Perbandingan Performansi Algoritma Adler-32 dan CRC-32 pada Library zlib

Ditto Narapratama (13504132)
Jurusan Teknik Informatika ITB, Bandung, email: dittonara@yahoo.com

Abstrak *– Algoritma Adler-32 dan CRC32 adalah termasuk ke dalam fungsi hash yang berfungsi untuk memeriksa integritas data. Perubahan yang dilakukan terhadap data saat transmisi atau saat di dalam penyimpanan dapat dideteksi dengan kedua algoritma ini. Kedua Algoritma ini cukup populer dan banyak diimplementasikan ke berbagai aplikasi. Library zlib adalah library untuk kompresi data yang populer dan memuat kedua algoritma ini. Pada makalah ini penulis akan menjelaskan sekilas tentang kedua algoritma ini, lalu melakukan perbandingan performansi atau kecepatan kalkulasi kedua algoritma ini.Perbandingan performansi algoritma dilakukan dengan cara menjalankan program console yang dibuat dalam bahasa c++. Program ini menjalankan algoritma CRC-32 dan Adler-32 yang ada di library zlib, lalu menghitung kecepatan kalkulasinya. Terdapat tiga file untuk test case yang dijalankan, setelah itu akan diambil kesimpulan dari percobaan ini.Dari hasil percobaan dapat diambil kesimpulan bahwa algoritma Adler-32pada library zlib lebih cepat daripada algoritma CRC-32.*

Kata Kunci*: hash ,zlib, adler-32, crc-32*

1. PENDAHULUAN

Seiring dengan meningkatnya kejahatan di internet, faktor keamanan menjadi hal yang semakin penting dan harus diperhatikan. Salah satu aspek keamanan yang harus diperhatikan adalah integritas data. Integritas data adalah layanan yang menjamin bahwa pesan masih asli/utuh atau belum pernah dimanipulasi selama pengiriman. Salah satu cara untuk memeriksa integritas data adalah dengan menggunakan fungsi hash.

Fungsi Hash adalah fungsi yang menerima masukan *string* yang panjangnya sembarang, lalu kemudian ditransformasikan ke *string* yang panjangnya tetap (umumnya berukuran jauh lebih kecil dari data semula).

Fungsi hash dapat menerima masukan *string* apa saja. Jika *string* menyatakan pesan, maka sembarang pesan *M* berukuran bebas dikompresi oleh fungsi hash *H* melalui persamaan.

$$H = H(M)$$

Keluaran dengan panjang tetap ini disebut dengan nilai hash (*hash value*), pesan ringkas (*message digest*) atau *checksum*.

Fungsi *hash* bersifat satu arah (*one way*), artinya pesan yang sudah diubah menjadi *message digest* tidak dapat dikembalikan lagi menjadi pesan semula. Sifat-sifat fungsi *hash* adalah :

1. Fungsi *H* dapat diterapkan pada blok data berukuran berapa saja
2. *H* menghasilkan nilai (*h*) dengan panjang tetap (*fixed-length output*)
3. $H(x)$ mudah dihitung untuk setiap nilai *x* yang diberikan
4. Untuk setiap *h* yang diberikan, tidak mungkin menemukan *x* sedemikian sehingga $H(x) = h$
5. Untuk setiap *x* yang diberikan, tidak mungkin mencari $y \neq x$ sedemikian sehingga $H(y) = H(x)$
6. Tidak mungkin (secara komputasi) mencari pasangan *x* dan *y* sedemikian sehingga $H(x) = H(y)$

Sebuah fungsi *hash* dianggap tidak aman jika secara komputasi dimungkinkan menemukan pesan yang bersesuaian dengan pesan ringkasnya, dan terjadi kolisi (*collision*), yaitu terdapat beberapa pesan berbeda yang mempunyai pesan ringkas yang sama.

