文章编号: 1000-5641(2018)05-0056-11

分布式数据库系统中的并行分组聚合实现

徐石磊, 魏 星, 江 红, 钱卫宁, 周傲英

(华东师范大学 计算机科学与软件工程学院, 上海 200062)

摘要: 伴随着新型互联网应用中对数据统计、分析需求的增大, 分组、聚合已经成为数据分析应用中出现频率最多的请求之一. 本文就类 OLAP(on-line transaction processing) 应用中常见的 Aggregation、GroupBy 原理进行了分析. 针对一般事务型数据库采用排序分组的缺点, 提出了两种 Hash 分组聚合的具体实现方案, 并提出一种利用统计信息动态决策 Hash 桶数、Hash 分组聚合方案的策略. 根据分布式数据库多副本的特点, 本文又提出了一种 Hash 分组聚合节点级的并行方案. 最后, 在开源数据库 OceanBase 进行了具体的实现. 通过实验证明, 本文提出的利用统计信息动态决策 Hash 分组聚合方案相比排序分组具有极大的效率提升.

关键词: OceanBase; GroupBy; Hash; 数据分布

中图分类号: TP392 **文献标志码**: A **DOI:** 10.3969/j.issn.1000-5641.2018.05.005

Implementation of the parallel GroupBy and Aggregation functions in a distributed database system

XU Shi-lei, WEI Xing, JIANG Hong, QIAN Wei-ning, ZHOU Ao-ying

(*School of Computer Science and Software Engineering, East China Normal University, Shanghai* 200062, *China*)

Abstract: With the increase in demand for data statistics and analysis in new Internet applications, data grouping and aggregation have become amongst the most common operations in data analysis applications. This paper analyzes the operating principles of the Aggregation and GroupBy functions commonly used in analytical applications. Based on the disadvantages of sort grouping for general-transactional databases, two kinds of Hash GroupBy implementations are proposed; in addition,a strategy for dynamically determining the number of Hash buckets and Hash GroupBy schemes, based on statistical information, is proposed. Based on the characteristics of distributed clusters, implementation of the Hash GroupBy operator push down is proposed. Experiments have shown that the use of statistical information to dynamically determine the Hash group option improves efficiency.

Keywords: OceanBase; GroupBy; Hash; Data distribution

收稿日期: 2018-07-04

基金项目: 上海市青年科技英才扬帆计划(17YF1427800)

第一作者: 徐石磊, 男, 硕士研究生, 研究方向为数据存储与数据挖掘. E-mail: xsl118857@sina.com.

通信作者: 江 红, 女, 副教授, 研究方向为现代信息系统及其开发技术.
E-mail: hjiang@cc.ecnu.edu.cn.

0 引　　言

随着企业管理系统管理和维护的数据量越来越大, 大多数企业往往会陷入早期配置的存储系统在使用一段时间后达到瓶颈的困境. 最初, 大家都是通过给系统替换内存更大、CPU 更多的设备来解决这类问题, 但是这种替换整个系统的解决方案耗费的代价十分高昂. 后来, 伴随着网络通信技术和分布式存储理论的成熟、发展, 分布式数据库逐渐登上了时代的舞台. 相比传统集中式数据库, 分布式数据库具有非常友好的扩展性. 因此, 为了安全、可靠、低成本地存储和管理这些海量的数据, 我们逐步选择将这些应用转移到分布式数据库中去.

在商业应用和政府工作中, 经常需要根据某些纬度做信息统计和分析. 比如, 公司需要统计各个产品的销售量、营业额、利润; 企业需要统计各地区的营业额、利润情况; 基金公司需要了解各个基金的收益情况; 政府部门需要统计各个地区的男女比例、各个年龄段的劳动力情况、每年的新生儿出生率情况. 这些需求都导致我们不可避免的需要调用分组和聚合等操作来实现. 分组和聚合已经成为数据分析应用中出现频率最多的操作之一. 因此, 如何提高分组和聚合的工作效率已经成为我们做统计分析的重中之重. 本文研究的重点便是如何在分布式数据库中提高分组和聚合的运算效率.

论文的内容组织如下: 第 1 节简要介绍了分布式数据库的基本结构以及目前分组和聚合的研究现状. 第 2 节详细介绍了本文提出的两种通过 Hash 算法进行分组聚合的方案, 以及一种根据统计信息动态决策 Hash 桶数目和这两种 Hash 分组聚合方案的策略. 第 3 节主要介绍了如何利用分布式数据库特点实现 Hash 分组聚合节点级的并行方案. 第 4 节从数据分布、是否并行、并行节点个数等方面对 Hash 分组聚合效率进行了全面的实验验证. 第 5 节对本文进行了总结.

