LIBI-Trie: Modifikasi HAT-Trie untuk DNS Suffix Blocking Yang Efisien

Kiswono Prayogo
Sistem Informasi
UPH Surabaya
Surabaya, Indonesia
kiswono.prayogo@gmail.com

***Abstrak*— Pada sebuah jaringan komputer, umumnya dibutuhkan aplikasi untuk memblokir akses konten internet tertentu. Pemblokiran ini dapat diimplementasikan dengan firewall, DNS poisoning proxy ataupun web filtering proxy. Pemblokiran nama domain dalam sebuah jaringan pada umumnya memblokir juga semua subdomain di dalamnya, misal: pemblokiran terhadap xx.com juga termasuk www.xx.com tetapi tidak pada axx.com. Pemblokiran ini bersifat suffix, yang artinya pencocokan tiap label nama domain dimulai dari label paling belakang. Dalam pemblokiran DNS dibutuhkan struktur data yang efisien dalam hal pencocokan suffix. Pengukuran kecepatan dan penggunaan memori dilakukan untuk memeriksa apakah struktur data cukup efisien. Kriteria efisien adalah perbandingan antara performa (jumlah pencocokan struktur data per detik) dibagi dengan penggunaan memori. Semakin tinggi performa semakin baik, dan semakin kecil penggunaan memori semakin baik pula struktur data tersebut. HAT-Trie merupakan struktur data yang hemat memori dan mampu melakukan exact-matching dengan cepat. Dalam penelitian ini diajukan desain struktur data baru yang mendukung suffix-matching dengan nama LIBI-Trie yang didasarkan pada HAT-Trie dan teknik-teknik kompresi sederhana.**

Kata kunci – DNS Poisoning, HAT-Trie, Trie, Pencocokan Suffix.

I. PENDAHULUAN

Pada sebuah jaringan komputer, umumnya dibutuhkan aplikasi untuk memblokir akses konten internet tertentu. Pemblokiran ini dapat diimplementasikan dengan firewall, DNS *poisoning proxy* ataupun *web filtering* proxy. Domain Name System (DNS) merupakan sistem basis data yang terdistribusi yang tersimpan dalam alat yang terkoneksi dalam sebuah jaringan komputer yang memiliki kegunaan utama sebagai penerjemah antara nama domain dengan alamat numerik (IP). Dengan adanya DNS, pengguna jaringan tidak perlu menghafalkan alamat numerik ketika ingin mengakses sebuah komputer atau website. Proxy merupakan *software* yang berguna untuk mewakilkan *request* service tertentu dari *client* menuju *server*. Sebagai contoh, sebuah web *caching* proxy mampu melakukan *forward request* HTTP dari client menuju server dan mengembalikan *response* dari server kembali pada client, serta melakukan caching agar client lain yang melakukan request yang sama tidak diteruskan pada server tetapi mengambil dari cache.

Cara kerja DNS poisoning adalah dengan memberikan informasi alamat numerik palsu pada sebuah nama domain. Pemblokiran nama *domain* dalam sebuah jaringan pada umumnya memblokir juga semua subdomain di dalamnya, misal: pemblokiran terhadap xx.com juga termasuk www.xx.com tetapi tidak pada axx.com. Pemblokiran ini bersifat suffix, yang artinya pencocokan tiap label nama domain dimulai dari label paling belakang. Setelah melakukan poisoning, biasanya terdapat sebuah website sesuai dengan alamat numerik *poison* yang melakukan pencatatan URL dan informasi client yang melakukan request. Beberapa contoh implementasi DNS poisoning: ChilliSpot (*captive portal*), OpenDNS, NortonDNS dan NawalaProject.

Seiring dengan meningkatnya daftar domain *blacklist* domain pada DNS server, dibutuhkan pula struktur data yang scalable dalam hal *retrieval record* DNS. Penelitian ini melanjutkan penelitian mengenai HAT-trie, tetapi difokuskan pada perbaikan pencarian label nama domain dari posisi belakang (*suffix resolving*). Penelitian HAT-Trie sebelumnya tidak dikhususkan untuk pencarian suffix, dan yang akan dilakukan sekarang adalah meneliti faktor-faktor yang dapat membuat sebuah struktur data dapat melakukan pencarian suffix secara cepat dan hemat memori.

Adapun tujuan dari penelitian ini adalah:

- Mengembangkan struktur data yang efisien dalam hal pencocokan suffix dan penggunaan memori yang spesifik untuk keperluan menyimpan daftar nama domain blacklist.
- Membandingkan performa penggunaan RAM serta durasi insertion dan retrieval berbagai struktur data umum dalam implementasinya untuk penyimpanan nama domain.

Sistematika pembahasan dalam paper ini adalah pendahuluan, metodologi penelitian, desain HAT-Trie dan LIBI-Trie, hasil dan pembahasan serta kesimpulan.

II. METODOLOGI PENELITIAN

Penelitian dilakukan dengan metode kuantitatif. Dataset daftar domain blacklist yang diujiakan diambil dari sumber-sumber berikut ini:

- IANA's TLD, daftar top level domain. http://data.iana.org/TLD/tlds-alpha-by-domain.txt
- MVPs.org, daftar nama domain yang mengandung spyware, iklan maupun trojan. http://winhelp2002.mvps.org/hosts.zip
- URLBlackList.com, daftar blacklist komersial untuk domain dan URL (Dansguardian). http://urlblacklist.com/cgi-bin/commercialdownload.pl ?type=download&file=bigblacklist
- Full Trust Positif DepKomInfo. http://trustpositif.depkominfo.go.id /files/index.php?download=blacklist%2F*%2 Fdomains&share=11. Dimana * adalah porn, kajian, pengaduan, redirector dan whitelist

Daftar nama domain yang menjadi blacklist akan digabungkan menjadi 1 file 37MB (2.1 juta nama domain). Terdapat 68.4 ribu nama domain lain (133 KB) yang tidak terdapat pada daftar blacklist yang digunakan sebagai pengujian ketidakberadaan nama domain dalam struktur data.

semua pengujian dialakukan dengan menggunakan spesifikasi komputer dengan spesifikasi processor: AMD Phenom X9850 Quad-Core 2.5 GHz; RAM 4 x VGen 2 GB PC-6400; Kernel: Linux 2.6.38.10-server (x86_64) s/d Linux 3.2.0.26-generic (x86_64); Compiler: GCC 4.5.2 (Ubuntu/Linaro 4.5.2-8ubuntu4), GCC 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5), CLang 3.1-3eudoxos1 (branches/release_31) dan compile flag: g++ -c -m64 -pipe -O2 -Wall -W, clang++ -c -pipe -O2 -D_REENTRANT -Wall –W.

