软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(10):3819–3843 [doi: 10.13328/j.cnki.jos.006384]

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

智能数据分区与布局研究*

刘 欢[1,2], 刘鹏举[1,2], 王天一[1,2], 何雨琪[2], 孙路明[1,2], 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)

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

摘 要: 大数据时代, 数据规模庞大, 由数据进行驱动的应用分析场景日益增多. 如何快速、高效地从这些海量数据中提取出用以分析决策的信息, 给数据库系统带来重大挑战. 同时, 现代商业分析决策对分析数据的实时性要求数据库系统能够同时快速处理 ACID 事务和复杂的分析查询. 然而, 传统的数据分区粒度太粗, 且不能适应动态变化的复杂分析负载; 传统的数据布局单一, 不能应对现代大量增加的混合事务分析应用场景. 为了解决以上问题, “智能数据分区与布局”成为当前的研究热点之一, 它通过数据挖掘、机器学习等技术抽取工作负载的有效特征, 设计最佳的分区策略来避免扫描大量不相关的数据, 指导布局结构设计以适应不同类型的工作负载. 首先介绍了智能数据分区与布局的相关背景知识, 然后对智能数据分区与布局技术的研究动机、发展趋势、关键技术进行详细的阐述. 最后, 对智能数据分区与布局技术的研究前景做出总结与展望.

关键词: 数据库系统; 分区策略; 布局策略; 机器学习

中图法分类号: TP311

中文引用格式: 刘欢, 刘鹏举, 王天一, 何雨琪, 孙路明, 李翠平, 陈红. 智能数据分区与布局研究. 软件学报, 2022, 33(10): 3819–3843. http://www.jos.org.cn/1000-9825/6384.htm

英文引用格式: Liu H, Liu PJ, Wang TY, He YQ, Sun LM, Li CP, Chen H. Survey of Intelligent Partition and Layout Technology in Database System. Ruan Jian Xue Bao/Journal of Software, 2022, 33(10): 3819–3843 (in Chinese). http://www.jos.org.cn/1000-9825/6384.htm

Survey of Intelligent Partition and Layout Technology in Database System

LIU Huan[1,2], LIU Peng-Ju[1,2], WANG Tian-Yi[1,2], HE Yu-Qi[2], SUN Lu-Ming[1,2], 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)

Abstract: In the era of big data, there are more and more application analysis scenarios driven by large-scale data. How to quickly and efficiently extract the information for analysis and decision-making from these massive data brings great challenges to the database system. At the same time, the real-time performance of analysis data in modern business analysis and decision-making requires that the database system can process ACID transactions and complex analysis queries. However, the traditional data partition granularity is too coarse, and cannot adapt to the dynamic changes of complex analysis load; the traditional data layout is single, and cannot cope with the modern increasing mixed transaction analysis application scenarios. In order to solve the above problems, “intelligent data partition and layout” has become one of the current research hotspots. It extracts the effective characteristics of workload through data mining, machine learning, and other technologies, and design appropriate partition strategy to avoid scanning a large number of irrelevant data and guide the layout structure design to adapt to different types of workloads. This paper first introduces the background knowledge of data partition and layout techniques, and then elaborates the research motivation, development trend, and key technologies of intelligent data partition

* 基金项目: 北京市自然科学基金(4212022); 国家重点研发计划(2018YFB1004401); 国家自然科学基金(61772537, 61772536, 62072460, 62076245)

收稿时间: 2021-01-19; 修改时间: 2021-03-09, 2021-04-15; 采用时间: 2021-05-24; jos 在线出版时间: 2021-08-03

and layout. Finally, the research prospect of intelligent data partition and layout is summarized and prospected.

Key words: database system; partitioning strategy; layout strategy; machine learning

大数据时代，数据规模庞大，数据管理应用场景复杂，而现代商业决策与科学研究对数据的依赖性越来越高. 如何快速、高效地从这些海量数据中提取出用以分析决策的信息，给数据库系统带来重大挑战. 传统的数据库系统一般采用大量的索引来加速查询处理，但大数据时代，在海量的数据上大规模覆盖索引将会带来难以承受的存储与维护成本，并且会增加由于磁盘随机访问所造成的巨大 I/O 开销，所以人们逐渐转向使用面向扫描的数据处理策略. 在这样的背景下，执行扫描所消耗的时间将会受到数据库中数据划分与组织形式的高度影响，而传统的数据分区策略由于粒度太粗、不能适应复杂分析负载的特性等原因，已经远远不能满足时代的需求. 同时，能否实时地处理最新数据，往往也会对现代商业决策分析的结果产生重大的影响，这就要求数据库系统能够同时快速地处理 ACID 事务和复杂的分析查询. 而传统的数据布局策略一般只能支持 OLTP 与 OLAP 中的一种负载，且必须要进行繁琐且昂贵的 ETL (extract-transform-load)操作，这样往往会错过对最新数据进行分析的最佳时机. 综上，如何使用更加智能化的方法对数据进行高效、合理的划分与存储布局，已成为当前数据库系统研究领域需要解决的关键问题之一.

数据分区与布局联系紧密：数据分区从逻辑上将满足一定语义条件的记录或者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，通常包括水平分区、垂直分区、水平+垂直的混合分区等；数据布局则在物理层面解决以何种形式在存储介质上组织数据使数据库性能表现最优，通常包括行存储、列存储、行列混合存储等. 传统的数据分区和布局方法主要基于专家设置的启发式规则，如根据选定字段进行哈希分区[1]、针对 OLAP 负载使用列存储[2]等. 然而，传统的分区策略分区粒度较粗，并且是静态的，不能根据负载的特征进行有针对性的分区优化；传统的布局方法单一，不能满足多样化的查询需求，且同样不能根据负载动态地改变数据库的存储策略.

随着人工智能技术的发展，机器学习等技术在数据库和数据管理领域的应用受到前所未有的关注[3]，智能数据存储技术也获得了更深入的研究. 智能存储技术的诞生主要是为了解决数据查询效率低、读与写优化布局间冲突等困境，同时减少了传统方法所需要的人力开销，提供针对不同类型负载的细粒度、动态的存储分布和布局策略. 本文主要关注智能数据分区和布局技术在数据库存储中的研究进展，从智能数据分区、智能数据布局、智能数据分区与布局重组这 3 个方面详细阐述相关研究进展. 其中，智能数据分区部分根据数据分区后的逻辑组织形式(行块、列块、行与列的交叉单元)划分为水平分区、垂直分区与混合分区(水平+垂直)；智能数据布局部分根据数据在底层的物理组织形式以及是否冗余存储划分为基于传统行存/列存优化的布局、基于 PAX 的布局、基于冗余 NSM+DSM 的布局和基于单一 NSM+DSM 的布局；智能数据分区与布局重组部分则根据重组策略的触发方式的不同划分为传统的重组策略、基于监视窗口的重组策略以及基于启发式的重组策略. 本文最后对智能数据分区与布局所涉及的技术进行总结与对比，并对未来数据分区与布局技术的发展进行展望，图 1 给出了本文内容所涉及的智能数据

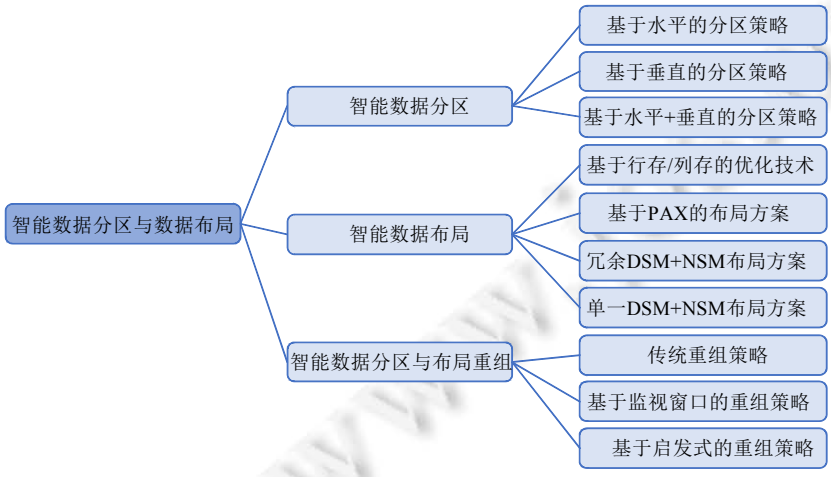


图 1　智能数据分区与布局技术体系架构图

分区与布局技术体系的架构图.

1 智能数据分区

数据库查询负载中的每条查询通常只与部分数据(行或列)有关, 因此只需使用数据库表的某个子集便可以得到准确的结果. 合理的数据库存储分区方式可以使大多数查询负载不需要扫描完整数据集就可以完成查询任务, 从而实现减少数据访问量、提高数据处理效率的目的. 因此, 数据分区问题通常将数据跳过的有效性作为目标[4], 设针对查询的负载为 W, $C(P,W)$代表在分区策略下能够跳过的数据量, 通常由跳过的元组和属性列的数量确定. 那么, 尽可能多地跳过无关数据的目标可以表示为

$$\text{argmax}_P C(P,W) \tag{1}$$

数据库分区的目的是使负载执行时尽可能多地跳过数据, 这涉及到每个块包含的数据数量的设定问题. 极端方式是将每一个元组作为一个分块并维护该块上的元数据, 虽然可以达到最细粒度地跳过不相关块的目的, 但与此同时产生了大量的元数据维护开销与磁盘 I/O 成本. 一般来说, 实际系统会根据 I/O 批处理和列式压缩等要求对块设定一个最小大小, 分区算法粒度的大小需要根据系统实际情况进行合理的设计[5]. 如今, 智能的存储策略通常分两步考虑如何将数据存储到内存/磁盘中, 即: 首先通过机器学习算法或启发式规则等策略确定将哪些列构成一个列组, 或哪些元组组成一个块; 然后在每个块内确定相应的布局策略, 即行存(n-array storage model, NSM)或列存(decomposition storage model, DSM)[2]. 本节将从基于水平分区、基于垂直分区和基于混合策略的分区这 3 个方面介绍智能数据分区问题, 具体的布局策略将在第 2 节给出介绍.

1.1 基于水平的分区策略

水平分区是指将表划分为若干水平块的过程. 每个水平分区包含表的若干个元组. 水平分区在提高查询性能方面起着重要作用. 给定一个大表, 可以根据一个或多个列的值进行水平方向的划分, 以便使每个分区中的某个或某些列的值受范围等条件的约束, 本节将这种策略得到的分区称为数据块. 系统为每个数据块维护描述本块数据特征的元数据(metadata), 如果元数据指示该数据块不包含一条查询的相关数据, 则该查询便可跳过该数据块. 根据指导该分区策略的属性列上的元数据数量和位置的不同, 数据分区的方法通常分为分区修剪与数据跳过两种策略[6]: 分区修剪(partition pruning)是数据仓库中常用的分区方法, 它仅维护指导该分区策略的属性列上的元数据. 如图 2(a)所示, 该销售表按照年份进行修剪分区, 每个分区子表上维护该分区年份列的值或范围, 以便能够快速索引到需要查询的年份的块的位置, 而图中被红线划去的其他无关块则被跳过, 从而节省大量的扫描成本. 数据跳过(data skipping)最早在文献[7]中提出, 它不同于分区修剪技术, 每个数据块不仅维护分区属性列上的元数据, 且维护其他能够指导跳过的数据列上的元数据, 如图 2(b)所示, 在维护年份属性的基础上, 在更细的粒度上维护收入列的最大值和最小值这个元数据进行数据块分区的指导.

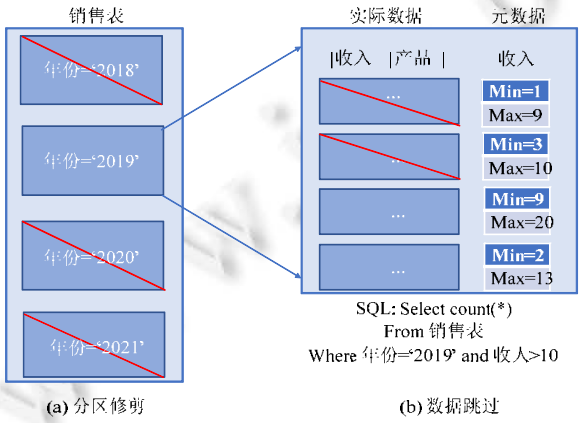


图 2 分区修剪与数据跳过

目前, 采用相关方法指导数据跳过的系统包括 Amazon Redshift[8]、IBM DB2 Blu[9]、Vertica、Google PowerDrill[10]、Snowake[11]、InfoBright[12]和一些 Hadoop[13,14]文件格式的系统.

水平分区不仅适用于粗糙粒度的查询, 同时也可以使用细粒度的分区算法指导精细的查询. 每个分区通常是文件或目录级别, 且可以在每个水平分区中构建索引. 常见的传统水平分区方式有范围分区、哈希分区、小型物化聚合(small materialized aggregates, SMA)[7]、Zone Maps[15]等. 智能水平分区可以应用在更精细的粒度上, 以进一步提高查询性能. 近年来较为经典的智能水平分区技术有基于负载统计的分区策略 Capser[16]、基于频繁项集挖掘的分区框架 SOP[17]、基于强化学习的分区方法 QD-tree[5]等.

