Penerapan Pohon pada *File Indexing* Sistem Manajemen Basis Data Relasional

Ikraduya Edian 13517106
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13517106@std.stei.itb.ac.id

***Abstract*—The need for an efficient Database Management System is a must in the era of information technology as it is today. The efficiency of the database management system is determined by many factors one of which is the use of indexes in organizing data. Through this paper, author will explain about the application of tree data structures in database indexes to speed up data access. The purpose of this paper is to provide readers with knowledge about the application of tree data structure in relational database management system.**

***Keywords*—efficiency, database index, tree, relational database management system.**

I. PENDAHULUAN

Basis Data adalah sekumpulan informasi yang diatur sedemikian rupa sehingga dapat diakses, dikelola dan diperbarui dengan mudah. Sistem Manajemen Basis Data (*Database Manajement System*) adalah perangkat lunak yang mampu mengakses, mengelola, dan memperbarui basis data. Seiring dengan bertambahnya data yang disimpan di basis data, kemangkusan dari suatu DBMS menjadi sangat penting.

Hingga saat ini terdapat dua konsep basis data, relasional dan non-relasional. Basis data relasional adalah basis data yang berbasis pada model relasional, berlawanan dengan basis data non-relasional yang tidak mengikuti model relasional. Pada makalah ini hanya akan dibahas mengenai penggunaan struktur data pohon pada Sistem Manajemen Basis Data Relasional (*Relational Database Management System*).

Indeks di RDBMS adalah teknik penggunaan struktur data untuk mengefisiensi proses pengaksesan, pengelolaan, dan perbaharuan tabel pada basis data relasional. Indeks seperti pedang bermata dua, penggunaan indeks yang tepat dapat membuat *database* yang lamban berkinerja lebih baik. Sebaliknya, penggunaan indeks yang tidak tepat dapat membuat *database* berkinerja baik menjadi lamban. Mempelajari implementasi struktur data di balik indeks adalah cara yang baik untuk menyusun strategi penggunaan *indexing*.

II. TEORI DASAR

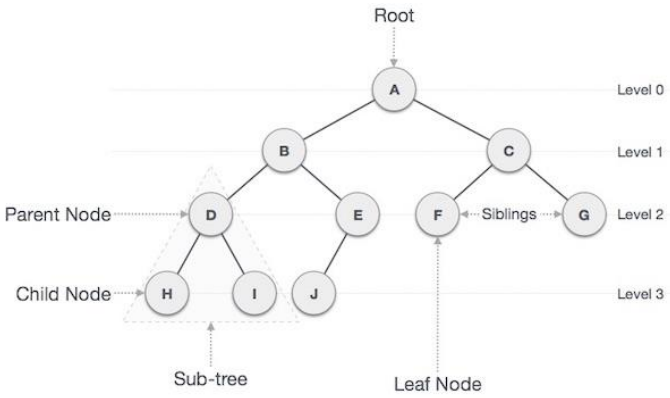
A. Pohon

Pohon (*tree*) adalah graf tak-berarah terhubung yang tidak mengandung sirkuit. Sirkuit adalah lintasan pada graf yang berawal dan berakhir pada simpul yang sama. Struktur data pohon adalah kumpulan simpul (*node*) yang saling terhubung oleh sisi (*edge*) dan mempunyai nilai (*key*) yang tersimpan pada tiap-tiap simpul [1].

Misalkan G = (V, E) adalah graf tak-berarah sederhana dan jumlah simpulnya n. Maka, semua pernyataan berikut adalah ekivalen:

1. G adalah pohon.
2. Setiap pasang simpul di dalam G terhubung dengan lintasan tunggal.
3. G terhubung dan memiliki m = n – 1 buah sisi.
4. G tidak mengandung sirkuit dan penambahan satu sisi pada graf akan membuat hanya satu sirkuit
5. G terhubung dan semua sisinya adalah jembatan [1]

Terdapat dua jenis pohon, yang pertama adalah pohon bebas (*free tree*). Pohon bebas adalah pohon yang tidak mempunyai terminologi seperti akar, daun, anak, dll. Ada juga pohon berakar (*rooted tree*), yang memiliki akar, daun, anak, orang tua, dll [1].



Gambar 2.1.1 Terminologi Pohon Berakar

Sumber :
https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm

Pohon mempunyai banyak terminologi. Terminologi pada pohon berakar diantaranya:

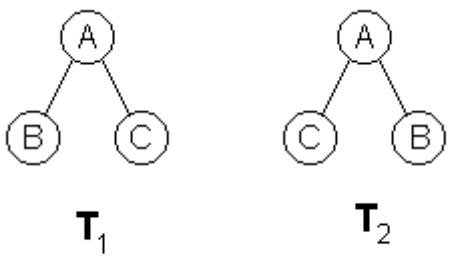
- Akar (*root*)
  Simpul di bagian paling atas pohon. Pada gambar, akar

dari pohon adalah ‘A’

