基于ORC元数据的Hive Join查询Reducer负载均衡方法

王华进[1,2] 黎建辉[1] 沈志宏[1] 周园春[1]
(中国科学院计算机网络信息中心 北京 100190)[1] (中国科学院大学 北京 100049)[2]

摘 要 负载不均衡问题位列影响大规模MapReduce集群性能因素的首位,而Hive join查询非常容易触发该问题。通用解决方案是基于中间键值对的key频率分布设计能够实现负载均衡的key划分算法。现有工作估算key频率分布时依赖于对map的输出进行监控采样,使得通信开销较大并显著延后了shuffle的启动。针对Hive join查询,提出了基于ORC元数据的key频率分布估计方法和相应的负载均衡key划分方法。该方法具有计算量小、通信开销小、不影响现有shuffle机制的优点。通过基准测试证明了该方法在key频率分布估算效率上的巨大提升及相应的key划分方法对Hive join查询性能的提升。

关键词 负载均衡,MapReduce,Hive,Join,Reducer,ORC
中图法分类号 TP391 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2018.03.025

ORC Metadata Based Reducer Load Balancing Method for Hive Join Queries

WANG Hua-jin[1,2] LI Jian-hui[1] SHEN Zhi-hong[1] ZHOU Yuan-chun[1]
(Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China)[1]
(University of Chinese Academy of Sciences, Beijing 100049, China)[2]

Abstract The load imbalance problem ranks first among the performance issues in large-scale MapReduce cluster, and it's very prone to be triggered by Hive join queries. An effective solution is to design reducer load balancing partitioning algorithms by consulting the key's frequency distribution histogram estimated from intermediate key-value pairs. The existing works of key histogram estimation rely on monitoring and sampling the output of map in a distributed way, which triggers huge network traffic load and notably delays the start of the shuffle. A novel key histogram estimation method based on ORC metadata and the corresponding load balancing partitioning strategy was proposed for Hive join queries. The proposals only need some light-weight computation before the start of the job, thus imposing no extra loads on network traffics and the shuffle. Benchmarking test proves the proposal's significant improvement on both the key histogram estimation and the reducer load balancing.

Keywords Load balancing, MapReduce, Hive, Join, Reducer, ORC

1 引言

负载不均衡是影响大规模MapReduce集群性能的首要因素。首先,负载不均衡导致集群的资源利用率变差:在少数计算资源被长期占用的同时,大量资源处于等待任务的状态。其次,MapReduce作业的运行时间取决于运行最慢的任务(即straggler),而straggler往往是负载最重的任务。因此,改善任务的负载均衡对提升MapReduce集群的资源利用率和优化作业的运行效率均具有重要的意义。

MapReduce作业的负载不均衡问题突出表现在reduce阶段。在map阶段输出的中间键值对中,拥有相同key的键值对被分配到同一个分区,一个分区启动一个reducer。MapReduce默认采用了一个简单的hash划分器对key进行划分:以key的hash值对目标分区的数量取模,结果即为该key对应的分区。这种方案在key的频率分布(以下简称key分布)比较均匀时能够基本保证reducer的负载均衡。但当key的分布严重倾斜时,负责较多高频key的reducer需要处理更多的中间键值对,从而导致这些reducer的负载远重于其他reducer。

Hadoop是当前MapReduce框架的主流实现,Hive是构

到稿日期:2017-01-15 返修日期:2017-06-15 本文受国家重点研发计划项目:科学大数据管理系统(2016YFB1000600),协同精密定位技术(2016YFB0501900)资助。

王华进(1987-),男,博士生,主要研究方向为大数据处理技术;**黎建辉**(1973-),男,高级工程师,博士生导师,主要研究方向为科学大数据管理、数据出版、数据挖掘,E-mail:lijh@cnic.cn(通信作者);**周园春**(1975),男,研究员,博士生导师,主要研究方向为大规模数据挖掘。

建于 Hadoop 之上的一个主流数据仓库查询引擎，能够将 SQL 语句转换为 MapReduce 作业来执行。Optimized Row Columnar(ORC)是一种广泛地应用在 Hive 集群中的列存储文件格式，可以提高查询宽表的少数列的效率。由于对宽表的少数列进行查询是 Hive 负载的主要构成部分，因此 ORC 格式逐渐成为了 Hive 的主流数据存储格式。在 Hive 中，join 查询受 reducer 负载不均衡的影响最为显著。这是因为 join 查询需要将参与 join 的字段(以下简称 join 字段)的全部数据发送到 reducer 端进行笛卡尔积的计算，因此 reducer 的负载很重，且在不恰当的划分下很容易造成 reducer 负载不均衡。本文研究基于 ORC 文件的 Hive join 查询 reducer 负载均衡问题。

