# KOMPRESI *INVERTED LIST* MENGGUNAKAN KOMBINASI *GOLOMB* DAN *HUFFMAN* BERDASARKAN *CONTIGUOUS SEQUENTIAL PATTERNS* (CSP)

**SKRIPSI**


oleh :

**MUHAMMAD RIFAI**

**0510960040-96**



**PROGRAM STUDI ILMU KOMPUTER**

**JURUSAN MATEMATIKA**

**FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM**

**UNIVERSITAS BRAWIJAYA**

**MALANG**


**2012**

UNIVERSITAS BRAWIJAYA

# KOMPRESI *INVERTED LIST* MENGGUNAKAN KOMBINASI *GOLOMB* DAN *HUFFMAN* BERDASARKAN *CONTIGUOUS SEQUENTIAL PATTERNS* (CSP)

**SKRIPSI**

Sebagai salah satu syarat untuk memperoleh gelar
Sarjana Komputer dalam bidang Ilmu Komputer


oleh :
**MUHAMMAD RIFAI**
**0510960040-96**



**PROGRAM STUDI ILMU KOMPUTER**
**JURUSAN MATEMATIKA**
**FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM**
**UNIVERSITAS BRAWIJAYA**
**MALANG**

**2012**


UNIVERSITAS BRAWIJAYA

**LEMBAR PENGESAHAN SKRIPSI**

**KOMPRESI *INVERTED LIST* MENGGUNAKAN KOMBINASI *GOLOMB* DAN *HUFFMAN* BERDASARKAN *CONTIGUOUS SEQUENTIAL PATTERNS* (CSP)**

Oleh :

**MUHAMMAD RIFAI**
**0510960040-96**

**Setelah dipertahankan di depan Majelis Penguji**
**pada tanggal 7 Juni 2012**
**dan dinyatakan memenuhi syarat untuk memperoleh gelar**
**Sarjana Komputer dalam bidang Ilmu Komputer**

| Pembimbing I | Pembimbing II |
|---|---|
| **Dian Eka R., S.Si , M.Kom** | **Yusi Tyroni Mursityo, S.Kom, MS** |
| **NIP. 197306192002122001** | **NIP. 198002282006041001** |

**Mengetahui,**
**Ketua Jurusan Matematika**
**Fakultas MIPA Universitas Brawijaya**

**Dr. Abdul Rouf A., MSc**
**NIP. 196709071992031001**

## LEMBAR PERNYATAAN

**Saya yang bertanda tangan di bawah ini :**

| | |
|---|---|
| **Nama** | **: Muhammad Rifai** |
| **NIM** | **: 0510960040-96** |
| **Jurusan** | **: Matematika** |
| **Program Studi** | **: Ilmu Komputer** |
| **Penulis skripsi berjudul** | **: Kompresi *Inverted List* Menggunakan Kombinasi *Golomb* dan *Huffman* Berdasarkan *Contiguous Sequential Patterns* (CSP)** |

**Dengan ini menyatakan bahwa :**

1. **Isi dari skripsi yang saya buat adalah benar-benar karya sendiri dan tidak menjiplak karya orang lain, selain nama-nama yang termaktub di isi dan tertulis di daftar pustaka dalam skripsi ini.**
2. **Apabila dikemudian hari ternyata skripsi yang saya tulis terbukti hasil jiplakan, maka saya akan bersedia menanggung segala resiko yang akan saya terima.**

**Demikian pernyataan ini dibuat dengan segala kesadaran.**

**Malang, 7 Juni 2012**
**Yang menyatakan,**

**(Muhammad Rifai)**
**NIM. 0510960040-96**

**KOMPRESI *INVERTED LIST* MENGGUNAKAN KOMBINASI *GOLOMB* DAN *HUFFMAN* BERDASARKAN *CONTIGUOUS SEQUENTIAL PATTERNS* (CSP)**

## Abstrak

*Index* merupakan daftar kata yang disertai posisi halaman pada sebuah buku. Secara implementasi, *index* pada koleksi dokumen disimpan dalam *disk*, tetapi dengan semakin bertambahnya jumlah dokumen akan mengakibatkan ruang penyimpanan *index* dalam *disk* menjadi besar. Oleh karena itu, diperlukan efisiensi ruang penyimpanan untuk *index*.

Sistem kompresi *inverted list* ini meliputi beberapa tahap. Tahap pertama yaitu *preproccessing* yang mengekstrak koleksi dokumen menjadi daftar *term* melalui proses *case folding, stop word*, dan *stemming*. Tahap kedua yaitu *indexing* terhadap daftar *term* yang menghasilkan *inverted list*. Tahap ketiga yaitu pendeteksian CSP pada *inverted list* menggunakan algoritma *Apriori* dan diefisiensi menggunakan *UpDown Tree*. Tahap akhir yaitu proses kompresi dimana jika inverted list berupa CSP dikodekan menggunakan *Huffman* sebaliknya jika berupa daftar integer dikodekan dengan *Golomb*.

Hasil pengujian untuk rasio kompresi *inverted list* dari 1000 dokumen menunjukkan bahwa metode kombinasi *Golomb* dan *Huffman* berdasarkan CSP pada daftar *document Id* lebih baik 20.77% dari *Gamma*, 3.96% dari *Golomb*, dan 17.96% dari kombinasi *Gamma* dan *Huffman* sedangkan pada *term frequencies* lebih baik 21.21% dari *Gamma*, 10.88% dari *Golomb*, dan 10.49% dari kombinasi *Gamma* dan *Huffman*.

Kata Kunci : *Inverted List, Contigous Sequential Pattern, UpDown Tree*.

**INVERTED LIST COMPRESSION USING GOLOMB AND HUFFMAN COMBINATION BASED ON CONTIGUOUS SEQUENTIAL PATTERNS (CSP)**


## Abstract

Index is a list of words with the position of book pages. In it's implementation, the index stored in the disk, but increasing number of documents will produce it to be large of disk space. Therefore, efficiency of the space required for the index.

Inverted list compression system includes several steps. The first step is preproccessing which extracts the document collection into a list of terms using case folding, stop word, and stemming. The second step is indexing which produce inverted lists. The third step is the detection of CSP on inverted list using apriori and UpDown Tree. The final step is compression process, if inverted list is a sequence of CSP would be encoded using Huffman but if it is integer using Golomb.

The test results for compression ratio of inverted list of 1000 documents showed that the combination Golomb and Huffman based on CSP at list of document Ids better 20.77% of the Gamma, 3.96% of Golomb, and 17.96% of the combination Gamma and Huffman while term frequencies better 21.21% of the Gamma, 10.88% of Golomb, and 10.49% of the combination Gamma and Huffman.



Keywords: Inverted List, Contiguous Sequential Pattern, UpDown Tree.

# KATA PENGANTAR

Puji syukur Penulis panjatkan kepada Allah SWT atas segala limpahan rahmat dan hidayah-Nya, sehingga Penulis dapat menyelesaikan skripsi ini sebagai salah satu syarat untuk memperoleh gelar Sarjana Komputer dalam bidang Ilmu Komputer.

Pada penyusunan skripsi ini, Penulis ingin mengucapkan terima kasih kepada:

1. Dian Eka Ratnawati, SSi., M.Kom selaku pembimbing utama atas segala arahan serta bimbingannya dalam penyusunan skripsi ini.
2. Yusi Tyroni Mursityo, S.Kom, MS selaku pembimbing pendamping atas segala arahan serta bimbingannya dalam penyusunan skripsi ini.
3. Drs. Marji, MT sebagai Ketua Program Studi Ilmu Komputer, Jurusan Matematika, Fakultas Mipa, Universitas Brawijaya.
4. Dr. Abdul Rouf A., MSc selaku Ketua Jurusan Matematika, FMIPA Universitas Brawijaya.
5. Segenap bapak dan ibu dosen yang telah mendidik dan mengajarkan ilmunya kepada Penulis selama menempuh pendidikan di Program Studi Ilmu Komputer Jurusan Matematika FMIPA Universitas Brawijaya.
6. Segenap staf dan karyawan di Jurusan Matematika FMIPA Universitas Brawijaya yang telah banyak membantu Penulis dalam usaha memperlancar urusan administrasi untuk pelaksanaan penyusunan skrispsi ini.
7. Kedua orang tua dan saudara-saudari Penulis, atas segala dukungannya baik berupa materi, motivasi, arahan dan doa restunya kepada Penulis.
8. Rekan-rekan di Program Studi Ilmu Komputer FMIPA Universitas Brawijaya yang telah banyak memberikan inspirasi, motivasi, dukungan, dan bantuannya kepada Penulis demi kelancaran pelaksanaan penyusunan skripsi ini khususnya kepada angkatan 2005.
9. Semua pihak yang telah membantu dalam penyusunan skripsi ini yang tidak dapat Penulis sebutkan satu per satu.

Penulis sadari bahwa masih banyak kekurangan dalam penyusunan skripsi ini. Oleh karena itu, segala kritik dan saran yang bersifat membangun sangat Penulis harapkan dari berbagai pihak guna peningkatan kualitas penelitian serupa di masa mendatang.

Malang, 28 Mei 2012

Penulis

# DAFTAR ISI

Halaman

**BAB III METODOLOGI DAN PERANCANGAN**

**BAB IV IMPLEMENTASI DAN PEMBAHASAN**

# DAFTAR GAMBAR

Halaman

# DAFTAR TABEL

Halaman

## DAFTAR *SOURCE CODE*

Halaman


UNIVERSITAS BRAWIJAYA

## DAFTAR LAMPIRAN

Halaman


UNIVERSITAS BRAWIJAYA

# BAB I
# PENDAHULUAN

## 1.1 Latar Belakang

*Index* merupakan daftar kata yang disertai posisi halaman pada sebuah buku. Secara umum *index* terletak pada halaman terakhir dari sebuah buku. *Index* difungsikan untuk mempermudah pencarian informasi sesuai kebutuhan pembaca, tanpa mengharuskan pembaca tersebut menelusuri buku secara keseluruhan. Konsep *index* inilah yang dipakai oleh *information retrieval* untuk mengefisiensi *query* pencarian pada koleksi dokumen digital. Secara implementasi, *index* pada koleksi dokumen disimpan dalam *disk*, tetapi dengan semakin bertambahnya jumlah dokumen akan mengakibatkan ruang penyimpanan *index* dalam *disk* menjadi besar. Oleh karena itu, diperlukan efisiensi ruang penyimpanan untuk *index*.

*Inverted file index* merupakan teknik *indexing* yang sangat popular saat ini, selain memiliki ukuran *index* yang relatif kecil juga menghasilkan efisiensi tinggi untuk *keyword* berdasarkan *query* (Chen dan Cook, 2007). *Inverted index* dibagi menjadi dua bagian yaitu *lexicon* dan *inverted list*. *Lexicon* merupakan sebuah *dictionary* dari *term* yang digunakan untuk pencarian, sedangkan *inverted list* merupakan sebuah daftar *stream integer* yang terdiri dari sebuah daftar *document identifiers* (*document Id*) yang memuat *term* tertentu, sebuah daftar *term frequencies* (frekuensi *term*) dalam setiap dokumen, dan sebuah daftar *term positions* (posisi *term*) dalam setiap dokumen (Delbru, dkk., 2010).

Tujuan utama kompresi *inverted index*, terutama kompresi pada *inverted list* adalah mengkodekan daftar *integer* (berukuran 32 bit atau 4 *byte*) menggunakan bit yang lebih kecil untuk menghemat ruang penyimpanannya (Delbru, dkk., 2010).

Beberapa metode yang sering digunakan untuk pengkodean *inverted list* antara lain *Gamma*, *Golomb*, dan sebagainya, Metode-metode tersebut hanya mengkodekan daftar *integer*-nya, padahal dalam *inverted list* banyak ditemukan *sequential pattern* (SP) dari daftar *integer*-nya. Oleh karena itu, dapat dilakukan perbaikan kompresi menggunakan *sequential pattern* (Chen dan Cook, 2007)

*Sequential pattern mining* adalah suatu cara menemukan *sub-sequence* yang sering muncul sebagai pola di dalam *database*.

*Sequential pattern* merupakan permasalahan *data mining* yang sangat penting dalam berbagai aspek seperti analisis dari perilaku belanja dari konsumen, *web access pattern*, eksperimen ilmiah, penanganan berbagai penyakit, bencana alam, *sequence* DNA, dan sebagainya (Pei., dkk, 2001). *Contiguous sequential pattern* (CSP) merupakan bentuk variasi dari *sequential pattern* (SP) dimana kemunculan item-item dalam sebuah *sequence* yang memuat sebuah pola, harus berdasarkan urutan dalam pola tersebut (Chen dan Cook, 2007).

Penelitian untuk kompresi *inverted list* menggunakan CSP sebelumnya pernah dilakukan oleh Chen dan Cook (2007) yaitu dengan menggunakan kombinasi metode *Gamma* dan *Huffman*. Pada penelitian Chen dan Cook ini dihasilkan bahwa kompresi *inverted list* dengan pendekatan CSP menggunakan kombinasi *Gamma* dan *Huffman* secara efektif memperbaiki rasio kompresi dibandingkan pengkodean dengan menggunakan metode *Gamma* tunggal.

Sementara itu, hasil penelitian William dan Zobel (1999) yang mengompresi data numerik dari *database Australian Research Council* menggunakan metode *Gamma* dan *Golomb* disimpulkan bahwa *Golomb* menghasilkan dua keunggulan dibandingkan dengan *Gamma* yaitu ukuran rasio kompresi yang lebih kecil dan proses dekompresi lebih cepat.

Berdasarkan latar belakang yang telah dipaparkan di atas, penulis melakukan penelitian dengan mengganti metode pengkodean pada penelitian Chen dan Cook (2007) yaitu dengan mengganti metode *Gamma* menjadi *Golomb*. Kemudian dilakukan pengujian untuk perbandingan metode kombinasi *Golomb* dan *Huffman* dengan metode kombinasi *Gamma* dan *Huffman* berdasarkan pendeteksian CSP baik segi rasio kompresi *inverted list* maupun waktu dekompresinya.

## 1.2 Rumusan Masalah

Berdasarkan latar belakang yang telah dijelaskan maka rumusan masalahnya adalah :

1. Bagaimana cara mengimplementasikan kompresi *inverted list* menggunakan kombinasi *Golomb* dan *Huffman* berdasarkan pendeteksian *contiguous sequential pattern* (CSP)?
2. Berapakah rasio kompresi dan kecepatan waktu dekompresi dari kompresi *inverted list* menggunakan kombinasi *Golomb* dan *Huffman* dibandingkan dengan kombinasi *Gamma* dan *Huffman*

berdasarkan pendeteksian CSP serta dibandingkan juga dengan metode kompresi menggunakan metode *Gamma* maupun *Golomb*?

3. Berapakah rasio kompresi dan waktu dekompresi dari metode kombinasi *Golomb* dan *Huffman* berdasarkan CSP dengan proses *filtering* seperti *case folding*, *stop word*, dan *stemming* dibandingkan dengan kombinasi *Golomb* dan *Huffman* tanpa proses *filtering*?

## 1.3 Batasan Masalah

Batasan masalah pada penelitian ini adalah :

1. Koleksi dokumen untuk penelitian ini menggunakan teks artikel berita yang berekstensi *file* *.txt yang bersumber dari *http://www.kompas.com* yang diambil mulai tanggal 1 Maret 2012 sampai dengan 31 Maret 2012.
2. Dokumen yang digunakan dalam bahasa Indonesia.
3. Teknik *preprocessing* yang digunakan untuk menghasilkan daftar *term* adalah *case folding*, *stop word*, dan *stemming*.
4. *Stemming* menggunakan Algoritma Nazief dan Adriani.
5. Tidak melakukan evaluasi atau pengujian *query* terhadap hasil kompresi *inverted list*. Evaluasi hanya menitikberatkan pada rasio kompresi dan kecepatan dekompresi dari *inverted list*.
6. CSP set dalam *inverted list* dikodekan dengan menggunakan pengkodean *Huffman*.

## 1.4 Tujuan

Tujuan yang ingin dicapai dalam penelitian adalah :

1. Pembuatan sistem kompresi *inverted list* menggunakan kombinasi *Golomb* dan *Huffman* berdasarkan pendeteksian CSP.
2. Pengujian rasio kompresi dan kecepatan dekompresi dari kompresi *inverted list* menggunakan kombinasi *Golomb* dan *Huffman* dibandingkan dengan kombinasi *Gamma* dan *Huffman* berdasarkan pendeteksian CSP serta menguji kompresi *inverted list* tanpa pendeteksian CSP dengan menggunakan metode tunggal *Gamma* ataupun *Golomb*.

## 1.5 Manfaat

Manfaat yang diharapkan dari skripsi ini adalah membantu menyelesaikan permasalahan rendahnya rasio kompresi maupun kecepatan dekompresi pada *inverted list*.

## 1.6 Sistematika Penulisan

Sistematika penulisan skripsi ini adalah sebagai berikut :

**BAB I : PENDAHULUAN**

Bab ini membahas mengenai latar belakang, rumusan masalah, batasan masalah, tujuan, manfaat dan sistematika penulisan skripsi.

**BAB II : TINJAUAN PUSTAKA**

Bab ini menjelaskan uraian dasar teori terkait yang menjadi dasar rujukan pada penelitian ini.

**BAB III : METODOLOGI DAN PERANCANGAN**

Bab ini berisi penjelasan metode yang digunakan dalam penelitian dan bagaimana kerangka dasar solusi yang diusulkan.

**BAB IV : IMPLEMENTASI DAN PEMBAHASAN**

Bab ini berisi uraian mengenai implementasi dari rancangan solusi yang diajukan dan pembahasan mengenai hasil pengujian.

**BAB V : PENUTUP**

Bab ini berisi kesimpulan dari penelitian yang telah dilakukan, dan saran untuk pengembangan penelitian selanjutnya.

# BAB II
# TINJAUAN PUSTAKA

## 2.1 *Information Retrieval*

Menurut Manning dkk (2009), *information retrieval* (IR) adalah menemukan material dari sesuatu yang bersifat tidak struktur untuk memenuhi kebutuhan informasi dari koleksi yang besar (umumnya disimpan dalam komputer).

### 2.1.1 Tokenisasi

Tokenisasi adalah proses pemisahan *sequence* karakter dari unit dokumen menjadi potongan kata, disebut juga dengan *token*. Tokenisasi juga berfungsi menghilangkan karakter yang khusus, seperti pungtuasi (Manning, dkk., 2009).

### 2.1.2 *Stop Word*

Sering sekali kata-kata umum yang muncul dengan frekuensi besar dalam dokumen hanya sedikit membantu pemilihan dokumen yang sesuai kebutuhan *user*, sehingga kata-kata umum tersebut perlu dihilangkan dari daftar *term* secara keseluruhan. Kata-kata ini disebut *stop words*. Langkah-langkah untuk menentukan sebuah *stop list* adalah *term* diurutkan sesuai frekuensi dokumen dan diambil sebagian frekuensi *term* yang terbesar. Akan tetapi lebih sering digunakan *hand-filtered* (secara manual) untuk konten semantik berhubungan dengan domain dari dokumen yang di-*index*, sebagai sebuah *stop list*, anggota yang ada kemudian dihilangkan selama proses *indexing* (Manning, dkk., 2009).

### 2.1.3 *Case Folding*

*Case folding* merupakan bentuk kapitalisasi. Langkah dari *case folding* adalah mengubah seluruh huruf menjadi huruf kecil atau *lower case* (Manning, dkk., 2009).

### 2.1.4 *Stemming*

*Stemming* merupakan suatu proses yang terdapat dalam sistem IR yang mentransformasi kata-kata yang terdapat dalam suatu dokumen

ke kata-kata dasarnya (*root word*) dengan menggunakan aturan-aturan tertentu. Sebagai contoh, kata bersama, kebersamaan, menyamai, akan di-*stem* ke kata dasarnya yaitu "sama". Proses *stemming* pada teks berbahasa Indonesia berbeda dengan *stemming* pada teks berbahasa Inggris. Pada teks berbahasa Inggris, proses yang diperlukan hanya proses menghilangkan sufiks, sedangkan pada teks berbahasa Indonesia, selain sufiks, prefiks, dan konfiks juga dihilangkan (Agusta, 2009).

#### 2.1.4.1 Algoritma Nazief dan Adriani

Menurut Asian dkk (2005), langkah-langkah *stemming* pada algoritma Nazief dan Adriani adalah:

1. Kata yang akan di-*stem* dicari di dalam kamus. Jika kata ditemukan, maka diasumsikan kata tersebut merupakan kata dasar (*root word*) dan algoritma berhenti.
2. *Inflection suffixes* ("-lah", "-kah", "-ku", "-mu", atau "-nya") dihilangkan. Jika ada sufiks (akhiran) yang dihilangkan dan sufiks merupakan partikel ("-lah" atau "-kah") maka langkah diulang lagi untuk menghilangkan *inflectional possessive pronoun suffixes* ("-ku", "-mu", atau "-nya").
3. *Derivation suffix* ("-i" atau "-an") dihilangkan. Jika sufiks ini berhasil dihilangkan lanjutkan ke langkah 4. Jika pada langkah 4 mengalami kegagalan, maka lakukan langkah berikut ini:
    a. Jika "-an" yang dihilangkan, dan karakter terakhirnya adalah "-k", maka "-k" juga dihilangkan dan lanjutkan proses ke langkah 4. Jika sebaliknya maka ke langkah 3b.
    b. Sufiks yang dihilangkan ("-i", "-an", atau "-kan") dikembalikan ke kata yang di-*stem* dan lanjutkan proses ke langkah 4.
4. *Derivation prefix* dihilangkan. Ada beberapa tahapan penghapusan prefiksnya:
    a. Jika sufiks pada langkah 3 dihilangkan, kemudian periksa tabel 2.1 kombinasi prefiks dan sufiks yang tidak diijinkan. Jika ditemukan pada tabel kombinasi, maka ulangi algoritmanya.
    b. Jika prefiks yang sedang diproses sama dengan prefiks sebelumnya, maka ulangi algoritmanya.

c. Jika tiga prefiks sebelumnya telah dihilangkan atau dihapus, maka ulangi algoritmanya.
d. Jenis prefiks ditentukan dengan aturan sebagai berikut:
    i. Jika prefiks dari kata adalah “di-”, “ke-“, atau “se-“, maka jenis prefiksnya secara berturut-turut “di-“, “ke-“, atau “se-“.
    ii. Jika prefiks adalah “te-“, “be-“, “me-“, atau “pe-”, maka dibutuhkan proses tambahan untuk mengekstrak set karakter untuk menentukan tipe prefiksnya. Sebagai contoh aturan untuk prefiks “te-“ ditunjukkan oleh tabel 2.2.
    iii. Jika dua karakter pertama tidak sama dengan “di-“, “ke-“, “se-“, “te-“, “be-”, “me-“, atau “pe-“, maka ulangi algoritmanya.
e. Jika jenis prefiks tidak ada, maka ulangi algoritmanya. Sebaliknya jika prefiks ditemukan pada tabel 2.3 maka prefiks tersebut dihapus.
f. Jika kata dasar masih belum ditemukan di kamus, maka lakukan langkah 4 secara rekursif untuk menghilangkan prefiks. Sebaliknya jika kata dasar ditemukan di kamus, maka ulangi algoritmanya.
g. Dilakukan *recoding*. Langkah ini tergantung jenis prefiksnya. Jika prefiks yang dihasilkan tidak sesuai dengan *rule*, maka prefiksnya ditambahkan ke kata yang di-*stem* dan dilakukan pengecekan di dalam kamus.

5. Seluruh langkah telah dilakukan, tetapi kata dasar belum juga ditemukan, maka algoritma mengembalikan kata asli sebelum di-*stem*.

Kombinasi prefiks dan sufiks yang tidak diperbolehkan ditunjukkan oleh tabel 2.1. Pengecualian hanya pada kata dasar “tahu” diijinkan dengan prefiks “ke-“ dan sufiks “-i”, misalnya kata “ketahui”.

**Tabel 2.1.** Kombinasi Awalan dan Akhiran yang Tidak Diijinkan.

| Awalan | Akhiran yang tidak dijinkan |
|---|---|
| be- | -i |
| di- | -an |
| ke- | -i, -kan |
| me- | -an |
| se- | -i, -kan |

Aturan untuk menentukan tipe awalan dari “te-” ditunjukkan oleh tabel 2.2, jika kata yang diawali “te-“ tidak sesuai dengan aturan, maka prefiksnya “tidak ada” sebagai hasil pengembalian dari algoritma. Aturan yang digunakan untuk awalan “be-“, “me-“, dan “pe-“ hampir sama dengan “te-“.

**Tabel 2.2.** Cara Menentukan Tipe Awalan untuk Kata yang Diawali dengan “te-”.

| Karakter-karakter setelah awalan | | | | Tipe Awalan |
|---|---|---|---|---|
| Set 1 | Set 2 | Set 3 | Set 4 | |
| “-r-“ | “-r-“ | - | - | tidak ada |
| “-r-“ | Vokal | - | - | ter-luluh |
| “-r-“ | bukan (vokal atau “-r-“) | “-er-“ | vokal | ter- |
| “-r-“ | bukan (vokal atau “-r-“) | “-er-“ | bukan vokal | ter- |
| “-r-“ | bukan (vokal atau “-r-“) | bukan “-er-“ | - | ter- |
| bukan (vokal atau “-r-“) | “-er-“ | vokal | - | tidak ada |
| bukan (vokal atau “-r-“) | “-er-“ | bukan vokal | - | te- |

**Tabel 2.3.** Jenis Awalan Berdasarkan Tipe Awalannya.

| Jenis Awalan | Awalan yang harus dihapus |
|---|---|
| di- | di- |
| ke- | ke- |
| se- | se- |
| te- | te- |
| ter- | ter- |
| ter-luluh | ter- |

Pada tabel 2.2 dan 2.3. Setelah penghapusan "ter-", awalan "r-" ditambahkan ke kata. Jika kata baru tidak terdapat pada kamus maka langkah 4 diulangi. Kemudian dilakukan pengecekan dalam kamus, jika tidak ditemukan juga dalam kamus maka hapus "r-" dan "ter-" dikembalikan, jenis prefiksnya diatur menjadi "tidak ada", dan ulangi algoritmanya.

## 2.2 *Inverted Index*

*Indexing* adalah sebuah mekanisme untuk menemukan sebuah *term* yang diberikan dalam sebuah dokumen. Dalam aplikasi-aplikasi yang melibatkan dokumen, struktur data yang sangat sesuai adalah *inverted index* (Witten, dkk., 1999).

Menurut Delbru dkk (2010), *inverted index* terdiri dari dua bagian:

1. *Lexicon*, berupa sebuah *dictionary* dari *term*.
2. *Inverted list*, merupakan *stream* dari *integer* yang terdiri dari sebuah daftar *document identifiers* (*document Id*) yang memuat *term* tertentu, sebuah daftar *term frequencies* (frekuensi *term*) dalam setiap dokumen, dan sebuah daftar *term positions* (posisi *term*) dalam setiap dokumen.

Diketahui sejumlah dokumen $D = \{d_1,d_2,\ldots\}$, setiap dokumen mempunyai *unique id* (dinotasikan $id_j$) dan set *term* $T = \{t_1,t_2,\ldots\}$. Struktur dari *inverted list* adalah $<id_j, f_{ij}, [\ o_1, o_2, \ldots, o_{[fij]}\ ]>$, dimana $id_j$ adalah identitas dari dokumen (*document Id*) yang memuat *term* $t_i$, $f_{ij}$ adalah frekuensi dari *term* $t_i$ dalam $d_j$, dan $o_k$ adalah posisi dari *term* $t_i$ dalam $d_j$ (Cho, 2008).

