基于自适应基数树的高性能文本倒排索引

——高级数据结构与算法课程实验一

*** *** ***

第一组
指导教师 卜佳俊

计算机学院
浙江大学

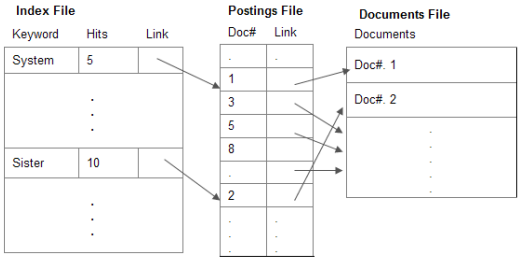
2024 年 3 月 15 日

提纲

基础倒排索引

倒排索引词项列表较大时，线性查找效率下降.

- 课堂和资料介绍：压缩倒排索引、外部归并排序等.
- 主流应用：数据库索引广泛使用 B 树和 B+ 树等树状结构.



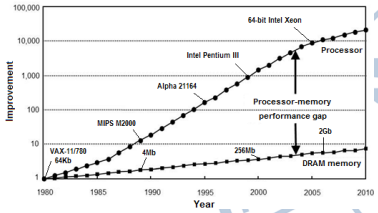
图：基础倒排索引实现

数据结构与算法的趋势

学在浙大上提供资料太过老旧（2008 年）：

- 假设不再适用：模型中内存仅有 40MB
- 操作复杂：使用压缩倒排索引、外部归并排序压缩空间
- I/O 开销大：大量词项索引存储在磁盘上

基于磁盘的数据索引已经不再适用于现代高性能计算，越来越多的数据库直接将索引存储在主存中．21 世纪以来，处理器的性能增长远远超过了内存的增长，主存访问速度成为了瓶颈．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：现代计算机 CPU 与内存性能增长差异

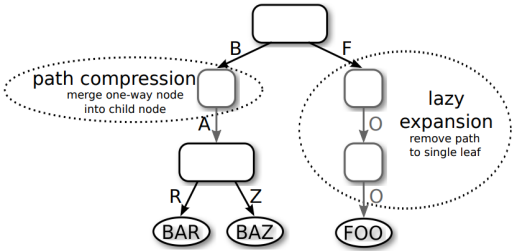
自适应基数树（IEEE 2013）

The Adaptive Radix Tree:
ARTful Indexing for Main-Memory Databases

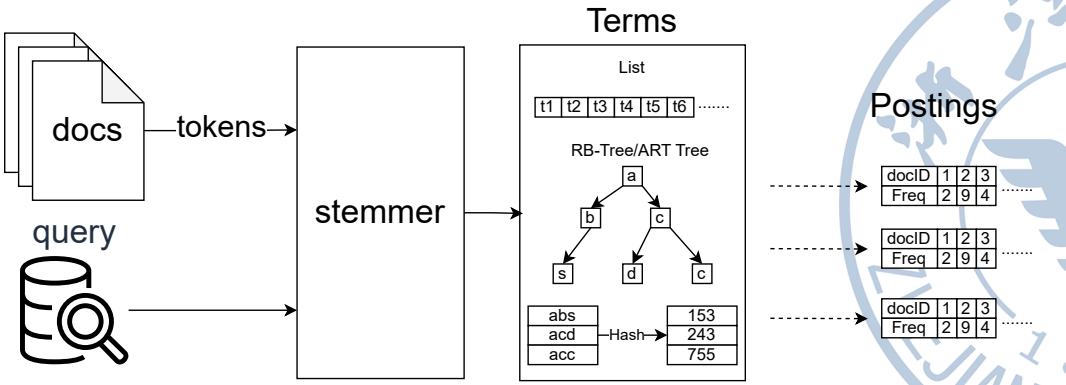
Viktor Leis, Alfons Kemper, Thomas Neumann

Fakultät für Informatik
Technische Universität München
Boltzmannstrae 3, D-85748 Garching
<lastname>@in.tum.de

- 动态调整节点大小和结构的，适应不同类型的键-值数据.
- 精巧设计节点的结构，高效利用 CPU 缓存



程序结构



在词项索引部分，我们替换多种实现比较性能.

