## **PostgreSQL’de Veri Tipleri**

PostgreSQL’de bir çok veri tipi vardır. 400’e yakın diyebiliriz.  
Fakat veri tipleri bunlarla da sınır değil, eklentiler sayesinde kendi veri tipimizi yaratabilir, kullanabiliriz.

Bu anlatım boyunca öncelikle PostgreSQL’de bulunan temel veritiplerini inceleyeceğiz, daha sonrasında ise Sıralama (Alignment) işleminin nasıl yapıldığına bakacağız.

Şimdi anlatımın ileriki safhalarında Sıralama (Alignment) işlemini anlatırken kullanacağımız bir senaryo düşünelim.  
Bu senaryoda gelişi güzel olarak kolonların sıraladığı bir bir tablo olsun.

Bu tablo aklımızın bir köşesinde dursun. Yazının sonunda geri döneceğiz. Şimdi PostgreSQL’de bulunan temel veritiplerini inceleyelim.

|  |  |  |
| --- | --- | --- |
| **Veri Adı** | **Veri Tipi** | **Veri Tipi Uzunluğu(byte)** |
| is\_shipped | bool | 1 |
| user\_id | int8 | 8 |
| order\_total | numeric | -1 |
| order\_dt | timestamptz | 8 |
| order\_type | int2 | 2 |
| ship\_dt | timestamptz | 8 |
| item\_ct | int4 | 4 |
| ship\_cost | numeric | 1 |
| receive\_dt | timestamptz | 8 |
| tracking\_cd | text | -1 |
| id | int8 | 8 |

## **Temel Veri Tipleri**

#### **Char**

1 byte veri saklamak için kullanılır.

#### **Char(n)**

n byte veri saklamak için kullanılır. Bu n ifadesi sabittir.  
Char(n) kullanımı bir çok senaryoda mantıksızdır.

Char(5) olduğunu hayal edin.  
İçersine “F” ifadesini yazıp saklarsak geriye 4 byte’lık boş alan kalacaktır.  
Char(n) sabit uzunlukta veriler saklanacağında mantıklı olabilir.

#### **Varchar(n)**

n byte’a kadar veri saklamak amacıyla kullanılır. 1 GB ile sınırlıdır.  
Char(n) yerine bunu kullanmak bir çok senaryoda çok daha mantıklı olacaktır.

**Not**

PostgreSQL’de verileri birbirine **cast** etmek için **::** operatörü kullanılır. Aşağıdaki gibi kullanılır.

'F'::varchar(5) **as** vc5;  
'F'::char(5) **as** c5;

Bu iki verinin boyutu birbirinden farklıdır.  
vc5’de yalnızca F’in boyutu kadar alan gerekli iken, c5’de sabit bir değer olan 5 byte kadar alan açılır.

#### **Text**

1 GB’a kadar string saklamak amacıyla kullanılır.

#### **smallint, int2**

2 byte’lık integer veri saklamak amacıyla kullanılır.  
smallint ve int2 ifadeleri birbirinin alias’ı durumundadır.

#### **integer, int4, int**

4 byte’lık integer veri saklamak amacıyla kullanılır.

#### **bigint, int8**

8 byte’lık integer veri saklamak amacıyla kullanılır.

#### **Double precission**

8 byte alan kaplar. 15 ondalık basamaklı veri saklamak amacıyla kullanılır.

#### **Reel**

4 byte alan kaplar. 6 ondalıklı basamaklı.

#### **Numeric**

20 ondalık basamaklı saklamak için kullanılır.

#### **Serial**

Aslında böyle bir veri tipi yoktur. Diğer database’lerde bulunan **Auto Increment** özelliğinin karşılığıdır.

0’dan başlar. Tabloya bir veri girildiğinde bir artırılır, 1 olur.  
Serial int4 tipindedir Integer tiplerine göre smallserial, bigserial türleri de vardır.

Serial’lar tablonun dışında ayrı bir nesne olarak saklanır. **Sequance** sayesinde bu yapı sağlanır.

**Direkt müdahale edebilir miyiz ?**

ID ifadesinin serial olarak tanımlandığını düşünün. Normalde tabloya her veri girdiğiizde ID ifadesine hiç dokunayız ve veriler düzenli bir şekilde 1’er artarak devam eder. Peki ID ifadesini elle yazabilir miyiz ?

Evet. Teknik olarak serial ifadesi bir integer değerin karşılığı olduğu için bu integer ifadesini tabloya elle veri girerken verebiliriz.

Serial ifadeler eşsiz olacak diye bir kural yoktur. Tabloda serial ifadesinin eşsiz olacağı belirtilmemiş ise aynı veri olsa bile elle girebiliriz.

