

基于 JDBC 的缓存数据细粒度管理的研究

韩 兵¹, 沈 冲¹, 方英兰²

(1. 北方工业大学 计算机学院, 北京 100144;

2. 大规模流数据集成与分析技术北京市重点实验室, 北京 100144)

摘 要: 现有的 JDBC 缓存技术对缓存的管理是粗粒度的, 然而由于 Web 请求的多样性, 粗粒度的缓存管理并不能合理地利用缓存的优势。为提高 Web 应用系统的效率以使用户获得更好的使用体验, 分析了内存数据库、进程内数据缓存技术和 JDBC 缓存的技术, 在原有对 JDBC 缓存的研究基础上进一步深入研究, 提出了基于 JDBC 的缓存数据自我管理、基于缓存生命周期的一致性维护策略及其实现方案。通过对 JDBC 缓存细粒度的管理, 实现了对缓存数据的查询和更新, 将数据库的一部分功能前置到了应用服务器中, 减少了数据库的负载, 因此提高了缓存的使用率、缩短数据请求的路径, 从而提升 Web 系统的综合性能。测试结果表明, 该方案能有效提高查询请求的命中率, 减少数据访问耗时。

关键词: JDBC; Web 应用; 缓存再利用; 一致性维护; 命中率

中图分类号: TP311.1

文献标识码: A

文章编号: 1673-629X(2019)12-0066-06

doi: 10.3969/j.issn.1673-629X.2019.12.012

Research on Fine-grained Management of Cached Data Based on JDBC

HAN Bing¹, SHEN Chong¹, FANG Ying-lan²

(1. School of Computer Science, North China University of Technology, Beijing 100144, China;

2. Beijing Key Laboratory on Data Integration and Analysis Technology of Large-scale Stream, Beijing 100144, China)

Abstract: Existing JDBC caching technologies manage caches at a coarse-grained level. However, due to the diversity of Web requests, coarse-grained cache management does not properly take advantage of caching. In order to improve the efficiency of Web application system and make users get better experience, we analyze the in-memory database, in-process data cache technology and JDBC cache technology, conduct further study on the basis of the original research on JDBC cache, and propose a JDBC-based cache data self-management, consistency maintenance strategy based on the cache life cycle and its implementation scheme. Through the fine-grained management of JDBC cache, we realize the query and update of the cached data, and preloads part of the database to the application server to reduce the load of the database, thus improving the utilization rate of the cache, shortening the path of data requests, and improving the comprehensive performance of the Web system. The test shows that the scheme can effectively improve the hit rate of query request and reduce the time of data access.

Key words: JDBC; Web application; cache reuse; consistency maintenance; hit rate

0 引言

伴随着信息技术和互联网的发展, 各类 Web 应用已经渗透到人们生活的方方面面, 随着越来越多用户的使用, Web 应用也遭受着巨大的考验。Web 应用的响应时间是很重要的考量因素, 响应时间越快, 用户继续停留的可能性越高, 进而用户使用率和 Web 应用的转化率会更高。在实际场景中, 如果等待的时间太长,

用户大概率会离开应用。所以如何降低 Web 应用的响应时间, 以提高用户体验, 成为了当前 Web 应用技术的重点。

缓存机制的广泛应用能有效提升 Web 系统的用户体验。通过将数据直接缓存到 JDBC (Java database connectivity) 的内存中, 而不借助第三方插件, 以此来提升对历史查询数据的再利用, 减少了磁盘和数据库

收稿日期: 2019-01-12

修回日期: 2019-05-14

网络出版时间: 2019-09-24

基金项目: 国家自然科学基金 (61672040)

作者简介: 韩 兵 (1971-), 男, 硕士, 副研究员, CCF 会员 (57372M), 研究方向为计算机应用、数据库优化、数据分析和数据挖掘; 沈 冲 (1994-), 男, 硕士研究生, 研究方向为计算机应用。