2. ADLER-32

Keluaran dari algoritma Adler-32 didapat dari perhitungan dua bilangan 16-bit *A* dan *B*. Bilangan *A* dan *B* ini kemudian akan dikonkatenasi menjadi bilangan integer 32 bit. *A* adalah jumlah dari semua *byte* di *string* ditambah dengan satu, sedangkan *B* adalah jumlah dari setiap nilai-nilai dari *A* dari setiap langkah.

Pada permulaan algoritma Adler-32, nilai *A* diinisialisasi dengan nilai 1, sedangkan *B* diinisialisasi dengan nilai 0. Hasil dari penjumlahan kemudian akan dimodulo dengan 65521, yang merupakan bilangan prima terbesar yang lebih kecil dari 2^{16}.

Fungsi Adler-32 ini dapat diekspresikan sebagai berikut :

$$A = 1 + D_1 + D_2 + \ldots + D_n \pmod{65521}$$

$B = (1 + D_1) + (1 + D_1 + D_2) + \ldots + (1 + D_1 + D_2 + \ldots + D_n) \pmod{65521}$
$= n \times D_1 + (n-1) \times D_2 + (n-2) \times D_3 + \ldots + D_n + n \pmod{65521}$

$Adler\text{-}32(D) = B \times 65536 + A$

Dimana *D* adalah *string byte* masukan, dan *n* adalah panjang dari *D*.

Sebagai contoh, kalkulasi *string*"Kriptografi" dengan menggunakan algoritma Adler-32 adalah sebagai berikut :

Kode Ascii (basis 10)	A	B
K : 75	1 + 75 = 76	0 + 76 = 76
r : 114	76 + 114 = 190	76 + 190 = 266
i : 105	190 + 105 = 295	266 + 295 = 561
p : 112	295 + 112 = 407	561 + 407 = 958
t : 116	407 + 116 = 519	958 + 519 = 1477
o : 111	519 + 111 = 630	1477 + 630 = 2107
g : 103	630 + 103 = 733	2107 +733 = 2840
r : 114	733 + 114 + 847	2840 + 847 = 3687
a : 97	847 + 97 = 944	3687 + 944 = 4631
f : 102	944 + 102 = 1046	4631 + 1046 = 5677
i : 105	1046 + 105 = 1151	5677 + 1151 = 6828

A = 1151 modulo 65521 = 1151 (basis 10) = 47F (basis 16)

B = 6828 modulo 65521 = 6828 (basis 10) = 1AAC (basis 16)

Output = 447480959 (basis 10) = 1AAC047F (basis 16)

3. CRC-32

CRC-32 adalah salah satu jenis algoritma *Cyclic Redundancy Check* (CRC). CRC adalah salah satu fungsi *hash* yang berguna untuk mendeteksi kerusakan data dalam proses transmisi ataupun penyimpanan. CRC menghasilkan suatu *checksum* yaitu suatu nilai dihasilkan dari fungsi hash-nya, dimana nilai ini akan digunakan untuk mendeteksi error pada transmisi ataupun penyimpanan data. Nilai CRC dihitung dan digabungkan sebelum dilakukan transmisi data atau penyimpanan, dan kemudian penerima akan melakukan verifikasi apakah data yang diterima tidak mengalami perubahan ataupun kerusakan.

CRC cukup terkenal karena mudah diterapkan dalam hardware, dan mudah dilakukan analisis secara matematika. Prinsip utama yang digunakan adalah dengan melakukan pembagian polinomial dengan mengabaikan bit-bit carry. Cara yang biasa digunakan adalah dengan menggunakan tabel CRC yang nilainya telah dihitung sebelumnya, sehingga dapat menghemat waktu dan meminimalisir kesalahan di tengah perhitungan.

CRC adalah menganggap file sebagai suatu string yang besar, yang terdiri dari bit-bit, dan kita operasikan sebagai suatu bilangan *polynomial* yang sangat besar. Untuk menghitung nilai CRC, kita membagi bilangan polynomial, sebagai representasi dari file, dengan suatu bilangan polynomial kecil yang sudah terdefinisi untuk jenis varian CRC tertentu. Nilai CRC adalah **sisa** hasil bagi tersebut, yang biasa disebut dengan **checksum**.

