ai엔지니어링반 김가영

2024. 03. 06

SQL 성능 최적화 전략

2차 기술 세미나

Table of Contents

01 인덱스 최적화

인덱스(index) 정의

- 데이터베이스 테이블의 성능 향상

- 검색 및 정렬 작업을 빠르게 수행하기 위한 데이터의 논리적 순서를 저장하는 데이터 구조

01 인덱스 최적화

인덱스(index) 장단점

- 장점
  1. 데이터 검색 속도 향상
  2. 정렬된 데이터 접근 용이
  3. 데이터 집합 연산 최적화
- 단점
  1. 스토리지 공간 요구
  2. 데이터 변경에 따른 오버헤드
  3. 인덱스 선택과 관리의 복잡성

01 인덱스 최적화

인덱스 종류(데이터 저장방식)

클러스터형 인덱스

- 테이블 당 **1개만** 존재 가능
- 데이터 정렬에 따른 영향
- 물리적으로 레코드 정렬

보조 인덱스

- 테이블 당 **여러 개** 존재 가능
- 데이터 정렬에 영향 미치지 않음
- 다양한 쿼리 유형 지원

01 인덱스 최적화

인덱스 생성 조건

• **카디널리티(Cardinality)**란?

데이터베이스에서 특정 열이나 인덱스의 고유한 값들의 개수

1. **카디널리티(Cardinality)**가 높은 컬럼에 지정

= 데이터 중복이 적은 컬럼 ex) ID, 주민번호

2. 검색빈도(활용도)