- Anak (*child* atau *children*) dan Orang tua (*parent*)
  Simpul ‘H’ dan ‘I’ adalah anak-anak dari simpul ‘D’, ‘D’ adalah orang tua dari anak-anak itu
- Lintasan (*path*)
  Lintasan dari ‘A’ ke ‘J’ adalah ‘A’, ‘B’, ‘E’, ‘J’. Panjang lintasa dari ‘A’ ke ‘J’ adalah 3.
- Aras (*level*) atau Tingkat
  Aras dari sebuah simpul menyatakan generasi dari simpul tersebut. Pada gambar, simpul ‘A’ mempunyai aras 0 sedangkan simpul ‘H’ beraras 3.
- Saudara kandung (*sibling*)
  Simpul ‘D’ dan ‘E’ merupakan saudara kandung karena orang tua mereka sama yaitu ‘B’. Sedangkan ‘E’ dan ‘F’ bukan merupakan saudara kandung.
- Keturunan (*descendant*) dan leluhur (*ancestor*)
  Jika terdapat lintasan antara simpul x dan y, aras x lebih kecil daripada arax y, maka x adalah leluhur y dan y adalah keturunan x. Contohnya pada gambar, ‘G’ adalah keturunan ‘A’ dan ‘A’ adalah leluhur ‘G’.
- Upapohon (*subtree*)
  Upapohon adalah bagian dari keseluruhan pohon, dapat juga didefinisikan sebagai keturunan dari sebuah simpul pohon. ‘D’, ‘H’ dan ‘I’ merupakan upapohon dari pohon keseluruhan.
- Derajat (*degree*)
  Derajat sebuah simpul adalah jumlah anak pada simpul tersebut. Derajat ‘E’ adalah 1, sedangkan derajat ‘C’ adalah 2.
- Daun (*leaf*)
  Daun adalah simpul yang berderajat nol (tidak mempunyai anak). Simpul ‘H’, ‘I’, ‘J’, ‘F’ dan ‘G’ adalah daun.
- Simpul dalam (*internal nodes*)
  Simpul yang mempunyai anak disebut simpul dalam. Simpul ‘A’, ‘B’, ‘C’, ‘D’, dan ‘E’ adalah simpul dalam.
- Tinggi (*height*) atau Kedalaman (*depth*)
  Aras maksimum dari suatu pohon disebut tinggi atau kedalaman pohon tersebut [1]

Terdapat beberapa jenis struktur data pohon yang umum digunakan, diantarnya:

1. Pohon Terurut (*Ordered Tree*)
   Pohon berakar yang urutan anak-anaknya penting disebut pohon terurut [1].

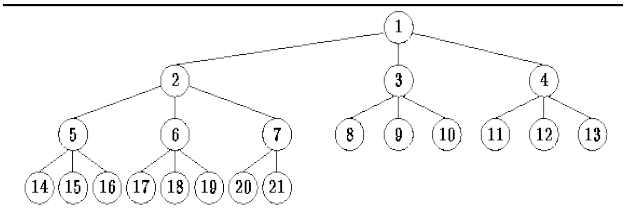


Gambar 2.1.2 Ilustrasi Pohon Terurut

Sumber : http://cs.lmu.edu/~ray/notes/orderedtrees/

Jika T_1 dan T_2 adalah pohon terurut, maka T_1 tidak sama dengan T_2 karena urutan anak-anaknya berbeda.

2. Pohon N-ary (N-ary *Tree*)
   Pohon berakar yang setiap simpul cabangnya mempunyai paling banyak n buah anak disebut pohon n-ary [1].

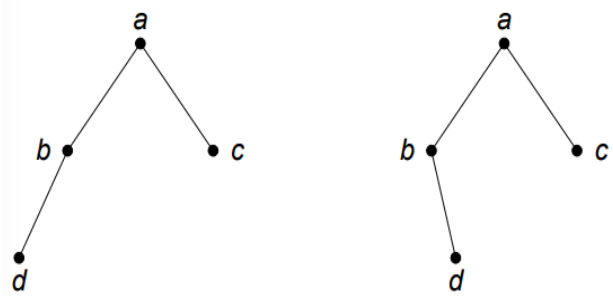


Gambar 2.1.3 Ilustrasi Pohon N-ary

Sumber : https://book.huihoo.com/data-structures-and-algorithms-with-object-oriented-design-patterns-in-c++/html/page356.html

Gambar 2.1.3 adalah ilustrasi pohon n-ary dengan n = 3.

3. Pohon Biner (*Binary Tree*)
   Pohon n-ary dengan n = 2 adalah pohon biner. Simpul pada pohon biner dapat dibedakan antara anak kiri (*left child*) dan anak kanan (*right child*) [1].

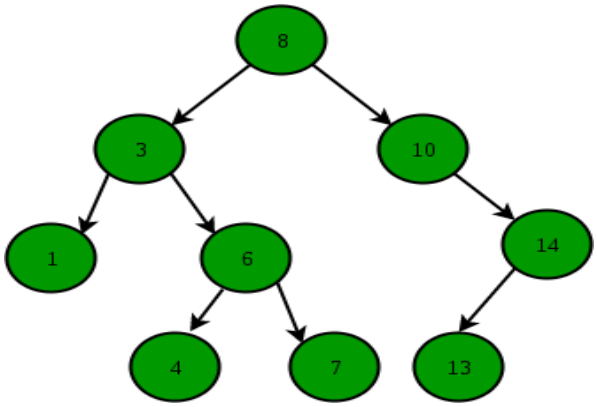


Gambar 2.1.4 Ilustrasi Pohon Biner

Sumber : http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2013-2014/Pohon%20(2013).pdf

Dua pohon pada gambar 2.1.4 merupakan pohon biner yang berbeda. Pada pohon pertama ‘d’ adalah anak kiri dari ‘b’ sedangkan pada pohon kedua ‘d’ adalah anak kanan dari ‘b’.

