YILDIZ TEKNİK ÜNİVERSİTESİ

Fiziksel Tasarım: Indexing, B+ Tree ve Hash

Öğr. Gör. Dr. Yasemin Topuz

Yıldız Teknik Üniversitesi

BLM3041 – Veritabanı Yönetimi

02 Ocak 2026 | YTÜ Bilgisayar Mühendisliği

Neler konuşacağız?

- Temel Kavramlar
- Sıralı İndeksler
  - Kümeleme İndeksi (Clustering Index)
  - Birincil İndeks (Primary Index)
  - İkincil İndeks (Secondary Index)
  - Çok Seviyeli İndeks (Multilevel Index)
  - B+ Ağaç İndeks Dosyaları
- Hashing

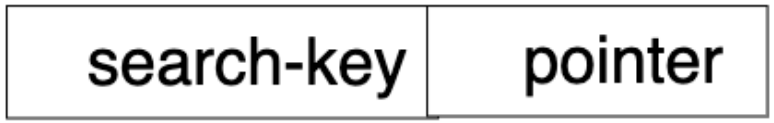
Ders Değerlendirme

Bu formu kullanarak dersin işlenişi, genel yapısı, dersin eğitmeni, laboratuvar süreçleri vb. hususlarda değerlendirmenizi iletmenizi rica ederim.

https://docs.google.com/forms/d/e/1FAIpQLSetwVaHkPTfMn8t_gcEXMHYIbMU0lYxsWE5vveqxlRyZ9Gefg/viewform

Temel Kavramlar (Basic Concepts)

- **İndeksleme (Indexing)**, istenen verilere daha hızlı erişebilmek için kullanılan mekanizmalardır.
  - **Örnek:** Bir kütüphanedeki yazar kataloğu, kitaplara hızlı ulaşmayı sağlar.
- **Arama Anahtarı (Search Key),** bir dosya içindeki kayıtları bulmak için kullanılan bir öznitelik ya da öznitelik kümesidir.
- **İndeks dosyası (Index File),** indeks girdileri (index entries) adı verilen kayıtlardan oluşur.
  Bu girdiler genel olarak şu yapıdadır:



  - Arama anahtarı, ilgili kaydı tanımlar.
  - İşaretçi (pointer), asıl kaydın bulunduğu konumu gösterir.
- İndeks dosyaları, genellikle asıl veri dosyasından çok daha küçüktür. Bu sayede arama işlemleri daha hızlı gerçekleştirilir.

İki temel indeks türü vardır:

- **Sıralı İndeksler (Ordered Indices):** Arama anahtarları sıralı (sorted) biçimde saklanır.
- **Hash İndeksler (Hash Indices):** Arama anahtarları, bir hash fonksiyonu kullanılarak bucket adı verilen yapılara dengeli şekilde dağıtılır.

İndeks Değerlendirme Ölçütleri (Index Evaluation Metrics)

Erişim Türleri: Değer tabanlı arama, aralık erişimi vb. gibi erişim türünü ifade eder.

- Belirli bir öznitelik değerine eşit olan kayıtların bulunması (ör. ogr_no = 12345)
- Öznitelik değeri belirli bir aralıkta olan kayıtların bulunması (ör. not BETWEEN 70 AND 90)

Erişim Süresi: Belirli bir veri öğesini veya öğe kümesini bulmak için gereken süreyi ifade eder.

- **Dosya tarama (File Scan):** yaklaşık 1 saniye, **İndeks tarama (Index Scan):** yaklaşık 10 milisaniye
- Tek seferde yalnızca 1 sorgu çalıştırılırsa, indeksin avantajı net olarak hissedilmeyebilir.
- 1000 öğrenci, her öğrenci saatte 10 sorgu gönderiyor. Toplam: 10.000 sorgu / saat
- **Dosya tarama kullanılırsa:** Toplam bekleme süresi: 1000 × 10 × 1 saniye = 10.000 saniye
- **İndeks tarama kullanılırsa:** Toplam bekleme süresi: 1000 × 10 × 10 ms = 100 saniye
- **Sonuç:** İndeksler, özellikle çok kullanıcılı ve yoğun sistemlerde büyük performans avantajı sağlar.

Ekleme Süresi (Insertion Time): Yeni bir kayıt eklenirken indeksin güncellenme maliyeti

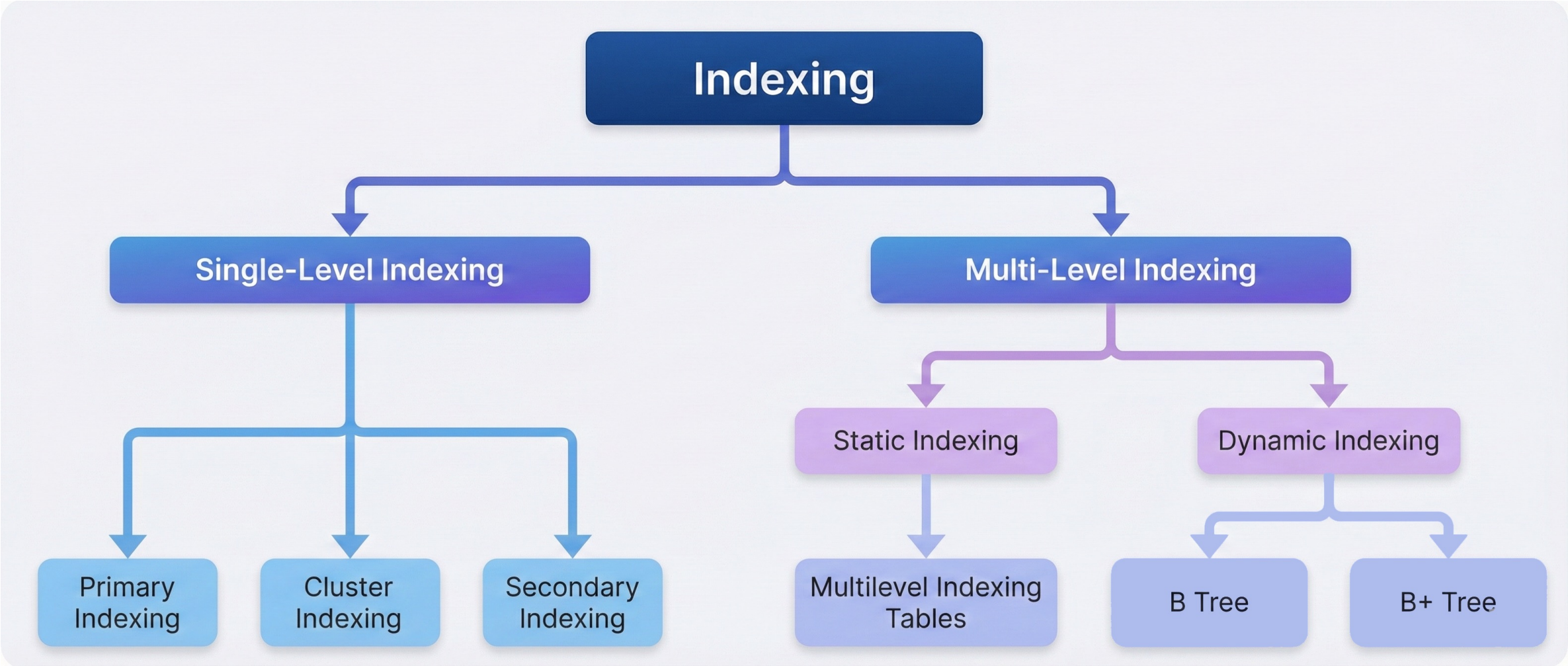
Silme Süresi (Deletion Time): Kayıt silinirken indeksin güncellenme maliyeti

Alan Yükü (Space Overhead): İndekslerin disk üzerinde kapladığı ek alan

Sıralı İndeksler (Ordered Indices)

- **Sıralı İndeks (Ordered Index):** İndeks girdileri, arama anahtarının değerine göre sıralı biçimde saklanır.
- **Kümeleme İndeksi (Clustering Index):** Sıralı (ardışık) olarak düzenlenmiş bir dosyada, dosyanın fiziksel sırasını belirleyen arama anahtarına sahip indekstir.
  - **Birincil indeks (Primary Index)** olarak da adlandırılır.
  - Birincil indeksin arama anahtarı, çoğu zaman birincil anahtar (primary key) olur; ancak zorunlu değildir.
- **İkincil İndeks (Secondary Index):** Dosyanın fiziksel (ardışık) sırasından farklı bir sırayı temel alan indekstir. Kümeleme yapmayan indeks (Nonclustering Index) olarak da adlandırılır.
- **İndeks-Sıralı Dosya (Index-Sequential File):** Dosyanın bir arama anahtarına göre sıralı olduğu, aynı arama anahtarı üzerinde bir kümeleme indeksi bulunduğu dosya yapısıdır.
  - Bu sunumda, B+ ağacı (B+ tree) yapısına geçilene kadar bu yaklaşım kullanılır.

