基于 Chunk Folding 的多租户数据库缓存管理机制

姚金成 张世栋 史玉良 李庆忠
(山东大学计算机科学与技术学院 济南 250101)

摘 要 Chunk Folding 是 SaaS 模式下常用的存储架构之一,通过共享数据库共享架构来存储租户的数据以获取规模经济效益,但基于传统数据库搭建的 Chunk Folding 共享存储架构,其缓存管理机制缺乏良好的多租户特性,导致数据库性能恶化,租户的 SLA 得不到保障.为此,提出了基于 Chunk Folding 的自适应多租户缓存管理机制,该机制以租户的 SLA 需求作为驱动,依据租户当前访问模式,动态生成缓存单元集并计算缓存单元集的 I/O 效益,通过贪婪算法来选择缓存单元集,使得租户 SLA 得到满足的同时最小化缓存资源的消耗.通过实验分析证明了该缓存管理机制的有效性.

关键词 共享存储架构;多租户;缓存管理;SLA
中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2011.02319

Multi-Tenant Database Memory Management Mechanism Based on Chunk Folding

YAO Jin-Cheng ZHANG Shi-Dong SHI Yu-Liang LI Qing-Zhong
(*School of Computer Science and Technology, Shandong University, Jinan* 250101)

Abstract Chunk Folding is one of the common storage architectures in SaaS, which employs shared databases and shared architectures to store tenants' data, so as to gain benefits of economies of scale. However, Chunk Folding based on traditional databases, in lack of multi-tenant properties in memory management, leads to performance degradation and consequently violates tenants' SLA. We propose a Self-Adaptive Multi-Tenant Memory Management (SAMTMM) to consistently achieve tenant's SLA requirement while the memory consumption is minimized, which dynamically generates a series of cache replacement units according to the current access model and computes the corresponding I/O yield, and then adopts a greedy algorithm to select the corresponding replacement units for each tenant. The effectiveness of our multi-tenant memory management is evaluated by our experiments.

Keywords shared storage architecture; multi-tenant; memory management; SLA

1 引 言

软件即服务(Software as a Service,SaaS)是一种基于互联网的服务提供模式、企业或个人租赁服务,并通过 Web 浏览器或 Web 客户端访问这个服务.在 SaaS 模式下,数据存储通常采用共享数据库共享存储架构[1],从而获取规模经济效益,如 Universal Table[2-3]、Pivot Table、Chunk Folding[4]等.这些共享存储架构各有其优缺点,Chunk Folding 相比于

收稿日期:2011-08-05;最终修改稿收到日期:2011-10-31.本课题得到国家科技支撑计划(2009BAH44B02)、国家自然科学基金(90818001)、山东省自然科学基金(2009ZRB019YT,ZR2010FQ026)、山东省科技攻关计划(2010GGX10105)资助.**姚金成**,男,1986 年生,硕士研究生,主要研究方向为数据库、云计算.**张世栋**(通信作者),男,1969 年生,博士,教授,主要研究领域为数据库、Web 数据集成、云计算.E-mail: zsd@sdu.edu.cn.**史玉良**,男,1978 年生,博士,副教授,主要研究方向为服务计算、云计算、数据库.**李庆忠**,男,1965 年生,博士,教授,主要研究领域为大规模网络数据管理及 Web 数据集成.

Universal Table 无存储空间的大量浪费,可更好地支持索引,查询时不需要处理大量的 null 值以及数据类型的转换;Chunk Folding 相比于 Pivot Table 减少了存储元数据和业务数据的比例,也降低了重构租户逻辑表的代价.

尽管 Chunk Folding 对比其它共享架构具有明显的优势,但基于传统数据库搭建的共享存储架构,仍然存在着很多的不足,如额外的 I/O 和自然连接[4],而文献[4]指出 I/O 是主要因素.因此,如何缓解 I/O 导致的性能下降就成为改善 Chunk Folding 共享存储架构性能的关键;对于 I/O 的优化,传统数据库通常采用缓存数据块的方式,但是在多租户共享存储架构下,传统数据库的缓存管理机制存在以下两点不足之处:

(1)传统数据库缓存机制以数据块作为缓存单元,而多租户共享存储架构下,任一数据块均包含了大量其他租户的无关数据,采用数据块作为缓存单元导致大量缓存资源的浪费.

(2)传统数据库缓存机制缺乏多租户的概念,对于来自租户的请求,传统缓存机制会从提高数据库整体性能的角度进行缓存管理,这就会导致租户间资源分配的极为不合理,如高频访问租户抢占低频访问租户的资源,使得低频访问租户的 SLA 响应时间需求得不到保障.

针对以上问题,本文提出了一种基于 Chunk Folding 的自适应的多租户缓存机制(Self-Adaptive Multi-Tenant Memory Management,SAMTMM).对每一个租户,SAMTMM 适应当前的租户访问模式,动态生成合理的候选缓存单元,并计算其对应的 I/O 效益,然后具备最高效益率(缓存单元的 I/O 效益与缓存单元占据内存大小的比值)的缓存单元被选中放入缓存,重复上述过程直到满足租户的 SLA 响应时间需求.本文将上述过程形式化定义为变种的背包问题,并采用贪婪算法来选择缓存数据集.

本文第 2 节介绍相关工作;第 3 节基于 Chunk Folding 共享存储架构的特性,给出相应的代价估计模型;第 4 节给出多租户的缓存机制;第 5 节是相应的实验环境和实验结果;最后是本文的总结.

2 相关工作

数据库缓存作为改善数据库性能的主要因素,已经被广泛的研究[5-7].文献[5]给出了一种自调节的缓存管理机制 STMM,通过为不同缓存使用者(如排序缓存、Hash 连接缓存、锁缓存、缓冲池等)建立缓存消耗和时间产出效益的模型,从而使得不同使用者之间有了统一的衡量标准,并将缓存分配给产出效益大的使用者,以达到提高系统整体性能的目的;与 STMM 类似,文献[6]通过对每一缓存对象定义 BYHR,即单位字节命中产出率,选择缓存那些高产出的缓存对象从而利用有限的缓存资源使得网络传输代价大幅度减少.文献[7]提出了 Fragment fencing 用于解决具有明确 QoS 需求的不同类型请求的缓存分配问题,以达到最小化缓存消耗的目的.该方案假设数据库磁盘块有统一的访问频率,并基于简化的事务模型:(1)事务响应时间直接与 I/O 数目相关;(2)命中率与缓存的磁盘块的数目成正比.如果某一类请求的响应时间不满足 QoS 所指定的响应时间,则通过当前已分配的缓存、当前响应时间与 QoS 响应时间之间的线性对应关系就可以计算出为满足 QoS 响应时间所需分配的缓存大小.但是在多租户共享存储模式下,不同租户的数据共享存储在某一磁盘块中,这就导致单纯以磁盘块数目而不考虑磁盘块的数据内容作为衡量缓存命中率指标的假设存在着不合理性.

随着云计算的兴起,传统数据库的缓存管理机制存在着不足之处[8].首先,租户间对共享缓存资源的竞争会导致每一租户获取的缓存资源不足以缓存其数据,从而导致低的缓存命中率和不好的用户体验.其次,对于缺乏多租户特性的共享缓存管理策略(LFU、LRU)将会倾向于将缓存分配给具备更高请求速率的租户,从而导致低请求速率租户的缓存命中率极低.据此,文献[8]指出在云计算环境下将缓存作为云服务的重要性并讨论了云缓存服务的系统原型 Blaze,Blaze 迭代地将缓存分配给具有最大收益的租户并采用基于 CLOCK 的多租户缓存替换策略,保证租户的 SLA 同时最大化系统性能.

针对共享存储模式下,传统磁盘和缓存管理策略在处理不同服务访问模式时显著的性能下降问题,Argon[9]采用预取/回写技术、缓存划分和量化磁盘时间调度 3 种机制使得每一服务达到预期的有效性指标(相对于独立运行时的效率比值),使得磁盘服务的有效性得到显著提高,其中预取/回写技术以及量化磁盘时间调度与本文的研究互补,可以很好的与本文的多租户缓存管理策略相结合.在服务间的缓存划分上,Argon 需要建立独立执行模式下缓存空间与 I/O 产出效益间的对应函数关系,其建

立过程时间较长，而本文的 SAMTMM 不需要建立此函数关系，因此没有准备阶段的资源消耗，可以更好地适应在线调整；此外，Argon 关注的是服务间的存储共享，而本文关注的是服务内租户间的存储共享，共享粒度的不同导致磁盘数据组织形式的不同，因此 Argon 并不直接适用于 SaaS 模式下的多租户数据库缓存管理.

