3장

SQL Antipatterns

순진한 트리

나무는 나무일 뿐입니다. 얼마나 더 봐야 하겠습니까?[1]

– 로널드 레이건(Ronald Reagan)

과학 기술 뉴스를 위한 유명한 웹 사이트에 근무하는 소프트웨어 개발자라고 생각해보자.

이건 현대적인 웹 사이트다. 독자들이 답글을 달 수 있고 심지어 답글에 대한 답글도 달 수 있기 때문에, 가지를 뻗어 깊게 확장하는 글 타래를 형성할 수 있다. 이런 답글 타래를 추적하기 위해, 각 답글은 답글을 다는 대상 글에 대한 참조를 가지도록 하는 단순한 해법을 선택했다.

Trees/intro/parent.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  parent_id    BIGINT UNSIGNED,
  comment      TEXT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Comments(comment_id)
);
```

1 (옮긴이) 1966년 로널드 레이건이 캘리포니아 주지사로서 레드우드 국립공원 확장을 반대하면서 한 말.

그러나 곧 답글의 긴 타래를 하나의 SQL 쿼리로 불러오기가 어렵다는 점이 명확해진다. 단지 고정된 깊이까지만, 즉 바로 아래 자식 또는 그 아래 손자뻘 되는 글까지 얻어낼 수 있다. 그렇지만 글타래는 깊이가 무제한이다. 특정 글타래에 대해 모든 답글을 얻기 위해서는 많은 SQL 쿼리를 실행해야 할 것이다.

생각할 수 있는 다른 방법은 모든 글을 불러온 다음, 학교에서 배운 전통적인 트리 알고리즘을 사용해 애플리케이션 메모리 안에서 트리 구조를 구성하는 것이다. 그러나 웹 사이트 담당자가 말하길 보통 하루에 수십 개의 기사가 발표되고 각 기사에는 수백 개의 답글이 달린다고 한다. 누군가 웹 사이트를 볼 때마다 매번 수백만 개의 답글을 정렬하는 것은 비현실적이다.

어떤 논의에 대한 글타래를 간단하고 효율적으로 가져올 수 있도록 글타래를 저장하는 더 좋은 방법이 있어야 한다.

3.1 목표: 계층구조 저장 및 조회하기

데이터가 재귀적 관계를 가지는 것은 흔한 일이다. 데이터는 트리나 계층적 구조가 될 수 있다. 트리 데이터 구조에서 각 항목은 노드라 불린다. 노드는 여러 개의 자식을 가질 수 있고 부모를 하나 가진다. 부모가 없는 최상위 노드를 뿌리(root)라 한다. 가장 아래에 있는 자식이 없는 노드를 종말노드(leaf)라 부른다. 중간에 있는 노드는 간단히 노드(non-leaf)라 한다.

앞에서 설명한 계층적 데이터에서는, 개별 항목을 조회하는 경우, 전체 중 관련된 부분만 포함한 부분집합만 조회하는 경우 또는 데이터 전체를 조회하는 경우가 있을 것이다.

트리 데이터 구조를 가지는 예에는 다음과 같은 것이 포함된다.

조직도: 직원과 관리자의 관계는 트리 구조 데이터의 교과서적 예제다. 조직도는 SQL에 대한 수많은 책과 글에 나타난다. 조직도에서 직원은 관리자를 가지는데 트리 구조에서는 직원의 부모를 나타낸다. 관리자 또한 직원이다.

글타래: 앞에서 봤듯이 답글에 대한 답글의 글타래에 트리가 사용될 수 있다. 이 트리에서 글의 자식은 답글이다.

이 장에서는 안티패턴과 그 해법을 보이는 데 글타래 예제를 사용할 것이다.

3.2 안티패턴: 항상 부모에 의존하기

책과 글에서 흔히 설명하는 초보적 방법은 parent_id 칼럼을 추가하는 것이다. 이 칼럼은 같은 테이블 안의 다른 글을 참조하며, 이 관계를 강제하기 위해 FK 제약조건을 걸 수 있다. 이 테이블을 정의하는 SQL은 다음과 같고, ERD는 그림 3.1과 같다.

Trees/anti/adjacency-list.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  parent_id    BIGINT UNSIGNED,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Comments(comment_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

이 설계는 인접 목록(Adjacency List)이라 불린다. 소프트웨어 개발자가 계층적 데이터를 저장하는 데 사용하는 가장 흔한 설계일 것이다. 다음은 글타래 계층구조를 보여주는 샘플 데이터고, 트리 구조는 그림 3.2에 표현되어 있다.

그림 3.1 인접 목록 ERD

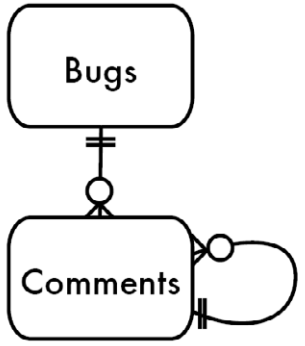
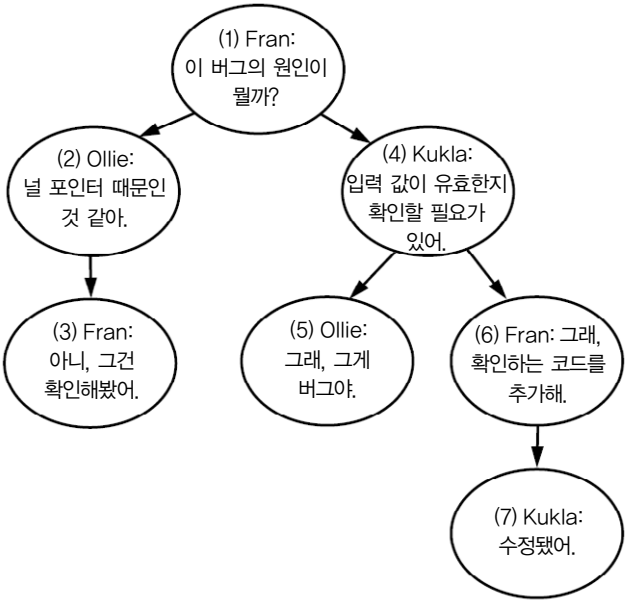


그림 3.2 글타래 트리



comment_id	parent_id	author	comment
1	NULL	Fran	이 버그의 원인이 뭘까?
2	1	Ollie	널 포인터 때문인 것 같아.
3	2	Fran	아니, 그건 확인해봤어.
4	1	Kukla	입력값이 유효한지 확인할 필요가 있어.
5	4	Ollie	그래, 그게 버그야.
6	4	Fran	그래, 확인하는 코드를 추가해.
7	6	Kukla	수정됐어.

인접 목록에서 트리 조회하기

인접 목록은 많은 개발자들이 기본으로 선택하는 방법이지만 트리에서 필요한 가장 흔한 작업 중 하나인 모든 자식 조회하기를 제대로 못 한다면 안티패턴이 될 수 있다.

답글과 그 답글의 바로 아래 자식은 비교적 간단한 쿼리로 얻을 수 있다.

Trees/anti/parent.sql

