BAB II

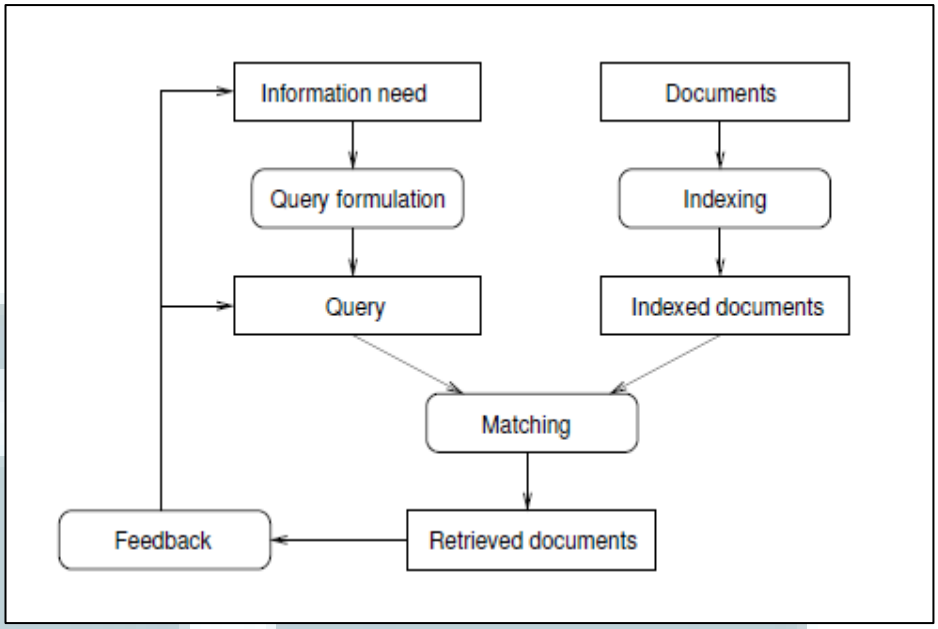
TINJAUAN PUSTAKA

2.1 Information Retrieval

Manning (2009, pp. 1) mendefinisikan *information retrieval* sebagai aktivitas pencarian material (umumnya dokumen) dari sesuatu yang tidak terstruktur (umumnya teks) guna memenuhi kebutuhan informasi dari sekumpulan koleksi. Menurut Hiemstra (2009, pp. 2), *information retrieval system* dapat dikatakan sempurna apabila dia hanya mengambil dokumen-dokumen yang relevan dan menyingkirkan yang tidak relevan.

Bagaimanapun, *information retrieval system* yang sempurna belum pernah ada dan tidak mungkin ada, karena *search statement* yang diberikan oleh pengguna tidak selalu lengkap dan relevansi masih bergantung pada pendapat subjektif pengguna. Dalam prakteknya, dua orang pengguna yang memberikan *query statement* yang sama ke dalam *information retrieval system* akan menilai relevansi secara berbeda, salah satu dari mereka mungkin ada yang menyukai hasilnya, sedangkan yang lainnya tidak (Heimstra, 2009, pp. 2).

Heimstra (2009, pp. 2) mengemukakan bahwa proses dasar di dalam *information retrieval system* terdiri dari tiga bagian yaitu, representasi konten dari suatu dokumen, reperesentasi dari informasi yang dibutuhkan pengguna, dan perbandingan dari kedua representasi tersebut. Proses ini dapat dijelaskan pada gambar 2.1, dimana kotak persegi menggambarkan data dan kotak bulat menggambarkan proses.



Gambar 2.1 *Information Retrieval Process*
Sumber: Information Retrieval Models (2009, pp. 2).

Menurut Heimstra (2009, pp. 2) merepresentasikan dokumen sering disebut dengan proses *indexing*. Proses ini dilakukan secara *offline*, sehingga pengguna akhir tidak terlibat secara langsung di dalamnya. Hasil dari proses ini adalah representasi dari dokumen tersebut. Pengguna tidak melakukan pencarian hanya untuk kesenangan semata, melainkan mereka membutuhkan informasi tertentu. Proses merepresentasikan kebutuhan informasi sering dianggap sebagai proses formulasi *query*. Dalam pengertian yang luas, formulasi *query* dapat diartikan sebagai dialog interaktif antara sistem dengan pengguna yang tidak hanya mengarahkan pada *query* yang sesuai, namun juga pemahaman yang lebih jelas mengenai kebutuhan informasi itu.

Perbandingan antara query dengan representasi dokumen disebut juga dengan *matching process*. *Matching process* biasanya menghasilkan jawaban

berupa daftar peringkat dari semua dokumen. Pengguna akan menelusuri keseluruhan daftar tersebut untuk mencari informasi yang mereka butuhkan. Sistem akan meletakan jawaban yang paling relevan pada urutan teratas untuk meminimalisir waktu yang dibutuhkan pengguna untuk membaca seluruh dokumen.

Pemodelan dalam sistem *information retrieval* bertujuan untuk menentukan detil dari representasi dokumen, representasi *query*, dan fungsionalitas *retrieval*. Alhenshiri (2007, pp. 58) mengemukakan beberapa model dasar dalam *information retrieval* dapat diklasifikasikan menjadi *boolean, vector,* dan *probabilistic model.*

1. *Boolean Model*

   Pada *boolean model*, dokumen diasosiasikan dengan sekumpulan kata kunci. Suatu *query* merupakan sekumpulan ekspresi berupa kata kunci yang dipisahkan dengan kata *AND, OR,* atau *NOT/BUT*. Fungsi dari sistem *retrieval* pada model ini menentukan apakah dokumen tersebut berisi informasi yang relevan atau tidak. Menurut faktanya, bahwa data yang ada di dalam *web* sangatlah redundan, kemungkinan untuk menghasilkan jutaan jawaban yang memiliki peringkat sama akan mempersulit dalam menentukan prioritas jawaban ketika model ini diimplementasikan.

2. *Probabilistic Model*

   Probabilistic model berdasarkan pada asumsi bahwa di dalam koleksi terdapat sekumpulan dokumen yang merupakan jawaban dari *query* pengguna. Dalam model ini, sekumpulan dokumen mula-mula dipilih dan

diamati oleh pengguna. Proses ini dilakukan menggunakan antarmuka interaktif dimana pengguna dapat segera memberikan *feedback* untuk mendapatkan kumpulan jawaban. Dengan kata lain, pendekatan dengan model ini sangat bergantung pada *feedback* pengguna pada waktu *query*.

3. *Vector Space Model*

Model ini adalah model yang paling dikenal dalam *information retrieval*. Setiap dokumen dan *query* direpresentasikan sebagai suatu vektor (*term/document*). Fungsi *retrieval* pada model ini adalah membandingkan vektor *query* dengan setiap baris yang merepresentasikan dokumen di dalam ruang vektor. Derajat kemiripan antara vektor *query* dengan seluruh dokumen yang direpresentasikan dalam ruang vektor digunakan untuk memberi peringkat pada setiap dokumen.

