文章编号: 1000-5641(2018)05-0067-12

面向 Cedar 的列存储设计与实现

俞文谦, 胡 爽, 胡卉芪

(华东师范大学 数据科学与工程学院, 上海 200062)

摘要: 随着数据规模和分析需求的日益增长, 数据库面向联机分析处理 (On-Line Analytical Processing, OLAP) 应用的查询性能变得愈发重要. Cedar 是一款基于读写分离架构的分布式关系数据库, 由于它主要面向联机事务处理 (On-Line Transaction Processing, OLTP) 业务, 在面对分析处理负载时性能表现不足. 对于这个问题, 很多研究表明列存储技术能够有效地提高 I/O (Input/Output) 效率, 进而提升分析处理的性能. 在 Cedar 上提出了一种列存储机制, 分析了其适用场景并针对这种机制改进了 Cedar 的数据扫描和批量更新方法. 实验结果表明, 该机制能大幅度地提升 Cedar 分析处理性能, 并且对事务处理性能的影响控制在 10% 以内.

关键词: 分布式数据库; 列存储; 联机分析处理 (OLAP)

中图分类号: TP392 **文献标志码**: A **DOI:** 10.3969/j.issn.1000-5641.2018.05.006

The designs and implementations of columnar storage in Cedar

YU Wen-qian, HU Shuang, HU Hui-qi

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

Abstract: With the growing size of data and analytical needs, the query performance of databases for OLAP (On-Line Analytical Processing) applications has become increasingly important. Cedar is a distributed relational database based on read-write decoupled architecture. Since Cedar is mainly oriented to the needs of OLTP (On-Line Transaction Processing) applications, it has insufficient performance for handling analytical processing workloads. To address this issue, many studies have shown that column storage technology can effectively improve the efficiency of I/O (Input/Output) and enhance the performance of analytical processing. This paper presents a column-based storage mechanism in Cedar. The study analyzes applicable scenarios and improves Cedar's data query and batch update methods for this mechanism. The results of an experiment demonstrate that the proposed mechanism can enhance the performance of analytical processing substantially, while limiting the negative impacts on transaction processing performance to within 10%.

Keywords: distributed database; column-based storage; OLAP

收稿日期: 2018-07-09

基金项目: 国家自然科学基金 (61702189); 上海市青年科技英才扬帆计划 (17YF1427800)

第一作者: 俞文谦, 男, 硕士研究生, 研究方向为分布式数据库系统. E-mail: wqyu_cs@163.com.

通信作者: 胡卉芪, 男, 助理研究员, 研究方向为数据库. E-mail: hqhu@dase.ecnu.edu.cn.

0 引 言

近年来, 计算机硬件的快速发展使得各行各业中数据的采集、存储、传输及处理能力日益提升, 数据规模逐渐扩大, 对数据分析处理的需求也不断增长. OLAP[1]是一种帮助企业从大量数据中提取价值信息的软件技术, 能为企业管理人员提供有效的决策支持. 在应用需求的推动下, OLAP 技术迅速发展起来. 与此同时, 数据库面向 OLAP 应用的查询性能也越来越受到研究学者的关注.

Cedar[2]是由华东师范大学数据科学与工程学院基于 OceanBase 0.4.2 研发的高通量、可伸缩、高可用的分布式关系数据库. 作为一款主要面向 OLTP 业务的数据库, Cedar 使用行式存储的方式组织数据, 以整行为单位操纵数据. 然而分析处理型负载的特点是涉及的数据量特别大, 并且只需要获取少数属性列的数据. 这使得 Cedar 在面对分析处理型负载时会有大量冗余的 I/O 开销, 导致查询性能表现不足. 因此, 优化分析处理的性能成为了 Cedar 面临的挑战之一.

对于这个问题, 很多学者的研究表明列存储技术能够有效地提高 I/O 效率, 进而提升分析处理的性能. 1985 年, Copeland等人首次提出并分析了列存储模型[3]. 与行存储相比, 列存储能够对数据进行更细粒度地读取, 查询时只需从磁盘读取需要的属性, 无需读取其他冗余的属性. 因此, 列存储在分析处理负载下能表现出更高的 I/O 效率[4]. 另外, 列存储的各列数据类型相同且列值连续存储, 使得采用编码技术获得的压缩效果通常优于行存储[5]. 随着数据规模和分析需求的急速增长, 列存储的优势变得越来越突出, 围绕着列存储的研究与实践也不断涌现出来: 相关技术如 Petraki等人提出的列存储上自适应的一体化索引[6]、Lang等人提出的针对冷数据压缩的列存储格式 Data Blocks[7]等; 基于行列混合存储的数据库系统如 SnappyData[8]、HANA ATR[9]等; 基于新型硬件 GPU 的数据库系统如 SQream DB[10]、MapD[11]等.

综上所述, 为提高 Cedar 面向 OLAP 应用的查询性能, 满足企业数据分析处理的需要, 本文基于 Cedar 设计并实现了列存储机制. 本文的主要贡献如下.

(1) 基于 Cedar 读写分离架构的特点设计了列存储机制, 并分析了其适用的场景.

(2) 针对列存储机制, 提出并分析了列存储数据扫描和批量更新算法.

