DB 동시성 관련

~~의식의 흐름대로 작성함~~

확인 방법 가이드

백명규 지음

01 MySQL Docker로 사용하기

```
PS C:\Users\qorau> docker pull mysql

PS C:\Users\qorau> docker run -d -p 3306:3306 -e MYSQL_ROOT_PASSWORD=1234 -name mysql mysql

PS C:\Users\qorau> docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS         PORTS                                NAMES
137e0219c573   mysql     "docker-entrypoint.s…"   21 hours ago   Up 7 seconds   0.0.0.0:3306->3306/tcp, 33060/tcp   mysql

PS C:\Users\qorau> docker exec -it mysql bash
bash-5.1# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 9.0.1 MySQL Community Server - GPL

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

02 잠금 관련 문제 설명

```
mysql> CREATE TABLE `tb_test_user_info` (
    ->   `id` int NOT NULL,
    ->   `emp_no` int DEFAULT NULL,
    ->   `first_name` varchar(10) COLLATE utf8mb4_general_ci DEFAULT NULL,
    ->   `last_name` varchar(20) COLLATE utf8mb4_general_ci DEFAULT NULL,
    ->   `hire_date` date DEFAULT NULL,
    ->   `count` int,
    ->   PRIMARY KEY (`id`),
    ->   KEY `ix_first_name_last_name` (`first_name`, `last_name`)
    -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

02 잠금 관련 문제 설명

```
mysql> INSERT INTO `tb_test_user_info` (id, emp_no, first_name, last_name, hire_date, count) VALUES
    -> (1,10001,'Georgi','Facello','1985-11-21',0),
    -> (2,10002,'Mary','Simmel','1986-08-28',0),
    -> (3,10003,'Mary','Bamford','1986-12-01',0),
    -> (4,10004,'Mary','Koblick','1989-09-12',0),
    -> (5,10005,'Mary','Maliniak','1989-06-02',0),
    -> (6,10006,'Mary','Preusig','1989-02-10',0),
    -> (7,10007,'Mary','Zielinski','1994-09-15',0),
    -> (8,10008,'Saniya','Kalloufi','1985-02-18',0),
    -> (9,10009,'Sumant','Peac','1989-08-24',0),
```

→ 데이터 상태 1: 현재 모든 컬럼은 83개로 모든 사람의 이름은 다름
→ 데이터 상태 2: 33개의 Mary라는 성을 가진 이름이 다른 33명이 있고 나머지는 성이 전부 다 다름.

02 잠금 관련 문제 상황 설명

문제 상황과 의문점 1:

1. 본인은 지금 DATABASE 내에서 저 연습용 유저 정보 테이블을 가져올 때 쿼리를 날리면서 어디가 얼마만큼 잠기는 지 확인하고 싶다면 어떻게 확인을 할 수 있을까?

→ 레코드의 변경 혹은 삭제가 테이블 내의 해당 레코드만 잠글까 아니면 전체 레코드를 잠글까?

2. 무엇보다 어딜 확인해야지 이걸 확인할 수 있을까?

→ MySQL 내의 어떤 변수 내에 이 내용이 저장이 되어 있을까?

02 잠금 관련 문제 상황 설명

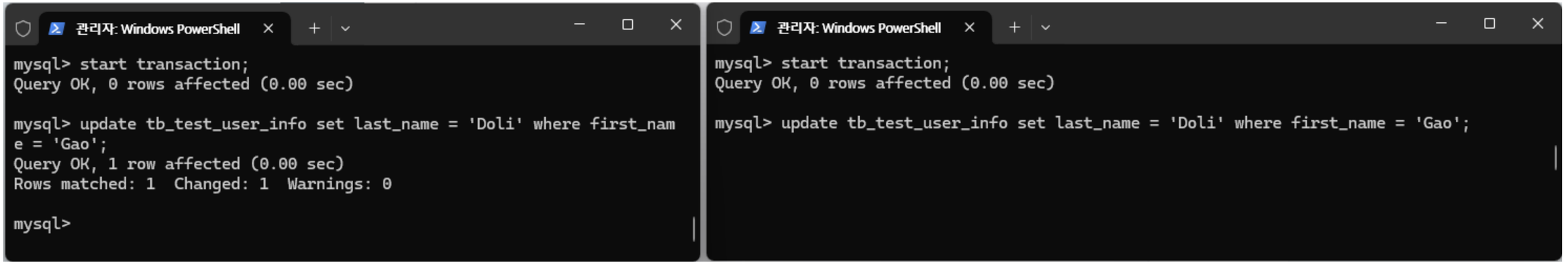
문제 상황과 의문점 2:

3. 인덱스가 조회 속도를 빠르게 해준다고 하는데, 그럼 업데이트 속도는 어떻게 바꿀까?

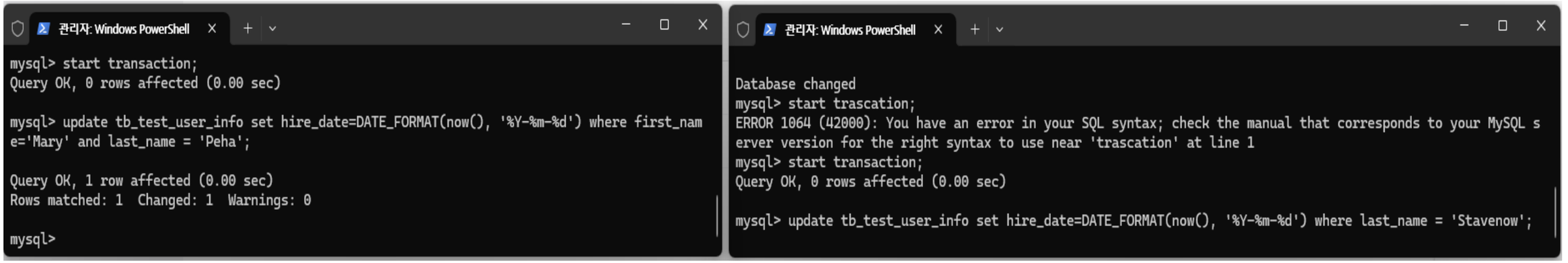
4. 인덱스와 락의 범위에 대해서는 어떻게 동작을 할까?

