Indexes and Query Performance with EXPLAIN

EXPLAIN/EXPLAIN ANALYZE, Seq Scan vs. Index Scan, bu projenin kendi gerçek idx_topic_category B-tree index'iyle CREATE INDEX, partial/expression index'ler, ve OFFSET'in maliyetine keyset pagination'la geri dönüş. Advanced PostgreSQL kategorisinin 3.'ü.

İleri 30 dk

Son güncelleme: 25 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/indexes-and-query-performance-with-explain

---

"Sorting, Limiting, and Pagination", `OFFSET` in maliyetinin offset'le büyüdüğünden bahsetmişti, ve "Constraints and Keys", `PRIMARY KEY` in otomatik olarak bir index inşa ettiğinden -- ikisi de buraya ertelenmişti. Bu ders, index'lerin ve sorgu performansının nihayet gerçekten öğretildiği yer -- soyut bir DBA bilgisi olarak değil, bu projenin kendi migration'larının onları gerçekten kullandığı şekilde: sorgu, `EXPLAIN`, planı oku, tarama türünü anla, `EXPLAIN ANALYZE`.

EXPLAIN: Sorgu Planını Görmek

`EXPLAIN`, PostgreSQL'in bir sorguyu gerçekten çalıştırmadan nasıl çalıştırmayı amaçladığını gösterir:

```
EXPLAIN SELECT * FROM topic WHERE category_id = 1;
```

```
Seq Scan on topic  (cost=0.00..1.14 rows=10 width=44)
  Filter: (category_id = 1)
```

`Seq Scan` (sequential scan), PostgreSQL'in `topic` in her satırını, baştan sona, her birini `category_id = 1` e karşı kontrol ederek okuduğu anlamına gelir -- kısayol yok. `cost=0.00..1.14`, PostgreSQL'in kendi içsel maliyet tahminidir (saniye değil -- çoğunlukla disk I/O'sunu ve CPU işini yansıtan soyut bir birim), `rows=10`, kaç satırın eşleşeceğine dair tahminidir, ve `width=44`, ortalama satır boyutunu byte cinsinden tahmin eder. Bu, tam olarak "PostgreSQL ve İlişkisel Model"in sorgu-veritabanına-hikayesinin gerçekten bakacak bir aracı hiç olmayan şeyi -- `EXPLAIN` o araç.

Seq Scan vs. Index Scan

Bir `Seq Scan`, doğası gereği kötü değildir -- küçük bir tabloda, bugün bu projenin kendi tablolarının neredeyse hepsi gibi, her satırı taramak çoğu zaman gerçekten en ucuz seçenektir, bir index'e danışmanın ek yükünden bile daha ucuz. Bir `Index Scan` alternatiftir: her satırı okumak yerine, PostgreSQL doğrudan eşleşen satırlara atlamak için bir index kullanır, her yazımda o index'i sürdürme maliyeti karşılığında. PostgreSQL'in belirli bir sorgu için birini ya da diğerini seçmesi, tablo boyutuna, koşulun ne kadar seçici olduğuna, ve kullanılabilir bir index olup olmadığına dayanan kendi maliyet tahminine bağlıdır -- ki bu tam olarak bu projenin kendi gerçek tablolarının, şu anki küçük boyutlarında, bu ders aşağıda index'leyecek kolonlarda bile bir `Seq Scan` göstereceğinin nedenidir, ve bu PostgreSQL'in verdiği doğru karardır, eksik bir şeyin işareti değil.

Bir B-Tree Index Oluşturmak, ve Planın Değiştiğini Görmek

`CREATE INDEX`, bu projenin kendi `V1__init_schema.sql`'inin zaten dört kez kullandığı aynı ifadedir:

```
CREATE INDEX idx_topic_category ON topic (category_id);
```

