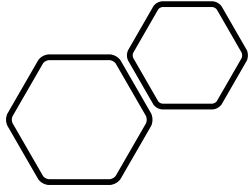




Database Tuning

Team Teaching MBD
Teknik Informatika - ITS



Meningkatkan Performa



Response Time

Rata-rata waktu untuk menunggu sebuah respon dari suatu *query*



Throughput

Jumlah *task* yang diselesaikan dalam satu satuan waktu tertentu (*transactions per second*)

Tuning

Pengukuran untuk meningkatkan performa

Level Aplikasi	Level Sistem	Level Hardware	Distribusi
<ul style="list-style-type: none">• Query: perancangan ulang skema, penggunaan indexing• Transaction: level isolasi, kode desain	<ul style="list-style-type: none">• Cache issues: ukuran cache, binding, ukuran I/O• Distribusi data pada device• Manajemen log	<ul style="list-style-type: none">• Jumlah CPU• Konfigurasi disk• Backup	<ul style="list-style-type: none">• Replikasi• Pendistribusian dan pemrosesan data

Redesign Schema

Perancangan Ulang Skema

Perancangan Ulang Skema == Denormalisasi

- Normalisasi mengurangi redundansi & menghindari anomali
- Normalisasi dapat meningkatkan performa
 - *Redundansi berkurang* => lebih banyak baris/page => lebih sedikit I/O
 - Dekomposisi => lebih banyak tabel => lebih banyak *clustered index* => index banyak tapi kecil-kecil

Normalisasi

Faktanya ...

Normalisasi dapat
menurunkan performa

- Contoh:

- FRS (NRP, KodeMK, Semester, Nilai)
- FD FRS: NRP \rightarrow NamaMhs
- Candidate Key FRS = (NRP, KodeMK, Semester)
- Jika NamaMhs menjadi atribut dari FRS maka table tersebut tidak memenuhi BCNF maupun 3NF, tetapi...
- Join yang dibutuhkan untuk menampilkan nama-nama mahasiswa yang mendapat A pada matakuliah CS305

```
SELECT M.NamaMhs
FROM Mahasiswa M, FRS F
WHERE M.NRP = F.NRP AND F.KodeMK = 'CS305'
AND F.Nilai = 'A'
```

- dan join cost-nya mahal.

Denormalisasi

- Tambahkan atribut NamaMhs di FRS

```
SELECT F.NamaMhs  
FROM FRS F  
WHERE F.KodeMK = 'CS305' AND F.Nilai = 'A'
```

- Join bisa dihindari, namun akan menambah redundansi
 - Modifikasi data akan lama (*update* pada data yang redundan perlu dilakukan di kedua table: FRS dan Mahasiswa)
 - Ukuran tabel membesar
 - Memungkinkan terjadinya inkonsistensi data

Perancangan ulang Skema: Partisi pada Tabel

- Sebuah tabel bisa menyebabkan performa *bottleneck*, jika
 - Tabel sangat sering digunakan, menyebabkan *lock contention*
 - Index tabel terlalu dalam (table memiliki banyak baris atau kunci pencariannya terlalu luas), meningkatkan konsumsi I/O
 - Baris data terlalu banyak, meningkatkan I/O
- Partisi tabel menjadi solusi pada permasalahan ini

Partisi Horisontal

- Jika akses hanya berpengaruh pada sebagian baris yang itu-itu saja, pertimbangkan untuk mempartisi tabel menjadi lebih kecil dan memuat baris-baris potensial tersebut.
 - Geografi kota (berdasarkan provinsi), kepegawaian (berdasarkan departemennya), mahasiswa aktif (berdasarkan masih aktif atau sudah lulus)

Keuntungan

- Membagi akses user dan mengurangi *contention* (terutama bila tabel-tabel tersebut diletakkan pada device server yang berbeda)
- Level index berkurang
- Baris data yang berpotensi muncul pada hasil query telah dikumpulkan ke *page-page* yang lebih sedikit, respon menjadi lebih cepat

Kerugian

- Kompleksitas bertambah
- Sulit menangani query yang melibatkan tabel di semua tabel partisi

Partisi Vertikal

- Bagi kolom-kolom menjadi dua subset dan replikasi key-nya
- Berguna bila table memiliki terlalu banyak kolom
 - Membedakan antara kolom yang sering diakses dan kolom yang jarang diakses
 - Bergantung pada tipikal query yang sering mengakses, beda query beda subset kolom
- **Contoh**
 - Kolom yang berkaitan dengan kompensasi pegawai (pajak, tunjangan, gaji) dipisahkan dengan kolom yang terkait pekerjaan (departemen, proyek, skill)
 - *Loseless*:
 - Karena NIP ada di setiap table partisi, maka masih memungkinkan untuk menarik semua informasi (kolom) dari seorang pegawai, meskipun membutuhkan sebuah perintah Join

Slide selanjutnya: Tugas Baca

Extents dan Storage Structures

Extent

- Sekumpulan blok-blok yang berdekatan pada penyimpanan masal yang bertindak sebagai sebuah alokasi unit untuk sebuah table atau index
 - Pengalokasikan sebuah ***extent*** untuk sebuah table memastikan page-page berdekatan dan mereduksi *latency* bila page lain dibutuhkan
 - Probabilitas skala pencarian berkurang → respon lebih cepat
 - Alternatifnya, semua extent bisa ditarik query dengan cost yang tidak lebih besar dari menarik sebuah page tunggal

Storage Structure

- *Heap*: baris-baris tidak terurut pada sebuah file (table tanpa *clustered index*)
- Tabel dan index yang terintegrasi dalam sebuah file (persebaran luas, menggunakan index clustered B⁺ tree atau hash)
- Baris-baris terurut pada sebuah file (padat, *clustered index* disimpan terpisah)

