

JOIN과 B-Tree

발표자: 김광일

사용된 데이터 및 테이블

departments

department_id	department_name
1	Engineering
2	Human Resources
3	Sales

employees

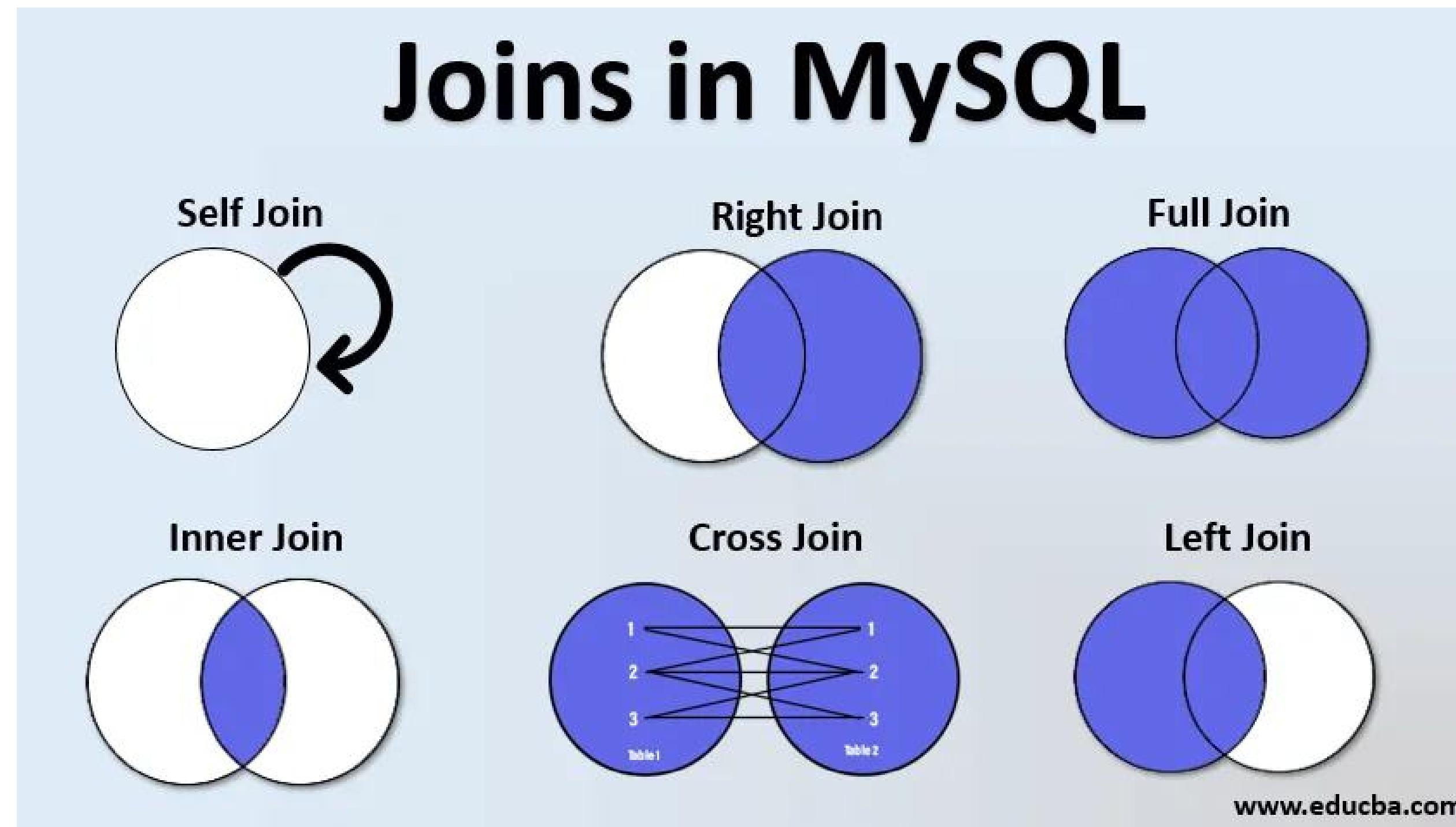
employee_id	first_name	last_name	department_id
1	John	Doe	1
2	Jane	Smith	2
3	Alice	Johnson	1
4	Bob	Brown	3
5	Charlie	Davis	NULL

발표자: 김광일

JOIN의 종류

발표자: 김광일

JOIN의 종류

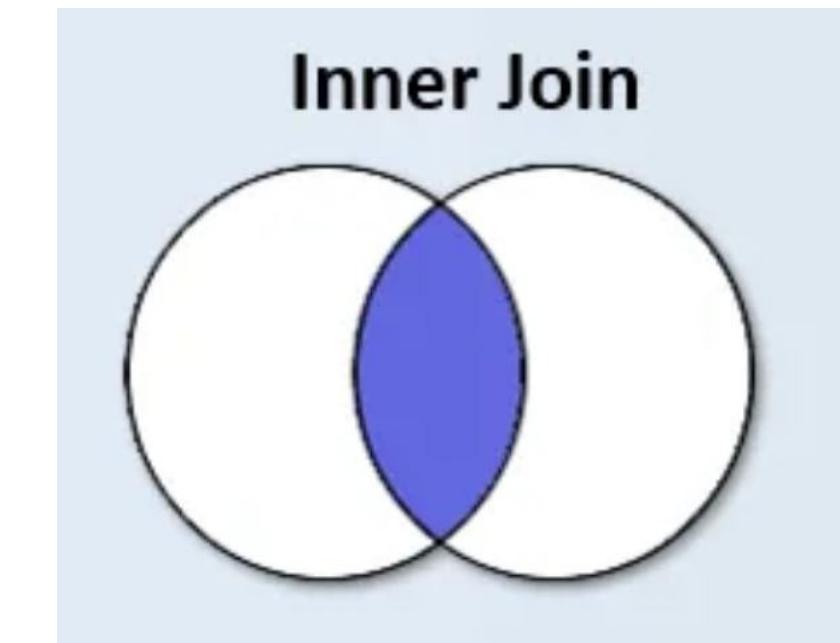


- 일단 여기서 정리하고 들어가면 JOIN은 관계 대수에서 말하는 JOIN과 SQL에서 JOIN은 약간의 차이가 존재
- 여기서 말하는 JOIN은 SQL에서 말하는 것임을 인지하고 가자.

INNER JOIN

두 테이블에서 지정된 조인 조건을 만족하는 행만 반환

```
SELECT e.employee_id, e.first_name, d.department_name  
FROM employees e  
INNER JOIN departments d ON e.department_id = d.department_id;
```



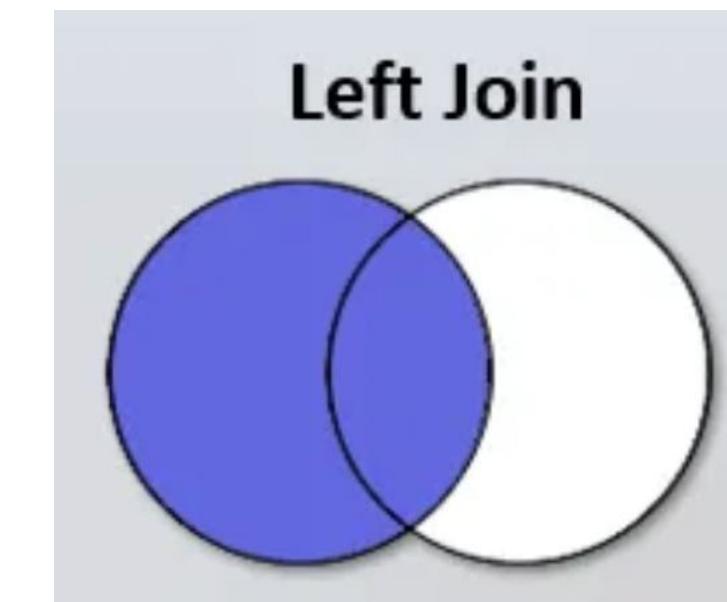
employee_id	first_name	department_name
1	John	Engineering
3	Alice	Engineering
2	Jane	Human Resources
4	Bob	Sales

LEFT JOIN

왼쪽 테이블의 모든 행을 반환하며, 오른쪽 테이블에서 조건이 일치하는 행이 없는 경우 NULL 값을 포함

```
SELECT e.employee_id, e.first_name, d.department_name  
FROM employees e  
LEFT JOIN departments d ON e.department_id = d.department_id;
```

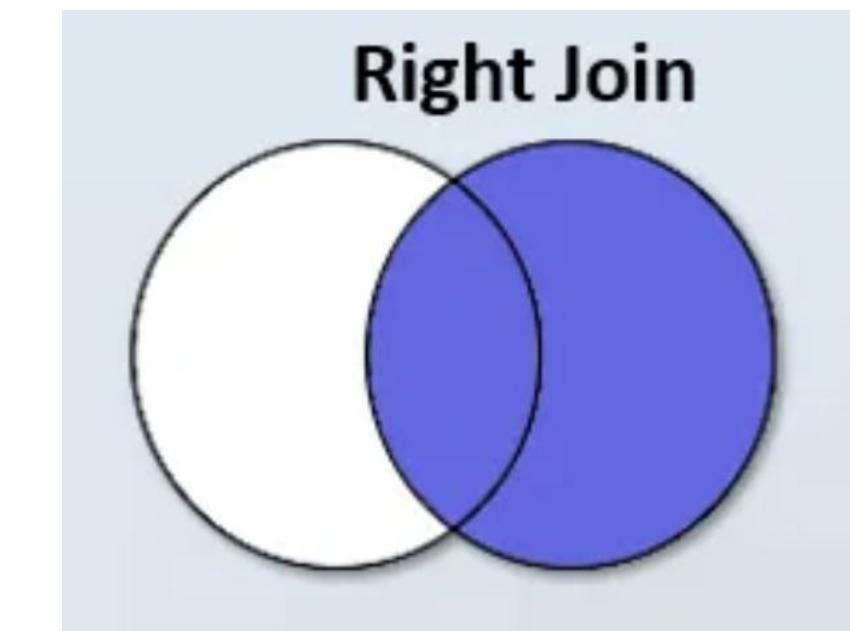
employee_id	first_name	department_name
1	John	Engineering
3	Alice	Engineering
2	Jane	Human Resources
4	Bob	Sales
5	Charlie	NULL



RIGHT JOIN

오른쪽 테이블의 모든 행을 반환하며, 왼쪽 테이블에서 조건이 일치하는 행이 없는 경우 NULL 값을 포함

```
SELECT e.employee_id, e.first_name, d.department_name  
FROM employees e  
RIGHT JOIN departments d ON e.department_id = d.department_id;
```



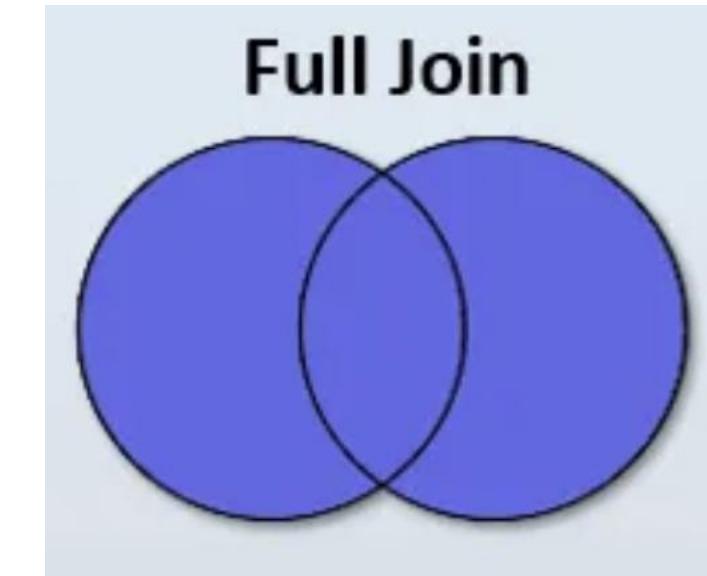
employee_id	first_name	department_name
1	John	Engineering
3	Alice	Engineering
2	Jane	Human Resources
4	Bob	Sales

FULL JOIN

양쪽 테이블의 모든 행을 반환하며, 일치하지 않는 경우 'NULL' 값을 포함

MySQL에서는 FULL JOIN을 직접적으로 지원 x

UNION을 이용해 비슷한 효과를 낼 수 있다.

