PENGEMBANGAN ALGORITMA FAST INVERSION DALAM MEMBENTUK *INVERTED FILE* UNTUK TEXT RETRIEVAL DENGAN DATA SKALA BESAR

Derwin Suhartono

Computer Science Department, School of Computer Science, Binus University
Jln. K. H. Syahdan No. 9 Palmerah Jakarta Barat 11480
dsuhartono@binus.edu

ABSTRACT

The rapid development of information systems generates new needs for indexing and retrieval of various kinds of media. The need for documents in the form of multimedia is increasing currently. Thus, the need to store or retrieve now becomes a primary problem. The multimedia type commonly used is text types, as widely seen as the main option in the search engines like Yahoo, Google or others. Essentially, search does not just want to get results, but also a more efficient process. For the purposes of indexing and retrieval, inverted file is used to provide faster results. However, there will be a problem if the making of an inverted file is related to a large amount of data. This study describes an algorithm called Fast Inversion as the development of base inverted file making method to address the needs related to the amount of data.

Keywords: *efficient search, inverted file, Fast Inversion algorithm*

ABSTRAK

Perkembangan yang sangat cepat dari sistem informasi menghasilkan suatu keperluan yang baru untuk indexing dan retrieval dari berbagai macam media. Keperluan untuk dokumen dalam bentuk multimedia sangat meningkat dalam kurun waktu belakang ini, sehingga kebutuhan untuk menyimpan ataupun me-retrieve menjadi satu permasalahan yang primer. Jenis multimedia yang umum digunakan adalah jenis teks, seperti yang banyak terlihat sebagai pilihan utama dalam search engine seperti Yahoo, Google atau juga yang lainnya. Pencarian yang diinginkan tentunya bukan hanya sekedar mendapatkan hasil, tapi juga pencarian yang lebih efisien. Untuk keperluan indexing dan retrieval, inverted file digunakan untuk memberikan hasil yang lebih cepat. Namun akan ditemukan masalah apabila pembuatan inverted file tersebut berkaitan dengan jumlah data yang besar. Pada paper ini akan dijelaskan sebuah algoritma yaitu Fast Inversion sebagai pengembangan dari metode pembuatan inverted file dasar untuk menjawab kebutuhan terkait dengan jumlah data.

Kata kunci: *pencarian efisien, inverted file, algoritma Fast Inversion*

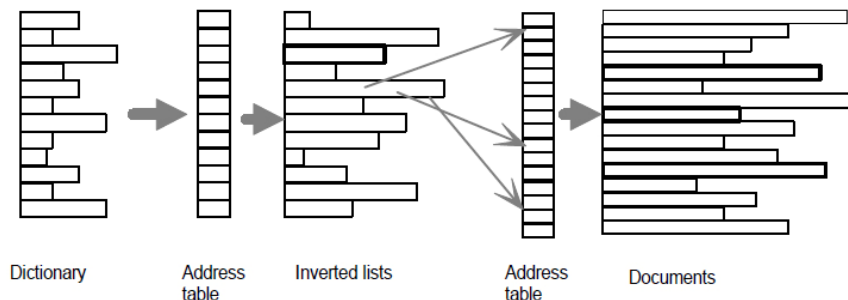
PENDAHULUAN

Pentingnya sistem retrieval pada teks terus bertambah secara terus menerus pada beberapa tahun ini karena pertambahan yang sangat cepat dalam hal kebutuhan kapasitas penyimpanan, bertambahnya performa pada semua tipe dari prosesor dan pertumbuhan eksponensial dari jaringan global yang menyediakan kebutuhan yang sangat besar dari dokumen-dokumen yang beraneka ragam. Pada sebuah dokumen terstruktur, informasi yang paling penting dari *image*, audio, atau video/animasi dapat ditemukan pada objek teks yang berasosiasi dengan objek media (Andreas & Giinter, 2009)

Salah satu faktor yang merupakan *critical factor* untuk *usability* dari sistem retrieval pada teks adalah performa dari *search engine* dan teknik yang mendasar untuk *indexing*. Proses dari *query evaluation* dan *indexing* untuk sistem retrieval teks yang memiliki performa tinggi biasanya terdiri dari beberapa langkah. Langkah-langkah yang biasanya termasuk di dalamnya (Candan & Sapino, 2010) adalah: (1) *query* pre-processing, contoh: beberapa tindakan normalisasi bahasa pada kata yang terdapat dalam *query*, ekstensi dari *query* dengan pengecekan melalui tesaurus untuk sinonimnya; (2) launching untuk *search engine* berdasarkan pada *pre-build indices*; (3) post-processing dari dokumen-dokumen kandidat untuk menyeleksi dokumen yang relevan dengan *query*; (4) *refinement* dari *query* berdasarkan *feedback* dari user dan evaluasi ulang dari *query*.

