基于自适应基数树的高性能文本倒排索引

Project 1 实验报告
高级数据结构 (2024 春学期)
浙江大学

学生姓名

学号
指导教师：卜佳俊

完成日期：2024 年 3 月 16 日

Made
With LaTeX

目录

第一章　简介

1.1　背景

随着信息技术的不断发展，文本信息的存储和检索成为了人们日常生活和工作中的重要需求。在信息检索领域，倒排索引是一种常用且高效的数据结构，用于快速定位包含特定词项的文档集合。倒排索引通过将每个词项映射到包含该词项的文档列表，为文本检索提供了强大的支持。

当倒排索引的词项列表较大时，线性查找的词项索引会导致检索效率下降。为了提高检索效率，研究者们提出了许多优化方法，例如压缩倒排索引、多级索引等。此外，树结构也被广泛应用于索引构建，以提高索引的组织和检索效率。例如，B 树和 B+ 树等树状结构被用于数据库索引中，以加快数据的检索速度和提高存储效率。

近年来，自适应基数树（Adaptive Radix Tree）[1] 作为一种新型的树状结构，引起了研究者们的广泛关注。它具有动态调整节点大小和结构的能力，能够适应不同类型的键-值数据。它精巧设计节点的结构，适应现代计算机体系结构，在基于主存的信息检索系统中表现出了良好的性能。本次实验将基于自适应基数树设计和实现一个高性能的文本倒排索引系统。我们将通过实验验证自适应基数树在文本检索中的性能表现，并对其进行分析和评价。

1.2　词干提取模块

在信息检索中使用词干提取（stemming）的原因是为了解决词形变化带来的检索问题。词形变化是指同一个词的不同形式，如单词的时态、复数形式、派生词等。例如，单词”run” 可能在不同上下文中出现为”running”、”runs” 或”ran”。

在信息检索系统中，如果不进行词干提取，那么当用户查询一个单词的时候，系统只能检索该单词在文本中的完全匹配，而无法考虑到这个单词的其他形式。这将导致以下几个问题：

- 词形变化带来的查询不匹配：用户可能使用不同的单词形式进行检索，但由于词形变化的存在，相关文档可能无法被正确检索出来。例如，用户可能搜索”run”，但文档中实际包含的是”running”。
- 文档中的冗余信息：由于同一个词的不同形式会分别出现在文档中，文档集合可能会包含大量相似的词语，导致冗余信息的存在。这不仅浪费了存储空间，还增加了检索的复杂性。
- 降低检索质量：由于词形变化的存在，可能会导致一些相关文档被错误地排除在搜索结果之外，或者一些无关文档被错误地包含在搜索结果中，从而降低了检索的准确性和质量。

通过使用词干提取算法，可以将不同形式的单词转化为它们的词干或基本形式，从而统一表示同一个词的不同变种。这样一来，就能够实现更全面的检索，将相关文档正确地匹配给用户的查询。词干提取的目标是去除单词的词缀，保留其词干部分，从而达到减少词形变化的影响，提高信息检索的召回率和准确性。

本次实验中，我们使用以下开源工具和数据集进行词项提取：

- 词干提取器：我们使用 GitHub 上开源的 C++ 词干提取器 `porter2_stemmer`[2] 进行词干提取

- 停用词列表：我们采用 MySQL 的两个知名全文本搜索引擎的停用词列表[3]：
  - MyISAM：543 词
  - InnoDB：36 词

从文档中提取的每一个单词，先进行词干提取，去除停用词，再作为词项加入倒排索引中。查询时，也需要对查询词进行同样的处理。在代码中，我们将其包装在工具函数 `stemming` 中，如下所示：

```
std::string stemming(std::string str)
{
    std::transform(str.begin(), str.end(), str.begin(), tolower);
    Porter2Stemmer::trim(str);
    Porter2Stemmer::stem(str);
    return str;
}
```

