南開大學
Nankai University

计算机学院

倒排索引求交算法的探究

姓名：田佳业

学号：2013599

专业：计算机科学与技术

目录

1 总述

该工作完成了在对倒排索引求交的串行算法进行入探究的基础上，使用 SIMD 进行可向量化部分(搜索和求交部分均有) 的加速，并使用 Pthread,OpenMP,MPI 进行了任务分配和算法上的并行化。其中 Pthread 由于能够显式的进行自定义的并行，对此进行了较详细的探究。同时对利用 GPU 进行加速也进行了探讨。

工作的亮点主要有以下几个方面：

1. 对倒排索引求交问题本身进行了细致和深入的探究。在基本算法的基础上，对更多的思路进行了探索，进行算法和体系结构上的优化。同时根据实际情况，提出了基于数值的双端搜索算法，并选择了合适的搜索策略，为数据量 (DocID) 较大时的倒排索引求交提供了新的方法。

2. 并行程序设计的过程中，不拘泥于简单的数据划分和任务划分，而是根据不同的并行模型尝试了不同的并行策略，并在并行程序设计中贯彻提前停止的思想，取得了较好的效果。

3. 以程序执行过程和结果为导向进行优化。综合运用数据分析、性能测试等方法进行拓展，思路清晰。

2 问题描述

2.1 问题背景

在当前的信息检索系统中，倒排索引作为一种简单高效的数据结构，被广泛应用于网页数据的存储和查询中。

我们可以将网络中的一个文档或者网页（Document）看作一组词的序列。而整个数据集中含有若干文档, 每一个文档都被赋予了一个从集合 1,2...,U 中选取的唯一的文档编号 (DocID), 其中 U 是整个数据集中的文档个数。这个集合通常按文档编号升序排列为一个升序列表，即称为倒排列表（Posting List）。所有词项的倒排列表组合起来就构成了整个数据集的倒排索引。

倒排列表求交 (List Intersection) 也称表求交或者集合求交, 当用户提交了一个 k 个词的查询, 查询词分别是 $t_1, t_2, \ldots, t_k$, 表求交算法返回 $\cap_{1\leq i\leq k}\ell(t_i)$。首先, 求交会首先按照倒排列表的长度对列表进行升序排序, 使得:

$$|l(t_1)| \leq |l(t_2)| \leq \cdots \leq |l(t_k)|$$

例如当用户搜索”2022 final cup” 时, 搜索引擎首先找在索引中出这个三个词“2022”, “final”, “cup” 所对应的索引表，如下图所示，然后求交返回三个索引表中的公共元素。

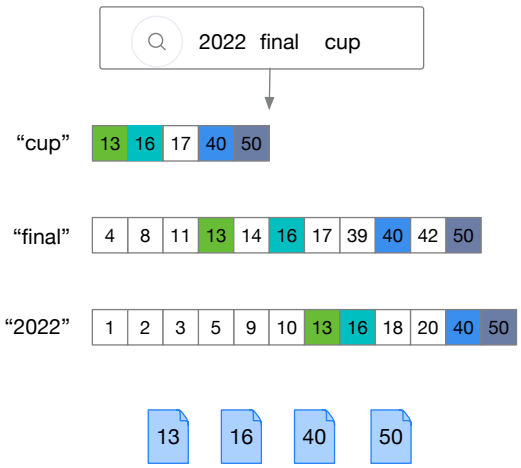


图 2.1: 倒排索引搜索示例

2.2 算法分析

2.2.1 求交算法

下面对目前较为常用的求交算法思想进行简要叙述。在后续的算法实践中将会介绍在工作中探索得到的更具有创新性、独特性的策略。

链表求交主要有两种方式按表求交（list-wise intersection）和按元素求交（element-wise intersection。

按表求交基本思想是：先使用两个表进行求交，得到中间结果再和第三条表求交，依次类推直到求交结束。这样求交的好处是，每一轮求交之后的结果都将变少，因此在接下来的求交中，计算量也将更少。这种策略被称作 zip 策略。[1]

其中，按表求交最典型的算法是 set_vs_set(SvS) 算法。对于 $\ell(t_1)$ 依次与其他元素求交，求交的过程中如果发现失配，那么就将失配元素从 $\ell(t_1)$ 中删去。最后 $\ell(t_1)$ 中元素即为所有元素的交集。

按元素求交算法会整体的处理所有的升序列表，每次得到全部倒排表中的一个交集元素。这类算法通常在 DAAT（Document at a time）查询下使用。

同时按表求交还有一种更简洁的方法：“拉链法”。在 c++ STL 库中的集合求交模板 `set_intersection` 就是用到这种方法 [2]。

其原理和归并排序类似。具体操作是：

两个指针指向首元素，比较元素的大小：

（1）如果相同，放入结果集，随意移动一个指针；

（2）否则，移动值较小的一个指针，直到链表尾。

Adaptive 算法（Adp）同样是由 Demaine 等人提出 [3] 的，也是其中比较典型，应用较广泛的算法。

经典的 Adp 算法从剩余元素最少的表中选择关键元素，下面也列出了 Adp 算法的伪代码。在这个算法中，各个索引表首先按照表长升序排列，并且选择第一个表中的第一个元素为关键元素。然后在每一个表中都去找这个元素。如果在搜索的过程中发现了不匹配，按照每一个表内剩余元素的个数

重新排序，并且从剩余元素数最少的表内选取关键元素继续查找。当然，如果在所有列表中都发现了某个元素，那么这个元素就在交集的集合中，放入结果集。当某一个表元素全部检测完毕，算法终止。

Algorithm 1 Adaptive 算法 (Adp)

Input: $\ell(t_1), \ell(t_2), \cdots, \ell(t_k)$, sorted by $|\ell(t_1)| \leq |\ell(t_2)| \leq \cdots \leq |\ell(t_k)|$

Output: $\cap_{1 \leq i \leq k} \boldsymbol{l}(t_i)$

```
set S ← ∅
while no list is empty do
    Sort the lists by increasing number of remaining elements, the subscript of each list is reordered to satisfy the above condition
    set e ← first undetected element of ℓ(t_1)
    set s ← 2
    repeat
        Search for e in ℓ(t_s)
        set s ← s + 1
    until s = k or e is not found in ℓ(t_s)
    if s = k and e is found in ℓ(t_s) then
        add e to S
    end if
end while
    return S
```

此外，按元素求交的算法还有 Barbay 等人提出的 Sequential 算法和 Culpepper 等人提出的 Max Successor 算法等。这些算法的都是使用某一个关键元素循环地在各个表中进行搜索。区别仅在于每次执行搜索算法前如何选取关键元素和循环的方式。

Sequential 算法是从当前表选择关键元素以避免寻找剩余元素最少的表带来的开销。关键元素以循环的方式在每一个表内执行搜索。一旦发生不匹配，选择当前表不匹配元素为关键元素，继续以循环的方式在各个表内执行搜索。当然也可以退化为 “守株待兔”，发生不匹配后继续对 $\ell(t_1)$ 中的下一元素进行查找。这种 “守株待兔” 的策略其实就是下面在算法实践中将要介绍的 simplified_Adp 算法。

而 Max Successor 算法在 Sequential 算法的基础上进一步改进，综合了 Sequential 算法和 simplified_Adp 算法，将当前失配元素和最短表的后继元素中的较大者作为关键元素。以期望要么尽快遍历完 DocID 所有元素，要么尽快把第一个表中的所有元素找完。

下面也给出了 Sequential 算法的伪代码供参考。而 Max Successor 算法仅是在 Sequential 的基础上添加了分支跳转，且会在接下来的实践中进行分析，伪代码在此也不再赘述。

Algorithm 2 Sequential 算法 (Seq)

Input: $\ell(t_1), \ell(t_2), \cdots, \ell(t_k)$, sorted by $|\ell(t_1)| \leq |\ell(t_2)| \leq \cdots \leq |\ell(t_k)|$

Output: $\cap_{1 \leq i \leq k} \boldsymbol{l}(t_i)$

```
set S ← ∅
set s ← 2
set e ← first undetected element of ℓ(t_1)
set p ← 1
while e is defined do
    Search for e in ℓ(t_s)
```

```
    if e is found in ℓ(t_s) then
        set p ← p + 1
        if p = k then
            add e to S
        end if
    end if
    if e is not found in ℓ(t_s) or p = k then
        set e to be the seccessor element in ℓ(t_s)
        set p ← 0
    end if
    set s ← s + 1 mod k
end whilereturn S
```

2.2.2 搜索算法

对于倒排链表内部的搜索策略，有如下几种方法：

• 线性搜索

最朴素的查找方法，比较次数时间复杂度 $O(n)$。

• 二分查找

需要 $O(\log n)$ 次比较。但由于每一次比较的结果都是依赖于数据的，因此也会最多发生 $O(\log n)$ 次分支预测失败。

• 索引分块查找

建立索引进行分段，每一段内进行线性搜索。分段方式和分块大小可以根据实际情况进行选择。

通过下面将要介绍的算法实践我们可以看到，不同的搜索策略对算法执行的效率有着决定性的作用，也是影响倒排索引求交性能的核心所在。在选择搜索策略的时候需要综合算法的复杂度和实际情况进行分析。

2.2.3 数据集介绍

给定的数据集是一个截取自 GOV2 数据集的子集，格式如下：

1) ExpIndex 是二进制倒排索引文件，所有数据均为四字节无符号整数（小端）。格式为：[数组 1] 长度，[数组 1]，[数组 2] 长度，[数组 2]....

