3. 계층 구조 설계

#0.강의/2.데이터베이스로드맵/4.설계2

계층 구조 설계가 필요한 이유

데이터베이스를 설계하다 보면 **계층 구조**를 저장해야 하는 상황을 자주 만나게 된다. 쇼핑몰을 예로 들면, 상품 카테고리가 대표적인 계층 구조이다.

```
전자제품
├── 컴퓨터
│   ├── 노트북
│   ├── 데스크탑
│   └── 태블릿
├── 스마트폰
│   ├── 애플
│   └── 삼성
└── 가전제품
    ├── TV
    └── 냉장고
```

이런 계층 구조는 어디에서나 볼 수 있다.

- 쇼핑몰의 상품 카테고리
- 회사의 조직도
- 게시판의 댓글과 대댓글
- 파일 시스템의 폴더 구조
- 메뉴 구조

그런데 관계형 데이터베이스는 기본적으로 **평면적인 테이블 구조**를 가진다. 행과 열로 이루어진 2차원 테이블에 트리 형태의 계층 구조를 어떻게 저장할 수 있을까? 이번 섹션에서는 계층 구조를 데이터베이스에 저장하고 조회하는 다양한 방법을 학습한다.

학습 목표

이번 섹션을 통해 다음 내용을 학습한다.

- 인접 리스트 모델로 계층 구조를 설계하는 방법
- 계층 구조 데이터를 조회할 때 발생하는 문제점
- CTE(Common Table Expression)를 활용한 재귀 쿼리
- 폐쇄 테이블 모델을 활용한 성능 최적화

인접 리스트 모델

계층 구조를 데이터베이스에 저장하는 가장 직관적이고 널리 사용되는 방법이 바로 **인접 리스트 모델(Adjacency List Model)**이다.

인접 리스트 모델이란?

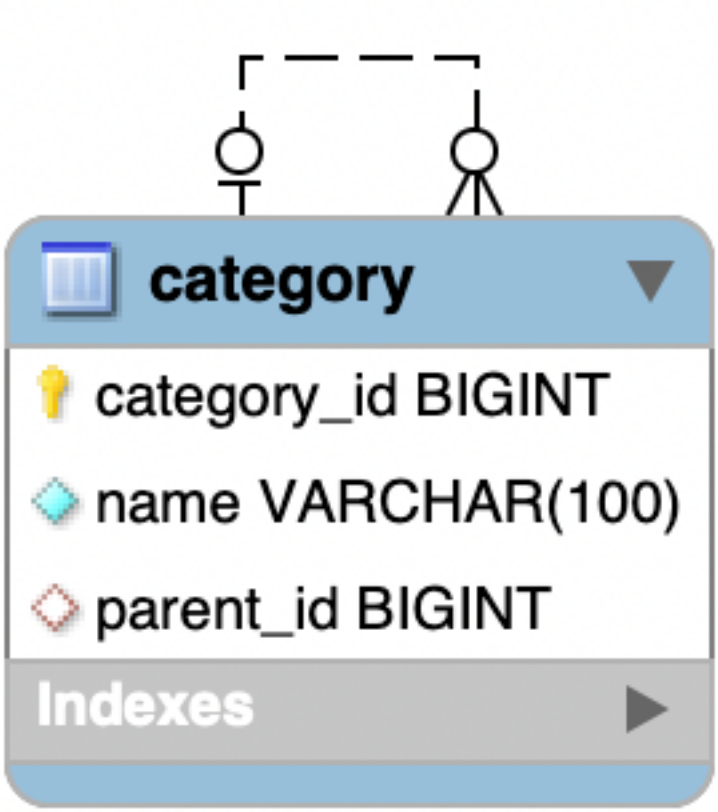
인접 리스트 모델은 각 행이 자신의 **부모를 참조**하는 방식이다. 마치 "나의 부모는 누구인가?"라는 질문에 답하는 구조이다.

핵심 아이디어는 간단하다.

- 각 노드는 자신의 부모 노드를 가리키는 외래 키를 가진다.
- 최상위 노드(루트)는 부모가 없으므로 `NULL`을 가진다.

테이블 설계

쇼핑몰의 상품 카테고리를 인접 리스트 모델로 설계해보자.




```
DROP TABLE IF EXISTS attribute_definition; -- 복습시 FK 제약조건으로 제거 필요(이후에 사용 테이블)
DROP TABLE IF EXISTS category;

CREATE TABLE category (
    category_id   BIGINT       NOT NULL AUTO_INCREMENT,
    name          VARCHAR(100) NOT NULL,
    parent_id     BIGINT       NULL,
    PRIMARY KEY (category_id),
    FOREIGN KEY (parent_id) REFERENCES category(category_id)
);
```

테이블 구조를 살펴보자.

- `category_id`: 카테고리의 고유 식별자
- `name`: 카테고리 이름
- `parent_id`: 부모 카테고리를 참조하는 외래 키. 자기 자신의 테이블을 참조하는 **자기 참조(Self-Referencing)** 관계이다.

`parent_id`는 같은 테이블의 `category_id`를 참조한다. 이렇게 자기 자신을 참조하는 외래 키를 **자기 참조 외래**

키라고 한다.

데이터 입력

이제 카테고리 데이터를 입력해보자.

```
-- 1단계: 최상위 카테고리 (루트)
INSERT INTO category (name, parent_id) VALUES ('전자제품', NULL); -- id:1
INSERT INTO category (name, parent_id) VALUES ('의류', NULL);    -- id:2
INSERT INTO category (name, parent_id) VALUES ('식품', NULL);    -- id:3

-- 2단계: 전자제품의 하위 카테고리
INSERT INTO category (name, parent_id) VALUES ('컴퓨터', 1);     -- id:4
INSERT INTO category (name, parent_id) VALUES ('스마트폰', 1);   -- id:5
INSERT INTO category (name, parent_id) VALUES ('가전제품', 1);   -- id:6

-- 3단계: 컴퓨터의 하위 카테고리
INSERT INTO category (name, parent_id) VALUES ('노트북', 4);
INSERT INTO category (name, parent_id) VALUES ('데스크탑', 4);
INSERT INTO category (name, parent_id) VALUES ('태블릿', 4);

-- 3단계: 스마트폰의 하위 카테고리
INSERT INTO category (name, parent_id) VALUES ('애플', 5);
INSERT INTO category (name, parent_id) VALUES ('삼성', 5);

-- 3단계: 가전제품의 하위 카테고리
INSERT INTO category (name, parent_id) VALUES ('TV', 6);
INSERT INTO category (name, parent_id) VALUES ('냉장고', 6);

-- 2단계: 의류의 하위 카테고리
INSERT INTO category (name, parent_id) VALUES ('남성의류', 2);
INSERT INTO category (name, parent_id) VALUES ('여성의류', 2);

-- 3단계: 남성의류의 하위 카테고리
INSERT INTO category (name, parent_id) VALUES ('셔츠', 14);
INSERT INTO category (name, parent_id) VALUES ('바지', 14);
```

전체 데이터를 조회해보자.

```sql
SELECT * FROM category;
```

[실행 결과]

category_id	name	parent_id
1	전자제품	NULL
2	의류	NULL
3	식품	NULL
4	컴퓨터	1
5	스마트폰	1
6	가전제품	1
7	노트북	4
8	데스크탑	4
9	태블릿	4
10	애플	5
11	삼성	5
12	TV	6
13	냉장고	6
14	남성의류	2
15	여성의류	2
16	셔츠	14
17	바지	14