Pengukuran RAM dilakukan dengan mengambil informasi dari kernel Linux yang tersedia pada file teks "/proc/self/stat". Pada kernel Linux, penggunaan memori fisik dikenal dengan *Resident Set Size* (RSS). Dalam pengujian, file swap (*pagefile* atau lebih dikenal dengan *virtual memory*) dinon-aktifkan sehingga seluruh program yang berjalan hanya menggunakan memori fisik tanpa swapping (karena swapping dapat mengurangi performa).

Pengukuran kecepatan dilakukan dengan melakukan operasi insert, search (data yang ada secara linear, data yang tidak ada, data yang ada secara acak) seara berulang hingga melebihi batas waktu tertentu. Durasi yang diperhitungkan adalah total durasi yang terpakai (dalam millisecond) dibagi dengan jumlah total data yang berhasil diproses. Pengukuran durasi mempergunakan class QElapsedTimer dari Library Qt.

III. HAT-TRIE DAN LIBI-TRIE

Struktur data berbasis trie mampu menyimpan string dengan cepat tetapi boros memori. Trie sering digunakan untuk kompresi teks dan kamus. Burst-Trie yang ditemukan oleh Zobel Heinz dan Williams pada tahun 2002 merupakan salah satu teknik mengurangi penggunaan memori trie. Burst trie mampu mengurangi penggunaan memori hingga 80% dengan sedikit atau tanpa pengurangan performa. Pendekatan yang dilakukan Burst-Trie adalah dengan secara selektif mengumpulkan string-string yang memiliki prefix (awalan) sama dalam sebuah bucket, dimana apabila bucket telah penuh akan dipecah menjadi node trie baru.[1]

Sekalipun cepat, burst-trie tidak *cache-conscious* (tidak memanfaatkan cache dengan baik). Seperti struktur data memori pada umumnya, pada prakteknya setiap terjadi random access pada RAM (atau *main memory*) membutuhkan ratusan clock cycle processor. Oleh karena itu prosesor mengimplementasikan sebuah hirarki dengan kapasitas kecil tetapi cepat antara CPU dan RAM yang disebut cache. Apabila data tidak terdapat pada cache, maka RAM diakses, hal ini disebut cache-miss. Mendayagunakan cache dengan maksimal merupakan hal yang sangat penting untuk mencegah bottleneck (kemacetan). Sekalipun programmer tidak memiliki kontrol terhadap cache, penggunaan cache tetap dapat dimaksimalkan dengan desain yang mendukung peningkatan *temporal locality* dan *spatial access locality*. Temporal locality adalah penggunaan resource secara berulang dalam durasi rendah.

[1] Heinz, S., Zobel, J. & Williams, H. E. *Burst tries: A fast, efficient data structure for string keys*, ACM trans. on Information Systems 20(2), pp. 192–223.

Spatial locality adalah penggunaan data yang tersimpan dalam jarak yang relatif dekat.[2]

Sekalipun boros memori, tetapi trie dapat dibuat menjadi cache-conscious. Burst-trie menyimpan bucket sebagai linked-list, dimana linked-list telah dikenal tidak efisien dalam memanfaatkan cache, karena alamat node berikutnya baru diketahui setelah node sebelumnya selesai diproses. Masalah ini dikenal dengan istilah *pointer chasing*. Hal ini dapat mengganggu kerja dari *hardware prefetchers* (bagian yang bertugas mentransfer data dari main memori ke dalam cache sebelum data tersebut dibutuhkan). Untuk mengatasi hal ini dibutuhkan struktur data yang cache-conscious seperti dynamic array. Array memiliki akses yang searah yang dapat dideteksi dan dieksploitasi oleh hardware prefetchers.[3]

Burst Trie dengan sukses mengurangi jumlah node trie dengan menggabungkan sisa string yang memiliki awalan sama ke dalam sebuah bucket. Bucket dipresentasikan sebagai linked-list dengan move-to-front on access, yang apabila jumlahnya telah melebihi treshold tertentu akan pecah menjadi sebuah node baru. Linked-list bukanlah struktur data yang cache-conscious, sehingga performa tidak akan membaik apabila diuji dalam prosesor modern yang berorientasi pada cache.

HAT-Trie merupakan pengembangan dari Burst-Trie yang mengganti bucket burst-trie dengan cache-conscious hash table. Terdapat 2 jenis HAT-Trie yaitu pure dan hybrid, dimana pada hybrid, sebuah bucket dapat ditunjuk oleh lebih dari satu indeks. HAT-Trie mampu bekerja hingga 80% lebih cepat dan 70% lebih hemat memori dibandingkan Burst Trie, dan dalam kasus tertentu mampu bekerja lebih baik atau mendekati cache-consious array hash yang merupakan struktur data terbaik untuk penyimpanan string yang tidak terurut.[4]

Terdapat 2 pendekatan yang menjadi dasar pembuatan HAT-Trie yaitu pure dan hybrid. Variasi terdapat pada bagaimana bucket dibuat dan dipecah. Pada pure HAT-Trie setiap bucket hanya memiliki sebuah prefix, sehingga hanya sisa string selain prefix yang disimpan. Pada hybrid HAT-Trie, karakter terakhir dari prefix juga disimpan dalam bucket yang sama, sehingga sebuah bucket dapat memiliki lebih dari 1 parent.