İndeksleme


Indexing
Single-Level Indexing
Multi-Level Indexing
Primary Indexing
Cluster Indexing
Secondary Indexing
Static Indexing
Dynamic Indexing
Multilevel Indexing Tables
B Tree
B+ Tree

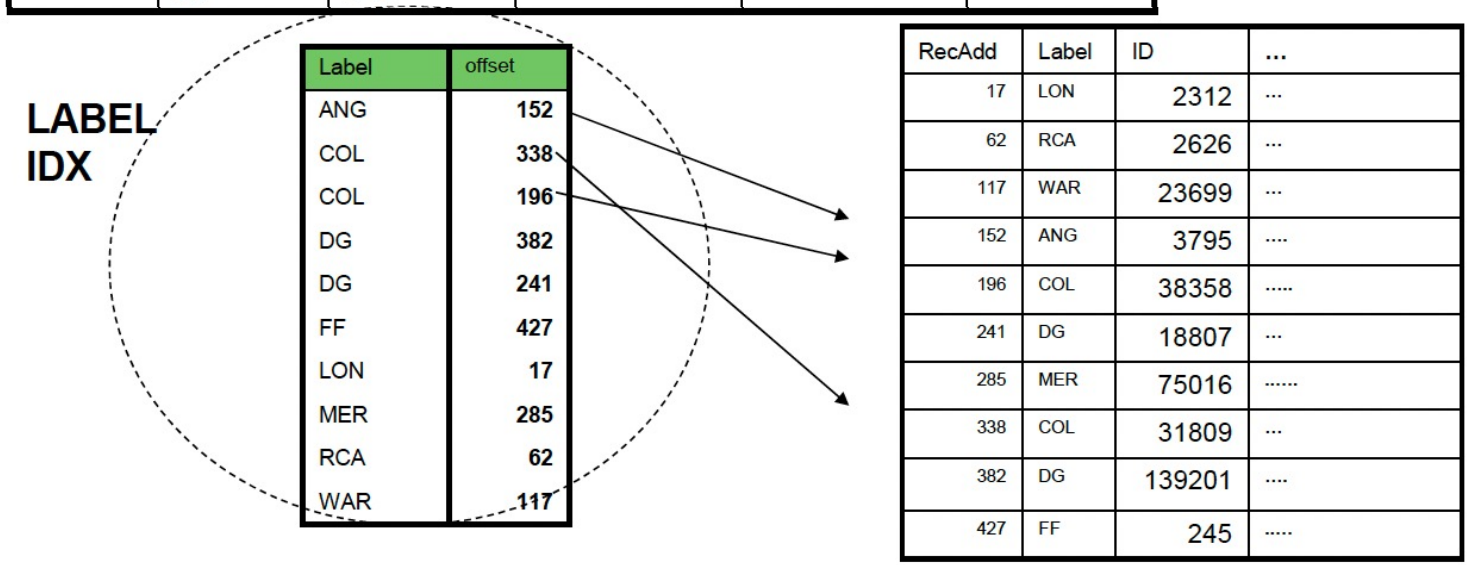
Primary Index

- Verilerin arama anahtarına göre sıralandığı ve veritabanı tablosunun birincil anahtarının indeks oluşturmak için kullanıldığı bir **kümelenmiş** indeksleme türüdür.
- Sıralı dosya organizasyonunu sağlayan varsayılan bir indeksleme biçimidir.
- Birincil anahtarlar benzersiz ve sıralı bir şekilde saklandığı için arama işleminin performansı oldukça verimlidir.

RecAdd	Label	ID	Title	Composer	Artist
17	LON	2312	Romeo-Juliet	Prokofiev	Maazel
62	RCA	2626	Quertet in C	Beethoven	Julliard
117	WAR	23699	Touchstone	Corea	Corea
152	ANG	3795	Symphony9	Beethoven	Giulini
196	COL	38358	Nebraska	Springstee	Springsteen
241	DG	18807	Symphony9	Beethoven	Karajan
285	MER	75016	Coq Suite	Rimsky	Leinsdorf
338	COL	31809	Symphony9	Dvorak	Berstein
382	DG	139201	Violin Con.	Beethoven	Ferras
427	FF	245	GoodNews	Sweet	Sweet

LABEL'ın benzersiz bir özellik olduğunu varsayalım. Belki de birincil anahtar.

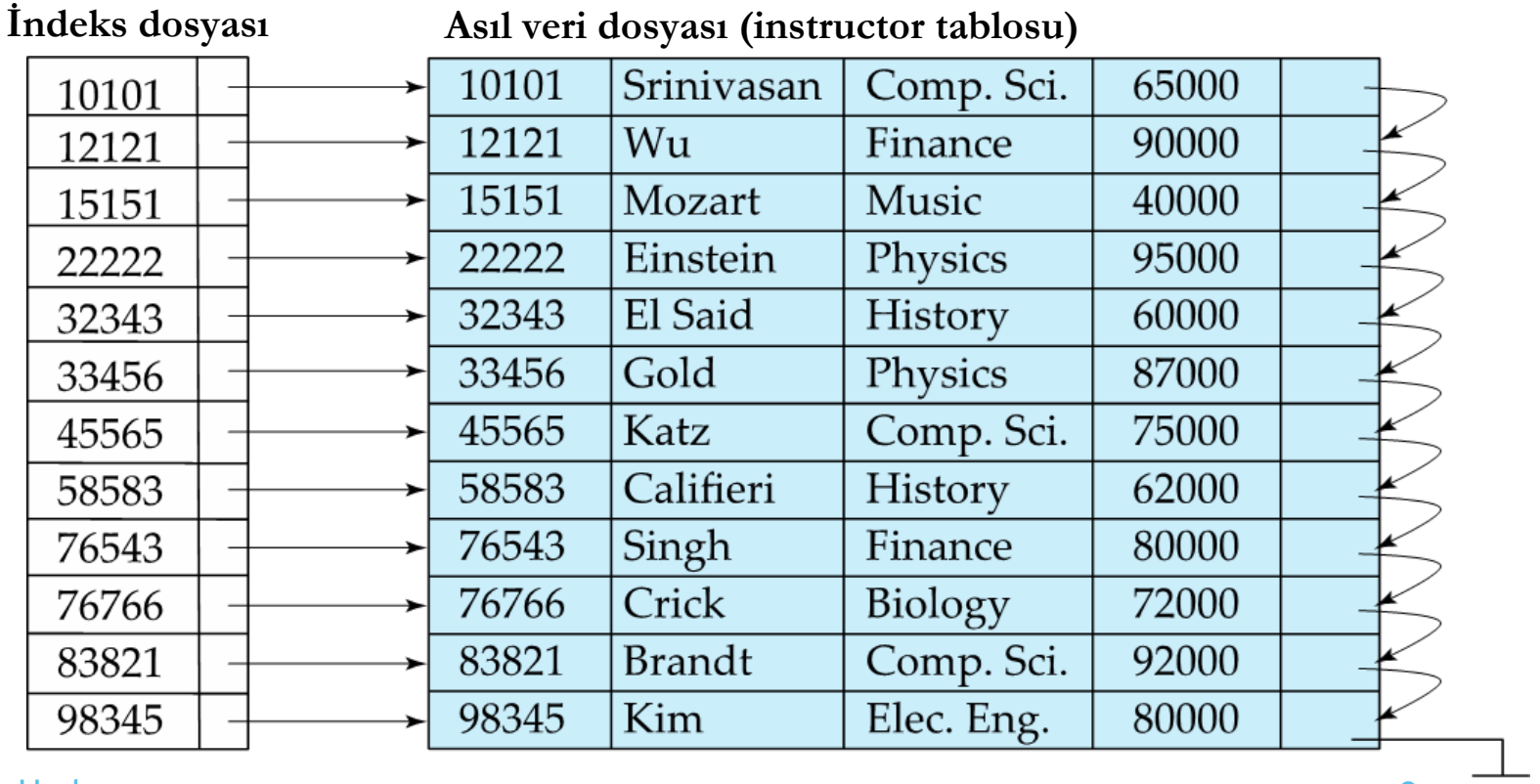
LABEL_IDX birincil idx olarak adlandırılır.