与 Argon 不同，文献[10]采用曲线拟合建立每一服务缓存与命中率(I/O 产出效益)之间的函数关系，动态地为每一服务分配缓存空间，满足每一服务 QoS 的同时尽可能地提高系统的整体缓存命中率；文献[11]提出了相对分化缓存服务模型，采用基于回馈的启发式缓存分配策略动态调整每一类请求所获取的缓存大小从而实现不同类别请求的缓存命中相对比. 它们均不需要 Argon 中建立缓存空间与 I/O产出效益函数的准备阶段的资源消耗，因而能够快速地适应访问模式的变化并进行在线调整. 但是与本文不同，文献[10]关注的是软 QoS，而文献[11]中缓存分配的最终优化目标是满足不同类型请求所指定的 QoS 比值，而不是满足其 QoS，这与本文多租户缓存管理机制中的 SLA 响应时间存在着需求上的差异.

在缓存替换策略方面，常用的缓存替换算法有 LRU、LFU 和 LRU-K[12] 等. 这些算法通常以页面作为替换单元，并采用引用流作为单一的评价页面缓存的标准，从而最小化页面缺失率；其中最为常用的为 LRU，因为它易于实现并且时间复杂度不高，但是其存在许多不足，如读取一系列的不经常使用的页面，则经常被引用的页面就会从缓存中驱逐出去，这就导致了缓冲池污染. 针对这一问题，许多基于 LRU 改进的算法被提出，如 LRU-K 算法，该算法跟踪最后一次对页面进行 K 引用的时间，并按该信息对页面进行排序；当需要读进新的页面时，则根据缓存页面的等级剔除等级低的页面从而防止缓存池被污染；与 LRU-K 极为类似，MySQL/InnoDB 采用带有中点策略的 LRU 替换策略[13]，该策略的 LRU 缓冲池链表中包括新链表和旧链表，对每一个读进缓存的新页面，他们均被放在新链表中，当且仅当规定的时间内该页面再次被访问，那么该数据块才会被放入旧链表中；目前这些成熟的缓存替换算法均采用页面作为缓存单元，但是未考虑 SaaS 共享存储架构下以页面作为替换单元存在着缓存浪费，而如何给出适应共享架构的缓存单元也并未被深入研究.

3　多租户缓存管理的理论基础

本节将介绍与多租户缓存管理相关的理论基础，为下一节基于 I/O 价值的动态缓存单元的多租户缓存管理机制做铺垫. 首先给出 Chunk Folding 和租户访问模式的简化模型，然后结合 Chunk Folding 共享存储架构的特性及 MySQL 的查询优化器给出其执行计划模型；最后结合该执行计划模型给出计算任意请求的 I/O 次数的公式化定义.

3.1　Chunk Folding 及租户访问模式

Chunk Folding 预定义一系列适当模式的 Chunk 表，其中预定义的 Chunk 表的每一属性列均有明确的数据类型. 对租户个性化定制后的逻辑表，Chunk Folding 采用列划分技术，将其存储到与其结构最为相近的若干 Chunk 表中. 其中，基本表的两个元数据列(Tenant，Row)和 Chunk 表的 4 个元数据列(Tenant，Table，Chunk，Row)用于还原租户的逻辑表，此外为了加速查询速度，在元数据列上建立组合索引 TR 和 TTCR.

例 1.　设有一个 CRM 管理系统，其基本表信息包括身份证和姓名，即 CRM(id，name)，设有 3 个租户其 ID 为 1、2、3，租户 1 未定制字段，租户 2 定制了邮编字段，租户 3 定制了年龄和家庭住址字段. 3 个租户的逻辑模式如图 1(a)所示，预定义的 Chunk Folding 表模式为 CRM_{base} 和 CRM_{chunk} 模式，其对应的共享存储架构如图 1(b)所示，其中灰色的属性列是元数据列. 为方便问题阐述，我们将不再区分基本表与 Chunk 表，即基本表与 Chunk 表均包含 4 个元数据列(Tenant，Table，Chunk，Row)和组合索引 TTCR.

CRM1

ID	Name
371001	张三

CRM2

ID	Name	Postcode
371098	王五	250101

CRM3

ID	Name	Age	Adress
372101	李四	36	济南
372122	杨六	25	威海

(a) Logical View

CRMbase

Tenant	Row	ID	Name
1	1	371001	张三
2	1	371098	王五
3	1	372101	李四
3	2	372122	杨六

CRMchunk

Tenant	Table	Chunk	Row	int1	str1
2	1	0	1	250101	#
3	2	0	1	36	济南
3	2	0	2	25	威海

(b) Chunk Folding

图 1　租户逻辑模式与 Chunk Folding 映射关系

为方便我们接下来的问题描述，我们将给出 Chunk Folding 的形式化定义. 设逻辑表 R，其属性集 $A=\{A_1,A_2,\cdots,A_n\}$，$C=\{C_1,C_2,\cdots,C_m\}$ 是采用 Chunk Folding 共享存储架构后台预定义的 Chunk

表,则逻辑表 R 与后台 Chunk Folding 的对应模式映射信息定义如下.

定义 1. 映射关系. 逻辑到物理 $\alpha: A_i \rightarrow (C_j, k)$. 表示关系 R 的属性 A_i 位于后台 C_j 块内,且为其分配的全局唯一编号为 k. 物理到逻辑 $\beta: (C_j, k) \rightarrow \{A_i, \cdots, A_{i+p}\}$ 用于获取编号为 k 的数据块 C_j 所包含的属性集合.

服务运营商可以跟踪一段时间内租户发起的数据库访问来获取租户访问模式,我们将其建模为一个有序对 (q, w_q), $q=1, \cdots, Q$. 其中 q 代表第 q 个请求,w_q 是第 q 个请求的频率. 需要注意的是这里的 Q 个请求都是非事务型的,但 SaaS 是面向事务型的应用,所以会包括事务型的请求. 对于事务型请求我们采用重写逻辑[4],将其分为两个阶段:(a) 查询阶段. 收集所有满足更新条件的行号(即元数据列,Tenant,Table,Chunk,Row). (b) 更新阶段. 更新满足更新条件的行所对应的更新列. 由于数据库缓存机制只会减少读操作的 I/O 次数,而并不能减少写操作的 I/O 次数(可通过延迟写,日志记录),所以本文将只考虑事务型请求的阶段(a)的 I/O 代价,而通过减少阶段(a)的 I/O 代价也可以显著提高事务型请求的处理效率. 简单起见,本文剩下的部分所有的请求集除非特别说明,将都是非事务型的.

3.2 执行计划

为执行一个请求,查询优化器需要生成所有的执行计划,然后选择代价最小的执行表访问. 主要有两种表扫描方式:顺序扫描和索引扫描. 索引扫描不需要扫描整个表,而只需要访问满足条件的元组所属的磁盘块即可. 使用索引扫描需要满足两个条件:(1) 在对应的查询中有索引可用,(2) 满足条件的元组的数目远小于表中所有元组的数目[14-15].

本文的代价估计是基于嵌套索引的左深树执行计划来进行计算的,理由如下:

(1) Chunk Folding 共享架构下的所有请求都需要被重写添加上元数据列[4],而元数据列上均建立了组合索引 TTCR,因此有索引可用.

(2) Chunk Folding 共享架构下 Chunk 表中存储了成千上万租户的数据,对于特定租户特定表的查询,满足条件的元组的数目相对于 Chunk 表中的元组的总数目而言很少.

(3) MySQL 的查询优化器的执行计划都是基于左深树的代价估计[16],对于其它数据库的代价估计则需要配合与对应的查询优化器相结合从而得出相应的 I/O 代价估计.

(4) 本文关注的重点是缓存管理机制,所以我们尽量避免由执行计划选择上的不确定性因素导致的缓存单元 I/O 代价估计的不准确. 因此,本文中的所有请求均事先经过 MySQL 的 analyze/explain 命令进行执行计划的验证,从而保证了代价估计的准确性,其验证过程如图 2 所示.

