面向TopK聚合的剪枝聚合算法

- 作者：李猛、柴华溢
- 单位：南京大学计算机科学与技术系

- 日期：2024年05月22日

1 背景

在面对带有 ORDER BY + LIMIT 的聚合（后面简称TopK聚合）任务时，往往实际只需要输出前TopK个聚合结果，而不需要输出全部的聚合结果，如下所示：

```
# Clickbench数据集名为hits，其中，UserID列的NDV约为17M
# 这里我们想输出出现频率最高的10个UserID
SELECT
  UserID, COUNT(*) AS c
FROM hits
GROUP BY UserID
ORDER BY c DESC
LIMIT 10;
```

如果按照传统的聚合方法，首先需要根据GROUP KEY进行全量的聚合，之后采用小顶堆的方式完成ORDER BY和LIMIT，求出TopK的结果。这个方法虽然通用，但很明显，大量的GROUP被聚合后并没有输出给上层算子，即大量的计算过程是浪费的。这里可以举个例子，在Clickbench数据集中，UserID列的NDV为17M，也即Number of Distinct UserID为1700万，但上述SQL只需要出现次数最多的前10个UserID，那么执行引擎实际会聚合出一个含有17M数据的哈希表，然后挑选其中value最大的前10条输出给上层算子，剩余的17M-10条数据被丢弃。

现阶段常见的TopK聚合任务有以下特点：

- TopK聚合任务往往需要输出的数据条数少于100；
- 聚合的数据集规模大，待聚合列的NDV远超CPU Cache的大小，聚合规模往往在GB级别，因此Cache Miss较为严重。

综上，我们考虑是否可以设计一种通用的面向TopK聚合任务的算法，核心目标是尽量避免那些不属于TopK的GROUP参与聚合过程，从而提升聚合速度。例如，在上述例子中，如果我们能在使用哈希表聚合前，识别出一些完全不可能是TopK的GROUP，并禁止这些GROUP插入到哈希表，则可以有效减小哈希表的大小以及显著提升聚合速度，假设我们能够识别出15M的GROUP一定不属于TopK，则最终只有剩余2M的GROUP被实际插入哈希表做聚合，哈希表的数据处理量减少接近8倍。

2 相关研究

本节介绍几种常见的哈希聚合算法

2.1 单线程，哈希聚合算法

1. 构建一个哈希表
2. 下层算子每输入一个Row，就提取这个Row的GROUP KEY，插入或更新哈希表

3. 循环执行步骤2，直到下层算子无输出为止，则聚合算子执行完毕
4. 执行ORDER BY和LIMIT的相关算子（往往封装为一个TopK算子）

2.2 多线程，全局共享哈希表算法

1. 构建一个全局多个线程共享的哈希表
2. 聚合过程与算法2.1类似，区别在于每个线程负责一部分的数据扫描和哈希表插入工作
3. 执行ORDER BY和LIMIT的相关算子（往往封装为一个TopK算子）

缺点：全局共享哈希表，并发性能可能受限

2.3 单线程，基于Shuffle的局部哈希表聚合算法

1. 设置一个分区数量n_partitions
2. 对于下层算子输入的每一个Row，根据其GROUP KEY将这个Row放入对应的分区，即用Hash(GROUP Key) % n_partitions确定分区的下标p，然后将这个Row放入分区p中
3. 步骤2执行完毕后，相同的GROUP KEY一定被划分到同一个分区。此时，只需要依次对每个分区执行局部哈希聚合即可
4. 遍历所有的局部哈希表，执行TopK算子

优点：相较于算法2.1，可以控制每个分区GROUP的NDV能够被Cache容纳，从而提升局部哈希表的聚合速度

缺点：增加了shuffle的代价，且shuffle的代价高于聚合的代价

2.4 多线程，基于Shuffle的局部哈希表聚合算法

1. 基本过程与算法2.3一致
2. 对于算法2.3中的步骤2和3，都可以用多线程的方式加速

优点：并发无锁

3 本文算法

3.1 基础概念

3.1.1 数据分区（又称shuffle或partition）

对于给定的一组数据，按照key哈希取模后，将数据写入对应分区，如下面的伪代码例子所示：

```
struct Row {
  int64_t key_;
  char *payload_;
```

```
};

void shuffle(const std::vector<Row> &raw_data,
             std::vector<std::vector<Row> &partitions>,
             const int64_t n_partitions) {
  partitions.resize(n_partitions);            // 初始化分区
  for (int64_t i = 0; i < raw_data.size(); ++i) {
    auto &row = raw_data[i];
    auto pos = hash(row.key_) % n_partitions; // 计算所属分区
    partitions[pos].push_back(row);           // 写入所属分区
  }
}
```