使用的开源代码

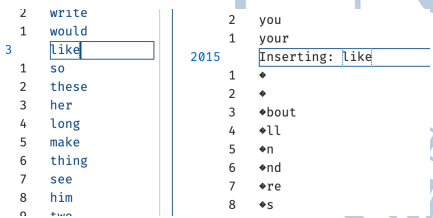
词干提取（2014，23★）

smassung/porter2_stemmer: C++ version of the porter2 english stemmer

ART 的 C++ 实现（2021，127★）

rafaelkallis/adaptive-radix-tree: An adaptive radix tree for efficient indexing in main memory

- 测试时发现 ART 库迭代器存在问题：插入某些键后，后续的遍历输出乱码.
- 对比基准程序，仅插入到某词后在特定位置上乱码，推测根节点扩张问题.



图：输出乱码

发现开源库存在问题

- 监视输出字符串，找到乱码字节\xc9来源于迭代器tree::it<T> key()函数.
- 对 tree::it<T> key() 单步调试，发现 art::art<T> begin() 得到的迭代器就已经存储有 \xc9 (-55) 作为部分键值.
- 追踪 tree::it<T>、tree::it<t>::step、child::it<T> 的构造函数，寻找错误的值如何在构造时进入节点，定位到一次函数调用 cur_partial_key_ = node_->prev_partial_key(-128);
- 父节点为 I (73)，注意到二进制值的关系 $-128+73=-55$，并研究不同节点类型设置的偏移值，找到 Bug 根源：节点扩张时数据迁移偏置量错误.

为什么这么久没有人发现？

因为 ART 节点的设计，其余函数使用相对位置，只要不通过迭代器获取键值，难以发现.

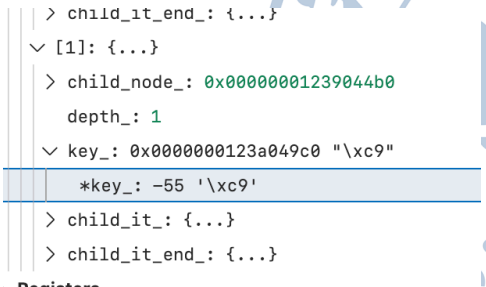
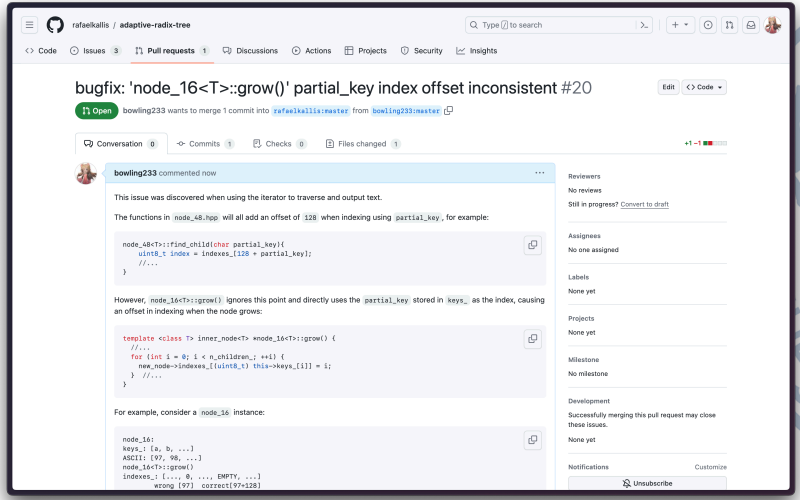


图: 调试过程

回馈开源社区


rafaelkallis / adaptive-radix-tree
Code
Issues 3
Pull requests 1
Discussions
Actions
Projects
Security
Insights
bugfix: 'node_16<T>::grow()' partial_key index offset inconsistent #20
Open
bowling233 wants to merge 1 commit into rafaelkallis:master from bowling233:master
Edit
Code
Conversation 0
Commits 1
Checks 0
Files changed 1
+1 −1
bowling233 commented now
This issue was discovered when using the iterator to traverse and output text.
The functions in node_48.hpp will all add an offset of 128 when indexing using partial_key, for example:
node_48<T>::find_child(uchar partial_key){
uint8_t index = indexes_[128 + partial_key];
//...
}
However, node_16<T>::grow() ignores this point and directly uses the partial_key stored in keys_ as the index, causing an offset in indexing when the node grows:
template <class T> inner_node<T> *node_16<T>::grow() {
//...
for (int i = 0; i < n_children_; ++i) {
new_node->indexes_[(uint8_t) this->keys_[i]] = i;
} //...
}
For example, consider a node_16 instance:
node_16:
keys_: [a, b, ...]
ASCII: [97, 98, ...]
node_16<T>::grow()
indexes_: [..., 0, ..., EMPTY, ...]
wrong [97] correct[97+128]
Reviewers
No reviews
Still in progress? Convert to draft
Assignees
No one assigned
Labels
None yet
Projects
None yet
Milestone
No milestone
Development
Successfully merging this pull request may close these issues.
None yet
Notifications
Customize
Unsubscribe