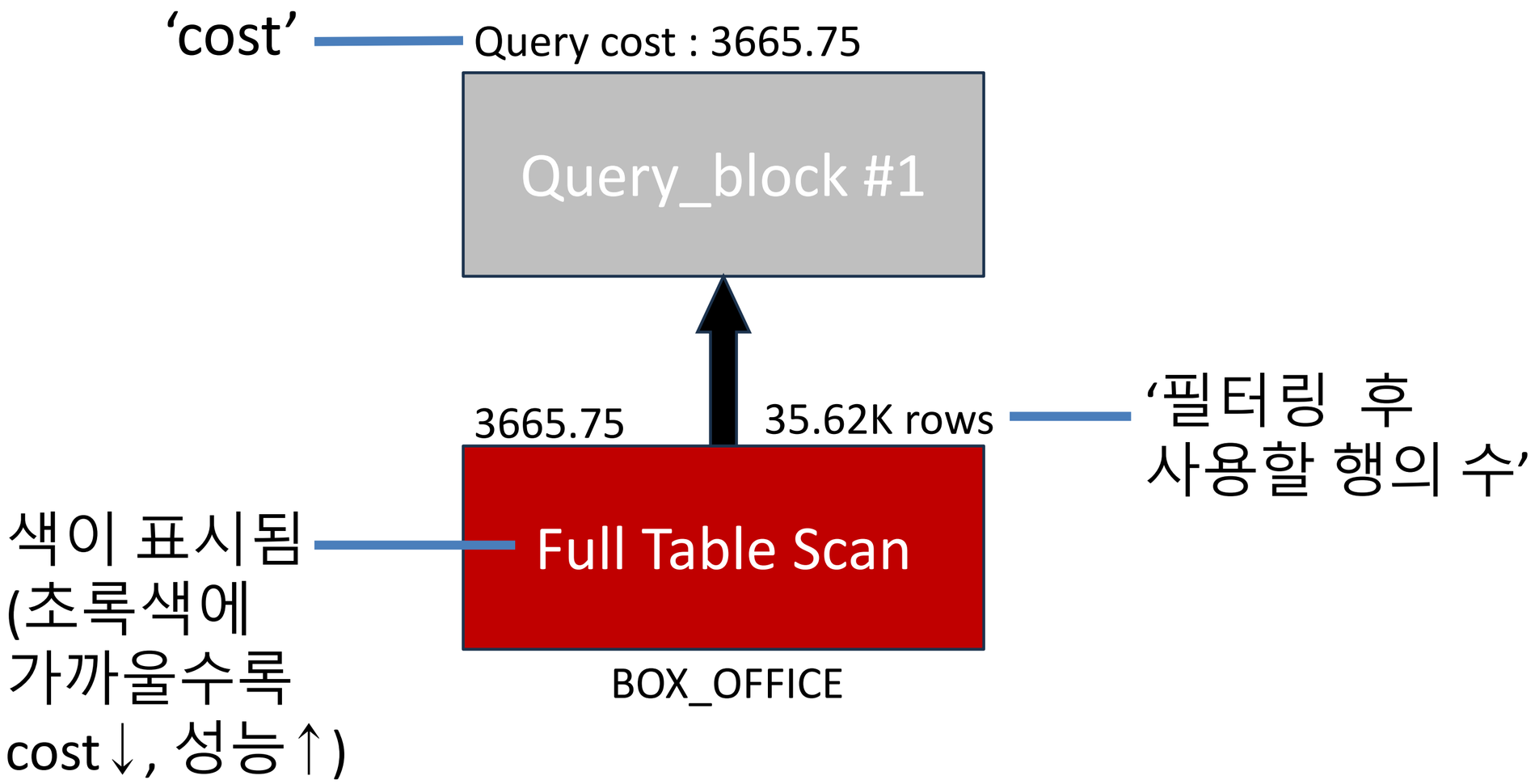
자주 검색되는 필드일수록 인덱스 생성하는 것이 좋음

3. 필터링과 정렬

WHERE 절에서 필터링이나 ORDER BY 절에서 정렬에 사용되는 필드

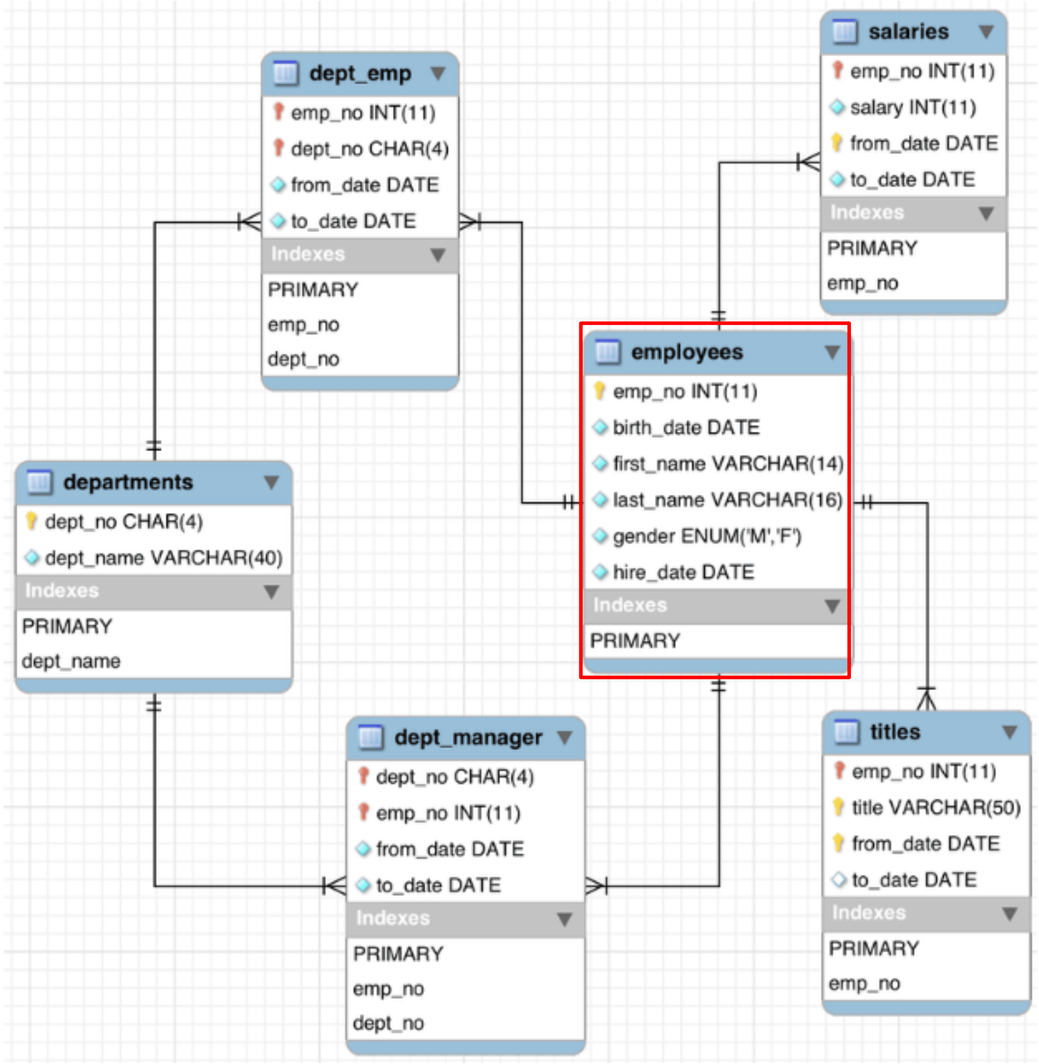
01 인덱스 최적화

- Execution Plan(실행 계획)



01 인덱스 최적화

Figure 1 The Employees Schema



- 사용한 데이터

MySQL 샘플 데이터

employees 테이블

300,024건

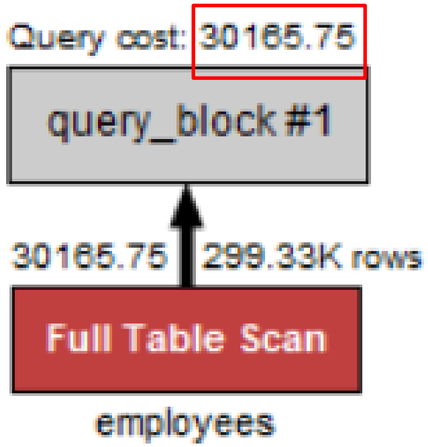
count(*)
300024

01 인덱스 최적화

인덱스 생성에 따른 성능 차이

```sql
SELECT
    last_name
FROM employees
WHERE employees.birth_date = '1953-09-02'
AND employees.first_name = 'Georgi';
```

SELECT last_name FROM employees WHERE employees.birth_date = '1953-09-02' AND employees.first...	1 row(s) returned	0.109 sec / 0.000 sec



01 인덱스 최적화

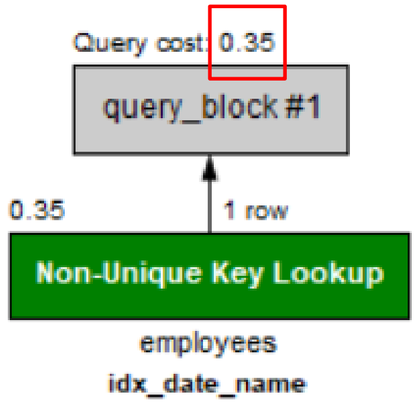
인덱스 생성에 따른 성능 차이

((birth_date, first_name) 인덱스 생성 한 후)

```
CREATE index idx_date_name ON gayoung.employees(birth_date, first_name);
SHOW index FROM gayoung.employees;
```

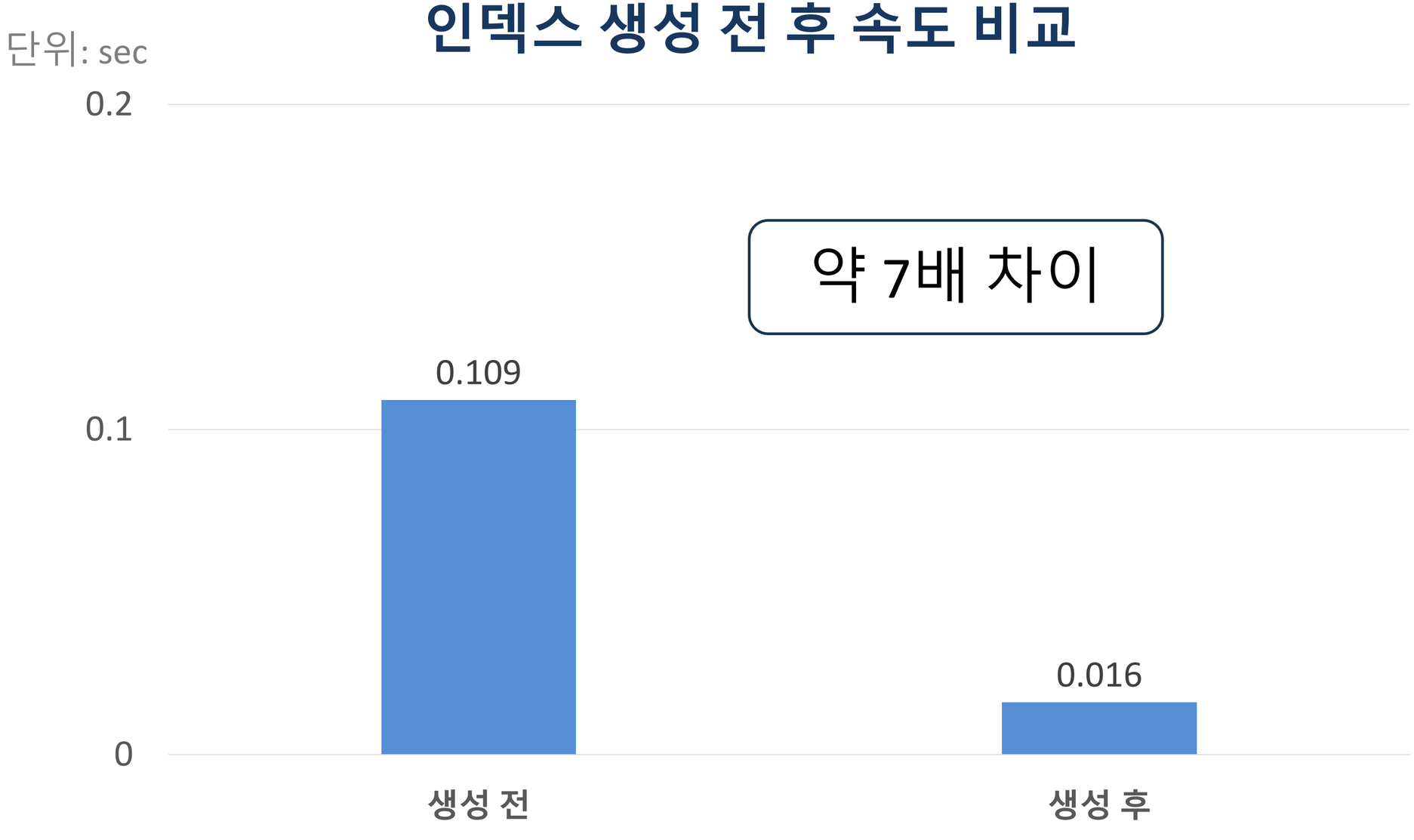
Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality
employees	0	PRIMARY	1	emp_no	A	299335
employees	1	idx_date_name	1	birth_date	A	4708
employees	1	idx_date_name	2	first_name	A	293405

