

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
Ağac 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
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.

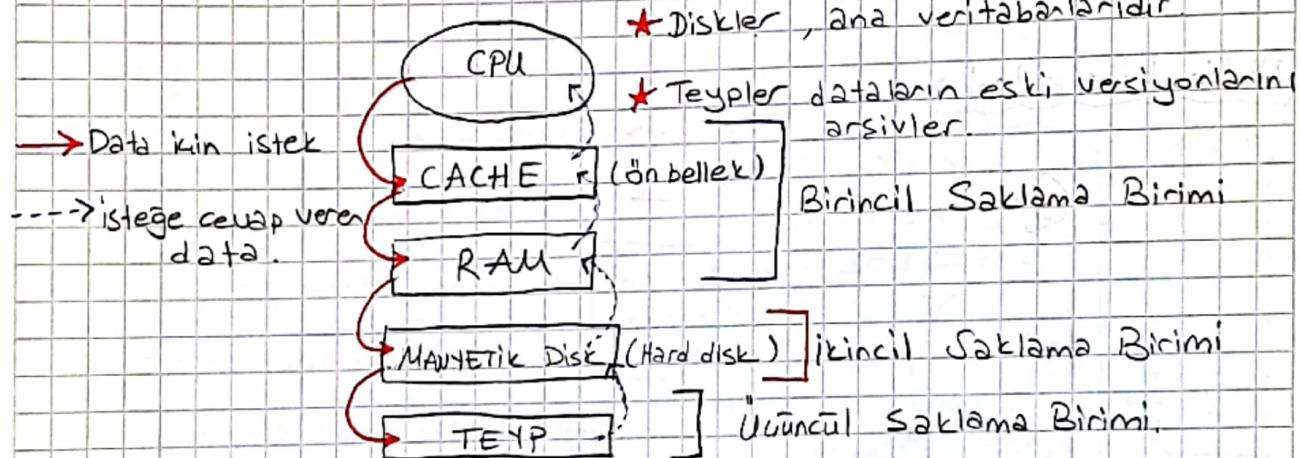
! Bunlar yüksek maliyetli işlemlerdir, dikkatli planlanmalıdır.

Bellek Hiyerarşisi:

* RAM belleği taşıyan kısımdır.

* Diskler, ana veritabanlarıdır.

* Teypler dataların eski versiyonlarını arşivler.



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

→ Fiyatı çok pahalı. (32 GB RAM = 1900 TL)

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

Saklama Birimlerinin Karşılaştırılması

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

* Teypleki verilere erişim

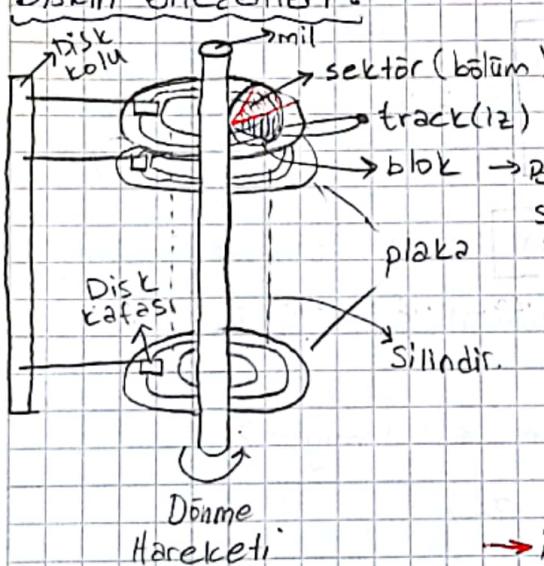
oldukça yavaştır. Genelde data arşivlemek için kullanılır.

→ Artık kullanılmıyor.

Diskler:

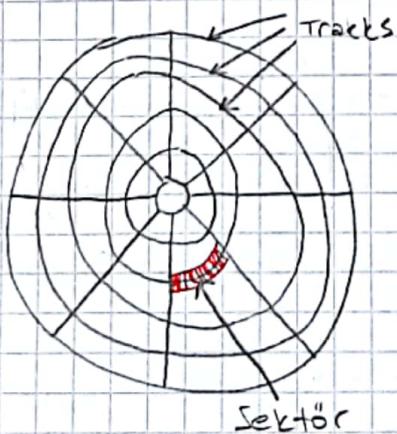
- İkincil Saklama Birimleridir.
- Datalar, disk bloğu veya sayfa adı verilen birimlerden okunur veya bu birimlere yazılır.
- RAM'den farklı olarak, datanın işlem görme zamanı, datanın diskteki yerine bağlıdır.
- ↳ Bu nedenle, sayfaların diske bağlı bir şekilde yerleştirilmesi, performansı artırır.

Diskin Bileşenleri:



- Plakalar döner (örn: 90 rps)
- Disk kolu hareket ederek, disk <sup>sanıgedeki
dönme 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.

★



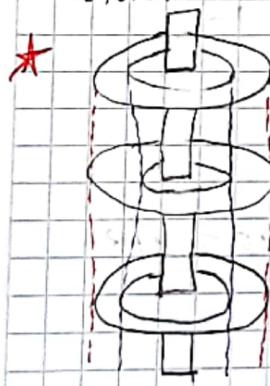
- Disk, merkezleri bir olan tracklerden oluşur.
- Trackler, sektörlerde bölünmüştür.
- Sektörler, diskteki adreslenebilir en küçük birimidir.

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 bir trackteki kayıt sayısı
 ↘ 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 ayrılabilir. hoca sabit dedi
- Bloklar sabit veya değişken uzunluğunda olabilir. (?)
- 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): istenen bloğun bulunduğu track
üzerine disk kafasının gelme süresidir. Disk plaqının ağı ile
doğru orantılıdır.

2-) Dönüşel Gecikme (Rotational Delay): istenen 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 istenen 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
şöyleden hesaplanır:

*Erişim zamanı = Arama Süresi + Dönüşel Gecikme + Aktarım Süresi

→ N Blok veriye erişilmek istenirse ;

→ Ardıtek yerleşmisse :

| Erişim süresi = Arama (Seek) + Dönüşel (rotational) + N * Aktarım (transfer)

→ Rastgele yerleşmisse :

| Erişim Süresi = N * (Arama + Dönüşel + 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 bulmustuk (soru 2).

$$\boxed{\boxed{512 * 50 * 2000 * 10 \cong 512 \text{ MB}}}$$

↴
 Bir yüzeyin boyutu
 Diskin boyutu

Soru 4-) Kaç silindire sahiptir?

Diskteki silindir sayısı bir yüzeydeki track sayısına esittir

$$\boxed{2000}$$

Soru 5-) Hangisi genelikle blok ^(bloklar sektörlerden oluşur) büyüklüğüdür neden?

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

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

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

$$\frac{1}{5400} \text{ dak} = \frac{1}{5400} * 60 \text{ sn} = 0.011 \text{ sn} = 11.1 \text{ milisaniye}$$

max rot. del

Bir track'in basında seviriye gelme süresi.

$$\text{ortalama} = \frac{1}{2} * \text{max} = \frac{1}{2} * 11.1 \text{ msn} = 5.55 \text{ msn}$$

60 sn'de 5400 dönmüş x'inde $\frac{1}{60} = 0.016666666666666666$ dk.

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

$$\text{bit Track boyutu} = \left\{ \frac{512 * 50}{11.1} = 2306.3 \text{ byte/msec} \right\}$$

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 bloğa ulaşma zamanı

$$\{ 10 + 5.55 + 100 * 0.44 = 59.55 \text{ msec} \}$$

↓ ↓ ↓
seek rot transfer

Soru 10-) Rastgele 100 bloğa ulaşma zamanı

$$\{ 100(10 + 5.55 + 0.44) = 1599 \text{ msec} \}$$

Örnek :

$$\text{Kitap} = 1000 \text{ sayfa}$$

$$\text{Sayfa} = 500 \text{ kelime}$$

$$\text{Kelime} = 8 \text{ harf}$$

$$\text{Harf} = 1 \text{ Byte}$$

CD-ROM = 600 MB, ise bir CD-ROM'a kaç adet kitap sığar?

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

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

————— 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 dağıtmaya.

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

→ Bu teknikte "round-robin" algoritması kullanılır. Bu algoritmda; 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 hatayı beraberinde getirir. Veri güvenliğinin sağlanması için onun çok sayıda kopyasının oluşturulması gereklidir. Bir disk arızası durumunda kopyalar kullanılarak veri kurtarılabilir.

→ Kopya oluşturulurken yazma işlemi olacağinden 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 $\frac{8}{4} = 200\%$)
 - 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.
 - $\text{max aktarım oranı} = \text{bir diskin aktarım oranı}$ (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, bölünenek 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 yapılabılır. (Blok olduğu için)
- Yeni Parite = (Eski Parite $\xrightarrow{\text{blok}} \oplus$ 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 efektiftir)
- 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 artınca performans da artar.

* Level 6 atlandı. (Read-Solomon algoritması %66 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)

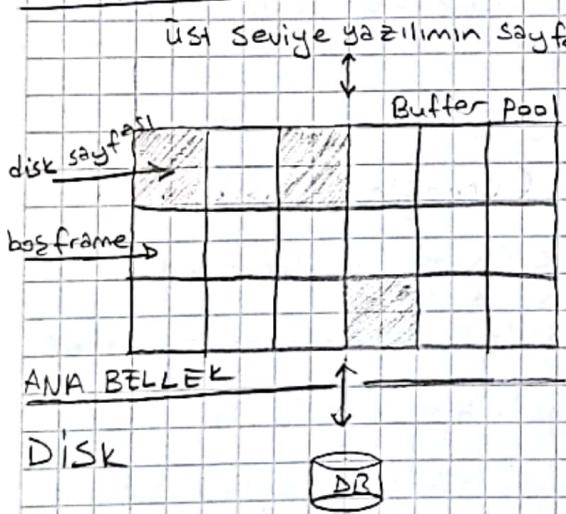
En üst seviye güvenlik için Level 6 tercih edilmeli.

Disk Alan Yönetimi

- Veritabanı yönetim sistemlerinde, disk alanını yöneten yazılımdır.
- Sayfa bazında okuma, yazma, yer ayırma/birakma işlemlerini gerçekleştirir.
- Sayfaların disk üzerindeki düzenli yerleştirilme talebi, disk üzerindeki düzenli bölmelerin tahsis edilmesiyle karşılanmalıdır. Daha yüksek seviyeli katmanların bu işin nasıl yapıldığını 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 össürir.)
- * Bellek, sayfa boyutüğünde parçalardır. (Her frame'e bir sayfa sığar)
- * Parçaların oluşturduğu yapıya tampon havuzu (buffer pool) denir.

* DBMS'nin 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 boşaltılıp 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'e yeni sayfayı oku. (Yarı 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 yapıyorsa) bazı sayfalar çağırılmadan (istek yapılmadan) getirilebilir (bu işlem pre-fetching denir). Zamanın 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)

*
*
* ÖZET *
*

Veritabanı yönetim sistemleri, büyük boyutlu veritabanları üzerinde işlemler yapabilirler. Veritabanı, içinde büyük miktarda verinin saklandığı bir dosyadır.

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

Öğrenci işleri bilgi sistemi üzerinde bir öğrenci kaydını yapmak istediği;

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 alanı 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 boşluga 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 kimin 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 yeni 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 yeni blok yazılmak istenirse öncelikle eski blok diske yazılır daha sonra üzerine yeni 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 yapılmırken #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 yazıları sayfanın (+) yerine yazılır.

11	2	3	4	5
6	7	8	9	+

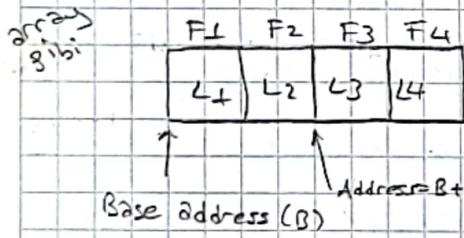
Bu durumda ardışık okuma yapılacağı zaman 1. sayfaya ihtiyaç vardır. tekrar LRU kullanarak 2 yerine + yazılır. Bu kez de 2 tablodan kaybolur. Onu getirmek için 3 yazınca 2 yazılır. Bu böyle devam eder.

MRU kullanılsaydı sadece sağ alt değişecekti. (11 ve 10)

DBMS vs İşletim Sistemi Dosya Yönetimi:

- DBMS yazılımları, işletim sisteminde bağımsız olmalıdır. (Tercih edilebilir olmalıdır)
- İşletim sisteminin dosya sisteminde dosyalar diske dağıtılmaz. Yani dosya boyutunu, disk boyutlarından küçük olmalıdır.
Fakat DBMS'de dosyalar bir diske sığamayacak kadar büyük olabilir. Bu durumda dosya, disklere dağıtilır.
- DBMS'de buffer yönetimi bu nedenle yapma becerisi gereklidir.
 - Buffer pool'a bir sayfa, birleştirmeye ve bir sayfa diske yazılmasına zorla.
 - Replacement policy'li döşemek, pre-fetch işlemi.
- ! DBMS'de dosyalar, kayıtlardan (records) meydana gelir.

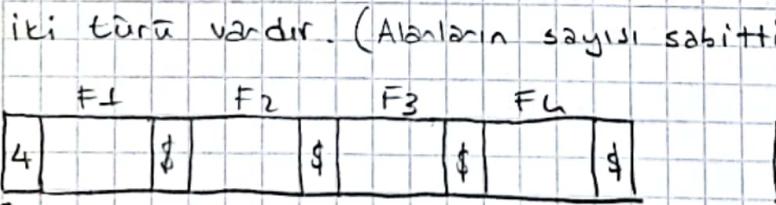
Kayıt Bicimi : Sabit Uzunluk (Record Formats: Fixed Length)



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

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

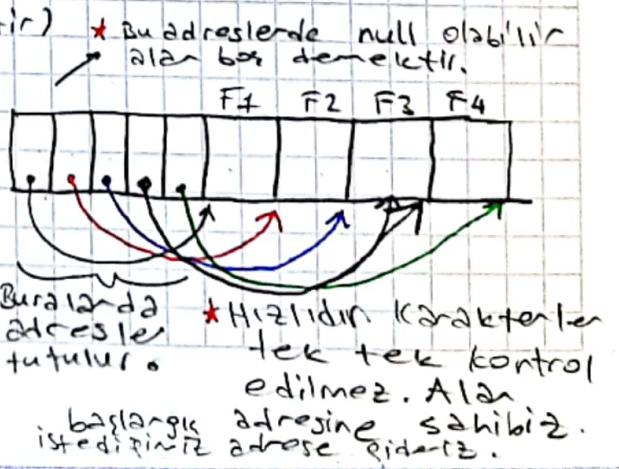
Kayıt Bicimi: Değişken Uzunluk (Record Formats : Variable Length)



* Alanın sonunda özel karakterler vardır. (Alanın bitişini gösterir)

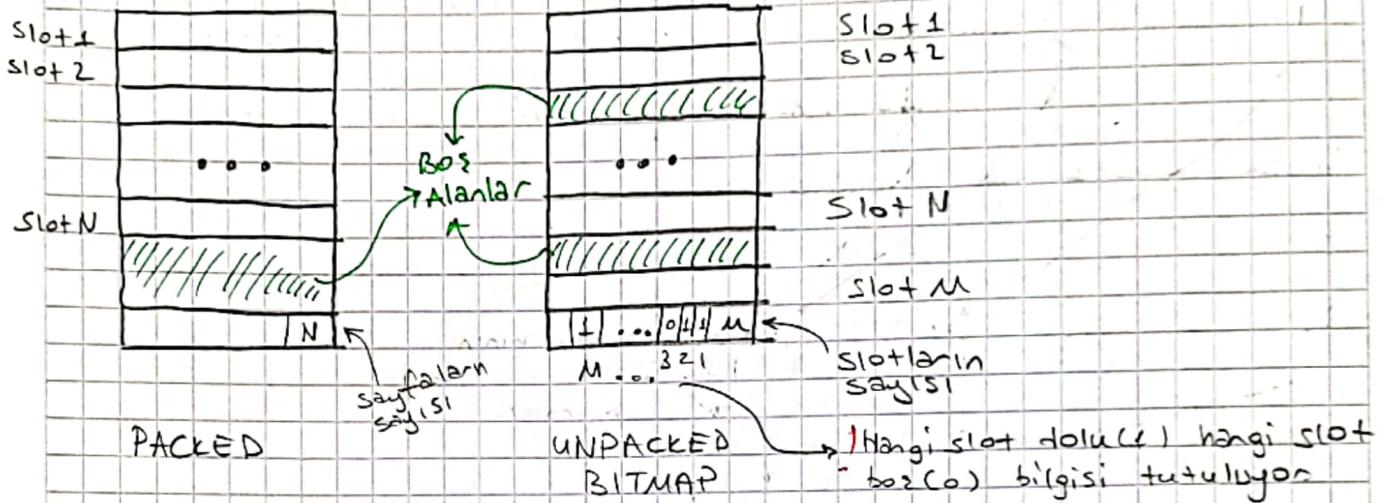
(Bu metod yavaştır)

(Her karakter tek tek bize (dai) için)



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

Geçen 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 alandan 1 slotluk yer alınır ve N + artar.

★ Yeri bir kayıt ekleneceği zaman slotların bilgisinin tutulduğu kısım (en alt) kontrol edilir. Eğer 0 olan (yani boş) bir slot varsa oraya yerleştir ve o slotun bilgisi + olur.
Silineceği zaman, o slotun biti, o yapıyı (record id değişmez)

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

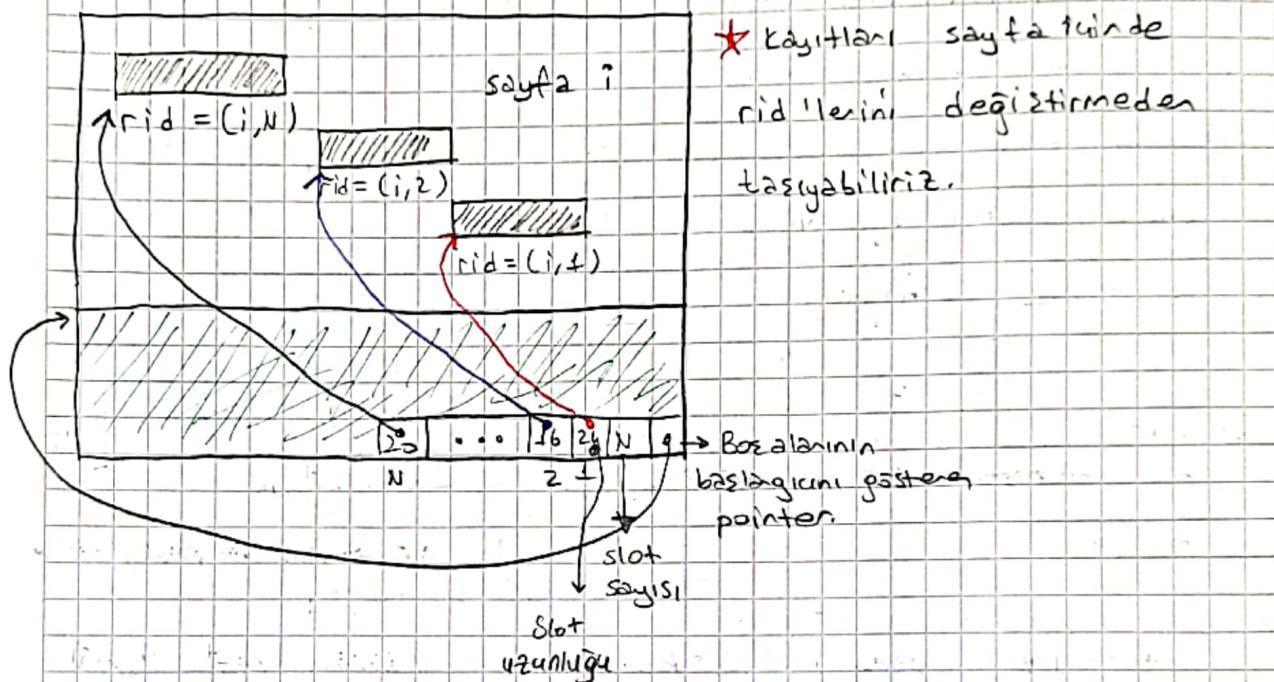
★ record id = < page id, slot # >
(kayıt id)

ilk (packed) sekuencte bir kayıt silindiği zaman sonraki kayıt onun yerine geleceğinden record id verme konusunda sıkıntı oluyor.

Örneğin 2. slot silindiğinde N. slot onun yerine gelecektir. rid
(rid=52) (rid=5N)
pageid

unique olmalıdır. Bu bir probleme yol açabiliyor.

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

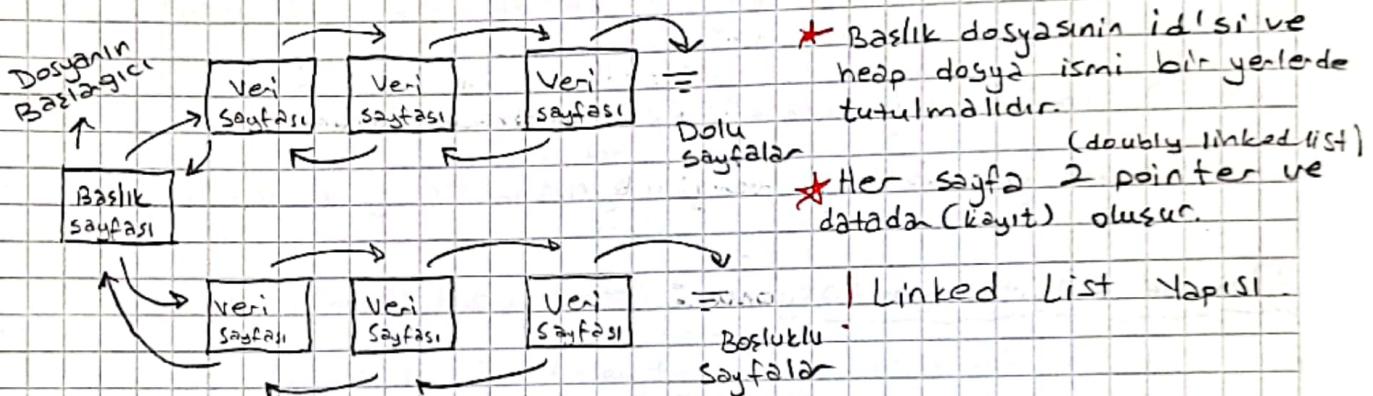


Kayıt Dosyaları: (Files of Records)

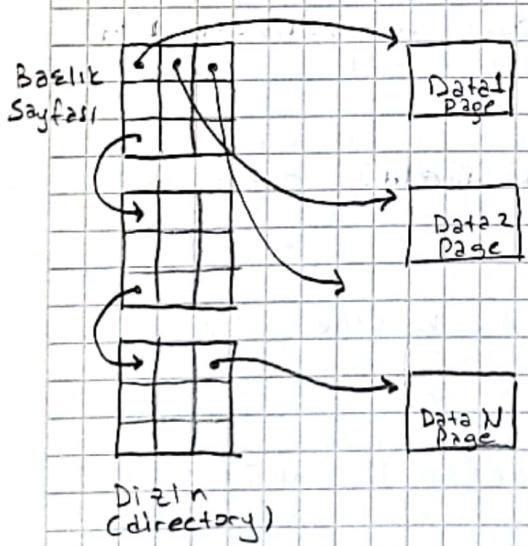
- DBMS diske veri yazma, diskten veri okuma gibi işleri yapar. Fakat daha yüksek seviyeli DBMS katmanları kayıtları ve kayıt dosyalarını yönetir.
- Her dosya sayflardan, her sayfa kayıtlardan oluşur.
- DBMS sunları desteklenelidir:
 - Kayıt ekleme / çıkarma / değiştirme
 - belirli bir kaydı okuma (record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

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

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



Sayfa dizini kullanan yığın dosyaları :



Her dizin bölünmüş vaziyette bulunur. Bu bölümlerde data sayfalarının adresleri ve o sayfa tarafından ulaşılabilir boş alanları tutar.

Yeni bir kayıt eklenecinde, basılıc sayfasından başarak dizin aranır. Yeterli büyüklükte bir kayıt alanı bulunduğuunda yerleştirilir. Örneğin; Data 2 sayfasında boşluğ varsa 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 hızlıktır. (boyutlarla)

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 Anahtarı : (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, yazısında göre aranız iste bunlar
 ikincil anahtardır.

! Geçer olarak her alan bir anahtar değildir. (Anahtarlar, bir aramada
 kullanılacak alanlara veya alanların bir kombinasyonunda 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 (küçük alan büyüklüğünü veildi)	★ Büyüük alanların sayısını yazmak için fazla bayt gerektir
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 | Kullanilmaz
03818 | Folk | File Structures | Kullanilmaz

2) Sabit kayıt depīken alan

Değişken Uzunluklu Kayıtlar (Variable-length records):

Sabit Alan Sayısı:

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

→ Burada her kaydın 3 alandan oluşan olduğu 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 almıyor. Diğer kayda geçilmek istendiğinde bu, digerine göre daha hızlıdır.

Kayıt Yapılanının Karşılaştırılması:

Tip	Avantaj	Desavantaj
Fixed-Length	* 1inci sıradaki kayda girmek kolaydır. (üçüncü kayıtla aynı altta)	* Boşluk (padding) yüzünden yer kaybı.
Variable-Length	* Yerde tasarruf sağlar (kayıt boyutları farklı olduğu için)	* Bir index dosyası olmadıkça 1inci kayda ulaşamazsın (index dosyası hafṭaya)

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 seyre göre sıralandır.)
(sorted)

Yığın Dosyaları: (Pile Files)

- Bir yığın dosyası, herhangi bir ek yapı olmaksızın, basitçe birbirini ardına yerleştirilen bir dizi kayıttır.
- Kayıtların uzunlukları değişebilir.
- İstener kaydı bulmak için ilk kayıtten başlayarak kayıt bulunana kadar ardışık (sequential) arama yapılır. (Buffer pooldaki 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 1. sırada olabilir.
- En kötü durumda aradığımız kayıt sonuncu (b 'inci) sırada olabilir.
- Ortalama durum. $\rightarrow \frac{1}{b} * b(b+1)/2 \Rightarrow \underline{\underline{b/2}}$
b farklı $1'den b'ye$ kadar olası durum olan tüm bloklar.

* Sıralı olmayan bir ardışık dosyada bir kaydı bulmak için ortalama süre:

$$T_f = s + r + (b/2)^* btt$$

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 tamsayı 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 frame'in içine aktarılır. (Sonuncu sayfada boş yer olduğunu varsayıyoruz) bu sayfaya yeni kayıt ekler. Bu işlem için gereken süre :

$$T_I = s + r + btt + (2r - btt) + btt$$

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

Sonuncu bloğu okumak için.
 $\Rightarrow s + r + btt + 2r$.

yazma
süresi

? 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 alır. 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 :

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

frame
içinde yazmanın
süresi

frame
içinde yazmanın
süresi

Once güncellmek istediğimiz kaydı buluruz. Bu kaydın olduğu bloğu buffer pool'a alırız. Buradan 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 yazi'ye 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.

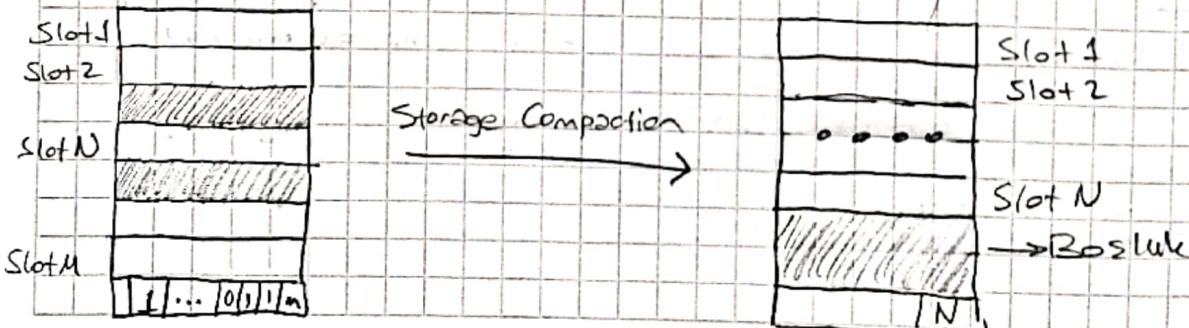
1-) Kayıt Silme ve Depo Sıkıştırma : (Record deletion and Storage Compaction)

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 silindikten 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 işleminden sonra 70 bloğa sağlanır.

Storage Compaction : (Depo Sıkıştırma)

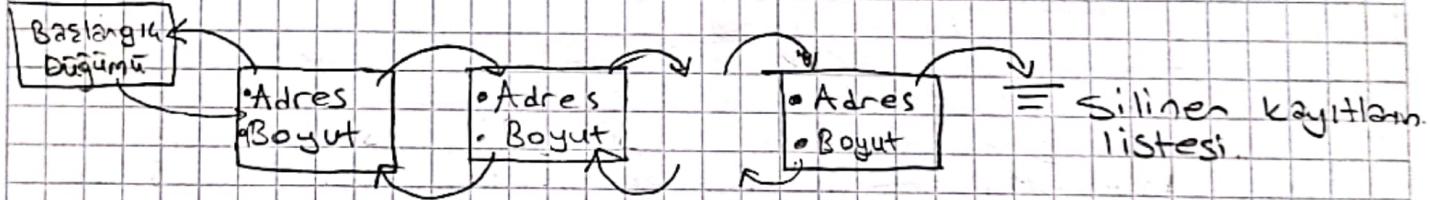


2-) Sabit Uzunluklu Kayıt Silme : (Deleting fixed-length records and reclaiming space dynamically)

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. Yerini bir kayıt ekleneceği 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 silmede, 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 adres'e gidip kayıt eklenir. Daha sonra o node listeden çıkarılır.

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

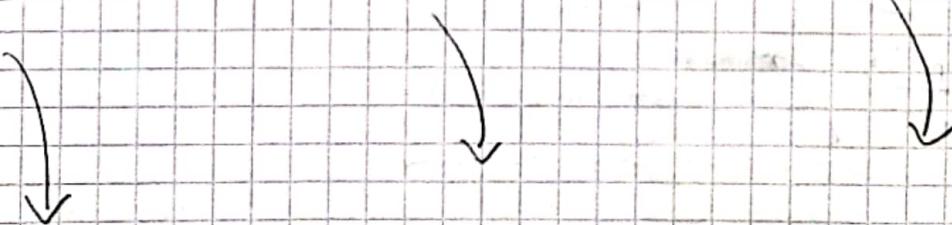
! Bazı araştırmalara göre worst-fit'in daha iyi olduğu görülmüştür.

PROBLEM 1

→ (original dosyada)
b tane bloktan oluşan bir dosya olsun. Bu dosya üzerine (strage 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 + b (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 :

$$\left\{ \left[\frac{n}{Bfr} \right] * (s+r+btt) \right\} \text{ Yazma Süresi}$$

üstte yararıza ($\frac{10}{2} = 5$ çıkar)

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

→ ii) Eğer iki disk sürücüsü varsa. (original dosyanın 1. sürücüde (drive) olduğunu varsayıyorum). 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 disktedeki okuma süresi:

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

İkinci diske ardışıl yazma süresi:

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

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

Toplam Süre

PROBLEM 2:

A ve B yiğin(pile) dosyalarıdır. Her ikisinde de 100.00 tane kayıt bulunmaktadır ($n=100.000$) ve bu kayıtlar herhangi bir sıralanmaya göre sıralanmamış. A ve B dosyalarındaki kayıtların yaklaşık $\frac{1}{2}$ 'si 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 (yani C bloğunu boşalt)

* Burada kayıtlar herhangi bir kriter'e 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
sayfasının boyutu
- 1 frame, ortak kayıtları yazmak için kullanılır.
- Geriye kalan 4164 frame A'dan blokları okumak için kullanılır.
- Bir sayfada kaç kayıt var?
 $bfr = \left\lfloor \frac{B}{R} \right\rfloor = \left\lfloor \frac{2400}{400} \right\rfloor = 6$
- Her bir dosya (A ve B) kaç sayfa (blok) 'tan oluşur?

$$\left\lceil \frac{n}{bfr} \right\rceil = \left\lceil \frac{100.000}{6} \right\rceil = 16667$$

↑ kayıt sayısi ↑ sayfa var 1 dosyada.
 ↑ sayfadaki kayıt sayısı

- A dosyasını baştan sondan 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:

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

→ random disk erişimi.

