ЛР3: Булев индекс

Задание

Требуется построить поисковый индекс, пригодный для булева поиска, по подготовленному в ЛР1 корпусу документов. Требования к индексу:

- Самостоятельно разработанный, бинарный формат представления данных. Формат необходимо описать в отчёте, в побайтовом представлении.
- Формат должен предполагать расширение, так как в следующих работах он будет меняться под требования новых лабораторных работ.
- Использование текстового представления или готовых баз данных не допускается.
- Кроме обратного индекса, должен быть создан «прямой» индекс, содержащий в себе как минимум заголовки документов и ссылки на них (понадобятся для выполнения ЛР4, при генерации страницы поисковой выдачи).
- Для термов должна быть как минимум понижена капитализация.

В отчёте должно быть отмечено как минимум:

- Выбранное внутренне представление документов после токенизации.
- Выбранный метод сортировки, его достоинства и недостатки для задачи индексации.

Среди результатов и выводов работы нужно указать:

- Количество термов.
- Средняя длина терма. Сравнить со средней длинной токена, вычисленной в ЛР1 по курсу ОТЕЯ. Объяснить причину отличий.
- Скорость индексации: общую, в расчёте на один документ, на килобайт текста.
- Оптимальна ли работа индексации? Что можно ускорить? Каким образом? Чем она ограниченна? Что произойдёт, если объём входных данных увеличится в 10 раз, в 100 раз, в 1000 раз?

Метод решения

1. Выбрать алгоритм индексации
2. Выбрать бинарный формат представления данных
3. Реализовать алгоритм
4. Провести анализ решения

- В качестве алгоритма индексации был выбран алгоритм SPIMI. Этот алгоритм является эффективным с точки зрения масштабирования. SPIMI использует термины, а не их идентификаторы, записыват словарь каждого блока на диск, а затем для нового блока начинает создавать новый словарь . При наличии достаточного объема памяти на диске с помощью алгоритма SPIMI можно проиндексировать коллекцию любого размера .

- Для сохранения индексов в побайтовом представлении был выбран инструмент Boost.Serialization. Данная библиотека позволяет преобразовывать объекты в последовательность байтов, которая может быть сохранена и загружена для восстановления объектов. Доступны различные форматы данных для определения правил генерации последовательностей байтов.

Результаты выполнения

Процесс индексации коллекции из ~1.8млн документов занимает ~50 минут.
В качестве ограничения используемой индексатором RAM был программно указан лимит в размере 100тыс статей на индекс, что соответствует ~550мб.

Исходный код

indexer.h

```h
#ifndef TINDEXER
#define TINDEXER

#include <iostream>
#include <unordered_map>
#include <vector>
#include <map>


class TIndexer
{
public:
    TIndexer() {};
    void BuildIndex(std::string input_articles_path, std::string output_indexes_path, unsigned long long batch_size);
    void Load(std::string index_path);
private:
    void WriteToIndex(std::string wiki_doc_path);
    bool InitBuilder(std::string input_articles_path, std::string output_indexes_path, unsigned long long batch_size);
    void Save(std::string counter);
    unsigned long long batch_size;
    std::string input_articles_path;
    std::string output_indexes_path;

    std::map<std::string, std::vector<unsigned long long>> index;
};

#endif // TINDEXER
```

indexer.cpp

```cpp
#include <iostream>
#include <sstream>
#include <chrono>
#include <map>
#include <filesystem>
#include <string>
#include <boost/archive/binary_oarchive.hpp>
#include <boost/archive/binary_iarchive.hpp>
#include <boost/serialization/map.hpp>
#include <boost/serialization/vector.hpp>
#include <boost/filesystem.hpp>
#include <boost/range/iterator_range.hpp>

#include "nlohmann/json.hpp"
#include "indexer.h"


bool FilesystemCreateFolderIdempotent(std::string path) {
    boost::filesystem::path dir(path);

    if (!(boost::filesystem::exists(dir))) {
        return boost::filesystem::create_directory(dir);
    }
    return true;
}


bool TIndexer::InitBuilder(std::string input_articles_path, std::string output_indexes_path, unsigned long long batch_size) {
    if (!boost::filesystem::exists(input_articles_path)) {
        std::cout << "input directory not found" << std::endl;
        return false;
    }

    if (boost::filesystem::is_empty(input_articles_path)) {
        std::cout << "input directory is empty" << std::endl;
        return false;
    }

    if (!FilesystemCreateFolderIdempotent(output_indexes_path)) {
        std::cout << "cannot create output indexes dir" << std::endl;
        return false;
    }

    this->input_articles_path = input_articles_path;
    this->output_indexes_path = output_indexes_path;
    this->batch_size = batch_size;
    this->index.clear();

    return true;
}


