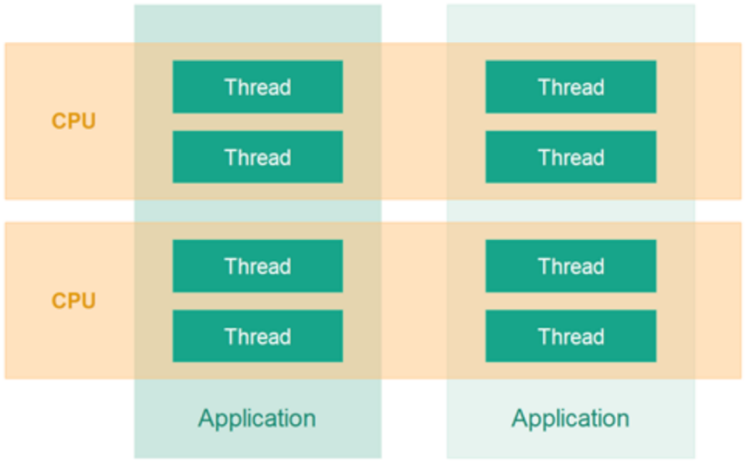
Bölüm-4

Java Çok Kanallı Programlama

Aynı uygulama içinde çalışabilen birden fazla alt işlemler açabilmeyi ifade eder. Bu kavrama Çok Kanallı Programlama (Multithread Programming) diyoruz. Çok kanallı programlama yöntemi ile geliştirilen uygulamalarda kodun farklı bölümleri aynı anda işletilebilirler. Ya da aynı kod parçası farklı veri kümeleriyle aynı anda işletilebilir.

Thread, böldüğümüz alt işlemleri ifade eder. Thread içinde bir kod parçası çalışır. Birden fazla Thread ile aynı anda birçok kod parçası eş zamanlı olarak çalıştırılabilir. Fakat, Thread diye bahsettiğimiz her iş parçası CPU demek değildir. Aynı CPU'da birden fazla Thread aynı anda çalışabilir. CPU kaynaklarını Thread'leri çeşitli zaman aralıklarında sıralı olarak işletir. Bir Thread işletilirken bir diğerine geçmek gerektiği zaman içerikleri hafızada saklanarak diğerine geçiş yapılır. Thread'ler arasında böyle geçişler yapılarak eş zamanlı çalışma sağlanır. Aynı uygulamanın Thread'leri (iş parçacıkları) farklı CPU'larda işlenebilir. Eğer bilgisayarın fiziksel olarak birden fazla CPU'su varsa bu iş parçacıkları gerçek manada eş zamanlı çalışabilir. Tek CPU'lu bilgisayarlarda eş zamanlı çalışabilmek için CPU zamanı minik dilimlere ayırır ve bu ayırdığı zaman dilimlerinden birinde iş parçacığının bir kısmını çalıştırır, ardından bir başka iş parçacığına geçer. Bunu o kadar hızlı yaparki tek CPU'lu bilgisayarlarda bile eş zamanlı çalışıyormuş hissi verir.



Geçmişte tek CPU'lu bilgisayarlar eş zamanlı programlama yöntemi olmadığı zamanlarda aynı anda sadece tek bir uygulama çalıştırabilmekteydiler. Yani bizler müzik dinliyorken, Word dokümanı yazamıyorduk. Veya konsolda kod yazıyorken başka bir işlem yapamıyorduk.

Multitasking

Aynı anda eş zamanlı olarak birçok programın CPU ve işletim sistemi kaynaklarını kullanarak işletilebilmesidir.

Multithreading

Aynı uygulamada, yani yazılımda, birden fazla alt iş parçacığının yine CPU ve işletim sistemi kaynaklarını kullanarak çalıştırılabilmesidir.

Neden Multithreading?

- Performans kazancı sağlamak
- CPU'nun (işlemci) zamanını verimli kullanabilmek
- CPU'nun hızından faydalanmak

Multithreading Zordur!

Multithread (çok kanallı) programlama yöntemiyle performanslı uygulamalar yazma fikri kulağa çok hoş gelebilir. Gerçekten de çok kanallı programlama yaparak uygulamanızın performansını arttırabilirsiniz. Fakat, bu yöntemle programlama yapabilmek bir o kadar zordur. Çünkü, birden fazla olan bu iş parçacıkları aynı hafıza kaynağını kullanıp okuma ve yazma işlemlerini tek bir hafızaya yapmaktadırlar. Her ne kadar işlemci sayısı artıp işlemciler bağımsız çalışma birimleri olsa da hafıza tek bütün halindedir. Ve bu hafızanın yazma ve okuma işlemlerinde kontrollü bir şekilde koordine edilmesi gerekir. Düzgün organize edilmemiş çok kanallı programlar kararsız bir şekilde çalışır. Hafızada tutulan verilere yanlış yazma işlemleri uygulayarak tutarsız veriler oluşmasına sebep olurlar. Böyle bir durumda program istenilen şekilde çalışmaz ve tek thread'li (iş parçacığı) yapılarda görmediğimiz sorunlar ortaya çıkabilir. Yazılımcı birden fazla CPU'ya sahip bir donanımda her iş parçacığını bağımsız olarak farklı CPU'larda çalıştırsa bile hafıza ortak olduğu için buraya yapılacak erişimleri koordine edecek kodları yazmalıdır. Hafızaya yapılacak yazma ve okuma işlemlerini sıralı erişime koymalıdır.

CPU Verimli Kullanmak

Diyelimki diskteki dosyaları okuyup işleyen bir program yazıyoruz. Bu yapılan işlemlerin süreleri aşağıdaki gibi ardışıl şekilde olsun.

A dosyasını okumak - 5 saniye
A dosyasını işlemek - 2 saniye
B dosyasını okumak - 5 saniye
B dosyasını işlemek – 2 saniye

Toplam süre: 14 saniye

Dosyalar sistemden okuma yapılırken CPU neredeyse tamamıyla boş durumdadır. CPU bu zamanı daha iyi değerlendirip birtakım işlemler yapabilirdi. İşte çok kanallı programlama ile bu boş süreleri değerlendirip CPU'yu daha verimli kullanabiliriz.

Bu tarz IO işlemlerinde, ki biz IO işlemleri giriş-çıkış işlemleri diyoruz, CPU boş bir şekilde beklemektedir.

Oysa CPU bu zamanda bir yandan dosya işleme işlemini başlatsa daha verimli çalışmış olurdu. Yazılımımızda daha performanslı çalışırdı.

A dosyasını okumak (5 saniye)
B dosyasını okumak (5 saniye) + A dosyasını işlemek (2 saniye)
B dosyasını işlemek (2 saniye)

Toplam süre: 12 süre

Görüldüğü gibi süre anlamında kazanç sağlamış oluyoruz.

Not: gerçekten çok kanallı programlama için ciddi bir sebep olmadan bu yönteme başvurmamak lazım. Performans kazancı gerektiren durumlarda bunu deneyebilirsiniz. Örneğin yoğun miktar veriyi işleyip veri kümesi içinden aradığınızı bulmanız gerektiğinde bu yönteme başvurulabilir. Böyle geçerli bir sebep olması gerekir.

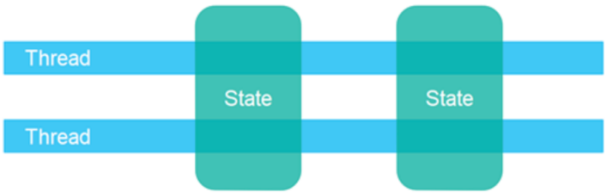
Not: Birden fazla iş parçacığı (Thread) olan bir uygulama yazarken, iş parçacıklarının ortak olarak kullandıkları kaynakları senkronize edip sıralı erişime açmak gerekir. Buradaki ortak kaynak genellikle aynı hafıza bölgesindeki bir veri parçası olabilir. Bir iş parçacığı ortak kaynağı kullanıyorken bu durumda diğerleri o işini bitirene kadar beklemelidir. Aksi durumda hafızaya yazma ve okuma esnasında tutarsızlıklar oluşabilir. Bunların tespiti ve düzeltilmesi ciddi zaman ve iş gücü maliyetine yol açabilir.

Not: Bir iş parçacığı CPU'da işletilirken birtakım kaynakları kullanır. İş parçacığının kullandığı veri, o anda kod icra edilirken kullanılan tüm kaynaklar iş parçacığının kullanımındadır. CPU ise zamanı dilimlere ayırarak aynı anda bir iş parçacığını işletebilir. Bu nedenle CPU iş parçacıklarının kullandığı bu kaynakları bir yere kaydedip, yeni iş parçacığını son haliyle yükleyip çalıştırmalı ve ardından bu işlemi iş parçacıkları arasında sürekli tekrarlamalıdır. Bu yorucu ve maliyetli bir işlemdir. Bu nedenle CPU'yu "context switching" darboğazı yaratmamak gerekir. Bu nedenle optimum sayıda Thread açılmalıdır. Yüzlerce iş parçacığı açmak uygulamayı hızlandırmaz aksine yavaşlatır. Genelde 5-10 iş parçacığı (Thread) iş görebilir. Bu birazda deneyimsel bir durumdur. Uygun sayıyı yazılımcı deneyerek bulabilir.

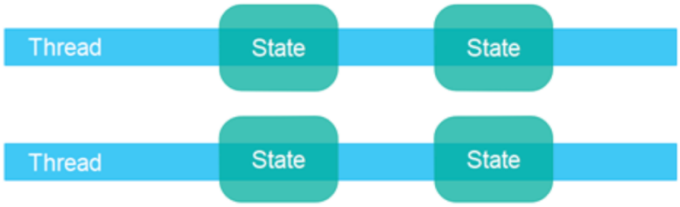
Paylaşımlı Durum ve Ayrık Durum

Çok kanallı programlamanın en önemli noktasından biri de kullanılan kaynağın paylaşılıp paylaşılmadığıdır. Yani bazı modellerde iş parçacıkları ortak bir "state"'i (durum) kullanıyor olabilirler. Yani "state"'den kastımız ortak bir veri kaynağını kullanıyorlardır. Örneğin, aynı

diziye eleman eklemeye çalışan birden fazla iş parçacığı olduğunu düşünün burada tümü ortak bir kaynağı, yani ortak bir “state”’i kullanıyorlardır.

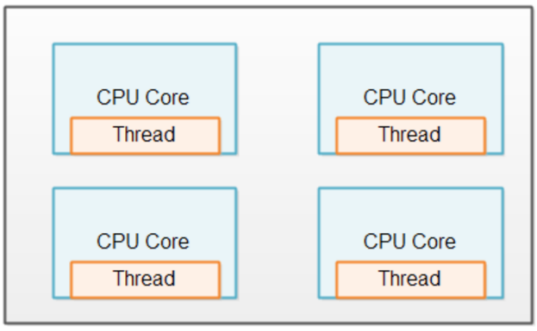


Ya da bazı problemlerde birden fazla iş parçacı bağımsız olarak ayrı ayrı “state”2lerde çalışabilirler. Yan, iş parçacıkları işleyecekleri verileri alırlar ve iş parçacıkları arasında herhangi bir senkronizasyon gerekmeden çalışabilirler. Bu durumda aslında çok kanallı programlama yapmak daha performanslı olabilir. Ayrıca, senkronizasyon için ekstra bir çaba harcanmaz. Daha az sorun çıkar. Örneğin elimizde 100 bin elemanlık bir dizi olsun ve bu dizinin toplamını hesaplamak isteyelim. Bu durumda 100 binlik diziyi 10 parçaya ayırıp her veri parçasını bir iş parçacığının işlemesini sağlayabiliriz. Bu iş parçacıkları kendi veri kümesinin toplamanı bulur. Ardından 10 iş parçası işini bitirdiğinde 10 sonucu bir araya getirip toplama işlemini hızlandırmış oluruz.

