PostgreSQL-Specific Data Types: UUID, JSON/JSONB, and Arrays

UUID (BIGSERIAL'a alternatif olarak, trade-off'larla), JSON vs. JSONB, ->/->>/@> ile JSONB sorgulama, ve array'ler (ANY/@>/unnest) -- bu projenin gerçek şemasının hiçbirini kullanmadığı dürüstçe belirtilerek, illüstratif ama bu projenin domain'ine bağlı örneklerle. Advanced PostgreSQL kategorisinin 2.'si.

İleri 30 dk

Son güncelleme: 25 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/postgresql-specific-data-types

---

"PostgreSQL Data Types", her ilişkisel veritabanının bir versiyonuna sahip olduğu türleri kapsadı -- tamsayılar, string'ler, boolean'lar, zaman damgaları. Bu ders, gerçekten PostgreSQL'in kendi olan üçünü kapsıyor: `UUID`, `JSON` / `JSONB`, ve array'ler. Baştan açıkça söylemeye değer: bu projenin kendi gerçek şeması üçünden de hiçbirini kullanmıyor -- şimdiye kadarki her tablo `BIGSERIAL` primary key'ler ve düz scalar kolonlar kullandı. Aşağıdaki örnekler, bu projenin kendi domain'inin bilinçli olarak illüstratif genişletmeleri, mevcut kolonlar değil -- ki bu tam olarak her türün gerçekten ona başvurmadan önce gerekli olup olmadığının adil bir testini yapan şey.

UUID: BIGSERIAL'a Bir Alternatif

"Constraints and Keys", `BIGSERIAL PRIMARY KEY` i `BIGINT` artı otomatik artan bir sequence olarak zaten kapsadı. `UUID`, tamamen farklı bir primary-key stratejisi -- geleneksel olarak `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx` olarak yazılan, sırayla dağıtılmak yerine rastgele (ya da UUID versiyonuna bağlı olarak başka girdilerden) üretilen 128-bit bir değer:

```
CREATE TABLE api_token (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    label VARCHAR(255) NOT NULL
);
```

`gen_random_uuid()`, PostgreSQL'in bir kolon varsayılanı olarak rastgele (versiyon 4) bir UUID üretmek için kendi yerleşik fonksiyonudur -- `BIGSERIAL` in örtük sequence'inin doğrudan UUID eşdeğeri, hiçbir uygulama kodu dahil olmadan taze bir id üretir. Bu projenin gerçek şemasının bugün buna ihtiyacı yok -- bu projenin şimdiye kadar ürettiği her `id` (`topic.id`, `category.id`, ve diğer her primary key) bir `BIGSERIAL` oldu, ama yukarıdaki varsayımsal `api_token` gibi bir tablo, bir UUID'in yerini tam olarak hak ettiği türden bir durum: hiç

eklenmeden önce bir istemci tarafından üretilmesi gerekebilecek bir id, ya da sayısal değeriyle kaç token olduğunu ifşa etmemesi gereken bir tane.

UUID vs. BIGSERIAL: Trade-off'lar

Bir `BIGSERIAL` id kompakttır (8 byte), sıralıdır (ki bu bir `PRIMARY KEY` index'inin altında yatan B-tree'sini -- bu kategoride daha sonra "Indexes and Query Performance with EXPLAIN"in ne anlama geldiğini kapsayacağı -- verimli sıralı tutar), ve insan-okunabilirdir, ama aynı zamanda tahmin edilebilirdir ve bilgi ifşa eder: `topic.id = 42`, herkese en az 42 topic'in oluşturulduğunu, tahmin edebilecekleri bir sırada, söyler. Bir `UUID` daha büyüktür (16 byte), tahmin edilemezdir, ve bir satır veritabanına hiç eklenmeden önce bir istemci ya da ayrı bir servis tarafından güvenle üretilebilir -- "bu satır hangi id'yi aldı" öğrenmek için bir gidiş-dönüşe gerek yok, ki bu bağımsız olarak kayıt üreten birden fazla servisi olan bir sistem için önemlidir, bu projenin kendi tek-uygulamalı mimarisinin (CLAUDE.md'de bilinçli bir kısıt olarak belgelenen, bu kursun icat ettiği bir şey değil) basitçe hiç ihtiyaç duymadığı bir senaryo. Hiçbiri evrensel olarak "daha iyi" değildir -- `BIGSERIAL`, harici-üretim ya da bilgi-gizleme gereksinimi olmayan bir içsel primary key için doğru varsayılan olmaya devam eder, ki bu projenin her tablosunun hâlâ onu kullanmasının tam nedeni budur.