2) ExpQuery 是文本文件，文件内每一行为一条查询记录；行中的每个数字对应索引文件的数组下标（term 编号）。

其中 ExpIndex 约 300M 左右，虽然无法和实际的搜索引擎存储量相比，但足以对我们算法的合理设计提出挑战。

通过对给出的数据进行初步分析，我们能够看到倒排链表库中包含 1,756 个关键词，每个关键词包含的文档数目长短不一且差距较大，最多不超过 30,000 个文档。文档的个数多达 25,205,174 个。每次搜索大概只会检索两三个词，最长的也不过有 5 个词，符合日常搜索的实际情况。

对数据集和查询实际情况的分析，也是对算法进行优化的重要方向。

3 算法层面的探索实践

3.1 性能评价约定

程序中通过指定 QueryNum[1,999] 进行规模的调整。之前实验中为了程序测试和执行的方便，均采用的是查询数量为 500 进行性能测试。考虑消除数据局部性的影响，在总报告中对所有设及的算法和程序进行了重新测试。因此约定本文中提到的执行时间均为全数据集上（查询数量为 1000）时的结果，重复执行三次取平均值。在不同平台上的测试结果会以 ARM 和 x86 进行标识。

平台信息：x86 平台为 CPU 为 Intel i7 6 核；ARM 平台为苹果 M1 芯片。

3.2 按表求交算法

3.2.1 基于查找的两表求交

在整个选题，首先完成的是 SvS 算法。最先进行的便是对 SvS 算法具体实现的优化。经典的 SvS 是先将 $\ell(t_1)$ 置为结果集，然后发现不匹配的元素便从中删除。但事实上对不匹配的元素删除代价是非常高的。首先倒排链表中成为交集的元素占其中的极少数，需要大量的对元素进行删除操作；而倒排链表是又是以向量的方式存储的，数据结构本身的缺陷也使得这一情况雪上加霜。

因此在 SvS 的实现上采用了这样的策略：在两两求交的过程中新建一个表, 用于存储得到的结果，然后再用新的表继续与其他的表进行求交。

另一个对于 SvS 的优化是对于搜索策略的优化。查找方法上，此处使用的是二分查找，以方便对算法核心思想的分析。后面中会对查找策略进行进一步的讨论和改进。在二分查找的过程中不管找到与否都能得到不小于这个元素的最小的元素，即高低两个指针最终相遇的地方。利用这一点和列表的有序性可以大大缩小每次查找的范围。即每次把该表查找到或失配的位置记录下来，下一次再从此处开始找。

通过以上的分析，我对基于查找的 SvS 算法进行了优化实现。核心代码可参照 SvS 优化算法代码。

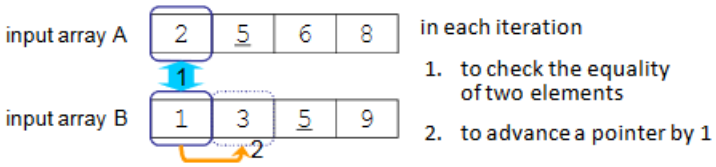
在 x86 平台上得到的性能为 1139.201ms。对经典的 SvS 算法也进行了实现，1000 次的查询需要 1542.158ms。可以看到仅仅是搜索策略的优化就达到了四分之一的加速，搜索算法的对性能的影响已经初见端倪。在 ARM 平台上对算法也进行了实践。

3.2.2 基于双指针的两表求交

在数据结构课程学习链表的过程中，我们接触了双指针的一些使用技巧。如算法设计中所讲，此方法可以说是所有算法中实现最简洁的。实现的代码见zip-zap 算法代码。

若假设两个求交的链表的长度分别是 m 和 n, 基于二分查找的复杂度为 $O(mlogn)$, 而基于双指针的情况下最坏只需要比较 $m+n$ 次，即复杂度为 $O(m+n)$。

与此同时，在并行程序设计的课程中，针对体系结构进行优化是非常重要的理念。我们对这个串行算法进行分析，能够感受到对于上述 `else if` 部分是比较难以进行分支预测的，因此我们此处可以使用 if 转换的 trick 进行优化，使用数据流代替控制流。这一点是在 [4] 的工作中得到的启示。基于以上分析，我们可以将上述 `while` 循环内的部分进行修改：




```
1  while (Apos < Aend && Bpos < Bend) {
2     if (A[Apos] == B[Bpos]) {
3        C[Cpos++] = A[Apos];
4        Apos++; Bpos++;
5     }
6     else if (A[Apos] > B[Bpos]) {
7        Bpos++;
8     }
9     else {
10       Apos++;
11    }
12 }
```

图 3.2: 优化前

```
1  while (Apos < Aend && Bpos < Bend) {
2     Adata = A[Apos];
3     Bdata = B[Bpos];
4     if (Adata == Bdata) {  // easy-to-predict branch
5        C[Cpos++] = Adata;
6        Apos++; Bpos++;
7     }
8     else { // advance pointers without conditional branches
9        Apos += (Adata < Bdata);
10       Bpos += (Bdata < Adata);
11    }
12 }
```

图 3.3: 优化后

对于按表求交而言，基于双指针的算法简单直观，突破了传统 SvS 算法“搜索”的窠臼。并且我们能够看到，比起前面实验中基于搜索的 SVS 算法，速度已经提升了接近一半。但是通过使用 SIMD，还可以继续进行优化。

对于 SIMD，容易想到我们可以由一次比较两表中对应各一个元素，转变为一次比较各四个元素。因此我们可以从以下问题开始：如何在两个短排序数组中查找和提取公共元素。当然，这时候我们需要对这两个表中元素的组合进行比较，而不只是直接比较。由于表是有序的，只需要比较四个组合。思路的具体实践参考了 Ilya Katsov 的博客 [5]。SSE 指令集提供了 `_mm_shuffle` 指令，可以让我们对向量内的元素进行重排。具体实现上如下图所示：我们可以对两个各由 4 个 32 位整数组成的段进行成对比较，产生一个位掩码，突出显示相等元素的位置。如果有两个 4 元素的寄存器 A 和 B，就有可能得到一个共同元素的掩码，将 A 与 B 的不同循环移位进行比较（下图的左边部分），将每次比较产生的掩码进行 OR（下图的右边部分）。

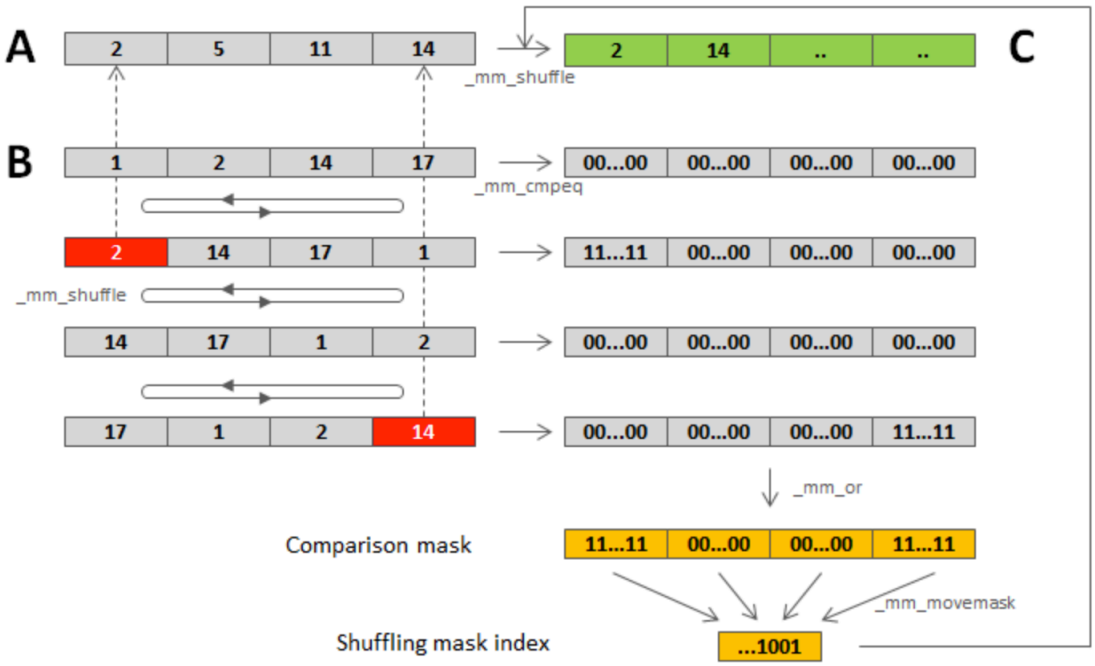


图 3.4: 两个短排序数组的求交

对于更长的表的求交，我们很容易参考串行算法，取两个表当前比较组中的最大元素进行比较，移动其中的较小着对应的链表的指针，一次可以移动 4 位。示意图如下所示：

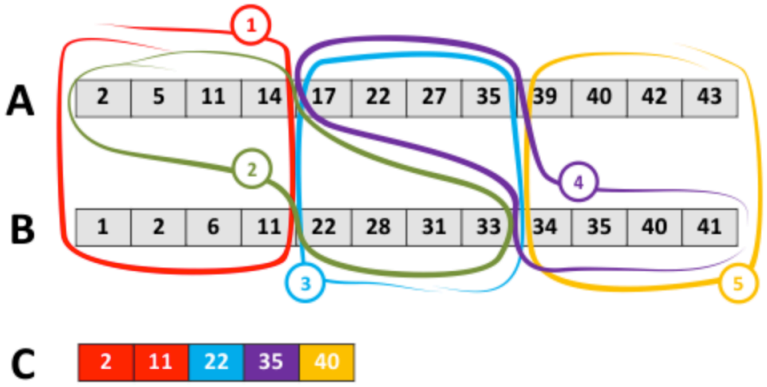


图 3.5: SIMD 求交整体过程

这里对上面的例子进行简要解释：在第一个循环中，比较了前 4 个元素的片段（红色）并复制出共同的元素（2 和 11）。与串行算法类似，向前移动列表 B 的指针，因为比较段的尾元素在 B 中较小（11 对 14）。在第二个循环中（绿色），将 A 的第一段与 B 的第二段进行比较，交集为空。我们将指针移动到 A。依此类推。上面的例子中，我们需要 5 次比较来处理两个列表，每个列表有 12 个元素。

这一部分的核心代码可以参见 zip-zap SIMD 算法代码。

3.2.3 性能分析

作为 baseline，使用 STL 库的 `set_intersection` 模板得到的性能为：x86：644.745ms；ARM：639.56ms。可以看到，基于查找的 SvS 相比模板的性能差距较大。而自己实现的双指针算法比标准库中的双指针稍好一点。主要是由于少了调用模板前对向量进行的 `resize` 等操作，没有算法实现优劣上的差距。

而对双指针进行的 SIMD 优化，在 x86 上性能提升了 100 多 ms。ARM 平台的 SIMD 由于对 `shuffle` 和 `mask` 操作支持并不好，所以并没有在 ARM 平台进行双指针的 SIMD 算法。尽管 SIMD 实现中控制流比串行算法多了一些，但仍旧受益于比较次数的减少和向量寄存器内联函数的速度优势，取得了较明显的提升。当然，这还不是按表求交的最佳策略。后续我们能看到，基于值的求交算法不仅能在按元素求交中取得很好的效果，即便是将其运用于按表求交，性能也是非常好的。

