



SQL 제대로 활용하기



07-1 스토어드 프로시저

07-2 SQL 프로그래밍

07-3 인덱스

07-4 뷰

07-5 스토어드 함수와 커서

07-6 트리거

- SQL도 파이썬과 같은 프로그래밍 언어에서 사용하는 조건문, 반복문이나 다양한 변수를 조합하는 등의 코딩 즉, 프로그래밍이 가능함
- 이와 같이 SQL로 프로그래밍하여 SQL에 저장하고, 그 내용을 재사용할 수 있도록 만드는 것이 바로 스토어드 프로시저(stored procedure) 임

▪ 스토어드 프로시저를 사용하는 이유

- 복잡한 쿼리를 이 프로시저 내부에 저장하고 호출하여 사용함으로써 다양한 이점이 있음
- 예를 들어 매일 보고서에 입력할 데이터를 추출하는 업무와 같이 자주 사용하는 쿼리들의 집합이 있을 때 스토어드 프로시저를 사용해 이를 프로그래밍해 놓으면 재사용할 수 있어서 매우 편리함

스토어드 프로시저의 장점

- **절차적 기능 구현:** SQL 쿼리는 절차적 기능을 제공하지 않지만, 스토어드 프로시저 내에서는 IF나 WHILE과 같은 제어 문장을 사용하여 절차적 프로그래밍을 할 수 있다.
- **유지·보수:** 스토어드 프로시저를 호출하는 곳에서는 스토어드 프로시저의 이름으로만 호출하므로, 스토어드 프로시저를 수정할 경우 호출한 곳에서는 별도의 수정 작업이 필요 없어 유지·보수 작업에 용이하다.
- **트래픽 감소:** 한 번의 요청으로 여러 SQL 문을 실행할 수 있다. SQL 문을 직접 작성하지 않고 프로시저에 매개변수만 담아 전달하면 되므로 클라이언트와 서버 간의 네트워크 트래픽이 감소한다.
- **보안:** MySQL의 스토어드 프로시저는 자체적으로 보안이 설정되어 있어서 스토어드 프로시저 실행 단위로 실행 권한을 부여할 수 있다. 즉, 세밀하게 권한을 제어할 수 있다.

스토어드 프로시저의 단점

- 스토어드 프로그램의 처리 성능은 다른 프로그래밍 언어에 비해서 느리다. 특히 다른 DBMS와 달리 MySQL은 스토어드 프로시저를 실행할 때마다 스토어드 프로시저의 코드를 분석(파싱)해 속도가 느려진다.
- 데이터베이스 제품에 따라 구문 및 규칙이 다르기 때문에, 다른 제품과 호환성이 낮다.
- 비즈니스 로직이 스토어드 프로시저에 있어 업무를 파악하거나 관리할 때, 관리해야 하는 요소가 늘어난다.

■ 스토어드 프로시저 생성하기

스토어드 프로시저의 기본 형식

```
DELIMITER &&  
CREATE PROCEDURE 프로시저명([IN | OUT | INOUT , 매개변수])  
BEGIN  
    변수 선언, 함수 실행 등을 위한 코드  
END &&  
DELIMITER ;
```

- ALTER 문으로는 스토어드 프로시저의 매개변수나 바디의 코드를 수정할 수 없다. 즉, 프로시저를 삭제한 후 다시 생성하는 방식으로만 수정할 수 있다.
- 스토어드 프로시저는 기본 반환값이 없다. 즉, RETURN 명령문을 사용할 수 없다.
- IN과 함께 정의된 매개변수는 입력 전용 매개변수를 의미한다.
- OUT과 함께 정의된 매개변수는 출력 전용 매개변수를 의미한다.
- INOUT과 함께 정의된 매개변수는 입력 및 출력 매개변수로 모두 사용할 수 있다.

■ 스토어드 프로시저 생성하기

- 실제 프로시저를 하나 생성해 보자. 워크벤치에서 새 쿼리 창을 열고 sakila 데이터베이스가 선택된 상태에서 다음 내용을 작성해 보자

스토어드 프로시저 생성

```
DELIMITER $$
CREATE PROCEDURE doit_proc()
BEGIN
    DECLARE customer_cnt INT;
    DECLARE add_number INT;
    SET customer_cnt = 0;
    SET add_number = 100;
    SET customer_cnt = (SELECT COUNT(*) FROM customer);
    SELECT customer_cnt + add_number;
END $$
DELIMITER ;
```

■ 스토어드 프로시저 호출하기

- 생성된 프로시저를 호출하기 위해서는 다음과 같이 CALL 문을 사용함

Do it! 스토어드 프로시저 호출

```
CALL doit_proc();
```

실행 결과

	customer_cnt + add_number
▶	699

- 이 쿼리를 통해 customer 테이블의 행 개수와 add_number에 저장된 100을 더한 결과인 699를 반환하는 것을 확인할 수 있음

■ 스토어드 프로시저 내용 확인하기

- 앞서 생성한 스토어드 프로시저의 내용을 확인하기 위해서는 SHOW CREATE PROCEDURE 명령문을 사용함

Do it! 스토어드 프로시저 내용 확인

```
SHOW CREATE PROCEDURE doit_proc;
```

실행 결과

	Procedure	sql_mode	Create Procedure	character_set_client	collation_connection	Database Collation
▶	doit_proc	ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLE...	CREATE DEFINER='root'@'localhost' PROCEDURE...	utf8mb4	utf8mb4_0900_ai_ci	utf8mb4_0900_ai_ci

■ 스토어드 프로시저 내용 확인하기

- 표시된 부분을 살펴보면 다음과 같이 생성된 프로시저의 코드를 보여 줌

```
CREATE DEFINER=`root` @`localhost` PROCEDURE `doit_proc`()
BEGIN
    DECLARE customer_cnt INT;
    DECLARE add_number INT;

    SET customer_cnt = 0;
    SET add_number = 100;

    SET customer_cnt = (SELECT COUNT(*) FROM customer);

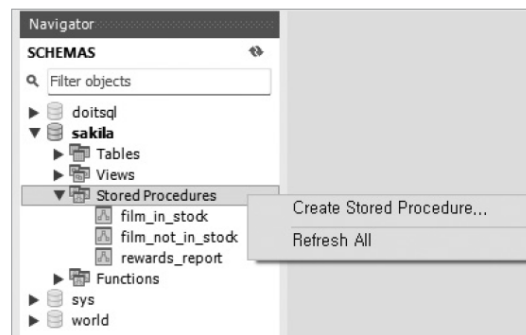
    SELECT customer_cnt + add_number;
END
```


■ 스토어드 프로시저 삭제하기

Do it! 스토어드 프로시저 삭제

```
DROP PROCEDURE doit_proc;
```

- 스토어드 프로시저가 삭제된 것을 확인하려면 프로시저를 CALL 문으로 호출해 보자
- 그 결과, 프로시저가 없다며 오류를 발생시킬 것임
- 또는 워크벤치 내비게이터에서 [Stored Procedures]를 확장해 보면 doit_proc 프로시저가 삭제된 것을 확인할 수 있음



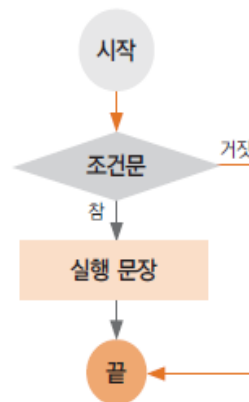
프로시저 삭제 확인 화면

- MySQL에서도 다른 프로그래밍 언어처럼 조건에 따라 선택적으로 명령을 실행하는 IF 문이나 CASE 문 또는 명령을 반복해 사용할 수 있는 WHILE 문, 그리고 다양한 조건을 결합하여 새로운 SQL 문을 작성하여 실행시킬 수 있는 동적 SQL 등을 활용해 다양하게 프로그래밍할 수 있음
- 가장 자주 사용하는 몇 가지 명령문을 알아보자
- MySQL에서는 일부 프로그래밍 명령문은 스토어드 프로시저로 작성해야 동작하는 것들이 있음

IF 문

IF 문의 기본 형식

IF 조건식 THEN (조건식이 참일 때) 실행할 식
ELSE (조건식이 거짓일 때) 실행할 식
END IF;



IF 문의 알고리즘 순서도

1. SELECT 문에 IF 문을 포함해 데이터를 조회.

Do it! IF 문을 활용한 데이터 조회

```
SELECT store_id, IF(store_id = 1, '일', '이') AS one_two  
FROM customer GROUP BY store_id;
```

실행 결과

	store_id	one_two
▶	1	일
	2	이

IF 문 조건에 따라 store_id=1이면 '일', store_id=2이면 '이'라는 글자가 출력

▪ IF 문

2. 복잡한 조건문으로 명령을 실행하려면 스토어드 프로시저로 생성해야 함.
다음 스토어드 프로시저는 store_id가 1이면 변수 s_id_one에 1~~씩 더하고~~,
store_id가 1이 아니면 변수 s_id_two에 1~~씩 더함~~ 2를 준다.

Do it! IF 문을 실행하기 위한 스토어드 프로시저 생성

```
DROP PROCEDURE IF EXISTS doit_if;
DELIMITER $$
CREATE PROCEDURE doit_if (customer_id_input INT)
BEGIN
    DECLARE store_id_i INT;
    DECLARE s_id_one INT;
    DECLARE s_id_two INT;
    SET store_id_i = (SELECT store_id FROM customer WHERE customer_id = customer_id_input);
    IF store_id_i = 1 THEN SET s_id_one = 1;
    ELSE SET s_id_two = 2;
    END IF;
    SELECT store_id_i, s_id_one, s_id_two;
END $$
DELIMITER ;
```

▪ IF 문

3. 다음 쿼리를 통해 프로시저를 호출해 보자.

Do it! 스토어드 프로시저 실행

```
CALL doit_if(1);
```

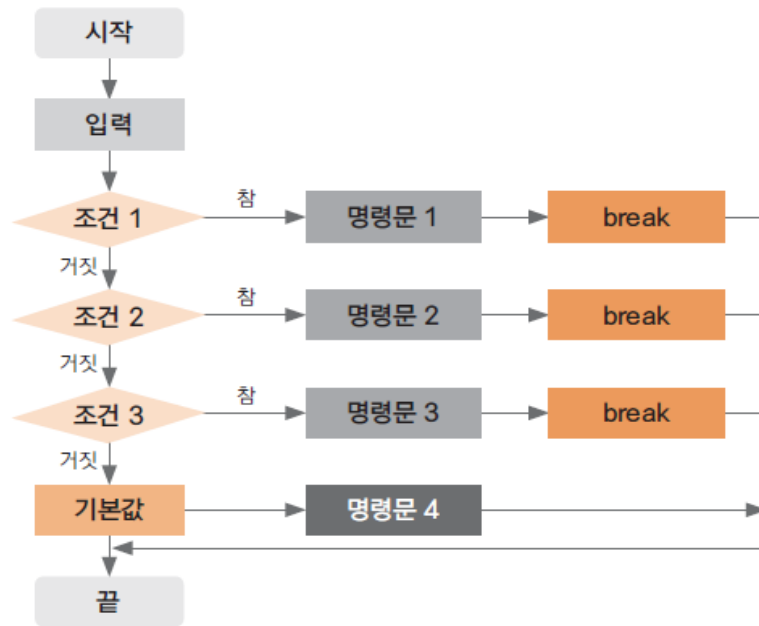
실행 결과

	store_id_i	s_id_one	s_id_two
▶	1	1	NULL

스토어드 프로시저를 호출할 때 입력 매개변수로 1을 입력하였고
스토어드 프로시저를 활용하면 입력받은 매개변수와 동일한
customer_id를 검색해 store_id 값을 store_id_i 변수에 할당함.

