID_v4	295 ms
CHAR(36) UUID_v7	334 ms
BINARY(16) UUID_v7	342 ms
CHAR(26) ULID	316 ms

세컨더리 인덱스와 달리
시간 차이가 나지 않는다

클러스터드 인덱스는 이미 정렬되어 있다

세컨더리 인덱스 범위 조회는 왜 다를까?

세컨더리 인덱스의 범위 조회 흐름

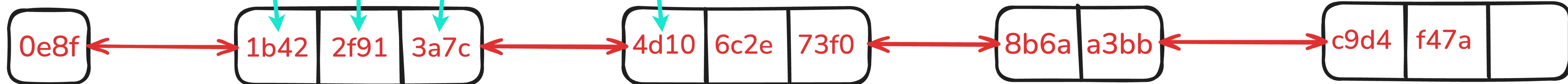


조회한 기본키로 다시 클러스터드 인덱스 탐색. 이때가 차이가 발생하는 지점!!!

UUID v7 / ULID — 생성 시간 순서 = 저장 순서

세컨더리 인덱스

클러스터드 인덱스



UUID v4 — 생성 시간 순서 ≠ 저장 순서

세컨더리 인덱스

클러스터드 인덱스

6. size

항목	data (MB)	index (MB)	total (MB)
BIGINT AUTO_INCREMENT	61.59	23.56	85.16
CHAR(36) UUID_v4	152.67	54.72	207.39
BINARY(16) UUID_v4	111.77	31.59	143.36
CHAR(36) UUID_v7	93.80	54.72	148.52
BINARY(16) UUID_v7	70.64	31.59	102.23
CHAR(26) ULID	82.72	43.66	126.38

6. size - 기본키는 가능한 짧게

How Secondary Indexes Relate to the Clustered Index

Indexes other than the clustered index are known as secondary indexes. In `InnoDB`, each record in a secondary index contains the primary key columns for the row, as well as the columns specified for the secondary index. `InnoDB` uses this primary key value to search for the row in the clustered index.

If the primary key is long, the secondary indexes use more space, so it is advantageous to have a short primary key.

For guidelines to take advantage of `InnoDB` clustered and secondary indexes, see Section 10.3, "Optimization and Indexes".

기본키가 길면 세컨더리 인덱스가 더 큰 공간을 사용하기 때문에

짧을 수록 더 이점이 있다.

기본키 전략은 단순한 선택이 아니다

인덱스를 이해해야 비로소 좋은 쿼리를 만들 수 있다.

삽입 성능　　조회 성능　　저장 공간

영속화 전 식별자가 필요하다면? → UUID v7 / ULID

감사합니다