4. Pohon Pencarian Biner (*Binary Search Tree*)
   Pohon Pencarian Biner adalah struktur data pohon biner yang digunakan untuk melakukan operasi pencarian dengan kunci tertentu pada simpul pohon. Keuntungan menggunakan pohon ini adalah waktu pencarian yang selalu seimbang.



Gambar 2.1.5 Ilustrasi Pohon Pencarian Biner

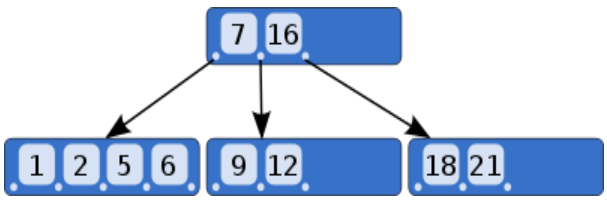
Sumber : https://www.geeksforgeeks.org/binary-search-tree-data-structure/

Struktur data BST memiliki beberapa sifat:

- Upapohon kiri sebuah simpul akar hanya berisi simpul-simpul yang nilainya lebih kecil dari nilai akarnya
- Upapohon kanan sebuah simpul akar hanya berisi simpul-simpul yang nilainya lebih besar dari nilai akarnya
- Upapohon kanan dan kiri juga merupakan pohon pencarian biner (definisi rekursif) [2]

5. B-Tree

   B-Tree adalah pohon pencarian seimbang (*self-balancing search tree*) m-ary, dengan m adalah orde atau jumlah maksimum anak tiap simpul. Tiap simpul dalam maupun daun menyimpan data dan kunci sekaligus. *Pointer* yang terletak sebelum sebuah kunci pastilah menunjuk ke simpul yang tiap datanya bernilai lebih kecil dari kunci tersebut. Sedangkan *pointer* yang terletak setelah sebuah kunci menunjuk ke simpul yang tiap datanya bernilai lebih besar dari kunci tersebut. Struktur data semacam ini menjaga data tetap terurut, sehingga memungkinkan operasi pencarian (*searching*), akses sekuensial (*sequential access*), penyisipan (*insertion*) dan penghapusan (*deletion*) dalam waktu logaritmik [4].



Gambar 2.1.6 Ilustrasi B-Tree

Sumber : https://commons.wikimedia.org/wiki/File:B-tree.svg

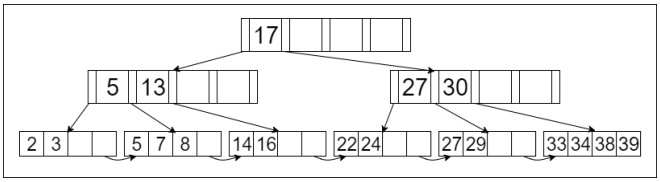
Gambar 2.1.6 merupakan ilustrasi B-Tree berorde 5 (m=5) dengan maksimal data tiap simpul berjumlah 4 (m-1).

Struktur data B-Tree memiliki beberapa sifat :

- Semua panjang lintasan dari akar menuju daun selalu sama.
- Jika sebuah simpul mempunyai n buah anak, simpul tersebut mempunyai n-1 buah kunci.
- Setiap simpul dalam (kecuali akar) setidaknya memiliki jumlah kunci yang setengah penuh.
- Akar mempunyai paling sedikit 2 anak jika bukan sebuah daun [3].

6. B+ Tree

   B+ Tree adalah lanjutan dari B-Tree yang tetap menjaga keterurutan data, sehingga memungkinkan operasi penyisipan, penghapusan dan pencarian yang efisien. Pada B-Tree, kunci dan data (*record*) dapat disimpan di simpul dalam dan simpul daun. Sedangkan pada B+ Tree, record hanya dapat disimpan di simpul daun sementara simpul dalam hanya menyimpan kunci. B+ Tree biasanya digunakan untuk menyimpan data dalam jumlah besar yang biasanya tidak dapat disimpan di memori utama (RAM). Dikarenakan memori utama biasanya sangat terbatas, simpul dalam (kunci untuk mengakses data) dari sebuah B+ Tree disimpan di memori utama, sedangkan simpul daun disimpan di memori sekunder (diska keras atau *hard disk*) [5].



Gambar 2.1.7 Ilustrasi B+ Tree

Sumber : https://cs.stackexchange.com/questions/50384/how-the-deletion-takes-place-in-b-tree

B. Sistem Manajemen Basis Data

Sistem Managemen Basis Data (*Database Management System* atau DBMS) adalah sebuah sistem perangkat lunak yang digunakan untuk membuat dan mengelola basis data. DBMS ini memungkinkan pengguna secara sistematis membuat, mengambil, merubah dan mengelola data di dalam basis data [6]. Terdapat beberapa model basis data populer beserta DBMS-nya seperti:

1. *Relational Database Management System* (RDBMS)

   RDBMS adalah sistem manajemen basis data yang berdasarkan pada model relasional yang pertama kali diperkenalkan oleh E. F. Codd. Sebagian besar RDBMS menggunakan bahasa SQL untuk mengakses data. Data pada RDBMS disimpan ke dalam bentuk objek *database* yang biasa disebut relasi (*relation*) atau tabel (*table*) [7].

CUSTOMER TABLE

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Gambar 2.2.1 Contoh Tabel Basis Data Relasional

Sumber : https://www.tutorialspoint.com/sql/sql-rdbms-concepts.htm

Pada gambar 2.2.1 setiap tabel disusun dari suatu entitas lain yang disebut *field*. *Field-field* pada tabel CUSTOMER diantaranya ID, NAME, AGE, ADDRESS, dan SALARY. Tiap baris pada pada tabel tersebut disebut *record*. Tabel CUSTOMER mempunyai 7 buah *record*.