Yoğun İndeks Dosyaları (Dense Index Files)

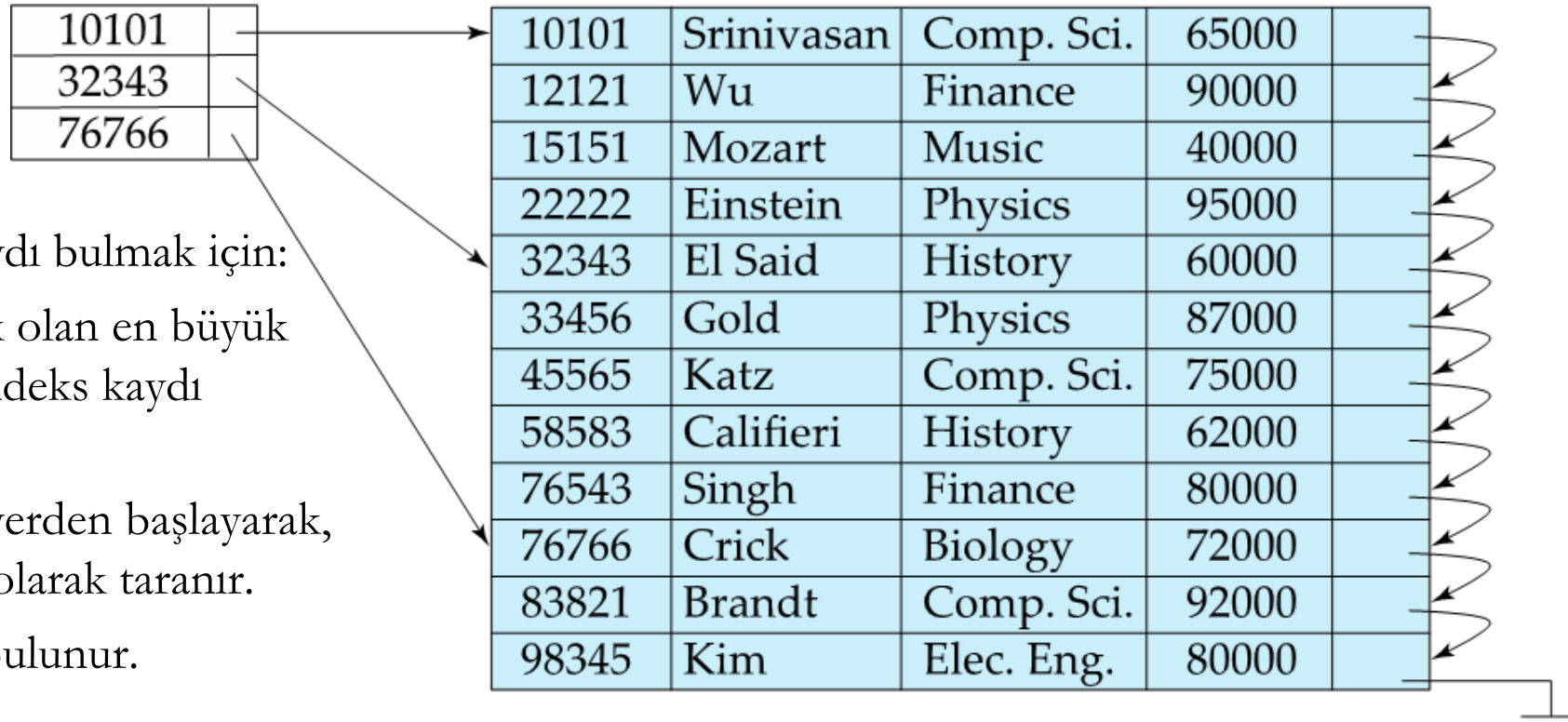
- **Yoğun indeks (Dense Index),** dosyada bulunan her bir benzersiz (unique) arama anahtarı değeri için ayrı bir indeks kaydı içeren indeks türüdür.
- Yani, tabloda yer alan her kayıt (ya da her benzersiz anahtar değeri) indeks dosyasında tek tek temsil edilir.

- Instructor ilişkisi (tablosu) üzerinde ID özniteliği için oluşturulan bir indeks düşünelim.
- Her ID değeri için indeks dosyasında bir kayıt bulunur.
- Her indeks kaydı, ilgili asıl kaydı (tuple) gösteren bir işaretçiye (pointer) sahiptir.



Seyrek İndeks Dosyaları (Sparse Index Files)

- **Seyrek indeks (Sparse Index),** dosyada bulunan tüm arama anahtarları için değil, yalnızca bazı arama anahtarı değerleri için indeks kaydı içerir.
- **Kullanım koşulu:** Kayıtlar arama anahtarına göre sıralı (sequentially ordered) ise uygulanabilir.



Bir Kaydın Bulunma Adımları

Arama anahtarı değeri K olan bir kaydı bulmak için:

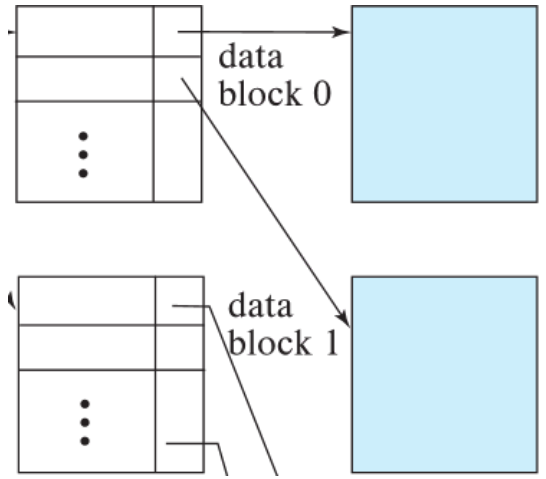
1) İndeks dosyasında, K'dan küçük olan en büyük arama anahtarı değerine sahip indeks kaydı bulunur.
2) Bu indeks kaydının işaret ettiği yerden başlayarak, veri dosyası ardışık (sequential) olarak taranır.
3) Aranan K değerine sahip kayıt bulunur.

Seyrek İndeks Dosyaları (Sparse Index Files)

- **Yoğun indekslerle karşılaştırıldığında**: Ekleme ve silme işlemleri için daha az yer kaplar ve daha az bakım maliyeti gerektirir. Ama kayıtları bulmak için yoğun indekslerden (genellikle) daha yavaştır.

İyi Bir Denge (Good Tradeoff)

- Kümelemeli indeks (Clustered index) için: Dosyadaki her veri bloğu için bir indeks kaydı içeren seyrek indeks kullanılır. Her indeks kaydı, ilgili bloktaki en küçük arama anahtarı değerine karşılık gelir.
- Bu yaklaşım, alan kullanımını azaltır, arama performansını kabul edilebilir düzeyde tutar



Kümeleme Olmayan İndeks (Unclustered Index) Durumu

- Seyrek indeks, yoğun indeksin üzerine kurulur.
- Bu yapı çok seviyeli indeks (multilevel index) olarak adlandırılır.

İkincil İndeks Örneği (Secondary Indices Example)

- Arama anahtarı, benzersiz (veya birincil) anahtar olmak zorunda değildir.
- Bu tür indekslere **İkincil ya da kümelenmemiş** indeks denir. Veri kayıtlarına birden fazla görünüm sağlar.

- Kullanıcı genellikle birincil anahtara göre bir kayıt aramaz (Çünkü birincil anahtar biraz gizli bir kavramdır).

- Örneğin, kullanıcı besteciye göre bir kayda erişmek istiyor (örneğin, Beethoven kaydı).

- **Sorgu:** Beethoven'ın 9. Senfonisinin tüm kayıtlarını bulun.

- İndeks olmadan, bu sorgu tüm tabloyu taramayı gerektirir.