1 相关工作

1.1 背景介绍

随着网络通信技术分布式存储理论的成熟和发展, 数据库的发展逐渐进入了分布式的阶段. 相比传统集中式数据库, 分布式数据库可以通过动态增加存储节点来实现存储容量的横向扩展. 并且, 分布式数据库为了保证数据的安全和可靠存储, 往往会采用备份的策略. 数据多副本存储不仅能够提供可靠、安全的存储, 还可以提高系统查询的并发量. 由于分布式数据支持横向扩展和多副本安全存储的特性, 越来越多的企业开始采用分布式数据库作为自己的数据存储分析平台. 分布式数据库一般分为物理数据存储节点和数据查询计算节点. 以 OceanBase[1]分布式数据库为例, 数据存储在 ChunkServer 服务器节点(以下简称 CS). 为了提高数据存储的可靠性, 一般设置多副本, 分别存储在不同 CS 节点上. 查询一般交由 MergeServer(以下简称为 MS)处理. 其他两类节点, UpdateServer(以下简称 UPS)和 RootServer(以下简称 RS)则分别负责系统的增量数据处理和集群中所有服务器节点的管理.

针对 GroupBy 功能的实现, 大部分关系型数据库采用的是一种排序分组的方式. 数据经过排序之后, 相同数据值的元组会排列在一起, 不同数据值只在各自边界时出现. 但是, 单纯对于分组来说, 利用排序进行数据分组是事倍功半的操作, 将会导致数据分组以及分组之后的聚合操作效率低下. 因此, 针对这一缺点, 我们提出通过 Hash 算法的方式去实现分组: Hash 算法可以将属性数据映射到一个整数类型区间; 然后通过取模运算将整型值分类到不同桶中, 这样相同数据值就会映射到同一桶内, 即实现了基本的分组功能. 因此, 使用 Hash

算法去设计 GroupBy 算子, 将极大地提高数据库做分组和聚合的效率.

1.2 相关研究

许多研究人员为了提高数据库的查询处理能力，从查询涉及的众多方面进行了深入的研究. Aggregation、GroupBy 作为查询语句中出现频率很多, 且耗费系统资源和占用查询时间较多的功能也吸引了很多学者的关注. 众多学者从各个角度对 Aggregation、GroupBy 提出了优化方案.

Ho[2]提出了 SUM 和 MAX 这两种聚合操作的快速算法. Roussopoulos 和 Gupta 等[3-4]认为通过维护恰当的物化视图可以加速 Aggregation、GroupBy 算子的运行. 但是, 建立一个物化视图对于系统来说, 需要占用大量内存资源. 并且对于需要频繁写入、修改的系统来说, 维护视图与数据库的数据一致性会拖慢写入和更新的效率. Govindarajan 等[5]提出一个 CRB-Tree 索引结构, Feng Y 等[6]提出 Ag+-Tree 索引结构, Sellis 等[7]提出了 BR+-Tree, 他们认为建立和维护一个高效的索引结构可以提高分组聚合查询的效率. 建立索引相比物化视图, 虽然可以减少内存消耗, 但是与物化视图具有相同的缺陷: 当系统是一个写密集型的系统时, 维护索引将拖慢写入和更新的效率. 因此针对写密集应用, 通过建立索引和物化视图等方法加速分组聚集具有一定的局限性.

Tao 等[8]分析传统分组聚合问题的特点, 将一个集合运算中出现同一个对象定义为 RASS 问题, 提供有效解决该问题的方案. Condie 等[9]在 Hadoop MapReduce[10]框架基础上修改并提出了 HOP, 使得 Aggregation 函数不仅可以由多任务协同处理, 而且还可以看到已经完成任务返回的数据结果, 极大地提高 Aggregation 的查询响应速度. 以上做法, 都是离开分组和聚合的具体实现来考虑问题. 无论是从索引和视图等角度优化, 还是通过并行优化, 都是在分组和聚合操作的上层进行考虑问题. 建立视图, 搭建 MapReduce 框架进行并行计算等优化方法的底层还是需要调用分组算子. 在这里, 我们从分组和聚合算子的具体实现角度提出新的优化方案, 并且本文实现的 hash 分组聚合方案完全兼容索引和并行等优化手段.

2 Hash GroupBy 的工作原理

Hash GroupBy 使用 Hash 算法进行分组. Hash 函数首先计算数据 Hash 值, 然后对计算后的 Hash 值进行分区. 比如: GroupBy c1, Hash 函数首先计算 c1 列数据的 Hash key, 然后将 Hash key 进行分区. 由于调用的 Hash 函数相同, 因此相同数据计算出的 Hash key 值相同, 分区也相同. 但是也可能存在不同数据计算出来的 Hash key 值相同情况, 我们在设计时需要特别考虑这类情况. 此外, 针对 Hash 冲突情况, 这里我们采用开链法处理冲突.