이 데이터를 트리 구조로 표현하면 다음과 같다.

```
```

```
전자제품 (id=1, parent=NULL)
├── 컴퓨터 (id=4, parent=1)
│   ├── 노트북 (id=7, parent=4)
│   ├── 데스크탑 (id=8, parent=4)
│   └── 태블릿 (id=9, parent=4)
├── 스마트폰 (id=5, parent=1)
│   ├── 애플 (id=10, parent=5)
│   └── 삼성 (id=11, parent=5)
└── 가전제품 (id=6, parent=1)
    ├── TV (id=12, parent=6)
    └── 냉장고 (id=13, parent=6)

의류 (id=2, parent=NULL)
├── 남성의류 (id=14, parent=2)
│   ├── 셔츠 (id=16, parent=14)
│   └── 바지 (id=17, parent=14)
└── 여성의류 (id=15, parent=2)

식품 (id=3, parent=NULL)
```

기본 조회 쿼리

루트 카테고리 조회

최상위 카테고리(루트)는 `parent_id`가 `NULL`인 카테고리이다.

```sql
SELECT * FROM category WHERE parent_id IS NULL;
```

[실행 결과]

category_id	name	parent_id
1	전자제품	NULL
2	의류	NULL
3	식품	NULL

특정 카테고리의 직속 자식 조회

특정 카테고리의 바로 아래 자식들을 조회하려면 `parent_id`로 필터링하면 된다.

```sql
-- 전자제품(id=1)의 직속 자식 조회
SELECT * FROM category WHERE parent_id = 1;
```

[실행 결과]

category_id	name	parent_id
4	컴퓨터	1
5	스마트폰	1
6	가전제품	1

```sql
-- 컴퓨터(id=4)의 직속 자식 조회
SELECT * FROM category WHERE parent_id = 4;
```

[실행 결과]

category_id	name	parent_id
7	노트북	4
8	데스크탑	4
9	태블릿	4

특정 카테고리의 부모 조회

특정 카테고리의 부모를 조회하려면 셀프 조인을 사용한다.

> **셀프 조인은 자기 자신의 테이블과 조인한다.**
> 셀프 조인에 대한 자세한 내용은 "기본편 ➜ 3. 조인2 - 외부 조인과 기타 조인 ➜ 셀프 조인"을 참고하자.

```sql
-- 노트북(id=7)의 부모 조회
SELECT p.*
FROM category c
JOIN category p ON c.parent_id = p.category_id
WHERE c.category_id = 7;
```

메인 테이블이 참조하는 부모 ID와 대상 테이블의 카테고리 ID를 조인하면 부모 테이블을 찾을 수 있다.

[실행 결과]

category_id	name	parent_id
4	컴퓨터	1

노트북(`category_id = 7`)의 부모인 '컴퓨터'가 조회되었다.

인접 리스트 모델의 장점

인접 리스트 모델은 다음과 같은 장점이 있다.

- **직관적인 구조**: 부모-자식 관계를 외래 키 하나로 표현하므로 이해하기 쉽다.
- **간단한 데이터 추가/수정/삭제**: 노드를 추가하거나 이동하는 작업이 단순하다.
- **저장 공간 효율성**: 각 노드당 하나의 행만 저장하므로 저장 공간이 효율적이다.
- **외래 키 제약조건 활용**: 데이터베이스의 외래 키 제약조건을 활용하여 참조 무결성을 보장할 수 있다.

인접 리스트 모델의 추가/수정/삭제

```sql
-- 새로운 카테고리 추가
INSERT INTO category (name, parent_id) VALUES ('게이밍노트북', 7);

-- 카테고리 이동 (노트북을 가전제품 아래로 이동)
UPDATE category SET parent_id = 6 WHERE category_id = 7;

-- 카테고리 삭제 (자식이 없는 경우)
DELETE FROM category WHERE category_id = 17;
```

인접 리스트 모델의 한계

그런데 인접 리스트 모델에는 한계가 있다. 바로 **깊은 계층의 데이터를 한 번에 조회하기 어렵다**는 점이다.

예를 들어, "전자제품 카테고리 아래의 **모든** 하위 카테고리를 조회해줘"라는 요청이 들어왔다고 가정하자.

전자제품 아래에는 컴퓨터, 스마트폰, 가전제품이 있고, 컴퓨터 아래에는 노트북, 데스크탑, 태블릿이 있다. 이 모든 것을 한 번의 쿼리로 가져올 수 있을까?

직속 자식만 조회하는 것은 쉽다.

전자제품(`category_id=1`)의 직속 자식을 조회하려면 `parent_id`가 1인 것을 찾으면 된다.

```
SELECT * FROM category WHERE parent_id = 1;
```

[실행 결과]

category_id	name	parent_id
4	컴퓨터	1
5	스마트폰	1
6	가전제품	1

하지만 손자, 증손자까지 포함한 **모든 자손**을 조회하려면 어떻게 해야 할까? 계층의 깊이가 얼마나 될지 모르는 상황에서 말이다.

다음 시간에는 이 문제를 해결하는 방법들을 하나씩 살펴보자.

계층 구조 조회의 어려움

인접 리스트 모델에서 직속 자식이나 부모를 조회하는 것은 쉽다. 하지만 **모든 자손**이나 **모든 조상**을 조회하는 것은 상당히 까다롭다. 이번 수업에서는 이 문제를 다양한 방법으로 해결해보자.

테이블 초기화

시작하기 전에 데이터를 처음 상태로 초기화하자.

```
DROP TABLE IF EXISTS category;

CREATE TABLE category (
    category_id   BIGINT       NOT NULL AUTO_INCREMENT,
    name          VARCHAR(100) NOT NULL,
    parent_id     BIGINT       NULL,
    PRIMARY KEY (category_id),
    FOREIGN KEY (parent_id) REFERENCES category(category_id)
);

-- 1단계: 최상위 카테고리 (루트)
INSERT INTO category (name, parent_id) VALUES ('전자제품', NULL);
INSERT INTO category (name, parent_id) VALUES ('의류', NULL);
INSERT INTO category (name, parent_id) VALUES ('식품', NULL);

-- 2단계: 전자제품의 하위 카테고리
INSERT INTO category (name, parent_id) VALUES ('컴퓨터', 1);
INSERT INTO category (name, parent_id) VALUES ('스마트폰', 1);
INSERT INTO category (name, parent_id) VALUES ('가전제품', 1);

-- 3단계: 컴퓨터의 하위 카테고리
INSERT INTO category (name, parent_id) VALUES ('노트북', 4);
INSERT INTO category (name, parent_id) VALUES ('데스크탑', 4);
INSERT INTO category (name, parent_id) VALUES ('태블릿', 4);

-- 3단계: 스마트폰의 하위 카테고리
INSERT INTO category (name, parent_id) VALUES ('애플', 5);
INSERT INTO category (name, parent_id) VALUES ('삼성', 5);

-- 3단계: 가전제품의 하위 카테고리
INSERT INTO category (name, parent_id) VALUES ('TV', 6);
INSERT INTO category (name, parent_id) VALUES ('냉장고', 6);

-- 2단계: 의류의 하위 카테고리
INSERT INTO category (name, parent_id) VALUES ('남성의류', 2);
INSERT INTO category (name, parent_id) VALUES ('여성의류', 2);
```

```sql
-- 3단계: 남성의류의 하위 카테고리
INSERT INTO category (name, parent_id) VALUES ('셔츠', 14);
INSERT INTO category (name, parent_id) VALUES ('바지', 14);
```

문제 상황

쇼핑몰에서 "전자제품" 카테고리 페이지에 접속하면, 전자제품과 그 하위의 모든 카테고리에 속한 상품들을 보여줘야 한다.

```
전자제품 (id=1)
├── 컴퓨터 (id=4)
│   ├── 노트북 (id=7)
│   ├── 데스크탑 (id=8)
│   └── 태블릿 (id=9)
├── 스마트폰 (id=5)
│   ├── 애플 (id=10)
│   └── 삼성 (id=11)
└── 가전제품 (id=6)
    ├── TV (id=12)
    └── 냉장고 (id=13)
```

전자제품(id=1)의 모든 자손인 `1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13`을 조회해야 한다.

