南開大學
Nankai University

计算机学院

并行程序设计期末开题报告

倒排索引求交

姓名：唐明昊 朱世豪

学号：2113927 2113713

专业：计算机科学与技术

2023 年 3 月 26 日

目录

1 问题描述

在传统的信息检索系统中，文档通常是按照文档编号或者时间等顺序排列，用户查询时需要逐一扫描文档库。这种方法随着数据量的增大，查询时间会逐渐变得无法接受。倒排索引就是为了解决这一问题而提出的。倒排索引将每个单词映射到包含这个单词的所有文档的集合中，从而提高查找的效率。通过倒排索引，我们可以很快地找到包含任何查询单词的文档，这使得搜索引擎可以快速地响应用户的查询。然而网页文档等互联网资源的规模急剧膨胀，为了快速、准确地应答每秒数以千万计的用户查询请求，需要高效的基于倒排索引的请求处理算法。

1.1 倒排索引

倒排索引（inverted index），又名反向索引、置入文档等，多使用在全文搜索下，是一种通过映射来表示某个单词在一个文档或者一组文档中的存储位置的索引方法。在各种文档检索系统中，它是最常用的数据结构之一。

对于一个有 U 个网页或文档（Document）的数据集，若想将其整理成一个可索引的数据集，则可以为数据集中的每篇文档选取一个文档编号（DocID），使其范围在 [1,U] 中。其中的每一篇文档，都可以看做是一组词（Term）的序列。则对于文档中出现的任意一个词，都会有一个对应的文档序列集合，该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，...，t_k，表求交算法返回 $\cap_{1\leq i\leq k} l(t_i)$。

首先，求交会按照倒排列表的长度对列表进行升序排序，使得：

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

例如查询“2014 NBA Final”，搜索引擎首先在索引中找到“2014”，“NBA”，“Final”对应的倒排列表，并按照列表长度进行排序：

$$l(2014) = (13,\ 16,\ 17,\ 40,\ 50)$$

$$l(NBA) = (4,\ 8,\ 11, 13, 14,\ 16, 17, 39, 40, 42, 50)$$

$$l(Final) = (1,\ 2,\ 3,\ 5,\ 9,\ 10,\ 13,\ 16,\ 18,\ 20,\ 40,\ 50)$$

求交操作返回三个倒排列表的公共元素，即：

$$l(2014) \cap l(NBA) \cap l(Final) = (13,\ 16,\ 40,\ 50)$$

1.2 文献综述

在实际应用中，需要对多个文档集合进行交集或并集操作，以获取相关文档的信息。这就需要使用倒排索引求交或求并的算法。随着数据规模的增大和计算机硬件的发展，倒排索引求交算法也在不断发展和优化，以提高计算效率和搜索精度。以下分别从串行算法和并行算法对已有的倒排索引求交算法进行分析与简要介绍。

1.2.1 串行算法

1. Hwang 和 Lin 的算法 [6, 7]：利用两个指针分别遍历两个有序数组，比较指针所指的元素，如果相等则输出，否则移动较小元素的指针。该算法可以推广到 k 个有序数组的情况。
   - 优点：简单易懂。
   - 缺点：在处理大规模的数据集时速度较慢。
2. Demaine 的自适应算法：利用一个**优先队列**维护 k 个有序数组的当前元素，每次取出最小元素并将其所在数组的下一个元素入队。如果最小元素在所有数组中都出现，则输出。该算法可以根据数组长度和重复度自适应地调整遍历顺序 [2, 3]。
   - 优点：可以适应不同的倒排索引规模、重复度、排序方式等因素，自动调整算法参数。
   - 缺点：在处理大规模的数据集时速度较慢。
3. Tsirogiannis 的动态探测算法：利用 CPU 的多级 Cache，动态地选择最佳的遍历顺序，以减少 Cache 未命中和内存访问延迟。该算法还可以实现多处理器上的负载均衡 [11]。
   - 优点：可以适应不同的倒排索引规模、重复度、排序方式等因素，自动调整算法参数，并利用 CPU 的多级 Cache，动态地选择最佳的遍历顺序，以减少 Cache 未命中和内存访问延迟。
   - 缺点：比较难以编写和调试，需要考虑多种情况和细节。受到 CPU 架构和 Cache 大小等硬件限制的影响。