Listing 1.1: 词干提取包装

第二章　数据结构和算法说明

2.1　基于 std::map 的简单实现

基于 STL 库的 `std::map`，我们在实验开始时实现了一个简单的倒排索引，代码如下：

```
class Database {
    std::vector<std::string> docIDs_;
    std::map<std::string, std::map<int, int>> invert_index_;
    std::set<std::string> stop_words_;
public:
    void readDoc(std::string file);
    void Database::queryWord(std::string word, std::ostream &out);
}
void Database::readDoc(std::string file) {
    std::ifstream in(file);
    int docID = docIDs_.size();
    docIDs_.push_back(file);

    std::map<std::string, int> word_count;
    for (std::string str; in >> str;) {
        std::string word = stemming(str);
        if (word.empty() || stop_words_.count(word))
            continue;
        word_count[word]++;
    }
    for (const auto &word : word_count)
        invert_index_[word.first][docID] = word.second;
    in.close();
}
void Database::queryWord(std::string word, std::ostream &out = std::cout) {
    word = stemming(word);
    if (stop_words_.count(word)) {
        std::cerr << "[queryWord] Can't Query Stop Word: " << word << std::endl;
        return;
    }
    if (invert_index_.count(word) == 0) {
        out << "[queryWord] Word Not Found: " << word << std::endl;
        return;

    for (const auto &doc : word_index) {
        out << docIDs_[doc.first] << " " << doc.second << std::endl;
    }
}
```

Listing 2.1: 基于 std::map 的简单实现

`std::map` 采用经典的红黑树实现，插入和查询时的时间复杂度为 $O(\log n)$。对于实验要求的莎士比亚文集，其效率已经足够好。但我们将在性能测试章节中看到，对于更大的数据集，这种简单实现的效率将远远不够。

2.2 对提供的资料的看法

课程提供了一些倒排索引的资料，但这些资料都太过老旧（2008 年），其做法已经不再适用于现代高性能计算。这些资料假设的模型中，内存仅有 40MB，不得不使用压缩倒排索引、外部归并排序等算法，并将大量词项索引存储在磁盘上。而现代计算机的内存已经远远超过这个数量级，越来越多的数据库选择直接将索引存储在主存中，极大地提高了检索效率。如图 2.1 所示，进入 21 世纪，处理器的性能增长远远超过了内存的增长，这使得主存成为了瓶颈。因此，现代高性能计算的关键点在于如何合理设计数据结构，充分利用 CPU 中的多级缓存，减少内存访问次数，提高计算效率。

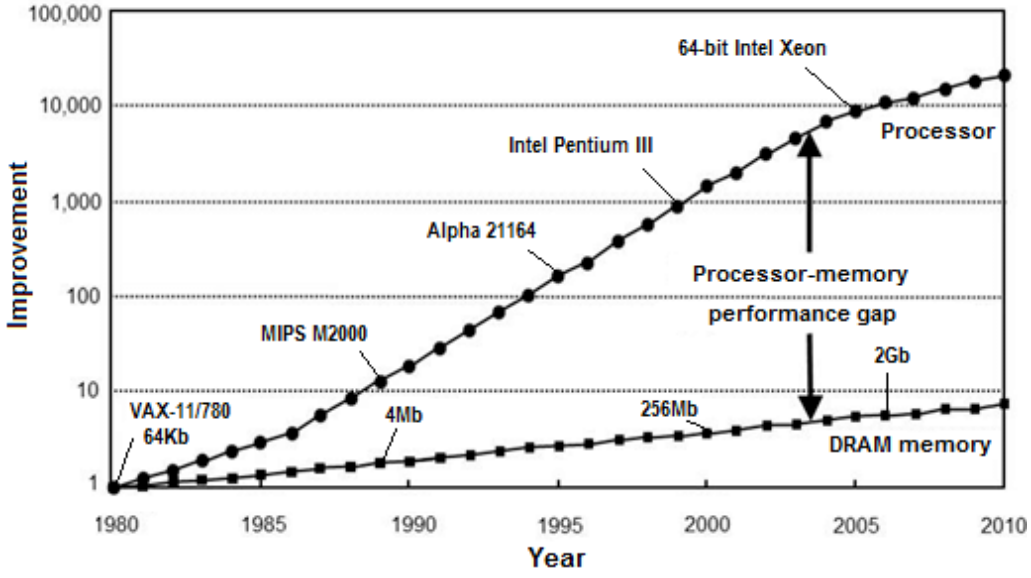


图 2.1: 处理器与内存的性能差距

2.3 基于 ART 的倒排索引设计

adaptive radix tree(ART)，中文名自适应基数树，是相对于红黑树、B+ 数等更加新也更加高效的数据结构。它的查找性能超过了高度调优的只读搜索树，同时也支持非常高效的插入和删除。同时，ART 具有很高的空间效率，通过自适应地为内部节点选择紧凑高效的数据结构，解决了困扰大多数根树的最坏情况空间消耗过大的问题。

由于本次实验中需要处理的数据基本都是由英文字母组成的字符串，这让使用 ART 储存词频数据时，数的高度能够控制在一个较小的范围内，这与单词的长度有关。相较于红黑树等高度在 $O(\log n)$ 的数据结构，ART 在插入和查询方面明显更加高效。