그리고 s_id_one에 1이라는 값이 할당되어 1이 조회되었고,
s_id_two는 어떠한 값도 할당되지 않았기 때문에 NULL이 조회됨

■ CASE 문



CASE 문의 알고리즘 순서도

CASE 문의 기본 형식

CASE

```
WHEN 조건 1 THEN 명령문 1
WHEN 조건 2 THEN 명령문 2
WHEN 조건 3 THEN 명령문 3
ELSE 명령문 4
```

END

■ CASE 문

1. custom_id 열을 그룹화하고 그룹별로 amount값을 합산한 뒤, 이를 바탕으로 회원 등급을 VVIP, VIP, GOLD, SILVER 4가지로 분류

Do it! CASE 문을 활용한 데이터 조회

```
SELECT customer_id, SUM(amount) AS amount,  
       CASE  
           WHEN SUM(amount) >= 150 THEN 'VVIP'  
           WHEN SUM(amount) >= 120 THEN 'VIP'  
           WHEN SUM(amount) >= 100 THEN 'GOLD'  
           WHEN SUM(amount) >= 80 THEN 'SILVER'  
           ELSE 'BRONZE'  
       END AS customer_level  
FROM payment GROUP BY customer_id;
```

실행 결과

	customer_id	amount	customer_level
▶	1	118.68	GOLD
	2	128.73	VIP
	3	135.74	VIP
	4	81.78	SILVER
	5	144.62	VIP
	6	93.72	SILVER
	7	151.67	VVIP

■ CASE 문

2. 앞서 실습한 쿼리를 활용하여 이번에는 스토어드 프로시저로 만들어 보자.
특정 고객 번호(customer_id)를 넣었을 때 해당 고객의 등급만을 조회하는 기능

Do it! CASE 문을 실행하기 위한 스토어드 프로시저 생성

```
DROP PROCEDURE IF EXISTS doit_case;
DELIMITER $$
CREATE PROCEDURE doit_case (customer_id_input INT)
BEGIN
    DECLARE customer_level VARCHAR(10);
    DECLARE amount_sum float;
    SET amount_sum = (SELECT SUM(amount) FROM payment WHERE customer_id = customer_id_input GROUP BY
customer_id);
    CASE
        WHEN amount_sum >= 150 THEN SET customer_level = 'VVIP';
        WHEN amount_sum >= 120 THEN SET customer_level = 'VIP';
        WHEN amount_sum >= 100 THEN SET customer_level = 'GOLD';
        WHEN amount_sum >= 80 THEN SET customer_level = 'SILVER';
        ELSE SET customer_level = 'BRONZE';
    END CASE;
    SELECT customer_id_input as customer_id, amount_sum, customer_level;
END $$
DELIMITER ;
```


■ CASE 문

3.

Do it! 스토어드 프로시저 실행

```
CALL doit_case(4);
```

실행 결과

	customer_id	amount_sum	customer_level
▶	4	81.78	SILVER

프로시저의 입력 매개변수로 4를 입력하였고,
즉 customer_id=4를 호출하며 이에 해당하는 amount=81.78임.

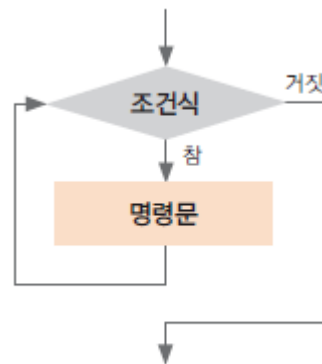
또한 이에 대한 고객 등급으로 SILVER가 출력된 것을 확인할 수 있음.
시험 삼아 다른 고객 번호를 넣어 등급을 확인해 보자

■ WHILE 문

- 반복문으로 사용자가 지정한 조건이 될 때까지 같은 내용을 반복함

WHILE 문의 기본 형식

```
WHILE 조건식 DO 명령문  
END WHILE;
```



WHILE 문의 알고리즘 순서도

■ WHILE 문

1. 첫 번째 매개변수 값을 두 번째 매개변수 값만큼 반복해서 더함

Do it! WHILE 문을 실행하기 위한 스토어드 프로시저 생성

```
DROP PROCEDURE IF EXISTS doit_while;
DELIMITER $$
CREATE PROCEDURE doit_while (param_1 INT, param_2 INT)
BEGIN
    DECLARE i INT;
    DECLARE while_sum INT;
    SET i = 1;
    SET while_sum = 0;
    WHILE (i <= param_1) DO
        SET while_sum = while_sum + param_2;
        SET i = i + 1;
    END WHILE;
    SELECT while_sum;
END $$
DELIMITER ;
```

Do it! 스토어드 프로시저 실행

```
CALL doit_while(10, 3);
```

실행 결과

	while_sum
▶	30

▪ WHILE 문

3. LEAVE 문은 반복문을 실행할 때, 특정 조건이 되면 반복문을 빠져나오게 하는 명령어임. 반복문이 실행되다가 100보다 크면 LEAVE를 통해 반복문을 빠져나오는 쿼리임

Do it! WHILE ~ LEAVE 문을 실행하기 위한 스토어드 프로시저 생성

```
DROP PROCEDURE IF EXISTS doit_while;
DELIMITER $$
CREATE PROCEDURE doit_while (param_1 INT, param_2 INT)
BEGIN
    DECLARE i INT;
    DECLARE while_sum INT;
    SET i = 1;
    SET while_sum = 0;
    myWhile:
    WHILE (i <= param_1) DO
        SET while_sum = while_sum + param_2;
        SET i = i + 1;
        IF (while_sum > 100) THEN LEAVE myWhile;
        END IF;
    END WHILE;
    SELECT while_sum;
END $$
DELIMITER ;
```

Do it! 스토어드 프로시저 실행

CALL doit_while(1000, 3);

실행 결과

	while_sum
▶	102

■ 동적 SQL

- 지금까지는 쿼리를 작성하거나 스토어드 프로시저를 활용한 쿼리를 사용할 때, 미리 정의된 로직에 따라 쿼리가 수행되며, 새로운 조건이 발생하더라도 해당 조건에 맞게 분기되거나 필터링되는 정도로만 동작함
- 만약 상황에 따라 쿼리문을 조합하여 실행할 수 있다고 한다면 여러 상황에서 더 다양한 명령을 실행할 수 있지 않을까?
- 예를 들어 보고서를 작성하기 위해 데이터를 추출하는데 SELECT 문이나 참조하는 테이블을 다르게 입력할 수 있다면 얼마나 편리하겠는가?

▪ 동적 SQL

- 변수값을 할당 받아 MySQL 서버 내부 또는 스토어드 프로시저에서 쿼리를 재작성하는 것을 동적 SQL이라고 함
- PREPARE 문으로 쿼리문을 준비하고, EXECUTE 문으로 쿼리를 실행함
- DEALLOCATE PREPARE 문으로 쿼리문을 해제할 수 있음

동적 SQL의 기본 형식

```
PREPARE 동적 쿼리명 FROM '쿼리 작성'  
EXECUTE 동적 쿼리명  
DEALLOCATE PREPARE 동적 쿼리명
```

■ 동적 SQL

1. customer 테이블에서 조건에 맞는 customer_id를 가진 사용자 정보를 조회

Do it! 동적 SQL을 활용한 데이터 조회

```
PREPARE dynamic_query FROM 'SELECT * FROM customer WHERE customer_id = ?';  
SET @a = 1;  
EXECUTE dynamic_query USING @a;  
DEALLOCATE PREPARE dynamic_query;
```

실행 결과

	customer_id	store_id	first_name	last_name	email	address_id	active	create_date	last_update
▶	1	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	5	1	2006-02-14 22:04:36	2006-02-15 04:57:20

■ 동적 SQL

2. 스토어드 프로시저로 만들되, 입력값에 따라 다양한 테이블을 조회할 수 있는 쿼리로 작성

Do it! 동적 SQL을 실행하기 위한 스토어드 프로시저 생성

```
DROP PROCEDURE IF EXISTS doit_dynamic;
DELIMITER $$
CREATE PROCEDURE doit_dynamic (t_name VARCHAR(50), c_name VARCHAR(50), customer_id INT)
BEGIN
    SET @t_name = t_name;
    SET @c_name = c_name;
    SET @customer_id = customer_id;
    SET @sql = CONCAT('SELECT ', @c_name, ' FROM ', @t_name, ' WHERE customer_id = ', @customer_id);
    SELECT @sql;
    PREPARE dynamic_query FROM @sql;
    EXECUTE dynamic_query;
    DEALLOCATE PREPARE dynamic_query;
END $$
DELIMITER ;
```

Do it! 스토어드 프로시저 실행

```
CALL doit_dynamic('payment', '*', 1);
```

실행 결과

payment_id	customer_id	staff_id	rental_id	amount	payment_date	last_update
1	1	1	76	2.99	2005-05-25 11:30:37	2006-02-15 22:12:30
2	1	1	573	0.99	2005-05-28 10:35:23	2006-02-15 22:12:30
3	1	1	1185	5.99	2005-06-15 00:54:12	2006-02-15 22:12:30
4	1	2	1422	0.99	2005-06-15 18:02:53	2006-02-15 22:12:30
5	1	2	1476	9.99	2005-06-15 21:08:46	2006-02-15 22:12:30
6	1	1	1725	4.99	2005-06-16 15:18:57	2006-02-15 22:12:30
7	1	1	2308	4.99	2005-06-18 08:41:48	2006-02-15 22:12:30

- 저장된 수많은 데이터 중에 특정 조건을 만족하는 데이터를 조회할 때마다 모든 데이터를 일일이 검사해서 필요한 데이터만 찾아야 한다면 매우 많은 조회 비용이 발생함
- 이는 데이터베이스의 응답이 느려진다는 것으로, 성능이 저하됨을 의미함
- 이러한 문제를 해결하기 위해 사용하는 기술이 인덱스(index)
- 인덱스는 필요한 데이터를 바로 찾을 수 있도록 참고할 수 있는 데이터임
- 인덱스는 열 단위로 지정할 수 있음
- 예를 들어 책에는 차례나 찾아보기가 있어서 필요한 내용을 빠르게 찾아갈 수 있듯이, 원하는 데이터를 빠르게 찾아갈 수 있도록 하는 기술이 데이터베이스의 인덱스임

■ 인덱스를 사용할 때 주의할 점

장점	단점
<ul style="list-style-type: none">• 원하는 데이터를 빠르게 검색할 수 있다.• 불필요한 검색 비용을 절약하고 I/O 성능을 높일 수 있다.• 데이터 검색뿐만 아니라 조인 시에도 빠른 성능을 얻을 수 있다. 특히 조인에서 데이터가 폭발적으로 증가할 수 있으므로 인덱스가 없다면 매우 느린 성능을 보여 준다.	<ul style="list-style-type: none">• 인덱스를 별도로 저장하는 인덱스 페이지를 구성하기 위해 추가 공간이 필요하다.• 데이터를 수정할 때 연결된 인덱스 정보도 함께 수정해야 하는 경우 추가 비용이 발생한다.• 인덱스 정보를 수정할 때 잠금이 발생해 데이터베이스 성능이 느려질 수 있다.

