海量数据上的近似连接聚集操作

韩希先[1)] 杨东华[2)] 李建中[1)]

[1)](哈尔滨工业大学计算机科学与技术学院 哈尔滨 150001)

[2)](哈尔滨工业大学基础与交叉科学研究院高性能计算中心 哈尔滨 150001)

摘 要 连接聚集操作是一种常用并且非常耗时的数据库操作.相对于准确查询,满足用户给定置信区间的近似结果由于其快得多的响应时间,更受用户的欢迎.作者分析发现现有的工作无法以既高效又满足给定的任意置信区间方式来处理近似连接聚集,因此提出了一种新的算法——(p,ε)-近似连接聚集查询($p\varepsilon$-AJA)来有效地返回满足任意置信区间的近似连接聚集结果.文章提出且预计算两个数据结构:连接随机样本 (JRS)和连接位置索引对表(JPIPT).利用 JRS,$p\varepsilon$-AJA 向用户返回近似结果的快速响应.如果利用 JRS 得到的近似结果没有满足给定的置信区间,$p\varepsilon$-AJA 利用 JPIPT 获得更多的随机连接元组.文中提出一种采样算法来获得 JPIPT 给定数量的样本,并且利用获得的 JPIPT 样本,该文提出的算法可通过对连接表的一遍顺序扫描获得连接元组.该文还提供了 JPIPT 和 JRS 有效的构建和维护算法.实验结果表明:$p\varepsilon$-AJA 可以获得相对于准确查询 1～5 个数量级的加速,并且可以有效地完成 JPIPT 和 JRS 的构建和维护操作.

关键词 $p\varepsilon$-近似连接聚集;连接位置索引对表;连接随机样本;海量数据

中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2010.01919

Approximate Join Aggregate on Massive Data

HAN Xi-Xian[1)] YANG Dong-Hua[2)] LI Jian-Zhong[1)]

[1)](*School of Computer Science and Technology, Harbin Institute of Technology, Harbin* 150001)

[2)](*Center for High Performance Computing, Academy of Fundamental and Interdisciplinary Sciences, Harbin Institute of Technology, Harbin* 150001)

Abstract Join aggregate is a commonly used but time-consuming operation in database. Comparing to exact queries, approximate results satisfying user-specified confidence intervals are more attractive for their much faster responses. None of the previous work can process approximate join aggregate with both high efficiency and an arbitrarily specified confidence interval. This paper proposes a novel algorithm, (p,ε)-Approximate Join Aggregate ($p\varepsilon$-AJA), which is able to return approximate results for arbitrary confidence interval efficiently. Two data structures, join random sample (JRS) and join positional index pair table (JPIPT), are presented and pre-computed in $p\varepsilon$-AJA. $p\varepsilon$-AJA first makes use of JRS to make a quick response of approximate results to users. If the approximate results from JRS do not satisfy the given confidence interval, JPIPT is exploited to obtain more random join tuples. A sampling algorithm is provided to sample JPIPT tuples of specified size. Algorithms are also presented to retrieve join tuples by sampled JPIPT tuples in one-pass sequential scan. The construction and maintenance of JPIPT and JRS are provided in this paper. The experimental results show that $p\varepsilon$-AJA obtains approximate results for

收稿日期:2010-08-22.本课题得到国家“九七三”重点基础研究发展规划项目基金(2006CB303005)、国家自然科学基金(60903016,60533110,60773063)、新世纪优秀人才支持计划(NCET-05-0333)、黑龙江省教育厅科学技术研究项目(11531276)与 NSFC-RGC of China (60831160525)资助.**韩希先**,男,1981 年生,博士研究生,主要研究方向为海量数据管理和数据密集型计算. E-mail: xxhan1981@163.com. **杨东华**,男,1976 年生,博士,讲师,主要研究方向为海量数据管理和数据密集型计算等. **李建中**(通信作者),男,1950 年生,教授,博士生导师,主要研究领域为数据库和传感器网络等. E-mail: lijzh@hit.edu.cn.

arbitrary confidence intervals with a speedup by 1 to 5 orders of magnitude compared to exact queries and the update operations for JPIPT and JRS are efficient.

Keywords $p\varepsilon$-AJA; join positional index pair table; join random sample; massive data

1 引 言

数据的指数级增长给数据库查询处理提出了更严峻的挑战.连接聚集查询是一种常用并且重要的数据库操作,基于一些公共信息,连接聚集返回两表或多表的的统计信息.TPC-H 是在线分析处理环境中的一种典型的评价标准,其中,所有的 22 个查询中有 16 个涉及到连接聚集操作.连接聚集也是一种很耗时的操作,因为它需要在大量数据上执行昂贵的连接和聚集操作.相对于准确查询,在少得多的响应时间内,返回满足用户给定置信区间的近似结果的近似查询方式更受用户欢迎.例如,在海量数据上执行统计分析(求和、平均等),用户关心的只是主要的前面几位精确数字.研究人员已经提出一些近似连接聚集算法来返回近似结果.

Join synopsis 方法[1]利用预计算的连接样本有效地返回结果.但是,该方法只能返回固定置信区间的近似结果,无法满足任意给定的置信区间.在线连接聚集[2-4]是另一类近似连接聚集方法.该方法在线地返回近似结果,直到满足给定的置信区间.但是,该方法对于元组随机获取的要求使得其无法有效处理海量数据上的查询.总的来说,之前工作无法以既高效又满足给定的任意置信区间方式来处理近似连接聚集.

本文提出一种新的近似连接聚集查询(p,ε)-Approximate Join Aggregate($p\varepsilon$-AJA).该算法可以有效地返回满足任意给定置信区间的近似连接聚集结果.$p\varepsilon$-AJA 预计算并且利用以下两个数据结构:连接随机样本(Join Random Sample,JRS)维护预计算的固定数量的随机连接元组,连接位置索引对表(Join Positional Index Pair Table,JPIPT)说明连接操作的每个结果元组的位置索引对信息.$p\varepsilon$-AJA 的执行包括两个阶段.在阶段 1,给定具有置信区间 $CI=(p,\varepsilon)$ 的查询 Q,$p\varepsilon$-AJA 首先利用 JRS 向用户返回近似结果的快速响应.如果利用 JRS 获得的近似结果满足 CI,则执行结束.否则,$p\varepsilon$-AJA 进入阶段 2 获得满足 CI 的更多的随机连接元组.本文提出采样算法获得 JPIPT 的给定数量的样本,然后,$p\varepsilon$-AJA 利用获得的 JPIPT 样本从连接表中获得需要的连接元组,计算并且返回连接聚集结果.本文提出一种新的元组抽取算法来保证只需要对数据表的一次顺序扫描操作就可以获得所需要的元组.现有的方法要么没有利用采样,要么利用采样来获得连接表的随机元组而不是随机连接结果元组.

本文提出了 JPIPT 和 JRS 的构建和维护算法.在本文中,数据表以面向列的方式存储[5].JPIPT 的构建操作相当于连接属性上的连接操作并且以位置索引为输出结果.本文提出一个缓存敏感的构建算法利用缓存来快速构建 JPIPT.利用获得的 JPIPT,JRS 可以通过对数据表执行采样来获得,从而不需要执行连接操作.本文提出部分解析的合并排序算法 MSPR (Merge Sort with Partial Resolution)来加快 JPIPT 的维护操作,本文指的解析是把文件的字节流转换成元组格式的 CPU 操作.MSPR 利用海量数据的 read/append-only 的特点来减少合并操作的解析费用从而加快其执行.和已有数据相比,新插入的数据非常小,所以 JRS 的更新操作可以较快完成,因为 JRS 中只有很少几个元组需要被新数据替换.这里需要注意的是,只要某个存储模型支持通过位置索引来获得元组,$p\varepsilon$-AJA 就可以应用于该存储模型.实际上,这包括了现在所有的存储模型(行存储和列存储).

实验结果表明,$p\varepsilon$-AJA 可以获得满足任意置信区间的近似结果,并且获得相对于准确查询 1～5 个数量级的加速,同时,对于 1250000 条元组插入 268G 数据的更新操作,$p\varepsilon$-AJA 只需要不到 380s 的时间来完成 JPIPT 和 JRS 的维护操作.

本文的主要贡献如下:

(1)本文提出一种新的近似聚集算法 $p\varepsilon$-AJA,该算法可以有效地处理满足任意置信区间的近似连接聚集,而不需要执行昂贵的 ad-hoc 连接操作.

(2)本文提出一种新的采样算法来获得指定数量的样本,并且提供了该采样算法的正确性证明.

(3)本文提出了 JPIPT 和 JRS 有效的构建和维护算法,分析了 MSPR 用到的优化桶大小.

(4)本文设计了较全面的实验来评价 $p\varepsilon$-AJA 的性能以及 JPIPT 和 JRS 的构建和维护性能.实验结果表明,$p\varepsilon$-AJA 可以有效地完成查询和维护操作.

本文第 2 节讨论相关工作；第 3 节介绍 $p\varepsilon$-AJA 的执行过程；第 4 和第 5 节分别讨论 JPIPT 和 JRS 的构建和维护操作；第 6 节是性能评价部分；第 7 节是本文的结论部分.

2　相关工作

2.1　近似查询处理

在海量数据上，返回精确结果的查询处理会导致很长的执行时间. 在某些场合下，以少得多的响应时间返回近似结果是更好的选择. Hellerstein[6] 等提出在线聚集的思想，在线地执行聚集查询并且连续地向用户提供反馈直到满足用户的要求. 该方法要求数据以随机的次序获得. Cheng[7] 和 Wu[8] 等把近似聚集查询的思想扩展到分布式环境. Hass[9] 为在线聚集操作的置信区间计算提供了计算公式. Acharya[1] 等利用预计算的 Join synopsis 来有效地回答近似连接聚集，但是该方法只能保证固定的置信区间. 在线连接聚集[2-4] 是另一类近似连接聚集方法. 该方法可以在线地执行并且持续地返回近似结果，一旦当前的近似结果满足用户的置信区间要求，则执行结束. 但是该方法的元组随机获取的前提使得其无法有效地执行. Spiegel[10] 等引入了 TuG Synopses 的概念来概括数据，近似查询处理作为概要图上的遍历操作执行. 本文中，$p\varepsilon$-AJA 不需要假设获取数据表的随机元组，只要存储模型支持通过位置索引来获得元组，$p\varepsilon$-AJA 就可以执行.

