内存实时快照共享机制及其在数据库系统中的应用

孟庆钟[1),2)] 周 烜[3)] 王 珊[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)

[3)](华东师范大学数据科学与工程学院 上海 200062)

摘 要 传统的数据分析程序需要从数据库系统中获取数据,然后对这些数据做进一步分析.虽然内存数据库技术发展迅速,但这种传统的架构依然没有改变.与 CPU 的处理速度相比,数据从内存数据库中传递到数据分析程序的速度仍然很慢.导致这个问题的原因之一是现代操作系统对进程间通信方式的支持程度不足.该文作者在 Linux 操作系统内核中实现了一种新的进程间通信方式,命名为 SWING.通过调用一个新增加的系统调用,一个进程就可以使用 SWING 实时的把自己的物理内存页面共享给其它进程.作者基于 SWING 做了一个内存分配器,称作 SwingMalloc,并且基于 SwingMalloc 开发出一个新的嵌入式内存数据库系统,命名为 SwingDB.使用该系统,应用程序可以在自己的进程空间中访问整个数据库中的数据,而不再受限于传统的进程间通信方式.

关键词 内存数据库;进程间通信;SWING;操作系统;Linux 内核

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2018.01912

Memory Instant Snapshot Sharing Mechanism and Its Application in Database

MENG Qing-Zhong[1),2)] ZHOU Xuan[3)] WANG Shan[1),2)]

[1)](*Key Laboratory of Data Engineering and Knowledge Engineering of Ministry of Education (Renmin University of China), Beijing* 100872)

[2)](*School of Information, Renmin University of China, Beijing* 100872)

[3)](*School of Data Science & Engineering, East China Normal University, Shanghai* 200062)

Abstract As the capacity of RAM is growing exponentially, RAM becomes an important instrument for big data processing. When possible, we prefer to store an entire dataset in RAM and conduct in-memory computing and data analytics. Such an approach can speedup business processes substantially. Data analytical programs need to retrieve data from a database management system before conducting analysis on the data. Apart from that, a common data analytical application usually involves a number of stages, which cooperate seamlessly to generate data analysis results. These stages often need to exchange large volume of data. Despite the prevalence of In-Memory Computing in data analytics, the traditional data transmission architecture from database to data analytical program has not changed and it becomes a performance bottleneck in the context of in-memory data analytics. One of the key reasons for this bottleneck is that the IPC (Inter-Process Communication) support of modern operating systems is inadequate. Pipe and Socket are slow. While shared-memory is fast, managing shared-memory is difficult, as we have to deal with memory allocation and data synchronization carefully. Some approaches, such as SAP HANA, try to avoid inter-process data exchange by injecting data processing programs into

收稿日期:2017-01-18;在线出版日期:2017-11-15.本课题得到国家"八六三"高技术研究发展计划项目基金(2015AA015307)、国家自然科学基金(61772202)资助.**孟庆钟**,男,1988 年生,博士研究生,研究方向为内存数据库、操作系统.E-mail:mqz@ruc.edu.cn.**周 烜**(通信作者),男,1979 年生,博士,教授,博士生导师,研究领域为高性能数据库及信息检索.E-mail:zhou.xuan@outlook.com.**王 珊**,女,1944 年生,教授,博士生导师,中国计算机学会(CCF)会士,研究领域为高性能数据库、知识工程、数据仓库、大数据.

the process space of DBMS. However, such a tight coupling approach does not suit all applications. We implemented a new IPC method named SWING in the Linux kernel. It is fast and convenient. It enables loose coupling between data processing programs. With SWING, any processes can share a segment of memory, and all of them can read the same contents. When one of them wants to write, operating system will apply copy-on-write for that process, so the write behavior will not affect other processes. This method is similar with fork, but it works for processes out of parent-child relation—in effect, more than one process can share multiple segments of memory to the same process, which cannot be achieved by fork. Based on SWING, we developed a memory allocator named SwingMalloc which makes SWING easy to use. SWING allocates a virtual memory space of 512 GB each time it is called, so it may waste a lot of logical space. SwingMalloc allows for fine grained space allocation, so it is more friendly to processes. Basically, SwingMalloc divides a COW memory into two parts. One part is managed with the buddy memory allocation algorithm. The other part is broken down into several blocks of fixed but different sizes. These blocks are allocated to processes based on their need. Based on SwingMalloc, we developed a new in-memory embedded DBMS called SwingDB. With SwingDB, each process accesses a database in its own memory space, without incurring inter-process communication. The data in an instance of SwingDB is completely stored in a Swing memory area, so that independent processes can share the snapshots of their database instances using the SWING mechanism. SwingDB is especially suitable to multi-stage in-memory data processing, in which several loosely coupled programs cooperate in performing data analysis. These applications can access the entire database in their own memory space, instead of resorting to expensive traditional IPC methods.

Keywords　in-memory database; inter-process communication; SWING; operating system; Linux kernel

《计算机学报》

1　引　言

内存的速度比磁盘快 4～5 个数量级. 随着内存容量的快速增长以及价格的下降，内存逐渐成为数据处理系统的重要资源. 在内存计算中，人们希望数据全部驻留内存，因为这可以极大地提高数据访问和数据处理的速度. 大多数数据分析处理过程都分为多个阶段，由一系列数据处理程序共同构成，这些程序相互协调，共同完成数据分析任务. 在一个具体的分析过程中，每个程序都将自己产生的结果传递给下一个程序，形成数据流. 比如，一个典型的数据分析处理情景如下：数据存储在数据库管理系统(DBMS)中，第一级数据分析程序向 DBMS 发送查询语句(比如 SQL)并获得自己需要的数据；然后它对这些数据进行统计分析，并把分析结果传递给下一级的分析程序；以此类推；分析过程结束后，它把最终结果传递给数据可视化进程，可视化进程把结果展示给用户. 在数据分析的过程中，各个程序之间通常需要交换大量的数据. 如果数据在进程之间的传输速度不够快，很难发挥内存计算的性能.

根据作者的研究，现代操作系统所提供的进程间通信方式不能很好的满足内存计算性能的需求. 目前，进程间交换大量数据时可以选用的进程间通信方式有管道、共享内存、Socket 等. 利用管道和 Socket 时，需要对数据进行多次物理层面的拷贝，并且可能会阻塞相关的进程，它们两个的传输速度很慢. 如果使用共享内存，虽然可以避免数据在物理层面的拷贝，但是仍然有三个问题：第一，需要其它同步机制来控制对共享内存的并发访问，这将带来额外的开销；第二，当多个进程同时使用同一块共享内存时，它们之间的代码耦合度将变得很强，这将给软件的开发和维护带来更多困难；第三，如果允许所有的程序通过共享内存的方式直接操作整个 DBMS 的内存，也会有安全问题，如某些程序的 BUG 会破坏整个数据库中的数据.

为了提高数据在进程之间传输的速度,同时降低程序的代码耦合度,本文提出了一种新的、基于写时复制的进程间通信方式,命名为 SWING. SWING 具有简单、易用、高效的特点. 使用 SWING 进行通信,既能避免对数据进行物理复制,又能保持程序之间比较松的耦合度. 这非常符合现代软件设计的原则,也能充分发挥内存计算的性能优势.

SWING 通过共享物理内存页面的方式实现进程间通信. 程序使用 SWING 方法申请的内存称作 COW 内存,它是具有下列性质的一段虚拟内存:

(1) 两个(或多个)COW 内存可以被映射到相同的物理内存页面集合中.

(2) 修改不同的 COW 内存(共享相同的物理内存页面)时,采用写时复制技术进行隔离.

进程之间共享数据时,利用第一个特点,可以避免数据在物理内存上进行复制. 利用第二个特点可以保证每个 COW 内存之间互不干扰.

进程 A 想把数据共享给进程 B 时,进程 A 需要先把数据放在一块 COW 内存中(一旦进程 A 把数据写入这块 COW 内存,操作系统就已经为此 COW 内存分配了物理内存页面),然后进程 B 分配一块 COW 内存,并把这块 COW 内存映射到相同的物理页面上. 整个过程不需要物理内存页面的复制,而两个进程可以在它们各自的 COW 内存中读取到相同的数据. 此后,这两个进程都可以独立的修改自己 COW 内存中的内容. 写时复制机制可以保证它们各自的修改对另外一方不可见,所以不需要额外的并发控制机制保证数据的一致性.

每次使用原始的 SWING 方法获取一块内存时,这块内存的大小至少是 512 GB,这是因为 SWING 方法为了提高共享速度,进程之间共享页表. 但这样大块的内存,并不方便程序的使用,因此我们在 SWING 之上做了一个内存分配器,称为 SwingMalloc,它的作用和 C 标准库中的 malloc 函数功能相似,并且都实现在用户空间.

