软件学报 ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*,2023,34(6):2804−2832 [doi: 10.13328/j.cnki.jos.006496]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

# 基于谱聚类的在线数据库垂直分区多阶段生成方法*

刘鹏举[1,2], 李好洋[1,2], 王天一[1,2], 刘 欢[1,2], 孙路明[1,2], 任逸飞[3], 李翠平[1,2], 陈 红[1,2]

[1](数据工程与知识工程教育部重点实验室 (中国人民大学), 北京 100872)

[2](中国人民大学 信息学院, 北京 100872)

[3](华为云数据库创新 Lab, 广东 深圳 518100)

通信作者: 李翠平, E-mail: licuiping@ruc.edu.cn

**摘 要**: 垂直数据分区技术从逻辑上将满足一定语义条件的数据库表属性存放在同一个物理块中, 进而降低数据访问成本, 提高查询效率. 数据库查询负载中的每条查询通常只与数据库表中的部分属性有关, 因此只需使用数据库表的某个属性子集便可以得到准确的查询结果. 合理的垂直数据分区方式可以使大多数查询负载不需要扫描完整数据库就可以完成查询任务, 从而达到减少数据访问量, 提高查询处理效率的目的. 传统的数据库垂直分区方法主要基于专家设置的启发式规则, 分区策略粒度较粗, 且不能根据负载的特征进行有针对性的分区优化. 同时, 当负载规模较大或者属性个数较多时, 现有垂直分区方法执行时间过长, 尤其无法满足数据库在线实时调优的性能需求. 为此, 提出在线环境下基于谱聚类的垂直数据分区方法 (spectral clustering based vertical partitioning, SCVP), 采用分阶段求解的思想, 减少算法时间复杂度, 加快分区执行速度. 首先通过增加约束条件缩小解空间 (即根据谱聚类生成初始分区), 然后对解空间设计算法进行精细的搜索 (即采用频繁项集和贪心搜索相结合的策略对初始分区进行优化). 为了进一步提升 SCVP 在高维属性下的性能, 提出了 SCVP 的改进版本 SCVP-R (spectral clustering based vertical partitioning redesign). SCVP-R 通过引入同域竞争机制、双败淘汰机制和循环机制, 对 SCVP 在分区优化过程中的合并方案进行了进一步优化. 在不同数据集上的实验结果表明, 相比于目前最好的垂直分区方法, SCVP 和 SCVP-R 有着更快的执行时间和更好的性能表现.

**关键词**: 垂直分区; 谱聚类; 频繁项集; 贪心搜索; 多阶段决策

**中图法分类号**: TP311

中文引用格式: 刘鹏举, 李好洋, 王天一, 刘欢, 孙路明, 任逸飞, 李翠平, 陈红. 基于谱聚类的在线数据库垂直分区多阶段生成方法. 软件学报, 2023, 34(6): 2804–2832. http://www.jos.org.cn/1000-9825/6496.htm

英文引用格式: Liu PJ, Li HY, Wang TY, Liu H, Sun LM, Shen YF, Li CP, Chen H. Multi-stage Method for Online Vertical Data Partitioning Based on Spectral Clustering. Ruan Jian Xue Bao/Journal of Software, 2023, 34(6): 2804–2832 (in Chinese). http://www.jos.org.cn/1000-9825/6496.htm

## Multi-stage Method for Online Vertical Data Partitioning Based on Spectral Clustering

LIU Peng-Ju[1,2], LI Hao-Yang[1,2], WANG Tian-Yi[1,2], LIU Huan[1,2], SUN Lu-Ming[1,2], REN Yi-Fei[3], LI Cui-Ping[1,2], CHEN Hong[1,2]

[1](Key Laboratory of Data Engineering and Knowledge Engineering of the Ministry of Education (Renmin University of China), Beijing 100872, China)

[2](School of Information, Renmin University of China, Beijing 100872, China)

[3](Huawei Cloud Database Innovation Lab, Shenzhen 518100, China)

* 基金项目: 国家重点研发计划 (2018YFB1004401); 国家自然科学基金 (61772537, 61772536, 62072460, 62076245, 62172424)
收稿时间: 2021-06-25; 修改时间: 2021-08-21; 采用时间: 2021-09-25; jos 在线出版时间: 2022-11-16
CNKI 网络首发时间: 2022-11-18

**Abstract**: Vertical data partitioning technology logically stores database table attributes satisfying certain semantic conditions in the same physical block, so as to reduce the cost of data accessing and improve the efficiency of query processing. Every query is usually related only to the table's some attributes in the database, so a subset of the table's attributes can be used to get the accurate query results. Reasonable vertical data partitioning can make most queries answered without scanning the whole table, so as to reduce the amount of data accessing and improve the efficiency of query processing. Traditional database vertical partitioning methods are mainly based on heuristic rules set by experts. The granularity of partitioning is coarse, and it can not provide different partition optimizations according to the characteristics of workload. Besides, when the scale of workload or the number of attributes becomes large, the execution time of the existing methods are too long to meet the performance requirements of online real-time tuning of database. Therefore, a method called spectral clustering based vertical partitioning (SCVP) is proposed for the online environment. The idea of phased solution is adapted to reduce the time complexity of the algorithm and speed up partitioning. Firstly, SCVP reduces the solution space by increasing the constraint conditions, that is, generating initial partitions by spectral clustering. Secondly, SCVP designs the algorithm to search solution space, that is, the initial partitions are optimized by combining frequent itemset mining and greedy search. In order to further improve the performance of SCVP under high-dimensional attributes, a new method called special clustering based vertical partitioning redesign (SCVP-R) is proposed which is an improved version of SCVP. SCVP-R optimizes the partitions combiner component of SCVP by introducing sympatric-competition mechanism, double-elimination mechanism, and loop mechanism. The experimental results on different datasets show that SCVP and SCVP-R have faster execution time and better performance than the current state-of-the-art vertical partitioning method.

**Key words**: vertical partitioning; spectral clustering; frequent itemsets; greedy search; multistage decision

垂直分区是数据库物理设计中的经典技术. 它将一个表或多个表纵向切分成不相交的多个列组, 每个列组拥有相同数量的元组. 其形式化定义为: 给定关系表$R$的属性集合$A=\{a_1,\ldots,a_n\}$、某时间段$t$内的查询工作负载$WD_t=\{q_1,\ldots,q_m\}$和一个成本函数$\mathbb{C}$, 寻找一个最优的垂直分区方案$P'=\{P_1,\ldots,P_k\}$, $k\leqslant n$, 使得数据库的查询执行成本最低, 即:

$$P'=\text{argmin}_P\mathbb{C}(WD_t,P) \tag{1}$$

例如, 给定某关系表$R(a_0,a_1,a_2,a_3,a_4)$和$R$上的两个查询:

$Q_1$: SELECT $a_2,a_3$ FROM $R$;

$Q_2$: SELECT $a_0,a_1,a_4$ FROM $R$.

查询$Q_1$和$Q_2$需要分别在属性$a_2$、$a_3$和$a_0$、$a_1$、$a_4$上执行投影操作. 未采取任何垂直分区策略时, 执行$Q_1$、$Q_2$需要访问$R$的全部属性$a_0$–$a_4$; 若将表$R$分为$P_1=(a_2,a_3)$和$P_2=(a_0,a_1,a_4)$两个分区, 则$Q_1$只需要访问分区$P_1$中的$a_2$、$a_3$属性, 而$Q_2$只需要访问分区$P_2$的$a_0$、$a_1$、$a_4$属性便可完成查询操作. 可以看出, 采用合适的垂直分区策略可以减少数据访问的代价, 提高查询处理的性能.

求解最优垂直分区方案的最直接方法是暴力枚举法, 即枚举所有可能的分区方案, 并为每个方案计算执行所有工作负载的预期访问成本, 选择成本最小的方案. Bell 数是指使用暴力枚举的方式求出垂直分区所有可能组合数. 对于含有$n$个属性关系表, 其对应的 Bell 数$B_n$能够通过公式$B_{n+1}=\sum_{k=0}^{n}\binom{n}{k}B_k$得到, 其中$\left\{ {n \atop n} \right\}=\left\{ {n \atop 1} \right\}=1$, $\left\{ {n \atop k} \right\}=\left\{ {n-1 \atop k-1} \right\}+k\cdot\left\{ {n-1 \atop k} \right\}$, $B_0=1$. 以 TPC-H 数据集中含有 8 个属性的 customer 表为例, 其垂直分区的可能方案共有$B_8=4140$个.

暴力枚举法的时间复杂度为$\mathrm{O}(n^n)$, 显然非常低效. 为此, 人们设计了多种高效的垂直分区方法. 目前垂直分区领域性能最好的算法是 HillsClimb[1]. 它将单列作为初始分区方案, 每次选择将使总 I/O 成本降低最多的两个分区合并, 在此基础上不断迭代, 直到 I/O 成本不再减少. HillsClimb 算法通过自底向上的迭代过程寻找较优解, 每次迭代, 都在上一轮迭代的方案下进行决策, 虽然不能保证全局下的最优解, 但避免了暴力枚举的复杂性. HillsClimb 的时间复杂度为$\mathrm{O}\left(m^3nt\right)$, 其中$m$为表的属性数, $n$为查询负载规模, $t$为迭代次数. 它在低维属性表和轻量负载下表现优异, 但一旦属性维度或负载规模提高时, 其算法执行时间便呈指数级和高斜率线性增长, 经实验验证, 面对 200 维属性的表、2 000 条查询规模的工作负载, 其平均执行时间高达 2 901 s.

在线分区系统[2−5]是一种根据工作负载特征变化的显著程度, 决定是否调用分区算法进行在线的分区部署, 它的目标是保证数据库系统在各个时间段都能采用合适的分区策略, 保持最佳的性能, 因此它对分区算法的执行时

间有着极高的要求. 显然, HillsClimb 更不适合于在线分区.

本文提出一个具有较低时间复杂度的垂直分区方法 SCVP. 首先根据工作负载, 提取属性的亲和度矩阵, 生成表属性的关系图并在该关系图上进行谱聚类, 得到属性聚簇, 作为初步的分区方案; 然后对初始方案进一步优化, 包括分区的分割和合并两个过程, 首先对工作负载进行频繁项集挖掘, 利用得到的频繁模式作为先验信息, 将属性聚簇拆分成最小粒度的分区单元, 再根据先验信息, 将分区单元进行组合, 形成最终的分区方案并在数据库中部署. 同时本文也提出新的成本模型 HDD-I (hard disk imputed cost model) 和 SCVP 的改进算法 SCVP-R, 该算法主要对 SCVP 的合并过程进行优化, 使性能发挥到最佳. 本文提出的方法将分区求解过程划分为两个优化阶段, 以减少每个阶段所面临的解空间, 因此更适用于高维属性; 其次, 由于第 1 阶段只利用了亲和度的普适性, 仅在第 2 阶段需要计算负载的执行成本, 从而减少了计算分区方案成本的总次数, 更适合于高负载. 实验证明, 在 8 个不同属性维度的数据集测试, SCVP 和 SCVP-R 分别达到 HillsClimb 性能的 92%、100.5%, 在执行时间上仅为 HillsClimb 的 1/226、1/17. 在 8 个不同负载规模的数据集中测试, SCVP 和 SCVP-R 分别达到 HillsClimb 性能的 97.6% 和 100%, 执行时间缩减到 1/35、1/16.

本文第 1 节介绍了垂直分区的相关研究工作. 第 2 节介绍了 SCVP、SCVP-R 算法的原理. 第 3 节按照分区特性进行分类, 并给出了成本模型 HDD-I. 第 4 节提出一种切割收益函数对 SCVP 和 SCVP-R 的执行时间进行优化. 第 5 节进行实验分析. 第 6 节对全文进行总结.

## 1 相关研究

从 1975 年发展至今, 数据库垂直分区技术的研究可从两个角度进行归纳: (1) 根据评价指标的演变将垂直分区研究划分为基于亲和度矩阵和基于成本模型两个阶段: 早期基于亲和度矩阵的方法认为如果将尽可能多的被共同访问的属性放在同一分区中, 这种划分就是有效的; 近年来的研究则提出采用模拟数据库实际环境的成本模型, 来作为评价分区方案优劣的标准. (2) 根据分区的生成策略, 可分为自底向上、自顶向下、基于数据挖掘技术的垂直分区生成方法这 3 类. 本文主要从分区生成策略的角度对垂直分区研究进行介绍.

(1) 自底向上的垂直分区生成方法

对暴力枚举法的一种改进思路是采用自底向上的方法来生成垂直分区. 算法开始时, 首先得到一个初始化分区, 随后逐步迭代, 在每一次迭代中分区数减少一个或多个, 并设置停止条件. 例如 HillsClimb 就是典型的从单列布局开始自底向上迭代的算法.

AutoPart[6]于 2004 年提出. 首先, 它基于选择谓词对表进行水平分区, 使每个分区都由不同的查询子集访问. 然后为每个水平分区进行垂直划分. 首先, AutoPart 生成一组主分区 (称为原子分区), 如果所有访问它的查询都引用分区中的所有属性, 则该分区具有原子性, 即不存在访问原子分区子集的查询. 此后, 在每次迭代中, 分区可以与原子分区或上一次迭代过的分区相结合, 直到查询工作负载的估计成本没有改善时, 迭代结束.

Hyrise[7]是 2010 年提出的用于内存数据库的垂直分区算法. 第 1 步, 它会生成一组主分区, 等同于 AutoPart 中的原子分区, 在此基础上建立主分区的亲和图, 其中主分区表示为图中的节点, 两个主分区的共同访问频率表示为边权重. 然后使用 Kmetis 分割器[8]对图进行切割, 使得切割后的每个子图最多包含 $K$ 个主分区 ($K$ 为常数). 第 2 步, Hyrise 分别求出每个子图内的最佳分区方案. 第 3 步, 进行主分区间的合并过程, 在每一次迭代中, 选择最大成本改进的两个主分区合并, 替换主分区, 重复迭代过程直到成本不再有任何改进为止.

Trojan[9]是于 2011 年提出的一种基于阈值剪枝的算法. 第 1 步, 它枚举所有可能的列组 (即分区), 采用列组兴趣度的度量方法, 只保留兴趣度较高的列组, 对兴趣度低于某一阈值的列组进行剪枝, 然后将分区问题转化为 0-1 背包问题, 最终把剪枝后的列组合并为若干个完整 (即包含所有属性) 且不相交的分区集合.

(2) 自顶向下的垂直分区生成方法

对暴力枚举法的另一种改进思路是采用自顶向下的方法把整个表作为单个分区, 随后逐步迭代, 在每一次迭代中分区数增加一个或多个, 并设置停止条件.

Navathe 等人[10]最早提出了一种基于亲和度的垂直分区方法, 根据给定一组属性和引用这些属性的查询, 构

造一个属性亲和度矩阵, 矩阵中的单元格 $(i, j)$ 表示属性 $i$ 与属性 $j$ 共出现的次数 (即属性间的亲和性), 然后使用键能算法 BEA[11]进行矩阵聚类, 最后利用一些规则将聚集的每个属性聚簇递归地拆分为更细粒度的分区.

Navathe 等人[12]随后提出使用图结构解决垂直分区问题, 将属性对的亲和度作为结点间的边构建无向带权图, 然后设计启发式的规则搜索闭合的"圈"作为一个分区.

O2P[13]对 Navathe 等人[10,12]的算法改进, 将其转化为一个时间复杂度较小并适合在线环境的垂直分区算法. 根据负载增加情况, 动态更新查询的关联矩阵, 使用 BEA 算法对其聚类, 为了得到最佳的垂直分区, 使用动态规划来记住上一步中非最佳垂直分区的成本, 然后 O2P 采用贪婪的方法在每个步骤中创建一个最好的垂直分区.

