Super-Linear Indices for Approximate Dictionary Searching

Leonid Boytsov
leo@boytsov.info

10 August 2012

Approximate String Searching

Dictionary version of the problem:

- Set of strings called dictionary *W*
- Pattern string *p*
- Threshold distance *k*
- Find all strings from *W* within distance *k* from the search pattern *p*
- Focus on the **Levenshtein** distance

Super-Linear Indices

- Indexing of deletion neighborhoods allows one to achieve very short times
- Index size grows exponentially with *k*
- One needs methods to decrease space requirements

String
k-Neighborhoods

Simple idea from 50-60s:

- Generate strings *p′* by *k* insertions, deletions, substitutions, applied to pattern *p*.

- Search *p*' in *W*.

k-Neighborhood Types

- Full neighborhood
- Partial and wildcard neighborhood
    - Deletion neighborhood
    - Reduced-alphabet neighborhood
- Condensed neighborhoods (only feasible to search for short substrings in a text over a small alphabet, e.g. a DNA sequence)

Full *k*-Neighborhoods

Example: p=**`find`** k**=**1

1. The original string **`find`**
2. 4 strings obtained by one deletion
3. 5×26 strings obtained by one insertion
4. 4×25 strings by one substitution
5. Ignore few duplicates

Problem: full neighborhood is huge (in our case 231 strings):

$$O(|p|^k|\Sigma|^k)$$

Wildcard *k*-Neighborhood (Approach 1)

`?` is a wildcard that matches any symbol.

1-Neighborhood has only 13 strings:

1. **`find`**
2. **`ind, fnd, fid, fin`**
3. **`?find, f?ind, fi?nd, fin?d, find?`**
4. **`?ind, f?nd, fi?d, fin?`**

Problem: how to search for wildcard patterns efficiently?

Approach 1: Focus of this Presentation

- A solution to efficient retrieval of strings with wildcards symbols `?` is inspired by the dynamic programming (DP) algorithm
- DP algorithm doesn't compute distance directly
- Instead it computes the cost of an optimal alignment

Optimal Alignment

F	I	N	D	
G	R	I	N	D

Optimal Alignment

F		I	N	D
G	R	I	N	D

Optimal Alignment

~~F~~		I	N	D
~~G~~	~~R~~	I	N	D

Positions of Deleted Chars: Adjusted by Preceding Dels

Multi-set *A* for the 1st word:

$(1) \rightarrow (1)$

Multi-set *B* for the 2d word:

$(1,2) \rightarrow (1,1)$

Positions of Deleted Chars Define Levenshtein Distance

Edit distance being at most *k* is equiv.:

$$|\boldsymbol{A}| + |\boldsymbol{B}| - |\boldsymbol{A} \cap \boldsymbol{B}| \leq k$$

In our example:

Levenshtein(**`find`**, **`grind`**) = 2 + 1−1 = 2

Straightforward Indexing

- Generate **residual strings** (obtained by up to *k* deletions)
- Keep multi-sets that represent deleted chars
- Index words and multi-sets using residual strings as keys (via a regular hash index)

Regular Hash (*k*=1)