以往 reducer 负载均衡的研究主要通过对 mapper 输出的中间键值对进行监控采样以估算 key 分布，据此来构建 reducer 负载均衡的 key 划分算法。监控采样会显著增加网络通信开销，为保证作业的执行效率，采样率不能过高，因此构建的直方图的准确性受到了限制。此外，MapReduce 作业的 shuffle 过程必须在采样、key 分布估算、key 划分等工作完成后才能启动，因此作业的执行时间被潜在延长了。

本文的主要贡献是提出了一种基于 ORC 元数据的 key 分布估算方法以及相应的负载均衡的 key 划分方法：在执行 join 查询前，首先访问 join 字段的 ORC 元数据以估算 key 频率分布，然后据此来构造负载均衡的 key 区间划分。

与现有工作相比，本文方法有两个显著优点：1)key 分布的估算效率大幅提升；2)不影响现有 shuffle 机制。此外，该方法简单易行，通过调用 Hadoop 的 API 将生成的负载均衡的区间划分方案传递给 Hive 的执行引擎，不涉及对现有框架的修改。由于 Hadoop 支持为某个作业单独指定 key 划分器，其他作业依旧可以采用默认的 hash 划分器。

2　相关工作

表 1 总结了现有 reducer 负载均衡的研究工作。其中大部分工作均可分为两个阶段：1)估算 key 分布；2)基于 key 分布设计负载均衡的 key 划分。

表 1　现有的 reducer 负载均衡方法

Table 1　Existing reducer load balancing methods

现有工作	key 分布估算	key 划分
Kwon 等人[1]	预先	区间划分
Hadoop InputSampler	预先	区间划分
LEEN[4]	运行时	逐 key 划分
TopCluster[5]	运行时	区间划分
LIBRA[6]	运行时	区间划分
Ke 等人[2]	预先	hash 再划分
Yan 等人[3]	运行时	hash 再划分
王卓等人[7]	运行时	hash 再划分
SkewTune[8]	—	自适应划分

key 分布估算又可分为两类：1)预先估算，在运行正式的作业之前，预先运行一次 MapReduce 作业或专门的计算任务，对 key 进行采样并统计 key 频率。显然预先估算会延后作业的启动时间。2)运行时估算，首先直接在目标作业运行时，监控每个 mapper 的输出并对 key 进行采样；然后将样本汇总给一个中央节点，构建全局的 key 分布。这种监控采样的采样率不能太高，否则会导致 map 阶段的网络拥堵[5]，因此估算出的 key 分布的准确性受限。此外，shuffle 被延后执行：在默认配置下，为充分利用 mapper 执行过程中闲置的网络带宽，Hadoop 在 5%的 mapper 执行完成后即启动 shuffle；而当引入监控采样后，shuffle 必须延到采样、key 分布估算、key 划分等工作完成后才能启动，作业的执行时间被潜在地延长了。

Key 划分方法可分为 4 类：1)区间划分，将 key 的取值区间切分为负载(key 频率之和)相同的子区间；2)逐 key 划分，依次将最高频的未分配 key 分配到负载最轻的分区上；3)hash 再划分，在 hash 划分的基础上合并小分区、拆分大分区，从而改善划分的负载均衡程度；4)自适应划分，调整 straggler 未处理数据的划分，以减轻 straggler 的负载。

Kwon 等人[1]提出在样本数据上预先执行一次目标作业，以采集 key 取值区间并进行区间划分。Hadoop 自带的 InputSampler. RandomSample 是目前最为常用的区间划分工具，其在执行目标作业前，先在 1 个节点上执行如下操作：1)从全部输入数据块中选出要进行采样的块；2)流式读取每个采样块并从中随机选取样本；3)计算出全部样本的 key，并按 key 值的大小对全部样本进行排序；4)将样本序列的等分位点作为区间划分的分割点。

LEEN[4]基于监控采样的结果进行逐 key 划分。由于需要记录每个 key 归属的分区，通过逐 key 划分得出的分区方案很复杂，用于查找 key 所属分区的开销太大，因此其并不实用。TopCluster[5]和 LIBRA[6]基于监控采样的结果进行区间划分。TopCluster 只对每个 mapper 输出中的高频 key 计算全局频率以减少通信开销。LIBRA 只对最先运行的一小部分 mapper 的输出进行采样以减轻对 shuffle 的拖延。

Ke 等人[2]提出预先执行一次目标作业，从而对 hash 划分的结果进行调整，以优化负载均衡。这种粗粒度的 hash 再划分虽然开销小，但不一定能够实现较好的负载均衡。为扩大了 hash 再划分的负载均衡提升空间，Yan 等人[3]提出了基于监控采样的多 hash 再划分：1)在 map 阶段同时使用多种 hash 函数对全体数据的 key 进行细粒度的划分；2)依次对每种 hash 划分的细粒度分区进行组合，得出负载尽量均衡的候选划分方案；3)从候选划分方案中选出负载最均衡者作为最终划分方案。王卓等人[7]提出了基于监控采样的多 hash 渐进再划分：1)在 map 的早期阶段就把部分分区分配给 reducer；2)随着 mapper 的执行，不断统计每个分区的负载量，并据此逐渐将剩余的分区均衡地分配给 reducer。与多 hash 再划分相比，多 hash 渐进再划分可以提早启动 shuffle。