网络出版地址: <http://kns.cnki.net/kcms/detail/61.1450.TP.20190924.1535.026.html>

的访问频率,从而提高了 Web 应用的整体访问效率是一种新的缓存解决方案。

通过对 JDBC 数据持久化优化技术的研究,文中提出了基于 JDBC 的缓存管理、缓存再利用和基于缓存生命周期的一致性维护的策略和实现方案。通过对 JDBC 的改造,实现了对缓存的细粒度管理,缩短了数据请求的路径长度,从而提高 Web 系统的性能。实验结果表明,该方案明显减少了系统响应时间,有效提高了系统性能。

1 研究背景

在 Web 系统中,磁盘的 I/O 操作、网络的连接开销、数据请求路径的长度是影响系统性能的关键因素^[1-2]。在数据库方面,内存数据库以内存为存储介质,没有数据访问时磁盘 I/O 的瓶颈,极大提升了数据访问的效率。

Memcached 是一个 key-value 存储系统,支持多种类型数据的存储,为了保证效率,这些数据都被存储到内存中,减少了大量的磁盘 I/O 操作^[3]。Redis 也是一种高性能的内存数据库,与 Memcached 不同的是,Redis 不仅可以单独作为内存数据库,还可以用作 MySQL 的缓存服务器,这样既兼顾了数据访问效率,又避免了内存空间的局限^[4]。这两种内存数据库都利用了内存的高吞吐能力,避免了磁盘的 I/O 操作,提高了 Web 系统的性能,但它们都无法避免网络连接的开销,也无法缩短数据请求的路径长度,而且都属于系统额外的插件,增加了开发成本和维护成本。

JDBC 是 Java Web 程序和不同数据库之间连接的桥梁,JDBC 为多种数据库提供了统一的访问方式,简化了开发流程^[5],但 JDBC 本身并没有提供数据缓存和数据再利用的功能,仅仅实现了数据通道和传输的功能。目前针对数据缓存功能有多种研究,其中 Hibernate 是最受欢迎的对象-关系映射框架。它对 JDBC 进行了轻量级的对象封装,让 Java 程序员更容易操作数据库^[6],但其缓存功能需要手动配置,配置的好坏直接影响着系统的性能,所以对程序员的经验要求较高。本课题组在文献^[7]提出基于 JDBC 的数据自主识别、前置驻留与快速访问的策略和实现方案,利用 Web 数据的特点,80% 的数据访问会落在 20% 的数据区域,将 20% 的热点数据以 key-value 结构缓存到应用服务器中,并且使改造后的 JDBC 优先从缓存中请求数据,减少了数据库的请求,也缩短了数据请求的路径长度,但它存储的是全表数据,并且只针对全表查询优化,所以其缓存利用率不高。

对于使用 JDBC 连接数据库的 Web 应用系统,如果能够将用户经常访问的热点数据缓存到应用服务

器,同时还将这些数据的 INSERT, DELETE, UPDATE, SELECT 操作也前置到应用服务器中,并且有适当的数据一致性维护,这样就既可以利用内存的数据高吞吐能力,减少网络连接的开销,缩短数据请求的路径长度,从而提高系统性能,减少系统响应时间。

2 基于 JDBC 的缓存数据管理的模型构建

基于 JDBC 的缓存数据管理的模型如图 1 所示。该模型主要包括三部分:最上层是视图层,用户通过该层进行系统功能访问;中间是应用服务器层,该层的核心是业务处理以及缓存数据的管理;最底层是数据库,对系统的数据进行存储。

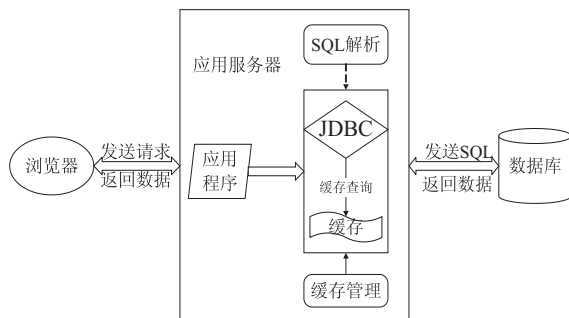


图 1 JDBC 数据管理模型

应用服务器作为应用层与数据库之间的桥梁,负责接收和处理应用层的请求,并在适当的时候向数据库发送 SQL 和接收数据库的返回数据,然后将数据返回给应用层。应用服务器层与数据库的频繁连接是很耗时的操作,而 JDBC 是 Java 程序与数据库通信的常用接口,适当扩充 JDBC 的功能以减少应用服务器与数据库的连接可以有效降低请求响应的时间,提高用户体验。