SELECT last_name FROM employees WHERE employees.birth_date = '1953-09-02' AND employees.first... 1 row(s) returned 0.016 sec / 0.000 sec



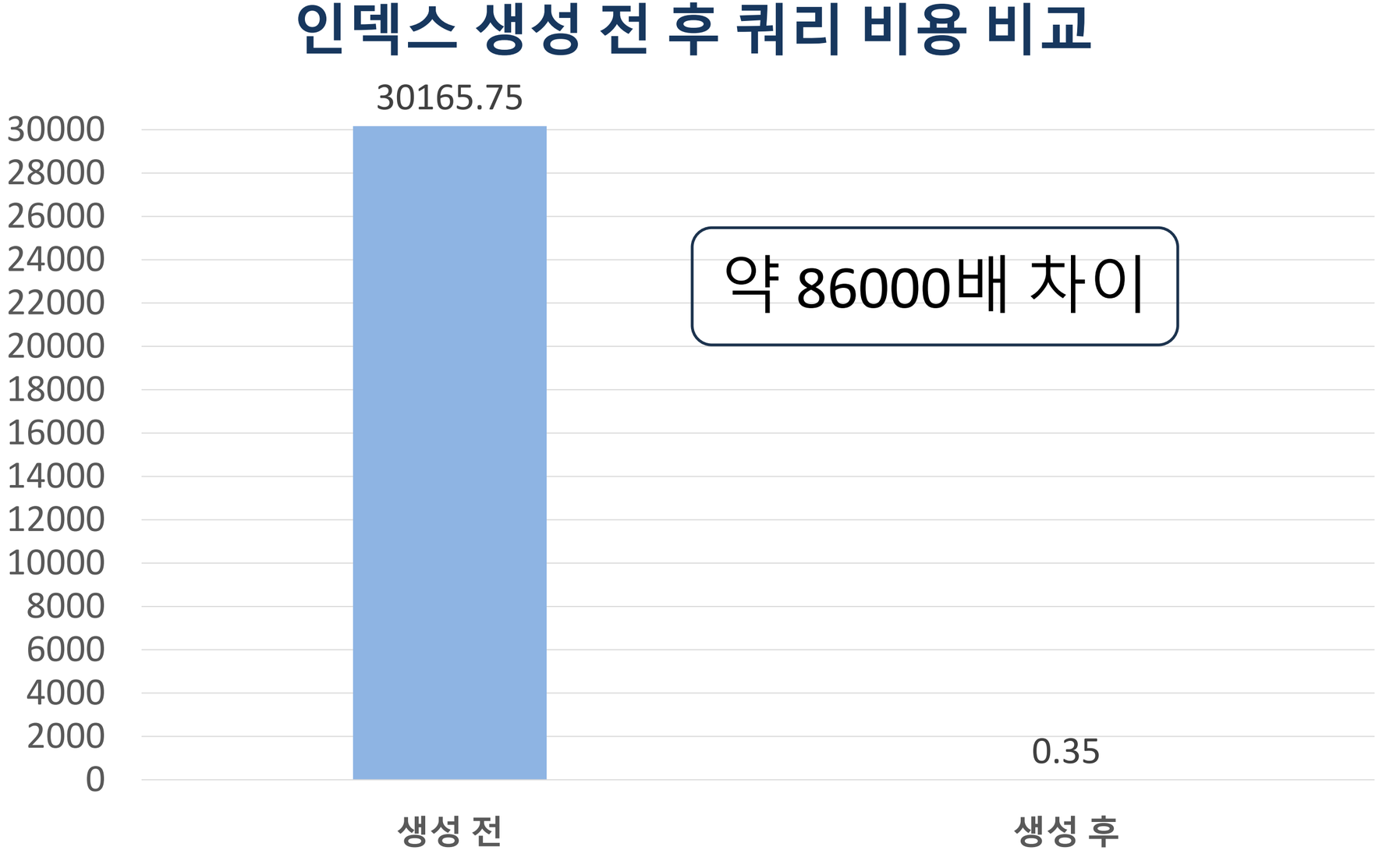
01 인덱스 최적화

인덱스 생성에 따른 성능 차이



01 인덱스 최적화

인덱스 생성에 따른 성능 차이



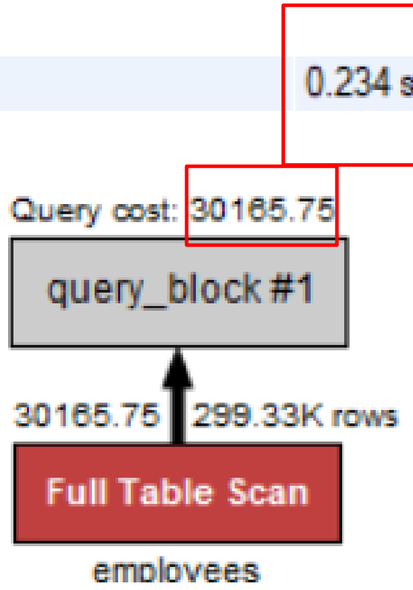
01 인덱스 최적화

인덱스 컬럼 순서에 따른 성능 차이

```sql
SELECT
    last_name
FROM employees
WHERE employees.birth_date != '1953-09-02'
AND employees.first_name = 'Georgi';
```