2. *NoSQL Database Management System*
   NoSQL DBMS adalah DBMS yang tidak menggunakan bahasa SQL standar untuk pengaksesan data. NoSQL database tidak berdasar pada model relasional (*relation-less*). Ada beberapa model basis data untuk NoSQL DBMS sesuai dengan penggunaanya seperti *Key* / *Value*, *Column*, *Document*, dan *Graph Model* [8].

Feature	NoSQL Databases	Relational Databases
Performance	High	Low
Reliability	Poor	Good
Availability	Good	Good
Consistency	Poor	Good
Data Storage	Optimized for huge data	Medium sized to large
Scalability	High	High (but more expensive)

Gambar 2.2.2 Perbedaan Basis Data NoSQL dan Basis Data Relasional

Sumber : https://blog.pandorafms.org/nosql-databases-the-definitive-guide/

Perbedaan NoSQL DBMS dengan RDBMS terletak pada fleksibilitas, dimana DBMS Relasional memerlukan struktur atau skema data sejak pertama kali dibuat sedangkan NoSQL tidak, hal ini menyebabkan NoSQL menawarkan sejumlah besar fleksibilitas yang tidak dimiliki RDBMS.

C. File Indexing Basis Data

Data dalam *database* disimpan ke dalam bentuk *record*. Setiap *record* mempunyai sebuah kunci *field* (*field key*) atau juga disebut kunci pencarian (*search key*) yang menjamin keunikan tiap *record*. *Indexing* adalah cara untuk mengoptimasi kinerja suatu *database* dengan meminimumkan jumlah akses diska ketika memproses suatu *query*. Indeks *database* adalah data struktur yang digunakan untuk menemukan dan mengakses data pada *database* dengan cepat. Terdapat 2 jenis index:

1. Indeks Terurut (*Ordered Index*)
   Indeks terurut didasarkan pada kunci yang terurut.
2. *Hash Index*
   Indeks yang didasarkan pada kunci yang didistribusikan secara merata ke berbagai *bucket*. Bucket yang ditempati sebuah index ditentukan oleh suatu fungsi *hash* [9].

Terdapat 3 tipe indeks terurut:

1. *Dense Index*
   Pada dense index terdapat index record untuk setiap *key field*. Hal ini menyebabkan pencarian menjadi lebih cepat namun membutuhkan lebih banyak ruang (*space*) untuk menyimpan *index record*. *Index record* terdiri dari nilai *key field* dan sebuah *pointer* yang menunjuk ke *record* sebenarnya (*actual record*) di dalam diska [10].

Gambar 2.3.1 Ilustrasi Dense Index

Sumber : https://www.tutorialspoint.com/dbms/dbms_indexing.htm

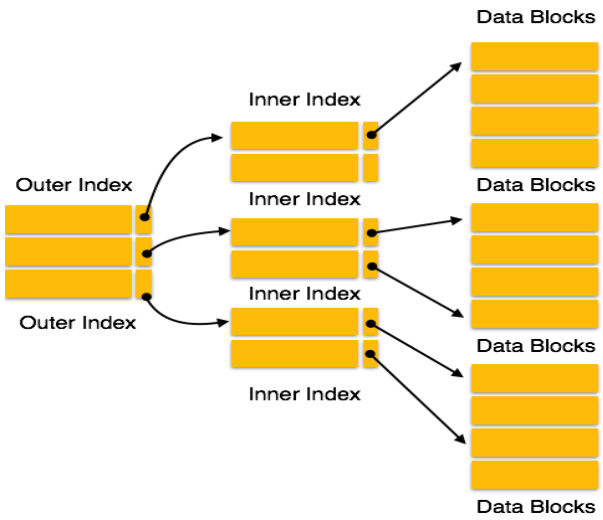
2. *Sparse Index*
   Pada *sparse index*, tidak terdapat *index record* untuk setiap nilai *key field*. Index record terdiri dari nilai *key field* dan sebuah *pointer* yang menunjuk ke *record* di dalam diska. Untuk mencari sebuah *record*, *index record* diproses hingga ke data yang ditunjuk *pointer*. Jika data yang dicari tidak terletak pada indeks yang ditunjuk, akan dilakukan pencarian sekuensial sampai data ditemukan [10].

Gambar 2.3.2 Ilustrasi Sparse Index

Sumber : https://www.tutorialspoint.com/dbms/dbms_indexing.htm

3. *Multi-level Index*
   Index record disimpan di dalam diska bersama dengan record yang sebenarnya. Ukuran *database* yang semakin besar, menyebabkan ukuran indeks-indeks juga ikut membesar. Jika indeks satu-level (*single-level*) digunakan, maka ukuran indeks yang besar tidak dapat disimpan ke dalam memori utama (RAM). Hal ini menyebabkan indeks akan disimpan di diska, sehingga

akan terjadi *multiple disk access* yang memperlambat waktu pencarian [10].



