7

7장 : 스토어드 프로시저

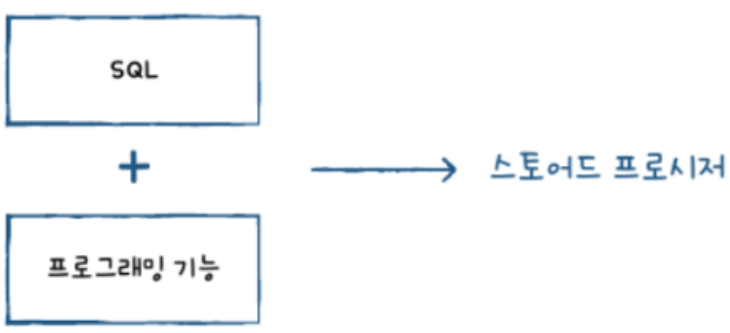
7장에서는 SQL을 한 문장씩 단순하게 사용하는 것뿐 아니라, 프로그래밍 로직까지 추가해서 사용하는 방법을 배운다. 기존에 다른 프로그래밍을 공부한 적이 있다면 이번 장은 쉽게 느껴질 것이다.

7-1 : 스토어드 프로시저 사용 방법

지금까지 배운 SQL을 자동화하지 않고 계속 반복적으로 사용하기엔 상당한 불편함과 한계가 있다. 스토어드 프로시저를 사용하면 MySQL 안에서도 다른 프로그래밍 언어처럼 프로그램 로직의 코딩이 가능하다.

SQL은 데이터베이스에서 사용되는 언어이다. 그런데 SQL을 사용하다 보면 다른 프로그래밍 언어의 기능이 필요할 때가 있다. C,자바,파이썬 등의 언어를 접해본 사람이라면 조건문이나 반복문을 사용해서 처리하면 더 편리하고 빠른 결과를 낼 수 있다는 것을 이미 알고 있다.

MySQL의 **스토어드 프로시저(Stored procedure)**는 SQL에 프로그래밍 기능을 추가해서 일반 프로그래밍 언어와 비슷한 효과를 낼 수 있다.



스토어드 프로시저 기본

스토어드 프로시저의 완전한 형식은 어렵게 느낄 수도 있지만, 실제로 사용하는 형식은 간단하다. 기본 형식을 먼저 익히고, 추가로 완전한 형식을 학습하도록 한다.

스토어드 프로시저의 개념과 형식

스토어드 프로시저(저장 프로시저)란 **MySQL에서 제공하는 프로그래밍 기능**이다. C, 자바, 파이썬 등의 프로그래밍과 조금 차이가 있지만, MySQL 내부에서 사용할 때 적절한 프로그래밍 기능을 제공한다.

또한 스토어드 프로시저는 쿼리 문의 집합으로도 볼 수 있으며, 어떠한 동작을 **일괄 처리**하기 위한 용도로도 사용한다. 자주 사용하는 일반적인 쿼리를 반복하는 것보다 스토어드 프로시저로 묶어놓고, 필요할 때마다 간단히 호출만 하면 훨씬 편리하게 MySQL을 운영할 수 있다.

스토어드 프로시저를 만드는 완전한 형식은 조금 복잡하다. 옵션을 모두 표현하면 오히려 복잡해 보이므로 가장 많이 사용되는 필수적인 형식만 살펴본다.

```
DELIMITER $$   → (1)
CREATE PROCEDURE 스토어드_프로시저_이름( IN 또는 OUT 매개변수) →(2)
BEGIN


이 부분에 SQL 프로그래밍을 코드로 작성


END $$
→ (1)
DELIMITER ;
```

(1) : 필수 항목으로 스토어드 프로시저를 묶어주는 기능을 한다. $$는 $ 한개만 사용해도 되지만 명확하게 표시하기 위해 2개를 사용한다. ##,%%,&&,// 등으로 바꿔도 된다.

(2) : 스토어드 프로시저의 이름을 정해준다. 가능하면 이름만으로도 스토어드 프로시저라는 것을 알 수 있도록 표현하는 것이 좋다. **(IN 또는 OUT)** 매개변수는 입,출력 매개변수인데 잠시 후에 살펴본다.

여기서 주의할 점이 있는데 두 번째 DELIMITER를 입력하고 세미콜론을 한 칸 뛰고 적어야 된다.(책에 안적혀 있음)

> 💬 **DELIMITER의 의미**
>
> DELIMITER는 '구분자'라는 의미이다. MySQL에서 구분자는 기본적으로 세미콜론(;)을 사용하는데, 스토어드 프로시저 안에 있는 많은 SQL의 끝에도 세미콜론을 사용한다. 문제는 세미콜론이 나왔을 때 이것이 SQL의 끝인지,스토어드 프로시저의 끝인지 모호해질 수 있다. 그래서 구분자를 $$로 바꿔서 $$가 나올 때까지는 스토어드 프로시저가 끝난 것이 아니라는 것을 표시하는 것이다.
>
> 즉 **세미콜론은 SQL의 끝으로만 표시하고 $$는 스토어드 프로시저의 끝으로 사용한다.** 그리고 마지막 행에서 DELIMITER를 세미콜론으로 바꿔주면 원래대로 MySQL의 구분자가 세미콜론으로 돌아온다.

CREATE PROCEDURE는 스토어드 프로시저를 만든 것 뿐이며, 아직 실행(호출)한 것은 아니다. 비유하자면 커피 자판기를 만든 것이지, 커피를 뽑은 것은 아니다.

스토어드 프로시저를 호출하는 형식은 다음과 같이 간단하다. **CALL** 다음에 스토어드 프로시저의 이름과 괄호를 붙여주면 된다. 필요하다면 괄호 안에 매개변수를 넣어서 사용할 수도 있다.

```
CALL 스토어드_프로시저_이름( );
```

스토어드 프로시저의 생성

먼저 간단한 스토어드 프로시저의 생성을 예로 살펴보도록 한다.

```sql
USE MARKET_DB;
DROP PROCEDURE IF EXISTS USER_PROC;

DELIMITER $$
CREATE PROCEDURE USER_PROC()
BEGIN
    SELECT * FROM MEMBER;
END $$
DELIMITER ;

CALL USER_PROC() ;
```

스토어드 프로시저 실습

스토어드 프로시저에는 프로그래밍 기능을 사용하고 싶은 만큼 적용할 수 있다. 그러면 더 강력하고 유연한 기능을 포함하는 스토어드 프로시저를 생성할 수 있다.

매개변수의 사용

스토어드 프로시저에서는 실행 시 **입력 매개변수**를 지정할 수 있다. 파이썬이나 R처럼 함수의 옵션처럼 사용할 수 있는 그 매개변수 맞다.