■ 인덱스의 종류

- 클러스터형 인덱스 - 사전식으로 데이터가 정렬되어 있고
인덱스 안에 데이터가 들어 있어 빠르게 데이터를 조회할 수 있음
- 비클러스터형 인덱스 - 목차식으로 차례에 적혀 있는 제목과 쪽 번호를 보고
필요한 곳을 찾아가서 본문을 확인하는 것처럼
데이터의 위치 정보를 인덱스가 가지고 있는 형태이며,
실제 데이터를 가지고 있지는 않고 주소(데이터의 위치 정보)만 관리함

■ 인덱스의 종류

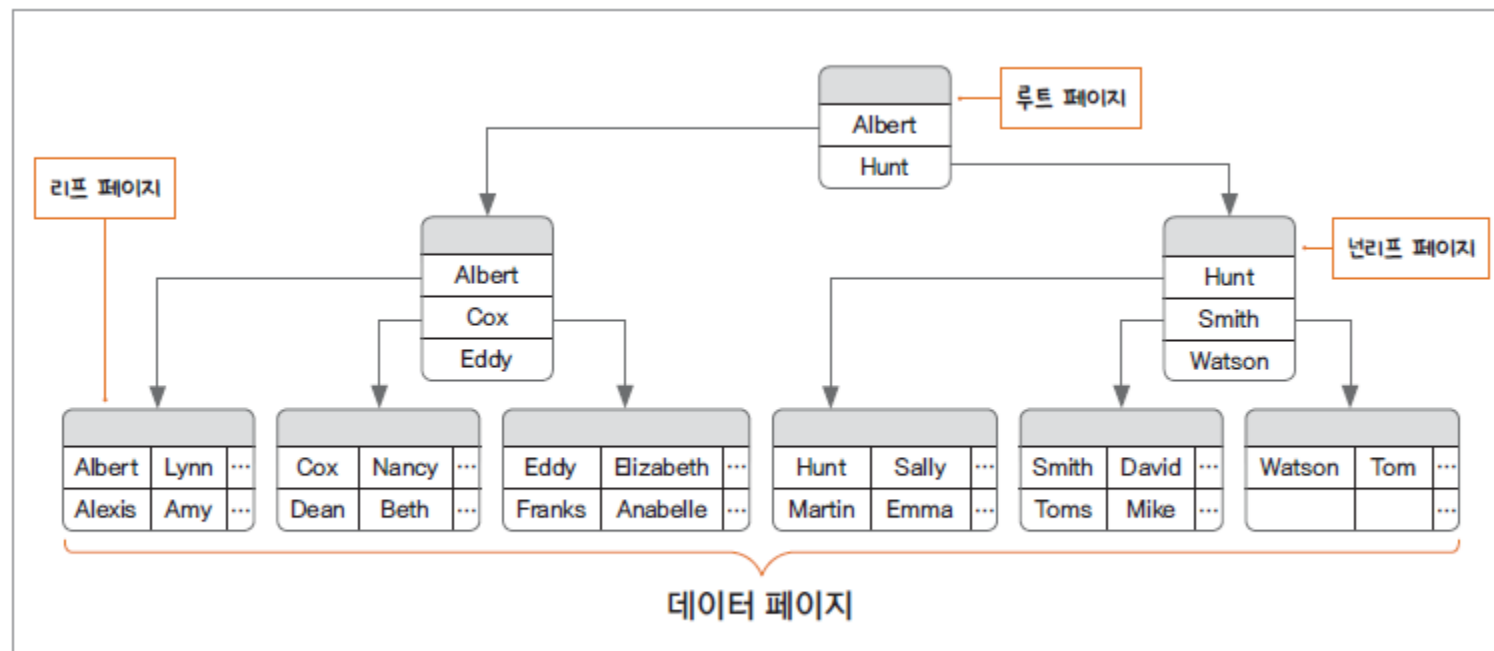
- **클러스터형 인덱스**(clustered index)
인덱스를 기준으로 데이터가 자동으로 정렬되어 저장되고,
인덱스의 리프 페이지에는 데이터가 존재하게 됨

클러스터형 인덱스의 특징

- 테이블당 1개만 존재할 수 있다.
- 기본키로 지정된 열은 클러스터형 인덱스가 자동으로 생성된다.
- 실제 저장된 데이터와 같은 순서로 물리적인 데이터 페이지 구조를 갖는다.
- 클러스터형 인덱스를 기준으로 데이터가 자동으로 정렬된다.
- 기본키를 변경하면 클러스터형 인덱스가 변경되므로 변경된 기본키를 기준으로 데이터가 다시 자동 정렬된다.

■ 인덱스의 종류

■ 클러스터형 인덱스



클러스터형 인덱스 구조를 표현한 예

■ 인덱스의 종류

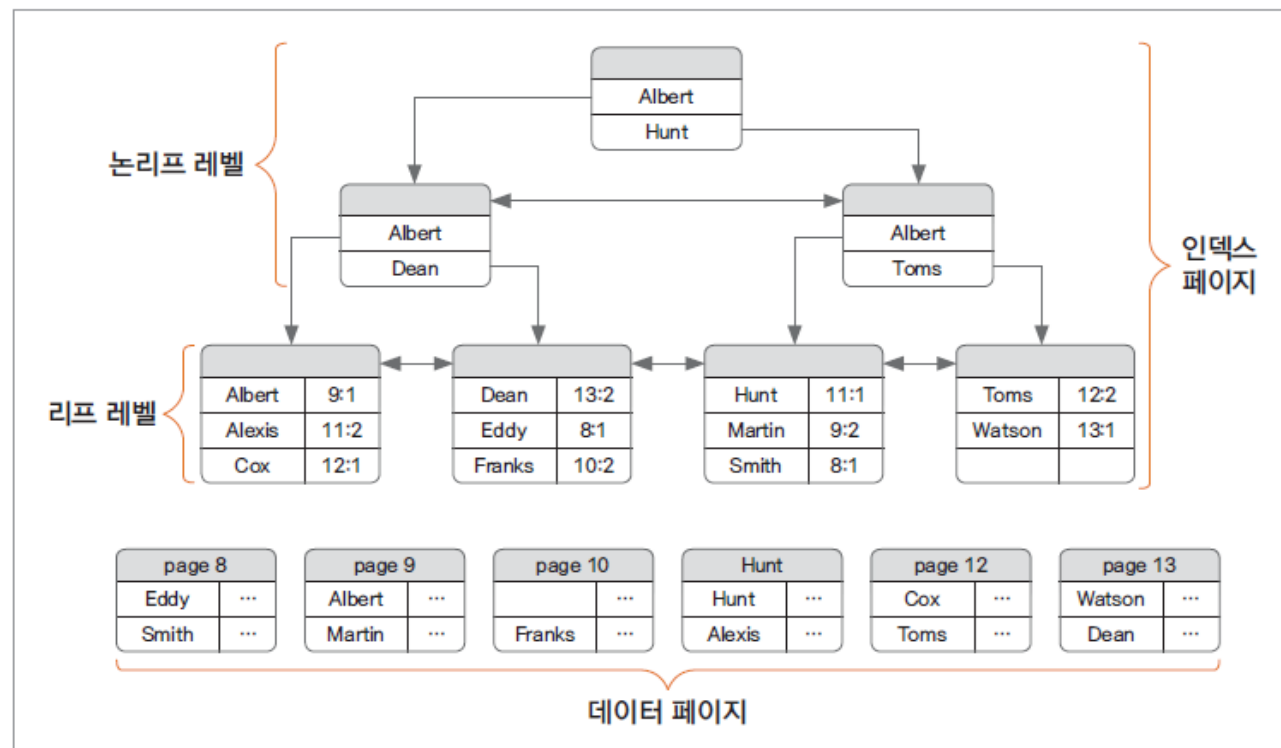
- **비클러스터형 인덱스**(non-clustered index 또는 secondary index)
실제 데이터가 위치한 주소 정보로 인덱스 페이지를 별도로 생성하여 사용하며 데이터를 정렬하지 않음

비클러스터형 인덱스의 특징

- 한 테이블에 여러 개 설정할 수 있다.
- UNIQUE 명령어로 고유 열을 지정할 때 비클러스터형 인덱스(보조 인덱스)가 자동으로 생성된다.
- 실제 저장된 데이터와 다른 물리적인 데이터 페이지 구조를 갖는다.
- 클러스터형 인덱스와 달리 데이터를 정렬하지 않는다.
- CREATE INDEX 문으로 비클러스터형 인덱스(보조 인덱스)를 직접 생성할 수 있다.

■ 인덱스의 종류

- 리프 페이지에는 실제 데이터의 주소인 RID(Row Identifier)값을 가지고 있으며, 루트와 논리프 노드는 인덱스 정보를 가지고 있음.
- 리프 페이지의 주소를 참조하여 실제 데이터 페이지로 가서 데이터의 정보를 읽는 방식임



비클러스터형 인덱스를 표현한 예

■ 인덱스 생성 및 삭제하기

- 인덱스 생성 실습을 진행하기 위해
실습용 테이블과 데이터를 먼저 생성해 보자
- 여기에서는 클러스터형과 비클러스터형 인덱스의 특징을 각각 알 수 있도록
동일한 형식의 테이블과 데이터를 두 세트 준비함
- 그리고 인덱스를 만들면서 어떤 변화가 발생하는지 살펴보자

■ 인덱스 생성 및 삭제하기

■ 클러스터형 인덱스 생성과 삭제 방법

- 클러스터형 인덱스를 기본키로 지정하면 자동으로 생성되고 클러스터형 인덱스는 테이블당 1개만 생성됨
- 클러스터형 인덱스의 가장 큰 특징은 데이터가 자동으로 인덱스 열에 따라 정렬된다는 것
- 국어 사전을 떠올려 보면 가나다순으로 단어들이 정렬되어 적혀 있음.
마찬가지로 클러스터형 인덱스에 따라 데이터들이 이러한 모습으로 정렬되어 저장되는 것임

■ 인덱스 생성 및 삭제하기

■ 클러스터형 인덱스 생성과 삭제 방법

1. 먼저 테이블을 생성하고 데이터를 입력해 보자

Do it! 실습을 위한 테이블과 데이터 생성

```
USE doitsql;  
DROP TABLE IF EXISTS doit_clusterindex;  
CREATE TABLE doit_clusterindex (  
  col_1 INT,  
  col_2 VARCHAR(50),  
  col_3 VARCHAR(50)  
);  
INSERT INTO doit_clusterindex VALUES (2, '사자', 'lion');  
INSERT INTO doit_clusterindex VALUES (5, '호랑이', 'tiger');  
INSERT INTO doit_clusterindex VALUES (3, '얼룩말', 'zbera');  
INSERT INTO doit_clusterindex VALUES (4, '코뿔소', 'Rhinoceros');  
INSERT INTO doit_clusterindex VALUES (1, '거북이', 'turtle');  
SELECT * FROM doit_clusterindex;
```

실행 결과

	col_1	col_2	col_3
▶	2	사자	lion
	5	호랑이	tiger
	3	얼룩말	zbera
	4	코뿔소	Rhinoceros
	1	거북이	turtle

■ 인덱스 생성 및 삭제하기

■ 클러스터형 인덱스 생성과 삭제 방법

2. col_1 열에 기본키를 지정하고
col_1 순서로 데이터가 정렬되어 저장되는 것을 확인해 보자.