Heap

- Terjadi bila tidak ada *primary key* atau *unique constraint* yang dideklarasikan pada CREATE TABLE
- Jika tidak ada index, maka SELECT, UPDATE, dan DELETE akan menscan seluruh baris table
 - Penggunaan I/O yang eksesif
 - *Contention* karena semua transaksi DML harus mengunci table secara keseluruhan
- Baris-baris baru selalu di-INSERT di akhir table
 - Contention, karena semua transaksi DML harus mengunci page terakhir secara eksklusif

Indexing

Index

- **Keuntungan**

- Menghindari table scan
 - Di kasus tertentu, menghindari pengaksesan table secara menyeluruh
- Menjamin keunikan
- Insert random, tidak di akhir page
- Membantu join

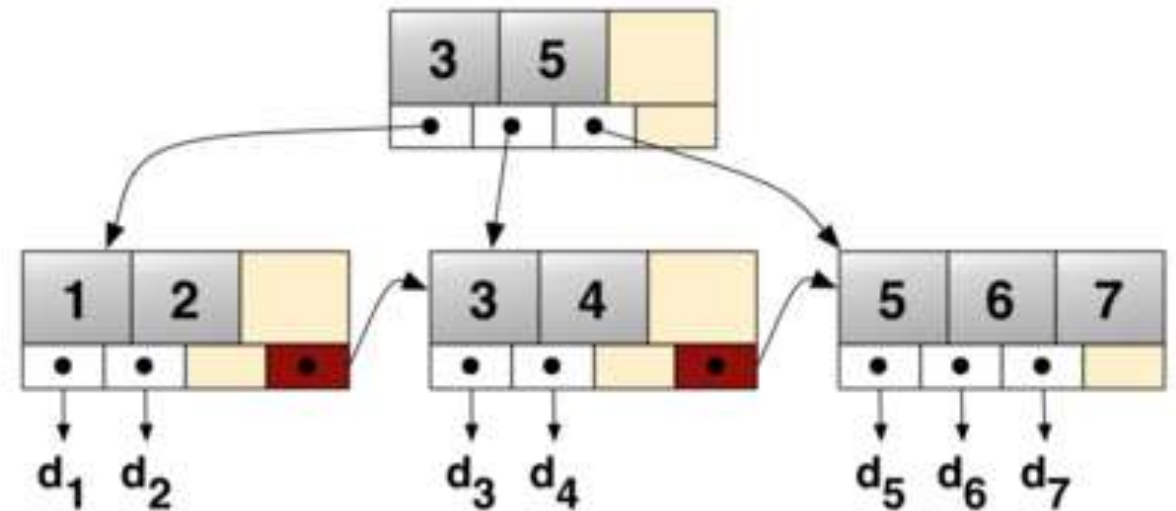
- Dibuat secara otomatis untuk menjamin primary key atau unique constraint

Clustered Index

- Satu clustered index hanya untuk satu table dan sebaliknya, karena pengelompokan tersebut menerangkan bagaimana baris-baris data tabel tersebut disimpan
- Umumnya terbentuk secara otomatis berdasarkan constraint PRIMARY KEY

Integrated Storage Structure: B⁺ Tree

- Sebuah clustered index diimplementasikan sebagai sebuah sparse tree di atas baris-baris terurut
 - Menghindari table scan dari sebagian besar statemen SQL
 - Sama-sama mendukung *range* maupun *point* query dengan baik
 - Tapi, membutuhkan data page splitting untuk menjamin keterurutan baris data