2.3.1 自适应节点

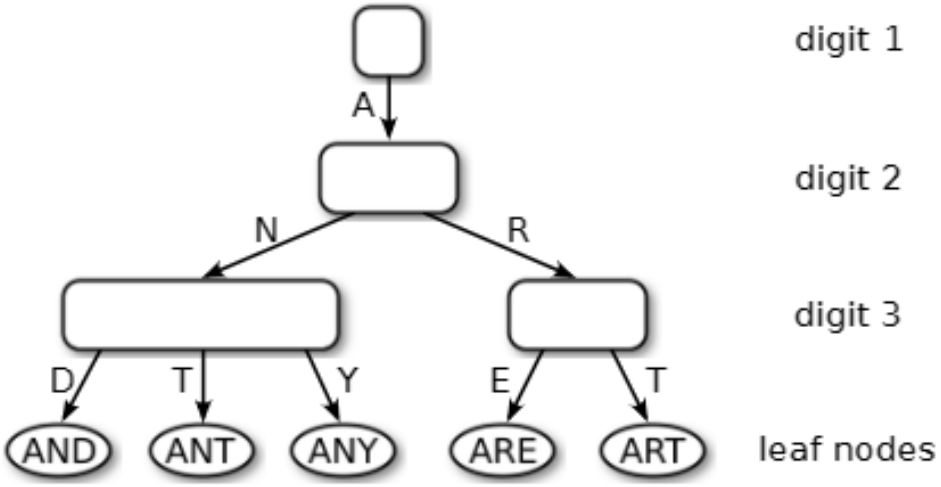
ART 中包含了两种类型的节点：

- 内部节点：将 key 映射到其他节点，用于表示路径。
- 叶子节点：存储实际的数据，本次 project 中储存的是指向该 term 索引指针

根据容量不同，ART 内部节点使用了四种不一样的内部结构。

Node4 是最小的节点类型，其由 4 个 key 以及对应 4 个子指针组成。key 和子指针存储在相同的位置，并以 key 为基准进行排序。Node16，Node48，Node256 是 Node4 的扩容版本，分别存储着 16、48、256 个 key 和子指针。

图 2.2: ART 树图示

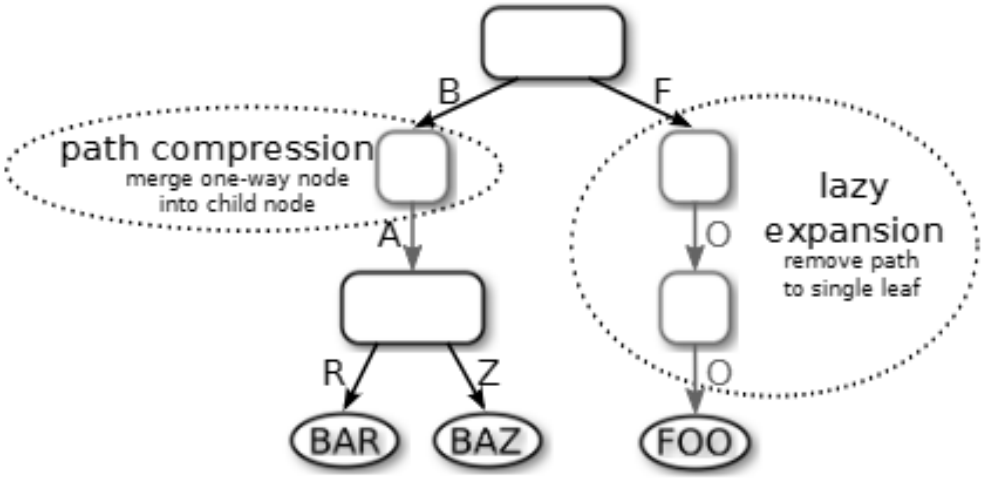


节点自适应即当一个节点容量过大/小时，需要替换成其他内部节点类型，此时执行以下流程：
1、对该节点和其父节点进行加锁，并创建一个新点节点，将旧节中的数据全部拷贝到新节点中。
2、通过原子操作修改父节点的子指针，让其指向新节点。
3、解锁旧节点，将其标记为删除。

2.3.2 内节点塌缩

内节点塌缩用于简化树，在某些情况下，特别是对于长键，每个键的空间消耗还是很大的。对此可以使用惰性展开路径压缩两种方法之一，通过减少节点数量来降低高度。这对于长键非常有效，可以显著提高这类索引的性能。同样重要的是，它们减少了空间消耗，并确保最小的最坏情况空间边界，使性能更加稳定。

