Journal of Computer Applications
计算机应用,2015,35(1):99-102,114
ISSN 1001-9081
CODEN JYIIDU
2015-01-10
http://www.joca.cn

文章编号:1001-9081(2015)01-0099-04
doi:10.11772/j.issn.1001-9081.2015.01.0099

基于 Linux 内核的 Key-Value 存储系统——KStore

谢沛东[1,2*],武延军[1,3]
(1. 中国科学院软件研究所 基础软件国家工程研究中心,北京 100190; 2. 中国科学院大学,北京 100190;
3. 中国科学院软件研究所 计算机科学国家重点实验室,北京 100190)
(*通信作者电子邮箱 peidong@nfs.iscas.ac.cn)

摘 要:Key-Value 存储系统在各种互联网服务中被广泛使用,但现有的 Key-Value 存储系统通常在用户态空间设计和实现,因为频繁的模式切换和上下文切换,导致访问接口、事务处理效率不高,在高并发、低延迟的数据存储需求中尤为突出。针对该问题,给出了一个内核态 Key-Value 存储系统的实现——KStore:提供内核空间的索引和内存分配机制,并在此基础上,通过基于内核 Socket 的远程接口以及基于文件系统的本地接口,保证了 KStore 的低延迟;同时,通过基于内核多线程的并发处理机制,保证了 KStore 的并发性。实验结果表明,与 Memcached 相比,KStore 在实时性和并发性方面都取得显著优势。

关键词:Key-Value 存储系统; Linux 内核; 文件系统; 内核 Socket; 内核线程; Slab 内存分配
中图分类号:TP311.52 **文献标志码**:A

KStore: Linux kernel-based Key-Value store system

XIE Peidong[1,2*], WU Yanjun[1,3]
(1. *National Engineering Research Center of Fundamental Software, Institute of Software, Chinese Academy of Sciences, Beijing* 100190, *China*;
2. *University of Chinese Academy of Sciences, Beijing* 100190, *China*;
3. *State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing* 100190, *China*)

Abstract: Nowadays Key-Value store system is widely used in various Internet services. However, the existing Key-Value store systems, mostly run in user-mode, can not meet the demands of high-concurrency and low-latency. It is mainly because user-mode usually provides inefficient access interfaces and transaction processing due to mode switch or context switch. To solve these problems, an in-kernel implementation of Key-Value store system, called KStore, was proposed in this paper. It had an in-kernel index and an in-kernel memory allocator, which were used to manage Key-Value data efficiently. To guarantee the low-latency response, KStore provided a remote interface based on in-kernel Socket, and a local interface based on file system. In addition, KStore processed concurrent requests with a novel mechanism based on in-kernel multi-thread. The experimental results show that KStore gains a remarkable advantage over Memcached in the characteristics of real-time and concurrency.

Key words: Key-Value store system; Linux kernel; file system; kernel Socket; kernel thread; Slab memory allocation

0 引言

Key-Value 存储系统(以下简称 K/V 系统)的数据模型简单,兼容各种数据类型[1],因此广泛应用于社交网络等各种互联网服务中。通过对其应用场景进行总结,可以发现 K/V 系统一般被用在高并发、对实时性要求高、本地访问的比例高的数据存储场合[2-3]。然而,现有的 K/V 系统并不能满足以上需求,主要体现在实时性和并发性两个方面:

1)现有的 K/V 系统实时性不足。由于服务器端和客户端是相互独立的进程,当客户端访问数据时,需要通过进程间通信(Inter-Process Communication, IPC)机制与服务器端交互。然而,多数 IPC 机制涉及进程上下文切换和数据重复拷贝,增加了通信的延迟。以 Memcached[4] 为例,当客户端存储数据时,需要将数据通过 Socket 传递给内核,内核将其封装为网络报文并发送到网络中,当报文到达服务器端时,其内核将数据解封,并传递给 Memcached 进程。可见,在此过程中,数据经过了多次拷贝,并且客户端和 Memcached 都需要切换上下文,造成额外的开销。