Menurut Wibowo (2011, pp. 3) di dalam *information retrieval*, jawaban-jawaban yang ditampilkan oleh suatu algoritma harus memenuhi beberapa persyaratan sebagai berikut:

1. *Recall* merupakan suatu nilai dimana sistem dapat menemukan seluruh dokumen yang relevan dalam koleksi. Nilai *recall* tertinggi adalah 1, yang berarti seluruh dokumen dalam koleksi berhasil ditemukan.

$$recall = \frac{jumlah\ dokumen\ relevan\ yang\ ditemukan}{jumlah\ dokumen\ relevan\ dalam\ koleksi} \quad \text{.....} \quad \text{rumus 2.1}$$

2. *Precision* merupakan suatu nilai dimana sistem hanya menemukan dokumen yang relevan saja dari dalam koleksi. Nilai *precision* tertinggi adalah 1, yang berarti seluruh dokumen yang ditemukan adalah relevan.

$$precision = \frac{jumlah\ dokumen\ relevan\ ditemukan}{jumlah\ dokumen\ ditemukan} \quad \text{..... rumus 2.2}$$

2.2 Answer Graph Generation

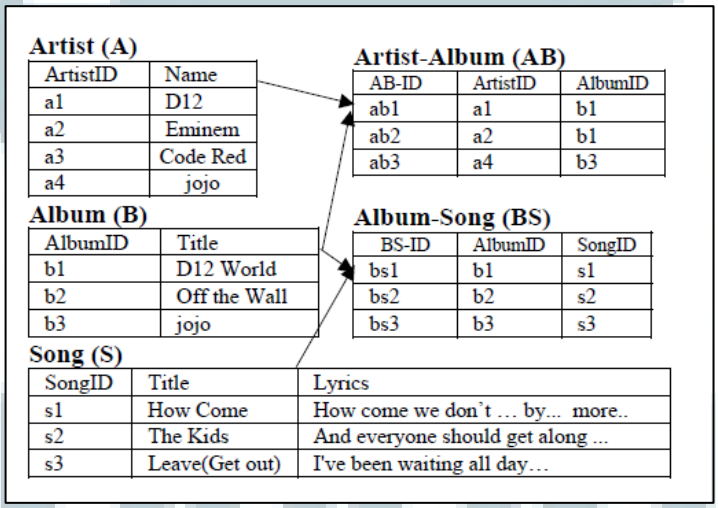
Answer Graph Generation merupakan sebuah *framework* yang digunakan untuk menghasilkan jawaban dari setiap *query* yang diberikan oleh pengguna. Algoritma ini merupakan modifikasi dari algoritma *Candidate Network Generation* yang ditemukan oleh Hriditis (2003). Liu (2006, pp. 3) menjelaskan pemodelan yang digunakan oleh *Answer Graph Generation* untuk menghasilkan jawaban:

1. *Database schema* dimodelkan dengan suatu *graph*, dimana *schema graph* merupakan *directed graph SG*.
2. Untuk setiap tabel *Ri* yang ada di dalam *database* merupakan *node* di dalam *schema graph*.
3. Jika terdapat relasi antara *primary* dengan *foreign key* dari tabel *Ri* ke tabel *Rj* pada *database*, maka digambarkan suatu *edge* dari tabel *Ri* ke tabel *Rj* pada *schema graph*.
4. Setiap table *Ri* memiliki *mi* ($mi \leq 0$) *text columns* $\{c^i_1, c^i_2, \ldots, c^i_{mi}\}$.
5. *Tuple Tree T* merupakan gabungan *tree* dari beberapa *tuple*.
6. Setiap *node ti* di dalam *T* merupakan *tuple* di dalam *database*.

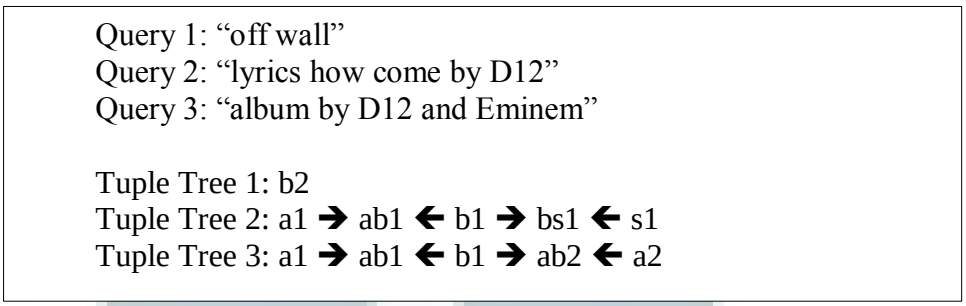
7. Anggap (R_i, R_j) merupakan suatu *edge* di dalam *schema graph*. $t_i \in R_i$, $t_j \in Rj$, dan (t_i *join* t_j) $\in$ (R_i *join* R_j). Maka (t_i, t_j) adalah suatu *edge* di dalam *tuple tree T*.
8. Ukuran dari *tuple tree T* adalah jumlah *tuple* yang ada di dalamnya.

Menurut Liu (2006, pp. 3) suatu *tuple tree T* dapat dikatakan sebagai jawaban atas *query Q* jika memenuhi beberapa persyaratan berikut.

1. Setiap *leaf node ti* di dalam *T* mengandung paling sedikit 1 buah kata kunci di dalam *Q*. (singkatnya, nilai dari *text column* pada *tuple ti* mengandung paling sedikit 1 buah kata kunci dari *Q* dan *leaf node* yang berbeda memungkinkan untuk mengandung kata kunci yang sama).
2. Setiap *tuple* hanya muncul paling banyak 1 kali di dalam *tuple tree*.



Gambar 2.2 Contoh *Lyrics Database*
Sumber: Effective Keyword Search in Relational Databases (2006. pp. 1).



Gambar 2.3 *Queries* dan *Tuple Trees*
Sumber: Effective Keyword Search in Relational Databases (2006. pp. 1).

Liu (2006, pp. 3) mengilustrasikan proses pembentukkan *Answer Graph Generation* pada *database* lirik musik. Untuk menghasilkan jawaban dari suatu *keyword Q, query tuple set* R^Q dari *table R* didefinisikan sebagai seluruh *tuple* yang ada di *R* dan mengandung paling sedikit 1 buah *keyword Q*. Contohnya, *query tuple set* dari tabel *Artist* untuk *Query* 1, *Query* 2, dan *Query* 3 secara berurutan adalah A^{Q1} = {}, A^{Q2}, = {a1}, dan A^{Q3} = {a1, a2}. Kita mendefinisikan *free tuple set* R^F dari tabel *R* sebagai semua *tuple* yang ada di *R*. Contohnya, *free query set* untuk tabel *Artist* A^F = {a1, a2, a3}.

Berdasarkan definisi dari suatu jawaban pada *query Q* yang diberikan, jika *tuple tree T* adalah suatu jawaban, maka setiap *leaf node* t_i yang ada di dalam tabel R_i merupakan milik *query tuple set* R_i^Q, dan setiap *non-leaf node* t_j di dalam tabel R_j milik *free tuple set* R_j^F. R^{QorF} merupakan notasi dari *tuple set* dimana dapat diartikan sebagai *query tuple set* atau *free tuple set*. Berdasarkan adanya relasi m:n (contoh, sebuah album dapat diproduksi oleh lebih dari 1 artis), *tuple set* dari tabel yang sama dapat muncul lebih dari sekali pada *join expression*. Jika hal ini terjadi, setiap kemunculan dari *tuple set* yang sama harus dianggap sebagai *alias* yang berbeda dari suatu *tuple set*. Jika muncul lebih dari satu *alias* dari *tuple*

set pada *join expression*, maka harus ditambahkan suatu kondisi seperti "$A^{1,Q}$.ArtistID != $A^{2,Q}$.ArtistID" untuk menghindari dari menghasilkan *tuple tree* seperti a1 ➔ ab1🡐 b1 ➔ ab1🡐 a1 dimana *tuple tree* ini melanggar kondisi kedua dari definisi suatu jawaban.

Jika ada suatu *edge* (R_i, R_j) di dalam *schema graph* SG, dan *n* adalah jumlah maksimum dari *tuple* berbeda di dalam *foreign tabel* R_j yang dapat di *join* dengan satu *tuple* di dalam tabel *primary* R_i, maka secara teori setiap *tuple set* dari R_i dapat dihubungkan pada sejumlah *n tuple set* dari R_j yang ada di dalam *answer graph* untuk menghasilkan seluruh *tuple tree* yang memungkinkan, masing-masing berisi satu *tuple* R_i yang dapat di-*join* dengan paling banyak sejumlah *n tuple* yang berbeda dari R_j. Dengan demikian, dapat dibuat 2 parameter, maxn (jumlah maksimum *tuple set* dari *foreign table* yang dapat di *join* dengan *primary table*) dan MAXN (jumlah maksimum *tuple set* di dalam suatu *answer graph*).