图 2.3: 惰性展开与路径压缩



惰性展开：只有在需要区分至少两个叶节点时才创建内部节点。即当我们插入一个节点时，如果该节点是其父母的唯一儿子，我们便不创建内部节点，当有其兄弟需要插入时再扩张该路径。因为扩张到叶子

的路径可能被截断，此优化需要被存储下来。下面示例中惰性展开通过截断到叶子“FOO”的路径来节省两个内部节点。如果插入另一个前缀为“F”的叶子，则扩展该路径。

路径压缩：删除只有一个子节点的所有内部节点，在每个内部节点，存储一个可变长度 (可能为空) 的部分键向量。它包含前面所有已删除的单向节点的密钥。在查找期间，将此向量与搜索键进行比较，然后再继续查找下一个子向量。实例中删除了存储部分键“A”的内部节点，储存了”AB”的部分键向量

2.4 搜索引擎设计

2.4.1 Search 算法

```
search (node, key, depth)
if node==NULL
  return NULL
if isLeaf(node)
  if leafMatches(node, key, depth)
    return node
  return NULL
if checkPrefix(node,key,depth)!=node.prefixLen
  return NULL
depth=depth+node.prefixLen
next=findChild(node, key[depth])
return search(next, key, depth+1)
```

图 2.4: ART 树 Search 算法伪代码

在查找时，由于有路径压缩，我们需要带上一个 depth, 表示实际上，我们已经到了第几层。在查找内部节点的时候，先判断前缀是否匹配，如果匹配，需要将 depth 加上前缀的长度。不匹配则返回不存在。

2.4.2 Insert 算法

在插入 key 时，先查找到应该插入的位置。结果有三种可能:

- 无法通过 key 的前缀走到叶子节点（包含空的节点）
- 通过 key 的前缀走到非空的叶子节点（此叶子节点不匹配）
- 通过 key 的前缀走到空的叶子节点

对于情况（1），在匹配的部分新建节点，相当于在此处新建一条 path，然后指向 key.
对于情况（2），同样是新建节点，再完善 key 与这个叶子节点不匹配的部分。
对于情况（3），直接向内部节点插入叶子节点即可。
另外，由于本次实验不需要进行删除操作，对 ART 的 Delete 算法便不多做赘述。

```
insert (node, key, leaf, depth)
if node==NULL // handle empty tree
  replace(node, leaf)
  return
if isLeaf(node) // expand node
  newNode=makeNode4()
  key2=loadKey(node)
  for (i=depth; key[i]==key2[i]; i=i+1)
    newNode.prefix[i-depth]=key[i]
  newNode.prefixLen=i-depth
  depth=depth+newNode.prefixLen
  addChild(newNode, key[depth], leaf)
  addChild(newNode, key2[depth], node)
  replace(node, newNode)
  return
p=checkPrefix(node, key, depth)
if p!=node.prefixLen // prefix mismatch
  newNode=makeNode4()
  addChild(newNode, key[depth+p], leaf)
  addChild(newNode, node.prefix[p], node)
  newNode.prefixLen=p
  memcpy(newNode.prefix, node.prefix, p)
  node.prefixLen=node.prefixLen-(p+1)
  memmove(node.prefix,node.prefix+p+1,node.prefixLen)
  replace(node, newNode)
  return
depth=depth+node.prefixLen
next=findChild(node, key[depth])
if next // recurse
  insert(next, key, leaf, depth+1)
else // add to inner node
  if isFull(node)
    grow(node)
  addChild(node, key[depth], leaf)
```

图 2.5: ART 树 Insert 算法伪代码

第三章　性能测试与分析

3.1　数据集

在整个项目过程中，我们使用了很多开源的数据集进行测试：

表 3.1: 文本数据集规模

名称	大小	词数	停用词
Vep: Shakespeare Collection	9.46MB	1,716,033	967,983(InnoDB) 1,011,321(MyISAM+InnoDB)
MS MACRO: Collection	2.85GB	506,232,115	239,578,257(MyISAM+InnoDB)
MS MACRO: FullDocs(half)	10.5GB	1,745,019,035	774,049,681(MyISAM+InnoDB)

- 文本数据集：
  - 莎士比亚全集[4]：实验要求的数据集。
  - MS MACRO[5]：文本检索排序领域最具代表性的数据集，收录了微软 Bing 搜索引擎和 Cortana 智能助手近百万查询词与 800 万文档在内的真实搜索场景数据。
- 查询数据集：
  - 常见 1000[6]、3000[7]、5000[8]、10000 词[9]。
  - 常见 7 字符词[6]：1371 词。
  - 全部 7 字符词[6]：40093 词。

