

실리콘밸리에서 날아온 데이터베이스

5. MySQL 고급기능 살펴보기

keeyonghan@hotmail.com

한기용

Harmonize, Inc

Contents

1. 트랜잭션 소개
2. 트랜잭션 실습 (SQL과 자바)
3. View 소개와 실습
4. Stored Procedure, Trigger 소개와 실습
5. 성능 튜닝: Explain SQL과 Index 튜닝과 실습
6. 다음 스텝과 맺음말

트랜잭션 소개

테이블 내용을 변경하는 SQL들이 연달아 실행되며
이것들이 마치 하나의 SQL처럼 다 같이 성공하던지 아니면
실패해야 한다면 트랜잭션의 사용이 필수!

◆ 트랜잭션이란? (1)

- ❖ Atomic하게 실행되어야 하는 SQL들을 묶어서 하나의 작업처럼 처리하는 방법
 - 이는 DDL이나 DML 중 레코드를 수정/추가/삭제한 것에만 의미가 있음.
 - SELECT에는 트랜잭션을 사용할 이유가 없음
 - BEGIN과 END 혹은 BEGIN과 COMMIT 사이에 해당 SQL들을 사용
 - ROLLBACK

◆ 트랜잭션이란? (2)

❖ 은행 계좌 이체가 아주 좋은 예

- 계좌 이체: 인출과 입금의 두 과정으로 이뤄짐
- 만일 인출은 성공했는데 입금이 실패한다면?
- 이 두 과정은 동시에 성공하던지 실패해야함 -> Atomic하다는 의미
- 이런 과정들을 트랜잭션으로 묶어주어야함
- 조회(SELECT)만 한다면 이는 트랜잭션으로 묶일 이유가 없음

◆ 트랜잭션이란? (3)

BEGIN; -- START TRANSACTION

A의 계좌로부터 인출;
B의 계좌로 입금;

END; -- COMMIT

이 명령어들은 마치 하나의
명령어처럼 처리됨. 다
성공하던지 다 실패하던지 둘중의
하나가 됨

- BEGIN과 START TRANSACTION은 같은 의미
- END와 COMMIT은 동일
- 만일 BEGIN 전의 상태로 돌아가고 싶다면 ROLLBACK 실행
- 이 동작은 autocommit 모드에 따라 달라짐!

◆ 트랜잭션 커밋 모드: autocommit (1)

❖ autocommit = True

- 모든 레코드 수정/삭제/추가 작업이 기본적으로 바로 데이터베이스에 쓰여짐. 이를 커밋(Commit)된다고 함.
- 만일 특정 작업을 트랜잭션으로 묶고 싶다면 **BEGIN**과 **END(COMMIT)/ROLLBACK**으로 처리

❖ autocommit = False

- 모든 레코드 수정/삭제/추가 작업이 **COMMIT**이 호출될 때까지 커밋되지 않음
- 즉 명시적으로 커밋을 해야함
- **ROLLBACK**이 호출되면 앞서 작업들이 무시됨

◆ 트랜잭션 커밋 모드: autocommit (2)

- ❖ 이는 SQL 클라이언트/라이브러리에 따라 달라짐
 - MySQL Workbench 기본은 autocommit이 True
 - 확인 방법: `SHOW VARIABLES LIKE 'AUTOCOMMIT';`
 - `SET autocommit=0` (혹은 1)의 실행으로 변경가능

◆ autocommit 실습

-- 아래 테스트 테이블 이름에서 **keeyong_** 대신에 각자 영문 이름을 사용할 것

-- test 데이터베이스는 guest 계정도 쓰기 권한이 있음

```
DROP TABLE IF EXISTS test.keeyong_name_gender;
```

```
CREATE TABLE test.keeyong_name_gender (
```

```
    name varchar(16) NOT NULL,
```

```
    gender enum('Male','Female') default NULL
```

```
);
```

```
INSERT INTO test.keeyong_name_gender VALUES('Keeyong', 'Male');
```

```
INSERT INTO test.keeyong_name_gender VALUES('Jane', 'Female');
```

```
INSERT INTO test.keeyong_name_gender VALUES('Unknown');
```

```
INSERT INTO test.keeyong_name_gender VALUES('Keeyong2', 'Male2');
```

◆ autocommit = True

```
SHOW VARIABLES LIKE 'AUTOCOMMIT';  
-- SET autocommit=1;  
SELECT * FROM test.keeyong_name_gender;  
  
BEGIN;  
DELETE FROM test.keeyong_name_gender;  
INSERT INTO test.keeyong_name_gender VALUES ('Kevin', 'Male');  
ROLLBACK;  
  
SELECT * FROM test.keeyong_name_gender;
```