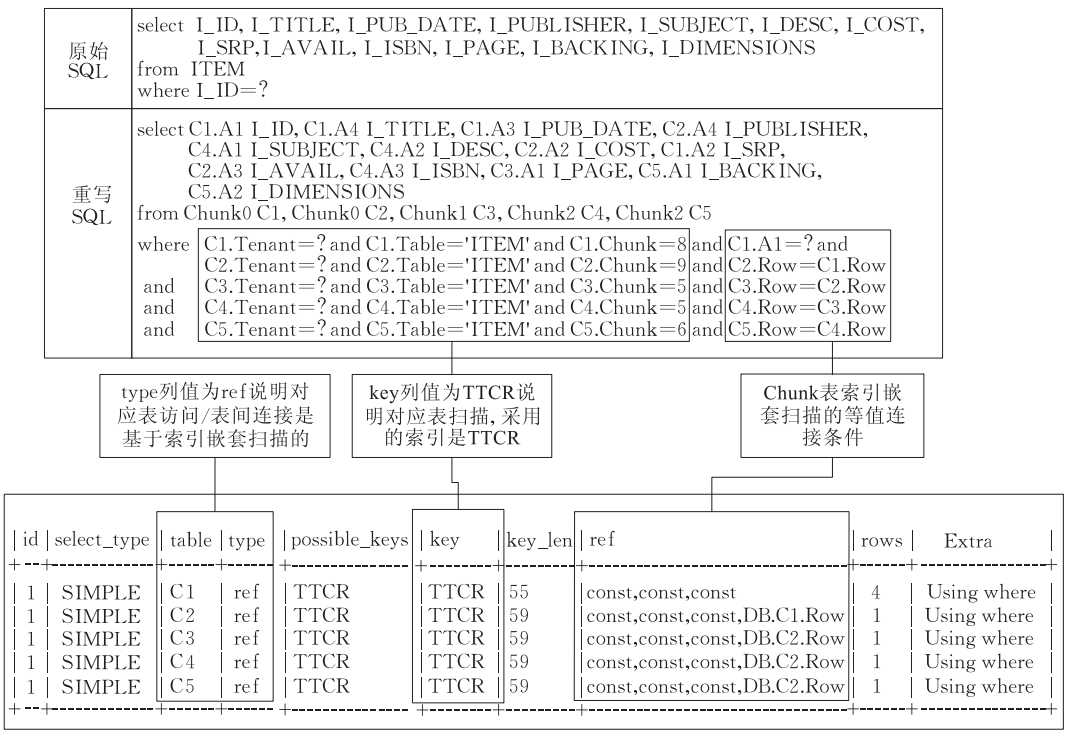


图 2 MySQL 基于嵌套索引的左深树执行计划验证

图 2 中我们以典型的 TPC-W 中的查看产品信息请求作为示例，其中的逻辑模式与后台 Chunk Folding 物理模式的映射关系见 5.2 节. 从图 2 中 MySQL 查询优化器的执行过程我们可以得出，Chunk 表的访问以及表间连接是基于索引的(对应 table，type 列)，而且每一 Chunk 表的访问均采用 TTCR 索引 (对应 key 列)，并利用过滤后的 TTCR 作为访问下一 Chunk 表的条件(对应 ref 列)，依次类推，该过程是一个典型的基于嵌套索引扫描的左深树执行计划. 为进一步验证对执行计划假设的正确性，我们分别测试了相同租户数目不同规模(租户)数据集下查询优化器的执行过程，结果显示与图 2 类似.

图 3 给出了基于嵌套索引扫描的左深树的 MySQL 执行计划图，该执行计划共涉及到 3 个 Chunk 表. 图中 IXSCAN 表示 Chunk 表的访问采用索引扫描的方式，TTCR 代表使用的索引 (Tenant，Table，Chunk，Row)，Fetch 操作用于访问 Chunk 表从而取得对应元组的相关的属性列，NLJOIN 表示采用嵌套索引扫描的方式. 对于每一个请求，本文中我们主要考虑 I/O 代价，也即每一个虚线框内的 I/O 代价之和，其中一个虚线框代表的是访问一个 Chunk 的代价.

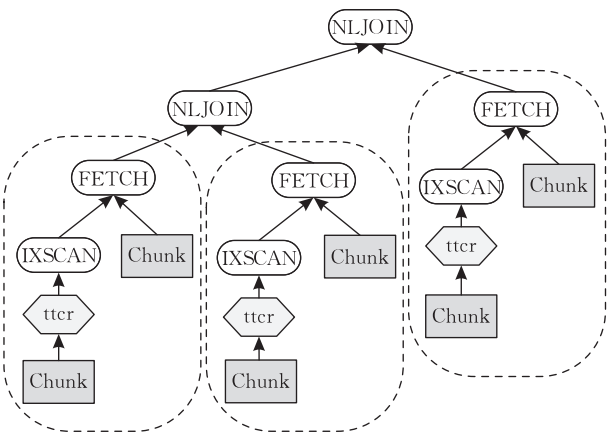


图 3　MySQL 基于嵌套索引的左深树执行计划示意图

3.3　I/O 代价估计

从图 3 所示的执行计划示意图我们可以得出，对于一个请求，其 I/O 代价为所有虚线框内的 FETCH 操作磁盘 I/O 次数与每一次 I/O 花费时间乘积之和；对于索引扫描，FETCH 操作的 I/O 次数与这个数据块被访问时满足条件的元组数目关系很密切，我们设第 i 个 Chunk 被扫描时，符合条件的元组的数目为 k_i，基于文献[14-15]给出的公式有第 i 个 Chunk 被扫描时需要的 I/O 次数即为

$$E(IO_i)=m_i\times\left(1-\frac{C_{n_i d}^{k_i}}{C_{n_i}^{k_i}}\right),\ d=1-1/m_i \quad (1)$$

在函数(1)中，m_i 表示第 i 个 Chunk 占据的磁盘空间对应的数据库逻辑页面的数目，n_i 表示第 i 个 Chunk 中的元组的数目. 对于一个请求，如果没有任何数据被缓存，则其需要的磁盘 I/O 次数为所有 Chunk 的 I/O 次数之和，即

$$E(IO)=\sum_{i=1}^{d}E(IO_i) \quad (2)$$

其中 d 是查询中涉及到的 Chunk 的数目；如果某一 FETCH 操作所涉及的属性集刚好全在缓存中，则对应的 FETCH 操作的 I/O 次数为 0.

通过式(1)和(2)我们可以计算出任意一个请求所需的 I/O 次数，接下来我们将给出如何计算每一次 I/O 的时间. 本文将借鉴文献[17]给出的在随机请求情况下的磁盘驱动模型来计算一次 I/O 的时间，其对应的所需要的设备参数和数据库参数如表 1 所示. 采用随机请求下的 I/O 时间估计是由于在共享存储模式下，租户间的数据相互交叉存储，因此每一租户的数据在磁盘上的分布是分散的，查询中所涉及的元组在磁盘上的位置呈现出随机分布的状况.

表 1　计算一次 I/O 平均时间所需的设备参数

参数	符号	参数	符号
单面磁盘磁道数	L	磁道 i 的扇区数	C_i
单盘面的扇区数	M	磁盘转速	R
短寻道磁道上界	Q	扇区大小	S
DB 数据块大小	P		

一次 I/O 所需要的时间主要包括寻道时间、旋转等待和传输时间 3 个部分，因此一次 I/O 的平均时间等于平均寻道时间、平均旋转等待和平均传输时间之和，即

$$\overline{Cost}_{\text{disk}}=\overline{Cost}_{\text{seek}}+\overline{Delay}_{\text{rot}}+\overline{Cost}_{\text{transfer}} \quad (3)$$

寻道时间与寻道长度 d 有关，其对应的计算公式如下：

$$Cost_{\text{seek}}(d)=\begin{cases}a_1\sqrt{d}+b_1, & d\le Q\\ a_2 d+b_2, & \text{其它}\end{cases} \quad (4)$$