(3) 在 Cedar 上实现了列存储机制, 并通过实验验证了该机制能大幅度地提升 Cedar 的查询性能, 且对其事务处理性能影响极小.

论文内容安排如下: 第 1 节描述 Cedar 系统架构; 第 2 节阐述 Cedar 列存储机制的设计, 分析 Cedar 列存储机制的适用场景; 第 3 节介绍 Cedar 列存储机制的实现, 包括列存储文件格式、基于该格式的数据扫描及批量更新算法; 第 4 节通过实验验证该列存储机制的高效性和分析其适用场景; 第 5 节总结全文.

1 Cedar 系统架构

OceanBase[12]是阿里巴巴研发的分布式关系数据库, 现已广泛应用于阿里巴巴、蚂蚁金服等关键业务. Cedar 在 OceanBase 0.4.2 开源版本上, 新增了高可用的三集群架构、存储过程、二级索引、快照隔离级别、可扩展事务提交等特性. Cedar 系统架构如图 1 所示.

Cedar 由一主集群、两个备集群组成; 在单个集群中有 4 类服务器, 是 RootServer (RS)、UpdateServer (UPS)、MergeServer (MS), 以及 ChunkServer (CS). RS 是 Cedar 的总控节点, 管理 CS 和 MS 的上下线、元数据及数据的分布, 负责集群之间的选主, 在一个集群中只有一

个 RS. UPS 是 Cedar 集群中唯一能接受写入的服务器, 负责执行数据更新的操作, 在内存表中维护着增量数据, 一个集群中只有一个 UPS. MS 负责接收并解析客户端的查询请求, 对请求中的 SQL 语句进行词法分析、语法分析及生成执行计划等步骤后, 将请求转发到相关的服务器, 最后将合并结果返回给客户端, 在一个集群中可部署多台 MS. CS 负责存储着基线数据, 在一个集群中可部署多台 CS.

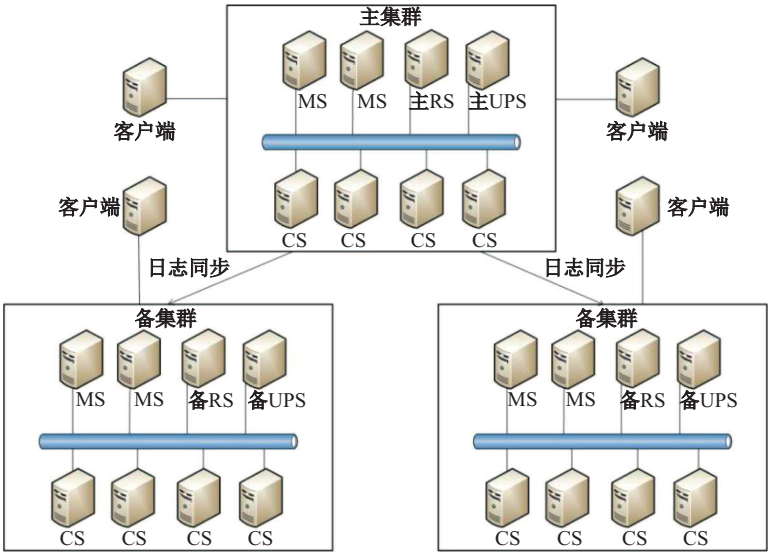

图 1　Cedar 系统架构

Fig. 1　The architecture of Cedar

Cedar 采用读写分离的结构, 把数据分为基线数据和增量数据[13]: 基线数据是只读的, 由 CS 负责存储和管理; 增量数据是新增、删改、更新的数据, 存储在 UPS 的内存中. 受限于 UPS 内存大小, 当增量数据的大小达到阈值时, Cedar 会冻结当前的增量数据, 且将其与原基线数据进行合并, 最终生成新基线数据.

2　列存储机制的设计

2.1　设计思路

Cedar 列存储机制的设计目标是提升其分析处理性能, 同时不影响其事务处理性能. 首先, 对 Cedar 的架构进行分析. 在 Cedar 中数据的更新不会立即持久化到磁盘上, 而是先保存在 UPS 的内存中, 等到合并阶段才进行批量持久化, 故 Cedar 具有“延迟更新, 批量写入”的特点. 对于列存储机制, “延迟更新”使得原有事务处理性能不受到影响; “批量写入”使得数据存储方式的选择更加灵活, 在合并阶段采用列式存储方式, 从而提高 Cedar 分析处理性能. 考虑到应用负载日益多元化, 行式存储和列式存储都有各自适用的场景. 为了让性能达到最优, Cedar 需要同时具备两种数据访问方法. 因此, 本文在原 Cedar 架构基础之上, 新增一个读写列存储数据的列式访问方法模块. 由于基线数据的存储与管理由 CS 负责, 故本文提出的列存储机制实现在 CS 上. 列存储机制架构设计如图 2 所示.

图 2 中 CS 的数据存取器用于磁盘存储访问, 实现数据的查询读取以及增量更新数据的批量写入的功能. 在 CS 磁盘上的数据可以行、列存储格式分别进行组织, 行式存储的数据文件为 SSTable, 列存储的数据文件为 Parquet, 其数据访问接口分别由 CS 中的行式访问方

法、列式访问方法实现. 该设计的优势是, 受益于Cedar“延迟更新, 批量写入”的特点, 列存储机制下事务处理产生的数据更新不会引起磁盘I/O, 而是维护在内存中, 这保证了事务处理的良好性能. 值得注意的是, 由于存储方式的选择权转移到了用户上, 故用户能否选择合适的存储方式变得尤为重要. 本文在第2.3节中将对这个问题进行具体分析.