JSON ve JSONB

PostgreSQL'in iki JSON türü var: `JSON`, gönderilen tam metni, byte byte saklar, her sorgulandığında yeniden ayrıştırır; `JSONB` ("B", binary için), ayrıştırılmış, daha verimli bir içsel temsil saklar, ve neredeyse her gerçek PostgreSQL şemasının başvurduğu şeydir -- sorgulaması daha hızlı, index'lemeyi destekler (düz `JSON`ın desteklemediği), orijinal key sırası ya da yinelenen key'ler gibi şeyleri korumama pahasına.

```
CREATE TABLE user_preference (
    user_id BIGINT PRIMARY KEY,
    settings JSONB NOT NULL DEFAULT '{}'::jsonb
);
```

Bir `JSONB` kolonu, yapılandırılmış, yarı-değişken veriyi doğrudan tutar -- ayrı bir tablo yok, katı, tam belirtilmiş bir kolon listesi yok -- farklı satırların gerçekten farklı veri şekillerine ihtiyaç duyduğu durumlarda özellikle kullanışlı, ki bu gerçek, dar da olsa gerçek bir boşluk: bu projenin kendi şeması, tamamen sabit, iyi-bilinen kolonlardan (`title`, `summary`, `published`) inşa edilmiş, bu tür bir esnekliğe hiç ihtiyaç duymadı, çünkü her satırın ihtiyaç duyduğu her alan zaten önceden biliniyor.

JSONB Sorgulamak: ->, ->>, ve @>

Üç operatör, çoğu `JSONB` sorgulamasını kapsar. `->`, bir değeri key'e göre çıkarır, `JSONB` olarak tutar; `->>`, aynı değeri metin olarak çıkarır; `@>`, bir `JSONB` değerinin başka birini içerip içermediğini kontrol eder:

```
SELECT settings -> 'theme' FROM user_preference;        -- "dark" (JSONB olarak)
SELECT settings ->> 'theme' FROM user_preference;        -- dark (metin olarak)
SELECT * FROM user_preference WHERE settings @> '{"theme": "dark"}';
```

`->` vs. `->>`, çıkarılan değerin düz bir string olarak karşılaştırılması ya da gösterilmesi gerektiği anda önemli olur -- `settings ->> 'theme' = 'dark'`, metni metinle karşılaştırır; `settings -> 'theme' = 'dark'`, `JSONB` i bir metin literal'iyle karşılaştırırdı ve "aynı görünseler" bile farklı türler oldukları için asla eşleşmezdi. `@>` ("içerir" operatörü), önce onu çıkarmadan iç içe geçmiş bir key'e göre satırları filtrelemeyi pratik kılan şeydir -- `WHERE settings @> '{"theme": "dark"}'`, `settings` JSON'ı o key-değer çiftine sahip olan, nesne başka ne kadar içerirse içersin, her satırı bulur.

Gerçekçi Bir JSONB Örneği

Bu projenin kendi gerçek `code_example` tablosunu (şu anda yalnızca `title` / `example_name` / `sort_order`) varsayımsal bir `metadata JSONB` kolonuyla genişletmek deseni somut olarak gösterir:

```
-- Varsayımsal genişletme, bu projede gerçek bir kolon değil
UPDATE code_example
SET metadata = '{"language": "java", "linesOfCode": 24, "tags": ["records",
"immutability"]}'::jsonb
WHERE example_name = 'PointRecordExample';

SELECT example_name FROM code_example
WHERE metadata @> '{"language": "java"}';
```

Bu, tam olarak ilişkisel bir `code_example` tablosunun doğal olarak sunmadığı türden bir şekil -- değişken uzunlukta bir `tags` listesi ve bir avuç isteğe bağlı, örneğe-özgü gerçek, her olası olan için sabit bir kolon olmadan. Bu projenin gerçekten böyle bir kolon eklemesi gerekip gerekmediği ayrı bir tasarım sorusu ("PostgreSQL Data Types"ın kendi kılavuzunu izleyerek, gerçek, sabit bir `language` `VARCHAR` kolonu, eğer varyasyon gösteren tek alan `language` sa muhtemelen daha iyi seçim olurdu) -- buradaki nokta yalnızca `JSONB` sorgulamasını bu projenin kendi domain'i için makul bir şekle karşı göstermek, bu projenin gerçek şemasının onu eksik olduğunu iddia etmek değil.