SELECT last_name FROM employees WHERE employees.birth_date != '1953-09-02' A...	252 row(s) returned	0.234 sec / 0.000 sec

Table	Non_unique	Key_name	Seq_in_index	Column_name
employees	0	PRIMARY	1	emp_no
employees	1	idx_date_name	1	birth_date
employees	1	idx_date_name	2	first_name



01 인덱스 최적화

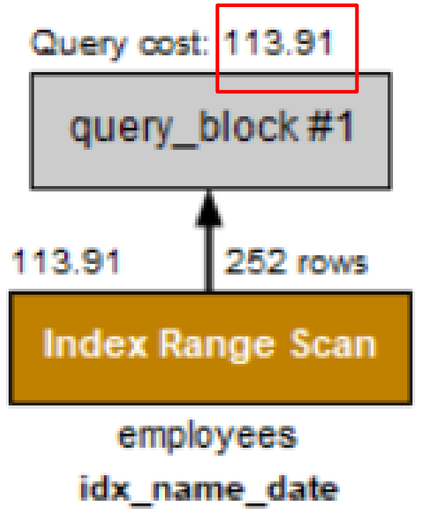
인덱스 컬럼 순서에 따른 성능 차이

(컬럼순서가 반대인 (first_name, birth_date) 인덱스 생성 한 후)

```
CREATE index idx_name_date ON gayoung.employees(first_name, birth_date);
SHOW index FROM gayoung.employees;
```

SELECT last_name FROM employees WHERE employees.birth_date != '1953-09-02' A... | 252 row(s) returned | 0.015 sec / 0.000 sec

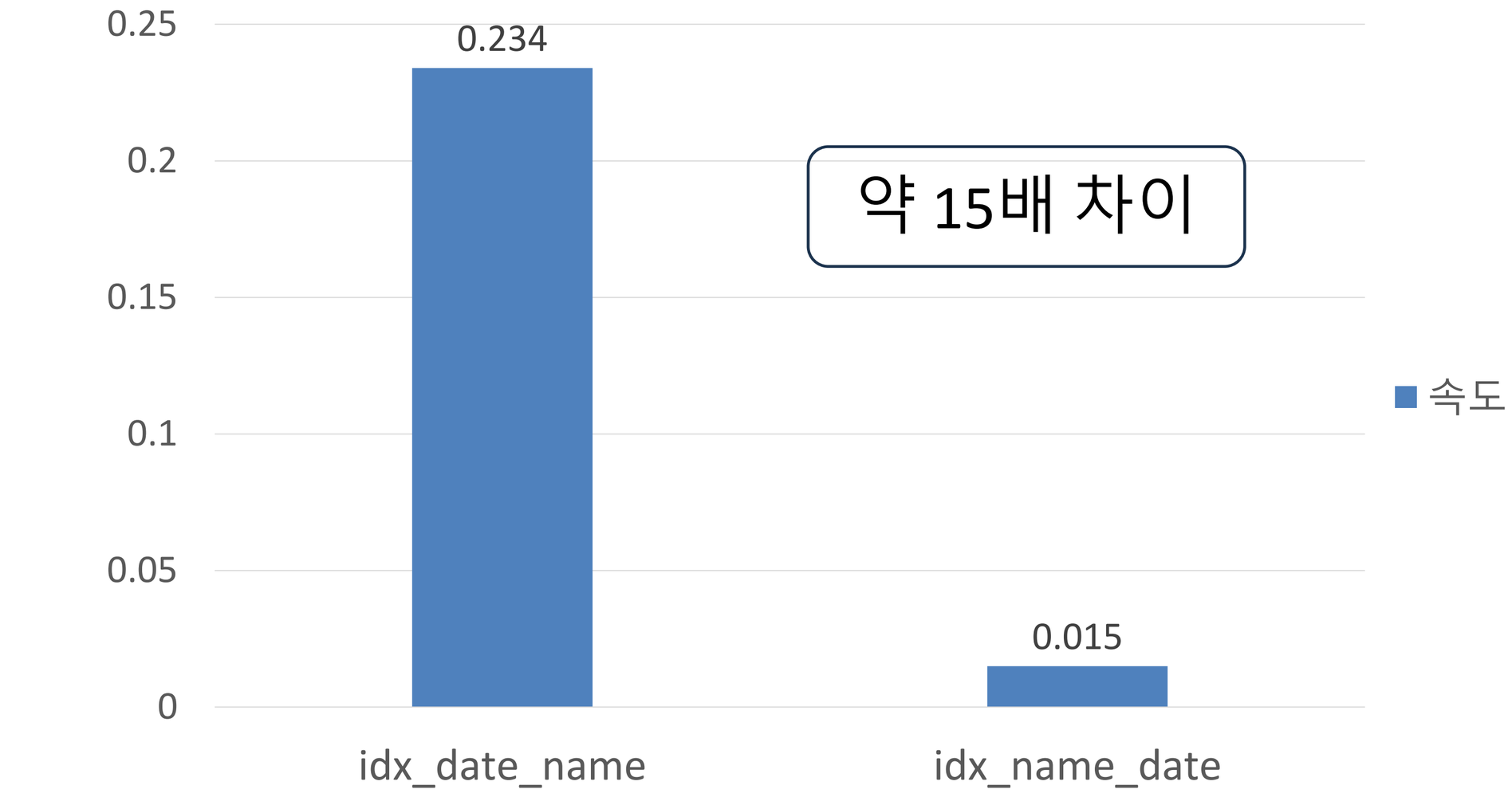
Table	Non_unique	Key_name	Seq_in_index	Column_name
employees	0	PRIMARY	1	emp_no
employees	1	idx_name_date	1	first_name
employees	1	idx_name_date	2	birth_date