#### **Boolean**

True ya da false değer tutmak için kullanılır. Oluşturulurken; Y, N, y, n, true, false, T, F, 1, 0 verilebilir, içerde true ve false’a dönüştürülür.

#### **Range**

Aynı kolon içersinde bir aralık tutabilmek amacıyla kullanılır. Arama sonuçlarını çok hızlı alabiliriz. Index’lenir. Alt ve üst değer olarak tanımlanır.

#### **Enum**

Statik, sıralı veri kümeleri saklamak için kullanılır.

**CREATE** **TYPE** haftanin\_gunleri **AS** ENUM ('Pazartesi', 'Salı', .... , 'Pazar');

#### **Domain**

Validasyon yapmak amacıyla kullanılır. Örneğin aşağıdaki örnekte US posta kodu için bir validasyon yapısı kurulmuştur. [**[kaynak]**](https://www.postgresql.org/docs/current/sql-createdomain.html)

**CREATE** **DOMAIN** us\_postal\_code **AS** TEXT  
CHECK(  
 VALUE **~** '^\d{5}$'  
OR VALUE **~** '^\d{5}-\d{4}$'  
);  
  
CREATE **TABLE** us\_snail\_addy (  
 address\_id SERIAL **PRIMARY** **KEY**,  
 street1 TEXT **NOT** **NULL**,  
 street2 TEXT,  
 street3 TEXT,  
 city TEXT **NOT** **NULL**,  
 postal us\_postal\_code **NOT** **NULL**  
**);**

#### **Date**

Tarih tutmak amacıyla kullanılır. Varsayılan olarak Y/M/D formatındadır. Datestyle gösterim şekli değiştiribilir.

#### **Time**

Saat saklamak içindir.

#### **Timestamp**

Date ve Time saklamak içindir. Timezone bulunmaz.

#### **Timestamptz**

Timestamp’a Timezone’un eklenmiş halidir.

#### **now()**

Transaction’a başlandığındaki zamanı gösterir. (current\_timestamp)

**Not**

PostgreSQL’de her şey transaction içersinde çalıştırılır.

#### **clock\_timestamp()**

Transaction’a başlama zamanı yerine gerçekten şuanki zamanı verir.

#### **Array**

PostgreSQL’in en güzel yanlarında biri de verilerin Array tipinde saklanabilmesidir.

**CREATE** **TABLE** sal\_emp (  
 name text,  
 pay\_by\_quarter integer[],  
 schedule text[][]  
);

#### **JSONB**

JSON verinin binary olarak sıkılaştırılarak tutulmasıdır.  
PostgreSQL’in NoSQL olarak tanımlanabilir, her nekadar yanlış olsa da.  
NoSQL veritabanlarından daha eski bir geçmişe sahiptir.  
Index’lenebilir.

### **Senaryolara göre doğru veritipini nasıl seçeceğiz?**

Bir senaryo çizelim. Tablomuzda T.C. kimlik numarası tutmanız gerekiyor. Hangi tipini kullanmak daha doğru olur? [**[kaynak]**](https://blog.gunduzdanismanlik.com/index.php?/archives/20-TC-KMLK-NUMARASINDA-KULLANILAN-VER-TP.html)

Belki de ilk aklınıza gelen char(11) olabilir. Peki araştırma yapmadan direkt olarak ilk akla geleni kullanmak sizce doğru olur mu?

İhtimalleri gözden geçirelim,  
Integer tipinde tutamayız çünkü 11 haneyi desteklemez,  
Big integer olarak tutabiliriz.  
Ya da belki de Text tipini tercih edebiliriz.

Elimizde 3 seçenek var, hangisinin maliyeti en düşünük buna bakmalıyız.

İhtimalleri tek tek ele aldığımızda, 1.000.000 TC kimlik numarası saklandığında maliyeti aşağıdaki gibir.

* VARCHAR(11), indexler dahil, 115 MB,
* TEXT, indexler dahil, 72 MB, 7.605ms,
* CHAR( ), indexler dahil, 72 MB, 7.847ms,
* BIGINT, indexler dahil, 56 MB

Bu durumda belki de Text ya da Char olarak tutmanın daha iyi olabileceğini düşündüğümüz T.C. kimlik numarası en ucuza Bigint türünde saklandığını görmüş olduk. Veri tiplerini doğru kullanırsak çok fazla performans ve maliyet kazancı sağlayabiliriz.

