Spell-checking, query correction, wildcards

Stanislav Protasov

Agenda

- Wildcards and regexp support
  - Wildcard types
  - Permuterm
  - Regexp support
- Spell-checking
  - Isolated words
  - Context-dependent approach
  - Soundex

Wildcards

Why do we need wildcards?

- uncertain of the spelling of a query term
- aware of multiple variants of spelling (colou?r)
- unsure about part of speech and stemming
- foreign words and spelling (**Universit**y street vs **Universit**etskaya ulitsa)

Syntax of wildcards

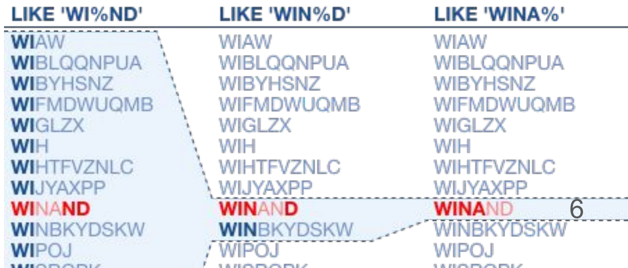
- *? - wildcards with joker symbols
  - *Trailing* wildcard query (**Mos***) — handled with **search trees**
  - *Leading* wildcard queries (***sity**) — handled with **reversed search** trees (**ytis...**)
  - Can we handle **Mos*ow** query? **M*S*K**?
- SQL
  - `column LIKE` **“%ab_d”**
- Regexp
  - `[a-zA-Z][a-zA-Z0-9]{0-30}`

How they do it in databases

“*[The SQL `LIKE` operator very often causes unexpected performance]*”

Only the part before the first wild card serves as an access predicate. The remaining characters do not narrow the scanned index range—non-matching entries are just left out of the result.

PostgreSQL: The optimizer can also use a **B-tree index** for queries involving the pattern matching operators `LIKE` and ~ if the pattern is a constant and is anchored to the beginning of the string — for example, `col LIKE 'foo%'` or `col ~ '^foo'`, but not ~~`col LIKE '%bar'`~~



Permuterm index

1. Add $ to the end of the word: `hello` → `hello$`
2. Compose all rotations of this word
   a. `hello$, ello$h, llo$he, lo$hel, o$hell, $hello`
3. Build a search tree (B-tree, trie) on this extended vocabulary
4. For a particular query run the following search algorithm (star * means *exhaustive search* of a subtree, with filtering maybe):
   a. **`X`** `lookup on` **`X$`**
   b. **`X*`** `lookup on` **`$X*`**
   c. **`*X`** `lookup on` **`X$*`**
   d. **`*X*`** `lookup on` **`X*`**
   e. **`X*Y`** `lookup on` **`Y$X*`**
   f. **`X*Y*Z – ?`**

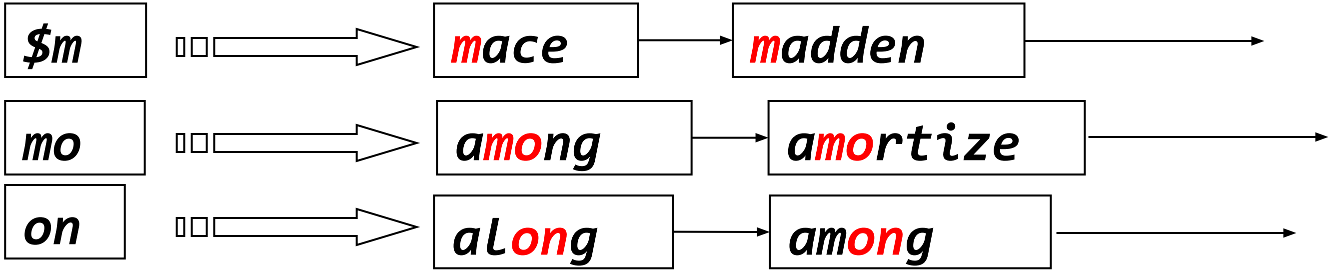
hel*o?

