学号：20184484

班级：计算机 1803

姓名：胥卜凡

目录

1 有一个 1GB 的文件，一行存放一个词，词的大小不超过 16 个字节，内存大小为 1MB，返回频数最高的 100 个词

1.1 伪代码：

算法 1 统计文件中频数最高的100个词

输入: 单词文件A

输出: 出现频数最高的100个词

```
while 读文件单词! =EOF do
    a=hash(x)存入文件X(a)中
end while
for a=1 to 5000 do
    if X(a)大于1M then;
        重复上述hash分文件过程
    end if;
end for
用常规方法如(Trie等)取出5000个文件频率最大的100词
对文件进行归并
return 0;
```

1.2 分析：

（1）　算法设计：

①分治法，先 hash 映射把大文件分成很多个小文件，具体操作如下：读文件中，对于每个词 x，取 hash(x)%5000，然后按照该值存到 5000 个小文件(记为 f0,f1,...,f4999)中，这样每个文件大概是 200k 左右（每个相同的词一定被映射到了同一文件中）

②对于每个文件 fi，都用 hash_map 做词和出现频率的统计，取出频率大的前 100 个词（topK 问题，建立一个 100 个节点的最小堆），把这 100 个词和出现频率再单独存入一个文件

③根据上述处理，我们又得到了 5000 个文件，归并文件取出 top100 个 K(大小的小根堆，然后遍历 Query，分别和根元素进行对比。

(2)复杂度分析：

可以给出 DGIM 的存储开销：桶的个数 O(logN)，每个桶可以用 O(logN)空间表示，保存表示一个窗口的所有桶所占的空间为 O(log2N)。

对于查询，首先寻找含最近 k 个位的拥有最大时间戳的桶 b，查询结果为在 k 个为中所有桶的大小，加上 b 的大小的一半。一次查询的时间复杂度为 O(logN)。

2 给定 y 一个大规模的平面上的点集 S 和查询点 x，设计高效的外存数据结构,用于查找 S 中和 d 点 x 距离在 R 内的点。分析该数据结构的存储 l 量，查询的 I/O 复杂度，插入和删除点的 I/O 复杂 d 度

2.1 采用的数据结构

O 树。

2.2 存储结构

①0 树的存储空间：0 树的存储空间是线性的。

②对于点集 S=（x，y），用 $\Theta(\sqrt{N/B}/\log_B N)$ 条垂直的线把平面分割成各个块，使每个块包含 S 中位数的个数为 $\frac{1}{2}\sqrt{NB}\log_B N \sim \sqrt{NB}\log_B N$ 。

每个块用 $\Theta(\sqrt{N/B}/\log_B N)$ 条水平线进一步划分各个单元，这样每个单元含点的个数为$\frac{1}{2}Blog_i^2N$~$Blog_i^2N$。0 树中对应垂直线包含一个 B 树 ，对应每个S_i建立一个水平线上的 B 树 ，最后对每个单元中的点构建 KDB 树，块S_i中的第 j 个单元对应的 KD 树称作 T_{ij} 。B 树共将使用 $O((\sqrt{N/B}/\log_B N)^2/B)=O(N/(B\log_B N)^2)$ 的空间，同时每个点被存储在一个线性空间的 KD 树中，因此 0 树的存储空间是线性的。

2.3 0 树的查询处理

0 树上范围查询的 I/O 复杂度是 $O(\sqrt{N/B}+T/B)$。

在 B 树T_v中的查询处理 I/O 开销为 $O(\sqrt{N/B})$ 。那么查询在块 sl 和 sr 中 $2\cdot O(\sqrt{N/B}/\log_B N)$ 棵 KD 树的 I/O 开销是

$$O((\sqrt{N/B}/\log_B N)\cdot O(\sqrt{B\log_B^2 N/B}))+O(T/B)=O(\sqrt{N/B}+T/B)$$