文中在原始 JDBC 上增加了全表查询结果及其增删改 SQL 语句的缓存功能、SQL 语句标准化功能、缓存查询功能、缓存更新功能以及缓存与数据库的一致性维护策略,通过对缓存细粒度的管理,降低了数据库的访问频率,提高了用户体验。JDBC 接收到全表查询后,向数据库查询数据并给用户返回查询结果的同时,也将全表查询的结果保存到缓存空间,后续的缓存查询和缓存更新都基于这些全表查询缓存结果集。JDBC 接收到用户的 SQL 请求后,将 SQL 语句中的关键信息提取出来,按操作类型将这些信息放入不同的解析模型中,方便后续对缓存的遍历和更新操作。

如果用户的 SQL 语句为查询操作,并且缓存中有对应的表,则直接取出缓存数据或者对缓存数据遍历,组装出结果集;如果缓存中没有查询语句对应的表,则直接向数据库查询,如果此查询为全表查询,向用户返回结果的同时将结果集保存到缓存空间中;如果用户的 SQL 语句为 INSERT、DELETE、UPDATE 操作并且

缓存中有对应的表数据,则使用 SQL 解析出来的信息对缓存表进行遍历、更新、删除的操作,同时将 SQL 语句也缓存起来,等待此缓存表被替换时,缓存的 SQL 语句按时间顺序执行回数据库。

3 关键算法的分析及实现

文中对 JDBC 缓存结果集进行分析,理解了缓存结果集的 5 个组成部分,针对这 5 个部分,提出了 SQL 标准化、缓存数据再利用、基于缓存生命周期的一致性维护策略这三个优化措施。

3.1 JDBC 缓存结果集的数据结构分析

JDBC 缓存结果集由 5 个主要部分组成,分别为数据库名、表名、数据库连接信息、列名序号映射表、每行数据列表。前三项定义了缓存数据的外部属性,后两项定义了缓存数据的内部属性,通过这 5 个主要部分的共同作用,数据库表被映射到了内存中,组成了缓存表在内存中的数据结构。现将各个部分做如下说明:

(1)数据库名、表名、数据库连接信息:数据库名指的是缓存数据对应的数据库数据所在的数据库名称,表名指的是缓存数据对应的数据库数据所在的表名,数据库连接信息指的应用服务器与缓存数据对应的数据所在的服务器的连接信息。一台数据库服务器上可能有多个数据库,每个数据库中可能有多张表,缓存表是数据库数据的一种映射,以上三个部分定义了缓存数据的外部属性,让缓存数据与数据库数据之间有了精准的映射关系,在后期需要将缓存数据同步到数据库中时,缓存数据的外部属性起到了至关重要的作用。

(2)列名序号映射表:它保存了字段名称与字段序号的映射关系,数据结构为 HashMap,以 key-value 的形式保存,key 为序号,value 为字段名,序号从 0 开始,到字段总数为止,分别给字段名称标上序号。在后续遍历每个字段的操作中,只需要对序号遍历就能定位到具体的字段。存储每条数据的每个字段信息时,不用一一存取每个字段的名称,只需要保存每个字段序号,这样减少了数据的占用空间。

(3)每行数据列表:它是一个有序列表,保存了每条数据的具体信息,每条信息的数据结构是 HashMap,以 key-value 的形式保存,key 为列序号,value 为本列的值,这样避免重复保存字段信息而占用大量空间的问题。每条信息的有序排列形成了原数据库表在内存中的组织形式,文中可以对这个有序列表进行遍历,也可以用字段名与列序号的映射关系快速地定位到所需要的列,进而可以在缓存表的基础上做细粒度的查询和更新操作。