1.1.1 传统水平分区策略

范围分区是传统数据库最常见的水平分区方式之一, 范围分区即给定一个大表, 可以对一个或多个列的值执行水平分区, 以便使每个分区中的分区列的值受范围的约束. 使用范围分区技术的典型系统有 Amazon Redshift、Google PowerDrill 等. 范围分区是分区修剪的形式之一, 在范围分区中每个分区包含若干元组, 各分区间的范围连续且不能重叠. 即如图 2(a)所示, 4 个分区分别存放对应年份内的元组.

哈希分区是另一种传统的经典水平分区方法, 它通过确定预先设定的分区个数来设计相应的映射函数, 之后使用映射函数将元组分配到不同的分区中. 哈希分区的局限性是只能针对整数进行哈希, 对于非整形的字段只能通过表达式将其转换成整数. 另外, 由于分区在创建表时已经固定, 故若新增或者收缩分区会造成大量数据迁移. 使用哈希分区的数据库系统包括早期的商用数据库系统 Oracle 和 MySQL 等.

范围和哈希分区粒度都比较大, 每个分区内仍包含大量元组, 且没有有效利用数据以及属性间的联系来组织分区策略, 故在这两种分区策略的基础上衍生出了另一种水平分区策略——SMA, 它通过在块上进一步设置索引, 以达到跳过更多数据的目的. 最流行的一种 SMA 方式是最小-最大值索引, 它存放某个或某些列上的最小值和最大值, 例如, 图 2(b)所示中维护的“收入”列上的最大-最小值. 即我们可以将范围或哈希分区后的块内根据其他属性列的值进行更细粒度的分区, 统计这些块中相应属性的最大-最小值, 并将这些信息记录在对应块的元数据中.

区域图(zone maps)[6]是为表构建的独立访问结构, 用于存储有关表区域的信息, 它使数据库能够修剪不满足表上列谓词的数据块. 区域图本质上是一个表, 保存着数据表每个区域的指定列上的最小/最大值范围. 区域是数据的任意定义单位, 可以等于表分区, 但通常要小于整个表分区. 如果区域小于表分区, 则区域图还可以存储表数据的每个区域或每个分区的最小/最大值范围. 如今, 很多商业数据库供应商提供了使用区域图的数据修剪技术, 如 IBM Netezza 数据仓库设备[18]提供了基于区域图的数据修剪, Amazon Redshift 允许在加载时对数据进行排序并创建最小/最大区域图[19].

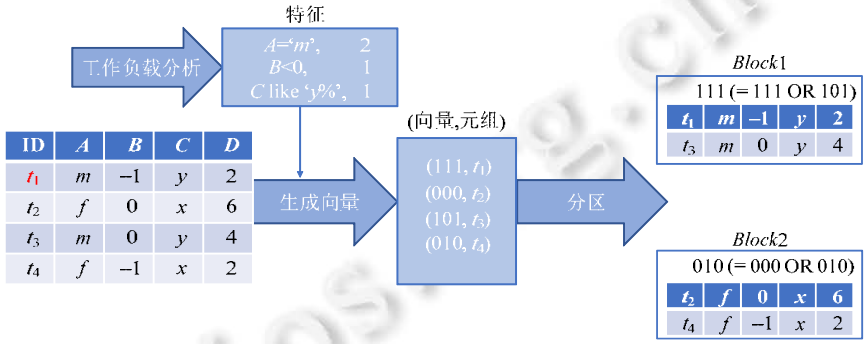
1.1.2 智能水平分区策略

传统的数据库分区策略虽然可以在一定程度上降低查询访问的数据量, 但是现代分析系统需要执行大量的查询语句, 仅根据数据本身的特性指导分区已经不能满足日益增长的业务场景的需求. 因此, 有效地利用工作负载的知识去支配分区, 是一项很有前景的工作. 然而, 传统的分区策略大多基于启发式, 不仅粒度过大、无法利用工作负载, 同时也无法针对时刻变化的工作负载实时调整分区. 为了弥补传统水平分区的缺陷, 基于工作负载的细粒度智能数据库分区策略逐渐受到越来越多研究者的关注.

传统的分区方法无法根据实际工作负载给出最有效的分区方式. 针对这个问题, Sun 等人在 2014 年提出了一个由工作负载驱动的基于跳过的细粒度水平分区框架 SOP. SOP 的核心思想是: 使用频繁项集挖掘[20]技术从工作负载中挖掘出若干具有代表性的过滤器作为特征, 以指导查询对数据块的跳过, 从而给出基于负载查询日志的最优分区方案.

图 3 通过一个简单的示例展示了 SOP 框架具体的工作流程, 设关系表 R 有 4 个属性, 表示为 $R(A,B,C,D)$, 首先利用频繁项集挖掘技术在给定的工作负载中提取 3 个特征作为过滤器筛选条件, 分别为 A=‘m’, B<0, C like ‘y%’, 根据这些过滤器中的谓词条件将表中的 4 个元组分别转换为 4 个向量: (1,1,1), (0,0,0), (1,0,1), (0,1,0), 其中, 1 代表元组的某列满足过滤器中对应列的谓词条件, 0 则表示不满足. 之后使用经典的自下而上

的聚类算法将上述元组划分成两个数据块, 分别为 *Block*1 和 *Block*2. 然后对这两个块内的向量取并集, 分别得到(1,1,1)和(0,1,0); 在查询执行时, 如果某个查询语句中包含了第 *i* 个特征, 那么这个查询向量的第 *i* 位将是 0. 将查询向量与块上联合向量取并运算, 当运算结果至少有一个位为 0 时, 则可跳过该分区. 例如, 查询语句 *Q*: select *B* from *R* where *A*='*m*', 按照上述规则生成对应的查询向量即为(0,1,1), 可以跳过 *Block*2. 相较于传统方法, SOP 针对特定的工作负载对分区算法进行了优化设计, 并且在 TPC-H 负载下数据扫描量只有传统范围分区算法的五分之一, 查询响应时间减少了约 2–5 倍, 大大增加了扫描数据的效率. 但 SOP 也存在以下局限性: (1) 过度依赖于选取的特征向量, 当特征向量长度过长时, 会造成难以承受的时间复杂度开销; (2) 虽然每个块可以使用包含在该块中的所有特征位图向量的"或"来描述, 但是这种描述是不完整的, 新插入的元组很可能找不到与之适配的分区; (3) 贪婪的合并算法并没有提供理论上的保证.



ID	A	B	C	D
t_1	m	−1	y	2
t_2	f	0	x	6
t_3	m	0	y	4
t_4	f	−1	x	2

图 3 SOP 框架示例

为了解决 SOP 存在的上述问题, Yang 等人[5]提出一个基于强化学习的细粒度水平分区框架, 称为查询数据路由树(QD-tree), 并设计了基于贪心算法和深度强化学习算法这两种构造查询数据路由树的分区策略.

其中, 基于贪心算法构件 QD-tree 的核心思想是: 首先从包含所有元组的根节点开始切割, 根据给定查询负载生成候选切割点集, 遍历切割点集以便寻找跳过代价最优的切割点 *p*, 之后寻找将节点切割成满足谓词条件 *p* 的左孩子节点和满足非 *p* 的右孩子节点. 在每次循环中, 切割的节点后所产生的两个孩子节点的大小都必须满足大于系统给定最小块的大小 *b*. 之后, 通过迭代的方式从上到下、从左到右切割当前树的叶子节点, 直到切割完成. 基于贪心算法构造的 QD-tree 相比于 SOP 的主要优势在于具有较低的时间复杂度, 给每个块只用维护简单 min-max 索引而非大量的特征向量, 且最后切分出来的块具有语义上的完整性, 在 TPC-H 数据集上的测试效果相比 SOP 有接近 2 倍的提升. 但基于贪心算法构造的 QD-tree 的缺陷在于: 每一次的切割只能考虑到局部的最优情况, 并不能得到一个全局最优的划分结果.

为了解决全局最优的问题, 同时又要避免动态规划[21]级别的时间复杂度的设计, 文献[21]的作者设计了基于深度强化学习算法(reinforcement learning, RL)构造的 QD-tree 以获得全局最优的路由树. 该算法具体流程是: 首先进行一次随机划分, 然后通过奖励逐渐学会识别更好的划分. 在尝试重建固定数目的树或者重建时间超过设定值后, 该算法找到最佳路由树. RL 方法不对查询或数据分布进行假设, 它只需要一个黑盒学习信号即这棵树的跳过能力. 图 4 描述了一个 QD-tree 的实例, 表具有两列(*cpu*,*mem*), 分别用 *cpu*<10%, *cpu*<5%, *men*=10 GB 切割节点 1–节点 3. 最后产生 4 个分区 *B*1–*B*4.

QD-tree 主要维护两类谓词: 一类是范围查询, 如(>,<,≥,≤), 通过维护元数据 *n.range*(*n* 表示某个节点)记录每个块对应属性列的范围条件, 另一类是相等比较(=和 in), 通过一个多维的向量 *n.categorical_mask* 标识, 若某一列满足对应列的谓词条件, 则该向量对应的位置置 1. 其后, 文献[21]的作者又推广到了更高级的谓词条件, 如 *A*=*B* 或 like 谓词等. 如今, 在云中存储数据的成本越来越廉价, 分析系统的一项理想功能是交换空间以潜在地缩短执行时间. 他们对原有的 QD-tree 加以扩展, 提出通过少量的数据复制和构建多个路由树的方法以优化块的跳过能力.

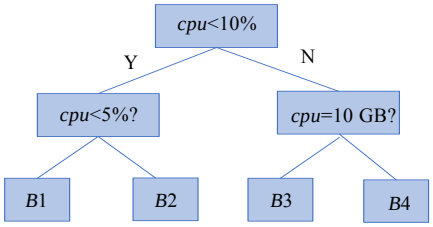


图 4　有 4 个叶块的 QD-tree 实例

虽然 SOP 与 QD-tree 在数据跳过方面已经取得了较大的进展, 并且大幅度地优化了分析负载的执行效率, 但并没有针对现代出现的混合事务分析工作负载(HTAP)进行优化设计, 代价函数也只是单方面地考虑了基于跳过的成本. 为了解决面对单一负载所设计的分区算法的局限性, 文献[16]设计了一个面向混合事务分析工作负载的内存存储引擎 Casper, 它依赖于列存储, 适用于任意但静态的工作负载配置文件. Casper 的核心贡献是: 设计了基于统计频率模型的方法, 将列布局问题转化为一个工作负载驱动的二进制整数优化问题, 并针对范围查询、点查询、插入、删除和更新这 5 种工作负载进行特定的建模设计, 以平衡读取和更新的性能; 同时, 通过引入 Ghost 值来创建可变长度的分区, 以保持其面对不同工作负载时的健壮性. 最后, 通过设计大量不同类型的实验, Capser 在混合工作负载上的吞吐量最少要比当前最优的设计高达两倍以上.

下面分 3 个方面对 Capser 的分区方案设计作一个较为详细的介绍.

(1) Capser 的成本建模: 首先, 使用频率模型捕获特定工作负载对每个基本块的访问操作, 通过基于随机读、随机写、顺序读和顺序写这 4 种基本访问模式进行计算, 得出 5 种工作负载所分别对应的每个块的成本 $cost_i$. 其设计目标是找到一种分区方案 P, 使得在数据库 D 上执行工作负载 W 的总成本降到最低:

$$\text{argmin}P_{cost}(W,D,P) \tag{2}$$

(2) Casper 分区的生成: 通过将之前基于工作负载统计生成的几种操作的 *cost* 代入成本函数模型, 并在每个块的两侧设置 p_i 标识(i 从 0 开始), p_i 值只能取 0 或 1, 当 p_i 为 1 时, 表示该位置为一个分区边界. 最终通过新变量的引入以及附加约束等条件, 将成本函数转化为 p_i 的线性函数, 通过 Mosek[22]解算器对成本函数求解;

(3) 引入 Ghost 值: 为了实现易于更新的分区模式, Casper 允许在各个分区的末尾引入若干个称为 Ghost 值的空槽来减少插入、删除、更新操作而导致的大量数据移动.

Slalom[23]提出了一种用于现场查询处理的在线分区和索引调节器, 它根据有关查询属性的值和访问频率的信息, 利用随机在线算法来定义逻辑分区, 以便在不进行物理重组的情况下将其虚拟地拆分为更易于管理的块. 这是一种非侵入的、灵活的分区方案, 除此之外, Slalom 还可以利用数据倾斜来创建水平分区.

智能水平分区技术同时将工作负载与数据间隐藏的联系考虑在内, 具有针对性地根据某一时刻工作负载与元组的特点做出最优决策. 这种方式不仅可以捕获列间数据的相关性, 并且可以较好地处理数据偏斜与工作负载偏斜等问题. 同时, 这种细粒度的水平分区策略可以与粗粒度的传统水平分区算法有效结合, 在不同粒度上, 为提高数据跳过的有效性这一目标服务. 表 1 对本节涉及的水平分区技术进行了总结.

