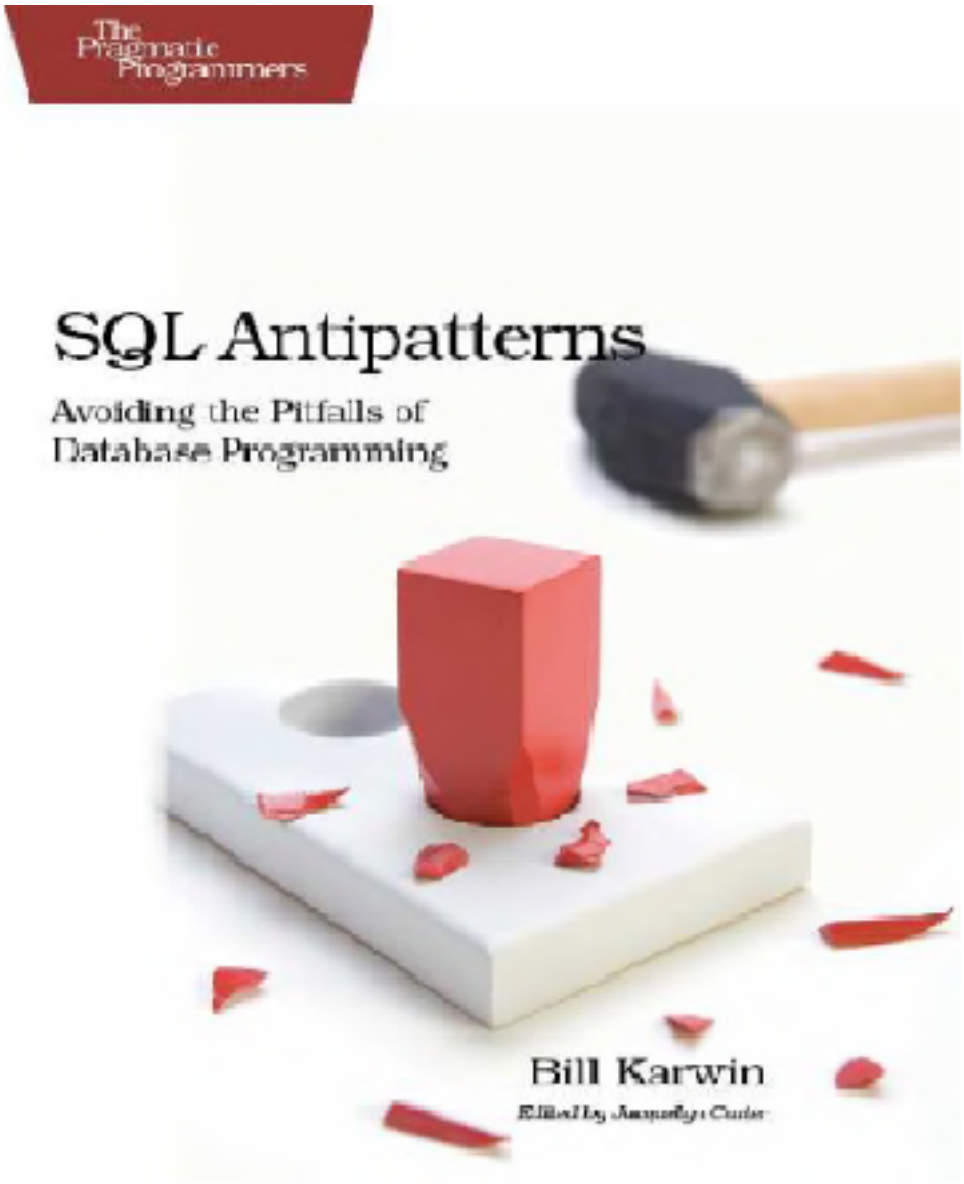
عنوان: **SQL Antipattern #1**
نویسنده: سید مرتضی حسینی
تاریخ: ۱۳۹۳/۰۴/۲۹ ۱۳:۲۰
آدرس: www.dotnettips.info
گروه‌ها: SQL Server, SQL, Antipatterns

در این سلسله نوشتار قصد دارم ترجمه و خلاصه چندین فصل از کتاب ارزشمند (SQL Antipatterns: Avoiding the Pitfalls of Database Programming (Pragmatic Programmers که حاصل تلاش گروه IT موسسه هدایت فرهیختگان جوان می‌باشد، را ارائه نمایم.



بخش اول : Jaywalking

در این بخش در حال توسعه ویژگی نرم افزاری هستیم که در آن هرکاربر به عنوان یک کاربر اصلی برای یک محصول تخصیص داده می‌شود. در طراحی اصلی ما فقط اجازه می دهیم یک کاربر متعلق به هر محصول باشد، اما امکان چنین تقاضایی وجود دارد

که چند کاربر نیز به یک محصول اختصاص داده شوند.

در این صورت، اگر پایگاه داده را به نحوی تغییر دهیم که شناسه‌ی حساب کاربران را در لیستی که با کاما از یکدیگر جدا شده‌اند ذخیره نماییم، خیلی ساده‌تر به نظر می‌رسد نسبت به اینکه بصورت جداگانه آنها را ثبت نماییم.

برنامه نویسان معمولا برای جلوگیری از ایجاد جدول واسطی [1] که رابطه‌های چند به چند زیادی دارد از یک لیست که با کاما داده‌هایش از هم جدا شده‌اند، استفاده می‌کنند. بدین جهت اسم این بخش antipatten, jaywalking می‌باشد، زیرا jaywalking نیز عملیاتی است که از تقاطع جلوگیری می‌کند.

1.1 هدف: ذخیره کردن چندین صفت

طراحی کردن جدولی که ستون آن فقط یک مقدار دارد، بسیار ساده و آسان می‌باشد. شما می‌توانید نوع داده‌ایی که متعلق به ستون می‌باشد را انتخاب نمایید. مثلا از نوع (int,date...)

