

Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной
математики

Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу «Информационный поиск»

Студент: В. М. Филиппов
Преподаватель: А. А. Кухтичев
Группа: М8О-410Б
Дата:
Оценка:
Подпись:

Москва, 2025

Лабораторная работа №4 «Булев индекс»

Необходимо разработать на языке C++ программу, которая индексирует корпус документов. Индекс корпуса необходимо хранить на диске в бинарном формате.

1 Описание

В рамках этой лабораторной работы я реализовал булев индекс, а также вспомогательную хэш таблицу, которая используется при индексации.

1 Булев индекс

Перед нами стоит задача поиска документов, в которых встречаются слова из запросов. В самом простом случае это можно реализовать так: для каждого документа составляем список токенов, которые в нём встречаются. Далее при поиске проходим по каждому документу, и если встречаем слово из запроса, то добавляем этот документ в выдачу. Такой способ индексации называется "прямым". Но этот способ хранения требует от нас хранить все токены всех документов, что может занимать очень много места. Также сам поиск будет медленным, поскольку при каждом запросе мы проходим по абсолютно всем документам, пытаясь найти нужный токен.

Более оптимизированным способом хранения является "обратный индекс". Его суть заключается в том, чтобы для каждого токена хранить список документов, в котором этот токен встречается. Это будет экономнее по памяти, а поиск будет быстрее, поскольку мы можем применять бинарный поиск по токенам или хэш-таблицы. Второе преимущество обратного индекса в том, что нам не нужно «листать» содержимое самих документов. Мы сразу получаем готовый список ID документов и работаем только с ним.

Булев индекс — это вариация инвертированного индекса, которая фиксирует лишь факт присутствия токена в документе без учета его веса или позиции. Такая структура позволяет выполнять поисковые запросы, используя операции булевой алгебры (И, ИЛИ, НЕ), путем пересечения или объединения списков документов.

2 Хэш-таблица

Поскольку по требованиям к лабораторным работам, использовать STL хэш-таблицы нельзя, я написал свою несложную реализацию. Для разрешения коллизий используется метод цепочек на основе односвязного списка. При возникновении коллизии (когда разные ключи имеют один и тот же хэш) новые элементы добавляются в начало списка соответствующего бакета. В качестве основы используется стандартный функтор `std::hash<std::string>`, результат которого приводится к размеру внутреннего массива бакетов с помощью операции взятия остатка по модулю.

3 Способ хранения на диске

Файл состоит из следующих блоков:

1. **Заголовок**, который содержит магическое число (Magic Number) для верификации формата, версию (используется для переключения между сжатым и несжатым режимами), а также общее количество проиндексированных документов и уникальных токенов.
2. **Таблица URL**, содержащая список исходных адресов документов в формате (length, string).
3. **Словарь токенов**, то есть набор структур фиксированного размера TermEntry. Каждая запись содержит:
 - (a) 64-битный хэш токена;
 - (b) Абсолютное смещение к строковому представлению токена;
 - (c) Абсолютное смещение к списку вхождений;
 - (d) Количество документов, содержащих токен.

Благодаря фиксированному размеру TermEntry и предварительной сортировке записей по значению хэша, поиск конкретного слова в индексе осуществляется за $O(\log N)$ с помощью бинарного поиска.

4. **Блок строковых данных**, в котором хранятся сами токены, на которые ссылается словарь.
5. **Блок списков вхождений**, содержащий пары (doc_id, term_frequency). Использование term_frequency понадобится в будущем, когда появится необходимость в использовании TF-IDF. Пока там нули.

2 Исходный код

При реализации был написан интерфейс `IndexSource`. Далее имеются два его наследника: `RamIndexSource` и `MappedIndexSource`, с помощью которых реализуется хранение индекса как в оперативной памяти, так и на диске при помощи маппинга. Внутри `RamIndexSource` используется `HashMap` в качестве хранилища, а внутри `MappedIndexSource` заммапленный файл на диске. уТакже имеется интерфейс `Indexator`, который реализует конкретный метод индексации (булев и в последствии TF-IDF). Основным методом этого класса - это `addDocument`, который добавляет документ в индекс.

