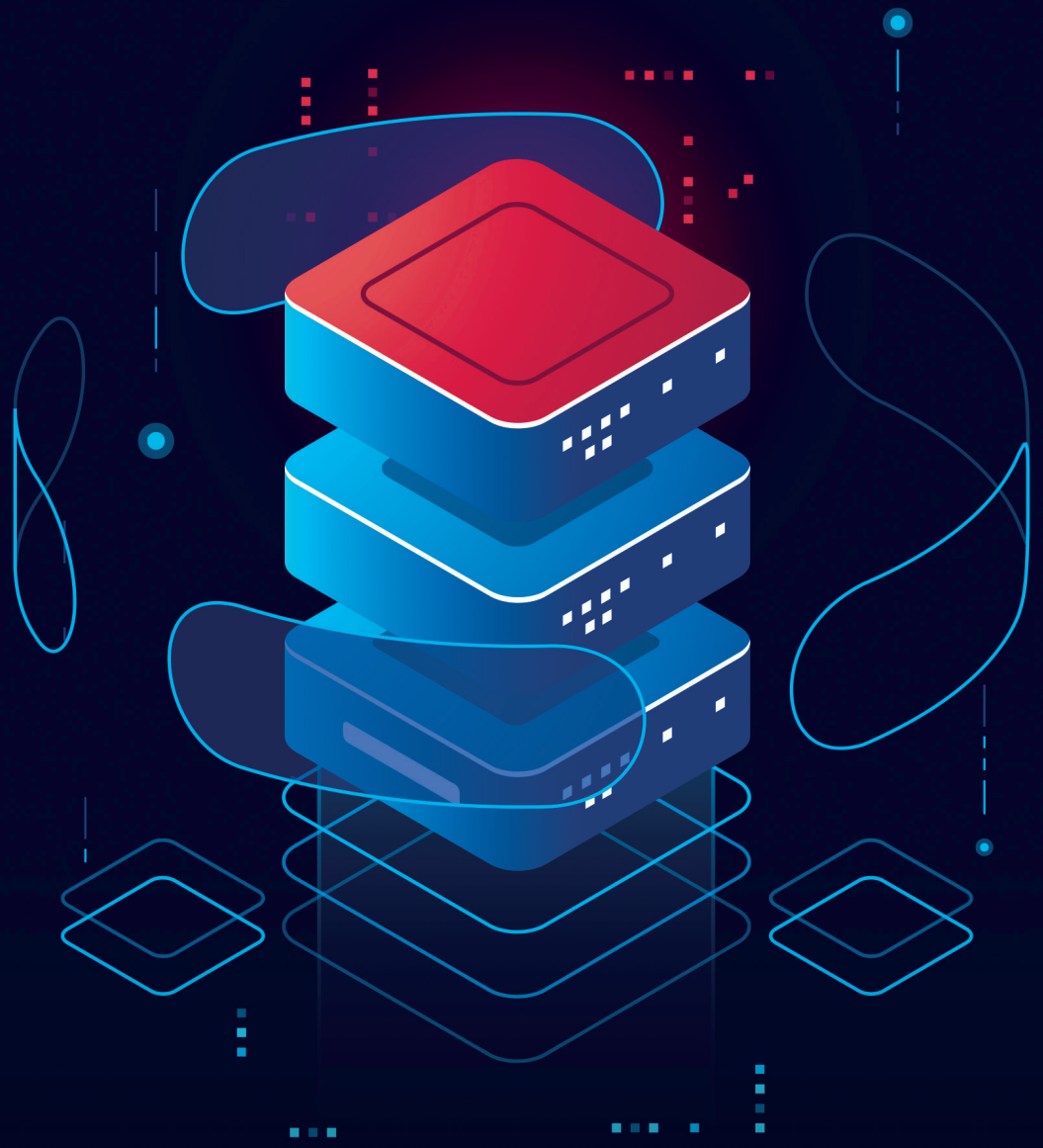
T.C. SANAYİ VE TEKNOLOJİ BAKANLIĞI

#MİLLİ TEKNOLOJİ HAMLESİ

TÜBİTAK

# COCKROACHDB: MODERN ÇAĞIN DAĞITIK VERİ TABANI ÇÖZÜMÜ

ARAŞTIRMA SERİSİ - SAYI 13



**BİLGEM**

YAZILIM TEKNOLOJİLERİ ARAŞTIRMA ENSTİTÜSÜ

**Yazarlar**
Deren TOY
Hatice DURMUŞ
İmtisal AKDEDE

**Yayın Koordinatörü**
Kübra ERTÜRK

**Editörler**
Berkan YILDIRIM
Beyza ŞENEL
Tuğçe YILMAZ

**Tasarım**
Şeyma KOÇER



**İletişim:** 0(312) 289 92 22 - yte.bilgi@tubitak.gov.tr

**https://bilgem.tubitak.gov.tr/yte/**



*Bu seride OpenAI ChatGPT (Sürüm 4) ile oluşturulan görseller kullanılmıştır.

# Simge ve Kısaltmalar

| Kısaltmalar | Açıklama |
|---|---|
| **ACID** | Atomicity (Atomiklik), Consistency (Tutarlılık), Isolation (İzolasyon), Durability (Dayanıklılık) |
| **AP** | Availability (Erişilebilirlik), Partition Tolerance (Bölünme Toleransı) |
| **BİLGEM** | Bilişim ve Bilgi Güvenliği İleri Teknolojiler Araştırma Merkezi |
| **CA** | Consistency (Tutarlılık), Availability (Erişilebilirlik) |
| **CAP** | Consistency (Tutarlılık), Availability (Erişilebilirlik), Partition Tolerance (Bölünme Toleransı) |
| **CP** | Consistency (Tutarlılık), Partition Tolerance (Bölünme Toleransı) |
| **CRDB** | CockroachDB |
| **DML** | Data Manipulation Language (Veri İşleme Dili) |
| **FDW** | Foreign Data Wrapper (Yabancı Veri Sarıcı) |
| **HA** | High Availability (Yüksek Erişilebilirlik) |
| **JSON** | JavaScript Object Notation (JavaScript Nesne Notasyonu) |
| **KV** | Key-Value (Anahtar-Değer) |
| **LRU** | Least Recently Used (En Eski Kullanılan) |
| **LSM** | Log-Structured Merge Tree (Log-Yapılı Birleştirme Ağacı) |
| **MemTable** | Memory Table (Bellek Tablosu) |
| **MVCC** | MultiVersion Concurrency Control (Çoklu Sürüm Eş Zamanlılık Kontrolü) |
| **OLAP** | Online Analytical Processing (Çevrim İçi Analitik İşleme) |
| **OLTP** | Online Transaction Processing (Çevrim İçi İşlem İşleme) |
| **PITR** | Point-in-Time Recovery (Zamana Bağlı Veri Kurtarma) |
| **RDBMS** | Relational Database Management System (İlişkisel Veri Tabanı Yönetim Sistemleri) |
| **SPOF** | Single Point of Failure (Tek Hata Noktası) |
| **SQL** | Structured Query Language (Yapılandırılmış Sorgulama Dili) |
| **SSTable** | Sorted String Table (Sıralı Dizi Tablosu) |
| **TÜBİTAK** | Türkiye Bilimsel ve Teknolojik Araştırma Kurumu |
| **Txn** | Transaction (Veri Tabanı İşlemi) |
| **UDF** | User-Defined Function (Kullanıcı Tanımlı Fonksiyon) |
| **UUID** | Universally Unique Identifier [Evrensel Tekil Tanımlama Numarası (ETTN)] |
| **WAL** | Write-Ahead Log (Önceden Yazma Logu) |
| **XA** | eXtended Architecture (Genişletilmiş Mimari) |
| **YTE** | Yazılım Teknolojileri Araştırma Enstitüsü |

# İçindekiler


Ön Söz 6

Giriş 7

Geleneksel Merkezi Veri Tabanı Sistemleri 8

NewSQL 9

Dağıtılmış SQL 9

1. CAP Teoremi 10

1.1 CA 11

1.2 AP 11

1.3 CP 11

2. CAP ve CockroachDB 12

CockroachDB 13

1. Küme Mimarisi 13

2. Yazılım Katmanları 16

2.1 SQL Katmanı 17

2.1.1 KV Depolama Sistemindeki Tablolar 18

2.1.1.1 Sütun Aileleri 19

2.1.1.2 Benzersiz Olmayan İndeksler 19

2.1.1.3 Benzersiz İndeksler 20

2.1.1.4 Ters Çevrilmiş İndeksler 21

2.1.1.5 STORING İfadesi 21

2.1.1.6 Tablo Tanımları ve Şema Değişiklikleri 22

2.2 Transaction Katmanı 22

2.2.1 MVCC 23

2.2.2 Transaction İş Akışı 24

2.2.3 Paralel Commit 26

2.2.4 Transaction Temizleme 27

2.2.5 Transaction'ın Gerçekleşme Adımları 28

2.2.6 Çatışmalar 29

2.3 Dağıtım Katmanı 31

2.3.1 Meta Aralıkları 31
2.3.2 Dedikodu Protokolü 32
2.3.3 Aralık Bölünmeleri 33
2.3.4 Çok Bölgeli Dağıtım 35
2.4 Replikasyon Katmanı 35
2.4.1 Raft Algoritması 35
2.4.1.1 Log Replikasyonu 36
2.4.1.2 Lider Seçimi 37
2.5 Depolama Katmanı 38
2.5.1 Log-Yapılı Birleştirme Ağaçları 38
2.5.1.1 LSM Yazma İşlemi 39
2.5.1.2 SSTable ve Bloom Filtreleri 40
2.5.1.3 LSM Okuma İşlemi 40
2.5.1.4 Blok Önbelleği 41
2.5.1.5 Çöp Toplama ve Korunan Zaman Damgaları 41
3. Bölümleme 42
3.1 Dikey Bölümleme 42
3.2 Yatay Bölümleme 43
4. SQL Performansını Artırmak için En İyi Yöntemler 43
5. Sütun Veri Tipi Değiştirme Kısıtlamaları 44
6. Ne Zaman Kullanılmalı? 45
7. Ne Zaman Kullanılmamalı? 45
CockroachDB ve PostgreSQL 46
1. Desteklenmeyen SQL Özellikleri 47
2. Yedekleme ve Kurtarma 48
Geliştirme Ortamında PostgreSQL'den CockroachDB'ye Geçiş 49
1. Docker ile CockroachDB Kurulumu 49
2. Spring Boot Uygulamasının CockroachDB ile Bağlanması 52
3. Karşılaşılan Sorunlar 52
Sonuç ve Öneriler 54
Kaynakça 55

# Ön Söz

TÜBİTAK BİLGEM Yazılım Teknolojileri Araştırma Enstitüsü (YTE), 2012 yılından bu yana yazılım teknolojilerinde Ar-Ge faaliyetleri yürüten bir araştırma kuruluşudur. Araştırma faaliyetlerinde elde ettiği birikimini stratejik, hassas ve kritik projeler yürüterek kamu adına hayata geçirmekte; kurumlarımıza dijital dönüşüm, yazılım geliştirme teknolojileri ve kalite süreçleri konusunda danışmanlık vermektedir.

TÜBİTAK BİLGEM YTE tarafından hazırlanan Araştırma Serisi ile kurum içi içerik üretme çalışmalarının yaygınlaştırılması ve hazırlanan içeriklerin sektörün erişimine açılması amaçlanmaktadır. Araştırma Serisi'nde yayınlanan çalışmalar TÜBİTAK BİLGEM YTE çalışanlarının projelerde elde ettiği bilgi birikimini paylaşmak adına derlenmiştir. Bu çalışmalar ile ülkemizin yazılım sektörüne katkı sağlanılması hedeflenmektedir.

# Giriş

Dijital çağın hızla değişen talepleri, veri tabanı teknolojilerinde yenilikçi yaklaşımların önünü açmaktadır. Bu dinamik ortamda modern uygulamalar, sadece yüksek performans değil, aynı zamanda yüksek erişilebilirlik ve esnek ölçeklenebilirlik gibi özellikler de beklemektedir. Geleneksel veri tabanı sistemleri, verileri sabit şemalarla yapılandırma ve tüm veri işlemlerini merkezi bir konum üzerinden yönetme eğiliminde oldukları için, bu yeni gereksinimleri karşılamakta yetersiz kalabilmektedir. Mevcut veri tabanı sistemlerinin, modern uygulamaların ihtiyacı olan bu özellikleri bir arada sunamaması üzerine NewSQL veri tabanı sistemleri geliştirilmiştir. Bu yeni yaklaşım, geleneksel veri tabanlarının sağladığı ACID ve sorgu dili esnekliğini; NoSQL veri tabanı sistemlerinin ölçeklenebilirlik, dağıtım kolaylığı ve hata toleransı gibi özellikleriyle birleştirmektedir.

CockroachDB, NewSQL kavramının somut bir örneği olarak bu talepleri karşılamak üzere tasarlanmış, güçlü özellikleriyle ön plana çıkan, dağıtık mimarili bir veri tabanı çözümüdür. Bu veri tabanı çözümü, yenilikçi mimarisi, yüksek düzeydeki hata toleransı, otomatik ölçeklenme yetenekleri ve veri bütünlüğüne olan katkılarıyla modern uygulama geliştiricilerinin karşılaştığı zorlu sorunlara etkili çözümler sunmaktadır.

PostgreSQL ile olan yüksek uyumluluğu sayesinde mevcut PostgreSQL kullanıcılarının CockroachDB'ye geçişi oldukça kolay bir şekilde gerçekleştirilebilmektedir. Bu durum, PostgreSQL'de bulunan SQL tabanlı sorgu dilleri ve araç setlerini kullanarak daha güçlü ve ölçeklenebilir bir sistem inşa edilmesine olanak tanımaktadır. Aynı zamanda, coğrafi olarak yayılmış veri merkezlerinde bile yüksek performans ve erişilebilirlik sağlayan CockroachDB, küresel çapta faaliyet gösteren uygulamalar için ideal bir tercih haline gelmektedir.

Bu araştırma serisinde, CockroachDB'nin özellikleri, avantajları ve potansiyel uygulama senaryoları üzerine yoğunlaşılacaktır. Aynı zamanda veri tabanı modelinin, günümüzün teknolojik zorlukları ve geliştiricilerin karşılaştığı teknik problemlere nasıl çözümler sunduğu incelenecektir. Bu kapsamda okuyucuları yenilikçi veri tabanı çözümüyle ayrıntılı bir şekilde tanıştırmak, teknolojik tercihlerini bilinçli bir biçimde şekillendirmelerine katkıda bulunmak hedeflenmektedir.

# Geleneksel Merkezi Veri Tabanı Sistemleri

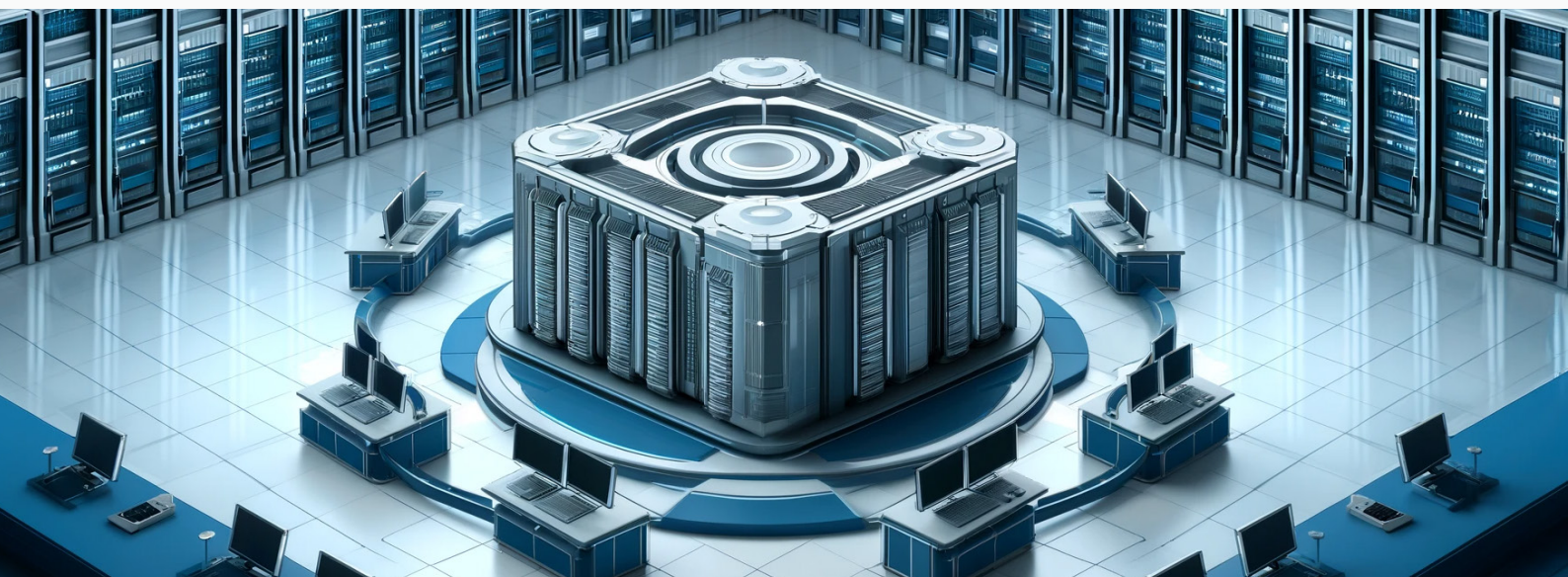
Geleneksel merkezi veri tabanı sistemleri, tüm verilerin tek bir yerde toplandığı, işlendiği ve yönetildiği veri yönetim sistemleridir. Veri erişimi, güncellemesi, yönetimi ve güvenliği tek bir sunucu aracılığıyla sağlanmaktadır. RDBMS olarak da anılan bu sistemler, SQL gibi sorgulama dillerini kullanarak veri sorgulama ve veriyle etkileşimde bulunma imkanı sunmaktadır. RDBMS'ler sayesinde kullanıcılar, veri tabanındaki verileri yönetebilmekte; bu da onlara veri oluşturma, okuma, güncelleme ve silme yetkisi tanımaktadır.

Bu sistemlerin sunduğu önemli avantajlardan biri, veri bütünlüğü ve güvenilirliği sağlamalarıdır. Birincil ve yabancı anahtarlar gibi bütünlük kısıtlamaları verinin doğruluğunu ve tutarlılığını korumaktadır. ACID özellikleri ile veri tabanı işlemleri (transaction'lar) güvenilir bir biçimde yürütülmekte; bu da sistemlerin güvenilirliğini artırmaktadır. Ek olarak bu sistemler, veri erişimini düzenleyen güvenlik özellikleri de sunmakta; kullanıcı bazlı izinler, şifreleme ve erişim kontrolleri sayesinde hassas verilerin korunmasını mümkün hale getirmektedir.

## ACID

ACID, bir transaction'ın doğruluğunu ve güvenilirliğini sağlamak için tasarlanmıştır ve transaction'ın dört temel prensibini ifade etmektedir:

1. **Atomiklik:** Bir transaction ya tamamen gerçekleşir ya da hiç gerçekleşmez. Başka bir deyişle, bir transaction'da birden fazla adım varsa ve bu adımlardan herhangi biri başarısız olursa, tüm transaction geri alınır. Eğer tüm adımlar başarılı olursa, transaction onaylanır.
2. **Tutarlılık:** Transaction'ların, veri tabanı bütünlük kurallarına ve kısıtlamalarına uygun bir şekilde gerçekleştirilmesini ve veri tabanının her zaman tutarlılığını korumasını sağlar.
3. **İzolasyon:** Paralel transaction'ların birbirlerini etkilemeden yürütülmesini sağlar. Bu özellik, eş zamanlı transaction'ların birbirlerine karışmasını engeller.
4. **Dayanıklılık:** Bir transaction onaylandıktan sonra, sonuçları kalıcıdır. Sistemin çökmesi veya güç kesilmesi gibi durumlarda dahi kaybolmamaktadır. Genellikle bu durum değişikliklerin sabit diske yazılmasıyla sağlanmaktadır.



Bu sistemler, sağladıkları olanaklara rağmen modern uygulamaların **artan kullanıcı sayısı ve işlem hacmi** karşısında, milyonlarca eş zamanlı isteği işleme, ölçeklendirme ve erişilebilir olma gibi konularda yetersiz kalabilmektedir. Yetersizlik nedenleri aşağıdaki gibidir:

- Sistem kaynakları (bellek, CPU ve depolama gibi) sınırlıdır. Bu kaynaklar yüksek talep altında tükenmekte ve sistemin performansı düşmektedir.
- Sunucunun kaynaklarını arttırmak (dikey ölçeklendirme) maliyetli ve belirli bir noktadan sonra pratik olmayan bir çözümdür.
- Sunucuda yaşanacak herhangi bir sorun, tüm sistem için kesintiye neden olabilmekte ve sistem erişilemez hale gelebilmektedir. Bu durum SPOF olarak adlandırılmaktadır.

Yaşanabilecek sorunları önlemek, modern uygulamaların yüksek erişilebilirlik ve ölçeklenebilirlik ihtiyaçlarını karşılamak amacıyla "NewSQL" terimi ortaya çıkmıştır.

# NewSQL

"NewSQL" terimi, geleneksel RDBMS'in sağladığı **ACID özellikleri** ile NoSQL sistemlerinin **ölçeklenebilirlik ve dağıtık işleme yeteneklerini** birleştiren yeni nesil veri tabanı sistemlerini tanımlamak için kullanılmaktadır.

NewSQL veri tabanı sistemleri, modern veri yönetimi ve işlem ihtiyaçlarına cevap vermek için tasarlanmıştır. Sunduğu özellikler aşağıdaki gibidir:

- **Yatay Ölçeklenebilirlik ve Elastiklik:** Birden fazla sunucu üzerinde veri ve işlemleri dağıtarak, sistem kaynaklarının genişletilmesi ve yüksek talebin karşılanabilmesini sağlar.
- **Yüksek Erişilebilirlik ve Dayanıklılık:** Verileri birden fazla düğümde (node) replike ederek, SPOF durumunda bile veri kaybının önlenmesini ve sürekli hizmet sunulmasını sağlar.