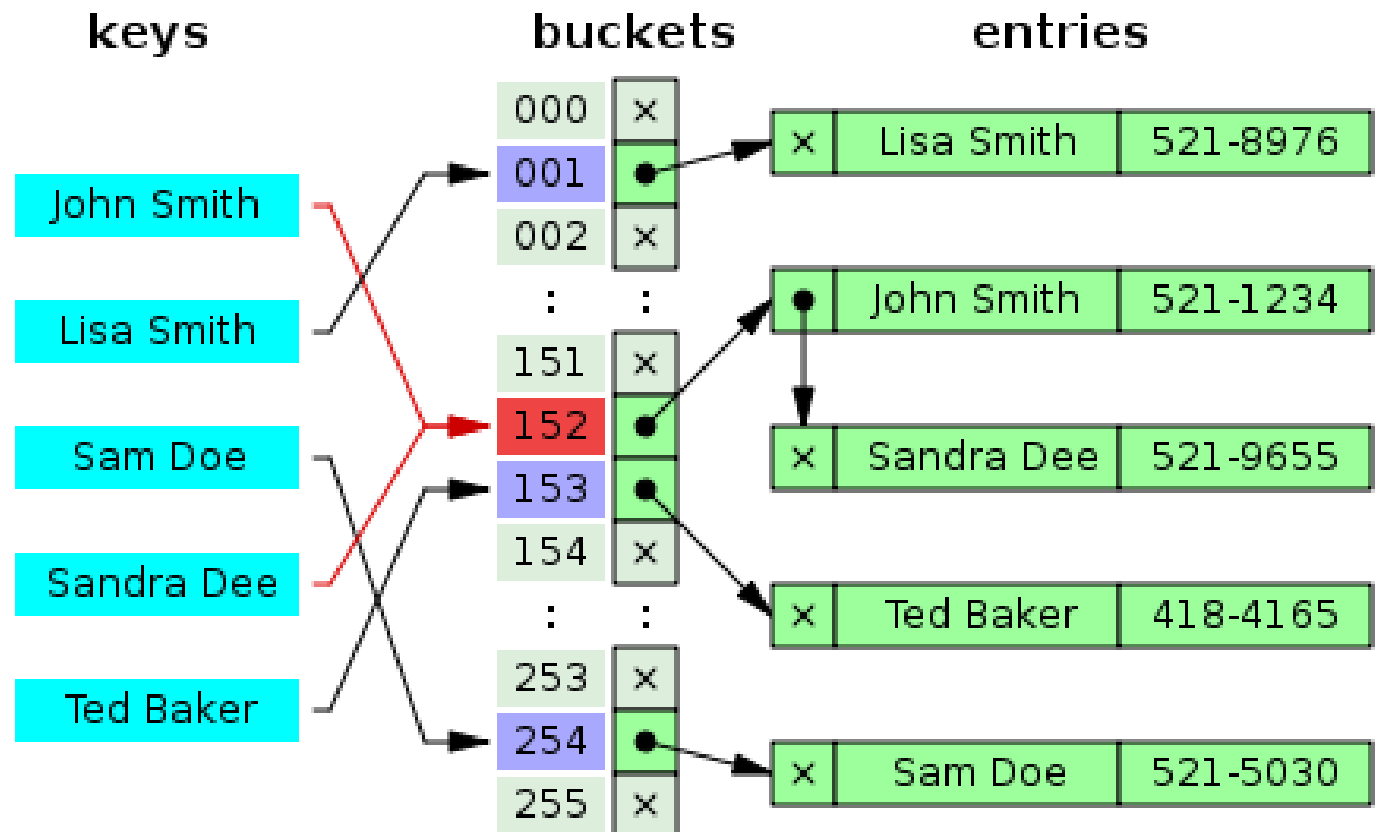


Sparse tree

Sorted rows

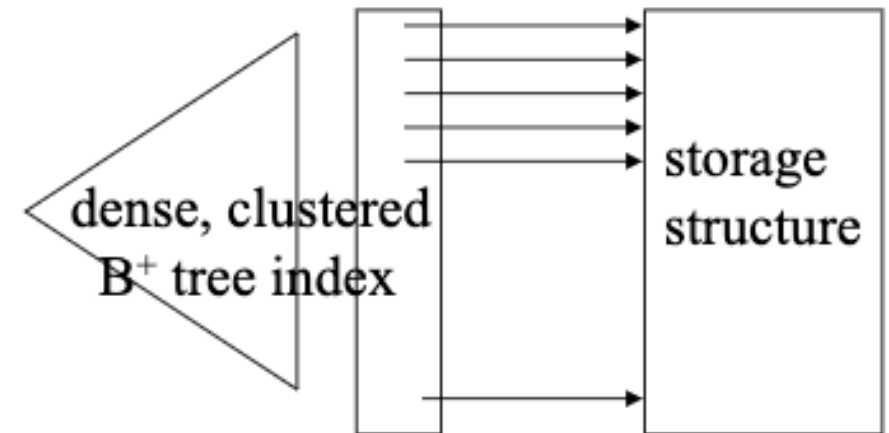
Integrated Storage Structure: Hash

- Sekumpulan bucket dengan hash yang terhubung
 - Tidak mendukung *range* query

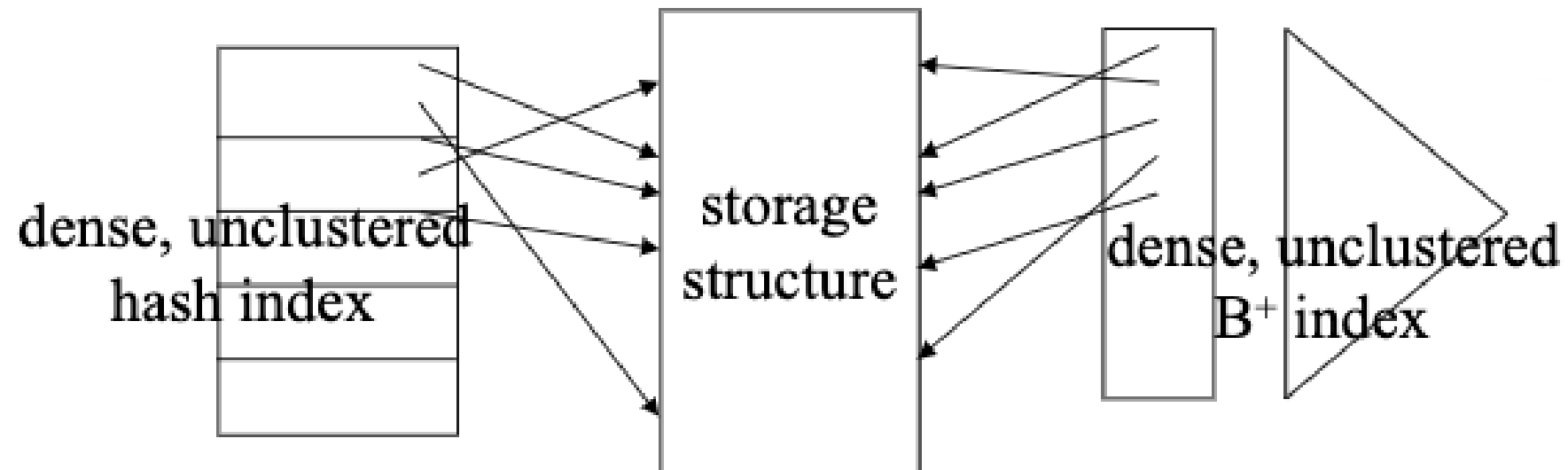


Clustered Index over Sorted File

- Clustered B+ tree index yang padat, disimpan terpisah, merujuk pada file terurut
 - Menghindari split data page: baris-baris data tidak harus berada di urutan tertentu karena index sudah padat
 - Jika baris tidak muat pada page, maka simpan baris tersebut di page lain masih dalam extent yang sama
 - Dukungan pencarian sama baiknya dengan integrated storage structure, tapi efisiensi akan menurun bila table terlalu dinamis (banyak eksekusi DML)



Unclustered Index



Unclustered Index lanj.