입력 매개변수를 지정하는 형식은 다음과 같다.

```
IN 입력_매개변수_이름 데이터_형식
```

입력 매개변수가 있는 스토어드 프로시저를 실행하기 위해서는 다음과 같이 괄호 안에 값을 전달하면 된다.

```
CALL 프로시저_이름(전달할 값);
```

스토어드 프로시저에서 처리된 결과를 **출력 매개변수**를 통해 얻을 수도 있다. 출력 매개변수는 커피 자판기에서 미리 준비하고 있는 컵이라고 보면 된다. 출력 매개변수의 형식은 다음과 같다.

```
OUT 출력_매개변수_이름 데이터_형식
```

출력 매개변수에 값을 대입하기 위해서는 주로 **SELECT ~ INTO** 문을 사용한다. 매개변수는 개념이 조금 어려울 수 있지만 잠시 후에 예제를 통해 확인한다.

출력 매개변수가 있는 스토어드 프로시저를 실행하기 위해서는 다음과 같이 사용한다.

```
CALL 프로시저_이름(@변수명);
SELECT @변수명;
```

입력 매개변수의 활용

입력 매개변수가 있는 스토어드 프로시저를 생성하고 실행해본다.

```sql
USE MARKET_DB;
DROP PROCEDURE IF EXISTS USER_PROC1;

DELIMITER $$
CREATE PROCEDURE USER_PROC1(IN USERNAME VARCHAR(10))
BEGIN
    SELECT * FROM MEMBER WHERE MEM_NAME = USERNAME;
END $$
DELIMITER ;

CALL USER_PROC1('에이핑크');
```

파이썬이나 R 등 프로그래밍 언어를 이전에 사용해봤으면 직관적일 것이다.

mem_id	mem_name	mem_number	addr	phone1	phone2	height	debut_date
APN	에이핑크	6	경기	031	77777777	164	2011-02-10

이번에는 2개의 입력 매개변수가 있는 스토어드 프로시저를 만들어본다.

```sql
DELIMITER $$
CREATE PROCEDURE USER_PROC2(
    IN USERNUMBER INT,
    IN USERHEIGHT INT)
BEGIN
    SELECT * FROM MEMBER
        WHERE MEM_NUMBER > USERNUMBER AND HEIGHT > USERHEIGHT;
END $$
DELIMITER ;

CALL USER_PROC2(6,165);
```

mem_id	mem_name	mem_number	addr	phone1	phone2	height	debut_date
GRL	소녀시대	8	서울	02	44444444	168	2007-08-02
TWC	트와이스	9	서울	02	11111111	167	2015-10-19

출력 매개변수의 활용

이번에는 출력 매개변수가 있는 스토어드 프로시저를 생성한다.

다음 스토어드 프로시저는 NOTABLE이라는 이름의 테이블에 넘겨 받은 값을 입력하고, ID열의 최대값을 알아내는 기능을 한다. ID 열의 최대값은 결국 방금 입력한 행의 순차 번호이다.

```sql
DELIMITER $$
CREATE PROCEDURE USER_PROC3(
IN TXTVALUE CHAR(10),
OUT OUTVALUE INT)
BEGIN
    INSERT INTO NOTABLE VALUES(NULL,TXTVALUE);
    SELECT MAX(ID) INTO OUTVALUE FROM NOTABLE;
END $$
DELIMITER ;
```

CALL을 사용하지 않아서 실행은 되지 않는다.

DESC SQL을 실행하면 NOTABLE이 없다고 오류 메시지가 나온다. 우리는 아직 NOTABLE을 만든 적이 없기 때문이다.

스토어드 프로시저를 만드는 시점에는 아직 존재하지 않는 테이블을 사용해도 된다. 단 CALL로 실행하는 시점에는 사용할 테이블이 있어야 한다.

이제 NOTABLE 테이블을 만든다. 간단히 ID열과 TXT 열을 만들면 된다.

```sql
CREATE TABLE IF NOT EXISTS NOTABLE(
    ID INT AUTO_INCREMENT PRIMARY KEY,
    TXT CHAR(10)
);
```

테이블을 만들었으니 스토어드 프로시저를 호출할 차례이다. 출력 매개변수의 위치에 **@변수명** 형태로 변수를 전달해주면 그 변수에 결과가 저장된다. 그리고 SELECT로 출력하면 된다.

다음 SQL을 계속 실행하면 값이 2,3,4,...로 증가한다.

```sql
CALL USER_PROC3('테스트1',@MYVALUE);
SELECT CONCAT('입력된 ID 값 ==>',@MYVALUE);

CALL USER_PROC3('테스트1',@MYVALUE);
SELECT CONCAT('입력된 ID 값 ==>',@MYVALUE);

CALL USER_PROC3('테스트1',@MYVALUE);
SELECT CONCAT('입력된 ID 값 ==>',@MYVALUE);

CALL USER_PROC3('테스트1',@MYVALUE);
SELECT CONCAT('입력된 ID 값 ==>',@MYVALUE);
```

CONCAT('입력된 ID 값 ==>',@MYVALUE)
입력된 ID 값 ==>4

SQL 프로그래밍의 활용

이번에는 스토어드 프로시저 안에 SQL 프로그래밍을 활용해본다.

조건문의 기본인 **IF~ELSE** 문을 사용해본다. 데뷔 연도가 2015년 이전이면 '고참', 2015년 이후이면 '신인'을 출력하는 스토어드 프로시저를 작성해본다.

```sql
DELIMITER $$
CREATE PROCEDURE IFELSE_PROC(
IN MEMNAME VARCHAR(10)
)
BEGIN
    DECLARE DEBUTYEAR INT; -- 변수선언
    SELECT YEAR(DEBUT_DATE) INTO DEBUTYEAR FROM MEMBER
        WHERE MEM_NAME = MEMNAME;
    IF (DEBUTYEAR >= 2015) THEN
        SELECT '신인' AS '메세지';
    ELSE
        SELECT '고참' AS '메세지';
    END IF;
END $$
DELIMITER ;
CALL IFELSE_PROC('오마이걸');
```

메세지
신인

> 💬 **날짜와 관련된 MySQL 함수**
>
> MySQL은 날짜와 관련된 함수를 여러 개 제공한다. **YEAR(날짜), MONTH(날짜), DAY(날짜)**를 사용할 수 있는데 날짜에서 연, 월,일을 구해주는 함수이다. 또 **CURDATE()** 함수는 현재 날짜를 알려준다.
>
> 다음 SQL은 현재 연,월,일을 출력한다.
>
> **SELECT YEAR(CURDATE()), MONTH(CURDATE()), DAY(CURDATE());**

이번에는 여러 번 반복하는 **WHILE** 문을 활용한다. 1부터 100까지의 합계를 계산해본다.

```
DELIMITER $$
CREATE PROCEDURE WHILE_PROC()
BEGIN
    DECLARE SUM INT; -- 합계
    DECLARE NUM INT; -- 1부터 100까지 증가
    SET SUM = 0;
    SET NUM = 1;

    WHILE (NUM<=100) DO -- 100까지 반복
        SET SUM = SUM+NUM;
        SET NUM = NUM+1; -- 숫자 증가
    END WHILE;
    SELECT SUM AS '1~100 합계';
END $$
DELIMITER ;

CALL WHILE_PROC();
```

1~100 합계
5050

일반 프로그래밍 언어와 비슷하게 스토어드 프로시저 안에서도 반복문 프로그래밍이 가능하다는 것을 확인할 수 있다. IF ~ ELSE 문이 반복되면서 SUM에는 1+2+3...+100까지 누적된다. NUM은 1,2,3,..100으로 값이 변경된다.