基于 SwingMalloc,我们开发了一款新的嵌入式内存数据库系统,命名为 SwingDB. 应用程序嵌入 SwingDB 之后,就可以在自己的内存空间中操作整个数据库,而不需要额外的进程间通信机制. 每一个 SwingDB 的数据库实例都被放在一个 COW 内存中,不同程序之间可以使用 SWING 方法共享整个数据库快照,实现实时的数据共享. SwingDB 非常适合多阶段的内存数据分析,它让几个耦合度低的程序能够相互协作,完成整个数据分析的过程.

我们在 Linux 内核中实现了 SWING①. 基于 SWING,在用户空间实现了 SwingMalloc. 又基于 Supersonic② 和 SwingMalloc 实现了 SwingDB. 本文通过大量实验测试了 SWING、SwingMalloc 和 SwingDB 的性能,同时把 SwingDB 和传统的内存数据库做了对比,从而证明了它们对内存数据分析应用的适用性. 本文的部分工作已经在 2016 年的 ADMS Workshop 上发表[1]. 本文在此基础上,增加了 SwingMalloc 这个功能模块,对 SWING 的实现进行了详细的描述,并且对 SwingDB 做了进一步的实验评估,得到了更丰富的实验结果.

本文第 2 节介绍相关工作;第 3 节介绍 Swing 的设计和实现;第 4 节介绍 SwingMalloc 的设计和实现;第 5 节介绍和讨论 SwingDB 的实现和潜在应用场景;第 6 节展示我们对系统进行性能测试的结果;第 7 节总结全文,并讨论未来的研究计划.

2 相关工作

对数据密集型的应用来说,移动数据的代价很大,但这又经常在 DBMS 和数据分析进程之间发生. 尤其当需要在大量的数据上做统计分析时,我们需要从 DBMS 向数据分析工具(比如 R、SAS 等)传输大量的数据. 在内存计算中,这样的数据传输代价很大,会直接影响性能.

2.1 内存数据库

以前的研究者们认为把程序放到数据中会比把数据放在程序中要快. 截止目前,有很多数据库系统集成了一些数据分析、数据挖掘工具[2]和专业的应用程序,用来在数据库系统的内部对数据进行分析处理. 有一些内存数据库系统甚至把数据库服务器和应用程序服务器合并到一起,用来减小数据移动的代价[3-4]. 但是数据库系统代码和数据分析系统代码之间过度耦合是一把双刃剑. 虽然这可以减小通信的代价,但也增加了软件开发和维护的成本[5-6]. 在软件工程中,关注点分离是一个核心的原则. 大多数情况下,DBMS 的开发者并不详细了解数据分析算法和数据分析程序的需求,这些应用程序的开发者对数据库系统的了解也并不深入. SwingDB 的目标就是减小数据交换的代价,同时尽可能减小 DBMS 和数据分析程序之间的代码耦合度.

① http://swinglinux.github.io/swing/
② https://github.com/google/supersonic

Hyper[7]是与 SwingDB 最相似的内存数据库系统. Hyper 被设计成同时处理 OLTP 和 OLAP 的系统. Hyper 的主进程负责整个数据库系统的完整性，它来完成所有的更新操作. 每当一个分析请求到来时，主进程都需要调用 fork 产生一个子进程，子进程对请求进行处理. 由于子进程可以和主进程共享所有的物理内存，所以子进程可以立即得到主进程的一个内存快照，子进程利用这个内存快照完成一系列的操作. 主进程和子进程之间的隔离通过操作系统提供的写时复制来保证. 根据我们的调查，SAP HANA[3-4]同样利用 fork 在 DBMS 和数据分析程序之间共享数据. 虽然 fork 可以成功地避免数据移动，但是它使用起来并不方便. 首先，如果使用 fork 的方法，就需要把数据分析程序嵌入到 DBMS 中(比如用动态链接库)，这将使得数据分析程序的开发过程变得很复杂. 更重要的是，因为一个子进程不可能有多个父进程，所以使用这种方式不能同时从多个进程得到内存快照，也就不支持需要把多个进程中的内存快照作为数据源的数据分析程序. 但使用 SwingDB 时，一个进程可以得到多个进程的内存快照. 对比发现，SwingDB 比 fork 更加易用.

最近也有一些技术用写时复制在内存数据库上实现并发[8-10]. 由于 SwingDB 和这些技术的使用场景不相同，所以它们之间并不相关.

2.2　进程间通信

现代操作系统中，最广泛使用的进程间通信方式有管道、命名管道、套接字和共享内存等. 传输数据时，管道和套接字都需要复制物理内存页面. 而且，它们都先把数据从数据源复制到缓存，再从缓存复制到目的地. 有时会存在多层缓存，数据会被复制多遍[11]. 在内存计算中，不停的复制物理内存页面代价很大.

共享内存是目前在进程间共享数据最快的方式，因为它不需要复制物理内存页面. 一般情况下，操作系统会提供两种使用共享内存的方式. 在共享模式下，任何进程对共享内存的更新操作，都会被其它进程立即看到. 在私有模式下，一个进程对共享内存的更新操作，只能被它自己和它的子进程看到，一旦它需要更新共享内存，操作系统立即会利用写时复制的方式保证写入操作不被其它进程看到.

考虑这种场景：主进程以共享模式使用共享内存，数据分析进程以私有模式使用共享内存. 这种场景看起来似乎是安全的，其实不然. 在这种情况下，主进程对共享内存的任何写入操作，都会被分析进程立即看到，从而导致分析进程可能会看到脏数据. 所以还需要额外的同步机制来保证主进程不会更新分析进程所看到的快照. 在 SwingDB 中，一旦对数据进行了共享，主进程的任何更新操作都不会被分析进程看到，因此不再需要额外的同步机制. 使用私有模式的共享内存时，分析进程的分析结果不能再共享给其它进程. 有时数据分析流分为多个阶段. 例如，在数据分析的早期阶段会产生一些中间结果，这些中间结果会被传递到数据分析的后续阶段. 这种多阶段的数据处理在共享内存上很难高效地实现. 然而，SwingDB 允许进一步的把数据共享给其它进程，非常适合多阶段的数据分析.

3　SWING 方法

3.1　数据共享模型

SWING 的数据共享模型如图 1 所示，进程 A 和进程 B 是两个应用程序，它们都可以把自己的数据共享给进程 C，进程 C 把两份数据进行整合并做一些其它的准备工作. 然后，进程 C 把它处理好的数据共享给进程 D 和进程 E，它们负责数据的分析. 理论上，这样的数据传递过程可以无限的进行下去. 经过数据共享之后，所有的进程都独立地在自己的数据拷贝上进行操作，它们对数据所做的修改彼此都不可见.

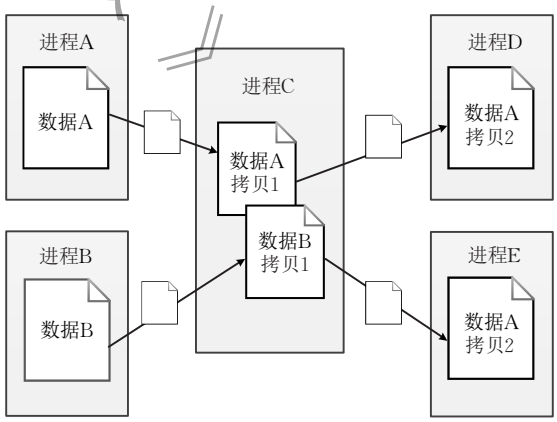


图 1　SWING 的数据共享模型

用管道和 socket 同样可以实现相同的效果. 与这些方法相比，SWING 不需要复制物理内存页面. 它仅仅把各个进程中相应的虚拟内存地址映射到相同的物理页面上. 真正的复制操作被推迟到有进程需要对这块内存进行写入的时候. 一旦有进程需要写入，写时复制就会发生，操作系统会分配一块新

的物理内存.在典型的数据处理场景下,写入操作比读取操作少的多.因此,数据复制带来的开销可以用SWING降到最低.

通过使用SWING方法,多个进程之间可以用传递快照的方式共享数据,并且可以保持较低的耦合度.这些进程仅仅需要共用解析数据的代码,不需要共用任何控制代码,比如保持数据一致性的代码.根据软件工程的原则[12],数据耦合比代码耦合更加灵活.如果使用共享内存,这些进程需要共享一些代码用来处理并发.

这个数据共享模型不能通过 *fork* 实现,因为进程C的父进程不可能同时是进程A和进程B.

3.2 COW内存

在SWING中,用来传输数据的内存空间被称为COW(Copy-On-Write)内存.一块COW内存是进程中一段连续的虚拟内存.COW内存需要映射到物理内存页面之后,才能被程序存取数据.不同的COW内存可以被映射到相同的一组物理页面上.如果不同的COW内存映射到相同的物理页面,则对这块物理内存的写操作将导致写时复制.

3.3 接 口

我们通过实现4个新的系统调用来实现SWING方法.