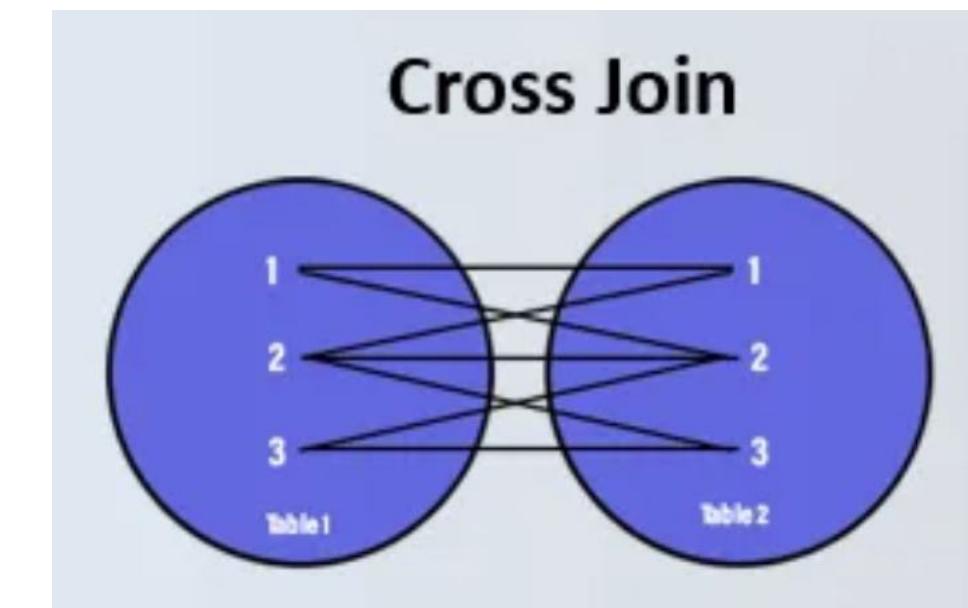


```
SELECT e.employee_id, e.first_name, d.department_name  
FROM employees e  
LEFT JOIN departments d ON e.department_id = d.department_id  
UNION  
SELECT e.employee_id, e.first_name, d.department_name  
FROM employees e  
RIGHT JOIN departments d ON e.department_id = d.department_id;
```

CROSS JOIN

두 테이블의 모든 행의 조합을 반환하여, 결과가 카티션 곱(Cartesian Product)이 된다.

```
SELECT e.first_name, d.department_name  
FROM employees e  
CROSS JOIN departments d;
```

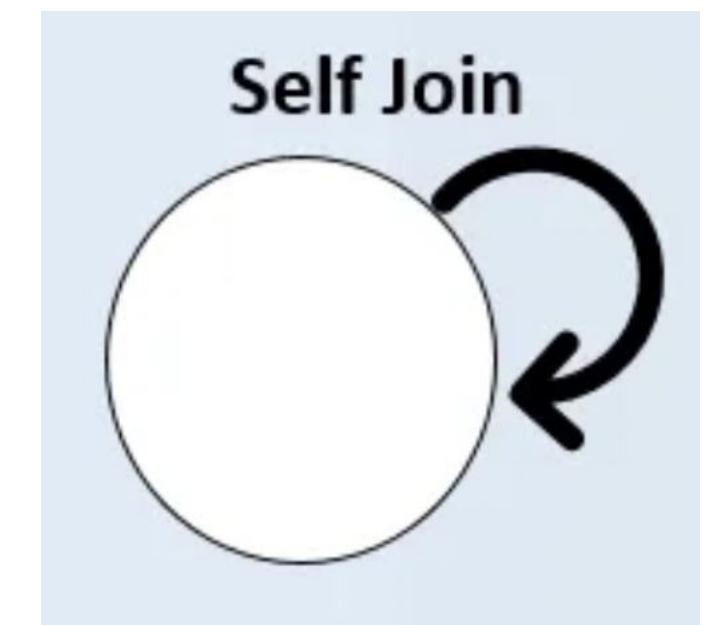


SELF JOIN

같은 테이블을 기준으로 자기 자신과 조인하는 방식

테이블이 스스로 조인되므로 테이블을 두 번 사용. 그렇기에 보통 별칭으로 구별

따로 MySQL에서는 지원하는 함수가 없음



```
SELECT e1.employee_id, e1.first_name, e2.first_name AS manager_name  
FROM employees e1  
LEFT JOIN employees e2 ON e1.employee_id = e2.employee_id;
```

MySQL에서의 실행 계획

발표자: 김광일

MySQL에서의 실행 계획

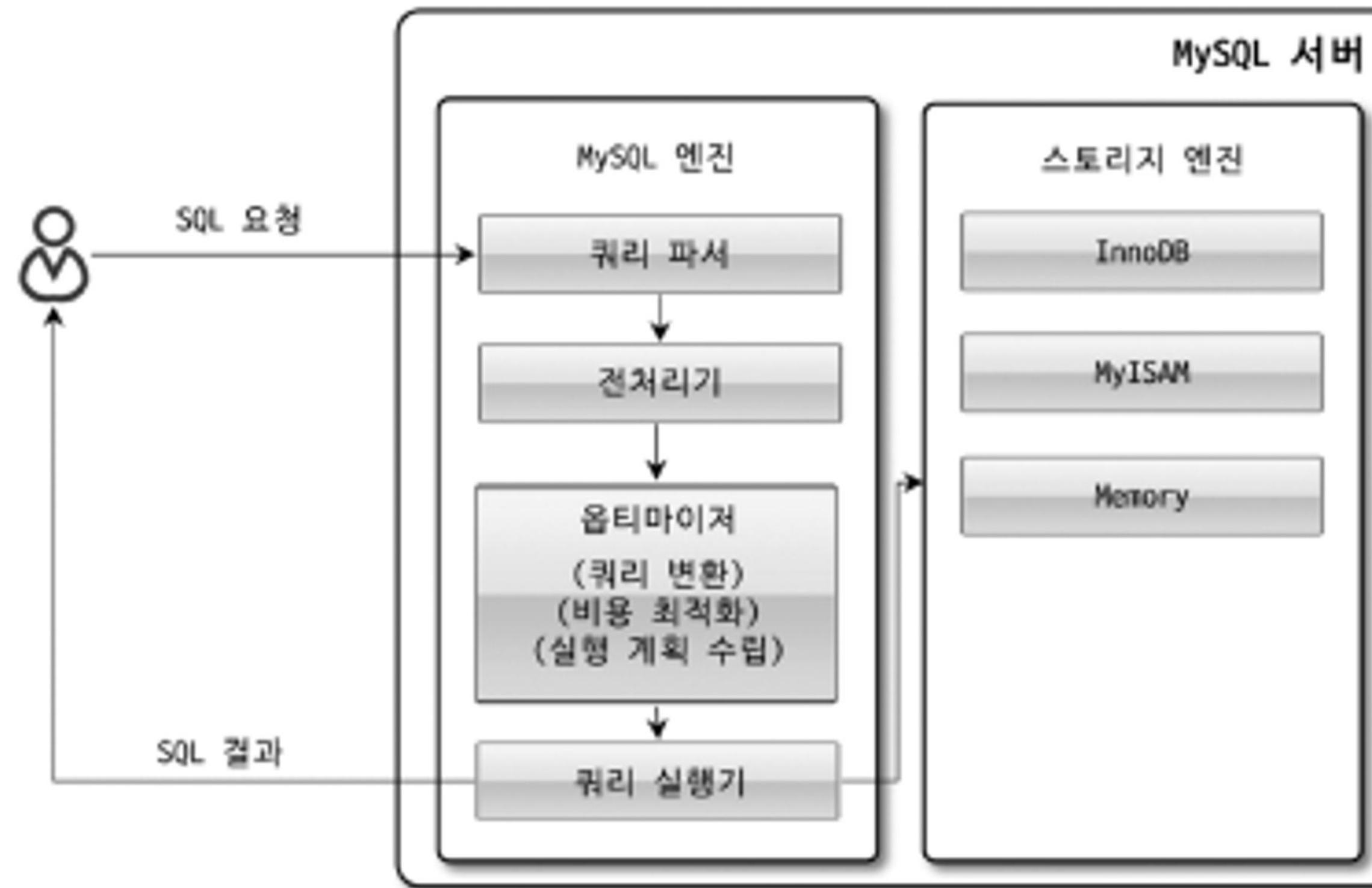


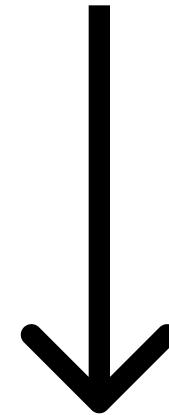
그림 4.6 쿼리 실행 구조

발표자: 김광일

MySQL에서의 실행 계획

Explain 키워드 사용!

```
EXPLAIN SELECT e.employee_id, e.first_name, d.department_name  
FROM employees e  
JOIN departments d ON e.department_id = d.department_id;
```



EXPLAIN SELECT e.employee_id, e.first_name, d.department_name																	
Enter a SQL expression to filter results (use Ctrl+Space)																	
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra						
1	SIMPLE	e	[NULL]	ALL	[NULL]	[NULL]	[NULL]	[NULL]	5	100	Using where						
2	SIMPLE	d	[NULL]	eq_ref	PRIMARY	PRIMARY	4	test2.e.department_id	1	100	[NULL]						

발표자: 김광일

MySQL에서의 실행 계획

컬럼에 대해 간단히 설명하면...