마지막으로 **동적 SQL**을 활용해본다. **동적 SQL**은 이름 그대로 SQL이 변경된다. 다음 예제는 테이블을 조회하는 기능을 한다. 그런데 테이블은 고정된 것이 아니라, 테이블 이름을 매개변수로 전달받아서 해당 테이블을 조회한다.

```
DELIMITER $$
CREATE PROCEDURE DYNAMIC_PROC(
    IN TABLENAME VARCHAR(20)
)
BEGIN
    SET @sqlQuery = CONCAT('SELECT * FROM ', TABLENAME);
    PREPARE MYQUERY FROM @sqlQuery;
    EXECUTE MYQUERY;
    DEALLOCATE PREPARE MYQUERY;
END $$
DELIMITER ;

CALL DYNAMIC_PROC('member');
```

빨간 박스 안에 FROM 다음 띄어쓰기 한칸을 하고 따옴표를 넣어야된다.

mem_id	mem_name	mem_number	addr	phone1	phone2	height	debut_date
APN	에이핑크	6	경기	031	77777777	164	2011-02-10
BLK	블랙핑크	4	경남	055	22222222	163	2016-08-08
GRL	소녀시대	8	서울	02	44444444	168	2007-08-02
ITZ	잇지	5	경남	NULL	NULL	167	2019-02-12
MMU	마마무	4	전남	061	99999999	165	2014-06-19
OMY	오마이걸	7	서울	NULL	NULL	160	2015-04-21
RED	레드벨벳	4	경북	054	55555555	161	2014-08-01
SPC	우주소녀	13	서울	02	88888888	162	2016-02-25
TWC	트와이스	9	서울	02	11111111	167	2015-10-19
WMN	여자친구	6	경기	031	33333333	166	2015-01-15

스토어드 프로시저는 다른 개체의 삭제와 마찬가지로 **DROP PROCEDURE 프로시저_이름** 구문을 사용해서 삭제한다.

7-2 : 스토어드 함수와 커서

스토어드 프로시저와 함께 SQL 프로그래밍 기능으로 사용되는 데이터베이스 개체로는 스토어드 함수와 커서가 있다. 스토어드 함수와 커서를 잘 활용하면 SQL의 단순한 기능을 더욱 강력하게 확장할 수 있다.

스토어드 함수는 MySQL에서 제공하는 내장 함수 외에 직접 함수를 만드는 기능을 제공한다. 즉, MySQL이 제공하는 함수를 그대로 사용할 수 없는 경우가 발생한다면 직접 스토어드 함수를 작성해서 사용할 수 있다.

스토어드 함수는 스토어드 프로시저와 형태가 비슷하지만 세부적으로는 다르다. 특히 용도가 다르며, RETURNS 예약어를 통해서 하나의 값을 반환해야 하는 특징을 가진다.

커서는 스토어드 프로시저 안에서 한 행씩 처리할 때 사용하는 프로그래밍 방식이다. 문법은 조금 복잡해 보이지만, 형태가 대부분 비슷하게 고정되어 있어서 한 번 익혀 놓으면 다음에는 어렵지 않게 활용할 수 있다.

스토어드 함수

스토어드 함수는 앞으로 배운 스토어드 프로시저와 비슷하다. 하지만 사용 방법이나 용도가 조금 다르니, 스토어드 프로시저와 별개로 알아 둘 필요가 있다.

스토어드 함수의 개념과 형식

MySQL은 다양한 함수를 제공한다. 앞에서 SUM(),CAST(),CONCAT(),CURRENT_DATE() 등을 사용해봤다. 하지만 MySQL이 사용자가 원하는 모든 함수를 제공하지 않기 때문에 필요하다면 사용자가 직접 함수를 만들어서 사용할 필요가 있다. 이렇게 직접 만들어서 사용하는 함수를 **스토어드 함수(Stored function)**라고 부르며 다음과 같은 형식으로 구성할 수 있다.

```
DELIMITER $$
CREATE FUNCTION 스토어드_함수_이름 (매개변수)
    RETURNS 반환형식
BEGIN

    이 부분에 프로그래밍 코딩
    RETURN 반환값;

END $$
DELIMITER ;
SELECT 스토어드_함수_이름();
```

- 스토어드 함수는 **RETURNS문**으로 반환할 값의 데이터 형식을 지정하고, 본문 안에서는 **RETURN 문**으로 하나의 값을 반환해야 한다.
- 스토어드 함수의 매개변수는 모두 입력 매개변수이다. 그리고 **IN**을 붙이지 않는다.
- 스토어드 프로시저는 **CALL**로 호출하지만, 스토어드 함수는 **SELECT** 문 안에서 호출된다.
- 스토어드 프로시저 안에서는SELECT문을 사용할 수 있지만, 스토어드 함수 안에서는 **SELECT를 사용할 수 없다.**
- 스토어드 프로시저는 여러 SQL문이나 숫자 계산 등의 다양한 용도로 사용하지만, **스토어드 함수는 계산을 통해서 하나의 값을 반환하는데 주로 사용한다.**

> 💬 **내장 함수**
>
> MySQL에서 제공하는 함수를 **내장함수(Built-in-function)**라고 부른다. 내장함수는 제어 흐름 함수, 문자열 함수, 수학 함수, 날짜/시간 함수, 전체 특스트 검색 함수, 데이터 형식 변환 함수 등으로 분류할 수 있으며 세부적으로 그 종류는 수백 개가 넘는다.

