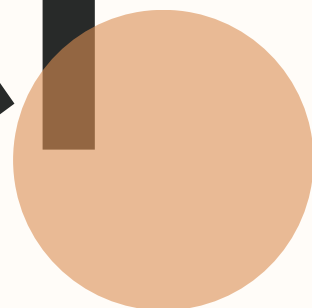


인덱스의 이해와 활용

MySQL을 중심으로

우리 FISA 2기 김경은

목차



01. 인덱스란

02. 인덱스의 종류


03. 인덱스의 동작 원리

04. 인덱스의 효율적인 설계와 활용

01. 인덱스란

01. 인덱스란

- 데이터베이스에서 인덱스란 “색인” 또는 “찾아보기”로 비유된다.
- 데이터베이스에서 원하는 데이터를 찾으려고 할때 (SELECT문), 인덱스를 사용하면 데이터의 **조회** 속도를 높일 수 있다.
- 그러나 **삽입, 수정, 삭제** 성능은 저하될 수 있으므로 적절한 사용이 필요하다.
- 인덱스는 **열(Column)**을 기준으로 생성한다.



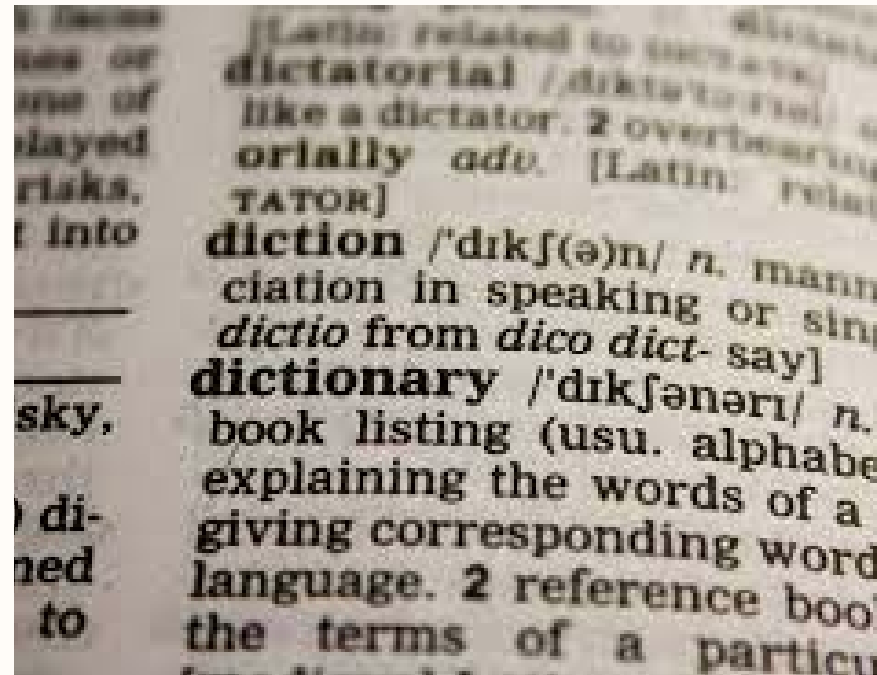
```
-- 테이블에 생성된 인덱스 확인  
SHOW INDEX FROM 테이블명;
```

```
-- 새 인덱스 생성  
CREATE INDEX 인덱스명 ON 테이블명(컬럼명);
```

```
-- 인덱스 삭제  
ALTER TABLE 테이블명 DROP INDEX 인덱스명;
```

02. 인덱스의 종류

02. 인덱스의 종류



클러스터링 인덱스

PRIMARY KEY 설정된 열에 적용

테이블 당 하나만 존재

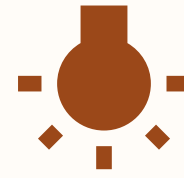
클러스터링 인덱스 기준으로 데이터 자체가
정렬됨



보조 인덱스

PK가 아닌 열에 적용

그 외 다른 열에 추가로 적용 가능
데이터 자체가 정렬되는 것이 아니
라 정렬된 색인(찾아보기)이 있음



02. 인덱스의 종류

B-TREE 인덱스

MySQL에서 일반적인 경우 B-Tree구조의 인덱스를 가지게 된다.