2.1 方案一桶内排序分组原理

如果数据集不同值个数很多, 即 GroupBy 得到的结果集很多. 此种情况下, 我们建议使用桶内排序. 先计算所有数据 Hash key 并插入桶中, 随后对每个桶内数据进行快速排序. 由于可能出现不同 row 值计算出来的 key 相同从而导致分到的桶相同的情况. 我们需要在保存 Hash key 的同时保存该列的值. 具体伪代码见算法 1.

(1) 首先计算 row 的 Hash key, 构建 <key, row*> 键值对, 并插入 Hash table 中.

(2) 重复执行 1, 直到所有 row 的 Hash key 都插入到 Hash table 中. 在此过程中, 使用一个数组 vec 保存所有不重复的 Hash key.

算法 1 桶内排序分组

输入: 统计信息和初始数据表
输出: 分组后的数据表

```
1:  bucket_num = statis_info; //根据统计信息确认Hash桶数
2:  Hash_table.create(bucket_num); //构建Hash桶
3:  while true
4:    取出一个row, 并计算row对应GroupBy列的Hash_key;
5:    res = Hash_table.set(Hash_key, &row);
6:    if res不为0 then //意味着Hash_key不在Hash表内
7:      vec.push(Hash_key); //将Hash_key存入vec数组中
8:    end if
9:  end while
10: while true
11:   if vec数组不为空then
12:     取出数组中Hash_key;
13:     清空临时数组tmp_sort_array中所有元素;
14:     while true
15:       根据Hash_key取出Hash表中对应元素, 加入到tmp_sort_array中;
16:     end while
17:     全部取完后, 对临时数组排序;
18:   end if
20:   将临时数组tmp_sort_array中元素加到全局sort_array数组中;
21: end while
```

(3) 从 vec 中取出一个 Hash key, 得到该 Hash key 对应桶中所有的 row*, 放到一个临时数组中排序. 将临时数组中排序后的 row* 都加入一个全局数组中后, 释放临时数组. 重复执行直到所有 Hash key 对应的桶数据都加入到全局数组中.

(4) 此时, 全局数组中相同数据皆相邻存储, 但全局并不有序. 然后, 根据聚合函数相邻相同数据执行聚合操作.

虽然与 MergeSort GroupBy 算子相似, 都是通过排序进行分组, 但是这里只是执行桶内排序. 与 MergeSort GroupBy 相比, 排序的量级从整个表数据减小为一个桶内数据. 因此, 该方案效率与桶内元组个数相关, 我们可以通过动态设置 Hash 桶数目调整同一桶内元组数目, 详细见 2.3. 但该方案桶内会存储重复元素, 且重复元素导致的同一桶内元素数目膨胀无法通过动态设置 Hash 桶解决. 因此, 该方案实际效率受数据分布影响. 如果数据相同值很多, 则同一桶内元素个数会很多, 会增加排序时间消耗, 因此该方案不适合处理相同值较多的情况. 这种做法适合于数据分布均匀的情况.

2.2 方案二频数分组聚合原理

如果数据集相同值个数较多, 即 GroupBy 得到的结果集不多. 此种情况下, 我们提出方案二, 对每一条数据 row, 查询 Hash table, 如果 Hash table 内已经有相同元素则记录频数. 我们在桶内只保存同一分组内的一条记录. 这样可以极大地减少 Hash 表的内存大小. 一个桶内的数据元素个数与聚集后的结果集个数相同. 具体伪代码见算法 2.

(1) 对每一条 row, 计算 row 的 Hash key 值, 然后确定该 row 对应的 bucket_pos.

(2) 如果该 bucket_pos 对应的桶为空, 则插入该元组, 并调用聚合聚集表达式计算该元组.

(3) 如果桶中非空, 则逐一搜索桶中元组并比较. 如果有相同元组, 不直接往链表中插入 row 值, 将该 row 对应频数加 1.

(4) 如果没有相同元组, 则插入该元组, 并调用聚合函数计算该元组.

(5) 重复 1-4, 直到所有元组都处理完. 输出值时须判断频数, 若为1, 则直接输出; 否则, 调用聚合函数计算后输出.

算法 2 频数分组聚合

输入: 统计信息和初始数据表
输出: 分组聚合后的数据表

```
1:  bucket_num = statis_info; //根据统计信息确认Hash桶数
2:  Hash_table.create(bucket_num); //构建Hash桶
3:  while true
4:    取出一个row, 并计算row对应GroupBy列的Hash_key;
5:    pair<&row, count>; //将<&row,int>组合成一个pair
6:    count = 1; //设置每个row的初始值为1
7:    res = Hash_table.set(Hash_key, &pair);
8:    if res不为0 then //意味着Hash_key不在Hash表内
9:      vec.push(Hash_key); //将Hash_key存入vec数组中
10:     直接调用聚合函数处理该row;
11:   else if res为0 then //意味着Hash_key在Hash表内已存在
12:     pair.count++; //计数加1
13:   end if
14: while数组不为空
15:   取出数组中Hash_key;
16:   根据Hash_key取出表内元素;
17:   if pair.count == 1 then //count为1, 则聚合结果值就是该row值
18:     result_row = row
19:   else //count不为1, 则聚合结果值为row*count
20:     result_row = row * count;
21:   end if
22: end while
```