(3) 基于数据挖掘技术的垂直分区生成方法

Hoffer[14]根据关系表中属性间的亲和度, 使用 BEA 对属性进行聚类. Cornell 等人[15]基于整数规划技术解决垂直分区问题, 通过设计一种最优的二进制递归分区算法来最小化磁盘访问成本. Cheng 等人[16]、Song 等人[17]和 Ng 等人[18]都是用遗传算法来解决垂直分区问题. Gorla 等人[19]首次提出使用 Apriori 算法[20]解决垂直分区问题, 通过设置合理的最小支持度, 产生原表的频繁项集, 然后按照支持度大小从高到低依次选择不重复的频繁模式, 作为候选分区方案, 最后根据成本模型找到最佳的分区方案. Rodriguez 等人[21]提出一种 CBPA 算法, 将频繁项集的支持度概念引入到亲和度矩阵, 将矩阵中的所有亲和度值作为数据集, 保留前 30% 频繁出现的亲和度作为属性的支持边, 依次遍历每个属性, 如果该属性未分配到某个分区, 则将包含该属性的所有支持边作为一个分区. Huang 等人[22]也使用了 Apriori 算法, 在项集组合成候选分区的思路和 Gorla 等人[19]一致, 只是在生成频繁项集时, 使用频繁模式的余弦相似度代替频数作为频繁模式的权重. Arulraj 等人[23]提出一种在线的列组划分方法, 主要分为两个阶段: ① 定义两个查询之间的距离: 只在一个查询中访问的属性数/表中的属性数, 并使用 K-means[24]确定哪些查询应该存放在同一聚簇中. ② 根据阶段一得到的每个聚簇的代表性查询语句, 将每个聚簇的权重降序排列, 利用贪心算法将每个聚簇所访问的属性划分到同一个分区中.

上述是垂直分区的研究现状, 整体来看, 算法以何种角度、分几个阶段生成分区其实并不决定性能优劣, 关键在算法逼近于暴力枚举法的程度与迭代复杂度之间的权衡. 文献 [1,6,7,9] 在得到主分区的基础上, 使用贪心策略逐步合并或转化为规划问题得到更优的分区方案, 首先主分区的合理性对后续的结果有极大影响, 上述研究并未探讨这一问题, 其次是合并策略迭代时间久; 文献 [10,12,13] 只依赖于亲和度矩阵, 而此矩阵不能反映多个属性间的亲和度, 分区结果必然存在瓶颈; 文献 [15−17,19,22] 对属性亲和度等指标挖掘出候选分区后, 再根据成本模型选择最佳分区, 没有将成本模型与数据挖掘技术直接结合. 由此看出, 现有技术往往没有兼顾到多个方面, 且认为分区生成方式必须是单一方向的, 实际上, 它们不仅可以通过改进瓶颈环节, 还能互相融合来提升性能.

本文提出的 SCVP 和 SCVP-R 方法解决了现有不足, 具体如下: (1) 在切割过程中使用了自顶向下的分区生成方法, 使用图聚类方法进行分区初始化, 然后依据频繁模式进行更细致的分区修剪, 且目标并非是试图找到最终的分区方案, 而是得到该成本模型环境下, 粒度最小的分区单元, 便于多阶段处理的衔接. (2) 在分区单元的合并过程中使用了自底向上的分区生成方法, 利用频繁模式指导合并, 而不是随机组合, 然后通过贪心策略确定分区单元合并的顺序, 通过降低组合空间最大程度减少迭代时间.

## 2 基于谱聚类的垂直分区方法 SCVP

为了应对高维属性的复杂组合空间, 同时减少对成本模型的调用次数, 本文提出两阶段的垂直分区求解方法 SCVP. 第 1 阶段, 从负载流中提取属性结点的亲和度矩阵, 使用谱聚类提取各个属性结点的特征向量, 经特征向量得到的聚簇结果作为初始分区方案. 第 2 阶段, 在初始方案的基础上依据成本模型 HDD-I (见第 3.2 节), 通过切割和合并两个修剪过程, 得到最终的垂直分区方案.

### 2.1 SCVP 模型的基本思想

SCVP 算法的核心思想是通过分阶段求解, 减少算法时间复杂度, 加快分区执行速度. 首先通过增加约束条件缩小解空间 (即根据谱聚类生成初始分区), 然后对解空间设计算法进行精细的搜索 (即采用频繁项集和贪心搜索相结合的策略对初始分区进行优化). SCVP 算法的整体架构如图 1 所示, 主要包含 4 个组件.

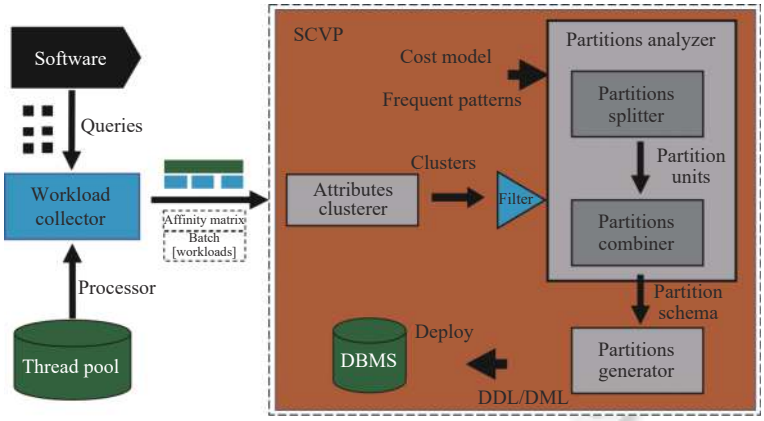



图 1 SCVP 架构

(1) 负载收集器 (workload collector): 模型外部组件, 用于接收软件系统用户端的负载信息流, 将其处理为批量 batch 数据, 这些 batch 将会分配到线程池的 processor 类, 待软件系统发送分区信号后, processor 生成亲和度矩阵和负载数据发送到属性聚类器.

(2) 属性聚类器 (attributes clusterer): 将亲和度矩阵转化为属性的无向带权图, 对图进行切割得到聚簇, 作为初始分区方案交由优化器处理.

(3) 分区优化器 (partitions analyzer): 依据负载数据生成的频繁模式作为先验信息, 使用测量分区方案性能的成本模型作为指导信息, 对初始分区方案分为两个阶段处理. 第 1 阶段由分区切割器 (partitions splitter) 完成, 使用自顶向下的策略对初始分区方案进行切割, 生成分区单元; 第 2 阶段由分区聚合器 (partitions combiner) 完成, 使用自底向上的策略对分区单元合并, 形成最终分区方案.

(4) 分区生成器 (partitions generator): 根据分区优化器产生的最终分区方案生成 DDL/DML 语句部署到 DBMS 中.

### 2.2 SCVP 算法的组件介绍

本节对 SCVP 模型的 4 个组件执行原理进行更加详细的介绍. 为便于读者理解, 以如下的含有 6 个属性的关系表 $R$ 和 4 类查询组成的工作负载 (共 55 个查询) 为例, 记为例子 1.

关系表: $R(a_1,a_2,a_3,a_4,a_5,a_6)$.

工作负载: $Q_1$: SELECT $a_1,a_4,a_5$ FROM $R$; 10 条.

$Q_2$: SELECT $a_5,a_6$ FROM $R$; 15 条.

$Q_3$: SELECT $a_1,a_2$ FROM $R$; 10 条.

$Q_4$: SELECT $a_4,a_5,a_6$ FROM $R$; 20 条.

#### 2.2.1 负载收集器

负载收集器对软件驱动程序 (通常是 ODBC 与 JDBC) 发送的查询流进行批处理, 如图 2 所示, 按照预先设定的时间片将查询逐个填充到 batch 中, 收集完成的 batch 将被发送到线程类 processor 作后续处理. Processor 由 batch 容器、线程池、亲和度矩阵这 3 部分组成, 其中 batch 容器类似于链式结构, 它的容量无限制但已添加的 batch 不可更新; 亲和度矩阵会根据容器内 batch 的负载特征进行计算; 线程池供后续组件使用.

(1) Batch 结构与分配规则

Batch 是一个字典结构, 包括时间片和容量为 8 的查询负载. 每个 batch 都被分配一个时间片, 它需要对复杂查询语句进行特征提取, 如多表查询需要拆分到多个单表的查询、对嵌套子查询进行合并、识别系统函数等, 最终形成的负载见图 2, 即包含一组查询, 每个查询的特征信息包括访问表、访问属性、频数、扫描键、选择度.

Batch 的分配规则主要解决两个问题: (1) 到达的每一个查询应分配到哪个 batch 中. (2) Batch 如何分配给

processor. 首先, 所有的 batch 将被分组, 保证不同组的 batch 内不允许出现重合属性的查询. 然后, 当到来一个新的查询时, 检测目前未装满的 batch, 将其添加到含有重合属性的组中, 如果出现多个组含有重合属性, 则将这些组合并成一个 batch 组, 以此来解决问题 1. 当负载收集器接收到软件系统发送的分区信号时, 所有 batch 将所分配查询的始末时间作为时间片, 每个 batch 组分配一个 processor, 以此解决问题 2.

(2) 从 batch 中提取亲和度矩阵

Processor 将 batch 组中的负载特征转化为亲和度矩阵记为 $W_t$, 下面介绍亲和度矩阵的计算方法.

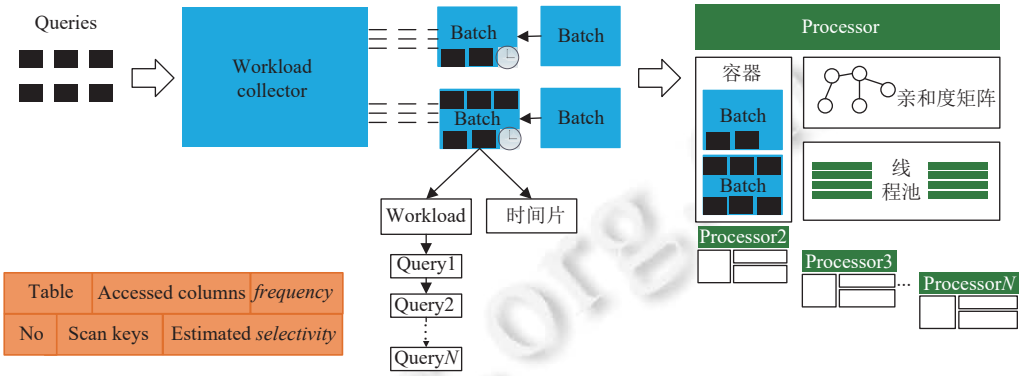



图 2 负载收集器的执行过程

亲和度反映两个或多个对象共同出现的概率程度, 通常用共同访问频率来表示. 这里以第 2.2 节的例子 1 为例进行说明, 根据例子 1 可提取负载数据下属性的使用矩阵 (attribute usage matrix, AUM) 见表 1, 对于每一行的查询, 0-1 代表该查询是否使用指定的属性, *frequency* 列表示一定时期内, 查询出现的次数. 则亲和度矩阵 $W$ 中属性 $a_i$ 和 $a_j$ 的亲和度可以定义为:

$$w_{ij}=aff(a_i,a_j)=\sum_{k=1}^{m} freq_k,\ u(q_k,a_i)=1\wedge u(q_k,a_j)=1 \tag{2}$$

表 1 属性使用矩阵

| Queries | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | *frequency* |
|---|---|---|---|---|---|---|---|
| $q_1$ | 1 | 0 | 0 | 1 | 1 | 0 | $freq_1=10$ |
| $q_2$ | 0 | 0 | 0 | 0 | 1 | 1 | $freq_2=15$ |
| $q_3$ | 1 | 1 | 0 | 0 | 0 | 0 | $freq_3=10$ |
| $q_4$ | 0 | 0 | 0 | 1 | 1 | 1 | $freq_4=20$ |

表 1 和公式 (2) 中, $freq_k$ 表示查询 $q_k$ 总的访问次数, $u(q_k,a_i)$ 表示查询 $q_k$ 是否访问属性 $a_i$, 1 代表访问, 0 代表不访问.

按照公式 (2), 代入具体的数值, 则表 1 中的属性使用矩阵生成的亲和度矩阵 $W$ 为:

$$W=\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{16} \\ w_{21} & w_{22} & \cdots & w_{26} \\ \vdots & \vdots & \ddots & \vdots \\ w_{61} & w_{62} & \cdots & w_{66} \end{bmatrix}=\begin{bmatrix} 20 & 10 & 0 & 10 & 10 & 0 \\ 10 & 10 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 10 & 0 & 0 & 30 & 30 & 20 \\ 10 & 0 & 0 & 30 & 45 & 35 \\ 0 & 0 & 0 & 20 & 35 & 35 \end{bmatrix}.$$

在亲和度矩阵 $W$ 中 $w_{ij}$ 表示属性 $a_i$ 和 $a_j$ 的亲和度, 等同于两个属性共同出现的次数. 如 $W_{11}=20$ 的计算方法为: 在表 1 中访问 $a_1$ 的查询有 $q_1$、$q_3$, 根据它们在负载中出现的次数, 可知属性 $a_1$ 共出现 $10+10=20$ 次; 同理 $W_{45}=30$ 的计算方法为: 在表 1 中共同访问 $a_4$、$a_5$ 的查询有 $q_1$、$q_4$, 它们在负载中出现 10 和 20 次, 则属性 $a_4$、

$a_5$ 共出现 $10+20=30$ 次.

(3) Processor 的并行处理

由于不同 processor 中的负载特征是不关联的, 对于拥有独立分区特点 (见第 3.1 节) 的数据库环境, 属性聚类器及其后续组件均可并行处理, 否则, 只能串行处理.

2.2.2 属性聚类器

在得到 processor 的亲和度矩阵后, 可以基于亲和度得到一个粗略的分区方案. 这样做有两个原因: 第一, 分区内的亲和度越大并不代表最终的分区成本低, 因此只能得到粗略的分区结果, 并非最优解. 第二, 算法初始状态下面临的解空间很大, 如果直接依据成本模型设计精细的算法, 时间复杂度很高.

SCVP 使用图分割算法谱聚类生成初始分区. 谱聚类[25]是一种基于图论的无监督聚类算法, 与传统 K-means 算法相比, 它对数据分布的适应性更强, 聚类效果也更优秀. 它的主要思想是把所有的数据看做空间中的点, 距离越远, 点之间的边权重值越小, 对所有数据点进行切割, 让切图后不同的子图间边权重和尽可能低, 子图内边权重尽可能的高, 从而到达聚类目的, 这里的权重可以用关系表中属性间的亲和度来表示.

对于无向图 $G$ 的切图过程, 设 $V$ 和 $E$ 分别代表 $n$ 个点和 $m$ 条边的集合, 目标是将图 $G(V,E)$ 切成相互没有连接的 $k$ 个子图, 每个子图点的集合为: $A_1,A_2,A_3,\ldots,A_k$, 它们满足 $A_i\cap A_j=\varnothing$ 且 $A_1\cup A_2\cup\ldots\cup A_k=V$.

**定义 1**. 对于任意两个子图 $A_x,A_y\subset V$, 定义 $A_x$ 和 $A_y$ 之间的权重为:

$$W\left(A_x,A_y\right)=\sum_{v_i\in A_x,v_j\in A_y}w_{ij} \tag{3}$$

其中, $w_{ij}$ 为点 $v_i$ 和 $v_j$ 的亲和度, 且 $w_{ij}=w_{ji}$.

**定义 2**. 对于图 $G$ 中任意一个点 $v_i\in V$, 它的度 $d_i$ 定义为和它相连的所有边的权重之和, 即 $d_i=\sum_{j=1}^{n}w_{ij}$. 因此, 所有点的度可以组成一个 $n\times n$ 的度矩阵 $D$, 它是一个对角矩阵, 只有主对角线有值, 对应第 $i$ 个点的度数.

$$D=\begin{pmatrix}d_1 & & & \\ & d_2 & & \\ & & \ddots & \\ & & & d_n\end{pmatrix} \tag{4}$$

谱聚类有两种流行的切图算法: (1) RatioCut 切图: 对每个切图, 不光考虑最小化 $W(A_1,A_2,\ldots,A_k)$, 它还同时考虑最大化每个子图中点的个数, 即: $\mathrm{RatioCut}(A_1,A_2,\ldots,A_k)=\frac{1}{2}\sum_{i=1}^{k}\frac{W\left(A_i,\bar{A}_i\right)}{|A_i|}$. (2) Ncut 切图: Ncut 切图和 RatioCut 切图类似, 只是把 RatioCut 的分母 $|A_i|$ 换成 $vol(A_i)$ 来更好表示一个子图内点的权重, 这里子图样本的个数 $|A_i|$ 越大并不代表图内权重和就大, 则 $\mathrm{NCut}(A_1,A_2,\ldots,A_k)=\frac{1}{2}\sum_{i=1}^{k}\frac{W\left(A_i,\bar{A}_i\right)}{vol(A_i)}$, 这里, $vol(A):=\sum_{i\in A}d_i$, $d_i$ 代表图 $G$ 中点 $v_i$ 的度.