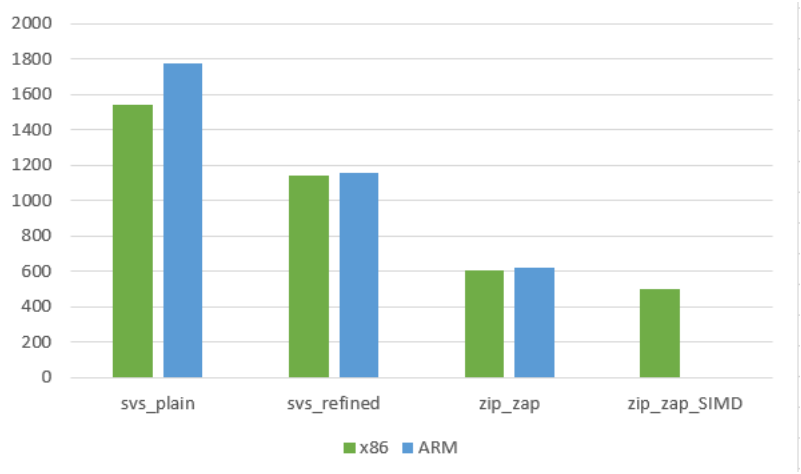


图 3.6: 按表求交算法性能比较

3.3 按元素求交算法

3.3.1 Adp 算法

相比于按表求交的 SvS 算法，按元素求交便不能使用双指针这样的简便方法了，不得不使用基于查找的方法，在“怎么找”上下功夫。

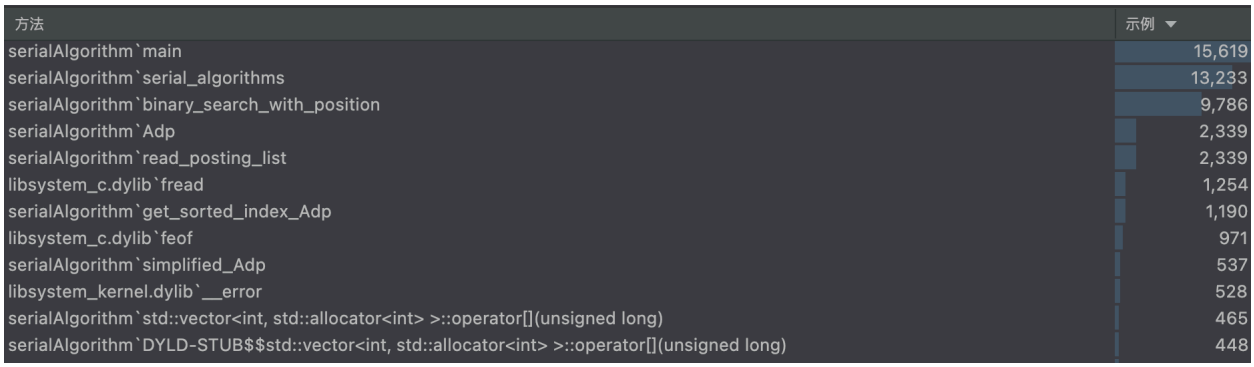
在编程实践的时候考虑按表求交到按元素求交的过渡，首先进行的其实是简化 Adp 算法的编写。事实上经过后面的分析很大程度上在复杂度和性能上表现比经典 Adp 更好，不失为“简化”。

简化 Adp 算法是在按元素求交的基础上，保留了 SvS 只从 $\ell(t_1)$ 中选取元素的特点。当然，保留了 Adp 算法停止条件的判断：如果发现当前关键元素下面寻找的表的最后一个元素都要大，那么说明这个表被找完了，停止算法。这些停止条件，对于 simplified_Adp 算法是额外的，但是能够使算法提前停止，提高效率。

对于倒排链表的排序算法，由于表数量少而“体积”大采用的是针对数组下标的插入排序。具体的实现可参照 简化 Adp 算法代码。

而 Adp 算法其实和 simplified_Adp 算法很相似，差别仅在于是否重新排序。

simplified_Adp 算法在 x86 上的性能为 618.397 ms，而传统 Adp 算法却达到了 1682.822 ms。是否重新排序的差别这么大吗？我利用利用 Clion 的 profiler 对其中排序的程序性能进行了分析：



方法	示例
serialAlgorithm`main	15,619
serialAlgorithm`serial_algorithms	13,233
serialAlgorithm`binary_search_with_position	9,786
serialAlgorithm`Adp	2,339
serialAlgorithm`read_posting_list	2,339
libsystem_c.dylib`fread	1,254
serialAlgorithm`get_sorted_index_Adp	1,190
libsystem_c.dylib`feof	971
serialAlgorithm`simplified_Adp	537
libsystem_kernel.dylib`__error	528
serialAlgorithm`std::vector<int, std::allocator<int> >::operator[](unsigned long)	465
serialAlgorithm`DYLD-STUB$$std::vector<int, std::allocator<int> >::operator[](unsigned long)	448

图 3.7: Adp 排序过程分析

其中 get_soreted_index_Adp 为 Adp 算法链表长度排序的函数。从中可以看到执行代价甚至超过了整个 simplified_Adp 算法!

我们可以考虑，实际上在关键元素变化的过程中，链表长度之间的关系并不会频繁的改变。而每一次对关键元素求交完后都按照经典算法所述进行排序，代价是非常大的。

当然，从 binary_research 占的比例再次印证了，倒排索引求交的过程中查找操作是影响算法性能的重要环节。

3.3.2 Sequential 算法和 Max_Succrssor 算法

和 sequential 算法提出的初衷一样，对于 sequential 算法是在 adp 算法代码的基础上进行的改进。取消了 adp 算法的每轮重排，并且增加记录上一轮关键元素是从哪一个倒排链表中选取的。在下一轮循环比较时可以直接跳过寻找该表。当然，在之前探索中实现的记录二分查找位置以及优化后的提前停止条件等优化方式也得到了延续。

max_successor 算法和 sequential 算法类似，只不过多了对当前元素和第一个表中元素的比较。在算法编写上也进行了多次 debug 和调优，尽量避免不必要的变量创建/销毁以及数据的比较。

核心代码可参见Sequential 算法代码。max_successor 算法主要是在 sequential 算法上的进一步改进。max_successor 算法在 sequential 算法的基础上增加的部分可以参见Max_Successor 算法代码

3.3.3 性能分析

下面展示了这两种算法与 simplified_Adp 算法的性能比较。

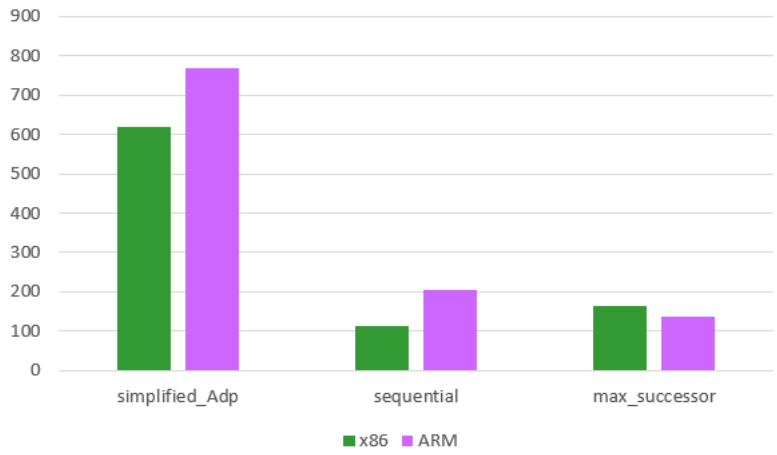


图 3.8: 三种按元素求交算法的性能比较

我们可以看到，在 x86 平台上 sequential 性能较好，而在 ARM 平台上 max_successor 算法更快。但实际上并没有证据表明哪一个更好一些。因为 sequential 算法虽然查找完成的循环次数可能比 max_successor 算法多一点，但也少了一些比较和判断的代价。

但是，这两种算法都相对于 Adp 算法有了较大的改进。显然，好的关键元素的选择策略，对于按元素求交算法来说，是非常重要的。“提前停止”的思想在按表求交的倒排索引算法中贯穿始终。

3.4 “增速提效”——查找优化

在进行了算法的基本实现之后，我们需要进一步挖掘影响倒排索引求交的“瓶颈”。通过实践部分的分析，其实已经意识到搜索策略的重要性。下面将会对其与性能的关系进行分析，并尝试做进一步的优化。

3.4.1 减少查找次数

从对 SvS 到 Adp 的改进再到改良算法（Sequential 和 Max successor）的实现，我们能看到每一次的改进，都伴随着性能近乎一倍的提升。而在各个算法的细节上，都注意了提前停止条件的应用。提前停止的直接目的，便是减少查找次数。下面对进行 1000 次查询的情况下各类算法的查找调用次数进行了统计：

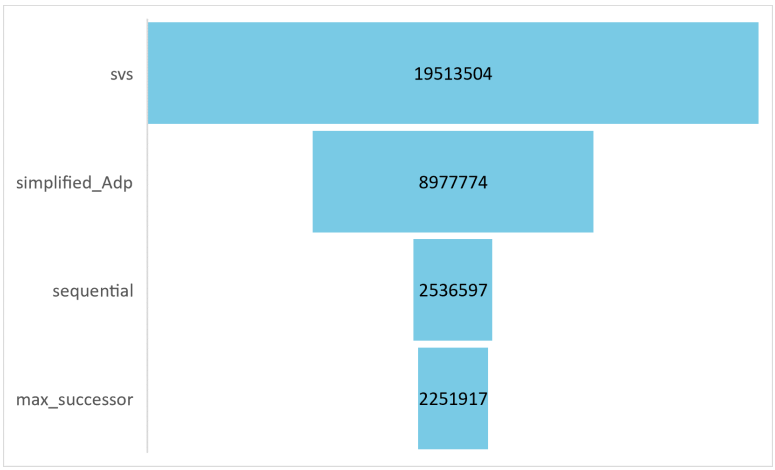


图 3.9: 查找次数的优化

可以看到，在减少查找调用次数上的努力，还是十分明显的，这也是性能提升的内在原因之一。

3.4.2 提高查找速度

在一开始的算法实践中均使用的是二分查找。它非常简结，而且时间复杂度为 $O(logn)$，对于最大长度为 30000 的链表，一定可以在 16 次比较之内就能得到不小于关键元素的元素所在的位置。

但是，二分查找一定就是最好的吗？未必。在这个问题中，简单的将二分查找换成 $O(n)$ 的顺序查找后（仍然记录失配位置），性能提高了至少 20%（max_successor 算法）。因此除了复杂度的分析，同样需要结合实际考虑。