该方案不需要进行任何排序, 但在处理每条 row 数据时都需要对 Hash table 中的所有数据进行一次遍历比较. 因此, 如果每一个 Hash key 值涉及到多个数据, 则这种逐一比较耗费时间较高. 但是该方案桶内不会有重复元组. 所以, 如果我们合理控制 Hash table 在初始化时 Hash 桶的数目, 便可以极大减小 Hash 冲突率从而降低这种逐一比较的代价. 因此, 我们在这里提出根据统计信息动态设置桶数, 由此控制每个桶内元素个数, 减少同一桶内元素个数, 详细见第 2.3 节.

考虑到计算节点上有限的内存, 当数据总量超过内存上限时如何初始化 Hash 桶便成为了一个难题. 针对这一限制条件, 本文提出一种方案: 首先对数据集调用 Hash 函数, 对数据进行分区操作, 分区操作可以将数据的量级减小. 然后对分区后的分区表构建 Hash 表. 如果分区之后的数据依然很大, 那么就进行递归分区, 直到数据适合内存大小. 这样, 我们对整个数据集的操作就变为对分区之后的分区表操作. 比如 group c1, 如果 c1 列数据非常多, 则根据 c1 将数据集 D 划分为 D1, D2, $\cdots$, 然后分别对 D1, D2, $\cdots$ 进行 Hash GroupBy 操作, 最后再汇聚即可.

2.3 动态决策

由于调用 Hash GroupBy 时, 桶数目和数据分布会影响最终的 Hash 分组聚合效率. 因此, 我们提出根据统计信息动态设定 Hash 桶数目以及动态决策最终采用的 Hash GroupBy 方案. 为此, 首先需要在系统中收集统计信息, 分别收集表、列的统计信息. 表的统计信息内主要包含表内数据的行数 count. 列的统计信息内包含各列不重复值个数 distinct_col, 该列中高频率出现的 10 个频数值 top[10].

2.3.1 Hash 桶数目的动态决策

实际分析应用时, 我们可能只需要根据一个属性做分析, 也可能根据多个属性信息进行分析. 因此, 我们根据分组列为一个属性或多个属性分类讨论. 当分组条件为单列时, 根据如下公式决定 bucket_num.

$$\mathrm{avg}_{\mathrm{num}} = \frac{\mathrm{count} - \Sigma_{i=1}^{10}\mathrm{top}[i]}{\mathrm{distinct\_col} - 10}, \tag{1}$$

$$\text{bucket\_num} = \begin{cases} \frac{\text{distinct\_col}}{N}, & \text{如果 avg}_{\text{num}} \leqslant N, \\ \text{distinct\_col}, & \text{若 avg}_{\text{num}} > N. \end{cases} \tag{2}$$

针对单列情况, 我们可以精细化地根据列中不同值个数进行 Hash 桶数目的选择, 使得 Hash 桶的数目大致等于该列不同值个数. 这就能保证 Hash table 中每个桶内仅存储对应值. 但是, 为了避免多数桶内只有一个元素的情况发生, 我们根据开发经验预估出 Hash table 中一个数据值的重复数目(avg_num). 为了避免个别高频值干扰, 我们先用桶内总行数 count 减去统计信息中记录的 10 个高频值 $\Sigma_{i=1}^{10}\text{top}[i]$, 然后除以不同值个数 distinct_col−10, 这样便求出每个值平均数目 avg_num. 并设置一个经验参数 N 与之对比, N 可以由实验得到. 例如, 设置 $N = 3$. 如果 avg_num 小于 3, 则我们认为根据不同值个数 distinct_col 创建桶可能导致大部分桶只有少量元素, 甚至只有一个. 所以我们设置 bucket_num=distinct_col/3. 反之, 如果大于 3, 我们认为每一个桶内元素很多, 则直接设置 bucket_num=distinct_col.