2)现有的 K/V 系统所能处理的并发请求数有限。一些 K/V 系统采用了单线程模式,只能串行地接收请求、处理请求;如 Redis[5] 内部只有一个线程负责处理请求,因此所支持的并发量有限。另一些 K/V 系统采用了多线程模型,但并发控制不高效;如 Memcached 使用锁来保护共享资源(如索引、内存池等),处理大量请求时会造成频繁的加锁和解锁,降低了系统的性能[6]。

对于以上问题,本文认为可以在操作系统内核层面予以解决[7],即在内核中实现 K/V 系统。内核态的 K/V 系统有两个方面的优势:

1)可提供高效的访问接口。由于在内核中,因此其客户端可通过系统调用直接与其通信,无需再通过 IPC 机制中转。

收稿日期:2014-07-16;**修回日期**:2014-08-04。
基金项目:中国科学院战略性科技先导专项(XDA06010600);核高基重大专项(2012ZX01039-004)。
作者简介:谢沛东(1988-),男,山东菏泽人,硕士研究生,主要研究方向:分布式存储系统、操作系统; 武延军(1979-),男,陕西延安人,高级工程师,博士生导师,主要研究方向:云计算、操作系统。

2)可直接调用内核态接口。调用内核态接口时无需上下文切换,相比在用户态调用等价的接口,性能更高;并且,相比用户态接口,内核态接口更加丰富和全面,如可以直接操作内存页、线程,调用网络接口等。

因此,本文设计和实现了一种内核态的 Key-Value 存储系统——KStore。首先,本文实现了 KStore 的核心:索引和内存分配机制,以高效地存储 K/V 数据;在此基础上,为了保证 KStore 的实时性,本文基于系统调用实现了高效的访问接口;最后,为了保证 KStore 的并发性,本文基于多种内核机制实现了并发请求处理机制。

1 系统架构

如图 1 所示,KStore 跨越内核态和用户态,包括内核态的 KStore 服务器和用户态的 KStore 客户端。KStore 服务器端被实现为可动态加载的内核模块(kstore. ko),KStore 客户端被实现为用户态库(libkstore)。

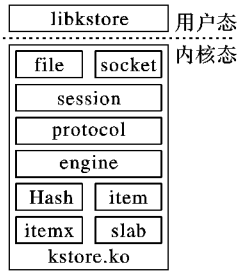

图 1 KStore 系统架构

libkstore 向上层应用程序提供各种常用的 Key-Value 接口,如 Get、Set 等。在内部,libkstore 将来自应用程序的请求封装为相应的报文,通过系统调用发送到 kstore. ko。应用程序无需关注 libkstore 与 kstore. ko 的通信细节,只需调用 libkstore 提供的接口即可透明地访问 KStore 中的数据。

kstore. ko 从上到下分为 3 个层次,分别是请求接收、请求处理、Key-Value 存储引擎。

1.1 请求接收层

该层负责从接口读取 libkstore 传递过来的请求报文,将其分发到请求处理层。为了支持远程访问和本地访问,该层同时实现了基于 Socket 的接口和基于 file 的接口。

1.2 请求处理层

该层可细分为 session 和 protocol。session 层创建多个工作线程,并发地处理上层分发下来的请求;protocol 层向 session 层提供解析协议的接口。

1.3 Key-Value 存储引擎层

该层负责存储数据,可细分为 engine、hash、itemx、item、slab。engine 层实现了处理各种 Key-Value 请求的具体代码,并向上提供 Key-Value 接口,这些接口与 libkstore 中的接口保持一致,被 session 层所调用。itemx 层实现了组织 K/V 数据所需的索引,slab 层实现了存储 K/V 数据所需的内存分配器。