Корпус размером в 305135 документов был проиндексирован за 468.388 секунды со скоростью 5817.64 КБ/с. Индекс получился размером в 1.2 ГБ.

```
1 | #pragma once
2 |
3 | #include <vector>
4 |
5 | namespace BinaryFormat {
6 |     struct Header {
7 |         uint32_t magic;
8 |         uint32_t version;
9 |         uint32_t num_docs;
10 |         uint32_t num_terms;
11 |     };
12 |
13 |     struct TermEntry {
14 |         size_t term_hash;
15 |         uint64_t term_offset;
16 |         uint64_t data_offset;
17 |         uint32_t doc_count;
18 |     };
19 |
20 |     const uint32_t MAGIC = 0xABC1234;
21 | } // namespace BinaryFormat
22 |
23 | template <typename T>
24 | struct HashNode {
25 |     std::string key;
26 |     std::vector<T> postings;
27 |     std::unique_ptr<HashNode> next;
28 |
29 |     HashNode(const std::string& k) : key(k), next(nullptr) {}
30 | };
31 |
32 | template <typename T>
33 | class HashMap {
34 | private:
```

```

35     static const size_t BUCKET_COUNT = 100000;
36
37     size_t getHash(const std::string& key) const { return std::hash<std::string>{}(key)
        % BUCKET_COUNT; }
38
39 public:
40     std::vector<std::unique_ptr<HashNode<T>>> buckets;
41     HashMap() { buckets.resize(BUCKET_COUNT); }
42
43     std::vector<T>& get(const std::string& key) {
44         size_t h = getHash(key);
45
46         HashNode<T>* curr = buckets[h].get();
47         while (curr) {
48             if (curr->key == key) {
49                 return curr->postings;
50             }
51             curr = curr->next.get();
52         }
53
54         auto newNode = std::make_unique<HashNode<T>>(key);
55         newNode->next = std::move(buckets[h]);
56         buckets[h] = std::move(newNode);
57
58         return buckets[h]->postings;
59     }
60
61     HashNode<T>* getNode(const std::string& key) {
62         size_t h = getHash(key);
63
64         HashNode<T>* curr = buckets[h].get();
65         while (curr) {
66             if (curr->key == key) {
67                 return curr;
68             }
69             curr = curr->next.get();
70         }
71
72         auto newNode = std::make_unique<HashNode<T>>(key);
73         HashNode<T>* rawPtr = newNode.get();
74
75         newNode->next = std::move(buckets[h]);
76         buckets[h] = std::move(newNode);
77
78         return rawPtr;
79     }
80
81     std::vector<T>* find(const std::string& key) {
82         size_t h = getHash(key);

```

```

83     HashNode<T>* curr = buckets[h].get();
84     while (curr) {
85         if (curr->key == key) {
86             return &curr->postings;
87         }
88         curr = curr->next.get();
89     }
90     return nullptr;
91 }
92
93 template <typename Func>
94 void traverse(Func callback) {
95     for (const auto& bucket : buckets) {
96         HashNode<T>* curr = bucket.get();
97         while (curr) {
98             callback(curr->key, curr->postings);
99             curr = curr->next.get();
100         }
101     }
102 }
103 };
104
105 struct TermInfo {
106     uint32_t doc_id;
107     uint32_t tf;
108 };
109
110 class IIndexSource {
111 public:
112     virtual ~IIndexSource() = default;
113
114     virtual std::vector<TermInfo> getPostings(const std::string& term) = 0;
115
116     virtual std::string getUrl(int doc_id) const = 0;
117
118     virtual uint32_t getTotalDocs() const = 0;
119 };
120
121 class RamIndexSource : public IIndexSource {
122 public:
123     std::vector<std::string> urls;
124     HashMap<TermInfo> index;
125
126     std::vector<TermInfo> getPostings(const std::string& term) override {
127         auto* node = index.getNode(term);
128         return node ? node->postings : std::vector<TermInfo>{};
129     }
130
131     void addUrl(std::string_view url);