```
SELECT c1.*, c2.*
FROM Comments c1 LEFT OUTER JOIN Comments c2
  ON c2.parent_id = c1.comment_id;
```

그러나 이 쿼리는 단지 트리의 두 단계만 조회한다. 트리의 특징 중 하나가 어느 깊이까지든 확장될 수 있다는 것이므로, 단계에 상관없이 후손들을 조회할 수 있어야 한다. 예를 들어, COUNT()로 글타래의 답글 수를 계산하거나, SUM()을 이용해 기계 조립에서 부품의 비용 합계[2]를 구할 수 있어야 한다.

인접 목록을 사용하면 이런 종류의 쿼리가 이상해진다. 트리의 각 단계를 조인으로 구해야 하는데, SQL 쿼리에서 조인 회수는 미리 고정되어야 하기 때문이다. 다음 쿼리는 트리에서 4단계까지 가져오지만 그 이상의 깊이에 있는

2 (옮긴이) 부품 조립도 계층적 데이터 구조로 자주 나오는 예제 중 하나다. 한 부품은 다른 부품의 일부가 될 수 있으며 부품 자신도 부품이므로, 글타래나 조직도에서와 마찬가지로 재귀적 계층 관계를 가진다.

데이터는 가져오지 못한다.

Trees/anti/ancestors.sql

```
SELECT c1.*, c2.*, c3.*, c4.*
FROM Comments c1                      -- 1단계
  LEFT OUTER JOIN Comments c2
    ON c2.parent_id = c1.comment_id   -- 2단계
  LEFT OUTER JOIN Comments c3
    ON c3.parent_id = c2.comment_id   -- 3단계
  LEFT OUTER JOIN Comments c4
    ON c4.parent_id = c3.comment_id; -- 4단계
```

또한 이 쿼리는 단계가 깊어질수록 칼럼을 추가하는 방식으로 후손을 포함시키기 때문에 주의를 요한다. 이렇게 하면 COUNT()와 같은 집계 수치를 계산하기가 어려워진다.

인접 목록에서 트리 구조를 조회하는 다른 방법은 글타래의 모든 행을 가져와서 트리처럼 보여주기 전에 애플리케이션에서 계층구조를 만들어내는 것이다.

Trees/anti/all-comments.sql

```
SELECT * FROM Comments WHERE bug_id = 1234;
```

데이터베이스에서 애플리케이션으로 대량의 데이터를 가져오는 방법은 엄청나게 비효율적이다. 꼭대기로부터 전체 트리가 필요한 게 아니라 단지 서브트리만 필요할 수도 있다. 또는 답글의 COUNT()와 같은 데이터의 집계 정보만 필요할 수도 있다.

인접 목록에서 트리 유지하기

인접 목록에서 새로운 노드를 추가하는 것과 같은 일부 연산은 간단해진다는 점을 인정해야겠다.

Trees/anti/insert.sql

```
INSERT INTO Comments (bug_id, parent_id, author, comment)
  VALUES (1234, 7, 'Kukla', 'Thanks!');
```

노드 하나 또는 서브트리를 이동하는 것 또한 쉽다.

Trees/anti/update.sql

```
UPDATE Comments SET parent_id = 3 WHERE comment_id = 6;
```

그러나 트리에서 노드를 삭제하는 것은 좀더 복잡하다. 서브트리 전체를 삭제하려면 FK 제약조건을 만족하기 위해 여러 번 쿼리를 날려 모든 자손을 찾은 다음, 가장 아래 단계부터 차례로 삭제하면서 올라가야 한다.

Trees/anti/delete-subtree.sql

```
SELECT comment_id FROM Comments WHERE parent_id = 4; -- 5와 6 리턴
SELECT comment_id FROM Comments WHERE parent_id = 5; -- 결과 없음
SELECT comment_id FROM Comments WHERE parent_id = 6; -- 7 리턴
SELECT comment_id FROM Comments WHERE parent_id = 7; -- 결과 없음