- **table:** 참조하는 테이블의 이름
- **partitions:** 쿼리에서 레코드가 매치되는 파티션
- **type:** 데이터를 어떻게 찾을지에 대한 정보
- **possible_keys:** 데이터를 조회할 때 사용할 수 있는 인덱스 리스트.
- **key:** 실제 사용하기로한 인덱스
- **key_len:** 실제 사용하기로한 인덱스의 길이
- **ref:** 테이블 조인시 어떤 조건으로 해당 테이블에 액세스 되었는지 알려주는 정보
- **rows:** MySQL이 쿼리를 실행하기 위해 조사해야 하는 행의 수
- **filtered:** 테이블 조건에 의해 필터링된 테이블 행의 추정 백분율
- **Extra:** SQL 문을 어떻게 수정할 것인지에 대한 추가 정보

EXPLAIN SELECT e.employee_id, e.first_name, d.department_name													Enter a SQL expression to filter results (use Ctrl+Space)			
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra					
1	SIMPLE	e	[NULL]	ALL	[NULL]	[NULL]	[NULL]	[NULL]	5	100	Using where					
2	SIMPLE	d	[NULL]	eq_ref	PRIMARY	PRIMARY	4	test2.e.department_id	1	100	[NULL]					

발표자: 김광일

MySQL에서의 실행 계획

<https://dev.mysql.com/doc/refman/8.4/en/explain-output.html>

<https://blog.naver.com/kamothi/223565720904>

JOIN의 구현 방식(알고리즘)

발표자: 김광일

JOIN의 구현 방식(알고리즘)

JOIN의 구현 방식은 다양한 알고리즘으로 구현

대표적인 알고리즘으로는

- **Nested Loop Join**
- **Sort-Merge Join**
- **Hash Join**

주의

현재 진행하고 있는 설명은 모두 MySQL을 기준으로 설명 중이다.

Nested Loop Join

루프의 첫 번째 테이블에서 행을 하나씩 읽고, 각 행을 조인의 다음 테이블을 처리하는 중첩 루프로 전달

이 프로세스는 조인할 테이블이 남아 있는 횟수만큼 반복

```
for each row in employees as e {  
    for each row in departments as d {  
        if e.department_id == d.department_id {  
            send (e.employee_id, e.first_name, d.department_name) to client  
        }  
    }  
}
```

일반적으로 index가 없다면 그리고 탐색 대상이 많다면 성능이 매우 나빠질 수 밖에 없다.

시간 복잡도: $O(N*M)$ → 외부 테이블의 행의 수 * 내부 테이블의 행의 수

발표자: 김광일

Nested Loop Join

departments

department_id	department_name
1	Engineering
2	Human Resources
3	Sales

employees

employee_id	first_name	last_name	department_id
1	John	Doe	1
2	Jane	Smith	2
3	Alice	Johnson	1
4	Bob	Brown	3
5	Charlie	Davis	NULL

```
SELECT e.employee_id, e.first_name, d.department_name  
FROM employees e  
JOIN departments d ON e.department_id = d.department_id;
```

발표자: 김광일

Nested Loop Join

쿼리 실행 계획

```
SELECT e.employee_id, e.first_name, d.department_name  
FROM employees e  
JOIN departments d ON e.department_id = d.department_id;
```

- Limit: 200 row(s) (cost=1.86 rows=3.75)
 - Nested loop inner join (cost=1.86 rows=3.75)
 - Table scan on d (cost=0.55 rows=3)
 - Index lookup on e using idx_employees_department_id (department_id=d.department_id) (cost=0.354 rows=1.25)

Block_Nested_Loop Join

대부분은 Nested Loop Join인데, 조인의 연결 조건이 되는 칼럼에 모두 인덱스가 있는 경우 사용되는 조인 방식

Nested Loop Join과의 차이점이라고 하면 조인 버퍼가 사용되는지 여부

조인에서 드라이빙 테이블과 드리븐 테이블이 어떤 순서로 조인되느냐이다.

옵티マイ저는 드라이빙 테이블에서 읽은 레코드를 메모리에 캐시한 후 드리븐 테이블과 이 메모리 캐시를 조인하는 형태로 처리

Block_Nested_Loop Join

```
EXPLAIN SELECT e.employee_id, e.first_name,
d.department_name
FROM employees e
JOIN departments d ON e.department_id =
d.department_id;
```

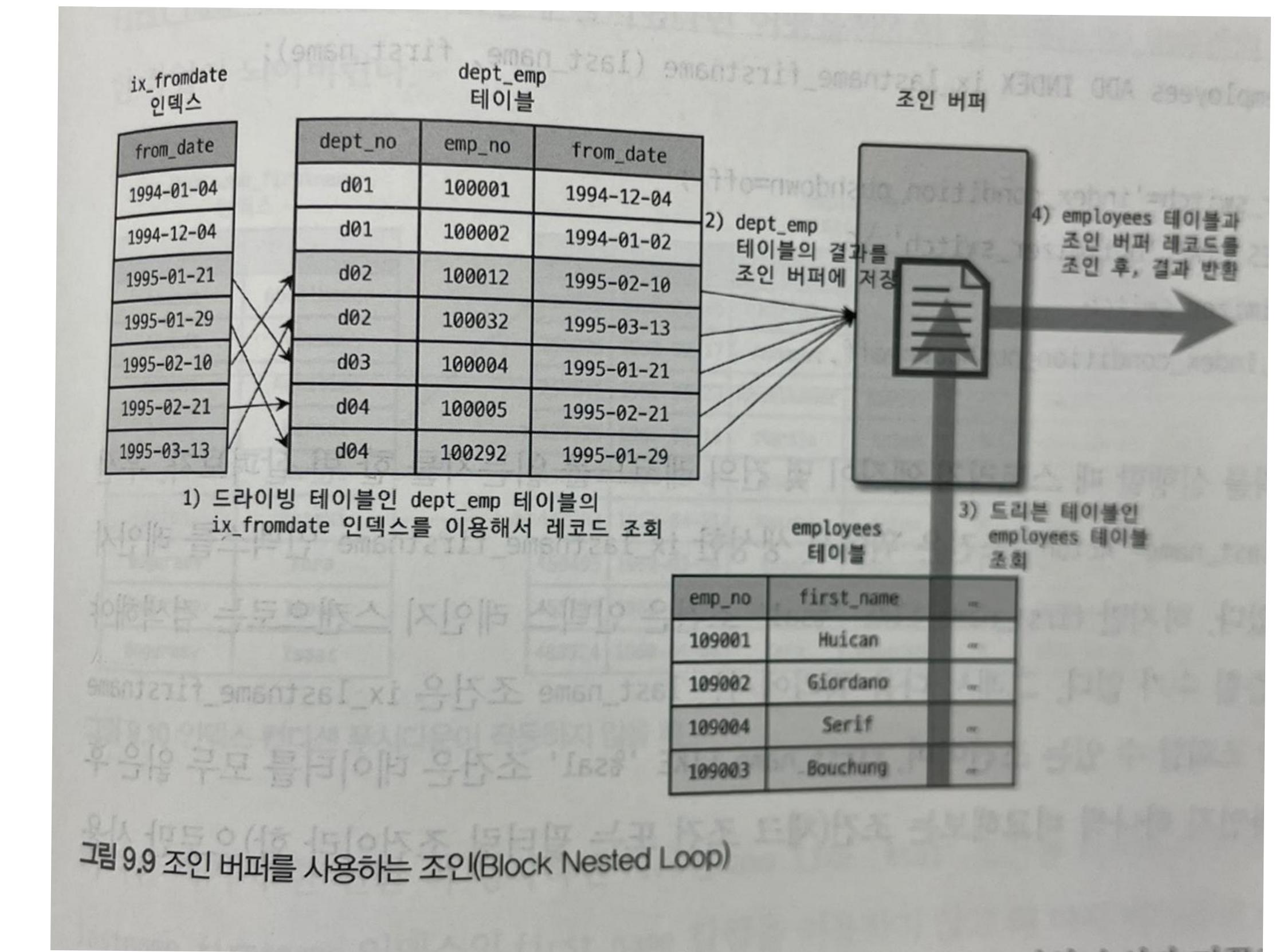


그림 9.9 조인 버퍼를 사용하는 조인(Block Nested Loop)

Block_Nested_Loop Join

중요한 점은 조인 버퍼가 사용되는 쿼리에서는 조인의 순서가 거꾸로인 것처럼 실행

그림에서 `dept_emp` 테이블이 드라이빙 테이블이 되고, `employees` 테이블이 드리븐 테이블

하지만 실제 드라이빙 테이블의 결과는 조인 버퍼에 담아두고, 드리븐 테이블을 먼저 읽고 조인 버퍼에서 일치하는 레코드를 찾는 방식으로 처리

일반적으로 조인이 수행된 후 가져오는 결과는 드라이빙 테이블의 순서에 의해 결정되지만, 조인 버퍼가 사용되는 조인에서는 결과의 정렬 순서가 흐트러질 수 있음을 기억

MySQL 5.x까지 사용되던 알고리즘이다. 현재(MySQL 8.20 버전부터)는 Hash 알고리즘으로 대체되었다. 그렇기에 실행 계획에서 더 이상 `Using Join Buffer(block nested loop)` 메세지는 볼 수 없다.

Sort-Merge Join

소트 머지 조인은 조인 컬럼 기준으로 양쪽 테이블을 정렬한 후, 정렬된 양쪽 테이블을 순차적으로 병합(Merge)하여 Join을 진행한다.

