Efficient and error-tolerant sequencing read mapping – supplementary materials

Piotr Jaroszyński, Norbert Dojer

1 Background

1.1 Rank operation

Definition 1.1 *Given a string $S_{1..n}$, $Rank_c(S, i)$ is equal to the number of occurrences of the symbol c in the substring $S_{1..i}$.*

1.2 Wavelet trees

Wavelet trees introduced in [6] allow to reduce the Rank operation on a sequence consisting of symbols from an arbitrary finite alphabet Σ to Rank operations on bit vectors. This is useful because, as we will see later, there exist efficient implementations of the Rank operation on bit vectors.

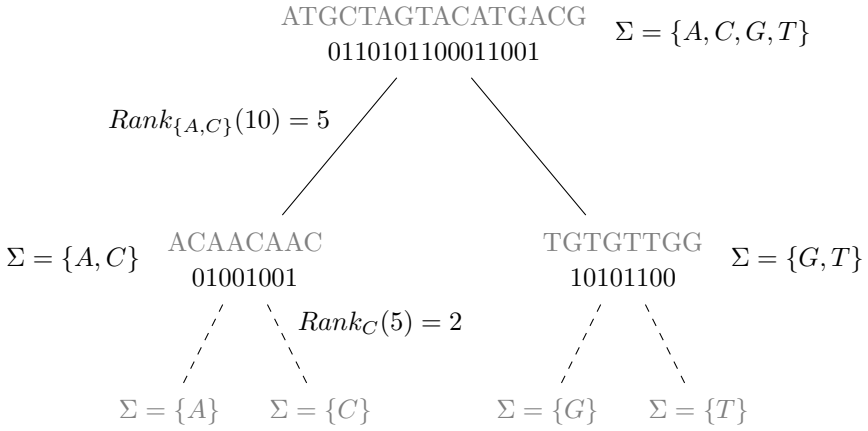

Figure 1: An example wavelet tree built for text "ATGCTAGTACATGACG". And the two-step process of calculating $Rank_C(10)$ on that text.

Given a string $S_{1..n}$ from alphabet Σ, the wavelet tree of that text is a binary tree on the alphabet Σ where each node V represents a subset of the alphabet Σ_V and its children V_l and V_r represent exclusive not empty subsets of Σ_V that sum up to Σ_V. The root node represents the whole alphabet and the leaves represent single symbols. Furthermore for each internal node V a subsequence S_V of S consisting only of symbols from Σ_V is considered, but only a bit vector of length $|S_V|$ is actually stored: $B_V[i] = 0$ iff $S_V[i] \in \Sigma_{V_l}$ and 1 otherwise. See Figure 1 for an example.

The algorithm to calculate $Rank_c(S, i)$ works as follows. For each node V that is encountered i is updated: $i = Rank_0(B_V, i)$ iff $c \in V_l$ and $i = Rank_1(B_V, i)$ otherwise. If $c \in V_l$ then the algorithm continues to node V_l and V_r otherwise. It starts at the root node and terminates with the result in i when a leaf node is encountered.

1.3 Suffix Array

Definition 1.2 *Given a text $T_{1..n}$, its suffix array SA is a sequence of integers such that suffixes $T_{SA[i]..n}$ for $i = 1..n$ are in lexicographical order.*

1.4 Burrows-Wheeler Transform

The *Burrows-Wheeler Transform (BWT)* [2] is a reversible permutation of text. It was first introduced as an aid in compression.

Definition 1.3 *Given a text $T_{1..n}$, where $T_n = \#$, and its suffix array $SA_{1..n}$ the Burrows-Wheeler Transform is defined as $BWT_i = T_{SA[i]-1}$, where it is assumed that $T_0 = T_n$.*

That is, BWT is a concatenation of the symbols preceding each suffix from the suffix array SA: $BWT = T_{SA[1]-1}T_{SA[2]-1}..T_{SA[n]-1}$.

A good conceptual way of thinking about the BWT is the following. Consider an array BA consisting of all the possible cyclic shifts of T (i.e. all strings of the form $T_{i..n}T_{1..i-1}$) sorted in lexicographical order. There is a close relation between the array BA and the suffix array SA as each row of BA has a different prefix of the form $T_{i..n}$ each ending with the symbol #, which is smaller than any other symbol. Hence the order of the rows in BA is the same as the order of corresponding suffixes in SA. See Figure 2.