Array'ler

PostgreSQL, herhangi bir kolon türünün, `[]` kullanılarak o türden bir array olarak bildirilmesine izin verir:

```
CREATE TABLE code_example (
    ...
    tags TEXT[]
);

INSERT INTO code_example (title, example_name, sort_order, tags)
VALUES ('Point Record Example', 'PointRecordExample', 1, ARRAY['records', 'immutability']);

SELECT * FROM code_example WHERE 'records' = ANY(tags);
SELECT * FROM code_example WHERE tags @> ARRAY['records'];
```

`ANY(tags)`, tek bir değerin array'in herhangi bir yerinde görünüp görünmediğini kontrol eder -- `IN` in array eşdeğeri, ama sabit bir literal listesine karşı değil, tek bir satırın kendi array kolonuna karşı. `@>`, array'lerde `JSONB` de çalıştığı aynı şekilde çalışır -- "bu array o array'i (ya da o tek-elemanlı array'i) içeriyor mu."

Array'leri Sorgulamak

Üyelik ötesinde, PostgreSQL bir array'e indekslemeyi (`tags[1]`, 1-indeksli, 0-indeksli değil -- Java'dan gerçek, unutulması kolay bir fark), array uzunluğunu (`array_length(tags, 1)`), ve bir array kolonunu eleman başına bir satıra genişleten `unnest(tags)` i destekler -- bir array'in bir an için satırlar olarak ele alınması, başka herhangi bir tablo verisi gibi join'lenmesi ya da aggregate edilmesi gerektiğinde kullanışlı. Bir `tags TEXT[]` kolonu ve bir `topic_tag` link tablosuyla join'lenen ayrı bir `tag` tablosu (Java tarafında "Relationships, Fetching, and the N+1 Problem"in zaten kapsadığı tam `@ManyToMany` deseni), benzer görünen bir problemi farklı çözer -- array, "hangi topic'ler bu tag'i paylaşıyor"u ölçekte verimli bir şekilde sorgulamaya gerek olmayan küçük, yapılandırılmamış bir liste için daha basittir; tag'ler kendi metadata'sına ihtiyaç duyduğunda, tek bir yerde yeniden adlandırılması gerektiğinde, ya da diğer yönde gerçekten verimli bir arama gerektiğinde gerçek bir join tablosu kazanır.

Yaygın Yanlış Anlamalar

"`JSON` ve `JSONB` birbirinin yerine geçebilir, `JSONB` yalnızca daha yeni." Pek değil -- `JSON`, tam girdi metnini korur (key sırası ve yinelenen key'ler dahil) ama her okumada yeniden ayrıştırır; `JSONB` normalize eder ve index'lenebilir, ve neredeyse her gerçek kullanım durumu için doğru varsayılandır, ama ikisi her senaryoda birbirinin yerine geçen değişimler değildir. **"Bir UUID primary key her zaman `BIGSERIAL` den daha güvenlidir."** Sıralı bilgiyi gizler, ki bu gerçek, dar bir fayda, ama "bir sonraki id'yi tahmin etmek daha zor" "güvenli" ile aynı şey değil -- erişim kontrolü, key türünden bağımsız olarak hâlâ uygulama /yetkilendirme katmanında gerçekleşmelidir. **"Veriyi `JSONB` olarak saklamak bir şema tasarlama ihtiyacını ortadan kaldırır."** Tasarım kararını erteler, ortadan kaldırmaz -- o `JSONB` kolonunun her tüketicisi, hangi key'lerin görünebileceği ve ne anlama geldiği konusunda, bunu zorunlu kılacak bir `CREATE TABLE` ifadesi ya da "Constraints and Keys" olmadan, gayri-resmi olarak hâlâ anlaşmalıdır.

Best Practices

- İçsel primary key'ler için varsayılan olarak `BIGSERIAL`e ("Constraints and Keys"teki `GENERATED ALWAYS AS IDENTITY` ya da) git, ve özellikle bir değerin veritabanı dışında üretilmesi gerektiğinde, ya da sıralı bilgiyi gizlemek gerçek bir gereksinim olduğunda `UUID`e başvur -- varsayılan bir "daha modern" seçim olarak değil.
- Tam girdi biçimlendirmesini korumak için özel bir neden olmadıkça düz `JSON` yerine `JSONB`i tercih et -- JSON saklayan neredeyse her gerçek şema `JSONB` kullanır.
- Bir `JSONB` ya da array kolonuna yalnızca gerçekten değişken, satır-başına-farklı veri için başvur -- bu projenin kendi şeması, tamamen önceden bilinen sabit kolonlardan inşa edilmiş, ikisine de hiç ihtiyaç olmadığı zamanın gerçek bir örneği.
- "Liste"nin kendi metadata'sına ihtiyaç duyduğu, tek bir yerde yeniden adlandırılması gerektiği, ya da diğer yönden verimli sorgulanması gerektiği (hangi topic'ler bu tag'i paylaşıyor) anda bir array kolonu yerine gerçek bir join tablosunu seç -- array, bu ihtiyaçların hiçbiri olmayan küçük, basit, satır-başına bir liste için iyi bir uyum olarak kalır.

