Aggregation and GROUP BY

COUNT/SUM/AVG/MIN/MAX, GROUP BY, ve HAVING -- Spring Data JPA'nın Pageable/Sort /projection kelime dağarcığında hiç karşılığı olmayan, tamamen yeni materyal -- bu projenin kendi gerçek "kategori başına topic" LEFT JOIN + GROUP BY sorgusuyla. PostgreSQL Foundations kategorisinin 10.'u ve SONUNCUSU.

Orta 20 dk

Son güncelleme: 25 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/aggregation-and-group-by

---

Şimdiye kadarki her sorgu, altta yatan verinin her satırı için bir satır döndürdü -- bir `WHERE`, hangi satırların göründüğünü daraltabilir, ama giren satır sayısı çıkan satır sayısına eşittir. Bu ders bunun değiştiği ilk yer: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`, ve `GROUP BY`, birçok satırı tek bir özet satıra, ya da grup başına bir özet satıra çöker -- Spring Data JPA kursunda zaten kapsanan `Pageable` / `Sort` /projection kelime dağarcığında bunun gibi hiçbir şey yok; bu gerçekten yeni materyal.

Aggregate Fonksiyonlar: COUNT, SUM, AVG, MIN, MAX

Bir aggregate fonksiyon, birçok satır alır ve hepsi üzerinde hesaplanmış tek bir değer döndürür:

```
SELECT COUNT(*) FROM topic;
SELECT AVG(estimated_minutes) FROM topic;
SELECT MIN(estimated_minutes), MAX(estimated_minutes) FROM topic;
```

`COUNT(*)`, herhangi bir kolonun değerinden bağımsız olarak satırları sayar; `COUNT(kolon)`, yalnızca o kolonun `NULL` olmadığı satırları sayar -- bu projenin kendi null-olabilir `estimated_minutes` kolonunda gerçek bir ayrım, herhangi bir satırda o ayarlanmamışsa `COUNT(*)` ve `COUNT(estimated_minutes)` gerçekten farklı olabilirdi. `AVG` / `SUM` / `MIN` / `MAX`, hepsi tek bir sayısal (ya da `MIN` / `MAX` için, sıralanabilir) kolon üzerinde çalışır, tek bir `NULL`ın tüm hesaplamayı zehirlemesine izin vermek yerine `NULL`ları otomatik olarak yok sayar.

GROUP BY Olmadan Aggregate: Tüm Tablo İçin Tek Satır

Yukarıdaki her örnek, tablonun kaç satırı olursa olsun, tam olarak bir satır döndürür -- `GROUP BY`siz bir aggregate fonksiyon, `FROM` / `WHERE` cümlelerinin tüm sonucunu tek bir grup olarak ele alır. Bir `WHERE` eklemek, herhangi bir başka sorgudaki gibi, aggregate'i besleyen satırları daraltır:

```
SELECT COUNT(*) FROM topic t
JOIN category cat ON t.category_id = cat.id
WHERE cat.slug = 'postgresql-foundations';
```

hâlâ geriye tek bir satır -- özellikle `postgresql-foundations`ın kendi topic'lerinin sayısı, tüm `topic` tablosu değil (sabit sayısal bir id yerine `slug` üzerinde bir join kullanarak, "Inserting, Updating, and Deleting Data"nın `INSERT ... SELECT` deseninin id'leri sabit kodlamamak için zaten kurduğu aynı akıl yürütme).