表 1　水平分区技术总结

文献/框架	负载类型	核心分区机制	主要优化指标	*Cost* 函数
SOP[17]	OLAP	频繁项集挖掘+层次聚类	扫描效率	$C(P_i)=\lvert P_i\rvert \sum_{1\leqslant j\leqslant m} w_j(1-\bar{v}(P_i)_j)$
QD-tree[5]	OLAP	强化学习+贪心算法	扫描效率	$C(P_i)=\lvert P_i\rvert \sum_{q\in W} S(P_i,q)$
Casper[16]	HTAP	二进制优化算法	读取性能, 更新性能和内存利用率	$C(P)=\sum_{i=0}^{N-1} cost(rw_i)+cost(sc_i)$

1.2 基于垂直的分区策略

与水平分区不同, 垂直分区是指将表划分为若干列组(group of column)的过程, 每个垂直分区只包含表中列的一个子集. 当表垂直分区后, 如果查询请求的所有列都位于同一垂直分区内, 则查询语句只需要访问此垂直分区, 进而避免访问其他列, 所以垂直分区可以非常有效地减少查询访问的列数. 但垂直分区所带来的问题也非常明显: 当查询需要从多个垂直分区读取列时, 它需要承担重组这些列的额外成本, 因为行的不同属性值可能不会连续存储. 所以在给定目标工作负载情况下的垂直分区方案设计中, 通常分区的算法需要在访问垂直分区数目和行重建开销之间进行反复权衡.

垂直分区最简单的思路是暴力枚举所有可能的垂直分区, 并为每个分区计算运行给定负载的预期成本. 但这种方法在列数较多的情况下就会造成难以承担的分区成本和时间复杂度, 所以早期有关垂直分区的解决方案大致分为两类.

- 第 1 类方法早在 20 世纪 80 年代, 一些学者开始探索如何在一些给定限制条件下寻找最优的垂直分区问题, 如 Hoffer[24]开发了一个非线性的 0-1 程序来解决垂直分区问题, 该程序约束了每个子文件的容量并最大程度地减少了存储、检索和更新成本. Eisner 和 Severance 在文献[25]中提出了如何在快速的主内存和慢速的辅助内存之间分配记录这一问题, 并证实了该问题与 mincut-max 流网络问题类似, 可以通过标准的 Ford/Fulkerson 算法解决. 然而, 该方法需要构建一个具有较多节点和边的图, 在实际应用中过于复杂. 在后续的研究中, March 和 Severance[26]扩展了先前的模型, 以合并用于主存储器和辅助存储器的块因子. Schkolnick[27]考虑了在 IMS-type 层次结构内对记录进行聚类的问题, 并确定了一种有效的解决方法. 然而, 事先做出一些限制性假设在某种程度上会影响算法的性能表现, 并且缺乏一种统一且具有说服力的测评标准, 所以后来逐渐被其他方法所取代;
- 与第 1 类方法相反, 第 2 类利用启发式规则处理垂直分区问题, 并由于其优越的可扩展性和具有较低成本消耗的优点一直被沿用至今, 其中最经典的是基于属性亲和度矩阵的各类分区算法. 基于属性亲和度矩阵分区算法往往利用基于事务的访问频率值生成属性亲和度矩阵, 并作为分区算法的输入. 比如 Hoffer 和 Severance[28]测量了“属性对之间的亲和力”, 并使用文献[29]中开发的键能算法(BEA), 根据属性的亲和力对它们进行聚类. 但是处理非重叠 n 元分区问题需要通过一种算法来完成, 其复杂度为 $O(2^n)$. 用 n 元分区解决 BEA 的唯一方法是通过重复使用二进制垂直分区算法, 但是, 每次迭代二进制分区都会大大增加问题的复杂程度. 文献[30]提出了一种基于图形技术的垂直分割算法. 它从属性的亲和矩阵出发, 把它看作一个完整的图, 称为亲和图, 之后形成一个线性连通树. 然后, 通过循环分割产生最终分区. 文献[31]提出一种混合分区方法, 并介绍了混合分区工具的必要组成部分, 以允许使用网格方法对全局进行最佳划分. 文献[32]也同样将属性亲和矩阵作为出发点, 基于亲和值生成初始组, 随后合并初始组以生成最终的分区.

AutoStore[33]重点研究如何根据动态负载进行实时分区更新, 其核心组件是负载监视器与分区优化器: 前者通过窗口机制确定分区的触发条件, 后者基于贪婪算法和亲和度矩阵设计支持在线的分区算法. 负载监视器将静态查询窗口的一系列查询作为输入, 应用 O^2P 算法得到分区候选项. O^2P 借鉴贪婪算法的策略, 初始状态定义与属性等长的切割矩阵 S, 0、1 分别代表该位置是否被切割. 对于到来的每个查询, 将引用的所有属性都记为切割点, 但每次只选择执行成本最低的切割点分割; 当下一个查询来临时, 将上一轮未使用的切割点与本轮切割点一并考虑, 继续选择执行成本最低的切割点进行分割. 以此类推. 在得到垂直分区候选项后, 分区优化器根据每次输入的查询, 计算两种属性同时出现的频数, 重新更新属性间的亲和性矩阵, 对各分区中的各分区单元进行排序, 为其选择合适位置以保证能够提供最大的网络贡献度. 最后, 通过成本模型和收益模型来比较当前分区方案 P 与分区分析器给出的分区方案 P'的转化成本和预期收益, 以指导分区方案的更新. AutoStore 通过提出一套在线垂直分区的系统组件, 保证部署的分区策略自适应特定的负载场景, 并在 TPC-H 负载上的测试结果优于传统的基于亲和度矩阵的方法, 但其局限性是: 其负载监视器的查询窗口的大小只能手动确定, 且无法根据负载的波动而自动调整.

H20[34]也是采用基于亲和度矩阵的惰性方法来生成新的分区策略，相比于 AutoStore 使用静态窗口，H2O 基于负载驱动设计了更加智能的动态监视窗口来统计各个查询语句的频率以及属性之间的相关程度. 属性之间的相关程度定义为在访问过程中一起访问它们的频数. 访问模式以两个关联属性矩阵形式[30]进行存储，一个矩阵存储 select 语句中的属性，另一个矩阵存储 where 语句中的属性，并把经常一起访问的属性放到一个列组中. 例如，某 20 个查询语句中有 5 个语句都访问了属性$\{a_1,a_5,a_8,a_9,a_{10}\}$，那么应该将这 5 个属性放到同一个列组中.

但是，基于亲和度矩阵的方法具有一个很大的局限性，就是当涉及两个以上属性时，亲和度矩阵将不能反映其亲和性. 因此，近年来研究者们开始探索更加灵活的算法，以发现关键的属性组合. Hyrise[35]是一个主存混合存储数据库系统，它提出了一个基于爬山算法的策略来找到最优的垂直分区，并设计了一种以 CPU 缓存未命中数为主要指标的成本评价函数，可以测量连续投影、非连续投影、选择、连接和聚合、填充容器和重构等操作的缓存未命中数，从而能够在更加细粒度的层面上全面、准确地评测分区方案的有效性. 对于列的分析查询，通过划分为窄分区而使查询性能更好；对于 OLTP 访问方式，又会自动形成宽分区. 研究者提出了分区分割算法主要由 3 个阶段构成，图 5 展示出其工作流程图.

- 候选项生成：第 1 阶段确定关系表 R 的所有主分区. 负载 W 中的每个查询都要涉及到某个关系操作. 第 1 个关系操作将属性集划分为两个子集，随后的查询对应的操作都以递归方式将每个子集继续拆分为更小的子集；
- 候选项合并：算法的第 2 阶段是检查主分区的排列，判断是否可以生成额外添加的候选分区，以减少工作负载总成本. 对于选择操作，合并两个主分区和有利于对属性进行广泛、随机的访问；对于投影操作，合并过程通常会增加额外的访问开销. 为了降低这种随机访问成本带来的益处和对少数列进行大扫描造成的损耗之间的矛盾关系，文献[35]中通过修剪许多潜在的候选分区来解决；
- 分区生成：对上一阶段产生的所有候选分区进行组合，形成一组覆盖关系表 R 但不重叠的分区列表，通过评估这一组分区列表构成的有效分区的成本，选择成本最低的那一组作为最佳分区策略.

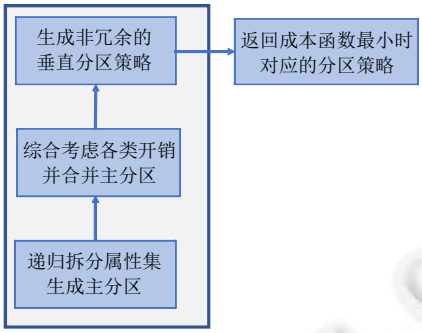


图 5 Hyrise 工作流程图

基于启发式的方法虽然具有高可扩展性和低时间复杂度等优点，但这些方法寻找的垂直分区方案却只能找到一个局部的最优解，早期寻找全局最优解的方法，如暴力枚举、动态规划等方法，由于其极高的时间复杂度让人望而却步. 然而，得益于近来年人工智能领域的迅速发展，有的研究者开始尝试使用更加高级的数据挖掘方法来解决这个问题，并取得了可喜的进展. 如 GridFormation[36]将在线自我管理分区选择任务转化为强化学习问题，其目的是建立一种能够利用经验并模仿专门方法的通用解决方案. 将外部预测的工作负载和当前的物理设计作为 agent 的输入，环境根据新的成本估算立即生成针对工作负载的奖励，并通过确定给定状态的可能操作来指导搜索过程. 为了在线使用，agent 学习一个固定长度的动作序列，该动作序列使预测工作量的时间奖励最大化.

垂直分区作为一项重要的数据库技术，数十年来已经得到了广泛的研究与应用. 如上文所述，传统的垂直分区技术可以概括为两类：第 1 类通常在一些限制性假设下寻找问题的最优解[26−29]；第 2 类使用一些启发

式方法, 如基于亲和度矩阵的方法[25,30−34]、基于爬山算法的方法[35]等, 这些启发式算法具有高可扩展性和运行成本低等优点, 但往往只考虑查询开销与元组重建成本, 且只能得到局部的最优收益. 而现阶段智能的垂直分区技术则使用了更加高级的数据挖掘技术, 考虑了 CPU 缓存命中数等更多的因素, 并且可以得到全局的最优收益, 但缺点是模型训练的成本较高且只能离线执行. 表 2 对本节涉及的具有代表性的垂直分区技术进行了简单归纳.

表 2 垂直分区技术总结

文献/框架	负载类型			核心分区机制	主要优化指标	*Cost* 函数
	OLAP	OLTP	HTAP			
AutoStore[33]	–	√	–	亲和度矩阵+爬山算法	扫描效率, 分区重建成本	–
Hyrise[35]	–	–	√	贪婪算法+组合优化	CPU 缓存未命中数	$C(r,\pi)=\sum_{i=0}^{n-1}Miss_i(r,\pi)\cdot\pi\cdot s$
H2O[34]	√	–	–	访问统计矩阵+动态窗口监视	扫描效率, 分区重建成本	$C(L)=\sum_{i=1}^{\lvert L\rvert}\max(cost_i^{IO},cost_i^{CPU})$
GridFormation[36]	√	–	–	强化学习	优化器估算结果+序列学习成本	–

1.3 基于水平+垂直的分区策略

单一的水平分区将一个大表分成了若干个子表, 每个子表包含全部的属性列和部分的元组. 而单一的垂直分区将表划分为若干个列组, 每个列组包含若干个属性和全部的元组项. 现代数据库所面对的事务更加多样且复杂, 仅使用水平或垂直分区往往不能适应多种类型的应用场景, 故现代数据库分区研究中涌现出了混合分区策略. 数据库混合分区策略主要可以分为两类.

(1) 先将表水平分区成若干个子表, 然后在每个子表内, 基于工作负载等信息设计最佳垂直分区策略. 由于不同列组内元组是对齐的, 所以不需要元组对齐重建开销, 如 Peloton[37];

(2) 先将表垂直分区成若干个列组, 然后再在每个列组内应用水平分区算法, 此时, 各列组内元组顺序被打乱, 需要为每个块中的元组维护额外的元数据并存在大量元组重建开销, 如 GSOP[38]和 GSOP-R[4]框架.

早期的研究大多基于单一的水平分区或垂直分区, 尽管很多研究已经转向了自动化物理数据库设计领域[39−43], 但文献[44]首次将水平和垂直分区以及对齐要求合并到一个数据库中, 其主要贡献是强调了在优化数据库物理设计以提高性能的同时纳入可管理性需求的重要性. 文献[44]定义了 *interest* 函数以用于捕获不同属性列之间的相关程度, 同时介绍了一种确定列组有效性的方法, 并可以使用该方法对列组的有效性进行过滤/排序. 虽然文献[44]提出了水平分区与垂直分区策略, 并同时考虑了索引对齐等问题, 但该文献也只研究了单一的水平分区或垂直分区, 并未真正地将水平分区和垂直分区同时应用在同一张表中.

Peloton 是一个支持 HTAP 负载的内存数据库管理系统, 为了适应灵活的 HTAP 型事务, Peloton 设计了更加灵活的分区策略.