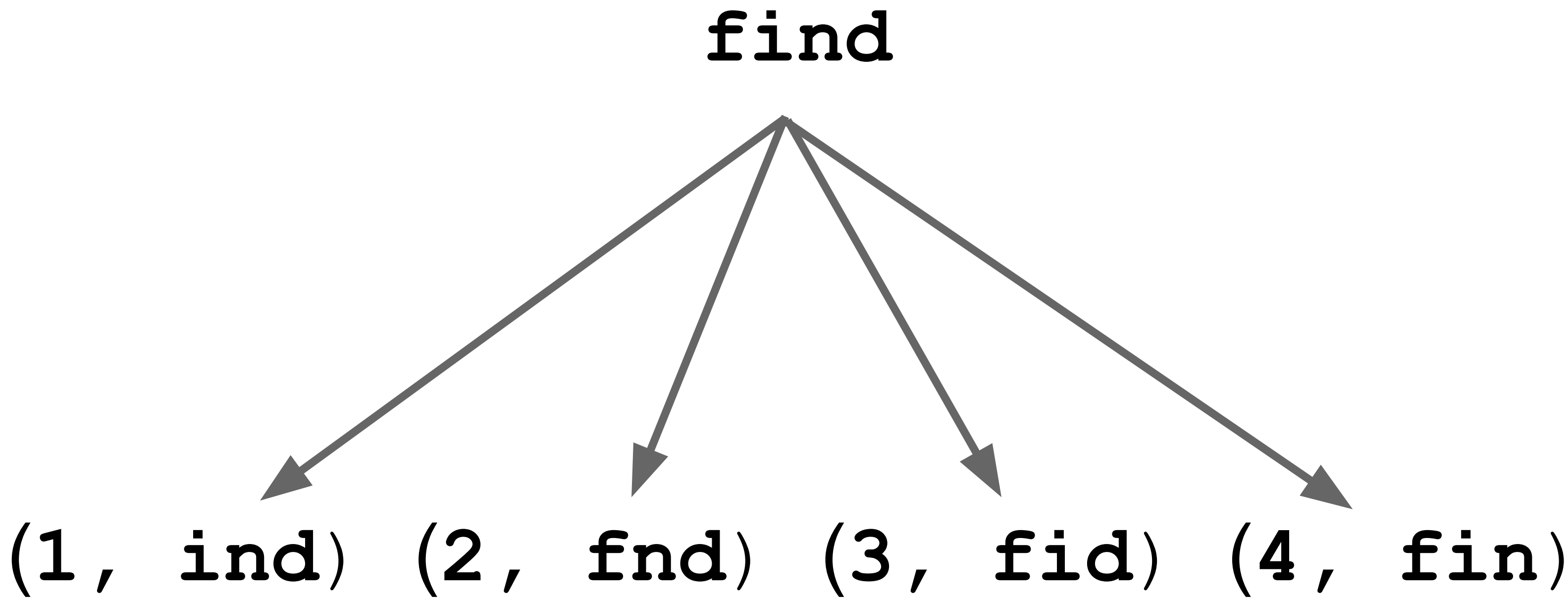
Dictionary: **`mind, bind`**

0	`ind:  (1, mind) (1, bind)` `mnd:  (2, mind)`
1	`mid:  (3, mind)` `min:  (4, mind)`
2	`bnd:  (2, bind)` `bid:  (3, bind)`
3	`bin:  (4, bind)`

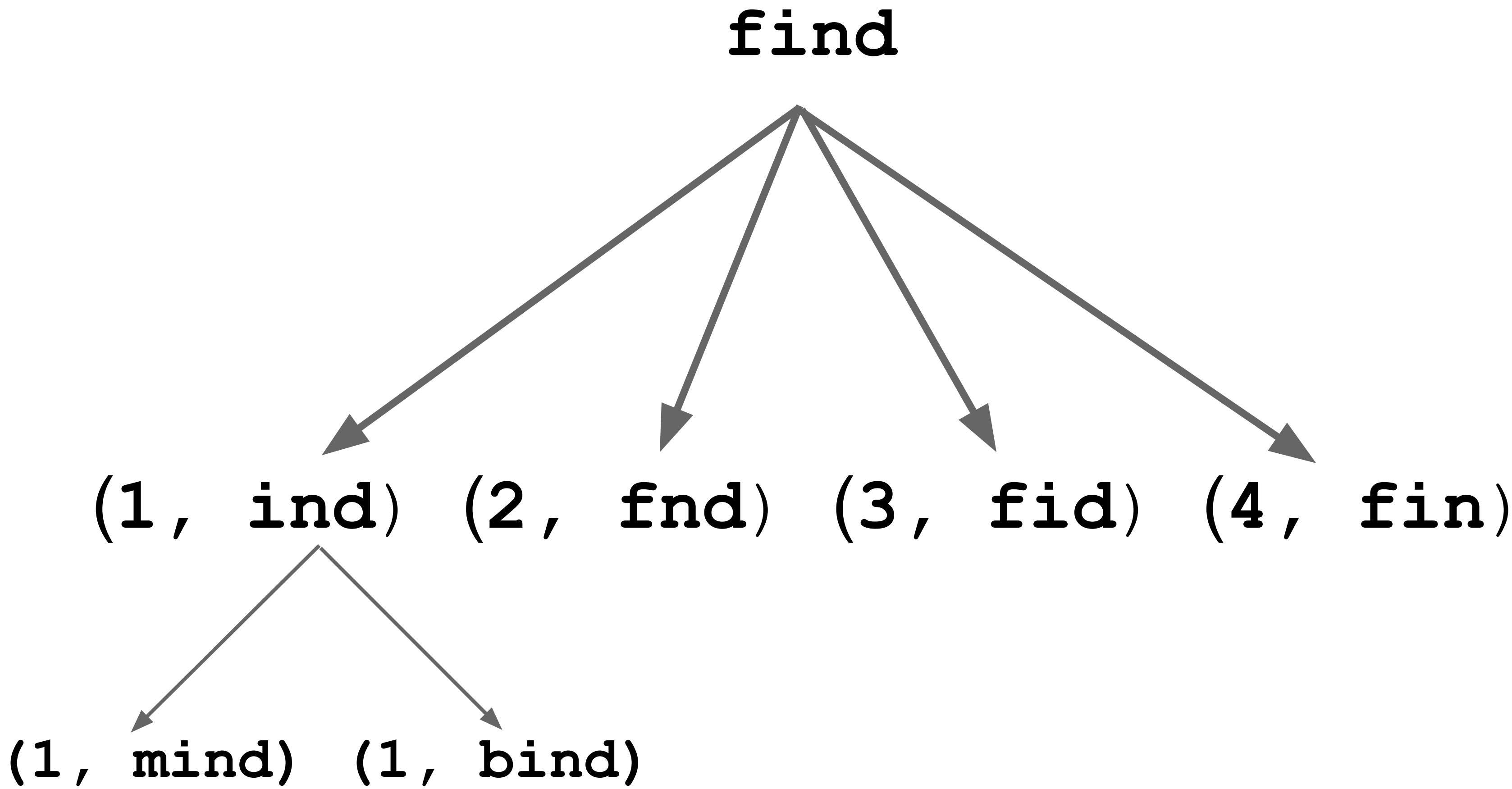
Somewhat Similar to Locality Sensitive Hashing

