Hak cipta dan penggunaan kembali:

Lisensi ini mengizinkan setiap orang untuk menggubah, memperbaiki, dan membuat ciptaan turunan bukan untuk kepentingan komersial, selama anda mencantumkan nama penulis dan melisensikan ciptaan turunan dengan syarat yang serupa dengan ciptaan asli.

Copyright and reuse:

This license lets you remix, tweak, and build upon work non-commercially, as long as you credit the origin creator and license it on your new creations under the identical terms.

Team project ©2017
Dony Pratidana S. Hum | Bima Agus Setyawan S. IIP

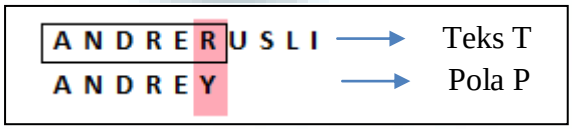
BAB II

TINJAUAN PUSTAKA

2.1 String Searching

String adalah urutan dari karakter, yang mana karakter ini dapat terdiri dari beberapa *alphabet* (Utomo, 2008). Sebagai contoh, ada *string* ASCII yang terdiri dari 256 alphabet, atau juga *string* biner yang terdiri dari karakter 0 dan 1 saja. *String searching*, atau dalam Bahasa Indonesia pencarian *string*, merupakan suatu proses pencarian suatu pola atau penggalan *string* di dalam suatu teks atau *string* yang lain yang memiliki suatu panjang tertentu. Secara teknis, *string searching* merupakan proses pencarian pola P yang memiliki panjang m, dalam suatu teks T yang memiliki panjang n.

String searching dianggap menemukan hasilnya bila pola yang dicari dan karakter pada teks yang dibandingkan adalah sama, sedangkan dianggap gagal bila terjadi kondisi sebaliknya (Utomo, 2008).



Gambar 2.1 Perbandingan *String*

* kotak merah muda menunjukkan adanya perbedaan atau ketidakcocokan *string*

Window atau kotak yang membatasi pola pada teks diatas berfungsi untuk membantu pencarian pola dalam teks. Ukuran window sama dengan ukuran pola yang ingin dicari. Pada pertama kali, *window* ini diletakkan di posisi paling kiri dari teks, lalu teks yang ada dibandingkan dengan teks yang dicari. Pergeseran *window* ke

kanan akan dilakukan bisa karena 2 sebab, yang pertama yaitu bila terjadi kecocokan, maka pergeseran dilakukan untuk mencari pola berikutnya bila ada didalam teks. Selain sebab itu, pergeseran juga dapat dilakukan bila terjadi ketidakcocokan, maka pola akan terus dicari di dalam teks. Proses ini berlangsung terus menerus sampai *window* melebihi batas kanan dari teks T.

Berikut *pseudocode* dari *string searching* yang paling dasar (Eppstein, 2013):

```
for (i=0; T[i] != '\0'; i++)
  {
  for (j=0; T[i+j] != '\0' && P[j] != '\0' && T[i+j]==P[j]; j++) ;
  if (P[j] == '\0') found a match
  }
```

Pada algoritma brute force ini, semua kemungkinan dicoba, walaupun dari iterasi sebelumnya dapat diketahui bahwa pada iterasi berikutnya, data akan mengalami ketidakcocokan.

```
  T: b  a  n  a  n  a  n  o  b  a  n  o

i=0: X
i=1:    X
i=2:       n  a  n  X
i=3:          X
i=4:             n  a  n  o
i=5:                X
i=6:                   n  X
i=7:                         X
i=8:                            X
i=9:                               n  X
i=10:                                 X
```

Gambar 2.2 Simulasi *String Searching* (Eppstein, 2013)

Pada gambar 2.1, setiap baris merupakan iterasi luar yang pada pseudocode sebelumnya dinyatakan oleh *i*. Dapat diperhatikan bahwa pada iterasi i=2, sudah dapat disimpulkan bahwa pada iterasi yang berikutnya yaitu i=3, data yang ada tidak mungkin cocok, namun tetap iterasi tersebut dilakukan.

Ada dua kali *looping* yang terjadi *loop* yang ada di dalam memiliki waktu pemrosesan sebesar O(m) dan *looping* yang diluar bernilai O(n), sehingga total waktu yang dibutuhkan untuk melakukan semua iterasi bernilai O(mn) (Eppstein, 2013).

2.2 Algoritma Knuth Morris Pratt

Algoritma Knuth Morris Pratt merupakan algoritma *string matching* atau pencocokan *string*. Algoritma ini dikembangkan bersama oleh Donald Knuth dan Vaughan Pratt pada tahun 1974, dan secara terpisah dikembangkan oleh James H. Morris. Lalu dipublikasikan bersama pada tahun 1977 pada SIAM *Journal of Computing* (Clow, 2010).