2.3 Tuples Tree Ranking

Liu (2006, pp. 5) mengemukakan bahwa pembobotan istilah dalam sebuah dokumen dapat ditentukan berdasarkan 3 faktor berikut.

1. *Term Frequency* (*tf* pada rumus 2.5), merupakan jumlah kemunculan suatu istilah pada sebuah dokumen. Secara intuitif, semakin banyak istilah yang muncul pada sebuah dokumen, semakin tinggi nilai bobot istilah tersebut. Bagaimanapun, istilah yang sama dapat muncul berulang kali di dalam dokumen yang panjang. Oleh karena itu, pembobotan istilah tidak boleh bergantung secara *linear* pada perhitungan *tf* pada umumnya ketika nilai *tf*

begitu tinggi. Solusi permasalahan ini diatasi dengan menggunakan normalisasi *ntf* yang dijabarkan pada rumus 2.5.

2. *Document Frequency* (*df* pada rumus 2.6), merupakan jumlah banyaknya dokumen dimana istilah tersebut muncul pada sebuah koleksi. Secara intuitif, semakin banyak dokumen yang memiliki istilah tersebut, semakin buruk istilah tersebut sebagai suatu diskriminator. Oleh karena itu, bobot yang diberikan pada istilah tersebut sangat kecil. Rumus 2.6 merupakan *inverse document frequency* yang digunakan untuk menormalisasi nilai frekuensi dokumen.
3. *Document Length* (*dl* pada rumus 2.7), merupakan nilai panjang dokumen, dapat berupa *byte* atau jumlah banyaknya istilah yang dikandung oleh dokumen tersebut. Semakin panjang dokumen, maka dokumen tersebut banyak mengandung istilah dan memiliki frekuensi istilah yang tinggi. Dokumen yang panjang cenderung memiliki nilai *inner product* yang lebih tinggi ketika suatu query diberikan. Oleh karena itu, rumus 2.7 menormalisasi nilai tersebut untuk mengurangi bobot istilah pada dokumen yang panjang, dimana *s* adalah suatu konstanta yang biasanya ditetapkan dengan nilai 0.2.

$Sim\ (Q, D) = \sum_{k \in Q,D} weight(k, Q) * weight(k, D)$ ..... rumus 2.3

weight (k, D) = $\frac{ntf}{ndl} * idf$ ..... rumus 2.4

ntf = 1+ ln(1 + ln(tf)) ..... rumus 2.5

idf = ln $\frac{N}{df+1}$ ..... rumus 2.6

ndl = (1 – s) + s * $\frac{dl}{avgdl}$ …….. rumus 2.7

Di dalam *information retrieval*, dokumen merupakan unit informasi dasar yang disimpan di dalam *text database*. Sedangkan unit informasi dasar yang disimpan di dalam *relational database* berbentuk *text column*, dimana jawaban yang dibutuhkan oleh pengguna berupa *tuple tree* yang dirangkai menggunakan sekumpulan *tuples*. Nilai kemiripan antara *query* yang diberikan dengan *tuple tree* harus dihitung untuk dapat memberikan peringkat pada setiap *tuple tree* (Liu, 2006, pp. 5).

$Sim(Q, T) = \sum_{k \in Q,T} weight(k, Q) * weight(k, T)$ …….. rumus 2.8

Anggap *T* sebagai *tuple tree* dan {*D1, D2, ..., Dm*} sebagai *text column* di dalam *T*. Setiap nilai *text column Di* didefinisikan sebagai *document* dan *T* sebagai *super-document.* Lalu kita dapat menghitung nilai kemiripan antara *query* dengan *super-document T* seperti yang dijabarkan pada rumus 2.8. Nilai kemiripan merupakan hasil dari perhitungan *dot product* antara vektor *query* dengan vektor *super-document*.

Dari permasalahan ini Liu (2006, pp. 5) melakukan beberapa modifikasi perhitungan nilai kemiripan yang telah dibuat sebelumnya oleh Hriditis (2003), untuk menyesuaikan dengan kondisi yang terjadi.

$weight(k, T) = \sum_{Di \in T} weight(k, Di)/size(T)$ …….. rumus 2.9

Empat fungsi normalisasi untuk menghitung nilai kemiripan antara *query* Q dengan *super-document* T yang dikemukakan oleh Liu (2006, pp. 5-7) adalah sebagai berikut.

$$weight(k, Di) = \frac{ntf * idfg}{ndl * Nsize(T)} \quad \text{rumus 2.10}$$

$$weight(k, T) = Comb(weight(k, Di), ..., weight(k, Dm)) \quad \text{rumus 2.11}$$

1. *Tuple Tree Normalization*

   Faktor ukuran *tuple tree*, yaitu *size(T)*, sangat mirip dengan panjang dokumen *(dl)*. *Tuple tree* yang memiliki banyak *tuple* cenderung memiliki banyak istilah dan memiliki frekuensi istilah yang tinggi sehingga penggunaan rumus *size(T)* yang biasa dirasa kurang optimal, khususnya pada *query* yang kompleks dimana jawaban yang relevan melibatkan banyak *tuple*.

$$Nsize(T) = (1 - s) + s * \frac{size(T)}{avgsize} \quad \text{rumus 2.12}$$

2. *Document Length Normalization*

   Faktor panjang dokumen harus juga diperhitungkan karena secara logika nilai perhitungan panjang dokumen didapatkan dengan menggabungkan beberapa dokumen menjadi satu *super-document T*. Perhitungan awal normalisasi dari panjang dokumen menggunakan konsep *local collection*.

Namun disini terjadi kekurangan karena bobot istilah pada dokumen yang panjang seharusnya bernilai lebih kecil. Oleh karena itu, salah satu solusi yang paling memungkinkan adalah menggunakan konsep *global collection*, yaitu dengan nilai tunggal dari *global avgdl*.

$$ndl = ((1 - s) + s * \frac{dl}{avgdl}) * (1 + \ln(avgdl)) \quad \text{..... rumus 2.13}$$

3. *Document Frequency Normalization*

Frekuensi dokumen juga mengalami masalah yang sama, yaitu masalah *local* dengan *global collections*. Oleh karena itu, perhitungan frekuensi dokumen menggunakan *global document statistic*, dimana df^g adalah *global document frequency* dari istilah (jumlah dokumen di dalam *database* dimana istilah tersebut muncul), dan N^g adalah jumlah keseluruhan dari dokumen di dalam *database*.

$$idf^g = \ln \frac{Ng}{dfg+1} \quad \text{..... rumus 2.14}$$

4. *Inter-Document Weight Normalization*

Dari tiga normalisasi yang telah dijabarkan diatas, bobot istilah di dalam *document Di* pada *T* sudah dapat dihitung. Untuk menghitung nilai *Comb()*, kita dapat menjumlahkan semua bobot istilah pada setiap *document* pada *T*. Istilah cenderung lebih sering muncul pada *T* yang memiliki ukuran yang lebih besar. Rumus 2.15 digunakan untuk menormalisasi *weight(k, T)*,

dimana max*Wgt* merupakan nilai maksimum dari *weight(k, Di)* untuk semua *Di* di dalam *T*.

$$Comb() = \max Wgt * (1 + \ln(1 + \ln \frac{sumWgt}{maxWgt})) \quad \text{..... rumus 2.15}$$

2.4 Nazief-Adriani Stemmer

Agusta (2009, pp. 1) mendefinisikan *stemming* sebagai suatu cara yang digunakan untuk meningkatkan performa *information retrieval* dengan cara mentransformasikan kata-kata dalam sebuah dokumen teks ke kata dasarnya. Menurut Frakes (1992) bahwa selain digunakan untuk meningkatkan efektivitas *retrieval*, *stemming* juga dapat digunakan mengurangi ukuran dari *file indexing*.