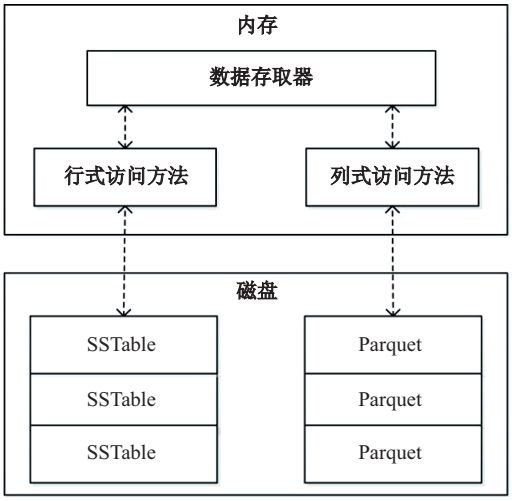

图2 CS上列存储机制架构

Fig. 2 The architecture of columnar storage on the CS

2.2 列存储数据的一致性

Cedar采用多副本策略管理数据, 将数据的副本存放在多个CS上. 通常情况下, Cedar对列存储数据进行访问时, 需要融合CS的基线数据和UPS的增量数据, 再将符合查询条件的数据返回. 由于此时CS的基线数据是只读的, 故在多个CS上读到的数据是相同的. 然而在合并阶段期间, 每一个CS的合并进度会各不相同. 例如一些CS已经完成数据的合并, 生成新版本的Parquet, 而另一些CS还未完成. 这就造成了一种情况的出现: 执行相同的查询语句, 从不同的CS上读取到Parquet的内容可能不同. 对此, Cedar为保证查询结果是一致的, 采取的策略是: 如果查询使用是旧版本的Parquet, 会将冻结数据和新的增量数据融合; 如果查询使用的是新版本的Parquet, 则只融合新的增量数据[12]. 如此, 列存储机制下数据的一致性得以保证.

2.3 列式存储适用场景的分析

当Cedar具有两种存储方式后, 要使Cedar的性能得到充分发挥, 用户需要考虑选择适合的存储方式. 为解决这个问题, 本文对Cedar在不同存储方式下的数据访问代价进行了对比分析, 得出了列式存储的适用场景. 以下列面向OLAP应用的查询为例进行分析, 令该查询为Query, 具体如下.

SELECT Class, COUNT(ALL Stu_name) AS Total FROM Student GROUP BY Class;

对于查询Query, 首先MS对其进行SQL解析, 生成并执行查询计划, 将数据读取请求发送到基线数据所在CS上; CS收到请求后从磁盘读取基线数据到内存, 同时向UPS拉取增量数据, 然后CS合并两种数据后将结果集返回给MS; MS对结果集进行处理后将最终结果返回客户端. 由于两种存储方式的区别在于磁盘上数据的组织方式, 因此本文对Cedar数据访问代价的对比分析主要集中在CS上. Cedar的数据访问需要经过磁盘数据传输和预处理两

个阶段, 其代价 Cost (Query) 近似为

$$\text{Cost}\,(\text{Query}) \approx \frac{k \cdot \text{Size}_{\text{block}}}{H_{\text{disk}}} + \text{Cost}\,(\text{preprocess}), \tag{1}$$

其中, k 为执行查询 Query 需要读入磁盘的数据块数目, $\text{Size}_{\text{block}}$ 为主存与磁盘之间传输数据单位的大小, H_{disk} 为磁盘在单位时间内的数据传输量, Cost (preprocess) 为数据预处理的代价.

由于硬件环境和查询请求相同, 数据传输单位大小、数据传输能力、结果集大小均相等, 因此行式存储方式与列式存储方式的数据访问代价主要区别在于数据传输量和预处理的开销上.

对于行式存储方式, 多个记录之间连续放置, 且通常采用 Snappy[14]压缩. 查询时需将请求的记录集完整地读入到内存, 然后通过投影操作获取需要的列, 则 k 的近似值为

$$k_{\text{row}} \approx \frac{N_{\text{record}} \cdot \text{Size}_{\text{record}} \cdot R_{\text{compress}}}{\text{Size}_{\text{block}}} \tag{2}$$

其中, N_{record} 为查询需要读取的记录数, $\text{Size}_{\text{record}}$ 为平均每条记录的存储空间大小, R_{compress} 为使用 Snappy 格式的压缩比.

由于从磁盘读取到内存的数据需要进行 Snappy 格式解压操作, 故预处理代价为

$$\text{Cost}_{\text{row}}\,(\text{preprocess}) = \frac{k_{\text{row}} \cdot \text{Size}_{\text{block}}}{H_{\text{decompress}}}, \tag{3}$$

其中 $H_{\text{decompress}}$ 为单位时间内进行 Snappy 格式解压的数据量.