谱聚类的执行流程为: ① 根据邻接矩阵 $W$ 构建度矩阵 $D$. ② 计算拉普拉斯矩阵 $L=D-W$, 并构建标准化后的拉普拉斯矩阵 $D^{-1/2}LD^{-1/2}$, 这里 $L$ 是一个对称矩阵, 便于后续求属性特征值时进行化简. ③ 计算 $D^{-1/2}LD^{-1/2}$ 最小的 $k_1$ 个特征值所各自对应的特征向量 $f$, 并将各属性对应的特征向量 $f$ 组成的矩阵按行标准化, 最终组成 $n\times k_1$ 维的特征矩阵 $F$. 这里的 $k_1$ 是对原有的维度 $n$ 进行降维得到的, 因为求解 NCut 是一个 NP 难问题. ④ 对 $F$ 中的每一行作为一个 $k_1$ 维的样本, 用传统聚类方法 K-means 或 DBscan 进行聚类, 聚类数量设置为 $k$, 得到簇划分 $C(C_1,C_2,\ldots,C_k)$.

下面从对谱聚类算法关键参数的设置进行分析, 包括 3 个方面.

(1) 邻接矩阵的选择和优化

将亲和度矩阵直接作为邻接矩阵, 此时属性的亲和度为边的权重. 如果 processor 被并行处理, 它的负载数据中将存在大量属性未访问, 这常常发生, 此时亲和度矩阵是一个稀疏矩阵. 为此, SCVP 采取三元组顺序表压缩亲和度矩阵, 减少存储空间, 在由三元组映射为邻接矩阵时, 只将被访问属性写入邻接矩阵中.

(2) 切割方式和聚类算法的选择

选择切割效果更好的 NCut 作为切割目标, 谱聚类第 4 步对属性特征矩阵 $F$ 聚类时, 使用 K-means 聚类.

(3) 聚簇数 $k$ 值的确定

谱聚类需要预先指定聚簇数 $k$, 传统评价聚类效果的指标如 Calinski-Harabasz[26]通过点坐标和聚簇中心点坐标来计算类间距离和类内距离, 由于邻接矩阵没有具体的点坐标和聚簇中心点, 并不适用. 因此 SCVP 重新设计一种评估聚类效果的指标 CH4SP (Calinski Harabasz for spectral clustering).

设 $W=\begin{bmatrix} w_{1,1} & \cdots & w_{1,n} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \cdots & w_{n,n} \end{bmatrix}$ 为 $n$ 个属性的邻接矩阵 $w_{i,j}$ 代表属性 $a_i$ 和 $a_j$ 的权重, 假设聚类后形成了 $k$ 个聚簇, 则其簇内距离 $B_k^{(i)}$ 可以用公式 (5) 表示簇内的每个点与簇内其他点的边权重和再除以簇内总边数. 簇间距离 $Z_k^{(i)}$ 可以表示为第 $i$ 个簇与其他簇的距离和, 其中簇与簇之间的距离表示为簇内每个点与簇外其他点的边权重和除以总边数, 并计算所有点的距离的平均值, 见公式 (6)、公式 (7). CH4SP 即为簇内距离 $B_k$ 与簇间距离 $Z_k$ 的比值, 指标值越大, 聚类效果越好.

$$B_k=\sum_{q=1}^{k}\frac{\sum_{i\in C_q}\sum_{j\in C_q}w_{ij}}{\left(|C_q|-1\right)\times|C_q|},\ i\neq j \tag{5}$$

$$Z_k=\sum_{q=1}^{k}\sum_{m=1}^{|C_q|}\sum_{n=1}^{|c_q|}d(C_m-C_n),\ m\neq n \tag{6}$$

其中, 类 $C_m$ 与 $C_n$ 间的距离定义为:

$$d(C_m-C_n)=\frac{\sum_{i\in C_m}^{|C_m|}\left(\sum_{j\in C_n}^{|C_n|}w_{i,j}/|C_n|\right)}{|C_m|} \tag{7}$$

根据例子 1 的数据, 得到最佳的 $k$ 值为 3, 聚类结果为 $\{a_1,a_2\}$、$\{a_3\}$、$\{a_4,a_5,a_6\}$.

2.2.3 分区优化器

分区优化器主要采用频繁项集挖掘和贪婪搜索相结合的方法来指导初始分区的更新过程, 包括分区的修剪、以及多个分区的合并和交叉等过程. 首先由 FP-Growth 算法[27]计算负载数据中访问属性的频繁项集, 并建立项数与频繁模式的 Hash 表, 作为先验信息, 然后分区优化器内部设置分区切割器和分区聚合器两个更新组件, 其中分区切割器负责分区修剪, 产生分区单元 (具有原子性, 对其任意分割都会造成整体分区方案成本增加, 是不可缩小的最小单元). 分区聚合器对分区单元进行合并和交叉, 得到最终分区方案.

2.2.3.1 分区切割器

分区切割器的功能是分区修剪, 将初始分区切分成具有原子性的分区单元. 原因在于: 初始分区方案是依据亲和度矩阵生成, 存在部分分区亲和度很高但由成本模型计算的代价成本很高, 需要对其进行切割降低成本, 每次切割只切一次, 即单个分区只能被切分两个分区, 新产生的所有分区将作为新的独立分区继续被分区切割器修剪. 分区修剪主要分为 6 个步骤.

步骤 1. 将属性聚类器得到的聚簇作为初始分区, 首先判断当前数据库环境是否拥有独立分区特点 (见第 3.1 节), 如果是, 则将每个分区交由线程池分发给指定的线程处理. 如果不是, 则由单个线程对所有分区顺序处理.

步骤 2. 线程将待切割的分区放入到队列中, 并创建一个空的待分配列表. 从工作负载提取最小支持度为 $\theta$ 的所有频繁模式, 并为其建立 Hash 表.

步骤 3. 从队列中 pull 一个待切割分区元素, 首先进行异常检测, 如果属性数小于 2, 则直接将其添加到分区单元集合中, 否则对其进行切割操作.

步骤 4. 从待分配列表中取出该待切割分区属性子集的所有频繁模式 (若待分配列表为空, 则从 Hash 表中取

出), 根据频繁模式将分区切为两段, 代入成本模型中计算按此频繁模式切割该分区的收益, 即 $Cost_{切割前}-Cost_{切割后}$, 并按照收益大小 (大于 0) 将其该频繁模式降序添加到待分配列表中.

步骤 5. 如果待分配列表不为空, 从待分配列表中取出第 1 个频繁模式, 即奖励最大的频繁模式作为切割点, 将该频繁模式 push 到队列中, 然后删除待分配列表中所有与该切割点有交集的频繁模式, 将分割后剩余的另一半分区作为待切割分区, 执行步骤 4; 如果待分配列表为空, 则将该待切割分区加入到分区单元集合中, 此时检测队列中是否有元素, 如果有, 转入步骤 3, 否则执行步骤 6.

步骤 6. 输出分区单元集合, 结束.

以例子 1 说明分区切割器的运行机理, 如图 3 所示, 假设当前为非独立分区, 则线程 1 处理初始分区{1, 2}、{4, 5, 6}、{3}, 将其放入 cluster queue 中, 从队列中取出一个分区{4, 5, 6}, 然后对其进行异常值检测, 观察到属性数大于 2, 需要进行切割处理, 求出范围{4, 5, 6}最小支持度为 $\theta$ 的频繁模式, 并根据它们的切割奖励收益降序排列, 取出奖励最大的频繁模式{5, 6}作为第 1 次切割方案, 并将该切割项添加到 cluster queue 中. 此时, 对未切割部分{4}重新进行异常值检测, 观察其属性数小于 2, 则将其添加到分区单元中, 作为第 2 次切割方案. 未切割部分只剩下空集, 表明分区{4, 5, 6}已切割完成, 继续从 cluster queue 取出首元素, 重新进行上述操作, 直到队列为空时, 停止. 最终得到的分区单元为{1, 2}、{3}、{4}、{5, 6}. 更具体的描述过程详见算法 1.

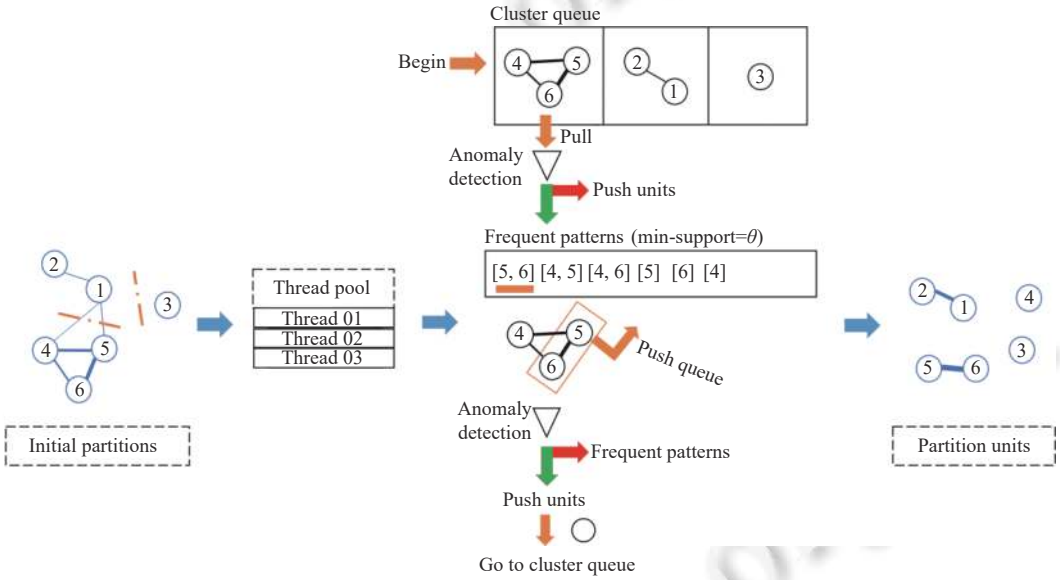



图 3 分区切割器的执行过程

**算法 1**. 对初始分区进行切割.

Input: 初始分区集合 $P^{split}$, 表属性 $A$, 工作负载 $WD$;
Output: 分区单元集合 $P_{units}$.

1 $L$=FPGrowth($WD$, $A$) //从负载 $WD$ 中提取全表属性下的频繁模式集合
2 Initialize(Queue) //存放切割点的队列, 初始为空
3 Initialize(Set) //待分配列表: 存放临时的频繁模式, 初始为空
4 **for** $P^{split}$ 中的每一个分区 $P_i^{split}$ **do**
5 push(Queue, $P_i^{split}$) //向队列中添加待切割分区
6 **while** 从 Queue 中 pop 出第一个元素 $P_0^{split}$ **do**

```
7          if size(P_0^split) == 1 do //如果分区属性小于 2, 则将其添加到分区单元列表
8              add(P_units, P_0^split)
9              continue
10         end if
11         L_i=getFrequentPattern(L, P_i^split ) //从 L 中提取分区 P_i^split 涉及属性范围下的频繁模式集合
12         for L_i 取出每一个频繁模式 L_ij do
13             P_l = P_0^split − L_ij
14             P_r = L_ij
15             reward=costmodel( P_0^split ) – costmodel([ P_l, P_r ])
16             //将每个切割点和其切割收益, 按照收益大小添加到待分配列表 Set 指定的位置
17             add(Set, ( L_ij , reward))
18         end for
19         Unallocate_P = P_0^split //切割后的分区, 初始状态为待切割分区 P_0^split
20         while 取出 Set 的第一个频繁模式 L_0 do
21             push(Queue, L_0) //将收益最大的 L_0 作为切割点加入到队列中
22             Unallocate_P = Unallocate_P − L_0
23             delete(Set, Unallocate_P) //删除 Set 中不属于分区 Unallocate_P 范围下的频繁模式
24             //更新待分配列表 Set 中在新的分区 Unallocate_P 下的切割收益
25             updateCutReward(Set, Unallocate_P)
26             sortByReward(Set) //按照切割收益对待分配列表降序排列
27         end while
28         //待分配列表中无正向收益的频繁模式时, 将剩余分区加入到分区单元列表中
29         add( P_units , Unallocate_P )
30     end while
31 end for
32 return P_units
```

2.2.3.2 分区聚合器

分区聚合器负责对分区切割器得到的分区单元进行组合. 分区聚合器放在分区切割器之后执行有两个原因: 第一, 分区单元作为粒度最小的单元, 在这个条件下进行合并操作, 能提供更多的组合方案, 便于找到最优解. 第二, 没有像 HillsClimb 算法一样选择从单个属性开始进行合并, 主要考虑到高维度下单个属性的组合规模太大, 显著影响执行时长.

关于分区聚合器中组合方法的设计, 一种朴素的方案是任意两个分区单元进行组合, 取出组合后收益最大的方案, 但导致的后果是, 每轮选择时都要列出所有的组合方案, 计算所有方案的成本, 并逐个选择成本最小的一种, 整个过程需要计算所有的组合方案. 分区聚合器采取了另外一种思路, 依据频繁模式这个先验信息, 进行分区单元的合并. 后续的实验证明这种思路带来的收益略低于朴素的方案, 但在高维属性下节约了将近 200 倍的时间成本. 分区的合并过程主要分为 5 个步骤.

步骤 1. 从工作负载提取最小支持度为 $\theta$ 的所有频繁模式, 并为其建立 Hash 表. 创建一个空的待分配列表, 负责存放后续要处理的频繁模式.

步骤 2. 从 Hash 表中依次取出每个频繁模式, 找到其所含属性涉及到的所有分区单元, 如果该分区单元组合被之前的频繁模式使用过, 则直接跳过该频繁模式, 如果未使用过, 则将其合并为一个分区, 计算合并前后的收益,

即 $Cost_{合并前}-Cost_{合并后}$, 并按照奖励大小 (大于 0) 将其按从大到小的顺序加入到待分配列表中, 直到所有频繁模式都计算完毕.

步骤 3. 从待分配列表中取出第 1 个频繁模式, 即收益最大的频繁模式作为合并点. 然后删除待分配列表中所有与该合并点有交集的频繁模式 (这里的交集指的是合并点与频繁模式所对应的分区单元组合是否有交集), 并将该合并点所对应的合并分区加入到最优分区方案中, 同时从分区单元集合中删除已被合并的分区单元, 如果待分配列表不为空, 执行步骤 4, 否则执行步骤 5.

步骤 4. 由于有新的合并分区产生, 需要重新更新待分配列表中所有频繁模式的合并奖励收益, 并对待分配列表进行降序排列, 执行步骤 3.

步骤 5. 将分区单元集合中未合并的分区单元添加到最优分区方案, 结束.

以例子 1 说明分区聚合器的运行机理, 如图 4 所示. 取出全部属性范围下的频繁模式, 找到每个频繁模式相关联的分区单元并加入待分配列表中, 例如频繁模式 [1, 4, 5] 关联的分区单元组合为{{2, 1}、{4}、{5, 6}}、频繁模式 [2, 1, 5, 6] 关联的分区单元组合为{{2, 1}、{5, 6}}等, 应注意, 被关联的分区单元组合不能重复出现在其他的频繁模式中, 例如频繁模式 [1, 5]、[2, 1, 5, 6] 的分区单元组合相同, 均为{{2, 1}、{5, 6}}, 若频繁模式 [2, 1, 5, 6] 已被加入列表中, 则频繁模式 [1, 5] 将被忽略. 将待分配列表中的频繁模式按照合并收益 (大于 0) 降序排列, 取出首元素 [1, 4, 5], 将其分区单元组合形成的合并分区{1, 2, 3, 4, 5, 6}加入到最优分区列表中, 作为第 1 次合并项; 假设当前为非独立分区, 需要重新更新待分配列表中频繁模式的合并收益, 降序排列, 取出首元素 [2, 1, 5, 6], 其分区单元中含有{5, 6}、{2, 1}已经被分配, 则删除该频繁模式, 重新执行上述操作, 直到待分配列表为空时, 停止. 输出最优分区方案{1, 2, 4, 5, 6}、{3}. 更具体的描述过程详见算法 2.

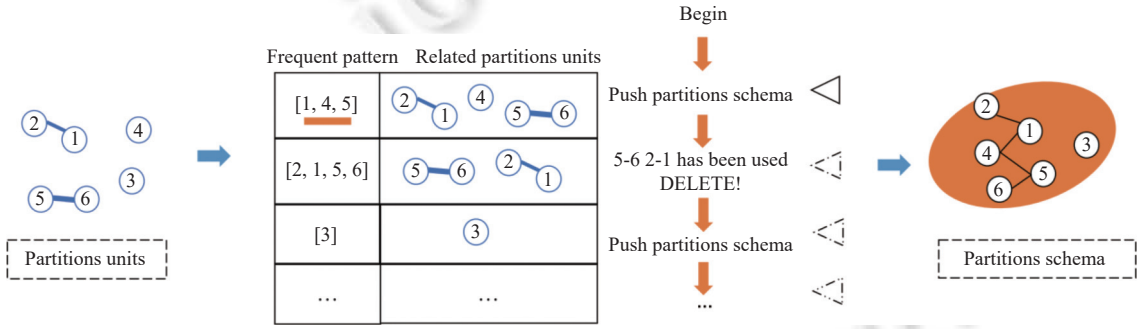



图 4 分区聚合器的执行过程

**算法 2**. 对分区单元进行合并.

Input: 分区单元集合 $P_{\text{units}}$, 全表范围下的频繁模式集合 $L$;
Output: 最终分区方案 $\hat{P}$.