→ 조건 1: 데이터가 100개일 때, 데이터가 1000개일 때, 데이터가 10000개 일 때
→ 조건 2: INDEX가 없는 컬럼에 대한 변경, INDEX가 최적의 효율을 내지 못하게 설정된 컬럼에 대한 변경, INDEX가 최적의 효율을 내는 환경에서의 컬럼에 대한 변경

03 시도 방법 1(여러 세션을 통한 잠금)

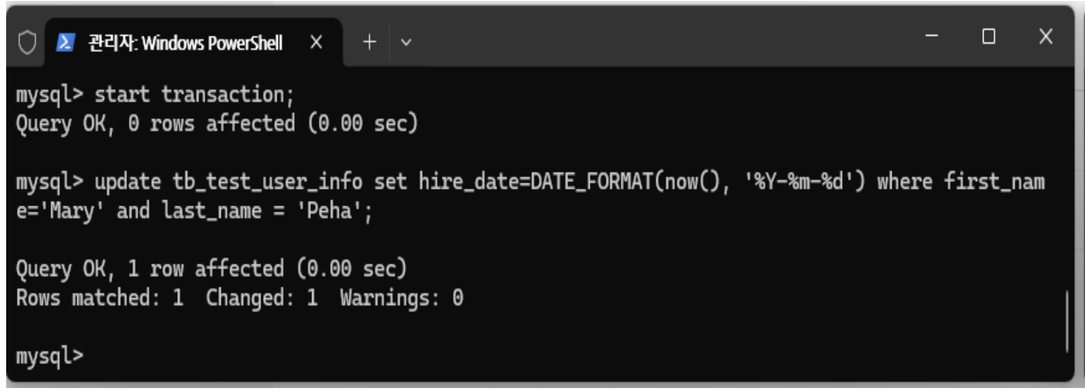


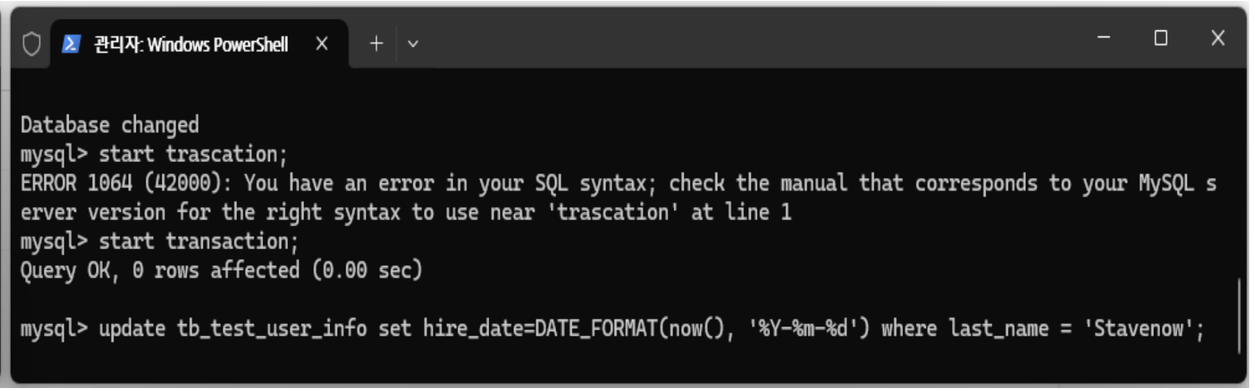
--> 당연히 같은 레코드니까 update에 대해서는 레코드 별 잠금으로 인해 잠기겠죠?



--> 왼쪽이 먼저 시도한 세션인데 분명히 Peha Mary씨의 입사 일을 바꿨는데, 오른쪽의 Stavenow씨의 입사일이 잠금 때문에 바뀌지 않네요?

03 시도 방법 1(여러 세션을 통한 잠금)





--> 왼쪽이 먼저 시도한 세션인데 분명히 Peha Mary씨의 입사일을 바꿨는데, 오른쪽의 Stavenow씨의 입사일이 잠금 때문에 바뀌지 않네요?

위의 방식으로 여러 세션을 통해서 잠금이 일어나는 것을 확인할 수 있지만, 지금 무엇이 원인인지 모르는 문제가 발생합니다.

원인에 대해서 추측이 가능하실까요?

04 시도 방법2 (MySQL 내의 격리 수준 확인과 설정)

```
mysql> show session variables like '%isola%';
+-----------------------+----------------+
| Variable_name         | Value          |
+-----------------------+----------------+
| transaction_isolation | REPEATABLE-READ |
+-----------------------+----------------+
1 row in set (0.01 sec)
```

```
mysql> show global variables like '%isola%';
+-----------------------+----------------+
| Variable_name         | Value          |
+-----------------------+----------------+
| transaction_isolation | REPEATABLE-READ |
+-----------------------+----------------+
1 row in set (0.01 sec)
```

```sql
-- 글로벌 격리 수준 설정
SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;

-- 세션 격리 수준 설정
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

예제랑은 멀지만 혹시나 몰라 넣어봤습니다.

SERIALIZABLE이랑 REPEATABLE READ는 next key lock 혹은 gap lock을 통해 다른 세션의 주변 레코드 변경을 막기 때문입니다.

04 시도 방법2 (MySQL 내의 격리 수준 확인과 설정)

참고로 꼭 MySQL bash에 안 들어가도 어플리케이션에서 메서드 단위 혹은 글로벌 변경이 가능합니다.

```java
@Transactional(isolation = Isolation.SERIALIZABLE)  qoraudrb
public void decrease(Long id, Long quantity) {
    // Stock 조회
    // 재고 감소시킨 뒤
    // 갱신된 값을 저장
    Stock stock = stockRepository.findById(id).orElseThrow();
    stock.decrease(quantity);

    stockRepository.saveAndFlush(stock);
}
```

```yaml
spring:
 transaction:
   default:
     isolation: SERIALIZABLE
```

05 시도 방법3 (EXPLAIN, PERFORMANCE SCHEMA)

MySQL bash 환경에서의 EXPLAIN 함수를 통한 실행 과정 살펴 보기:
EXPLAIN UPDATE tb_test_user_info SET count = count + 1 WHERE first_name = ‘Mary’ AND last_name = ‘Simmel’;

```
mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> EXPLAIN UPDATE tb_test_user_info SET count = count + 1 WHERE first_name = 'Mary' AND last_name = 'Simmel';
+----+-------------+------------------+------------+-------+-------------------------+-------------------------+---------+-------------+------+----------+-------------+
| id | select_type | table            | partitions | type  | possible_keys           | key                     | key_len | ref         | rows | filtered | Extra       |
+----+-------------+------------------+------------+-------+-------------------------+-------------------------+---------+-------------+------+----------+-------------+
|  1 | UPDATE      | tb_test_user_info | NULL       | range | ix_first_name_last_name | ix_first_name_last_name | 126     | const,const |    1 |   100.00 | Using where |
+----+-------------+------------------+------------+-------+-------------------------+-------------------------+---------+-------------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