Bu sistemler, aynı zamanda dağıtık mimarileri destekler. Bu nedenle "Dağıtılmış (Distributed) SQL" olarak da sınıflandırılmaktadırlar.

# Dağıtılmış SQL

Dağıtılmış SQL, veri tabanlarının dağıtık bir mimaride çalışmasını ve verilerin birden fazla fiziksel lokasyonda saklanmasını sağlayan SQL tabanlı sistemleri tanımlamaktadır. Bu sistemlerin ana amacı, transaction'ların birden fazla sunucu veya düğüm arasında dağıtılmasını sağlayarak yüksek ölçeklenebilirlik ve dayanıklılık sunmaktır.

### Tablo 1. Dağıtık ve Geleneksel Mimari Karşılaştırılması

| Özellik / Mimari | Dağıtık Mimari | Geleneksel Merkezi Mimari |
|---|---|---|
| **Veri Saklama Yöntemi** | Verileri birden çok sunucu veya veri düğümü üzerinde dağıtık bir şekilde saklar. Veriler, farklı sunucular arasında parçalara bölünür. | Tek bir merkezi sunucu üzerinde çalışır. Bütün veriler bu merkezi sunucuda saklanır ve yönetilir. Bu sunucu, tüm sorgu işleme görevlerini gerçekleştirir. |
| **Ölçeklenebilirlik** | Kolayca ölçeklenebilirler. Yeni sunucular eklemek, sistem performansını artırmak için basittir. | Sunucu kaynakları belli bir kapasite ile sınırlıdır. İhtiyaç halinde kaynakları artırmak maliyetleri yükseltir. |
| **Yüksek Erişilebilirlik** | Bir sunucunun arızalanması durumunda diğer sunucular devreye girer. Bu sayede, hizmette kesinti yaşanmaz. | Tek bir merkezi sunucu üzerinde çalıştığı için, sunucunun arızalanması durumunda kesintiler oluşabilir. |
| **Veri Erişimi ve Performans** | Sorgular daha paralel bir şekilde işlenebilir. Bu sayede, daha yüksek performans sağlar. | Sorgular tek bir sunucuya yönlendirilir. Bu, yüksek trafikte sorgu gecikmelerine neden olur. |
| **Coğrafi Dağıtım** | Veriler farklı coğrafi bölgelere dağıtılabilir. Bu sayede, düşük gecikme süresi sağlar. | Veriler genellikle tek bir konumda bulunur ve coğrafi olarak dağıtılamaz. |

## 1. CAP Teoremi

CAP Teoremi, dağıtılmış sistemlerin temel sınırlamalarını tanımlamaktadır. CAP, Tutarlılık (Consistency), Erişilebilirlik (Availability) ve Bölünme Toleransı (Partition Tolerance) kelimelerinin baş harflerinden oluşmaktadır. Teorem ise dağıtılmış veri tabanının, CAP'i oluşturan bu üç özellikten yalnızca **ikisini aynı anda tam** olarak sağlayabileceğini belirtmektedir.


*Şekil 1. CAP Teoremi*

1. **Tutarlılık:** Her düğümde aynı verinin görüntülenebilmesidir.
2. **Erişilebilirlik:** Her istekte yanıt alınabilmesidir.
3. **Bölünme Toleransı:** Sistemin, bazı iletişim kesintilerine (ağ bölünmelerine) rağmen çalışmaya devam edebilmesidir.

Bu bağlamda veri tabanları genelde **CA, CP** ve **AP** olmak üzere 3 gruba ayrılmaktadır. Her grup, CAP teoreminin iki özelliğine odaklanmakta ve bunun sonucunda diğer üçüncü olan özellikten ödün vermektedir.

## 1.1 CA

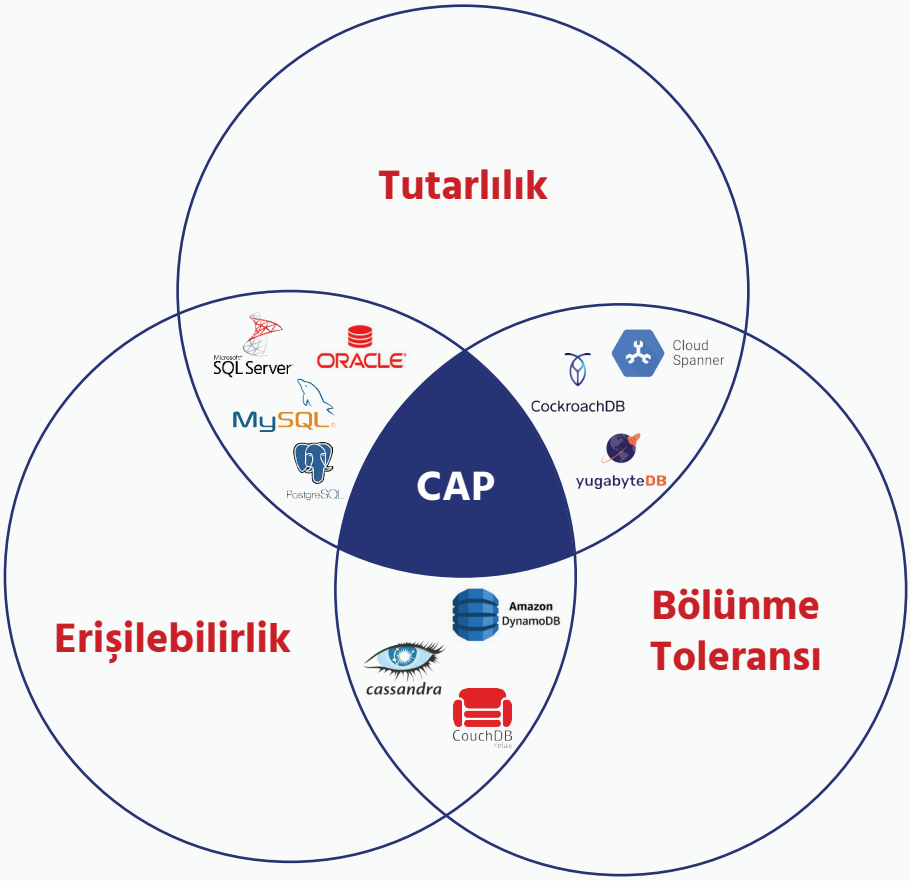
- Bu sistemler, veri tutarlılığı ve erişilebilirlik sağlarlar. Yani her sorgu için güvenilir bir cevap verirler ve veri hep güncel tutulur.
- Bölünme toleransına sahip değildir. Ağ kesintisi durumunda, sistem çökebilir.
- Microsoft SQL Server, Oracle, MySQL ve PostgreSQL örnek olarak verilebilir.

## 1.2 AP

- Bu sistemler, her zaman erişilebilir ve ağ bölünmelerine karşı dayanıklıdır. Yani herhangi bir ağ kesintisinde bile sisteme erişim mümkündür ve sistem isteklere yanıt vermeye devam eder.
- Tutarlılık bazen göz ardı edilir. Yani tüm düğümlerde her zaman en güncel veri olmayabilir ve veri tutarsızlıkları meydana gelebilir. Bunlar genellikle “nihai tutarlılık” (eventual consistency) ile çözülür.
- CouchDB, Apache Cassandra ve Amazon DynamoDB örnek olarak verilebilir.

## 1.3 CP

- Bu sistemler, ağ bölünmelerine karşı dayanıklıdır ve tutarlılık sağlarlar. Yani ağ kesintisi durumunda bile çalışmaya devam edebilir ve veri tutarlılığını koruyabilirler.
- Ağ bölünmeleri durumunda, AP sistemlerinden farklı olarak, tutarlılığı ön planda tutarlar. Bu nedenle erişilebilirlik bazen göz ardı edilebilir ve sistem, tüm isteklere hızlı bir şekilde yanıt veremeyebilir.
- Google Cloud Spanner, CockroachDB ve YugabyteDB örnek olarak verilebilir.




*Şekil 2. Veri Tabanları ve CAP Teoremi*

## 2. CAP ve CockroachDB

CockroachDB, tutarlı ve bölünme toleranslı (CP) bir sistemdir. Bu kapsamda ağ bölünmesi durumunda, tutarsız sonuçlara yol açabilecek herhangi bir işlem yerine sistemi kullanılamaz hale getirmeyi tercih etmektedir. CockroachDB, HA özelliği de taşımaktadır. HA, CAP teoremindeki erişilebilirlikten farklıdır.

CAP teoreminde erişilebilirlik kesin bir yargı içermektedir; sisteme erişim ya vardır ya da yoktur. HA'da ise erişilebilirlik yüzdelik (ör. %99.999 erişilebilirlik) olarak ifade edilmektedir. CockroachDB'nin CP ve HA olması; sunucuların yarısından fazlası birbirleriyle iletişim kurabiliyorsa, işlemlerin devam edebileceği anlamına gelmektedir. Örneğin, üç veri merkezine dağıtılmış bir sistemde; bir bağlantı kesilirse, diğer iki veri merkezi birkaç saniyelik kesintiyle normal işlem yapmaya devam edebilmektedir.

# CockroachDB

CockroachDB, yüksek erişilebilirlik, ölçeklenebilirlik ve güvenilir veri tutarlılığı sağlamayı hedefleyen, dağıtılmış bir SQL veri tabanıdır. Sunduğu özellikler aşağıdaki şekilde sıralanabilir:

- **Yatay Ölçeklenebilirlik:** Kolayca düğüm/sunucu ekleyerek ve çıkararak ölçeklenebilir.
- **Yüksek Erişilebilirlik:** Sunucu arızalarına dayanıklı ve sürekli erişilebilir.
- **Merkezi Olmayan (Masterless) Yapı:** Tüm düğümleri eşittir, her biri okuma ve yazma işlemleri için kullanılabilir.
- **Küresel (Global) Dağılım:** Veriler coğrafi olarak farklı lokasyonlarda tutulabilir. Bu sayede verilere yakınlık prensibiyle düşük gecikme süresi sağlanabilir.
- **Otomatik Parçalama (Sharding):** Veri tabanı, verileri birden çok düğüm arasında otomatik olarak bölümlere (shard) ayırır. Bu, okuma ve yazma işlemlerini yüksek derecede ölçeklenebilir kılar.
- **Otomatik Yük Devretme (Failover):** Bir düğüm arızalanırsa, işlemler otomatik olarak diğer düğümlere yönlendirilir. Manuel müdahaleye gerek kalmaz.
- **ACID Uyumlu İşlemler:** Güçlü veri tutarlılığı sağlar ve veri bütünlüğünü korur.
- **SQL Tabanlı Sorgulama:** Standart SQL sorgularını destekler. Bu durum mevcut uygulamalar ve araçlar ile kolay entegrasyon sağlar.

CockroachDB'nin amacı, çok çeşitli donanım platformlarında çalışabilen, ölçeklenebilir, yüksek erişilebilirlikli, tutarlı, coğrafi olarak dağıtılmış, yüksek performanslı ve SQL destekli bir ilişkisel veri tabanı sistemi sağlamaktır. CockroachDB'nin mimarisi, bu amaçlara uygun olacak şekilde tasarlanmıştır. Bu mimari, bir veya birden fazla, bağımsız ve eşit düğümden oluşan bir küme (cluster) yapısını temel almaktadır. Kümedeki düğümler, dağıtılmış veri tabanı sisteminin tek bir mantıksal görünümünü sunmak için iş birliği yapmaktadır. Her biri kendi içinde SQL işleme, transaction yönetimi, veri replikasyonu, dağıtım ve depolama gibi temel veri tabanı hizmetlerini katmanlar halinde sağlamaktadır. CockroachDB, geliştiricilerin bu katmanlarda neler yapıldığını derinlemesine anlamalarına gerek kalmadan verimli bir şekilde çalışmalarına olanak tanımaktadır.

## 1. Küme Mimarisi

- CockroachDB, her biri kendi özel depolama alanına sahip bir veya daha fazla veri tabanı sunucusu sürecinden (process) oluşmaktadır. Bu dağıtım modelinde, tüm düğümler eşit ve bağımsızdır; yani hiçbir düğüm diğerlerinden özel veya birincil olarak ayrılmamaktadır.
- Depolama alanı genellikle CockroachDB sunucusunun yüklü olduğu makineye doğrudan bağlıdır, fakat veriler fiziksel olarak paylaşılan bir depolama sisteminde de bulunabilmektedir.

- Bu dağıtık mimarinin yüksek erişilebilirlik, ölçeklenebilirlik ve dayanıklılık sağlamasında rol oynayan temel yapı taşları aralık (range) ve replikalardır.

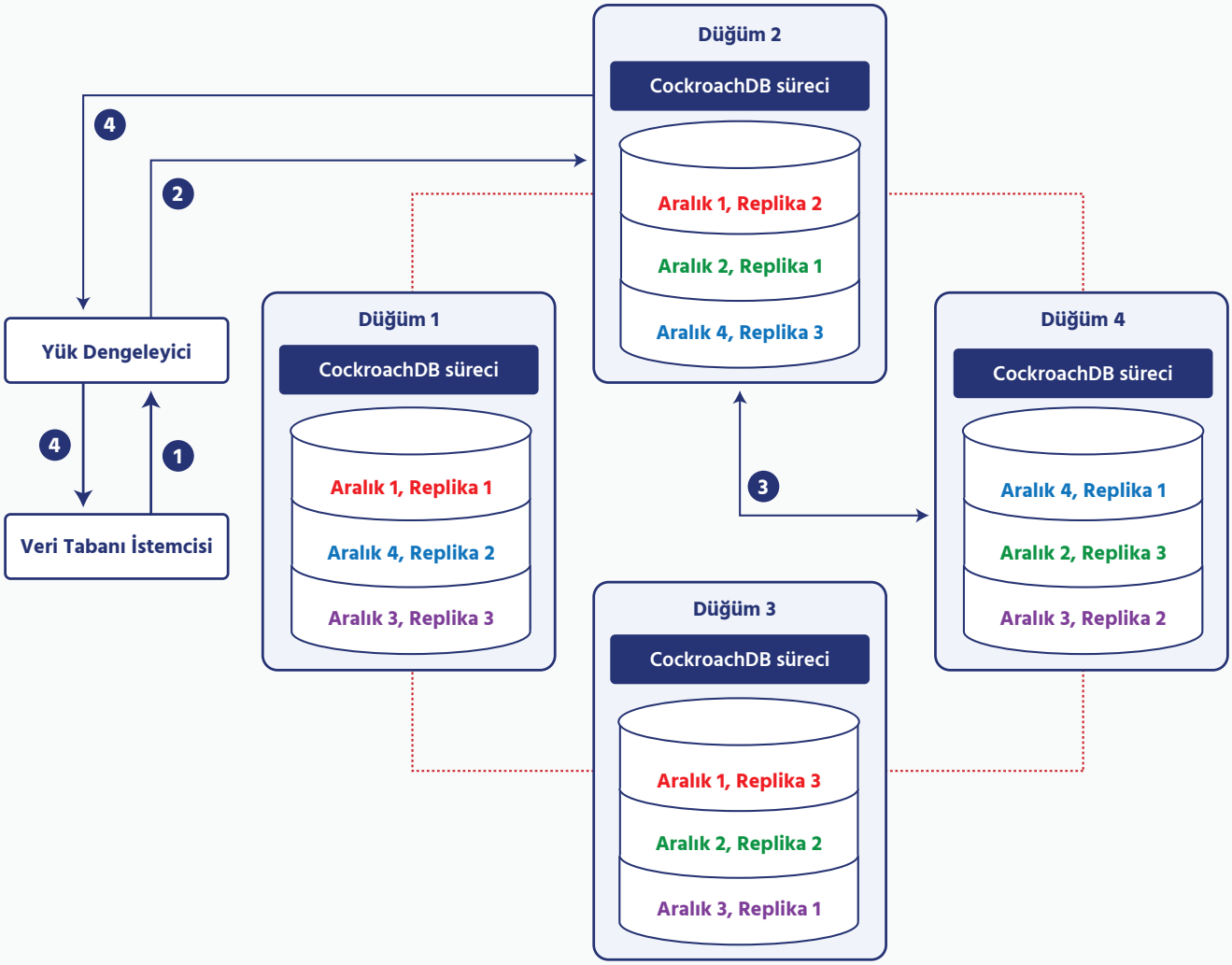
  **Aralık:** CockroachDB'deki bir veri aralığıdır. Tüm veri tabanı, belirli anahtar değerleri bazında daha küçük parçalara bölünmüş ve bu parçalar aralık olarak adlandırılmıştır. Her aralık, genellikle veri tabanındaki belirli bir anahtar setini içermekte ve bu anahtar seti üzerinde işlemler gerçekleştirmektedir. Aralıkların boyutu genellikle 512 MB'tır. Veri, aralıklar aracılığıyla küme boyunca fiziksel olarak dağıtılmakta ve her bir aralık, veri tutarlılığını ve erişilebilirliğini sağlamak için kümenin en az üç üyesine çoğaltılmaktadır. Bu yapı, veri sorgularının ve işlemlerinin farklı düğümler arasında paralel olarak gerçekleştirilmesine olanak tanımakta, bu da yüksek performans ve güvenilirlik sağlamaktadır.

  **Replika:** CockroachDB'deki bir aralığın kopyasıdır. Her aralık, kümenin farklı düğümlerinde bulunan birkaç replikaya sahiptir. Bu replikalar, aralığın verilerini saklamakta ve veri güvenliği ile yüksek erişilebilirlik sağlamaktadır. Replikalar, veri kaybını önlemek ve düğüm arızalarına karşı dayanıklılık sağlamak için kullanılmaktadır. Bir düğümdeki bir aralık arızalandığında veya erişilemez olduğunda, diğer düğümlerdeki replikalar devreye girerek veriye erişimi sürdürmekte ve transaction'ların sürekliliğini sağlamaktadır. CockroachDB, raft konsensüs algoritmasını kullanarak replikalar arasında veri tutarlılığını yönetmekte ve her bir aralık için bir lider replika seçmektedir. Bu lider replika, yazma işlemlerini kabul etmekte ve değişiklikleri diğer replikalara yayarak veri bütünlüğünü korumaktadır.

- Uygulamalar, yönetim konsolları ve CockroachDB kabuğu gibi veri tabanı istemcileri, küme içindeki bir CockroachDB sunucusuna bağlanmaktadır. Veri tabanı sunucusu ile istemci arasındaki iletişim, PostgreSQL wire protokolü üzerinden gerçekleşmektedir. Bu protokol, SQL istekleri ve yanıtlarının PostgreSQL istemcisi ile sunucusu arasında nasıl aktarıldığını tanımlamaktadır. CockroachDB, PostgreSQL wire protokolünü kullandığı için, herhangi bir PostgreSQL sürücüsü (driver) CockroachDB sunucusu ile iletişim kurmak amacıyla kullanılabilmektedir.

- Daha karmaşık dağıtımlarda, yük dengeleyiciler (load balancer) bağlantıları düğümler arasında eşit ve akılcı bir şekilde dağıtmakla sorumludur. Yük dengeleyici, istemciyi küme içindeki uygun bir düğüme bağlayarak ağ geçidi sunucusu (gateway server) görevi görmektedir.

- İstemci tarafından yapılan bir istek, küme içindeki tek bir düğümde veya birden fazla düğümde veri okuma ve yazma işlemlerini içerebilmektedir. Belirli bir KV aralığı için bir leaseholder (kiracı) düğüm, o aralığa yönelik okuma ve yazma işlemlerini kontrol etme görevini üstlenmektedir. Leaseholder düğüm, genellikle raft lideridir ve veri replikalarının doğru şekilde yönetilmesinden sorumludur.




*Şekil 3. CockroachDB Küme Mimarisi*

Şekil 3'te bu kavramlardan bazıları gösterilmektedir:

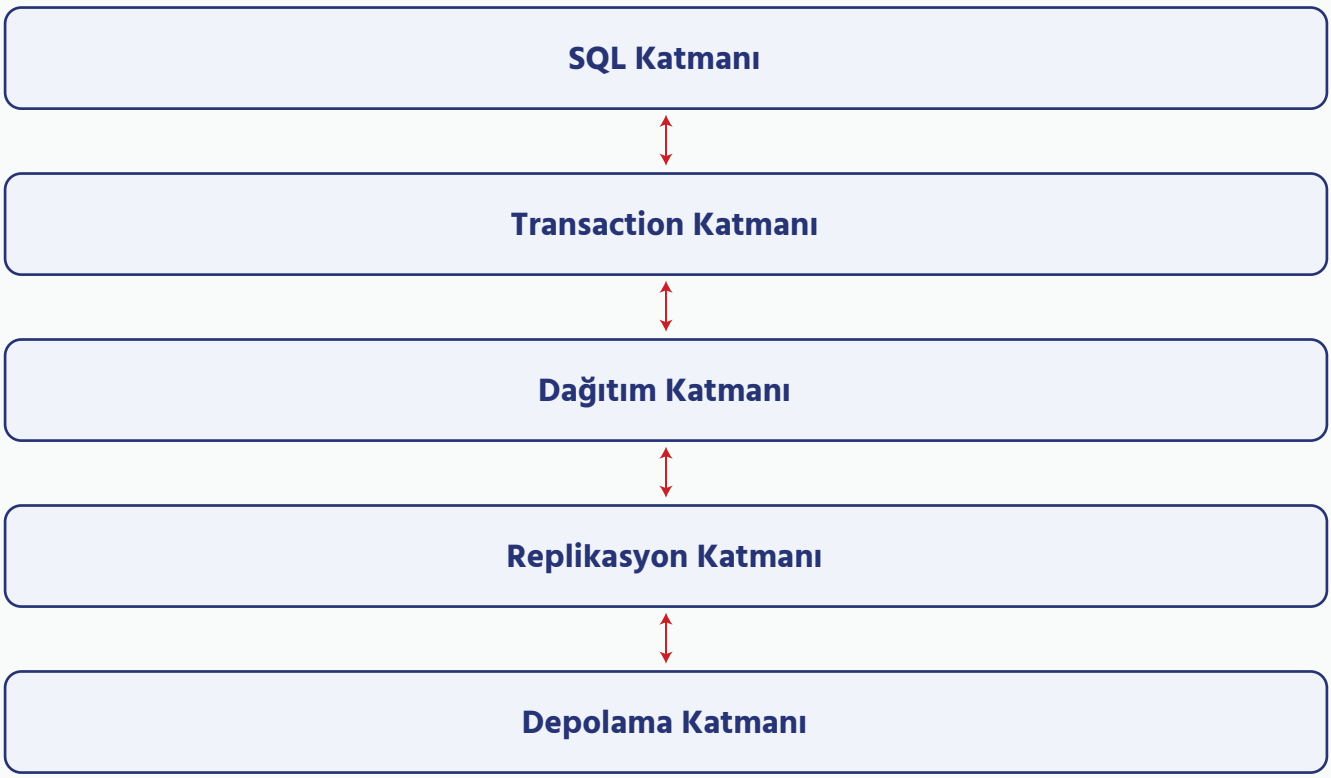
1. Bir veri tabanı istemcisi, CockroachDB kümesi için proxy (vekil sunucu) görevi gören bir yük dengeleyiciye bağlanır.
2. Yük dengeleyici, istekleri kullanılabilir bir CockroachDB düğümüne yönlendirir.
3. Bu düğüm, bu bağlantının ağ geçidi düğümü olur. İstekte, aralık 4'teki veriler talep edildiği için ağ geçidi düğümü, bu aralık için leaseholder olan düğümle iletişim kurar; leaseholder, verileri ağ geçidine döndürür.
4. Ağ geçidi düğüm de gerekli verileri yük dengeliyicisi aracılığıyla veri tabanı istemcisine döndürür.