1.2.2 并行算法

1. Inoue 等人的 SSE 优化算法：利用 SSE 指令集，一次比较多个元素，并使用位操作来记录匹配结果，减少分支预测失败数。该算法可以提高数据吞吐量和计算效率 [8]。
   - 优点：充分利用 Cache，减少内存访问延迟和 Cache 未命中。能适应不同的记录大小和关键字类型，提高排序性能。
   - 缺点：需要使用特定的硬件和编译器，不具有良好的移植性和兼容性。需要对代码进行复杂的优化和调试，可能增加开发难度和出错概率。不适合处理非常大的数据集，因为 SSE 寄存器的数量和大小有限。
2. Ding 等人的分段归并算法：利用 GPU 平台，将每个有序数组分为多个段，并在 GPU 上执行段与段之间的归并计算。该算法可以充分利用 GPU 的并行能力 [4]。
   - 优点：可以处理大规模的数据集。
   - 缺点：对于小规模倒排索引效果不理想。
3. Wu 等人的二分搜索算法：利用 GPU 平台，以二分搜索作为基准搜索算法，在 GPU 上执行每个有序数组与目标值之间的二分搜索 [12, 13]。
   - 优点：该算法可以实现简单而有效地（5 倍左右的加速比）并行计算。
   - 缺点：需要额外的空间来存储目标值。
4. Zhang 等人的 Bloom Filter 算法：利用 GPU 平台，使用 Bloom Filter 来过滤掉不可能匹配的元素，并在 GPU 上执行剩余元素之间的求交计算 [14]。

- 优点：该算法可以实现高效而精确地进行并行计算，达到了 21 倍左右的加速比。
- 缺点：需要额外的空间来进行存储（Bloom Filter），不能保证返回所有公共 docID。

2 研究方案

2.1 基本实现

2.1.1 按表求交

先使用两个表进行求交，得到中间结果再和第三个表求交，依次类推直到求交结束。

Algorithm 1 按表求交算法 (SvS)

Input: $l(t_1), l(t_2), \ldots, l(t_k)$, sorted by $|l(t_1)| \leq |l(t_2)| \leq \ldots \leq |l(t_k)|$
Output: Intersection results $\cap_{1 \leq i \leq k} l(t_i)$

```
1:  function SetVersusSet
2:      S = l(t_1)
3:      for i = 2 to k do
4:          for each element e ∈ S do
5:              found = find(e, (l_i))
6:              if found = FALSE then
7:                  Delete e from S
8:              end if
9:          end for
10:     end for
11:     return S
12: end function
```

这是常用的一种算法，它首先是找出最短的两个集合，依次查找第一个集合里的元素是否出现在第二个集合内部。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。

Demaine 考虑的 $Swapping_{SvS}$ 算法和上述算法有稍微的不同，即是在每次比较后，取包含更少元素的集合来与再下一个集合进行比较，这种算法在第一个集合和第二个集合比较之后第二个集合反而更少的情况下效果更好，但实验表明这种情况并不多见。

2.1.2 按元素求交

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。

Algorithm 2 按元素求交算法 (Adaptive)

Input: $l(t_1), l(t_2), \ldots, l(t_k)$, sorted by $|l(t_1)| \leq |l(t_2)| \leq \ldots \leq |l(t_k)|$
Output: Intersection results $\cap_{1 \leq i \leq k} l(t_i)$

```
1: function Adp
2:     S ← ∅
3:     while No empty list do
4:         Reorder the lists by increasing number of undetected elements.
5:         e ← FirstUnfoundElement(l(t_1))
```

```
        s ← 2
        repeat
            found ← find(e, l(ts))
            s ← s + 1
        until s = k or found = FALSE
        if s = k and found = TRUE then
            S.add(e)
        end if
    end while
    return S
end function
```

DAAT 的基本思路就是在各个列表中寻找当前文档，当在所有列表中寻找完某个文档之后，每条链表的剩余的未扫描文档数量也不相同，但只要其中一条链表走到尽头，则本次求交结束。**因此，每次都从剩余文档数量最短的链表进行扫描能够加速这个过程**。而 Adaptive 算法的基本思路就是每次都从剩余文档数量最少的链表开始扫描，这样能够尽量缩短链表的扫描过程。