- Index yang padat (B⁺ Tree / Hash), tersimpan di file yang terpisah
 - Terbentuk otomatis Ketika UNIQUE constraint dideklarasikan
 - Jumlah unclustered index bisa lebih dari satu per table
 - Kemampuan pencariannya sama seperti clustered index, tetapi kurang efisien
 - Mendukung *covering index*
 - Overhead bertambah bila table dimodifikasi

Index Eksplisit

- Index dapat dibuat secara eksplisit
 - `CREATE CLUSTERED INDEX index_name ON table_name (search_key_attribute_list)`
 - Menyebabkan struktur penyimpanan di-reorganisasi
 - `CREATE UNCLUSTERED INDEX index_name ON table_name (search_key_attribute_list)`

Studi Kasus: Indexing

Skema Contoh

Primary Key

Student	Id	Name	Address	...
---------	----	------	---------	-----

Professor	Id	Name	DeptId	Salary	...
-----------	----	------	--------	--------	-----

Department	Id	Name	...
------------	----	------	-----

Transcript	StudId	CrsCode	Semester	Grade
------------	--------	---------	----------	-------

Index Covering

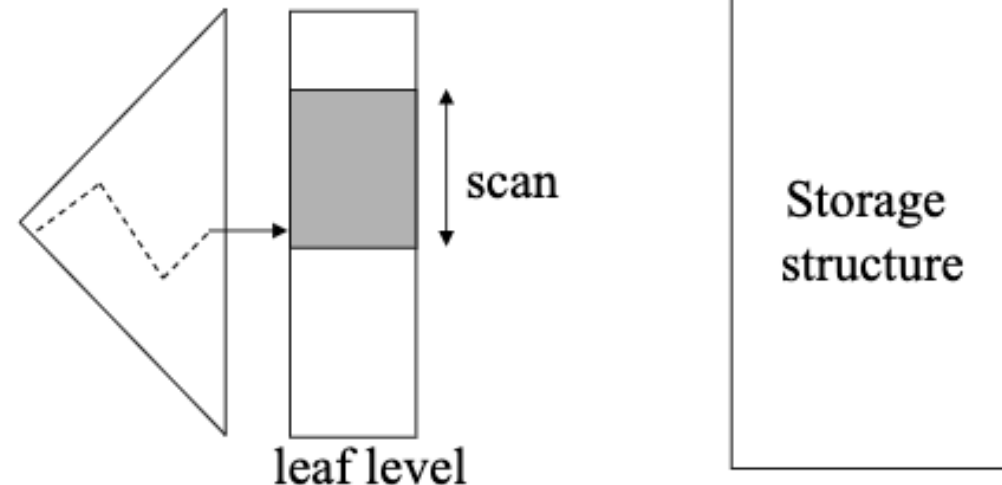
- Jika semua atribut query termasuk dalam *search key* pada (clustered/unclustered) index B⁺ tree, maka hasil dapat diperoleh dari index itu saja

Index Covering lanj.

- **Matching case:** atribut-atribut yang digunakan pada klausa WHERE termasuk prefix pada search key
 - Pencarian turun dari root index, kemudian scan segment pada leaf level
 - Mis.: dense index pada (DeptId, Name) menjawab query

Sudah terindeks pada tabel Department

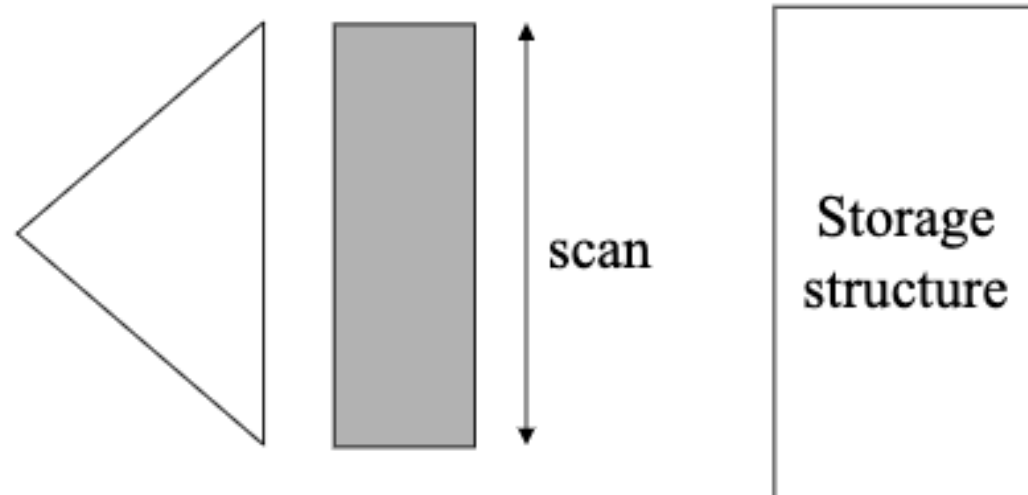
```
SELECT P.Name  
FROM Professor P  
WHERE P.DeptId = 'CS'
```



Index Covering lanj.

- **Non-matching case:** atribut-atribut pada klausa WHERE tidak termasuk prefix pada search key
 - Scan seluruh leaf level
 - Mis.: dense index pada (Id, Name, Address) menjawab query