针对缓存数据的细粒度处理,文中主要的操作为遍历、匹配和更新。针对遍历操作,在缓存对象上扩展 NEXT()方法,此方法初始指向缓存表的第一条数据,每次调用都取出所指向的数据,并调整指向到下一条数据,对缓存表从头到尾依次遍历,直到遍历完缓存表的所有数据为止。针对匹配操作,在缓存对象上扩展 MATCH()方法,使用从 SQL 中解析出来的条件信息对遍历的每一条数据进行匹配,若满足条件则取出此条数据做进一步操作。针对更新操作,在缓存对象上扩展 UPDATE()方法,首先拷贝一份每行数据列表,然后遍历拷贝表的每一条数据,匹配每一条符合条件的数据,使用从 SQL 中解析出来的更新信息对满足匹配条件的数据做更新操作,遍历完所有数据后,这些更新后的数据组成一个新的每行数据列表,最后用新的每行数据列表替换原始的每行数据列表。

3.2 SQL 标准化

文中针对的是热点数据的全表缓存数据的细粒度管理,热点数据有查询多、但较少更新的特点。为了保证对缓存数据操作的性能和减少出错的可能,文中过滤所有带有特殊函数的 SQL 语句,例如 COUNT()、CURDATE()、NOW()等特殊函数,遇到此类 SQL 语句,不做 SQL 标准化函数处理,而是直接交由数据库处理。

文中把每个 SQL 请求作为一个解析项,JDBC 使用这些解析数据对缓存表进行细粒度管理,每个 SQL 解析项记为 S,为描述方便,将 SQL 解析项的详细结构表示为如下六元组: $S = (O, OP, T, C, U, W)$ 。

六元组的每个属性的含义如下:

(1)O 是原始 SQL 语句,保留了最原始的 SQL 语句。

(2)OP 是 SQL 语句的操作类型,INSERT, DELETE, UPDATE, SELECT 其中一种,不同操作类型的 SQL 语句在结构上有所区别。文中在对缓存表的管理操作中,根据操作类型的不同使用相应的 SQL 解析数据对缓存表管理。

(3)T 是 SQL 语句操作的表名,文中针对的是单表操作,每条 SQL 语句对应一张数据库表,此项记录了本数据库表的表名。

(4)C 是与 SQL 操作相关的列名列表,此项针对 INSERT 和 SELECT 操作而设,用于记录各列的列名,此项为了方便缓存管理操作快速定位到某条数据的具体字段上。

(5)U 是 SQL 更新语句的更新操作列表,此项针对 UPDATE 操作而设,在 UPDATE 操作中,形如 column = value 的操作被此项以三元组的形式记录下来,分别记为操作列名、操作符、值,多个子操作组成了

此条 SQL 更新语句的更新操作列表。

(6) W 是 SQL 语句的条件项列表,记录了 WHERE 关键字后面的条件语句,条件语句由多个子条件组成,并且由逻辑运算符连接而成。文中将每个子条件项以三元组的形式记录下来,分别记为列名、条件运算符、值,多个子条件和逻辑运算符组成了 SQL 语句的条件项列表。