DELETE FROM Comments WHERE comment_id IN ( 7 );
DELETE FROM Comments WHERE comment_id IN ( 5, 6 );
DELETE FROM Comments WHERE comment_id = 4;
```

삭제할 노드의 자손을 현재 노드의 부모로 이어 붙인다거나 이동하지 않고 항상 함께 삭제한다면, FK에 ON DELETE CASCADE 옵션을 주어 이를 자동화할 수 있다.

그러나 자식이 있는 노드를 삭제하면서 그 자식을 자신의 부모에 이어 붙인다거나 또는 트리의 다른 곳으로 이동하고 싶은 경우에는, 먼저 자식들의 parent_id를 변경한 다음 원하는 노드를 삭제해야 한다.

Trees/anti/delete-non-left.sql

```
SELECT parent_id FROM Comments WHERE comment_id = 6; -- 4 리턴
UPDATE Comments SET parent_id = 4 WHERE parent_id = 6;

DELETE FROM Comments WHERE comment_id = 6;
```

인접 목록 모델을 사용할 때 여러 단계가 필요한 작업의 예를 살펴봤다. 상당히 많은 코드를 작성해야 했는데, 이런 작업은 데이터베이스에서 좀더 단순하고 효율적으로 할 수 있어야 한다.

3.3 안티패턴 인식 방법

다음과 같은 말을 듣는다면, 순진한 트리 안티패턴이 사용되고 있음을 눈치챌 수 있다.

- "트리에서 얼마나 깊은 단계를 지원해야 하지?"
  재귀적 쿼리를 사용하지 않고 어떤 노드의 모든 후손 또는 모든 조상을 얻기 위해 노력하고 있는 것이다. 트리를 제한된 깊이까지만 지원하는 것으로 타협할 수도 있겠지만, 그 다음에 나올 질문도 뻔하다. 어느 정도의 깊이까지 지원해야 할까?
- "트리 데이터 구조를 관리하는 코드는 건드리는 게 겁나."
  계층구조를 관리하는 좀더 정교한 해법 중 하나를 채용했지만, 잘못된 것을 사용하고 있다. 각 기법은 어떤 작업을 쉽게 해주는 대신 다른 작업을 어렵게 한다. 상황에 맞는 최선의 방법이 아닌 다른 방법을 선택한 것일 수도 있다.
- "트리에서 고아 노드를 정리하기 위해 주기적으로 스크립트를 돌려야 해."
  애플리케이션이 트리에서 자식이 있는 노드를 삭제하면서 연결이 끊긴 노드가 생긴 것이다. 복잡한 데이터 구조를 데이터베이스에 저장할 때는, 변경을 한 후에도 일관성 있는 유효한 상태로 데이터 구조를 유지해야 한다. 깨지지 않는 단단한 상태로 데이터 구조를 저장하기 위해 트리거나 FK 제약조건과 함께 이 장의 뒷부분에서 설명하는 방법을 사용할 수 있다.

3.4 안티패턴 사용이 합당한 경우

인접 목록이 애플리케이션에서 필요한 작업을 지원하는 데 적당할 수도 있다. 인접 목록의 강점은 주어진 노드의 부모나 자식을 바로 얻을 수 있다는 것이다. 또한 새로운 노드를 추가하기도 쉽다. 계층적 데이터로 작업하는 데 이 정도만으로도 충분하다면, 인접 목록은 적절한 방법이다.

너무 복잡하게 만들지 마라

나는 어떤 데이터 센터의 장비 관리 애플리케이션을 작성했다. 어떤 장비는 컴퓨터 안에 설치되었다. 예들 들어, 캐싱 디스크 컨트롤러는 랙에 위치한 서버에 설치되었고, 확장 메모리 모듈은 디스크 컨트롤러에 설치됐다.

나는 이런 계층구조를 쉽게 추적하기 위한 SQL 해법이 필요했다. 또한 장비 사용률, 할부상환, 투자수익률(ROI, Return on Investment)과 같은 회계 보고서를 만들어내기 위해 각 장비를 추적해야 했다.

관리자가 말하길 장비는 다른 부품 장비를 포함할 수 있으므로, 이론적으로 트리 깊이는 얼마든지 깊어질 수 있다고 했다. 데이터베이스 내 트리 조작, 사용자 인터페이스, 관리와 리포팅까지 모든 코드를 만드는 데 몇 주가 걸렸다.

그러나 실제로 보니, 장비 관리 애플리케이션에서 장비 구조를 트리로 나타냈을 때 한 단계의 부모–자식 관계 이상으로 깊어지는 경우는 없었다. 내 고객이 장비 요구사항에 대해 이 정도면 충분하다는 점을 인정했다면, 우리는 많은 노력을 줄일 수 있었을 것이다.

어떤 DBMS는 인접 목록 형식으로 저장된 계층구조를 지원하기 위한 SQL 확장 기능을 지원한다. SQL-99 표준에서는 WITH 키워드에 CTE(Common Table Expression)를 사용한 재귀적 쿼리 문법을 정의했다.

Trees/legit/cte.sql