GROUP BY: Grup Başına Bir Satır

`GROUP BY`, bir aggregate'i "toplamda bir satır"dan, gruplandığı şeyin "her ayrı değeri için bir satır"a değiştirir:

```
SELECT difficulty, COUNT(*)
FROM topic
GROUP BY difficulty;
```

Bu projenin kendi gerçek `topic` tablosuna karşı çalıştırıldığında, bu, gerçekte mevcut olan her `Difficulty` değeri için bir satır döndürür -- `BEGINNER`, `INTERMEDIATE`, `ADVANCED` -- her biri şu anda o zorlukta kaç topic olduğuyla eşleştirilerek. `SELECT` listesindeki her kolon, ya bir aggregate fonksiyon ya da `GROUP BY`de adlandırılan kolonlardan biri olmak zorundadır -- `SELECT slug, difficulty, COUNT(*) ... GROUP BY difficulty` reddedilir, çünkü belirli bir `difficulty` grubu için PostgreSQL'in raporlayacak tek bir `slug`ı yoktur; birden fazla olabilir.

Gerçek Bir Örnek: Kategori Başına Kaç Topic

```
SELECT cat.name AS category_name, COUNT(t.id) AS topic_count
FROM category cat
LEFT JOIN topic t ON t.category_id = cat.id
GROUP BY cat.name;
```

Bu, "JOINs"i (bilinçli olarak seçilen bir `LEFT JOIN`, böylece sıfır topic'i olan bir kategori bile, bir `INNER JOIN`ın onu sessizce düşüreceği şekilde değil, `0` sayımıyla görünür) tek bir sorguda `GROUP BY`yle birleştirir -- kategoriye göre gruplayarak, her birine kaç `topic` satırının join'lendiğini sayarak. Bu projenin kendi gerçek verisine karşı çalıştırıldığında, `PostgreSQL Foundations`, bu dersin kendisi dahil, şu anda `10` sayımı gösterir. Burada özellikle `COUNT(*)` yerine `COUNT(t.id)` önemlidir: `LEFT JOIN`la, eşleşen topic'i olmayan bir kategori hâlâ, her `t.*` kolonunun `NULL` olduğu bir çıktı satırı üretir -- `COUNT(*)`, o satırı `1` olarak sayar, oysa `COUNT(t.id)`, onu doğru şekilde `0` olarak sayar, çünkü `t.id`nin kendisi onun için `NULL`dır.

Tek Bir Sorguda Birden Fazla Aggregate

Herhangi sayıda aggregate fonksiyon birlikte görünebilir, her biri grup başına ayrı ayrı hesaplanır:

```
SELECT difficulty, COUNT(*) AS topic_count, AVG(estimated_minutes) AS avg_minutes
FROM topic
GROUP BY difficulty;
```

Her `difficulty` için bir satır, o grup içinde bağımsız olarak hesaplanan hem bir sayım hem bir ortalama ile -- bu projenin gerçek `topic` tablosunun tamamına karşı (yalnızca bu kurs değil, her kursa yayılan) çalıştırıldığında, `ADVANCED` topic'lerin (Spring Data JPA kursundaki "The Persistence Context and Locking" gibi) `BEGINNER` olanlardan daha yüksek bir `avg_minutes` taşıyıp taşımadığını kontrol etmenin gerçek bir yolu -- tek satırlı hiçbir sorgunun cevaplayamayacağı bir soru.

HAVING: Satırları Değil Grupları Filtrelemek

`WHERE`, gruplama gerçekleşmeden *önce* tek tek satırları filtreler; `HAVING`, aggregation'dan *sonra*, aggregate'in kendi sonucuna göre tüm grupları filtreler:

```
SELECT category_id, COUNT(*) AS topic_count
FROM topic
GROUP BY category_id
HAVING COUNT(*) >= 9;
```

Bu yalnızca `9` ya da daha fazla topic'i olan kategorileri tutar -- `WHERE`de ifade etmek anlamsız bir koşul, çünkü `COUNT(*)`, filtrelenen herhangi bir tek satır için henüz mevcut değildir; yalnızca gruplama zaten gerçekleştikten sonra var olur. `HAVING`, tam sorguya bağlı olarak bir aggregate'e doğrudan (yukarıdaki gibi) ya da `SELECT` listesinde tanımlanan bir alias'a referans verebilir -- ama `SELECT`in yapamadığı aynı nedenle, asla ham, aggregate-olmayan, gruplanmamış bir kolona değil.