SQL 标准化的流程如图 2 所示。

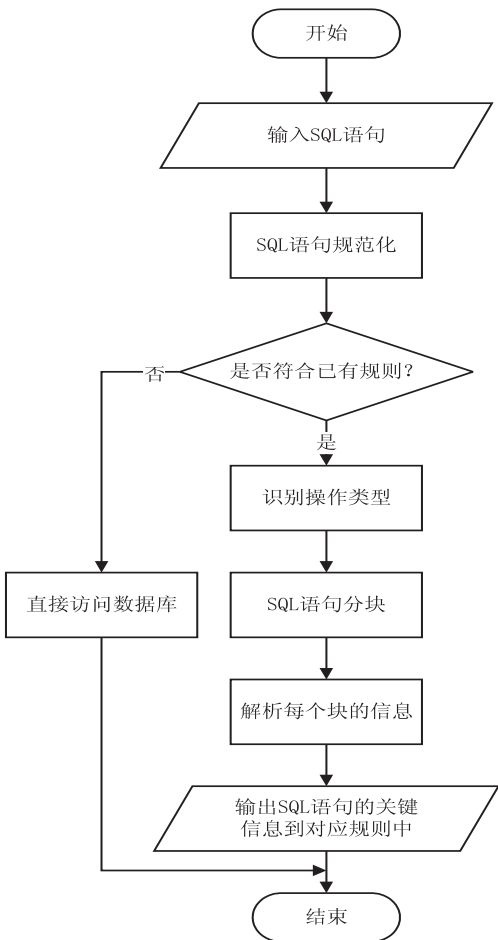


图 2 SQL 标准化流程

3.3 缓存数据再利用

Web 数据访问具有不均匀的特点,80% 的访问量落在 20% 的数据区域内^[8],并且大部分访问是读取数据,因此,针对这少部分的热点数据的优化对系统的响应时间有很重要的作用。当前基于 JDBC 的缓存管理只做了 SQL 语句与全表查询结果集的粗粒度模式缓存,只能针对全表查询优化,显然,当前的缓存管理方案并没有充分利用缓存的优势。

因此,在增、删、改、查多种请求类型共存的实际应用场景下,缓存的细粒度管理以及再利用就有很重要的作用了,这样就可以把更多的数据请求放在应用服务器层,减少数据库的访问。文中将缓存再利用分为基于缓存数据的条件查询和基于缓存数据的短路径更新,基于缓存数据的条件查询是一种在全表缓存数据

上的查询操作,使用 SQL 语句解析出的规则在缓存数据中找出匹配的数据,这些数据的集合即为查询结果集。基于缓存数据的短路径更新是一种在全表结果集上的更新操作,在以往的 JDBC 缓存管理研究中,接收到 INSERT, DELETE, UPDATE 操作后,都先将对应的缓存结果集抛弃,然后将 SQL 语句提交至数据库执行,这样的操作会导致 JDBC 频繁的替换数据,并且数据操作也要经过数据库,增加了数据响应时间。

文中对 JDBC 中的 ResultSet 接口进行扩展^[9-10],实现缓存数据的更新操作,将 UPDATE, DELETE, INSERT 操作临时作用在缓存数据上,然后在后期缓存失效时将这些操作按时间顺序执行回数据库。这样避免了缓存数据的频繁替换,也减少了数据库的连接次数,只需要在缓存失效时连接一次数据库并集中执行 SQL 语句,可以有效减少系统响应时间。缓存数据再利用的算法流程如图 3 所示。