What’s wrong with permuterm?

- Not handling **multiple** joker symbols
- **4-10-times increases** vocabulary size

K-gram index (q-grams)

1. Mark word start/end with $
2. 3-grams of `` `castle` `` are: **`$ca, cas, ast, stl, tle, le$`**.
3. Maintain a second inverted index from k-grams to dictionary terms that match each k-gram (here k=2)



4. Convert wildcards into **boolean queries**

`re*ve → $re & ve$.`

5. *Comment*: as words, not all k-grams of fixed k are equally useful (**zzz** vs **the**).

Multigrams (V-grams)

Each k-gram can have specific ***k***: “**`<a hre`**” and “**`.mp3`**”

Selectivity of x-gram is the fraction of data units which contain at least one occurrence of the gram. **Filter factor**.

```
FF = 1 - selectivity(x)
```

Take only grams with high FF.

Order query parts by filter factor.

*Comment: hard to maintain **online***

Algorithm 3.1 Multigram index
Input: database
Output: index: multigram index
Procedure
[1] $k = 1$, expand $= \{\cdot\}$ // $\cdot$ is a zero-length string
[2] While (expand is not empty)
[3] k-grams := all k-grams in database whose $(k$-$1)$-prefix $\in$ expand
[4] expand := $\{\}$
[5] For each gram x in k-grams
[6] If $\text{sel}(x) \leq c$ Then // check selectivity
[7] insert(x, index) // the gram is useful
[8] Else
[9] expand := expand $\cup \{x\}$
[10] $k := k + 1$

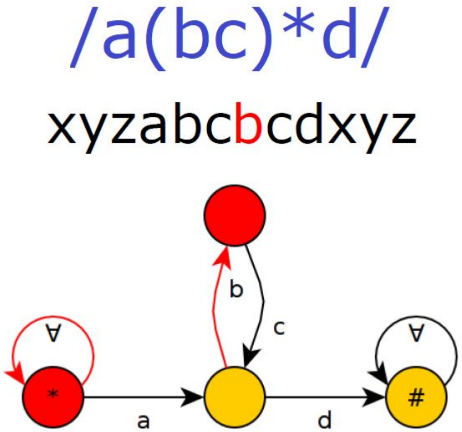
Regexp support

[Index support for regular expression search]()

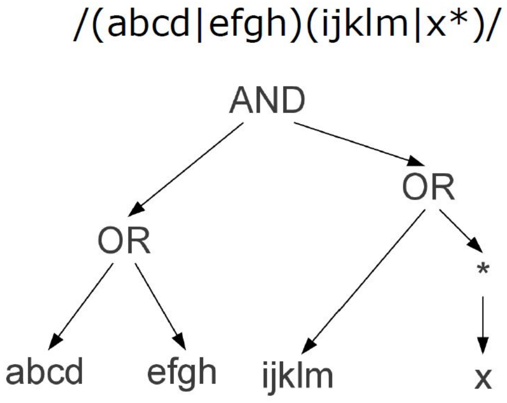
expressing same class of “languages” as finite automata.

General idea is similar:

```
/[ab]cde/ => (acd OR bcd) AND cde
```

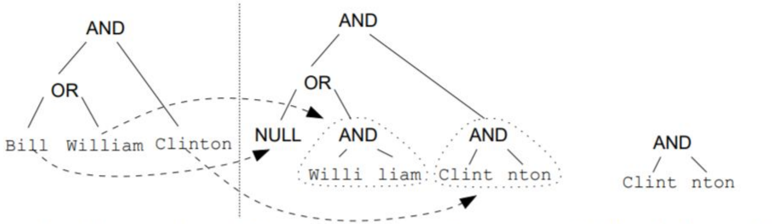


Regexp support methods: FREE



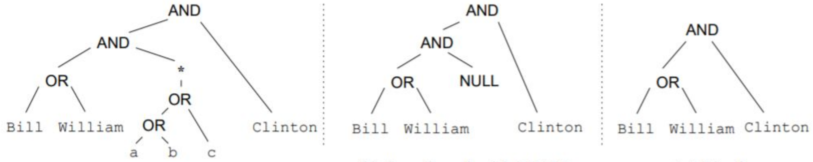
FREE (2002):