显然，所有key相同的Row一定会被写入相同的分区

3.1.2 细粒度聚合

对于给定的一组数据，使用哈希表对其进行**准确聚合**，如下面的伪代码例子所示：

```
struct Row {
  int64_t key_;
  char *payload_;
};

void fine_grained_agg(const std::vector<Row> &raw_data,
                      std::unordered_map<int64_t, AggResult> &ht) {
  for (int64_t i = 0; i < raw_data.size(); ++i) {
    auto &row = raw_data[i];
    auto it = ht.find(row.key_); // 查询key是否插入过
    if (it == ht.end()) {        // 若未插入过，则初始化聚合结果
      ht[row.key_] = InitAggResult();
    } else {                     // 若插入过，则更新聚合结果
      UpdateAggResult(it->second, row.payload_);
    }
  }
}
```

这与第2节种所提到的聚合概念一致，就是实际去完成聚合

3.1.3 粗粒度统计

对于给定的一组数据，遍历一遍，记录这组数据的一些统计信息，包括数据总行数、聚合列的最大值、聚合列的最小值、聚合列的和、聚合列的NDV等。

下面给出一个示例：假设进行如下查询，我们需要对给定的一组数据做COUNT(*)的统计信息，这里的一组数据，其实就是3.1.1节中的一个分区

```
# Clickbench数据集名为hits，其中，UserID列的NDV约为17M
# 这里我们想输出出现频率最高的10个UserID
SELECT
  UserID, COUNT(*) AS c
FROM hits
GROUP BY UserID
ORDER BY c DESC
LIMIT 10;
```

统计信息的计算方式如下

```
struct Row {
  int64_t userid;
};

// 注：这个示例sql只涉及到COUNT(*)的查询，
// 因此这里的统计信息只简化为一个count_，
// 如果SQL聚合查询还包含SUM、MIN、MAX等，
// 则再添加相应的统计信息即可
struct Statistic {
  int64_t count_;
};

void coarse_grained_agg(const std::vector<Row> &raw_data,
                        Stastic stat) {
  stat.count_ = 0;
  for (int64_t i = 0; i < raw_data.size(); ++i) {
    auto &row = raw_data[i];
    stat.count_++; // 这里只展示记录这组数据的行数
  }
}
```

3.2 分区剪枝

粗粒度的统计信息用于分区剪枝

- 假设我们现在采用第2.3节的算法（单线程，基于shuffle的局部哈希表聚合算法），shuffle出了1024个独立的分区partitions[0..1023]
- 每个分区在做shuffle的过程中，顺便做了3.1.3节的粗粒度统计，得到stats[0..1023]
- 随后，按照2.3节的算法，我们应该从partitions[0]开始，到partitions[1023]为止，为每个分区做细粒度聚合
    - 我们对partitions[0]做完细粒度聚合后，就能对这部分的聚合结果做TopK计算，假设partitions[0]的聚合结果中，第TopK个聚合结果的COUNT(*)为4000，此时，我们将4000设置为TopKThreshold，即TopK阈值。
    - 下面对partitions[1]做细粒度聚合，但在聚合之前，我们要检查一下stat[1]是否大于TopKThreshold，即判断stat[1]>4000。若为真，则去做细粒度聚合，并更新TopKThreshold；若为假，则跳过该分区。
    - 后面的分区都是先检查是否满足TopKThreshold条件，决定是执行细粒度聚合还是跳过。

这个TopKThreshold过滤的原理是什么呢?