01 인덱스 최적화

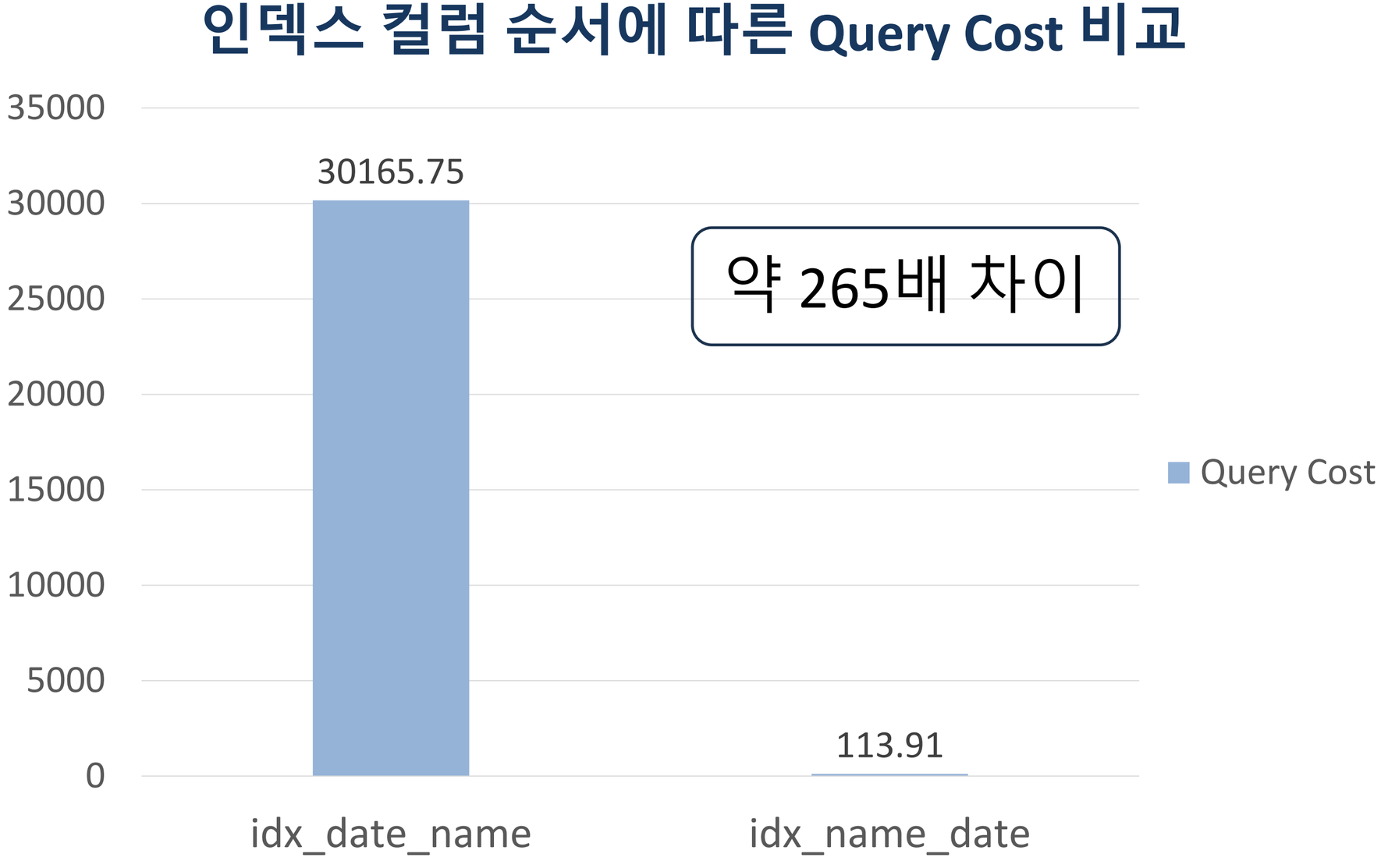
인덱스 컬럼 순서에 따른 성능 차이

단위 : sec



01 인덱스 최적화

인덱스 컬럼 순서에 따른 성능 차이



01 인덱스 최적화

인덱스 컬럼 순서에 따른 성능 차이

- 기존 (birth_date, first_name) 인덱스

birth_date != '1953-09-02' & **first_name = 'Georgi'** :

'1953-09-02'가 아닌 모든 데이터에 대해 first_name = 'Georgi' 비교를 수행해야 함.

01 인덱스 최적화

인덱스 컬럼 순서에 따른 성능 차이

- (first_name , birth_date) 인덱스

birth_date != ‘1953-09-02’ & **first_name = ‘Georgi’** :

first_name = ‘Georgi’ 인 것만 발견 => 이름이 달라지는 부분이

나타나기 전까지만 데이터 스캔이 필요

01 인덱스 최적화

인덱스 컬럼 순서에 따른 성능 차이

- 결론

다중 컬럼 인덱스 내 컬럼의 순서는 성능에 큰 영향을 미칠 수 있음

=> 어떤 방식의 조회가 자주 일어나는지, 필터링 방식

고려해봐야 함.

01 인덱스 최적화

다중 컬럼 인덱스의 사용 가능 여부

Table	Non_unique	Key_name	Seq_in_index	Column_name
employees	0	PRIMARY	1	emp_no
employees	1	idx_name_date	1	first_name
employees	1	idx_name_date	2	birth_date

```
EXPLAIN
SELECT
    last_name
FROM employees
WHERE employees.birth_date = '1953-09-02';
```

01 인덱스 최적화

다중 컬럼 인덱스의 사용 가능 여부

(쿼리 실행 계획)

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	NULL	ALL	NULL	NULL	NULL	NULL	299335	10.00	Using where

존재하는 인덱스 idx_name_date 사용하지 X,

거의 모든 row 스캔했음 알 수 있음.

=> name으로 먼저 정렬한 후, date 컬럼으로 정렬되었기 때문

선행하는 컬럼에 대한 조건 없는 경우

'작업 범위 결정 조건' 정하지 못해 인덱스 사용 못함.

01 인덱스 최적화

다중 컬럼 인덱스의 사용 가능 여부

- 결론

다중 컬럼 인덱스에서 선행하는 컬럼에 대한 조건을

WHERE문에 포함하지 않는다면

=> 인덱스 사용 불가능

02 파티셔닝 최적화

파티셔닝 개념

파티셔닝 정의

대량의 테이블을 물리적으로 **여러 개**의 테이블로 쪼개는 것

=> 데이터베이스 성능↑, 데이터 관리 간소화, 백업 및 복구 시간 ↓

02 파티셔닝 최적화

파티셔닝의 유형

- RANGE 파티셔닝

연속적인 값 범위 기반으로 데이터 분할

- LIST 파티셔닝

명시적으로 정의된 값 목록 기반으로 데이터 분할

- HASH 파티셔닝

해시 함수 사용하여 데이터 분할

02 파티셔닝 최적화

파티셔닝 성능 최적화 팁

- 적절한 파티션 키 선택

WHERE 절에 자주 사용되는 열이어야 함.