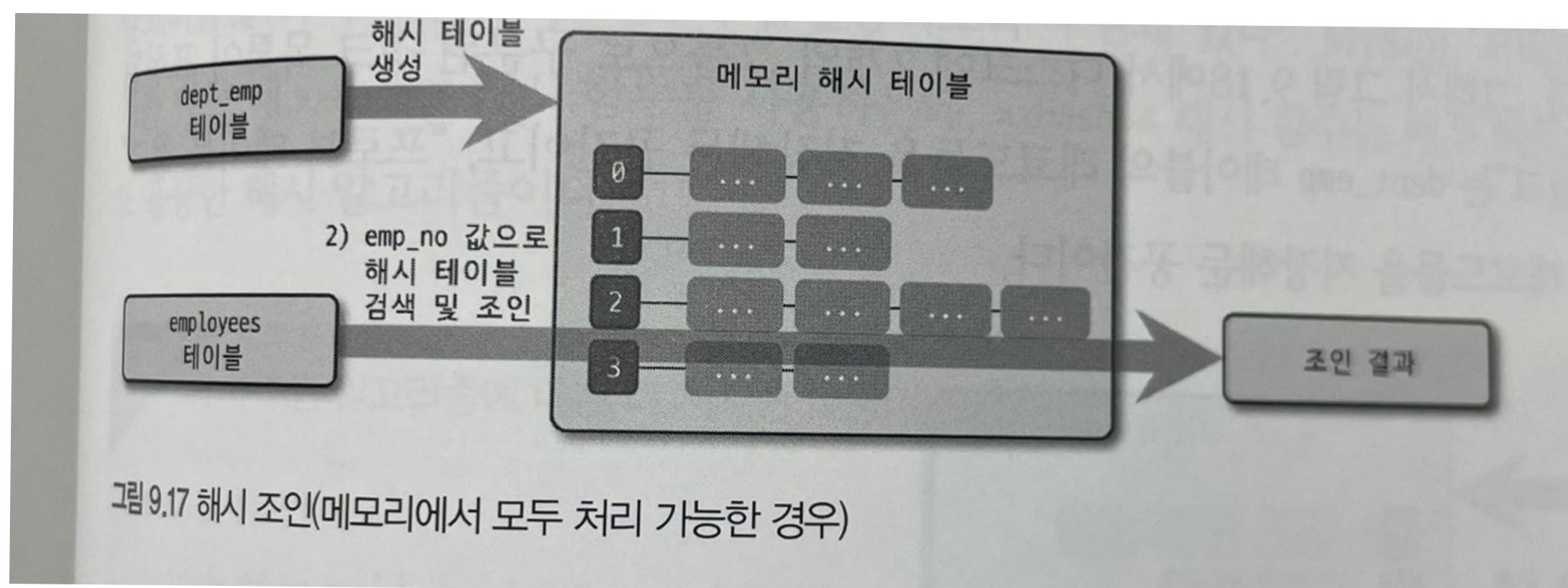
다만, 정렬 연산이 발생하기 때문에, 조인 컬럼에 인덱스가 없을 경우 정렬 비용이 발생하게 된다.

mysql에서 제공하지 않으니 패스

Hash Join

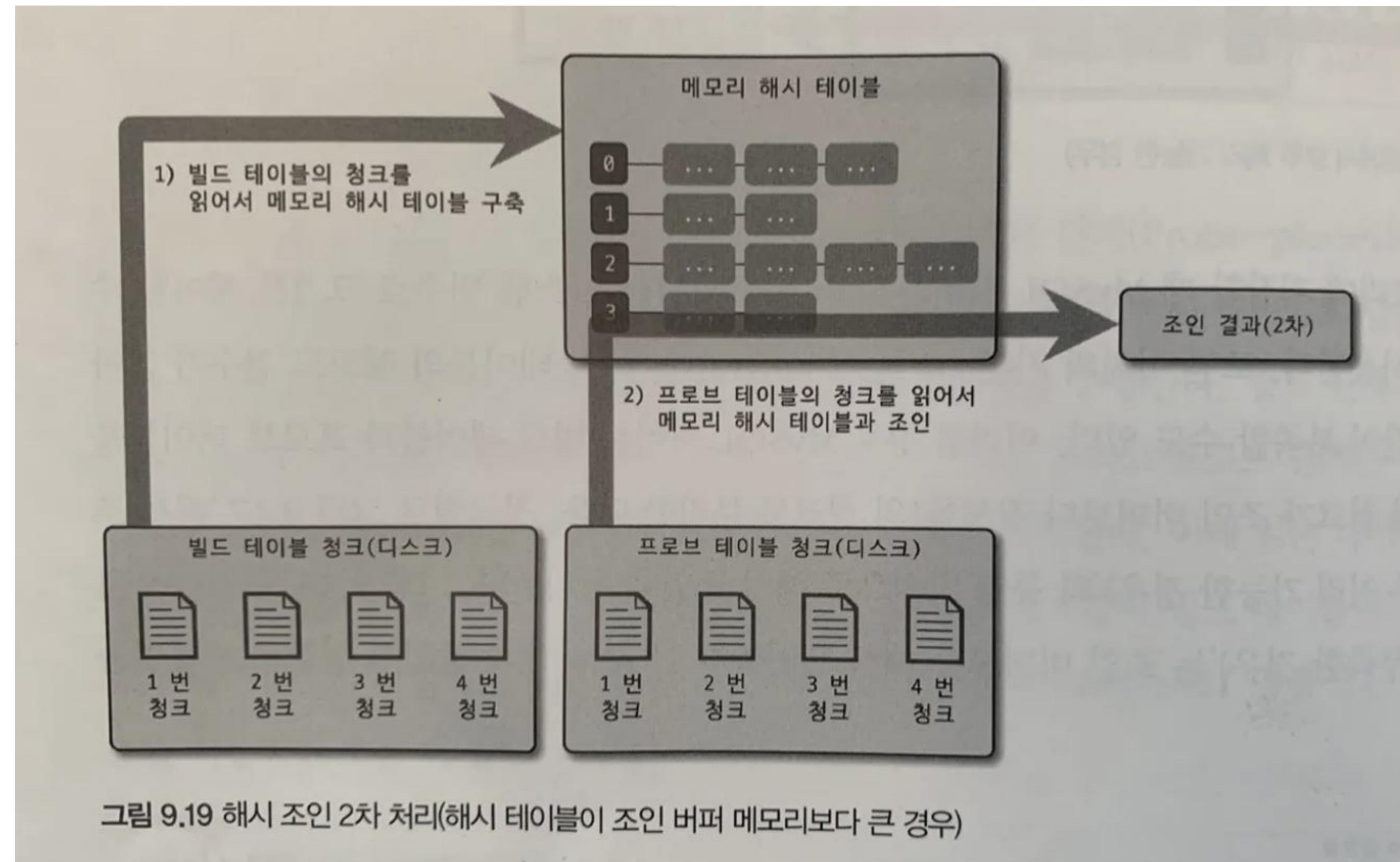
해시 조인은 빌드 단계와 프로브 단계로 나뉘어 처리

빌드 단계에서는 조인 대상 테이블 중에서 레코드 건수가 적어서 해시 테이블로 만들기에 용이한 테이블을 골라서 메모리에 해시 테이블을 생성(빌드)하는 작업을 수행



Hash Join

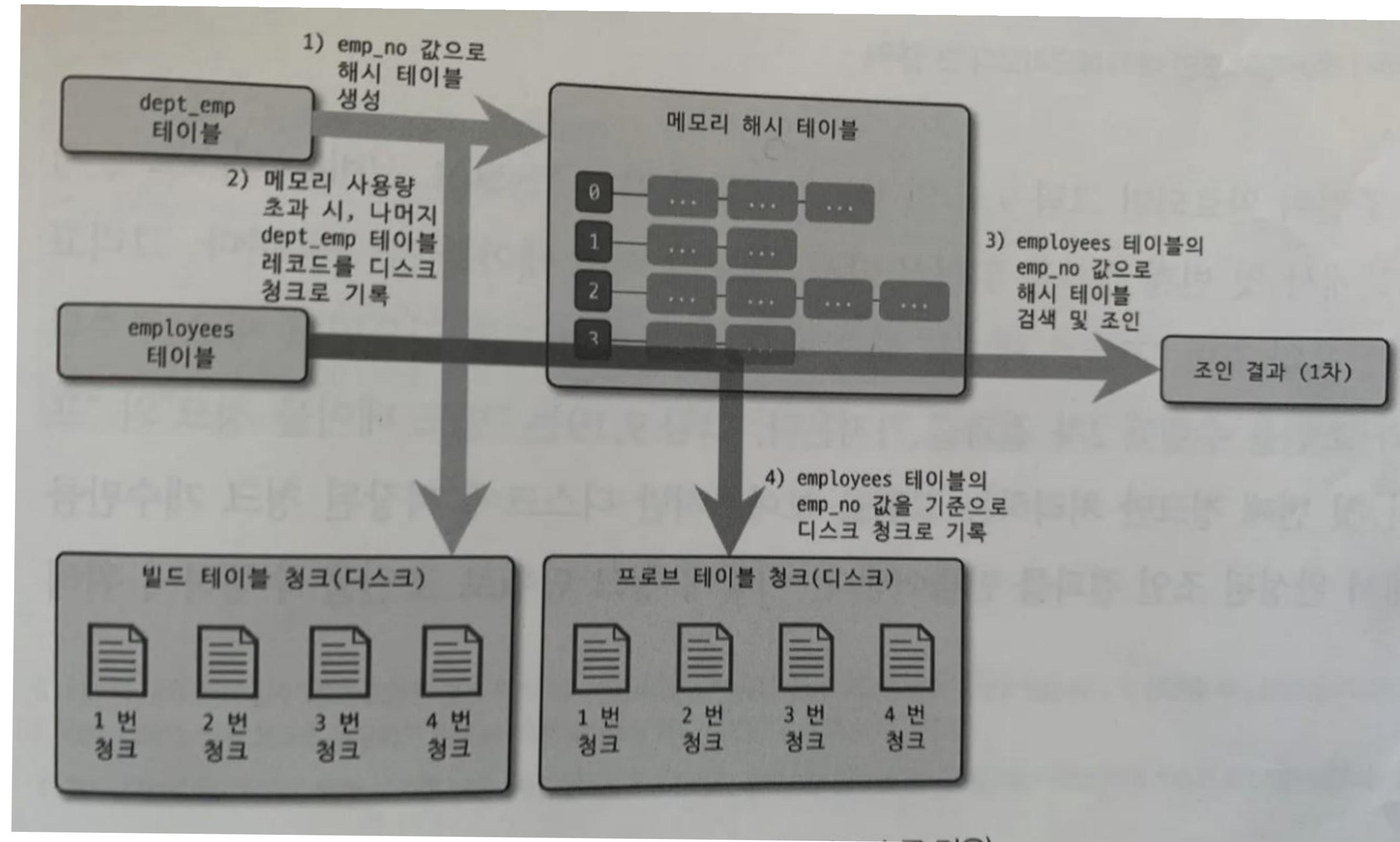
버퍼가 부족하다면?(버퍼의 기본 크기는 256KB)