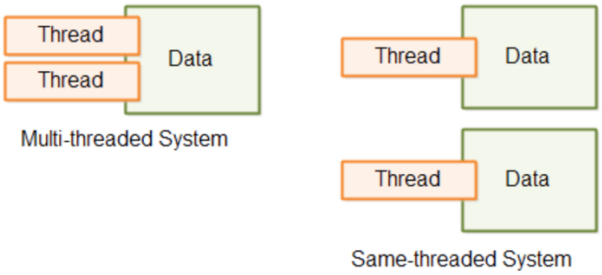


Her bir iş parçacığı için bir CPU

Eğer birden fazla CPU içeren bir donanımınız varsa her CPU’ya bir iş parçacığı atanır. Bu durumla CPU’lar verimli bir şekilde kullanılır.



Yukarıdaki duruma ek olarak ortak paylaşılan herhangi bir kaynak (state) yoksa her CPU’ya atanan iş parçacığı birbirleri arasında senkronizasyona ihtiyaç duymazlar. Bu duruma “Same-Threaded System” denilmektedir. Eğer iş parçacıkları ortak paylaşımlı bir kaynak (state) kullanıyorsa bu durumda iş parçacıklarının bu ortak kaynaklara erişimi senkronize edilmelidir. Bu duruma “Multi-Threaded System” denilmektedir.



Java Thread'lerin Oluşturulması ve Kullanılması

Java'da herhangi bir uygulamayı çalıştırdığımızda varsayılan olarak ana (main) bir iş parçacığı (Thread) ayağa kaldırılır. Bu ana iş parçacığına ek olarak yazılımcılarda alt iş parçacıkları çalıştıralabilir. Bundan zaten bahsetmiştik. Şimdi Java dilinde Thread nasıl tanımlanır ve çalıştırılır onu inceleyelim.

Java'da iş parçacığı oluşturmak için "Thread" isminde bir sınıf bulunmaktadır. Böylece, basit anlamda iş parçacığı açmış oluruz. Tabi, unutmamak lazım Thread açmak sisteme maliyeti olan bir iştir. Sürekli Thread üretmek kaynak kullanımını olumsuz etkiler. Bu nedenle bu sorunu çözmek Thread Pooling kavramı vardır. Bu maliyetli nesneler ilk başta belli bir miktarda yaratılır ve hazır durumda olacak şekilde havuza konulur. Thread ihtiyacı olanlar bu havuzdan bir Thread'i kullanır ve sisteme geri iade eder. Böylece, performans kazancı yanı sıra kaynak kullanımı da iyi bir hale getirilir.

```
Thread thread = new Thread();
```

Yukarıda görüldüğü gibi "Thread" sınıfından bir nesne oluşturup bir iş parçacığı üretmiş olduk. Bu iş parçacığını çalışmaya başlatmak için "start" fonksiyonunu çağırmamız gerekecektir.

```
thread.start();
```

Böylece, iş parçacığımız işini bitirene kadar çalışmaya devam edecektir. Fakat, yukarıdaki örnekte iş parçacığının çalıştıracağı bir kod parçası vermedik. Bunu verebilmenin iki yolu vardır.

1. "Thread" sınıfından kalıtım alan bir alt sınıf yaratıp, onun "run" fonksiyonunu override (ezme) etmek gerekir.
2. "Runnable" interface'den kalıtım alan bir alt sınıf yaratmak ve "run" fonksiyonunu override etmek, ardından "Thread" sınıfının kurucusuna nesne olarak göndermek.

```
public class SimpleThread extends Thread {

        @Override
        public void run() {
```

```
                // o an çalışan Thread'in ismini alıyoruz.
                String threadName = Thread.currentThread().getName();

                System.out.println("My summation " + threadName + " is
started!");

                int total = 0;
                for(int i=0; i < 1000; i++)
                {
                        total += i;
                }

                System.out.println("Total: " + total);
        }
}
```

Yukarıda “SimpleThread” isminde Java’nın “Thread” sınıfından kalıtım alan bir sınıf tanımladık. Bu sınıf içindeki “run” metodu içine Thread’de çalıştırmak istediğimiz kodları yazıyoruz. Bu kod parçası işletim sistemi düzeyinde herhangi bir CPU’da eş zamanlı olarak çalıştırılacaktır. Unutulmamalı ki yarattığımız iş parçacığı hazırladığımız Java konsol uygulamasının bir alt iş parçacığıdır.

```
SimpleThread simpleThread = new SimpleThread();
simpleThread.start();
```

“SimpleThread” sınıfından bir nesne üretiyoruz. Ardından, “start” fonksiyonunu çağırdığımızda işletim sistemi bize bir Thread kaynağı yaratıyor ve “SimpleThread” sınıfında override ettiğimiz “run” fonksiyonu işletilmeye başlanıyor. “run” fonksiyonu içindeki kodlar artık ayrı bir Thread içinde işlem görmeye başlıyorlar. Aynı şekilde bir nesne daha üretip “start” dediğimizde yeni bir Thread daha oluşturulup başka bir iş parçacığı oluşturulur.

```
SimpleThread simpleThread2 = new SimpleThread();
simpleThread2.start();
```

Sonuçlar aşağıdaki gibidir. Görüldüğü üzere iki farklı Thread aynı anda işletilmiştir.

```
My summation Thread-0 is started!
Total: 499500
My summation Thread-1 is started!
Total: 499500
```

Runnable interface ile Thread Kullanımı

Thread’lere Runnable tipinde nesne vererek kod parçasını bir iş parçacığı olarak işletebiliriz.

```
public class SimpleRunnable implements Runnable {

	@Override
	public void run() {

		// o an çalışan Thread'in ismini alıyoruz.
		String threadName = Thread.currentThread().getName();

		System.out.println("My summation " + threadName + " is
started!");

		int total = 0;
		for(int i=0; i < 1000; i++)
		{
			total += i;
		}

		System.out.println("Total: " + total);
	}

}
```

Yukarıda “Runnable” interface’den kalıtım alan “SimpleRunnable” isminde bir sınıf tanımladık. Bu sınıf içindeki “run” metodunu override ederek iş parçacığı içinde çalıştırılacak olan kodu yazıyoruz.

```
// Runnable interface'den kalıtım almış olan "SimpleRunnable" sınıfından
bir nesne oluşturuyoruz.
SimpleRunnable simpleRunnable = new SimpleRunnable();

// Runnable tipindeki nesneyi Thread kurucusuna gönderiyoruz.
Thread simpleThread3 = new Thread(simpleRunnable);

// start fonksiyonunu çağırdığımızda "SimpleRunnable" sınıfı içindeki "run"
fonksiyonu işletilecektir.
simpleThread3.start();
```

“Runnable” tipindeki sınıfımızdan bir nesne oluşturuyoruz. Ardından, Thread sınıfından bir nesne oluşturup kurucu metodun içine “Runnable” tipte oluşturduğumuz nesneyi gönderiyoruz. Bununla birlikte 3 Thread (iş parçacığı) tanımlamış oluyoruz.

```
My summation Thread-0 is started!
My summation Thread-1 is started!
Total: 499500
Total: 499500
My summation Thread-2 is started!
Total: 499500
```

3 iş parçacığının sonuçları yukarıdadır.

Not: Thread.currentThread() ile o anda aktif olarak çalışan iş parçacığının referansını alabiliyoruz. Burada aldığımız Thread nesnesi iş parçacığı içinde çalıştırılan kodu ifade eder.

Thread'i beklemeye almak

Java'da bir iş parçacığını belli bir süre bekletmek istersek, Thread sınıfına ait "sleep" fonksiyonunu kullanabiliriz.

"sleep" fonksiyonu milisaniye cinsinden bir değer bekler. Yani örneğin 3000 değeri 3 saniyeye karşılık gelmektedir.

```
try {
    Thread.sleep(10L * 1000L);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Yukarıda Thread.sleep ile iş parçacığı 10 saniye beklemeye alınmıştır. "sleep" fonksiyonu hata fırlatabilen bir fonksiyon olduğundan önceki konularda da görmüştük try-catch bloğu içinde kontrol edilmelidir veya fırlattığı hatayı "throws" anahtar kelimesi ile bir üste doğru fırlatmamız gerekmektedir.

Thread'i durdurmak

Bir iş parçacığı çalışmaya başladığında bir CPU tarafından işletilmeye başlanır. İş parçacığını durdurmak için Thread sınıfı içinde "stop" fonksiyonu vardır. Fakat, bu fonksiyon iş parçacığını durdurma garantisi vermez. O nedenle Thread içinde çalışacak kodu tasarlarken çalışan kod parçasının nasıl durdurulacağını yazılımcı kendisi kodlamalıdır. Bu nedenle Thread sınıfında veya Runnable interface'den türemiş bir sınıf içinde boolean bir değer tutarak kod parçasını sonlandırmayı garanti altına alabiliriz.

Bunu bir örnek ile inceleyelim.

```
public class ThreadSleeper {

        public static void sleep(long milliseconds) {

                try {
                        Thread.sleep(milliseconds);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
```

```
}
```

“ThreadSleeper” isminde bir sınıf oluşturduk. Bu sınıf içindeki “sleep” fonksiyonu iş parçacığını belli bir süre bekletmeyi sağlamaktadır.

Ardından “SimpleRunnable” sınıfımızdaki “run” metodu içine yazdığımız kod parçasında güncelleme yapıyoruz. Çalışan kod parçacığını durdurmak için “live” isminde boolean bir değişken tanımladık. Bu değişken ile Thread içinde çalışan çalışmayı durduruyoruz.

```
public class SimpleRunnable implements Runnable {

	private boolean live = true;

	@Override
	public void run() {

		// o an çalışan Thread'in ismini alıyoruz.
		String threadName = Thread.currentThread().getName();

		System.out.println("My summation " + threadName + " is
started!");

		int total = 0;
		while(live)
		{
			total += 1;

			// yarım saniye bekletiyoruz.
			ThreadSleeper.sleep(500);
		}

		System.out.println("Total: " + total);
	}

	public void stop() {
		this.live = false;
	}

}
```

Thread’i start fonksiyonu ile çalışmaya başlatıyoruz. Ardından ana programı 10 saniye bekletiyoruz. Bekleme bitince “SimpleRunnable” içindeki “stop” fonksiyonunu çağırıp çalışan kod parçasını durduruyoruz.

```
// Runnable interface'den kalıtım almış olan "SimpleRunnable" sınıfından
bir nesne oluşturuyoruz.
SimpleRunnable simpleRunnable = new SimpleRunnable();
```

```
// Runnable tipindeki nesneyi Thread kurucusuna gönderiyoruz.
Thread simpleThread3 = new Thread(simpleRunnable);

// start fonksiyonunu çağırdığımızda "SimpleRunnable" sınıfı içindeki "run"
fonksiyonu işletilecektir.
simpleThread3.start();

ThreadSleeper.sleep(10000);

simpleRunnable.stop();
```

Kritik Bölümler (Critical Sections) ve Veriye Erişim Yarışması (Race Condition)

Tek Thread şeklinde çalışan uygulamalar hafıza bölgesine ve diğer sistem kaynaklarına erişirken herhangi bir senkronizasyon ihtiyacı bulunmaz. Program çalışır ve kaynakları kullanır, ardından kaynakları sisteme iade eder. Fakat, çok kanallı programlamada birden fazla iş parçacığı (Thread) aynı anda ortak bir kaynağa erişmeye kalkışırlarsa bunları sırayla erişim vermek gerekecektir. Bir iş parçacığı kaynağı kullanıyorken diğerleri onu beklemelidirler. İşte birden fazla iş parçacığının kullandığı bu paylaşımlı ortak alanlara “Critical Sections” adı verilmektedir. Buna en güzel örnek bir değişkenin değerinin değiştirilmesidir. Çünkü, değişkenin değerinin değişmesi demek hafıza bölgesinde bir değişim yapmak demektir. Böyle ortak bir kaynağı kullanan Thread’ler okuma ve yazma yaparken bu “Critical Section”’a sırayla erişmeleri gerekir.