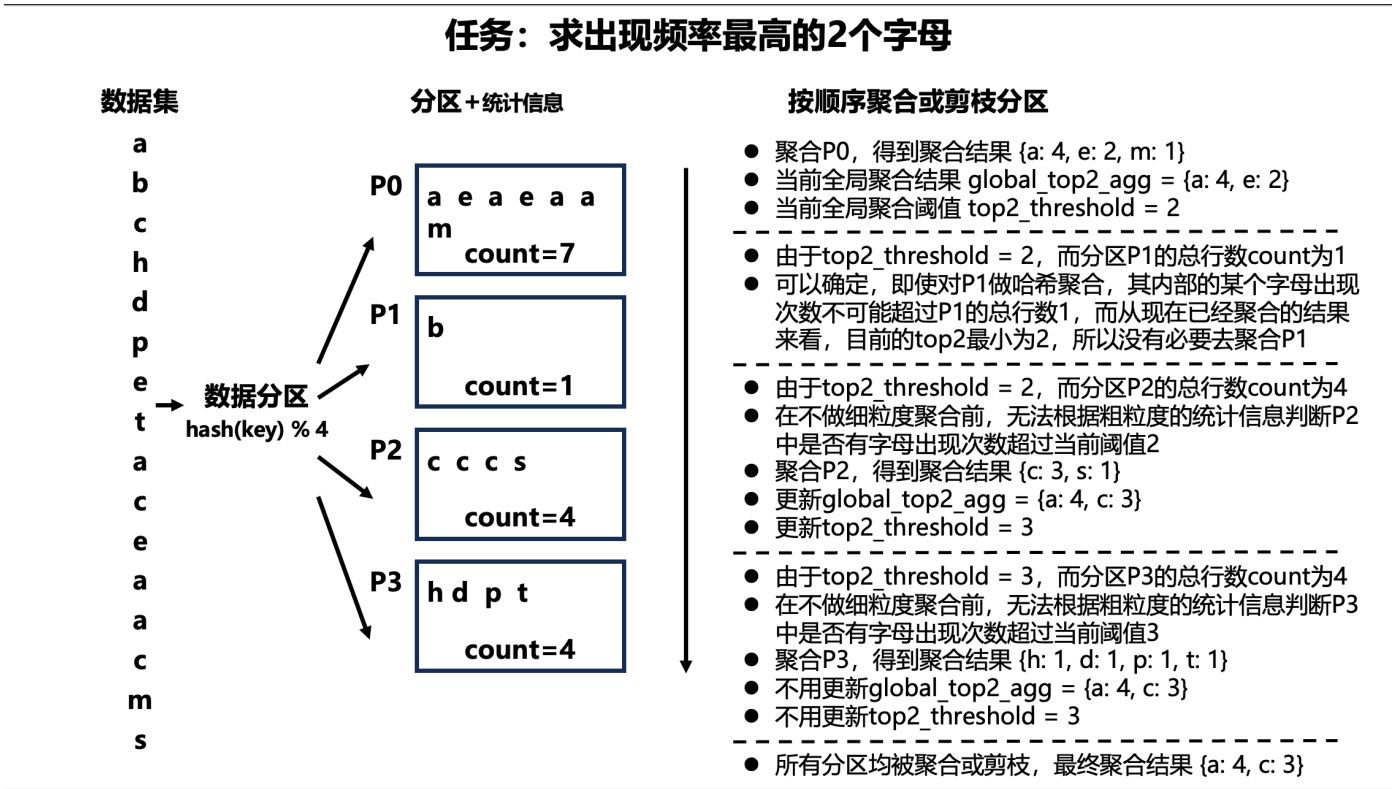
- 这里拿COUNT(*)来举例子
- 我们对partitions[0]使用细粒度聚合，则可以得到partitions[0]中每个GROUP的准确COUNT(*)聚合结果，假设partitions[0]中第TopK个COUNT(*)的值是T，我们将这个值作为全局阈值
- partitions[1]我们在shuffle阶段就已经做了粗粒度统计，即统计了partitions[1]中的总COUNT数为多少，注意，这里没有做任何聚合。如果partitions[1]的总行数COUNT都小于partitions[0]中第TopK个COUNT，即partitions[1].COUNT < T，则即使对partitions[1]做细粒度聚合，其内部的任何一个GROUP的COUNT一定小于等于partitions[1].COUNT小于T，则一定不可能属于TopK，因此，在这种情况下，完全没有必要对partitions[1]做细粒度聚合，直接过滤掉即可
- 同理，假如partitions[2]的统计信息COUNT大于等于T，则partitions[2]中才有可能有GROUP的COUNT大于T，则对partitions[2]执行细粒度聚合，然后更新全局聚合结果，并找出找出其中第TopK个聚合结果，如果TopK.COUNT > T，则更新T值
- 后续的partitions[3..1023]处理流程与上面一致，都先将partitions[i].COUNT与T进行比较，决定是否再执行细粒度聚合