Do it! 기본키(기본 인덱스) 생성

```
ALTER TABLE doit_clusterindex  
ADD CONSTRAINT PRIMARY KEY  
(col_1);  
SELECT * FROM doit_clusterindex;
```

실행 결과

	col_1	col_2	col_3
▶	1	거북이	turtle
	2	사자	lion
	3	얼룩말	zbera
	4	코뿔소	Rhinoceros
	5	호랑이	tiger

결과를 살펴보면 기본키이자 인덱스로 지정한 열 col_1을 기준으로
오름차순으로 데이터가 정렬된 것을 확인할 수 있음

■ 인덱스 생성 및 삭제하기

■ 클러스터형 인덱스 생성과 삭제 방법

3. 새로 추가된 데이터는 어떻게 되는지 다음과 같이 입력해 보자.

Do it! 새로운 데이터 입력

```
INSERT INTO doit_clusterindex VALUES (0, '물고기', 'fish');
```

```
SELECT * FROM doit_clusterindex;
```

실행 결과

	col_1	col_2	col_3
▶	0	물고기	fish
	1	거북이	turtle
	2	사자	lion
	3	얼룩말	zbera
	4	코뿔소	Rhinoceros
	5	호랑이	tiger

새롭게 입력된 데이터 또한 기본키를 기준으로 저장된 것을 확인할 수 있음

■ 인덱스 생성 및 삭제하기

■ 클러스터형 인덱스 생성과 삭제 방법

4. 기존에 생성된 기본키를 삭제하고 col_2 기준으로 기본키를 생성한 뒤 데이터를 조회해 보자.

Do it! col_2 열로 인덱스 변경

```
ALTER TABLE doit_clusterindex  
    DROP PRIMARY KEY,  
    ADD CONSTRAINT PRIMARY KEY doit_clusterindex (col_2);  
  
SELECT * FROM doit_clusterindex;
```

실행 결과

	col_1	col_2	col_3
▶	1	거북이	turtle
	0	물고기	fish
	2	사자	lion
	3	얼룩말	zbera
	4	코뿔소	Rhinoceros
	5	호랑이	tiger

col_2 열을 인덱스로 지정한 결과, 데이터가 가나다순으로 정렬된 것을 확인할 수 있음

■ 인덱스 생성 및 삭제하기

■ 클러스터형 인덱스 생성과 삭제 방법

5. 영문 데이터가 있는 col_3 열을 기본키로 지정해 보자.

Do it! col_3 열로 인덱스 변경

```
ALTER TABLE doit_clusterindex  
    DROP PRIMARY KEY,  
    ADD CONSTRAINT PRIMARY KEY (col_3);  
  
SELECT * FROM doit_clusterindex;
```

실행 결과

	col_1	col_2	col_3
▶	0	물고기	fish
	2	사자	lion
	4	코뿔소	Rhinoceros
	5	호랑이	tiger
	1	거북이	turtle
	3	얼룩말	zbera

col_3 열을 인덱스로 지정한 결과, 데이터가 알파벳순으로 정렬된 것을 확인할 수 있음

■ 인덱스 생성 및 삭제하기

■ 클러스터형 인덱스 생성과 삭제 방법

6. 열을 2개 이상 합한 인덱스를 만들어 보자.
 이러한 인덱스를 복합키(composite key) 인덱스라고 하는데,
 인덱스의 순서에 따라 인덱스 정렬이 발생하므로 열의 순서가 매우 중요함

Do it! 복합키 인덱스 생성

```
ALTER TABLE doit_clusterindex
  DROP PRIMARY KEY,
  ADD CONSTRAINT PRIMARY KEY(col_1, col_3);
SHOW INDEX FROM doit_clusterindex;
```

실행 결과

	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
▶	doit_clusterindex	0	PRIMARY	1	col_1	A	6	NULL	NULL		BTREE			YES	NULL
	doit_clusterindex	0	PRIMARY	2	col_3	A	6	NULL	NULL		BTREE			YES	NULL

■ 인덱스 생성 및 삭제하기

■ 클러스터형 인덱스 생성과 삭제 방법

7. 생성한 인덱스들을 삭제하려면 DROP 문을 사용함.

Do it! 인덱스 삭제

```
ALTER TABLE doit_clusterindex DROP PRIMARY KEY;  
SHOW INDEX FROM doit_clusterindex;
```

실행 결과

	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_com
--	-------	------------	----------	--------------	-------------	-----------	-------------	----------	--------	------	------------	---------	-----------

■ 인덱스 생성 및 삭제하기

■ 비클러스터형 인덱스 생성과 삭제 방법

1. 비클러스터형 인덱스는 데이터를 정렬해서 저장하진 않고 별도의 인덱스 페이지에 데이터 주소를 정렬하여 저장한다.

Do it! 실습을 위한 테이블과 데이터 생성

```
USE doitsql;  
  
DROP TABLE IF EXISTS doit_nonclusterindex;  
CREATE TABLE doit_nonclusterindex (  
col_1 INT,  
col_2 VARCHAR(50),  
col_3 VARCHAR(50)  
);  
INSERT INTO doit_nonclusterindex VALUES (2, '사자', 'lion');  
INSERT INTO doit_nonclusterindex VALUES (5, '호랑이', 'tiger');  
INSERT INTO doit_nonclusterindex VALUES (3, '얼룩말', 'zbera');  
INSERT INTO doit_nonclusterindex VALUES (4, '코뿔소', 'Rhinoceros');  
INSERT INTO doit_nonclusterindex VALUES (1, '거북이', 'turtle');  
  
SELECT * FROM doit_nonclusterindex;
```

실행 결과

	col_1	col_2	col_3
▶	2	사자	lion
	5	호랑이	tiger
	3	얼룩말	zbera
	4	코뿔소	Rhinoceros
	1	거북이	turtle

■ 인덱스 생성 및 삭제하기

■ 비클러스터형 인덱스 생성과 삭제 방법

2. col_1 기준으로 비클러스터형 인덱스를 생성해 보자.

Do it! 비클러스터형 인덱스 생성

```
CREATE INDEX ix_doit_nonclusterindex_1 ON doit_nonclusterindex  
(col_1);  
SELECT * FROM doit_nonclusterindex;
```

실행 결과

	col_1	col_2	col_3
▶	2	사자	lion
	5	호랑이	tiger
	3	얼룩말	zbera
	4	코뿔소	Rhinoceros
	1	거북이	turtle

결과를 살펴보면 인덱스를 생성했지만
인덱스 열(여기선 col_1)을 기준으로 데이터가 정렬되지 않은 것을 확인할 수 있음

■ 인덱스 생성 및 삭제하기

■ 비클러스터형 인덱스 생성과 삭제 방법

3. 데이터를 하나 추가해 보자.
데이터가 추가되어도 입력된 순서대로 저장되고 정렬되지는 않음

Do it! 새로운 데이터 입력

```
INSERT INTO doit_nonclusterindex VALUES (0, '물고기', 'fish');  
SELECT * FROM doit_nonclusterindex;
```

실행 결과

	col_1	col_2	col_3
▶	2	사자	lion
	5	호랑이	tiger
	3	얼룩말	zbera
	4	코뿔소	Rhinoceros
	1	거북이	turtle
	0	물고기	fish

■ 인덱스 생성 및 삭제하기

■ 비클러스터형 인덱스 생성과 삭제 방법

4. 각 열을 모두 비클러스터형 인덱스로 만들어 보자.

Do it! 각 열별로 인덱스 생성

```
CREATE INDEX ix_doit_nonclusterindex_2 ON doit_nonclusterindex (col_2);  
CREATE INDEX ix_doit_nonclusterindex_3 ON doit_nonclusterindex (col_3);  
  
SELECT * FROM doit_nonclusterindex;
```

실행 결과

	col_1	col_2	col_3
▶	2	사자	lion
	5	호랑이	tiger
	3	얼룩말	zbera
	4	코뿔소	Rhinoceros
	1	거북이	turtle
	0	물고기	fish

인덱스가 여러 개 생성되었지만 각 열의 데이터 정렬 순서는 여전히 변함이 없음

■ 인덱스 생성 및 삭제하기

■ 비클러스터형 인덱스 생성과 삭제 방법

5. 열을 2개 이상 묶어서 인덱스를 생성해 보자

Do it! 복합키 인덱스 생성

```
CREATE INDEX ix_doit_nonclusterindex_1_2 ON doit_nonclusterindex (col_1, col_2);
CREATE INDEX ix_doit_nonclusterindex_1_3 ON doit_nonclusterindex (col_1, col_3);
```

```
SHOW INDEX FROM doit_nonclusterindex;
```

실행 결과

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index
doit_nonclusterindex	1	ix_doit_nonclusterindex_1	1	col_1	A	5	NULL	NULL	YES	BTREE
doit_nonclusterindex	1	ix_doit_nonclusterindex_2	1	col_2	A	6	NULL	NULL	YES	BTREE
doit_nonclusterindex	1	ix_doit_nonclusterindex_3	1	col_3	A	6	NULL	NULL	YES	BTREE
doit_nonclusterindex	1	ix_doit_nonclusterindex_1_2	1	col_1	A	6	NULL	NULL	YES	BTREE
doit_nonclusterindex	1	ix_doit_nonclusterindex_1_2	2	col_2	A	6	NULL	NULL	YES	BTREE
doit_nonclusterindex	1	ix_doit_nonclusterindex_1_3	1	col_1	A	6	NULL	NULL	YES	BTREE
doit_nonclusterindex	1	ix_doit_nonclusterindex_1_3	2	col_3	A	6	NULL	NULL	YES	BTREE

doit_nonclusterindex 테이블에 생성되어 있는 모든 인덱스의 이름과 해당 인덱스에 포함된 열의 정보 및 순서를 확인할 수 있음

■ 인덱스 생성 및 삭제하기

■ 비클러스터형 인덱스 생성과 삭제 방법

6. 생성된 인덱스들을 삭제하려면 DROP 문을 사용.