HAT-Trie dibuat dan dicari secara top-down, tiap terjadi tranversal akan mengurangi prefix. String yang pendek dapat disimpan oleh sebuah flag end-of-string pada pure bucket atau trie node. Alternatif lain dalam penyimpanannya adalah dengan mempergunakan struktur data tambahan. Bucket dapat dimodifikasi untuk menyimpan data field tertentu.

HAT-Trie dimulai dengan sebuah hybrid bucket yang diisi hingga penuh. Bucket tidak menyimpan data yang kembar, sehingga penambahan data hanya terjadi ketika proses pencarian tidak menghasilkan data. Burst (pemecahan) atau split (pembagian) bucket terjadi ketika bucket penuh bergantung pada tipe HAT-Trie. Burst membuat satu atau lebih trie baru bergantung populasi data dengan membuang sebuah karakter awal.

Pada Hybrid HAT-Trie melakukan split menjadi 2 berdasarkan algoritma B-Trie. Ketika split, pure bucket diubah menjadi hybrid bucket dengan membuat node trie yang mengarah ke hybrid bucket tersebut, lalu dilanjutkan dengan proses split untuk hybrid bucket. Untuk melakukan split hybrid bucket, dilakukan pencarian titik pemotongan yang sedapat mungkin mendistribusikan secara rata. Cara mencari titik potong adalah dengan menghitung frekuensi karakter pertama dari tiap data dalam bucket, jumlah tersebut dijelajahi secara urut dan dihitung dengan rasio jumlah string yang dipindah dibagi dengan string yang tidak dipindah harus mencapai minimal 0.75. Rasio tersebut dipilih agar penggunaan memori menjadi efisien untuk B-Tree. Frekuensi karakter yang mencapai rasio tersebut dipilih sebagai split-point.

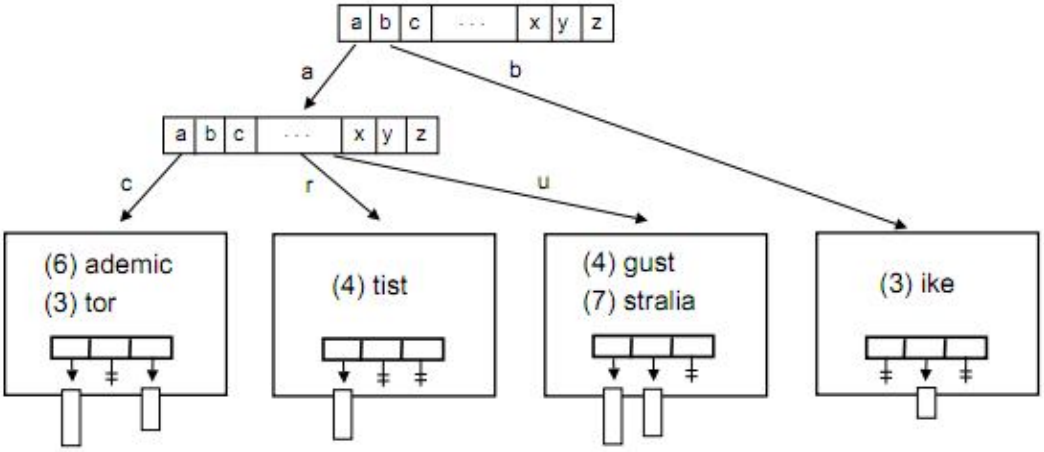
Apabila tidak terdapat string yang tersisa, frekuensi karakter yang terakhir diakses dipilih menjadi split point, yang dapat menyebabkan pembuatan bucket kosong. Kedua bucket baru yang dibuat dapat diklasifikasikan sebagai pure atau hybrid, bergantung pada split-point dan distribusi string. Sebuah string yang hanya terdiri dari 1 karakter dapat dihapus beserta bucket kosong dan node trie pointer parentnya diarahkan ke null. Pemecahan berhenti apabila kedua bucket tidak melebihi kapasitas bucket, apabila tidak, pemecahan akan terus berlangsung.

---

[2] Nikolas Askitis dan Ranjan Sinha, *HAT-trie: A Cache-conscious Trie-based Data Structure for Strings.*, 2007., p. 3

[3] A. R. Lebeck. *Cache conscious programming in undergraduate computer science.*, In Proc. SIGCSE Technical Symp on Computer Science Education, 1999. p. 247-251

[4] Nikolas Askitis dan Ranjan Sinha, *HAT-trie: A Cache-conscious Trie-based Data Structure for Strings.*, 2007., p. 3



Gambar 1.
Pure HAT-Trie

Dari sisi implementasi array hash, bucket merupakan array yang berisi n+1 pointer, yang kosong ataupun menunjuk pada slot array dinamis yang berisi length-encoded string. Pointer pertama digunakan untuk menyimpan informasi jenis bucket atau range karakter, penanda akhir karakter dan jumlah string yang disimpan dalam bucket.

Untuk memasukkan atau mencari data dalam bucket, dilakukan hashing dari sisa string menggunakan fungsi hash fast bitwise hash function. Setelah slot diketahui, dilakukan pencarian per karakter yang apabila tidak cocok akan langsung loncat pada string berikutnya pada array. Penambahan data hanya terjadi apabila string tidak ditemukan atau slot kosong. Penambahan data dilakukan dengan memperbesar atau membuat array baru yang berukuran tetap sesuai kebutuhan lalu menambahkan sisa string baru ke paling belakang, operasi ini cache-efficient.

Pure HAT-Trie mampu mencapai kecepatan array hash dan penggunaan memori yang lebih sedikit, dimana hybrid HAT-Trie mampu lebih hemat dibandingkan struktur data apapun tetapi tidak secepat pure HAT-Trie. Penelitian HAT-Trie memberikan kesimpulan bahwa HAT-Trie mampu mencapai kecepatan dan efisiensi hash table sekaligus mampu menyimpan string dalam keadaan terurut. Kesimpulan lainnya menunjukkan bahwa array dinamis dalam desain struktur data lebih efektif dan efisien dibandingkan struktur data yang menggunakan banyak pointer (pointer-intesinve).