3.3 单线程剪枝算法流程

综上，根据3.1和3.2的内容，这节介绍我们单线程版本的算法流程

1. 设置分区数为n_partitions，执行shuffle算法，将所有的数据均匀打散到这些分区中，可以得到分区partitions[0 ... n_partitions]
2. 从partitions[0]开始，按照顺序处理每一个分区，如果这个分区没有被TopKThreshold过滤掉，则执行细粒度聚合，并更新TopKThreshold。

具体流程看如下示例



如上图所示，任务目标是统计出现频率最高的2个字母，答案是{a: 4次，c: 3次}

1. 首先将左侧数据集shuffle到4个分区
2. 遍历分区P0～P3，根据全局阈值top2_threshold和每个分区的p.count决定分区是否被剪枝，还是做细粒度聚合

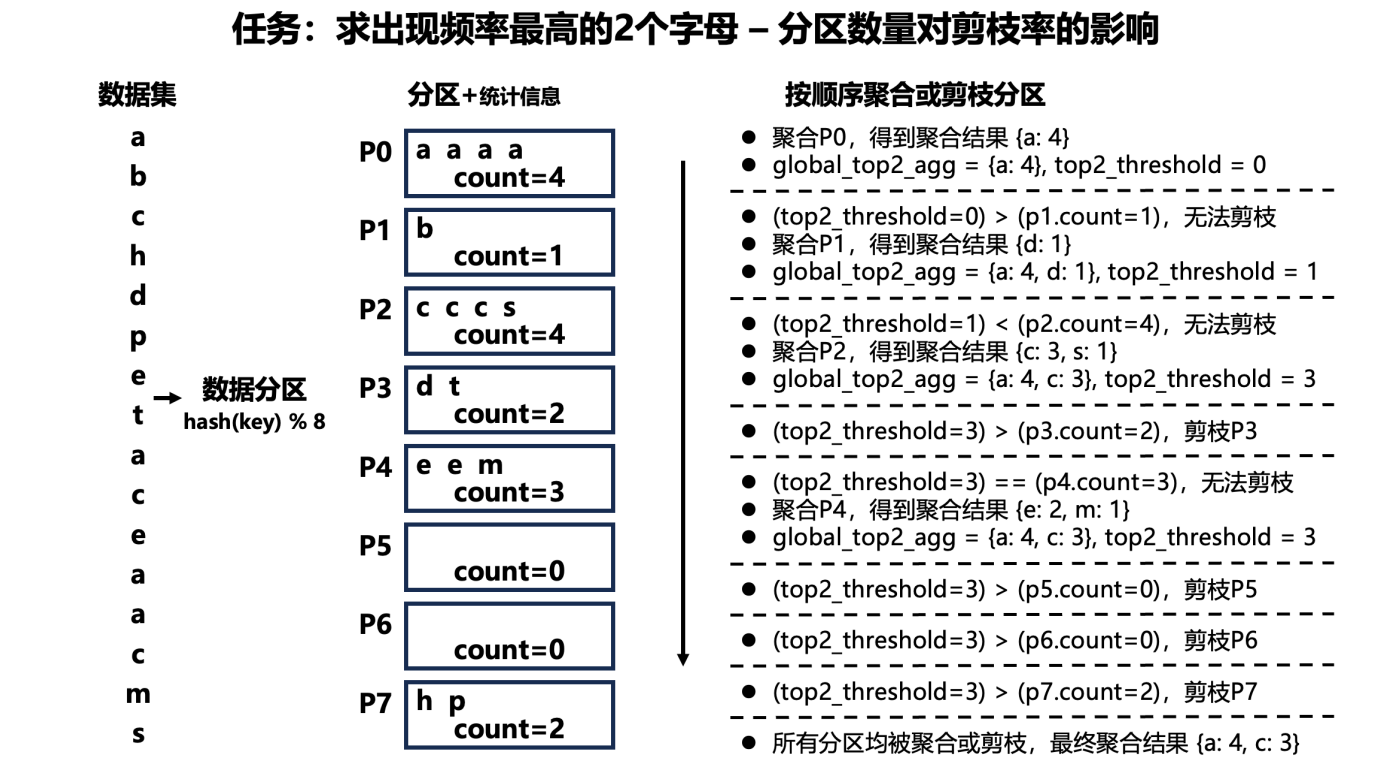
在这个例子中，只有分区P1被剪枝，分区P0P2P3都执行了细粒度聚合

3.4 挑战

简单地使用3.3节描述的算法在一些情况下效果并不好，剪枝率会比较低，从而达不到预期的性能提升。本节介绍两个对于3.3节算法Bad Case的情况。

3.4.1 分区数量对剪枝率的影响

在shuffle时，如何设置分区数n_partitions对于剪枝率有很大影响，可以拿下图和3.3节的图做对比，两个示例的数据集是一样的，只是分区数从4变为8。在分区数为4时，剪枝率为25%，在分区数为8时，剪枝率为50%

