SIGMOD 2013 Programing Contest
Implementation of Fast Matching System for Dynamic Query

Team mofumofu (Ryosuke Okuta, Tomoki Komatsu, Takashi Katsura, Yuhei Otomo)
Graduate School of Information Sciences, Tohoku University, Japan

CONTEST OVERVIEW: The general idea is to filter a stream of documents using a dynamic set of exact and approximate continuous keyword match. the goal is to maximize the throughput with which documents are disseminated to active queries. Whenever a new document arrives, the system must quickly determine all queries satisfied by this document.

Task Details

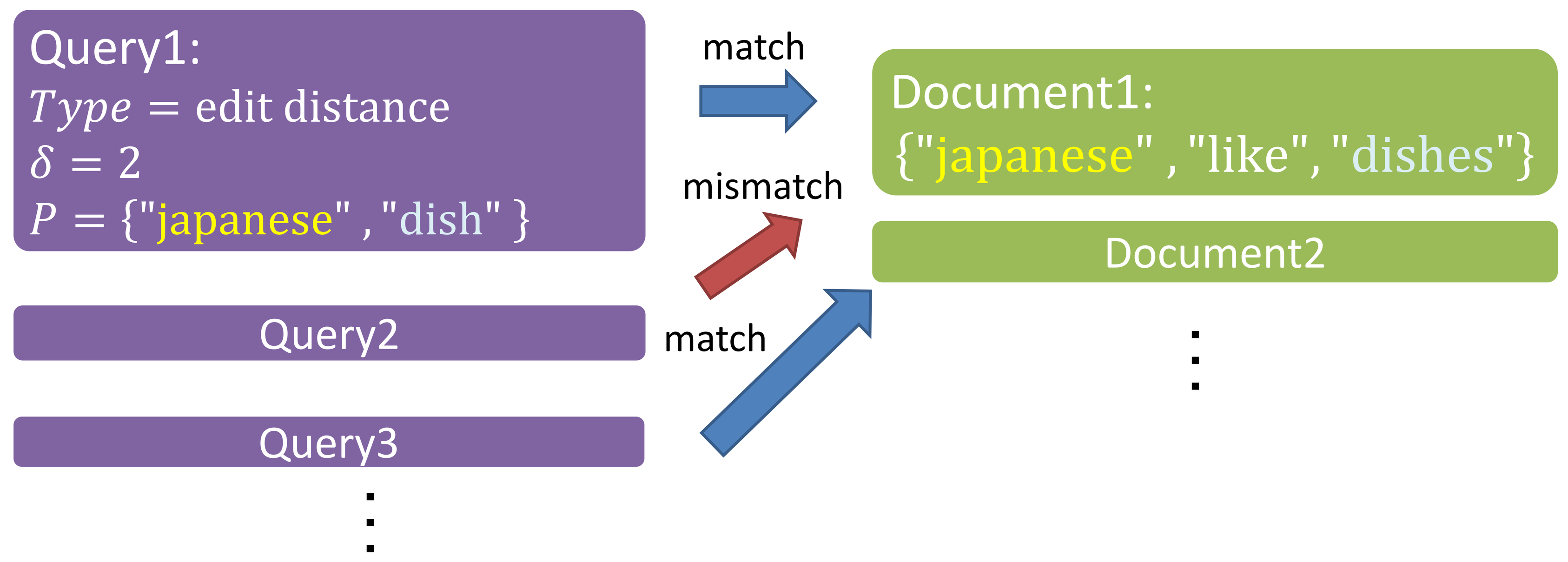
Given some queries $Q = (Type, \delta, P)$ and documents D, output sets of queries that match each document. Our task is developing the system that computes the above problem as quickly as possible.

A query consists of the required matching type $Type$, the matching distance threshold $\delta \leq 3$ and a set of words $P = \{p_1, p_2, \dots, p_n\}$. There are three types of matching : exact, hamming distance and edit distance. If $Type$ is exact matching, δ is 0. δ is at most 3. A document D is a set of words $t_1, t_2, \dots, t_m$.

Q match D if there is t_j such that the distance between p_i and t_j with respect to $Type$ is not greater than δ for any $1 \leq i \leq n$.

