Index Range Scan

인덱스 루트 블록에서 리프 블록까지 수직적으로 탐색한 후에 리프 블록을 필요한 범위만 스캔하는 방식

B-Tree 인덱스의 가장 일반적이고 정상적인 액세스 방식

인덱스를 스캔하는 범위를 얼마만큼 줄일 수 있느냐와 테이블로 액세스하는 횟수를 얼마만큼 줄일 수 있느냐

인덱스를 구성하는 선두 컬럼이 조건절에 사용되어야 index Range Scan 가능

Index Range Scan 과정을 거쳐 생성된 결과집합은 인덱스 컬럼 순으로 정렬, order by 연산을 생략하거나 min/max값을 빠르게 추출 가능

between, like, 부등호 조건 처리

Multi-Column indexes 사용시에 range scan

우선 설명을 위해 다음과 같은 테이블을 정의하였습니다.

```
CREATE TABLE `example`.`user` (
  `id` BIGINT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NOT NULL,
  `age` INT NOT NULL,
  `birthday` DATE NOT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_user_birthday_name_age` (`birthday`,`name`,`age`)
);
```

위 테이블에서는 multi-column index를 사용하고
해당되는 column은 birthday, name, 그리고 age 정보입니다.

그럼 해당 테이블에서 아래와 같은 query를 수행하면 결과가 어떻게 될까요?

```
explain analyze select * from user where birthday = '1997-08-27' and name = 'James' and age = 1;
```

결과를 살펴보기 앞서,
explain analyze 명령어가 익숙하지 않은 분들을 위해 간단하게 설명하고 넘어가겠습니다.

먼저, **explain은 query의 수행 계획을 조회**하기 위한 명령어 입니다.

explain은 MySQL 8.0.19 이상부터 지원하며,
optimizer로 부터 query 수행 계획을 가져와 보여줍니다.

그렇다면 optimizer는 무엇일까요?
이름에서 알 수 있듯이, 우리가 실행하려는 query를 최적화(optimize)하는 역할을 합니다.

최적화 과정은 불필요한 조건을 제거하거나 쿼리를 재조합하는 등의 동작이 포함되어있습니다.

explain analyze는 explain에 추가적으로 다음과 같은 정보를 우리에게 제공합니다.

- query 수행 예상 비용
- 결과 row의 예상 개수
- 결과의 첫번째 row가 반환되는 시간(milliseconds)
- 모든 결과가 반환되는 시간(milliseconds)
- iterator가 반환하는 row의 개수
- loop가 수행되는 횟수

↑ 'iterator'가 순회를 실시하는 총 횟수

explain analyze의 결과는 tree의 형태로 출력되며,
위 항목들에 대한 자세한 설명은 query의 결과를 살펴보면서 조금 더 자세히 설명하겠습니다.

다시 예시로 돌아와서,

```
explain analyze select * from user where birthday = '1997-08-27' and name = 'James' and age = 1;
```

위 query를 수행해보면 다음과 같은 결과를 확인할 수 있습니다.

```
-> Index lookup on user using idx_user_birthday_name_age
      (birthday=DATE'1997-08-27', name='James', age=1)
      (cost=1.72 rows=6) (actual time=0.013..0.016 rows=6 loops=1)
```

위 결과가 무엇을 의미하는 지 천천히 살펴보겠습니다.
우선 첫번째 줄에서는 해당 query가 index를 사용했으며, 어떤 index를 사용했는지를 알려줍니다.
(위의 query에서는 birthday)

두번째 줄에서는 위 index에서 어떤 column을 사용했는지를 알려주고,
마지막 줄에서는 위에서 설명한 qeury 수행에 대한 세부 정보를 표기합니다.

- cost=1.72
  - query가 수행되는데 예상되는 비용(1.72)을 표기합니다.

- rows=6
  - 예상되는 결과 row의 갯수

- actual time=0.013..0.016
  - query가 수행되는데 걸린 시간을 나타냅니다.
  - 0.013은 첫번째 row가 반환되는 시간을 의미합니다.
  - 0.016는 모든 row가 반환되는데 걸리는 시간을 의미합니다.

- loops=1
  - loop가 수행된 횟수

- rows=6
  - 각 iterator가 반환하는 rows의 개수

그럼 이번에는 query를 조금 수정해보겠습니다.