当分组条件为多列时, 由于列与列之间相互独立, 两列之间不同值个数也并无关系. 因此无法根据某列不同值个数决定桶数. 比如 GroupBy c1、c2 和 c3, 我们无法根据 distinct_col$_{\text{c1}}$、distinct_col$_{\text{c2}}$、distinct_col$_{\text{c3}}$ 得到最终的不同值数目 distinct_col. 因此, 无法像单列那样精细处理. 但我们可以根据表行数进行 bucket_num 的粗略估计. 同时, 我们需要考虑列的平均值个数 avg_num. 避免我们有 100万 数据, 只有十几个不同数据, 但我们设置 10万 个桶的极端情况. 以 GroupBy c1, c2, c3 为例, 我们分别求出 avg_num$_{\text{c1}}$、avg_num$_{\text{c2}}$ 和 avg_num$_{\text{c3}}$. 根据 c1、c2 分组求出的组内平均数肯定小于根据 c1 分组求出的组内平均数 avg_num$_{\text{c1}}$, 因为分组条件越多, 分组分的越细, 即 avg_numc_1、c_2、c_3 < min(avg_num$_{\text{c1}}$、avg_num$_{\text{c2}}$、avg_num$_{\text{c3}}$). 因此选择 min(avg_num$_{\text{c1}}$、avg_num$_{\text{c2}}$、avg_num$_{\text{c3}}$) 作为 avg_num. 设置经验常数 P 作为对比, P 可以由实验得到, 具体计算如公式 (3). 当 avg_num 很大时, 我们认为数据重复值太多, 因此我们需要用数据总行数除以一个值, 适当减少 buckt_num; 反之, 当 avg_num 很少时, 即每个数据重复值较少, 则我们可以直接使用 count 作为桶数.

$$\text{bucket\_num} = \begin{cases} \frac{\text{count}}{P}, & \text{如果 avg}_{\text{num}} > P, \\ \text{count}, & \text{若 avg}_{\text{num}} \leqslant P. \end{cases} \tag{3}$$

2.3.2　Hash GroupBy 方案的动态确定

由于我们提出的两种方案针对不同数据分布各有特点, 因此, 我们提出根据统计信息动态决策具体的 Hash GroupBy 方案. 与 2.3.1 节动态决定 Hash 桶数目相同, 根据单列和多列进行分别讨论.

首先, 我们分析单列情况, 与 2.3.1 中单列情况下 Hash 桶数目动态决策的方法相同. 根据公式 (1), 大致可以得到每个值平均数目 avg_num. 在实际实现过程中, 根据系统设计经验设置一个经验参数 M, 这个 M 具体值可以通过实验求出. 当 avg_num> M 时, 说明每个值都有大于 M 个重复值. 此时, 我们认为表中数据相同值较多, 选择方案二效率更好. 当 avg_num⩽ M 时, 说明每个值都只有少数相同值, 选择方案一桶内排序效率更好(M 参数的具体数值可以通过实验调参得到).

当分组条件为多列时, Hash 桶数目动态决策策略与 2.3.1 中提及的策略相同. 通常情况下, 数据表中列与列之间相互独立. 因此, 两列分别求出的 avg_num 并无关联. 通过公式(1)的计算, 我们可以分别得到每一列的 avg_num 值, 从而选择最小的 avg_num 进行粗略估算(选择 min(avg_num$_{\text{c1}}$、avg_num$_{\text{c2}}$、avg_num$_{\text{c3}}$)作为 avg_num). 多列执行方案的选择与单列相同, 将 avg_num 与 M 相比. 若大于 M, 则认为方案二效率更高; 小于等于 M, 则说明每个值都只有少数相同值, 选择方案一桶内排序效率更高.

3 分布式系统中 Hash GroupBy 的并行工作原理

分布式数据库集群一般部署多个数据存储节点, 每个数据节点负责管理和维护自己节点的数据并提供简单的读取和计算功能. 当分布式数据库存储一张大表时, 系统会将大表数据根据主键水平切分成为多个分片, 一个分片存储一部分数据. 为了保证系统的高可靠性, 分布式数据库还会设置多副本存储, 即将同一份数据备份存储在多个数据存储节点上.

考虑到既然各个存储节点可以完成简单的数据存取和计算功能, 且各个节点相互独立、互不影响. 那么我们设计 Hash GroupBy 算子时完全可以将算子下压到各个数据存储节点上从而实现节点间的并行执行. 在算子执行过程中, 各物理存储节点各自读取、计算自己节点的分片数据后, 分别调用 Hash GroupBy 对自己节点的分片数据进行分组聚合. 因此, 通过算子的分发执行便可以实现 Hash GroupBy 算子在不同节点间的并行执行.

以 OceanBase 分布式数据库为例, 假设集群有三个物理存储节点 CS1、CS2、CS3, 副本配置为 3. 由于数据表 R1 很大, 数据库会将 R1 切分成三个分片 R11、R12、R13. 同时复制 R11、R12、R13 进行副本备份, 并根据负载均衡算法将各个分片及副本分散存储在各 CS 节点上. 我们实现将 Hash GroupBy 算子下压到 CS 上的功能, 即会在 CS1、CS2、CS3 上分别生成一个 Hash GroupBy 算子.