- 적절한 파티션 크기 선택

너무 많거나 작으면 활용하기 어려움.

- 파티션 유지 관리

정기적으로 유지 관리 해야함. (삭제, 추가)

- 인덱스 사용

파티션 키와 함께 사용되는 열에 인덱스 생성하면 쿼리 성능 향상됨.

02 파티셔닝 최적화

파티셔닝 속도, 성능 차이

```
CREATE TABLE employees (
    emp_no      INT             NOT NULL,
    birth_date  DATE            NOT NULL,
    first_name  VARCHAR(14)     NOT NULL,
    last_name   VARCHAR(16)     NOT NULL,
    gender      ENUM ('M','F')  NOT NULL,
    hire_date   DATE            NOT NULL
)
PARTITION BY RANGE(year(birth_date)) (
    PARTITION part1 VALUES LESS THAN (1953),
    PARTITION part2 VALUES LESS THAN (1960),
    PARTITION part3 VALUES LESS THAN MAXVALUE
);
```

출생년도 별로 3개의 파티션으로 구분

part1 : 1953년 미만

part2 : 1953년~1960년 미만

part3 : 1960년 이상

02 파티셔닝 최적화

파티셔닝 속도, 성능 차이

```
SELECT
    TABLE_SCHEMA, TABLE_NAME, PARTITION_NAME, PARTITION_ORDINAL_POSITION, TABLE_ROWS
FROM INFORMATION_SCHEMA.PARTITIONS
WHERE TABLE_NAME='employees';
```

TABLE_SCHEMA	TABLE_NAME	PARTITION_NAME	PARTITION_ORDINAL_POSITION	TABLE_ROWS
gayoung	employees	part1	1	42563
gayoung	employees	part2	2	322571
gayoung	employees	part3	3	233537

파티션 정보 확인

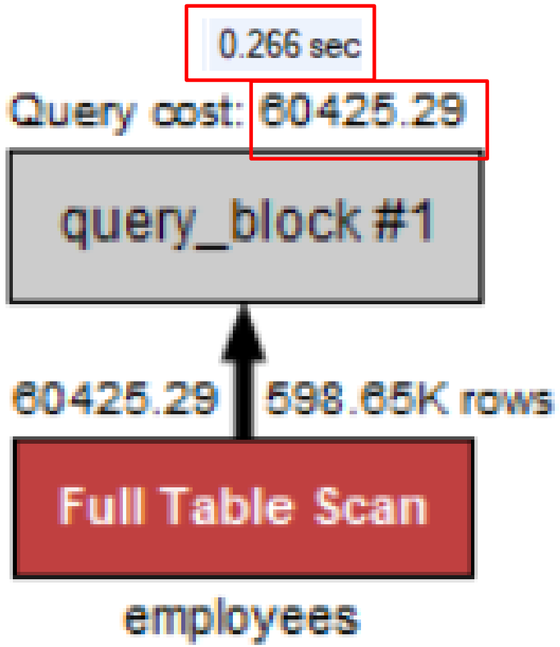
02 파티셔닝 최적화

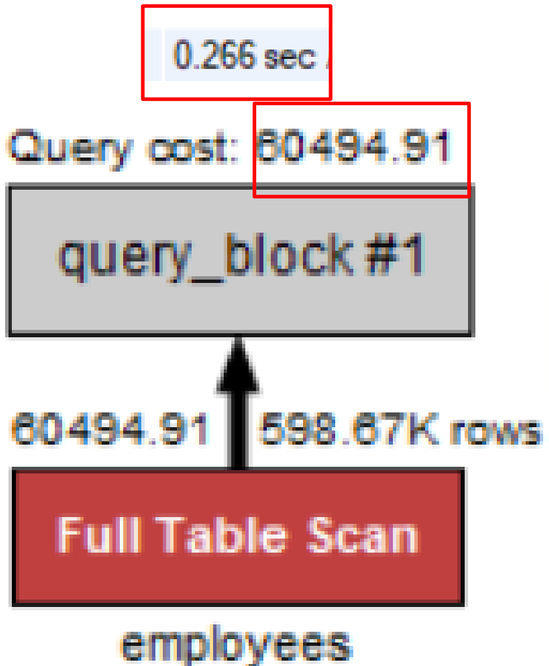
파티셔닝 속도, 성능 차이 -1953년 이전 출생한 직원 수 검색

```
SELECT
    count(emp_no)
FROM employees
WHERE year(birth_date) < 1953;
```

count(emp_no)
42418

파티션 x





파티션 o
적용 x

02 파티셔닝 최적화

파티셔닝 속도, 성능 차이

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	part1,part2,part3	ALL	NULL	NULL	NULL	NULL	598671	100.00	Using where

02 파티셔닝 최적화

파티셔닝 속도, 성능 차이

	속도(sec)	쿼리비용
파티션 X	0.266	60425.29
	=	<
파티션O	0.266	60494.91

파티션을 만들더라도

쿼리를 제대로 작성하지 않으면 속도, 쿼리비용에 도움 되지 않음!

02 파티셔닝 최적화

파티셔닝 속도, 성능 차이 -1953년 이전 출생한 직원 수 검색

```
SELECT
    count(emp_no)
FROM employees
WHERE birth_date < DATE_FORMAT('1953-01-01', '%Y%m%d');
```