### 2.2.1 Kompresi *Inverted List*

Menurut Witten dkk (1999), Ukuran dari sebuah *inverted file* dapat dikurangi dengan cara mengompresinya. Setiap *inverted list* disimpan sebagai *sequence* dari *integer* secara *ascending*. Secara umum, *inverted list* untuk sebuah *term t* adalah $<f_t; d_1,d_2,.. ,d_{ft}>$, dimana $f_t$ adalah jumlah frekuensi kemunculan *t* dan $d_k < d_{k+1}$. Dikarenakan *inverted list* dari *document Id* dalam urutan *ascending*, dan setiap urutan posisi *document Id* merupakan *sequential* dari posisi *document Id* yang pertama sehingga *inverted list* dapat disimpan sebagai posisi pertama dari *document Id* yang diikuti dengan daftar *d-gap* dari *document Id* berikutnya, dimana *d-gap* adalah selisih antara $d_{k+1} - d_k$.

Sebagai contoh, sebuah *term* muncul pada delapan dokumen, masing-masing *Id* dari dokumen adalah 3,5,20,21,23,76,77,78. *Inverted file* dari *term* tersebut dapat dituliskan menjadi sebuah daftar <8; 3,5,20,21,23,76,77,78>. Daftar untuk *term* tersebut dapat disimpan menggunakan *d-gap* menjadi <8; 3,2,15,1,2,53,1,1> (Witten, dkk., 1999).

#### 2.2.1.1 *Unary Code*

*Unary Code* dari *integer n* positif didefinisikan sebagai *n*-1 angka satu diikuti oleh sebuah angka nol tunggal atau alternatif lainnya, didefinisikan sebagai *n*-1 angka nol diikuti oleh sebuah angka satu tunggal. Panjang dari *unary code* untuk *integer n* yaitu menjadi *n* bit (Salomon, 2007). Sebagai contoh *unary* ditunjukkan oleh tabel 2.4.

**Tabel 2.4.** Beberapa Contoh *Unary Code*.

| *n* | *Code* | *Alternative Code* |
|---|---|---|
| 1 | 0 | 1 |
| 2 | 10 | 01 |
| 3 | 110 | 001 |
| 4 | 1110 | 0001 |
| 5 | 11110 | 00001 |

#### 2.2.1.2 *Gamma Code* (γ)

Menurut González (2008), representasi sebuah *integer x* untuk pengkodean *Gamma* dibagi menjadi dua bagian:

- $\lfloor \log_2 x \rfloor + 1$ dikodekan dalam bentuk *unary*.
- $x - 2^{\lfloor \log_2 x \rfloor}$ dikodekan dalam bentuk *binary* menggunakan $\lfloor \log_2 x \rfloor$ bit.

Langkah-langkah untuk mendekodekan *Gamma* menurut Cho (2008), antara lain:

- Membaca dan memeriksa jumlah angka nol dari *bit stream* sampai ditemukan angka satu yang pertama. Jumlah angka nol disimbolkan sebagai *K*. Dianggap bahwa angka satu yang pertama kali muncul sebagai permulaan digit dari *integer*.
- Sisa *K* bit dari *integer* dibaca sebagai representasi nilai *x*.

Menurut González (2008), perkiraan panjang bit dari *Gamma code*, dapat dihitung dengan:

$$L_x(\gamma) = 1 + 2\lfloor \log_2 x \rfloor \quad \textbf{(2.1)}$$

#### 2.2.1.3 *Golomb Code*

Menurut Salomon (2007), *Golomb code* untuk *integer n* positif bergantung pada pemilihan dari sebuah parameter *m*. Langkah pertama dalam perhitungan *Golomb code* dari *integer n* positif adalah menghitung tiga kuantitas *q* (*quotient*), *r* (*remainder*), dan *c*:

$$q = \left\lfloor \frac{n}{m} \right\rfloor \quad \textbf{(2.2)}$$

$$r = n - q.m \quad \textbf{(2.3)}$$

$$c = \lceil \log_2 m \rceil \quad \textbf{(2.4)}$$

Menurut Cho (2008) representasi *Golomb code* dapat dibentuk menjadi:

- Pertama, sebuah representasi *unary* dari $q + 1$, dimana $q$ dihitung dengan persamaan 2.2.

– Kedua, representasi *binary* dari *r* dikodekan secara khusus. Representasi dari sebuah *remainder r* membutuhkan $\lfloor \log_2 m \rfloor$ atau $\lceil \log_2 m \rceil$ bit. Untuk efisiensi ruang penyimpanan, pengkodean beberapa *remainder* pertama menggunakan $\lfloor \log_2 m \rfloor$ bit dan sisanya menggunakan $\lceil \log_2 m \rceil$ bit. Pengkodean *remainder* pertama yang disimbolkan sebagai *d*, menggunakan *c* bit, dan *d* dihitung dengan:

$$d = 2^c - m \qquad \textbf{(2.5)}$$

Jika *remainder r* lebih kecil dari *d* atau *r* < *d*, maka sisa *remainder* dikodekan dalam representasi *binary* dengan jumlah bit sebanyak $\lfloor \log_2 m \rfloor$ dan sebaliknya jika *remainder r* lebih besar sama dengan *d* atau r ≥ *d*, maka dikodekan dalam representasi *binary* dengan jumlah bit sebanyak $\lceil \log_2 m \rceil$.

Sebagai contoh penentuan *remainder* dari *r* misalkan *m* = 3 menghasilkan *c* = 2 dan tiga remaindernya 0,1, dan 2. Nilai *d* dihitung $2^2$-3=1, yang berarti bahwa 1 *remainder* pertama dikodekan dengan *c*-1 bit dengan pengkodean biner yaitu "0" dan dua *remainder* lainnya dikodekan dalam *c* bit yang dikodekan dengan $11_2$ yaitu "10" dan "11" sedangkan parameter *m* = 5 menghasilkan *c* = 3 dan mempunyai 5 *remainder* dari 0 sampai 4. Nilai *d* dihitung $2^3$-5=3, yang berarti bahwa 3 *remainder* pertama dikodekan dalam biner dengan *c*-1 bit dan 2 *remainder* lainnya dikodekan menggunakan akhiran $111_2$ dengan panjang *c* bit. Oleh karena itu kode biner dari *remainder* dengan *m*=5 adalah "00", "01", "10", "110", dan "111".

Aturan yang digunakan untuk mengkodekan angka-angka *c* bit adalah dengan akhiran "1" sebanyak *c*. Pertama-tama ambil nilai terbesar dari angka *c*-1 bit, dinotasikan dalam *b*, kemudian dibentuk biner dari *b*+1 dengan panjang *c*-1 bit dan tambahkan sebuah karakter "0" pada bagian kanannya. Hasilnya adalah angka pertama dari *c* bit dan remainder lainnya dihasilkan dengan kenaikan +1 dari angka pertama *c*-bit dalam format biner. Seperti pada contoh sebelumnya, parameter *m* = 5 dengan *c* = 3 mempunyai *remainder* terakhir pada *c*-1 bit dalam biner adalah "10" atau 2 dalam desimal. Oleh karena itu *b* = 2+1 = 3, sehingga kode biner *remainder* berikutnya menjadi "11"+"0" = "110" sedangkan untuk kode biner

selanjutnya $b$+1 = 4 sehingga "111" proses ini berlanjut sampai jumlah *remainder*-nya sama dengan $m$-1.

Pemilihan parameter $m$ untuk pengkodean *Golomb* sangat berpengaruh pada efisiensi ruang penyimpanan (*space*) dan waktu kompresinya. Secara umum persamaan untuk parameter $m$ adalah:

$$m \approx 0.69 \, x \, mean(v) \quad \textbf{(2.6)}$$

Dimana 0.69 adalah nilai konstanta, *mean* adalah nilai rata-rata dari $v$ (daftar *integer*). Model untuk pemilihan parameter $m$ ini berdasarkan sebuah model *Bernoulli* (William dan Zobel, 1999).

Menurut Salomon (2007), *Golomb code* didekodekan dengan merekonstruksi $n$ menggunakan nilai $q$ dan $r$:

$$n = r + q.m \quad \textbf{(2.7)}$$

Sementara itu, untuk mendekodekan sebuah *bit stream* yang menggunakan *Golomb*, menurut Cho (2008) adalah sebagai berikut:

1. Kode *unary* dari *quotient* $q$ + 1 bit didekodekan.
2. Dihitung $c$ dan $d$ menggunakan persamaan 2.4 dan 2.5.
3. Dapatkan nilai bit selanjutnya sebanyak $c$ bit kemudian nilai konversi *integer*-nya ditentukan sebagai *remainder* $r$.
4. Jika $r \geq d$ maka lanjutkan pembacaan sebanyak 1 bit dan tambahkan bit tersebut pada bit terakhir dari $r$. kemudian hitung:

$$r = r - \mathrm{d} \quad \textbf{(2.8)}$$

5. Didapatkan hasil $n$ menggunakan persamaan 2.7.

Menurut González (2008), perkiraan panjang bit dari *Golomb code*, dapat dihitung dengan:

$$L_x\,(G) = 1 + \left\lfloor \frac{n}{m} \right\rfloor + \lfloor \log_2 m \rfloor \quad \textbf{(2.9)}$$

Penjelasan simbol matematika yang digunakan pada beberapa persamaan seperti persamaan 2.2 dan 2.4 ditunjukkan oleh tabel 2.5.

**Tabel 2.5.** Penjelasan Simbol Matematika.

| Simbol | Nama Simbol | Definisi |
|---|---|---|
| $\lfloor x \rfloor$ | *floor* dari *x* | *integer* dibulatkan ke $\leq x$. |
| $\lceil x \rceil$ | *ceil* dari *x* | *integer* dibulatkan ke $\geq x$. |

Pada tabel 2.6 menunjukkan pengkodean angka *integer* dari 1 sampai 7 dengan menggunakan *unary*, *Gamma*, dan *Golomb code*.

**Tabel 2.6.** Contoh Pengkodean Angka 1 sampai 7.

| *Desimal* | *Binary* | *Unary* | *Gamma* | *Golomb* (*m*=3) | *Golomb* (*m*=10) |
|---|---|---|---|---|---|
| 1 | 00000001 | 1 | 1 | 1 10 | 1 001 |
| 2 | 00000010 | 01 | 0 10 | 1 11 | 1 010 |
| 3 | 00000011 | 001 | 0 11 | 01 0 | 1 011 |
| 4 | 00000100 | 0001 | 00 100 | 01 10 | 1 100 |
| 5 | 00000101 | 00001 | 00 101 | 01 11 | 1 101 |
| 6 | 00000110 | 000001 | 00 110 | 001 0 | 1 1100 |
| 7 | 00000111 | 0000001 | 00 111 | 001 10 | 1 1101 |

### 2.2.2 *Compression Performance*

Menurut Salomon (2007), untuk mengukur kinerja kompresi (*compression performance*) digunakan beberapa metode pengukuran hasil kompresi, yaitu:

1. *Compression Ratio*

   Jika nilai rasio bernilai 0.6 yang berarti bahwa ukuran data hanya mengambil 60% dari ukuran *file* asli setelah kompresi. Jika nilai rasio lebih besar dari 1 maka akan menyembabkan ukuran *output stream* lebih besar dari *input stream* (*negative compression*). *Compression ratio* didefinisikan sebagai:

$$compression\ ratio = \frac{size\ of\ output\ stream}{size\ of\ input\ stream} \qquad \textbf{(2.10)}$$

2. Persentase *Compression Ratio*

   Jika nilai CP 60 berarti bahwa ukuran *output stream* hanya mengambil 40% dari ukuran file asli atau hasil kompresi menghemat

ruang penyimpanan sebesar 60%. Persentase dari *compression ratio* didefinisikan sebagai:

$$Persentase\ CP = 100 * (1 - compression\ ratio) \quad \textbf{(2.11)}$$

## 2.3 *Sequential Pattern Mining*

Permasalahan *sequential pattern mining* pertama kali diperkenalkan oleh Agrawal dan Srikant (1994): "Diberikan satu set *sequence*, dimana setiap *sequence* terdiri dari sebuah daftar elemen dan setiap elemen terdiri dari satu set item, dan diberikan sebuah batasan *minimum support* yang ditentukan oleh pengguna, *sequential pattern mining* adalah menemukan seluruh *sub-sequence* yang sering muncul, dengan kata lain *sub-sequence* dimana frekuensi kemunculannya tidak lebih kecil dari *minimum support"* (Pei, dkk., 2001).

Terdapat dua aturan yang digunakan untuk menemukan *contiguous sequential pattern* **(**CSP) yaitu setiap elemen dalam sebuah *sequence* terdiri dari satu item dan kemunculan item-item dalam sebuah *sequence* yang mengandung sebuah pola, harus berdasarkan urutan (Chen dan Cook, 2007).

Diketahui $I = \{i_1, i_2, \ldots, i_n\}$ merupakan satu set item. Sebuah *itemset* adalah *subset* dari *I*, dinotasikan sebagai $(x_1, x_2, \ldots, x_k)$, dimana $x_i \in I, i \in \{1, \ldots, k\}$. Sebuah *sequence s* adalah sebuah daftar dari *itemset*, dinotasikan sebagai $<s_1, s_2, \ldots, s_m>$, dimana $s_i \subseteq I, , i \in \{1, \ldots, m\}$. Jumlah objek dari *itemset* pada *s* disebut juga panjang dari *s*. Sebuah *sequence* $a = <a_1, a_2, \ldots, a_j>$ adalah sebuah *contiguous sub-sequence* dari *sequence* lainnya yaitu *sequence* $b = <b_1, b_2, \ldots, b_k>$, $k \geq j$, jika *integer i*, $1 \leq i \leq k\text{-}j+1$, sehingga $a_1 \subseteq b_i$, $a_2 \subseteq b_{i+1}, \ldots, a_j \subseteq b_{i+j-1}$. Dalam kasus ini *a* termuat di dalam *b*, dinotasikan sebagai $a \subseteq b$, dan *b* merupakan sebuah *contiguous super-sequence* dari *a*.

Sebuah *database sequence* adalah sebuah *set tuple* <*sid*, *s*>,dimana setiap *sid* adalah sebuah *sequence id* dan *s* adalah sebuah *sequence*. Sebuah *tuple* <*sid*, *s*> dikatakan termuat dalam *sequence* $\beta$ jika $\beta \subseteq$ s. *Support* dari sebuah *sequence* $\beta$ dalam *database D* didefinisikan sebagai $Sup_D(\beta) = |\{<sid, s> \mid (\beta \subseteq s) \wedge (<sid, s> \in D)\}|$ , dan *support* rata-rata dari $\beta$ adalah $Sup_D(\beta) / |D|$. Diberikan sebuah nilai positif *minSup* (*minimum support*) sebagai *threshold* dari

*support*, $\beta$ adalah *contiguous sequential pattern* (CSP) dalam *D* jika $Sup_D(\beta) \geq minSup$ (Chen dkk, 2009).

### 2.3.1 Algoritma *Apriori*

Langkah awal dari algoritma ini hanya menghitung jumlah kemunculan item untuk menentukan *large* 1-*itemset*. Kemudiah Langkah selanjutnya, terdiri dari dua fase. Fase pertama *large itemset* $L_{k-1}$ ditemukan di dalam iterasi ke- (*k*-1) yang digunakan untuk menghasilkan *candidate itemset* $C_k$, menggunakan fungsi *apriori-gen* yang dijelaskan pada subbab2.3.2. Dan fase keduanya, *dataset* diperiksa dan *support* dari kandidat dalam $C_k$ dihitung. $L_k$ merupakan set *large k-itemset* dengan *minimum support* sedangkan $C_k$ merupakan set *candidate k-itemset* yang berpotensial menjadi *large itemset* (Agrawal dan Srikant, 1994). Algoritma *Apriori* ditunjukkan oleh gambar 2.1.

```
1)  L_1 = {large 1-itemsets};
2)  for ( k = 2; L_{k-1} ≠ ∅; k++ ) do begin
3)     C_k = apriori-gen(L_{k-1}); // New candidates
4)     forall transactions t ∈ D do begin
5)        C_t = subset(C_k, t); // Candidates contained in t
6)        forall candidates c ∈ C_t do
7)           c.count++;
8)     end
9)     L_k = {c ∈ C_k | c.count ≥ minsup}
10) end
11) Answer = ⋃_k L_k;
```

**Gambar 2.1** Algoritma *Apriori*.

### 2.3.2 *Apriori Candidate Generation*

Fungsi *Apriori-Gen* mengambil argumen $L_{k-1}$, sebagai seluruh set *large (k-1)-itemset*. *Itemset* ini mengembalikan *superset* dari seluruh *large k-itemset*. Fungsi ini bekerja sebagai berikut. Pertama, menggunakan *join step* yaitu menggabungkan $L_{k-1}$ dengan $L_{k-1}$. Algoritma *join step* ditunjukkan oleh gambar 2.2.

```
insert into C_k
select p.itemset_1, p.itemset_2, ..., p.itemset_{k-1}, q.itemset_{k-1}
from L_{k-1} p, L_{k-1} q
where p.itemset_1 = q.itemset_1, ..., p.itemset_{k-2} = q.itemset_{k-2};
```

**Gambar 2.2** Proses *Join Step* Pada Fungsi *Apriori-Gen*.

Kedua, menggunakan *prune step* yaitu dengan menghapus seluruh *itemset* $c \in C_k$ sedemikian sehingga beberapa ($k$-1)-*subset* dari $c$ tidak terdapat pada $L_{k-1}$. Algoritma *prune step* ditunjukkan oleh gambar 2.3 (Agrawal dan Srikant, 1995).

```
forall sequences c ∈ C_k do
  forall (k − 1)-subsequences s of c do
    if (s ∉ L_{k−1}) then
      delete c from C_k;
```

**Gambar 2.3** Proses *Prune Step* Pada Fungsi *Apriori-Gen*.

### 2.3.3 CSP *Mining* Menggunakan *UpDown Tree*

#### 2.3.3.1 Algoritma *UpDown Tree*

*UpDown Tree* merupakan struktur data untuk menemukan CSP *mining*. Sebuah *UpDown Tree* mengkombinasikan *suffix tree* dan *prefix tree* untuk mengefisiensi penyimpanan seluruh *sequence* yang memuat sebuah item. Untuk *sequence S* yang memuat item *k*. Jika *k* terdapat pada posisi item ke-*m* dari *sequence S*, maka Didefinisikan sufiks dan prefiks dari *k* dalam *S* sebagai *sub-sequence S* dengan aturan:

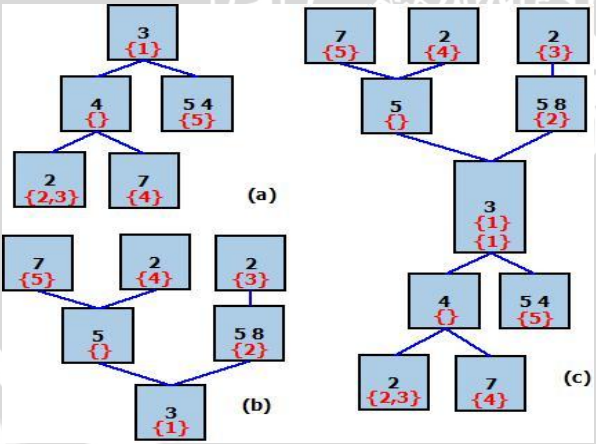
- Pembentukan sufiksnya dimulai dari posisi item *k* pada item ke-*m* sampai item terakhir dari *sequence S*.
- Pembentukan prefiksnya dimulai dari item pertama hingga ke posisi item *k* pada item ke-*m* dari *sequence S*.

Digunakan *Down Tree* untuk merepresentasi seluruh sufiks dari *k*, dan *Up Tree* untuk merepresentasikan seluruh prefix dari *k*. *UpDown Tree* merupakan kombinasi *root nodes* dari *Up* dan *Down Tree* (Chen dan Cook, 2007). Contoh pembentukan *UpDown Tree* ditunjukkan oleh tabel 2.7 dan gambar 2.4.

**Tabel 2.7.** Contoh *Sequence* yang Memuat Item 3.

| *Seq Id* | *Sequence* | *Full Prefix of 3* | *Full Suffix of 3* |
|---|---|---|---|
| 1 | 3 | 3 | 3 |
| 2 | 58342 | 583 | 342 |
| 3 | 258342 | 2583 | 342 |
| 4 | 25347 | 253 | 347 |
| 5 | 75354 | 753 | 354 |

Table 2.7 menunjukkan sebuah *dataset*. Untuk mendeksi CSP yang memuat sebuah item, misalkan item 3, diperlukan pengecekan seluruh *sequence* yang mengandung item 3. *Sequence* dapat dibagi menjadi dua *sub-sequence* sebelum atau sesudah item 3 (prefiks lengkap atau sufiks lengkap dari 3). Sufiks lengkap dari 3 secara efisien disimpan menggunakan *suffix trie*. Dapat dikatakan *trie* ini sebagai *Down Tree* karena akar (*root*) *node*-nya adalah level pertama. Untuk setiap sufiks lengkap dari 3 ditambahkan *sequence Id*-nya ke set *Id* dari *node* dalam *Down Trie* terhubung sampai item terakhir dari sufiks lengkapnya. Demikian juga, representasi seluruh prefiks lengkap dari 3 dengan sebuah *Up Trie* (*Up Tree*). *Root node* dari *Up Tree* berada pada level terbawah. Karena *up* dan *Down Tree* memakai *root* yang sama, keduanya diintegrasi ke dalam sebuah *UpDown Tree* (Chen dan Cook, 2007).




**Gambar 2.4** Contoh *UpDown Tree*.

Gambar 2.4 menunjukkan pembentukkan *UpDown Tree* dari tabel 2.7 dimana (a) adalah sebuah *Down Tree*, (b) adalah *Up Tree*, dan (c) adalah *UpDown Tree* yang merupakan gabungan dari (a) dan (b).

### 2.3.3.2 Permasalahan Partisi

Diketahui $\{x_1, x_2, \ldots, x_t\}$ merupakan *sequence set* dari 1-CSP atau *frequent itemset* dalam *database D*, $x_1 < x_2 < \ldots < x_t$, berdasarkan *Apriori rule* (subbab 2.3.1). Seluruh set CSP dalam *D* dapat dibagi atau dipisah menjadi *subset t*. *subset* ke-*i* ($1 \leq i \leq t$) adalah set CSP yang memuat $x_i$ dan *frequent itemset* lebih kecil dari $x_i$ (Chen dan Cook, 2007).

### 2.3.3.3 *Maximal Sub-Sequence Set*

Diberikan sebuah *sequence* $b = <b_1, b_2, \ldots, b_j>$, diketahui *k* adalah sebuah *frequent itemset* yang termuat dalam $b_i$, *frequent prefix* dan *suffix set* dari *k* dalam *b* adalah *contiguous sub-sequences* dari *b* yang diakhiri dan diawali pada posisi *k* dalam $b_i$, dimana seluruh *itemset*-nya dalam setiap *contiguous sub-sequences* yang sering muncul. *Maximal prefix* dan *suffix set* (MPS dan MSS) dari *k* adalah sebuah *subset* dari *frequent prefix* dan *suffix set* dari *k*, dimana *id* dari setiap *itemset* dalam setiap *maximal prefix* dan *suffix S* lebih kecil dari *id* dari *k*, dan bukan merupakan *sub-sequence* dari *S* dengan bagian yang sama (tidak ada bagian yang saling tumpang tindih) dalam *frequent prefix* dan *suffix set*. Untuk setiap kemunculan dari sebuah *itemset k*, terdapat sebuah MPS dan MSS. Menggabungkan setiap *sequence a* dalam MPS dengan setiap *b* dalam MSS (hilangkan *itemset* pertama dari *b* dimana sudah diwakili oleh *itemset* terakhir dari *a*), hasil *sequence set*-nya disebut *maximal sub-sequence set* (MSSS) dari *k* bersesuaian dengan setiap kemunculannya (Chen dkk , 2009).

Sebagai contoh diberikan sebuah *sequence* <(1) (5, 6) (3) (6) (7) (2) (4) (8) (1)>, jika 5, 6, 7, 3, 2, dan 4 adalah *frequent itemset*, kemudian *frequent prefix set*-nya dari 7 adalah{<(5) (3) (6) (7)>, <(6) (3) (6) (7)>, <(3) (6) (7)>, <(6) (7)>, <(7)>}, dan MPS-nya dari 7 adalah {<(5) (3) (6) (7)>, <(6) (3) (6) (7)> }. *Frequent suffix set*-nya dari 7 adalah {<(7) (2) (4)>, <(7) (2)>, <(7)>}, dan MSS-nya dari 7adalah <(7) (2) (4)>. MSSS-nya dari 7 adalah {<(5) (3) (6) (7) (2) (4)>, <(6) (3) (6) (7) (2) (4)>}.

### 2.3.3.4 Algoritma Untuk Mendeteksi CSP

Menurut Chen dkk (2009), algoritma untuk mendeteksi seluruh CSP adalah:

1. Algoritma untuk mendeteksi seluruh CSP dalam *subset i*.

*Input* : *UpDown Tree* dari $x_i$ dan *minSup* sebagai
*Output* : Seluruh CSP dalam subset *i*
Metode :

a. Membuat sebuah CSP set kosong dari $x_i$
b. Untuk setiap *node j* dalam *Up Tree* dari $x_i$ dalam *depth first order* (dimulai dari *leaf node*).
c. Tambahkan *sequence-id set*-nya dari *node j* ke *parent sequence-id set*-nya.
d. Membuat sebuah CSP *leaf set* kosong dan mengambil *node-node* terakhir dalam *Down Tree* dari seluruh *sequence* dalam *id set* dari *node j*. Untuk setiap *node* terakhir, letakkan *Id*-nya dari seluruh *sequence* yang berakhir pada *node* tersebut dalam *endIdSet*-nya.
e. *Enqueue* setiap *node* terakhir sebagai antrian berdasarkan urutan *descending* dari tinggi *tree*-nya dalam *Down Tree*.
f. *Dequeue* setiap *node*-nya dalam antrian, sampai antrian menjadi kosong. Untuk setiap *node k* yang di-*dequeue*, jika ukuran dari *endIdSet*-nya tidak lebih kecil dari *minSup*, tambahkan *node k* ke CSP *leaf set*, dan diberikan tanda (*mark*) seluruh *node* sepanjang bagian dari *root* ke *node k* sebagai yang diujikan. Sebaliknya gabungkan *endIdSet* ke *parent node*-nya dari *endIdSet* dan *enqueue parent node*-nya jika *parent*-nya tidak ada di dalam antrian dan diberikan tanda (*mark*) sebagai yang diujikan.
g. Untuk setiap *node q* dalam CSP *leaf set* , buatlah sebuah CSP dengan menggabungkan kunci *itemset* dalam bagian dari *j* sampai *q*, dan tambahkan CSP-nya ke CSP *set*.

2. Algoritma untuk mendeteksi seluruh CSP dalam sebuah *dataset*.

*Input* : Sebuah *sequence database D, minSup*
*Output* : Seluruh CSP dalam *database*
Metode :

a. Mendeteksi seluruh *frequent itemset* (FI).

b. Memberi tanda setiap FI dengan sebuah *unique id*. Untuk setiap FI dibuat sebuah set kemunculan yang menyimpan seluruh kemunculannya <*sequence-id, position*>.
c. Untuk setiap FI-nya dalam urutan *descending*.
d. Menentukan MSSS *collection* dari $x_i$ dan buatlah *UpDown Tree* berdasarkan definisi subbab 2.3.4.3.
e. Mendeteksi seluruh CSP dari $x_i$ menggunakan algoritma pada poin 1 dari subbab 2.3.4.4.

## 2.4 Penggunaan CSP untuk Kompresi *Inverted List*

Berdasarkan subbab 2.2 dan 2.3, pada *inverted list* yakni daftar *document identifiers* (*document Id*), *term frequencies* dan *term positions* kemungkinan terdapat beberapa pola (*pattern*) pada *sequence integer*-nya. Penggunaan *contiguous sequential pattern* (CSP) pada *inverted list* adalah untuk memperbaiki rasio kompresi *inverted list* (Chen dan Cook, 2007).