Yaygın Hatalar

- `->`-ile-çıkarılmış bir `JSONB` değerini `->>` kullanmak yerine düz bir metin literal'iyle karşılaştırmak (`settings -> 'theme' = 'dark'`) -- bu, açık bir hata üretmek yerine, `JSONB` ve `text` farklı türler olduğu için sessizce asla eşleşmez.
- Bir şemaya karar vermekten kaçınmanın bir yolu olarak bir `JSONB` kolonuna başvurmak, sonra her sorgunun yalnızca konvansiyonla tam key adlarını ve türlerini bilmesi gerektiğini keşfetmek, "Constraints and Keys"in gerçek kolonlar için zaten kapsadığı garantilerin hiçbiri olmadan.
- PostgreSQL array'lerinin 1-indeksli olduğunu unutmak (`tags[1]` ilk elemandır), ve Java alışkanlığından 0-indeksli olduğunu varsayarak ya `NULL` ya da bir birer-kayma sonucu almak.
- Kendi tanımına ya da kendi açıklamasına ihtiyaç duyan, ya da verimli "bu tag'e sahip tüm topic'leri bul" aramaları gerektiren gerçekten ilişkisel bir çoktan-çoğa'yı gerçek bir join tablosu yerine bir array kolonu olarak modellemek, sonra tam olarak bu şekildeki problem için "Relationships, Fetching, and the N+1 Problem"in zaten kapsadığı deseni kullanmak yerine daha sonra array'in sınırlamalarını dolanmak zorunda kalmak.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- `UUID` (genellikle `gen_random_uuid()` ile üretilir), primary key'ler için `BIGSERIAL`e bir alternatiftir -- kompakt ve sıralı yerine büyük ve tahmin edilemez, özellikle bir id'nin veritabanı dışında üretilmesi gerektiğinde ya da sıra bilgisini ifşa etmemesi gerektiğinde kullanışlı.

- `JSONB` (neredeyse her gerçek durumda düz `JSON`a tercih edilir), yarı-yapılandırılmış, satır-başına-değişken veri saklar; `->`/`->>`, bir key'i sırasıyla `JSONB`/metin olarak çıkarır, ve `@>` içermeyi kontrol eder.
- Array'ler (`TEXT[]`, vb.), tek bir kolonda doğrudan değişken uzunlukta bir liste tutar; `ANY(...)` üyeliği kontrol eder, `@>` içermeyi kontrol eder, ve `unnest(...)` bir array'i satırlara genişletir.
- Bu üçünden hiçbiri bu projenin kendi gerçek şemasında hiçbir yerde görünmez -- buradaki her örnek bilinçli olarak etiketlenmiş, illüstratif bir genişletme, mevcut bir kolon değil, ki bu tam olarak nokta budur: bu türlere yalnızca gerçek bir ihtiyaç var olduğunda başvur.
- Gerçek bir join tablosu (zaten "Relationships, Fetching, and the N+1 Problem"de kapsanan), içindeki verinin kendi yapısına, kısıtlarına, ya da diğer yönden verimli sorgulanmaya ihtiyaç duyduğu anda bir `JSONB`/array kolonu yerine doğru seçim olarak kalır.

Cheat Sheet

```
id UUID PRIMARY KEY DEFAULT gen_random_uuid()

data JSONB
data -> 'key'           -- JSONB olarak
data ->> 'key'           -- metin olarak
data @> '{"key": "v"}'  -- içerir

tags TEXT[]
'x' = ANY(tags)          -- üyelik
tags @> ARRAY['x']        -- içerme
tags[1]                   -- 1-indeksli
unnest(tags)               -- eleman başına bir satır
```

Terimler Sözlüğü

- **UUID**: tipik olarak rastgele üretilen, sıralı bir primary key'e alternatif olarak kullanılan 128-bit bir tanımlayıcı.
- **JSONB**: PostgreSQL'in binary, index'lenebilir JSON türü -- neredeyse her durumda düz `JSON`a tercih edilen JSON saklama türü.
- **Containment operatörü (@>)**: bir `JSONB` değerinin ya da array'in başka birini içerip içermediğini kontrol eder.
- **unnest()**: bir array kolonunu eleman başına bir satıra genişleten bir fonksiyon.