SkewTune[8]是一种自适应划分方案：只为剩余运行时间最长的 reducer(即 straggler)的未处理数据重新划分 key，从

而将 straggler 的部分负载迁移到其他节点上。对于小作业，SkewTune 的重划分开销不可忽略[8]。此外，自适应划分只能在现有划分的基础上改进负载均衡，并不一定能够实现 reducer 的负载均衡。

本文提出的方法属于预先估算/区间划分。与现有的预先估算方案相比，key 分布估算基于 ORC 的元数据完成，不通过 MapReduce 作业构建，开销可以忽略不计(见图 7)。由于没有增加作业的运算环节，在同等负载均衡水平下，作业运行效率也优于运行时的估算方案。Hive 查询 ORC 文件的流程如图 1 所示。

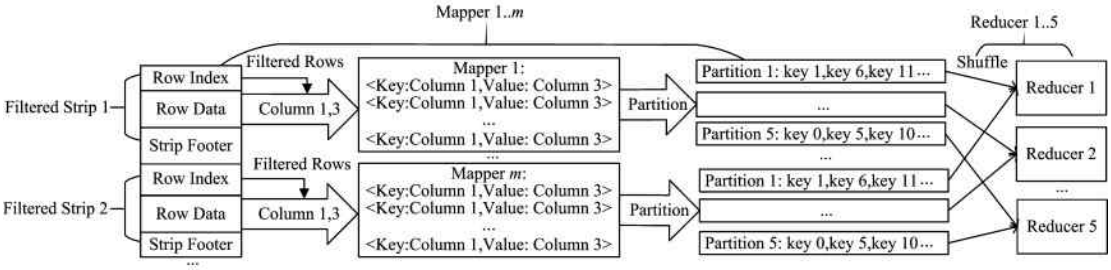


图 1 Hive 查询 ORC 文件的流程

Fig. 1 Process of Hive querying ORC files

3 背景：基于 ORC 文件的 Hive Join

3.1 ORC 文件格式

ORC 文件格式遵循“先对行分组，再按列存储”的理念：数据以 strip 为逻辑单位分为行组，strip 的大小一般等同于 HDFS block 的大小，这就保证了 strip 内的所有行都能保存在同一个节点上；Strip 内再以列为单位顺序存储。这样既可以保证查询部分列时只读取该列的数据，同时又保证属于同一行的所有列都保存在同一个节点上，避免了查询多个列时要跨节点通信组装元组的情况。

如图 2 所示，ORC 元数据主要由 4 部分组成：Postscript，File Footer，File metadata，Strip row index。Postscript 存储文件的长度和 File metadata 的长度信息；File Footer 存储整个文件中各列的最大值、最小值、记录数量等信息；File metadata 存储每个 strip 的最大值、最小值、记录数量等信息；stripe row index 存储了该 strip 的每列每 10000 行记录的最大值、最小值、记录数量、存储位置等信息。

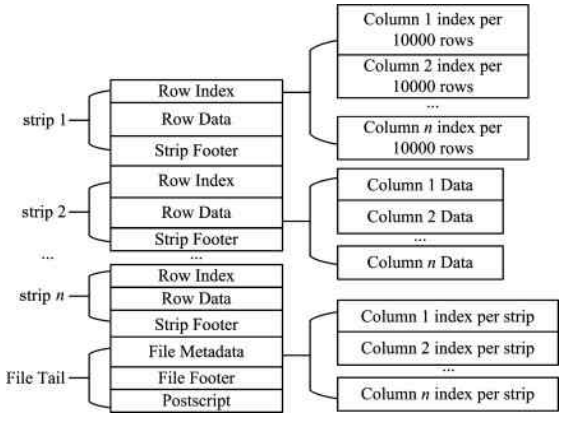


图 2 ORC 文件格式

Fig. 2 ORC file format

3.2 基于 ORC 文件的一般 Hive 查询

Hive 查询 ORC 文件的流程简述如下。

1)制定 query plan。Hive 接收到用户提交的 SQL 查询后，制定一个由 MapReduce 作业组成的有向无环图(DAG)，并为 DAG 中需要读取 ORC 文件的作业依次执行如下操作：访问 ORC 文件的 File Footer 和 File metadata，以分别跳过与查询不相关的文件和 strip；将剩余的每一个 strip 作为作业的一个输入分块；估算作业的总输出数据量并除以单个 reducer 可以处理的数据量(默认为 1GB)，从而得出需要启动的 reducer 的数量。在图 1 中，Hive 查询了某表格的第 1 列和第 3 列，过滤掉了不相关的 strip，并计划启动 5 个 reducer。

