

DATA MANAGEMENT

AND

FILE STRUCTURES

Furkan EKİCİ - 2017555017

Storing Data : Disk and Files	1
RAID	9
Disk Alanı Yönetimi	13
Dosya Organizasyonu	26
İndeksleme	43
Ağacı Yapılı indexler	49
ISAM	51
B+ Ağacı	54
Bulk-Loading Algoritması	60
Hash-Based Index	63
Statik Hashing	65
Extendible Hashing	67
Linear Hashing	70
K-d Tree	73
Grid Files	78
Vize için Örnekler	83
External Sorting	85
DBMS	102
The Entity-Relationship Model	107
The Relation Model	117
Schema Refinement and Normal Forms	132
<hr/>	
DATABASE MANAGEMENT SYSTEMS	145
Relational Algebra	147
SQL	153
Query Evaluation	164
Physical Database Design	173
Transaction Management	179
Crash Recovery	186
Security and Authorization	189
SON	194

## CHAPTER 9 - STORING DATA: DISKS AND FILES

### Disk and Files:

→ DBMS (Data Base Management Systems), bilgiyi hard disklerde depolar.

→ DBMS dizaynında iti temel husus vardır:

- Read (Okuma): Diskteki bilgiler hafızaya (RAM) getirilir.

- Write (Yazma): Hafızadaki (RAM) bilgiler diske kaydedilir.

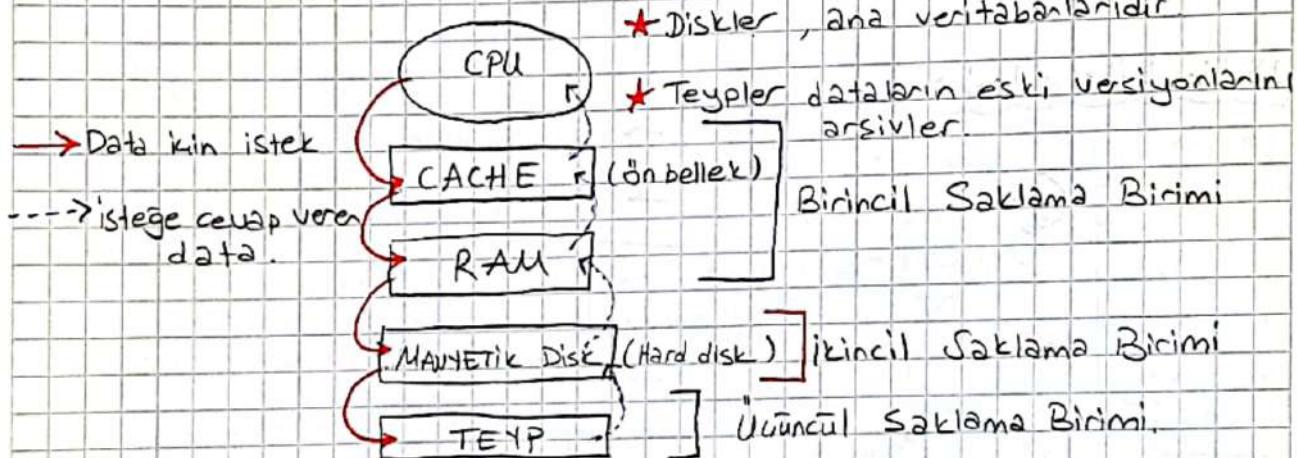
! Bunlar yüksek maliyetli işlemlerdir, dikkatli planlanmalıdır.

### Bellek Hiyerarşisi:

\* RAM bilgisi taşıyan kısımlar.

\* Diskler, ana veritabanlarıdır.

\* Teypler dataların eski versiyonlarını arşivler.



\*? Neden tüm datalar ana bellek (RAM) 'de saklanmıyor?

→ Fiyatı çok pahalı. ( $32\text{ GB RAM} = 1900\text{ TL}$ )

→ Kaydedilmiyor. (RAM'deki bilgiler geçicidir. Herhangi bir elektrik kesintisinde tüm bilgiler gidebilir)

### Saklama Birimlerinin Karşılaştırılması

Tür	Kapasite	Erişim zamanı (Hz)	Fiyat
RAM	32 GB	$5 \times 10^{-7}$ saniye	1900 TL
Hard disk	6 TB	$15 \times 10^{-3}$ saniye	1000 TL
Floppy Disk	1.44 MB	$1 \times 10^{-4}$ saniye	?
CD-Rom	650 MB	75 milisaniye	1.5 TL
DVD	4.7 GB	112 milisaniye	1 TL

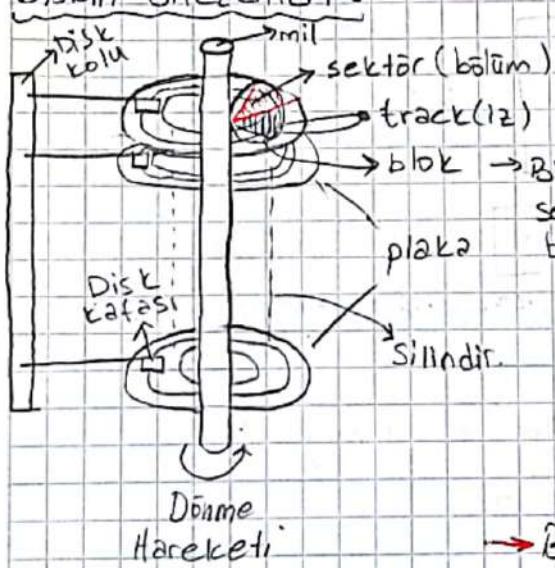
\* Teypleki verilere erişim olukca yavaştır. Genelde data arşivlemek için kullanılır.

→ Artık kullanılmıyor.

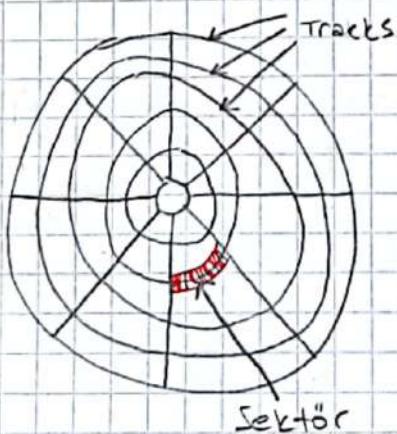
### Diskler:

- İkincil Saklama Birimleridir.
- Datalar, disk bloğu veya sayfa adı verilen birimlerden okunur veya bu birimlere yazılır.
- RAM'den farklı olarak, datanın işlem görme zamanı, datanın diskteki yerine bağlıdır.
- ↳ Bu nedenle, sayfaların diske bağlı bir şekilde yerleştirilmesi, performansı artırır.

### Diskin Bileşenleri:



- Plakalar döner (örn: 90 rps)
- Disk kolu hareket ederek, disk <sup>saniyedeki  
dönmeye sayısı</sup> kafasını istenen track'e getirir.
- Kafanın altındaki trackler hayali bir silindir oluşturur.
- Disk kafası, okuma ve yazma işlemlerini yapabilir.
- Blok boyutluğu sektörün bir tamsayı katıdır.



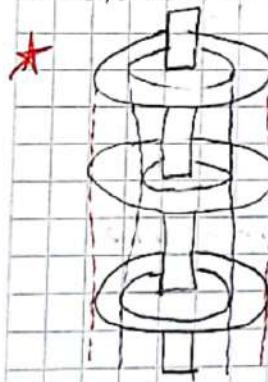
- Disk, merkezleri bir olan tracklerden oluşur.
- Trackler, sektörlerde bölünmüştür.
- Sektörler, diskteki adreslenebilir en küçük birimidir.

### Veriye Erisim:

- Bir program bilgisayardan bir bayt okuduğunda, işletim sistemi bu baytin olduğu tracke ve sektörre disk kafasını getirir ve erişilen bu veri RAM'de tampon (buffer) adı verilen bir kısma aktarılır.
- Okuma/yazma kolunun mekanik hareketlerinden dolayı bir darboğaz oluşturabilir. Bunu önlemek için data tracklerde yan yana yazılmalıdır. Farklı tracklere yazarsak disk kafası o dataya erişmek için mekanik bir hareket yapacak ve zaman kaybı yaşanacaktır.

### Silindirler:

Aynı yarıda sahip tüm trackler bir araya gelerek silindirleri oluşturur.



- Bir silindirdeki tüm bilgiye okuma yazma kafası hareket etmeden ulaşılabilir. (plakalar döndüğü için)
- Böyle ülülü bir plakada okuma yazma kafalarının hepsi aynı pozisyonda bulunur.

### Kapasiteleri Hesaplama:

- Track Kapasitesi = sektör sayısı \* sektör boyutu (byte)
- Silindir Kapasitesi = her bir silindirdeki track \* track kapasitesi.
- Sürücü (drive) Kapasitesi = silindir sayısı \* silindir kapasitesi.
- Silindir sayısı = yüzeydeki track sayısı

Örnek:

Her sektör 512 byte

Her trackte 40 sektör var.

Her silindirde 12 track var

Silindir sayısı = 1331 (record)

Soru 1-) 20.000 kayıt olduğunu ve her bir data kaydı 256 byte yer kaplıyorsa, bu veriyi saklamak için kaç silindir gerektir?

1 sektör 512 byte olduğuna göre, 1 sektör 2 kayıt tutar.

O zaman bir silindir  $12 * 40 * 2 = 960$  tane kayıt tutar.

↕                      ↗  
 bir silindirdeki      bir sektörün taşıdığı  
 track adedi              kayıt sayısı  
 ↘                      ↗  
 bir trackteki      sektör adedi

$20.000 / 960 = 20.83$  yani 21 adet silindir gerektir.

Soru 2-) Diskin kapasitesi nedir?

$$512 * 40 * 12 * 1331 = 327,106,560 \text{ bytes} = 327 \text{ MB}$$

Trackleri Bloğa Göre Organize Etme:

- Trackler sektörler yerine kullanıcı tarafından tanımlanan bloklardan ayrılabılır.
- Bloklar sabit veya değişken uzunluğunda olabilir. (?) noca sabit dedi
- Blocking Factor = Bir blokta tutulan kayıt sayısı
- Blok adresleme zemansında her veri bloğuna, blok hakkında ekstra bilgi içeren bir veya daha fazla alt blok eşlik eder.

## Disk Sayfasına (Blok) Erişim:

Bir bloğa erişim süresi 3 bileşene ayrılmıştır:

1-) Arama Süresi (Seek Time): istenilen bloğun bulunduğu track üzerine disk kafasının gelme süresidir. Disk plaqinin ağı ile doğru orantılıdır.

2-) Dönüşel Gecikme (Rotational Delay): istenilen bloğun disk kafasının konumuna gelme süresidir.

$\text{max dönüşel gecikme} = 1 \text{ tam tur dönmeye zamanı}$

$\text{ortalama " " } = 1/2 \text{ tur dönmeye zamanı}$

3-) Aktarım Süresi (Transfer Time): Disk kafası istenilen bloğun üzerine geldikten sonra o veriyi okuma veya yazma süresidir.

## Sayfaları Diskte Yerlestirme:

Bir dosyadaki bloklar, arama (seek) ve dönmeye (rotation) gecikmesini en az indirmek için disk üzerinde sıralı olarak düzenlenmelidir.

## Disk Erişim Zamanı:

Diske bir veri yazma veya diskten bir veri okuma süresi eylem hesaplanır:

$$\text{Erişim zamanı} = \text{Arama Süresi} + \text{Dönüşel Gecikme} + \text{Aktarım Süresi}$$

→ N Blok veriye erişilmek istenirse;

→ Ardılık yerleşmisse;

$$\text{Erişim süresi} = \text{Arama (Seek)} + \text{Dönüşel (rotational)} + N * \text{Aktarım (transfer)}$$

→ Rastgele yerleşmisse;

$$\text{! Erişim Süresi} = N * (\text{Arama} + \text{Dönüşel} + \text{Aktarım})$$

Örnek:

Bölüm (sector) = 512 byte

Her yüzeyde 2000 track

Her trackte 50 bölüm (sector)

5 adet 2 yüzü plakaya sahip bir hard disk için:

Soru 1-) Bir track kaç byte büyüklüğündedir?

1 track  $\rightarrow$  50 bölüm ve 1 bölüm 512 byte ise

$$\boxed{50 * 512 = 25,600 \text{ byte}}$$

Soru 2-) Bir yüzeyin (surface) kapasitesi nedir?

1 yüzeyde  $\rightarrow$  2000 track var 1 trackte 50 sektör var

$$\boxed{512 * 50 * 2000 = 51,200,000 \text{ bytes} \cong 51 \text{ MB}}$$

↓              ↓              ↓  
 sektör      track içinde      yüzeyinde  
 byte          sektör          track

Soru 3-) Diskin kapasitesi nedir?

5 tane 2 yüzü plaka toplam 10 yüz eder. Yukarıda 1 yüzeyin kapasitesini biliyoruz (soru 2).

$$\boxed{\begin{array}{c} 512 * 50 * 2000 * 10 \cong 512 \text{ MB} \\ \hline \text{Bir yüzeyin boyutu} \\ \text{Diskin boyutu} \end{array}}$$

Soru 4-) Kaç silindire sahiptir?

Diskteki silindir sayısı bir yüzeydeki track sayısına eşittir.

$$\boxed{2000}$$

Soru 5-) Hangisi geneli blok büyüklüğüdür neden?

- 256 byte  $\times$   $256/512 = 0.5$  tam sayı olmaliydi  
↳ sector boyutu
- 2048 byte  $\checkmark$   $2048/512 = 4$
- 51200 byte  $\times$   $51200/512 = 100$ , sonuc 50 veya daha küçük olmalı  
günkü bir trackteki sector sayısı 50'dir.

Soru 6-) Disk plakasının döndürme hızı 5400 rpm ise max

dönmeye gecikmesi (rotational delay) ve ort. dönmeye gecikmesi nedir?

$$\left\{ \begin{array}{l} \frac{1}{5400} \text{ dak} = \frac{1}{5400} * 60 \text{ sn} = 0.011 \text{ sn} = 11.1 \text{ milisaniye} \\ \text{max rot. del} \end{array} \right. \quad \left. \begin{array}{l} \text{Bir track'in} \\ \text{başından sonura} \\ \text{gelme süresi} \end{array} \right\}$$

$$\left\{ \begin{array}{l} \text{ortalama} = \frac{1}{2} * \text{max} = \frac{1}{2} * 11.1 \text{ msn} = 5.55 \text{ msn} \\ \text{60sn'de 5400 dönmeye} \\ \text{x'inde } \frac{1}{60} \text{ kez } \text{Gr.} \end{array} \right. \quad \left. \begin{array}{l} \text{Geleneksel} \\ \text{ve modern} \end{array} \right\}$$

Soru 7-) Aktarım hızı nedir? (Bir trackteki tüm veride erişim süresi)

$$\text{bit Track boyutu} = \left\{ \frac{512 * 50}{11.1} = 2306.3 \text{ byte/msec} \right. \quad \left. \begin{array}{l} \text{max rot. del} \\ \text{bit transfer hızı} \end{array} \right\}$$

Soru 8-) 1 blokta erişim süresi nedir? (1 blok = 2 bölüm (sector), okuma süresi) (Avg. seek time = 10 msec.)

$$\text{Erişim zamanı} = \text{Seek Time} + \text{rot. del} + \text{transfer time}$$

$$\text{Blok transfer zamanı} = \frac{11.1}{50/2} = 0.44 \text{ msec.}$$

↑ max rot. del  
↓ blok sayısı

$$10 + 5.55 + 0.44 = 15.99 \text{ msec}$$

↓ seek      ↓ rot      ↓ transfer

Soru 9-) Sıralı 100 bloğ'a ulaşma zamanı

$$\{ 40 + 5.55 + 100 * 0.44 = 59.55 \text{ msec} \}$$

↓      ↓      ↓  
seek    rot    transfer

Soru 10-) Rastgele 100 bloğ'a ulaşma zamanı

$$\{ 100(40 + 5.55 + 0.44) = 1599 \text{ msec} \}$$

Örnek:

Kitap = 1000 sayfa

Sayfa = 500 kelime

Kelime = 8 harf

Harf = 1 Byte

CD-ROM = 600 MB, ise bir CD-ROM'lu kaan adet  
kitap sigar?

$$\{ \text{Kitap} = 8 * 500 * 1000 \approx 4 \text{ MB} \}$$

$$\boxed{600 / 4 = 150 \text{ Kitap.}}$$

HAFTA I. SON

## RAID (Redundant Array of Independent Disks)

Disk dizisi anlamına gelir. Saklama biriminin performansını ve güvenilirliğini artırmak amacıyla birden fazla sayıda diskin bir araya getirilmesiyle oluşur. Bunu gerçekleştirmek için iki teknik kullanılır:

1) Veri Parcalama (Data Striping): Veriyi çok sayıda diskin üzerine dağıtmak.

Böylece çok büyük ve hızlı tek bir disk varmış izlenimi verilir.

→ Bu teknikte "round-robin" algoritması kullanılır. Bu algoritmada; 3 disk ve 5 bitlik bir verimiz olduğunu düşündür. 1. bit → 1. diske, 2. bit → 2. diske, 3. bit → 3. diske, 4. bit → 1. diske, 5. bit → 2. diske yazılır. Yani veri esit parçalara bölünerek disklere dağıtilır.

→ Güvensizdir. Disklerden biri bozulursa veri kaybı yaşanabilir. Güvenliği artırmak için veri tekrarlama (redundancy) kullanılır.

2) Veri Tekrarlama (Redundancy): Daha çok disk, daha çok hatayi beraberinde getirir. Veri güvenliğinin sağlanması için onun çok sayıda kopyasının oluşturulması gereklidir. Bir disk arızası durumunda kopyalar kullanılarak veri kurtarılabilir.

→ Kopya oluşturulurken yazma işlemi olacağından performans kaybı olur.

\* RAID, veri parçalama ve veri tekrarı yöntemlerini kullanan disk dizilerine verilen ismidir.

\* RAID 0-6 arasında farklı özelliklere sahip seviyelere sahiptir. Bunlar:

Level 0: Veri tekrarlama yok, sadece veri parçalama kullanılıyor.

Avantajlar:

- Gereksiz bilgi depolanmaz.
- Ucuz maliyetlidir. Çünkü, örneğin; 4 disklik bir veri için yalnızca 4 disk kullanılır  $\frac{4}{4} = \%100$ .
- En iyi yazma performansına sahiptir. Çünkü yedek bilgi tutulmaz.

Dezavantajlar:

- Güvenilirlik (reliability) azdır. Çünkü bir disk bozulursa verilerin kopyası olmadığı için veri kaybı yaşanabilir.
- Okuma performansı çok iyi degildir. Çünkü paralel okuma mümkün değildir (tek bir kopya olduğu için).

Level 1: Veriler aynalarır (Kopyalanır).

- Verinin 2 kopyası tutulur. (4 veri için 8 disk gereklidir  $\frac{8}{4} = \%50$ )
  - Okuma hızıdır. (paralel okumadan dolayı)
  - Yazma yavaştır. Çünkü 2 adet diske yazılır. (Birinci diske yazma başarılı olursa ikinciye geçilir)
  - Veri parçalama yoktur. (Veri disklere dağıtılmaz)
  - Güvenilirdir. Çünkü disk arızası durumunda elimizde bir yedek bulunur.
  - max aktarım oranı = bir diskin aktarım oranı (transfer rate)
- ↳ (Yazma hızından dolayı)
- Maliyeti en yüksek olan düzeydir. (Çünkü 2 disklik veri için 2 disk kullanılıyor)

Level 0+1: Parçalama ve aynalama yapılır. (Level 0 ve 1'in birleşmesi)

- Veri, bölünerek disklere dağıtilir. Aynı zamanda kopyası olusur.
- Yazma hızı; Level 1'den hızlı. Level 0'dan yavaştır. (Tane 2 diske yazılır)
- 4 disklik veri için 8 disk gereklidir. ( $\frac{4}{8} = 50\%$ )  
↳ Space utilization  
(Alan Kullanımı)
- Paralel okuma vardır.
- max aktarım oranı = toplam bant genişliği  
(transfer rate) (aggregate bandwidth)

\* Level 2 atlandı.

Level 3: Parite biti (Bit-Interleaved Parity)

Level 1'den sonra her levelde veri parçalama var.

- Veri bit seviyesinde parçalanır. (Veri parçalama birimi = 1 bit)
- Parite kontrolünü sağlamak için fazla bir disk bulunur. Örneğin (parity disk)  
4 disklik bir veri 5 diskte tutulur. ( $\frac{4}{5} = 80\%$ )
- Güvenilirlik tek bir disk ile sağlanır.
- Tüm okuma-yazma istekleri disklerin hepsini içenir. Bu nedenle aynı anda sadece tek işlem yapılır.
- Level 0, 1, 0+1'e göre daha iyidir.

Level 4: Parite bloğu (Block-Interleaved Parity) (Level 3'e benzer)

- Veri blok seviyesinde parçalanır. (Veri parçalama birimi = 1 blok)
- Bir tane kontrol diskii vardır. (4 disklik veri için 5 disk  $\frac{4}{5} = 80\%$ )
- Küçük istekler için paralel okuma mümkündür, büyük istekler tam bant genişliğini kullanabilir (Hızlıdır)
- Yazma işlemleri paralel yapılabilir. (Blok olduğu için)
- Yeni Parite  $\rightarrow$  bloğu = (Eski Parite XOR Yeni Parite) XOR Eski Parite.
- Level 3'ün daha iyidir.

### Level 5: Dağıtılmış parity bloğu (Block-Interleaved Distributed Parity)

- Level 4' e benzerdir. Fakat parity blokları tüm disklere dağıtılmış şekilde bulunur. (Level 4'te parity diskinde (tek disk) tutuluyordu). (Bu yüzden Level 5 daha eteffektif)
- 4 diske sağlanacak veri için 5 disk gerektir. ( $\frac{4}{5} = 80\%$ )
- Burada tek bir parity disk'i olmadığından darboğazdan etkilenmeyecek paralel yazma işlemi yapılabilir.
- Okuma isteklerinde yüksek paralellik seviyesi vardır. Çünkü tek bir kontrol disk'i yoktur. Okuma istekleri, tüm disk'i kapsayabilir.
- Tüm RAID levelleri arasında en iyi performansa sahiptir.

! Genelde RAID leveli arttıkça performans da artar.

\* Level 6 atlandı. (Read-Solomon algoritması 966 space util.)

\* Hangi RAID seviyesi seçilmeli?

Veri kaybi önemli değilse  $\Rightarrow$  Level 0

Level 0+1 > Level 1 (Güvenlik konusunda)

Level 5 > Level 4 (Veri erişimi konusunda)

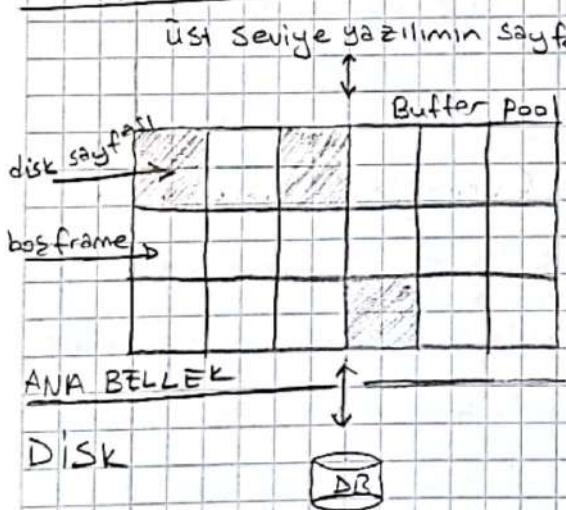
En üst seviye güvenlik için Level 6 tercih edilmeli.

## Disk Alan Yönetimi

- Veritabanı yönetim sistemlerinde, disk alanını yöneten yazılımdır.
- Sayfa bazında okuma, yazma, yer ayırma/birleştirme işlemlerini gerçekleştirir.
- Sayfaların disk üzerindeki düzenli yerleştirilme talebi, disk üzerindeki düzenli bölgelerin tahsis edilmesiyle karşılanmalıdır. Daha yüksek seviyeli katmanların bu işin nasıl yapılacağını bilmesi gerekmek. Böyle yaparak donanıma ait detaylar gizlenmemelidir.

! (Sayfa = Block)

## Tampon Yöneticisi : (Buffer Manager)



- \* İhtiyaç halinde diskten belleğe sayfaları alan yazılımdır. (Belleği olduğundan daha büyük esstir.)
- \* Bellek, sayfa boyutlarında parçalardır. (Her frame'e bir sayfa sığar)
- \* Parçaların oluşturduğu yapıya tampon havuzu (buffer pool) denir.
- \* DBMS'in çalışması için veriler RAM'de olmalıdır. (Yani buffer pool'da)

\* Diskten gelen sayfa boş frame'le yazılır. Eğer tüm frame'ler doluysa. replacement policy, hangi frame'in bozaltılıp gelen sayfanın oraya yazılacağına karar verir.

\* Buffer pool'da her frame bir id'ye sahiptir. Frame id ve sayfa id bir tabloda tutulur. (RAM'de). Böylece disk sayfaları ve buffer yönetimi sağlanır.

Üst seviye yazılım sayfa isteği gönderdiğinde =

- Eğer sayfa buffer pool'da ise direkt olarak karşılanır bu istek.
- Değilse, bir frame seçilir (replacement policy ile). Eğer frame dirty ise (yani o frame'deki data güncellenmişse) bunu diske yaz. Daha sonra boş olan bu frame'ye yeni sayfayı oku. (Yani dolu frame'i buffer'den diske yazıp istenen vei için yer açılıyor).
- Ardından istenen sayfa pinlenir ve adresi döndürülür. (Aslında pin-count adında bir değişken tutulur. Bu değişken istenen sayfaların sayısını ifade eder. istenen sayfanın pinlenmesi pin-count 1 arttır anlamına taşır)
- ★ Eğer istek öngürebilir bir istekse (örneğin ardışık tarama yapılıyorsa) bazı sayfalar çağırılmadan (istek yapılmadan) getirilebilir (bu işlem pre-fetching denir). Zamanlı tasarruf edilmiş olur.
- Sayfa isteği tamamlandıktan sonra pin-count 1 azaltılır bu işlem unpinning denir.
- Eğer sayfa değiştirilmisse dirty bit 1 olur ve diske geri yazılır.
- Havaudaki bir sayfa birden fazla kez istenebilir. Bu durumda pin-count kullanılır. Her çağırıcı pin-countu 1 artırır. işlem sonlandığında ise 1 azaltılır. Bir sayfa ancak ve ancak pin-count=0 ise yerlestirilmeye adaydır. (buffer pool'da bu sayfayı isteyen bir işlem yoksa)

\*<sup>4</sup>  
\* ÖZET \*  
\*

Veritabanı yönetim sistemi, büyük boyutlu veritabanlarını üzerinde işlemler yapabilirler. Veritabanı, içinde büyük mikarda verinin saklandığı bir dosyadır.

Örneğin ; öğrenci işleri veritabanı yönetim sisteminde tüm öğrencilerin bilgilerinin tutıldığı dosyayı, bir öğrenci veritabanı olarak düşünebiliriz. Bu dosya 2 birden fazla diske sağlanabilecek büyüklükte olabilir. Bu gibi durumlarda RAID sistemleri kullanılır.

Öğrenci işleri bilgi sistemi üzerinde bir öğrenci ders kaydı yapmak istediğiinde ;

Bu kaydı yapmak için sisten, öğrencinin bilgilerini ister. Öğrenci işleri yönetim sistemi programında, öğrencinin diskteki kaydı için bir istek gönderilir. Bu istek sonucunda bu öğrencinin kaydının hangi diskte, hangi trackte, hangi blokta olduğu bilgisi disk alan yöneticisi tarafından tespit edilir.  
(disk space manager)

Ardından bu blok RAM'de buffer pool denilen alana aktarılır.

Buffer pool frame'lerden oluşur, her frame'de bir disk sayfası (bloğu) vardır. Buffer pool'da boşluk varsa blok o boşluğa aktarılır. Fakat hiç yer yoksa (bütün frame'ler doluyra) o zaman bir page replacement algoritmasına göre boşaltılacak frame seçilir. Seçilen frame'deki dirty bit 1 ise (yani veri güncellendi) o güncellmemeyi kaybetmemek için öncelikli o blok diske yazılır. Daha sonra, diskten istenen blok boşaltılan frame'e aktarılır. Bu işlemlerin yapılması için 2 değişken kullanılır. dirty bit ve pin-count.

pin-count: buffer pooldaki bir blok için o bloğun üzerinde kaç istek olduğunu bite gösterir. O yüzden yerine yerlestireceğimiz frame'deki pin-count'un sıfır olması lazımdır. (Yani o sayfayı kimse kimse istememesi lazımdır). O sayfa için istek olduğunda pin-count 1 artırılır. Bu da pinning denir. İstek sağlandığında pin-count 1 azaltılır. Bu da unpinning denir. Dirty bit ise o sayfadaki verilerin değişip değişmediğini kontrol ediyor. Dirty bit 0 ise veriler değişmemiştir. Onun üzerine yeri bir veri okunacaksa o eski bloğu diske geri yazmaya gerek yok. Çünkü veri zaten değişmemiştir. (Disk'teki original hali gibi duruyor). Fakat dirty bit 1 ise ve üzerine yeri blok yazılmak istenirse öncelikle eski blok diske yazılır daha sonra üzerine yeri blok yazılır.

Tam bu işlemleri veritabanı yönetim sisteminin (DBMS) file manager denilen katmanı yapıyor. File manager içerisinde buffer management, disk space management var.

## Buffer Pool'a Yeni Sayfa Alma Yöntemleri ; (Buffer Replacement Policies)

- Eğer buffer pool doluyسا, replacement policy belli bir yöntemle göre istenilen sayfayı pool'a alır. Bu yöntemler ; LRU(Least Recently Used), MRU(Most Recently Used), Clock, Random... Policy, erişim modeline bağlı olarak I/O sayısında (disk erişim) büyük etkiye sahip olabilir.
- Ardışık okuma yapılarken # buffer frame < # dosyadaki sayfa olursa MRU kullanmak daha iyidir.

Örnek: 10 frameli bir buffer pool ve 11 adet ardışık sayfa olsun. Öncelikle;

1	2	3	4	5
6	7	8	9	+

bos olan framelerde sayfalar sırayla teker teker yerlesir. LRU metodunda tüm framelerin dolu olması halinde yazılacak sıradaki sayfa ilk yazılan sayfanın (+) yerine yazılır.

11	2	3	4	5
6	7	8	9	+

Bu durumda ardışık okuma yapılacak zaman 1. sayfaya ihtiyaç vardır. tekrar LRU kullanarak 2 yerine + yazılır. Bu kez de 2 tablodan kaybolur. Onu getirmek için 3 yazınca 2 yazılır. Bu böyle devam eder.

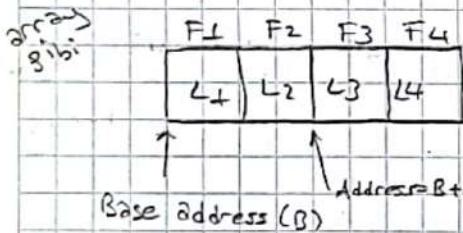
MRU kullanılsaydı sadece sağ alt değişecekti. (1 ve 10)

## DBMS vs İşletim Sistemi Dosya Yönetimi:

- DBMS yazılımları, işletim sisteminde bağımsız olmalıdır. (Taşınabilir olmalıdır)
- İşletim sisteminin dosya sisteminde dosyalar diske dağıtılmaz. Yani dosya boyutu, disk boyutlarından küçük olmalıdır.  
Fakat DBMS'de dosyalar bir diske sığamayacak kadar büyük olabilir. Bu durumda dosya, disk'e dağıtılr.
- DBMS'de buffer yönetimi bunları yapma becerisi gereklidir.
  - Buffer pool'a bir sayfa, pinle ve bir sayfa, diske yazılmasına zorla.
  - Replacement policy'li ayırtlamak, pre-fetch işlemi.

! DBMS'de dosyalar, kayıtlardan (records) meydana gelir.

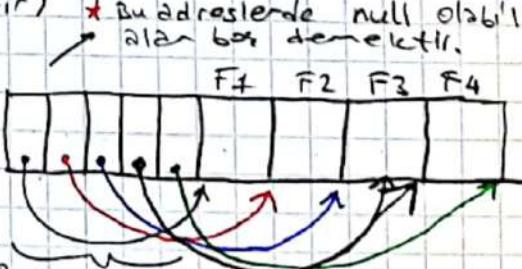
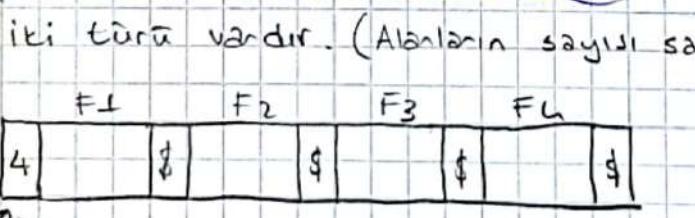
## Kayıt Bicimi : Sabit Uzunluk (Record Formats: Fixed Length)



\* Kayıtlar sabit uzunluktan oluşmustur.  
Yani bu bit' öğrenci kayıt programı olsaydı  
L1 iin sabit uzunluk (örn öğrenci no 10 bayt)  
L2 iin sabit uzunluk (örn isim 400 bayt)

\* O alanının bilgileri (örn kaan bayt olduğu) başka bir dosyada  
tutur (sistem katalogunda)

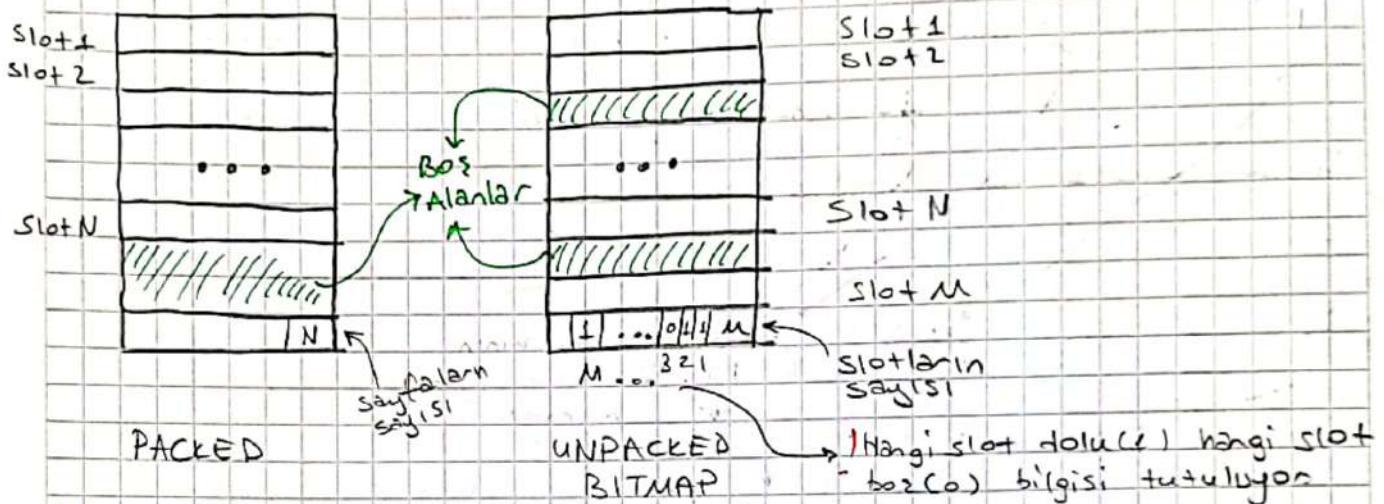
## Kayıt Bicimi: Değişken Uzunluk (Record Formats : Variable Length)



\* Burada adresler tek tek karakterler tutulur.  
\* Hizlidir. Karakterler tek tek kontrol edilmez. Alan başlangıç adresine sahibiz.  
istediğiniz adresse gitiriz.

Sayfa Biçimi : Sabit Uzunlukta Kayıtlar : (PageFormat: Fixed Length Records)

Gelen sayfadaki kayıtlar, sayfalarda (bloklarda) tutulur.



\* Sabit uzunlukta slotlara bölünürler ve her slotta bir kayıt (record) vardır.  
Kayıtların uzunlukları esittir.

\* Her slot birbirine komşudur.  
Aralarında boşluk yoktur.  
Eğer bir slot silinirse N. slot o slotun yerine gelir, ve N + 1 artar.  
Eğer yeri bir slot eklenirse boş alanlardan 1 slotluk yer alınır ve N + 1 artırılır.

\* Yeri bir kayıt ekleneceğinin zamanı slotların bilgisinin tutulduğu kısım (en alt) kontrol edilir. Eğer 0 olur (yani boş) bir slot varsa oraya yerleştir ve o slotun bilgisi 1 olur.  
Silineceği zaman, o slotun bit'i 0 yapılı (record id değişmez)

\* Digerine göre daha iyi bir dizayndır.  
(Record id problemi yok). Dezavantajı slotların bitlerinin tutulması yer maliyeti getirir (cok az da olsa)

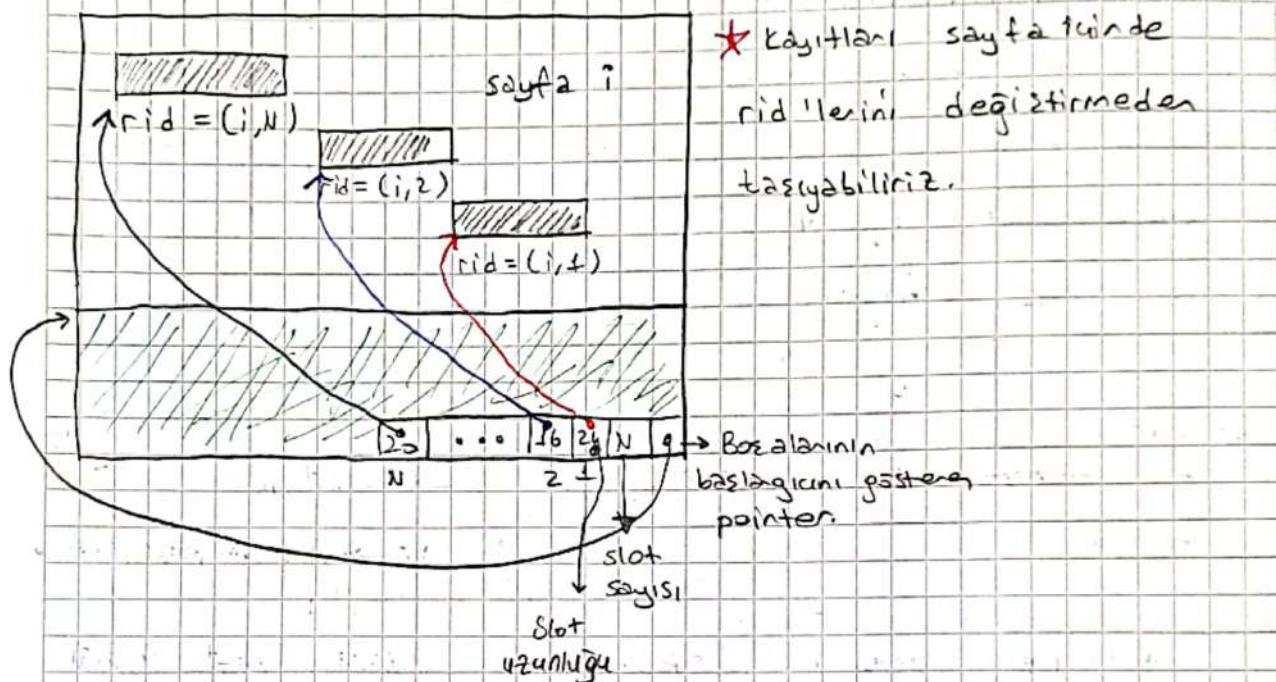
\* record id = < page id, slot # >  
(kayıt id)

İlk (packed) seviyedeki bir kayıt silindiğinde sonraki kayıt onun yerine geleceğinden record id verme konusunda sıkıntı oluyor.

Örneğin 2. slot silindiğinde N. slot onun yerine gelecektir. rid  
(rid=52) (rid=SN)  
pageid

unique olmalıdır. Bu bir probleme yol açabiliyor.

### Sayfa Blimi: Değişken Uzunlukta Kayıtlar: (Page Formats : Variable Length Records)

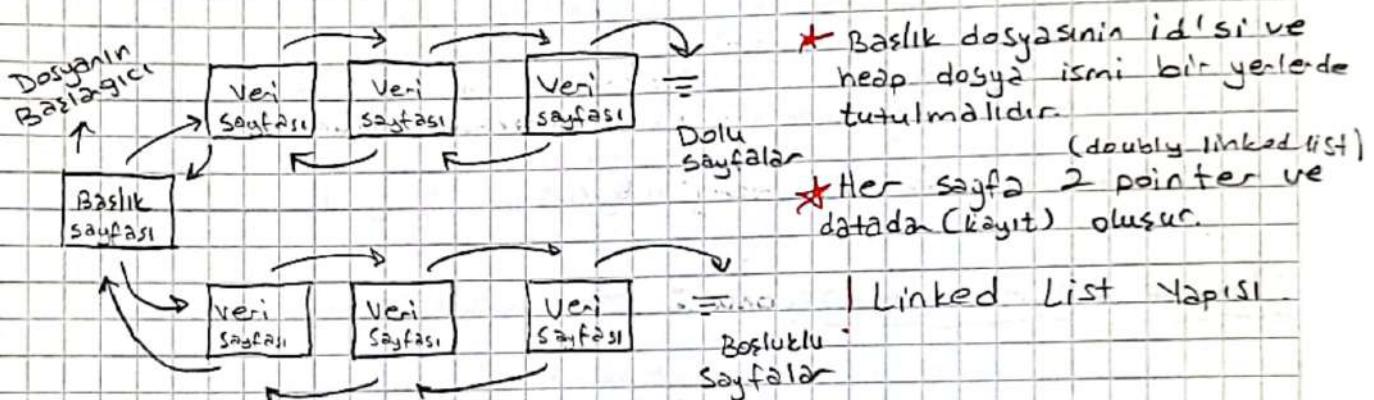


### Kayıt Dosyaları: (Files of Records)

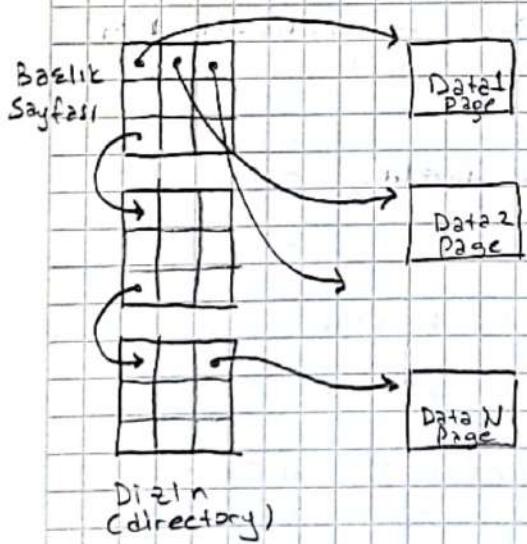
- DBMS diske veri yazma, diskten veri okuma gibi işleri yapar. Fakat daha yüksek seviyeli DBMS katmanları kayıtları ve kayıt dosyalarını yönetir.
- Her dosya sayflardan, her sayfa kayıtlardan oluşur.
- DBMS sunları desteklemelidir:
  - Kayıt ekleme / çıkarma / değiştirme
  - belirli bir kaydı okuma (record id ile)
  - tüm kayıtları tarama

## Sırasız (Yığın) Dosyalar : (Unordered (Heap) Files)

- En basit dosya yapısıdır. Belirli bir sırası olmayan kayıtlardan oluşur.
- Dosyanın boyutu değişikçe (büyütüp - küçültükçe) allocation ve de-allocation işlemleri olur.
- ★ Liste olarak uygulanan yığın dosyaları :



## Sayfa dizini kullanan yığın dosyaları :



★ Her dizin bölünmüştür. Bu bölümlerde data sayfalarının adresleri ve o sayfa tarafından ulaşılabilir boş alanları tutar.

★ Yeni bir kayıt eklenecinde, başlık sayfasından başaranak dizin aranır. Yeterli büyüklükte bir kayıt alanı bulunduğu yerleştirilir. Örneğin; Data2 sayfasında boşluksa bu sayfa buffer poola alınır. Veri yazılır. Ardından diske yeniden kaydedilir. Buna bitince o sayfanın boş yer bilgisi güncellenecektir.

★ Bu kullanım liste kullanımına göre daha忌くktür. (boyut olarak)

## Temel Dosya Yapısı Kavramları

Dosyalar en iki biçimde görülebilir:

- Bir bayt akışı şeklinde (Stream File)
- Alanlara sahip bir kayıt koleksiyonu

### Aralıksız Dosya (Stream File):

- Dosya, baytların aralıksız bir setinde dizilmesiyle oluşur.

87359 Carroll Alice in wonderland 38180 Folkfile Structures

- Veri anlamını yitirmiştir. Bu veriyi anlamının bir yolu yoktur!

### Alan ve Kayıt Organizasyonu: (Field and Record Organization)

- Dosya, kayıtlardan oluşur.
- Kayıtlar, ilişkili alanlardan oluşur. (Örneğin bir kitap kaydı ISBN numarası, yazar, isim --- gibi alanlarda meydana gelir)
- Alan(Field), bir dosyanın en küçük, anlamlı ve mantıksal birimidir.
- Anahtar (key), kaydın içindeki bir alanıdır (field) o kaydın id'sini tutar.
- Bir kitap dosyasında, her satır bir kayda karşılık gelir. Her kaydatta ISBN, yazar, isim gibi alanlar vardır.

Bu örnekte ISBN numarası bir anahtardır (key). Çünkü her kitabın ISBN'si farklıdır.

## Kayıt Anahtarları : (Record Keys)

Birincil Anahtar : 0 kaydı eşsiz bir şeilde tanımlayan anahtar.  
(Primary key)  
 Örneğin ; ISBN, Öğrenci id'si, TC kimlik no.

İkincil Anahtar : Arama için kullanılan anahtarlardır. Bir kitabı  
(Secondary Key)  
 ISBN'ıme göre değil adına, yazarında göre aranız iste bunlar  
 ikincil anahtarlarıdır.

! Genel olarak her alan bir anahtar değilidir. (Anahtarlar, bir aramada  
 kullanılabilecek alanlara veya alanların bir kombinasyonundan karşılık  
 gelir). Örneğin sayfa sayısı bir key değildir.

## Alan Yapıları : (Field Structures)

### - Sabit Uzunluklu Alanlar (Fixed-length Fields) :

Her alan için belirli bir uzunluk vardır.

• 8 7 5 3 9 Carroll Alice in wonderland  
 • 3 8 1 8 0 Folk File structures

### - Alanın Başında Uzunluk Göstergesi (Length Indicator) :

Her alanın uzunluğunu başında yazıyor

• 05 87539 07 Carroll 19 Alice in wonderland.  
 • 05 38180 04 Folk 15 File structures.

### - Her Alanın Sonunda Özel Karakter (Delimiter) :

Her alanının sonunda onun sonlandığını gösteren karakter var.

• 87359 | Carroll | Alice in wonderland |  
 • 38180 | Folk | File structures |

### - Anahtarla Tutan Alan :

• ISBN=87359 | A4=Carroll | T1=Alice in wonderland |  
 • ISBN=38180 | A4=Folk | File structures |

Alan Yapılarının Karşılaştırılması :

Tip.	Avantaj	Dezavantaj
Fixed-length	★ Dikte okuyup diske yazması kolay.	★ Boşluklar (padding) boşuna yer kaplıyor.
Length-based	★ Alanın sonuna gitmek kolay (çünkü alan boyluğun veildi)	★ Bünyük alanların sayısını yazmak için fazla bayt gereklidir.
Delimited	★ Length-based'e göre daha az alan harcar	★ Özel karaktere karşı her alan bayti kontrol edilmeli.
Keyword	★ Alanlar iyi bir şekilde tanımlanıyor. Eksik alanlara izin veriliyor.	★ Anahtarlar için bir yer ayrılmıyor. (waste space with keywords)

Kayıt Yapıları : (Record Structures)Sabit Uzunluklu Kayıtlar (Fixed -Length Record) :

iki türlüdür. (Her satır bir kayıttır)

87359	Carroll	Alice in wonderland
03818	Folk	File Structures

1) ↳ Sabit kayıt sabit alan  
(record) (field)

87359   Carroll   Alice in wonderland	Kullanılmaz
03818   Folk   File Structures	Kullanılmaz.

2) ↳ Sabit kayıt değişken alan

## Değişken Uzunluklu Kayıtlar (Variable-length records):

### Sabit Alan Sayısı:

87359 | Carroll | Alice in wonderland | 38180 | Folk | File Structures | ...

→ Burada her kaydın 3 alandan oluşanluğu biliniyor. Bitiş sembollerini sayılarak kayıtlar alınabilir.

### Uzunluğu Verilen Kayıtlar:

3387359 | Carroll | Alice in wonderland | 2638180 | Folk | File Structures | ...

→ Burada her kaydın önünde o kaydın uzunluğu yer alıyor.  
Diğer kayda geçilmek istendiğinde bu, digerine göre daha hızlıdır.

### Kayıt Yapılanının Karşılaştırılması:

Tip	Avantaj	Dezavantaj
Fixed-Length	* 1inci sıradaki kayda girmek kolaydır. (üçüncü kayıtla sınırlı)	* Boşluk (padding) yüzünden yer kaybı.
Variable-Length	* Yerde tasarruf sağlar (kayıt boyutları farklı olsa da) iin	* Bir index dosyası olmadıkça 1inci kayda ulaşamazsın (index dosyası hafıza)

Hafta II. Son

## Dosya Organizasyonu:

### Sıralı (Yığın) Dosyaları : (Sequential (pile) Files)

- Kayıtlar, depolama cihazında bitisik olarak bulunur.
  - Ardışık dosyalar başlangıçtan sona kadar okunur.
  - Ardışık dosyalarda bazı işlemler kolaylıkla yapılabilir (ortalaması, minimum, maximum bulmak vs...)
- İki çeşittirler

- Sıralanmamış ardışık dosyalar. (pile files)  
(Unordered sequential files)
- Sıralanmış ardışık dosyalar. (Kayıtlar belirli bir seye göre sıralardır.)  
(sorted)

### Yığın Dosyaları: (Pile Files)

- Bir yığın dosyası, herhangi bir ek yapı olmaksızın, basitçe birbirini ardına yerleştirilen bir dizi kayıttır.
- Kayıtların uzunlukları değişebilir.
- istenilen kaydı bulmak için ilk kayıtтан başlayarak kayıt bulunana kadar ardışık (sequential) arama yapılır. (Buffer pool'daki frame'ler üzerinde arama yapılır)

### Sıralı Dosyalarda Arama: (Searching Sequential Files)

Bu konuda kümük  $b \rightarrow$  blok sayısına karışıklık gelecek.

- En iyi durumda aradığımız kayıt t. sırada olabilir.
- En kötü durumda aradığımız kayıt sonuncu ( $b$ 'inci)sırada olabilir.
- Ortalama durum. →  $\frac{1}{b} * b(b+1)/2 \Rightarrow \underline{\underline{b/2}}$

$b$  farklı  $\underline{\underline{j}}$ 'den  $\underline{\underline{b}}$ 'ye kadar  
olası durum olan tüm bloklar.