등호나 부등호, BETWEEN이나 LIKE를 사용한 쿼리에 적합하다.



MyISAM, InnoDB(B+Tree)

R-TREE 인덱스

MySQL 8.0부터 공간 데이터를 위한 공간 인덱스를 지원한다.

R-tree는 B-tree 기반으로 동작하며 도형의 포함관계를 바탕으로 트리구조가 생성된다.



MyISAM, InnoDB
(다른 storage engine은 b-tree로 동작)

FULL TEXT 인덱스

FULL TEXT 인덱스는 전문 검색을 위한 인덱스로 단어 단위로 인덱싱된다.

B-Tree 기반으로 동작하고 텍스트 데이터를 효율적으로 처리하기 위한 특화된 기능을 제공한다.



MyISAM, InnoDB(5.6.4 이후)

03. 인덱스 동작원리

B-Tree 구조

03. B-Tree 구조

인덱스 key값(학번) 자식 노드의 주소

SELECT * FROM students
WHERE id=201710084

101번 페이지

201610743	201
201710084	202

102번 페이지

201810903	203
201923512	204

201번 페이지

201610743	홍혜진	여	경영학과
201623211	박재현	남	경제학과

202번 페이지

201710084	조유정	여	컴퓨터과학과
201710234			

● ● ●

루트 페이지

브랜치 페이지
(중간 페이지)