스토어드 함수의 사용

스토어드 함수를 사용하기 위해서는 먼저 다음 SQL로 스토어드 함수 생성 권한을 허용해줘야 한다. 구문이 조금 어렵게 보이지만, MySQL에서 한 번만 설정해주면 이후에는 사용하지 않아도 된다.

```
SET GLOBAL LOG_BIN_TRUST_FUNCTION_CREATORS=1;
```

먼저 간단한 스토어드 함수를 만들어서 사용해본다. 숫자 2개의 합계를 계산하는 스토어드 함수를 만들어본다.

```
DELIMITER $$
CREATE FUNCTION SUMFUNC(NUMBER1 INT, NUMBER2 INT) -- 2개의 정수형 매개변수
    RETURNS INT    -- 함수가 반환하는 데이터 형식 지정
BEGIN
    RETURN NUMBER1+NUMBER2; -- 정수형 결과 반환
END $$
DELIMITER ;

SELECT SUMFUNC(100,100) AS 'SUM'; --  SELECT로 호출
```

	SUM
▶	200

결과

이번에는 데뷔 연도를 입력하면, 활동 기간이 얼마나 되었는지 출력해주는 함수를 만들어본다.

```
DELIMITER $$
CREATE FUNCTION CALDATE(DYEAR INT) -- 매개변수로 데뷔년도
    RETURNS INT
BEGIN
    DECLARE RUNYEAR INT; -- 활동기간(연도)
    SET RUNYEAR = YEAR(CURDATE()) - DYEAR;
    RETURN RUNYEAR;
END $$
DELIMITER ;

SELECT CALDATE(2010) AS '활동연도';
```

	활동 연도
▶	15

결과

필요하다면 다음과 같이 함수의 반환 값을 **SELECT ~ INTO~**로 저장했다가 사용할 수도 있다. 함수의 반환값을 각 변수에 저장한 후, 그 차이를 계산해서 출력했다. 즉, 데뷔 연도와 활동 햇수 차이가 출력되었다.

```
SELECT CALDATE(2007) INTO @DEBUT2007;
SELECT CALDATE(2013) INTO @DEBUT2013;
SELECT @DEBUT2007-@DEBUT2013 AS '2013과 2007 차이';
```

	2013과 2007 차이
▶	6

함수는 주로 테이블을 조회한 후, 그 값을 계산할 때 사용한다. 회원 테이블에서 모든 회원이 데뷔 한지 몇 년이 되었는지 조회해본다.

YEAR() 함수는 연도만 추출해주는 함수이다. CALDATE (연도)로 함수를 사용해서 각 회원별 활동 햇수를 출력해본다.

```
SELECT MEM_ID,MEM_NAME,CALDATE(YEAR(DEBUT_DATE)) AS '활동햇수'
    FROM MEMBER;
```

	MEM_ID	MEM_NAME	활동 햇수
▶	APN	에이핑크	14
	BLK	블랙핑크	9
	GRL	소녀시대	18
	ITZ	잇지	6
	MMU	마마무	11
	OMY	오마이걸	10
	RED	레드벨벳	11
	SPC	우주소녀	9
	TWC	트와이스	10
	WMN	여자친구	10

함수의 삭제는 DROP FUNCTION 문을 사용하면 된다.

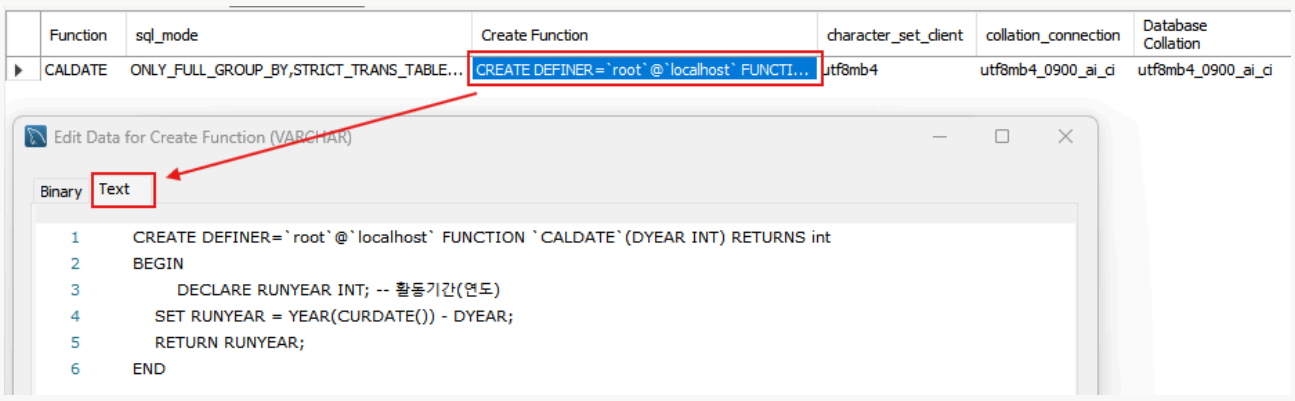
```
DROP FUNCTION CALDATE;
```

💬 **스토어드 함수의 내용 확인**

기존에 작성된 스토어드 함수의 내용을 확인하려면 다음과 같은 쿼리 문을 사용하면 된다.

SHOW CREATE FUNCTION 함수_이름 ;

그리고 **[Create Function]**에서 마우스 오른쪽 버튼을 클릭하고 **[Open Value in Viewer]**를 선택하면 Edit Data for Create Function (VARCHAR) 창의 **[Text]** 탭에서 작성했던 스토어드 함수의 코드를 확인할 수 있다.