- There are several hash functions $h_i()$
- Strings s are indexed based on hash values $h_i(s)$

Retrieval Algorithm (k=1)



Retrieval Algorithm (*k*=1)



Checking Levenshtein Distance (k=1)

$A = (1)$
$B = (1)$

Using the formula (slide 14) we obtain that:

$|A| + |B| - |A \cap B| = 1 + 1 - 1 = 1 \leq k$

Advantages

- Only $O(|p|^k)$ buckets are tested
- Buckets are scanned sequentially
- No need to compute edit distance
- Hence, method is very fast

The Disadvantage

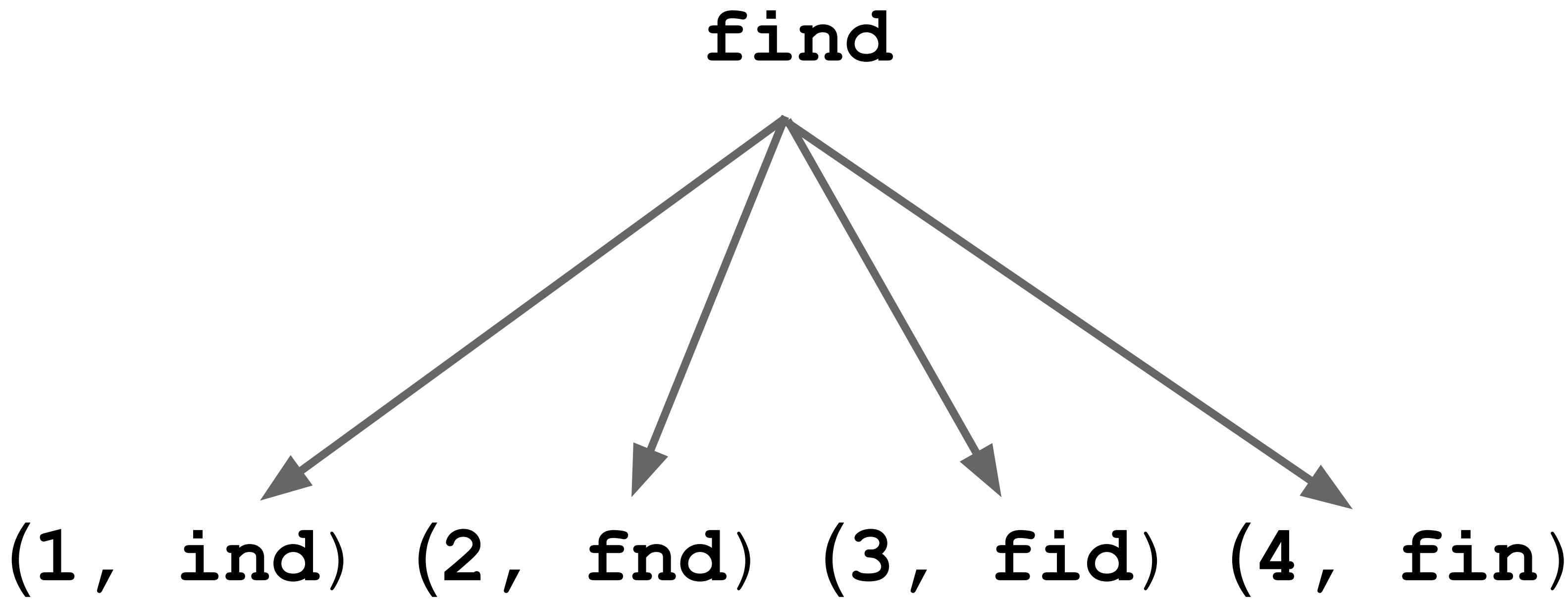
- Index size is huge: $O(M^k)$, where M is a max string length
- Should be precomputed for all k