假设有一个分组聚合 SQL: select count(col1) from R1 GroupBy col1. 理想情况下, 查询服务节点 MS 会将查询数据请求均匀分发给三个节点. 例如, 请求 CS1 有关 R11 分片数据, 请求 CS2 节点 R12 分片数据, 请求 CS3 节点 R13 分片数据. Hash GroupBy 算子下压到 CS 意味着, 每个 CS 数据存储节点在读取磁盘数据以及其他计算功能之后, 还可以执行 Hash GroupBy 算子, 即 CS1 上会调用 Hash GroupBy 对 R11 部分数据进行分组聚合, 在 CS2、CS3 上同样会调用 Hash GroupBy 对 R12、R13 部分数据进行分组聚合. 最终各节点将计算后的结果返回给 MS. MS 上同样会有一个 Hash GroupBy 算子用于对各 CS 最终返回的结果集进行分组聚合. 具体算子执行原理如图 1.

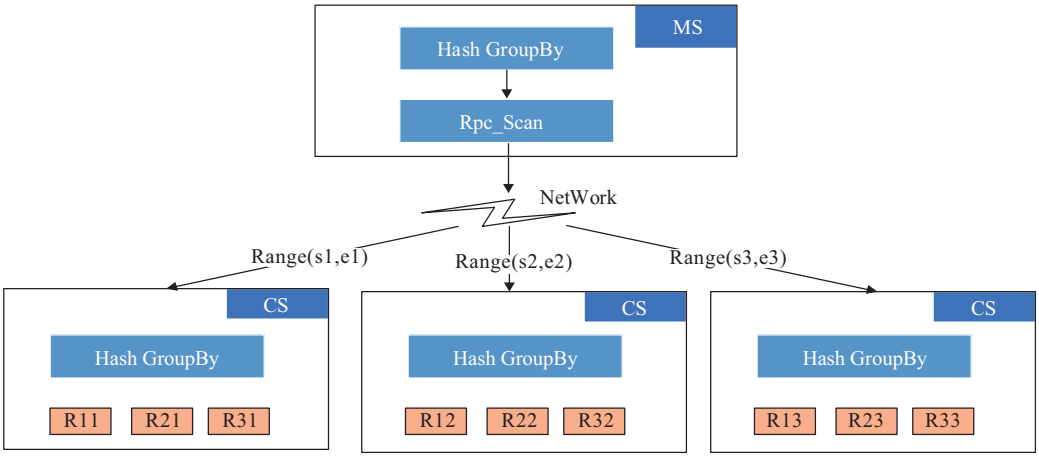

图 1 Hash GroupBy 下压执行计划

Fig. 1 Hash GroupBy push down execution plan

4 实验分析

首先测试随机数据分布情况下, Hash GroupBy 相比 MergeSort GroupBy 是否有效率提升. 接着, 测试不同数据分布和单节点非并行情况下, 两种不同 Hash GroupBy 方案相比 MergeSort

GroupBy 的效率提升. 最后, 测试特定数据分布、统计信息动态决策方案和三节点并行情况下, Hash GroupBy 方案相比 MergeSort GroupBy 的效率提升.

4.1 实验环境

本文选择 OceanBase 0.4 系统作为实验测试系统, 实验采用五台服务器部署数据库功能节点. 其中控制节点 RS 和事务节点 UPS 公用一台服务器, 查询节点 MS 使用一台服务器节点, 其余三台服务器部署数据存储节点 CS.

本文实验测试所使用的硬件环境如表 1 所示, 其中磁盘为 SSD(Solid State Drive).

表 1　集群服务器配置

Tab. 1　The cluster server configuration

角色	CPU	内存 / GB	磁盘 / TB	网络
CS/MS	20核40线程(Intel(R)Xeon(R)CPU E5-2620 V3@2.30GHz)*4	500	1	万兆网
RS/UPS	20核40线程(Intel(R)Xeon(R)CPU E5-2620 V3@2.30 GHz)*1	500	1.5	万兆网

4.2 实验数据集

本文测试所用到的数据表模式如表 2 所示.

表 2　测试表的 Schema

Tab. 2　The schema of the test table

属性	是否为主键	数据类型	数据大小 / Byte
K	是	Int32	4
C1	否	Int32	8
C2	否	Int64	8
C3	否	Double	8
C4	否	Float	8
C5	否	Decimal(15,5)	8
C6	否	Varchar(50)	50

由于不同数据类型占用系统内存大小、耗费网络传输时间不同, 特别是数据比较时间不同, 因此生成的数据包含多种数据类型. 为了尽可能控制变量, 本文测试了每种数据类型采用不同分组方法的效率差异, 我们的测试语句皆以单独列进行分组测试, 比如, 采用 select count(c1) GroupBy c1 测试 int32. 此外, 由于 Hash 函数存在冲突及 Hash 函数映射值空间有限等情况, Hash GroupBy 实现效率还与桶内数据个数有关. 因此, 我们设计了桶内不同数据个数分组聚合情况. 实验涉及的主要表、数据分布情况以及数据量如表 3 所示.