```
public class QMatic implements Runnable {

        private int orderNo;

        public QMatic() {
                this.orderNo = 0;
        }

        @Override
        public void run() {

                // a little bit delay to see race condition!
                ThreadSleeper.sleep(50);

                // Critical section for all threads!
                this.orderNo = this.orderNo + 1;

                StringBuilder builder = new StringBuilder();
                builder.append(Thread.currentThread().getName());
                builder.append(" thread got ");
                builder.append(this.orderNo);
```

```
                builder.append(" from Qmatic!");

                System.out.println(builder.toString());
        }
}
```

Yukarıdaki örnekte bir banka şubesinde sıra numarası alınan bir cihaz olduğunu düşünün. Gelen müşteri sıraya göre bir fiş alıp numarasını görmektedir. “QMatic” ismindeki sınıfın içindeki “run” metodu içinde “orderNo” isimli değişkenin değeri değiştirilmektedir. Müşteriler cihaz butonuna basınca sıra numarası bir artmaktadır. Şube içinde tek cihaz üzerinden tüm müşteriler numara alırsa sorun yoktur. Tek cihaz tek Thread! Fakat, şube müdürü verimlilik adına şube içine bu cihazlardan bir tane eklettiğini düşünün. 2 tane numara alma cihazımız oldu. Fakat, 2 cihaz olsa da sıra numaralarını yine ardışıl ve tutarlı vermemiz gerekiyor. 2 Thread 1 Critical Section! 2 cihaz demek bizim için 2 Thread demek, sıra numarası ise bizim için “Ciritical Section” demektir. Buradaki sıra numarasını düzgün yönetmemiz gerekiyor.

Yukarıdaki örnek kodda “Critical Section” olan kısım `this.orderNo = this.orderNo + 1;` ifadesidir. Bu tek satırlık bir komut gibi gözükebilir ama arka planda işlemci seviyesinde bu işlem birden fazla adımdan oluşmaktadır.

Qmatic1: this.orderNo değişkenindeki değeri hafızadan 0 olarak okur.
Qmatic2: this.orderNo değişkenindeki değeri hafızadan 0 olarak okur.
Qmatic2: Elindeki 0 değerine 1 ekler.
Qmatic2: “orderNo” değişkenini 1 olarak değiştirir. orderNo değişkeninin değeri şuan 1’dir.
Qmatic1: Elindeki 0 değerine 1 ekler.
Qmatic1: “orderNo” değişkenini 1 olarak değiştirir. orderNo değişkeninin değeri şuan 1’dir.

Görüldüğü gibi iş parçacıkları (Thread’ler) için okuma ve yazma işlemleri senkronize edilmezse aynı sıra numarasını birden fazla kişiye verebiliriz. İşte bu olaya “Race Condition” denilmektedir. Çünkü, Thread’ler ortak kaynağa erişmek için birbiriyle yarışır. İlk erişen işlemlerini yapmaya başlar, fakat diğer Thread’ler onun işini bitirmesini beklemezlerse yukarıdaki gibi tutarsız durumlar oluşur. Bu nedenle Thread’lerin ortak kaynağa erişimi sıralı olmalıdır.

Sonuçlar :

```
Thread-2 thread got 1 from Qmatic!
Thread-9 thread got 1 from Qmatic!
Thread-1 thread got 1 from Qmatic!
Thread-0 thread got 2 from Qmatic!
Thread-7 thread got 2 from Qmatic!
Thread-3 thread got 2 from Qmatic!
Thread-4 thread got 2 from Qmatic!
Thread-8 thread got 2 from Qmatic!
Thread-5 thread got 4 from Qmatic!
Thread-6 thread got 4 from Qmatic!
```

Java “synchronized” Anahtar Kelimesi

Yukarıdaki gibi “Ciritical Section” olan kod bölgelerinde “Race Condition” durumuna engel olmak için kullanılabilecek yöntemlerden biri de “synchronized” anahtar kelimesidir. Bu anahtar kelime ile “Critical Section” kod bölgesini Thread’ler arasında sıralı erişime açabilirsiniz.

“synchronized” anahtar kelimesini bir değişkene, bir bloğu parçasına veya metoda verebilirsiniz. Yukarıdaki örneğimizi şimdi “Thread Safe” bir hale getirelim.

```
public class QMatic implements Runnable {

        private int orderNo;

        private Object LOCK = new Object();

        public QMatic() {
                this.orderNo = 0;
        }

        @Override
        public void run() {

                // a little bit delay to see race condition!
                ThreadSleeper.sleep(50);

                // Critical section for all threads!

                synchronized (LOCK) {
                        this.orderNo = this.orderNo + 1;

                        StringBuilder builder = new StringBuilder();
                        builder.append(Thread.currentThread().getName());
                        builder.append(" thread got ");
                        builder.append(this.orderNo);
                        builder.append(" from Qmatic!");

                        System.out.println(builder.toString());
                }

        }
}
```

QMatic isimli “Runnable” interface’den türemiş sınıfımızda “LOCK” isminde Object türünden bir kilit nesnesi oluşturuyoruz. Ardından “Critical Section” olarak belirttiğimiz tüm Thread’lerin ortak kullandığı “orderNo” değişkeniyle işlem yapan kod bloğunu “synchronized” anahtar kelimesiyle korumaya alıp, Thread’ler için sıralı erişime açıyorum.

Eğer bir Thread “Critical Section” olarak işaretlediğim kod bloğuna girip kaynakları kullanmaya başlarsa diğer Thread’ler o işini bitirene kadar beklemek zorundadırlar.

```
Thread-9 thread got 1 from Qmatic!
Thread-1 thread got 2 from Qmatic!
Thread-3 thread got 3 from Qmatic!
Thread-7 thread got 4 from Qmatic!
Thread-2 thread got 5 from Qmatic!
Thread-4 thread got 6 from Qmatic!
Thread-5 thread got 7 from Qmatic!
Thread-0 thread got 8 from Qmatic!
Thread-6 thread got 9 from Qmatic!
Thread-8 thread got 10 from Qmatic!
```

Yukarıda “synchronized” olarak belirttiğimiz kod bloğunu bir metod içine alsaydık. Aşağıdaki gibi yapabilirdik.

```
private synchronized void increment() {

        this.orderNo = this.orderNo + 1;

        StringBuilder builder = new StringBuilder();
        builder.append(Thread.currentThread().getName());
        builder.append(" thread got ");
        builder.append(this.orderNo);
        builder.append(" from Qmatic!");

        System.out.println(builder.toString());
}
```

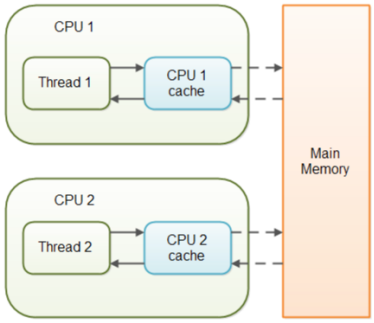
Java “volatile” Anahtar Kelimesi

“volatile” anahtar kelimesi değişkenin sakladığı değerin Thread’ler tarafından okunmaya çalışıldığında hepsinde aynı değerin okunacağının garantisini verir. Bilgisayar mimarisinde ana hafıza bölgemiz vardır. Biz program çalıştığı süre boyunca işlediği verileri ve programın komutlarını bu ana hafıza bölgesinde saklarız. Bu hafıza bölgesi RAM diye bilinir. Ayrıca, bilgisayar mimarisinde işlemciler (CPU) vardır. CPU ile hafıza bölgesi sürekli haberleşme halindedir. Sıkı bir iletişim trafiği vardır.

Bu nedenle aşağıdada görüldüğü gibi CPU tarafında küçük hafıza bölgeleri bulunur. Bu hafıza bölgeleri sık kullanılan değişken değerlerini buraya cache’lerler. Böylece sürekli ana hafızaya giderek zaman kaybetmek yerine daha hızlı işlem görürler.

“volatile” anahtar kelimesi ile işaretlenmiş bir değişkenin değerine erişmek gerektiğinde direkt olarak ana hafızadan alınacağını ve ilgili değişkene yazma işlemi uygulanacaksa yine direkt olarak ana hafıza bölgesine yazılacağını belirtmiş oluruz. Böylece bu değişken üzerinde

işlem yapan tüm Thread'ler aynı değeri görecekleri garanti altına alınmış olunur. Normalde "volatile" demeseydik değişkenimiz CPU cache bölgesinden okunabilirdi. Bu durumda tutarsız durumlar oluşabilirdi. Kritik değerler için "volatile" anahtar kelimesi kullanabilirsiniz.




```
private volatile int orderNo;
```

Yukarıda QMatic örneğimizde "volatile" anahtar kelimesinin kullanımını gördük.

Kilit Nesneler ile Çok Kanallı Programlama (Locks in Java)

"synchronized" anahtar kelimesi ile "Critical Sections" kod bölgelerini sıralı erişime açıp Thread'lerin ortak kullandığı bu alanı düzgünce kullanabilmelerini sağlıyorduk. Aynı şekilde "Lock" yani kilit mekanizmaları ile belli kod bloklarını "synchronized" gibi sıralı erişime açabiliriz.

Bunun için "ReadWriteLock" sınıfını kullanarak hem okuma hem de yazma işlemleri için sıralı erişim olanağı veren bir örnek geliştireceğiz. Bunun için "SimpleCounter" isminde "Runnable" interface'den kalıtım alan bir sınıf tasarlıyoruz.

```
public class SimpleCounter {

	private volatile int counter;

	private final ReadWriteLock readWriteLock = new
ReentrantReadWriteLock();

	public SimpleCounter() {
		this.counter = 0;
	}

	public void incrementCounterWithThreadSafety(String threadName) {

		this.readWriteLock.writeLock().lock();

		try {
			this.counter++;
```

```
                        System.out.println("Counter was updated to '" +
this.counter + "' from " + threadName);
                }
                finally {
                        this.readWriteLock.writeLock().unlock();
                }

        }


        public int readCounterWithThreadSafety() {

                this.readWriteLock.readLock().lock();

                try {
                        return this.counter;
                }
                finally {
                        this.readWriteLock.readLock().unlock();
                }

        }
}
```

Yukarıdaki örnekte “readWriteLock” isminde bir nesne oluşturuyoruz. Bizim kod içindeki kilit mekanizmamızı bu nesne yönetecektir.

“incrementCounterWithThreadSafety” metodu içinde değişkenin değerini güncelleyeceğimiz için bir yazma işlemi yapacağız bu nedenle aşağıdaki komutla bir yazma kilidi alıyoruz.

```
this.readWriteLock.writeLock().lock();
```

Ardından koruma altına almak istediğim kod bloğunu bir “try” bloğu içine yerleştiriyorum. İşlemler bitince ise aşağıdaki gibi kilidi serbest bırakıyorum.

```
finally {
                        this.readWriteLock.writeLock().unlock();
                }
```

Hatırlayacaksınızdır, finally anahtar kelimesi ile “try” bloğunda hata olsun veya olmasın mutlaka çalıştırılırdı. Mutlaka kilidi öyle ya da böyle serbest bırakıyorum. Kilidi serbest bırakamazsam sıkıntılı sorunlarla boğuşabiliriz.

“readCounterWithThreadSafety” metodunda ise “counter” isimli değişkenin değerini okuyacağız ve burada da okuma yaptığımız kod bloğunu korumaya almak istiyoruz diyelim. Bu durumda ise okuma kilidini alıp kullanacağız ve sisteme geri iade edeceğiz.