在 B 树 Ti 处理查询，即访问 $O(\sqrt{N/B}/\log_B N)$ 个被 x 范围[q1,q2]横跨的块 si,其 I/O 开销为

$$O(\sqrt{N/B}/\log_B N)\cdot O(\log_B(\sqrt{N/B}/\log_B N)+O(T/B)=O(\sqrt{N/B}+T/B)。$$

最后查询 x 范围[q1,q2]所包含的块 si 中与 y 范围[q3,q4]相交的单元中的 KD 树 IO 开销为

$$2\cdot O(\sqrt{N/B}/\log_B N)\cdot O(\sqrt{B\log_B^2 N/B})+O(T/B)=O(\sqrt{N/B}+T/B)$$

因此 O 树上范围查询的 I/O 复杂度是 $O(\sqrt{N/B}+T/B)$。

2.4 O 树的插入和删除

为了删除一个点 p，我们同样需要执行一个点的查询来找到相关联的 KD 树 T_{ij}（包含 p 的单元）。然后从 T_{ij} 中删除点 p。如果该单元现在包含的点的个数少于 $\frac{1}{4}B\log_B^2 N_0$，我们就把它和它的一个相邻的单元合并。移除这两个单元，同时从这两个单元收集到 $\frac{1}{2}B\log_B^2 N_0\sim\frac{6}{4}B\log_B^2 N_0$ 的点，还要删除包含 p 的块 s_i 中在这两个单元之间的水平分割线。假如收集到的点的数量为 $\frac{1}{2}B\log_B^2 N_0\sim B\log_B^2 N_0$，我们就为这个单元构造一个新的 KD 树。否则，用一条水平的分割线把这些点分为两个部分，即把新的单元分成两个包含 $\frac{1}{2}B\log_B^2 N_0\sim\frac{3}{4}B\log_B^2 N_0$ 个点的单元，并向 T_i 插入一条水平分割线，同时我们为这两个单元构造两个新的 KD 树。相似的，假如包含点 p 的块 s_i 中的点少于 $\frac{1}{4}\sqrt{N_0 B}\log_B N_0$，我们就把它和相邻的块合并，从 T_v 中删除两个块之间的垂直线，并且从这两个块中的 KD 树中收集到 $\frac{1}{2}\sqrt{N_0 B}\log_B N_0\sim\frac{6}{4}\sqrt{N_0 B}\log_B N_0$ 个点。假如收集到的点的数量为 $\frac{1}{2}\sqrt{N_0 B}\log_B N_0\sim\sqrt{N_0 B}\log_B N_0$，我们就可以在该块中使用 $\Theta(\sqrt{N_0 B}/\log_B N_0)$ 条水平线来构造新的单元，每个单元包含点的数量为 $\frac{1}{2}B\log_B^2 N_0\sim B\log_B^2 N_0$。我们为该块创建一个新的 B 树，同时还要为每个单元构建一个 KD 树。否则我们就要先使用一条垂直的分割线来构造两个新的块，每个块中的点为 $\frac{1}{2}\sqrt{N_0 B}\log_B N_0\sim\frac{3}{4}\sqrt{N_0 B}\log_B N_0$，并把它插入 T_v 中，同时在每个块中构建 $\Theta(\sqrt{N_0 B}/\log_B N_0)$ 个单元，每个单元也会包含 $\frac{1}{2}B\log_B^2 N_0\sim B\log_B^2 N_0$ 个点。

我们使用一个全局的重建策略来有效地更新 O 树。假设 N_0 是重建后的结构中的点的数量，我们在 $N_0/2$ 次操作后再次重建的 I/O 开销为 $O\left(\frac{N_0}{B}\log\frac{N_0}{B}\right)$，或者说平摊 I/O 开销为 $O\left(\frac{1}{B}\log\frac{N_0}{B}\right)=O(\log_B N)$。考虑到在每个块中的点的数量为 $\frac{1}{4}\sqrt{N_0B}\log_B N_0\sim\frac{5}{4}\sqrt{N_0B}\log_B N_0$，且每个单元中的点为 $\frac{1}{4}B\log_B^2 N_0\sim\frac{5}{4}B\log_B^2 N_0$，我们就可以以 $O(\log_B N)$ 为平摊 I/O 开销更新 O 树，下面将对此进行具体描述。