\* Sıralı olmayan bir ardışık dosyada bir kaydı bulmak için ortalaması süre:

$$TF = S + R + (b/2)^* btt$$

## Terimler : (Nomenclature)

s : average seek time (ortalama arama süresi)

r : average rotational delay (ortalama döndürsel gecikme)

btt : transfer time (Aktarım zamanı (1 bloğu aktarmak için geçen süre))

b : number of blocks in the file (Dosyadaki blok sayısı)

Bfr : Blocking factor (Bir bloktaki kayıt sayısı)

B : size of a block (Bir bloğun büyüklüğü)

R : size of a record (Bir kaydın büyüklüğü)

$$\star \frac{B}{R} = Bfr \quad \text{Örneğin; } B = 2000, R = 600 \text{ olsun bu durumda}$$

$Bfr = \frac{2000}{600} \approx 3.3$  ama tam sayı olması için tam sayı olmalı onun için floor işlemi yaparak  $Bfr = 3$  olur

## Dosyanın Ayrıntılı Okunması: (Exhaustive Reading of the File)

Tüm kayıtları okumak ve işlemek için geçen süre:

$$\boxed{T_x = s + r + b * btt.}$$

$\star$  Ortalama bulma, minimum, maximum, toplama gibi işlemler için tüm kayıtların incelenmesi ve izlenmesi gereklidir.

$\rightarrow$  Örneğin; personel kayıtlarının olduğu bir dosyada ortalama maaşı bulmak için ilk kayıttan son kayda kadar tüm maaşlar toplanır ve kayıt (kisi) sayısına bölünür. Bu işlem için ayrıntılı okuma yapılmıştır. Bu işlem için geçen süre, formülde verilen sürecdır.

### Yeni Kayıt Ekleme : (Inserting a New Record)

Yeni kayıt eklerken sırasız olduğu için, ekleme algoritması kaydı sona ekler.

\* Dosyanın son bloğu buffer pool'da bir framelinin içine aktarılır. (Sonuncu sayfada boş yer olduğunu varsayıyoruz) bu sayfaya yeni kayıt eklenir. Bu işlem için gereken süre :

$$T_I = \boxed{s+r+btt} + (2r - btt) + btt$$

sonuncu bloğu diske yazıp sonra  
disk hattının sonuna gelmesi için zaman  
(yazma olmadan)

Sonuncu bloğu okumak için. yazma  
süresi

$\Rightarrow s+r+btt+2r$

? Eğer sonuncu blok doluya?

Öncelikle  $s+r+btt$  kadar zamanda sonuncu bloğu belleğe okur. Kontrol eder ve sonuncu blok doluya, bellekten yeni bir boş blok alınır. Yeni blok oraya yazılır ve bu blok diske geri yazılır. Bunun için de bu yeni bloğu diske geri yazmak için gereken süreyi de buna eklememiz gerektir.

### Kayıt Güncelleme : (Updating a Record)

\* Burada alınan süreler disk erişimi için gereken sürelerdir. İstenci

süresi, diske göre çok hızlı olduğundan bu süre hesaba katılmaz.

Güncelleme süresi :

$$T_u \text{ (sabit uzunluk)} = T_F + 2r$$

framelerde güncellenecek  
bloğu diske yazma süresi

Once güncellmek istediğimiz kaydı buluruz. Bu kaydın olduğu bloğu buffer pool'a aktarırlı. Burada güncellir ve diske geri yazılır.

$$T_u \text{ (değişken uzunluk)} = T_D + T_I$$

süresi  
ekleme

Kaydı güncellediğimizde kayit uzunluğun değişebilir. Once eski kayıt silinir. Daha sonra güncellmiş kayıt dosyaya yeni kayıt gibi eklerir.

## Kayıt Silme : (Deleting Records)

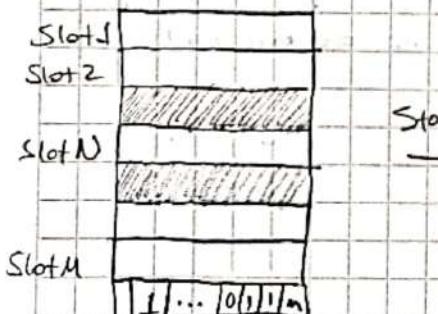
Kayıt silme işlemi diğer işlemlerden biraz farklıdır. Çünkü bir kaydı silmek istediğinizde onu direkt dosyadan çıkarıp atamıyoruz. Silme işlemini yapmak için 3 tane algoritma vardır. DBMS bunlardan bir tanesini kullanır.

1-) Kayıt Silme ve Depo Sıkıştırma : (Record deletion and Storage Compaction)  
Silmek istenen kaydın başına onun silindiğini gösteren bir işaret konur (Yıldız (\*) gibi) veya sayfanın sonundaki slot dizinine özel bir bit koymabilir. ( $0 \rightarrow$  kayıt silindi gibi)

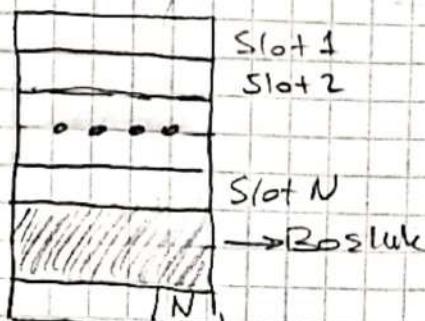
Bu işaret konulunca kayıt aslında orada kalmaya devam ediyor. (Üstüne yeni bilgi yazılmadığı sürece). - Yani silinen bilgi geri alınabilir.

Bu işlem fazla yapılınca sayfada boşluklar olusur. Bunun için depo sıkıştırma denen bir algoritma uygulanır. Belirli bir sayıda kayıt silindiğten sonra DBMS tarafından bu algoritma uygulanır. Bu algoritma, sayfaları tek tek okuyup silinmemiş kayıtları başka bir sayfaya ardışık olarak ekler. Böylece örneğin; depo sıkıştırmadan önce 100 bloğa sahip bir dosya içerisinde sonra 70 bloğa sağlanır.

## Storage Compaction : (Depo Sıkıştırma)



Storage Compaction

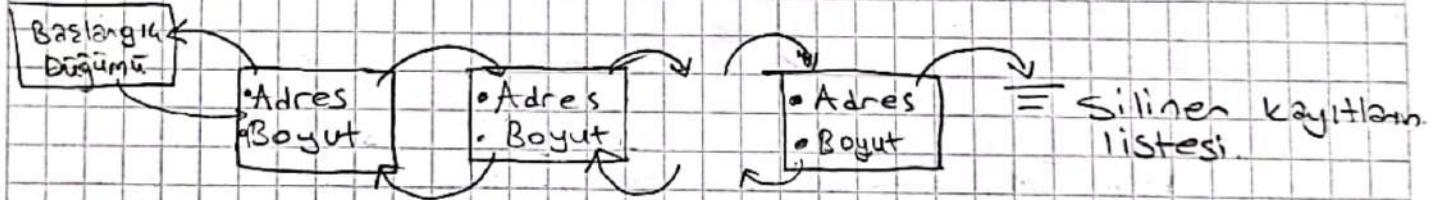


## 2-) Sabit Uzunluklu Kayıt Silme : (Deleting fixed-length records and reclaiming space dynamically)

Sabit uzunluklu kayıtlar silinirken öncelikle yine silinen sayfaya bir işaret konur. Ardından silinen kaydın adresi bir bağlı listede tutulur. Bu listeye AVAIL LIST (yazar listesi) denir. Yeni bir kayıt eklenceği zaman bu listeye bakularak boş yere gidip yazılır. (Herhangi bir depo sıkıştırma işlemi yapılmaz)

## 3-) Değişken Uzunluklu Kayıt Silme : (Deleting variable length record)

Burada yine işaret konur, AVAIL LIST'e eklenir. Bu kez adresin yanında bu kaydın büyüklüğü de AVAIL LIST'e eklenir. Büyüklüğünün eklenmesinin nedeni, yeni kayıt ekleneceğinde yeni kaydın boyutunun oraya sığıp sığamayacağını belirlemek.



## Yerleştirme Stratejileri : (Placement Strategies)

\* Bu stratejiler, değişken uzunluklu kayıt silmeden, yeni kayıt eklenirken AVAIL LIST'ten hangi yerin alınacağını belirler.

1-) First-fit : AVAIL LIST'teki elementler bir sıraya göre sıralanmış değil (kayıt silindiğinde kaydın adresi listenin sonuna eklenir). Kayıt eklenmek istendiğinde baştan başlanarak yeterli büyüğüye sahip ilk yere eklenir.

2-) Best-fit: AVAIL LIST'e silinen kayıtların boyutları küçükten büyüğe doğru sıralanır. Yeni bir kayıt ekleneneğinde boyutlar arastırılarak o kaydın sigabileceği ilk elementin adresi listeden alınarak o adrese gidip kayıt eklenir. Daha sonra o node listeden çıkarılır.

3-) Worst-fit: Silinen kayıtların boyutları büyükten küçüğe sıralanır. Yeni eklenenek kayıt listenin ilk adresindeki yere eklenir (boyut  $>$  yeni kayıt boyutu olması şartıyla). Sonra kullanıcılar alan kadarlık yer boyuttan çıkarılırak ve adresi güncellenerek tekrar AVAIL List'e eklenir. (boyut sırası bozulmadan)

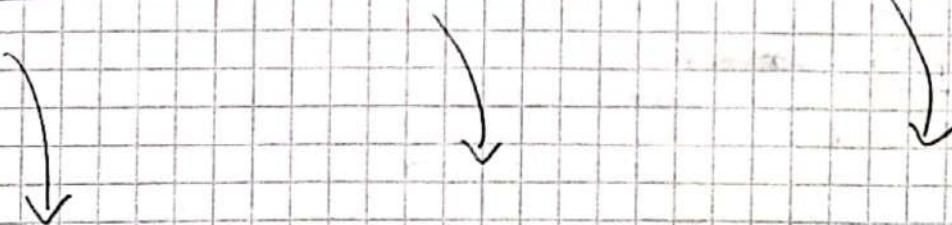
! Bazı araştırmalarda göre worst-fit'in daha iyi olduğu genülmüşür.

### PROBLEM 1

→ (original dosyada)  
b tane bloktan oluşan bir dosya olsun. Bu dosya üzerine (strategic compaction) depo sıkıştırma uygulamak için geçen süreyi hesaplamak için bir formül oluşturun. ( $n \rightarrow$  blok sayısı)

- i) 1 disk sürücüsü için  
(yeni dosyada)
- ii) 2 disk sürücüsü için

Gözüm:



### Algoritması

Dosyadaki her blok ( $A$ ) için :

Bloğu ( $A$ ) oku

Buffer pool'da boş bir blok ( $B$ ) ayır

$A$  bloğundaki her kayıt ( $r$ ) için :

Eğer  $r$  silinmişse :

$A$  ile (skip).

Else :

Eğer  $B$ 'de yeterli yer varsa :

$r$ 'yi  $B$ 'ye yaz.

Else :

$B$ 'yi diske yaz. ( $B$ 'yi boşalt)

$r$ 'yi  $B$ 'ye yaz.

→ i) Bir disk sürücüsü varsa okuma ve yazma işlemi random olur. Bu yüzden, orijinal dosyayı okumak için gerek süre :

$$\{ b * (s+r+btt) \} \text{ Okuma Süresi}$$

Diske yazmak için gerek süre :

$$\left\{ \left[ \frac{n}{Bfr} \right] * (s+r+btt) \right\} \xrightarrow{\text{üste yararıza } (\frac{10}{2} = 5 \text{ çıkar})} \text{ Yazma Süresi}$$

$$\Rightarrow \left\{ \left( b + \left[ \frac{n}{Bfr} \right] \right) * (s+r+btt) \right\} \text{ Toplam Süre}$$

→ ii) Eğer iki disk sürücüsü varsa. (original dosyanın 2. sürücüde (drive) olduğunu varsayıyalım). Buradan bir bloğu buffer pool'a okuyacağız. Buradaki kayıtları buffer pool'da, yeni bir bloğa aktaracağız. Yeni blok dolduğu zaman ikinci sürücüye gidip bu bloğu yazacağız. Dolayısıyla her iki diskte de ardışıl bir okuma-yazma vardır.

Birinci diskten okuma süresi:

$$\left\{ S + r + b * btt \right\} \text{ Okuma Süresi}$$

İkinci diske ardışıl yazma süresi:

$$\left\{ S + r + \left\lceil \frac{n}{Bfr} \right\rceil * btt \right\} \text{ Yazma Süresi.}$$

$$\Rightarrow \left\{ 2S + 2r + \left( b + \left\lceil \frac{n}{Bfr} \right\rceil \right) * btt \right\}$$

Toplam Süre

## PROBLEM 2:

A ve B yiğin(pile) dosyalarıdır. Her ikisinde de 100.000 tane kayıt bulunmaktadır ( $n=100.000$ ) ve bu kayıtlar herhangi bir sıralanmaya göre sıralanmamış. A ve B dosyalarındaki kayıtların yaklaşık  $\approx 70$ 'lu ortaktır. Bu iki dosyanın kesişiminin (ortak elemanlarının) bulan ve bunu diske yazan algoritmanın çalışma süresi nedir? (her kayıt ( $R$ ) = 400 byte, bu işlem için erişilebilen bellek  $= 10 \text{ MB}$ ,  $S = 16 \text{ ms}$ ,  $r = 8.3 \text{ ms}$ ,  $btt = 0.84 \text{ ms}$ , bir blok ( $B$ ) = 2400 byte)

Algoritması 1: öncelikle her iki dosyanın da en başına gitilir.

A dosyasının sonuna gelinmediği sürece :

A'dan M ardışık bloğu buffer pool'a oku.

B dosyasının sonuna gelinmediği sürece :

B'den buffer pool'a 1 blok oku.

A ve B'deki kayıtları karşılaştır.

Eğer eşlesen kayıt bulunursa :

Bu kayıt buffer pool'da C bloğuna yaz.

C bloğu doldugu zaman diske yaz (veya C bloğunu boşalt)

\* Burada kayıtlar herhangi bir kriterde göre sıralanmadığı için A dosyasından okunan bir blok (veya kayıt) B'nin tüm elemanları içinde aranır.

- Buffer pool'a kaç sayfa sigar?
- 1 frame, B dosyasında blok okumak için boyut  
↳ buffer pool boyutu  
↳ aradığı yuvaya (frame)  
↳ bir sayfanın boyutu
- 1 frame, ortak kayıtları yazmak için kullanıyor.
- Geriye kalan 4164 frame A'dan blokları okumak için kullanıyor.
- Bir sayfada kaç kayıt var?  
 $bfr = \left\lceil \frac{B}{R} \right\rceil = \left\lceil \frac{2400}{400} \right\rceil = 6$   
↳ Sayfa boyutu  
↳ kayıt boyutu
- Her bir dosya (A ve B) kaç sayfa (blok) 'tan oluşur?

$$\left\lceil \frac{n}{bfr} \right\rceil = \left\lceil \frac{100.000}{6} \right\rceil = 16667 \rightarrow \text{Sayfa var 1 dosyada.}$$

↳ kayıt sayısı  
↳ sayfadaki kayıt sayısı

- A dosyasını baştan sona okumak için geçen süre :

$$A = \left\lceil \frac{16667}{4164} \right\rceil * (str + 4164 * btt)$$

↳ buffer pool'da A için bu kadar alan var.

- $B'$  den okumak için geçen süre :

$$B = \lceil \frac{16667}{5} \rceil * (s + r + 1 * btt) \Rightarrow \lceil 16667 * (s + r + btt) \rceil$$

↳ Buffer pool'da  $B$ 'in bu kadar alan var

- Kesiciim dosyasını diske yazmak için geçen süre :

$$\left\lceil \frac{100.000 * 0.70}{6} \right\rceil * (s + r + btt)$$

Kesiciim dosyasındaki sayfa sayısı

\* Bu üç zaman birimini topladığımızda -kesiciimleri bulup bunu diske yazma süresini hesaplamış oluruz.

### Sıralı Ardışılı Dosyalar (Sorted Sequential Files)

- Kayıtlar belirli bir kriter'e göre sıralı olarak bulunur. Sayfalar, disk üzerinde ardışılı pozisyondadır. Örneğin; öğrenci kayıtlarının bulunduğu dosyada kayıtlar, öğrenci numarası yada öğrenci ismine göre sıralı olarak bulunuyor.
- Dosya sıralı olduğunu, yeni bir kayıt eklemek istendiğinde sıralı yapıyı bozmamak için kaydırma (shift) işlemi yapmak gerekiyor (araya ve basa eklenirken). Ancak dosya üzerinde shift işlemi yapamayız. Bu yüzden yeni kaydı tasma alanına (overflow area) ekleriz.

- Taşıma alanında kayıtlar sıralı bulunuyor. (Eklenen) sıradı bulunuyor.)
- Bu tarz bir dosya yapısında kayıt aranırken önce sıralı olan alanında arama yapılır. Eğer orada bulamazsa overflow (taşıma) kısmında arama yapılır.
- ! Eğer çok fazla ekleme yapıldıysa overflow kısmının boyutu çok büyük olur. Aramak için geçen süre sanksi unordered (sırasız) ardışılı dosyadaki gibi olacaktır.

★ Bu sebepten dolayı, bu dosya yapısında belirli sayıda eklenmeden sonra tekrar düzelleme yapılarak kayıtlar sıralı hale getirilir.

### Kayıt Arama : (Searching for a record)

Sıralı ardışılı dosyada arama yaparken, sıralı kism için ikili (binary) arama, overflow kismi için sequential aramayı kullanırız.

Sorted part  
x blok

overflow  
y blok

dosyadaki  
toplam blok  
 $(x+y=b)$  sayısı

→ Tüm sorted (sıralı) kismını aramak için :

↗ binary search olduğu için

$$TF = \underline{\log_2 X * (\text{str} + \text{btt})}$$

Ortalıki  
kayıt least  
olmak kabul  
edilir.

→ Tüm overflow (taşıma) kismını aramak için :

$$TF = \underline{s + r + (y/2) * btt}$$

⇒ Toplam:  $\left\{ TF = \log_2 X * (\text{str} + \text{btt}) + s + r + (y/2) * btt \right\}$

PROBLEM 3:

Blok boyutu = 2400 byte

Dosya boyutu = 40MB

Blok aktarım süresi (btt) = 0.84 ms

S = 16 ms

T = 8.3 ms , olsun.

1-) Pile file için arama süresi ve sorted file için (overflow olmayan) gerek süreleri hesapla. (En kötü durum için)

Pile File için TF :  $\star$  Ortalama oluydu  $S+r+b \cdot btt$  olurdu.

$$TF = S+r+b \cdot btt = 16 + 8.3 + \left[ \frac{40\text{MB}}{2400\text{byte}} \right] * 0.84 = 14.024.5\text{ms}$$

$$\approx 14 \text{ saniye} \quad \star$$

Sorted File için TF :

$$TF = \log_2 \left( \frac{40\text{MB}}{2400} \right) * (S+r+btt) = 15 * (16+8.3+0.84) = 377.1\text{ms}$$

$$\approx 0.38 \text{ saniye} \quad \star$$

2-) 10000 kayıt aranırsa :

$\star$  Eğer bu arama 10k kayıt için yapılacaksa yukarıdaki her bir sonuc 10k ile çarpılır. Bu durumda aradaki farklı git gide büyüyecektir.

$\star$  10k sayıda arama işlemi yapılacaksız sorted seq file kullanmak daha avantajlıdır.

## Eş Sıralı İşleme : (Co-sequential Processing)

- Co-seq proc, tek bir dosyası oluşturmak için iki veya daha fazla sıralı dosyanın koordineli olarak işlenmesini içeri.  
Bunun içi türü vardır: Matching (intersection) ve merging (union)

### 1-) Matching:

- i) Kesim dosyasını bulmak
- ii) Yığın Güncellemesi (Batch Update)

- Master file: Banka hesabı bilgisi (hesapno, isim vs.) { Her ikisi de hesapno'ya göre sıralanmıştır.
- Transaction file: Hesap güncellemleri (hesapno, kredi/borc, info vs.)

### 2-) Merging

- i) Alfabetik sırayı koruyarak iki sınıf listesini birleştirme.
- ii) Büyük dosyaları sıralamak. (Küçük parçalara ayır, parçaları sırala ve birleştir)

PROBLEM 4: Problem 2'yi sıralı (sorted) dosyalara göre çöz.

A ve B iki sıralı dosya, kayıtların  $\% 70$ 'i ortak. Her ikisinde 100.000 kayıt var ( $n=100\text{ k}$ ) her kayıt ( $R$ ) = 400 byte. Buffer pool 10MB  
ortak kayıtların bulunup yazılma süresi nedir? ( $S=16\text{ ms}$   
 $r=8.3\text{ ms}$ ,  $btt=0.84\text{ ms}$ ,  $B=2400\text{ byte}$ )

### Algoritması

A'dan M ardışık bloğu buffer pool'a al

B'den M ardışık bloğu buffer pool'a al

A ve B 'nin sonuna gelinmediği sürece:

Adım 3:

$r_A$  ve  $r_B$  kayıtlarını karşılaştır.

Eğer celesirlerse:

- Hər buffer pool'dakı banka bir C bloğuna yax.
- C doldugunda, C'yi diskə yax (yəni bozət)

Else-if r<sub>A</sub>'nin keyfi r<sub>B</sub>'nin keyinden kiçikse:

- A'nın sonrası kaydına git

Adım 3'e döñ

Else-if r<sub>A</sub>'nin keyfi r<sub>B</sub>'nin keyinden böyükse:

- B'nin sonrası kaydına git

Adım 3'e döñ

Else:

Eğer A'dan gelen tüm kayıtlar kontrol edildiyse:

- Sonraki M bloğu A'dan al

Adım 3'e döñ

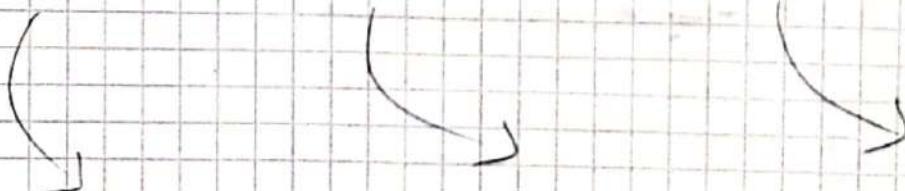
Eğer B'den gelen tüm kayıtlar kontrol edildiyse:

- Sonraki M bloğu B'den al

Adım 3'e döñ

**\* Bu algoritmda A'nın ve B'nin üzerinde yalnızca 1 defa  
geçilir. File file olunca f'dan okunan her M blok için  
B dosyasını baştan sona tekrar okumak gerekiyordu.**

Sorunun Sayısal Gözümlü



• Buffer pool'daki frame sayısı :

$$\left\lfloor \frac{10 \text{ MB}}{2400 \text{ byte}} \right\rfloor = \underline{\underline{4466}}$$

(10 milyon byte olmak üzere)

• Aşağı okumak için 2082 frame

• B'yi okumak için 2082 frame

• Ortakları yazmak için 2 frame kullanalım.

• 1 Sayfadaki Kayıt Sayısı : (Bfr)

$$\left\lfloor \frac{B}{R} \right\rfloor = \left\lfloor \frac{2400}{400} \right\rfloor = \underline{\underline{6}}$$

• 1 Dosyadaki sayfa sayısı :

$$\left\lceil \frac{n}{Bfr} \right\rceil = \left\lceil \frac{1000000}{6} \right\rceil = \underline{\underline{16667}}$$

① A için gerek süre :

$$\left\lceil \frac{16667}{2082} \right\rceil * (s+r+2082*btt) \rightarrow 2082 \text{ frame}$$

Her seferinde 2082 sayfa okunur (Bozalılıyor).

→ 2082 buffer pool'a kaydediliyor. Doluca diske yazılıyor. Sonra tekrar aynı işlem

② B için gerek süre :

$$\left\lceil \frac{16667}{2082} \right\rceil * (s+r+2082*btt) \rightarrow 2082 \text{ frame}$$

③ Ortak kayıtlar için gerek süre :

$$\left\lceil \frac{1000000 * 0.70}{6} \right\rceil * \frac{1}{2} * (s+r+2 * btt) \rightarrow \text{her disk 2 frame ayrıldıktı.}$$

İçerisinde 2082 frame varsa, 2'ye bölündü.

\* Bu tür zamanı toplayarak toplam zamanı buluyor.

## Eş Sıralı Tırnak Güncelleme Algoritması : (Algorithm for Co-req. Batch Update)

Örneğin; Banka sisteminde on binlerce müsteri olabilir. Bu müşteriler aynı anda işlem yapabilirler. Yapıları her işlemi hemen ardından veri tabanına doğrudan yansıtırsak bu işlem çok yavaş olacaktır. Çünkü bu güncelleme DB'ye yansıtılırken ilgili blok üzerine bir kilit konulması gereklidir. İlgili bloktaki bir başka müşterinin bilgileri varsa o kilit açılana kadar (okuma - yazma bitene kadar) beklemesi gerektir. Bu da çok uzun zaman alır bir işlemidir. Bu yüzden güncelleme işlemleri belirli bir sayıda güncelleneden sonra Toplu okruk yapılır. Bunun için 2 tane dosya var. Master file ve transaction file.

- Master file → Banka hesap bilgilerinin olduğu dosya.
- Transaction file → Güncellenenin bulunduğu dosya
- \* Her iki dosya da hesap numarasına göre sıralı olarak bulunur.

### Algoritma :

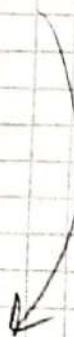
Pointerler her iki dosyanın da ilk kaydını gösterir.

Pointerler dosyanın sonuna gelene kadar:

Kayıtları keyfini kıyasla.

Uygun öntemleri al. \* (!Sonraki sayfaya bak açıklama için)

Pointerlerden birini (veya ikisini) ilerlet



## Önlemler \*

- Eğer master key < transaction key : (Master file'deki banka hesabıyla ilgili herhangi bir işlem yapılmıyor)
    - Master file'deki kaydı yeni master file'ıza aynı kopyala.
    - Bir sonraki kayda git (M. file iin)
  - Eğer master key > transaction key : (Master file'deki o kayıt tüm bir işlem yapılmıyor. (Yeni bir banka hesabı açılmış olabilir))
    - Eğer hesap eklene işlemiyse :
      - T. file'deki kaydı yeni M. file'nin sonuna kopyala.
      - Else :
        - HATA. (Lütfen olmayan bir banka hesabı üzerinde işlem yapın.)
        - Bir sonraki kayda git (T. file iin)
  - Eğer master key = transaction key : (Bu işlem güncelleme veya silme olabilir ama eklenme olmaz. Eklenme ise aynı hesap numarasına sahip farklı bir hesap açılması isteniyor demekti.)
    - Eğer güncellene yapılmışsa :
      - Yeni kayıt, yeni dosyaya yazılır.
  - Eğer silme işlemiyse :
    - HİBİTR ZEY yapma (Mevcut kayıt yeni dosyaya aktarılmayınca silinmiş oluyor)  - Eğer eklenme işlemiyse :
    - HATA.  - Bir sonraki kayda git (M. file ve T. file iin)
- ! Dosyaların sonuna gelindiğinde güncelleneler uygulanır.

## İndeksleme (Indexing)

- Basit bir index yapısını dizi (array) olarak düşünebiliriz.
- Dosyada bulunan her bir kayıt için kaydın anahtarı ve o kaydın başlangıç adresi tutulur.
- Index sayesinde dosyadan kayıtları daha hızlı bir şekilde buluruz. Çünkü genelde kayıtlar sıralı olmayan (pile) dosyalarda tutulur. Bu dosyanın içinde arama yapmak için ardışıl (Seq.) arama yapmak gereklidir. Index dosyasının büyüklüğün, data dosyasından çok daha küçük olduğunu indexler belleğe sigabilecek kadar küçük olur ve bellekte ikili arama yapılacağından çok daha hızlı bir arama olur.

## Index Kullanımı : (Uses of an index)

- Tek bir veri dosyası üzerinde n tane farklı index oluşturabilir.
- Örneğin ; öğrenci dosyasında index, isme, numaraya --- vs göre oluşturulabilir. Bu sayede hangi kriterde göre arama yapılıyorsa bu aramayı daha hızlı gerçekleştirebiliriz.
- Öğrenciler belirli bir sıralamaya göre dizilmemiş olabilir. Bu durumda indexleri kullanarak sıralama da yapabiliriz.

\* \* \* Büyük bir dosyada arama yaparken ; (en kötü durum olsun) aranan kayıt sonda ise → Bu dosya m parçaya bölündüp parçalar tek tek buffer pool'a aktarılacak ve gezi diske yazılacak.

Index olunca → Index dosyası, tek seferde memory'e alınacak ve orda arama yapılacak. Yani daha hızlı.

Hem veri hem de index dosyası diske saklanır.

Sırasız Dosya için Basit Index: (A simple index for a pile file)

Etiğet ID	İsim	Sənətçi
17 LON 2312	9. Sinfoni	Beethoven Giulini
62 RCA 2626	Romeo ve Juliet	Prokofiev Maazel
117 WAR 23699	Nebraska	---
152 ANG 3795	Keman Koncertosu	---

Adres

Key = Etiğet + ID

Index RAM'de sıralı olarak bulunur.

Kayıtlar, girildikleri sırayla dosyada görünür.

Anahtar	Referans (Adres)
ANG 3795	152
LON 2312	17
RCA 2626	62
WAR 23699	117

Verilen anahtara göre bilgi arama yapılır. Bulunan kaydın adresine gidilerek diskten okunur.  
\* (Alfabeye göre sıralandı)

Indexed Filelarda İşlemler: (Operations to maintain an indexed file)

- Veri dosyası ve boş bir indexin oluşturulması.
- Veri dosyasından indexin elde edilmesi.
- Indexin RAM'e alınması (kullanılmadan önce)
- Indexin diske giri yazılması (kullanıldıktan sonra)
- Kayıt ekleme, silme, güncelleme işlemlerinin indexe de yansıtılması
- Index güncellendikten sonra tekrar diske yazılması.

### Index Dosyasını Bellekten Diske Yazma: (Rewrite the index file from memory)

- Veri dosyası kapatıldığında, RAM'deki indexin diskteki index dosyasına geri yazılması gerektir.
- Henüz bellekteki index diskte yazılmadan bir hata meydana gelirse ne olur? (Elektrik kesintisi, makineyi kapatma vb.)  
Bu durumu engellemek için 2 önlem alınır.
  - Index dosyasının başında durum flagi bulundurmak.
  - Index dosyasının güncel olmadığı tespit edilirse, veri dosyasından tekrar index oluşturmak için bir prosedür çağrılır.

### Pile Dosyası ve Birincil Anahtar Indexi: (Pile File and Primary Key Index)

	Etket	ID	Başlık	Sözler
12	LON	2312	9. Sonfori	Beethoven, Giulini
62	RCA	2626	Pomeo ve Juliet	Protofieu, Maazel
117	WAR	23699	Nebraska	- - -
152	ANG	3795	Keman Konseri	- - -

Adres              Pile Dosyası

Anahtar	Referans	Anahtara göre sıralı
ANG3795	152	
LON2312	12	
RCA2626	62	
WAR23699	117	

Birincil Anahtar Indexi Dosyası

- İsteme işlemini hızlandırmak için index kullanılır.
- Yeni bir kayıt ekleneceği zaman, bu kayıt pile file'in sonuna ekleniyor. Yeni kaydın anahtar ve adresinin key index'e de eklenmesi gerektir. Bu ekleme sırasında alfabetik sıranın bozulması gerektir. Örneğin; 'B' ile başlayan bir anahtar bu indexte ikinci sıraya yerlesir. Diğer kayıtların birer satır aşağı kaydırılması gerektir. Eğer key index'in boyutu勩ukse ve RAM'de tutuluyorsa, bu kaydırma işlemi hızlı yapıılır. Bu işlemleri diskte yapmak oldukça (shift) maliyetlidir. Çünkü diskte kaydırma işlemi yapılamaz. Bunun yerine yeni dosya oluşturulup veriler oraya sıralı biçimde yazılır.

- Bir kayıt silineceği zaman, bu kayıt pile file'den silinir. (Bunun için 3 algoritma vardır: özel karakter kullanma, AVAIL LIST kullanma vs.). Bu işleminden sonra bu kaydın anahtar ve adres ikilisini key index'ten de silmemiz gerekiyor. Indexten silindiğten sonra da sırayı bozmamak için silinen elementden sonraki elementleri birer satır yukarı kaydirmak gerekiyor. Index, RAM'de bulunuyorsa bu işlem kolay. Diskte maliyetli. Bu yüzden bazı sistemler index'ten silemek adres kısmına silindi anlamına taşıyan bir işaret kayar. Örneğin; -1 → (silindi tekrar eklet.)
- Bir kayıt güncelleneceği zaman, pile file'de bu kayıt güncellenir. Güncellemeden sonra kaydın key'i aynı kalmırsa (sadece adresi değişiyorsa) key'in karşısına yeni adresi yazılır. Eğer key değişiyorsa (key = Etiket + ID olduğundan etiket değişiyor olabilir) yine key index'teki sırayı bozmadan index'e eklemek gerekiyor.

\* Birden fazla key indexi elde edebiliriz. Bu sayede istenilen parametreye göre daha hızlı bir arama yapabiliyoruz.

Birincil Anahtar = Etiket + ID

İkincil Anahtarlar = Başlık, Bestezi, Sanatçı ...

6 Bunun gibi ikincil anahtarlar elde edebiliriz...

## Pile File ve ikinci Anahtar Indexi : (Pile File and Secondary Key Index)

İkinci Anahtar	Birinci Anahtar
Beethoven	ANG3795
Beethoven	DG139201
Beethoven	DG18807
Beethoven	RCA2626
Córea	WAR23699
Dvorák	COL31809
Prokofiev	LON2312

→ secondary key index'te anahtarları tekrar edebilir. Çünkü, bu örnek için; Aynı tişinin bestelediği birden fazla eser olabilir.

→ sec. key'ler kendi içinde primary keye göre sıralanır.

\* sec. key'de adres tutulmaz onun yerine prim. key tutulur. Bunun sebebi,

- ikinci Anahtar Index Dosyası -  
örneğin, güncelleme işlemi sonrasında adres değişebilir. Bu durumda tüm sec. key indexlerdeki adresin değişimini görebilir. Bu da fazladan işleme sebep olur.

→ Bir kayıt ekleniğinde, pile file'a eklenir. Bu kaydın primary keyi ve adresi primary key index'e eklenmelidir. Daha sonra primary key ve sec. key'inin sec. key index'e sıralamaları bozmayacak şekilde eklenir.

→ Bir kayıt silineceğinde, silinen kaydın prim. key indexindeki adres değeri işaretlenir (örneğin -1 yapılır). Böyle bir durumda sec. key index'te herhangi bir değişiklik olmaz. Sec. key ile arama yaparken prim keye bakılır. Daha sonra prim key'in gösterdiği adres'e gidilir.

→ Bir kayıt güncelleneceğinde, güncellene türüne göre işlem yapılır:

↳ Kaydın adresi değişirse; prim. keyde adres kısmı değişir sec key aynıdır.

↳ Kaydın prim. key'i değişirse; prim key ve sec key düzeltlenip tekrar sıralanır.

↳ Kaydın sec key'i değişirse; sec. key index'te o kayıt silinip (bu örnekte besteci) yerine yeni kayıt yerleştirilir ve sıralanır.

### İkincil Anahtar Kombinasyonlarının Kullanarak Erişim: (Retrieval using combinations of secondary keys)

- Bir dosya üzerinde birden fazla sayıda ikincil anahtar indexi olabilir. (Baslik, bestekan ismi vb göre) (Aramayı kolaylastirmak için)
- Bu ikincil anahtarları "and" ya da "or" ile izlenece sokarak aranan esyleri bulabiliyoruz.

Örnek: Beethoven tarafından bestelenen ve basılıgı 9. Senfoni olan kayıtları bul.

Besteciyle eşleşen kayıtlar	Basılıkla eşleşen kayıtlar	Bu ikisinin 've' işlemi
ANG3795	ANG3795	ANG3795
DG139201	COL31809	DG18807
DG18807	DD18807	
RCA2626		

\* Bu primary keyler bize Beethoven tarafından bestelenmiş ve basılıgı 9. Senfoni olan tüm kayıtlardır.

Bu primary keyler ile primary key indexinden adresleri alır, diskteki ilgili adreslere giderken aradığımız kayıtları bulmaya çalışırız.

### İkincil index yapısını iyileştirme: Tersine Liste: (Improving 2ndary index structure: inverted list)

Adres	
0	Beethoven 3
1	Görea 2
2	Duroak 5
3	Prokofiev 7
Secondary Key Index File	
0	LON2312 -1
1	RCA2626 -1
2	NAR23699 -1
3	ANG3795 6
4	DG18807 1
5	COL31809 -1
6	DG139201 4
7	ANG36193 0
Etiyet+ID Lистри Dosya	

\* Sec. key index'in boyutunu kısaltmak amaçlanır. Böylece her daim RAM'e sağlanabilir (Hızlı erişim)

\* Örnek bir adres: Beethoven'in eserlerine erişim şu şekilde olur. Sec keydeki adres alınıp listede o adresle gidili. Burada prim key ve adres tutulur. -1 giren kadar adresleri takip ederek tüm eserlere ulaşılır.

\* Beethoven'ın yedi bir eser eklediğinde sec. key index değişmez. Sağdaki listenin en altına yedi kayıt olarak eklenir (8 adresine sahip). 1. adresin değer -1 yerine 8 yapılır. 8. kaydın adres kısmının -1 yazılır.

## CHAPTER 40:

### - Ağac Yapılı Indexler -

Basit indexli yapıtlarda bazı sorunlar yaşanır. Örneğin; basit index çok büyükse tamamı RAM'e sağlanır. Bu nedenle diske yazılması gereklidir. Diskten okumak da maliyetli bir işlemidir.

$\rightarrow N$	$\log(n+1)$	$\rightarrow$ Binary search
15 key	4	
1000	$\sim 10$	
100.000	$\sim 17$	
1.000.000	$\sim 20$	
tanrı key için		↳ kez diske erişmek gereklidir.

\* Disk üzerinde arama yapılırlar 3-4 erişimden fazla uzun sürebilir.

→ Bir diğer problem, dosyaya yeni veri ekleyip sildiğimizde index'i de uygun şekilde güncellendirmek gerekiyor. Yani çok fazla shift işlemi yapmamız gerekiyor. Eğer bu index diskte tutuluyorsa  $O(N)$  kader erişim yapmamız gerekiyor. (index kısmının sayfa sayısı)

Bu sebeplerden dolayı ağac yapılı indexler geliştirilmiştir.

### Indexli Ardışıl Dosyalar: (Indexed Sequential Files)

Ağac yapılı indexler, indexli ardışıl dosya yapısı olarak isimlendirilir.

Indexed: index kısmını kullanarak anahtar alana göre kayıtları hızlı bir şekilde arayabiliyoruz.

Sequential: Bu dosya bir ardışıl dosyadır. Dosyayı oluşturan sayfalar diskte ard-arda pozisyonlarda bulunur. Böyle olunca dosyayı baştan sona okumak daha hızlı oluyor.

### Örnek : Öğrencilik sistemi :

Öğrenci numarasına göre indexlenmiş bir dosyada öğrenciyi hızlı bir şekilde bulabiliyoruz. Öğrenci notlarında güncelleme olacağın zaman dünkü toplu biimde (batch update) yapabiliyoruz.

### Kredi Kartı sistemi :

Index kısmını kullanarak kredi kartı bilgisini veya banka hesaplarını hızla sorgulayabiliyoruz. Ardışık dosya yapısını kullanarak da hesaplar üzerindeki güncellemleri hızlı bir şekilde sequential processing algoritmasını uygulayarak gerçekleştirebiliriz.

\* Yani bu dosya yapısında bir index limiz var bir de veri dosyası ardışık biimde tutuluyor.

### Index Yapılarına Giriş : (Introduction to Index Structures)

Kitapta, indexlerle ilgili olarak  $k^*$  notasyonu geçiyor. Bu 3 anlamda olabilir ( $k$ , bir sayı veya bir string olabilir.  $k \rightarrow$  anahtar anlamında)

- Anahtar değeri  $k$  olan veri kaydı } prim key
- $k$  değeri ile anahtarı  $k$  olan rild değeri } sec key
- $k$  değeri ile anahtar değeri  $k$  olan kayıtların, kayıt id'lerinin listesi

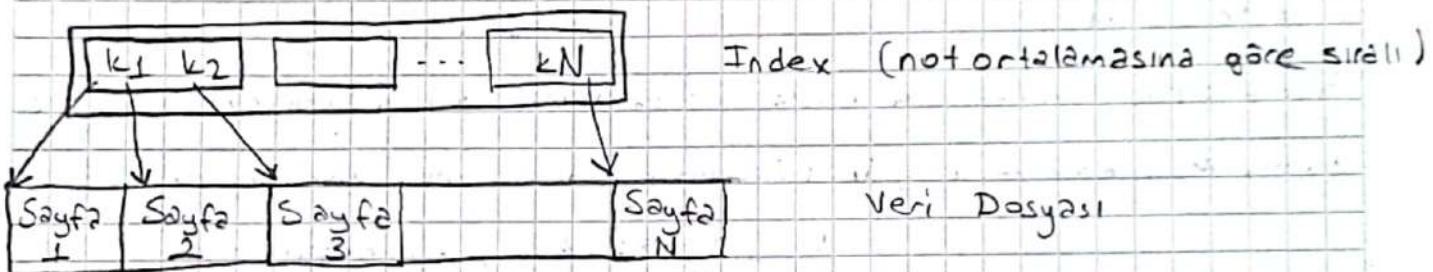
\* Ağacı yapılı dizinler range search ve equality search'ü destekler (mesai)

\* 2 tane ağacı yapılı dizin gereceğiz. ISAM (statik yapılı) BT tree (dinamik). Bu iki yapı da günümüzde DBMS'de sıkılıkla kullanılan veri yapılarıdır.

## Range Searches :

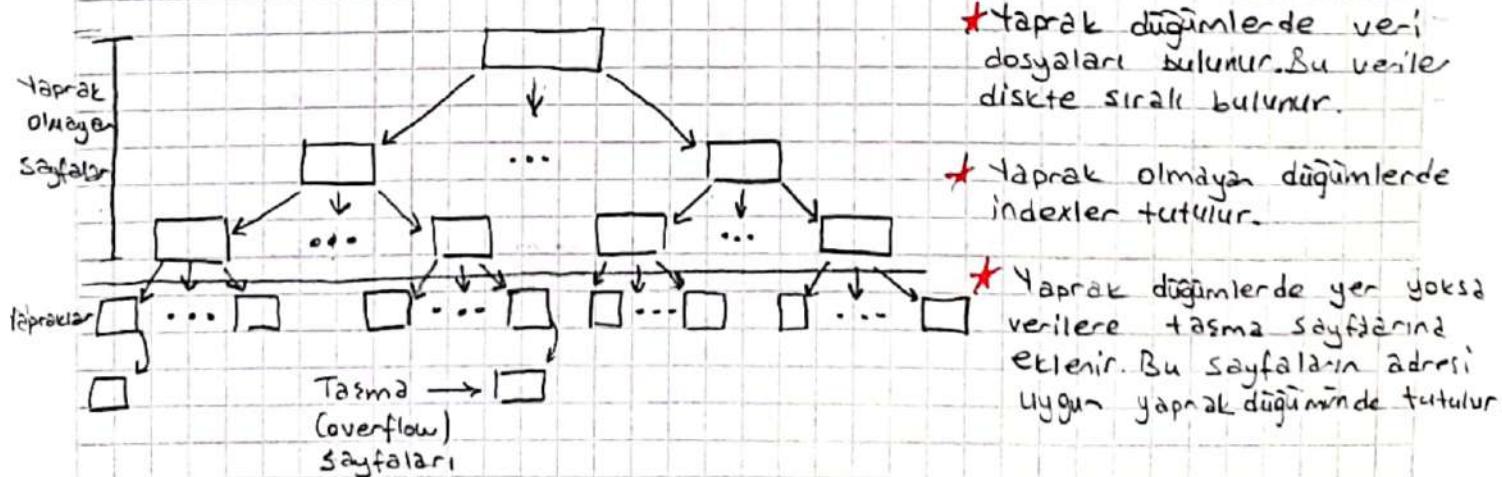
Indexli ardışılı dosya yapısında hem aralık sorgularını hem de eşitlik (range) sorgularını çok etkin ve hızlı bir şekilde yapabiliyoruz. Aralık sorgusuna bir örnek:

- not ortalaması  $> 3.0$  olan öğrencileri bul.



→ Index üzerinde, not ortalaması  $3.0$ 'ın üzerinde olan ilk öğrenci bulunur. O öğrencinin record id (rid) bilgisi alınır. Diskte o sayfaya gidilir. Dosya ardışılı olduğu için o öğrenciden, dosya sonuna kadar okuma yapılır. Böylece ortalaması  $3.0$  üzerinde olan her öğrenci bulunmuş olur.

## ISAM (Indexed Sequential Access Method) : (Dizinli Sıralı Erişim Yarıtı)



\* ISAM, tahta sayfaları hariç tamamen statiktir.

ISAM Olusturmak : (Comments on ISAM)

Öncelikle yaprak (data) sayfaları için ardışılı yer allocate edilir, (leaf)  
 Sonrasında arama anahtarına göre sıralanır yapraklar. Ardından index sayfaları (yaprak olmayan düğümler) allocate edilir. Yeri bir data ekleneneğinde yaprak düğümde yer yoksa overflow düğümleri allocate edilir.

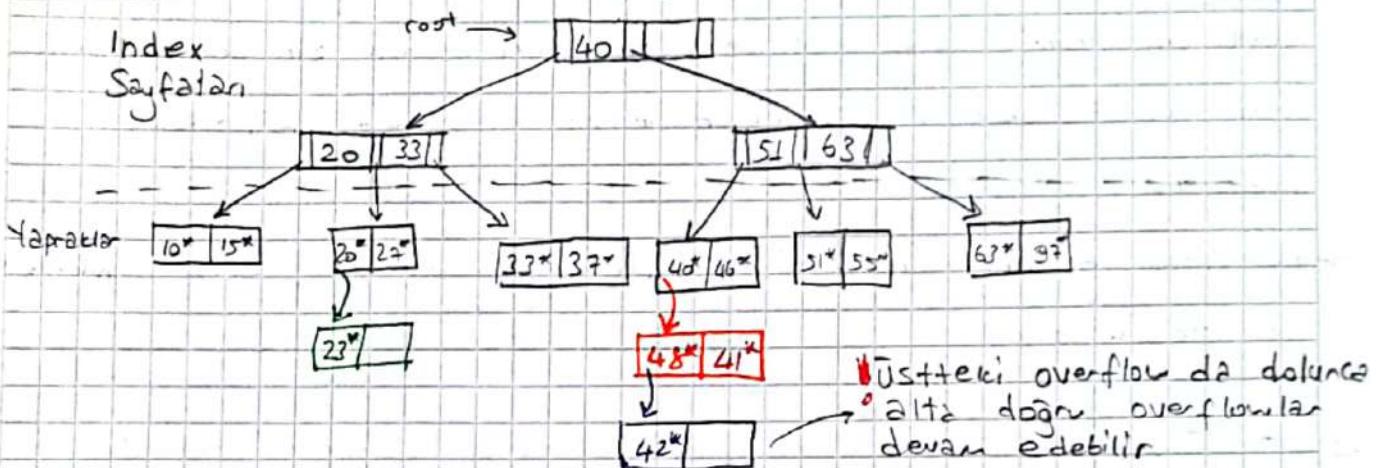
Arama : root düğümünden başlar anahtarları karşılaştırarak yönünt bulur. Maliyeti =  $\log_F N$  ( $F = \text{fanout}$ ,  $N = \text{yaprak düğüm sayısı}$ )  
 (ortalaması  
 cocuk sayısı / index sayfası sayısı)

Ekleme : eklenen yaprakta önceki arama yapılarak uygun yere geliniç. Yaprakta boş yer varsa direkt olarak yaprak düğümü veri eklenir. Aksi halde overflow sayfası allocate etmemiz ve bu sayfanın adresini yaprak düğümde tutmamız gereklidir.

Silme : Bir veriyi silerken; önce silinecek veri ağacta aranır ve silinir. Eğer veri overflowda bulunuyorsa ve overflow, veri silindiğinden sonra boş kalmırsa bu overflow düğümü deallocate edilir. Onun dışında overflowda silindikten sonra element kalacaktır ve veri yapraktaysa sadece silme işlemi olur.

\* Ekleme - çıkarma işlemleri index sayfalarını etkilemez. Yaprak ve overflowu etkiler.

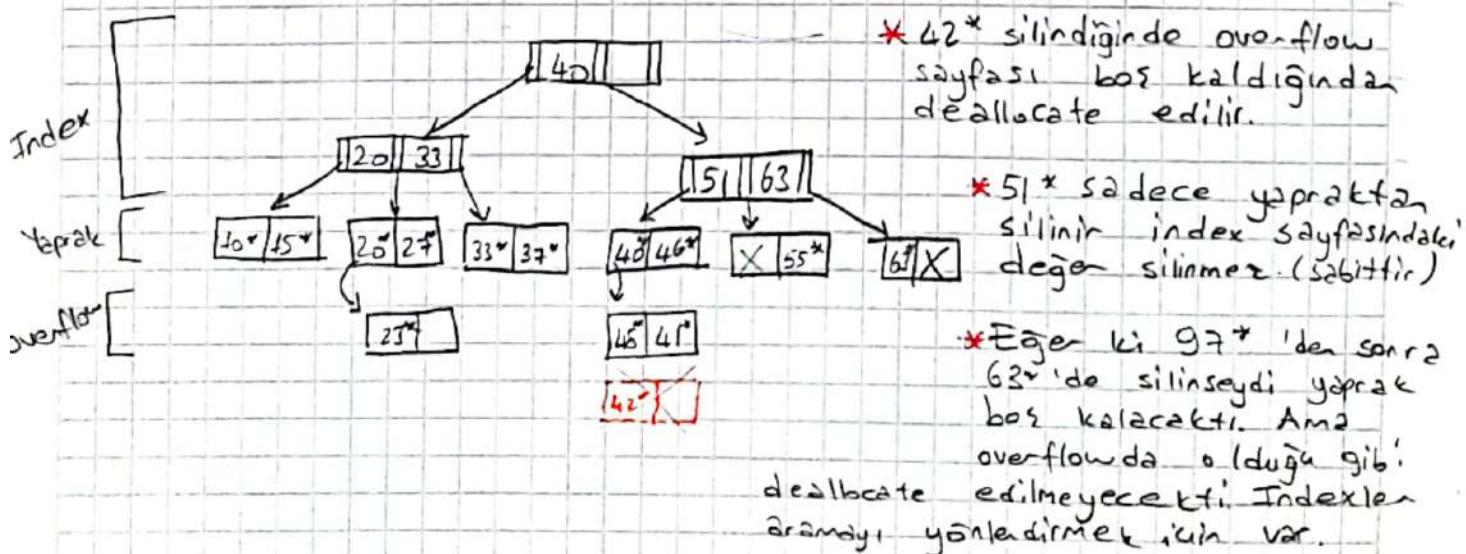
Örnek : ISAN'a  $23^*$ ,  $48^*$ ,  $41^*$ ,  $42^*$  ekle (sırayla)



$23^*$ 'in eklenmesi : Önce root'ta bakılır.  $23 < 40$  olduğundan sol tarafa gidilir. Daha sonra 20 ile karşılaştırılır.  $23 > 20$  olduğundan sağ taraftaki 33 ile karşılaştırılır.  $23 < 33$  olduğundan ortadaki kısma gidilir. Yapraklarda yer olmadığından overflow allocate edilir ve  $23^*$  değeri oraya yazılır. Bu overflowun adresi de  $20^*$  yaprakında tutulur.