2 关键技术

本文接下来通过介绍 KStore 的 4 项关键技术,以说明 KStore 如何保证实时性和并发性。

2.1 访问接口

本文针对远程访问的场合,设计了基于 Socket 的通用访问接口,并且,考虑到在分布式计算等应用中,服务器端与客户端通常位于同一节点,本地访问相比远程访问更为频繁[3],因此本文又设计了基于文件系统的接口,以提升本地访问的性能。

远程访问接口基于内核态 Socket 实现。KStore 开启内核态 Socket 监听来自 Client 的连接,当接收到连接后,交由工作线程,之后的数据传输和处理均由工作线程完成。为了提高效率,KStore 并不会阻塞监听,而是采用异步 I/O 的方式监听请求。KStore 为 Socket 注册了 sk_state_change 等回调函数,在其中实现了接收和分发连接的逻辑,当 Socket 状态发生改变时,回调函数会被中断处理例程调用,因此 KStore 无需阻塞在监听,仍能在连接到来时立即对其进行处理。

为了便于客户端与 KStore 交互,本文实现了用户态的 Key-Value 接口库 libkstore。如图 2 所示,libkstore 在标准 Socket 基础上,封装了 5 种常用的 Key-Value 接口函数(Get, Set, Add, Replace, Delete),libkstore 在内部通过本文自定义的协议与 KStore 通信。

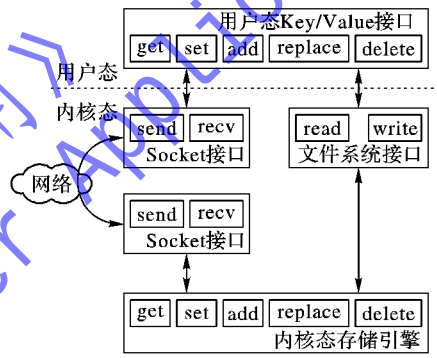

图 2 KStore 远程、本地访问接口

KStore 的远程访问接口基于 Socket 实现,无论在远程还是在本地,都可以通过该接口访问 KStore。然而,在本地使用该接口时性能并不高,因为数据会经过网络协议栈,涉及网络包的封装和解封,实际上本地访问时,客户端和 KStore 并没有进行真正的网络通信,网络包只在内核中周转,造成了无谓的开销。为此,应该开辟一个通道,使得用户态的客户端能够与内核态的 KStore 直接通信,以提高本地访问的性能。

针对本地访问,KStore 以系统调用的形式提供接口给客户端。为了保证可移植性和部署的便利性,KStore 没有在内核中增加新的系统调用,而是复用内核中已有的文件系统接口。KStore 实现了虚拟文件系统(Virtual File System, VFS)层[8]中定义的文件读写接口 kstore_read/kstore_write,当系统调用 read/write 被调用时,最终会跳转到 KStore 中的 kstore_read/kstore_write 上执行。在其中,实现了请求的接收和分发,请求的处理则交由工作线程完成。如图 2 所示,与远程访问接口类似,本文在 read/write 基础上封装了 5 种 Key-Value 接口,并集成到 libkstore 中。

本地访问接口与远程访问接口提供相同的接口,且采用相同的协议与 KStore 通信,只是在传输方式上不同。通过本地访问接口访问 KStore 时,无需经过网络协议栈,因此效率比远程访问接口高。

2.2 并发请求处理

为了高效地处理大量请求,本文实现了一种并发处理请求的机制。

首先,充分利用多核 CPU。KStore 创建多个内核态线程

(Worker),以并行地处理大量请求。同时,为了避免过多线程竞争 CPU 资源,造成频繁的线程切换和 Cache 失效,KStore 将线程的数量限制为 CPU 核数,并分别绑定到不同的核之上[9],以提高多核 CPU 的利用率。

传统的 K/V 系统中,通常会依次处理每个请求,直到上一个请求处理结束才会处理下一个请求。Worker 线程在处理某些请求时,可能会因为 I/O 而阻塞,导致当前请求和其他请求都得不到处理,Worker 线程实际上处于闲置状态。