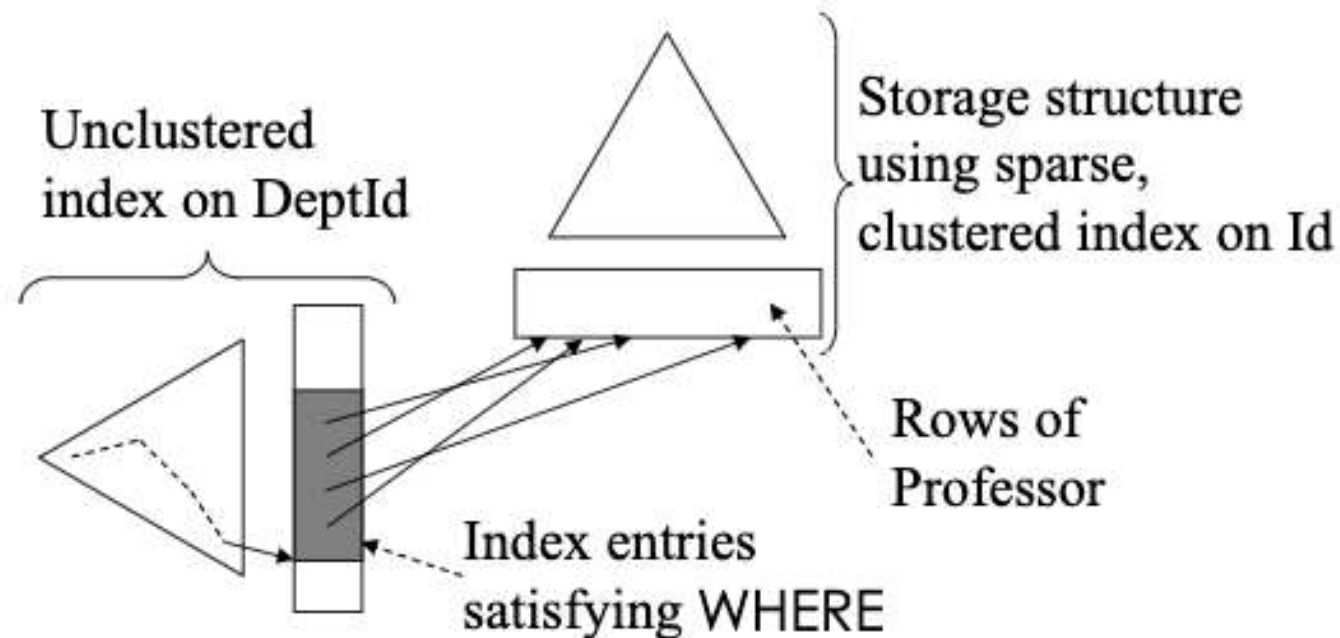
```
SELECT S.Id, S.Name  
FROM Student S  
WHERE S.Address = '1 Lake St'
```



```
SELECT P.Id, P.Name  
FROM Professor P  
WHERE P.DeptId = 'CS'
```

Memilih Index – Contoh 1

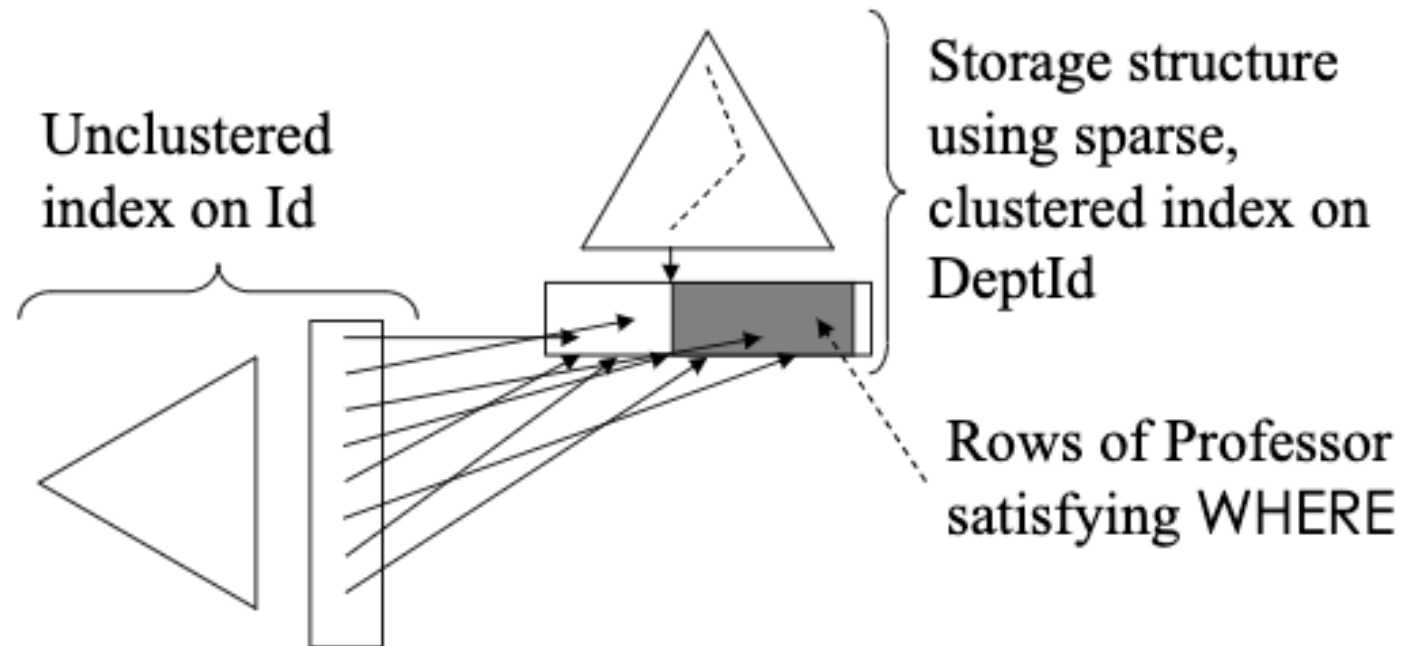
- Pilih index pada idDep
 - Jika *clustered index* sudah ada untuk NIP (karena primary key), maka index pada idDep berupa *unclustered*



```
SELECT P.Id, P.Name  
FROM Professor P  
WHERE P.DeptId = 'CS'
```


Memilih Index – Contoh 1 (lanj.)

- Tapi *unclustered index* bukan ide yang baik bila result set besar
 - Gunakan *unclustered index* untuk primary key (Id) dan *clustered index* untuk DeptId



Memilih Index – Contoh 2

```
SELECT S.Name  
FROM Student S, Transcript T  
WHERE S.Id = T.StudId AND T.CrsCode = 'CS305'
```




The diagram shows two curly braces under the WHERE clause. The first brace is under 'S.Id = T.StudId' and is labeled 'Join Condition'. The second brace is under 'T.CrsCode = 'CS305'' and is labeled 'Select Condition'.