2.2　采　样

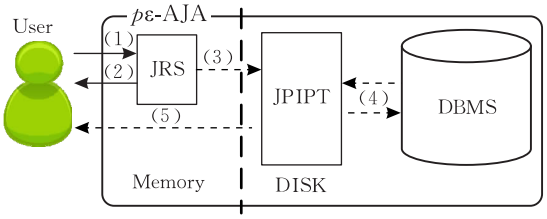
采样是数据库操作中一种很有用的技术，并且已被人们广泛研究[11]. Chaudhuri[12] 等提出最小化给定工作负载的误差的采样方法. Hass[13] 等提出了双层 Bernoulli 采样模式，结合元组级别和页级别来获得最优的处理速度和统计精度的折衷. Gemulla[14] 等提出方法来增量地维护变化的随机样本. Olken[11] 等提出数据库操作的采样算法，并且提供了对连接操作结果的采样方法. 可是，它要求在每个连接属性上构建索引，并且要求在单个表上构建完全的统计信息. Chaudhuri[15] 等提出了一种采样技术，只需要在第一个关系上构建部分统计信息就可以获得连接操作的随机样本.

3　(p,ε)-近似连接聚集算法

3.1　$p\varepsilon$-AJA 概述

假设 T_1 和 T_2 是参与连接操作的表. 在执行阶段，T_1 作为外关系而 T_2 作为内关系. 用户提交的连接聚集查询 Q 的形式为："select op(expression(t_i)) from T_1, T_2 where $T_1.key=T_2.key$ *specified* $CI=(p,\varepsilon)$". 这里，op 表示统计分析操作（sum，avg 等），t_i 表示 $T_1 \bowtie T_2$ 的连接元组，$CI=(p,\varepsilon)$ 是用户给定的置信区间. 假设查询 Q 的准确解是 μ，φ 是当前获得近似结果. 如果 φ 满足 CI，即：$\mu\in(\varphi-\varepsilon,\varphi+\varepsilon)$，则称 φ 是可接受的. 本文以 sum 和 avg 为例来讨论 $p\varepsilon$-AJA 的操作，其它的聚集操作可以类似处理.

我们已经说明，由于 $p\varepsilon$-AJA 只要求可以通过位置索引获得元组，$p\varepsilon$-AJA 支持在行存储和列存储上的查询，$p\varepsilon$-AJA 采用面向列的方式存储数据表只是为获得更好的性能. 通过列存储方式（每个属性的数据存储于独立的文件），我们不但可以减少扫描时的引用步长（具有引用步长为 k 的访问模式指的是顺序访问连续空间第 k 个字节的数据，k 表示跳过的字节数. 步长越小，则空间局部性就越好，在存储器中以大步长访问不同位置的数据的程序不具有良好的空间局部性），而且还可以只读取涉及到的列文件. $p\varepsilon$-AJA 处理查询 Q 的操作包括两个阶段，其执行体系结构如图 1 所示. 连接随机样本（JRS）是内存中维护的一组 $T_1 \bowtie T_2$ 的随机样本. 阶段 1，$p\varepsilon$-AJA 直接利用 JRS 来回答查询 Q. 如果得到的近似结果满足 CI，则 $p\varepsilon$-AJA 执行结束. 否则，$p\varepsilon$-AJA 进入第 2 阶段，利用 JPIPT 来获得 $T_1 \bowtie T_2$ 的更多的随机元组，保证近似结果满足置信区间.



(1) submit Q with $CI=(p,\varepsilon)$, (2) return result by JRS to user, (3) If result by JRS does not satisfy CI, send Q to JPIPT for sampling more data, (4) Sample data from DBMS by JPIPT, (5) return data of (4) to user.

图 1　$p\varepsilon$-AJA 体系结构

3.2　阶段 1：利用 JRS 执行查询

我们首先给出本节用到的一些定义.

定义 1（随机连接元组）.　给定表 T_1 和 T_2，$T_1.key_1$ 和 $T_2.key_2$ 分别是 T_1 和 T_2 的连接属性，$T_1 \bowtie T_2$ 的结果包括 N 个元组，则 (t_1,t_2) 是 $T_1 \bowtie T_2$ 的随机连接元组，如果：(1) $t_1\in T_1$，$t_2\in T_2$；(2) $t_1.key_1=t_2.key_2$；(3) $Pr((t_1,t_2))=1/N$（$Pr((t_1,t_2))$ 是 (t_1,t_2) 被选中的概率）.

定义 2（连接随机样本）.　给定表 T_1 和 T_2，

JRS 是 $T_1 \bowtie T_2$ 的大小为 k 的连接随机样本，如果：(1) JRS 中的元组是各不相同的；(2) $\forall (t_1, t_2) \in JRS$，则 (t_1, t_2) 是 $T_1 \bowtie T_2$ 的一个随机连接元组；(3) $|JRS| = k$.

算法 1. $p\varepsilon$-AJA-$stage1()$

```
//ζ1 = n and ζ2 = |JPIPT| if op = sum
//ζ1 = 1 and ζ2 = n if op = avg
1. set x = 0, y = 0 and value = 0
2. for i = 1 to n do
3.    value += expression(JRS[i]) × ζ1
4.    x += [expression(JRS[i])]^2
5.    y += expression(JRS[i])
6. end for
7. Ȳn = value / n
8. sd = sqrt((n-1)^-1 × (x - n × (y/n)^2))
9. m = (zp × sd/ε)^2
10. if m ≤ n then
11.    return Rs1 = Ȳn × ζ2 / n
12. else
13.    call pε-AJA-stage2(N, m, m-n, Ȳn)
14. end if
```

由于内存容量和维护复杂性的限制，JRS 维护 $T_1 \bowtie T_2$ 的 n 个随机元组，n 的值取决于分配的内存容量 M_{JRS}、T_1 的元组大小 TS_1 和 T_2 的元组大小 TS_2，通常 $n = \lfloor M_{JRS}/(TS_1 + TS_2) \rfloor$. JRS 的构建和维护算法分别在第 4 节和第 5 节介绍. 假设利用 JRS 获得的近似结果是 $\bar{Y}_n$，其计算公式如下：

$$\bar{Y}_n = \frac{1}{n} \times \sum_{i=1}^{n} (expression(t_i) \times \zeta_1), \quad \zeta_1 = \begin{cases} n, & op = sum \\ 1, & op = avg \end{cases} \tag{1}$$

假设 μ 是查询 Q 的准确解，$\bar{Y}_n$ 满足置信区间 CI 意味着 $P(|\bar{Y}_n - \mu| \leq \varepsilon) = p$. 根据中心极限定理，大量独立随机变量的和可以很好地近似为正态分布. 给定足够大的 JRS，随机变量 $\bar{Y}_n$ 可以表示为期望为 μ，方差为 σ^2 的正态分布. σ^2 是随机变量 $expression(t_i)$ 的方差，由于 n 较大，σ 可以用 $expression(t_i)$ 的标准差来代替. 所以，查询 Q 的置信区间计算公式如下所示：

$$P(|\bar{Y}_n - \mu| \leq \varepsilon) = P\left(\left|\frac{\sqrt{n}(\bar{Y}_n - \mu)}{\sigma}\right| \leq \frac{\varepsilon\sqrt{n}}{\sigma}\right) = 2\varphi\left(\frac{\varepsilon\sqrt{n}}{\sigma}\right) - 1 \tag{2}$$

假设 $2\varphi(\varepsilon\sqrt{n}/\sigma) - 1 = p$，使得 $P(|\bar{Y}_n - \mu| \leq \varepsilon) = p$ 成立，并且让 z_p 表示标准正态分布的 $(p+1)/2$ 分位数. 要使得近似结果满足 $CI = (p, \varepsilon)$，则需要随机元组的数量 $m = (z_p \times \sigma/\varepsilon)^2$. σ^2 的计算公式如下：

$$\sigma^2 \approx sd^2 = \frac{1}{n-1} \times \sum_{i=1}^{n} (expression(t_i) - T_n)^2 = \frac{1}{n-1} \times \left[\left(\sum_{i=1}^{n} expression^2(t_i)\right) - n \times T_n^2\right] \tag{3}$$

其中，$T_n = n^{-1} \times \left(\sum_{i=1}^{n} expression(t_i)\right)$. 根据 σ 的计算公式，$expression(t_i)$ 的标准差可以在对 JRS 计算近似结果的同时计算，所以阶段 1 只需要对 JRS 执行一次扫描. 阶段 1 的执行过程如算法 1 所示.

假设 R_{s1} 是阶段 1 获得的近似结果，其计算公式如下所示：

$$R_{s1} = \frac{\bar{Y}_n \times \zeta_2}{n}, \quad \zeta_2 = \begin{cases} N, & op = sum \\ n, & op = avg \end{cases} \tag{4}$$

其中，N 是 $T_1 \bowtie T_2$ 操作的元组数量. 如果所需要的随机元组数量 $m \leq n$，则 R_{s1} 就满足给定的置信区间，$p\varepsilon$-AJA 可以直接把 R_{s1} 返回给用户. 否则，$p\varepsilon$-AJA 进入阶段 2.

3.3 阶段 2：利用 JPIPT 执行查询

我们先给出本节用到的一些定义.

定义 3(位置索引). 给定表 T，元组 $t \in T$ 的位置索引(Positional Index，PI)是 i，如果 t 是 T 的第 i 元组.

我们用 $T(i)$ 表示 T 中位置索引为 i 的元组.

定义 4(连接位置索引对表). 给定表 T_1 和 T_2，$T_1.key_1$ 和 $T_2.key_2$ 分别是 T_1 和 T_2 的连接属性，$T_1 \bowtie T_2$ 包括 N 个元组，JPIPT 是 $T_1 \bowtie T_2$ 的连接位置索引对表，如果 JPIPT 满足条件：(1) $|JPIPT| = N$；(2) $\forall i, j, (PI_1 : i, PI_2 : j) \in JPIPT$，当且仅当 $T_1(i).key_1 = T_2(j).key_2$.

在本文中，我们可以把 JPIPT 看作具有两个属性 (PI_1, PI_2) 的表. 阶段 2 需要利用 JPIPT 获得额外的 $(m-n)$ 个随机连接元组. JPIPT 的构建和维护分别在第 4 节和第 5 节中介绍. 由于篇幅的限制，本文只讨论 $(m-n)$ 个随机连接元组可以放入内存的情况(这也是绝大多数的情况)，处理数据量超过内存容量的情况见文献[16].