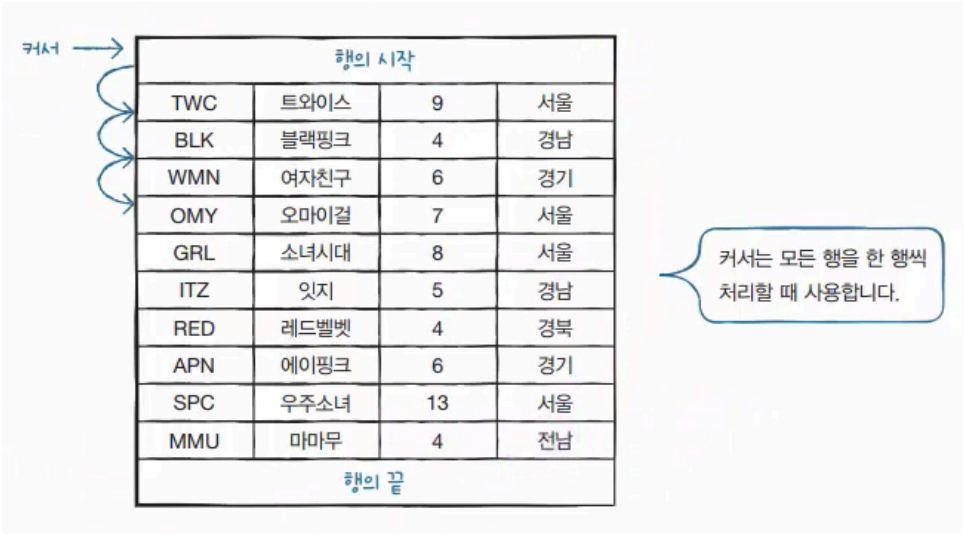


커서로 한 행씩 처리하기

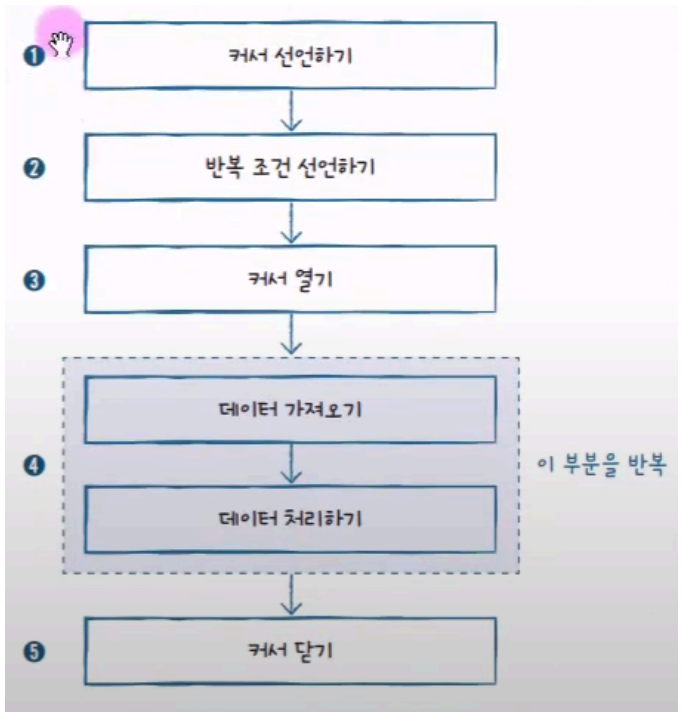
커서(cursor)는 테이블에서 한 행씩 처리하기 위한 방식이다. 스토어드 프로시저 내부에서 커서를 사용하는 방법을 알아본다.

커서의 기본 개념

커서는 첫 번째 행을 처리한 후에 마지막 행까지 한 행씩 접근해서 값을 처리한다. 다음 그림과 같이 처음에는 커서가 행의 시작을 가리킨 후에 한 행씩 차례대로 접근한다.



커서는 일반적으로 다음의 작동 순서를 통해 처리가 된다. 조금 복잡해보이지만 항상 이런 형태로 커서가 사용되니 기억하고 있어야 한다.



커서는 대부분 **스토어드 프로시저와 함께 사용된다**. 커서의 세부 문법을 외우기보다는 커서를 사용하는 전반적인 흐름에 초점을 맞춰서 실습을 진행한다.

커서의 단계별 실습

그룹의 평균 인원수를 구하는 스토어드 프로시저를 작성해본다. 그런데 이번에는 커서를 활용하여 한 행씩 접근해서 회원의 인원수를 누적시키는 방식으로 처리해본다. 먼저 전체 형식을 먼저 살펴본다.

```sql
DELIMITER $$
CREATE PROCEDURE CURSOR_PROC()
BEGIN
    DECLARE MEMNUMBER INT;
    DECLARE CNT INT DEFAULT 0;
    DECLARE TOTNUMBER INT DEFAULT 0;
    DECLARE ENDOFROW BOOLEAN DEFAULT FALSE;

    DECLARE MEMBERCURSOR CURSOR FOR
        SELECT MEM_NUMBER FROM MEMBER;

    DECLARE CONTINUE HANDLER
        FOR NOT FOUND SET ENDOFROW = TRUE;

    OPEN MEMBERCURSOR;

    CURSOR_LOOP : LOOP
        FETCH MEMBERCURSOR INTO MEMNUMBER;

        IF ENDOFROW THEN
            LEAVE CURSOR_LOOP;
        END IF;

        SET CNT = CNT+1;
        SET TOTNUMBER  = TOTNUMBER + MEMNUMBER;
    END LOOP CURSOR_LOOP;

    SELECT (TOTNUMBER/CNT) AS '회원의 평균 인원 수';
    CLOSE MEMBERCURSOR;
END $$
DELIMITER ;
```

매우 길지만 천천히 하나씩 살펴본다.

1. **사용할 변수 준비**

   회원의 평균 인원수를 계산하기 위해서 각 회원의 인원수(memNumber), 전체 인원의 합계(totNumber), 읽은 행의 수(cnt) 변수를 3개 준비한다.

   전체 인원의 합계와 읽은 행의 수를 누적시켜야 하기 때문에 **DEFAULT 문**을 사용해서 초기값을 0으로 설정한다.

   추가로 행의 끝을 파악하기 위한 변수 **ENDOFROW**를 준비한다. 처음에는 당연히 행의 끝이 아닐테니 FALSE로 초기화시켰다.

```sql
DECLARE MEMNUMBER INT;
DECLARE CNT INT DEFAULT 0;
DECLARE TOTNUMBER INT DEFAULT 0;
DECLARE ENDOFROW BOOLEAN DEFAULT FALSE;
```

2. **커서 선언하기**

   이제 **커서를 선언**한다. 커서라는 것은 결국 SELECT문이다. 회원 테이블을 조회하는 구문을 커서로 만들어 놓으면 된다. 커서 이름은 MEMBERCURSOR로 지정했다.

```sql
DECLARE MEMBERCURSOR CURSOR FOR
    SELECT MEM_NUMBER FROM MEMBER;
```

3. **반복 조건 선언하기**

   이제는 '행이 끝나면 어떻게 설정해야 더 이상 반복하지 않을까?'를 생각해볼 차례이다. 행의 끝에 다다르면 앞에서 선언한 **ENDOFROW** 변수를 **TRUE**로 설정한다. 설정한 내용은 잠시 후 나오는 반복 코드와 깊게 연관되어 있다.

   DECLARE CONTINUE HANDLER는 반복 조건을 준비하는 예약어이다. 그리고 **FOR NOT FOUND**는 더 이상 행이 없을 때 이어진 문장을 수행한다. 즉, 행이 끝나면 ENDOFROW에 TRUE를 대입한다.

```
DECLARE CONTINUE HANDLER
    FOR NOT FOUND SET ENDOFROW = TRUE;
```

4. **커서 열기**

   이제 커서를 열 차례이다. 앞에서 준비한 커서를 간단히 OPEN으로 열면 된다.

```
OPEN MEMBERCURSOR;
```

5. **행 반복하기**

   커서의 끝까지 한 행씩 접근해서 반복할 차례이다. 코드의 형식은 다음과 같다.

```
CURSOR_LOOP: LOOP
    이 부분을 반복
END LOOP CURSOR_LOOP
```

CURSOR_LOOP는 반복할 부분의 이름을 지정한 것이다. 여기서는 커서를 이용한 실습이므로 알아보기 쉽도록 CURSOR_LOOP로 지정했다. 그런데 이 코드는 무한 반복하기 때문에 코드 안에 반복문을 빠져나갈 조건이 필요하다. 앞에서 행의 끝에 다다르면 ENDOFROW를 TRUE로 변경하기로 설정했다. 그러므로 반복되는 부분에는 다음 코드가 필수로 들어가야 한다.