일어난 연산 타입

선택된 동작 방식

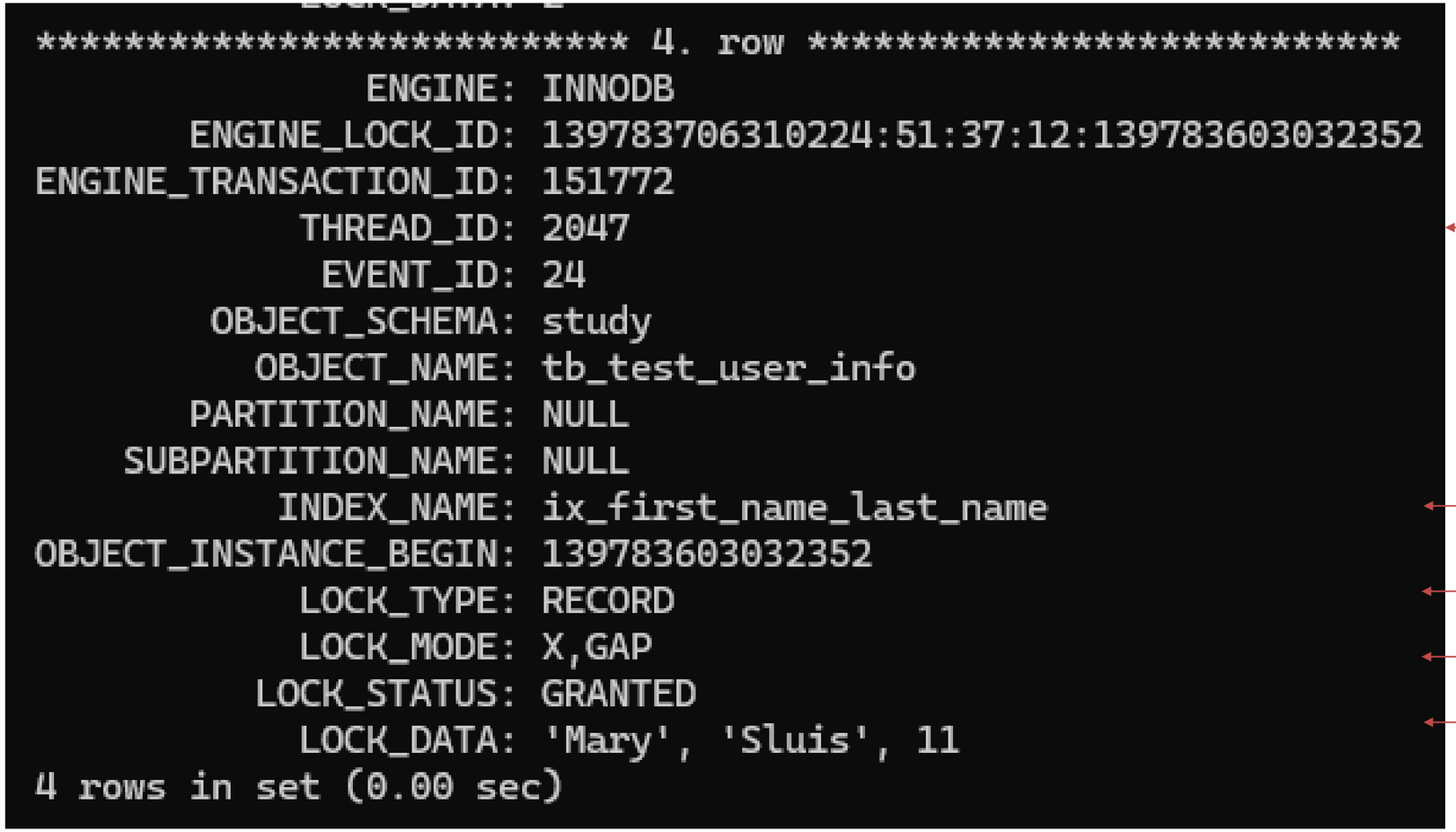
사용된 인덱스

조회 컬럼 수

“SELECT * from performance_schema.data_locks\G” 명령어를 통해 Lock 상태 체크가 가능

```
mysql> UPDATE tb_test_user_info SET count = count + 1 WHERE first_name = 'Mary' AND last_name = 'Simmel';
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from performance_schema.data_locks\G
*************************** 1. row ***************************
               ENGINE: INNODB
       ENGINE_LOCK_ID: 139783706310224:1117:139783603034576
ENGINE_TRANSACTION_ID: 151772
            THREAD_ID: 2047
             EVENT_ID: 24
        OBJECT_SCHEMA: study
          OBJECT_NAME: tb_test_user_info
       PARTITION_NAME: NULL
    SUBPARTITION_NAME: NULL
           INDEX_NAME: NULL
OBJECT_INSTANCE_BEGIN: 139783603034576
            LOCK_TYPE: TABLE
            LOCK_MODE: IX
          LOCK_STATUS: GRANTED
            LOCK_DATA: NULL
```


*************************** 4. row ***************************
ENGINE: INNODB
ENGINE_LOCK_ID: 139783706310224:51:37:12:139783603032352
ENGINE_TRANSACTION_ID: 151772
THREAD_ID: 2047
EVENT_ID: 24
OBJECT_SCHEMA: study
OBJECT_NAME: tb_test_user_info
PARTITION_NAME: NULL
SUBPARTITION_NAME: NULL
INDEX_NAME: ix_first_name_last_name
OBJECT_INSTANCE_BEGIN: 139783603032352
LOCK_TYPE: RECORD
LOCK_MODE: X,GAP
LOCK_STATUS: GRANTED
LOCK_DATA: 'Mary', 'Sluis', 11
4 rows in set (0.00 sec)
Lock 점유 세션 쓰레드 번호
적용된 인덱스
Lock 점유 범위
점유 락 종류
점유 중인 데이터

```
LOCK_TYPE: RECORD
LOCK_MODE: X,GAP
```

이 부분이 읽을 때 가장 모르시지 않을까 생각이 듭니다.

공유 잠금(S): 특정 레코드를 읽을 때 사용, Shared Lock 사용 시 여러 트랜잭션이 동시에 읽을 수 있습니다.
대신, 수정이 불가능합니다.

배타적 잠금(X): 특정 레코드를 수정할 때 사용, Exclusive Lock을 얻은 트랜잭션은 레코드를 읽고 수정이 가능하지만 다른 트랜잭션은 해당 레코드를 읽거나 수정 불가능

의도 잠금:
Intention Shared Lock(IS) : Shared Lock을 획득하려는 Lock으로 다른 트랜잭션이 Exclusive Lock 걸었을 시 걸 수 없다.

Intention Exclusive Lock(IX): Exclusive Lock을 획득할 때로 다른 트랜잭션이 Shared Lock이나 Exclusive Lock을 걸면 걸 수 없다.

Shared and Exclusive(SIX): 여러 개의 레코드를 동시에 락을 걸 때 사용되어 여러 레코드를 수정하고 읽을 때 사용됨.