Non-Primary KEY: Composer	*reference*
Beethoven	62
Beethoven	**152**
Beethoven	**241**
Beethoven	382
Corea	117
Dvorak	338
Prokofiev	17
Rimsky	285
Springstee	196
Sweet	427

Non-Primary KEY: Title	*reference*
Coq Suite	285
GoodNews	427
Nebraska	196
Romeo-Juliet	17
Quertet in C	62
Symphony9	**152**
Symphony9	**241**
Symphony9	338
Touchstone	117
Violin Con.	382

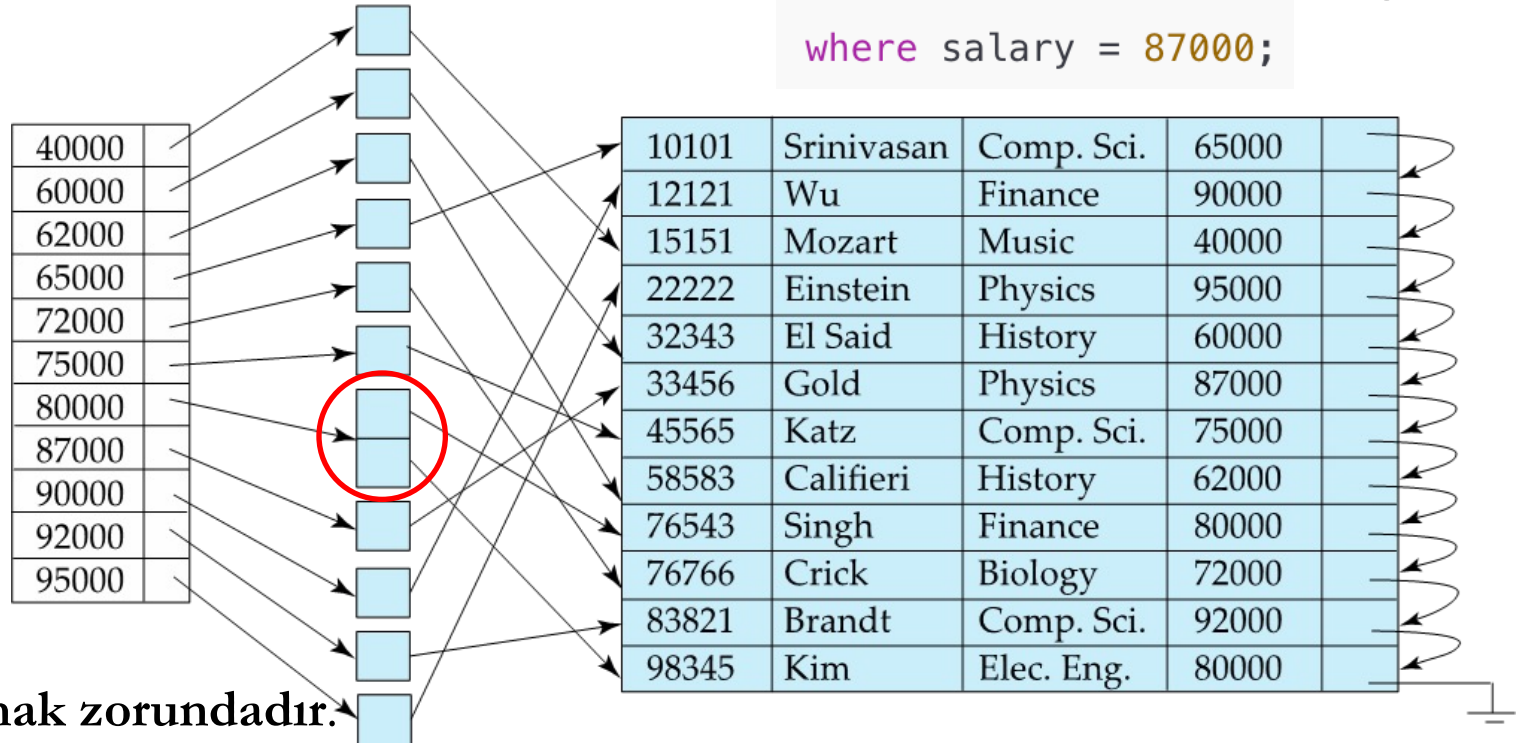
İkincil İndeks Örneği (Secondary Indices Example)

```
select *
from instructor
where salary = 87000;
```

- Instructor tablosunun salary (maaş) alanı üzerinde oluşturulmuş bir ikincil i örneği gösterilmektedir.

Yapının Çalışma Mantığı

- Her maaş değeri, bir arama anahtarı (search key) olarak yer alır.
- Her arama anahtarı, bir bucket (kova) yapısını işaret eder.
- Bucket, aynı maaş değerine sahip tüm gerçek kayıtların işaretçilerini (pointer) içerir.



İkincil indeksler mutlaka **yoğun** (dense) **olmak zorundadır.**

- Dosya, bu özniteliğe göre sıralı değildir. Her arama anahtarı değeri için indeks kaydı bulunmalıdır.

- Aynı arama anahtarına sahip birden fazla kayıt olabileceğinden, indeks kaydı doğrudan tek bir kaydı değil, bir bucket'ı işaret eder.

Kümelemeli ve Kümeleme Olmayan İndeksler

- İndeksler, kayıtları arama sırasında önemli performans avantajları sağlar.

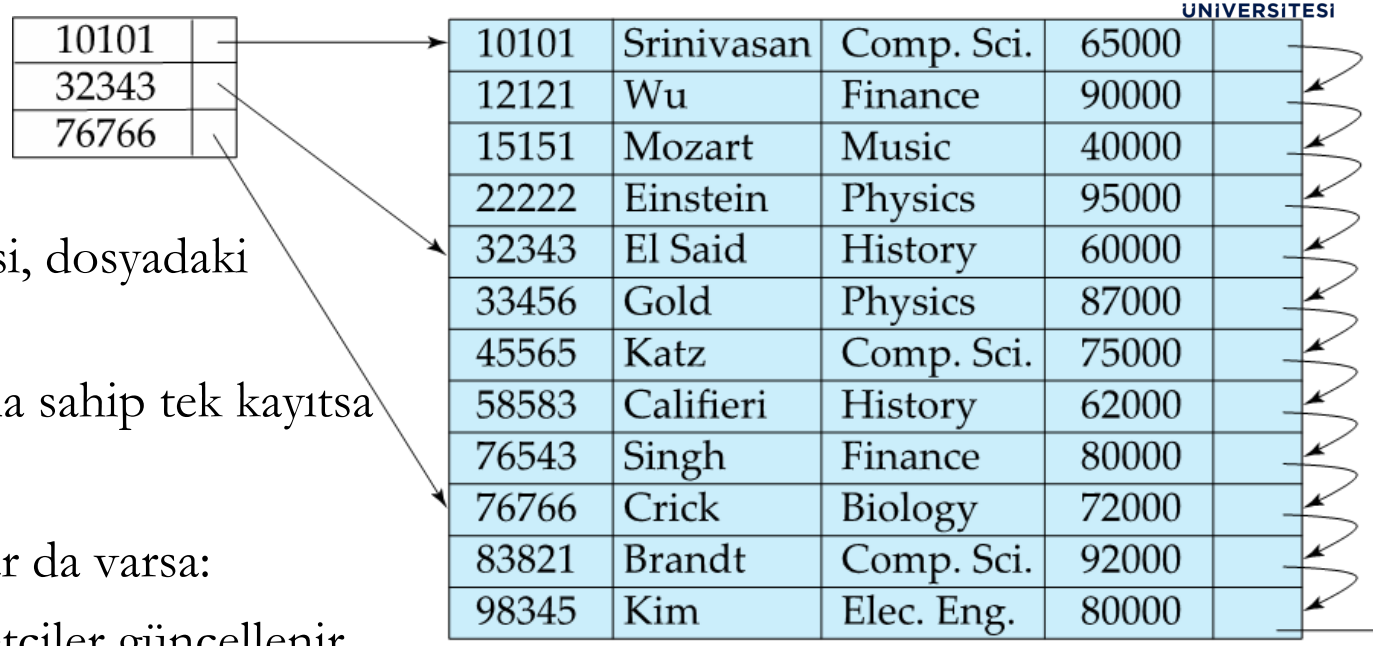
İndekslerin Getirdiği Ek Yük (Overhead)

- Ancak, indeksler veritabanı üzerinde ek maliyet oluşturur:
  - Bir kayıt eklendiğinde veya silindiğinde, tablo üzerindeki tüm indeksler güncellenmelidir.
  - Bir kayıt güncellendiğinde, güncellenen öznitelik üzerinde tanımlı indeks(ler) de güncellenir.