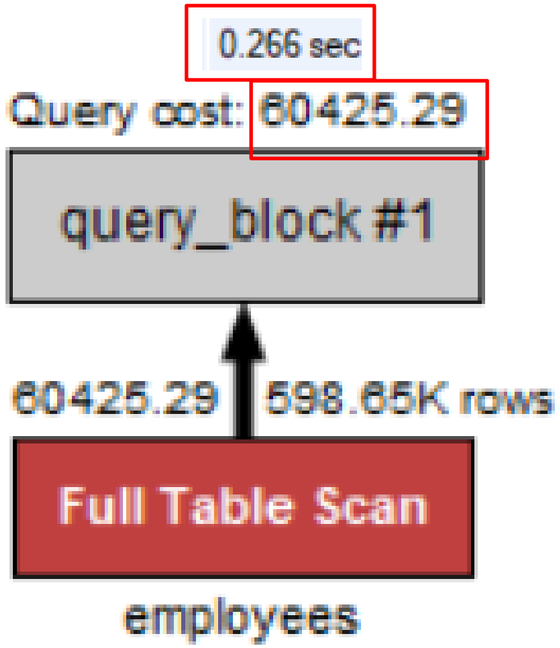
파티션 생성o

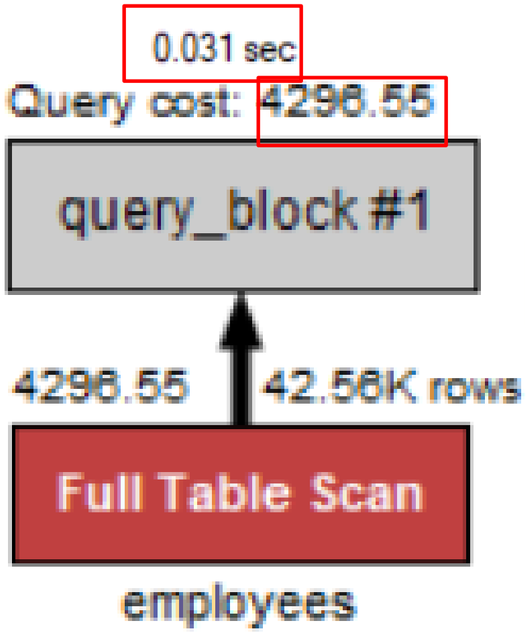
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	part1	ALL	NULL	NULL	NULL	NULL	42563	33.33	Using where