2.2 其他算法

除了两个基本算法外，我们还搜集了一些其他算法，并计划在其上进行并行化研究。

2.2.1 拉链法

两个指针指向首元素，比较元素的大小：如果相同，放入结果集，随意移动一个指针；否则，移动值较小的一个指针，直到队尾。

Algorithm 3 拉链法 (Zipper)

Input: list A, list B

Output: Intersection Results C

```
function INTERSECTION(A, B)
    Sort(A)
    Sort(B)
    pointerA ← 0
    pointerB ← 0
    C ← {}
    while pointerA < len(A) and pointerB < len(B) do
        if A[pointerA] == B[pointerB] then
            C.add(A[pointerA])
            pointerA ← pointerA + 1
        else if A[pointerA] < B[pointerB] then
            pointerA ← pointerA + 1
        else
            pointerB ← pointerB + 1
        end if
    end while
```

17: **return** C
18: **end function**

这是一种简单易懂的算法。这种方法的好处是：集合中的元素最多被比较一次，时间复杂度为 O(n)，多个有序集合可以同时进行。

2.2.2 跳表法

跳表法是在拉链法基础上优化，通过在倒排列表中加入跳指针，以减少在交集操作中不必要的比较次数，从而提高了查询效率。用空间换时间，也可以建立多级索引，跳过不必要的比较，提高执行效率 [1]。

Algorithm 4 跳指针法 (Skip Pointers)

Input: list l_1, list l_2 and skip pointers
Output: Intersection Results

1: **function** IntersectWithSkips(l_1, l_2)
2: $Result \leftarrow <>$
3: **while** $l_1 \neq NULL$ and $l_2 \neq NULL$ **do**
4: **if** $docID(l_1) = docID(l_2)$ **then** ADD($result, docID(l_1)$)
5: $l_1 \leftarrow next(l_1)$
6: $l_2 \leftarrow next(l_2)$
7: **else if** $docID(l_1) < docID(l_2)$ **then**
8: **if** $hasSkip(l_1$ and $(docID(skip(l_1)) \leq docID(l_2))$ **then**
9: **while** $hasSkip(l_1)$ and $(docID(skip(l_1)) \leq docID(l_2)$ **do**
10: $l_1 \leftarrow skip(l_1)$
11: **end while**
12: **else**
13: $l_1 \leftarrow next(l_1)$
14: **end if**
15: **else if** $hasSkip(l_2$ and $(docID(skip(l_2)) \leq docID(l_1))$ **then**
16: **while** $hasSkip(l_2$ and $(docID(skip(l_2)) \leq docID(l_1))$ **do**
17: $l_2 \leftarrow skip(l2)$
18: **end while**
19: **else**
20: $l_2 \leftarrow next(l_2)$
21: **end if**
22: **end while**
23: **end function**

有序链表集合求交集，跳表是最常用的数据结构，它可以将有序集合求交集的复杂度由 O(n) 降至接近 O(log(n))。

2.2.3 bitmap 法

使用 bitmap 来表示集合，将集合中的每个元素映射为一个唯一的二进制位，然后使用位运算来求解两个集合的交集。

每条链表用一个位向量表示，每个 bit 对应一个 DocID，某位为 1 表示该链表包含此 Doc、为 0 表示不包含。对于另一集合中的每个元素，我们可以查询对应的位是否为 1，从而求交运算就变为两个位向量的位与运算。

Algorithm 5 bitmap 法

Input: list A, list B

Output: Intersection Results C

function BitMap(A, B)
  $bitmap_A \leftarrow 0$
  $bitmap_B \leftarrow 0$
  $C \leftarrow \{\}$
  for $a \in A$ **do**
    $bitmap_A \leftarrow bitmap_A$ OR $(1 << a)$ ▷ 将元素映射为二进制位
  end for
  for $b \in B$ **do**
    $bitmap_B \leftarrow bitmap_B$ OR $(1 << b)$ ▷ 将元素映射为二进制位
  end for
  $intersect\_bitmap \leftarrow bitmap_A$ AND $bitmap_B$ ▷ 使用位运算求解交集
  for $i \leftarrow 0$ to size($intersect\_bitmap$) **do**
    if $intersect\_bitmap$ has bit i **then**
      C.add(i) ▷ 将二进制位转换回元素
    end if
  end for
  return C
end function