```
this.readWriteLock.readLock().lock();
```

işimiz bitince aşağıdaki gibi finally bloğu içinde mutlaka kilidi serbest bırakıyoruz.

```
finally {
                        this.readWriteLock.readLock().unlock();
                }
```

İş Parçacığı Havuzları (ThreadPooling)

Thread yaratmak cidden maliyetli bir olaydır. Her Thread için sistemde belli bir kaynak ayrılır. Bu kaynaklar CPU, Hafıza gibi önemli olanlardır. Uygulamamız çalışırken belli miktarda bir Thread ile sınırlandırmak isteyebiliriz. Bu nedenle Thread havuzu oluşturup bu havuzu önceden oluşturulmuş ve kullanıma hazır Thread nesneleri ile doldururuz. Böylece, performans kazanımı ve sistem kaynaklarının verimli kullanımını sağlayabiliriz.

```
ExecutorService executor = Executors.newFixedThreadPool(15);
```

Yukarıdaki Java'da hazır bulunan "Executors" sınıfındaki "newFixedThreadPool" metodunu çağırarak bir Thread havuzu oluşturabilirsiniz. Ardından bize 15 Thread'in kullanıma hazır halde bulunduğu bir havuz oluşturup verecektir. Bu fonksiyon ayrıca bize bu havuz üzerinde Thread kullanımını yönetecek "ExecutorService" tipinde bir nesne verecektir. Bu nesne üzerindeki "execute" fonksiyonuyla havuzdaki bir Thread'i kullanıp işimiz bitince tekrar sisteme iade edeceğiz.

```
ExecutorService executor = Executors.newFixedThreadPool(15);

QMatic qmatic = new QMatic();