LEAVE는 반복할 이름을 빠져나간다. 결국 행의 끝에 다다르면 반복 조건을 선언한 3번에 의해서 ENDOFROW가 TRUE로 변경되고 반복하는 부분을 빠져나가게 된다.

```
IF ENDOFROW THEN
    LEAVE CURSOR_LOOP;
END IF;
```

이제 반복할 부분을 전체 표현한다.

FETCH는 한 행씩 읽어는 것이다. 2번에서 커서를 선헌할 때 인원수 행을 조회했으므로 MEMNUMBER 변수에는 각 회원의 인원수가 한 번에 하나씩 저장된다.

SET 부분에서 읽은 행의 수(CNT)를 하나씩 증가시키고, 인원 수도 TOTNUMBER에 계속 누적시킨다.

```
CURSOR_LOOP : LOOP
    FETCH MEMBERCURSOR INTO MEMNUMBER;

    IF ENDOFROW THEN
        LEAVE CURSOR_LOOP;
    END IF;

    SET CNT = CNT+1;
    SET TOTNUMBER  = TOTNUMBER + MEMNUMBER;
END LOOP CURSOR_LOOP;
```

이제 반복을 빠져나오면 최종 목표였던 회원의 평균 인원수를 계산한다. 누적된 총 인원수를 읽은 행의 수로 나누면 된다.

```
SELECT (TOTNUMBER/CNT) AS '회원의 평균 인원 수';
```

6. **커서 닫기**

   모든 작업이 끝났으면 커서를 닫는다.

```
CLOSE MEMBERCURSOR;
```

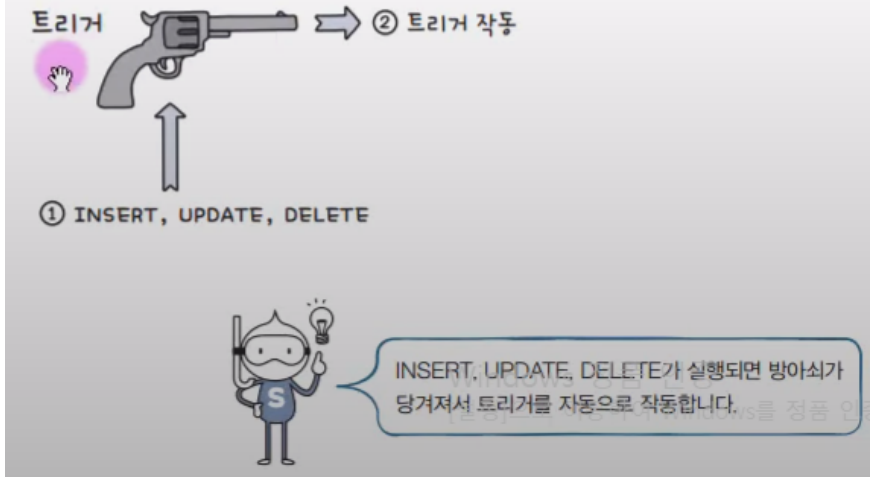
이제 스토어드 프로시저를 실행해서 결과를 확인해본다. 회원의 평균 인원수는 6.6명이 나온다.

7-3 : 자동 실행되는 트리거

트리거(Trigger)는 INSERT, UPDATE, DELETE 문이 작동할 때, 자동으로 실행되는 프로그래밍 기능이다. 트리거를 활용하면 데이터가 삭제될 때 해당 데이터를 다른 곳에 자동으로 백업할 수 있다.

트리거는 자동으로 수행하여 사용자가 추가 작업을 잊어버리는 실수를 방지해준다. 회사원이 퇴사하면 직원 테이블에서 삭제하면 된다. 그런데 나중에 퇴사한 직원이 회사에 다녔던 기록을 요청할 수도 있다. 이를 미리 예방하려면 직원 테이블에서 삭제하기 전에 퇴사자 테이블에 옮겨 놓아야 한다. 문제는 이런 작업을 수동으로 할 경우 백업하지 않고 데이터를 삭제할 수 있다는 것이다.

트리거는 이런 실수를 방지할 수 있다. 직원 테이블에서 사원을 삭제하면 해당 데이터를 자동으로 퇴사자 테이블에 들어가도록 설정할 수 있다. 즉, 트리거를 사용하면 데이터에 오류가 발생하는 것을 막을 수 있다. 이런 것을 **데이터의 무결성**이라고 부르기도 한다.



트리거 기본

트리거는 사전적 의미로 방아쇠를 뜻한다. 총의 방아쇠를 당기면 자동으로 총알이 나가듯이, 트리거는 테이블에 무슨 일이 일어나면 자동으로 실행된다.

트리거 개요

트리거는 테이블에 INSERT, UPDATE, DELETE 작업이 발생하면 실행되는 코드이다.

예를 들어 MARKET_DB의 회원 중 블랙핑크라는 회원이 탈퇴하는 경우, 회원에서 탈퇴하면 간단히 회원 테이블(MEMBER)에서 블랙핑크의 정보를 DELETE 문으로 삭제하면 된다. 그런데 나중에 회원에서 탈퇴한 사람의 정보를 어떻게 알 수 있을까? 원칙적으로 블랙핑크는 이미 데이터베이스에 존재하지 않기 때문에 알 수 있는 방법이 없다.

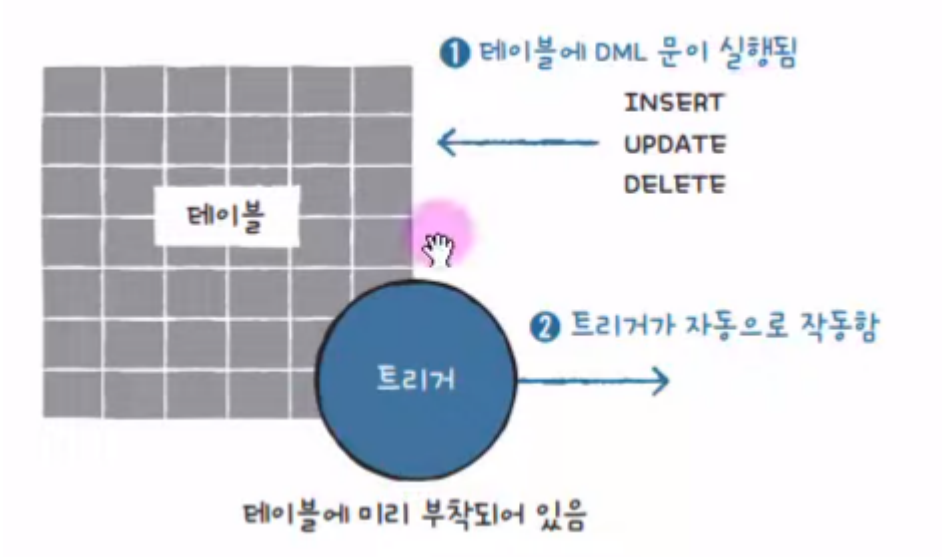
이를 방지하는 방법은 블랙핑크의 행을 삭제하기 전에 그 내용을 다른 곳에 복사해 놓으면 된다. 그런데 이런 작업을 매번 수작업으로 할 경우, 깜빡 잊고 데이터를 복사해 놓는 것을 잊어버릴 수도 있다.