```
WITH CommentTree
    (comment_id, bug_id, parent_id, author, comment, depth)
AS (
    SELECT *, 0 AS depth FROM Comments
    WHERE parent_id IS NULL
  UNION ALL
    SELECT c.*, ct.depth+1 AS depth FROM CommentTree ct
    JOIN Comments c ON (ct.comment_id = c.parent_id)
)
SELECT * FROM CommentTree WHERE bug_id = 1234;
```

Microsoft SQL Server 2005, Oracle 11g, IBM DB2, PostgreSQL 8.4가 CTE를 사용한 재귀적 쿼리를 지원한다.

MySQL, SQLite, Informix는 이 문법을 지원하지 않는다. 여전히 널리 사용되고 있는 Oracle 10g 역시 이를 지원하지 않는다. 나중에는 재귀적 쿼리 문법이 모든 인기 DBMS에서 사용할 수 있게 될 것이라 가정할 수 있다. 그렇게 되면 인접 목록의 사용에도 제약이 없어질 것이다.

Oracle 9i와 10g는 WITH절을 지원하지만 CTE를 통한 재귀적 쿼리 문법은 지원하지 않는다. 대신 START WITH와 CONNECT BY PRIOR를 이용한 전용 문법이 있다. 재귀적 쿼리를 실행하기 위해 이 문법을 사용할 수 있다.

Trees/legit/connect-by.sql

```
SELECT * FROM Comments
START WITH comment_id = 9876
CONNECT BY PRIOR parent_id = comment_id;
```

3.5 해법: 대안 트리 모델 사용

계층적 데이터를 저장하는 데는 인접 목록 모델 외에도 경로 열거(Path Enumeration), 중첩 집합(Nested Sets), 클로저 테이블(Closure Table)과 같은 몇 가지 대안이 있다. 다음 세 절에서는 이런 대안을 통해 글타래와 같이 트리 구조를 저장하고 조회하는 데 있어 안티패턴 절에서 제기했던 문제를 해결하

는 방법을 설명할 것이다.

이런 방법에 익숙해지는 데는 약간 시간이 걸린다. 처음에는 인접 목록보다 복잡해 보일 수 있지만, 인접 목록에서는 매우 어렵거나 비효율적이었던 트리 조작을 쉽게 해준다. 애플리케이션에서 이런 조작을 수행해야 한다면 다음과 같은 방법이 단순한 인접 목록보다 좋은 선택이 될 것이다.

경로 열거

인접 목록의 약점 중 하나는 트리에서 주어진 노드의 조상들을 얻는 데 비용이 많이 든다는 것이다. 경로 열거 방법에서는 일련의 조상을 각 노드의 속성으로 저장해 이를 해결한다.

디렉터리 구조에서도 경로 열거 형태를 볼 수 있다. /usr/local/lib/와 같은 UNIX 경로는 파일 시스템에서의 경로 열거다. usr은 local의 부모고, local은 lib의 부모다.

Comments 테이블에 parent_id 칼럼 대신, 긴 VARCHAR 타입의 path란 칼럼을 정의한다. 이 칼럼에 저장되는 문자열은 트리의 꼭대기부터 현재 행까지 내려오는 조상의 나열로, UNIX 경로와 비슷하다. 심지어 '/'를 구분자로 사용해도 된다.

Trees/soln/path-enum/create-table.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  path         VARCHAR(1000),
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

comment_id	path	author	comment
1	1/	Fran	이 버그의 원인이 뭘까?
2	1/2/	Ollie	널 포인터 때문인 것 같아.
3	1/2/3/	Fran	아니, 그건 확인해봤어.
4	1/4/	Kukla	입력값이 유효한지 확인할 필요가 있어.
5	1/4/5/	Ollie	그래, 그게 버그야.
6	1/4/6/	Fran	그래, 확인하는 코드를 추가해.
7	1/4/6/7/	Kukla	수정됐어.

조상은 현재 행의 경로와 다른 행의 경로 패턴을 비교해 조회할 수 있다. 예를 들어 경로가 1/4/6/7/인 답글 #7의 조상을 찾으려면 다음과 같이 한다.

Trees/soln/path-enum/ancestors.sql

```
SELECT *
FROM Comments AS c
WHERE '1/4/6/7/' LIKE c.path || '%';
```

이렇게 하면 조상의 경로로 만든 패턴 1/4/6/%, 1/4/%, 1/%가 매치된다.

LIKE의 인수를 반대로 하면 후손을 구할 수 있다. 경로가 1/4/인 답글 #4의 후손을 찾으려면 다음과 같이 하면 된다.

Trees/soln/path-enum/descendants.sql

```
SELECT *
FROM Comments AS c
WHERE c.path LIKE '1/4/' || '%';
```

패턴 1/4/%는 후손의 경로 1/4/5/, 1/4/6/, 1/4/6/7/과 매치된다.

트리의 일부나 트리의 꼭대기까지 조상의 연결을 쉽게 조회할 수 있다면, 서브트리에서 노드의 비용 SUM()을 계산한다든가 또는 단순히 노드의 수를 세는 것과 같은 다른 쿼리도 쉽게 할 수 있다. 예를 들어, 답글 #4부터 시작하는 서브트리에서 글쓴이당 답글 수를 세려면 다음과 같이 할 수 있다.

Trees/soln/path-enum/count.sql

```
SELECT COUNT(*)
FROM Comments AS c
WHERE c.path LIKE '1/4/' ¦¦ '%'
GROUP BY c.author;
```

새로운 노드를 삽입하는 방법은 인접 목록 모델에서와 비슷하다. 다른 노드를 수정하지 않고도 종단이 아닌(non-leaf) 노드를 삽입할 수 있다. 새 노드의 부모 경로를 복사한 다음 여기에 새 노드의 아이디를 덧붙이면 된다. 삽입할 때 PK(Primary Key) 값이 자동으로 생성되는 경우라면 먼저 행을 삽입한 다음, 삽입한 새로운 행의 아이디를 이용해 경로를 갱신해야 한다. 예를 들어, MySQL에서는 내장 함수 LAST_INSERT_ID()를 이용해 현재 세션에서 가장 최근에 삽입된 행에 대해 생성된 아이디 값을 얻을 수 있다. 경로의 나머지 부분은 새 노드의 부모로부터 얻어낸다.

Trees/soln/path-enum/insert.sql