Let F be the first column of BA and L be the last column of BA. F has a very simple structure as it consists of all the symbols from T sorted in lexicographical order. And given the cyclic form of the rows of BA, L consists of symbols preceding all the suffixes of T and thus $L = BWT$.

Moreover there exists a mapping between the column L and F, called the *LF-mapping* [2]. Consider the order of occurrences of symbol c in F: it depends only upon what they are followed by in T. Now consider the order of occurrences of c in L: given the cyclic nature of BA it again depends only upon what follows them in T. And hence for each symbol c from T, i-th occurrence of c in L

Figure 2: Construction of the BWT array BA. First column F in blue and last column $L = BWT$ in green. Text following suffixes is gray and corresponding values from the suffix array are orange.

corresponds to the i-th occurrence of c in F. Correspondence here means that both occurrences come from the same position in the original text T.

Definition 1.4 *$LF(i)$ is the position in F of the corresponding occurrence of symbol L_i.*

Let $C(c)$ be the number of symbols lexicographically smaller than c in T. And let $Occ(c, i) = Rank_c(L, i)$. Given these two functions we can calculate LF: $LF(i) = C(L_i) + Occ(L_i, i)$. $C(L_i)$ is the beginning of occurrences of symbol L_i in F and $Occ(L_i, i)$ gets us to the specific occurrence L_i.

Given how BA was constructed we know that F_i directly follows L_i in T. Hence given L_i and its corresponding occurrence $F_{LF(i)}$ we know that $L_{LF(i)}$ directly precedes L_i in T. This allows us to go backwards in T by performing the LF-mapping.

Given that $T_n = \#$ and that $\#$ is the smallest symbol we know that $F_1 = T_n = \#$ and hence $L_1 = T_{n-1}$.

Combining these two properties we can compute any T_i for $i = 1..n-1$ from $BWT = L$: $T_i = BWT[LF^{n-i-1}(1)]$ and thus reverse the BWT.

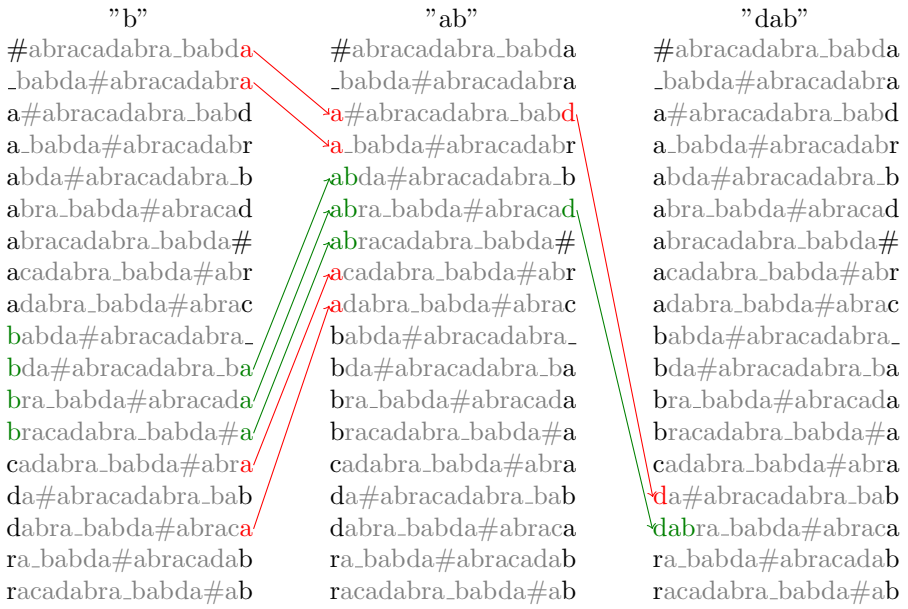

Figure 3: Searching for "dab" in "abracadabra_babda#". Green symbols are at some point part of the match and in the F column they represent the currently matched range. Red symbols are the occurrences of the symbol considered at each step that do not end up in the match. Arrows show the LF-mapping.

