Subqueries, CTEs, and Window Functions

Scalar ve korelasyonlu subquery'ler, WITH (CTE) zincirleme, ve window fonksiyonları (ROW_NUMBER, RANK, PARTITION BY) -- GROUP BY'nin aksine satırları çökertmeden onlar arasında hesaplama, bu projenin kendi gerçek topic tablosuyla. Advanced PostgreSQL kategorisinin 1.'i.

İleri 30 dk

Son güncelleme: 6 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/subqueries-ctes-and-window-functions

---

PostgreSQL Temelleri, günlük SQL araç kutusunu inşa etti -- `SELECT`, `JOIN`, `GROUP BY`. Burada başlayan İleri PostgreSQL, PostgreSQL'e özgü derinliğin yoğunlaştığı yer. Subquery'ler, CTE'ler, ve window fonksiyonları tek bir fikri paylaşır: bir satır kümesi üzerinde bir şey hesaplamak, `GROUP BY`nin yaptığı gibi onları daha az satıra ÇÖKERTMEDEN -- her biri, yalnızca o tek satırın ötesine bakması gereken bir soruyu cevaplarken bile her orijinal satırı bozulmadan tutar.

Subquery'ler: Bir Sorgu İçindeki Bir Sorgu, Tekrar

"Inserting, Updating, and Deleting Data" ve "JOINs" zaten subquery'leri -- başka bir ifadenin içine gömülü bir `SELECT` -- bir id'yi slug'a göre aramak için kullanmıştı. Bir subquery, bir karşılaştırmada kullanılan tek bir değerin yerine de geçebilir:

```
SELECT slug, estimated_minutes
FROM topic
WHERE estimated_minutes > (SELECT AVG(estimated_minutes) FROM topic);
```

Buradaki parantez içine alınmış `SELECT`, bir **scalar subquery**dir -- tam olarak bir satır ve bir kolon döndürmelidir, çünkü bir literal sayının yapacağı gibi `estimated_minutes`le `>` ile karşılaştırılıyor. Bu projenin kendi gerçek verisine karşı çalıştırıldığında, bu platform-geneli ortalamadan daha uzun süren her topic'i döndürür -- gerçekten kullanışlı, ve subquery olmadan hiçbir tek `WHERE` koşulunun ifade edemeyeceği bir şey, çünkü ortalamanın kendisi önce tüm tablonun taranmasına bağlıdır.

Korelasyonlu Subquery'ler

Yukarıdaki subquery, dış sorgunun satırlarından tamamen bağımsız olarak tek bir sayı hesaplar. Bir **korelasyonlu subquery** farklıdır -- dış sorgudan bir kolona referans verir, ve dış satır başına bir kez yeniden değerlendirilir:

```
SELECT t.slug, t.estimated_minutes
FROM topic t
WHERE t.estimated_minutes > (
    SELECT AVG(t2.estimated_minutes)
    FROM topic t2
    WHERE t2.category_id = t.category_id
);
```

Bu, yukarıdakinden gerçek, anlamlı ölçüde farklı bir soru: "platform-geneli ortalamadan daha uzun" değil, "kendi KATEGORİSİNİN ortalamasından daha uzun" -- `t2.category_id = t.category_id`, onu korelasyonlu yapan şeydir, dış sorgudaki bir satırın ait olduğu her kategori için iç ortalamayı ayrı ayrı yeniden hesaplar. Korelasyonlu subquery'ler güçlüdür ama büyük tablolarda yavaş olabilir, çünkü iç sorgu kavramsal olarak dış satır başına yeniden çalışır -- bu kategoride daha sonra "Indexes and Query Performance with EXPLAIN", tam olarak bu tür bir maliyet hakkında akıl yürütmeye geri döner.