Setiap operasi pembagian pasti menghasilkan suatu sisa hasil bagi (meskipun bernilai 0), tetapi ada perbedaan dalam melakukan pembagian pada penghitungan CRC ini. Secara umum (prinsip aljabar biasa), pembagian dapat kita lakukan dengan mengurangi suatu bilangan dengan pembaginya secara terus-menerus sampai menghasilkan suatu sisa hasil bagi (yang lebih kecil dari bilangan pembagi). Dari nilai hasil bagi, sisa hasil bagi, dan bilangan pembagi kita bisa mendapat bilangan yang dibagi dengan mengalikan bilangan pembagi dengan hasil bagi dan menambah dengan sisa hasil bagi. Dalam penghitungan CRC, operasi pengurangan dan penjumlahan dilakukan dengan mengabaikan setiap nilai *carry* yang didapat. Tentu saja hal ini juga akan berpengaruh pada proses pembagian yang menjadi dasar utama dalam melakukan penghitungan nilai CRC. Operasi dalam CRC juga hanya melibatkan nilai 0 dan 1, karena secara umum kita beroperasi dalam level bit. Contoh penghitungan dalam CRC adalah sebagai berikut:

```
(1) 1101  (2) 1010   1010
    1010-     1111+  1111-
    ----      ----   ----
    0011      0101   0101
```

Pada contoh tersebut, operasi pertama (1) adalah operasi yang umum digunakan dalam operasi aljabar, yaitu dengan menghitung nilai *carry* yang dihasilkan, sedangkan operasi kedua (2) adalah operasi dasar yang akan kita gunakan dalam proses penghitungan nilai CRC. Nilai carry diabaikan, sehingga operasi pengurangan dan penambahan akan menghasilkan suatu nilai yang sama. Kedua operasi ini bisa dilakukan dengan melakukan penjumlahan dan dimodulo 2, atau dalam dunia pemrograman lebih dikenal dengan operasi XOR (istilah ini yang akan

lebih sering kita gunakan untuk menyebut penjumlahan pada operasi penghitungan CRC).
Pada proses pembagian yang dilakukan, akan tampak sekali bedanya karena pengurangan yang dilakukan dilakukan seperti melakukan penambahan. Nilai hasil bagi diabaikan, karena kita tidak menggunakannya, jadi hanya sisa hasil bagi (*remainder*) yang kita perhatikan. Dan *remainder* inilah yang akan menjadi dasar bagi nilai CRC yang dihasilkan.

Contoh pembagian :

```
10011/1101011010000\110000101
      10011|||||||| -
      -----||||||||
       10011|||||||
       10011||||||| -
       -----|||||||
        00001||||||
        00000|||||| -
        -----||||||
         00010|||||
         00000||||| -
         -----|||||
          00101||||
          00000|||| -
          -----||||
           01010|||
           00000||| -
           -----|||
            10100||
            10011|| -
            -----||
             01110|
             00000| -
             -----|
              11100
              10011 -
              -----
               1111 -> sisa hasil bagi
```

Nilai remainder inilah yang menjadi nilai CRC.

Selain dengan cara aljabar ada juga dengan cara menggunakan tabel, cara ini jauh lebih efektif dan hampir semua algoritma CRC-32 yang ada menggunakan cara ini, temasuk library zlib. Tetapi karena keterbatasan tempat, maka cara ini tidak akan dijelaskan pada makalah ini.

4. PENGUJIAN

4.1 Metode Pengujian

Pengujian dilakukan dengan menggunakan algoritma CRC-32 dan Adler-32 pada library zlib. Library zllb ditulis oleh Jean-loup Gailly dan Mark Adler, dan merupakan library kompresi yang dipakai oleh banyak aplikasi terkenal, seperti linux kernel, apache, HTTP server, dpkg, FFmpeg dan lain-lain. Algoritma CRC-32 pada zlib ada dua macam, yang dioptimasi dan yang tidak. Pada percobaan ini penulis menjalankan kedua macam algoritma CRC-32 tersebut.
Program yang dipakai untuk mengukur kecepatan kalkulasi algoritma adalah program *console* yang dibuat dengan bahasa C++. Program ini menjalankan algoritma CRC-32 dan Adler-32 dari library zlib dan kemudian mengukur kecepatan kalkulasinya.