چگونه می‌توانیم مجموعه‌ایی از مقادیری که به یکدیگر مرتبط هستند را در یک ستون ذخیره نماییم ؟

در مثال ردیابی خطای پایگاه داده، ما یک محصول را به یک کاربر، با استفاده از ستونی که نوع آن integer است، مرتبط می‌نماییم. هر کاربر ممکن است محصولاتی داشته باشد و هر محصول به یک contact اشاره کند. بنابراین یک ارتباط چند به یک بین محصولات و کاربر برقرار است. برعکس این موضوع نیز صادق است؛ یعنی امکان دارد هر محصول متعلق به چندین کاربر باشد و یک ارتباط یک به چند ایجاد شود. در زیر جدول محصولات را به صورت عادی آورده شده است:

```
CREATE TABLE Products (
product_id SERIAL PRIMARY KEY,
product_name VARCHAR(1000),
account_id BIGINT UNSIGNED,
-- . . .
FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder' , 12);
```

2.1 Antipattern : قالب بندی لیست هایی که با کاما از یکدیگر جدا شده اند

برای اینکه تغییرات در پایگاه داده را به حداقل برسانید، می‌توانید نوع شناسه‌ی کاربر را از نوع varchar قرار دهید. در این صورت می‌توانید چندین شناسه‌ی کاربر که با کاما از یکدیگر جدا شده‌اند را در یک ستون ذخیره نمایید. پس در جدول محصولات، شناسه‌ی کاربران را از نوع varchar قرار می‌دهیم.

```
CREATE TABLE Products (
product_id SERIAL PRIMARY KEY,
product_name VARCHAR(1000),
account_id VARCHAR(100), -- comma-separated list
-- . . .
);

INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder' , '12,34' );
```

اکنون مشکلات کارایی و جامعیت داده‌ها را در این راه حل پیشنهادی بررسی می‌نماییم .

بدست آوردن محصولاتی برای یک کاربر خاص

بدلیل اینکه تمامی شناسه‌ی کاربران که بصورت کلید خارجی جدول Products می‌باشند به صورت رشته در یک فیلد ذخیره

شده‌اند و حالت ایندکس بودن آنها از دست رفته است، بدست آوردن محصولاتی برای یک کاربر خاص سخت می‌باشد. به عنوان مثال بدست آوردن محصولاتی که کاربری با شناسه‌ی 12 خریداری نموده به‌صورت زیر می‌باشد:

```
SELECT * FROM Products WHERE account_id REGEXP '[[:<:]]12[[:>:]]' ;
```

بدست آوردن کاربرانی که یک محصول خاص خریداری نموده اند

Join کردن account_id با Accounts table یعنی جدول مرجع نامناسب و غیر معمول می‌باشد. مساله مهمی که وجود دارد این است که زمانیکه بدین صورت شناسه‌ی کاربران را ذخیره می‌نماییم، ایندکس بودن آنها از بین می‌رود. کد زیر کاربرانی که محصولی با شناسه‌ی 123 خریداری کرده‌اند را محاسبه می‌کند.

```
SELECT * FROM Products AS p JOIN Accounts AS a
ON p.account_id REGEXP '[[:<:]]' || a.account_id || '[[:>:]]'
WHERE p.product_id = 123;
```

ایجاد توابع تجمیعی [2]

مبنای چنین توابعی از قبیل avg ,()count ,()sum() بدین صورت می‌باشد که بر روی گروهی از سطرها اعمال می‌شوند. با این حال با کمک ترفندهایی می‌توان برخی از این توابع را تولید نماییم هر چند برای همه آن‌ها این مساله صادق نمی‌باشد. اگر بخواهیم تعداد کاربران برای هر محصول را بدست بیاوریم ابتدا باید طول لیست شناسه‌ی کاربران را محاسبه کنیم، سپس کاما را از این لیست حذف کرده و طول جدید را از طول قبلی کم کرده و با یک جمع کنیم. نمونه کد آن در زیر آورده شده است:

```
SELECT product_id, LENGTH(account_id) - LENGTH(REPLACE(account_id, ',' , '' )) + 1
AS contacts_per_product
FROM Products;
```

ویرایش کاربرانی که یک محصول خاص خریداری نموده‌اند

ما به راحتی می‌توانیم یک شناسه‌ی جدید را به انتهای این رشته اضافه نماییم؛ فقط مرتب بودن آن را بهم میریزیم. در نمونه اسکریپت زیر گفته شده که در جدول محصولات برای محصولی با شناسه‌ی 123 بعد از کاما یک کاربر با شناسه‌ی 56 درج شود:

```
UPDATE Products
SET account_id = account_id || ',' || 56
WHERE product_id = 123;
```

برای حذف یک کاربر از لیست کافیست دو دستور sql را اجرا کرد: اولی برای fetch کردن شناسه‌ی کاربر از لیست و بعدی برای ذخیره کردن لیست ویرایش شده. بطور مثال تمامی افرادی که محصولی با شناسه‌ی 123 را خریداری کرده‌اند، از جدول محصولات انتخاب می‌کنیم. برای حذف کاربری با شناسه‌ی 34 از لیست کاربران، بر حسب کاما مقادیر لیست را در آرایه می‌ریزیم، شناسه‌ی مورد نظر را جستجو می‌کنیم و آن را حذف می‌کنیم. دوباره مقادیر درون آرایه را بصورت رشته درمیاوریم و جدول محصولات را با لیست جدید کاربران برای محصولی با شناسه‌ی 123 بروزرسانی می‌کنیم.

```
<?php
$stmt = $pdo->query(
"SELECT account_id FROM Products WHERE product_id = 123");
$row = $stmt->fetch();
$contact_list = $row['account_id' ];


// change list in PHP code
$value_to_remove = "34";
$contact_list = split(",", $contact_list);
$key_to_remove = array_search($value_to_remove, $contact_list);
unset($contact_list[$key_to_remove]);
$contact_list = join(",", $contact_list);
$stmt = $pdo->prepare(
"UPDATE Products SET account_id = ?
WHERE product_id = 123");
$stmt->execute(array($contact_list));
```

اعتبارسنجی شناسه‌ی محصولات

به دلیل آنکه نوع فیلد account_id،varchar می‌باشد احتمال این وجود دارد داده‌ای نامعتبر وارد نماییم چون پایگاه داده‌ها هیچ عکس العملی یا خطایی را نشان نمی‌دهد، فقط از لحاظ معنایی دچار مشکل می‌شویم. در نمونه زیر banana یک داده‌ی غیر معتبر می‌باشد و ارتباطی با شناسه‌ی کاربران ندارد.

```
INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder' , '12,34,banana' );
```

انتخاب کردن کاراکتر جداکننده

نکته قابل توجه این است که کاراکتری که بعنوان جداکننده در نظر می‌گیریم باید در هیچکدام از داده‌های ورودی ما امکان بودنش وجود نداشته باشد .