Semua tahapan ini sangat penting untuk kualitas dari hasil, namun pada paper ini hanya akan difokuskan pada tahap ke-2 dari tahapan di atas, yang sering disebut sebagai *indexing* dan *search engine*, lebih khusus lagi dalam perihal pengembangan pembentukan *inverted file*.

Dalam beberapa tahun ini, sudah ada beberapa metode yang diusulkan untuk pengindeksan dalam *text retrieval*, salah satunya adalah teknik berdasarkan *inverted list* (Justin & Alistair, 2006). *Inverted list* menyediakan performa yang bagus untuk pencarian dengan *single-keys*, akan tetapi performanya akan menurun ketika ukuran *query* bertambah, yang mana hal ini merupakan kepentingan yang utama untuk *text retrieval*. *Query* tertentu dimungkinkan untuk berisi beberapa *index term*. Lebih lanjut lagi, *query* tertentu dari user bisa dikombinasikan dengan beberapa term tambahan (seperti sinonim), yang membuat ukuran *query* bertambah besar secara signifikan. Perkembangan berikutnya yaitu pada *compressed inverted list*, hasilnya ialah mempertinggi performanya secara signifikan dalam hal *storage requirement*, termasuk di dalamnya waktu aksesnya. Namun sekali lagi akan terhambat ketika berhadapan dengan jumlah data yang besar. Di dalam paper ini akan diajukan satu algoritma yang diharapkan bisa untuk mengatasi permasalahan yang berkaitan dengan kuantitas dari dokumen tersebut. Model dari struktur data pada skema indexing ini dijelaskan pada Gambar 1.



Gambar 1. Struktur index secara umum.

Dictionary menampung semua *index term*, disarankan untuk menyimpannya pada main memory. Sebenarnya hal lebih tepat apabila dalam bahasa Inggris, karena jika untuk bahasa lain

seperti Rusia ataupun Finlandia, bentuk word yang disimpan di dalam dictionary akan secara signifikan bertambah besar. Namun disini diasumsikan bahwa setidaknya bagian yang penting dari dictionary bisa diletakkan pada main memory.

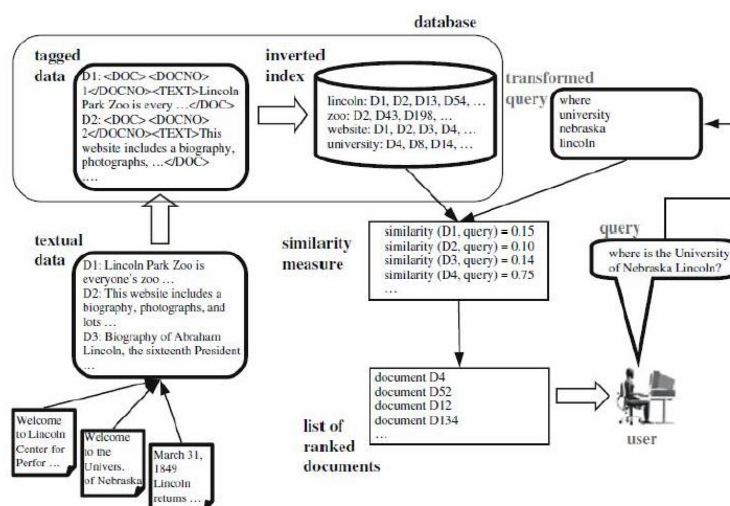
Inverted list disimpan dalam bentuk terkompresi pada disk. Algoritma kompresi memberikan kemungkinan untuk memproses list "on the fly" yang artinya bisa dikerjakan tanpa melakukan dekompresi dari data. Namun, teknologi ini hanya untuk *sequential scan* dari list, yang bisa menyebabkan *extra disk access* selama pemrosesan dari *inverted list* yang besar.

Kemudian kita mengasumsikan bahwa dokumen atau paragraf dari dokumen diberikan nomor secara sekuensial dan bisa di akses dengan *addresing table*.