Compact Index (*k*=1) Based on Perfect Hashing

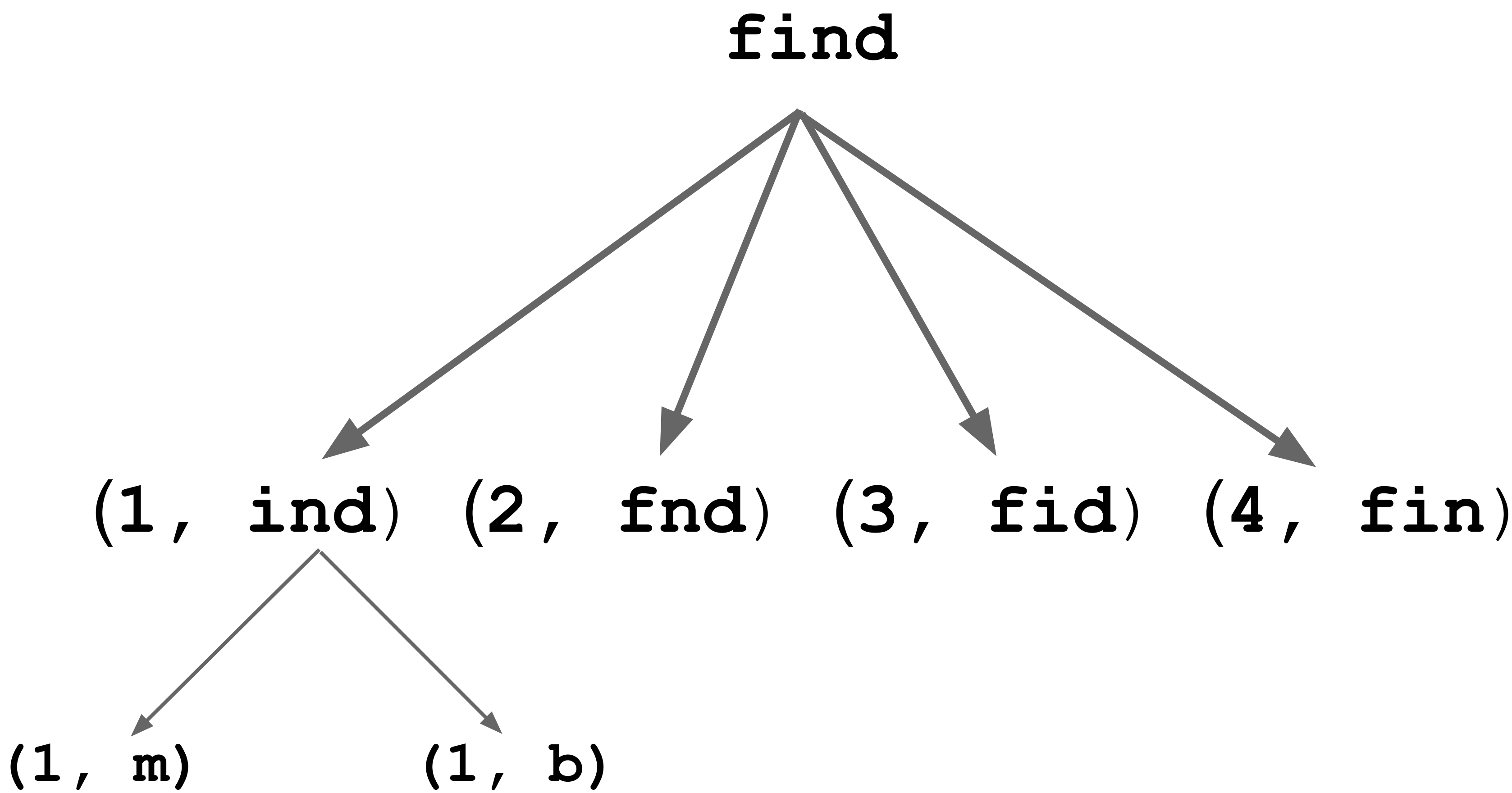
Dictionary: **`mind, bind`**

0	**`ind`**	**`(1, m) (1, b)`**
1	**`mnd`**	**`(2, i)`**
2	**`mid`**	**`(3, n)`**
3	**`min`**	**`(4, d)`**