방법1: 애플리케이션에서 반복 호출

가장 단순한 방법은 애플리케이션 코드에서 데이터베이스를 여러 번 호출하는 것이다.

```
1단계: 전자제품(id=1) 조회
2단계: 전자제품의 자식들(id=4,5,6) 조회
3단계: 각 자식의 자식들(id=7,8,9,10,11,12,13) 조회
4단계: 더 이상 자식이 없으면 종료
```

이 방식을 SQL로 표현하면 다음과 같다.

```sql
-- 1단계: 루트 조회
SELECT * FROM category WHERE category_id = 1;

-- 2단계: 1단계 결과의 자식들 조회
SELECT * FROM category WHERE parent_id = 1;

-- 3단계: 2단계 결과의 자식들 조회
SELECT * FROM category WHERE parent_id IN (4, 5, 6);

-- 4단계: 3단계 결과의 자식들 조회 (결과가 없으면 종료)
SELECT * FROM category WHERE parent_id IN (7, 8, 9, 10, 11, 12, 13);
```

이 방식은 구현이 간단하지만, **계층의 깊이만큼 데이터베이스를 호출**해야 한다. 계층이 10단계라면 10번의 쿼리가 필요하다. 네트워크 비용과 성능 문제가 발생할 수 있다.

방법2: JOIN을 사용한 고정 깊이 조회

계층의 최대 깊이가 정해져 있다면, JOIN을 사용하여 한 번의 쿼리로 조회할 수 있다.

예를 들어, 최대 3단계까지만 조회한다고 가정하자.

```sql
SELECT
    c1.category_id AS level1_id,
    c1.name AS level1_name,
    c2.category_id AS level2_id,
    c2.name AS level2_name,
    c3.category_id AS level3_id,
    c3.name AS level3_name
FROM category c1
LEFT JOIN category c2 ON c2.parent_id = c1.category_id
LEFT JOIN category c3 ON c3.parent_id = c2.category_id
WHERE c1.category_id = 1;
```

[실행 결과]

level1_id	level1_name	level2_id	level2_name	level3_id	level3_name

1	전자제품	4	컴퓨터	7	노트북
1	전자제품	4	컴퓨터	8	데스크탑
1	전자제품	4	컴퓨터	9	태블릿
1	전자제품	5	스마트폰	10	애플
1	전자제품	5	스마트폰	11	삼성
1	전자제품	6	가전제품	12	TV
1	전자제품	6	가전제품	13	냉장고

이 방식은 한 번의 쿼리로 결과를 얻을 수 있지만, 몇 가지 문제가 있다.

- 계층의 깊이가 고정되어 있어야 한다.
- 깊이가 늘어나면 JOIN도 늘어나야 한다.
- 결과가 중복으로 표시되어 처리가 필요하다.

방법3: UNION ALL을 사용한 조회

각 레벨별로 쿼리를 작성하고 `UNION ALL`로 합칠 수 있다.

```
-- 레벨 1: 전자제품 자신
SELECT category_id, name, parent_id, 1 AS level
FROM category
WHERE category_id = 1

UNION ALL

-- 레벨 2: 전자제품의 직속 자식
SELECT category_id, name, parent_id, 2 AS level
FROM category
WHERE parent_id = 1

UNION ALL

-- 레벨 3: 레벨 2의 자식들
SELECT c.category_id, c.name, c.parent_id, 3 AS level
```

```
FROM category c
WHERE c.parent_id IN (
    SELECT category_id FROM category WHERE parent_id = 1
);
```

[실행 결과]

category_id	name	parent_id	level
1	전자제품	NULL	1
4	컴퓨터	1	2
5	스마트폰	1	2
6	가전제품	1	2
7	노트북	4	3
8	데스크탑	4	3
9	태블릿	4	3
10	애플	5	3
11	삼성	5	3
12	TV	6	3
13	냉장고	6	3

이 방식도 계층의 깊이만큼 UNION을 추가해야 하므로 유연하지 않다.

조상 조회의 어려움

자손을 조회하는 것뿐만 아니라, **조상을 조회**하는 것도 마찬가지로 어렵다.

예를 들어, "노트북" 카테고리의 모든 조상을 조회하려면 어떻게 해야 할까?

```
노트북 → 컴퓨터 → 전자제품
```

이것도 JOIN을 여러 번 사용해야 한다.

```sql
-- 노트북(id=7)의 조상들을 조회
SELECT
    c1.category_id AS id1, c1.name AS name1,
    c2.category_id AS id2, c2.name AS name2,
    c3.category_id AS id3, c3.name AS name3
FROM category c1
LEFT JOIN category c2 ON c1.parent_id = c2.category_id
LEFT JOIN category c3 ON c2.parent_id = c3.category_id
WHERE c1.category_id = 7;
```

[실행 결과]

id1	name1	id2	name2	id3	name3
7	노트북	4	컴퓨터	1	전자제품

특정 깊이까지만 조회

때로는 "전자제품 카테고리에서 2단계까지만 보여줘"라는 요구사항이 있을 수 있다.

```sql
-- 전자제품(id=1) 기준 2단계까지 조회
SELECT category_id, name, parent_id, 1 AS depth
FROM category
WHERE category_id = 1

UNION ALL

SELECT category_id, name, parent_id, 2 AS depth
FROM category
WHERE parent_id = 1;
```

[실행 결과]

category_id	name	parent_id	depth
1	전자제품	NULL	1
4	컴퓨터	1	2
5	스마트폰	1	2
6	가전제품	1	2

정리

지금까지 살펴본 방법들은 모두 **계층의 깊이를 미리 알아야 하거나**, **여러 번의 쿼리가 필요**하다는 한계가 있다.

방법	장점	단점
애플리케이션 반복 호출	구현 간단	네트워크 비용, 성능 저하
JOIN 고정 깊이	한 번의 쿼리	깊이 고정, 유연성 부족
UNION ALL	레벨별 분리 가능	깊이 고정, 쿼리 복잡

실무에서 계층 구조의 깊이가 가변적인 경우가 대부분이다. 상품 카테고리가 3단계일 수도, 5단계일 수도, 10단계일 수도 있다. 지금까지 설명한 JOIN, UNION을 사용하는 방식으로는 이런 문제를 해결하기 어렵다. 결과적으로 **계층의 깊이가 가변적일 때 대응하기 어렵다**는 것이다. 그렇다고 애플리케이션에서 반복 호출하게 되면 무수히 많은 네트워크 호출이 발생하고, 결과적으로 성능이 저하된다.

이런 문제를 해결하기 위해 SQL 표준에 **재귀 쿼리(Recursive Query)**가 도입되었고, 이를 **CTE(Common Table Expression)**로 작성할 수 있다. 다음 수업에서 CTE에 대해 자세히 알아보자.