Pada saat pengujian Windows XP berada dalam keadaan *safe mode*. Hal ini dilakukan agar tidak ada aplikasi-aplikasi lain yang membebani prosesor dan kalkulasi kecepatan algoritma menjadi lebih akurat.

Spek komputer yang dipakai untuk pengujian adalah :
Laptop Vaio T350P
- OS : Windows XP Professional SP2
- Prosesor : Intel Pentium M 1,2 GHz
- Instruksi : MMX, SSE, SSE2
- Memory : DDR 504MB

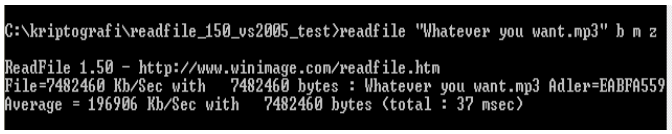
Algoritma CRC-32 dan Adler-32 dilakukan pada beberapa file *test case* dengan ukuran yang berbeda-beda. Terdapat tiga file *test case*, yaitu :

-Test file 1 (Whatever your want.mp3) : 7,13MB
-Test file 2 (For the birds.avi) : 62,6 MB
-Test file 3 (Atonement-cd1.avi) : 658 MB

4.2 Hasil Pengujian

Algoritma Adler 32 :

File Test 1 :



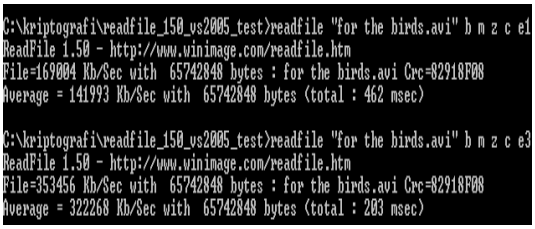
File Test 2 :

File Test 3 :

Algoritma CRC-32 (*non-optimized dan optimized*)

File Test 1 :

File Test 2 :



File Test 3 :

Dilihat dari hasil percobaan diatas, terlihat bahwa algoritma Adler-32 pada library zlib lebih cepat daripada CRC-32 yang sudah dioptimasi sekalipun. Tetapi menurut pemilik algoritma ini [2], algortima ini masih memiliki kemungkinan untuk adanya error yang tidak terdeteksi. Tetapi belum tentu Adler-32 lebih cepat dari CRC-32 secara umum, karena CRC-32 masih bisa lebih dioptimasi lagi bila dengan memanfaatkan *hardware*.

5. KESIMPULAN

Dari percobaan di atas, dapat diambil kesimpulan bahwa algoritma Adler-32 pada library zlib lebih cepat dari algoritma CRC-32, baik yang belum dioptimasi maupun yang sudah dioptimasi. Tetapi algoritma Adler-32 tetap memiliki kemungkinan untuk tidak mendeteksi error, walaupun kemungkinannya kecil
.
Kesimpulan lain yang bisa diambil adalah percobaan ini tidak bisa menyimpulkan bahwa Adler-32 lebih cepat dari CRC-32secara umum, karena algoritma CRC-32 masih bisa dioptimasi lagi dengan memanfaatkan *hardware*.

DAFTAR REFERENSI

[1]http://findarticles.com/p/articles/mi_qa3899/is_200301/ai_n9226373/pg_1
[2] http://ietf.org/rfc/rfc1950.txt
[3] www.zlib.net
[4] http://www.winimage.com/misc/readfile_test.htm
[5]http://guru.multimedia.cx/crc32-vs-adler32/
[6]Penggunaan CRC32 dalam Integritas Data, Indra Sakti Wijayanto