(1) *long createarea*(*long length*).一个进程使用这个系统调用申请一块新的COW内存.输入参数 *length* 用来指定需要申请的COW内存的大小.返回值(在x86_64平台上占64位)包括两部分:低12位作为一个 *token*,高52位指向这段COW内存的起始地址.在整个系统中,任何两个 *token* 的值都不同,每个 *token* 用来代表一块COW内存,它与文件系统中的文件描述符类似.一般情况下,内存的页面的大小是4KB,所以COW内存最少4KB对齐,但由于共享页表的原因,实际每块COW内存都是512GB对齐,也正是因为这个原因,我们又做了SwingMalloc.进程获取COW内存之后,并没有获得真正的物理内存.当进程向这段COW内存写入内容时,操作系统才会给进程分配相应的物理内存页面.

(2) *long hook*(*int token*).一个进程可以使用这个系统调用来获取一块COW内存,同时让这块COW内存和已经存在的某块COW内存共享相同的物理内存页面.输入参数 *token* 就是已经存在的这块COW内存的标识符.该系统调用的返回值和 *createarea* 相同,也是包含一个 *token* 和一个地址.返回的 *token* 和输入的 *token* 不同,因为它们表示不同的COW内存.

(3) *void changehookstatus*(*int token*, *int status*).一个进程可以使用这个系统调用控制一块COW内存的状态,该状态表示是否允许其它进程 *hook* 这一块COW内存.输入参数 *token* 是一个COW内存的标识符.如果 *status* 非0,则表示允许这块COW被 *hook*;如果 *status* 为0,则表示这块COW内存不允许被 *hook*.因为一个进程在完成某些操作之前,不希望让其它进程读到脏数据,所以可以通过这个系统调用来禁止其它进程 *hook*.如果某个标识符为 *token* 的COW内存禁止被 *hook*,则 *hook*(*token*)会返回-1.一块COW内存在最开始创建的时候,被设置为禁止 *hook*.

(4) *void release*(*int token*).一个进程可以使用这个系统调用释放由 *createarea* 和 *hook* 所生成的COW内存.当一块COW内存被释放后,它的标识符 *token* 会被回收,可以被重新利用.即使用户不释放COW内存,当进程结束时,COW内存也会被自动释放.

3.4 实 现

在现代的软硬件环境下,当一个进程需要访问内存中的一个字节,这个字节的虚拟地址首先被转换成真正的物理地址,处理器通过地址总线用这个物理地址进行寻址[13].在典型的x86_64平台下,处理器使用4级页表进行转换.一个线性地址空间被分成大小相同的页面,典型的为4KB.进程中的虚拟内存页面被映射到一个真实的物理内存页面之后,才能被进程读写所使用.虚拟内存到物理内存的映射规则,记录在页表中.

不同的页表项可以指向相同的物理页面,因此不同的进程可以共享同一段物理内存.比如,当调用 *fork* 时,父进程要复制整个页表给子进程,所以它们所看到的内存内容相同.调用 *fork* 之后,父子进程的所共享的物理内存几乎都被标记成只读(共享模式的共享内存等除外),当一个进程尝试进行写操作时,写时复制就会发生,操作系统会分配新的物理内存页面给这个进程,复制原始页面的内容到这个新页,再把这个新页设置成可写.然后,进程就可以在新的页面上面完成写操作.

为了实现SWING,我们可以采用 *fork* 所采用的方式(复制页表),但我们没有这样做.与 *fork* 复制整个页表的方法类似,我们可以复制与COW内

存有关的那一部分页表，但是复制这一部分页表仍然有代价. 更糟糕的是，在复制页表的时候，需要阻塞与之相关的两个进程，如果调用 *hook* 的频率稍大，就可能导致整个系统无法工作，全部的资源都用在复制页表上. 为了避免页表的复制，我们放弃了复制页表的方法，转而采用了进程之间共享页表或者部分页表的方法[14]. 调用 *hook* 之后，进程 A 中与被 *hook* 的 COW 内存相关的那一部分页表被共享给进程 B，共享的状态如图 2 所示，页表项中的 R 表示只读. 整个过程仅仅需要更改进程 B 页表的一个页表项. 在此之后，两个进程都可以在自己的 COW 内存中看到相同的内容.

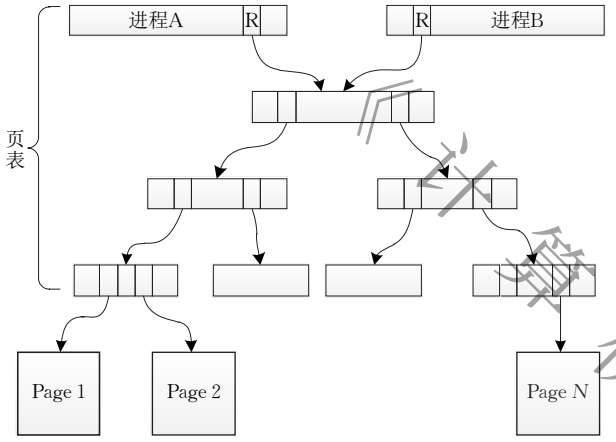


图 2　进程 B 调用 *hook* 之后

进程 B 调用 *hook* 之后，如果进程 A 和 B 其中一个需要写被共享页表指向的数据页，操作系统会给此进程分配一个新的物理页面用来写入数据，同时写时复制也会在页表上发生，算法的主要步骤如算法 1 所示. 限于篇幅，算法中省略了加锁解锁动作、相对次要的判断条件以及异常判断等，只给出了最重要的逻辑顺序，详细代码可以在 github① 找到. 下面均以 X86_64 为例，同时假设使用四级页表. 进程中的四级页表在 Linux 内核中分别称作 *pgd*、*pud*、*pmd*、*pte*[15-17]. 整个进程只有一个页面用来存放所有 *pgd*，共 512 项. 每一个 *pgd* 都可以指向一个物理内存页面，此页面可以存放 512 个 *pud*. 每个 *pud* 都可以指向一个物理内存页面，此页面可以存放 512 个 *pmd*. 每个 *pmd* 可以指向一个物理内存页面，此页面可以存放 512 个 *pte*. 每个 *pte* 都可以指向一个物理内存页面，此页面供进程读写使用. 当然，大部分 *pgd*、*pud*、*pmd*、*pte* 都为空，只有程序所使用的那部分地址对应的 *pgd* 等才非空. Linux 内核可以跟踪物理页面被映射的次数. 每个非空的 *pgd*、*pud*、*pmd*、*pte* 都有一个属性表示读写权限（只读或者可写）.

算法 1.　缺页中断程序片段 *__handle_mm_fault*.

输入：进程 B 的页表 *mm*，不可写的地址 *address*

输出：为进程 B 所分配的新的物理内存页面

```
1.  pgd=pgd_offset(mm,address)
2.  IF(NotNull(pgd)&&ReadOnly(pgd))
3.     handle_pgd_cow(mm,pgd,address)
4.  pud=pud_alloc(mm,pgd,address)
5.  IF(NotNull(pud)&&ReadOnly(pud))
6.     handle_pud_cow(mm,pud,address)
7.  pmd=pmd_alloc(mm,pud,address)
8.  IF(NotNull(pmd)&&ReadOnly(pmd))
9.     handle_pmd_cow(mm,pmd,address)
10. pte=pte_offset_map(pmd,address)
11. handle_pte_fault(mm,address,pte)
```

当页表中的读写权限标志与指令请求的权限不同时，CPU 会自动调用缺页中断算法，必然会调用 *__handle_mm_fault*. 我们在这个函数中添加了部分代码，自顶向下，依次处理每层页表，如果发现某层页表只读，则调用相应的处理函数分配新页，用来存放 *pud*、*pmd* 等. 其中 *handle_pte_fault* 早已在 Linux 内核中存在，这里不再赘述. 其它三个处理函数非常相似，下面只给出 *handle_pgd_cow* 的关键执行路径，如算法 2 所示.

算法 2.　*pgd* 处理程序.

输入：页表 *mm*，*pgd*，地址 *address*

输出：可写的 *pgd*

```
1.  oldpgd_val=*pgd
2.  page=pgd_page(oldpgd_val)
3.  share_count=atomic_read(&page->_mapcount)
4.  IF(share_count>-1)
5.     pgd->pgd=0
6.     pud=pud_alloc(mm,pgd,address)
7.     memcpy(pgd_page_vaddr(*pgd),
8.        pgd_page_vaddr(oldpgd_val),4096)
9.     pud=pgd_page_vaddr(*pgd)
10.    src_pud=pgd_page_vaddr(oldpgd_val)
11.    FOR i=1 to 512
12.       IF (pud_none(pud[i]))
13.          CONTINUE
14.       make_readonly(src_pud[i])
15.       make_readonly(pud[i])
```

① https://github.com/swinglinux/swing/blob/swing/mm/memory.c

计算机学报

16. *atomic_inc*(&*pud_page*(*pud*[*i*])->*_mapcount*)
17. *atomic_dec*(&*page*->*_mapcount*)
18. ELSE
19. *make_writable*(*oldpgd_val*)

算法 2 的主要处理逻辑如下. 获取被 *pgd* 所指向的物理页面 *page*,判断 *page* 是否被共享. 如果 *page* 被共享,则分配一个新的物理页面,把原页面的内容(512 个 *pud*)复制到新页面,同时把原页面和新页面中的每一个 *pud* 设置成只读. 对 *pud* 的继续处理由 *handle_pud_cow* 完成,此函数的工作方式与 *handle_pgd_cow* 类似. 如果 *page* 没有被共享,直接把 *pgd* 设置成可写即可.