Menurut Agusta (2009, pp. 1), algoritma *stemming* untuk bahasa yang satu berbeda dengan algoritma *stemming* bahasa lainnya. Sebagai contoh bahasa inggris memiliki morfologi yang berbeda dengan bahasa indonesia sehingga algoritma *stemming* untuk kedua bahasa tersebut juga berbeda. Proses *stemming* pada teks berbahasa indonesia lebih rumit/kompleks karena terdapat variasi imbuhan yang harus dibuang untuk mendapatkan kata dasar dari sebuah kata. Penggunaan algoritma *stemming* yang sesuai akan mempengaruhi performa sistem *information retrieval*.

Dalam penelitian yang telah dilakukan oleh Asian (2005, pp. 5) terhadap 5 buah algoritma *stemmer* berbahasa indonesia (*Nazief-Adriani*, *Ahmad*, *Idris*, *Arifin*, *Vega*), disimpulkan bahwa algoritma *Nazief-Adriani* memiliki peringkat paling tinggi dalam menghasilkan kata dasar yang benar, yaitu sebesar 93%.

Tala (2003, pp. 3) menjelaskan bahwa morfologi kata-kata dalam bahasa Indonesia dapat meliputi dua struktur, yaitu infleksi dan derivasi. Infleksi adalah struktur paling sederhana dimana suatu kata ditambahkan suatu akhiran (*suffixes*) tanpa mengubah makna dari kata dasarnya. Infleksi akhiran ini dapat terbagi menjadi dua kelompok:

1. Akhiran (*-lah, -kah, -pun, -tah*). Jenis akhiran ini disebut sebagai partikel atau kata fungsional yang tidak memiliki arti. Kehadiran partikel ini hanya digunakan untuk penekanan saja, contohnya:

   dia + kah ⇒ diakah (penekanan untuk sebuah pertanyaan)

   saya + lah ⇒ sayalah (untuk menekankan)

2. Akhiran (*-ku, -mu, -nya*). Jenis akhiran ini digunakan untuk membentuk kata ganti milik (*possesive pronouns*), contohnya:

   tas + mu ⇒ tasmu

   sepeda + ku ⇒sepedaku

Setiap akhiran pada kedua kelompok tersebut dapat muncul bersamaan pada kata yang sama (Tala, 2003, pp. 4). Ketika mereka berdua muncul, mereka harus mengikuti aturan dimana akhiran dari kelompok kedua akan mendahului kelompok pertama yang dijabarkan pada rumus 2.16.

inflectional := (root + possesive_pronouns) |

(root + particle) |

(root + possesive_pronouns + particle) …….rumus 2.16

Struktur derivasi dalam bahasa Indonesia mengandung awalan (*prefixes*), akhiran (*suffixes*) dan kombinasi keduanya (*confixes*). Awalan yang paling sering

muncul adalah *ber-, di-, ke-, meng-, peng-, per-, ter-* (Tala, 2003, pp. 4). Berikut ini adalah beberapa contoh penggunaan awalan (prefix):

ber + lari ⇒ berlari

di + makan ⇒ dimakan

ke + kasih ⇒ kekasih

meng + ambil ⇒ mengambil

peng + atur ⇒ pengatur

per + lebar ⇒ perlebar

ter + baca ⇒ terbaca

Tala (2003, pp. 4) menyatakan bahwa beberapa awalan seperti *ber-, meng-, peng-, per-, ter-* dapat muncul dalam beberapa bentuk berbeda. Bentuk dari masing-masing awalan ini tergantung pada karakter pertama dari kata yang akan ditempelkan. Tidak seperti struktur infleksi, pada struktur ini pengejaan kata dapat berubah ketika awalan ditambahkan. Contohnya seperti pada kata "menyapu" yang dibentuk dari awalan "meng-" dan kata dasar "sapu". Awalan "meng-" berubah menjadi "meny-" dan karakter pertama dari kata dasar meluluh. Aturan penambahan awalan ini dapat dilihat pada tabel 2.1.

Tabel 2.1 Aturan Derivasi pada Penambahan Awalan

Aturan	Format Kata	Pemenggalan
1	berV...	ber-V... | be-rV...
2	berCAP...	ber-CAP... dimana C!='r' & P!='er'
3	berCAerV...	ber-CaerV... dimana C!='r'
4	belajar	bel-ajar
5	beC1erC2...	be-C1erC2... dimana C1!={'r'|'l'}

Tabel 2.1 Aturan Derivasi pada Penambahan Awalan (lanjutan)

Aturan	Format Kata	Pemenggalan
6	terV...	ter-V... | te-rV...
7	terCerV...	ter-CerV... dimana C!=’r’
8	terCP...	ter-CP... dimana C!=’r’ dan P!=’er’
9	teC1erC2...	te-C1erC2... dimana C1!=’r’
10	me{l|r|w|y}V...	me-{l|r|w|y}V...
11	mem{b|f|v}...	mem-{b|f|v}...
12	mempe...	mem-pe...
13	mem{rV|V}...	me-m{rV|V}... |
14	men{c|d|j|z}...	men-{c|d|j|z}...
15	menV...	me-nV... | me-tV
16	meng{g|h|q|k}.	meng-{g|h|q|k}...
17	mengV...	meng-V... | meng-kV...
18	menyV...	meny-sV...
19	mempV...	mem-pV... dengan V!=’e’
20	pe{w|y}V...	pe-{w|y}V...
21	perV...	per-V... | pe-rV...
22	perCAP	per-CAP... dimana C!=’r’ dan P!=’er’
23	perCAerV...	per-CAerV... dimana C!=’r’
24	pem{b|f|V}...	pem-{b|f|V}...
25	pem{rV|V}...	pe-m{rV|V}... |
26	pen{c|d|j|z}...	pen-{c|d|j|z}...
27	penV...	pe-nV... | pe-tV...
28	peng{g|h|q}...	peng-{g|h|q}...
29	pengV...	peng-V... | peng-kV...
30	penyV...	peny-sV...
31	pelV...	pe-lV... kecuali “pelajar” yang menghasilkan “ajar”
32	peCerV...	per-erV... dimana C!={r|w|y|l|m|n}
33	peCP...	pe-CP... dimana C!={r|w|y|l|m|n} dan P!=’er’

Sumber: Penggunaan Algoritma Semut dan Confix Stripping Stemmer Untuk Klasifikasi Dokumen Berita Berbahasa Indonesia (2008, pp. 3)

Selain derivasi pada akhiran. Struktur derivasi lainnya yang perlu diperhatikan adalah derivasi akhiran *-i, -kan, -an*. Contoh derivasi tersebut:

gula + i ⇒ gulai

makan + an ⇒ makanan

beri + kan ⇒ berikan

Berbeda dengan awalan, penambahan pada akhiran tidak pernah mengubah pengejaan kata dasar pada kata yang diderivasi (Tala, 2005, pp. 4). Struktur derivasi lainnya adalah konfiks, yaitu gabungan antara awalan dan akhiran yang ditambahkan pada kata dasar untuk menghasilkan derivasi kata baru, contohnya:

per + main + an ⇒ permainan

ke + menang + an ⇒ kemenangan

ber + jatuh + an ⇒ berjatuhan

meng + ambil + kan ⇒ mengambilkan

Tidak seluruh kombinasi dari awalan dan akhiran dapat digabungkan untuk membentuk konfiks. Ada beberapa kombinasi dari awalan dan akhiran yang tidak diperbolehkan. Tabel 2.2 berisi semua daftar konfiks yang tidak diperbolehkan.

Tabel 2.2 Pasangan Konfiks yang Tidak Diperbolehkan

Awalan (prefix)	**Akhiran (Suffix)**
ber	i
di	an
ke	i | kan
meng	an
peng	i | kan
ter	An

Sumber: A Study of Stemming Effects on Information Retrieval in Bahasa Indonesia (2003, pp. 5)

Sebuah awalan/konfiks dapat ditambahkan ke kata yang sudah memiliki awalan/konfiks, dimana menghasilkan *double prefix structure*. Sama seperti pembentukkan konfiks, tidak semua awalan/konfiks dapat ditambahkan pada kata tertentu yang sudah memiliki awalan/konfiks. Tabel 2.3 menjelaskan lebih lanjut bagaimana aturan urutan penambahan awalan/konfiks yang diperbolehkan.