for(int i=0; i < 100; i++) {
 executor.execute(qmatic);
}
```

Yukarıdaki örnekte "QMatic" isminde önceden de kullandığımız Runnable tipinde sıra numarası veren sınıftan bir nesne yaratıyoruz. Bu kod parçasını "execute" fonksiyonuyla havuzdaki bir Thread'i kullanarak çalıştırıyoruz. Görüldüğü gibi havuz 15 kapasiteli olmasına rağmen döngüde 100 kez Thread kullanma talebi gelmiş. Eğer havuzda uygun boş bir Thread yoksa beklemede kalacaktır. Havuzdan ilk boşa çıkan Thread nesnesini alıp çalışacaktır.

SOLID Prensipler

Daha temiz, okunabilir, bakımı kolay ve esnek bir yapıya sahip kodlar yazabilmek için prensipler bütünüdür. Her prensibin özel bir ismi vardır. Böylece İngilizce isimlerin baş

harflerin bir araya gelmesiyle SOLID olarak isimlendirilmiştir. Bu kavram ilk kez Robert C. Martin isimli yazılımcının “Design Principles and Design Patterns” isimli kitabında geçmiştir.

S: Single Responsibility Principle (Tek Sorumluluk)
O: Open/Closed Principle (Açık-Kapalı Olma Durumu)
L: Liskov Substitution Principle (Liskov Prensibi)
I: Interface Segregation Principle (Arayüzlerin Ayrılması Prensibi)
D: Dependency Inversion Principle (Bağımlılıkları Tersine Çevirme)

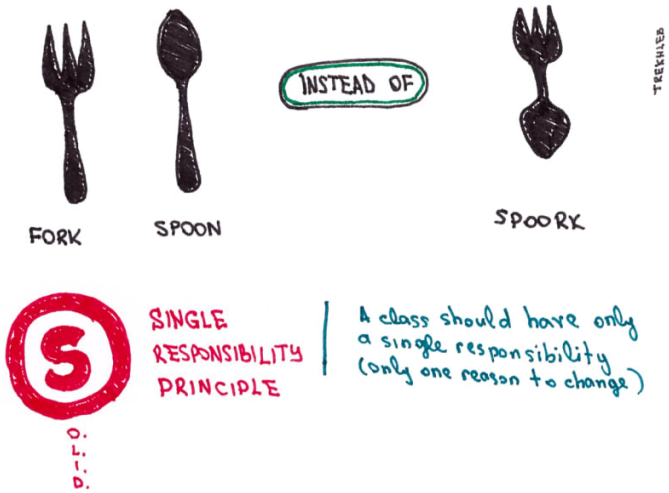
Şimdi her prensibi ayrı ayrı inceleyelim. Konu sonunda da hepsini içeren bir örnek uygulama yapmış olacağız.

Single Responsibility Prensibi

Yazılımları oluştururken projeler büyüdüğünde karmaşıklık boyutu da artmaya başlar. Böyle durumlarda sınıfların dizaynında ve sınıfların bir araya getirdiği mimari boyutunda sorunlar ortaya çıkabilir. Proje büyüdükçe bazı sınıflar gereğinden fazla sorumluluğu fark etmeden üzerlerine çekmeye başlarlar. Bu tarz sınıflar “God Class” ismi verilir. Sorumluluğu dışında birçok iş yapıp birçok bağımlılığı bünyesinde barındırır.

Tasarladığımız sınıfların tek bir işi yerine getiriyor olması gerekir. Aynı şekilde mantıksal olarak birbirine yakın sınıflar bir araya gelmelidir. Inheritance (Kalıtım) veya Aggeration (Biraraya Getirme) gibi ilişki biçimleriyle sınıfları da organize etmeliyiz.

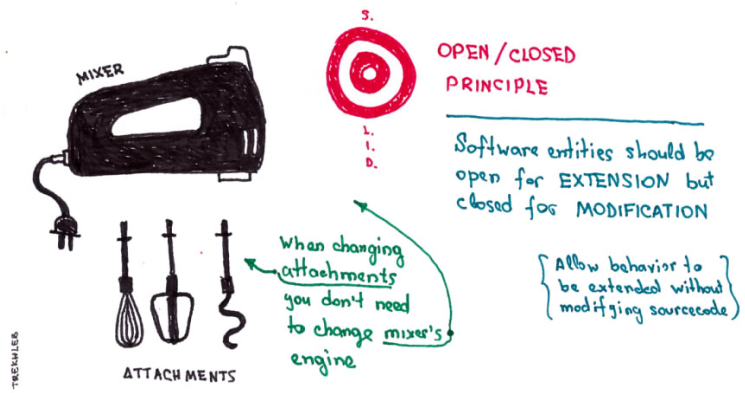
Bu prensip bizlere bir sınıfın ve bir fonksiyonun sadece bir görevi olduğunu söylemektedir.



Open/Closed Prensibi

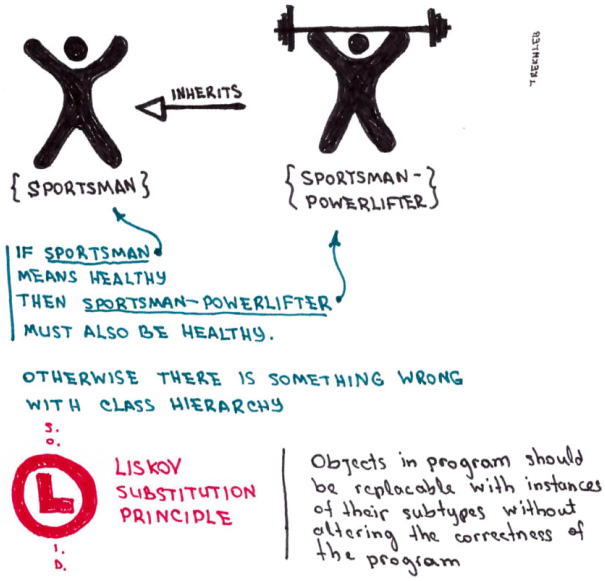
Yazdığımız kodların birbirinden iyi izole olmuş olması gerekmektedir. Böylece, bakımı kolay programlar yazılabilir. Tasarladığımız sınıflarda kod değişikliği yapabilmek için gerçekten tek

bir nedenimizin olması gerekmektedir. Bir üstte bahsettiğimiz Single Responsibility prensibi gereği her sınıf tek bir işi yapmakla yükümlü olmalıdır. Diyelim ki yazılım üzerine yeni bir istek geldi. Eğer bu durumda hemen sınıf içinde birçok kod değişikliği yapmaya başlıyorsak, tasarladığımız sınıf Open/Closed prensibe uygun değildir. Bunun yerine ek bir özellik için sınıflarımız genişleyebilir olmalıdır. Sınıf içinde yapılan değişiklik bir başka sorunun (bug) ortaya çıkmasına sebep olabilir. Örümcek ağı gibi birbirine bağlı karmaşık bir sınıf yapısı var ise bu değişiklik pahalıya patlayabilir. Bu nedenle sınıflarımız gelen değişim isteklerine kalıtım yoluyla başka sınıflar eklenebilir olma özelliğiyle cevap vermelidir. Yani, bir değişiklik olduğunda ilgili sınıfın kodlarını değiştirmek yerine, yeni özelliği ayrı bir sınıf şeklinde tasarlayıp kullanabilmek gerekmektedir. Bu birbirinden iyi izole olmuş sınıflar olmasını sağlayacaktır. Buna değişime kapalı, genişlemeye açık sınıflar diyoruz.



Liskov Substitution Prensibi

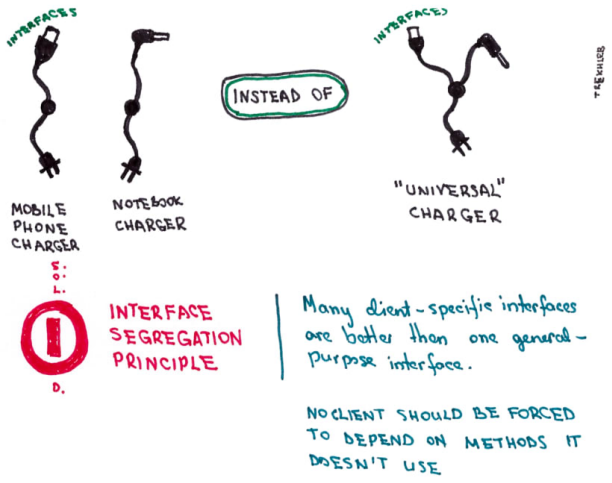
SOLID prensipler içinde en muğlak kalan başlıklardan biridir. Özetle, kalıtım yoluyla birbiriyle ilişkisi olan A ve B sınıfımız olsun. A sınıfı ATA (üst) sınıf olsun. B ise A sınıfından türemiş alt sınıfımız olsun. B herhangi bir özellik kaybına veya değişime uğramadan A referansında kullanılabilmelidir. Yani, A b = new B(); dediğimizde B'de fonksiyon kaybına veya bir bozulmaya sebep vermemeliyiz.



Liskov Prensibinin tanımı: “Türeyen sınıf yani alt sınıflar ana(üst) sınıfın tüm özelliklerini ve metotlarını aynı işlevi gösterecek şekilde kullanabilme ve kendine ait yeni özellikler barındırabilmelidir.”

Interface Segregation Prensibi

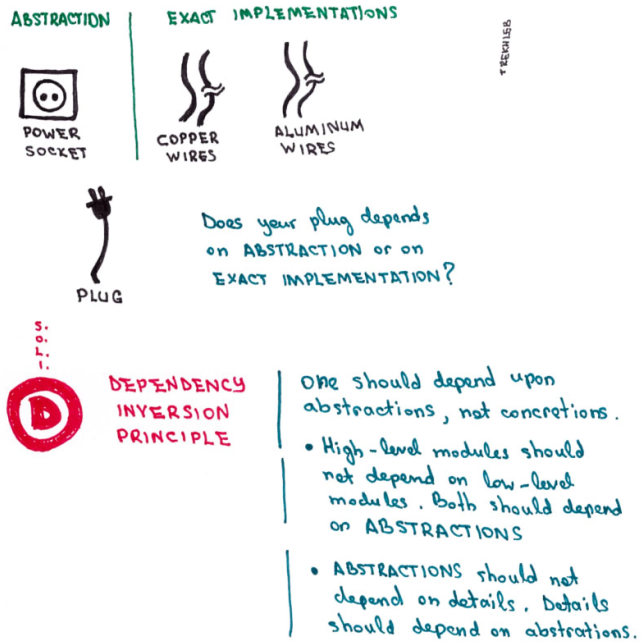
Interface tanımlamak geliştirdiğimiz uygulamadaki soyutlama gücümüzü arttırmaktadır. Interface’ler ile bir nesne bulunması gereken fonksiyonları belirtiriz. Fakat, bu fonksiyonların nasıl işleyeceği alt sınıflarda sınıfın ihtiyacına göre doldurulur. Fakat, Interface’ler de belli bir boyutta olmalıdır. Yani gereksiz fonksiyon tanımlarını bünyelerinde bulundurmamaları gerekmektedir. O yüzden interface içinde tanımlı olması gereken fonksiyonları mantıksal olarak nesne ile ilişkisi varsa dahil etmeliyiz. Genel amaçlı birçok özelliği içinde barındıran interface’lerden kaçınmalıyız.



Dependency Inversion Prensibi (Bağımlılığın tersine çevrilmesi)

Yazılımları gerçekleştirirken onlarca sınıf yazarız. Bu sınıflar bir diğerine mutlaka ihtiyaç duyar. Bu ihtiyaç sonucu bazı sınıfları diğer sınıfların özelliklerini “Aggregation” yoluyla veya “Inheritance” yoluyla bünyesine alır. İşte bu durumda diğer sınıfın özelliklerini kullanan sınıf ona karşı bir bağımlılığı oluşur. Sınıflar arası bağımlılığın olması çok normaldir. Fakat, bu sınıflar arası bağımlılığın nasıl düzgün yönetileceği önemli bir konudur. İşte bağımlılığı tersine çevirme de bu kolaylığı sağlayan prensiptir. Bir sınıf başka bir sınıfa ihtiyaç duyduğunda onun alt sınıflarından birine referans alıp sınıf içinde oluşturmak yerine, kalıtım aldığı ATA sınıf tipinde bir referans tutarak bu nesnenin dışarıdan gönderilmesini bekler. Bu bağımlılığı tersine çevirmek için önemli bir adımdır.

Burada amaç oluşan bağımlılığın referansının interface veya abstract class tipinde olmasıdır. Ayrıca diğer önemli nokta sınıf içinde “new” anahtar kelimesi ile bu interface veya abstract class’dan türeyen bir alt sınıfın yaratılmaması gerekmektedir.



Veri Tabanı Yönetim Sistemleri

Kurumsal uygulamaların hemen hemen hepsi veri tabanları ile çalışırlar. Dolayısıyla Veri tabanı Yönetim Sistemleriyle de çalışmış olurlar. Kurumsal projelerde üretilen veya işlenen veriler kalıcı olarak veri tabanlarına kaydedilirler. Örneğin, alışveriş yaptığınız bir e-ticaret sitesinde geçmiş siparişleriniz, kişisel bilgileriniz, adres bilgileriniz birçok veri veri tabanlarında depolanırlar. Böylece, kalıcı verileri düzgün ve tutarlı bir şekilde tek bir noktada yönetebilir şekilde depolayabiliriz.

Veri her türden sayısal veya sözel birimlerden oluşabilir. Örneğin göz renginiz, boyunuz, kilonuz da veriyi ifade eder. Yahut öğrenci olduğunuzu düşünün okuduğunuz bölümle ilgili alanlar, dersler, notlarınız ve kimliğiniz bir veri parçasıdır.

Veri tabanı, en basit tabirle bilgiyi depolayan yazılım çözümüdür. Aslında, veri tabanları veriyi kalıcı diskte saklasa da bunu özel bir formatta sorgulanabilir ve tutarlı olacak şekilde saklar. Bir veri tabanı oluşturulduktan sonra içinde alakalı tabloları bir arada tutar. Veri tabanı bu tabloların evrensel kümesi gibidir, hepsini kapsar. Aynı zamanda veri üzerinde performanslı, verimli ve esnek işlemler yapabilmeyi sağlayarak yönetilebilir bir yapı sağlar.

Veri tabanı genel özellikleri:

- Birbiriyle ilişkili olan verileri düzenli bir yapı formatında tutar. Bu düzenli yapılar veriler topluluğunu oluşturur.
- Birçok kullanıcı veri tabanına erişip işlem yapabilir.
- Veri bütünlüğü ve tutarlılığı sağlanır.
- Veri tekrarının önüne geçer.
- Verilerin tek bir merkezi noktadan yönetilmesini sağlar.

Veri tabanları üzerinde çalışan ve yönetilebilme olanağı veren yazılımsal çözümlere de Veri tabanı Yönetim Sistemleri denir.

Veri Tabanı Yönetim Sistemleri

Veri tabanları üzerinde yönetim sağlayan sistemlerdir. Veri tabanı Yönetim Sistemi yeni veri tabanı oluşturmak, güncellemek, kullanmak, veri tabanı üzerinde kullanıcılar tanımlamak, kullanıcıların yetkilerini belirlemek gibi yönetsel olanaklar sağlamaktadır. Aşağıdaki imkanları sağlar.

- Veri Tekilliği ve Veri Bütünlüğü: Veriler merkezi bir noktada tutularak, yazılımların sürekli mükerrer ve tutarsız veriler üretmesini önler. Veriler tek noktada toplanır, güncellenir veya tek noktadan sorgulanabilir.
- Veri Güvenliği: Veri tabanları üzerinde kullanıcılar tanımlanabilir. Veritabanlarına erişim için kullanıcı adı ve şifre koruması sağlanabilir. Tanımlanan kullanıcılar belli yetkiler çerçevesinde işlemler yapabilir. Yetkisi yoksa veri tabanını göremez veya üzerinde işlem yapamayabilir.
- Eş Zamanlılık: Veri tabanı Yönetim Sistemleri veriler üzerinde eş zamanlı erişim yapılabilmesini sağlarlar. Böylece, saniyeler içinde binlerce kullanıcı erişim sağlayabilir.

Birden fazla veri tabanı yönetim sistemi çözümü vardır. Ticari lisanslarının yanında, özgür yazılım lisansına sahip Veri tabanı Yönetim Sistemleri vardır. Örneğin, PostgreSQL, MariaDB özgür yazılım lisansına sahip olanlarıdır. MSSQL, Oracle, DB2 gibi çözümler ise ticari lisansa sahiptirler.

Veri Tabanı ile Geliştirilen Uygulamaların Mimarisi

Veri tabanı kullanan yazılımlar genellikle 3 katmanlı mimari şeklinde bir yaklaşımlar veriyle etkileşime geçerler. Katmanlara ayırmamızın sebebi birbiriyle ilişkili olan işlemlerin o katmanda yapılmasıdır. Böylece, sağlam bir izolasyon sağlanmış olacaktır. Bakımı kolay performanslı, az hata çıkaran yazılımlar gerçekleştirmek mümkün olacaktır. Bu üç katman aşağıdaki gibidir.

- Veri Katmanı (Data Layer)
- İş Katmanı (Business Process Layer)
- Sunum Katmanı (Presentation Layer)

Veri Katmanı

Veri tabanına verilerin eklenmesi, güncellenmesi veya sorgulanabilmesi gibi veritabanıyla direkt etkileşim halinde olan katmandır. Veriye erişim katmanı olarak bilinir.

İş Katmanı

Veri tabanından alınan veya değiştirilen verilerle çalışan iş akışlarının algoritmaların olduğu katmandır. Örneğin, bankada bir EFT veya Havale işleminin nasıl gerçekleşeceği bir algoritma çerçevesinde işletilir. İş bu iş akışlarının bütününün yazılımsal olarak gerçekleştirildiği katmandır. İşlenen, üretilen veya kullanılan veri bu katmandan bir alt katman olan “Veri Katmanına” iletilir. Genellikle “Transaction” yönetimi bu katmanda yapılır. Mülakatta gelebilir ☺

Sunum Katmanı

İş katmanından gelen veriler ön yüzde gösterilir veya önyüzden yapılan bir işlem işlenmek amacıyla iş katmanına iletilir. Kullanıcı ile etkileşimin sağlandığı katmandır. Bir mobil ara yüzü, bir web sayfası veya bir Veri tabanı Yönetim Sistem istemci yazılımı olabilir.

İlişkisel Veri tabanı Yönetim Sistemi

İlişkisel veri tabanı yönetim sistemlerinde veriler satır ve sütun şeklinde bir formatta tablolar halinde saklanır. Oluşturulan tablodaki sütunlar bir nesnenin niteliklerini ifade eder. O nesneyle ilişkili özellikler sütunlar halinde bir tabloda bir araya getirilir. İşte bu veri formatı nedeniyle “İlişkisel Veri tabanı Yönetim Sistemleri” denilmektedir. Tablolar arasında ilişki kurulabildiği için bu ismi almamıştır. Aksine birbiriyle ilişkili niteliklerin bir tabloda toplanmasıyla oluşan veri formatından dolayı ismini almıştır. Örneğin: Ders isminde bir varlığımız olduğunu düşünürsek Ders nesnesini tanımlayan alakalı özellikleri bir araya getirmeye çalışırız. Dersin ismi, kaç saatten oluştuğu, hangi dönem olduğu gibi birçok özellik bir araya gelip “Ders” tablosunu oluşturur.

İlişkisel veri tabanlarında tablolar arasında da ilişki kurulabilir. Örneğin: Öğrencinin kimlik bilgileri bir tabloda yer alırken, öğrenciye ait adres bilgileri başka tabloda yer alabilir. Doğru olanı da tasarımsal açıdan böyledir zaten. Öğrenci ve Adres tabloları arasında bir ilişki kurabiliriz.

İlişki tipleri:

- Bire bir (1-1)
- Bire çok (1-N)
- Birçoğa bir (N-1)
- Birçoğa Birçok (N-N)

ACID (Atomicity, Consistency, Isolation, Durability)