- Jika tidak ada index yang bisa digunakan, DBMS akan
 - Menggunakan join *block-nested loops* berdasarkan kondisi join
 - Melanjutkan ke seleksi berdasarkan kondisi select
 - Tidak efisien – sebagian besar baris yang memenuhi kondisi join, gagal memenuhi kondisi select

Memilih Index – Contoh 2 (lanj.)

```
SELECT S.Name  
FROM Student S, Transcript T  
WHERE S.Id = T.StudId AND T.CrsCode = 'CS305'
```



The diagram shows two curly braces under the WHERE clause. The first brace is under 'S.Id = T.StudId' and is labeled 'Join Condition'. The second brace is under 'T.CrsCode = 'CS305'' and is labeled 'Select Condition'.

- Alternatifnya:
 - Pilih *clustered index* pada Transcript dengan search key (CrsCode)
 - Clustered index dengan search key (CrsCode, Semester, StudId) sudah ada karena atribut-atribut tersebut merupakan primary key dari Transcript
 - Pilih index pada Student dengan search key (Id)
 - Index dengan search key (Id) sudah ada karena atribut tersebut primary key dari Student
 - Index bisa berupa B⁺ tree atau hash, clustered maupun unclustered

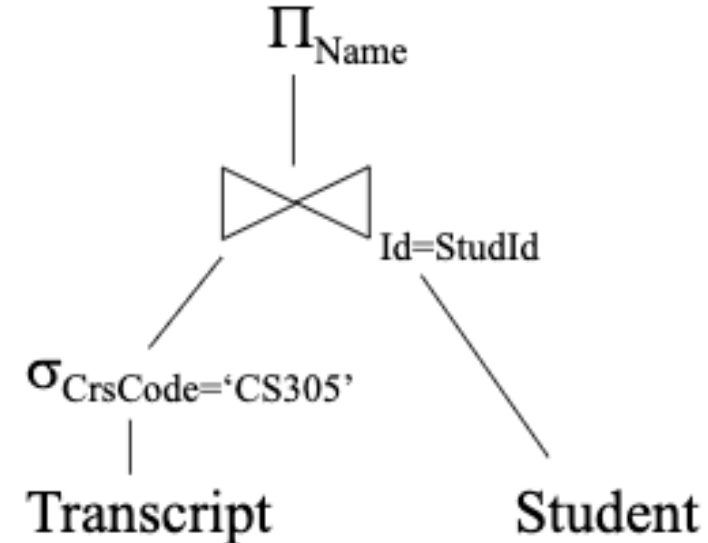
Memilih Index – Contoh 2 (lanj.)

```
SELECT S.Name  
FROM Student S, Transcript T  
WHERE S.Id = T.StudId AND T.CrsCode = 'CS305'
```

Join Condition

Select Condition

- DBMS selanjutnya dapat menerapkan join *index-nested loops*:
 - Ambil semua baris pada Transcript yang memenuhi kondisi seleksi
 - Untuk setiap baris hasil, gunakan index pada Student untuk mendapatkan baris unik yang memenuhi kondisi join (*index nesting* secara spesifik lebih efektif di kasus ini)



Memilih Index – Contoh 3

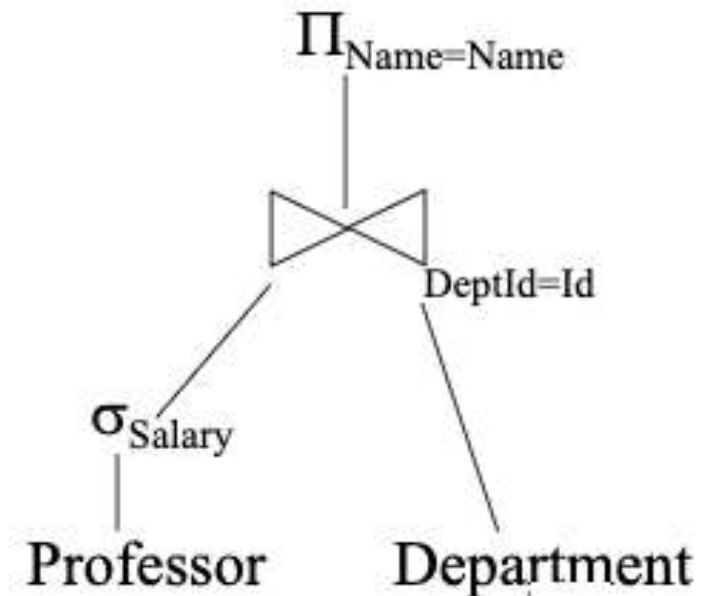
```
SELECT P.Name, D.Name  
FROM Professor P, Department D  
WHERE P.Salary BETWEEN 60000 AND 70000  
AND P.DeptId = D.Id
```

- Pilih index pada Professor dengan search key (Salary)
 - Seharusnya menggunakan B+ tree karena mendukung range query
 - Seharusnya *clustered* karena banyak menerapkan pencocokan

Memilih Index – Contoh 3 (lanj.)

```
SELECT P.Name, D.Name  
FROM Professor P, Department D  
WHERE P.Salary BETWEEN 60000 AND 70000  
AND P.DeptId = D.Id
```

- Pilih index pada Department dengan search key (Id)
 - Hash atau B+ tree karena sebuah departemen yang unik terhubung dengan baris di Professor
 - Cukup dengan menggunakan *unclustered*
 - Jangan membuang resource untuk *clustered*
- DBMS dapat menggunakan join *index-nested loops*:
 - Mengambil baris-baris Professor menggunakan *clustered index* pada Salary
 - Untuk tiap baris hasil, cocokkan baris dengan index pada Id