↳ 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 biriminini 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 taşma alanına (overflow area) ekleriz.

- Taşıma alanında kayıtlar sıralı bulunuyor. (Eksendiğin
sıradı bulunuyor.)
- Bu tarz bir dosya yapısında kayıt sırasında 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
eklemeden sonra tekrar düzenleme yapılarak kayıtlar
sıralı hale getirilir.

Kayıt Arama : (Searching for a record)

Sıralı ardışılı dosyada arama yaparken, sıralı kismın için
ikili (binary) arama, overflow kismı 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

ortadaki
kayıt test
olmak kabul
edilir.

$$TF = \log_2 X * (str + btt)$$

→ Tüm overflow (taşıma) kismını aramak için :

$$TF = S + r + (y/2) * btt$$

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

PROBLEM 3:

Blok boyutu = 2400 byte

Dosya boyutu = 40MB

Blok aktarım süresi (btt) = 0.84 ms

S = 16 ms

r = 8.3 ms , olsun.

1-) Pile file için arama süresi ve sorted file için (overflow olmayan) gerek süreleri hesapla. (En kötü durum için) $\left(\frac{40 \text{ MB}}{2400 \text{ byte}}\right)^{\log_2 10^4}$

Pile File için TF : * ortalamaya göre $S + r + b * btt$ olurdu.

$$TF = S + r + b * btt = 16 + 8.3 + \left[\frac{40 \text{ MB}}{2400 \text{ byte}} \right] * 0.84 = 14024.5 \text{ ms}$$

$$\approx 14 \text{ saniye} //$$

Sorted File için TF :

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

$$\approx 0.38 \text{ saniye} //$$

2-) 10000 kayıt aranırsa :

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

* 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 içeriin.
- 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 hesap no'ya göre sıralanmıştır. }
- Transaction file: Hesap güncelllemeleri (hesapno, tred, borg, 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:

- Həydi buffer pool idarəəi baxıa bir C bloğuna yax.
- C dolduğunda, C'yi diskə yax (yəni bozəlt)

Else-if r_A'nın keyfi r_B'nin keyfindən kiçikse:

- A'nın sonrasi kaydına git
- Adım 3'e döñ

Else-if r_A'nın keyfi r_B'nin keyfindən böyükse:

- B'nin sonrasi kaydına git
- Adım 3'e döñ

Else:

Eğer A'dan gelen tüm kayıtlar kontrol edildiyse:

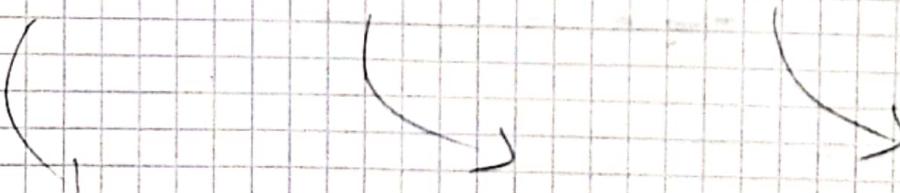
- Sonraki M bloğu A'dan al
- Adım 3'e döñ

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 üzərində yalnızca 1 defə gedilir. File file olunca f'dan okunan her M blok iñin B dosyasını baştan sona təkrar okumak gerciyordu.

Sorunun Sayısal Gözünü



• 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 (Bozuluyor).

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

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 :

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

İçerisinde dosya klasörleri de okunur.

* Bu üç zamanı toplayarak toplam zamanı buluruz.

Eş Sıralı Tıgın Güncelleme Algoritması : (Algorithm for Co-Sort Batch Update)

Örneğin; Banka sisteminde on binlerce müsteri olabilir. Bu müşteriler aynı anda işlem yapabilirler. Yapıları her işlemi hemen ardından veri tabanına doğrudan yansıtırıksak bu işlem çok yavaş olacaktır. Çünkü bu güncelleme DB'ye yansıtıldığında ilgili blok üzerine bir kilit konulması gereklidir. İlgili bloktaki bir başka müşterinin bilgileri varsa o kilit sülhane kadar (okuma - yazma bitece kadar) beklemesi gerektir. Bu da çok uzun zaman alır bir işlemidir. Bu yüzden güncelleme işlemleri belirli bir sayıda güncelleneden sonra toplu olarak 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 :

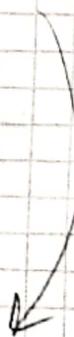
Pointerler her iki dosyanın da ilk kaydını gösterir.

Pointerler dosyanın sonuna gelene kadar:

Kayıtların keylerini 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 ayıren 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 /yeri M. file'nin sonuna kopyala.

Else:

HATA. (ünüt olmayan bir banka hesabı üzerinde işlem yapılmıyor)

Bir sonraki kayda git (T. file iin)

- Eğer master key = transaction key :

Eğer güncellene yapılmışsa:

Yeni kayıt, yeni dosyaya yazılır.

Eğer silme işlemiyse:

Hiçbir şey 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 düşünübiliriz.
- 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) en kötü durum olsun
aranan kayıt sonda ise → Bu dosya m parçaya bölündüp
parçalar tek tek buffer pool'a aktarılacak ve gezi diske
(memory) 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	Sıra数
17 LON 2312	9. Sinfoni Beethoven Giulini	
62 RCA 2626	Romeo ve Juliet Prokofiev Maazel	
117 WAR 23689	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 23689	117

→ Verilen anahtarla göre ibili arama yapılır. Bulunan kaydın adresine gidilerek diskten okunur.
* (Alfabeye göre sıralandı)

Indexed Filelerde 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 tetraf 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, machineyi 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 çağırılır.

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

	Etket ID	Başlık	Sözler
12	LON/2312	9. Sonfoni Beethoven	Giulini
62	RCA/2626	Pomeo ve Juliet Prokofiev	Maazel
117	WAR/23699	Nebraska	- - -
152	ANG/3795	Keman Konertosu	- - -

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 yapılır. Bu işlemleri diskte yapmak oldukça (shift) maliyetlidir. Çünkü diskte kaydırma işlemi yapılmaz. 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 işlemlerden sonra bu kaydın anahtar ve adres ikilisini key index'ler'de silmemiz gerekiyor. Index'ten silindiğten sonra da sırayı bozmamak için silinen elementden sonraki elementleri birer satır yukarı kaydırarak gerekiyor. Index, RAM'de bulunuyorsa bu işlem kolay. Diskte maliyetli. Bu yüzden bazı sistemler index'ten silerkten adres kısmına silindi anlamına taşıyan bir işaret kayar. Örneğin; -1

→ Bir kayıt güncelleneceği zaman, pile file'de bu kayıt güncellenir. → (silindi tekrar eklenir.)
Güncellenmeden sonra kaydın key'i aynı kalıyorsa (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, Besteçi, Sanatçı ...

6 Bunun gibi ikincil anahtarlar elde edebiliriz...

Pile File ve İkinci 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 kişinin bestelediği birde fazla eser olabilir.

→ sec. key'ler kendi içinde primary key'e 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 key'i ve adresi primary key index'e eklenmelii. Daha sonra primary key ve sec. key'inin sec. key index'e sıralamaları bozmayacak şekilde eklenir.

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

→ Bir kayıt güncelleneceğinde, güncellene türne göre işlen yapılır:

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

↳ 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 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, bestekar ismi vb göre) (Aranmayı kolaylastırmak için)
- Bu ikincil anahtarları "and" ya da "or" ile izlenece sokarak aranan esyleri bulabiliyoruz.

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

Besteciyle eşleşen kayıtlar	Başlıkla eşleşen kayıtlar	Bu ikisiinin "ve" işlemi
ANG3795	ANG3795	ANG3795
DG139201	COL31809	DG18807
DG18807	DD18807	
RCA2626		

* Bu primary keyler bize Beethoven tarafından bestelenmiş ve başlığı 9. Senfoni olan tüm kayıtlardan

Bu primary keyler ile primary key indexten adresleri alır, diskteki ilgili adreslere giderken aradığımız kayıtları bulmuş oluruz.

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
Etiket+ID Lистesi Dosyası	

* Sec. key index'in boyutunu kısaltmak amaçlanır. Böylece her sadece RAM'e sağlanabilir (Hızlı arama için)

* Örnek bir akış: Beethoven'in eserlerine erişim şu şekilde olur. Sec keydeki adres alınıp listede o adresle gidili. Burada prim key ve adres tutulur. -1 girené 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). 4. adresin değer -1 yerine 8 yapıılır. 8. kaydın adres kısmının -1 yazılır.

CHAPTER 10:

- 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.

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

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

→ Bir diğer problem, dosyaya yeni veri ekleyip sildiğimizde index'i de uygun şekilde güncellememiz 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 ardı ardı 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 bunu toplu biçimde (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 biçimde tutuluyor.

Index Yapılarında Giriş : (Introduction to Index Structures)

Kitapta, indexlere 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 id'değer } sec key
- k değeri ile anahtar değeri k olan kayıtların, kayıt id'lerinin listesi } (mesafe)

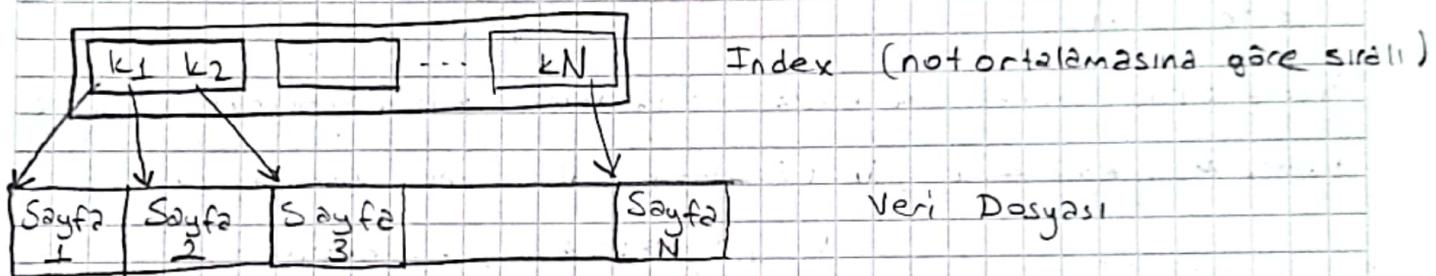
★ Ağacı yapılı dizinler range search ve equality search'ü destekler (esitlik)

★ 2 tane ağacı yapılı dizin gereceğiz. ISAM (statik yapılı) B+ tree (dinamik). Bu iki yapı da günümüzde DBMS'de sık sık kullanlan veri yapılarıdır.

Range Searches :

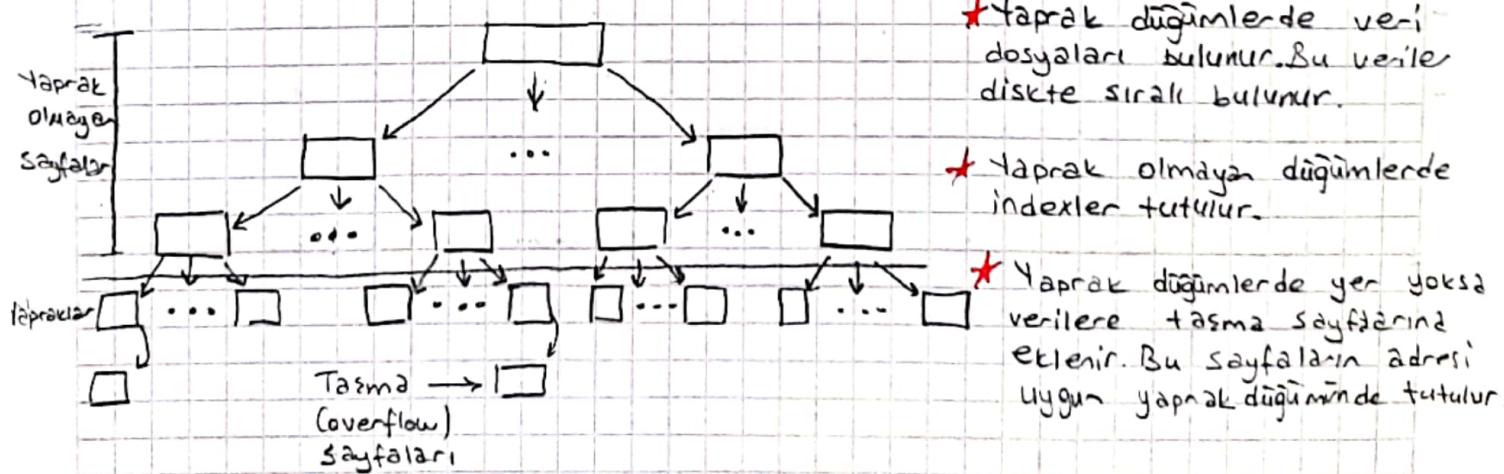
Indexli ardışılı dosya yapısında hem aralık sorgularını hem de eşitlik (range) sorgularını çok etkin ve hızlı bir şekilde yapabiliyoruz. Aralık sorgusuna bir örneği :

- not ortalaması > 3.0 olan öğrencileri bul.



→ Index üzerinde, not ortalaması 3.0'in 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, taşıma 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. Yeni bir data ekleneneğinde yaprak düğümde yer yoksa overflow düğümleri allocate edilir.

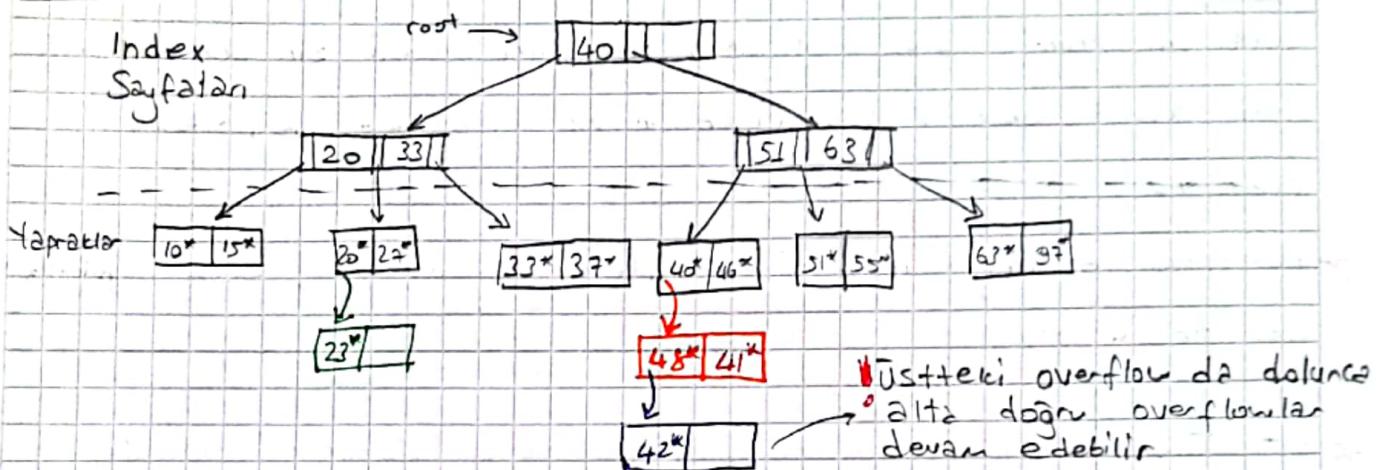
Arama : root düğümünden başlar anahtarları karşılaştırarak yönünt bulur. Maliyeti = $\log_F N$ ($F = \text{fanout}$, $N = \text{yaprak düğüm sayısı}$)
 (Ortalama
cocuk sayısı / index sayfası sayısı)

Ekleme : eklenen yaprakken öncelikle arama yapılarak uygun yere gelinir. Yaprakta boş yer varsa direkt olarak yaprak düğümne veri eklenir. Aksi hálde 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 silindiğinden sonra element kalacaksa ve veri yapraktaysa sadece silme işlemi olur.

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

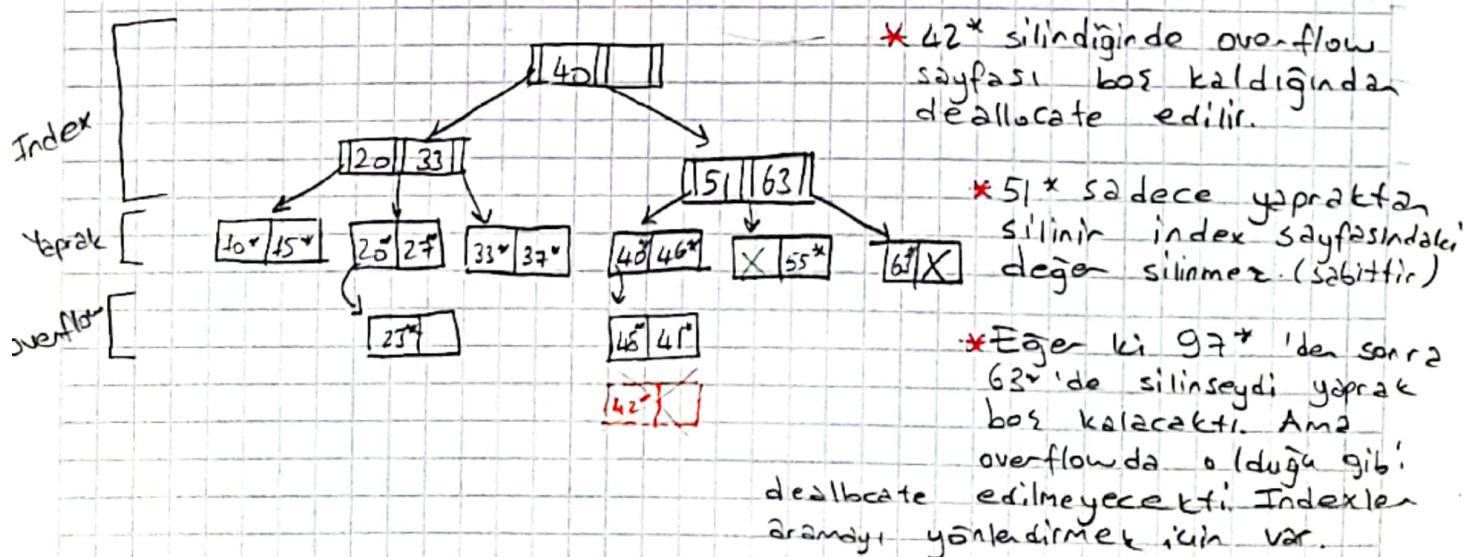
Örnek : ISAN'a 23^* , 48^* , 41^* , 42^* ekle (sırayla)



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 olmadığında overflow allocate edilir ve 23^* değeri oraya yazılır. Bu overflowun adresi de 20^* yaprakında tutulur.

* Overflow → random access, yapraklar → sequential access.

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



B+ Tree : (Most Widely Used Index)

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

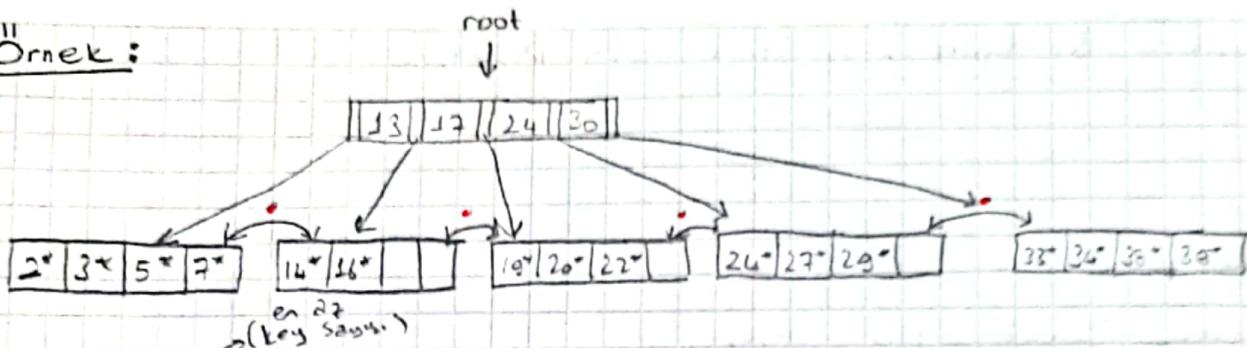
- * Ağacın arama, ekleme, silme işlemlerinin maliyeti, ağacın yüksekliği ile doğru orantılıdır. Bunun maliyeti $\log_F N$ ($F = \text{fanout}$, $N = \text{yazılım sayısı}$, $\log_F N = \log_2 N$)
- * B+ ağacı oluşturulurken, kök düğüm hariç, bütün düğümlerde en az ' m ' tane eleman olmalı. Eğer ağacın 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. (Ardısal durumunu bozmamak için)

Örnek:



- Bu örnekte order yarı $d=2$. 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'un solundaki adres alınır ve o düğümde sırayla arama yaparak 5* değeri bulunur.
- 15* değeri aranırken: root'a batılıp gerekli karşılastırımlar yapılır. 13-17 arasındaki adres'e gidilip burada arama yapılır. Overflow olmadığı ve veriler düzenli olduğu için yalnızca bu düğüme batılarak bu eleman yoktur denebilir.
- $\geq 24*$ aralığını ararken: root'a batılıp gerekli karşılastırımlar yapılır. 24-30 arasındaki adres'e gidilerek şartı sağlayan en küçük değerden başlanarak dosyanın sonuna kadar gidilir. Böylece şartı sağlayan tüm değerlere ulaşılmış olur.

* * *
 * * * Yaparak düğümler, disktte 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 keyi var bir düğünde)
Ortalama

Doluluk oranı = %67

Ortalama fanout = 133 olsun.
(çocuk sayısı)

Bu durumda ;

4.yükseklikte : $133^4 = 312,900,900$ kayıt

3.yükseklikte : $133^3 = 2,352,637$ 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.

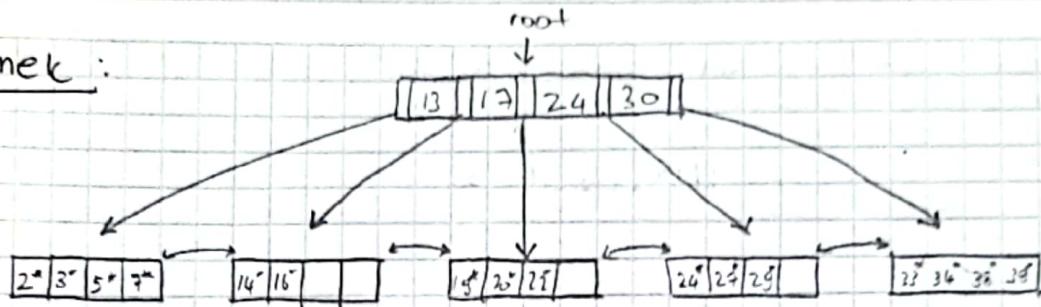
Else :

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

(Eşit yer en ve ortadaki keyi yukarı kopyala)

* Ağacı bölmek (split), ağacı büyütür. Eğer root'u bölersen
ağacın yüksekliği artar.

Örnek :

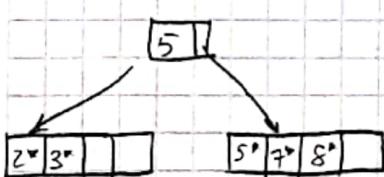


→ Bu ağaca 8^* eklenmesi istenirse :

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

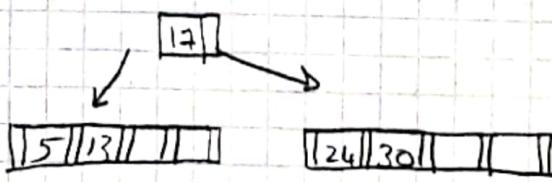
- İlk 'd' tane kayıt (bu örnekte $d=2$) aynı sayfada kalır, diğerleri yeni sayfaya eklenir ve ortadaki değer bir üst seviyeye kopyalanacak. (Bunun yanında yeni sayfanın adresi de bir üst düğüme kopyalanır) Yani 2* ve 3* aynı yerinde kalacak, 5*, 7* ve 8* yeni sayfaya gidecektir ve 5 ile yeni sayfanın adresi bir üst düğüme gönderecek (ortadaki eleman)
- Yukarıda da yer olmadığı için (5'li 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



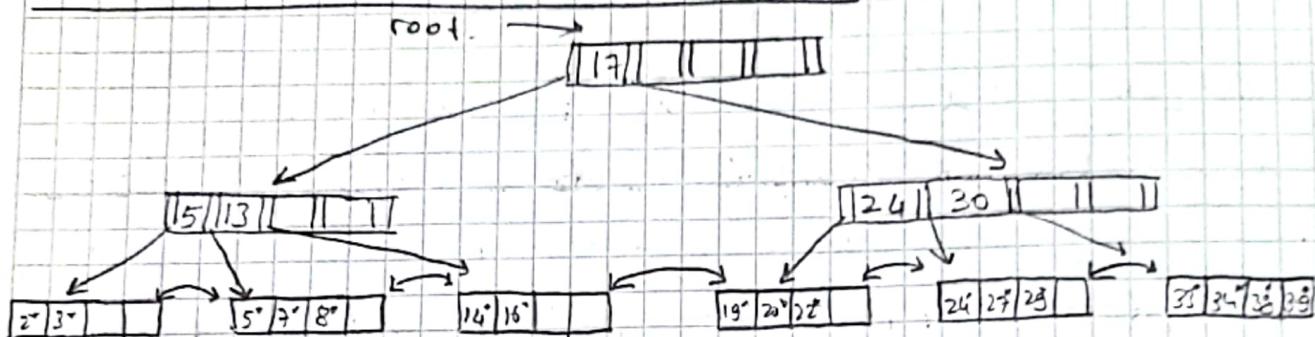
I. Ortadaki değer indexinde tane
taşındı

II. adımda



II. Ortadaki değer bir üstte tasiındı.
root bölündüğünden yükseltik 1 arttı.

Bu işlemlerden sonra ağacın son hali :



B+ Ağacında Veri Silme :

Silinerek veriyi bul. (L düğümünde olsun veri)

Veriyi sil.

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

Silme işlemini bitir.

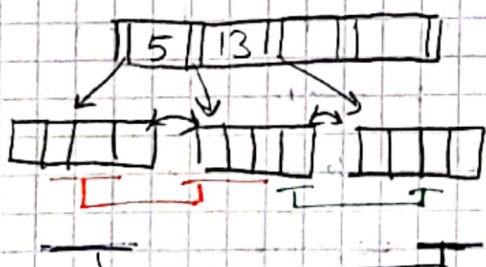
Eğer silmeden sonra d - 1 veri kalmırsa düğümde :

Komsudan bir eleman al ve üsteki indexi güncelle.

Eğer komsuda d + 1'den daha az eleman varsa :

L düğümünü ve bunun kardeşini birleştir.

* Kardes düğüm derken su kastediliyor.

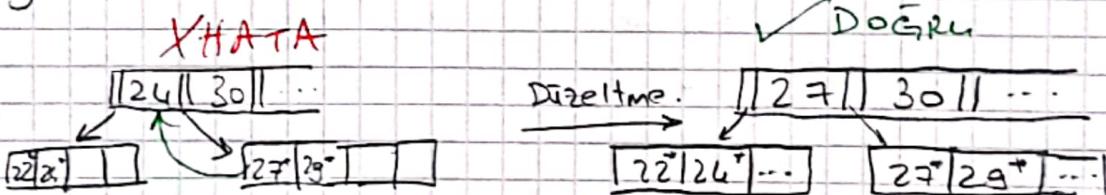


! Kardes olmaları için aynı indexin sağına ve soluna yerlesmeleri gereklidir. Örneğin turuncu ve yeşil yerler kardeş fakat mavi kılımdaki düğümler kardeş değil.

Örnek: Buradaki BT ağacı baz alınarak 19° , 20° ve 24° 'ü sil.

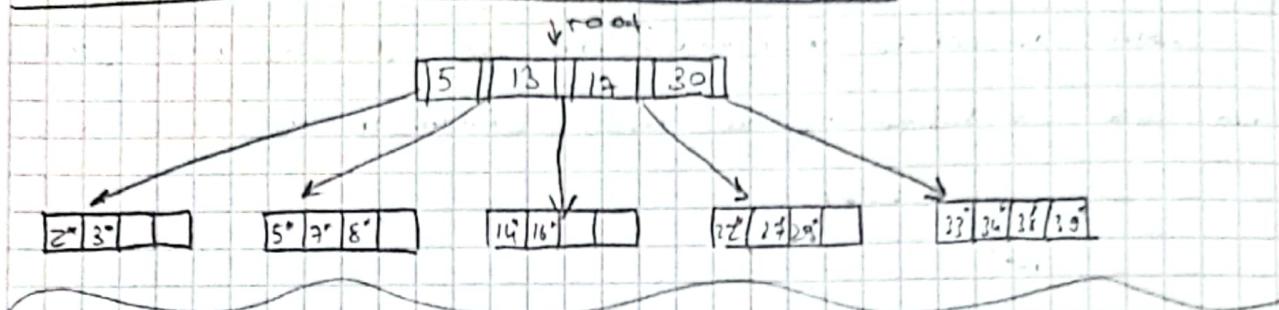
→ 19° lu silmek için: Öncelikle 19° ağacına bulunur ve silinir. Silindiğten sonra eşiğin BT olma özelliğini koruyor mu diye bakılır. Silinen sayfadaki kayıt sayısı en az d (bu örnekte $d=2$) tane olmak zorundadır. 20° ve 22° nin olduğu görülür ve 19° silindiğinde işlem biter.

→ 20° silindiğinde: Aynı sayfada 1 tane kayıt kalıyor. Bu durum BT ağacı olma özelliğini bozduğu için öncelikle kardeşinden 1 tane elemen ödünç alıyoruz (24° 'ü). Kardeşten ödünç almak için kardeste en az 3 tane kayıt olması gerekiyor. Görüldüğü üzere bu şart sağlanır ve 24° değerini sayfamızda alırız. Bu işlemlerden sonra komşu sayfadaki en küçük kayıt anahtar olarak yukarı yollanır.

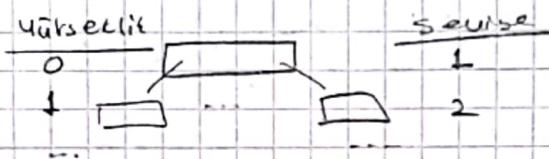


→ 24° silindiğinde: Silinen sayfada 1 kayıt kalıyor. Komsudan da alamıyoruz çünkü komsuda $d+1$ (3) kadan kayıt yok. Bunun için silinen sayfa ve komşu sayfa birleştirilir. Sayfa birleştirilirken bir sayfa silinir. Bir sayfanın silinmesi demek bir üst düzeyde 1 index ve 1 adres değerinin de silinmesi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

Bu işlemlerden sonra ağacın son hali :



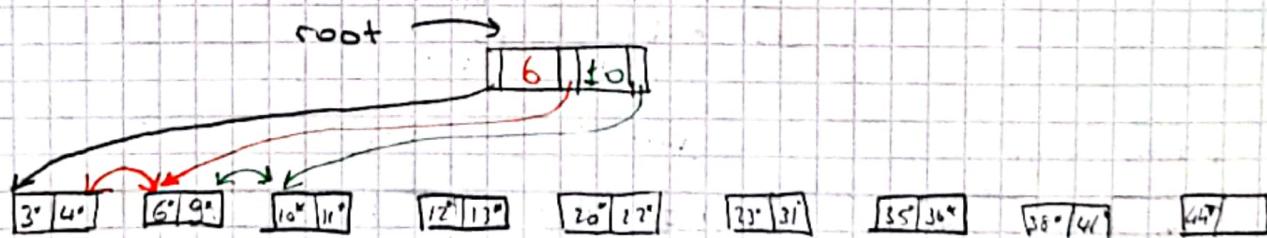
* Bu ağacta arama maliyeti $\log_F N$ dır. Bu da B+ ağacının (ortasına çok sıkıştırılmış) yüksekliğini azaltır ve daha hızlı bir aramaya olur.