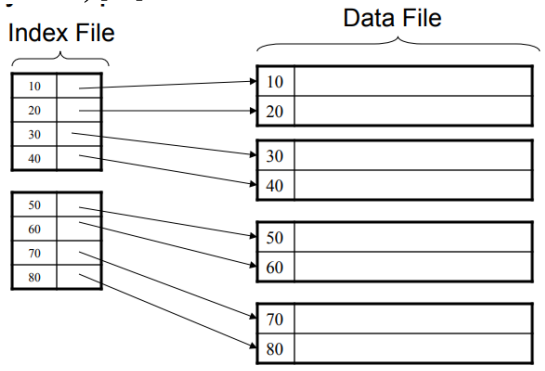
Gambar 2.3.3 Ilustrasi Multi-level Index

Sumber :
https://www.tutorialspoint.com/dbms/dbms_indexing.htm

Multi-level index memecah indeks menjadi indeks-indeks kecil agar indeks paling luar (*outermost*) bisa dengan mudah disimpan di memori utama pada saat proses pencarian. Implementasi dari *multi-level index* ini biasa menggunakan struktur data B-Tree dan B+ Tree.

Berdasarkan keterurutan indeks dan keterurutan *actual record*, indeks dapat diklasifikasi menjadi dua:

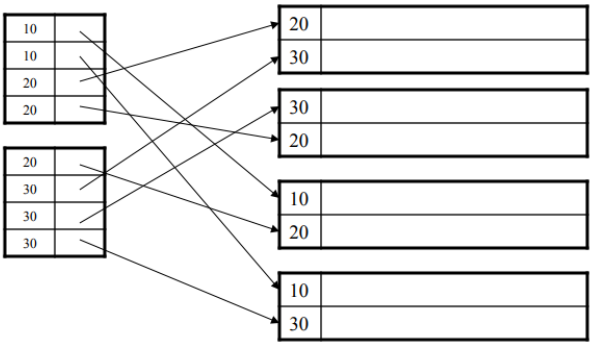
1. *Clustered Index*
   Clustered Index adalah ketika *index record* yang berdekatan menandakan bahwa lokasi *record* sebenarnya juga berdekatan. Biasanya, pada jenis index ini data *record* sebenarnya terletak di simpul daun (bukan sebuah keharusan) [11].



Gambar 2.3.4 Ilustrasi Clustered Index

Sumber:
https://courses.cs.washington.edu/courses/cse444/09sp/lectures/lecture15.pdf

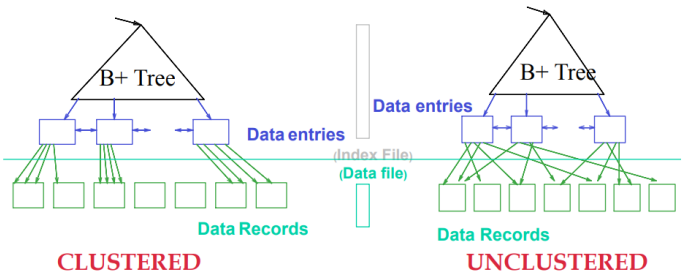
2. *Unclustered Index*
   Unclustered Index adalah ketika urutan indeks tidak menentukan keterurutan *record* sebenarnya. Pada jenis indeks ini *pointer* pada simpul daun menunjuk ke data *record* sebenarnya [11].



Gambar 2.3.5 Ilustrasi Unclustered Index

Sumber :
https://courses.cs.washington.edu/courses/cse444/09sp/lectures/lecture15.pdf

Multi-level index dengan B+ Tree juga dapat dibedakan atas sifat *clustered* dan *unclustered* ini. Simpul daun B+ Tree biasa juga disebut *data entries* [11]. Pada gambar 2.3.6 terlihat perbedaan *clustered* dan *unclustered index* menggunakan implementasi B+ Tree. Umumnya, pada *clustered* B+ Tree *index*, *data entries* sama dengan *data records*.



Gambar 2.3.6 Ilustrasi Perbedaan Clustered Index dan Unclustered Index

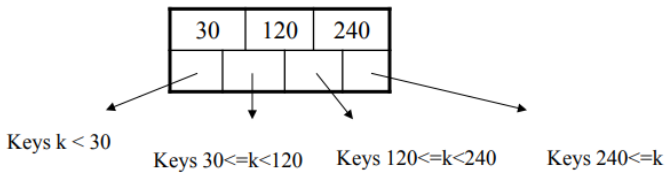
Sumber :
https://courses.cs.washington.edu/courses/cse444/09sp/lectures/lecture15.pdf

III. Penggunaan B+ Tree pada Indeks Basis Data

A. Struktur

B+ Tree pada beberapa aspek merupakan generalisasi dari Pohon Pencarian Biner (*Binary Search Tree*). Perbedaan paling mendasar terletak pada jumlah simpul anak yang dapat ditunjuk oleh simpul B+ Tree lebih banyak dibandingkan BST yang terbatas hanya mempunyai dua anak. Karena Tujuan dari penggunaan B+ Tree adalah untuk meminimalkan akses diska ketika melakukan pencarian *record*, tinggi dari pohon ini haruslah sekecil mungkin. Hal ini dapat dilakukan dengan memperbanyak jumlah kunci pada setiap simpul dalam [12].

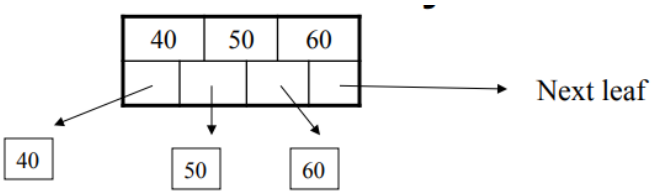
Implementasi simpul pada B+ Tree untuk *indexing* terdiri dari beberapa kunci dan *pointer*. Tiap *pointer* pada simpul dalam menunjuk ke anak.