### **Örneğimize geri dönelim.**

Şimdi veri tiplerinin genel hatlarıyla tanıdığımıza göre konunun başında verdiğimiz örneğe geri dönebiliriz.

|  |  |  |
| --- | --- | --- |
| **Veri Adı** | **Veri Tipi** | **Veri Tipi Uzunluğu(byte)** |
| is\_shipped | bool | 1 |
| user\_id | int8 | 8 |
| order\_total | numeric | -1 |
| order\_dt | timestamptz | 8 |
| order\_type | int2 | 2 |
| ship\_dt | timestamptz | 8 |
| item\_ct | int4 | 4 |
| ship\_cost | numeric | -1 |
| receive\_dt | timestamptz | 8 |
| tracking\_cd | text | -1 |
| id | int8 | 8 |

Bir yazılımcı olarak bu kolonları nasıl sıralardınız ? Hepiniz farklı bir cevabı olabilir. Kolayına nasıl gelirse. Yanlış.

Bu kolonların sıralaması performansı ya da veritabanı boyunu etkiler mi konusu hiç aklınıza geldi mi ?

## **Sıralama (Alignment)**

PostgreSQL’de verile 8 byte’lık kolonlar halinde tutulur. Şimdi elimizde aşağıdaki veriler olduğunu düşünün.

* smallint (2byte),
* bigint (8),
* int (4)

Bu veriler yukarıdaki gibi sıralandığında postgresql üzerinde aşağıdaki gibi tutulacaktır.

Toplam 20 byte’lık alan kullanmış olduk.

------------ ------------ ------------  
| 8 Byte | | 8 Byte | | 8 Byte |  
 ------------ ------------ ------------  
 | | | | | | | |  
 -- ----- ------- ----  
 | | | |---------------  
 | | | |  
 | | |------------| |  
 | | | |  
 | |------- | |  
 | | | |  
 | | | |  
 | | | |  
 | | | |  
 v | v v  
 2 Byte v 8 Byte 4 Byte  
 smallint 6 Byte bigint int  
 padding

### **Büyükten küçüğe doğru sıralasak nasıl bir sonuç alırdık?**

* bigint (8),
* int (4),
* smallint (2byte)

**Şeklinde sıralasak nasıl olurdu ?**

Bu durumda 14 byte alan kullanılacaktır. Gördüğünüz gibi yalnızca sıralayı farklı yaparak veritabanımızda büyük ölçüde yer tasarafu yaptık.

En optimum sonuç için veritiplerine göre büüykten küçüğe sıralamak uygun olacaktır.

Böylece hem okuma süreci hızlanır, hem alandan tasarruf yapılır hem de i/o azalır.

------------ ------------ ------------  
| 8 Byte | | 8 Byte | | 8 Byte |  
 ------------ ------------ ------------  
 | | | | | |  
 ----------- ----- --  
 | | |  
 | | |  
 | | |-----------------|  
 | | |  
 | |------- |  
 | | |  
 | | |  
 | | |  
 | | |  
 v | v  
 8 Byte v 2 Byte  
 bigint 4 Byte smallint  
 int

### **ORM'ların performans ile durumu nasıl?**

Şimdi düşünelim. SQL sorgularını biz yazmıyor, db’e direkt müdahale etmiyor olalım. Yani ORM kullanalım, performans kaybı yaşar mıyız ? Evet.

Örneğin çoğu ORM int’leri bigint olarak tutar. Her integer değeri için 4 byte kayıp yaşadığınızı düşünün.

### **Nasıl doğru sıralamayı bulacağız?**

Bir tablo yaratırken veri tiplerinin boyutlarını düşünüp sıralamak can sıkıcı olabilir. O yüzden en başta gelişi güzel yapıp ardından aşağıdaki SQL sorgusu ile doğru sıralama bulunabilir. [**[kaynak]**](https://www.2ndquadrant.com/en/blog/on-rocks-and-sand/#column-tetris)

**SELECT** a.attname, t.typname, t.typalign, t.typlen  
 **FROM** pg\_class **c**  
 **JOIN** pg\_attribute a **ON** (a.attrelid **=** **c**.oid)  
 **JOIN** pg\_type t **ON** (t.oid **=** a.atttypid)  
 **WHERE** **c**.relname **=** 'user\_order'  
 **AND** a.attnum **>=** 0  
 **ORDER** **BY** t.typlen **DESC**;

Bizim örneğimize göre en uygun kolon sıralaması aşağıdaki gibi yapılmalıdır. Yalnızca sıralamayı değiştirerek takribi olarak 20% alan tasarrufu sağlamış olduk.

|  |  |  |
| --- | --- | --- |
| **Veri Adı** | **Veri Tipi** | **Veri Tipi Uzunluğu(byte)** |
| id | int8 | 8 |
| user\_id | int8 | 8 |
| order\_dt | timestamptz | 8 |
| ship\_dt | timestamptz | 8 |
| receive\_dt | timestamptz | 8 |
| item\_ct | int4 | 4 |
| order\_type | int2 | 2 |
| is\_shipped | bool | 1 |
| tracking\_cd | text | -1 |
| ship\_cost | numeric | -1 |
| order\_total | numeric | -1 |