$p\varepsilon$-AJA 的第 2 阶段首先需要获得 JPIPT 的 $k = (m-n)$ 个随机样本点. 已知 JPIPT 的元组个数 N，则对 JPIPT 的采样问题可以形式化定义为：给定具有 N 个元素的集合，要求从集合中均匀无放回地抽取 k 个元素. 该采样问题的一种简单的想法是：产生 $[1, N]$ 的 k 个不同的随机数，并且从 JPIPT 中

抽取对应位置索引的元组. 为保证可以通过一遍顺序扫描就获得样本，需要对 k 个随机数排序，否则采样产生的较多磁盘随机寻道操作会严重影响采样效率. 然而当 k 较大时，排序操作就需要较高的费用，所以该方法并不合适. 另一个比较明显的想法是：顺序扫描 N 个元组，以概率 k/N 选择每个元组. 该采样方法满足二项分布，它只能保证获得随机元组个数的期望值是 k，其方差为 $k\times(1-k/N)$，从而该方法无法保证能够获得恰好 k 个随机元组. $p\varepsilon$-AJA 提出一种新的采样方法(其伪代码在算法 2 中给出)来解决这一问题. 在对 N 个元组的采样过程中，如果已经扫描了 t 个元组，并且采样操作已经选择了 k' 个元组，则算法 2 选择第 $t+1$ 个元组为随机样本点的概率是 $(k-k')/(N-t)$. 算法 2 采用条件概率而不是绝对概率对数据进行采样.

这里需要注意的是，算法 2 获得的样本不包括 JRS 中的元组，从而保证采样的随机性. JRS 中的元组对应的 JPIPT 元组的位置索引在有序数组中存储. 当对 JPIPT 进行扫描的时候，算法 2 忽略位置索引在有序数组中出现的元组. 定理 1 和 2 给出了算法 2 的正确性证明.

算法 2.　*Sample-JPIPT*(int N, int k).

```
//N=|JPIPT|: the number of join results
//random-generator: return random value in (0, 1)
 1. set t=0 and k'=0
 2. JPIPT-tuple JPIPT-sample[k]
 3. p=random-generator(0,1)
 4. if( p<(k-k')/(N-t) ) then
 5.    read current JPIPT tuple to JPIPT-sample[k']
 6.    k'+=1 and t+=1
 7.    if (k'<k) then
 8.       goto 2
 9.    else
10.       return JPIPT-sample
11.    end if
12. else
13.    skip current JPIPT tuple
14.    t+=1
15.    if (N-t==k-k') then
16.       read left JPIPT tuples to
              JPIPT-sample[k',...,k]
17.       return JPIPT-sample
18.    else
19.       goto 2
20.    end if
21. end if
```

定理 1.　算法 2 恰好获得 k 个 JPIPT 元组.

证明.　很明显，算法 2 获得的样本点肯定不会超过 k 个. 所以，我们只需要证明算法 2 获得样本点不会少于 k 个. 而这也是不可能的，已知 $k<N$(因为如果 $k\geq N$ 的话，算法 2 只需要返回所有的 JPIPT 元组)，如果未处理的 JPIPT 元组数刚好等于 $k-k'$，那么我们就选择所有剩余的元组. 所以算法 2 获得元组数量不会少于 k 个.　证毕.

定理 2.　算法 2 获得 JPIPT 的 k 个随机元组.

证明.　根据定理 1，算法 2 可以获得恰好 k 个元组，假设获得的 k 个元组在 JPIPT 中的位置索引是 $(PI_0=0)<1\leq PI_1<PI_2<\cdots<PI_k\leq N<(PI_{k+1}=N+1)$. 已知算法 2 是否选择位置索引值为 t 的元组的概率 p_t 是：

$$p_t=\begin{cases}\dfrac{N-(t-1)-k+k'}{N-(t-1)}, & PI_{k'}<t<PI_{k'+1}\\ \dfrac{k-k'}{N-(t-1)}, & t=PI_{k'+1}\end{cases}\tag{5}$$

如果 $PI_{k'}<t<PI_{k'+1}$，当前位置索引为 t 的元组没有被算法 2 选择. 已知在处理该元组之前，算法已经选择 k' 个元组，所以算法不选择该元组的概率为 $1-(k-k')/(N-(t-1))=(N-(t-1)-k+k')/(N-(t-1))$. 当 $t=PI_{k'+1}$ 时，由于之前已经选择 k' 个元组，所以选择该元组的概率为 $(k-k')/(N-(t-1))$. 根据 P_t 的定义，我们知道算法 2 获得 k 个元组的概率 $p=p_1\times p_2\times\cdots\times p_N$. 很明显，$p$ 的分母为 $N!$，对于没有被选择的那些元组，p_t 的分子包含项 $N-k,N-k-1,\cdots,1$，被选中的 k 个元组对应的 p_t 的分子包含项 $k,k-1,\cdots,1$. 所以 $p=\dfrac{(N-k)!\times k!}{N!}=\dfrac{1}{C(N,k)}$，$C(N,k)$ 是从 N 个元组中抽取出 k 个元组的组合数. 因此算法 2 获得的 k 个元组是 JPIPT 的 k 个随机样本点.　证毕.

假设 *JPIPT-sample* 是一个缓冲区，用来存储算法 2 获得的 k 个 JPIPT 随机元组. 根据定义我们知道，*JPIPT-sample* 中的元组对应的 T_1 和 T_2 元组对是 $T_1\bowtie T_2$ 的 k 个连接随机元组. 接下来我们介绍如何利用 *JPIPT-sample* 来获得 $T_1\bowtie T_2$ 的随机元组.

根据用户提交的查询表达式 $expression(t_i)$，$p\varepsilon$-AJA 的处理需要考虑 3 种情况：(1) $expression(t_i)$ 只涉及表 T_1 的属性；(2) $expression(t_i)$ 只涉及表 T_2 的属性；(3) $expression(t_i)$ 涉及表 T_1 和 T_2 的属性. 以下对 3 种情况分别进行讨论.

情况 A. $expression(t_i)$只涉及表 T_1的属性.

已知 JPIPT 的元组根据属性 PI_1排序(其证明见 4.1 节). $JPIPT\text{-}sample$ 是 JPIPT 的子序列,所以 $JPIPT\text{-}sample$ 也根据 PI_1排序. 情况 A 中,近似查询 Q 只涉及表 T_1的属性,所以利用 $JPIPT\text{-}sample$ 元组的属性 PI_1的值,$p\varepsilon$-AJA 只需要对表 T_1执行一次顺序扫描就可以获得指定位置索引对的 T_1元组.

算法 3 描述了情况 A 下 $p\varepsilon$-AJA 第 2 阶段的基本处理过程. 对表 T_1执行扫描过程时,算法 3 直接跳过不需要处理的元组,从而加快了处理速度.

情况 B. $expression(t_i)$只涉及表 T_2的属性.

情况 B 下,$p\varepsilon$-AJA 根据 $JPIPT\text{-}sample$ 元组的 PI_2属性的值来获得相应的 T_2元组. 注意到,和 PI_1属性不同,PI_2属性的值不是单调的(详细情况见 4.1 节),$p\varepsilon$-AJA 无法直接利用 PI_2的值对表 T_2执行一遍顺序扫描而获得结果,如果利用非排序的 PI_2域直接抽取指定位置索引的元组将导致大量的磁盘随机读取操作,从而严重影响磁盘效率. 所以,情况 B 需要根据 PI_2属性对 $JPIPT\text{-}sample$ 执行一次排序操作. 排序后情况 B 的操作类似于情况 A.

算法 3. $p\varepsilon\text{-}AJA\text{-}stage2\text{-}A$(int N,int m,int k, double $\overline{Y}_n$).

//k: the sample size in stage 2
//ζ_1: n if $op=sum$; 1 if $op=avg$
//$flag$: true if $op=sum$; false if $op=avg$
1. $JPIPTSample=sample\text{-}JPIPT(N,k)$
2. int $currpi=0$, $value=0$
3. T1_tuple tup
4. for $i=1$ to $|JPIPTsample|$
5. $\quad skiptuple(JPIPTSample[i].PI_1-currpi-1)$
6. $\quad$ read current tuple to tup
7. $\quad value+=expression(tup)\times\zeta_1$
8. $\quad currpi=JPIPTsample[i].PI_1+1$
9. end for
10. $\overline{Y}_k=\dfrac{value}{n}$
11. if (flag) then
12. $\quad$ return $\dfrac{(\overline{Y}_n+\overline{Y}_k)\times N}{m}$
13. else
14. $\quad$ return $\dfrac{\overline{Y}_n\times n}{m}+\dfrac{\overline{Y}_k\times k}{m}$
15. end if

情况 C. $expression(t_i)$涉及表 T_1和 T_2的属性.

情况 C 的处理方法要比情况 A 和 B 复杂,因为聚集操作涉及到两个表的属性. 令 BUF_T_1和 BUF_T_2是两个缓冲区,分别存储 T_1和 T_2的随机元组. 在情况 C 下,$p\varepsilon$-AJA 的操作步骤如下:首先利用 $JPIPT\text{-}sample$ 获得对应的 T_1的元组(见情况 A 的处理),然后利用 $JPIPT\text{-}sample$ 获得对应的 T_2的元组(见情况 B 的处理),最后对 BUF_T_I和 BUF_T_2构成的连接元组执行聚集操作.

和情况 A 和 B 的处理相似,利用 $JPIPT\text{-}sample$ 可以利用对 T_1和 T_2的一次顺序扫描获得 BUF_T_1和 BUF_T_2. 如情况 B 所示,由于在获得 BUF_T_2前,$JPIPT\text{-}sample$ 需要根据 PI_2属性执行一次排序操作,所以当前 BUF_T_1和 BUF_T_2的相同顺序位置的元组不是 $T_1\bowtie T_2$的元组. $p\varepsilon$-AJA 需要对 BUF_T2 根据 PI_1的值再执行一次排序操作得到 $SBUF\_T_2$,此时 BUF_T_1和 $SBUF_T_2$的相同顺序位置的元组才构成一个随机连接元组. 算法 4 介绍了情况 C 下 $p\varepsilon$-AJA 第 2 阶段的基本处理过程. 定理 3 证明了算法 4 的正确性.

算法 4. $p\varepsilon\text{-}AJA\text{-}stage2\text{-}C$(int N,int m,int k, double $\overline{Y}_n$).

//k: the sample size in stage 2
//ζ_1: n if $op=sum$; 1 if $op=avg$
//$flag$: true if $op=sum$; false if $op=avg$
1. $JPIPTSample=sample\text{-}JPIPT(N,k)$
2. set $value=0$
3. BUF_T_1 is obtained as in case A
4. BUF_T_2 is obtained as in case B
5. $sort(BUF\_T_2)$ on PI_1 to obtain $SBUF\_T_2$
6. for $i=1$ to k do
7. $\quad value+=expression(BUF\_T_1[i],$ $SBUF\_T_2[i])\times\zeta_1$
8. end for
9. $\overline{Y}_k=\dfrac{value}{n}$
10. if(flag) then
11. $\quad$ return $\dfrac{(\overline{Y}_n+\overline{Y}_k)\times N}{m}$
12. else
13. return $\dfrac{\overline{Y}_n\times n}{m}+\dfrac{\overline{Y}_k\times k}{m}$
14. end if

定理 3. BUF_T_1和 $SBUF_T_2$的相同位置索引的元组是 $T_1\bowtie T_2$的元组.