Query direpresentasikan (untuk langkah *index lookup*) sebagai sekumpulan *index term* yang seharusnya muncul pada dokumen yang sama (atau paragraf yang sama), serta kombinasi dari kondisi yang ada. *Query evaluation* terdiri dari retrieval dari semua term yang disebutkan di *query* dari dictionary dan kemudian merge dari *inverted list* yang berhubungan, untuk memperoleh list kandidat dari dokumen yang relevan.

List kandidat di-*passing* ke langkah berikutnya dari *query evaluation*, yang mengikutsertakan *document scan* untuk pengecekan kondisi yang mungkin saja terlewat ketika pengecekan oleh *inverted list*. Langkah yang paling mahal (*time-consuming*) dari *query evaluation* adalah merge daripada *inverted list*.

Secara garis besar arsitektur sistem IR diperlihatkan pada Gambar 2. Ada dua pekerjaan yang ditangani oleh sistem ini, yaitu melakukan *pre-processing* terhadap *database* dan kemudian menerapkan metode tertentu untuk menghitung kedekatan (relevansi atau *similarity*) antara dokumen di dalam *database* yang telah di-*preprocess* dengan *query* pengguna. Pada tahapan *preprocessing*, sistem yang berurusan dengan dokumen *semi-structured* biasanya memberikan tag tertentu pada term-term atau bagian dari dokumen; sedangkan pada dokumen tidak terstruktur proses ini dilewati dan membiarkan term tanpa imbuhan tag. *Query* yang dimasukkan pengguna dikonversi sesuai aturan tertentu untuk mengekstrak term-term penting yang konsisten dengan term-term yang sebelumnya telah diekstrak dari dokumen dan menghitung relevansi antara *query* dan dokumen berdasarkan pada term-term tersebut. Hasilnya, sistem mengembalikan suatu daftar dokumen terurut sesuai nilai kemiripannya dengan *query* pengguna.



Gambar 2. Arsitektur sistem information *retrieval*.

Setiap dokumen (termasuk *query*) direpresentasikan menggunakan model *bag-of-words* yang mengabaikan urutan dari kata-kata di dalam dokumen, struktur sintaktis dari dokumen dan kalimat. Dokumen ditransformasi ke dalam suatu wadah berisi kata-kata independen. Kata disimpan dalam suatu *database* pencarian khusus yang ditata sebagai sebuah inverted index. Index ini merupakan konversi dari dokumen asli yang mengandung sekumpulan kata ke dalam daftar kata yang berasosiasi dengan dokumen terkait dimana kata-kata tersebut muncul.

Pembangunan index dari koleksi dokumen merupakan tugas pokok pada tahapan *preprocessing* di dalam IR. Kualitas index mempengaruhi efektifitas dan efisiensi sistem IR. Index dokumen adalah himpunan term yang menunjukkan isi atau topik yang dikandung oleh dokumen. Index akan membedakan suatu dokumen dari dokumen lain yang berada di dalam koleksi. Ukuran index yang kecil dapat mengakibatkan hasil buruk dan mungkin dapat kehilangan beberapa item yang relevan. Index yang besar memungkinkan retrieval banyak dokumen bermanfaat sekaligus dapat menaikkan jumlah dokumen yang tidak relevan dan juga dapat menurunkan kecepatan pencarian (*searching*).

Terkadang, banyak dokumen yang di-retrieve dari *query* yang sudah diberikan memberikan dokumen-dokumen yang tidak relevan. (Young-In Song et al, 2003). Oleh karenanya, pembuatan *inverted index* harus melibatkan konsep *linguistic processing* yang bertujuan mengekstrak term-term penting dari dokumen yang direpresentasikan sebagai *bag-of-words*. Ekstraksi term biasanya melibatkan dua operasi utama berikut: (1) penghapusan *stop-words. Stop words* didefinisikan sebagai term yang tidak berhubungan (*irrelevant*) dengan subyek utama dari *database* me*skip*un kata tersebut sering kali hadir di dalam dokumen. Contoh *Stop words* adalah *a, an, the, this, that, these, those, her, his, its, my, our, their, your, all, few, many, several, some, every, for, and, nor, bit, or, yet, so, also, after, although, if, unless, because, on, beneath, over, of, during, beside*, dll. *Stop words* termasuk pula beberapa kata yang didefinisikan tertentu yang terkait dengan subyek *database*, misal pada *database* yang menampung daftar paper penelitian terkait dengan *heart diseases*, maka kata *heart* dan *disease* sebaiknya dihapus; (2) *stemming* – kata-kata yang muncul di dalam dokumen sering mempunyai banyak varian morfologik. Karena itu, setiap kata yang bukan stop-words direduksi ke stemmed word (term) yang cocok yaitu kata tersebut di-*stem* untuk mendapatkan bentuk akarnya dengan menghilangkan awalan atau akhiran. Dengan cara ini, diperoleh kelompok kata yang cocok dimana kata-kata di dalam kelompok tersebut merupakan varian sintaktis dari satu sama lain dan dapat menghimpun hanya satu kata per kelompok. Sebagai contoh, kata disease, diseases, diseased berbagi-pakai term stem umum disease, dan dapat diperlakukan sebagai bentuk lain dari kata ini.