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

Bir dosya için, sıfırdan bir B+ ağacı yaparken; kayıtları tek tek birbir bir B+ ağacına eklemeye zorluklarla遭遇する。Bu algoritma söyle çalışır: öncelikle eklemek istenen kayıtlar, anahtar değerine göre sıralanır, ilk sayfanın adresini yeni oluşturulan kök düğümde saklıyoruz. ikinci sayfadaki en küçük değerini kök düğümde ekliyoruz ve 2. sayfanın adresini yukarı taşıyoruz. 3. sayfanın en küçük değerini kök düğümde ekliyoruz ve 3. düğümün adresi de root'a eklenir.

(2 değer tutan düğümler için gerekli)



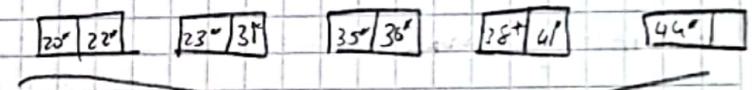
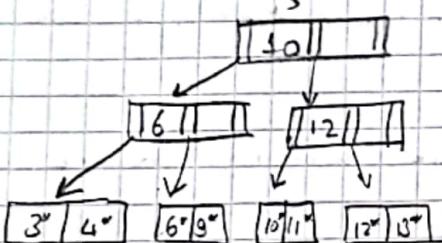
I. adım

II. adım

III. adım

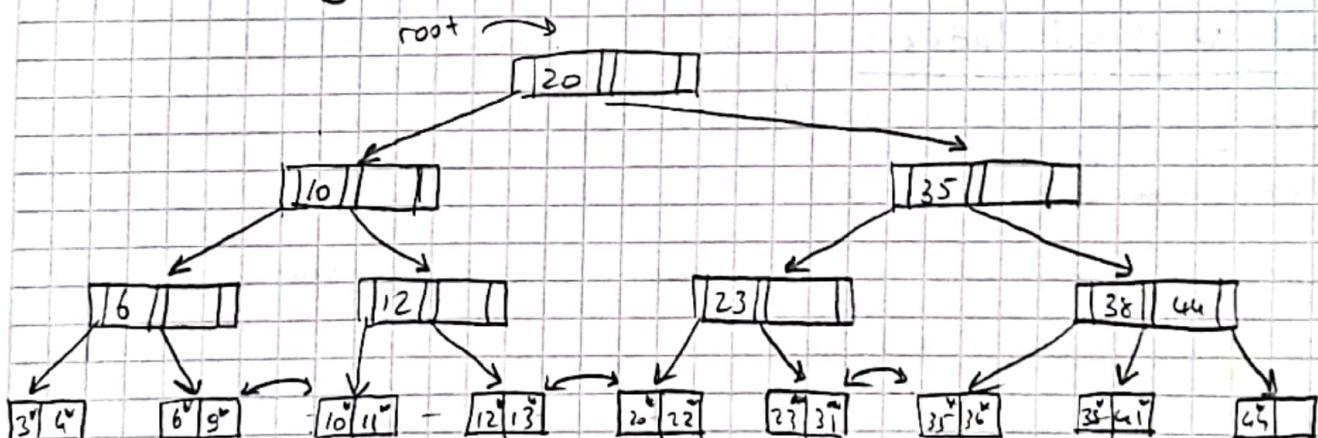
* Artık root'ta yeterli yer yok. 12' ile başlayan düğümü eklemek için mevcut root'u bölgüyoruz. İlk d+ane key aynı yerde kalır. Ardakta ($d=1$ bu örnekte) yukarıya gider ve kalan d+ane yeni sayfaya gider.

root



Henüz etkilemediyor.

Tüm işlemler yapıldıktan sonra ağacın son hali



* Fotokipediki açıklamayı da okuyabilisin

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

HAFTA IV. SON

→ HW2. pdf'yi incele. (örnek soru ve çözümleri)

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

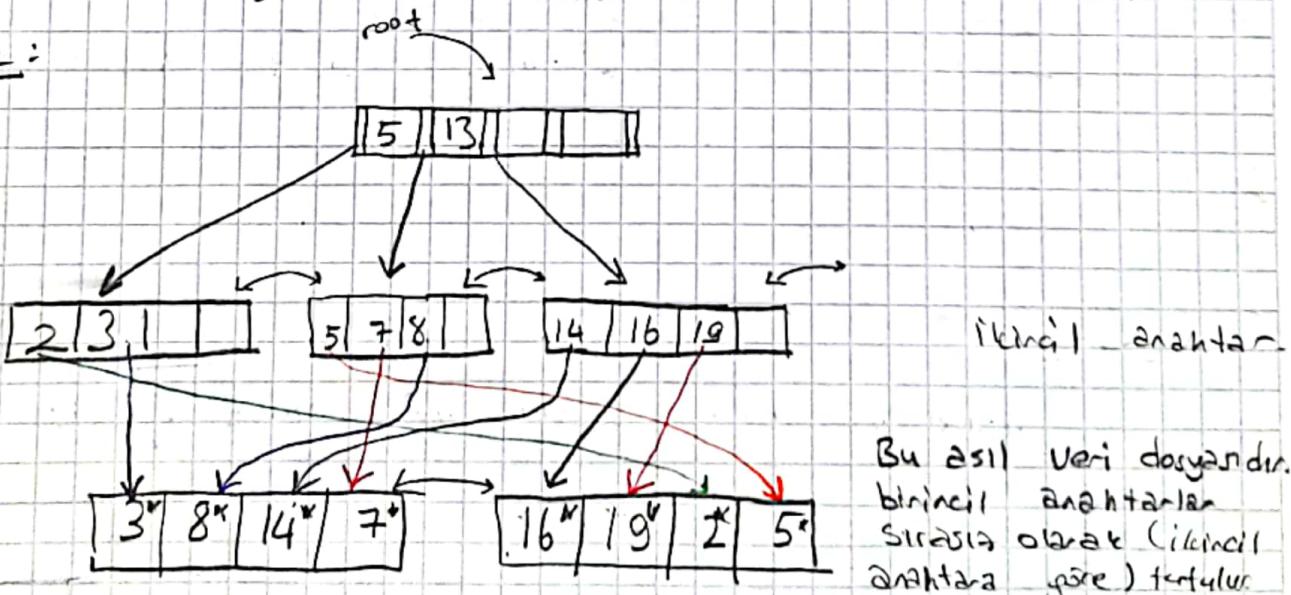
B+ ağacı ve ISAM'de order kavramı önemlidir. Çünkü, B+ ağacının bir düğümünde en az 'd' en fazla '2d' tane kayıt ya da anahtar bulunmalıdır. Yalnızca kök düğüm bu kriteri uymayabilir. (d'den daha az sayıda anahtar bulunabilir) d sayısı, ağactaki düğümlerin doluluk oranının en az %50 olmasını garanti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

- * Yapraklardaki kayıt büyüklüğün, 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 sıralanmamıssa 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+ ağacı varsa : $O(\log N)$ veriyi arama süresidir. Index sayfları RAM'de tutuluyorsa veri ağacına aranıp bulunduktan sonra diske tek bir erişimle veriye ulaşabiliriz.
- 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+ ağacından daha iyi). Fakat aralık (range) sorgularını desteklemez.
- * Hashing yönteminin de statik ve dinamik versiyonları var. (Tipki B+ 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^+ versisi buckete yazılır.
- ★ Hash fonksiyonunun, anahtar değerleri bucket'lar üzerine düzgün dağıtması genetikir. Aksi halde $O(1)$ 'dan $O(N)$ 'e doğru evrilir.

Tasarım Faktörleri = (Design Factor)

Bucket Büyüklüğü = Aynı adres'e 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 olduğu bilgisini verir.

Tümleştirme Tekniği = Bir adres'e gönderilen kayıt sayısı
(Overflow Resolution Technique)
ordaki bucket'a sigamayacak şekildeyse overflow gerçekleşir. Uzun overflow zincirleri, olnca performans kötüleşeceğinden birtakım algoritmalar kullanılarak bu sorun üstünlmeye çalışılır.

Hash Fonksiyonları: (Hash Functions)

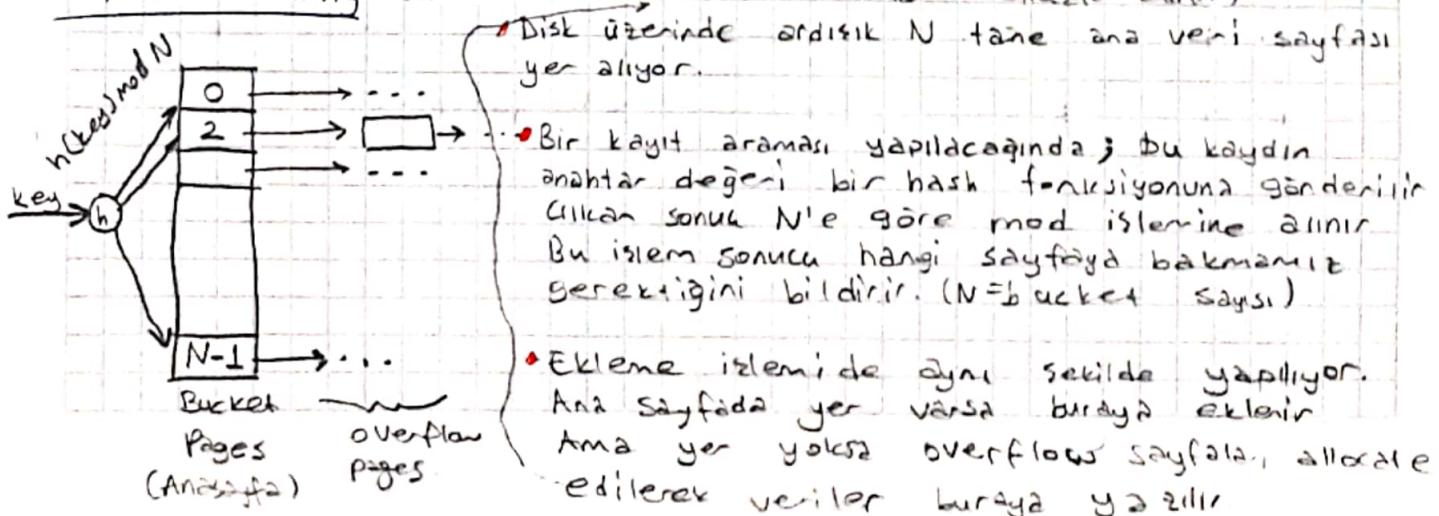
- Key mod N Yöntemi: En çok kullanılan yöntemdir. $N = \text{bucket sayisi}$. N sayısı asal sayı olursa daha iyi performans sağlanır.
 - Uzun integer'lardan oluşan anahtarlar için aşağıdaki yöntemler uygundur.
- Folding Yöntemi: 123456789 bize anahtarımız olsun. Bu anahtar, 123 | 456 | 789 diye ayrılır 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 \leftrightarrow tabanında bir sayı gibi (taban dönüştürücü) ele alınır. Daha sonra bu sayı \rightarrow tabanına çevrilip truncation yöntemi uygulanır.

Static Hashing: (ISAM'e 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 gerecelli adreslerdir.
- h fonksiyonu kendisine gelen anahtar, aynı zamanda döndürüyor.

→ Sırayla 12, 35, 44, 60, 6, 46, 57, 33, 62, 17 anahtarlarının, ekleme

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

Birincil
Alan

overflow

* Veriler birincil alanda saza, arama yaparken tek disk erisimi ile o veri bulunabilir. Ama overflowda ise daha fazla disk erisimi yapılır.

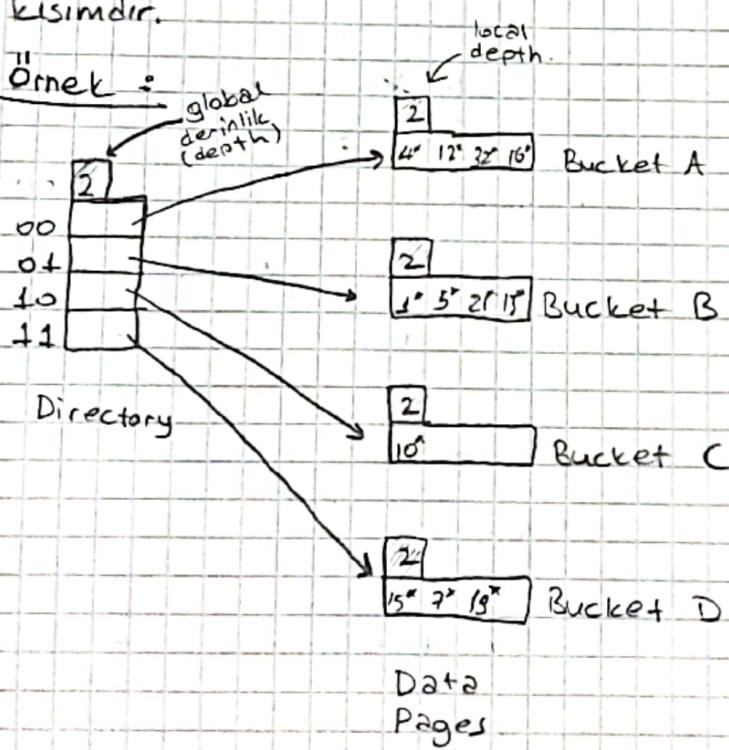
Örneğin 17'yi aramat için, 17 hash fonksiyonuna sokular ve mod N yapılır. Sonuç = 2. 2. adresi gidilir bir disk erisimi ile. Burada yoksa bir sonraki sayfanın (overflow sayfasının) adresi alınır ve bu sayfa için diske bir daha erisim yapılır.

Extendible Hashing :

- Static hashing yetki overflow sayfalarını ortadan kaldırıyor, ama uygun dinamik bir yapıdır.
 - Dolu olan sayfaya bir kayıt eklenmek istendiğinde sayfa ikiye bölünür.
 - İki kısımdan oluşur. Dizin (directory) ve veri sayfaları (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 anahtarini hash fonksiyonuna gönderiyoruz. Ucuk sonuc binary'e geliriyor. Sonrasında, bu binary sayının sondan global depth kadar bite bakiyoruz. Örneğin; Yeni kayıt eklenecek ve hash fonk'tan üçük sonucun binary halinin sonu 00. Bu kayıt bucket A'ya eklenir.

★ Local depth, o kayittaki sayıların sondan kaç bite baki olarak buraya yerleştirildigini bize söyler.

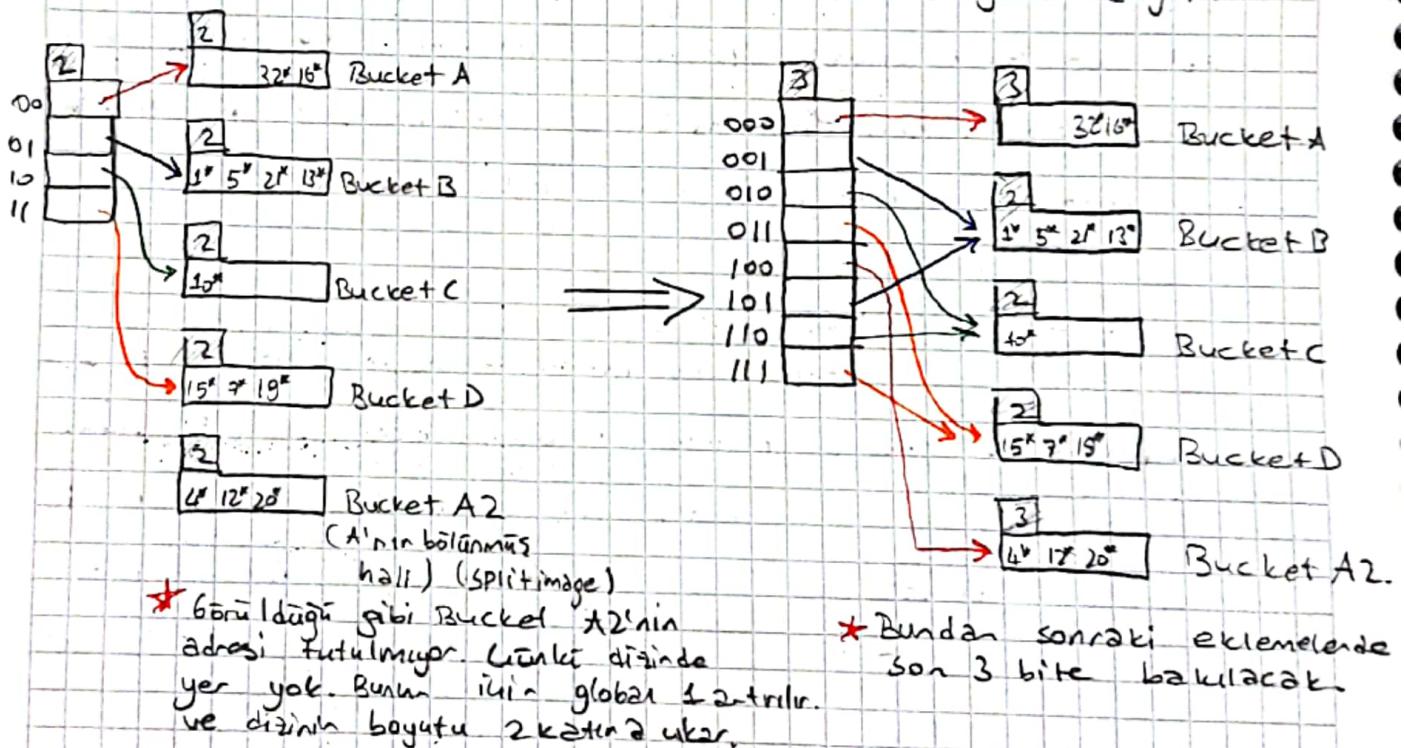
★ Global ve local depthleri kıyaslayarak bir sayfada ne zaman overflow oldugunu buluyuz. Overflow olan sayfa 2'ye bölünür ve local depth'i 1'artır. (Yani 2^n 2 bit yerine son 3 bite baki olarak yerlestirme gerekli)

overflow sonrası
local > global
oluyorsa
2'ye böl.

Örneğin 20 eklenmek istendiğinde :

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

Son 2 bite bakılınca 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'ncı döngü gibi Bucket A2'nin adresi tutulmuyor (ünkisi dizinde yer yok). Bunun için global 1 artırılır. 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 fonksiyonunun binary'e çevrilir 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 sağlanamayacak kadar büyüyebilir ve directory diske yazılabilir (RAM'e sağlanamadığı 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 sağlanır.
- ★ Silme işlemi yapılırken, silmek istenen verinin anahtarını hash fonk. sonucu binerye 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. (B+ ağacı gibi). Birleştirme sonucu her bir bucket'lu 2 ismetci gösteriyorsa, global depth 1 arttırılarak dizin yapısının boyutu yarıya indirilir.
- Fakat pratikte, yukarıda bahsedilen birleştirme ve dizin boyutunu yarıya indirme işlemleri yapılmaz. Çünkü iteride veri eklenirken bu işlemleri için vakit kaybı olacağından (yani tekrar kümülatif teteras büyük gibi) yapmak istemez.
- ★ Hash based index'te aralık sorgusunun yapılamamasının nedeni, verinin sıralı olarak yerleştirilmemesidir.

Linear Hashing :

Dinamik bir hash yapısıdır. Extendible hashinge alternatif olarak geliştirilmiştir. Overflow zincirlerini ortadan kaldırmayı amaçlar. Etkielli gibi bir directory kısmını yaratır. 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 bölünecek 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 sonrası 2. kısım)
- henüz split edilmemi.

* 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 maide mi olduğu belirlenir.

* Yeni bir kayıt eklendiğinde, h_{level} veya $h_{level+1}$ 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 olusmamış olur.

Örnek:

- Level = 0 → (h1'de gerek bakılacak)

- $N=4$ (Bucket sayıısı)

- Her bucket 4 kayıtlıdır. → (Ters döner sırasıyla bitirilmesiyle bağlantılıdır.)

Level = 0

		43 eklenmek istenirse										
		(101011)										
h1	h0	32°	44°	36°		000	00	32°			Birimci sayıları	Overflow
000	00	32°	44°	36°		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°	43°
011	11	31°	35°	7°	11°	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 ilerlege gider.

★ h0 için Next = 3 olduktan sonra bir daha overflow gelirse, h1'e gelir ve Next = 0 olup, başa döner. Bucket sayısı = 8 (level 2) olur.

★ h1 için Next = 7 olduktan sonra overflow olursa h2'ye gelir ve Next = 0 olur. Bucket sayısı = 16 olur (level 3)

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

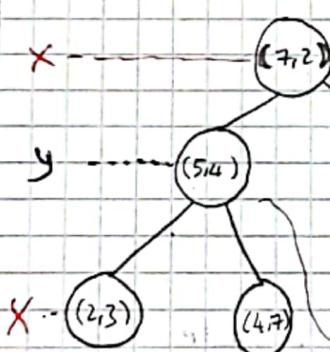
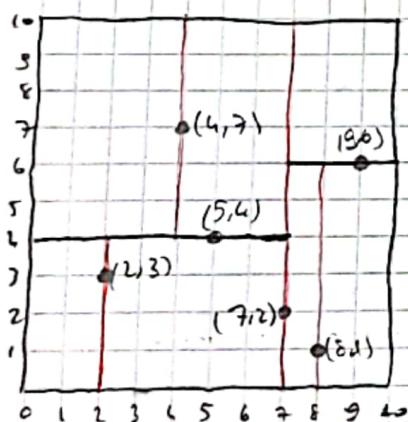
- * LH aslında EH'nin bir varyasyonudur.
- * EH'de, bir dizin yapısı vardır. Bu yapı bucket'ların adresini tutuyor. Bir kayıt eklenirken ilgili bucket'ta overflow oluyorsa ve dizinde yeteri kadar yer yoksa, dizinin boyutunu iki katına çıkarıp overflow olan bucket'i ikiye bölayoruz.
- * LH'de dizin yapısı 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 bölündüğü zaman aslında directory'li iki katına çıkarılmış gibi oluyoruz.

HAFTA II. SON

- K-d Trees -

- ★ K-d ağacı (k boyutlu ağac), primary ve secondary keylerin birlikte kullanılan 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 nitelikle göre yapıyorduk. Fakat k -boyutlu ağacta k tane nitelikle göre arama yapılabilir.
- ★ ikili arama ağacının genişletilmiş bir versiyonu diyebiliriz. (Burada da aradığımız değer küçükse veya eşitse sola, büyükse sağa gitceğiz.)
- ★ k -boyutlu ağacta, her düğümde tek bir değer yerine bir vektör bulunur. ($k = \text{boyutunda}$)

Örnek :



★ Kök düğümde karşılaştırma yapılırken x doğrine göre karşılaştırılır. (Yani x koordinatı 7'den küçük olan bir değer aranırsa sola, büyük olan aranırsa sağa gitilir.)

İkinci seviyede y koordinatına göre arama yapıyormuş. (Yani soldaki düğüm için y 'yi 4'ten büyükse sağa, küçükse sola x hesabına katılmamış.)

★ Kırmızı çizgiler x boyutundaki, mavi çizgiler y boyutundaki bölgeleri gösterir. İlk bölüme $(7,2)$ den olmustur. Onun için x kütlesi 0 va.

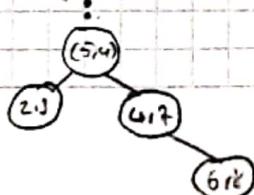
→ Ağaca $(6,8)$ eklemek istenirse:

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

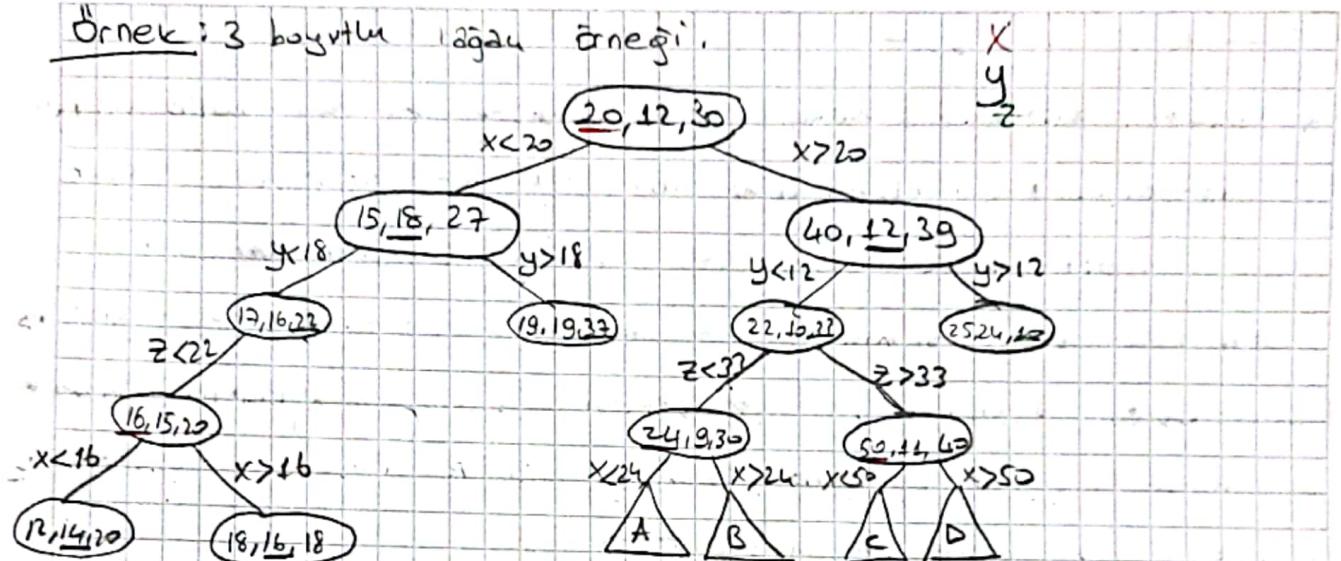
Seviye 2: y 'ye göre karşılaştır. (Sağa git)

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

Seviye 4: Buranın sağına yerles.



Örnek: 3 boyutlu lagau örneği.



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

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

B: $x > 24$, $y < 12$, $z < 33$
 (her 20 hem 24'te büyüğebilir,
 sayı kesinlikle 24'ten büyük olur.)

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

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

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

* A, B, C ve D'nin x, y, z 'sinin belirlenmesi için önceki dallanmalara bakınız. Örneğin A'nın bulunduğu konum en üstte $x > 20$ şartıyla ayrılmış ve A'dan bir önceki adımda $x < 24$ denerken tekrar ayrılmış. Bu iki değer dikkate alınarak A'nın x 'deleri 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 (ortanca 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 kullanılırsa düğümlerde veri noktalarının kendisi bulunur. mean yöntemi kullanılırsa 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 DÜĞÜMÜNDEN başlayarak eklemek istenen düğümleri boyutlarının değerleri karşılaştırılarak sağa veya sola gidilir. Yaprak düğümde ulaşılınca buranın sağına veya soluna eklenir.
- * Yukarıdaki gibi ekleme yapmak ağacın dengesiz 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 kullanıp yerlestirme yapılıyor. Bu dengeyi sağlamak için çeşitli algoritmalar oluşturulmuştur. (pseudo k-d tree, K-D-B tree, h3 tree ...)
- * Eğer en baştan medyanları bulunarak ekleme yapılırsa baştan dengeli bir ağac olusur.

Örnek: Pemuk Prencesi ve 7 Küçeler masalındaki karakterlerin isim, boy, kilo ve diğer bilgileri verilmiş olsun. Boy ve kilo verilerini kullanarak bir K boyutlu ağacı üretelim.

İsim	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 hitpi koymağız)

~~Altı çizili kisim şunu ifade eder: eğer bir düğüm bu düğüm ile karşılaştırılıyorsa, bu karşılaştırma boy 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 (38,51)

IV-) Sleepy (36,48)

$h < 36$

$h > 36$

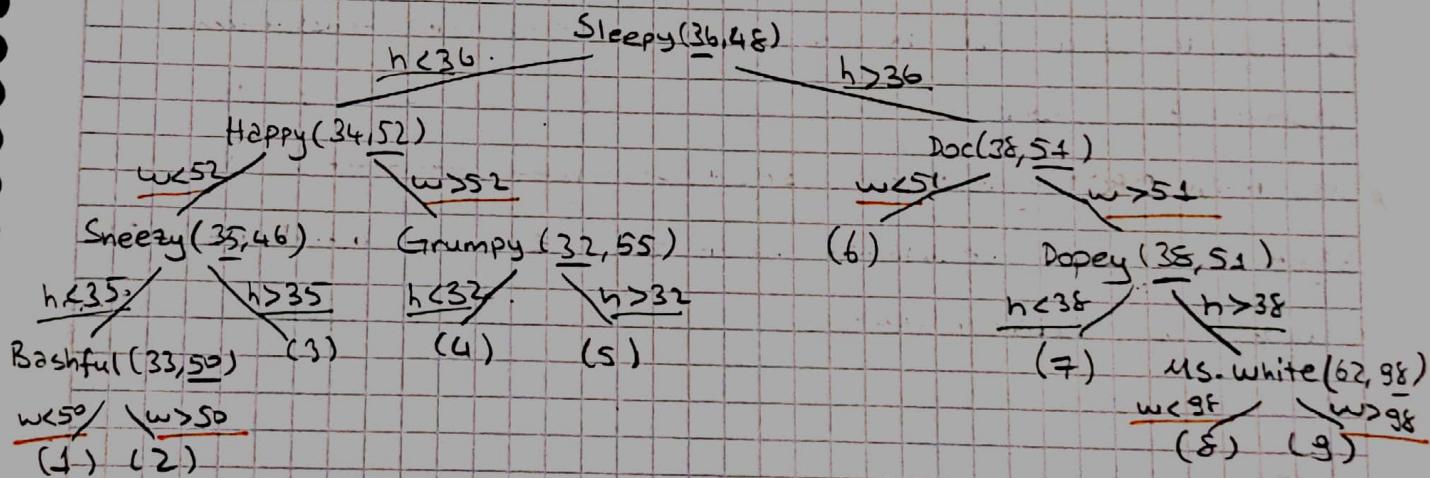
Happy (34,52)

$h < 38$

$h > 38$

Dopey (37,54)

→ Bu işlemler sürdürülerek en son suna ağaca ulaşılır:



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: $32 \leq h < 36, w > 52$ | 6: $h > 36, w \leq 51$

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

Karmaşıklık: (Complexity)

- medyana göre ağac oluşturuluyorsa: $O(n \log n)$ zaman alır
- Dengeli ağacı düğüm ekleterken: $O(\log n)$ zaman gerer.
- Dengeli ağactan düğüm silinirken: $O(\log n)$ zaman gerer.
- Query (sorgu) yaparken (carollık sorusu ise): $O(n^{1-\frac{1}{k}} + m)$
- (m : istenilen düğümlerin sayısı, k : boyut sayısı) Örn. boyu 36'da boyut 3 olanlar için $m=3$, $k=2$ 'dir.)

K Boyutlu Ağacların Kullanıldığı Alanları (Applications)

- Sensör ağlarında soru işlenme (query processing in sensor networks)
- En yakın komşu aramaları (nearest-neighbor searches)
- Optimizasyon (optimization)
- Işın 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 sorulama 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 tutabiliirdik.
 Mesela departmana göre bir hash (esitlik sorusu olduğu için), maaş sorusu için B+ ağacı (aralık sorusu) olabilirdi.
 Böyle bir durumda ; departman kısmından 'oyuncak' getiriliyor,
 maaş kısmından $\geq 50K$ getiriliip kesişimlerini alabildik.

Yöntem 3:

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

Örnek : Hangi index yapısı kullanılabilir?

- 1-) Departman = 'Satış' , Maaş = 20k (Departman → hash, Maaş → hash)
- 2-) Departman = 'Satış' , Maaş > 20k (Departman → hash, Maaş → BT)
- 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 sorulaması yapmak mümkün. Bunun yanında tek bir alana göre de soru yapabiliriz. (Birden fazla alana göre soru yapacağı diye bir kurall yok!)

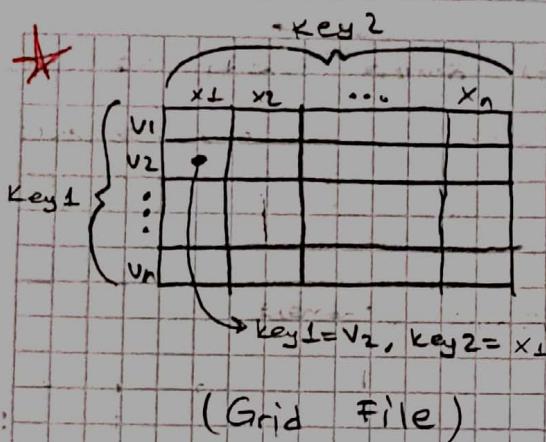
→ Coğrafi bilgi sistemleri için oluşturulan veri kümelerini sorulamak için grid file yapısı çok uygun bir yapıdır.