Setiap *inverted list* merupakan sebuah *sequence* dan nilai *integer* pada *list* merupakan sebuah item, deteksi *sequential pattern* (SP) menurut Chen dan Cook (2007) ditentukan dengan empat aturan:

1. Setiap elemen dalam *sequence* hanya terdiri dari satu item.
2. Setiap kemunculan item dalam *sequence* yang memuat sebuah pola, harus berdasarkan urutan yang terdefinisi dalam pola tersebut.
3. Jika sebuah pola muncul *k* kali dalam sebuah *sequence*, *support*-nya ditambahkan *k* bukannya 1. Alasannya adalah bahwa jika sebuah pola muncul berulang kali dalam *inverted list* yang sama, setiap kemunculan dapat dapat dikompresi dengan menggunakan sebuah *pattern Id* secara terpisah.
4. Jika dua pola saling tumpang tindih dalam *sequence* yang sama, hanya salah satu pola yang digunakan.

Untuk sebuah *inverted list* $<d_1,d_2,d_3,\ldots,d_{ft}>$, berdasarkan pendeteksian CSP pada subbab2.4, dapat dituliskan ulang sebagai $<g_1,g_2,\ldots,g_k>$, dimana $g_i$, merupakan sebuah item atau *pattern Id* dari sebuah CSP, dan *k* adalah jumlah total dari seluruh item dan CSP. Dapat dikatakan bahwa *inverted list* dengan format baru ini, adalah representasi *cluster* dari *inverted list* yang asli (Chen dan Cook, 2007). Untuk mengkodekan sebuah *inverted list* dalam representasi *cluster*, diperlukan pengecekan secara jelas apakah $g_i$ merupakan

sebuah item atau sebuah CSP. Algoritma di bawah ini menjelaskan bahwa informasi CSP (jumlah total, posisi) sebagai *header* dari *inverted list* yang terkompresi, dikodekan menggunakan *Gamma code* dan setiap *pattern Id* dari CSP dalam *inverted list* menggunakan *Huffman code*, dimana *Huffman code* ini sesuai untuk model sebaran dari *pattern Id* (Chen dan Cook, 2007).

Menurut Chen dan Cook (2007), algoritma kompresi *inverted list* menggunakan CSP berdasarkan *Gamma* adalah :

*Input* : sebuah *inverted list* dalam representasi *cluster*.

*Ouput* : CSP berdasarkan *Gamma code* untuk daftar tersebut.

Metode :

1. Mengkodekan jumlah total CSP dalam *inverted list* menggunakan *Gamma code*.
2. Mengkodekan posisi setiap kemunculan CSP dalam representasi *cluster* dari *list* yang secara sequensial menggunakan *Gamma code*.
3. Mengkodekan setiap item pada daftar *sequential*. Jika item tersebut adalah *integer* dikodekan dengan *Gamma code*. Sebaliknya jika item tersebut adalah sebuah CSP, *pattern Id* dikodekan dengan *Huffman code*.

## 2.5 Algoritma *Huffman*

Algoritma *Huffman* menggunakan prinsip pengkodean yang mirip dengan kode *morse*, yaitu tiap karakter (simbol) dikodekan hanya dengan rangkaian beberapa bit, dimana karakter yang sering muncul dikodekan dengan rangkaian bit yang pendek dan karakter yang jarang muncul dikodekan dengan rangkaian bit yang lebih panjang. (Wardoyo, dkk., 2005).

### 2.5.1 Pembentukan Pohon *Huffman*

Kode *Huffman* pada dasarnya merupakan kode prefiks (*prefix code*). Kode prefiks adalah himpunan yang berisi sekumpulan kode biner, dimana pada kode prefik ini tidak ada kode biner yang menjadi awal bagi kode biner yang lain. Kode prefiks biasanya direpresentasikan sebagai pohon biner yang diberikan nilai atau label. Untuk cabang kiri pada pohon biner diberi label 0, sedangkan pada cabang kanan pada pohon biner diberi label 1. Rangkaian bit

yang terbentuk pada setiap lintasan dari akar ke daun merupakan kode prefiks untuk karakter yang berpadanan. Pohon biner ini biasa disebut pohon *Huffman*. Langkah-langkah pembentukan pohon *Huffman* menurut Wardoyo (2005) adalah sebagai berikut :

1. Baca semua karakter di dalam teks untuk menghitung frekuensi kemunculan setiap karakter. Setiap karakter penyusun teks dinyatakan sebagai pohon bersimpul tunggal. Setiap simpul di-*assign* dengan frekuensi kemunculan karakter tersebut.
2. Terapkan strategi algoritma *greedy* sebagai berikut : gabungkan dua buah pohon yang mempunyai frekuensi terkecil pada sebuah akar. Setelah digabungkan akar tersebut akan mempunyai frekuensi yang merupakan jumlah dari frekuensi dua buah pohon-pohon penyusunnya.
3. Ulangi langkah 2 sampai hanya tersisa satu buah pohon *Huffman*. Agar pemilihan dua pohon yang akan digabungkan berlangsung cepat, maka semua yang ada selalu terurut menaik berdasarkan frekuensi.

### 2.5.2 Proses *Encoding*

*Encoding* adalah cara menyusun *string* biner dari teks yang ada. Proses *encoding* untuk satu karakter dimulai dengan membuat pohon *Huffman* terlebih dahulu. Setelah itu, kode untuk satu karakter dibuat dengan menyusun nama *string* biner yang dibaca dari akar sampai ke daun pohon *Huffman* (Wardoyo, dkk., 2005).

Langkah-langkah untuk men-*encoding* suatu *string* biner menurut Wardoyo dkk (2005) adalah sebagai berikut

1. Tentukan karakter yang akan di-*encoding*
2. Mulai dari akar, baca setiap bit yang ada pada cabang yang bersesuaian sampai ketemu daun dimana karakter itu berada
3. Ulangi langkah 2 sampai seluruh karakter di-*encoding*

Sebagai contoh, dalam kode ASCII *string* 7 huruf “ABACCDA” membutuhkan representasi 7 × 8 bit = 56 bit (7 byte), dengan rincian sebagai berikut: A = 01000001, B = 01000010, A = 01000001, C = 01000011, C = 01000011, D = 01000100, A = 01000001. Dengan frekuensi kemunculan A = 3, B = 1, C = 2, dan D = 1.

Proses pembuatan pohon *Huffman*-nya ditunjukkan oleh gambar 2.5, proses *encoding* setiap karakter penyusunnya ditunjukkan oleh tabel 2.8.

**Tabel 2.8.** Kode *Hufman* untuk Karakter "ABCD".

| Karakter | String Biner *Huffman* |
|---|---|
| A | 0 |
| B | 110 |
| C | 10 |
| D | 111 |

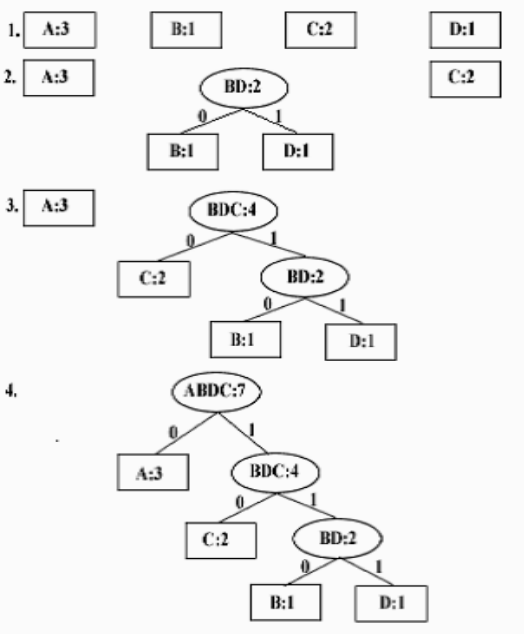
### 2.5.3 Proses *Decoding*

*Decoding* merupakan kebalikan dari *encoding*. *Decoding* berarti menyusun kembali data dari *string* biner menjadi sebuah karakter kembali. *Decoding* dapat dilakukan dengan dua cara, yang pertama dengan menggunakan pohon *Huffman* dan yang kedua dengan menggunakan tabel kode *Huffman* (Wardoyo, dkk., 2005).

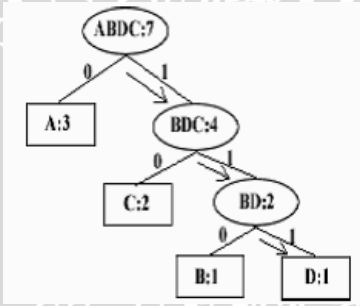
Langkah-langkah men-*decoding* suatu *string* biner dengan menggunakan pohon *Huffman* menurut Wardoyo dkk (2005) adalah sebagai berikut:

1. Baca sebuah bit dari *string* biner.
2. Mulai dari akar
3. Untuk setiap bit pada langkah 1, lakukan *traversal* pada cabang yang bersesuaian.
4. Ulangi langkah 1, 2 dan 3 sampai bertemu daun. Kodekan rangkaian bit yang telah dibaca dengan karakter di daun.
5. Ulangi dari langkah 1 sampai semua bit di dalam *string* habis.

Cara yang kedua adalah dengan menggunakan tabel kode *Huffman*. Sebagai contoh untuk merepresentasikan *string* "ABACCDA" dengan menggunakan tabel 2.8 akan direpresentasikan menjadi rangkaian bit 0110010101110 (Wardoyo, dkk., 2005). Proses *decoding*-nya ditunjukkan oleh gambar 2.6.




**Gambar 2.5**. Pohon *Huffman* untuk Karakter "ABACCDA".




**Gambar 2.6**. Proses *Decoding* dengan Menggunakan Pohon *Huffman*.


UNIVERSITAS BRAWIJAYA

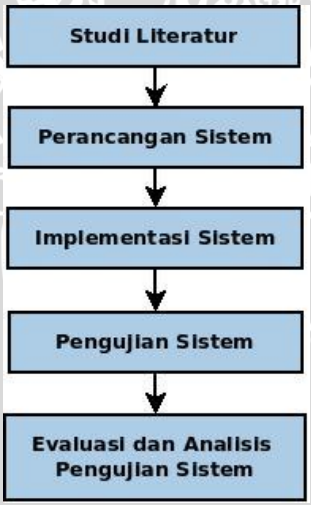
# BAB III
# METODOLOGI DAN PERANCANGAN

Pada bab ini akan dibahas metode dan langkah-langkah yang digunakan untuk kompresi *inverted list* berdasarkan *contiguous sequential patterns* (CSP) menggunakan kombinasi *Golomb* dan *Huffman*.

Penelitian dilakukan dengan langkah-langkah sebagai berikut:

1. Melakukan studi literatur yang berkaitan dengan kompresi *inverted list*, *sequential pattern mining*, algoritma *Apriori*, algoritma *UpDown Tree*, algoritma *Golomb*, dan algoritma *Huffman*.
2. Merancang sistem untuk kompresi *inverted list*.
3. Implementasi sistem berdasarkan analisis dan perancangan yang dilakukan.
4. Melakukan pengujian terhadap sistem.
5. Melakukan evaluasi tingkat keberhasilan sistem dan analisis hasil pengujian.

Langkah-langkah yang digunakan ditunjukkan oleh gambar 3.1.




**Gambar 3.1.** Diagram Alir Langkah-Langkah Penelitian.

### 3.1 Analisis Data

Koleksi dokumen yang digunakan dalam penelitian ini merupakan artikel-artikel berita berbahasa Indonesia dalam format *.txt. Artikel-artikel berita tersebut bersumber dari media *online* http://www.kompas.com yang diambil mulai tanggal 1 Maret 2012 hingga 31 Maret 2012, dengan total jumlah dokumen sebanyak 1000 dokumen. Untuk memperoleh artikel-artikel tersebut penulis menyimpan secara manual setiap halamannya satu per satu dengan format *.html kemudian setiap *file*-nya dikonversi ke bentuk teks (format *.txt) dengan menghilangkan *tag html*-nya menggunakan *jsoup*. *Jsoup* merupakan *parser html* yang digunakan untuk bahasa pemrograman Java, diunduh dari http://www.jsoup.org.

### 3.2 Perancangan Sistem

Dalam subbab ini akan dijelaskan deskripsi secara umum dan batasan sistem yang digunakan untuk kompresi *inverted list* berdasarkan pendeteksian *sequential pattern mining* pada seluruh *list*-nya.

#### 3.2.1 Deskripsi Umum Sistem

Sistem yang akan dibangun untuk kompresi *inverted list* ini terbagi menjadi dua sistem, yaitu:

1. Sistem Kompresi.

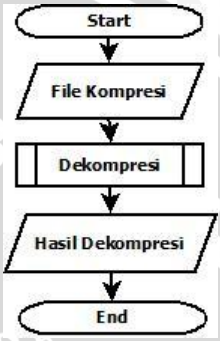
   Sistem kompresi ini bertujuan untuk menghasilkan ukuran *file* dari *inverted list* yang jauh lebih kecil daripada *file* aslinya sehingga dapat mengurangi ruang penyimpanannya. Langkah pertama sistem kompresi ini, koleksi dokumen berbahasa Indonesia dimasukkan dalam sistem. Kemudian pada langkah selanjutnya dilakukan proses *preproccessing* yang terdiri dari sub proses *case folding* dan tokenisasi, penghapusan *stop word*, dan *stemming*. Setelah *preproccessing* dilanjutkan dengan proses *indexing*, proses pendeteksian CSP set dari *inverted list*, dan terakhir dilakukan proses kompresi. Sistem kompresi ditunjukkan oleh gambar 3.2.

2. Sistem Dekompresi.

   Pada sistem dekompresi bertujuan mengembalikan *file* dari *inverted list* menjadi bentuk semula (sebelum dikompresi). Langkah

sistem dekompresi ini adalah *file* dari *inverted list* yang terkompresi dari *disk* dimasukkan dalam sistem kemudian dilakukan proses dekompresi. Sistem dekompresi ditunjukkan oleh gambar 3.3.


**Gambar 3.2.** *Flowchart* Sistem Kompresi.




**Gambar 3.3.** *Flowchart* Sistem Dekompresi.

### 3.2.2 Batasan Sistem

Batasan-batasan sistem yang digunakan pada penelitian ini, antara lain:

1. Koleksi dokumen dalam bentuk format *.txt dan berada dalam media penyimpanan lokal (*local disk*).
2. Sistem diimplementasi menggunakan bahasa pemrograman Java berupa *desktop application.*

### 3.3 Perancangan Proses

#### 3.3.1 *Preproccessing*

Pada tahap *preproccessing* meliputi *case folding*, tokenisasi, penghapusan *stop word*, dan *stemming*.

##### 3.3.1.1 *Case Folding* dan Tokenisasi

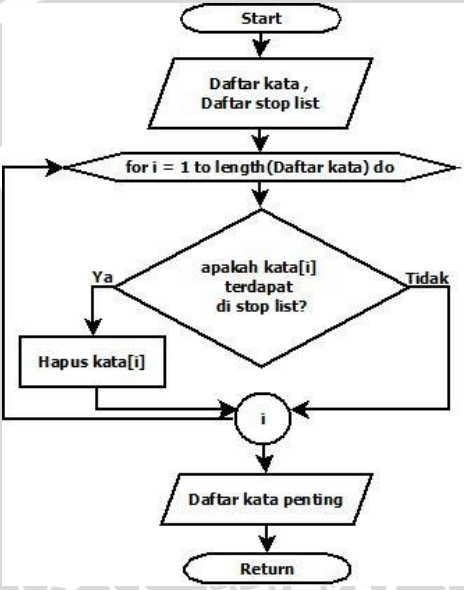
*Case folding* dan tokenisasi bertujuan mengubah seluruh huruf dalam dokumen menjadi huruf kecil (*lower case*), kemudian dokumen dipecah menjadi per kata (*term*) dan membuang tanda baca maupun karakter angka. Langkah pertama dokumen dibaca oleh sistem, kemudian dilakukan proses pengubahan seluruh isi dokumen menjadi huruf kecil semua. Pada dokumen yang sudah di-*case folding*, dilakukan pemecahan teks menjadi kata-kata kemudian sistem akan dihasilkan sebuah daftar kata. Proses ini ditunjukkan oleh gambar 3.4.


**Gambar 3.4.** *Flowchart* Proses *Case Folding* dan Tokenisasi.

##### 3.3.1.2 Penghapusan *Stop Word*

Penghapusan *Stop Word* bertujuan menghilangkan kata-kata yang tidak penting**.** Proses penghapusannya membutuhkan daftar *stop list* sebagai acuan penghapusan. *Stop list* merupakan daftar kata yang

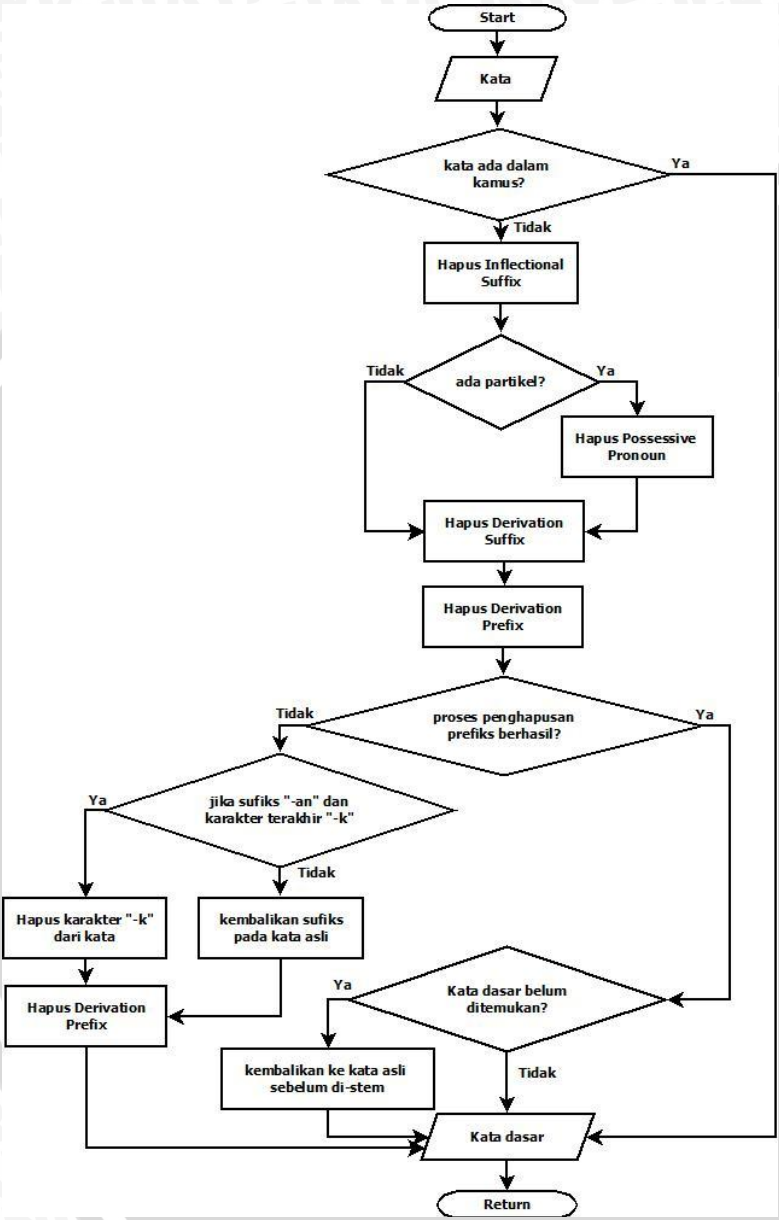
tidak penting dan daftar *stop list* ini di-*download* dari sumber http://fpmipa.upi.edu/staff/yudi/stop_words_list.txt. Setiap kata pada daftar kata diperiksa apakah ada dalam *stop list* jika ada maka dilakukan penghapusan kata. Proses ini ditunjukkan oleh gambar 3.5.




**Gambar 3.5.** *Flowchart* Proses Penghapusan *Stop Word*.

### 3.3.1.3 *Stemming*

*Stemming* bertujuan mencari kata dasar dari kata-kata pada dokumen. Algoritma yang dipakai untuk *stemming* ini adalah algoritma Nazief dan Adriani. Algortima *stemming* Nazief-Adriani ini membutuhkan pengecekan kamus kata dasar. Kamus kata dasar yang dipakai sebanyak 31.243 kata, yang diperoleh dari sumber http://bahasa.kemdiknas.go.id/kbbi. Langkah-langkah pada proses ini menggunakan aturan *stemming* yang telah dijelaskan pada subbab 2.1.4.1. Secara garis besar proses *stemming* ditunjukkan oleh gambar 3.6.




**Gambar 3.6.** *Flowchart* Proses *Stemming* Nazief-Adriani.

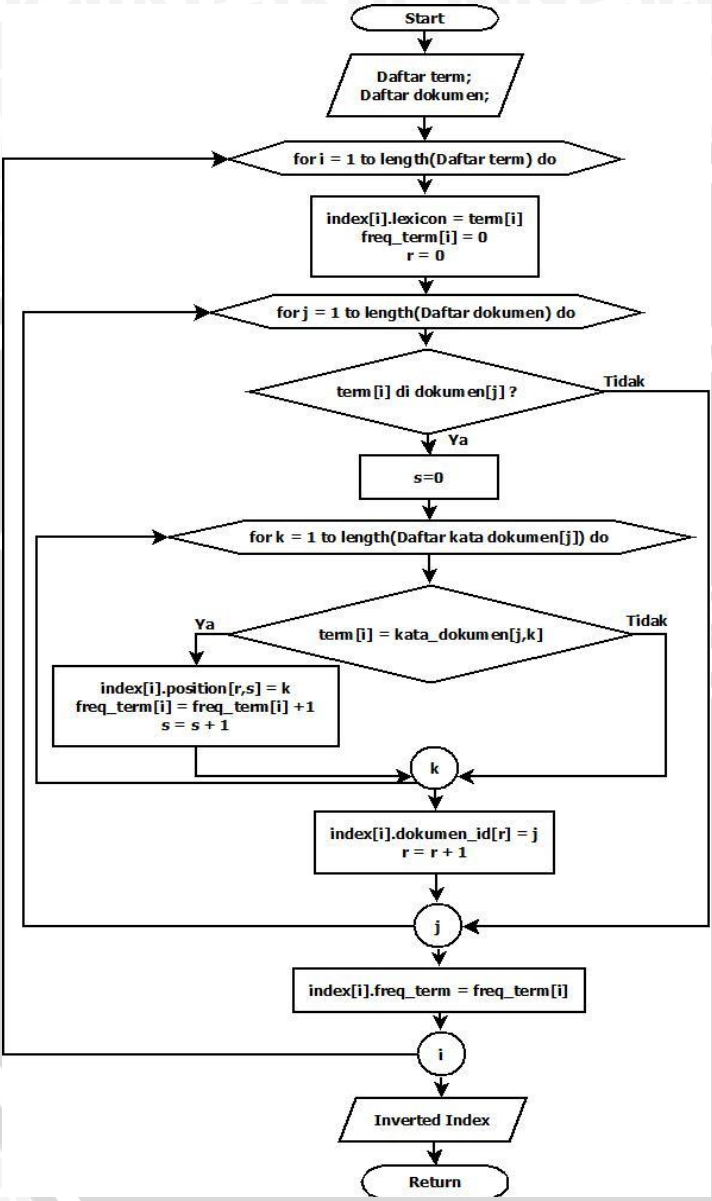
### 3.3.2 *Indexing*

Proses *indexing*, proses ini bertujuan untuk membentuk *inverted index*. *Inverted index* ini akan dibagi menjadi dua bagian yaitu *lexicon* (*dictionary* dari *term*) dan *inverted list* yang terdiri dari daftar *document identifiers*, daftar *term frequencies*, dan daftar *term positions*. *Inverted index* yang dihasilkan kemudian disimpan dalam *local disk*, dimana *file* dari *lexicon* dan *inverted list* disimpan secara terpisah. Data masukkan proses ini berupa daftar *term* unik yang dihasilkan oleh proses *preprocessing*. Langkah pertama pada proses ini adalah dilakukan iterasi terhadap daftar *term* dan setiap *term*-nya disimpan dalam *inverted index* pada daftar *lexicon*. Kemudian langkah keduanya adalah dilakukan pendeteksian kemunculan atau keberadaan setiap *term* dalam setiap dokumen dan setiap kemunculan *term*-nya ditandai dengan nomor identitas dari setiap dokumen yang disimpan dalam *inverted list* pada daftar *document Id*. Pada langkah kedua juga dilakukan pengecekan posisi setiap *term*-nya yang disimpan pada daftar *term positions* dan dilakukan perhitungan frekuensi setiap *term* yang disimpan pada *daftar term frequencies*.

Proses *indexing* ini ditunjukkan oleh gambar 3.7, dimana variabel *r* merupakan jumlah iterasi dari daftar *document Id*, jika *term*[*i*] ditemukan pada dokumen[*j*] sedangkan variabel *s* merupakan jumlah iterasi posisi kemunculan *term*[*i*] pada dokumen[*j*]. Variabel *freq_term*[*i*] berfungsi untuk menyimpan jumlah frekuensi total *term*[*i*] pada seluruh dokumen.

### 3.3.3 Pendeteksian CSP (*Contiguous Sequential Pattern*)

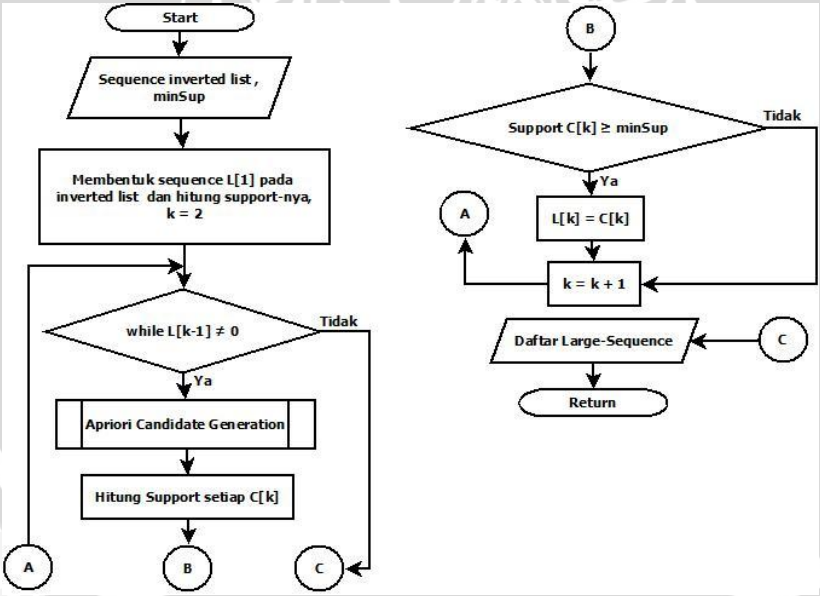
Tujuan proses ini yaitu dilakukannya pendeteksian *sequential pattern* yang terdapat dalam daftar *integer* dari *inverted list*. Langkah pertama adalah mencari *sequential pattern* dengan aturan *Apriori* yang telah dijelaskan pada subbab 2.3.1. Kemudian seluruh *sequential pattern* yang dihasilkan *apriori* diefisiensi dengan menggunakan struktur data *UpDown Tree* yang dijelaskan pada subbab 2.3.3.