1 Initialize(UnitsMap) //存放已被合并过的分区单元组合, 初始为空
2 Initialize(Queue) //存放切割点的队列, 初始为空
3 Initialize(Set) //待分配列表: 存放临时的频繁模式, 初始为空
4 **for** $L$ 中的每一个频繁模式 $L_i$ **do**
5 $P^i_{\text{units}}$ = findRelatedUnits($P_{\text{units}}$, $L_i$) //寻找与频繁模式 $L_i$ 相关的所有分区单元
6 //判断该分区单元组合是否被使用, 若是处理下个循环, 否则, 将该组合添加到 UnitsMap 中
7 **if** UnitsMap.haskey($P^i_{\text{units}}$) **do**
8 continue
9 **else**

put(UnitsMap, $P^i_{\text{units}}$)
**end if**
$P^i$ =mergeUnits($P^i_{\text{units}}$) //合并分区单元
reward= costmodel($P^i_{\text{units}}$) – costmodel($P^i$) //计算合并收益
// 将每个合并点、合并分区及其合并收益, 按照收益大小添加到待分配列表 Set 指定的位置
add(Set, ($L_i$, $P^i_{\text{units}}$, reward))
**end for**
**while** 取出 Set 的第一个频繁模式 $L_0$ 及其分区单元组合 $P^0_{\text{units}}$ **do**
$P^0$ = mergeUnits($P^0_{\text{units}}$) //合并分区单元
push(Queue, $P^0$) //将收益最大的 $L_0$ 作为合并点, 并将合并后的分区加入队列
delete($P_{\text{units}}$, $P^0_{\text{units}}$) //从总的分区单元集合中删除被使用的分区单元
delete(Set, $P^0_{\text{units}}$) //删除 Set 中的涉及到相关分区单元的所有频繁模式
updateMergeReward(Set, $P^0$) //合并分区后, 更新待分配列表 Set 中所有频繁模式的合并收益
sortByReward(Set) //按照合并收益对待分配列表降序排列
**end while**
$\hat{P} = P_{\text{units}} \cup$ Queue
**return** $\hat{P}$

2.2.3.3 分区生成器

分区生成器负责将垂直分区方案在数据库中在线部署, SCVP 提供了两种部署方法, 第 1 种是根据分区方案由程序自动拼接分区的 DDL/DML 语句通过 PDBC 在数据库中执行, 第 2 种是创建数据库自定义函数, 设置触发器条件, 根据输入的分区方案在线建立分区. SCVP 参考 KingBase、达梦 (DM) 数据库文档, 对部署过程中作如下约束.

(1) 垂直分区算法需要根据原表结构数据及分区方案, 自动创建多个子表以及所有子表连接的分区基表.

(2) 分区基表不保存实际数据, 只保存表定义、分区信息, 实际数据保存在分区子表中.

(3) 建立垂直分区的表必须对主键建立聚集索引, 即 CLUSTER PK, 并且此聚集索引不允许删除. 一般对原表的主键建立聚集索引, 并复制到各个子表中.

(4) 不支持任何列同时出现在多个分区子表中, 即不支持列重复.

2.2.3.4 基于 SCVP 的性能优化模型: SCVP-R

从第 2.2 节可以看出, SCVP 中分区聚合器组件对分区单元的合并思想设计很简单, 因为 SCVP 旨在考虑降低时间复杂度. 但同时也暴露出合并过程中的一些明显问题.

问题 1. 如果一个分区单元组合被选中合并, 则后续关联该组合的频繁模式将不会被考虑, 这是因为不同频繁模式的分区单元组合如果相同, 则对应的合并分区也相同, 属于互斥关系, 只选择一个即可. 但与此同时带来的问题是, 多个不同的频繁模式被当做一种情况处理, 分区单元组合就不能完全反映频繁模式的特征.

问题 2. 待分配列表中的频繁模式一旦被选中作为合并项, 其分区单元将被使用, 则后续含有该分区单元的频繁模式将被删除, 但如果存在多个频繁模式属于合并项的互不重叠的子集, 它们的合并收益之和大于排序靠前的频繁模式的合并收益, 则理应选择前者作为合并项.

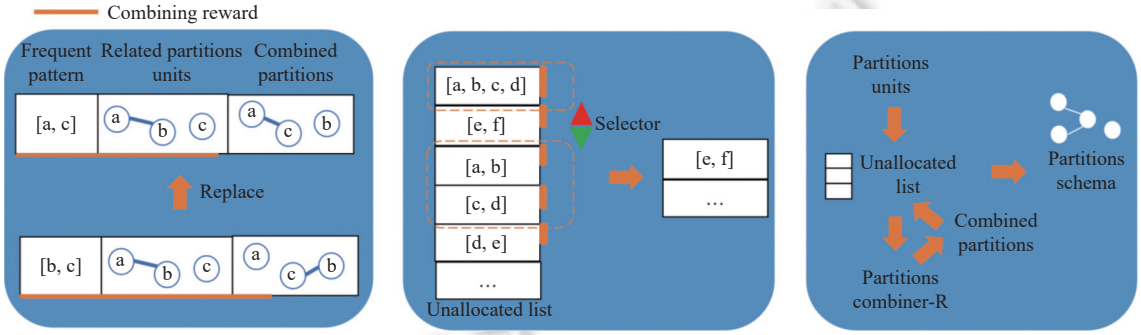
问题 3. SCVP 仅在分区单元的基础上进行合并, 合并后的分区方案即作为最终分区. 由于贪心机制, 第 1 轮合并过程中, 部分收益小的频繁模式被忽略, 因此合并后的分区方案, 仍有继续合并的可能性.

为此, 本节将针对这些问题, 提出了新的分区聚合器-R (partitions combiner-R) 组件, 替换原有的分区聚合器组件, 以此构建基于 SCVP 的性能优化模型: SCVP-R, 它的目标就是提升目前垂直分区方法在高维属性下的性能瓶颈问题, 同时保证时间复杂度在可控的范围内 (非指数级) 增长. 分区聚合器-R 通过引入 3 种机制: 同域竞争机制、

双败淘汰机制、循环机制来解决分区聚合器存在的 3 个问题, 具体解决思路如下.

(1) 同域竞争机制 (sympatric-competition mechanism)

在构建待分配列表时, 每个频繁模式关联的分区单元组合, 将合并为两个分区: 频繁模式本身、剩余属性组成的分区, 替换原先只合并为单个分区的思路. 例如 [a, c] 关联的分区单元组合为{{a, b}、{c}}, 则该组合最终合并为{a, c}、{b}两个分区, 但这样做也将导致了大量的候选频繁模式, 为了降低复杂度, 分区聚合器-R 在待分配列表的构建时, 引入同域竞争机制, 关联相同分区单元组合的多个频繁模式只保留合并收益较大的一方, 如图 5(a), 频繁模式 [a, c]、[b, c] 的分区单元组合相同, 均为{{a, b}、{c}}, 且 [a, c] 已经被添加到待分配列表中, 但后续比较中, [b, c] 的合并收益更大 (图中橘色线条更长), 则 [b, c] 将会替换 [a, c].




(a) Sympatric-competition mechanism (b) Double-elimination mechanism (c) Loop mechanism

图 5 分区聚合器-R 的 3 种机制

(2) 双败淘汰机制 (double-elimination mechanism)

在构建待分配列表时, 会按照合并收益排序, 目的是选择收益最大的合并项, 并及时淘汰与之属性域有冲突的频繁模式, 此为单败制. 引入双败制后, 从待分配列表中取出的第 1 个元素, 原本应作为合并项, 但由于双败的考虑, 还会进一步比较, 如果发现列表中存在该合并项的多个不重叠的子项, 它们的合并收益总和大于原始合并项, 则淘汰原始合并项, 将其子项作为第 1 次合并项. 如图 5(b), 选择器进行计算后, 发现频繁模式 [a, c]、[c, d] 的合并收益大于原始合并项 [a, b, c, d], 则选择 [a, c]、[c, d] 作为新的合并项, 并删除与之属性域有冲突的所有频繁模式.

(3) 循环机制 (loop mechanism)

如图 5(c), 对分区单元建立待分配列表, 然后执行合并过程对列表中的频繁模式分配, 得到合并方案; 将合并方案作为初始输入, 重新建立待分配列表, 执行重复的合并过程, 直到没有正向收益的频繁模式时, 停止循环, 得到最终分区方案.

## 3 相关定义与成本模型

本节将按照分区间的特性对分区进行分类, 包括独立分区与耦合分区的概念, 并介绍后续实验所使用的基于磁盘数据库的分区成本模型 HDD-I.

### 3.1 独立分区与耦合分区

根据数据库实际运行环境的不同, 可以将分区间的关系划分为独立分区与耦合分区.

(1) 独立分区: 如果某个分区的内部划分 (即一个分区可能被拆解成多个分区) 如何改变, 都不影响其他分区的现有收益.

(2) 耦合分区: 一个分区内部划分的改变会影响到其他分区的当前收益情况, 但收益变化未知. 其中, 耦合分区又根据某个分区结构改变后其他分区的收益变化趋势是否相同, 划分为耦合反馈分区和耦合相对反馈分区.

• 耦合反馈分区: 如果该分区划分后的收益变化趋势与其他分区的收益变化相同, 则称分区为耦合正反馈分区;

若变化趋势相反, 则分区为耦合负反馈分区. 易知, 独立分区是典型的耦合正反馈分区.

• 耦合相对反馈分区: 如果一个分区改变后, 其他分区的收益变化趋势不相同, 有的收益增加, 有的收益减少, 但收益值排序的相对顺序不变, 则称该分区为耦合相对正反馈分区, 如果相对顺序完全相反, 则称该分区为耦合相对负反馈分区.

根据目前垂直分区领域的已有的成本函数[22,28], 存在两点结论.

结论 1: 单机数据库的分区属于独立分区或耦合正反馈分区.

结论 2: 分布式数据库的分区属于耦合正反馈分区或耦合相对正反馈分区.

### 3.2 HDD-I 成本模型

HDD (hard disk cost model) 成本模型[22]能够估算在设置的数据库环境中一种分区方案的磁盘 I/O 访问成本. 该模型规定: 如果一个分区包含查询 $Q$ 的扫描键, 则称该分区为查询 $Q$ 的主分区; 如果某个分区包含查询 $Q$ 的部分访问属性却不含扫描键, 则称该分区为次分区; 次分区上必须建立至少一个与主分区相同的列索引, 一个查询只能访问一个主分区, 但可以访问多个次分区, 总访问成本等于主分区和次分区的成本之和.

查询 $Q$ 访问主分区的扫描情况有 3 种, 如果该分区的扫描键已建有索引, 则视为聚集索引扫描 (clustered index scan), 若在该分区的其他属性建有索引, 使用非聚集索引扫描 (unclustered index scan) 方式计算, 否则进行顺序扫描 (sequential scan). 访问主分区的成本计算公式 (8)–公式 (11) 如下:

$$Cost_{\mathrm{MF}} = \min(C_{\mathrm{cic}}, C_{\mathrm{uic}}, C_{\mathrm{sc}}) \tag{8}$$

聚集索引下扫描成本:

$$C_{\mathrm{cic}} = \left\lceil \frac{Cardinality \times selectivity \times Fragment\_length}{(page\_size)} \right\rceil \tag{9}$$

非聚集索引下的扫描成本:

$$C_{\mathrm{uic}} = Cardinality \times selectivity \tag{10}$$

顺序扫描成本:

$$C_{\mathrm{sc}} = \left\lceil \frac{Cardinality \times Fragment\_length}{(page\_size) \times (prefetch\_blocking\_factor)} \right\rceil \tag{11}$$

其中, MF 是主分区 (main fragments) 的缩写, *Cardinality* 表示原表的元组总数, *Fragment_length* 表示分区的长度 (字节), *selectivity* 指表中所有数据满足查询的扫描键中谓词的元组比例, *prefetch_blocking_factor* 指用于减少磁盘 I/O 预取的页数, 默认为 10, *page_size* 表示一个物理页所能存放的数据大小 (字节), 默认为 5 KB.

查询 $Q$ 访问所有次分区的扫描成本可用公式 (12) 计算:

$$Cost_{\mathrm{SF}} = (The\_number\_of\_secondary\_fragments) \times (Cardinality) \times (selectivity) \tag{12}$$

其中, SF 是次分区 (secondary fragments) 的缩写, *The_number_of_secondary_fragments* 表示次分区的数量.

HDD 存在一些不足, 如:

(1) 次分区的成本计算没有考虑到分区的长度.

(2) 没有考虑分区之间的连接成本, 导致生成次分区的优先级显著高于主分区.

(3) 查询的扫描键只考虑到一个属性, 而现实的查询往往是多个.

为此, 本文对 HDD 进行简化和改进, 提出新的成本模型 HDD-I, 包括改进了主分区和次分区访问成本 $Cost_{\mathrm{MF}}$、$Cost_{\mathrm{SF}}$ 的计算方法、在次分区访问成本中考虑到分区间进行 JOIN 操作的连接成本 $Join\_SF$、将查询的扫描键扩展到多个属性, 并依据 KingBase、达梦 (DM) 等数据库产品的规定: 对原始表垂直划分得到的所有子分区, 额外增加一个主键字段, 并在主键上建立聚集索引.

对于查询 $Q$, 其扫描键中所有属性涉及的分区记为主分区, 其他访问属性涉及的分区记为次分区, 查询 $Q$ 可以访问多个主分区和次分区, 它的磁盘 I/O 成本可用公式 (13)–公式 (15) 来估计:

$$EstimateCost = (Cost_{\mathrm{MF}} + Cost_{\mathrm{SF}}) \times frequency \tag{13}$$

$$\begin{cases} Cost_{\mathrm{MF}} = \sum_{i=1}^{m} Main\_Fragment_i \\ Main\_Fragment_i = \dfrac{(Cardinality \times Fragment\_length_i)}{(page\_size) \times (prefetch\_blocking\_factor)} \end{cases} \tag{14}$$

$$\begin{cases} Cost_{\mathrm{SF}} = \sum_{i=1}^{m} Secondary\_Fragment_i + Join\_SF_i \\ Secondary\_Fragment_i = \dfrac{(Cardinality \times Fragment\_length_i \times selectivity)}{page\_size} \\ Join\_SF_i = \dfrac{(Cardinality \times sf\_primary\_key\_length_i)}{page\_size} \end{cases} \tag{15}$$

其中, MF、SF 分别是主分区和次分区的缩写, *frequency* 代表查询 $Q$ 出现的频数, *Cardinality* 表示原表的元组总数, *Fragment_length* 表示分区的长度, *selectivity* 指表中所有数据满足查询的扫描键中谓词的元组比例, *prefetch_blocking_factor* 指用于减少磁盘 I/O 预取的页数, *page_size* 表示一个物理页所能存放的数据大小, *sf_primary_key_length* 表示次分区主键的长度.

## 4 SCVP 和 SCVP-R 的执行速度优化

本节介绍一种切割收益更新函数用于对 SCVP 和 SCVP-R 算法加速, 并将其应用于成本模型 HDD-I.

### 4.1 切割收益更新函数的设计

在分区切割器中, 对单个分区通常要进行多次切割, 每次切割都要重新计算所有频繁模式的切割收益, 重新代入成本函数计算切割前和切割后的成本变化, 成本函数至少达到 O(分区数×负载规模) 的数量级, 则整个过程的时间复杂度为 O(分割平均候选集×分割次数×分区数×负载规模), 虽然属性聚类器得到的初始分区相对于原表长度已经线性缩小, 显著减少了单个分区切割的次数, 但如果存在某个分区属性很多, 延长了单个分区修剪的时间, 整体优化时间就会延长. 为了避免算法执行的这种不稳定性, 同时减少算法整体执行时间, 一种思路是从成本模型的执行时间出发, 对工作负载进行预处理, 提取具有典型特征的查询, 控制负载在一定规模下, 降低成本模型计算时间. 另一种思路是从成本模型的执行次数考虑, 如果经过第 1 轮切割后, 后续再更新每个切割点的收益时, 能够以常数项 O($C$) 的时间复杂度, 代替原始的 O(分区数×负载规模). 由于第 1 种思路不具有现实性, 本节将设计一种切割收益更新函数来实现第 2 种思路, 其限制是必须在独立分区环境下进行.