Gambar 3.1.1 Ilustrasi Simpul Dalam B+ Tree

Sumber : https://courses.cs.washington.edu/courses/cse444/09sp/lectures/lecture15.pdf

Sedikit berbeda dengan simpul daun dimana terdapat satu atau dua *pointer* yang menunjuk ke simpul daun lain.



Gambar 3.1.2 Ilustrasi Simpul Daun B+ Tree

Sumber : https://courses.cs.washington.edu/courses/cse444/09sp/lectures/lecture15.pdf

Hal ini menyebabkan terbentuknya *linked list* (satu *pointer*) atau *doubly linked list* (dua *pointer*) pada simpul-simpul daun. Adanya *list* memfasilitasi operasi pencarian kisaran (*range search* atau *range query*).

Struktur *database index* dibedakan menjadi *logical* dan *physical* seperti yang terlihat pada gambar 3.1.3. Bagian *logical* juga disebut *Index File*, dibagi menjadi simpul akar, simpul cabang, dan simpul daun yang juga disebut d*ata entries*. Sedangkan bagian *physical* adalah *Data File* berupa *actual record* yang tersimpan di dalam diska [13].

B. Implementasi

Actual record maupun *index record* disimpan dan diambil kembali dalam sebuah satuan yang dinamakan *disk blocks* atau *pages* [14].

Di dalam prakteknya, sebuah simpul B+ Tree index berukuran sama dengan sebuah *disk page*. Implementasi satu simpul B+ Tree biasanya mempunyai sekitar 100 hingga 200 simpul anak. Ramakhrishnan dan Gehrke menyatakan bahwa *fill-factor* dari sebuah simpul umumnya sebesar 67%, dengan rata-rata faktor percabangan atau *fanout* sejumlah 133 (tiap simpul cabang mempunyai 133 anak). B+ Tree dengan tinggi 4 dan *fanout* rata-rata 133 dapat digunakan untuk mengindeks sebuah tabel yang mempunyai hingga 300 juta *record* (tepatnya 133^4 = 312,900,700). B+ Tree dengan tinggi 3 dapat digunakan untuk mengindeks sebuah tabel dengan 133^2 = 2,352,637 *record*. Dengan hanya mengakses lima *disk page* (jumlah simpul yang dilewati dari akar hingga adalah tinggi pohon ditambah 1), RDBMS dapat mencari sebuah *record* dari 300 juta *record*. Jika ukuran sebuah *page* pada umumnya sebesar 8Kbytes, dua *level* pertama dari B+ Tree (1 + 133 = 134 *disk page*) hanya sekitar 1Mbyte dan bisa di-*cache* ke dalam memori utama (*cache friendly*). Hal ini menyebabkan RDBMS hanya perlu mengakses tiga *disk pages*, yaitu tiga *level* bawah dari B+ Tree, untuk mencari sebuah *record* dari 300 juta *record* yang ada [12].

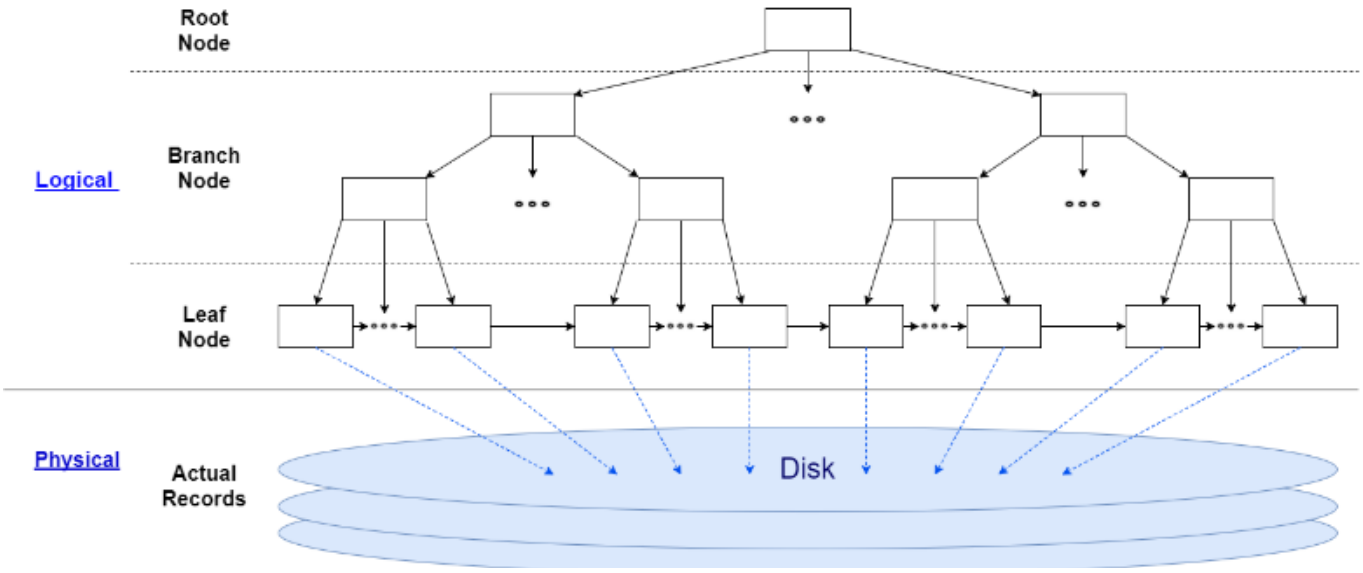
C. Operasi

a.) Pencarian (*Searching*)

Ada dua macam pencarian, pencarian ekuitas (*equity search*) dan pencarian kisaran (*range search*).