**Gambar 3.7.** *Flowchart* Proses *Indexing*.

### 3.3.3.1 Proses *Apriori*

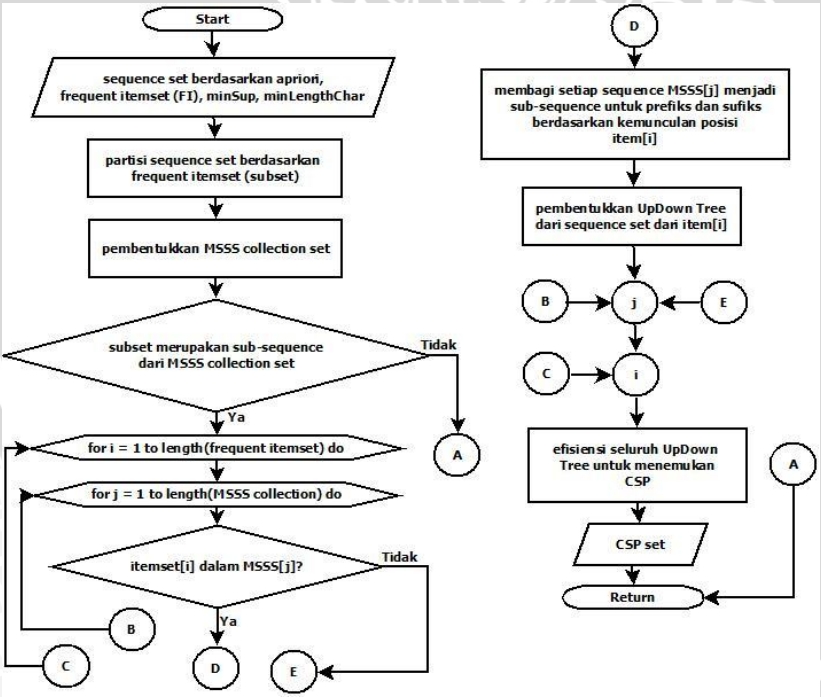
Terdapat beberapa tahapan untuk *Apriori* yaitu tahap pertamanya adalah dibentuk *Large* 1-*Sequence* (L1) dari setiap item *integer* pada *inverted list*. Selanjutnya tahap kedua, membentuk *Candidate* (*k*-1)-*Sequence* yakni *Candidate* 2-*Sequence* (C2), *Candidate* 3-*Sequence* (C3) sampai tidak ditemukan *candidate sequential pattern* lagi dan dihitung juga *support* atau frekuensinya pada *inverted list* tersebut. Untuk tahap kedua berdasarkan subbab 2.3.2, terdapat proses *apriori candidate generation*. *Apriori candidate generation* ini mempunyai dua bagian proses utama yaitu *join step* dan *prune step*, dari dua proses menghasilkan *Candidate* (*k*-1)-*Sequence*. Setelah *Candidate* (*k*-1)-*Sequence* dibentuk, kemudian pada tahap terakhir dibandingkan *support* untuk *Candidate* (*k*-1)-*Sequence* dengan *minSup* (*minimum support*), jika lebih besar dari *minSup*-nya maka akan menjadi *Large* (*k*-1)-*Sequence*. Dimana *k* adalah jumlah iterasi dari pencarian *sequential pattern* dari *apriori*. Hasil akhir proses ini berupa daftar *sequence set* yang merupakan gabungan seluruh *large sequence*. Proses ini ditunjukkan oleh gambar 3.8.




**Gambar 3.8.** *Flowchart* Proses *Apriori*.

### 3.3.3.2 *Updown Tree*

Proses *UpDown Tree* bertujuan untuk mengefisiensi *sequence set* yang ditemukan pada *Apriori* mengikuti aturan pada subbab 2.3.3. Tahap pertamanya adalah mengambil *Large* 1-*Sequence* dari *Apriori* sebagai *Frequent Itemset* (FI) kemudian mempartisi *sequence set* menjadi *subset* berdasarkan setiap FI-nya. Tahap kedua adalah pembentukan MSSS (*Maximal Sub-Sequence Set*) yang dijelaskan pada subbab 2.3.3.3, kemudian dilakukan pengecekan kondisi, jika *subset* merupakan *sub-sequence* dari MSSS maka CSP set dapat ditemukan, tapi jika sebaliknya proses berhenti. Jika kondisi terpenuhi, dilakukan proses efisiensi *sequence dari* MSSS menggunakan *UpDown Tree*. Hasil akhirnya berupa CSP set yang digunakan untuk proses kompresi. Proses ini ditunjukkan oleh gambar 3.9.




**Gambar 3.9.** *Flowchart* Proses *UpDown Tree.*

### 3.3.4 Kompresi

Proses kompresi mempunyai aturan jika daftar *inverted list* berupa sebuah *pattern Id* dari CSP yang ditemukan maka dikodekan dengan *Huffman code* sedangkan jika hanya berupa sebuah *integer* dan bukan berupa CSP maka dikodekan dengan *Gamma* atau *Golomb* (dijelaskan pada subbab 2.4). Informasi posisi CSP pada *list* disimpan dalam *header* yang akan digunakan pada saat dekompresi.

#### 3.3.4.1 *Huffman Encoding*

Langkah pertama *Huffman encoding* adalah mengurutkan (*sorting*) secara *descending* setiap *sequence* dari CSP beserta frekuensinya (*support*-nya) kemudian membentuk pohon biner (*tree*). Implementasi *sorting* yang dipakai adalah fungsi *Arrays.Sort*() dari utilitas bahasa pemrograman Java. Dilakukan iterasi sampai ditemukan jika hanya ada 1 pohon *Huffman* yang terbentuk, setiap iterasi dilakukan proses penggabungan dua pohon yang mempunyai jumlah frekuensi terkecil kemudian diurutkan. Tabel *Huffman* berisikan kode *Huffman* untuk setiap CSP yang terdapat pada *tree*. Untuk pengkodean setiap *sequence* dari CSP yaitu dengan mencocokkan tabel *Huffman* dengan *sequence* sehingga didapatkan kode binernya. Proses pembentukkan tabel ditunjukkan oleh gambar 3.10.

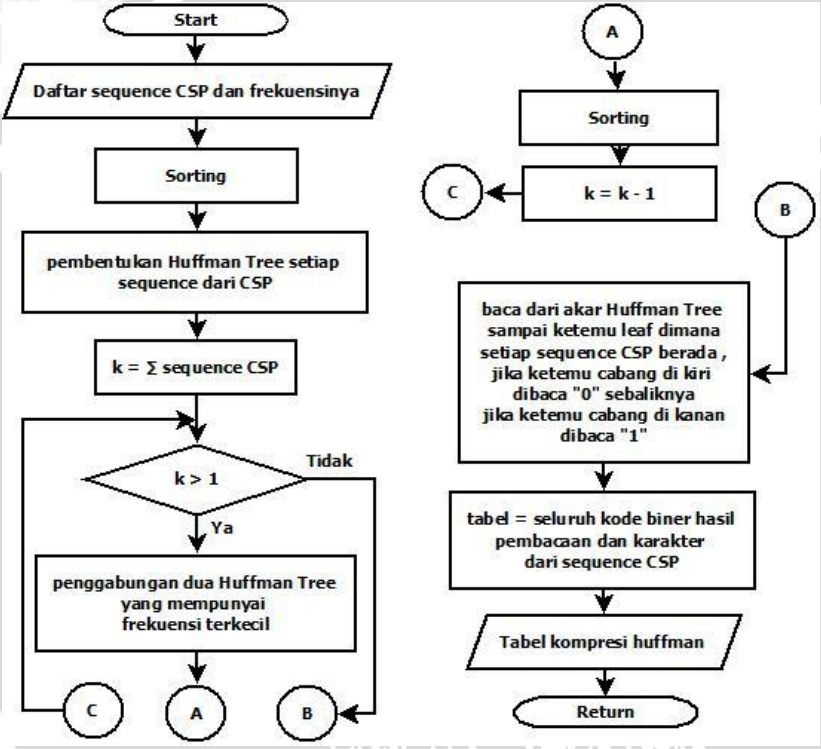
#### 3.3.4.2 *Gamma Encoding*

Langkah-langkah *Gamma encoding* menggunakan aturan yang dijelaskan pada subbab 2.2.1.2, yaitu sebuah *integer x* sebagai data masukan dikodekan dalam dua bagian. Bagian pertama, dihitung $\lfloor \log_2 x \rfloor + 1$ dalam representasi *unary* (subbab 2.2.1.1). Kemudian bagian keduanya dikodekan $x - 2^{\lfloor \log_2 x \rfloor}$ dalam bentuk biner dengan panjang bit $\lfloor \log_2 x \rfloor$. Hasil akhirnya menggabungkan angka biner di bagian pertama dan kedua yang tiada lain merupakan gabungan *unary* dan *binary*. Proses ini ditunjukkan oleh gambar 3.11.

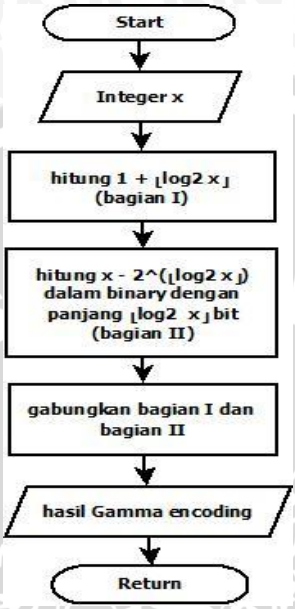
#### 3.3.4.3 *Golomb Encoding*

Langkah-langkah *Golomb encoding* menggunakan aturan yang dijelaskan pada subbab 2.2.1.3, yaitu sebuah *integer n* dan konstanta *m* sebagai data masukan dikodekan dalam dua bagian. Bagian

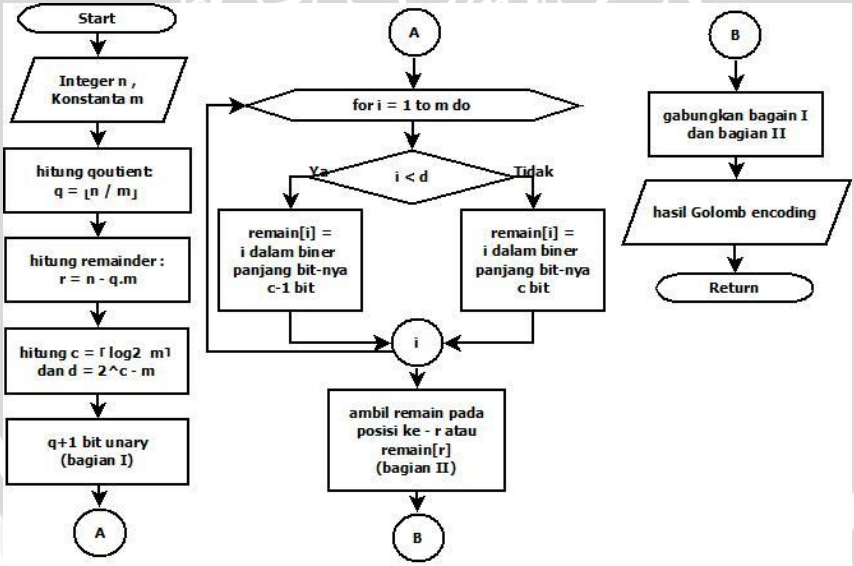
pertama, dihitung *quotient* (*q*) menggunakan persamaan 2.2 dan *q*+1 direpresentasikan dalam bentuk *unary* (subbab 2.2.1.1). Kemudian bagian keduanya dihitung *remainder* (*r*) menggunakan persamaan 2.3 dan *r* dikodekan secara khusus dalam bentuk biner (subbab 2.2.1.3). Hasil akhirnya menggabungkan angka biner di bagian pertama dan kedua. Proses ini ditunjukkan oleh gambar 3.12.




**Gambar 3.10.** *Flowchart* Proses Pembentukan Tabel *Huffman*.



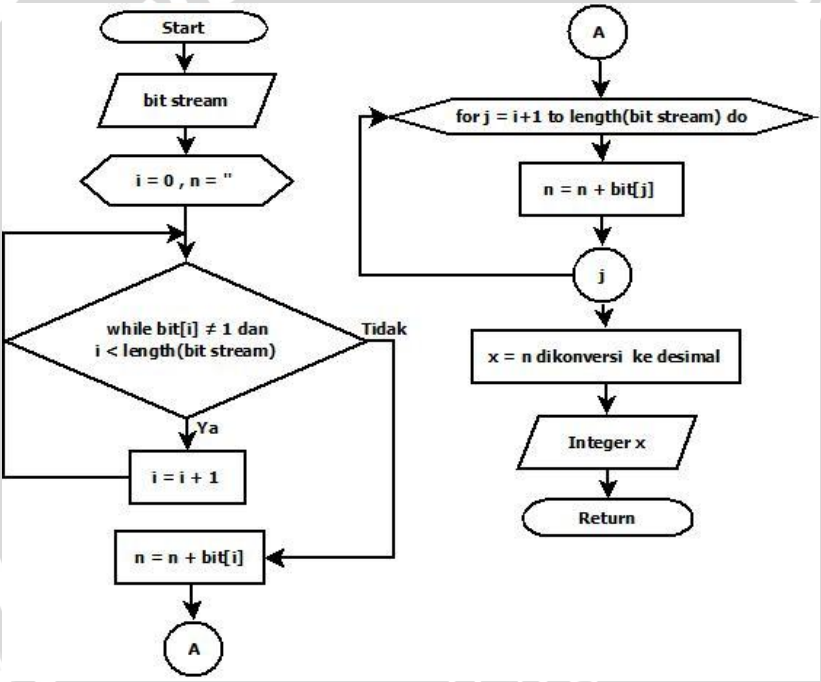

**Gambar 3.11.** *Flowchart* Proses *Gamma Encoding*.



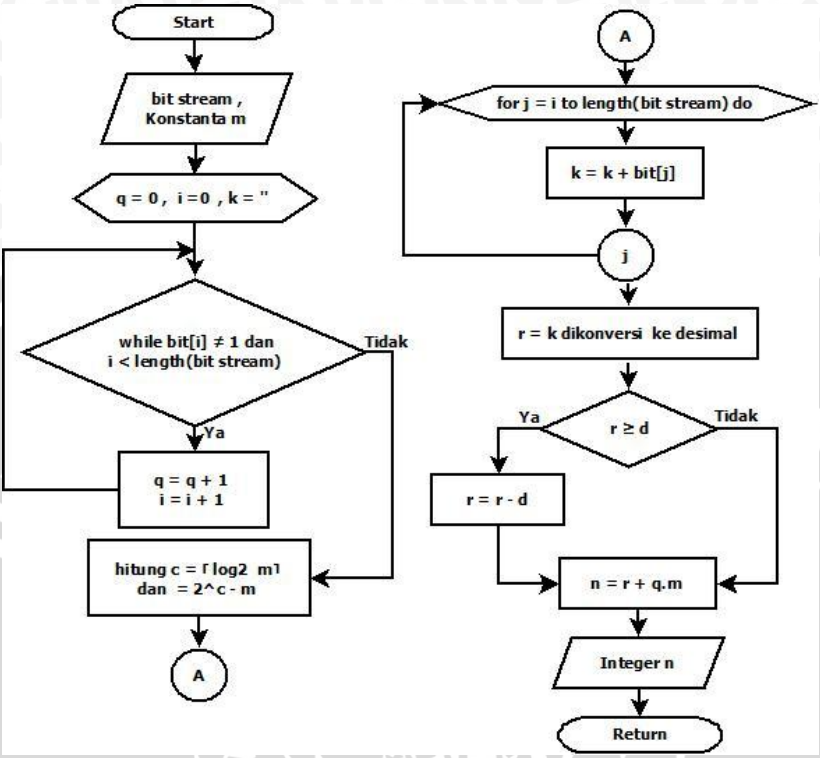

**Gambar 3.12.** *Flowchart* Proses *Golomb Encoding*.

### 3.3.5 Dekompresi

Proses dekompresi bertujuan mengembalikan *file* hasil kompresi ke bentuk asli sebelum dikompresi. Langkah pertama dilakukan pembacaan *header* dan *file* yang memuat tabel *Huffman*. *Header* ini berisi seluruh informasi posisi CSP dalam *list*. Langkah selanjutnya dilakukan pembacaan *bit stream* pada *file* kompresi yang memiliki aturan yaitu jika ditemukan posisi CSP pada *bit stream* akan didekodekan dengan *Huffman*. Jika sebaliknya maka didekodekan dengan *Gamma* ataupun *Golomb*. Proses dekompresi Untuk *Gamma*, dan *Golomb* ditunjukkan oleh gambar 3.13, dan 3.14.




**Gambar 3.13.** *Flowchart* Proses *Gamma Decoding.*




**Gambar 3.14.** *Flowchart* Proses *Golomb Decoding.*

## 3.4 Contoh Perhitungan Manual

### 3.4.1 Contoh Proses *Indexing*

Misalkan diberikan 5 buah dokumen yang ditunjukkan oleh tabel 3.1. Diasumsikan bahwa sebelum proses *indexing* sudah dilakukan *preproccessing* terlebih dahulu yaitu *case folding* dan tokenisasi, penghapusan *stopword*, dan *stemming* seperti yang sudah dijelaskan pada subbab 3.3.1. Contoh hasil *indexing* dari dokumen tabel 3.1 ini ditunjukkan oleh tabel 3.2 yang terdiri dari *lexicon* atau *term*, total frekuensi, *document Id*, dan *d-gap document Id*. *D-gap document Id* ini yang akan digunakan untuk pendeteksian CSP *mining*.

**Tabel 3.1.** Contoh Beberapa Dokumen.

| *Document Id* | Teks |
|---|---|
| 1 | *Bakal calon Gubernur DKI Jakarta Alex Noerdin mengatakan, menjabat gubernur di DKI Jakarta lebih mudah dibandingkan menjadi gubernur di wilayah lain.* |
| 2 | *Bakal calon Gubernur DKI Jakarta Alex Noerdin membantah, bahwa ia dan pasangannya, Nono Sampono, hanya asal umbar janji bisa mengatasi berbagai masalah klise di Jakarta dalam waktu tiga tahun.* |
| 3 | *Di tengah ingar bingar pencalonannya sebagai calon Gubernur DKI Jakarta, Alex Noerdin tiba-tiba saja dihantam isu tak sedap.* |
| 4 | *Nama Gubernur Sumatera Selatan Alex Noerdin, tiba-tiba saja menyeruak di antara para bakal calon gubernur DKI Jakarta.* |
| 5 | *Gubernur Sumatera Selatan, Alex Noerdin optimistis maju dalam pemilihan umum kepala daerah (Pilkada) DKI Jakarta yang berlangsung tahun ini.* |

**Tabel 3.2.** Contoh Hasil *Indexing* dari Tabel 3.1.

| No. | *Lexicon* | Total | *Document Id* | *D-gap Document Id* |
|---|---|---|---|---|
| 1 | Alex | 5 | [1, 2, 3, 4, 5] | [1, 1, 1, 1, 1] |
| 2 | Atas | 1 | [2] | [2] |
| 3 | Bakal | 3 | [1, 2, 4] | [1, 1, 2] |
| 4 | Banding | 1 | [1] | [1] |
| 5 | Bantah | 1 | [2] | [2] |
| 6 | Bingar | 1 | [3] | [3] |
| 7 | Calon | 5 | [1, 2, 3, 4] | [1, 1, 1, 1] |
| 8 | Daerah | 1 | [5] | [5] |
| 9 | Dki | 6 | [1, 2, 3, 4, 5] | [1, 1, 1, 1, 1] |
| 10 | gubernur | 8 | [1, 2, 3, 4, 5] | [1, 1, 1, 1, 1] |
| 11 | hantam | 1 | [3] | [3] |
| 12 | ingar | 1 | [3] | [3] |
| 13 | Isu | 1 | [3] | [3] |

| | | | | |
|---|---|---|---|---|
| 14 | Jabat | 1 | [1] | [1] |
| 15 | Jakarta | 7 | [1, 2, 3, 4, 5] | [1, 1, 1, 1, 1] |
| 16 | Janji | 1 | [2] | [2] |
| 17 | Kepala | 1 | [5] | [5] |
| 18 | Klise | 1 | [2] | [2] |
| 19 | Maju | 1 | [5] | [5] |
| 20 | Mudah | 1 | [1] | [1] |
| 21 | Noerdin | 5 | [1, 2, 3, 4, 5] | [1, 1, 1, 1, 1] |
| 22 | Nono | 1 | [2] | [2] |
| 23 | Optimistis | 1 | [5] | [5] |
| 24 | Pasang | 1 | [2] | [2] |
| 25 | Pilih | 1 | [5] | [5] |
| 26 | Pilkada | 1 | [5] | [5] |
| 27 | Sampono | 1 | [2] | [2] |
| 28 | Sedap | 1 | [3] | [3] |
| 29 | Seruak | 1 | [4] | [4] |
| 30 | Sumatera | 2 | [4, 5] | [4, 1] |
| 31 | Tiba | 4 | [3, 4] | [3, 1] |
| 32 | Umbar | 1 | [2] | [2] |
| 33 | Wilayah | 1 | [1] | [1] |

### 3.4.2 Contoh Proses Pendeteksian CSP

Pada tabel 3.2 didapatkan *sequence set* dari *inverted list* yang berasal dari daftar *d-gap document Id* sebagai data masukan untuk pendeteksian CSP-nya.

#### 3.4.2.1 Contoh *Apriori*

Langkah pertama pendeteksian *sequence* CSP adalah dengan mencari seluruh *sequence* pada *inverted list* yang pada *document Id* berdasarkan algoritma *Apriori* dengan nilai *minimum support*-nya sebesar 5% dari 33 jumlah *inverted list*-nya (33 x 5% = 1,65, pembulatan ke bawah menjadi 1) dan panjang minimal karakter CSP adalah 2. Untuk menghitung jumlah *support* atau frekuensi dari setiap *sequence* yang berupa *integer* digunakan aturan yang

dijelaskan pada subbab 2.4 pada langkah ke-3. Secara garis besar aturan pendeteksian CSP mengikuti aturan yang ada pada subbab 2.4. Pembentukan *Large* 1-*Sequence* atau *Frequent Itemset* (FI) ditunjukkan oleh tabel 3.3. Hasil akhir seluruh *sequence* yang ditemukan pada proses *Apriori* ini ditunjukkan oleh tabel 3.4, yang kemudian akan diefisiensi menggunakan *UpDown Tree*.

**Tabel 3.3.** *Frequent Itemset* (FI) dari Tabel 3.2.

| 1-*Sequence* | *Support* |
|---|---|
| 1 | 21 |
| 2 | 10 |
| 3 | 13 |
| 4 | 9 |
| 5 | 14 |

**Tabel 3.4.** Seluruh *Sequence Set* Pada *Apriori* dari Tabel 3.2.

| *Sequence Id* | *Sequence* | *Support* |
|---|---|---|
| 1 | 1 | 37 |
| 2 | 2 | 9 |
| 3 | 3 | 6 |
| 4 | 4 | 2 |
| 5 | 5 | 6 |
| 6 | 12 | 1 |

### 3.4.2.2 Contoh Partisi *Subset* dan MSSS *Collection*

Partisi *subset* adalah mengelompokkan seluruh *sequence* pada proses *Apriori* berdasarkan FI-nya (elemen tunggal yang diperoleh dari *Large* 1-*Sequence*). Partisi ini digunakan sebagai pengecekan terhadap MSSS *collection* dari daftar *d-gap document Id* pada tabel 3.2, apakah subset merupakan *sub-sequence* dari MSSS tersebut. Sebagai contoh, dikelompokkan *sequence* pada tabel 3.4 berdasarkan kehadiran item 2, jika ditemukan item 2 pada *sequence set* dibentuk *sequence subset* item 2 dan setiap *sequence* pada tabel 3.4 setiap item penyusunnya harus lebih kecil atau sama dengan item 2. Proses yang sama juga dilakukan untuk item 1, 3, 4, dan 5. Contoh pembentukan

*sequence subset* item 2 ditunjukkan oleh tabel 3.5 sedangkan MSSS dari daftar *d-gap document Id* pada tabel 3.2 ditunjukkan oleh gambar 3.6 dan pengelompokkan MSSS dari item 1 dan 2 ditunjukkan oleh gambar 3.7 dan 3.8.

**Tabel 3.5.** *Subset* dari Item 2.

| No. | *Sequence* |
|---|---|
| 1 | 2 |
| 2 | 12 |

**Tabel 3.6.** MSSS Collection dari Daftar *D-gap Document Id* Pada Tabel 3.2.

| *Id* | *Sequence* |
|---|---|
| 1 | 11111 |
| 2 | 112 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |

**Tabel 3.7.** Pengelompokkan MSSS Berdasarkan Item 1.

| No. | *Sequence* | *Set Id* | *Prefix* | *Suffix* |
|---|---|---|---|---|
| 1 | 11111 | 1 | 1 | 11111 |
| 2 | 112 | 2 | 1 | 112 |

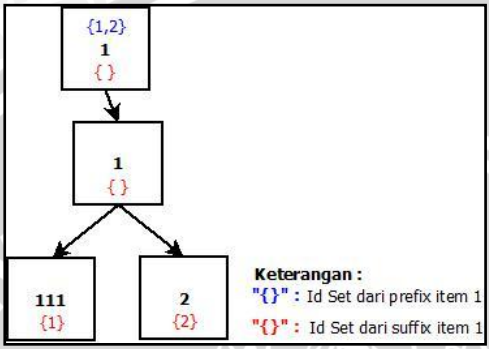
**Tabel 3.8.** Pengelompokkan MSSS Berdasarkan Item 2.

| No. | *Sequence* | *Set Id* | *Prefix* | *Suffix* |
|---|---|---|---|---|
| 1 | 112 | 2 | 112 | 2 |

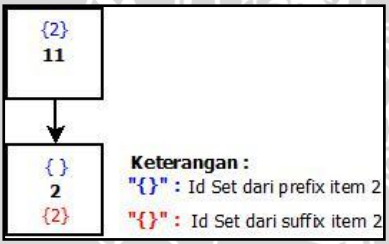
### 3.4.2.3 Contoh Pembentukan *UpDown Tree*

*UpDown Tree* dibentuk dari kombinasi *Up Tree* dan *Down Tree* untuk setiap *Frequent Itemset* (FI). *Up Tree* dan *Down Tree* dibentuk berdasarkan *prefix* dan *suffix* dari MSSS *sequence*. Sebagai contoh pembentukan *UpDown Tree* dari tabel 3.7 dan 3.8 ditunjukkan oleh gambar 3.15 dan 3.16.

Pengecekan *UpDown Tree* dari item 1, 2, 3, 4, dan 5 berdasarkan aturan pada subbab 2.3.4.4 sehingga dihasilkan CSP set yaitu (1111) dan (112) dengan masing-masing *support*-nya yaitu 6 dan 1.




**Gambar 3.15.** *Updown Tree* dari Item 1.

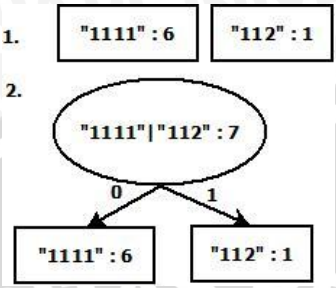



**Gambar 3.16.** *Updown Tree* dari Item 2.

### 3.4.3 Contoh Kompresi

#### 3.4.3.1 Contoh Kompresi *Huffman*

Pembentukan *Huffman tree* untuk CSP set yang ditemukan pada subbab 3.4.2.3, langkah pertamanya adalah mengurutkan setiap CSP set berdasarkan *support*-nya secara *descending* kemudian digabungkan 2 *sequence* CSP dengan nilai *support* terkecil dan dibentuk *Huffman Tree*. Proses ini berlangsung hingga menyisakan sebuah *Huffman Tree*. Hasil akhir dari kompresi *Huffman* ini adalah *sequence* (1111) dikodekan dengan 0 dan *sequence* (112) dikodekan dengan 1. Proses ini ditunjukkan oleh gambar 3.17.




**Gambar 3.17.** Contoh *Huffman Tree*.

### 3.4.3.2 Contoh Kompresi *Gamma* dan *Golomb*

Dimisalkan angka 15 akan dikompresi menggunakan *Gamma* dan *Golomb*. Dengan nilai $\log_2 15 = 3{,}9068$.

Berdasarkan subbab 2.2.3.3, contoh pengkodean *Gamma*-nya:

- $x = 15$, $\lfloor \log_2 15 \rfloor = 3$,
- $x + 1 = 3 + 1 = 4$, 4 dalam *unary* menjadi 0001.
- $15 - 2^{\lfloor \log_2 15 \rfloor} = 7$ (dalam *binary* dengan panjang bitnya sebanyak $\lfloor \log_2 15 \rfloor = 3$ bit, yaitu 010).
- hasil pengkodeannya 0001111.