Do it! 인덱스 삭제

```
DROP INDEX ix_doit_nonclusterindex_1_2 ON doit_nonclusterindex;
DROP INDEX ix_doit_nonclusterindex_1_3 ON doit_nonclusterindex;
```

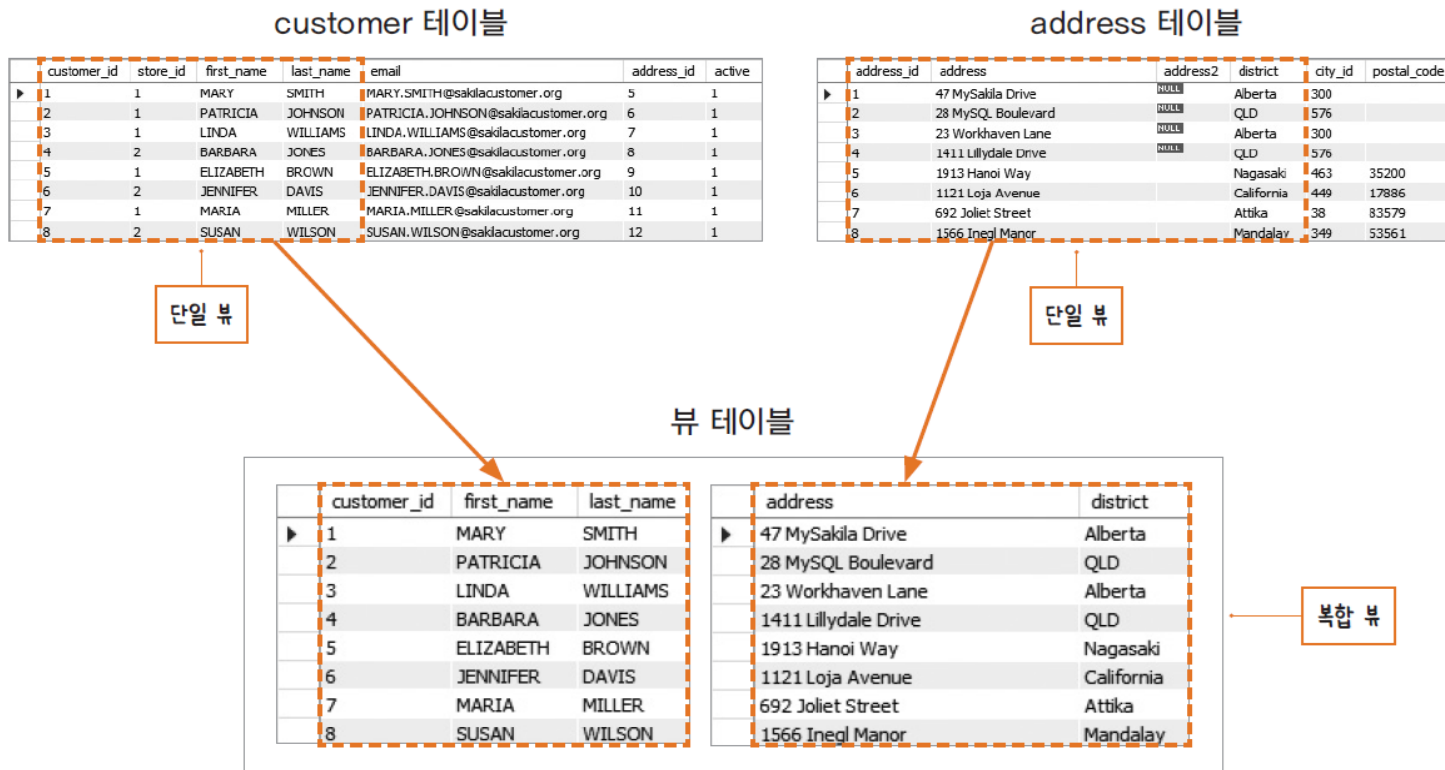
```
SHOW INDEX FROM doit_nonclusterindex;
```

실행 결과

	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Vis
▶	doit_nonclusterindex	1	ix_doit_nonclusterindex_1	1	col_1	A	5	NULL	NULL	YES	BTREE			YES
	doit_nonclusterindex	1	ix_doit_nonclusterindex_2	1	col_2	A	6	NULL	NULL	YES	BTREE			YES
	doit_nonclusterindex	1	ix_doit_nonclusterindex_3	1	col_3	A	6	NULL	NULL	YES	BTREE			YES

- 뷰(view)는 데이터베이스에 존재하는 가상의 테이블
- 뷰는 테이블이나 데이터를 직접 소유하지 않고
테이블의 형태만 차용하여 테이블처럼 사용할 수 있게 해주는 역할을 함
- 뷰는 SELECT 문으로 만들며,
실제 사용자가 뷰를 호출했을 때 SELECT 문이 실행되고 그 결과가 화면에 출력됨

- 뷰는 한 테이블만 구성할 수도 있고, (단일 뷰)
다수의 테이블로 조인하여 구성할 수도 있음 (복합 뷰)



단일 뷰와 복합 뷰 예

■ 뷰를 사용하는 이유

- 뷰는 편의성, 보안성, 유지 보수성 등의 장점이 있어 많이 사용함
- 예를 들어 회사에서 부서별로 필요한 고객 정보 외에 다른 정보를 보여주고 싶지 않을 때, 별도의 테이블을 만들어 데이터를 복사하는 방식이 아닌 부서별로 필요한 열로만 구성된 가상의 뷰 테이블을 사용하면 보안 및 편의성이 강화됨

장점	단점
<ul style="list-style-type: none">• 복잡한 쿼리를 단축시켜 놓기 때문에 사용자는 편리하게 사용할 뿐만 아니라 유지·보수에도 유리하다.• 테이블과 열 이름 등을 숨길 수 있으며, 권한에 따라 필요한 열만 구성해 사용할 수 있으므로 보안성이 우수하다.	<ul style="list-style-type: none">• 한 번 정의된 뷰는 변경할 수 없다.• 삽입, 삭제, 갱신 작업을 하는 데 제한 사항이 많다.• 뷰 테이블은 인덱스를 가질 수 없다.

■ 뷰 생성 및 조회하기

1.

뷰 생성 기본 형식

```
CREATE VIEW 뷰 이름  
AS  
    <SELECT 문>
```

Do it! 일부 열을 보여주는 뷰 생성

```
CREATE VIEW v_customer  
AS  
    SELECT first_name, last_name, email FROM customer;  
  
SELECT* FROM v_customer;
```

실행 결과

	first_name	last_name	email
▶	MARY	SMITH	MARY.SMITH@sakilacustomer.org
	PATRICIA	JOHNSON	PATRICIA.JOHNSON@sakilacustomer.org
	LINDA	WILLIAMS	LINDA.WILLIAMS@sakilacustomer.org
	BARBARA	JONES	BARBARA.JONES@sakilacustomer.org
	ELIZABETH	BROWN	ELIZABETH.BROWN@sakilacustomer.org
	JENNIFER	DAVIS	JENNIFER.DAVIS@sakilacustomer.org
	MARIA	MILLER	MARIA.MILLER@sakilacustomer.org

이렇게 뷰를 만들어서 사용하면
사용자는 실제 뷰가 내부적으로는 어떤 테이블의 데이터를 보여주는지 알 수 없음.
뷰에 대한 접근 권한만 부여하면 원래의 테이블에 대한 민감한 정보를 노출하지
않고 필요한 정보만 최소한으로 노출할 수 있음

■ 뷰 생성 및 조회하기

2. 이번에는 2개 이상의 테이블로 뷰를 만들어 보자.

생성한 뷰는 customer 테이블과 payment 테이블을 조인하여
결제 금액(amount)과 고객 정보(first_name, last_name, email)를 표시함

Do it! 2개의 테이블을 조인해 원하는 데이터를 보여주는 뷰 생성

```
CREATE VIEW v_payuser
AS
    SELECT first_name, last_name, email, amount, address_id
    FROM customer AS a
        INNER JOIN (SELECT customer_id, SUM(amount) AS amount FROM
payment
        GROUP BY customer_id) AS b ON a.customer_id = b.customer_id;
SELECT * FROM v_payuser;
```

■ 뷰 생성 및 조회하기

2.

실행 결과

	first_name	last_name	email	amount
▶	MARY	SMITH	MARY.SMITH@sakilacustomer.org	118.68
	PATRICIA	JOHNSON	PATRICIA.JOHNSON@sakilacustomer.org	128.73
	LINDA	WILLIAMS	LINDA.WILLIAMS@sakilacustomer.org	135.74
	BARBARA	JONES	BARBARA.JONES@sakilacustomer.org	81.78
	ELIZABETH	BROWN	ELIZABETH.BROWN@sakilacustomer.org	144.62
	JENNIFER	DAVIS	JENNIFER.DAVIS@sakilacustomer.org	93.72
	MARIA	MILLER	MARIA.MILLER@sakilacustomer.org	151.67

이와 같이 뷰를 생성할 때 일반 테이블을 조회하듯이 사용하며 됨.
즉, v_payuser 뷰를 조회할 때 WHERE 문, GROUP BY 문 등 모두 사용할 수 있음

■ 뷰 생성 및 조회하기

3. 뷰 테이블과 일반 테이블을 조인할 수도 있음.
조금 전 생성한 v_payuser 테이블과 일반 테이블을 조인하여 조회함

Do it! 뷰 테이블과 일반 테이블 조인

```
SELECT a.*, b.*
FROM v_payuser AS a
INNER JOIN address AS b ON a.address_id = b.address_id;
```

실행 결과

	first_name	last_name	email	amount	address_id	address_id	address	address2	district	city_id	postal
▶	MARY	SMITH	MARY.SMITH@sakilacustomer.org	118.68	5	5	1913 Hanoi Way		Nagasaki	463	35200
	PATRICIA	JOHNSON	PATRICIA.JOHNSON@sakilacustomer.org	128.73	6	6	1121 Loja Avenue		California	449	17886
	LINDA	WILLIAMS	LINDA.WILLIAMS@sakilacustomer.org	135.74	7	7	692 Joliet Street		Attika	38	83579
	BARBARA	JONES	BARBARA.JONES@sakilacustomer.org	81.78	8	8	1566 Inegl Manor		Mandalay	349	53561
	ELIZABETH	BROWN	ELIZABETH.BROWN@sakilacustomer.org	144.62	9	9	53 Idfu Parkway		Nantou	361	42399
	JENNIFER	DAVIS	JENNIFER.DAVIS@sakilacustomer.org	93.72	10	10	1795 Santiago de Compostela Way		Texas	295	18743

■ 뷰 수정하기

1.

뷰 수정 기본 형식

ALTER VIEW 뷰 이름

AS

<SQL 문>

v_customer 뷰를 수정함.
first_name, last_name,
email을 반환하는 뷰에서
customer_id, first_name,
last_name, email,
address_id를 반환하도록
작성함.