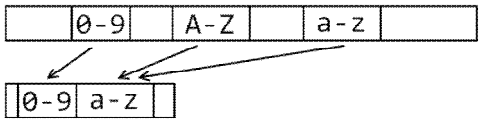
Pada bagian ini akan dibahas mengenai rencana perubahan terhadap HAT-Trie. Terdapat 7 rencana modifikasi yang meliputi perubahan trie node, hash node dan leaf.

3.1. Character Packing

Packing merupakan metode kompresi sederhana untuk mengurangi penggunaan bit dari 2^N suatu data menjadi 2^{N-M}. Packing mungkin untuk dilakukan karena jumlah karakter yang valid pada DNS adalah 38, sehingga lebar node dari trie yang dibutuhkan dapat dikurangi. Kompresi dan dekompresi ini dapat diimplementasikan dengan 2 buah array. Character packing memiliki tujuan mengurangi semua pointer kosong yang akan dibuat di trie node, dengan harapan penggunaan memori dapat dikurangi, dan proses dapat berjalan lebih cepat karena lokasi memori yang berdekatan.

Karakter set DNS yang hanya memiliki total 38 jenis karakter yang terdiri dari 26 karakter huruf, 10 karakter angka dan 2 karakter tanda baca yaitu “-“ dan “.”memperbolehkan reduksi ukuran trie untuk ASCII yaitu 128 elemen pointer pada HAT-Trie menjadi 40 elemen pointer pada LIBI-Trie. Elemen pertama digunakan sebagai header dari Trie. Mapping ini dapat diimplementasikan dengan 2 buah array global untuk encoding dan decoding yang awalnya berisi angka 0. Array encoding diisi untuk tiap indeks huruf yang dipakai dengan indeks yang selalu meningkat. Array decoding diisi untuk tiap indeks yang dipakai dengan huruf bersangkutan seperti pada array encoding. Sebagai contoh: encoding[‘a’] = 1, maka decoding[1] = ‘a’, berikutnya encoding[‘b’] = 2, decoding[2] = ‘b’, dan seterusnya. Dengan

menggunakan kedua array decoding dan encoding tersebut, maka worst case dari operasi packing dan unpacking untuk sebuah karakter adalah O(1).



Gambar 2
Character Packing

Character packing dapat juga diimplementasikan untuk himpunan karakter lainnya, tidak hanya pada himpunan karakter DNS. Sebagai contoh dalam autocomplete case insensitive pada text editor dapat juga menggunakan character packing. Berikut adalah tabel sebagian hasil pengkodean dari karakter DNS menjadi indeks di dalam array-trie dengan character packing:

3.2. Hash Function Modification

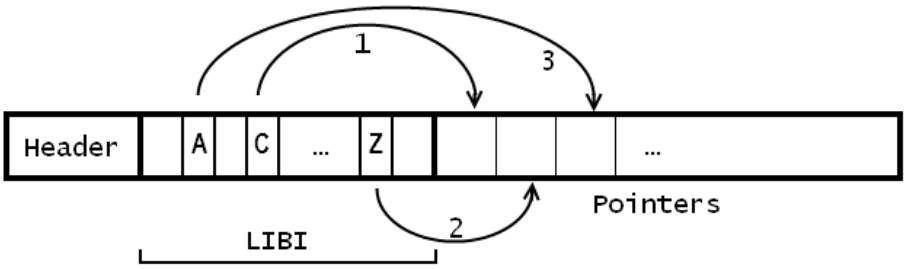
Fungsi hash pada string pada umumnya bergantung pada panjang string. Untuk meningkatkan performa, hashing yang baru hanya mendayagunakan 3 karakter pertama. Tiga karakter pertama dipilih agar semua leaf yang memiliki 3 karakter pertama sama pasti masuk ke dalam sebuah slot yang sama, sehingga peluang kompresi leaf (prefix compression, lihat bagian 5.1.7) menjadi lebih besar. Fungsi hash tersebut juga dimodifikasi sedemikian rupa sehingga menghasilkan nilai hash 1 hingga 255 untuk kepentingkan kompresi node yang akan dijelaskan pada bagian berikutnya (bagian 3.3). Fungsi hash dibuat sedemikan rupa sehingga frekuensi distribusi tertinggi dan terrendah yang dihasilkan memiliki margin yang kecil (selisih frekuensi terbanyak dan tersedikit yang mungkin masuk dalam suatu node), yang memiliki arti bahwa fungsi hash menghasilkan nilai hash yang cukup merata.

Terdapat sebuah ketentuan tambahan agar pencocokan suffix dapat berjalan dengan benar, yaitu fungsi hash harus menghasilkan nilai yang sama untuk string yang berawalan sama sebelum titik ataupun akhir dari nama domain. Sebagai contoh, string “a.x”, “a.y” dan “a” harus menghasilkan nilai yang sama. Apabila hasil “a.x” berbeda dengan “a”, maka pencarian nama domain akan dilanjutkan pada slot hashtable yang salah, sehingga pencocokan suffix pasti gagal.

3.3. Node Compression

Sebuah node trie yang telah melalui proses packing berisi 40 elemen pointer dengan ukuran word size komputer yaitu 64-bit (8-byte) pada komputer pengujian. Sebuah node hash berisi 256 elemen pointer, berbeda dengan HAT-Trie yang menggunakan 512 elemen, dimana elemen pertama berfungsi sebagai header. Angka 256 dipilih untuk alasan kompresi LIBI, karena 1 byte mampu menyimpan angka dari 0 hingga 255, sedangkan apabila lebih, tidak dapat disimpan dalam 1 byte. Pemborosan memori terjadi apabila slot node yang terpakai hanya sedikit, misal: pada node trie yang terpakai hanya huruf ‘a’, ‘c’, ‘f’ dan ‘z’, atau misal pada node hash yang terpakai hanya slot 1 dan 3 (karena jumlah data yang masuk masih sedikit). Pemborosan memori ini dapat diatasi dengan melakukan kompresi pada node (node compression) yang memiliki spesifikasi:

- Compressed node, yaitu node memiliki array bertipe byte (yang penulis istilahkan dengan nama “local immediate block index” atau LIBI) berukuran 40 untuk trie node (5 word), dan 256 untuk hash node (32 word). Node tersebut memiliki 1 word untuk header dan sejumlah word (1 word adalah 64-bit pada komputer pengujian). Ukuran total compressed node dibulatkan ke 2^N word terdekat. Array byte indeks lokal berfungsi sama seperti array encoding pada character packing, yaitu menyimpan lokasi indeks pointer. Sebagai contoh apabila dalam sebuah node hash telah dimasukkan nilai hashing 7, 13 dan 74, maka array indeksLokal[7] = 1, indeksLokal[13] = 2 dan indeksLokal[74] = 3, dimana angka 1 hingga 3 menunjukkan indeks pointer pada node bersangkutan yang akan terus meningkat, yaitu pointer pertama hingga ketiga. Berikut ini adalah contoh gambar untuk Compressed Trie Node, apabila urutan memasukkannya adalah ‘C’, ‘Z’ lalu ‘A’:



Gambar 3
Local Immediate Block Index

- Full Trie Node (TrieFull) membutuhkan 40 word, compressed trie node membutuhkan 32 word untuk Medium Trie Node (TrieMedium) dan 16 word untuk Small Trie Node (TrieSmall). TrieMedium mampu menyimpan 26 pointer (yang didapat dari 32 dikurangi 1 word header dan 5 word array indeks lokal), TrieSmall mampu menyimpan 10 pointer.
- Full Hash Node (HashFull) membutuhkan 256 word, compressed hash node membutuhkan 128 word untuk Medium Hash Node (HashMedium) dan 63 word untuk Small Hash Node (HashSmall). HashMedium mampu menyimpan 95 pointer (yang didapat dari 128 dikurangi 1 word dan 32 word array lokal indeks), sedangkan HashSmall mampu menyimpan 31 pointer.

Node trie dapat berkembang dari TrieSmall menjadi TrieMedium, lalu menjadi TrieFull, bergantung pada indeks yang terisi pada trie. Node hash dapat berkembang dari HashSmall menjadi HashMedium, lalu menjadi HashFull, bergantung pada jumlah indeks hash yang yang terisi. HashFull dapat berubah (burst) menjadi salah satu node trie apabila salah satu leaf-nya telah melebihi 250 substring, dimana angka tersebut dipilih agar tidak melebihi batas maksimum string di dalam sebuah leaf.

3.4. Leaf Slot for Hash Node

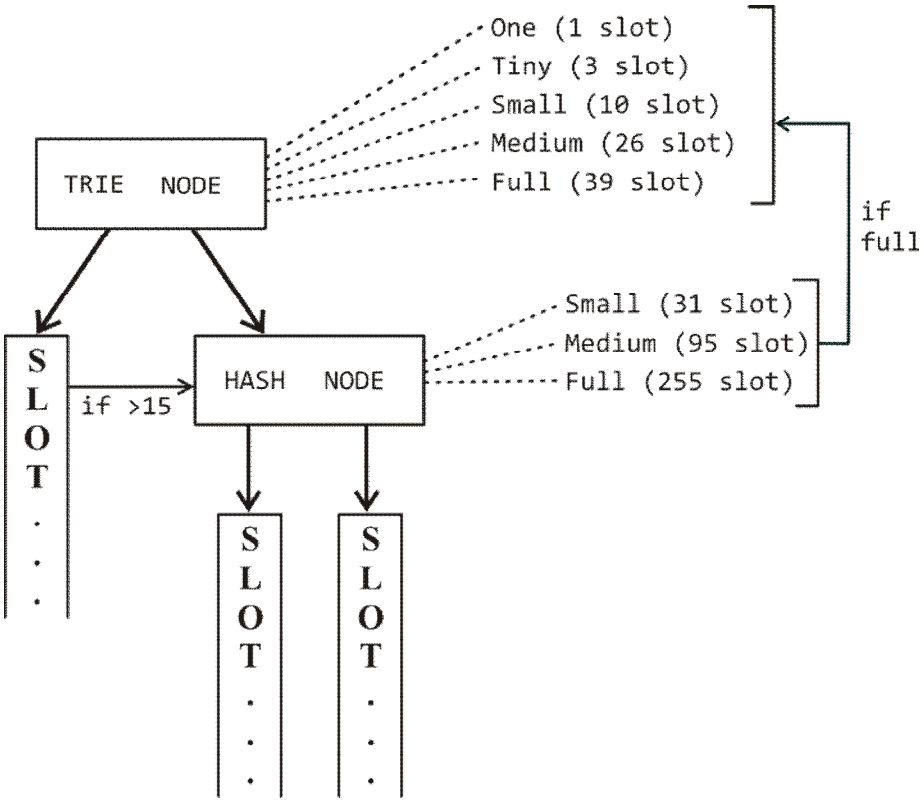
Leaf Slot for Hash Node artinya node hash diganti dengan slot leaf (HashLeaf) apabila jumlah data pada slot hash node masih sedikit (lebih kecil dari 32, angka 32 didapat dari jumlah collision rata-rata yang disarankan oleh pembuat implementasi HAT-Trie yaitu 16384 elemen dibagi dengan 512 jumlah slot hash HAT-Trie). Apabila HashLeaf telah penuh akan digantikan dengan HashSmall sesuai spesifikasi pada bagian sebelumnya. Dalam implementasinya angka yang dipilih adalah 30, karena jumlah maksimal slot HashSmall adalah 31.

3.5. Simple List for Trie Node

Simple List for Trie Node artinya node trie diganti dengan list sederhana (TrieOne atau TrieTiny) apabila indeks yang terisi hanya 1 hingga 3 (angka 1 dan 3 didapat dari 2^1 dan 2^2 word yang telah dikurangi 1 word header). Berbeda dengan node compression pada bagian sebelumnya, karena jumlahnya hanya sedikit, indeks lokal disimpan pada header dan bukan berupa array lengkap. Worst case untuk pencarian pada simple list ini adalah konstan O(1) karena jumlah list maksimal 3. Ketika indeks yang terisi telah melebihi batas, maka TrieOne berubah menjadi TrieTiny, dan berubah menjadi TrieSmall.