证明. 已知 $JPIPT\text{-}sample$ 维护着 k 个 JPIPT 随机元组,BUF_T_1保存类似于情况 A 操作获得的 T_1元组,BUF_T_2保存类似于情况 B 操作获得的没有经过第 2 次排序的 T_2元组. JPIPT 的 PI_1域的数据可以看作序列 $S_1=\{i_1,i_2,\cdots,i_k\}$,$PI_2$域的数据可以看作序列 $S_2=\{j_1,j_2,\cdots,j_k\}$. 由于 JPIPT 根据 PI_1排序,所以 $i_1\leq i_2\leq\cdots\leq i_k$,$BUF_T_1$保存的元

组具有新的位置索引 $P_1=\{1,2,\cdots,k\}$(BUF_T_1 中的第 i 个元组具有新的位置索引 i)，并且 P_1 的元素和 S_1 的元素具有 1-1 对应关系（虽然 S_1 可能存在重复值，可是这些重复值表示相同的 T_1 元组，所以其顺序不会影响结果的正确性）. 我们把 S_1 和 P_1 结合而获得一个新的序列 $SP_1=\{(i_1,1),(i_2,2),\cdots,(i_k,k)\}$. 类似的，我们获得序列 $SP_2=\{(j_1,(i_1,1)),(j_2,(i_2,2)),\cdots,(j_k,(i_k,k))\}$，其中 $(j_a,(i_a,a))$ 表示 $T_2(j_a)$ 是 $JPIPT(a)$ 对应的连接结果的一部分. 因为 SP_2 不是根据 j_a 排序的，为方便对 T_2 的顺序扫描来获得结果，情况 B 的处理对 SP_2 根据 j_a 排序，获得 $SP_2'=\{(j_1',(i_{s_1},s_1)),(j_2',(i_{s_2},s_2)),\cdots,(j_k',(i_{s_k},s_k))\}$，$j_1'\leq j_2'\leq\cdots\leq j_k'$，$\{s_1,s_2,\cdots,s_k\}$ 是 P_1 的一个排列. 很明显，BUF_T_1 和 BUF_T_2 中具有相同的新位置索引的元组不是连接操作的结果元组，因为通常 $s_a\neq a$. 要保证结果的正确性，BUF_T_2 需要执行另一次根据 s_a 排序的操作. 由于 S_1 和 P_1 的元素有 1－1 对应关系，所以 BUF_T_2 也可以根据 i_{s_a} 排序，即根据对应的 PI_1 排序获得 $SBUF\_T_2$. 证毕.

$p\varepsilon$-AJA 在第 2 阶段完毕时已经获得了满足指定置信区间的 $m=(z_p\times\sigma/\varepsilon)^2$ 个随机连接元组，已经可以返回满足用户要求的近似结果. 结合第 1 阶段和第 2 阶段的结果可以得到近似结果 $R_{p\varepsilon\text{-AJA}}$，计算公式如下：

$$R_{p\varepsilon\text{-AJA}}=\begin{cases}\dfrac{(\bar{Y}_n+\bar{Y}_k)\times|JPIPT|}{m}, & op=sum\\ \dfrac{\bar{Y}_n\times n}{m}+\dfrac{\bar{Y}_k\times(m-n)}{m}, & op=avg\end{cases}\tag{6}$$

其中，$\bar{Y}_k$ 是阶段 2 得到的计算结果，其计算公式和 $\bar{Y}_n$ 类似. 至此，$p\varepsilon$-AJA 的执行就结束了，把结果 $R_{p\varepsilon\text{-AJA}}$ 返回给用户.

讨论.　这里我们简单讨论 $p\varepsilon$-AJA 如何处理检索条件. 本文对于检索条件的扩展比较容易. $p\varepsilon$-AJA 可以利用多维索引来组织 JPIPT，其中，连接结果元组的属性域（值）作为索引键，如果给定连接元组的属性值落入索引的属性域（值），则对应的 JPIPT 元组作为对应索引键的索引值. 给定查询的检索条件，$p\varepsilon$-AJA 的处理和无检索条件情况的处理类似，但是此时算法 2 对 JPIPT 进行采样的时候只需要对满足条件的索引值进行采样.

在这一节我们看到 $p\varepsilon$-AJA 如何利用 JRS 和 JPIPT 来处理近似连接聚集查询. 在接下来两节，我们分别介绍如何构建和维护 JPIPT 和 JRS.

4　构建 JPIPT 和 JRS

在 4.1 节中，本文提出一种高速缓存敏感的构建算法来快速构建 JPIPT. 利用获得的 JPIPT，4.2 节介绍了如何快速构建 JRS.

4.1　构建 JPIPT

在典型的海量数据环境中，例如数据仓库、科学计算等，涉及到外键-主键的连接操作非常普遍. 在 TPC-H 中，所有的 19 个连接操作都是外键-主键连接. 此外，人们一般都知道最经常用到的连接属性. JPIPT 可以有效地支持涉及到连接聚集的不同属性的统计操作，因此预计算并且维护 JPIPT 是值得的，并且获得的 JPIPT 可以反复使用直到下一次更新操作. 和源数据文件相比，JPIPT 比较小. 在我们的试验中，268GB 的数据集只需要维护 11GB 的 JPIPT 数据. 所以，数据仓库可以构建多个连接属性对的 JPIPT 信息，来有效支持该数据仓库上的连接聚集操作.

在本文中，我们可以把 JPIPT 看作具有两个属性的表. $p\varepsilon$-AJA 把 T_1 和 T_2 的连接属性存储为两个独立的列文件：C_1 和 C_2. 每个列文件都包含一个隐含的属性：PI. C_1 和 C_2 的模式可以分别看作 $C_1(key_1,PI_1)$ 和 $C_2(key_2,PI_2)$. 从本质上来说，JPIPT 的构建操作就是执行如下的连接操作：“select PI_1,PI_2 from C_1, C_2 where $C_1.key_1=C_2.key_2$”.

传统的 Hash 连接算法并没有考虑高速缓存，从而导致了较低的高速缓存和 CPU 利用率[17]. 本文提出 C3-JPIPT 算法（Cache-Conscious Construction of JPIPT），该算法充分利用高速缓存来更快地构建 JPIPT，C3-JPIPT 的伪代码如算法 5 所示.

C3-JPIPT 的执行包括两个步骤：划分和连接. 划分把每个表划分成 $N_{p1}=\dfrac{2\times(S_{C_1}+S_{C_2})}{M}$ 个子表. 其中，S_{C_1} 和 S_{C_2} 分别表示列文件 C_1 和 C_2 的大小，M 表示 C3-JPIPT 利用的内存容量. C3-JPIPT 产生比传统 Hash 连接操作更多的分片数量，这是因为该算法的连接阶段要求同时把内关系和外关系的单个分片对读入内存，以便接下来的操作. 在划分时，C3-JPIPT 实例化每个元组的位置索引值. 注意：如果和外关系相比，内关系非常地小，那么 JPIPT 的构建过程就采用传统的 Hash 算法而不是 C3-JPIPT.

当 C3-JPIPT 进入连接操作时，每一对子表 C_{1i} 和 $C_{2i}(1\leq i\leq N_{p1})$ 依次读入内存，并且划分为 $N_{p2}=$

$\frac{S_{C_{2i}}}{C}$个分片，使得 C_{2i} 的每个分片可以放入高速缓存. 其中，$S_{C_{2i}}$ 表示读入内存的 C_{2i} 大小，C 表示高速缓存的大小. 如果 $N_{p2}>64$，则该划分操作采用多轮划分方法来减少 TLB miss(TLB：翻译后备缓冲器，存储最近处理过的 64 个虚拟页-物理页地址影射，如果当前请求的虚拟页地址转换没有存储在 TLB，则说发生一次 TLB miss 操作)，使得每一轮划分的分片数量少于 64，并且前一轮获得的分片进一步划分成多个子分片. 该过程继续执行直到获得的分片数量达到 N_{p2}. C3-JPIPT 采用的多轮划分方法类似于 Radix-cluster 算法[17].

算法 5. $C3\text{-}JPIPT(C_1,C_2)$.

1. int $N_{p1}=\frac{2\times(S_{c_1}+S_{c_2})}{M}$
2. *split* C_1 and C_2 into N_{p1} partitions
3. for $i=1$ to N_{p1} do
4. read C_{1i} and C_{2i} into memory
5. int $N_{p2}=\frac{S_{C_{2i}}}{C}$
6. *split* C_{1i} and C_{2i} into N_{p2} sub-tablet pairs
7. for $j=1$ to N_{p2} do
8. $JPIPT_{ij}=hash\text{-}join(j^{\text{th}}$ sub-tablet of C_{1i} and $C_{2i})$
9. end for
10. $JPIPT_{ij}=MergeSort(JPIPT_{ij})$ $(1\leq j\leq N_{p2})$
11. end for
12. $JPIPT=MergeSort(JPIPT_i)$ $(1\leq i\leq N_{p1})$

经过第二次的划分，C_{1i} 和 C_{2i} 分别划分为两个含有 N_{p2} 个分片的集合. Hash 连接操作在 C_{1i} 和 C_{2i} 的每个分片对执行. 因为当前的 C_{2i} 分片可以放入高速缓存，所以构建和探测 Hash 表的操作都可以在高速缓存中执行. C_{1i} 和 C_{2i} 的第 j 分片对的连接结果是对应第 j 分片对的连接位置索引对表 $JPIPT_{ij}$. 我们在内存中维护 $JPIPT_{ij}$ 以方便接下来的操作.

定理 4. 产生的 $JPIPT_{ij}$ 根据 PI_1 域排序.

证明. $\forall k_1,k_2(k_1<k_2),(PI_1:m_{k1},PI_2:n_{k1})\in JPIPT_{ij},(PI_1:m_{k2},PI_2:n_{k2})\in JPIPT_{ij}$. 假设在 $JPIPT_{ij}$ 中，$(PI_1:m_{k1},PI_2:n_{k1})$ 出现的要早于 $(PI_1:m_{k2},PI_2:n_{k2})$. 这意味着，读取 $T_1(m_{k1})$ 的时间不能晚于读取 $T_1(m_{k2})$ 的时间. 已知在连接操作中，T_1 作为外关系，并且 T_1 的元组按顺序读取来探测 Hash 表，所以 $m_{k1}\leq m_{k2}$. 证毕.