为了解决该问题,本文借鉴进程分时执行的思想。如图 3 所示,将各种请求(Request)的处理函数拆分成多个 Step 函数,拆分时保证耦合性强的代码位于同一 Step 中,这些函数会被依次执行。在涉及 I/O 等可能阻塞的地方,全部使用非阻塞调用,以保证 Worker 线程一直处于运行状态。为了实现分时处理,Worker 每次从请求队列中取一个请求,执行该请求当前的 Step 函数,然后将请求放回队列,并重新从队列中取下一个请求来处理。

可见,处理请求的粒度被细化到 Step,并采用分时机制,循环处理各请求,降低了被个别请求阻塞的概率,提高了线程的利用率。

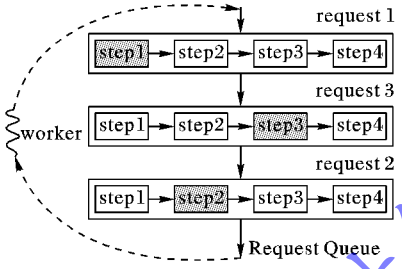

图 3　分时处理请求

2.3　索引

为了应对大量数据的查询和插入,需要设计一种索引,将大量 K/V 数据高效地组织起来。常用的索引存在一些不足,如:Hash 表难以确定表的初始大小;SkipList[10] 维护复杂,在插入和删除节点时,需要修改多个相关节点。

本文设计了区别于传统 Hash 表的索引,即 3 级 Hash 表。如图 4 所示,类似于操作系统中的页表,3 级 Hash 由 3 个级别的 Hash 表构成,上一级 Hash 表中保存下一级 Hash 表的首地址,最后一级 Hash 表则保存 Item 链表的指针。当利用该索引进行查询时,将 Key 划分成 3 个 SubKey,每一级 SubKey 用于在对应的 Hash 表中查找下一级 Hash 表,最终找到目标 Item。

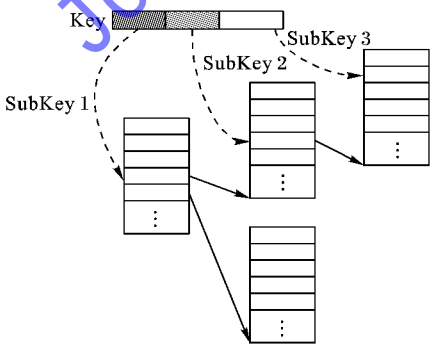

图 4　3 级 Hash 表

查询每一级 Hash 表时,都只需一步计算即可定位到目标索引项,时间复杂度是 $O(1)$,在最后一级 Hash 表中定位到 Item 链表后,需要遍历该链表,而 Item 链表的长度是常数级的,因此时间复杂度仍是 $O(1)$。与查询一样,添加和删除 Item时,3 级 Hash 表仍具备 $O(1)$的时间复杂度。

3 级 Hash 表采用写时分配的策略,即只有当相应的 Hash 表第一次插入 Item 时,才会为其分配内存,因此不会占用过多的内存。KStore 也会定期地扫描 Hash 表,以删除空白 Hash 表。可见,3 级 Hash 表所占的内存量始终与 Item 数量保持一致,提高了内存的利用率。

3 级 Hash 表继承了传统 Hash 表查询快速、维护简单的优点,克服了传统 Hash 表初始大小难定、数据量大时性能显著下降的缺点。

2.4　内存分配

为了在内存中存储 K/V 数据,需要设计一种高效的内存分配机制。

本文在内核 Slab[11] 基础上实现了面向不定长 K/V 数据的内存分配机制。初始化时,依次创建 N 个以 2^i B(其中 i 从 4 到 $N+3$)为粒度的内存池(本文称为 Bucket),以应对各种可能尺寸的数据。之后分配内存时,通过简单的计算,可得到最匹配数据长度的 Bucket,如对于 500 B 的 K/V 数据,512 B 的内存块最匹配,而对应的 Bucket 是 Bucket[5],定位到 Bucket 之后,取其队尾指针指向的内存块,即可分配给用户。