Bu mimari, yükü kümenin tüm düğümleri arasında eşit bir şekilde dağıtmaktadır. Ağ geçidi görevleri, yük dengeleyici tarafından kümenin tüm düğümleri arasında adil bir biçimde paylaştırılmakta; leaseholder görevleri de, tüm düğümlerdeki veri aralıklarına göre benzer şekilde dağıtılmaktadır. Ayrıca bir sorgu birden fazla aralıktan veri gerektiriyorsa veya verilerin değiştirilmesi gerekiyorsa iş akışı daha fazla adım içermektedir.

## 2. Yazılım Katmanları

Her CockroachDB düğümü, çok iş parçacıklı tek bir süreç (multithreaded process) olan CockroachDB yazılımının bir kopyasını çalıştırmaktadır. Bu yazılım, veri tabanı işlevselliğini sağlamak için gerekli olan 5 mantıksal katmandan oluşmaktadır.

- **SQL katmanı,** PostgreSQL wire protokolündeki SQL isteklerini kabul etmektedir. SQL isteklerini ayrıştırıp optimize etmekte ve istekleri alt katmanlar tarafından işlenebilecek KV işlemlerine dönüştürmektedir.
- **Transaction katmanı,** tüm transaction'ların ACID ilkelerine göre yürütülmesini garanti etmekte, bu da transaction'ların güvenilir ve tutarlı olmasını sağlamaktadır. Ayrıca, transaction'ların serileştirilebilir izolasyonunu sağlayarak, eş zamanlı transaction'ların birbiri üzerinde beklenmeyen etkiler yaratmasını önlemektedir.
- **Dağıtım katmanı,** verilerin aralıklara bölünmesinden, bu aralıkların küme boyunca dağıtılmasından ve leaseholder'ların belirlenmesinden sorumludur.
- **Replikasyon katmanı,** bir düğüm arızası durumunda yüksek erişilebilirliğe olanak sağlamak için verilerin küme genelinde çoğaltılmasını sağlamaktadır. Ayrıca tüm düğümlerin herhangi bir veri öğesinin mevcut durumu üzerinde hemfikir olmasını sağlamak için raft algoritmasını kullanmaktadır.
- **Depolama katmanı,** verilerin yerel diskte kalıcılığından ve bu veriler üzerindeki düşük düzeyli sorguların (low-level query) ve güncellemelerin işlenmesinden sorumludur.




*Şekil 4. CockroachDB Yazılım Katmanları*

## 2.1 SQL Katmanı

PostgreSQL wire protokolü üzerinden gelen SQL isteklerini ele almaktan sorumlu katmandır. SQL katmanı iki önemli görev üstlenmektedir:

1. **Sorgu Kontrolü:**
   - **Syntax Doğruluğu:** SQL sorgusunun yazım hatası içermediğinden emin olur.
   - **Yetkilendirme:** Kullanıcının istediği işlemi yapmaya yetkisi olup olmadığını kontrol eder.
2. **Sorgu Çevirme:**
   - SQL sorgularını, KV'ye çevirir ve sorgular, diğer katmanlarda KV formatında kullanılır. Diğer katmanlar, SQL dilinden tamamen habersiz olur, istekleri daha kolay anlar ve işler. Böylece veri tabanı sorguları daha hızlı ve daha az kaynak kullanarak çalışır. KV'ye çevrilmesi CockroachDB'nin büyük veri kümeleriyle kolayca çalışmasını ve farklı türde veri kullanan uygulamalarla uyumlu olmasını sağlar.

Genellikle CockroachDB'nin tüm SQL sorgularında gerekli işlemi yapmasının birden fazla yolu vardır. Bu nedenle en iyi erişim yolunu belirlemek için optimizer kullanılmaktadır. CockroachDB'nin SQL optimizer'ı, SQL sorgusunun en hızlı ve en verimli şekilde çalıştırılmasını sağlamak için optimize edilmiş bir yürütme planı oluşturmaktadır.

CockroachDB, veri gruplarını işlemenin hızını artırmak için vektörleştirilmiş bir yürütme motorunu desteklemektedir. Bu motor; veriyi, her veri setinde aynı satırdaki verileri içeren satır odaklı formattan, her veri setinde aynı sütundan veri içeren sütun odaklı formata çevirerek çalışmaktadır. Büyük veri kümelerini işlemek için ideal bir araçtır. Hız, verimlilik ve ölçeklenebilirlik sunarak karmaşık sorguların ve analitik işlemlerin daha hızlı ve daha az kaynak kullanarak çalışmasını sağlamaktadır.

### 2.1.1 KV Depolama Sistemindeki Tablolar

KV depolama sistemindeki her değer benzersiz bir anahtarla eşleştirilmektedir. Bu anahtar, veriyi yerleştirmek ve veriye erişmek için kullanılmaktadır. Geleneksel tablo yapısını bir KV depolama sisteminde simüle etmek için bu tür bir anahtar düzeni kullanılmaktadır. Bu düzen, SQL sorgularının KV depolama sistemine nasıl yansıtılabileceğini anlamaya yaramaktadır. Her KV çifti, tablodaki bir satırın tüm verilerini veya belirli sütun verilerini içerebilmektedir. Anahtar şu şekilde tanımlanmaktadır:

**/<tabloID>/<indeksID>/<IndeksAnahtarDeğerleri>/<SütunAilesi>**

Varsayılan tabloID, tablo adı; varsayılan indeksID ise "primary"dir.

```
CREATE TABLE stok (
       id INT PRIMARY KEY,
       isim STRING,
       fiyat FLOAT
)
```

| ID | isim | fiyat |
|---|---|---|
| 1 | Bat | 1.11 |
| 2 | Ball | 2.22 |
| 3 | Glove | 3.33 |

| Anahtar | Deger |
|---|---|
| /stok/primary/1 | "Bat" , 1.11 |
| /stok/primary/2 | "Ball" , 2.22 |
| /stok/primary/3 | "Glove" , 3.33 |

*Şekil 5. KV Depolama Sistemindeki Tablo Yapısı*

Şekil 5'te, stok tablosuna ait verilerin bir KV veri yapısında nasıl saklanabileceği gösterilmektedir. Her satır için benzersiz bir anahtar oluşturulmakta ve bu anahtarların her biri ilgili stok ismi ve fiyatıyla ilişkilendirilmektedir.

Örneğin, ilişkisel veri tabanında ID'si 1 olan veriye erişmek için aşağıdaki gibi bir sorgu kullanılıyorken:

```
SELECT * FROM stok WHERE id = 1;
```

KV depolamada aynı bilgiye erişmek için, önceden tanımlanmış bir anahtar kullanılmaktadır:

**/stok/primary/1**

### 2.1.1.1 Sütun Aileleri

Bir tabloda sütunlar, genellikle birbirleriyle ilişkili veri grupları halinde düzenlenmektedir. Bu gruplara **sütun aileleri (column families)** denir. Tablodaki her sütun ailesi, ayrı bir KV girişi olarak saklanmaktadır. Bu durum, veri tabanı önbelleğinin daha verimli çalışmasını sağlamaktadır. Ayrıca veri tabanı üzerinde eş zamanlı güncellemeler yapılırken oluşabilecek çakışmaları önlemektedir.

```
CREATE TABLE insan(
       id INT PRIMARY KEY,
       isim VARCHAR(100) NOT NULL,
       soyIsim VARCHAR(100) NOT NULL,
       dogumGunu DATE NOT NULL,
       foto BLOB,
       FAMILY f1 (isim, soyIsim, dogumGunu),
       FAMILY f2 (foto)

);
```

| ID | isim | soyIsim | dogumGunu | foto |
|---|---|---|---|---|
| 1 | Fred | Codd | 26-AUG-2018 | \xff\xd8\xff\xe1#\<br>x05Exif\x00\x00II*\x00<br>... |

| Anahtar | Deger |
|---|---|
| /insan/primary/1/f1 | 'FRED' , 'CODD' , '26-AUG-2018' |
| /insan/primary/1/f2 | \xff\xd8\xff\xe1#\x05Exif\x00\x00II*\x00... |

*Şekil 6. Sütun Aileleri*

Şekil 6'daki gibi, bir kişinin adı, soyadı ve doğum tarihi gibi temel bilgileri bir sütun ailesinde; fotoğrafı gibi daha büyük veya daha az erişilen verileri ise başka bir sütun ailesinde saklanabilmektedir.

### 2.1.1.2 Benzersiz Olmayan İndeksler

- KV girişlerinin anahtarları, tablo adı, indeks adı, indeks değeri ve birincil anahtar değerini içermektedir.
- Değer boş bırakılır çünkü benzersiz olmayan indeksler (non-unique indexes) yalnızca hızlı erişim ve sıralama için kullanılır ve değerler sorgulama sırasında gereksizdir. Anahtarlar, yeterli bilgiyi sağlamakta ve satırlar birincil anahtarlar kullanılarak bulunmaktadır.

```
CREATE TABLE stok (
       id INT PRIMARY KEY,
       isim STRING,
       fiyat FLOAT
       INDEX isim_idx (isim)
)
```

| ID | isim | fiyat |
|---|---|---|
| 1 | Bat | 1.11 |
| 2 | Ball | 2.22 |
| 3 | Glove | 3.33 |
| 4 | Bat | 4.44 |

| Anahtar | Deger |
|---|---|
| /stok/isim_idx/"Bat"/1 | ø |
| /stok/isim_idx/"Ball"/2 | ø |
| /stok/isim_idx/"Glove"/3 | ø |
| /stok/isim_idx/"Bat"/4 | ø |

*Şekil 7. Benzersiz Olmayan İndeksler*

### 2.1.1.3 Benzersiz İndeksler

- KV girişlerinin anahtarları, tablo adı, indeks adı ve indeks değerini içermektedir. Her indeks değeri benzersizdir ve yalnızca bir satıra karşılık gelmektedir.
- Değer kısmında, bu anahtara karşılık gelen birincil anahtarın değeri bulunmaktadır. Bu durumda, değer gereklidir. Çünkü benzersiz indeksin (unique index) amacı, sadece hızlı erişim sağlamak değil, aynı zamanda o anahtarla ilişkili tam bir satırı tanımlamaktır. Benzersiz bir indeksin değer kısmına birincil anahtar değerini koymak, KV depolamasında tek bir sorguyla ilgili satıra hızlı erişim için yapılmaktadır. Sorgularken, önce indekse bakılmakta, buradan birincil anahtar değeri elde edilmekte ve doğrudan ilgili veri çekilmektedir. Bu, ek bir sorgu yapma ihtiyacını ortadan kaldırmakta ve veri erişimini hızlandırmaktadır.

```
CREATE TABLE stok (
       id INT PRIMARY KEY,
       isim STRING,
       fiyat FLOAT
       UNIQUE INDEX isim_idx (isim)
)
```

| ID | isim | fiyat |
|---|---|---|
| 1 | Bat Acme | 1.11 |
| 2 | Ball | 2.22 |
| 3 | Glove | 3.33 |
| 4 | Bat Sears | 4.44 |

| Anahtar | Deger |
|---|---|
| /stok/isim_idx/"Bat Acme" | /1 |
| /stok/isim_idx/"Ball" | /2 |
| /stok/isim_idx/"Glove" | /3 |
| /stok/isim_idx/"Bat Sears" | /4 |

*Şekil 8. Benzersiz İndeksler*

Şekil 8'deki gibi, "Bat Sears" için bir sorgu yapıldığında, KV depolama sistemi doğrudan /stok/isim_idx/"Bat Sears" anahtarını taramaktadır. Bu anahtarın birincil anahtar değeri olan /4, sorgulanan satırın tablodaki ID sütunundaki konumunu göstermektedir. Bu bilgi kullanılarak doğrudan ID'si 4 olan satıra ulaşılmakta ve ilgili tüm sütun verileri tek bir sorgu ile çekilmektedir. Bu yöntem, ürün ismine göre hızlı bir arama yapılmasını sağlamakta ve ayrı bir sorgu yaparak ID üzerinden veri çekme gerekliliğini ortadan kaldırmaktadır. Bu, özellikle büyük ve sık sorgulanan veri setleri için performansı önemli ölçüde artırmaktadır.

### 2.1.1.4 Ters Çevrilmiş İndeksler

Ters çevrilmiş indeksler (inverted indexes), dizilerde (array) veya JSON belgelerinde bulunan değerlere yönelik indeksli aramalara izin vermektedir. Bu durumda, anahtar değerler, Şekil 9'daki gibi, birincil anahtarla birlikte JSON yolunu ve değerini içermektedir.

```
CREATE TABLE stok (
       id INT PRIMARY KEY,
       veri JSONB,
       INVERTED INDEX veri_idx(veri)
)
```

| ID | isim |
|---|---|
| 1 | {"isim": "Bat", "fiyat": 1.11} |
| 2 | {"isim": "Ball", "fiyat": 2.22} |
| 3 | {"isim": "Glove", "fiyat": 3.33} |

| Anahtar | Deger |
|---|---|
| /stok/veri_idx/isim/Bat/1 | Ø |
| /stok/veri_idx/fiyat/1.11/1 | Ø |
| /stok/veri_idx/isim/Ball/2 | Ø |
| /stok/veri_idx/fiyat/2.22/2 | Ø |
| /stok/veri_idx/isim/Glove/3 | Ø |
| /stok/veri_idx/fiyat/3.33/3 | Ø |

*Şekil 9. Ters Çevrilmiş İndeksler*

Bu indeksler, her bir JSON belgesindeki benzersiz özniteliklerin bir listesini tutmaktadır. Bu özellik, kompleks JSON belgeleri içeren bir satırda, her biri farklı bir özniteliği işaret eden çok sayıda indeks girişi oluşturulmasına neden olmaktadır. Bu durum, ters çevrilmiş indekslerin, geleneksel indekslere kıyasla daha fazla yer kaplamasına ve daha yüksek bakım maliyetlerine yol açabilmektedir.

### 2.1.1.5 STORING İfadesi

STORING (Depolama) ifadesi, bir indeks tanımında kullanıldığında, belirtilen sütunların değerlerinin, indeksin anahtar değerleri ile birlikte KV deposunda saklanmasını ve sütun değerlerine erişmek için ana tabloya dönmeden sadece indeksi kullanarak sorgulamasını sağlamaktadır. STORING ifadesi, diske giriş/çıkış ve karmaşık JOIN işlemlerini azaltmakta, veri önbelleğini daha verimli kullanmakta ve çok kullanıcılı sistemlerde eş zamanlılık sorunlarını azaltmaktadır.

```
CREATE TABLE insan(
       id INT PRIMARY KEY,
       isim VARCHAR(100) NOT NULL,
       soyIsim VARCHAR(100) NOT NULL,
       dogumGunu DATE NOT NULL,
       telefonNo int not null,
       digerSutunlar blob,
       INDEX (isim, soyIsim, dogumGunu) STORING (telefonNo)
);
```

| ID | isim | soyIsim | dogumGunu | telefonNo | dogumGunu |
|---|---|---|---|---|---|
| 1 | Fred | Codd | 26-AUG-2018 | +1-033-333-3333 | …. . |

| Anahtar | Deger |
|---|---|
| /insan/indeksIsmi/Fred/Codd/26-Aug-1918/1 | +1-033-333-3333 |

*Şekil 10. STORING İfadesi*

Şekil 10'da isim ve doğum tarihine göre bir kişinin telefon numarasını ararken, bu bilgi doğrudan indeksten alınabilir, çünkü telefon numarası indekse STORING kullanılarak eklenmiştir.

### 2.1.1.6 Tablo Tanımları ve Şema Değişiklikleri

- Tablolara ilişkin şema tanımları, tablo tanımlayıcı (table descriptor) adı verilen özel bir anahtar alanda saklanmaktadır. Her tablo için benzersiz bir tablo tanımlayıcısı bulunmaktadır. Performansı artırmak amacıyla, bu benzersiz tablo tanımlayıcıları her düğümde çoğaltılmaktadır. Tablo tanımlayıcısı, SQL'in ayrıştırılması, optimize edilmesi ve bir tabloyla ilgili doğru KV işlemlerinin yapılması için kullanılmaktadır.
- Şema değişiklikleri, iş ihtiyaçlarının değişmesine yanıt olarak veri yapılarını güncellemek, performansı iyileştirmek ve veri bütünlüğünü korumak için gereklidir. Ayrıca veri tabanının ölçeklenmesini kolaylaştırmakta ve eş zamanlı işlemlerin verimli bir şekilde yönetilmesine olanak tanımaktadır.

## 2.2 Transaction Katmanı

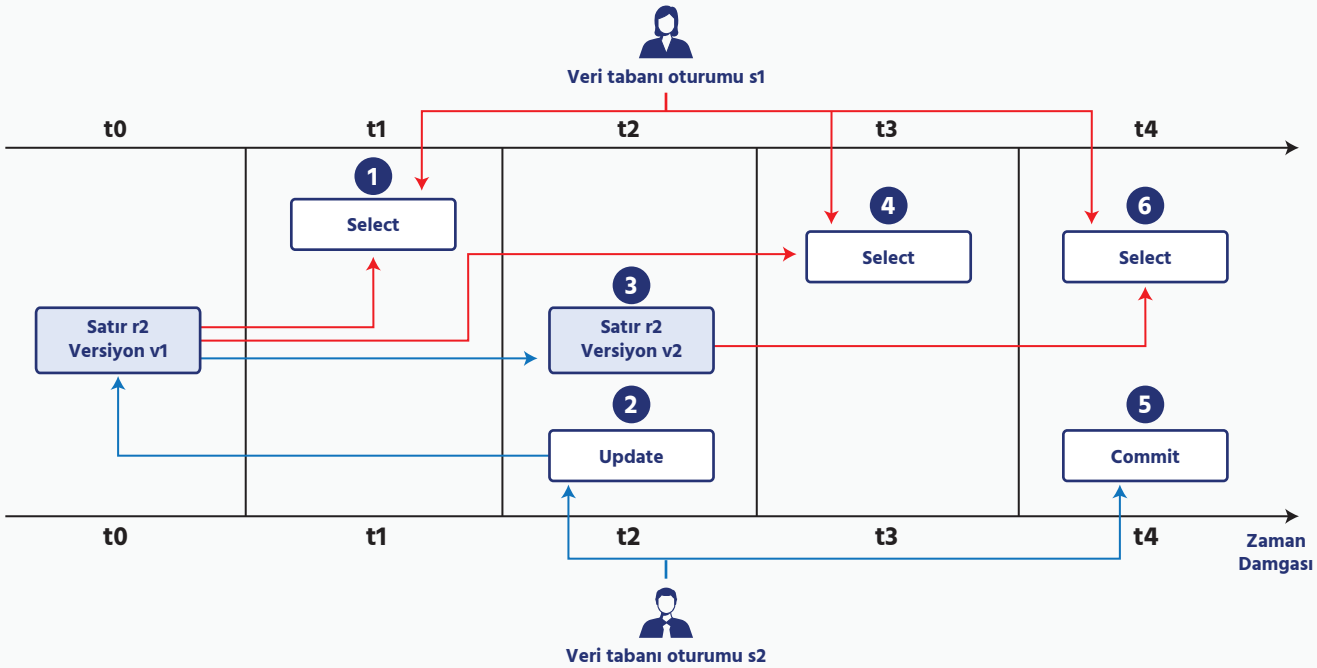
Transaction katmanı, bir transaction'daki tüm işlemlerin gerçekleştirilmesini (committed) veya iptal edilmesini (aborted) sağlayarak transaction'ların atomikliğini korumaktan sorumludur. Ek olarak, transaction katmanı transaction'lar arasında serileştirilebilir izolasyon sağlamaktadır. Bu da transaction'ların diğer transaction'ların etkilerinden tamamen izole edildiği anlamına gelmektedir. Aynı anda birden fazla transaction devam ediyor olsa da, her bir transaction'ın deneyimi, sanki transaction'lar teker teker yürütülüyormuş gibi serileştirilebilir izolasyon düzeyindedir.

Transaction katmanı, SQL katmanı tarafından üretilen KV işlemlerini işlemektedir. Bir transaction, bazıları tek bir SQL ifadesinin sonucu olabilen birden fazla KV işleminden oluşmaktadır. Tablo girişlerinin güncellenmesinin yanı sıra indeks girişlerinin de güncellenmesi gerekmektedir. Her koşulda mükemmel tutarlılığı korumak, birden fazla karmaşık algoritmayı gerektirir.