06 부하 툴 없이 스프링 애플리케이션을 통해 실험하기

```
mysql> CREATE TABLE `tb_test_user_info` (
    ->   `id` int NOT NULL,
    ->   `emp_no` int DEFAULT NULL,
    ->   `first_name` varchar(10) COLLATE utf8mb4_general_ci DEFAULT NULL,
    ->   `last_name` varchar(20) COLLATE utf8mb4_general_ci DEFAULT NULL,
    ->   `hire_date` date DEFAULT NULL,
    ->   PRIMARY KEY (`id`),
    ->   KEY `ix_first_name` (`first_name`)
    -> ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci
    -> ;
Query OK, 0 rows affected (0.03 sec)
```

JPA 변환!

```
@Entity  7 usages
@ToString
public class TbTestUserInfo {

    @Id
    private Long id;

    private int empNo;
    private String firstName;
    private String lastName;
    private LocalDate hireDate;
    private int count;
```

06 부하 툴 없이 스프링 애플리케이션을 통해 실험하기

데이터 리포지토리 계층 UPDATE 코드 작성

```
@Repository  3 usages
public interface TbTestUserInfoRepository extends JpaRepository<TbTestUserInfo, Long> {

@Modifying  4 usages
@Transactional
@Query(value = "UPDATE tb_test_user_info SET count = count + 1 WHERE first_name = :firstName AND last_name = :lastName", nativeQuery = true)
int updateCountByFirstNameAndLastName(@Param("firstName") String firstName, @Param("lastName") String lastName);
```

더미데이터 삽입용 테스트 코드 방식

```
@Test
void 더미데이터_삽입_10000() {
    for (int i = 1000; i < 10000; i++) {
        TbTestUserInfo tester = new TbTestUserInfo((long) i+100, 10100 + i, "test" + i, "test" + i, LocalDate.now(), 0);
        tbTestUserInfoRepository.save(tester);    }
}
```

06 부하 툴 없이 스프링 애플리케이션을 통해 실험하기

```java
private String[] lastNames = {"Simmel", "Bamford", "Koblick", "Maliniak", "Preusig"};  2 usages
@Test
void 인덱스X_잠금이_많은_환경_update_1000() throws InterruptedException {
    int threadCount = 1000;
    ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 32);
    CountDownLatch latch = new CountDownLatch(threadCount);

    for (int i = 0; i < threadCount; i++) {
        final int idx = i;
        executorService.submit(() -> {
            try {
                tbTestUserInfoRepository.updateCountByFirstNameAndLastName("Mary", lastNames[idx%5]);
            } finally {
                latch.countDown();
            }
        });
    }
    latch.await();
}
```

06 부하 툴 없이 스프링 애플리케이션을 통해 실험하기

```
private String[] firstNames_2 = {"Georgi", "Saniya", "Sumant", "Duangkaew", "Patricio"};  2 usages
private String[] lastNames_2 = {"Facello", "Kalloufi", "Peac", "Piveteau", "Bridgland"};  2 usages

@Test
void 인덱스X_잠금이_적은_환경_update_1000() throws InterruptedException {
    int threadCount = 1000;
    ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 32);
    CountDownLatch latch = new CountDownLatch(threadCount);

    for (int i = 0; i < threadCount; i++) {
        final int idx = i;
        executorService.submit(() -> {
            try {
                tbTestUserInfoRepository.updateCountByFirstNameAndLastName(firstNames_2[idx%5], lastNames_2[idx%5]);
            } finally {
                latch.countDown();
            }
        });
    }
    latch.await();
}
```

인덱스가 **PK만** 있는 경우

```
mysql> show index from tb_test_user_info;
+----------------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
| Table          | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+----------------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
| tb_test_user_info |          0 | PRIMARY  |            1 | id          | A         |       10180 |     NULL |   NULL |      | BTREE      |         |               | YES     | NULL       |
+----------------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
1 row in set (0.00 sec)
```

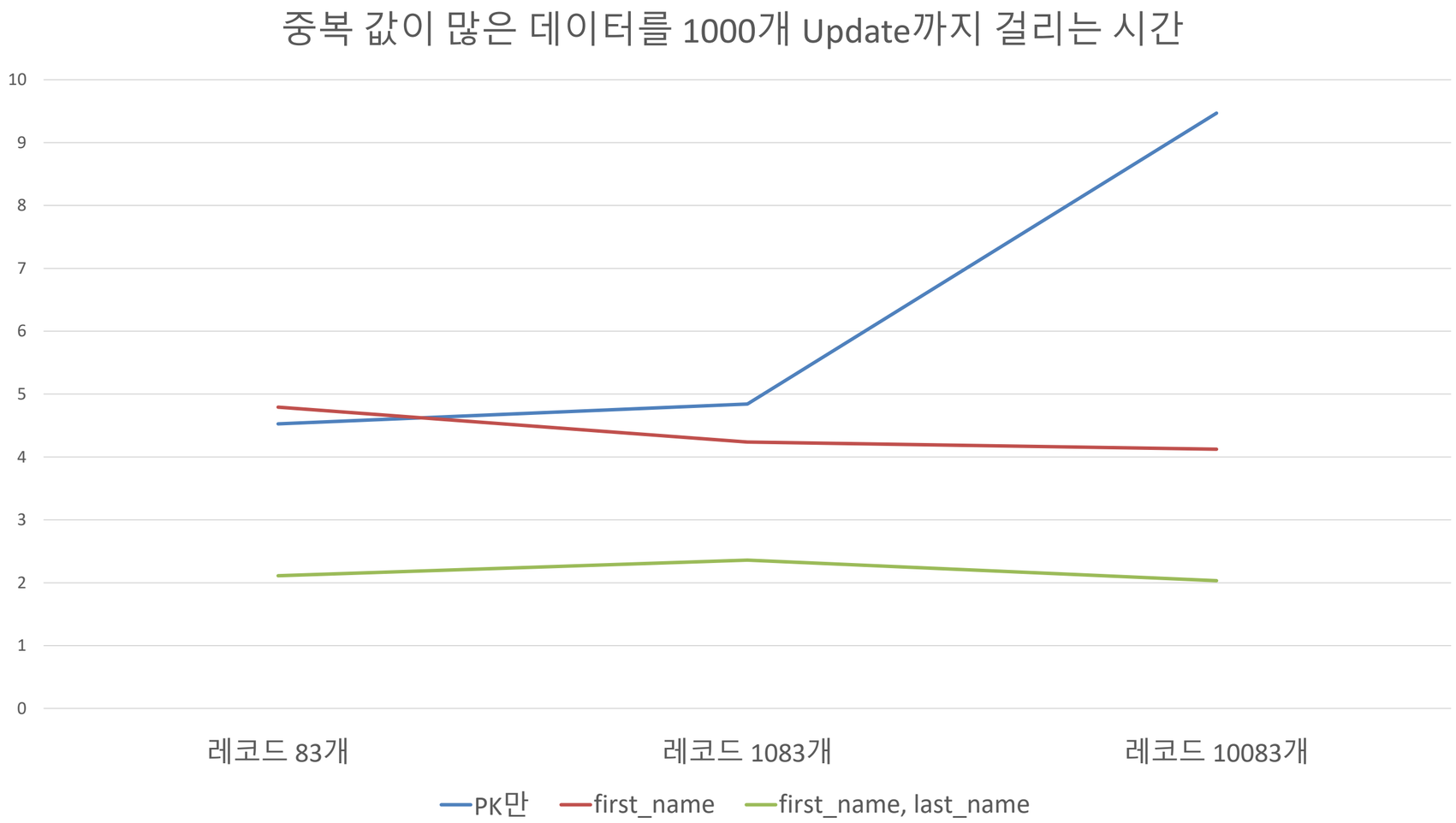
first_name을 대상으로 인덱스 생성

```
mysql> show INDEX FROM tb_test_user_info;
+----------------+------------+---------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
| Table          | Non_unique | Key_name      | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+----------------+------------+---------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
| tb_test_user_info |          0 | PRIMARY       |            1 | id          | A         |          83 |     NULL |   NULL |      | BTREE      |         |               | YES     | NULL       |
| tb_test_user_info |          1 | ix_first_name |            1 | first_name  | A         |          50 |     NULL |   NULL | YES  | BTREE      |         |               | YES     | NULL       |
+----------------+------------+---------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
```