Ardışık Tarama (Sequential Scan) Karşılaştırması (BETWEEN)

- Kümelemeli indeks (Clustering index) kullanılarak yapılan ardışık tarama:
  - Verimlidir, çünkü kayıtlar disk üzerinde birbirine yakın bloklarda yer alır.
- İkincil (kümeleme olmayan / nonclustering) indeks kullanılarak yapılan ardışık tarama:
  - Maliyetlidir, özellikle manyetik disklerde.
    - Her kayıt erişimi, diskte farklı bir blok okunmasını gerektirebilir.
    - Manyetik disklerde bir blok okuma süresi yaklaşık 5–10 milisaniyedir.

Index Update: Deletion



Tek Seviyeli İndeks Girdisi Silme

1) Yoğun İndeksler (Dense Indices)

- Yoğun indekslerde, arama anahtarının silinmesi, dosyadaki kayıt silmeye benzer şekilde yapılır.
- Eğer silinen kayıt, dosyada bu arama anahtarına sahip tek kayıtsa
  → İlgili arama anahtarı indeksten de silinir.
- Eğer aynı arama anahtarına sahip başka kayıtlar da varsa:
  → İndeksteki anahtar korunur, yalnızca işaretçiler güncellenir.

2) Seyrek İndeksler (Sparse Indices)

- Seyrek indekslerde silme işlemi daha dikkatli yapılır.
- Eğer silinen kayıt için indekste bir giriş (entry) varsa, bu giriş dosyada (arama anahtarı sırasına göre) bir sonraki arama anahtarı değeri ile değiştirilir.

Özel durum: Eğer bir sonraki arama anahtarı değeri zaten indekste mevcutsa, mevcut indeks girişi doğrudan silinir, yerine başka bir anahtar yazılmaz.

Index Update: Insertion

Tek Seviyeli İndeks Ekleme

- Öncelikle, eklenecek kaydın arama anahtarı (search key) değeri kullanılarak indekste arama (lookup) yapılır.

1) Yoğun İndeksler (Dense Indices)

- Eğer arama anahtarı değeri indekste yoksa, yeni bir indeks girişi eklenir.
- İndeksler ardışık (sequential) dosyalar olarak tutulur.
- Yeni indeks girdisi için yer açılması gerekir. Gerekirse taşma (overflow) blokları oluşturulabilir.

2) Seyrek İndeksler (Sparse Indices)

- Eğer indeks, dosyadaki her blok için bir indeks girdisi tutuyorsa, yeni bir veri bloğu oluşmadıkça, indekste herhangi bir değişiklik yapılmaz.
- Yeni bir blok oluşturulursa, bu yeni blokta yer alan ilk arama anahtarı değeri, indekse eklenir.

Çok Seviyeli İndekslerde Ekleme ve Silme

- Çok seviyeli (multilevel) indekslerde: Ekleme ve silme algoritmaları, tek seviyeli indeks algoritmalarının doğal ve basit bir genişletilmiş hâlidir.

Birden Fazla Anahtar Üzerinde İndeksler

Bileşik Arama Anahtarı (Composite Search Key): Birden fazla özniteliğin birlikte kullanılmasıyla oluşturulan arama anahtarıdır.

- Örnek: instructor tablosu üzerinde (name, ID) özniteliklerini içeren bir indeks.

Sıralama Kuralı (Lexicographical Order)

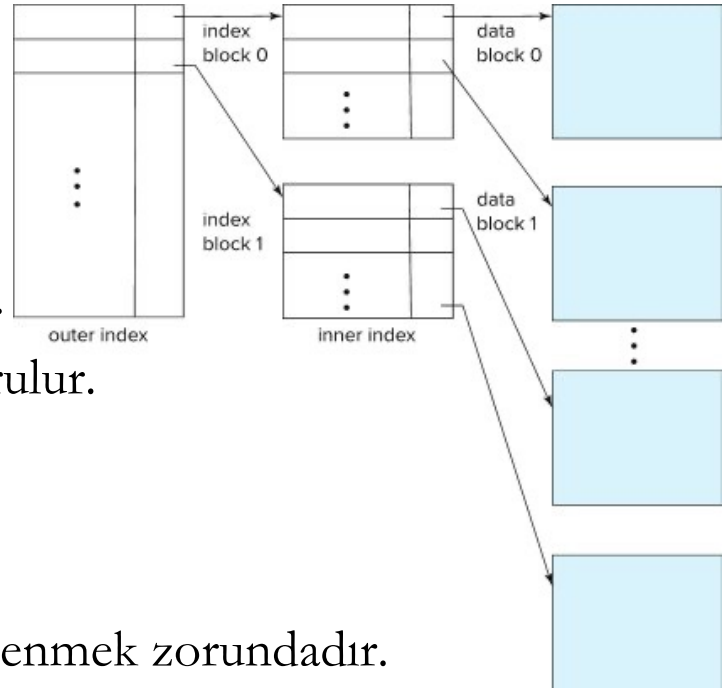
- Bileşik anahtar değerleri sözlük (lexicographical) sırasına göre sıralanır.
- Önce ilk öznitelik, eşitlik varsa ikinci öznitelik dikkate alınır.
  - (John, 12121) < (John, 13514)
  - (John, 13514) < (Peter, 11223)

Sorgulama Yeteneği

- Bu tür bir indeks kullanılarak,
  - Sadece name özniteliğine göre sorgu yapılabilir.
  - Ya da birlikte (name, ID) özniteliklerine göre sorgu yapılabilir.

Çok Seviyeli İndeks (Multilevel Index)

- Eğer bir indeks belleğe (main memory) sığmıyorsa, indeks erişimi maliyetli ve yavaş hâle gelir.
- **Çözüm:** Diskte tutulan indeksi ardışık (sequential) bir dosya gibi ele alıp, bu indeksin üzerine seyrek bir indeks oluşturmak.



İndeks Seviyeleri

- İç indeks (Inner Index): Asıl / temel indeks dosyasıdır.
- Dış indeks (Outer Index): İç indeksin üzerinde oluşturulmuş seyrek indekstir.
- Eğer dış indeks bile ana belleğe sığmıyorsa: Bir üst seviye indeks daha oluşturulur.
  Bu işlem, indeks belleğe sığana kadar katmanlı olarak devam eder.

Güncelleme Maliyeti

- Ekleme (insert) veya silme (delete) işlemlerinde, tüm indeks seviyeleri güncellenmek zorundadır.
- Bu durum, çok seviyeli indekslerde ek bakım maliyeti oluşturur.

B+ Ağaç İndeks Dosyaları (B+ Tree Index Files)

İndeks-Sıralı Dosyaların (Indexed-Sequential Files) Dezavantajları

- Dosya büyüdükçe, çok sayıda taşma (overflow) bloğu oluşur.
- Bu durum, erişim performansının zamanla düşmesine neden olur.
- Performansı koruyabilmek için, tüm dosyanın periyodik olarak yeniden düzenlenmesi gerekir.

B+ Ağaç İndekslerinin Avantajları

- Ekleme ve silme işlemleri sırasında, küçük ve yerel (local) değişikliklerle otomatik olarak kendini yeniden düzenler.
- Performansı korumak için tüm dosyanın baştan sona yeniden düzenlenmesine gerek yoktur.

B+ Ağaçların Dezavantajları

- Ekleme ve silme işlemlerinde, ek işlem maliyeti (overhead) oluşur. Ek disk alanı (space overhead) gerektirir.

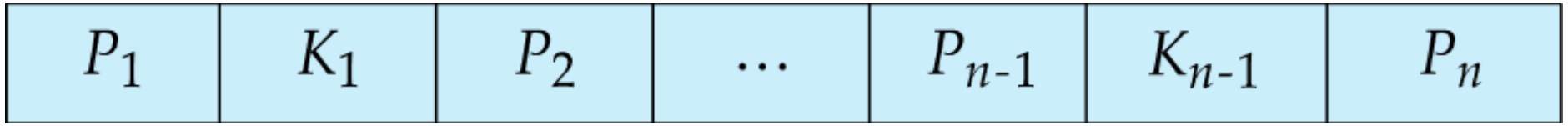
Genel Değerlendirme

- B+ ağaçların avantajları, dezavantajlarına göre çok daha ağır basar.
- Bu nedenle, B+ ağaçlar veritabanı sistemlerinde yaygın olarak kullanılmaktadır.