### 2.2.1 MVCC

Çoğu işlemsel veri tabanı sistemi gibi CockroachDB de MVCC modelini uygulamaktadır. MVCC, veriler başka bir transaction tarafından değiştirilirken dahi okuyucuların tutarlı bir bilgi görünümü elde etmelerine olanak tanımaktadır. Aynı zamanda MVCC, aynı veri üzerinde okuma ve yazma işlemlerinin birbirini bloke etmeden, yani kilit mekanizmalarına daha az bağlı kalarak gerçekleşmesini sağlamaktadır. Böylece, sistemler yüksek düzeyde eş zamanlı transaction performansı sunabilmektedir. MVCC'nin temel prensipleri aşağıdaki gibidir:

1. **Birden Fazla Versiyon:** Her veri öğesi için birden fazla versiyon tutulur. Bir veri öğesi güncellendiğinde, eski veriler silinmez; bunun yerine yeni bir versiyon oluşturulur. Böylece, farklı transaction'lar aynı anda verinin farklı versiyonlarını okuyabilir.
2. **Tutarlı Okuma:** Okuma işlemleri, verinin tutarlı bir görünümünü alır, yani bir transaction veriyi okurken başka bir transaction tarafından yapılan güncellemeleri görmez. Bu, verinin o anki transaction için dondurulmuş bir versiyonunu okumak gibi düşünülebilir.
3. **Yazma İşlemleri:** Bir veri öğesi üzerindeki güncelleme veya silme transaction'ı, o veri öğesinin yeni bir versiyonunu yaratır.
4. **Versiyon Zaman Damgaları (Timestamp):** Her versiyon, transaction'ın gerçekleştiği zamana ilişkin bir zaman damgası ile ilişkilendirilir. Bu, veri tabanının hangi versiyonlarının bir transaction tarafından görülebileceğini belirlemesine yardımcı olur.




*Şekil 11. MVCC*

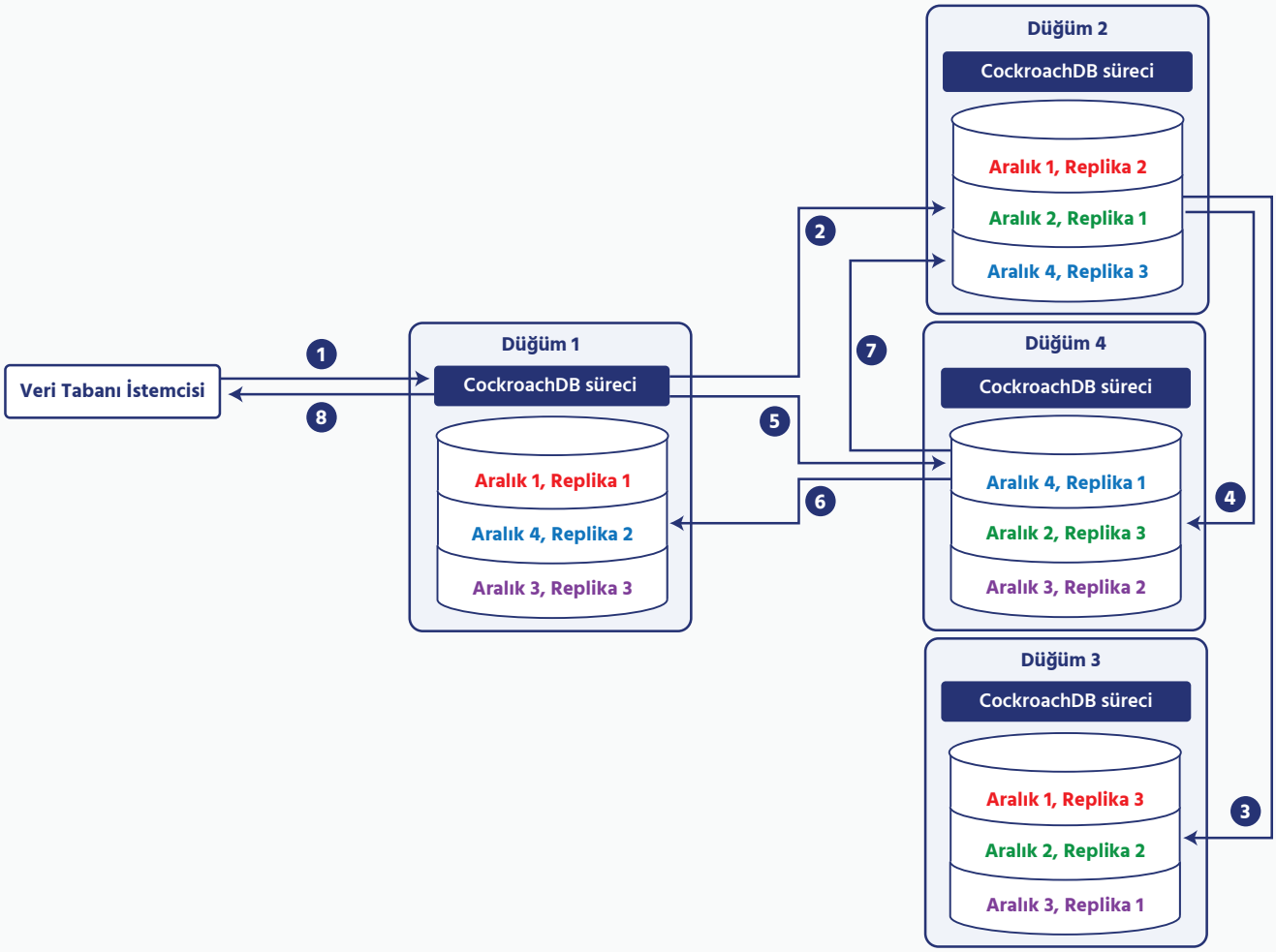
Şekil 11, MVCC'nin temel prensiplerini göstermektedir:

1. t1 zamanında, oturum s1, r2 satırından okuma yapar ve bu satırın v1 versiyonuna erişir.
2. t2 zamanında, başka bir veri tabanı oturumu olan s2, aynı satırı günceller.
3. Böylece satırın v2 versiyonu oluşur.
4. t3 zamanında, s1 satırı tekrar okur, ancak s2 henüz değişikliği onaylamadığı için v1 versiyonundan okumaya devam eder.
5. s2 değişiklikleri onaylar.
6. t4 zamanında, s1 yeni bir okuma yapar ve artık satırın v2 versiyonuna erişir.

MVCC'nin uygulanması, veri tabanı sistemlerinin karmaşık ve çok kullanıcılı ortamlarda bile yüksek performans ve tutarlılık sunmasını sağlamaktadır. CockroachDB gibi dağıtık sistemlerde, MVCC'nin sağladığı bu avantajlar, sistemlerin ölçeklenebilirliğini ve dayanıklılığını önemli ölçüde artırmaktadır.

## 2.2.2 Transaction İş Akışı

- Dağıtılmış sistemlerdeki bir transaction'ın tamamlanabilmesi için her bir düğümün transaction'ın temelini oluşturması ve transaction'ı gerçekleştirebileceğini bildirmesi gerekmektedir.




*Şekil 12. Temel Transaction Akışı*

Şekil 12, transaction hazırlığının oldukça basitleştirilmiş bir akışını göstermektedir:

**1.** İki ifadeden (statement) oluşan bir transaction CockroachDB ağ geçidi düğümüne gönderilir.

**2.** İlk ifade, aralık 2'deki veriye ait bir değişikliği içerdiği için bu istek, o aralığın leaseholder'ına (düğüm 2) gönderilir.

**3** ve **4.** Bu düğüm yeni bir geçici satır sürümü oluşturur ve değişiklikleri replika düğümlere yayar.

**5.** İkinci ifade, 4 numaralı aralığı etkilediği için istek, bu aralığın ilk replikasını bulunduran leaseholder'a (düğüm 4) gönderilir.

**6** ve **7.** Bu değişiklik de aralık 4'ü bulunduran diğer düğümlere yayılır.

**8.** Tüm değişiklikler doğru bir şekilde yayıldığında, transaction tamamlanır ve istemci transaction'ın başarılı olduğu konusunda bilgilendirilir.

## Yazma Niyeti (Write Intents)

Yazma niyeti, bir transaction'ın ilk adımlarında ortaya çıkmakta ve transaction'ın sonuçlarını geçici olarak saklamaktadır. Böylece transaction'ın ileride başarılı olup olmayacağının henüz bilinmediği durumlarda, değişikliklerin geçici bir ön izlemesini sağlamaktadır.

Yazma niyetleri, aynı veri kaydı üzerinde eş zamanlı güncelleme girişimlerini engelleyerek, veri tutarlılığını korumaktadır. Bu, transaction'ın kilitlenmesine benzer, ancak MVCC modelinde gerçekleşmekte, böylece veri tabanı performansı ve eş zamanlılık kapasitesi artmaktadır.

## Transaction Kaydı

Transaction kaydı (record), bir transaction'ın tam ve kesin durumunu kaydetmektedir. CockroachDB gibi bazı dağıtık veri tabanı sistemlerinde, transaction tarafından değiştirilen ilk aralık içinde bir kayıt oluşturulur. Bu kayıt, transaction'ın tüm detaylarını ve son durumunu içermekte ve transaction'ın hangi aşamada olduğunu belirten bir durum bilgisi taşımaktadır. Örneğin, Şekil 12'de, bu transaction kaydı, transaction'da ilk olarak değiştirilen 2 numaralı aralıkta saklanmaktadır.

Transaction durumu aşağıdakilerden biri olarak kaydedilmektedir:

- **PENDING (Beklemede):** Transaction henüz devam etmektedir ve tamamlanmamıştır.
- **STAGING (Hazırlık Aşamasında):** Transaction'la ilgili tüm yazma işlemleri tamamlanmıştır fakat transaction'ın başarıyla bitip bitmeyeceği garanti edilmemiştir. Bu aşama, transaction'ın son kontrolünün yapıldığı ve tamamlanmak üzere olduğu anlamına gelmektedir.
- **COMMITTED (Onaylandı):** Transaction başarılı bir şekilde tamamlanmıştır. Bu, tüm değişikliklerin kalıcı olarak veri tabanına yazıldığını ve transaction'ın başarılı olduğunu göstermektedir.
- **ABORTED (İptal Edildi):** Transaction iptal edilmiştir. Bu durumda, transaction sırasında yapılan tüm değişiklikler geri alınmakta ve veri tabanındaki verilerin önceki durumuna dönülmektedir.

### Ağ Geçidi Düğümü (Gateway Node)

Transaction'ı başlatan ve yöneten düğümdür. İstemci tarafından gönderilen bir transaction isteğini alan ve bu transaction sürecini koordine eden düğümdür. Transaction sırasında, gerekli tüm veri okuma ve yazma işlemlerini yönlendirmekte, diğer düğümlerle iletişim kurmakta ve transaction'ın başarılı bir şekilde tamamlanmasını sağlamaktadır.

## 2.2.3 Paralel Commit

- CockroachDB, dağıtılmış transaction'larda geleneksel yöntemlerin neden olduğu gecikmeleri azaltmak için "Paralel Commit" adında yenilikçi bir protokol kullanmaktadır. Geleneksel commit işlemleri, transaction sırasında oluşturulan yazma niyetlerini tek tek işlemekte ve her biri başarıyla tamamlandıktan sonra transaction kaydını güncellemektedir. Özellikle çok sayıda yazma niyeti bulunan durumlarda, bu süreç transaction'ın tamamlanma süresini uzatabilmektedir. Genellikle, dağıtılmış bir transaction'ın gerçekleştirilmesi için ağ üzerinden en az iki tur yapılması gerekmekte; bu sürece klasik algoritmalarda "İki Aşamalı Commit (Two-Phase Commit)" denmektedir.
- Paralel commit protokolünün temel amacı, ağ gidiş-dönüş sürelerinin transaction gecikmesi üzerindeki etkisini azaltmaktadır. Bu protokolle, bir transaction'ın iptal edilmesi artık mümkün olmayan bir noktaya geldiğinde, transaction henüz tam anlamıyla onaylanmamış olsa bile, ağ geçidi, transaction'ın başarılı olduğunu istemciye hızlı bir şekilde bildirebilmektedir. Bu yaklaşım, istemciye yanıt verme süresini önemli ölçüde kısaltmaktadır. Çünkü CockroachDB, transaction'ın bir kısmını istemciden gizli tutarak arka planda tamamlamaktadır. Böylece, istemci tarafından algılanan gecikme, dağıtılmış sistemlerdeki en büyük engellerden biri olan ağ gidiş-dönüş sürelerinin etkisini azaltmaktadır.

Paralel commit işlemi aşağıdaki adımları içermektedir:

1. **Transaction Hazırlığı ve STAGING Durumuna Geçiş:**
   - Transaction sırasında, son yazma işlemleri gerçekleştirilirken transaction kaydı STAGING durumuna geçer. Bu durum, transaction'ın geri alınamaz bir noktaya ulaştığını, ancak henüz tam olarak onaylanmadığını gösterir.
   - STAGING durumunda, transaction'ın gerçekleştirdiği tüm yazma işlemlerinin anahtarları transaction kaydında tutulur. Bu, transaction'ın hangi veri öğeleri üzerinde değişiklik yaptığını belirlemek için kullanılır.
2. **Erken Başarı Bildirimi:**
   - Ağ geçidi düğümü, transaction'ın artık geri alınamayacağından emin olduğunda, istemciye başarı bildirimi gönderir.

- Bu erken bildirim, istemcinin beklemesini gerektiren ağ gecikmesini azaltır ve istemci için transaction'ın tamamlanma süresini önemli ölçüde kısaltır.

**3. Arka Plan Çözümlemesi:**

- İstemciye başarı bildirimi gönderildikten sonra, ağ geçidi düğümü, transaction'ın son çözümlemesini arka planda başlatır ve yazma işlemlerinin durumunu kontrol eder. Bu aşama, transaction'ın nihai olarak onaylanıp onaylanmayacağının belirlendiği aşamadır.

**4. STAGING Durumundaki Transaction'ların Çözümlenmesi:**

- Eğer orijinal ağ geçidi düğümü başarısız olursa, STAGING durumundaki transaction kaydıyla karşılaşan başka bir düğüm, transaction'ın onaylanıp onaylanmayacağını belirlemek için her bir yazma işleminin durumunu sorgular.
- Transaction kaydının ve her bir yazma niyetinin kaydedilmiş olması sayesinde, transaction orijinal ağ geçidi düğümü tarafından veya başka bir düğüm tarafından çözümlense de sonuç aynı olur.

**5. Kilitlerin Serbest Bırakılması ve Transaction'ın Tamamlanması:**

- Transaction tamamen çözümlendikten sonra, transaction tarafından tutulan kilitler serbest bırakılır ve transaction kaydı COMMITTED veya ABORTED durumuna geçer. Bu, transaction'ın tamamlandığını ve veri tabanı üzerindeki etkilerinin kalıcı hale geldiğini gösterir.
- Bu aşama, mevcut transaction'ın diğer transaction'lar üzerindeki etkilerinin tamamen uygulandığını ve veri tabanının tutarlı bir durumda olduğunu gösterir.

### 2.2.4 Transaction Temizleme

CockroachDB'deki transaction temizleme süreci, bir transaction'ın onaylanmasından sonra gerçekleşmektedir. Bu süreç, transaction sırasında oluşturulan ve veri tabanındaki değişiklikleri geçici olarak belirten yazma niyetlerini, gerçek MVCC kayıtlarına dönüştürerek veri tabanındaki değişiklikleri kalıcı hale getirmekle ilgilidir.

Transaction temizleme işlemi genellikle aşağıdaki adımları içermektedir:

**1. Commit ve Transaction Kaydının Güncellenmesi:**

- Bir transaction başarıyla onaylanması transaction'ın tamamlandığını ve artık geri alınamayacağını gösterir. CockroachDB, bu durumu transaction kaydında bir değişiklik yaparak işaretler, yani transaction'ın durumunu "COMMITTED" olarak günceller. Bu işlem, transaction'ın başarılı olduğunu ve değişikliklerin kalıcı hale getirileceğini belirtir.

**2. Yazma Niyetlerinin Dönüştürülmesi ve Asenkron Sonlandırma:**

- Transaction onaylandıktan sonra, yazma niyetlerini MVCC kayıtlarına dönüştürme işlemi

başlar. Bu dönüşüm, yazma niyetlerinin artık kalıcı veri değişiklikleri olarak veri tabanında saklandığı anlamına gelir. Bu işlem, asenkron bir şekilde gerçekleşir, yani onaylama işleminden hemen sonra başlar ancak bir miktar zaman alabilir.

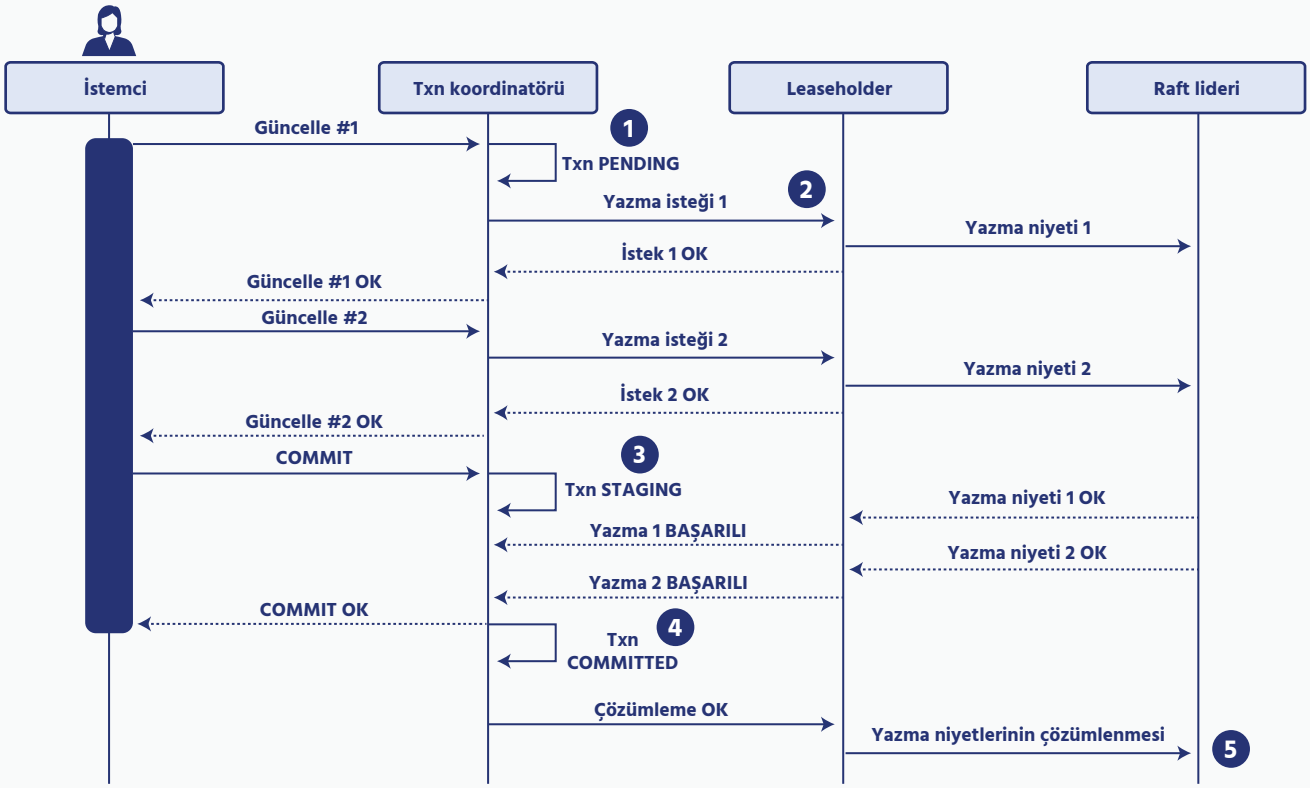
**3. Diğer Transaction'ların Yazma Niyetleriyle Karşılaşması:**

- Herhangi bir asenkron işlemde olduğu gibi, bu sonlandırma işleminde de gecikmeler olabilir. Diğer transaction'lar, temizlenmemiş yazma niyetleriyle karşılaşabilir.
- Bir transaction, henüz temizlenmemiş bir yazma niyeti ile karşılaşırsa, bu transaction, yazma niyetinin durumunu belirlemek için ilgili transaction kaydını kontrol eder. Eğer yazma niyeti "COMMITTED" olarak işaretlenmişse, bu yazma niyeti güvenli bir şekilde kalıcı veri kaydına dönüştürülebilir.

**4. Yazma Niyetlerinin Temizlenmesi:**

- Yazma niyetlerinin temizlenmesi, veri tabanındaki geçici işaretlemelerin kaldırılması anlamına gelir, transaction'ın başarıyla tamamlandığını ve veri tabanı üzerindeki etkilerinin kalıcı hale geldiğini garanti eder. Bu, CockroachDB'nin eş zamanlılık kontrol mekanizmalarının önemli bir parçasıdır ve sistemdeki veri tutarlılığını sağlar.

## 2.2.5 Transaction'ın Gerçekleşme Adımları




*Şekil 13. Transaction'ın Gerçekleşme Adımları*

Şekil 13, iki adımlı bir transaction'ın başarıyla tamamlanma akışını göstermektedir:

1. **Transaction PENDING:**
   - İstemci, bir güncelleme komutu gönderir. Bu komut ile, transaction koordinatörü tetiklenir.
   - Transaction koordinatörü, transaction durumunu "PENDING" olarak işaretler ve istekteki veriye ait aralık için leaseholder'a bir yazma isteği gönderir.
2. **Yazma İstekleri ve Yazma Niyetleri:**
   - Leaseholder, yazma isteğini alır ve Raft liderine yazma niyetini gönderir. Yazma niyetlerinin tamamlanması beklenmeden transaction koordinatörüne başarı mesajı dönülür.
   - İkinci güncelleme komutu için de işlemler tekrarlanır.
3. **Transaction STAGING:**
   - İstemci tüm güncellemelerin onayını aldıktan sonra "COMMIT" komutunu gönderir.
   - Transaction koordinatörü, transaction durumunu "STAGING" olarak işaretler ve yazma işlemlerinin başarıyla tamamlandığını belirten geri bildirimleri bekler.
4. **Transaction COMMITTED:**
   - Raft lideri, yazma niyetlerini başarıyla tamamlar ve başarı mesajını leaseholder'a gönderir. Leaseholder, transaction koordinatörüne işlemin başarıyla tamamlandığını bildirir. Transaction koordinatörü de, transaction durumunu "COMMITTED" olarak günceller.
   - Sonrasında leaseholder, istemciye transaction'ın onaylandığını bildirir.
5. **Yazma Niyetlerinin Çözülmesi (Resolve Write Intents):**
   - Transaction onaylandıktan sonra, leaseholder yazma niyetlerini çözme işlemini başlatır.
   - Bu adım, transaction'ın tamamlandığını, yani yazma niyetlerinin MVCC kullanılarak kalıcı veri kayıtlarına dönüştürüldüğünü ve veri tabanındaki değişikliklerin kalıcı hale getirildiğini belirtir.