对于列式存储方式, Cedar 先对数据进行 RLE[15]编码, 再使用 Snappy 压缩降低磁盘存储的开销. 执行查询时只需读入请求需要的列, 无需读入多余的列, 则 k 的近似值为

$$k_{\text{column}} \approx \frac{N_{\text{record}} \cdot N_{a_i} \cdot \text{Size}_{a_i} \cdot R_{\text{encode}} \cdot R_{\text{compress}}}{\text{Size}_{\text{block}}}, \tag{4}$$

其中, N_{a_i} 为查询请求需要的列数目, $1 \leqslant N_{a_i} \leqslant p$, Size_{a_i} 为关系表 T 属性列的平均占用存储空间大小, R_{encode} 为 RLE 编码的压缩比.

与行式存储方式不同, 列式存储方式从磁盘读取到内存的数据不仅需要进行解压操作, 还需要进行解码操作. 因此预处理代价为

$$\text{Cost}_{\text{column}}(\text{preprocess}) = \frac{k_{\text{column}} \cdot \text{Size}_{\text{block}}}{H_{\text{decompress}}} + \frac{N_{\text{record}} \cdot N_{a_i} \cdot \text{Size}_{a_i} \cdot R_{\text{encode}}}{H_{\text{decode}}}, \tag{5}$$

其中 H_{decode} 为单位时间内 RLE 解码的数据量.

通过两种存储方式的数据访问代价对比分析, 可以推出行式存储的数据访问代价大于列式存储的数据访问代价时, 即 $\text{Cost}_{\text{row}}(\text{Query}) > \text{Cost}_{\text{column}}(\text{Query})$ 等价于不等式

$$\left(\frac{1}{H_{\text{disk}}} + \frac{1}{H_{\text{decompress}}}\right) \cdot R_{\text{compress}} \cdot (\text{Size}_{\text{record}} - N_{a_i} \cdot \text{Size}_{a_i} \cdot R_{\text{encode}}) > \frac{N_{a_i} \cdot \text{Size}_{a_i} \cdot R_{\text{encode}}}{H_{\text{decode}}}. \tag{6}$$

由式 (6) 可以看出, 当查询涉及的列数越少、RLE编码的压缩比越低、单位时间内 RLE 解码的数据量越大时, 列存储机制的数据访问代价相比行存储的越低. RLE 编码的压缩比与数据重复率有关, 这是因为 RLE 编码使用数值和数值重复的次数来表示一些连

续相同的数据, 从而达到压缩数据的效果. 例如字符串 AAABBBCCCC, 经过 RLE 编码后得到 3A3B4C, 降低了文件所占用的存储空间. 当数据重复率越高时, RLE 编码的压缩比就越低. 需要注意的是, 单位时间内 RLE 解码的数据量是由系统采用的算法决定的. 因此, 在用户的角度上, 需要考虑的主要因素有查询涉及的列数和数据重复率. 在查询涉及的列数较少、数据重复率较高的场景下, 采用列式存储方式更具有明显的优势. 该结论将在本文第 4 节的实验中进行验证.

3 列存储机制的实现

根据列存储机制的设计, 需要在 Cedar 上增加一个列式访问方法模块, 用于完成对列存储数据读取和写入的基本操作. 该模块的实现主要包括 3 个部分: 列存储文件格式、基于该格式的数据扫描算法及批量更新算法. 列存储文件格式是列存储机制中最基本的数据结构, 对列存储数据的操作都在其上进行; 数据扫描算法决定 Cedar 是如何访问列存储文件; 数据批量更新算法决定 Cedar 是如何生成列存储文件. 在本节中将依次介绍这 3 个部分.

3.1 列存储文件格式

列存储文件格式是 Cedar 列存储机制的核心部分, 它决定数据在磁盘上的组织方式, 关系着数据存取的效率和磁盘存储空间的开销. 考虑到减少对上层应用系统的侵入性, 提高系统应用场景的扩展性, 本文采用的列存储文件格式为 Parquet[16]. Parquet 是一种流行的开源列式存储格式, 支持投影下推和谓词下推, 并能使用多种数据压缩和编码方式处理海量数据. Parquet 数据结构如图 3 所示.

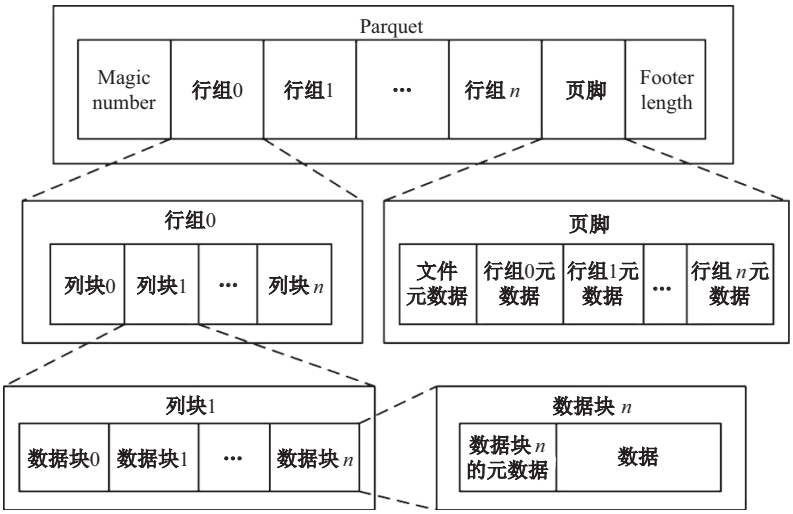

图 3 Parquet 数据结构

Fig. 3 The data structure of Parquet

Parquet 由 Magic number、行组、页脚和 Footer length 组成. Magic number 用来校验该 Parquet, 关系表被水平切分存放在多个行组中, 一个行组包含多个列块, 列块由多个数据块组成, 并且各数据块块头存放着该块的元数据. Footer length 存储着元数据的信息, 可利用它来定位页脚. 页脚存放着文件的元数据和各个行组的元数据.