WITH: Common Table Expression'lar (CTE'ler)

Bir `WITH` cümlesi, bir subquery'yi baştan adlandırır, ana sorgunun ona gerçek bir tablo gibi referans vermesine izin verir -- bir sorgu aynı ara sonuç hakkında birden fazla kez akıl yürütmesi gerektiği anda, ya da yalnızca adlandırılmış adımlara bölündüğünde daha iyi okunduğu için kullanışlı:

```
WITH category_counts AS (
    SELECT category_id, COUNT(*) AS topic_count
    FROM topic
    GROUP BY category_id
)
SELECT cat.name, cc.topic_count
FROM category cat
JOIN category_counts cc ON cc.category_id = cat.id
ORDER BY cc.topic_count DESC;
```

`category_counts`, bu projenin şemasında hiçbir yerde gerçek bir tablo değil -- yalnızca bu tek sorgu süresince var olur, `WITH` cümlesi tarafından bir kez hesaplanır ve sonra takip eden `SELECT`te herhangi bir başka tablo gibi sorgulanır. Bu, "Aggregation and GROUP BY"nin zaten inşa ettiği aynı aggregation (kategori başına topic), şimdi bir ad verilmiş ve tek seferlik hesaplanmak yerine aynı ifadede `category`ye karşı join'lenmiş.

Gerçek Bir Çok-Adımlı CTE Örneği

CTE'ler zincirlenebilir, her biri bir öncekinin üzerine inşa eder -- çok-adımlı bir hesaplamayı, tek derin iç içe geçmiş bir sorgu yerine okunabilir aşamalar olarak ifade etmek için kullanışlı:

```
WITH category_counts AS (
    SELECT category_id, COUNT(*) AS topic_count
    FROM topic
    GROUP BY category_id
),
above_average AS (
    SELECT category_id, topic_count
    FROM category_counts
    WHERE topic_count > (SELECT AVG(topic_count) FROM category_counts)
)
SELECT cat.name, aa.topic_count
FROM category cat
JOIN above_average aa ON aa.category_id = cat.id;
```

`above_average`, hemen öncesinde tanımlanan CTE olan `category_counts`a referans verir -- her aşama İngilizce bir cümle gibi okunur ("kategori başına topic'leri say," sonra "yalnızca ortalamanın üzerindekileri tut"), oysa aggregation'ın başka bir subquery'nin içindeki bir subquery'nin içine gömüldüğü eşdeğer tek sorgu, tek geçişte okumak çok daha zor olurdu.

Window Fonksiyonları: Satırları Çökertmeden Onlar Arasında Hesaplamak

Zaten kapsanan `GROUP BY`, her zaman birçok satırı daha aza indirger -- grup başına bir satır. Bir **window fonksiyonu**, her orijinal satırı sonuçta tutarken, satır BAŞINA aggregate-benzeri bir değer hesaplar:

```
SELECT slug, category_id, estimated_minutes,
       AVG(estimated_minutes) OVER (PARTITION BY category_id) AS category_avg
FROM topic;
```

Her tek `topic` satırı çıktıda hâlâ görünür -- hiçbir şey çökertilmez -- ama şimdi her biri kendi kategorisinin ortalamasını da yanında taşır. `OVER (...)`, buradaki `AVG(...)`i düzenli bir aggregate yerine bir window fonksiyonu olarak işaretleyen şeydir; onsuz, bu bir `GROUP BY` gerektirirdi ve içinde listelenmeyen her kolonu kaybederdi.

ROW_NUMBER(), RANK(), ve PARTITION BY

Window fonksiyonu olarak kullanılan aggregate'lerin ötesinde, PostgreSQL'in yalnızca window fonksiyonu olarak anlam ifade eden fonksiyonları var -- `ROW_NUMBER()` ve `RANK()`, en yaygın olanlar arasında:

```
SELECT slug, category_id, estimated_minutes,
       ROW_NUMBER() OVER (PARTITION BY category_id ORDER BY estimated_minutes DESC) AS
rank_in_category
FROM topic;
```

`PARTITION BY category_id`, satırları bağımsız gruplara böler, tam olarak `GROUP BY`nin aggregation için yapacağı gibi -- ama her grubu tek bir satıra çökertmek yerine, `ROW_NUMBER()`, `ORDER BY estimated_minutes DESC`in kurduğu sırayla, her yeni `category_id`de `1` den yeniden başlayarak satırları her PARTITION İÇİNDE numaralandırır. `RANK()`, neredeyse özdeş davranır, ancak eşit değerler (birebir aynı `estimated_minutes`li iki topic), aynı rank'ı alır, ve ondan sonraki sıradaki rank buna göre ileri atlar (`1, 1, 2` değil, `1, 1, 3` bir rank) -- `ROW_NUMBER()` asla eşitlenmez, her zaman katı bir `1, 2, 3` üretir.