比起传统的串行算法，更适合将其并行化。但是倒排链表通常是比较稀疏的——表示为位向量的话绝大多数 bit 均为 0、极少数为 1，这就造成存储空间和串行计算时间可能都远不如倒排链表方式，即使并行效率很高可能最终性能也不如倒排链表。

所以，在位图存储的基础上，还可以采取建立二级索引，将位向量分块等优化策略。

2.2.4 分治法

利用分治思想，取出较短集合中的中间元素，在较长集合中搜索该元素，于是将较短和较长集合均分为两部分，再对这两部分递归搜索下去即可 [5]。

Algorithm 6 分治法

Input: list A, list B

Output: Intersection Results C

function MutualPartition(A, B)
  if $m > n$ **then** Swap A and B; Swap n and m;
  end if
  Let m be the length of B and n be the length of A;
  Let p be the median element of B ($p = b_{m/2}$);
  Binary search for the position of p in A, say $a_j < p < a_{j+1}$;

7: **if** $p = a_i$ **then** Print p;
8: **end if**
9: Compute recursively the intersection $A[1, j] \cap B[1, m/2]$;
10: Compute recursively the intersection $A[j + 1, n] \cap B[m/2 + 1, m]$;
11: **end function**

在实际应用中，如果集合中元素的分布比较均匀，分治法可以获得很好的性能，但如果集合中元素的分布不均匀，分治法的性能可能会受到影响。

2.2.5 hash 分段法

基于 SvS 算法进行改进，将集合中的元素存储在哈希表中，并对于另一个集合中的元素，可以直接从哈希表开始查找。

建立辅助索引 hash 桶。将每个倒排列表的 DocID 依次映射到对应的 hash 段里，同一段内元素按升序排列，在查找比较时，可以直接定位 hash 段，再从 hash 段找回比较的列表，很少的步骤即可完成一轮查找。

3 学习计划

3.1 SIMD

我们准备在以上几个基础串行算法 (SvS, Adaptive，Zipper，Skip Pointers 等算法) 的基础上，使用 SIMD 指令进行并行化以加快算法的速度。

因为升序列表求交操作本身不是标准的单指令流多数据流模式，直接进行向量化有很多困难。**所以，除了对倒排链表的存储格式及其上的串行求交算法直接进行 SIMD 并行化之外，还可考虑其他更利于向量化的存储方式和求交算法**。

考虑使用位图存储方式——每条链表用一个位向量表示，将求交运算就变为两个位向量的位与运算。在此之上还可考虑一些优化策略，解决位图存储比较稀疏的问题，例如建立二级索引——将位向量分块，对 DocID 进行重排，令位向量更聚集等策略。

还了解到一些其他存储方法，都可以在后续实验中进行学习与实现。例如使用 32 位整数，将所有文档编号存储在一个 SIMD 向量中，所有出现次数存储在另一个 SIMD 向量中，这样就可以同时处理多个文档的信息。另外 SSE 指令集允许使用一条指令（_mm_cmpeq 内在指令）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，得到相等元素的位置。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。

除了以上角度以外，我们还准备了以下文献留作进一步的学习使用：

- lemire 教授的 SIMDCompressionAndIntersection 库 [10]: 其中有 SIMD, galloping, scalar, highly-scalable_intersect_SIMD 等多种算法和 benchmark 程序，对不同算法效果进行对比。
- HIGHLY SCALABLE 博客 [9]: 详细介绍了 Scalar Intersection 和对他的一系列 SIMD 优化如向量化求交分治向量化求交等算法，并对他们的性能进行了评估。

以上均是在未学习 SIMD 情况下的浅薄调查，在学习 SIMD 后进行实验时再进行深入研究，从而对于倒排索引求交的 SIMD 优化算法进一步进行改进与优化。

3.2 PThread

在倒排索引求交问题中，考虑 **Query 间并行**和 **Query 内并行**两种方法，用 pthread 进行并行化上述算法。但需要注意的是，在使用 pthread 进行并行化时，需要考虑线程的创建和销毁所带来的开销，以及不同线程之间的同步问题；线程的数量也需要控制在合理的范围，以避免过多的线程导致的线程调度开销和资源竞争问题；这些都是学习过程中需要重点关注的问题。另外，我们还去查询了一些位图存储上并行化的方法，在后续的学习中，我们将着重于对于**位图存储多线程化**的学习，并进一步查找相关文献以支持我们的研究。