Do it! 뷰 수정

ALTER VIEW v_customer

AS

SELECT customer_id, first_name, last_name, email, address_id
FROM customer;

SELECT * FROM v_customer;

실행 결과

	customer_id	first_name	last_name	email	address_id
▶	1	MARY	SMITH	MARY.SMITH@sakilacustomer.org	5
	2	PATRICIA	JOHNSON	PATRICIA.JOHNSON@sakilacustomer.org	6
	3	LINDA	WILLIAMS	LINDA.WILLIAMS@sakilacustomer.org	7
	4	BARBARA	JONES	BARBARA.JONES@sakilacustomer.org	8
	5	ELIZABETH	BROWN	ELIZABETH.BROWN@sakilacustomer.org	9
	6	JENNIFER	DAVIS	JENNIFER.DAVIS@sakilacustomer.org	10

■ 뷰 수정하기

2. 만약 뷰가 없는 상태에서
뷰를 수정하려고 시도하면 당연히 오류가 발생함.
또한 무작정 CREATE VIEW를 하면
뷰가 이미 생성되어 있을 수 있어 오류가 발생함.

이때 다음 구문을 사용하면
기존 생성된 뷰를 대체하거나 존재하지 않을 때에도
뷰를 생성할 수 있음

뷰 생성 또는 교체 기본 형식

```
CREATE OR REPLACE VIEW 뷰 이름  
AS
```

```
<SQL 문>
```

■ 뷰 수정하기

2. v_customer이라는 뷰를 생성하는데,
이때, v_customer이라는 뷰가 존재하면
동일한 이름으로 '뷰가 이미 있으면 수정, 없으면 생성'으로 교체함.

Do it! v_customer 뷰 생성 및 교체

```
CREATE OR REPLACE VIEW v_customer  
AS  
    SELECT '뷰가 이미 있으면 수정, 없으면 생성';  
SELECT * FROM v_customer;
```

실행 결과

	뷰가 이미 있으면 수정, 없으면 생성
▶	뷰가 이미 있으면 수정, 없으면 생성

■ 뷰 정보 확인하기

- 내가 아닌 다른 사람이 뷰를 만들었을 경우
해당 뷰가 어떤 열을 가지고 있고, 데이터 유형은 무엇인지 파악이 필요함.

1. 뷰 정보를 확인하려면 다음과 같이 DESCRIBE 문을 사용함.
DESCRIBE과 함께 뷰 이름을 입력해 보자.

Do it! 뷰 정보 확인

```
DESCRIBE v_payuser;
```

실행 결과

	Field	Type	Null	Key	Default	Extra
▶	first_name	varchar(45)	NO		NULL	
	last_name	varchar(45)	NO		NULL	
	email	varchar(50)	YES		NULL	
	amount	decimal(27,2)	YES		NULL	
	address_id	smallint unsigned	NO		NULL	

이를 통해 뷰에 포함되어 있는 열의 이름 및 데이터 유형 정보,
NULL 허용 여부 등에 대한 정보를 얻을 수 있음

■ 뷰 정보 확인하기

2. 뷰 정보를 SQL 문으로도 확인할 수 있음.

Do it! SQL 문으로 뷰 정보 확인

```
SHOW CREATE VIEW v_payuser;
```

DESCRIBE와 차이점은 DESCRIBE는 열 이름 및 데이터 유형에 대해서만 보여주는 반면, SHOW CREATE VIEW 명령은 뷰를 생성할 때 사용되었던 쿼리문을 보여줌.

실행 결과

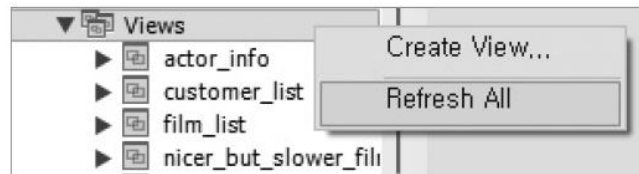
	View	Create View	character_set_client	collation_connection
▶	v_payuser	CREATE ALGORITHM=UNDEFINED DEFINER=`r...	utf8mb4	utf8mb4_0900_ai_ci

```
CREATE           ALGORITHM=UNDEFINED
DEFINER=`root`@`localhost` SQL SECURITY
DEFINER
VIEW `v_payuser` AS select `a`.`first_name` AS
`first_name`,`a`.`last_name` AS
`last_name`,`a`.`email` AS `email`,`b`.`amount`
AS `amount`,`a`.`address_id`
AS `address_id` from (`customer` `a` join (select
`payment`.`customer_id` AS
`customer_id`,sum(`payment`.`amount`)      AS
`amount` from `payment` group by `pay
ment`.`customer_id`) `b` on((`a`.`customer_id` =
`b`.`customer_id`)))
```

■ 뷰 삭제하기

Do it! 뷰 삭제

```
DROP VIEW v_customer;  
DROP VIEW v_payuser;
```



■ 뷰를 사용해 데이터 조작하기

- INSERT, UPDATE, DELETE 등을 사용해
뷰 테이블에 데이터를 입력하고 수정, 삭제를 진행해 보자

1. 실습을 위해 테이블과 뷰를 생성

Do it! 테이블과 뷰 생성

```
CREATE TABLE tbl_a (  
col_1 INT NOT NULL,  
col_2 VARCHAR(50) NOT NULL  
);  
CREATE TABLE tbl_b (  
col_1 INT NOT NULL,  
col_2 VARCHAR(50) NOT NULL  
);  
INSERT INTO tbl_a VALUES(1, 'tbl_a_1');  
INSERT INTO tbl_a VALUES(2, 'tbl_a_2');  
INSERT INTO tbl_b VALUES(1, 'tbl_b_1');  
INSERT INTO tbl_b VALUES(2, 'tbl_b_2');  
CREATE VIEW v_tbl_a  
AS  
SELECT col_1, col_2 FROM tbl_a;  
SELECT * FROM v_tbl_a;
```

실행 결과

	col_1	col_2
▶	1	tbl_a_1
	2	tbl_a_2

■ 뷰를 사용해 데이터 조작하기

2. 단일 뷰의 데이터를 수정

Do it! 단일 뷰 데이터 수정

```
SET SQL_SAFE_UPDATES = 0;  
UPDATE v_tbl_a SET col_2 = 'tbl_a 열 수정'  
WHERE col_1 = 1;  
SELECT * FROM v_tbl_a;
```

실행 결과

	col_1	col_2
▶	1	tbl_a 열 수정
	2	tbl_a_2

쿼리를 실행한 결과,
뷰 테이블에서 데이터가 수정된 것을 확인할 수 있음

■ 뷰를 사용해 데이터 조작하기

3. 뷰 테이블에 데이터를 추가

Do it! 단일 뷰 데이터 추가

```
INSERT v_tbl_a VALUES (3, 'tbl_a_3');  
SELECT * FROM v_tbl_a;
```

실행 결과

	col_1	col_2
▶	1	tbl_a 열 수정
	2	tbl_a_2
	3	tbl_a_3

4. 뷰를 사용하여 데이터를 삭제

Do it! 단일 뷰 데이터 삭제

```
DELETE FROM v_tbl_a WHERE col_1 = 3;  
SELECT * FROM v_tbl_a;
```

실행 결과

	col_1	col_2
▶	1	tbl_a 컬럼 수정
	2	tbl_a_2

■ 뷰를 사용해 데이터 조작하기

5. v_tbl_a와 별개로 tbl_a 테이블을 참조하는 열이 하나만 있는 뷰를 하나 더 생성. 그리고 나서 그 뷰에 데이터를 추가해 보자. 오류가 발생한다

Do it! 새로운 뷰 생성 후 데이터 추가

```
CREATE VIEW v_tbl_a2  
AS  
    SELECT col_1 FROM tbl_a;  
INSERT v_tbl_a2 VALUES (5);
```

Error Code: 1423. Field of view 'doitsql.v_tbl_a2' underlying table doesn't have a default value

실제 원본 테이블은 두 개의 열이 있고 NULL을 허용하지 않기 때문에, 뷰와 원본 테이블이 서로 열이 일치하지 않아 오류가 발생. 이러한 이유 때문에 데이터를 입력, 삭제, 수정하는 작업은 가급적 뷰보다는 원본 테이블에서 직접 수행하는 것을 권장함

■ 뷰를 사용해 데이터 조작하기

6. 이번에는 복합 뷰를 생성해 보자.

Do it! 복합 뷰 생성

```
CREATE VIEW v_tbl_a_b
AS
    SELECT
        a.col_1 as a_col_1,
        a.col_2 as a_col_2,
        b.col_2 as b_col_2
    FROM tbl_a AS a
        INNER JOIN tbl_b AS b ON a.col_1 = b.col_1;
SELECT * FROM v_tbl_a_b;
```

실행 결과

	a_col_1	a_col_2	b_col_2
▶	1	tbl_a 컬럼 수정	tbl_b_1
	2	tbl_a_2	tbl_b_2

v_tbl_a_b에 정의된 대로
tbl_a의 데이터와 tbl_b의 데이터가 조인되어 하나의 뷰 테이블로 묶인 것

■ 뷰를 사용해 데이터 조작하기

7. 복합 뷰 데이터를 수정.

Do it! 복합 뷰 데이터 수정

```
UPDATE v_tbl_a_b SET a_col_2 = 'tbl_a 컬럼 수정', b_col_2 = 'tbl_b 컬럼 수정'  
WHERE a_col_1 = 1;
```

오류 발생

```
Error Code: 1393. Can not modify more than one base table through a join view 'doitsql.v_tbl_a_b'
```

복합 뷰의 경우 데이터를 수정할 수 없다

■ 뷰를 사용해 데이터 조작하기

8. 복합 뷰 테이블의 데이터를 입력.

Do it! 복합 뷰 데이터 입력

```
INSERT v_tbl_a_b VALUES (3, 'tbl_a_3', 'tbl_b_3');
```

오류 발생.

```
Error Code: 1394. Can not insert into join view 'doitsql.v_tbl_a_b' without fields list
```

입력 또한 수정과 마찬가지로 복합 뷰의 경우 데이터를 입력할 수 없다

■ 뷰를 사용해 데이터 조작하기

9. 뷰가 생성되어 있는 상태에서 뷰에서 참조하고 있는 테이블을 삭제해 보자.
문제없이 잘 삭제됨.

Do it! 참조 테이블 삭제

```
DROP TABLE tbl_a;
```

그리고 뷰를 조회하면 오류가 발생함.

Do it! 참조 테이블 삭제된 뷰 조회

```
SELECT * FROM v_tbl_a_b;
```

```
Error Code: 1356. View 'doitsql.v_tbl_a_b' references invalid table(s) or column(s)
or function(s) or definer/invoker of view lack rights to use them
```

v_tbl_a_b 뷰에서 참조하는 테이블이 유효하지 않아서 발생하는 오류임

■ 뷰를 사용해 데이터 조작하기

10. CHECK TABLE로 뷰의 상태를 확인해 보자.

앞서 tbl_a 테이블을 삭제했기 때문에
뷰가 참조하는 테이블이 존재하지 않아 오류가 발생.