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

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

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

gibi sorular grid file yapısı kullanılarak cevaplanabilin

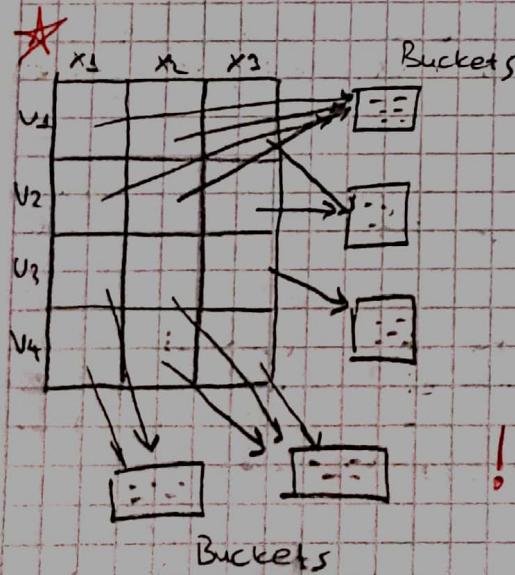
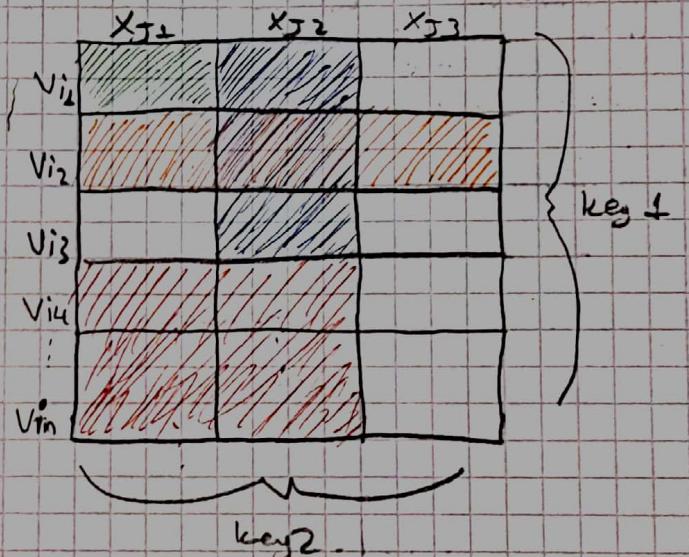


- Bunu aslında içi boyutlu dizi olarak saklayabiliriz. Bu içi boyutlu dizinin her bir elemanında bir pointer tutulur. Bu pointer da o kriteri sağlayan kayıtların bulunduğu bucket'in adresini veriyor.

|| Burada key1 = V2, key2 = x1 dan
 o kayıtların bulunduğu bucketin
 adresi tutuluyor.

→ Bu yapı kullanarak aşağıdaki sorular kolayca bulunur:
 ↗ (kesirim)

- key1 = Vi1 \wedge key2 = xj1
- key1 = Vi2
- key2 = xj2
- key1 $>$ Vi3 \wedge key2 $<$ xj3



• Grid yapısı yalnızca bucket adresini tutar.

• Grid file yeterince küçükse RAM'de tutulur.

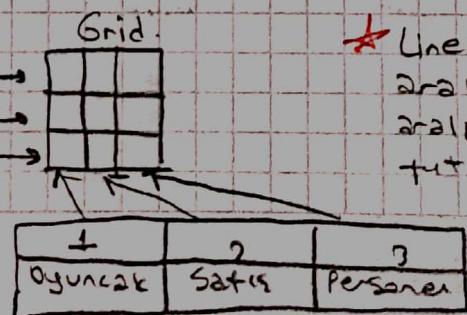
• Bucket'lar diskte saklanır.

! Bu V1, V2, V3, V4 veya x1, x2, x3 kısımlarına bir aralık da yazılabilir. Örneğin,

0-20k	1
20-50k	2
50-∞	3

Maas

↑
 Linear scale →



★ Linear scale bir aralık için kullanılıyor aralığın max değerini tutar sadece.

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

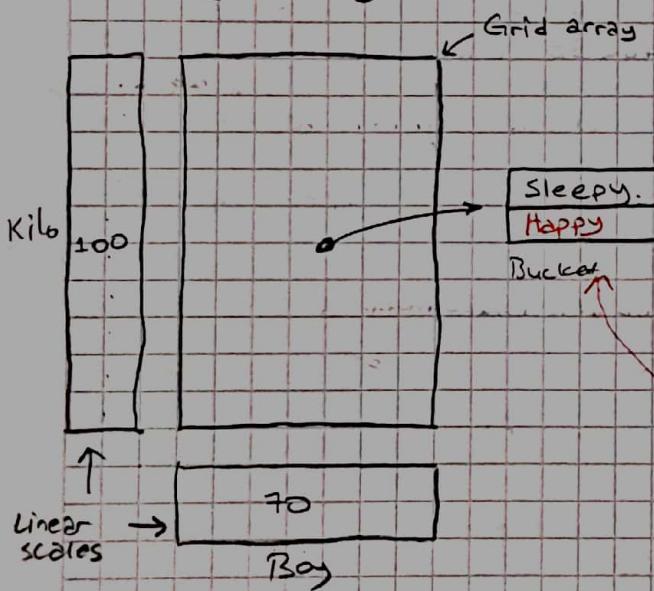
İsim	Boy	Kilo	Diger Veriler
Sleepy	36	48	- - -
Happy	34	52	- -
Doc	38	51	
Dopey	37	54	- -
Grumpy	32	55	-
Sneezy	35	46	-
Bashful	33	50	- -
Ms-White	62	98	

- Bu verileri, verilen sirada 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 :

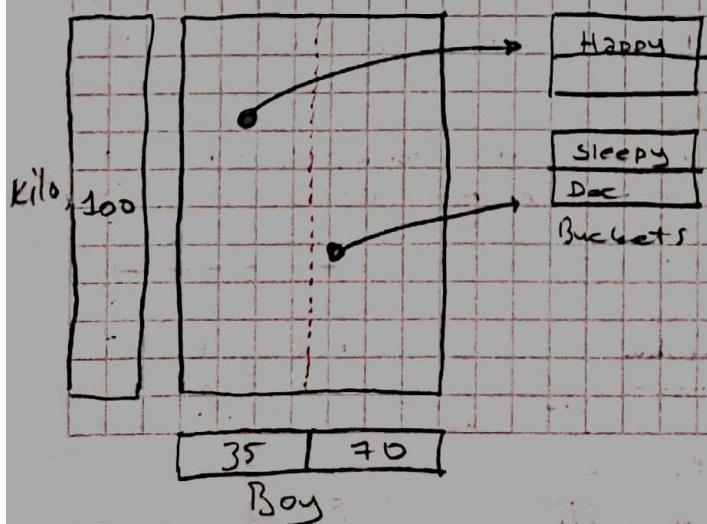


★ Boy ve kilonun üst sınırları belirlenenin türlerin yuvarlanır ve aralık şeklinde linear scale'la yerleştirilir.

★ Sleepy'nin olduğu konumunu ifade eder. Sleepy'nin boyu 0-70 aralığı 6-100 aralığının daddır. (Bu nokta en basit rastgele bir yerde olabilir)

→ Happy'i eklemek istersen gridin gösterdiği bucket'te yer olduğunda bu eklenir.

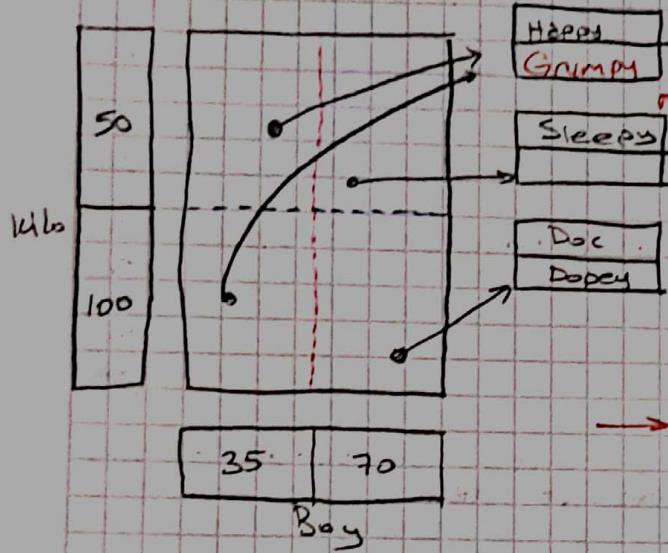
→ 3. kayıt olan Doc eklemek istendiğinde bucket'ta yer olmadığı görülür. Bu sebepten grid 1 boy'dan ilkiye bölünür. (Sırayla pidelerdir)



★ Burada solda kalanlar boyunca 35'den büyük, sağda kalanlar 35 < x < 70 aralığında olacak şekilde düzelenler.

★ Her bir grid bir adres saklar. Burada bölündükten sonra 2 grid 2 adres varken (Sonra yok)

→ Dopey ekleneneceğinde, boyu $37 \geq$ olduğu için Sleepy ve Doc'un olduğu yere eklemeli. Bucket dolu olduğundan bu kez kilo'dan ikiye bölünür. (bir boy, bir kilo, bir boy...)



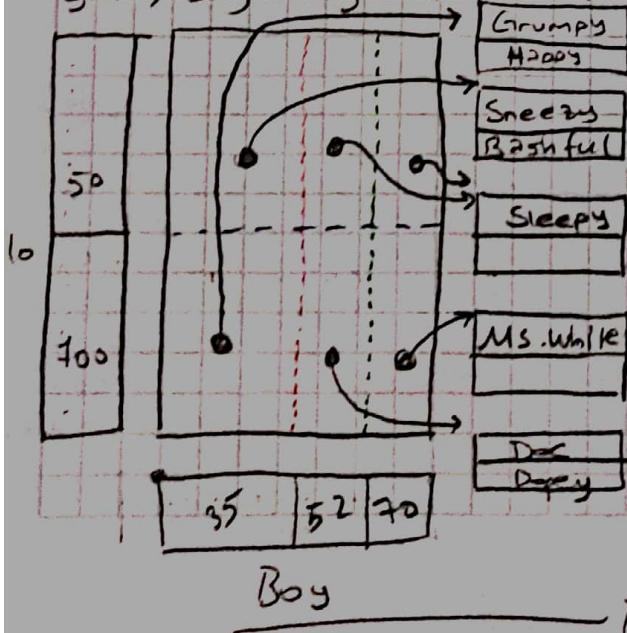
* 4 grid'in 4 adres olması gereğinden fazladan adresi Happy'nin olduğu bucket'i 2 bağladı.

→ Grumpy (32, 55) eklenirken; $32 < 35$ olduğundan solda, $55 > 50$ olduğundan sağda olacak. Buradaki pointer Happy'nin 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 sola, $50 \leq 50$ olduğundan yukarı eklenerek. Değe öncesi buraya Sneezy'i yazmıştık. Onu altına ekliyoruz.

→ Ms. White (62, 38) eklenirken, $62 > 35$ olduğundan sağ, $38 > 50$ olduğundan sağa yeleştiriliyor. 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 ve ilerilerdeki)

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

* Göktu key, aralık ve eşitlik soruları, bunun uygundur. Ancak aynı aralığı 2 farklı eleme yaparsak arka RAM'e sigaramayarak düşmeye gelir.

- 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 ; (ortalaması sek=16ms)

a-) Bir plakanın kapasitesi nedir?

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

* Plakalar çift yüzeyli olduğu için 2 ile çarpılmış.

* Plaka \rightarrow yüzey \rightarrow track \rightarrow sektör

b-) Bir track'in boyutu nedir?

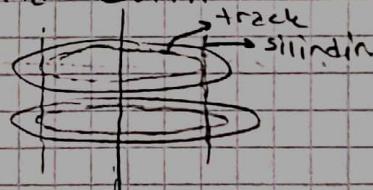
$$200 \times 100$$

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

c-) Kaç silindir vardır?

400

* Silindir sayısı bir yüzeydeki track sayısına eşittir.



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 çarpılmış.

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

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

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

* Ortalaması dörtlüsel gecikme disk okuma yazma başlığı ilgili sektörde yerlestikten sonra plaka üzerinde bulunacak verinin başlangıcını beklerken geçen süredir.
↓
Bir dönüş bu kadar zaman alıyor.

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

1 track, 100 sektör içeriğinde
 $\frac{100}{5} = 20$ 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'i 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 boyutu $R = 400 \text{ byte}$.

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.

Pile 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 kayıt alımı süresi nedir?

→ Sorted seq kısımları $\log_2 b$, overflow kısımları pile file gibi dövranır. Bunların ikisini hesaplayıp toplayacağız.

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

sayfa

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

$$= 1.76 \text{ saniye}$$

d-) Bu daya bir biner index yapısı bulunduruyorsa ve bu index yapısı RAM'e sığa bilerek büyüklerde bir kayıt bulmak için, RAM'de arama yaptıktan sonra, bir disk erişimi gerer.

$$16 + 8 \cdot 3 + 0.84 = 25.16 \text{ ms} \approx \boxed{0.025 \text{ saniye}}$$

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

$S=16$, $n=8.3$, $btt=0.84$, 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.
 → 6 sayfada $6 \times 6.67 = 39.99 \approx 40$ kayıt bulunmaktadır.

$$\text{Master file da } \frac{100.000}{6} = 16667 \text{ sayfa}$$

$$\text{Transaction file da } \frac{60.000}{6} = 10.000 \text{ sayfa.}$$

$$\text{Güncelleme dosyası, } 100.000 + 60.000 + \frac{30}{100} - 60.000 + \frac{20}{100} = 106.000 \text{ kayıt}$$

(Yazılan kayıt eklene
ve silme işleminden
sonra oluşan yeni dosya)

$$\frac{106.000}{6} = 17.667 \text{ sayfa}$$

* 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:

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

$$TRF = 10.000 * (16 + 8.3 + 0.84) = 251600 \text{ ms}$$

$$TOF = 17.667 * (16 + 8.3 + 0.84) = 444148.38 \text{ ms}$$

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

* 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 katı变得更大了。

4-) Bulk loading algoritmasını kullanarak, 10, 13, 15, 16, 18, 20, 21, 22, 25, 40, 45, 60, 17 derecesi ($d=2$) olan ağaca bu anah talarını ekle.

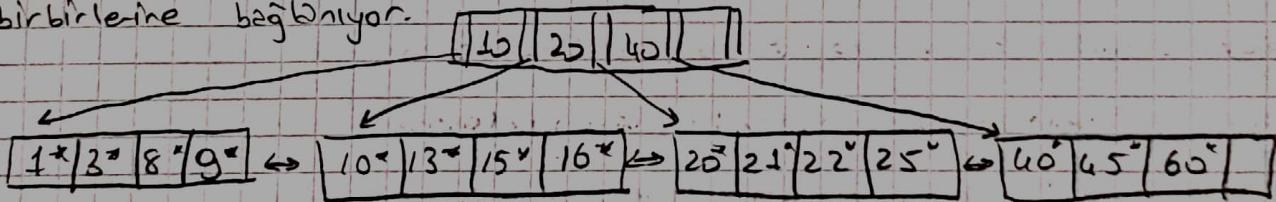
→ Öncelikle bu algoritma için sayıları sıralamalıız.

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 en fazla 4 kayıt (root hariç) bulunduğunu anlıyoruz. (En fazla bulunan kaydın %50'si d'yi vermelii). Yani bu sayıları 4'erli şekilde gruplayacağız.

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.

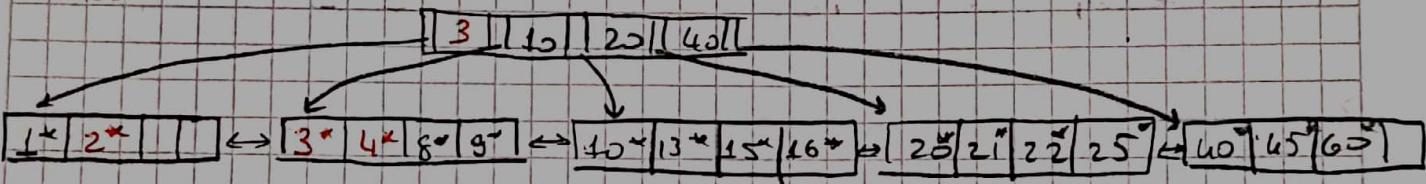


+ Böylece B+ ağacını tüm kurallara uygun birimde oluşturduk.

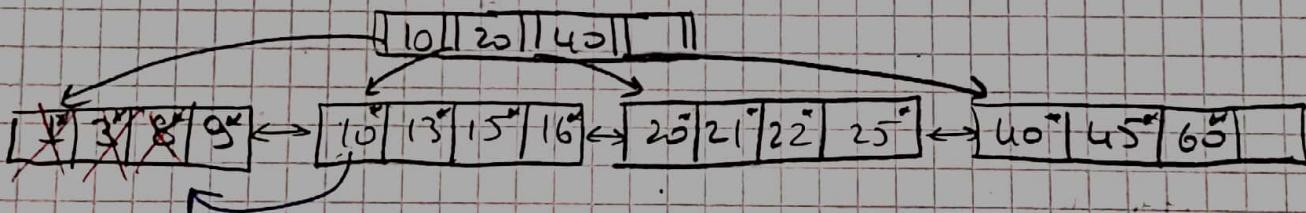
2 ve 4 eklemek isterset:

2 eklemek istediğimizde ilk node'ı yerlescedektir. 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 boşluyoruz. (Üste taşınan değer sırası bozmayacak şekilde yerleştirilmeli).

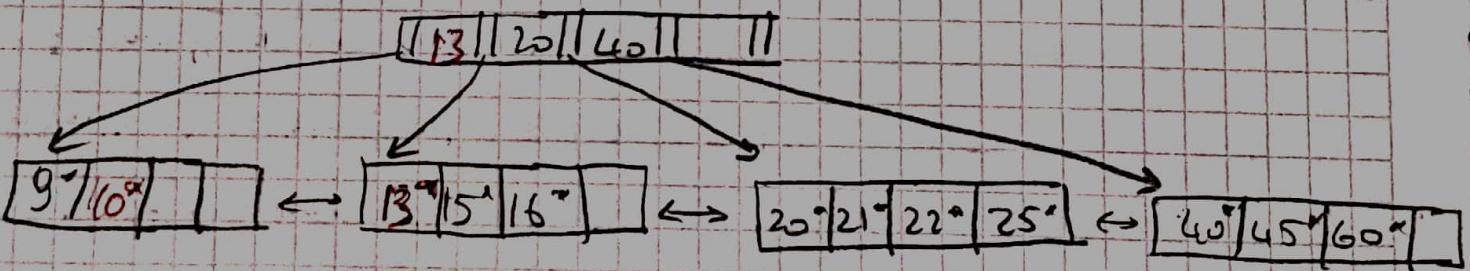
Daha sonra 4 eklemek istendiğinde boşluğun node'da yer olacağından direkt olarak buraya yerleştirilir.



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

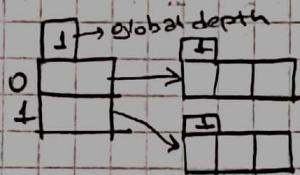


\star $d=2$ olduğundan bir node da en az iki kayıt olmalı. Bunu kurallı bozmamak adına komşu node dan (komşu node atasından sağlı sollu dağılmış node lardır. Örn 10'un sağı ve solu) eğer yete ri kadar kayıt varsa (en az $d+1$ kadar) en küçük elemen sağ tarafe aktarılır. Bu aktarım sonrası root kısmı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



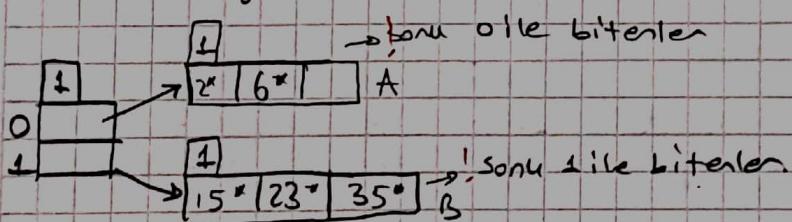
5-) 2, 15, 23, 6, 35, 13, 8, 26, 1, 41 anahtarına sahip kayıtları hash tablosuna extendible hashing algoritması ekle.
 $h(key) = \text{key} \bmod \text{Global depth} = 1$ ve her bir data page 3 kayıt tutsun.

→ Öncelikle hash tablosunu oluşturalım.

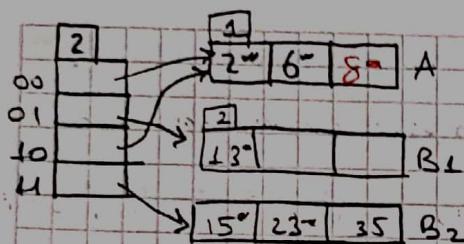


→ Yukarıda verilen sayıların keylerinin binary karşılığı bulunur ve bu binary sayının sondan depth kadar bitine bakılarak genelde yerleştirilir.

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

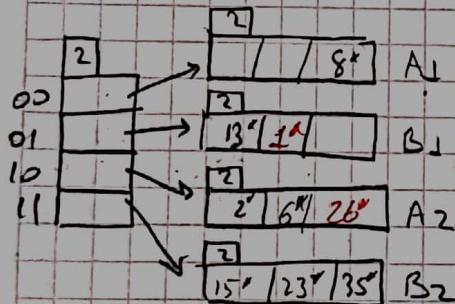


→ $13 = (1101)_2$ eklemek istendiğinde son basamakı '1' olduğunu için aşağıdaki bucket'a yerleştirilmesi genelde. Fakat bu bucket dolu. Bunun için bu bucket'i 2 ye böülüyoruz. Bu bölünme sonucu 3 tane bucket oldu. Bunların adreslerini soldaki yapıda tutuyoruz. Ama burada 2 adres tutmak için yeter var. Bunun için global depth'i + artırıyoruz. (Artık eklenen yapanken son 2 bas. baktırıcaz.)



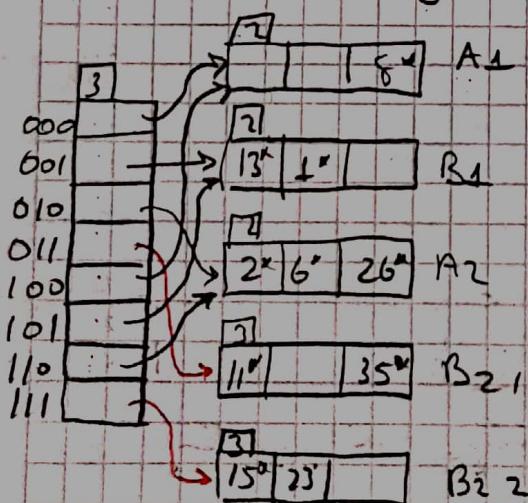
$8 = (1000)_2$ hiz yer varicos
bunu da yerleştirdik.

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



$1 = (01)_2$ olduğundan ve boş yer
olduğundan ekliyoruz.

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

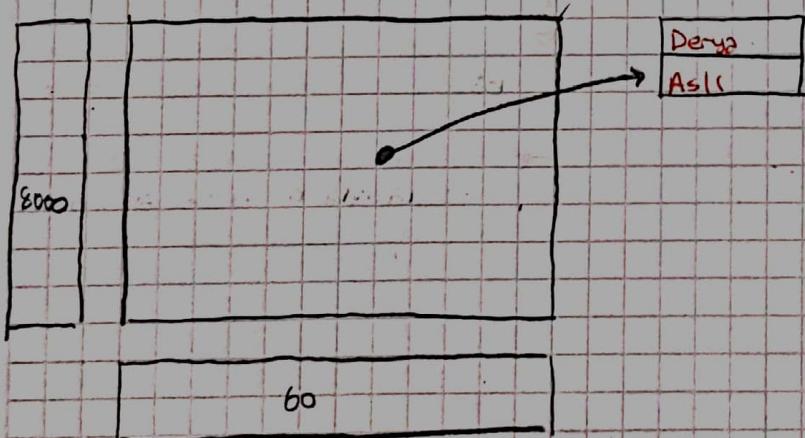


* Bucket'la eklenme sonucunda
local depth > global depth
oluyorsa global depth + artırlın.

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

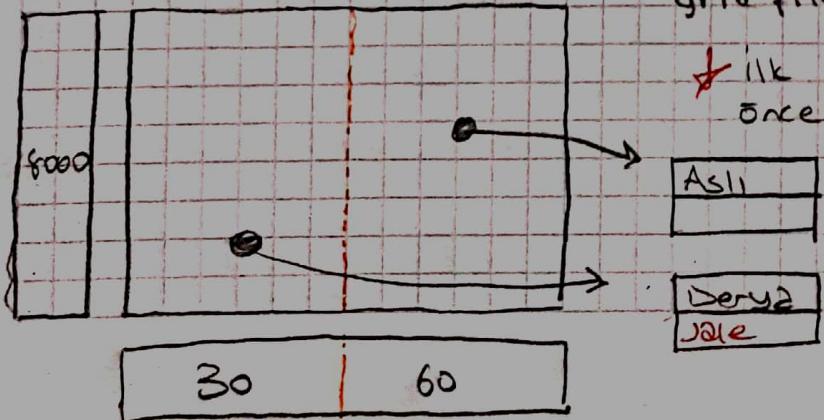
isim	yaş	maaş
Derya	28	3500
Aslı	35	5000
Jale	18	2700
Deniz	27	3500
Mehmet	45	5500
Engin	55	5500
Sami	19	2600
Rüya	50	5500

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



→ Aslında soldaki grid file
elemanı sağda bucketların
adresini tutar
Burada şart $0 < \text{yaş} < 60$
 $0 < \text{maaş} < 8000$

→ Jale (18, 2700) eklemek istendiğinde bucket'ta yer olmadığından grid file'i ikiye bömeniz gerekiyor.



→ ilk boyut yaş olduğu için
önce yaş bölünle.

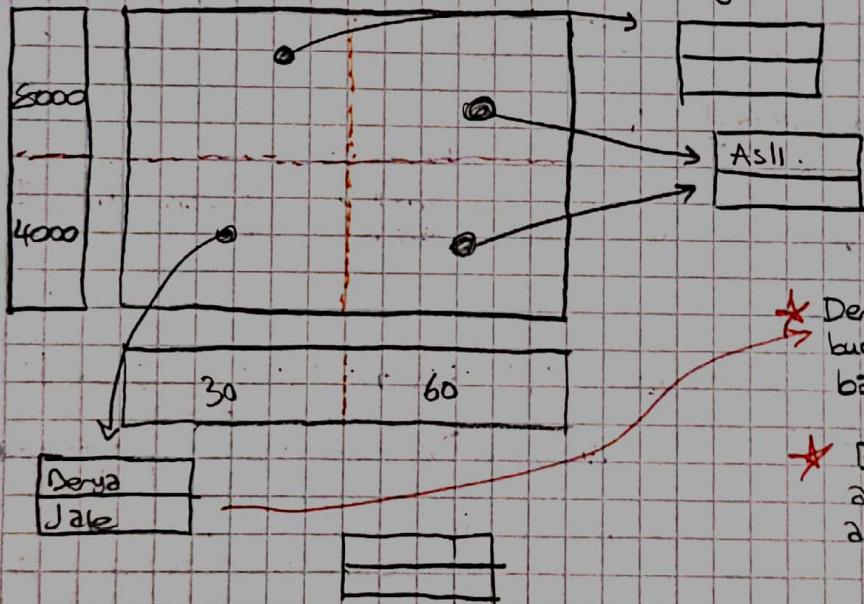
→ Bu bölünme sonrası
yaş 30'da kırıktır.
Olar solda büyük
olar sağda kaldı.

★ ★ ★

Burada unutulmaması gereken kurallar eklerken hep aynı logutu bize alarak etliyoruz (Bu örnekte yaz). Yalnızca tasmaz durumlarında sırayla bölüyoruz.

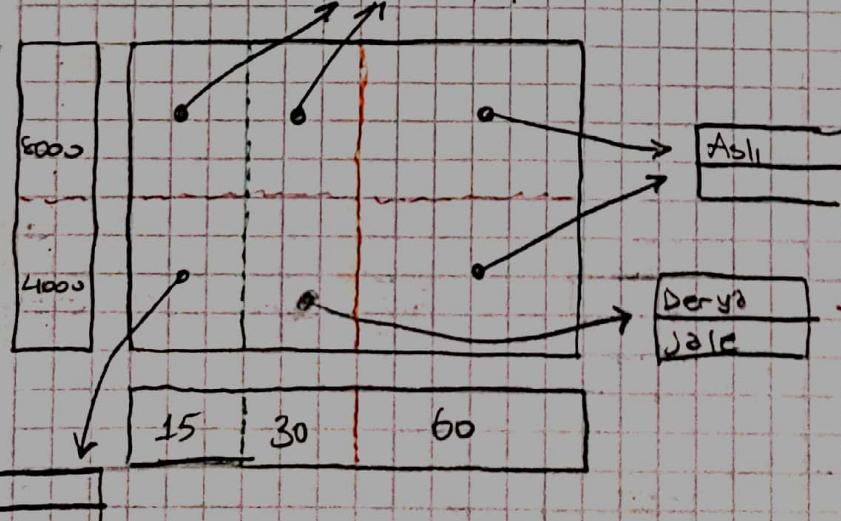
★ ★ ★

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



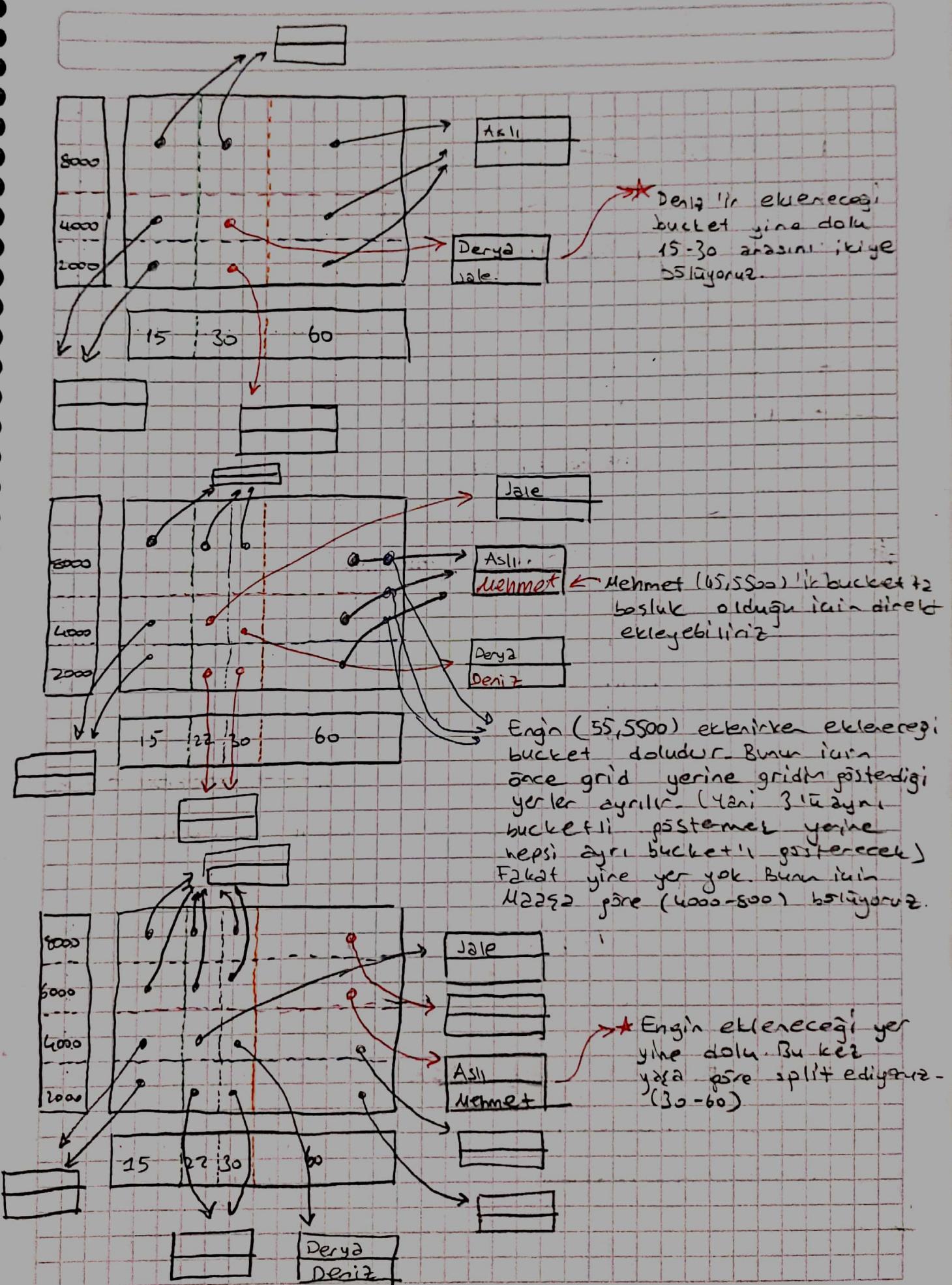
★ Deniz (27,3500)'in ekleneceği bucket yine dolu yine bölüyoruz. Bu kez yaşına göre.