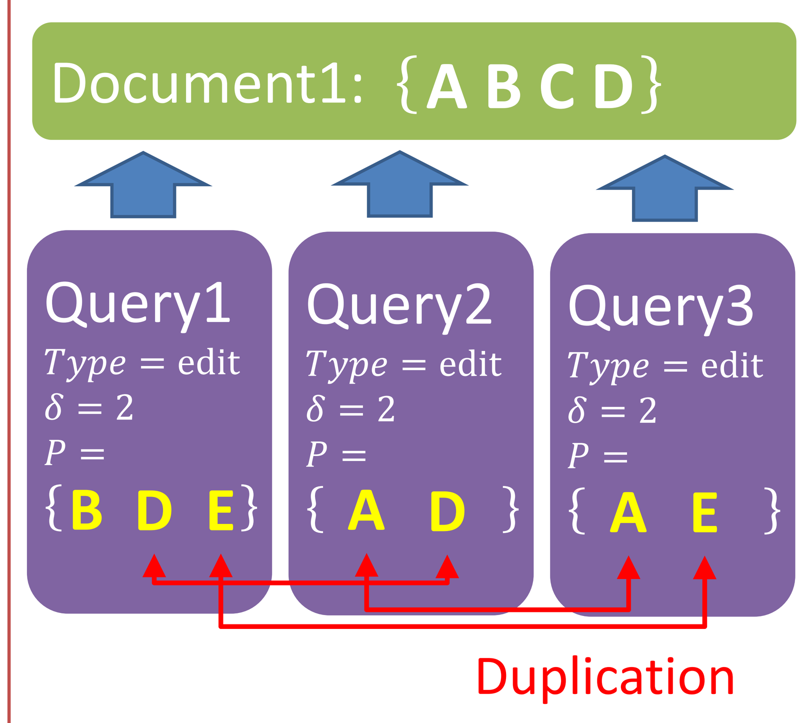
Matching Types

1. Exact Match
   "poke" ="coke" ✗
   "pork" ="pork" ✓
2. Hamming distance
   "angel" / "angle" → 2 "abc" / "ab" → ∞ no match length
3. Edit distance
   "spread" –deletion→ "sp ead" –substitution→ "speed" –insertion→ "speedy"
   distance=3



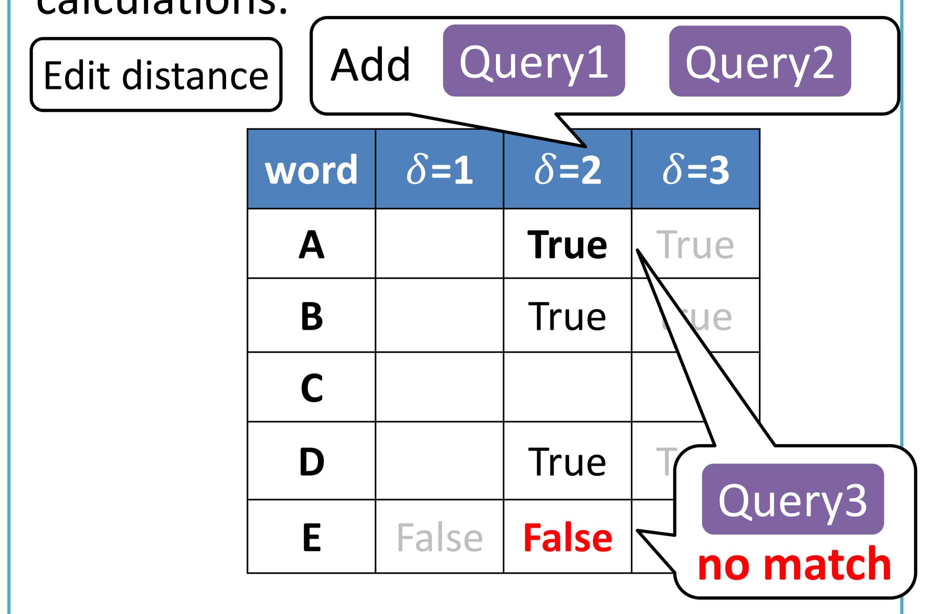
Memorizing calculation results

Problem Among queries in the contest, there are often duplications of words.