발표자: 김광일

Hash Join

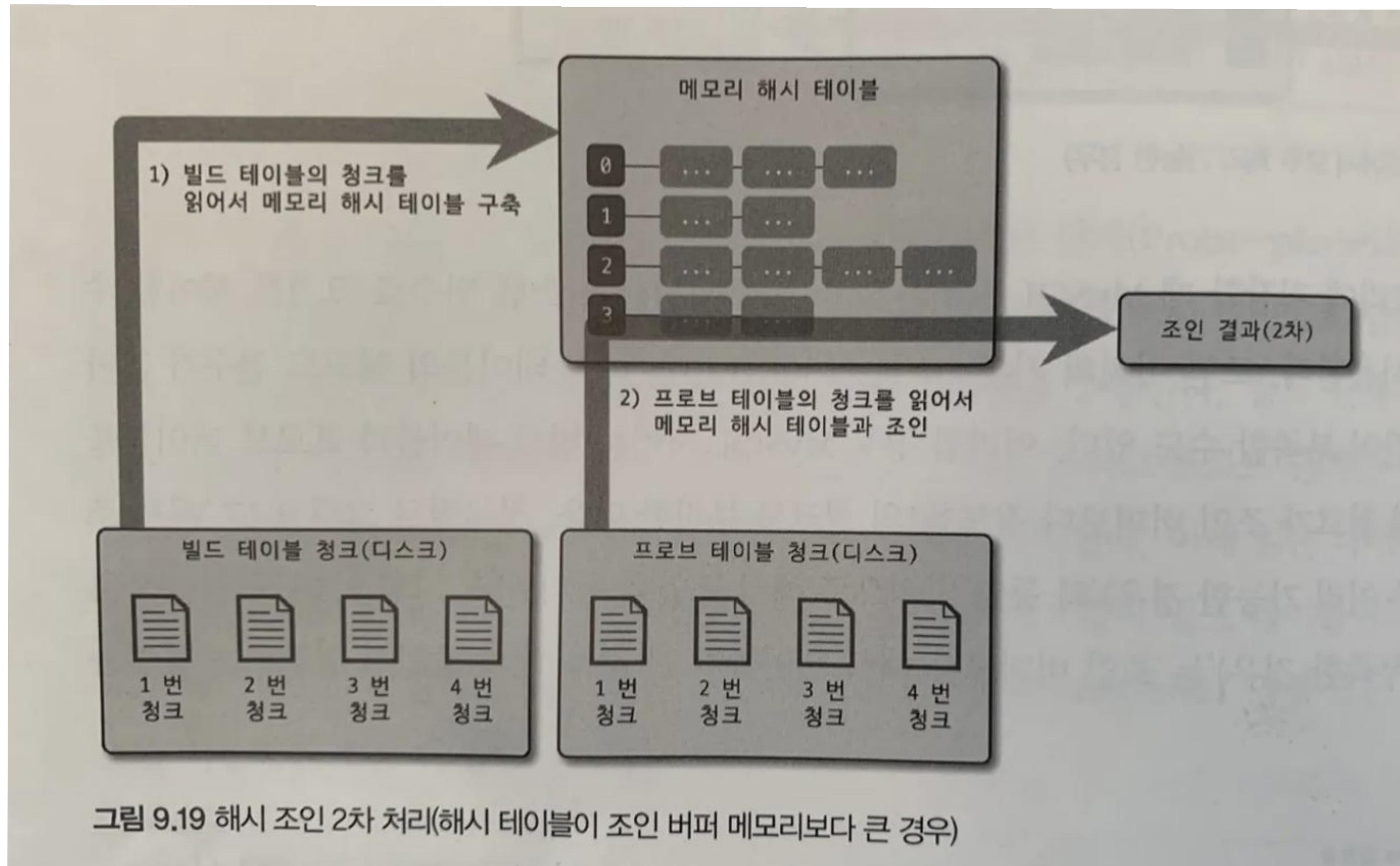
버퍼가 부족하다면?(버퍼의 기본 크기는 256KB)



발표자: 김광일

Hash Join

버퍼가 부족하다면?(버퍼의 기본 크기는 256KB)



발표자: 김광일

Hash Join

쿼리 실행 계획

```
EXPLAIN FORMAT= TREE SELECT e.employee_id, e.first_name, d.department_name  
FROM employees e JOIN departments d ON e.department_id = d.department_id;
```

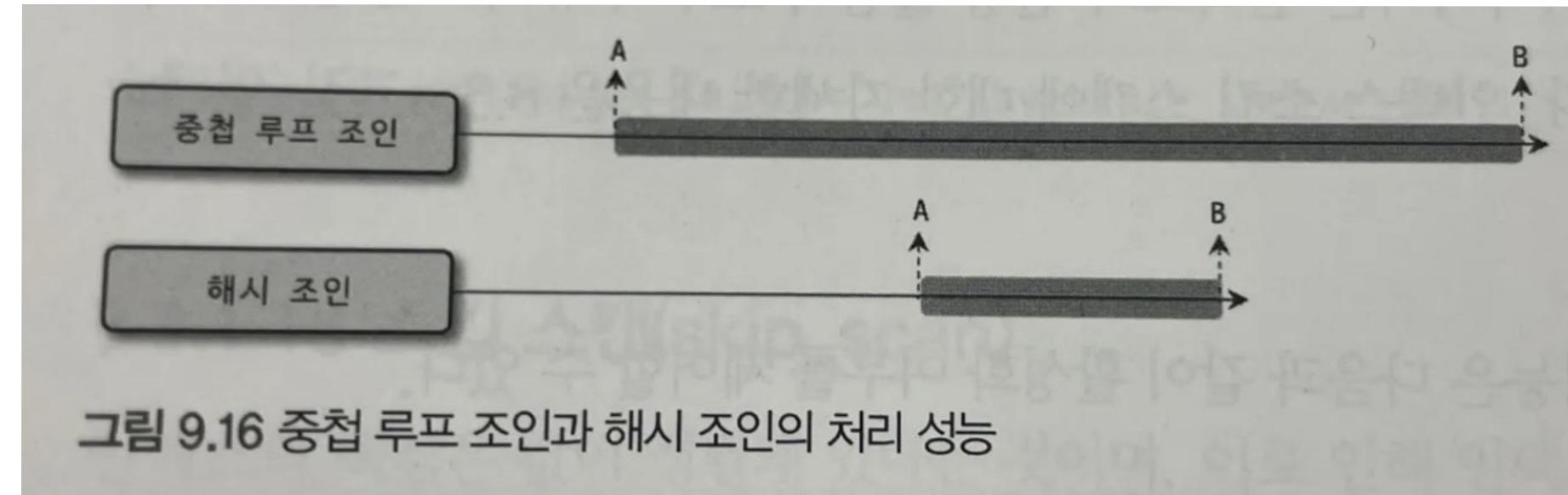
- > Limit: 200 row(s) (cost=2.3 rows=3)
 - Inner hash join (e.department_id = d.department_id) (cost=2.3 rows=3)
 - Table scan on e (cost=0.117 rows=5)
 - Hash
 - Table scan on d (cost=0.55 rows=3)

Hash Join

MySQL 8.0.18 버전부터 해시 조인이 추가로 지원되기 시작

많은 사람들이 해시 조인에 기대하는 것은 성능

그럼 Hash를 쓰면 성능이 무조건 높을까?



Hash Join

MySQL 서버는 주로 조인 조건의 칼럼이 인덱스가 없다거나 조인 대상 테이블 중 일부의 레코드 건수가 매우 적은 경우 등에 대해서만 해시 조인 알고리즘을 사용하도록 설계되어 있음

인덱스가 있다면 빠르게 대응하는 레코드를 찾겠지만 없으면 문제

즉 MySQL 서버의 해시 조인 최적화는 네스티드 루프 조인이 사용되기에 적합하지 않은 경우를 위한 차선책 같은 기능

JOIN 최적화 기법

join 쿼리 실행 계획 최적화를 위한 알고리즘

- **Exhaustive** 검색 알고리즘
- **Greedy** 검색 알고리즘

JOIN되는 테이블의 순서를 지정할 수 없을까?

발표자: 김광일

실행계획 확인 및 이모저모

EXPLAIN SELECT e.employee_id, e.first_name, d.depar															Enter a SQL expression to filter results (use Ctrl+Space)		
비	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra					
1	1	SIMPLE	d	[NULL]	ALL	PRIMARY	[NULL]	[NULL]	[NULL]	3	100	[NULL]					
2	1	SIMPLE	e	[NULL]	ALL	[NULL]	[NULL]	[NULL]	[NULL]	5	20	Using where; Using join buffer (hash join)					

왜 순서가 강제되는걸까?

- 테이블 크기
 - 일반적으로 테이블 크기가 작은 것을 먼저 읽고 join을 수행
 - 조인 과정에서 비교해야 할 데이터 양을 줄일 수 있어 성능을 높일 수 있음
- 조인 알고리즘
- 조인 조건과 인덱스

힌트를 줘보자

결국은 mysql의 옵티마이저가 알아서 결정함. 그러면 내가 원하는데로 강제는?

첫 번째 방식은 **STRAIGHT** 키워드로 순서를 정하는 것

정확히는 이것은 힌트에 해당하는 키워드인데 옵티마이저에게 이거 먼저해줘
또는 이거 해줘라는 힌트를 주면 힌트를 바탕으로 옵티마이저는 수행하게 된다.

```
EXPLAIN SELECT e.employee_id, e.first_name, d.department_name
FROM employees e
STRAIGHT_JOIN departments d ON e.department_id = d.department_id;
```

힌트를 줘보자

EXPLAIN SELECT e.employee_id, e.first_name, d.department_name														Enter a SQL expression to filter results (use Ctrl+Space)			
번호	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra					
1	1	SIMPLE	e	[NULL]	ALL	[NULL]	[NULL]	[NULL]	[NULL]	5	100	Using where					
2	1	SIMPLE	d	[NULL]	eq_ref	PRIMARY	PRIMARY	4	test2.e.department_id	1	100	[NULL]					

여기서 주의할 점은 FROM 절에 사용된 테이블의 순서를 조인 순서에 맞게 변경해야 하는 번거로움이 있다.

STRAIGHT_JOIN은 한번 사용되면 FROM 절에 명시된 모든 테이블의 조인 순서가 결정되기 때문

일부는 조인 순서를 강제하고 나머지는 옵티마이저에게 순서를 결정하게 맞기는 것이 불가능했다.

그래서..

- JOIN_FIXED_ORDER
- JOIN_ORDER
- JOIN_PREFIX
- JOIN_SUFFIX

JOIN 성능과 인덱스의 중요성

발표자: 김광일

JOIN 성능과 인덱스의 중요성

JOIN은 특히 조건을 만족하는 데이터를 빠르게 찾기 위해 인덱스의 영향을 크게 받음

JOIN 조건이 걸린 컬럼에 인덱스가 없을 경우, 데이터베이스는 모든 행을 스캔해야 하므로 성능이 크게 저하

반면, JOIN 조건이 걸린 컬럼에 인덱스가 있으면 필요한 데이터만 빠르게 조회할 수 있어 JOIN 성능이 크게 향상

위에서 우리는 내부에서 사용되는 JOIN 알고리즘과 실제 JOIN 과정에서의 정책을 보았다.

인덱스의 유무, 레코드의 사이즈 등에 따라서 옵티마이저는 그에 맞는 것을 선택하게 된다.

3중 JOIN?