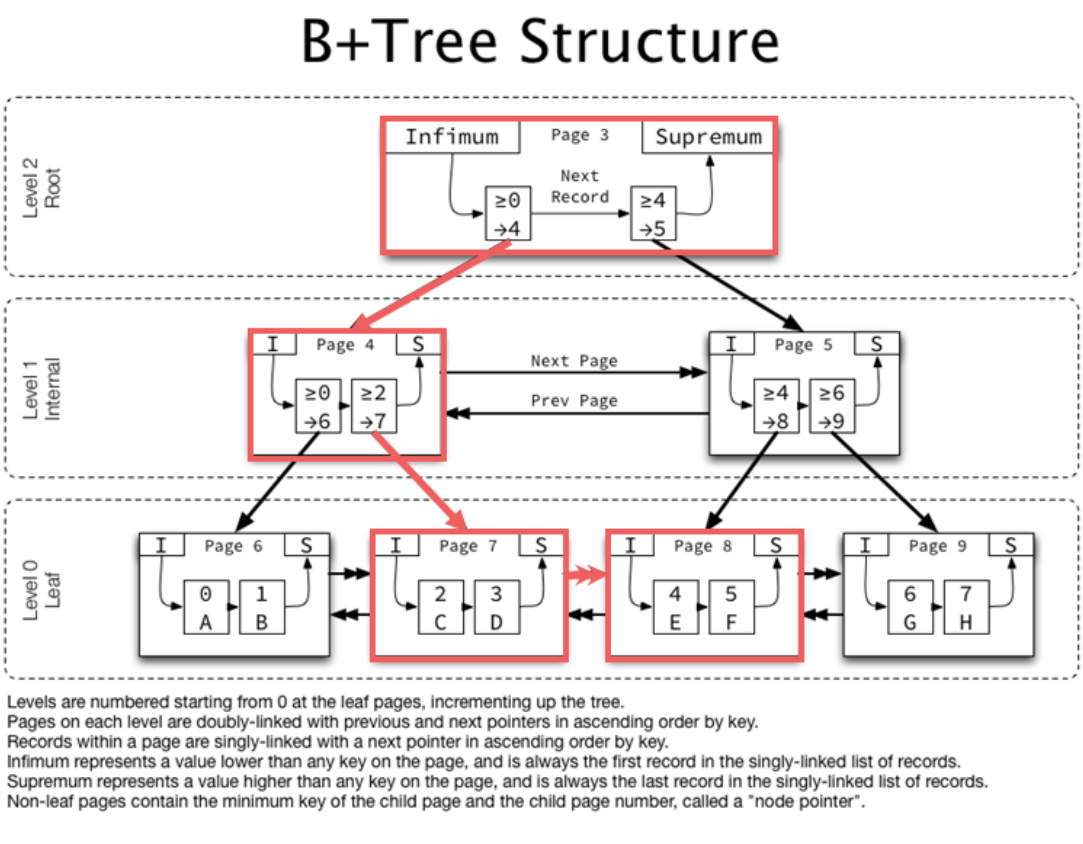
```
explain analyze select * from user where birthday >= '1997-08-27' and name = 'James' and age = 1;
```

이번에는 생일에 대하여 range scan을 수행하였습니다.
위 qeury의 결과는 다음과 같습니다.

```
> Filter: ((`user`.age = 1) and (`user`.`name` = 'James') and (`user`.birthday >= DATE'1997-08-27'))
        (cost=3490.02 rows=157) (actual time=0.904..2.598 rows=6 loops=1)
    -> Index range scan on user using idx_user_birthday_name_age
        (cost=3490.02 rows=15663) (actual time=0.025..1.942 rows=9080 loops=1)
```

위 query 역시 index를 활용하였지만 이전의 예제와는 결과가 조금 다릅니다.

우선, **Index lookup**이 **아닌 Index range scan**을 수행하였습니다.
MySQL(InnoDB)는 다음과 같이 B+Tree 자료구조로 index를 관리합니다.



Levels are numbered starting from 0 at the leaf pages, incrementing up the tree.
Pages on each level are doubly-linked with previous and next pointers in ascending order by key.
Records within a page are singly-linked with a next pointer in ascending order by key.
Infimum represents a value lower than any key on the page, and is always the first record in the singly-linked list of records.
Supremum represents a value higher than any key on the page, and is always the last record in the singly-linked list of records.
Non-leaf pages contain the minimum key of the child page and the child page number, called a "node pointer".

InnoDB Index B+Tree structure(출처: https://blog.jcole.us/2013/01/10/btree-index-structures-in-innodb/)

index range scan은 위의 그림에 표시된 것과 같이 모든 index를 탐색하지 않고
특정 level까지 탐색한 뒤, 같은 level의 page 참조를 통해,
우리가 원하는 range에 해당하는 값을 찾게됩니다.
(여기에서 중요한 것은 tree를 통해 range scan을 한다는 사실이기 때문에
tree 내부의 구체적인 구조에 대해서는 다루지 않겠습니다.)