Tabel 2.3 Urutan *Double Prefix*

Awalan (Prefix) 1	**Awalan (Prefix) 2**
meng	per
di	ber
ter	
ke	

Sumber: A Study of Stemming Effects on Information Retrieval in Bahasa Indonesia (2003, pp. 5)

derivational := prefixed | suffixed | confixed | double_prefix

where

prefixed := prefix + root

suffixed := root + suffix

confixed := prefix + root + suffix

double_prefix := (prefix + prefixed) | (prefixed + confixed) | (prefix + prefixed + suffix) …… rumus 2.17

Algoritma *Stemming Nazief-Adriani* dikembangkan pada aturan morfologi bahasa Indonesia yang mengelompokkan dan mengekapsulasi imbuhan-imbuhan, termasuk di dalamnya adalah awalan (*prefix*), sisipan (*infix*), akhiran (*suffix*), dan gabungan awalan-akhiran (*confixes*). Algoritma ini

menggunakan kamus kata dasar dan mendukung *recoding*, yaitu penyusunan kembali kata-kata yang mengalami proses *stemming* terlebih dahulu.

Berikut ini penjelasan mengenai tahapan-tahapan pada algoritma *Nazief-Adriani* yang dikemukakan oleh Keke (2012, pp. 3):

1. Kata yang hendak di-*stemming* dicari terlebih dahulu di kamus. Jika kata ditemukan dalam kamus, berarti kata tersebut sudah berbentuk kata dasar (*root word*). Algoritma berhenti, jika tidak maka tahapan selanjutnya akan dilakukan.
2. Membuang akhiran infleksi (*inflection suffixes*) *-lah, -kah, -mu,* atau *–nya*. Jika berupa partikel *–lah, -kah, -tah, atau –pun* maka langkah ini diulangi lagi untuk menghapus kata ganti milik (*possesive pronouns) -ku, -mu,* atau *–nya*, jika ada.
3. Menghilangkan akhiran derivasi (*derivation suffixes*) *–i, -kan,* atau *–an*. Jika kata ditemukan di dalam kamus, maka algoritma berhenti. Jika tidak maka ke langkah 3a.
    a. Jika *–an* telah dihapus dan huruf terakhir dari kata tersebut adalah *–k*, maka *–k* juga dihapus. Jika kata tersebut ditemukan dalam kamus maka algoritma berhenti. Jika tidak ditemukan maka lakukan langkah 3b.
    b. Akhiran yang dihapus *-i, -an,* atau *-kan* dikembalikan, lanjut ke langkah 4.
4. Menghilangkan awalan derivasi (*derivation prefixes*) *di-, ke-, se-, me-, be-, pe-,* atau *te-* dengan iterasi maksimum 3 kali:

a. Langkah 4 berhenti jika:
    1. Terjadi kombinasi awalan dan akhiran yang terlarang seperti pada tabel 2.2.
    2. Awalan yang dideteksi saat ini sama dengan awalan yang dihilangkan sebelumnya.
    3. Tiga awalan telah dihilangkan.
b. Identifikasi tipe awalan dan hilangkan. Awalan ada dua jenis:
    1. Awalan standar *di-*, *ke-*, *se-* yang dapat langsung dihilangkan dari kata.
    2. Awalan kompleks *me-*, *be-*, *pe-*, *te-* adalah tipe-tipe awalan yang dapat bermorfologi sesuai kata dasar yang mengikutinya. Oleh karena itu, gunakan aturan pada tabel 2.1 untuk mendapatkan pemenggalan yang tepat.
    3. Cari kata yang telah dihilangkan awalannya ini di dalam kamus. Apabila tidak ditemukan, maka langkah 4 diulangi kembali. Apabila ditemukan, maka keseluruhan proses dihentikan.

5. Apabila setelah langkah 4 kata dasar masih belum ditemukan, maka proses *recoding* dilakukan dengan mengacu pada aturan pada tabel 2.1. *Recoding* dilakukan dengan menambahkan karakter *recoding* di awal kata yang dipenggal. Pada tabel 2.1, karakter *recoding* adalah huruf kecil setelah tanda hubung (‘-’) dan terkadang berada sebelum tanda kurung. Sebagai contoh kata “menangkap” (aturan 15), setelah dipenggal menjadi “nangkap”.

Karena tidak valid, maka *recoding* dilakukan dan menghasilkan kata "tangkap".

6. Jika semua langkah telah selesai tetapi tidak juga berhasil maka kata awal diasumsikan sebagai kata dasar dan proses selesai.

Keke (2012, pp. 11) dalam penelitiannya mengemukakan beberapa kelemahan *Nazief-Adriani* sebagai berikut.

1. Penyamarataan makna variasi kata.
2. Jumlah *database* kata dan kata dasarnya harus besar. Kesalahan terjadi bila kata tidak ditemukan di *database* dan kemudian dianggap kata dasar, padahal bukan.
3. Lamanya waktu yang diperlukan dalam proses pencarian kata di dalam kamus.

2.5 Bloom Filter

Bloom filter merupakan struktur data probablistik yang *space-efficient* guna mendukung *membership queries* (Tarkoma, 2010, pp. 1). Menurut Mitzenmacher (2002, pp. 1), *Bloom Filter* secara sederhana dapat dikatakan sebagai suatu *bit array* acak dimana memiliki probabilitas untuk menghasilkan *false positive* yang menentukan apakah suatu elemen ada di dalam *set* tersebut atau tidak. Hal ini membuat *Bloom Filter* berguna untuk berbagai macam pekerjaan yang melibatkan *list* atau *set*.

Operasi dasar dari *Bloom Filter* meliputi penambahan elemen ke dalam *set* dan *query* kemungkinan adanya suatu elemen dalam suatu *list* atau *set*. Tingkat

akurasi dari *Bloom Filter* bergantung pada ukuran dari *filter*, jumlah fungsi *hash* yang dipergunakan pada *filter*, dan jumlah elemen yang dimasukkan ke dalam *set*. Semakin banyak elemen yang ditambahkan ke dalam *Bloom Filter* akan meningkatkan kemungkinan hasil *query* yang *false positives* (Wendy, 2012).

Mill (2013) pada halaman webnya menggambarkan secara sederhana mengenai konsep *Bloom Filter*. Dia menggunakan *bit array* sebanyak 15 *bit* yang digambarkan dalam sebuah tabel dimana setiap selnya memiliki nomor *bit*. Untuk menambahkan elemen ke dalam set, elemen tersebut akan di-*hashing* (fungsi *hash* fnv dan murmur) beberapa kali dan akan diberikan nilai 1 pada sel *bit array* sesuai dengan nomor yang dihasilkan dari proses *hashing* tersebut.

Saat melakukan *membership query*, elemen yang akan dicari akan melalui proses *hashing* untuk mendapatkan posisi sel pada *bit array*. Jika posisi sel tersebut bernilai 1, maka sistem akan memberitahukan bahwa elemen tersebut mungkin ada di dalam *set*. Akan tetapi jika pada posisi sel tersebut bernilai nol maka sistem akan meberitahukan bahwa elemen tersebut tidak ada di dalam *set*. Semakin banyak elemen yang dimasukkan ke dalam *set*, probabilitas untuk menghasilkan *false positive* semakin tinggi.

Tarkoma (2010, pp. 2) dalam penelitiannya mengemukakan penggunaan konsep *Bloom Filter* dalam *membership query*.