4	**`bnd`**	**`(2, i)`**
5	**`bid`**	**`(3, n)`**
6	**`bin`**	**`(4, d)`**

Retrieval Algorithm (*k*=1)



Retrieval Algorithm (*k*=1)



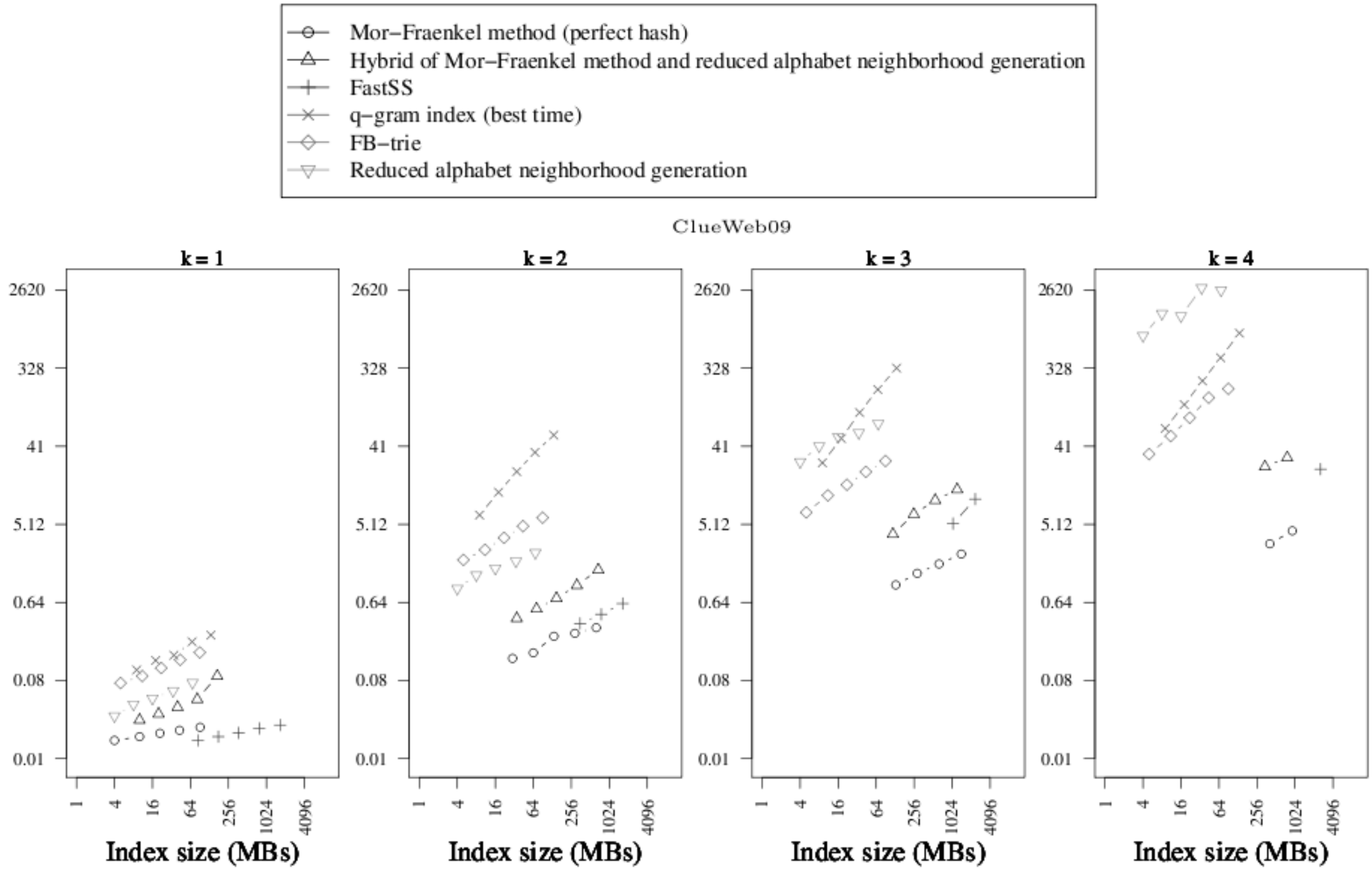
Do These Records Represent Dictionary Strings?