设访问分区 $P$ 的工作负载为 $WD$, 成本模型 $C = f(P, WD)$, 待切割的分区为 $p^{\mathrm{split}}$, 该分区范围下候选的频繁模式序列为 $L = L_1, L_2, \ldots, L_k$ $(1 \leqslant k \leqslant n)$. 当以频繁模式 $L_i$ 作为切割点时, 分区拆分成 $P_l^i$、$P_r^i$, 设 $n^i$、$n_l^i$、$n_r^i$ 分别为既访问 $P_l^i$ 又访问 $P_r^i$、访问 $P_l^i$ 不访问 $P_r^i$、访问 $P_r^i$ 不访问 $P_l^i$ 的查询集合, 不难得到 $WD = \{n^i, n_l^i, n_r^i\}$. 根据独立分区的概念, 易知 $P_l^i$ 和 $P_r^i$ 的分区奖励互不影响, 则 $L_i$ 的切割收益 $R_i$ 可计算为:

$$R_i = C\left(P_l^i, \{n^i, n_l^i\}\right) + C\left(P_r^i, \{n^i, n_r^i\}\right) - C\left(p^{\mathrm{split}}, \{n^i, n_l^i, n_r^i\}\right) \tag{16}$$

进行第 1 轮切割后, 如果待分配列表中存在收益大于 0 的切割点, 则进行第 2 轮切割, 此时需要重新计算列表中所有频繁模式的切割收益, 例如取 $L_j$ 作为第 2 个切割点的切割收益时, 此时 $L_j$ 的切割有 3 种情形, 如图 6 所示, 设 $L_t$ 为已选中的第 1 条切割点, 图中 $P_l^t$、$P_r^t$ 分别表示第 1 次切割的频繁模式以及剩余的属性段, 同理 $P_l^j$、$P_r^j$ 分别表示 $L_t$ 切割后, 第 2 次切割的频繁模式以及剩余的属性段, $P_r^{j(1)}$ 表示未进行 $L_t$ 切割时, 以频繁模式 $L_j^{(1)}$ 作为第 1 次切割点时剩余的属性段. 情形 1、2、3 分别表示第 2 轮切割点 $P_l^j$ 位于 $P_r^t$ 的左段、右段和中段时, $P_r^j$ 和 $P_r^{j(1)}$ 的分布情况. 对于独立分区的环境, 分区内属性的顺序对负载的访问成本没有影响, 因此情形 3 的两个 $P_r^j$ 拼接在一起时, 等同于情形 1 和情形 2 的分区 $P_r^j$. 同理, 情形 2、3 的两个 $P_r^{j(1)}$ 拼接成一个分区时, 与情形 1 等价. 对于独立分区环境而言, 情形 2 和情形 3 本质上是情形 1 的特殊形式, 与情形 1 的收益完全相同. 因此, 不同复杂程度的切割方式都可以等价于情形 1.

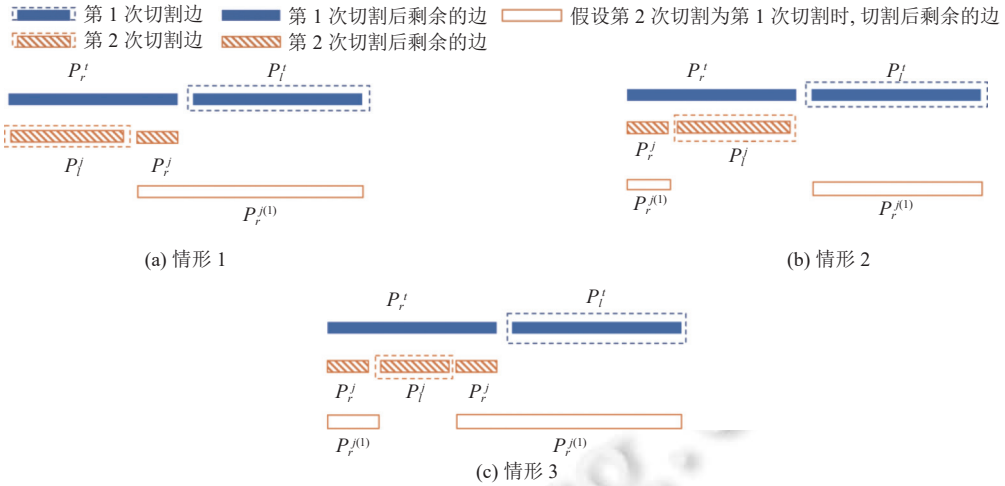



图 6 以频繁项 $L_j$ 作为第 2 轮切割边切割聚簇时存在的 3 种情形

下面以情形 1 为例, 探究如何计算 $L_j$ 的切割奖励: 根据公式 (16), 只需要计算访问 $P_l^j$ 、$P_r^j$ 的查询集合 $n^j$、$n_l^j$、$n_r^j$. 根据第 1 轮切割点 $L_t$ 和 $L_j^{(1)}$ 的中间结果, 能够得到如下关系: 公式 (17) 表示切割前后, 由于 $L_j$ 的 $P_l^j$ 为变化, 对应的 $n^j$ 不变; 公式 (18) 表示切割后, $P_r^j$ 的访问查询 $n_r^j$ 可由切割前 $P_l^{j(1)}$ 的访问查询 $n_r^{j(1)}$ 与第 1 次切割时 $P_l^t$ 的访问查询 $n_l^t$ 的交集得到; 公式 (19) 要计算切割后同时访问 $P_l^j$ 、$P_r^j$ 的查询集合 $n^j$, $n_r^t \cup n^t$ 表示切割前访问 $P_r^t$ 的查询总量, 等同于切割后访问 $n^j$、$n_l^j$、$n_r^j$ 的总和, 因此取 $n_r^t \cup n^t$ 与 $n_l^j \cup n_r^j$ 的交集即可得到 $n^j$.

$$n_l^j = n_l^{j(1)} \tag{17}$$

$$n_r^j = n_r^{j(1)} \cap n_l^t \tag{18}$$

$$n^j = (n_r^t \cup n^t) \cap \left(n_l^j \cup n_r^j\right) \tag{19}$$

将参数 $n^j$、$n_l^j$、$n_r^j$ 代入公式 (16) 即可得到第 2 轮切割 $L_j$ 的切割收益. 第 3 轮切割时, 可以依据第 2 轮的中间结果来计算, 依次类推. 因此, 切割的奖励计算函数 $f_w$ 的一般形式可表述为:

$$f_w = \begin{cases} R_i^{(1)} = C\left(P_l^i, \{n^i, n_l^i\}\right) + C\left(P_r^i, \{n^i, n_r^i\}\right) - C\left(p^0, \{n^i, n_l^i, n_r^i\}\right) \\ R_j^{(k)} = C\left(P_l^j, \{n^j, n_l^j\}\right) + C\left(P_r^j, \{n^j, n_r^j\}\right) - C\left(p^{k-1}, \{n^j, n_l^j, n_r^j\}\right) \\ n_l^j = n_l^{j(k-1)} \\ n_r^j = n_r^{j(k-1)} \cap n_l^t \\ n^j = (n_r^t \cup n^t) \cap \left(n_l^j \cup n_r^j\right) \end{cases} \tag{20}$$

其中, $p^0$ 表示原始待切割分区, $p^{k-1}$ 表示切割 $k-1$ 次后待切割的分区, $R_i^{(1)}$ 代表第 1 轮切割点 $L_i$ 计算的切割收益, $R_j^{(k)}$ 代表第 $k$ 轮切割时若以 $L_j$ 为切割点的收益, 且 $k-1$ 轮的切割边为 $L_t$.

### 4.2 切割收益更新函数在 HDD-I 的应用

下面以成本模型 HDD-I 为例, 应用切割奖励更新函数 $f_w$ 进行奖励的计算, 过程如下.

假设分区 $p^{\text{split}} = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7\}$ 作为待切割分区, 模型的第 1 轮切割点选择时, 需要计算所有频繁模式的切割收益, 为此, 首先介绍第 1 轮切割的计算过程, 不妨设计算主分区和次分区的成本函数分别为 $f_{mf}$ 和 $f_{sf}$, 切割后主分区和次分区的收益分别为 $R_{mf}$、$R_{sf}$. 如图 7, 以某个频繁项如 $L_t = \{a_2, a_6, a_7\}$ 进行切割时, 分区 $p^{\text{split}}$ 分为 $\{a_1, a_5, a_3, a_4\}$、$\{a_2, a_6, a_7\}$ 两部分, 原始长度 $a$ 被切割成两段 $b$、$c$. 设次分区只访问 $b$、只访问 $c$、既访问 $b$ 又访问 $c$ 的查询数量分别为 $n_1$、$n_2$、$n_3$, 设主分区只访问 $b$、只访问 $c$、同时访问 $b$、$c$ 的查询数量为 $n_4$、$n_5$、$n_6$. 记录这 5 类查询在负载 $WD$ 中的下标分别为 $nx_1, \ldots, nx_5$, 并根据下标计算它们的平均选择度, 记为 $avgsel_1, \ldots, avgsel_3$. 因

此切割前后, 对于$n_1$、$n_2$、$n_3$这 3 种查询的切割收益分别为:

$$R_1 = f_{sf}(n_1, c, avgsel_1).$$

$$R_2 = f_{sf}(n_2, b, avgsel_2).$$

$$R_3 = f_{sf}(n_3, a, avgsel_3) - \left[f_{sf}(n_3, b, avgsel_3) + f_{sf}(n_3, c, avgsel_3)\right].$$

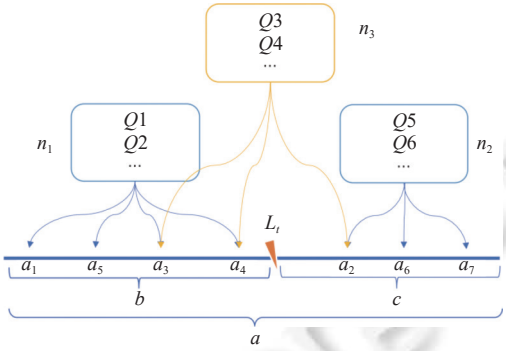



图 7 对分区 $P^{\text{split}}$ 采用频繁模式 $L_t$ 切割后负载的分布变化

因此, 切割后次分区各类查询的总切割收益为:

$$R_{sf} = R_1 + R_2 + R_3.$$

同理$n_4$、$n_5$、$n_6$主分区查询的切割收益为:

$$R_{mf} = f_{mf}(n_4 + n_5 + n_6, a) - \left[f_{mf}(n_4, b) + f_{mf}(n_5, c) + f_{mf}(n_4 + n_5, a)\right].$$

分区 $P^{\text{split}}$ 总的切割收益为:

$$R = R_{mf} + R_{sf} \tag{21}$$

经过第 1 轮切割后, 在后续切割点的选择时, 需要重新计算频繁模式的收益, 根据公式 (21) 可知, 如果能根据切割后分区属性范围的缩小情况, 计算出关键参数$n_1,\ldots,n_6$、$avgsel_1,\ldots,avgsel_3$以及$a$、$b$、$c$, 即可快速更新每个频繁模式的切割收益.

对某个频繁模式例如$L_j = \{a_3, a_4\}$, 在$L_t$第 1 次切割前, 它的参数变量用上标 (1) 表示, 在$L_t$切割后, 它的新的参数变量用上标 $u$ 表示, 如图 8 所示.

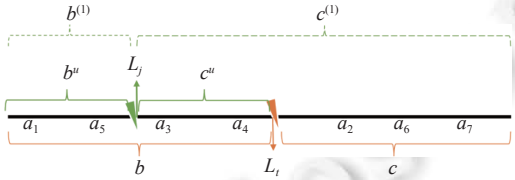



图 8 第 2 轮切割 $L_j$ (绿色切点) 时, 相比于第 1 次切割点 $L_t$ (橘色切点), 负载分布相关参数的变化

首先计算参数$a^u$、$b^u$、$c^u$, 易知:

$$\begin{cases} a^u = b \\ b^u = b^{(1)} \\ c^u = a^u - b^u \end{cases}.$$

然后依据公式 (17)–公式 (19), 计算参数$n_1^u, n_2^u, n_3^u$:

$$\begin{cases} n_1^u = n_1^{(1)} \\ n_2^u = n_2^{(1)} - n_1 \\ n_3^u = n_2 + n_3 - \left(n_1^u + n_2^u\right) \end{cases} \tag{22}$$

同理, 计算参数$n_4^u, n_5^u, n_6^u$:

$$\begin{cases} n_4^u = n_4^{(1)} \\ n_5^u = n_5^{(1)} - n_4 \\ n_6^u = n_4 + n_5 - \left(n_4^u + n_5^u\right) \end{cases}.$$

最后计算参数$avgsel_1^u, \ldots, avgsel_3^u$, 根据已知$avgsel_i^{(1)}$、$n_i^{(1)}$和$n_i^u$等参数, 可知$avgsel_i^u$的计算公式为:

$$avgsel_i^u = \frac{avgsel_i^{(1)} \times n_i^{(1)} - \left|\Delta nx_i^u\right| \times \overline{selectivity}}{n_i^u},\ i = 1,2,3 \tag{23}$$

其中, $nx_i$表示第 $i$ 类查询在负载$W$中的下标, $\Delta nx_i^u$表示切割后第 $i$ 类查询集合的被删除的下标, $\overline{selectivity}$表示$\Delta nx_i^u$查询集合在 $W$ 中对应查询的平均选择度.

如果得到$\Delta nx_i^u$, 就能从负载 $W$ 中计算对应的平均选择度, 从而得到$avgsel_i^u$. 不妨假设$n_i^u$中对应的查询下标为$nx_i^u$, 则$\Delta nx_i^u$就是$nx_i^{(1)}$与$nx_i^u$的差集, 即$\Delta nx_i^u = nx_i^{(1)} - nx_i^u$, 利用公式 (22) 可求出集合$\Delta nx_1^u$、$\Delta nx_2^u$、$\Delta nx_3^u$的具体值, 见公式 (24)–公式 (26), 并将结果代入公式 (23), 即可求出参数$avgsel_1^u, \ldots, avgsel_3^u$.

$$\Delta nx_1^u = nx_1^{(1)} - nx_1^u = nx_1^{(1)} - nx_1^{(1)} = \varnothing \tag{24}$$

$$\Delta nx_2^u = nx_2^{(1)} - nx_2^u = nx_2^{(1)} - \left(nx_2^{(1)} - nx_1\right) = nx_1 \tag{25}$$

$$\begin{aligned} \Delta nx_3^u =& nx_3^{(1)} - nx_3^u = nx_3^{(1)} - (nx_2 + nx_3 - nx_1^u - nx_2^u) \\ =& nx_3^{(1)} - \left(nx_2 + nx_3 - nx_1^{(1)} - nx_2^{(1)} + nx_1\right) \\ =& nx_1^{(1)} + nx_2^{(1)} + nx_3^{(1)} - (nx_1 + nx_2 + nx_3) \end{aligned} \tag{26}$$

将计算的所有参数包括$a^u$、$b^u$、$c^u$和$n_1^u, \ldots, n_6^u$, 以及$avgsel_1^u$、$avgsel_2^u$、$avgsel_3^u$, 代入公式 (21), 即可更新第 1 次切割后频繁模式$L_j = \{a_3, a_4\}$的切割收益, 同理, 第 2 次切割后, 按照此方法快速更新所有频繁模式的切割收益, 依次类推.

## 5 实验设计与结果分析

本节首先介绍数据集、实验环境及对比的基准算法, 然后在数据集上分析不同算法的性能差异由哪些因素引起, 最后从分区性能、执行时间两个指标对比不同分区算法的特点.

### 5.1 实验环境和数据集

#### 5.1.1 运行环境和数据库环境

SCVP 算法适用于各种类型的成本模型, 算法能够根据成本模型的设计得出适应不同环境的分区方案, 本节将以 HDD-I 作为成本模型. 下面介绍实验中使用的模型运行环境、数据库环境和数据导入方式.

(1) 模型运行环境: SCVP 和 SCVP-R 基于 Python 实现, 不依赖于预训练模型和 GPU, 引用 Pandas、Numpy、Scikit-learn 等常见的数据处理和机器学习包, 能够被打包成轻量级的 Python 库, 方便调用.