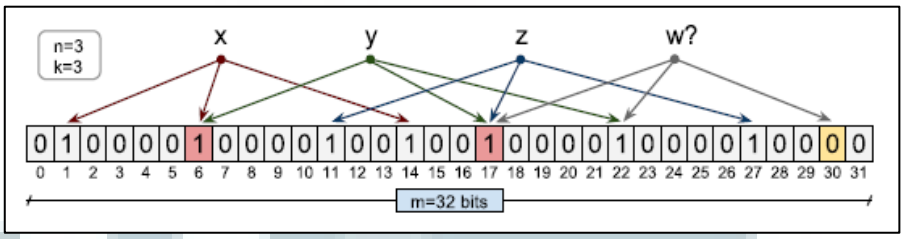
1. *Bloom Filter* terdiri dari *m bit array* yang digunakan untuk merepresentasikan *set S* = {*x1, x2, …, xn*} dari *n* elemen.
2. Seluruh *bit array* diinisialisasikan dengan nilai nol.

3. Tujuan penggunaan fungsi *k hash* dimana *hi(x), 1 ≤ i ≤ k* adalah untuk memetakan setiap *item x ∈ S* ke dalam nomor acak yang seragam dalam jarak 1, … *m*. Fungsi *hash* diasumsikan sama (algoritma *hash* MD5 salah satu pilihan populer dari banyaknya fungsi hash yang ada).
4. Ketika suatu elemen *x ∈ S* ditambahkan ke dalam *filter* maka posisi *bit hi(x)* akan diberikan nilai 1 untuk *1 ≤ i ≤ k.*
5. *y* diasumsikan sebagai salah satu anggota dari *S* jika posisi *bit hi(x)* telah di-*set* dengan nilai 1, dan dianggap bukan merupakan anggota dari *S* jika posisi dari *bit hi(x)* belum pernah di-*set* atau bernilai 0.

Tarkoma (2010, pp. 2) menjelaskan proses yang terjadi pada *Bloom Filter* dengan ilustrasi sebagai berikut.

1. Diilustrasikan *Bloom Filter* menggunakan *bit array* dengan panjang 32 *bit* dan diinisialisasikan dengan nilai 0.
2. Tiga buah elemen (*x, y, z*) ditambahkan ke dalam *set* dimana setiap elemen kemudian di-*hashing* menggunakan k = 3 fungsi *hash* untuk mendapatkan posisi *bit*. *Bit* yang terkait dengan elemen tersebut sekarang telah bernilai 1 (gambar 2.4).
3. Ketika elemen *w* akan dicari, elemen tersebut akan di-*hashing* menggunakan 3 fungsi *hash* yang sama dengan sebelumnya untuk mendapatkan posisi *bit*.
4. Pada kasus ini salah satu posisi dari *w* bernilai 0, kemudian *Bloom Filter* akan melaporkan bahwa elemen tersebut tidak ada di dalam *set*.
5. Ada kemungkinan dimana seluruh posisi *bit* telah terisi dengan nilai 1. Ketika hal ini terjadi *Bloom Filter* akan memberikan laporan yang salah

bahwa elemen tersebut merupakan anggota dari *set*. Pelaporan yang salah ini sering disebut dengan *false positive*.

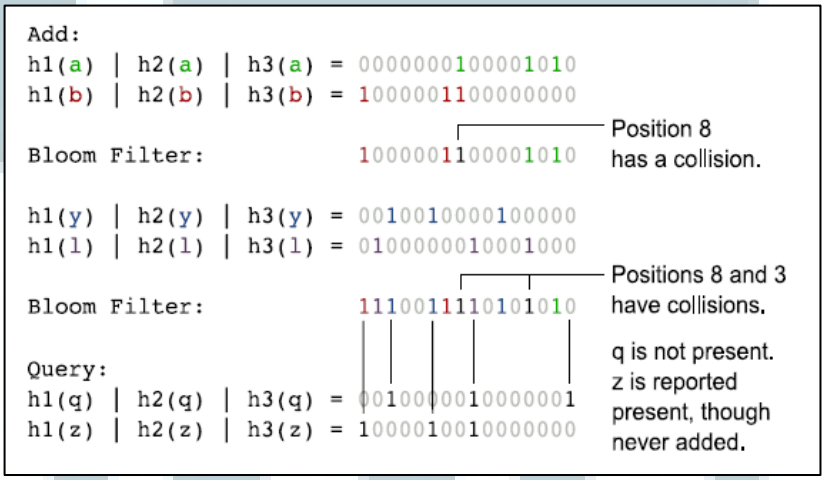


Gambar 2.4 Ilustrasi *Bloom Filter*
Sumber: Theory and Practice of Bloom Filters for Distributed System (2010, pp. 2)

Selain itu Tarkoma (2010, pp. 2) juga menjelaskan proses penambahan dan *query* elemen dengan ilustrasi sebagai berikut.

1. Diilustrasikan *Bloom Filter* menggunakan *bit array* dengan panjang 16 *bit* dan diinisialisasikan dengan nilai 0. Posisi *bit* dinomori dari 0 sampai 15, dari kanan ke kiri.
2. Tiga fungsi *hash* (*h1, h2, h3*) yang digunakan adalah MD5, SHA1, CRC32 secara berurutan.
3. Elemen yang akan ditambahkan merupakan 1 buah karakter dari string teks.
4. Ketika menambahkan elemen, nilai dari fungsi *h1* sampai *h3* (*modulus* 16) dihitung dan posisi *bit* yang terkait akan diberi nilai 1.
5. Setelah menambahkan elemen *a* dan *b*. Posisi 15, 9, 8, 3, dan 1 telah di-*set* dengan nilai 1 oleh *Bloom Filter*. Dalam kasus ini kedua elemen memiliki posisi yang sama yaitu pada nomor 8.
6. Setelah menambahkan elemen *y* dan *l*. Posisi 15, 14, 13, 10, 9, 8, 7, 5, 3 dan 1 telah di-*set* dengan nilai 1.

7. Ketika elemen *z* dan *q* ingin dicari, kedua elemen tersebut akan di-*hashing* untuk mencari tahu posisi *bit*-nya. Jika tiga bit dari posisi tersebut telah di-*set* dengan nilai 1 maka elemen tersebut dianggap ada di dalam *set*. Pada kasus *q*, posisi *bit* 0 belum di-*set* sehingga elemen *q* dianggap bukan termasuk salah satu anggota *set*. Sedangkan untuk elemen *z*, semua posisi bit yang terkait telah di-*set* dengan nilai 1 sehingga elemen tersebut dianggap sebagai salah satu dari anggota *set*. Laporan mengenai *z* merupakan *false positive* karena elemen yang sesungguhnya tidak ada di dalam set dilaporkan ada (gambar 2.5).



Gambar 2.5 Penambahan dan Query Menggunakan *Bloom Filter*
Sumber: Theory and Practice of Bloom Filters for Distributed System (2010, pp.2)

Bloom Filter dibangun berdasarkan pada *S* yang memiliki ruang *O(n)* dan dapat menjawab *query* keanggotaan dalam waktu *O(1)*. Bila $x \in S$, *Bloom Filter* akan selalu melaporkan bahwa *x* merupakan anggota *S*. Tetapi bila $y \notin S$, ada

kemungkinan bahwa *Bloom Filter* akan melaporkan bahwa $y \in S$ (Tarkoma, 2010, pp. 3).