Memilih Index – Contoh 4

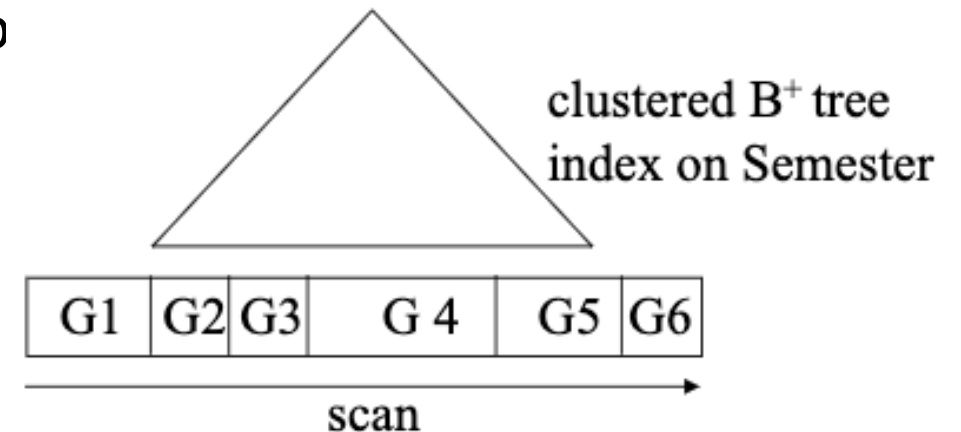
```
SELECT T.Semester, COUNT(*)  
FROM Transcript T  
WHERE T.Grade < 'A'  
GROUP BY T.Semester
```

- Pilih index pada Grade
 - Gunakan index untuk mendapatkan baris-baris yang memenuhi WHERE
 - B+ tree, karena range sudah dispesifikasi
 - *Clustered*, karena banyak baris yang memenuhi WHERE
 - Urutkan hasil berdasarkan Semester dan hitung barus pada tiap group
- Bukan Langkah yang tepat karena kondisi WHERE tidak selektif dan banyak hasil *intermediate* yang harus diurutkan
 - Mungkin akan lebih tepat bila kondisi seleksi T.Grade < 'C', range hasil tidak sebesar T.Grade < 'A'

Memilih Index – Contoh 4 (lanj.)

```
SELECT T.Semester, COUNT(*)  
FROM Transcript T  
WHERE T.Grade < 'A'  
GROUP BY T.Semester
```

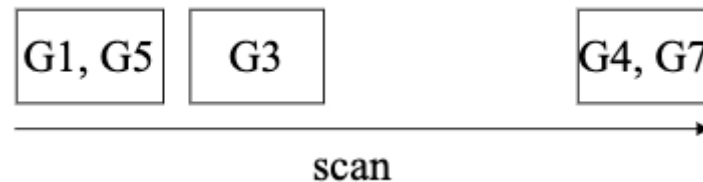
- Pilih index pada Semester
 - Gunakan *clustered index* pada Semester, sehingga baris-baris hasil terkelompokkan secara otomatis
 - Scan Transcript, hitung baris hasil di tiap group
 - Index bisa berupa B⁺ tree:



Memilih Index – Contoh 4 (lanj.)

```
SELECT T.Semester, COUNT(*)  
FROM Transcript T  
WHERE T.Grade < 'A'  
GROUP BY T.Semester
```

- Index pada Semester bisa berupa hash:
 - Hash bisa diterima, karena query tidak menerapkan *range* pada Semester
 - Hash menyimpan semua row dari group tertentu pada *bucket*; lebih mudah untuk menghitung jumlah baris yang ada di tiap group
 - Scan *bucket*



- Alternatif yang baik, karena WHERE tidak selektif; seandainya sebaliknya, maka scan akan memakan cost yang eksesif

Memilih Index – Contoh 5

```
(1) SELECT T.CrsCode  
      FROM Transcript  
      WHERE T.StudId = :studid
```

```
(2) SELECT T.StudId  
      FROM Transcript T  
      WHERE T.CrsCode = :code  
            AND T.Semester = :sem
```

- Kedua query sering ditanyakan
 - Solusi 1 : *clustered index* pada StudId untuk (1), *unclustered index* pada (CrsCode, Semester) untuk (2)
 - Permasalahan: kedua result set berukuran sedang, penggunaan *unclustered index* untuk (2) menyebabkan overhead yang eksekusi
 - Solusi 2 : *clustered index* pada (CrsCode, Semester) untuk (2), *unclustered index* pada (StudId, CrsCode) untuk (1)
 - (1) menggunakan *index covering (matching case)*

Pemilihan *Clustered Index (CI)*

DO

- Gunakan CI untuk mendukung:
 - Point query dengan result set besar
 - Range query
 - Klausa ORDER BY
 - *Index-nested* (mendapatkan semua baris yang memenuhi nilai search key tertentu) dan *sort-merge join*

DON'T

- Jika kondisi tidak seperti di DO
- Jika search key terlalu sering diupdate
 - Akan memakan resource untuk mereorganisasi index:
 - Hash: dari bucket ke bucket
 - B+ tree: bila node dekat root perlu dimodifikasi
- B+ tree:
 - bila search key merupakan primary key dan secara monoton bertambah melalui insertion data (karena semua inserts disimpan pada leaf page → menyebabkan *contention*)
 - Misal: invoice no., date, time

Pemilihan *Unclustered Index (UI)*

- Gunakan UI untuk mendukung:
 - Query dengan result set kecil
 - Join *index-nested loops*, bila bobot dari atribut join kecil/sedikit
 - *Index covering*
 - *Point query* dengan result set kecil

Tips Tambahan!