Pencarian ekuitas dilakukan dengan menelusuri akar hingga ke daun. Jika *record* yang dicari mempunyai kunci K. Berikut tahapannya.

1. Lakukan *binary search* terhadap nilai *search key* di simpul saat ini (hal demikian dapat dilakukan karena B+ Tree menjaga keterurutan kunci di tiap simpul). Cari K_i dimana $K_i \leq K < K_{i+1}$



Gambar 3.1.3 Ilustrasi Struktur Indeks Basis Data

Sumber : https://use-the-index-luke.com/sql/anatomy/the-tree#sb-log

2. Jika simpul saat ini adalah simpul dalam, telusuri simpul yang berkaitan dengan K_i lalu ulangi tahapan 1.
3. Jika simpul saat ini adalah daun, maka:
    a. Jika K = K_i, *record* ditemukan, kirim *record* yang dicari
    b. Jika tidak, K tidak ditemukan [12].

Pencarian kisaran menggunakan tahapan awal yang sama dengan pencarian ekuitas. *Record* pertama yang dicari adalah batas bawah atau batas atas kisaran, sesuai dengan urutan *search key*. Setelah *record* pertama ditemukan, *record* lainnya dapat ditemukan dengan menelusuri simpul daun yang berbentuk *linked list*, hal ini dilakukan secara sekuensial. Penelurusan dihentikan apabila *search key* melewati batas kisaran lain [12].

b.) Penyisipan (*Insertion*)

Penyisipan pada B+ Tree mirip dengan penyisipan pada struktur data pohon pada umumnya, yaitu *record* baru selalu disisipkan di simpul daun. Penyisipan menjadi rumit ketika simpul daun sudah penuh sehingga diperlukan penambahan simpul daun yang baru [12].. Tahapan penyisipan adalah sebagai berikut

1. Cari posisi yang sesuai dengan kunci baru yang akan disisipkan menggunakan tahapan yang sama seperti di operasi pencarian.
2. Simpul daun L adalah simpul dimana indeks *record* yang baru akan disisipkan
3. Jika L tidak penuh maka *record* baru dialokasikan di *page disk*, kemudian *search key* dan *pointer* baru akan dibentuk.
4. Jika L penuh, dilakukan pembelahan simpul daun. L_{baru} adalah simpul daun baru yang merupakan sepupu dari L. *Search key* yang ada di L bersama *search key* baru didistribusikan secara merata ke L dan L_{baru}. L_{baru} disisipkan ke *linked list* sebagai sepupu L, kemudian *search key* paling kecil L_{baru} disalin dan disisipkan ke P yang merupakan *parent* L (ini dilakukan agar L_{baru} mempunyai *parent* dan menjadi bagian dari pohon). Penyisipan salah satu *search key* ke P mempunyai 2 buah kondisi, yaitu:
    a. Jika P penuh, terjadi pembelahan simpul dalam. Pembelahan simpul dalam sedikit berbeda dengan pembelahan simpul daun. *Search key* P dan *search key* baru tetap didistribusikan secara merata ke P dan P_{baru}, namun, kunci yang berada di tengah akan dipindahkan ke *parent* P. Pembelahan ini dapat terus berlanjut ke arah akar.
    b. Jika kunci ditambahkan ke simpul akar yang penuh, akar dibelah menjadi dua dan kunci tengah akan dijadikan akar yang baru. Ini adalah cara satu-satunya B+ Tree menambah ketinggian [12].

Pembentukan sebuah B+ Tree baru menggunakan penyisipan yang berulang-ulang sangatlah lambat. Terdapat cara untuk mengefisienkan hal tersebut yaitu dengan teknik *bulk loading*. Bulk loading pada sebuah B+ Tree akan membuat *data entries* (simpul daun) terlebih dahulu, kemudian membentuk pohon ke arah atas [15].

c.) Penghapusan (*Deletion*)

Penghapusan pada B+ Tree perlu mempertahankan agar setidaknya tiap simpul berisi setengah penuh. Operasi ini menjadi rumit ketika penghapusan menyebabkan kondisi *underflow* (kurang dari jumlah minimum kunci yang diperbolehkan yaitu sebesar m/2) pada sebuah simpul daun. Ketika *underflow* terjadi, simpul sepupu yang berdekatan diperiksa, jika salah satu simpul tersebut mempunyai jumlah kunci lebih dari m/2 maka beberapa *key* akan diambil dan diberikan ke simpul yang *under-flowing*. Jika kedua simpul sepupu yang berdekatan tersebut juga mempunyai jumlah kunci minimum, akan dilakukan penggabungan simpul [12]. Tahapan menghapus sebuah *record* adalah sebagai berikut.