式(4)给出了寻道距离为 d 的寻道时间，平均寻道时间则假设磁头以相同的概率位于每一磁道上，也即 $1/L$，那么以任意磁道作为起点的平均寻道时间之和为平均寻道时间. 因此，

$$\overline{Cost}_{\text{seek}}=\sum_{l=1}^{L}(\text{起点在 } l \text{ 时平均寻道时间})\times \text{起点在 } l \text{ 的概率}$$

$$=\sum_{l=1}^{L}\sum_{d=1}^{L-1}(\text{寻道长度为 } d \text{ 的概率}\times \text{寻道长度为 } d \text{ 的代价})\times\frac{1}{L}$$

$$=\sum_{l=1}^{L}\sum_{d=1}^{L-1}\left(\frac{\text{与 } l \text{ 距离为 } d \text{ 的磁道扇区数之和}}{\text{单个盘面的扇区数目}}\times Cost_{\text{seek}}(d)\right)\times\frac{1}{L}$$

$$=\frac{1}{L}\times\sum_{l=1}^{L}\sum_{d=1}^{L-1}\left(\frac{C_{l+d}+C_{l-d}}{M}\times Cost_{\text{seek}}(d)\right) \quad (5)$$

平均旋转等待时间是磁盘旋转半周的时间，因此，

$$\overline{Delay}_{\text{rot}}=\text{磁盘转动半圈的时间}=\frac{1}{2R} \quad (6)$$

平均传输时间等于磁道被访问的概率与对应磁道读取单个数据库数据块所需时间代价的乘积之和，因此，

$$\begin{aligned}\overline{Cost}_{\text{transfer}}&=\sum_{i=1}^{L}(\text{磁道 } i \text{ 被访问的概率}\times \text{磁道 } i \text{ 读取单个数据块的时间代价})\\&=\sum_{i=1}^{L}\left(\frac{\text{磁道 } i \text{ 的扇区数目}}{\text{单盘面的扇区数目}}\times \text{磁道 } i \text{ 读取单个数据块的时间代价}\right)\\&=\sum_{i=1}^{L}\left(\frac{C_i}{M}\times\frac{P}{S\times C_i\times R}\right)=\sum_{i=1}^{L}\frac{P}{S\times M\times R}\\&=\frac{L\times P}{S\times M\times R} \quad (7)\end{aligned}$$

4 多租户缓存管理机制

在多租户环境下，分配较多的缓存给一个租户可以减少该租户的 I/O 次数，但是留给其他租户的缓存就会不足；分配较少缓存给一个租户则会导致该租户的 I/O 次数增多，留给其他租户的缓存量会相对充裕. 在本节我们将给出适应租户访问模式的多租户缓存分配策略，该策略为每一个租户分配保障其 SLA 响应时间需求的尽量少的缓存，从而尽可能多的缓存可被留给其他租户.

4.1 基于租户访问模式的缓存单元及 *ROC* 值计算

传统数据库的缓存单元通常是一个数据库逻辑页，但是在 SaaS 多租户共享存储架构中，以数据库逻辑页作为缓存单元存在诸多的不合理性，如：(1)多个租户的数据分散存储在一个共享架构中，一个数据库逻辑页中会包含大量其他租户的无用数据，仅仅出于一个租户的性能考虑而缓存整个数据库逻辑页，导致大量缓存空间的浪费；(2)在 SaaS 中的服务请求均是基于 Web 提交的，而 Web 页面的访问遵从 Zipf 分布，即仅有 20%的请求是经常被使用的，而这些请求都是基于 HTML 模板提交的，变化的只有参数，这就说明某几个属性列被经常使用[18]，也就意味着采用元组作为缓存单元同样有其不合理性.

出于上述两点问题的考虑，本文采用列缓存代替传统数据库的页缓存；传统数据库的页缓存机制中以固定页作为缓存单元，而在我们的列缓存中，我们采用动态生成的列属性集作为缓存单元，我们称为列缓存单元. 对于一个涉及 n 个属性的请求集，则其备选列缓存单元数目为 2^n，如果考虑所有的备选列缓存单元，则代价很高. 文献[19]提出了基于访问模式驱动的缓存单元生成策略，以当前的访问模式作为输入，输出一系列缓存单元. Query Access Set (QAS)这个概念在文献[19]中被使用，它被定义为一个关系中被某个请求访问的属性的集合. 借鉴该思路，结合 Chunk Folding 特性，我们给出 QAS 的重新定义，并将 QAS 作为初始候选缓存单元从而大幅减少备选缓存单元的数量.

定义 2. Query Access Set (QAS). QAS 或者是一个查询中的谓词集(Predicate Set，PS)，或者是一个查询中所涉及的位于同一 Chunk 编号的普通属性集(Ordinary Attribute Set，OAS).

将 QAS 作为初始候选缓存单元的理由，如图 3 中所示，对于一个查询，将其谓词存放在缓存中可以过滤掉不满足条件的元数据索引项，这样在进行下一个基于索引的 FETCH 操作时需要读取的元组数目会减少，从而 I/O 次数会减少；而查询中涉及到的位于同一 Chunk 编号的属性集作为替换单元是因为对于任意一次 FETCH 操作，当且仅当该查询所涉及位于一个 Chunk 编号的属性集作为一个整体同时被缓存才会减少 I/O，而将其子集进行缓存不会达到减少 I/O 的效果.

生成初始候选缓存单元的算法如算法 1 所示.

算法 1. 生成初始候选替换单元(Generate Initial Candidate Replacement Units，GICRU).

1. IN：D，Tenant's data mapping information
2. IN：Q，Query Access Model
3. OUT：*CRU*，Generated Candidate Replacement Units
4. BEGIN
5. SET $CRU:=\varnothing$

```
6. for the i th query q_i in Q
7.    for each predicate p of query q_i
8.       CRU := QAS(q_i, α(p)) ⋃ CRU
9.    endfor
10.   for the j th chunk of tenant T's physical chunk
      layout with the identifier k
11.      CRU := {β(C_{T_j}, k) ⋂ OAS_i} ⋃ CRU
12.   endfor
13. endfor
14. return CRU
15. END
```

算法 1 以租户的访问模式信息及租户逻辑模式与物理模式之间的映射信息作为输入，输出对应初始候选缓存单元. 对访问模式中的每一个请求(对应第 6～13 行)，首先将其谓词添加到候选缓存单元 CRU 中(对应第 7～9 行)，然后对于其它普通属性集则按租户的物理模式的分块信息进行划分，将属于同一 Chunk 中的属性作为缓存单元添加到初始候选缓存单元 CRU 中(对应第 10～12 行).

为更形式化地描述我们的缓存单元生成算法，我们通过具体的例子来查看如何生成初始候选缓存单元.

例 2.　逻辑表 $R(A_1,A_2,A_3,A_4,A_5)$，逻辑模式与后台 Chunk 表的映射函数：$\alpha(A_1,A_2)=(C_1,0)$，$\alpha(A_3)=(C_2,0)$，$\alpha(A_4,A_5)=(C_3,0)$，也即$\{A_1,A_2\}$被映射到 Chunk 表 C_1，并赋予其编号 0；$\{A_3\}$被映射到 Chunk 表 C_2，并赋予其编号 0；$\{A_4,A_5\}$被映射到 Chunk 表 C_3，并赋予其编号 0. 访问模式统计信息及生成的初始候选缓存单元如表 2 所示，对于每一个访问所涉及的属性分为谓词属性集(Predicate Set，PS)和除谓词外的普通属性集(Ordinary Attribute Set，OAS).

表 2　访问模式及初始候选缓存单元集

	ID	SQL	Freq	PS	OAS
访问模式信息	1	Select A_1 from T where $A_4>?$	w_1	$\{A_4\}$	$\{A_1\}$
	2	Select A_2 from T where $A_3=?$	w_2	$\{A_3\}$	$\{A_2\}$
	3	Select A_4,A_5 from T where $A_3<?$	w_3	$\{A_3\}$	$\{A_4,A_5\}$
候选缓存单元	$QAS(q_1,(C_1,0))=\{A_1\}$, $QAS(q_1,(C_3,0))=\{A_4\}$, $QAS(q_2,(C_1,0))=\{A_2\}$, $QAS(q_3,(C_3,0))=\{A_4,A_5\}$, $QAS(q_2,(C_2,0))=QAS(q_3,(C_2,0))=\{A_3\}$				

算法 1 给出了如何计算候选缓存单元，而对于每一个缓存单元必须定义其对应的 I/O 效益率，也即放入缓存后减少的 I/O 次数和占用的缓存空间大小的比值，我们将该值定义为 ROC(Return on Consumption)，ROC 的计算需要两个参数，(1)缓存单元放入缓存后占据的缓存大小 M 和(2)减少的 I/O 次数 D，因此对缓存单元 i，有

$$ROC_i=D_i/M_i \tag{8}$$

M_i可以利用 QAS 内属性类型大小之和与元组数目的乘积来获得；D_i可以结合式(2)，利用缓存单元 i 不在缓存时所有请求的 I/O 次数减去缓存单元 i 在缓存时的所有请求的 I/O 次数：

$$D_i=\sum_{q=1}^{Q}(({}^{q}_{S}E(IO)-{}^{q}_{S\cup U_i}E(IO))\times w_q) \tag{9}$$

其中 S 表示已经贮存在缓存中的属性集，$S\cup U_i$表示将缓存单元 U_i 放入缓存；${}^{q}_{S}E(IO)$表示缓存内容为 S 时第 q 个请求的 I/O 次数.