3.2 列存储的数据扫描算法

面对分析处理型负载时, Cedar 需要从 CS 上获取列存储基线数据. 为此, 本文基

于 Parquet 的结构特点, 提出了一种数据扫描算法——列存储数据扫描算法. 该算法负责根据查询条件从 Parquet 中读取数据到内存, 实现列存储机制的数据访问功能, 具体见算法 1 所示. 由于 Parquet 的元数据记录该行组中各属性的最大值和最小值, 可通过这些值实现基于行组的过滤. 在载入行组元信息后, 由于列存数据文件按照逻辑行的主键排序, 如果查询条件中包含了主键, 则使用二分查找法过滤掉一部分的行组 (行 2-8); 然后使用除主键以外的列查询条件, 遍历剩下的行组元数据, 根据每个列的查询条件, 得到其结果集, 并判断是否与行组对应列上的范围值产生重叠, 若其交集为空集, 则过滤掉这个行组, 否则, 将该行组读入到内存 (行 9-18).

算法 1　列存储数据扫描算法

输入: 关系表 T, 投影属性列集合 $\{a_1, a_2, \cdots, a_i\}$, 查询条件 c

输出: 查询结果集

```
1: 通过文件索引 file_index 中定位属性列集合所在的存储节点和 parquet_file;
2: 读入 parquet_file的行组元数据;
3:  /*若查询条件含有主键, 则使用二分法过滤部分行组*/
4:  if 查询条件含有主键 then
5:      scan_row_groups为二分法过滤后的行组集合;
6:  else
7:      scan_row_groups为parquet_file的全部行组;
8:  end if
9:  for all scan_row_groups do
10:     读入行组元数据对应的列值范围 [min, max];
11:     在非主键列的查询条件下对行组进行查询;
12:     if 得到的结果集与行组对应列值范围交集为空 then
13:         过滤该行组;
14:         continue;
15:     else
16:         将行组中投影需要的列块读入内存;
17:     end if
18:  end for
```

算法 1 是 Cedar 列存储机制性能优化的重点, 其算法效率对查询性能具有巨大的影响. 因此, 本文对该算法进行时间复杂度分析. 假设关系表 T 有 N 个记录和 p 个属性列, 每个行组的记录数目为 k, 投影列的数目为 l, 在非主键列的查询条件下对行组的选择率为 s, 可知关系表 T 被切分成 $\frac{N}{k}$ 个行组, 非主键列的查询条件下对行组进行过滤, 剩下 $\frac{N}{k} \cdot s$ 个行组, 该算法的时间复杂度为 $O(\frac{l}{p} \cdot s \cdot N)$. 而现有行存储的数据扫描算法, 只对含有主键的查询条件使用二分查找法过滤掉一部分记录, 不使用除主键以外的列查询条件进行过滤, 其时间复杂度为 $O(N)$. 因此, 算法 1 的性能比现有的算法较好, 特别是在投影列的数目和行组的选择率越低时, 性能优势就越显著.

3.3　列存储的数据批量更新算法

在合并阶段, Cedar 需要将 CS 的旧版本列存储数据与 UPS 的增量数据进行融合, 生成新版本列存储数据. 因此, 本文提出了一种数据批量更新算法, 负责将 Cedar 的数据转换成 Parquet, 具体见算法 2 所示. 算法 2 中, 数据表中每一个属性列由一个列数据写入器 column_writer 负责, 每个列数据写入器拥有各自的数据块, 存放着数据缓存. 在算法开始时会依次初始化列存储的数据文件 parquet_file 和各列对应的列数据写入器; 然后不断地从基线数据存取器获取需要

写入的记录, 遍历每一条记录的数据项, 各列数据写入器将对应的数据项进行RLE编码写入其数据块缓存中, 当这个数据块缓存达到阈值时, 将数据写到列块 column_chunk 中; 最后当前处理的记录达到 parquet_file 的行组限制时, 将各个列块 column_chunk 进行 Snappy 格式压缩后写入parquet_file, 若当前 parquet_file 大小达到阈值, 则创建新的 parquet_file 进行存储.