◆ autocommit = False

```
SET autocommit=0;  
SHOW VARIABLES LIKE 'AUTOCOMMIT';
```

```
SELECT * FROM test.keeyong_name_gender;  
-- BEGIN이 없음  
DELETE FROM test.keeyong_name_gender;  
INSERT INTO test.keeyong_name_gender VALUES ('Kevin', 'Male');  
ROLLBACK;
```

```
SELECT * FROM test.keeyong_name_gender;
```

◆ DELETE FROM vs. TRUNCATE

❖ DELETE FROM table_name (not DELETE * FROM)

- 테이블에서 모든 레코드를 삭제
- vs. DROP TABLE table_name
- WHERE 사용해 특정 레코드만 삭제 가능:
 - DELETE FROM raw_data.user_session_channel WHERE channel = 'Google';

❖ TRUNCATE table_name도 테이블에서 모든 레코드를 삭제

- DELETE FROM은 속도가 느림
- TRUNCATE이 전체 테이블의 내용 삭제시에는 여러모로 유리
- 하지만 두가지 단점이 존재
 - TRUNCATE는 WHERE을 지원하지 않음
 - TRUNCATE는 Transaction을 지원하지 않음



트랜잭션 실습

SQL과 자바로 트랜잭션 실습을 해보기

◆ 실습

❖ MySQL Workbench로 실습

- [실습 SQL 링크](#)

❖ 자바 트랜잭션 실습

- https://github.com/keeyong/assorted_code/blob/main/java/jdbc-mysql/Main.java



View 소개와 실습

가상 테이블

◆ View란? (1)

- ❖ 자주 사용하는 **SQL 쿼리 (SELECT)**에 이름을 주고 그 사용을 쉽게 하는 것
 - 이름이 있는 쿼리가 **View**로 데이터베이스단에 저장됨
 - **SELECT** 결과가 테이블로 저장되는 것이 아니라 **View**가 사용될 때마다 **SELECT**가 실행됨
 - 그런 이유로 가상 테이블이라고 부르기도 함 (Virtual Table)
 - **CREATE OR REPLACE VIEW** 뷰이름 **AS SELECT ...**

◆ View란? (2)

- ❖ 예를 들어 아래 **SELECT**를 기본으로 자주 사용한다면

```
SELECT s.id, s.user_id, s.created, s.channel_id, c.channel  
FROM session s  
JOIN channel c ON c.id = s.channel_id;
```

```
CREATE OR REPLACE VIEW test.session_details AS  
SELECT s.id, s.user_id, s.created, s.channel_id, c.channel  
FROM session s  
JOIN channel c ON c.id = s.channel_id;
```

```
SELECT * FROM test.session_details;
```

◆ SQL 실습

❖ MySQL Workbench로 실습

- [View 실습 SQL 링크](#)



Stored Procedure, Trigger 소개와 실습

◆ Stored Procedure란? (1)

- ❖ MySQL 서버단에 저장되는 SQL 쿼리들
 - CREATE PROCEDURE 사용
 - DROP PROCEDURE [IF EXISTS]로 제거
- ❖ 프로그래밍 언어의 함수처럼 인자를 넘기는 것이 가능
- ❖ 리턴되는 값은 레코드들의 집합 (SELECT와 동일)
- ❖ 간단한 분기문(if, case)과 루프(loop)를 통한 프로그램이 가능
- ❖ 디버깅이 힘들고 서버단의 부하를 증가시킨다는 단점 존재

◆ Stored Procedure란? (2)

❖ 정의 문법

DELIMITER //

CREATE PROCEDURE procedure_name(parameter_list)

BEGIN

statements;

END //

DELIMITER ;

❖ 호출 문법

CALL stored_procedure_name(argument_list);

◆ Stored Procedure란? (3)

❖ 정의 예

```
DELIMITER //  
CREATE PROCEDURE return_session_details()  
BEGIN  
    SELECT *  
    FROM test.keeyong_session_details;  
END //  
DELIMITER;
```

❖ 호출 예

```
CALL return_session_details();
```

◆ Stored Procedure란? (4) - IN 파라미터

```
DROP PROCEDURE IF EXISTS return_session_details;
DELIMITER //
CREATE PROCEDURE return_session_details(IN channelName varchar(64))
BEGIN
    SELECT *
    FROM test.keeyong_session_details
    WHERE channel = channelName;
END //
DELIMITER ;

CALL return_session_details('Facebook');
```