4.2　基于贪婪算法的缓存策略

上一节给出了如何计算初始候选缓存单元及其 ROC 值，接下来我们将给出如何选择缓存单元放入缓存以及如何动态更新候选缓存单元集和对应的 ROC 值. 首先，选择 ROC 值最高的缓存单元放入缓存，并将该缓存单元从候选缓存单元集中去掉；其次，更新与刚被选中放入缓存的缓存单元有交集的那些候选缓存单元，将其交集部分去掉从而生成新的候选缓存单元；最后，计算更新后的候选缓存单元集的 ROC 值，其计算过程同样采用式(9). 重复上述过程直到满足租户的 SLA 响应时间，也即

$$\frac{\sum_{i=1}^{Q}IO(q_i)\times w_i}{\sum_{i=1}^{Q}w_i}\times\overline{Cost}_{\text{disk}}\le T_{\text{sla}} \tag{10}$$

其中 $I/O(q_i)$代表第 i 个请求的 I/O 次数，可以通过式(2)计算得出；T_{sla}代表租户的 SLA 响应时间需求.

表 3　将 $QAS(q_1,(C_3,0))=\{A_4\}$放入缓存后的候选缓存单元集

初始候选缓存单元	$QAS(q_1,(C_1,0))=\{A_1\}$, $QAS(q_1,(C_3,0))=\{A_4\}$, $QAS(q_2,(C_1,0))=\{A_2\}$, $QAS(q_3,(C_3,0))=\{A_4,A_5\}$, $QAS(q_2,(C_2,0))=QAS(q_3,(C_2,0))=\{A_3\}$
更新候选缓存单元	$QAS(q_1,(C_1,0))=\{A_1\}$, $QAS(q_2,(C_1,0))=\{A_2\}$, $QAS(q_3,(C_3,0))=\{A_5\}$, $QAS(q_2,(C_2,0))=QAS(q_3,(C_2,0))=\{A_3\}$

继续例 2 所示的例子，现在我们假设经过计算 $QAS(q_1,(C_3,0))=\{A_4\}$的 ROC 值最高，因此其被选中进入缓存；由于剩下的候选缓存单元中 $QAS(q_3,(C_3,0))=\{A_4,A_5\}$与 $QAS(q_1,(C_3,0))=\{A_4\}$的交集为$\{A_4\}$，所以不是空集，因此 $QAS(q_3,(C_3,0))=\{A_4,A_5\}$将会更新成 $QAS(q_3,(C_3,0))=\{A_5\}$，更

新后的候选缓存单元如表 3 所示.

算法 2. 最优属性集选择算法(Best Attribute Set Selecting, BASS).

1. IN: D, tenant's data statistics
2. IN: W, workload statistics
3. IN: SLA, tenant's SLA requirement
4. OUT: AS, Attribute Set should be placed in memory
5. BEGIN
6. SET $AS:=\varnothing$
7. SET CRU
8. SET ROC
9. $CRU:=GICRU(W,D)$
10. $ROC:=ComputeROC(CRU,D,W)$
11. while ($T_{AS}>T_{SLA}$)
12. $AppCRU:=CRUwithMax(ROC)$
13. $AS:=AS\cup AppCRU$
14. $CRU:=UpdateCRU(CRU,AppCRU)$
15. $ROC:=UpdateROC(CRU,D,W,AS)$
16. $T_{AS}:=AverageI/Os(W,D,AS)\times\overline{Cost}_{disk}$
17. endwhile
18. return AS
19. END

上述的缓存单元选择过程类似于经典的背包问题,唯一不同的地方在于物品和物品的价值会因放入背包的物品的不同而改变,我们称之为变种的背包问题,并采用贪婪算法来选择属性集,具体实现如算法 2 所示.算法以租户的访问模式信息、租户数据的统计信息以及租户的 SLA 响应时间需求作为输入,输出最终的缓存属性集.该算法首先计算初始候选缓存单元及其 ROC 值(对应第 9～10 行),然后重复选择 ROC 值最高的候选缓存单元并更新剩余候选缓存单元及其 ROC 值(对应第 11～17 行),其中第 12 行选择 ROC 值最高的作为此次应用的候选缓存单元,并将其添加到最终缓存属性集 AS 中(对应第 13 行),接着更新剩余缓存单元及其 ROC 值(对应第 14～15 行),第 16 行计算了针对当前访问模式下租户的平均响应时间并与租户 SLA 响应时间进行比较作为循环终止条件.

为便于我们更好地对算法 2 进行时间复杂性分析,我们先定义 3 个参数:请求模板数目 m、模板的实例化数目 $n(m\ll n)$和模板的平均属性数目 A;所谓请求模板指的是 SaaS 应用中所包含的 SQL 模板,模板中包含若干个需要用户填充即时值的属性,对于给定的 SaaS 应用来说,模板数目是个常量.实际运行时用户填充即时值到对应的 SQL 模板中称为模板的实例化,实例化出来的不同的 SQL 数目称为模板的实例化数目;模板的平均属性数目是指每一个模板所包含的属性数目的平均值.通过分析算法 2 我们可以发现,该算法主要包含 3 部分:

(1)计算候选缓存单元(第 9 行);

(2)计算候选缓存单元的 ROC 值(第 10 行);

(3)候选缓存单元的迭代选择过程(第 11～17 行).

对于第 1 部分计算候选缓存单元(对应算法 1),其时间复杂性与请求模板数目和模板属性集在后台的数据分块的分布有关,最坏情况下假设一个数据分块只含有一个属性,故其时间复杂性为 $O(mA)$.

对于第 2 部分计算候选缓存单元的 ROC 值,其时间复杂性与候选缓存单元的数目以及每一个候选缓存单元涉及到的实例化模板数目有关,最坏情况下有 mA 个候选缓存单元,并且每一候选缓存单元与所有的实例化模板均有关,故其时间复杂性为 $O(mnA)$.

对于第 3 部分候选缓存单元的迭代过程,其时间复杂性与迭代次数以及每一次迭代所需要的计算量有关,由于我们最多有 mA 个候选缓存单元,因此迭代次数最坏情况下为 mA 次,而每一次迭代中所需要的计算(对应第 12～16 行)只有第 15 行是耗时最多的,其它均为 $O(1)$的,其中第 15 行事实上是第 2 部分的重复执行,其时间复杂性也为 $O(mnA)$,故该部分的时间复杂性为 $O(nm^2A^2)$.

通过上述分析我们可以看出该算法的时间复杂性为 $O(nm^2A^2)$,故其时间复杂性是多项式的,因此该算法可以高效的运行.

4.3 自适应缓存机制的实现

上一节给出了缓存替换单元的生成和选择过程,本节我们将给出自适应缓存机制的完整实现过程,主要包括两个阶段:

(1)监测阶段.统计最近一段时间内租户访问模式的统计信息和每一请求的响应时间.为了预防随机噪声产生严重的震荡,我们将记录足够长的时间间隔内的平均响应时间,并将该时间和租户 SLA 响应时间进行比较.由于响应时间的统计差异,我们认为租户 SLA 没有满足当且仅当当前平均响应时间超过了容忍范围 δ.如果目标 SLA 没有满足,那么我们将进入阶段(b),否则我们将结束当前监测并进入下一轮监测.

(2)微调阶段.在这一阶段我们将采用阶段(1)

收集的相关信息并结合 4.2 节给出的缓存单元的选择过程，计算当前访问模式下的缓存属性集，剔除不应在缓存的属性集，添加尚不在缓存的属性集，并进入阶段(1).