그런데 위의 결과값에는 무엇을 기준으로 range scan을 하고 있는지 나와있지 않습니다.

```
explain format=JSON select * from user where birthday >= '1997-08-27' and name = 'James' and age = 1;
```

이를 확인하기 위해 위와 같은 query를 다시 수행합니다.

직전의 query와 다른 점은 **analyze 명령어 대신 format=JSON을 사용했다는 점**입니다.
위의 query를 수행하면, 다음과 같은 결과를 얻을 수 있습니다.

```
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "3389.49"
    },
    "table": {
      "table_name": "user",
      "access_type": "range",
      "possible_keys": [
        "idx_user_birthday_name_age"
      ],
      "key": "idx_user_birthday_name_age",
      "used_key_parts": [
        "birthday"
      ],
      "key_length": "189",
      "rows_examined_per_scan": 15163,
      "rows_produced_per_join": 151,
      "filtered": "1.00",
      "using_index": true,
      "cost_info": {
        "read_cost": "3374.32",
        "eval_cost": "15.16",
        "prefix_cost": "3389.49",
        "data_read_per_join": "29K"
      },
      "used_columns": [
        "id",
        "name",
        "age",
        "birthday"
      ],
      "attached_condition": "((`example`.`user`.`age` = 1) and (`example`.`user`.`name` = 'James') and (`example`.`us
er`.`birthday` >= DATE'1997-08-27'))"
    }
  }
}
```


기존 explain analyze를 사용했을 때보다 많은 정보들을 확인 할 수 있습니다.
그 중, **key와 used_key_parts의 값을 확인해보면 idx_user_birthday_name_age index를 사용했고,**
그 중 birthday column을 활용했다는 것을 확인할 수 있습니다.

그런데 우리가 정의한 user table schema를 다시 가져와보면,

```
CREATE TABLE `example`.`user` (
  `id` BIGINT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NOT NULL,
  `age` INT NOT NULL,
  `birthday` DATE NOT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_user_birthday_name_age` (`birthday`,`name`,`age`)
);
```

idx_user_birthday_name_age index는 birthday, name, age colum으로 구성된 multi-colum index이고,

```
select * from user where birthday >= '1997-08-27' and name = 'James' and age = 1;
```

우리가 수행한 query의 조건에는 해당 index를 구성하는 모든 column을 사용하였는데,
사용된 column은 birthday 하나입니다.

range scan의 경우,

multi-column index여도 range scan에 사용된 column 이후의 index column은 사용되지 않습니다.

그렇다면 index column의 순서를 변경하면 어떻게 될까요?

```
ALTER TABLE `user`
DROP INDEX `idx_user_birthday_name__age` ,
ADD INDEX `idx_user_age_birthday_name` (`age` ASC, `birthday` ASC, `name` ASC) VISIBLE;
```

multi-column의 순서를 birthday, name, age에서
age, birthday, name으로 변경한 뒤, 위와 동일한 query를 수행해보겠습니다.

```
explain format=JSON select * from user where birthday >= '1997-08-27' and name = 'James' and age = 1;
```

해당 query의 결과는 다음과 같습니다.


```
{
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "1016.43"
    },
    "table": {
      "table_name": "user",
      "access_type": "range",
      "possible_keys": [
        "idx_user_age_birthday_name"
      ],
      "key": "idx_user_age_birthday_name",
      "used_key_parts": [
        "age",
        "birthday"
      ],
      "key_length": "189",
      "rows_examined_per_scan": 4544,
      "rows_produced_per_join": 454,
      "filtered": "10.00",
      "using_index": true,
      "cost_info": {
        "read_cost": "970.99",
        "eval_cost": "45.44",
        "prefix_cost": "1016.43",
        "data_read_per_join": "88K"
      },
      "used_columns": [
        "id",
        "name",
        "age",
        "birthday"
      ],
      "attached_condition": "((`example`.`user`.`age` = 1) and (`example`.`user`.`name` = 'James') and (`example`.`us
er`.`birthday` >= DATE'1997-08-27'))"
    }
  }
}
```

used_key_parts를 보면, 이전과는 다르게 **age와 birthday column를 사용**한 것을 알 수 있습니다.

위의 실험들을 통해 우리가 알 수 있는 것은
multi-column index를 설계함에 있어, cardinality 뿐만 아니라
해당 index의 용도에 대해서 고민해볼 필요가 있다는 것
이라고 생각합니다.

multi-column index의 첫번째 column이 range scan에만 사용될 경우,
나머지 column들은 전혀 활용되지 못하기 때문입니다.

그럼 여기까지 multi-column index range scan에 대한 글이었습니다.
감사합니다.