TrieTiny memiliki 3 elemen, yaitu mid (tengah), lo (kiri) dan hi (kanan). Urutan memasukkan dimulai dari tengah, lalu kiri, lalu kanan, dan selalu dalam keadaan terurut secara nilai yaitu kiri, tengah dan kanan. Sebagai contoh apabila huruf yang dimasukkan adalah ‘C’, ‘Z’ lalu ‘A’, maka pertama kali ‘C’ masuk ke tengah, lalu ‘C’ berpindah ke kiri dan ‘Z’ masuk ke tengah, dan pada langkah terakhir ‘Z’ berpindah ke kanan, ‘C’ pindah ke tengah dan ‘Z’ masuk ke kiri. Aturan ini dimaksudkan agar jumlah komparasi sewaktu pencarian nilai di dalam node maksimal adalah 2 karena pencarian selalu dimulai dari nilai tengah. Dalam contoh di atas, apabila huruf ‘D’ yang dicari, maka akan dibandingkan dengan nilai tengah yaitu ‘C’, karena ‘D’ lebih besar dari ‘C’ maka mencari ke nilai kanan, karena nilai kanan yaitu ‘Z’ berbeda dengan ‘D’ maka tidak ditemukan.

LIBI-Trie pada gambar berikut ini merupakan desain modifikasi dari HAT-Trie. LIBI-Trie sendiri tidak harus berbasis pada HAT-Trie, LIBI-Trie dapat diimplementasikan tanpa hashtable, yang bentuknya akan menyerupai Burst Trie.



Gambar 4
Desain Libi Trie

3.6. Sorted Leaf Slot

Bucket pada HAT-Trie diimplementasikan dengan menggunakan array dinamis[5]. Pada saat terjadi collision ataupun pencarian, membutuhkan worst case O(n). Hal ini dapat diperbaiki dengan mengurutkan string dalam collision bucket (array dinamis), sehingga pencarian suatu string yang pasti tidak tersimpan pada struktur data dapat dipercepat. Pencarian linear untuk data terurut dan data tidak terurut memiliki worst case yang sama, yaitu O(n), data terurut dapat bekerja lebih baik data yang dicari selalu bernilai lebih kecil dari data terakhir (apabila pengurutan secara ascending). Pengurutan juga dilakukan untuk mendukung kompresi leaf (bagian 3.7).

3.7. Prefix Compression

Penggunaan string dalam bucket dapat dikurangi lebih lanjut dengan prefix compression pada bucket, yang dikarenakan data dalam bucket selalu memiliki prefix yang sama, terpacking dan dalam keadaan terurut, bit pertama dapat digunakan sebagai kode kompresi. Sebagai contoh pada HAT-Trie, 4 data string “apple”, “application”, “apples” dan “abbys” akan disimpan berdasarkan urutan insert menjadi “[5]apple[10]application[6]apples[5]abbys”, tetapi pada LIBI-Trie string-string tersebut akan diurutkan secara descending dan dilakukan prefix compression menjadi “[10]application[4,2]es[5,0][5]abbys”. Kompresi hanya terjadi apabila 3 atau lebih karakter pertama memiliki kesamaan, karena fungsi hashing yang telah dimodifikasi, maka 3 karakter pertama dari sebuah string akan selalu masuk ke dalam leaf yang sama.

IV. HASIL DAN PEMBAHASAN

Terdapat 2 jenis LibiTrie yang diujikan, yaitu:

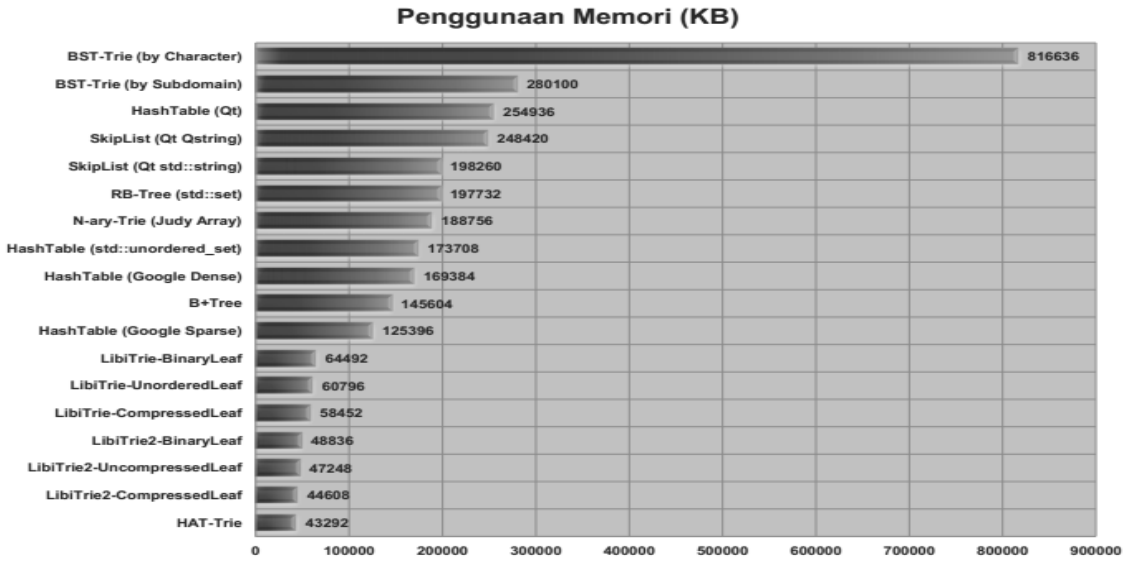
- LibiTrie, yaitu modifikasi HAT-Trie yang sesuai spesifikasi yang tertuang pada bab 3.. Perubahan dari HashLeaf menuju HashSmall dilakukan apabila jumlah data Leaf telah

[5] Nikolas Askitis dan Ranjan Sinha, HAT-trie: A Cache-conscious Trie-based Data Structure for Strings., 2007., p. 3

melebihi 30 data, dan HashNode berubah menjadi TrieNode ketika salah satu SlotLeaf-nya telah melebihi 250 string.