CTE와 재귀 쿼리 1

이전 수업에서 인접 리스트 모델의 한계를 살펴보았다. 계층의 깊이를 모르는 상황에서 모든 자손이나 조상을 조회하기

가 매우 어려웠다. 이 문제를 해결하기 위해 **CTE(Common Table Expression)**와 **재귀 쿼리**를 학습한다.

CTE가 필요한 이유

앞서 살펴본 방법들의 공통적인 문제점은 **계층의 깊이가 가변적일 때 대응하기 어렵다**는 것이다.

실무에서 발생하는 상황을 생각해보자.

- 쇼핑몰 카테고리: 어떤 카테고리는 2단계, 어떤 카테고리는 5단계
- 조직도: 부서에 따라 계층의 깊이가 다름
- 댓글: 대댓글, 대대댓글... 깊이를 예측할 수 없음

계층의 깊이를 모르는 상황에서 "내려갈 수 있는 데까지 내려가서 모든 자손을 찾아와"라는 요구사항을 처리하려면 **재귀(Recursion)**가 필요하다.

MySQL 8.0부터 **재귀 CTE(Recursive CTE)**를 지원하면서, 이런 문제를 SQL 한 번으로 해결할 수 있게 되었다.

> **CTE는 SQL 표준이다**
> 다행히 CTE는 특정 데이터베이스에만 존재하는 독자적인 기능이 아니다. SQL 표준에 등재된 문법이다. 즉, 여러분이 여기서 CTE를 한 번 제대로 배워두면, 나중에 다른 데이터베이스를 다루게 되더라도 똑같이 사용할 수 있다는 뜻이다. 이것이 바로 표준을 공부하는 이유다.

CTE란?

우선 기본부터 배워보자. CTE(Common Table Expression)는 쿼리 내에서 임시로 사용할 수 있는 **이름 있는 결과 집합**이다. `WITH` 절을 사용하여 정의한다. 해당 SQL 내에만 잠시 존재하는 임시 뷰(View)로 볼 수도 있다.

기본 CTE 문법

```
WITH cte_name AS (
    -- CTE를 정의하는 쿼리
    SELECT ...
)
-- CTE를 사용하는 메인 쿼리
SELECT * FROM cte_name;
```

간단한 예제를 살펴보자.

```
WITH top_categories AS (
    SELECT * FROM category WHERE parent_id IS NULL
)
SELECT * FROM top_categories;
```

이 쿼리에서 `top_categories`라는 CTE를 정의하고, 메인 쿼리에서 이를 테이블처럼 사용했다.

[실행 결과]

category_id	name	parent_id
1	전자제품	NULL
2	의류	NULL
3	식품	NULL

CTE 자체는 서브쿼리와 비슷해 보일 수 있지만, 몇 가지 장점이 있다.

- 쿼리의 가독성이 좋아진다.
- 같은 CTE를 여러 번 참조할 수 있다.
- **재귀 쿼리**를 작성할 수 있다.

재귀 CTE 문법

CTE는 스스로 재귀 쿼리를 작성할 수 있는 재귀 문법도 제공한다.
재귀 CTE는 **자기 자신을 참조하는 CTE**이다. `WITH RECURSIVE` 키워드를 사용한다.

```
WITH RECURSIVE cte_name AS (
    -- 기본 케이스 (Anchor Member): 재귀의 시작점
    SELECT ...

    UNION ALL

    -- 재귀 케이스 (Recursive Member): 자기 자신을 참조
    SELECT ... FROM cte_name WHERE ...
```

```
)
SELECT * FROM cte_name;
```

- 기본 케이스와 재귀 케이스를 구분하는 곳에 UNION 또는 UNION ALL을 적어주어야 한다. 이 부분은 문법상 필수이다. 쉽게 이야기해서 기본 케이스와 재귀 케이스의 결과를 합친다고 생각하면 된다.

재귀 CTE는 두 부분으로 구성된다.

1. **기본 케이스(Anchor Member)**: 재귀의 시작점. 처음 실행되는 쿼리이다.
2. **재귀 케이스(Recursive Member)**: CTE 자신을 참조하여 반복 실행되는 쿼리이다.

실행 흐름은 다음과 같다.

1. 기본 케이스를 실행하여 초기 결과를 얻는다.
2. 재귀 케이스를 실행하여 새로운 행을 얻는다.
3. 새로운 행이 없을 때까지 2번(재귀 케이스)을 반복한다.
4. 모든 결과를 합쳐서 반환한다.

모든 자손 조회

이제 재귀 CTE를 사용하여 "전자제품"의 모든 자손을 조회해보자.

```
WITH RECURSIVE descendants AS (
    -- 기본 케이스: 시작 노드 (전자제품)
    SELECT category_id, name, parent_id, 1 AS depth
    FROM category
    WHERE category_id = 1

    UNION ALL

    -- 재귀 케이스: 이전 결과의 자식들을 찾음
    SELECT c.category_id, c.name, c.parent_id, d.depth + 1
    FROM category c
    JOIN descendants d ON c.parent_id = d.category_id
)
SELECT * FROM descendants;
```

- 여기서 핵심은 재귀 케이스에 `JOIN descendants` 부분이다. 자신의 결과를 활용한다.
- 먼저 기본 케이스를 실행하고, 그 결과를 새로운 `descendants`에 담는다.
- 재귀 케이스를 실행한다. 이때 앞서 실행한 결과인 `descendants`를 사용한다.

    - 만약 결과가 있다면 결과를 새로운 `descendants`에 담고 재귀 케이스를 다시 실행한다.
    - 만약 결과가 없다면 종료한다.
- 지금까지 기본 케이스, 재귀 케이스에서 반환한 모든 데이터를 새로운 `descendants`에 담아서 반환한다.
- 최종적으로 `SELECT * FROM descendants`를 통해 `descendants`를 출력한다.

[실행 결과]

category_id	name	parent_id	depth
1	전자제품	NULL	1
4	컴퓨터	1	2
5	스마트폰	1	2
6	가전제품	1	2
7	노트북	4	3
8	데스크탑	4	3
9	태블릿	4	3
10	애플	5	3
11	삼성	5	3
12	TV	6	3
13	냉장고	6	3

한 번의 쿼리로 전자제품 아래의 모든 자손을 조회했다. 각 노드가 몇 번째 깊이에 있는지도 함께 확인할 수 있다.

실행 과정을 단계별로 살펴보자.

```
1차 실행 (기본 케이스 - 호출1):
  - category_id=1 (전자제품) 조회, 조회 결과를 새로운 descendants에 포함

2차 실행 (재귀 케이스 - 호출1):
  - parent_id=1인 노드들 조회 → 컴퓨터(4), 스마트폰(5), 가전제품(6), 결과를 새로운
descendants에 포함

3차 실행 (재귀 케이스 - 호출2):
```

```
  - parent_id IN (4,5,6)인 노드들 조회 → 노트북(7), 데스크탑(8),..., 결과를 새로운 descendants에 포함

4차 실행 (재귀 케이스 - 호출3):
  - parent_id IN (7,8,9,10,11,12,13)인 노드들 조회 → 결과 없음

종료: 더 이상 새로운 행이 없으므로 지금까지 조회한 모든 데이터를 누적해서 새로운 descendants에 담고 종료
```

직접 하나하나 실행할 때 살펴본 다음 내용과 비슷하다.

```sql
-- 1단계: 루트 조회(기본 케이스 - 호출1)
SELECT * FROM category WHERE category_id = 1;

-- 2단계: 1단계 결과의 자식들 조회(재귀 케이스 - 호출1)
SELECT * FROM category WHERE parent_id = 1;

-- 3단계: 2단계 결과의 자식들 조회(재귀 케이스 - 호출2)
SELECT * FROM category WHERE parent_id IN (4, 5, 6);

-- 4단계: 3단계 결과의 자식들 조회 (결과가 없으면 종료) (재귀 케이스 - 호출3)
SELECT * FROM category WHERE parent_id IN (7, 8, 9, 10, 11, 12, 13);
```

CTE와 재귀 쿼리 2

이번에는 다양한 예제를 통해 CTE 활용 방안을 알아보자.