Algoritma *stemming* paling umum untuk bahasa Inggris dan dinyatakan efektif adalah algoritma Porter. Algoritma ini terdiri dari lima fase reduksi kata yang diterapkan secara urut. Di dalam setiap fase terdapat berbagai konvensi untuk memilih aturan seperti memilih aturan dari setiap grup aturan yang menerapkan terhadap akhiran paling panjang. Di dalam fase pertama, konvensi tersebut digunakan mengikuti aturan grup berikut:

Rule	**Contoh**
SSES → SS	caresses → caress
IES → I	ponies → poni
SS → SS	caress → caress
S →	cats → cat

Menurut (UMS IR & Klaf) terdapat lima langkah pembangunan *inverted index*, yaitu: (1) penghapusan format dan markup dari dalam dokumen (*markup removal*); (2) pemisahan rangkaian term (*tokenization*); (3) penyaringan (*filtration*); (4) pengembalian term ke bentuk akar kata (*stemming*); (5) pemberian bobot terhadap term (*weighting*).

Paper ini memperkenalkan algoritma Fast Inversion yang dimodifikasikan sehingga tidak hanya berkenaan untuk menjawab kebutuhan mengenai kuantatias data dengan skala yang besar, akan tetapi juga mengetengahkan permasalahan efisiensinya.

METODE

Algoritma Fast Inversion (Existing Algorithm)

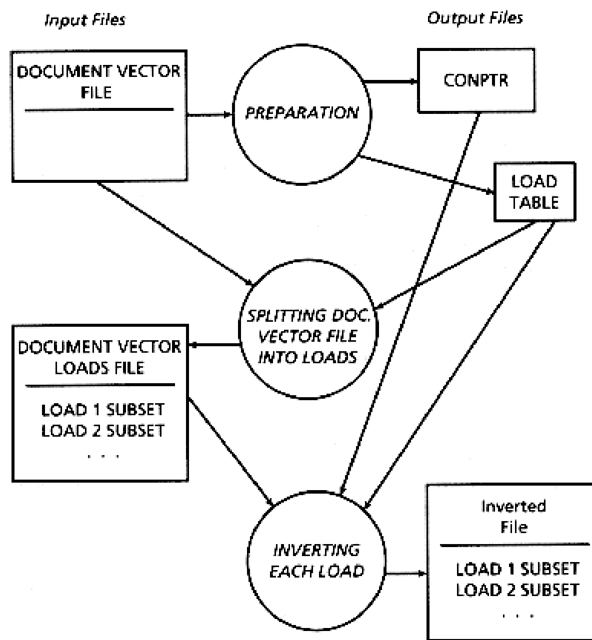
Algoritma Fast Inversion (FAST-INV) merupakan salah satu modifikasi dari teknik dasar dalam pembuatan *inverted file*. Teknik ini mengambil benefit dari dua prinsip, yaitu *primary memory* yang besar pada komputer saat ini, dan *inherent order* dari data *input*. Prinsip pertama penting karena PC dengan kapasitas *primary memory* lebih dari 1 MB sudah sangat umum, dan juga *mainframe* bisa memiliki lebih dari 100 MB memory. Bahkan apabila *database* ada dalam ukuran 1 GB, jika dapat di-*split* menjadi *memory load* yang bisa dengan cepat diproses dan dikombinasikan, *cost* secara keseluruhan akan bisa diminimalisasikan. Prinsip kedua juga krusial karena *file* yang besar akan sangat mahal untuk menggunakan polinomial atau bahkan algoritma n log n sorting. *Cost* ini akan digabungkan jika memori tidak digunakan, sehingga nantinya *cost* ada untuk *disk operation*.