만약 회원 테이블에서 DELETE 작업이 일어날 경우 해당 데이터가 삭제되기 전에 다른 곳에 자동으로 저장해주는 기능인 트리거를 사용한다면 사용자가 수작업으로 데이터를 복사할 필요가 없다.

트리거의 기본 작동

트리거는 테이블에서 **DML**(Data Manipulation Language)문 (INSERT, UPDATE, DELETE 등)의 **이벤트**가 발생할 때 작동한다. 테이블에 미리 **부착(attach)**되는 프로그램 코드라고 생각하면 된다.

여기서 말하는 트리거는 AFTER 트리거이다. BEFORE 트리거는 작동 방식이 조금 다르다. 일반적으로는 AFTER 트리거를 많이 사용하므로, 트리거는 AFTER 트리거를 기준으로 설명하도록한다.



트리거는 스토어드 프로시저와 문법이 비슷하지만, **CALL** 문으로 직접 실행시킬 수는 없고 오직 테이블에 INSERT, UPDATE, DELETE 등의 이벤트가 발생할 경우에만 자동으로 실행된다. 또한 스토어드 프로시저와 달리 트리거에는 IN,OUT 매개변수를 사용할 수 없다.

우선 간단한 트리거를 보고 그 작동에 대해 이해해본다. 테스트로 사용할 간단한 테이블을 만든 후 트리거를 부착해본다.

```
USE MARKET_DB;
CREATE TABLE IF NOT EXISTS TRIGGER_TABLE(ID INT, TXT VARCHAR(10));
INSERT INTO TRIGGER_TABLE VALUES(1,'레드벨벳');
INSERT INTO TRIGGER_TABLE VALUES(2,'잇지');
INSERT INTO TRIGGER_TABLE VALUES(3,'블랙핑크');

DELIMITER $$
CREATE TRIGGER MYTRIGGER     -- 트리거 이름 지정
    AFTER DELETE             -- DELETE 문 발생 이후 작동하라는 의미
    ON TRIGGER_TABLE        -- 트리거를 부착할 테이블 지정
    FOR EACH ROW             -- 각 행마다 적용한다는 의미(트리거에 항상 써준다고 보면됨)
BEGIN
    SET @MSG = '회원이 삭제됨'; -- 트리거 실행 시 작동되는 코드
END $$
DELIMITER ;
```

코드를 실행하면 아무 결과도 나오지 않는다. @MSG 변수에 빈 문자를 넣고 INSERT 문을 실행했다. 그런데 TRIGGER_TABLE에는 DELETE에만 작동하는 트리거를 부착시켜 놓는다. 그러므로 트리거가 작동하지 않아서 빈 @MSG가 그대로 출력된 것이다. UPDATE 문도 마찬가지로 트리거가 작동하지 않았다.

```
SET @MSG = '';
INSERT INTO TRIGGER_TABLE VALUES(4,'마마무');
SELECT @MSG;
UPDATE TRIGGER_TABLE SET TXT ='블핑' WHERE ID=3;
SELECT @MSG;
```

이제 DELETE 문을 테이블에 적용시켜본다. 예상대로 DELETE 문을 실행하니 트리거가 작동해서 @MSG 변수에 트리거에서 설정한 내용이 입력된 것을 확인할 수 있다.

```
DELETE FROM TRIGGER_TABLE WHERE ID =4;
SELECT @MSG;
```

	@MSG
▶	회원이 삭제됨

이렇게 트리거는 테이블에 부착해서 사용할 수 있다. 이 예제에서는 간단히 @MSG에 값을 대입하는 내용만 코딩했지만, 그 부분을 실제로 필요로 하는 복잡한 SQL문들로 대치하면 유용한 트리거로 작동할 것이다.

트리거 활용

트리거는 테이블에 입력/수정/삭제되는 정보를 백업하는 용도로 활용할 수 있다.

다음과 같은 사례를 생각해본다. 은행의 창구에서 새로 계좌를 만들 때는 INSERT를 사용한다. 계좌에 입금하거나 출금하면 UPDATE를 사용해서 값을 변경하며, 계좌를 폐기하면 DELETE가 작동한다.

그런데 계좌라는 중요한 정보를 누가 입력/수정/삭제했는지 알 수 없다면 나중에 계좌에 문제가 발생했을 때 원인을 파악할 수 없을 것이다. 이럴 때를 대비해서 데이터에 입력/수정/삭제가 발생할 때, 트리거를 자동으로 작동시켜 데이터를 변경한 사용자와 시간 등을 기록할 수 있다.

이런 개념을 적영해서 MARKET_DB의 고객 테이블(MEMBER)에 입력된 회원의 정보가 변경될 때 변경한 사용자, 시간, 변경 전의 데이터 등을 기록하는 트리거를 작성해본다.

먼저 회원 테이블의 열을 간단히 아이디,이름,인원,주소 4개 열로 구성된 가수 테이블로 복사해서 진행해본다.

```
USE MARKET_DB;
CREATE TABLE SINGER(SELECT MEM_ID,MEM_NAME, MEM_NUMBER,ADDR FROM MEMBER);
```

가수 테이블에 INSERT나 UPDATE 작업이 일어나는 경우, 변경되기 전의 데이터를 저장할 백업 테이블을 미리 생성한다. 백업 테이블에는 추가로 수정 또는 삭제인지 구분할 변경된 타입(MODTYPE), 변경된 날짜(MODDATE), 변경한 사용자(MODUSER)를 추가했다.

```
CREATE TABLE BACKUP_SINGER
(MEM_ID CHAR(8) NOT NULL,
 MEM_NAME VARCHAR(10) NOT NULL,
 MEM_NUMBER INT NOT NULL,
 ARRD CHAR(2) NOT NULL,
 MODTYPE CHAR(2), -- 변경된 타입. 수정 또는 삭제
 MODDATE DATE, -- 변경된 날짜
 MODUSER VARCHAR(2) -- 변경한 사용자
 );
```

이제 본격적으로 변경과 삭제가 발생할 때 작동하는 트리거를 SINGER 테이블에 부착한다.