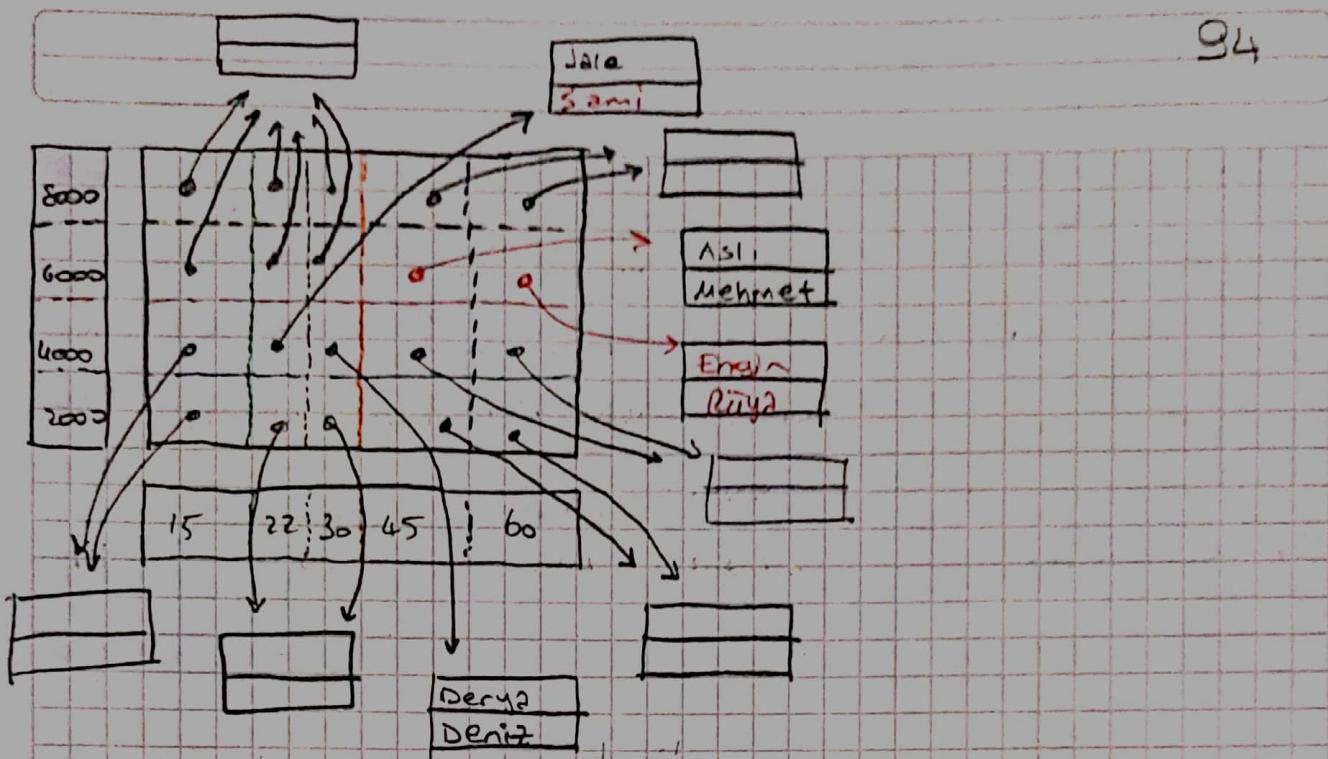
★ Bütünecek bucket 0-30 aralığında olduğundan bu aralığı bölüyoruz.



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

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





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

— HAFTA VII. SON —

— VİZE —

— HAFTA VIII. SON —

CHAPTER 13:

- EXTERNAL SORTING -

Neden sıralama (sort) yapmaya ihtiyacınız?

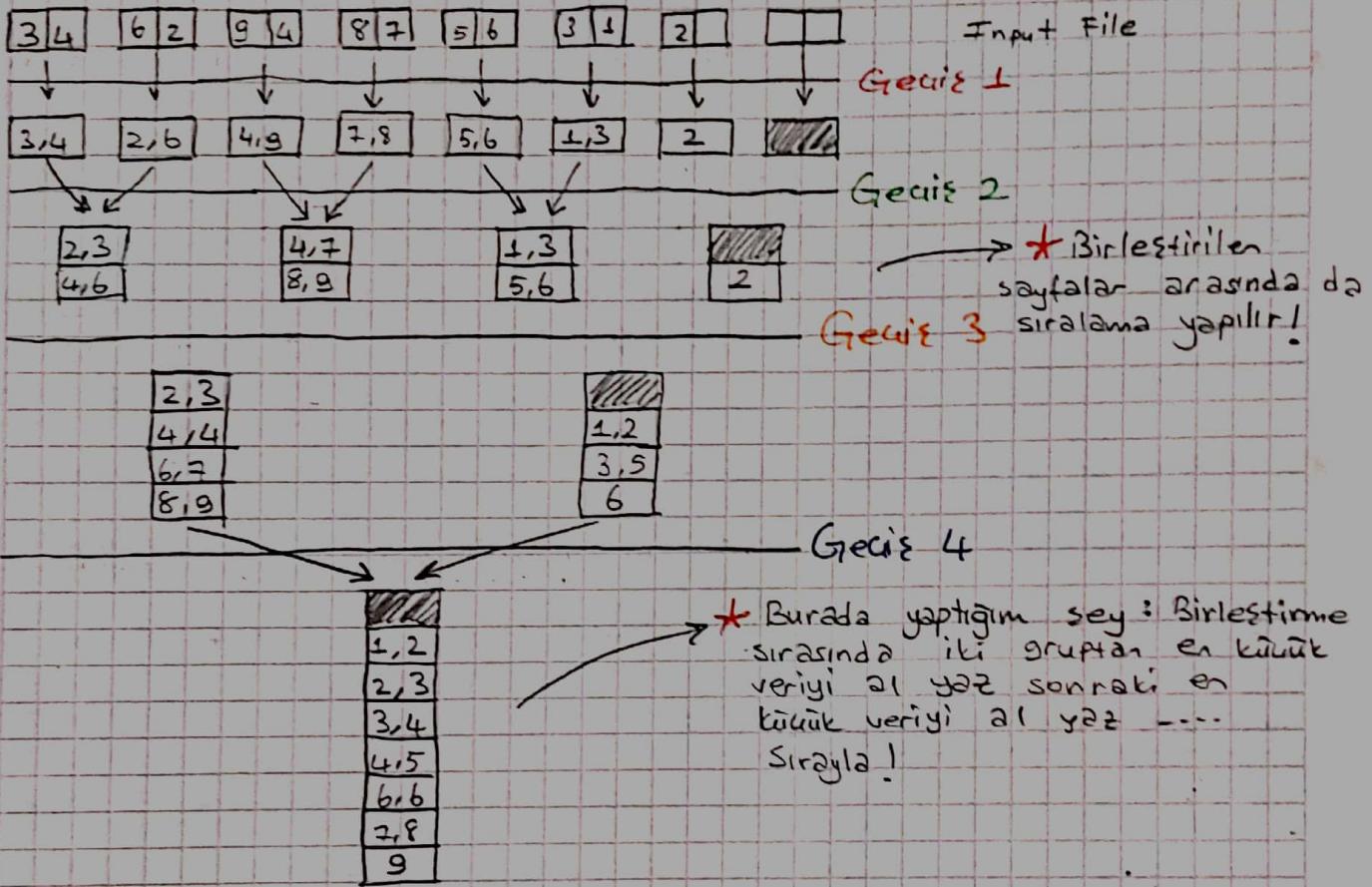
- ★ Sıralama aslında bilgisayar bilimlerinin klasik problemidir.
- Veriler sıralı bir şekilde istenmiş olabilir. (örnegin öğrencilerin ortalamalarını artan sırada isteyebiliriz)
- B+ ağacı oluşturmak için uygulanan Bulk-Loading algoritması sıralama kullanır.
- ★ 10 GB'lık bir veri 1 GB'lık bir RAM kullanılarak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

Two-Way External Merge Sort :

Bu algoritma belleğin çok küçük olduğunu farz eder. (sadexe 3 tanesi buffer page olduğu varsayıllır). Bu 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 yazar.
- ikinci ve sonraki geçişlerde, sayfalar kendi içlerinde sıralı olduğundan iki sayfayı diskten belleğe okur ve bunları merge eder (birleştirir). Bu oluşan yeni sayfayı diske yazar.

Örnek :



* Bu örnekte, dosyada 7 sayfa vardır. (En sağdaki sayfa örneği daha iyi anlatmak için var). Bu sayfaların içinde kaytların keyleri var. Birinci geçiste tüm sayfalar kendi içlerinde sıralanır. ikinci ve daha sonraki geçislerde bu sayfalar sıralı olacak şekilde birleştirilmeye devam eder.

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

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

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

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

↳ 2N olmasının nedeni burası .

* Sure maliyetini bulmak için :

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

(random access olup olmadığı için)

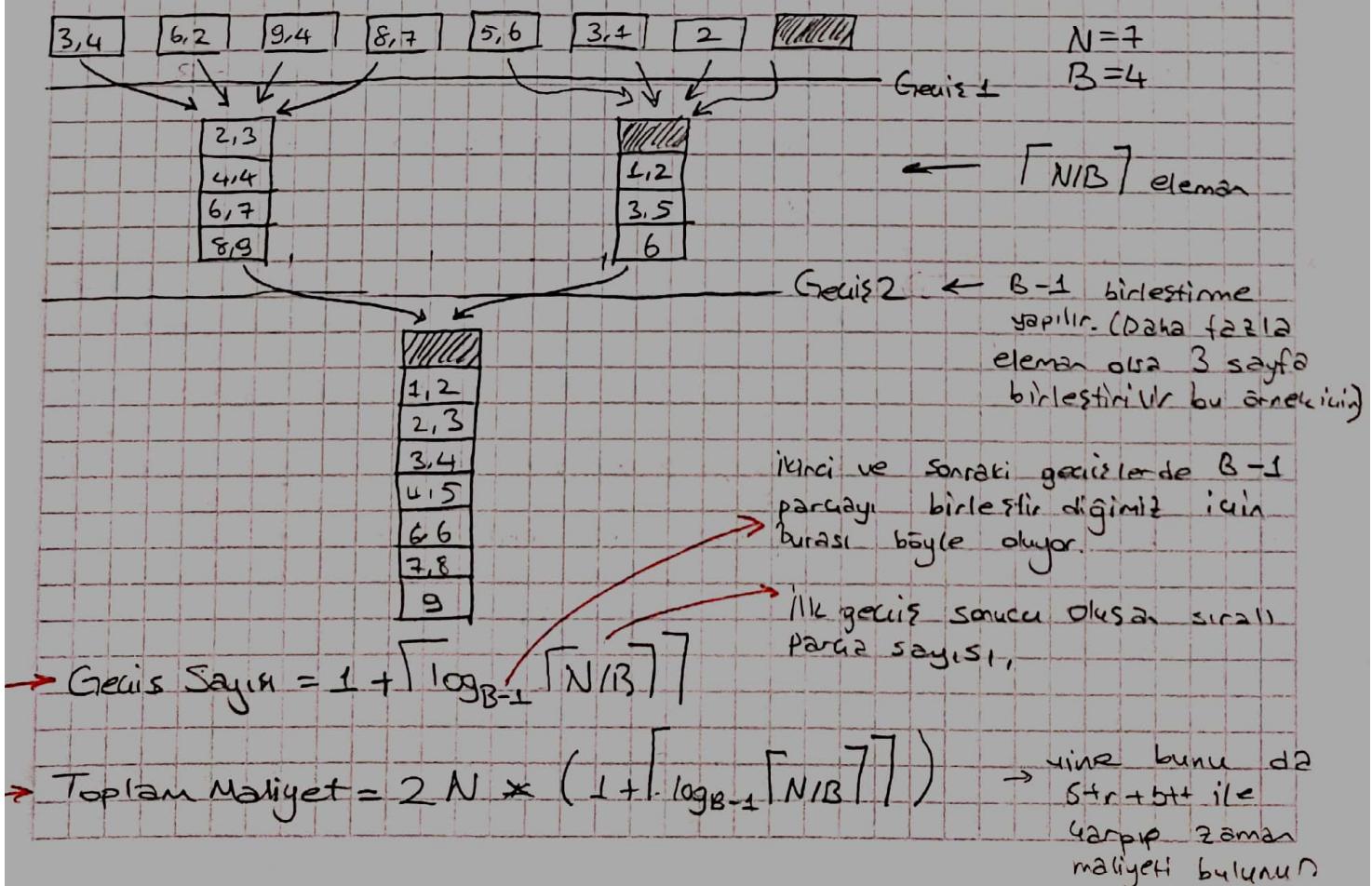
General External Merge Sort:

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

Fakat pratikte bellekte çok daha fazla alanı kullanabiliyoruz. 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 birlestiriyoruz. Bellekte B sayfa vardı \lceil tanesini merge sonucunu yazmak için alırsak geriye $B-1$ kalmıştır.
- * Two-ways algoritması, bu algoritmanın $B=3$ olduğunu hatırlatır.

Örnek:



Örnek: External Merge Sort kullanılacak.

$$N=108, B=5$$

(sayfa)
(sayı)

↳ buffer pool
frame sayıları

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

(gecis)

- Birinci geçis: $\lceil 108/5 \rceil = 22$ sıralı dosya parçası. (son parçanın 3 sayfanın birlenesmesi ile olur)
- İkinci geçis: $\lceil 22/4 \rceil = 6$ sıralı dosya parçası. (birinci geçisten sonra) (birinci geçiste $B-1$ 'e böl)
- Üçüncü geçis: $\lceil 6/4 \rceil = 2$ sıralı dosya parçası
- Dördüncü geçis: Tüm dosya sıralandı!

Internal Sort Algorithm:

External Sort Algoritması üzerinde iyileştirme yapılmış halidir.

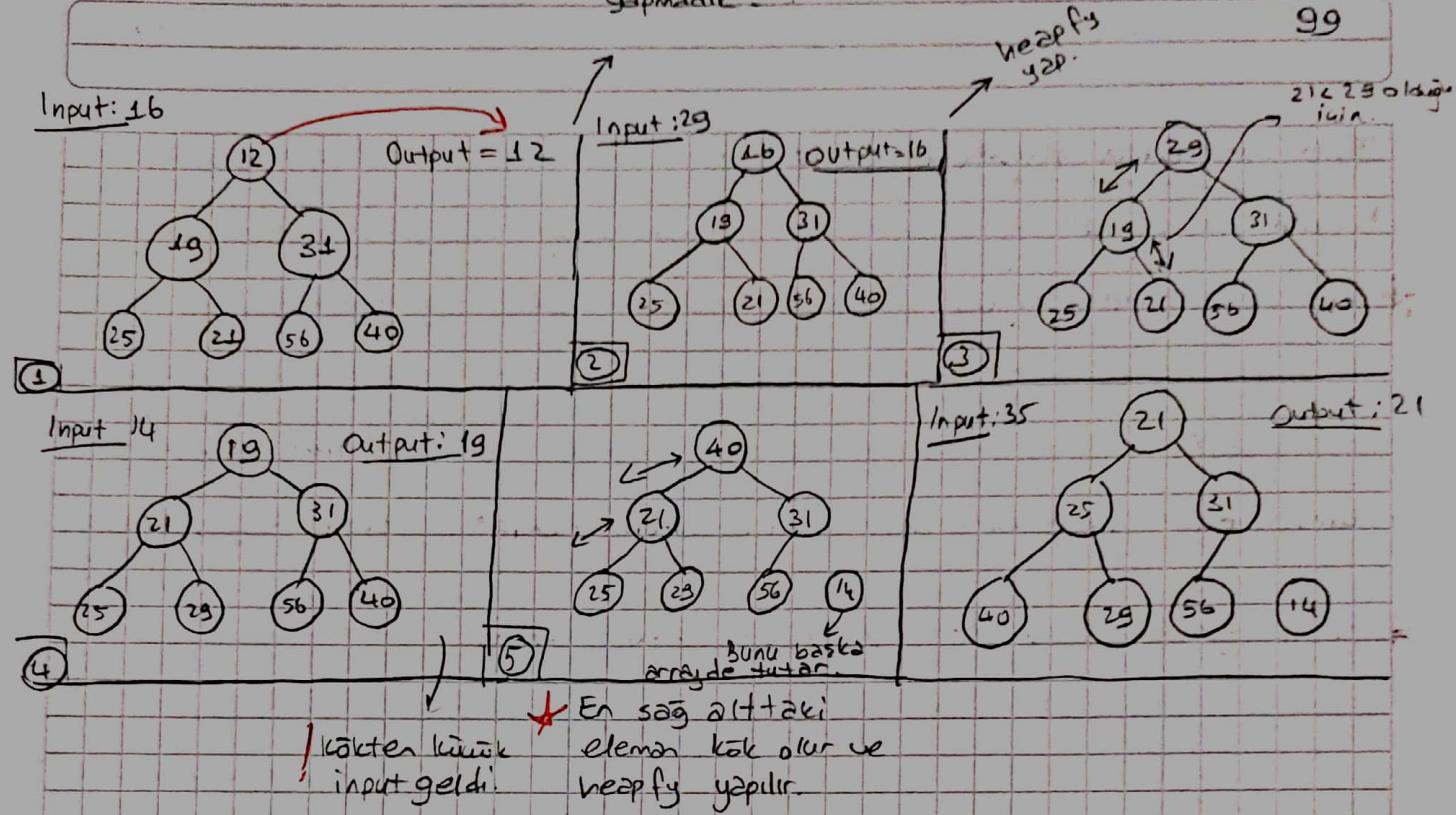
* Buradaki fark; birinci geçiste B tane bloğu buffer'a aktardıktan sonra sıralamak için quick sort yerine, heap sort algoritmasını kullanılır. Bu sayede B sayfalık veri sıraladıktan sonra $\approx B$ uzunlığında sıralı dosya parçası elde ederiz.

Şu şekilde oluşturular:

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 değer köke taşınır. Bu işlem heapify yap.

16 değerini sıralamayı
bozmadığından heapify
yapmadık

99



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

* Aslında tek yaptığımız 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 almak yerine, ardışık olarak ' b ' kadar sayfayı alırız. Bu sayede merge edilecek 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 yapmanız 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 yapmak için kullanılır. Dolayısıyla bellekte B sayfa varsa bunun yarısı diskten veri okumak-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.)

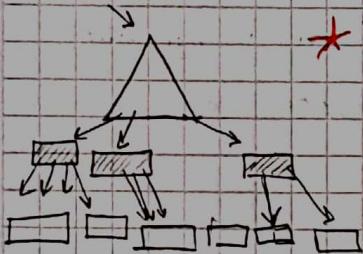
* Toplam beklemeye süresi azalır. Çünkü işlenici olayları ve disk okuma yazması paralel gerçekleşir. (Dünyanın da)

* Toplam sürede azalma olur!!!

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

Sıralamak istediğimiz bir dosya ve bu dosyanın B+ ağacı indeksinin daha önceki olusturulduğunu varsayıyalım.

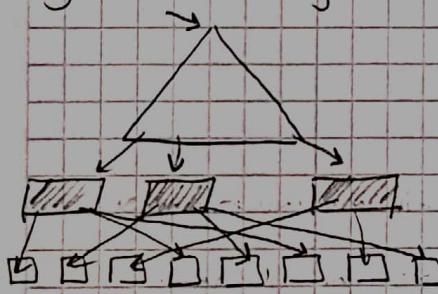
- Eğer bu B+ ağacı clustered (kümelemmiş) ise :



* 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 (kümelemmemiş) ise :



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

* 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 problemleri üzmemek için veriyi nasıl modellereceğimizi bize gösterir. Öğrenci, ders \rightarrow örn: kayıt yaptırma.
- İlişkisel veri modeli varlıklar ve ilişkilerden oluşur. (Öğrenci dersے kayıt olur) (entities) (relationship)

Files vs DBMS:

- Veri aslında dosyalar halinde saklanıyor. Fakat bu dosyaların diske 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 yüzlerce öğrenci sisteme eş zamanlı erişim sağlıyor ve bu kayıt esnasında değişimlere göre sistemin tutarlılığı anlık bir şekilde sağlanıyor.)
- Sistem вокмелерinde, DBMS, verinin tutarlılığını sağlar.
- DBMS, veritabanındaki verinin güvenliğini sağlar. (örn: Farklı kullanıcılar farklı yetkiler voerek, o kullanıcının sadece kendine verilen yetki doğrultusunda işlem yapması sağlanıyor)

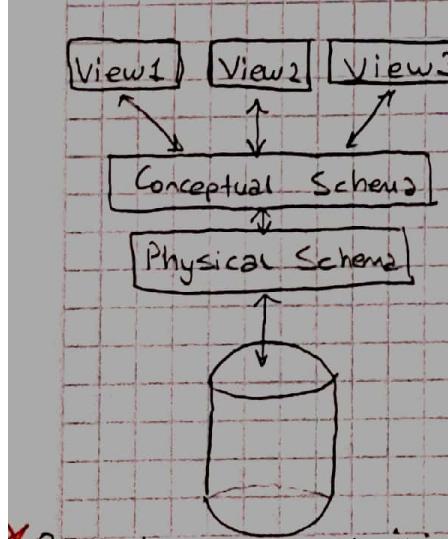
- Neden DBMS kullanıyoruz?

- Verinin bağımsız olmasını ve veriye etkin bir şekilde erişmek için kullanılır. (Veriye eiden programların doğrudan erişilememesi)
- Uygulamayı geliştirmekten gelen süreyi kısaltır.
- Veri tutarlığını ve güvenliğini sağlar.
- Veri yönetiminin tek bir eden yapmasını sağlar. (DB Administrator)
- Eş zamanlı erişim ve sisteme nüfusinde verinin kurtarılmasını sağlar.

Veri Modelleri : (Data Models)

- Her DBMS bir Veri modeline dayanır. Veri modeli, veriyi nasıl tanımlayacağımız ifade eden kavramlar bütündür. Bu modeli kullanarak verinin seması çıkarılır ve veritabanına aktarılır.
- Bugün en çok kullanılan Veri modeli = ilişkisel veri modelidir.
İş ilişkisel veri modeli ilişkiden oluşur. (Yani tablodan oluşur)
(relation) ~~bu anlamladır ve gelir~~ ↗ 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 nedir?)

Sayıtlama Seviyeleri : (Levels of Abstract)



- Görüldüğü gibi DBMS'de katmanlı bir sayıtlama yapısı vardır
↗ Her tablo bir dosyadır.
- Veritabanı, disk üzerinde saklanan tablolardır. ↗ Bir dosyadır.
- Fiziksel sema, her bir dosyanın türünü ve bu dosya üzerinde kullanılan index yapılarını tanımlamamızı sağlar.
(file file yapıında B+ ağacı index örn.)
- Kavramsal sema, her bir tablonun sütunlarını yanı niteliklerini ve bunların veri tiplerini tanımlamamızı sağlar.
- Aynı veritabanı üzerinde çok sayıda view oluşturulabilir. Her bir view, kullanıcının veriyi nasıl göreceğini tanımlar. Dönen her öğrenciin yalnızca kendi notlarını görmesini view sağlar.
- Bu yapıları oluşturmak için view oluşturulabilir. Her bir view, kullanıcının veriyi nasıl göreceğini tanımlar. Dönen her öğrenciin yalnızca kendi notlarını görmesini view sağlayabilir.
- Bu yapıları oluşturmak için view oluşturulabilir. Her bir view, kullanıcının veriyi nasıl göreceğini tanımlar. Dönen her öğrenciin yalnızca kendi notlarını görmesini view sağlayabilir.

Veri tanımlama dili ile tanımlanır. (SQL'yi de karsın)
Sorgular veri işlemi dili ile dir. (SQL'yi de karsın)

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 tablo 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 dosyalar üzerinde tanımlı olan index yapılarının tanımlandığı semadır.
- Tüm tablolar pile file şeklinde 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 oderni konu kişinin 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 ağıacı 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 diskeki veri okuması gerçekleşse onu bekletip ikinci programın öncelik veriyor. Daha sonra birinde dönüp okuduğu verileri hesaplanması için CPU'ya gönderiyor... Bu yapılı interleaving actions denir.
- Tüm bunları yaparak kullanıcıları, sistemi kullanın sadice kendisini 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 sonlanır ya da (bir hata meydana gelirse) hiç başlamaması gibi davranışmalıdır.
- Transactions, DBMS tarafından birdizi okuma/yazma işlemlerinden ibarettir.

Scheduling Concurrent Transactions:

Es 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 neden için Strict 2PL gibi bir yöntemle mesaj olan nesneler kilitlenir.

Strict 2PL'in dezavantajları da vardır. Örneğin; I. transaction X'ı okusun ve II. transaction da Y'ı okusun ve bunları kilitlensün. I. program Y'yi II. program X'ı okumak isterse burada bir sonsuz döngü olusur ve İI. transaction da iptal edilir.

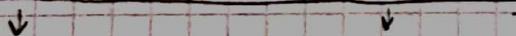
Ensuring Atomicity:

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 buyu da log'a yazılır.

DBMS Kullanıcı Grupları:

End-user: (Über sistemini kullanan öğrenci ve öğretmenler.) DB'nin nasıl kullanıldığı bilgilendirme
DBMS programı (saficiisi) → {gerek yok!}

DB app programmer ve DB admin.



Überi kodlayan
kisiler

Überin DB yapısını
oluşturan kisiler.

DBMS Yapısı: (Structure of DBMS)



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

Query
Relational optimization
Execution
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 çözeceğiniizi öğreneceğiz.
- Bir gerçek hayat problemi için ilişkisel veri modelini kullanarak nasıl kavramsal tasarım yapacağınızı öğreneceğiz. Bu kavramsal tasarım yapmak için şu soruların cevabı aranmalıdır :

- Problemdeki varlıklar ve ilişkiler nelerdir? (Cübis sisteminde
 (entities) (relationship)
varlıklar: öğrenci, öğretmen, dersler ... İşikiller: 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 kayıtları özel olmalıdır

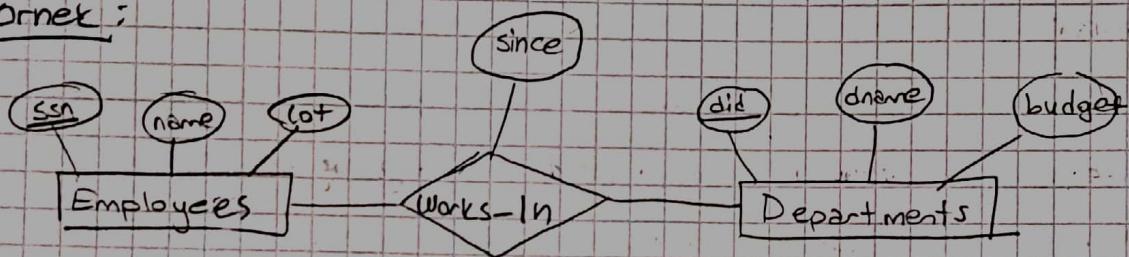


* Varsıklar dikdörtgen içinde yer alır. Nitelikler ellips içinde yer alır. Anahtar niteliğin altını çiziyoruz (Primary keyler yani).

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

* İlişkiler, iki ya da daha fazla varlık arasında tanımlanmış ilişkiseliklerdir. Aynı türden ilişkiler ilişkî kümelerini oluşturur.

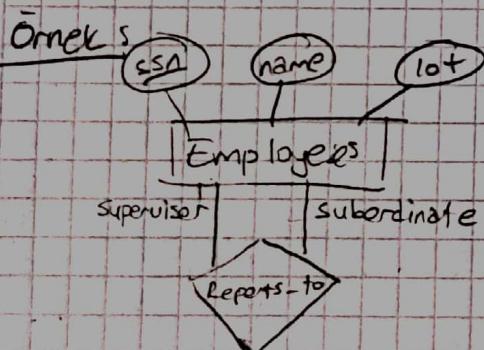
Örnek:



→ Bu örnekte: "İşçiler, departmanlarda çalışır" denmektedir.
 ↴ ↴ ↴
 varlık varlık ilişkî

* İlişkî kümeleri eskenar dörtgen ile ifade edilir.

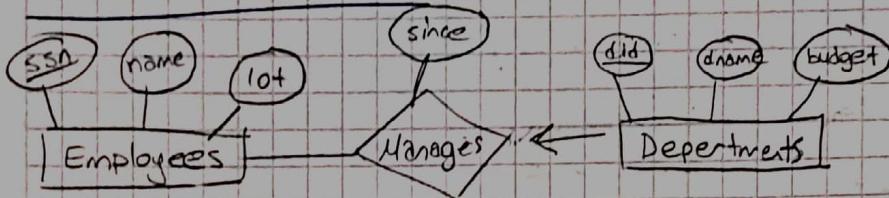
* İlişkî kümelerinin de niteliği vardır. Bu örnekte "ne zamanдан beri çalışır" sorusunun cevabı tutturur 'since' niteliğinde. İlişkiler aynı türden varlıklar arasında da kurulabilir.



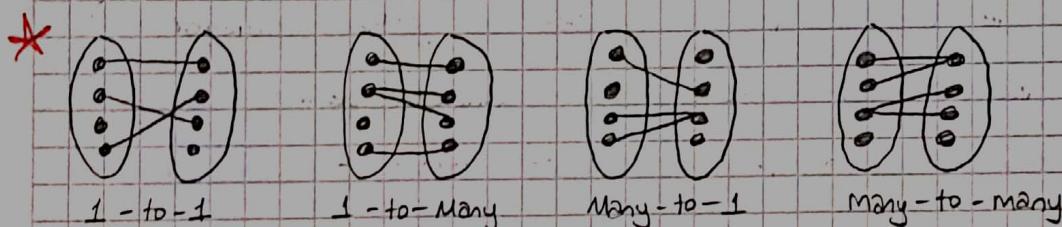
* Bazı çalışanlar, bazı çalışanlara "rapor yolları" ilişkisi ile bağlıdır.

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

Anahtar Sınırlaması: (Key Constraints)

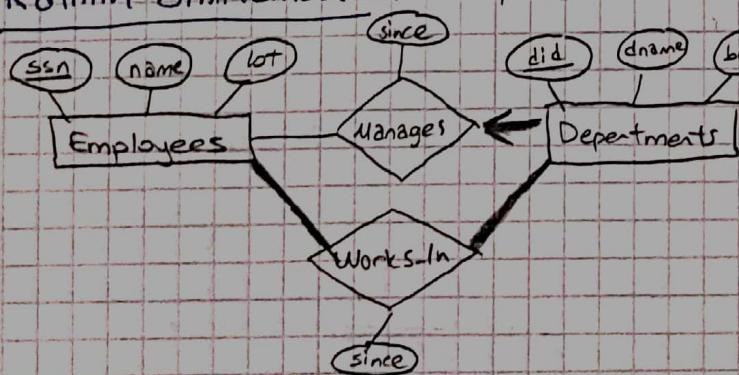


* Buradaki gibi bir okla su anlatılır: "Departmanlar, manages ilişkisine en fazla 1 kez katılabilir" bu şekilde bir sınırlama, anahtar sınırlamasıdır.



→ 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 anlamladır.

"Bazı çalışanlar manages ilişkisine katılır. Ailesi katılmak zorunda değil!"

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

→ Bir personel en az 2 bir departmanda çalışmak zorunda.