Bu işlem sırasında, müşteriye gecikme süresi olarak yalnızca ilk iki yazma isteğinin tamamlanma süresi yansımaktadır. Yazma niyetlerinin çözülmesi ve transaction'ın veri tabanında tam olarak onaylanması, müşteriye başarılı bir yanıt gönderildikten sonra arka planda gerçekleşmektedir. Bu, işlem süresini önemli ölçüde kısaltmakta ve müşterinin veri tabanıyla etkileşimini hızlandırmaktadır.

### 2.2.6 Çatışmalar

Birkaç durum dışında genellikle eş zamanlı transaction'lar çözülmesi gereken çatışmalar (conflicts) yaratmaktadır. Karşılaşılabilecek çatışmalar aşağıdaki şekilde ele alınmaktadır:

1. **Yazma Çatışmaları:**
   - İki transaction, aynı veri kaydını güncellemeye çalıştığında çatışma oluşur.
   - Eş zamanlı yazma niyetleri nedeniyle bir kayda sadece bir transaction işlem yapabilir; diğer transaction ya beklemeli ya da iptal edilmelidir.
   - Transaction'lar aynı önceliğe sahipse, yazma niyeti oluşturmayan ikinci transaction bekler.
   - İkinci transaction'ın önceliği daha yüksekse, ilk transaction iptal edilir ve yeniden denemek zorunda kalır.
2. **Transaction Öncelikleri:**
   - Transaction öncelikleri, **SET TRANSACTION** komutu ile ayarlanabilir ve çatışmaların çözümünde önemli bir rol oynar.
3. **TxnWaitQueue Mekanizması:**
   - Bu mekanizma, transaction'ların neyi ve kimi beklediğini izler.
   - Her transaction için bekleyen işlemler ve bu işlemlerin durumu, raft liderinin içinde bulunduğu aralıkta tutulur.
   - Bir transaction tamamlandığında veya iptal edildiğinde, TxnWaitQueue güncellenir ve bekleyen transaction'lar bilgilendirilir.
4. **Kilitlenme (Deadlock) Önleme:**
   - İki transaction birbirlerinin yazma niyetlerini bekliyorsa kilitlenme oluşabilir.
   - Bu durumda, transaction'lardan biri rastgele iptal edilerek kilitlenme çözülür.
5. **Okuma/Yazma Çatışmaları:**
   - Bir okuyucu transaction, onaylanmamış bir yazma niyetiyle karşılaşırsa, beklemesi gerekebilir.
   - Okuma işlemi, yazma transaction'ı tamamlanana veya iptal edilene kadar duraklatılabilir.
6. **Okumaların Engellenmesini Önleme:**
   - Eğer okuyucu transaction'ın önceliği yüksekse, CockroachDB, yazma transaction'ının zaman damgasını ileri bir zamana iter. Yani, yazma işleminin daha geç bir zamanda gerçekleşmiş gibi davranmasını sağlar ve okumanın tamamlanmasına izin verir.
   - Bu itme işlemi, etkilenen yazma transaction'ını geçersiz kılabilir ve yeniden başlatılmasını gerektirebilir.
   - **AS OF SYSTEM TIME** kullanımı, belirli bir geçmiş zaman noktasındaki verileri okurken engellemelerin oluşmasını önler.
7. **Transaction İptalleri ve Yeniden Denemeler:**
   - Birçok transaction çatışması otomatik olarak yönetilir ve uygulama tarafından özel bir müdahale gerektirmez.

- Uygulamanın, iptal edilen transaction'ları işlemesi ve yeniden denemesi gereken durumlar vardır. Bu durumlar, üstel geri çekilme (exponential backoff) ile yeniden deneme, transaction izolasyon seviyelerinin ayarlanması, idempotent işlemlerin tasarımı veya uygulama mantığı içinde özel hata yönetimi gibi yöntemlerle ele alınabilir.

## 2.3 Dağıtım Katmanı

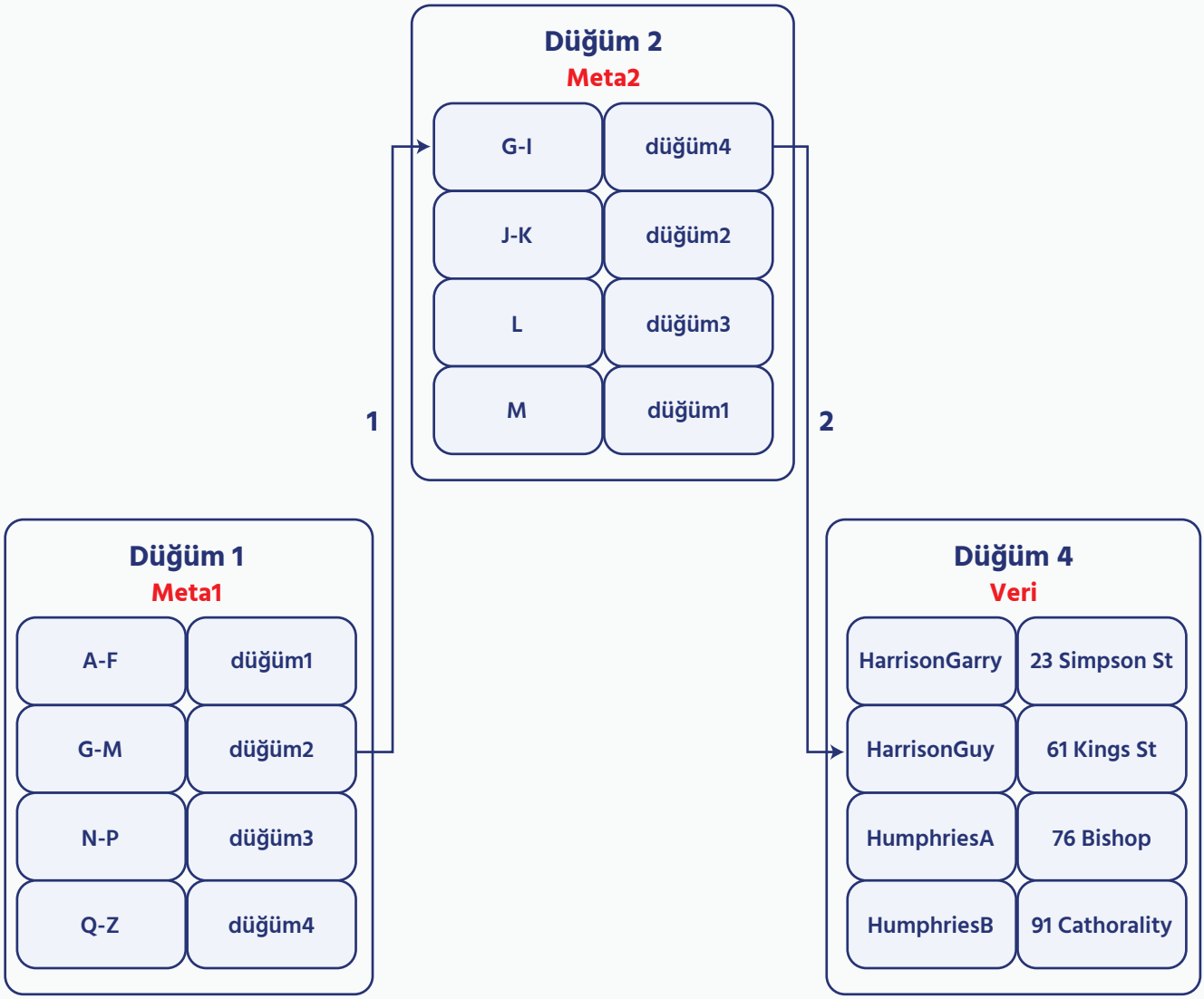
Dağıtım katmanı, verilerin coğrafi olarak dağıtılmış çok sayıda düğüm arasında nasıl depolandığını, yönetildiğini ve erişildiğini kontrol eden katmandır. Aynı zamanda, CockroachDB'nin dağıtık doğasını yöneten ve kullanıcıların veri tabanına yüksek performans, ölçeklenebilirlik ve güvenilirlikle erişebilmelerini sağlayan bir bileşendir. CockroachDB'de verilerin depolandığı aralıkların konum bilgisini tutan meta aralıkları dağıtık mimarinin temel taşlarından biridir.

### 2.3.1 Meta Aralıkları

Meta aralıkları, CockroachDB'nin veri tabanındaki herhangi bir veri parçasının yerini hızlı bir şekilde bulmasını sağlamaktadır. Bir kullanıcı veya uygulama veri istediğinde, veri tabanı sistemi önce meta aralıklarını sorgulamakta, hedef veri aralığının konumunu bulmakta ve ardından bu verilere erişim için ilgili düğümlere yönlendirmektedir. Ayrıca, düğümler arasında veri yeniden dengelendiğinde veya bir düğüm başarısız olduğunda, meta aralıkları, veri tabanının hızlı bir şekilde güncellenmiş konum bilgisine dayanarak işlemlere devam etmesini sağlamaktadır.

Sistem içindeki rolleri ve sakladıkları bilgiler açısından farklılık gösteren iki tür meta aralığı bulunmaktadır:

- **Meta1 Aralığı:** Tüm aralıkların adres bilgilerinin en üst düzey indeksidir. Meta1 aralığı, tüm Meta2 aralıklarının adreslerini saklar. Sistem, bir veri parçasının nerede olduğunu bulmak için önce Meta1 aralığına bakar.
- **Meta2 Aralığı:** Her bir veri aralığının replikasının hangi düğümde saklandığına dair bilgileri içerir.




*Şekil 14. Meta Aralıkları*

Şekil 14'te, düğüm 1 "HarrisonGuy" anahtarına ait veriye erişmek istediğinde öncelikle kendine ait meta1 kopyasına bakmaktadır. Meta1, G-M aralığı için meta2 bilgisinin düğüm 2'de bulunduğunu belirtmektedir. Meta2 de, G-I aralığı için bu verinin düğüm 4'te tutulduğunu göstermektedir.

## 2.3.2 Dedikodu Protokolü

CockroachDB, düğümler arasında geçici bilgileri paylaşmak için dedikodu (gossip) protokolünü kullanmaktadır. Dedikodu protokolü, bilgileri ağ üzerinde viral bir şekilde yayarak çalışmakta; yani her düğüm, aldığı bilgileri diğer düğümlere aktarmakta, bu da bilginin hızlı ve verimli bir şekilde tüm düğümlere ulaştırılmasını sağlamaktadır. Dedikodu protokolünün ana işlevleri aşağıdaki gibidir:

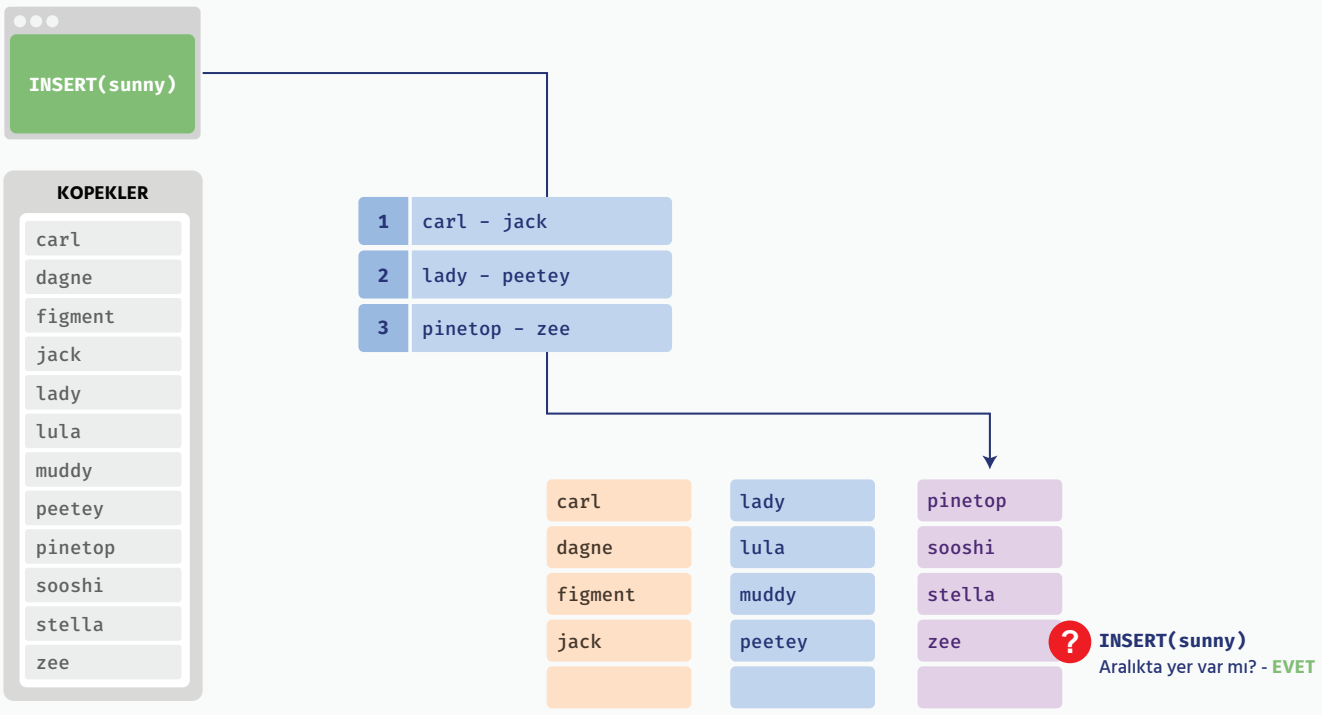
1. **Düğüm Keşfi:** Yeni bir düğüm sisteme katıldığında, dedikodu protokolü bu düğümün varlığını diğer düğümlere bildirir. Böylece, yeni düğüm hızla sistemdeki diğer düğümlerle iletişim kurabilir ve veri replikasyonu gibi işlemlere katılabilir.

2. **Durum Bilgisi Paylaşımı:** Dedikodu, düğümlerin canlılık durumu, sistem yükü, ağ gecikmesi gibi önemli sistem metriklerini paylaşmalarını sağlar. Bu bilgiler, sistemdeki yük dengesinin ve kaynak yönetiminin optimize edilmesine yardımcı olur.
3. **Yapılandırma Değişikliklerinin Yayılması:** Sistem yapılandırması veya politikaları gibi merkezi olmayan ayarlar değiştiğinde, dedikodu protokolü bu değişiklikleri tüm düğümlere hızlı bir şekilde yayarak sistem genelinde tutarlılığı sağlar.
4. **Hata Toleransı:** Dedikodu protokolü, düğümlerin başarısız olması veya ağ kesintileri gibi durumlarda sistem genelinde hızlı bir şekilde yeniden düzenlenmesini sağlar. Düğümler arızalandığında veya çevrim dışı olduğunda, dedikodu yoluyla bu bilgi paylaşılır ve sistem, veri erişilebilirliğini ve tutarlılığını korumak için gerekli düzeltmeleri yapar.

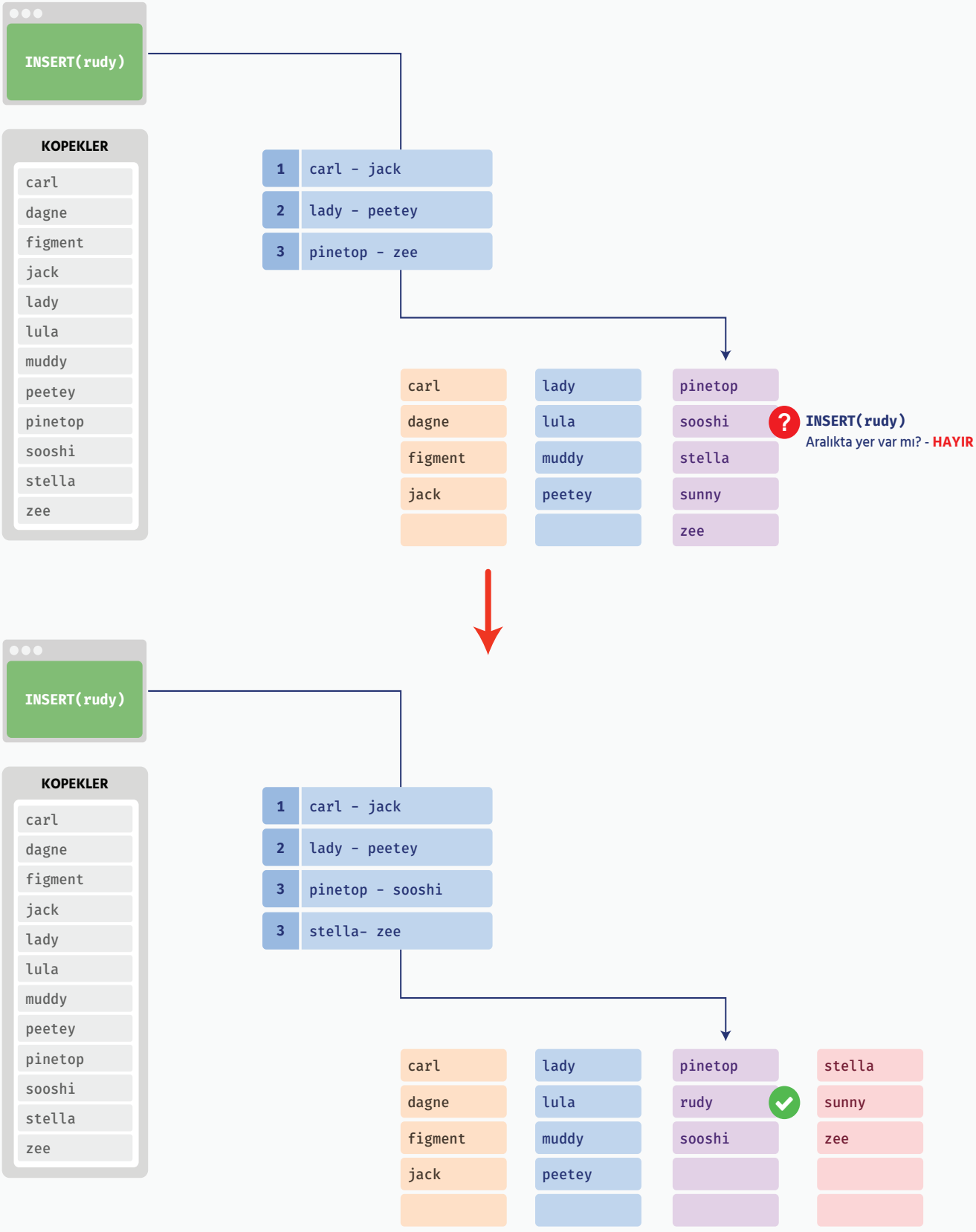
### 2.3.3 Aralık Bölünmeleri

CockroachDB, bir aralığın 512 MB'tan az kalmasını sağlamaya çalışmaktadır. Bir aralık bu boyutu aştığında, daha küçük ve ardışık iki aralığa bölünür. Aralıklar, bir yük eşiğini aştıklarında da bölünebilmektedir. Bu eşik değeri kullanıcı tarafından belirlenebilmektedir. Yüke dayalı olarak bölünürken, iki yeni aralık eşit boyutlarda olmayabilir. Varsayılan olarak, aralık, iki yeni aralık üzerindeki yükün kabaca eşit olacağı noktada bölünecektir.

Şekil 15'te, başlangıçta üç aralık içeren köpekler veri tabanına “sunny” isimli yeni bir köpek eklenirken, ilgili aralıkta yeterli alan bulunduğundan herhangi bir aralık bölünmesine gerek kalmadan ekleme işlemi tamamlanmıştır. Şekil 16'da ise, mevcut aralıkta yeterli alan olmadığı bir durumda “rudy” isimli yeni bir köpek eklenirken, aralık bölünmesinin gerçekleştiği gösterilmiştir.




*Şekil 15. Aralığa Veri Eklenmesi*




*Şekil 16. Aralık Bölünmesi*

Aralıklar, ALTER TABLE ve ALTER INDEX ifadelerinin SPLIT AT maddesi kullanılarak manuel olarak da bölünebilmekte ve gerekli durumlarda birleştirilebilmektedir. Eğer DELETE komutu aralıklardan veri siler ve aralık belirli bir boyut eşiğinin altına düşerse, CockroachDB aralığı komşu bir aralıkla birleştirebilmektedir.

### 2.3.4 Çok Bölgeli Dağıtım

CockroachDB'nin ücretli versiyonunun bir özelliği olan coğrafi bölümlendirme, verilerin belirli bir coğrafi bölge içinde konumlandırılmasını sağlamaktadır. Bu özellik, veri tabanı uygulamalarının yüksek erişilebilirlik, düşük gecikme süreleri ve iş gereksinimlerine uygun veri yerelleştirme ihtiyaçlarını karşılamasına olanak tanımaktadır.

CockroachDB, çok bölgeli dağıtımı kolaylaştırmak için çeşitli araçlar ve komutlar sunmaktadır. Veri tabanı yöneticileri, SQL komutları kullanarak veri tabanlarını, tabloları ve hatta belirli satırları farklı bölgelere atayabilmektedir. Ayrıca, veri tabanı düzeyinde, tablo düzeyinde veya satır düzeyinde replikasyon faktörlerini ve bölge tercihlerini ayarlayabilmektedir.

## 2.4 Replikasyon Katmanı

Replikasyon katmanı, veri tabanının yüksek erişilebilirliğini ve dayanıklılığını sağlamak için tasarlanmıştır. CockroachDB, veri kaybını önlemek ve bir düğüm arızalandığında sistemin kullanılabilirliğini sağlamak için verilerin birden fazla replikasını tutmaktadır. Replikalar arasında konsensüs sağlamak için raft algoritması kullanılmaktadır.