Berdasarkan subbab 2.2.3.4, contoh pengodean *Golomb*-nya:

- $m$=10 (parameter), $n$=15,
- hitung *qoutient* $q$ menggunakan persamaan 2.2, $q = \left\lfloor \frac{15}{10} \right\rfloor = \lfloor 1.5 \rfloor = 1$, $q$+1=1+1=2 dikonversi menjadi *unary* yaitu 01.
- hitung *remainder* $r$ menggunakan persamaan 2.3, $r = 15 - 1 * 10 = 5$,
- hitung $c$ menggunakan persamaan 2.4, $c = \lceil \log_2 10 \rceil = 4$,
- pengecekan apakah $r < d$, jika benar maka jumlah bitnya adalah $c$ bit sebaliknya jumlah bitnya adalah ($c$-1) bit , $5 < 2^4 - 10 = 6$ hasilnya lebih kecil $r$ sehingga $r$ nilainya tetap, kemudian $r$ dikonversi ke *binary* menjadi 101 (panjangnya 3 bit).
- Hasil pengodeannya berdasarkan penggabungan $q$ dan $r$ adalah 01101.

#### 3.4.3.3 Kompresi *Inverted List dengan* CSP

Berdasarkan penjelasan subbab 2.4, CSP set yang ditemukan pada subbab 3.4.2.3 sebelum dilakukan kompresi, langkah pertamanya adalah mentransformasikan *inverted list* (terdapat pada tabel 3.2) yang memuat CSP yang ditunjukkan oleh tabel 3.9. Kemudian hasil kompresi dari tabel 3.9 yang mengunakan pengkodean *Huffman* dan *Golomb* ditunjukkan oleh tabel 3.10.

**Tabel 3.9.** Hasil Transformasi *Inverted List* Berdasarkan CSP Set.

| Posisi *List* | Jumlah *Patern* | Posisi *Pattern* | Hasil Transformasi |
|---|---|---|---|
| 1 | 1 | [1] | [1111, 1] |
| 3 | 1 | [1] | [112] |
| 7 | 1 | [1] | [1111] |
| 9 | 1 | [1] | [1111, 1] |
| 10 | 1 | [1] | [1111, 1] |
| 15 | 1 | [1] | [1111, 1] |
| 21 | 1 | [1] | [1111, 1] |

**Tabel 3.10.** Hasil Kompresi dari Tabel 3.9.

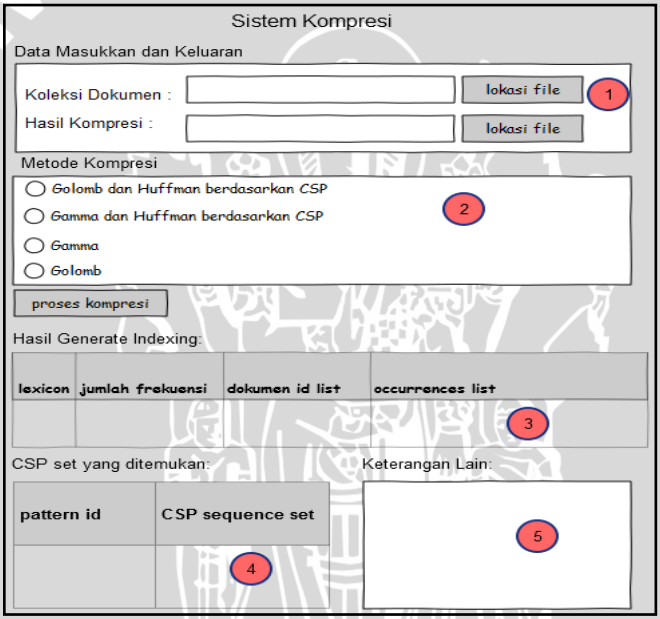
| Posisi *List* | Jumlah *Patern* | Posisi *Pattern* | Hasil Transformasi |
|---|---|---|---|
| 1 | 1001 | [1001] | [0, 1001] |
| 3 | 1001 | [1001] | [1] |
| 7 | 1001 | [1001] | [0] |
| 9 | 1001 | [1001] | [0, 1001] |
| 10 | 1001 | [1001] | [0, 1001] |
| 15 | 1001 | [1001] | [0, 1001] |
| 21 | 1001 | [1001] | [0, 1001] |

### 3.5 Rancangan *User Interface*

Rancangan *user interface* untuk sistem kompresi *inverted list* ditunjukkan oleh gambar 3.18 sedangkan rancangan *user interface* untuk sistem dekompresinya ditunjukkan oleh gambar 3.19.

Terdapat beberapa bagian dari *user interface* sistem kompresinya, antara lain:

1. Data masukkan dan keluaran, data masukkan berupa *file* koleksi dokumen dan data keluaran berupa *file* hasil kompresi.
2. Metode Kompresi, berupa *radiobutton* untuk memilih jenis kompresi mana yang akan digunakan oleh pengguna.
3. Hasil *Indexing*, berupa sebuah tabel yang menginformasikan daftar *inverted index*.
4. Hasil CSP set, berupa sebuah tabel yang menginformasikan pendeteksian beberapa CSP dalam *inverted list*.
5. Keterangan lain, yaitu berupa *textarea* yang menampilkan ukuran *file* asli, ukuran *file* kompresi dari *inverted list* dan menampilkan jumlah rasio kompresinya.

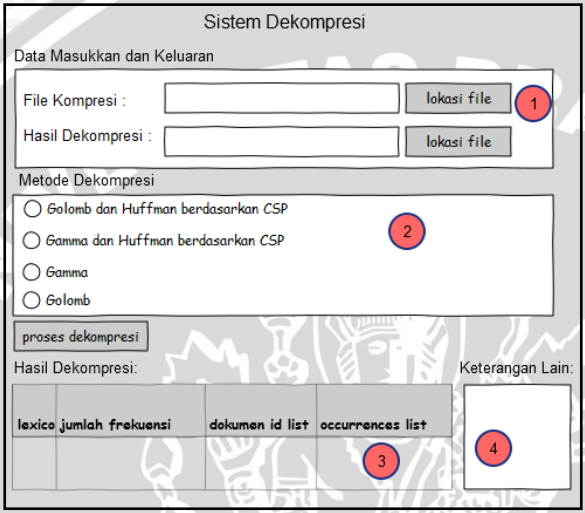



**Gambar 3.18.** Rancangan *User Interface* Sistem Kompresi.

Pada sistem dekompresinya, terdapat beberapa bagian dari *user interface*, antara lain:

1. Data masukkan dan keluaran, data masukkan berupa *file* kompresi *inverted list* dan data keluaran berupa *file* hasil dekompresinya.
2. Metode Dekompresi, berupa *radiobutton* untuk memilih jenis dekompresi mana yang akan digunakan oleh pengguna.

3. Hasil Dekompresi, berupa sebuah tabel yang menginformasikan hasil dekompresi *file* menjadi *inverted list*.
4. Keterangan lain, yaitu berupa *textarea* yang menampilkan ukuran *file* setelah didekompresi beserta kecepatan waktu dekompresinya.




**Gambar 3.19.** Rancangan *User Interface* Sistem Dekompresi.

## 3.6 Rancangan Pengujian

Berdasarkan penelitian sebelumnya yaitu pada penelitian Chen dan Cook (2007) digunakan jumlah dokumen sebanyak 250000, *minimum support* (*minSup*) sebesar 10% , dan panjang karakter CSP-nya adalah 10 sedangkan pada penelitian ini digunakan jumlah dokumen sebanyak 1000, *minSup* sebesar 10% dan panjang karakter minimal CSP-nya adalah 2 karakter. Dikarenakan penelitian ini menggunakan 1000 dokumen, akan sangat sukar mendeteksi CSP yang panjang karakternya minimalnya 10. Oleh karena itu, penulis mengasumsikan bahwa 2 karakter *sequence* sudah merupakan pola dari CSP.

Pengujian yang akan dilakukan adalah memasukkan koleksi dokumen pada sistem kompresi. Percobaan ini dilakukan terhadap tiga jenis *file* dari *inverted list* yaitu daftar *document identifiers*, *term frequencies*, dan *term positions*. Jumlah dokumen yang diujikan

bervariasi yaitu pada interval dokumen 250, 500, 750, dan 1000. Hal ini dimaksudkan untuk mengetahui pengaruh besarnya jumlah dokumen terhadap rasio dan waktu kompresi dengan menggunakan metode *Gamma*, *Golomb*, kombinasi *Gamma* dan *Huffman* berdasarkan pendeteksian CSP, dan kombinasi *Golomb* dan *Huffman* berdasarkan pendeteksian CSP.

Pengujian jumlah CSP pada *inverted list* yang terdiri dari daftar *document identifiers*, *term frequencies*, dan *term positions*. Pengujian ini dilakukan untuk mengetahui apakah pada ketiga jenis *inverted list* dapat dikompresi dengan bantuan pendeteksian CSP (jika jumlah CSP > 0 pada *inverted list*).

Perhitungan rasio kompresi berdasarkan persamaan 2.10 dan ukuran *file* dalam satuan *kilobytes* atau KB. Rancangan pengujian kehadiran CSP set dan proses kompresinya ditunjukkan oleh tabel 3.11 dan 3.12.

**Tabel 3.11.** Rancangan Hasil Uji Coba Jumlah Pendeteksian CSP.

| Jumlah Dokumen | Jumlah CSP | | |
|---|---|---|---|
| | *Document Id* | *Term Positions* | *Term Frequencies* |
| 250 | | | |
| 500 | | | |
| 750 | | | |
| 1000 | | | |

Pada pengujian sistem dekompresinya adalah memasukkan *file* yang terkompresi kemudian dilakukan proses dekompresi menggunakan 4 metode yang digunakan pada proses kompresi. Hasil dekompresinya dianalisis berdasarkan waktu dekompresi dan jumlah *inverted list* yang benar dari hasil dekompresi yang dicocokkan dengan *inverted list* aslinya. Rancangan pengujian dekompresinya ditunjukkan oleh tabel 3.13.

**Tabel 3.12.** Rancangan Hasil Uji Coba Proses Kompresi.

| Metode Kompresi | Ukuran *File* Asli (KB) | Ukuran *File* Kompresi (KB) | Rasio Kompresi | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | | | | |
| *Golomb* | | | | |
| *Gamma* dan *Huffman*(CSP) | | | | |
| (*) *Golomb* dan *Huffman*(CSP) | | | | |

**Tabel 3.13.** Rancangan Hasil Uji Coba Proses Dekompresi.

| Metode Dekompresi | Jumlah *Inverted List* | Jumlah Kesesuaian *Inverted List* | Persentase Benar (%) | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | | | | |
| *Golomb* | | | | |
| *Gamma* dan *Huffman* (CSP) | | | | |
| (*) *Golomb* dan *Huffman* (CSP) | | | | |

Sebagai tambahan, pemilihan parameter *m* untuk kompresi dengan metode *Golomb* sangat berpengaruh terhadap hasil kompresi dari segi efisiensi penyimpanan dalam *disk* maupun terhadap kecepatan waktu kompresi. Oleh karena itu, dilakukan pemilihan *m* berdasarkan persamaan 2.6 pada subbab 2.2.1.3 yaitu $m \approx 0.69 \times mean(v)$ dimana *v* adalah daftar *integer* dari seluruh *inverted list* dan dihitung perkiraan panjang bit sebuah *integer n* (*n* dimisalkan sebagai nilai maksimal dari daftar *integer*) dengan menggunakan persamaan 2.9. Kemudian panjang bit dari parameter *m* yang menggunakan nilai dari persamaan 2.6 dibandingan dengan panjang bit parameter *m* yang menggunakan nilai statis. Nilai statis parameter *m* pada penelitian ini yang digunakan adalah 3 dan 10 (berdasarkan

tabel 2.6 subbab 2.2.1.3). Tabel perbandingannya ditunjukkan oleh tabel 3.14.

**Tabel 3.14.** Rancangan Perbandingan Pemilihan Parameter *m* dari *Golomb*.

| Jumlah Dokumen | *n* = *max* | *Mean* | *m* = 0.69 x *mean* | Panjang Bit dari *n* | | |
|---|---|---|---|---|---|---|
| | | | | *m1* = *m* | *m2* = 3 | *m3* = 10 |
| 250 | | | | | | |
| 500 | | | | | | |
| 750 | | | | | | |
| 1000 | | | | | | |

Pada tabel 3.14, nilai *n* yang dipakai adalah nilai maksimal. Jika panjang bit dari parameter *m* dengan persamaan 2.6 lebih kecil dari panjang bit dari parameter *m* dengan nilai statis 3 dan 10 dari daftar *integer*, maka parameter *m* dipilih dengan hasil perhitungan persamaan 2.6 begitu pula sebaliknya. Pemilihan parameter *Golomb* ini berdasarkan pada nilai *m* yang menghasilkan panjang bit terkecil. Hal ini disesuaikan dengan tujuan kompresi yaitu menyimpan setiap *integer* dikodekan dengan panjang bit yang terkecil (Delbru, 2010).

Pada kompresi *inverted list* baik yang menggunakan metode *Gamma* maupun *Golomb*, *unary code* yang digunakan adalah representasi *n*-1 angka 0 dan diakhiri angka 1. Dimana *n* merupakan bilangan desimal.

Sementara itu, untuk mengetahui pengaruh pemakaian teknik *filtering* (*case folding*, *stop word*, dan *stemming*) terhadap koleksi dokumen percobaan pada kompresi maka dilakukan pengujian kombinasi *Golomb* dan *Huffman* baik yang menggunakan *filtering* dokumen maupun tidak. Pengujian ini ditunjukkan oleh tabel 3.15.

**Tabel 3.15.** Rancangan Perbandingan Rasio Kompresi dan Waktu Dekompresi Berdasarkan Proses *Filtering* Dokumennya.

| No | Jumlah Dokumen | *Golomb* dan *Huffman* | *Golomb* dan *Huffman* (tanpa *filtering*) |
|---|---|---|---|
| 1 | 250 | | |
| 2 | 500 | | |
| 3 | 750 | | |
| 4 | 1000 | | |

Keterangan:

Tanda (*) merupakan metode kompresi yang digunakan pada penelitian ini.

# BAB IV
# IMPLEMENTASI DAN PEMBAHASAN

## 4.1 Lingkungan Implementasi

Lingkungan implementasi sistem kompresi *inverted list* ini terdiri dari lingkungan perangkat keras (*hardware*) dan perangkat lunak (*software*).

### 4.1.1 Lingkungan Perangkat Keras

Spesifikasi perangkat keras yang digunakan untuk implementasi sistem ini adalah:

1. *Processor* Intel Pentium® Dual-Core T4300 @ 2.10 Ghz.
2. RAM *memory* 2.00 GB DDR2.
3. *Harddisk* 320 GB.

### 4.1.2 Lingkungan Perangkat Lunak

Spesifikasi perangkat lunak yang digunakan untuk implementasi sistem ini adalah:

1. Sistem Operasi Microsoft Windows 7 Ultimate.
2. Bahasa Pemrograman Java (JDK 1.6).
3. NetBeans IDE 7.0 (IDE untuk Pemrograman Java).

## 4.2 Implementasi Program

Berdasarkan perancangan sistem kompresi dan dekompresi *inverted list* menggunakan metode *Golomb* dan *Huffman* dengan pendeteksian CSP yang dijelaskan pada bab 3, maka akan dilakukan implementasi program (*source code*) mengikuti perancangan sistem tersebut.

Langkah pertama, dimasukkannya koleksi dokumen berita dalam format *.txt sebagai data masukan yang dilanjutkan dengan proses pembacaan setiap *file* dokumen sebagai *string*. Langkah kedua, dilakukan proses *preprocessing*, *indexing*, pendeteksian CSP dalam *inverted list*, dan kompresi. *Inverted list* yang digunakan dalam sistem ini terdiri dari tiga jenis daftar yaitu *document Id*, *term positions*, dan *term frequencies*.

### 4.2.1 Implementasi *Preproccessing*

Tahap *preprocessing* menggunakan sub proses *case folding*, tokenisasi, penghapusan *stop word*, dan *stemming*.

#### 4.2.1.1 *Case Folding*

*Case folding* merupakan bagian dari *proccessing* yang difungsikan untuk mengubah seluruh huruf dalam dokumen menjadi huruf kecil (*lower case*). Sub proses *case folding* ditunjukkan oleh *source code* 4.1. Fungsi *case folding* pada *source code* 4.1 ditunjukkan oleh baris 2.

```
1  private void caseFolding(String docs) {
2          this.docs = docs.toLowerCase();
3  }
```

*Source Code* 4.1 *Case Folding*.

#### 4.2.1.2 Tokenisasi

Tokenisasi berperan untuk mengubah hasil pembacaan dokumen menjadi pecahan kata-kata atau disebut juga *array* kata. Sub proses tokenisasi ditunjukkan oleh *source code* 4.2. Pada *source code* 4.2 terdapat *delimiter* pada baris ke 2, yang merupakan sebuah variabel yang dipakai sebagai *filter* karakter dalam fungsi *split*() yang ada pada baris ke 3. Fungsi *split*() sendiri digunakan untuk memotong *string* sebuah dokumen dipisahkan berdasarkan spasi sehingga menjadi *array* kata.

```
1  private void tokenizer(String docs) {
2          String delimiter = "[^a-z^A-Z]+";
3          terms = docs.split(delimiter);
4
5          Object[] tmp_terms = terms.clone();
6          ArrayList termlist = new ArrayList
7                                  (Arrays.asList(tmp_terms));
8          termlist.removeAll(Arrays.asList("","-"));
9          terms = termlist.toArray();
10 }
```

*Source Code* 4.2 Tokenisasi.

#### 4.2.1.3 Penghapusan *Stop Word*

Penghapusan *stop word* atau *stop word removal* merupakan sub proses yang berfungsi menghilangkan kata-kata yang tidak begitu penting seperti kata "yang", "di", "pada", dan sebagainya. Langkah pertamanya adalah memasukkan daftar kata *stop word* (*stop list*) dari *file*. Kemudian dibandingkan *array* kata dari hasil tokenisasi, jika terdapat kata yang ada pada daftar *stop word* maka kata tersebut akan dihapus dari *array* kata. Sub proses penghapusan *stop word* ditunjukkan oleh *source code* 4.3.

```
1  private void stopwordsRemoval() {
2          Object[] tmp_terms = terms.clone();
3          ArrayList termlist = new ArrayList
4                               (Arrays.asList(tmp_terms));
5          termlist.removeAll(Arrays.asList(""));
6          termlist.removeAll(Arrays.asList(stop_list));
7          terms = termlist.toArray();
8  }
```

*Source Code* 4.3 Penghapusan *Stop Word*.

#### 4.2.1.4 *Stemming*

*Stemming* adalah bagian *preproccesing* yang berfungsi untuk mengubah kata pada daftar kata menjadi kata dasar. Pada *source code* 4.4 ditunjukkan hasil implementasi *stemming* berdasarkan subbab 2.1.4 dan 3.3.1.3. Langkah pertamanya adalah penghapusan *inflection suffixes* yang ditunjukkan oleh *source code* 4.4 pada baris ke 2 dan secara lebih detil ditunjukkan pada *source code* 4.5. Langkah keduanya, menghilangkan *derivation suffix* yang ditunjukkan oleh *source code* 4.4 pada baris ke 3 dan secara lebih detil ditunjukkan pada *source code* 4.6. Langkah ketiganya, menghilangkan *derivation prefix* yang ditunjukkan oleh *source code* 4.4 pada baris ke 4 dan secara lebih detil ditunjukkan oleh *source code* 4.7.

```
1  private void Stemming() {
2          inflectionalSuffixRemoval();
3          derivationSuffixRemoval();
4          derivationPrefixRemoval();
5
6          if (!is_prefix_removed && !isExistRootWord()) {
7              derivationSuffixConditional();
8              if (rootDictionarySearch(kata_tmp))
9                kata_dasar = kata_tmp;
10             else {
```

```
                derivationPrefixRemoval();
                if (!is_prefix_removed) {
                    derivationSuffixConditional();
                    derivationPrefixRemoval();
                }
            }

            if (kata_dasar.equals("")) kata_dasar =
                kata_asli;
        }
}
```

*Source Code* 4.4 Implementasi *Stemming*.

```
private void inflectionalSuffixRemoval() {
        String in_sufiks;
        length = kata_tmp.length();

        if (kata_tmp.endsWith("ku") ||
            kata_tmp.endsWith("mu")) {
            in_sufiks = kata_tmp.substring(length-
                2,length);
            kata_tmp = kata_asli.substring(0, length-
                in_sufiks.length());
            inflection_sufiks += in_sufiks;
        }
        else if (kata_tmp.endsWith("nya")) {
            in_sufiks = kata_tmp.substring(length-
                3,length);
            kata_tmp = kata_asli.substring(0, length-
                in_sufiks.length());
            inflection_sufiks += in_sufiks;
        }
        else if (kata_tmp.endsWith("lah") ||
                 kata_tmp.endsWith("kah") ||
                 kata_tmp.endsWith("tah") ||
                 kata_tmp.endsWith("pun")) {
            in_sufiks = kata_tmp.substring(length-
                3,length);
            kata_tmp = kata_asli.substring(0, length-
                in_sufiks.length());
            inflection_sufiks += in_sufiks;
            inflectionalSuffixRemoval(); // rekursif
        }
}
```

*Source Code* 4.5 Penghapusan *Inflection Suffixes*.

```
private void derivationSuffixRemoval() {
        length = kata_tmp.length();
        if (kata_tmp.endsWith("i")) {
            kata_tmp = kata_tmp.substring(0,length-1);
            sufiks = "i";
        }
        else if (kata_tmp.endsWith("an")) {
```

```
8               kata_tmp = kata_tmp.substring(0, length-2);
9               sufiks = "an";
10          }
11
12          if(rootDictionarySearch(kata_tmp))
13              kata_dasar = kata_tmp;
14      }
```

*Source Code* 4.6 Penghapusan *Derivation Suffix*.

```
1   private void derivationPrefixRemoval() {
2           String der_prefiks;
3           String kata_jenis_prefiks = "";
4           length = kata_tmp.length();
5           try {
6               der_prefiks = kata_tmp.substring(0,2);
7           } catch (Exception e) {
8               der_prefiks = kata_tmp.substring(0,1);
9           }
10          if (combinationDisallowed(der_prefiks, sufiks))
11          {
12              is_prefix_removed = false;
13              return;
14          }
15          prefiks.add(der_prefiks);
16          int current = prefiks.size()-1;
17          String cur_prefix = prefiks.get(current)
18                                     .toString();
19          if (prefiks.size() > 1) {
20              String prev_prefix = prefiks
21                      .get(current-1).toString();
22              if (cur_prefix.equals(prev_prefix) ||
23                      (prefiks.size() > 3) ||
24                  (prev_prefix.equals("me") &&
25                          cur_prefix.equals("be"))) {
26                  is_prefix_removed = false;
27                  return;
28              }
29          }
30          jenis_prefiks = typeOfPrefix(cur_prefix);
31          if (jenis_prefiks.equals("")) {
32              prefiks.remove(current);
33              is_prefix_removed = false;
34              return;
35          }
36          else {
37              kata_tmp = prefixConditionRemoval
38                          (jenis_prefiks, kata_tmp);
39              if (jenis_prefiks.contains("luluh")) {
40                  if (jenis_prefiks.matches("ter-
41                          luluh|ber-luluh"))
42                      kata_tmp = "r" + kata_tmp;
43                  else if (jenis_prefiks.matches("men-
44                          luluh|pen-luluh"))
45                      kata_tmp = "t" + kata_tmp;
```

```
46                  else if (jenis_prefiks.matches("meng-
47                          luluh|peng-luluh"))
48                      kata_tmp = "k" + kata_tmp;
49                  else if (jenis_prefiks.matches("meny-
50                          luluh|peny-luluh"))
51                      kata_tmp = "s" + kata_tmp;
52                  else if (jenis_prefiks.matches("mem-
53                          luluh|pem-luluh"))
54                      kata_tmp = "p" + kata_tmp;
55
56                  kata_jenis_prefiks = jenis_prefiks
57                  .substring(0,jenis_prefiks
58                          .indexOf("-"));
59              }
60          }
61          if(rootDictionarySearch(kata_tmp)) {
62              is_prefix_removed = true;
63              kata_dasar = kata_tmp;
64              return;
65          }
66          else {
67              if (jenis_prefiks.contains("luluh")) {
68                  kata_depan = kata_tmp.substring(0,1);
69                  kata_tmp = kata_tmp.substring(1,
70                          kata_tmp.length());
71                      if (!rootDictionarySearch(kata_tmp))
72                      {
73                          if (kar_akhir_prefiks
74                                  .matches("m|n"))
75                              kata_tmp = kar_akhir_prefiks
76                              + kata_tmp;
77                          else
78                              kata_tmp =
79                              kata_jenis_prefiks + kata_tmp;
80                      }
81              }
82
83              if(!rootDictionarySearch(kata_tmp)) {
84                  derivationPrefixRemoval();
85                  if (kata_dasar.equals("")) {
86                      kata_tmp =  kata_tmp+
87                                  inflection_sufiks;
88                      if(rootDictionarySearch(kata_tmp))
89                          kata_dasar = kata_tmp;
90                  }
91              }
92              else kata_dasar = kata_tmp;
93          }
94  }
```

*Source Code* 4.7 Penghapusan *Derivation Prefix*.

### 4.2.2 Implementasi *Indexing*

*Indexing* merupakan proses pembentukkan *inverted index* yang terdiri dari *lexicon* dan *inverted list*. Proses *indexing* ditunjukkan oleh *source code* 4.8. Penjelasan *source code* 4.8, langkah-langkahnya adalah:

1. Langkah pertama, menentukan panjang *index* berdasarkan jumlah *term* keseluruhan dari koleksi dokumen.
2. Langkah kedua, membentuk *lexicon* yaitu dengan menyimpan setiap *term* ke-*i* yang ditunjukkan pada baris ke 5.
3. Langkah ketiga, membentuk *inverted list* yaitu dengan menyimpan posisi kemunculan *term* ke-*i* pada dokumen ke-*j* kemudian menyimpannya sebagai *document Id*.
4. Langkah keempat, menyimpan setiap posisi *term* ke-*i* pada dokumen ke-*j* sebagai *term positions* yang ditunjukkan pada baris ke 27-37.
5. Langkah kelima, menghitung frekuensi *term* yang ditunjukkan pada baris ke 40 yang disimpan menjadi *term frequencies*.
6. Langkah keenam, menghitung *d-gap* dari *document Id* ditunjukkan pada baris ke 41 sedangkan untuk menghitung *d-gap* dari *term positions* ditunjukkan pada baris ke 42.

```
1     public void indexProccessing() {
2             int r,s; // variabel untuk iterasi
3             for (int i = 0; i < terms.length; i++) {
4                 index[i] = new inverted_index();
5                 index[i].lexicon = terms[i].toString();
6                 index[i].total_frequency = 0;
7                 index[i].total_documents = 0;
8                 index[i].document_list = new
9                                       ArrayList<Integer>();
10                index[i].occurrence_lists =
11                  new HashMap<Integer,ArrayList<Integer>>();
12
13                for (int j = 0; j < document_terms.length;
14                    j++) {
15                    ArrayList occur = new ArrayList();
16
17                    if (document_terms[j].contains
18                       (terms[i])) {
19                        // simpan dokumen list
20                        index[i].document_list.add(j+1);
21                        index[i].total_documents++;
22                        int first_pos = document_terms[j]
23                                              .indexOf(terms[i]);
24                        int last_pos = document_terms[j]
25                           .lastIndexOf(terms[i]);
26
27                    for (int k = first_pos; k < last_pos+1;
```

```
                    k++) {
                    if (terms[i].equals(
                        document_terms[j].get(k))) {
                            occur.add(k+1);
                            index[i].occurrence_lists
                                    .put(j+1, occur);
                            index[i].total_frequency++;
                        }
                    }
                }
            }

            index[i].retrievalFrequencyList();
            index[i].calculateDocumentGap();
            index[i].calculateOccurrenceGap();
            index[i].setPrint();
        }
}
```

*Source Code* 4.8 *Indexing*.