02 파티셔닝 최적화

파티셔닝 속도, 성능 차이 -1953년 이전 출생한 직원 수 검색

파티션 x

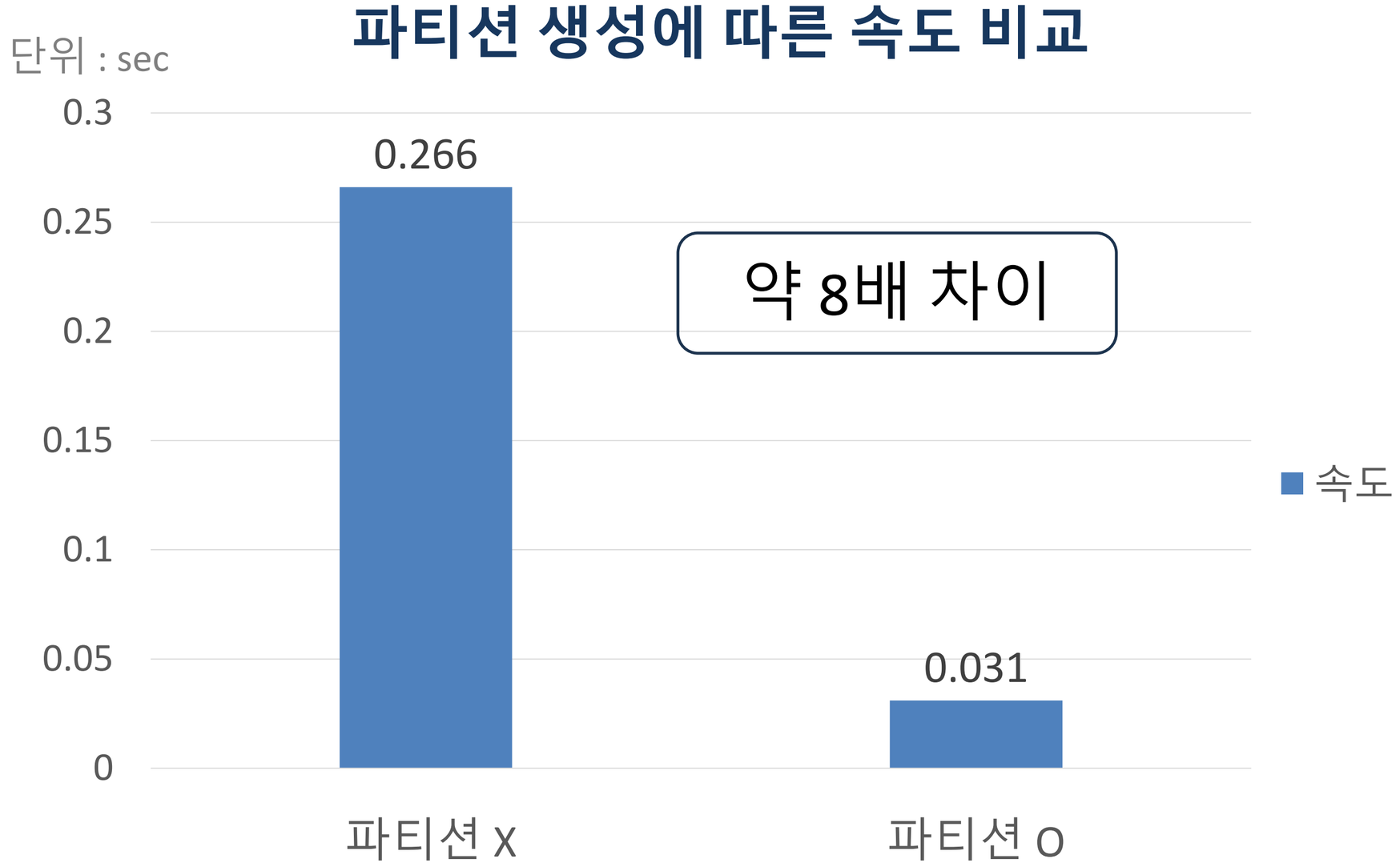




파티션 o

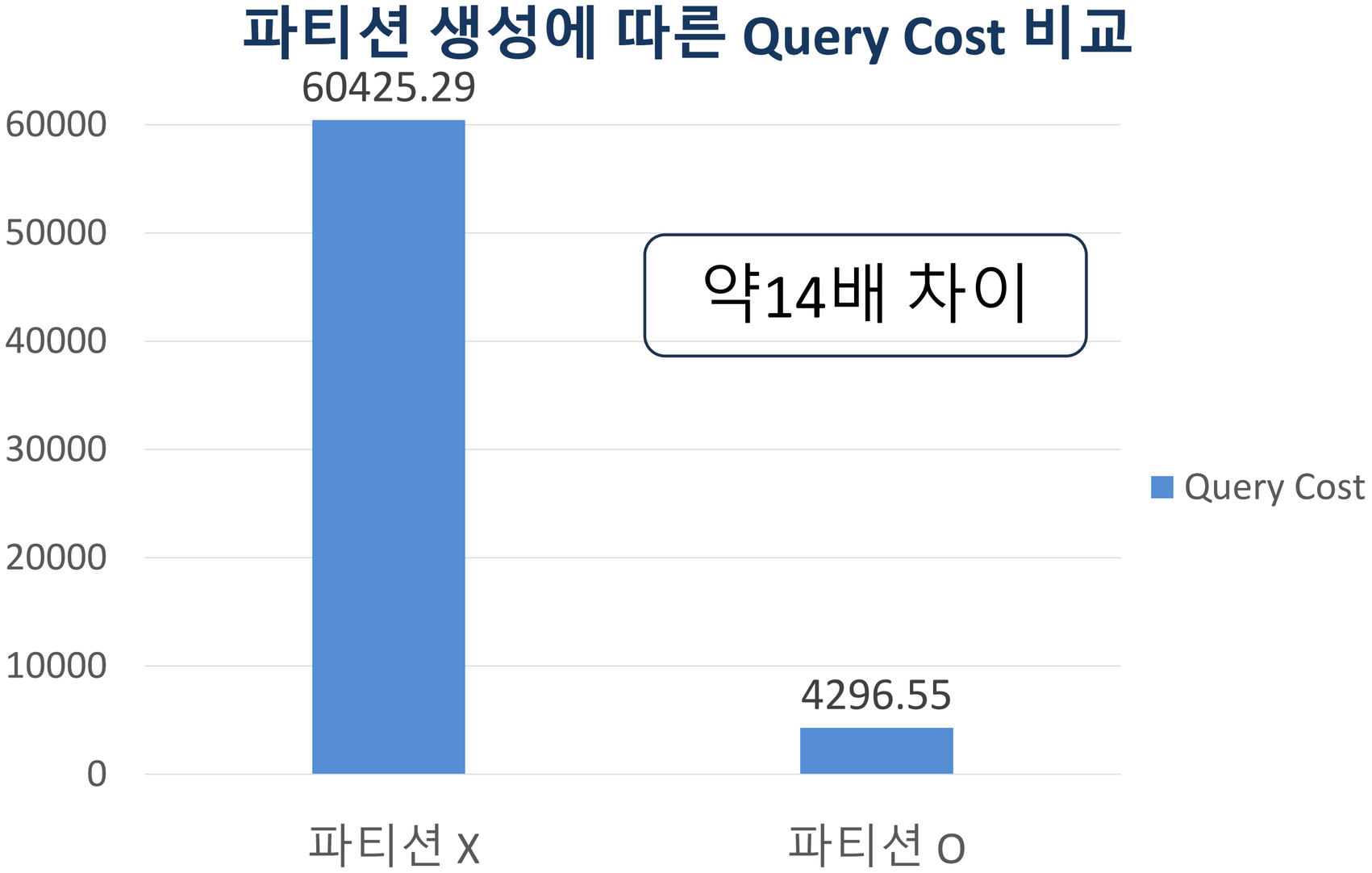
02 파티셔닝 최적화

파티셔닝 속도, 성능 차이



02 파티셔닝 최적화

파티셔닝 속도, 성능 차이



02 파티셔닝 최적화

파티셔닝 속도, 성능 차이 -1953년 이전 출생한 직원 수 검색

```
SELECT
    count(emp_no)
FROM employees
WHERE year(birth_date) < 1953;
```

'birth_date'의 연도를 추출하여

비교하기 때문에 파티션 적용 X!

```
SELECT
    count(emp_no)
FROM employees
WHERE birth_date < DATE_FORMAT('1953-01-01', '%Y%m%d');
```

'birth_date' 자체를 직접 비교하는 조건

-> 파티션 적용 O

03 쿼리 튜닝

‘EXPLAIN’ 명령어의 활용

쿼리 실행 계획을 확인

```
EXPLAIN
SELECT
    last_name
FROM employees
WHERE employees.first_name = 'Parto';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employees	NULL	ref	idx_name_date	idx_name_date	58	const	228	100.00	NULL

03 쿼리 튜닝

‘EXPLAIN’ 명령어의 활용

구분	설명
id	아이디(id)로 SELECT 구분하는 번호. 실행 순서 표시
select_type	SQL문을 구성하는 SELECT문의 유형 출력(SIMPLE, PRIMARY, UNION 등)
table	테이블 명(별명)
type	테이블의 데이터를 어떻게 찾을지에 관한 정보 제공. 조인 혹은 조회 타입
possible_keys	데이터 조회할 때 DB에서 사용할 수 있는 인덱스 리스트
key	실제로 옵티마이저가 최적화 검색에 사용한 인덱스가 표시되는 필드
key_len	선택된 인덱스의 길이
ref	key 컬럼에 나와 있는 인덱스에서 값 찾기 위해 선행 테이블의 어떤 컬럼이 비교되었는지
rows	원하는 행 찾기 위해 얼마나 많은 행을 읽어야 할 지에 대한 예측값
extra	옵티마이저가 동작하는 방식에 대한 부가정보

03 쿼리 튜닝

‘EXPLAIN’ 명령어의 활용

- type

현재 실행되는 쿼리의 테이블 접근 방법

(Best -> Worst 순)

system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range > index > all

03 쿼리 튜닝

‘EXPLAIN’ 명령어의 활용

- type

- **system** : 단일 행을 반환하는 쿼리에 대한 최적화된 방식
- **const** : 기본 키(Primary) 또는 고유 인덱스 값으로 검색하는 경우
- **eq_ref** : JOIN 시에 기본 키(Primary) 또는 고유 인덱스를 사용하는 경우, 하나의 레코드만 검색
- **range** : **인덱스**(Index)를 **범위**로 검색하는 경우
- **Index** : **인덱스**(Index)를 Full Scan하는 경우
- **all** : 테이블을 Full Scan 하는 경우

03 쿼리 튜닝

‘EXPLAIN’ 명령어의 활용

- key

MySQL 옵티마이저가 사용하기로 결정한 인덱스(Index) 정보,

NULL일 경우 인덱스 사용 X

- rows

MySQL 옵티마이저가 쿼리 실행 시 예상되는 총 행의 수,

이 값이 적을수록 성능↑

03 쿼리 튜닝

‘EXPLAIN’ 명령어의 활용

- extra

쿼리 실행과 관련된 추가 정보 **(Best -> Worst 순)**

- **using index** : 인덱스(Index)를 사용해 데이터 추출
- **using where** : where 조건으로 데이터 추출
- **using temporary**
  - MySQL에서 임시 테이블 생성해 추출
  - ORDER BY, GROUP BY등의 연산, Sub Query 시 사용 가능
  - 디스크 저장 -> I/O 비용으로 인한 성능 하락 가능
- **using filesort** : ORDER BY 수행 시, 인덱스 X 정렬하는 경우 사용됨. 성능 하락 가능.

03 쿼리 튜닝

‘EXPLAIN’ 명령어의 활용

(‘EXPLAIN’ 명령어)

한계 : 분석의 정확성 떨어질 수 있음

원인 : 통계 정보의 부족 OR 불일치

‘ANALYZE’ 명령어

감사합니다