在实际运行过程中,发现内存的有效利用率很低,如存储 500 MB的数据,却要消耗 1 GB 的内存。经过分析,发现大量小尺寸 K/V 数据造成了过多的内存碎片。如图 5(a)所示,虽然每个 K/V 数据的尺寸小,但仍需占用一块内存空间,如 513 B的 K/V 数据需占用 1 024 B 的空间,从而浪费了511 B的内存,当存储了大量这种 K/V 数据时,累积浪费的内存就会很可观。

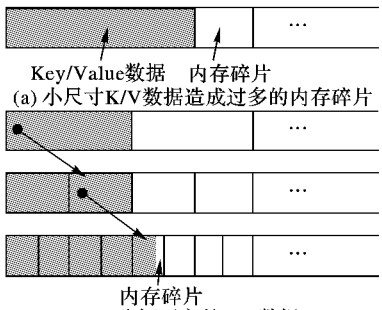

图 5　分解数据以减少内存碎片

根本原因是,本文所创建的一系列内存池,其分配粒度呈指数级递增,跨度较大,不适合为不定长 K/V 数据的分配内存。本文采用分治法解决该问题,将不定长数据分解为定长数据,对于每个分解后的定长数据,分别使用内存池为其分配内存。由于定长数据能够有效利用内存块,因此不会产生大量内存碎片。如图 5(b)所示,在分解不定长 K/V 数据时,按照 $2^i + 2^{i-1} + 2^{i-2} + 2^{i-3} + \cdots$ 的形式分解为多个定长片段,并依次为每个片段查找合适的内存块。在性能方面,由于每次查找的时间复杂度都是 $O(1)$,且 Bucket 的总量有限,因此总的时间复杂度仍然是 $O(1)$。由于分解后的片段离散地存储在不同的 Bucket 中,为了之后能够快速还原 K/V 数据,本文在每个内存块头部添加了额外的链表指针,用于将属于同一个 K/V 数据的片段连接起来。

3 实验验证

本文设计了两组实验分别对 KStore 的实时性和并发性进行测试。其中:实时性测试侧重验证 KStore 响应客户端请求的低延迟;并发性测试侧重验证 KStore 并发处理大量请求的能力。

本文选取 Memcached 与 KStore 进行对比测试。Memcached 和 KStore 作为服务器端,相应的客户端为 Memcached 的标准测试工具 Memslap。为了排除网络情况等外围因素的干扰,本文在单机系统上进行测试,单机的配置为:24 核 CPU,64 GB 内存,Ubuntu Server 12.04,Linux Kernel 版本为3.8。

3.1 实时性测试

本实验验证 KStore 的平均响应时间低于 Memcached。首先,利用单个客户端,向 KStore 连续发起一批写请求(Set),等待 KStore 响应,统计响应时间;然后,逐渐增加每批请求中的请求数量,统计响应时间在请求数量逐渐增大时的变化情况。对于 Memcached,采用相同的方式进行测试。

由于 KStore 和 Memcached 都有多种访问接口,KStore 提供基于文件系统的接口(KStore-Local)、基于 TCP Socket 的接口(KStore-TCP),Memcached 提供了基于 UNIX Socket 的接口(Memcached-Local)、基于 TCP Socket 的接口(Memcached-TCP)。为了保证测试的全面性,本实验分别对以上 4 种接口进行测试。