1. Extract tree of continuous string fraction from regex.
   - * = NULL, NULL “eats” parent OR
   - AND eats child NULL
2. Transform those continuous fractions to **multigrams**
3. Use inverted index on multigrams for query evaluation



(a) Generation of physical access plan

(b) Final physical access plan



(a) Original

(b) * replaced with NULL

(c) Final

Regexp support methods: GCS

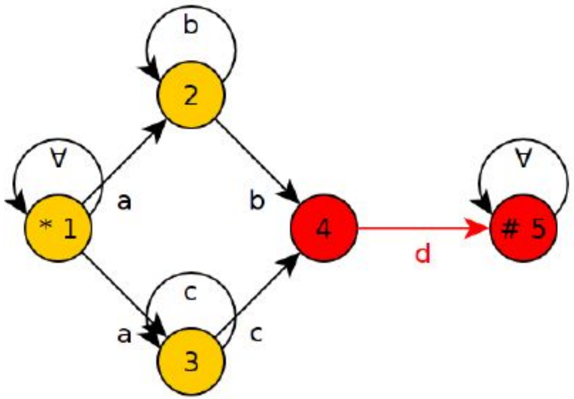
Regular Expression Matching with a Trigram Index (Google Code Search, 2006)

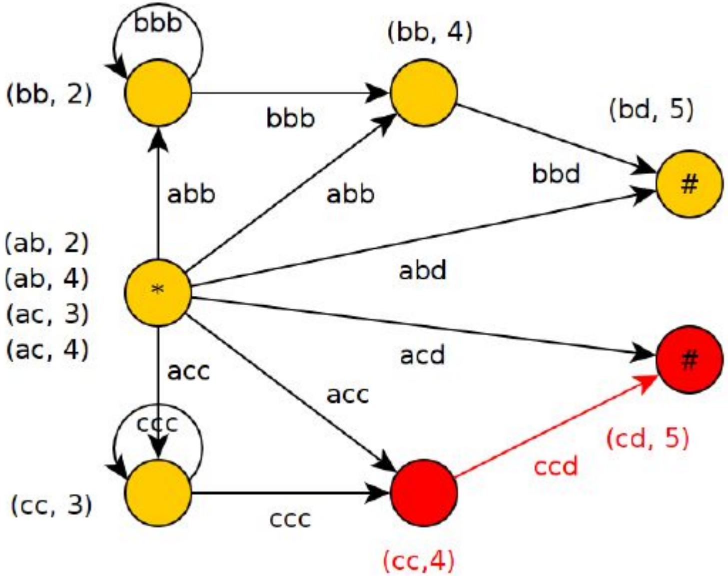
- Get 5 characteristics about each part of regex: *emptyable*, *exact*, *prefix*, *suffix*, *match*.
- Recursively union them (with possible simplification)
- Use inverted index of trigrams for query evaluation (similar to pg_trgm)

```
Original regex: /a(bc)+d/
a: {exact: a}
bc: {exact: bc}
d: {exact: d}
(bc)+: {prefix:bc, suffix: bc}
a(bc)+: {prefix:abc, suffix:bc}
a(bc)+d: {prefix:abc, suffix:bcd} == abc AND bcd
```

Regexp support methods: [automaton transformation](#) (2012)

1. Procedure for automaton transformation into corresponding graph
2. Procedure to simplify a graph
3. Collect path matrix and convert to binary query