- 首先, Peloton 提出了一种基于 tile 体系结构的水平分区方案. 通过调整 DBMS 配置存储在一个块组中元组的数量, 以便生成最适合 CPU 进行高速缓存的块; 并且, 对于具有不同模式的两个表, DBMS 可以选择为每个块组存储不同数量的元组, 以支持扩展更加复杂的水平分区策略;
- 其次, 在水平分区的基础上, Peloton 设计了一种基于负载的动态垂直分区策略, 其分区执行过程具体分为以下两个阶段.
  - 采取在线 *k*-means 聚类算法, 动态地根据查询输入确定具有相似的访问特征的属性列, 并将其分在同一类中, 其中, 一个属性可能出现在多个聚簇中. Peloton 采用两个查询间的距离来定义相似度, 距离=两个查询中不重叠访问的属性类别数/表 T 中的属性数;
  - 根据查询的成本确定每个聚簇的权重, 降序排列各聚簇, 使用贪心算法, 根据聚类算法输出的

属性聚簇, 先分配优先级高的聚簇到物理块 tile, 后续聚簇如果存在已经被分配的属性则不再为其分配. 以此类推, 直到所有属性都分配到某一物理 tile. 由于受块 tile 自身大小的限制, 一个聚簇通常需要占用多个物理 tile, 这自然形成了同一水平 tile 组构成的水平分区.

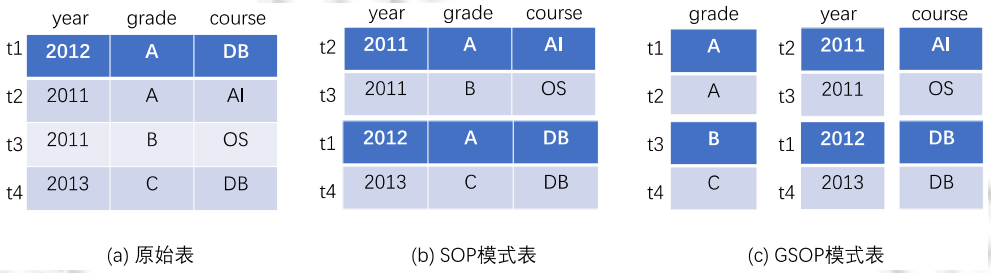
Peloton 虽然支持水平与垂直分区, 但在水平分区方面只使用了最简单的划分策略, 并未对水平分区进行深度的优化; 且用于垂直分区的 *k*-means 算法的聚类中心的选择缺乏理论依据, 并会对分区结果造成较大的影响.

GSOP 框架在只支持水平分区的 SOP 框架扩展了垂直分区算法, 并对水平分区和垂直分区进行了协同优化设计, 具体如图 6(a)所示. 若考虑下面两种查询 Q1 和 Q2,

- Q1: select * from *R* where *grade*='*A*';
- Q2: select * from *R* where *year*>2011^*course*='*DB*',

假设这两种查询频率相等, 若只考虑 Q1, 根据启发式规则, 希望将图 6(a)水平分成 t_1t_2 和 t_3t_4 这两个列组; 而若只考虑 Q2, 很明显, 希望将表水平分成 t_1t_4 和 t_2t_3 这两个列组. 因为这两种查询频率相等, 所以对于此功能冲突无法找到同时适合这两种查询的分区策略. 如图 6(b)所示, SOP 框架无法同时找到最适应上述两个查询语句的分区方案.

针对这些问题, 在 SOP 的基础上, Sun 等人进一步设计了 GSOP 框架. 如图 6(c)所示, GSOP 通过引入列分组和局部特征选择, 以允许不同的列组内存在不同的水平分区方案, 从而解决了 SOP 存在的特征冲突的问题. 但同时, 由于列分组而引入了额外的元组重建的开销: 由于每个列组内元组的顺序不同, 为了物化查询结果, 除了读取实际数据之外, 还需要读取元组的 ID.



(a) 原始表 (b) SOP模式表 (c) GSOP模式表

图 6 GSOP 与其他分区方案对比

因此, GSOP 分区方案设计需要在跳过性能与元组重建成本之间寻找一个权衡. 为了更加精准地评估这些开销, Sun 等人为 GSOP 设计了一个成本函数, 并将各种成本折中考虑在内. 算法具体流程如图 7 所示: 首先, 提取负载中具有代表性的过滤器作为全局特征; 之后, 基于目标函数将列划分为列组, 对于每个列组, 选择一个全局特征子集作为其局部特征, 以指导列组内部的水平分区; 最后, 迭代地进行这个过程, 直到成本函数达到最优. 但是, GSOP 框架存在一个严重问题, 即在查询过程中, 即使要访问某个块中的一个元组, 也要遍历该块内所有的元组.

GSOP-R 框架在 GSOP 框架的基础上, 进一步解决了在同一列上的不同查询谓词的冲突问题. 此外, 允许进行少量的数据复制以减少查询时扫描的元组数. GSOP-R 设计了选择性复制和完整性复制两种方案. 选择性复制方案受经典数据库中实例化视图的启发, 在预先评估经常使用的查询并构建实例化视图的基础上, 使多次执行的相同查询直接转化为实例化视图中查询, 而不必访问原始的数据; 而完整性复制方案则对所有特征进行聚类, 并设计策略以减少每个聚类内的功能冲突, 之后构建与聚类数目相同的数据库副本以运行包含对应特征的查询语句. 然而, 上述两种复制方案的分区粒度过大, 故 GSOP-R 框架又进一步利用适量的细粒度数据复制, 以提高数据跳过能力和查询性能的有效性.

表 3 对本节涉及的水平+垂直分区技术进行了总结: 水平+垂直方向的混合分区问题较多使用了机器学习算法. 将机器学习相关算法应用于此类分区问题, 可以更加快速并准确地提取工作负载中的相关信息. 然而

目前, 机器学习只用于指导对混合分区中垂直分区部分的决策. 这种混合的分区模型进一步提高了查询时跳过的有效性和查找时对数据的针对性.

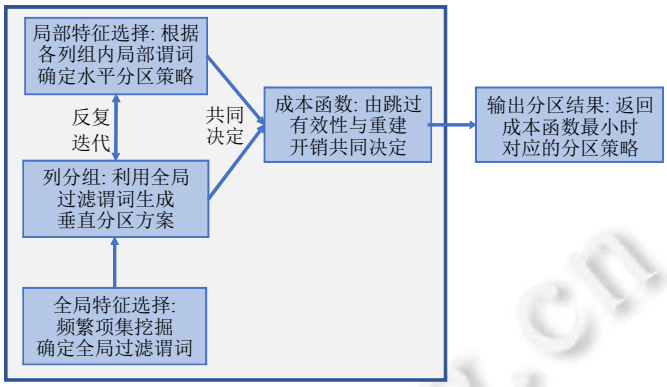


图 7 GSOP 工作流程

表 3 水平+垂直分区技术总结

文献/框架	负载类型	核心分区机制	主要优化指标	*Cost* 函数
Peloton[37]	HTAP	*k*-means 聚类+贪心策略	给定负载执行的时间	–
GSOP[38]	OLAP	聚类算法+ 频繁项集挖掘	扫描效率, 行重建开销	$C(W,G)=\sum_{q\in W}\sum_{G_i\in G}\lvert G_i\cap C^q\rvert\cdot r_i^q$
GSOP-R[4]	OLAP	聚类算法+频繁项集 挖掘+分区复制	扫描成本, 重建 成本, 存储成本	$C(P)=q\_cost(q,L,R,p)+s\_cost(L,R)$

1.4 小 结

数据库上的水平分区和垂直分区从两个角度出发来优化查询等操作的成本, 传统的基于代价模型或启发式规则的分区算法在面对复杂的查询负载和数据分布时, 性能会大打折扣. 本节从多个角度对智能数据存储分区的研究进展进行了较为详尽的介绍与对比, 本节所主要涉及的数据库分区技术总结见表 4, 总的来说, 智能数据库分区存在以下优点.

- 改善查询性能: 对分区对象的查询可以仅搜索自己关心的分区, 以提高检索速度;
- 增强可用性: 如果表的某个分区出现故障, 则表在其他分区的数据仍然可用;
- 维护方便: 如果表的某个分区出现故障, 需要修复数据, 则只修复该分区即可;
- 均衡 I/O: 跨多个磁盘来分散数据查询, 以获得更大的查询吞吐量. 当同时在涉及 *SUM*(·)或 *COUNT*(·)这类聚合函数的查询时, 可以在每个分区上并行处理, 最终只需要汇总所有分区得到的结果.

目前, 虽然在数据库分区领域的研究已经比较完善, 但同时也存在一些缺陷与不足.

- 针对动态负载的垂直分区, 相关研究大多采取冗余存储的策略, 增加了数据库的存储压力;
- 针对基于工作负载的水平分区问题, 目前的研究只适用于脱机处理分区问题, 无法针对实时的工作负载做出分区决策;
- 目前, 在水平与垂直两方向混合的分区策略研究较少, 且现有的研究难以同时兼顾两个方向的最优决策;
- 目前, 对各种分区算法成本的评估尚缺少统一的代价模型, 有的只是通过通用基准测试其扫描效率, 而有的则专一对磁盘 I/O 开销或 CPU 效率进行成本估算, 缺乏统一的代价模型, 使得各类模型之间难以进行客观的比较.

随着人工智能的迅猛发展, 越来越多的机器学习算法已经应用到了数据库分区上, 未来或许将成为数据库分区的一个重要的研究着手点.

表 4 数据库分区技术总结

文献/框架	分区策略			主要优化指标	核心分区机制
	水平	垂直	水平+垂直		
SOP[17]	√	–	–	扫描效率	聚类算法+数据挖掘
Casper[16]	√	–	–	读取性能、更新性能和内存利用率	二进制优化算法
QD-tree[5]	√	–	–	扫描效率	强化学习+贪心算法
AutoStore[33]	–	√	–	查询性能收益、分区重建成本	亲和度矩阵+爬山算法
H2O[34]	–	√	–	查询成本、分区重建成本	动态窗口监视
GridFormation[36]	–	√	–	优化器估算成本	强化学习
Hyrise[35]	–	√	–	CPU 缓存未命中数	贪婪算法+组合优化
Peloton[37]	–	–	√	执行给定负载的时间	*k*-means 聚类+贪心策略
GSOP[38]	–	–	√	扫描效率、行重建开销	聚类算法+数据挖掘
GSOP-R[4]	–	–	√	扫描成本、重建成本、存储成本	聚类算法频繁项集挖掘

2 智能数据布局

在数据布局的设计问题上, 很早就诞生了一种思想流派, 认为“One size does not fit all”, 即提倡专业化解决方案, 设计不同的存储引擎以应对不同的应用场景, 在此基础上发展了以行存NSM和列存[2]DSM为主的两种传统数据布局方式, 分别面向以事务为主的联机事务处理(on-line transaction processing, OLTP)和以分析为主的联机分析处理(on-line analytical processing, OLAP). 然而, 随着混合事务分析处理(hybrid transactional analytical processing, HTAP)在系统应用场景的增多, 例如, 决策支持下的实时分析、推荐系统、动态监控: OLTP 和 OLAP 负载经常混合出现[45], 传统行存、列存的缺点不断放大, 在传统的布局结构上的优化工作提升性能有限.

第 2 种思想流派主张一种普遍的解决方案, 认为从构建多个应用程序的角度来看, 对多个存储引擎的开发成本和维护成本都是巨大的, 理应设计一种能够解决所有负载问题的布局结构. 这种简约的思想给出了一个美妙的憧憬, 如何实现第 2 种思想流派这种通用型的设计, 是研究者不得不考虑的因素. 受到学术界[46−50]和工业界(如 SAP[51])的支持, 智能布局技术逐渐发展, 以研究如何根据实际负载情况, 设计高性能的存储结构以减少系统负载成本为目标, 产生了许多著名的研究成果. 基于这种目标, 设 C_{est} 为原布局方案 P 下执行优化策略 S 后的负载执行成本, W_t 为当前时间 t 的工作负载情况, 则智能布局优化问题可表示为寻找一种合适的布局策略 S', 使当前负载的执行成本最低或接近最低:

$$S'=\text{argmin}_S C_{est}(W_t,P(S)) \tag{3}$$

智能布局技术包括提出多种优化思路来解决 HTAP 带来的挑战: (1) 在行存、列存上进行优化; (2) 提出新的布局结构, 如 PAX、冗余 NSM+DSM、单一 NSM+DSM.

2.1 基于行存/列存布局的优化技术

在行存、列存结构上优化 HTAP 负载主要有两种思路: (1) 基于行存结构的分析负载优化; (2) 基于列存结构的事务负载优化. 列存结构在数据组织相比行存更灵活, 便于数据压缩和索引结构的设计, 本节将重点介绍基于第 2 种优化方式的研究工作.

行式存储将完整的行从表头开始依次存放, 每个页面的最后有一个索引, 存放了页内各行的起始偏移量, 结构如图 8(a)所示, 即依次存放$\{a_1,b_1,c_1,d_1,e_1,\ldots,a_5,b_5,c_5,d_5,e_5\}$. Oracle[52]、DB2[53]、Informix[54]、Microsoft SQL Server 2000[55]等偏重于处理 OLTP 事务的数据库系统都采用行存储. 随着内存容量的提高, 内存数据库(main memory database, MMDB)技术也得到极大发展, 很多 MMDB 采用行式存储, 例如: SQL Server 2014 新增了 Hekato[55]组件, 实现了内存最佳化数据表与索引功能; VoltDB[56]将所有数据都存入内存中, 进行行式存储; MySQL Cluster 基于 Shared-Nothing 结构, 采用行级存储模式, 将记录分为内存部分和磁盘部分.