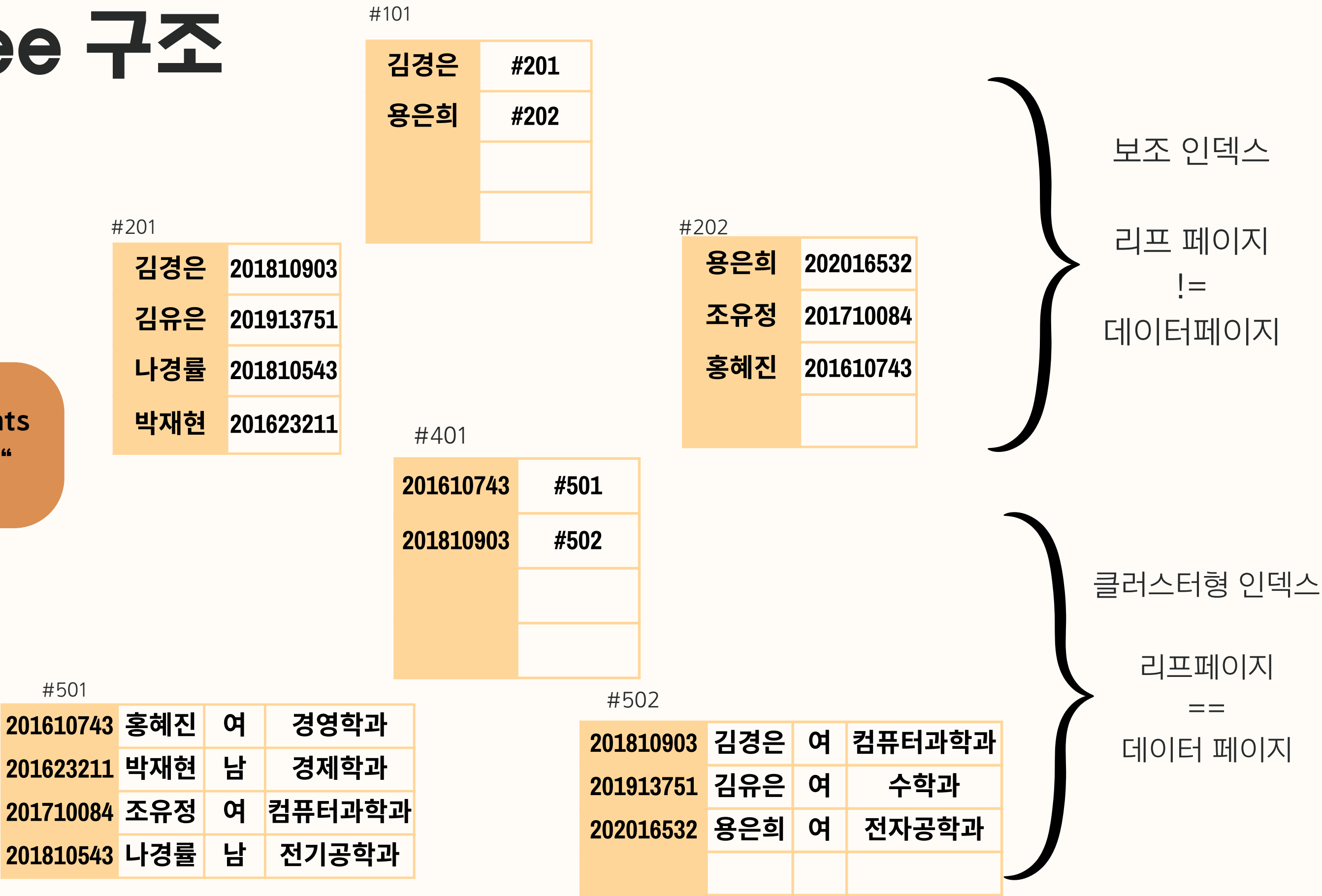
리프 페이지
==
데이터 페이지

03. B-Tree 구조



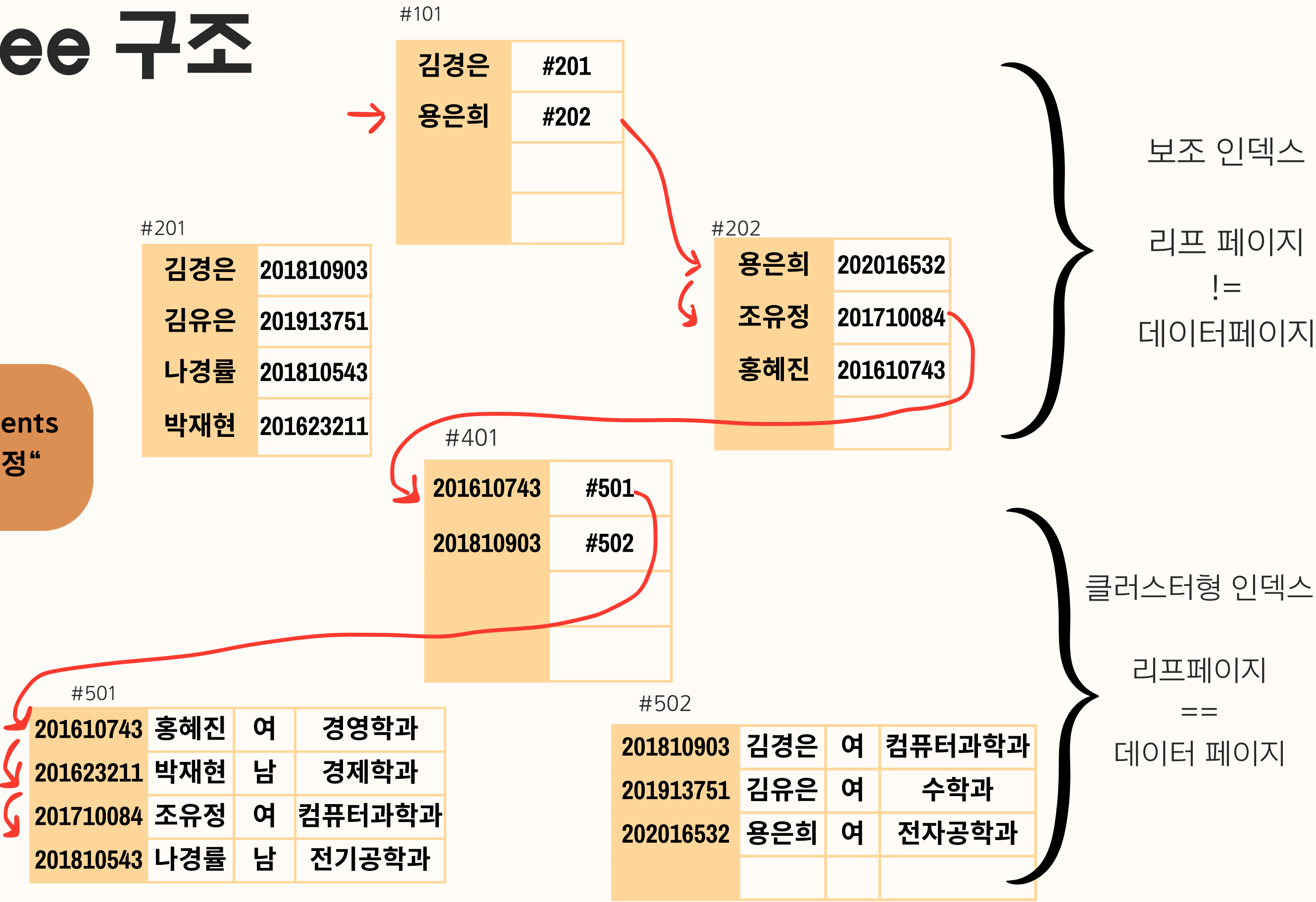
03. B-Tree 구조

SELECT * FROM students
WHERE name="조유정"



03. B-Tree 구조

SELECT * FROM students
WHERE name="조유정"



04. 인덱스의 효율적인 설계와 활용

04. 효율적인 인덱스 설계

첫번째.

카디널리티(분포도)가 높은 열에 걸어야 효과가 있다.

두번째.

자주 사용되어야 가치가 있다. SELECT가 자주 일어나는 열에 거는 것이 효과적이다.

세번째.

테이블의 20%이상을 스캔하는 경우에는 full scan이 더 효과적일 수 있다.

네번째.

WHERE절에서 연산을 하는 경우 인덱스가 사용되지 않을 수 있다.

다섯번째.

복합 인덱스의 경우에는 순서를 고려하여 인덱스를 설계해야한다.



04. 인덱스의 활용

- first_name에 인덱스를 적용하지 않은 경우 / 적용한 경우

Table: **employees**

Columns:

<u>emp_no</u>	int PK
birth_date	date
first_name	varchar(14)
last_name	varchar(16)
gender	enum('M','F')
hire_date	date

✓	2	20:3...	SELECT * FROM employees WHERE first_name = "Parto"	228 row(s) returned	0.141 sec / 0.000068...
✓	3	20:3...	CREATE INDEX `idx_employees_first_name` ON `empl...	OK	0.000 sec
✓	4	20:3...	SELECT * FROM employees WHERE first_name = "Parto"	228 row(s) returned	0.0028 sec / 0.00012...

-> 약 50.36배 빨라짐

- gender에 인덱스를 적용하지 않은 경우 / 적용한 경우

✓	5	20:3...	SELECT * FROM employees WHERE gender = 'M'	179973 row(s) returned	0.0023 sec / 0.180 sec
✓	6	20:3...	CREATE INDEX `idx_employees_gender` ON `employe...	OK	0.000 sec