→ Bir departmanın en az 1 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 katılım sınırlaması ve anahtar sınırlaması ilişkisi tanımlanmak zorundadır. Yani her çocuğun bir sağlık sigortası ile sigortalanmalıdır.

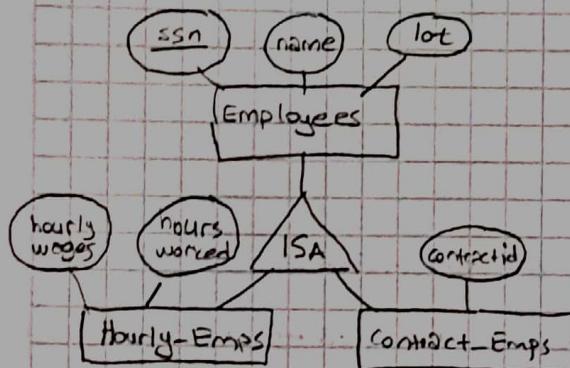
→ Yukarıdaki ER diğramı sunu anlatır:

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

* Policy'nin kayıtlarının nedeni: zayıf varlığı ayırt eden ilişkileri kümesi olmasından kaynaklidir.

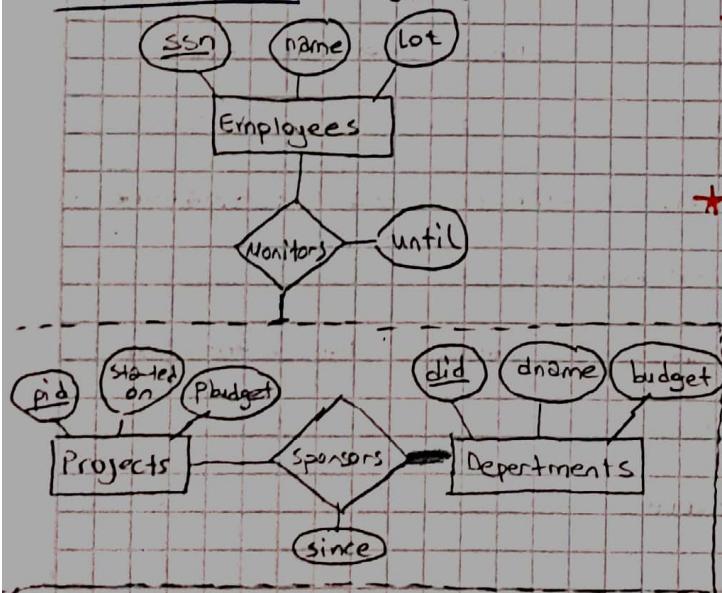
HAFTA XI SON

ISA (is a) Hierarchisi : (ISA ('is a') Hierarchies)



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

Birlestirme : (Aggregation)



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

* Aslında aşağıdaki ilişkiyi, kesikli üçjili dikkâtgen içinde alarak, bir varılmış gibi gösteriyoruz.

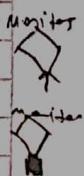
* Bu örnekte su ifade ediliyor:

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 edemeyecegi (denetler) (destekler) için aggregation kullanıldı.

* Monitors ilişkisine de ilave sınırlamalar verilebilir. Örneğin;

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



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

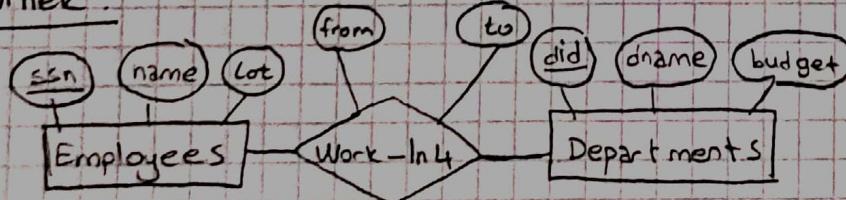
Varlık vs Nitelik : (Entity vs Attribute)

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

- Bu da bir örnek vermek gerekirse, 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 calisan varlıklar arasında ilişki kurulur.
- Eğer adresle ile göre, ilçeye göre, mahalleye göre sıralama yapmak istiyorsa, adresi varlık olarak tanımlayıp il, ilçe, mahalle bilgilerini de adres varlığının niteliklerini olarak tutmalıyız ve bir ilişki ile calisan varlığına bağlamalıyız.
- Eğer calisanın bir adresi varsa ve bu adresle ilgili herhangi bir soru yapılmayacağsa (sadece bilgi olarak tutacaksak), adresi personelin niteligi olarak tanımlanır.

* Bir personelin birden fazla türde (ev adresi, iş adresi vs) adresi varsa, adres değeri kesinlikle nitelik olamaz. Çünkü nitelikler atomik (tek, kümeye olmayan) değerlerdir.

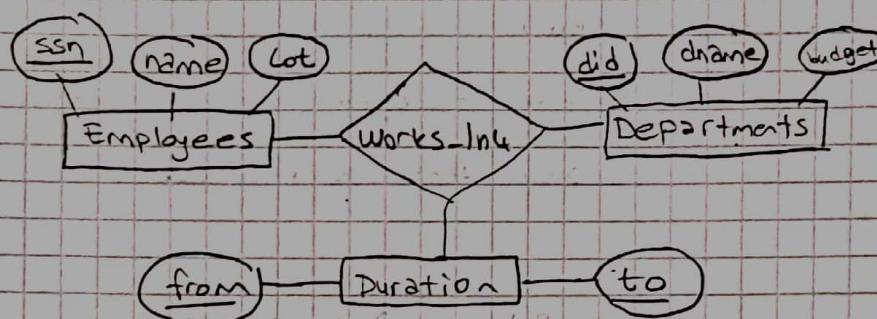
Örnek .



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

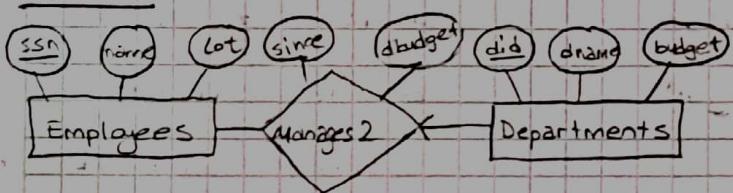
Ancak bir personel önce X departmantında 3 yıl, sonra Y departmantında 4 yıl ve son olarak Z departmantında 7 yıl çalışmışsa bu diagram yalnızca son çalıştığı departmantta ne kadar süre çalıştığını bilgisini verir.

* Eğer hangi departmanda kaç yıl çalıştı, hepsini öğrenmek istiyorsak ER diagramını şu şekilde düzenleriz.



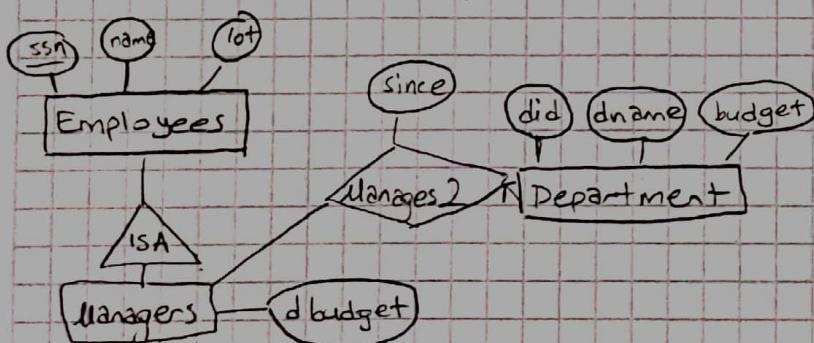
* Duration varlığı, ile yukarıda istediğimiz bilgilerin hepsini tutabiliyoruz.
(from ve to = primary keys)

"Örnek:



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

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

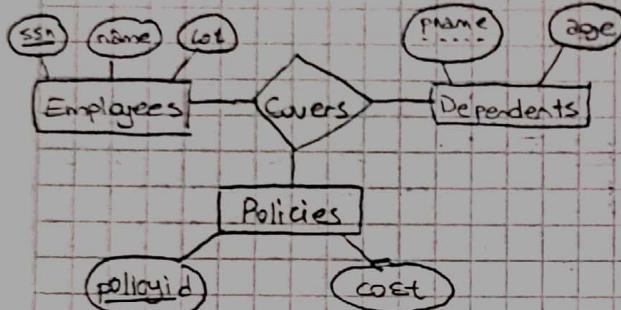


* Personellerin bazıları yönetici dir. Yöneticiler toplam bütçe (dbudget) verilmektedir. Yöneticiler, departmanları X tərəfindən itibaren yönetir.

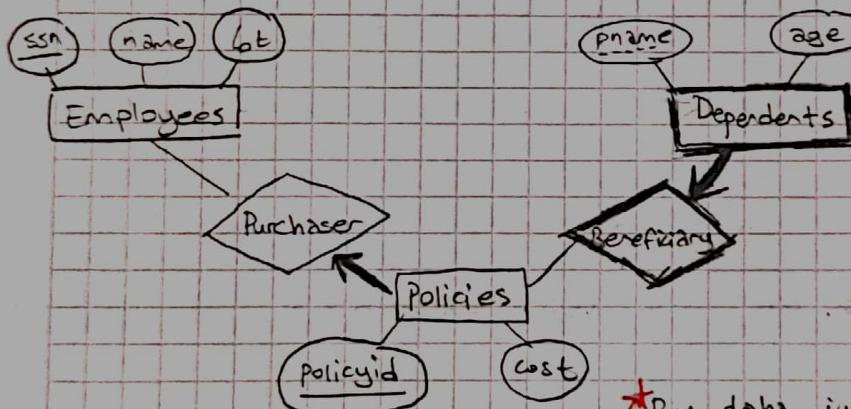
* Böylece her bir yöneticinin sahip olduğu toplam bütçe tutarını bir kez saklarız. Böylece veri tekrarından kurtuluruz.

* Burada dbudget kişiye özel veriliyor. Kişi 2'yi yönetmek için 5'ti alırsa bilyi yönetmek içinde 5'tü alır.

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



→ Çalışanlar, çocuklar ve politikeler arasında kapsar (covers) ilişkisi kurulmuştur. Bu, çok genel bir ER diagramıdır. Belirsiz sayıda birin dâirinde, örneğin burada bir çalışan diğer çocuklarından ayrılanız, çünkü zayıf varlığı ve keyfi yok.



* Bu daha iyi bir dizayndır.

→ Personeller polikileri satın alır. (Her bir polis, bir personel tarafından satın alınmak zorunda). Çocuklar her biri kesintile bir polikeden faydalamarak zorunda. Bağımlılık zayıf varlığı policyid kullanarak ayırt edebiliriz.

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

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

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

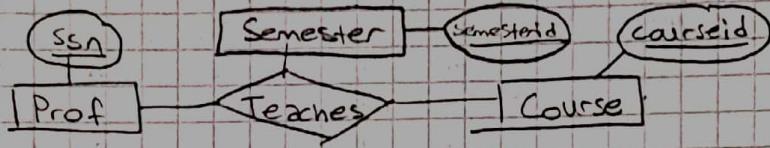
Problem :

Bir Üniversite DB'si profesörler(prim kdg = ssn), ve kurslardan(prim kdg = courseid) oluşsun. Profesörler kursu öğretir.

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

Örnek 1: Profesörler aynı dersi farklı dönemlerde verebilir ve (semester)

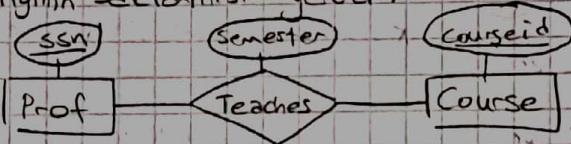
her dönemde ait bilginin saklanması gereklidir.



* Semester varlık olarak karmalı çünkü her dönemde bilgisinin tutulması istenir. Teacher varlığının bir niteliği olsaydı Semester, sadexe son dönemde bilgileri tutulurdu.

Örnek 2: Profesörler aynı dersi farklı dönemlerde verebilir ve son dönemde

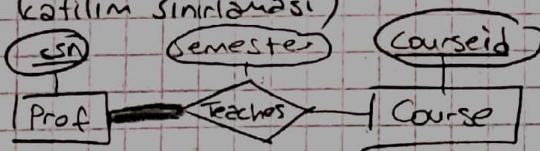
ait bilginin saklanması gereklidir.



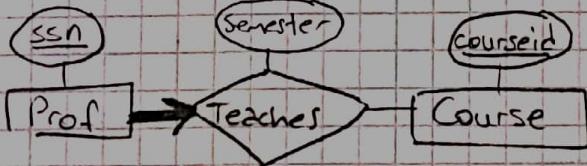
Burda sonraki örneklerde bu örnek baz alınmıştır.

Örnek 3: Her profesör en azından bir tane kurs vermek zorunda (sayı üzerine sınırlama yok).

(Toplam katılım sınırlaması)



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

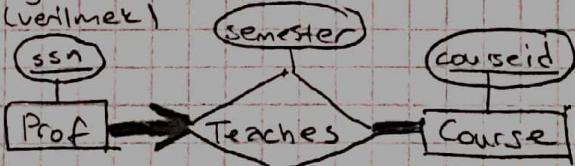


* Toplam katılım ve en az bir sınırlaması birlikte kullanılmıştır.

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

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

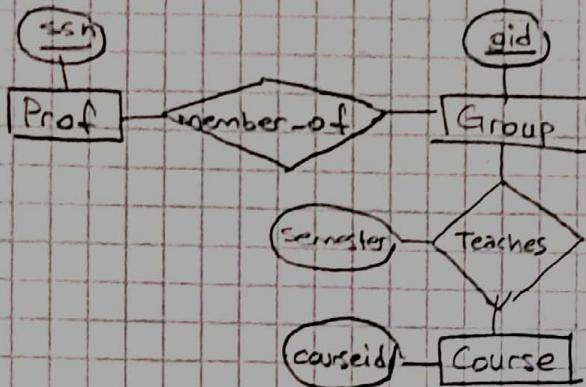
ders öğretilmek zorunda. (Her ders Teaches ilişkisine katılmak zorunda) (verilmeyen)



* Kac kere katılacak ile ilgili bir sınırma yok (≥ 1)

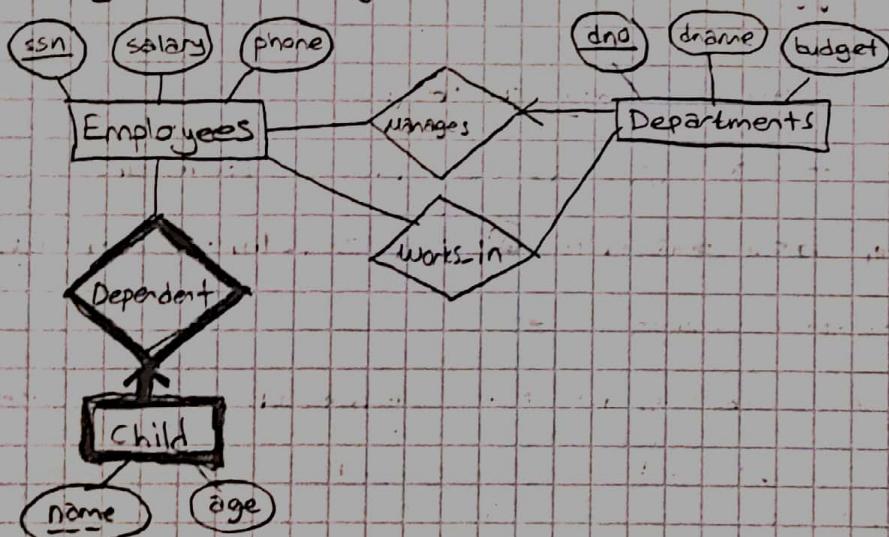
Örnek 6: Belirli dersler belirli bir grup profesör tarafından verilir.

Yani profesörler grulara katılacek gruplar ders verecek.



(Herhangi bir sınırlama yok, toplam
katılımın key sınırlaması yok)

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



Bunu ER diagrame
gösteremiyorum.
SQL sorgusunda
yazabilirim.

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 şemasının 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`
! (S yazmasaydı buraya de Students yazındı)

→ Bunun anlamı : Students tablosundan yaşı 18'e eşit olan tüm satırları çek. (Yukarıdaki tabloya uygulansak bu soruyu $1. ve 2. \text{ önceliğ} gelir$)
 $(yazılım = 18)$

* 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 getir. (SELECT S.name S.login yazsaydı, 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	cid	grade
53831	Carnatic 101	A
53831	Reggae 203	B
53650	Topology 112	A
53666	History 105	B

↳ S login
barası C
amazonus
değmez.

↳ Bu iki tablo üzerine su soruyu yazarsak :

- `SELECT S.name, E.cid 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 eşitleyerek bu iki tabloyu birleştirir. Daha sonra birleştirilmemiş olan bu tabloda notu "A" olan satırları seçer. Seçilen satırlardan name ve cid sütunlarını alır.

OUTPUT :

* Yukarıda bahsi geçen birleştirme işlemi su

S.name	E.cid
Smith	Topology 112

şekilde yapılır:

→ Students tablolarındaki ilk öğrenciyi Aldı. 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 (eslesfikleri için) birleştirilir.

* Bu birleştirme su anlama gelir = Numarası (sid) 53666 olan Jones isimli öğrenci History105 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 (nitelikleri) 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

↳ Mevcut 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 eklendiğinde, tabloda daha önceden ekli olan verilerin bu yeni niteliği 'null' olarak atanır.

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

- Daha önceki olusturulmus bir tabloya Yeni eklemek icin :
- INSERT INTO Students (sid, name, login, age, gpa) VALUES (53688, 'Smith', 'Smith@ee', 18, 3.2)
- ★ Eğer eklenenek kaydin sutunları tablodaki sırayla girilecektse (yukarıdaki gibi) o zaman altı gizili kismi yazmaya gerek yoktur.
- INSERT INTO Students VALUES (53688, 'Smith', 'Smith@ee', 18, 3.2)

- Tablodan bir kaydi (satiri) silmek icin :
- DELETE FROM Students WHERE S.name = 'Smith'
- ↳ Students tablosundan ismi Smith olan kayitlari sil.
- ★ Sadece DELETE FROM Students yazsaydik, tüm kayitlar silinirdi ama tablo varligini sürdürürdü. (Bos bir tablo olarak kalır.)

- Bütünük Sınırlamaları - (Integrity Constraints)

Her kaydin saglamasi gereken koşullardir. Anahtar sınırlaması, genel sınırlamalar ve domain sınırlamaları olmak üzere üç basitçe incelenir.
(veri tipi sınırlaması)

- ★ Bu sınırlamalar tablo olusturulurken belirlenir. Deni bir kayıt eklenegi veya bir kayıt güncelleneceği zaman bu kurallar çerçevesinde değerler girilir.

Birincil Anahtar Sınırlaması : (Primary Key Constraints)

- * Anahtar, tabloda bulunan bir kaydı diğer kayıtlardan ayıran, bir ya da birden fazla niteliğin bir araya 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 anahtarın 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 anahtاردır. 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 saklındır) 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 anahtar denir. (Genelde diğer tablonunun birincil anahtarına atıfta bulunur)

- ★ Buna örnek olarak, Enrolled tablosunda sid bulunmasını verebiliriz. Bunun nüansı : 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 ekleneneğinde sid değerinin Students tablosunda bulunup bulunmadığı kontrol edilir. Bulunmuyorsa eklenmez.

- Bunun dışında, bir öğrenci kaydını sildiğinde Enrolled tablosundan tüm kayıtları silinebilir. , silinmesine izin verilmez ve bu bilgiler saklanmak istenir ya da silinen öğrencinin Enrolled tablosundaki sid listmine default bir değer yerleştirilebilir.
↳ (Bunu yapmak Enrolled tablosundaki primary keyi bozabilir)

- * Tüm bu seçenekleri foreign key tanımında belirtiyoruz.
- Foreign key komutundan sonra silme ve update için yapılmak istenen sey seçilmelidir:
- Eğer hiubtr sey yazmasa default olarak NO ACTION komutu gelir.
Bunisanda, Students tablosundan bir kayıt silinmek/güncellenecek istendiğinde reddedilir.
- CASCADE, dersen Students tablosundan silinen öğrencinin sahip olduğu sid değerini buludur satırlar Enrolled tablosundan da silinir. Ya da güncellene yapılsrsa Enrolled tablosuna da yansır.
- SET NULL /SET DEFAULT, devlirse Students'te silinen güncellenen kaydin, Enrolled tablosundaki sid yerine null veya default değere güncellenir. (Her iki işlem için 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 )
  
```

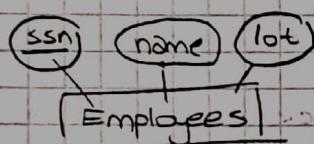
→ Silinirken yapılacak işlem
→ Güncellendirken yapılacak işlem

ER Modelden Tabloya 1 (Logical DB Design & ER to Relational)

Varlık ilişkisi semasındaki varlıklar, ilişkileri ve sınırlamaları nasıl veritabanı haline getireceğimizi göreceğiz.

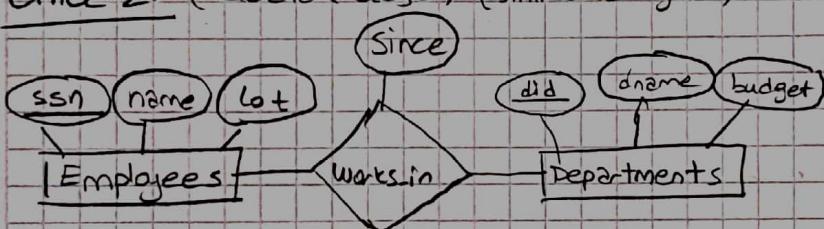
Varlık kümelerinin her biri için bir tablo tanımlanmalıdır.

Örnek-1:



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

Örnek-2 = (içten içe) (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 olmazdı)

Bu iki varlık arasındaki ilişki için ayrı bir tablo yapıyorum. (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 yapılmıştı)

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

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

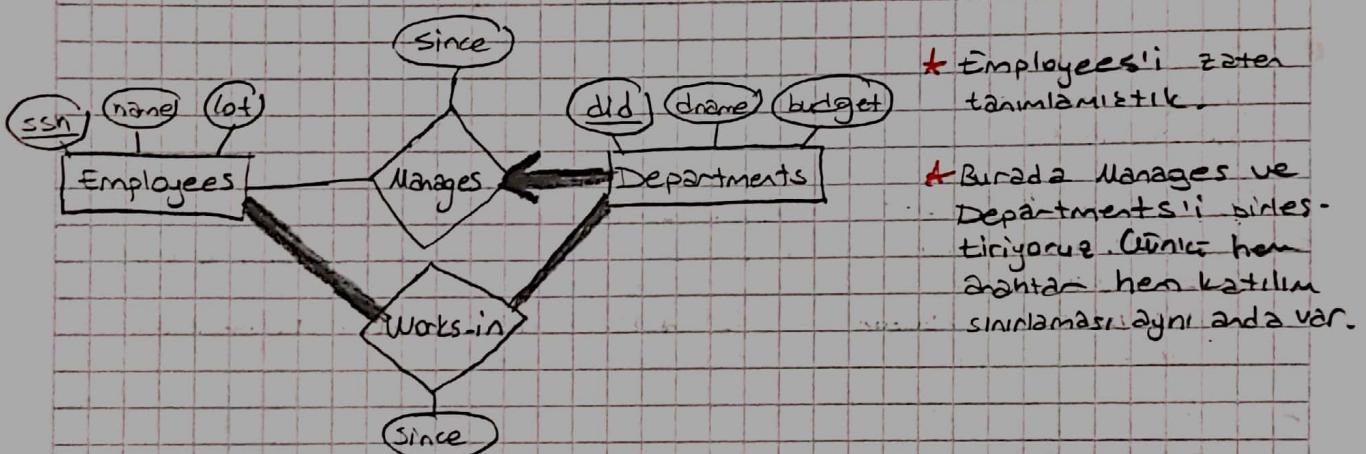
CREATE TABLE Dept-Mgr

```

(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 birleştirilir.
- * Sadece toplam katılım sınırlaması varsa ayrı tanımlıyoruz.