Query correction

Spell correction

- Two principal uses
  - Correcting **document**(s) being indexed
  - Correcting user **queries** to retrieve “right” answers
- Two principles:
  - proximity-base (take closest)
  - probability-base (take most frequent)

Spell correction

- Two main flavors:
  - Isolated word
    - Check each word on its own for misspelling
    - Will not catch typos resulting in correctly spelled words e.g., ***from*** → ***form***
  - Context-sensitive
    - Look at surrounding words,
    - e.g., ***I flew <u>form</u> Heathrow to Narita.***
- Two approaches
  - Fix the query and retrieve documents
  - Suggest corrected query option[s]

Document correction

Especially needed for OCR'ed documents

- Correction algorithms are tuned for this: **rn/m**
- Can use domain-specific knowledge
  - E.g., OCR can confuse O and D more often ...
  - … than it would confuse O and I (adjacent on the QWERTY keyboard, so more likely interchanged in typing).

But also: web pages and even printed material have typos

Goal: the dictionary contains fewer misspellings

Isolated words correction

Isolated word correction

Fundamental premise – **there is a lexicon** from which the correct spellings come

Two basic choices for this

- A **standard** lexicon such as
  - Webster’s English Dictionary
  - An “industry-specific” lexicon – hand-maintained
- The lexicon of the **indexed corpus**
  - E.g., all words on the web
  - All names, acronyms etc.
  - (Including the mis-spellings)

Isolated word correction

Given a lexicon and a character sequence Q, return the words in the lexicon closest to Q

What's "closest"?

- Edit distance (Levenshtein distance) and LCS (longest common subsequence)
- Weighted edit distance
- *n*-gram overlap

Edit distance

Given two strings S_1 and S_2, the minimum number of operations to convert one to the other

Operations are typically character-level
- Insert, Delete, Replace, (Transposition)

E.g., the edit distance from ***dof*** to ***dog*** is 1
- From ***cat*** to ***act*** is 2 (Just 1 with transpose.)
- from ***cat*** to ***dog*** is 3.

Generally found by dynamic programming.

$$D(i,j) = \begin{cases} 0, & i=0,\ j=0 \\ i, & j=0,\ i>0 \\ j, & i=0,\ j>0 \\ \min\{ & \\ \quad D(i,j-1)+1, & \\ \quad D(i-1,j)+1, & j>0,\ i>0 \\ \quad D(i-1,j-1)+\mathrm{m}(S_1[i],S_2[j]) & \\ \} & \end{cases}$$

Weighted edit distance

As above, but the weight of an operation depends on the character(s) involved

1. Meant to capture OCR or **keyboard errors**
   Example: ***m*** more likely to be mis-typed as ***n*** than as ***q***
2. Therefore, replacing ***m*** by ***n*** is a smaller edit distance than by ***q***
3. This may be formulated as a probability model

Requires weight matrix as input

Edit distance to all dictionary terms?!

Given a (mis-spelled) query – do we compute its edit distance to every dictionary term?

- Expensive and slow
- Alternative?

How do we cut the set of candidate dictionary terms?

One possibility is to use *n*-gram overlap for this.

This can also be used by itself for spelling correction.

n-gram overlap

Enumerate all the *n*-grams in the query string as well as in the lexicon

Use the *n*-gram index to retrieve all lexicon terms matching any of the query *n*-grams

Threshold by number of matching *n*-grams

- Variants – weight by keyboard layout, etc.