当进程 A 想写入 Page 1 时,会产生 Page 1 的一个副本来应付进程 A 的写入,页表中会产生从顶级页表到这个数据页的一条路径,与此路径无关的部分仍然被共享,如图 3 所示. 由于在其它页面上也会发生写入,两个进程之间共享的页表会越来越少. 共享的那部分页表被标记成只读的,只属于某个进程的页表被标记成可写的. 没有标记读写权限的页表项仍然保持原有的权限,因为一旦某一层的页表项被标记成只读,被该页表项所控制的所有更低层页表项都是只读的. 当一个进程退出或者删除相应的 COW 内存时,仅被这个进程独占的页表会被删除.

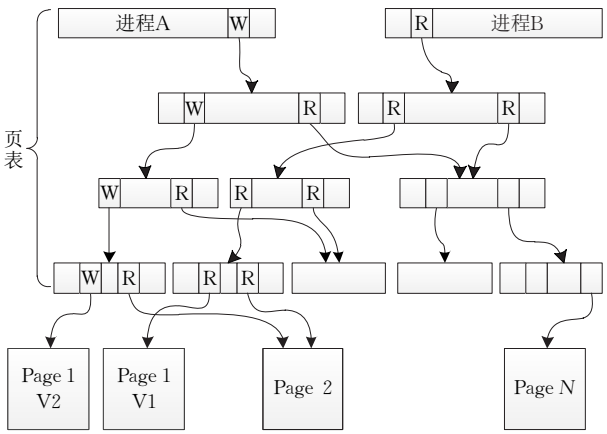


图 3 进程 A 更新 Page 1 之后

在 SWING 方法中,数据共享(即调用 *hook*)速度非常快,几乎不阻塞进程. 因此,它可以被频繁的调用. 虽然后续的写时复制会有额外的开销,但对于修改较少的应用场景,这种开销是可控的. 系统会跟踪每个被 *hook* 的 COW 内存因写时复制所消耗物理内存的数量,当发现这个数量大于某个阈值(可以由用户指定,比如占这个 COW 内存大小的 50%)时,再次 *hook* 这个 COW 内存时会失败.

4 SwingMalloc 内存分配器

通过 SWING 方法获取的内存,地址都按 512 GB 对齐,目前,每次最大可以申请 512 GB 内存. 很多情况下,进程需要多次分配内存,但是需要的内存却很小. 此时,有两种办法:第一种方法是进程直接申请 512 GB 内存,然后再把它分割成小的内存块,用在合适的地方,进程自己管理这些小的内存块;第二种方法是多次调用 SWING 的 *createarea* 函数. 但第一种方法加重了程序员的负担. 第二种方法存在严重的问题,因为 *createarea* 返回的地址按 512 GB 对齐,进程的整个地址空间中,符合这个条件的地址很有限(约 250 个),所以,符合这种条件的地址很快就会被耗尽,导致 *createarea* 无法返回可用的地址. 为了解决上述问题,我们设计并实现了 SwingMalloc 内存分配器. 以上所说的内存,均是指虚拟内存,分配之后,如果没有写操作,并不占用真正的物理内存. 第二种方法所耗尽的,也是合适的虚拟内存地址,不是真正的物理内存.

4.1 架 构

每个进程都有自己的虚拟地址空间,进程在运行过程中,只使用其中的一部分地址,这些地址并不是连续的. 这些地址由多个段组成,单个段内的地址是连续的. Linux 系统已经具有的一些段包括栈、堆以及其它内存文件映射段等,这些段和系统的系统调用、某些库函数的关系如图 4 所示. 加入 SWING 和 SwingMalloc 之后,它们的关系如图 5 所示. 由两幅图可以看出 SWING 和 SwingMalloc 的地位分别与原生 Linux 系统中 *mmap* 和 *malloc* 的地位相当. *mmap* 和 SWING 由操作系统内核提供. 默认情况下,*malloc* 由 C 标准库提供(实现在用户空间). SwingMalloc 也在用户空间实现.

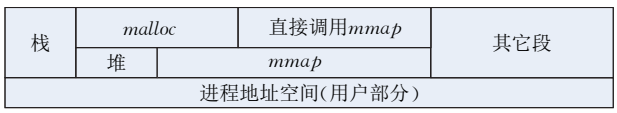


图 4 Linux 内存段

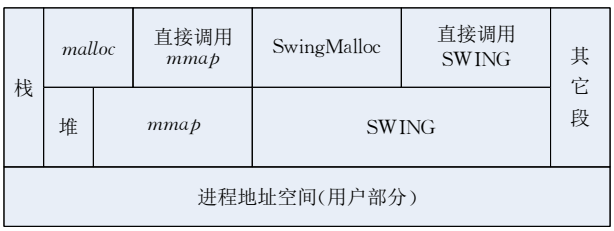


图 5 加入 SWING 之后,Linux 的内存段

C 标准库中的 *malloc* 实现方式如下：当申请的内存较小时，直接在堆中分配；当申请的内存较大时，直接调用 *mmap* 系统调用. 原生的 *mmap* 系统调用也可以供给进程直接使用. 因此图 4 中，*malloc* 在堆和 *mmap* 的上方，而且上层还可以直接调用 *mmap*. SwingMalloc 基于 SWING，因此 SwingMalloc 在 SWING 的上方. SWING 也可以直接供给进程调用，因此它们的关系如图 5 所示.

4.2 接　口

4.2.1　内存分配

用户使用 SwingMalloc 分配内存的过程如下：

(1) 首先调用 SWING 中的 *createarea*(512GB) 或 *hook* 获取一块 COW 内存.

(2) 然后用步骤(1)中得到的 *token*、实际需要的内存的大小作为参数，调用 *swingmalloc*(*int token*, *unsigned long size*).

swingmalloc 这个函数会在 *token* 所代表的 COW 内存中分配一块大小为 *size* 的内存. 如果分配成功，该函数将返回一个结构体，结构体的定义如下：

```
typedef struct AddressInfomation{
    char * addr;
    unsigned long size;
} AddressInfomation;
```

结构体中的 *addr* 表示可用的内存地址. 结构体中 *size* 的值表示实际分配的内存大小，它可能大于进程请求分配的内存大小.

现在系统中广泛使用的内存分配器有以下几种：glic 中的 ptmalloc①、google 开发的 tcmalloc②、facebook 开发的 jemalloc③. 它们都只返回可用的内存地址，不返回实际分配内存的大小. *swingmalloc* 不但返回内存地址，还同时返回实际分配的内存大小. 实际的应用程序经常需要动态的增加某个数组的大小，supersonic 就是这样. 在重新为某个数组分配更大的空间时，很容易造成内存碎片. 虽然这些碎片可以不占用实际的物理内存，但很可能会导致无法分配较大的内存(虽然有很多空闲的内存碎片，但却找不到一段较大而且连续的内存). 如果应用程序能够合理的利用这个返回值，就可以减少应用程序重新分配内存的次数(因为 *swingmalloc* 本来就可能就给程序多分配很多内存).

如果 SwingMalloc 不能为应用程序成功的分配内存，则返回的结构体中两个变量都为 0.

使用 SwingMalloc 需要注意以下两点：(1) 一旦一块 COW 内存使用 *SwingMalloc* 的方式管理，则不能再直接使用 *createarea* 和 *hook* 返回的地址，否则 COW 内存中的数据可能会被破坏；(2) 一块 COW 内存使用 *SwingMalloc* 方式管理之后，仍然可以被共享给其它进程，并且其它进程仍然可以在此基础上使用 *SwingMalloc*.

4.2.2　释放内存

如果一块内存通过 *swingmalloc* 获得，那么应该使用 *swingfree* 函数释放，这个函数的参数就是 *swingmalloc* 返回的结构体中的 *addr*.

4.3 实　现

在 SwingMalloc 中，我们把一块 512 GB 的 COW 内存分成二部分. 当进程申请的内存小于等于 512 MB 时，将从第一部分中分配. 当进程申请的内存大于 512 MB 时，SwingMalloc 将从第二部分内存分配.

第一部分占 128 GB，使用伙伴算法④管理，该算法被广泛应用的在各种系统中，具体细节不再赘述. 我们把 128 GB 的内存区间用一棵线段树组织起来，这是一棵二叉树，并且用静态数组实现，树中每个节点的占 1 个字节，为了减小这个数组的大小，我们让每个节点所表示的区间最小为 128 字节，因此 128 字节也是 *swingmalloc* 可以分配内存的最小单位. 管理 128GB 的空间，这棵树需要占用 512MB 的空间. 树中每个节点都代表一段固定的区间.