1.4.1 Searching

It is also possible to search for occurrences of query $Q_{1..m}$ in text $T_{1..n}$ that we constructed BWT from. The method was first introduced in [4] and was called the backward search. The name comes from the fact that the search starts by finding occurrences of the last symbol of the query first and works backwards to its beginning.

The algorithm performs m steps starting with $i = m$ and going down to $i = 1$. At each step it maintains a range $[sp_i, ep_i]$ (inclusive) of rows from the BWT array BA prefixed by $P_{i..m}$.

For $i = m$ the step is easy as we have already introduced function C, which does exactly what we want:

$$sp_m = C(Q_m) + 1$$
$$ep_m = C(Q_m + 1)$$

Where $Q_m + 1$ is the next symbol after Q_m in lexicographical order.

Following steps are a bit more involved:

$$sp_i = C(Q_i) + Occ(Q_i, sp_{i+1} - 1) + 1$$
$$ep_i = C(Q_i) + Occ(Q_i, ep_{i+1})$$

If at any point ep_i becomes smaller than sp_i then we know that $Q_{i..m}$ does not occur in the text T. Otherwise the size of the resulting range is the number of occurrences - i.e. $ep_i - sp_i + 1$.

This relation is the easiest to understand when looking at Figure 3. Consider the step between "b" and "ab". We have the range representing matches for "a" and we want to transform it to a range matching "ab". Given each occurrence $F_{b_{1..k}}$ of "b" in F we know that the only candidates that can match "ab" start at $L_{b_{1..k}}$ as they directly precede $F_{b_{1..k}}$. Knowing that and the fact that occurrences of the same symbol in F and L preserve order we can find the new range by counting "a"s that occur in L before L_{b_1} and up to L_{b_k} (this is what the Occ function does) and take that subrange of the range representing matches of "a" (calculated with function C) .

2 Implementation

2.1 Bit vectors

The biggest and most used bit vectors in the implementation are roughly of the size of the reference genome. As shown in [5] the best implementation for long bit vectors is a very simple single level caching one. The description follows.

The bits are stored explicitly as an array of 32-bit or 64-bit unsigned integers. In addition there is a second array that stores precomputed results of the Rank operation for every $CacheEvery_{bv}$ bit. The results are stored as 32-bit unsigned integers limiting the size of the bit vector to $2^{32} - 1$, which is just slightly more than the size of the human genome.

Having the precomputed results, calculating the rank for any position is easy: get the closest precomputed value and then count rest of the bits linearly. The counting itself uses precomputation as well: the number of bits in every 16-bit sequence (this is a slight change - the implementation described in the paper used 8-bit sequences) is stored in an array at the position which the bit sequence represents as an unsigned integer (e.g. bitcount[1234] =bitcount[1024 + 128 + 64 + 16 + 2] =bitcount[0000010011010010_b] = 5). This counting is represented by the *bitcount* function, which splits the *BaseSize* bits into 16-bits chunks first and adds up the partial bitcounts. The pseudo-code is showed as Algorithm 2.1 and an example is presented in Figure 4.

2.2 Wavelet tree

The implementation of the wavelet tree is simplified thanks to the small DNA alphabet. For the text of length n there is one bit vector of length n as the root node. And two bit vectors for $\{A, C\}$ and $\{G, T\}$ nodes that together are also

Algorithm 2.1 Pseudo-code for the Rank operation on a bit vector

```
1: function RANK_1(k)
2:     r ←precomputed[k/CacheEvery_bv] ▷ Get the closest precomputed value
3:     first ← k − k mod CacheEvery_bv ▷ Position of the first bit that needs to be counted directly
4:     for b ← first/BaseSize, k/BaseSize do ▷ Loop over the bits that can be counted fully in BaseSize chunks
5:         r ← r+bitcount(bits[b])
6:     end for
7:     r ← r+bitcount_first(bits[k/BaseSize], k mod BaseSize) ▷ Add the first k mod BaseSize bits from the last chunk
8: end function
```

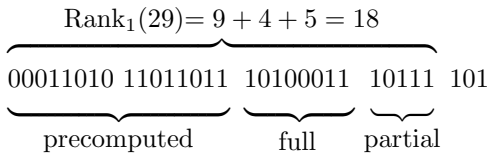

Figure 4: Calculating $\text{Rank}_1(29)$ on a bit vector where BaseSize is 8 and CacheEvery is 16. $\text{Rank}_1(16)$ is precomputed, next BaseSize bits can be counted directly and to count the last 5 bits the last 3 bits have to be masked off first.