### 4.2.3 Implementasi Pendeteksian CSP *Mining*

Pendeteksian CSP adalah proses untuk menemukan pola (*pattern*) yang terdapat pada *inverted list*, proses ini dijelaskan pada subbab 2.3.3. Pendeteksian CSP dibagi menjadi dua tahapan yaitu tahap pertama, pendeteksian CSP awal menggunakan *Apriori*. Tahap kedua adalah efisiensi CSP pada tahap pertama menggunakan struktur data *UpDown Tree*.

#### 4.2.3.1 *Apriori*

Implementasi pada algoritma *Apriori* berdasarkan subbab 2.3.1. Proses utama dari algoritma ini adalah *findingLargeSequences*(), yang berfungsi untuk mencari seluruh *sequence pattern* berdasarkan kombinasi aturan dari *Apriori*. Proses ini ditunjukkan oleh *source code* 4.9.

```
public void findingLargeSequences() {
        ArrayList sequence; // sequence penampung
        ArrayList support; // support penampung
        // candidate 1-sequence
        findingFrequentItemset();
        sequence = candidate_frequent_itemset;
        support = getCountingSupport(sequence);
        candidate_sequence.put(1, new
                sequence_properties(sequence, support));
        // large 1-sequence
        sequence = sequence_functions
                .getSequence(candidate_sequence, 1);
```

```
13          support = sequence_functions
14                  .getSupport(candidate_sequence, 1);
15          thresholdSupport(sequence, support);
16          large_sequence.put(1, new
17              sequence_properties(current_seq,
18                      current_sup));
19          // membentuk large (k-1)-sequence
20          int k = 2;
21          ArrayList last_sequence = sequence_functions
22                    .getSequence(large_sequence, 1);
23          while (!last_sequence.isEmpty()) {
24              // candidate k-sequence
25              sequence = joinStep(last_sequence);
26              sequence = pruneStep(sequence, last_sequence);
27              support = getCountingSupport(sequence);
28              candidate_sequence.put(k, new
29                      sequence_properties(sequence,
30                       support));
31
32              // large k-sequence
33              sequence = sequence_functions
34                            .getSequence(candidate_sequence,
35                              k);
36              support = sequence_functions
37                            .getSupport(candidate_sequence,
38                              k);
39              thresholdSupport(sequence, support);
40              large_sequence.put(k, new
41                sequence_properties(current_seq,
42                      current_sup));
43              // update nilai last_sequence
44              last_sequence = sequence_functions
45                            .getSequence(large_sequence, k);
46              k++;
47          }
48          // remove empty large dan candidate pada index
49          candidate_sequence.remove(candidate_sequence
50               .size());
51          large_sequence.remove(large_sequence.size());
52  }
```

*Source Code* 4.9 Pendeteksian CSP Awal dengan *Apriori*.

#### 4.2.3.2 *UpDown Tree*

Implementasi pada algoritma *UpDown Tree* berdasarkan subbab 2.3.4. Proses ini ditunjukkan oleh *source code* 4.10. Penjelasan *source code* 4.10, langkah-langkahnya adalah:

1. Langkah pertama, mencari seluruh *leaf node* yang terdapat pada *Up Tree* yang ditunjukkan pada baris 5-6.
2. Langkah kedua, mencari seluruh *leaf node* yang terdapat pada *Down Tree* yang ditunjukkan pada baris 7-8.

3. Langkah ketiga, menggabungkan setiap *leaf node* pada *Up Tree* sampai ke *root*-nya yang ditunjukkan pada baris 14-22.
4. Langkah keempat, menggabungkan *root* pada *Down Tree* sampai ke *leaf node*-nya yang ditunjukkan pada baris 24-44.
5. Langkah kelima atau tahap akhirnya, menggabungkan hasil dari *Up Tree* dan *Down Tree* menjadi CSP set dari *UpDown Tree* yang ditunjukkan pada baris 46-97.

```
1   private void detectingAllPatternMining() {
2           int size = up_tree.size();
3
4           for (int i = 0; i < size; i++) {
5               ArrayList<trienode> up_leaf = up_tree
6                               .get(i).searchLeafNodes();
7               ArrayList<trienode> down_leaf = down_tree
8                               .get(i).searchLeafNodes();
9               ArrayList<trienode> parent_leaf_set = new
10                               ArrayList<trienode>();
11              ArrayList<trienode> csp_leaf_set = new
12                               ArrayList<trienode>();
13
14              trienode j_leaf;
15              for (int j = 0; j < up_leaf.size(); j++) {
16                  j_leaf = upTreeFromLeafToRoot
17                          (up_tree.get(i),  up_leaf
18                          .get(j));
19                  if (j_leaf != null) {
20                      parent_leaf_set.add(j_leaf);
21                  }
22              }
23
24              trienode k_leaf;
25              for (int k = 0; k < down_leaf.size(); k++) {
26                  k_leaf = downTreeFromRootToLeaf
27                  (down_tree.get(i), down_leaf.get(k),
28                   minSup);
29                  trienode root = down_tree.get(i)
30                          .getRoot();
31
32                  if (k_leaf != null) {
33                      if (k_leaf.getIdSets().size()
34                          >= minSup) {
35                          if (root.getIdSets().isEmpty()) {
36                           k_leaf.setValue(root
37                          .getInfo() + k_leaf.getInfo(), 0,
38                                  k_leaf.getIdSets(),
39                                  k_leaf.getParent());
40                          }
41                          csp_leaf_set.add(k_leaf);
42                      }
43                  }
44              }
```

```
45
46              for (int j = 0; j < parent_leaf_set.size();
47                   j++) {
48                   String str_parent = parent_leaf_set
49                           .get(j).getInfo();
50                   ArrayList set_id_parent = parent_leaf_set
51                           .get(j).getIdSets();
52                    int level_parent =  parent_leaf_set
53                          .get(j).getLevel();
54                  // looping csp_leaf_set
55                 for (int k = 0; k < csp_leaf_set.size();
56                    k++) {
57                       String str_csp = csp_leaf_set.get(k)
58                                       .getInfo();
59                       ArrayList set_id_csp = csp_leaf_set
60                                       .get(k).getIdSets();
61                       int level_csp = csp_leaf_set.get(k)
62                                       .getLevel();
63
64                       if (isSubsetOnIdSet(set_id_parent,
65                         set_id_csp) || isSubsetOnIdSet
66                         (set_id_csp, set_id_parent)) {
67                          String csp;
68                          String up_root_info = up_tree
69                                .get(i).getRoot().getInfo();
70                          String down_root_info = up_tree
71                                .get(i).getRoot().getInfo();
72
73                          if ((str_parent.endsWith
74                          (up_root_info) &&
75                          str_csp.startsWith
76                          (down_root_info))
77                          && (level_parent ==
78                                level_csp))
79                                {
80                          str_csp = str_csp.substring
81                                (up_root_info.length(),
82                                      str_csp.length());
83                                csp = str_parent + str_csp;
84                          }
85                          else
86                                csp = str_parent + str_csp;
87
88                          Object[] sequence_of_csp =
89                                      sequence_functions
90                                    .sequenceSplit(csp);
91                          if (sequence_of_csp.length >=
92                              minLengthChar)
93                              insertToCSPSet(csp);
94                       }
95                   }
96               }
97           }
98
99       Object[] objArray = CSP_set.toArray();
```

```
100             String[] strArray = new String[objArray.length];
101             for (int l = 0; l < objArray.length; l++)
102                 strArray[l] = objArray[l].toString();
103             new sort_operations<String>()
104                 .orderByLength(strArray);
105             CSP_set.clear();
106             CSP_set.addAll(Arrays.asList(strArray));
107     }
```

*Source Code* 4.10 Efisiensi CSP dengan *UpDown Tree*.

### 4.2.4 Implementasi Kompresi

Setelah ditemukan CSP maka selanjutnya dilakukan transformasi yang dijelaskan pada subbab 2.4, hasil transformasi tersebut kemudian dikompresi dengan *Huffman*. Langkah-langkah kompresi adalah:

1. Pada kompresi *Huffman* dilakukan proses pembuatan tabel *Huffman* yang dijelaskan pada *source code* 4.11. Pada *source code* 4.11 terdapat variabel *result_tree* merupakan hasil akhir dari *Huffman Tree* yang ditunjukkan pada baris ke 10. Variabel *result_tree* dihasilkan oleh fungsi *createDataListArray* (dijelaskan pada *source code* 4.12).
2. Setiap karakter atau *sequence* dari CSP dikodekan dengan *Huffman* yang dijelaskan pada *source code* 4.13 sedangkan untuk angka-angka *integer* selain karakter yang terdapat pada CSP set dikompresi dengan *Gamma* atau *Golomb* yang dijelaskan pada *source code* 4.14 untuk *Gamma* dan *source code* 4.15 untuk *Golomb*.

```
1   public void construct() {
2           int size = character_list.size();
3
4           // sorting by value
5           ArrayList character_keys = new ArrayList
6                     (character_list.keySet());
7           ArrayList frequency_values = new ArrayList
8                     (character_list.values());
9
10          result_tree = createDataListArray
11                (character_keys.toArray(),
12                 frequency_values.toArray(),   size);
13
14          if (result_tree != null) {
15             // membentuk tabel huffman
16             for (Iterator it =character_keys.iterator();
17                  it.hasNext();) {
```

```
                String str = (String) it.next();
                String encode = result_tree
                        .encode(result_tree.root, str);
                huffman_table.put(String.valueOf(str),
                                String.valueOf(encode));
            }
        }
}
```

*Source Code* 4.11 Konstruksi Tabel *Huffman*.

```
private huffman_tree createDataListArray(Object[]
character,
     Object[] frequency_values, int size) {
        huffman_tree tmp_tree;
        data_list_array = new huffman_data[size];
        // mengisi index data_list_array pertama
        data_list_array[0] = new huffman_data();
        data_list_array[0].setKey(character);
        data_list_array[0].setValue(frequency_values);
        data_list_array[0] = greedyAlgorithm
        (data_list_array[0],0);

        int k = 1;
        while (k < size) {
            data_list_array[k] = greedyAlgorithm
                            (data_list_array[k-1], k);
            k++;
        }

        huffman_tree ending_tree = data_list_array
                    [data_list_array.length-1].getTree();
        for (int i = data_list_array.length-1;
                    i > 0; i--) {
            if (data_list_array[i].getTree() != null) {
                if (i < data_list_array.length-1) {
                    tmp_tree = data_list_array[i]
                                .getTree();
                    ending_tree.root = ending_tree
                                .cloneChild
                            (ending_tree.root,
                             tmp_tree.root);
                }
            }
        }
        return ending_tree;
}
```

*Source Code* 4.12 Implementasi Pembuatan *Huffman Tree*.

```
public String HuffmanEncode(String str) {
        String codeword = "";
        // looping
        for (Iterator it = huffman_table.entrySet()
                            .iterator(); it.hasNext();) {
```

```
            Map.Entry<String, String> entry =
                    (Map.Entry)
                        it.next();
            String character = entry.getKey();
            String code = entry.getValue();
            // pengecekan karakter dengan tabel huffman
            if (character.equals(str)) {
                codeword = code;
                break; // berhenti dari looping
            }
        }
        return codeword;
}
```

*Source Code* 4.13 *Huffman Encoding*.

```
public static String GammaEncode(int x) {
        if (x == 0) return "0";

        int log2_x = math_libraries.floor(
                    math_libraries.log2(x) );
        int u = 1 + log2_x;
        String unary = unary_code.encode(u, prefix);
        int m = x - math_libraries.power(2, log2_x);
        int length_bit = log2_x;

        String binary_str;
        if (m > -1 && m < 256)
            binary_str = math_libraries
                        .convertToBiner8bit(m);
        else
            binary_str = math_libraries.toBinary(m);

        int len = binary_str.length();
        try {
            binary_str = binary_str.substring
                        (len-length_bit, len);
        } catch (Exception e) {
            binary_str =

         file_operations.repeatChar(String.valueOf("0"),
              length_bit-len) + binary_str;
        }

        String encode_str = unary.concat(binary_str);
        return encode_str;
}
```

*Source Code* 4.14 *Gamma Encoding*.

```
public static String GolombEncode(int n, int m) {
        int q = math_libraries.floor( n / m );
        int r = n - (q * m);
        int c = math_libraries.ceil(math_libraries
                .log2(m));
```

```
6           int d = math_libraries.power(2, c) - m;
7           String unary = unary_code.encode(q+1, prefix);
8           String[] remainder = new String[m];
9           int b = d-1; // nilai terakhir dari d remainder
10          int bz = 0;
11          boolean first = true;
12          for (int i = 0; i < m; i++) {
13              String biner = "";
14              if (m != math_libraries.power(2, c) ) {
15                  if (i < d) {
16                      biner = math_libraries.toBinary(i);
17                      biner = addPrefixZero(biner, c-1);
18                  }
19                  else {
20                      if (first) {
21                        b = b + 1; // jumlahkan nilai b
22                        biner = math_libraries.toBinary(b);
23                        biner = addPrefixZero(biner, c-1) +
24                                "0";
25                        bz = math_libraries.toDecimal(biner);
26                        first = false;
27                      }
28                      else {
29                        bz++;
30                        biner = math_libraries
31                                .toBinary(bz);
32                        biner = addPrefixZero(biner, c);
33                      }
34                  }
35              }
36              else biner = math_libraries.toBinary(i);
37              remainder[i] = biner;
38          }
39          String binary_str = remainder[r];
40          String encode_str = unary.concat(binary_str);
41          return encode_str;
42  }
```

*Source Code* 4.15 *Golomb Encoding*.

### 4.2.5 Implementasi Dekompresi

Proses *decoding* terbagi menjadi dua bagian sub proses. Sub proses pertama adalah mendekodekan CSP menggunakan tabel *Huffman* yang tersimpan dalam *file*. Sub proses kedua adalah mendekodekan setiap *inverted list* selain dari karakter CSP dengan menggunakan *Gamma* atau *Golomb*. Proses *Huffman decoding*-nya ditunjukkan oleh *source code* 4.16 sedangkan *Gamma* dan *Golomb decoding*-nya ditunjukkan oleh *source code* 4.17 dan 4.18.

```
1   public static String HuffmanDecode(String str_code,
2   HashMap<String, String> save_huffman_table) {
```

```
3           String word = "";
4           // looping
5           for (Iterator it = save_huffman_table.entrySet()
6                   .iterator(); it.hasNext();) {
7               Map.Entry<String, String> entry = (Map.Entry)
8                                   it.next();
9               String character = entry.getKey();
10              String code = entry.getValue();
11              // pengecekan karakter dengan karakter huffman
12              if (code.equals(str_code)) {
13                  word = character;
14                  break; // berhenti dari looping
15              }
16          }
17          return word;
18  }
```

*Source Code* 4.16 *Huffman Decoding*.

```
1   public static int GammaDecode(String encode_str) {
2           int m = 0; // jumlah karakter 0 atau 1
3
4           if (encode_str.equals("0")) return 0;
5
6           // membaca jumlah bit 0
7           int i = 0;
8           while(i < encode_str.length()) {
9               if (encode_str.charAt(i) == '0')
10                  m++;
11              else break;
12              i++;
13          }
14
15          String bit_str = encode_str;
16          if (m < encode_str.length()) {
17              try {
18                  bit_str = encode_str.substring(m,
19                          encode_str.length());
20              } catch (Exception e) {
21                  System.out.println("error gamma");
22              }
23          }
24
25          int decode_int =
26  math_libraries.toDecimal(bit_str);
27          return decode_int;
28  }
```

*Source Code* 4.17 *Gamma Decoding*.

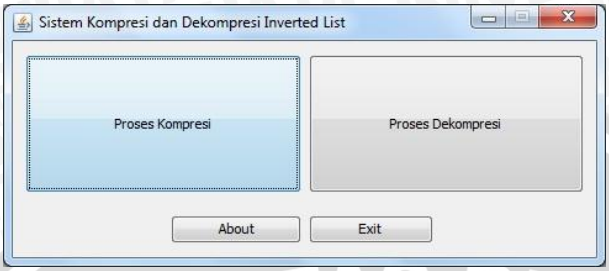
```
1   public static int GolombDecode(String encode_str, int m) {
2           int q = 0;
3
4           if (encode_str.equals("0")) return 0;
5
```

```
6           // membaca jumlah bit 0
7           int i = 0;
8           while(i < encode_str.length()) {
9               if (encode_str.charAt(i) == '0')
10                  q++;
11              else break;
12              i++;
13          }
14
15          int c = math_libraries.ceil
16                  ( math_libraries.log2(m) );
17          int d = math_libraries.power(2, c) - m;
18
19          String bit_str = "";
20          try {
21              bit_str = encode_str.substring(q+1,
22                          encode_str.length());
23          } catch (Exception e) {
24              System.out.println("error :" +
25                e.getMessage());
26          }
27
28          int r = math_libraries.toDecimal(bit_str);
29          if (r >= d) r = r - d;
30
31          int decode_n = (q * m) + r;
32          return decode_n;
33  }
```

*Source Code* 4.18 *Golomb Decoding*.

## 4.3 Implementasi Antar Muka

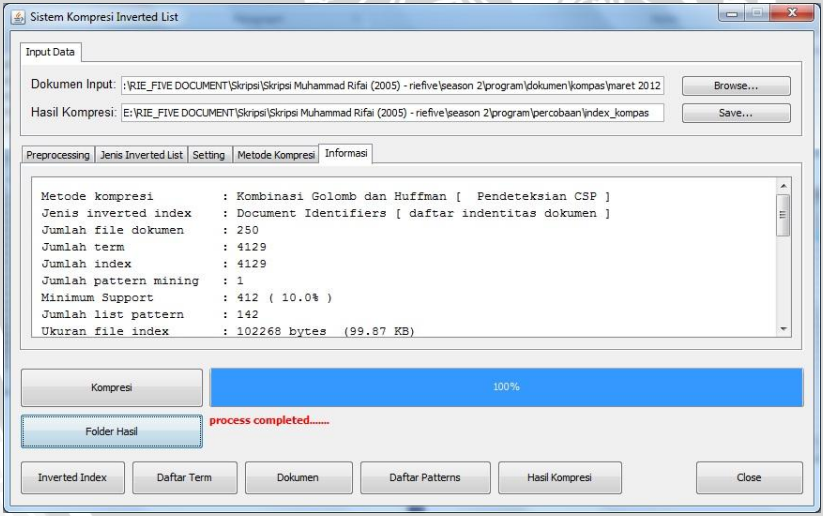
Berdasarkan implementasi *source code* pada subbab 4.2, maka implementasi antar muka terbagi dalam beberapa *form* yaitu *form* utama, *form* kompresi dan *form* dekompresi. Pada *form* utamanya terdapat empat buah *button* atau tombol, dimana dua buah tombol dengan ukuran besar merupakan pilihan untuk proses kompresi dan dekompresi sedangkan dua buah tombol kecil berfungsi sebagai tambahan yaitu tombol "*about*" dan "*exit*". Antar muka untuk *form* utama ditunjukkan oleh gambar 4.1.




**Gambar 4.1.** Tampilan Antar Muka *Form* Utama.

### 4.3.1 Halaman Kompresi

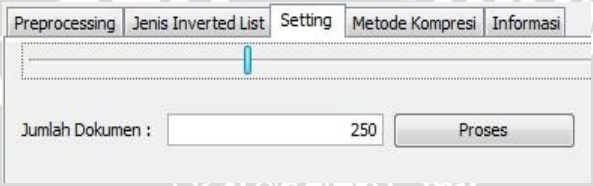
*Form* kompresi merupakan antar muka yang difungsikan untuk kompresi pada *inverted list* dengan berberapa pengaturan yang dikehendaki oleh *user*. *Form* ini dibagi dalam empat bagian yaitu bagian pertama yaitu *input* data, bagian kedua yaitu pengaturan, bagian ketiga yaitu proses kompresi, dan bagian keempat yaitu informasi hasil kompresi. *Form* ini ditunjukkan oleh gambar 4.2.



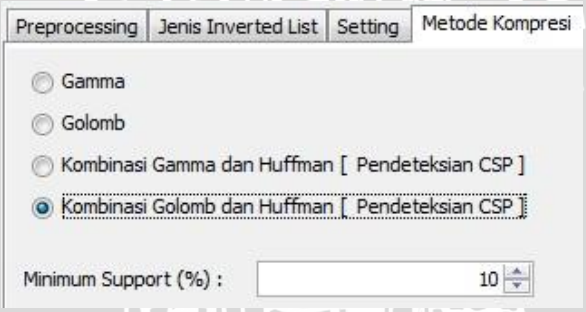

**Gambar 4.2.** Tampilan Antar Muka *Form* Kompresi.

Pada bagian pengaturan berfungsi untuk menentukan jenis *inverted list*, jumlah dokumen digunakan, dan metode kompresi beserta pengaturan jumlah *minimum support*. Gambar untuk setiap pengaturan berturut-turut ditunjukkan oleh gambar 4.3, 4.4, dan 4.5.


**Gambar 4.3.** Pengaturan Jenis *Inverted List*.

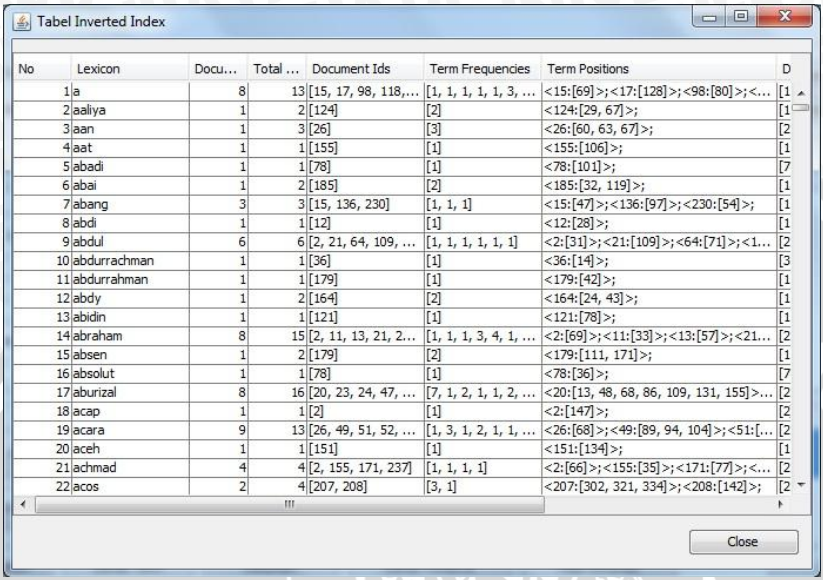



**Gambar 4.4.** Pengaturan Jumlah Dokumen.

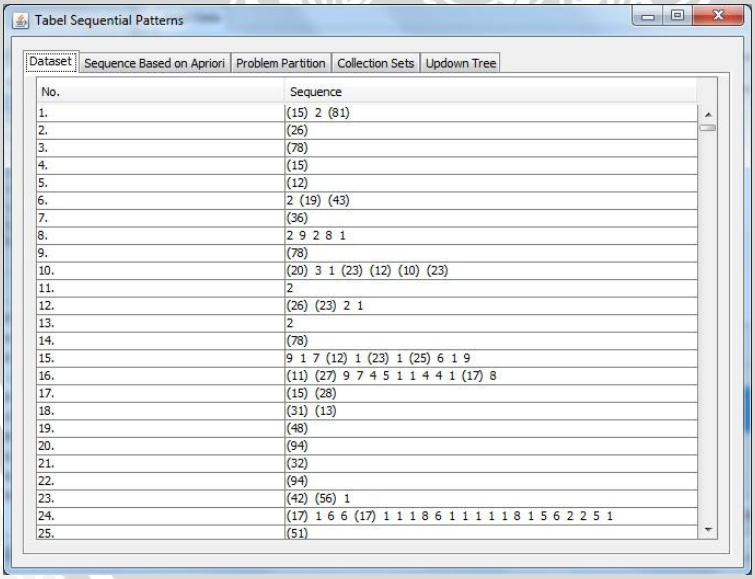



**Gambar 4.5.** Pengaturan Metode Kompresi.

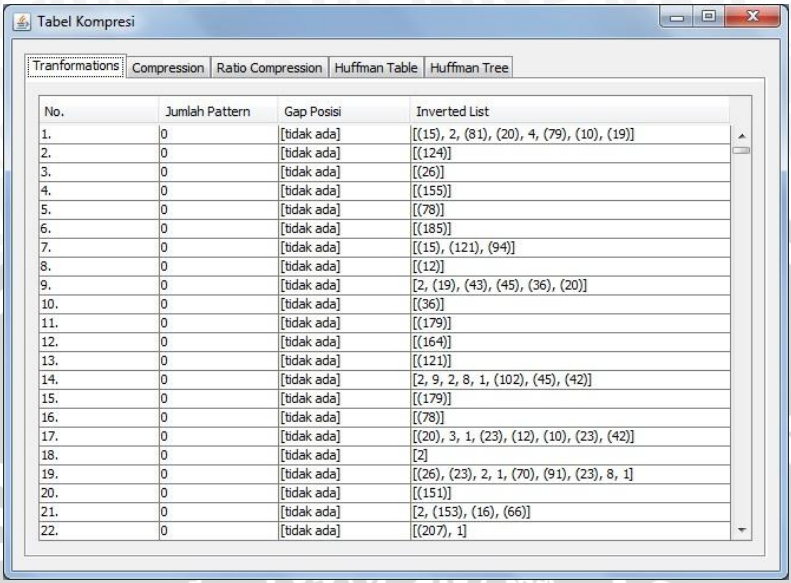
Pada bagian informasi hasil kompresi yang terdiri dari tombol-tombol untuk menampilkan informasi hasil *indexing* yang berupa daftar *inverted index*, daftar hasil *sequential pattern*, dan hasil kompresi. Gambar untuk setiap tampilan informasi berturut-turut ditunjukkan oleh gambar 4.6, 4.7, dan 4.8.




**Gambar 4.6.** Daftar Seluruh *Inverted Index*.



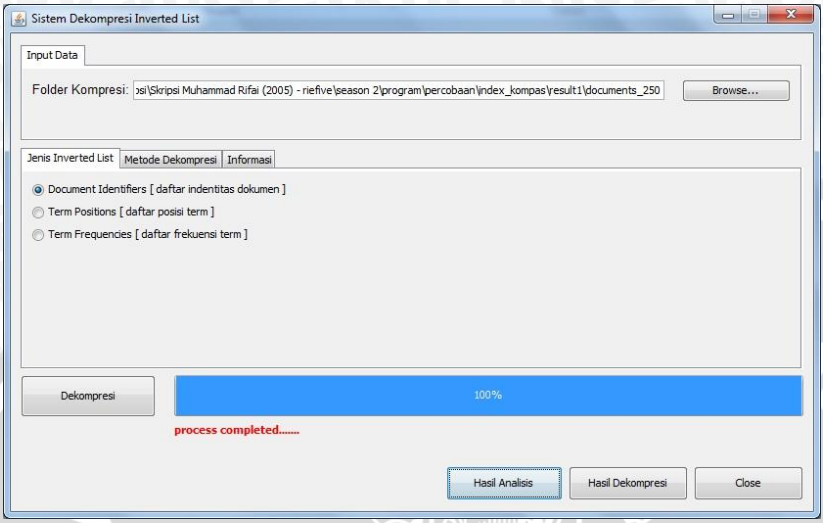

**Gambar 4.7.** Daftar Hasil *Sequential Pattern*.




**Gambar 4.8.** Hasil Kompresi.

### 4.3.2 Halaman Dekompresi

*Form* dekompresi berfungsi untuk mendekodekan *file* dari *inverted list* yang telah terkompresi. Langkah pertama, memasukkan lokasi *folder* dari *file* hasil kompresi. Langkah kedua, mendekodekan *file* hasil kompresi dengan menggunakan pengaturan metode dan jenis *file* dari *inverted list* yang dipilih oleh *user*. *Form* ini ditunjukkan oleh gambar 4.9.




**Gambar 4.9.** Tampilan Antar Muka *Form* Dekompresi.

## 4.4 Hasil Evaluasi

Berdasarkan pembahasan subbab 3.1, sumber data yang digunakan menggunakan koleksi dokumen berformat *.txt yang berjumlah sebanyak 1000 dokumen. Pengujian dari 1000 dokumen tersebut dilakukan secara bertahap, yaitu pada jumlah dokumen 250, 500, 750, dan 1000.