列式存储[2]最早可以追溯到 20 世纪 70 年代移项文件(transposed files)的研究, 它将关系表按列垂直分割成多个子列, 每个列中的值以连续的形式在数据页内组织, 页的尾部有一个索引, 图 8(b)是一个 DSM 布局方式的示例.

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	d_2	e_2
a_3	b_3	c_3	d_3	e_3
a_4	b_4	c_4	d_4	e_4
a_5	b_5	c_5	d_5	e_5

(a) Row Major Layout

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	d_2	e_2
a_3	b_3	c_3	d_3	e_3
a_4	b_4	c_4	d_4	e_4
a_5	b_5	c_5	d_5	e_5

(b) Column Major Layout

图 8 行存/列存存储示意图

2005 年, C-Store[57]实现了基于多副本的列式存储和多种索引方式, 以优化 OLTP 事务. 它基于 DSM 设计了两个存储系统: 写优化存储(writeable store, WS)和读优化存储(read-optimized store, RS), 其中, WS 组件通过引入存储秘钥 SK 和 B 树结构支持频繁插入和更新等事务操作, 并将元组批量地转移到 RS 中. MonetDB[58]采用内存映射方式存储, 充分考虑了硬件性能, 将 Cache 的各性能参数量化, 统一计算权重, 以达到合理评估选择计算模型的目的. Google 研发了用于大规模只读数据的查询系统 Dremel[59], 并引入了一种紧凑的数据列存格式: 存储数据序列化之后的两类辅助信息. L-Store[60]与 C-Store 解决策略类似, 同样采用读写分层的方式兼顾了写入的性能, 将列数据分成了两个部分: 只读的 Base Page 与只写的 Tail Page, 通过指针关联. 与 C-Store 不同, L-Store 旨在改善事务性能, 通过设计一种新的基于 Epoch 的无争用页面分配方法, 在后台周期性地执行延迟和无冲突列数据合并, 不仅将稳定的数据从写优化的列式布局惰性地、独立地转移到读优化的列式布局, 而且不阻塞正在进行的和新创建的事务.

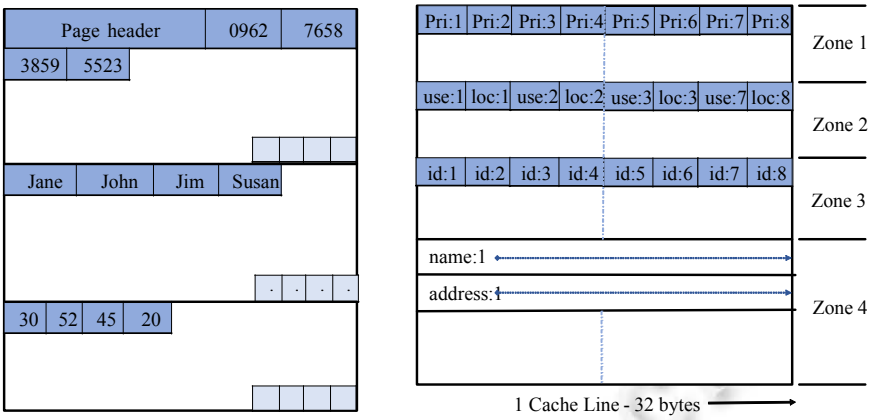
Database Cracking[61]根据查询负载, 动态排序列存中的元组, 并使用辅助数据结构最小化元组重建成本. 文献[62,63]同样根据查询负载, 通过列重复和列排序, 进行了宽表的布局优化.

基于传统行存与列存上的布局优化技术, 一般都是在一种既有布局方式的基础上再添加其他不同的数据组织形式或优化策略来支持 HTAP 负载, 这种方式的优势在于 OLTP 和 OLAP 负载请求共享相同的底层存储, 因此 OLTP 提交的对数据的修改可以被立即用于 OLAP 请求, 但是由于采用的不是单纯的行存就是单纯的列存布局, 所以无论对于事务处理还是查询分析处理性能的提升都非常局限, 不能从根本上解决大数据时代 HTAP 负载所带来的挑战.

2.2 基于PAX的布局方案

为了从根本上解决 HTAP 负载所带来的问题, 一些研究人员开始尝试脱离从传统布局进行优化的编码和开发模式, 直接开发适应 HTAP 负载的全新布局方式. 在这样的背景下, 文献[64]提出了一种新的数据布局策略 PAX (partition attributes across), 如图 9(a)所示. PAX 的思想非常简单, 即: 首先, 根据存放表中属性的数量将一个 Page 划分为多个 Mini Page; 之后, 表的每个属性按列组织的方式对应存放到每个 Mini Page 中. 由于表中的每个属性都存放到同一个 Page 中, 这样进行面向记录的操作节省了跨页链接的成本; 而由于每个 Mini Page 里面的数据都按列存放, 相比 NSM 的布局方式在查询读取时大大提高了 Cache 命中率, 这样就同时也显著提高了面向 OLAP 负载的性能.

然而, PAX 在执行表中一小部分属性的访问计划时, 仍然会导致较多的缓存未命中情况. 针对这个问题, 文献[65]在 PAX 的基础上提出了一种优化后的布局策略 Data Morphing, 如图 9(b)所示. 相比 PAX 在 Mini Page 里只存放单一的属性, Data Morphing 充分利用了数据的局部性. 根据查询负载, 将经常一起访问的属性放在同一个 Mini Page 里面, 以进一步增加 Cache 的命中率, 从而提供了更加灵活、通用的布局策略, 以适应工作负载的各种变化. 但 Data Morphing 由于优化时并没有考虑到磁盘 I/O 等其他成本, 在增大数据量后收益可能会降低; 并且由于是针对 OLTP 数据库上的分析性能优化, 对于缓存的利用率和分析负载的执行效率仍然达不到列存数据库的高度.



(a) An example data PAX page (b) An example Data Morphing page

图 9 PAX 和 Data Morphing 的页面组织

近年来, 随着列存数据库以及内存数据库的飞速发展, PAX 的布局策略又被应用于列存数据库上的 OLTP 优化, 以使列存数据库系统能够适应 HTAP 工作负载, 并在保持高查询性能的同时又能维持高事务吞吐量. 这类应用比较经典的系统是 Hyper[55]和 NoisePage[66]. Hyper 在 PAX 布局的基础上添加了小型物化聚合 SMA[6], 以方便在扫描时能够进行数据块跳过, 并根据工作负载将关系划分为读优化分区和写优化分区, 在写优化分区进行更新操作, 在读优化分区针对每个不同的属性采用不同的压缩策略, 以节省主存中的空间占用. NoisePage 则通过在块头中添加一个有效位图, 并为系统中的每个可变长度值添加额外等长的事务元数据, 事务只访问该元数据而不是底层存储, 通过只向事务元数据写入更新, 最后通过一个轻量级的转换算法进行数据合并, 从而将可变长度的更新转换为固定长度的更新, 解决了列存数据中写放大的问题.

基于 PAX 的布局方案颠覆了传统的布局设计方式, 在页级布局上同时融合了 NSM 布局与 DSM 布局的优点, 能够很好地同时兼顾 OLTP 负载和 OLAP 负载, 且能够灵活地应用在各类系统中, 具有非常好的发展前景. 但作为一种固定的布局方式, 不能根据负载的变化而动态地改变底层的存储结构, 且对于缓存的利用还是达不到 DSM 的高度, 所以仍然不是业界主流的布局方案.

2.3 基于冗余NSM+DSM的布局方案

冗余 NSM+DSM 布局是指同时存在以 NSM、DSM 结构来组织数据的混合布局方式, 并允许出现冗余的副本数据, 如元组或列数据. 比如, 经常被 OLTP 和 OLAP 两种负载同时访问的数据可能会同时存在以 NSM 和 DSM 组织方式的两份副本. 目前常见的冗余布局技术有镜像分割、中央视图、列组等.

Fractured Mirrors[67]技术的核心理念就是往存储引擎中写数据的时候写两份: 一份数据用 NSM 的存储格式, 另一份用 DSM 的存储格式, 然后根据业务的不同需求访问不同存储格式的数据, 这样可以有效地结合两种存储模式的优点, 服务于 Ad-hoc OLAP 查询负载. 但很明显, 简单、粗暴地维护完整的数据副本不仅仅会增加大量存储空间的负担, 而且会带来很高的镜像同步成本, 大大降低了现代数据分析的时效性.

与 Fractured Mirrors 不同, Octopus DB[68]通过维护一个包含常用数据不同布局副本的中央日志来解决 HTAP 负载问题. 为了不局限于任何类型的布局和工作负载, Octopus DB 通过引入逻辑日志作为主要存储结构, 它包括多个存储视图(storage view, SV), 即一个覆盖主日志或其子集的物理数据表示形式, 以维护存储在不同布局的数据库的多个副本. 所有数据都集中在一个中央日志中, 所有插入和更新操作都在该日志中新建 log-entry. 根据这个日志, 可以定义几种 SV, SV 将日志的子集表现在不同的物理布局中, 通过创建正确的 SVs, 可以适应不同负载. 图 10 列出了 Octopus DB 的几个用例, Octopus DB 完全基于工作负载动态创建存储视图, 并且在面对混合事务分析工作负载时的性能最高能够达到传统数据库的 5 倍, 在存储消耗上也要远远小于维护完整的数据副本. 但是, Octopus DB 架构存在着自身体制的问题, DBMS 的查询计划器不可能生成覆

盖所有物理结构的查询计划. 另外, 由于存在多种数据副本, 其同步成本也非常高.

Use-Case (traditional systems)	Storage view definition	
	type	example query
row store	Row SV	any
Column store	Col SV	any
PAX	PAX SV	any
Fractured mirrors	Row SV and Col SV	same query for both
Column groups	Row SV and Col SV	$\pi_{a_1,\ldots,a_k}$ $\pi_{a_{k+1},\ldots,a_m}$
Index	Index SV	any
Indexed row store	Index SV(Row SV)	any
Indexed column store	Index SV(Col SV)	any
Read-optimized indexed column store+differential write-optimized row store	Index SV(Col SV) Row SV	$\sigma_t < now()$-1day $\sigma_t \geq now()$ 1day
Partial index	Index SV	$\sigma_{420} \leq a_k \leq 42000$
Projection index	Col SV	π_{a_k}
Partial projection index	Index SV(Col SV)	$\pi_{a_k}(\sigma_{420} \leq a_k \leq 42000)$
DSMS	Index SV	$\sigma_t > now()$ 5min
DSMS+archive	Index SV and Col SV	$\sigma_t < now()$-5min $\sigma_t \geq now()$ 5min
Snapshot	any	any
Replicated row store	Row SV Row SV	same query for both
Query	any	any
Dynamic view	any	any
Materialized	any	any

图 10 Octopus DB[68]用例说明

为了避免大规模数据副本引发的同步问题, H2O 混合系统可以根据不断变化的 HTAP 工作负载动态调整存储布局, 即 Adapter Store 的特性. 它采用列组(即行与列结构的组合)为主的布局方式, 允许同一时间在单个数据库中存在不同的存储布局, 例如行存、列存和列组. 当某一属性有不同的访问方式时, H2O 采取冗余存储的策略, 同一属性以多种布局策略存储在数据库中, 针对不同查询模式, 匹配合适的数据布局与数据访问模式以达到理想状态. 如果存在频率较高的查询同时访问(*C*,*D*,*E*)和(*A*,*B*,*C*)两组属性, H20 会进行列复制, 如列 *C*, 存放(*A*,*B*,*C*)和(*C*,*D*,*E*)这两个列组. 与 Octopus DB 面临相同的困境, 当布局方式千变万化时, 为所有的存储策略定制执行策略是不现实的, 为此, H2O 提供了自适应混合查询执行引擎, 即在查询计划中, 数据布局应该与适当的执行策略相结合, 为查询计划中查询运算符创建定制代码, 例如在过滤器中增强谓词求值等方法.

GSOP-R 框架也设计了一种基于列组的布局方式. 在列组中采用 PAX 样式布局[64], 然后将每个列组水平划分为块, 并在每个块内使用列式布局, 可以支持细粒度的行或列的复制, 以提高数据跳过的有效性.

基于冗余 NSM+DSM 的布局方案利用硬件以及分布式数据库发展所带来的优势, 使用大量额外的存储空间以换取更快的负载响应时间, 并且可以避免单一的布局方案而引起的各种冲突与妥协, 逐渐成为一种业界主流的布局方案. 但是由于其巨大的额外存储开销和数据同步成本, 在某些具有低存储和低延迟需求的应用场景仍存在很大的局限性, 如主存数据库等.