\* Overflow  $\rightarrow$  random access, yapraklar  $\rightarrow$  sequential access.

$42^*$ ,  $51^*$  ve  $97^*$  değerlerini silmek :



## B+ Tree : (Most Widely Used Index)

ISAM'ın dinamik hale getirilmiş formudur. Dengeli bir ağactır (AVL gibi). Yeni kayıt eklenip silindiğinde index kısmında değişiklik olabilir.

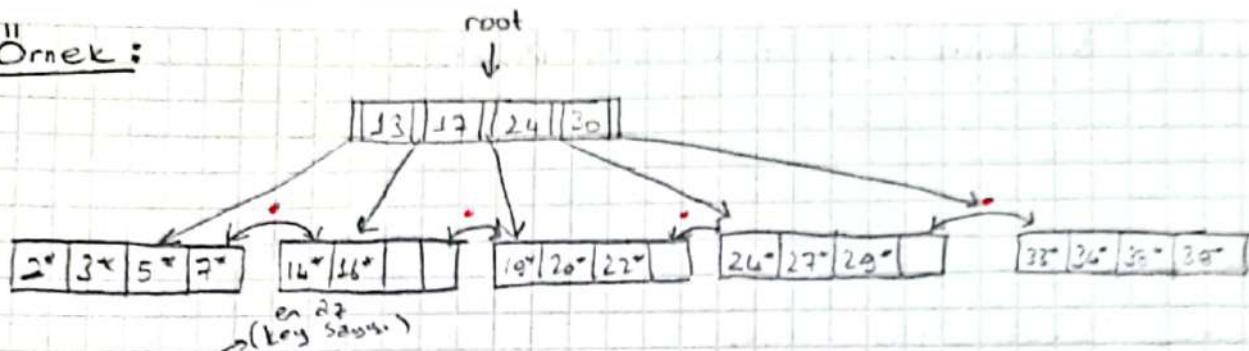
- \* Ağacın arama, ekleme, silme işlemlerinin maliyeti, ağacın yüksekliği ile doğru orantılıdır. Bunun maliyeti  $\log_F N$  ( $F = \text{fanout}$ ,  $N = \text{yazarken sayısı}$ ) (ortalamalı maliyet)
- \* B+ ağacı oluştururken, kök düğüm hariç, bütün düğümlerde en az ' $m$ ' tane eleman olmalı. Eğer ağacın derecesi (*order*) ' $d$ ' ise  $d \leq m \leq 2d$  olmalı. (en az %50'si dolu olmalı)
- \* Eşitlik ve aralık sorgularını hızlı bir şekilde yapan bir index yapısıdır. (Günümüzde en çok kullanılan index yapısı.)

## B+ Ağacının Özellikleri : (Formal definition of B+ Tree Properties)

Derecesi  $d$  olan bir B+ ağacı iain:

- Kök düğüm hariç her düğümde en az ' $d$ ' en fazla ' $2d$ ' tane anahtar olmalı
- Kök düğümünün en az  $2$  ilâ ucuğu olması gereklidir. (Kökk, yaprak değilse)
- Tüm yaprak düğümler aynı düzeyde bulunur. (Yükseklik açısından değil)
- Bir index düğümünde  $k$  tane anahtar varsa bu düğümün  $k+1$  ucuğu olmalı
- Tüm yaprak düğümleri, birbirine doubly linked list gibi bağlı olmalıdır. (Ardışılı durumunu bozmamak için)

Örnek:



- Bu örnekte order yani  $d=2$ . Çünkü her düğümde en fazla 4 anahtar var %50'si 2 yapıyor. Bu demektir ki her düğümde en az 2 anahtar olmalı.
- Kök düğümde 4 tane anahtar ( $k$ ) olduğu için 5 tane ( $k+1$ ) işaretçisi var.
- Kırmızı nokta ile gösterilen yerler sayesinde düğümler birbirine çift yönlü olarak bağlanır.
- 5\* değeri aranırken: root'taki ilk değere karşılaştırılır.  $5 < 13$  olduğunda 13'in solundaki adres alınır ve o düğümde sırayla arama yaparak 5\* değeri bulunur.
- 15\* değeri aranırken: root'a batılıp gerekli karşılastırımlar yapılır. 13-17 arasındaki adresse gidilip burada arama yapılır. Overflow olmadığı ve veriler düzeli olduğu için yalnızca bu düğüme batılarak bu eleman yoktur denebilir.
- $\geq 24*$  aralığını ararken: root'a batılıp gerekli karşılastırımlar yapılır. 24-30 arasındaki adresse gidilerek şartı sağlayan en küçük değerden başlanarak dosyanın sonuna kadar gidilir. Böylece şartı sağlan tüm değerlere ulaşılmış olur.

\* \* \*  
\* \* \*  
Yaprak düğümler, diskte ardışılı bulunmak zorunda değil. Bu ardışılılığı sağlamak için her sayfa, kendinden önceki ve sonraki sayfanın adresini tutuyor. (Şekilde kırmızı nokta ile gösterilen yer)

## Pratikte B+ Ağacı : (B+ Trees in Practice)

$d=100$  (En az 100, en fazla 200 kayıt var bir düğünde)  
ortalama

Doluluk oranı = %67

ortalama fanout = 133 olsun.  
(çocuk sayısı)

Bu durumda ;

4. yükseltlikte :  $133^4 = 312,900,900$  kayıt  
3. yükseltlikte :  $133^3 = 2,352,637$  kayıt bulunur

Seviye 1'de 1 sayfa tutulur ve boyutu 8 kB'dır

Seviye 2'de 133 sayfa tutulur ve boyutu 1 MB'dır

Seviye 3'de 17,689 sayfa tutulur ve boyutu 133 MB'dır

\* Görüldüğü gibi ilk 3 seviye RAM'de kolayca tutulabilir.

## B+ Ağacına Veri Ekleme :

B+ ağacına ekleme algoritması :

Doğru yaprak düğümü bul. (L)

Bu datayı L'ye koy.

Eğer L'de boşluk varsa :

Veriyi koy. ve işlemi bitir.

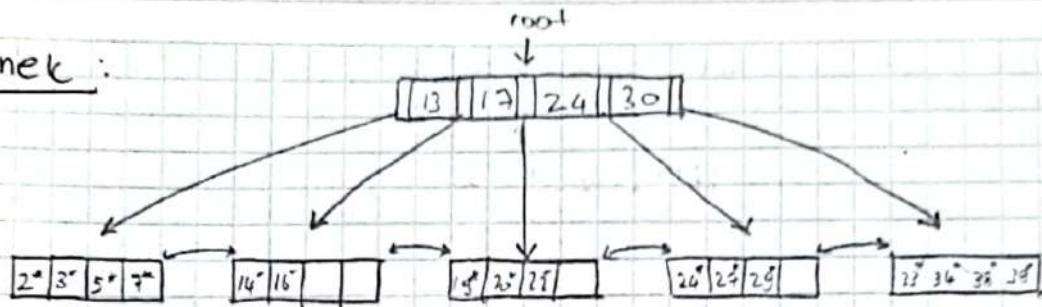
Else :

Yeni yer almak için düğümü ayır. (L ve L2 olar)

(Eşit yer en ve ortadaki keyi yukarı kopyala)

\* Ağacı bölmek (split), ağacı büyütür. Eğer root'u bölerseniz ağacın yükseliği artar.

Örnek :

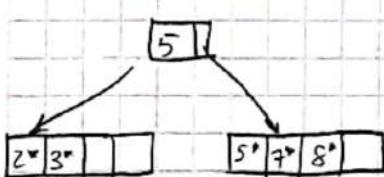


→ Bu ağaca  $8^*$  eklenmesi istenirse:

root'tan başladık.  $8 \leq 13$  olduğundan 13'un solundaki adresine gidilir ve düğüme bakılır. Bu sayfada boş yer olmadığı için 2'ye bölmemiz gerekiyor. (Yani yeni bir sayfa ekleyeceğiz)

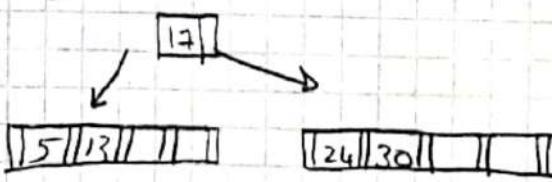
- İlk 'd' tane kayıt (bu örnekte  $d=2$ ) aynı sayfada kalır, diğerleri yeni sayfaya eklenir ve ortadaki değer bir üst seviyeye kopyalanacak. (Bunun yanında yeni sayfanın adresi de bir üst düğüme kopyalansın) Yani  $2^*$  ve  $3^*$  aynı yerinde kalacak,  $5^*$ ,  $7^*$  ve  $8^*$  yeni sayfaya gidecektir ve 5 ile yeni sayfanın adresi bir üst düğüme gönderilicek (ortadaki eleman)
- Yukarıda da yer olmadığı için (5'i eklemek için) bu düğüm de ikiye bölünür. Index düşümleri bölerten ilk d tane anahtar, aynı sayfada kalır, ortadaki değer (bu örnekte 17), bir üst seviyeye geçer, sonraki d tane anahtar yeni sayfaya gider. 5 ve 13 aynı sayfada, 17 bir yukarıya, 24 ve 30 yeni sayfaya giterek ve bunda göre de pointerler güncellenir.

I. adımda



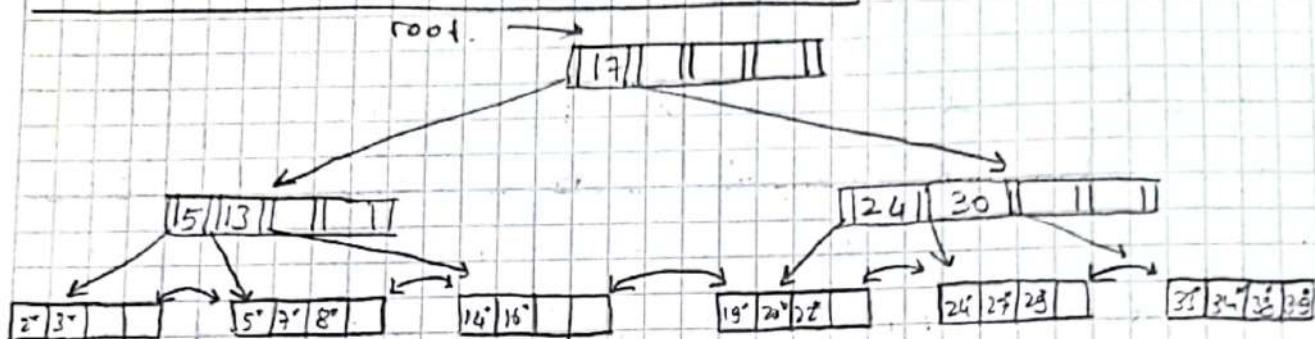
| Ortadaki değer indexinde  
taşındı

II. adımda



| Ortadaki değer bir üstte taşındı.  
root bölündüğünden yükseliş 1 arttı.

Bu işlemlerden sonra ağacın son hali :



B+ Ağacında Veri Silme :

Silinecek veriyi bul. ( L düğümünde olsun veri )

Veriyi sil.

Eğer silme işleminda sonra L, en az %50 dolu olmaya devam ediyorsa :

Silme işlemini bitir.

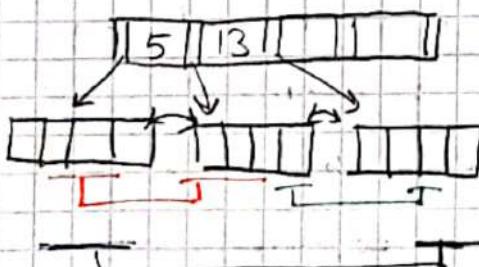
Eğer silmeden sonra d-1 veri kalmırsa düğümde :

Komşudan bir eleman al ve üsteki indexi güncelle.

Eğer komşuda d+1'dan daha az eleman varsa :

L düğümünü ve bunun kardeşini birleştir.

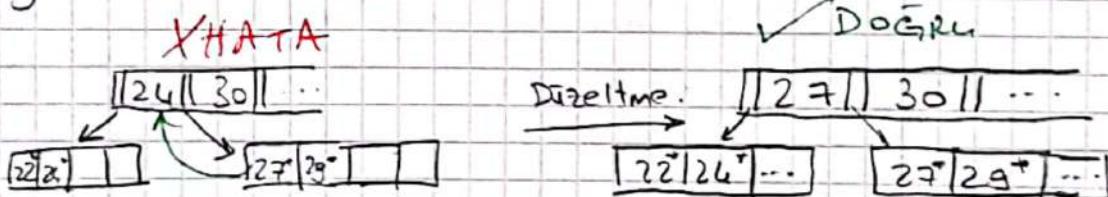
† Kardes düğüm derken su kastedilir.



! Kardes olmaları için aynı indexin sağına ve soluna yerlesmeleri gereklidir. Örneğin turuncu ve yeşil yerler kardeş fakat mavi kısımındaki düğümler kardeş değil.

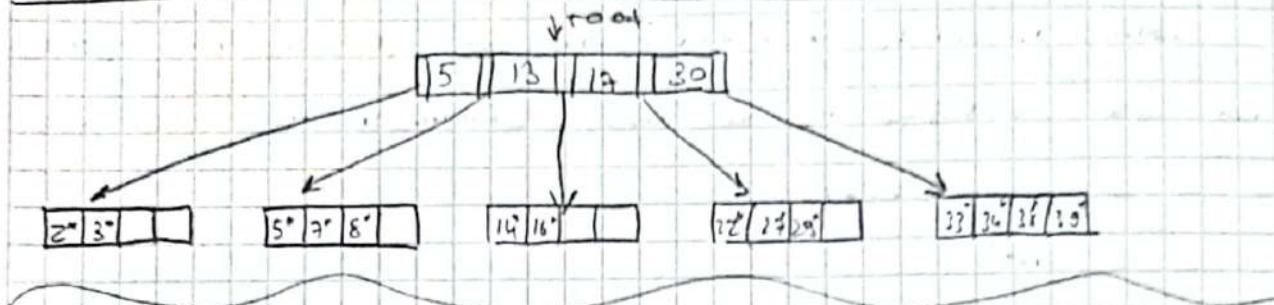
Örnek: Buradaki BT ağacı baz alınarak  $19^\circ$ ,  $20^\circ$  ve  $24^\circ$ 'dir.

- $19^\circ$  lu silmek için: Öncelikle  $19^\circ$  ağacı bulunur ve silinir. Silindiğten sonra eşiğin BT olma özelliğini koruyor mu diye bakılır. Silinen sayfadaki kayıt sayısı en az  $d$  (bu örnekte  $d=2$ ) tane olmak zorundadır.  $20^\circ$  ve  $22^\circ$  nin olduğu görülür ve  $19^\circ$  silindiğinde işlem biter.
- $20^\circ$  silindiğinde: Aynı sayfada 1 tane kayıt kalıyor. Bu durum BT ağacı olma özelliğini bozduğu için öncelikle kardeşinden 1 tane eleman ödünç alıyoruz ( $24^\circ$ 'ü). Kardeşten ödünç almak için kardeste en az 3 tane kayıt olması gerekiyor. Görüldüğü üzere bu şart sağlanır ve  $24^\circ$  değerini sayfamızda alırız. Bu işlemlerden sonra komşu sayfadaki en küçük kayıt anahtar olarak yukarı yollanır.

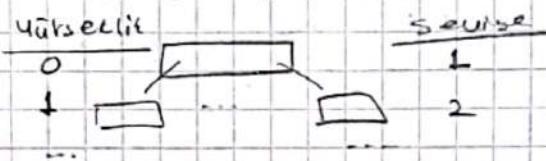


- $24^\circ$  silindiğinde: Silinen sayfada 1 kayıt kalıyor. Komsudan da alamıyoruz çünkü komsuda  $d+1$  (3) kadan kayıt yok. Bunun için silinen sayfa ve komşu sayfa birleştirilir. Sayfa birleştirilirken bir sayfa silinir. Bir sayfanın silinmesi demek bir üst düzeyde 1 index ve 1 adres değerinin de silinmesi anlamına gelir. 1 index silindiğinde üst düzeylerde yine BT ağacı özelliği bozulabilir. (En az 2 eleman olma) ve üstteki sayfların da aynı silme algoritmasında tabi tutulması gerekebilir. Önce komsudan iste, komsuda yoksa birleştir. Root'un çocukları birleştirse ve root ortadan kaldırırsa yükseliğ 1 azalır. Bu örnekte olduktan sonra.

Bu işlemlerden sonra ağacın son hali :



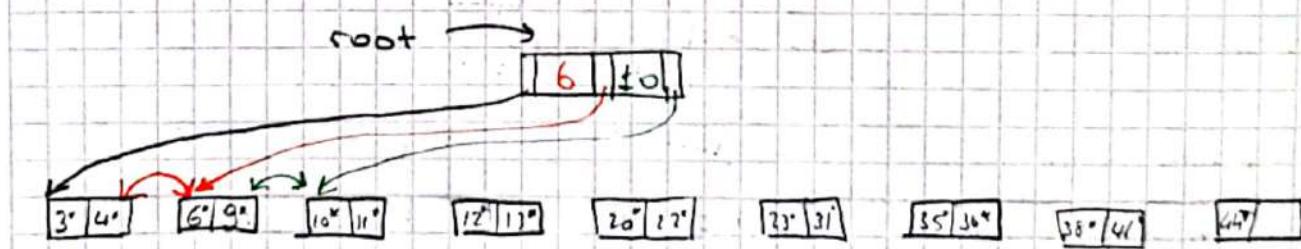
\* Bu ağacta arama maliyeti  $\log_F N$  dır. Bu da B+ ağacının (ortakta en iyi) arama maliyetini azaltır ve daha hızlı bir ortamda olur.



Toplu Yüklenme Algoritması : (Bulk Loading of a B+ Tree)

Bir dosya için, sıfırdan bir B+ ağacı yaparken; kayıtları tek tek birbir bir B+ ağacına eklemek şekilde yapabiliriz. Ama bu çok yavaş olsugundan bu algoritma kullanılır. Bu algoritma şöyledir:

Öncelikle eklemek istenen kayıtlar, anahtar değere göre sıralanır, ilk sayfanın adresini yeni oluşturulan kök düğümde saklıyoruz. ikinci sayfadaki en küçük değeri kök düğüme ekliyoruz ve 2. sayfanın adresini yukarı tazeledik. 3. sayfanın en küçük değeri kök düğüme eklenir ve 3. düğümün adresi de root'a eklenir. (2 değer tutan düğümler için genel)



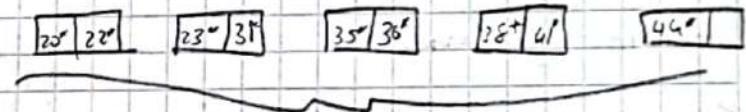
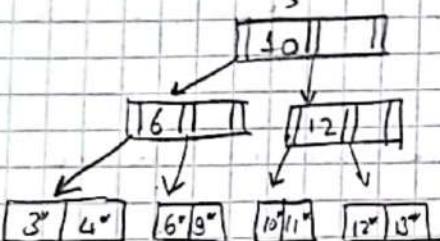
I. adım

II. adım

III. adım

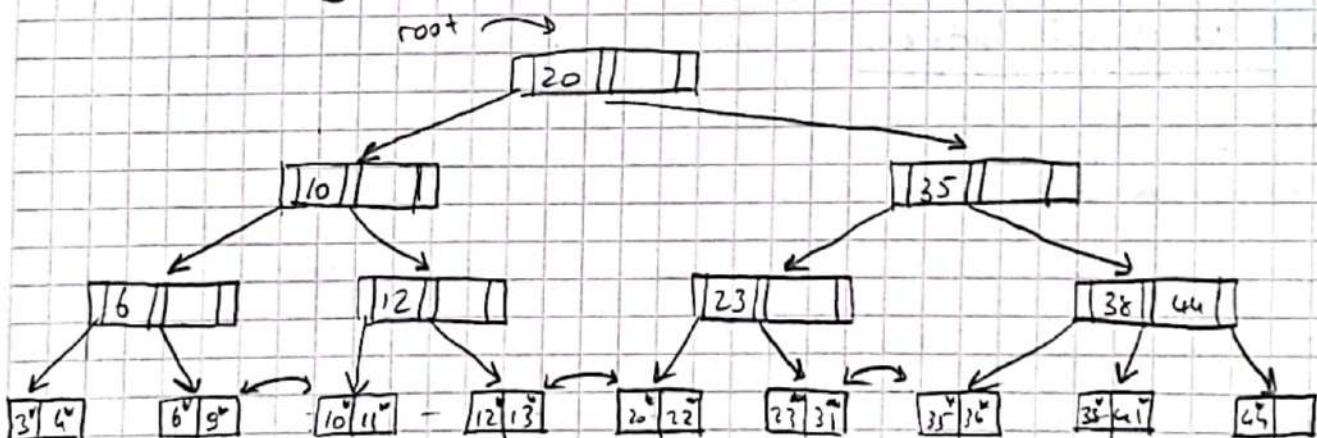
\* Artık root'ta yeterli yer yok. 12' ile başlayan düğümü eklemek için mevcut root'u belliyoruz. İlk d+ane key aynı yerde kalır. Ardakta;  $d=1$  bu örnekte (e) yarışçaya gider ve kalan d+ane yeni sayfaya gider.

root



Henüz etmemediler.

Tüm işlemler yapıldıktan sonra ağacın son hali



\* Fotokipedi açıklamayı da okuyabilisin

- Bulk loading algoritması eşzamanlılık kontrolüne sahiptir. (Concurrency control)
- Bir ağacı oluştururken az sayıda disk erizimi yapar.
- Yapraklar sıralı bulunur. (Aynı zamanda bağlıdır.)
- Doluluk oranını kontrol edebiliriz. (Örneğin %67 doluluk belirleyeliir.)

#### HAFLA IV. SON

→ Hw2. pdf'yi incele. (örnek soru ve çözümleri)

### Order (d) Kavramı : (A note on 'order')

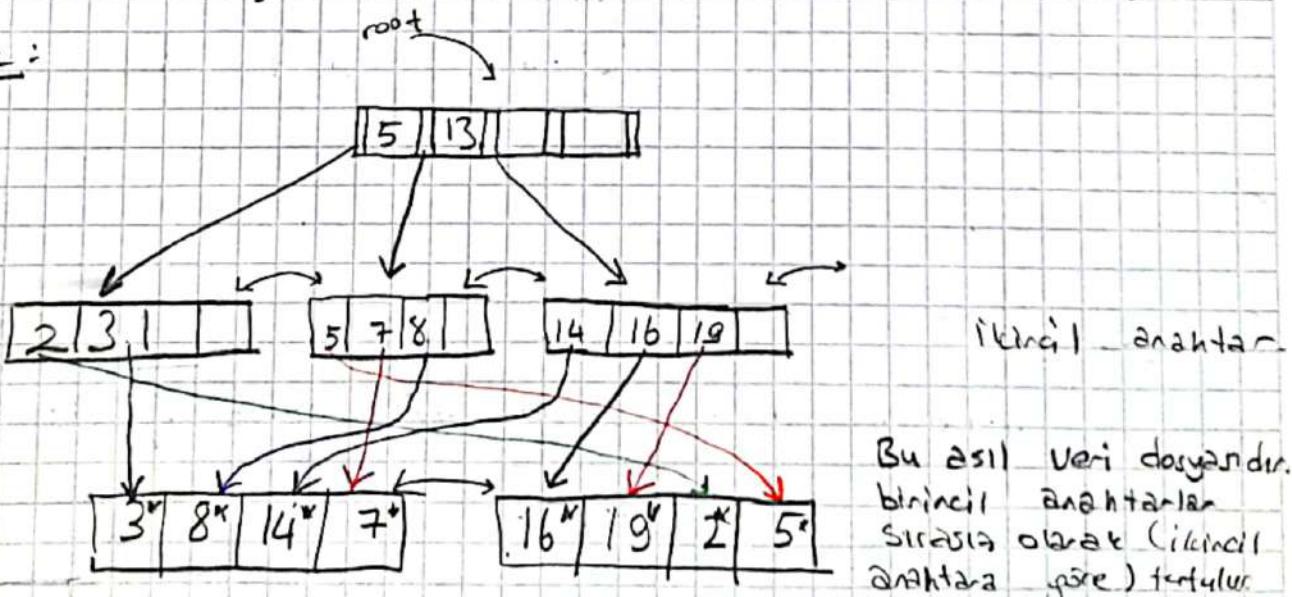
B+ ağacı ve ISAM'de order kavramı önemlidir. Çünkü, B+ ağacının bir düğümünde en az 'd' en fazla '2d' tane kayıt ya da anahtar bulunmalıdır. Yalnızca kök... düğüm bu kriteri uymayabilir. (d'den daha az sayıda anahtar bulunabilir) d sayısı, ağactaki düğümlerin doluluk oranının en az %50 olmasını garanti eder.

- \* Yapraklardaki kayıt büyüklüğü, index düğümlerindeki key'lerden büyük olduğu için yapraklarda daha az sayıda kayıt bulunabilir. (Indexlerde daha fazla anahtar bulunabilir)

### Birincil ve ikincil Index : (Primary vs Secondary Index)

- Eğer yaprak düğümlerdeki kayıtlar anahtar degere göre sıralıysa bu bir Prim. Key index'tir.
- Eğer yapraktaki kayıtlar (yani veri) anahtara göre sıralanmamışsa bu bir secon. key index'tir.

Örnek :



- \* Bu B+ ağacını kullanarak buradaki verilere sırayla erişmek mümkün. Ama kayıtlar diske sıralı bulunmadığından çok fazla disk erişimi yapılır. Bu işlem de maliyetlidir.

## CHAPTER 11:

### - HASH-BASED INDEXES -

Arama işlemini hızlandırmak için hash-based index kullanılır.

- Eğer bir ardışılı dosya varsa (index kullanılmayan) ve bu dosyada bir arama yapmak istediğimizde yapmanız gereken disk erişim sayısı :  $O(N)$  ile orantılıdır. ( $N$  = Dosyadaki sayfa sayısı)
- Eğer bir B<sub>t</sub> ağacı varsa :  $O(\log N)$  veriyi arama süresidir. Index sayfları RAM'de tutuluyorsa veri ağacına aranıp bulunduktan sonra diske tek bir erişimle veriye ulaşabiliyoruz.
- Eğer hashing yöntemi kullanılıyorsa :  $O(1)$  yalnızca 1 disk erişimi ile veriye ulaşır.

- \* \* \* Hashing yönteminde, kaydın anahtarı bit hash fonksiyonuna gönderilir ve bu fonksiyon bir adres değeri döndürür. Bu döndürülen adrese veri kaydedilir. Arama yapılacak zaman yine bu fonksiyon kullanılarak veri bulunur.
- \* Hash tabanlı indexler eşitlik sorguları için uygunlardır (B<sub>t</sub> ağacından daha iyi). Fakat aralık (range) sorgularını desteklemeyez.
- \* Hashing yönteminin de statik ve dinamik versiyonları var. (Tipki B<sub>t</sub> ve ISAM gibi)

### Hash-Based Index :

- Hash yapılı indexlerde veriler bucket denen soyut yapılar içinde tutuluyor.
- Bucket, bir ana veri sayfasından ve onun ardına eklenmiş '0' veya daha fazla sayıda overflow sayfalarından oluşur

- Bir  $k$  anahtar değeri verildiğinde, bu anahtar değer öncelikle bir hash fonksiyonuna gönderilir. Bu fonksiyon bize o kaydın ( $h$  ile gösterilir) yazılacağı bucket'ın adresini döndürür.  $h(k)$ ,  $k^*$  verisi buckete yazılır.
- Hash fonksiyonunun, anahtar değerleri bucket'lar üzerine düzgün dağıtması genetir. Aksi halde  $O(1)$ 'dan  $O(N)$ 'e doğru evrilir.

### Tasarım Faktörleri = (Design Factor)

Bucket Büyüklüğü : Aynı adrese tutunabilecek kayıt sayısı (size)

Doluluk Oranı : Veri doyasındaki kayıt sayısının bucketların toplam kapasitesine oranıdır. (Örneğin : B+ ağacında %67 doluluk olabiliyor.)  
Bu durumda dosya ihtiyacından daha büyük bir yere ihtiyaç duyuyor.  
Lading factor %50 ise ve 100 MB verimiz varsa 200 MB'lik bir disk alanına ihtiyaç duyulur.

\* Hash Fonksiyonu = Kendisine verilen anahtar değerini bir adres değerine döndüren bir fonksiyondur. Bu fonksiyon, verinin hangi bucket'ta olduğunu bilgisini verir.

Tesla Gözümlene Tekniği : Bir adrese gönderilen kayıt sayısı ordaki bucket'a sigamayacak şekildeyse overflow gerçekleşir. Uzun overflow zincirleri, oldukça performans kötüleşeceğinden birtakım algoritmalar kullanılarak bu sorun üstünlüğe caalisılır.

## Hash Fonksiyonları: (Hash Functions)

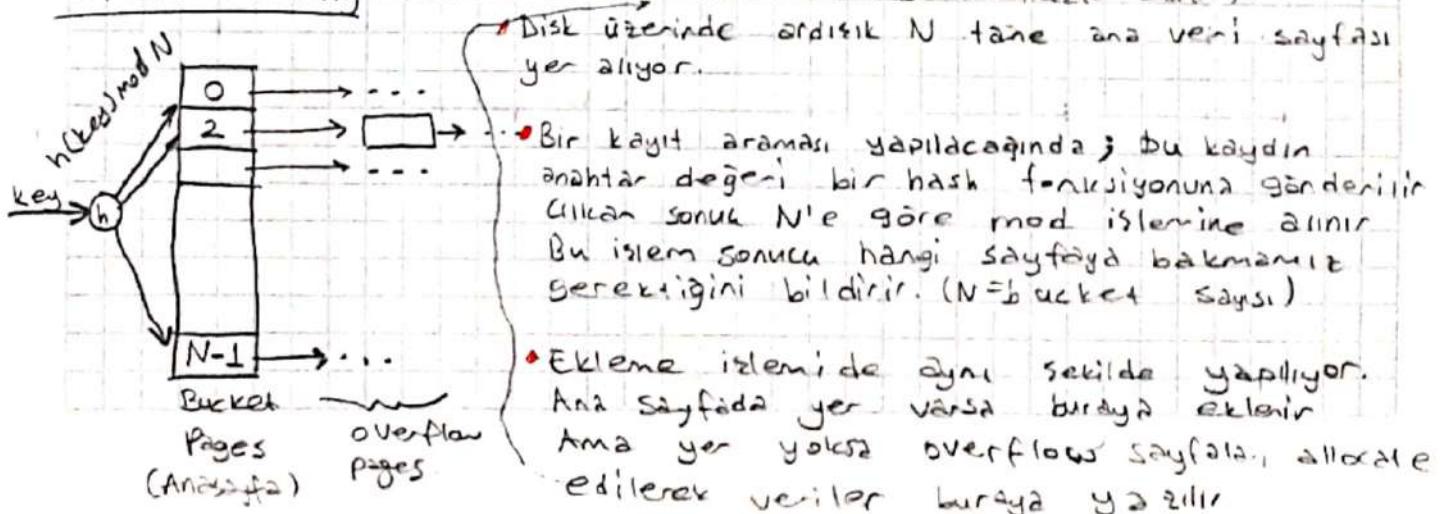
- Key mod N Yöntemi: En çok kullanılan yöntemdir.  $N = \text{bucket}$  sayısı.  $N$  sayısı asal sayı olursa daha iyi performans sağlanır.
  - Uzun integer'lardan oluşan anahtarlar için aşağıdaki yöntemler uygundur
- Folding Yöntemi: 123456789 bize anahtarımız olsun. Bu anahtar, 123 | 456 | 789 diye ayrılır ve her bir parça toplanır. Daha sonra bu toplamın modu alınır.

Truncation Yöntemi: 123456789 bize anahtarımız olsun. Bu (kesme) anahtar 3 basamaklı bir adrese dönüştürmek istenirse; içinden 3 tane basamak seçip birleştiriyoruz. (ilk 3, son 3, 1.3.5. basamak gibi yöntemler olabilir)

Squaring Yöntemi: Anahtarın karesi alınır ve daha sonra truncation yöntemi uygulanır.

Radix Conversion Yöntemi: Anahtar 11 tabanında bir sayı gibi (taban dönüştürümlü) ele alınır. Daha sonra bu sayı 10 tabanına çevrilip truncation yöntemi uygulanır.

Static Hashing: (ISAM'a benzeyen) (boz overflowlar deallocate edilebilir)



Örnek:  $N=5$  olsun,  $h(k)=k$  ve her bucket 3 kayıt tutuyor.

0			
1			
2			
3			
4			

Birincil  
Alan.

- Her satır bir bucket. Her bucket 3 kayıt tutabiliyor. Soldaki sayılar ise gerekeli adreslerdir.
- $h$  fonksiyonu kendisine gelen anahtar, aynı zamanda döndürüyor.

→ Sırayla  $12, 35, 44, 60, 6, 46, 57, 33, 62, 47$  anahtarlarının ekleme sırası.

12' için:  $h(12)=12$  olur.  $12 \% 5 = 2$  olduğundan tablonun 2. yazan kısmına eklenir.

Digerleri de eklenir!

17. için:  $h(17)=17$ ,  $17 \% 5 = 2$  fakat 2. adresi dolu bu yüzden bir overflow sayfasına ihtiyaç duyulur.

Son hali:

0	35	60	
1	6	46	
2	12	57	62
3	33		
4	44		

Birinci  
Alan

overflow

\* Veriler birinci alanında, arama yaparken tek disk erisimi ile o veri bulunabilir. Ama overflowda ise daha fazla disk erisimi yapılır.

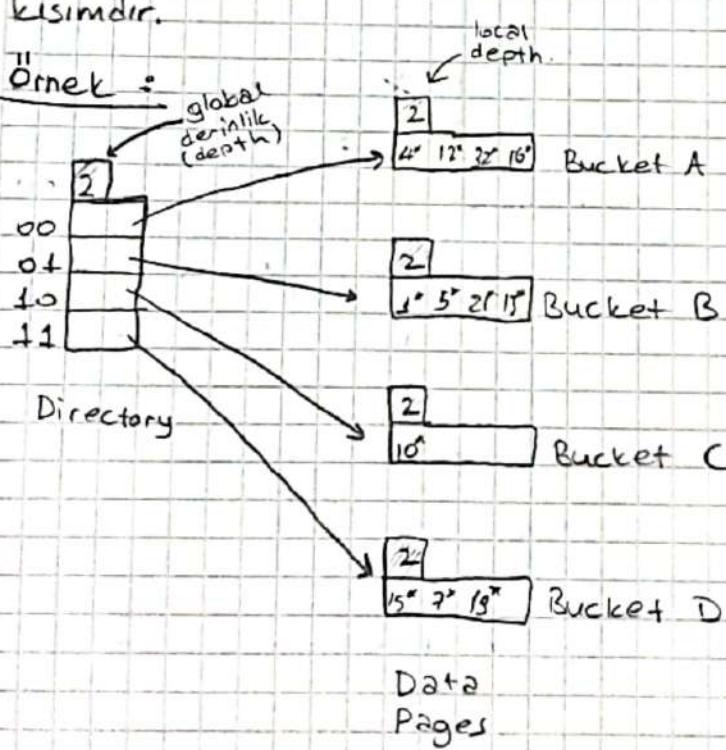
Örneğin 17'yi aramat için, 17 hash fonksiyonuna sokular ve mod  $N$  yapılır. Sonuç = 2. 2. adresi gidilir bir disk erisimi ile. Burada yoksa bir sonraki sayfanın (overflow sayfasının) adresi alınır ve bu sayfa için diske bir daha erişim yapılır.

## Extendible Hashing :

- Static hashing Yeri overflow sayfalarını ortadan kaldırır. amaçlayan dinamik bir yapıdır.
  - Dolu olan sayfaya bir kayıt eklenmek istendiğinde sayfa ikiye bölünür.
  - İki kısımdan oluşur. Dizin (directory) ve veri sayfları (data pages)
- Directory kısmı : Dizi olarak bellekte saklanır ve bu kısımda her bir bucketin adresi tutulur.

Data Pages kısmı : Bucketlerin olduğu ve kayıtların depolandığı kısımdır.

### Örnek :



★ Dizinde her bir bucketin adresi tutulduguundan ve 4 tane bucket olduguundan directoryde 4 tane adres tutmamız gerekiyor. Burun için global depthin 2 olması lazımlı. (2 bitte 4 farklı kombinasyon elde edildiği için)

★ Bir kayıt eklenecegi, aranacagi zaman, bu kaydin anahtarını hash fonksiyonuna gönderiyoruz. Çekim sonucu binary'e geliriyor. Sonrasında, bu binary sağının sondan global depth kadar bite batıyor(2. Örneğin; Yeni kayıt ekleset ve hash fonk'tan ilk 2 sonucun binary halinin sonu 00. Bu kayıt bucket A'ya eklenir)

★ Local depth, o kayittaki sayıların sondan kaç bite baktılarak buraya yerleştirildiğini bize söyler.

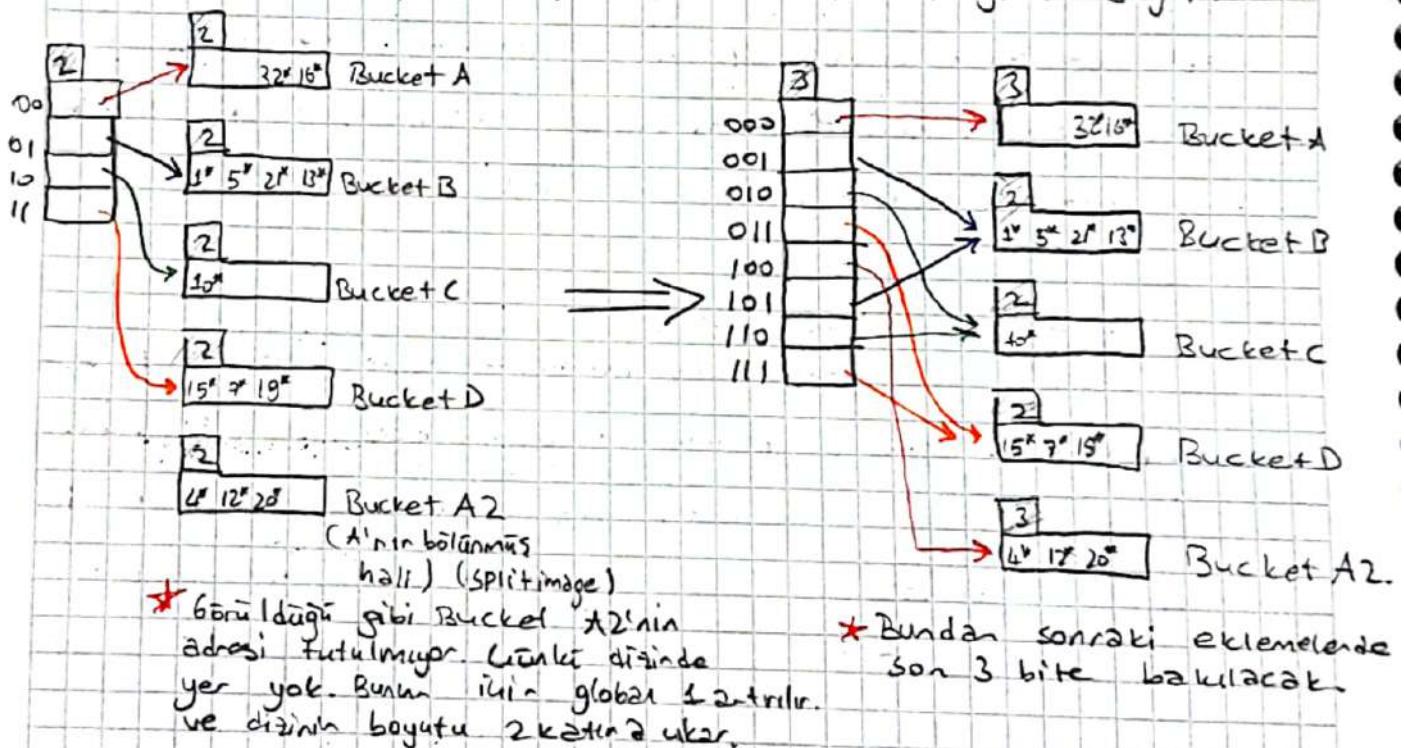
★ Global ve local depthleri kıyaslayarak bir sayfada ne zaman overflow olduguunu buluyuz. Overflow olan sayfa 2'ye bölünür ve local depthi 1'artır. (Yani son 2 bit yerine son 3 bite baktılarak yerlestirme gerekli)

{ overflow sonrası  
 local > global  
 oluyorsa  
 2'ye böl.

Örneğin 20 eklenmek istendiğinde :

$$h(t) = 20 = (\underline{10} \underline{100})_2$$

Son 2 bite bakılıncı dizindeki 00'ın gösterdiği yer olan Bucket A'ya eklemesi gereklidir. Fakat Bucket A dolu. Bu yüzden Bucket A'yı ikiye bölgüp sonu 000 olanları bir yere 100 olanları bir yere ayıriz. Daha sonrasında dizin yapısını da değiştiriniz. Çünkü overflow sonrası local  $\rightarrow$  global oluyor.



\* 6'lı olduğu gibi Bucket A2'nin adresi tutulmuyor (ünkisi dizinde yer yok). Bunun için global 1 satırılır. ve dizin boyutu 2 katına çıkar.

\* Bundan sonraki eklemelerde son 3 bite bakılacak.

Örneğin son 3 biti 101 olan bir kayıt eklenmek istendiğinde:

Dizinde 101'in gösterdiği Bucket B'ye girilir. Burada yer yok o zaman ikiye bölmeye geçerleşir. Sonu 001 olanlar ve sonu 101 olanlar ayrırlar. Dizinde adresleri tutmak için yeterli yer olmadığından global depth değişmez.

\* Arama yaparken, değer hash fonksiyonunu binary'e çevirilir ve dizinde aranır global depth'ın uzunluğuna göre. Dizide işaret edilen Bucket'la tek disk erişimi ile ulaşılıp aranın değer bulunur.

- ★ Hash fonksiyonu keyleri düzgün dağıtmazsa her seferinde global depth artarak directory boyutunu artırabilir. Bunun sonucunda directory, RAM'e sigamayacak kadar büyüyebilir ve directory diske yazılabilir (RAM'e sigmadığı için). Bu durumda diske 1 yerine 2 erişim yapmak gerektir. (1. erişim directory içi, 2. erişim veri içi)
- ★ Hash fonksiyonu düzgünse extendible hashing oldukça iyidir. Örneğin; 100 Mbitlik bir dosya olsun. Her bir kayıt 100 byte, 4K sayfa ve her sayfa 1:000.000 kayıt içersin. Bu 25.000 elemanlı bir directory'e karşılık gelir. Bu directory rahatlıkla RAM'e sığar.
- ★ Silme işlemi yapılırken, silmek istenilen verinin anahtarını hash fonk. sonuca binerle çeviriyoruz. Sonrasında ilgili kısma gidilip kayıt silinir. Eğer silme işlemi sonucu bucket boşalırsa, onu daha önceki hali ile (bir eksik global depth'teki hali) birleştiriyoruz. (Bt organ gibi). Birleştirme sonucu her bir bucket'ta 2 ismetci gösteriyorsa, global depth 1 arttırılarak dizin yapısının boyutu yarıya indrilir.
- Fakat实践中, yukarıda bahsedilen birleştirme ve dizin boyutunu yarıya indirme işlemleri yapılmaz. Çünkü iteride veri eklenirken bu işlemleri için vakti kaybı olacağından (yani tekrar kümülatif tedaris büyüt gibi) yapılmak istemez.
- ★ Hash based index'te aradık sorgusunun yapılamamasının nedeni, verinin sıralı olarak yerleştirilmemesidir.

## Linear Hashing :

Dinamik bir hash yapısıdır. Extendible hashinge alternatif olarak geliştirilmiştir. Overflow zincirlerini ortadan kaldırmayı amaçlar. Eski gibi bir directory kısmını yoktur. Bunun yerine  $h_1, h_2, \dots$  gibi uude sayıda hash fonksiyonu kullanır.

$$h_i(\text{key}) = h(\text{key}) \bmod (2^i N) \quad (N \rightarrow \text{başlangıçtaki bucket sayısı})$$

\* Bu yapıda Next adında bir pointer var. Bu pointer bir overflow olduğunda buluncak bucket'ı işaret eder.

Bu bölünme işlemleri sırayla her bucket'ıza uygulanır (round-robin algoritması ile). Bir round'un ortasında şu şekilde görünür.



- Split (bölünmüş) bucketlar.
- split image (böldündükten sonra 2. kısım)
- henüz split edilmemiş.

\* Arama yaparken, aradığım kayıt yeşil kısımdaysa direkt ulaşabiliriz. Ama kırmızı kısımdaysa bu sayfalar split edilmiştir. Bu yüzden + fazla bite batarak aranan kaydın adresinin kırmızısının yoksa mavidemi olduğunu belirlenir.

\* Yeni bir kayıt eklenirken, hane veya hane+1 kullanarak hangi bucket'a eklemek gerektiğini buluyoruz. Eğer veri ekleyeceğiniz bucket doluysa, o bucket'ıza bir overflow sayıda eklenir ve yeni kayıt overflow'ıza eklenir. Overflow'ıza

eleman eklenirse, Next'in gösterdiği bucket ikiye bölünür ve Next bir sonraki bucket'i gösterir. Bu sayede kullan overflow zincirleri olusmamız olur.

### Örnek:

- Level = 0      (hala gerek bakılacak)

-  $N=4$  (Bucket sayısı)

- Her bucket 4 kayıtlıdır.  $\rightarrow$  (Tessadüf olursa bir bitlerin 4. bitiniyle bağlantıları gider)

Level = 0

$h_1$	$h_0$	Next = 0	43 eklenmek istenirse (101011)	$h_1$	$h_0$	Birinci sayfa	Overflow
000	00		32°   44°   36°	000	00	32°	
001	01		9°   25°   5°	001	01	9°   25°   5°	
010	10		14°   18°   10°   30°	010	10	14°   18°   10°   30°	
011	11		31°   35°   7°   11°	011	11	31°   35°   7°   11°	43°
				100	00	44°   36°	

! \* overflow olmadığı sürece, eklenmede herhangi bir bölme işlemi olmaz.

\* Sonu 11 ile biten sayfada boş yer yoktu. Biz de 43°'ü overflow sayfasına ekledik ve Next'in gösterdiği bucket'i bıldırdık. Sonrasında Next sonraki sayfayı gösterdi.

\* Örneğin sonu '11' ile biten başka bir kayıt eklemek istenirse 43°'ün yanında yazılır ve bu overflow olduğu için Next'in gösterdiği bucket yine bölünür ve Next bir ileride gider.

\*  $h_0$  iin Next = 3 olduktan sonra bir daha overflow gelirse,  $h_1$ 'e geçilir ve Next = 0 olup, başa döner. Bucket sayısı = 8 olur. (level 2)

\*  $h_1$  iin Next = 7 olduktan sonra overflow olursa  $h_2$ 'ye geçilir Next = 0 olur. Bucket sayısı = 16 olur (level 3)

## Linear Hashing (LH) ve Extendible Hashing (EH):

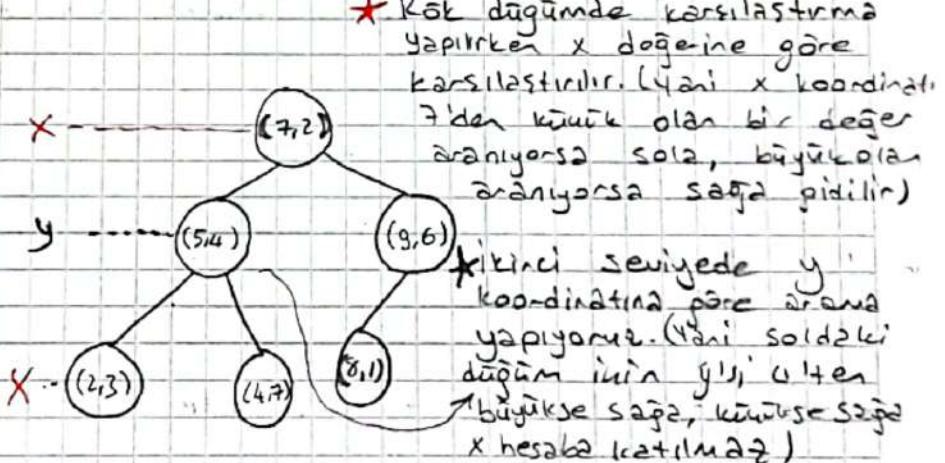
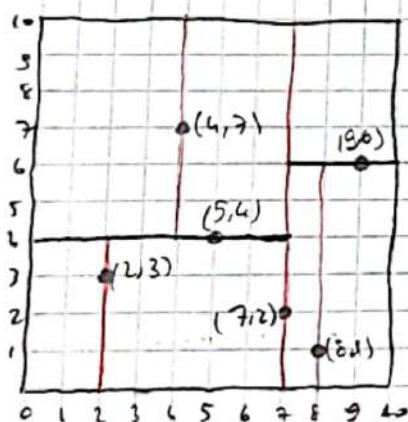
- \* LH aslında EH'nin bir varyasyonudur.
- \* EH'de, bir dizin yapısı vardır. Bu yapı bucket'ların adresini tutuyor. Bir kayıt eklenirken ilgili bucket'ta overflow oluyorsa ve dizinde yeteri kadar yer yoksa, dizinin boyutunu iki katına çıkarıp overflow olan bucket'i, ikiye bölayoruz.
- \* LH'de dizin yapısı yoktur. Onun yerine primary data sayfaları diskte art arda yerleştirilir. Next adında bir pointer var. Bu pointer, hangi bucket'i, ikiye böleceğimizi işaret eder. Ekleme yaparken overflow olursa, Next'in gösterdiği sayfa ikiye bölünür ve bütün ana sayfalar ikiye bölündüğün zaman asılonda directory li bir katına uitkamış gibi oluyoruz.

HAFTA II. SON

## - K-d Trees -

- ★ K-d ağacı ( $k$  boyutlu ağac), primary ve secondary keylerin birlikte kullanıldığı bir yapıdır.
- ★ Aralık (range) soruları için kullanılır. (örnegin; 4000 TL'den fazla maaş alan ve en az iki ucuğu olan işçileri bul)
- ★ Bundan önce aramayı tek bir niteliğe göre yapıyorduk. Fakat  $k$ -boyutlu ağacta  $k$  tane niteliğe göre arama yapılabilir.
- ★ ikili arama ağacının genişletilmiş bir versiyonu diyebiliriz. (Burada da aradığımız değer küçükse veya eşitse sola, büyükse sağa gideceğiz.)
- ★  $k$ -boyutlu ağacta, her düğümde tek bir değer yerine bir vektör bulunur. ( $k =$  boyutunda)

Örnek :



- ★ Kırmızı çizgiler  $x$  boyutundaki, mavi çizgiler  $y$  boyutundaki bölgeleri gösterir. İlk bölme  $(7,2)$  den olmuştur. Onun için kırka olsun.