In naïve method, calculations of distances for duplicated words are performed repeatedly.

Approach We performed speed-up of matching by memorizing results of calculations.



Referring to the above table, we may not calculate distances for duplicated words. Moreover, calculations can be reduced by processing queries in order of increasing number of words.

Other implementation techniques

- Use special data structures to deal with string (8, 16, 32 bytes)
- Use bit operations and SIMD operations
- Minimization of the number of calls to "malloc"
- Use Bit-Vector algorithm for edit distance [1]
- Use B-tree for Map and Set [2]

[1] H. Hyyro. "A Bit-Vector Algorithm for Computing Levenshtein and Damerau Edit Distances." Nordic Journal of Computing, 10:1-11, 2003.
[2] http://code.google.com/p/cpp-btree/

Filtering by occurrence bit vector

Problem Calculating edit distance is high cost.

Approach We can filter words out by using the length and the number of each character of each word.

For string s, the occurrence bit vector $OV(s)$ is a bit vector obtained by turning bit corresponding to each character in s into 1. For string s_1 and s_2, let $DOC(s_1, s_2)$ be the difference of occurring characters between s_1 and s_2, i.e., the number of 1-bit in $XOR(OV(s_1), OV(s_2))$.

	a	b	c	d	e	…	l	m	n	o	p	q	r	s	t	u	…	y	z
OV("tuple")	0	0	0	0	1	…	1	0	0	0	1	0	0	0	1	1	…	0	0
OV("apply")	1	0	0	0	0	…	1	0	0	0	1	0	0	0	0	0	…	1	0
XOR	1	0	0	0	1	…	0	0	0	0	0	0	0	0	1	1	…	1	0

Number of 1-bit = 5 → DOC("tuple", "apply")=5

For two strings s_1 and s_2, in order to $s_1 = s_2$, it is necessary that $|s_1| = |s_2|$ and $DOC(s_1, s_2) = 0$. Let $WeakED(s_1, s_2)$ be the minimum number of operations that transform s_1 to s'_1 such that $|s'_1| = |s_2|$and $DOC(s'_1, s_2) = 0$. Let $ED(s_1, s_2)$ be the exact edit distance between s_1 and s_2. Now, following theorem holds.

Theorem
For any two string s_1 and s_2, $WeakED(s_1, s_2) \leq ED(s_1, s_2)$.

We can prove the above theorem easily by contradiction. For matching threshold δ, we must determine $ED(s_1, s_2) \leq \delta$. From above theorem, $WeakED(s_1, s_2) \leq ED(s_1, s_2) \leq \delta$ holds. Thus, **if $WeakED(s_1, s_2) > \delta$, then we see that $ED(s_1, s_2) \nleq \delta$ immediately**. $WeakED(s_1, s_2)$ is obtained as follows:

$$WeakED(s_1, s_2) = \underbrace{abs(|s_1| - |s_2|)}_{①} + \underbrace{\max\left(\frac{DOC(s_1, s_2) - abs(|s_1| - |s_2|)}{2}, 0\right)}_{②}$$

① is the cost for $|s'_1| = |s_2|$ performed by insertion or deletion and ② is the cost for $DOC(s'_1, s_2) = 0$ performed by substitution. With respect to ②, DOC value can be decreased by operations for ①. Moreover, DOC value is decreased at most 2 by substitution.

	a	b	c	…	z
OV("abc")	1	1	1	…	0
OV("ab")	1	1	0	…	0
XOR	0	0	1	…	0

DOC value decreased at most **1**

Change of DOC by insertion or deletion

	a	b	c	d	…	z
OV("abc")	1	1	1	0	…	0
OV("abd")	1	1	0	1	…	0
XOR	0	0	1	1	…	0

DOC value decreased at most **2**

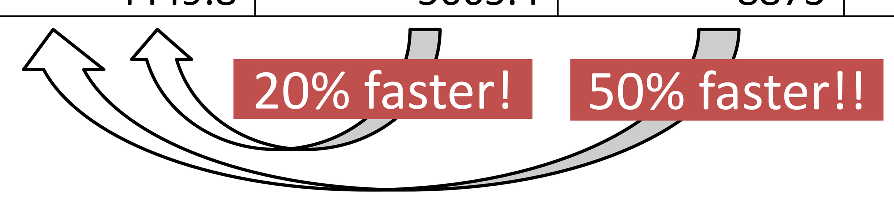
Change of DOC by substitution

Experimental Result

In order to analyze the performance of our methods, we do two experiments. The first measures the execution time given test input with and without these methods. Following table shows that memorization method improves performance by 20% and filtering improves performance by 50%.