第二部分占 384 GB，并把这部分内存划分成大小不等的空闲内存块，不同空闲块的大小和每种大小的内存块所对应的数量如表 1 所示. 一旦进程需要使用这部分的内存，*swingmalloc* 就给进程分配一个完整的空闲内存块. 这个分配算法很简单，找到一个大小合适的空闲内存块，然后把相应的地址返回给进程即可. 用户也可以根据具体的应用，调整表 1 中的分配方案.

表 1　内存块的大小和相应的个数

大小/GB	数量/个
1	64
2	32
4	16
8	8
16	4
32	2

① http://www.malloc.de/
② http://goog-perftools.sourceforge.net/doc/tcmalloc.html
③ http://jemalloc.net/
④ https://en.wikipedia.org/wiki/Buddy_memory_allocation

《计算机学报》

管理第一部分内存所需的线段树、管理第二部分内存所需的空闲块信息等都被称为 *SwingMalloc* 的元信息.这些元信息被放在 COW 内存的开头.紧挨着元信息之后,还预留了约 512 MB 内存,这一段内存可以供应用程序直接写入任何内容,不受伙伴算法的管理,这一段内存的起始地址是固定的.进程之间需要共享整个 COW 内存的时候,可以把需要传递的一些重要信息写在这段内存中.由于 COW 内存的开头保存了 *SwingMalloc* 的元信息,并且又预留了约 512 MB 内存,因此第一部分内存中,可供程序使用的内存约 127GB,第二部分的 384GB 完全可以供程序使用.

5 SwingDB

SwingDB 是一个为数据分析程序而设计的嵌入式内存数据库.它提供了单机版关系型数据库的基本功能,因为它简单的设计,也使得它很容易被扩展到集群中.它允许应用程序把整个数据库映射到自己的内存空间,所以可以避免高代价的进程间通信.SwingDB 应用 SWING 达到了这种效果.

5.1 SwingDB 的功能

SwingDB 是一个嵌入式内存数据库,需要获取数据的应用程序可以把整个数据库包含在自己的程序中,每一个数据库的实例都被包含在某个应用程序中,所以对数据库的操作,不需要传统的进程间通信.其它程序如果想在某个数据库上做统计分析等操作,可以获取整个数据库的快照,并且把这个快照放在它自己的进程空间中,所以它可以很快的访问整个数据库.此进程可以进一步把它自己的快照共享给其它进程,以便其它进程做进一步的分析.这个过程如图 6 所示.

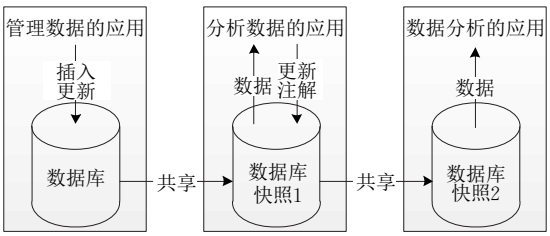


图 6 SwingDB 支持多阶段共享

SwingDB 给每一个数据库快照指定一个不同的名字,一个进程可以通过下面的函数调用来获取某个数据库的快照,并把它放在自己的内存空间中:

bool getsnapshot(*string proposedname*, *string targetname*).

这个函数可以把名字为 *proposedname* 的数据库以快照的形式映射到自己的内存空间中,并指定新的快照的名字为 *targetname*.此后这个进程可以在它自己的这个数据库快照上用 SQL 进行操作,或者用更底层的接口来对这个快照进行操作,以便来完成更复杂的数据分析和数据挖掘任务.可以看出,某一个数据库快照都仅仅被一个进程所独占.当一个进程不再需要一个数据库的快照,或者这个快照过期,进程可以用下面的函数来删除一个快照:

bool discardsnapshot(*string name*).

SwingDB 的优点体现在它可以快速共享快照的能力上,并且几乎没有开销.由于利用了 SWING 技术,共享数据库快照并不需要物理的移动数据.

由于 SwingDB 具有快速共享快照的能力,使得它原生的支持高并发,只要一个进程拿到整个数据库的快照之后,就可以为客户端提供服务,并且完全和其它进程隔离.SwingDB 是面向分析型应用的数据库,这些应用的特点是:整个数据库中有一个表特别大.我们可以把这个数据表水平分片,在集群中的每个节点上安装 SwingDB,每个节点存储一片数据,就可以让很多节点并行处理大量的数据,最后把结果汇总,然后展现给用户.因此,SwingDB 可以非常容易的扩展到分布式的系统中.

SwingDB 不面向 OLTP,因此它里面的数据需要从某些数据源(比如 OLTP 系统)中导入,多个进程使用 SwingDB 通过多个阶段完成的一个任务可以看作是一个事务.系统异常宕机时,正在执行的事务会失败,系统恢复之后,需要重新导入数据,然后重启这个事务.已经完成的事务,最终结果已经通过某种方式展现给用户,并且这种事务并不改变最初导入 SwingDB 的数据,因此 SwingDB 暂时不需要日志.在现实中,有的查询需要消耗几小时,甚至几十小时才能完成,我们正在进行相关研究,使得宕机时,查询不需要从头开始,而可以利用某些中间结果,尽可能缩短查询重启的时间.以后我们会把相关的研究成果应用到 SwingDB 上.

5.2 SwingDB 的实现

SwingDB 把每一个数据库实例或者一个数据库快照存储在一块单独的 COW 内存中.每当创建一个新的数据库,就会为这个数据库创建一个新的 COW 内存.最初阶段,虽然这段 COW 内存占用 512GB 虚拟内存,但是不占用真正的物理内存.当越来越多的数据被插入到数据库时,操作系统会给相应的 COW 内存分配更多的物理内存.SwingDB

使用 *SwingMalloc* 内存分配器管理 COW 内存，如图 7 所示. 基于这种架构，函数 *getsnapshot*() 可以用 SWING 方法中的 *hook* 高效的实现.

SwingDB 基于 Supersonic 而开发，Supersonic 是 google 开发的一个列存储查询引擎. 我们重写了 Supersonic 的存储层，把整个存储空间移动到 SWING 的 COW 内存中. 为了支持快照共享，我们把数据库的元信息存储在 COW 内存中预留的那段内存中(详见 4.3 节)，因此这个数据库可以很容易的被新的进程所识别. SwingDB 的源代码被托管在 Bitbucket 上①. SwingDB 的实现细节如下.

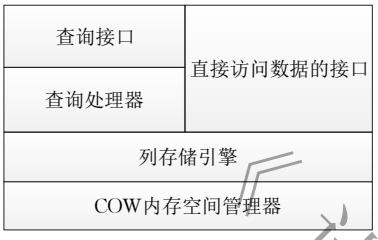


图 7　SwingDB 的架构

在 SwingDB 中，数据被保存在数据表中，应用程序可以把数据按行插入或批量导入到数据表中. SwingDB 中数据表直接使用 Supersonic 中 Table 的实现，每个 Table 包含一个或多个 Column，数据按列存储在每个 Column 中，每个 Column 中的数据存储在一个很大的数组中，给这个数组分配空间就使用 *swingmalloc* 方法. 如果数组需要扩容，则先调用 *swingfree*，然后调用 *swingmalloc*. SwingDB 的元信息会记录每列还有多少剩余的空间，这可以很好的利用 *swingmalloc* 的返回值，大大减少数组扩容的次数，因此可以减少内存的碎片.

当一个进程需要把整个数据库共享给其它进程时，该进程需要先调用 *share* 函数. 这个函数会把数据表中每个列的信息写入 COW 内存中预留的 512 MB 内存中，这些信息包括：表名、行数、列的数量，每个列上的数据类型、数组的地址等. 另一个进程调用 *getsnapshot*，*getsnapshot* 就可以根据这些信息把数据库重建起来. 这里的重建仅仅是指分配一些类的实例，填上一些数组的指针，数据库中真实的数据并没有发生任何复制.

5.3　应用程序场景

在传统的多阶段数据分析系统中，数据被物理的从一个进程传递到另一个进程. 每一个进程都从前一个进程接收数据，执行一个特定类型的数据处理程序，然后把结果和原始数据传递给下一个进程. 数据传递可以通过几种不同的方式，每一个进程可以单独的传递数据，或者所有的进程共用同一个中间数据库存取数据. 这种多阶段的数据分析在现代科学研究中很常见[18-20].

进行内存数据分析时，我们把所有的数据都存在内存中. 在这种场景下，我们希望整个数据分析的过程可以在几秒内完成，以达到良好的交互性[21]，这种数据分析程序是典型的数据密集型程序. 如果数据从一个进程物理的移动到另一个进程，或者利用中间数据库进行数据传输，不可能达到这样的速度. SwingDB 为这种多阶段的内存数据分析提供了一个新的解决方案. 数据永远保存在 SwingDB 中，数据的快照在不同的进程中传递. 每一个进程从上一个进程接收数据快照，在它自己的进程空间中进行数据分析，把包含结果的快照传递给下一个进程. 不管数据流有多复杂，都不会因为数据共享而复制物理内存页面.