这其实是是意料之外，情理之中的。二分查找虽然在一般情形下够快，但是它对于查找边缘中的元素并不友好。对于 $\frac{1}{2^n}$ 位置的元素，二分查找会很快命中，而对于最边上的元素，却总是最坏情况。而在这个问题中，数据的分布恰恰有相对的局部性：含有关键词的 DocID 往往是连续的。我随机选取了三个倒排链表，使用 python 进行数据读取并作出了其数据分布的直方图，如下所示。它们都表现出了局部性的特点。每次查找后记录查找位置以缩短以后的查找时间，在之前的实验中能够看到确实性能达到了成倍的提升，但也不是完美的：这种方法恰恰助长了上述“最坏情况”的发生。

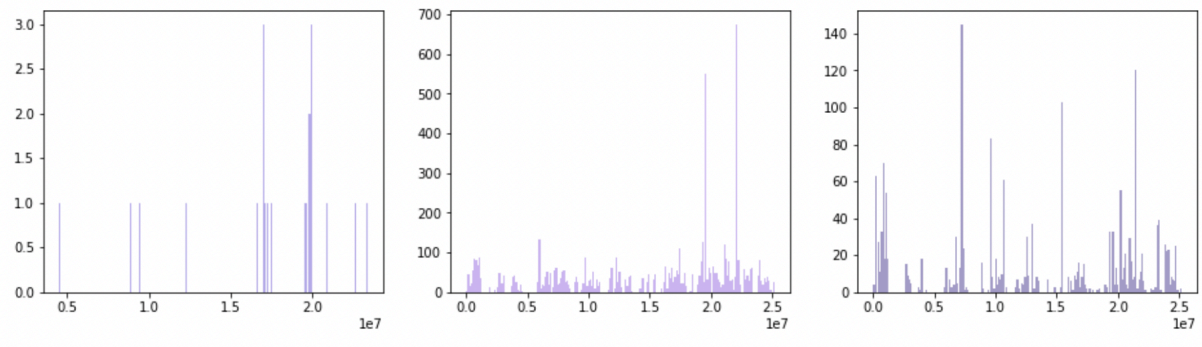


图 3.10: 三个倒排链表 DocID 数据分布直方图

从体系结构的方面我们也可以看到导致二分查找性能差的原因和优化的效果。

在鲲鹏服务器上使用 perf 对 max_successor 算法使用两种搜索策略进行了 cache 命中率的分析，结果如下图所示：

```
Samples: 9K of event 'L1-dcache-load-misses:u', Event count (approx.): 4222917
Overhead  Command  Shared Object        Symbol
  58.47%  max      libc-2.17.so         [.] _IO_fread
  38.15%  max      max                  [.] binary_search_with_position
```

图 3.11: 二分查找 cache miss 占比

```
  77.15%  max_ss   libc-2.17.so      [.] _IO_fread
  15.70%  max_ss   max_ss            [.] serial_search_with_location
   3.08%  max_ss   max_ss            [.] max_successor
```

图 3.12: 线性查找 cache miss 占比

由图所示，线性查找的 L1 cache 命中率较二分查找有明显提升。一方面减少了比较的次数，另一方面二分查找数据局部性相比顺序查找差很多。

顺水推舟，我进行了顺序查找的 SIMD 优化。由一次比较一个元素转变为一次可以比较四个元素。这个 SIMD 优化在 x86 和 ARM 平台上都进行了。当然由于两个平台上 SIMD 指令集的不同，实现方式上也有所不同。

下面以 max_successor 算法为例，展示不同的搜索算法对其性能的影响。

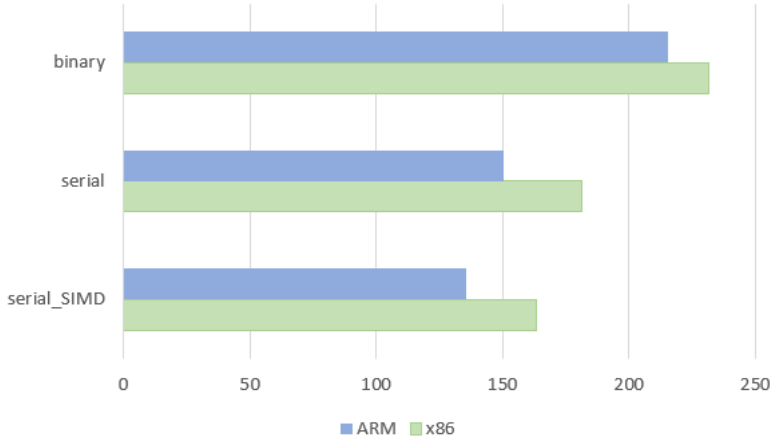


图 3.13: 不同搜索算法对 max_successor 性能的影响

3.4.3 哈希分块查找

基于此我也对哈希索引查找的尝试：在块内保持一定的局部性；同时由于最终成为交集的元素在关键元素中占很少数，使用哈希分块的方式能够更快的判断某个元素不在块内——只要其所在的哈希块内没有对应元素，便可以立即判断其不在块内。

因此，接下来尝试在已有的倒排列表中建立辅助索引来提高查询速度。查询文档可以根据辅助索引快速地获取它们在其它表中可能存在的位置范围。通常情况下，这个范围只是它所在倒排表的一小部分。也就是说，这种分段策略可以将查找过程限定在很小的范围内。结合上一段内所阐述的，应该能期待降低了查询比较的次数，提升算法的整体效率。

具体实现中，参考文献 [6], 首先将每个倒排列表划分为多个哈希段 (也称哈希桶)。对于倒排列表中的每个文档 d，它归属于哈希段 h 当且仅当 $h = h(d)$ 成立。其中 $h(x)$ 是分段哈希函数，负责计算

文档编号到其所属段号的映射。在他们的实现中，哈希函数定义为:

$$h(x) = \lfloor x/2^{k-p} \rfloor$$

其中 k 是满足 $U \leq 2^k$ 的最小正整数 (U 是数据集中最大的文档编号), P 满足条件 $p < k$ 。这个哈希函数的本质, 就是根据 doclD 的高 $(k-p)$ 位划分倒排列表, 同一段内文档的 doclD 仅最低 P 位不同。分段后段内元素仍按升序排列。可以看出, 如果某个文档在倒排列表中出现, 那么它一定位于对应的哈希段内。这种分段策略的突出优势有两点, 第一是获取某个 doclD 的段号仅需要完成一次移位计算, 时间复杂度为 $o(1)$ 。最坏的情况也不过是遍历所有块内的元素。第二是辅助索引只存储各段首元素的地址, 并不会占用太多的额外内存空间。

基于哈希的查找有一个很大的弱点, 即无法利用最近失配元素的位置进行进一步的查找。因此, 仅在不关心最近失配元素的 simplified_Adp 算法上对哈希查找进行了实验。

下面是随着 P 的变化，哈希求交的时间与 simplified_Adp 算法的比较:

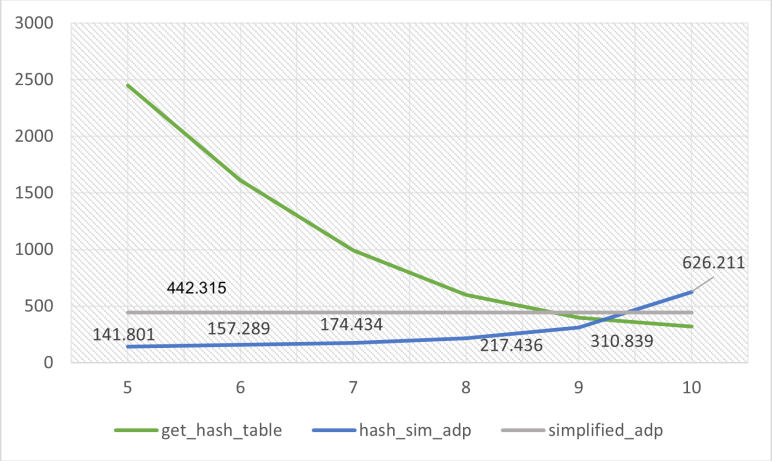


图 3.14: 哈希分块查找的性能分析

我们可以看到，P 越小即分的块越多，每个块内的元素越少。显然，越细致的块划分对于哈希查找越有利，相应的建立索引的时间就更长。当然，在倒排索引中比起索引预处理的时间，我们更关心的是直接关系到查询响应速度的求交时间，只要建立索引的时间不太过分，都是在可以接受的范围内的。在 P=5 时相比二分查找的 simplified_Adp 算法有了大约 3 倍的提升。但相比于 sequential 和 max_successor 算法还是比较逊色。

在分段方式上，也对采用取模运算的方式进行了尝试。本质上上面的方式是取模的一种特殊情况。而显然采用位与的方式计算效率要高一些。

同时在实验中，也考虑了基于位图的分块存储策略，将倒排链表中的元素映射到位图中，这样求交操作便变成了位与操作，求交复杂度为 $O(m)$，其中 m 为所有链表末尾元素中的最小者。加上和哈希分段类似的策略，只对存在 1 的段进行求交。如下图所示。

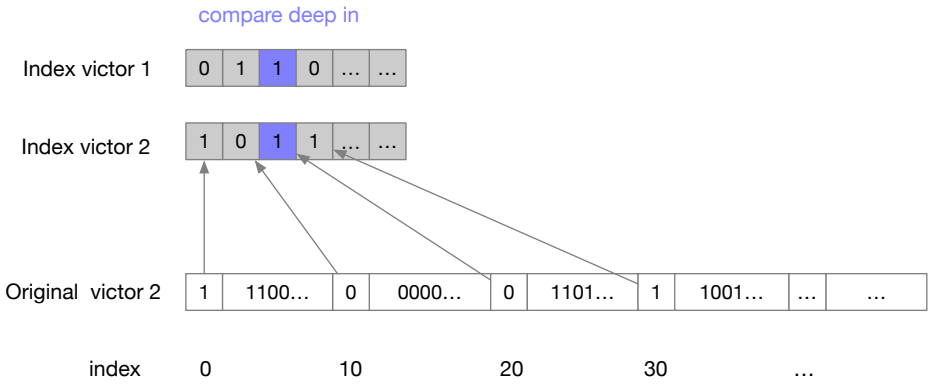


图 3.15: 基于位图存储的分段求交

想法很美好，现实很骨感。即便是尝试了使用 SIMD 加速，也和链表的存储方式差距非常的大。这是因为复杂度中的 m 实在是太大了。毕竟最大的 DocID 为 25,205,174。如此大数量级的位图长度，即便是位与，也是不小的开销。