C_{1i} 和 C_{2i} 的连接操作产生 N_{p2} 个连接位置索引对表. 根据定理 1，每个 $JPIPT_{ij}$ 根据 PI_1 域排序. 我们利用一次多路合并排序操作来合并 $JPIPT_{ij}$ $(1\leq j\leq N_{p2})$ 获得 $JPIPT_i$，$JPIPT_i$ 输出到磁盘. 类似的操作在每个 C_{1i} 和 C_{2i} $(1\leq i\leq N_{p1})$ 上执行，获得 N_{p1} 个 $JPIPT_i$ 文件. 此时，再一次执行多路合并排序来合并 $JPIPT_i$ $(1\leq i\leq N_{p1})$ 获得最终的 JPIPT.

4.2 构建 JRS

获得 JPIPT 后，构建 JRS 的操作就比较明显了. 该构建操作类似于 3.3 节的情况 C，只是不需要执行聚集计算. 假设 JRS 包括 n 个元组，算法 2 可以产生 n 个 JPIPT 随机元组. 利用获得的 $JPIPT\text{-}sample$，BUF_T_1 和 $SBUF\_T_2$ 可以采用类似于情况 C 的方式获得. 假设 T_1 有 n_1 个属性，T_2 有 n_2 个属性，JRS 存储为 n_1+n_2 个向量. 每个向量对应于 BUF_T_1 或 $SBUF\_T_2$ 的一个属性.

5 维护 JPIPT 和 JRS

众所周知，read/append-only 是海量数据的一个重要特点，新数据批量地插入已有数据的末尾. 本文利用该特点有效地执行 JPIPT 和 JRS 的更新操作.

5.1 维护 JPIPT

假设更新操作把 ΔT_1 插入到 T_1 的末尾得到 T_1'，把 ΔT_2 插入到 T_2 的末尾得到 T_2'. JPIPT 需要执行更新操作来反映最新的 $T_1'\bowtie T_2'$ 的结果. 如图 2 所示，$T_1'\bowtie T_2'=(T_1\bowtie T_2)\cup(T_1\bowtie\Delta T_2)\cup(\Delta T_1\bowtie T_2)\cup(\Delta T_1\bowtie\Delta T_2)$. $T_1'\bowtie T_2'$ 的连接位置索引对表 $JPIPT'=JPIPT_1\cup JPIPT_2\cup JPIPT_3\cup JPIPT_4$，$JPIPT_1$、$JPIPT_2$、$JPIPT_3$ 和 $JPIPT_4$ 分别表示 $T_1\bowtie T_2$、$T_1\bowtie\Delta T_2$、$\Delta T_1\bowtie T_2$ 和 $\Delta T_1\bowtie\Delta T_2$ 的连接位置索引对表. 由于 ΔT_i 插入到 T_i 的后面，更新操作不影响现有元组的位置索引，即 $JPIPT_1=JPIPT$. $JPIPT_i$ $(2\leq i\leq 4)$ 计算如下：$JPIPT_2=C3\text{-}JPIPT(T_1,\Delta T_2)$，$JPIPT_3=C3\text{-}JPIPT(\Delta T_1,T_2)$，$JPIPT_4=C3\text{-}JPIPT(\Delta T_1,\Delta T_2)$.

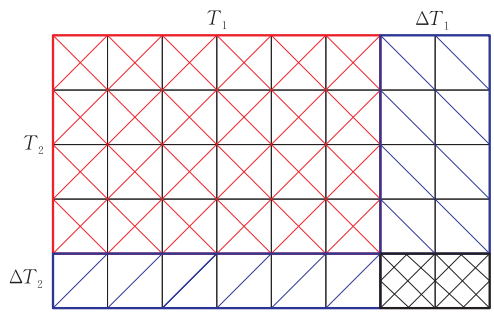


图 2 $T_1'\bowtie T_2'$ 示意图

下面，我们介绍如何有效地实现 $JPIPT'=JPIPT_1\cup JPIPT_2\cup JPIPT_3\cup JPIPT_4$，并且仍

旧保持 $JPIPT'$ 在 PI_1 上的有序性. 由图 2 可知，$JPIPT_1$ 和 $JPIPT_2$ 的 $PI_1 \in [1, |T_1|]$，$JPIPT_3$ 和 $JPIPT_4$ 的 $PI_1 \in [|T_1+1|, |T'_1|]$. 很明显，在 JPIPT′中，$JPIPT_1$ 和 $JPIPT_2$ 的位置在 $JPIPT_3$ 和 $JPIPT_4$ 的前面. 我们可以分别计算 $JPIPT_1 \cup JPIPT_2$ 和 $JPIPT_3 \cup JPIPT_4$，然后把 $JPIPT_3 \cup JPIPT_4$ 顺序添加到 $JPIPT_1 \cup JPIPT_2$ 的后面就得到了 JPIPT′.

$JPIPT_3 \cup JPIPT_4$ 的处理比较简单，因为 $JPIPT_3$ 和 $JPIPT_4$ 的数据量都较小. $JPIPT$ 的维护工作重点在于 $JPIPT_1 \cup JPIPT_2$ 的计算. 常用的合并排序必须扫描和解析每个 $JPIPT_1$ 元组和 $JPIPT_2$ 元组来执行比较操作. 由于历史数据量很大，从而造成 $JPIPT_1$ 的数据量也比较大（在实验中，268GB 的数据文件产生 11GB 的 $JPIPT_1$ 数据），如果采用常用的合并排序来处理 $JPIPT_1 \cup JPIPT_2$ 需要耗费较多的时间. $JPIPT_1$ 和 $JPIPT_2$ 的两个属性可以帮助更快的完成 $JPIPT_1 \cup JPIPT_2$ 的合并操作：(1) $JPIPT_1$ 和 $JPIPT_2$ 都在 PI_1 属性上排序；(2) 和 $JPIPT_1$ 相比，$JPIPT_2$ 的 PI_1 域非常稀疏. 本文提出部分解析的合并排序算法 MSPR(Merge Sort with Partial Resolution)，这里的解析指的是把文件的字节流转换成元组格式的 CPU 操作，该算法充分利用 $JPIPT_1$ 和 $JPIPT_2$ 的两个属性来改进合并操作的性能.

MSPR 算法

MSPR 算法引入一个定义在 $JPIPT_1$ 上的数据结构：位置索引范围桶 PIRB(Positional Index Range Bucket)，其每个元组 $(bid|minVal|maxVal|minOffset|maxOffset)$ 表示每个桶的 PI_1 域的最小值、最大值和对应的 $JPIPT_1$ 文件的偏移量. 除了最后一个桶，每个桶范围都包含相同数量的 JPIPT 元组. bid 表示当前桶的 ID 号，该值不需要实例化，可以根据 PIRB 元素的位置索引获得.

MSPR 比较当前的 $JPIPT_2$ 元组 $jpip$ 和当前的 PIRB 元组 $pirb$. 如果 $jpip.PI_1$ 不大于 $pirb.minVal$，则 $jpip.PI_1$ 肯定不大于 $pirb$ 所指的当前桶范围中的所有 $JPIPT_1$ 元组的 PI_1 值，$jpip$ 可以安全地输出到目的文件. 如果 $jpip.PI_1$ 不小于 $pirb.maxVal$，则 $jpip.PI_1$ 肯定不小于 $pirb$ 所指的当前桶范围中的所有 $JPIPT_1$ 元组的 PI_1 值，$pirb$ 所指的当前桶范围中的所有 $JPIPT_1$ 元组可以安全地输出到目的文件，而且还不需要任何解析操作. 如果 $jpip.PI_1$ 的值在范围 $(pirb.minVal, pirb.maxVal)$，则调用常用的合并操作来处理剩余的 $JPIPT_2$ 元组和 $pirb$ 所指的当前桶范围中的 $JPIPT_1$ 元组. 这样，MSPR 不需要解析所有的 $JPIPT_1$ 元组，从而加快了合并操作的执行. 如图 3 所示，MSPR 节省了桶 2 的解析费用.

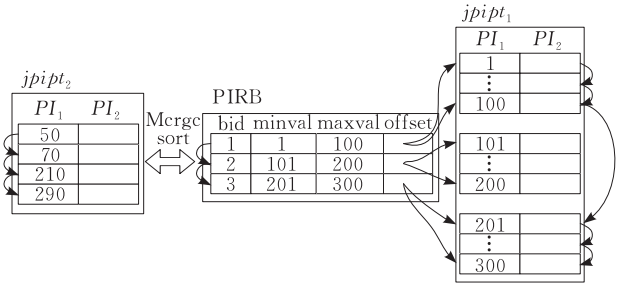


图 3　MSPR 示意图

确定优化的桶大小

PIRB 的桶大小 BS 表示每个桶范围包括的 $JPIPT_1$ 的元组个数. 在 MSPR 的执行过程中，BS 起着十分重要的作用. 如果 BS 比较小，则处理 $JPIPT_1$ 的费用较小，但是构建和处理 PIRB 的费用就较大；如果 BS 比较大，则处理 $JPIPT_1$ 的费用较大，但是构建和处理 PIRB 的费用就较小. 下面，我们介绍如何确定优化的 BS 来最小化 MSPR 的执行费用.

我们用 $cost_p(D)$ 来表示对数据量为 D 的数据的处理费用，S_{JPIPT} 表示 JPIPT 元组的大小，S_{PIRB} 表示 PIRB 元组的大小. 假设 $N_1 = |JPIPT_1|$，$N_2 = |JPIPT_2|$. 为方便讨论，假设 $JPIPT_1$：PI_1 和 $JPIPT_2$：PI_1 的值在 $[1, N_1]$ 内均匀分布. PIRB 需要维护 N_1/BS 个桶记录，在每个桶的范围内包括的 $JPIPT_2$ 的元组个数是 $(N_2 \times BS)/N_1$. 通常，$BS < N_1/N_2$（因为如果 $BS \geq N_1/N_2$ 意味着所有的桶都需要处理），则 $(N_2 \times BS)/N_1 < 1$，$(N_2 \times BS)/N_1$ 表示 MSPR 中需要处理的桶的比例. 如果忽略不需要解析的 $JPIPT_1$ 的处理费用，则 MSPR 的执行费用如下所示：

$$
\begin{aligned}
cost_{MSPR} &= cost_p(JPIPT_1) + \\
&\quad cost_p(JPIPT_2) + cost_p(PIRB) \\
&= cost_p\left(\frac{N_2 \times BS}{N_1} \times \frac{N_1}{BS} \times BS \times S_{JPIPT}\right) + \\
&\quad cost_p(N_2 \times S_{JPIPT}) + 3 \times cost_p\left(\frac{N_1}{BS} \times S_{PIRB}\right) \\
&\geq cost_p(2 \times \sqrt{3 \times S_{JPIPT} \times S_{PIRB} \times N_1 \times N_2} + \\
&\quad N_2 \times S_{JPIPT})
\end{aligned}
$$

当 $N_2 \times BS \times S_{JPIPT} = 3 \times (N_1/BS) \times S_{PIRB}$ 时，$cost_{MSPR}$

取最小值. 所以,优化桶大小 $BS_{opt}=\sqrt{\frac{3\times N_1\times S_{PIRB}}{N_2\times S_{JPIPT}}}$. $3\times cost_p\left(\frac{N_1}{BS}\times S_{PIRB}\right)$包括 PIRB 的构建费用(一次读和写操作)和处理费用(一次读).