MS MACRO 分为 FullDoc 和 Collection 两个数据集，各自是一个单独的文件。我们采用 Linux 实用工具 `split` 进行切分：

- Collection 数据比较规整，每 1000 行切分为一个文件，得到 8842 个文件，每个文件词数在 55000 左右。
- FullDocs 中有极少数的非 ASCII 字符和部分非字母字符，stemmer 会将其去除。受内存限制，取数据集的前一半切分为 500,000 个文件。平均每个文件 3490 词。

下面展示了两个数据集文章内容的片段，可以看出 Collection 主要是普通文章，FullDocs 则是各类网络资源的大杂烩。

```
1 0 The presence of communication amid scientific minds was equally important to the success of the Manhattan
      Project as scientific intellect was. The only cloud hanging over the impressive achievement of
```

Listing 3.1: Collection

```
The total energy flux at a spherical surface of Radius R is Q = q· ·R² = · ·T^4· ·R² Hence the radius is R = √ ( 2.7x10+32 W / (1 · 5.67x10-8W/m²K^4 · (1100K)^4 · ) ) = 3.22x10+13 mSource (s):http://en.wikipedia.org/wiki/Stefan_bolt...schmiso · 1 decade ago0 18
```

Listing 3.2: FullDocs

3.2 测试设计

名称	数据集	停用词	查询	校验	输出
正确性	Shakespeare	InnoDB	1000+ 常见 7 字符	是	全部
基础	Collection	MyISAM + InnoDB	全部查询	否	Top20
极限	FullDocs(half)	MyISAM + InnoDB	全部查询	否	Top10

- 评价项：索引时间、排序时间、查询时间、内存占用，以及它们随索引大小的变化。
- 正确性测试：输出详细调试数据，打开文本文件核对词频。通过正确性测试后，其余测试不再核对。

3.3 测试平台

CPU	系统	虚拟机内存	磁盘文件系统	编译器
Intel i5-12400	Ubuntu WSL2	15.5GB+4GB SWAP	Ext4	Clang 14

编译指令 `clang++ -std=c++11 -Wall -pedantic -I. -O3 -DBENCHMARK`

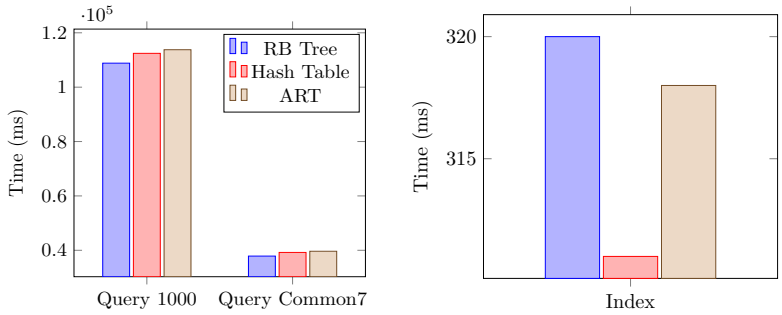
3.4 测试结果及分析

3.4.1 莎士比亚全集（查询时校验）

- 查询：无法体现，主要耗时在 I/O（校对文件、输出完整 Posting）。1000 命中率高，故耗时比 Common7 高。
- 索引：哈希表最快，ART 次之，红黑树最慢。
- 内存：ART 树较大，因为不能完全利用 Node48。
- 排序：Posting 排序长度和算法相同，本质上测试的是迭代器速度。ART 库的迭代器使用栈实现，未进行针对性优化，显著劣于标准库。

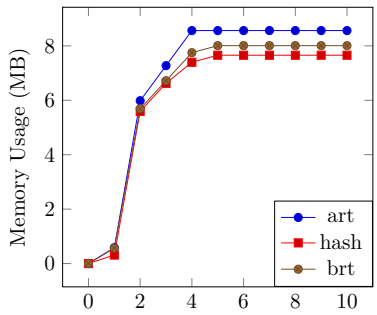
3.4.2 MS MACRO：Collection

- 查询：三者中最快，但差距不大。
- 索引：与红黑树拉开差距。
- 排序：迭代器性能不佳。
- 内存：经过 stemmer，键值串只有小写字符和撇号，Node48 利用效率仅为 27/48。且 Node48 结构特殊，占用 256 个键值位。