API

```
class Database {
    using postings = std::vector<std::pair<int, int>>;
    void readDoc(string file);
    void readStopWord(string file);
    void readDocList(string filelist);
    void queryWord(string word, ostream &out);
    bool checkQueryWord(string word, shared_ptr<postings> posting);
    void queryList(string wordfile, ostream &out);
    void showInfo(ostream &out);
    void showTerms(ostream &out);
}
```

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
```

CPU Cache Size

L1: 480KB, L2: 7.5MB, L3: 18.0MB

ART 树最大节点 2MB，能够十分高效地利用 Cache 进行节点操作.

测试数据集

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：实验要求的数据集.
  - MS MACRO：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、3000、5000、10000 词.
  - 常见 7 字符词：1371 词.
  - 全部 7 字符词：40093 词.
- 停用词：并集为 471 词
  - MyISAM：465 词
  - InnoDB：36 词

MS MACRO 数据集预处理

MS MACRO 数据集是单个大文件，每一行一篇文章. 下面是预处理步骤：

- 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.

右侧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: Collection

```
1 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: FullDocs

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.

对照实现：Red-Black Tree & HashTable

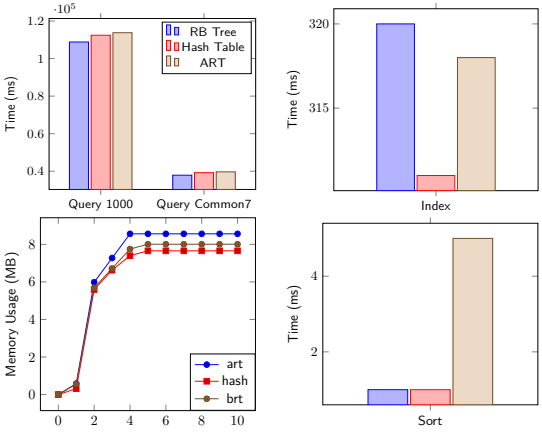
```
class Database {
using postings = vector<pair<int, int>>;
#if defined(ART)
    art::art<shared_ptr<postings>>
    invert_index_;
#elif defined(RBT)
    map<string, shared_ptr<postings>>
    invert_index_;
#elif defined(HASH)
    unordered_map<string, shared_ptr<
    postings>> invert_index_;
#endif
#ifdef CHECK
    vector<string> docIDs_;
#else
    int docIDs_ = 0;
#endif
    set<string> stop_words_;
}
```

```
void Database::readDoc(string file){
    ifstream in(file);
    int docID = docIDs_.size();
    docIDs_.push_back(file);
    map<string, int> word_count;
    for (string str; in >> str;) {
        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].
    push_back({docID, word.second});
}
```

理论分析

类型	数据结构	查询	插入
`std::map`	红黑树	$O(\log n)$	$O(\log n)$
`std::unordered_map`	哈希表	$O(1)$	$O(\log n)$
`art::art`	ART 树	$O(\log k)$	$O(\log k)$

在莎士比亚文集上的表现（查询时校验）

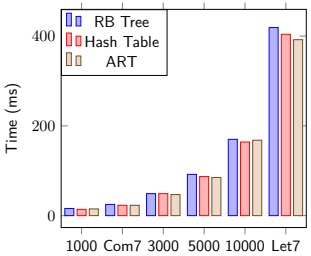


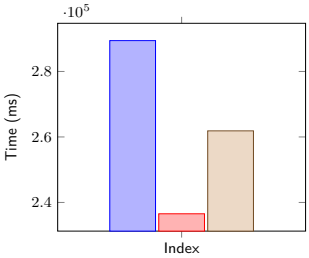
- 查询：无法体现，主要耗时在 I/O（校对文件、输出完整 Posting）. 1000 命中率高，故耗时比 Common7 高.
- 索引：哈希表最快，ART 次之，红黑树最慢.
- 内存：ART 树较大，因为不能完全利用 Node48.
- 排序：Posting 排序长度和算法相同，本质上测试的是迭代器速度. ART 库的迭代器使用栈实现，未进行针对性优化，显著劣于标准库.