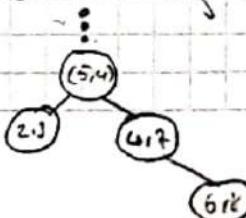
→ Ağaca  $(6,8)$  eklemek istenirse:

Seviye 1:  $y$ 'e göre karşılaştır. (Sola git)

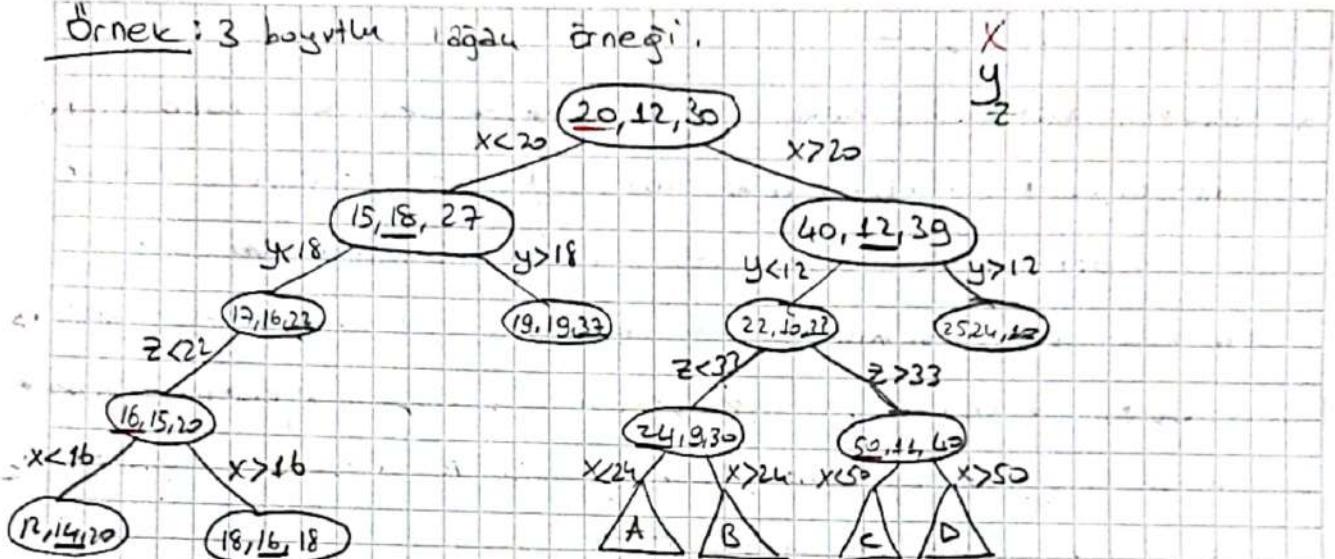
Seviye 2:  $y$ 'ye göre karşılaştır. (Sağa git)

Seviye 3:  $x$ 'e göre karşılaştır. (Sağa git)

Seviye 4: Buranın sağına yerles.



Örnek: 3 boyutlu lagau örneği.



A, B, C, D düğümlerinin  $x, y, z$  değerleri için ne söylerebilir?

- A:  $20 < x < 24$ ,  $y < 12$ ,  $z < 33$

B:  $x > 24$ ,  $y < 12$ ,  $z < 33$   
(hem  $z$  hem  $z$  ite büyüğebilir, sayı kesinlikle  $z$  ite büyüğebilir.)

C:  $20 < x < 50$ ,  $y < 12$ ,  $z > 33$

D:  $x > 50$ ,  $y < 12$ ,  $z > 33$

\* Her düğüm genisine hangi nitelikle göre değerlendiriliyorsa ondan büyük veya küçük olduğunu belli teek yaz. (örn.  $x < 20 \rightarrow x > 20$ )

\* A, B, C ve D'nin  $x, y, z$  'sinin bellilemek için önceki dallanmalara baktır. Örneğin A'nın bulunduğu konum en üstte  $x > 20$  şartıyla ayrılmış ve A'dan bir önceki adında  $x < 24$  denenek tekrar ayrılmış. Bu iki değer dikkate alınarak A'nın x 'degeri için bir aralık belirlenir.

### Ağacı Oluşturma : (Construction)

- \* K boyutlu ağacı oluşturmak için medyan yöntemi kullanılabiliyor.
- \* Bu yöntemde; örneğin  $3$  boyutlu ( $xyz$ ) bir ağacınız olsun.  
Tüm  $x$  değerleri sıralanır ve medyanı (ortalama değeri) bulunarak küt düğüme yerleştirilir. Daha sonra onun çocuklarını için kalan düğümlerin değerlerini kendi aralarında sıralayıp, onların da medyanını bulup ekleyebiliriz. Bu şekilde yaparak dengeli bir ağacı oluşturabiliyoruz.
- \* Ağacın dengeli olması arama, ekleme, silme... gibi işlemlerin süresini en aza indirecektir.
- \* Fakat bu dengeli ağacı oluştururken her bir adımda medyan bulunacağından, sıralama yapmak gerekiyor. Sıralama yapmak maliyetli bir işlemidir. Bu işlem sonucu  $O(n \cdot \log n)$  zamanda ağacı oluşturabiliyoruz.
- \* Bu yöntemin yanı sıra mean (ortalama) kullanılarak da bu ağacı oluşturabiliyoruz. Aradaki fark şudur; medyan yöntemi kullanırsa düğümlerde veri noktalarının kendisi bulunur. mean yöntemi kullanılsa düğümlerde veri noktaları yerine ortalama değerler bulunur.
- \* Medyan bulmak ortalamayı bulmaktan daha maliyetlidir. Medyan için  $n \cdot \log n$  'lik bir zaman gereklirken (sıralama için) ortalama bulurken  $n^2$ 'lik bir zaman yeterli olacaktır.

## Ağaca Ekleme : (Insertion)

- \* Kök düğümden başlayarak eklemek istenen düğümleri boyutlarının değerleri karşılaştırılarak sağa veya sola gidilir. Yaprak düğüme ulaşınca buranın sağına veya soluna eklenir.
- \* Yukarıdaki gibi ekleme yapmak ağacın dengezi olmasına sebep olabilir. Yani ağacın yükseliği artar. ( $\log n$ ; yükseliğe)
- \* Bu sebepten dolayı AVL Tree gibi dengeli olmalı ağacımız. Fakat k-d tree'de AVL'deki gibi dandürme işlerini kolaylıkla yapılımaz. Çünkü birde fazla boyut kullanılıp yerlestirme yapılmıyor. Bu dengeyi sağlamak için çeşitli algoritmalar oluşturulmuştur. (pseudo k-d tree, K-D-D tree, h3 tree ...)
- \* Eğer en baştan medyanları bulunarak ekleme yapılırsa baştan dengeli bir ağac olusur.

Örnek: Pamuk Prences ve 7 Cüceler masalındaki karakterlerin isim, boy, kilo ve diğer bilgileri verilmiş olsun. Boy ve kilo verilerini kullanarak bir K boyutlu ağacı üretelim.

isim	Boy	Kilo	Diger veriler
Sleepy	36	48	...
Happy	34	52	- - -
Doc	38	54	...
Dopey	37	54	...
Grumpy	32	55	- - -
Sneezy	35	46	- - -
Bashful	33	50	- - -
s.white	62	98	- - -

\* Bu ağacı oluştururken karakterleri sırayla ekleyeceğiz. (medyan kullanmayız)

\* 1. seviyede boy ikinci seviyede kilo  
3. seviyede boy... diye devam ederek (Hangi değere gəreyse altına tırpi koymağız)

~~A~~ Altı çizili kisim şunu ifade eder: eğer bir düğüm bu düğüm ile karşılaştırılıyorsa, bu karşılaştırma boyut değerleri üzerinde yapılır.

77

I-) Sleepy (36,48)

II-) Sleepy (36,48)

~~h < 36~~

Happy (34,52)

III-) Sleepy (36,48)

~~h < 36~~

~~h > 36~~

Happy (34,52)

Doc (38,54)

IV-) Sleepy (36,48)

~~h < 36~~

~~h > 36~~

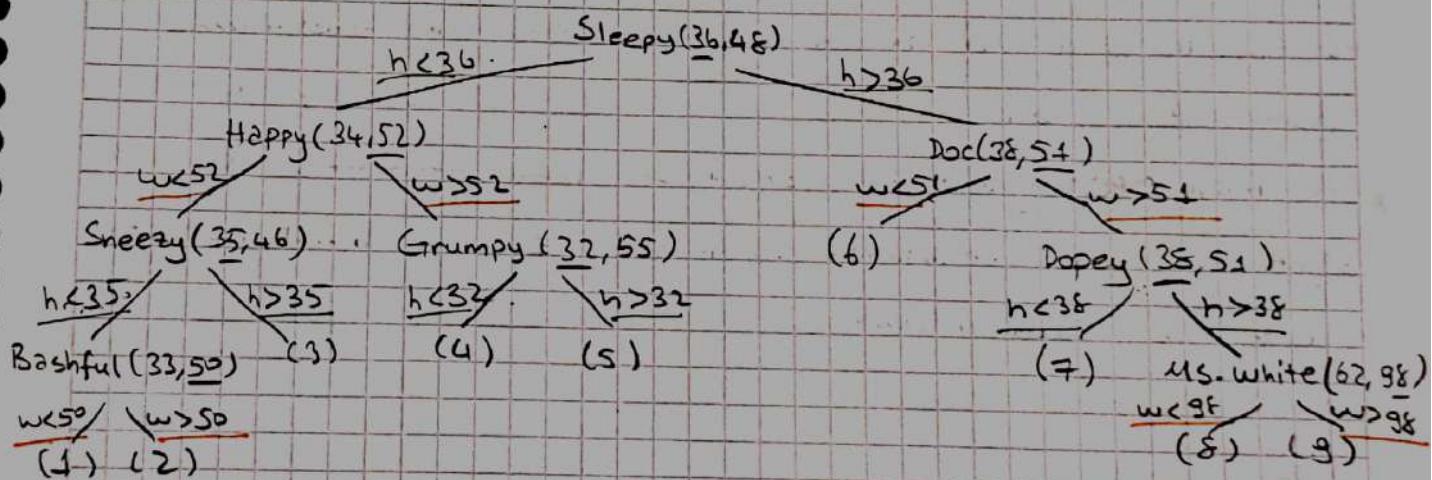
Happy (34,52)

Doc (38,54)

~~h > 36~~

Dopey (37,54)

→ Bu işlemler sırdanlı olarak en son son ağaca ulaşır:



1:  $h \leq 35, w \leq 50$  | 2:  $h \leq 35, 50 \leq w \leq 52$  | 3:  $35 \leq h \leq 36, w \leq 52$

4:  $h \leq 32, w > 52$  | 5:  $32 \leq h \leq 36, w > 52$  | 6:  $h > 36, w \leq 52$

7:  $36 \leq h \leq 38, w > 51$  | 8:  $h > 38, 51 \leq w \leq 98$  | 9:  $h > 38, w > 98$

Karmaşıklık: (Complexity)

• medyana göre ağac oluşturuyorsa:  $O(n \log n)$  zaman alır.

• Dengeli ağaca düğüm eklenirken:  $O(\log n)$  zaman gecers.

• Dengeli ağactan düğüm silinirken:  $O(\log n)$  zaman gecers.

• Query (sorgu) yaparken (aralık sorusu i.e.):  $O(n^{1-\frac{1}{2k}} + m)$

( $m$ : istenilen düğümlerin sayısı,  $k$ : boyut sayısı) ~~\* Örn. boyu 36'da büyük olanlar için~~

$m=3, k=2$  dir

(3 elemanlı)

## K Boyutlu Ağacının Kullanıldığı Alanları (Applications)

- Sensör ağlarında soru işlenme (query processing in sensor networks)
- En yakın komşu aramaları (nearest-neighbor searches)
- Optimizasyon (optimization)
- Izin izleme (ray tracing) → Bilgisayar grafiği metodu
- Bir den fazla anahtar ile arama (database search by multiple keys)

### — Grid Files —

\* k-boyutlu ağacı gibi birden fazla anahtara göre sorgulama yapmanızı sağlar.

\* Örneğin ; DEPT = "Toy" AND SAL > 50K  
 (Oyuncak departmanı ve maaşı 50 binden fazla )  
 böyle bir soruyu cevaplamak için grid file yapısı kullanılır.

→ Yukarıdaki örnekte verilen kayıtları bulmak için en yöntemler izlenebilir.

#### Yöntem 1 :

Sadece departman alanına göre indeximiz olsaydı. Departmanı 'oyuncak' olan kayıtları bulup, tek tek maaşlarına bakardık.

#### Yöntem 2 :

Departman ve maaş alanlarına göre ayrı ayrı index tutabilirdik.  
 Mesela departmana göre bir hash (esitlik sorusu olduğu için), maaş sorusu için  $B+$  ağacı (aralık sorusu) olabildi.  
 Böyle bir durumda ; departman kısmından 'oyuncak' getiriliyor,  
 maaş kısmından  $\geq 50K$  getiriliip kesişimlerini alabildik.

### Töntem 3:

Departman ve maaş üzerinde tanımlı bir grid file index yapısı kullanabiliriz.

Örnek : Hangi index yapısı kullanılabilir?

- 1-) Departman = 'Satış' , Maaş = 20 k (Departman → hash , Maaş → hash)
- 2-) Departman = 'Satış' , Maaş > 20k (Departman → hash , Maaş → B+)
- 3-) Departman = 'Satış' (hash)
- 4-) Maaş = 20k (hash)

→ Bu dört soru için de grid file yapısı uygundur çünkü aynı anda birden fazla alana göre hem eşitlik hem aralık sorgulaması yapmak mümkün. Bunun yanında tek bir alana göre de soru yapabiliyoruz. (Birden fazla alana göre soru yapacağınız diye bir kurall yok!)

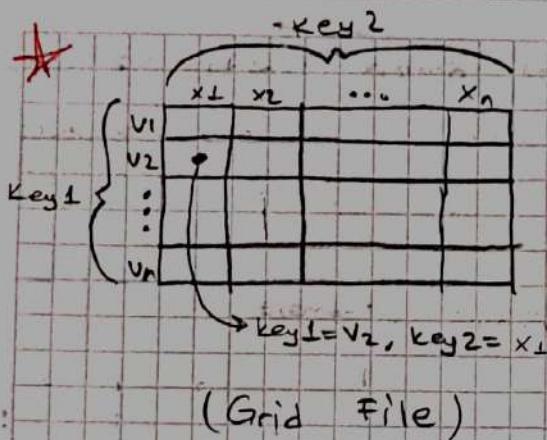
→ Coğrafi bilgi sistemleri için oluşturulan veri kümelerini sorgulamak için grid file yapısı çok uygun bir yapıdır.

o  $\langle x_i, y_i \rangle$  koordinatında hangi şehir vardır?

o  $\langle x_i, y_i \rangle$  koordinatının 5 km kapında hangi yerler var?

o  $\langle x_i, y_i \rangle$  koordinatında en yakın noktası neresidir?

gibi sorular grid file yapısı kullanılarak cevaplanabilin

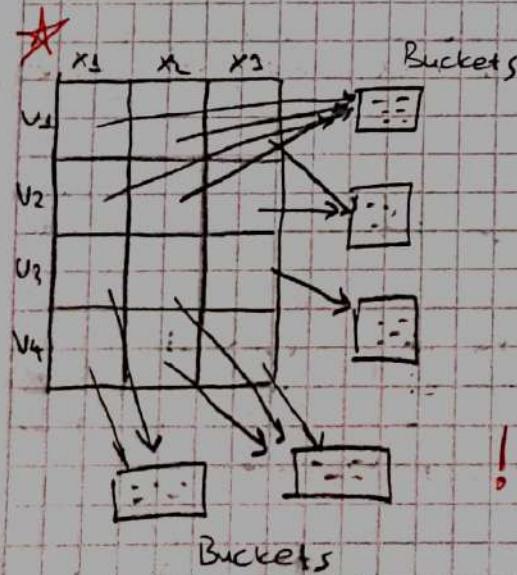
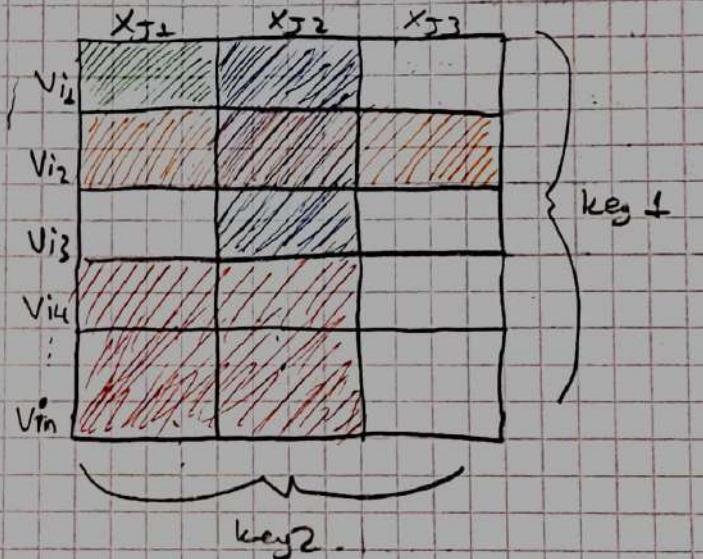


- Bunu aslında içi boyutlu dizi olarak saklayabiliriz. Bu içi boyutlu dizinin her bir elemanında bir pointer tutulur. Bu pointer da o kriteri sağlayan kayıtların bulunduğu bucket'in adresini veriyor.

|| Burada  $key1 = v_2$ ,  $key2 = x_1$  dan  
o kayıtların bulunduğu bucketin  
adresi tutuluyor.

→ Bu yapı kullanarak aşağıdaki sorular kolayca bulunur:

- $key1 = v_{i_1} \wedge key2 = x_{j_1}$
- $key1 = v_{i_2}$
- $key2 = x_{j_2}$
- $key1 > v_{i_3} \wedge key2 < x_{j_3}$



Grid yapısı yalnızca bucket adresini tutar.

Grid file yeterince küçükse RAM'de tutulur.

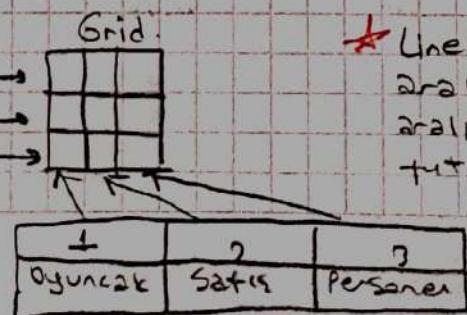
Bucket'lar diskte saklanır.

! Bu  $v_1, v_2, v_3, v_4$  veya  $x_1, x_2, x_3$  kümelerine  
bir aralık da yazılabilir. Örneğin,

0-20k	1
20-50k	2
50-∞	3

Maaş

↑  
Linear scale



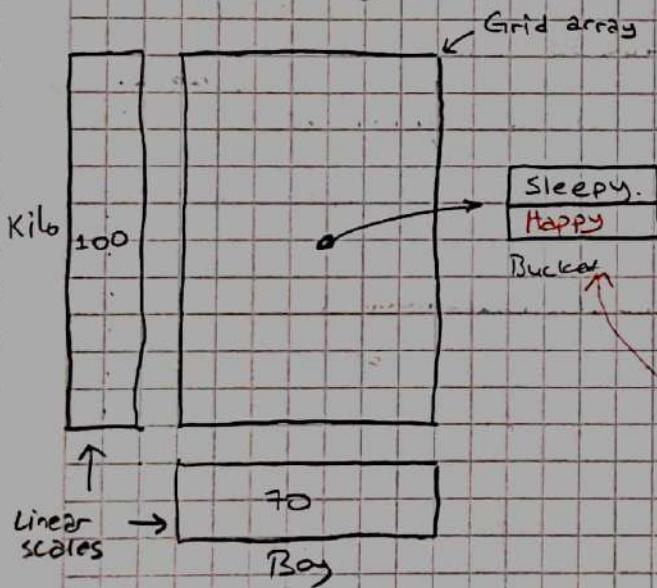
\* Linear scale bir aralık için kullanılır ve aralığın max değerini tutar sizece.

Örnek: K-boyutlu ağacı için yaptığımız örneği burada da yapalım.

İsim	Boy	Kilo	Diger Veriler
Sleepy	36	48	- - -
Happy	34	52	- -
Doc	38	51	-
Dopey	37	54	- -
Grumpy	32	55	-
Sneezy	35	46	-
Bashful	33	50	-
Ms-White	62	98	-

- Bu verileri, verilen sırada oluşturulan grid file'in durumunu bakaçagınızda 2 tane boyut kullanacağınız (boy ve kilo) ve her boyut için bir linear scale kullanacağınız (2 tane).
- Başlangıçta 1 bucket olacak ve her bucket 2 kayıt kapasitesine sahip.

ilk kaydı ekleyerek başlayalım:

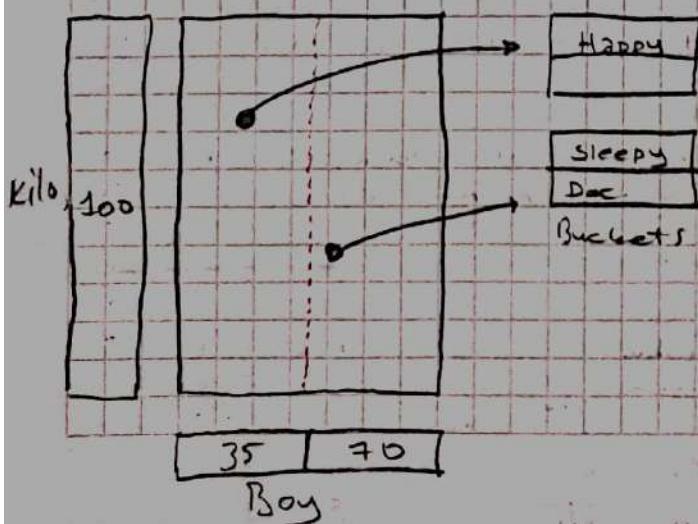


\* Boy ve kilonun üst sınırları belirlenerek bunlar yuvarlanır ve aralık şeklinde linear scale'le yerleştirilir.

\* Sleepy'nin olduğu konum sunucu ifade eder. Sleepy'nin boyu 0-70 kilosu 0-100 aralığının daddır. (Bu nokta en başta rastgele bir yerde olabilir)

\* Happy'i eklemek isterseniz gridin gösterdiği bucket'te yer olduğunda bu eklenir.

→ 3. kayıt olan Doc eklemek istendiğinde bucket'ta yer olmadığı görülür. Bu sebepten grid 1 boy'dan ilkiye bölünür. (Sırada gideceğiz)



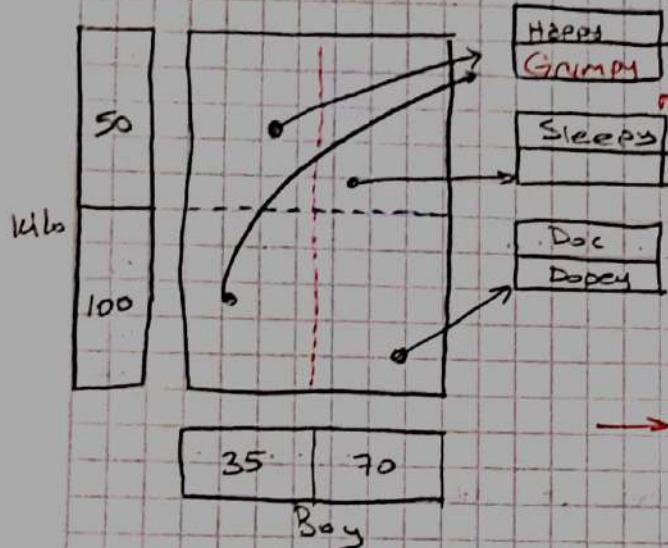
\* Burada solda kalanlar boyun 35'den küçük, sağda kalanlar 35 < x < 70 aralığında olacak şekilde düzüllerler.

\* Her bir grid bir adres saklar. Burada bölündükten sonra 2 grid 2 adres varken (Sonuç yok)

→ Dopey ekleneneceğinde, boyu  $37 \geq$  olduğu için Sleepye.

Doc'un olduğu yere eklenmemeli. Bucket dolu olduğundan

bu koz kilo'dan ikiye bölünür. (bir boy, bir kilo, bir boy...)



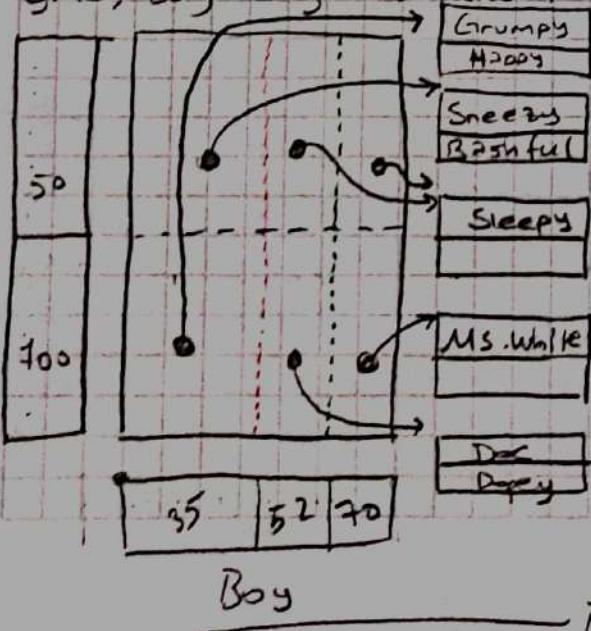
\* 4 grid'in 4 adres olması gerektiğinden fazladan adresi Happy'inin olduğu bucket'i bağladı.

→ Grumpy (32, 55) eklenirken;  $32 < 35$  olduğundan solda,  $55 > 50$  olduğundan sağda olacak. Buradaki pointer Happy'inin olduğu bucket'i işaret ediyor. Bu buckette boş yer var.

→ Sneezy (35, 46),  $35 \leq 35$  olduğundan sol,  $46 < 50$  olduğundan yukarı. Bu pointer'in işaret ettiği bucket dolu. Fakat burada gridi bölmeye gerek yok çünkü 2 farklı grid tek bir bucket'i işaret ediyor. Bucket'i ayırmamız yeterli.

→ Bashful (33, 50) eklenirken,  $33 \leq 35$  olduğundan sol,  $50 \leq 50$  olduğundan yukarı eklenerek. Dahası önce buraya Sneezy'i yazmıştık. Onu altına ekliyoruz.

→ Ms. White (62, 38) eklenirken,  $62 > 35$  olduğundan sağ,  $38 > 50$  olduğundan sağa yelestiriliyor. Ancak burası dolu. Bu yüzden grid, boy 2'ye bölündür.



\* Bir örnek soru yazalım:

+ Kilonu 50'den büyük olanları bul (Bunun için alt kısımdaki gridlerin gösterdiği adreslere gidilir ve bucketlardaki veiler zaten)

Grumpy, Happy, Ms. White, Doc, Dopey.

\* Göklü key, aralık ve eşitlik soruları, ilkin uygundur. Ancak aynı aralığa sürekli eklemeye yaparsak arka RAM'e sigamayarak düşmeye gelebilir.

## Vize için SORULAR

1-) Bir sektör = 200 byte, 1 yüzeyde 400 track, her track'te 100 sektör  
4 tane çift yüzeyli plakadan oluşan bir disk için ; (ortalama secc=16ms)

a-) Bir plakanın kapasitesi nedir?

$$\frac{200 \times 100 + 400 \times 2}{\text{Bir track boyutu}} \\ \frac{\text{Bir yüzey boyutu}}{\text{Bir plaka boyutu}}$$

Plakalar çift yüzeyli olduğu için 2 ile çarptık.

Plaka  $\rightarrow$  yüzey  $\rightarrow$  track  $\rightarrow$  sektör

b-) Bir track'in boyutu nedir?

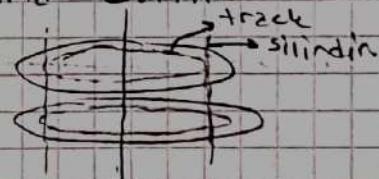
$$200 \times 100$$

Her sektör 200 byte ve bir track'te 100 sektör var.

c-) Kaç silindir vardır?

400

Silindir sayısı bir yüzeydeki track sayısına esittir.



d-) Bir silindirin kapasitesi nedir?

$$200 \times 100 \times 2 \times 4$$

$200 \times 100 \rightarrow$  bir track boyutu Silindir. tüm plakalardaki aynı hızda bulunan trackler bütünüdür. 4 plaka ve her plakada 2 yüzey olduğundan 2 ve 4 ile çarptık.

e-) Eğer plakanın dönmeye hızı 2500 rpm ise ortalama dönüsel gecikme (avg. rot. del) ne kadardır?

$$\text{Max rot. del} = \frac{1}{2500} * 60 * 1000 = 24 \text{ ms}$$

$$\text{Avg. rot. del} = \frac{\text{Max}}{2} = 12 \text{ ms}$$

sanİYEyi milisaniyeYE çevirdik.

Bir dönüş  
bu kadar  
zaman  
alıyor.

Ortalama dönüsel gecikme disk okuma yazma başlığı ilgili sektörde yerlestikten sonra plaka üzerinde bulunacak verinin başlangıcını belli etken geçen süredir.

f-) Bir blok 5 sektör içeriğinde blok transfer time nedir?

1 track, 100 sektör içeriğinden  
 $\frac{100}{5} = 20$  blok içeriğinde.

$$btt = \frac{\text{Max.rot-del}}{\text{Blok sayısı}} = \frac{24}{20} = 1.2 \text{ ms}$$

\* Bir bloğun okuma süresine btt denir.

\* Max rot del = Bir track'ı gezinme süresi. Yani 20 bloğu gezme süresi. 1 tane bloğun süresini bulmak için 20'ye bölüyoruz

2-)  $s = 16 \text{ ms}$ ,  $r = 8.3 \text{ ms}$ ,  $btt = 0.84 \text{ ms}$ ,  $B = 2600 \text{ byte}$

Dosyada 80.000 kayıt var ve her bir kaydın büyüklüğünü  $R = 400 \text{ byte}$ .

a-) Eğer dosya file file ise, belirli bir kaydı almak için geçen zaman nedir?

Her blok(sayfa)  $\frac{2600}{400} = 6$  tane kayıt tutabilir.

Bu durumda bu dosya  $\frac{80.000}{6} = 13334$  bloktan oluşur.

Fİle file'da bir kaydı alma süresi  $T_f = s + r + \frac{b}{2} * btt$

$$= 16 + 8.3 + \frac{13334}{2} * 0.84 \approx 5624.58 \text{ ms} \approx 5.6 \text{ saniye.}$$

b-) Eğer dosya sorted sequential file ise, bir kaydı almak için geçen zaman nedir?

$$T_f = (\log_2 b) * (s + r + btt)$$

$$= (\log_2 13334) * (16 + 8.3 + 0.84) \approx 344.53 \text{ ms} \approx 0.344 \text{ saniye.}$$

c-) Eğer bu dosya sorted sequential file ise ve 1/4 oranında kayıtlar overflow alanında ise, bir kaydı alma süresi nedir?

→ Sorted seq kısımları  $\log_2 b$ , overflow kısımları pile file gibi dövranır. Bu narin ikisini hesaplayıp toplayacagız.

$$\text{overflow} = \frac{13334}{4} \cong \boxed{3334}, \text{ sorted} = 13334 - 3334 = \boxed{10000}$$

sayfa

$$= (\log_2 10000) * (16 + 8 \cdot 3 + 0.84) + 16 + 8 \cdot 3 + \frac{3334}{2} * 0.84 = 1758.67 \text{ ms}$$

$$= 1.76 \text{ saniye}$$

d-) Bu dosya bir birincil index yapısı bulunduruyorsa ve bu index yapısı RAM'e sığa bilerek büyükükteyse bir kayde bulmak için, RAM'de arama yaptıktan sonra, bir disk erişimi gererir.

$$16 + 8 \cdot 3 + 0.84 = 25.14 \text{ ms} \cong \boxed{0.025 \text{ saniye}}$$

3-) Bir bankacılık sisteminde batch update yaptığımız düşünelim. Master file 100.000 banka hesabı bilgisini tutsun ve her bir kayıtlar 300 byte olsun. Transaction file 60.000 işlem barındırsın ve her biri 300 byte olsun. TF'de %30 yeni hesap açma, %60 hesap kapatma gibi kavram güncelleme işlemi yapıyor. Hiçbir kaydın bir den fazla bloğu kapsamasına izin verilmemişini varsayıyalım. Bu işlem için 3 tane buffer page ayrıldığını varsayıyalım.

$S=16$ ,  $r=8.3$ ,  $btt=0.84$  ve  $B=2000$  byte olsun. Bu durumda bu batch update'in süresi nasıl bulunur?

1 sayfada  $\frac{2000}{300} \approx 6.67 = 6$  kayıt bulunmaktadır.  
 → 1 sayfada 6 kayıt bulunmaktadır. Çünkü bir sayfanın kapasitesi 6'tır.  
 6 kayıt 16 byte'lık bir frame'ı tam olarak doldurmayı gerektirir.

$$\text{Master file da } \frac{100000}{6} = 16667 \text{ sayfa}$$

$$\text{Transaction file da } \frac{60000}{6} = 10000 \text{ sayfa.}$$

$$\text{Günlük dosyası, } 100000 + 60000 * \frac{30}{100} - 60000 * \frac{20}{100} = 106000 \text{ kayıt}$$

(Yapılan kayıt eklenme  
ve silme işlemlerinden  
sonra oluşan yeni dosya)

$$\frac{106000}{6} = 17667 \text{ sayfa}$$

\* Master ve transaction dosyalarının okunması ve output dosyasının diske yazılması için buffer pool kullanıyoruz. Her bir işlem için buffer pool da 1 frame veriliyor. Bunu için:

$$TMF = 16667 * (16 + 8.3 + 0.84) = 419008.38 \text{ ms}$$

$$TRF = 10000 * (16 + 8.3 + 0.84) = 251600 \text{ ms}$$

$$TOF = 17667 * (16 + 8.3 + 0.84) = 444148.38 \text{ ms}$$

$$\text{Toplam zaman} = 1114556.76 \text{ ms} \equiv 18.6 \text{ dakika.}$$

\* Örneğin master file, için buffer pool da 2 yer verilseydi.

$$TMF = \frac{16667}{2} * (16 + 8.3 + 2 * 0.84) \text{ olurdu.}$$

→ Çünkü her seferinde 2 sayfa okunacağından dosyanın yarısı kadar okuma yapılır.

Her seferinde 2 sayfa okunduğu için okuma zamanı 2 kat artar.

4-1) Bulk loading algoritmasını kullanarak, 20, 25, 10, 1, 16, 8, 3, 21, 22, 15, 9, 40, 45, 60, 13 derecesi ( $d=2$ ) olan ağaca bu sayıları ekleyin.

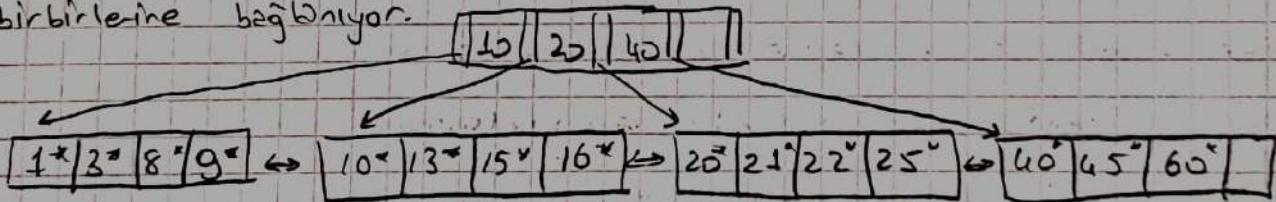
→ Öncelikle bu algoritma için sayıları sıralamalıız.

1, 3, 8, 9, 10, 13, 15, 16, 20, 21, 22, 25, 40, 45, 60.

→  $d=2$  Olduğundan her bir node'da en az 2 ve fazla 4 kayıt ( $\text{root} + \text{hariç}$ ) bulunduğunu anlıyoruz. (En fazla bulunan kaydın %50'si  $d$ 'yi vermelidir.) Yani bu sayıları 4'erli şeklinde grüplüyoruz.

1*	3*	8*	9*	10*	13*	15*	16*	20*	21*	22*	25*	40*	45*	60*	
----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	--

→ Daha sonra bir B+ ağacı oluşturmak için ikinci gruptan başlanarak her grubun en küçük üyesi bir yukarı kısma yazılıp pointerlar ile birbirlerine bağlanıyor.

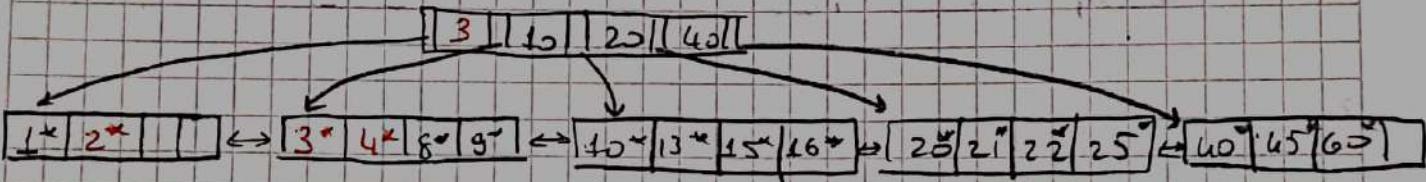


+ Böylece B+ ağacını tüm kurallara uygun birimde oluşturduk.

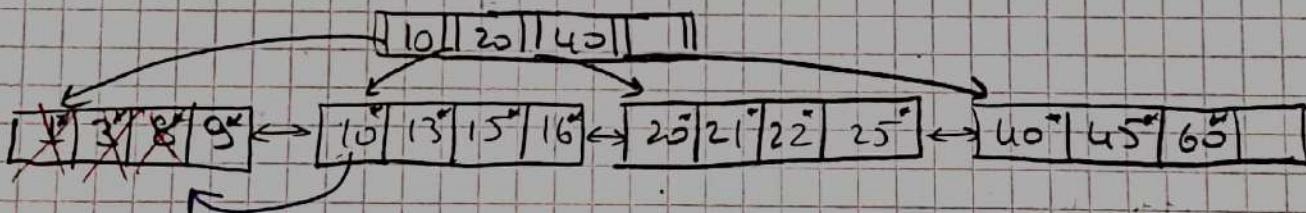
2 ve 4 eklemek istersen:

2 eklemek istediğimizde ilk node'la yerleşecektir. Fakat bu node dolu. Bunun için 2 eklemiş gibi sıralama yapır (1 2 3 8 9) ortadaki (3) değerini bir üstte taşıyarak bu node'u boşaltır. (Üste taşınan değer sırası bozmayacak şekilde yerleştirilmeli).

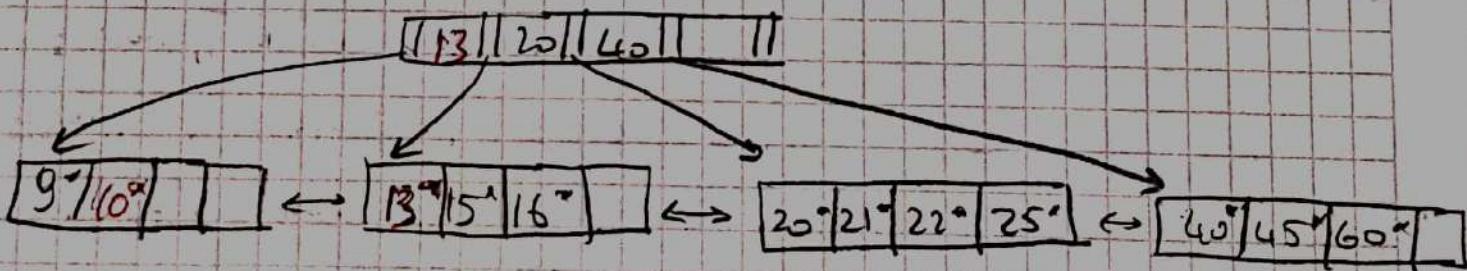
Daha sonra 4 eklemek istendiğinde bölünmüş node'da yer olacağinden direkt olarak buraya yerleştirilir.



Ağacın ilk halinden (2 ve 4 eklenmeden önce) 1, 3 ve 8'i silersek:



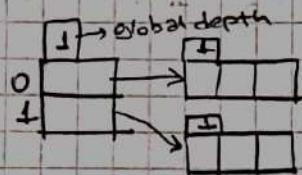
$\star$   $d=2$  olduğundan bir node da en az iki kayıt olmalı. Bunu kurallı bozmamak adına komşu node dan (komşu node atasından sağlı sollu dağılmış node lardır. Örn 10'un sağı ve solu) eğer yeteri kadar kayıt varsa (en az  $d+1$  kadar) en küçük elemen sağ tarafe aktarılır. Bu aktarım sonrası root kısmı bozulduğundan sola gelenin yerine sağda kalan en küçük anahtarın değeri yazılır. (Eğer komşusunda yeteri kadar yer olmasaydı merge (birleştirme) yapacaktı)



5-) 2, 15, 23, 6, 35, 13, 8, 26, 1, 11 anahtarına sahip, kayıtları hash tablosuna extendible hashing algoritması ekleyin.

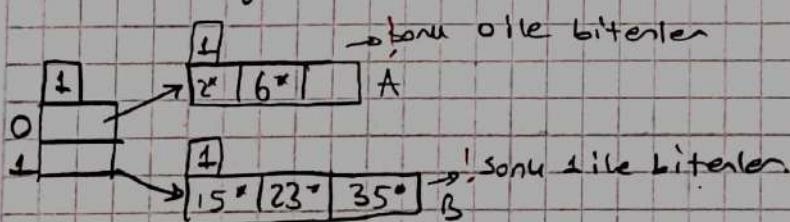
$h(key) = \text{key} \bmod \text{Global depth} = 1$  ve her bir data page 3 kayıt tutsun.

→ Öncelikle hash tablosunu oluşturalım.

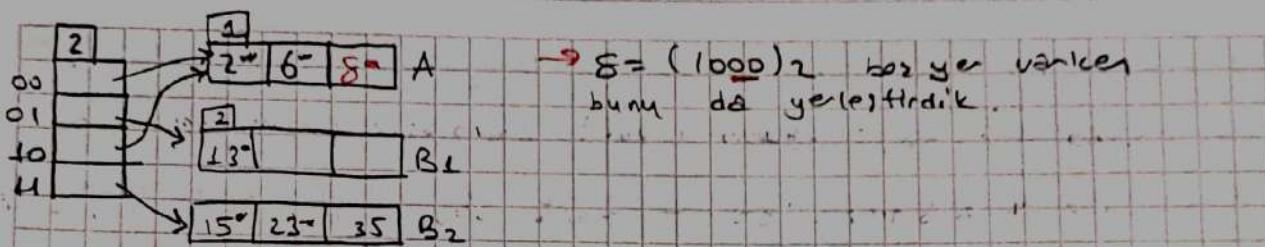


→ Yukarıda verilen sayıların keylerinin binary karşılığı bulunur ve bu binary sayının son dan depth kadar bitine bakılarak genetik yere yerleştirilir.

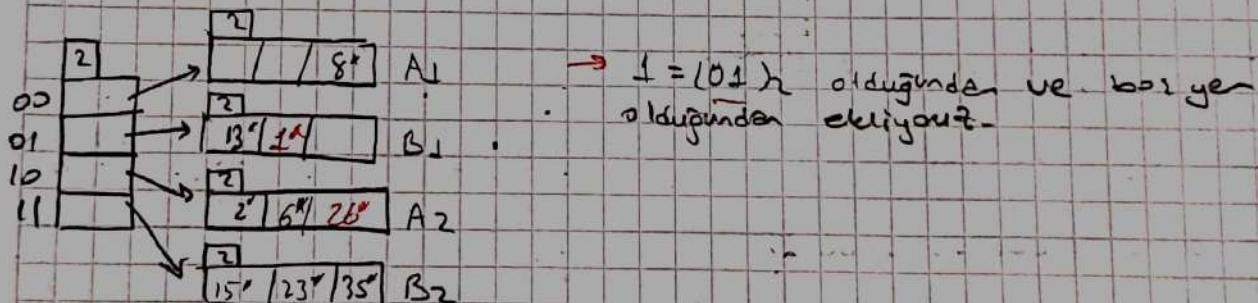
→  $2 = (10)_2$ ,  $15 = (1111)_2$ ,  $23 = (10111)_2$ ,  $6 = (110)_2$ ,  $35 = (100011)_2$  bunlar herhangi bir sıkıntı olmadan son 4 hanelerine göre yerleştirilir.



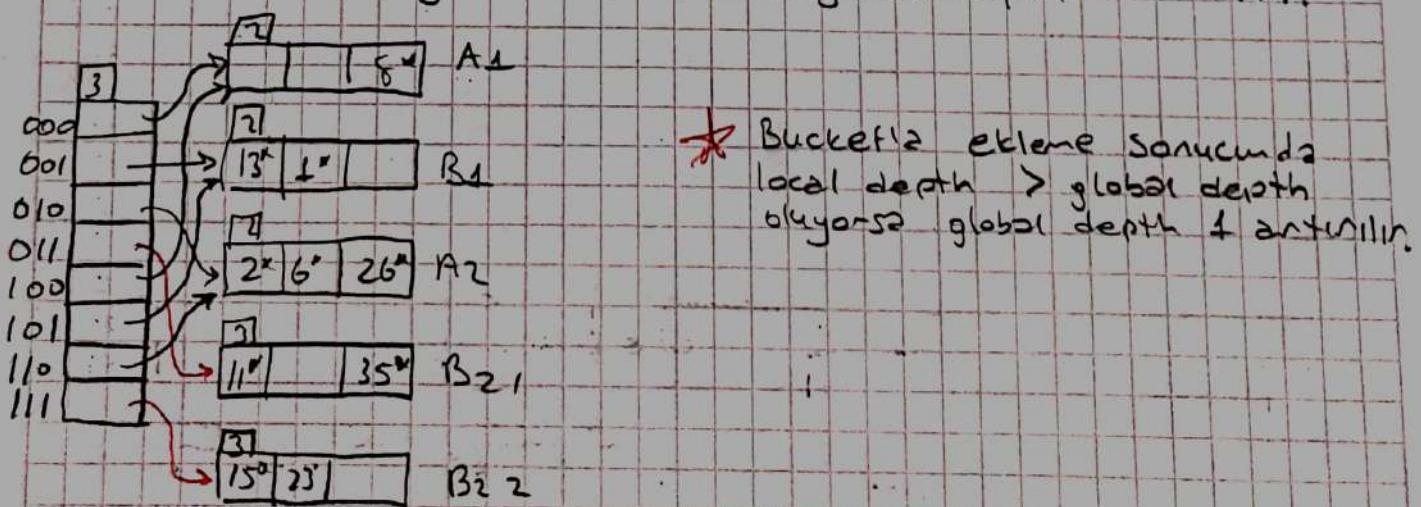
→ 13 = (1101)<sub>2</sub> eklemek istendiğinde son basamağı '1' olduğunu için aşağıdaki bucket'a yerleştirilmesi gereklidir. Fakat bu bucket dolu. Bunun için bu bucket'ı 2 ye bölyoruz. Bu bölünme sonucu 3 tane bucket oldu. Bütün adreslerini soldaki yepide tutuyoruz. Ama burada 2 adres tutmak için yeter var. Bunun için global depth'i 1 artırıyoruz. (Artık eklenen yepiken son 2 bas. baktırılacak.)



$\rightarrow 26 = (11010)_2$  eklemek istenirse 101'in gösterdiği A bucket'ına gidecek ama burası dolu. Bunun için bu bucket ikiye bölünür. toplam 4 bucket oldu. Sol tarafta yeterince adres tutmamış yer olduğu için global depth'i artırmak yerine 00 ve 101'in gösterdiği yerleri ayıryoruz.



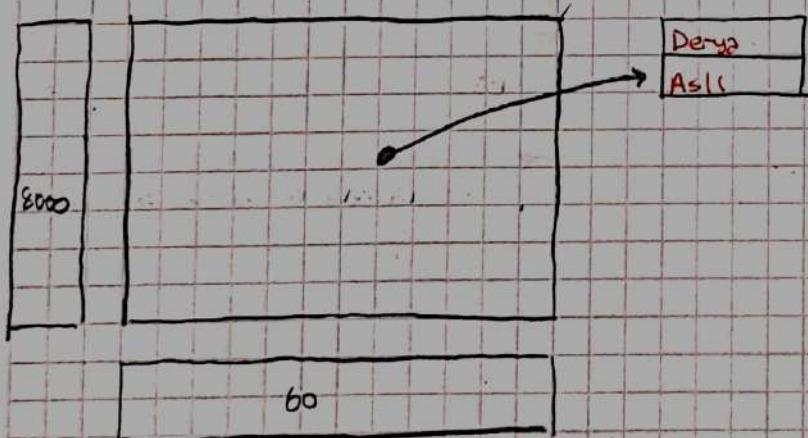
$\rightarrow 11 = (1011)_2$  - 11 dolu sayfayı bölince 5 bucket ama solda max 4 bucket ile yer var. Bunun için global depth + artırlın.



6-) Aşağıda verilen tabloyu grid file yapısına uygula.  
 (yaş 0-60 , maaş 0-8000 arası) Sırayla ekle. Her bir bucket'in kapasitesi = 2 . İlk boyut yaş , ikinci boyut maaş .  
 (Yani bir bölme işlemi olacağında önce yaş sonra maaşa göre bölme yapılacak)

isim	yaş	maaş
Derya	28	3500
Aslı	35	5000
Jale	18	2700
Deniz	27	3500
Mehmet	45	5500
Engin	55	5500
Sami	19	2600
Ruya	50	5500

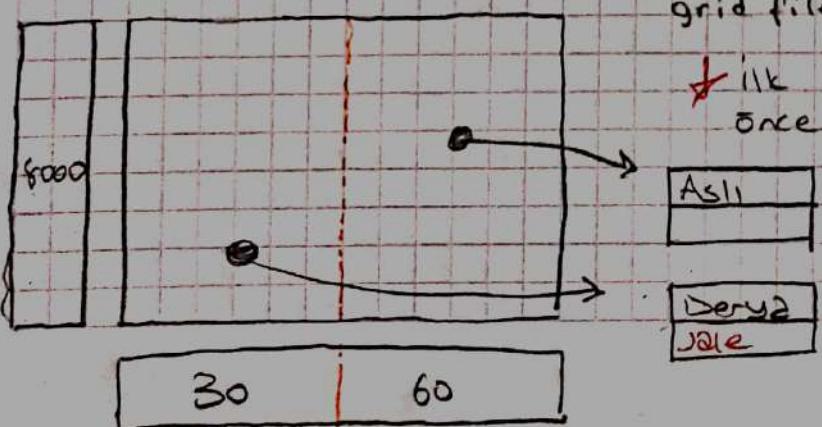
→ Bucket'ta yer olduğu için ilk iki kaydı direkt ekleyebiliriz.



→ Aslında soldaki grid file'ın sağsı sağdan bucketların adresini tutar.

Burada saat 0 < yaşı < 60  
 0 < maaşı < 8000

→ Jale (18, 2700) eklemek istendiğinde bucket'ta yer olmadığından grid file'li ikiye bölmeli gereklidir.



→ İlk boyut yaş olduğu için önce yaş bölüne.