Bu gerçek index, tam olarak `topic.category_id`nin ("Constraints and Keys"in zaten bir foreign key'in neden `REFERENCES`e ihtiyaç duyduğunu kapsadığı) sürekli filtrelenen ve join'lenen bir foreign key olduğu için var -- her `WHERE category_id = ...`, ve "JOINs"teki her `JOIN category ... ON t.category_id = cat.id`, ondan yararlanır. `USING <yöntem>` belirtilmeden, `CREATE INDEX` varsayılan olarak bir **B-tree**dir -- değerleri sıralı düzende tutan dengeli bir ağaç yapısı, PostgreSQL'in eşleşen bir satırı (ya da bir satır aralığını) her satırı doğrusal olarak taramak yerine kabaca logaritmik zamanda bulmasına izin verir; eşitlik ve aralık koşulları (`=`, `<`, `>`, `BETWEEN`) için doğru varsayılandır, ve pratikte açık ara en yaygın index türüdür, bu projenin kendi şemasının tanımladığı her index dahil. Bunun önemli olacağı kadar büyük bir tabloda, bu index var olduktan sonra `EXPLAIN`i tekrar çalıştırmak, `Seq Scan` yerine `Index Scan using idx_topic_category` gösterirdi -- bu projenin kendi tabloları bugün PostgreSQL'in index'i henüz kullanmaya değer bulması için basitçe çok küçük, ki bu, varsaymak yerine `EXPLAIN`in çıktısından okunabilecek gerçek, kullanışlı bir şeyin ta kendisi.

EXPLAIN ANALYZE: Gerçekte Ne Oldu

Tek başına `EXPLAIN` yalnızca tahmin eder; `EXPLAIN ANALYZE`, sorguyu gerçekten çalıştırır ve gerçekte ne olduğunu raporlar:

```
EXPLAIN ANALYZE SELECT * FROM topic WHERE category_id = 1;
```

```
Seq Scan on topic  (cost=0.00..1.14 rows=10 width=44)
                   (actual time=0.012..0.018 rows=10 loops=1)
  Filter: (category_id = 1)
```

İkinci satır yeni -- `actual time`, gerçek geçen milisaniyelerdir, buradaki `rows=10` gerçek döndürülen sayımdır (bir tahmin değil), ve `loops=1`, bu adımın kaç kez çalıştığını sayar (belirli join stratejilerinin iç tarafı gibi bir şey için birden fazla). Düz `EXPLAIN`den tahmini `rows`u `EXPLAIN ANALYZE`den gerçek `rows`la karşılaştırmak, bu araç çiftinin sunduğu en kullanışlı şeylerden biridir -- aralarındaki büyük bir boşluk, PostgreSQL'in tablo hakkındaki kendi istatistiklerinin eski ya da yanıltıcı olduğu anlamına gelir, ki bu kendi başına onun aksi hâlde seçeceğinden daha kötü bir plan seçmesine neden olabilir.

Partial Index'ler

Bir index her satırı kapsamak zorunda değildir -- bir **partial index**, index tanımının kendisine bir `WHERE` cümlesi ekler, yalnızca eşleşen satırları index'ler:

```
CREATE INDEX idx_topic_translation_published_en
    ON topic_translation (topic_id)
    WHERE language = 'en' AND published = true;
```

Bu, "JOINs"teki bu projenin kendi gerçek sorgu deseni `LEFT JOIN ... WHERE en.id IS NULL`den (henüz İngilizce'de yayınlanmamış topic'leri bulma) gerçekten kullanışlı, varsayımsal bir index -- dil ya da yayın durumundan bağımsız olarak her `topic_translation` satırı üzerindeki bir index'ten daha küçük, ve her yazımda sürdürmesi daha hızlı, çünkü yalnızca `WHERE` koşuluna uyan satırların onda güncellenmesi gerekir. Bir partial index gerçek bir trade-off'tur: yalnızca kendi koşulu index'in `WHERE` cümlesiyle eşleşen (ya da onun tarafından ima edilen) sorgulara yardımcı olur -- `language = 'tr'`de filtreleyen bir sorgu, `WHERE language = 'en'` olarak tanımlanan bir index'ten hiç fayda görmez.

Expression Index'ler

Bir index, ham bir kolon değeri yerine *bir ifadenin sonucu* üzerine de inşa edilebilir -- bir sorgu bir kolonun hesaplanmış ya da dönüştürülmüş bir versiyonuna göre filtrelediği anda kullanışlı:

```
CREATE INDEX idx_topic_slug_lower ON topic (LOWER(slug));
```

"Databases, Schemas, Tables, and Basic SQL Syntax" zaten bu projenin kendi tırnaksız identifier'larının otomatik olarak küçük harfe katlandığını kurmuştu, bu yüzden bu spesifik örnek bu projenin gerçek şeması için varsayımsal -- ama genel desen gerçek ve yaygın: `WHERE LOWER(email) = 'user@example.com'` (büyük/küçük harf duyarsız arama) gibi bir sorgu, yalnızca index'in kendisi `LOWER(email)` üzerine inşa edilmişse bir index kullanabilir, doğrudan `email` üzerine değil -- `email` üzerindeki düz bir index, onun dönüştürülmüş bir

versiyonuna göre filtreleyen bir sorguya yardımcı olmaz, çünkü index ham değerleri saklar, hesaplananları değil.