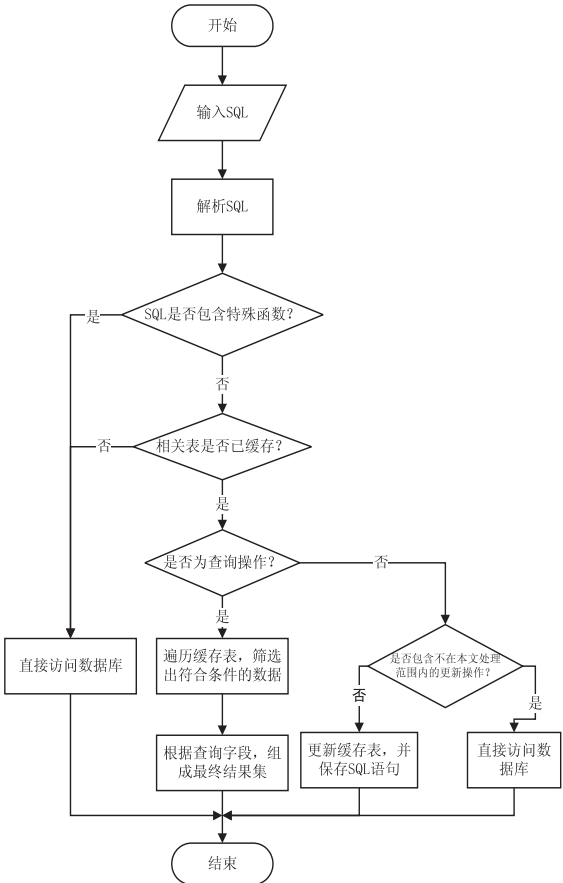


图 3 缓存再利用流程

JDBC 接收到 SQL 语句后,首先将 SQL 语句标准化,如果 SQL 语句中包含一些特殊函数,则将 SQL 语句直接送至数据库执行,若属于文中处理范围内的 SQL 语句且目标表存在于缓存空间中,则使用 SQL 解析出来的关键信息做进一步处理;若为查询操作,则遍历缓存表,匹配出符合条件的数据,组成新的结果集返回给用户;若为 INSERT, DELETE, UPDATE 操作,则

在原有的缓存表的基础上做增删操作或者匹配出符合条件的数据,对其做相应的修改。

3.4 基于缓存生命周期的一致性维护策略

文中对缓存数据进行细粒度管理,将部分查询和更新操作迁移到应用服务器,减少了数据库的访问,但缓存中的数据终究会有失效的时候,比如在缓存空间达到上限时,需要把某些缓存表移出缓存空间,以便保存新的缓存表。缓存替换时,需要用合适的方法将缓存表的历史修改同步到原数据库中,这样才能保证缓存数据与数据库数据的一致性。

文中定义缓存表的生命周期为从缓存记录某张表为起点到本张缓存表被替换为终点的时间段。在缓存表的生命周期内,可能会有若干次 INSERT,DELETE,UPDATE 操作,文中管理多张表的 SQL 语句使用的数据结构是 ConcurrentHashMap,此数据结构是线程安全的,支持读和写的并发操作^[11-12],以表名为 key,此表相关的 SQL 语句列表为 value,管理单张表的 SQL 语句使用的数据结构为有序列表 ArrayList。

文中提出了基于缓存生命周期的一致性维护策略,在缓存空间不足时触发一致性维护。使用 LRU (least recently used, 最近最少使用) 缓存替换算法^[13-14],使用 LinkedHashMap 维护多张表的缓存结果集。LinkHashMap 是 HashMap 的子类,在 HashMap 的基础上额外维护着一个双向链表,它可以做到按存入顺序或访问顺序对元素有序迭代^[15]。每次查询操作若能从缓存中获取结果集,则将元素移到双向链表的尾部,循环往复,因此链表头部的元素是最近最少被使用的。在缓存生命周期内,可能有若干条更新操作,这些 SQL 语句都有序的存在于 SQL 语句管理队列中,执行一致性维护策略时,先连接缓存表对应的数据库,然后按时间先后顺序一条一条执行 SQL。该一致性维护策略将缓存表生命周期之内的多次 SQL 连接操作减少为一次,减少了网络请求的时间,并且维护了缓存表数据与数据库数据的一致性。一致性维护的流程如图 4 所示。

4 实验设计与验证

为了验证文中方案对 Web 系统性能提高的有效性,分别在一个电子商务网站上应用原始的 JDBC 方案、现有的 JDBC 优化方案、文中提出的 JDBC 方案,对比这三种方案的缓存命中率和响应时间。测试环境有三台服务器,分别部署原始的 JDBC 方案、现有的 JDBC 优化方案、文中提出的 JDBC 优化方案,Web 应用服务器为 Tomcat 服务器,配置为 CPU @3.40 GHz,操作系统为 Windows 7,内存为 8 GB,数据库使用 MySQL5.7 版本。