```
INSERT INTO Comments (author, comment) VALUES ('Ollie', 'Good job!');

UPDATE Comments
  SET path = (SELECT path FROM Comments WHERE comment_id = 7)
    ¦¦ LAST_INSERT_ID() ¦¦ '/'
WHERE comment_id = LAST_INSERT_ID();
```

경로 열거는 2장 「무단횡단」에서 설명했던 것과 비슷한 단점이 있다. 데이터베이스는 경로가 올바르게 형성되도록 하거나 경로 값이 실제 노드에 대응되도록 강제할 수 없다. 경로 문자열을 유지하는 것은 애플리케이션 코드에 종속되며, 이를 검증하는 데는 비용이 많이 든다. VARCHAR 칼럼의 길이를 아무리 길게 잡아도 결국 제한이 있기 때문에, 엄격히 말하면 지원할 수 있는 트리의 깊이 또한 제한된다.

경로 열거를 사용하면, 구분자 사이의 요소 길이가 같은 경우, 데이터를 계층구조에 따라 쉽게 정렬할 수 있다.[3]

중첩 집합

중첩 집합은 각 노드가 자신의 부모를 저장하는 대신 자기 자손의 집합에 대한 정보를 저장한다. 이 정보는 트리의 각 노드를 두 개의 수로 부호화(encode)해 나타낼 수 있는데, 여기서는 nsleft와 nsright로 부르겠다.

Trees/soln/nested-sets/create-table.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  nsleft       INTEGER NOT NULL,
  nsright      INTEGER NOT NULL,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs (bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

각 노드의 nsleft와 nsright 수는 다음과 같이 주어진다. nsleft 수는 모든 자식 노드의 nsleft 수보다 작아야 하고, nsright는 모든 자식의 nsright 수보다 커야 한다. 이런 숫자는 comment_id 값과는 아무런 상관이 없다.

이 값을 할당하는 쉬운 방법 중 하나는, 트리를 깊이 우선 탐색하면서 값을 하나씩 증가시켜가면서 할당하는 것인데, 자손으로 한 단계씩 내려갈 때는 nsleft에 값을 할당하고 가지를 한 단계씩 올라올 때는 nsright에 값을 할당하는 것이다.

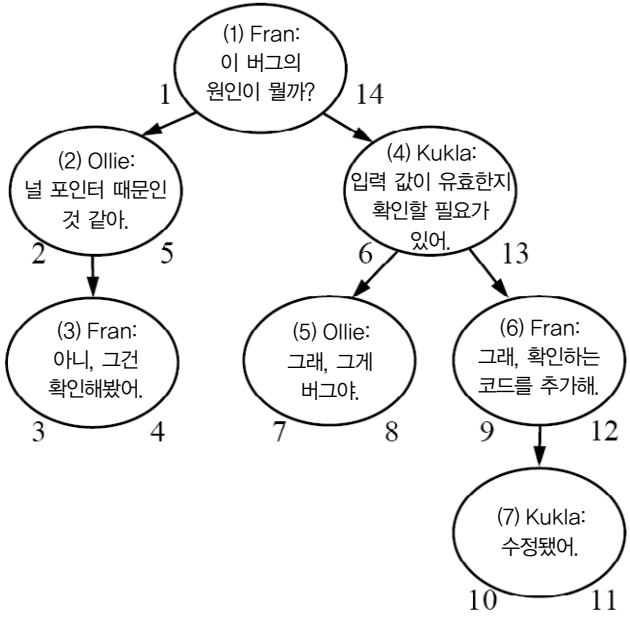
이 설명보다는 그림 3.3을 보는 것이 이해가 쉬울 것이다.

일단 각 노드에 이 값을 할당하면, 이를 이용해 주어진 노드의 조상이나 자손을 찾을 수 있다. 예를 들어, nsleft 값이 현재 노드의 nsleft와 nsright 사이에 있는 노드를 검색하면 답글 #4와 그 자손을 얻을 수 있다.

---

3 그러나 무단횡단 안티패턴과 같이 냄새가 너무 심하다.

그림 3.3 중첩 집합



comment_id	nsleft	Nsright	author	comment
1	1	14	Fran	이 버그의 원인이 뭘까?
2	2	5	Ollie	널 포인터 때문인 것 같아.
3	3	4	Fran	아니, 그건 확인해봤어.
4	6	13	Kukla	입력값이 유효한지 확인할 필요가 있어.
5	7	8	Ollie	그래, 그게 버그야.
6	9	12	Fran	그래, 확인하는 코드를 추가해.
7	10	11	Kukla	수정됐어.

Trees/soln/nested-sets/descendants.sql

```
SELECT c2.*
FROM Comments AS c1
  JOIN Comments as c2
    ON c2.nsleft BETWEEN c1.nsleft AND c1.nsright
WHERE c1.comment_id = 4;
```

답글 #6과 그 조상은 nsright 값이 현재 노드의 숫자 사이에 있는 노드를 검색해 얻을 수 있다.

Trees/soln/nested-sets/ancestors.sql