모든 조상 조회

이번에는 반대로 "노트북" 카테고리의 모든 조상을 조회해보자.

```sql
WITH RECURSIVE ancestors AS (
    -- 기본 케이스: 시작 노드 (노트북)
    SELECT category_id, name, parent_id, 1 AS depth
    FROM category
```

```
    WHERE category_id = 7

    UNION ALL

    -- 재귀 케이스: 부모를 찾아 올라감
    SELECT c.category_id, c.name, c.parent_id, a.depth + 1
    FROM category c
    JOIN ancestors a ON c.category_id = a.parent_id
)
SELECT * FROM ancestors;
```

[실행 결과]

category_id	name	parent_id	depth
7	노트북	4	1
4	컴퓨터	1	2
1	전자제품	NULL	3

노트북 → 컴퓨터 → 전자제품 순으로 조상이 조회되었다.

자손 조회와 조상 조회의 차이점을 살펴보자.

- **자손 조회**: `c.parent_id = d.category_id` (자식의 `parent_id`가 부모의 `id`와 같음)
- **조상 조회**: `c.category_id = a.parent_id` (부모의 `id`가 자식의 `parent_id`와 같음)

특정 깊이까지만 조회

재귀 CTE에서 `WHERE` 조건을 사용하면 특정 깊이까지만 조회할 수 있다.

```
WITH RECURSIVE descendants AS (
    SELECT category_id, name, parent_id, 1 AS depth
    FROM category
    WHERE category_id = 1

    UNION ALL
```

```
    SELECT c.category_id, c.name, c.parent_id, d.depth + 1
    FROM category c
    JOIN descendants d ON c.parent_id = d.category_id
    WHERE d.depth < 2  -- 2단계까지만 재귀
)
SELECT * FROM descendants;
```

[실행 결과]

category_id	name	parent_id	depth
1	전자제품	NULL	1
4	컴퓨터	1	2
5	스마트폰	1	2
6	가전제품	1	2

`depth < 2` 조건으로 인해 2단계까지만 조회되었다.

재귀 탈출 조건: 왜 `depth <= 2`가 아닐까?

아마도 이 부분에서 고개를 갸웃했을 것이다. "깊이가 2인 곳까지 조회하고 싶은데, 왜 조건은 `depth < 2`로 작아야 한다고 했을까? `depth <= 2`가 더 직관적이지 않나?"

이 의문을 해결하려면 `WHERE` **조건이 검사하는 대상**이 누구인지 정확히 파악해야 한다.

재귀 쿼리에서 `WHERE` 절은 '새로 만들어질 자식'이 아니라, **'현재 기준이 되는 부모(d)'**를 검사한다. 즉, **"부모의 깊이가 얼마일 때 자식을 찾으러 갈 것인가?"**를 결정하는 멈춤 신호다.

과정을 단계별로 따라가 보면 명확해진다. 우리가 원하는 건 **Depth 2**까지다.

1. **초기 단계 (Anchor Member)**
    - `depth = 1`인 '전자제품'이 선택된다.
    - 이 데이터는 `descendants` 결과 집합에 들어간다.

2. **재귀 단계 1 (Recursive Member)**
    - 이제 `JOIN`을 수행하려고 한다. 이때 `WHERE d.depth < 2` 조건을 확인한다.

- 현재 `descendants`에 포함되어 있는 부모(전자제품)의 `depth`는 **1**이다.
- `1 < 2`는 **참(True)**이다.
- 조건을 통과했으므로 자식을 찾으러 간다. 그 결과 `depth = 2`인 '컴퓨터', '스마트폰', '가전제품'이 조회된다.

3. **재귀 단계 2**
   - 방금 조회된 `depth = 2`인 친구들이 이제 부모가 되어 자식을 찾으려 한다.
   - `WHERE d.depth < 2` 조건을 다시 확인한다.
   - 현재 부모(컴퓨터 등)의 `depth`는 **2**다.
   - `2 < 2`는 **거짓(False)**이다.
   - 조건을 만족하지 못하므로 **더 이상 자식을 찾지 않고 여기서 멈춘다.**

만약 `depth <= 2`라고 썼다면 어떻게 될까?

- **재귀 단계 2**에서 부모의 `depth`가 **2**일 때, `2 <= 2`는 **참(True)**이 된다.
- 그러면 쿼리는 멈추지 않고 컴퓨터의 자식인 '노트북'(`depth = 3`)까지 조회해버린다.
- 결국 우리가 원한 '2단계까지 조회'가 아니라 '3단계까지 조회'가 되는 것이다.

정리하자면 다음과 같다.

- `depth < 2`: 부모가 1단계일 때만 자식을 찾아라. (결과: 자식인 2단계까지 생성됨)
- `depth <= 2`: 부모가 2단계일 때도 자식을 찾아라. (결과: 손자인 3단계까지 생성됨)

그래서 재귀 쿼리에서 깊이를 제한할 때는 **"내가 원하는 깊이보다 작다"**를 조건으로 걸어야 한다.

경로 표시

카테고리의 전체 경로를 이쁘게 문자열로 표시하는 것도 가능하다.

```
WITH RECURSIVE category_path AS (
    -- 기본 케이스: 루트 카테고리
    SELECT category_id, name, parent_id,
           name AS path
    FROM category
    WHERE parent_id IS NULL

    UNION ALL
```

```
    -- 재귀 케이스: 경로에 현재 카테고리 이름 추가
    SELECT c.category_id, c.name, c.parent_id,
           CONCAT(cp.path, ' > ', c.name)
    FROM category c
    JOIN category_path cp ON c.parent_id = cp.category_id
)
SELECT category_id, name, path
FROM category_path
ORDER BY path;
```

[실행 결과]

category_id	name	path
3	식품	식품
2	의류	의류
14	남성의류	의류 > 남성의류
17	바지	의류 > 남성의류 > 바지
16	셔츠	의류 > 남성의류 > 셔츠
15	여성의류	의류 > 여성의류
1	전자제품	전자제품
6	가전제품	전자제품 > 가전제품
12	TV	전자제품 > 가전제품 > TV
13	냉장고	전자제품 > 가전제품 > 냉장고
4	컴퓨터	전자제품 > 컴퓨터
8	데스크탑	전자제품 > 컴퓨터 > 데스크탑
7	노트북	전자제품 > 컴퓨터 > 노트북
9	태블릿	전자제품 > 컴퓨터 > 태블릿
5	스마트폰	전자제품 > 스마트폰
10	애플	전자제품 > 스마트폰 > 애플

11	삼성	전자제품 > 스마트폰 > 삼성

각 카테고리의 전체 경로가 문자열로 표시되었다. 이런 경로 정보는 빵 부스러기(Breadcrumb) 네비게이션에 유용하게 활용할 수 있다.