Do it! 뷰 정보 확인

```
CHECK TABLE tbl_a_b;
```

실행 결과

	Table	Op	Msg_type	Msg_text
▶	doitsql.v_tbl_a_b	check	Error	View 'doitsql.v_tbl_a_b' references invalid table(s) or column(s) or function(s) or definer/invoker of view lack rights to use them
	doitsql.v_tbl_a_b	check	error	Corrupt

- 스토어드 함수(stored function)는 사용자 함수
- 스토어드 프로시저와 형태와 작성 방법이 비슷하지만 차이점이 있음
- 가장 크게 다른 점으로
스토어드 프로시저는 SELECT 문으로 데이터를 반환할 수 있는 반면,
스토어드 함수는 반드시 RETURN으로 하나의 값만 반환하는 특징이 있음
- 커서(cursor)는 데이터를 1행씩 처리해야 할 때
스토어드 프로시저 안에서 사용할 수 있는 프로그래밍 방식으로,
성능 문제를 야기할 수 있어 자주 사용하지 않음

- 스토어드 함수 이해하기

- 스토어드 함수 만들기

스토어드 함수 생성 기본 형식

```
DELIMITER $$  
CREATE FUNCTION 함수 이름(인수)  
           RETURNS 반환 데이터 유형  
BEGIN  
           로직 작성  
           RETURN 반환값;  
END $$  
DELIMITER ;
```

- 스토어드 함수 이해하기

- 스토어드 함수 만들기

1. 스토어드 함수를 생성하려면 먼저 생성 권한을 부여해야 함.

Do it! 함수 생성 권한 부여

```
SET GLOBAL log_bin_trust_function_creators = 1;
```

■ 스토어드 함수 이해하기

■ 스토어드 함수 만들기

2.

Do it! 스토어드 함수 생성

```
USE doitsql;  
DROP FUNCTION IF EXISTS user_sum;  
DELIMITER $$  
CREATE FUNCTION user_sum(num_1 INT, num_2  
INT)  
        RETURNS INT  
BEGIN  
        RETURN num_1 + num_2;  
END $$  
DELIMITER ;
```

Do it! 스토어드 함수 호출

```
SELECT user_sum (1, 5);
```

실행 결과

user_sum (1, 5)	
▶	6

이 함수는 숫자 2개를 입력받아서 더한 값을 반환함

■ 스토어드 함수 이해하기

■ 스토어드 함수 내용 확인 및 삭제하기

1. 스토어드 함수 내용을 확인하려면 SHOW 문을 사용

Do it! 스토어드 함수 내용 확인

```
SHOW CREATE FUNCTION user_sum;
```

실행 결과

	Function	sql_mode	Create Function	character_set_client	collation_connection	Database Collation
▶	user_sum	ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLE...	CREATE DEFINER=`root`@`localhost` FUNCTI...	utf8mb4	utf8mb4_0900_ai_ci	utf8mb4_0900_ai_ci

■ 스토어드 함수 이해하기

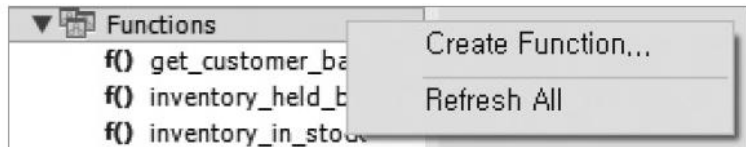
■ 스토어드 함수 내용 확인 및 삭제하기

2. 생성한 스토어드 함수를 삭제할 때는 DROP 문을 사용

Do it! 스토어드 함수 삭제

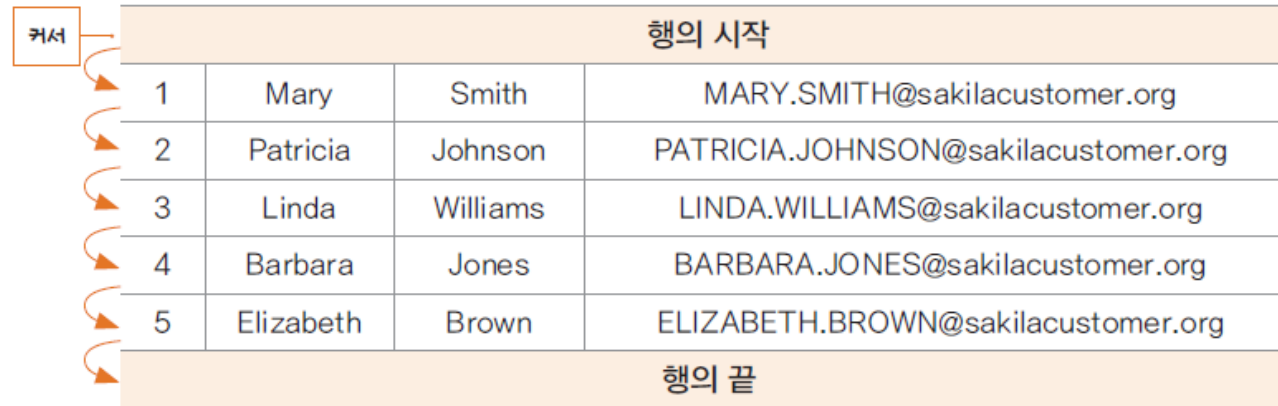
```
DROP FUNCTION user_sum;
```

앞서 스토어드 함수를 삭제하고 나면 스토어드프로시저를 삭제했을 때와 마찬가지로 워크벤치의 내비게이터에서 [Functions]를 확장하여 삭제된 것을 확인할 수 있음



■ 커서 알아 두기

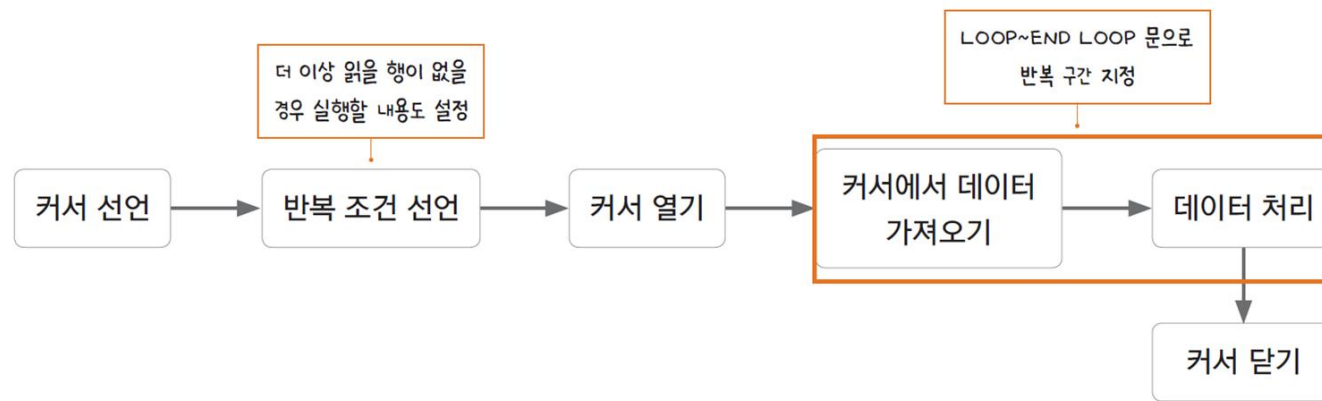
- 테이블의 데이터를 1행씩 처리하는 방법으로 스토어드 프로시저 내부에서 사용할 수 있음
- 기본 동작 원리는 첫 번째 행을 시작으로 각 행을 처리한 후에 마지막 행까지 데이터를 모두 처리하면 동작이 완료됨



행의 시작			
1	Mary	Smith	MARY.SMITH@sakilacustomer.org
2	Patricia	Johnson	PATRICIA.JOHNSON@sakilacustomer.org
3	Linda	Williams	LINDA.WILLIAMS@sakilacustomer.org
4	Barbara	Jones	BARBARA.JONES@sakilacustomer.org
5	Elizabeth	Brown	ELIZABETH.BROWN@sakilacustomer.org
행의 끝			

커서 동작 원리

■ 커서 알아 두기



커서 작동 순서

■ 커서 알아 두기

1. 실습하면서 커서의 동작 원리를 살펴보자.

지금 생성할 커서는 payment 테이블에서 staff_id가 1인 경우에 해당하는 결제 금액(amount)을 한 행씩 읽으면서 값을 더함.

사실 SUM 함수로 한번에 집계할 수도 있지만,
커서를 사용하여 한 행씩 더하는 방법으로 작성해보자

■ 커서 알아 두기

1.

Do it! 커서 생성

```
DROP PROCEDURE IF EXISTS doit_cursor;
DELIMITER $$
CREATE PROCEDURE doit_cursor()
BEGIN
    DECLARE endOfRow BOOLEAN DEFAULT FALSE;

    -- 커서에 사용할 변수
    DECLARE user_payment_id INT; -- payment_id를          저장할 변수
    DECLARE user_amount DECIMAL(10,2) DEFAULT 0; -- amount를 저장할 변수
    DECLARE idCursor CURSOR FOR -- 커서 선언
    SELECT payment_id FROM payment WHERE staff_id = 1;

    -- 반복 조건 선언
    DECLARE CONTINUE HANDLER -- 행의 끝이면 endOfRow 변수에 TRUE 대입
    FOR NOT FOUND SET endOfRow = TRUE;
    -- 커서 열기
    OPEN idCursor;
```

■ 커서 알아 두기

1.

Do it! 커서 생성

```
-- 반복 구문
sum_loop : LOOP
    FETCH idCursor INTO user_payment_id; -- 첫 번째 데이터 가져오기
    IF endOfRow THEN
        LEAVE sum_loop; -- 마지막 행이면 종료
    END IF;

-- 데이터 처리
    SET user_amount = user_amount + (SELECT amount FROM payment WHERE payment_id
= user_payment_id);
    END LOOP sum_loop;
-- 데이터 결과 반환
    SELECT user_amount;

-- 커서 닫기
    CLOSE idCursor;

END$$
DELIMITER ;
```

■ 커서 알아 두기

1. 커서를 사용하면 각 행을 더하면서 실행 중간에 지금까지 연산된 중간 값을 활용할 수 있다는 장점이 있음.