图 6(a)为实验结果,记录的是随着客户端发起的请求数量的增加,服务器端处理这些请求所需时间(即响应时间)的变化。

从结果可以看出,KStore 的响应时间都显著低于 Memcached。其中,KStore-Local 的响应时间最低,即使在处理 10 万个请求时,只消耗了约 2 s 的时间。之所以有如此大的差距,主要是由于 KStore 运行在内核态,客户端可以直接访问 KStore 中的数据,相比需要经过 IPC 机制才能访问的 Memcached,数据拷贝以及状态切换的次数更少。实验结果证明了基于内核的存储系统的有效性,并且基于 TCP Socket 的接口比本地接口的性能差,如 KStore-TCP 的响应时间是 KStore-Local 的 10 倍以上,这是由于网络协议栈引入了过多额外的开销,而 KStore 本地接口则没有这些开销。总之,无论通过本地接口还是远程接口访问 KStore,响应时间都比 Memcached 低,与预期的效果一致。

3.2 并发性测试

本实验的目的是验证 KStore 的并发处理能力优于 Memcached。为此,利用多个客户端,并发地向 KStore 发起写请求(Set),统计单位时间内 KStore 所能够处理的 TPS(Transactions Per Second),即每秒事务数(或每秒请求数),并逐渐增加客户端的数量,以统计 TPS 的变化趋势。

对于 Memcached,由于可以通过参数调整内置线程数,为了保证测试的公平性,本测试会逐渐增加 Memcached 的线程数,以期测出 Memcached 最优情况下的并发处理能力。

图 6(b)为实验结果,展示了随着客户端数量的增多,KStore 和 Memcached 的 TPS 的变化。其中横轴为客户端(Memslap)的数量,纵轴为 KStore/Memcached 的每毫秒所处理的并发请求数。

从结果可以看出,KStore 的 TPS 始终高于 Memcached,即使与最优情况下的 Memcached(4 线程)相比,也有近 2 倍的优势。随着客户端数量的增多,KStore 和 Memcached 的 TPS 都有下降的趋势,这是由于 KStore/Memcached 的性能已经处于饱和状态(CPU 使用率 100%),随着客户端的增多,内部线程的竞争更加激烈,导致并发性下降。从结果还可以看出,4 线程 Memcached 的性能超过了 8 线程 Memcached,可见 Memcached 在设计时针对 4 线程进行了优化[12]。总之,实验结果表明 KStore 的并发性优于 Memcached,证明本文设计的并发处理机制具有一定的有效性。