5.2 维护 JRS

表的更新操作完成后,JRS 需要执行更新操作使得新的 JRS 是 $T'_1\bowtie T'_2$的连接随机样本. 如图 2 所示,$T'_1\bowtie T'_2=(T_1\bowtie T_2)\cup(T_1\bowtie\Delta T_2)\cup(\Delta T_1\bowtie T_2)\cup(\Delta T_1\bowtie\Delta T_2)$. 我们用 JRS'来表示$T'_1\bowtie T'_2$的连接随机样本. 为保证样本的随机性,JRS'从 $T'_1\bowtie T'_2$的 4 个部分中按比例抽取随机元组. $JRS'=JRS_1\cup JRS_2\cup JRS_3\cup JRS_4$,其中,$JRS_i$ $(1\leq i\leq 4)$包括 $JPIPT_i$对应的$n\times\frac{JPIPT_i}{JPIPT'}$个连接随机元组. 现有 JRS需要丢弃$\left(n-n\times\frac{JPIPT_1}{JPIPT'}\right)$个连接元组,而从 $T'_1\bowtie T'_2$的其它 3 部分中选择并增加相应的新连接元组. 定理 5 证明了 JRS′是更新后的 $T'_1\bowtie T'_2$的大小为 n 的连接随机样本.

定理 5. JRS'是 $T'_1\bowtie T'_2$的大小为 n 的连接随机样本.

证明. 假设 $T'_1\bowtie T'_2=(T_1\bowtie T_2)\cup(T_1\bowtie\Delta T_2)\cup(\Delta T_1\bowtie T_2)\cup(\Delta T_1\bowtie\Delta T_2)=JOIN_1\cup JOIN_2\cup JOIN_3\cup JOIN_4$. 要证明$JRS'$是 $T'_1\bowtie T'_2$的大小为 n 的随机样本,我们只需要证明 JRS′中的每个元组从 $T'_1\bowtie T'_2$中被选中的概率是 $1/|T'_1\bowtie T'_2|$. 给定 JRS′中的任意元组 jrs,jrs 来自$JOIN_i$的概率是$|JOIN_i|/|T'_1\bowtie T'_2|$. 已知,$jrs$ 是 $JOIN_i$的一个随机元组,则其在 $JOIN_i$中被选中的概率是 $1/|JOIN_i|$. 所以,在 $T'_1\bowtie T'_2$中,jrs 被选中的概率是 $\frac{|JOIN_i|}{|T'_1\bowtie T'_2|}\times\frac{1}{|JOIN_i|}=\frac{1}{|T'_1\bowtie T'_2|}$. 证毕.

6 性能评价和分析

我们将通过实验来评价 $p\varepsilon$-AJA 的性能. 实验代码用 Java 语言编写,jdk 版本为 jdk-6u17-Windows-x64,在 HP xw8600 Workstation(8×2.8GHz Xeon CPU+6MB L2 Cache+32GB 内存+1.4TB 硬盘+64bit Windows)上执行. 本文的实验部分比较 $p\varepsilon$-AJA 和在 Oracle 11g 64x 上执行准确查询的性能,而没有同时比较 Join synopsis、在线连接聚集和连接实例化视图,这是由于:(1) Join synopsis 在返回近似结果时所能保证的置信度可能会很低,甚至会低于 1%[18],它的执行时间可能较快,但是如此低置信度的结果是没有什么意义的;(2) 在线连接聚集利用二级随机索引来获得连接表的随机元组,在海量数据环境下,大量的磁盘随机寻道操作使得其操作时间较长,甚至会超过精确查询的时间[19];(3) 在海量数据中,查询涉及到的表通常都很大,所以连接实例化视图也非常大,大大增加了对磁盘空间的需求和维护费用[20],本文实验用到的 268GB 数据表产生的连接实例化视图的大小是 402GB,实验需要大约 3600s 的时间来完成对该实例化视图的扫描操作,其操作时间远超过 $p\varepsilon$-AJA 的执行时间,甚至接近准确查询所需的执行时间. 所以,本文的实验部分选择比较 $p\varepsilon$-AJA 和准确查询的性能.

表 1 实验用到的查询

	查询
Q_A	*select avg(l_discount×l_extendedprice) from lineitem, orders where l_orderkey=o_orderkey*
Q_B	*select avg(o_totalprice) from lineitem, orders where l_orderkey=o_orderkey*
Q_C	*select avg(l_discount×o_totalprice) from lineitem orders where l_orderkey=o_orderkey*

利用 TPC-H,我们生成定长元组的 lineitem 和 orders 表,每个 lineitem 元组大小是 160 字节,每个 orders 元组的大小是 128 字节,1sf 的 lineitem 和 orders 的大小分别是 0.89GB(6000000 条元组)和 0.18GB(1500000 条元组). 在实验中,我们生成 50sf、100sf、150sf、200sf 和 250sf 的数据表. 如表 1 所示,我们用 Q_A、Q_B和 Q_C分别作为情况 A、B 和 C 的查询.

在 Oracle 11g 64x 上执行的准确查询采用 Hash Join 策略,其中 orders 作为内关系表,而 lineitem 作为外关系表. Oracle 先对 orders 和 lineitem 表执行相应划分操作,使得 orders 表的分片可以放入内存,然后依次利用每个 ordes 分片构建 Hash 表,利用 lineitem 分片的元组探测 Hash 表,对满足连接条件的结果元组执行聚集操作,最终获得准确聚集结果.

实验从 3 个方面评价 $p\varepsilon$-AJA 的性能:数据量、信任度和相对误差. 参数设置表 2 所示. 当考虑数据

表 2 实验参数设置

参数	数据量/sf	信任度/%	相对误差/%
默认值	150	99	1
变化范围	50～250	80～99	1～5

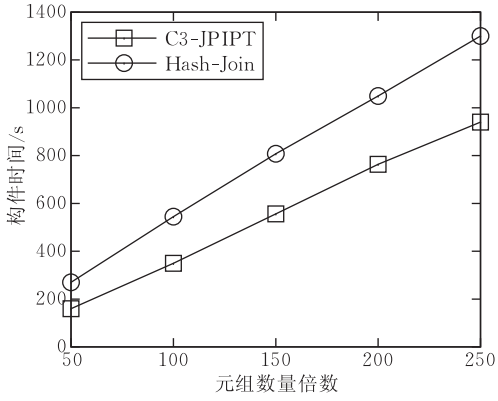
量的效果时，实验评价数据从 50sf(53GB)增长到 250sf(268GB)时 $p\varepsilon$-AJA 的性能；当考虑信任度的效果时，实验评价信任度从 80%增长到 99%时 $p\varepsilon$-AJA 的性能；当考虑相对误差的效果时，实验评价相对误差从 1%增长到 5%时 $p\varepsilon$-AJA 的性能.

6.1　系统初始化

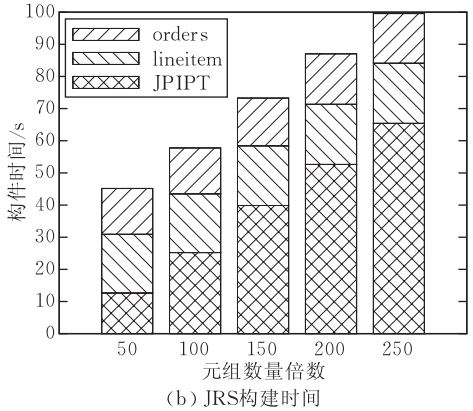
系统通过构建 JPIPT 和 JRS 执行 $p\varepsilon$-AJA 的初始化操作，JPIPT 和 JRS 被反复使用直到下次更新，实验在 5 组数据上运行. 图 4(a)说明了 JPIPT 的构建时间. 当 sf＝50 时，JPIPT 的构建时间是 159.5s，而在 sf＝250 时，JPIPT 的构建时间是 940.25s. JPIPT 的构建时间随着数据量的增加而线形增长. 但是即使对于 250sf(268GB)的数据集，构建时间也不到 1000s. 本文利用 C3-JPIPT 算法构建 JPIPT，C3-JPIPT 虽然执行了两次划分操作(分别在磁盘和内存)，但是它的执行时间仍然比 Hash-Join 快，这也说明了充分利用高速缓存对查询处理的重要性. JRS 的构建时间包括 3 个部分：JPIPT 采样时间、获取 lineitem 元组时间和获取 orders 元组的时间. 如图 4(b)所示，当数据量变化时，从 lineitem 和 orders 中获得元组的时间基本不变，而对 JPIPT 的采样操作的时间则随着数据量的增加而增长. 这是因为，JRS 的大小是固定的(实验设置 JRS 大小为 5000)，从数据文件中获取固定数量的样本元组的时间基本不变. 但是，对于 JPIPT 的采样操作则需要处理 JPIPT 的所有元组，所以它的时间随着数据量的增大而增加. 但是即使在 250sf 的数据集，构建 JRS 的时间仍然不到 100s，该操作费用较低.