발표자: 김광일

B-Tree, B+Tree

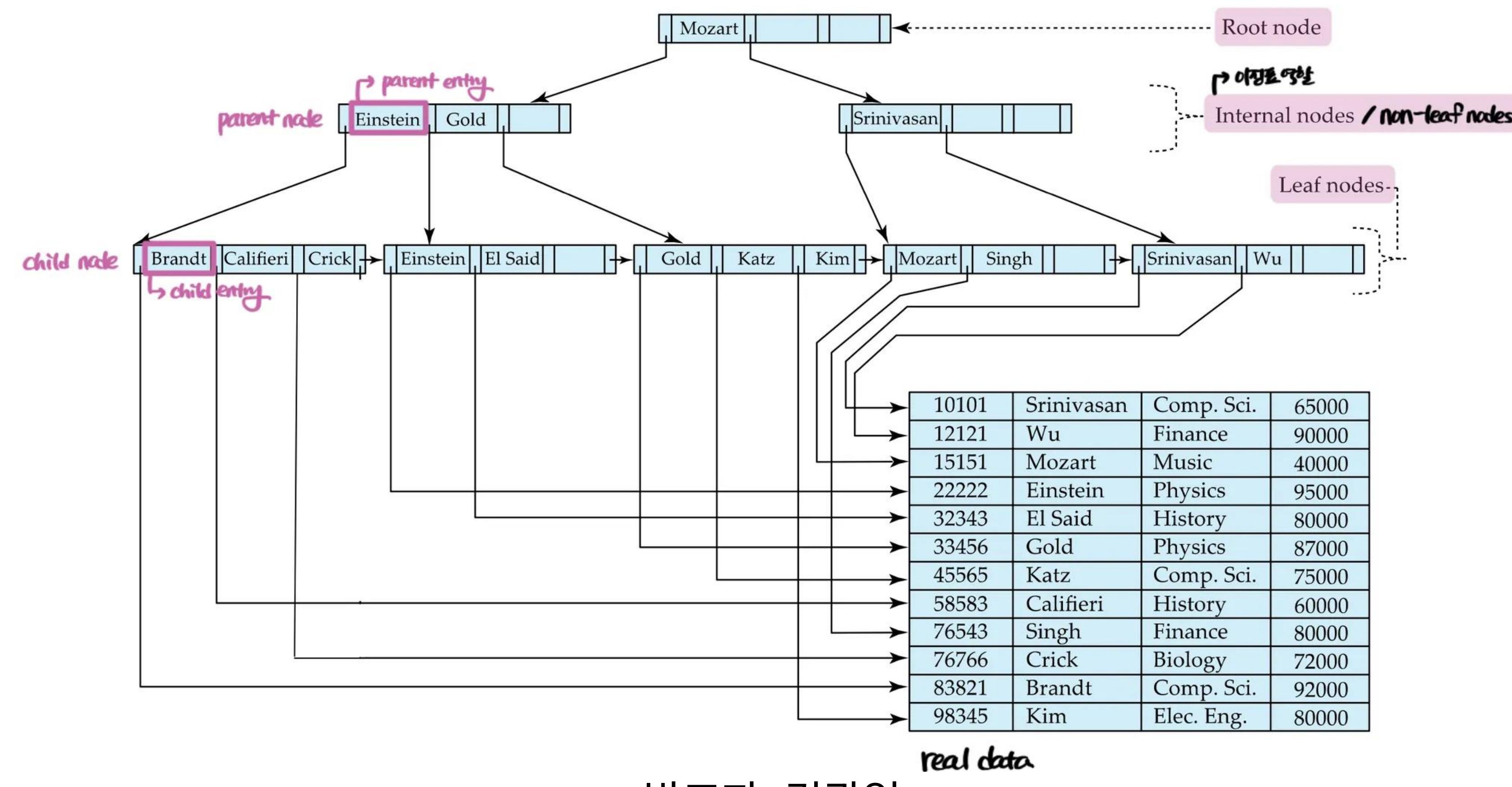
발표자: 김광일

B+Tree

- 인덱스
- 트리의 루트에서 단말 노드까지 모든 경로의 길이가 같은 균형 트리(Balanced tree) 형태
- 트리에서 루트 노드를 제외하고 비단말 노드는 $[n/2]$ 와 n 사이의 자식을 가진다.
 - 즉, 50% 이상이 차있어야 하는 것
- n 은 특정한 트리에 고정된 값
- 루트는 2개에서 n 개의 자식을 가진다.
- leaf 노드는 $\lceil(n-1)/2\rceil \sim n-1$ 사이의 개수만큼 value를 가짐

B+Tree의 구조

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	-----	-----------	-----------	-------



발표자: 김광일

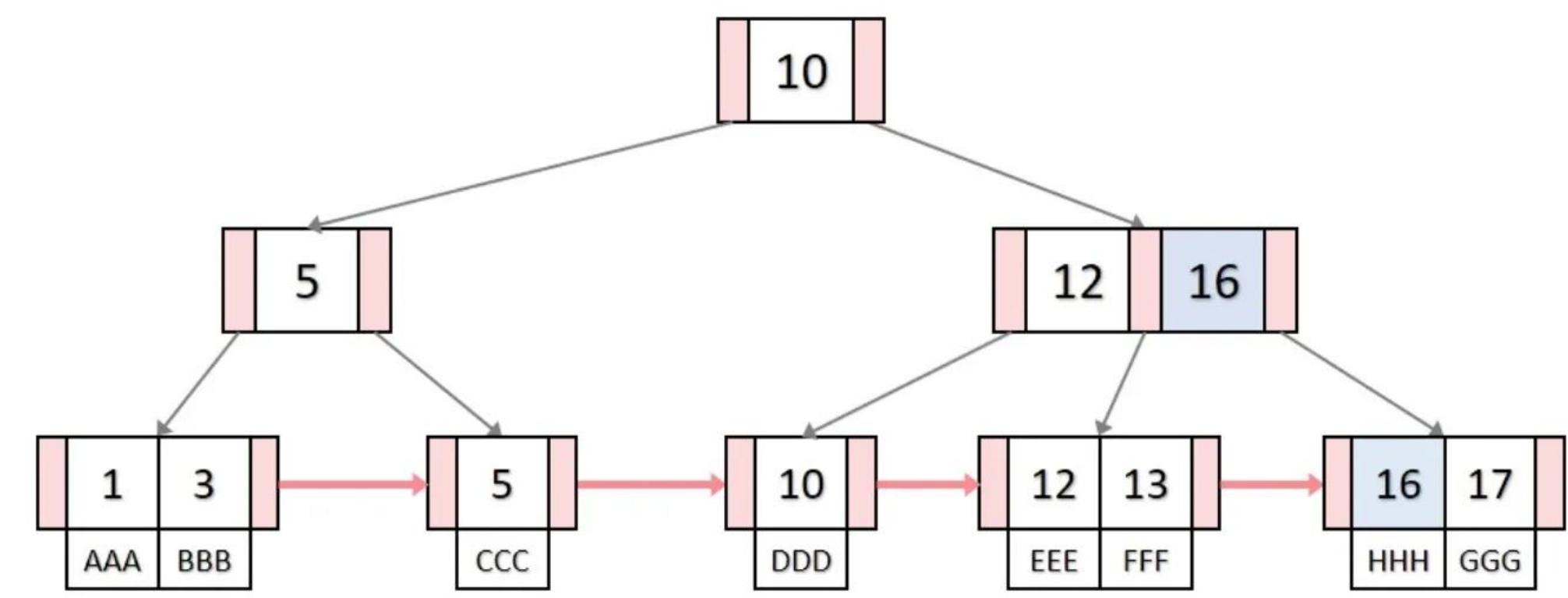
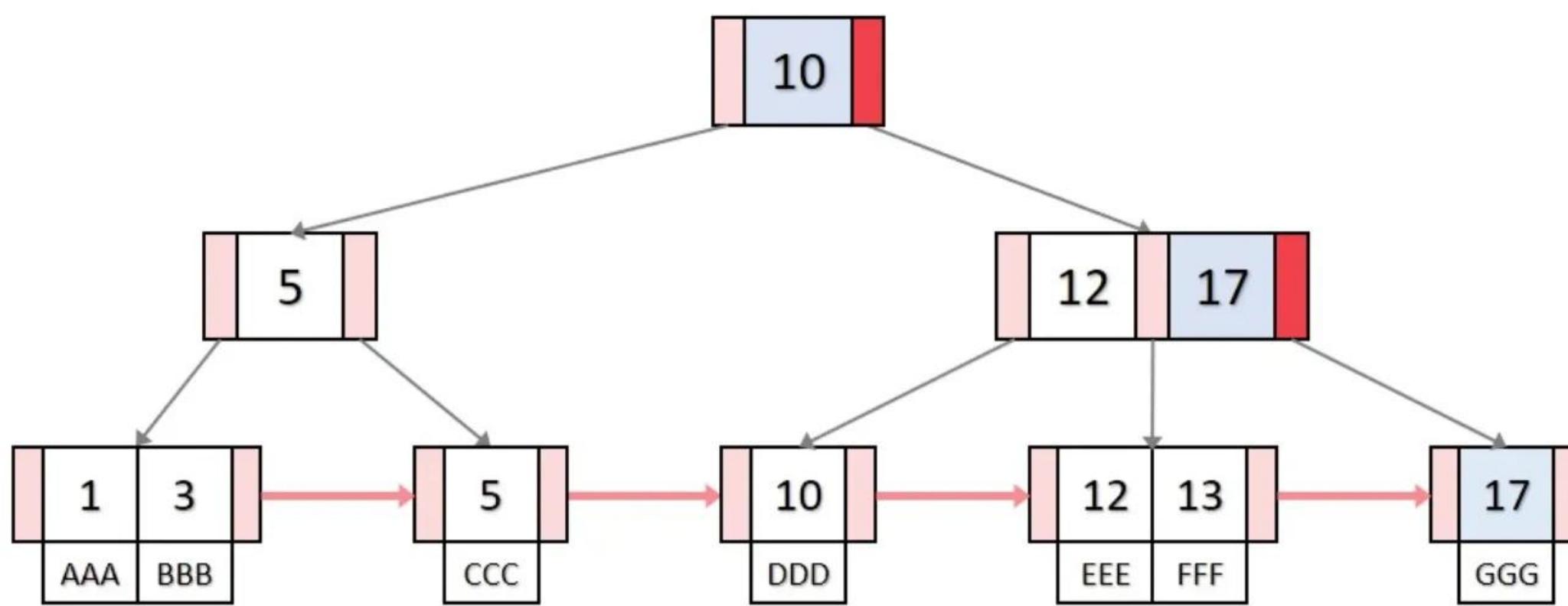
B+Tree의 질의

1. 특정 값을 원하는 point query → **find(v)**
2. 범위 값을 원하는 range query → **find(lb,ub)**

시간 복잡도는? 좋을까? B-Tree랑 비교하면?