2.4 基于单一NSM+DSM的布局方案

冗余的数据布局方案的缺陷十分明显, 维护大量的冗余数据往往需要很高的成本, 因此单一的 NSM+DSM 存储布局成为了一个新的热点研究方向. 单一 NSM+DSM 布局方案是指根据负载的特性把数据较为精确地分开使用 NSM 结构和 DSM 结构组织, 但不会出现重复的元组或列数据, 比如, 经常被 OLTP 负载访问的

数据使用 NSM 的结构组织, 而经常被 OLAP 负载访问的数据则使用 DSM 的结构组织. 较具代表性的技术有 Hyrise、Peloton 系统以及数据分层技术.

Hyrise 是一个主存混合存储引擎, 它支持细粒度混合存储模型, 在不同的垂直分区中, 数据可以按照元组的布局方式组织, 每个分区称为一个容器, 容器以大型连续内存块的列表形式物理存储, 不同数据类型被字典压缩到固定长度字段中, 以允许直接访问任何给定位置, 这种物理设计能够比纯行或纯列方法的混合负载扫描速度快 20%到 400%. 当负载为宽投影时, 与 HYRISE 相比, PAX 在扫描一个表中的许多列时会产生更多的缓存未命中(因为它必须从内存中的一列跳到下一列); 与 Data Morphing 方法相比, 前者没有对多种负载情况进行建模, 而 Hyrise 通过缓存未命中数来计算模型的细腻度, 并且对物理数据库设计算法进行优化. 此外, Hyrise 可以扩展到具有数十个或数百个属性的表, 而 Data Morphing 不能扩展到具有大量属性的表.

Peloton 是一个根据 HTAP 负载设计的智能数据库系统, 它提出一种融合了 NSM 和 DSM 的灵活存储模型(flexible storage model, FSM)和一种基于 Tiles 的 DBMS 体系结构. Peloton 使用基于 Tile 的抽象存储结构来描述 FSM 的表现形式, 一个逻辑 Tile 类似于表格的垂直或水平段, 隐藏物理 Tile 的具体细节. 图 11 中颜色相同且相连的部分组成一个 Physical Tile, 即作为内存中存储的一个单位. 若干个元组构成一个 Physical Tile, 若干个 Physical Tile 组成一个 Tile Group, 属于同一个 Group 的 Physical Tile 包含相同数目的元组. 根据不同的工作负载, 可对同一个表构建不同的 Tile 分配方式. 例如: 一个 Tile 包含 1~m 个属性列, 当每个 Tile 属性数减小时, 布局向 DSM 收敛; 相反, 当 Tile 属性数增加时, 布局向 NSM 收敛, 如 Tile Group C. 因此, FSM 能够适应复杂负载的特征, 充分发挥 Adapter Store 的特性.

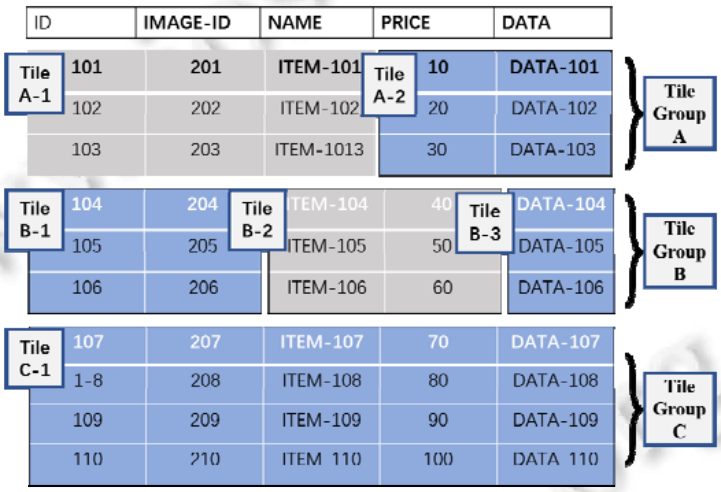


图 11 Physical Tile 和 Tile Group 存储示例

对于一个执行引擎无法处理存储在不同存储布局中的数据这一问题, Peloton 认为, 使用多个执行引擎进行查询处理, 这些引擎需要针对不同布局进行优化, 再把不同执行引擎中的运算符生成结果结合到一起, 这会导致高昂的代价. 故 Peloton 通过引入 Logical Tile, 将布局细节隐藏在执行引擎中, 允许 DBMS 在不同布局上执行查询计划而无需使用多个执行引擎. 图 13 描述了逻辑 Tile 到 Physical Tile 的物化过程, 逻辑 Tile X 指向两个 Physical Tile: A-1 和 A-2 的数据, 第 1 列逻辑 Tile 存放 Physical Tile A-1 前两个属性列的前 3 行元组, 第 2 列逻辑 Tile 存放物理 Tile A-1 的第 3 列属性和物理 Tile A-2 的第 1 列属性, 第 3 列逻辑 Tile 存放物理 Tile A-2 的第 1 列属性. 在查询执行期间, DBMS 动态地选择将逻辑 Tile 中间查询结果映射成 Physical Tile. 假设根据查询负载, 查询处理引擎构造了偏移列表分别为(1,2,3)、(1,1,1)、(1,2,2)的 3 个逻辑 Tile, DBMS 对逻辑 Tile 物化, 如图 12 中逻辑 Tile 的第 1 行第 1 列中的值表示 A-1 第 1 行元组的前两个属性中的值, 在物化过程中, DBMS 将其转换为{101,201}.

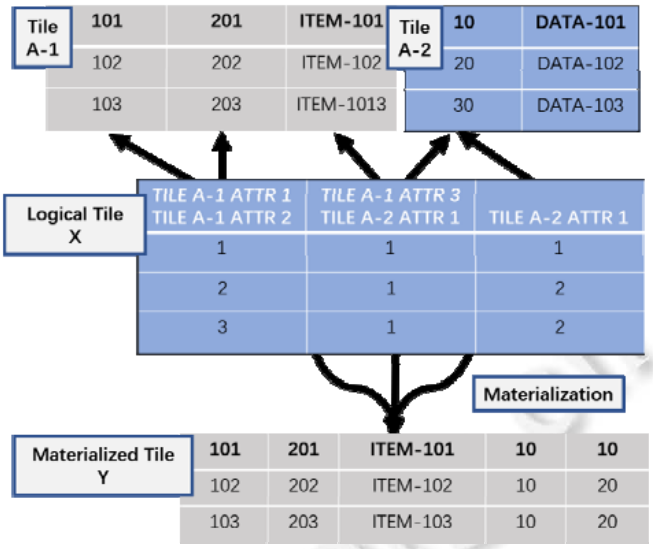


图 12 逻辑 Tile 和物理 Tile 之间的映射

Pareto[69]基于 Hyrise 研究混合存储引擎的数据库分区和布局问题, 它主要关注数据分层以及如何使用混合数据结构来改进冷数据驱逐和热数据加载. 通过设计一个主存预算模型, 结合整数线性规划进行列选择, 将其以列存的方式存储在主存中, 而其余的属性以行存的布局存放在二级存储器中. 此外, 为二级存储添加更多的格式, 如磁盘优化列格式, 以便于其仍然允许扫描很少过滤的属性. 分层表的每一列都以一种单独的格式完整地存储, 而不需要任何复制操作或其他数据结构. 如图 13 所示, 内存驻留列(memory-resident column, MRC)执行面向列存储的顺序读操作, 例如过滤和连接, 使用 Order-Preserving 字典编码, 在 MRCs 上也应用了著名的优化技术, 如矢量化、SIMD 以及处理延迟物化的压缩数据. 辅助存储列组(secondary storage column group, SSCG)类似于磁盘的行存, 主要针对以元组为中心的访问, 例如元组重建或 Probing 进行优化. 当这种数据分层策略用于处理大量数据被逐出到辅助存储时, 它可以保持内存中数据库的性能, 进而解决了 Database Cracking[51]、Auto Admin、SPA HANA 的 Storage Advisor[70,71]等优化技术存在的忽略连续列过滤影响的缺陷.

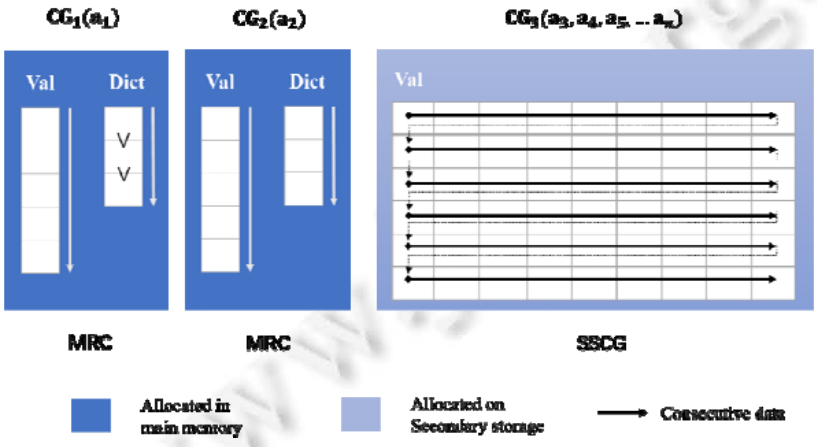


图 13 含有 3 个列组(内存驻留列组、辅助存储列组)的混合表格布局存储示例

表 5 分别从布局结构、存储介质、Adapter Store 特性以及特点这 4 个角度对比了常见的冗余 NSM+DSM 技术的异同. Hyrise 从缓存未命中数层面设计细腻度的主存预算模型, 通过学习工作负载特征, 指导划分更好

的列组方案. Peloton 旨在实现智能数据库 Adapter Store 的概念, 通过引入 Logical Tile 对混合存储布局进行抽象, 避免混合存储引擎增加的复杂性为数据库的高级功能带来了新的复杂性. Pareto 是 2018 年提出的研究工作, 它总结磁盘和内存数据库关于混合数据布局领域的发展现状, 认为在主存驻留数据库中进行数据分层设计, 将冷数据驱逐到较廉价的存储层, 具有很大的发展潜力.

表 5 单一 NSM+DSM 技术对比

文献 / 角度	Hyrise[35]	Peloton[37]	Pareto[69]
布局结构	列组	灵活存储结构 FSM	内存驻留列和辅助存储列组
存储介质	内存	内存	内存和磁盘
Adapter Store 特性	无	有	无
特点	构建细腻度缓存未命中数成本模型; 可以扩展到宽表	引入 Logical Tile 无需多个查询执行引擎; 自适应查询负载	采用数据分层应对 HTAP 负载以减少辅助存储对性能的负面影响

2.5 小 结

智能数据布局为数据库社区的布局结构转变提供了新的方向, 完全抛弃 OLAP 和 OLTP 分离的解决方案, 提供多种优化技术和新的布局结构, 以适用于 HTAP 场景, 减少应用开发多个引擎的开发成本和维护成本. 本节从传统的行存与列存布局角度出发, 主要从基于 PAX 的布局方案、基于冗余 NSM+DSM 的布局方案和基于单一 NSM+DSM 的布局方案这 3 个方向介绍了智能数据布局的发展概况. 基于 PAX 的布局方案设计了全新的布局结构, 以解决在传统行存/列存布局上优化的局限性; 基于冗余 NSM+DSM 的布局方案则利用额外的存储空间对不同的负载有针对性地建立不同布局的数据副本, 以解决单一布局带来的冲突和妥协问题; 而基于单一 NSM+DSM 的布局方案则通过负载驱动设计了更加精细的布局方案和转换算法, 以解决冗余布局方案所带来的巨大空间存储和数据同步成本问题. 表 6 从布局结构、重组触发机制这两个角度对本节的重要技术进行归纳. 虽然数据布局领域现有的研究工作已经比较完善, 但仍然存在以下几个具有挑战性的问题亟待解决. 首先, 现有的数据布局策略往往仅从负载执行与存储成本等较单一的角度来进行优化, 但数据库系统作为一个不可分割的整体, 如何结合数据布局的改变对数据分区、分布式事务、数据压缩、新硬件存储特性的适配等其他方面所带来的影响来进行较为综合的优化还有待探究; 同时, 由于目前还没有专门面向 HTAP 数据库系统的统一测评负载, 所以各个面向 HTAP 负载而设计的布局模式的效果还难以进行公平而客观的比较.

表 6 数据库布局技术总结

文献	布局结构				布局重组策略
	NSM	DSM	PAX	冗余/单一	
C-Store[57]	–	√	–	冗余	DBA/自动计划脚本
L-Store[60]	–	√	–	冗余	定时重组(周期)
MonetDB[58]	–	√	–	单一	–
Database cracking[61]	–	√	–	单一	定时重组
GSOP-R[4]	–	√	–	冗余	启发式重组
PAX[64]	–	–	√	单一	–
NoisePage[66]	–	–	√	单一	启发式
Data Morphing[65]	–	–	√	冗余	定时重组
Peloton[37]	√	√	–	单一	增量执行
Fractured Mirrors[67]	√	√	–	冗余	定时重组
AutoStore[33]	√	√	–	单一	静态窗口
H20[34]	√	√	–	冗余	动态窗口
HANA[51]	√	–	–	冗余	增量执行
Hyper[51]	–	–	√	冗余	定时重组

3 智能数据分区与布局重组

仅仅考虑分区与布局结构的设计而忽略 DBMS 波动的负载, 将无法有效发挥新储存结构的性能提升. 传统的存储结构重组方式依赖于 DBA 根据商业智能软件的实际运行场景, 通过调整分区与布局结构来提高数

据库的运行性能. 但在实际情况下, 即便是有经验的数据库管理员, 也很难在复杂的负载下寻找合适的时间点调整最佳的存储策略. 为了避免这种循环复杂的工作任务, 如何结合机器学习和深度学习, 让机器自动调优并减少转换开销, 是智能重组技术研究的关键. 现阶段的智能重组技术通常在智能分区与布局技术的基础上, 根据负载特征和规模确定重组触发的阈值, 以执行最优的重组策略. 目前常用的智能重组方法有传统重组策略、基于监视窗口、基于启发式规则, 图 14 给出了本节内容的结构图.