محدودیت طول لیست

در زمان طراحی جدول، برای هر یک از فیلدها باید حداکثر طولی را قرار دهیم. به عنوان نمونه ما اگر varchar(30) در نظر بگیریم بسته به تعداد کاراکترهایی که داده‌های ورودی ما دارند می‌توانیم جدول را پرنماییم. اسکریپت زیر شناسه‌ی کاربرانی که محصولی با شناسه‌ی 123 را خریده‌اند، ویرایش می‌کند. با این تفاوت که با توجه به محدودیت لیست، در نمونه‌ی اولی شناسه‌ی کاربران دو کاراکتری بوده و 10داده بعنوان ورودی پذیرفته است در حالیکه در نمونه‌ی دوم بخاطر اینکه شناسه‌ی کاربران شش کاراکتری می‌باشد فقط 4 شناسه می‌توانیم وارد نماییم.

```
UPDATE Products SET account_id = '10,14,18,22,26,30,34,38,42,46'
WHERE product_id = 123;
```

```
UPDATE Products SET account_id = '101418,222630,343842,467790'
WHERE product_id = 123;
```

3.1 موارد تشخیص این Antipattern:

سؤالات زیر نشان می‌دهند که Jaywalking antipattern مورد استفاده قرار گرفته است:

• حداکثر پذیرش این لیست برای داده ورودی چه میزان است؟

• چه کاراکتری هرگز در داده‌های ورودی این لیست ظاهر نمی‌شود؟ **4.1 مواردی که استفاده از این Antipattern مجاز است:**

دی نرمال کردن کارایی را افزایش می‌دهد. ذخیره کردن شناسه‌ها در یک لیست که با کاما از یکدیگر جدا شده‌اند نمونه بارزی از دی نرمال کردن می‌باشد. ما زمانی می‌توانیم از این مدل استفاده نماییم که به جدا کردن شناسه‌ها از هم نیاز نداشته باشیم. به همین ترتیب، اگر در برنامه، شما یک لیست را از یک منبع دیگر با این فرمت دریافت نمایید، به سادگی آن را در پایگاه داده خود به همان فرمت بصورت کامل می‌توانید ذخیره و بازیابی نمایید و احتیاجی به مقادیر جداگانه ندارید. درهنگام استفاده از پایگاه داده‌های دی‌نرمال دقت کنید. برای شروع از پایگاه داده نرمال استفاده کنید زیرا به کدهای برنامه شما امکان انعطاف بیشتری می‌دهد و کمک می‌کند تا پایگاه داده‌ها تمامیت داده(Data integrity) را حفظ کند. **5.1 راه حل: استفاده کردن از جدول واسط**

در این روش شناسه‌ی کاربران را در جدول جداگانه‌ایی که هر کدام از آنها یک سطر را به خود اختصاص داده اند، ذخیره می‌نماییم.

```
CREATE TABLE Contacts ( product_id BIGINT UNSIGNED NOT NULL,
account_id BIGINT UNSIGNED NOT NULL,
PRIMARY KEY (product_id, account_id),
FOREIGN KEY (product_id) REFERENCES Products(product_id),
FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);
```

جدول Contacts یک جدول رابطه ایی بین جداول Products,Accounts

بدست آوردن محصولات برای کاربران و موارد مربوط به آن

ما براحتی می‌توانیم تمامی محصولاتی که مختص به یک کاربر هستند را بدست آوریم. در این شیوه خاصیت ایندکس بودن شناسه‌ی کاربران حفظ می‌شود به همین دلیل queryهای آن برای خواندن و بهینه کردن راحت‌تر می‌باشند. در این روش به کاراکتری برای جدا کردن ورودی‌ها از یکدیگر نیاز نداریم چون هر کدام از آنها در یک سطر جداگانه ثبت می‌شوند. برای ویرایش کردن کاربرانی که یک محصول را خریداری نموده اند، کافیست یک سطر از جدول واسط را اضافه یا حذف نماییم. درنمونه کد زیر، ابتدا در جدول Contacts کاربری با شناسه‌ی 34 که محصولی با شناسه‌ی 456 را خریداری کرده، درج شده است و در خط بعد عملیات حذف با شرط آنکه شناسه‌ی کاربر و محصول به ترتیب 34،456 باشد روی جدول Contacts اعمال شده است.

```
INSERT INTO Contacts (product_id, account_id) VALUES (456, 34);

DELETE FROM Contacts WHERE product_id = 456 AND account_id = 34;
```

ایجاد توابع تجمیعی

به عنوان نمونه در مثال زیر براحتی ما می‌توانیم تعداد محصولات در هر حساب کاربری را بدست آوریم:

```
SELECT account_id, COUNT(*) AS products_per_account
FROM Contacts
GROUP BY account_id;
```

اعتبارسنجی شناسه محصولات

از آنجاییکه مقادیری که در جدول قرار دارند کلید خارجی می‌باشند میتوان صحت اعتبار آنها را بررسی نمود. بعنوان مثال Contacts.account_id به Account.account_id اشاره می‌کند. در ضمن برای هر فیلد نوع آن را می‌توان مشخص کرد تا فقط همان نوع داده را بپذیرد.

محدودیت طول لیست

نسبت به روش قبلی تقریبا در این حالت محدودیتی برای تعداد کاراکترهای ورودی نداریم.

مزیت‌های دیگر استفاده از جدول واسط

کارایی روش دوم بهتر از حالت قبلی می‌باشد چون ایندکس بودن شناسه‌ها حفظ شده است. همچنین براحتی میتوانیم فیلدی را به این جدول اضافه نماییم مثلا (time, date...)

Intersection Table [1]

Aggregate [2]

نظرات خوانندگان

نویسنده: م منفرد
تاریخ: ۱۳۹۳/۰۴/۳۰ ۰:۴۸

سلام
منظورتان از گروه it موسسه هدایت فرهیختگان چه بود؟ ایشان کتاب را ترجمه کردند؟ چگونه می‌توان آن را تهیه کرد؟
با تشکر

نویسنده: فرید
تاریخ: ۱۳۹۳/۰۶/۲۶ ۲۰:۱۸

سلام؛ به نظر میرسه این خصیصه مربوط به mysql است. من نتوانستم آن‌را در sqlserver اجرا کنم.

عنوان: **SQL Antipattern #2**
نویسنده: سید مرتضی حسینی
تاریخ: ۱۳۹۳/۰۴/۳۰ ۱۳:۲۵
آدرس: www.dotnettips.info
گروه‌ها: SQL Server, SQL, Antipatterns

بخش دوم : Naive Trees

فرض کنید یک وب سایت حرفه‌ای خبری یا علمی-پژوهشی داریم که قابلیت دریافت نظرات کاربران را در مورد هر مطلب مندرج در سایت یا نظرات داده شده در مورد آن مطالب را دارا می‌باشد. یعنی هر کاربر علاوه بر توانایی اظهار نظر در مورد یک خبر یا مطلب باید بتواند پاسخ نظرات کاربران دیگر را نیز بدهد. اولین راه حلی که برای طراحی این مطلب در پایگاه داده به ذهن ما می‌رسد، ایجاد یک زنجیره با استفاده از کد sql زیر می‌باشد:

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

البته همان طور که پیداست بازیابی زنجیره‌ای از پاسخ‌ها در یک پرس‌وجوی sql کار سختی است. این نخ‌ها معمولا عمق نامحدودی دارند و برای به دست آوردن تمام نخ‌های یک زنجیره باید پرس‌وجوهای زیادی را اجرا نمود.