WHERE vs. HAVING: Her Biri Ne Zaman Çalışır

İki cümle, aynı sorgunun gerçekten farklı aşamalarında çalışır, ve hangisinin hangisini yaptığını karıştırmak, en yaygın `GROUP BY` hatalarından biridir:

```
SELECT category_id, COUNT(*) AS topic_count
FROM topic
WHERE difficulty = 'ADVANCED'
GROUP BY category_id
HAVING COUNT(*) >= 2;
```

`WHERE difficulty = 'ADVANCED'` önce çalışır, herhangi bir gruplama gerçekleşmeden önce `ADVANCED`-olmayan satırları tamamen atar; `GROUP BY category_id` sonra hayatta kalan satırları gruplar; `HAVING COUNT(*) >= 2` son çalışır, yalnızca sayımı (zaten filtrelenmiş satırların) eşiği karşılayan sonuç gruplarını tutar. `WHERE`, bir aggregate'in gördüğü satırları daraltır; `HAVING`, hangi hesaplanmış grupların nihai sonuca gireceğini daraltır -- "SELECT and Filtering"in `WHERE` kapsamının hiç çizmesi gerekmediği bir ayrım, çünkü o dersin hiçbir gruplama aşaması yoktu.

Yaygın Yanlış Anlamalar

"`COUNT(*)` ve `COUNT(kolon)` her zaman aynı sayıyı döndürür." Yalnızca o kolonun hiç `NULL`ı olmadığında -- bu projenin kendi null-olabilir `estimated_minutes`i, gerçekten farklılaşabilecekleri gerçek bir kolon. **"`WHERE`, `COUNT(*)` gibi bir aggregate üzerinde filtreleyebilir."** Filtreleyemez -- `WHERE`, aggregation var olmadan önce çalışır, ki bu tam olarak `HAVING`in `WHERE`e eklenmiş ekstra bir koşul yerine ayrı bir cümle olmasının nedeni. **"`GROUP BY`nin yanında seçilen her kolonun `GROUP BY` cümlesinde listelenmesi gerekir, yoksa sorguda bir hata var demektir."** Bu bir hata değil -- PostgreSQL'in aktif olarak zorunlu kıldığı bir kural: aggregate-olmayan, gruplanmamış bir kolonun grup başına basitçe tek, iyi tanımlanmış bir değeri yoktur, bu yüzden PostgreSQL keyfi olarak birini seçmek yerine sorguyu doğrudan reddeder.

Best Practices

- Sıfır-sayımlı gruplar önemli olduğunda, özellikle bir `LEFT JOIN`dan sonra `COUNT(*)` yerine `COUNT(t.id)`i seç -- bu projenin kendi "kategori başına topic" sorgusu, boş bir kategori için `1` yerine doğru şekilde `0` raporlamak için tam olarak bu ayrıma dayanır.
- `HAVING`e yalnızca koşul gerçekten bir aggregate'in sonucuna (bir sayım, bir toplam) bağlı olduğunda başvur -- düz bir kolon üzerindeki bir koşul `WHERE`e ait, ki bu daha erken çalışır ve PostgreSQL'in herhangi bir gruplama işi yapmadan önce eşleşmeyen satırları atmasına izin verir.
- Yalnızca PostgreSQL'in kuralını karşılamak için değil, her aggregate-olmayan `SELECT` kolonunu bilinçli olarak `GROUP BY`de listele -- her biri "bir grup"un gerçekte ne anlama geldiğini değiştirir, potansiyel olarak amaçlanandan çok daha fazla grup üretir.
- Aggregate başına bir sorgu çalıştırmak yerine (`COUNT` + `AVG` örneğindeki gibi) birden fazla aggregate'i tek bir sorguda birlikte hesapla -- her biri tek bir geçişte aynı gruplanmış satırlardan hesaplanır.