下表为在 10 次查询时测量 3 次得到的平均值。

串行	SIMD	循环展开	索引	索引 + 块内 SIMD
393.194	377.301	1753.01	159.855	160.660

表 1: 基于位图的算法性能比较（单位：ms）

我们可以看到，即便是采用了分块 +SIMD，10 次查询所用的时间已经超过使用 max_successor 算法查询 1000 次的时间。

3.5 基于 DocID 的求交算法

3.5.1 问题分析

我还尝试了对倒排链表之间的数据分布尝试了可视化描述。对应的 Python 代码可在 Github 仓库查看。下图为编号为 3-6 的倒排链表的数据分布：

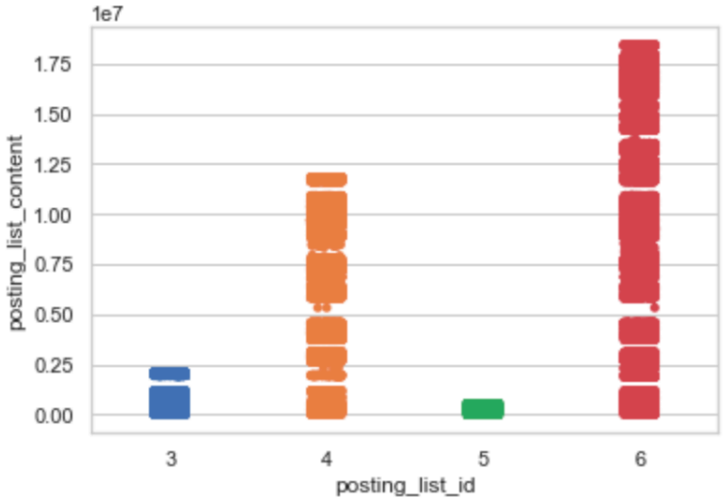


图 3.16: 四个倒排链表 DocID 数据分布关系

从中不仅可以看到数据的局部性，也可以看到各个倒排链表的数据大小相差很大。比如第 5 和第 6 个倒排链表的最后一个元素分别为 541631 和 18455635。这印证了基于元素大小的提前停止的必要性。

更重要的是，我们能够看到，各个倒排链表元素的分布范围也差距很大，比如第 5 个倒排链表的 DocID 分布范围就很小。因此萌生了一个想法：之前的算法出来 max_successor 算法以外，并没有充分的利用倒排链表中的数值特性。而 max_successor 的出色表现，也吸引着我去对针对数值特性进行算法设计。

3.5.2 算法思想

基于以上分析，提出了一种非常简洁激进的基于数值的算法：双端搜索。经过文献查找，目前还没有注意到对这种方法的研究。虽然不能完全确定其创新性，但确为独立思考得到的方法。

这种方法形象的讲类似于“两边夹”。每轮进行两个操作：

1）选择所有链表左端值最大的 DocID 对其他链表进行“左端查找”；“左端查找”返回每个链表不小于该 DocID 的元素位置，并使用头指针记录下来。

2）选择所有链表右端最小的 DocID 对其他链表进行“右端查找”；“右端查找”返回每个链表不大于该 DocID 的元素位置，并使用尾指针记录下来。

如果在此过程中找到所有链表中都出现的元素，加入到结果集中。直到某一个表的头指针大于尾指针为止。

当然，这种方法得到的结果集并不是有序的。不过，在倒排索引求交的问题中，结果集的 DocID 除标识文档外并没有其他的含义，如权重等。因此 DocID 是否有序对结果的有效性并没有影响。

3.5.3 算法实践

首先对采用这种方法的两表求交进行了实验，代码参见 基于值的按表求交。

当然，在得到两表求交结果后，需要对结果进行排序，再与第三个表求交。

通过二分查找的两种不同的编程方式，可以容易的分别实现“左端查找”和“右端查找”。x86 平台上 1000 次查询得到的结果是 361.331ms。这比使用 SIMD 优化的双指针法还要快 150 多 ms。(SIMD 优化的双指针法为 361.331ms)

由此可以看到这种方式针对搜索到实际情况，比双指针法机械的遍历链表中的每一个元素要快的多。即便是多了排序的代价，依然有很大的优势。

根据算法原理的分析，这种方式是适用多表求交上。当使用二分搜索时，x86 平台上的性能为 158.802 ms。可以看到，相比于 Adp 算法，已经非常快了，而距离 sequential 和 max_uccessor 算法还有一点差距。

首先进行简单的分析：对于这种方法，对表按照长度进行排序显得并没有必要。但是，对此进行测试发现，在 x86 平台上的结果为 160.406 ms。排不排序性能相差并不大。一方面是排序相较于对长链表的频繁查找工作并属于影响性能的瓶颈，另一方面由于数据的局部性分布，短的链表很大可能也会是数值分布范围窄的链表，放到前面可以在最后一轮中提前终止，但其对性能的影响也微乎其微。

另一方面，尝试和 max_successor 算法一样采用顺序搜索会不会取得更好的效果？经过尝试，得到的结果是 212.895 ms。比二分搜索慢了很多。

但是，思考顺序搜索在双端搜索算法中慢的原因，主要是当 DocID 表值域差距较大时，选择的数值会更靠中间，而在长的表中会被迫从两端进行搜索，顺序搜索时显然会因此连续搜索大量的元素，造成性能的下降。

自然的能想到一个好的解决方案：第一轮循环采用二分搜索，迅速减小所有表的值域范围；后面的循环使用顺序搜索，较好的运用数据的局部性。当然前面编写的顺序 SIMD 算法在此也进行了运用。得到的结果令我惊讶：98.435ms！(x86) 虽然领先 max_successor 算法并不多，但一定程度上也说明了工作的意义。

3.6 其他算法

3.6.1 混合算法

从对双端查找算法的实践中可以感受到，也许并不应该囿于某种单一的策略，而是可以尝试不同策略的组合。

一个直观的想法是，既然 max_successor 算法表现很好，双端搜索也表现的很好，可不可以将两者结合起来？当完成双端搜索的算法后，我十分有兴致的对此进行了尝试。但失望的发现效果并不好。在 x86 平台上的性能仅为 215.532 ms。比这两个算法中的任何一个都慢了不少。经过分析，可以发现原因主要有以下两个：

1）进行了多余的查找操作。

max_successor 通过一端的查找期望要么把第一个表找完，要么尽快找到比某一个表末尾元素大的元素。这种查找本身跳跃性就较强。虽然双端查找好像加快了首尾相遇的过程，也缩小了查找的范围，但不能忘记这是以首尾双倍的查找次数为代价的。通过对查找总次数的分析，可以看到 max_successor 算法为 2251917 次，而加入双端查找后却为 2759723 次。显然潜在的提前停止的可能并没有抵消查找次数的增加。

2）不能有效的更新链表左右端点。

max_successor 的策略是仅当失配前更新端点位置。而发现失配之后继续进行下一轮查找而不会对后面没有查找的链表长度进行更新。这就使得查找一直会在一个更大的范围内进行，不利于快速进行查找。

下面介绍的算法，也可能有潜力成为较好的算法，但目前还没有进行尝试：

3.6.2 small Adaptive 算法

在研读论文时，发现了一种被作者称作 small_Adaptive 的算法 [7]，它也是一种混合算法，像是结合了 SvS 和 Adaptive 的一种算法。

这个算法对于最小集合中的每个元素，它在第二小集合上执行快速搜索。如果找到一个公共元素，则在剩余的集合中执行新的搜索，以确定该元素是否确实在所有集合的交集中，否则执行新的搜索。得到的结果集同样是递增的。

3.6.3 Baeza Yates 算法

Baeza Yates 是一种用于两个排序列表求交的方法。它包含了二分和递归的思想。

首先获取较短链表的中间元素，并在较长列表中搜索它。如果该元素出现在较长的列表中，则将其添加到结果集中。较短链表的中位数和较长链表中位数的秩插入将问题分成两个子问题。递归地求解每对子集形成的实例，总是取较小子集的中值，并在较大的子集中搜索它。如果任何子集为空算法结束。当然，根据算法的思想，也可以选择值域而不是链表的长度为标准。

4 多线程/多进程编程实践

4.1 Pthread 并行优化

4.1.1 Query 间并行

Query 间并行进行了两种思路的任务划分。一种像是 OpenMP 一样的风格（虽然类比并不很妥当），在某一时刻开始指定并行执行，等这几个线程都执行完了再去进行下一轮的并行。这种方法明显的缺点就是涉及到大量线程的重新创建，线程等待的时间也较长。另一种便是典型的任务划分。将所有的查询每个线程尽可能平均分配。

query 数为 1000，ARM 平台上 sequential 算法进行了实验：

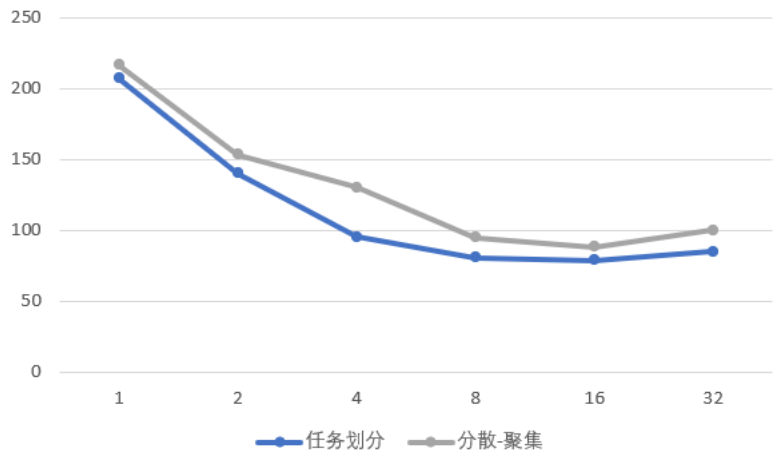


图 4.17: 线程个数对执行时间的影响

由于是在自己的笔记本电脑上进行的，因此对更多线程的划分进行了探索。但事实上在线程过多时反而会导致速度的减慢。毕竟线程的创建和销毁代价也是相当高的。不管再分多少线程，相对于同等算法的加速比也仅仅是 2 倍多一点的提升。而且当只有一个线程的时候，是比串行算法慢的。同时在汇总结果时也进行了显式的线程等待，因此效果并不好。