表 3　测试表的大小和分布

Tab. 3　Test data table information

表名	数据分布(桶内相同元素个数)	数据量(行数) / 万
G1	1	20
G2	3	60
G3	5	100
G4	10	100
G5	30	100
G6	50	100

以 G1 为例, 数据分布指的是 Hash 桶内的元素个数, 1 代表 Hash 桶内只有一个相同数据元素, 其他类比. 桶内元素个数, 我们可以通过改变 Hash 桶数调整, 但是相同数据元素必定排列在同一个桶内, 因此, 我们必须考虑此种情况. 除此之外, 我们还创建了 6 张表, 数据量分别为

1万、10万、50万、100万、500万、1 000万, 其数据类型与 G1 等相同, 但数据分布为随机生成, 毫无规律.

4.3 随机数据分布下方案一桶内排序的 Hash GroupBy 效率提升

实验目的: 测试方案一桶内排序 Hash GroupBy 相比 MergeSort GroupBy 是否有效率提升. 数据设置: Hash 桶数为 80万左右, 数据表大小分别是 1万、10万、50万、100万、500万、1 000万. 数据全都是随机生成, 毫无规律. 实验结果如图 2 所示.

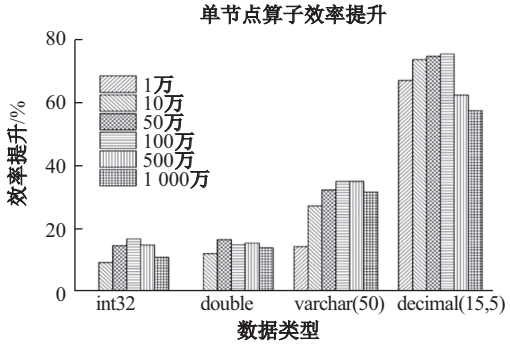

图 2 随机数据分布下方案一Hash GroupBy效率提升图

Fig. 2 The efficiency graph of the Hash GroupBy under random data distribution with option 1

从实验数据上看, 对于方案一来说, Hash GroupBy 相比 MergeSort GroupBy 而言, 是有效率提升的. 其中, int32 与 double 效率提升基本相同, 复杂数据类型比如 varchar(50)、decimal (15,5), 效率提升尤为明显.

4.4 数据分布对于不同 Hash 分组聚合方案的效率影响

由于不同数据分布对于分组效率影响较大, 因此我们针对方案一桶内排序和方案二频数聚合分组设计实验, 对不同数据分布情况进行了测试. 实验结果如图 3 和图 4.

从实验结果分析来看, 对于方案一桶内排序来说, 随着桶内相同数据元素不断增多, 桶内排序效率提升越来越小, 甚至当同一桶内数据元素为 50 时, 对于 int64、double、varchar(50)数据类型, 效率已经不如 MergeSort GroupBy. 对于方案二频数分组聚合恰好相反, 同一桶内元素个

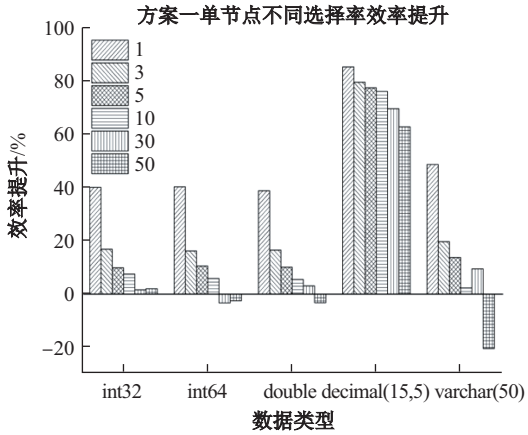

图 3 方案一效率提升

Fig. 3 The efficiency graph under option 1

数很小时, 效率提升非常细微. 但随着同一桶内数据元素越来越多, Hash 分组聚合相比 MergeSort GroupBy 效率提升越来越大.

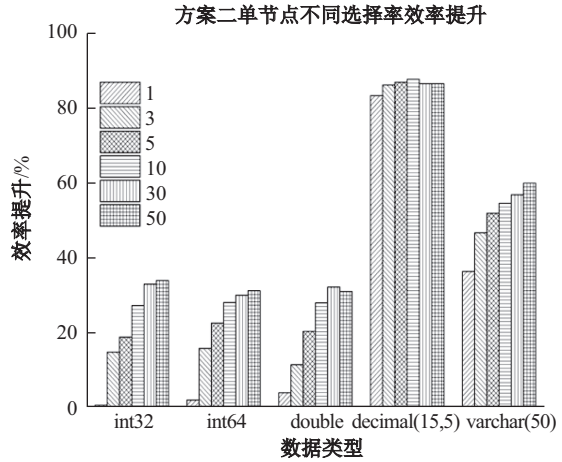

图 4　方案二效率提升

Fig. 4　The efficiency graph under option 2

4.5　Hash分组聚合节点级并行效率提升

由于分布式数据库多副本备份特点, 每个数据节点都可以提供查询和简单计算功能. 因此, 我们针对本文提出的 Hash 分组和聚合方案分别设计了 1 和 3 不同数据节点数进行实验, 具体实验结果如图 5.