Sebuah algoritma yang cepat untuk melakukan inverse dari data magnetic sudah dikembangkan (Yaoguo & Douglas, 2003). Peng*input*an pada FAST-INV adalah sebuah *document vector file* yang memuat *concept vector* untuk beberapa dokumen dari *collection* untuk di berikan index. Contoh dari *document vector file* bisa dilihat pada Gambar 3.

DOC	CON.
1	3
1	5
1	12
1	14
2	1
2	3
2	4
2	11
2	12
3	2
3	4
3	5

Gambar 3. Contoh *document vector.*

Document number terdapat pada kolom bagian kiri, dan *concept number* dari kata pada setiap dokumen terdapat pada kolom bagian kanan. Hal ini sama seperti *word list* untuk metode dasar, kecuali di sisi katanya, karena disini kata direpresentasikan oleh *concept number*. Satu *concept number* untuk setiap kata yang unik dari *collection* (contoh apabila ada 250000 *unique word* maka jumlah *unique concept number* juga akan menjadi 250000). Perhatikan bahwa *document vector file* ada dalam format terurut, sehingga *concept number* pun terurut sesuai dengan *document number*, dan *document number* diurutkan dengan *collection*. Hal ini penting untuk FAST-INV bisa bekerja dengan tepat. Gamabr 4 berikut memuat skema keseluruhan Fast-INV.



Gambar 4. Skema keseluruhan FAST-INV.

Preparation

Untuk bisa menjelaskan algoritma FAST-INV dengan baik, beberapa definisi dibutuhkan:

```
HCN = highest concept number in dictionary
L = number of document/concept (or concept/document) pairs in the
collection
M = available primary memory size, in bytes
```

Pada langkah pertama, seluruh *document vector file* dapat dibaca dan akan dihasilkan dua *file* baru yaitu CONPTR (*concept postings/pointers*) dan sebuah *load table*. Diasumsikan bahwa *M* jauh lebih besar daripada *HCN*, sehingga kedua *file* ini bisa dibangun di dalam *primary memory*. Namun, diasumsikan juga bahwa *M* lebih kecil dari *L*, sehingga beberapa *primary memory load* akan dibutuhkan untuk memproses data dokumen. Karena entri di dalam *document vector file* akan dikelompokkan oleh *concept number*, dengan konsep ini dalam urutan *ascending*, hal ini tepat untuk bisa melihat data yang bisa sewaktu-waktu ditransformasikan sebelumnya ke bagian dari *j* sehingga:

```
 L/j < M, sehingga setiap bagian akan fit ke dalam primary memory
HCN/j concept, kira-kira, akan diasosiasikan dengan setiap bagian
```

Hal ini memungkinkan setiap bagian dari j dibaca ke dalam *primary memory*, diinversikan di sana, dan *output*-nya akan mudah ditambahkan ke dalam *inverted file* akhir.

Secara spesifik, proses *preparation* terdiri dari: (1) alokasi *array*, con_entries_cnt, of size HCN, diinisialisasikan menjadi nol; (2) untuk setiap entri dari <doc#,con#> pada *document vector file*: tambahkan con_entries_cnt[con#]; (3) gunakan hanya con_entries_cnt yang dibangun untuk membuat disk version dari CONPTR; (4) inisialisasikan *load table*; (5) untuk setiap pasangan <con#,count> yang didapatkan dari con_entries_cnt: jika tidak ada ruangan untuk document dengan konsep ini untuk fit di dalam *load* yang sudah ada, maka buat entri baru dari *load* table dan inisialisasikan entri untuk *load* selanjutnya, dan juga update informasi untuk entri dari *load* tabe; sebelumnya.