4.1.2 Query 内并行

相对 Query 间的并行，Query 内并行更能显著提高单个用户的响应速度，与算法上的优化可以完全等同而论。在这个选题中，创新性的利用上一次实验得到的 simplified_Adp 算法进行算法的并行化研究。

下面还将介绍对基本思路的一个重要优化。在基本思路中，对 simplified_Adp 算法从最短的倒排链表中选取关键元素这一过程进行任务划分，彼此是不相关的。整个算法结束取决于最慢的线程执行的时间。但是事实上提前停止的思路依旧可以应用到并行算法中。当发现提前停止的条件之后，我们不仅停止本线程的查找，也按照线程号，给在这个线程之后的线程们发送消息，通知他们停止，也就是说“你们线程块里肯定没有结果，就别找了”。这样虽然执行时间还是会取决于前面执行最长的线程的执行时间，但大大降低了分配不均的可能，也让系统少做了很多无用功。下面展示了这两种并行算法在 8 个线程时相对于串行算法的执行时间。

在 8 个线程的情况下 ARM 平台的结果为 329.567ms，相较于 simplified_Adp 算法已经有了 3 倍的提升。而且就加速比而言，针对算法的提前停止优化，效果是远远好于任务划分的。尽管这个性能相比 max_successor 算法和双端搜索算法还是有较大的差距。

同时，还有一种适用于 sequential 算法的 Pthread 并行思路。

基本思路是把最短表划分给多个线程，每个线程负责在其他表中搜索分配给自己的 DocID，在开始搜索之前可以通过对边界元素进行二分搜索划出搜索边界。

尽管思路比较清晰，但实施起来的细节是非常多的。在此用了较多的时间 debug。需要在分配线程前确定实际需要的线程数。因为如果按照第一个表均匀划分，多数情况下后面的元素在划分的时候就已经确定不会是交集中的元素了。然后需要根据实际需要的线程去构建包含分割元素位置及每一段的元素长度的表。每一个线程至少要维护自己负责的链表的起始位置和终止位置。同时还有一个重要的事情是：分割后的表的相对长度也发生了变化，需要对其进行重新排序。

在程序中也多处使用简化的表达式写法以尽量避免不必要的分支预测或重复计算。程序采用顺序 SIMD 搜索。

在 8 个线程的 ARM 平台上得到的结果是 122.230ms。尽管不如任务划分，但在 ARM 平台上已经超过了 max_successor 算法，表明了并行设计提高相应速度的可能。

4.2 OpenMP 和 MPI 并行优化

4.2.1 OpenMP

openMP 通过编译指示（pragma）告知编译器对程序进行并行化。相比于 pthread，我们可以将一些在 pthread 中需要显式说明的操作让编译器自动完成。同时也可以通过增加一些特定的编译指令实现对运算符规约化，以及更加合理的进行任务调度等。但是，并非所有“元素级”循环都能使用 openMP 进行并行化。。而在倒排索引求交的研究中，提前停止技术是对倒排索引求交优化中必不可少的一部分。也就注定了倒排索引的算法（特别是 Adp 及从属的算法）并不很适合使用 openMP 进行“自动化”的并行加速。

但在此次工作中仍旧尝试了使用 OpenMP 对程序进行了并行化。

通过添加 omp parallel for 编译指令，并指定在放置倒排链表结果时显式串行执行来进行自动化的并行执行。

在 openMP 中，线程的调度方式也会对性能产生影响。默认的方式是 static。而指定为 dynamic 后逻辑上形成一个任务池，包含所有迭代步，而每个线程都能近似“不停歇”的执行循环内任务。这

比 pthread 要更“智能”，在固定迭代次数，每个任务执行时间有差距时，通常能比静态划分方法取得更好的效果。

下面是在 ARM 平台上 simplified_Adp，500 queries 时不同调度方式的比较：

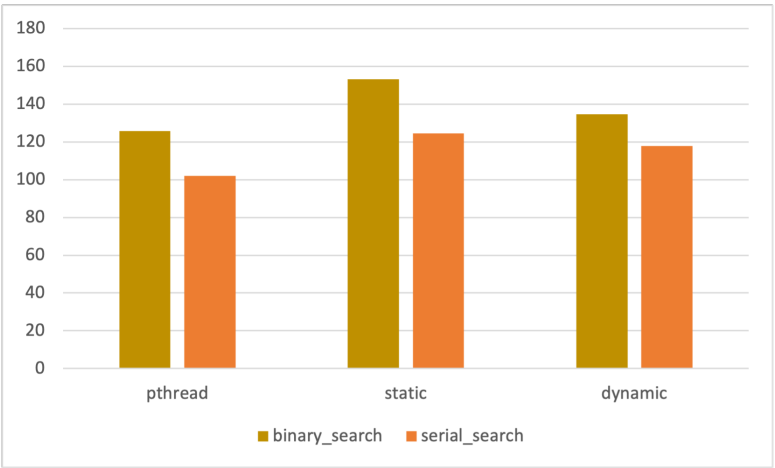


图 4.18: 不同调度方式的比较

由结果可以看到，OpenMP 单纯做任务划分的效果总体上和 pthread 相当，但略有差距。考虑到加速效果，并没有对 OpenMP 加速其他算法进行编写。

4.2.2 MPI

x86 环境安装 Microsoft MPI 并在 Clion 中进行配置。使用 mpiexec 进行 MPI 执行模拟；ARM 环境安装 openMPI 并在 Clion 中进行配置，使用 mpic++ 进行 mpi 执行模拟。

在 MPI 实验中对较多的方法进行了并行化研究，其中较有价值的是 max_successor 算法，结合 SIMD 的顺序查找策略的并行化。在 x86 平台上 8 个进程能将性能提至 35.924 ms。效果相当迅猛。

4.3 GPU 算法

通过查阅文献，得到一种较有潜力的 GPU 求交的思路、[8]。

首先类似于索引分块求交，对倒排索引进行预处理：

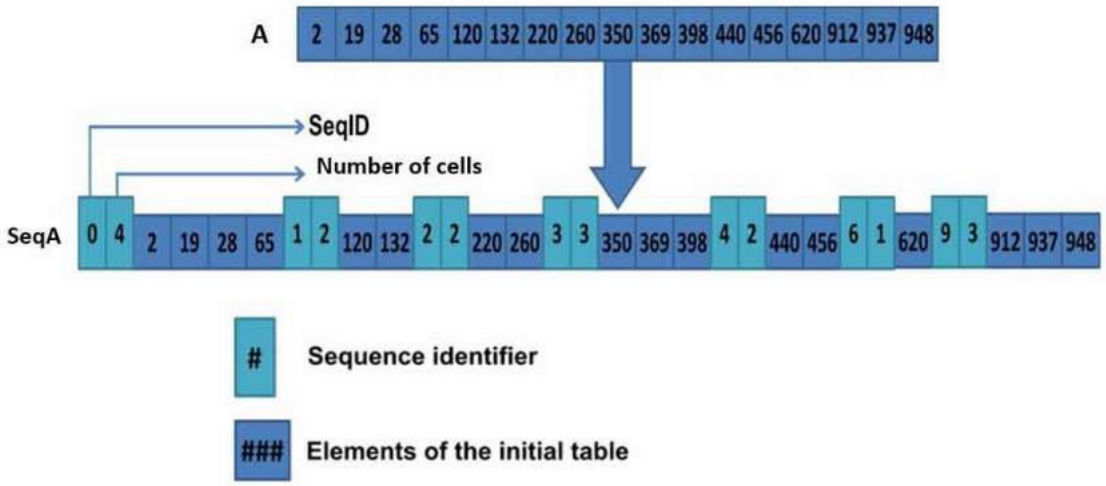


图 4.19: 预处理

列表最初由 CPU 读取，转换为序列，然后传输到 GPU 的全局内存。我们为每个序列分配了一个线程，该线程将比较 SeqID，并寻找具有相同 SeqID 的两个序列之间的公共元素。

每当两个 SeqIDs 相等时，相应的线程就搜索这两个序列之间的交集。如果第一个表的 SeqID 值高于第二个表的 SeqID 值，则线程转到第二个表中的下一个序列，并将其与当前序列的 SeqID 进行比较，否则结束。当两个表中的一个没有序列时，算法结束。

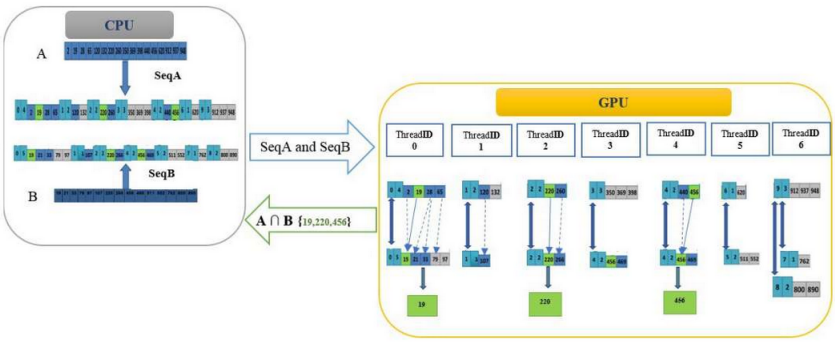


图 4.20: GPU 求交

由于自己的电脑没有显卡，之后尝试使用英伟达平台也没有成功，因此没能够对 GPU 算法进行实践。

5 总结

由以上的工作可以看到，对于倒排索引的求交的问题，如何根据实际情况进行算法和体系结构上的优化比单纯的调用并行模型的 API 重要的多，得到的性能提升也更为实在。整个实验中得到的性能最好的串行算法为双端搜索算法，结合二分搜索和顺序搜索的 SIMD。虽然 MPI 程序的并行效果是非常好的，但也仅仅局限于吞吐量上。

我从并行程序设计的课程中得到的收获很多。掌握的不仅仅是各种并行工具的范式，更是从算法体系结构方面对程序优化的思想。同时也是大学生活中第一次对一个具体课题从查阅文献到自主设计全方位的研究，为以后的科研也打下了基础。最后感谢老师和助教在课程上的付出！

6 程序测试

6.1 输入：

1) ExpIndex 是二进制倒排索引文件，所有数据均为四字节无符号整数（小端）。格式为：[数组 1] 长度，[数组 1]，[数组 2] 长度，[数组 2]....