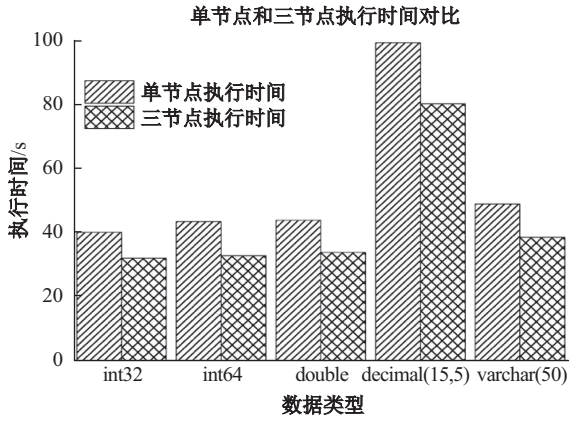

图 5　三节点不同数据分布效率提升图

Fig. 5　The efficiency graph of three nodes with various data distributions

从实验结果可以发现, 当我们存储节点设置为 3 时, 相比 1 个存储节点是有明显效率提升的, 针对各个数据类型, 效率提升都在 20% 之上. 并且我们在监控服务器 CPU 利用率上, 发现三个 CS 物理存储节点将数据读取和计算压力大致平均分散在了三个服务器节点上. 因此, 我们的节点并行方案在大负载、高并发下相比单节点会有更高的效率提升.

5　总　　结

本文提出了两种在分布式系统中采用 Hash 方法进行分组聚合的方案, 并且提出了一种利用统计信息对两种方案进行动态决策的策略. 最后, 针对分布式数据库多节点特性, 又提出一种 Hash GroupBy 算子节点级的并行方案. 在分布式数据库 OceanBase 上具体实现了这些方案. 实

验结果表明, Hash GroupBy 相比 MergeSort GroupBy 具有极大的效率提升; 并且针对不同数据分布情况, 两种 Hash GroupBy 的实现方案各有优点. 因此, 根据我们提出的利用统计信息进行动态选择方案的策略, 可以使我们的 Hash 分组和聚合具有更高的效率提升. 但是该方案还有进一步需要优化的空间. 目前各数据存储节点对本节点内数据分组和聚合后, 数据查询节点尚且需要汇总各数据节点返回数据再做一次总的分组和聚合, 这将是以后优化的重点.

[参 考 文 献]

[1] 杨传辉. 大规模分布式存储系统 [M]. 北京: 机械工业出版社, 2013.

[2] HO C T, AGRAWAL R, MEGIDDO N, et al. Range queries in OLAP data cubes [C]// ACM SIGMOD International Conference on Management of Data. ACM, 2008: 73-88.

[3] ROUSSOPOULOS N. Materialized views and data warehouses [C]// ACM SIGMOD International Conference on Management of Data. ACM, 1998: 21-26.

[4] GUPTA H, MUMICK I S. Selection of views to materialize in a data warehouse [J]. IEEE Transactions on Knowledge & Data Engineering, 1997, 17(1):24-43.

[5] GOVINDARAJAN S, AGARWAL P K, ARGE L. CRB-Tree: An efficient indexing scheme for range-aggregate queries [J]. Lecture Notes in Computer Science, 2003, 2572: 143-157.

[6] TAO Y, PAPADIAS D. Range aggregate processing in spatial databases [J]. IEEE Transactions on Knowledge & Data Engineering, 2004, 16(12): 1555-1570.

[7] SELLIS T. The R+-Tree: A dynamic index for multi-dimensional object [C]// Proceeding of the 13th VLDB Conf. VLDB, 1987: 507-518.

[8] TAO Y, SHENG C, CHUNG C W, et al. Range aggregation with set selection [J]. IEEE Transactions on Knowledge & Data Engineering, 2014, 26(5): 1240-1252.

[9] CONDIE T, CONWAY N, ALVARO P, et al. Online aggregation and continuous query support in MapReduce [C]// ACM SIGMOD International Conference on Management of Data. ACM, 2010: 1115-1118.

[10] DEAN J, GHEMAWAT S. MapReduce: simplified data processing on large clusters [J]. Communications of The ACM, 2008, 51(1): 107-113.

(责任编辑: 李万会)