Parameters	
n	Number of bits
BaseSize	Size of the unsigned integers used to store the bits - 32bit or 64bit
CacheEvery	Every how many bits should a precomputed value be stored
CacheSize	Size of the cached value - 32 bits
Properties	
Size	$n(1+\text{CacheSize}_{bv}/\text{CacheEvery}_{bv})$
Time to Rank	$O(\text{CacheEvery}_{bv})$

Table 1: Parameteres and properties of a bit vector.
The $O(\text{CacheEvery}_{bv})$ time indicates a linear space-time trade-off. Making the cache twice as dense makes the Rank operation roughly twice as fast. BaseSize does have an effect as well - at least using 64-bit integers on a 64-bit processor is faster than using 32-bit integers.

of length n. Figure 1 is an example of such a tree. The pseudo-code for Rank is presented as Algorithm 2.2.

Algorithm 2.2 Pseudo-code for the Rank operation on a wavelet tree

```
1: function RANK_c(k)    ▷ B_root, B_AC and B_GT are the bit vectors from the nodes
2:     if c ∈ {A, C} then
3:         r ←Rank_0(B_root, k)
4:         if c = A then
5:             r ←Rank_0(B_AC, r)
6:         else
7:             r ←Rank_1(B_AC, r)
8:         end if
9:     else
10:        r ←Rank_1(B_root, k)
11:        if c = G then
12:            r ←Rank_0(B_GT, r)
13:        else
14:            r ←Rank_1(B_GT, r)
15:        end if
16:    end if
17:    return r
18: end function
```

Parameters	
n	Length of the text
CacheEvery_{bv}	See bit vector
CacheSize_{bv}	See bit vector
Properties	
Size	$2 \cdot n(1+\text{CacheSize}/\text{CacheEvery})$
Time to Rank	$O(2 \cdot \text{CacheEvery})$

Table 2: Parameters and properties of a wavelet tree

2.3 Index

The index implemented in *Bmap* is based on SSA. It was chosen after analysing results described in [7] and [3]. Also preliminary tests were implemented that leveraged the implementations available on the Pizza & Chili site [1] referenced by [3]. The idea is to build a BWT from the reference genome and store it as a wavelet tree.

2.3.1 Find* and Count

To implement all these operations only the wavelet tree is necessary. The pseudo-code for *Find* and *Count* is presented as Algorithm 2.3.1. *FindSuffixes* instead of returning a single (sp, ep) range returns an array of them as

calculated in each step. *FindContinue* on the other hand skips the initialization of sp and ep and some of the iterations of the loop.

Algorithm 2.3 Pseudo-code for Find and Count

```
1:  function FIND(Q_{1..m})
2:      sp ← C(Q_m)
3:      ep ← C(Q_m + 1) − 1
4:      for i ← m − 1, 1 do
5:          sp = C(Q_i) + Occ(Q_i, sp − 1) + 1
6:          ep = C(Q_i) + Occ(Q_i, ep)
7:          if ep > sp then
8:              break                          ▷ No matches, jump out
9:          end if
10:     end for
11:     return (sp, ep)   ▷ The opaque result is just a range in the BWT array
12: end function
13: function COUNT(R)
14:     if R_sp > R_ep then
15:         return 0
16:     else
17:         return R_ep − R_sp + 1
18:     end if
19: end function
```

2.3.2 Locate

As mentioned earlier Find returns a (sp, ep) range from BWT. To get the corresponding positions in text each of the BWT positions has to be converted. The inefficient way to do this would be to recursively use the LF-mapping unless it gets to the special character # and count the number of mappings that had to be done to get there. That approach would be linear in time with respect to the text size though. To make it constant in time a simple caching is added where for every $CacheEvery_{bwt}$ text position the mapping from the corresponding BWT position to that text position is saved. It is done in this way because LF-mapping is mapping BWT positions that correspond to consecutive text positions and hence it guarantees that no more than $CacheEvery_{bwt}$ mappings have to be done before finding a position that has a cached value. To implement it an additional bit vector is necessary indicating which bwt positions are cached and an array to save the actual mappings.