Yaygın Hatalar

- Bir aggregate koşulunu `HAVING` yerine `WHERE`e yazmak (`WHERE COUNT(*) >= 5`) ve yanlış bir cevap değil gerçek bir SQL hatası almak -- fark edildiğinde, tam olarak hangi cümlenin gerekli olduğunun kullanışlı bir sinyali.

- Aggregate-olmayan, gruplanmamış bir kolonu seçmek ve PostgreSQL'in reddiyle kafa karışıklığı yaşamak, sorgunun her grup içinde o kolon için hangi satırın değerinin raporlanacağı konusunda gerçekten belirsiz olduğunu fark etmek yerine.
- Gerçek niyet "kaç eşleşen satır" olduğunda bir `LEFT JOIN` dan sonra `COUNT(*)` kullanmak, ve eşleşmesi olmayan gruplar için `0` yerine `1` almak, çünkü `COUNT(*)`, bir `LEFT JOIN` ın eşleşmeyen bir sol-taraf satırı için ürettiği tek, tamamen-`NULL` satırı sayar.
- `GROUP BY` ve `DISTINCT` in aynı işi yaptığını varsaymak çünkü ikisi de yinelenen-görünen satırları çökertebilir -- `DISTINCT`, yalnızca bir sonuç kümesinden tam satır yinelemelerini kaldırır; `GROUP BY` nin gerçekten yapabildiği şekilde grup başına bir sayım, toplam, ya da ortalama hesaplayamaz.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Aggregate fonksiyonlar (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`), birçok satır üzerinde tek bir değer hesaplar; `GROUP BY` olmadan, tüm filtrelenmiş sonuç tek bir grup olarak ele alınır.
- `GROUP BY <kolon>`, o kolonun her ayrı değeri için bir satır üretir; her aggregate-olmayan `SELECT` kolonu `GROUP BY` de görünmelidir.
- Bu projenin kendi gerçek "kategori başına topic" sorgusu, boş kategorilerin doğru şekilde `0` göstermesi için bir `LEFT JOIN` ı `GROUP BY` ve `COUNT(t.id)` (`COUNT(*)` değil) ile birleştirir.
- `HAVING`, aggregate'in kendi sonucuna dayanarak aggregation'dan sonra grupları filtreler; `WHERE`, gruplama gerçekleşmeden önce tek tek satırları filtreler -- ikisi aynı sorgunun gerçekten farklı aşamalarında çalışır.
- `GROUP BY` nin, bu müfredatta zaten kapsanan Spring Data JPA'nın `Pageable` / `Sort` /projection kelime dağarcığında hiçbir yerde eşdeğeri yok -- bu ders tamamen yeni materyal, zaten öğretilmiş bir şeyin ham-SQL yeniden ifadesi değil.

Cheat Sheet

```
SELECT COUNT(*) FROM t;
SELECT COUNT(col) FROM t;            -- NULL'ları yok sayar
SELECT AVG(col), MIN(col), MAX(col) FROM t;

SELECT g, COUNT(*) FROM t GROUP BY g;

SELECT g, COUNT(*) FROM t
GROUP BY g
HAVING COUNT(*) >= 5;

SELECT g, COUNT(*) FROM t
WHERE col = 'x'
GROUP BY g
HAVING COUNT(*) >= 5;
```

Terimler Sözlüğü

- **Aggregate fonksiyon**: birçok satırdan tek bir değer hesaplayan bir fonksiyon (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`).
- **GROUP BY**: adlandırdığı kolonların her ayrı değeri (ya da değer kombinasyonu) için bir sonuç satırı üreten bir cümle.
- **HAVING**: bir aggregate'in sonucuna dayanarak aggregation'dan sonra grupları filtreleyen bir cümle -- gruplanmış bir sorgunun `WHERE`i.
- **Grup**: `GROUP BY` kolonlarında aynı değer(ler)i paylaşan, tek bir çıktı satırına çökertilen satır kümesi.