```

```cpp
void TIndexer::WriteToIndex(std::string wiki_doc_path) {
    std::ifstream wiki_article(wiki_doc_path);
    nlohmann::json wiki_doc = nlohmann::json::parse(wiki_article);
    std::string doc_id_str = wiki_doc["id"];

    unsigned long long wiki_doc_id = std::stoi(doc_id_str);

    std::string tokenized_text = wiki_doc["text"];
    std::istringstream token_stream(tokenized_text);
    std::string token;

    while(token_stream >> token) {
        auto index_iterator = this->index.find(token);
        if (index_iterator == this->index.end()){
            this->index[token] = std::vector<unsigned long long >{wiki_doc_id};
        } else {
            if (index_iterator->second.back() != wiki_doc_id) {
                index_iterator->second.emplace_back(wiki_doc_id);
            }
        }
    }
}


void TIndexer::BuildIndex(std::string input_articles_path, std::string output_indexes_path, unsigned long long batch_size) {
    if (!this->InitBuilder(input_articles_path, output_indexes_path, batch_size)) {
        return;
    }

    // statistics variables
    unsigned long long articles_counter = 0;
    unsigned long long articles_counter_general = 0;
    std::chrono::steady_clock::time_point time_clock_begin = std::chrono::steady_clock::now();
    std::chrono::steady_clock::time_point time_clock_end;


    for (auto& entry : boost::make_iterator_range(boost::filesystem::directory_iterator(this->input_articles_path), {})) {
        articles_counter++;
        articles_counter_general++;

        if (articles_counter < this->batch_size) {
            this->WriteToIndex(entry.path().string());
        } else {
            for (auto& arr : this->index) {
                std::sort(arr.second.begin(), arr.second.end());
            }
            this->Save(std::to_string(articles_counter_general));

            time_clock_end = std::chrono::steady_clock::now();
            std::cout << "exec time for batch = " <<std::chrono::duration_cast<std::chrono::minutes>(time_clock_end - time_clock_begin).count() << "[m]" << std::endl;
```

```cpp

            articles_counter = 0;
            time_clock_begin = std::chrono::steady_clock::now();
            this->index.clear();
            this->WriteToIndex(entry.path().string());
        }
    }
    if (this->index.size() > 0) {
        this->Save(std::to_string(articles_counter_general));
    }
}


void TIndexer::Save(std::string postfix)
{
    std::cout << this->output_indexes_path << std::endl;

    std::string index_path = this->output_indexes_path + "/wiki_articles_" + postfix + ".index";
    std::ofstream f(index_path, std::ios::binary);
    if (f.fail()) {
        std::cout << "error" << std::endl;
        return;
    }

    boost::archive::binary_oarchive oa(f);

    auto indexValue = this->index;
    oa << indexValue;
}

void  TIndexer::Load(std::string index_path) {
    this->index.clear();
    std::ifstream f(index_path, std::ios::binary);
    if (f.fail()) {
        std::cout << "error while load data in index" << std::endl;
        return;
    }
    boost::archive::binary_iarchive ia(f);
    ia >> this->index;
}
```

Выводы:

В результате выполнения данной лабораторной работы нам удалось проиндексировать ~1.8млн документов российской википедии за 50 мин. Процесс оказался достаточно трудоемким в следствие особенностей языка c+. Работать с json оказалось

затруднительно. Понравилось использовать библиотеку boost как для операций с файловой системой, так и для сериализации/десериализации данных.