- Query 间并行：不同查询之间的求交可以并行化处理，**每个线程负责处理一个查询**。这种情况下，可以通过 pthread 库中的 pthread_create() 函数创建多个线程，将不同查询的处理分配到不同的线程上，实现高吞吐率。需要注意，由于不同查询的倒排列表大小不一，因此可以通过动态分配内存的方式来避免不必要的内存浪费。

- Query 内并行：对于一个查询，**将倒排列表中的每个倒排项视为一个子问题**，通过并行化处理来提高求交的效率。使用 Adp 算法，对于每个倒排项，将其与其他倒排项进行求交，并将结果存储在一个结果集合中。

  还可以考虑将要计算的集合分为多个子集，**每个子集分配一个线程来处理**。在每个线程中，使用 SvS 算法或其他并行化算法来计算每个子集的交集。当所有线程都完成计算后，将每个子集的交集合并成最终的交集。在这种实现中，需要注意以下几点：

  这种情况下，创建多个线程，将不同的倒排项或子集的处理分配到不同的线程上。为了避免多个线程同时访问共享的结果集合，需要使用到 pthread_mutex_lock() 和 pthread_mutex_unlock() 函数来保护共享资源的访问。

- 对位图存储方式进行多线程优化，可以采用并行位图算法，其中每个线程独立地对一个位图进行操作，最后将结果合并。在多个线程之间共享的数据结构中，可以使用锁或无锁数据结构来避免竞争和冲突。具体来说，可以采用以下几种方法：

  - 分段处理：将位图分成若干个连续的段，每个线程独立地处理一个或多个段，避免了竞争和冲突。这种方法的缺点是线程间的负载可能不均衡，导致某些线程的处理速度较慢。
  - 原子位操作：使用原子位操作来避免竞争和冲突，例如 ___sync_fetch_and_or() 函数可以原子地将一个 bit 位置为 1。这种方法的缺点是原子操作会带来额外的开销。
  - 位图压缩：将位图压缩成一个紧凑的向量，然后对向量进行操作。这种方法可以避免竞争和冲突，并且能够利用 SIMD 指令进行并行化，但需要对位图进行压缩和解压缩，带来额外的开销。

3.3 MPI

MPI（Message Passing Interface）是一种基于消息传递的并行计算编程模型，可用于分布式内存系统中的并行计算。对于倒排索引的优化，**MPI 思路上类似多线程并行**，主要可以利用其消息传递机制和分布式计算能力来实现多进程并行化，大致可以从如下方面入手：

- 划分倒排索引：首先需要将倒排索引划分为多个部分，并将其分配到不同的进程上，每个进程只处理其所分配到的部分。这可以参照 PThread 中用到的分割方法。

- 并行计算部分交集：对于每个进程所分配的部分，可以采用线程级并行化方法来计算部分交集，以提高计算效率。
- 汇总结果：每个进程都会计算出其所分配的部分的交集结果，需要将所有结果汇总起来，以得到整个查询的最终结果。
- 通信和同步：MPI 使用消息传递机制进行进程间通信，在并行计算过程中，需要进行**进程间数据交换和同步**，以保证计算正确性。这可以通过 MPI 提供的通信函数来实现。

通过 MPI 并行化倒排索引，可以充分利用分布式计算能力，加速计算过程。但是，MPI 编程较为复杂，需要处理进程间通信和同步等问题，需要后续过程有针对性的认真学习。

3.4 GPU

将 GPU 作为 CPU 的协处理器，利用其强大的并行计算能力来处理查询，进而提升查询处理吞吐率。**在多个线程块间参考多线程进行线程块任务划分；在线程块内，参考 SIMD 进行并行化**。

对于倒排索引问题，我们由参考文献了解到，CPU-GPU 协同查询处理模型由如下 5 部分组成：CPU 预处理，CPU 到 GPU 数据传输，GPU 批次集合求交，GPU 到 CPU 数据传输和 CPU 后处理。