6.2　数据量的效果

我们用 5 组数据来评价信任度和相对误差固定而数据量变化情况下 $p\varepsilon$-AJA 的性能. 实验考虑 $p\varepsilon$-AJA 的 3 种查询情况. 当数据量变化时，回答给定置信区间的查询需要的样本点数量基本不变，对 A、B 和 C 三种情况分别是 60000、44000 和 66000 个点. Q_A 涉及 lineitem 的两个属性，其执行时间包括两部分：对 JPIPT 的采样和获取 lineitem 元组. 前者耗费了整个执行时间的大部分. 如图 5(a)所示，$p\varepsilon$-AJA 的执行时间比 Oracle 平均减少了 47.48 倍. 类似的，在情况 B 和 C 下，$p\varepsilon$-AJA 的执行时间比 Oracle 平均减少了 31.35 倍和 35.7 倍. 图 5(b)表示的是近似值相对于精确结果的相对误差. 图 5 中的顶部的横线表示用户给定的相对误差，可以看到，$p\varepsilon$-AJA 获得近似结果满足用户给定的置信区

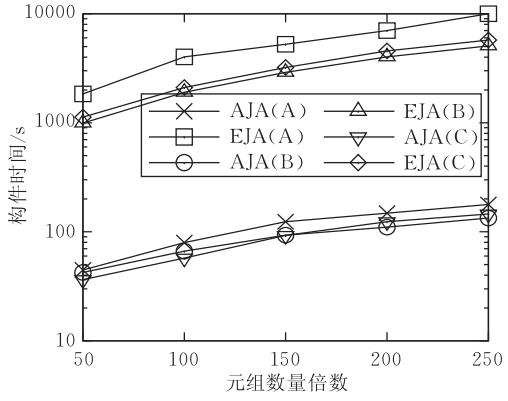


(a) JPIPT构建时间

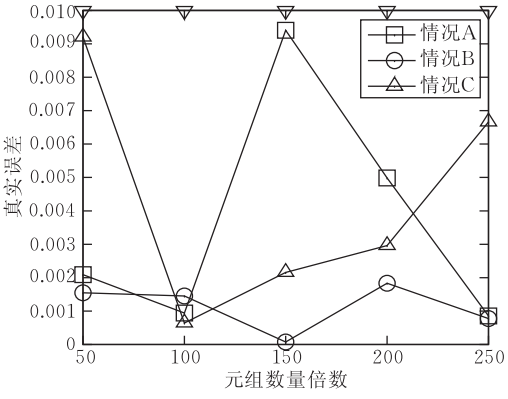


(b) JRS构建时间

图 4　系统初始化



(a) 查询执行时间



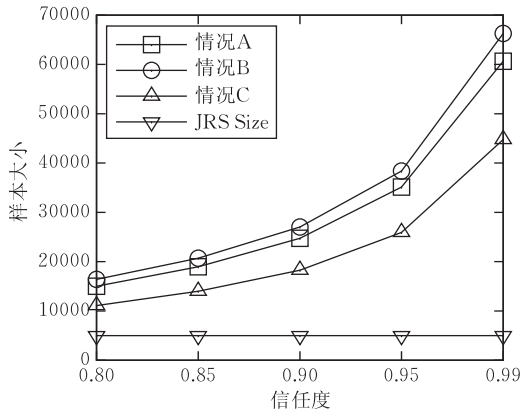
(b) 真实误差

图 5　变化数据量的效果

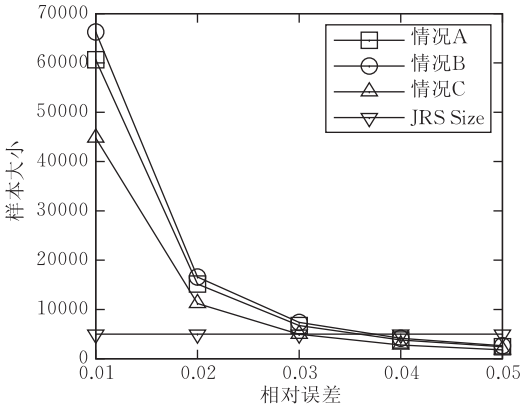
间.这里需要注意的是,海量数据本身是一个相对的术语,而本文的实验以 268G 的数据集合为例来评价 $p\varepsilon$-AJA 的性能.实验结果表明,随着数据量的增大,$p\varepsilon$-AJA 对 Oracle 的查询性能的优势也在增大,所以我们相信,在更大的 TB/PB 级数据量上,$p\varepsilon$-AJA 将获得更好的性能.

6.3 信任度的效果

这组实验评价当数据量和相对误差固定而信任度变化时 $p\varepsilon$-AJA 的性能.如图 6(a)所示,当信任度从 80%增长到 99%,需要的样本数量增加了 4 倍.图 7(a)说明了在信任度变化时 Q_A、Q_B 和 Q_C 的执行时间.因为此时用到的数据量不变,所以准确查询的时间也保持不变,但是 $p\varepsilon$-AJA 的执行时间却随着信任度的提高而逐渐增加.在相对误差和数据量固定的情况下,信任度的增加意味着查询需要更多的样本.所以 $p\varepsilon$-AJA 需要更多的时间来对 JPIPT 采样和获取元组.由于对 JPIPT 的采样操作需要处理 JPIPT 的所有元组,而获取元组操作只需要涉及到数据文件的很小一部分,所以对 JPIPT 的采样操作耗费了整个查询操作的大部分时间.更高的信任度引起的采样操作的时间增量相对较小,这是因为更多的样本只是需要在对 JPIPT 的扫描过程中选择更多的 JPIPT 元组.相对于精确查询,$p\varepsilon$-AJA 在 3 种情况下获得加速比分别为 47.12、34.61 和 38.7.如图 7(b)所示,获得近似结果满足给定的置信区间.

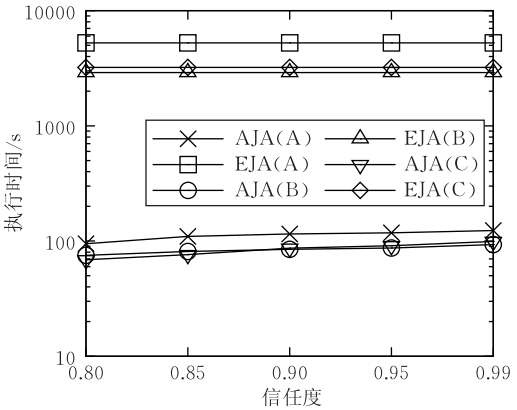


(a) 样本大小(信任度变化)

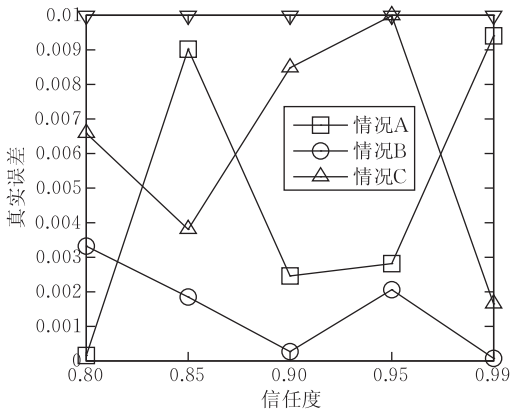


(b) 样本大小(相对误差变化)

图 6 样本大小



(a) 查询执行时间



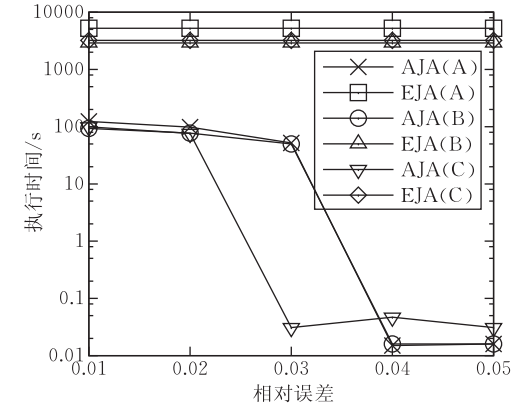
(b) 真实误差

图 7 变化信任度的效果

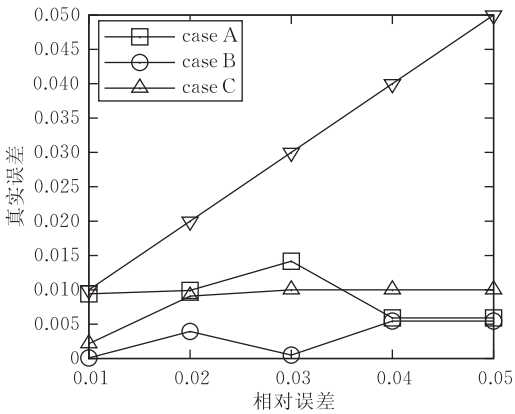
6.4 相对误差的效果

这组实验评价当数据量和信任度固定而相对误差变化时 $p\varepsilon$-AJA 的性能.如图 6(b)所示,当相对误差从 1%增加到 5%时,所需样本数量减少了 25 倍.图 6(b)中底部的横线是 $p\varepsilon$-AJA 的 JRS 大小.如图 6 所示,表示 JRS 大小的横线和情况 A 与 B 需要的样本大小在(3%,4%)区间内相交,而和情况 C 需要的样本大小在 3%左右相交.如图 8(a)所示,$p\varepsilon$-AJA 的执行时间随着相对误差的增加而降低.尤其是在 3%和 4%之间,执行时间从 50s 降到了 0.01s.在这组实验中,$p\varepsilon$-AJA 的执行时间平均比准确查询减少了 5 个数量级.这是由于,如果内存中维护的 JRS 足够返回满足置信区间的近似结果,则就不需要执行磁盘操作.如图 8(b)所示,$p\varepsilon$-AJA 的近似结果满足给定的置信区间,并且我们注意到,在情况 C 中,在 3%以后的相对误差以及情况 A 和 B 中,4%以后的相对误差保持不变.这是因为,此时 JRS 足

够返回满足用户要求的近似结果，从而提供了固定的置信区间.