算法 3. 自适应多租户缓存管理算法(Self-Adapt Multi-Tenant Memory Managment, SAMTMM).

```
1. BEGIN
2. for each tenant T
3. IN: D, tenant T's data statistics
4. IN: AS, Attribute Set has been placed in memory
5. SET QS := ∅
6.   while (the current time slot is not elapsed)
7.     if (there is a request Q)
8.       QS := QS∪Q
9.       Time := Time+Q_t
10.      counter++
11.  endif
12. endwhile
13. T_OBS := Time/counter
14. if (|T_OBS − T_SLA|/T_SLA < δ)
15.  Discard collected information and start the next
     time slot
16. else
17.  TmpAS := BASS(QS, D, T_SLA)
18.  Kick out AS − TmpAS
19.  Read in TmpAS − AS
20.  AS := TmpAS
21.  endif
22. endfor
23. END
```

算法 3 中第 6～12 行对应监测阶段，第 13 行计算对应时间段内观察到的平均响应时间，第 14 行判断该时间段内的响应时间是否满足租户 SLA 需求，不满足租户 SLA 需求时，则进入微调阶段对应第 16～21 行.

5　性能测试

5.1　实验环境的搭配

本文采用 TPC-W[20] 作为我们的测试基准平台，采用 Apache Tomcat 作为应用服务器，后台采用 MySQL/InnoDB 作为数据存储引擎，之所以选择 InnoDB 作为存储引擎，是由于 InnoDB 对于事务型请求的加锁粒度基于行的，因此租户可以并行地访问对应的共享架构中的数据而不影响其他租户的执行. Apache 服务器和 MySQL 数据库部署在 PC 机上，采用 Intel 酷睿 2 双核处理器，主频为 2.33 GHz，拥有 2 GB 大小的内存和 250 G 硬盘. 硬盘具体参数如表 4 所示，由于希捷官网并没有公布短寻道磁道上界，而仅公布了平均寻道时间，所以这里的平均寻道时间取 8 ms，而平均旋转延迟和平均传输时间均可由第 3 节公式求得.

表 4　硬盘设备参数(希捷 250G)

参数	值	参数	值
单面磁盘磁道数	3876168	磁道扇区数	63
单盘面的扇区数	2390311315	磁盘转速	7200
短寻道磁道上界	#	扇区大小	512 B
DB 数据块大小	16 KB	平均传输时间	0.43 ms
平均寻道时间	<8.15 ms	平均旋转延迟	4.16 ms

在我们的实验中共有 120 个租户，也即 120 个零售书店，每一个零售书店允许同时在线的用户数最大为 100. 我们采用 10 个相同配置的终端来模拟外部顾客的访问请求，每一个终端可被配置用来模拟 1200 个顾客的针对不同零售书店的请求. 每一顾客提交请求(请求均是经 MySQL 执行计划事先验证处理过的)并等待一个响应；在提交另一个请求之前，我们采用平均值分别为 7 s，14 s，21 s 的负指数分布来模拟高频、中频和低频顾客的思考时间.

5.2　实验数据集

我们采用 TPC-W 基准测试平台中的数据库模式作为我们的测试模式(即租户逻辑模式)，如图 4 所示，括号中列出了每一个属性的数据类型，通过观察我们发现该模式共有 4 种数据类型(INT, DOUBLE, DATE, VARCHAR)，因此我们后台的基于 Chunk Folding 共享存储架构的设计如图 5 所示，其中 Chunk0 包含 4 个业务数据属性列 A1, A2, A3, A4，分别对应数据类型 INT, DOUBLE, DATE, VARCHAR；Chunk1 包含两个业务数据属性列 A1, A2，其数据类型均为 INT；Chunk2 包含 3 个业务数据属性列 A1, A2, A3，其对应的数据类型均为 VARCHAR；这 3 个 Chunk 表的前 4 个属性列为元数据列，用于还原租户的逻辑模式，其中 Tenant 属性用于标记不同的零售书店，并且在这 4 个属性列上建立组合索引 TTCR. 需要注意的是由于文献[4]并未给出如何设计一个最优的基于 Chunk Folding 的共享存储架构，因此图 5 中的基于 Chunk Folding 的共享存储架构并不是一个针对该逻辑模式的最优的共享存储架构；同样本实验中采用的逻辑模式和共享存储模式之间的映射关系也并不一定是最优的，而只是我们目前能找到的比较紧凑的映射关系，其对应的映射关系如表 5 所示.

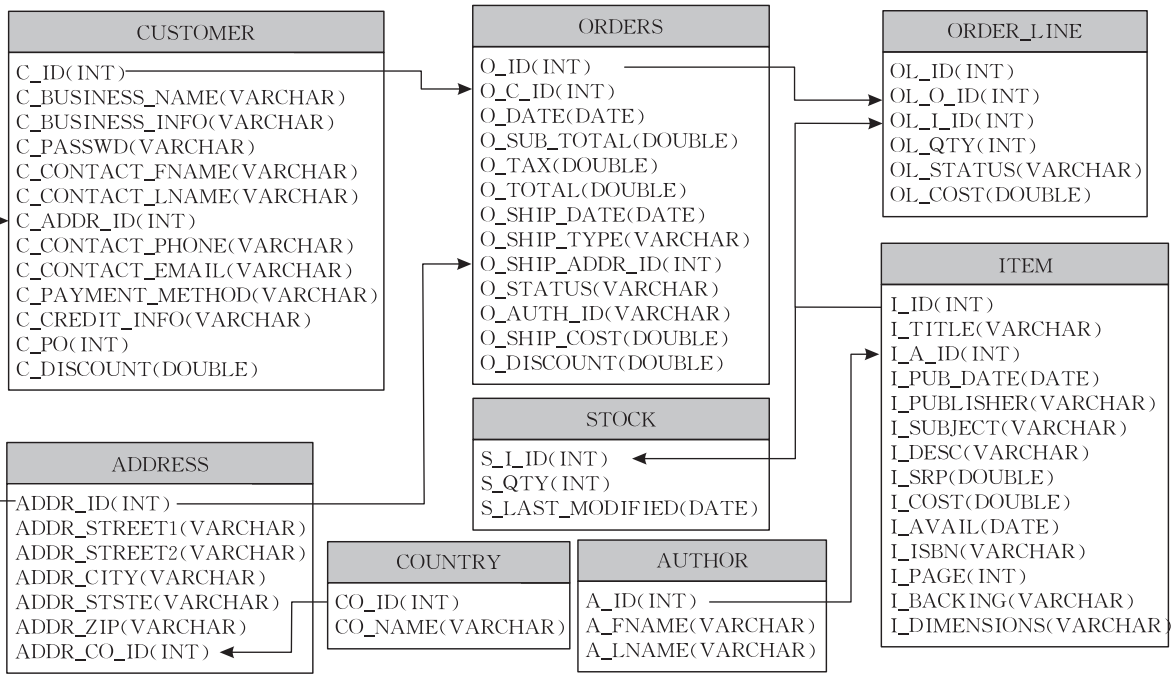


图 4 测试中采用的 TPC-W 中的逻辑模式

表 5 TPC-W 的逻辑模式与 Chunk Folding 共享存储架构的映射关系(#代表空值)