실무 활용 예시: 카테고리별 상품 수 집계

실무에서 자주 필요한 요구사항 중 하나는 "각 카테고리와 그 하위 카테고리에 속한 상품 수를 보여줘"이다.

먼저 상품 테이블을 만들고 데이터를 입력하자.

```
DROP TABLE IF EXISTS product;

CREATE TABLE product (
    product_id   BIGINT       NOT NULL AUTO_INCREMENT,
    name         VARCHAR(200) NOT NULL,
    category_id  BIGINT       NOT NULL,
    price        INT          NOT NULL,
    PRIMARY KEY (product_id),
    FOREIGN KEY (category_id) REFERENCES category(category_id)
);

-- 상품 데이터 입력
INSERT INTO product (name, category_id, price) VALUES ('맥북 프로', 7, 2500000);
INSERT INTO product (name, category_id, price) VALUES ('맥북 에어', 7, 1500000);
INSERT INTO product (name, category_id, price) VALUES ('삼성 노트북', 7,
1200000);
INSERT INTO product (name, category_id, price) VALUES ('아이맥', 8, 2000000);
INSERT INTO product (name, category_id, price) VALUES ('아이패드', 9, 800000);
INSERT INTO product (name, category_id, price) VALUES ('아이폰 15', 10,
1300000);
INSERT INTO product (name, category_id, price) VALUES ('갤럭시 S24', 11,
1200000);
INSERT INTO product (name, category_id, price) VALUES ('LG TV', 12, 1500000);
INSERT INTO product (name, category_id, price) VALUES ('삼성 냉장고', 13,
2000000);
```

- 전자제품에 9개의 상품을 등록한다.

먼저 전자제품에 속하는 모든 카테고리를 조회해보자.

```
WITH RECURSIVE descendants AS (
    SELECT category_id
    FROM category
    WHERE category_id = 1

    UNION ALL

    SELECT c.category_id
    FROM category c
    JOIN descendants d ON c.parent_id = d.category_id
)
SELECT * FROM descendants;
```

[실행 결과]

category_id
1
4
5
6
7
8
9
10
11
12
13

이제 전자제품 카테고리의 총 상품 수를 구해보자.

```
WITH RECURSIVE descendants AS (
    SELECT category_id
    FROM category
    WHERE category_id = 1

    UNION ALL

    SELECT c.category_id
    FROM category c
    JOIN descendants d ON c.parent_id = d.category_id
)
SELECT COUNT(*) AS product_count
FROM product
WHERE category_id IN (SELECT category_id FROM descendants);
```

[실행 결과]

product_count
9

전자제품과 그 하위 카테고리에 총 9개의 상품이 있다.

CTE의 장점 정리

CTE를 사용한 재귀 쿼리의 장점을 정리하면 다음과 같다.

- **가변 깊이 지원**: 계층의 깊이가 얼마든 상관없이 모든 자손/조상을 조회할 수 있다.
- **단일 쿼리**: 여러 번의 데이터베이스 호출 없이 한 번의 쿼리로 결과를 얻을 수 있다.
- **표준 SQL**: SQL 표준 문법이므로 MySQL, PostgreSQL, SQL Server, Oracle 등 대부분의 RDBMS에서 사용할 수 있다.
- **가독성**: 복잡한 계층 쿼리를 직관적으로 작성할 수 있다.
- **추가 정보**: 깊이(depth), 경로(path) 등의 정보를 함께 계산할 수 있다.

정리

재귀 CTE가 등장하면서 계층 구조 데이터를 다루기가 훨씬 쉬워졌다. 과거에는 복잡한 저장 프로시저나 여러 번의 쿼리가 필요했던 작업을 이제는 단일 쿼리로 처리할 수 있다.

대부분의 계층 구조 요구사항은 **인접 리스트 모델 + CTE 조합**으로 충분히 해결할 수 있다. 하지만 데이터의 양이 매우 많거나, 조회 성능이 극도로 중요한 경우에는 다른 방법을 고려해야 할 수 있다.

다음 수업에서는 조회 성능을 극대화하기 위한 **폐쇄 테이블(Closure Table) 모델**을 학습한다.

폐쇄 테이블 모델 1

인접 리스트 모델과 CTE 조합으로 대부분의 계층 구조 요구사항을 처리할 수 있다. 하지만 다음과 같은 상황에서는 성능 문제가 발생할 수 있다.

- 카테고리 데이터가 수 만 개 이상
- 계층의 깊이가 10단계 이상
- 매우 빈번한 계층 조회

이런 상황에서 조회 성능을 극대화하기 위해 **폐쇄 테이블(Closure Table) 모델**을 사용할 수 있다.

폐쇄 테이블 모델이란?

폐쇄 테이블 모델은 **모든 조상-자손 관계를 미리 계산해서 별도의 테이블에 저장**하는 방식이다.

인접 리스트 모델에서는 각 노드가 직속 부모만 참조한다. 폐쇄 테이블 모델에서는 모든 조상-자손 쌍을 명시적으로 저장한다.

예를 들어, 노트북의 경우:

- **인접 리스트**: 노트북 → 컴퓨터 (직속 부모만)
- **폐쇄 테이블**: 노트북 → 노트북, 노트북 → 컴퓨터, 노트북 → 전자제품 (모든 관계)

핵심 아이디어는 **조회 시점의 재귀를 저장 시점으로 옮기는 것**이다. 데이터를 저장할 때 미리 모든 관계를 계산해두면, 조회할 때는 단순한 JOIN만으로 매우 빠르게 결과를 얻을 수 있다.

참고로 모든 숨겨진 조상-자손 관계까지 미리 다 찾아서 테이블에 가두어(Close) 버렸다는 의미로 `closure`라는 이름을 사용한다.

테이블 설계

폐쇄 테이블 모델은 두 개의 테이블이 필요하다.

1. **노드 테이블**: 실제 데이터를 저장
2. **경로 테이블**: 모든 조상-자손 관계를 저장

```sql
DROP TABLE IF EXISTS category_path;
DROP TABLE IF EXISTS category_closure;

-- 카테고리 테이블 (노드 테이블)
CREATE TABLE category_closure (
    category_id   BIGINT       NOT NULL AUTO_INCREMENT,
    name          VARCHAR(100) NOT NULL,
    PRIMARY KEY (category_id)
);

-- 경로 테이블 (관계 테이블)
CREATE TABLE category_path (
    ancestor_id   BIGINT NOT NULL,  -- 조상 노드
    descendant_id BIGINT NOT NULL,  -- 자손 노드
    depth         INT    NOT NULL,  -- 거리 (깊이 차이)
    PRIMARY KEY (ancestor_id, descendant_id),
    FOREIGN KEY (ancestor_id) REFERENCES category_closure(category_id),
    FOREIGN KEY (descendant_id) REFERENCES category_closure(category_id)
);

-- 조회 성능을 위한 인덱스
CREATE INDEX idx_descendant ON category_path(descendant_id);
CREATE INDEX idx_depth ON category_path(depth);
```

- 테이블 이름은 `category_closure` 대신에 단순하게 `category`를 사용해도 된다. 여기서는 앞서 만든 인접 리스트의 `category` 테이블을 학습 차원에서 남겨두기 위해 이름 뒤에 `_closure`를 사용했다.

`category_path` 테이블의 각 컬럼을 살펴보자.