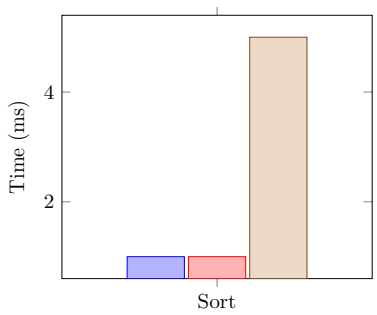


(a) 查询

(b) 索引

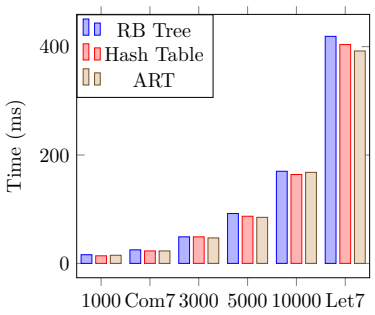


(c) 内存使用

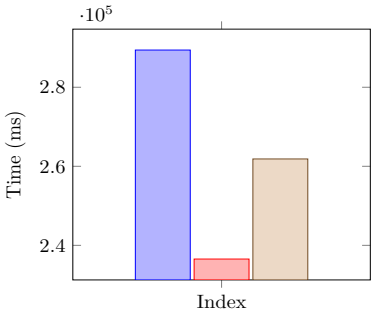


(d) 排序（迭代器）

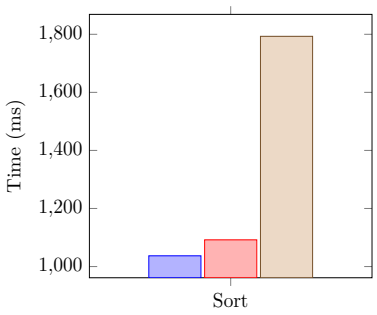
图 3.1: 莎士比亚全集（校验）的性能表现



(a) 查询



(b) 索引



(c) 排序（迭代器）

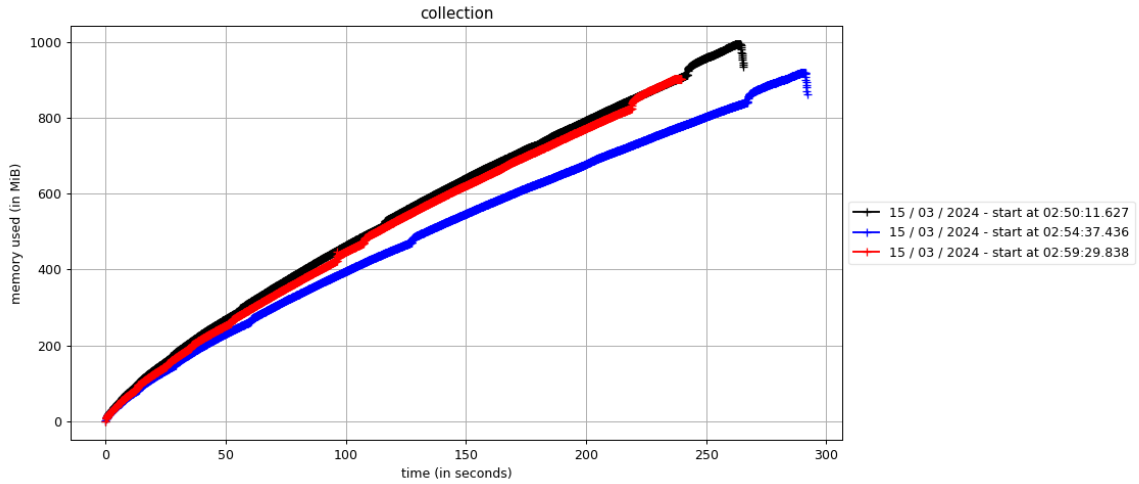
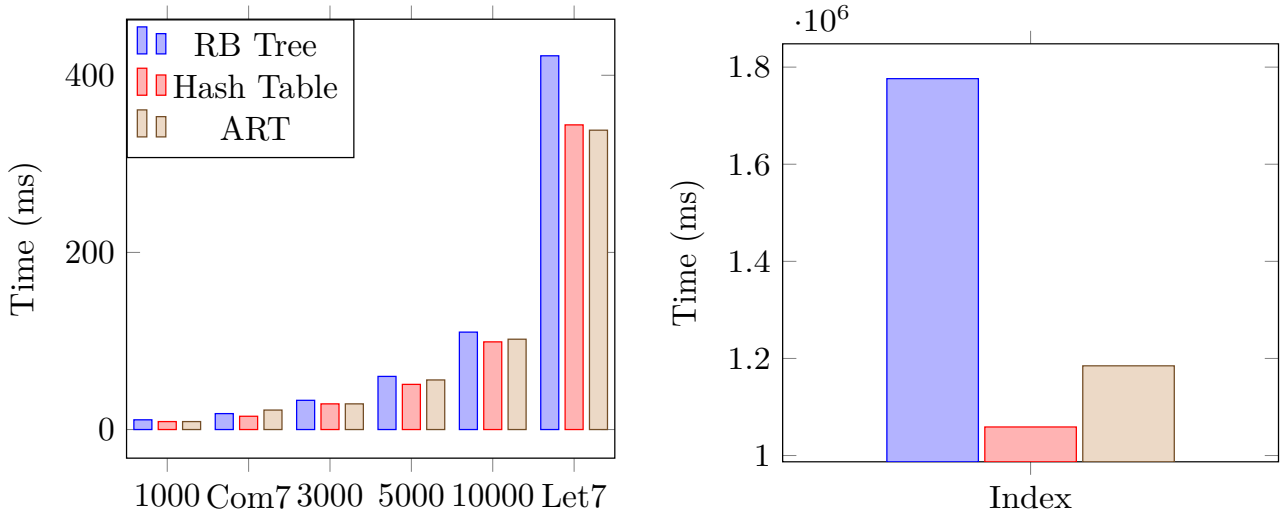