```
SELECT c2.*
FROM Comments AS c1
  JOIN Comment AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 6;
```

중첩 집합 모델의 주요 강점 중 하나는, 자식을 가진 노드를 삭제했을 때 그 자손이 자동으로 삭제된 노드 부모의 자손이 된다는 것이다. 앞의 설명(그림)에서는 각 노드의 오른쪽, 왼쪽 값이 인접한 형제노드 또는 부모와 비교했을 때 항상 차가 1인 연속된 수열이었지만, 중첩 집합 모델에서 계층구조를 유지하기 위해 반드시 그럴 필요는 없다. 따라서 노드를 삭제해 값들 사이에 간격이 생기더라도 트리 구조에는 아무런 문제가 없다.

예를 들어, 주어진 노드의 깊이를 구하고, 이 노드의 부모를 삭제한 다음, 다시 노드의 깊이를 구해보면, 깊이가 1 줄어든 것을 볼 수 있다.

Trees/soln/nested-sets/depth.sql

```
-- depth = 3으로 나온다
SELECT c1.comment_id, COUNT(c2.comment_id) AS depth
FROM Comment AS c1
  JOIN Comment AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 7
GROUP BY c1.comment_id;
```

```
DELETE FROM Comment WHERE comment_id = 6;

-- depth = 2로 나온다
SELECT c1.comment_id, COUNT(c2.comment_id) AS depth
FROM Comment AS c1
  JOIN Comment AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 7
GROUP BY c1.comment_id;
```

그러나 자식이나 부모를 조회하는 것과 같이 인접 목록 모델에서는 간단했던 일부 쿼리가 중첩 집합 모델에서는 더욱 복잡해진다. 주어진 노드 c1의 부모는 그 노드의 조상이지만 부모노드와 자식노드 사이에 다른 노드는 존재할 수 없다. 따라서 부가적인 외부 조인을 사용해 c1의 조상이고 부모의 자손인 노드를 검색할 수 있다. 이런 노드를 찾지 못한 경우(즉 외부 조인의 결과가 NULL인 경우)가 c1의 조상이자 직접적인 부모가 되는 것이다.

예를 들어, 답글 #6의 부모는 다음과 같이 찾을 수 있다.

Trees/soln/nested-sets/parent.sql

```
SELECT parent.*
FROM Comment AS c
  JOIN Comment AS parent
    ON c.nsleft BETWEEN parent.nsleft AND parent.nsright
  LEFT OUTER JOIN Comment AS in_between
    ON c.nsleft BETWEEN in_between.nsleft AND in_between.nsright
    AND in_between.nsleft BETWEEN parent.nsleft AND parent.nsright
WHERE c.comment_id = 6
  AND in_between.comment_id IS NULL;
```

중첩 집합 모델에서는 노드를 추가, 이동하는 것과 같은 트리 조작도 다른 모델을 사용할 때보다 복잡하다. 새로운 노드를 추가한 경우 새 노드의 왼쪽 값보다 큰 모든 노드의 왼쪽, 오른쪽 값을 다시 계산해야 한다.

여기에는 새로 추가한 노드의 오른쪽 형제들, 그 조상들, 조상의 오른쪽 형제들이 포함된다. 새로 추가한 노드가 종말 노드가 아닌 경우에는 그 자손들도 포함된다. 새로 추가하는 노드가 종말 노드인 경우에는 다음 문장으로 필

요한 모든 것을 갱신할 수 있다.

Trees/soln/nested-sets/insert.sql

```
-- NS값 8과 9에 공간 확보
UPDATE Comment
  SET nsleft = CASE WHEN nsleft >= 8 THEN nsleft+2 ELSE nsleft END,
      nsright = nsright+2
WHERE nsright >= 7;

-- 답글 #5의 자식 생성, NS값 8, 9를 차지
INSERT INTO Comment (nsleft, nsright, author, comment)
  VALUES (8, 9, 'Fran', 'Me too!');
```

중첩 집합 모델은 각 노드에 대해 조작하는 것보다는 서브트리를 쉽고 빠르게 조회하는 것이 중요할 때 가장 잘 맞는다. 노드를 추가하고 이동하는 것은 왼쪽, 오른쪽 값을 재계산해야 하기 때문에 복잡하다. 트리에 노드를 삽입하는 경우가 빈번하다면, 중첩 집합은 좋은 선택이 아니다.

클로저 테이블

클로저 테이블은 계층구조를 저장하는 단순하고 우아한 방법이다. 클로저 테이블은 부모-자식 관계에 대한 경로만을 저장하는 것이 아니라, 트리의 모든 경로를 저장한다.

Comments 테이블에 더해 두 개의 칼럼을 가지는 TreePaths 테이블을 생성한다. TreePaths 테이블의 각 칼럼은 Comments에 대한 FK다.

Trees/soln/closure-table/create-table.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