为了插入一个点 p，先处理点的查询，找到包含 p 的单元(设对应 KD 树 T_{ij})。然后把 p 插入 T_{ij} 中。假如该单元包含的点的数量超过的了 $\frac{5}{4}B\log_B^2 N_0$，就使用水平线把它分割成两个包含大致 $\frac{5}{8}B\log_B^2 N_0$ 个点的单元，移除原来的单元，同时为这两个新的单元构建两个新的 KD 树。同时将一条新的水平线插入包含 p 的块 s_i 中的树 T_i 中。相似的，假如块 s_i 现在包含超过 $\frac{5}{4}\sqrt{N_0B}\log_B N_0$ 个点，则用一条垂直线把它分割成两个包含大致 $\frac{5}{8}\sqrt{N_0B}\log_B N_0$ 个点的块，在 T_v 中插入一条线，同时在新的块中使用 $\Theta(\sqrt{N_0B}/\log_B N_0)$ 条水平线构建新单元，每个新单元包含点的数量为 $\frac{1}{2}B\log_B^2 N_0\sim B\log_B^2 N_0$。我们丢弃建立在原来的块中水平线上的 B 树 T_i，同时为两个新的块创建两个新的 B 树，最后我们在每个新的单元中的点上构建一棵 KD 树。

O 树更新算法平摊 I/O 开销为 $O(\log_B N)$。

初始化的点查询的 I/O 开销为 0,这样在单元或块没有变得过大或过小的情况下，在一棵 KDB 树中完成更新，其 I/O 开销为 $O(\log_B^2(B\log_B^2 N_0))=O(\log_B N)$。

假如一个单元变得过大或者过小（包含点的个数小于 或者超过 ），便执行 O(1)次分割或者合并，在 B 树 Tv 中的块 si 中的 O（1）次更新中共使用 $O(\log_B(\sqrt{N_0/B}/\log_B N_0))=O(\log_B N_0)$ 次 I/O，同时将使用 $O\left(\frac{B\log_B^2 N_0}{B}\log\frac{B\log_B^2 N_0}{B}\right)=O(\log_B^2 N_0\cdot\log\log_B^2 N_0)$ 次 I/O 移除和构建大小为 $O(B\log_B^2 N_0)$ 的 O(1) 棵 KD 树。因为一个新建的单元包含的点数为 $\frac{1}{2}B\log_B^2 N_0\sim B\log_B^2 N_0$,所以更新操作的平摊 I/O 开销的上界为

$$O(\log_B^2 N_0\cdot\log\log_B^2 N_0)/\left(\frac{1}{4}B\log_B^2 N_0\right)=O\left(\frac{\log\log_B^2 N_0}{B}\right)=O(\log_B N).$$

假如一个块变得太大或者太小，便执行 O(1)次分割或者合并，共将使用 $O(\log_B(\sqrt{N_0/B}/\log_B N_0))=O(\log_B N_0)$ 次 I/O 在 B 树 Tv 执行 O(1)次更新,使用 次 I/O 移除并构建 O（1）个在块 si 中的 B 树 Ti, 并且使用

$O(\sqrt{N_0/B}/\log_B N_0)\cdot O(\log_B^2 N_0\cdot \log\log_B^2 N_0)$次 I/O 移除和构建$O(\sqrt{N_0/B}/\log_B N_0)$个大小为 $B\log_B^2 N_0$ 的 KD 树。因为新建的块包含点的个数为 $\frac{1}{2}\sqrt{N_0 B}\log_B N_0\sim\sqrt{N_0 B}\log_B N_0$，所以更新 I/O 开销的平摊上界为

$O(\sqrt{N_0/B}/\log_B N_0)\cdot O(\log_B^2 N_0\cdot \log\log_B^2 N_0)/\left(\frac{1}{4}\sqrt{N_0/B}/\log_B N_0\right)=O\left(\frac{\log\log_B^2 N_0}{B}\right)=O(\log_B N)$。,因此 O 树更新算法平摊 I/O 开销为$O(\log_B N)$。