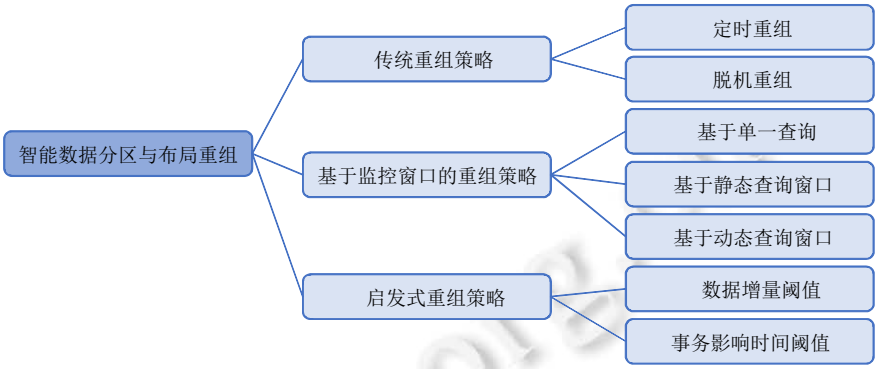


图 14 智能数据分区与布局重组结构图

3.1 传统重组策略

在数据库使用较长一段时间后, 由于一些更新操作, 记录会分布在更多的数据页上, 表和索引的大小也会增加, 相关数据会变得混乱, 从而影响了数据库性能. 数据库布局重组从索引、单表、表空间等方面进行空间回收, 以加速数据访问. 本节将介绍数据库重组的传统方式——脱机定时重组.

虽然脱机重组会造成数据库脱机, 但是相对于在线重组, 会更快、更简洁. 在脱机重组中, 数据从数据库导出到转储文件, 然后数据库对象重新排序并进行备份, 最后被传回到原来的表空间. 其中, 索引被隐式恢复. Data Morphing 是一个代表性的脱机定时重组技术, 只有被频繁访问的页才会被重组, 具有不同分区的页面可能共存于同一文件中. 完全由工作负载决定存储模式的 Octopus DB 系统, 通过设置 Check Point 定期清理中央日志进行布局重组. 需要注意的一点是: 脱机重组在 Unload 期间更新表空间将会造成数据不一致. C-Store 通过设计一个自动计划脚本, 定期地将所有查询的日志作为训练负载, 以确定投影、段、排序键和连接索引的集合. L-Store 对基本页和尾页进行周期性、无争用的合并, 以便将最近的更新数据合并到基本页.

3.2 基于监视窗口的重组策略

传统的重组方法无法适应动态 HTAP 工作负载, 定时的脱机重组不仅增大了 DBA 的管理工作量, 还会影响数据库的实时性能表现. 为此, 有相关研究提出了适应动态负载访问窗口的重组触发机制. 基于负载访问窗口的重组策略通常是设定可容纳一定查询负载的监视窗口, 当窗口中负载达到指定条件时, 触发布局重组操作, 并根据负载特点来指导布局重组过程. 常见的监视策略有单一查询语句、静态窗口、动态窗口.

基于单一查询语句进行重组的观点在文献[51]中被首次提出, 通过将布局重组作为查询评估过程的一个子部分, 以不断调整数据库组织方式, 即数据库存储结构分裂的可行性. Database Cracking 进一步具体数据库分裂技术的实现过程, 在查询评估方面, 每个查询不仅被解释为对特定结果集的请求, 还被解释为将物理数据库存储拆分为更小的部分, 它用一个查询来描述每一部分, 根据查询负载, 动态排序列存中的元组, 并使用辅助数据结构最小化元组重建成本. 所有查询都被组合在一个 Cracker 索引中, 以加速将来的搜索, 这也是根据单一查询进行布局优化的技术代表.

AutoStore 构建了动态工作负载监视器的组件, 主要解决何时进行重新分区操作. 通过设计一个定长 N 的查询窗口, 当静态窗口被查询填满时, 每扫描一个新的查询操作, 窗口滑动一下, 滑动固定次数 N 后, 触发一次重新分区操作, 以解决动态负载的自动分区问题.

H2O 对 AutoStore 的静态窗口进行改进, 它使用可容纳若干个查询语句的动态窗口监视工作负载变化. 将新查询与上一个监视窗口中观察到的查询进行比较, 以检测工作负载变化, 根据监视窗口的统计信息及时对监视窗口的大小 N 做出调整, 即负载变化显著时, 可以通过减小监视窗口以逐步协调产生新的分区策略, 而当工作负载趋于稳定时, 可以增大监视窗口. 为了减少对查询延迟的影响, H2O 在后台执行数据重组.

3.3 启发式重组策略

数据仓库不经常进行更新操作, 数据通常批量插入或批量删除. 频繁的数据重组会导致过高的重组成本, 因此比较适合采用启发式的重组策略. 常用的启发式重组策略有基于数据增量阈值的重组策略和基于事务访问时间阈值的重组策略.

SOP[17]是一种基于数据增量的启发式重组策略, 由于该框架中分区算法执行后产生了尺寸平衡的数据块, 插入的元组应该放入新的块中, 而不是修改现有的块, 这些新块可以很好地为即将到来的查询语句服务. 一旦积累了足够容量的新数据, 系统就会调用 SOP 中的算法, 重新组织这些数据生成新的分区策略.

Qd-tree[5]从离线数据中学习一个分区函数, 以进行动态的数据摄取. 同时, 为了节省分区重组成本, Qd-tree 主要关注当前分析的元组与即将输入的元组具有相同分布的场景, 即使处理动态的数据, 重新分区的问题仍然可以转化成为对静态数据的分区问题.

NoisePage[66]则在基于列式存储上进行的 OLTP 事务优化的布局方案设计上, 用户可以根据他们希望系统转换冷热块的积极程度来修改阈值的时间值, 在阈值时间内未被事务进行修改操作的块将会被标记为主要采用列存优化的冷块, 但阈值过低会降低事务性能, 频繁的转换会浪费资源, 设置得太高会降低读取的效率, 因此, 基于工作负载来选取最佳的时间值, 是一个重要的研究方向.

3.4 小 结

针对需要实时更新数据并执行查询的系统, 或者虽然是静态但是前后负载变化较大的情况, 静态且单一的分区与布局策略无法对全局的访问操作做出最好的应答, 故寻找一种能够触发数据重新组织的机制至关重要. 上文主要从传统的策略、基于监视窗口和启发式这 3 个方面介绍了这一机制. 表 7 从存储结构重构的层次、重组触发机制、触发条件这 3 个角度对常见的几种重组技术进行了归纳. 智能数据分区与布局重组技术虽然在一定程度上缓解了静态且单一的分区与布局策略的局限性, 但现有的数据分区与布局重组技术大多只能基于当前已经到来的负载进行组织和优化, 是一种在问题发生后的补救措施, 这对于一些低延迟的应用场景是不能接受的. 所以, 如何预测即将到来的负载并提前做好分区与布局重组准备, 是一个需要亟待解决的问题.

表 7 智能数据存储重组技术总结

文献	存储结构重构层次			重组触发机制	触发条件
	分区	布局			
		结构	数据		
Data Morphing[72]	–	√	–	定时重组	时间
Octopus DB[68]	–	√	–	检查点定期检测	时间
C-Store[58]	–	–	√	DBA/自动计划脚本	周期时间
L-Store[60]	–	–	√	定时重组	时间
Database cracking[61]	–	–	√	单个或多个查询语句	查询语句
NoisePage[66]	–	√	–	启发式	修改操作时间
AutoStore[33]	√	–	–	固定窗口	负载容量
H20[34]	√	–	–	动态窗口	负载特征变化
SOP[17]	√	–	–	新增元组数量	新增负载数量
QD-tree[5]	√	–	–	新增元组数量	新增负载数量

4 总结与展望

近年来, 智能数据库分区与布局的研究越来越受到数据库领域的关注, 并取得了一系列的研究进展. 本文主要从智能数据库分区、智能数据库布局以及智能数据库分区与布局重组这 3 个方面介绍了大数据时代智

能化方法在数据库存储优化领域的应用与实践. 智能数据库分区是指使用机器学习算法或者高级统计方法来分析给定负载, 从而指导生成比传统分区策略更加高效或者更加细粒度的数据水平划分、垂直划分或者混合划分方案; 智能数据布局则是通过设计全新的布局方案、各种布局策略叠加混合使用或者对数据布局进行动态的转换, 来解决大数据时代背景下单一的传统行存与列存布局难以应对的各种复杂 HTAP 负载场景所带来的挑战. 智能数据库分区与布局重组则是结合机器学习与深度学习方法设计多种策略, 使数据库能够在线地、自动地对由波动的数据与负载引起的过期分区与布局进行更新重组. 最后, 本文提出一些智能数据库分区与布局存在的挑战性问题, 并对其未来可能的研究方向做出以下展望.

- 智能数据分区技术方向. 现有的数据分区技术存在以下挑战性问题: 首先, 目前对各种分区算法成本的评估尚缺少统一的代价模型, 有的只是通过通用基准测试其扫描效率, 有的模型则采用基于跳过的成本估计模型, 而有的则专一对磁盘 I/O 开销或 CPU 效率进行成本估算, 缺乏统一的代价模型, 使得各类模型之间难以进行比较; 此外, 现有的大部分人工智能赋能的数据分区优化技术通常只使用了较为简单的浅层机器学习模型, 如频繁项集挖掘、简单聚类算法、图分割算法、强化学习等, 传统的模型往往只能寻找局部最优解, 并存在时间复杂度高、计算精读低、难以处理大批量数据等局限性. 近年来, 深度学习技术的发展突飞猛进, 产生了众多相比于传统机器学习算法更加强大且高效的研究成果, 如图卷积神经网络[73]、深度强化学习[74]等方向, 如何将这些最新的深度学习技术迁移应用到智能数据分区领域, 是一个非常具有前景的研究方向;
- 智能数据布局技术方向. 数据布局领域现有的研究工作已经比较完善, 但仍然存在以下几个具有挑战性的问题亟待解决: 首先, 现有的数据布局策略往往仅从负载执行与存储成本等较单一的角度切入来进行优化, 但数据库系统作为一个不可分割的整体, 如何结合数据布局的改变对数据分区、分布式事务、数据压缩、新硬件(如固态硬盘、非易失性内存和 SMR 磁盘等)存储特性的适配等其他方面所带来的影响来进行较为综合的优化还有待探究; 同时, 由于目前还没有专门面向 HTAP 数据库系统的统一测评负载, 所以各个面向 HTAP 负载而设计的布局模式的效果还难以进行公平、客观的比较, 因此, 建立统一的 HTAP 测评基准, 也将会是一项非常有价值的工作;
- 智能数据分区与布局重组技术方向. 智能数据分区与布局重组技术虽然在一定程度上缓解了静态且单一的分区与布局策略的局限性, 但现有的数据分区与布局重组技术大多只能基于当前已经到来的负载进行组织和优化, 是一种在问题发生后的补救措施, 这对于一些低延迟的应用场景是不能接受的. 所以, 如何预测即将到来的负载并提前做好分区与布局重组准备, 是一个需要亟待解决的问题. 目前, 有关未来负载预测已经有一定的研究[75−77], 但这些技术并没有运用到数据分区和布局重组技术的优化方面. 如何对将要到来的负载进行更精准的预测, 并且根据未来负载提前做出最优的分区和布局重组策略, 是未来一个非常有价值的研究方向.

References:

[1] Lee K, Liu L. Scaling queries over big RDF graphs with semantic hash partitioning. Proc. of the VLDB Endowment, 2013, 6(14): 1894−1905.

[2] Copeland GP, Khoshafian SN. A decomposition storage model. In: Proc. of the 1985 ACM SIGMOD Int'l Conf. on Management of Data. 1985. 268−279.

[3] Sun LM, Zhang SM, Ji T, Li CP, Chen H. Survey of data management techniques powered by artificial intelligence. Ruan Jian Xue Bao/Journal of Software, 2020, 31(3): 600−619 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5909.htm [doi: 10.13328/j.cnki.jos.005909]

[4] Sun L. Skipping-oriented data design for large-scale analytics [Ph.D. Thesis]. UC Berkeley, 2017.

[5] Yang Z, Chandramouli B, Wang C, *et al*. QD-tree: Learning data layouts for big data analytics. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. 2020. 193−208.

[6] Ziauddin M, Witkowski A, Kim YJ, *et al*. Dimensions based data clustering and zone maps. Proc. of the VLDB Endowment, 2017, 10(12): 1622−1633.