```

```

132     void addDocument(const std::string& token, uint32_t doc_id);
133
134     std::string getUrl(int doc_id) const override {
135         if (doc_id >= 0 && doc_id < (int)urls.size()) return urls[doc_id];
136         return "";
137     }
138
139     uint32_t getTotalDocs() const override { return (int)urls.size(); }
140     void dump(const std::string& file, bool zip);
141 };
142
143 class MappedIndexSource : public IIndexSource {
144     int fd = -1;
145     size_t file_size = 0;
146     const char* map_addr = nullptr;
147
148     const BinaryFormat::TermEntry* term_directory = nullptr;
149     uint32_t num_terms = 0;
150     std::vector<std::string> urls;
151     uint32_t file_version = 0;
152
153     const BinaryFormat::TermEntry* findTermEntry(const std::string& term) const;
154
155 public:
156     MappedIndexSource(const std::string& filename, int expected_version) { load(
157         filename, expected_version); }
158     ~MappedIndexSource();
159
160     void load(const std::string& filename, int expected_version);
161
162     std::vector<TermInfo> getPostings(const std::string& term) override;
163     std::string getUrl(int doc_id) const override;
164     uint32_t getTotalDocs() const override { return (int)urls.size(); }
165 };

```

```

1  #pragma once
2
3  #include "index.h"
4  #include "tokenizer.h"
5
6  class IIndexator {
7  protected:
8      std::shared_ptr<Tokenizer> tokenizer;
9      std::shared_ptr<RamIndexSource> source;
10
11 public:
12     IIndexator(std::shared_ptr<RamIndexSource> src, std::shared_ptr<Tokenizer> tok);
13     virtual void addDocument(const std::string_view& url_view, const std::string_view&
14         doc_view) = 0;

```

```

15 |
16 | class BooleanIndexator : public IIndexator {
17 | public:
18 |     BooleanIndexator(std::shared_ptr<RamIndexSource> src, std::shared_ptr<Tokenizer>
        tok);
19 |     void addDocument(const std::string_view& url_view, const std::string_view& doc_view
        ) override;
20 | };

1 | #include "index.h"
2 |
3 | #include <fcntl.h>
4 | #include <sys/mman.h>
5 | #include <sys/stat.h>
6 | #include <unistd.h>
7 |
8 | #include <algorithm>
9 | #include <cstdint>
10 | #include <cstring>
11 | #include <fstream>
12 | #include <iostream>
13 | #include <string_view>
14 |
15 | uint32_t stringHash(std::string_view str) {
16 |     uint32_t hash = 2166136261u;
17 |     for (char c : str) {
18 |         hash ^= (uint8_t)c;
19 |         hash *= 16777619u;
20 |     }
21 |     return hash;
22 | }
23 |
24 | void writeVarInt(std::ofstream& out, uint32_t value) {
25 |     while (value >= 128) {
26 |         out.put((char)((value & 127) | 128));
27 |         value >>= 7;
28 |     }
29 |     out.put((char)value);
30 | }
31 |
32 | int getVarIntSize(uint32_t value) {
33 |     int size = 1;
34 |     while (value >= 128) {
35 |         size++;
36 |         value >>= 7;
37 |     }
38 |     return size;
39 | }
40 |
41 | void RamIndexSource::addUrl(std::string_view url) { urls.emplace_back(url); }

```

```

42
43 void RamIndexSource::addDocument(const std::string& token, uint32_t doc_id) {
44     std::vector<TermInfo>& postings = index.get(token);
45
46     if (postings.empty() || postings.back().doc_id != doc_id) {
47         postings.push_back({doc_id, 1});
48     }
49 }
50
51 void RamIndexSource::dump(const std::string& filename, bool zip) {
52     std::ofstream ofs(filename, std::ios::binary);
53     if (!ofs) throw std::runtime_error("Cannot open file for writing");
54
55     std::vector<std::pair<std::string, std::vector<TermInfo>>> sorted_index;
56     index.traverse([&](const std::string& term, const std::vector<TermInfo>& docs) {
57         sorted_index.push_back({term, docs}); });
58
59     std::sort(sorted_index.begin(), sorted_index.end(),
60         [](const auto& a, const auto& b) { return stringHash(a.first) < stringHash
61             (b.first); });
62
63     BinaryFormat::Header header = {BinaryFormat::MAGIC, zip ? 2u : 1u, (uint32_t)urls.
64         size(), (uint32_t)sorted_index.size()};
65     ofs.write(reinterpret_cast<char*>(&header), sizeof(header));
66
67     for (const auto& url : urls) {
68         uint32_t len = (uint32_t)url.size();
69         ofs.write(reinterpret_cast<char*>(&len), sizeof(len));
70         ofs.write(url.data(), len);
71     }
72
73     uint64_t current_term_offset = (uint64_t)ofs.tellp() + (sorted_index.size() *
74         sizeof(BinaryFormat::TermEntry));
75     uint64_t current_data_offset = current_term_offset;
76     for (const auto& [term, docs] : sorted_index) current_data_offset += term.size() +
77         1;
78
79     for (const auto& [term, docs] : sorted_index) {
80         BinaryFormat::TermEntry entry;
81         entry.term_hash = stringHash(term);
82         entry.term_offset = current_term_offset;
83         entry.data_offset = current_data_offset;
84         entry.doc_count = (uint32_t)docs.size();
85         ofs.write(reinterpret_cast<char*>(&entry), sizeof(entry));
86
87         current_term_offset += term.size() + 1;
88
89         if (zip) {
90             uint32_t prev_id = 0;