在以上应用场景中，只有一个进程对某一个 COW 内存进行写操作，其它进程都只需要读，SwingDB 暂时主要为了支持这些场景. 多个进程同时修改一个原始页面之后，需要用某些策略进行合并，我们会在后续的版本中支持这个特性.

6　性能测试

我们做了多组实验来测试 SWING 和 SwingDB 的性能特点. 实验环境的硬件是一台 HP Z820 工作站，它搭载了 2.6 GHz 的 Intel Xeon E5-2670 处理器，配有 64 GB 的 DDR3 内存. 操作系统是 CentOS 7.1.

6.1　SWING 实验

这一小节从多个方面测试了 SWING 的性能. 首先用最简单的程序测试，然后把它应用到 Redis② 系统上进行测试.

6.1.1　SWING 的代价

我们从两个方面测试 SWING 的代价：数据传输的代价和写时复制的代价.

第一组实验测试数据传输的代价. 我们把 SWING、FIFO 及 socket 做了对比. 对 FIFO 来说，代价是传输所有数据所需要的时间. 对共享内存来说，我们读取数据的进程在读取数据的过程中需要阻塞发送数据的进程，它的代价是调用 *mmap*()、锁定数据和扫描数据所用的时间之和. 可以利用更细粒度的同步方式来对共享内存的方式进行优化，但由于实现的复杂性，我们在试验中没有这样做. 对

① https://bitbucket.org/jlumqz/swingdb/
② https://redis.io/

计算机学报

SWING 来说,它的代价是执行 *hook* 所用的时间.在实验中,我们把数据量从 1GB 逐渐增加到 8GB,测量结果如图 8 所示.和预想的结果一致,SWING 比 FIFO 和共享内存快 5 个数量级.

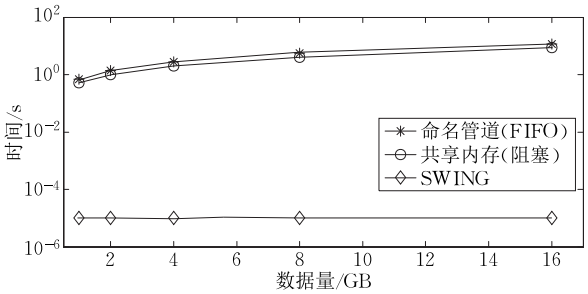
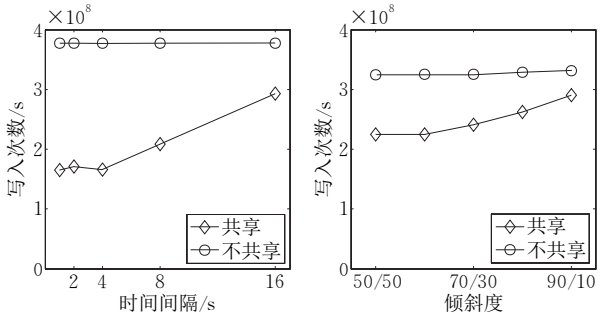


图 8 数据传输的代价

第二组实验测试写时复制的代价,这组实验包括两个子实验.在这组实验中,进程 A 分配一块 8GB 的 COW 内存,并且不停地更新这块内存中的数据;然后,进程 B 周期性的 *hook* 这块 COW 内存.这样,进程的更新操作就会引起写时复制.这组实验的目的是测量进程 A 因为写时复制被拖慢了多少.

第一个子实验我们让进程 A 对数据按从头到尾的顺序进行更新.通过改变进程 B 调用 *hook* 的频率来观察进程 A 吞吐率的变化.我们把这个结果和进程 A 在没有数据共享时的吞吐率进行对比.如图 9(a)所示,写时复制确实对性能会造成影响,这种影响随着共享频率的增大而增大.显然,这种代价是可以控制的.在最坏的情况下,进程 A 更新操作的性能下降了大约 50%.如果把共享频率保持在一个适当的水平(比如,每 20s 共享一次),性能的损失就会很小.



(a) 频率的影响 (b) 局部性的影响

图 9 写时复制的代价

在第二个子实验中,我们让进程 B 每隔 8 秒调用一次 *hook* 函数,并且让进程 A 对数据进行随机的更新.我们改变更新位置分布的倾斜度,并且观察进程 A 的吞吐率,从而可以得到数据局部性对写时复制的影响.如图 9(b)(x 轴上的 70/30 表示 70% 的更新集中到 30% 的数据上)所示,当更新的局部性增强时,写时复制的代价会降低.大多数现实中应用访问的数据都表现出很强的局部性.因此,在大多数情况下,写时复制带来的开销并不大.

6.1.2 OLTP 性能测试

这组实验的目标是测试 SWING 的写时复制会给数据库的 update 造成什么影响.因为 SwingDB 不是为 OLTP 定做的,所以我们选用了 Redis,它是一个为 OLTP 而设计的内存数据库.我们把 SWING 应用到 Redis 上面,将 Redis 的整个存储空间放到一块 COW 内存中.然后让一个数据分析进程周期性的 *hook* 相应的 COW 内存.数据分析进程 *hook* 这块内存之后,将对这块内存进行一遍顺序扫描.同时,我们用 YCSB Benchmark[22] 对 Redis 进行测试,观察数据共享对 Redis 的性能会造成什么影响.我们把 SWING 方法和 FIFO 及共享内存做了对比(使用共享内存的情况下,分析进程在读取数据的时候,需要阻塞 Redis,因为 Redis 不支持更细粒度的同步机制).

在实验中,我们把 YCSB 中 *recordcount* 的值在 workload A、B、C、F 中设置成 900000,在 D,E 中设置成 500000.此时,Redis 刚好使用 8GB 内存.我们把 *operationcount* 设置成 10000000.其它参数,都使用 YCSB 的默认值.我们把调用 *hook* 的频率从每隔 8 秒一次变化到每隔 80 秒一次.Redis 的性能测试结果如图 10 所示(由于版面限制,在此只给出了 workload A、B、D、F 的结果).

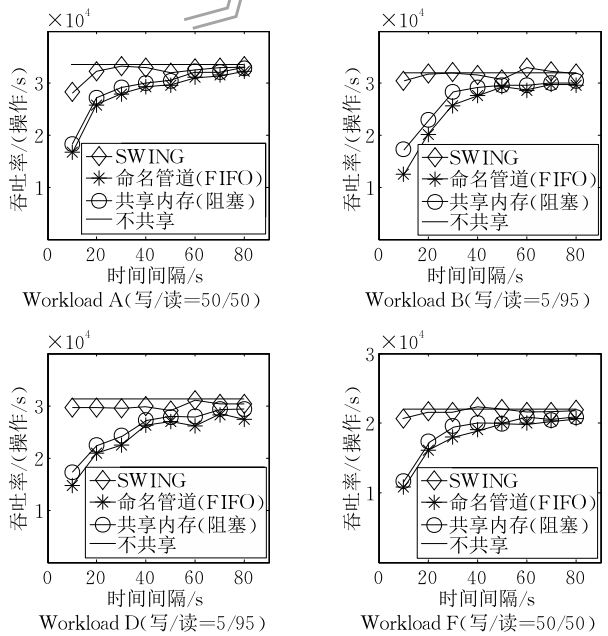


图 10 YCSB 性能测试

可以看出，SWING 对 Redis 的正常工作并不会产生太大的影响. 甚至当数据共享的频率增加到每隔 10 秒一次时，我们还看不到 Redis 的性能有太大的下降(原始 Redis 的 TPS 是 30 000 稍多一点. 虽然现在很多实验性的内存数据库[23]号称可以达到 1000 000TPS，但这对 Redis 并不适用，因为 Redis 是一个单线程的系统).

因为 YCSB 的更新操作表现出很强的数据局部性，所以写时复制带来的性能损失很小(YCSB 访问的数据服从 zipfian 分布). 对于更新密集型的负载来说，比如 workload A，使用 SWING 时性能稍有下降，这表示写时复制会对更新密集型的应用带来代价，虽然这个代价影响很小. 与 SWING 形成对比的是，FIFO 和共享内存都对 Redis 的性能产生了很大的影响，尤其是共享频率变大的时候.

6.2　SwingMalloc 实验

这一节的实验是为了测试 SwingMalloc 的性能，不对分配出来的内存进行任何读写操作. 我们把 SwingMalloc 和 ptmalloc、tcmalloc、jemalloc 作了对比. 这节的实验由两部分组成.