### 2.4.1 Raft Algoritması

Raft, dağıtılmış sistemlerde güçlü tutarlılık ve veri bütünlüğü sağlamak için tasarlanmış bir konsensüs algoritmasıdır. Her veri aralığı bağımsız bir raft grubudur ve her bir grubun konsensüsü ayrı ayrı belirlenir. Sistemdeki her replika, okuma ve yazma işlemleri için kullanılabilir, ancak bir güncellemenin sistem genelinde kabul edilip onaylanabilmesi için ilgili raft grubundaki çoğunluk oyun alınması gerekir.

Her raft grubunda esas olarak üç ana rol vardır:

1. **Lider:** Sistemdeki tüm değişiklikleri yönetir ve tüm veri değişikliklerinin diğer düğümlere (takipçilere) uygulanmasını sağlar. Herhangi bir veri değişikliği talebi önce lider tarafından alınır ve ardından takipçilere kopyalanır.
2. **Takipçi:** Lider tarafından gönderilen veri değişikliklerini kabul eder ve uygular. Takipçiler, liderin çöktüğünü veya erişilemez olduğunu algılarsa, yeni bir lider seçimi başlatılabilir.
3. **Aday:** Lider çökerse veya belirli bir zaman aşımına uğrarsa, takipçilerden bazıları liderlik için aday olabilirler.

Raft, temel anlamda aşağıdaki şekilde çalışmaktadır:

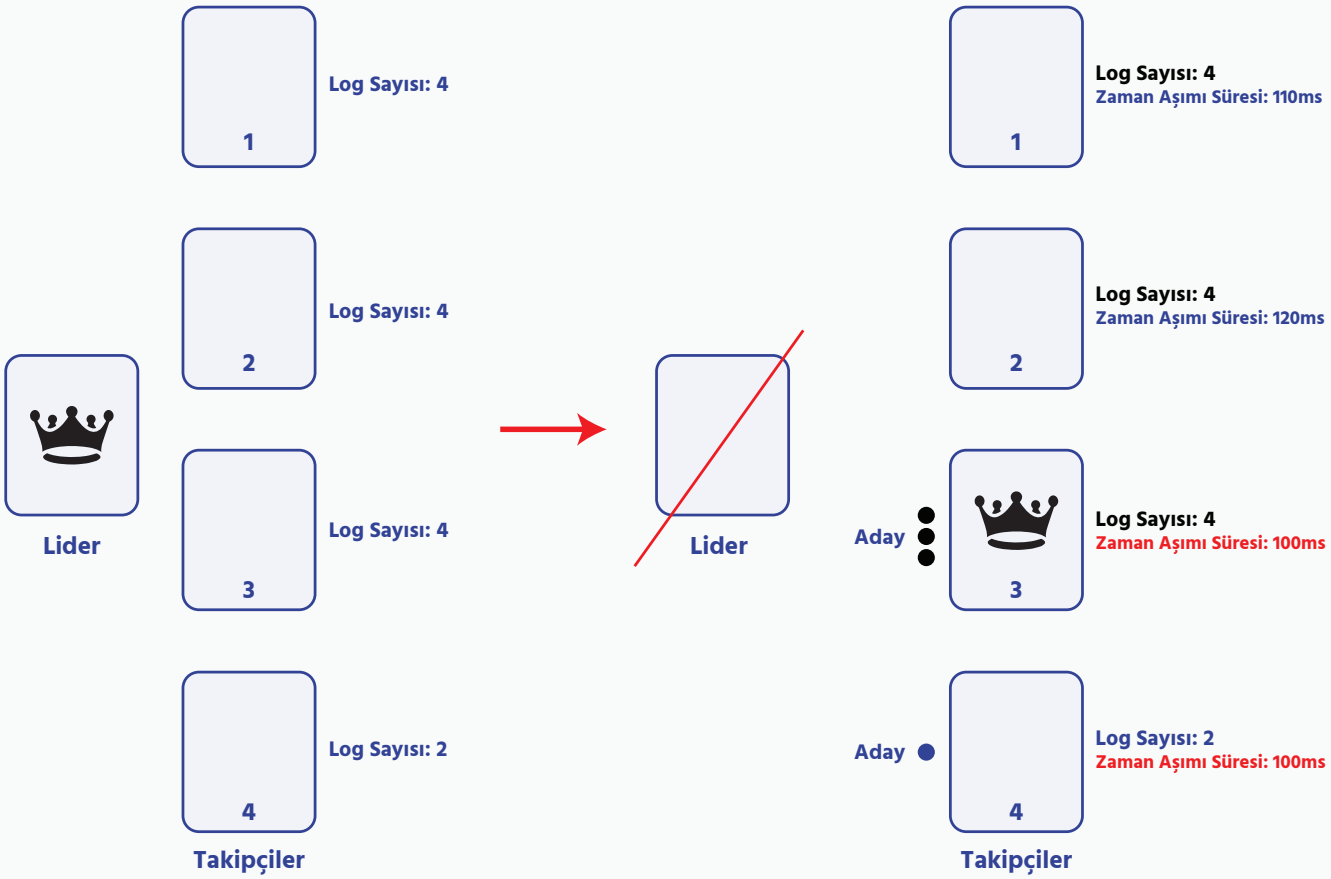
### 2.4.1.1 Log Replikasyonu

Lider, değişiklikleri bir loga yazmakta ve bu logu takipçilere göndermektedir. Takipçiler, logdaki değişiklikleri kendi replikalarına uygulayarak veri tabanını güncel tutmaktadır. Bir güncelleme isteği geldiğinde izlenen adımlar genellikle şu şekildedir:

1. **Güncelleme İsteğinin Alınması ve Liderin Bildirimi:** Güncelleme isteği, sistemdeki lider tarafından alınır. Lider, güncelleme isteğini diğer düğümlere bildirir.
2. **Takipçilerin Onayı:** Takipçi düğümler, güncelleme isteğini alır ve kendi loglarına ekler. Daha sonra, her takipçi düğüm, güncellemenin loglarına başarıyla eklendiğini belirten bir "APPENDED" mesajını lider düğüme geri gönderir.
3. **Çoğunluk Onayının Kontrolü:** Lider düğüm, sistemdeki tüm düğümlerin yarısından fazlasından onay mesajı alıp almadığını kontrol eder. Örneğin, sistemde lider dahil dört düğüm varsa, çoğunluk için en az üç düğümün onayı gerekir.
4. **Commit Mesajının Gönderilmesi:** Lider, kendisi dahil en az üç düğümden onay aldığında, güncellemenin onaylandığını ve artık tüm düğümler tarafından uygulanabileceğini belirten bir "COMMITTED" mesajı yayınlar.
5. **Güncellemenin Uygulanması:** Lider, güncellemenin onaylandığını duyurduktan sonra, takipçi düğümler güncellemeyi kendi veri setlerine uygular. Bu işlem, loglarda işaretlenen sırayla gerçekleşir.
6. **İstemciye Yanıt:** Güncelleme başarıyla onaylandıktan ve uygulandıktan sonra, lider düğüm istemciye işlemin başarılı bir şekilde tamamlandığına dair bir yanıt gönderir.

### 2.4.1.2. Lider Seçimi

Lider çöktüğünde veya belirli bir süre içinde iletişim kurmadığında, bir veya birden fazla takipçi aday olmakta ve diğer düğümlerden lider olmak için oy istemektedir. En çok oyu alan düğüm yeni lider olmaktadır. Tüm replikalar oylama yapmak zorunda değildir. Oy kullanmayan replikalar, yazma işlemleri sırasında düşük gecikmeli okumalar sağlamaktadır.




*Şekil 17. Raft Algoritması Lider Seçimi*

Şekil 17'de görülen örnekte, zaman aşımı süresi en kısa olup en fazla loga sahip olan üç numaralı düğüm yeni lider seçilir.

- **Güvenlik:** Raft, yanlış bir düğümün lider olarak seçilmesini engelleyen ve yalnızca en güncel verilere sahip düğümlerin yeni lider olabileceğini garantileyen mekanizmalar içerir.
- **Log Eşleştirme Prensibi:** Bir adayın lider olabilmesi için, en güncel log girdisine sahip olması gerekir. Bu, lider seçilen düğümün tüm önemli verilere sahip olduğunu garanti eder.
- **Log Ekleme Kuralı:** Lider, loga yalnızca yeni girdiler ekleyebilir; mevcut girdileri silmez veya değiştirmez. Bu, veri tutarlılığını korur.
- **Seçim Güvenliği:** Aynı term'de ("term" kavramı, liderlik dönemlerini tanımlamak için kullanılır) yalnızca bir lider seçilebilir, bu da sistemde birden fazla liderin olmasını önler.

## 2.5 Depolama Katmanı

Depolama katmanı, replikasyon katmanından gelen verileri okumakta ve yazmaktadır. Bu, verilerin çoğaltılabilirliğini ve dayanıklılığını sağlamak için kullanılan bir özelliktir. Her CockroachDB düğümü, düğüm başlatıldığında belirtilen ve cockroach sürecinin disk üzerindeki verilerini okuduğu ve yazdığı en az bir depolama alanı içermektedir. Bu veri, depolama motoru kullanılarak diskte KV çiftleri olarak depolanmaktadır. CockroachDB, daha önce kullanılan RocksDB'nin yerine kendi geliştirdiği PebbleDB depolama motorunu kullanmaya başlamıştır. PebbleDB, özellikle dağıtık SQL veri tabanı sistemleri için tasarlanmış, açık kaynaklı ve yerleşik bir KV depolama çözümüdür. Bu yeni motor, RocksDB'nin sağladığı performans avantajlarını korurken, daha basit bir yapıda ve CockroachDB'nin ihtiyaçlarına daha uygun şekilde optimize edilmiştir. PebbleDB, veri saklama ve sorgulama işlemlerinde yüksek performans ve verimlilik sağlamak için **LSM** yapısını kullanmaktadır.

### 2.5.1 Log-Yapılı Birleştirme (LSM) Ağaçları

LSM ağacı, bir veri depolama ve yönetim yapısıdır. Bu yapı, veriyi disk üzerinde etkili bir şekilde saklamak ve veriye hızlı bir şekilde erişim sağlamak amacıyla tasarlanmıştır. LSM ağacı, verileri disk üzerine log olarak yazan, arka planda birleştirme işlemleri ile veri düzenini ve sıkıştırmayı sağlayan bir yapıdır. Bu yapı, yüksek ekleme hızlarına ulaşmayı sağlarken etkili rastgele okuma erişimi de sunmaktadır. LSM ağacının bu özellikleri, özellikle büyük veri setleri ve yoğun yazma işlemleri içeren uygulamalar için idealdir.

LSM ağacı, temelde geçici bir bellek alanı olan "MemTable" ve kalıcı depolama alanı olan "SSTable" bileşenlerinden oluşmaktadır. MemTable, yeni eklenen verilerin hızlı bir şekilde yazılmasını sağlamaktadır. SSTable ise disk üzerinde sıralı bir şekilde KV çiftlerini depolayan veri bloklarıdır. SSTable, sıralı ve indekslenmiş olduğu için arama işlemi hızlıca yapılabilmektedir. SSTable değişmezdir (immutable); bir MemTable, SSTable'a dönüştüğünde üzerinde değişiklik yapılamamaktadır. Güncelleme yapmak için yeni değer eklemek yeterlidir, çünkü sistem eski versiyonlarını yok saymaktadır.

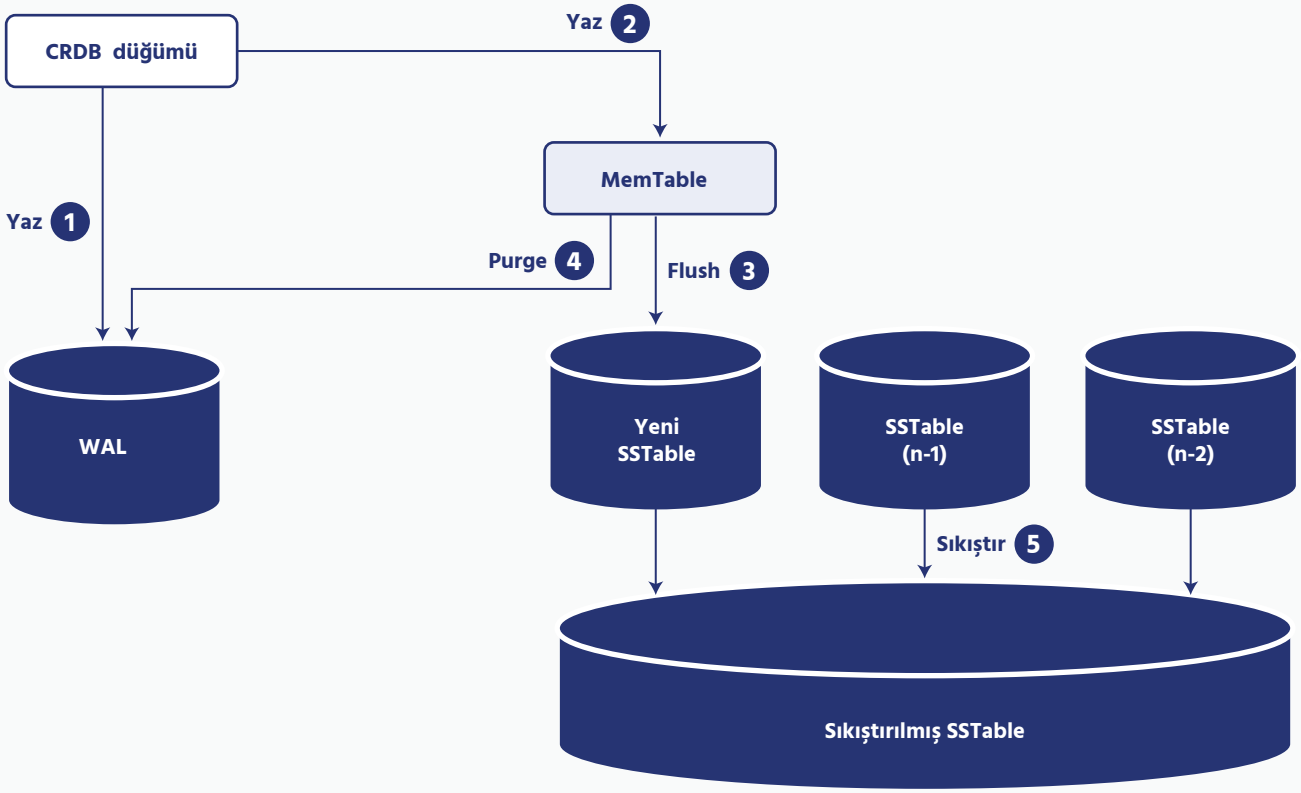
LSM ağacı mimarisinde, veriler L0'dan L6'ya kadar olan farklı seviyelerde tutulur. Bunun temel nedeni, yazma ve okuma işlemlerinin performansını optimize etmek ve disk giriş/çıkış işlemlerini etkin bir şekilde yönetmektir. L0 en üst seviyedir. L6 ise en alt seviyedir. Yeni veriler L0'a eklenmekte ve zamanla alt seviyelere doğru birleştirilmektedir.

1. **L0 Seviyesi:** MemTable'dan disk üzerine aktarılan verilerin ilk olarak depolandığı bölgedir. Bu seviye, düzensiz olarak yerleştirilmiş bir dizi SSTable'dan oluşur. Veriler, bu seviyede geçici bir süre tutulur ve ileri bir aşamada daha sistemli bir yapıya dönüştürülmek üzere birleştirme işlemi gerçekleştirilir.
2. **L1'den L6'ya Seviyeler:** L1 seviyesinden başlayarak, her bir üst kademe, alt kademeden alınan verileri büyük ve düzenli SSTable'lara birleştirir. Bu işlem, verilerin sıralı ve benzersiz bir biçimde

depolanmasını sağlar. Bu yöntem, her bir anahtarın yalnızca bir SSTable içerisinde bulunmasını ve veriye erişim süreçlerinin hızlanması garantiler. Veri erişim süreçlerinin optimizasyonunu sağlayarak, veri tabanı sistemlerinin performansını artırır.

### 2.5.1.1 LSM Yazma İşlemi

Şekil 18'de, LSM ağacı mimarisi kullanan bir CockroachDB (CRDB) düğümünde veri yazma işleminin aşamaları gösterilmektedir:




*Şekil 18. LSM Yazma İşlemi*

1. **WAL'e Yazılması:** Yazma işlemi gerçekleştiğinde, ilk olarak WAL'e kaydedilir. WAL, çökme durumunda SSTable'a yazılmamış verilerin kaybolmasını önler, log üzerinden tamamlanmamış yazma işlemleri kurtarılabilir.
2. **MemTable'a Yazılması:** Yazma işlemi WAL'e uygulandıktan sonra MemTable'a da uygulanır.
3. **Flush İşlemi:** MemTable belirli bir boyuta ulaştığında, içeriği diske yazılıp yeni bir SSTable oluşturulur.
4. **Purge İşlemi:** Başarılı bir flush işleminden sonra, veriler artık güvenli bir şekilde diskteki SSTable'da bulunduğundan, WAL'daki ilgili kayıtlar temizlenebilir.
5. **Sıkıştırma (Compaction):** Zamanla, disk üzerinde birden fazla SSTable oluşturulur. Sıkıştırma işlemi, okuma performansını optimize etmek ve kullanılan alanı azaltmak için SSTable'lardaki verileri birleştirerek, tekrar eden veya silinmiş girdileri kaldırır. Böylece veriyi daha düzenli ve daha az yer kaplayacak şekilde yeniden yapılandırır.

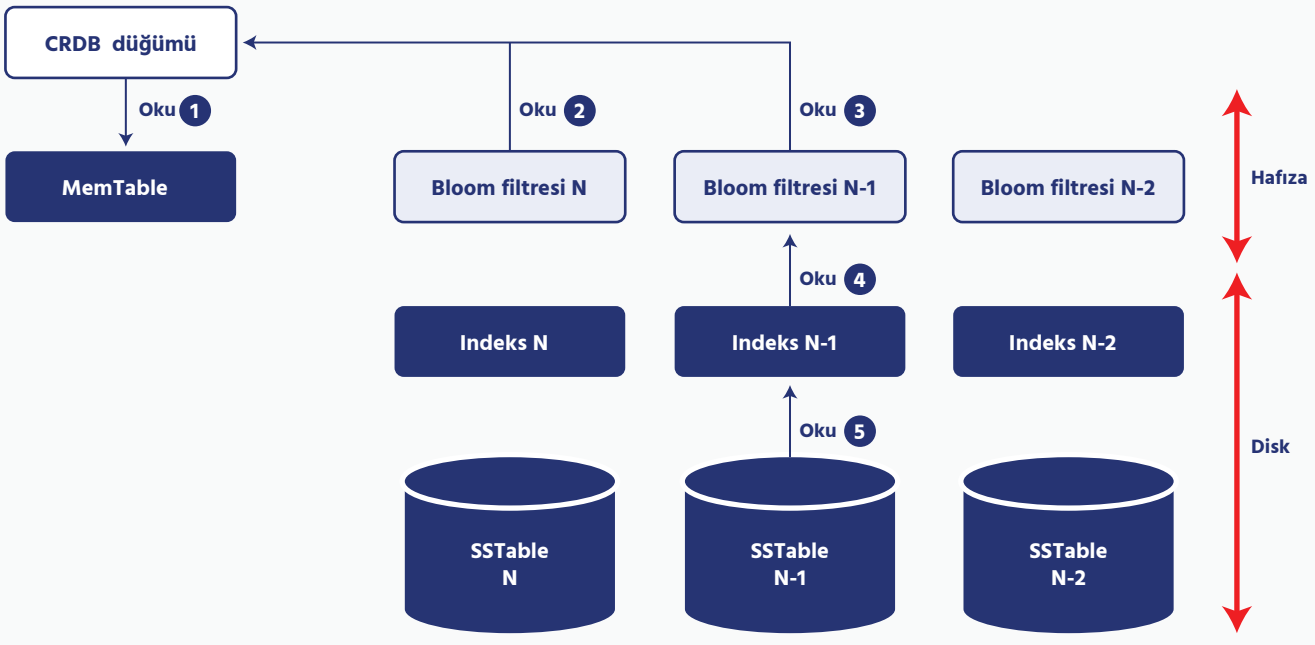
Bu süreç, yazma işlemlerinde verimlilik sağlamak için bellek içi yapıları kullanmakta ve yazma işlemi tamamlandıktan sonra olası bir sistem çökmesi durumunda verilerin güvende kalmasını sağlamaktadır. Ayrıca, zamanla biriken ve okuma sürelerini uzatabilecek çok sayıda SSTable'ı birleştirerek yeniden düzenlemekte, okuma performansını korumaya yardımcı olmaktadır. Okuma performansını arttırmak için bloom filtreleri de kullanılmaktadır.

### 2.5.1.2 SSTable ve Bloom Filtreleri

Her SSTable indekslenmiştir. Ancak, disk üzerinde birçok SSTable bulunabilmekte ve bu da arama işlemlerinde bir çarpan etkisine sebep olmaktadır. Çünkü istenen satırı bulmak için her bir SSTable'da bütün indeksleri incelemek gerekebilmektedir. Birden çok indeks aramasının üstesinden gelmek için bloom filtreleri kullanılmaktadır. Bloom filtresi, bir dosyada belirli bir değerin bulunup bulunmadığını kontrol etmek için kullanılan bir yapıdır. Eğer bloom filtresi bir değerin var olduğuna dair pozitif bir yanıt verirse, bu, değerin dosyada olabileceğini göstermektedir. Ancak bu her zaman kesin değildir. Öte yandan, bloom filtresi bir değerin bulunmadığını söylerse, bu bilgi kesindir. Bu özellik sayesinde, bloom filtresi bir anahtarın belirli bir SSTable içinde olmadığını belirttiğinde, arama yapmadan o SSTable atlanabilmektedir. CockroachDB, bloom filtrelerini bir anahtarın bir sürümünü hangi SSTable'ların içerdiğini hızlı bir şekilde belirlemek için kullanmaktadır.