◆ Stored Procedure란? (5) - INOUT 파라미터

```
DROP PROCEDURE IF EXISTS return_session_count;
DELIMITER //
CREATE PROCEDURE return_session_count(IN channelName varchar(64), INOUT
totalRecord int)
BEGIN
    SELECT COUNT(1) INTO totalRecord FROM test.keeyong_session_details
    WHERE channel = channelName;
END //
DELIMITER ;

SET @facebook_count = 0
CALL return_session_count('Facebook', @facebook_count);
SELECT @facebook_count;
```


◆ Stored Function이란? (1)

- ❖ 값(Scalar)을 하나 리턴해주는 서버쪽 함수 (특정 데이터베이스 밑에 등록됨)
 - 리턴값은 Deterministic 혹은 Non Deterministic
 - 현재 guest 계정으로 test 데이터베이스 밑에 생성 가능
- ❖ 모든 함수의 인자는 IN 파라미터
- ❖ SQL 안에서 사용가능: Stored Procedure와 가장 다른 차이점
- ❖ CREATE FUNCTION 사용

◆ Stored Function이란? (2)

```
DELIMITER $$
CREATE FUNCTION test.Channel_Type(channel varchar(32))
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
    DECLARE channel_type VARCHAR(20);

    IF channel in ('Facebook', 'Instagram', 'Tiktok') THEN
        SET channel_type = 'Social Network';
    ELSEIF channel in ('Google', 'Naver') THEN
        SET channel_type = 'Search Engine';
    ELSE
        SET channel_type = channel;
    END IF;
    -- return the customer level
    RETURN (channel_type);
END$$
```

Stored Function 호출 예:

```
SELECT channel, test.Channel_Type(channel)
FROM prod.channel;
```

◆ Trigger란? (1)

- ❖ CREATE TRIGGER 명령을 사용
- ❖ INSERT/DELETE/UPDATE 실행 전후에 특정 작업을 수행하는 것이 가능
 - 대상 테이블 지정이 필요
- ❖ NEW/OLD modifier
 - NEW는 INSERT와 UPDATE에서만 사용가능
 - OLD는 DELETE와 UPDATE에서만 사용가능

CREATE TRIGGER 트리거이름

{BEFORE | AFTER} {INSERT | UPDATE| DELETE }

ON table_name FOR EACH ROW

trigger_body;

◆ Trigger란? (2)

❖ 중요 테이블의 경우 감사(audit)가 필요

- 레코드에 변경이 생길 때마다 변경전의 레코드를 저장하는 트리거를 만들어보자

```
CREATE TABLE test.keeyong_name_gender_audit (  
    name varchar(16),  
    gender enum('Male', 'Female'),  
    modified timestamp  
);
```

-- 트리거 정의

```
CREATE TRIGGER test.before_update_keeyong_name_gender  
    BEFORE UPDATE ON test.keeyong_name_gender  
    FOR EACH ROW  
    INSERT INTO test.keeyong_name_gender_audit  
    SET name = OLD.name,  
        gender = OLD.gender,
```

```
-- 트리거 사용 예  
UPDATE test.keeyong_name_gender  
SET name = 'Keeyong'  
WHERE name = 'Keeyong2';  
  
SELECT * FROM test.keeyong_name_gender_audit;
```

◆ SQL 실습

❖ MySQL Workbench로 실습

- [Stored Procedure & Stored Function & Trigger 실습 SQL 링크](#)

성능 튜닝: Explain SQL과 Index 튜닝과 실습

◆ Explain SQL (1)

- ❖ **SELECT/UPDATE/INSERT/DELETE** 등의 쿼리가 어떻게 수행되는지 내부를 보여주는 **SQL 명령**
 - MySQL이 해당 쿼리를 어떻게 실행할지 **Execution Plan**을 보여줌. 이를 바탕으로 느리게 동작하는 쿼리의 최적화가 가능해짐
 - 보통 느린 쿼리의 경우 문제가 되는 테이블에 인덱스를 붙이는 것이 일반적

◆ Explain SQL (2)

```
-- EXPLAIN  
EXPLAIN SELECT  
  LEFT(s.created, 7) AS mon,  
  c.channel,  
  COUNT(DISTINCT user_id) AS mau  
FROM session s  
JOIN channel c ON c.id = s.channel_id  
GROUP BY 1, 2  
ORDER BY 1 DESC, 2;
```


◆ Index 소개 (1)