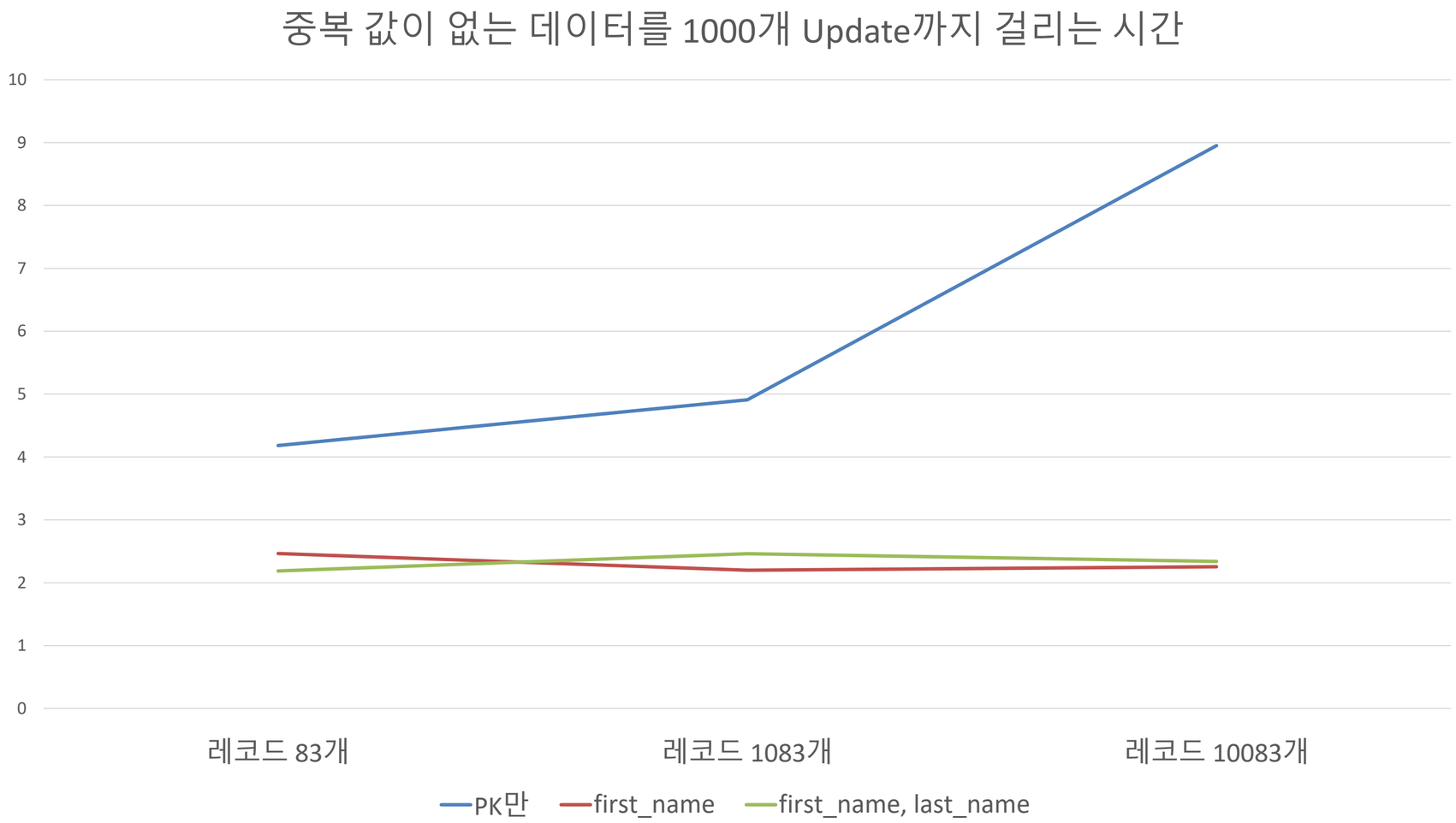
first_namer과 last_name을 대상으로 인덱스 생성

```
mysql> show index from tb_test_user_info;
+----------------+------------+----------------------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
| Table          | Non_unique | Key_name                   | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+----------------+------------+----------------------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
| tb_test_user_info |          0 | PRIMARY                    |            1 | id          | A         |       10180 |     NULL |   NULL |      | BTREE      |         |               | YES     | NULL       |
| tb_test_user_info |          1 | ix_first_name_last_name    |            1 | first_name  | A         |       10050 |     NULL |   NULL | YES  | BTREE      |         |               | YES     | NULL       |
| tb_test_user_info |          1 | ix_first_name_last_name    |            2 | last_name   | A         |       10082 |     NULL |   NULL | YES  | BTREE      |         |               | YES     | NULL       |
+----------------+------------+----------------------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
3 rows in set (0.02 sec)
```

07 측정 결과


중복 값이 많은 데이터를 1000개 Update까지 걸리는 시간
10
9
8
7
6
5
4
3
2
1
0
레코드 83개
레코드 1083개
레코드 10083개
PK만
first_name
first_name, last_name

07 측정 결과


중복 값이 없는 데이터를 1000개 Update까지 걸리는 시간
10
9
8
7
6
5
4
3
2
1
0
레코드 83개
레코드 1083개
레코드 10083개
PK만
first_name
first_name, last_name

07 분석 결과

왜 인덱스를 걸었는데 Update 쿼리에서 더 빠르게 동작을 할까?

1. Update 쿼리를 날렸을 때, 테이블에서 삭제하는 비용 총 계는 다음과 같다

인덱스가 존재하는 경우:
인덱스를 통한 레코드 탐색 비용 + 세컨더리 인덱스 변경 비용 + 클러스터링 데이터 변경 비용
인덱스가 존재하지 않는 경우:
풀 테이블 스캔을 통한 레코드 탐색 비용 + 클러스터링 데이터 변경 비용

→ 이번 케이스는 인덱스를 통해 스캔함으로써 줄인 스캔 비용이 인덱스 변경 관련 비용보다 훨씬 크기 때문에 인덱스가 update를 날린 경우 훨씬 빨랐음.