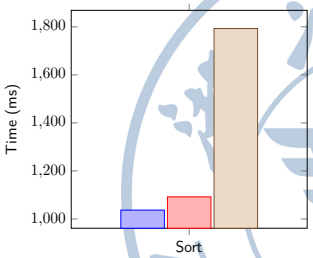
看起来 ART 树表现不佳？

索引树太小了，让我们看看后面的数据集.

在 Collection 上的表现



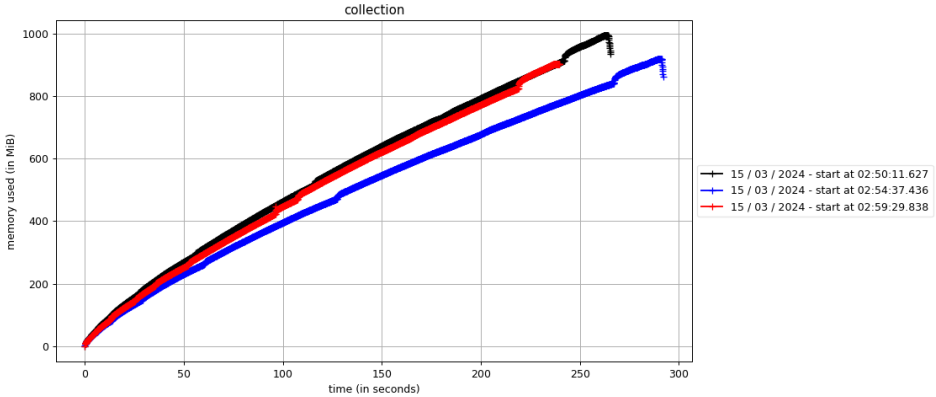




- 查询：三者中最快
- 索引：与红黑树拉开差距
- 排序：迭代器性能不佳

在 Collection 上的表现（mprof）

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．图例：ART 树（Black）哈希表（Red）红黑树（Blue）.

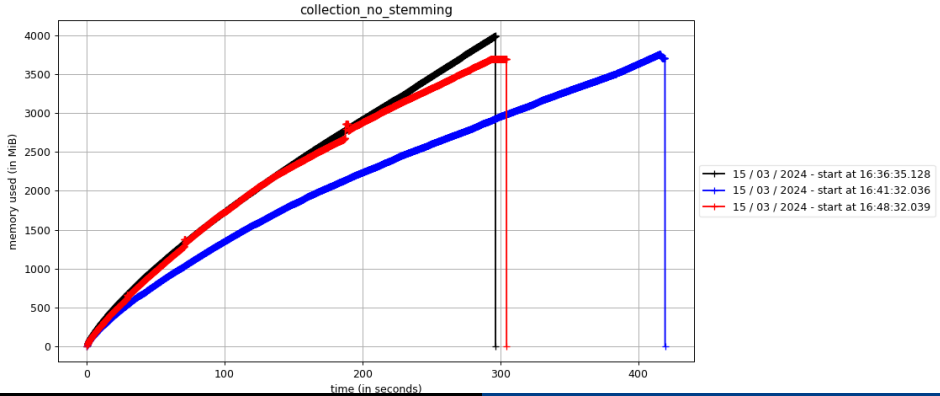


关于内存利用率优化的思考

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.

附加测试：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）.