Setelah langkah pertama melalui *input*, *file* CONPTR sudah dibangun dan *load table* dibutuhkan dalam langkah berikutnya dari algoritma yang telah dibuat. Perhatikan bahwa pengujian untuk ruangan dalam *load* yang diberikan menjalankan constraint yang data untuk sebuah *load* akan fit ke dalam *available memory*. Secara spesifik:

```
Let LL = length of current load (i.e., number of concept/weight pairs)
S = spread of concept numbers in the current load (i.e., end concept -
start concept + 1 )
8 bytes = space needed for each concept/weight pair
4 bytes = space needed for each concept to store count of postings for it
```

Kemudian *constraint* yang harus ditemukan untuk konsep yang lain untuk ditambahkan ke dalam *load* yang sebelumnya adalah:

```
8 * LL + 4 * S < M
```

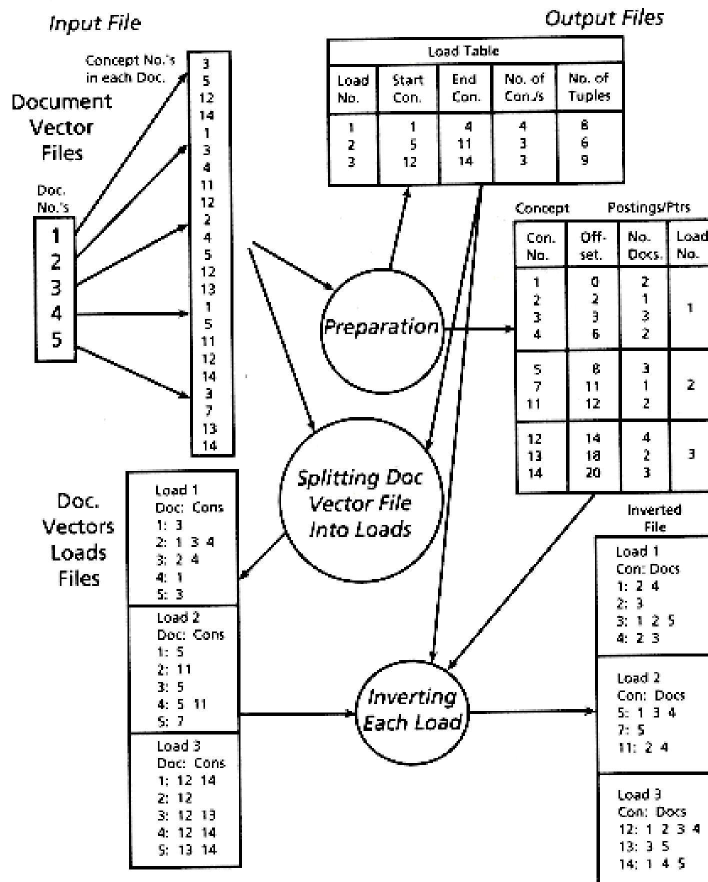
Splitting *Document vector* File

Load table menunjukkan kisaran konsep yang harus diprotes untuk setiap *primary memory load*. Ada dua pendekatan untuk menangani banyaknya *load*. Pendekatan yang pertama, yang saat ini digunakan, adalah dengan membuat sebuah langkah melalui *document vector file* untuk mendapatkan masukan untuk setiap *load*. Hal ini memiliki keuntungan tidak memerlukan tempat penyimpanan tambahan (me*skip*un dapat dihindarkan melalui penggunaan *magnetic tape*), tetapi memiliki kelemahan membutuhkan disk I/O yang mahal. Pendekatan kedua adalah untuk membangun sebuah kopian baru dari *document vector collection*, dengan pemisahan yang diinginkan ke dalam *load*. Hal ini dengan mudah dapat dilakukan dengan menggunakan *load table*, karena ukuran setiap *load* diketahui, hanya dalam satu langkah dengan melewati *input*. Karena setiap *document vector* dibaca, dipisahkan menjadi bagian untuk setiap rentang konsep dalam *load table*, dan bagian-bagian yang ditambahkan ke akhir dari bagian yang berkaitan dengan *document collection file*. Dengan buffering I/O, biaya operasi ini sebanding dengan ukuran *file*, dan pada dasarnya biaya sama dengan menyalin *file*.

Inverting each *load*

Ketika *load* akan diproses, bagian yang sesuai dari *file* CONPTR diperlukan. Ukuran *array* dari *output* yang sama dengan *subset* dari *input* pada *document vector file* diperlukan. Karena setiap *document vector* diproses, *offset* (sebelumnya tercatat dalam CONPTR) untuk suatu konsep yang diberikan digunakan untuk menempatkan entri dari *document/weight* yang sesuai, dan kemudian *offset*-nya bertambah. Dengan demikian, data CONPTR memungkinkan *input* yang akan langsung dipetakan ke *output*, tanpa *sorting* apapun. Pada akhir *load input*, *output* yang baru dibangun ditambahkan ke dalam *inverted file*.