OFFSET'in Maliyetine Geri Dönmek: Keyset Pagination

"Sorting, Limiting, and Pagination", `OFFSET`in maliyetini -- atlanan her satırı taramak ve atmak -- henüz bir düzeltme yolu olmadan adlandırmıştı. **Keyset pagination** (bazen "seek pagination" olarak adlandırılır), `OFFSET`i, önceden görülen son satır üzerinde bir `WHERE` koşuluyla değiştirir:

```
-- OFFSET 300 LIMIT 20 yerine (300 satır taranmalı ve atılmalı)
SELECT slug, sort_order FROM topic
WHERE sort_order > 300
ORDER BY sort_order
LIMIT 20;
```

`sort_order` üzerinde bir index'le, bu `WHERE sort_order > 300`, "sayfa 16"nın sonuçların ne kadar derininde olduğundan bağımsız olarak doğrudan doğru başlangıç noktasına atlayan bir `Index Scan` olur -- maliyeti bir sayfa sonuçların ne kadar derininde otururs otursun büyümeye devam eden `OFFSET 300`in aksine. Trade-off gerçektir: keyset pagination, `OFFSET` in yapabildiği gibi keyfi bir sayfa numarasına atlayamaz (yalnızca bilinen bir satırdan ileri hareket edebilir), ki bu tam olarak `OFFSET`-tabanlı sayfalamanın, bu projenin kendisi gibi küçük tablolar için daha basit ve daha esnek, bir tablonun boyutu ve erişim deseni gerçekten alternatifi gerektirene kadar doğru varsayılan olarak kalmasının nedeni.

Yaygın Yanlış Anlamalar

"Bir index her zaman bir sorguyu daha hızlı yapar." Otomatik olarak değil -- küçük bir tabloda (bu projenin kendisinin neredeyse hepsi gibi), bir `Seq Scan` çoğu zaman gerçekten daha ucuzdur, ve her index de o tabloya her `INSERT` / `UPDATE` / `DELETE`de bir şeye mal olur, bir `SELECT` üzerindeki `EXPLAIN`in hiç göstermediği bir maliyet. **"Daha fazla index her zaman daha iyidir."** Her biri, planlayıcı tarafından hiç seçilsin ya da seçilmesin, yazma yükü ve depolama ekler -- kullanılmayan bir index, hiçbir sorgunun hiç fayda görmediği saf bir maliyettir. **"`EXPLAIN` ve `EXPLAIN ANALYZE`, yalnızca farklı biçimlendirmeyle aynı şeyi gösterir."** Tek başına `EXPLAIN` sorguyu asla çalıştırmaz -- tahmin eder; `EXPLAIN ANALYZE`, onu gerçekten çalıştırır (bir `SELECT`-olmayan ifade için herhangi bir yan etki dahil), veri yazan bir şey üzerinde çalıştırmadan önce hatırlanmaya değer.

Best Practices

- Yavaş bir sorgunun bir index gerektirdiğini varsaymadan önce `EXPLAIN`e başvur -- bu projenin kendi tabloları, şu anki boyutlarında, `Seq Scan`ın zaten doğru plan olduğu bir durumun gerçek bir örneği, ve bir index eklemek yalnızca sıfır okuma faydasıyla yazma yükü ekler.

- Bu projenin kendi gerçek `idx_topic_category` / `idx_category_course` desenini (ikisi de sürekli join'lenen bir foreign key kolonunu index'liyor) izleyerek, tahmin etmek yerine gerçekten sık filtrelenen ya da join'lenen kolonları index'le.
- Bir sorgu deseni tutarlı olarak bir tablonun küçük, kararlı bir alt kümesine filtrelediği anda (bu projenin kendi varsayımsal "yalnızca yayınlanmış İngilizce çeviriler" durumu gibi) bir partial index'e başvur -- tam-tablo bir index'ten daha küçük ve sürdürmesi daha ucuz kalır.
- Bir sorgunun performansı gerçekten şaşırtıcı olduğunda `EXPLAIN`in tahmini `rows`unu `EXPLAIN ANALYZE`in gerçek `rows`uyla karşılaştır -- aralarındaki büyük bir boşluk, sorgunun kendi mantığından çok eski tablo istatistiklerine işaret eden, kendi başına kullanışlı bir teşhistir.