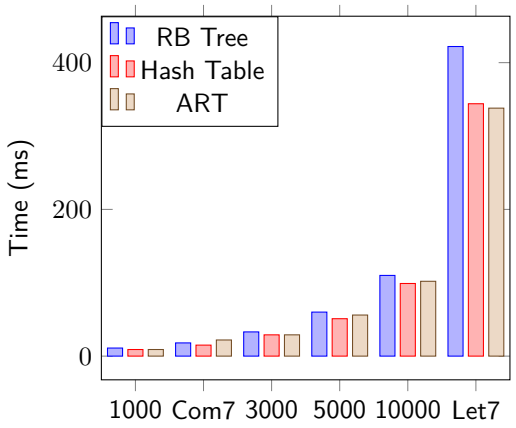


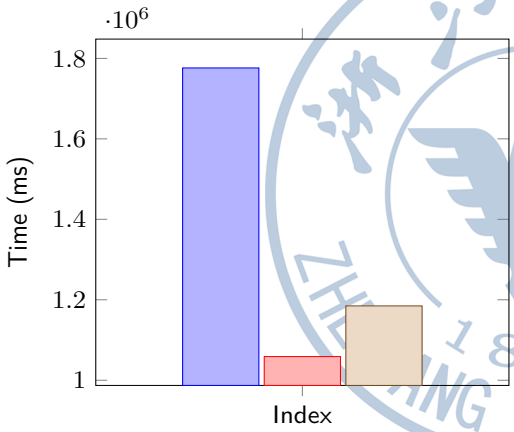
附赠：新的 Bug

```
bowling233@bowling-desktop ~/ads_p1 (main)> make run                                    (base)
clang++ -std=c++11 -Wall -pedantic -I. -O3 -DBENCHMARK -DNO_STEMMING -o ads_p1_art main.cpp porter2_stemmer.o -DART
clang++ -std=c++11 -Wall -pedantic -I. -O3 -DBENCHMARK -DNO_STEMMING -o ads_p1_hash main.cpp porter2_stemmer.o -DHASH
clang++ -std=c++11 -Wall -pedantic -I. -O3 -DBENCHMARK -DNO_STEMMING -o ads_p1_rbt main.cpp porter2_stemmer.o -DRBT
if [ ! -d result ]; then mkdir result; fi
echo collection
collection
mprof run ./ads_p1_art collection < in.collection 2>&1 > result/art_collection.out
terminate called after throwing an instance of 'std::out_of_range'
  what():  provided partial key does not have a successor
```

还是 `node_48<T>` 的偏移量造成的．关闭 stemmer 后有 Unicode 字符进入，`next_partial_key` 遍历到达末尾后抛出异常，因此该处的键（ASCII 127：DEL）无法存储．使用一般文本测试难以发现该问题．

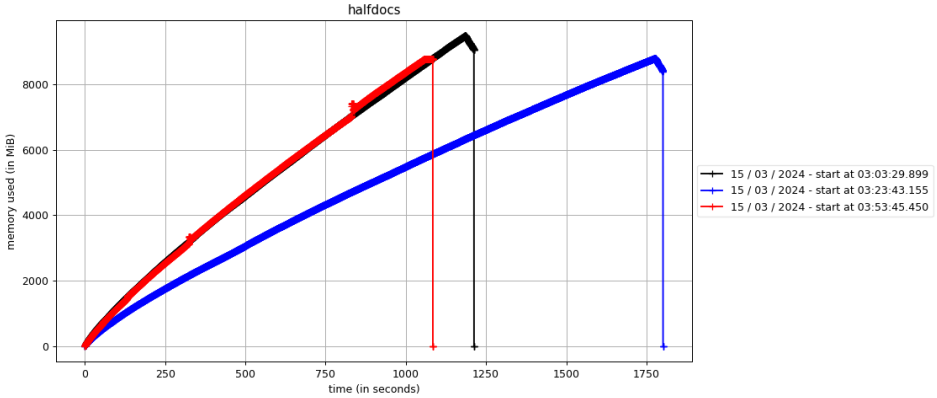
在 FullDocs（half）上的表现





在 FullDocs（half）上的表现（mprof）

图例：ART 树（Black）哈希表（Red）红黑树（Blue）.



在 FullDocs（half）上的表现分析

- 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.

展望

- 键值较大时，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.

心得

- 通过本次实验，我们对现代高性能计算的数据结构和算法有了更深入的了解.
- 通过学习和调试开源库，对编程语言和数据结构与算法的理解能力得到了提升，对优秀数据结构和算法的实现有了更深刻的认识.

参考文献

- Leis, V., Kemper, A., & Neumann, T. (2013). The adaptive radix tree: ARTful indexing for main-memory databases. 2013 IEEE 29th International Conference on Data Engineering (ICDE), 38–49. https://doi.org/10.1109/ICDE.2013.6544812
- Bajaj, P., Campos, D., Craswell, N., Deng, L., Gao, J., Liu, X., Majumder, R., McNamara, A., Mitra, B., Nguyen, T., Rosenberg, M., Song, X., Stoica, A., Tiwary, S., & Wang, T. (2018). MS MARCO: A Human Generated MAchine Reading COmprehension Dataset.
- **完整参考文献见实验报告**