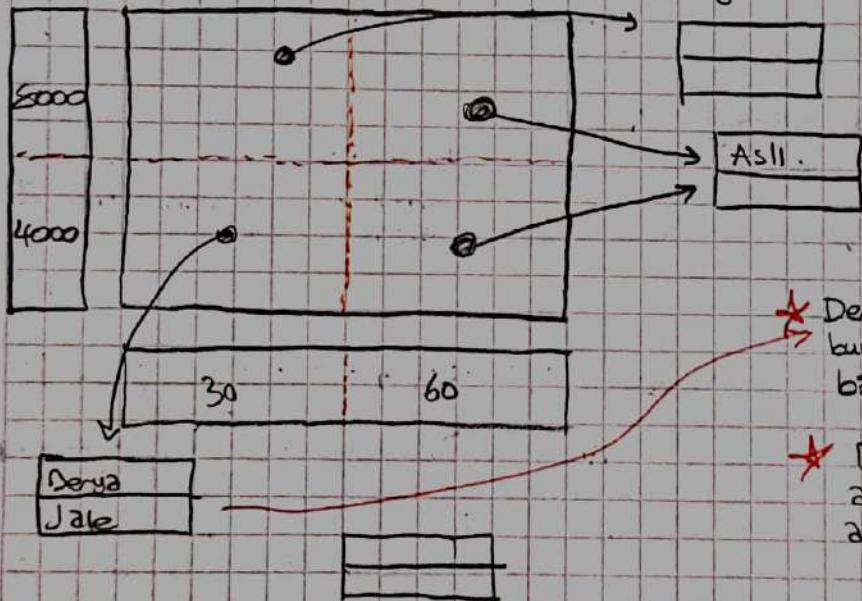
→ Bu bölünme sonrası yaş 30'da küçük olanlar solda büyük olanlar sağda kaldır.

★ ★ ★

Burada unutulmaması gereken kurallar eklerken hep aynı logolu b2t alarak etliyoruz (Bu örnekte yazar). Yalnızca tasm2 durumlarında sırayla bölüyoruz.

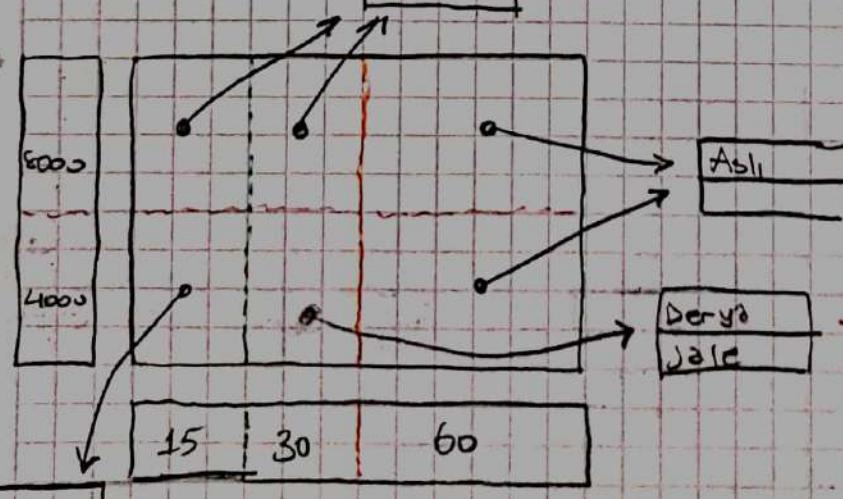
★ ★ ★

→ Deniz (27,3500) eklenmek isteniyor.  $Y_{27} < 30$  olduğundan sol tarafın gösterdiği bucket'a gider. Bucket dolu. Bölme gerçekleşir. Fakat bu kez maaşa göre.



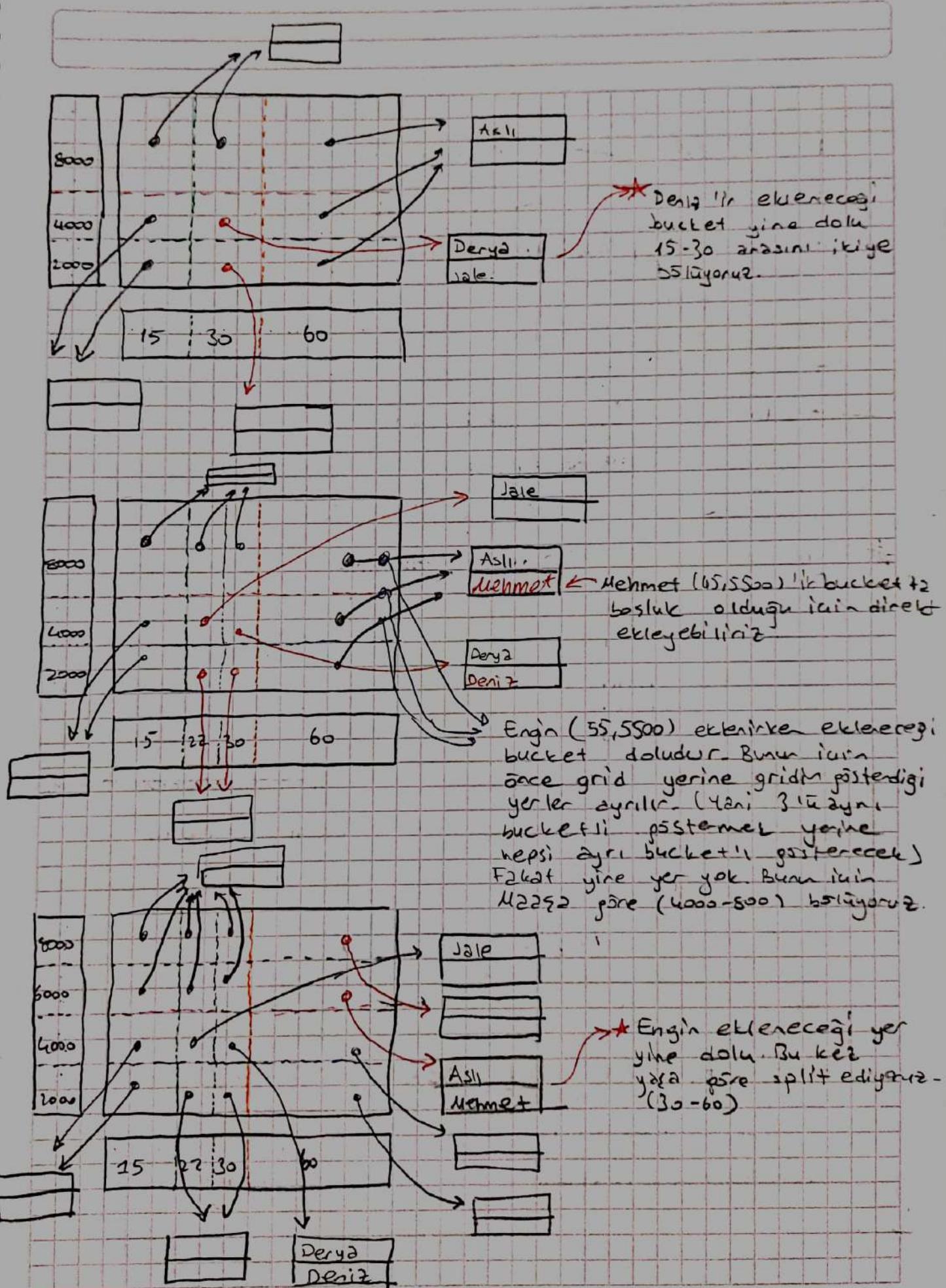
★ Deniz (27,3500)'in ekleneceği bucket yine dolu yine bölüyoruz. Bu kez yaşla göre.

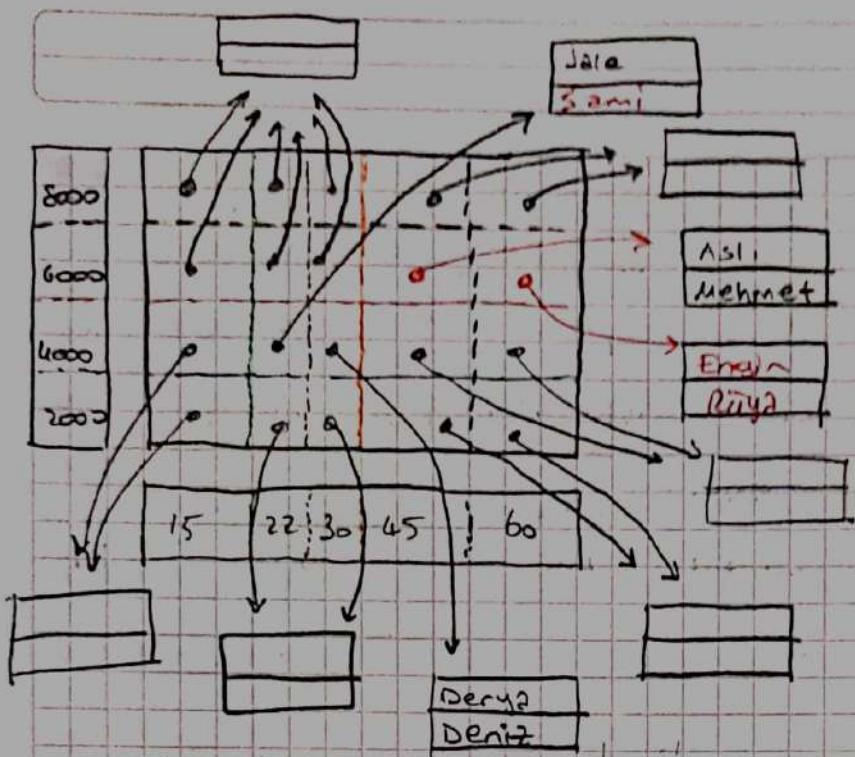
★ Bütünerek bucket 0-30 aralığında olduğundan bu aralığı bölüyoruz.



→ Deniz'in eklenceği bucket yine dolu. Bu kez maaştan bölüyoruz.

★ Bütünerek bucket 0-4000 aralığında olduğundan bu aralığı bölüyoruz.





★ Boşluk olduğunu için Sami (19, 2600) ve Rüya (50, 5500) kayıtlarında ekledik.

HAFTA VII. SON

VİZE

HAFTA VII. SON

## CHAPTER 13:

### - EXTERNAL SORTING -

Neden sıralama (sort) yapmaya ihtiyacınız?

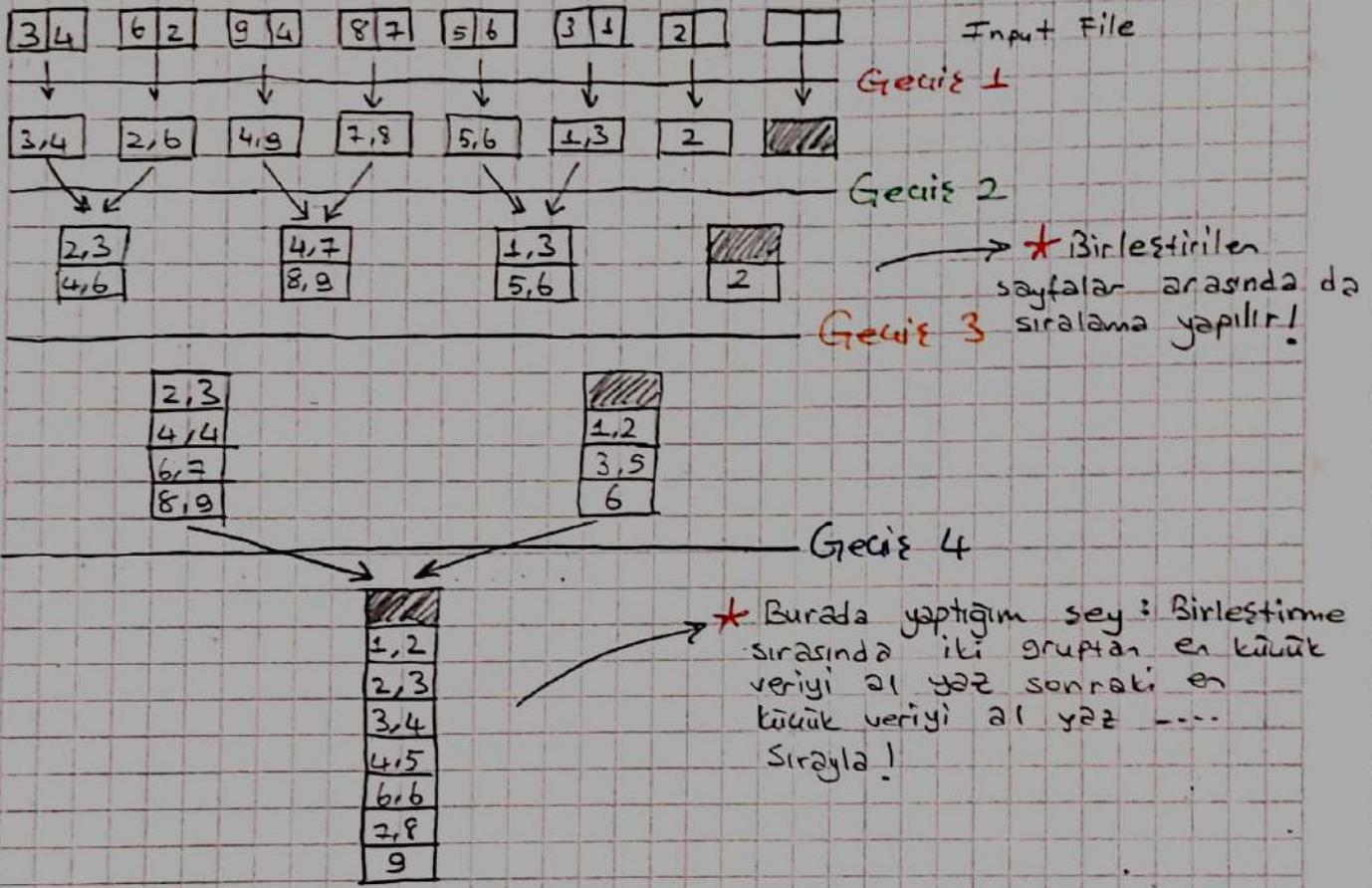
- ★ Sıralama aslında bilgisayar bilimlerinin klasik problemidir.
- Veriler sıralı bir şekilde istenmiş olabilir. (örnegin öğrenciilerin ortalamalarını artan sırada ile isteyebiliriz)
- B+ ağacı oluşturmak için uygulanan Bulk-Loading algoritması sıralama kullanır.
- ★ 10 GB'lık bir veri 1 GB'lık bir RAM kullanılarak sıralama işlemine alınmak istenirse External Sort kullanılır. Yani belleğe sağlanamayan (Harici Sıralama) büyüklerdeki veriler için kullanılır.

### Two-Way External Merge Sort :

Bu algoritma belleğin çok küçük olduğunu farz eder. (sadexe 3 tanesi buffer page olduğu varsayıllır). Su şekilde çalışır:

- Birinci geçişte, bellekten bir sayfalık yer kullanır ve dosyanın her bir sayfasını belleğe okur. Bellekte bir sıralama algoritması (quick sort, insertion sort, merge sort ...) ile kayıtları sıralayıp sıralanan kayıtları tekrar diske yazır.
- ikinci ve sonraki geçişlerde, sayfalar kendi içlerinde sıralı olduğundan iki sayfayı diskten belleğe okur ve bunları merge eder (birlestirir). Bu olusan yeni sayfayı diske yazar.

Örnek :



\* Bu örnekte, dosyada 7 sayfa vardır. (En sağdaki sayfa örneği daha-  
yi anlatmak için var). Bu sayfaların içinde kayıtların keyleri var.  
Birinci geciste tüm sayfalar kendi içlerinde sıralanır. ikinci ve  
daha sonraki gecislerde bu sayfalar sıralı olacak şekilde birles-  
tirilmeye devam eder.

\* Her bir adımda her bir sayfa diske okunup işleminden sonra diske geri yazılır.

N=sayfa sayısı olmak üzere : ( $N=7$  bu örnekte, sondaki sayılmaz)

$$\rightarrow \text{Gecis Sayısı} = \lceil \log_2 N \rceil + 1 \xrightarrow{\text{genel}} \text{Genel Formül}$$

$$\rightarrow \text{Toplam maliyet} = 2N * \text{Gecis Sayısı} = 2N * (\lceil \log_2 N \rceil + 1)$$

L 2N olmasının nedeni burası.

\* Sure maliyetini bulmak için :

$$\text{Toplam Maliyet} \rightarrow (s+r+btt)$$

(random access olduğunu için)

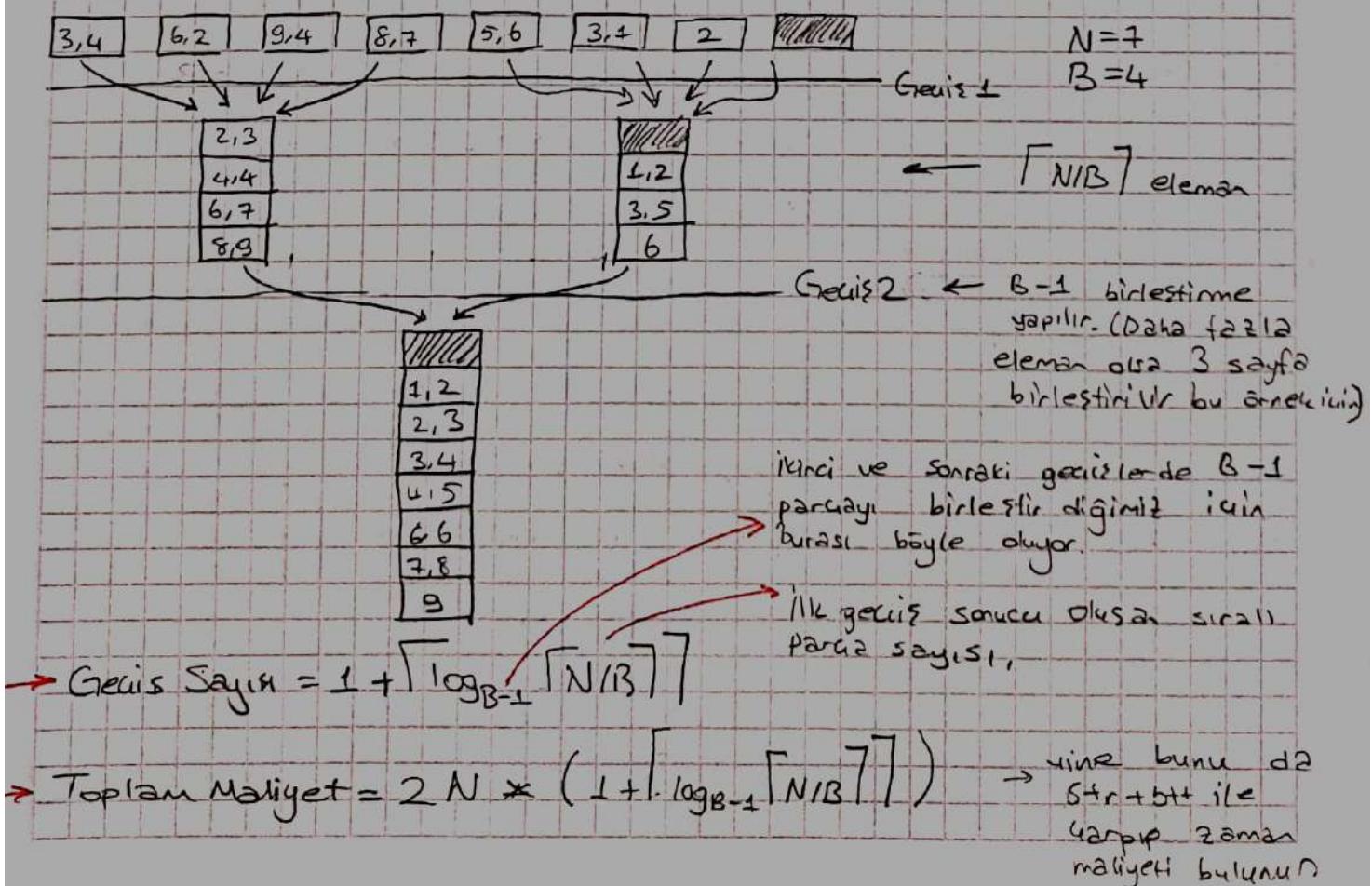
### General External Merge Sort:

Bir önceki algoritmda (Two-way) belleğin çok küçük olduğu varsayılmıştı.  
( $B$  buffer)

Fakat pratikte bellekte çok daha fazla alanı kullanabiliriz. Bu yüzden bir önceki algoritma üzerinde değişiklik yaparak bellekten  $B$  tane buffer pagelin sıralama için kullanıldığını varsayıyoruz.

- Birinci geçişte sayfaları birer birer okumak yerine  $B$  tane sayfayı belleğe okuyup, sıralayıp, diske yazar. (Dosya sonuna kadar)  
Birinci geçisin sonunda elimizde  $\lceil N/B \rceil$  tane sıralı dosya parçası olur.
  - ikinci ve sonraki geçislerde ikili birlestirmek yerine  $B-1$  tane parçayı birlestiririz. Bellekte  $B$  sayfa vardı  $\lceil$  tanesini merge sonucunu yazmak için alırsak geriye  $B-1$  kalır.
- \* Two-ways algoritması, bu algoritmanın  $B=3$  olunmuş halidir.

Örnek:



Örnek: External Merge Sort kullanılacak.

$$N=108, B=5$$

(sayfa)  
(frame sayısı)

↳ buffer pool  
↳ frame sayısı

$$\text{Formül} = 1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$$

(gecis)

- Birinci geçis:  $\lceil 108/5 \rceil = 22$  sıralı dosya parçası. (son parça 3 sayfanın toplamı ile olustur)
- İkinci geçis:  $\lceil 22/4 \rceil = 6$  sıralı dosya parçası. (birinci geçisten sonra 3'te böl)
- Üçüncü geçis:  $\lceil 6/4 \rceil = 2$  sıralı dosya parçası.
- Dördüncü geçis: Tüm dosya sıralandı!

### Internal Sort Algorithm:

External Sort Algorithması üzerinde iyileştirme yapılmış halidir.

\* Buradaki fark; birinci geçiste  $B$  tane bloğu buffer'a aktardıktan sonra sıralamak için quick sort yerine, heap sort algoritması kullanılır. Bu sayede  $B$  sayfalık veri sıraladıktan sonra  $>B$  uzunlığında sıralı dosya parçası elde ederiz.

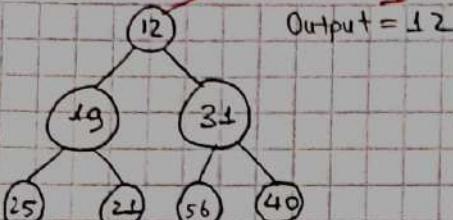
### Şu şekilde oluşturulur:

Diskten  $B$  sayfalık veriyi okuruz. Bu veriyi kullanarak bellekte binary min heap oluşturulur. Sonrasında gelen input değeri kökten büyük olduğu sürece; kökü output buffer'a yaz, gelen değeri köke aktar. Eğer kökten daha küçük bir değer mevcutsa bu değeri köke taşımak için heapify yap.

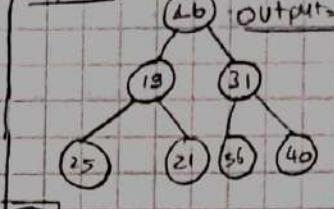
16 değerini sıralamayı  
bozmadığından heapify  
yapmadık

99

Input: 16



Input: 29

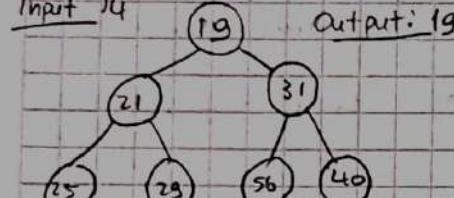


heapify  
428

99

21 < 23 olasılığ  
için

Input: 14



15

Bunu başka  
arrayde tutar.

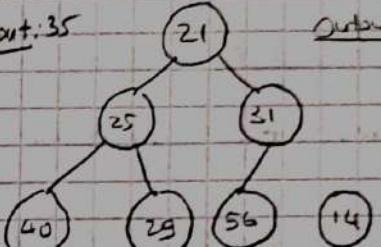
\* En sağ alt + 2 ki  
eleman kök olur ve  
heapify yapılır.

4

| kökten küçük  
input geldi.

| En sağ alt + 2 ki  
eleman kök olur ve  
heapify yapılır.

Input: 35



Output: 21

\* Bu kar küreme örneği ile ifade edebiliriz. Bir kar küreme aracı olsun. Bu aracın bir cember üzerinde sabit bir hızla dolastığını ve kar yağdığını sırada karları küredığını varsayalım. İlk durumda cember üzerinde B büyüklüğünde kar olsun. Küreme işlemi başladığında henüz kürinemeyen yerlere de kar yağacağı için cemberde bir tur atınca 2B büyüklüğünde kar temizlenmiş olur.

\* Aslında tek yapmayız şey quick sort yerine heap sort kullanmak.

$$\rightarrow \text{Geçiş Sayısı} = 1 + \lceil \log_{B-1} \lceil N/2B \rceil \rceil$$

$$\rightarrow \text{Toplam Maliyet} = 2N * (1 + \lceil \log_{B-1} \lceil N/2B \rceil \rceil)$$

\* Bu sayede disk erişim sayı büyük ölçüde azaltılmış olur.

\* Sadexe 1. geçiste uygulanıyor !!!

## I/O for External Merge Sort:

Bu da başka bir iyileştirmedir.

\* Bu iyileştirme 2. ve sonraki geçişler için uygulanır!!!

Buradaki fark; merge işlemi yaparken her sıralı dosya parçasından birer sayfa okumak yerine, ardışık olarak 'b' kadar sayfayı okuruz. Bu sayede merge edilerek sıralı dosya parçası sayısı azalır.

\* Birinci geçişte heap sort, ikinci ve daha sonraki geçişlerde bu yöntem kullanılırsa buna optimal sort denir.

$$\text{Geçiş Sayısı: } 1 + \lceil \log_{B/b} - 1 \lceil N/2B \rceil \rceil$$

\* Burada merge edilen sıralı dosya parça sayısı azaldığından geçiş sayılarında bir artma meydana gelir. Fakat b tane sayfaya ardışık disk erişimi yapmak için okuma süresinde azalma olur.

## Double Buffering:

General External Merge Sort algoritmasındaki bir diğer iyileştirmedir.

\* Bu iyileştirmeye göre; buffer pooldaki sayfaların yarısı diskten veri okuma-yazma için kullanılırken diğer yarısı da işlenici tarafından işlem yapılır. İzin, kullanımın doğasıyla bellekte B. sayfa varsa bunun yarısı diskten veri okuma-yazmak için kullanılırken diğer yarısı örneğin; birleştirmeyi (merge) hesaplamak için kullanılır gibi.

\* Bu yüzden geçiş sayısı artar. ( $B-1$  merge etmek yerine  $B/2-1$  edilir.)

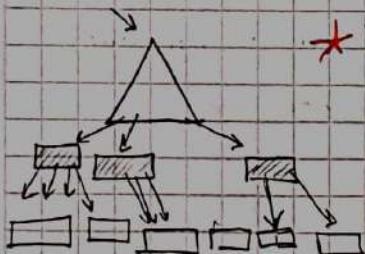
\* Toplam beklemeye süresi azalır. Çünkü işlenici olayları ve disk okuma yazması paralel gerçekleşir. (Dünyada)

\* Toplam sürede azalma olur!!!

## B+ Ağacı Kullanarak Sıralama : (Using B+ Trees for Sorting)

Sıralamak istediğimiz bir dosya ve bu dosyanın B+ ağacı indeksinin daha önceden oluşturulanını varsayıyalım.

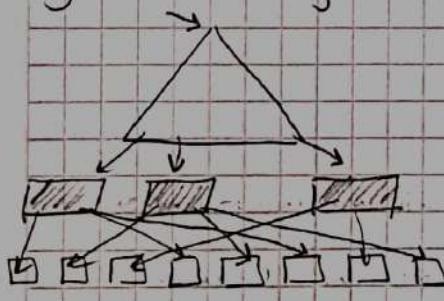
- Eğer bu B+ ağacı clustered (kümelmemis) ise :



\* Veriler birincil anahtara göre sıralandıkları için diskte de sıralı halde bulunurlar. Verilerin hepsini tek bir disk erişimi yaparak sıralı halde RAM'e aktarabiliriz.

\* Her zaman general external merge sort'tan daha iyidir. ✓

- Eğer bu B+ ağacı unclustered (kümelmemis) ise :



\* Veriler ikincil anahtara göre sıralandıkları için ağacta sıralı olmalarına rağmen diskte sıralı degillendir. Verilerin hepsini sıralı almak için tek tek kayelerin gösterdiği yere gitmemiz gerekdir.

\* Diske rastgele erişim olacağından kötü bir performansa sahiptir. ✗

---

HAFTA IX SON

---

## CHAPTER 1 :

### - DATABASE MANAGEMENT SYSTEMS -

- DBMS, veritabanlarını saklamak, sorulamak, yönetmek için gerekli olan yazılım parçalarının olduğu bir yazılım paketidir.
- Her DBMS bir veri modeline dayanır. Bu derste ilişkisel veri modelini göreceğiz. (relational Data Model)
- Veri modeli, gerçek hayatı反映する problemsi için veriyi nasıl modellereceğimizi bize gösterir. Öğrenci, ders  $\rightarrow$  özn: kayıt yaptırma.
- İlişkisel veri modeli varlıklar ve ilişkilerden oluşur. (Öğrenci ders kayıt olur) (entities) (relationship)

### Files vs DBMS:

- Veri asıl olarak dosyalar halinde saklanıyor. Fakat bu dosyaların diskte saklanması, dosyalara yeni veri ekleme, silme, güncelleme, veri sorulama gibi işlemleri DBMS ile yapıyoruz. DBMS olmasaydı, bu işlemlerin her birini kendi elimizle yapmamız gerekti.
- DBMS içinde sorulama dili kullanarak ekleme, silme... gibi işlemleri yapıyoruz. (örn: SQL)
- DBMS sayesinde Veri tutarlılığı ve veriye eş zamanlı erişim yapılması sağlanır. (örn: Öğrenci işleri sisteminde kayıt zamanı yüzece öğrenci sisteme eş zamanlı erişim sağlıyor ve bu kayıt esnasında değişimlere göre sistemin tutarlılığı anlık bir şekilde sağlanıyor.)
- Sistem wökmelerinde, DBMS, verinin tutarlılığını sağlar.
- DBMS, veritabanındaki verinin güvenliğini sağlar. (örn: Farklı kullanıcılar farklı yetkiler voererek, o kullanıcının sadece kendine verilen yetki doğrultusunda işlem yapması sağlanıyor)

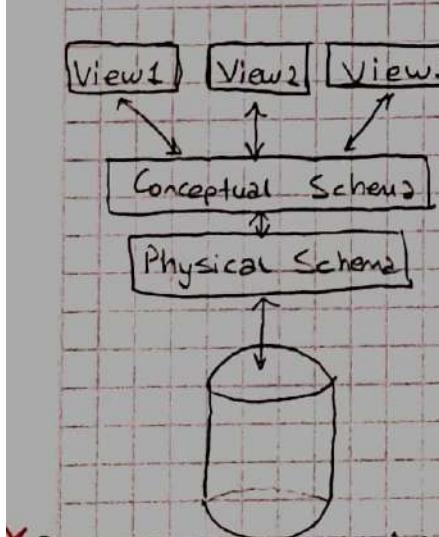
## - Neden DBMS kullanıyoruz?

- Verinin bağımsız olmasını ve veriye etkin bir şekilde erişmek için kullanılır. (Veriye bir programın doğasıyla etkin erişim)
- Uygulamayı geliştirmek gecen süreyle kısıtlar.
- Veri bütçülüğünü ve güvenliğini sağlar.
- Veri yönetiminin tek bir eden yapısını sağlar. (DB Administrator)
- Eski zamanlarda erişim ve sisteme hakemelikte verinin korunmasını sağlar.

## Veri Modelleri : (Data Models)

- Her DBMS bir Veri modeline dayanır. Veri modeli, veriyi nasıl tanımlayacağımız ifade eden kavramlar bütündür. Bu modeli kullanarak verinin semasi açıklanır ve veritabanına aktarılır.
- Bugün en çok kullanılan Veri modeli : ilişkisel veri modelidir.  
İşlekisel Veri modeli ilişkiden olur. (Yani tablodan olur)  
(relation) ~~bu anlama denegeli~~ ↗ relationship'te ilişki demek karıştırma !!!  
→ Her satırda bir kayıt, her sütundada bir nitelik bulunur
- Her tablonun (relation) bir seması vardır. Bu sema sütunları tanımlar.  
(schema)  
(Yani bu tabloda hangi özellikler, nitelikler bulunur ve veri tipleri nelerdir?)

## Sayıtlama Seviyeleri : (Levels of Abstract)



- ★ Görüldüğü gibi DBMS'de katmanlı bir sayıtlama yapısı vardır.  
↗ Her tablo dosyası独立运作
- ★ Veritabanı, disk üzerinde saklanan tablolardır.  
↗物理文件
- ★ Fiziksel sema, her bir dosyanın türünü ve bu dosya üzerinde kullanılan index yapılarını tanımlamamızı sağlar.  
(file file yapısında BT adı da index örnek)
- ★ Kavramsal sema, her bir tablonun sütunlarını yanıtları ve bunların veri tiplerini tanımlamamızı sağlar.
- ★ Aynı veritabanı üzerinde çok sayıda view oluşturulabilir. Her bir view, kullanıcının veriyi nasıl göreceğini tanımlar. Dönen her ögrencinin yalnızca kendi notlarını görmesini view sağlar.
- ★ Bu yapıları oluşturmak için Veri tanımlama dili ile tanımlanır. (SQL gibi) kullanarak sağlanabilir.
- ★ "Sorgular veri istenilen dili ile döndür. (SQL ikişini de kapsar)"

Örnek: (Université veritabanı)

→ Kavramsal Sema (conceptual schema) :

- Students ( sid:string, name:string, login:string, age:integer, gpa:real )
- Courses ( cid:string, cname:string, credits:integer )
- Enrolled ( sid:string, cid:string, grade:string )

→ Bu veri tabanında üç tane takip olduğu görülmüyör. Her bir tablonun sütunları ve bu sütunların veri tipleri kavramsal semada bulunur.

\* Kayıt (Enrolled) tablosu, öğrenci ve dersler arasındaki ilişkisi sağlayan bir tablodur. Yani hangi öğrenci hangi dersde kayıt yaptırmış ve bu dersden hangi notu almış bilgilerini tutar. Enrolled tablosunda kullanılan sid ve cid değerlerinin tipleri diğer tablolardaki ile aynı olmalı.

→ Fiziksel Sema :

- \* Tablolardan her birinin hangi dosya türünde saklanacağı ve dosyaların üzerinde tanımlı olan index yapılarının tanımlandığı semadır.
- Tüm tablolar pile file setinde tutulsun.
- Students tablosunda sid'ye göre index oluşturulsun.

→ View :

- Course\_info ( cid:string, enrollment:integer )
- \* Belirli kullanıcıların verinin belirli kısımlarını görmesini sağlamak için kullanılır. (Çok sayıda tanımlanabilir)
- \* Bu view'a sahip bir kullanıcı yalnızca ders kodunu ve ödersi konusunun aldığı görebilir. Burun dışındaki herhangi bir bilgiyi göremez. (Hatta dersin ismini bile göremez!)

### Veri Bağımsızlığı: (Data Independence)

- Veri bağımsızlığı, uygulamanın verinin nasıl saklandığından ve mantıksal yapısından etkilenmemesidir.
- Mantıksal veri bağımsızlığı, örneğin; kavramsal şemada yeni bir sütun eklemesinin uygulanmayı etkilemeyeceği anlamına gelir.
- Fiziksel veri bağımsızlığı, örneğin; file file olan tablo üzerine BT ağıci tanımlamış olsun. Bu değişikliğin uygulanmayı etkilemesine fiziksel veri bağımsızlığı denir.

### Eşzamanlılık Kontrolü: (Concurrency Control)

- Aynı anda birden fazla kullanıcının veya programın aynı DB üzerinde çalışmasıdır.
- Programlar DB üzerinde değişikliklere sebep olacaksa bu işi belirli bir sırada yapar. Örneğin: Birinci programın diskten veri okuması gerekiyorsa onu bekletip ikinci programa öncelik veriyor daha sonra birinde dönüp okuduğu verileri hesaplanması için CPU'ya gönderiyor... Bu yapıya interleaving actions denir.
- Tüm bunları yaparak kullanıcılar, sistemi kullanın sadece kendi yaradılmış gibi bir algı oluşturur.

### Transaction: An Execution of a DB Program:

- Eşzamanlılık kontrolü için tanımlı olan en önemli kavramdır.
- Bir DB programının bir kez çalıştırılması bir transactiondır.
- Transaction atomic olmalıdır. Yani transaction başladığında ya başarıyla sonlanmalı ya da (bir hata meydana gelirse) hiç başlamaması gibi davranışmalıdır.
- Transactions, DBMS tarafından birdizi okuma/yazma işlemlerinden ibarettir.

### Scheduling Concurrent Transactions:

Eş zamanlı gerçekleşen transactionların veri tutarlığını sağlamak için belirli bir sıraya konulup çalıştırılması gereklidir. Bir transaction bir nesneyi okurken başka bir transaction aynı nesneye ulaşmaya çalışırsa veri tutarlığını bozulabilir. Bu nedenle Strict 2PL gibi bir yöntemle mesajlı olan nesnelerde kilit kullanılır.

Strict 2PL'in dezavantajları da vardır. Örneğin; I. transaction X'ı nesnesini ve II. transaction da Y'ı nesnesini okusun ve bunlara kilit uygulayın. I. program Y'yi II. program X'ı okumak isterse burada bir sonsuz döngü olusur ve iki transaction da iptal edilir.

### Ensuring Atomicity:

Transactionların atomik olması gerektiği belirtilmisdir. Bu nedenle log (geçmiş) kaydı tutarak sağlayız. Bir transaction, nesne üzerinde değişiklik yapmadan önce nesne, log dosyasına kaydedilir. Değişiklik başarılı olursa buju da log'a yazılır.

### DBMS Kullanıcı Grupları:

End-user: (Über sistemini kullanan öğrenci ve öğretmenler.) DB'nin nasıl kullanıldığı bilgilerine gerek yok!

### DBMS programı satıcıısı

### DB app programmer ve DB admin.

↓                          ↓  
Überi kodlayan              Überin DB yapısını  
visiler                      oluştururan kişiler

### DBMS yapısı: (Structure of DBMS)



→ Disk Space → Buffer → Files and Access → operators  
DB Management      management methods

Query  
Relational      optimization  
and  
Execution.

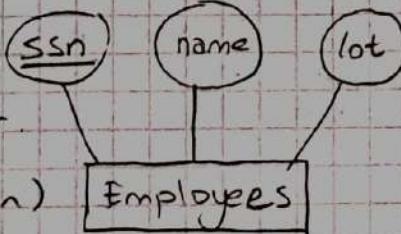
## CHAPTER 2

### -The Entity-Relationship Model-

- Günümüzde pek çok veri modeli ilişkisel veri modeline dayalıdır.
- Bu chapterda ilişkisel DB kullanarak bir problemi nasıl çözeceğimizi öğreneceğiz.
- Bir gerçek hayat problemi için ilişkisel veri modelini kullanarak nasıl kavramsal tasarım yapacağımızı öğreneceğiz. Bu kavramsal tasarım yapmak için şu soruların cevabı aranmalıdır :
- Problemdeki varlıklar ve ilişkiler nelerdir? (Cübis sisteminde entities (varlıklar) relationship (ilişkiler))  
varlıklar: öğrenci, öğretmen, dersler ...    ilişkiler: ders alır, ders verir....)
- Bu varlıklar ve ilişkilerle ilgili hangi bilgileri DB'de saklamalıyız?  
 (Öğrencinin numarası, adı, soyadı, adresi, ortalamasını saklamak gibi)
- Büfönlük sınırlaması ve iş kuralları nelerdir? (Örneğin: Not 0-100 arasında olmalı diye bir kural koymak)  
 Tüm bu soruları sorup bunların yanıtlarına göre bir şeit çiziz.  
 Buna ER diagramı denir. Bu ER diagramını kullanarak DB tablolarını oluşturacağız.

### ER Modeli : (ER Model Basics)

- \* Varlıklar, diğer nesnelerden ayırt edilebilen nesnelerdir. Varlıklar bir ya da daha fazla nitelik(attribute) kullanarak tanımlanır.
- \* Örneğin personel varlığı, sosyal güvenlik no(ssn), adı(name), maaşı(lot) birer niteliktir.
- \* Aynı türden varlıklar bir araya gelerek varlık küməsini oluşturur. (entity set)  
 (Örneğin; bir fabrikadaki tüm personeller, personel varlık küməsinde bir araya gelir.)
- \* Her varlık küməsinin anahtarı vardır. (altı çizili kısım) Bu anahtarlar kayda özel sayı olmalıdır T.C. gibidır

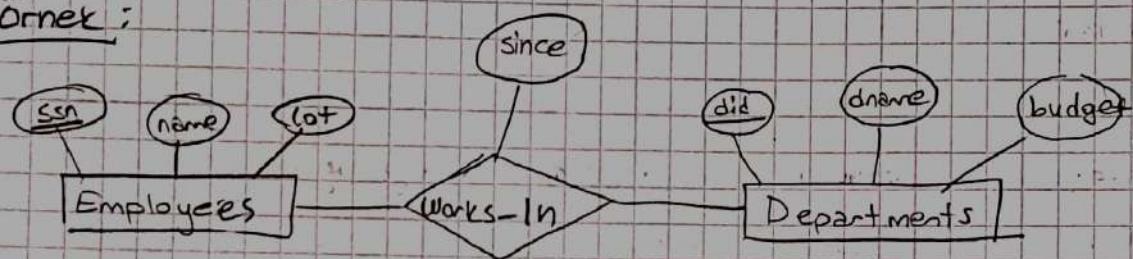


\* Varsayıclar dikdörtgen içinde yer alır. Nitelikler ellips içinde yer alır.  
Anahtar niteliğin altını çiziyoruz. (Primary keys yani).

Her niteliğin belirli bir tipi vardır. (örn: ssn = int, name = 50 karakterli string...)

\* İlişkiler, iki ya da daha fazla varlık arasında tanımlanmış  
birliktekliliklerdir. Aynı türden ilişkiler ilişkili kümeyi oluşturur.

Örnek:



→ Bu örnekte: "İşçiler, departmanlarda çalışır" denmektedir.

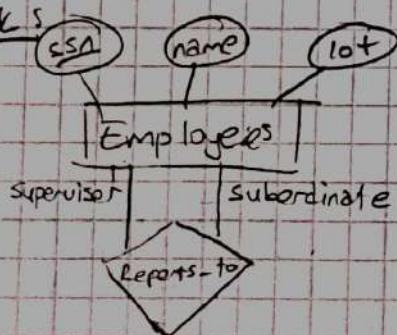
↓                  ↓                  ↓  
varlık            varlık            ilişkili

\* İlişkili kümeleri eşkenar dörtgen ile ifade edilir.

\* İlişkili kümelerin de niteliği vardır. Bu örnekte "ne zamanдан beri çalışır" sorusunun cevabı tutturur 'since' niteliğinde.

İlişkiler aynı türden varlıklar arasında da kurulabilir.

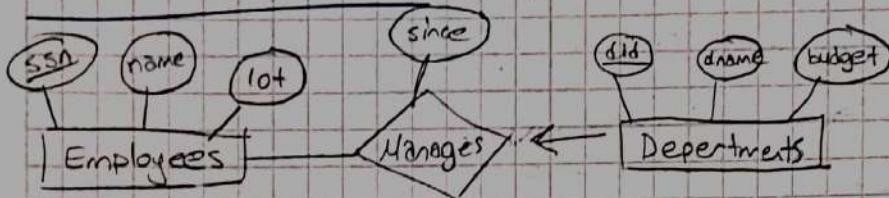
Örnek 2



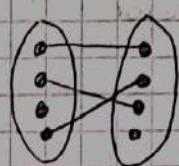
\* Bazı çalışanlar, bazı çalışanlara "Rapor yolları" ilişkisi ile bağlıdır.

\* Bu durumda hangi personelin hangisine rapor verdiğini belirlemek için roller tanımlanır. (Yönetici ve alt çalışan gibi)  
(Supervisor and subordinate)

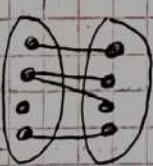
### Anahtar Sınırlaması: (Key Constraints)



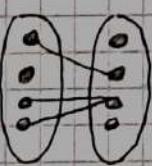
\* Buradaki gibi bir okla su anlatılır: "Departmanlar, manages ilişkisine en fazla 1 kez katılabilir" bu şekilde bir sınırlama, anahtar sınırlamasıdır.



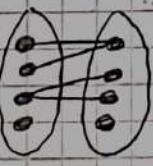
1 -to-1



1 -to-Many



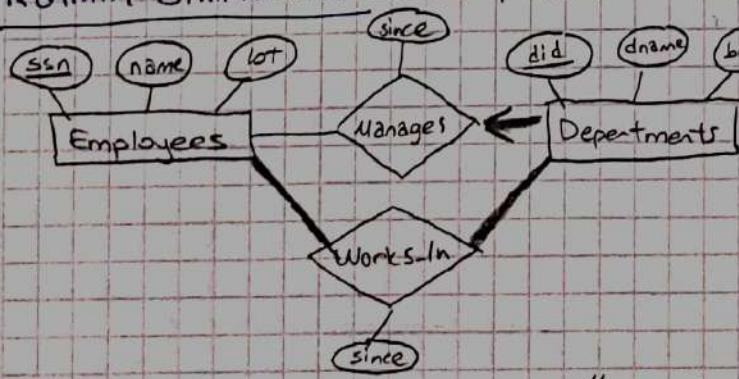
Many -to-1



Many -to -Many

→ Yukarıdaki örnek 1-to-Many relationship'tir. Sol kümeye personeller, sağ kümeye departmanlar olur. Yani her departmanın 1 tane yönetici var. Bir personel birden fazla departman yönetebilir veya bazı personeller hiç departman yönetemeyebilin.

### Katılım Sınırlaması: (Participation Constraints)



\* Koyu renkle gösterilir.

\* Departmanlar ve manages arasındaki koyu çizgi: "Tüm departmanlar manages ilişkisine katılmak zorundadır" anlamına gelir. Burada anahtar sınırlaması da olduğundan tam anlamda sudur:

"Tüm departmanların bir tane yöneticiyi dirmek zorunda."

\* Koyu çizgi yoksa (örneğin Employees ve manages) su anlamadır.

"Bazı çalışanlar manages ilişkisine katılır. Aksi takdirde katılmak zorunda değil"

\* Tüm personeller ve departmanlar works-in ilişkisine katılmak zorunda.

↳ Bir personel en az 22 bir departmanda çalışmak zorunda.

↳ Bir departmana en az 22 bir personeli olmak zorunda.

## Zayıf Varlıklar; (Weak Entities)



\* Birincil anahtarı olmayan varlıklara zayıf varlık denir.

Bu örnekte dependents bir zayıf varlık kümesidir. Çünkü birincil (baskımla yükümlü olunan kişiler, (es, çocuk))

anahtarı yok. Bu örnekte çalışanlar kuvvetli varlıktır. Çalışanların baskımla yükümlü olduğu kişilerin isimleri ve yaşları tutuluyor.

\* Her bir zayıf varlığı ayırt etmek için bir kuvvetli varlıkla arasında ikâfiyî sınırlaması ve anahtar sınırlaması ilişkisi tanımlanmak

zorundadır. Yani her çocuğun bir sağlık sigortası ile sigortalanmalıdır.

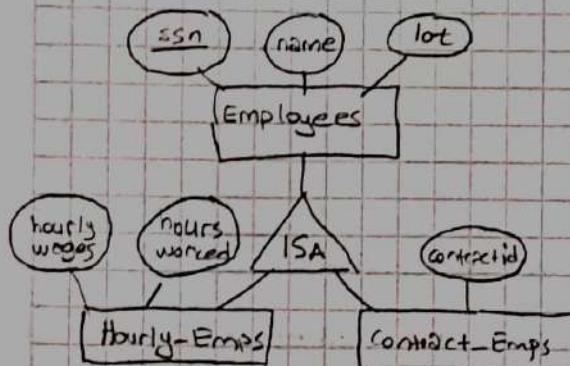
→ Yukarıdaki ER diğramı sunu anlatır :

"Bazı personellerin çocuklar var ve bu çocukların her birisi için bir sağlık sigortası düzenlenmelidir"

\* Policy'nin kayıtlarının nedeni: Zayıf varlığı ayırt eden ilişkileri türnesi olması kaynaklidir.

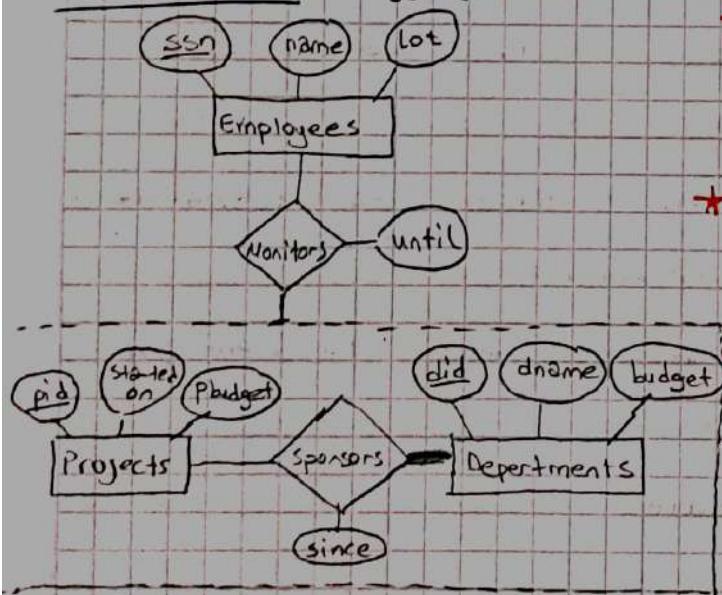
HAFTA XI SON

## ISA (is a) Hiyerarkisi : (ISA ('is a') Hierarchies)



→ ISA, nesneye dayalı programlarındaki kalıtım ilişkisi gibidir. Hourly-Emps → part-time çalışanları, Contract-Emps → kontraktlu ifade eder. Bu iki varlık Employees varlığından, onun özelliklerini miras alır.

## Birleştirmeye : (Aggregation)



\* Bir varlıkla bir ilişki arasında, başka bir ilişki tanımlamak istiyorsak (kısaca iki tane eskenar üçgeni bağlamak istiyorsak) aggregation kullanılır.

\* Aslında aşağıdaki ilişkiyi, kesikli uzaklı dikkatgenbine olarak, bir varılmış gibi gösteriyoruz.

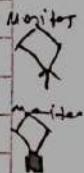
\* Bu örnekte su ifade ediliyor:

Projeler, departmanları tarafından x tarihinden itibaren (since'den dolayı) desteklenir. Her departman en az bir projeyi desteklemek zorunda (katılım sınırlamasından dolayı). Bu destekleme ismini personeller denetler(monitors) y tarihine kadar (until'den dolayı).

\* Monitors ilişkisi direkt olarak sponsors ilişkisiyle temas edemeyeceği için aggregation kullanıldı.