ایده‌ی دیگر می‌تواند بازیابی تمام نظرها و ذخیره‌ی آن‌ها در حافظه‌ی برنامه به صورت درخت باشد. ولی این روش برای ذخیره هزاران نظری که ممکن است در سایت ثبت شود و علاوه بر آن مقالات جدیدی که اضافه می‌شوند، تقریبا غیرعملی است.

1.2 هدف: ذخیره و ایجاد پرس‌وجو در سلسله‌مراتب

وجود سلسله مراتب بین داده‌ها امری عادی محسوب می‌گردد. در ساختار داده‌ای درختی هر ورودی یک گره محسوب می‌گردد. یک گره ممکن است تعدادی فرزند و یک پدر داشته باشد. گره اول پدر ندارد، ریشه و گره فرزند که فرزند ندارد، برگ و گره‌ای دیگر، گره‌های غیربرگ نامیده می‌شوند.

مثال‌هایی که از ساختار درختی داده‌ها وجود دارد شامل موارد زیر است:

Organizational chart : در این ساختار برای مثال در ارتباط بین کارمندان و مدیر، هر کارمند یک مدیر دارد که نشان‌دهنده‌ی پدر یک کارمند در ساختار درختی است. هر مدیر هم یک کارمند محسوب می‌گردد.

Threaded discussion : در این ساختار برای مثال در سیستم نظردهی و پاسخ‌ها، ممکن است زنجیره‌ای از نظرات در پاسخ به نظرات دیگر استفاده گردد. در درخت، فرزندان یک گره‌ی نظر، پاسخ‌های آن گره هستند.

در این فصل ما از مثال ساختار دوم برای نشان دادن Antipattern و راه حل آن بهره می‌گیریم.

2.2 Antipattern : همیشه مبتنی بر یکی از والدین

راه حل ابتدایی و ناکارآمد

اضافه نمودن ستون parent_id . این ستون، به ستون نظر در همان جدول ارجاع داده می‌شود و شما می‌توانید برای اجرای این رابطه از قید کلید خارجی استفاده نمایید. پرس‌وجویی که برای ساخت مثالی که ما در این بحث از آن استفاده می‌کنیم در ادامه آمده است:

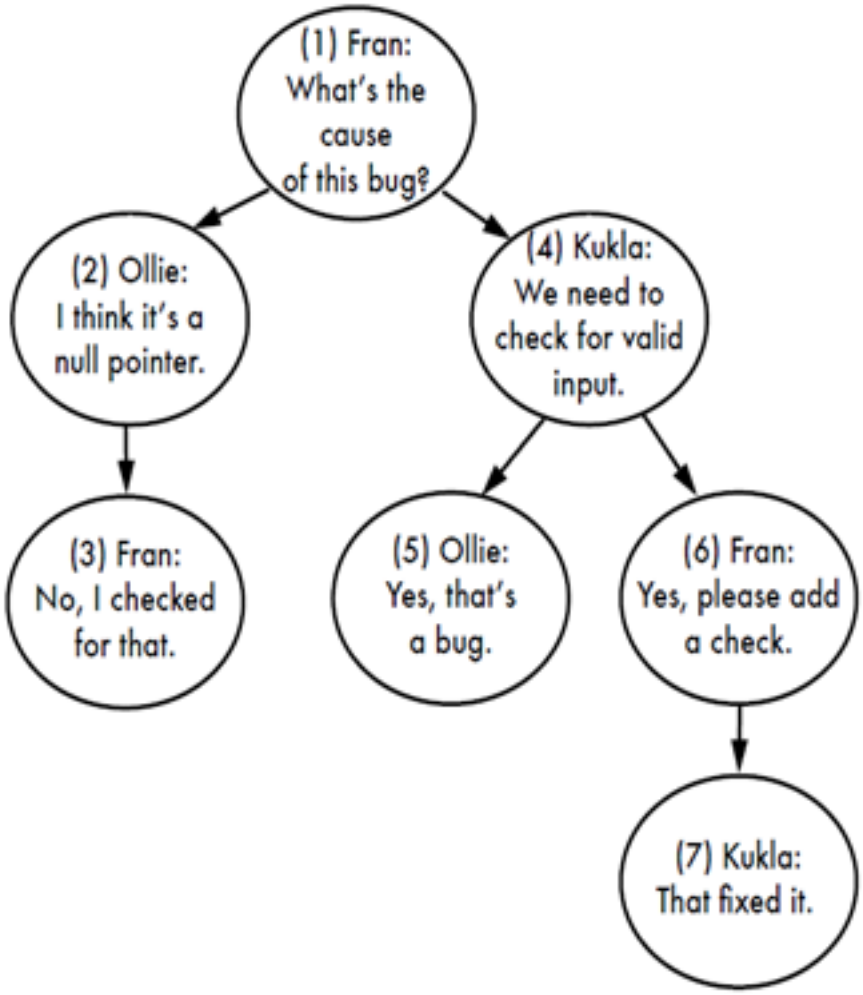
```
CREATE TABLE Comments (  comment_idSERIAL PRIMARY KEY,
parent_idBIGINT UNSIGNED,
bug_idBIGINT UNSIGNED NOT NULL,
author BIGINT UNSIGNED NOT NULL,
comment_dateDATETIME NOT NULL,
comment TEXT NOT NULL,
FOREIGN KEY (parent_id)REFERENCES Comments(comment_id),
FOREIGN KEY (bug_id)         REFERENCES Bugs(bug_id),
FOREIGN KEY(author)          REFERENCES Accounts(account_id)
);
```

مثالی از پرس‌وجوی فوق را می‌توانید در زیر ببینید:

comment_id	parent_id	author	comment
1	NULL	Fran	What's the cause of this bug?
2	1	Ollie	I think it's a null pointer.
3	2	Fran	No, I checked for that.
4	1	Kukla	We need to check for invalid input.
5	4	Ollie	Yes, that's a bug.
6	4	Fran	Yes, please add a check.
7	6	Kukla	That fixed it.

لیست مجاورت :

لیست مجاورت خود می‌تواند به عنوان یک antipattern در نظر گرفته شود. البته این مطلب از آنجایی نشأت می‌گیرد که این روش توسط بسیاری از توسعه‌دهندگان مورد استفاده قرار می‌گیرد ولی نتوانسته است به عنوان راه حل برای معمول‌ترین وظیفه‌ی خود، یعنی ایجاد پرس‌وجو بر روی کلیه فرزندان، باشد.