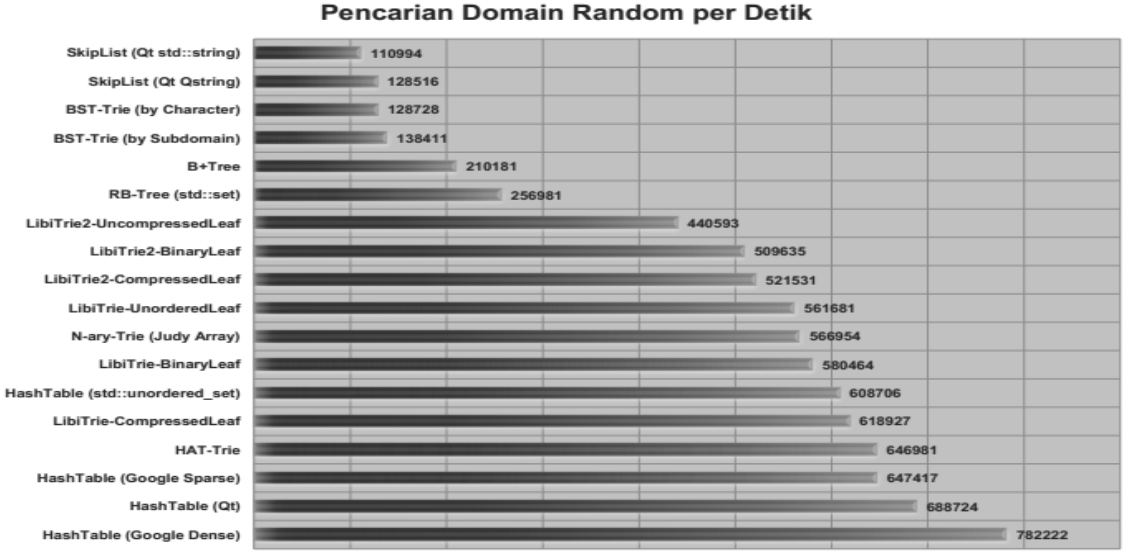
- LibiTrie2, yaitu modifikasi HAT-Trie tanpa hash-table, sehingga desainnya lebih dekat pada Array-Trie dan tetap mendukung suffix-matching. Perubahan Leaf menuju TrieNode terjadi ketika jumlah data pada Leaf telah melebihi 200 string.

Berikut ini adalah tabel hasil pengujian dan perbandingannya dengan HAT-Trie asli yang hanya mendukung exact-matching.



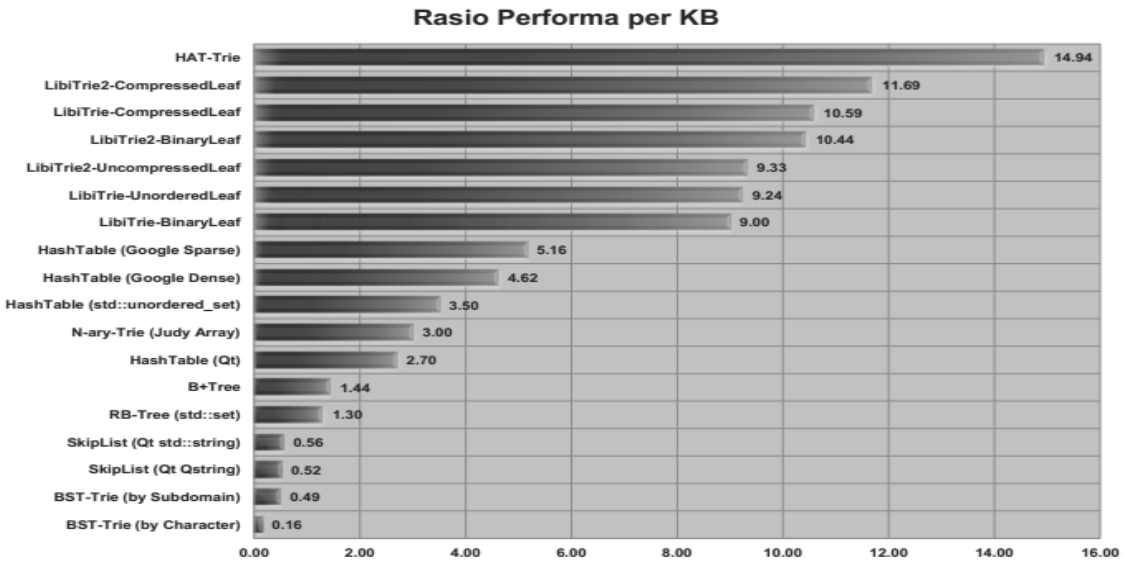
Gambar 5
Grafik Perbandingan Penggunaan Memori

Pada grafik (Gambar 5) dapat dilihat bahwa penggunaan memori HAT-Trie dan LIBI-Trie berkisar antara 43 hingga 65 MB, sedangkan struktur data lainnya berkisar antara 125 hingga 816 MB.



Gambar 6
Grafik Perbandingan Kecepatan PencarianAcak

Pada grafik (Gambar 6) dapat dilihat bahwa 3 posisi teratas untuk performa terbaik diisi oleh hashtable, diikuti oleh HAT-Trie dan LIBI-Trie.



Gambar 7
Grafik Perbandiangan Rasio Kecepatan dan Penggunaan Memori

Pada grafik (Gambar 7) dapat disimpulkan bahwa performa HAT-Trie dan LIBI-Trie sebanding dengan penggunaan memorinya.