Tabel 2.4 Parameter Kunci *Bloom Filter*

Parameter	**Penambahan**
Jumlah fungsi *hash* (*k*)	Waktu komputasi bertambah, *false positive rate* berkurang *k→kopt*
Ukuran *filter* (*m*)	Ukuran bertambah, *false positive rate* berkurang
Jumlah elemen di dalam *set* (*n*)	Jumlah elemen bertambah, *false positive rate* bertambah

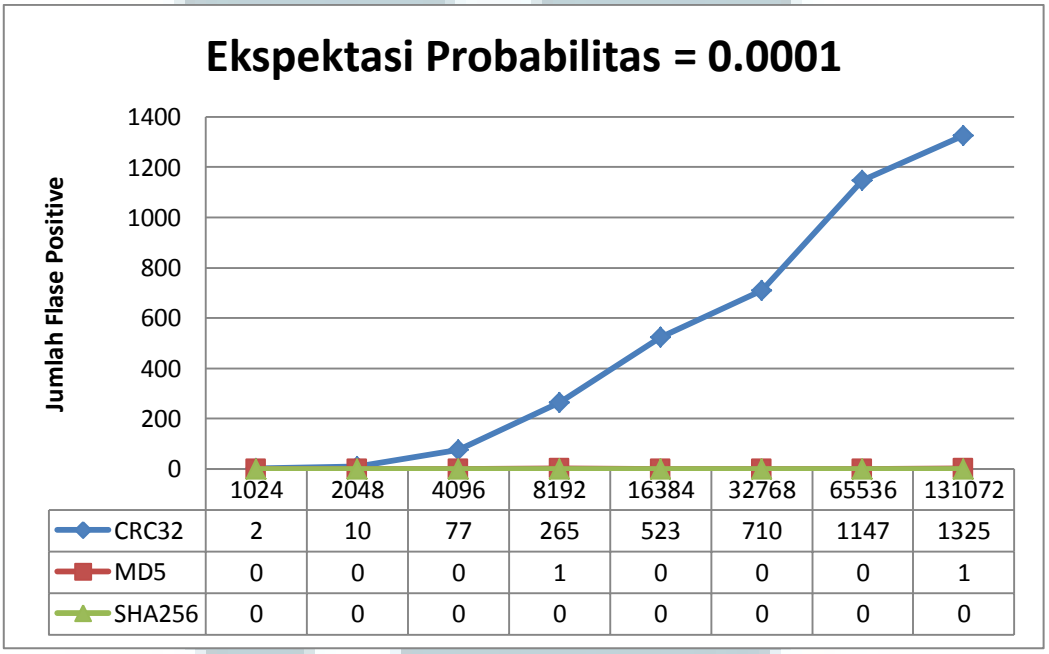
Sumber: Theory and Practice of Bloom Filters for Distributed System (2010, pp. 3)

Untuk menghitung nilai probabilitas *false positive* dari *Bloom Filter* dan jumlah optimal dari penggunaan fungsi *hash* dapat dimulai dengan mengasumsikan bahwa setiap fungsi *hash* memilih setiap posisi di dalam *set* dengan probabilitas yang sama. *m* dianggap sebagai jumlah bit di dalam *Bloom Filter*, ketika elemen ditambahkan ke dalam *filter*, probabilitas bahwa *bit* tertentu tidak di-*set* dengan nilai 1 oleh fungsi *hash* adalah

$1 - \frac{1}{m}$ ..... rumus 2.18

Berdasarkan penelitian yang telah dilakukan Wendy (2012, pp. 6) terhadap beberapa fungsi *hash* (CRC32, MD5, SHA256), fungsi *hash* optimal dengan harapan probabilitas tertinggi (0.0001) adalah SHA256. Dengan hasil

tersebut maka SHA256 merupakan fungsi *hash* terbaik ketika mengimplementasikan *Bloom Filter* karena memiliki *false positive* terkecil dan tidak memiliki *false negative* (Wendy, 2012, pp. 6).



	1024	2048	4096	8192	16384	32768	65536	131072
CRC32	2	10	77	265	523	710	1147	1325
MD5	0	0	0	1	0	0	0	1
SHA256	0	0	0	0	0	0	0	0

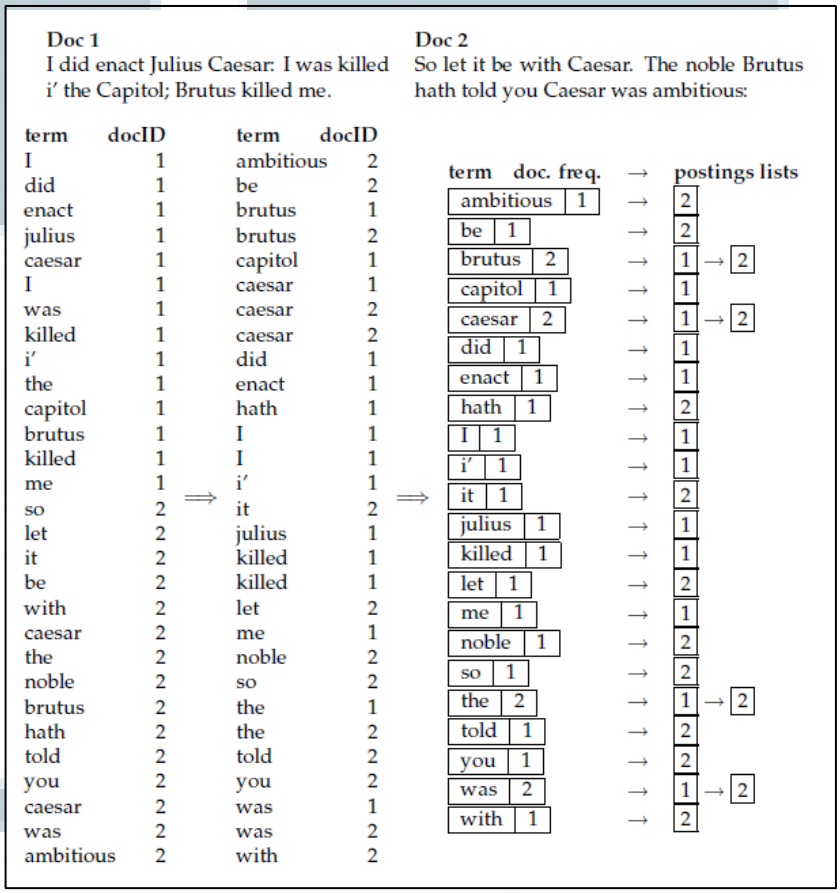
Gambar 2.6 Grafik Jumlah *False Positive* pada *Expected Probability* (eP) 0.0001
Sumber: Optimasi Query Pencarian Menggunakan algoritma Bloom Filter Pada Sistem Sekolah (2012, pp. 3)

2.6 Index Construction

Manning (2009, pp. 69-81) dalam bukunya menjelaskan beberapa jenis algoritma *indexing* yang umumnya digunakan pada *information retrieval system*, diantaranya:

1. *Block sort-based indexing*. Langkah dasar dalam membangun *nonpositional index* digambarkan pada gambar 2.7. Diawali dengan memproses seluruh koleksi, lalu merangkai semuanya menjadi pasangan *term-docID*. Kemudian pasangan dari *term-docID* ini diurutkan dengan aturan dimana *term* sebagai

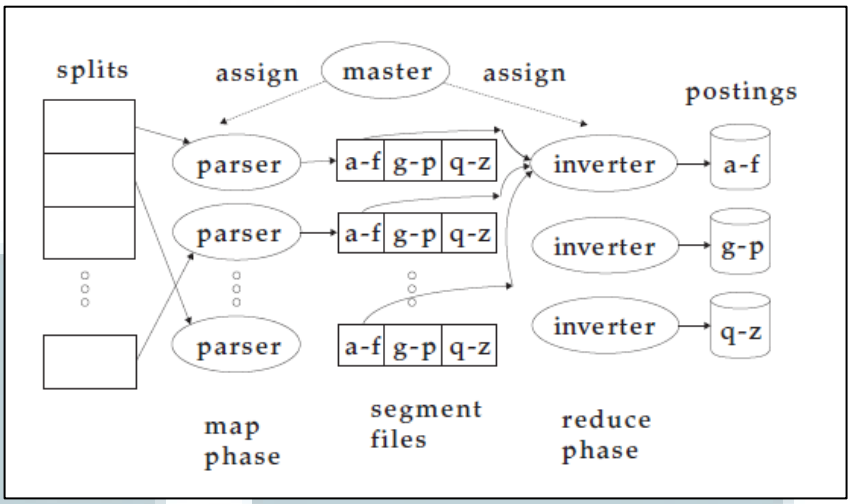
dominant key sedangkan *docID* sebagai *secondary key*. Terakhir *docID* dari setiap *term* dikelola ke dalam sebuah *posting list* dengan menghitung statistik dari *term* dan *document frequency*. Algoritma ini sangat efektif ketika ukuran dari koleksi tidak teralu besar. Namun jika ternyata ukurannya cukup besar, kita membutuhkan algoritma yang lebih baik, yaitu *block sort-based indexing*. Algoritma ini membagi koleksi kedalam ukuran yang sama lalu mengurutkan *posting* (*termID, docID*) dan menyimpan *intermediate result* ke dalam *disk*. Terakhir menggabungkan keseluruhan *intermediate result* untuk mendapatkan *index*.