• با استفاده از پرس‌وجوی زیر می‌توان فرزند بلافاصله‌ی یک "نظر" را برگرداند:

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

ضعف پرس‌وجوی فوق این است که فقط می‌تواند دو سطح از درخت را برای شما برگرداند. در حالیکه خاصیت درخت این است که شما را قادر می‌سازد بدون هیچ گونه محدودیتی فرزندان و نوه‌های متعدد (سطوح بی‌شمار) برای درخت خود تعریف کنید.
بنابراین به ازای هر سطح اضافه باید یک join به پرس‌جوی خود اضافه نمایید. برای مثال اگر پرس‌وجوی زیر می‌تواند درختی با چهار سطح برای شما برگرداند ولی نه بیش از آن:

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

این پرس‌وجو به این دلیل که با اضافه شدن ستون‌های دیگر، نوه‌ها را سطوح عمیق‌تری برمی‌گرداند، پرس‌وجوی مناسبی نیست.

در واقع استفاده از توابع تجمیعی ، مانند COUNT() مشکل می‌شود.

راه دیگر برای به دست آوردن ساختار یک زیردرخت از لیست مجاورت برای یک برنامه، این است که سطرهای مورد نظر خود را از مجموعه بازیابی نموده و سلسه‌مراتب مورد نظر را در حافظه بازیابی نماییم و از آن به عنوان درخت استفاده نماییم:

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

نگهداری کردن یک درخت با استفاده از لیست مجاورت

البته برخی از عملکردها با لیست مجاورت به خوبی انجام می‌گیرد. برای مثال اضافه نمودن یک گره (نظر)، مکان‌یابی مجدد برای یک گره یا یک زیردرخت .

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

بازیابی دوباره مکان یک نود یا یک زیردرخت نیز آسان است:

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

با این حال حذف یک گره از یک درخت در این روش پیچیده است. اگر بخواهیم یک زیردرخت را حذف کنید باید چندین پرس‌وجو برای پیدا کردن تمام نوه‌ها بنویسیم و سپس حذف نوه‌ها را از پایین‌ترین سطح شروع کرده و تا جایی که قید کلید خارجی برقرار شود ادامه دهیم. البته می‌توان از کلید خارجی با تنظیم ON DELETE CASCADE استفاده کرد تا این کارها به طور خودکار انجام گیرد.
حال اگر بخواهیم یک نود غیر برگ را حذف کرده یا فرزندان آن را در درخت جابجا کنیم، ابتدا باید parent_id فرزندان آن نود را تغییر داده و سپس نود مورد نظر را حذف می‌کنیم:

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

3.2 موارد تشخیص این Antipattern:

سؤالات زیر نشان می‌دهند که Naive Trees antipattern مورد استفاده قرار گرفته است:
چه تعداد سطح برای پشتیبانی در درخت نیاز خواهیم داشت؟

من همیشه از کار با کدی که ساختار داده‌ی درختی را مدیریت می‌کند، می‌ترسم

من باید اسکریپتی را به طور دوره‌ای اجرا نمایم تا سطرهای یتیم موجود در درخت را حذف کند.

4.2 مواردی که استفاده از این Antipattern مجاز است:

قدرت لیست مجاورت در بازیابی پدر یا فرزند مستقیم یک نود می‌باشد. قرار دادن یک سطر هم در لیست مجاورت کار ساده‌ای است. اگر این عملیات، تمام آن چیزی است که برای انجام کارتان مورد نیاز شما است، بنابراین استفاده از لیست مجاورت می‌تواند مناسب باشد.
برخی از برندهای RDBMS از افزونه‌هایی پشتیبانی می‌کنند که قابلیت ذخیره‌ی سلسله مراتب را در لیست مجاورت ممکن می‌سازد. مثلا SQL-99، پرس‌وجوی بازگشتی را تعریف می‌کند که مثال آن در ادامه آمده است:

```
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 CommentTreect
JOIN Comments c ON (ct.comment_id = c.parent_id)
)
SELECT * FROM CommentTree WHERE bug_id = 1234;
```

Oracle 9i.پشتیبانی می‌کنند Microsoft SQL Server 2005، Oracle 11g، IBM DB2 و PostgreSQL 8.4 نیز از پرس‌وجوی بازگشتی و 10g از عبارت WITH استفاده می‌کنند، ولی نه برای پرس‌وجوهای بازگشتی. در عوض می‌توانید از پرس‌وجوی زیر برای ایجاد پرس‌وجوی بازگشتی استفاده نمایید:

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

5.2 راه حل: استفاده از مدل‌های درختی دیگر

جایگزین‌های دیگری برای ذخیره‌سازی داده‌های سلسله مراتبی وجود دارد. البته برخی از این راه حل‌ها ممکن است در لحظه‌ی اول پیچیدتر از لیست مجاورت به نظر آیند، ولی برخی از عملیات درخت که در لیست مجاورت بسیار سخت یا ناکارآمد است، را آسان‌تر می‌کنند. **شمارش مسیر :**

مشکل پرهزینه بودن بازیابی نیاکان یک گره که در روش لیست مجاورت وجود داشت در روش شمارش مسیر به این ترتیب حل شده است: اضافه نمودن یک صفت به هر گره که رشته‌ای از نیکان آن صفت در آن ذخیره شده است.