B+ Ağaç Düğüm Yapısı (B+ Tree Node Structure)

Tipik Bir Düğüm (Typical Node)

- Bir B+ ağaç düğümü genel olarak aşağıdaki bileşenlerden oluşur:



Düğüm Bileşenleri

- K_i (Anahtar Değerler): Düğüm içinde saklanan arama anahtarı (search-key) değerleridir.
- P_i (İşaretçiler / Pointers):
  - Yaprak olmayan (internal) düğümlerde alt düğümlere (children) işaret eder.
  - Yaprak (leaf) düğümlerde, gerçek kayıtlara veya aynı anahtar değerine sahip kayıtların tutulduğu bucket yapısına işaret eder.

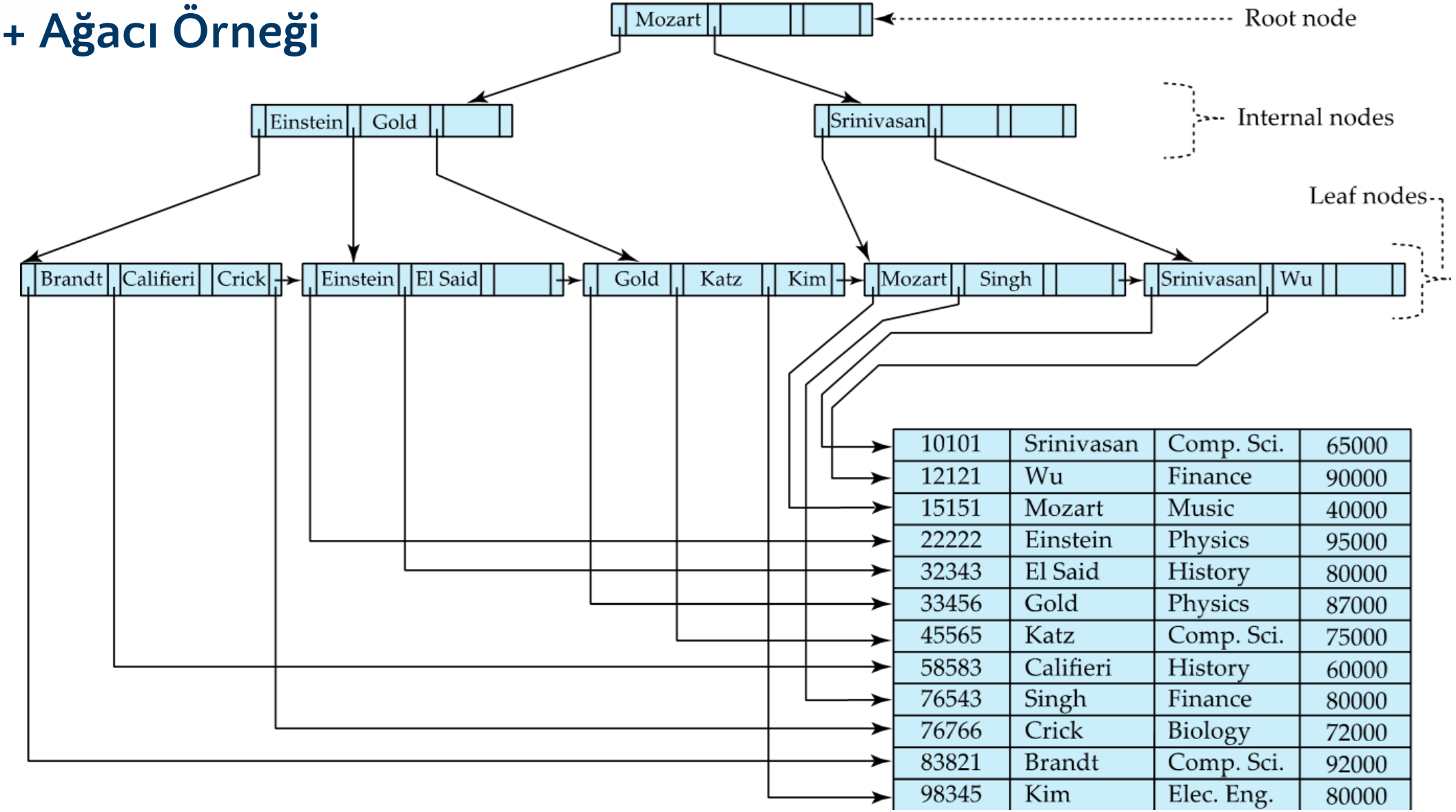
Bir düğüm içindeki arama anahtarları sıralıdır:

$$K_1 < K_2 < K_3 < \cdots < K_{n-1}$$

Bu sıralama, arama işlemlerinin hızlı ve deterministik yapılmasını sağlar.

(Başlangıçta yinelenen anahtar olmadığını varsayılır.)

B+ Ağacı Örneği



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

B+ Ağaç İndeks Dosyaları (B+ Tree Index Files)

- Bir B+ ağacı, aşağıdaki özellikleri sağlayan köklenmiş (rooted) bir ağaçtır:
  - Kökten (root) yapraklara (leaf) giden tüm yolların uzunluğu aynıdır. Ağaç dengelidir (balanced).
  - Kök ve yaprak olmayan her düğümün; En az $\lceil n/2 \rceil$, en fazla n adet çocuğu (child) vardır.
  - Bir yaprak düğüm; En az $\lceil (n-1)/2 \rceil$, en fazla $(n-1)$ adet anahtar değeri içerir.

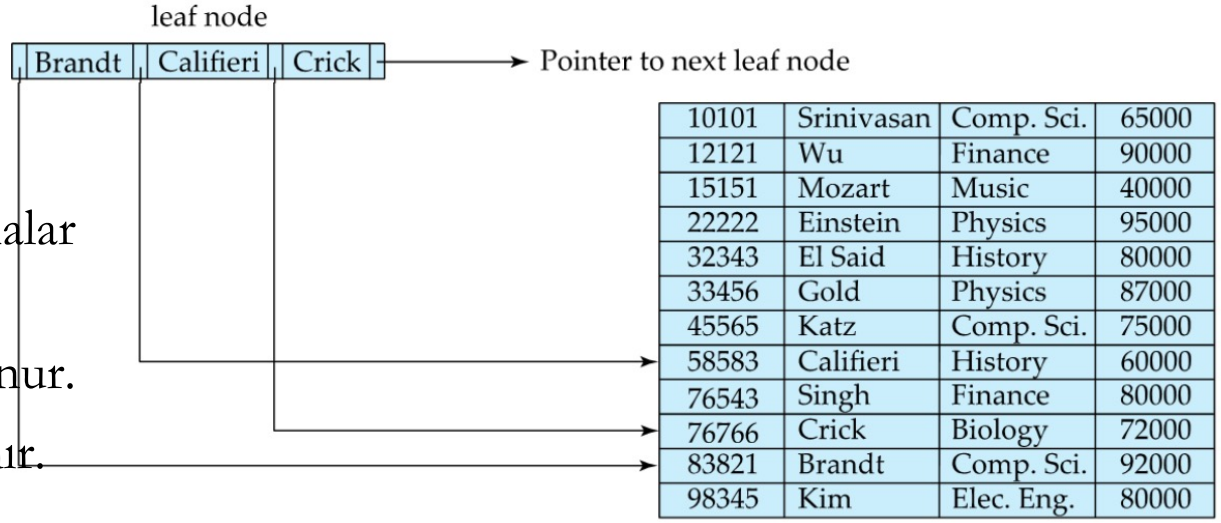
Özel Durumlar (Special Cases)

- Eğer kök düğüm bir yaprak değilse; en az 2 çocuğa sahip olmak zorundadır.
- Eğer kök düğüm aynı zamanda yaprak düğümse (yani ağaçta başka düğüm yoksa): 0 ile $(n-1)$ arasında anahtar değeri içerebilir.
- Tüm yaprakların aynı seviyede olması sayesinde; arama, ekleme ve silme işlemleri öngörülebilir ve hızlıdır.
- Düğümlerdeki alt–üst sınırlar; ağacın çok fazla büyümesini veya dengesizleşmesini engeller.
- Bu yapı; disk tabanlı indeksleme için son derece uygundur.