算法 2 列存储数据批量更新算法

输入: 关系表 T 的所有数据记录 $\{r_1, r_2, \cdots, r_n\}$

输出: 列存储数据文件

```
1: 初始化 parquet_file;
2: for all 关系表 T 中的属性列 {a_1, a_2, ..., a_i} do
3:     初始化该属性列 a_i 对应的数据写入器 colum_writer_i;
4: end for
5: while 基线数据存取器获取一个需要写入的记录 r_i ∈ {r_1, r_2, ..., r_n} do
6:     /*将行组写入文件*/
7:     if 当前处理的记录数超过 parquet_file 的行组限制 then
8:       if 当前数据文件大于阈值 then
9:           创建新的 parquet_file;
10:        end if
11:        for all i ∈ {1, 2, ..., p} do
12:            将列块 colum_chunk_i 进行 Snappy 格式压缩后写入 parquet_file;
13:        end for
14:    end if
15:    /*数据写入器将记录的每个列数据迭代填入各自数据块缓存*/
16:    for all 属性列的数据项 do
17:        if 当前该列的数据块缓存大于阈值 then
18:            将该列数据块缓存中的列数据写入列块 colum_chunk_i;
19:            创建该列新的数据块缓存;
20:        end if
21:        对数据项进行RLE编码后写入该列对应的数据块缓存;
22:    end for
23: end while
```

算法 2 的效率决定着 Cedar 列存储机制数据持久化的性能, 故对其进行时间复杂度分析. 假设关系表 T 有 N 个记录和 p 个属性列, 由于 Cedar 上层查询引擎以行存储格式处理数据, 持久化成列存储格式时需要额外的转换, 故该算法需要将每条记录的每个列数据迭代地填入各自的对应数据块缓存, 其时间复杂度为 $O(p \cdot N)$. 而在现有行存储的数据批量更新算法中, 只需以整条记录为粒度处理并填入块缓存, 无需遍历每条记录的每个列数据, 时间复杂度为 $O(N)$. 因此, 算法 2 的复杂度比行存储的较高, 当关系表的属性列越多时, 算法 2 的劣势越大.

4 实 验

4.1 实验环境

本文在开源的分布式数据库Cedar上实现了列存储机制. 实验服务器配置信息如下: Inter(R) Xeon(R) E5-2680 CPU, 168 G内存, 千兆以太网, CentOS release 6.5(Final) 64位操作系统, 在一台服务器上 RootServer(RS)、UpdateServer(UPS)、MergeServer(MS), 以及一个 ChunkServer(CS) 各部署一个.

4.2 单表聚合函数性能测试

实验目的: 测试单表情况下列存储机制对系统的聚合函数性能影响.

实验设置: 本测试使用某企业具有128列属性的核心业务表, 并生成50万行、100万行、200万行、500万行及1 000万行数据集, 分别在原Cedar版本和引入列存储机制后的新版本上进行测试. SQL语句包含OLAP应用常见的聚合操作, 例如COUNT、SUM等. 实验结果如图4所示.

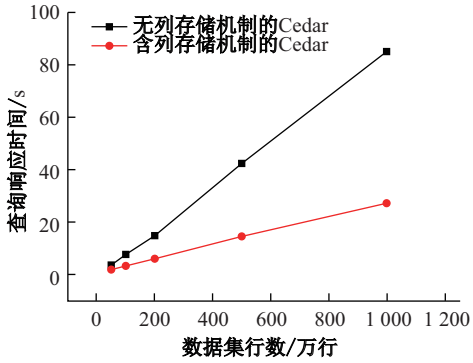

图4　聚合函数性能

Fig. 4　The performance of the aggregate functions

图4描述的是Cedar引入列存储机制前后的聚合函数性能对比, 横坐标表示数据集大小, 纵坐标表示查询响应时间. 可观察到数据集较小的时候, 两者的查询响应时间差距较小, 随着数据集规模的扩大, 两者的响应时间都呈现增长趋势, 但引入列存储机制后的增长率低于原Cedar的增长率; 当数据集达到1 000万行时, 原Cedar的查询响应时间是引入列存储机制后的3.11倍. 由于现实业务表中数据的重复项较多, 这造成了数据重复率会随着数据集规模的增大而提高, 压缩的效果也越显著, 降低了I/O开销. Cedar在引入列存储机制后的查询性能得到提升, 且随着数据集的增大, 性能的提升越显著, 展现出其良好的扩展性.

4.3　事务处理性能测试

实验目的: 测试列存储机制对系统的事务处理性能影响.

实验设置: 本测试使用具有50万行、128列属性的数据表, 不断增加连接并发数, 分别在原Cedar版本和引入列存储机制后的新版本上进行测试. 测试工具为sysbench. 事务包含的语句有插入一条新记录的操作、一次4个主键列的等值查询. 对18个非主键列进行一次更新操作. 结果如图5所示.

图5展现了Cedar引入列存储机制前后的事务处理性能的变化, 横坐标表示Cedar连接并发线程数目, 纵坐标表示Cedar每秒处理的事务数量(Transaction Per Second, TPS). 可观察到随着并发线程数目的增多, 两者的TPS都呈现增长趋势, 但差距并不明显. 可见Cedar在引入列存储机制后对事务处理性能的影响较小, 这是因为Cedar中数据的更新不会立即持久化到磁盘上, 节省了I/O的开销. Cedar列存储机制的性能损失一直维持在10%以内, 体现出列存储机制具有低侵入性. 由于Cedar行存储机制采用了块索引、块缓存、行缓存的三级缓存设计, 对查询性能进行了优化处理. 相比之下, 列存储机制并没有设计缓存, 其在这个查询上的性能处于劣势, 最终造成了列存储机制的TPS略低于行存储机制的结果. 因此, 针对列存储机制的缓存设计将是进一步深入优化的方向之一.