Execution time with and without each methods

	With ALL methods	Without Memorization	Without filter	Sample Program
Execution Time [ms]	4449.8	5665.4	8873	1959967

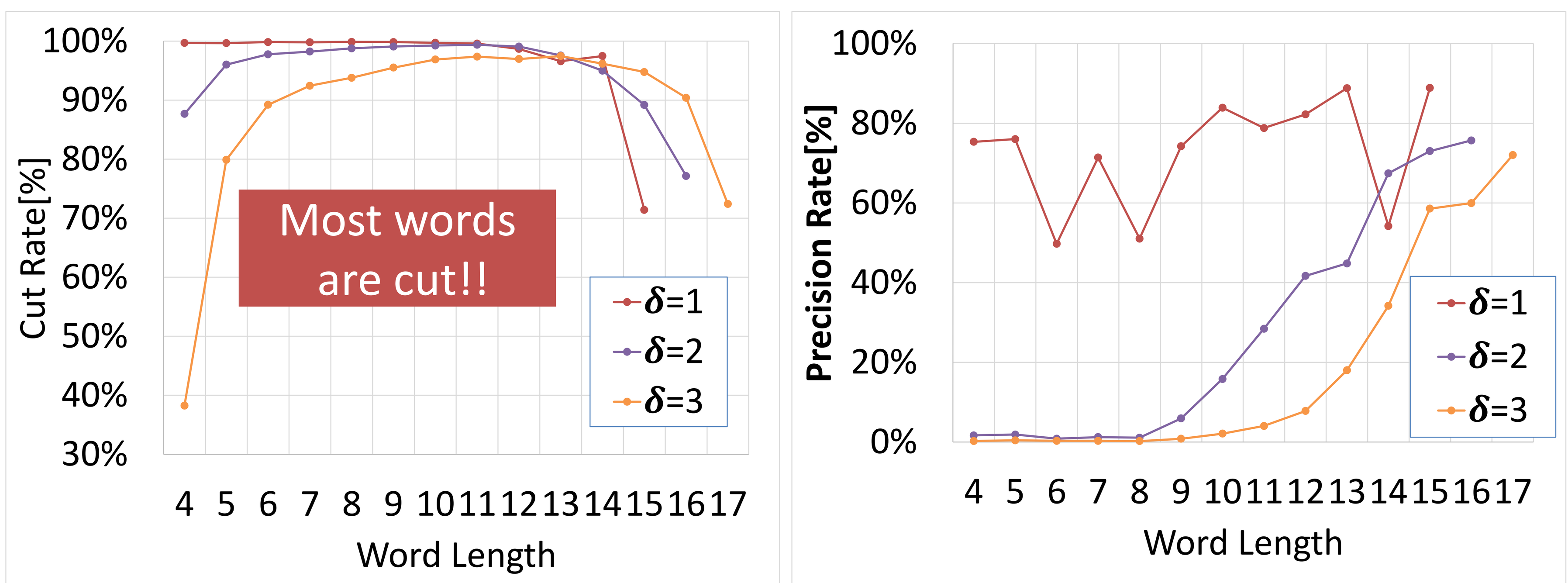


Next experiment measures the cut rate and the precision rate as follow:

$$\text{Cut rate} = \frac{The\ number\ of\ words\ removed\ by\ filter}{The\ number\ of\ words\ requested\ to\ calculate\ edit\ distance}$$

$$\text{Precision rate} = \frac{The\ number\ of\ words\ which\ match\ document}{The\ number\ of\ words\ passed\ filter}$$

Following Graph shows that we can process most queries without calculating exact edit distance. The cut rate of long words looks bad. However, the precision rate of that is high. It means that long words in queries and documents are same in many cases.



Cut rate and precision rate for every word length