```
CREATE TABLE TreePaths (
  ancestor    BIGINT UNSIGNED NOT NULL,
  descendant  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY(ancestor, descendant),
  FOREIGN KEY (ancestor) REFERENCES Comments(comment_id),
  FOREIGN KEY (descendant) REFERENCES Comments(comment_id)
);
```

트리 구조에 대한 정보를 Comments 테이블에 저장하는 대신 TreePaths를 사용한다. 이 테이블에는 트리에서 조상/자손 관계를 가진 모든 노드 쌍을 한 행으로 저장한다. 또한 각 노드에 대해 자기 자신을 참조하는 행도 추가한다. 노드 쌍이 어떻게 표현되는지는 그림 3.4를 참조하기 바란다.

ancestor	descendant	ancestor	descendant	ancestor	descendant
1	1	1	7	4	6
1	2	2	2	4	7
1	3	2	3	5	5
1	4	3	3	6	6
1	5	4	4	6	7
1	6	4	5	7	7

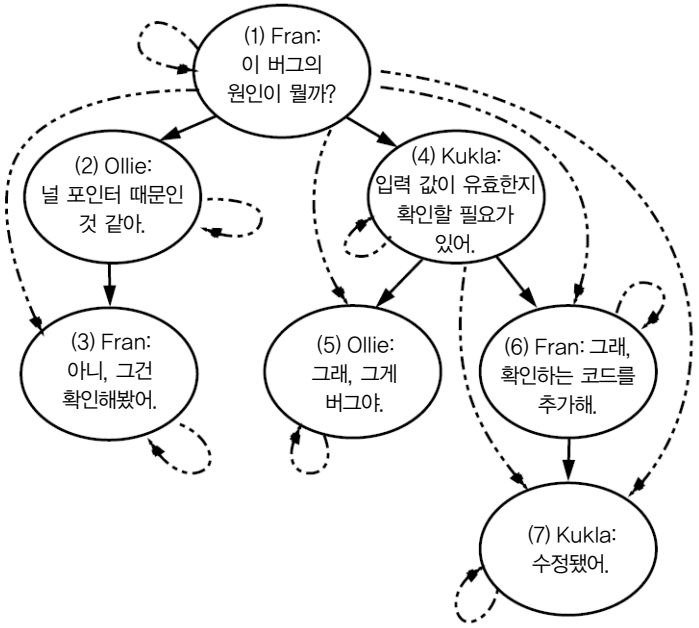
이 테이블에서 조상이나 자손을 가져오는 쿼리는 중첩 집합에서보다 훨씬 직관적이다. 답글 #4의 자손을 얻으려면, TreePaths에서 ancestor가 4인 행을 가져오면 된다.

Trees/soln/closure-table/descendants.sql

```
SELECT c.*
FROM Comments AS c
  JOIN TreePaths AS t ON c.comment_id = t.descendant
WHERE t.ancestor = 4;
```

답글 #6의 조상을 얻으려면, TreePaths에서 descendant가 6인 행을 가져오면 된다.

그림 3.4 클로저 테이블



Trees/soln/closure-table/ancestors.sql

```
SELECT c.*
FROM Comments AS c
  JOIN TreePaths AS t ON c.comment_id = t.ancestor
WHERE t.descendant = 6;
```

새로운 종말 노드, 예를 들어 답글 #5에 새로운 자식을 추가하려면, 먼저 자기 자신을 참조하는 행을 추가한다. 그 다음 TreePaths에서 답글 #5를 descendant로 참조하는 모든 행(답글 #5 자신을 참조하는 행도 포함)을 복사해, descendant를 새로운 답글 아이디로 바꿔 넣는다.

Trees/soln/closure-table/insert.sql

```
INSERT INTO TreePaths (ancestor, descendant)
  SELECT t.ancestor, 8
  FROM TreePaths AS t
  WHERE t.descendant = 5
 UNION ALL
  SELECT 8, 8;
```

종말 노드, 예를 들어 답글 #7을 삭제할 때는 TreePaths에서 답글 #7을 descendant로 참조하는 모든 행을 삭제한다.

Trees/soln/closure-table/delete-left.sql

```
DELETE FROM TreePaths WHERE descendant = 7;
```

서브트리, 예를 들어 답글 #4와 그 자손을 삭제하려면, TreePaths에서 답글 #4를 descendant로 참조하는 모든 행과 답글 #4의 자손을 descendant로 참조하는 모든 행을 삭제한다.

Trees/soln/closure-table/delete-subtree.sql

```
DELETE FROM TreePaths
WHERE descendant IN (SELECT descendant
                     FROM TreePaths
                     WHERE ancestor = 4);
```

TreePaths에서 행을 삭제한다고 답글 자체를 삭제하는 것은 아님에 유의하기 바란다. Comments 예제에서는 이상해 보일지 몰라도, 제품 일람표에서의 분류나 조직도에서의 직원과 같은 다른 종류의 트리를 작업하는 경우라면 이렇게 하는 것이 더 의미 있다. 다른 노드에 대한 관계를 바꾼다고 해서 노드까지 삭제하고 싶지는 않을 것이다. 경로를 별도 테이블에 저장하면 이를 융통성 있게 관리하기가 좋다.

서브트리를 트리 내 다른 위치로 이동하고자 할 때는, 먼저 서브트리의 최상위 노드와 그 노드의 자손들을 참조하는 행을 삭제해 서브트리와 그 조상의 연결을 끊는다. 예를 들어, 답글 #6을 답글 #4의 자식에서 답글 #3의 자식으로 옮기

려면, 다음 삭제문으로 시작한다. #6 자신에 대한 참조는 삭제하지 않도록 주의한다.

Trees/soln/closure-table/move-subtree.sql

```
DELETE FROM TreePaths
WHERE descendant IN (SELECT descendant
                     FROM TreePaths
                     WHERE ancestor = 6)
  AND ancestor IN (SELECT ancestor
                   FROM TreePaths
                   WHERE descendant = 6
                   AND ancestor != descendant);
```

#6 자신을 포함하지 않는 #6의 조상과 #6을 포함한 #6의 자손을 선택해 삭제하면, #6의 조상으로부터 #6과 그 자손으로의 경로를 삭제할 수 있다. 다시 말해, 이 DELETE문은 (1, 6), (1, 7), (4, 6), (4, 7)을 삭제하고, (6, 6), (6, 7)은 삭제하지 않는다.

그러고 나서, 새로운 위치의 조상들과 서브트리의 자손들에 대응하는 행을 추가해서 고아가 된 서브트리를 붙인다. CROSS JOIN 문법으로 카테시안 곱(Cartesian product)을 생성해 새 위치의 조상과 서브트리의 모든 노드를 대응시키는 데 필요한 행을 만들어 낼 수 있다.

Trees/soln/closure-table/move-subtree.sql