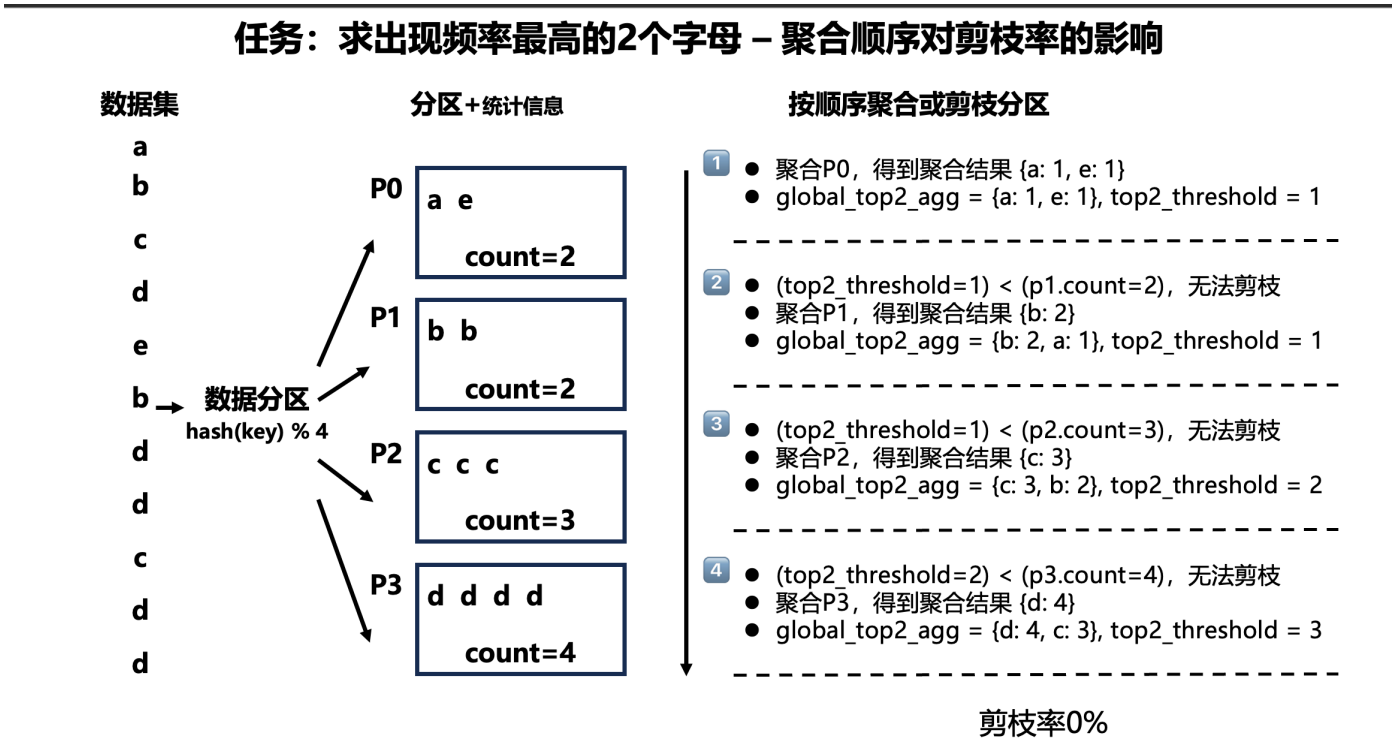