- ❖ Index는 테이블에서 특정 찾기 작업을 빠르게 수행하기 위해서 MySQL이 별도로 만드는 데이터 구조를 말함
 - 컬럼별로 만들어짐
 - Primary Key나 Foreign Key로 지정된 컬럼은 기본적으로 Index를 갖게 됨
 - 특정 컬럼을 바탕으로 검색을 자주 한다면 Index 생성이 큰 도움이 될 수 있음
- ❖ INDEX와 KEY는 동의어
- ❖ Index는 SELECT/DELETE/JOIN 명령을 빠르게 하지만 대신 INSERT/UPDATE 명령은 느리게 하는 단점이 존재
 - 테이블에 너무 많은 인덱스를 추가하면 인덱스의 로딩으로 인한 오버헤드로 인해 시스템이 전체적으로 느려질 수 있음

◆ Index 소개 (2)

❖ Index는 CREATE TABLE시 지정 가능 (컬럼 속성)

```
CREATE TABLE example (  
    id INT NOT NULL AUTO_INCREMENT,  
    index_col VARCHAR(20),  
    PRIMARY KEY (id),  
    INDEX index_name (index_col)  
);
```

◆ Index 소개 (3)

- ❖ Index는 테이블 생성 후 나중에 **ALTER TABLE** 혹은 **CREATE INDEX** 함수로 생성하는 것도 가능

```
ALTER TABLE testalter_tbl ADD INDEX (column1);
```

```
ALTER TABLE testalter_tbl ADD UNIQUE (column1);
```

```
ALTER TABLE testalter_tbl ADD FULLTEXT (column1);
```

```
ALTER TABLE testalter_tbl DROP INDEX (column1)
```

```
;
```

```
CREATE UNIQUE INDEX index_name ON table_name ( column1, column2,...);
```

◆ Index 실습 (1)

- ❖ 인덱스가 있는 경우와 없는 경우의 **SELECT** 필터링의 성능을 비교해보자
- ❖ 대상 테이블들은 `prod.session`과 `prod.session_with_index`
 - 인덱스는 `user_id`에 적용 (`id`와 `channel_id`에는 이미 걸려있음)

```
CREATE TABLE prod.session_with_index (  
  id int NOT NULL auto_increment,  
  user_id int not NULL,  
  created timestamp not NULL default CURRENT_TIMESTAMP,  
  channel_id int not NULL,  
  PRIMARY KEY(id),  
  FOREIGN KEY(channel_id) references channel(id),  
  INDEX user_id(user_id)
```

```
);
```

◆ Index 실습 (2)

❖ 다음 2개의 GROUP BY 작업의 성능을 비교

```
SELECT user_id, COUNT(1)  
FROM prod.session  
GROUP BY 1;
```

```
SELECT user_id, COUNT(1)  
FROM prod.session_with_index  
GROUP BY 1;
```

◆ SQL 실습

❖ MySQL Workbench로 실습

- [Explain SQL과 Index 최적화 실습 SQL 링크](#)



다음 스텝과 맞음말

◆ 백엔드 개발 핵심 기술: 관계형 데이터베이스와 SQL

❖ 백엔드 개발자 뿐만 아니라 모든 개발 직군이 알아야하는 기술

- 프론트엔드 및 데이터 직군 등등 포함
- 디지털 마케터나 프로덕트 매니저들의 경우에도 SQL은 알면 좋은 기술임

❖ 하지만 처음 배우는 거라면 이해가 쉽지 않을 수 있음

- 포기하지 않고 계속해서 자문자답하고 질문하기

❖ Spring Boot와 같은 백엔드 개발 프레임워크에서 꼭 필요

- 관계형 데이터베이스는 백엔드에서 빠질 수 없는 컴포넌트임
- 데이터 모델 설계의 중요성을 꼭 기억!!

- ◆ 하지만 관계형 데이터베이스는 귀중한 리소스라는 점 명심
 - ❖ 관계형 데이터베이스는 용량증대에 한계가 존재
 - Polyvore와 Udemy에서의 경험!
 - ❖ 정말로 서비스 운영에 필요한 데이터만 저장
 - 정보 저장 측면에서는 필요하지만 서비스 운영에 직접적으로 필요 없다면 다른 스토리지를 사용

◆ 배움의 전형적인 패턴

The Learning Curve

여기서 어떻게 하느냐가 아주 중요!

1. 가장 중요한 것은 버티는 힘
 - a. 이걸 즐겨야함 :)
2. 내가 뭘 모르지는지 생각해봐야함
 - a. 내가 어디서 막혔는지 구체적으로 질문할 수 있나?
3. 잘 하는 사람 보고 기죽지 않기