2.3.3 Extract

Extract is similar to Locate in that it also has a linear implementation that can be made constant with caching. To extract text from $[begin, end)$ range, it

Algorithm 2.4 Pseudo-code for Locate

```
1: function LOCATE(bwt)
2:     dist ← 0
3:     while not B_cached[bwt] do      ▷ B_cached is a bit vector indicating which positions are cached
4:         bwt ← LF(bwt)
5:         dist ← dist + 1
6:     end while
7:     return cached[Rank_1(B_cached, bwt)] + dist      ▷ Get the cached position and add the number of mappings that had to be done to get the text position
8: end function
```

could start at $BWT[1]$ (which corresponds to the last symbol of the text) and apply the LF-mapping until it gets to the *end* position and then for the next $end - start$ steps extract the symbols that LF maps to.

To make the algorithm constant in time the bwt position of every $CacheEvery_{text}$ is saved. Each entry is 32bit. The pseudo-code is in Algorithm 2.5.

Algorithm 2.5 Pseudo-code for Extract

```
1: function EXTRACT(begin, end)
2:     p ← cache[⌈end/CacheEvery_text⌉]      ▷ Get the closest cached position after end
3:     dist ← end − end mod CacheEvery_text
4:     while dist > 0 do      ▷ LF-map to the end position
5:         p ← LF(p)
6:     end while
7:     dist ← end − begin
8:     result = ϵ
9:     while dist > 0 do      ▷ LF-map and extract next begin − end characters
10:        result = BWT[p] + result      ▷ Prepend current character to the result
11:        p ← LF(p)
12:    end while
13:    return result
14: end function
```

2.3.4 Embedding text

A different approach is to embed the text in the index. That makes the *Extract* operation trivial. In case of DNA each symbol can be stored in 2 bits (as there are only 4 possible nucleotides). That means that embedding the text takes as much space as adding the cache with $CacheEvery_{text} = 16$. As the trivial *Extract* is a lot faster than the one with caching it does not make sense to use $CacheEvery_{text}$ smaller or equal to 16.

2.4 Summary

There are two index types: *SSA* and *SSAT*. The only difference is how the *Extract* option is implemented - whether it uses caching (*SSA*) or embedding (*SSAT*). Moreover, for each index type there are a few parameters controlling the balance between memory usage and execution time. We list them in Table 3.

Parameters	
n	Length of the text
CacheEvery_{bv}	How dense is the caching in bitvectors
CacheSize_{bv}	32-bit
CacheEvery_{bwt}	How dense is the caching of bwt
CacheSize_{bwt}	32-bit
CacheEvery_{text}	How dense is the caching of text (*SSA* only)
CacheSize_{text}	32-bit

Table 3: Parameters of the SSA and SSAT index

All the combinations of index type and parameter settings result in different memory usage/execution time trade-offs. The size of the index depending upon the parameters is presented in Table 4.

Size in bits	SSA	SSAT
Wavelet tree	$2 \cdot \text{n}(1+\text{CS}/\text{CE}_{bv})$	
Locate	$\text{n}(1+\text{CS}/\text{CE}_{bv}+\text{CS}/\text{CE}_{bwt})$	
Extract	$\text{n}(\text{CS}/\text{CE}_{text})$	2n
Total	$3\text{n}+\text{n}(3\text{CS}/\text{CE}_{bv}+\text{CS}/\text{CE}_{bwt}+\text{CS}/\text{CE}_{text})$	$5\text{n}+\text{n}(3\text{CS}/\text{CE}_{bv}+\text{CS}/\text{CE}_{bwt})$

Table 4: Size of the SSA and SSAT index depending upon parameters

Four different indexes have been chosen for a speed/size comparison. The parameters are shown in Table 5. They were chosen so that SSA and SSAT are compared with size similar to what bwa uses by default - 2.4 GB. In addition both a smaller SSA and a bigger SSAT index was benchmarked.