B+ Ağaçlarda Yaprak Düğümler (Leaf Nodes in B+ Trees)

Bir Yaprak Düğümün Özellikleri

- $i = 1, 2, \ldots, n-1$ için, P_i işaretçisi, arama anahtarı değeri K_i olan gerçek bir dosya kaydına (veya bu anahtara ait kayıtları içeren bir yapıya) işaret eder.
- Yaprak düğümler arama anahtarlarına göre sıralıdır:
  - L_i ve L_j yaprak düğümler ise ve $i < j$ ise, L_i'nin arama anahtar değerleri L_j'nin arama anahtar değerlerinden küçük veya eşittir.
- P_n işaretçisi, arama anahtarı sırasına göre bir sonraki yaprak düğümü gösterir.
- Yaprak düğümlerin birbirine bağlı (linked) olması sayesinde, aralık sorguları (range queries), sıralı taramalar (sequential scan) çok verimli şekilde yapılabilir.
- Tüm gerçek kayıtlar, sadece yaprak düğümlerde bulunur.
- İç düğümler yalnızca yönlendirme (indexing) amaçlıdır.



B+ Ağaçlarındaki Yaprak Olmayan Düğümler

- Yaprak olmayan (internal / non-leaf) düğümler, yaprak düğümler üzerinde kurulmuş çok seviyeli bir seyrek indeks yapısını oluşturur.
- Bir yaprak olmayan düğümde m adet işaretçi (pointer) bulunduğunu varsayalım:
  - P_1'in işaret ettiği alt ağaçtaki tüm arama anahtarları: K_1'den küçüktür.
  - $2 \leq i \leq n-1$ için: P_i'nin işaret ettiği alt ağaçtaki tüm arama anahtarları: K_{i-1}'den büyük veya eşit, K_i'den küçüktür.
  - P_n'in işaret ettiği alt ağaçtaki tüm arama anahtarları: K_{n-1}'den büyük veya eşittir.

Hashing

Statik Hashing (Static Hashing)

- Bucket (kova), bir veya daha fazla girdi içeren bir depolama birimidir (bir kova tipik olarak bir disk bloğudur).
- Bir girdinin hangi bucket'ta tutulacağı, girdinin arama anahtarı (search-key) değeri üzerinden bir hash fonksiyonu kullanarak belirlenir.
- **Hash fonksiyonu (h):** Arama anahtarları kümesi K'dan, bucket adresleri kümesi B'ye eşleme yapan bir fonksiyondur.

$$h: K \rightarrow B$$

- Hash fonksiyonu, arama (access), ekleme (insertion), silme (deletion) işlemlerinin tamamında kullanılır.

- Farklı arama anahtarı değerleri, aynı bucket'a eşlenebilir. Bu durumda, ilgili bucket içindeki tüm girdiler sırayla taranır. Bu durum çakışma (collision) olarak adlandırılır.
- **Hash index:** Bucket'lar, kayıtlara işaret eden indeks girdilerini (pointer) tutar.
- **Hash file-organization:** Bucket'lar, gerçek kayıtların kendisini doğrudan tutar.

Bucket Taşmalarının Yönetimi (Handling of Bucket Overflows)

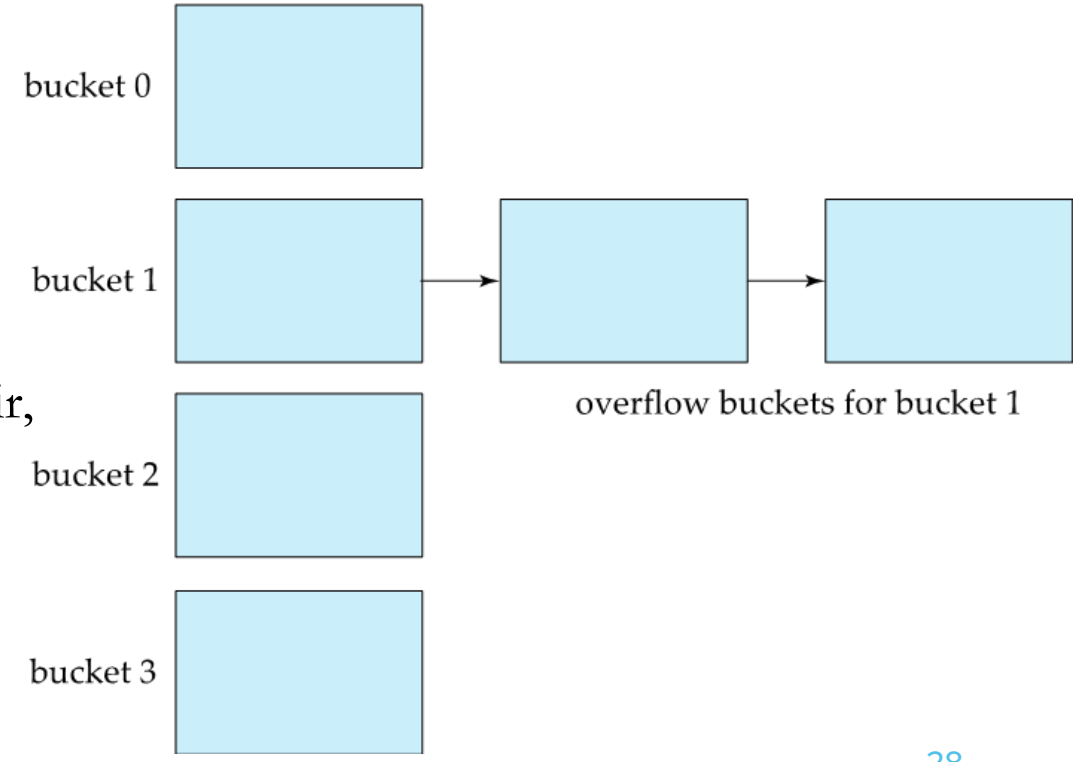
- Bucket taşması (overflow) aşağıdaki nedenlerle ortaya çıkabilir:
  - Yetersiz sayıda bucket bulunması: Başlangıçta ayrılan bucket sayısı, veri miktarı için yetersiz olabilir.
  - Kayıtların dağılımında dengesizlik (skew): Bu durum iki temel nedenden kaynaklanabilir
    - ✓ Birden fazla kaydın aynı arama anahtarı değerine sahip olması
    - ✓ Seçilen hash fonksiyonunun, anahtarları eşit (uniform) şekilde dağıtamaması

Taşmaların Yönetimi

- Bucket taşmasının oluşma olasılığı azaltılabilir, ancak tamamen ortadan kaldırılamaz.
- Bu nedenle taşmalar, taşma bucket'ları (overflow buckets) kullanılarak yönetilir.
- Ana bucket dolduğunda yeni kayıtlar, ona bağlı taşma bucket'larına yönlendirilir.

Bucket Taşmalarının Yönetimi (Handling of Bucket Overflows)

- Taşma zincirleme (overflow chaining) yönteminde bir bucket dolduğunda, o bucket'a ait taşma bucket'ları, bağlı liste (linked list) şeklinde birbirine bağlanır.
- Bu yaklaşım genellikle **kapalı adresleme (closed addressing)** olarak adlandırılır.
  - Bazı kaynaklarda Closed hashing veya Open hashing terimleri de kullanılabilir.
- Açık adresleme (open addressing) yöntemi taşma bucket'ları kullanmaz. Çakışan kayıtlar, farklı boş bucket'lara yerleştirilir.
- Ancak bu yöntem veritabanı uygulamaları için uygun değildir, çünkü disk tabanlı sistemlerde çok sayıda rastgele erişim performans kaybına yol açar.



Hash Dosya Organizasyonu Örneği

- Bu örnekte, instructor dosyası için hash dosya organizasyonu gösterilmektedir.
- Anahtar (key) olarak dept_name (bölüm adı) kullanılmaktadır.
- Bir karakterin ikili (binary) gösteriminin, o karakterin sıra numarasına (i) karşılık geldiği varsayılmaktadır.

Hash fonksiyonu:

Anahtarı oluşturan karakterlerin ikili gösterimlerinin toplamını alır, sonucu 8'e göre mod alarak bucket numarasını üretir.

- $h(\text{key}) = \left(\sum \text{karakterlerin ikili değerleri}\right) \bmod 8$

Örnek Hesaplamalar

- M=13, U=21, S=19, I=9, C=3 Toplam = 13+21+19+9+3 = 65 ≡ 1 (mod 8)
- h(Music) = 1, h(History) = 2, h(Physics) = 3, h(Elec. Eng.) = 3
- Farklı anahtarlar aynı bucket'a eşlenebilir.
  Bu durum çakışma (collision) oluşturur ve gerekirse taşma bucket'ları kullanılır.