- `ancestor_id`: 조상 노드의 ID

- `descendant_id`: 자손 노드의 ID
- `depth`: 두 노드 사이의 거리 (자기 자신은 0, 직속 부모/직속 자식은 1)

데이터 입력

폐쇄 테이블에 카테고리 데이터와 경로 데이터를 입력해보자.

여기서는 예제를 단순화 하기 위해 전자제품과 그 자손들만 사용하겠다.

```
-- 카테고리 데이터
INSERT INTO category_closure (category_id, name) VALUES (1, '전자제품');
INSERT INTO category_closure (category_id, name) VALUES (2, '컴퓨터');
INSERT INTO category_closure (category_id, name) VALUES (3, '스마트폰');
INSERT INTO category_closure (category_id, name) VALUES (4, '노트북');
INSERT INTO category_closure (category_id, name) VALUES (5, '데스크탑');
INSERT INTO category_closure (category_id, name) VALUES (6, '애플');
INSERT INTO category_closure (category_id, name) VALUES (7, '삼성');

-- 경로 데이터
-- 자기 자신에 대한 경로 (depth = 0)
-- (ancestor_id, descendant_id, depth)
INSERT INTO category_path VALUES (1, 1, 0);
INSERT INTO category_path VALUES (2, 2, 0);
INSERT INTO category_path VALUES (3, 3, 0);
INSERT INTO category_path VALUES (4, 4, 0);
INSERT INTO category_path VALUES (5, 5, 0);
INSERT INTO category_path VALUES (6, 6, 0);
INSERT INTO category_path VALUES (7, 7, 0);

-- 전자제품(1)의 자식들
INSERT INTO category_path VALUES (1, 2, 1);  -- 전자제품(1) -> 컴퓨터(2)
INSERT INTO category_path VALUES (1, 3, 1);  -- 전자제품(1) -> 스마트폰(3)
INSERT INTO category_path VALUES (1, 4, 2);  -- 전자제품(1) -> 노트북(4) (손자)
INSERT INTO category_path VALUES (1, 5, 2);  -- 전자제품(1) -> 데스크탑(5) (손자)
INSERT INTO category_path VALUES (1, 6, 2);  -- 전자제품(1) -> 애플(6) (손자)
INSERT INTO category_path VALUES (1, 7, 2);  -- 전자제품(1) -> 삼성(7) (손자)

-- 컴퓨터(2)의 자식들
INSERT INTO category_path VALUES (2, 4, 1);  -- 컴퓨터(2) -> 노트북(4)
INSERT INTO category_path VALUES (2, 5, 1);  -- 컴퓨터(2) -> 데스크탑(5)

-- 스마트폰(3)의 자식들
```

```sql
INSERT INTO category_path VALUES (3, 6, 1);  -- 스마트폰(3) -> 애플(6)
INSERT INTO category_path VALUES (3, 7, 1);  -- 스마트폰(3) -> 삼성(7)
```

구조를 시각화하면 다음과 같다.

```
전자제품 (id=1)
├── 컴퓨터 (id=2)
│   ├── 노트북 (id=4)
│   └── 데스크탑 (id=5)
└── 스마트폰 (id=3)
    ├── 애플 (id=6)
    └── 삼성 (id=7)
```

카테고리 테이블의 데이터를 확인해보자.

```sql
SELECT * FROM category_closure;
```

[실행 결과]

category_id	name
1	전자제품
2	컴퓨터
3	스마트폰
4	노트북
5	데스크탑
6	애플
7	삼성

경로 테이블의 데이터를 확인해보자.

```sql
SELECT * FROM category_path ORDER BY ancestor_id, depth, descendant_id;
```

[실행 결과]

ancestor_id	descendant_id	depth
1	1	0
1	2	1
1	3	1
1	4	2
1	5	2
1	6	2
1	7	2
2	2	0
2	4	1
2	5	1
3	3	0
3	6	1
3	7	1
4	4	0
5	5	0
6	6	0
7	7	0

전자제품(`id=1`)을 보면, 자기 자신과 모든 자손들(`2,3,4,5,6,7`)에 대한 경로가 저장되어 있다. 그리고 각 경로별 깊이도 저장되어 있다. 결국 모든 경로를 미리 다 계산해서 저장해둔 것이다.

경로가 이미 계산되어 있기 때문에 불러서 빠르게 조회할 수 있다.

모든 자손 조회

폐쇄 테이블 모델에서 모든 자손을 조회하는 것은 매우 간단하다.

```
-- 전자제품(id=1)의 모든 자손 조회
SELECT *
FROM category_closure c
JOIN category_path p ON c.category_id = p.descendant_id
WHERE p.ancestor_id = 1;
```

[실행 결과]

category_id	name	ancestor_id	descendant_id	depth
1	전자제품	1	1	0
2	컴퓨터	1	2	1
3	스마트폰	1	3	1
4	노트북	1	4	2
5	데스크탑	1	5	2
6	애플	1	6	2
7	삼성	1	7	2

- 부모로 전자제품(`ancestor_id = 1`)을 가지고 있는 모든 카테고리 경로를 조회하면 된다.

재귀 쿼리 없이 단순한 JOIN만으로 모든 자손을 조회했다. 이 쿼리는 인덱스를 사용하므로 매우 빠르게 실행된다.

폐쇄 테이블 모델 2

이번에는 다양한 예제를 통해 폐쇄 테이블 모델의 활용 방안을 알아보자.

모든 조상 조회

조상 조회도 마찬가지로 간단하다.

```
-- 노트북(id=4)의 모든 조상 조회
SELECT *
FROM category_closure c
JOIN category_path p ON c.category_id = p.ancestor_id
WHERE p.descendant_id = 4;
```

- 자식으로 노트북(`descendant_id = 4`)을 가지고 있는 모든 카테고리 경로를 탐색하면 된다.

[실행 결과]

category_id	name	ancestor_id	descendant_id	depth
1	전자제품	1	4	2
2	컴퓨터	2	4	1
4	노트북	4	4	0

- 노트북의 모든 조상인 컴퓨터, 전자제품과 자기 자신이 조회되었다.

직속 자식만 조회

`depth = 1` 조건을 추가하면 직속 자식만 조회할 수 있다.

```
-- 전자제품(id=1)의 직속 자식만 조회
SELECT *
FROM category_closure c
JOIN category_path p ON c.category_id = p.descendant_id
WHERE p.ancestor_id = 1 AND p.depth = 1;
```

[실행 결과]

category_id	name	ancestor_id	descendant_id	depth
2	컴퓨터	1	2	1

3	스마트폰	1	3	1

- 전자제품의 직속 자식인 컴퓨터와 스마트폰만 조회되었다.