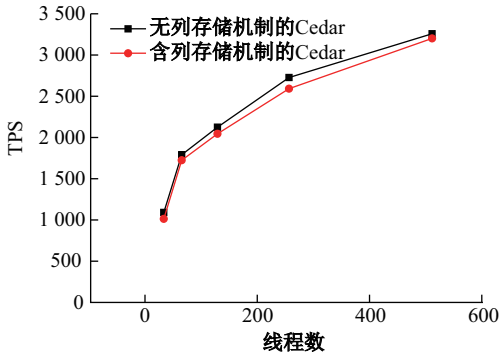

图 5 事务处理性能

Fig. 5 Comparison transaction processing performance

4.4 查询涉及的列数对性能的影响

实验目的: 测试单表情况下查询涉及的列数对系统的查询性能的影响.

实验设置: 本测试使用具有 50 万行、128 列属性并且各字段长度相等的数据表, 分别在原版本和引入列存储机制后的新版本上进行测试. SQL 语句包含有 COUNT、SUM 等聚合函数. 查询语句涉及的列数分别为 1 列、2 列、4 列、8 列、16 列、32 列及 64 列. 考虑到原版本缓存机制会缓存规模较小的数据, 而 OLAP 负载涉及的数据规模特别大, 因此本文消除了缓存带来的影响, 使测试结果更具意义. 结果如图 6 所示.

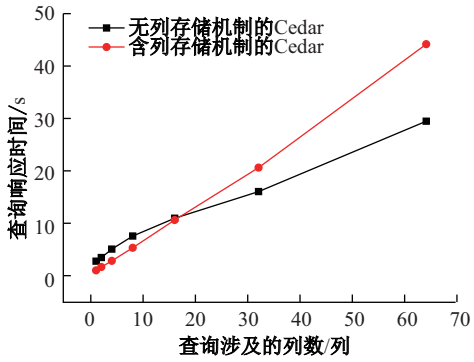

图 6 查询涉及的列数对性能的影响

Fig. 6 The performance impact of the number of columns in a query

图 6 描述的是查询涉及的列数对系统的查询性能的影响, 横坐标表示查询涉及的列数, 纵坐标表示查询响应时间. 可看观察到查询涉及的列数较少时, 引入列存储机制后的性能要优于引入前的性能; 随查询涉及的列数的增多, 两者的响应时间都呈现增长趋势, 但列存储机制的查询响应时间增长率明显高于原 Cedar 的增长率. 在查询涉及列数达到 16 列以后, 引入列存储机制后的性能要劣于引入前的性能; 当查询涉及列数达到 64 列时, 引入列存储机制后的查询响应时间是原 Cedar 的 1.29 倍. 这验证了本文第 2.3 节中的结论: 查询涉及的列数是影响两种存储方式数据访问代价差异的主要因素之一; 当查询涉及的列数越多时, 列存储机制的劣势越突出. 因此, 可以总结该机制适合于查询涉及的列数较少的场景中.

4.5 数据重复率对查询性能的影响

实验目的: 测试单表情况下数据重复率对 Cedar 的查询性能影响.

实验设置: 本测试使用 10 张具有 50 万行、128 列属性并且各字段长度相等的数据表, 其各

列平均数据重复率分别为0%、25%、50%、99%、99.9%. 在原版本和引入列存储机制后的新版本上进行测试. SQL 语句包含有在 2 个非主键列上做全表的 COUNT、SUM 的聚合函数操作. 基于同样的原因, 测试消除了原版本缓冲机制的影响. 结果如图 7 所示.

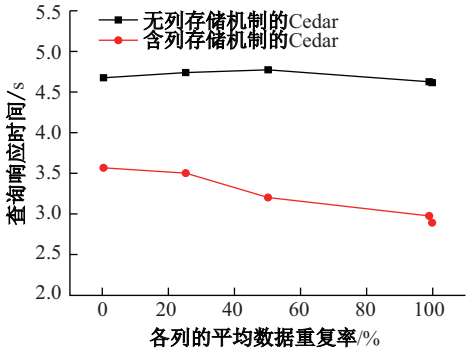

图 7　数据重复率对查询性能的影响

Fig. 7　The effect of data repetition on performance

图 7 描述的是数据重复率对系统的查询性能的影响, 横坐标表示数据表各列的平均数据重复率, 纵坐标表示查询响应时间. 可观察到随数据重复率的增长, 原行存储机制的查询响应时间几乎无变化, 而列存储机制的查询响应时间不断地降低, 两者响应时间的差距逐渐增大; 当数据重复率达到 99.9% 时, 原行存储机制的查询响应时间是列存储机制的 1.59 倍. 这验证了本文 2.3 节的结论: 数据重复率是影响两种存储方式数据访问代价差异的主要因素之一; 随着数据重复率提高, 列存储机制的性能优势越大. 因此, 该机制更适合于数据重复率较高的场景中.

5　结　　语

随着各行各业数据规模和分析需求的增长, 数据库面向 OLAP 应用的查询性能越来越受到重视. 本文基于读写分离架构的分布式数据库 Cedar, 设计了一种列存储机制, 分析了其适用场景, 并在 Cedar 中实现了该机制. 通过实验证明, 本文所设计的列存储机制能够有效地提升 Cedar 的分析处理性能, 并且对事务处理的性能影响极低; 在查询请求涉及列数较少、各列中数据重复率较高的场景下, 该机制对查询性能提升的效果越显著. 但本文的列存储机制仍有更深入优化的空间, 如缓存、延迟物化、基于代价的查询执行策略及面向新硬件的查询优化将会成为未来研究的重点.