第一部分包括三个实验，我们让内存分配器多次分配、释放内存，三个实验每次操作的内存大小分别为：小于 1 KB、1 KB 至 64 MB、64 MB 至 16 GB. 实验结果如图 11～图 13 所示，横坐标表示分配、释放内存操作的总次数，纵坐标表示这些操作花费的总时间. 可以看出：当频繁分配、释放的内存很小(小于 1 KB)时，SwingMalloc 的性能略逊于其它 3 个内存分配器，这是因为 jemalloc 等内存分配器对这些小的内存分配请求做了很好的优化，SwingMalloc 主要面向大块的内存分配请求，暂时没有对小的内存分配请求做优化；当分配的内存比较大时(大于 1 KB)，SwingMalloc 的优势非常明显，尤其当分配的内存大于 60 MB 时，SwingMalloc 比其它三个内存分配器快 1～2 个数量级. 因为 SwingMalloc 对大块内存的分配算法非常简单，其它 3 个分配器可能需要调用 *mmap* 等系统调用.

第二部分实验模拟了 supersonic 的真实使用场景. 当把数据按行添加到 supersonic 时，supersonic 会因为最初给数组分配的内存不足而不断重新申请内存，这将产生一个申请和释放内存的序列. 在这个实验中，我们把这个序列作为输入，让几种内存分配器完成这个序列中申请、释放内存的操作. 实验结果如图 14 所示，横坐标表示最终向 supersonic 中插入的数据总量，纵坐标表示完成这些操作所消耗的时间. 可以看出，SwingMalloc 比其它几个内存分配器快 1～2 个数量级，这是因为这些操作所申请、释放的内存都比较大，SwingMalloc 处理较大的内存分配、释放请求有很大的优势.

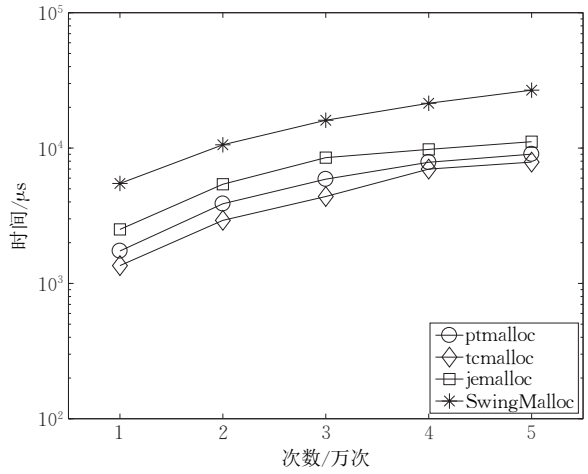


图 11　SwingMalloc 性能测试(内存块小于 1 KB)

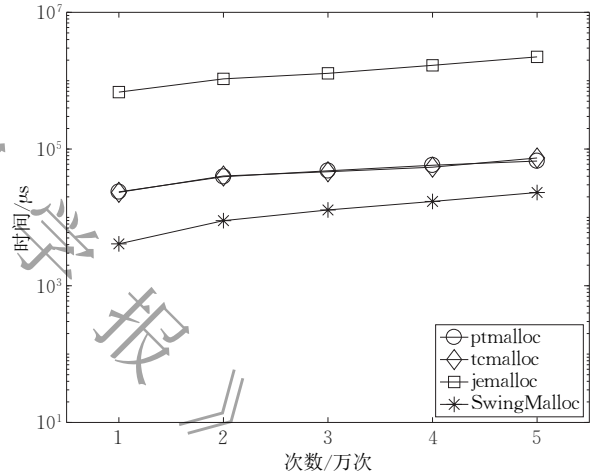


图 12　SwingMalloc 性能测试(内存块为 1 KB～64 MB)

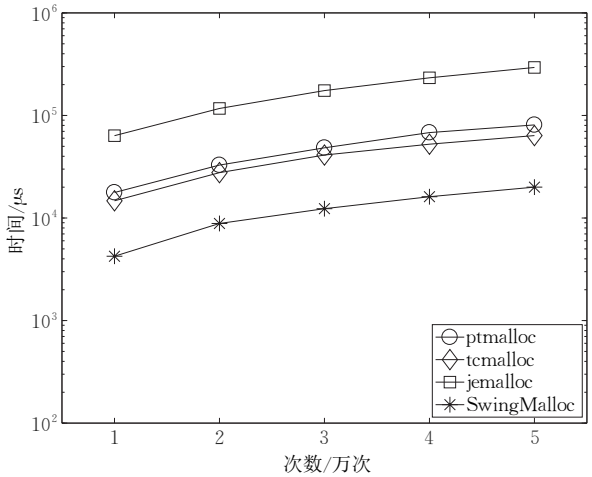


图 13　SwingMalloc 性能测试(内存块为 64 MB～16 GB)

《计算机学报》

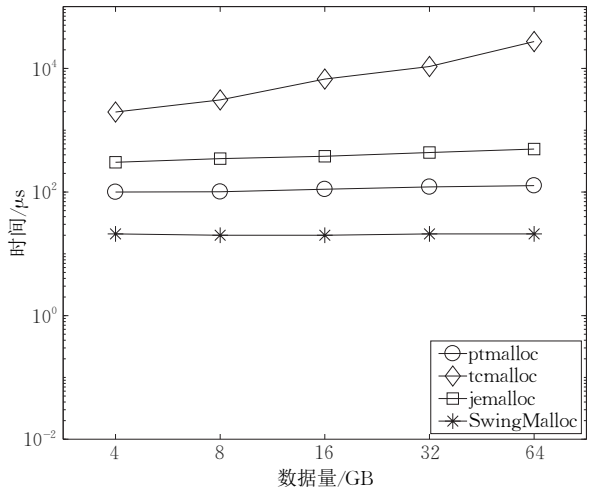


图 14 模拟 Supersonic 的内存分配请求

综合来看,SwingMalloc 可以避免用户自己管理内存,在性能上可以接受.

6.3 SwingDB 实验

为了测试 SwingDB 的性能,我们把它和 Vectorwise(4.2.0 版本)做了对比,Vectorwise 是 OLAP 领域性能最好的内存数据库之一. SwingDB 在数据传输方面比 Vectorwise 快很多. 我们把两个系统放到了由 DBMS 和数据分析程序所组成的系统中. SwingDB 和 Vectorwsie 作为数据库系统,负责数据的存储和查询处理. 数据分析程序用 SQL 从 DBMS 查询到数据,并且在这些数据上做进一步的分析. 对于 Vectorwise,分析程序通过它的 API 调用它,给它发送 SQL 请求,并且把结果集拷贝到自己的内存空间. 对于 SwingDB,分析程序首先通过 SWING 机制获取整个数据库的快照,然后在自己的进程空间内执行 SQL 请求,并且做进一步的分析. 这两种方法主要的区别是 SwingDB 不需要把数据库中的数据在进程间移动.

我们做了三组不同的实验,在不同的侧面把 SwingDB 和 Vectorwise 进行对比. 前两组实验基于车载数据分析场景,使用的数据集[①]中包含深圳市出租车 7 天的车载 GPS 信息,约 1.7 亿条数据,大小约 7 GB. 每条数据包含时间、车辆所在的经纬度、车速、是否载客等信息. 第三个实验基于用户话费分析场景使用人工合成的数据,共 5 亿条,每条描述一个通信用户的话费信息,总大小约 10 GB.

6.3.1 第一组:数据传输量较大的场景

这个实验的目的是为了说明:在数据传输量较大的场景下,SwingDB 的表现远远优于 Vectorwise. 并且在数据传输方面,SwingDB 的速度比 Vectorwise 快 4 个数量级以上.

在这个实验中,我们获取整个数据库的数据,找出乘客上下车的地点,对这些地点进行聚类. 聚类的目的是把整个城市划分成不同的交通小区. 乘客上下车的地点近似为出租车状态(是否载客)变化的点,聚类算法选择 K-means. 此算法需要计算每个点和中心点的距离,我们采用多线程加速.

在这个场景中,需要进行全表扫描,数据库系统内部需要进行的计算极少. 但是大量的数据(约 4 GB)需要从数据库传到 K-means 算法所在的进程,Vectorwise 花费了大量的时间(约 4 min)完成传输过程,SwingDB 则仅仅需要几微秒. 如果聚类算法使用多线程技术,聚类过程会更快,此时 Vectorwise 的传输速度会显得更慢. 整个程序执行(从获取数据开始,到聚类结束)的总时间如图 15 所示,可以看出,在使用 64 个线程的时候,SwingDB 的速度是 Vectorwise 的 16 倍.

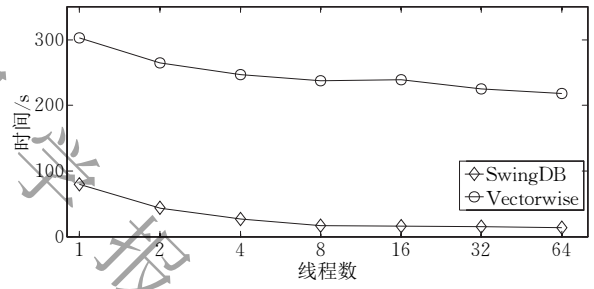


图 15 程序执行的总时间

图 16 把数据传输所耗的时间和聚类所耗的时间独立展示出来.