(a) 查询执行时间

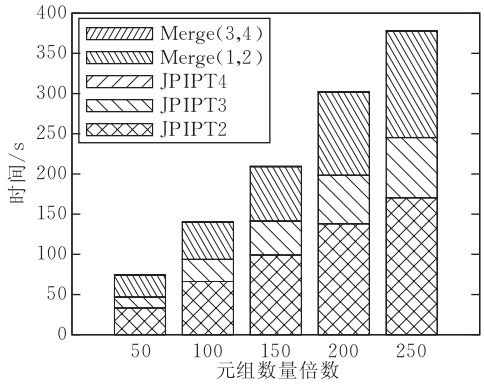


(b) 真实误差

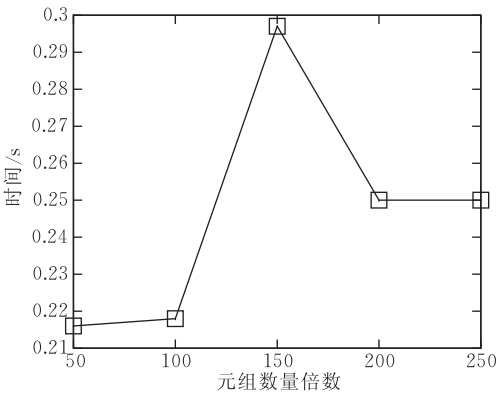
图 8　变化相对误差的效果

6.5　更新处理

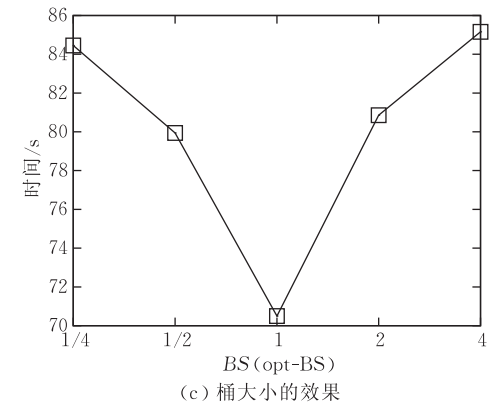
这组实验评价 JPIPT 和 JRS 的更新操作性能. 我们生成 1000000 条 lineitem 元组和 250000 条 orders 元组. 考虑到时间局部性，新生成的数据中 20%的连接属性数据落入已有连接属性数据的范围，而剩余的 80%的数据则不在已有连接属性的数据范围. 图 9(a)和(b)说明了 JPIPT 和 JRS 的更新时间. JPIPT 的更新主要包括 3 个部分：$JPIPT_2$ 构建、$JPIPT_3$ 构建和归并 $JPIPT_1$ 和 $JPIPT_2$，构建 $JPIPT_4$ 以及归并 $JPIPT_3$ 和 $JPIPT_4$ 的时间比较小，所以在图中不明显. 随着数据量的增加，更新 JPIPT 的时间也随着增长. 但是更新操作所需要的时间较快，即使在最大的 268G 数据集，更新 JPIPT 的时间也不到 380s. JRS 的更新操作更快，如图 9(b)所示，只需要仅仅 0.2s～0.3s 的时间. 这是因为，已有数据相对于新数据是非常大的，所以 JRS 只需要更新很小部分元组. 图 9(c)说明了优化桶大小的效果，使用的数据集是 150sf，x 轴表示使用的桶大小和优化桶大小的比值. 如果 BS 比较小，则处理 $JPIPT_1$ 的费用较小，但是构建和处理 PIRB 的费用就较大；如果 BS 比较大，则处理 $JPIPT_1$ 的费用较大，但是构建和处理 PIRB 的费用就较小. 如



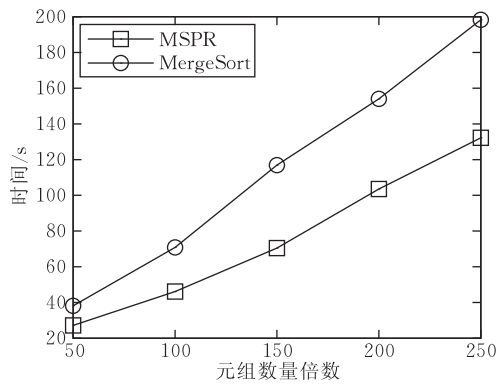
(a) JPIPT更新时间



(b) JRS更新时间



(c) 桶大小的效果



(d) MSPR和合并排序的比较

图 9　更新处理

图 9(c),桶大小的最优折衷在 B_{opt},这符合我们之前对优化桶大小的分析.利用 PIRB,如图 9(d),由于更少的解析费用,MSPR 的执行比普通的合并操作快.

7 结 论

本文提出一种新的近似连接聚集算法 $p\varepsilon$-AJA,该算法可以有效地返回满足任意置信区间的近似结果,$p\varepsilon$-AJA 利用预计算的连接随机样本 JRS 和连接位置索引对表 JPIPT 处理查询.$p\varepsilon$-AJA 首先利用 JRS 来给出近似结果的快速响应.如果利用 JRS 的结果不满足给定的置信区间,$p\varepsilon$-AJA 利用 JPIPT 获得更多的连接随机元组来计算结果.本文提出一种新的采样算法来获得 JPIPT 给定数量的样本.利用获得的 JPIPT 样本,本文提出元组抽取算法通过对连接表的一遍顺序扫描获得连接元组.本文提出了 JRS 和 JPIPT 的有效的构建和维护算法.实验结果表明:$p\varepsilon$-AJA 可以有效获得满足任意置信区间的近似结果,并且获得相对于准确查询 1～5 个数量级的加速.对于 1250000 条元组插入 268G 数据的更新操作,$p\varepsilon$-AJA 只需要不到 380s 的时间来完成 JPIPT 和 JRS 的维护操作.

参 考 文 献

[1] Acharya Swarup, Gibbons Phillip, Poosala Viswanath, Ramaswamy Sridhar. Join synopses for approximate query answering//Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD'99). Philadelphia, Pennsylvania, USA, ACM, 1999: 275-286

[2] Hass Peter, Hellerstein Joseph. Ripple joins for online aggregation//Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD'99). Philadelphia, Pennsylvania, USA, ACM, 1999: 287-298

[3] Luo Gang, Ellmann Curt, Haas Peter, Naughton Jeffrey. A scalable hash ripple join algorithm//Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02). Madison, Wisconsin, USA, 2002: 252-262

[4] Jermaine Christopher, Dobra Alin, Arumugam Subramanian, Joshi Shantanu, Pol Abhijit. A disk-based join with probabilistic guarantees//Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD'05). Baltimore, Maryland, USA, 2005: 563-574

[5] Stonebraker Mike, Abadi Daniel, Batkin Adam et al. C-Store: A column-oriented DBMS//Proceedings of the 31st International Conference on Very Large Data Bases (VLDB'05). Trondheim, Norway, 2005: 553-564

[6] Hellerstein Joseph, Hass Peter, Wang Helen. Online aggregation//Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD'97). Tucson, Arizona, USA, ACM, 1997: 171-182

[7] Cheng Siyao, Li Jianzhong, Ren Qianqian, Yu Lei. Bernoulli sampling based (epsilon, delta)-approximate aggregation in large-scale sensor networks//Proceedings of the 29th IEEE International Conference on Computer Communications (INFOCOM'10). San Diego, CA, USA, IEEE, 2010: 1181-1189

[8] Wu Sai, Jiang Shouxu, Ooi Beng Chin, Tan Kian-Lee. Distributed online aggregation//Proceedings of the 35th International Conference on Very Large Data Bases (VLDB'09). Lyon, France, VLDB Endowment, 2009: 443-454

[9] Hass Peter. Large-sample and deterministic confidence intervals for online aggregation//Proceedings of the 9th International Conference on Scientific and Statistical Database Management (SSDBM'97). Olympia, Washington, USA: IEEE Computer Society, 1997: 51-63

[10] Spiegel Joshua, Polyzotis Neoklis. Tug synopses for approximate query answering. ACM Transactions on Database Systems, 2009, 34(1): 3

[11] Olken Frank, Rotem Doron. Simple random sampling from relational databases//Proceedings of the 12th International Conference on Very Large Data Bases (VLDB'86). Kyoto, Japan, Proceedings, Morgan Kaufmann, 1986: 160-169

[12] Chaudhuri Surajit, Das Gautam, Narasayya Vivek. Optimized stratified sampling for approximate query processing. ACM Transactions on Database Systems, 2007, 32(2): 9

[13] Haas Peter, Konig Christian. A bi-level Bernoulli scheme for database sampling//Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data(SIGMOD'04). Paris, France, ACM, 2004: 275-286

[14] Gemulla Rainer, Lehner Wolfgang, Haas Peter. A dip in the reservoir: Maintaining sample synopses of evolving datasets//Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB'06). Seoul, Korea, 2006: 595-606

[15] Chaudhuri Surajit, Motwani Rajeev, Narasayya Vivek. On random sampling over joins//Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD'99). Philadelphia, Pennsylvania, USA, ACM, 1999: 263-274

[16] Han X, Li J, Yang D. PI-Join: Efficiently processing join query on massive data. Harbin Institute of Technology: Technical Report tr1004-DKERC, 2010

[17] Boncz Peter, Manegold Stefan, Kersten Martin. Database architecture optimized for the new bottleneck: Memory access//Proceedings of the 25th International Conference on Very Large Data Bases (VLDB'99). Edinburgh, Scotland, UK, Morgan Kaufmann, 1999: 54-65

[18] Abhijit Pol, Christopher M. Jermaine and subramanian arumugam, maintaining very large random samples using the ge-

ometric file. The VLDB Journal, 2008, 17(5): 997-1018

[19] Chen P M, Lee E L, Gibson G A et al. RAID: High-performance, reliable secondary storage. ACM Computing Surveys, 1994, 26(2): 145-185

[20] Ashish Gupta, Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques and applications. IEEE Data Engineering Bulletin, 1995, 18(2): 3-18

HAN Xi-Xian, born in 1981, Ph. D. candidate. His research interests include massive data processing and data intensive computing.

YANG Dong-Hua, born in 1976, Ph. D., lecturer. His research interests include massive data processing and data intensive computing.

LI Jian-Zhong, born in 1950, professor, Ph. D. supervisor. His research interests include database and sensor network.

Background

This paper focuses on the research for query processing issues on massive data. Approximate join aggregate is an important operation because it takes much less time to return approximate results with some confidence interval. Researchers have proposed some methods to deal with approximate join aggregate. Some researchers propose to make use of precomputed join synopsis to answer approximate join aggregate. Its execution is fast but it can only provide fixed confidence interval for its fixed-size join synopsis. Online join aggregation is another kind of approximate join aggregate method. This method computes join aggregate results online and return results if given confidence interval is satisfied. But the demand of random tuple retrieval makes it nearly impossible to perform efficiently since large amount of disk random seek operation degrades efficiency significantly. This paper proposes a novel algorithm $p\varepsilon$-AJA to perform approximate join aggregate with arbitrary confidence interval. $p\varepsilon$-AJA precomputes and makes use of two data structure to efficiently answer approximate join aggregate: JRS and JPIPT. $p\varepsilon$-AJA first returns a quick response of approximate result to users by JRS, if the given confidence interval is not satisfied, JPIPT is used to retrieve more join random tuples. A novel sampling method is presented here to obtain samples of fixed size. The main contribution of this paper is to propose a method to perform approximate join aggregate without expensive ad-hoc join operation. The construction and maintenance of JRS and JPIPT is also presented in this paper.

This work is supported in part by the National Grand Fundamental Research 973 Program of China, the Key Program of National Natural Science Foundation of China, the NSF of China and the NSFC-RGC of China. These foundations focus on the research of various areas of data intensive super computing. Our group has been working on the research of database for many years, and many good papers have been published in worldwide conferences and transactions, such as SIGMOD, VLDB, ICDE, KDD, INFOCOM, TKDE, The VLDB Journal et al. This paper proposes an efficient approximate join aggregate algorithm. It gives a solution on topic of query processing on massive data.