07 분석 결과

왜 인덱스를 걸었는데 Update 쿼리에서 더 빠르게 동작을 할까?

2. Update 쿼리를 날렸을 때, **레코드가 잠기는 범위가 훨씬 좁다**.

인덱스가 존재하는 경우:
인덱스 관련 레코드 부분만 잠김

인덱스가 존재하지 않는 경우:
풀 테이블 스캔을 하면서 테이블 전체 레코드를 잠궈 버림.

테이블이 커질수록 인덱스를 제대로 걸지 않아버리면 풀테이블 스캔 쪽에서의 잠금이 넓은 범위로 걸리기 때문에 성능 저하가 일어날 수 있음.

→ 인덱스를 걸면 무조건 검색이 빨라지는 것도 아니지만, 인덱스를 걸었다고 무조건 테이블 변경 비용이 늘어나는 것이 아님!

07 분석 결과

```
*************************** 1088. row ***************************
               ENGINE: INNODB
       ENGINE_LOCK_ID: 139783706309416:51:13:25:139783603027056
ENGINE_TRANSACTION_ID: 211007
            THREAD_ID: 291
             EVENT_ID: 30
        OBJECT_SCHEMA: study
          OBJECT_NAME: tb_test_user_info
       PARTITION_NAME: NULL
    SUBPARTITION_NAME: NULL
           INDEX_NAME: PRIMARY
OBJECT_INSTANCE_BEGIN: 139783603027056
            LOCK_TYPE: RECORD
            LOCK_MODE: X
          LOCK_STATUS: GRANTED
            LOCK_DATA: 1098
*************************** 1089. row ***************************
               ENGINE: INNODB
       ENGINE_LOCK_ID: 139783706309416:51:13:26:139783603027056
ENGINE_TRANSACTION_ID: 211007
            THREAD_ID: 291
             EVENT_ID: 30
        OBJECT_SCHEMA: study
          OBJECT_NAME: tb_test_user_info
       PARTITION_NAME: NULL
    SUBPARTITION_NAME: NULL
           INDEX_NAME: PRIMARY
OBJECT_INSTANCE_BEGIN: 139783603027056
            LOCK_TYPE: RECORD
            LOCK_MODE: X
          LOCK_STATUS: GRANTED
            LOCK_DATA: 1099
1089 rows in set (0.01 sec)
```

```
*************************** 3. row ***************************
               ENGINE: INNODB
       ENGINE_LOCK_ID: 139783706309416:51:6:3:139783603026024
ENGINE_TRANSACTION_ID: 211015
            THREAD_ID: 291
             EVENT_ID: 38
        OBJECT_SCHEMA: study
          OBJECT_NAME: tb_test_user_info
       PARTITION_NAME: NULL
    SUBPARTITION_NAME: NULL
           INDEX_NAME: PRIMARY
OBJECT_INSTANCE_BEGIN: 139783603026024
            LOCK_TYPE: RECORD
            LOCK_MODE: X,REC_NOT_GAP
          LOCK_STATUS: GRANTED
            LOCK_DATA: 2
*************************** 4. row ***************************
               ENGINE: INNODB
       ENGINE_LOCK_ID: 139783706309416:51:9:59:139783603026368
ENGINE_TRANSACTION_ID: 211015
            THREAD_ID: 291
             EVENT_ID: 38
        OBJECT_SCHEMA: study
          OBJECT_NAME: tb_test_user_info
       PARTITION_NAME: NULL
    SUBPARTITION_NAME: NULL
           INDEX_NAME: ix_first_name_last_name
OBJECT_INSTANCE_BEGIN: 139783603026368
            LOCK_TYPE: RECORD
            LOCK_MODE: X,GAP
          LOCK_STATUS: GRANTED
            LOCK_DATA: 'Mary', 'Sluis', 11
4 rows in set (0.00 sec)
```

Figure 1. 별도의 인덱스가 없는 경우(좌), 인덱스가 단일 레코드만 찾을 수 있게 설정된 경우(우)

07 분석 결과

인덱스를 잘 걸어야 하는 이유

1. 인덱스를 제대로 걸지 않게 되면 인덱스를 통한 레코드 찾는 비용이 증가하게 된다.

인덱스가 first_name 컬럼에만 걸렸을 때

인덱스를 스캔하는 과정에서 **first_name**만으로 해당 레코드를 명확하게 찾을 수 없고 **index**를 필터링 조건으로 사용할 수 없어 물리적 데이터 접근이 발생!

→ 인덱스가 있음에도 데이터를 찾는 비용 증가

→ 이번 확인을 통한 교훈은 인덱스를 통해 필터링해서 필요한 데이터 **1**개에만 접근할 수 있도록 **select** 쿼리 뿐만 아니라 그 이외의 **DML**에 대해서도 신경을 써야한다!