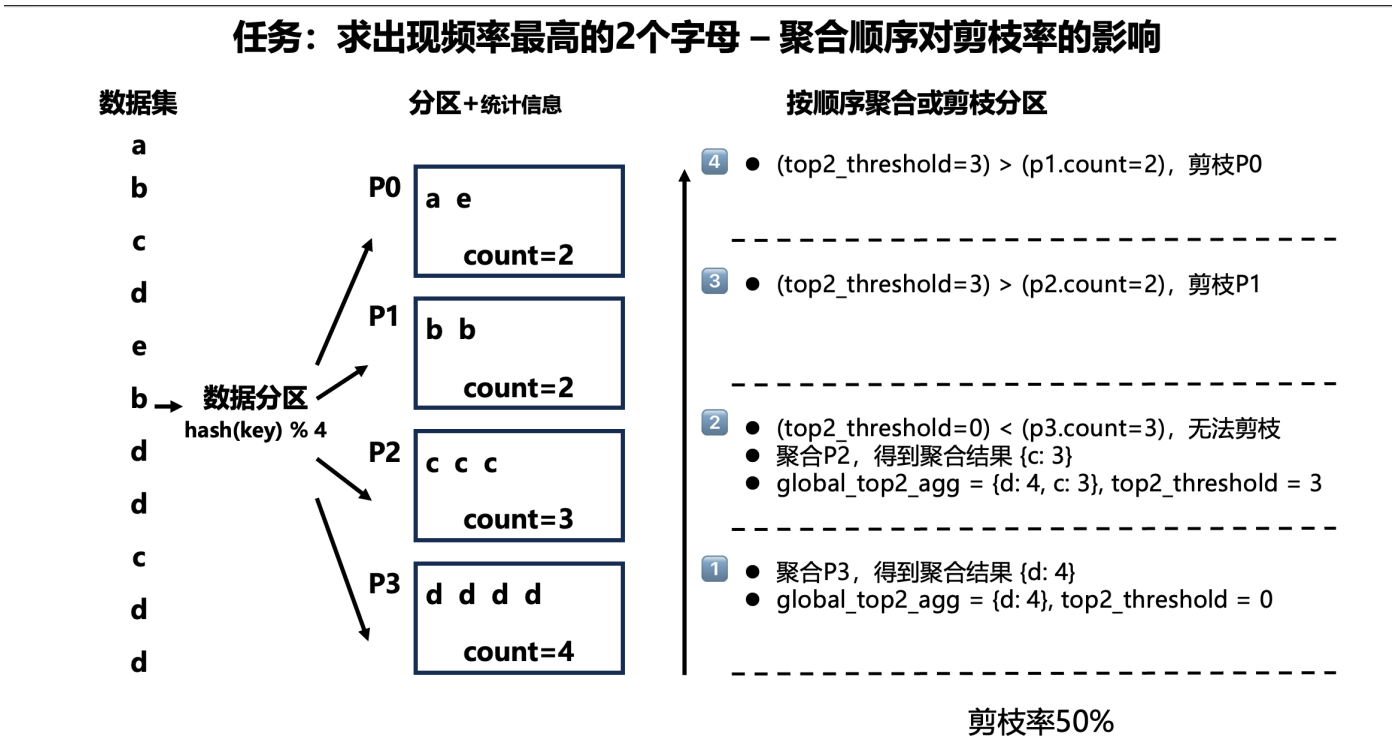
为什么分区数量对剪枝率有影响呢?