1. CPU 预处理：CPU 不断接受查询，并将一定量的查询打包成一个批次交由 GPU 处理。需要设置一个阈值来决定了每个批次的最小计算量。较大的阈值可以充分发挥 GPU 的并行计算能力，同时少批次大数据量可以**最小化交互频率**，提高处理通信速率。然而，也需要做出权衡，保证每个批次的处理时间处在合理区间。
2. CPU 到 GPU 数据传输：CPU 将整个压缩的倒排索引集传输到 GPU 全局内存中。
3. GPU 批次集合求交：首先为所有查询的最短倒排列表中的每一个 docID 分配负责搜索它的线程。为了充分发挥 GPU 的强大并行计算能力，在线程和最短倒排列表的 docID 之间建立“一一对应”关系。每个线程分配到了一个 docID 之后，通过搜索、扫描和收集三个步骤来求出交集结果。

   在搜索的过程中，每个线程通过搜索来确定它负责的 docID 是否为交集结果。不同于 CPU，GPU 通常采用**按元素求交方式**进行高效搜索——每一个线程独立地依次在较长的倒排列表 $l_{t_2}, l_{t_3}, \ldots, l_{t_k}$ 中搜索最短倒排列表中的唯一 DocID，且这一步所需的时间最多，所以要**重点考虑对这一过程进行优化**。后续学习过程中将参考文献中的插值搜索，二分搜索以及线性搜索等方法进行优化。

   扫描和收集两步调用相应的内核函数从交集标识数组和扫描结果数组获取交集结果。
4. GPU 到 CPU 数据传输和 CPU 后处理：GPU 求出当前批次的交集后，需要将交集发送给 CPU 进行后续处理。这一过程可以通过流水线技术掩盖。

4 小组分工

- 对于基本算法：由唐明昊研究按表求交算法，朱世豪研究按元素求交算法，并在后续实验中进行针对性的并行化研究。
- 对于补充的其他算法，由唐明昊负责分治算法与 hash 分段算法，由朱世豪负责拉链法，跳表法和 bitmap 算法，各自在后续实验中进行针对性的并行化研究。

- 对于 SIMD 实验，初步规划按照上述算法分工（包括位图存储），对每个算法进行并行化后，分别尝试对两个文献进行调研学习，提出一些不同想法，加深对 SIMD 的理解。
- 对于 PThread 实验，初步计划为由唐明昊进行 query 间并行实现，朱世豪进行 query 内并行，后续实验时再对位图存储的多线程化工作进行具体分工。
- 对于 MPI 实验，除参照 PThread 进行实验分工外，各自分别用分配的算法在进程中进行划分、并行求交、汇总、通信和同步等工作。
- 由于对 GPU 实验知识掌握不足，具体实验分工需要在学习了相应知识后，在具体实验报告中给出。

参考文献

[1] Prabhakar Raghavan Christopher D. Manning and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[2] 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.

[3] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, volume 1, pages 91–104. Springer, 2001.

[4] Shuai Ding, Jinru He, Hao Yan, and Torsten Suel. Using graphics processors for high performance ir query processing. In *Proceedings of the 18th international conference on World wide web*, pages 421–430, 2009.

[5] Paolo Ferragina. *The magic of Algorithms!* Universita di Pisa, 2019.

[6] Frank K. Hwang and Shen Lin. Optimal merging of 2 elements with n elements. *Acta Informatica*, 1(2):145–158, 1971.

[7] Frank K. Hwang and Shen Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1(1):31–39, 1972.

[8] 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.

[9] ILYA KATSOV. Fast intersection of sorted lists using sse instructions, 2012. https://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/.

[10] lemire. Simd compression and intersection, 2022. https://github.com/lemire/SIMDCompressionAndIntersection.

[11] Dimitris Tsirogiannis, Sudipto Guha, and Nick Koudas. Improving the performance of list intersection. *Proceedings of the VLDB Endowment*, 2(1):838–849, 2009.

[12] Di Wu, Fan Zhang, Naiyong Ao, Fang Wang, Xiaoguang Liu, and Gang Wang. A batched gpu algorithm for set intersection. In *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pages 752–756. IEEE, 2009.

[13] Di Wu, Fan Zhang, Naiyong Ao, Gang Wang, Xiaoguang Liu, and Jing Liu. Efficient lists intersection by cpu-gpu cooperative computing. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[14] Fan Zhang, Di Wu, Naiyong Ao, Gang Wang, Xiaoguang Liu, and Jing Liu. Fast lists intersection with bloom filter using graphics processing units. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 825–826, 2011.