```
INSERT INTO TreePaths (ancestor, descendant)
  SELECT supertree.ancestor, subtree.descendant
  FROM TreePaths AS supertree
    CROSS JOIN TreePaths AS subtree
  WHERE supertree.descendant = 3
    AND subtree.ancestor = 6;
```

이렇게 하면 #3을 조상으로 하는 경로와 #6을 자손으로 하는 경로가 새로 생성된다. 따라서 경로 (1, 6), (2, 6), (3, 6), (1, 7), (2, 7), (3, 7)이 새로 생성된다. 이 결과 답글 #6에서 시작하는 서브트리가 답글 #3의 자식으로 위치가 바뀌게 된다. 크로스 조인은, 서브트리가 트리에서 높은 단계 또는 낮은 단계로

이동하더라도, 필요한 모든 경로를 생성한다.

클로저 테이블 모델은 중첩 집합 모델보다 직관적이다. 조상과 자손을 조회하는 것은 두 방법 모두 빠르고 쉽지만, 클로저 테이블이 계층구조 정보를 유지하기가 쉽다. 두 방법 모두 인접 목록이나 경로 열거 방법보다 자식이나 부모를 조회하기 편리하다.

그러나 부모나 자식 노드를 더 쉽게 조회할 수 있도록 TreePaths에 path_length 속성을 추가해 클로저 테이블을 개선할 수 있다. 자기 자신에 대한 path_length는 0, 자식에 대한 path_length는 1, 손자에 대한 path_length는 2와 같은 식이다. 이제 답글 #4에 대한 자식을 찾는 것은 다음과 같이 간단해진다.

Trees/soln/closure-table/child.sql

```
SELECT *
FROM TreePaths
WHERE ancestor = 4 AND path_length = 1;
```

어떤 모델을 사용해야 하는가?

각 모델은 나름대로의 장점과 단점이 있다. 어떤 조작이 가장 효율적이어야 하는지 생각해보고 설계 모델을 선택하기 바란다. 그림 3.5에 각 설계 모델에 대해 어떤 조작이 쉽고 어떤 조작이 어려운지 표시되어 있다. 또한 다음과 같은 각 모델의 장점과 단점을 고려할 수 있다.

- 인접 목록은 가장 흔히 사용되는 모델로 많은 소프트웨어 개발자가 알고 있다.
- WITH나 CONNECT BY PRIOR를 이용한 재귀적 쿼리는 인접 목록 모델을 좀더 효율적으로 만들지만, 이 문법을 지원하는 데이터베이스를 써야 한다.
- 경로 열거는 브레드크럼(breadcrumb)[4]을 사용자 인터페이스에 보여줄 때

는 좋지만, 참조 정합성을 강제하지 못하고 정보를 중복 저장하기 때문에 깨지기 쉬운 구조다.

- 중첩 집합은 영리한 방법이다. 지나치게 영리한 것일 수도 있다. 역시 참조 정합성을 지원하지는 못한다. 트리를 수정하는 일은 거의 없고 조회를 많이 하는 경우 적합하다.
- 클로저 테이블은 가장 융통성 있는 모델이고 한 노드가 여러 트리에 속하는 것을 허용하는 유일한 모델이다. 관계를 저장하기 위한 별도 테이블이 필요하다. 깊은 계층구조를 인코딩하는 데는 많은 행이 필요하고, 계산을 줄이는 대신 저장 공간을 많이 사용하는 트레이드오프(tradeoff)가 발생한다.

SQL로 계층적 데이터를 저장하고 조작하는 것에 대해 배워야 할 것이 더 있다. 계층적 쿼리를 다루는 좋은 책으로 Joe Celko의 『Trees and Hierarchies in SQL for Smarties』[Cel04]가 있다. 다른 책으로 트리뿐 아니라 그래프까지 다루는 Vadim Tropashko의 『SQL Design Patterns』[Tro06]이 있다. 이 책은 좀더 정형적이고 학구적인 스타일이다.

그림 3.5 계층적 데이터 모델 비교

모델	테이블	자식 조회	트리 조회	삽입	삭제	참조 정합성
인접 목록	1	쉽다	어렵다	쉽다	쉽다	가능
재귀적 쿼리	1	쉽다	쉽다	쉽다	쉽다	가능
경로 열거	1	쉽다	쉽다	쉽다	쉽다	불가능
중첩 집합	1	어렵다	쉽다	어렵다	어렵다	불가능
클로저 테이블	2	쉽다	쉽다	쉽다	쉽다	가능

4 (옮긴이) 사용자 인터페이스에서 내비게이션을 편하게 해주는 도구로, 사용자가 프로그램이나 문서에서 현재 위치를 쉽게 추적할 수 있게 해준다. 브레드크럼은 빵 부스러기란 뜻으로, 헨젤과 그레텔이 빵 부스러기로 길에 흔적을 남긴 이야기에서 유래했다.

SQL Antipatterns Tip

계층구조에는 항목과 관계가 있다. 작업에 맞도록 이 둘을 모두 모델링해야 한다.