Contoh:
Gambar 5 mengilustrasikan proses FAST-INV menggunakan data sampel. *File input* dari *document vector* dibaca secara langsung untuk menghasilkan *file concept postings/pointers* (disimpan dalam disk sebagai CONPTR) dan *load table*. Tiga *load* akan dibutuhkan, untuk konsep dalam rentang 1-4, 5-11, dan 12-14. Ada sepuluh konsep yang berbeda, dan HCN adalah 14.



Gambar 5. Contoh FAST-INV.

Tahap kedua pengolahan menggunakan *load table* untuk *split file input document vector* dan membuat *file document vector load*. Ada tiga bagian, sesuai dengan tiga *load*. Dapat dilihat bahwa *document vector* pada setiap *load* dipersingkat karena hanya konsep-konsep dalam *range* yang diijinkan untuk beban itu dimasukkan.

Tahap terakhir dari pengolahan melibatkan penginversian setiap bagian dari *file document vector load*, dengan menggunakan *primary memory*, dan menambahkan hasilnya ke dalam *inverted file*. Bagian yang sesuai dengan *file* CONPTR digunakan sehingga inversi hanyalah menyalin data ke tempat yang benar, bukan hanya sekedar sort.

HASIL DAN PEMBAHASAN

Algoritma yang Diusulkan (Fast Inversion with *Skip*ping)

Algoritma tersebut merupakan algoritma murni dari Fast Inversion. Yang diusulkan disini adalah pada bagian *Inverting Each Load*, ditambahkan metode *Skipping*, sehingga lebih memperkecil ukuran dari *inverted file* untuk membuat indexing dan retrieval menjadi lebih baik. Scalability adalah sebuah konsen sentral dari setiap perancangan sistem (Riley et al., 2008).

Misalkan $k = | C |$ kandidat akan di uji dengan sebuah *compressed inverted list* yang memuat *pointer* p $<d, f_{d,t}>$. Kemudian katakanlah costnya t_d detik untuk me-*decode* satu *pointer*.

Ketika $k << p$, performa yang lebih cepat dimungkinkan jika lokasi tambahan yang mana *decoding* dapat memulai diperkenalkan kepada *compressed inverted list*. Sebagai contohnya, anggaplah bahwa *synchronization point* p_1 dimungkinkan. Maka index ke dalam *inverted list* memuat *pointer* p_1 "document number, bit address" dan bisa menyimpan dirinya sendiri sebagai *compressed sequence* dari "*difference in document number, difference in bit address*" *run length*. Jika pengkompresan nilai ini disisipkan dengan *run length* dari list sebagai sebuah *sequence* dari *skip*, sebuah *single self-indexing inverted file* dibuat.

Sebagai contohnya, *set* dari *pointer* $<d, f_{d,t}>$

<5, 1><8, 1><12, 2><13, 3><15, 1><18, 1><23, 2><28, 1><29, 1>. . . .

Disimpan sebagai *d-gaps*, yang direpresentasikan

<5, 1><3, 1><4, 2><1, 3><2, 1><3, 1><5, 2><5, 1><1, 1>. . . .

Dengan *skip* setiap tiga *pointer*, inverted list menjadi sequence dari grup dengan jarak satu sama lain setiap 3 *pointer*, dengan *skip* yang memisahkan grupnya. Contoh list yang sama adalah

<<5, *a*2>><5, 1><3, 1><4, 2><<13, *a*3>><1, 3><2, 1><3, 1>

<<23, *a*4>><5, 2><5, 1><1, 1><<40, *a*5>>. . . ,

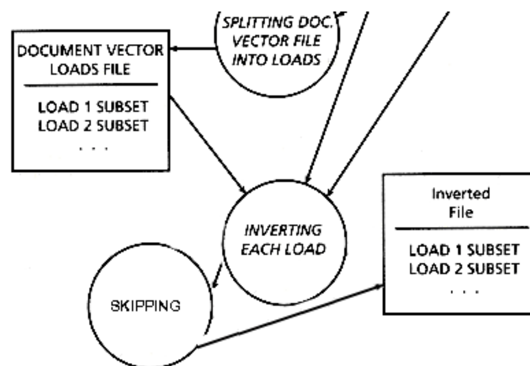
Dimana a2 adalah address dari bit pertama dari *pointer* kedua yang di *skip*, a3 adalah address dari bit pertama dari *pointer* ketiga yang di *skip* dan seterusnya. Format ini masih memuat redundancy di dalamnya, dimana kedua list dari document number yang di *skip* dan list dari address bit dapat di kodekan sebagai perbedaan, dan document number pertama pada setiap set dari tiga value $<d, f_{d,t}>$ tidak lagi dibutuhkan.