먼저 변경이 발생했을 때 작동하는 SINGER_UPDATE_TRG 트리거를 만든다.

```
DELIMITER $$
CREATE TRIGGER SINGER_UPDATE_TRG -- 트리거 이름
   AFTER UPDATE -- 변경 후에 작동하도록 설정
   ON SINGER -- 트리거를 부착할 테이블
   FOR EACH ROW
BEGIN
   INSERT INTO BACKUP_SINGER VALUES ( OLD.MEM_ID, OLD.MEM_NAME, OLD.MEM_NUMBER, OLD.ADDR, '수정', CURDATE(),CURRENT_USER() );
   ## OLD 테이블은 UPDATE나 DELETE가 수행될 때, 변경되기 전의 데이터가 잠깐 저장되는 임시 테이블이다. OLD 테이블에 UPDATE가 작동되면 이 행에 의해서 업데이트 이전의
   ## 데이터가 백업 테이블(BACKUP_SINGER)에 입력된다. 즉, 원래 데이터가 보존된다.
END $$
DELIMITER ;
```

OLD 테이블은 MySQL에서 내부적으로 제공되는 테이블이다.

이번에는 삭제가 발생했을 때 작동하는 SINGER_DELETE_TRG 트리거를 생성해본다 UPDATE 트리거와 매우 비슷하다.

```
DELIMITER $$
CREATE TRIGGER SINGER_DELETE_TRG -- 트리거 이름
   AFTER DELETE -- 변경 후에 작동하도록 설정
   ON SINGER -- 트리거를 부착할 테이블
   FOR EACH ROW
BEGIN
   INSERT INTO BACKUP_SINGER VALUES ( OLD.MEM_ID, OLD.MEM_NAME, OLD.MEM_NUMBER, OLD.ADDR, '삭제', CURDATE(),CURRENT_USER() );
END $$
DELIMITER ;
```

하나의 테이블에 여러 개의 트리거를 부착할 수 있다.

이제 데이터를 변경해본다. 한 건의 데이터를 업데이트하고, 여러 건을 삭제시켜본다. 그러고 수정 또는 삭제된 내용이 잘 보관되어 있는지 결과를 확인해본다.

```
UPDATE SINGER SET ADDR='영국' WHERE MEM_ID='BLK';
DELETE FROM SINGER WHERE MEM_NUMBER >=7;

SELECT * FROM BACKUP_SINGER;
```

MEM_ID	MEM_NAME	MEM_NUMBER	ARRD	MODTYPE	MODDATE	MODUSER
BLK	블랙핑크	4	경남	수정	2025-01-30	root@localhost
GRL	소녀시대	8	서울	삭제	2025-01-30	root@localhost
OMY	오마이걸	7	서울	삭제	2025-01-30	root@localhost
SPC	우주소녀	13	서울	삭제	2025-01-30	root@localhost
TWC	트와이스	9	서울	삭제	2025-01-30	root@localhost

TRUNCATE TABLE 테이블_이름은 모든 행 데이터를 삭제하는 기능이다.

백업 테이블을 조회했을 때, 1건이 수정되고 4건이 삭제된 것을 확인할 수 있다. 수정 또는 삭제 전의 데이터가 백업 테이블에 잘 보관되어 있다.

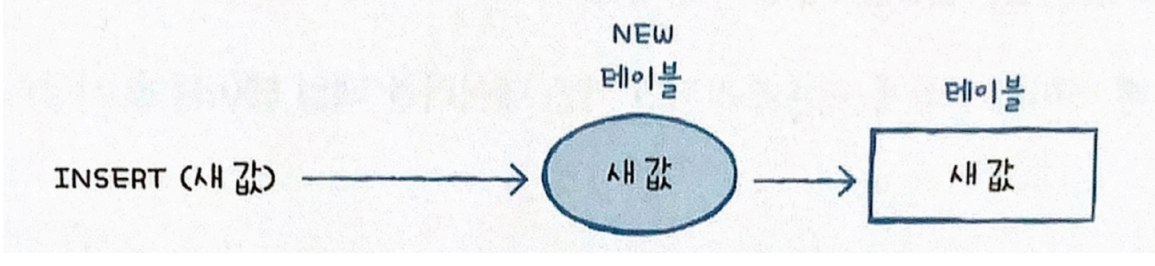
이번엔 테이블의 모든 행 데이터를 삭제해본다. DELETE문 대신 **TRUNCATE TABLE** 문으로 삭제하고 삭제가 잘 되었는지 백업 테이블을 확인해면 백업 테이블에 삭제된 내용이 들어가지 않는 것을 알 수 있다. DELETE 트리거는 오직 DELETE에서만 작동하기 때문이다.

MEM_ID	MEM_NAME	MEM_NUMBER	ARRD	MODTYPE	MODDATE	MODUSER
BLK	블랙핑크	4	경남	수정	2025-01-30	root@localhost
GRL	소녀시대	8	서울	삭제	2025-01-30	root@localhost
OMY	오마이걸	7	서울	삭제	2025-01-30	root@localhost
SPC	우주소녀	13	서울	삭제	2025-01-30	root@localhost
TWC	트와이스	9	서울	삭제	2025-01-30	root@localhost

트리거가 사용하는 임시 테이블

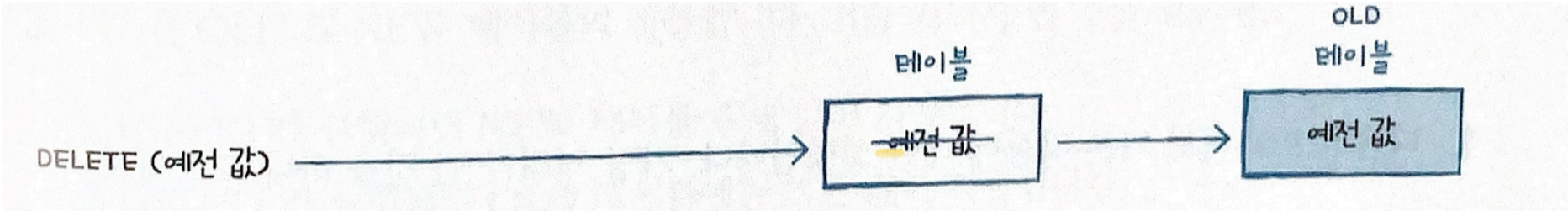
테이블에 INSERT, UPDATE, DELETE 작업이 수행되면 임시로 사용되는 시스템 테이블이 2개 있는데, 이름은 **NEW**와 **OLD** 이다. 두 테이블은 사용자가 만드는 것이 아니고 MySQL이 알아서 생성하고 관리하므로 신경 쓸 피료는 없다.

먼저 NEW 테이블은 **INSERT** (새 값)문이 실행되면 다음과 같이 작동한다.



지금까지 배운 바로는 INSERT 형태로 테이블에 새 값이 바로 들어간다. 하지만 사실 새 값은 테이블에 들어가기 전에 NEW 테이블에 잠깐 들어가 있다.

이번엔 **DELETE** (예전값) 의 작동을 살펴본다. OLD 테이블은 DELETE 문이 실행되면 다음 그림과 같이 작동한다.



이제 SINGER_DELETE_TRG 트리거의 INSERT문 안에 있는 **OLD.열 이름**의 의미를 파악했을 것이다.