图 3.3: MS MACRO Collection 内存表现：ART 树（Black）哈希表（Red）红黑树（Blue）

3.4.3 MS MACRO：FullDocs（Half）



- 查询：键长增加，ART 树查询效率与红黑树拉开差距。
- 索引：显著优于红黑树，与哈希表差距缩小。
- 内存：无明显变化。

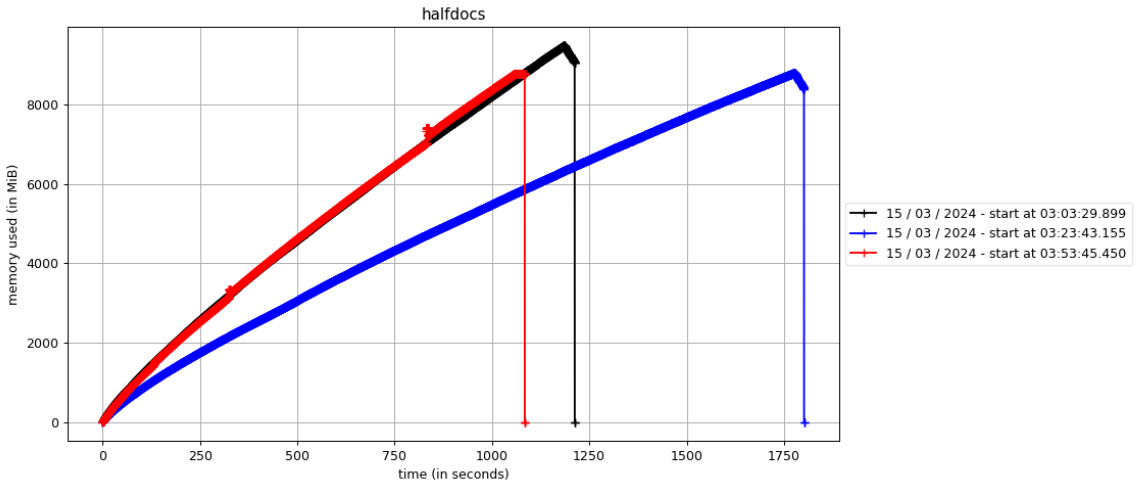


图 3.5: MS MACRO FullDocs(half) 内存表现：ART 树（Black）哈希表（Red）红黑树（Blue）

第四章　思考与展望

4.1　内存利用率优化

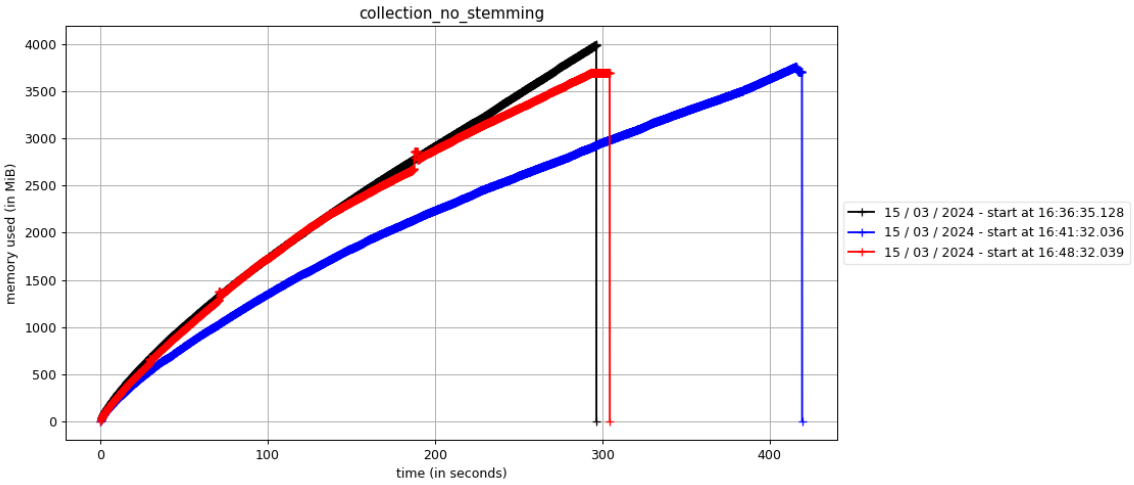
1. 关闭 stemmer，自然词项下一字符大多不会有超过 48 种可能，完整利用 Node48。
2. 对于复杂文本，上面的做法也可能造成更低的利用率 95/256（ASCII 可打印字符）。具体的优化措施需要根据实际数据集进行，基于统计数据设计节点度数。