B+Tree 삽입 및 삭제

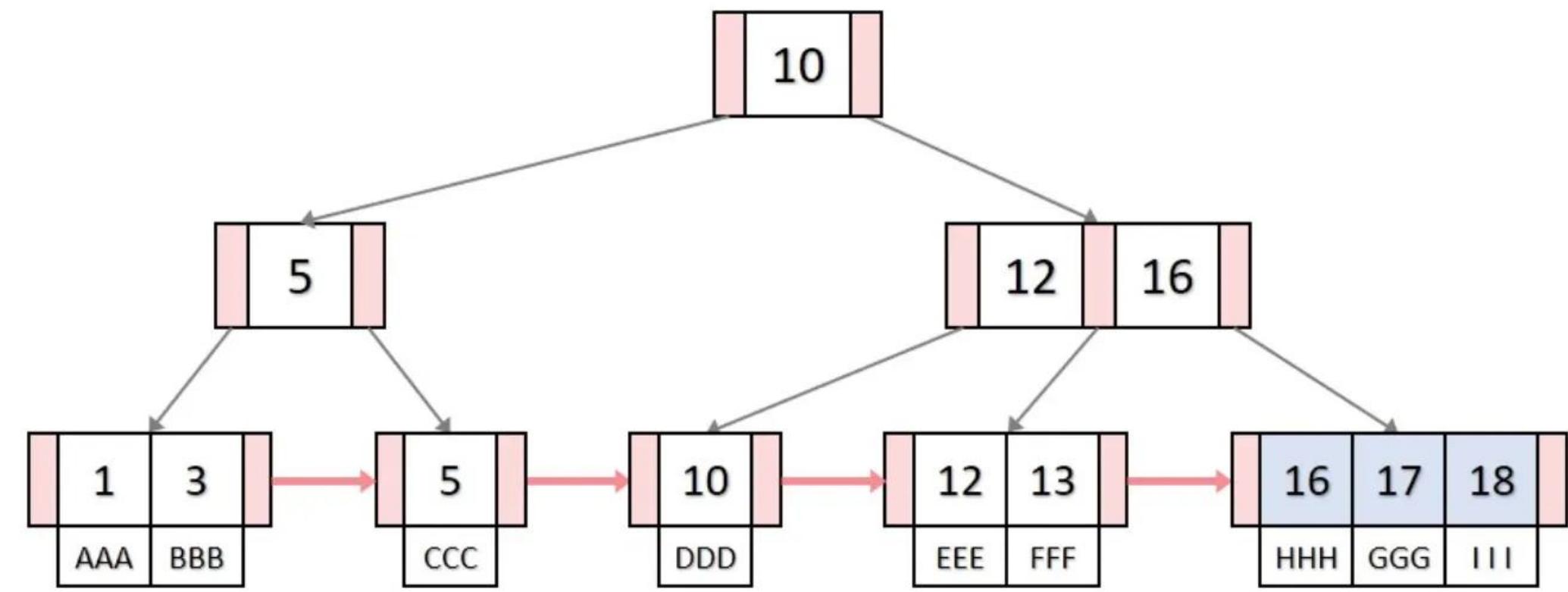
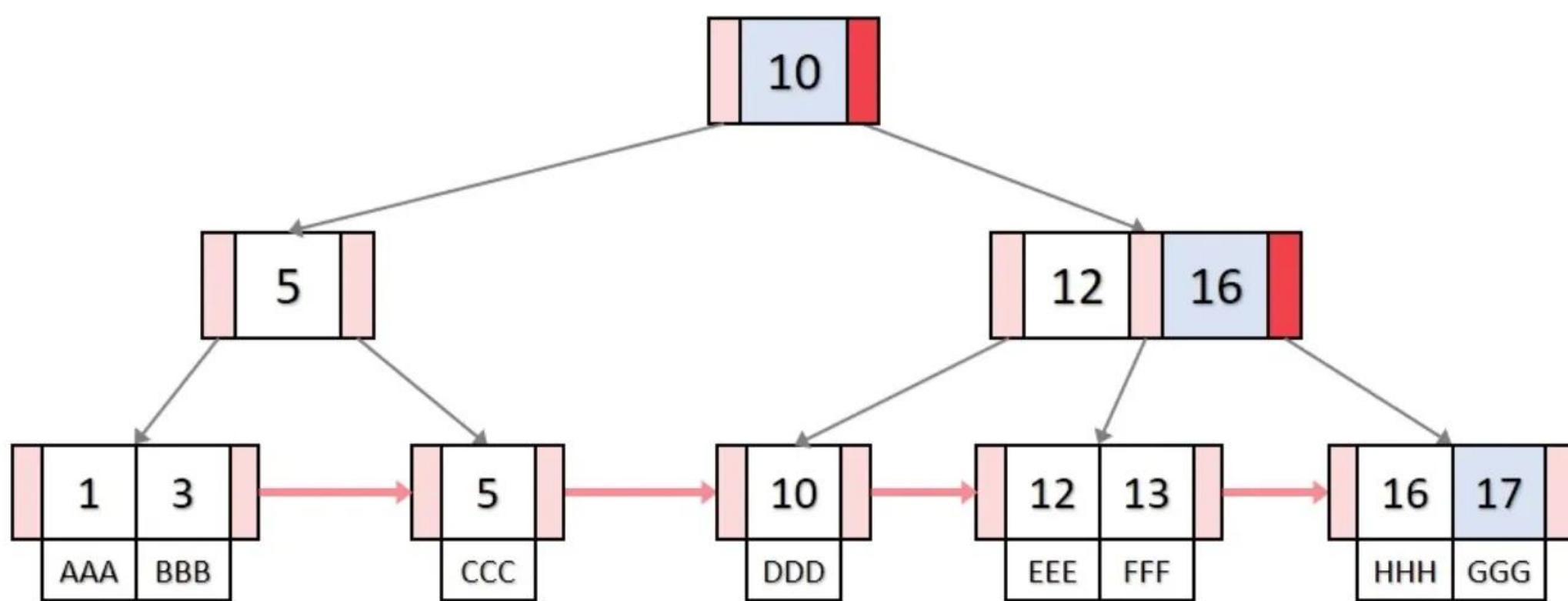
16을 삽입



발표자: 김광일

B+Tree 삽입 및 삭제

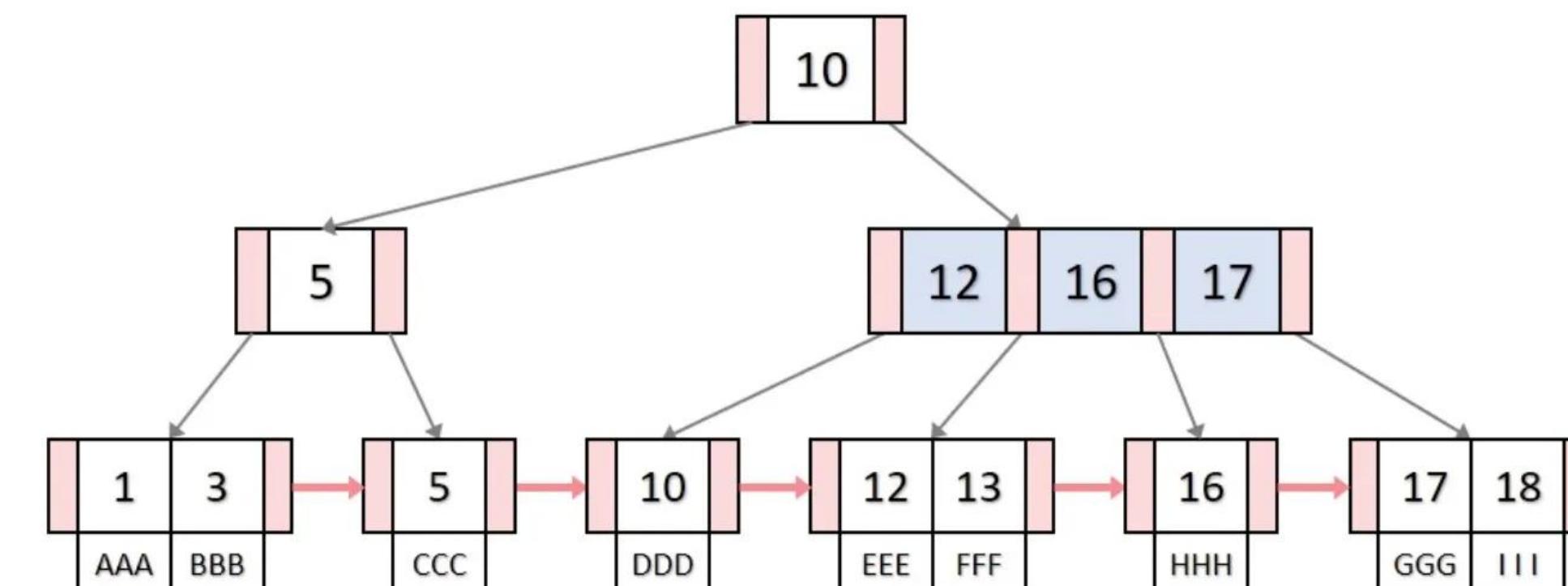
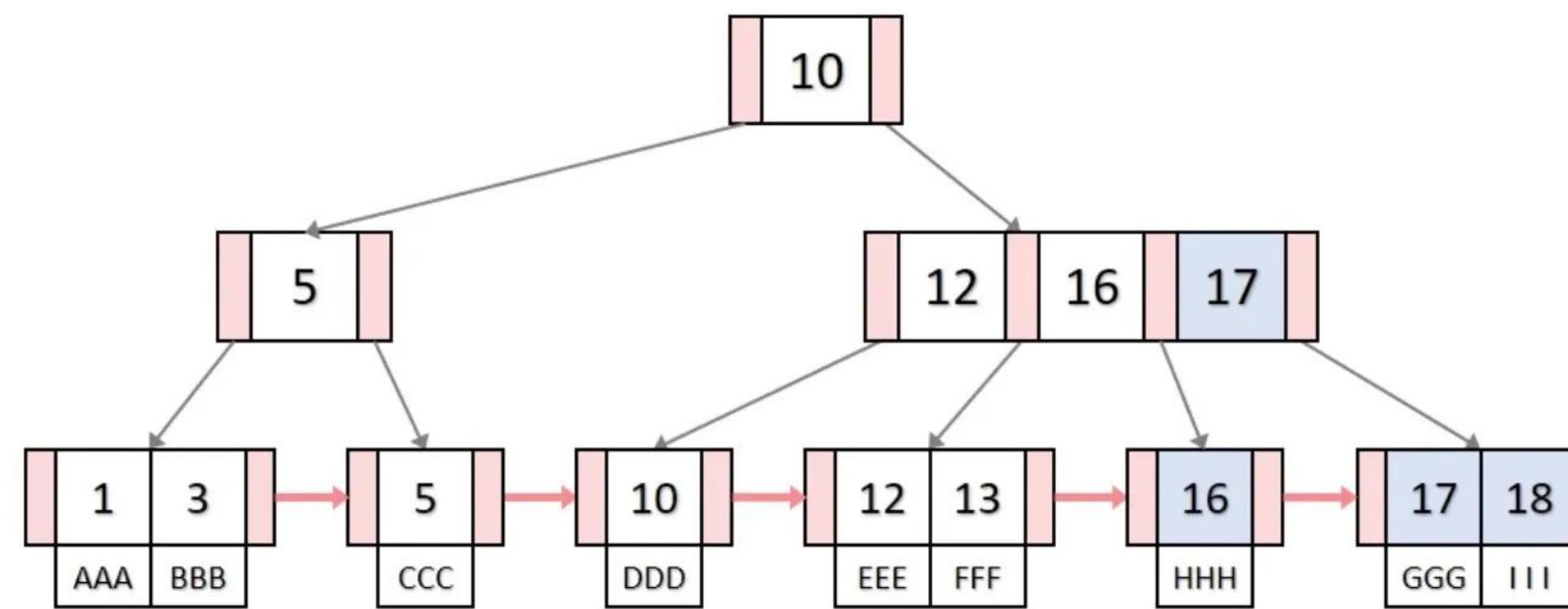
18을 삽입