2)执行 map。MapReduce 引擎根据输入分块的分布来启动 mapper，其中一个分块(即 strip)启动一个 mapper。当分块很多时，mapper 之间可能需要共用一个物理节点。Mapper 读取 strip 时，访问其 row index，通过比较 row index 每个条目的最大值、最小值与查询条件的匹配情况，过滤掉不符合查询条件的行。由于 row index 每个条目对应 10000 行记录的统计信息，过滤的最小粒度也为 10000 行。mapper 的输出被称为中间键值对。在图 1 中，中间键值对的 key 为第 1 列，value 为第 3 列。

3)划分 key。MapReduce 引擎根据中间键值对的 key 来求取键值对所属的分区：

分区编号＝key.hashCode() % reducer 数量

每个分区启动 1 个 reducer，不同的 reducer 可以共用一个物理节点。

4)执行 shuffle。当 5%的 mapper 执行完成后，reducer 开始执行 shuffle：从每个 mapper 的输出中拉取归自己负责的分区的中间键值对。

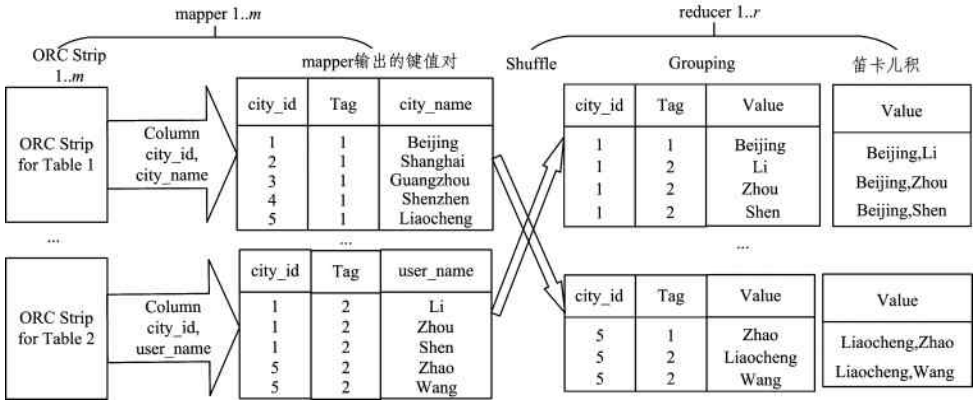
5)执行 reduce。reducer 拉取完自己的中间键值对后执行具体的 reduce 操作。

3.3 基于 ORC 文件的 Hive join 查询

如图 3 所示，join 查询的特殊性体现在：

1)map 阶段。所有表格的数据被统一读取，以 join 字段值为 key，以 select 字段值为 value，并打上来源表格的标签，生成中间键值对。

2）reduce 阶段。首先，key 相同的中间键值对的 value 被划分到同一组，同组内来自同一个表（即标签相同）的 value 被继续划分到同一个子组（Grouping）；然后，计算同组内子组间的 value 的笛卡尔积，从而得出 join 的结果。



注：SELECT city_name，user_name FROM table_1 JOIN table_2 ON（table_1. city_id＝table_2. city_id）

图 3　基于 ORC 文件的 Hive join 查询示例

Fig. 3　Sample of Hive join querying based on ORC files

4　问题描述

本节建立 Hive join 查询的 reducer 代价模型，并据此提出使 reducer 负载均衡的最优化目标。

4.1　reducer 代价模型

以集合 $\mathcal{T}=\{t_1,t_2,\cdots\}$ 表示参与 join 的表格，集合 $\mathcal{K}=\{k_1,k_2,\cdots\}$ 表示中间键值对 key 的所有取值，集合 $\mathcal{R}=\{r_1,r_2,\cdots\}$ 表示各 reducer 输入的数据量。

定理 1　Hive join 查询的 reducer 的时间复杂度和空间复杂度均为 $O(r)$，其中 r 为该 reducer 的输入数据量。

证明：以 g_i^j 表示某 reducer n 上 key i 表 j 对应的中间键值对 value 子组，显然：

1）reducer n grouping 的时间复杂度和空间复杂度均为 $O(r_n)$。其中，$r_n=\sum_{i=1}^{|\mathcal{K}|}\sum_{j=1}^{|\mathcal{T}|}|g_i^j|$。