Name	Size	Type	Parameters
bmapS	1.8GB	SSA	$\text{CE}_{bv} = 128$, $\text{CE}_{bwt} = 64$, $\text{CE}_{text} = 64$
bmapB	2.4GB	SSA	$\text{CE}_{bv} = 256$, $\text{CE}_{bwt} = 16$, $\text{CE}_{text} = 32$
bmapT	2.4GB	SSAT	$\text{CE}_{bv} = 256$, $\text{CE}_{bwt} = 32$
bmapL	2.9GB	SSAT	$\text{CE}_{bv} = 128$, $\text{CE}_{bwt} = 16$

Table 5: Parameters of the indexes chosen for comparison

3 User manual

Bmap is a C++ project which can be built with CMake.

3.1 Required dependencies

- boost
- gtest - `http://code.google.com/p/googletest/downloads/list`
- bamtools - `https://github.com/pezmaster31/bamtools`
- libdivsufsort - `http://code.google.com/p/libdivsufsort/downloads/list`

3.2 Usage

3.2.1 Buildng an index

```
Usage: indexer type rebwt fasta-file bwt-file index-file
```

- type - see Table 6
- rebwt - 0/1 - indicates whether BWT has to be computed. If 0 then BWT from the bwt-file is used. If 1 then BWT is computed and saved to bwt-file before building the index
- fasta-file - file in FASTA format containing reference sequences
- bwt-file - file containing BWT (see above)
- index-file - output file where index is saved

3.2.2 Mapping

```
Usage: mapper [options] reads_fastq output_bam

Main options:
--help                    produce help message
-i [ --index ] arg        path to the index
-r [ --read-size ] arg    size of the reads
-e [ --errors ] arg (=0) maximum number of errors
-l [ --limit ] arg (=-1) process only that many reads
Advanced options:
-c [ --cutoff ] arg (=100) cutoff
-s [ --substrings ] arg (=0)   number of substrings to divide into
```

Type	$CE_{bv}/64$	CE_{bwt}	CE_{text}
ssa_64_1_8_32	1	8	32
ssa_64_2_8_32	2	8	32
ssa_64_4_8_32	4	8	32
ssa_64_1_8_64	1	8	64
ssa_64_2_8_64	2	8	64
ssa_64_4_8_64	4	8	64
ssa_64_1_16_32	1	16	32
ssa_64_2_16_32	2	16	32
ssa_64_4_16_32	4	16	32
ssa_64_1_16_64	1	16	64
ssa_64_2_16_64	2	16	64
ssa_64_4_16_64	4	16	64
ssa_64_1_32_32	1	32	32
ssa_64_2_32_32	2	32	32
ssa_64_4_32_32	4	32	32
ssa_64_1_32_64	1	32	64
ssa_64_2_32_64	2	32	64
ssa_64_4_32_64	4	32	64
ssa_64_1_64_32	1	64	32
ssa_64_2_64_32	2	64	32
ssa_64_4_64_32	4	64	32
ssa_64_1_64_64	1	64	64
ssa_64_2_64_64	2	64	64
ssa_64_4_64_64	4	64	64
ssat_64_1_8	1	8	
ssat_64_2_8	2	8	
ssat_64_4_8	4	8	
ssat_64_1_16	1	16	
ssat_64_2_16	2	16	
ssat_64_4_16	4	16	
ssat_64_1_32	1	32	
ssat_64_2_32	2	32	
ssat_64_4_32	4	32	
ssat_64_1_64	1	64	
ssat_64_2_64	2	64	
ssat_64_4_64	4	64	

Table 6: Parameters in supported index types

References

[1] Pizza&chili. `http://pizzachili.dcc.uchile.cl`.

[2] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[3] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice! *CoRR*, abs/0712.3360, 2007.

[4] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52:552–581, July 2005.

[5] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05)*, pages 27–38, Greece, 2005. CTI Press and Ellinika Grammata.

[6] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[7] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39, April 2007.