✓	7	20:3...	SELECT * FROM employees WHERE gender = 'M'	179973 row(s) returned	0.0041 sec / 0.245 sec

-> 오히려 성능이 안좋아짐

05. 인덱스의 활용

- emp_no에 연산이 이루어지는 경우

```
EXPLAIN SELECT * FROM employees WHERE emp_no % 2 = 0;
```

Table: **employees**

Columns:

emp_no	int PK
birth_date	date
first_name	varchar(14)
last_name	varchar(16)
gender	enum('M','F')
hire_date	date

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	NULL	ALL	NULL	NULL	NULL	NULL	299468	100.00	Using where

-> 옵티마이저의 실행계획에 인덱스가 포함되어 있지 않다.

- 복합 인덱스의 순서에 따라 달라지는 쿼리 성능

```
SELECT * FROM employees WHERE first_name = "Parto" AND last_name = "Bamford" AND hire_date < '1990-01-01';
```

CREATE INDEX idx_hire_date_first ON employees(hire_date, first_name, last_name)	0 row(s) affected Recor...	0.434 sec
SELECT * FROM employees WHERE first_name = "Parto" AND last_name = "Bamfor..."	1 row(s) returned	0.00094 sec / 0.000...
DROP INDEX `idx_hire_date_first` ON `employees`.`employees`	OK	0.000 sec
CREATE INDEX idx_name_first ON employees(first_name, last_name, hire_date)	0 row(s) affected Recor...	0.596 sec
SELECT * FROM employees WHERE first_name = "Parto" AND last_name = "Bamfor..."	1 row(s) returned	0.00045 sec / 0.000...

Thank you

궁금한 점을 물어보세요