(2) 数据库环境: PostgresSQL 13.3 (Ubuntu 18.04).

(3) 数据导入方式: 为不同数据集建立独立的模式, 编写 shell 脚本根据预先定义好的表结构 (DDL) 和 Insert 语句, 建立原表, 并导入表数据, 同时, 将负载集中存放到 Workload 表.

#### 5.1.2 数据集和负载

本文在以下 3 个数据集上进行实验.

(1) TPC-H

TPC-H 是决策支持的基准测试, 提供了 22 个复杂的查询, 共有 customer、lineitem、nation、orders、part、partsupp、region、supplier 这 8 张表. 本实验使用 DBGEN 产生了规模系数 (scale factor, SF) 为 1 的表数据, 每张表约含$10^6$条元组, 并将 TPC-H 的 22 条查询作为负载数据, 拆解到 8 张表中的负载分别为 8、17、9、12、8、5、3、10 条查询.

(2) SSB (star schema benchmark)

和 TPC-H 类似, SSB 也是一种基于现实商业应用的数据模型来模拟决策支持的基准测试, 包括 1 个事实表 lineorder 和 4 个维表 customer、part、date、supplier, 以及 13 条标准 SQL 查询测试语句, 包括统计查询、多表关联、复杂条件、group by、order by 等组合方式, 拆分到 5 个表的负载规模分别为 13、7、6、13、10. 在实验中我们生成了规模系数 SF 为 1 的数据集.

(3) 随机数据集

为了测试算法在大规模负载下的性能表现, 额外增加了两组随机数据集, 数据集中每张表均含有 $10^6$ 条元组, 分别为其设置不同的属性维度和负载规模, 来测试算法的执行效率对这些参数的敏感性.

• randomAttr: 表的维度不同, 分别在属性数为 30、50、100、200、500 的关系表上生成 1 000 条查询.

• randomQue: 负载规模不同, 分别在 50 个属性的关系表上生成 50、200、500、1 000、5 000 个查询.

在随机数据集上进行实验时, 通常会设置多份数据集, 每份数据集包括 randomAttr、randomQue 两组数据, 取所有数据集的平均值作为最终的测试值. 下面介绍随机数据集的生成策略: 为了能够让算法划分尽可能多的列组, 对查询的随机生成制定如下规则.

• 设置具有明显特征的查询: 少部分查询作为高频查询, 其 *frequency* 参数为负载规模的 1/10 到 1/8, 其他查询的 *frequency* 参数为 1–20.

• 获得较多的分区数: 将给定的属性范围划分为等分的 4–6 个区间, 并对每个区间适当延长, 使彼此有重叠部分, 如设置重叠比例为 10%. 随机产生一个分布在 [4, 6] 的整数, 则下一个查询访问的属性范围需位于该整数对应的区间内.

• 查询的其他参数的生成策略: *selectivity* 随机分布在 0.01–0.05, scan key 参数从访问属性列表中随机选择. 参数 accessed columns 确定范围如 [$a$, $b$] 后, 随机产生 1–10 个遵循 $u=\lfloor(a+b)/2\rfloor$、$\sigma^2=(b-u)\times\eta$ 的高斯分布的访问属性, 其中 $\eta$=0.2.

5.1.3 负载数据的流入

模拟软件系统发送查询请求, 计划从 workload 表中, 依次读取对应测试数据集的查询, 并将读取时间作为查询到达时间, 待所有查询读取完毕, 向负载收集器发送分区信号, 执行分区操作.

### 5.2 性能评估

5.2.1 对比的 baseline 算法

本文选取了 10 个经典的垂直分区算法用于对比实验, 其中前 3 个不基于成本模型生成垂直分区, 而后 7 个基于成本模型生成垂直分区. 基于成本模型的算法一般比不基于成本模型的算法效果要好.

• Pure_NSM: 将原表单独作为一个分区, 分区数为 1.

• Pure_DSM: 每个属性都作为一个单独的分区, 分区数为属性数.

• Rodriguez[21]: 将频繁项集挖掘应用到亲和度矩阵, 建立属性间亲和度的支持度, 依据筛选的亲和度按照从高到低依次生成垂直分区.

• HillsClimb[1]: 自底向上的算法, 从 Pure_DSM 分区出发开始迭代, 在每次迭代中, 算法都会合并两个分区, 使合并后在预期的查询开销方面会有最好的改善, 对应的分区数减 1. 当合并任意两个分区都不会使当前的查询开销改善时, 算法停止迭代.

• HYF[22]: 使用 Apriori 算法, 生成属性的频繁项集, 按照频繁模式的长度从高到低依次选择不重叠的项集作为分区方案.

• NAVATHE[10]: 自顶向下的算法, 构建属性的属性亲和矩阵, 使用 BEA 算法对矩阵单元进行聚类, 然后将聚集的属性集递归地拆分为垂直分区.

• AutoPart[6]: 自底向上的算法, 首先根据查询中引用的属性组, 生成原子分区, 然后按照 HillsClimb 的模式进行迭代.

• Trojan[9]: 自底向上的算法, 提出一种基于列组兴趣度的度量方法, 只保留兴趣度高的列组, 然后对列组进行0-1 规划选择兴趣度最高但不重叠的垂直分区方案.

• O2P[13]: 自顶向下的算法, 在 NAVATHE 的算法基础上, 使其适用在线分区, 根据负载增加情况, 动态更新查询的关联矩阵, 使用动态规划在每个阶段都得到最佳的垂直分区.

• Hyrise[7]: 自底向上的算法, 根据查询中引用的属性组, 生成原子分区, 建立原子分区的亲和图并切割成子图, 然后对子图的原子分区进行随机合并, 直到成本不再改进为止.

5.2.2 基准测试数据上的算法质量比较

在基准数据集 TPC-H、SSB 上进行实验, 首先从 HDD-I 成本模型计算的磁盘 I/O 访问成本来比较各类算法的性能, 然后通过引入另外两个分区指标: 不必要数据的读比例和元组重建平均次数来测量各类算法的分区特性.

(1) 磁盘 I/O 访问成本

图 9 表示在 TPC-H 提供的 22 条标准 SQL 上比较不同算法计算的优化方案, 分别在 TPC-H 中 8 个表的访问成本 (标准化), 表 2 显示了 TPC-H 数据集的平均访问成本. 总体来看, 基于成本模型的算法中, HillsClimb、SCVP 和 SCVP-R 在所有表中均得到了最优分区, 平均性能最高, 成本达到 7 930.125; 次优的算法是 Hyrise, 达到 8 109, 且 AutoPart 与 Hyrise 接近; O2P 平均性能最差, 为 10 311. 在独立于成本模型的算法中, Pure_DSM (列存) 性能最好, Pure_NSM (行存) 性能最差. 从单个表来看, 表的属性维度对性能有影响, 例如在 nation、part、region 这 3 个属性维度较小的表中, 各分区算法性能差距不大, 基本上都能得到最佳方案, 而 lineitem 表属性较多, 算法间的性能差距也比较大.

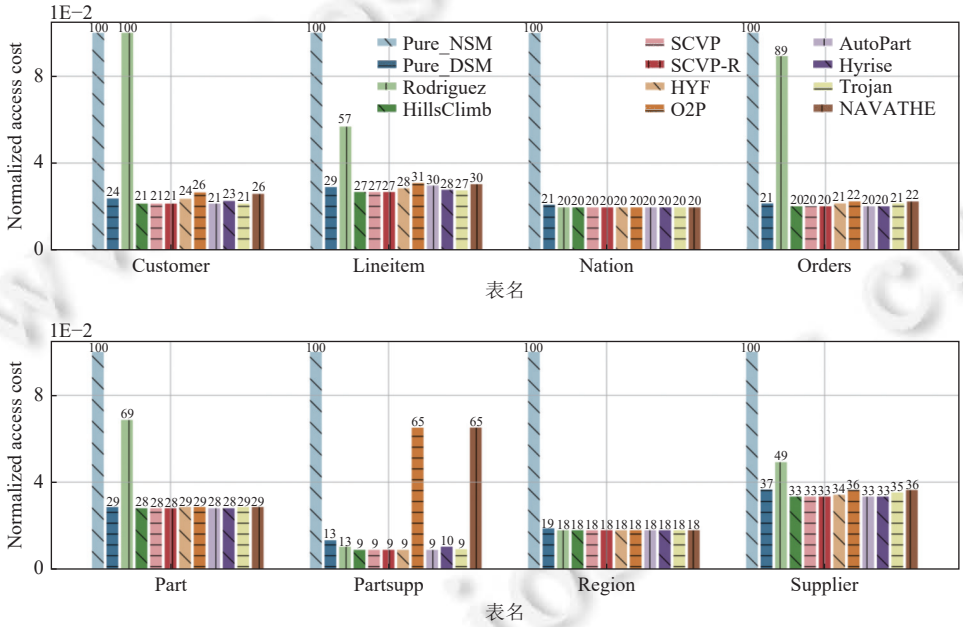



图 9 TPC-H 数据集的算法性能比较 (cost model: HDD-I, benchmark: TPC-H)

表 2 TPC-H 数据集各算法的平均 I/O 成本比较

| 算法 | Pure_NSM | Pure_DSM | HillsClimb | Rodriguez | HYF | SCVP/SCVP-R | O2P | AutoPart | Hyrise | Trojan | NAVATHE |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Avg. I/O (块) | 33 700 | 8 645 | **7 930** | 19 050 | 8 288 | **7 930** | 10 311 | 8 179 | 8 109 | 8 171 | 10 241 |

图 10 表示在 SSB 数据集中的 13 条标准 SQL 上比较不同算法计算的优化方案, 分别在 SSB 中 5 个表的估计访问成本 (归一化), 表 3 显示了 SSB 数据集的平均访问成本. SSB 数据集与 TPC-H 数据集类似, 查询负载规模不大, 负载特征比较典型, 区别在于 SSB 负载中查询的选择度都为 1. 从总体性能上看, 基于成本模型的算法中,

HillsClimb 和 SCVP-R 相比于其他算法表现最好, 在各张表中均达到最优性能, 平均性能为 5 504; SCVP 和 AutoPart 仅在 lineorder 表略低于 HillsClimb、SCVP-R, 其他表完全一致, 平均性能为 5 536; 排名第 3 的方法是 HYF, 平均性能为 5 568; 而 O2P 性能表现最差, 为 6 488. 在独立于成本模型的算法中, Rodriguez 性能最好, Pure_NSM 性能最差. 从单个表来看, 在属性维度最大的 lineorder 表, 各种算法优化效果差异最明显. 由于 TPC-H 和 SSB 数据集的表维度和查询规模都比较小, 导致分区特征明显, 因此测试结果显示各个算法的访问成本比较接近, 得到相似的分区方案, 但也在一定程度上反映 HillsClimb、SCVP-R、SCVP 相对于其他算法, 性能更好, 表现更稳定.

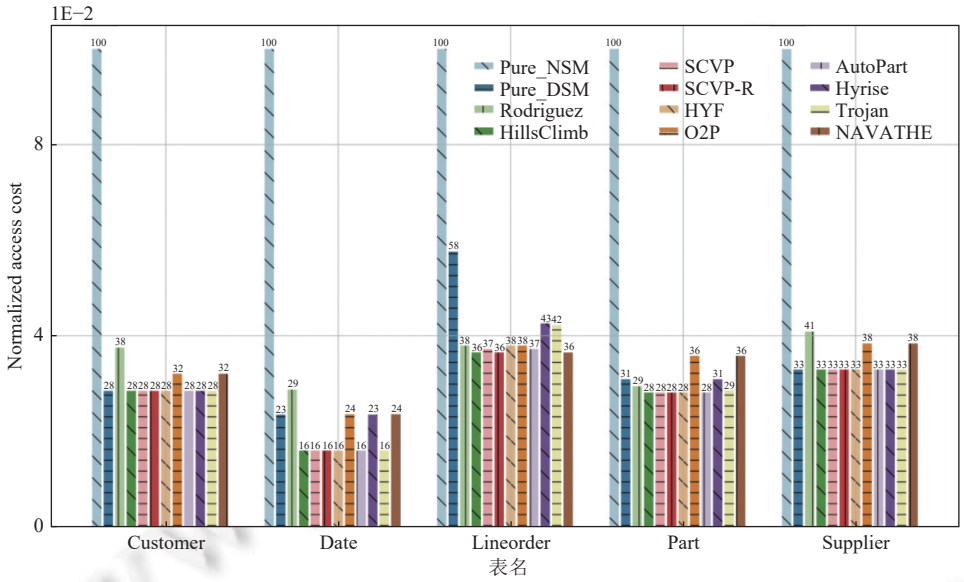



图 10 SSB 数据集的算法性能比较 (cost model: HDD-I, benchmark: SSB)

表 3 SSB 数据集各算法的平均 I/O 成本比较

| 算法 | Pure_NSM | Pure_DSM | HillsClimb | Rodriguez | HYF | SCVP/SCVP-R | O2P | AutoPart | Hyrise | Trojan | NAVATHE |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Avg. I/O (块) | 19 568 | 6 884 | **5 504** | 6 880 | 5 568 | 5 536/**5 504** | 6 488 | 5 536 | 6 212 | 5 772 | 6 424 |

(2) 不必要数据的平均读比例

Agrawal 等人[29]提出一种列组有效性 (column group effectiveness) 的指标, 即所有查询访问某个分区时读的数据占分区内所有数据的比例, 这是从访问数据的角度评价分区优劣的指标, 即列组有效性越高, 每个分区 (列组) 中对不必要数据的平均读比例越低. 不必要数据的平均读比例计算公式如下:

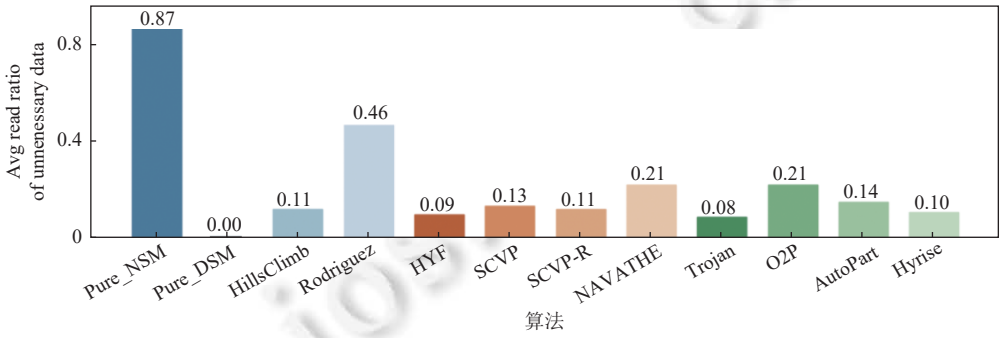
$$E_{\text{column group}}=\frac{\sum_{c\in g} width(c)\times|Occurrence(c)\times\overline{sel}|}{\sum_{c\in g} width(c)\times\left|Y_{c\in g}Occurrence(c)\right|} \tag{27}$$

$$Fraction_{\text{unnecessary data}}=1-E_{\text{column group}} \tag{28}$$

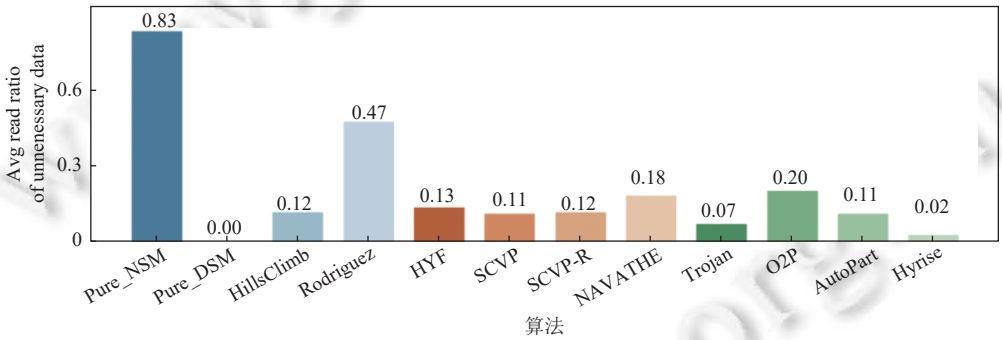
其中, $c$ 代表列组 $g$ 中的一列, $width(c)$ 是属性 $c$ 的平均宽度 (属性 $c$ 可能是变长字段), $Occurrence(c)$ 是负载中包含属性 $c$ 的查询集合, $\overline{sel}$ 表示所有以 $c$ 为扫描键的 query 的平均 *selectivity*, $\left|Y_{c\in g}Occurrence(c)\right|$ 表示负载中所有访问列组 $g$ 的查询集合, $Fraction_{\text{unnecessary data}}$ 表示列组不必要数据的平均读比例.