3 写出对有向图 G=（V，E）进行外存 BFS 的伪代码，给出其 I/O 开销的复杂度，并进行简单的说明

3.1 伪代码

算法 2 有向图BFS算法的伪代码

输入: 有向图G=(V,E),搜索起点s

输出: 无

```
T=ϕ
Q=s
while Q != ϕ do
    V=TOP(Q)
    E=Extract(T,V)
    for each(v,ω)∈ E do
        Delete(P(v),(v,ω))
    end for;
    if P(v)==ϕ then
        POP(Q)
    else
        repeat
            (v,ω)=DeleteMin(P(V))
            PUSH(ω)
            for each(x,ω)∈ E do
                INSERT(T,(x,ω))
            end for
        until P(v)! =ϕ
    end if
end while
```

3.2 分析

（1）算法设计：

用一个队列 P 代替 DFS 的栈决定图 G 的定点的访问顺序。

当访问顶点 v 时，从 T 中提取出来 v 指向访问过的定点的出边，并从 P（v）中删除这些边，用一系列的 DeleteMin 操作得到 P（v）中保留的边。

对每一条得到的边（v，w），定点 v 为 w 的父节点，定点 w 被添加到队列的

末尾，w 的所有入边被插入到 BRT 中。

一旦优先队列 P（v）以这种方式被清空，下一个要被访问的定点从队列中被检索出来。

（2）复杂度分析：

I/O 开销为

$$O((|V|+|E|/B)\log_2|V|)。$$

证明：

该算法的 I/O 复杂度主要是花费在更新 BRT、优先队列和访问邻接表的 I/O 开销$O(|V|+|E|/B)$

因为邻接表的每个顶点只被访问一次，访问 G 的所有顶点的邻接表的 I/O 开销为

$$O(|V|+|E|/B)$$

在一开始时，算法在队列上执行$O(|E|)$次操作。

对于初始状态，G 的每条边(v,w)被插入到一个优先队列即 P(v)中。

初始化之后，队列上只有 DeleteMin 和 Delete2 种操作，而在所有优先队列为空之前只执行了$|E|$次优先队列上的操作。如果用缓冲库树实现堆叠，那么在主存中可以构建一个大小为 B 的缓存，总 I/O 开销为$O(sort(|E|))$。

对于$|V|$个不同的优先队列，假设$|V|B>M$，那么算法要为当前顶点 v 的队列生成一个大小为 B 的缓存。另一个顶点活动前，无论当前顶点优先队列 P（v）有多少元素，其缓存也将被清空。另外，该清空操作对顶点本身而言的 I/O 开销为 O（1）。又因为算法访问不同顶点的次数是$O(|V|)$。因而，该阶段总 I/O 时间是$O(|V|+sort(|E|))$。

最后，算法在 BRT 上执行$O(|E|)$次 Insert 操作和$O(|V|)$次 Extract 操作。而，每个 Insert 操作的平摊 I/O 开销是$O(\frac{1}{B}log_2\frac{|E|}{B})=O(\frac{1}{B}log_2|E|)$。每一个 Extract 操作的平摊 I/O 开销是$O(log_2|V|)$。因此，花费在更新 BRT 上的总 I/O 开销是 $O((|V|+|E|/B)log_2|V|)$。

综合上述分析，BFS 的总 I/O 开销是

$$O((|V|+|E|/B)log_2|V|)$$

4 参考文献

[1] 一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前 10 个词，请给出思想，给出时间复杂度分析
https://blog.csdn.net/gamesofsailing/article/details/18040583
[2] 经典算法题：大数据处理常见算法题
https://blog.csdn.net/wangbaochu/article/details/52998175?utm_medium=distribute.pc_relevant.none-task-blog-baidujs_title-0&spm=1001.2101.3001.4242
[3]王宏志. 大数据算法[M]. 机械工业出版社, 2015.