2）reducer n 笛卡尔积计算输入的数据量为 r_n，而笛卡尔积的计算过程至少需要对全部输入数据遍历 1 次，因此笛卡尔积计算的时间复杂度和空间复杂度均为 $O(\max\{r_n,\Delta_n\})$。其中，$\Delta_n(\Delta_n=\sum_{i=1}^{|\mathcal{K}|}\sum_{j=1}^{|\mathcal{T}|}|g_i^j|)$，为笛卡尔积计算出的 join 结果数量。由于笛卡尔积计算通常存在对计算量的限制（对于 $\forall_i$，最多只允许存在一个 j 使得 $|g_i^j|>1$），因此有 $r_n>\Delta_n$，即笛卡尔积计算的时间复杂度和空间复杂度均为 $O(r_n)$。

由于 Grouping 和笛卡尔积计算在 reducer 上是串行执行的，因此 reducer n 的时间复杂度和空间复杂度均为 $O(r_n)$。

4.2　最优化目标

reducer 的负载均衡等价于最小化负载最重的 reducer。因此负载均衡的最优化目标为：调整 key 划分方案以实现负载最重 reducer 负载的最小化。

根据定理 1，可以直接用 r_n 表示 reducer 的负载。因此，上述目标可以形式化地表示为：

$$\mathrm{Min}_{key\text{划分}}\{\mathrm{Max}_{i=1}^{|\mathcal{R}|}r_i\}\geqslant\left[AVGLOAD=\frac{\sum_{i}^{|\mathcal{R}|}r_i}{|\mathcal{R}|}\right]\tag{1}$$

其中，$AVGLOAD$ 为负载均值，是负载均衡时负载最重 reducer 的负载可能达到的最小值。

5　方法设计

根据式（1），为实现 reducer 负载均衡，首先需要计算出各 reducer 输入的数据量。假设已知：

1）join 查询的全部中间键值对的 key 分布，以矩阵表示为：

$$\boldsymbol{F}=\{f_{ij}\},1\leqslant i\leqslant|\mathcal{T}|,1\leqslant j\leqslant|\mathcal{K}|$$

其中，f_{ij} 表示来自表格 i 的 key j 的数量。

2）中间键值对的 key 划分以矩阵表示为：

$$\boldsymbol{P}=\{p_{ij}\},p_{ij}=\begin{cases}1, & \text{若 key } i \text{ 分配给 reduer } j\\ 0, & \text{否则}\end{cases}$$

则：

1）各 reducer 来自各表格的输入数据量可以由下式求得：

$$\boldsymbol{R}=\boldsymbol{FP}=\{r_{ij}\}$$

其中，r_{ij} 为 reducer j 中来自表格 i 的输入数据量。

2）某 reducer j 的总输入数据量 r_j 可以由下式求得：

$$r_j=\sum_{i=1}^{|\mathcal{T}|}r_{ij}$$

因此，为实现 reducer 负载均衡，需要先估算出全部中间键值对的 key 分布，然后再求取使 reducer 负载均衡的 key 划分。

5.1　key 分布的估算

ORC 元数据中统计粒度最小的是 row index，row index 的每个条目对应 10000 行记录的最小值、最大值等统计信息。因此不能通过 ORC 元数据获取每个 key 的具体频率，但可以获得 row index 中的每个条目的 key 取值区间、记录数量等统计信息，并基于这些统计信息估计 key 的区间频率分布直方图。

对于 key 全局有序的 ORC 文件，其 row index 所有条目的 key 区间均不存在交叉，因此从 row index 中抽取各条目的 key 区间、记录数量来构成 key 频率分布直方图。而对于 key

非全局有序的 ORC 文件，则需要从交错的 row index 条目 key 区间中估算出不重叠区间的频率。

估算时，首先将 key 的整个取值范围划分为等间距的桶(bucketing)。然后遍历全部 row index 条目，将每个被遍历的 row index 条目的记录数量按该条目的 key 区间与各个桶的 key 区间的重合比例分配给各个桶(即假设了每个条目的 key 区间内部为均匀分布)。遍历结束生成的桶列表即为所求的 key 频率分布直方图，如图 4 所示。如果 join 查询有过滤条件，还需要遍历桶列表以过滤掉不符合查询条件的 bucket。

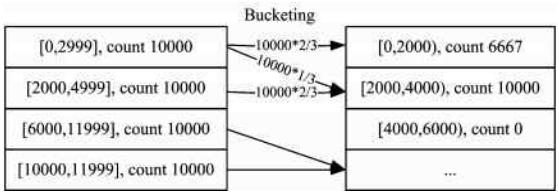


图 4 row index 中条目的 bucketing 过程

Fig. 4 Bucketing process of row index items

显然，估算的误差与 row index 条目的 key 区间的交错程度成正比，假如每个 row index 条目的 key 区间都覆盖了 key 的整个取值范围，则估算出来的直方图将退化为均匀分布，失去了参考价值。针对这种估算误差很大的情况，终止最优化求解，退而采用默认的 hash 划分。