本文使用不必要数据的平均读比例测量不同算法分区方案的特性, 分析该指标与算法性能的关系. 图 11 展示了在 TPC-H 和 SSB 数据集上测试不同算法所有分区的不必要读比例的平均值. 由于 Pure_DSM (列存) 使单列作

为一个分区, 不必要读比例一定是 0, 对应的列组平均有效性为 1, 同理 Pure_NSM (行存) 读取了最多的不必要数据; 在 TPC-H 数据集上, 不必要读比例处在 0.11、0.13、0.11 时, 对应 HillsClimb、SCVP 和 SCVP-R 算法的分区方案, 整体达到了最佳的性能, 此时读取的不必要数据至少占总数据的 11%; 而对于 SSB 数据集, 不必要读比例为 0.12 时, 对应 HillsClimb 和 SCVP-R 算法, 此时至少需要读取 12% 比例的不必要数据使数据库达到最佳性能. 可以看出, 不必要数据的平均读比例与成本模型的访问成本并不呈正相关关系, 例如 SSB 数据集上, Hyrise 算法的不必要读比例为 0.02, 仅次于 Pure_DSM, 但图 9 中其性能排名第 7, 且要强于 Pure_DSM. 因此, 不必要数据读比例越低, 访问成本不代表越低, 这是由于数据库环境的复杂性, 不必要数据的平均读比例降低了, 但可能导致元组重建成本增加或者大量不必要数据集中在无索引的分区中, 因此该指标并不代表算法的实际性能, 但性能排名靠前的算法不必要读比例都处于相对较低值, 可见从一定角度该指标能反映出不同分区方案对查询执行速度的增益值.




(a) Cost model: HDD-I, benchmark: TPC-H




(b) Cost model: HDD-I, benchmark: SSB

图 11 基准数据集上算法不必要数据的平均读比例比较

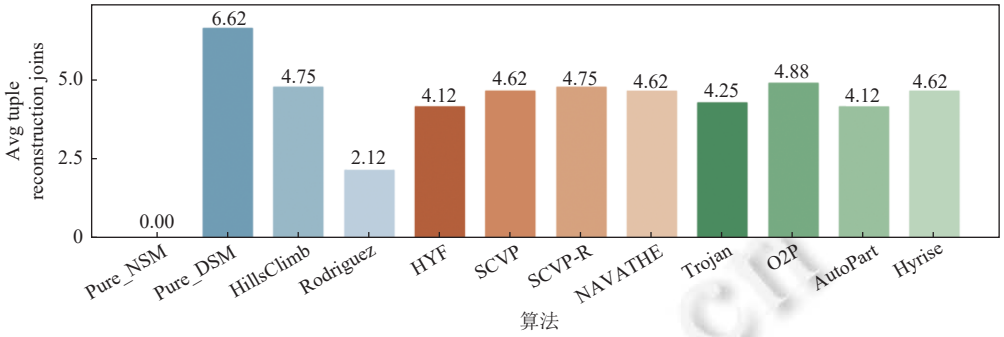
(3) 元组平均重建次数

分区数量越多, 元组重建的次数也会增多, 元组重建平均次数能很好地评判算法给出的优化方案中分区数量的大小, 计算公式如下:

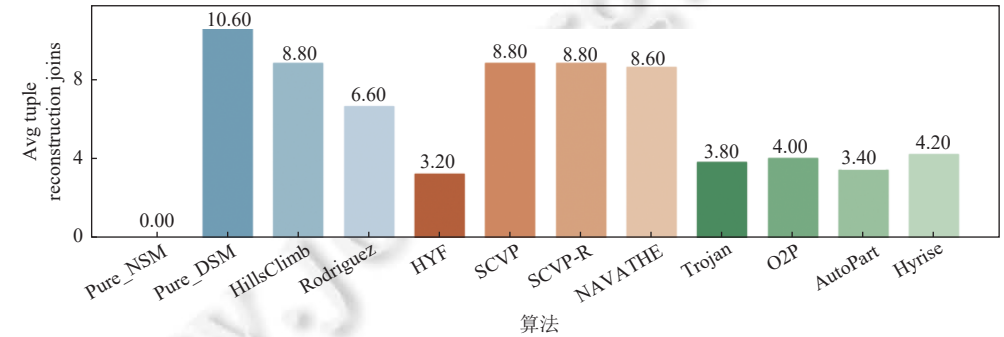
$$N_{\text{Tuple-reconstruction joins}} = N_{\text{vertical partitions}} - 1 \tag{29}$$

本文主要探究不同算法的性能是否和分区数量有关系. 图 12 表示在 TPC-H 和 SSB 数据集上测试的各类算法的元组平均重建次数. TPC-H 数据集上各种算法的平均分区数差异不大, 稳定在 4–5 之间; 而 SSB 数据集上刚好相反, 不同算法差异较大, 分布在 3–9 之间. 结合图 9 和图 10 可以看出, 元组平均重建次数的排名和算法性能排名并无直接联系, 重建成本的大小并不能成为算法性能的主导因素, 但从差异程度来看, 却有直接联系, 例如 TPC-H 数据集的各种算法性能差异也要明显小于 SSB 数据集, 这与不同算法重建次数的差异相对应, 说明 SSB 数据集的负载特征相对于 TPC-H 更复杂, 导致不同算法的分区策略出现明显差异, 最终的分区数量和性能都表现出明显

的不同. 因此不同算法性能的差异不仅仅反映在分区内容上, 一定程度取决于能否在具体的负载环境下找到一个合适的分区数目.




(a) Cost model: HDD-I, benchmark: TPC-H




(b) Cost model: HDD-I, benchmark: SSB

图 12 基准数据集中不同算法的元组平均重建次数比较

5.2.3 随机数据集上的算法质量比较

本小节将在 randomAttr、randomQue 两组随机数据集进行实验, 为了减少数据集的随机性对结果造成影响, 每次实验都对两组数据集随机生成 5 份, 取 5 次测试的平均值作为最终的指标值.

5.2.3.1 属性维度

图 13 展示了 randomAttr 数据集的测试结果 (标准化), 表 4 显示了 randomAttr 数据集的平均访问成本. 整体来看, SCVP-R 性能最好, 平均访问成本为 355 859; HillsClimb 与 SCVP 次优, 平均访问成本为 356 447、364 047; HYF、AutoPart 两种算法比较接近, 其中 HYF 性能较好, 达到 417 656. Pure_DSM 虽然不依赖于成本模型, 但也达到了不错的性能, 接近于 O2P, 原因在于列存有效降低了非必要数据的读取比例. 从属性维度上看, 随着维度增加, 算法的性能差异变大, 例如 30 维属性时, 不同算法性能差异在 0–11% 之间, 而 150 维时, 差异扩大到 0–50%, 其中 HillsClimb、SCVP 和 SCVP-R 的差异变化幅度不大, 始终维持在 2% 以内.

下面对性能接近的 3 种算法 HillsClimb、SCVP 和 SCVP-R 进一步比较, 通过对属性的维度进一步细化, 探究它们的性能和优化时间的差异.

图 14 展示了 3 种算法的性能差异, 总体来看, SCVP-R 的性能最佳, 平均访问成本为 360 545.5, HillsClimb 的平均访问成本为 360 994.6, 接近于 SCVP-R, 而 SCVP 性能稍微差一些, 为 369 193.5. 以 Pure_DSM 的访问成本为基准来看, SCVP-R 和 SCVP 分别达到 HillsClimb 性能的 100.5%、92%, 值得注意的是, 在中等维度 (如 80 个属性及以下) 的表, 3 种算法的性能差异仅保持在 1.073%, 说明在一定属性维度的限制下, SCVP 基本达到 SCVP-R 和 HillsClimb 的性能表现.

图 15 对比了不同属性维度, 3 种算法的优化时间, 在表 5 显示了具体的数值. 随着属性维度增大, 算法执行时间都在延长, 但 HillsClimb 和 SCVP-R 都接近于以 2 为底的指数倍增大, 但比例系数相差很大; 而 SCVP

更趋向于线性增长态势. 总体来看, HillsClimb 的平均优化时间分别是 SCVP 和 SCVP-R 的 226.0 倍、17.5 倍, 且随着维度增大, 算法的执行时间差距不断扩大. 对于维度不限制的环境下, SCVP 的执行时间对属性维度不敏感, 更适合在线环境, 如果表属性能够维持在 100 维以下, SCVP-R 的执行时间将保持较低的增长速度, 也能用于在线环境.

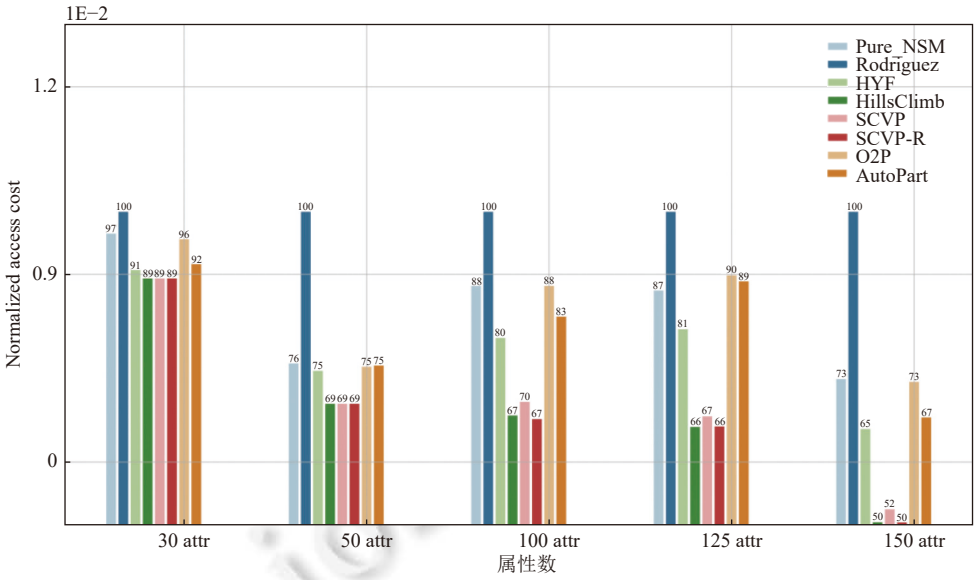



图 13 在 randomAttr 数据集中不同算法的平均性能比较 (cost model: HDD-I, benchmark: randomAttr)

表 4 randomAttr 数据集各算法的平均 I/O 成本比较

| 算法 | Pure_DSM | Rodriguez | HYF | HillsClimb | SCVP | SCVP-R | O2P | AutoPart |
|---|---|---|---|---|---|---|---|---|
| Avg. I/O (块) | 451 597 | 548 223 | 417 656 | 356 447 | 364 047 | **355 859** | 452 418 | 434 414 |

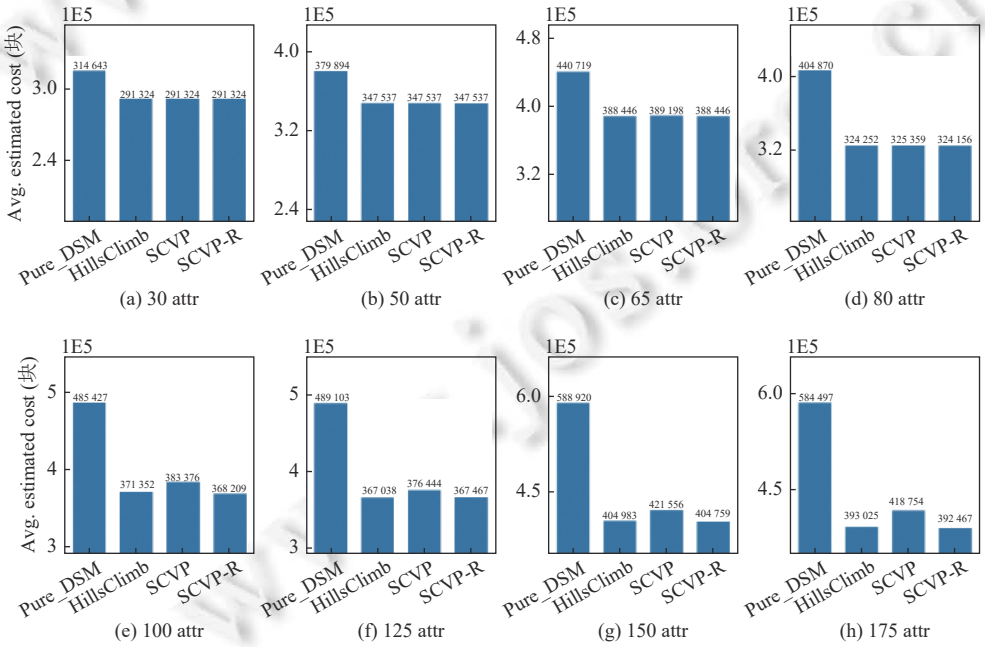



图 14 randomAttr 数据集扩大到 8 维时, 4 种算法的平均性能比较

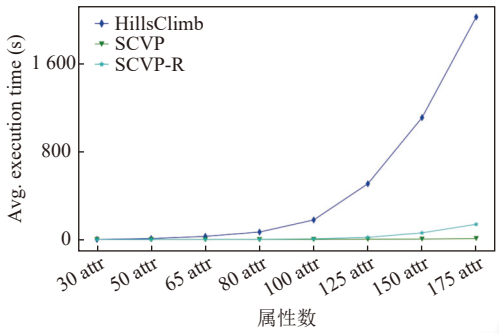



图 15　在 randomAttr 数据集中 HillsClimb、SCVP 和 SCVP-R 算法的平均优化时间比较

表 5　在 randomAttr 数据集中 HillsClimb、SCVP 和 SCVP-R 算法的平均优化时间 (s)

| 算法 | 30 attr | 50 attr | 65 attr | 80 attr | 100 attr | 125 attr | 150 attr | 175 attr |
|---|---|---|---|---|---|---|---|---|
| HillsClimb | 1.20 | 9.00 | 29.20 | 68.20 | 178.00 | 506.40 | 1 106.80 | 2 020.00 |
| SCVP | 0.16 | 0.28 | 0.50 | 0.56 | 1.08 | 2.33 | 3.52 | 8.91 |
| SCVP-R | 0.16 | 0.47 | 1.29 | 1.56 | 5.66 | 19.29 | 59.75 | 136.11 |

5.2.3.2　负载规模

图 16 展示了不同算法在 randomQue 数据集的测试情况, 表 6 显示了 randomQue 数据集的平均访问成本. 总体来看, SCVP-R 和 HillsClimb 平均性能最佳, 在各个数据集中性能均达到最优, 平均访问成本为 563 819, 与之接近的是 SCVP 算法为 564 101. O2P 和 Hyrise 表现较差, 仅比 Pure_DSM 略优. 从负载规模来看, 随着查询数量增多, 各类算法性能差异基本稳定, 说明属性维度在对各类算法性能差异的影响程度上, 相比于负载规模更具有决定性作用. 从属性维度来看, 50 维的数据集上 HillsClimb、SCVP-R 和 SCVP 性能整体接近, 偏差小于 1%.

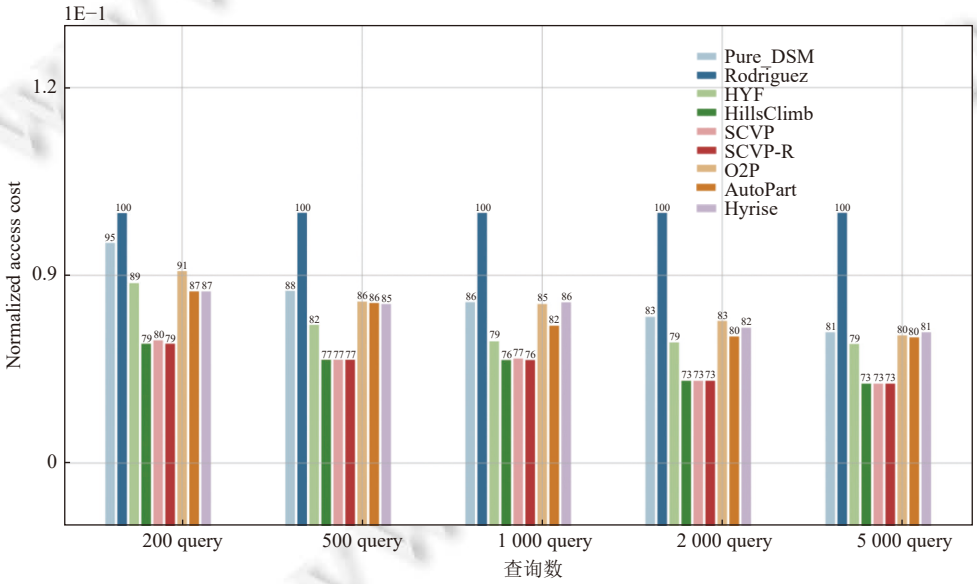



图 16　在 randomQue 数据集中不同算法的平均性能比较 (cost model: HDD-I, benchmark: randomQue)

下面对性能最好的 3 类算法 HillsClimb、SCVP 和 SCVP-R 进一步比较, 通过对增加更多负载的规模, 探究它们的性能和优化时间的差异.