\* Monitors ilişkisine de ilave sınırlamalar verilebilir. Örneğin;

- Her destekleme max 1 kişi tarafından denetlensin. (key constraints)
- Birbirin destekleme işleri denetlemek zorundadır (katılım sınırlaması)
- Kesinlikle bir kez denetlemek zorunda (key + katılım)



!!! \* Bir problem için birden fazla ER tasarımını yapabiliriz.

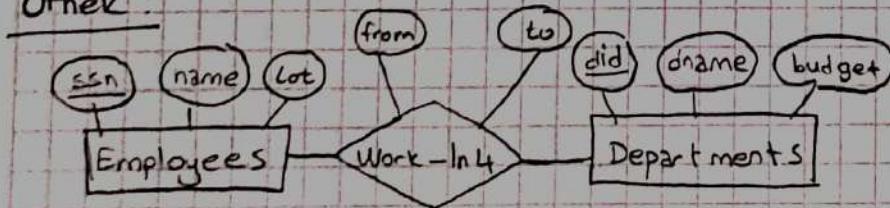
### Varlık vs Nitelik : (Entity vs Attribute)

Bir Lâvrâmin varlık mı yoksa nitelik mi olacağını nasıl karar vereceğiz?

- Bu na bir örnek vermek gerekiyor, personel adres bilgisini DB'de saklamak istiyoruz.
- Eğer, bir personelin çok sayıda adresini (ev adresi, iş adresi vb.) tutmak istiyorsak, adresi varlık olarak tanımlamalıyız. Sonra da adres ve çalışan varlıklar arasında ilişki kurulur.
- Eğer adreste ile göre, ilçeye-göre, mahalleye göre sıralama yapmak isteniyorsa, adresi varlık olarak tanımlayıp il, ilçeye, mahalle bilgilerini de adres varlığının niteliklerini olarak tutmalıyız ve bir ilişki ile gâisen varlığına bağlamalıyız.
- Eğer çalışanın bir adresi varsa ve bu adresle ilgili herhangi bir soru yapılmayacaksız (sadece bilgi olarak tutacağımız), adresi personelin niteliği olarak tanımlarız.

\* Bir personelin birden fazla türde (ev adresi, iş adresi vs) adresi varsa, adres degeri: kesinlikle nitelik olamaz. Çünkü nitelikler atomik (tek, kümeye olmayan) değer alır.

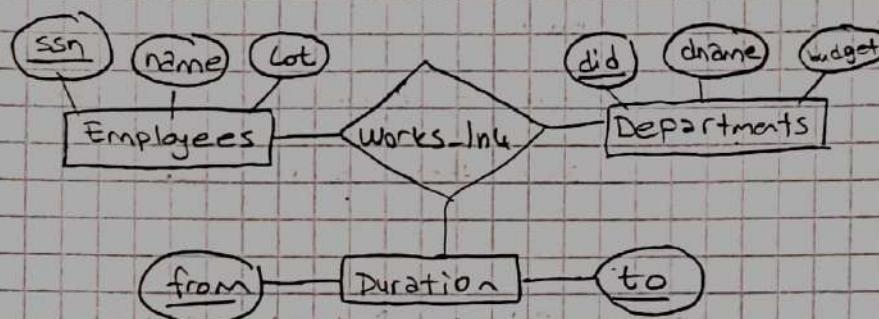
### Örnek .



Ancak bir personel önce X departmanında 3 yıl, sonra Y departmanında 4 yıl ve son olarak Z departmanında 7 yıl çalışmışsa bu diagram yalnızca son çalıştığı departmanında ne kadar süre çalıştığı bilgisini verir.

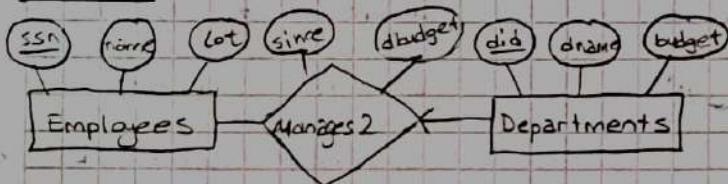
\* Bu ER diagramında bir personelin bir departmanında ne kadar süre çalıştığı bilgisi from ve to yardımıyla tutulmaktadır.

\* Eğer hangi departmanda kaç yıl çalışti, hepsini öğrenmek istiyorsak ER diagraşının şu şekilde düzenleriz.



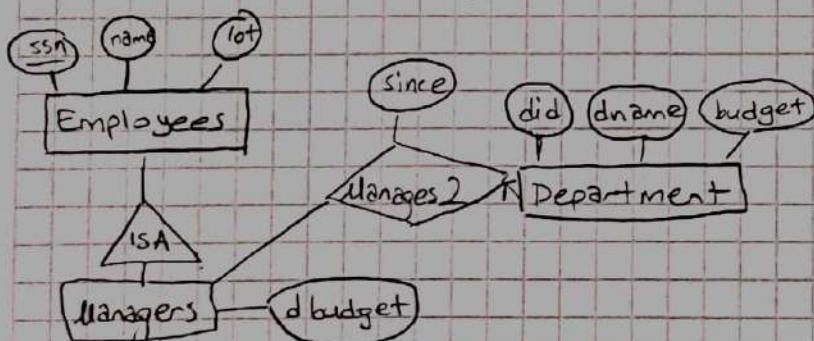
\* Duration yılğı, ile yukarıda istediğimiz bilgilerin hepsini tutabiliyoruz.  
(from ve to = primary keys)

Örnek:



→ Personeller, departmanları yönetir.  
(Her departmanın bir yöneticiisi var)  
(Bir personel herhangi bir sayıda departmanı yönetebilir)

\* Burada bir problem; dbudget niteliği personelin departmanı yönetirken kullanıldığı bütçe mi yoksa bu personelle o departmanı yönetmesi için ayrılmış bütçe mi? Eğer personel birden fazla departman yönetiyorsa her departman için dbudgeti tekrar etmemiz gerekiyor. Bu nü düzeltmek için;

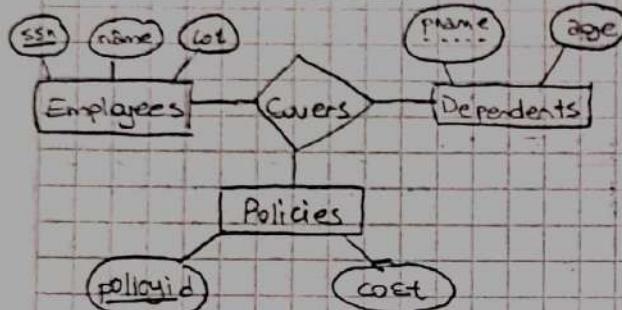


\* Personellerin bazıları yönetici dir. Yöneticilerin toplam bütçe (dbudget) verilmektedir. Yöneticiler, departmanları X tarihinden itibaren yönetir.

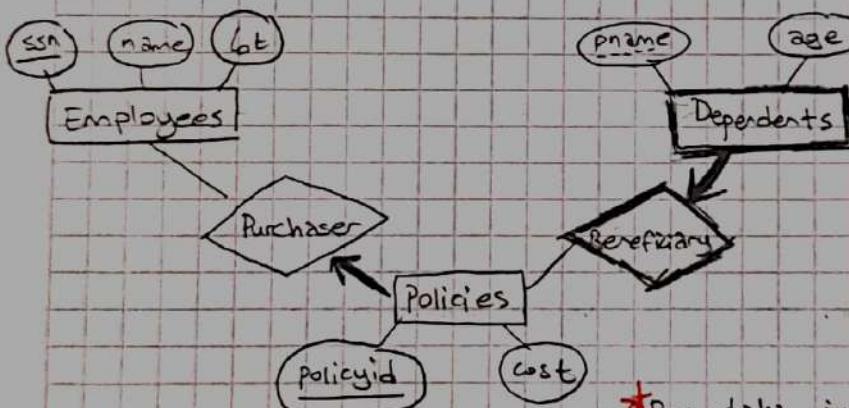
\* Böylece her bir yönetimin sahip olduğu toplam bütçe tutarını bir kez saklıyoruz. Böylece veri tedarikinden kurtuluyoruz.

\* Burada dbudget kişiye özel veriliyor. Kisisi 2'yi yönetmek için 5'ti alıysa bilyi yönetmek içinde 5'tü alır.

## İkili ve Üçlü İlişki :-(Binary vs Ternary Relationships)



→ Çocuklar, uşular ve parıcıeler arasında kapsar (covers) ilişkisi kurulmuştur. Bu, çok genel bir ER diagramıdır. Belirsiz öğeler barındırır. Örneğin burada bir kişiye diğer uşuların ayırtlanmasında konu zagıt varlığı ve keyfi yok.



★ Bu daha iyi bir dizayndır.

→ Personeller polıcıeleri satın alır. (Her bir poliso, bir personel tarafından satın alınmak zorunda). Çocuklar her biri kesinlikle bir policeden faydalannmak zorunda. Bağlılıkla zagıt varlığı policyid kullanarak ayırt edebiliriz.

! ★ Her zaman ikili ilişki, üçlüden daha iyi olacak diye bir kural yok.

Örneğin; departmanlar, tedarikçiler ve parçalarından oluşan varlık kümeleri olsun. Departmanlar bazı parçaları belirli tedarikçilerden belirli miktarlarda satın alıyor olsun. Bunu üçlü ilişkilerde ifade ederiz.

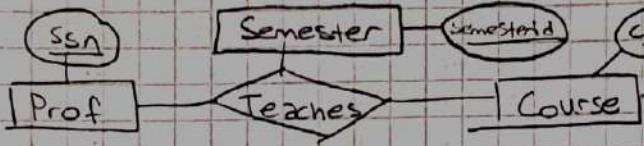
Bunu ikili ilişkilerle ifade etmek çok zor. Hangi departman hangi parçayı hangi tedarikçiden kas tane satın almış bilgisini ifade etmek üçlü ilişkide daha kolaydır.

## Problem :

Bir Üniversite DB'si profesörler(primekey = ssn), ve kurslarından(primekey = courseid) olsun. Profesörler kursu öğretir.

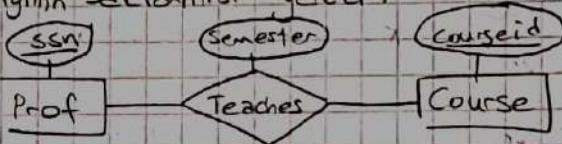
Aralanındaki ilişki için örnekler sun能做到的写在上面:

Örnek 1: Profesörler aynı dersi farklı dönemlerde verebilir ve her dönemde ait bilginin saklanması gereklidir.



\* Semester varlık olarak kuralı türkçe her dönem bilgisinin tutulması istenir. Teacher varlığının bir niteliği olsaydı Semester, sadexe son dönem bilgileri tutulurdu.

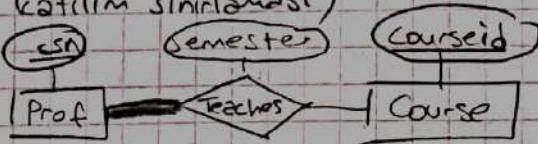
Örnek 2: Profesörler aynı dersi farklı dönemlerde verebilir ve son dönemde ait bilginin saklanması gereklidir.



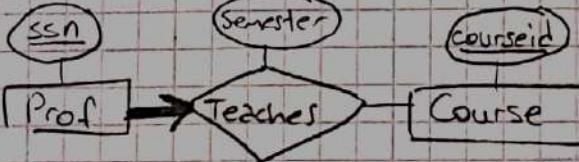
Burda sonraki örneklerde bu örnek baz alınmıştır.

Örnek 3: Her profesör en azından bir tanrı kurs vermek zorundadır (sayı üzerinde sınırlama yok).

(Toplam katılım sınırlaması)



Örnek 4: Her profesör tam olarak 1 ders vermek zorunda (ne 0 ne de birden fazla ders vermez)

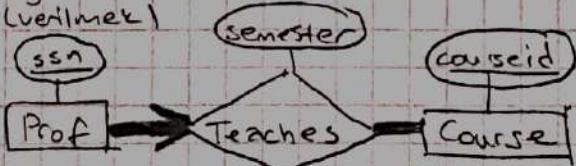


\* Toplam katılım ve en az bir sınırlaması birlikte kullanılmıştır.

\* Bir dersi birden fazla kişi veriyor olabilir veya bazı kurslar verilmeyecektir.

Örnek 5: Her profesör tam olarak 1 ders vermek zorunda ve her bir

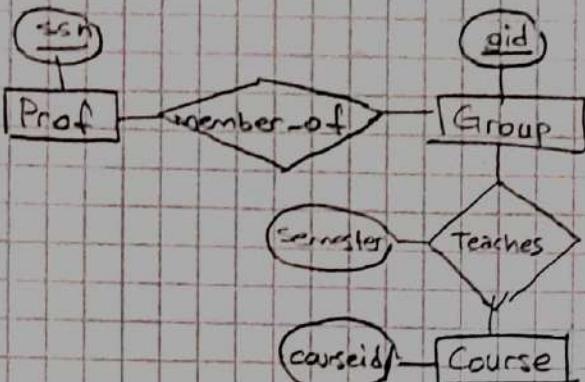
ders öğretilmek zorunda. (her ders teaches ilişkisine katılmak zorunda) (verilmek)



\* her kere katılacak, ile ilgili bir sınırma yok ( $\geq 1$ )

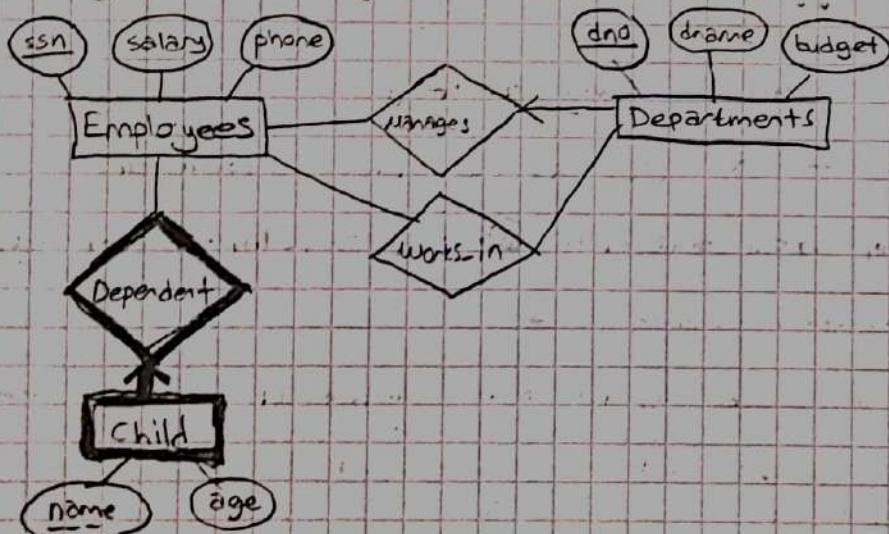
Örnek 6: Belirli dersler belirli bir grup profesör tarafından verilir.

Yani profesör gruplara katılacek gruplar ders verecek.



(Herhangi bir sınırlama yok, toplam  
katılım rekey sınırlaması yok)

Örnek: Bir şirketin DB'sinde personeller (primkey=ssn, salary, phone),  
departmanlar (primkey=dno, dname, budget) ve çocukları (isim, yaşı) vardır.  
Personeller departmanlarda çalışır. Her departman bir personel tarafından  
yönetilir. Bir çalışanın kendi ebeveyni aracılığı ile saptanabilir. (Çocuk zayıf varlığı yani)  
(Ebeveyn şirketten ayrılsa çocukları da DB'den çıkarmalıyız)



Bunu ER diagrame  
gösteremiyoruz.  
SQL sorgusunda  
yazabiliyoruz.

## CHAPTER 3:

### - The Relational Model -

Burada bir varlık-iliski şemasını (ER Model), ilişkisel veritabanına dönüştürüceğiz.

\* satır sayısı = cardinality

Sütun sayısı = degree/arity.

\* İlişki seması, ilişkinin adını ve o ilişkide yer alan niteliklerin ismiyle veri tipini tanımlar. Örneğin;

`Students (sid:string, name:string, login:string, age:integer, gpa:real)`

\* İlişki örneği, ilişki semasının tablo halidir. Örneğin;

sid	name	login	age	gpa	Cardinality = 3. (satır sayısı)
53666	Jones	Jones@cs	18	3.4	Degree = 5 (sütun sayısı)
53688	Smith	Smith@eecs	18	3.2	
53650	Smith	smith@math	19	3.8	* Tekrarlayan satır veya sütun yok!

### İlişkisel Soru Dilleri : (Relational Query Languages)

Veritabanındaki bilgiyi kolay sorular yazarak alabilmemizi sağlarlar.

\* Bu diller içinde en önemli sorulama dili SQL'dir. (Structured Query Language)  
IBM tarafından 1970 yılında geliştirilmiştir.

Örnek: 18 yaşındaki tüm öğrencileri bulmak için veritabanında su soru iletilir.

→ Students tablosuna verilen herka isimde  
(yazılıması zorunlu değil)

• `SELECT * FROM Students S WHERE S.age = 18`

! (Sırasızsaydık buraya de Students yazınca hata gelir)

→ Bunun anlamı: Students tablosundan yaşı 18'e esit olan tüm satırları çek. (Yukarıdaki tabloya uygulansak bu soruyu 1. ve 2. öğrenci gelir)  
 $1 + 2 = 18$

\* Genel olarak söyle ifade edebiliriz: SELECT ile başlayan SQL komutları, FROM cümlesiinde verilen tabloyu input olarak alır. Bu tablo üzerinde

WHERE koşulunu uygun satırları alır. SELECT \* olduğu için sonda uygun satırları tüm sütunlarda alır. (SELECT S.name S.login yazsaydık, tüm sütunlar yerine name ve login sütunları gelirdi)

### Gökku Tablolarda Sorulama : (Querying Multiple Relation)

Students

sid	name	login	age	gpa
53666	Jones	Jones@cs	18	3.4
53688	Smith	smith@eeecs	18	3.2
53650	Smith	smith@math	19	3.8

Enrolled (Kayitli)

sid	cld	grade
53831	Carnatic 101	A
53831	Reggae 203	B
53650	Topology 112	A
53666	History 105	B

1  
↳ S login  
burası C  
amazonus  
değmez.

↳ Bu iki tablo üzerine su soruyu yazarsak :

- `SELECT S.name, E.cld FROM Students S, Enrolled E WHERE S.sid = E.sid AND E.grade = "A"`

↳ Bu soruda; Students ve Enrolled tablolarını alıyor. Student tablolarındaki sid ve Enrolled tablolarındaki sid'leri birbirine eşitlerek bu iki tabloyu birleştirir. Daha sonra birleştirilmiş olan bu tabloda notu "A" olan satırları seçer. Seçilen satırlardan name ve cld sütunlarını alır.

OUTPUT :

\* Yukarıda bahsi geçen birleştirme işlemi su

şekilde yapılır:

→ Students tablolarındaki ilk öğrenciyi alır. Alınan

öğrencinin öğrenci numarasını (sid), Enrolled tablosunda aradı. Enrolled'un son satırında bu sid'yi buldu. Bu işleminden sonra Students'in birinci satırı ve Enrolled'in son satırı (esleştikleri için) birleştirilir.

\* Bu birleştirme su anlama gelir = Numarası (sid) 53666 olan Jones isimli öğrenci History 105 dersine kayıt yaptırmış ve bu dersten 'B' notunu almış.

## - SQL'de Tablo Oluşturma - (Creating Relations in SQL)

- CREATE TABLE Students (sid:CHAR(20), name:CHAR(20), login:CHAR(10), age:INTEGER, gpa:REAL)

↳ İçerisinde, sid, name, login, age ve gpa sütunları olan Students ismindeki (nitelikler) tabloyu oluştur. (Niteliklerin veri tipleri tablo oluşturulurken veriliyor)

\* Bu tabloya yeni bir kayıt eklemek istersen ve belirtilen veri tiplerine uygun (örneğin 13 uzunluklu login) bir veri eklemek istediğimizde DBMS hata verir.

## - Silme ve Değiştirme - (Destroying and Altering Relations)

- DROP TABLE Students

↳ Students tablosunu tamamen siler. (Hem kayıtlar silinir hem de tablonun tanımı silinir)

\* Bunu yazdıktan sonra Students tablosu ile işlem yapmak istenirse hata verir.

- ALTER TABLE Students ADD COLUMN firstYear:integer

↳ Herhangi taboda değişiklik yapmayı sağlar. Örneğin yukarıdaki komut yeni bir nitelik (sütun) ekler. Bir sütun silmek için  $\Rightarrow$  DROP COLUMN kullanılır (ADD COLUMN yerine)

\* Yeni bir sütun eklediğinde, tabloda daha önceden ekli olan verilerin bu yeni niteliği 'null' olarak atanır.

## Yeni Kayıt Ekleme ve Kayıt Silme : (Adding and Deleting Tuples)

- Daha önceki oluşturulan bir tabloya Yeni eklemek için :
- INSERT INTO Students (sid, name, login, age, gpa) VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)
- ★ Eğer eklenecek kaydın sütunları tablodaki sırayla girilecektse (yukarıdaki gibi) o zaman altı çizili içsmi yazmaya gerek yoktur.
- INSERT INTO Students VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)
  
- Tablodan bir kaydı (satırı) silmek için :
- DELETE FROM Students WHERE S.name = 'Smith'
- ↳ Students tablosundan ismi Smith olan kayıtları sil.
- ★ Sadecə DELETE FROM Students yazsaydık, tüm kayıtları silinirdi ama tablo varlığını sürdürürdü. (Bos bir tablo olarak kalır.)

## Bütünlik Sınırlamaları - (Integrity Constraints)

Her kaydın sağlanması gereken koşullardır. Anahtar sınırlaması, genel sınırlamalar ve domain sınırlamaları olmak üzere üç basitçe incelenir.  
(veri tipi sınırlaması)

- ★ Bu sınırlamalar tablo oluşturulurken belirlenir. Deni bir kayıt ekleneceğι veya bir kayıt güncelleneceğι zaman bu kurallar uygulanır.

### Birincil Anahtar Sınırlaması : (Primary Key Constraints)

- \* Anahtar, tabloda bulunan bir kaydı diğer kayıtlardan ayıran, bir ya da birden fazla niteliğin bir örüye gelmesiyle oluşan bir yapıdır.
- \* Primary key, hem her bir kaydı birbirinden ayırt etmeli, hem de primary keyi oluşturan (Eğer birden fazla nitelik varsa) niteliklerin herhangi bir alt kümnesinin bir anahtar özelliği taşımaması gerekiyor. Eğer alt kümnesini aldığımızda bir anahtar özelliği taşıyorsa bunu super key denir.

→ Örneğin;

Öğrenci tablosunda sid bir primary keydir. Öğrenci no yanında gpa'yı ekleyip ikisini birlikte kullandığımızda bu da bir anahtarıdır. Fakat bunun alt kümnesini aldığımızda (mesela sid) sid'de tek başına anahtar olduğu için  $\{sid, gpa\}$  kümlesi superkeydir.

\* Bir tablo için birden fazla anahtar olabilir. Örneğin;

sid, gpa, name, login, age ve T.C kimlik no niteliklerine sahip bir tablo olsun. Bu tablo için primary key sid seçilirse, TC-kimlik no aday (candidate) key olur.

| Candidate olmasını nedeni her öğrenci için T.C kimlik numarasının farklı olması yani primary key olarak kullanılabilecek seviyede olmasıdır.

\* Primary key için, minimal sayıda nitelik içeren ve bir kaydı diğerlerinden ayırmaya yarayan nitelik ya da nitelik kümlesi bulunmaya çalışılır.

## SQL'de Birincil ve Aday Anahtarları. (Primary and Candidate Keys in SQL)

Tablo oluştururken tanımlanır. (Birincil  $\rightarrow$  PRIMARY KEY, aday  $\rightarrow$  UNIQUE)

Örnek:

- CREATE TABLE Enrolled  
(sid CHAR(20),  
cid CHAR(20),  
grade CHAR(2),  
PRIMARY KEY (sid,cid))

sid	cid	grade
53831	Carnatic 101	C
53831	Reggae 203	B
53650	Topology 112	A
53666	History 105	B

\* Bu doğru tanımlanadır. Primary key {sid, cid} ikilisi ile belirlenir. Bu da, bir öğrencinin bir dersে yalnızca bir defa kayıt yaptırabileceğini garanti eder. (Yani bir öğrenci x dersini alıyorsa tekrar kayıt yapmaya çalışığında öğrenci no ve x dersinin cid'si tabloda deha önceden bulunuyorsa {sid, cid} ikilisi seklinde) bu kayıt reddedilir.)

- CREATE TABLE Enrolled  
(sid CHAR(20),  
cid CHAR(20),  
grade CHAR(2),  
PRIMARY KEY (sid),  
UNIQUE (cid, grade))

sid	cid	grade
53831	Carnatic 101	C
53831	Reggae 203	B
53650	Topology 112	A
53666	History 105	B

\* Bu yanlış bir tanımlanadır. Çünkü tanımlı genelğî;  
 → Bir öğrenci sadece bir dersে kayıt yaptırabiliyor. Çünkü tabloda sid kısmı prim olarak tanımlı.  
 → Bir dersten yalnızca bir kişi belirli harf notunu alıyor. Örneğin; yalnızca 53666 nolu öğrenci History 105 dersinden B alabiliyor. Diğerler History 105 dersinden B almıyor çünkü {cid, grade} ikilisi UNIQUE olarak tanımlanmış (TC gibi). Bu nedenle bu ikilinin de kayıtda özel olması gerekiyor.

### Yabancı Anahtar : (Foreign Keys, Referential Integrity )

Bir tablodaki bir ya da birden fazla nitelik, başka bir tabloya atıfta bulunuyorsa bunlara yabancı anahtarlar denir. (Genelde diğer tablonunun birincil anahtarına atıfta bulunur)

- ★ Buna örnek olarak, Enrolled tablosunda sid bulunmasını verebiliriz. Bunu mantığı : öğrenci tablosunda olmayan bir kişinin Enrolled tablosunda olmaması gerekiyor.

Bunu SQL'de şu şekilde ifade ediyoruz: (Tablo oluştururken yazıyoruz)

#### ● CREATE TABLE Enrolled

(sid CHAR(20), cid CHAR(20), grade CHAR(2),

PRIMARY KEY (sid,cid),

FOREIGN KEY (sid) REFERENCES Students )

- ★ Students tablosunda 'sid' sütununun değeri 'xxx' olsaydı söyle yazardık  
FOREIGN KEY (sid) REFERENCES Students (Students.xxx)

- ★ Enrolled tablosuna yeni bir kayıt eklerkeninde sid değerinin Students tablosunda bulunup bulunmadığı kontrol edilir. Bulunmuyorsa eklemez.

- Bunun之外, bir öğrenci kaydını silerken Enrolled tablosundan tüm kayıtları silinebilir. , silinmesine izin verilmey ve bu bilgiler saklanmak istenir ya da silinen öğrencinin Enrolled tablosundaki sid listesine default bir değer yerleştirilebilir.  
↳ (Bunu yapmak Enrolled tablosundaki prim key'i bozabilir)

- \* Tüm bu seçenekleri foreign key tanımında belirtiyoruz.
- Foreign key komutundan sonra silme ve update iin yapılmak istenen sey seçilmelidir:
- Eğer hiçbir sey yazmasa default olarak NO ACTION komutu gelir. Buviso Students tablosundan bir kayıt silmek/güncellmek istendiğinde reddedilir.
- CASCADE, dersen Students tablosunda silinen öğrencinin sahip olduğu sid değerini bulundur satırlar Enrolled tablosundan da silinir. Ya da güncellene yapılsa Enrolled tablosuna da yansır.
- SET NULL /SET DEFAULT, dellirse Students'te silinin güncellenen kaydını, Enrolled tablosundaki sid yerine null veya default değere güncellenir. (Her iki işlem iin de).
- \* Bu seçeneklerden hangisini seçeceğimize dikkatli karar vermeliyiz.

Örnek:

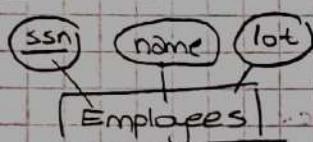
```
CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid),
   FOREIGN KEY (sid) REFERENCES Students
     ON DELETE CASCADE
     ON UPDATE SET DEFAULT )
```

→ Silinirken yapılacak işlem  
→ Güncellirken yapılacak işlem

## ER Modelden Tabloya 1 (Logical DB Design : ER to Relational)

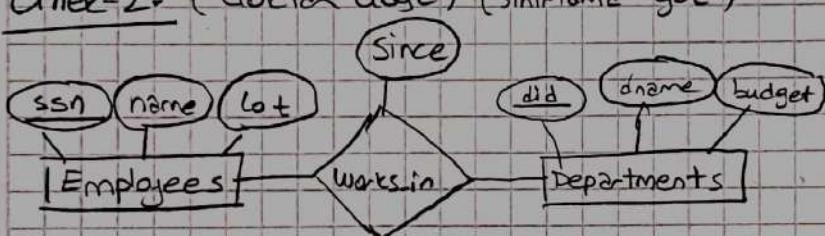
Varlık ilişkisi semasındaki varlıklar, ilişkileri ve sınırlamaları nasıl veritabanı haline getireceğimizi göreceğiz.  
Varlık kümelerinin her biri için bir tablo tanımlanıyor.

### Örnek-1:



CREATE TABLE Employees  
(ssn CHAR(11),  
name CHAR(20),  
lot INTEGER  
PRIMARY KEY (ssn))

### Örnek-2: (Güçten ölçü) (sınırlama yok)



CREATE TABLE Departments  
(did INTEGER,  
dname CHAR(20),  
budget REAL,  
PRIMARY KEY (did))

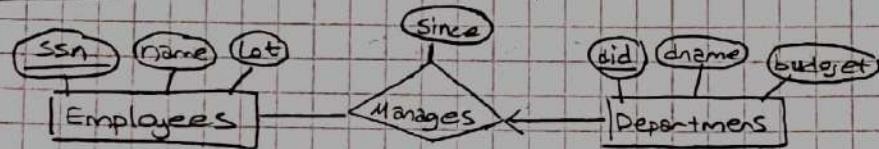
\* Tüm varlıklar tek tek tablolara oluşturuyoruz. (Emp, ve depart... iki oluşturdu)

Bu iki varlık arasındaki ilişki için ayrı bir tablo yapıyoruz. (Çünkü bu ilişki many to many bir ilişkidir. Yani bir personel birden fazla departmanda çalışabilir, bir departmanda birden fazla kişi çalışabilir.)

CREATE TABLE Works-in  
(ssn CHAR(11),  
did INTEGER,  
since DATE,  
PRIMARY KEY (ssn,did)  
FOREIGN KEY (ssn)  
REFERENCES Employees,  
FOREIGN KEY (did)  
REFERENCES Departments)

! many to many  
olduğu için

### Örnek-3: (Anahtar sınırlaması)



Bunu ikinci şekilde yapabiliriz:

- 1-) Her biri için ayrı tablo (Employees ve departments için zaten yapılmıştır)

#### CREATE TABLE Manages

```

(ssn CHAR(11),
did INTEGER,
since DATE,
PRIMARY KEY (did),
FOREIGN KEY (ssn)
REFERENCES Employees,
FOREIGN KEY (did)
REFERENCES Departments)
    
```

★ Key constraint'ı ifade etmek için yaptık bunu  
(anahtar sınırlaması)  
Çünkü her department bir yönetici olabilir.  
Yani did'ler Manages tablosunda tekrar edilmez.

- 2-) iki tabloyu birlestirip tek bir tablo elde edebiliriz (manages+department)

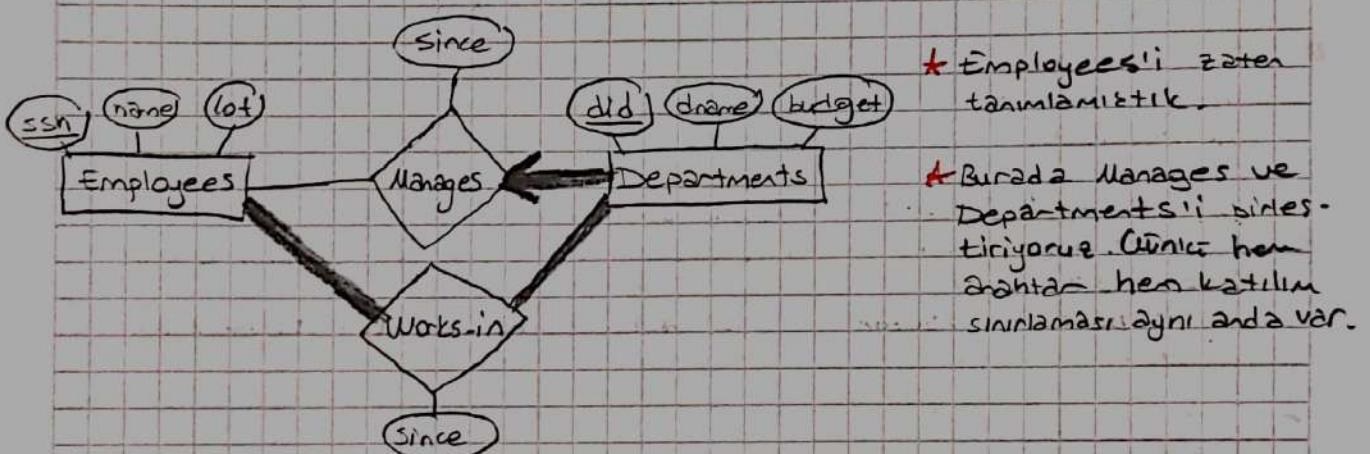
#### CREATE TABLE Dept-Mgr

```

(did INTEGER,
dname CHAR(20),
budget REAL,
ssn CHAR(11),
since DATE,
PRIMARY KEY (did),
FOREIGN KEY (ssn),
REFERENCES Employees)
    
```

### Örnek 4: (Katılım sınırlaması)

- \* Anahat sınırlaması ve katılım sınırlaması aynı anda versa tablolar bireleştirilir.
- \* Sadexe toplam katılım sınırlaması varsa aynı tanımlıyoruz.



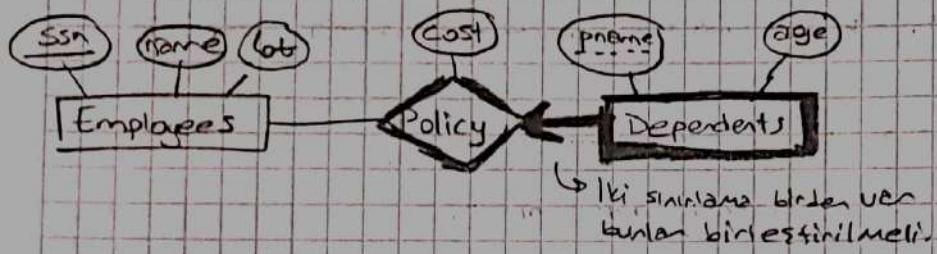
### CREATE TABLE Dept-Mgr

```

  (did INTEGER,
   dname CHAR(20),
   budget REAL,
   ssn CHAR(11) NOT NULL,      → Her departmanın yönetici olmak zorunda
   since DATE,
   PRIMARY KEY (did),         → Her departman 4 kez bu tabloda olabilir.
   FOREIGN KEY (ssn) REFERENCES Employees,
   ON DELETE NO ACTION)       → Bunu yazmalıyız. Yani bir yönetici silindiğinde
                                departmanda oın bilgisi tutulmaya devam
                                edilmeli.
  
```

- \* Son satırda CASCADE yapsaydık yönetici silindiği zaman departmanın tüm bilgileri silinmiş olur. Bu da mantıksız bir durundur.

\* NO ACTION yapınca, örneğin departman yöneticisi i̇sten ayrılaçaksa önce dept-mgr tablosunda o departmanın yeni bir yönetici atanın dahe sonra ayrılacek kişi Employees'ten silinebilir.

Öner-5: (zayıf varlıklar)

## ① CREATE TABLE Dep\_Policy

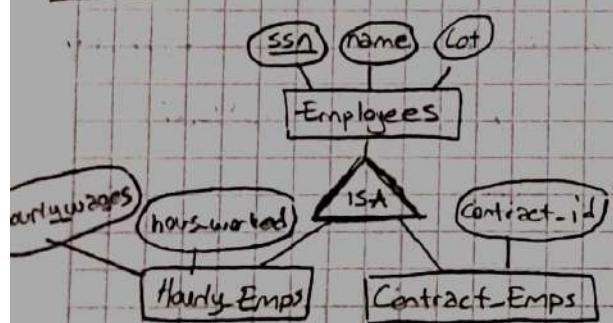
(pname CHAR(20),  
age INTEGER  
cost REAL)

ssn char(11) NOT NULL, → Her çalışan bir ebevnyi olsalı.

PRIMARY KEY (pname, ssn),

FOREIGN KEY (ssn) REFERENCES Employees,

ON DELETE CASCADE ) → Bir çalışan silindiğinde onun ailekânni da DB'den siliyoruz.

Öner-6: (ISA hierarşisi)

Burada iki yöntem var

1-) Her birisi kendi ayrı tablo yapmak

- Employees (ssn, name, lot)

→ foreign key

- Hourly\_Emps (ssn, hourly-wages, hours-worked)

→ foreign key

- Contract\_Emps (ssn, contract-id)

2-) Personeller bu iki seçenekten bir olmak üzere dayasa:

- Hourly\_Emps (ssn, name, lot, hourly-wages, hours-worked)

- Contract\_Emps (ssn, name, lot, contract-id)

HAFTA XII. SON

## Views :

Mantıksal bir tablodur. Fiziksel olarak diskte saklanmaz. Sadece tanımı vardır.

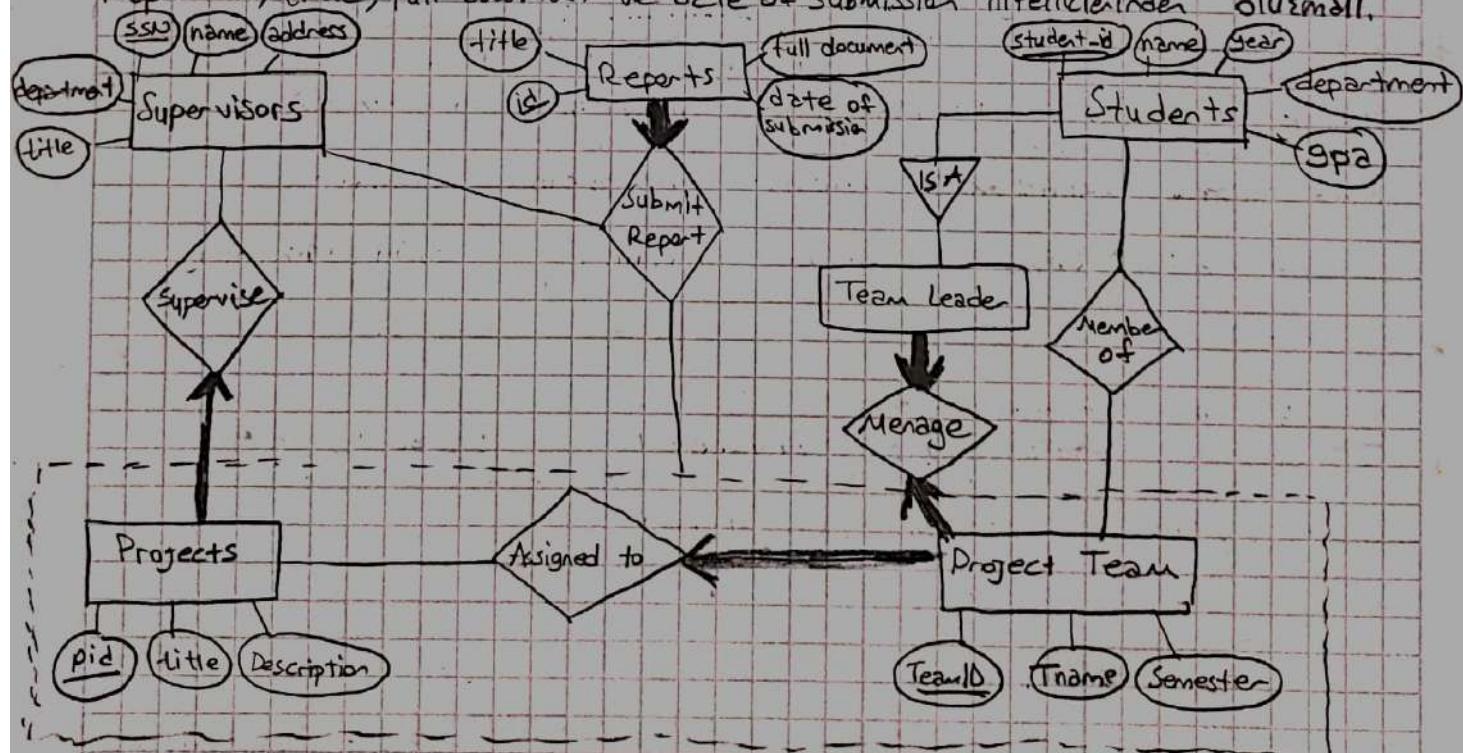
Tablo adı	Nitelikler
CREATE VIEW YoungActive Students (name, grade)	
AS SELECT S.name, E.grade	
FROM Students S,Enrolled E	
WHERE S.sid=E.sid and S.age<21	

→ Bu komutun anlamı: Yası 21'den küçük olan ve Enrolled tablosunda bulunan öğrencilerin isimlerini ve kayıtlı olduğu derslerden aldıkları notları gösterir.

- \* İçindeki SELECT sorgusu sayesinde veriler alınır ve tablo oluşturulur.
- \* Bazı kullanıcılar bu view tablosu üzerinden işlem yetkisi verilirse sadece bu view'daki verileri görebilir. (örneğin yaşı 21'den büyük öğrencileri göremez. Öğrenci id'sini, hangi dersi aldığı vb. göremez)
- \* DROP VIEW tablo\_adi komutu kullanılarak view silinebilir.
- \* Kullanıcılarla belirli özet bilgileri sunmak için bazı bilgileri saklamak, gizlemek için kullanılır.

## "Örnek"

- Bir Mezuniyet Projesi Yönetimi db'si olsun ve bu db'ı Supervisors, Students ve Projects varlıklarından olussun.
- Supervisorlar SSN, name, address, title ve department niteliklerinden olusun.
- Studentlar student-id, name, year, department ve gpa niteliklerinden olusun. Studentslar projects teams'ı yönetir.
- Her Project Team TeamID, team-name ve semester niteliklerine sahiptir. Bazı öğrenciler Team Leader'ıdır. Her Team Leader tam olarak bir Project Team'i yönetir. Her Project Team'in tam olarak bir tane Team Leader'ı vardır.
- Project ler pid, title, description ve supervisor niteliklerine sahiptir. Her Project kesinlikle tam olarak bir tane supervisor'a sahip olmali. Bir supervisor herhangi bir saydakı Project 'i yönetebilir.
- Project 'ler , Project Teams'ıne atanır. Bir Project Team'e tam olarak bir tane Project atanır. ama herhangi bir saydakı Project Teams aynı project'ı yapabilin.
- Project Teams , Project atanaları hakkında bilgileri Supervisor'a rapor eder. Rapor id, title, full document ve date of submission niteliklerinden olumallı.



## DataBase Eserleri :

Supervisors ( SSN, name, address, title, department )

Students ( student-id, name, year, department, gpa )

Projects ( pid, title, description, SSN )

Team Leader ( student-id )

Project Team ( TeamID, Tname, Semester, pid, student-id )

Member of ( student-id, TeamID )

Reports ( id, title, full document, date of submission, TeamID, SSN )

## SQL Komutları :

- CREATE TABLE Students ( student-id CHAR(11), name CHAR(50), year INTEGER, department CHAR(30), gpa REAL, PRIMARY KEY (student-id) )
- CREATE TABLE Supervisors ( SSN CHAR(11), name CHAR(50), address CHAR(100), title CHAR(15), department CHAR(30), PRIMARY KEY (SSN) )
- CREATE TABLE Projects ( pid CHAR(10), title CHAR(100), description CHAR(200), SSN CHAR(11) NOT NULL, PRIMARY KEY (pid), FOREIGN KEY SSN REFERENCES Supervisors ) ★ Her projenin supervisoru olmak zorunda
- CREATE TABLE Team Leader ( student-id CHAR(11), PRIMARY KEY (Student-id), FOREIGN KEY Student-id REFERENCES Students )
- CREATE TABLE Project Team ( TeamID CHAR(10), Tname CHAR(20), Semester CHAR(10), pid CHAR(10) NOT NULL, student\_id CHAR(11) NOT NULL, PRIMARY KEY (TeamID), UNIQUE (student\_id), FOREIGN KEY pid REFERENCES Projects, FOREIGN KEY student\_id REFERENCES TeamLeader )
- ★ Unique olmasının nedeni: Manages ilişkisine one-to-one katılımdır.
- CREATE TABLE Member of ( Student\_id (CHAR (11)), TeamID, CHAR (10), PRIMARY KEY (Student\_id, TeamID), FOREIGN KEY Student\_id REFERENCES Students, FOREIGN KEY TeamID REFERENCES Project Team ) dördüncü ekran
- CREATE TABLE Reports ( id CHAR(6), title CHAR(100), Full document BLOB, Date of submission DATE, TeamID CHAR(10) NOT NULL, SSN CHAR(11), PRIMARY KEY (id), FOREIGN KEY TeamID REFERENCES Project Team, FOREIGN KEY SSN REFERENCES Supervisors )

İPUCU: Each dediği zaman toplam katılım sınırlaması, key Each'ten sonra gelen varlığı.

Exactly one dediği zaman anahtar sınırlaması, key exactly one'dan sonra gelen varlığı.

★ ilişkiye key constr. ile katılım tarafının prim key'i ilişkisinin prim key'i olur. Eğer key const yorsa ikisinin keylerinin birleşiminde key.

## CHAPTER 19:

### - Schema Refinement and Normal Forms —

Bir veritabanı örnmasını nasıl iyileştirebileceğimizi görelereğiz.

Bunu, veri tekrarlarını tespit etme ve bu tekrarların sebebi olduğu problemleri çözme yolu ile yapacağız.

ssn	name	rating	hourly wages	hours worked
123-22-3666	Attishoo	8	10	40
231-31-5368	Smiley	8	10	30
131-24-3650	Smethurst	5	7	30
434-26-3751	Guldu	5	7	32
612-67-4134	Madayan	8	10	40

\* Bu tabloya bakıldığında her bir rating için saat ücretinin sabit olduğunu görürüz. Yani saat ücretlerini tekrar ediyor.

\* Bu tekrar eden verileri belli başlı sıkıntılarla sebebi olur. Dikte gereksiz yer kapları, tabloda silme eklenme, güncelleme yaparken de probleme yol açar.

\* Bu tekrar eden verileri ortadan kaldırmanız gerekiyor.

\* Tekrar eden verileri fonsiyonel bağımlılıkları (functional dependencies) kullanarak ortadan kaldırıyoruz. Fonsiyonel bağımlılıkları ER diagrámında belirleyemiyoruz. Problemin tanımından bunu çıkarıyoruz.

- SSN → name, hours-worked, rating (sosyal güvenlik numarası (ssn) sağıdaki değerleri belirler)
- rating → hourly-wages (rating, hourly-wages'ı belirler)

\* Bu fonsiyonel bağımlılıkları tanımlayarak gereksiz veriye karar verilir ve tabloları parçalara ayıracagız. Örneğin ;  
Yukarıdaki tabloda rating ve hourly-wages sıyrıılır ve ayrı tablo yapılır.

rating	hourly-wages
8	10
5	7

## Fonksiyonel Bağımlılık : (Functional Dependencies)

Fonksiyonel bağımlılık  $X \rightarrow Y$  - ( $X, Y$ 'yi belirler) şeklinde yazdığımız kurallarıdır.

A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1

$AB \rightarrow C$

\* Yukarıdaki gibi bir kural : A ve B ikilisi, C'yi belirler. Yani aynı olan AB ikilileri için C'de aynıdır.

\* Bize verilen bir tablodan fonksiyonel bağımlılığın sağlanıp sağlanmadığını kontrol edebiliriz. Fakat bunu kendimiz tablodan çıkaramayız.

\* Eğer K niteliği / nitellikleri, R tablosu için aday anahtarsa (candidate key)

her zaman juin  $K \rightarrow R$  fonksiyonel bağımlılığı tanımlanabilir. (R = tablodaki tüm nitellikler)

### Örnek:

Geçen sayfaki tablo örneğine bakarsak, (Tablo ya ismiyle ya da niteliklerin ismi ile ifade edilir)

$ssn = S$ ,  $name = N$ ,  $rating = R$ ,  $hourly = W$ ,  $hours = H$  olsun.

$S \rightarrow SNRWH$

$R \rightarrow W$

\* Bu tablo üzerine iki tane fonksiyonel bağımlılık tanımlanmıştır.

\* Fonksiyonel bağımlılıklar üzerine Armstrong Aksiyomları uygulanabilir.

- Reflexivity :  $X \subseteq Y$  ise  $Y \rightarrow X$
- Augmentation :  $X \rightarrow Y$  ise  $XZ \rightarrow YZ$  (herhangi bir Z deðeri için)
- Transitivity :  $X \rightarrow Y$  ve  $Y \rightarrow Z$  ise  $X \rightarrow Z$
- Union :  $X \rightarrow Y$  ve  $X \rightarrow Z$  ise  $X \rightarrow YZ$
- Decomposition :  $X \rightarrow YZ$  ise  $X \rightarrow Y$ ,  $X \rightarrow Z$ .

\* Tüm durumları (bu kuralları deneiktten sonra) fonksiyonel bağımlılıkların kumesine  $F^+$  denir.

- \* Burada belli problemler vardır. Örneğin 1. kaydın (rating=8 olan) hourly-wages değerini 11 yaparsak, ratingi 8 olan diğer elemanları için sonuc olusur. Bu 'probleme' update anomaly denir.
- \* Rating değeri 7 olan bir kişiyi eklemek için onun hourly-wages değerini de biliyor olmam lazımdır ki tabloda tutarsızlık olmasın. Buna da insertion anomaly denir.
- \* Ratingli 5 olan tüm personelleri tablodan silseyd eger Rating 5'in hourly-wages karşılığı 7'dir bilgisini de kaybetmiş oluruz. Buna da deletion anomaly denir
- \* Tüm bu sorunları ortadan kaldırmak için önceki sayfalarda yaptığı gibi tekrar eden veriler için yeni bir tablo oluşturuyoruz. Ana tabloya sadece Rating değerini ekliyoruz. Böylece update yapacağımız zaman sonraki oluşturduğumuz tabloda tek bir değişiklik yaparak rating ve hourly-wages değerlerini tüm çalışanlar için erit hale getiriyoruz. Aynı zamanda insertion ve deletion anomaly'de çözülmüş.

HAFTA XIII. SON

Örnek: Contracts ( $\underline{c}_id, \underline{s}_id, \underline{j}_id, \underline{d}_id, \underline{p}_id, g_id, value$ )

$C \rightarrow CSJDPQV$ ,  $JP \rightarrow C$ ,  $SD \rightarrow P$  olsun. Bu durumda.

→ Transitivity kullanarak :  $JP \rightarrow CSJDPQV$

→ Augmentation kullanarak :  $SD \rightarrow P$  ise  $SD \rightarrow JP$

→ ilk eşitliği kullanarak :  $SD \rightarrow CSJDPQV$

\* Buaksiyonları kullanarak çok sayıda fonk. bğm. elde edilir, bunların kümeleri de  $F^+$ 'dır.

