Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект

по курсу «Информационный поиск»

Студент: И. С. Глушатов

Преподаватель: А. А. Кухтичев

Группа: М8О-407Б-19

Дата:

Оценка:

Подпись:

Москва, 2022

1. Задание

В курсовой работе необходимо расширить язык запросов булева поиска новым элементом - поиском запросов с джокером общего вида. Синтаксис этого элемента, следующий:

- [красив*] – звёздочка указывает на то, что вместо неё может быть подставлена любая последовательность любого количества символов. В таком случае результат будет включать объединение всех документов, в которых встречаются слова, подходящие под заданный шаблон.
- [кр*ж*к] – возможно несколько пропусков в любом месте слова (начало, конец, середина).
- Новый элемент может комбинироваться с другими стандартными средствами булева поиска,
- например:
- [кр*ж*к & ягода]
- [кр*ж*к || “вкусное варенье”]
- [бе*славные && *л*дки]

Для реализации шаблонного поиска нужно использовать k-граммный индекс. Т. е. для каждой последовательности k символов построить индекс, аналогичный обратному, где последовательности будет соответствовать список слов, в которых она встречается.

2. Описание

В начале следует привести формат k-граммного индекса:

Trigram_index.data								
Триграмма$_1$				**Триграмма$_n$**				
8 байт	8 байт	8 байт		8 байт	8 байт	8 байт		
Номер триграммы	Частота встречаемости	Указатель на список слов	…	Номер триграммы	Частота встречаемости	Указатель на список слов		

Trigram_index.data_ID									
Триграмма$_1$					**Триграмма$_n$**				
8 байт	8 байт		8 байт		8 байт	8 байт		8 байт	
Количество слов	хэш$_1$	…	хэш$_{k1}$	…	Количество слов	хэш$_1$	…	хэш$_{kn}$	

В первом файле для каждой встреченной триграммы хранится её значение, частота и указатель на список слов, в которых эта триграмма встретилась. Во втором файле хранятся списки слов, представленных хэшами.

Процесс разбиения на триграммы:

- К слову прибавляется символ окончания слова ‘$’.
- Слово разбивается на триграммы и для каждой сохраняется структура триграмма-хэш.

К примеру: “дом” => “дом$” => [дом, ом$, м$д, $до].

Номер триграммы – это её представление в виде числа. Так как каждая триграмма это три символа по два байта типа *wchar_t*, то для кодирования всей триграммы достаточно 8 байтного *size_t*. Тогда 6 байтов будут использоваться, а ещё два останутся неинформативными.

После составления k-граммного индекса поиск запроса с джокером общего вида будет подразумевать разбиение слова на триграммы без учёта тех, что включают звёздочку. Пересечение списков слов для полученных триграмм и объединение списков документов для всех полученных слов.

К примеру: “кр*ж*к” => “кр*ж*к$” => [кр*, р*ж, *ж*, ж*к, *к$, к$к, $кр] => [к$к, $кр] => достаем слова по триграммам к$к и $кр, пересекаем списки, получаем список подходящих слов => достаем списки документов для каждого слова, объединяем их => получаем результат.

3. Исходный код

Файл Joker.h:

```h
#pragma once
#include <vector>
#include <string>

class Joker
{
public:

    static std::vector<size_t> trigram(const std::wstring&);
    static std::wstring trigram_from_size_t(size_t);
};
```

Файл Joker.cpp:

Здесь реализованы статические функции преобразования слова в вектор триграмм и преобразование числового значения триграммы в символьное представление.

```cpp
#include "Joker.h"

std::vector<size_t> Joker::trigram(const std::wstring& s) {
    std::wstring ms = s + L"$";

    std::vector<size_t> result(ms.size());

    for (size_t i = 0; i < ms.size(); i++) {
        size_t cur = 0;
        cur |= ms[(i + 2) % ms.size()]; cur <<= sizeof(wchar_t) * 8;
        cur |= ms[(i + 1) % ms.size()]; cur <<= sizeof(wchar_t) * 8;
        cur |= ms[i];
        result[i] = cur;
    }

    return result;
}

std::wstring Joker::trigram_from_size_t(size_t s) {
    std::wstring result = L"000";

    auto shift = (1llu << 16) - 1;
    for (int i = 0; i < 3; i++) {
        wchar_t w = (((s & (shift << (16 * i))) >> (16 * i)));
        result[i] = w;
    }

    return result;
}
```

Файл Tokenization.cpp:

Ниже приведена функция, создающая триграммный индекс по уже отсортированному файлу со структурами триграмма-хэш.

```
void concatenate_trigrams(const std::string& path, const std::string& out) {
    FILE* input, * output, * output_ID;

    OPENMACROS(fopen_s(&input, path.c_str(), "rb, ccs=UNICODE"));
    OPENMACROS(fopen_s(&output, out.c_str(), "wb, ccs=UNICODE"));
    OPENMACROS(fopen_s(&output_ID, (out + "_ID").c_str(), "wb, ccs=UNICODE"));

    TRIGRAM last_TG, TG;
    size_t sum_freq = 0;
    std::vector<size_t> fileHashes;

    while (fread(&TG, sizeof(TRIGRAM), 1, input) > 0) {
        if (sum_freq == 0) {
            last_TG = TG;
            sum_freq++;
            fileHashes.push_back(TG.hash);
        }
        else if (last_TG.trig == TG.trig and last_TG.hash == TG.hash) {
            sum_freq++;
        }
        else if (last_TG.trig == TG.trig) {
            sum_freq++;
            last_TG = TG;
            fileHashes.push_back(TG.hash);
        }
        else {
            size_t fileHashes_size = fileHashes.size();
            POINTER pointer_output_ID = GET_POINTER(output_ID);
            fwrite(&last_TG.trig, sizeof(size_t), 1, output);
            fwrite(&sum_freq, sizeof(size_t), 1, output);
            fwrite(&pointer_output_ID, sizeof(POINTER), 1, output);

            fwrite(&fileHashes_size, sizeof(size_t), 1, output_ID);
            Compressor::compress(output_ID, fileHashes);

            last_TG = TG;
            sum_freq = 1;
            fileHashes.clear();
            fileHashes.push_back(TG.hash);
        }
    }

    if (fileHashes.size()) {
        size_t fileHashes_size = fileHashes.size();
        POINTER pointer_output_ID = GET_POINTER(output_ID);
        fwrite(&last_TG.trig, sizeof(size_t), 1, output);
        fwrite(&sum_freq, sizeof(size_t), 1, output);
        fwrite(&pointer_output_ID, sizeof(POINTER), 1, output);

        fwrite(&fileHashes_size, sizeof(size_t), 1, output_ID);
        Compressor::compress(output_ID, fileHashes);
    }

    fclose(output_ID);
    fclose(output);
    fclose(input);
}
```

Файл BooleanSearching.cpp:

Ниже приведена функция, возвращающая документы, удовлетворяющие введенному шаблону – запросу с джокером общего вида.

```
std::vector<CompareType> getDocsByPattern(const CORPUS& Corpus, std::wstring& buff)
{
	std::vector<CompareType> result;

	std::vector<size_t> trigrams = Joker::trigram(token_to_terms(buff));

	if (trigrams.size() == 0) return result;

	std::vector<size_t> hashes = getHashes(Corpus, trigrams[0]);

	for (size_t i = 1; i < trigrams.size(); i++) {
		std::vector<size_t> hashesi = getHashes(Corpus, trigrams[i]);
		hashes = op_intersect_hashes(hashes, hashesi);
	}

	if (hashes.size() == 0) return result;

	for (size_t i = 0; i < hashes.size(); i++) {
		size_t hash = hashes[i];
		index_dictionary_type::iterator iter =
Corpus.coords_dictionary.find(hash);
		if (iter != Corpus.coords_dictionary.end()) {
			result = op_union(result, get_indexes(Corpus, hash, iter-
>second));
		}
	}

	return result;
}
```

4. Анализ

Тестирование проводилось на нескольких запросах. Ниже представлен отчёт по ним:

Запрос		Время (в ms)	Количество найденных файлов
1.	кры*вник	62.2569	136
2.	ба*илоза*р	25.0741	14
3.	э*прес*о	20.0296	324
4.	**ленинград* & площадь**	**1225.78**	**8379**
5.	г*нический & осц*ятор & "свободные колебания"	152.079	1
6.	("осада столицы" & Византийск* & импери*) | (Падение Константинополя)	629.623	517
7.	**по & площад***	**3486.8**	**265386**
8.	**из*стн* архитектор**	**962.857**	**225004**
9.	обряд поклон*	205.316	19490

По приведённым данным можно сделать вывод, что запрос с джокером в каких-то ситуациях сильно замедляет работу поиска, а в других практически никак на него не влияет. Сложно предсказать, как поведет себя поиск с тем или иным запросом.

Тестирование проводилось на корпусе размером в миллион файлов.
Время построение k-граммного индекса около часа.
Размер получившегося индекса:

- trigram_index.data - 789Кб
- trigram_index.data_ID - 305.5Мб

Всего получилось 33 641 триграммы. Это 85.6% от вообще возможного количества (34^3).
На каждую триграмму в среднем приходится по 1161 слово.

5. Выводы

В ходе курсовой работы я сконструировал k-граммный индекс для поиска запросов с джокером общего вида. Трёхграммный индекс увеличил время токенизации, однако не сильно нагрузил саму поисковую систему по памяти. Время, затрачиваемое для обработки шаблонного запроса, сильно варьируется в зависимости от самого запроса и часто бывает очень высоким, что подталкивает к тому, чтобы пользователь реже обходился данным инструментом. Ещё одним минусом является именно моя реализация алгоритма. Так как в k-граммном индексе сохраняются хэши слов, а не сами слова, то во время поиска я не могу выполнить пост-фильтрацию, из-за чего могут попадаться документы, которые не содержат слова, удовлетворяющего шаблону. Как по мне, данный индекс лучше организовывать, как множество trie'ев. Однако тогда придется писать сложные реализации сериализации, десериализации и пересечения деревьев. Ещё один минус, что часть информации из шаблона может теряться, если, допустим, в шаблоне есть две звёздочки, между которыми только одна буква. В таком случае этот символ не войдет ни в одну триграмму без звёздочки. Но вообще это является вытекающим из отсутствия пост-фильтрации.

Список литературы

[1] Маннинг, Рагхаван, Шютце Введение в информационный поиск — Издательский дом «Вильямс», 2011. Перевод с английского: доктор физ.-мат. наук Д. А. Клюшина — 528 с. (ISBN 978-5-8459-1623-4 (рус.))