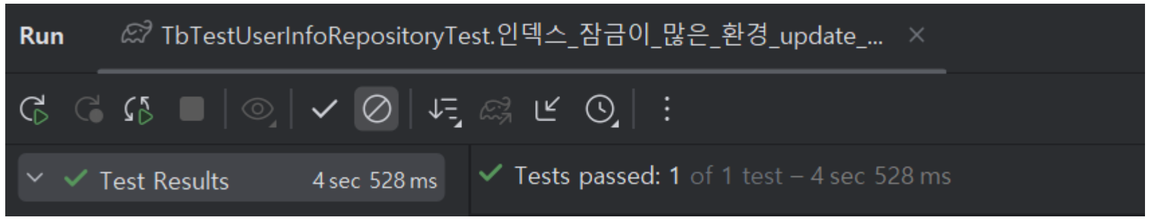
07 분석 결과

```
mysql> EXPLAIN UPDATE tb_test_user_info SET count = count + 1 WHERE first_name = 'Mary' AND last_name = 'Simmel';
+----+-------------+--------------------+------------+-------+---------------+---------------+---------+-------+------+----------+-------------+
| id | select_type | table              | partitions | type  | possible_keys | key           | key_len | ref   | rows | filtered | Extra       |
+----+-------------+--------------------+------------+-------+---------------+---------------+---------+-------+------+----------+-------------+
|  1 | UPDATE      | tb_test_user_info | NULL       | range | ix_first_name | ix_first_name | 43      | const |   33 |   100.00 | Using where |
+----+-------------+--------------------+------------+-------+---------------+---------------+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

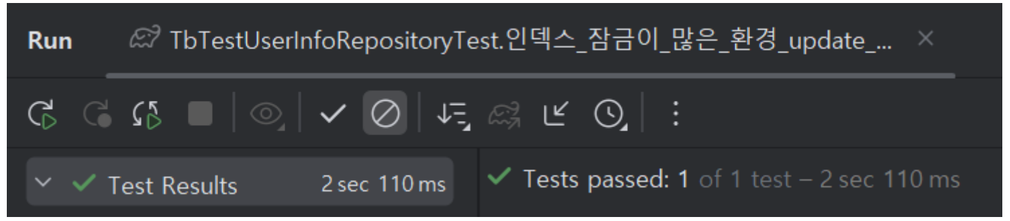
Figure 2. 레코드 1083개 일 때, first_name만 인덱스가 있는 환경에서 실행 계획

01 실험 데이터 목록

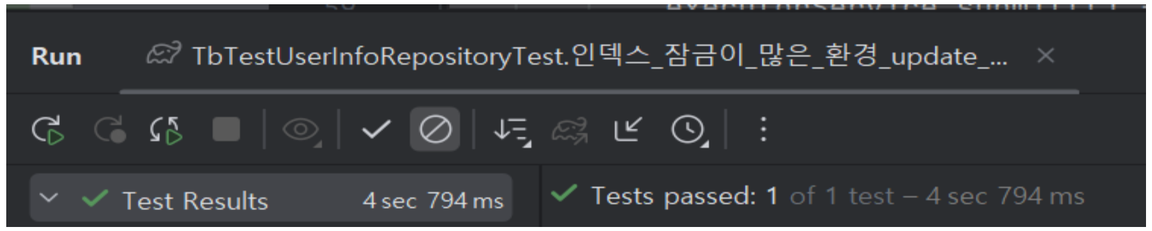
레코드 83, 인덱스 X, Mary 변화



레코드 83, 인덱스 first name, last name에만, Mary 변화

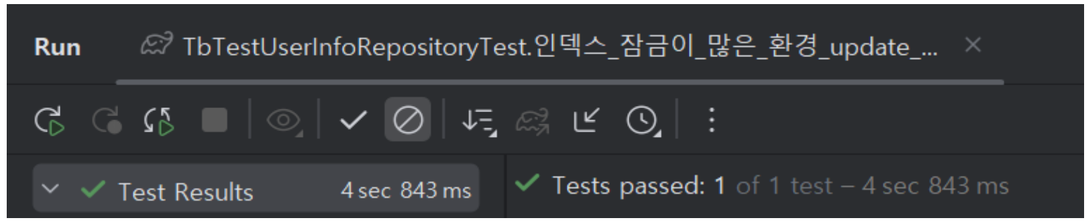


레코드 83, 인덱스 first name에만, Mary 변화