5.2 负载均衡的 key 划分算法

由于获得的 key 频率分布直方图以 bucket 为最小单位，因此划分算法也以 bucket 为最小单位。采用存储和检索开销都很小的区间划分对全部 bucket 进行组合，以形成最终的 key 划分。

key 划分的计算过程如图 5 所示。首先，以桶的最大 key 值的升序为顺序，将所有 join 字段的桶列表放在一起排序，得到 ranked_buckets 桶列表；其次，以每个桶的最大 key 值处为分割点，进一步将 ranked_buckets 桶列表分割并组合为一个不存在交错区间的桶列表 clipped_buckets(见算法 1)；最后，以负载均值 *AVGLOAD* 为目标，将相邻的 clipped_buckets 组合为记录数量为 *AVGLOAD* 的区间(见算法 2)。

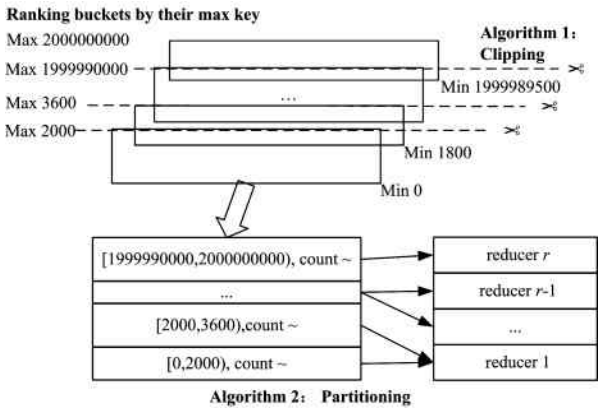


图 5 基于桶列表的 key 划分过程

Fig. 5 Key partitioning process based on bucket lists

算法 1 Buckets clipping

```
Input:ranked_buckets
Output:clipped_buckets
clipped_buckets={}
for i in ranked_buckets.length-1..0 do
  clipped_part_min=bucket_{i-1}^max, clipped_part_max=bucket_i^max
  if clipped_part_min<=bucket_i^min then
      clipped_part_count=bucket_i^count, ranked_buckets.pop(i)
  else if bucket_i^max==bucket_{i-1}^max then
      continue
  else
      clipped_part_count= bucket_i^density * (bucket_i^max-bucket_{i-1}^max)
      bucket_i^count-=clipped_part_count
      bucket_i^max=clipped_part_min
  end if
  for j in ranked_buckets.length-1..i+1 do
      if clipped_part_min<=bucket_j^min then
      clipped_part_count+=bucket_j^count
      ranked_buckets.pop(j)
      continue
  end if
      clipped_load=bucket_j^density * (bucket_j^max-clipped_part_min)
      clipped_part_count+=clipped_load
      bucket_j^count-=clipped_load
      bucket_j^max=clipped_part_min
  end for
  clipped_buckets.append(clipped_part)
end for
return clipped_buckets
```

算法 2 Partitioning

```
Input:clipped_buckets,AVGLOAD
Output:partition_lst //区间划分的分割点
partition_lst={}
partition_load=0
for i in clipped_buckets.length-1..0 do
  partition_load +=bucket_i^count
  remain=partition_load-AVGLOAD
  if remain<0 then
    continue
  end if
  bucket_i^min=bucket_i^max-remain/(bucket_i^max-bucket_i^min)
  bucket_i^count=remain
  partition_lst.append(bucket_i^min)
  partition_load=bucket_i^count
end for
return partition_lst
```

6 验证

验证过程中的 Hadoop 集群由 12 个物理节点组成，每个

节点有 12 个物理 CPU 核心，64GB 内存，所有节点连接在同一个万兆交换机上，Hadoop 版本为 2.6(CDH 5.8)。

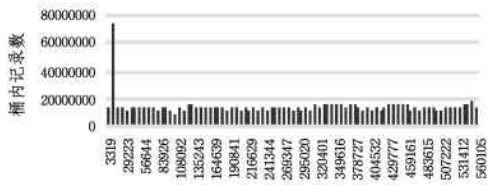
测试前对 Hive 进行如下配置：1)固定 reducer 的数量为 11，以消除 reducer 数量的变动对 Hive 查询的执行效率带来的影响；2)将 Hadoop 自带的区间划分器 TotalOrderPartitioner 作为 join 查询的 key 划分器。

测试工具为大数据时代具有代表性的 SQL 基准测试套件 TPCx-BB。测试数据集大小为 1TB。从 TPCx-BB 的测试负载中选择了两个典型查询 q29 和 q12 的最主要的 join 查询部分作为查询语句。具体查询语句如下：

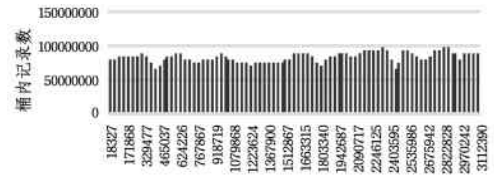
Q29：CREATE TEMPORARY TABLE test_tmp_q29 as SELECT ws_item_sk FROM web_sales ws，item i WHERE ws. ws_item_sk=i. i_item_sk；