```

```

86         for (const auto& p : docs) {
87             current_data_offset += getVarIntSize(p.doc_id - prev_id) + getVarIntSize
                (p.tf);
88             prev_id = p.doc_id;
89         }
90     } else {
91         current_data_offset += docs.size() * sizeof(uint32_t) * 2;
92     }
93 }
94
95 for (const auto& [term, docs] : sorted_index) {
96     ofs.write(term.c_str(), term.size() + 1);
97 }
98
99 for (const auto& [term, docs] : sorted_index) {
100     uint32_t prev_id = 0;
101     for (const auto& p : docs) {
102         if (zip) {
103             writeVarInt(ofs, p.doc_id - prev_id);
104             writeVarInt(ofs, p.tf);
105             prev_id = p.doc_id;
106         } else {
107             ofs.write(reinterpret_cast<const char*>(&p.doc_id), sizeof(uint32_t));
108             ofs.write(reinterpret_cast<const char*>(&p.tf), sizeof(uint32_t));
109         }
110     }
111 }
112 }
113
114 MappedIndexSource::~MappedIndexSource() {
115     if (map_addr && map_addr != MAP_FAILED) {
116         munmap((void*)map_addr, file_size);
117         map_addr = nullptr;
118     }
119     if (fd != -1) {
120         close(fd);
121         fd = -1;
122     }
123     term_directory = nullptr;
124     num_terms = 0;
125 }
126
127 void MappedIndexSource::load(const std::string& filename, int expected_version) {
128     fd = open(filename.c_str(), O_RDONLY);
129     if (fd == -1) throw std::runtime_error("Cannot open index file");
130
131     struct stat sb;
132     if (fstat(fd, &sb) == -1) {
133         close(fd);

```

```

134     fd = -1;
135     throw std::runtime_error("Cannot stat file");
136 }
137 file_size = sb.st_size;
138
139 map_addr = (const char*)mmap(nullptr, file_size, PROT_READ, MAP_PRIVATE, fd, 0);
140 if (map_addr == MAP_FAILED) {
141     close(fd);
142     fd = -1;
143     map_addr = nullptr;
144     throw std::runtime_error("mmap failed");
145 }
146
147 auto* header = reinterpret_cast<const BinaryFormat::Header*>(map_addr);
148 if (header->magic != BinaryFormat::MAGIC) throw std::runtime_error("Invalid magic")
149 ;
150 if (header->version != expected_version) throw std::runtime_error("Invalid version"
151     + std::to_string(header->version));
152
153 file_version = header->version;
154 num_terms = header->num_terms;
155
156 const char* ptr = map_addr + sizeof(BinaryFormat::Header);
157
158 urls.reserve(header->num_docs);
159 for (uint32_t i = 0; i < header->num_docs; ++i) {
160     uint32_t len = *reinterpret_cast<const uint32_t*>(ptr);
161     ptr += sizeof(uint32_t);
162     urls.emplace_back(ptr, len);
163     ptr += len;
164 }
165
166 term_directory = reinterpret_cast<const BinaryFormat::TermEntry*>(ptr);
167 }
168
169 const BinaryFormat::TermEntry* MappedIndexSource::findTermEntry(const std::string&
170     term) const {
171     if (!term_directory || num_terms == 0 || !map_addr) return nullptr;
172
173     size_t h = stringHash(term);
174
175     auto it = std::lower_bound(term_directory, term_directory + num_terms, h,
176         [](const BinaryFormat::TermEntry& entry, size_t val) {
177             return entry.term_hash < val; });
178
179     while (it != term_directory + num_terms && it->term_hash == h) {
180         const char* stored_term = map_addr + it->term_offset;
181         std::string_view stored_view(stored_term);