Gerçek Bir Window Fonksiyonu Örneği

Bu projenin kendi `postgresql-foundations` kategorisine karşı çalıştırıldığında, yukarıdaki `ROW_NUMBER()` sorgusu gerçekten bilinmeye değer bir şeyi yüzeye çıkarır: o kategoride hangi topic'in en uzun sürdüğü, hangisinin en kısa sürdüğü, ve arasındaki her şey -- tek başına `GROUP BY`nin üretmesinin hiçbir yolu olmayan bir sıralama, çünkü yalnızca kategori başına bir özet satır raporlayabilir, asla onun içinde satır-başına bir konum değil. Window fonksiyonlarının kattığı somut fark bu: bir aggregate "grubun bir bütün olarak neyin doğru olduğu"nu cevaplar; bir window fonksiyonu, satırın kendisini hiç gözden kaçırmadan, "*bu* satırın kendi grubu içinde nerede durduğu"nu cevaplar.

Yaygın Yanlış Anlamalar

"Bir CTE her zaman eşdeğer subquery'den daha hızlıdır." Kesinlikle değil -- bir `WITH` cümlesi öncelikle bir okunabilirlik ve yeniden kullanılabilirlik aracıdır; PostgreSQL birçok durumda bir CTE'yi ve eşdeğer iç içe geçmiş bir subquery'yi benzer şekilde optimize etmekte özgürdür, ve performans karakteristikleri gerçekten sorguya bağlıdır. **"Window fonksiyonları ve `GROUP BY` aynı şeyi yapar, yalnızca farklı sözdizimiyle."** Yapmazlar -- `GROUP BY` satır sayısını azaltır; bir window fonksiyonu asla azaltmaz, ki bu ikisinin, ikisi de `AVG` gibi bir aggregate'ten başlasa bile, farklı problemleri çözmesinin tam nedenidir. **"`RANK()` ve `ROW_NUMBER()` birbirinin yerine geçebilir."** Yalnızca `ORDER BY` kolonunda hiç eşitlik yokken -- iki satır bir değeri paylaştığı anda ayrışırlar, ve yanlışını seçmek sessizce ince bir şekilde yanlış bir sıralama üretir.

Best Practices

- Koşul gerçekten dış satıra referans vermeye ihtiyaç duyduğunda (bu kategorinin kendi ortalamasının üzerinde gibi) korelasyonlu bir subquery'ye başvur -- değer dış satıra hiç bağlı olmadığında, tek seferde hesaplanan korelasyonsuz bir scalar subquery, daha basit ve normalde daha ucuzdur.
- Bu dersin "ortalamanın üzerindeki kategoriler" örneğinin yaptığı gibi, çok-adımlı bir hesaplamayı birkaç seviye derin subquery'ler iç içe geçirmek yerine adlandırılmış CTE'lere böl -- SQL seviyesinde hiçbir maliyeti yoktur ve problemin İngilizce açıklamasına çok daha fazla benzer okunur.

- Katı, eşitliksiz bir sıralama gerektiğinde (her satıra tam olarak bir rank atamak gibi) `ROW_NUMBER()` i, ve özellikle eşit değerlerin bir konumu paylaşması gerektiğinde `RANK()` i seç -- ikisi arasında seçim yapmak eşitlikler hakkında bir karardır, stilistik bir karar değil.
- "Satır başına" gerçekten "kendi grubu içinde satır başına" (kategori başına, zorluk başına) anlamına gelmesi gerektiği anda bir window fonksiyonuna `PARTITION BY` ekle -- onu dışarıda bırakmak, tüm sonucu tek bir partition olarak ele alır, hesaplanan değerin ne anlama geldiğini sessizce değiştirir.