### 2.5.1.3 LSM Okuma İşlemi

CRDB düğümü, ilk olarak en güncel bilgilerin saklandığı MemTable'dan ve en üst seviye olan L0'daki SSTable'lardan başlayarak okuma yapmaktadır. Gerekli veri bulunmazsa, sistem daha eski verilerin bulunduğu alt seviyedeki SSTable'lara doğru ilerlemektedir. Bu süreç, aranan veriye ulaşılana kadar devam etmektedir. Veri okuma işlemleri Şekil 19'daki gibi gerçekleşmektedir:




*Şekil 19. LSM Okuma İşlemi*

1. **MemTable'dan Okuma:** Veri tabanı isteği ilk olarak MemTable'dan okur. Aranan değer burada bulunursa işlem tamamlanır.
2. **Bloom Filtrelerinin Kullanılması:** MemTable'da değer bulunmazsa, en üst seviyedeki (L0) tüm SSTable'lar için bloom filtreleri kullanılır. Bloom filtresi, aranan anahtar değerin SSTable'da bulunmadığını gösteriyorsa, diskteki okuma işlemine gerek kalmaz.
3. **Bloom Filtreleri ile Verinin Bulunması:** Bloom filtresi, aranan anahtarın belirli bir SSTable'da bulunabileceğini gösteriyorsa, ilgili anahtarı içeren her bir seviyedeki SSTable kontrol edilir.
4. **SSTable İndeks Kullanımı:** Bloom filtresi, aranan anahtarın bir SSTable içinde bulunabileceğini gösteriyorsa SSTable'ın kendisindeki indeks kullanılarak aranan değere doğrudan erişmek için arama yapılır. Bu yöntem, verinin hızlı bir şekilde bulunmasını sağlar.
5. **SSTable İçindeki Değerin Aranması:** Eşleşen değer bulunduğunda, daha eski SSTable'larda arama yapmaya gerek kalmaz çünkü LSM yapısında yeni veriler daha eski verileri geçersiz kılar.

Okuma işlemi sırasında, özellikle sık erişilen veriler için blok önbelleği devreye girmektedir.

#### 2.5.1.4 Blok Önbelleği

PebbleDB'de, sık kullanılan verilere hızla erişmek için bir blok önbellek (block cache) bulunmaktadır. Bu, MemTable'dan, bloom filtreleri ve bellek içi indekslerden bağımsız çalışan bir sistemdir. Blok önbelleği, LRU prensibine göre çalışmaktadır. Yeni veriler geldiğinde, en uzun süre önce kullanılan veri önbellekten silinmekte ve yerine yeni veri eklenmektedir. Bu sayede, sistem birden fazla SSTable ve bloom filtresini inceleme gerekliliğini atlayarak doğrudan blok önbellekten veri okuyabilmekte; böylece okuma işlemleri daha hızlı hale gelmektedir.

Blok önbelleğin sağladığı verimliliğinin yanında, bir veri tabanı yönetim sisteminin depolama optimizasyonu ve sistem performansının sürekliliği için bir diğer kritik süreç de çöp toplama mekanizmasıdır.

#### 2.5.1.5 Çöp Toplama ve Korunan Zaman Damgaları

CockroachDB, diskte biriken veri miktarını yönetmek için çöp toplama (garbage collection) mekanizmasını kullanmaktadır. Çöp toplama, artık kullanılmayan eski MVCC sürümlerini silerek diskteki veri boyutunu optimize etmektedir. Ancak bazı durumlarda eski verileri korumak önemlidir. Bu durumlarda, korunan zaman damgaları özelliğinden yararlanılabilmektedir.

Korunan zaman damgaları, belirli bir zamandaki verileri çöp toplamadan korumak için kullanılmaktadır. Örneğin, bir yedekleme işlemi başlatıldığında, yedekleme kapsamındaki tüm veriler için bir korunan zaman damgası oluşturulmaktadır. Bu sayede, yedekleme işlemi tamamlanana kadar bu veriler silinmemektedir. Bu sistem, CockroachDB'nin tarihsel verilere dayalı işlemlerinin güvenliğini ve tutarlılığını sağlayan önemli bir bileşenidir.

## 3. Bölümleme

Verileri bölmek (Data partitioning), bir uygulamaya ait verileri ayrı parçalara veya bölümlere (partition) ayırma işlemini ifade etmektedir. Bu bölümler, ayrı ayrı depolanabilmekte, erişilebilmekte ve yönetilebilmektedir. CockroachDB, verileri **otomatik** olarak bölümlere ayırmaktadır. Bu özelliği; veri tabanının performansını artırmak, ölçeklenebilirliği sağlamak ve yüksek erişilebilirlik elde etmek için kullanılmaktadır. Genel olarak veriler, yatay ve dikey olarak bölünmektedir. Tablo 2'deki veriler iki yöntemle de bölünebilmektedir.

**Tablo 2. Örnek Veri Tablosu**

| id | isim | sehir | maas |
|---|---|---|---|
| 1 | theo | london | 213 |
| 2 | kee | portland | 75444 |
| 3 | julian | new york | 645 |
| 4 | jasper | boston | 342 |
| ... | ... | ... | ... |
| 9998 | syd | shanghai | 5145 |
| 9999 | nige1 | santigo | 4350 |
| 10000 | marichka | london | 873 |
| 10001 | luke | new york | 2091 |

## 3.1 Dikey Bölümleme

Dikey bölümleme, tablonun sütun temelli bölündüğü durumdur. Burada maaş sütunu çok sık güncelleniyorsa, ancak isim ve şehir sütunları nispeten sabitse, tabloyu dikey olarak bölmek ve yüksek performanslı bir sunucuda **Bölüm 2**'yi bulundurmak mantıklı olabilmektedir. Diğer yandan, **Bölüm 1** verisi, kullanıcı tarafından daha az kullanıldığı için daha düşük performanslı makinelerde depolanabilmektedir.

**Tablo 3. Dikey Bölümleme**

**Bölüm 1**

| id | isim | sehir |
|---|---|---|
| 1 | theo | london |
| 2 | kee | portland |
| 3 | julian | new york |
| 4 | jasper | boston |
| ... | ... | ... |
| 9998 | syd | shanghai |
| 9999 | nige1 | santigo |
| 10000 | marichka | london |
| 10001 | luke | new york |

**Bölüm 2**

| id | maas |
|---|---|
| 1 | 213 |
| 2 | 75444 |
| 3 | 645 |
| 4 | 342 |
| ... | ... |
| 9998 | 5145 |
| 9999 | 4350 |
| 10000 | 873 |
| 10001 | 2091 |

## 3.2 Yatay Bölümleme

Yatay bölümleme, tablonun satırlara göre bölündüğü durumdur. Tablo 2'deki örnekte 10001 adet veri bulunmaktadır. Bu veriler 2 bölüme 1-500 ve 501-10001 olacak şekilde yatay olarak bölünmüştür.

CockroachDB, performansı ve ölçeklenebilirliği artırmak amacıyla genellikle tabloları birden fazla sunucuya yatay olarak böler, bu işleme **sharding** denir. Bu sayede artan yük ve talepler karşılanır, yüksek erişilebilirlik sağlanır ve hata durumlarında sistem kendini iyileştirebilir.

Ayrıca, bölümleme işlemleri bireysel de yönetilebilir ancak çoğu zaman buna gerek yoktur (Detaylı bölümleme özellikleri ücretli olarak sunulmaktadır.).

**Tablo 4. Yatay Bölümleme**

**Bölüm 1**

| id | isim | sehir | maas |
|---|---|---|---|
| 1 | theo | london | 213 |
| 2 | kee | portland | 75444 |
| 3 | julian | new york | 645 |
| 4 | jasper | boston | 342 |
| ... | ... | ... | ... |
| 500 | alfonso | mex.cityc | 435435 |

**Bölüm 2**

| id | isim | sehir | maas |
|---|---|---|---|
| 500 | alfonso | mex.cityc | 435435 |
| ... | ... | ... | ... |
| 9998 | syd | shanghai | 5145 |
| 9999 | nige1 | santigo | 4350 |
| 10000 | marichka | london | 873 |
| 10001 | luke | new york | 2091 |

## 4. SQL Performansını Artırmak için En İyi Yöntemler

1. **Çok Satırlı DML İşlemleri Kullanmak: INSERT, UPSERT** ve **DELETE** işlemleri için, birden fazla satırı tek bir işlemde eklemek, çok sayıda tek satırlı işlem yapmaktan daha hızlıdır.
2. **UPSERT Kullanmak:** İkincil indeks içermeyen tablolar için **UPSERT** kullanımı, **INSERT ON CONFLICT** ile karşılaştırıldığında daha hızlıdır çünkü okuma yapmadan doğrudan yazma işlemi gerçekleştirir.

3. **Toplu Ekleme İşlemleri için IMPORT Kullanmak:** Yeni tablolara toplu veri eklerken **IMPORT** kullanmak, **INSERT** ile karşılaştırıldığında daha verimlidir.
4. **TRUNCATE Kullanarak Hızlı Silmek:** Tüm satırları silmek için **DELETE** yerine **TRUNCATE** komutu kullanmak, tabloyu sıfırdan oluşturarak işlemi hızlandırır.
5. **Partiler Halinde Silmek:** Büyük miktarda verinin silinmesi gerektiğinde, satırları küçük gruplar halinde iteratif olarak silmek daha verimlidir.
6. **Sütun Ailelerini Atamak:** Çok sık güncellenmeyen sütunlar varsa sütun aileleri kullanılarak performans artırılabilir.
7. **Çok Sütunlu Birincil Anahtarlar Kullanmak:** Eğer doğru tasarlanmışsa, çok sütunlu birincil anahtarlar daha iyi performans sunabilir.
8. **ID Üretiminde UUID Kullanmak:** CockroachDB'de, geleneksel veri tabanı mimarilerinde sıkça tercih edilen otomatik olarak artan ardışık sayılarla oluşturulan birincil anahtarların (ID'lerin) kullanılması, tabloya veri eklendiğinde sıralı bir kayıt düzeni oluşturmaktadır. Bu yöntem, ardışık veri ekleme işlemlerinde verilerin tek bir veri aralığına yoğunlaşmasına ve bu aralığı yöneten tek bir leaseholder'ın darboğaz oluşturmasına neden olabilir. Ardışık ID kullanımının doğurduğu bu darboğaz, yazma işlemlerinin hızını ve dağıtılmış sistemdeki genel verimliliği olumsuz yönde etkileyebilir. Bu nedenle, CockroachDB'de ID'lerin UUID formatında tutulması tavsiye edilmektedir. UUID kullanımı, 128 bitlik benzersiz değerler sayesinde, verilerin tablo boyunca daha eşit bir şekilde dağılmasını sağlar. Bu eşit dağılım, yazma ve okuma işlemlerinin küme genelinde dengeli bir şekilde dağıtılmasını kolaylaştırır, CockroachDB'nin dağıtılmış mimarisinin verimliliğini artırır ve böylece sistemin ölçeklenebilirliği ile performansını iyileştirir.

## 5. Sütun Veri Tipi Değiştirme Kısıtlamaları

CockroachDB'de tablo sütunlarının veri tiplerini değiştirmeye yönelik bazı kısıtlamalar vardır. Sütunun veri tipinin değiştirilemeyeceği durumlar aşağıda belirtilmiştir:

1. **Sütun Bir İndeksin Parçasıysa:** Eğer bir sütun bir indeks tarafından kullanılıyorsa, bu sütunun veri tipi değiştirilemez. İndeksler sütunların veri tiplerine bağlı olarak çalıştığından, veri tipinde bir değişiklik yapmak indeksin işlevselliğini bozabilir. Bu nedenle, indeksli bir sütunun veri tipi değiştirmek isteniyorsa önce indeksi kaldırmak, veri tipini değiştirmek ve ardından gerektiğinde indeksi tekrar oluşturmak gerekir.
2. **Sütunun CHECK Kısıtlamaları Varsa:** CHECK kısıtlamaları, sütunun alabileceği değerleri sınırlar. Sütunun veri tipini değiştirmek, bu kısıtlamaların ihlal edilmesine neden olabilir.
3. **Sütun Bir Diziye (Sequence) Sahipse:** Diziler, genellikle birincil anahtarlar için otomatik olarak artan sayılar üretmek üzere kullanılır. Sütunun veri tipi değişirse, indeksin ürettiği değerler sütunla uyumsuz hale gelebilir.

4. **ALTER COLUMN TYPE İfadesi Birleştirilmiş Bir ALTER TABLE İfadesinin Parçasıysa:** Birden fazla değişiklik tek bir ALTER TABLE komutu içinde yapılmaya çalışılırsa, bu komutların bir parçası olarak sütunun veri tipi değiştirilemez.
5. **ALTER COLUMN TYPE İfadesi Açık (Explicit) Transaction İçindeyse:** Veri tipi değişikliğinin bağımsız bir transaction olarak yürütülmesi gerekir. Bu nedenle başka transaction'ların içinde yer alamaz.

Bir sütunun veri tipini değiştirmenin, veri tabanındaki diğer nesnelerle ilişkili karmaşık etkileşimler yaratabileceğini ve bu nedenle veri tipi değişikliği yapmadan önce her zaman tam bir yedek alınması ve dikkatli olunması gerektiğini unutmamak önemlidir.

## 6. Ne Zaman Kullanılmalı?

1. **Büyük Ölçekli Projeler:** Büyük ölçekli projeler, genellikle yüksek hacimli veri işleme ve birden fazla işlemi eş zamanlı olarak yönetme kapasitesi gerektirmektedir. CockroachDB, veri dağıtımını ve çoğaltmayı otomatik olarak yönetmekte, böylece büyük miktarda veri üzerinde çalışırken yüksek performans ve güvenilirlik sağlamaktadır.
2. **Küresel Dağıtım Gerektiren Projeler:** CockroachDB, küresel dağıtım için optimize edilmiş bir veri tabanıdır; verileri coğrafi olarak dağıtarak, her kullanıcıya en yakın sunucudan hizmet verilmesini sağlamaktadır. Coğrafi bölümleme (geo-partitioning) özelliği ile verileri belirli bölgelerde saklayarak yerel yasalara uyumu ve düşük gecikme süreleri garanti etmektedir. Bu yaklaşım, küresel ölçekte yüksek kullanılabilirlik ve hızlı veri erişimi sunmaktadır. Bu nedenle, çok bölgeye yayılmış uygulamalar için idealdir.
3. **Yüksek Erişilebilirlik Gerektiren Projeler:** Kesintisiz hizmet sunması gereken uygulamalar için CockroachDB, hata toleransı ve otomatik yük devretme özellikleriyle yüksek erişilebilirlik sağlamaktadır. Bu özellikler, bir düğüm veya veri merkezi çöktüğünde bile uygulamanın çalışmaya devam etmesini sağlamaktadır.
4. **Ölçeklenebilirlik İhtiyacı Olan Projeler:** İş yükünün ve veri hacminin zaman içinde değişebileceği projeler için CockroachDB, ölçeklenebilir bir yapı sunmaktadır. Yeni düğümler kolayca eklenebilir. Ek olarak, sistem veri dağıtımını ve yük dengesini otomatik olarak yeniden düzenleyerek ölçeklenmeye izin vermektedir.

## 7. Ne Zaman Kullanılmamalı?

1. **Küçük Ölçekli Projeler:** CockroachDB'nin sunmuş olduğu karmaşıklık ve yönetim gereksinimleri, genellikle küçük ölçekli projeler için fazla olabilir, bu durumda daha basit bir veri tabanı kullanılabilir.

2. **Gelişmiş Analitik ve Büyük Veri İşlemleri İçeren Projeler:** CockroachDB, OLTP uygulamaları için uygundur. Ancak büyük veri analitiği veya karmaşık veri depolama gibi gelişmiş OLAP içeren uygulamalar için uygun değildir.
    - **OLTP,** gerçek zamanlı işlem işleme sistemlerini ifade eder. Bu tür sistemler, veri tabanı güncellemeleri, ekleme ve silme gibi kısa, hızlı işlemleri yönetmek üzere tasarlanmıştır. Örnek olarak, perakende satış noktası sistemleri, bankacılık işlemleri veya çevrim içi rezervasyon sistemleri sayılabilir.
    - **OLAP,** büyük veri kümeleri üzerinde karmaşık sorgular yapmak için tasarlanmıştır. Bu sistemler, genellikle veri ambarlarında kullanılır ve veri analizi, raporlama veya iş zekâsı uygulamaları için uygundur. Örnek olarak, finansal raporlama ve büyük veri analizi verilebilir.
3. **Belge Tabanlı Veri Modelleri Kullanan Projeler:** Belge tabanlı bir veri tabanı gerektiğinde, örneğin MongoDB gibi, CockroachDB bu spesifik ihtiyacı tam anlamıyla karşılayamayabilir.

# CockroachDB ve PostgreSQL

CockroachDB, PostgreSQL'in çoğu SQL dil özelliğini, veri tiplerini ve sorgularını destekler. Bu uyumluluk, PostgreSQL'den CockroachDB'ye geçişi kolaylaştırır, böylece mevcut PostgreSQL sorguları ve veri tabanı yapıları genellikle doğrudan veya minimal değişikliklerle CockroachDB'ye taşınabilir (CockroachDB'nin desteklediği SQL özelliklerine https://www.cockroachlabs.com/docs/v23.1/sql-feature-support adresinden erişilebilir).

## 1. Desteklenmeyen SQL Özellikleri

1. **XML Fonksiyonları:** xmlparse, xmlserialize, xpath, xmlelement vb.
2. **Saklı Yordamlar (Stored Procedures):** CockroachDB, saklı yordamlar yerine UDF'leri destekler. Saklı yordamlar, karmaşık iş mantığını ve işlemleri veri tabanı seviyesinde kapsüllemek için daha güçlüdür. UDF'ler daha basit işlemler için uygundur, ancak karmaşık işlemler için sınırlı olabilir.
3. **Tetikleyiciler ve Olaylar (Triggers and Events):** CockroachDB, veri tabanı seviyesinde tetikleyici desteği sunmaz; bunlar uygulama mantığına entegre edilmelidir. PostgreSQL'de tetikleyiciler, veri tabanındaki belirli olaylara (örneğin bir tabloya ekleme, güncelleme veya silme işlemi) yanıt olarak otomatik şekilde çalıştırılan fonksiyonlardır. Bu, veri bütünlüğünü korumak, otomatik işlemleri gerçekleştirmek veya karmaşık iş mantığını uygulamak için kullanılabilir.
4. **Birincil Anahtarı Kaldırmak (Drop Primary Key):** CockroachDB'de her tablonun birincil anahtarı olmak zorundadır. Tek bir transaction içinde eskisini kaldırıp yenisini eklemek mümkündür.
5. **Sütun Düzeyinde Ayrıcalıklar (Column-Level Privileges):** PostgreSQL'in sütun düzeyinde ayrıcalıklar özelliği, daha ince ayarlı bir güvenlik kontrolü sağlar. Bu özellik, belirli sütunlara erişimi kısıtlamak veya belirli kullanıcılara sadece bazı sütunlara erişim izni vermek için kullanılabilir. CockroachDB'de bu özelliğin bulunmaması, hassas verilerin ve detaylı güvenlik gereksinimlerinin olduğu durumlarda problem yaratabilir.
6. **Bir Tablodan Tek Bir Bölümü Kaldırmak:** CockroachDB'nin dağıtılmış mimarisi, veri dağılımını ve replikasyonunu bölümler arasında otomatik olarak yönetir. Bölümlerin ayrı ayrı silinmesi, veri bütünlüğü ve sorgu performansı üzerinde beklenmeyen etkilere yol açabilir. Bu nedenle, CockroachDB bütünsel bir bölümleme yönetimi sunar. Ayrıca mevcut bölümleri yeniden yapılandırmak ve verileri farklı bölümlere taşımak mümkündür.
7. **XA Söz Dizimi (XA Syntax):** XA söz dizimi, birden fazla kaynak üzerinde dağıtılmış işlemleri yönetmek için kullanılan bir standarttır. Bu, genellikle iki fazlı commit protokolünü içerir. CockroachDB, XA işlemlerini ihtiyaç duymaz çünkü dağıtık işlem yönetimi için raft algoritmasını kullanır.
8. **Şablondan (Template) Veri Tabanı Oluşturma:** Şablondan veri tabanı oluşturma, özel yapılandırmaları veya veri yapılarını yeni veri tabanlarına hızlı bir şekilde kopyalamak için kullanışlıdır. CockroachDB, daha basit ve doğrusal bir veri tabanı oluşturma sürecine odaklanır ve şablon mekanizmaları sunmaz.
9. **FDW:** PostgreSQL, FDW aracılığıyla, harici veri kaynaklarına ve farklı veri tabanlarına erişim sağlar. Bu özellik, veri tabanı heterojenliğini yönetmek ve farklı veri kaynakları arasında entegrasyon sağlamak için kullanılır. CockroachDB, FDW özelliğini desteklemez. Bunun yerine, kendi içinde tutarlı ve ölçeklenebilir dağıtılmış transaction'lara odaklanır.

10. **Danışman Kilit (Advisory Lock) Fonksiyonları:** Danışman kilit fonksiyonları, geliştiricilerin uygulama içinde belirli işlemleri koordine etmek için kullanabileceği, veri tabanı tarafından zorunlu kılınmayan opsiyonel kilitlerdir. Bu kilitler, özellikle işlemler arası çakışmayı önlemek amacıyla tasarlanmıştır. CockroachDB bu özelliği desteklememektedir.

Bu farklılıklar, CockroachDB'nin öncelikle dağıtılmış mimariye ve ölçeklenebilirliğe odaklanmasından kaynaklanmaktadır. PostgreSQL'e özgü bu özelliklerin eksikliği, CockroachDB'nin belirli kullanım senaryoları için uygun olmadığı anlamına gelebilmektedir (Desteklenmeyen özelliklere https://www.cockroachlabs.com/docs/stable/postgresql-compatibility adresinden erişilebilir).

## 2. Yedekleme ve Kurtarma

Tablo 5'te görüldüğü üzere, CockroachDB'nin ücretsiz sürümü sadece tam ve manuel yedekleme işlemlerini desteklemektedir. Bu sebeple, farklı bir yedekleme yöntemi kullanılmak istenirse, CockroachDB'nin ücretsiz sürümü yeterli olmayabilir. Alternatif olarak, CockroachDB'nin ücretli sürümü tercih edilebilir.