Pada penelitian ini digunakan empat buah metode kompresi yakni *Gamma*, *Golomb*, kombinasi *Gamma* dan *Huffman* berdasarkan pendeteksian CSP, serta kombinasi *Golomb* dan *Huffman* berdasarkan pendeteksian CSP.

Pada rancangan pengujian subbab 3.6, persentase *minimum support* yang digunakan untuk pendeteksian CSP sebesar 10% dan panjang karakter minimal dari CSP sebesar 2 karakter.

### 4.4.1 Hasil Pendeteksian CSP

Pengujian ini dilakukan terhadap koleksi dokumen yang berjumlah 250, 500, 750, dan 1000. Jumlah *inverted list* dari *document Id*, *term frequencies*, dan *term positions* ditunjukkan oleh tabel 4.1 sedangkan hasil pendeteksian CSP ditunjukkan oleh tabel 4.2.

**Tabel 4.1.** Jumlah *Inverted List.*

| Jumlah Dokumen | Jumlah *Inverted List* | | |
|---|---|---|---|
| | *Document Id* | *Term Positions* | *Term Frequencies* |
| 250 | 4129 | 21437 | 4129 |
| 500 | 5566 | 42619 | 5566 |
| 750 | 6540 | 64203 | 6540 |
| 1000 | 7353 | 85229 | 7353 |

**Tabel 4.2.** Hasil Pendeteksian CSP.

| Jumlah Dokumen | Jumlah CSP | | |
|---|---|---|---|
| | *Document Id* | *Term Positions* | *Term Frequencies* |
| 250 | 1 | 0 | 3 |
| 500 | 2 | 0 | 6 |
| 750 | 3 | 0 | 7 |
| 1000 | 4 | 0 | 7 |

Diketahui dari tabel 4.2 bahwa pada baris pertama atau jumlah dokumen 250 yaitu jumlah CSP untuk *document Id* bernilai 1, *term positions* sama dengan 0, dan *term frequencies* berjumlah 3.

Pada jumlah dokumen 250, 500, 750, dan 1000 *inverted list* untuk daftar *term positions* pada sama sekali tidak ditemukan CSP (jumlah CSP-nya = 0). Sebaliknya daftar *document Id* dan *term frequencies* ditemukan CSP (jumlah CSP-nya > 0), bahkan setiap penambahan jumlah dokumen pada kedua *inverted list* ini cenderung bertambah jumlah CSP-nya. Oleh karena itu, dapat diketahui bahwa *inverted list* jenis daftar *term positions* tidak dapat dikompresi dengan menggunakan metode kombinasi *Golomb* dan *Huffman* berdasarkan pendeteksian CSP dikarenakan pada *inverted list* tersebut tidak ditemukan pola CSP.

### 4.4.2 Hasil Pemilihan Parameter *m* dari *Golomb Code*

Berdasarkan subbab 3.6, pemilihan parameter *m* untuk *Golomb* menentukan tingkat efisiensi ruang penyimpanan dan waktu kompresi metode *Golomb*. Oleh karena itu, digunakan persamaan 2.6 pada subbab 2.2.1.3 untuk perhitungan parameter *m*. Hasil percobaan

pemilihan parameter *m* untuk *document Id* ditunjukkan oleh tabel 4.3 dan untuk *term frequencies* ditunjukkan oleh tabel 4.4.

**Tabel 4.3.** Perbandingan Pemilihan Parameter *m* dari *Golomb* untuk *Document Id*.

| Jumlah Dokumen | *n* = *max* | *Mean* | *m* = 0.69 x *mean* | Panjang Bit dari *n* | | |
|---|---|---|---|---|---|---|
| | | | | *m1* = *m* | *m2* = 3 | *m3* = 10 |
| 250 | 249 | 31 | 21 | 16 | 85 | 28 |
| 500 | 500 | 42 | 29 | 22 | 168 | 54 |
| 750 | 750 | 50 | 35 | 27 | 252 | 79 |
| 1000 | 1000 | 58 | 40 | 30 | 335 | 104 |

Pada tabel 4.3 untuk jumlah dokumen 250 didapatkan *m* = 21, dihitung dengan persamaan 2.6 yaitu 0.69 x 31 = 21.39 dan dibulatkan menjadi 21 (21.39 ≈ 21). Begitu pula untuk jumlah dokumen 500, 750, dan 1000 dilakukan perhitungan yang sama. Diketahui pada tabel 4.3 bahwa panjang bit untuk jumlah dokumen 250, *m1* < *m2* dan *m1* < *m3* yaitu 16 < 85 dan 16 < 28 sehingga *m1* adalah parameter *Golomb* yang dipakai pada dokumen 250. Begitu juga pada jumlah dokumen 500, 750, dan 1000 panjang bit *m1* lebih kecil dari *m2* maupun *m3* sehingga *m1* yang dipilih sebagai parameter *Golomb* dari jumlah dokumen-dokumen percobaan ini.

**Tabel 4.4.** Perbandingan Pemilihan Parameter *m* dari *Golomb* untuk *Term Frequencies*.

| Jumlah Dokumen | *n* = *max* | *Mean* | *m* = 0.69 x *mean* | Panjang Bit dari *n* | | |
|---|---|---|---|---|---|---|
| | | | | *m1* = *m* | *m2* = 3 | *m3* = 10 |
| 250 | 30 | 2 | 1 | 31 | 12 | 7 |
| 500 | 76 | 2 | 1 | 77 | 27 | 11 |
| 750 | 76 | 2 | 1 | 77 | 27 | 11 |
| 1000 | 76 | 2 | 1 | 77 | 27 | 11 |

Pada tabel 4.4, hasil *m* dari *inverted list* jenis *term frequencies* pada jumlah dokumen 250, 500, 750, dan 1000 adalah sebesar 1, dengan menggunakan persamaan 2.6 parameter *m* dapat dihitung 0.69 x 1 = 0.69 dan dibulatkan menjadi 1 atau menghasilkan *m* = 1. Diketahui pada tabel 4.4 bahwa panjang bit yang terkecil untuk setiap dokumen 250, 500, 750, dan 1000 masing-masing adalah 7, 11, 11, dan 11 dimana parameter yang digunakan adalah *m3* yaitu 10. Sehingga pada *term frequencies*, parameter *Golomb* yang dipakai adalah nilai statis 10 bukan hasil pada persamaan 2.6.

Untuk perhitungan perkiraan panjang bit dari *Golomb* pada tabel 4.3 dan 4.4 menggunakan persamaan 2.9.

### 4.4.3 Hasil Kompresi *Inverted List*

Berdasarkan pengujian pada subbab 4.4.1 diketahui bahwa hanya dua jenis *inverted list* saja yang dapat dikompresi dengan hasil pendeteksian CSP-nya yaitu daftar *document Id* dan *term frequencies*. Sehingga hanya daftar *document Id* dan *term frequencies* saja yang diujikan pada percobaan kompresi ini sedangkan daftar *term positions* tidak diujikan.

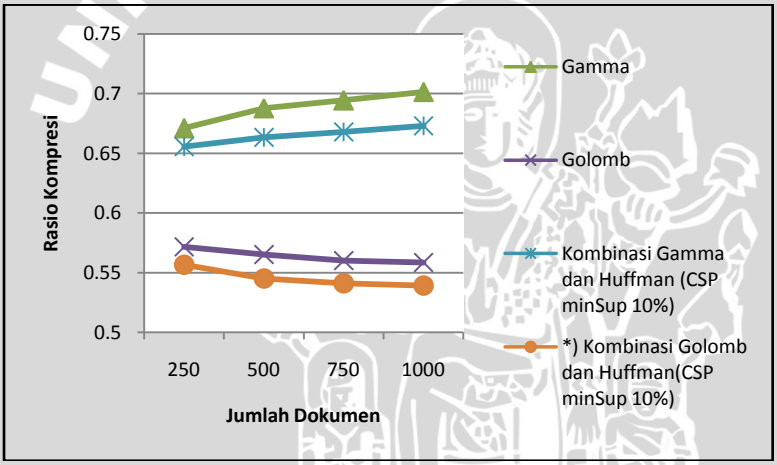
#### 4.4.3.1 Kompresi untuk *Document Id*

Berdasarkan rancangan pengujian pada subbab 3.6, hasil percobaan rasio kompresi untuk jenis *inverted list* yaitu daftar *document Id* untuk 4 metode yang digunakan pada penelitian ini dengan jumlah dokumen 250, 500, 750, dan 1000 ditunjukkan oleh tabel 4.5 dan gambar 4.10 (hasil percobaan kompresi untuk jumlah dokumen 250, 500, 750, dan 1000 secara detil disertakan pada lampiran 1).

Perhitungan rasio kompresi berdasarkan persamaan 2.10 pada subbab 2.2.2. Penggunaan parameter *m* untuk kompresi *inverted list* jenis *document id* yang menggunakan metode kompresi *Golomb* berdasarkan tabel 4.3.

**Tabel 4.5.** Rasio Kompresi Pada *Document Id*.

| No. | Jumlah Dokumen | Metode Kompresi | | | |
|---|---|---|---|---|---|
| | | *Gamma* | *Golomb* | *Gamma* dan *Huffman* (CSP) | *Golomb* dan *Huffman* (CSP) |
| 1 | 250 | 0.671 | 0.5717 | 0.6556 | 0.5567 |
| 2 | 500 | 0.6878 | 0.5653 | 0.6634 | 0.5451 |
| 3 | 750 | 0.6943 | 0.5602 | 0.668 | 0.5411 |
| 4 | 1000 | 0.7014 | 0.5585 | 0.6731 | 0.5393 |
| **Rata-rata** | | **0.6886** | **0.5639** | **0.6650** | **0.5455** |



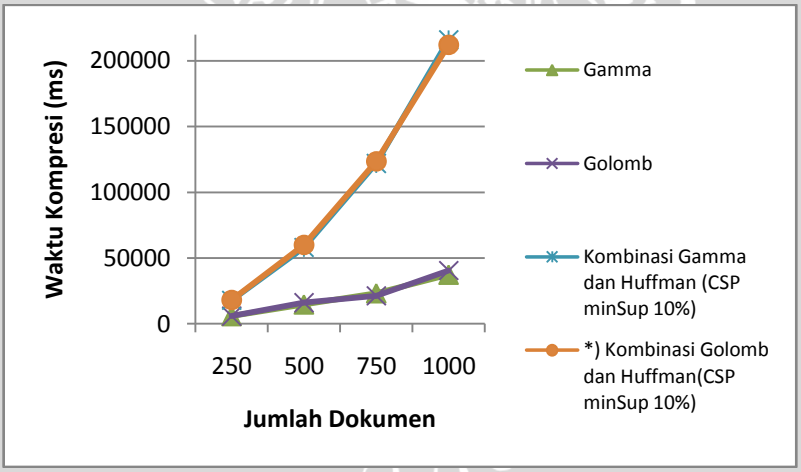

**Gambar 4.10.** Grafik Rasio Kompresi Pada *Document Id*.

Berdasarkan tabel 4.5 dan gambar 4.10 diketahui bahwa perbandingan rasio kompresi untuk daftar *document Id* dari empat metode menunjukkan bahwa metode kombinasi *Golomb* dan *Huffman* dengan CSP merupakan rasio terkecil dengan nilai rasio rata-rata sebesar 0.5455 atau *file* kompresi hanya mengambil 54.55% ukuran *file* dari data asli sedangkan metode *Gamma* merupakan rasio terbesar dengan nilai rasio rata-rata sebesar 0.6886 atau memakai 68.86% ukuran *file* dari data asli. Dengan menggunakan persamaan 2.11 untuk presentase perbandingan rasio diketahui bahwa metode kombinasi *Golomb* dan *Huffman* untuk rasio kompresinya lebih baik

sebesar 20.77% dari *Gamma*, 3.25% dari *Golomb*, dan 17.96% dari kombinasi *Gamma* dan *Huffman*.

Perbandingan jumlah penggunaan dokumen percobaan berdasarkan tabel 4.5 dan gambar 4.10 menunjukkan bahwa setiap penambahan jumlah dokumen, nilai rasio kompresi pada metode kombinasi *Golomb* dan *Huffman* untuk daftar *document Id* semakin berkurang, meskipun penurunannya tidak signifikan. Sehingga dapat dikatakan bahwa kompresi *inverted list* untuk daftar *document Id* dengan penggunaan jumlah dokumen percobaan tertentu tidak terlalu mempengaruhi hasil kompresi. Jika nilai rasio semakin berkurang pada pada jumlah dokumen tertentu maka hasil kompresi semakin menjadi baik.

Gambar 4.11 merupakan grafik hasil perbandingan waktu kompresi yang dibutuhkan untuk daftar *document Id* diujikan pada jumlah dokumen mulai dari 250, 500, 750, dan 1000 dokumen.




**Gambar 4.11.** Grafik Perbandingan Waktu Kompresi Pada *Document Id*.

Berdasarkan gambar 4.11 diketahui bahwa waktu kompresi untuk daftar *document Id* dengan menggunakan metode pendeteksian CSP lebih lama dari pada metode kompresi tunggal. Hal ini disebabkan metode kompresi dengan bantuan pendeteksian CSP harus

melakukan proses pencarian dan efisiensi pola CSP dalam *inverted list* sehingga waktu yang dibutuhkan lebih lama.

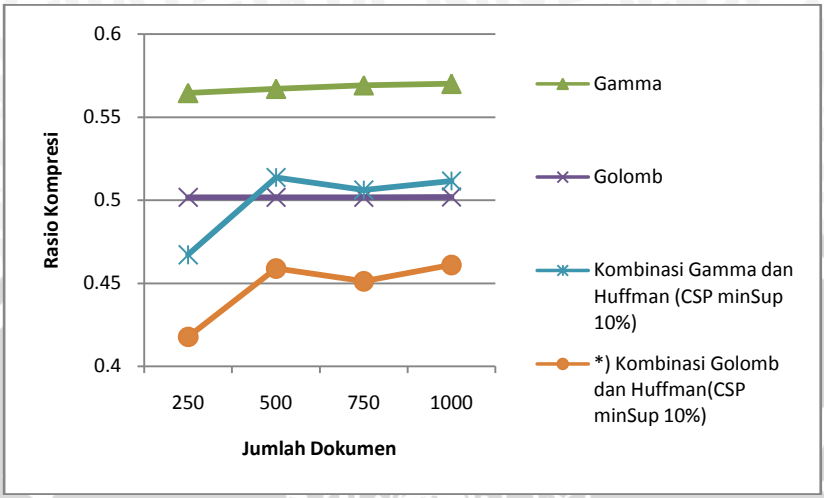
#### 4.4.3.2 Kompresi untuk *Term Frequencies*

Berdasarkan rancangan pengujian pada subbab 3.6, hasil percobaan rasio kompresi untuk jenis *inverted list* yaitu daftar *term frequencies* untuk 4 metode yang digunakan pada penelitian ini dengan jumlah dokumen 250, 500, 750, dan 1000 ditunjukkan oleh tabel 4.6 dan gambar 4.12 (hasil percobaan kompresi untuk jumlah dokumen 250, 500, 750, dan 1000 secara detil disertakan pada lampiran 2).

Perhitungan rasio kompresi berdasarkan persamaan 2.10 pada subbab 2.2.2. Penggunaan parameter *m* untuk kompresi *inverted list* jenis *term frequencies* yang menggunakan metode kompresi *Golomb* berdasarkan tabel 4.4.

**Tabel 4.6.** Rasio Kompresi Pada *Term Frequencies*.

| No. | Jumlah Dokumen | Metode Kompresi | | | |
|---|---|---|---|---|---|
| | | *Gamma* | *Golomb* | *Gamma* dan *Huffman* (CSP) | *Golomb* dan *Huffman* (CSP) |
| 1 | 250 | 0.5645 | 0.5018 | 0.4672 | 0.4178 |
| 2 | 500 | 0.5671 | 0.5018 | 0.5137 | 0.4589 |
| 3 | 750 | 0.5691 | 0.5018 | 0.5062 | 0.4512 |
| 4 | 1000 | 0.57 | 0.5019 | 0.5116 | 0.461 |
| **Rata-rata** | | **0.5676** | **0.5018** | **0.4996** | **0.4472** |

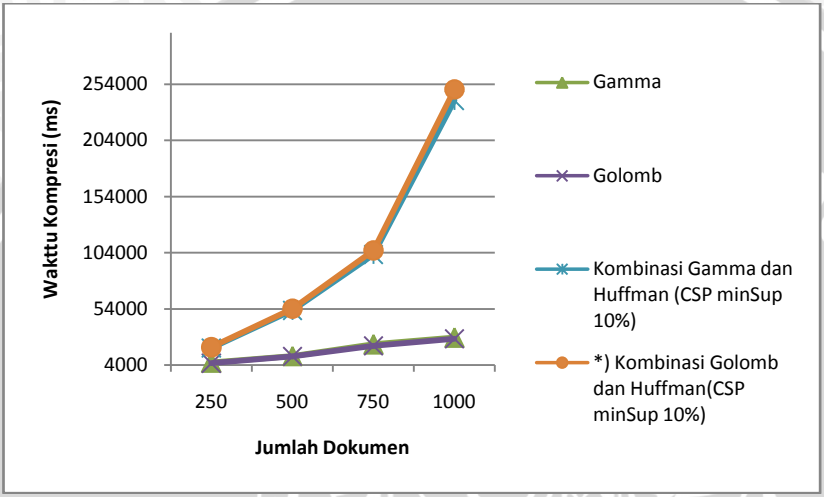



**Gambar 4.12.** Grafik Rasio Kompresi Pada *Term Frequencies*.

Berdasarkan tabel 4.6 dan gambar 4.12 diketahui bahwa perbandingan rasio kompresi untuk daftar *term frequencies* dari empat metode menunjukkan, metode kombinasi *Golomb* dan *Huffman* dengan CSP merupakan rasio terkecil dengan nilai rasio rata-rata sebesar 0.4472 atau *file* kompresi hanya mengambil 44.72% ukuran *file* dari data asli sedangkan metode *Gamma* merupakan rasio terbesar dengan nilai rasio rata-rata sebesar 0.5676 atau memakai 56.76% ukuran *file* dari data asli. Dengan menggunakan persamaan 2.11 untuk presentase perbandingan rasio diketahui bahwa metode kombinasi *Golomb* dan *Huffman* untuk rasio kompresinya lebih baik sebesar 21.21% dari *Gamma*, 10.88% dari *Golomb*, dan 10.49% dari kombinasi *Gamma* dan *Huffman*.

Perbandingan jumlah penggunaan dokumen percobaan berdasarkan tabel 4.6 dan gambar 4.12 menunjukkan bahwa setiap penambahan jumlah dokumen, nilai rasio kompresi pada metode kombinasi *Golomb* dan *Huffman* untuk daftar *term frequencies* tidak teratur, pada jumlah dokumen 250 rasio kompresinya cukup kecil yaitu sebesar 0.4178. Sehingga dapat dikatakan bahwa kompresi *inverted list* untuk daftar *term frequencies* dengan penggunaan jumlah dokumen percobaan tertentu cukup mempengaruhi hasil kompresi. Jika nilai rasio semakin berkurang pada pada jumlah dokumen tertentu maka hasil kompresi semakin menjadi baik.

Gambar 4.13 merupakan grafik hasil perbandingan waktu kompresi yang dibutuhkan untuk daftar *term frequencies* diujikan pada jumlah dokumen mulai dari 250, 500, 750, dan 1000 dokumen.




**Gambar 4.13.** Grafik Perbandingan Waktu Kompresi Pada *Term Frequencies*.

Berdasarkan gambar 4.13 diketahui bahwa waktu kompresi untuk daftar *term frequencies* dengan menggunakan metode pendeteksian CSP lebih lama dari pada metode kompresi tunggal. Hal ini disebabkan metode kompresi dengan bantuan pendeteksian CSP harus melakukan proses pencarian dan efisiensi pola CSP dalam *inverted list* sehingga waktu yang dibutuhkan lebih lama.

### 4.4.4 Hasil Dekompresi

#### 4.4.3.1 Dekompresi untuk *Document Id*

Berdasarkan rancangan pengujian pada subbab 3.6, hasil percobaan waktu dekompresi untuk jenis *inverted list* yaitu daftar *document Id* untuk 4 metode yang digunakan pada penelitian ini dengan jumlah dokumen 250, 500, 750, dan 1000 ditunjukkan oleh tabel 4.7 (hasil percobaan dekompresi untuk jumlah dokumen 250, 500, 750, dan 1000 secara detil disertakan pada lampiran 3). Penggunaan parameter *m* untuk dekompresi *file inverted list* jenis

*document id* yang menggunakan metode dekompresi *Golomb* berdasarkan tabel 4.3.

**Tabel 4.7.** Waktu Dekompresi Pada *Document Id*.

| No. | Jumlah Dokumen | Waktu Dekompresi (ms) | | | |
|---|---|---|---|---|---|
| | | *Gamma* | *Golomb* | *Gamma* dan *Huffman* (CSP) | *Golomb* dan *Huffman* (CSP) |
| 1 | 250 | 1607 | 1591 | 1622 | 1685 |
| 2 | 500 | 1966 | 1981 | 2309 | 2153 |
| 3 | 750 | 2371 | 2356 | 2714 | 2636 |
| 4 | 1000 | 2824 | 2776 | 2839 | 3136 |
| **Rata-rata** | | **2192** | **2176** | **2371** | **2402.5** |

Hasil pengujian dari tabel 4.7, dapat diketahui bahwa waktu dekompresi yang digunakan metode kombinasi *Golomb* dan *Huffman* rata-rata sebesar 2402.5 ms dan merupakan waktu terlama untuk dekompresi dibandingkan tiga metode lainnya. Selisih waktu rata-rata dekompresi metode kombinasi *Golomb* dan *Huffman* lebih lama dari *Gamma* adalah 210.5 ms, dari *Golomb* adalah 226.5 ms, dan dari kombinasi *Gamma* dan *Huffman* adalah 31.5 ms. Dengan kata lain bahwa waktu dekompresi 4 metode tersebut mempunyai selisih rata-rata waktu dekompresi untuk *document Id* kurang dari 1000 ms atau 1 detik.

#### 4.4.3.2 Dekompresi untuk *Term Frequencies*

Berdasarkan rancangan pengujian pada subbab 3.6, hasil percobaan waktu dekompresi untuk jenis *inverted list* yaitu daftar *term frequencies* untuk 4 metode yang digunakan pada penelitian ini dengan jumlah dokumen 250, 500, 750, dan 1000 ditunjukkan oleh tabel 4.8 (hasil percobaan dekompresi untuk jumlah dokumen 250, 500, 750, dan 1000 secara detil disertakan pada lampiran 4). Penggunaan parameter *m* untuk dekompresi *file inverted list* jenis *term frequencies* yang menggunakan metode dekompresi *Golomb* berdasarkan tabel 4.4.

**Tabel 4.8.** Waktu Dekompresi Pada *Term Frequencies.*

| No. | Jumlah Dokumen | Waktu Dekompresi (ms) | | | |
|---|---|---|---|---|---|
| | | *Gamma* | *Golomb* | *Gamma* dan *Huffman* (CSP) | *Golomb* dan *Huffman* (CSP) |
| 1 | 250 | 1576 | 1592 | 1653 | 1623 |
| 2 | 500 | 1990 | 2021 | 1982 | 2059 |
| 3 | 750 | 2445 | 2458 | 2480 | 2491 |
| 4 | 1000 | 2980 | 2792 | 3401 | 3073 |
| **Rata-rata** | | **2247.75** | **2215.75** | **2379** | **2311.5** |

Hasil pengujian dari tabel 4.8, dapat diketahui bahwa waktu dekompresi yang digunakan metode kombinasi *Golomb* dan *Huffman* rata-rata sebesar 2311.5 ms dan merupakan waktu tercepat ketiga untuk dekompresi dari 4 metode yang digunakan. Selisih waktu rata-rata dekompresi metode kombinasi *Golomb* dan *Huffman* lebih lama dari *Gamma* adalah 64 ms, dari *Golomb* adalah 95.75 ms, dan lebih cepat dari kombinasi *Gamma* dan *Huffman* adalah 67.5 ms. Dengan kata lain bahwa waktu dekompresi 4 metode tersebut mempunyai selisih rata-rata waktu dekompresi yang cukup kecil untuk *term frequencies* yaitu kurang dari 100 ms. Bahkan waktu dekompresi metode kombinasi *Golomb* dan *Huffman* lebih cepat dibandingkan kombinasi *Gamma* dan *Huffman* pada jenis *inverted list* yaitu daftar *term frequencies*.

### 4.4.5 Hasil Perbandingan Kompresi Kombinasi *Golomb* dan *Huffman* Berdasarkan Teknik *Filtering* Dokumen

#### 4.4.5.1 Perbandingan Rasio Kompresi dan Waktu Dekompresi *Document Id*

Berdasarkan rancangan pengujian pada subbab 3.6, hasil percobaan rasio kompresi dan waktu dekompresi untuk jenis *inverted list* yaitu daftar *document Id* untuk metode kombinasi *Golomb* dan *Huffman* dengan *filtering* dokumen (*case folding*, *stop word*, dan *stemming*) dibandingkan tanpa menggunakan *filter* dengan jumlah dokumen 250, 500, 750, dan 1000 ditunjukkan oleh tabel 4.9 dan 4.10 (hasil percobaan ini disertakan pada lampiran 6).

**Tabel 4.9.** Perbandingan Rasio Kompresi dengan *Filter* dan Tanpa *Filter* Pada *Document Id.*

| No. | Jumlah Dokumen | *Golomb* dan *Huffman* | *Golomb* dan *Huffman* (tanpa *filter*) |
|---|---|---|---|
| 1 | 250 | 0.5567 | 0.5426 |
| 2 | 500 | 0.5451 | 0.5356 |
| 3 | 750 | 0.5411 | 0.5297 |
| 4 | 1000 | 0.5393 | 0.5269 |
| **Rata-Rata** | | **0.5456** | **0.5337** |

**Tabel 4.10.** Perbandingan Waktu Dekompresi dengan *Filter* dan Tanpa *Filter* Pada *Document Id.*

| No. | Jumlah Dokumen | *Golomb* dan *Huffman* | *Golomb* dan *Huffman* (tanpa *filter*) |
|---|---|---|---|
| 1 | 250 | 1685 | 2389 |
| 2 | 500 | 2153 | 3453 |
| 3 | 750 | 2636 | 4357 |
| 4 | 1000 | 3136 | 5804 |
| **Rata-Rata** | | **2402.5** | **4000.75** |

Berdasarkan hasil tabel 4.9 dan 4.10 diketahui bahwa rasio kompresi untuk *document id* pada metode kombinasi *Golomb* dan *Huffman* tanpa *filter* hanya sedikit lebih baik dari metode yang menggunakan *filter*. Hal ini disebabkan pada metode yang tidak menggunakan *filter* ditemukan jumlah CSP yang lebih banyak dari metode yang menggunakan *filter* (CSP untuk hasil percobaan kombinasi *Golomb* dan *Huffman* tanpa *filter* disertakan pada lampiran 6). Namun pada metode tanpa *filter* menpunyai kelemahan yaitu waktu dekompresi jauh lebih lama serta ukuran *file* untuk *inverted list* juga lebih besar daripada metode yang menggunakan *filter*.

#### 4.4.5.2 Perbandingan Rasio Kompresi dan Waktu Dekompresi *Term Frequencies*

Berdasarkan rancangan pengujian pada subbab 3.6, hasil percobaan rasio kompresi dan waktu dekompresi untuk jenis *inverted list* yaitu daftar *term frequencies* untuk metode kombinasi *Golomb* dan *Huffman* dengan *filtering* dokumen (*case folding*, *stop word*, dan *stemming*) dibandingkan tanpa menggunakan *filter* dengan jumlah dokumen 250, 500, 750, dan 1000 ditunjukkan oleh tabel 4.11 dan 4.12 (hasil percobaan ini disertakan pada lampiran 6).