특정 깊이까지만 조회

깊이 조건을 사용하면 특정 깊이까지만 조회할 수 있다.

```
-- 전자제품(id=1) 기준 2단계까지만 조회
SELECT c.*, p.depth
FROM category_closure c
JOIN category_path p ON c.category_id = p.descendant_id
WHERE p.ancestor_id = 1 AND p.depth <= 2;
```

[실행 결과]

category_id	name	depth
1	전자제품	0
2	컴퓨터	1
3	스마트폰	1
4	노트북	2
5	데스크탑	2
6	애플	2
7	삼성	2

노드 추가

새로운 노드를 추가할 때는 경로 테이블도 함께 업데이트해야 한다.

```
```

```sql
-- 노트북(id=4) 아래에 '게이밍노트북' 추가
INSERT INTO category_closure (category_id, name) VALUES (8, '게이밍노트북');

-- 경로 추가: 자기 자신
INSERT INTO category_path VALUES (8, 8, 0);

-- 경로 추가: 게이밍노트북의 모든 조상들
-- 노트북의 조상들을 찾아서 게이밍노트북과의 관계를 추가
INSERT INTO category_path (ancestor_id, descendant_id, depth)
SELECT ancestor_id, 8, depth + 1
FROM category_path
WHERE descendant_id = 4;
```

추가된 경로를 확인해보자.

```sql
SELECT * FROM category_path WHERE descendant_id = 8;
```

[실행 결과]

ancestor_id	descendant_id	depth
1	8	3
2	8	2
4	8	1
8	8	0

게이밍노트북(8)은 전자제품(1)에서 3단계, 컴퓨터(2)에서 2단계, 노트북(4)에서 1단계 떨어져 있다.

노드 삭제

노드를 삭제할 때는 해당 노드와 관련된 모든 경로를 삭제해야 한다.

```sql
-- 게이밍노트북(id=8) 삭제
```

```sql
-- 1. 경로 테이블에서 관련 경로 삭제
DELETE FROM category_path WHERE descendant_id = 8;

-- 2. 카테고리 테이블에서 노드 삭제
DELETE FROM category_closure WHERE category_id = 8;
```

만약 자식이 있는 노드를 삭제하려면, 자손들의 경로도 함께 처리해야 한다. 이 부분은 복잡하므로 보통 애플리케이션 레벨에서 처리하거나, 자식이 있는 노드는 삭제하지 못하도록 제약을 둔다.

이처럼 폐쇄 테이블은 노드를 관리하는 부분이 복잡하다. 참고로 서브트리를 이동하려면 기존 관계를 제거하고 새로운 관계를 다시 만들어야 한다. 이렇게 관리가 복잡한 것이 폐쇄 테이블 모델의 주요 단점이다.

인접 리스트 vs 폐쇄 테이블 비교

두 모델을 비교해보자.

항목	인접 리스트	폐쇄 테이블
저장 공간	효율적 (n개 행)	비효율적 ($O(n^2)$ 가능)
직속 자식 조회	빠름	빠름
모든 자손 조회	CTE 필요 (재귀)	매우 빠름 (단순 JOIN)
모든 조상 조회	CTE 필요 (재귀)	매우 빠름 (단순 JOIN)
노드 추가	매우 간단	경로 추가 필요
노드 삭제	간단	경로 삭제 필요
서브트리 이동	매우 간단	복잡
구현 복잡도	낮음	높음

언제 폐쇄 테이블을 사용해야 할까?

폐쇄 테이블 모델은 다음 상황에서 고려해볼 만하다.

- **조회가 매우 빈번한 경우**: 쓰기보다 읽기가 압도적으로 많은 경우. 예를 들어, 쇼핑몰 카테고리는 자주 조회되지만 구조 변경은 드물다.
- **데이터가 매우 많은 경우**: 카테고리가 수만 개 이상이고, 계층의 깊이가 깊은 경우.
- **재귀 쿼리를 사용할 수 없는 경우**: 레거시 시스템이나 MySQL 5.7 이하를 사용하는 경우.

그러나 대부분의 경우에는 **인접 리스트 모델 + CTE**로 충분하다. 폐쇄 테이블 모델은 구현과 유지보수가 복잡하므로, 성능 문제가 실제로 발생하기 시작할 때 도입을 검토하는 것이 좋다.

캐시 활용: 계층 구조가 자주 변경되지 않는다면, 조회 결과를 애플리케이션 레벨에서 캐시하는 것도 좋은 방법이다.

정리

계층 구조 설계가 필요한 이유

- 관계형 데이터베이스는 기본적으로 평면적인 테이블 구조를 가진다.
- 쇼핑몰 카테고리, 조직도, 댓글 등 트리 형태의 계층 구조를 저장하고 조회하기 위한 설계 전략이 필요하다.

인접 리스트 모델

- 계층 구조를 저장하는 가장 직관적이고 널리 사용되는 방법이다.
- 각 행이 자신의 부모를 참조(`parent_id`)하는 자기 참조(Self-Referencing) 방식을 사용한다.
- **장점**: 구조가 직관적이며 데이터의 추가, 수정, 이동이 간단하고 저장 공간 효율이 높다.
- **단점**: 특정 깊이 이상의 모든 자손이나 조상을 한 번의 쿼리로 조회하기 어렵다.

계층 구조 조회의 어려움

- 인접 리스트 모델은 직속 자식이나 부모 조회는 쉽지만, 계층 전체 조회는 까다롭다.
- 애플리케이션 반복 호출은 네트워크 비용과 성능 저하를 유발한다.
- JOIN을 이용한 고정 깊이 조회나 UNION ALL 방식은 계층 깊이가 가변적일 때 대응하기 어렵다.
- 가변적인 깊이를 가진 계층 구조를 효율적으로 조회하기 위해 재귀 쿼리가 필요하다.

CTE와 재귀 쿼리 1

- **CTE(Common Table Expression)**: 쿼리 내에서 임시로 사용하는 이름 있는 결과 집합이며 `WITH` 절로 정의한다.
- **재귀 CTE**: `WITH RECURSIVE`를 사용하여 자기 자신을 참조하는 CTE이다.
- **구조**:
  - **기본 케이스(Anchor)**: 재귀의 시작점.

    - **재귀 케이스(Recursive)**: CTE 자신을 참조하여 반복 실행되는 쿼리.
    - 두 결과는 `UNION ALL`로 결합된다.
- 계층의 깊이를 몰라도 한 번의 쿼리로 모든 자손을 조회할 수 있다.

CTE와 재귀 쿼리 2

- **활용**: 모든 조상 조회, 특정 깊이 제한, 경로(Path) 문자열 생성 등에 활용된다.
- **장점**: 가변 깊이 지원, 단일 쿼리 처리, 표준 SQL 사용, 가독성 우수.
- **실무 예시**: 카테고리별 상품 수 집계 등 복잡한 요구사항을 인접 리스트 모델 + CTE 조합으로 해결 가능하다.

폐쇄 테이블 모델 1

- **개념**: 모든 조상-자손 관계를 미리 계산하여 별도의 경로 테이블에 저장하는 방식이다.
- **구조**:
    - **노드 테이블**: 실제 데이터 저장.
    - **경로 테이블**: 조상(`ancestor`), 자손(`descendant`), 거리(`depth`)를 저장.
- **특징**: 조회 시점의 재귀 연산을 저장 시점으로 옮겨 조회 성능을 극대화한다.
- **조회**: 재귀 없이 단순 JOIN만으로 모든 자손을 매우 빠르게 조회할 수 있다.

폐쇄 테이블 모델 2

- **활용**: 직속 자식 조회, 특정 깊이 조회 등도 인덱스를 타는 단순 쿼리로 가능하다.
- **데이터 조작**:
    - **추가**: 새 노드의 자기 자신 경로 및 모든 조상과의 경로를 계산하여 입력해야 한다.
    - **삭제/이동**: 관련된 모든 경로를 수정해야 하므로 관리가 복잡하다.
- **비교 및 선택**:
    - **인접 리스트 + CTE**: 구현이 쉽고 밸런스가 좋아 대부분의 경우에 적합하다.
    - **폐쇄 테이블**: 조회가 압도적으로 많고 데이터가 방대하며 구조 변경이 적은 경우에 적합하다.