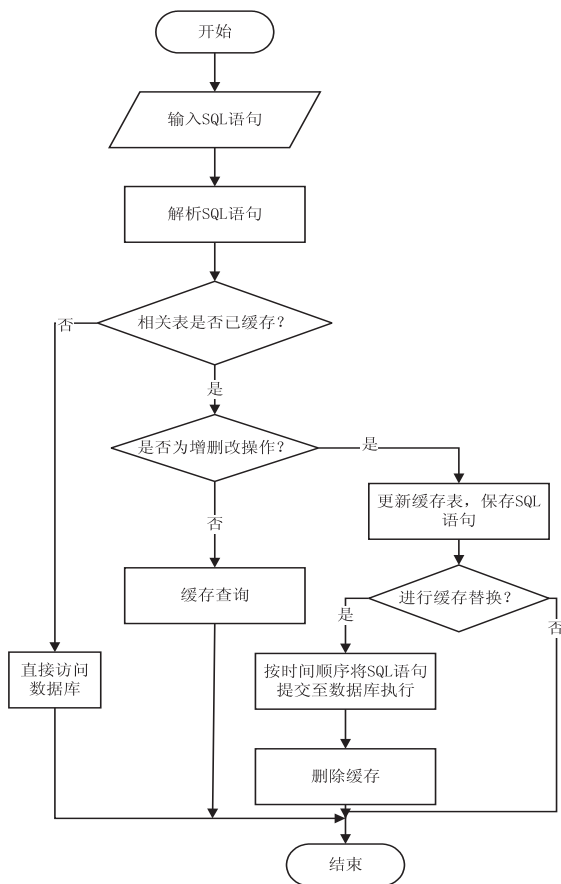


图 4 一致性维护流程

4.1 实验方案

数据库中有 20 张表,其中 100 条数据的表有 4 张,1 000 条数据的表有 10 张,10 000 条数据的表有 6 张,每条数据的大小为 1 K。设定两种优化的 JDBC 方案缓存初始空间大小为 50 M,缓存使用空间大于 50 M 时,自动进行缓存置换,将最早进入缓存空间的表移出缓存空间。为了模拟真实的使用场景,随机生成一些 SQL 语句,其中全表查询的占 30%,带条件的查询语句占 50%,增删改操作占 20%。用这些 SQL 语句分别向三种 JDBC 方案的应用服务器连续发出请求,对比三种方案在不同的请求条数下的命中率和响应时间。

4.2 缓存命中率对比分析

原始的 JDBC 没有缓存功能,所以这里比较文中方案和现有的 JDBC 优化方案的缓存命中率。每次查询请求的时候,如果请求结果取自缓存,文中记一次命中,连续请求完成时,总命中次数/总请求次数即为缓存命中率。结果如图 5 所示,观察图表可以发现,文中方案明显优于现有的方案。因为文中方案可以在全表查询的结果集上做细粒度的查询,使得缓存的利用率更高。随着查询次数的增多,缓存表越来越多,查询命中率也越来越高,但因为缓存空间的大小有限,缓存表的数量也是有上限的,所以随着查询次数的增多,缓存

命中率会趋于平缓。

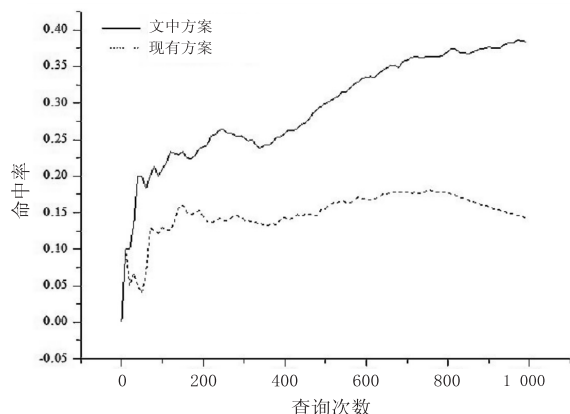


图5 缓存命中率对比

4.3 响应时间对比分析

响应时间对比分析的是原始 JDBC、现有的 JDBC 优化技术、文中优化的 JDBC 技术这三种方案的 Web 应用的请求响应时间,结果如图 6 所示。观察图表可以发现,原始的 JDBC 方案每次操作都直接访问数据库,因此响应时间最长;现有的方案只在全表请求时才使用缓存,缓存使用场景较少,因此响应时间较短;文中方案对缓存表的相关请求都有优化效果,并且增删改的操作都会优先使用缓存结果集,使得缓存的再次使用的场景更广,因此响应时间最短。