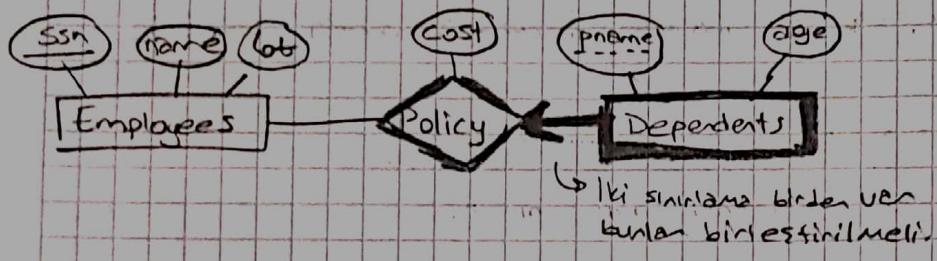


CREATE TABLE Dept-Mgr

```
(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 yapmadık yönetici silindiği zaman departmanın tüm bilgileri silinmiş olur. Bu da mantıksız bir durundur.

- * NO ACTION yapınca, örneğin departman yöneticisi i̇sten ayrıllacaksa önce dept-mgr tablosunda o departman'a yeni bir yönetici atanır. Daha sonra ayrılacek kişi Employees'ten silinebilir.

Örnek-5: (zayıf varlıklar)

CREATE TABLE Dep_Policy

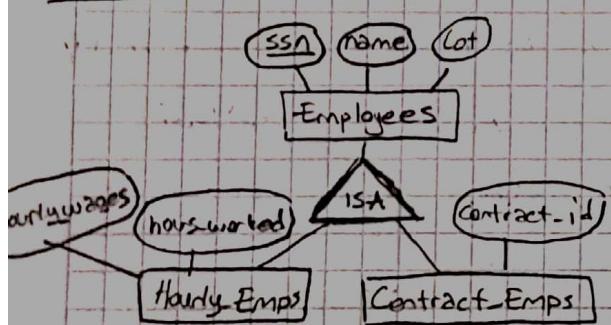
```
(pname CHAR(20),
age INTEGER,
cost REAL)
```

ssn char(11) NOT NULL, → her satırın bir ebevini olmalı.

PRIMARY KEY (pname, ssn),

FOREIGN KEY (ssn) REFERENCES Employees,

ON DELETE CASCADE) → Bir satırın silindiğinde onun çocuklarını da DB'den siliyoruz.

Örnek-6: (ISA hiyerarşisi)

Burada iki yöntem var

1-) Her birisi için 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 zorundaysa :

- 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: Yaşı 21'den küçük olan ve Enrolled tablosunda bulunan öğrencilerin isimlerini ve kayıtlı olduğu derslerden aldığı 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)

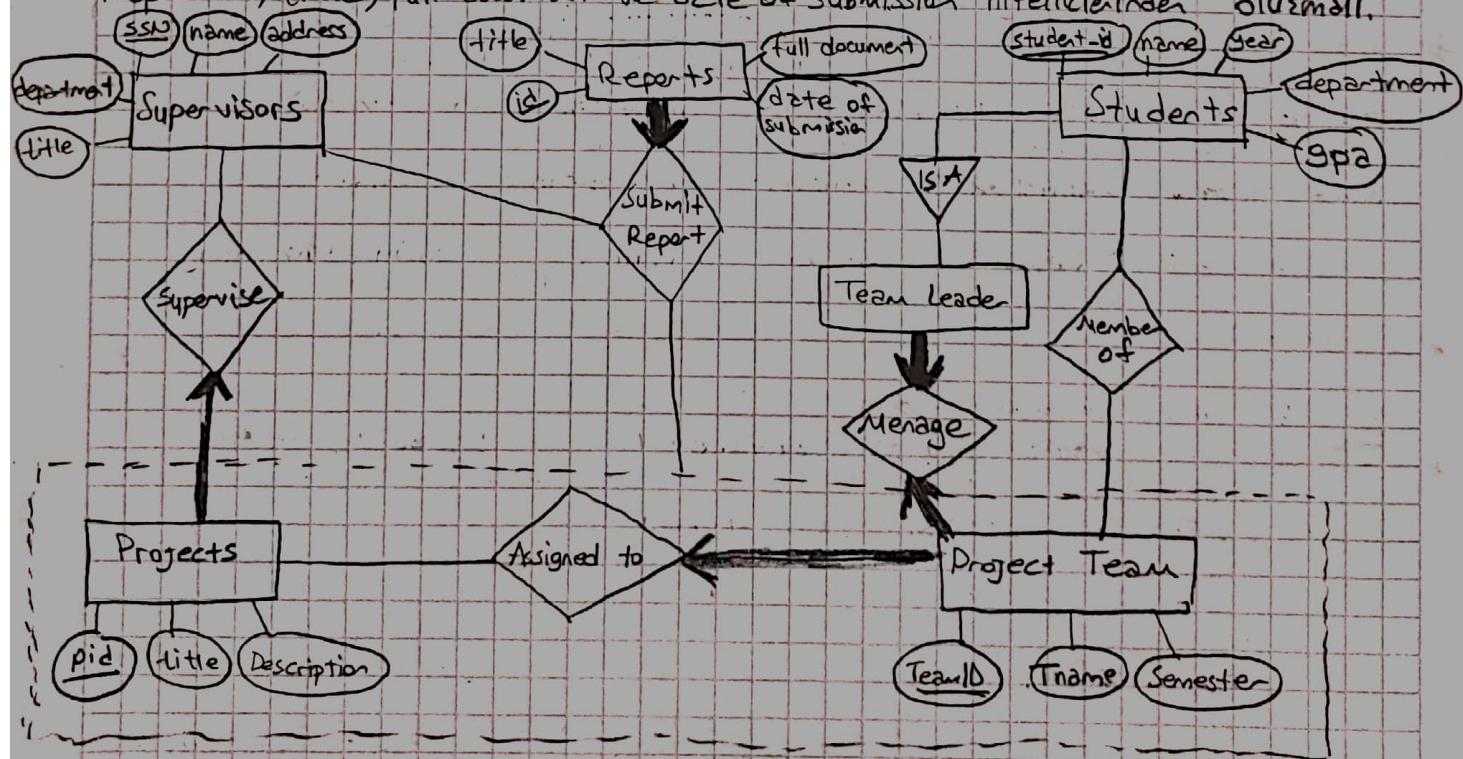
* DROP VIEW tablo_adi komutu kullanılarak view silinebilir.

* Kullanıcılara belirli özel 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. Other Project Team'in tam olarak bir "Team Leader"ı vardır.
- Project ler pid, title, description ve supervisor niteliklerine sahiptir. Her Project kesinlikle tam olarak bir tane supervisor'a sahip olmalı. Bir supervisor herhangi bir sayida Project 'i yönetebilir.
- Project 'ler , Project Teams'ıne atanır. Bir Project Team'e tam olarak bir tane Project atanır. Ama herhangi bir sayida Project Teams aynı project'ı yapabilin.

- Project Teams , Project atanaları halindeki bilgileri Supervisor'a rapor eder. Rapor id, title, full document ve date of submission niteliklerinden olusmali.



DataBase Emması :

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)

İpuçlu: 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.

* iliskiye key constr. ile katılım tarafının prim key'i ilişkisinin prim key'i olun. Eğer key const yoksa ikisinin keylerinin birleşiminde key.

CHAPTER 19:

- Schema Refinement and Normal Forms —

Bir veritabanı şemasını nasıl iyileştirebileceğimizi göreceğiz.

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 veriler belki başlı sıkıntılara sebebi olur. Dikte gereksiz yer kapları, tabloda silme ekleme, 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 diagraminda belirleyemiyoruz. Problemin tanımından bunu çıkarıyoruz.

- $SSN \rightarrow name, hours_worked, rating$ (sosyal güvenlik numarası (ssn) sağıdaki değerleri belirler)
- $rating \rightarrow hourly_wages$ (rating, hourly-wages'ı belirler)

* Bu fonsiyonel bağımlılıkları tanımlayarak gerekiz veriye karar verilir ve tabloları parçalara ayıracagız. Örneğin;

Yukarıdaki tabloda rating ve hourly-wages çıkarılır 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 kurallardır.

A	B	C	D	$AB \rightarrow C$
a1	b1	c1	d1	
a1	b1	c1	d2	
a1	b2	c2	d1	
a2	b1	c3	d1	

* 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 için $K \rightarrow R$ fonksiyonel bağımlılığı tanımlanabilir. (R = tablodaki tüm nitellikler)

Örnek:

Geçen sayfaki tablo örneğine bakarsak, (Tablo ya ismiyle ya da niteliklerini 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 değeri 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 denedikten 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 elemanlar için sona olusur. Bu 'probleme' update anomaly denir.
- * Rating değeri 7 olan bir kişiyi eklemek için onun hourly-wages değerini de biliyor olmam lazımdır ki tablodan tutarsızlık olmasın. Buna da insertion anomaly denir.
- * Ratingli 5 olan tüm personelleri tablodan silseysek eğer. 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 yaptığımız gibi tekrar eden veriler için yeni bir tablo oluşturuyoruz. Ana tabloya sadece Rating değerini ekliyoruz. Böylece update yapacağımız zaman sonraki oluşturduğumuz tablodan tek bir değişiklik yaparak rating ve hourly-wages değerlerini tüm çalışanlar için erit hale getiriyoruz. Aynı zamanda insertion ve deletion anomaly'de çözümün

HAFTA XIII. SON

Örnek: Contracts ($\underline{c}_id, \underline{s}_id, \underline{j}_id, \underline{d}_id, \underline{p}_id, \underline{q}_ty, \underline{v}_alue$)

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

→ Transitivity kullanarak : $JP \rightarrow CSJDPQV$

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

→ ilk eşitliği kullanarak : $SD \rightarrow CSJDPQV$

* Bu aksiyonları 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 gözlemlemektir.

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

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

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

* A^+ bulma işlemi şu şekilde yapılır:

Once A'nın kendisiyle başlar (1)

Kurallarda sol tarafta sadece A olan kuralların

sağ tarafını da yazıyoruz (2). Şimdi içine

AB oldu AB'inin herhangi bir alt kümесinin veya

kendisinin kuralın solunda olduğu bir fonk. bğmlilik

var mı diye bakılır. (A, B, AB sequençlerine bakılır). A kuralını

aldığımız için tekrar alımıyoruz. B kuralını alıyoruz ve C'yi.

kümeye dahil ediyoruz. (3) ABC'inin herhangi bir alt kümesi

bu solda bulunmuyor. Bu yüzden kümeye böyle katılır. Son halde

sağ tarafta E olmadığı için, $A \rightarrow E$ fonk. bğm. F^+ da yoktur. denir.

→ $AD \rightarrow E$ F^+ da var mı?

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

$$= \{ADB\}$$

$$\{ADBC\}$$

$$\{ADBCE\}$$

* 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 kümeli 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 deriz.
- * $A \rightarrow B$ 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 olmayacağı garanti ediyorum.

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 kurallar biri sağlanıysa
bu tablo BCNF idir. Dolayısıyla
insert, update, delete anomaly 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)
- X, R iin anahtardır.
- A, R iin olan bazı anahtarların alt kümelerdir
 - (Primtayobi) (Parçasıdır)
 - (Super-tayobi)

} Bunalardan herhangi birini sağlaması gerekiyor

* Sorunlu tabloları önce BCNF 'e dönüştürmeye çalışınız. Eğer olmazsa 3NF'ye 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 kümeli değıldir. (Kosul-1 sağlanmadı)
- Bu tablo iin anahtar \underline{SBD} olur. (bu anahtarın bir alt kümeli degi birinci fonk. bğm. dolayı)
- Fakat ikinci fonk. bğm. dolayı anahtar \underline{BDC} 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'lerine yol açmaz.

Tabloyu Ayırma : (Decomposition of a Relational Schema)

n elemanlı bir tabloda verilen tüm fonk. bağımlılık için (örneğin 3NF'ye dönüştürmek istiyorsak) 3NF'ye göre bu fonk. bağımlılıkları sağlanıp sağlanmadığını kontrol ediyoruz.

Örnek: SNLRWHT nitelikleri olan tabloda $S \rightarrow^* SNLRWHT$ (I)

$R \rightarrow^* W$ fonk. bağımlılık verilmiş olsun. (II)

- 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 değildir. 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 sağındakilerin niteliği (W), nitelik kümelerinden çıkarıp geriye kalanlarla başka bir tablo yapıyoruz. Yani son durumda elimizde 2 tablo var. Birinin özellikleri SNLRH, diğerinin özellikleri RW. (Galiba Örneğindeki gibi)

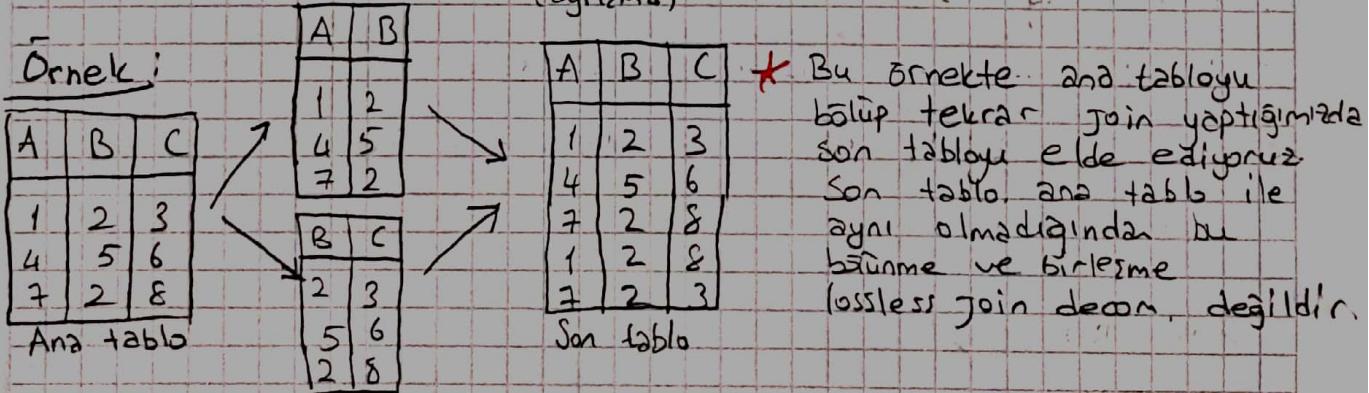
* Ayırma işleminden sonra, 3NF'ye uymayan tablo varsa sorunlu tablo bölmeye devam edilir.

! Bir tabloyu daha küçük parçalara böldüğümüzde su sorunlarla karşılaşabiliriz:

- Galisan örnekinde yola cıkalım (SNLRH ve RW tabloları var)
- Galisan 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 lazımdır. Bazı hatalarda veri kaybı olabilir. (Galisan örneği için böyle bir durum yoktur.)
- Bir tabloyu böldüğümüz zaman bazı fonk. bağımlı sağlanması sağlanmadığına takmat için küçük tabloları birleştirmek gerekebilir. (Galisan 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)



* Lossless join decom olmasi 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 ikermez.

ve C, B'nin elementi değildir.

→ BCNF'yi sağlamadığı için tabloyu bölüyoruz. 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 aldığımızda F^+ 'ya eşit oluyorsa bu bir dependency preserving decompositiondur.

$$(F_X \cup F_Y)^+ = F^+$$

Örnek: CSJPQV nitelikli tabloda C keydir ve $JP \rightarrow C \quad SD \rightarrow P$ fonk bağımlılık verilmiştir.

BCNF decom. yaparsak; SJ DV 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 bağımlılıkları olsun. Dijelimki bu tabloyu AB ve BC diye böldük.

? Bu bölünme dependency preserving mi?

→ AB tablosunun fonk. bağımlılık 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 kapsam. Bu durumda $C \rightarrow A$ basta kaldırıg için cevap HAYIR olur.

? Bu bölünme lossless decomp mu?

$$\{AB\} \cap \{BC\} = \{B\} \quad \text{Eğer } B \quad \underline{\text{AB'yi}} \quad \text{veya } \underline{\text{BC'yi}} \text{ belirlerse kabul.}$$

Bunun için $B^+ = \{B\}$

$$\begin{array}{c} \{B\} \\ \{BC\} \\ \{BCA\} \end{array}$$

Bölünen tablolar.

* Cevabımız Evet.

BCNF'te Bölümme : (Decomposition into BCNF)

Bir tabloyu BCNF'ye çevirirken tablo üzerindeki fonk. bağımlı.
tek tek bölgelere BCNF'e uygun fonk. bağımlı bulunduğu
zaman o fonk bağımlılıktaki 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 II III IV

- I. Kuralda herhangi bir sorun yok (BCNF'ye uygun)
- II. Kuralda JP'de bir anahtarı asılarda burada da sorun yok.
(JP'den dolayı)
- III. Kural BCNF'i saglamaz. Çünkü:

SD anahtar değildir ve P, SD'nin elemanı değildir.
(closure'ı allnarak bulunur)

$$SD^+ = \{SD\} \\ \{SDP\} \rightarrow \text{anahtar değil.}$$

- IV. Kural BCNF'i saglamaz. Çünkü:

J tek başına anahtar değildir ve S, J'nin alt kümlesi değildir.

$$JT = \{J\} \\ \{JS\} \rightarrow \text{anahtar değil. (S'nin tek başına solda olduğu kural yok!)} \\ \text{ondan } SD \rightarrow \text{e kuralını dahil etmediğimizde}$$

* BCNF': saglamayan fonk. bağımlı. herhangi birini alıyoruz (örn: $SD \rightarrow P$)

Bu fonk. bağımlı nitelikler için bir tablo yapılır. (SDP için). Kuralın

sagindan eleman ana tablodan çıkarılır. Sonuca elimizde

C SJ D Q V ve SDP tabloları olusur. Daha sonra tekrar tüm

kurallara bakılır. (I ve II'de sorun yok fü). III'de de sorun
(P tablodan yok artik)

kalmaZ çünkü SD artik 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:) anahtar değil: $J^+ = \{J\}$ ve $S \neq J$
ITS

* Anahtar olmas iin bölümüz ana tablodaki tüm elemanları
içermeliydi (CS) DQV (P olmayan tablo)

143

* IV fonk. bağımlılıktaki sorunu gözlemek iin ana tabloyu bir kez daha bölüyoruz. J ve S'yi birlestir (JS), S'yi ana tablodan at. Sonuç olarak elimizde: CJDQV, SAP ve JS tabloları var.

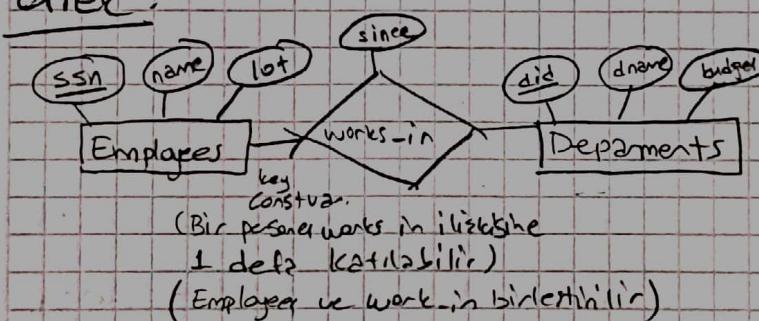
* Bu yöntemle yapılmış bölünme her zaman lossless going denemektir. Farklı dependency preserving olma garantisı yoktur. (Zaten bu örnekte de II. kuralı elde edemediğimiz iin dependency preserving değildir)

* 3NF'e dönüştürmek de BCNF'e dönüştürmek gibidir.

* Fonk. bağımlı. ER diagramlarının iyileştirilmesinde de kullanılır.

(Employee metrisinin birdesini)

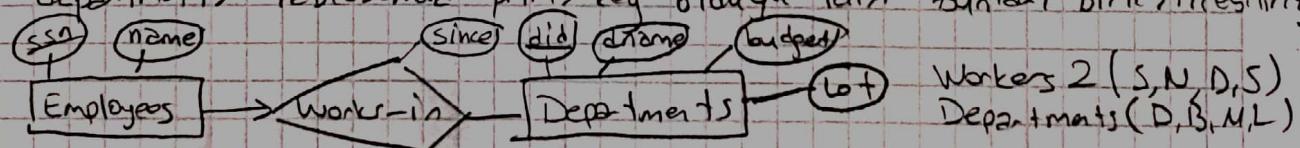
Örnek:



Workers (SN, L, D, S)
Departments (D, M, B)

Örneğin: $D \Rightarrow L$ diye bir sınırlama olsun. (Aynı departmant dalında çalışanlar aynı vendijde olacak)

! D \Rightarrow L fonk. bağımlılık BCNF veya 3NF'e uymaz. Dolayısıyla bu tabloyu ikiye böleriz. (SNDs ve DL diye). D niteliği yeni oluşturulan tabloda ve departments tablosunda prim. key olduğundan bunları birlestirebiliriz.



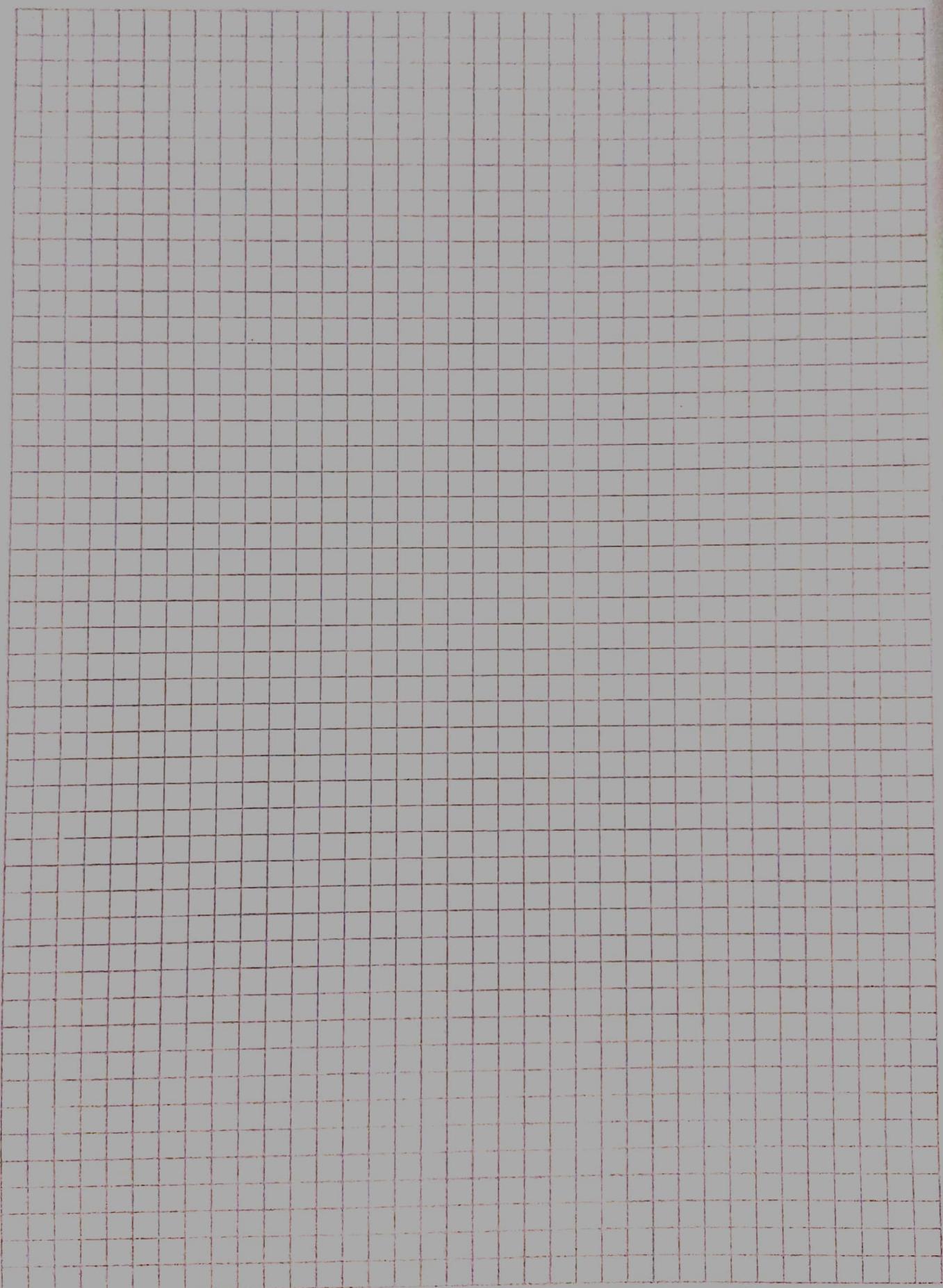
HAFTA XIV. SON

HAFTA XV. SON

DÖNEM

SONU

144



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 istediğimi ve ile ilgilenir

* Soru, tablo örnekleri üzerinde calısır. Soru da input (tablo veya tablolar) ve output (tablo) vardır.

Basit islemler:

- Selection ($\sigma_{\text{seçim}} - \sigma$): Input olarak bir tablo alır. Bu tablodan belirli kriterlere uygun satırları seçer.
- Projection (π): Belirtilen sutunları ekran da gösterir.
- Cross-product (Kartezyen çarpım - X): İki tabloyu birleştirirken kullanılır.
- Set-difference (Küme farkı - -): Birinci taboda olan ama ikinci taboda olmayan kayıtları gösterir.
- Union (Birleşme - U): Her iki tabloyu da birleştirir.
- Kesişim, Katılım, Bölme (intersection, join, division)

Projection : (π)

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

tablo
ismi
 $\pi_{sname, rating} (S2)$

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

- INPUT -

- OUTPUT -

* Bu isekinde tekrarlıyan satırlar varsa, tekrar eden satırlar elenin örnegi:

$\pi_{age} (S2)$

age
35.0
55.5

→ Tekrarlayan veriler silindi

Selection : (σ)

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 yapılabilir. Bu durumda projection ile tablodan alınan sütunlara Selection uygulanabilir. Aksi halde hata alırsınız.

Union - Intersection - Set Difference ($\cup \cap -$)

* Birleşim, Kesim ve fark işlemlerini 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.

$S1 \cap S2$
→
(intersection)

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$S1 - S2$

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 sütunları.

Örnek: Veren sayfadaki S1, R1 tablolarını birleştirin.

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 satırıyla eşleştirildi.

Join (\bowtie)

Kartezyen çarpımın bir türüdür. Kartezyen çarpımı üzerine bir selection işlemi uygularsağ 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 değerini S1.sid değerinden büyük olan veriler seçildi.

- Eğer selection işleminde beliri bir özelliği (veya özellikleri) eşit olan ifadeler arıysak buna equi-join denir.

Örnek:

$S1 \bowtie_{R1.bid} R1$

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.

- İki tabloda ismi aynı olan sütunlar varsa bunların değerlerini eşitleyerek 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 :

Sno	Pno
S1	P1
S1	P2
S1	P3
S1	P4
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4

Pno
P1
P2

Pno
P1
P2

Pno
P2
P4

A

Pno
P1
P2

Pno
P2
P4

B2

Sno
S1
S2
S3
S4

Sno
S1
S4

Sno
S1

* Burada mantık eudur.
B1 tablosundaki elemanları
2 karşılık gelen Sdeğerler.
A tablosunda var mı?
Örneğin A/B2 için B2
tablosundaki p2 ve p4'e
ayrı anda karşılık
gelen elemanlar var mı?
Evet S1 ve S4.
Örneğin A/B3 için B3
tablosundaki p1, p2 ve
p4'e A tablosunda
ayrı anda karşılık
gelen eleman 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 : $\Pi_{\text{sname}} ((\sigma_{\text{bid}=103} \text{Reserves}) \bowtie \text{Sailors}) = \underline{\underline{\text{rusty}}}$

58 103 11/12/96 → Jini ile sideleri eşit olanlar kaldı

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, ($O_{\text{color}=\text{'red'}} \vee O_{\text{color}=\text{'green'}} \text{Boats}$))

$\Pi_{\text{name}} ((\text{TempBoats} \bowtie \text{Reserves}) \bowtie \text{Sailors})$

→ 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 edelerin id'si

P (TempGreen, $\Pi_{\text{id}} ((O_{\text{color}=\text{'green'}} \text{Boats}) \bowtie \text{Reserves})$)

→ Yeşil reserve edelerin id'si

$\Pi_{\text{name}} ((\text{TempRed} \cap \text{TempGreen}) \bowtie \text{Sailors})$

→ Buluların kesişimi

→ 5-) Tüm botları reserve eden kışilerin isimleri.

P (TempSids, ($\Pi_{\text{id}, \text{bid}} \text{Reserves} / (\Pi_{\text{bid}} \text{Boats})$))

$\Pi_{\text{name}} (\text{TempSids} \bowtie \text{Sailors})$

———— HAFTA IV. SON ————

CHAPTER 5:

- SQL : Queries, Constraints, Triggers -

Gelen haftaki örneği kullanıyorum.

- 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 kiralayabilir.
 Eğer sid ve bid primary key olsaydı;
 → aynı kişi aynı botu yalnızca bir defa kiralayabilir. (tarih değişse birer)
 Eğer sadece sid primary key olsaydı
 → Bir kişi yalnızca bir bot kiralayabildi. (Baska hiçbir zaman bot kiralayamaz)

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 sonda ; 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 çarpımı tabii tutulur.
 Koşulu sağlayan satırların sname sütunu doldurulur.

(Bu sorgu 103 numaralı botu kiralayan denizci(ler)in isim(lerini) doldurur)

SONUÇ : +

sname
Rusty

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 edilin.
 Çünkü from cümlesinde tablolara kisa isim verilerek bu isimlerin
 where cümlesinde kullanırı saglanır. 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 ettiyse yukarıdaki
 soru sonunda 5 kere sid'si yazılır. 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 gördüğün 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 cümlesiinde
 kullanılabilir. Soru sonucu
 oluştururan sütuna isim vermek
 için = veya AS kullanılabilir.

* WHERE cümleinde ise 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 deseydiğimizdeki soruda kırmızı veya AND koysaydık sonucu, AND (B.color='red' OR B.color='green'))bos dönerdi(bir bot hem yeşil hem kırmızı olamaz.) Bu yüzden aşağıdaki gibi yazıp yerine INTERSECT yazarsınız.

* 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 işlemleri yaparken iki select sorusunu sonucunda (union) (intersect) (except) oluşan tabloların aynı sayıda sütunlarından oluşması gerekiyor. (Bu örnekte her iki soru sonucu da tek sütunkut tablolar oluşturur.)

İç içe Sorgular:

SQL'in genel Özelliklerinden biridir. Bir select sorusunun içine başka bir select sorusu yazabiliriz. İç içe soru parantez içinde olmalı ve where, from, having cümlelerine yazılmalıdır.
(en çoktu)

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 değerlerine 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 kişilerin 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)
```

★ içteki 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 kişilerin isimleri döner)

Küme Operatörleri :

- IN, EXISTS, UNIQUE → küme içinde kıyaslama yapmak için kullanılır (NOT ile de kullanılır. NOT EXISTS... gibi)
- ANY, ALL, IN operatörleri $>$, $<$, $=$, \geq , \leq , \neq ifadeleri ile kullanılır.

Örnek : ismi Horatio olan kişilerin rating'lerinin herhangi birinden daha büyük ratinge sahip kişilerin tüm bilgilerini göster.

```
SELECT *  
FROM Sailors S  
WHERE S.rating > ANY (SELECT S2.rating  
                      FROM Sailors S2  
                      WHERE S2.name = 'Horatio')
```

★ içteki ve distati soruda aynı tablo farklı isimlerle kullanılmış.
★ ALL kullanmadık içteki tablodan dönen rating değerlerinin hepsinden büyük olan ratinglere sahip kişilerin bilgileri döndürülecekti.

★ Benzer bir mantıkla kırmızı ve yeşil botları, reserve eden kişileri de iç içe sorğu ile bulabiliriz. (INTERSECT kullanmadan)

HAFTA VI. SON

SQL'de Bölme

★ Bölme işlemi için sayıla 151'e git.

Örnek: Tüm botları reserve etmiş kişilerin isimleri.

I. yöntem

• `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 botları reserve etmiştir.

(Bu döşetlenmeyen)

II. yöntem

• `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 tekranan satırlardan)
`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.

Örnek:

`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 soru
X $\xrightarrow{\text{Dönen deger}}$
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 soru
 $\checkmark \xrightarrow{\text{Dönen deger}}$
değildir.

sname	age
lubber	55.5

★ Yukarıdaki soruyu yazarken $S.age = (\text{SELECT} \dots)$ olmalı. $(\text{SELECT} \dots) = S.age$ olursa hata verir.

Gruplama :

SELECT gösterilecek - sütunlar
FROM tablolar.
WHERE koşul
GROUP BY sütun(clar)
HAVING grup koşulu.

} From cümlesindeki tabloların katetlenen
varlığı sağlanır. WHERE cümleşindeki şart
göre SELECT cümleşindeki sütunlar sağlanır.
Yalnızca GROUP BY cümleşindeki sütunlar direkt
olarak SELECT'e yapılabilir. SELECT tekli
diğer sütunlar aggregate operatörleri ile yapılabilir.

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	65.0
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 2 kez
Bunları rating değerlerine
göre grupper. Eger aynı
ratingden en az iki
kisi varsa bunları seçti.
Bu seviyen grupların içindeki
en küçük bireyler ve ratinge,
döndürildi.
(Slaytta güzel anlatmış gibi)

★ ratingi 10 olan iki eleman var. Fakat
zarbanın yaşı 16 olduğu için
cumlesinde eleman ve groupby katılmaz.

* having cümlesine :

HAVING COUNT (*) > 1 AND EVERY (S.age <= 60)

yazsaydık dönen tablo.

rating	minage
7	35.0
8	25.5

olurdu. \hookrightarrow Burada saylenen sayı & gruptaki her elementin yaşının 60'tan küçük olması gereklidir. rating = 3 için 'bob' bu durumu doğrudan tabloya rating = 3 dahil etilmedi.

\hookrightarrow Eğer EVERY yerine ANY yazsaydık herhangi bir elementin ratingi 60'tan küçükse o rating grubunu alır. Bu durumda rating = 3 te tabloda yer almaz.

* where cümlesine

WHERE S.age >= 18 AND S.age <= 60

yazsaydık dönen tablo

rating	minage
3	25.5
7	35.0
8	25.5

olurdu. \hookrightarrow bob en başta eleindi ama ratingi = 3 olan ikisi de aynı yaşta eleman var. Bütün de ratingi en az 3 olacak şekilde istiyordu.

Örnek : Kirmizi renkli botları reserve eden kaç kişi vardır? (Tüm kirmizi botlar aynı yaşı.)

```
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
```

\rightarrow Renk kirmizi 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 > 18 olan) 1 kişi de fazla eleman varsa bunlardan küçük olanın rating ve yaşını döndür. (Yani en az 2 kişi kişi olrsa 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)
```

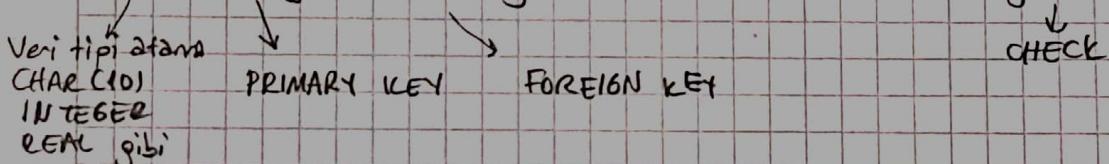
Örnek: Her ratingin ortalamasını bul ve ortalaması en düşük olan ratingi dönden.

Aggregate operatörler
için ike kullanılmaz.

- ~~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)~~

Sınırlamalar

Domain, primary ve foreign key sınırlamalarını gördük. Şimdi genel sınırlamaları göreceğiz.



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 ('Interlace' <>
        (SELECT B.bname
        FROM Boats B
        WHERE B.bid=bid)))
```

* 'Interlace' ismi Interlake olur bottar reserve edilemez. Constraint direkt turlara 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 gerçekleştiğten sonra taze seyein otomatik olarak çalışmasını isteyebiliriz. Bu gibi durumlarda kullanılin Üç kısımdan oluşan

- Event (Triggeri uygulayacak olay)
- Condition (Bu olayın olma şartı)
- Action (Trigger uygulandı ne olacak)

Örnek:

CREATE TRIGGER youngSailorUpdate
AFTER INSERT ON SAILORS
REFERENCING NEW TABLE NewSailors
FOR EACH STATEMENT

→ Event

INSERT
INTO Young Sailors (sid, name, age, rating)
SELECT Sid, name, age, rating
FROM NewSailors N
WHERE N.age <= 18

→ Action

(Kosul yok)

Sailors kısımına ekleme yapılırsa tetiklenecektir. 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, =, <, >, olabilir)

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 halidir.

- Natural Join iki tabloyu birleştirir ve ortak sütunları 1 defa yazır.
(esit sütunları)

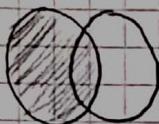
SELECT *

FROM employee NATURAL JOIN department

Normalde employee ve department tablosundaki her iki DepartmentID

değeri de sonucunda tabloda gösterilmeli ama natural join
oldugu için tek degere gösterilir.

- ④ Left Outer Join iki tabloyu join yaparken soldaki tablonun tüm kayıtlarını alır. Soldaki tablodan 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 hesaplamağa göre maliyetinin nasıl təhmin edildiğini öğreneceğiz.

Sorgu Planı :

Verilen bir SQL sorgusunun ilişkisel cebir kullanarak bir ağaca dönüştürülmesi.

Bu ağacın her adımda kullanılan ilişkisel cebir fonksiyonu gösterilir.

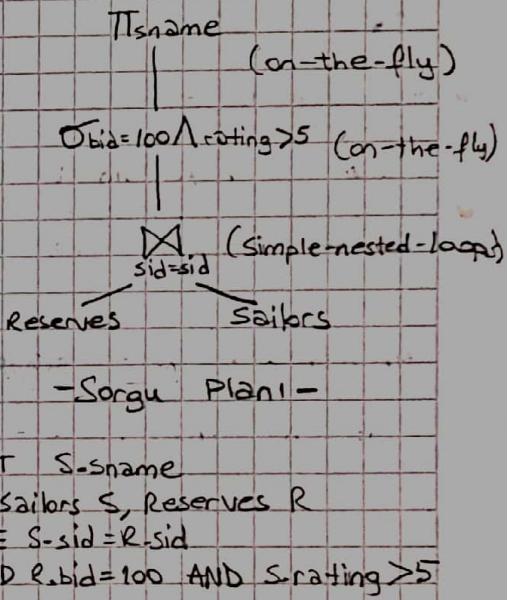
→ Simple-nested-loops algoritması equi-join işlemi yapar.

→ on-the-fly: Bir önceki işlem hesaplandıktan sonra (materialized yazsaydı bir önceki işlemin sonucu diske yazılır ve sonraki işlem bunu disetten okurdu)

* Query Optimizer, sorguyu hazırlarken hangi algoritmaların kullanılacağına karar verir.

→ Query Optimization (sorgu optimizasyonu) için iki husus var: hangi planlar göz önüne alınacak ve her bir planın maliyeti nasıl təhmin edilecek.

* Bir sorgudaki ilişkisel cebir işlemlerini daha düşük maliyetle gerçekleştirmek için = Indexing, Iteration, Partitioning yöntemlerinden biri veya birkaç kullanılır.



```

SELECT S.Sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
      AND R.bid=100 AND S.rating>5
  
```

Indexing: Genelde selection ve join işlemlerinde tercih edilir.

Iteration: Tüm 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 plana 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 'nin 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 yazılmasa)

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 yazılmasa)

Karmaşık Seçim:

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))

(day < 8/9/94 AND rname = 'Paul') OR bid=5 OR sid=3

↓ query optimizer

(day < 8/9/94 OR bid=5 OR sid=3) AND (rname = 'Paul' OR bid=5 OR sid=3)

Seçim İleminde Yaklaşım

Seçim işleminde, en az sayıda disk erişimi yapacak yöntem seçilmelidir. Bu amaca göre index kullanılır ya da dosya baştan sona taranır.

Örneğin:

day < 8/9/94 AND bid=5 AND sid=3

Eğer day üzerinde tanımlı B+ ağacı indexi varsa kullanılır. Eğer day üzerinde hash index tanımlı olsaydı bu kullanılmazdı. Çünkü hash index ile aralık sorusu yapılmaz.