**Tablo 5. Yedekleme ve Kurtarma**

| | PostgreSQL | CockroachDB | Ücretli CockroachDB |
|---|---|---|---|
| Tam Yedekleme | ✔ | ✔ | ✔ |
| Manuel Yedekleme | ✔ | ✔ | ✔ |
| Artımlı Yedekleme | ✔ | | ✔ |
| Zamanlanmamış Yedekleme | | | ✔ |
| Coğrafi Olarak Dağıtılmış Yedekleme | | | ✔ |
| Zamana Bağlı Veri Kurtarma | ✔ | | ✔ |
| Bulut Depolama Entegrasyonu | ✔ | | ✔ |

- **Tam Yedekleme (Full Backup),** veri tabanının bütün durumunun bir kopyasını oluşturur.
- **Manuel Yedekleme,** otomatikleştirilmiş bir süreç olmadan, elle veri yedekleme komutlarının çalıştırılmasıdır.
- **Artımlı Yedekleme (Incremental Backup),** en son ki yedeklemenin ardından, sadece değişen verilerin yedeklendiği yöntemdir.
- **Zamanlanmış Yedekleme (Scheduled Backup),** verilerin önceden tanımlanmış zamanlarda otomatik olarak kopyalanması için planlanan bir yedekleme işlemidir. PostgreSQL, zamanlanmış yedeklemeleri doğrudan desteklemez, ancak işletim sistemi seviyesinde zamanlanmış görevler (cron jobs) ile bu işlevsellik dolaylı olarak sağlanabilir.
- **Coğrafi Olarak Dağıtılmış Yedekleme (Geographically Distributed Backup),** verilerin felaket ve arıza durumlarında erişilebilirliğini sağlamak için farklı konumlarda yedeklerinin oluşturulması işlemidir.

- **PITR,** veri tabanını belirli bir tarih ve saate kadar olan verileri içerecek şekilde geri yükleyebilme işlemidir.
- **Bulut Depolama Entegrasyonu (Cloud Storage Integration),** veri yedekleme, saklama ve erişim için sistemlerin ve uygulamaların bulut tabanlı depolama hizmetleriyle birleştirilmesi sürecidir.

# Geliştirme Ortamında PostgreSQL'den CockroachDB'ye Geçiş

PostgreSQL ile çalışan bir uygulamanın geliştirme ortamında CockroachDB'ye nasıl geçirilebileceği ve geçiş esnasında karşılaşılabilecek sorunlar aşağıdaki gibidir:

## 1. Docker ile CockroachDB Kurulumu

**Adım 1:** Docker kullanılan bir projede, docker-compose.yaml dosyasına CockroachDB düğümlerini eklemek gerekir.

```
roach1:
    image: cockroachdb/cockroach:v23.1.13
    hostname: roach1
    command: start --insecure --join=roach1,roach2,roach3
    volumes:
      - ~/DockerVolumes/tedarik/cockroach/node1:/cockroach/cockroach-data
      - ./create-dbs.sql:/docker-entrypoint-initdb.d/create-dbs.sql
    ports:
      - "26257:26257"
      - "8082:8080"
    restart: always

roach2:
    image: cockroachdb/cockroach:v23.1.13
    hostname: roach2
    command: start --insecure --join=roach1,roach2,roach3
    volumes:
      - ~/DockerVolumes/tedarik/cockroach/node2:/cockroach/cockroach-data
    restart: always

roach3:
    image: cockroachdb/cockroach:v23.1.13
    hostname: roach3
    command: start --insecure --join=roach1,roach2,roach3
    volumes:
      - ~/DockerVolumes/tedarik/cockroach/node3:/cockroach/cockroach-data
    restart: always
```

Bu yapılandırmadan, üç tane CockroachDB düğümü (roach1, roach2, roach3) oluşur.

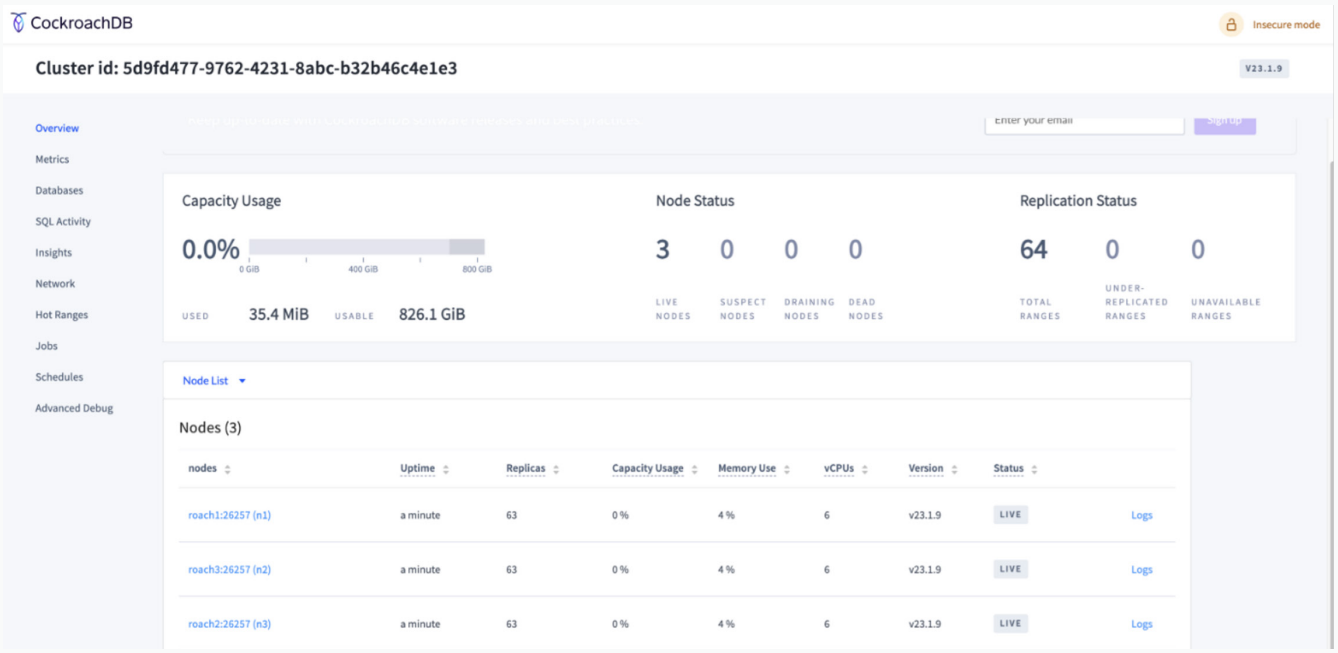
- **image: cockroachdb/cockroach:v23.1.13:** CockroachDB'nin Docker imajını ve kullanılacak olan sürümünü belirtir.
- **hostname:** Konteynerin ağ içinde tanımlanacağı adı belirler. Docker ağında konteynerler arası iletişimde kullanılır.
- **ports:** Konteyner ve lokal arasındaki port yönlendirmelerini tanımlar.
    - **"26257:26257":** CockroachDB'nin varsayılan SQL portunu (26257) lokaldeki aynı porta yönlendirir.
    - **"8082:8080":** CockroachDB'nin web arayüzü için 8080 portunu lokaldeki 8082 portuna yönlendirir.
- **volumes:** Konteyner içindeki dosya sistemini lokal ile eşleştirir.
    - **~/DockerVolumes/tedarik/cockroach/node1:/cockroach-data:** Veri tabanı verilerini saklamak için lokaldeki bir dizini konteyner içindeki **/cockroach-data** dizinine bağlar.
    - **./create-dbs.sql:/docker-entrypoint-initdb.d/create-dbs.sql**: İlk başlatma sırasında çalıştırılacak SQL scriptini (create-dbs.sql) konteyner içindeki belirli bir konuma yerleştirir.
- **command: start --insecure --join=roach1,roach2,roach3:** CockroachDB'nin güvensiz (insecure) modda birden fazla düğüm ile başlatılmasını sağlar.
- **restart: always:** Konteyner herhangi bir nedenle durdurulduğunda otomatik olarak yeniden başlatılmasını sağlar.

**Adım 2:** Docker-compose.yaml dosyasındaki değişiklikleri tamamladıktan sonra **docker compose up -d** komutunu terminalde çalıştırarak konteynerlerin başlatılması sağlanır.

**Adım 3:** Cockroach kümesi ilk kez oluştuğu için, bir sefere mahsus başlatılması gerekir (Bu herhangi bir düğümün terminalinden yapılabilir).

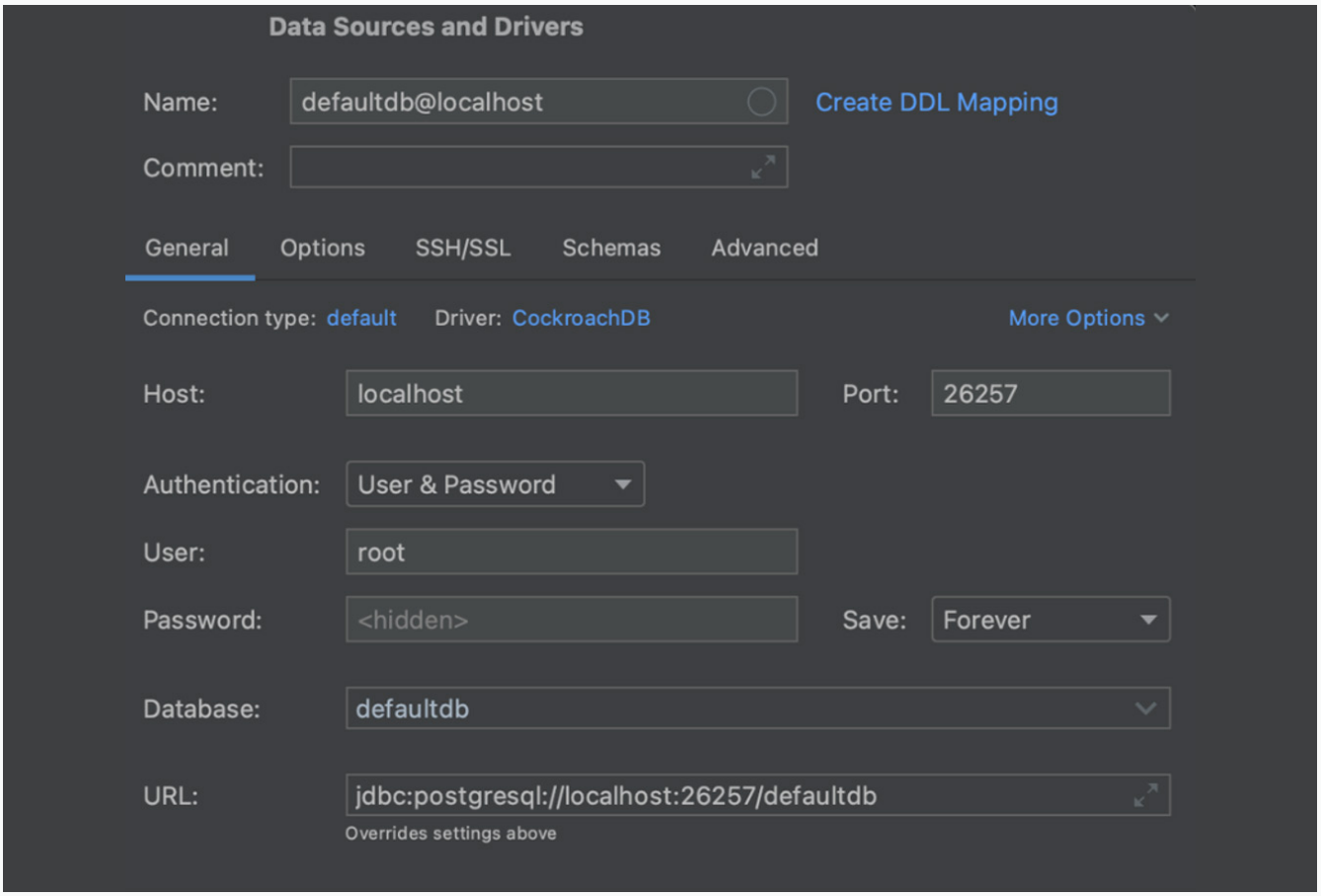
```
cockroach init --insecure --host=roach1
```

**Adım 4:** CockroachDB kullanıma hazır hale gelir ve lokalde 8082 portuna giderek CockroachDB arayüzüne ulaşılabilir.




*Şekil 20. CockroachDB Arayüzü*

**Adım 5:** Veri tabanı bağlantısı Şekil 21'deki gibi oluşturulur (Güvensiz modda başlatıldığı için şifre girilmesine gerek yoktur ve varsayılan kullanıcı adı "root"tur).




*Şekil 21. CockroachDB Veri Tabanı Bağlantı Bilgileri*

**Adım 6:** Uygulamada kullanılacak veri tabanlarını yaratmak için roach1'e volume olarak eklenen create-dbs.sql dosyasını düğümlerden birinde çalıştırmak gerekir.

- create-dbs.sql dosyası aşağıdaki gibidir:

```
create database mydb;
```

- Düğümde Çalıştırılması Gereken Komut:

```
cockroach sql --insecure --host=roach1 --execute="$(cat /docker-entrypoint-
initdb.d/create-dbs.sql)"
```

## 2. Spring Boot Uygulamasının CockroachDB ile Bağlanması

Uygulamanın properties dosyasındaki veri kaynağı (datasource) ayarlarını, CockroachDB ayarlarına uygun olacak şekilde güncellemek gerekir:

```
spring.datasource.url=jdbc:postgresql://localhost:26257/mydb
spring.datasource.username=root
```

## 3. Karşılaşılan Sorunlar

Uygulama ayağa kalkarken mevcuttaki SQL komutlarında bazı uyumsuzluklar ortaya çıkabilir. Karşılaşılan sorunlar ve çözümler aşağıdaki gibidir:

**1. Sorun:** Birincil anahtar olarak tanımlanmış alan not null kısıtlamasına (constraint) sahip olmadığı için hata alınmaktadır.

```
Caused by: org.postgresql.util.PSQLException Create breakpoint : ERROR: cannot use nullable column "urun_tanimi_id" in primary key
    at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl.java:2713)
```

**Çözüm:** Alanı not null olarak tanımlamak gerekir.

**2. Sorun:** CockroachDB'nin varsayılan hali ALTER COLUMN TYPE komutunu desteklemez. Sütun tipini değiştirmek, dağıtılmış bir veri tabanında karmaşık bir işlemdir çünkü bu değişiklik kümenin her düğümünde tutarlı ve eş zamanlı olarak gerçekleştirilmelidir. Bu tür bir değişiklik sistemin performansını etkileyebilir, bu nedenle dikkatli yapılmalıdır.

```
... 59 common frames omitted
Caused by: org.postgresql.util.PSQLException: ERROR: ALTER COLUMN TYPE from decimal to decimal is only supported experimentally
  Hint: See: https://go.crdb.dev/issue-v/49329/v23.1
--
you can enable alter column type general support by running `SET enable_experimental_alter_column_type_general = true`
```

**Çözüm:** enable_experimental_alter_column_type_general flagini ALTER COLUMN TYPE içeren sorguyu çalıştırmadan önce true olarak ayarlamak gerekir.

```
SET enable_experimental_alter_column_type_general = true
```

**3. Sorun:** PostgreSQL kullanırken tablo ilk yaratıldığında herhangi bir birincil anahtar oluşturulmamış ve daha sonrasında bu tabloya bir birincil anahtar eklenmiştir. CockroachDB, bir tabloyu oluştururken her zaman birincil anahtar bulundurmak zorunda olduğu için böyle bir durumda kendisi default olarak **rowid** isminde bir sütun ekleyerek bunu birincil anahtar olarak tanımlamaktadır. Bu nedenle yeni bir birincil anahtar tanımlamak istendiğinde birden çok birincil anahtar tanımlanamayacağı ile ilgili hata alınmaktadır.

```
Caused by: org.postgresql.util.PSQLException Create breakpoint : ERROR: multiple primary keys for table "alt_kurulusa_ait_yetkiler" are not allowed
    at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl.java:2713)
    at org.postgresql.core.v3.QueryExecutorImpl.processResults(QueryExecutorImpl.java:2401)
```

**Çözüm:** Desteklenmeyen SQL özellikleri kısmındaki 4. maddede anlatıldığı gibi, yeni bir birincil anahtar tanımlaması yapılacağında aynı transaction içinde önce mevcut birincil anahtar kaldırmalı, sonrasında yenisi eklenmelidir.

```
BEGIN;
ALTER TABLE alt_kurulusa_ait_yetkiler DROP CONSTRAINT "eski_pkey";
ALTER TABLE alt_kurulusa_ait_yetkiler ADD CONSTRAINT "yeni_pkey" PRIMARY KEY (id);
COMMIT;
```

# Sonuç ve Öneriler

CockroachDB; modern, dağıtık uygulamalar için yüksek erişilebilirlik, ölçeklenebilirlik ve coğrafi olarak dağıtılmış veri tutarlılığı sağlamak üzere tasarlanmıştır. Verilerin otomatik olarak çoğaltılması ve coğrafi olarak dağıtılması sayesinde, sistemlerin tek bir noktada hata vermesi riski azaltılmaktadır. Ayrıca düşük gecikme süreleriyle veriye hızlı erişim sağlanmaktadır. Geleneksel merkezi sistemlerin sınırlarını aşarak küresel kullanıcı tabanlarına hizmet verebilme yeteneğiyle, özellikle büyük ölçekli ve küresel dağıtım gereksinimleri olan projelerde tercih edilmektedir. Ancak, küçük ölçekli projeler veya gelişmiş analitik ve büyük veri işlemleri gerektiren, belge tabanlı veri modellerini kullanan durumlarda tam anlamıyla uygun olmayabilir. CockroachDB'nin güçlü yanları, dağıtılmış sistemlerin ihtiyaç duyduğu dayanıklılık ve ölçeklenebilirlik üzerine kurulmuştur, bu da onu kritik iş uygulamaları için tercih edilen bir çözüm haline getirmektedir.

CockroachDB, veri tabanının boyutunu basit ve zahmetsiz bir şekilde genişletme olanağı sunmaktadır. Böylece ihtiyaç duyuldukça küçük bir yapıdan büyük bir yapıya kolayca geçiş yapılabilmektedir. Şema güncellemelerinin otomatik yapılması ve verinin akıllıca dengelenmesi sayesinde yönetimsel yükler azalmakta ve sistemlerin daha verimli çalışması sağlanmaktadır. Özellikle kullanıcı trafiği veya iş yükü dalgalanmaları yaşayan uygulamalar için bu özellikler kritik öneme sahiptir.

CockroachDB'nin SQL tabanlı olması, geliştiricilerin mevcut bilgilerini kullanarak veri tabanı ile etkileşim kurmalarını sağlamakta; bu da öğrenme eğrisini azaltmaktadır. Ayrıca CockroachDB'nin PostgreSQL ile olan uyumluluğu, PostgreSQL'in geniş özellik setinden ve zengin ekosisteminden yararlanmayı mümkün kılmaktadır. Geliştiriciler, PostgreSQL için tasarlanmış uygulamaları, minimum değişiklikle CockroachDB üzerinde çalıştırabilmektedir. Bu durum, geçiş sürecini ve yeni projelerin geliştirilmesini kolaylaştırmaktadır.

CockroachDB'nin güçlü tutarlılık garantileri ve ACID uyumlu işlemleri, kritik iş yükleri için güvenilir veri yönetimi ve işlem bütünlüğü sunmaktadır. Bu özellikler, veri tabanının, dağıtık sistemlerde sık karşılaşılan zorlukların üstesinden gelmesine ve yüksek oranda tutarlı ve güvenilir bir veri tabanı çözümü sunmasına olanak tanımaktadır.

CockroachDB'nin açık kaynaklı doğası, kullanıcıların veri tabanının işleyişini incelemelerine ve ihtiyaç duyulduğunda özelleştirmelerine olanak tanımaktadır. Aktif bir topluluk tarafından desteklenmesi, sorunların hızla çözülmesine ve yeni özelliklerin düzenli olarak eklenmesine katkıda bulunmaktadır.

Sonuç olarak, veri tabanı seçim sürecinde, uygulamanın spesifik gereksinimleri, beklenen yük, ölçeklenme ve coğrafi yayılım gibi faktörlerin özenle incelenmesi gerekmektedir. Bu kapsamlı değerlendirme ile CockroachDB'nin sunduğu avantajlar ve olası kısıtlar dikkate alınarak projenin büyüklüğüne ve ihtiyaçlarına en uygun veri tabanı çözümü belirlenebilmektedir.

# Kaynakça


[1] Cockroach Labs, URL: https://www.cockroachlabs.com (Erişim Zamanı: Şubat, 23, 2024)

[2] CockroachDB: The Definitive Guide, URL: https://learning.oreilly.com/library/view/cockroachdb-the-definitive/9781098100230/ (Erişim Zamanı: Şubat, 10, 2024)

[3] Understand RAFT without breaking your brain, URL: https://www.youtube.com/watch?v=IujMVjKvWP4&t=3s (Erişim Zamanı: Kasım, 17, 2023)

[4] CAP Theorem for Distributed Systems, URL: https://www.linkedin.com/pulse/cap-theorem-distributed-systems-francis-mutiso/ (Erişim Zamanı: Ekim, 20, 2023)

[5] OpenAI, ChatGPT (Sürüm 4), URL: https://chat.openai.com/ (Erişim Zamanı: Mart, 20, 2024)

T.C. SANAYİ VE
TEKNOLOJİ BAKANLIĞI

#MİLLİ TEKNOLOJİ HAMLESİ

TÜBİTAK

İşçi Blokları Mahallesi Muhsin Yazıcıoğlu Caddesi No:51/C 06530 Çankaya/ANKARA

+90 (312) 289 92 22 - yte.bilgi@tubitak.gov.tr

TÜBİTAK - BİLGEM Yazılım Teknolojileri Araştırma Enstitüsü (YTE)