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Statik Hashing'in Eksiklikleri

- Statik hashing'de, hash fonksiyonu (h), arama anahtarlarını sabit (fixed) sayıda B bucket adresine eşler.
- Ancak: Veritabanları zamanla büyür veya küçülür.

Ortaya Çıkan Problemler

- Başlangıçta bucket sayısı az seçilirse dosya büyüdükçe çok sayıda overflow bucket oluşur, performans ciddi şekilde düşer.
- İleride büyüme olur diye fazla alan ayrılırsa, çok sayıda bucket boş veya yarı dolu (underfull) kalabilir, disk alanı boşa harcanır. Veritabanı küçülürse, ayrılmış alan yine boşa gider.

Olası Çözüm

- Dosyanın, yeni bir hash fonksiyonu ile periyodik olarak yeniden organize edilmesi
- Bu çözüm pahalıdır. Normal sistem çalışmasını kesintiye uğratır.

Daha İyi Çözüm

- Bucket sayısının dinamik olarak değiştirilebilmesi yaklaşımı
- Dinamik hashing yöntemleri (örn. Extendible Hashing, Linear Hashing)

Dinamik Hashing

YTÜ YILDIZ TEKNİK ÜNİVERSİTESİ

Periyodik Rehashing

- Eğer hash tablosundaki kayıt sayısı tablo boyutunun (örneğin) 1.5 katına çıkarsa:
  - Yeni, daha büyük bir hash tablosu oluşturulur (ör. 2 katı), tüm kayıtlar yeniden hashlenir.
- **Sorun:** Bu işlem tek seferde çok maliyetlidir.

Linear Hashing

- Rehashing işlemi kademeli (incremental) olarak yapılır.
- Tüm tabloyu bir anda yeniden düzenlemek yerine Bucket'lar adım adım bölünür.
- Büyük duraklamalar yoktur ve veritabanları için uygundur.

Extendible Hashing

- Özellikle disk tabanlı hashing için tasarlanmıştır. Bucket'lar birden fazla hash değeri tarafından paylaşılabilir.
- Bucket sayısını iki katına çıkarmadan hash tablosundaki giriş sayısı iki katına çıkabilir.

Sıralı İndeksleme (Ordered Indexing) ve Hashing Karşılaştırması

- Periyodik yeniden düzenleme (re-organization) maliyeti
  - Bazı indeksleme yöntemleri, performansı koruyabilmek için dosyanın zaman zaman yeniden organize edilmesini gerektirir. Bu işlem maliyetli olabilir ve sistem performansını geçici olarak düşürebilir.
- Eğer sistemde çok sık insert ve delete işlemi yapılıyorsa indekslerin güncellenme maliyeti önemli bir faktör haline gelir. **Hashleme** genellikle bu tür işlemlerde daha hızlıdır, ancak yan etkileri vardır.
- Eğer ortalama erişim süresini optimize etmek, en kötü durum süresini feda etmek kabul edilebilirse, **Hashleme** iyi bir tercihtir.
- Finansal işlemler gibi gerçek zamanlı (real-time) sistemlerde, nadir bile olsa kötü en kötü durum performansı kabul edilemez.
- Belirli bir anahtar değerine göre arama (eşitlik sorguları), hashleme genellikle daha etkilidir.
- Aralık sorguları (range queries) yaygınsa, sıralı indeksler (ordered indices) tercih edilmelidir.
- PostgreSQL, hash indekslerini destekler. PostgreSQL 10'dan itibaren kullanımı teşvik edilmektedir.
- Oracle, Statik hash dosya organizasyonunu destekler. Ancak hash index desteği yoktur.
- SQL Server, Sadece B+ ağaç (B+-tree) tabanlı indeksleri destekler.

Çoklu Anahtar (Multiple-Key) Erişim

- Bazı sorgu türlerinde birden fazla indeks birlikte kullanılarak daha verimli erişim sağlanabilir.

```
SELECT ID FROM instructor WHERE dept_name = 'Finance' AND salary = 80000;
```

- Bu sorguda iki farklı öznitelik üzerinde koşul vardır: *dept_name ve salary*

Tekil İndeksler Kullanılarak Olası Stratejiler

- Önce dept_name = 'Finance' olanlar bulunur, ardından bu kayıtlar üzerinde salary = 80000 koşulu test edilir.
- Önce salary = 80000 olanlar bulunur, ardından bu kayıtlar üzerinde dept_name = 'Finance' koşulu test edilir.
- dept_name indeksi ile Finance bölümüne ait tüm kayıtların adresleri bulunur. Salary indeksi ile maaşı 80000 olan kayıtların adresleri bulunur. Bu iki adres kümesinin kesişimi (Index Intersection) alınır.

Diğer Yaklaşımlar

- **Bileşik indeks (Composite Index):** (dept_name, salary) özniteliklerini birlikte içeren tek bir indeks kullanılır.
- **Bitmap indeks:** Eğer her iki indeksin seçiciliği (selectivity) düşükse tercih edilebilir.
- **Sıralı tarama (Sequential Scan):** İndeksler mevcut olsa bile, bazı durumlarda tüm tabloyu taramak daha verimli olabilir.

Çoklu Anahtarlar Üzerinde İndeksler (Indices on Multiple Keys)

- **Bileşik arama anahtarı (Composite search key):** Birden fazla öznitelikten oluşan arama anahtarıdır.
  - Örnek: (dept_name, salary)
- **Leksikografik (sözlük) sıralama**: İkili anahtarlar şu kurala göre sıralanır:

$$(a_1, a_2) < (b_1, b_2) \text{ise:}$$

$$a_1 < b_1 \text{ veya}$$

$$a_1 = b_1 \text{ve } a_2 < b_2$$

- Önce birinci öznitelik karşılaştırılır. Eşitse, ikinci öznitelik dikkate alınır.
- Bu sayede bileşik indeksler hem ilk alan hem de (ilk, ikinci) birlikte sorgular için etkilidir.

Birden Fazla Öznitelik Üzerinde İndeksler (Indices on Multiple Attributes)

- Bir tablonun birden fazla özniteliğini birlikte içeren indekslere **bileşik (composite) indeks** denir.
- Örnek bileşik anahtar: (dept_name, salary)

Eşitlik koşulları `WHERE dept_name = 'Finance' AND salary = 80000`

- Bileşik indeks doğrudan ve verimli şekilde kullanılır. Yalnızca her iki koşulu da sağlayan kayıtlar getirilir.
- Ayrı ayrı indekslere göre daha etkilidir.

İlk öznitelik eşit, ikinci öznitelik aralık koşulu `WHERE dept_name = 'Finance' AND salary < 80000`

- Bileşik indeks etkin şekilde kullanılabilir. Çünkü indeks sıralaması (dept_name → salary) şeklindedir.
- Önce Finance, sonra salary aralığı taranır.

İlk öznitelikte aralık koşulu (verimsiz) `WHERE dept_name <= 'Finance' AND salary = 80000`

- Bileşik indeks etkin kullanılamaz. İndeks sıralaması bozulur.
- dept_name koşulunu sağlayan çok sayıda kayıt okunabilir, ancak bunların çoğu salary = 80000 koşulunu sağlamaz.

İndekslerin Oluşturulması (Creation of Indices)

```
CREATE INDEX <indeks_adı>
ON <tablo_adı> (<öznitelik_listesi>);
```

- CREATE UNIQUE INDEX, arama anahtarının aday anahtar (candidate key) olmasını zorunlu kılar; yani aynı değerden birden fazla kayıt olamaz.
  Not: Eğer SQL sistemi UNIQUE kısıtını destekliyorsa, ayrıca unique index tanımlamak zorunlu değildir.

- Çoğu veritabanı sistemi, indeks türünün (B+-tree, hash vb.) ve clustering bilgisinin belirtilmesine izin verir.
- Primary key veya unique constraint olan öznitelikler için indeksler tüm veritabanı sistemlerinde otomatik olarak oluşturulur.
- Bu alanlarda tekil değer garantisi vardır. Arama, join ve bütünlük kontrolleri çok sık yapılır. İndeks performansı ciddi şekilde artırır.
- Bazı veritabanları, foreign key alanları üzerinde de otomatik indeks oluşturur.

YILDIZ TEKNİK ÜNİVERSİTESİ

Yasemin Topuz

Yıldız Teknik Üniversitesi

ytopuz@yildiz.edu.tr