İlişkisel veri tabanı yönetim sistemleri mimarisinde iş süreci (transaction) ve veri bütünlüğünü sağlamak için uyulması gereken kurallara ACID denilmektedir. İş süreci (transaction) kavramı bir işleminin ya bütünüyle yapılmasını ya da yapılan işlemlerin bütünüyle geri alınıp veri tutarlılığın sağlanmasını garanti altına demektir. Örneğin, hesabınızda 100 TL'yi arkadaşınızın hesabına havale yoluyla göndermek istiyorsun diyelim. Bu iş süreci (transaction) iki işlem parçasından oluşur. 100 TL senin hesabından azaltılıp sana ait bakiye bilgileri güncellenir, ardından 100 TL arkadaşın hesabına +100 olarak işlenir ve güncellenir. İşte bu işlem bütünlüğü garanti altına alınmalıdır. Alınamaz ise tutarsız verilerle karşılaşılır. Böylece veri bütünlüğü bozulur. Eğer sizden 100 TL çektikten sonra bir hata oluşursa ve arkadaşınızın hesabına bu ücret yatırılamaz ise tutarsız bir durum oluşacağı aşikardır. Yine mülakatlarda size soru olarak gelebilecek bir bilgidir. ACID'i açıklayabilmek ve mantığını anlamak gerekir.

Atomicity (Bölünmezlik): Bir iş sürecinde (transaction) yapılacak işlemler sıralı bir şekilde birden fazla olabilir. Veri tabanında yapılacak olan bu işlemler kümesi bir bütün olarak ele alınır. Yapılan işlemlerinden herhangi bir tanesi tamamlanamaz ise diğer tüm işlemlerde geçersiz sayılmalıdır. Eğer böyle hatalı bir durum varsa o ana kadar yapılan tüm işlemler geri alınır. Fakat, işlemlerin hepsi başarılı ise o iş süreci sonlandırılır.

Consistency (Tutarlılık): Veri tabanında yapılan işlemlerde hep aynı girdiler ile hep aynı çıktıları almalıyız. İşte bu tutarlılığı gösterir.

Isolation (İzolasyon): Bir transaction'ın tüm işlemleri tamamlanana kadar diğer transactionlar tarafından yapılan değişiklikler ilgili transaction tarafından görülmez. Her transaction birbirinden bağımsız çalışır. İşlem sırasında birbirlerine müdahale etmezler. Veya dışarıdan bir müdahaleyi kabul etmezler. Bu izolasyonu ifade eder.

Durability (Sağlamlık): Bir transaction içinde hata oluşursa geri dönme yeteneğine sahip olmalıdır. Hata oluşursa bir önceki ilk noktaya dönülebilmelidir. Eğer transaction başarılı bir şekilde biterse bu durum loglanmalıdır ve başarılı olduğuna dair mesaj verilmelidir.

İlişkisel veri tabanı yönetim sistemlerinde veriyi sorgulamak için özel bir programlama dili kullanılır. Dünyada neredeyse standart halini almış bu dil SQL'dir. (Structural Query Language) (Yapısal Sorgulama Dili)

SQL içinde işlevlerine göre farklı komutlar yer almaktadır. Bu komutlar mantıksal bir şekilde aşağıdaki gibi sınıflandırılmıştır.

1- DDL (Data Definition Language – Veri Tanımlama Dili)

Veri tabanı veya tablolarda yapısal değişiklikler yapabilmek için var olan SQL komutlarıdır. Bu komutlar ile yeni bir veri tabanı oluşturmak, yeni bir tablo oluşturmak veya var olan tablodan sütun eklemek-silmek veya tüm bunları silebilmek gibi işlemler yapılabilir.

CREATE, ALTER, DROP, TRUNCATE, COMMENT, RENAME gibi komutlar bu işlemleri yapabilir.

2- DML (Data Manipulation Language – Veri İşleme Dili)

Tablolardaki veriler üzerinde kayıt ekleme, silme, değiştirme ve sorgulama yapabilmeyi sağlayan SQL komutlarıdır.

SELECT, INSERT, UPDATE, DELETE

3- DCL (Data Control Language – Veri Kontrol Dili)

Veri tabanı veya tablolar üzerinde yetkilendirmeler yapabileceğimiz SQL komutlarıdır.

GRANT, REVOKE, ALTER DEFAULT PRIVILEGES

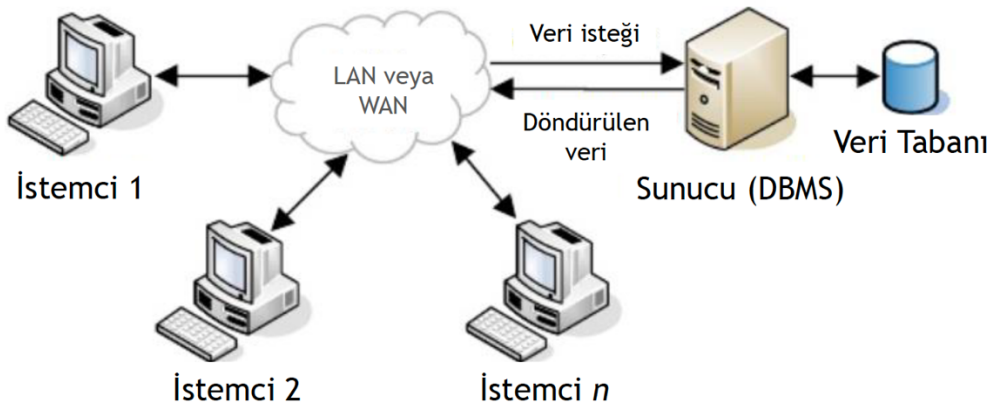
4- TCL (Transaction Control Language – İşlem Kontrol Dili)

Tabloların içindeki verileri değiştirmek için kullandığımız DELETE, UPDATE, INSERT gibi DML komutlarımız vardı. Bu komutlarla bir transaction içinde ardışıl şekilde değişiklikler yapabiliriz. İşte bu değişiklikleri yönetebilmek için kullanılan SQL komutlarıdır. Bir transaction başarılı ise DML komutlarının meydana getirdiği değişiklikler COMMIT komutuyla kalıcı olarak tablolara yansıtılır. Özetle veri kalıcı olarak değişir. Eğer, bir sorun varsa ROLLBACK komutu ile o ana kadar oluşan tüm değişimler geri alınır.

COMMIT, ROLLBACK, SAVEPOINT

Veri Tabanı Yönetim Sistemi Mimarisi

Modern veri tabanı yönetim sistemleri istemci-sunucu (Client-Server) mimarisine sahiptir. Bu nedenle MySQL kurulumu yaptığımızda aslında sunucu şeklinde isteklere cevap verebilen bir yazılım çözümünü ayağa kaldırmış oluruz. Kullanıcılar veya yazdığımız programlar veri tabanlarıyla ve onların içindeki tablolarla etkileşim kurmak istediğinde veri tabanı sunucu yazılımına istekler ulaşır. Sunucu yazılımı bu istekleri ilgili veri tabanında işletir ve sonuçları istemciye geri döner.



Veri Modeli

Veri tabanları verilerin tablolar halinde saklandığı alanlardı. Bu veriler kalıcı diskte (Hard-Disk) belli bir format biçiminde saklanır. Veri tabanı aşağıdaki 3 yapıdan oluşur.

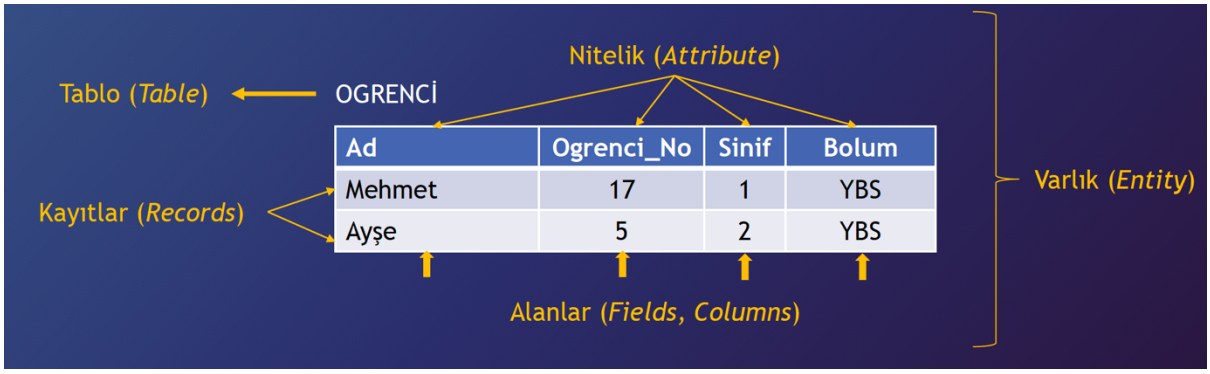
Entity (Varlık): Gerçek hayat nesnesini veya kavramını ifade eder. Örneğin, öğrenci, çalışan personel, adres, maaş gibi kavramlar veya nesneler varlıkları ifade eder. Varlıklar genelde veritabanı dünyasında tablolar şeklinde ifade edilir.

Nitelik (Attribute): Yukarıda bahsettiğimiz gerçek hayat varlığı veya kavramının niteliklerini ifade eder. Örneğin, öğrenciyi tanımlayan nitelikler numarası, bölümü, ismi, soy ismi gibi nitelikleridir. İşte bu nitelikler tablodaki sütunları ifade ederler. Her nitelik bir sütunu ifade edebilir.

İlişki (Relationship): İki varlık arasındaki mantıksal bağlantıyı ifade eder. Örneğin Öğrenci ile Ders varlıkları arasında doğası gereği bir ilişki söz konusudur. Yahut, Personel ile Maaş arasında da benzer bir ilişki vardır. Varlıklar arasındaki ilişki veri tabanı dünyasında tablolar arasındaki ilişkiyi ifade eder.

İlişkisel Veri Modeli

Varlıkların veri tabanı tarafında tabloları ifade ettiğinden bahsetmiştik. Ayrıca, bu varlıkların birçok niteliği bulunmaktaydı. Bu nitelikler tablolardaki sütunları ifade etmektedir. Bu birbiriyle ilişki niteliklerin bir araya gelmesiyle ilişki veri modeli ortaya çıkmaktadır.



Yukarıdaki örnekte “OGRENCI” isminde bir tablo oluşturulmuştur. “Ogrenci_No”, “Sinif” ve “Bolum” isminde nitelikler bir araya gelerek bu tablo yapısını meydana getirmiştir. Tablodaki her bir satır ise bir öğrenci kaydını ifade etmektedir. Yani bir gerçek hayat varlığının verileriyle birlikte tabloda yer almasını ifade eder.

Birincil Anahtar (Primary Key) ve Yabancı Anahtar (Foreign Key)

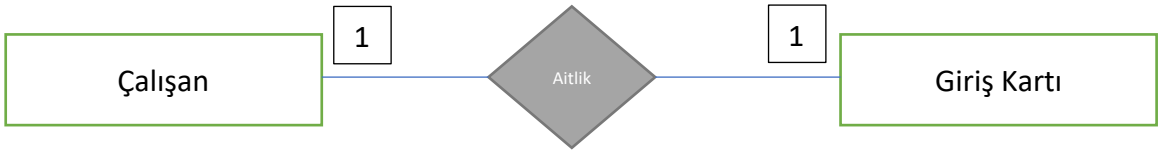
Birincil Anahtar: Primary Key alanı tablodaki her bir satırın tekil olmasını sağlar. Her bir satıra eşsiz bir değer verir. Böylece, kayıtlara ait tekil bir sütun oluşur. Primary Key tablo üzerinde bir sütuna uygulanır. Bu özel sütundaki bir değer bir daha kesinlikle tekrar edemez. Tekrar aynı değerden üretilmeye çalışılırsa hata verir. Örneğin kişinin TC numarası tekil bir alandır ve Primary Key olabilir. Eğer varlığın böyle karakteristik bir tekil niteliği yoksa, Primary Key sütununun otomatik artan olması sağlanarak tekil yapılabilir. Bir tablonun birden fazla Primary Key sütunu olabilir. Bu birden fazla Primary Key sütunu bir araya gelip tekil bir değer oluşturabilir.

Yabancı Anahtar: Foreign Key olarak isimlendirilirler. Bu anahtar sütunlar ile başka tablolarla ilişki kurulabilir. İlişki kuracağı tablodaki Primary Key sütunundaki değeri referans olarak kendi tablosunda bir sütunda tutar ve bu ikisi arasında bir ilişki kurar. Böylece tablolar birbiriyle ilişkilendirilmiş olur.

İlişki Türleri

Bire Bir İlişki

1-1’e ilişki biçimi bir tablodaki kayıt ile ilişki kurulan diğer tabloda sadece bir kayıt ile eşleşir. Örnek olarak Çalışan tablosunda bir personelin sadece bir tane giriş kartı olabilir. Aynı şekilde Giriş Kartı tablosunda bir giriş kartına ait kayıt sadece bir kişiye ait olabilir. Çalışan ile Giriş Kartı arasında bire bir ilişki vardır.



Bire Çok İlişki

Bir tablodaki kayıt diğer tablodaki bir veya birden fazla kayıt ile eşleşebilir. Örneğin, çalışan personel ile maaş varlıkları arasında bire çok ilişki biçimi vardır. Çünkü, bir çalışanın birden fazla maaş kaydı olabilir. Fakat, bir maaş kaydı sadece bir çalışana aittir.

Çoktan Çoğa İlişki

Bir tablodaki bir kayıt diğer tablodaki bir veya birden çok kayıt ile eşleşebilir. Aynı şekilde diğer tablodaki tek bir kayıt ilişkili olduğu tabloda bir veya birden çok kayıtla eşleşebilir. Örneğin Çalışan tablosundaki bir personelin birden çok unvanı olabilir. Aynı şekilde Unvan tablosundaki bir görev tanımı birden fazla çalışanda olabilir. Yani bir çalışan hem Yazılım Mühendisi unvanına hem de Takım Lideri unvanına sahip olabilir. Aynı şekilde Yazılım Mühendisi unvanı birden fazla çalışana ait olabilir. Bu ilişki biçiminde mutlaka ara bir tablo oluşur. Bu ara tabloda her iki tablodan ilişkiyi tutan sütunlar yer alır.

Normalizasyon

Veritabanı dünyasında bir varlığı temsil bir tablo oluşturduğumuzda onunla ilişkili nitelikleri de aynı tabloda sütun olarak açıyorduk. Fakat, o varlıkla ilgili tüm nitelikleri veya alanları aynı tabloya doldurup tek bir tabloda yüksek sayıda sütunlardan oluşan geniş bir tablo yapısı kurmak çok sağlıklı değildir. Bunun büyük zararı her kayıt eklendikçe gereksiz veri tekrarları yaşanmasına sebep olacaktır. Örneğin Çalışan verilerini bir tabloda sütunlarla temsil ettik diyelim. İsim, soy isim, doğum tarihi vb sütunlar açtık. Ardından, çalışana ait adres verilerini tutmak gerektiğinde gidip bu nitelikleri Çalışan tablosunda birer sütun olarak açarsak veri tekrarına sebep oluruz. Çünkü, çalışanın birden fazla adresi olabilir. Her yeni adres kaydı eklendiğinde bütünüyle yeni bir satır eklenir. Bu durumda çalışanın isim, soy isim ve doğum tarihi gibi verileri sürekli tekrar eder. Böylece, veri tabanının veri tutma kapasitesini doğru kullanmamış oluruz.

İşte bu sebeple bir tablodaki çok fazla sütun ve satırdan oluşan tabloyu tekrarlardan arındırmak için alt kümelerden oluşan yeni tablolar oluşturup bu varlıklar arası ilişki kurabiliriz. Bu alt kümelere, yani alt tablolara ayırma işlemine normalizasyon denir.

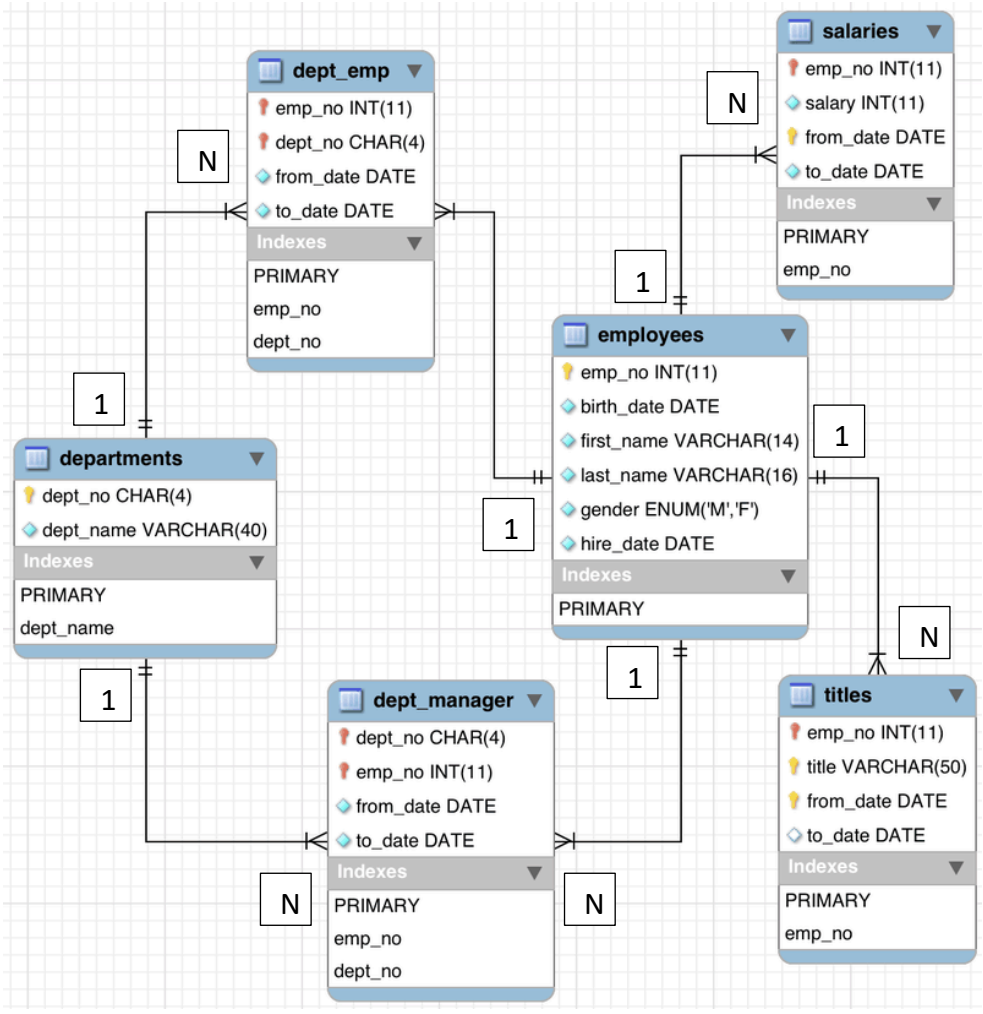
Normalizasyon amaçları:

- Veri tekrarını önlemek
- Performans ve veri tutma kapasitesini verimli kullanmak
- Veri tutarlılığını arttırmak

Çalışan Veri Tabanı Örneği

Örneklerimizi gerçekleştirmek için MySQL'in sağladığı örnek bir veri tabanı şeması üzerinden gideceğiz. Bu veri tabanında çalışan verileri, çalışan maaşları, çalışanların unvanları, departman verileri gibi tablolar yer alacaktır. Bu tablolar arasında Bire-Çok (1-N), Çoğa-Çok (N-N) gibi ilişki biçimleri vardır. N-N ilişkilerde mutlaka ara bir tablo oluşur.

Ayrıca, Primary Key ve Foreign Key ile ilişkiler kurulmuştur. Örnek şemayı aşağıda veriyoruz.



“employees” tablosu çalışan verilerini depolar. “titles” tablosu çalışana ait unvanları depolar. Bir çalışanın birden fazla unvanı olabilir. “salaries” tablosu çalışana ait maaşları depolar. Yine bir çalışanın her yıl için bir maaş tutarı vardır. Bu durumda çalışan ile maaş kayıtları arasında bire çok ilişki vardır. “departments” tablosu şirkette bulunan departmanları depolar. Çalışan geçmişte birden fazla departmanda çalışmış olabilir veya halen aynı departmanda çalışmaya devam ediyor olabilir. Bu durumda “departments” tablosu ile “employees” arasında çoğa-çok bir ilişki vardır. O nedenle “dept_emp” tablosu bu çoğa çok ilişkideki kayıtları tutar. Aynı şekilde bir çalışan departman yönetici konumunda olabilir. Hatta, geçmişte birden çok departmanda yöneticilik de yapmış olabilir. Bu nedenle yine “departments” tablosu ile “employees” tablosu arasında çoğa-çok bir ilişki vardır. Bu çoğa çok ilişki sonucunda bir ara tablo ihtiyacı doğar. “dept_manager” tablosunda çalışanların hangi departmanlarda veya departmanda görev aldıklarının kayıtları vardır.

Temel SQL

SQL programlama dilinin ilişkisel veri tabanı yönetim sistemlerinde veya ilişkisel formattaki veri tabanlarında veri sorgulama dili olduğundan bahsetmiştik. Şimdi bu sorgulama diliyle yapılabilecekleri yukarıdaki veri tabanını örnek alarak inceleyelim.

DDL SQL Komutları (Data Definition Language)

DDL komutları veri tabanı ve tablolar üzerinde yapısal değişiklikler yapmayı sağlar.

Veri tabanı oluşturmak