Gambar 2.7 Membangun *index* dengan mengurutkan dan mengelompokkan
Sumber: An Introduction to Modern Information Retrieval (2009, pp. 8)

2. *Single-pass in-memory indexing*. Algoritma ini lebih baik dari algoritma sebelumnya. Ide utama dari algoritma ini adalah memecah koleksi ke dalam beberapa bagian, kemudian dari setiap bagian akan dibentuk menjadi *dictionary* yang berisi *term*. Untuk setiap *token* yang merupakan milik *dictionary* tersebut akan diambil *posting* yang dimilikinya, jika tidak *posting list* kosong akan dibuatkan untuk *token* tersebut. *Term* kemudian diurutkan lalu blok dan *dictionary* ditulis ke dalam *disk*.
3. *Distributed Indexing*. *information retrieval system* yang memiliki ukuran koleksi cukup besar tidak dapat hanya mengandalkan 1 buah mesin saja untuk membangun indeks. Oleh karena itu, tugas ini dibagi ke dalam beberapa mesin. Kita dapat menentukan partisi berdasarkan pada *posting* atau kata kunci (*document-partition* atau *term-partition*). Ide utama dari algoritma ini adalah menggunakan mesin *cluster* dimana tiap mesin ini dapat direpresentasikan sebagai suatu *node* yang mengerjakan *sub-task* tertentu. Sebuah *node* dapat dimungkinkan untuk mengalami kegagalan, dalam kasus ini *sub-task* dari *node* tersebut akan direlokasikan kepada *node* lainnya. Arsitektur dari model *indexing* ini menggunakan konsep *MapReduce* dimana koleksi dipecah menjadi beberapa bagian untuk meningkatkan efisiensi. Pada *map phase* setiap bagian memproses *posting list* menggunakan *parser*. Hasil dari *parsing* tersebut disimpan ke dalam *local intermediate files* yang dinamakan *segment files*. Setiap *segment files* inilah yang mengandung *posting* dari beberapa istilah (*term-partition*). Pada *reduce phase term-partition* akan diproses menggunakan *inverter* untuk

mengumpulkan *posting* mana saja yang saling terkait dari dalam *segment files*.



Gambar 2.8 Ilustrasi *Distributed Indexing* Menggunakan MapReduce
Sumber: An Introduction to Modern Information Retrieval (2009, pp. 76)

4. *Dynamic Indexing*. dari beberapa model *indexing* yang telah dijelaskan sebelumnya, semuanya sangat efisien jika perubahan dalam koleksi tidak teralu sering terjadi. Namun jika dokumen berubah secara periodik, hal ini akan menjadi masalah besar. Oleh karena itu, model *indexing* ini secara periodik membangun indeks yang baru dan menyimpannya dalam *auxiliary index*. *Auxiliary index* juga berguna dalam mengurangi *disk seeks*.

2.7 CodeIgniter

CodeIgniter adalah suatu *application development framework* yang digunakan untuk pembuatan *website* dengan bahasa pemrograman PHP. Tujuan utama dari CodeIgniter yaitu, memungkinkan kita untuk mengerjakan suatu proyek dengan lebih cepat dibanding memulai semuanya dari awal dengan

menyediakan sekumpulan *libraries* yang dapat digunakan untuk mengerjakan berbagai macam tugas pada umumnya (EllisLab, 2013).

Menurut Li (2011) beberapa keuntungan dari penggunaan CodeIgniter adalah sebagai berikut.

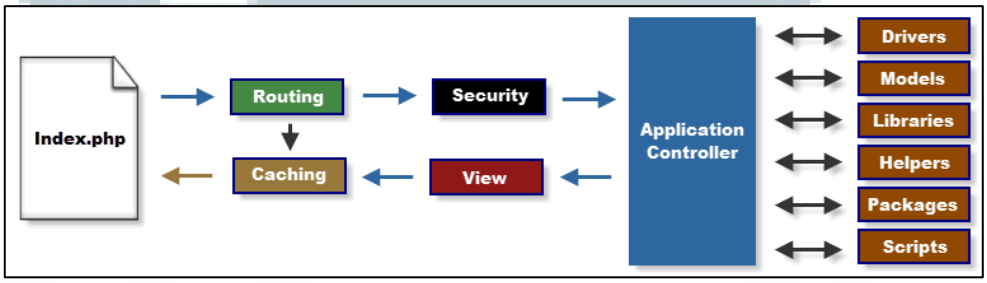
1. Kecil, cepat, sederhana, dan mudah untuk dipelajari.
2. Memudahkan kita untuk melakukan migrasi dari satu *server* ke *server* lainnya.
3. CodeIgniter mudah dalam instalasi, hanya membutuhkan beberapa menit saja.
4. Mudah dalam melakukan *debug.*
5. Implementasi *active record* yang cukup baik dan mudah diingat.
6. Kumpulan *libraries* yang cukup baik.
7. Fitur utama yang terpenting adalah CodeIgniter terdokumentasi dengan baik.

CodeIgniter dibuat berdasarkan *Model-View-Controller development pattern.* MVC adalah suatu bentuk pendekatan perangkat lunak yang memisahkan logika aplikasi dari presentasinya. Dalam prakteknya, hal ini dapat memungkinkan suatu halaman *web* hanya berisi sedikit kode PHP karena adanya pemisahan presentasi dengan PHP *scripting* (EllisLab, 2013). Berikut ini beberapa penjelasan mengenai konsep MVC pada CodeIgniter.

1. *Model* menggambarkan suatu struktur data. Biasanya setiap kelas *model* akan mengandung beberapa fungsi untuk membantu dalam mengambil, menambahkan, dan memperbaharui informasi di dalam *database*.

2. *View* merupakan informasi yang akan dipresentasikan kepada pengguna. Sebuah *view* pada umumnya dianggap sebagai halaman *web*, tetapi di dalam CodeIgniter, *view* dapat juga dianggap sebagai bagian halaman seperti *header* atau *footer*. Dapat juga sebagai halaman RSS, atau jenis lain dari "halaman".
3. *Controller* bertugas sebagai perantara antara *model*, *view*, dan *resource* lain yang dibutuhkan dalam memproses *HTTP request* dan membangun suatu halaman *web*.

Dengan memahami alur kerja dari *framework* CodeIgniter dapat memudahkan kita dalam memahami cara kerja CodeIgniter secara lebih baik. Skema berikut akan menjelaskan alur kerja *framework* CodeIgniter.



Gambar 2.9 Skema Alur Kerja CodeIgniter
Sumber: EllisLab (2013).

1. index.php sebagai *front controller* melakukan kegiatan inisialisasi semua *base resources* yang dibutuhkan untuk menjalankan CodeIgniter.
2. *Router* melakukan kegiatan pemeriksaan pada setiap *HTTP request* untuk menentukan apa yang harus dikerjakan.
3. Jika *cache file* tersedia, maka file tersebut akan langsung dikirimkan ke *browser* dengan cara melakukan *bypassing the normal execution system*.

4. *Security*. Sebelum aplikasi *controller* dipanggil, HTTP *request* dan semua *user data* yang dikirim akan di-*filter* sebagai salah satu bagian dari keamanan.
5. *Controller* akan memanggil *model, core libraries, helpers*, dan *resources* lainnya yang dibutuhkan untuk memproses setiap permintaan.
6. Hasil akhir dari *view* akan dimuat kemudian akan dikirim ke *browser* untuk ditampilkan. Jika fitur *caching* diaktifkan, maka *view* akan di-*cached* terlebih dahulu sehingga dapat melayani permintaan berikutnya.