1. Jika atribut unik, maka deklarasikan
 - Query optimizer akan menggunakannya pada query planning
 - Hanya satu row yang memenuhi/menjawab atribut join atau atribut pencarian
2. Atur **fill factor**
 - Menjadi 100% bila table *read-only*
 - Atur ke nilai lebih kecil bila table dinamis (sering mengalami modifikasi)
3. Gunakan search key seminimal mungkin untuk mereduksi level index
4. Bijaklah menambahkan *unclustered index* hanya jika diperlukan saja

Tuning SQL

Tuning SQL

- PETUNJUK

- Hindari sort
 - Gunakan sort-merge join
 - Penggunaan DISTINCT, UNION, EXCEPT, ORDER BY, dan GROUP BY menyebabkan sort. Hindari penggunaannya sebisa mungkin.
- Meminimalisir komunikasi
 - Jangan gunakan cursor (karena komunikasi mungkin diperlukan pada setiap penarikan baris hasil) – review kembali materi tentang Cursor di Active DB
 - Gunakan *stored procedure* hanya jika agregasi informasi diperlukan oleh aplikasi

Tuning SQL (lanj.)

- PETUNJUK

- Hindari penggunaan View jika mungkin, menyebabkan join yang tidak diperlukan
- Pertimbangkan untuk merestrukturisasi query
 - Perbedaan penulisan sintaks bisa berimbas perbedaan cost, tergantung pada kondisi table dan index yang tersedia – review kembali materi tentang Aljabar Relasional

Query Optimizer

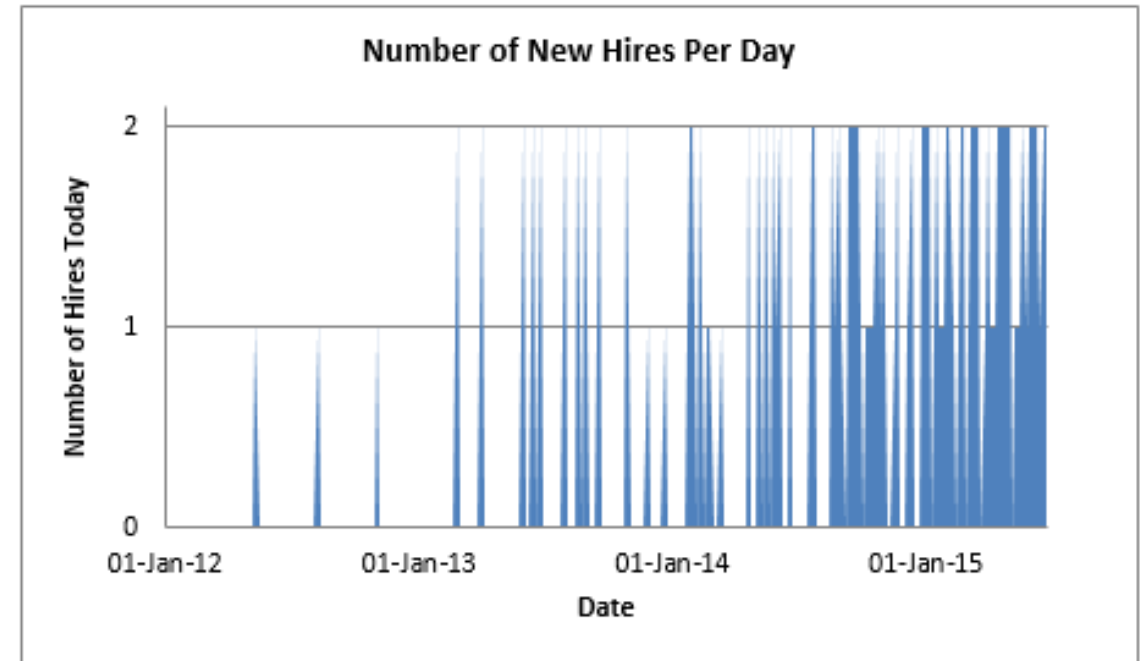
- Pada dasarnya sebagian besar DBMS telah memiliki *engine* query optimizer yang dijalankan secara otomatis tiap eksekusi query untuk memilih *path execution plan* yang paling efisien
- Namun, DBA juga dapat membantu menginterferensi Query Optimizer agar bekerja lebih optimal

Interferensi *Query Optimizer*

- Statistik: Digunakan oleh optimizer untuk mengestimasi cost dari query plan (didasarkan pada ukuran result setnya)
 - Tabel: jumlah baris, jumlah distinct value dari atribut (ragam variasi nilai), nilai max dan nilai min dari atribut
 - Index: kedalaman level, banyaknya leaf page, banyaknya distinct value dari search key
 - Histogram dari nilai-nilai atribut

Interferensi *Query Optimizer* (lanj.)

- Histogram dari nilai-nilai atribut
 - Contoh: Gunakan UI jika histogram menunjukkan bahwa jumlah baris dengan nilai atribut yang spesifik seperti di query, sedikit jumlahnya. Selain itu, gunakan scan.
 - Harus diupdate secara periodical jika table termasuk dinamis
 - DBA bisa membuat script untuk mengupdate histogram



Interferensi *Query Optimizer* (lanj.)

- Yang perlu diperhatikan sebelum mengeksekusi SQL statement
 - Urutan JOIN
 - Metode JOIN: right outer join, left outer join, self join, dll.
 - Index yang digunakan

System Level Tuning

Cache

- Menyimpan *page* yang terakhir dirujuk pada main memory dikarenakan kemungkinan *page* itu digunakan kembali lebih besar
- Penting untuk mengacu pada performa yang realistis
 - Hit rate minimal 90% masih wajar
- Digunakan pada data *page* (data cache) dan query plan *page* (procedure cache – *stored procedure*)

Log

- Log transaksi merupakan contoh dari [heap storage structure](#): record selalu ditambahkan di akhir
- Letakkan log pada device/server yang terpisah dengan database
 - Menghindari *contention* dengan akses database – tidak crash

Referensi

- Database Systems, An Application-Oriented Approach (2nd ed.) – Chapter 12. Michael Kifer, Arthur Bernstein, Philip M. Lewis. Pearson: Addison Wesley, 2006.