- 如果分区数量过少，则每个分区的数据量将非常大，由于分区的统计信息粗略地认为整个分区是一个GROUP，那么分区内的数据量越多则统计信息越不准
- 以COUNT(*)聚合为例，当分区数过少时，最终聚合结果的第TopK个COUNT(*)很有可能远小于每个分区的数据量，因此，分区数过小时，剪枝率往往在0%
- 在分区这个角度来看，分区数越大越好，分区越大，每个分区内的数据量越少，则每个分区的统计信息越准，使得剪枝率越高
- 当分区数达到一定地步时，剪枝率无限接近于100%，此时聚合的代价几乎可以忽略，但分区的代价就随着分区数的增大而提高

3.4.2 分区聚合顺序对剪枝率的影响

在3.3节的介绍中，shuffle完成后，我们从partitions[0]开始，按照顺序去聚合或剪枝分区，但是下面的例子展示了在顺序聚合时对剪枝率的影响，在相同数据集和数据分区下，第一个图是从上往下聚合，第二个图是从下往上聚合，两者的剪枝率分别为0%和50%





- 如果在扫描阶段的前期就聚合到接近真实的第TopK的GROUP，则后续被扫描到的分区大概率能被剪枝

- 相反，如果真实的第TopK个GROUP被最后扫描到，那么前面的大部分分区往往都得不到剪枝，因此导致剪枝率接近0%
- 那么如何确定分区的聚合顺序，尽量让那些包含前TopK个GROUP的分区先聚合，会极大影响剪枝率

3.4.3 数据分区耗时

经过在实际系统中的实验验证，我们发现分区部分的耗时超过聚合部分耗时，即可以理解为，2.3节介绍的算法中步骤2的耗时大于步骤3的耗时

- 目前的分区算法，涉及到大量数据的随机拷贝
- 特别是在多轮分区的情况下，这个现象更加明显

3.5 优化

本节介绍对3.4节提出的两个问题的解决方案。

3.5.1 根据统计信息的排序来确定分区聚合顺序

- 当shuffle分区并且统计完毕每个分区的统计信息后，按照统计信息对分区进行排序（例如，对于COUNT(*)的聚合任务，此时每个分区的统计信息是分区内数据的行数，此时我们按照每个分区的数据行数从高到低进行排序）
- 我们从排序结果中挑选出前2*TopK个分区，执行细粒度聚合，随后剩余的分区再按照顺序执行聚合或剪枝

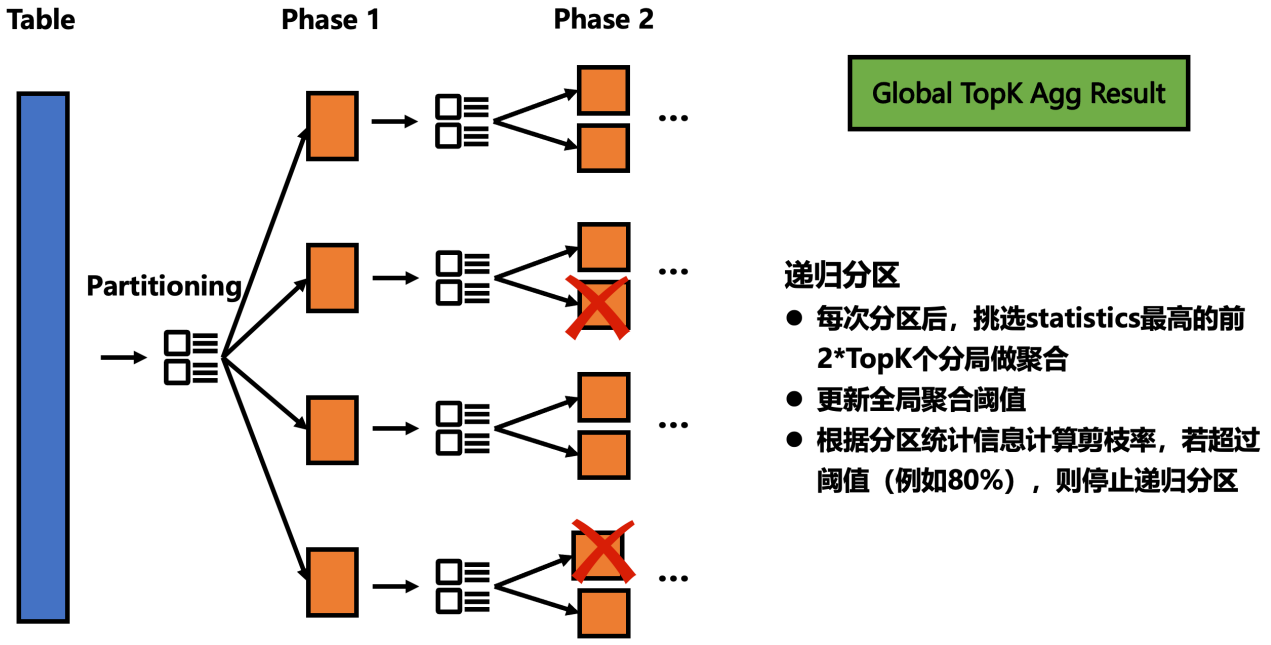
这里我们认为，如果一个分区的粗粒度统计信息排名越靠前，则其中包含TopK数据的概率越高，而由于shuffle的影响，前TopK个GROUP很有可能分散在TopK个不同的分区，因此这里取2*TopK个分区去首先做细粒度聚合，就是期望（并不能保证）能够拿到真实的第TopK个GROUP所属的分区，从而最大化剪枝率。