Yaygın Hatalar

- Gerçekte birden fazla satır döndürebilen bir scalar subquery yazmak, ve veri ikinci bir eşleşen satır ürettiğinde ancak gerçek bir runtime hatasıyla ("more than one row returned by a subquery used as an expression") karşılaşmak.
- Daha sonraki bir CTE'ye daha erken bir CTE'den referans vermek -- CTE'ler (`RECURSIVE` olmadan) yalnızca aynı `WITH` cümlesinde kendilerinden önce tanımlananlara, sırayla, sorgunun kendisini okumakla aynı yukarıdan-aşağıya bağımlılık yönünde referans verebilir.
- "Kategori başına" ya da "zorluk başına" hesaplaması amaçlanan bir window fonksiyonunda `PARTITION BY` yi unutmak, ve bunun yerine tüm tablo genelinde hesaplanmış tek bir değer almak -- sorgu hatasız çalışır, sessizce amaçlanandan daha geniş bir soruyu cevaplar.
- Bir window fonksiyonunun, `GROUP BY` li bir aggregate'in yapacağı gibi satır sayısını azaltacağını varsaymak -- asla azaltmaz; `OVER (...)` eklendikten sonra daha az çıktı satırı bekleyen bir sorgu, her orijinal satırın hâlâ mevcut olduğunu görünce şaşırır.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Bir scalar subquery tam olarak bir değer döndürür ve bir literal'in kullanılabileceği her yerde kullanılabilir; korelasyonlu bir subquery dış sorgunun satırına referans verir ve dış satır başına yeniden değerlendirilir -- bu projenin kendi "kendi kategorisinin ortalamasından daha uzun süren topic'ler"i gerçek bir korelasyonlu örnek.
- `WITH <ad> AS (...)` , bir CTE tanımlar, adlandırılmış, sorgu-başına-tek-kullanımlık bir subquery, takip eden ifadede bir tablo gibi referans verilebilir -- CTE'ler zincirlenebilir, her biri bir öncekinin üzerine inşa eder.
- Window fonksiyonları (`OVER (...)`), `GROUP BY` nin aksine sonucu çökertmeden satır başına aggregate-benzeri ya da sıralama değeri hesaplar -- her orijinal satır hayatta kalır.
- `PARTITION BY` , bir window fonksiyonu için satırları bağımsız gruplara böler, `GROUP BY` nin gruplamasının window-fonksiyonu eşdeğeri; `OVER (...)` içindeki `ORDER BY` , `ROW_NUMBER()` / `RANK()` in sayacağı sırayı kurar.

- `ROW_NUMBER()`, her zaman katı, eşitliksiz bir dizi üretir; `RANK()`, eşit satırlara aynı rank'ı verir ve sonrasında ileri atlar -- ikisi arasındaki seçim, eşitliklerin nasıl ele alınması gerektiğiyle ilgilidir, stilistik bir tercih değil.

Cheat Sheet

```
-- Scalar subquery
SELECT a FROM t WHERE a > (SELECT AVG(a) FROM t);

-- Korelasyonlu subquery
SELECT a FROM t t1 WHERE a > (SELECT AVG(a) FROM t t2 WHERE t2.g = t1.g);

-- CTE
WITH x AS (SELECT g, COUNT(*) AS c FROM t GROUP BY g)
SELECT * FROM x WHERE c > 1;

-- Window fonksiyonları
SELECT a, AVG(a) OVER (PARTITION BY g) FROM t;
SELECT a, ROW_NUMBER() OVER (PARTITION BY g ORDER BY a DESC) FROM t;
SELECT a, RANK() OVER (PARTITION BY g ORDER BY a DESC) FROM t;
```

Terimler Sözlüğü

- **Scalar subquery**: tam olarak bir satır ve bir kolon döndürmesi gereken, tek bir değerin beklendiği her yerde kullanılabilen bir subquery.
- **Korelasyonlu subquery**: dış sorgudan bir kolona referans veren, dış satır başına bir kez yeniden değerlendirilen bir subquery.
- **CTE (Common Table Expression)**: `WITH` ile tanıtılan, ifadenin geri kalanında bir tablo gibi referans verilebilen adlandırılmış bir subquery.
- **Window fonksiyonu**: `GROUP BY` nin yaptığı gibi çıktı satır sayısını azaltmadan, tanımlanmış bir ilişkili satır kümesi ("window"u) üzerinde satır başına hesaplanan bir fonksiyon.
- **PARTITION BY**: bir window fonksiyonu için satırları bağımsız gruplara böler, aggregation için `GROUP BY` ye benzer.