逻辑表名	映射函数
CUSTOMER	α(C_ID,C_DISCOUNT,#,C_BUSINESS_NAME)=(Chunk0,0) α(C_ADDR_ID,C_PO)=(Chunk1,0) α(C_BUSINESS_INFO,C_PASSWD,C_CONTACT_FNAME)=(Chunk2,0) α(C_CONTACT_LNAME,C_CONTACT_PHONE,C_CONTACT_EMAIL)=(Chunk2,1) α(C_PAYMENT_METHOD,C_CREDIT_INFO,#)=(Chunk2,2)
ORDERS	α(O_ID,O_SUB_TOTAL,O_DATE,O_SHIP_TYPE)=(Chunk0,1) α(O_C_ID,O_TAX,O_SHIP_DATE,O_STATUS)=(Chunk0,2) α(O_SHIP_ADDR_ID,O_TOTAL,#,O_AUTH_ID)=(Chunk0,3) α(#,O_SHIP_COST,#,#)=(Chunk0,4) α(#,O_DISCOUNT,#,#)=(Chunk0,5)
ORDER_LINE	α(OL_ID,OL_COST,#,OL_STATUS)=(Chunk0,6) α(OL_O_ID,OL_I_ID)=(Chunk1,1) α(OL_QTY,#)=(Chunk1,2)
ADDRESS	α(ADDR_ID,ADDR_CO_ID)=(Chunk1,3) α(ADDR_STREET1,ADDR_STREET2,ADDR_CITY)=(Chunk2,3) α(ADDR_STSTE,ADDR_ZIP,#)=(Chunk2,4)
STOCK	α(S_I_ID,S_QTY)=(Chunk1,4) α(#,#,S_LAST_MODIFIED,#)=(Chunk0,7)
ITEM	α(I_ID,I_SRP,I_PUB_DATE,I_TITLE)=(Chunk0,8) α(I_A_ID,I_COST,I_AVAIL,I_PUBLISHER)=(Chunk0,9) α(I_PAGE,#)=(Chunk1,5) α(I_SUBJECT,I_DESC,I_ISBN)=(Chunk2,5) α(I_BACKING,I_DIMENSIONS,#)=(Chunk2,6)
COUNTRY	α(CO_ID,#,#,CO_NAME)=(Chunk0,10)
AUTHOR	α(A_ID,#)=(Chunk1,6) α(A_FNAME,A_LNAME,#)=(Chunk2,7)

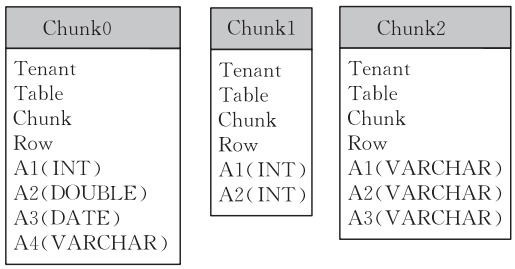


图 5 测试中采用的 Chunk Folding 共享存储架构

实验中数据集,其中AUTHOR表中的A_LNAME和ITEM表的I_TITLE字段是整个模式的基本字段,采用TPC-W提供的工具程序WGEN生成,其它字段按照TPC-W规范随机产生.对于每一个租户我们生成大约20M左右的数据,不同租户的数据是随机交错的插入数据库的,也即每一个租户的数据在磁盘上是分散存储的.需要提及一点的是对于逻辑模式中的主键和外键约束,在我们的Chunk Folding共享架构中均被删除,这样可以方便我们的I/O代价估计,同时也使得影响实验的其它不确定性因素减少,从而使得我们能够更加专注于缓存机制对查询性能的影响.

5.3 实验结果

第1个实验中我们通过变换租户的SLA响应时间需求,对比采用传统MySQL/InnoDB带有中点策略的LRU缓存策略以及本文提出的多租户缓存策略下的缓存消耗,图6给出了单租户条件下的实验结果.

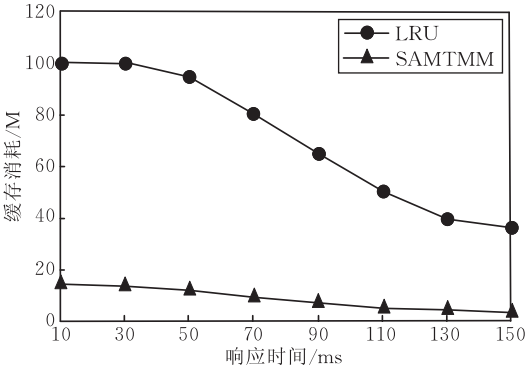
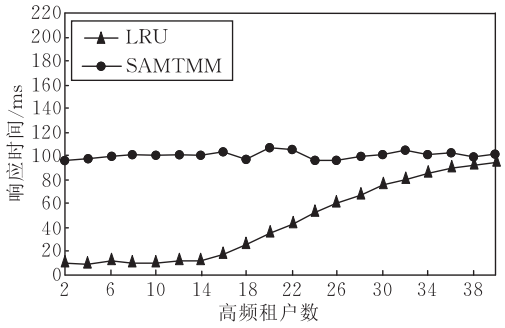


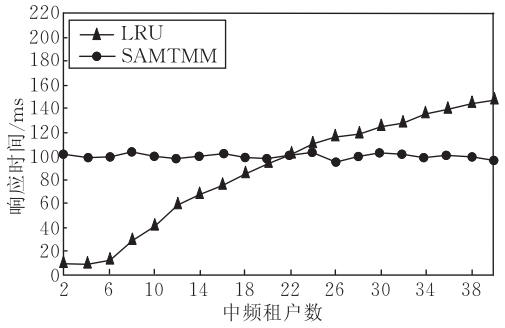
图 6　不同响应时间需求，采用LRU与SAMTMM时的缓存消耗

从图 6 中我们可以看出不同响应时间需求下，SAMTMM 所需要的缓存明显低于 LRU 所需要的缓存. 上一节我们指出每一租户的数据大小为 20 M，这就意味着理想情况下缓存单个租户的全部数据至多需要 20M 左右的缓存. 从图 6 所示的实验结果来看，在响应时间需求为 10 ms 时，SAMTMM 使用了约 16 M 左右的缓存从而达到了这一目标需求，其大小十分接近租户数据的实际大小，然而采用 LRU 缓存策略却花费了大约 100 M 左右的缓存，造成这一现象的根本原因在于 LRU 缓存策略是以数据块作为缓存单元的，而在多租户共享存储架构下，数据块中存储了大量其他租户的无关数据，浪费了缓存空间.

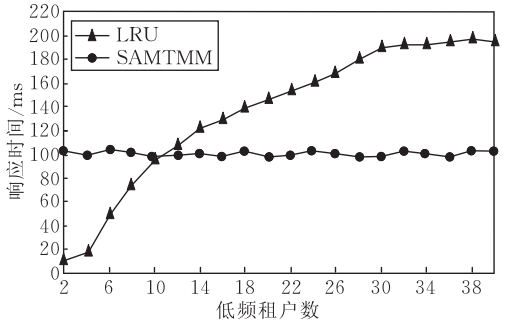
第 2 个实验中我们设置 MySQL 的缓存大小为 700 M，并设租户的 SLA 响应时间需求均为 100 ms，从图 6 可以看出为满足这一需求，SAMTMM 为每一租户分配大约 5 M 缓存（该值不随活跃租户数目改变），而 LRU 为每一租户分配大约 50 M 缓存（该值随活跃租户数目的增多会逐渐减少）；依据租户的访问频率将其分为 3 个类别：高频访问、中频访问和低频访问（通过设置不同零售商的顾客平均思考时间来模拟实现）. 通过等比例地增加不同类别租户的数目（一次增加 6 个，低频、中频和高频各 2 个），对比 LRU 缓存机制与 SAMTMM 缓存机制下不同类别租户平均响应时间. 从图 7(a)中可以看出，LRU 缓存策略下高频租户的响应时间明显低于租户 SLA 响应时间需求；从图 7(b)和(c)中可以看出在租户数目较少时，中频和低频的响应时间均比较快，当租户数目增多的时候，中频和低频的响应时间都会明显变慢，并且远远高于租户 SLA 响应时间需求，此外低频租户的响应时间增长的明显比中频租户的要快很多，出现这种现象的原因在于，LRU 缓存机制从数据库整体性能考虑，在初始租户数目较少的时候，缓存相对多，所以采用 LRU 缓存策略就将大量缓存分配给了这些租户，使得其响应时间很快（均低于 SLA 需求）；而当租户数目逐渐增多时，缓存就变得紧张，采用 LRU 缓存策略就会将更多的缓存分配给了高频访问的租户，所以中频和低频访问的租户大量的缓存被抢占了，从而响应时间会



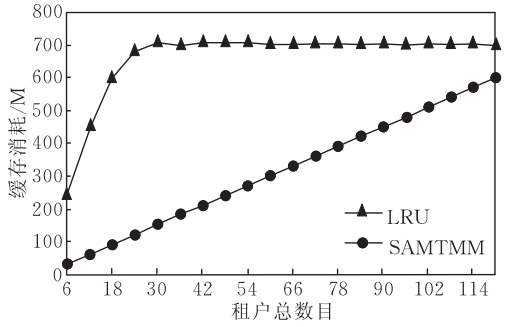
(a) 高频租户响应时间随租户数目的变化情况



(b) 中频租户响应时间随租户数目的变化情况



(c) 低频租户响应时间随租户数目的变化情况



(d) 缓存消耗与租户总数的对应情况

图 7　采用 LRU 和 SAMTMM 策略时 3 种访问频率租户的平均响应时间以及对应的整体缓存消耗

明显变慢;对于采用 SAMTMM 缓存机制时,高频、中频和低频租户的响应时间均在租户 SLA 响应时间附近波动,而并没有出现缓存资源分配的不合理现象.从图 7(d)中可以看出采用 SAMTMM 缓存机制,分配的缓存数量与租户数目成线性增长,这是因为我们这里的租户除了访问频率不一样以外,其它各方面都是完全一样的,因此每一租户为满足其 SLA 所需要的缓存就基本相同;而对于 LRU 缓存机制来说,开始一段时间分配的缓存数量随租户数目成线性增长,而后期保持 700M 不变,这是由于起始租户数目不多时,每一租户的数据都被完全缓存,而采用块缓存导致缓存被快速消耗,随租户数目的进一步增多,LRU 会替换掉低频和中频租户的缓存以留给高频租户.