[7] Moerkotte G. Small materialized aggregates: A light weight index structure for data warehousing. In: Proc. of the 24th Int'l Conf. on Very Large Data Bases. 1998. 476−487.

[8] Gupta A, Agarwal D, Tan D, *et al*. Amazon redshift and the case for simpler data warehouses. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. 2015. 1917−1923.

[9] Raman V, Attaluri G, Barber R, *et al*. DB2 with BLU acceleration: So much more than just a column store. Proc. of the VLDB Endowment, 2013, 6(11): 1080−1091.

[10] Hall A, Bachmann O, Büssow R, *et al*. Processing a trillion cells per mouse click. Proc. of the VLDB Endowment, 2012, 5(11): 1436−1446.

[11] Dageville B, Cruanes T, Zukowski M, *et al*. The snowflake elastic data warehouse. In: Proc. of the 2016 Int'l Conf. on Management of Data. 2016. 215−226.

[12] Ślęzak D, Wróblewski J, Eastwood V, *et al*. Brighthouse: An analytic data warehouse for ad-hoc queries. Proc. of the VLDB Endowment, 2008, 1(2): 1337−1345.

[13] Huai Y, Chauhan A, Gates A, *et al*. Major technical advancements in apache hive. In: Proc. of the 2014 ACM SIGMOD Int'l Conf. on Management of Data. 2014. 1235−1246.

[14] Apache Parquet. http://parquet.apache.org

[15] Graefe G. Fast loads and fast queries. In: Proc. of the Int'l Conf. on Data Warehousing and Knowledge Discovery. Berlin, Heidelberg: Springer, 2009. 111−124.

[16] Athanassoulis M, Bøgh KS, Idreos S. Optimal column layout for hybrid workloads. Proc. of the VLDB Endowment, 2019, 12(13): 2393−2407.

[17] Sun L, Franklin MJ, Krishnan S, *et al*. Fine-grained partitioning for aggressive data skipping. In: Proc. of the 2014 ACM SIGMOD Int'l Conf. on Management of Data. 2014. 1115−1126.

[18] IBM Netezza data warehouse appliance. http://www.ibm.com/software/data/netezza/

[19] Amazon Redshift. Database developer guide (API Version 2012-12-01). Choosing sort keys. http://docs.aws.amazon.com/redshift/latest/dg/t_Sorting_data.html

[20] Yamamoto Y, Iwanuma K, Fukuda S. Resource-oriented approximation for frequent itemset mining from bursty data streams. In: Proc. of the 2014 ACM SIGMOD Int'l Conf. on Management of Data. 2014. 205−216.

[21] Moerkotte G, Neumann T. Dynamic programming strikes back. In: Proc. of the 2008 ACM SIGMOD Int'l Conf. on Management of Data. 2008. 539−552.

[22] Andersen ED, Andersen KD. The MOSEK Interior Point Optimizer for Linear Programming: An Implementation of the Homogeneous Algorithm——High Performance Optimization. Boston: Springer, 2000. 197−232.

[23] Olma M, Karpathiotakis M, Alagiannis I, *et al*. Slalom: Coasting through raw data via adaptive partitioning and indexing. Proc. of the VLDB Endowment, 2017, 10(10): 1106−1117.

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

[25] Eisner MJ, Severance DG. Mathematical techniques for efficient record segmentation in large shared databases. Journal of the ACM (JACM), 1976, 23(4): 619−635.

[26] March ST, Serverance DG. The determination of efficient record segmentations and blocking factors for shared data files. ACM Trans. on Database Systems (TODS), 1977, 2(3): 279−296.

[27] Schkolnick M. A clustering algorithm for hierarchical structures. ACM Trans. on Database Systems (TODS), 1977, 2(1): 27−44.

[28] Hoffer JA, Severance DG. The use of cluster analysis in physical data base design. In: Proc. of the 1st Int'l Conf. on Very Large Data Bases. 1975. 69−86.

[29] McCormick Jr WT, Schweitzer PJ, White TW. Problem decomposition and data reorganization by a clustering technique. Operations Research, 1972, 20(5): 993−1009.

[30] 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. 1989. 440−450.

[31] Navathe S, Karlapalem K, Ra M. A mixed fragmentation methodology for initial distributed database design. Journal of Computer and Software Engineering, 1995, 3(4): 395−426.

[32] Marir F, Najjar Y, AlFaress MY, *et al*. An enhanced grouping algorithm for vertical partitioning problem in DDBS. In: Proc. of the 22nd Int'l Symp. on Computer and Information Sciences. IEEE, 2007. 1−6.

[33] Jindal A, Dittrich J. Relax and let the database do the partitioning online. In: Proc. of the Int'l Workshop on Business Intelligence for the Real-time Enterprise. Berlin, Heidelberg: Springer, 2011. 65−80.

[34] Alagiannis I, Idreos S, Ailamaki A. H2O: A hands-free adaptive store. In: Proc. of the 2014 ACM SIGMOD Int'l Conf. on Management of Data. 2014. 1103−1114.

[35] Grund M, Krüger J, Plattner H, *et al*. Hyrise: A main memory hybrid storage engine. Proc. of the VLDB Endowment, 2010, 4(2): 105−116.

[36] Durand GC, Pinnecke M, Piriyev R, *et al*. 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. 2018. 1−7.

[37] 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. 2016. 583-598.

[38] Sun L, Franklin MJ, Wang J, *et al*. Skipping-oriented partitioning for columnar layouts. Proc. of the VLDB Endowment, 2016, 10(4): 421−432.

[39] Agarawal S, Chaudhuri S, Narasayya V. Automated selection of materialized views and indexes for SQL databases. In: Proc. of the 26th Int'l Conf. on Very Large Databases. Cairo, 2000. 191−207.

[40] Chaudhuri S, Narasayya VR. An efficient, cost-driven index selection tool for Microsoft SQL server. In: Proc. of the VLDB, Vol.97. 1997. 146−155.

[41] Rao J, Zhang C, Megiddo N, *et al*. Automating physical database design in a parallel database. In: Proc. of the 2002 ACM SIGMOD Int'l Conf. on Management of Data. 2002. 558−569.

[42] Zeller B, Kemper A. Experience report: Exploiting advanced database optimization features for large-scale sap *r*/3 installations. In: Proc. of the 28th Int'l Conf. on Very Large Databases (VLDB 2002). Morgan Kaufmann Publishers, 2002. 894−905

[43] Zilio DC, Jhingran A, Padmanabhan S. Partitioning key selection for a shared-nothing parallel database system. IBM TJ Watson Research Center, 1994. https://www.researchgate.net/publication/2623565_Partitioning_Key_Selection_for_a_Shared-Nothing_Parallel_Database_System

[44] 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. 2004. 359−370.

[45] 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.

[46] Cao Y, Chen C, Guo F, *et al*. ES 2: A cloud data storage system for supporting both oltp and OLAP. In: Proc. of the 27th IEEE Int'l Conf. on Data Engineering. IEEE, 2011. 291−302.

[47] Chen C, Chen G, Jiang D, *et al*. Providing scalable database services on the cloud. In: Proc. of the Int'l Conf. on Web Information Systems Engineering. Berlin, Heidelberg: Springer, 2010. 1−19.

[48] Kemper A, Neumann T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: Proc. of the 27th IEEE Int'l Conf. on Data Engineering. IEEE, 2011. 195−206.

[49] Lang H, Mühlbauer T, Funke F, *et al*. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In: Proc. of the 2016 Int'l Conf. on Management of Data. 2016. 311−326.

[50] Neumann T, Mühlbauer T, Kemper A. Fast serializable multi-version concurrency control for main-memory database systems. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. 2015. 677−689.

[51] Färber F, May N, Lehner W, *et al*. The SAP HANA database—An architecture overview. IEEE Data Engineering Bulletin, 2012, 35(1): 28−33.

[52] Banerjee S, Krishnamurthy V, Krishnaprasad M, *et al*. Oracle8i—The XML enabled data management system. In: Proc. of the 16th Int'l Conf. on Data Engineering. IEEE, 2000. 561−568

[53] Cheng J, Xu J. XML and DB2. In: Proc. of the 16th Int'l Conf. on Data Engineering. IEEE, 2000. 569−573.

[54] Informix object translator. 2001. http://www.informix.com/idn-secure/webtools/ot/

[55] SQL server magazine. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsqlmag2k/html/TheXMLFiles.asp

[56] Stonebraker M, Weisberg A. The VoltDB main memory DBMS. IEEE Data Engineering Bulletin, 2013, 36(2): 21−27.

[57] Stonebraker M, Abadi DJ, Batkin A, *et al*. C-Store: A column-oriented DBMS. In: Proc. of the 31st Int'l Conf. on Very Large Databases. 2005. 553−564.

[58] Blockhaus P, Broneske D, Schäler M, *et al*. Combining two worlds: MonetDB with multi-dimensional index structure support to efficiently query scientific data. In: Proc. of the 32nd Int'l Conf. on Scientific and Statistical Database Management. 2020. 1−4.

[59] Melnik S, Gubarev A, Long JJ, *et al*. Dremel: Interactive analysis of Web-scale datasets. Proc. of the VLDB Endowment, 2010, 3(1−2): 330−339.

[60] Sadoghi M, Bhattacherjee S, Bhattacharjee B, *et al*. L-Store: A real-time OLTP and OLAP system. In: Proc. of the 21st Int'l Conf. on Extending Database Technology (EDBT 2018). 2018.

[61] Idreos S, Kersten ML, Manegold S. Database cracking. In: Proc. of the CIDR, Vol.7. 2007. 68−78.

[62] Bian H, Yan Y, Tao W, *et al*. Wide table layout optimization based on column ordering and duplication. In: Proc. of the 2017 ACM Int'l Conf. on Management of Data. 2017. 299−314.

[63] Bian H, Tao Y, Jin G, *et al*. Rainbow: Adaptive layout optimization for wide tables. In: Proc. of the 34th IEEE Int'l Conf. on Data Engineering (ICDE). IEEE, 2018. 1657−1660.

[64] Ailamaki A, DeWitt DJ, Hill MD, *et al*. Weaving relations for cache performance. In: Proc. of the VLDB, Vol.1. 2001. 169−180.

[65] Hankins RA, Patel JM. Data morphing: An adaptive, cache-conscious storage technique. In: Proc. of the 2003 VLDB Conf. Morgan Kaufmann Publishers, 2003. 417−428.

[66] Li T, Butrovich M, Ngom A. Mainlining databases: Supporting fast transactional workloads on universal columnar data file formats. arXiv preprint arXiv: 2004.14471, 2020.

[67] Ramamurthy R, DeWitt DJ, Su Q. A case for fractured mirrors. The VLDB Journal, 2003, 12(2): 89−101.

[68] Dittrich J, Jindal A. Towards a one size fits all database architecture. In: Proc. of the CIDR. 2011. 195−198.

[69] Boissier M, Schlosser R, Uflacker M. Hybrid data layouts for tiered HTAP databases with Pareto-optimal data placements. In: Proc. of the 34th IEEE Int'l Conf. on Data Engineering (ICDE). IEEE, 2018. 209−220.

[70] Lee J, Muehle M, May N, *et al*. High-performance transaction processing in SAP HANA. IEEE Data Engineering Bulletin, 2013, 36(2): 28−33.

[71] Sikka V, Färber F, Lehner W, *et al*. Efficient transaction processing in SAP HANA database: The end of a column store myth. In: Proc. of the 2012 ACM SIGMOD Int'l Conf. on Management of Data. 2012. 731−742.

[72] Halim F, Idreos S, Karras P, *et al*. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. Proc. of the VLDB Endowment, 2012, 5(6): 502−513.

[73] Hamilton WL, Ying R, Leskovec J. Inductive representation learning on large graphs. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. 2017. 1025−1035.

[74] Mnih V, Kavukcuoglu K, Silver D, *et al*. Playing atari with deep reinforcement learning. arXiv preprint arXiv: 1312.5602, 2013.

[75] Higginson AS, Dediu M, Arsene O, *et al*. Database workload capacity planning using time series analysis and machine learning. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. 2020. 769−783.

[76] Ma L, Van Aken D, Hefny A, *et al*. Query-based workload forecasting for self-driving database management systems. In: Proc. of the 2018 Int'l Conf. on Management of Data. 2018. 631−645.

[77] Smyl S. A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting. Int'l Journal of Forecasting, 2020, 36(1): 75−85.

附中文参考文献:

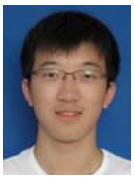
[3] 孙路明, 张少敏, 姬涛, 李翠平, 陈红. 人工智能赋能的数据管理新技术研究. 软件学报, 2020, 31(3): 600−619. http://www.jos.org.cn/1000-9825/5909.htm [doi: 10.13328/j.cnki.jos.005909]

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.

刘鹏举(1997－), 男, 博士生, 主要研究领域为智能数据库, 数据库存储, 负载预测, 基于深度学习的组合优化问题研究.

王天一(1999－), 女, 硕士生, 主要研究领域为数据分区.

何雨琪(2001－), 女, 本科生, 主要研究领域为数据分区, GPU 数据库.



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

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.