발표자: 김광일

B+Tree 삽입 및 삭제

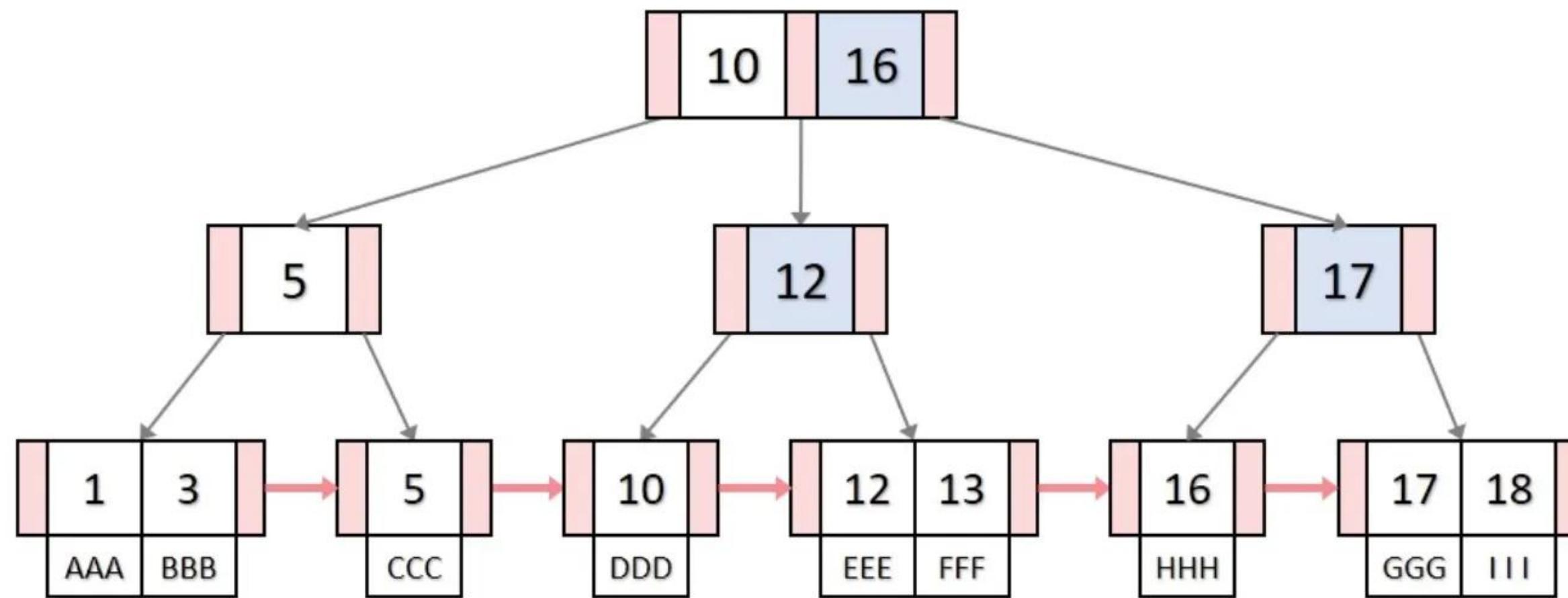
18을 삽입



발표자: 김광일

B+Tree 삽입 및 삭제

18을 삽입



발표자: 김광일

B-Tree

차이점은 B-Tree는 검색 키 값이 차지한 중복된 저장 공간을 제거한다는 것

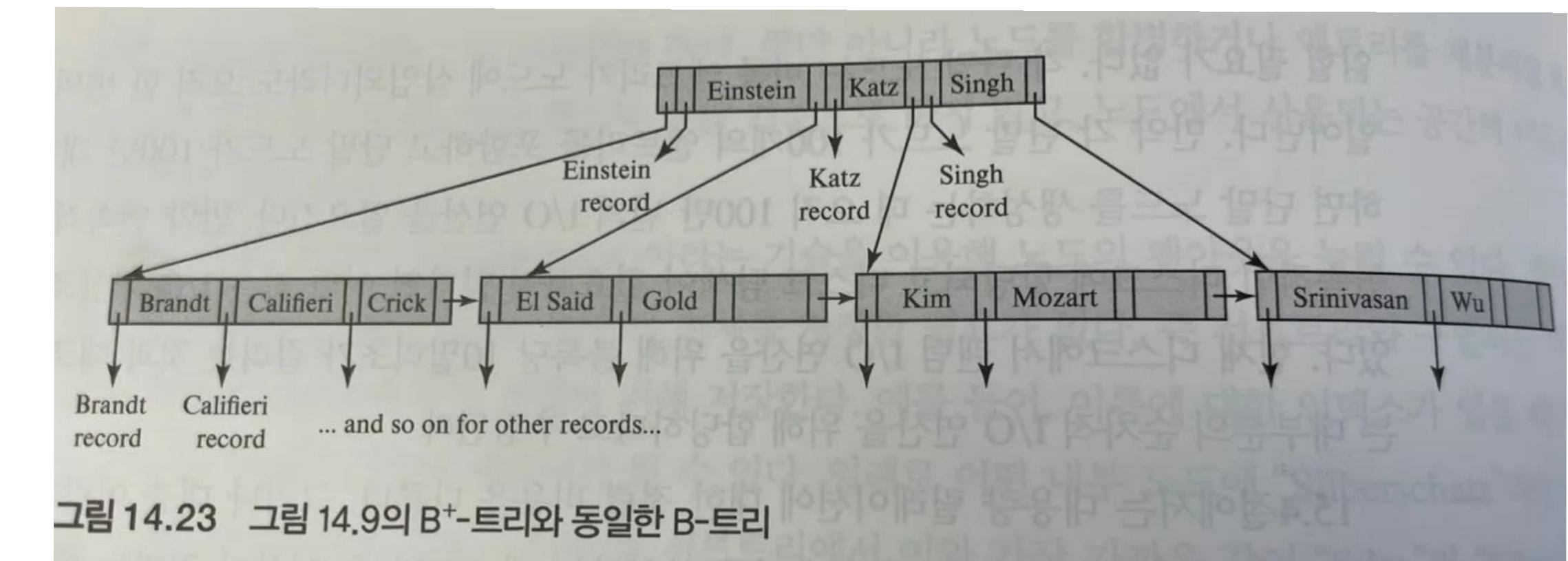
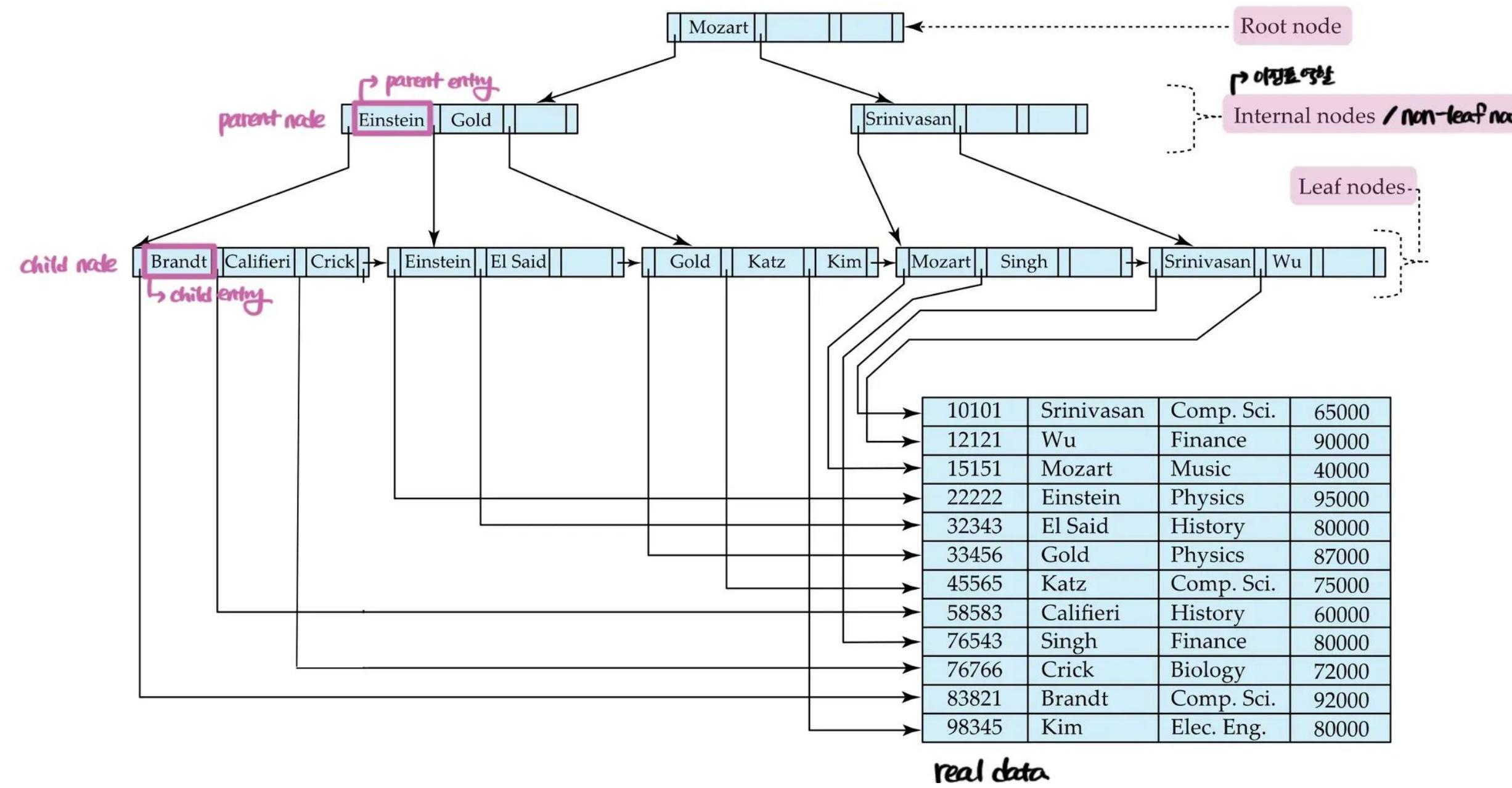


그림 14.23 그림 14.9의 B⁺-트리와 동일한 B-트리

검색키가 반복되지 않기 때문에 B+Tree에 비해 더 적은 트리 노드로 인덱스를 저장할 수 있다.

**그렇다면, B+Tree가 B-Tree에 비해 반드시 좋다고 할 수 있을까요?
그렇지 않다면 어떤 단점이 있을까요?**

**오름차순으로 정렬된 인덱스가 있다고 할 때, 내림차순 정렬을 시도할 경우 성능이 어떻게 될까요?
B-Tree/B+Tree의 구조를 기반으로 설명해 주세요.**

RBT 쓰면 안되는거야?

발표자: 김광일