6 总 结

本文我们给出了基于 Chunk Folding 的多租户共享架构下的缓存管理机制,通过采用自适应的缓存单元可以大幅减少由于共享架构导致的以数据块作为缓存单元导致的资源浪费;通过租户 SLA 作为驱动为每一个租户分配缓存可以防止租户间资源分配的不合理,如高频租户抢占大量缓存,最终使得每一租户获得满足其 SLA 需求的尽量少的缓存.本文假设租户进入系统后保持在一定的活跃状态,而如何监测租户的活跃程度并设定一个合理的阈值,驱逐不活跃租户的缓存空间,保证每一租户的 SLA 需求的同时提高整体利用率是下一步需要解决的问题.另外本文目前给出的是单节点下的缓存策略,如何考虑多数据节点下的数据分布并将其拓展到多节点上以满足多租户 SLA 需求是下一步需要解决的问题.

参 考 文 献

[1] Aulbach S, Jacobs D, Kemper A, Seibold M. A comparison of flexible schemas for software as a service//Proceedings of the 35th International Conference on Management of Data (SIGMOD). Providence, Rhode Island, USA, 2009: 881-888

[2] Weissman C D, Bobrowski S. The design of the force.com multitenant internet application development platform//Proceedings of the 35th International Conference on Management of Data(SIGMOD). Providence, Rhode Island, USA, 2009: 889-896

[3] Maier D, Ullman J D. Maximal objects and the semantics of universal relation databases. ACM Transactions on Database Systems, 1983, 8(1): 1-14

[4] Aulbach S, Grust T, Jacobs D, Kemper A, Rittinger J. Multi-tenant databases for software as a service: Schema-mapping techniques//Proceedings of the 34th International Conference on Management of Data(SIGMOD). Vancouver, BC, Canada, 2008: 1195-1206

[5] Storm A J, Garcia-Arellano C, Lightstone S S, Diao Y, Surendra M. Adaptive self-tuning memory in DB2//Proceedings of the 32nd International Conference on Very Large Data Bases(VLDB). Seoul, Korea, 2006: 108-1092

[6] Malik T, Burns R, Chaudhary A. Bypass caching: making scientific databases good network citizens//Proceedings of the 35th International Conference on Data Engineering (ICDE). Tokyo, Japan, 2005: 94-105

[7] Brown K P, Carey M J, Livny M. Managing memory to meet multiclass workload response time goals//Proceedings of the 19th International Conference on Very Large Data Bases(VLDB). Dublin, Ireland, 1993: 328-341

[8] Chockler G, Laden G, Guy Laden, Vigfusson Y. Data caching as a cloud service//Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware (LADIS). Zürich, Switzerland, 2010

[9] Wachs M, Abd-El-Malek M, Thereska E, Ganger G R. Argon: Performance insulation for shared storage servers//Proceedings of the 5th USENIX Conference on File and Storage Technologies(FAST). San Jose, CA, 2007: 61-76

[10] Patrick C M, Garg R, Son S W, Kandemir M. Improving I/O performance using soft-QoS based dynamic storage cache partitioning//Proceedings of the 2009 International Conference on Cluster Computing and Workshops (CLUSTER). New Orleans, Louisiana, 2009: 1-10

[11] Lu Y, Abdelzaher T F, Avneesh Saxena. Design, implementation, and evaluation of differentiated caching services. IEEE Transactions on Parallel and Distributed Systems, 2004, 15(5): 440-452

[12] O'Neil Elizabeth J, O'Neil Patrick E, Weikum Gerhard. The LRU-K page replacement algorithm for database disk buffering//Proceedings of the 19th International Conference on Management of Data (SIGMOD). Washington, DC, USA, 1993: 297-306

[13] The InnoDB Buffer Pool. http://dev.mysql.com/doc/refman/5.1/en/innodb-buffer-pool.html

[14] Yao S B. Approximating block accesses in database organizations. Communications of the ACM, 1977, 20(4): 260-261

[15] Selinger P G, Astrahan M M, Chamberlin D D, Lorie R A, Price T G. Access path selection in a relational database management system//Proceedings of the 1979 International Conference on Management of Data (SIGMOD). Boston, MA, 1979: 23-34

[16] Baron Scbwartz, Peter Zaitsev, Vadim Tkacbenko, Jeremy Zawodny D, Arjen Lentzn, Derek J Balling. High Performance MySQL. 2nd Edition. O'Reilly Media, 2008

[17] Triantafillou P, Christodoulakis S, Georgiadis C A. A comprehensive analytical performance model for disk devices under random workloads. IEEE Transactions on Knowledge and Data Engineering, 2002, 14(1): 140-155

[18] Luo Q, Naughton J F. Form-based proxy caching for database-backed web sites//Proceedings of the 19th International Conference on Very Large Data Bases (VLDB). Roma, Italy, 2001: 191-200

[19] Wang X, Malik T, Burns R, Papadomanolakis S, Ailamaki A. A workload-driven unit of cache replacement for mid-tier database caching//Lecture Notes in Computer Science 4443. Berlin: Springer, 2007: 374-385

[20] Transaction Processing Council. The TPC-W Benchmark. http://www.tpc.org

YAO Jin-Cheng, born in 1986, M.S. candidate. His research interests include database, cloud computing.

ZHANG Shi-Dong, born in 1969, Ph. D., professor. His research interests include database, web data integration, cloud computing.

SHI Yu-Liang, born in 1978, Ph. D., associate professor. His research interests include service computing, cloud computing and database.

LI Qing-Zhong, born in 1965, Ph.D., professor, Ph.D. supervisor. His research interests include large-scale network data management, web data integration.

Background

With the development of network and maturity of application software, Software-as-a-Service, i.e. SaaS, is becoming a new software delivery model where owner-ship and management of applications are outsourced to a service provider.

In order to gain benefit with scale effect, service provider adopts shared storage shared architecture to store tenants' business data. However, due to the lack of the consideration of multi-tenancy, multi-tenant database constructed from the traditional database presents insufficient in the management of database memory. First, traditional database memory management mechanism adopts page as the cache replacement unit, while in multi-tenant shared architecture each page contains lots of other tenants' data which is irrelevant to the request, so use page as the replacement unit will result in a waste of memory. Second, memory management of the traditional databases lack the property of multi-tenancy, it conducts the memory management from the perspective of improving the overall performance without considering each tenant's SLA requirement which will lead to a unreasonable memory allocation among tenants. Tenants with high access frequency will take over a majority part of the memory, while tenants with low access frequency can't get the necessary memory they need to satisfy their SLAs.

In order to solve the problem, we propose a Self-Adaptive Multi-Tenant Memory Management mechanism (SAMT-MM). It adapts with the current access model and combines with properties of the shared storage schema, then the memory management problem is abstracted as a modified Knapsack problem and a greedy algorithm is adopted to compute the corresponding memory for each tenant until it satisfies the tenant's SLA.

The research is supported by the National Key Technologies R&D Program No. 2009BAH44B02; the National Natural Science Foundation of China under Grant No. 90818001; the Natural Science Foundation of Shandong Province of China under Grant Nos. 2009ZRB019YT, ZR2010FQ026; Key Technology R&D Program of Shandong Province under Grant No. 2010GGX10105.