3.5.2 递归分区确定分区数

采用递归分区的方式来确定合适的分区数

递归停止的条件有两个：

- 当分区剪枝率超过阈值（例如80%时），停止递归
- 当分区数达到阈值时，停止递归



如上图所示：

- 在phase1第一轮分区时，挑选出了2*TopK个候选分区做细粒度聚合，并将结果更新到全局GlobalTopKAggResult中，由于分区数较少，没有任何剪枝，未达到停止分区的条件
- 在phase2第二轮分区时，将第一轮的分区中那些没有被执行细粒度聚合的分区做进一步分区，并且再挑选2*TopK个候选分区做细粒度聚合，并将结果更新到全局GlobalTopKAggResult中，在第二轮中，有剪枝率25%，未达到停止分区的条件
- phase 3...，不断递归下去，直到达到最大分区数，或者所有分区均被聚合或剪枝为止

3.5.3 避免执行数据物理分区

前面提到了数据分区耗时较高，是由于涉及了大量的随机内存拷贝，那么我们是否可以不执行真实的数据拷贝，只更新分区的统计信息呢?

- 经过实验观察，顺序扫描一遍内存中的所有数据的代价远小于一次数据分区的代价
- 因此，没有必要对数据执行真实的物理分区，而是只对每个分区做3.1.3节介绍的粗粒度统计即可
- 算法：
    1. 递归多轮
    2. 每轮扫描原始数据，并对它们做逻辑分区（不实际移动数据，只做分区级别的粗粒度统计信息收集）
    3. 每轮都挑选统计信息排名最靠前的TopK个分区作为候选分区，执行细粒度聚合，并将聚合结果更新到全局聚合结果中
    4. 每轮做完TopK个分区的细粒度聚合后，执行剪枝流程，尝试根据每个逻辑分区的统计信息进行剪枝，如果剪枝率达到要求，则停止递归
    5. 下一轮，分区数增多，重复执行步骤2-4

4 实验结果

4.1 实验设置

环境

- OS：Ununtu 22.04
- CPU：Intel(R) Xeon(R) Gold 5318Y CPU @ 2.10GHz * 2
- Core：16
- RAM：32GB
- GCC：g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0

算法

- 数据集均为100M大小，即1亿条uint64_t的数据
- 查询：SELECT userid, count(*) AS c FROM table GROUP BY userid ORDER BY c DESC LIMIT 10;
- 优化级别 -O2
- Agg1为2.1和2.2节介绍的方法
- Agg2为2.3和2.4节介绍的方法
- Agg3为本文方法

4.2 单线程实验结果

数据集	Agg1	Agg2	Agg3（ours）
userid（100M）	7.962264s	6.107236s	0.884809s

4.3 多线程实验结果

数据集	线程	Agg1	Agg2	Agg3（ours）
userid（100M）	1	7.962264s	6.107236s	0.884809s
userid（100M）	4	5.320061s	1.340502s	0.293516s
userid（100M）	8	4.114508s	0.793166s	0.188613s
userid（100M）	16	3.434719s	0.582622s	0.124328s

4.4 总结

- 本文算法相较于传统的单哈希表法（Agg1），最高有28倍的性能提升
- 本文算法相较于分区聚合算法（Agg2），最高有7倍性能提升