```

```

179         if (stored_view == term) {
180             return it;
181         }
182         it++;
183     }
184     return nullptr;
185 }
186
187 uint32_t readVarInt(const char*& ptr) {
188     uint32_t value = 0;
189     uint32_t shift = 0;
190     while (true) {
191         uint8_t byte = *reinterpret_cast<const uint8_t*>(ptr++);
192         value |= (static_cast<uint32_t>(byte & 127) << shift);
193         if (!(byte & 128)) break;
194         shift += 7;
195     }
196     return value;
197 }
198
199 std::vector<TermInfo> MappedIndexSource::getPostings(const std::string& term) {
200     const auto* entry = findTermEntry(term);
201     if (!entry) return {};
202
203     std::vector<TermInfo> results;
204     results.reserve(entry->doc_count);
205
206     const char* data_ptr = map_addr + entry->data_offset;
207     uint32_t last_doc_id = 0;
208     for (uint32_t i = 0; i < entry->doc_count; ++i) {
209         if (file_version == 1) {
210             uint32_t doc_id = *reinterpret_cast<const uint32_t*>(data_ptr);
211             uint32_t tf = *reinterpret_cast<const uint32_t*>(data_ptr + sizeof(int));
212             results.push_back(TermInfo{static_cast<uint32_t>(doc_id), static_cast<
                uint32_t>(tf)});
213             data_ptr += sizeof(int) * 2;
214         } else {
215             uint32_t delta = readVarInt(data_ptr);
216             uint32_t current_doc_id = last_doc_id + delta;
217
218             uint32_t tf = readVarInt(data_ptr);
219
220             results.push_back({current_doc_id, tf});
221
222             last_doc_id = current_doc_id;
223         }
224     }
225
226     return results;

```

```

227 | }
228 |
229 | std::string MappedIndexSource::getUrl(int doc_id) const {
230 |     if (doc_id >= 0 && doc_id < (int)urls.size()) return urls[doc_id];
231 |     return "";
232 | }

1 | #include "indexator.h"
2 |
3 | IIndexator::IIndexator(std::shared_ptr<RamIndexSource> src, std::shared_ptr<Tokenizer>
   | tok)
4 |     : source(src), tokenizer(std::move(tok)) {}
5 |
6 | BooleanIndexator::BooleanIndexator(std::shared_ptr<RamIndexSource> src, std:::
   | shared_ptr<Tokenizer> tok)
7 |     : IIndexator(src, std::move(tok)) {}
8 |
9 | void BooleanIndexator::addDocument(const std::string_view& url_view, const std:::
   | string_view& doc_view) {
10 |     auto ramSource = std::static_pointer_cast<RamIndexSource>(source);
11 |
12 |     uint32_t doc_id = ramSource->getTotalDocs();
13 |
14 |     ramSource->addUrl(url_view);
15 |
16 |     tokenizer->tokenize(doc_view);
17 |     std::vector<std::string> tokens = tokenizer->getTokens();
18 |
19 |     for (const std::string& token : tokens) {
20 |         ramSource->addDocument(token, doc_id);
21 |     }
22 | }

```

3 Выводы

За время выполнения этой лабораторной работы я узнал о том, что такое булев индекс, а также поработал с отображением файла в память. Узнал, как сделать бинарный поиск прямо по данным, которые лежат на диске, создав специальный формат файлов для этого.

Список литературы

- [1] Маннинг, Рагхаван, Шютце *Введение в информационный поиск* — Издательский дом «Вильямс», 2011. Перевод с английского: доктор физ.-мат. наук Д. А. Ключина — 528 с. (ISBN 978-5-8459-1623-4 (рус.))
- [2] Список использованных источников оформлять нужно по ГОСТ Р 7.05-2008