2) ExpQuery 是文本文件，文件内每一行为一条查询记录；行中的每个数字对应索引文件的数组下标（term 编号）。

实际测试的时候只需要能够正确读取文件即可。

6.2 输出：

1）每个查询记录对应的倒排索引求交结果，存储在向量中。

2）求交过程执行的时间。

6.3 测试时需要注意的

1. 如果 readdata.h 两个文件的相对路径不起作用，可以将其替换为对应于测试用机的绝对路径。

2.QueryNum 的取值范围是 0-999。

3. 对于所有的求交结果，打印出了前 5 个查询的结果以供验证。可以在 test 模块修改以打印出更多测试结果。

6.4 程序演示

```
read_posting_list: 2890.523 ms
read_query_list: 0.761 ms
query_num: 500
result 0: 0

result 1: 0

result 2: 892
302 305 308 3791 3795 4117 4513 4524 4679 5033 6701 6702 7138 7873 8490 9244 10261 10265 13385
result 3: 766
15 113 117 118 131 139 200 1999 2000 2367 2463 2531 2535 2557 2634 3444 3520 3940 3946 4068 523
result 4: 211
536 5447 11076 17980 20228 27579 27638 42052 46158 103926 106009 106044 106054 106776 107193 10
max_successor: 87.705 ms
```

图 6.21: 程序演示

输出格式为：result+ 查询编号 + 交集中 DocID 数 + 交集中所有 DocID。

7 核心代码

7.1 SvS 优化算法代码

[参数说明]

queried_posting_list 为该轮查询中对应的倒排链表结构体数组。结构体中保存了倒排链表的长度和链表内容。query_word_num 为该轮查询的关键词数量。将倒排链表求交结果保存在 result_list 里。

```
void SvS_refine(POSTING_LIST *queried_posting_list, int query_word_num,
↪ vector<unsigned int> &result_list) {
    int *sorted_index = new int[query_word_num];
    //Sort the inverted list by the length
    get_sorted_index(queried_posting_list, query_word_num, sorted_index);
    //Record the position of the element every time it is found
    vector<int> finding_pointer(query_word_num, 0);
    //Put the shortest inverted list at the front
    for (int i = 0; i < queried_posting_list[sorted_index[0]].len; i++) {
        result_list.push_back(queried_posting_list[sorted_index[0]].arr[i]);
    }
    for (int i = 1; i < query_word_num; i++) {
        vector<unsigned int> temp_result_list;
        for (unsigned int &j: result_list) {
            int location =
            ↪ binary_search_with_position(&queried_posting_list[sorted_index[i]],
            ↪ j, finding_pointer[i]);
            if (queried_posting_list[sorted_index[i]].arr[location] == j) {
                temp_result_list.push_back(j);
            }
            //Update the finding pointer
            finding_pointer[sorted_index[i]] = location;
        }
        //Update the result list
        result_list = temp_result_list;
    }
    delete[] sorted_index;
}
```

7.2 zip-zap 算法代码

```
void SvS_zip_zap(POSTING_LIST *queried_posting_list, int query_word_num,
↪ vector<unsigned int> &result_list) {
    int *sorted_index = new int[query_word_num];
```

```
    get_sorted_index(queried_posting_list, query_word_num, sorted_index);
    for (int k = 0; k < queried_posting_list[sorted_index[0]].len; k++) {
        result_list.push_back(queried_posting_list[sorted_index[0]].arr[k]);
    }
    for (int i = 1; i < query_word_num; i++) {
        vector<unsigned int> temp_result_list;
        unsigned int p0 = 0;
        unsigned int pi = 0;
        while (p0 < result_list.size() && pi < queried_posting_list[sorted_index[i]].len) {
            if (result_list[p0] == queried_posting_list[sorted_index[i]].arr[pi]) {
                temp_result_list.push_back(result_list[p0]);
                p0++;
                pi++;
            } else if (result_list[p0] < queried_posting_list[sorted_index[i]].arr[pi]) {
                p0++;
            } else {
                pi++;
            }
        }
        result_list = temp_result_list;
    }
}
```

7.3 zip-zap SIMD 算法代码

```
        while (p0 < rounded_len0 && pi < rounded_leni) {
load the result array in unsigned int vector register
        __m128i v_0 = _mm_loadu_si128(reinterpret_cast<__m128i *>(result_list.data() + p0));
        __m128i v_i = _mm_loadu_si128(reinterpret_cast<__m128i *>(temp_array + pi));
        //get the last element of v0 and vi in an unsigned int
        int l0_max=_mm_extract_epi32(v_0,3);
        int li_max=_mm_extract_epi32(v_i,3);
        //move pointers
        p0+=(l0_max<=li_max)?4:0;
        pi+=(li_max<=l0_max)?4:0;
        //compute mask of common elements in v0 and vi with cycle shift
```

```
        __m128i v_mask1 = _mm_cmpeq_epi32(v_0, v_i);
        v_i = _mm_shuffle_epi32(v_i, _MM_SHUFFLE(0,3,2,1));
        __m128i v_mask2 = _mm_cmpeq_epi32(v_0, v_i);
        v_i = _mm_shuffle_epi32(v_i, _MM_SHUFFLE(0,3,2,1));
        __m128i v_mask3 = _mm_cmpeq_epi32(v_0, v_i);
        v_i = _mm_shuffle_epi32(v_i, _MM_SHUFFLE(0,3,2,1));
        __m128i v_mask4 = _mm_cmpeq_epi32(v_0, v_i);
        __m128i
        ↪  cmp_mask=_mm_or_si128(_mm_or_si128(v_mask1,v_mask2),_mm_or_si128(v_mask3,v_mask4))
        int mask = _mm_movemask_ps((__m128)cmp_mask);
        //get the position of the common elements in v0 and vi
        if(mask!=0) {
            if(mask&1<<0)
                temp_result_list.push_back(_mm_extract_epi32(v_0,0));
            if(mask&1<<1)
                temp_result_list.push_back(_mm_extract_epi32(v_0,1));
            if(mask&1<<2)
                temp_result_list.push_back(_mm_extract_epi32(v_0,2));
            if(mask&1<<3)
                temp_result_list.push_back(_mm_extract_epi32(v_0,3));
        }
    }
```

7.4 简化 Adp 算法代码

```
void simplified_Adp(POSTING_LIST *queried_posting_list, int query_word_num,
↪  vector<unsigned int> &result_list) {
//start with sorting the posting list to find the shortest one
int *sorted_index = new int[query_word_num];
get_sorted_index(queried_posting_list, query_word_num, sorted_index);
bool flag;
unsigned int key_element;
vector<int> finding_pointer(query_word_num, 0);
for (int k = 0; k < queried_posting_list[sorted_index[0]].len; k++) {
    flag = true;
    key_element = queried_posting_list[sorted_index[0]].arr[k];
    for (int m = 1; m < query_word_num; m++) {
        int mth_short = sorted_index[m];
        POSTING_LIST searching_list = queried_posting_list[mth_short];
        //if the key element is larger than the end element of a list ,it means
        ↪  any element larger than the key element can not be the intersection
```

```
15          if (key_element > searching_list.arr[searching_list.len - 1]) {
16              goto end;
17          }
18          int location =
            ↪  binary_search_with_position(&queried_posting_list[mth_short],
            ↪  key_element,finding_pointer[mth_short]);
19          if (searching_list.arr[location] != key_element) {
20              flag = false;
21              break;
22          }
23          finding_pointer[mth_short] = location;
24      }
25      if (flag) {
26          result_list.push_back(key_element);
27      }
28  }
29  end: delete[] sorted_index;
30
```

7.5 Sequential 算法代码

```
1   void sequential(POSTING_LIST *queried_posting_list, int query_word_num,
    ↪  vector<unsigned int> &result_list) {
2
3   //get the key element from the list which just failed to find in the binary search
    ↪  each time
4   //start with sorting the posting list to find the shortest one
5   int *sorted_index = new int[query_word_num];
6   get_sorted_index(queried_posting_list, query_word_num, sorted_index);
7   bool flag;
8   unsigned int key_element;
9   vector<int> finding_pointer(query_word_num, 0);
10  key_element =
    ↪  queried_posting_list[sorted_index[0]].arr[finding_pointer[sorted_index[0]]];
11  //we definitely need not search the key element in the list which key element is
    ↪  chosen
12  int gaping_mth_short = 0;
13  while (true) {
14      flag = true;
15      for (int m = 0; m < query_word_num; ++m) {
16          if (m == gaping_mth_short)
```

```
17              continue;
18          else {
19              int mth_short = sorted_index[m];
20              POSTING_LIST searching_list = queried_posting_list[mth_short];
21              int location =
                ↪  binary_search_with_position(&queried_posting_list[mth_short],
                ↪  key_element,finding_pointer[sorted_index[m]]);
22              if (searching_list.arr[location] != key_element) {
23                  if (searching_list.len == location) {
24                      //all the elements in the list are smaller than the key
                        ↪  element, algorithm end
25                      goto end_Seq;
26                  }
27                  flag = false;
28                  //update the key element and the pointer
29                  //the location indicates the first element larger than the key
                    ↪  element
30                  key_element = searching_list.arr[location];
31                  finding_pointer[mth_short] = location;
32                  gaping_mth_short = m;
33                  break;
34              }
35              finding_pointer[mth_short] = location;
36          }
37      }
38      if (flag) {
39          //if the key element is found in all the lists, we choose the key element
            ↪  from the list used in last turn
40          result_list.push_back(key_element);
41          finding_pointer[sorted_index[gaping_mth_short]]++;
42          key_element = queried_posting_list[sorted_index[gaping_mth_short]].arr
43          [finding_pointer[sorted_index[gaping_mth_short]]];
44      }
45
46      if (finding_pointer[sorted_index[gaping_mth_short]] ==
47          queried_posting_list[sorted_index[gaping_mth_short]].len) {
48          //all the elements in the list are smaller than the key element, algorithm
            ↪  end
49          goto end_Seq;
50      }
51  }
52  end_Seq:
```

```
53  delete[] sorted_index;
54
```

7.6 Max_Successor 算法代码

```
1       if (queried_posting_list[sorted_index[0]].arr[finding_pointer[sorted_index[0]]
        ↪  + 1] >
2       searching_list.arr[location])
3   {
4       key_element =
        ↪  queried_posting_list[sorted_index[0]].arr[++finding_pointer[sorted_index[0]]];
5       gaping_mth_short = 0;
6   } else {
7       key_element = searching_list.arr[location];
8       //this is optional, and it just allows us to find one less element in the
        ↪  first list.
9       finding_pointer[sorted_index[0]]++;
10      gaping_mth_short = m;
11  }
12  finding_pointer[mth_short] = location;
13      break;
```