[参　考　文　献]

[1] CODD E F, CODD S B, SALLEY C T. Providing OLAP (On-Line Analytical Processing) to User-Analysts: An IT mandate [J]. Codd and Date, 1993, 32: 3-5.

[2] 华东师范大学. 面向大型银行应用的高通量可伸缩分布式数据库系统 Cedar[DB/OL]. [2018-05-16]. https://github.com/daseECNU/Cedar.

[3] COPELAND G P, KHOSHAFIAN S N. A decomposition storage model [C]// ACM SIGMOD International Conference on Management of Data. New York: ACM, 1985: 268-279.

[4] ABADI D, MADDEN S, HACHEM N, Column-stores vs. row-stores: How different are they really?//Proceedings of the 2008 ACM SIGMOD international conference on Management of data. New York: ACM, 2008: 967-980.

[5] RAMAN V, ATTALURI G, BARBER R, et al. DB2 with BLU acceleration: So much more than just a column store [J]. Proceedings of the VLDB Endowment, 2013, 11: 1080-1091.

[6] PETRAKI E, IDREOS S, MANEGOLD S. Holistic Indexing in main-memory column-stores [C]// ACM SIGMOD International Conference on Management of Data. New York: ACM, 2015: 1153-1166.

[7] LANG H, FUNKE F, BONCZ P A, et al. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation [C]// International Conference on Management of Data. New York: ACM, 2016: 311-326.
[8] RAMNARAYAN J, MOZAFARI B, WALE S, et al. SnappyData:A hybrid transactional analytical store built on spark [C]// International Conference on Management of Data. New York: ACM, 2016: 2153-2156.
[9] LEE J, HAN W S, NA H J, et al. Parallel replication across formats for scaling out mixed OLTP/OLAP workloads in main-memory databases [J]. The VLDB Journal, 2018, 27(3): 421-444.
[10] SQream. SQream SQream DB [DB/OL]. [2018-06-16]. https://sqream.com/.
[11] ROOT C, MOSTAK T. MapD: A GPU-powered big data analytics and visualization platform [C]// Proceeding of the SIGGRAPH '16 ACM SIGGRAPH 2016 Talks. New York: ACM, 2016: Article No 73. DOI:10.1145/2897839.2927468.
[12] 阳振坤. OceanBase 关系数据库架构 [J]. 华东师范大学学报 (自然科学版), 2014(5): 141-148, 163.
[13] 黄贵, 庄明强. OceanBase 分布式存储引擎 [J]. 华东师范大学学报 (自然科学版), 2014(5): 164-172.
[14] GOOGLE. Google Snappy [DB/OL]. [2018-06-16]. https://github.com/google/snappy.
[15] SALOMON D. Data Compression: The Complete Reference [M]. New York: Springer-Verlag Inc, 2000.
[16] APACHE. Apache parquet [DB/OL]. [2018-06-16]. http://parquet.apache.org/.

(责任编辑: 李 艺)

(上接第 40 页)

[70] KIM H J, LEE Y S, KIM J S. NVMeDirect: A User-space I/O framework for application-specific optimization on NVMe SSDs[C]// Proceedings of HotStorage. 2016.
[71] BJØRLING M, GONZÁLEZ J, BONNET P. LightNVM: The linux open-channel SSD subsystem[C]// Proceedings of FAST'17. USENIX Association. 2017: 359-374.
[72] CAULFIELD A M, MOLLOV T I, EISNER L A, et al. Providing safe, user space access to fast, solid state disks[J]. ACM SIGARCH Computer Architecture News, 2012, 40(1): 387-400.
[73] YANG J, MINTURN D B, HADY F. When poll is better than interrupt[C]// Proceedings of FAST'12. USENIX Association. 2012.
[74] LI C, DING C, SHEN K. Quantifying the cost of context switch[C]// Proceedings of the 2007 Workshop on Experimental Computer Science. ACM, 2007: 2.
[75] SHIN W, CHEN Q, OH M, et al. OS I/O path optimizations for flash solid-state drives[C]// USENIX Annual Technical Conference. 2014: 483-488.
[76] YU Y J, SHIN D I, SHIN W, et al. Optimizing the block I/O subsystem for fast storage devices[J]. ACM Transactions on Computer Systems (TOCS), 2014, 32(2): 6.
[77] XU J, SWANSON S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories [C]// Proceedings of FAST'16. USENIX Association. 2016: 323-338.
[78] VOLOS H, NALLI S, PANNEERSELVAM S, et al. Aerie: Flexible file-system interfaces to storage-class memory[C]//Proceedings of the Ninth European Conference on Computer Systems. ACM, 2014: 14.
[79] KANNAN S, ARPACI-DUSSEAU A C, ARPACI-DUSSEAU R H, et al. Designing a true direct-access file system with DevFS[C]//Proceedings of the 16th USENIX Conference on File and Storage Technologies. 2018: 241-256.

(责任编辑: 林 磊)