Tabel 1
Hasil Pengujian

Nama Struktur Data	Random Domain Search / Detik	Penggunaan Memori (KB)	Perf / KB
BST-Trie (by Character)	128728	816636	0.16
BST-Trie (by Subdomain)	138411	280100	0.49
SkipList (Qt Qstring)	128516	248420	0.52
SkipList (Qt std::string)	110994	198260	0.56
RB-Tree (std::set)	256981	197732	1.30
B+Tree	210181	145604	1.44
HashTable (Qt)	688724	254936	2.70
N-ary-Trie (Judy Array)	566954	188756	3.00
HashTable (std::unordered_set)	608706	173708	3.50
HashTable (Google Dense)	782222	169384	4.62
HashTable (Google Sparse)	647417	125396	5.16
LibiTrie-BinaryLeaf	580464	64492	9.00
LibiTrie-UnorderedLeaf	561681	60796	9.24
LibiTrie2-UncompressedLeaf	440593	47248	9.33
LibiTrie2-BinaryLeaf	509635	48836	10.44
LibiTrie-CompressedLeaf	618927	58452	10.59
LibiTrie2-CompressedLeaf	521531	44608	11.69
HAT-Trie	646981	43292	14.94

V. KESIMPULAN

Kesimpulan yang penulis dapatkan selama pembuatan penelitian ini adalah:

- HAT-Trie merupakan struktur data terbaik dalam hal exact-matching dan penggunaan memori diantara semua struktur data yang diujikan.
- Dense_hash Google merupakan struktur data tercepat dalam hal pencarian string secara terurut dari dataset, diikuti oleh HAT-Trie dan Hashtable Qt.
- Judy_map HP yang disusun sebagai trie label terkompresi merupakan struktur data tercepat dalam hal mengetahui keberadaan subdomain yang tidak tersimpan dalam struktur data, diikuti oleh Dense_hash Google dan Hashtable Qt.
- Hashtable merupakan struktur data tercepat dalam pencarian data secara acak dari dataset, diikuti HAT-Trie.
- HAT-Trie merupakan struktur data tercepat dalam hal insertion data dari dataset.
- HAT-Trie merupakan struktur data terhemat dalam hal penyimpanan data.

- Modifikasi HAT-Trie dengan menambahkan LIBI ternyata tidak mampu mengurangi penggunaan memori, karena bottleneck penggunaan memori terbesar HAT-Trie terletak pada Leaf, bukan pada Node.
- Modifikasi Leaf pada LIBI-Trie dengan menambahkan kompresi leaf mampu mengurangi penggunaan memori dan mempercepat pencarian data dari dataset.
- Dengan membuang Hashtable dari LIBI-Trie mampu mengurangi penggunaan memori hingga 1/3-nya tetapi mengorbankan performa pencarian.
- Desain LIBI-Trie lebih cepat dibandingkan struktur data umum selain Hashtable, N-ary-Trie dan HAT-Trie dalam hal pencarian data secara acak.
- Desain LIBI-Trie lebih hemat memori dibandingkan struktur data umum selain HAT-Trie.
- Desain LIBI-Trie tercepat 4.53% lebih lambat dibandingkan HAT-Trie dan Desain LIBI-Trie terhemat 3.04% lebih boros memory dibandingkan HAT-Trie untuk 2.1 juta data pengujian.

Penelitian dapat dilanjutkan untuk aplikasi dalam bidang lainnya seperti parser dan autocomplete untuk IDE, penyimpanan data asosiatif.

REFERENSI

[1] "A. R. Lebeck. 1999. Cache conscious programming in undergraduate computer science., In Proc. SIGCSE Technical Symp on Computer Science Education.

[2] Askitis, Nikolas., 2007. Efficient Data Structures for Cache Architectures.

[3] Askitis, Nikolas. and Sinha, Ranjan. 2007. HAT-trie: A Cache-conscious Trie-based Data Structure for Strings. http://crpit.com/confpapers/CRPITV62Askitis.pdf

[4] Askitis, N. and Zobel, J. Januari 2011. Redesigning The String Hash Table, Burst Trie, and BST To Exploit Cache. http://www.naskitis.com/naskitis-acmjea10.pdf

[5] Clement, Julien, Philippe Flajolet and Brigitte Valle. 1998. The Analysis of Hybrid Trie Structures. http://algo.inria.fr/flajolet/Publications/ClFlVa98.pdf

[6] Heinz, S., Zobel, J., and Williams, H. E. 2002. Burst Tries: A Fast, Efficient Data Structure for String Keys. ACM trans. on Information Systems 20(2). http://www.csse.unimelb.edu.au/~jz/fulltext/acmtois02.pdf

[7] Knuth, Donald. 1998.The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd Edition. Addison-Wesley.

[8] Michael T. Goodrich, Roberto Tamassia dan David Mount. 2011. Data Structures and Algorithms in C++., 2nd Edition., John Wiley & Sons, Inc.

[9] Monge , Alvaro. B+-Tree Indexes. More efficient retrieval of records via table indexes. http://www.cecs.csulb.edu/~monge/classes/share/B+Tree Indexes.html (diakses tanggal 28 Mei 2012)

[10] Nokia Corporation. Qt Reference Documentation., http://doc-snapshot.qt-project.org/5.0/qmap.html (diakses tanggal 28 Mei 2012)

[11] Pete Becker, 2010. C++ Unordered Associative Container., Working Draft, Standard for Programming Language C++.

[12] Silverstein, Alan. 2002. Judy IV Shop Manual.

[13] Silverstein, Craig, Sparsehash, An extremely memory-efficient hash_map implementation., http://code.google.com/p/sparsehash/ (diakses tanggal 28 Mei 2012)

[14] Sinha, R. and Wirth, A. 2010. Engineering Burstsort: Toward Fast In-Place String Sorting.

[15] Spiegel, Michael. Mei 2011. Cache-conscious Concurrent Data Structures. http://www.cs.virginia.edu/ ~ms6ep/publications/michael-spiegel-dissertation.pdf

[16] Sukono, Pandu A. Juni 2009. DNS Prinsip Kerja Beserta Contohnya. http://te.ugm.ac.id/~risanuri/v01/wp-content/uploads/2009/06/DNS%20Prinsip%20Kerja%20Beserta%20Contohnya.pdf