Berhubungan dengan perubahan ini, inverted list akhirnya menjadi:

<<5, *a*2>><1><3, 1><4, 2><<8, *a*3 - *a*2>><3><2, 1><3, 1>

<<10, *a*4 - *a*3>><2><5, 1><1, 1><(17, *a*5 - *a*4>>. . . .

Bentuk inverted list setelah dilakukan *skipping* akan menjadi lebih singkat. Tentunya hal ini akan meningkatkan dalam proses indexing dan retrievalnya. Sebelum masuk ke *inverted file* final, dilakukan *skip*ping terlebih dahulu. Arsitektur dari algoritma akan dimodifikasikan menjadi (Gambar 6):



Gambar 7. Fast-inversion & skipping.

Berdasarkan diskusi di atas, dapat dilihat bahwa FAST-INV merupakan algoritma linier dari ukuran *file input*, *L*. *File input* disk tersebut harus dibaca tiga kali, dan ditulis dua kali (menggunakan skema pemisahan kedua). Pengolahan dalam *primary memory* dibatasi untuk scan melalui *input*, dengan perhitungan yang sesuai (murah) yang diperlukan pada setiap entri.

Jadi, sudah jelas bahwa FAST-INV memberikan performa lebih baik dibandingkan dengan metode inverse lainnya, seperti yang digunakan dalam sistem lain seperti Sire dan SMART. Untuk menunjukkan fakta ini, berbagai tes dilakukan. Tabel 1 merangkum hasil untuk mengindeks koleksi 12.684 document/8.68 INSPEC megabyte.

Tabel 1
Hasil FAST-INV

Method	Comments	Indexing	Inversion
SIRE	Dictionary built during inversion	35	72
SMART	Dictionary built during indexing	49	11
FAST-INV	Dictionary built during indexing	49	1:14

Keefisienan dalam pencarian dalam skala data besar ini ditambahkan lagi satu fitur dalam pembuatan *inverted file*, yaitu *skip*ping, yang menghasilkan *output* berkurangnya *data redundancy*.

PENUTUP

Pada paper ini, diajukan sebuah teknik kombinasi antara algoritma Fast Inversion dengan *skip*ping pada bagian sebelum pembentukan *inverted file* akhir. Teknik ini selain meningkatkan efisiensi dari retrieval juga mampu untuk meng-*handle* permasalahan jumlah data dalam skala yang besar. Implementasi dari pengembangan algoritma ini tentunya akan menambah *value* dalam pemrosesan *indexing* dan *retrieval* dari multimedia data dalam bentuk teks.

DAFTAR PUSTAKA

Andreas, H. & Giinter, R. (2009). *Combining Multimedia Retrieval and Text Retrieval Search Structured Documents in Digital Libraries*. Bamberg, Otto-Friedrich University of Bamberg, Faculty of Social and Economic Sciences.

Candan, K Selçuk. & Sapino, Maria Luisa. (2010). *Data Management for Multimedia Retrieval*. England: Cambridge University Press.

Riley, M., Heinen, E., & Ghosh, J. (2008). A text retrieval approach to content-based audio hashing. *ISMIR*, 295-300.

Yaoguo, L. & Douglas, O. W. (2003). Fast inversion of large-scale magnetic data using wavelet transforms and a logarithmic barrier method. *Geophys Journal International,* 152, 251–265.

Young-In Song, Kyoung-Soo Han, Hee-Cheol Seo, Sang-Bum Kim, Hae-Chang Rim. (2003). *Biomedical Text Retrieval System at Korea University*. Diakses dari http://trec.nist.gov/pubs/trec12/papers/koreau.genomics.pdf.

Zobel, Justin & Moffat, Alistair. (2006). Inverted files for text search engines. *ACM Computing Surveys,38* (2), doi:10.1145/1132956.1132959.