```
CREATE DATABASE mycompany_db;
```

CREATE DATABASE SQL komutu ile yeni bir veri tabanı yaratabilirsiniz.

Veri tabanını seçmek

```
USE mycompany_db;
```

USE komutu ile çalışmak istediğiniz veri tabanını seçebiliyorsunuz. Biliyorsunuz ki veirtabanı yönetim sistemleri birden fazla veri tabanı içerebilirler. Hangi SQL komutlarının hangi veri tabanında çalıştırılacağı USE komutu ile belirtilir.

Tablo oluşturmak

Veri tabanını oluşturduktan sonra veri tabanı içinde birden fazla tablo oluşturabiliriz.

```
CREATE TABLE employees (
    emp_no      INT             NOT NULL,
    birth_date  DATE            NOT NULL,
```

```
    first_name  VARCHAR(14)      NOT NULL,
    last_name   VARCHAR(16)      NOT NULL,
    gender      ENUM ('M','F')   NOT NULL,
    hire_date   DATE             NOT NULL,
    PRIMARY KEY (emp_no) );
```

Yukarıda CREATE TABLE komutu ile yeni bir tablo oluşturuyoruz. Bu komuttan hemen sonra tablo isminin ne olacağını yazıyoruz. Biz “employees” isminde bir tablo oluşturduk. Tabloyu oluştururken bu ilişkisel veri formatında hangi sütunlar yer alacak onları belirtiyoruz. Bu sütunlar hatırlayacaksanız ki varlığın nitelikleriydi. Bu niteliklere sütun isimleri ve veri tipinin ne olacağını veriyoruz. Bu konuda veri tipleri detaylı işlenmeyecektir. İlgili veri tipleri veri tabanı yönetim sistemlerinin resmi dokümanlarında bulabilirsiniz. Sütunların isimlerini ve veri tiplerini girdikten sonra artık tabloda hangi sütunun Primary Key olacağını belirtmek gerekiyor. Bunun için “PRIMARY KEY” komutu kullanılır. Bu komuttan hemen sonra () parantezler içinde tekil anahtar olarak kullanmak istediğiniz sütunun ismini yazarsınız. Bu örnekte “emp_no” sütunu “employees” tablosunun. PRIMARY KEY’i olacaktır.

Foreign Key ile İlişkili tablolar yaratmak

Foreign Key (Yabancı Anahtar) hatırlayacağınız üzere bir tablo ile diğeri arasında ilişki kurmayı sağlıyordu.