- We have to restore original strings by re-insertion of characters **`m`** and **`b`**
- Check if reconstructed strings belong to *W*
- Do it only **once** for each bucket!

Evaluation Results: Benchmarks

1. Compact Mor-Fraenkel index (this paper)
2. Straightforward Mor-Fraenkel index (FastSS by Bocek et al [2007])
3. Reduced alphabet neighborhood generation (Approach 2)
4. Hybrid of compact Mor-Fraenkel index and reduced alphabet neighborhood generation
5. Q-gram indices [Behm et al 2010, Boytsov 2011]
6. FB-trie: a pair of tries built over original and reversed strings [Mihov and Schulz 2004].

Evaluation Results



Summary of Results

- MF-methods work best for natural language data, where they outperform other methods
- Index sizes are 4-8 times smaller with perfect hashing
- Retrieval times are comparable (for regular and perfect hashing)
- Associative Searches Involve a Tradeoff (larger index or longer retrieval times)

Compact Mor-Fraenkel Index: Is it Straightforward?

- **No.** One needs to generate perfect hash for **dozens or even hundreds millions** of strings

- The library CMPH [Botelho 2008] can handle this task, but not always: for ClueWeb09 strings it fails at around 100M mark.

- Two-level hashing scheme fixes this problem

Extensions to generic spaces?

- Perhaps, we can use it for vector spaces.
- Create a VA-file [Webber et al 1998] and index subvectors of approximated vectors.
- It is possible that this approach has already been implemented by somebody. If not, it is worth trying.

Thank you!

Questions are welcome!

Appendix

Neighborhood Generation Search Complexities

Full neighborhood	$O(\|p\|^k \|\Sigma\|^k)$
Reduced-alphabet neighborhood	$O(\|p\|^k \|\sigma\|^k (1+\alpha))$, σ is reduced alphabet
Deletion neighborhood	$O(\|p\|^k (1+\beta))$

- α and β depend on the dictionary "density"
- In the worst case, the last 2 methods can be worse than full neighborhood generation.

Historical Notes

- Indexing of deletion neighborhoods was originally proposed by Mor and Fraenkel for k=1 in 1982. The generalization for k>1 was independently described by Bocek et al. [2007] and Boytsov [2011].

- The scheme based on perfect hashing was described by Belazzougui [2009] for k=1. To my best knoweledge, it has been never tested before.

Optimal Alignment: Interpretation with Wildcards

?	?	I	N	D
G	R	I	N	D

Wildcard *k*-Neighborhood (Approach 2)

`0` denotes **`[a-m]`**, **`1`** denotes **`[n-z]`**.

Hash function *h(s)* **reduces** original strings into sequences of ones and zeros:

h(**`find`**) **`= 0010`**

The full 1-neighborhood of **`0010`** has only 14 unique elements.

Problem: how to index and search efficiently?

References

1. Behm, A., Vernica, R., Alsubaiee, S., Ji, S., Lu, J., Jin, L., Lu, Y., Li, C.: UCI Flamingo Package 4.0 (2010)

2. Belazzougui, D.: Faster and space-optimal edit distance 1 dictionary. Volume 5577 of Lecture Notes in Computer Science. (2009) 154–167

3. Bocek, T., Hunt, E., Stiller, B.: Fast similarity search in large dictionaries (2007)

4. Botelho, F.C.: Near-Optimal Space Perfect Hashing Algorithms. PhD thesis (2008)

5. Boytsov, L.: Indexing methods for approximate dictionary searching: Comparative analysis. J. Exp. Algorithmics 16 (May 2011)

6. Mihov, S., Schulz, K.U.: Fast approximate string search in large dictionaries. Computational Linguistics, 30(4) (2004) 451–47

7. Mor, M., Fraenkel, A.S.: A hash code method for detecting and correcting spelling errors. Communications of the ACM 25(12) (1982) 935–938

8. R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for Similarity Search Methods in High Dimensional Spaces. In Proceedings of the 24th International Conference on Very Large Data Bases (VLDB), 1998, pp. 194-205