I. Yöntem: B+ ile istenilen aralıkta bulunan day kayıtları kullanılarak bid=5 ve sid=3 olan kayıtlar bulunur.

II. Yöntem: bid ve sid hash indexlerinden 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 Seçim:

```
SELECT *
FROM Reserves R
WHERE R.rname < 'C%'
```

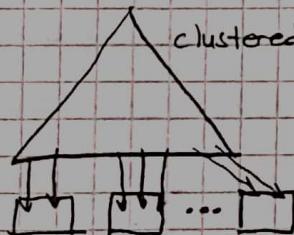
t'smi C harfinden küçük bir karakter ile başlayan reserves kayıtlarının tüm sütunlarını göster.

* 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.

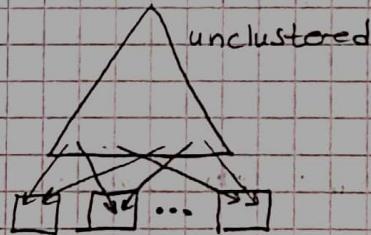
→ Bu tablo üzerinde hiçbir index yapısı yoksa filescan yapılmak zorunda (başta sona tarama). 1000 sayfa olduğuna göre bu sorgunun maliyeti 1000 disk erişimi dir.

→ 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 doğrudan şekilde bulunduklarından daha fazla disk erişimi yaparız. Her kayıt için baktığımız zaman maliyeti (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 iki 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 ($2 * n * \text{gecis sayısı}$) bu tabloyu sıraladığını düşündürsek (gecis=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 ikiliyi bir bucket'a gönderir. Bu işlemin sonunda her bir bucket kontrol edilecek (tekrar için). Bu diğerine göre daha düşük maliyetli olabiliç.

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.

} iç içe döngü olduğu
} için nested loopsdır.

* S tablosu üzerinde index tanımlıysa
disk erişim

$$\text{Maliyet} = M + ((M * P_r)^* \text{ eslesen } S \text{ maliyeti})$$

$M : R$ 'deki sayfa sayısı
 $P_r : r$ 'teki sayfaların 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 yaptıgımızı varsayıyalım. (M : disk döngüde bulunan tablonun sayfa sayısı, P_r : disk tablodaki her bir sayfada bulunan kayıt sayısı)

1-) Sailor iç tablo olsun. $M : 1000$, $P_r : 100$ olsun ve hash index olsun.

\rightarrow surade sid (clustered)
 \rightarrow min key (AHT2)

$$\text{Maliyet} = 1000 + ((1000 * 100) * (1.2 + 1)) = 221.000$$

2-) Reserves iç tablo olsun, $M : 500$, $P_r : 80$ olsun ve hash index olsun (clustered)

sid e için prim R'nin kayıt sayısı, S'nin kayıt sayısı (AHT2)
 key olmadığından 2 de bir den fazla
 sid değeri olabilir.

$$100.000 / 40.000 = \boxed{2.5} \rightarrow \text{yazılım}\downarrow \text{değer}$$

Burada sid prim key değil.

$$\text{Maliyet} = 500 + ((500 * 80) * (1.2 + 2.5)) = \boxed{148.500}$$

* 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 degeridir) aynıysa join işlemini yapar. Eşit değilse soldakı tabloda bir sonraki özelliğe gecilir.

Maliyet = M tablosunu sıralama maliyeti + N tablosunu sıralama maliyeti + $(M+N)$
 sıralama maliyeti $\xrightarrow{2 \text{ okuma}} \xrightarrow{\text{maliyeti}}$

Sıralama maliyeti = $2 * M * \text{gecis sayısı}$ } $M = \text{sayfa sayısı}$

Geçis Sayısı = $\lceil \log_{B-1} \lceil \frac{M}{B} \rceil \rceil + 1$ } $B = \text{buffer pool'daki sayfa sayısı}$.

Örnek: (Burada sadece sayfa sayıları önemli, öncekiden farklı olur)

- Önceki sayfadaki örnek üzerinden gittik.

Sailors = 500 sayfa , Reserves = 1000 sayfa , 300 buffer page , 2 geçis.

\rightarrow Sailors için geçis sayısı = $\lceil \log_{300-1} \lceil \frac{500}{300} \rceil \rceil + 1 = 2 \xrightarrow{\text{ilkisi için de geçerli.}} \xrightarrow{\text{G}_S \text{ (gecis s)}}$

\rightarrow Reserves için geçis sayısı = $\lceil \log_{300-1} \lceil \frac{1000}{300} \rceil \rceil + 1 = 2 \xrightarrow{\text{G}_R \text{ (gecis r)}}$

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 performansa 148.500 adet disk giriş çöküğü yaparken burada sadece 7500 kez yaptıktı.

System R Optimizer

Düşük maliyetli sorgu planı oluşturmak için DB'ler tarafından kullanılan bir yöntemdir.

- ★ Önceden, selection ve projection işlemlerini mümkün olduğunda join işleminden önce gerçekleştirir. Böylece join işlemine katılacak tablolardan boyutlarını kırıltımış 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 olacağından diske yazmak çok maliyetlidir.

Maliyet Tahmini:

Sorgu planı oluşturulurken maliyetinin de hesaplanması isteniyor. Bu hesaplamayı sorgu planının tümü için yaparsa çok uzun süren bunun yerine tahmin yolu ile bunu yapar. Bu tahmin, seçilen algoritmeye ve tablonun boyutuna bağlıdır.

Bu tahmin şu formülle olur:

$$\rightarrow \text{max kayıt sayısı} = \text{from cümlesiındaki tablolardan element sayılarının çarpımı}$$

$$\boxed{\text{Maximum kayıt sayısı} * \text{tüm reduction faktörlerin çarpımı}}$$

$\rightarrow \text{red.faktör} = \text{where cümleındaki her bir term için ayrı ayrı hesaplanır.}$

$$\underline{\text{col} = \text{value}} \rightarrow 1 / N\text{keys}(I)$$

$$\underline{\text{col } 1 = \text{col } 2} \rightarrow 1 / \text{MAX}(N\text{keys}(I_1), N\text{keys}(I_2))$$

$$\underline{\text{col} > \text{value}} \rightarrow (\text{High}(I) - \text{value}) / (\text{High}(I) - \text{Low}(I))$$

↓
i indisindeki en yüksek anahtar değeri.

$N\text{keys}(I) \rightarrow \text{index üzerinde tanımlı anahtar sayısı}$

Örnek:

```

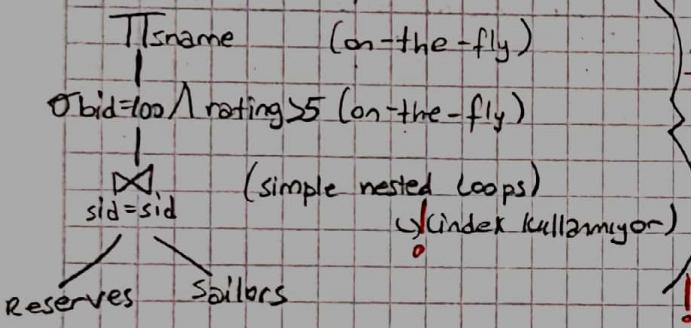
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
R.bid=100 AND S.rating>5
  
```

Reserves

Her kayıt 40 byte, her sayfada 100 kayıt
toplam 1000 sayfa

Sailors

Her kayıt 50 byte, her sayfada 80 kayıt
toplam 500 sayfa.

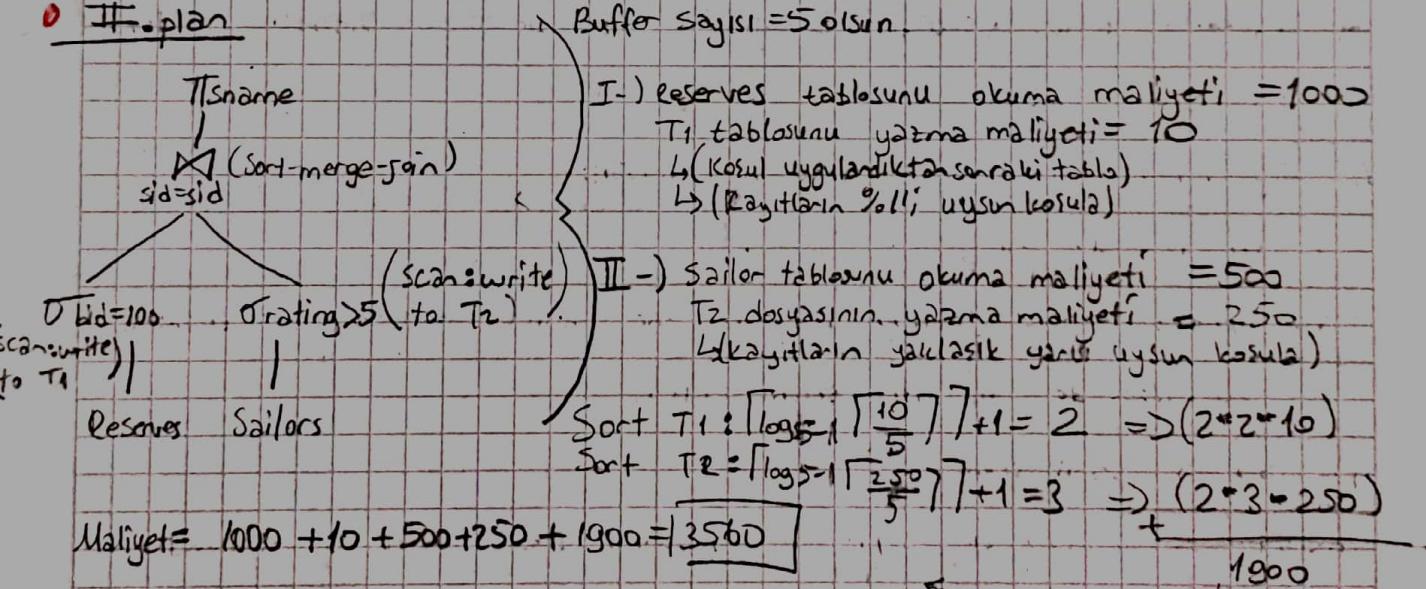
① I.plan

$$\text{Maliyet} = 500 + 500 * 1000 = 500500$$

her bir S için bir R okuyor

(Sailor dizi döngüde kaç sayfa sayılır
az diğer farklı dizi 1000 + 1000 * 500 olurdu)

! Diğer işlemler on-the-fly olduğundan
diske yazılmazlar. O yüzden hesaplamaya
dahil degiller

② II.plan

$\sigma = 100$ (%1'i olsun)

$\sigma > 5$ (%50'si olsun)

173

(selection için)

III. Plan

Ttsname (on-the-fly)

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)

Rating > 5 (on-the-fly)

$$\text{Toplam maliyet} = 1200 + 10 = \boxed{1210}$$

sid=sid (index nested loops)

pipelining ile

(hash index) $\sigma_{bid=100}$ Sailors (hash index)
burada da var.

Reserves

HAFTA IX SON

CHAPTER 20:

- Physical Database Design -

Bu chapter'da ; view tanımlama, bir nitelik üzerinde hangi indexin olusturulacağı,
kanuları ele alınıyor.

İş yükü : projemizde sık kullandığımız sorular

* Bir tabloyu oluştururken

• Hangi indexleri hangi tablolar üzerinde ve hangi nitelikleri baz alarak
olusturmamız?

• Bir tablo üzerinde kaç index tanımlanmamız?

gibi sorulara yanıt aranır.

* Normalizasyon kullanarak BCNF'ye dönüştürülmüş tablolar üzerinde

(ve 3NF)
yapılan sorular düşük performansı yaşasabilir. Bu sorunu gidermek için
join işlemleri yapıldıktan sonra gecilebilir. (denormalization)

Join iin Index Seçimi

- Index nested loops algoritmasında iu döngünün bir hash indexinin olması performansı artırır. Ayrıca, eger join niteliği (σ_{ia}) iu döngü iin primary key degilse 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) karar vereceğiz.

Bu soruya göre
 hangi indexleri
 olusturmamız gerektigine
 karar vereceğiz.

- Öncelikle where cümlesine bakıyoruz. (join yapmadan önce verileri olabildigince kümültmeye çalışıyoruz ilk önce selection işlenlerini yaparak) dname iin bir index olusturmalıyız ki ismi 'Toy' olanları hızla 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 dis döngüde bulunuyor. Dis 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 işleminin fromden 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 BT index, hobby iin hash index). Fakat oluşturulan her indexin diskte yer kaplayacağı unutulmamalıdır. Aynı zamanda tablo üzerinde değişiklik 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 = 'Toy' 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 şemasını oluşturduktan sonra sorguların daha hızlı çalışması için birçok işlem yapılabılır. (3NF formuna dönüştürmek, BCNF'e dönüştürmek, normalizasyonu geri almak, bazı tablolara ilave sütunlar eklemek, daha fazla decomposition yapmak, yatay bölme ...)

Örnek :

Contracts (Cid, sid, jid, did, pid, Qty, Val)

Bu tablo üzerinde şu 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分割 tablodan atıyoruz) ama dep. preserving degildir. (Çünkü $JP \rightarrow C$ bosta kalıyor).

↳ Dikkatimiz, sorguların büyük bölümünde Q, P ve C birlikte kullanılıyor. Böyle bir durumda oğlu zaman yukarıdaki bölgelerimiz iki tabloyu (CSJDOV ve SDP) join etmemiz gerekiyor. Bu da çok maliyetli olduğundan tabloyu original halde birleştirmek 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 $1\text{-}1 \rightarrow B$ 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. ($\text{JP} \rightarrow C$ 'den dolayı)
- SDP, CSJ DQV ve CJP: preserving sorunu düzeltti (Ama burada da veri tekrarı var)

* İki tabloya bölerek ve $\text{JP} \rightarrow C$ fonksiyonel bağımlılığını kontrol edebilecek bir Assertion tanımları \exists (Assertion, birden fazla tabloyu 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 içeriyorsa CD'yi ayrı tablo yapmak mantıklı olabilir (CSJ DQV tablosunda işlem yapmaktaşa CO'de işlem yapmak daha performanslıdır)

Horizontal Decomposition (Yatay Bölünme)

Daha önce gördüklerimiz dikey bölünmeydi. Yani tabloyu sütunlardan ayırmıştık. Ama yatay bölünmelerde tablo satırlarını bölünür. (Yani bölünen tablolardaki sütunlar hala aynı)

Örneğin; Contract tablosunda value > 10000 olan satırlar için çok fazla soru geliyor. O zaman bu tabloyu yatay bölüyoruz. (>10000 ve <10000 diye) (small large)

Sorguyu Yeniden Yazma:

Bazen yazdığımız bir sorgu beklediğimizden daha kolay anlaşılır. 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 işlemleri varsa.
- ★ in iae sorguları tek blokta yazmaya利于!

```
SELECT DISTINCT *
FROM Sailors S
WHERE S.name IN
  (SELECT T.sname
   FROM YoungSailors T)
```

```
SELECT DISTINCT *
FROM Sailors S, YoungSailors Y
===== WHERE S.sname = Y.sname
```

↳ in sorguda distinct, aggregate operatörler veya null karşılaştırması varsa (count, min, max) yukarıdaki gibi yapamayız.

★ Sorgunun daha iyi anlaşılması için DISTINCT yazmaktan kaçın.

★ Gerçekten ihtiyaç yoksa GROUP BY ve HAVING kullanmaktan kaçın.

```
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

Dos işlenmedi (Bayram Tatili)

HAFTA XII. SON

CHAPTER 16 :

- Transaction Management -

Transactions :

Bir kullanıcının veritabanı üzerinde bir işlem gerçeklestirmeye transaction denir.
(okuma-yazma)

Birden fazla kullanıcı bir DB üzerinde aynı anda işlem (transaction) gerçekleştirmek istiyor olabilir. Bu işlemlerin yönetilmesi gereklidir.

* Kullanıcı programları DB üzerinde eşzamanlı (concurrent) olarak çalışır. Bu eşzamanlı çalışma sırasında DB'deki verilerin tutarlı olması gerekiyor
(Önceden gördüğümüz 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 doğrultusunda ve bu transaction başlamadan önce DB de tutarlıya transaction sonrası DB yine tutarlı olmalıdır (Transaction kodları düzgün yazılmalı)
- Isolation: Birden fazla transaction eşzamanlı çalışırken her birinin diğerlerinden izole olmalıdır. Yani onların çalışmalarından 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 tablo cizelge denir.
(Schedule)

→ Bu iki transactionun es zamanlı
calışacığını varsayıyoruz.

* $T_1 : B$ hesabından 100\$ parayi, A hesabına aktarıyor.

* $T_2 : \text{Her iki hesaba da } \%6 \text{ fazit uyguluyor.}$

→ Her iki hesapta da 200\$ olduğunu varsayıyoruz. T_1 ve T_2 bittiğinden
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
geçici bir cizelgedir. Son durumda
 $A = 318 \$$, $B = 106 \$$

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)

* Aslında DBMS transactionları yukarıdaki gibi işlemler şeklinde görmez

bu şekilde görür (cizelge-2 baz alındı)

$T_1 : R(A), W(A),$

$R(B), W(B)$

$T_2 : R(A), W(A), R(B), W(B)$

A nesnesini
oku(read) A nesnesini
yaz(write)

T_1 T_2

→ Bu şekilde de
gösterilebilir

$R(A)$	$W(A)$
$R(A)$	$W(A)$
$R(B)$	$W(B)$
$R(B)$	$W(B)$

* Bir cizelgedeki transactionların sırası
değiştirilememeli.

Scheduling Transactions

- Serial Schedule : (Seri çizelge) : Bir transaction başlayınca , bitinceye kadar çalıştırılır. Yani sırayla baştan sona herşi çalışır. (Geçiş yok)
- Equivalent Schedule : (Eşit çizelge) : Eğer iki çizelgenin son hallerindeki etkisi aynıysa bunlar eşit çizelgelerdir. Örneğin; ömeki sayfada bulunan örneğin çizelgesi ve 1 numaralı çizelge . (İkisinin de net etkisi aynıdır)
- Serializable Schedule : (Seçilebilir çizelge) : Bir çizelgenin net etkisi bir seri çizelgeye eşitse (örneğin 1 numaralı çizelge) bunun seçilestirilebilir deriz. Bu tarz çizelgeler çalışmadan önce DB tutarlıysa bunlar çalışıktan sonra da tutarlı durumda olur. (2 numaralı çizelge bu sınıfa girmez ve DB'nin tutarsız olmasına neden olur)

Eşzamanlı Çalışma 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 değiştiirdiği veriyi T2 okudu ve T2 bu veri (A) üzerinde değişiklik yapıp DB'ye commit etti. Fakat T1 belli bir yerde abort (iptal) edildi. Transactionların atomikdma prensibinden 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) :

$T_1 = R(A),$	$R(A), W(A), C$
$T_2 = R(A), W(A), C$	

→ Bir transaction güncellemediği bir veriyi tekrar okudüğünde bu veri değişmişse turada problem vardır.

★ T_1 A'yi okudu sonrasında T_2 A'yi okudu, değişiklik yaptı ve commit etti. T_1 A'yi güncellememiş olmasına rağmen güncellenmiş bir A verisini DB'den okudu.

Overwriting Uncommitted Data (WW Conflicts) :

$T_1 : W(A),$	$W(B), C$
$T_2 : W(A), W(B), C$	

→ Commit edilmemiş bir veri üzerine tekrar bir verinin yazılması probleme yol açan.

★ T_1 A varisini değiştirdi ama henüz commit etmedi. T_2 başladı ve A'yi güncelledi. Burada WW Conflict olur. Çünkü T_1 commit etmediği için onun güncellediği verinin üzerine başka bir transaction güncelleme yaptı ve T_1 'in yaptığı güncelleme kayboldu. B'nin güncelleme işleminden (T_2 iden T_1 'e geçerken) sorun yok. Çünkü T_2 geçişten önce commit etti.

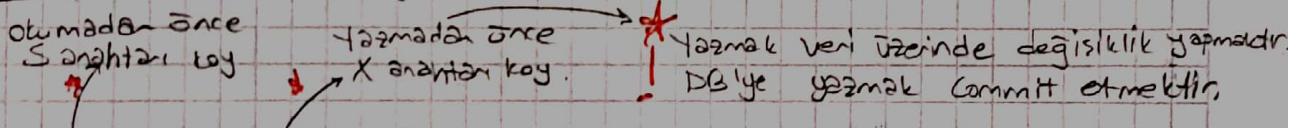
→ Bu taraş sorunlarından korunmak için DBMS bazı yöntemler kullanır.

Lock-Based Concurrency Control

Önceki kısımdaki sorunları çözmek için "Strict Two-phase Locking" ya da kısaca 2PL yöntemi sıklıkla kullanılır. Bu yöntemde bir transaction bir veritabanı nesnesini okuyacağı zaman bu nesne üzerine bir paylaşımı (shared) kilit koyar (S). Bu kiliti koyamayorsa o nesneyi okuyamaz. Eğer bir yazma işi yapılacaksa nesne üzerine exclusive lock (X) koyması gereklidir. Bu kiliti koyamazsa yazma yapamaz. Bir nesne üzerinde shared lock varsa diğer transactionlar da onun üzerine shared lock koyarak okuma yapabilir. Fakat bir nesne üzerinde (veri) exclusive lock varsa diğer transactionlar onun üzerine 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



T₁ : S(A), R(A), X(A), W(A), S(B), R(B), X(B), W(B), Abort

T₂ :

S(A), R(A), X(A), W(A), C

Bu kısımda T₂ çalışmaya başladı. Fakat A üzerinde X lock olduğunuandan T₂, A'nın üzerindeki anahtarın kalkmasını bekledi.

* Burada T₁'in abort edilmesi durumu artık sorun çıkarmayan çünkü T₂, T₁ bittiğinden sonra çalışıyor (Tutarlılık sağlandı ✓)

Örnek: RW Conflict with 2PL

T₁: S(A), R(A)

DBMS T₁ ile devam eder. Burada da T₁in
R(A) → A'yi güncellemesi için X-lock koyması gereklidir.
T₂ S lock koymadığundan burası yapmaz. (T₂ bekler)

T₂:

S(A), R(A)

→ ① X lock koyması gereklidir. Ama T₁'de A üzerinde
S lock olduğu için yapmaz. (T₁'i bekler)



Burada bir Deadlock olusur. (nâdiren olusur). Timeout metodu ile
bu yöntem uygulanır. Şikintili transactionlar abort edilir ve başka bir
üzerinde kullanılarak yeniden çalıştırılır.

Aborting a Transaction:

Bir transaction abort edileceği zaman o ana kadar yaptığı tüm
işlemlerin geri alınması gereklidir. Geri alınma işlemi yapıldıktan sonra, abort edilen
transactionun güncellendiği veri üzerindeki işlem yaparılan transactionlar da
geri alınmaktadır. Bunların 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 çökfüğü zaman hangi transactionlar
aktif (yani commit veya abort edilmemiş, sonlanmamış) ve buffer
pool'da 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 ölümden bir transaction başlatıyoruz. Transaction bitirmek için COMMIT veya ROLLBACK komutu kullanılır.

COMMIT: O ana kadar yapılan işlemler DB'ye yansıtılır.

ROLLBACK: save pointe kadar yapılan değişiklikleri geri alır

⋮
} Buradaki değişiklikler kaydedildi.
COMMIT

SAVEPOINT <savepoint-name>

⋮
} Buradaki değişiklikler geri alındı.

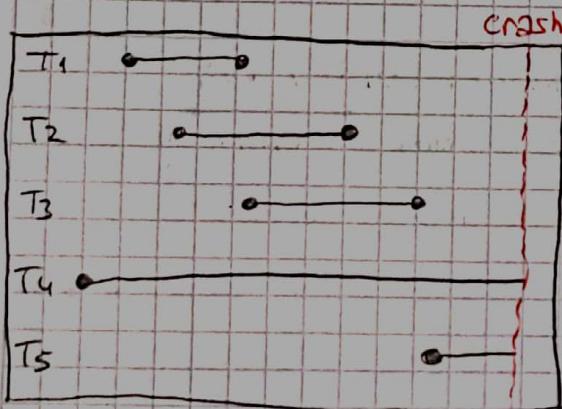
ROLLBACK TO SAVEPOINT <savepoint-name>

HAFTA XII. 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 $\text{yapt}(\varnothing)$ işleminin geri alınması gereklidir (abort). Bunu 2PL ile yapar.

Handling the Buffer Pool

- * Bir transaction commit edipse yapılan değişikliklerin diske anında yansıtılmasının Force denir. Fakat sürekli diske erişmek maliyetli bir istir. Bunun için transactionlar toplu şekilde diske kaydedilir. (no force)
- * Buffer pool doluysa 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 yapanı de Steal denir. Peki diske erken yazılan bu transaction abort ederse ne olacak? → (Toplu değişiklikler log ile geri alınır)

Logging

Ayrı bir disk üzerine ardışık olarak kayıtlar yazmaya loglama denir.

Bir log'lu sekildedir:

<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. (Atomiklığı garanti eder)

2-) DBMS, commit eden bir transaction'in o transaction'un tüm log

kayıtlarının yazılmış olması garanti edilmeli. (Durabilityyi garanti eder)

* Her log kaydının bir numarası vardır Buna Log Sequence Number (LSN)

denir. (id gibi). \rightarrow bir veri sayfasının da pageLSN değeri vardır. (Bu sayfaya

yapılan en son güncellemein log kaydının numarası). Sistem yazılmış olan

log kayıtlarının numaralarını flushed LSN ile tutar. (\rightarrow 0 ana kadar yazılmış

olan LSN değeri).

↳ Buna göre bir sayfa yazılmadan önce $\text{pageLSN} \leq \text{flushed LSN}$ olmalıdır.

* DBMS, periyodik olarak checkpoint oluşturur. Sistem yıkımı durumunda

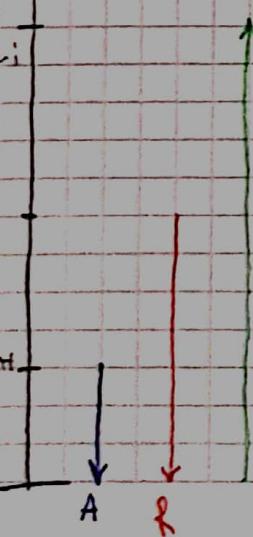
son checkpoint baz alınarak işlemler yapılır. (Daha önceki diske yazılmıştır)

Crash öncesi durum
(tüm transactionları geri
alındı.)

Aktifde
yazılıması

Son checkpoint

CRASH



Aktif olan transactionlar bulunur. • Analysis
Aktiflerin diskteki etkileri geri alınır. • Redo
Crash öncesi hale getirme. • Undo

↳ transaction işlemleri
geri alınır.

Örnek: WAL ile elde edilen log kayıtları şu şekilde olsun.

LSN	LOG
00	Checkpoint oluştur
05	Check point 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 gitilir (00). Bu checkpointten itibaren log kayıtlarına tek tek bakılır. Hangi transactionların aktif olduğunu tespit edilir (T_1, T_2, T_3 checkpointten sonra geldi). Buraların yaptığı işler REDO edilir (Veritabanına yansıtılır). Daha sonra (60) 'ta genelde gidilerek UNDO yapılır (T_2, P_5 ve P_3 'ü değiştirmisti bunlar geni silinir VS., bitmemis transactionlara UNDO yapılır!).

→ Kısaca: Crash olursa checkpointe git ve o işlemleri tekrar yap eger bitmemis transaction varsa bunların işlenmelerini disk'e yansıtma.
(Yani undo T_2 ve T_3 için yapılır)

CHAPTER 21:**-Security and Authorization -**

- Kullanıcılar sadık kendine izin verilen verileri görebilmeli. (Secrecy)
 - Verinin bütünlüğü sağlanmalıdır. (kullanıcı, izni olmayan bir şeyi değiştirememeli) (Integrity)
 - Verinin, izin verilen kişiler tarafından değiştirilebilmesi gereklidir. (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 tanımlıdır. 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.
- ★ Kullanıcılar 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 yapabılır.

GRANT DELETE ON Sailors TO Yuppy WITH GRANT OPTION

Yuppy isimli kullanıcı, Sailors tablosu üzerinde DELETE işlemi yapabilir ve bu işlemi yapma yetkisini diğer kullanıcılar da verebilir. Yuppy'nin yetkisi REVOKE ile geri alınırsa Yuppy'nin hak verdiği visilerin DELETE hakkını da elliinden alınıc.

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 Guppy, Yuppy

Guppy ve Yuppy 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 önemi olabilir. (Kullanıcılarla belli seyler göstermek). GRANT / REVOKE ile kullanırsız güclü bir veri erişim kontrolü sağlanır.

* Veri güvenliğini : Kullanıcılarla hizmet 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 tamsayısı 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 aralarında asal sayılar olacak
(bu işlemin sonucu S olsun)
 - Encryption fonksiyonu: $I^e \text{ mod } L$ → (Bu formül bize şifrelenmiş sayıyi verir)
 - Sifre çözme için bir d sayısı seçilir. Öyle ki bu d sayısı $d * e = 1 \text{ mod } ((p-1) * (q-1))$ olacak şekilde seçilmediğidir.
 - Decryption fonksiyonu: $S^d \text{ mod } L$ → (Şifrelenmiş sayıyı original haline döndürür)

- ★ SSL, SET gibi sertifikalar almak da bir güvenlik önlemdir.
-) Mandatory Access Control: Her veritabanı nesnesinin ve kullanıcıların bir güvenlik sınıfı vardır. Bu sınıflara göre kullanıcılar belirli verileri okuyup yazabılır.

* D kate nügetli lisisinden Horsie isimli bir tablo oluşturduğunu düşünelim. Justin isimli kullanıcıya Horsie tablosu üzerinde INSERT yetkisi versin. (Justin'in bundan haber yok). D, Justin'in kullanacağı kodları .tabloğa INSERT yapacağı şekilde güncelleme olsun. Justin'in yazdığı 'gizli' veriler Horsie tablosuna da kaydedilir. Böylece bu gizli bilgiler D'nin eline geçmeks olur.

(Discretionary Control Güvenlik Problemi - Truva Atı -)

Bell-La Padula Model

Örnek sayfadaki güvenlik zayıflığının önüne geçmeyi planlar.

- Nesneler = Tablolar, viewlar, tuple'lar
- Öznelер = Kullanıcılar, kullanıcı programları
- Güvenlik Sınıfları = Top Secret (TS), secret (S), confidential (C), unclassified (U)

$$\text{TS} > \text{S} > \text{C} > \text{U}$$

* Her nesnenin ve öznenin 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 yazabilir.

* Bu durumda truva atı sorunu söyle engelleniyor:

D kişi C sınıfında olsun. Bu durumda Hansie 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 Hansie tablosuna veri yazamaz.

* Tablolardaki her kayıt için güvenlik sınıfı verilebilir.

Örnek:

→ TS ve S sınıfı tüm satırları görebilir.
C sınıfı yalnızca 2. satırı görebilir
U sınıfı hiçbir satırı göremez.

bid	bname	color	class
101	Salsa	Red	S
102	Pinto	Brown	C

! Burada bir sorun var: Öğrenir bu tabloya? C sınıfında kullanıcı <101, Pasta, Blue, C> versini eklemek istesin. (Kullanıcı ilk satırı göremez). Eğer 101 id'li

kayıdı eklemek istersek PRIMARY KEY özelliğini kaybetmeye Ama eklemesek bunu kez kullanılcı kendisinden daha üst seviye bir güvenlik seviyesine sahip kaydın varlığınıından haberden olur. (bid ve class'ı PRIMARY KEY yaparsak sorun ortadan kalkar)

Statistical DB Security : (İstatistiksel)

Bu veritabanı modelinde veriler DB'de saklanır. Fakat DB'de sorgu yaparken kayıt bazında sorgu yapamayız. Genellikle bu yöntem, kişisel verileri korumak için kullanılır. (Hastane veritabanlarında).

Örneğin : Bir kişinin adını biliyorsak "bunun yaşı nedir?" diye bir sorgu yapamayız. Bunun yerine "DB'de kişiler ortalaması yaşı nedir?" gibi bir sorguya 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?" sorgusunu cevap 1 olana kadar gönderebiliriz. Böylece o kişinin yaşına erisiriz.

Bunun önüne geçmek için kullanıcının göndereceği sorgular üzerinde bir sınırlama yapılmıştır. Sorgu sayısı arttıkça DB'nin dönderdiği cevaplar üzerine gürültü eklenir (gittikçe artan gürültü)

HAFTA XIV SON

Ders işlenmedi

HAFTA XV. SON

Son