Q12：CREATE TEMPORARY TABLE test_tmp_q12 as SELECT wcs _ item _ sk FROM web _ clickstreams，item WHERE wcs_item_sk=i_item_sk。

在本实验所采用的测试数据中，join 字段 *ws_item_sk* 的值被全局有序化地存储在 ORC 文件中，而 join 字段 *wcs_item_sk* 的值被非全局有序化地存储在ORC文件中。如图6所示，*ws_item_sk* 值的频率分布倾斜严重，而 *wcs_item_sk* 值的频率分布较为均匀。



(a)*ws_item_sk* 的频率分布直方图



(b)*wcs_item_sk* 的频率分布直方图

图 6　在测试数据中 join 字段的 key 频率分布直方图

Fig. 6　Join key distribution histograms of benchmark datasets

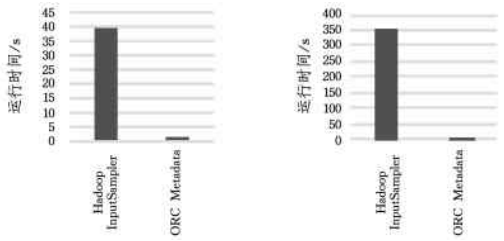
测试项目有：key 分布的估算效率、key 划分对 join 查询性能的影响、key 划分的负载均衡程度。

6.1　Key 分布的估算效率

分别为 q29 和 q12 执行如下测试过程：1)预抽取参与 join 的字段的 ORC 元数据的 row index 部分至同一个物理节点上。由于仅当对应的 Hive 表格更新时 ORC 元数据才会更新，这一步可以视为预处理，所耗时间不计入 key 分布的估算。2)在上述节点上执行单机 bucketing 算法以生成 key 的频率分布直方图(见 5.1 节)。其中 bucket 的数量设置为 11000，即 reducer 数量的 1000 倍，这是一个经验倍数。bucket 的数量过大会导致 bucketing 过程缓慢，过小则会导致下一步的 key 划分的粒度过粗。3)根据查询条件对上一步生产的桶列表执行过滤操作。

作为对比，我们测试了目前最常用的区间划分工具 InputSampler. RandomSampler 来估算 key 分布的效率。由于 Hive 不能直接调用该采样接口，我们实现了一个模拟 Hive join 主要机制的 MapReduce 程序，并调用了上述工具，设置目标样本数与 bucket 的数量相等(即 11000)。

测试结果如图 7 所示。从图 7 中可以看出，基于 ORC 元数据的 key 分布估算方法的时间开销很小，几乎可以忽略不计。作为对比，Hadoop InputSampler 估算 key 分布的时间开销很大。由此证明基于 ORC 元数据的 key 分布估算方法的效率远远超过了目前的主流方案。



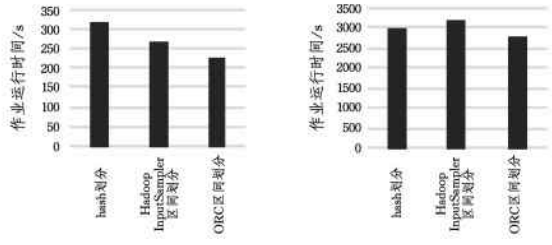
(a)TPCx-BB q29　　(b)TPCx-BB q12

图 7　key 分布的估算效率的对比

Fig. 7　Comparison of estimation efficiency for the key distribution

6.2　key 划分对 Hive join 查询性能的影响

如图 8(a)所示，与默认的 hash 划分相比，本文提出的基于 ORC 文件元数据的区间划分对 key 分布倾斜严重的 join 查询的执行效率有显著的改进，查询的执行时间减少了 30%，而目前最为常用的区间划分工具 HadoopInputSampler 仅有 16%左右的改进。如图 8(b)所示，新方法对 key 分布较为均匀的 join 查询的性能改进微小，而 HadoopInputSampler 甚至拖累了查询的执行。



(a)TPCx-BB q29　　(b)TPCx-BB q12

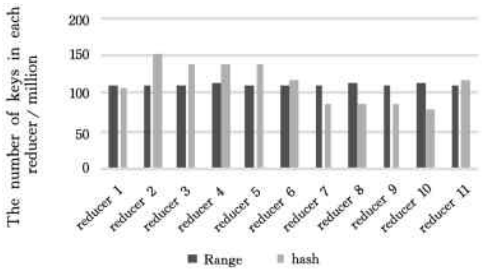
图 8　不同 key 划分方法的 join 查询效率的对比

Fig. 8　Comparison of join querie efficiency under different key partition methods