Yaygın Hatalar

- Küçük bir tabloya bir index eklemek ve `EXPLAIN` hâlâ `Seq Scan` gösterdiğinde kafa karışıklığı yaşamak -- bu illa bir hata ya da boşa harcanmış bir index değil; PostgreSQL'in tablonun şu anki boyutunda index'i henüz kullanmaya değer bulmadığına doğru şekilde karar vermesi.
- `EXPLAIN ANALYZE`i, gerçek yazımın istenmediği bir bağlamda bir yazma ifadesi (`UPDATE` / `DELETE`) üzerinde çalıştırmak, `ANALYZE`in ifadeyi yalnızca tahmin etmek yerine gerçekten çalıştırdığını unutarak -- sonrasında geri alınan bir transaction içine sarmak, bir yazımın etkisini korumadan planını incelemenin güvenli yoludur.
- Bir kolon üzerinde düz bir index inşa etmek, sonra onun dönüştürülmüş bir versiyonuna göre filtrelemek (`LOWER(kolon)`, `kolon + 1`) ve index'in kullanılmadığını görünce şaşırmak -- index'in, sorgunun gerçekte filtrelediği aynı ifade üzerine inşa edilmesi gerekir.
- Hedeflenmiş bir düzeltme yerine varsayılan olarak keyset pagination'a başvurmak, sonra `OFFSET`-tabanlı sayfalamanın -- bu projenin kendi mevcut yaklaşımı -- hâlâ desteklediği keyfi bir sayfa numarasına atlama yeteneğini kaybetmek.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- `EXPLAIN`, sorguyu çalıştırmadan PostgreSQL'in amaçladığı sorgu planını ve maliyet tahminini gösterir; `EXPLAIN ANALYZE`, onu gerçekten çalıştırır ve gerçek geçen zamanı ve satır sayılarını raporlar.
- `Seq Scan`, her satırı okur; `Index Scan`, doğrudan eşleşen satırlara atlamak için bir index kullanır -- PostgreSQL'in hangisini seçtiği, sabit bir kural değil, kendi maliyet tahminine bağlıdır, ve küçük bir tablodaki (bu projenin kendisi gibi) bir `Seq Scan`, çoğu zaman gerçekten daha ucuz seçimdir.
- `CREATE INDEX`, varsayılan olarak bir B-tree'dir, bu projenin kendi `V1__init_schema.sql`'inin `idx_topic_category` ve foreign key kolonları üzerindeki üç diğer gerçek index için zaten kullandığı aynı ifade.

- Bir partial index (`CREATE INDEX ... WHERE ...`), yalnızca satırların bir alt kümesini index'ler; bir expression index (`CREATE INDEX ... (ifade)`), ham bir kolon yerine hesaplanmış bir değeri index'ler -- ikisi de yalnızca kendi koşulu index'lenen şeyle eşleşen bir sorguya yardımcı olur.
- Keyset pagination (`OFFSET` yerine `WHERE sort_order > <son görülen>`), "Sorting, Limiting, and Pagination"ın zaten adlandırdığı maliyeti, keyfi sayfa atlamalarını kaybetmek pahasına düzeltir -- `OFFSET`, bir tablonun boyutu gerçekten alternatifi gerektirene kadar daha basit, doğru varsayılan olarak kalır.

Cheat Sheet

```
EXPLAIN SELECT ...;
EXPLAIN ANALYZE SELECT ...;

CREATE INDEX idx_ad ON tablo (kolon);
CREATE INDEX idx_ad ON tablo (kolon) WHERE koşul;
CREATE INDEX idx_ad ON tablo (LOWER(kolon));

-- Keyset pagination
SELECT * FROM t WHERE sort_key > :son_gorulen ORDER BY sort_key LIMIT :sayfa_boyutu;
```

Terimler Sözlüğü

- **Query plan**: PostgreSQL'in bir sorguyu çalıştırmak için seçtiği kendi stratejisi, `EXPLAIN` tarafından gösterilir.
- **Seq Scan**: bir tablonun her satırını sırayla okur, her birini sorgunun koşuluna karşı kontrol eder.
- **Index Scan**: eşleşen satırları, her satırı okumadan doğrudan bulmak için bir index kullanır.
- **B-tree**: dengeli bir ağaç yapısı, PostgreSQL'in varsayılan index türü, eşitlik ve aralık koşulları için verimli.
- **Keyset pagination**: `OFFSET` yerine son-görülen-satır üzerinde bir `WHERE` koşulu kullanarak sonuçlar arasında sayfalamak, atlanan satırları tarama ve atma maliyetinden kaçınarak.