1. Cari posisi *record* yang akan dihapus dengan menggunakan kunci *record* tersebut. Pencarian ini akan berakhir di simpul daun L.
2. Jika L mempunyai jumlah kunci yang lebih besar dari jumlah minimum kunci (lebih dari m/2 – 1), maka indeks dengan kunci tersebut akan dihapus dan *record* sebenarnya di-dealokasi.
3. Jika L mempunyai jumlah minimum kunci, maka indeks kunci tersebut akan dihapus dan digantikan indeks kunci lain untuk menjaga keterurutan. Untuk mencari index kunci tersebut, simpul sepupu kiri (L_{kiri}) dan simpul sepupu kanan (L_{kanan}) akan diperiksa. Terdapat 3 kondisi:
    a. Jika salah satu L_{kiri} atau L_{kanan} mempunyai kunci dengan jumlah melebihi jumlah minimum kunci, maka beberapa *record* di simpul sepupu ini akan dipindahkan ke L. Ini adalah tindakan heuristik, dilakukan untuk menunda terjadinya *underflow* selanjutnya selama mungkin.
    b. Jika L_{kiri} dan L_{kanan} hanya mempunyai jumlah kunci minimum, maka kunci yang berada di L dipindahkan ke salah satu simpul sepupu kemudian L akan dihapus. Penggabungan ini akan menghapus salah satu kunci yang ada di *parent* L maupun simpul sepupu L, yang juga dapat menyebabkan simpul *parent* menjadi *underflow*. Simpul *parent* atau simpul dalam yang *underflow* ditangani dengan cara yang sama seperti simpul daun.
    c. Jika dua anak dari simpul akar digabung, maka hasil penggabungan simpul ini akan menjadi akar yang baru, ketinggian pohon berkurang [12].

IV. KESIMPULAN

Penggunaan struktur data pohon pada sistem basis data relasional salah satunya terletak pada teknik *indexing*. Jenis

indexing yang umum digunakan ialah indeks terurut dengan *multi-level index*. B+ Tree umum digunakan untuk mengimplementasikan multi-level index sebab struktur data ini *cache* dan *memory friendly* (beberapa *level* atas dapat disimpan ke dalam cache atau memori) dan dapat meminimalkan akses diska (tinggi pohon yang kecil) dengan cara memperbanyak jumlah kunci di setiap simpul. Namun keefektifan indeks dengan B+ Tree dikembalikan lagi kepada pengguna, jika diimplementasikan secara tepat dapat meningkatkan kinerja basis data, sebaliknya jika tidak tepat dapat menurunkan kinerja.

VI. Ucapan Terima Kasih

Pertama penulis mengucapkan puji syukur kehadirat Tuhan Yang Maha Esa, sehingga dapat menyelesaikan penulisan makalah matematika diskrit dengan tepat waktu. Setelah itu penulis mengucapkan terima kasih kepada Bapak Rinaldi Munir selaku dosen IF 2120 Matematika Diskrit yang telah membagikan ilmunya kepada penulis selama satu semester. Kemudian, penulis juga menyampaikan ucapan terima kasih kepada orang tua penulis, keluarga, dan teman-teman yang telah membantu dari segi nonteknis.

Referensi

[1] Munir, R. (2013). Pohon - Bahan kuliah IF2120 matematika diskrit [Online]. Tersedia: http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2013-2014/Pohon%20(2013).pdf diakses tanggal 6 Desember 2018 pukul 19.00.

[2] Binary search tree [Online]. Tersedia: https://www.geeksforgeeks.org/binary-search-tree-data-structure/ diakses tanggal 6 Desember 2018 pukul 20.00.

[3] B-Trees [Online]. Tersedia: https://www.cs.cornell.edu/courses/cs3110/2012sp/recitations/rec25-B-trees/rec25.html diakses tanggal 6 Desember 2018, pukul 21.00.

[4] Comer, D. (1979). Ubiquitous B-Tree. *ACM Computing Surveys*, *11*(2), 121-137.

[5] B+ Tree [Online]. Tersedia: https://www.javatpoint.com/b-plus-tree diakses tanggal 7 Desember 2018 pukul 09.00.

[6] Database Management System (DBMS) [Online]. Tersedia: https://searchsqlserver.techtarget.com/definition/database-management-system diakses tanggal 7 desember 2018 pukul 11.00.

[7] SQL - RDBMS concepts [Online]. Tersedia: https://www.tutorialspoint.com/sql/sql-rdbms-concepts.htm diakses tanggal 7 desember 2018 pukul 14.30.

[8] Altarad, M. NoSQL databases: The definitive guide [Online]. Tersedia: https://blog.pandorafms.org/nosql-databases-the-definitive-guide/ diakses tanggal 7 desember 2018 pukul 14.45.

[9] Indexing in database | Set 1 [Online]. Tersedia: https://www.geeksforgeeks.org/indexing-in-databases-set-1/ diakses tanggal 7 desember 2018 pukul 16.00.

[10] DBMS - Indexing [Online]. Tersedia: https://www.tutorialspoint.com/dbms/dbms_indexing.htm diakses tanggal 7 desember 2018 pukul 17.00.

[11] Introduction to database systems - Lecture 15: Data storage and indexes [Online]. Tersedia: https://courses.cs.washington.edu/courses/cse444/09sp/lectures/lecture15.pdf diakses tanggal 8 desember 2018 pukul 11.30.

[12] B+-Tree indexes [Online]. Tersedia: http://web.csulb.edu/~amonge/classes/common/db/B+TreeIndexes.html diakses tanggal 8 desember 2018 pukul 14.00.

[13] Winand, M. The search tree (B-Tree) makes the index fast [Online]. Tersedia: https://use-the-index-luke.com/sql/anatomy/the-tree diakses tanggal 8 desember 2018 pukul 16.00.

[14] Koutris, P. (2015). Storing data: Disk and files. Tersedia: http://pages.cs.wisc.edu/~paris/cs564-f15/lectures/lecture-09.pdf diakses tanggal 8 desember 2018 pukul 17.00.

[15] Tree-Structured indexes. Tersedia: http://www.csbio.unc.edu/mcmillan/Media/Comp521F10Lecture14.pdf diakses tanggal 9 desember pukul 10.00.

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 9 Desember 2018

Ikraduya Edian
13517106