7.7 顺序查找的 SIMD 优化

7.7.1 x86 平台

```
1       int serial_search_with_location_using_SIMD(POSTING_LIST *list, unsigned int
        ↪  element, int index) {
2   const __m128i keys = _mm_set1_epi32(element);
3
4   const auto len_list = list->len;
5   const auto remainder = len_list % 4;
6
7   for (size_t i=index; i < len_list-remainder; i += 8) {
8
9       const __m128i vec1 = _mm_loadu_si128(reinterpret_cast<const
        ↪  __m128i*>(&list->arr[i]));
10      const __m128i vec2 = _mm_loadu_si128(reinterpret_cast<const
        ↪  __m128i*>(&list->arr[i + 4]));
11      const __m128i cmp1 = _mm_cmpgt_epi32(vec1, keys);
12      const __m128i cmp2 = _mm_cmpgt_epi32(vec2, keys);
```

```
        const __m128i tmp  = _mm_packs_epi32(cmp1, cmp2);
        const uint32_t mask = _mm_movemask_epi8(tmp);

        if (mask != 0) {
            int pos = i + __builtin_ctz(mask)/2;
            if(pos==index)
                return index;
            else if(list->arr[pos-1]==element)
                return pos-1;
            else
                return pos;
        }
    }

    for (size_t i=len_list-remainder; i < len_list; i++) {
        if (list->arr[i] >= element)
            return i;
    }
    return list->len;

```

7.7.2 ARM 平台

```
        int serial_search_with_location_using_SIMD(POSTING_LIST *list, unsigned int
        ↪  element, int index) {
    //using NEON SIMD instructions, so we can compare 4 elements at a time
    int len_list=list->len-index;
    int remainder=len_list%4;
    for (int i = index; i < list->len-remainder; i += 4) {
        //duplicate the element to compare with using NEON SIMD instructions
        uint32x4_t element_vec = vdupq_n_u32(element);
        uint32x4_t list_vec = vld1q_u32(list->arr + i);
        //compare the element with the list using NEON SIMD instructions
        uint32x4_t result_vec = vcgeq_u32(list_vec, element_vec);
        unsigned int* result_ptr = (unsigned int*) &result_vec;
        //if the element is not found, return the index of the first element that is
        ↪  not less than the element
        if(result_ptr[0]|result_ptr[1]|result_ptr[2]|result_ptr[3])
            for(int j = 0; j < 4; j++) {
                if(result_ptr[j] != 0)
                    return i + j;
```

```
        }
    }
    //look for the rest of the elements
    for (int i = list->len-remainder; i <list->len; i++) {
        if (list->arr[i] >= element)
            return i;
    }
    return list->len;

```

7.8 哈希分段函数

```
void get_hash_table(POSTING_LIST *posting_list_hash_container, POSTING_LIST *posting_list_container) {
//The hash role is x>>p
//So the docID of a document within the same segment differs only in the lowest p bits of the docID.
//the hash table stores the segments of the posting list and the position of the first element of the segment.
for (int i = 0; i < POSTING_LIST_NUM; i++) {
    //get the length of hash table according to the last element of the posting list
    posting_list_hash_container[i].len =
            (posting_list_container[i].arr[posting_list_container[i].len - 1] >> P);
    posting_list_hash_container[i].arr = new unsigned int[posting_list_hash_container[i].len];
    for (int j = 0; j < posting_list_hash_container[i].len; j++) {
        posting_list_hash_container[i].arr[j] = -1;
    }
    //-1 means the hash table is empty
    unsigned int hash_key;
    for (int j = 0; j < posting_list_container[i].len; j++) {
        hash_key = posting_list_container[i].arr[j] >> P;
        if (posting_list_hash_container[i].arr[hash_key] == -1) {
            posting_list_hash_container[i].arr[hash_key] = j;
        }
    }
}

```

7.9 基于值的按表求交

7.9.1 两表求交

```
void SvS_range_based(POSTING_LIST *queried_posting_list, int query_word_num, vector<unsigned int> &result_list) {
int *sorted_index = new int[query_word_num];
get_sorted_index(queried_posting_list, query_word_num, sorted_index);
//Put the shortest inverted list at the front
for (int i = 0; i < queried_posting_list[sorted_index[0]].len; i++) {
    result_list.push_back(queried_posting_list[sorted_index[0]].arr[i]);
}
for (int i = 1; i < query_word_num; i++) {
    vector<unsigned int> temp_result_list;
    int start_0 = 0;
    int end_0 = result_list.size() - 1;
    int start_i = 0;
    int end_i = queried_posting_list[sorted_index[i]].len - 1;
    while (start_0 <= end_0 && start_i <= end_i) {
        //move start pointer
        //"bigger head" list search the start element in the "smaller head" list
        if (result_list[start_0] >= queried_posting_list[sorted_index[i]].arr[start_i])
            start_i = binary_search_higher_position(queried_posting_list[sorted_index[i]].arr, result_list[start_0],
                                                    start_i, end_i);
        else
            start_0 = binary_search_higher_position(&result_list[0],
                                                    queried_posting_list[sorted_index[i]].
                                                    start_0,
                                                    end_0);
          if(start_i>end_i||start_0>end_0)
              break;
Without above annotated code,the start_i or start_0 is possible to be out of range.
But as we only read the values out of range rather than write them,it is safe.
And we can get 40ms speedup.
        if (result_list[start_0] == queried_posting_list[sorted_index[i]].arr[start_i]) {
            temp_result_list.push_back(result_list[start_0]);
            start_0++;
```

```
32              start_i++;
33          } else {
34              result_list[start_0] >
                ↪  queried_posting_list[sorted_index[i]].arr[start_i] ? start_i++ :
                ↪  start_0++;
35          }
36
37          //lower tail search the end element in the "bigger tail" list
38          if (result_list[end_0] <=
            ↪  queried_posting_list[sorted_index[i]].arr[end_i])
39              end_i =
                ↪  binary_search_lower_position(queried_posting_list[sorted_index[i]].arr,
                ↪  result_list[end_0],
40                                                  start_i, end_i);
41          else
42              end_0 = binary_search_lower_position(&result_list[0],
                ↪  queried_posting_list[sorted_index[i]].arr[end_i],
43                                                  start_0, end_0);
44            if(start_i>end_i||start_0>end_0)
45                break;
46  Without above annotated code,the start_i or start_0 is possible to be out of range.
47  But as we only read the values out of range rather than write them,it is safe.
48          if (result_list[end_0] ==
            ↪  queried_posting_list[sorted_index[i]].arr[end_i]) {
49              temp_result_list.push_back(result_list[end_0]);
50              end_0--;
51              end_i--;
52          } else {
53              result_list[end_0] < queried_posting_list[sorted_index[i]].arr[end_i]
                ↪  ? end_i-- : end_0--;
54          }
55      }
56      result_list = temp_result_list;
57      sort(result_list.begin(), result_list.end());
58  }
59
```

7.9.2 多表求交

```
void adp_range_based(POSTING_LIST *queried_posting_list, int query_word_num, vector<unsigned int> &result_list) {

//get the key element from the list which just failed to find in the binary search each time
//start with sorting the posting list to find the shortest one
int *sorted_index = new int[query_word_num];
get_sorted_index(queried_posting_list, query_word_num, sorted_index);
vector<int>start_index_container(query_word_num,0);
vector<int>end_index_container(query_word_num,0);
for(int i=0;i<query_word_num;i++){
    end_index_container[i]=queried_posting_list[sorted_index[i]].len-1;
}
while(not_empty(start_index_container,end_index_container))
{
    //get "biggest head" list
    unsigned int biggest_head = 0;
    unsigned int biggest_head_index = 0;
    for(int i=0;i<query_word_num;i++){

        if(queried_posting_list[sorted_index[i]].arr[start_index_container[i]]>biggest_hea

            biggest_head=queried_posting_list[sorted_index[i]].arr[start_index_container[i
            biggest_head_index=i;
        }
    }
    //get "smallest tail" list
    unsigned int smallest_tail = UINT32_MAX;
    unsigned int smallest_tail_index = 0;
    for(int i=0;i<query_word_num;i++){

        if(queried_posting_list[sorted_index[i]].arr[end_index_container[i]]<smallest_tail

            smallest_tail=queried_posting_list[sorted_index[i]].arr[end_index_container[i]
            smallest_tail_index=i;
        }
    }
    //search the biggest head in other lists
    bool flag=false;
    for(int i=0;i<query_word_num;i++){
```

```
35          if(i==biggest_head_index) continue;
36
            ↪ start_index_container[i]=binary_search_higher_position(queried_posting_list[sorted
37
            ↪ if(queried_posting_list[sorted_index[i]].arr[start_index_container[i]]!=biggest_he
38              flag=true;
39          }
40          else
41          {
42              start_index_container[i]++;
43          }
44      }
45      start_index_container[biggest_head_index]++;
46      if(!flag){
47          result_list.push_back(biggest_head);
48      }
49      //search the smallest tail in other lists
50      flag=false;
51      for(int i=0;i<query_word_num;i++){
52          if(i==smallest_tail_index) continue;
53
            ↪ end_index_container[i]=binary_search_lower_position(queried_posting_list[sorted_in
54
            ↪ if(queried_posting_list[sorted_index[i]].arr[end_index_container[i]]!=smallest_tai
55              flag=true;
56          }
57          else
58          {
59              end_index_container[i]--;
60          }
61      }
62      end_index_container[smallest_tail_index]--;
63      if(!flag){
64          result_list.push_back(smallest_tail);
65      }
66
67  }
68
69  delete[] sorted_index;
70
```

8 项目源代码

x86 版 Github 源码

ARM 版 Github 源码

实验代码内容基本可以从文件名上得到指示。

参考文献

[1] 宋省身. 时空高效的倒排索引压缩和求交算法研究. PhD thesis, 国防科技大学, 2018.

[2] [EB/OL]. https://en.cppreference.com/w/cpp/algorithm/set_intersection.

[3] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 743–752, 2000.

[4] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *Proceedings of the VLDB Endowment*, 8(3):293–304, 2014.

[5] [EB/OL]. https://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/.

[6] 张帆. 搜索引擎中索引表求交和提前停止技术优化研究. PhD thesis, 南开大学, 2012.

[7] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger. An experimental investigation of set intersection algorithms for text searching. *Journal of Experimental Algorithmics (JEA)*, 14:3–7, 2010.

[8] Faiza Manseur, Lougmiri Zekri, and Mohamed Senouci. A new fast intersection algorithm for sorted lists on gpu. *Journal of Information Technology Research (JITR)*, 15(1):1–20, 2022.