در جدول Comments به جای استفاده از parent_id، یک ستون به نام path که توع آن varchar است تعریف شده است. رشته‌ای که در این ستون تعریف شده است، ترتیبی از فرزندان این سطر از بالا به پایین درخت است. مانند مسیری که در سیستم عامل UNIX، برای نشان دادن مسیر در سیستم فایل استفاده شده است. شما می‌توانید از / به عنوان کاراکتر جداکننده استفاده نمایید. دقت کنید برای درست کار کردن پرس‌وجوها حتما در آخر مسیر هم این کاراکتر را قرار دهید. پرس‌وجوی تشکیل چنین درختی به شکل زیر است:

```
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	What's the cause of this bug?
2	1/2/	Ollie	I think it's a null pointer.
3	1/2/3/	Fran	No, I checked for that.
4	1/4/	Kukla	We need to check for invalid input.
5	1/4/5/	Ollie	Yes, that's a bug.
6	1/4/6/	Fran	Yes, please add a check.
7	1/4/6/7/	Kukla	That fixed it.

در این روش، هر گره مسیری دارد که شماره خود آن گره هم در آنتهای آن مسیر قرار دارد. این به دلیل درست جواب دادن پرس‌وجوهای ایجاد شده است.

می‌توان نیاکان را با مقایسه‌ی مسیر سطر کنونی با مسیر سطر دیگر به دست آورد. برای مثال برای یافتن نیاکان گره (نظر) شماره‌ی 7 که مسیر آن 7/6/4/1/ می‌باشد، می‌توان چنین نوشت:

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

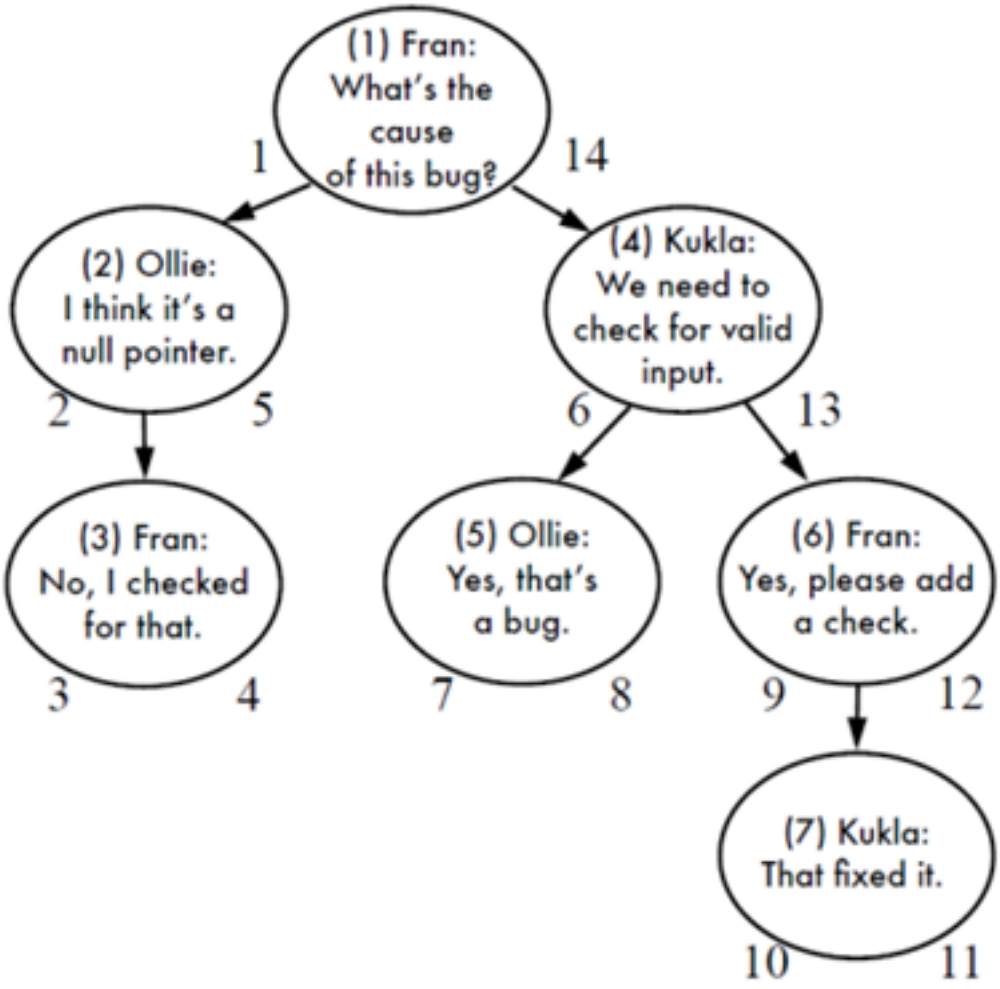
این پرس‌وجو الگوهایی را می‌یابد که از مسیرهای 1/4/6/%، 1/4/% و 1/% نشأت می‌گیرد.
همچنین فرزندان (نوه‌های) یک گره، مثلا گره‌ی 4 را که مسیرش 1/4/ است را می‌توان با پرس‌وجوی زیر یافت:

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

الگوی 1/4/% با مسیرهای 1/5/4/، 1/6/4/ و 1/7/6/4/ تطابق می‌یابد.
همچنین می‌توان پرس‌وجوهای دیگری را نیز در این مسیر به سادگی انجام داد؛ مانند محاسبه‌ی مجموع هزینه‌ی گره‌ها در یک زیردرخت یا شمارش تعداد گره‌ها.
اضافه نمودن یک گره هم مانند ساختن خود مدل است. می‌توان یک گره‌ی غیر برگ را بدون نیاز به اصلاح هیچ سطری اضافه نمود. برای این کار مسیر را را از گره‌ی پدر کپی کرده و در انتها شماره‌ی خود گره را به آن اضافه می‌کنیم.
از مشکلات این روش می‌توان به عدم توانایی پایگاه داده‌ها در تحمیل این نکته که مسیر یک گره درست ایجاد شده است و یا تضمین وجود گره‌ای در مسیری خاص، اشاره نمود. همچنین نگهداری رشته‌ی مسیر یک گره مبتنی بر کد برنامه است و اعتبارسنجی آن کاری هزینه‌بر است. این رشته اندازه‌ای محدود دارد و درخت‌هایی با عمق نامحدود را پشتیبانی نمی‌کند.

مجموعه‌های تودرتو :

مجموعه‌های تودرتو، اطلاعات را با هر گره‌ای که مربوط به مجموعه‌ای از نوه‌هایش است، به جای این که تنها مربوط به یک فرزند بلافصلش باشد، ذخیره می‌کنند.



این اطلاعات می‌توانند به وسیله‌ی هر گره‌ای که در درخت با دو شماره‌ی nsleft و nsright ذخیره شده، نمایش داده شوند:

```
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)
);
```

comment_id	nsleft	nsright	author	comment
1	1	14	Fran	What's the cause of this bug?
2	2	5	Ollie	I think it's a null pointer.
3	3	4	Fran	No, I checked for that.
4	6	13	Kukla	We need to check for invalid input.
5	7	8	Ollie	Yes, that's a bug.
6	9	12	Fran	Yes, please add a check.
7	10	11	Kukla	That fixed it.

شماره‌ی سمت چپ یک گره از تمام شماره‌های سمت چپ فرزندان آن گره کوچک‌تر و شماره‌ی سمت راست آن گره از تمام شماره‌های سمت راست آن گره بزرگ‌تر است. این شماره‌ها هیچ ارتباطی به comment_id مربوط به آن گره ندارند.

یک راه حل ساده برای تخصیص این شماره‌ها به گره‌ها این است که از سمت چپ یک گره آغاز می‌کنیم و اولین شماره را اختصاص می‌دهیم و به همین به گره‌ای سمت چپ فرزندان می‌آییم و شماره‌ها را به صورت افزایشی به سمت چپ آن‌ها نیز اختصاص می‌دهیم. سپس در ادامه به سمت راست آخرین نود رفته و از آن جا به سمت بالا می‌آییم و به همین ترتیب به صورت بازگشتی تخصیص شماره‌ها را ادامه می‌دهیم.

با اختصتص شماره‌ها به هر گره، می‌توان از آن‌ها برای یافتن نیاکان و فرزندان آن گره بهره جست. برای مثال برای بازیابی گره‌ی 4 و فرزندان (نوه‌های) آن باید دنبال گره‌هایی باشیم که شماره‌های آن گره‌ها بین nsleft و nsright گره‌ی شماره4 باشد:

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

همچنین می‌توان گره‌ی شماره‌ی 6 و نیاکان آن را با دنبال نمودن گره‌هایی به دست آورد که شماره‌های آن‌ها در محدوده‌ی شماره‌ی گره‌ی 6 باشد:

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

یک مزیت مهم روش مجموعه‌ای تودرتو، این است که هنگامی که یک گره را حذف می‌کنیم، نوه‌های آن به طور مستقیم به عنوان فرزندان پدر گره‌ی حذف شده تلقی می‌شوند.
برخی از پرس‌وجوهایی که در روش لیست مجاورت ساده بودند، مانند بازیابی فرزند یا پدر بلافصل، در روش مجموعه‌های تودرتو پیچیده‌تر می‌باشند. برای مثال برای یافتن پدر بلافصل گره‌ی شماره‌ی 6 باید چنین نوشت:

```
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;
```

دست‌کاری درخت، اضافه، حذف و جابجا نمودن گره‌ها در آن درروش مجموعه‌های تودرتو از مدل‌های دیگر پیچیده‌تر است.
هنگامی که یک گره‌ی جدید را اضافه می‌کنیم، باید تمام مقادیر چپ و راست بزرگ‌تر از مقدار سمت چپ گره‌ی جدید را مجددا محاسبه کنیم؛ که این شامل برادر سمت راست گره‌ی جدید، نیاکان آن و برادر سمت راست نیاکان آن می‌باشد. همچنین اگر گره‌ی جدید به عنوان گره‌ی غیربرگ اضافه شده باشد، شامل فرزندان آن هم می‌شود. برای مثال اگر بخواهیم گره‌ی جدیدی به گره‌ی 5 اضافه نماییم، باید چنین بنویسیم:

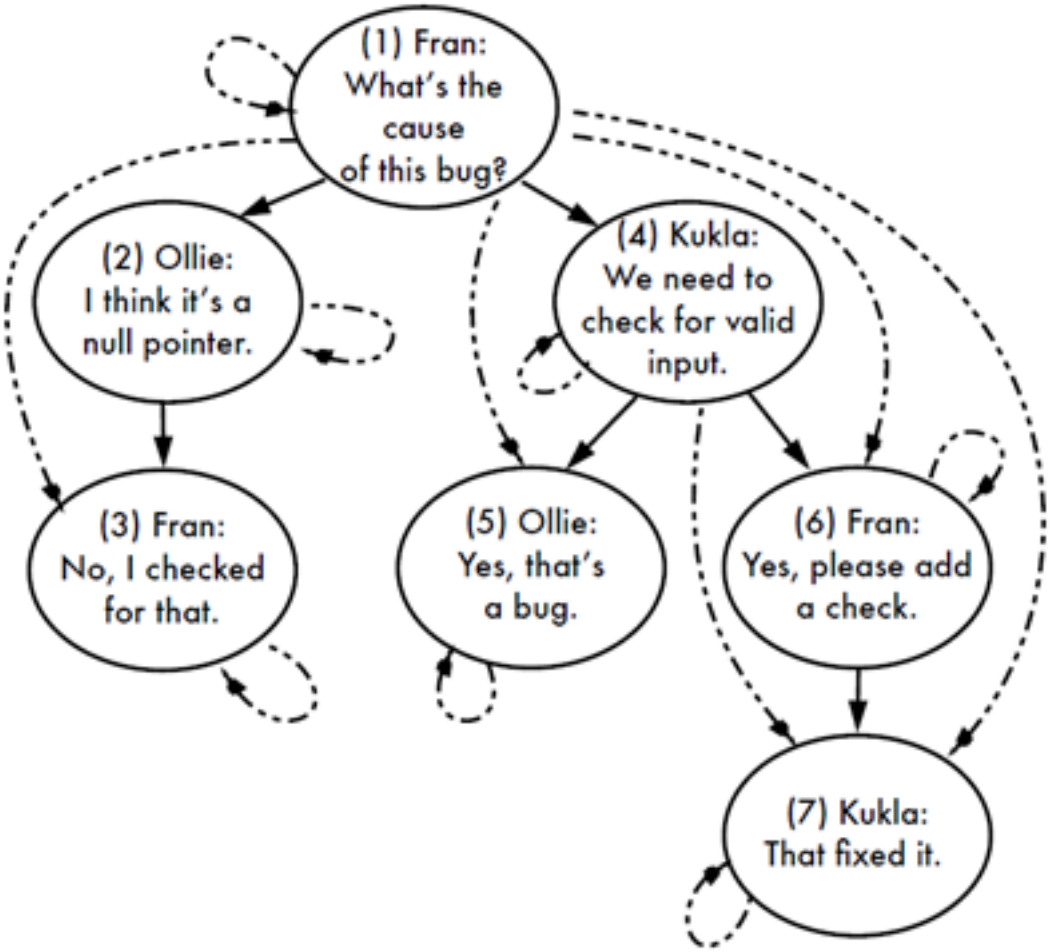
```
-- make space for NS values 8 and 9
UPDATE Comment
SET nsleft = CASE WHEN nsleft >= 8 THEN nsleft+2 ELSE nsleft END,
nsright = nsright+2
WHERE nsright >= 7;

-- create new child of comment #5, occupying NS values 8 and 9
INSERT INTO Comment (nsleft, nsright, author, comment)
VALUES (8, 9, 'Fran' , 'Me too!' );
```

تنها مزیت این روش نسبت به روش‌های قبلی ساده‌تر و سریع‌تر شدن ایجاد پرس‌وجوها برای پیدا کردن فرزندان یا پدران یک درخت است. اگر هدف استفاده از درخت شامل اضافه نمودن متعدد گره‌ها است، مجموعه‌های تودرتو انتخاب خوبی نیست.

Closure Table

راه حل closure table روشی دیگر برای ذخیره‌ی سلسه‌مراتبی است. این روش علاوه بر ارتباطات مستقیم پدر- فرزندی، تمام مسیرهای موجود در درخت را ذخیره می‌کند.



این روش علاوه بر داشتن یک جدول نظرها، یک جدول دیگر به نام TreePaths با دو ستون دارد که هر کدام از این ستون‌ها یک کلید خارجی به جدولComment هستند:

```
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)
);
```

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

به جای استفاده از جدول Comments برای ذخیره‌ی اطلاعات مربوط به یک درخت از جدول TreePath استفاده می‌کنیم. به ازای هر یک جفت گره در این درخت یک سطر در جدول ذخیره می‌شود که ارتباط پدر فرزندی را نمایش می‌دهد و الزاما نباید این دو پدر فرزند بلافصل باشد. همچنین یک سطر هم به ازای ارتباط هر گره با خودش به جدول اضافه می‌گردد.

پرس‌وجوهای بازیابی نیاکان و فرزندان (گره‌ها) از طریق جدول TreePaths ساده‌تر از روش مجموعه‌های تودرتو است. مثلا برای بازیابی فرزندان (نوه‌های) گره‌ی شماره‌ی 4، سطرهایی که نیاکان آن‌ها 4 است را به دست می‌آوریم:

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

برای به دست آوردن نیاکان گره‌ی شماره‌ی 6، سطرهایی از جول TreePaths را به دست می‌آوریم که فرزندان آن‌ها 6 باشد:

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

برای اضافه کردن گره‌ی جدید، برای مثال به عنوان فرزند گره‌ی شماره‌ی 5، ابتدا سطری که به خود آن گره برمی‌گردد را اضافه می‌کنیم، سپس یک کپی از سطوری که در جدول TreePaths، به عنوان فرزندان (نوه‌های) گره‌ی شماره5 هستند (که شامل سطری که به خود گره‌ی 5 به عنوان فرزند اشاره می‌کند) به جدول اضافه نموده و فیلد descendant آن را با شماره‌ی گره‌ی جدید جایگزین می‌کنیم:

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

در این جا می‌توان به اهمیت ارجاع یک گره به خودش به عنوان پدر (یا فرزند) پی برد.
برای حذف یک گره، مثلا گره‌ی شماره‌ی 7، تمام سطوری که فیلد descendant آن‌ها در جدول TreePaths برابر با 7 است حذف می‌کنیم:

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

برای حذف یک زیردرخت کامل، برای مثال گره‌ی شماره‌ی 4 و فرزندان (نوه‌های) آن، تمام سطوری که در جدول TreePaths دارای فیلد descendant با مقدار 4 هستند، حذف می‌کنیم. علاوه بر این باید نودهایی که به عنوان descendant به فیلد descendant گره‌ی 4، ارجاع داده می‌شوند نیز باید حذف گردد:

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

دقت کنید وقتی گره‌ای را حذف می‌کنیم، بدان معنی نیست که خود گره (نظر) را حذف می‌کنیم. البته این برای مثال نظر و پاسخ آن مقداری عجیب است ولی در مثال کارمندان در چارت سازمانی امری معمول است. هنگامی که ارتباطات یک کاربر را تغییر می‌دهیم، از حذف در جدول TreePaths استفاده می‌کنیم و این قضیه که ارتباطات کارمندان در جدول جداگانه‌ای ذخیره شده است به ما انعطاف‌پذیری بیشتری می‌دهد.

برای جابجایی یک زیردرخت از مکانی به مکان دیگری در درخت، سطرهایی که ancestor گره‌ی بالایی زیردرخت را برمی‌گردانند و فرزندان آن گره را حذف می‌کنیم. برای مثال برای جابجایی گره‌ی شماره‌ی 6 به عنوان فرزند گره‌ی شماره‌ی 4 و قرار دادن آن به عنوان فرزند گره‌ی شماره‌ی 3، این چنین عمل می‌کنیم. فقط باید حواسمان جمع باشد سطری که گره‌ی شماره‌ی 6 به خودش ارجاع داده است را حذف نکنیم:

```
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);
```

آن‌گاه این زیردرخت جدا شده را با اضافه کردن سطرهایی که با ancestor مکان جدید و descendant زیردرخت، منطبق هستند، به جدول اضافه می‌کنیم:

```
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;
```

روش Closure Table آسان‌تر از روش مجموعه‌های تودرتو است. هر دوی آن‌ها روش‌های سریع و آسانی برای ایجاد پرس‌وجو برای نیاکان و نوه‌ها دارند. ولی Closure Table برای نگهداری اطلاعات سلسله مراتب آسان‌تر است. در هر دو طراحی ایجاد پرس‌وجو در فرزندان و پدر بلافصل سرراست‌تر از روش‌ای لیست مجاورت و شمارش مسیر می‌باشد.
می‌توان عملکرد Closure Table را برای ایجاد پرس‌وجو روی فرزندان و پدر بلافصل را آسان‌تر نیز نمود. اگر فیلد path_length را به جدول TreePaths اضافه نماییم این کار انجام می‌شود. path_length گره‌ای که به خودش ارجاع می‌شود، صفر است. path_length فرزند بلافصل هر گره 1، path_length نوه‌ی آن 2 می‌باشد و به همین ترتیب path_lengthها را در هر سطر مقداردهی می‌کنیم. اکنون یا فتن فرزند گره‌ی شماره‌ی 4 آسان‌تر است:

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

از کدام طراحی استفاده نماییم؟

در این جا این سؤال مطرح است که ما باید از کدام طراحی استفاده نماییم. در پاسخ به این سؤال باید گفت که هر کدام از این روش‌ها نقاط قوت و ضعفی دارند که ما باید نسبت به عملیاتی که می‌خواهیم انجام دهیم از این طراحی‌ها استفاده کنیم. جدولی که در ادامه آمده است، مقایسه‌ای است میان میزان سهولت اجرای این طراحی‌ها در استفاده از پرس‌وجوهای متفاوت.

Design	Tables	Query Child	Query Tree	Insert	Delete	Referential Integration
Adjacency List	1	Easy	Hard	Easy	Easy	Yes
Recursive Query	1	Easy	Easy	Easy	Easy	Yes
Path Enumeration	1	Easy	Easy	Easy	Easy	No
Nested Sets	1	Hard	Easy	Hard	Hard	No
Closure Table	2	Easy	Easy	Easy	Easy	Yes

لازم به ذکر است در اینجا ستون سوم (Query Child) به معنای پرس‌وجوهایی است که با فرزندان کار می‌کند و ستون چهارم (Query Tree) به معنای پرس‌وجوهایی است که با کل درخت کار می‌کنند، می‌باشد.

نظرات خوانندگان

نویسنده: علیرضا صبوری
تاریخ: ۱۳۹۳/۰۵/۰۵ ۱۴:۴۷

فکر میکنم عموما پرس‌وجوی بازگشتی اگر ساپورت بشه توسط دیتابیس بهترین روش همان لیست مجاورت هستش که مدیریت درخت رو برامون ساده میکنه و دیتابیس کنترل بشتری رو هر نود ما داره. البته به غیر از مواردی خاص... ممنون از مطلب مفیدتون ولی سوالی که دارم اینه از نظر Performance مقایسه ای انجام شده که آیا استفاده از لیست بازگشتی چقدر از نظر سرعت در بازیابی اطلاعات با سایر روش‌ها تفاوت داره ؟ مبنعی اگر سراغ دارید ممنون میشم معرفی کنین.