图 17 展示了 3 种算法的性能差异, 所有的工作负载都在 50 维的属性表执行. 可见, 在低维表下, 无论在何种

负载规模, HillsClimb、SCVP-R 都达到了同等的性能, 平均访问成本为 687 700.8; SCVP 的性能十分接近前两种算法, 平均访问成本为 689 643.6, 实验结果再次验证了低维属性下 3 种算法拥有同等的性能表现, 因此, 在低纬属性表中, 随着负载规模的增大, 算法优化时间将成为 3 种算法重要的评价标准.

表 6 randomQue 数据集各算法的平均 I/O 成本比较

| 算法 | Pure_DSM | Rodriguez | HYF | HillsClimb | SCVP | SCVP-R | O2P | AutoPart | Hyrise |
|---|---|---|---|---|---|---|---|---|---|
| Avg. I/O (块) | 633 362 | 766 846 | 609 598 | **563 819** | 564 101 | **563 819** | 628 358 | 619 215 | 628 365 |

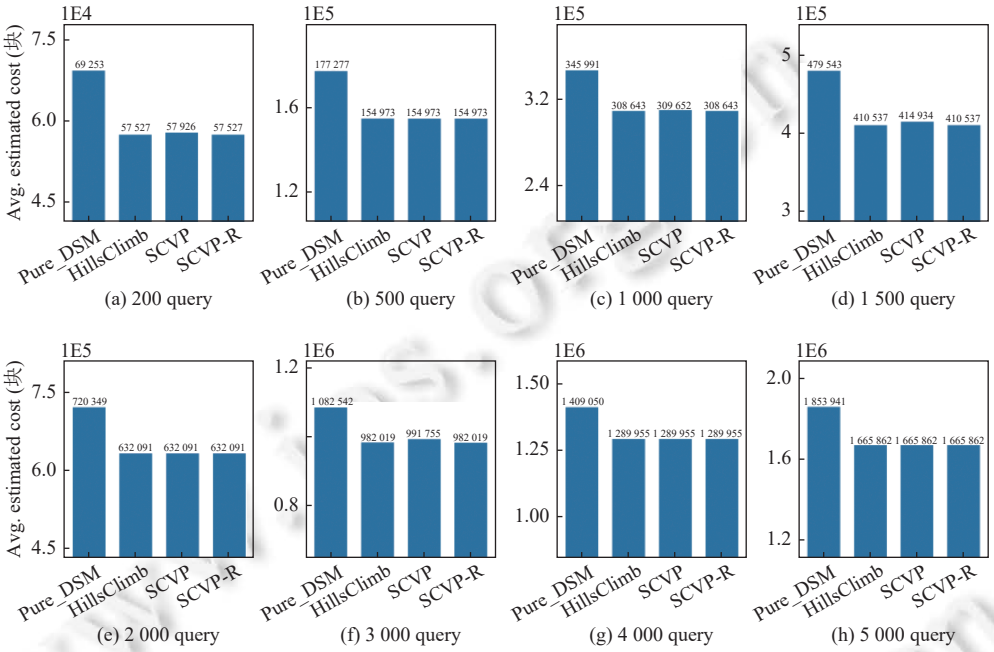



图 17 randomQue 数据集扩大到 8 种规模, 4 种算法的平均性能比较

如图 18 显示了 HillsClimb 和 SCVP 算法在 8 个负载规模下的平均优化时间, 表 7 显示了具体的数值. 从负载规模来看, 随着负载规模的增大, 3 种算法均呈线性增长, 且 HillsClimb 的增长系数远大于 SCVP 和 SCVP-R, 由于负载规模增大时, 对成本模型的执行时间将有较大影响, 是算法优化时间增加的主要因素, 因此图 18 中的增长系数很大程度能反映出对各类算法对成本模型的调用次数, 即 HillsClimb>SCVP-R>SCVP. 从总体执行时间看, HillsClimb 的平均优化时间分别是 SCVP 和 SCVP-R 的 35.6 倍、16.5 倍, 且随着负载规模增大, HillsClimb 与 SCVP、SCVP-R 的执行时间差异不断加大.

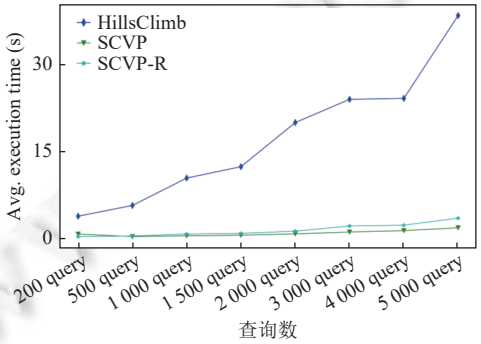



图 18 在 randQue 数据集中 HillsClimb、SCVP 和 SCVP-R 算法的平均优化时间比较

表 7 在 randQue 数据集中 HillsClimb、SCVP 和 SCVP-R 算法的平均优化时间 (s)

| 算法 | 200 query | 500 query | 1 000 query | 1 500 query | 2 000 query | 3 000 query | 4 000 query | 5 000 query |
|---|---|---|---|---|---|---|---|---|
| HillsClimb | 3.82 | 5.67 | 10.40 | 12.37 | 19.98 | 23.99 | 24.15 | 38.46 |
| SCVP | 0.15 | 0.17 | 0.26 | 0.32 | 0.44 | 0.64 | 0.79 | 1.13 |
| SCVP-R | 0.19 | 0.27 | 0.52 | 0.60 | 0.87 | 1.64 | 1.71 | 2.61 |

## 6 总结与展望

为了解决现有垂直分区算法应用于在线环境时, 存在执行时间过长、性能差等无法满足实时应用需求的难题, 本文提出一个基于谱聚类的在线垂直分区算法 SCVP. 它首先通过增加约束条件缩小解空间, 然后对解空间设计算法进行精细的搜索. 为了进一步提升 SCVP 在高维属性下的性能, 进一步提出了 SCVP 的改进版本 SCVP-R. SCVP-R 通过引入同域竞争机制、双败淘汰机制和循环机制, 对 SCVP 在分区优化过程中的合并方案进行了优化. 在不同数据集上的实验结果表明, 相比于目前最好的垂直分区方法, SCVP 和 SCVP-R 有着更快的执行时间和更好的性能表现. 目前 SCVP 基于传统机器学习方法, 未来将尝试使用图神经网络以及深度强化学习技术[30,31]来解决分区问题. 另外将重点研究在线分区系统中分区触发器的设计, 借鉴控制领域中的 PID 控制器[32]和最优自适应控制器[33], 来决定离散时间环境下容纳多少事务特征、何时执行分区操作.

**References:**


[1] Hankins RA, Patel JM. Data morphing: An adaptive, cache-conscious storage technique. In: Proc. of the 29th Int'l Conf. on Very Large Data Bases. Berlin: VLDB Endowment, 2003. 417–428.

[2] Rodríguez L, Li XO, Cuevas-Rasgado AD, García-Lamont F. DYVEP: An active database system with vertical partitioning functionality. In: Proc. of the 10th IEEE Int'l Conf. on Networking, Sensing and Control (ICNSC). Evry: IEEE, 2013. 457–462. [doi: 10.1109/ICNSC.2013.6548782]

[3] Li LZ, Gruenwald L. SMOPD-C: An autonomous vertical partitioning technique for distributed databases on cluster computers. In: Proc. of the 15th IEEE Int'l Conf. on Information Reuse and Integration. Redwood City: IEEE, 2014. 171–178. [doi: 10.1109/IRI.2014.7051887]

[4] Sun LW, Franklin MJ, Krishnan S, Xin RS. Fine-grained partitioning for aggressive data skipping. In: Proc. of the 2014 ACM SIGMOD Int'l Conf. on Management of Data. Utah: ACM, 2014. 1115–1126. [doi: 10.1145/2588555.2610515]

[5] Shanbhag A, Jindal A, Madden S, Quiane J, Elmore AJ. A robust partitioning scheme for ad-hoc query workloads. In: Proc. of the 2017 Symp. on Cloud Computing. California: ACM, 2017. 229–241. [doi: 10.1145/3127479.3131613]

[6] Papadomanolakis S, Ailamaki A. AutoPart: Automating schema design for large scientific databases using data partitioning. In: Proc. of the 16th Int'l Conf. on Scientific and Statistical Database Management. Santorini Island: IEEE Computer Society, 2004. 383–392.

[7] Grund M, Krüger J, Plattner H, Zeier A, Cudre-Mauroux P, Madden S. Hyrise: A main memory hybrid storage engine. Proc. of the VLDB Endowment, 2010, 4(2): 105–116. [doi: 10.14778/1921071.1921077]

[8] Karypis G, Kumar V. Analysis of multilevel graph partitioning. In: Proc. of the 1995 ACM/IEEE Conf. on Supercomputing (Supercomputing1995). San Diego: IEEE, 1995. 29. [doi: 10.1109/SUPERC.1995.242800]

[9] Jindal A, Quiané-Ruiz JA, Dittrich J. Trojan data layouts: Right shoes for a running elephant. In: Proc. of the 2nd ACM Symp. on Cloud Computing. Cascais: ACM, 2011. 21. [doi: 10.1145/2038916.2038937]

[10] Navathe S, Ceri S, Wiederhold G, Dou JL. Vertical partitioning algorithms for database design. ACM Trans. on Database Systems, 1984, 9(4): 680–710. [doi: 10.1145/1994.2209]

[11] McCormick Jr WT, Schweitzer PJ, White TW. Problem decomposition and data reorganization by a clustering technique. Operations Research, 1972, 20(5): 993–1009. [doi: 10.1287/opre.20.5.993]

[12] Navathe SB, Ra M. Vertical partitioning for database design: A graphical algorithm. In: Proc. of the 1989 ACM SIGMOD Int'l Conf. on Management of Data. Oregon: ACM, 1989. 440–450. [doi: 10.1145/67544.66966]

[13] Jindal A, Dittrich J. Relax and let the database do the partitioning online. In: Proc. of the 5th Int'l Workshop on Business Intelligence for the Real-time Enterprise. Seattle: Springer, 2011. 65–80. [doi: 10.1007/978-3-642-33500-6_5]

[14] Hoffer JA. An integer programming formulation of computer data base design problems. Information Sciences, 1976, 11(1): 29–48. [doi:

10.1016/0020-0255(76)90035-9]

[15] Cornell DW, Yu PS. An effective approach to vertical partitioning for physical design of relational databases. IEEE Trans. on Software Engineering, 1990, 16(2): 248–258. [doi: 10.1109/32.44388]

[16] Cheng CH, Lee WK, Wong KF. A genetic algorithm-based clustering approach for database partitioning. IEEE Trans. on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 2002, 32(3): 215–230. [doi: 10.1109/TSMCC.2002.804444]

[17] Song SK, Gorla N. A genetic algorithm for vertical fragmentation and access path selection. The Computer Journal, 2000, 43(1): 81–93. [doi: 10.1093/comjnl/43.1.81]

[18] Ng V, Law DM, Gorla N, Chan CK. Applying genetic algorithms in database partitioning. In: Proc. of the 2003 ACM Symp. on Applied Computing. Melbourne: ACM, 2003. 544–549. [doi: 10.1145/952532.952639]

[19] Gorla N, Betty PWY. Vertical fragmentation in databases using data-mining technique. Int'l Journal of Data Warehousing and Mining (IJDWM), 2008, 4(3): 35–53. [doi: 10.4018/jdwm.2008070103]

[20] Agrawal R, Imieliński T, Swami A. Mining association rules between sets of items in large databases. In: Proc. of the 1993 ACM SIGMOD Int'l Conf. on Management of Data. Washington: ACM, 1993. 207–216. [doi: 10.1145/170035.170072]

[21] Rodriguez L, Li XO. A vertical partitioning algorithm for distributed multimedia databases. In: Proc. of the 22nd Int'l Conf. on Database and Expert Systems Applications. Toulouse: Springer, 2011. 544–558. [doi: 10.1007/978-3-642-23091-2_48]

[22] Huang YF, Lai CJ. Integrating frequent pattern clustering and branch-and-bound approaches for data partitioning. Information Sciences, 2016, 328: 288–301. [doi: 10.1016/j.ins.2015.08.047]

[23] Arulraj J, Pavlo A, Menon P. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In: Proc. of the 2016 Int'l Conf. on Management of Data. California: ACM, 2016. 583–598. [doi: 10.1145/2882903.2915231]

[24] Shukla S, Naganna S. A review on K-means data clustering approach. Int'l Journal of Information & Computation Technology, 2014, 4(17): 1847–1860.

[25] Von Luxburg U. A tutorial on spectral clustering. Statistics and Computing, 2007, 17(4): 395–416. [doi: 10.1007/s11222-007-9033-z]

[26] Caliński T, Harabasz J. A dendrite method for cluster analysis. Communications in Statistics, 1974, 3(1): 1–27.

[27] Han JW, Pei J, Yin YW, Mao RY. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data Mining and Knowledge Discovery, 2004, 8(1): 53–87. [doi: 10.1023/B:DAMI.0000005258.31418.83]

[28] Son JH, Kim MH. An adaptable vertical partitioning method in distributed systems. Journal of Systems and Software, 2004, 73(3): 551–561. [doi: 10.1016/j.jss.2003.04.002]

[29] Agrawal S, Narasayya V, Yang B. Integrating vertical and horizontal partitioning into automated physical database design. In: Proc. of the 2004 ACM SIGMOD Int'l Conf. on Management of Data. Paris: ACM, 2004. 359–370. [doi: 10.1145/1007568.1007609]

[30] Durand GC, Pinnecke M, Piriyev R, Mohsen M, Broneske D, Saake G, Sekeran MS, Rodriguez F, Balami L. Gridformation: Towards self-driven online data partitioning using reinforcement learning. In: Proc. of the 1st Int'l Workshop on Exploiting Artificial Intelligence Techniques for Data Management. Houston: ACM, 2018. 1. [doi: 10.1145/3211954.3211956]

[31] Yang ZH, Chandramouli B, Wang C, Gehrke J, Li YN, Minhas UF, Larson PÅ, Kossmann D, Acharya R. Qd-tree: Learning data layouts for big data analytics. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 193–208. [doi: 10.1145/3318464.3389770]

[32] Chen KJ, Zhou YL, Cao Y. Online data partitioning in distributed database systems. In: Proc. of the 18th Int'l Conf. on Extending Database Technology. Brussels: EDBT, 2015. 1–12.

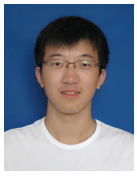
[33] Lewis FL, Vrabie D, Vamvoudakis KG. Reinforcement learning and feedback control: Using natural decision methods to design optimal adaptive controllers. IEEE Control Systems Magazine, 2012, 32(6): 76–105. [doi: 10.1109/MCS.2012.2214134]

**刘鹏举**(1997－), 男, 博士生, 主要研究领域为智能数据分区, 组合优化, 负载预测, 强化学习.

**李好洋**(1999－), 男, 博士生, 主要研究领域为数据库分区优化, 自然语言处理.

**王天一**(1999－), 女, 硕士生, 主要研究领域为数据库分区与布局, 机器学习.

**刘欢**(1997－), 男, 硕士生, 主要研究领域为数据分区, 机器学习, 图神经网络.



**孙路明**(1994－), 男, 博士生, 主要研究领域为机器学习, 数据库系统.

**任逸飞**(1987－), 男, 高级工程师, 主要研究领域为 NoSQL 数据库, 主存数据库, 分布式键值存储引擎.

**李翠平**(1971－), 女, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为社交网络分析, 社会推荐, 大数据分析及挖掘.

**陈红**(1965－), 女, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为数据库技术, 新硬件平台下的高性能计算.