**Tabel 4.11.** Perbandingan Rasio Kompresi dengan *Filter* dan Tanpa *Filter* Pada *Term Frequencies.*

| No. | Jumlah Dokumen | *Golomb* dan *Huffman* | *Golomb* dan *Huffman* (tanpa *filter*) |
|---|---|---|---|
| 1 | 250 | 0.4178 | 0.4179 |
| 2 | 500 | 0.4589 | 0.4572 |
| 3 | 750 | 0.4512 | 0.4518 |
| 4 | 1000 | 0.461 | 0.4523 |
| **Rata-Rata** | | **0.4472** | **0.4448** |

**Tabel 4.12.** Perbandingan Waktu Dekompresi dengan *Filter* dan Tanpa *Filter* Pada *Term Frequencies.*

| No. | Jumlah Dokumen | *Golomb* dan *Huffman* | *Golomb* dan *Huffman* (tanpa *filter*) |
|---|---|---|---|
| 1 | 250 | 1623 | 2107 |
| 2 | 500 | 2059 | 2879 |
| 3 | 750 | 2491 | 3729 |
| 4 | 1000 | 3073 | 4568 |
| **Rata-Rata** | | **2311.5** | **3320.75** |

Berdasarkan hasil tabel 4.11 dan 4.12 diketahui bahwa rasio kompresi dan waktu dekompresi untuk *term frequencies* pada metode kombinasi *Golomb* dan *Huffman* tanpa *filter* sedikit lebih buruk dari metode yang menggunakan *filter*.

## 4.5 Analisis Hasil Pengujian

Berdasarkan hasil pengujian kompresi terhadap tiga jenis *inverted list* dapat diketahui bahwa *term positions* tidak dapat dikompresi dengan menggunakan metode kombinasi *Golomb* dan *Huffman* berdasarkan pendeteksian CSP. Hal ini disebabkan pada *term positions* tidak ditemukan pola CSP sedangkan untuk *document Id* dan frekuensi *term* dapat dikompresi dengan metode kombinasi *Golomb* dan *Huffman* berdasarkan pendeteksian CSP.

Pemilihan parameter *m* untuk *Golomb* yang digunakan pada penelitian ini untuk jumlah dokumen 250, 500, 750, dan 1000 berdasarkan tabel 4.3 dan 4.4 dapat diketahui bahwa nilai *m* untuk *document id* secara bervariasi dihitung menggunakan persamaan 2.6 sedangkan untuk *term frequencies* menggunakan nilai statis yakni nilai *m* = 10. Pemilihan parameter ini bergantung dari rata-rata panjang bit yang dihasilkan dari persamaan 2.9, pada tiga nilai *m* (*m1*, *m2*, dan *m3*) yang diujikan dipilih nilai *m* yang memiliki panjang bit terkecil sehingga menghasilkan hasil kompresi yang lebih maksimal.

Kompresi *inverted list* berupa *document Id* berdasarkan hasil tabel 4.5 diketahui bahwa metode kombinasi *Golomb* dan *Huffman* memiliki rasio kompresi yang lebih baik daripada metode *Gamma*, *Golomb*, dan kombinasi *Gamma* dan *Huffman*. Pada kompresi *inverted list* yang berupa daftar *document Id* menggunakan kombinasi *Golomb* dan *Huffman*, rata-rata rasio kompresinya sebesar 0.5455. Sementara itu, kompresi *inverted list* berupa *term frequencies* berdasarkan hasil tabel 4.6 diketahui bahwa metode kombinasi *Golomb* dan *Huffman* memiliki rasio kompresi yang lebih baik daripada metode *Gamma*, *Golomb*, dan kombinasi *Gamma* dan *Huffman*. Pada kompresi *inverted list* yang berupa daftar *term frequencies* menggunakan kombinasi *Golomb* dan *Huffman*, rata-rata rasio kompresinya sebesar 0.4472.

Hasil kompresi *inverted list* baik pada *document Id* maupun *term frequencies* menunjukkan bahwa penentuan jumlah dokumen (250, 500, 750, dan 1000) sebagai data masukkan sistem kompresi ini, tidak mempengaruhi nilai rasio kompresi secara signifikan hal ini dapat dilihat dari gambar 4.10 dan 4.12.

Hasil pengujian untuk perbandingan waktu dekompresi diketahui bahwa daftar *document Id* dan *term frequencies* menggunakan

metode *Gamma*, *Golomb*, kombinasi *Gamma* dan *Huffman* dengan CSP, dan kombinasi *Golomb* dan *Huffman* dengan CSP menghasilkan hasil dekompresi untuk *inverted list* yang terkompresi sama dengan *inverted list* yang sebelum dikompresi. Parameter keberhasilan dekompresinya diukur dari presentase kesesuaian jumlah *inverted list* yakni menghasilkan nilai sebesar 100% atau data yang terdekompresi sama persis dengan data sebelum dikompresi. Untuk perbandingan waktu, metode *Gamma* merupakan metode yang tercepat waktu dekompresinya dibandingkan tiga metode lainnya. Tetapi selisih perbandingan waktu dekompresi antara metode *Gamma* dengan metode lainnya tidak sampai 1 detik. Waktu dekompresi *Golomb* dan *Huffman* terlama baik untuk *document Id* maupun *frequency term* adalah sebesar 2402.5 ms dan 2311.5 ms atau sekitar 2 detik lebih.

Untuk metode dekompresi menggunakan pendeteksian CSP (kombinasi *Gamma* dan *Huffman* atau kombinasi *Golomb* dan *Huffman*) membutuhkan rata-rata waktu yang lebih lama daripada metode kompresi tunggal (*Gamma* atau *Golomb*) disebabkan karena dilakukannya proses detransformasi CSP pada saat proses dekompresi (berdasarkan subbab 2.4).

Penggunaan *filter* dokumen seperti *case folding*, *stop word*, dan *stemming* pada metode kombinasi *Golomb* dan *Huffman* secara umum mengurangi ukuran *file* dari *inverted list* dikarenakan jumlah *index*-nya jauh lebih kecil. Pada rasio kompresi yang dihasilkan hanya sedikit lebih baik bahkan hampir mempunyai nilai yang sama dari metode tanpa menggunakan *filter*. Akan tetapi waktu dekompresinya jauh lebih baik metode yang menggunakan *filter* dibandingkan tanpa menggunakan *filter*.

# BAB V
# KESIMPULAN DAN SARAN

## 5.1 Kesimpulan

Berdasarkan penelitian yang telah dilakukan maka dapat disimpulkan beberapa hal, antara lain:

1. Metode kompresi menggunakan kombinasi *Golomb* dan *Huffman* berdasarkan pendeteksian CSP telah berhasil diimplementasikan untuk kompresi *inverted list*. Langkah-langkah untuk pengimplementasian sistem kompresi ini secara umum adalah:
    a. Pembentukan *inverted list* melalui proses *indexing.*
    b. Pendeteksian *Contiguous Sequential Pattern* (CSP) pada *inverted list*.
    c. Tranformasi Kemunculan CSP pada *inverted list.*
    d. Kompresi *inverted list* yang telah ditransformasi. Jika *inverted list* berupa karakter CSP maka dikodekan dengan *Huffman* sebaliknya jika *inverted list* hanya berupa daftar *integer* maka dikodekan dengan *Golomb*.
2. Hasil evaluasi menunjukkan bahwa rasio kompresi untuk *document Id* dan *term frequencies* menggunakan kombinasi *Golomb* dan *Huffman* lebih baik daripada tiga metode lainnya yaitu metode *Gamma*, *Golomb*, dan kombinasi *Gamma* dan *Huffman*. Sementara itu, untuk perbandingan waktu dekompresinya diperlukan waktu yang lebih lama dari tiga metode lainnya yakni dengan waktu sekitar 2402.5 ms dan 2311.5 ms untuk *document Id* dan *term frequencies*. Namun waktu dekompresinya dibandingkan dengan tiga metode lainnya hanya terpaut tidak lebih dari 1 detik. Sehingga kombinasi *Golomb* dan *Huffman* berdasarkan pendeteksian CSP cukup baik digunakan untuk kompresi *inverted list.*
3. Penggunaan *filter* dokumen seperti *case folding, stop word*, dan *stemming* pada metode kombinasi *Golomb* dan *Huffman* berdasarkan CSP tidak terlalu mempengaruhi nilai rasio kompresi tetapi berpengaruh pada kecepatan waktu dekompresinya dibandingkan metode kombinasi *Golomb* dan *Huffman* berdasarkan CSP tanpa menggunakan *filter*.

## 5.2 Saran

Saran untuk penelitian lanjutan adalah sebagai berikut.

1. Digunakan jumlah dokumen untuk penelitian yang lebih besar misalkan 5000, 10000, atau 100000 dokumen.
2. Pada penelitian ini tidak diperhitungan pemilihan tema atau topik dari dokumen. Sebaiknya perlu diujikan penggunaan satu tema atau topik berita untuk koleksi dokumen misalkan topik politik, olahraga, ekonomi, teknologi, dan sebagainya.
3. Dilakukan evaluasi atau pengujian *query* terhadap *inverted list* dibandingkan antara *inverted list* yang dikompresi dan yang tidak dikompresi.
4. Untuk penelitian selanjutnya perlu dilakukan pengujian terhadap jumlah *minSup* lebih kecil dan lebih besar dari 10% untuk pendeteksian CSP-nya. Hal ini dilakukan untuk mengetahui pengaruh jumlah kehadiran CSP terhadap hasil kompresi.

# DAFTAR PUSTAKA


Agrawal, R. dan Srikant, R. 1994. *Fast Algorithms for Mining Association Rules*. IBM Almaden Research Center. Proceedings of the 20th VLDB Conference Santiago, Chile.

Agrawal, R. dan Srikant, R. 1995. *Mining Sequential Patterns*. Department of Computer Science, University of Wisconsin, Madison.

Agusta, Ledy. 2009. *Perbandingan Algoritma Stemming Porter dengan Algoritma Nazief & Adriani untuk Stemming Dokumen Teks Bahasa Indonesia*. Konferensi Nasional Sistem dan Informatika, Bali.

Asian, J., Williams, H. E. dan Tahaghoghi, S.M.M. 2005. *Stemming Indonesian*. School of Computer Science and Information Technology, RMIT University, Melbourne, Australia.

Chen, J. dan Cook, T. 2007. *Mining Contiguous Sequential Patterns from Web Logs*. Proceedings of the WWW2007, Canada.

Chen, J. dan Cook, T. 2007. *Using d-gap Patterns for Index Compression*. Proceedings of the WWW2007, Canada.

Chen, J., Shankar, S., Kelly, A., Gningue, S. dan Rajaravivarma, R. 2009. *A Two Stage Approach for Contiguous Sequential Pattern Mining*. IEEE IRI 2009.

Cho, C.W. 2008. *Inverted Index and It's Compression*. http://140.122.184.128/presentation_1/08-10-16/Inverted%20Index%20and%20Its%20Compression.ppt. Tanggal akses: 10 Oktober 2011.

Delbru, R., Campinas, S., Samp, K., dan Tummarello, G. 2010. *Adaptive Frame Of Reference For Compressing Inverted Lists*. DERI (Digital Enterprise Research Institute), National University of Ireland, Galway , IDA Business Park, Lower Dangan, Galway, Ireland.

González, R. B. 2008. *Index Compression for Information Retrieval Systems*. University of A Coruña.

Manning, C.D., Raghavan, P. dan Schütze, H. 2009. *An Introduction to Information Retrieval*. Cambrige University Press, England.

Mohlenkamp, M. 2010. *Mathematical Symbols*. http://www.ohio.edu/people/mohlenka/goodproblems/symbols.pdf. Department of Mathematics, Ohio University, USA. Tanggal akses: 12 Desember 2011.

Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., dan Mei-Chun Hsu. 2001. *PrefixSpan: Mining Sequential Patterns Efficiently by Prefix Projected Pattern Growth*. Natural Sciences and Engineering Research Council of Canada (grant NSERC-A3723), the Networks of Centres of Excellence of Canada (grant NCE/IRIS-3), and the Hewlett-Packard Lab, USA.

Salomon, David. 2007. *Data Compression, The Complete Reference Fourth Edition*. Springer, Northridge, CA, USA.

Wardoyo, I., Kusdinar, P. dan Taufik, I. H. 2005. *Kompresi Teks dengan Menggunakan Algoritma Huffman*. Jurusan Teknik Informatika, Sekolah Tinggi Teknologi Telkom, Bandung.

Williams, H. E. dan Zobel, J. 1999. *Compressing Integers for Fast File Access*. Department of Computer Science, RMIT University, Melbourne, Australia.

Witten, I. H., Moffat, A., dan Timothy, C. B. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann Publishers, San Francisco CA.

# LAMPIRAN

**Lampiran 1.** Hasil Pengujian Kompresi *Inverted List* untuk *Document Id***.**

Tabel 1. Hasil Kompresi Pada Jumlah Dokumen 250 untuk *Document Id*.

| Metode Kompresi | Ukuran *File* Asli (KB) | Ukuran *File* Kompresi (KB) | Rasio Kompresi | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 99.87 | 67.01 | 0.671 | 5850 |
| *Golomb* | 99.87 | 57.1 | 0.5717 | 5834 |
| *Gamma* dan *Huffman* (CSP) | 99.87 | 65.47 | 0.6556 | 17893 |
| (*) *Golomb* dan *Huffman* (CSP) | 99.87 | 55.6 | 0.5567 | 18189 |

Tabel 2. Hasil Kompresi Pada Jumlah Dokumen 500 untuk *Document Id*.

| Metode Kompresi | Ukuran *File* Asli (KB) | Ukuran *File* Kompresi (KB) | Rasio Kompresi | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 188.23 | 129.47 | 0.6878 | 14773 |
| *Golomb* | 188.23 | 106.41 | 0.5653 | 16084 |
| *Gamma* dan *Huffman* (CSP) | 188.23 | 124.88 | 0.6634 | 58079 |
| (*) *Golomb* dan *Huffman* (CSP) | 188.23 | 102.61 | 0.5451 | 60030 |

Tabel 3. Hasil Kompresi dengan Jumlah Dokumen 750 Pada *Document Id*.

| Metode Kompresi | Ukuran *File* Asli (KB) | Ukuran *File* Kompresi (KB) | Rasio Kompresi | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 276.34 | 191.86 | 0.6943 | 23213 |

| | | | | |
|---|---|---|---|---|
| *Golomb* | 276.34 | 154.81 | 0.5602 | 21356 |
| *Gamma* dan *Huffman* (CSP) | 276.34 | 184.59 | 0.668 | 121945 |
| (*) *Golomb* dan *Huffman* (CSP) | 276.34 | 149.54 | 0.5411 | 123474 |

Tabel 4. Hasil Kompresi dengan Jumlah Dokumen 1000 Pada *Document Id*.

| Metode Kompresi | Ukuran *File* Asli (KB) | Ukuran *File* Kompresi (KB) | Rasio Kompresi | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 361.65 | 253.66 | 0.7014 | 37403 |
| *Golomb* | 361.65 | 201.98 | 0.5585 | 40460 |
| *Gamma* dan *Huffman* (CSP) | 361.65 | 243.42 | 0.6731 | 216014 |
| (*) *Golomb* dan *Huffman* (CSP) | 361.65 | 195.05 | 0.5393 | 212191 |

**Lampiran 2.** Hasil Pengujian Kompresi *Inverted List* untuk *Term Frequencies***.**

Tabel 1. Hasil Kompresi dengan Jumlah Dokumen 250 Pada *Term Frequencies*.

| Metode Kompresi | Ukuran *File* Asli (KB) | Ukuran *File* Kompresi (KB) | Rasio Kompresi | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 99.87 | 56.38 | 0.5645 | 6255 |
| *Golomb* | 99.87 | 50.11 | 0.5018 | 5631 |
| *Gamma* dan *Huffman* (CSP) | 99.87 | 46.66 | 0.4672 | 19172 |
| (*) *Golomb* dan *Huffman* (CSP) | 99.87 | 41.73 | 0.4178 | 19953 |

Tabel 2. Hasil Kompresi dengan Jumlah Dokumen 500 Pada *Term Frequencies*.

| Metode Kompresi | Ukuran *File* Asli (KB) | Ukuran *File* Kompresi (KB) | Rasio Kompresi | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 188.23 | 106.74 | 0.5671 | 12043 |
| *Golomb* | 188.23 | 94.45 | 0.5018 | 11762 |
| *Gamma* dan *Huffman* (CSP) | 188.23 | 96.69 | 0.5137 | 52728 |
| (*) *Golomb* dan *Huffman* (CSP) | 188.23 | 86.38 | 0.4589 | 54247 |

Tabel 3. Hasil Kompresi dengan Jumlah Dokumen 750 Pada *Term Frequencies*.

| Metode Kompresi | Ukuran *File* Asli (KB) | Ukuran *File* Kompresi (KB) | Rasio Kompresi | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 276.34 | 157.26 | 0.5691 | 22507 |
| *Golomb* | 276.34 | 138.67 | 0.5018 | 21255 |
| *Gamma* dan *Huffman* (CSP) | 276.34 | 139.87 | 0.5062 | 102461 |

| | | | | |
|---|---|---|---|---|
| (*) *Golomb* dan *Huffman* (CSP) | 276.34 | 124.68 | 0.4512 | 106156 |

Tabel 4. Hasil Kompresi dengan Jumlah Dokumen 1000 Pada *Term Frequencies*.

| Metode Kompresi | Ukuran *File* Asli (KB) | Ukuran *File* Kompresi (KB) | Rasio Kompresi | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 361.65 | 206.13 | 0.57 | 28423 |
| *Golomb* | 361.65 | 181.5 | 0.5019 | 27550 |
| *Gamma* dan *Huffman* (CSP) | 361.65 | 185.01 | 0.5116 | 239677 |
| (*) *Golomb* dan *Huffman* (CSP) | 361.65 | 166.72 | 0.461 | 249512 |

**Lampiran 3.** Hasil Pengujian Dekompresi *Inverted List* untuk *Document Id*.

Tabel 1. Percobaan Dekompresi Pada *File* Kompresi dengan Jumlah Dokumen 250 untuk *Document Id*.

| Metode Dekompresi | Jumlah *Inverted List* | Jumlah Kesesuaian *Inverted List* | Persentase Kesesuaian (%) | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 4129 | 4129 | 100 | 1607 |
| *Golomb* | 4129 | 4129 | 100 | 1591 |
| *Gamma* dan *Huffman* (CSP) | 4129 | 4129 | 100 | 1622 |
| (*) *Golomb* dan *Huffman* (CSP) | 4129 | 4129 | 100 | 1685 |

Tabel 2. Percobaan Dekompresi Pada *File* Kompresi dengan Jumlah Dokumen 500 untuk *Document Id*.

| Metode Dekompresi | Jumlah *Inverted List* | Jumlah Kesesuaian *Inverted List* | Persentase Kesesuaian (%) | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 5566 | 5566 | 100 | 1966 |
| *Golomb* | 5566 | 5566 | 100 | 1981 |
| *Gamma* dan *Huffman* (CSP) | 5566 | 5566 | 100 | 2309 |
| (*) *Golomb* dan *Huffman* (CSP) | 5566 | 5566 | 100 | 2153 |

Tabel 3. Percobaan Dekompresi Pada *File* Kompresi dengan Jumlah Dokumen 750 untuk *Document Id*.

| Metode Dekompresi | Jumlah *Inverted List* | Jumlah Kesesuaian *Inverted List* | Persentase Kesesuaian (%) | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 6540 | 6540 | 100 | 2371 |

| | | | | |
|---|---|---|---|---|
| *Golomb* | 6540 | 6540 | 100 | 2356 |
| *Gamma* dan *Huffman* (CSP) | 6540 | 6540 | 100 | 2714 |
| (*) *Golomb* dan *Huffman* (CSP) | 6540 | 6540 | 100 | 2636 |

Tabel 4. Percobaan Dekompresi Pada *File* Kompresi dengan Jumlah Dokumen 1000 untuk *Document Id*.

| Metode Dekompresi | Jumlah *Inverted List* | Jumlah Kesesuaian *Inverted List* | Persentase Kesesuaian (%) | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 7353 | 7353 | 100 | 2824 |
| *Golomb* | 7353 | 7353 | 100 | 2776 |
| *Gamma* dan *Huffman* (CSP) | 7353 | 7353 | 100 | 2839 |
| (*) *Golomb* dan *Huffman* (CSP) | 7353 | 7353 | 100 | 3136 |

**Lampiran 4.** Hasil Pengujian Dekompresi *Inverted List* untuk *Term Frequencies*.

Tabel 1. Percobaan Dekompresi Pada *File* Kompresi dengan Jumlah Dokumen 250 untuk *Term Frequencies*.

| Metode Dekompresi | Jumlah *Inverted List* | Jumlah Kesesuaian *Inverted List* | Persentase Kesesuaian (%) | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 4129 | 4129 | 100 | 1576 |
| *Golomb* | 4129 | 4129 | 100 | 1592 |
| *Gamma* dan *Huffman* (CSP) | 4129 | 4129 | 100 | 1653 |
| (*) *Golomb* dan *Huffman* (CSP) | 4129 | 4129 | 100 | 1623 |

Tabel 2. Percobaan Dekompresi Pada *File* Kompresi dengan Jumlah Dokumen 500 untuk *Term Frequencies*.

| Metode Dekompresi | Jumlah *Inverted List* | Jumlah Kesesuaian *Inverted List* | Persentase Kesesuaian (%) | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 5566 | 5566 | 100 | 1990 |
| *Golomb* | 5566 | 5566 | 100 | 2021 |
| *Gamma* dan *Huffman* (CSP) | 5566 | 5566 | 100 | 1982 |
| (*) *Golomb* dan *Huffman* (CSP) | 5566 | 5566 | 100 | 2059 |

Tabel 3. Percobaan Dekompresi Pada *File* Kompresi dengan Jumlah Dokumen 750 untuk *Term Frequencies*.

| Metode Dekompresi | Jumlah *Inverted List* | Jumlah Kesesuaian *Inverted List* | Persentase Kesesuaian (%) | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 6540 | 6540 | 100 | 2445 |

| | | | | |
|---|---|---|---|---|
| *Golomb* | 6540 | 6540 | 100 | 2458 |
| *Gamma* dan *Huffman* (CSP) | 6540 | 6540 | 100 | 2480 |
| (*) *Golomb* dan *Huffman* (CSP) | 6540 | 6540 | 100 | 2491 |

Tabel 4. Percobaan Dekompresi Pada *File* Kompresi dengan Jumlah Dokumen 1000 untuk *Term Frequencies*.

| Metode Dekompresi | Jumlah *Inverted List* | Jumlah Kesesuaian *Inverted List* | Persentase Kesesuaian (%) | Waktu (ms) |
|---|---|---|---|---|
| *Gamma* | 7353 | 7353 | 100 | 2980 |
| *Golomb* | 7353 | 7353 | 100 | 2792 |
| *Gamma* dan *Huffman* (CSP) | 7353 | 7353 | 100 | 3401 |
| (*) *Golomb* dan *Huffman* (CSP) | 7353 | 7353 | 100 | 3073 |

**Lampiran 5.** Hasil Pendeteksian *ContiIguous Sequential Patterns* (CSP) dari *Inverted List*.

Tabel 1. Pendeteksian CSP dari *Inverted List* untuk *Document Id*.

| CSP Ke - | Dokumen | | | |
|---|---|---|---|---|
| | 250 | 500 | 750 | 1000 |
| 1 | 111 | 1111 | 11111 | 11111 |
| 2 | - | 12 | 12 | 112 |
| 3 | - | - | 13 | 13 |
| 4 | - | - | - | 14 |

Tabel 2. Pendeteksian CSP dari *Inverted List* untuk *Term Frequencies*.

| CSP Ke - | Dokumen | | | |
|---|---|---|---|---|
| | 250 | 500 | 750 | 1000 |
| 1 | 111111 | 211111111 | 2111111111 | 211111111111 |
| 2 | 112 | 11212 | 11212 | 11212 |
| 3 | 13 | 2122 | 2122 | 2122 |
| 4 | - | 213 | 1123 | 1123 |
| 5 | - | 214 | 213 | 213 |
| 6 | - | 113 | 214 | 214 |
| 7 | | - | 113 | 113 |

**Lampiran 6.** Hasil Percobaan Rasio Kompresi, Waktu Dekompresi, dan Pendeteksian CSP untuk *Inverted List* Tanpa *Filter* Dokumen.

Tabel 1. Hasil Rasio Kompresi *Inverted List* Tanpa *Filter* untuk *Document Id*.

| Dok | ∑*Inverted List* | ∑ CSP | *m* | Ukuran *File* Asli (KB) | Ukuran *File* Kompresi (KB) | Rasio | Waktu (ms) |
|---|---|---|---|---|---|---|---|
| 250 | 7277 | 2 | 21 | 176.86 | 95.96 | 0.5426 | 39483 |
| 500 | 10090 | 3 | 29 | 335.36 | 179.61 | 0.5356 | 118291 |
| 750 | 11968 | 3 | 36 | 490.99 | 260.08 | 0.5297 | 296421 |
| 1000 | 13601 | 4 | 41 | 643.26 | 338.92 | 0.5269 | 482286 |

Tabel 2. Percobaan Dekompresi Tanpa *Filter* untuk *Document Id*.

| Jumlah Dokumen | Jumlah *Inverted List* | Jumlah Kesesuaian *List* | Persentase (%) | Waktu (ms) |
|---|---|---|---|---|
| 250 | 7277 | 7277 | 100 | 2389 |
| 500 | 10090 | 10090 | 100 | 3453 |
| 750 | 11968 | 11968 | 100 | 4357 |
| 1000 | 13601 | 13601 | 100 | 5804 |

Tabel 3. Hasil Rasio Kompresi *Inverted List* Tanpa *Filter* untuk *Term Frequencies*.

| Dok | ∑*Inverted List* | ∑ CSP | *m* | Ukuran *File* Asli (KB) | Ukuran *File* Kompresi (KB) | Rasio | Waktu (ms) |
|---|---|---|---|---|---|---|---|
| 250 | 7277 | 3 | 10 | 176.86 | 73.91 | 0.4179 | 51474 |
| 500 | 10090 | 5 | 10 | 335.36 | 153.33 | 0.4572 | 143949 |
| 750 | 11968 | 7 | 10 | 490.99 | 221.84 | 0.4518 | 237280 |
| 1000 | 13601 | 7 | 10 | 643.26 | 290.95 | 0.4523 | 350421 |

Tabel 4. Percobaan Dekompresi Tanpa *Filter* untuk *Term Frequencies*.

| Jumlah Dokumen | Jumlah *Inverted List* | Jumlah Kesesuaian *List* | Persentase (%) | Waktu (ms) |
|---|---|---|---|---|
| 250 | 7277 | 7277 | 100 | 2107 |
| 500 | 10090 | 10090 | 100 | 2879 |
| 750 | 11968 | 11968 | 100 | 3729 |
| 1000 | 13601 | 13601 | 100 | 4568 |

Tabel 5. Pendeteksian CSP dari *Inverted List* Tanpa *Filter* untuk *Document Id*.

| CSP Ke - | Dokumen | | | |
|---|---|---|---|---|
| | 250 | 500 | 750 | 1000 |
| 1 | 1111 | 1111 | 11111 | 11111 |
| 2 | 12 | 12 | 112 | 112 |
| 3 | - | 13 | 13 | 13 |
| 4 | - | - | - | 14 |

Tabel 6. Pendeteksian CSP dari *Inverted List* Tanpa *Filter* untuk *Term Frequencies*.

| CSP Ke - | Dokumen | | | |
|---|---|---|---|---|
| | 250 | 500 | 750 | 1000 |
| 1 | 111111 | 211111111 | 21111111111 | 2111111111111 |
| 2 | 112 | 11212 | 11212 | 11212 |
| 3 | 13 | 2122 | 2122 | 2122 |
| 4 | - | 213 | 1123 | 1123 |
| 5 | - | 113 | 213 | 213 |
| 6 | - | - | 214 | 214 |
| 7 | - | - | 113 | 113 |