```
CREATE TABLE salaries (
    emp_no      INT             NOT NULL,
    salary      INT             NOT NULL,
    from_date   DATE            NOT NULL,
    to_date     DATE            NOT NULL,
    FOREIGN KEY (emp_no) REFERENCES employees (emp_no) ON DELETE CASCADE,
    PRIMARY KEY (emp_no, from_date)
);
```

Yukarıda “salaries” isminde bir tablo oluşturuyoruz. Yukarıda tablo oluşturmak için bahsettiğimiz her şey burada da geçerlidir. Fakat, bu SQL komutunda bir fark bulunmaktadır. “salaries” tablosu ile “employees” tablosu arasında bire-çok ilişki bulunmaktadır. FOREIGN KEY komutu ile iki tablo arasında bir ilişki kuruyoruz. Bu komuttan hemen sonra () parantezler arasında “salaries” tablosunda hangi sütunun FOREIGN KEY verisini tutacağını belirtiyoruz. Böylece, “employees” tablosunda hangi çalışan ise onun tekil verisi bu sütunda tutulacaktır. Ardından REFERENCES komutu ile hangi tablo ile ilişki kurmak istiyorsak onun ismini yazıyoruz. Bu örnekte “employees” tablosu ile ilişki kuracağımız için onun ismini yazdık. İlişki kurulacak tabloyu belirledikten sonra () içine bu tablonun hangi sütununu kullanacağımızı belirtiyoruz. Genellikle ilişki kurulan tablonun PRIMARY KEY sütunu burada belirtilir. Bizim örneğimizde “employees” tablosu olduğu için onun “emp_no” isimli PRIMARY KEY sütunuyla ilişki kuruyoruz.

emp_no	salary	from_date	to_date
10001	60117	1986-06-26	1987-06-26
10001	62102	1987-06-26	1988-06-25
10001	66074	1988-06-25	1989-06-25
10001	66596	1989-06-25	1990-06-25
10001	66961	1990-06-25	1991-06-25
10001	71046	1991-06-25	1992-06-24
10001	74333	1992-06-24	1993-06-24
10001	75286	1993-06-24	1994-06-24
10001	75994	1994-06-24	1995-06-24
10001	76884	1995-06-24	1996-06-23
10001	80013	1996-06-23	1997-06-23
10001	81025	1997-06-23	1998-06-23
10001	81097	1998-06-23	1999-06-23
10001	84917	1999-06-23	2000-06-22
10001	85112	2000-06-22	2001-06-22
10001	85097	2001-06-22	2002-06-22
10001	88958	2002-06-22	9999-01-01
10002	65828	1996-08-03	1997-08-03
10002	65909	1997-08-03	1998-08-03
10002	67534	1998-08-03	1999-08-03
10002	69366	1999-08-03	2000-08-02
10002	71963	2000-08-02	2001-08-02

Yukarıda “salaries” tablosuna ait kayıtlar görülmektedir. Tablodaki “emp_no” sütunundaki veriler ilişki kurduğumuz “employees” tablosunun PRIMARY KEY alanından gelmektedir. Böylece, her çalışana ait ne kadar maaş kaydı var bulmak çok kolay oluyor.

Veri tabanı silmek

```
DROP DATABASE mycompany_db;
```

DROP DATABASE komutundan sonra silinmesini istediğimiz veri tabanı ismini veririz. Böylece veri tabanı silinmiş olur.

Tablo silmek

```
DROP TABLE salaries;
```

DROP TABLE komutundan sonra tümüyle silinmesini istediğimiz tablonun ismini yazıyoruz. Tablo yapısal olarak içindeki verilerle birlikte silinir.

Tablo verilerini tümüyle silmek

```
TRUNCATE TABLE salaries;
```

TRUNCATE TABLE komutu ile tabloyu yapısal olarak koruyarak sadece içindeki verilerin silinmesini sağlayabiliriz.

Tabloyu yapısal olarak değiştirmek

Tablolar üzerinde bazen yeni sütun eklemek çıkarmak veya değiştirmek gerekir. Bu yapısal anlamda tabloyu değiştirmek demektir.

Tabloya yeni bir sütun eklemek

```
ALTER TABLE salaries
ADD COLUMN yeni_sutun VARCHAR(10);
```

“salaries” tablosuna “yeni_sutun” isminde bir alan eklemek istediğimizi düşünelim. Bu durumda ALTER TABLE komutundan sonra değişiklik yapılacak tablonun ismi verilir. ADD COLUMN komutundan sonra yeni sütunun ismi verilir. Yeni sütunu tanımlarken veri tipinin ne olacağı da mutlaka belirtilmelidir. Bu örnekte maksimum 10 karakter uzunluğunda yazı tabanlı bir veri tutacağımızı söylüyoruz.

Tablodan bir sütunu silmek

```
ALTER TABLE salaries
DROP COLUMN yeni_sutun;
```

DROP COLUMN komutu ile silmek istediğimiz sütunu tablodan çıkarabiliriz.

Tablonun bir sütununu değiştirmek

```
ALTER TABLE salaries
MODIFY yeni_sutun INT(10);
```

Tabloda bir sütunun veri tipini veya ismini değiştirmek isteyebiliriz. Sütun üzerinde değişiklik yapabilmek için MODIFY komutu kullanılabilir.

DML SQL Komutları (Data Manipulation Language)

DML komutları ile tablolardaki veriler üzerinde sorgulama, değiştirme ve silme işlemleri uygulayabiliriz.

Tablodan veri çekmek

```
select * from employees;
```

SELECT komutu ile tablodan veri çekeceğimizi belirtiyoruz. FROM kelimesi ile hangi tablodan bu verinin alınacağını belirtiyoruz. * işareti tabloda ne kadar sütun varsa hepsini sorgu ve getir demektir.

```
select * from employees where emp_no = 10004;
```

yukarıdaki örnekte ise koşullu bir sorgulama işlemi yapıyoruz. WHERE komutundan sonra sorgunun koşulunu oluşturuyoruz. WHERE'den sonra birden fazla sorgulama koşulu verebilirsiniz. Bu örnekte "emp_no" sütununda verisi "10004" olan kayıtları getirmişiz.

```
select * from salaries where emp_no = 10001 and salary >= 80000;
```

Yukarıdaki örneğimizde WHERE ifadesinden sonra birden fazla koşulu bir arada kullanmışız. İki koşulu birbirine AND ifadesi ile bağlamışız. AND ifadesi ve bağlacı anlamındadır. Java'daki && operandı gibi işlevi vardır. Örneğin burada 10001 nolu kaydın 80000'den büyük olan maaş bilgilerini "salaries" tablosundan getiriyoruz.

Çoklu tabloları bir araya getirerek sorgulamak

Veri tabanı tablolarını bir araya getirip sorgulama yapabiliriz.

```
select emp.*, s.salary from employees emp
inner join salaries s on s.emp_no = emp.emp_no
where emp.emp_no = 10004;
```

Bu örneğimizde INNER JOIN birleştirme işlemini göreceğiz. INNER JOIN bir araya getirilen tablolardaki kayıtları bire bir eşleştirerek getirir. Eğer bire bir eşleşmiyorsa o kayıt sorgu sonucunda gelmez. INNER JOIN ifadesinden sonra bir arada kullanmak istediğimiz tablo ismini vermektir. Ardından bu tabloların hangi sütunları üzerinden bağlantı kurulması gerektiği belirtilmelidir. Bunun için ON ifadesinden sonra "salaries" tablosundaki "emp_no" sütunu ile "employees" tablosundaki "emp_no" sütununu kıyaslıyoruz. Her iki sütunda birbirine eşit olan kayıtları JOIN'lenmiş biçimde adeta tek bir tabloymuş gibi sorgu sonucunda bir araya getiriyoruz.

emp_no	birth_date	first_name	last_name	gender	hire_date	salary
10001	1953-09-02	Georgi	Facello	M	1986-06-26	60117
10001	1953-09-02	Georgi	Facello	M	1986-06-26	62102
10001	1953-09-02	Georgi	Facello	M	1986-06-26	66074
10001	1953-09-02	Georgi	Facello	M	1986-06-26	66596
10001	1953-09-02	Georgi	Facello	M	1986-06-26	66961
10001	1953-09-02	Georgi	Facello	M	1986-06-26	71046
10001	1953-09-02	Georgi	Facello	M	1986-06-26	74333
10001	1953-09-02	Georgi	Facello	M	1986-06-26	75286
10001	1953-09-02	Georgi	Facello	M	1986-06-26	75994
10001	1953-09-02	Georgi	Facello	M	1986-06-26	76884
10001	1953-09-02	Georgi	Facello	M	1986-06-26	80013
10001	1953-09-02	Georgi	Facello	M	1986-06-26	81025
10001	1953-09-02	Georgi	Facello	M	1986-06-26	81097
10001	1953-09-02	Georgi	Facello	M	1986-06-26	84917
10001	1953-09-02	Georgi	Facello	M	1986-06-26	85112
10001	1953-09-02	Georgi	Facello	M	1986-06-26	85097
10001	1953-09-02	Georgi	Facello	M	1986-06-26	88958
10002	1964-06-02	Bezalel	Simmel	F	1985-11-21	65828
10002	1964-06-02	Bezalel	Simmel	F	1985-11-21	65909
10002	1964-06-02	Bezalel	Simmel	F	1985-11-21	67534
10002	1964-06-02	Bezalel	Simmel	F	1985-11-21	69366
10002	1964-06-02	Bezalel	Simmel	F	1985-11-21	71963
10002	1964-06-02	Bezalel	Simmel	F	1985-11-21	72527
10003	1959-12-03	Parto	Bamford	M	1986-08-28	40006
10003	1959-12-03	Parto	Bamford	M	1986-08-28	43616

```sql
select emp.*, dm.* from employees emp
left join dept_manager dm on dm.emp_no = emp.emp_no;
```

Yukarıdaki komutta ise yine bir araya getirme örneği inceliyoruz. Fakat, LEFT JOIN ile tabloları birleştirmek INNER JOIN'e göre farklıdır. INNER JOIN'de bire bir eşleşen kayıtları bir araya getirip tek bir tablo gibi sunmaktadır. LEFT JOIN'de ise sol tarafta kalan "employees" tablosunda kayıt olmasına rağmen eğer ki sağdaki "dept_manager" tablosunda eşleşen bir kayıt yoksa da sorgu sonucunda "employees"'e ait kayıt getirilir. "dept_manager" tablosunda eşleşmeyen kısımlar ise NULL olarak getirilir.

emp_no	birth_date	first_name	last_name	gender	hire_date	emp_no	dept_no	from_date	to_date
10001	1953-09-02	Georgi	Facello	M	1986-06-26	10001	d001	1985-01-01	1991-10-01
10002	1964-06-02	Bezalel	Simmel	F	1985-11-21	10002	d001	1991-10-01	9999-01-01
10003	1959-12-03	Parto	Bamford	M	1986-08-28	10003	d002	1985-01-01	1989-12-17
10005	1955-01-21	Kyoichi	Maliniak	M	1989-09-12	10005	d003	1985-01-01	1992-03-21
10004	1954-05-01	Chirstian	Koblick	M	1986-12-01	NULL	NULL	NULL	NULL
10006	2020-04-05	Batuhan	Duzgun	M	2020-04-05	NULL	NULL	NULL	NULL
10007	2020-04-05	Batuhan	Düzgün	M	2020-04-05	NULL	NULL	NULL	NULL

```sql
select emp.*, dm.* from dept_manager dm
right join employees emp on dm.emp_no = emp.emp_no;
```

RIGHT JOIN ise LEFT JOIN ile aynı sadece yönü tersidir. Yukarıdaki ile aynı sonucu verir. RIGHT ve LEFT JOIN sorgular mülakatlarda gelebilir. İyi kavramakta fayda vardır.

Mesela LEFT JOIN'in anlamlı bir şekilde kullanıldığı bir örnek verelim. Diyelim ki, şu anda aktif olarak yönetici olan çalışan kişileri bulmak istediğimizde LEFT JOIN işimize yarayacaktır. "employees" tablosunu, "dept_manager" tablosunu ve "departments" tablosunu bir araya getirip 3 tabloyu tek bir tablo olarak LEFT JOIN ile bir araya getirdik. Daha sonra "dept_manager" tablosundaki "to_date" sütunu bugünden büyük olan çalışanlar halen aktif olarak yöneticilik yapıyor demektir.

```sql
-- şuan aktif olarak yönetici olan personel
select emp.*, dm.*, dp.* from employees emp
left join dept_manager dm on dm.emp_no = emp.emp_no
left join departments dp on dp.dept_no = dm.dept_no
where dm.to_date >= NOW();
```