\*  $F^+$ 'yi hesaplamak oldukça maliyetlidir. Bu yüzden  $X \rightarrow Y$  gibi bir fonk. bğmliliğin  $F^+$  da olup olmadığını anlamak için  $X^+$ 'ya bakıyoruz. Eğer  $X^+$ 'nın içinde  $Y$  varsa,  $X \rightarrow Y$  fonk. bğm  $F^+$  kümelerinde vardır. denir.

Örnek:  $F = \{A \rightarrow B, B \rightarrow C, CD \rightarrow E\}$  ise  $F^+$  içinde  $A \rightarrow E$  var mı? (Yani A, E'yi belirler mi?)

\* Bunun kolay yolu yukarıda da belirtildiği gibi  $A^+$ 'ya bakıp burada E olup olmadığını gözlemeaktır.

$$A^+ = \{A\} \quad (1)$$

$$\{AB\} \quad (2)$$

$$\{ABC\} \quad (3)$$

\*  $A^+$  bulma işlemi şu şekilde yapılır:

Once A'nın kendisiyle başlar (1)

Kurallarda sol tarafta sadexe A olan kuralların sağ tarafını da yazıyoruz (2). Şimdi içine

AB oldu AB'nin herhangi bir alt kümelerinin veya kendisinin kuralın solunda olduğu bir fonk. bğmlilik var mı diye bakılır. (A, B, AB sequençlerine bakılır). A kuralını alındığımız için tekrar almıyoruz. B kuralını alıyoruz ve C'yi.

kümeye dahil ediyoruz. (3) \* ABC'inin herhangi bir alt kümeliği solda bulunmuyor. Bu yüzden kümeye böyle kalar. Son halde sağ tarafta E olmadığı için,  $A \rightarrow E$  fonk. bğm.  $F^+$  da yoktur deniz.

→  $AD \rightarrow E$   $f^+$  da var mı?

$$AD^+ = \{AD\}$$

$$= \{AD, DB\}$$

$$\{ADBC\}$$

$$\{ADBCE\}$$

\* CD alt kümeli solda ola kural var E yi belirliyor onu da yazdık.

(Evet)

\* AD ikilisi aranmaktadır. (Tümki tüm elementleri belirliyor. Ama A aranır değildir. Tüm elementleri belirleyemez.)

- \* Yaptığımız  $A^+$ ,  $AD^+$  ... bulma işlemine 'attribute closure' denir. Bu sayede primary key veya superkey bulabiliriz. (Bulduğumuz anahtarın herhangi bir alt kümesi primary key değilse yalnızca kendisi primary key ise bu anahtar primary keydir. Herhangi bir alt kümeli primary key ise bu anahtar superkeydir)
- \* Eğer tabloda hiç fonk bağımlılığı yoksa o tabloda hiç tekrar eden veri yoktur deniz.
- \*  $A \rightarrow B$  gibi bir bağımlılık için A'nın değerinin aynı olduğu yerlerde B'nin değerinin de aynı olması beklenir.
- \* Eğer bu fonk bağımlılığını kullanarak tablonun Boyce-Codd Normal Formu (BCNF) veya üçüncü normal formda olduğunu garanti edersek problem çözüleceğinden garanti ediyoruz.

### Boyce-Codd Normal Form (BCNF) :

- Bir R tablosu üzerinde,  $F^+$  da yer alan  $X \rightarrow A$  şeklindeki tüm fonk. bağımlılıkları:
- $A \in X$  ( $A$ ,  $X$ 'in elemanıdır)
  - $X$ , R için anahtardır.
- } Bu iki kuraldan biri sağlanıysa  
bu tablo BCNF idir. Dolayısıyla  
insert, update, delete anomali sorunları olmaz.

### Third Normal Form (3NF):

Bir R tablosunda,  $F^+$  'da yer alan tüm  $X \rightarrow A$  şeklindeki fonk. bğml. iin :

- $A \in X$  ( $A, X$ 'in elemanıdır)

} Buna herhangi birini sağlama gerekiyor

- $X, R$  iin anahtardır.

- $A, R$  iin bazı anahtarın alt kümelerdir  
 (Primitif) (Parçasıdır)  
 (Superprimitif)

\* Sorunlu tabloları önce BCNF 'e dönüştürmeye çalışınız. Eğer olmazsa 3NF'le dönüştürmeye çalışınız.

\* Bir tablo 3NF olsa bile yine bir veri tekrarı olabilir.

Örnek: Reserves tablosunda SBD nitelikleri olsun ve

bu tablo üzerine tanımlı  $S \rightarrow C$ ,  $C \rightarrow S$  fonk. bğml. olsun.

- $S$  tek başına bu tablo iin anahtar değildir. Çünkü  $S^+$  yi alırsak tüm elemanları kapsamadığını görürüz. (Kosul-2 sağlanmadı)

- $S, C$ 'nin alt kümeleri değildir. (Kosul-1 sağlanmadı)

- Bu tablo iin anahtar  $\underline{SBD}$  olur. (bu anahtarın bir alt kümeli de  $\underline{\text{birinci fonk. bğm. dolayı}}$ )

Fakat ikinci fonk. bğm. dolayı anahtar  $\underline{BC'D}$  de anahtar olur.

$C'$  de bu anahtarın bir parçası olduğu iin (2. fonk. bğml.) (Kosul-3 sağlanır)

\* Tablo 3NF olmasına rağmen veri tekrarı vardır. Fakat bu veri tekrarı insert, update, delete anomaly'ine yol aymaz.

### Tabloyu ayırmak : (Decomposition of a Relational Schema)

n elementli bir tabloda verilen tüm fonk. bağımlılık için (örneğin 3NF'ye dönüştürmek istiyorsak) 3NF'ye göre bu fonk. bağımlılıkları sağlanıp sağlanmadığını kontrol ediyoruz.

Örnek :  $S \rightarrow^{\text{I}} SNLRWHT$  nitelikleri olan tabloda  $S \rightarrow^{\text{II}} SNLRWHT$

$R \rightarrow W$  fonk. bağımlılık verilmiş olsun.

- I. fonk. bağımlılık 3NF'ye uyar mı? ✓

Evet, çünkü S, anahtardır.

- II. fonk. bağımlılık 3NF'ye uyar mı? X

Hayır. Çünkü W, R'nin alt kümeleri değil. R anahtar değil. W, herhangi bir primary key'in parçası değil.

↳ Böyle bir durumda, 3NF'ye uymayan fonk. bağımlılığının oluşturduğu niteliklerden (RW) bir tablo yapıyoruz. Daha sonra bu fonk. bağımlılığının sağindaki niteliği (W), nitelik kümelerinden cıkarıp geriye kalanlarla başka bir tablo yapıyoruz. Yani son durumda elimizde 2 tablo var. Birinin özellikleri SNLRH, diğerinin özellikleri RW.  
(Galiba örnekindeki gibi)

\* Ayırma işleminden sonra, 3NF'ye uymayan tablo varsa sorunlu tablo bölmeye devam edilir.

! Bir tabloyu daha küçük parçalara böldüğümüzde su sorunlarla karşılaşabiliriz:

- (Galison örneğinde yola ukalım (SNLRH ve RW tabloları var))
- Galison maaşlarını hesaplayacağımız zaman  $H^*W$  isteminin sonucunu bulmamız gerekiyor. Dolayısıyla bu hesaplamayı yapabilmek için bu iki tabloyu birleştirmemiz gerekiyor. Bu da bir maliyettin.
- Ana tabloyu böldükten sonra, parça tablolar tekrar birleştirildiğinde ana tablo ile tutarlı olması lazımlı. Bazı bölünmelerde veri kaybı olabilir. (Galison örneği için böyle bir durum yoktur.)
- Bir tabloyu böldüğümüz zaman bazı fonk. bağıml. sağlanıp sağlanmadığını takip etmek için küçük tabloları birleştirmek gerekebilir. (Galison örneği için böyle bir durum yok)

### Lossless Join Decomposition: (Kayıpsız birleşim ayrımı):

Eğer bir tabloyu bölüp oluşan yeni tablolar tekrar birleştirildiğinde ana tablo elde ediliyorsa bu decomposition'a lossless join decomposition adı verilir (ayrımcıma).

Örnek:

Ana tablo			A	B	A	B	C
A	B	C	1	2	1	2	3
4	5	6	4	5	4	5	6
7	2	8	7	2	7	2	8
					1	2	8
					7	2	3

\* Bu örnekte ana tabloyu bölüp tekrar join yaptığımızda son tabloyu elde ediyoruz. Son tablo, ana tablo ile aynı olmadığından bu bölümme ve birleşme lossless join decom. değildir.

\* Lossless join decom olması için: R tablosu olsun, ( $X$  ve  $Y$  nitelikleri var) Böldük  $X$  ve  $Y$  tablosu olarak.

$X \cap Y \rightarrow X$  veya  $X \cap Y \rightarrow Y$  (fonk. bağımlı).  $F^*$ 'nin içinde olmalı.

Örnek:

A	B	C
1	3	4
2	5	9

Ana tablo

$$F = \{A \rightarrow B, B \rightarrow C\}$$

(I)                  (II)

Verilmiş olsun. Fonksiyonel bağımlılıklar BCNF'yi sağlıyor mu?

- I. fonk. bağımlı. BCNF'yi sağlar. Çünkü A keydir.

$$A^+ = \{A\}$$

$$\{AB\}$$

$\{ABC\} \rightarrow$  Tüm elementleri kapsıyor.

- II. fonk. bağımlılığı BCNF'yi sağlamaz. Çünkü B key değildir,

$$B^+ = \{B\}$$

$\{BC\} \rightarrow$  tüm elementleri içermez.

ve C, B'nin elementi değildir.

→ BCNF'yi sağlamadığı için tabloyu bölyyoruz. Bölerken BCNF'yi sağlanan fonk. bağımlılığının elementleri birleştirilir. (BC) ve sağıdaki element ana tablodan silinir. (Yani AB ve BC tablolar olur.)

A	B	C
1	3	4
2	5	9

A	B
1	3
2	5

B	C
3	4
5	9

? Bu bölünme lossless join decomposition midir?

→ Bunu bulmak için  $\{AB\} \cap \{BC\}$  işlemi yapılır. Sonuç:  $\{B\}$

→ Eğer 'B', AB'yi veya BC'yi belirliyorsa. lossless join deco. deniz.

Bunun için  $B^+$ 'yi buluyoruz:

$$B^+ = \{B\}$$

$$\{BC\} \checkmark$$

\* Bu tablo bölünmesi lossless join decompositiondur.

## Dependency Preserving Decomposition:

Bir R tablosunu X ve Y niteliklerinden oluşan iki tabloya böldüğümüzde X tablosu üzerinde tanımlı olan fonk. bağımlılıklarla, Y tablosu üzerinde tanımlı fonk. bağımlılıkların birleşiminin closure'ını aldığımda  $F^+$ 'ya eşit oluyorsa bu bir dependency preserving decompositiondur.

$$(F_X \cup F_Y)^+ = F^+$$

Örnek: CSJDPQV nitelikli tabloda C kaydır ve  $JP \rightarrow C \quad SD \rightarrow P$  fonk. bağımlılık verilmiştir.

BCNF decom. yaparsak: SJ DPQV ve SDP tabloları elde edilir.

Bunların fonk. bağımlılıklarının birleşiminin closure'sı alınırsa  $JP \rightarrow C$ 'yi iyermez dolayısıyla dependency preserving decomp. degildir.

Örnek: ABC niteliklerinden oluşan bir tabloda  $A \rightarrow B$ ,  $B \rightarrow C$ ,  $C \rightarrow A$  fonk. bağıml. tanımlı olsun. Digelimki bu tabloyu AB ve BC diye böldük.

? Bu bölünme dependency preserving mi?

→ AB tablosunun fonk. bağıml. kumesi içinde yalnızca A ve B geçen kurallardan BC " " " " " " B ve C geçen kurallardan olur.

\* Dolayısıyla AB tablosu I. fonk. bağımlılığı, BC ise II. fonk. bağımlılığı kapsar. Bu durumda  $C \rightarrow A$  hatta kodluğunu için cevap HAYIR olur.

? Bu bölünme lossless decomp mu?

$$\{AB\} \cap \{BC\} = \{B\}$$

Eğer B  $AB$ 'yi veya  $BC$ 'yi belirlerse kabul.

Buna göre  $B^+ = \{B\}$

$$\begin{array}{c} \{BC\} \\ \{BCA\} \end{array}$$

Bölünme tablolar.

\* Cevabımız EVET.

## BCNF'te Bölümme : (Decomposition into BCNF)

Bir tabloyu BCNF'ye çevirirken tablo üzerindeki fonk. bağımlılıkları tek tek bâzıyonuz. BCNF'e uygun fonk. bağımlılıkları bulunduğu zaman o fonk bağımlılıklarının nitelikleri kullanarak bir tane tablo yapıyoruz.

Örnek : Tabloda CJS DPQV nitelikleri olsun. C anahtarıdır. ( $C \Rightarrow^I CJS DPQV$ ),  $JP \Rightarrow^I C$ ,  $SD \Rightarrow^I P$ ,  $J \Rightarrow^I S$  fonk. bağımlılıkları tanımlansın.

- I. Kuralda herhangi bir sorun yok (BCNF'ye uygun)
- II. kuralda  $JP$ 'de bir anahtarı asılarda burada da sorun yok. ( $JP \Rightarrow^I C$  den dolayı.)
- III. kural BCNF'i sağlamaz. Çünkü:

$SD$  anahtar değildir ve  $P$ ,  $SD$ 'nin elemanı değildir.  
(closure'ı alınarak bulunur)

$$SD^+ = \{ SD \}$$

$$\{ SD \} \rightarrow \text{anahtar değil.}$$

- IV. kural BCNF'ii sağlamaz. Çünkü:

$J$  tek başına anahtar değildir ve  $S$ ,  $J$ 'nın alt kümlesi değildir.

$$J^+ = \{ J \}$$

$$\{ J \} \rightarrow \text{anahtar değil. (S'nin tek başına solda olduğu kural yok!)} \\ \text{ondan } SD \rightarrow \text{e kuralını dahil etmedik.}$$

\* BCNF': sağlanamayan fonk. bağımlılıkları herhangi birini alıyoruz (örn:  $SD \Rightarrow P$ )

Bu fonk. bağımlılık nitelikler için bir tablo yapılır. ( $SDP$  için). Kuralın

sagindakı eleman ana tablodan çıkarılır. Sonuca elimizde

$CJS DPQV$  ve  $SDP$  tabloları olusur. Daha sonra tekrar tüm

kurallara baktır. (I ve II'de sorun yoktu). III'de de sorun yok (P tablodan yok artık)

Kalmaz çünkü  $SD$  artı ikinci tablonun keyidir. IV'de sorun var.

$$\rightarrow SD^+ = \{ SD \}$$

$$\{ SDP \} \rightarrow \text{tüm ikinci tablo elemanları} \\ \text{var. Dolayısıyla keyidir.}$$

IV. fonk. bağımlı sorun: ) anatara değil:  $J^+ = \{J\}$  ve  $S^+$   
ITS

\* Anatara olmas için bolumuz ana tablodaki tüm elementleri  
içermeliydi (CJDQV) (P olmayan tablo)

143

\* IV fonk. bağımlılıktaki sorunu gözlemek için ana tabloyu bir kez deha bölgüyoruz. J ve S'yi birlestir (JS), S'yi ana tablodan at. Sonuç olarak elimizde: CJDQV, SPP ve JS tabloları var.

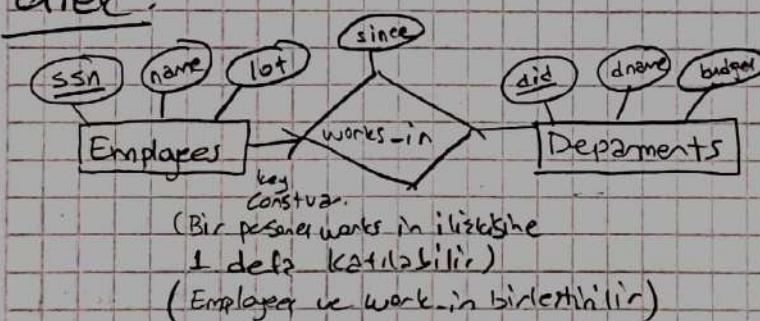
\* Bu yöntemle yapılan bölünme her zaman lossless join decomposition. Farak dependency preserving olma garantisı yoktur. (Zaten bu örnekte de II. kuralı elde edemediğimiz için dependency preserving değildir)

\* 3NF'e dönüştürmek de BCNF'e dönüştürmek gibidir.

\* Fonk. bağımlı ER diagramlarının iyileştirilmesinde de kullanılır.

(empire way'in lidesini)

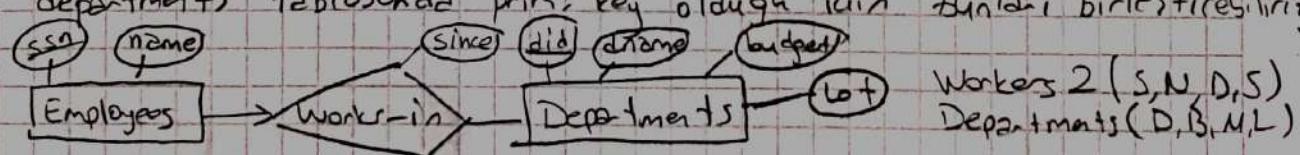
Örnek:



Workers (S, N, L, D, S)  
Departments (D, M, B)

Örneğin:  $D \rightarrow L$  diye bir sınırlama olsun. (Aynı departman daliği çalışanlar aynı vendiyede olacak)

! D 2. fonk. bağımlılık BCNF veya 3NF'e uymaz. Dolayısıyla bu tabloyu ikiye böleriz. (SNDs ve DL diye). D niteliği yeni oluştur tabloda ve departments tablosunda prim. key olduğundan bunları birlestirebiliriz.



Workers 2 (S, N, D, S)  
Departments (D, B, M, L)

HAFTA XIV. SON

HAFTA XV. SON

DÖNEM

SONU



DATABASE

MANAGEMENT

SYSTEMS

Furkan EKİCİ - 2017 555 017

146

Ders islemmedi

### HAFTA I. SON

Sayfa 102 - 116 arasını isledi

### HAFTA II. SON

Sayfa 117 - 131 arasını isledi

### HAFTA III. SON

## CHAPTER 4:

### - Relational Algebra -

Bu chapterda ilişkisel veritabanları üzerinde tanımlı sorulama dillerinden bahsederceğiz. İki tane matematiksel sorulama dili vardır.

- Relational Algebra: Soruların nasıl yazılacağı, hangi islemlerin yapılacağı ile ilgilenir.

- Relational Calculus: Soru sonucunda görmek istediğimiz voi ile ilgilenir.

\* Soru, tablo örnekleri üzerinde calısır. Soru da input (tablo veya tablolar) ve output (tablo) vardır.

Basit islemler:  $\sigma$ ,  $\pi$

• Selection ( $\sigma$ ): Input olarak bir tablo alır. Bu tablodan belirli kriterlere uygun satırları seçer.

• Projection ( $\pi$ ): Belirlen satırları ekran da gösterir.

• Cross-product (Kartezyen Çarpım - X): İki tabloyu birleştirerek kullanılır.

• Set-difference (Küme farkı - -): Birinci tabloda olan ama ikinci tabloda olmayan kayıtları gösterir.

• Union (Birlesme - U): Her iki tabloyu da birleştirir.

• Kesişim, Katılım, Bölme  
(intersection, join, division)

## Projection : ( $\pi$ )

Belirtilen sütunları tablodan alır.

Örnek :

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

S2  
tablosu

table  
ismi  
 $\pi_{sname, rating} (S2)$

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

- INPUT -

- OUTPUT -

\* Bu iskende tekrarlıyan satırlar varsa, tekrar eden satırlar elenir. Örneğin:

$\pi_{age} (S2)$

age
35.0
55.5

→ Tekrarlıyan veriler silindi.

## Selection : ( $\sigma$ )

Verilen koşula uygun satırları alır.

Örnek: Yine yukarıdaki S2 tablosu üzerinde soru yazıyoruz.

$\sigma_{rating > 8} (S2)$

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

\* Soru sonucu şart sağlayan verilerin tüm sütunları gelir.

Örnek: İki operatörü birlesdirebiliriz.

$\pi_{sname, rating} (\sigma_{rating > 8} (S2))$

sname	rating
yuppy	9
rusty	10

\* "rating" > 8'den büyük olan verilerin yalnızca adını ve rating'ini göster.

! Önce projection, sonra selection da yapabiliirdik. Bu durumda projection ile tablodan alınan sütunlara Selection uygulanabiliriz. Aksi halle hata alırız.

## Union - Intersection - Set Difference ( $U \cap -$ )

\* Birleşim, kesim ve fark islemlerini yapmak için iki tablo gereklidir. Bu tabloların union-compatible olması gereklidir. Yani tabloların sütun sayıları birbirine eşit olmalı ve sütunların veri tiplerinin uyumlu olması gereklidir.

Örnek :

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S1

sid	sname	rating	age
28	guppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

S2

S1 $\cup$ S2 → (union)	sid	sname	rating	age
22	dustin	7	45.0	
31	lubber	8	55.5	
58	rusty	10	35.0	
44	guppy	5	35.0	
28	guppy	9	35.0	

\* Tetrar eden veriler + koz yağılırlar.

S1 $\cap$ S2 → (intersec)	sid	sname	rating	age
31	lubber	8	55.5	
58	rusty	10	35.0	

S1-S2

S1-S2	sid	sname	rating	age
22	dustin	7	45.0	

## Cross-Product (X)

iki tabloyu birleştirmek için kullanılan bir işlemidir. Kartezyen çarpım yaparken, tabloların sütun sayısı ve bunların veritipleri arasında benzerlik olmasına gereklidir. Sonuçta oluşan tablodada her iki tablonun da bütün sütunları görünür. Satır sayısı, tablo1'in satır sayısı  $\times$  tablo2'nin satır sayısı şeklinde dir. Kartezyen çarpımda aynı isme sahip olan sütunlar bulunabilir. Çünkü bunlar ayrı tablolardan geliyor ve herpsi gösteriliyor sütunlarını.

Örnek: Gelen sayıdakileri S1, R1 baz al.

sid	bid	day
22	101	10/10/96
58	103	11/12/96

R1

(S1)		(R1)				
(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8.	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

\* S1'in her satır, R1'in her satıyla eşleştirildi.

## Join ( $\bowtie$ )

Kartezyen çarpımın bir türüdür. Kartezyen çarpımı üzerine bir selection işlemi uygularsak bu aslında join işlemidir.

- Eğer selection işleminde küçükten, büyüğün, eşit değil gibi ifadeler kullanılıyorsa buna condition join denir.

Örnek:

S1  $\bowtie$  S1.sid < R1.sid R1

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

\* Önce iki tablo kartezyen çarpımı tabii tutıldı.

Daha sonra R1.sid değeri S1.sid değerinden büyük olan veriler getildi.

- Eğer selection işleminde beliri bir özelliği (veya özellikleri) eşit olan ifadeler arıysak buna equi-join denir.

Örnek:

S1  $\bowtie$  bid R1

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	11/12/96

\* Verilen özelliklerin aynı olan satırları birleştirilir.

\* sid özelliğinin birisi de olmalıdır.

tümünün

- İki tabloda ismi aynı olan sütunlar varsa, bunların değerlerini eşitlemeye birleştirmeye natural join denir. (S1 R1 şeklinde gösteriliyor.)

## Division : ( / ) (Tanel operatör degidir)

iki tablo arasında olur. Bu tablolardan arasında en az bir tane ortak sütun olmalıdır.

\* içinde all gelen sorularda kullanılır.

Örnek :

A		B1		B2			
Sno	Pno	Pno	P1	Pno	P1	P2	P4
S1	P1						
S1	P2		P1				
S1	P3		P2				
S1	P4		P3				
S2	P1			Pno	P1		
S2	P2			P2	P2		
S3	P2			P4	P4		
S4	P2						
S6	P4						

Sno
S1
S2
S3
S4

Sno
S1
S4

Sno
S1

\* Burada mantık hatalı  
B1 tablosundaki elementler  
a karşılık gelen S değerleri  
A tablosunda var mı?  
Örneğin A/B2 için B2  
tablosundaki P2 ve P4'e  
aynı anda karşılık  
gelen elementler var mı?  
Evet S1 ve S4.  
Örneğin A/B3 için B3  
tablosundaki P1, P2 ve  
P4'e A tablosunda  
aynı anda karşılık  
gelen element var mı?  
Evet S1.

\* Diğer operatörleri kullanarak division elde edebiliri :-

istenmeyen sno değerleri için :  $\Pi_{\text{Sno}} ((\Pi_{\text{Sno}} (\text{A}) \times \text{B}) - \text{A})$

A/B :  $\Pi_{\text{Sno}} (\text{A}) - \text{istenmeyen sno değerleri}$

Örnekler :

sid	bid	day
22	101	10/10/96
58	103	11/12/96

R1  
(Reserves)

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S1  
(Sailors)

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

B1  
(Boats)

→ 1-) 103 bid'li botu reserve etmis sailor'u bul.

Cözüm 1 :  $\Pi_{\text{sname}} ((\underbrace{\sigma_{\text{bid}=103} \text{Reserves}}_{58 \ 103 \ 11/12/96} \bowtie \text{Sailors})) = \underline{\text{rusty}}$

58 103 11/12/96 → Jini ile sid'leri eşit olanlar kaldırıldı  
yani R1 ve S1'in son satırlarının birleşimi

Cözüm 2 :  $\Pi_{\text{sname}} (\sigma_{\text{bid}=103} (\text{Reserves} \bowtie \text{Sailors}))$

→ 2-) Kırmızı botları reserve eden kışilerin isimleri.

$\Pi_{\text{name}} ((O_{\text{color}=\text{"red"} \text{Boats}}) \bowtie \text{Reserves} \bowtie \text{Sailors})$

Kırmızı botları bul

Kırmızı botları reserve edenlerin id'sini  
buluyoruz.

Bu id'lerle Sailors tablosundan verileri getiyoruz.

En sonunda ise bu kışilerin isimlerini çekiyoruz.

→ 3-) Kırmızı veya yeşil renkli botları reserve eden kışilerin isimleri.

\* P operatörü ile output ita elde edilen tabloya isim verebiliriz.

p (TempBoats, ( $O_{\text{color}=\text{"red"} \text{Boats}}$ )  $\vee$   $O_{\text{color}=\text{"green"} \text{Boats}}$ )

$\Pi_{\text{name}} ((\text{TempBoats} \bowtie \text{Reserves}) \bowtie \text{Sailors})$

→ 4-) Kırmızı ve yeşil renkli botları reserve eden kışilerin isimleri.

p (TempRed,  $\Pi_{\text{id}} ((O_{\text{color}=\text{"red"} \text{Boats}}) \bowtie \text{Reserves})$ )

→ Kırmızı reserve edenlerin id'si

: p (TempGreen,  $\Pi_{\text{id}} ((O_{\text{color}=\text{"green"} \text{Boats}}) \bowtie \text{Reserves})$ )

→ Yeşil reserve edenlerin id'si

$\Pi_{\text{name}} ((\text{TempRed} \wedge \text{TempGreen}) \bowtie \text{Sailors})$

→ Buların kesişimi

→ 5-) Tüm botları reserve eden kışilerin isimleri.

p (Tempids, ( $\Pi_{\text{id}, \text{id}} \text{Reserves}$ ) / ( $\Pi_{\text{id}} \text{Boats}$ ))

$\Pi_{\text{name}} (\text{Tempids} \bowtie \text{Sailors})$

———— HAFTA IV. SON ————

## CHAPTER 5:

### - SQL : Queries, Constraints, Triggers -

Gelen haftaki örneği kullanıyoruz.

- Sailors (sid, sname, rating, age)

- Boats (bid, bname, color)

- Reserves (sid, bid, day)

- ↳ Reserves tablosunun üç niteliği birlesip primary key olduğu için;  
 → aynı kişi aynı botu farklı tarihte kiralayabiliyor.  
 Eğer sid ve bid primary key olsaydı;  
 → aynı kişi aynı botu yalnızca bir defa kiralayabilir. (tarih değişse bire)  
 Eğer sadece sid primary key olsaydı  
 → Bir kişi yalnızca bir bot kiralayabiliirdi. (Baska birbir zaman bot kiralayabilir)

#### Basit SQL Sorgusu :

SELECT      bösterilecek satırlar  
 FROM        tablo isimleri  
 WHERE        koşul

\* Bu bir select sorgusudur. Buradaki select, from ve where komutlarının sırası değiştirilemez.  
 \* Sonuç sonucu oluşan tabloda tekrar eden satırlar olabilir. Bunları elenek için SELECT komutundan sonra DISTINCT ifadesini kullanıyoruz.

\* Bu sorguda öncelikle FROM cümlesiındaki tablolardan kartezyen çarpımı alınır. Bu kartezyen çarpım üzerinde WHERE cümlesiındaki koşul kontrol edilir. Bu koşulu sağlayan satırlar alınır. En son, SELECT cümlesiındaki sütun isimleri WHERE sonucunda oluşan tablodan alınarak doldurulur.

Örnek: Sayfa 151'deki Örnekler kısmında bulunan tablolar kullanıldı

SELECT S.sname  
 FROM Sailors S,Reserves R  
 WHERE S.sid=R.sid AND R.bid=103

\* Sailors ve Reserves tablosu kartezyen çarpma tabii tutulur. Koşulu sağlayan satırların sname sütunu doldurulur.

(Bu sorgu 103 numaralı botu kirayan denizci(lerin) isim(lerini) doldurur)

sname
Rusty

SONUÇ :

SELECT S.sname  
 FROM Sailors S, Reserves R  
 WHERE S.sid = R.sid AND bid = 103 }  
 { SELECT sname  
 FROM Sailors, Reserves  
 WHERE Sailors.sid = Reserves.sid  
 AND bid = 103  
 V  
 (Tercih)

\* Bu iki soru da aynı işi yapar. Fakat 1. kullanım tercih edilir.

Günük from cümlesinde tablolara kisa isim verilecek bu isimlerin where cümlesinde kullanırı sağlarız. Böylece daha az yazı yazız.  
Aynı zamanda okunabilirlikte artar.

Örnek: En az bir bot rezerve etmiş kişilerin id'leri.

SELECT S.sid  
 FROM Sailors S, Reserves R  
 WHERE S.sid = R.sid } ? SELECT 'ten sonra DISTINCT  
 yazarsak sonucu değişir mi?

Cevap: Evet sonucu değişir. Bir kişi 5 bot rezerve ettiğe yukarıdaki soru sonunda 5 kere sid'si yazır. ama DISTINCT kullanırsak yalnızca 1 defa görünür.

! S.sid yerine S.sname yazsaydık yanılıtıcı bir sonuc alabilirdik. Örneğin 2 ayrı denizcinin aynı isme sahip olduğunu düşünelim. DISTINCT ifadesi iki tane aynı isim gördüğünü için bunları birlestirecek ama aslında bunlar farklı kişilerdi.

Aritmetik işlemler ve Stringler:

SELECT S.age, age1 = S.age - 5, 2 \* S.age AS age2  
 FROM Sailors S  
 WHERE S.name LIKE 'B-%B'  
 Aritmetik işlemler  
 yalnızca select cümleinde  
 kullanılabilir. Soru sonucu  
 oluştururan sütuna isim vermek  
 için = veya AS kullanılır.

\* WHERE cümleinde işe bir düzenli ifade görüyoruz (RegEx). Bu düzenli ifadeye uygun isimlerin yaş bilgisi, yaşının 5 eksigi ve yaşının iki katı sütunları (yani toplamda 3 sütun) oluşturuluyor.

age	age1	age2
:	:	:

Örnek 1 Kırmızı veya yeşil renkli botları reserve eden kişilerin sid'leri.

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
AND (B.color='red' OR B.color='green')
```

} Kırmızı ve yeşil olanları reserve eden deseydik soldaki soruda kırmızı yine AND koysaydık sonuc, boş dönerdi(bir bot her yeşil hem kırmızı olamaz.) Bu yüzden aşağıdaki gibi yazıp etmeniz gereklidir.

\* Bu soruyu aşağıdaki gibi UNION ile de yapabiliriz

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
AND B.color='red'
```

} Kırmızı renkli botlar

**UNION**

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
AND B.color='green'
```

} Yeşil renkli botlar.

\* Birleşim, kesim, fark işlemi yaparken iki select sorusunu sonucunda (union) (intersect) (except) oluşan tablolardan aynı sayıda sütunlardan oluşması gerekiyor. (Bu örnekte her iki soru sonucu da tek sütunlu tablolardır.)

İç içe Sorgular:

SQL'in genel özellikleriinden biridir. Bir select sorusunun içine başka bir select sorusu yazabilirisiz. İç içe soru parantez içinde olmalı ve where, from, having cümlelerine yazılmalıdır.

Örnek 1 103 numaralı botu reserve eden kişilerin sname'leri.

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R
                WHERE R.bid=103)
```

} Sayfa 153 en alttaki örnek ile aynı işi yapar.

} İç Sorguda: 103 numaralı botu reserve eden kişilerin sid'leri döndürülür.

\* Dış Sorguda: Dene sid değerine karşılık gelen isimlerden yeni tablo oluşturulur.

→ 103 numaralı botu reserve etmeyen kişileri bulmak için NOT IN yaz.

**★** Örnek Relational Algebra ve SQL iin hafta 5'in videolarına bak. Ders dosyalarından da pdf'yi bulabilirsin.

156

Örnek : 103 numaralı botu reserve eden kisilerin isimleri. (Farklı bir yöntem)

```
SELECT S.sname  
FROM Sailors S  
WHERE EXISTS (SELECT *  
               FROM Reserves R  
               WHERE R.bid = 103 AND S.sid = R.sid)
```

\* içeriği tablo boş dönmeyece yani bir sonuc döndürürse exists'den dolayı burası true olur ve dönen tablodan sname seçilebilir.

\* UNIQUE kelimesi ise, tekrar eden satır yoksa true döner.

(\* yerine R.bid , EXISTS yerine UNIQUE yazarak 103 nolu botu 1 kez reserve etmiş kisilerin isimleri döner)

Kümə Operatörleri :

- IN, EXISTS, UNIQUE → kümə içindeki eşleştirme yapmak için kullanılır (NOT ile de kullanılır. NOT EXISTS... gibi)
- ANY, ALL, IN operatörleri  $>$ ,  $<$ ,  $=$ ,  $\geq$ ,  $\leq$ ,  $\neq$  ifadeleri ile kullanılır.

Örnek : ismi Horatio olan kisilerin rating'lerinin herhangi birinden daha büyük ratinge sahip kisilerin tüm bilgilerini göster.

```
SELECT *  
FROM Sailors S  
WHERE S.rating > ANY (SELECT S2.rating  
                      FROM Sailors S2  
                      WHERE S2.name = 'Horatio')
```

\* içeriği ve distati sorğuda aynı tablo farklı isimle kullanılmış.

\* ALL kullanısaydık içeriği tablodan dönen rating değerlerinin hepsinden büyük olan ratinglere sahip kisilerin bilgileri döndürülecekti.

\* Benzer bir mantıkla kırmızı ve yeşil botları, reserve eden kisileri de iç içe sorğu ile bulabiliriz. (INTERSECT kullanmadan)

HAFTA VI. SON

## SQL'de Bâlme

★ Bâlme istemi için sayıla 151'e git.

Önek: Tüm bottarı reserve etmiş kişilerin isimleri.

I. İsteme

• `SELECT S.sname  
FROM Sailors S  
WHERE NOT EXISTS  
((SELECT B.bid  
FROM Boats B)  
EXCEPT  
(SELECT R.bid  
FROM Reserves R  
WHERE R.sid = S.sid))`

→ Tüm bot id'lerden o kişinin reserve ettiği bot id'lerini utarıyoruz. Eğer böyle bir bot id yoksa bu kişi bütün bottarı reserve etmiştir.

(Bu döştekenmeyen)

II. İsteme

• `SELECT S.sname  
FROM Sailors S  
WHERE NOT EXISTS (SELECT B.bid  
FROM Boats B  
WHERE NOT EXISTS (SELECT R.bid  
FROM Reserves R  
WHERE R.bid = B.bid  
AND R.sid = S.sid))`

## Aggregate Operators

★ Bu operatörler sadece SELECT cümlesi veya HAVING cümlesi içinde kullanılır.

Bu operatörler:

`COUNT(*)` → Satırları sayı. (distinct ile kullanırsa teteşanın satırlandırır)  
`COUNT([DISTINCT] A)` → Tek sütundaki satır sayısı.  
`SUM([DISTINCT] A)` → Sütunun toplamı.  
`AVG([DISTINCT] A)` → Sütunun ortalaması.  
`MIN(A)` → Sütundaki min değer.  
`MAX(A)` → Sütundaki max değer.)

Bu işlemler sonucunda 1x1 bir tablo döner.

A: Tekbir sütunu ifade eder.

Önek:

`SELECT S.sname  
FROM Sailors S  
WHERE S.rating = (SELECT MAX(S2.rating)  
FROM Sailors S2)`

★ Sailor tablosunda ratingi en büyük olan kişinin ismini verin.

Örnek: En yaşlı denizcinin ismi ve yaşı nedir?

SELECT S.sname, MAX(S.age)  
FROM Sailors S

Bu sorgu  
X Dönen  
değer  
hatalıdır.

sname	age
dustin	55.5
lubber	55.5
rusty	55.5

SELECT S.sname, S.age  
FROM Sailors S  
WHERE S.age =  
(SELECT MAX(S2.age)  
FROM Sailors S2)

Bu sorgu  
Dönen  
değer  
dogrudur

sname	age
lubber	55.5

↳ Önce en fazla yaşı bul sonra bu yaşa ait kişiyi bul

★ Yukarıdaki sorguyu yazarken  $S.age = (\text{SELECT} \dots)$  olmalı.  $(\text{SELECT} \dots) = S.age$  olursa hata verir.

### Gruplama :

SELECT gösterilecek - sütunlar  
FROM tablolari.  
WHERE koşul  
GROUP BY sütun(lar)  
HAVING grup koşulu.

} From cümlesindeki tablolari kartesinden  
seçipini alınır. WHERE cümlesindeki şartı  
göre SELECT cümlesindeki sütunları alınır.  
★ Talmazca GROUP BY cümlesindeki sütunlar direkt  
olarak SELECT'e yazılabilir. SELECT tek  
diğer sütunlar aggregate operatörleri ile yazıllır.

Örnek: Yaşı 18'den büyük olan ve rating değeri en az 2 kez tekrarlanmış  
olan ratinglendeli en küçük denizciyi bul.

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zarba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frado	3	25.5

SELECT S.rating, MIN(S.age) AS minage  
FROM Sailors S  
WHERE S.age >= 18  
GROUP BY S.rating  
HAVING COUNT(\*) > 1

rating	minage
3	25.5
7	35.0
8	25.5

★ Sailors tablosundan yaş 18'den büyük olanları aldı.  
Bunları rating değerlerine göre gruplandı. Ege aynı  
ratingden en az iki kişi varsa bunları seçti.  
Bu seviyen grupların içindeki en küçük bireyler ve ratingler  
döndürildi.  
★ ratingi 10 olan iki eleman var. Fakat  
zarbanın yaşı 16 olduğu için where  
cümlesi içinde eleman ve aynıratingi kaldırılmış. (Slaytta güzel anlatmış gibi)

★ ratingi 10 olan iki eleman var. Fakat  
zarbanın yaşı 16 olduğu için where  
cümlesi içinde eleman ve aynıratingi kaldırılmış.

### \* having cümlesine :

HAVING COUNT(\*) > 1 AND EVERY (S.age <= 60)

yazsaydık dönen tablo.

rating	minage
7	35.0
8	25.5

Burada saylenen sayı & gruptaki her elemanın yaşının 60'tan küçük olması gereklidir. rating = 3 için 'bob' bu durumu doğrudan tabloya rating = 3 dahil ettilmedi.

Eğer EVERY yerine ANY yazsaydık herhangi bir elemanın ratingi 60'tan küçükse o rating grubunu alır. Bu durumda rating = 3 te tabloda yer almır.

### \* where cümlesine

WHERE S.age >= 18 AND S.age <= 60

yazsaydık dönen tablo

rating	minage
3	25.5
7	35.0
8	25.5

tob en başta eleindi ama ratingi = 3 olan ikisi de aynı yaşta eleman var. Bütün de ratingi en az iki kez tekrar edebili istiyordu.

Örnek : Kirmizi renkli botları reserve eden kaç kişi vardır? (Tüm kirmizi botlar ayrı ayrı.)

SELECT B.bid, COUNT(\*), AS scount

FROM Sailors S, Boats B, Reserves R

WHERE S.sid = r.sid AND R.bid = B.bid AND B.color = 'red'

GROUP BY B.bid

→ Renk kırmızı olan botları al ve bot id'ye göre grupta. Sonra her bir grupta kaç eleman olduğunu say.

Örnek : Her rating grubundaki kişileri say eger (yaşı > 16 olan) 1 kişide fazla eleman varsa bunlardan kaçının ratingi ve yaşını döndür. (Yani en az iki kişi olasılık ratingdeki min yaşa sahip olanı döndür.)

SELECT S.rating, MIN(S.age)

FROM Sailors S

WHERE S.age > 18

GROUP BY S.rating

HAVING 1 < (SELECT COUNT(\*)

FROM Sailors S2

WHERE S.rating = S2.rating)

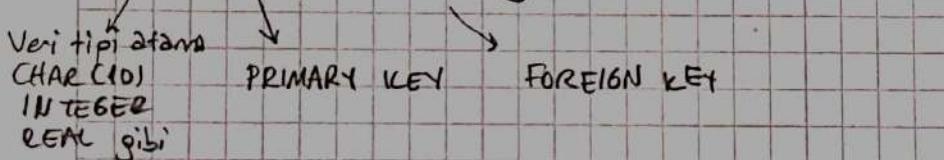
Örnek: Her ratingin ortalamasını bul ve ortalaması en düşük olan ratingi dönden.

- ~~SELECT S.rating  
FROM Sailor S  
WHERE S.age = (SELECT MIN(AVG(S2.age)), FROM Sailors S2)~~
- ~~SELECT Temp.rating, Temp.avgage  
FROM (SELECT S.rating, AVG(S.age) AS avgage  
FROM Sailor S  
GROUP BY S.rating) AS Temp  
WHERE Temp.avgage = (SELECT MIN(Temp.avgage)  
FROM Temp)~~

Aggregate operatörler  
için ike kullanılmaz.

### Sınırlamalar

Domain, primary ve foreign key sınırlamalarını gördük. Şimdi genel sınırlamaları görelim.



### Örnek :

```
CREATE TABLE Sailors (
    sid INTEGER,
    sname CHAR(10),
    rating INTEGER,
    age REAL,
    PRIMARY KEY (sid),
    CHECK (rating >= 1 AND rating <= 10))
```

\* rating değerinin 1-10 aralığında  
bir integer olduğu garanti edilir.

```
CREATE TABLE Reserves (
    sname CHAR(10),
    bid INTEGER,
    day DATE,
    PRIMARY KEY (bid, day, sname),
    CONSTRAINT noInterlake Ries,
    CHECK ('Interlake' <>
        (SELECT B.bname
        FROM Boats B
        WHERE B.bid = bid)))
```

\* 'Interlake' olmamalı diğer reserve edilemez. Constraint direkt türkçe isim verilmiştir.

\* Birden fazla tabloyu aynı anda iletilendiren sınırlamalar yazmak istersen bunu ASSERTION olarak yazabiliriz.

Örnek: Bot ve denizcilerin toplamı < 100 olsun.

CREATE ASSERTION smallClub  
CHECK

((SELECT COUNT (S.sid) FROM Sailors S)  
+ (SELECT COUNT (B.bid) FROM Boats B)) < 100 )

### Trigger

Belli koşullar gerçekleştiği sonra taze söylem otomatik olarak çalışmasını isteyebiliriz. Bu gibi durumlarda kullanılin Üç kısımdan oluşan

- Event (Triggeri çalıştıracak olay)
- Condition (Bu olayın olma şartı)
- Action (Trigger çalışınca ne olacak)

Örnek:

CREATE TRIGGER young Sailor Update  
AFTER INSERT ON SAILORS  
REFERENCING NEW TABLE New Sailors  
FOR EACH STATEMENT

INSERT

INTO Young Sailors (sid, name, age, rating)  
SELECT Sid, name, age, rating  
FROM NewSailors N  
WHERE N.age <= 18

→ Event

→ Action

(Kosul yok)

Sailors kısmına eklenme yapılması tetiklenecek. İni eklenen kayıttta yaş değeri <= 18 ise bu kaydı young Sailors adındaki diğer tabloya da ekliyor.

HAFTA VI. SON

## Join

- Inner Join iki tablo arasında belirtilen şartlara uygun satırları birleştirir.

SELECT employee.LastName, employee.DepartmentID, department.DepartmentName

FROM employee INNER JOIN department ON

employee.DepartmentID = department.DepartmentID

→ Bu komut aslında aşağıdaki ile aynırıdır

Join koşulu  
(Bu komutta, =, <, >, obilir)

SELECT E.LastName, E.DepartmentID, D.DepartmentName

FROM employee E, department D

WHERE E.DepartmentID = D.DepartmentID

- Equi Join iki tablo arasında belirtilen ve esit olan değerleri birleştirir.

SELECT \*

FROM employee JOIN department ON employee.DepartmentID=department.DepartmentID

SELECT \*

FROM employee, department

WHERE employee.DepartmentID=department.DepartmentID

SELECT \*

FROM employee INNER JOIN department USING (DepartmentID)

→ Bu üç soru da aynı işi yapar. Aslında innerjoinin eşitlik olan hâlindir.

- Natural Join iki tabloyu birleştirir ve ortak sütunları 1 defa yazar.  
(esit sütunları)

SELECT \*

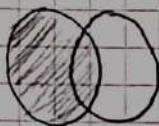
FROM employee NATURAL JOIN department

Normalde employee ve department tablosundaki her iki DepartmentID

değeri de sonuc tabloda gösterilmeli ama natural join

oldugu için tek değer gösterilir.

- ④ Left Outer Join iki tabloyu join yaparken soldaki tablonun tüm kayıtlarını alır. Soldaki tabloda sağdaki ile birleşmeyecek satırlar varsa yanında NULL yazılır.



- ④ Right Outer Join

- ④ Full Outer Join

- ④ Cross Join Kartezyen çarpım.

---

HAFTA VII SON

---

VİZE

---

Hafta VIII. SON

---

## CHAPTER 12:

### -Query Evaluation -

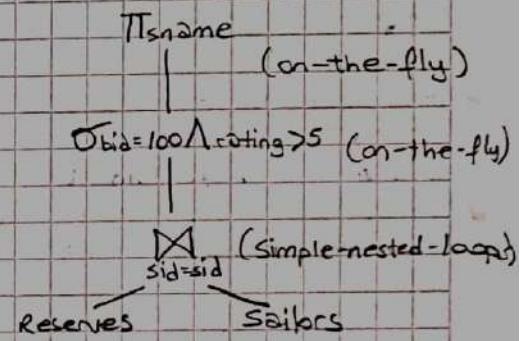
Bu chapter'da, bir sorgunun nasıl hesaplandığını ve bu hesaplama göre maliyetinin nasıl tahmin edildiğini öğreneceğiz.

### Sorgu Planı :

Verilen bir SQL sorgusunun ilişkisel cebir kullanarak bir ağaca dönüştürülmesi.  
Bu ağacın her adımında kullanılan ilişkisel cebir fonksiyonu gösterilir.

→ Simple-nested-loops algoritması equi-join işlemi yapar.

→ on-the-fly : Bir önceki işlem hesaplandıktan sonra (materialized yazsaydı bir önceki işlemin sonucu diske yazılır ve sonraki işlem bunu disetten okurdu)



### -Sorgu Planı -

```

SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid
      AND R.bid = 100 AND S.rating > 5
  
```

\* Query Optimizer, sorguyu ualıstırırken hangi algoritmaları kullanacağına karar verir.

→ Query Optimization (sorgu optimizasyonu) için iki husus var : hangi planlar göz önüne alınacak ve her bir planın maliyeti nasıl tahmin edilecek.

\* Bir sorgudaki ilişkisel cebir işlemlerini daha düşük maliyetle genelleştirmek için = Indexing, Iteration, Partitioning yöntemlerinden biri veya birkaç kullanılır.

Indexing: Genelde selection ve join işlemlerinde tercih edilir.

Iteration: Tam kayıtları tek tek taramak.

Partitioning: Kayıtları belirli parçalara ayıranak üzerinde ilişkisel cebi uygulamak.

### Maliyet Hesaplama Gereksinimleri

- Kataloglar -

- Her bir tablo için : Tablodaki kayıt sayısı, tablonun sayfa sayısı

- Her bir index için : Indexteki farklı anahtar sayısı, indexin sayfa sayısı

- Her bir ağacı index için : Ağacın yüksekliği, indexte bulunan min-max değerli anahtarlar.

→ Bu değerlere göre sorgu planının maliyeti hesaplanır ve bu planda hangi algoritmaların tercih edileceği belirlenir.

\* Kataloglar her değişiklikten sonra değil belirli periyotlarda güncellenir.

### Erişim Yolu (Access Path)

Kayıtların diskten hangi yöntem ile alınacağına access path denir.

→ Dosyayı baştan sona tarayarak kayıtlara erişilebilir. (file scan)

→ Bir index kullanarakta kayıtlara erişilebilir. Böyle bir durumda

indexin türü de önemli. Örneğin ; bir ağacı indexi kullanırsak ve

bu indexteki anahtar  $\langle a, b, c \rangle$  niteliklerinden oluşuyorsa secim

islemi yalnız  $a$ 'nın olduğu,  $a$  ve  $b$ 'nın olduğu veya  $a, b, c$ 'nin birlikte

olduğu kriterleri ararken kullanılabiliriz. (Yani  $a=5 \text{ AND } b=3$ ,  $a=5 \text{ AND } b>6$

gibi sorgular yazılabilirken  $b=3$  sorgusu tek başına yazılamaz.)

Örneğin ; hash index varsa sadece  $a, b, c$  niteliklerinin olduğu eşitlik

sorguları yazılabilir. (Yani  $a=5 \text{ AND } b=3 \text{ AND } c=7$  yazılabilirken,

$a=5 \text{ AND } b=3$ ,  $b=3$ ,  $a>5 \text{ AND } b=3 \text{ AND } c=7$  gibi sorgular yazılmaz)

### Karmaşık Sorgum:

AND ve OR islemlerinin birlikte kullanıldığı uzun ve karmaşık sorgularda query optimizer bu sorgu koşulunu AND'in ana operatör olduğu formata dönüştürür. (conjunctive normal form (CNF))

$(\text{day} < 8/9/94 \text{ AND } \text{rname} = 'Paul') \text{ OR } \text{bid} = 5 \text{ OR } \text{sid} = 3$

↓  
query optimizer

$(\text{day} < 8/9/94 \text{ OR } \text{bid} = 5 \text{ OR } \text{sid} = 3) \text{ AND } (\text{rname} = 'Paul' \text{ OR } \text{bid} = 5 \text{ OR } \text{sid} = 3)$

### Sorgum Elemanına Yaklaşım

Sorgum işleminde, en az sayıda disk erişimi yapacak yöntem seçilmelidir.