Jaccard coefficient (IoU)

A commonly-used measure of overlap

Let *X* and *Y* be two sets; then IoU is $|X \cap Y| / |X \cup Y|$

Equals 1 when *X* and *Y* have the same elements and zero when they are disjoint

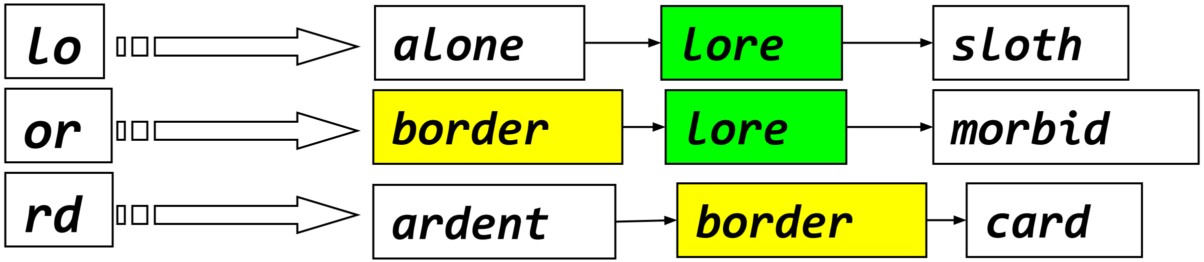
X and *Y* don't have to be of the same size

Always assigns a number between 0 and 1

- Now threshold to decide if you have a match
- E.g., if IoU > 0.8, declare a match

Matching trigrams

Consider the query ***lord*** – we wish to identify words matching 2 of its 3 bigrams (***lo, or, rd***)



Standard postings "merge" will enumerate …

Adapt this to using Jaccard (or another) measure.

Context sensitive correction

Context-sensitive spell correction

Text: ***I flew <u>from</u> Heathrow to Narita.***

Consider the phrase query ***"flew <u>form</u> Heathrow"***

We'd like to respond:

Did you mean "***flew from Heathrow***"?

because no docs matched the query phrase.

Context-sensitive correction

1. Retrieve **dictionary terms close** (in weighted edit distance) **to** each **query term**

Now try all possible resulting phrases with **one word "fixed" at a time**

- ***flew from heathrow***
- ***fled form heathrow***
- ***flea form heathrow***

Hit-based spelling correction: Suggest the alternative that has lots of hits.

2. **Biword statistical approach**

Break phrase query into a **conjunction of biwords**.
Look for **biwords** that need **only one term corrected**.
Enumerate only phrases containing "common" biwords.

General issues in spell correction

We enumerate multiple alternatives for “Did you mean?”

Need to figure out which to present to the user
- The alternative hitting most docs
- Query log analysis

More generally, rank alternatives probabilistically

$$\text{argmax}_{corr}\ P(corr \mid query)$$

- From Bayes rule, this is equivalent to

$$\text{argmax}_{corr}\ P(query \mid corr) * P(corr)$$

Noisy channel

Language model

Soundex: phonetic correction

Soundex motivation

Class of heuristics to expand a query into phonetic equivalents

- Language specific – mainly for names
- E.g., ***chebyshev*** → ***tchebycheff***

Invented for the U.S. census … in 1918

Soundex – typical algorithm

1. Turn every token to be indexed into a **4-character reduced form**

2. Do the same with **all query terms**

3. Build and **search** an **index** on the reduced forms (when the query calls for a soundex match)

Soundex — part 1

1. Retain the first letter of the word.
2. Change all occurrences of the following letters to `0`:
   `'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.`
3. Change (similar) letters to digits as follows:
   `B, F, P, V` → `1`
   `C, G, J, K, Q, S, X, Z` → `2`
   `D, T` → `3`
   `L` → `4`
   `M, N` → `5`
   `R` → `6`

Soundex — part 2

4. **Remove** all pairs of **consecutive digits**.
5. **Remove** all **zeros** from the resulting string.
6. **Pad** the resulting string with **trailing zeros** and return the first four positions, which will be of the form
   `<uppercase letter> <digit> <digit> <digit>`.

E.g., ***Herman*** becomes H655.

H e r m a n n ?

Soundex and improvements

Even used by all databases, it is not very efficient in typo fixing. High recall, but very low precision.

Other phonometric algorithms exist:

Phonetic string matching (1996) =
= editorial distance + phoneme representation

Senc iu fo itenshn!