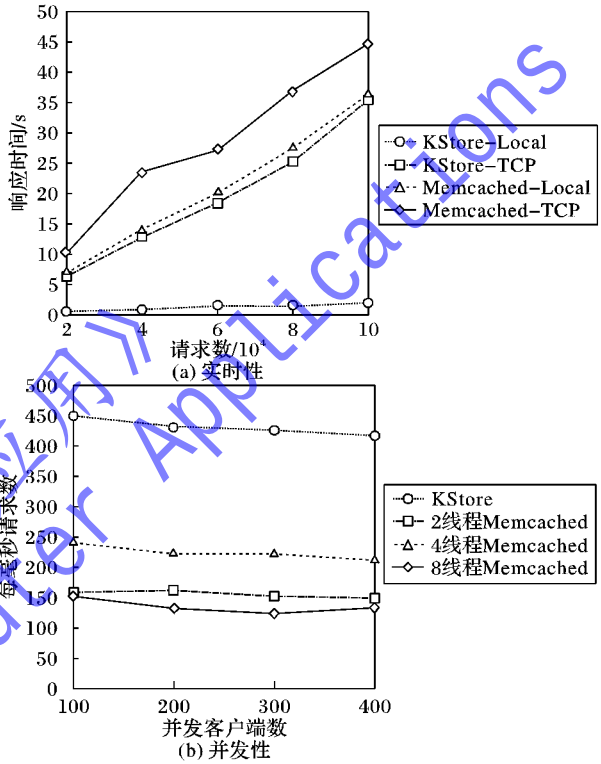

图 6 实时性、并发性测试结果

4 结语

为了解决现有 K/V 系统实时性和并发性不足的问题,本文认为将 K/V 系统放入内核是解决问题的有效途径,设计和实现了内核态 Key-Value 存储系统——KStore。首先,介绍了 KStore 的系统架构,包括内核态的 Kstore 模块和用户态的 libkstore,对其结构和功能进行了阐述;然后,详细阐述了 KStore 解决实时性和并发性问题的关键技术,即访问接口、并发请求处理、索引、内存分配;最后,通过多项实验测试了 KStore 的性能。实验结果表明,与 Memcached 相比,KStore 在实时性和并发性方面都取得显著优势。本文下一步将研究如何将 KStore 应用在分布式计算中,以提高迭代计算、流计算的性能。

参考文献:

[1] SADALAGE P J, FOWLER M. NoSQL distilled: a brief guide to the emerging world of polyglot persistence [M]. Boston: Addison Wesley, 2012: 31 - 33.

[2] NISHATALA R, FUGAL H, GRIMM S, *et al*. Scaling Memcache at Facebook [C]// NSDI'13: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation. Berkeley: USENIX Association, 2013: 385 - 398.

(下转第 114 页)

准确性上就差了一些。

表 7 Top-3 统计结果 2

算法	查询结果 1			查询结果 2			查询结果 3		
	key	支配分数	属性（属性 1，属性 2）	*key*	支配分数	属性（属性 1，属性 2）	*key*	支配分数	属性（属性 1，属性 2）
Ranking-*k*	C	9	(10,10)	G	7	(8,9)	A	6	(7,8)
BSA	C	9	(10,10)	G	7	(8,9)	A	6	(7,8)
DA	C	9	(10,10)	G	7	(8,9)	A	6	(7,8)
TDEP	C	9	(10,10)	G	7	(8,9)	A	6	(7,8)

5 结语

本文在小规模数据上执行任意 skyline 准则的 Top-*k* dominating 查询问题，提出一种新的 Top-*k* dominating 查询算法——Ranking-*k* 查询算法。该算法为每个属性构建有序列表，可以有效减少查询涉及到的 I/O 消耗，能有效地返回查询结果。为提高任意 skyline 准则的 Top-*k* 支配查询速度，本文主要从两个方面优化和设计：1）采用统计学模型判断候选元组在未扫描的属性上出现的概率，提高了计算终结元组支配分数准确性。2）提高求解终结元组的准确支配分数的速度，提出一种有效的准确支配分数的计算方法，利用已经获得的候选元组的信息和扫描信息就可获得结果。实验结果表明，Ranking-*k* 查询算法与现有算法相比具有较好的查询性能。下一步主要的工作是在海量数据上执行任意 skyline 准则的 Top-*k* dominating 查询，通过减少候选元组的数量，降低 round-robin 方式扫描深度来提高查询效率。

参考文献：

[1] FAGIN R, LOTEM A, NAOR M. Optimal aggregation algorithms for middleware [C]// PODS'01: Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. New York: ACM, 2001: 102 – 113.

[2] VAGELIS H, NICK K, PAPAKONSTANTINOU Y. PREFER: a system for the efficient execution of multi-parametric ranked queries [C]// Proceeding of the 2001 ACM SIGMOD International Conference on Management of Data. New York: ACM, 2001: 259 – 270.

[3] BORZSONYI S, KOSSMANN D, STOCKER K. The skyline operator [C]// Proceedings of the 17th International Conference on Data Engineering. Piscataway: IEEE, 2001: 421 – 430.

[4] PAPADIAS D, TAO Y, FU G, *et al.* Progressive skyline computation in database systems [J]. ACM Transactions on Database Systems, 2005, 30(1): 41 – 82.

[5] YIU M, MAMOULIS N. Efficient processing of Top-*k* dominating queries on multi-dimensional data [C]// VLDB'07: Proceedings of the 33rd International Conference on Very Large Data Bases. [S. l.]: VLDB Endowment, 2007: 483 – 494.