4.2　长键测试

关闭 stemmer 后在 Collection 上的内存表现。此时平均键长显著增大，索引速度甚至快于哈希表。符合前面的分析，内存没有显著优化，但是索引速度有所提升。

图中可以看到显著的哈希表扩张现象。

图例：ART 树（Black）哈希表（Red）红黑树（Blue）。



4.3　ART 树的应用前景

- 键值较长时，相比其他树状结构，存储量显著减少。
- 大量的 INT、FLOAT 作为键值数据时，ART 树对内存的优化将体现出来。
- 在查询速度比肩哈希表的同时，能够高效实现模糊查找、范围搜索等。

附录 A　引用的开源项目

- Typesense/typesense: Open source alternative to Algolia + Pinecone and an easier-to-use alternative to Elasticsearch - fast, typo tolerant, in-memory fuzzy search engine for building delightful search experiences[10]
- Armon/libart: Adaptive radix trees implemented in C[11]
- Rafaelkallis/adaptive-radix-tree: An adaptive radix tree for efficient indexing in main memory.[12]

附录 B　声明

I hereby declare that all the work done in this project titled “Roll Your Own Mini Search Engine” is of our independent effort.

Signatures:

参考文献

[1] LEIS V, KEMPER A, NEUMANN T. The adaptive radix tree: Artful indexing for main-memory databases[C/OL]//2013 IEEE 29th International Conference on Data Engineering (ICDE). 2013: 38-49. DOI: 10.1109/ICDE.2013.6544812.

[2] SMASSUNG. Smassung/porter2_stemmer: C++ version of the porter2 english stemmer[J/OL]. GitHub, 2015. https://github.com/smassung/porter2_stemmer.

[3] MARIADB. Full-text index stopwords[J/OL]. MariaDB KnowledgeBase, 2017. https://mariadb.com/kb/en/full-text-index-stopwords/.

[4] PROJECT V. Vep shakespeare collection[J/OL]. Visualizing English Print, 2016. https://graphics.cs.wisc.edu/WP/vep/vep-shakespeare-collection/.

[5] BAJAJ P, CAMPOS D, CRASWELL N, et al. Ms marco: A human generated machine reading comprehension dataset[A]. 2018. arXiv: 1611.09268.

[6] POWERLANGUAGE. Powerlanguage/word-lists: Lists of english words. perhaps good for word games[J/OL]. GitHub, 2015. https://github.com/powerlanguage/word-lists.

[7] HYPER NEUTRINO. 3000 common english words[J/OL]. Gist, 2020. https://gist.github.com/hyper-neutrino/561f120125ae0e7c1d22777eebf083c8.

[8] MICHAELWEHAR. Michaelwehar/public-domain-word-lists: A collection of plain text or csv formatted public domain word lists.[J/OL]. GitHub, 2017. https://github.com/MichaelWehar/Public-Domain-Word-Lists.

[9] ERIC P. 10000 word list[J/OL]. MIT, 2007. https://www.mit.edu/~ecprice/wordlist.10000.

[10] TYPESENSE. Typesense/typesense: Open source alternative to algolia + pinecone and an easier-to-use alternative to elasticsearch - fast, typo tolerant, in-memory fuzzy search engine for building delightful search experiences[J/OL]. GitHub, 2024. https://github.com/typesense/typesense.

[11] ARMON. Armon/libart: Adaptive radix trees implemented in c[J/OL]. GitHub, 2021. https://github.com/armon/libart/.

[12] RAFAELKALLIS. Rafaelkallis/adaptive-radix-tree: An adaptive radix tree for efficient indexing in main memory.[J/OL]. GitHub, 2022. https://github.com/rafaelkallis/adaptive-radix-tree/.