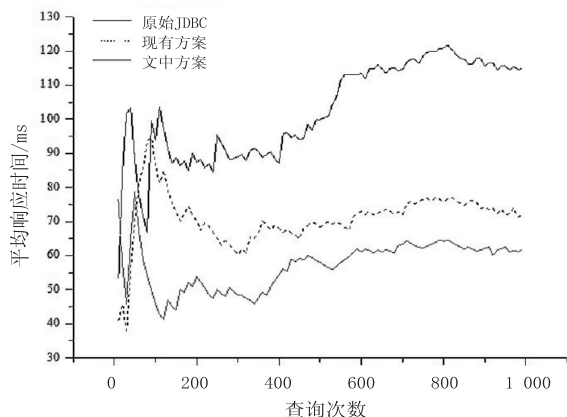


图6 响应时间对比

5 结束语

研究了内存数据库技术、应用服务器缓存技术、JDBC 缓存的技术,分析了各自的优劣,并对基于 JDBC 的 Web 应用系统缓存优化进行了深入的研究,分析了 JDBC 缓存的几个关键部分,在原有的 JDBC 缓存研究基础上对缓存空间进行更细粒度的管理,提高了缓存的利用率,减少了网络请求,降低了数据库的压力,提高了系统的整体性能。

参考文献:

- [1] 王亚楠,吴华瑞,黄 锋. 高并发 Web 应用系统的性能优化分析与研究[J]. 计算机工程与设计,2014,35(8):2976-2981.
- [2] 康国胜,刘建勋,唐明董,等. 面向多请求的 Web 服务全局优化选择模型研究[J]. 计算机研究与发展,2013,50(7):1524-1533.
- [3] 刘 亮,徐步东,谭艳艳. 基于 Memcached 内存对象缓存技术应用研究[J]. 计算机技术与发展,2015,25(11):204-208.
- [4] 郎泓钰,任永功. 基于 Redis 内存数据库的快速查找算法[J]. 计算机应用与软件,2016,33(5):40-43.
- [5] LAWRENCE R, BRANDSBERG E, LEE R. Next generation JDBC database drivers for performance, transparent caching, load balancing, and scale-out[C]//Proceedings of the symposium on applied computing. Marrakech, Morocco: ACM, 2017:915-918.
- [6] 张少应,程传旭. 基于 Hibernate 持久化层的设计与实现[J]. 计算机技术与发展,2014,24(12):101-104.
- [7] 韩 兵,江燕敏,方英兰. 基于 JDBC 的数据访问优化技术[J]. 计算机工程与设计,2017,38(8):1991-1996.
- [8] KUMAR V R, SWATI M. Cache replacement algorithms for coordinated cooperative social wireless networks[J]. International Journal of Computer Science and Mobile Computing, 2014,3(10):718-725.
- [9] TERRA R, MIRANDA L F, VALENTE M T. Qualitas. class corpus: a compiled version of the qualitas corpus[J]. ACM SIGSOFT Software Engineering Notes, 2013,38(5):1-4.
- [10] 欧阳宏基,葛 萌,陈 伟. 基于 JDBC 的数据持久化层性能优化研究[J]. 网络新媒体技术,2016,5(5):9-15.
- [11] SCHWALB D, DRESELER M, UFLACKER M, et al. NVC-hashmap: a persistent and concurrent hashmap for non-volatile memories[C]//Proceedings of the 3rd VLDB workshop on in-memory data management and analytics. Kohala Coast, HI, USA: ACM, 2015:4.
- [12] 郑雅洁,张冬雯,张 杨,等. 并行环境下 Java 哈希机制的对比及重构[J]. 河北工业科技,2017,34(6):414-420.
- [13] 王国卿,黄 韬,刘 江,等. 一种基于逗留时间的新型内容中心网络缓存策略[J]. 计算机学报,2015,38(3):472-482.
- [14] 王 准,何元烈. 基于混合价值计算的云存储缓存替换方案[J]. 计算机工程与设计,2017,38(6):1651-1656.
- [15] 母红芬,李 征,霍卫平,等. HashMap 优化及其在列存储数据库查询中的应用[J]. 计算机科学与探索,2016,10(9):1250-1261.