[6] YIU M, MAMOULIS N. Multi-dimensional Top-*k* dominating queries [J]. The International Journal on Very Large Data Bases, 2009, 18(3): 695 – 718.

[7] KONTAKI M, PAPADOPOULOS Y. Continuous Top-*k* dominating queries [J]. IEEE Transactions on Knowledge and Data Engineering, 2010, 24(5): 840 – 853.

[8] ZHANG W, LIN X, ZHANG Y, *et al.* Threshold-based probabilistic Top-*k* dominating queries [J]. The International Journal on Very Large Data Bases, 2010, 19(2): 283 – 305.

[9] TIAKAS E, PAPADOPOULOS A N, MANOLOPOULOS Y. Progressive processing of subspace dominating queries [J]. The International Journal on Very Large Data Bases, 2011, 20(6): 921 – 948.

[10] HAN X, YANG D, LI J. An efficient Top-*k* dominating algorithm on massive data title [J]. Chinese Journal of Computers, 2013, 33(8): 1405 – 1417. (韩希先, 杨东华, 李建中. TKEP: 海量数据上一种有效的 Top-*k* 查询处理算法[J]. 计算机学报, 2013, 33(8): 1405 – 1417.)

[11] TAO Y, XIAO X, PEI J. SUBSKY: efficient computation of skylines in subspaces [C]// Proceedings of the 22nd International Conference on Data Engineering. Piscataway: IEEE, 2006: 65 – 76.

[12] XIA T, ZHANG D. Refreshing the sky: the compressed skycube with efficient support for frequent updates [C]// Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. New York: ACM, 2006: 491 – 502.

[13] YUAN Y, LIN X, LIU Q, *et al.* Efficient computation of the skyline cube [C]// Proceedings of the 31st International Conference on Very Large Data Bases. [S. l.]: VLDB Endowment, 2005: 241 – 252.

[14] TAO Y, XIAO X, PEI J. Efficient skyline and Top-*k* retrieval in subspaces [J]. IEEE Transactions on Knowledge and Data Engineering, 2007, 19(8): 1072 – 1088.

[15] ZHANG Z, LU H, OOI C, *et al.* Understanding the meaning of a shifted sky: a general framework on extending skyline query [J]. The International Journal on Very Large Data Bases, 2010, 19(2): 181 – 201.

（上接第 102 页）

[3] ZHANG S, HAN J, LIU Z, *et al.* Accelerating MapReduce with distributed memory cache [C]// Proceedings of the 15th International Conference on Parallel and Distributed Systems. Piscataway: IEEE, 2009: 472 – 478.

[4] DORMANDO. Memcached [EB/OL]. [2014-06-20]. http://memcached.org/.

[5] VMWARE. Redis [EB/OL]. [2014-06-10]. http://redis.io/.

[6] MICHAEL M M. High performance dynamic lock-free hash tables and list-based sets [C]// Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures. New York: ACM, 2002: 73 – 82.

[7] ANDREWS B. In-Kernel Berkeley DB [EB/OL]. [2014-06-15]. http://www.fsl.cs.sunysb.edu/project-kbdb.html.

[8] KLEIMAN S R. Vnodes: an architecture for multiple file system types in Sun UNIX [C]// Proceedings of the 1986 Summer USENIX Conference. Berkeley: USENIX Association, 1986: 238 – 247.

[9] LOVE R. Kernel korner: CPU affinity [J]. Linux Journal, 2003, 2003(111): 8.

[10] PUGH W. Skip lists: a probabilistic alternative to balanced trees [J]. Communications of the ACM, 1990, 33(6): 668 – 676.

[11] BONWICK J. The slab allocator: an object-caching kernel memory allocator [C]// USTC'94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference. Berkeley: USENIX Association, 1994: 6 – 6.

[12] WIGGINS A, LAGNSTON J. Enhancing the scalability of Memcached [EB/OL]. [2014-06-05]. https://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached-0.