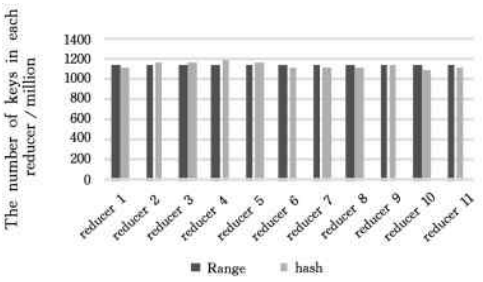
6.3　key 划分的负载均衡程度

测试得到的负载分配直方图如图 9 所示。从图 9 中可以看出：1)图 9(a)采用了基于 ORC 元数据构造的区间划分后，

各 reducer 的负载分配较为一致，其中在 hash 划分下负载最重的 reducer 的负载有了显著的下降。这就是图 8(a)基于 ORC 元数据的区间划分比 hash 划分查询执行快得多的原因。2)图 9(b)采用 hash 划分时 reducer 的负载分配已经比较均衡了，因此采用区间划分并没有将负载最重的 reducer 的负载减少很多。这就是图 8(b)中 hash 划分和区间划分作业执行时间相近的原因。



(a)TPCx-BB q29



(b)TPCx-BB q12

注：Range 表示基于 ORC 元数据的 key 划分，hash 表示 Hadoop 默认的 key 划分

图 9 不同 key 划分方法的 reducer 负载均衡的对比

Fig. 9 Comparison of load balance on reducers under different key partition methods

结束语 Hive join 查询非常容易触发 reducer 负载不均衡。通用的解决方案是基于中间键值对的 key 分布设计 reducer负载均衡的 key 划分算法。现有 key 分布估算的开销较大，延后了 shuffle 的启动，从而潜在延长了作业的执行时间。本文针对 Hive join 查询，提出了基于 ORC 元数据的 key 分布估算方法及相应的负载均衡 key 划分算法。新的 key 分布估算方法开销较小，不会拖延 shuffle 的执行。

本文在由 12 个物理节点组成的 Hadoop 2.6 集群上对新方法进行了测试。结果表明：新方法估算 key 分布的效率大幅领先于现在的主流方案 HadoopInputSampler，而构造出来的 key 划分也较好地实现了 reducer 的负载均衡，即将一个 key 分布严重倾斜的 join 查询的执行时间减少了 30%。

新方法存在一个明显的缺点：若 join 字段的 ORC row index 条目的 key 区间交错严重，则估算出来的 key 分布误差很大，基本不能从中得出负载均衡的 key 划分。因此新方法不适用于这类 ORC 文件的 join 查询。

未来，我们将把基于 ORC 元数据的负载均衡方法应用到 map 阶段。此外，我们还将针对异构集群的各个节点处理能力不同的情况，把 mapper 和 reducer 在节点上的分配情况纳入负载均衡的调节因素中。

参 考 文 献

[1] KWON Y C,BALAZINSKA M,HOWE B,et al. Skew-resistant parallel processing of feature-extracting scientific user-defined functions[C]// ACM Symposium on Cloud Computing (SoCC). ACM,2010:75-86.

[2] KE Q,PRABHAKARAN V,XIE Y,et al. Optimizing Data Partitioning for Data-Parallel Computing[C]// Proceedings of the 13th USENIX conference on Hot topics in operating systems (HotOS). USENIX,2011:13.

[3] YAN W,XUE Y,MALIN B. Scalable and robust key group size estimation for reducer load balancing in MapReduce[C]// IEEE International Conference on Big Data. IEEE,2013:156-162.

[4] IBRAHIM S,JIN H,LU L,et al. LEEN:Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud[C]// IEEE Second International Conference on Cloud Computing Technology and Science. IEEE,2010:17-24.

[5] GUFLER B,AUGSTEN N,REISER A,et al. Load Balancing in MapReduce Based on Scalable Cardinality Estimates[C]// International Conference on Data Engineerin (ICDE). IEEE Computer Society,2012:522-533.

[6] CHEN Q,YAO J,XIAO Z H. LIBRA:Lightweight Data Skew Mitigation in MapReduce[J]. IEEE Transactions on Parallel & Distributed Systems,2015,26(9):2520-2533.

[7] WANG Z,CHEN Q,LI Z H,et al,An Incremental Partitioning Strategy for Data Balance on MapReduce[J]. Chinese Journal of Computers,2016,39(1):19-35. (in Chinese)
王卓，陈群，李战怀，等. 基于增量式分区策略的 MapReduce 数据均衡方法[J]. 计算机学报，2016,39(1):19-35.

[8] KWON Y,BALAZINSKA M,HOWE B,et al. SkewTune:mitigating skew in mapreduce applications[C]// ACM SIGMOD International Conference on Management of Data. ACM,2012:25-36.