Bu amaca göre index kullanılır ya da dosya baştan sona taranır.

### Örnekler:

$\text{day} < 8/9/94 \text{ AND } \text{bid} = 5 \text{ AND } \text{sid} = 3$

Eğer day üzerinde tanımlı BT ağacı indexi varsa kullanılır. Eğer day üzerinde hash index tanımlı olsaydı bu kullanılmazdı. Çünkü hash index ile aralık sorgusu yapılmaz.

I. Yöntem: BT ile istenilen aralıkta bulunan day kayıtları kullanılarak bid=5 ve sid=3 olan kayıtlar bulunur.

II. Yöntem: bid ve sid hash indexinden bulunur. Bulunan her kayıt için day < 8/9/94 koşulunu sağlayıp sağlanmadığı kontrol edilir.

► Bu yöntemlerden hangisi daha az sayıda disk erişimi yapıyorsa o yöntem tercih edilir.

## Index Kullanarak Sorgıim:

```
SELECT *
FROM Reserves R
WHERE R.rname < 'C%'
```

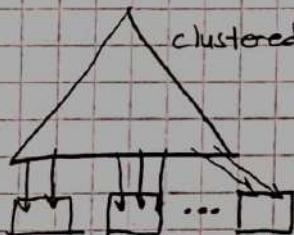
İsmi C harfinden küçük bir karakter ile başlayan reserves kayıtlarının tüm sütunlarını göster.

\* Bu şartı sağlayan kayıtlar %10

→ Eğer Reserves tablosu 1000 sayfadan oluşuyorsa ve her sayfada 100 kayıt varsa toplamda 100.000 kayıtlı bir tablo var demektir.

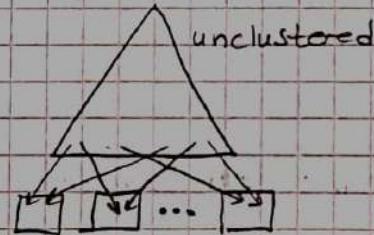
→ Bu tablo üzerinde hiçbir index yapısı yoksa filescan yapılmaktır zannedilir. (baştan sona tarama). 1000 sayfa olduğuna göre bu sorgunun maliyeti 1000 disk erişimidir.

→ Bu tablo üzerinde BT ağacı indexi tanımlıysa, bu ağacın clustered ya da unclustered olması maliyeti büyük ölçüde etkiler.



Burada ilk C ile başlayan kayıtlar bulunur ve ondan önceki tüm kayıtlar alınır.  
(100 disk erişimi olur)

⇒ %10'du en başta.



Burada ise yapraklar dengesiz şekilde bulunduklarından daha fazla disk erişimi yaparız.  
Her kayıt için baktanız lazımdır.  
(10000 disk erişimi olur)

Yani index kullanmak her zaman maliyeti düşürmez.

## Projection Algoritmaları

```
SELECT DISTINCT R.sid, R.bid
FROM Reserves R
```

A) 1000 sayfalık bir tabloya

- ★ Distinct olmasaydı, 1000 tane disk erişimi yapıp tüm verilere erişerek bunların sid ve bid sütunlarını gösterilirdi.
- ★ Ama buradaki gibi Distinct varsa tekrar eden satırlar elemeğinden ilave olarak su ikinci algoritmdan bir tercih edilir.
- Sorting Approach: sid ve bid sütunlarından oluşan daha küçük tablo sıralanır. Örneğin ikisi 100 sayfaya sağlanabilir. (sid ve bid'ın byte cinsinden boyutuna göre bu telirleniyor). 1000 disk erişimi Reserves tablosunu okumak için yaptı, 100 disk erişimi ile küçük tabloyu diske yazdı. External Merge Sort ile ( $2 * n * \text{geniş sayısı}$ ) bu tabloyu sıraladığını düşünüsek (geniş=2 olsun): 400 disk erişimi ile bu tabloyu sıralar. Sıralanmış sid, bid sütunlarından oluşan tablodaki tekrarları elmek için 100 sayfanın üzerinde 1 kez daha genilir ( $1000 + 400 + 100 = \underline{\underline{1500}}$ )
- Hashing Approach: 1000 disk erişimi ile Reserves alındı. sid ve bid değerlerini tablodan çekti. Her  $\langle \text{sid}, \text{bid} \rangle$  ikilisi için bir hash fonksiyonunu uygurdu. Hash'ten dönen sonucu göre bu ikiligi bir bucket'a gönderir. Bu işlemin sonunda her bir bucket kontrol edilecek (tekrar için). Bu diğerine göre daha düşük maliyetli olabiliyor.

### Join: Index Nested Loops Algoritması

foreach tuple  $r$  in  $R$  do  
 foreach tuple  $s$  in  $S$  where  $r_i = s_j$  do  
 add  $\langle r, s \rangle$  to result.

} i<sup>th</sup> inner loop olduğu  
 } i<sup>th</sup> nested loops dair.

\* S tablosu üzerinde index tanımlıysa  
 disk erişim

Maliyet =  $M + ((M * P_r)^* \text{ eslesen } S \text{ maliyeti})$

$M : R$  'deki sayfa sayısı

$P_r : r$  sayfası kayıt sayısı

→ Eğer  $S$  üzerinde bir hash index varsa kırmızı kismı 1.2 ortalaması disk erişimi olarak kabul ediyoruz.

→ Eğer  $S$  üzerinde B+ ağacı varsa kırmızı kismı 2,3 veya 4 disk erişimi olarak kabul ediyoruz. → primary key ise

→ Clustered ise kırmızı yere 1 eklenir, unclustered ise eslesen her  $S$  kaydı için 1 disk erişimi daha ekliyoruz.

"Örnek: Reserves ve Sailors tablolarını sid üzerinden join yaptığımızı varsayıyalım. ( $M$  : dis. döngüde bulunan tablonun sayfa sayısı,  $P_r$  : dis tablodaki her bir sayfada bulunan kayıt sayısı)

1-) Sailor in tablo olsun.  $M = 1000$ ,  $P_r = 100$  olsun ve hash index olsun.  
 Maliyet =  $1000 + ((1000 * 100) * (1.2 + 1)) = 221.000$

2-) Reserves in tablo olsun.  $M = 500$ ,  $P_r = 80$  olsun ve hash index olsun (clustered)  
 sid e için prim key olmadiğinden 2'de birer fazla SID değeri olabilir.  
 $R'$  nin kayıt sayısı /  $S$ 'nin kayıt sayısı  $= 100.000 / 40.000 = 2.5$  → yarık yok  
 Burada sid prim key değil  
 $Maliyet = 500 + ((500 * 80) * (1.2 + 2.5)) = 148.500$

\* Query Optimizer 2. yöntemini tercih eder.

## Join : Sort-Merge

iki tabloyu da birleştirilecek özelliğe göre sıralar. Daha sonra birinci tablodaki ilk kayıt ile ikinciyi karşılaştırır. Bunların özellikleri (örnekte bu özellik  $SID$  de濂idir) aynıysa join işlemini yapar. Eşit değilse soldakı tabloda bir sonraki özelliğe geçilir.

Maliyet =  $M$  tablosunu sıralama maliyeti +  $N$  tablosunu sıralama maliyeti +  $(M+N)$   
 ↓  
 sıralama maliyeti =  $2 * M * \text{geçiş sayısı}$  }  $M = \text{sayfa sayısı}$   
 ↓  
 Geçiş Sayısı =  $\lceil \log_{B-1} \lceil \frac{M}{B} \rceil \rceil + 1$  }  $B = \text{buffer pool'daki sayfa sayısı}$

Toplam Maliyeti

Örnek: (Burada sadece sayfa sayıları önemli, önceden farklı olabilir)

- Önceki .sayfadaki örnek üzerinden gittik.

Sailors = 500 sayfa , Reserves = 1000 sayfa , 300 buffer page , 2 giriş.

→ Sailors için giriş sayısı =  $\lceil \log_{300-1} \lceil \frac{500}{300} \rceil \rceil + 1 = ② \rightarrow G_S$  (giriş S)

→ Reserves için giriş sayısı =  $\lceil \log_{300-1} \lceil \frac{1000}{300} \rceil \rceil + 1 = ② \rightarrow G_R$  (giriş R)

Toplam Maliyet =  $2 * S * G_S + 2 * R * G_R + (S+R)$

$$\begin{aligned} & 2 * 500 * 2 + 2 * 1000 * 2 + (500 + 1000) \\ & = \boxed{7500} \end{aligned}$$

★ Bir önceki sayfada aynı işlem için en iyi performansla 148.500 adet disk giriş oluşturmak yerine burada sadece 7500 kez yaptıktı.

## System R Optimizer

Düşük maliyetli soru planı oluşturmak için DB'ler tarafından kullanılan bir yöntemdir.

- ★ Örneğin, selection ve projection işlemini mümkün olduğunda join işleminden önce gerçekleştirir. Böylece join işlemine katılacak tablolardan boyutlarını kısaltmış olur.
- ★ Bir önceki ilişkisel cebir sonucu elde edilenleri diske yazmadan bir sonraki adıma aktarmaya çalışır. (left-deep plans)
- ★ Kartezyen çarpımdan kaçınır. Çünkü hem kartezyen çarpım yapmak zahmetli bir istir hem de kartezyen çarpım sonucu ortaya çıkan tablo büyük boyutlu olacağndan diske yazmak çok maliyetlidir.

### Maliyet Tahmini:

Soru planı oluşturulurken maliyetinin de hesaplanması isteniyor. Bu hesaplamayı soru planının tümü için yaparsa çok uzun süren. Bunun yerine tahmin yolu ile bunu yapar. Bu tahmin, sorulanan algoritmeye ve tablonun boyutuna bağlıdır.

Bu tahmin şu formülle olur:

$$\text{Maximum kayıt sayısı} * \text{tüm reduction faktörlerin}\underbrace{\text{eleman sayılarının çarpımı}}_{\substack{\rightarrow \text{məsələdəki tablolardan}\\ \text{her bir term}\\ \text{için ayrı ayrı}\\ \text{hesaplanır.}}}$$

$$\underline{\text{col} = \text{value}} \rightarrow 1 / N\text{keys}(I)$$

$$\underline{\text{col } I_1 = \text{col } I_2} \rightarrow 1 / M \times (N\text{keys}(I_1), N\text{keys}(I_2))$$

$$\underline{\text{col } > \text{value}} \rightarrow (\text{High}(I) - \text{value}) / (\text{High}(I) - \text{Low}(I))$$

↓  
i indeksindeki en yüksek anahtar değeri.

$N\text{keys}(I) \rightarrow$  index üzerinde tanımlı anahtar sayısı

Örnek:

```

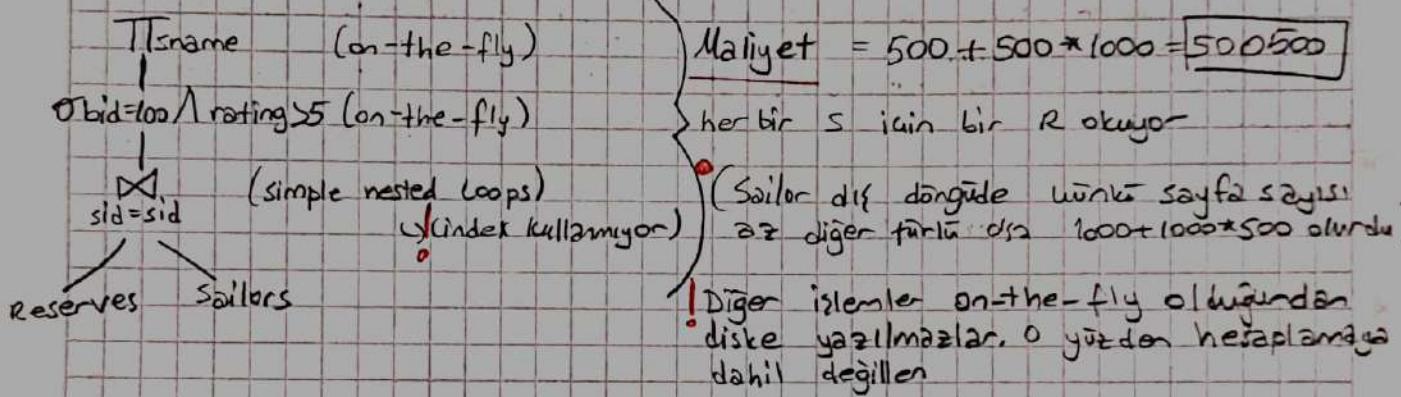
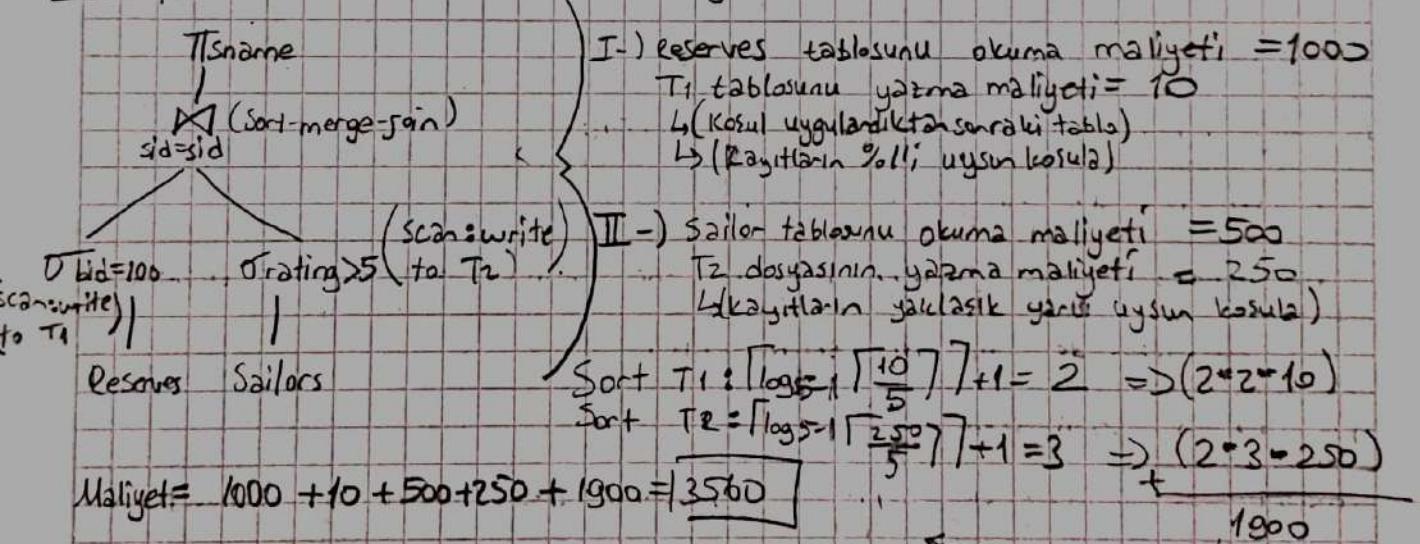
- SELECT S.sname
  FROM Reserves R, Sailors S
 WHERE R.sid=S.sid AND
       R.bid=100 AND S.rating>5
  
```

Reserves

Her kayıt 40 byte, her sayfada 100 kayıt  
toplam 1000 sayfa

Sailors

Her kayıt 50 byte, her sayfada 80 kayıt  
toplam 500 sayfa.

I. planII. plan

$\sigma = 100$  (%1'i olsun)

$\sigma > 5$  (%50'si olsun)

173

(selection için)

### III. Plan

Ttsname (on-the-fly)

rating > 5 (on-the-fly)

sid=sid (index nested loops)  
pipelining ile

10 disk erişimi yaparak Reserves tablosu  
üzerinden selection yaparız. (1000 kayıt.)  
Her Reserves kaydı için Sailors tablosundaki  
sid değeri aranır. (hepsi için ortalaması 1.2)  
 $(1000 * 1.2) = 1200$  (join için)

$$\text{Toplam maliyet} = 1200 + 10 = \boxed{1210}$$

(hash index)  $\sigma_{bid=100}$  Sailors (hash index)  
burada da var.

Reserves

## HAFTA SON

### CHAPTER 20:

#### - Physical Database Design -

Bu chapter'da ; view tanımlama, bir nitelik üzerinde hangi indexin oluşturulacağı,  
konuları ele alınıyor.

İş yükü : Projemizde sık kullandığımız sorular.

\* Bir tabloyu oluştururken

• Hangi indexleri hangi tablolar üzerinde ve hangi nitelikleri baz olarak  
oluşturmalıyız?

• Bir tablo üzerinde kaç index tanımlanmalı?

gibi sorulara yanıt aranır.

\* Normalizasyon kullanarak BCNF'ye dönüştürülmüş tablolar üzerinde

(ve 3NF)  
yapılan sorular düzük performansa ulaşabilir. Bu sorunu gidermek için  
join işlemleri yapıp normalizasyondan vazgeçilebilir. (denormalization)

### Join iin Index Seçimi

- Index nested loops algoritmasında iu döngünün bir hash indexinin olması performansı artırır. Ayrıca, eğer join niteliği (size) iu döngü iin primary key değilse iu döngüdeki indexin clustered olması performansı artırır.
- Sort-Merge algoritmasında join yapılacak iki tablodaki join nitelikleri üzerinde clustered B+ ağacı tanımlamak performansı artırır.

### Örnek 1:

SELECT E.ename, D.mgr  
 FROM Emp E,Dept D  
 WHERE D.dname = 'Toy' AND E.dno = D.dno

) Bu soruya göre  
 hangi indexleri  
 olusturmamız gerektigine  
 karar verecegiz.

- Öncelikle where cümlesine bakıyoruz. (join yapmadan önce verileri olabildigince küçültmeye çalışıyoruz ilk önce selection islemlerini yaparak) dname iin bir index olusturmalıyız ki ismi 'Toy' olanları hızlıca bulabilisin. Bu indexin türü hash index'tir (esitlik sorusu olduğu için) Departman tablosu iin daha fazla indexe ihtiyaç yok. Çünkü where cümlesinin ikinci koşulunda bir join yapılmış (index nested loops) ve Departman tablosu die döngüde bulunuyor. Die döngüdeki tablo iin de indexe ihtiyaç yoktur. Fakat iu döngüdeki tabloda indexe ihtiyaç vardır. Employee tablosundaki dno üzerine hash index tanımlanır.
- ↳ where cümlesine ... AND E.age = 25 eklenirse : selection isleminin goinden önce olması gereğinden bu önce yapılır. age niteliği iin bir hash index olusturulur. Hem Department hem de Employee tablosunda selection yapıldığından joine girecek tabloların boyutu küçük olacaktır. Bu nedenle dno üzerindeki indexe arlık ihtiyaç olmaz.

Örnek 2:

```

SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE E.sal BETWEEN 10000 AND 20000
    AND E.hobby = 'Stamps' AND E.dno=D.dno
  
```

→ Bu sorguda E tablosu iin 2 tanrı selection işlemi var. Bu sebeple join yaparken index nested loops algoritmasında E outer loop'ta olacak. Yani D tablosu içi döngüde kullanılacak. D tablosunun dno niteliğine göre bir hash index oluşturulmalı. E tablosu üzerinde hangisine göre index yapmalıyız? E tablosu üzerinde birden fazla index oluşturabilir miyiz?

→ E tablosu üzerinde birden fazla index oluşturabiliriz. (sal iin B+ index, hobby iin hash index). Fakat oluşturulan her indexin diskte yer kaplayacağı unutulmamalıdır. Aynı zamanda tablo üzerinde degrıkkılık yapıldığında tüm indexlerin güncellenmesi gerektiğinden aynı tablo üzerinde birden fazla index tanımlamak performans kaybına neden olur. Bu sebeple sal ve hobby niteliklerinden yalnızca biri iin index oluşturmalıyız. Esitlik sorguları, aralık sorgularından muhtemelen daha az elemen getireceğinden hobby üzerine hash index tanımlamak daha mantıklıdır.

★ Sonuç olarak: dno iin hash index, hobby iin hash index tanımlanır.  
(optimizer tarafından)

## Clustering and Joins

### Örnek :

```
SELECT E.ename, D.mgr
FROM Emp E, Dept D
WHERE D.dname = 'Tog' AND E.dno = D.dno
```

→ dname için hash index yaparız. D tablosu dis, E içi döngüde dur.

E'nin dno niteliği için hash index yaparız. dno, E için primary key olmadığından clustered hash index yapmamızı.

## Tuning the Conceptual Schema

Bir veritabanı semasını oluşturduktan sonra sorguların daha hızlı çalışması için bir takım işlemler yapılabilir. (3NF formuna dönüştürmek, BCNF'e dönüştürmek, normalizasyonu gev altı, bazı tablolara ilave sütunlar eklemek, daha fazla decomposition yapmak, yatay bölme ...)

### Örnek :

Contracts (Cid, sid, jid, did, pid, Qty, Val.)

Bu tablo üzerinde en fonksiyonel bağımlılıklar olsun:  $JP \rightarrow C$ ,  $SP \rightarrow P$ , C prima key (JP adayı anahtarıdır (çünkü C'yi belirler))

→ Bu tablo BCNF degildir. (SD anahtar değil, P SD'nin alt kümlesi değil) ama 3NF'dir (P, bir anahtarın alt kümlesi) . Bu tabloyu BCNF hale getirmek için  $SD \rightarrow P$ 'yi kullanıyoruz. Bu bölünme lossless J-decomposition (P'yi arıtablodan atıyoruz) ama dep. preserving degildir. (çünkü  $JP \rightarrow C$  bosta kalıyor).

↳ Dikketimiz, sorguların büyük bölümünde Q, P ve C birlikte kullanılıyor. Böyle bir durumda çoğu zaman yukarıdaki böldüğümüz iki tabloyu (CSJDOV ve SDP) join etmemiz gerekiyor. Bu da çok maliyetli olduğundan tabloyu original halde bırakmak daha mantıklıdır.

## Denormalization

"Bu sözleşmenin değeri departman bütçesinden az mı?" gibi bir soru bizim için önemliyse Department tablosundan bütçe bilgisini Contracts tablosuna eklemeliyiz. Bunun için  $D \rightarrow B$  gibi bir fonksiyonel bağımlılık yazılır. Bu durumda Contracts tablosu artık 3NF'de değildir.

## Choice of Decompositions

Contracts tablosunu BCNF yapmak için iki yol var:

- SDP ve CSJ DQV : losses join ama preserving değil. ( $JP \rightarrow C$  'den dolayı)
- SDP, CSJ DQV ve CJP : preserving sorunu düzeltti (Ama burada da veri tekrarı var)

\* İki tabloya bölerek ve  $JP \rightarrow C$  fonksiyonel bağımlılığını kontrol edebilecek bir Assertion tanımları (Assertion, bir den fazia tablosu aynı anda ilgilendiren dayalar için yazılır)

\* Tabloları bölmek veya onları birleştirmek sık gelen sorulara göre maliyet hesabı yaparak belirlenir. Örneğin; Cogu soru yalnızca C ve D niteliklerini içermesine rağmen CD'yi ayrı tablo yapmak mantıklı olabilir (CSJ DQV tablosunda işlem yapmaktaşa CO'de işlem yapmak daha performanslıdır.)

## Horizontal Decomposition (Yatay Bölünme)

Daha öre gördüklerimiz dikey bölünmeydi. Yani tabloyu sütunlardan ayırmıyordu. Ama yatay bölünmelerde tablo satırlarını bölünür. (Yani bölünen tablolardaki sütunlar hala aynı)

Örneğin; Contract tablosunda value > 1000 olan satırlar için çok fazla soru geliyor. O zaman bu tabloyu yatay bölüyoruz. ( $>1000$  ve  $<1000$  diye)  
 (small                      large)

## Sorguyu Yeniden Yazma :

Bazen yazdığımız bir sorgu beklediğimizden yavaş gelisabilir. Bunun birkaç nedeni olabilir.

- Kullandığımız index eski olabilir (Artık o tabloya uygun değil)
- Sorguda null değer içeren koşullar olabilir.
- Aritmetik veya string ifadeden kaynaklı olabilir.
- OR koşulundan dolayı olabilir.
- Join işlemi varsa.
- ★ → İKİ İKE SORGULARI TEK BLOKTA YAZMAYA ULAŞ!

```
SELECT DISTINCT *
FROM Sailors S
WHERE S.name IN
  (SELECT Y.sname
   FROM YoungSailors Y)
```

```
SELECT DISTINCT *
FROM Sailors S, YoungSailors Y
  WHERE S.sname=Y.sname
```

↳ İKİ SORGUDA DISTINCT, AGGREGATE OPERATORLAR VEYA NULL KARŞILAŞTIRMASI VARSÄ YUKARIDAKİ GİBİ YAPAMAYIZ.

★ SORGUNUN DAHA İYİ UÇLUŞMASI İÇİN DISTINCT YAZMAKTAN KAÇIN.

★ GERÇEKTE İHTİYAC YOKSA GROUP BY VE HAVING KULLANMAKTAN KAÇIN.

```
SELECT MIN(E.age)
FROM Employee E
GROUP BY E.dno
HAVING E.dno=102
```

```
SELECT MIN(E.age)
FROM Employee E
  WHERE E.dno=102
```

★ Ara tablo yapmaktan kaçın (SELECT \* INTO Temp gibi yanı)

## HAFTA XI SON

DAS İŞLEMİDİ (Bayram Tatili)

## HAFTA XII. SON

## CHAPTER 16 :

### - Transaction Management -

#### Transactions :

Bir kullanıcının veritabanı üzerinde bir işlem gerçeklestirmesine transaction denir.  
(okuma-yazma)

Birden fazla kullanıcı bir DB üzerinde aynı anda işlem (transaction) gerçeklestirmek istiyor olabilir. Bu işlemlerin yönetilmesi gereklidir.

\* Kullanıcı programları DB üzerinde eşzamanlı (concurrent) olarak çalışır. Bu eşzamanlı çalışma sırasında DB'deki verilerin tutarlı olması gereklidir. (Önceden gerekğimize sınırlamalar ile bu tutarlılık sağlanabilir).

#### ACID :

Transactions su 4 özelliği sağlıyorsa bunlar eşzamanlı olarak çalışabilir:

- Atomicity: Transaction atomik olmalıdır. Yani bir transaction başladığın zaman ya başarıyla sonlanır ya da transaction çalışırken bir hata oluşursa sanki bu transaction hiç başlatılmamış gibi davranışır.
- Consistency: (Tutarlılık) Transaction'un yaptığı hesaplamalar doğruya ve bu transaction başlamadan önce DB de tutarlıysa transaction sonrası DB yine tutarlı olmalıdır. (Transaction kodları düzgün yazılmalı)
- Isolation: Birden fazla transaction eşzamanlı çalışırken her birinin diğerlerinden izole olmalıdır. Yani onların çalışmalardan etkilenmemelidir.
- Durability: Eğer bir transaction 'commit' ediyorsa (yani başarıyla sonluyorsa), transaction'un yaptığı etkilerin DB'ye yansıtılması olması gereklidir.

\* Yani transaction atomikse ; commit edince DB'ye yazılması, abort (iptal) ediyorsa transaction daslatılmamış gibi yapılın. (disk üzerindeki değişikler geri alınmak). Bu da log kayıtları sayesinde olur.

Örnek :

$T_1 : \text{ BEGIN } A = A + 100, B = B - 100 \text{ END}$
$T_2 : \text{ BEGIN } A = 1.06 \cdot A, B = 1.06 \cdot B \text{ END}$

Bu tabloya cizelge denir.  
(Schedule)

→ Bu iki transactionun es zamanlı  
calışacığını varsayıyalım.

\*  $T_1 : B$  hesabından 100\$ parayi,  $A$  hesabına aktarıyor.

\*  $T_2 : \text{ Her iki hesaba da } \%6 \text{ faiz uyguluyor.}$

→ Her iki hesapta da 200\$ olduğunu varsayıyalım.  $T_1$  ve  $T_2$  bittikten sonra  $A$  hesabında 318\$,  $B$  hesabında 106\$ olur.

\* İki transaction arası geçiş yaparak ikisini de yürütmeye interleaving denir. Yukarıdaki örnek için bir cizelge daha oluşturyalım.

①

$T_1 : A = A + 100$	$B = B - 100$	→ Bu cizelgede $T_1$ ve $T_2$ sırası sırası değiştirilmemiştir. Bu gecici bir cizelgedir. Son durumda $A = 318\$$ , $B = 106\$$
$T_2 : A = 1.06 \cdot A$	$B = 1.06 \cdot B$	

Başka bir alternatif cizelgeyi ele alalım :

②

$T_1 : A = A + 100$	$B = B - 100$	→ Bu seferli bir cizelge değildir. Son durumda $A = 318\$$ , $B = 112\$$ ( $6\$$ fazla)
$T_2 : A = 1.06 \cdot A, B = 1.06 \cdot B$		

\* Aslında DBMS transactionları yukarıdaki gibi işlemler şeklinde görmez.

İşte şekilde görür (Cizelge-2 baz alındı)

$T_1 : R(A), W(A),$	$R(B), W(B)$	→ Bu şekilde de gösterilebilir.
$T_2 : R(A), W(A), R(B), W(B)$		
A nesnesini oku (read)	A nesnesini yaz (write)	$T_1$ $R(A)$ $W(A)$  $T_2$ $R(A)$ $W(A)$ $R(B)$ $W(B)$  $R(B)$ $W(B)$

\* Bir cizelgedeki transactionların sırası değiştirilememeli.

## Scheduling Transactions

- Serial Schedule : (Seri çizelge) : Bir transaction başlayınca , bitinceye kadar galistirilin tari sırayla baştan sona nöpsi çalışır. (Geçiş yok)
- Equivalent Schedule : (Esit çizelge) : Eğer iki çizelgenin son hallerindeki etisi aynıysa bunlar esit çizelgelerdir. Örneğin; ömeki sayfada bulunan örneğin çizelgesi ve 1 numaralı çizelge . (İkisinin de net etisi aynıdır)
- Serializable Schedule : (Serializeli olabilir çizelge) : Bir çizelgenin net etisi bir seri çizelgeye eşitse (örneğin 1 numaralı çizelge) bunu serialize edilebilir deriz. Bu tarz çizelgeler çalışmadan önce DB tutarlıysa bunlar çalışıktan sonra da tutarlı durumda dur. (2 numaralı çizelge bu sınıfa girmez ve DB'nin tutarsız olmasına neden olur)

## Eşzamanlı Gelişme Sırasındaki Anomaliler

### Reading Uncommitted Data (WR conflicts - Dirty Read)

T1 : R(A), W(A)	R(B), W(B), Abort	Commit etmemiş bir transaction, güncellediği bir veriyi diğer transaction okuyorsa burada problem var demektir.
T2 :	R(A), W(A), C	

\* T1 çalışmadan, T1'in değiştirdiği veriyi T2 okudu ve T2 bu veri (A) üzerinde değişiklik yapıp DB'ye commit etti. Fakat T1 belli bir yerde abort (iptal) edildi. Transactionların atomik ama prensiplerden dolayı T1'in yaptığı değişiklik geri alınmalı. Fakat T2, T1'in güncellediği A verisi üzerinde işlem yaptığı ve bunları commit ettiği için DB üzerinde T1'in yaptığı değişiklik kalıcı hale geldi ve A verisi serialinmedi. Böylece veri tutarsızlığı oldu.

\* Commit etmemiş bir transactionun yazmış olduğu veriyi diğer transactionlar okumamalı.

### Unrepeatable Reading (RW Conflicts):

T<sub>1</sub> : R(A),

R(A), W(A), C

T<sub>2</sub> :

R(A), W(A), C

Bir transaction güncellmediği bir veriyi tekrar okudüğünde bu veri değişmişse turada problem vardır.

★ T<sub>1</sub> A'yi okudu sonrasında T<sub>2</sub> A'yi okudu, değişiklik yaptı ve commit etti. T<sub>1</sub> A'yi güncellememiş olmasına rağmen güncellenmiş bir A verisini DB'den okudu.

### Overwriting Uncommitted Data (WW Conflicts):

T<sub>1</sub> : W(A),

W(B), C

T<sub>2</sub> :

W(A), W(B), C

Commit edilmemiş bir veri üzerine tekrar bir verinin yazılması probleme yol açar.

★ T<sub>1</sub> A varisini değiştirdi ama henüz commit etmedi. T<sub>2</sub> başladı ve A'yi güncelledi. Burada WW Conflict olur. Çünkü T<sub>1</sub> commit etmediği için onun güncellediği verinin üzerine başka bir transaction güncelleme yaptı ve T<sub>1</sub>'in yaptığı güncelleme kayboldu. B'nin güncelleme işleminden (T<sub>2</sub> 1'den T<sub>1</sub>'e geçerken) sorun yok. Çünkü T<sub>2</sub> geçerden önce commit etti.

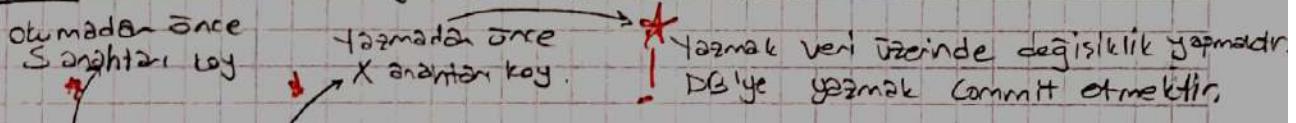
→ Bu taraflı sorunlardan korunmak için DBMS bazı yöntemler kullanır.

## Lock-Based Concurrency Control

Önceki kısımdaki sorunları çözmek için "Strict Two-phase Locking" ya da kısaca 2PL yöntemini sırtına kullanılır. Bu yöntemde bir transaction bir veritabanı nesnesini okuyacağı zaman bu nesne üzerinde bir paylaşımı (shared) kilit koyar (S). Bu kiliti koyamıyorsa o nesneyi okuyamaz. Eğer bir yazma işi yapılacaksa nesne üzerinde exclusive lock (X) koyması gereklidir. Bu kiliti koyamazsa yazma yapamaz. Bir nesne üzerinde shared lock varsa diğer transactionlar da onun üzerinde shared lock koyarak okuma yapabilir. Fakat bir nesne üzerinde (veri) exclusive lock varsa diğer transactionlar onun üzerinde ne shared ne de exclusive lock koyabilir. (Yani onun kullanılmasını beklemek zorundalar)

Transaction bittiği zaman lock'ları kaldırır. (non-strictte titmeden de kilit bırakılabiliyor)

### Örnek : WR Conflict - Dirty Read Sorunu Üzüm



$T_1 : S(A), R(A), X(A), W(A), , S(B), R(B), X(B), W(B), \text{Abort}$

$T_2 : , S(A), R(A), X(A), W(A), C$

Bu kısımda  $T_2$  çalışmaya başladı. Fakat A üzerinde X lock olduğunuandan  $T_2$ , A'nın üzerindeki anahtarın kalkmasını bekledi.

→ Burada  $T_1$ 'in abort edilmesi durumu artık sorun çıkarmayan biriu  $T_2$ ;  $T_1$  bittiğinden sonra çalışıyor (Tutarlılık sağlandı ✓)

## Örnek: RW Conflict with 2PL

T<sub>1</sub>: S(A), R(A)

DÜS T<sub>1</sub> ile devam eder. Burada da Tili A'ya girebilmesi için X lock koyması gereklidir. T<sub>2</sub> S lock koymadığundan bunu yapmaz. (T<sub>2</sub> bitti)

T<sub>2</sub>:

S(A), R(A)

X lock koyması gereklidir. Ama T<sub>1</sub>'de A üzerinde S lock olduğu için yapmaz. (T<sub>1</sub> bitti)



Burada bir Dead lock olusur. (nâdiren olusur). Timeout metodu ile bu yöntem kullanılır. Sıkıntıları transactionlar abort edilir ve başka bir uyelege kullanılarak yeniden çalıştırılır.

## Aborting a Transaction:

Bir transaction abort edileceği zaman o ana kadar yaptığı tüm işlemlerin geri alınması gereklidir. Geri alınma işi yapıldıktan sonra, abort edilen transactionun güncellediği veri üzerinde işlem yapan transactionlar da geri alınmaktadır. Bunaın hepsine cascading abort denir (domino etkisi gibi).

Bu işlemleri gerçekleştirebilmek için log kayıtları tutulur (diskte bulunur).

\* Veriyi değiştirmeye, commit ve abort durumlarında log kaydı tutulur.

## Recovering From a Crash

Sistem çökmesi durumunda DB'deki verinin tutarlı hale getirilemesi için çeşitli algoritmalar vardır. Bunlardan en bilineni ARIES algoritmasıdır.

Bu algoritma 3 fazdan oluşur:

- Analysis: Bu kısımda, sistem çöküğü zamanı hangi transactionlar aktif (yani commit veya abort edilmemiş, sonlanmamış) ve buffer poolda bulunan hangi sayfalarda veri güncellenesi yapılmış bunlar belirlenmeye çalışılır.

• Redo: Buffer poolda güncellenen sayfalar diske yazılır. Log kaydındaki güncellemeler de diske yansıtılır.

• Undo: Sistem çökmesi anında aktif olan transactionların yaptığı tüm güncellemeler geri alınır. (Log kullanılarak yapılır.)

↳ Bu adımlar sonunda var, tutarlı hale gelmiş olur.

### Transaction Support in SQL:

SELECT, UPDATE, CREATE komutlarını kullandığımızda aslında bir transaction başlatıyoruz. Transactionu bitirmek için COMMIT veya ROLLBACK komutu kullanılır.

COMMIT: O ana kadar yapılan işlemler DB'ye yansıtılır.

ROLLBACK: Save pointe kadar yapılan değişiklikleri geri alın

⋮                    } Buradaki değişiklikler kaydedildi.  
COMMIT

SAVEPOINT <savepoint-name>

⋮                    } Buradaki değişiklikler geri alındı.

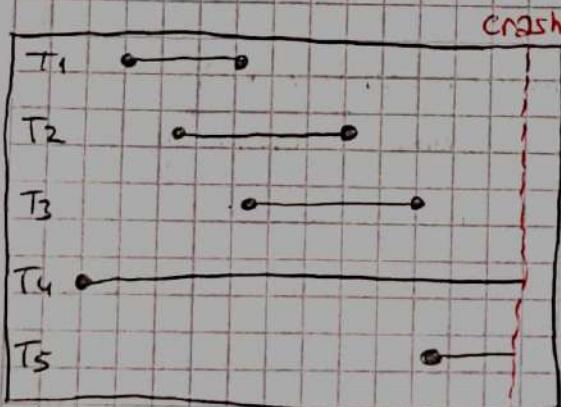
ROLLBACK TO SAVEPOINT <savepoint-name>

———— HAFTA XIII. SON ———

## CHAPTER 18:

### - Crash Recovery -

Bu chapter'da sistem çökmesi durumunda veriyi nasıl kurtaracağımızı, öğreneceğiz.



\* Recovery Manager, sistem çökmesi anında hangi transactionların aktif olduğunu araştırır. ( $T_4, T_5$ ) Sistem çökmesi anından önce commit etmiş transactionları ( $T_1, T_2, T_3$ ) DB'ye yansıtır. Sonlanmayanların yaptığı işlerin geri alınması gereken (abort) Bunu 2PL ile yapar.

### Handling the Buffer Pool

- \* Bir transaction commit edipsa yapılan değişikliklerin diske anında yansıtılmasının force denir. Fakat sürekli diske erişmek maliyetli bir istir. Bunun için transactionlar toplu sekilde diske kaydedilir. (no force)
- \* Buffer pool doluyسا yeni gelen veri için buffer poolda bir sayfa ayrılmalı (sayfa 13). Bu ayrılan sayfada commit etmemiş transaction varsa bunun yaptığı değişiklik commit etmemesine rağmen diske yazılır. Bu现象e de Steal denir. Peki diske erken yazılan bu transaction abort ederse ne olacak? → (Toplu değişiklikler log ile geri alınır)

## Logging

Aynı bir disk üzerine ardışık olarak kayıtlar yazmaya loglama denir.

Bir log'lu sekildeşidir:

<transaction-id, sayfa-id, offset, uzunluk, eski-veri, yeni-veri >

\* Bu kayıtlar oluşturulurken Write-Ahead Logging (WAL) kullanılır.

1-) Önce log kaydını yaz sonra değişikliği diske yansıt. (Atomiclığı garanti eder)

2-) DBMS, commit eden bir transaction'in o transaction'un tüm log

kayıtlarının yazılmış olması garanti edilmeli. (Durability'yi garanti eder)

\* Her log kaydının bir numarası vardır. Bu kara Log Sequence Number (LSN)

denir. (id gibi). Bir veri sayfasının pageLSN değeri vardır. (Bu sayfaya yapılan en son güncellemein log kaydının numarası). Sistem yazılmış olan log kayıtlarının numaralarını flushedLSN ile tutar. (0 ana kadar yazılmış olan LSN değeri).

↳ Buna göre bir sayfa yazılmadan önce pageLSN  $\leq$  flushedLSN olmalıdır.

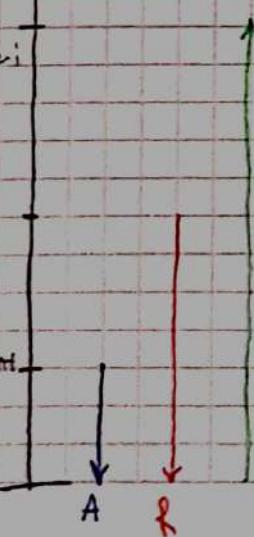
\* DBMS, periyodik olarak checkpoint oluşturur. Sistem yoknesi durumunda son checkpoint baz alınarak işlemler yapılır. (Daha öncekiler diske yazılmıştır)

Crash öncesi durum  
(tüm transaction'ları geri  
alınır.)

Aktifde  
yazılmış

Son checkpoint

CRASH



Aktif olan transactionlar bulunur. • Analysis  
Aktifin diskeki etkileri geri alınır. • Redo  
Crash öncesi hale getirme. • Undo  
↳ transaction işleneleri  
geri alınır.

Örnek: WAL ile elde edilen log kayıtları şu şekilde olsun.

LSN	LOG
00	Checkpoint oluştur
05	Checkpoint sonu
10	T1, P5 sayfasını güncelledi
20	T2, P3 sayfasını güncelledi.) ←
30	T1 abort
40	10 numaralı LSN'yi geri al (undo)
45	T1 sonlandı
50	T3, P1 sayfasını güncelledi
60	T2, P5 sayfasını güncelledi.

X CRASH, RESTART

\* Crash olduğunda checkpointe gidilir (00). Bu checkpointten itibaren log kayıtlarına tek tek bakılır. Hangi transactionların aktif olduğunu tespit edilir (T1, T2, T3 checkpointten sonra geldi). Bunaın yaptığı işler REDO edilir. Veritabanına yansır. Daha sonra (60) T2 gecike gidilerek UNDO yapılır (T2, P5 ve P3'ü değiştirmisti bunlar geri alınır vs., bitmemiş transaction'a UNDO yapılır!)

→ Kısaca: Crash olursa checkpointe git ve o işlemleri tekrar yap eğer bitmemiş transaction varsa bunların ilerlerini diske yansıtma. (Yani undo T2 ve T3 için yapılır)

**CHAPTER 21:****-Security and Authorization -**

- Kullanıcılar sadexe kendine izin verilen verileri görebilmeli (Secrecy)
- Verinin bütünlüğü sağlanmalıdır. (Kullanıcı, izni olmayan bir şeyi değiştirememeli (Integrity))
- Verinin, izin verilen kişiler tarafından değiştirilebilmesi gerekir (Availability)

★ DB'de güvenlik, kullanıcılara bazı izinler vererek sağlanır. 2 farklı erişim kontrolü yöntemi vardır. (isteğe bağlı, zorunlu)

### 1-) Discretionary Access Control : Veritabanı nesnesi (tablo veya view)

Üzerinde belirli kullanıcı veya gruplara haklar tanımlanır. Nesneyi oluşturan kişi tüm haklara sahiptir. Diğerlerine bazı haklar verebilir. Bu işi aşağıdaki SQL cümlesi ile yapar.

{GRANT privileges ON object TO users [WITH GRANT OPTION]}

- ↳ privilege yerine → SELECT, INSERT(column), DELETE ---
- ↳ object yerine → Tablo veya view ismini
- ↳ users yerine → Kullanıcı veya kullanıcılar.
- ↳ WITH GRANT OPTION → Kendisine verilen yetkiyi başlarına da verebilir.
- ★ Kullanıcılarla hiçbir zaman CREATE, ALTER ve DROP yetkileri verilemez.
- ★ REVOKE komutu ile verilen yetkiler geri alınır.

#### Örnek :

GRANT INSERT, SELECT ON Sailors TO Horatio

Horatio isimli kullanıcı, Sailors tablosu üzerinde INSERT ve SELECT yapabiliyor.

GRANT DELETE ON Sailors TO Guppy WITH GRANT OPTION

→ Guppy isimli kullanıcı, Sailors tablosu üzerinde DELETE işlemi yapabilir ve bu işlemi yapma yetkisini diğer kullanıcılar da verebilir. Guppy'nin yetkisi REVOKE ile geri alınırsa Guppy'nin hak verdiği visilerin DELETE hakları da elliinden alınır.

GRANT UPDATE(rating) ON Sailors TO Dustin

→ Dustin isimli kullanıcı, Sailors tablosundaki yalnızca 'rating' sütunu üzerinde UPDATE işlemi yapabilir. (sadece UPDATE yazısında tüm sütunlar üzerinde UPDATE işlemi yapabilirdi).

GRANT SELECT ON ActiveSailors TO Grumpy, Guppy,

→ Guppy ve Guppy isimli kullanıcılar, ActiveSailors view'i üzerinde SELECT işlemi yapabilir. (Sailors tablosu üzerinde yetkileri yoktur)

\* Eğer bir kişi bir tablo kullanarak view oluşturmuşsa ve bu kişiden tablo üzerindeki yetkiler alınırsa view silinir. Tıpkı bu kişinin WITH GRANT OPTION hakkı elinden alınırsa view üzerinde de taskalarına yetki' veremez.

\* View tanımlamakta bir güvenlik önlemi olabilir. (Kullanıcılarla belli seyler göstermek). GRANT / REVOKE ile kullanırsız güclü bir veri erişim kontrolü sağlanır.

\* Veri güvenliğini : Kullanıcılarla haklar vererek, view kullanarak ve şifreleme algoritmaları kullanarak sağlıyoruz.

## RSA Public-Key Encryption :

- Sifrelemek istenen veri bir tamsayı olsun ( $I$  isminden).
- $I$  tamsayılarından çok daha büyük bir  $L$  tam sayısı seçilir. (random olarak).
- $L$  tamsayısi çok büyük bir sayı olmalı ve iki tane farklı asal sayının çarpımına eşit olmalı. ( $L = p \cdot q$ ,  $p$  ve  $q$  asal)
- Sifreleme için  $1 < e < L$  olacak şekilde bir  $e$  sayısı seçilir.
- $e$  sayısı  $(p-1) * (q-1)$  işleminin sonucuna göre arasında asal sayılar olacak  
(bu işlemin sonucu  $S$  olsun)
- Encryption fonksiyonu :  $I^e \text{ mod } L$   $\rightarrow$  (Bu formül bize sifrelenmiş sayıyı verir)
- Sifre çözme için bir  $d$  sayısı seçilir. Böyle ki bu  $d$  sayısı  $d * e = 1 \text{ mod } ((p-1) * (q-1))$  olacak şekilde seçilmiştir.
- Decryption fonksiyonu :  $S^d \text{ mod } L$   $\rightarrow$  (Sifrelenmiş sayıyi original haline döndürür)

★ SSL, SET gibi sertifikalar almaktan bir güvenlik önemiidir.

2-) Mandatory Access Control: Her veritabanı nesnesinin ve kullanıcıların bir güvenlik sınıfı vardır. Bu sınıflara göre kullanıcılar belirli verileri okuyup yazabilir. Bu erişim kontrolü sadece çok güvenli olan sistemlerde kullanılır. (Askeriye)

★ D kate niyetli liselerin Horse isimli bir tablo oluşturduğunu düşünelim. Justin isimli kullanıcıya Horse tablosu üzerinde INSERT yetkisi versin. (Justin'in bundan haberi yok). D, Justin'in kullanacağı kodları, tabloya INSERT yapacağı şekilde güncellenmiş olsun. Justin'in yazdığı 'gizli' veriler Horse tablosuna da kaydedilir. Böylece bu gizli bilgiler D'nin eline geçmiş olur.

(Discretionary Control Güvenlik Problemi - Truva Atı - )

## Bell-La Padula Model

Örnek sayfadaki güvenlik zayıflığının önüne geçmeyi planlar.

- Nesneler = Tablolar, viewlar, tuple'lar
- Özneler = Kullanıcılar, kullanıcı programları
- Güvenlik Sınıfları = Top Secret (TS), secret (S), confidential (C), unclassified (U)

$$TS > S > C > U$$

\* Her nesnenin ve özenin güvenlik sınıfı vardır.

- Bir kullanıcı kendi güvenlik sınıfından daha düşük veya ona eşit nesneleri okuyabilir.
- Bir kullanıcı kendi güvenlik sınıfından daha yüksek veya ona eşit nesneleri yazabılır.

\* Bu durumda truva atı sorunu söyle engelleniyor:

D kişi C sınıfında olsun. Bu durumda Hansle tablosunun da güvenlik sınıfı C veya daha düşük olacaktır. Justin de S sınıfında olsun. Bu durumda S sınıfına mensup olan Justin kendi güvenlik sınıfında daha düşük olan C sınıfına ait Hansle tablosuna veri yazamaz.

\* Tablolardaki her kayıt için güvenlik sınıfı verilebilir.

Örnek:

bid	bname	color	class
101	Salsa	Red	S
102	Pinto	Brown	C

→ TS ve S sınıfı tüm satırları görebilir  
C sınıfı yalnızca 2. satırı görebilir  
U sınıfı hiçbir satırı göremez.

! Burada bir sorun var: Öğün bu tabloya? C sınıfında kullanıcı <101, Pasta, Blue, C> verisini eklemek istesin. (Kullanıcı ilk satırı göremiyor). Eğer 101 id'li

kayıdı eklemek istersen PRIMARY KEY özelliğini kaybetmez. Ama eklemesek bunu kez kullanılsın kendisinden daha üst seviye bir güvenlik seviyesine sahip kaydın varlığından haberden olur.  
(bid ve class'ı PRIMARY KEY yaparsak sorun ortadan kalkar)

## Statistical DB Security : (statistiksel)

Bu veritabanı modelinde veriler DB'de saklanır. Fakat DB'de soru yaparken kayıt bazında soru yapamayız. Genellikle bu yöntem, kişisel verileri korumak için kullanılır. (Hastane veritabanlarında).

Örneğin : Bir kişinin adını biliyorsak "bunun yaşı nedir?" diye bir soru yapamayız. Bunun yerine "DB'de kişiler ortalaması yaşı nedir?" gibi bir soruya izin veriliyor.

Burada söyle bir problem olabilir: Eğer bir kişinin DB'deki en yaşlı kişi olduğunu biliyoruz ve yaşını bulmak istiyoruz. Sırayla "X yaşında daha yaşlı kişi var?" sorusunu cevap 1 olana kadar gönderebiliriz. Böylece o kişinin yaşına erisiriz.

Bunun önüne geçmek için kullanıcıının göndereceği sorular üzerinde bir sınırlama yapılmıştır. Soru sayısı arttıkça DB'in dönderdiği cevaplar üzerine gürültü eklenir (gittikçe artan gürültü)

HAFTA XIV SON

Ders işlenmedi

HAFTA XV. SON

Son