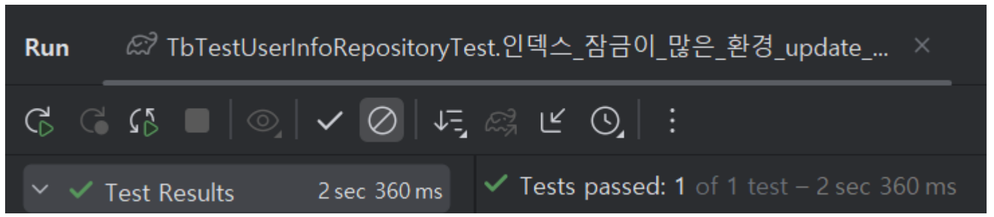


01 실험 데이터 목록

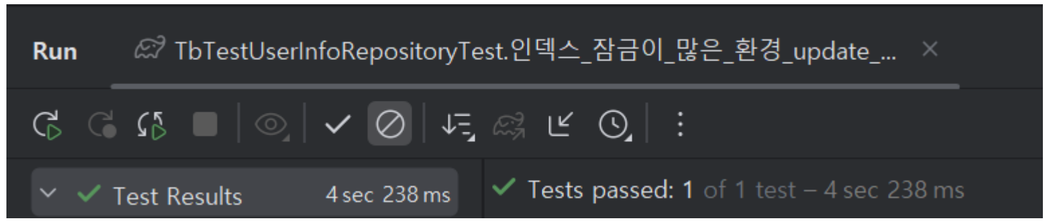
레코드 1083, 인덱스 X, Mary 변화



레코드 1083, 인덱스 first name, last name에만, Mary 변화

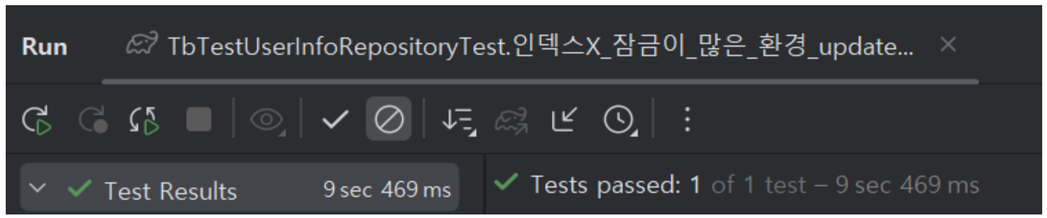


레코드 1083, 인덱스 first name에만, Mary 변화

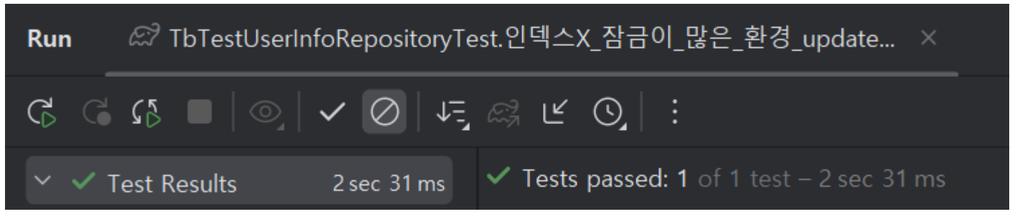


01 실험 데이터 목록

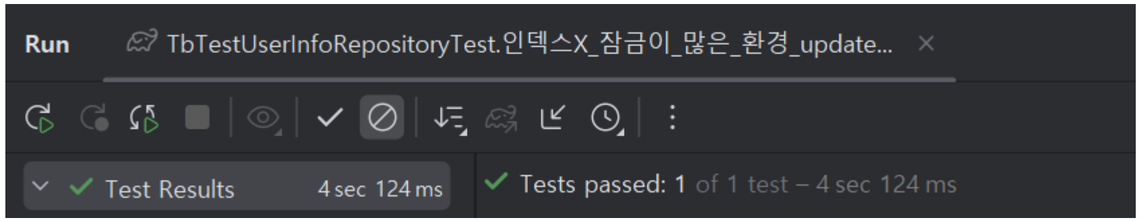
레코드 10083, 인덱스 X, Mary 변화



레코드 10083, 인덱스 first name, last name에만, Mary 변화

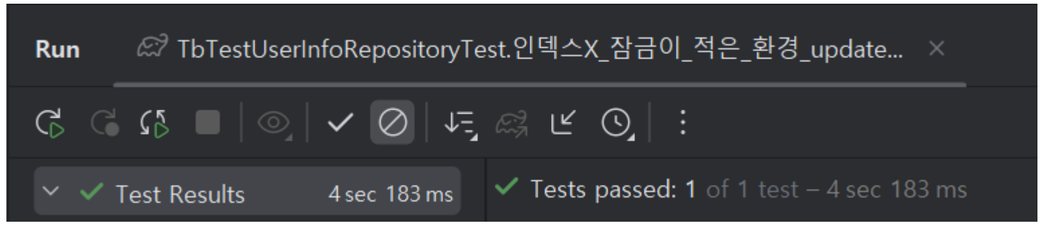


레코드 10083, 인덱스 first name에만, Mary 변화

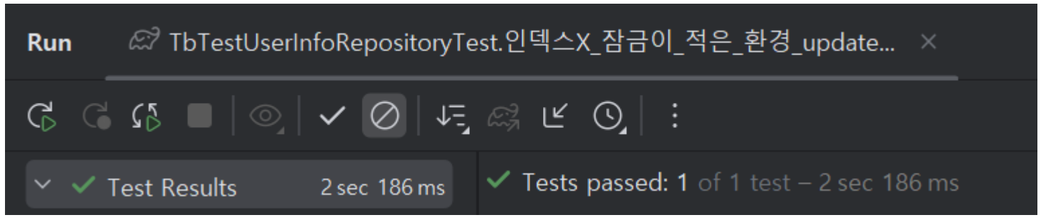


01 실험 데이터 목록

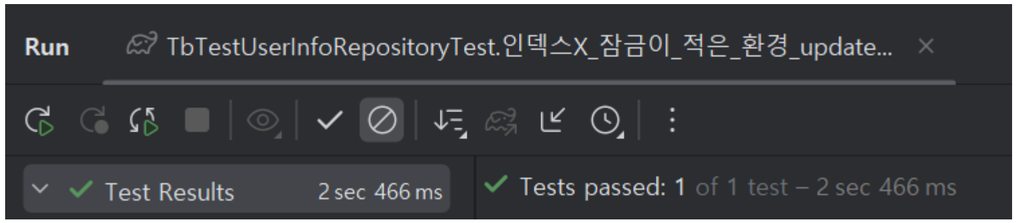
레코드 83, 인덱스 X, Georgi 변화



레코드 83, 인덱스 first name, last name에만, Georgi변화

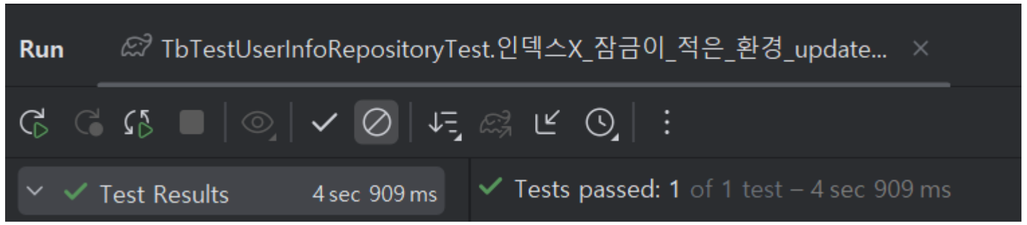


레코드 83, 인덱스 first name에만, Georgi 변화

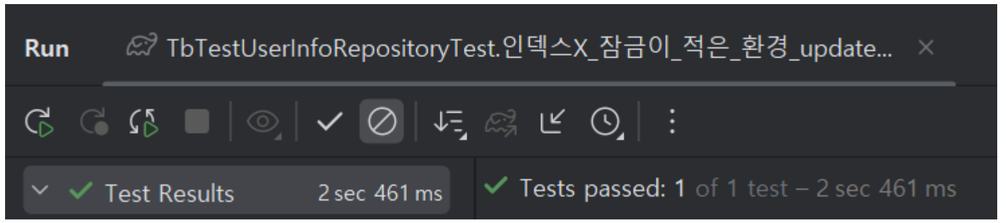


01 실험 데이터 목록

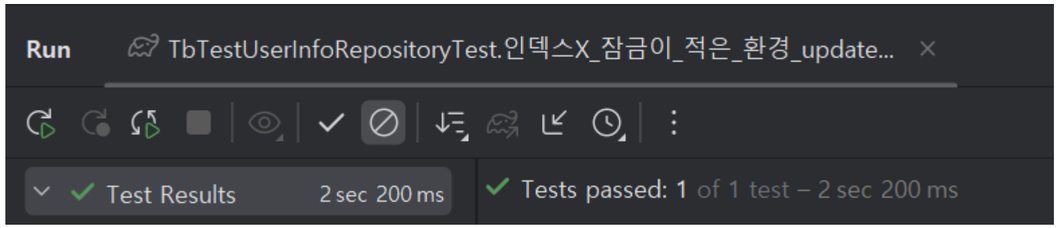
레코드 1083, 인덱스 X, Georgi 변화



레코드 1083, 인덱스 first name, last name에만, Georgi 변화

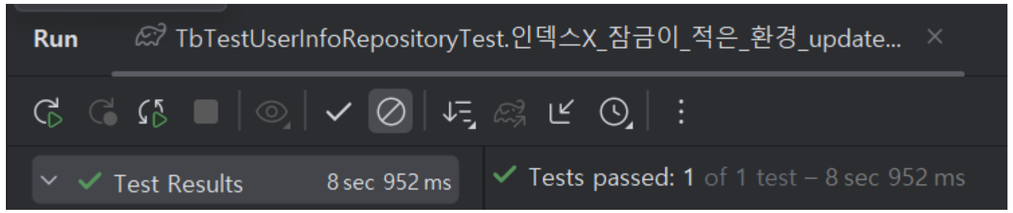


레코드 1083, 인덱스 first name에만, Georgi 변화

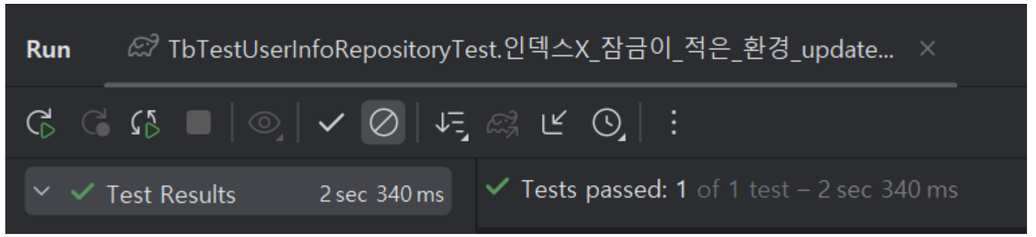


01 실험 데이터 목록

레코드 10083, 인덱스 X, Mary 변화



레코드 10083, 인덱스 first name, last name에만, Mary 변화



레코드 10083, 인덱스 first name에만, Mary 변화