여기에서는 단순히 값을 더하는 것만 실행해 보았지만,
각 행의 값을 읽으면서 다양한 연산을 할 수도 있음을 알아 두자

■ 커서 알아 두기

2. 커서를 생성한 스토어드 프로시저를 실행하여 커서의 결과를 확인해 보자

Do it! 스토어드 프로시저 실행

```
CALL doit_cursor();
```

실행 결과

	user_amount
▶	33482.50

2. 실제 SUM 함수를 사용했을 때와 합계가 동일한지 비교하여 커서가 제대로 실행되었는지를 최종 확인해 보자

Do it! 결과 비교를 위한 쿼리 입력

```
SELECT sum(amount) FROM payment WHERE staff_id = 1;
```

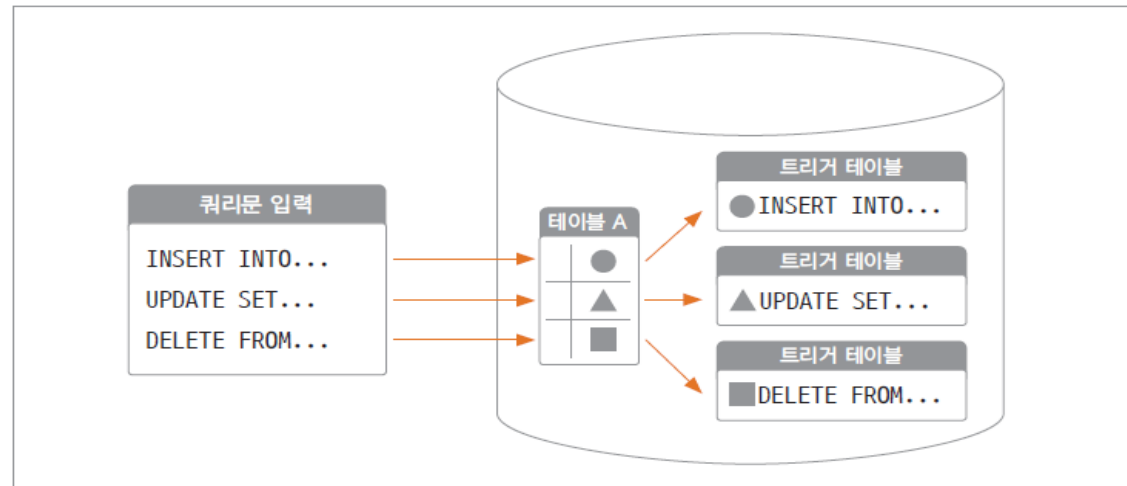
실행 결과

	user_amount
▶	33482.50

- 트리거(trigger)는 방아쇠가 당겨졌을 때 총알이 발사되는 것처럼

데이터베이스에서 테이블에 입력, 수정, 삭제 등 이벤트(방아쇠)가 발생했을 때 미리 정해진 규칙에 따라 자동으로 실행됨

- MySQL에는 트리거 기능이 제공되고 있지만 다른 DBMS와 다르게 DDL 문을 수행될 때 동작하는 트리거는 제공하지 않음
- 트리거를 설정해 놓으면 INSERT, UPDATE, DELETE와 같은 DML 문이 수행될 때 데이터베이스에서 자동으로 동작함



트리거 동작 과정

- 트리거는 어떠한 테이블에 데이터 수정과 같은 작업이 발생하였을 때, 이전 값의 상태를 별도의 테이블에 보관하고 싶을 때 많이 사용함
- 예를 들어 상품 입출고에 따른 재고 변경이나 판매 금액의 변동이 발생할 때, 이전의 값과 새로운 값을 각각 다른 테이블에 기록함으로써 정상적으로 데이터가 수정되었는지 등을 검증할 수 있음

MySQL 트리거의 특징

- 테이블에 DML 문(INSERT, UPDATE, DELETE 등) 이벤트가 발생해야만 자동으로 작동한다.
- 변경 전을 기록하는 BEFORE, 변경 후를 기록하는 AFTER 트리거가 있다.
- IN, OUT 매개변수, 즉 입력과 출력 매개변수를 사용할 수 없다.
- MySQL에서는 뷰에 트리거를 부착할 수 없다.

■ 트리거의 종류

- 행마다 실행되는 행 트리거와
명령문 단위로 실행되는 문장 트리거로 구분할 수 있음

■ 트리거의 종류

■ 행 트리거

- 테이블에서 INSERT, UPDATE, DELETE 등의 명령문으로 인해 영향을 받는 행 각각에 이벤트가 실행됨
- 아래 표는 각 명령문별로 행의 변경 전후를 기록하는지 여부를 정리한 것임
- INSERT의 경우에는 이전 행이 존재하지 않으므로 OLD가 존재하지 않고, DELETE의 경우 이후 상태가 존재하지 않으므로 NEW가 존재하지 않음

이벤트	OLD	NEW
INSERT	X	O
UPDATE	O	O
DELETE	O	X

- 트리거의 종류

- 문장 트리거

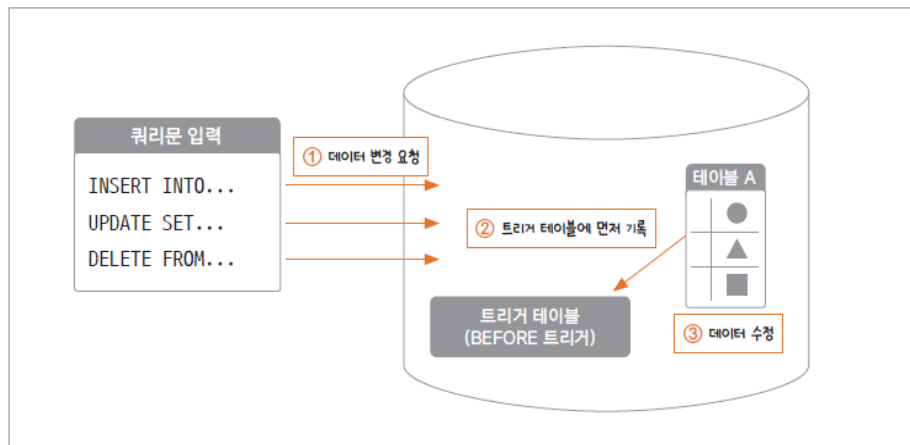
- 문장 트리거는 영향을 받는 행의 개수와 관계없이 INSERT, UPDATE, DELETE 문에 대해 한 번만 실행됨
 - 즉, 행수에 상관없이 트랜잭션에 대해 명령문 트리거가 한 번 실행됨

■ 트리거 실행 시기

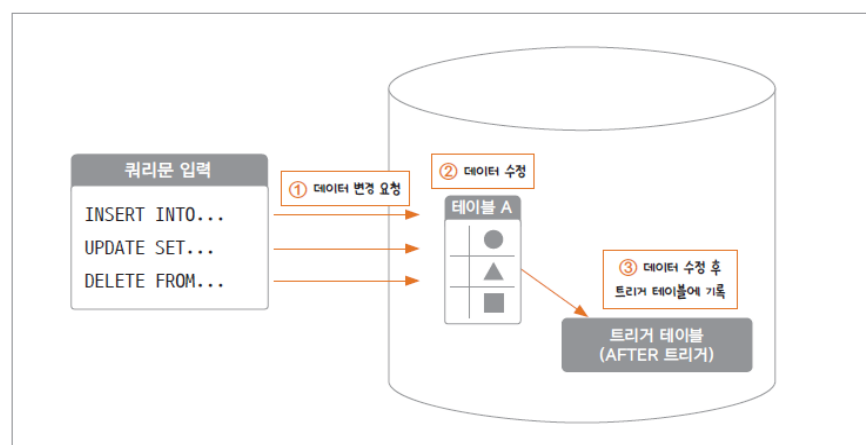
- 변경 전후 트리거가 각각 존재하는 이유는
현재 상태값과 변경된 상태값을 비교할 때, 변경 이전에 기록을 확인할 것인지,
변경 이후 기록을 확인할 것인지를 선택할 수 있도록 하기 위함

트리거 실행 시기를 정하는 명령문

- AFTER: 쿼리 이벤트 작동 후 실행
- BEFORE: 쿼리 이벤트 작동 전 실행(미리 데이터 확인 가능)



BEFORE 트리거 동작 과정



AFTER 트리거 동작 과정

■ 트리거 생성하기

- 트리거는 CREATE TRIGGER 문을 사용하여 생성할 수 있음

트리거 생성 기본 형식

```
DELIMITER $$  
CREATE TRIGGER 트리거명  
[BEFORE | AFTER] [INSERT | UPDATE | DELETE]  
ON 원본 테이블  
FOR EACH ROW  
  
BEGIN  
    <트리거 작동 시 실행할 코드>  
  
END $$  
DELIMITER ;
```


■ 트리거 생성하기

1. 트리거 생성 실습을 위해 새로운 테이블 tbl_trigger_1, tbl_trigger_2 를 생성하고

tbl_trigger_1 테이블에만 데이터를 입력해 보자.

실행 결과

	col_1	col_2
▶	1	데이터 1 입력

Do it! 테이블 생성

```
USE doitsql;
```

```
CREATE TABLE tbl_trigger_1 (  
  col_1 INT,  
  col_2 VARCHAR(50)  
);
```

```
CREATE TABLE tbl_trigger_2 (  
  col_1 INT,  
  col_2 VARCHAR(50)  
);
```

```
INSERT INTO tbl_trigger_1 VALUES (1, '데이터 1 입력');
```

```
SELECT * FROM tbl_trigger_1;
```

■ 트리거 생성하기

2. 트리거를 생성. UPDATE가 발생하였을 때 트리거가 동작되도록 설정하였고,
트리거 발생 시기는 AFTER로 지정함
tbl_trigger_1 테이블에 변경이 발생하면 tbl_trigger_2 테이블에 변경 내역을 기록

Do it! UPDATE 발생 시 동작하는 트리거 생성

```
DELIMITER $$

CREATE TRIGGER dot_update_trigger
AFTER UPDATE
ON tbl_trigger_1
FOR EACH ROW

BEGIN
    INSERT INTO tbl_trigger_2 VALUES (OLD.col_1, OLD.col_2);

END $$
DELIMITER ;
```

■ 트리거 생성하기

3. tbl_trigger_1 테이블의 데이터를 변경.
그리고 기존 tbl_trigger_1 테이블과 이벤트가 발생한 뒤
트리거가 발생되어 변경 내용을 저장한 tbl_trigger_2 테이블을 모두 조회.

Do it! 트리거 실행

```
SET SQL_SAFE_UPDATES = 0;  
UPDATE tbl_trigger_1 SET col_1 = 2, col_2 = '1을 2로 수정';  
  
SELECT * FROM tbl_trigger_2;  
  
SELECT * FROM tbl_trigger_1;
```

tbl_trigger_1 테이블 조회 결과

	col_1	col_2
▶	2	1을 2로 수정

tbl_trigger_2 테이블 조회 결과

	col_1	col_2
▶	1	데이터 1 입력

tbl_trigger_2 테이블에서는
tbl_trigger_1 테이블의 데이터가
수정되기 이전의 상태가 저장된 것을 확인할
수 있음

■ 트리거 생성하기

3. 이런 식으로 트리거를 잘 활용하면 테이블을 감시하는 역할도 수행할 수 있음

예를 들어 어떤 사용자가 데이터를 조작하려고 할 때,
이전의 값과 변경된 값을 트리거 테이블에 기록하여
만약의 사태가 일어났을 때 전후를 비교해 데이터를 복구할 수도 있음.

하지만 트리거는 데이터베이스에 부하를 유발하므로
트래픽이 많은 데이터베이스에서는 사용을 자제하는 것이 좋음