Algoritma Knuth Morris Pratt atau KMP merupakan algoritma yang digunakan untuk pencocokan *string* yang diinput dengan *string* pada berkas dokumen yang dicari. Algoritma ini merupakan tipe algoritma *Exact String Matching Algorithm* yang berarti algoritma ini mencocokan suatu kata dengan kata yang sama pula tanpa ada perbedaan (Ekaputri, 2006). Misalnya bila kata “algoritma” yang dicari, maka hasil pencarian akan mengembalikan dokumen yang cocok dengan kata “algoritma” juga. Pada algoritma ini, informasi akan disimpan untuk dilakukannya pergeseran *string* yang lebih jauh. Berikut beberapa definisi dari algoritma Knuth Morris Pratt (Ekaputri, 2006) :

a. Awalan (prefix) dari x adalah upa-*string* (*substring*) u dengan u = x1x2...xk - 1, k ∈ {1, 2, ..., k - 1} dengan kata lain, x diawali dengan u,

b. Akhiran (suffix) dari x adalah upa-*string* (sub*string*) u dengan u = xk - b xk - b + 1...xk , k ∈ {1, 2, ..., k – 1}dengan kata lain, x di akhiri dengan v, dan

c. Pinggiran (*border*) dari x adalah upa-*string* r sedemikian sehingga r = x1x2...xk - 1 dan u = xk - b xk - b + 1...xk , k ∈ {1, 2, ..., k-1}, dengan kata lain, pinggiran dari x adalah upa-*string* yang keduanya awalan dan juga akhiran sebenarnya dari x.

Ide dari algoritma ini adalah bagaimana dapat memanfaatkan karakter-karakter pola yang sudah diketahui ada di dalam teks sampai terjadinya ketidakcocokan untuk melakukan pergeseran.

Kompleksitas algoritma pencocokan *string* Knuth Morris Pratt ini dihitung dari kompleksitas untuk menghitung fungsi pinggiran dan pencarian *string*. Untuk menghitung fungsi pinggiran dibutuhkan waktu dengan penghitungan *O(m)* sedangkan untuk pencarian dibutuhkan waktu *O(n)*. Sehingga kompleksitas keseluruhan algoritma yaitu *O(m+n)* (UC Irvine, 1996).

2.3 Memori Cache Pada Mesin Pencari

Menyimpan hasil *query* dalam suatu memori cache merupakan suatu cara yang efektif untuk meningkatkan performa suatu mesin pencari atau *search engine* (Lempel, 2003). Hal tersebut disebabkan karena tingginya tingkat lokalitas suatu memori *cache* terhadap *query* yang dilakukan pengguna, secara suatu memori *cache*

dapat berada pada berbagai tempat seperti pada sisi klien, *proxy servers*, atau pada server dari suatu *website*. Adanya suatu memori *cache* berefek pada meningkatnya responsivitas dari suatu mesin pencari *website* karena ada *query* yang dilakukan oleh pengguna yang diproses oleh memori *cache* tanpa harus benar-benar melakukan *query* ke memori utama. Selain itu, ditemukannya data pada memori *cache* menghemat *resources* yang dibutuhkan oleh mesin pencari untuk menghitung hasil-hasil yang dikembalikan ke pengguna (Fagni, 2006).

2.4 Cache Based Query Processing

Teknologi *Cache Based Query Processing* digunakan untuk membuat suatu memori *cache* untuk menyimpan hasil-hasil *query* paling sering dimasukkan oleh *user*. Suatu memori *cache* yang menyimpan hasil-hasil tersebut dinamakan result *cache*. Result *cache* nantinya akan menyimpan hasil-hasil *query* yang terbaru atau yang diakses di akhir-akhir oleh *user*. *Result caching* secara umum berarti bahwa bila ada *query* yang sama atau identik dengan *query* lainnya yang baru saja diproses, maka sistem akan mengembalikan hasil sebelumnya (dengan asumsi bahwa tidak ada perubahan besar pada koleksi dokumen) (Zhang, 2008). Sehingga suatu result cache menyimpan hasil *top-k* dari semua *query* yang baru saja diproses oleh mesin pencari.

Dalam membuat result *cache* yang menyimpan hasil-hasil *query* yang paling sering diakses dan paling baru tersebut, ada dua pendekatan yaitu dengan membuat *aggregate ranking* dari *query* yang serupa, dan juga dengan membuat *inverted index*. Kedua cara tersebut hanya berbeda pada cara mendapatkan hasilnya. Kedua cara tersebut sama-sama mengambil masukan yang terbaru dari *user* untuk menghasilkan data pada result *cache*. Dalam penelitian ini digunakan pendekatan dengan membuat

aggregate ranking untuk *query* yang serupa. Pendekatan tersebut membuat suatu set *query* yang diidentifikasi serupa dengan masukkan *user* dan yang memasukkannya sebagai *result cache.* Berikut tiga pendekatan pembentukan *result cache* (q* = target *query*) (Cambazoglu, 2012) :

a. Memilih *query* di dalam *cache* yang *terms* nya membentuk suatu subset

$$\mathcal{Q}_{q^*}^{\text{sub}} = \{q : q \in \mathcal{Q}_{\text{C}} \wedge q \subset q^*\}.$$ .....Rumus 2.1

b. Memilih *query* dalam *cache* yang memiliki semua terms dalam masukkan *user*.

$$\mathcal{Q}_{q^*}^{\text{sup}} = \{q : q \in \mathcal{Q}_{\text{C}} \wedge q^* \subset q \wedge |q| = |q^*| + 1\}.$$ .....Rumus 2.2

c. Menggabungkan kedua hasil diatas

$$\mathcal{Q}_{q^*}^{\text{sub+sup}} = \mathcal{Q}_{q^*}^{\text{sub}} \cup \mathcal{Q}_{q^*}^{\text{sup}}.$$ .....Rumus 2.3

Mesin pencari yang dibuat berfungsi untuk menemukan data yang cocok dengan *query* yang dimasukkan *user*. Oleh karena itu, pendekatan yang kedua merupakan pendekatan yang selanjutnya akan digunakan dalam menyeleksi data yang akan disimpan pada memori *cache.*