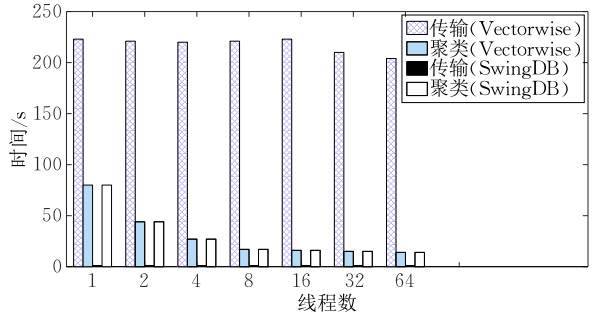


图 16 各部分执行时间

可以看出,Vectorwise 把时间大量的浪费在数据传输上,SwingDB 传输数据所耗时间几乎为零.

① http://www.mcm.edu.cn/html_cn/node/cc3aa1fc07fe689c-77198eaba613678d.html

《计算机学报》

对这个场景来说，SwingDB 完全避免了数据传输的巨大开销.

6.3.2　第二组：数据传输量较小的场景

这个实验的目的是为了说明：在数据传输量较小的应用中，SwingDB 并不会表现出对 Vectorwise 的明显优势. 这类应用一般是计算密集型应用.

在这个实验中，我们计算某地区在某时间段内的平均车速. 把每 15 s 的一个时间段当作一个时间点，所以每天需要计算 5760 个点的平均速度. 我们计算 1 天～7 天的每个时间点的平均速度.

SwingDB 利用 SWING 技术传输整个数据库的时间在 1 ms 以内，Vectorwsie 传输结果集最多需要 70 ms. 由于整个计算过程所花的代价在秒的级别，所以只给出整个执行过程所花费的总时间，不再将数据传输的时间单独列出，如图 17 所示. 可以看出，当需要计算的天数较少时，SwingDB 的性能略低于 Vectorwise，但当天数增加时，SwingDB 的性能越来越接近 Vectorwise. 因为 Vectorwise 是使用多线程技术高度优化的系统，而 SwingDB 是单线程系统，所以并行度较高的时候 Vectorwise 表现要更好. SwingDB 的优点是数据传输快，而不是数据处理快. 在天数较多的时候，单线程的 SwingDB 能够接近 Vectorwise 的性能，因为此时参与计算的数据量有所增加，二者内部处理机制不同.

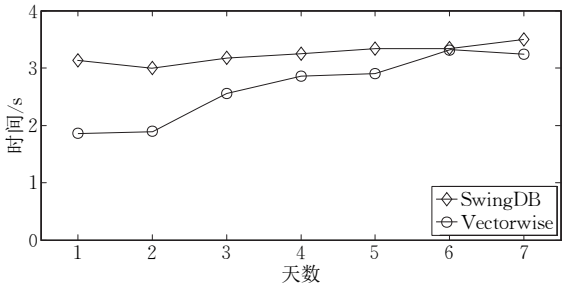


图 17　计算平均速度所花费的总时间

6.3.3　第三组：数据传输量对性能的影响

这个实验的目的是为了说明：随着数据传输量的增大，SwingDB 越来越优于 Vectorwise.

在这个实验中，我们创建一个仅仅包括一张关系表的数据库，并且加载了 10 GB 数据. 我们用一个查询语句来获取数据，并且改变这个查询的选择率. 获取数据之后，应用程序在这些数据上做统计分析，我们选择的统计分析操作是求这些数据的标准差. 整个分析过程的总执行时间如图 18 所示.

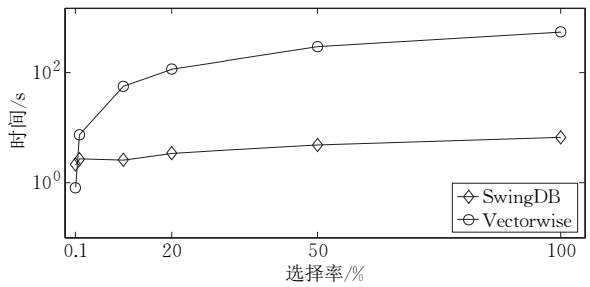


图 18　数据分析的总执行时间

可以看出，当结果集很小的时候（比如查询的选择率为 1%），SwingDB 的性能并不比 Vectorwise 好. 这种情况下，查询执行的时间占了大部分的比例，而且 SwingDB 在查询执行方面并没有做特别的优化，不比 Vectorwise 好. 当查询的选择率逐渐增大时（大于 5%），SwingDB 的优势开始显现出来. 这种情况下，传输结果集占用了大部分的时间. Vectorwise 把结果集传输给分析进程耗费的时间远远大于查询执行的时间. 当大量数据需要传输的时候，SwingDB 的优势非常明显，例如，当选择率高达 100%时，Vectorwise 需要传输 2 GB 数据，因此此时 SwingDB 比 Vectorwise 快两个数量级.

如果我们把整个执行过程的执行时间按不同执行阶段进行分解，如图 19 所示，可以看到 Vectorwise 把时间大部分都花在了进程间数据传输上. 在大规模的数据分析中，数据传输代价很大，会抵消内存数据库的性能优势. 与之相比，SwingDB 利用 SWING 成功的避免了数据的物理移动. 因此，SwingDB 在多阶段数据分析中比传统的内存数据库更加高效.

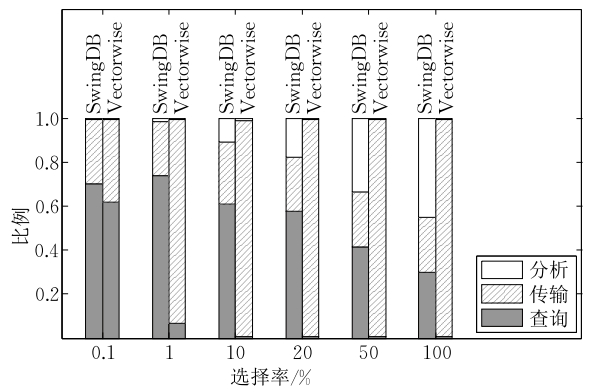


图 19　执行时间的分解

7　总　结

本文介绍了 SWING 和 SwingDB. SWING 是一种新的进程间通信方式，SwingDB 是一个基于

SWING 的嵌入式内存数据库. 和传统数据库不同的是,SwingDB 可以在进程间快速的共享整个数据库快照. 这种把数据库和操作系统进行一体式设计的方式已经被证明对多阶段数据分析系统是有用的,这些系统比较松散的耦合在一起,或者很多模块都产生中间结果. 我们相信这种应用会变的越来越普遍,就像现在的科学数据管理一样[18-19]. 我们在新系统上做了扩展的实验,结果显示 SwingDB 在快照共享方面性能卓越,同时由于写时复制而带来的开销是完全可控的.

作为未来的工作,我们将继续丰富 SwingDB 的功能,例如保存某些中间结果用来加速恢复的速度、合并某些被多个进程同时修改的页面,使它变得更像一个数据库系统和一个高级的数据分析工具. 我们也将邀请数据库和操作系统方面的社区来加入我们,使得 SWING 可以变成数据处理平台上的一个标准工具.

致 谢 审稿人和编辑老师们为本文付出了宝贵的时间,提出了很多宝贵的意见和建议,在此表示感谢!本文获得中国人民大学的萨师煊大数据管理和分析中心的支持,该中心获国家高等学校学科创新引智计划(111 计划)等资助!

参 考 文 献

[1] Meng Q, Zhou X, Chen S, Wang S. SwingDB: An embedded in-memory DBMS enabling instant snapshot sharing//Proceedings of the International Workshop on In-Memory Data Management and Analytics. New Delhi, India, 2016: 134-149

[2] Sarawagi S, Thomas S, Agrawal R. Integrating association rule mining with relational database systems: Alternatives and implications//Proceedings of the ACM SIGMOD International Conference on Management of Data. Seattle, USA, 1998: 343-354

[3] Färber F, Cha S K, Primsch J, et al. SAP HANA database: Data management for modern business applications. ACM Sigmod Record, 2011, 40(4): 45-51

[4] Sikka V, Färber F, Goel A, et al. SAP HANA: The evolution from a modern main-memory data platform to an enterprise application platform. Proceedings of the VLDB Endowment, 2013, 6(11): 1184-1185

[5] Castellano G V. System object model (SOM) and Ada: An example of CORBA at work. ACM SIGAda Ada Letters, 1996, 16(3): 39-51

[6] Crnkovic I, Schmidt H, Stafford J, et al. 4th ICSE workshop on component-based software engineering: Component certification and system prediction. ACM SIGSOFT Software Engineering Notes, 2001, 26(6): 33-40

[7] Kemper A, Neumann T. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots//Proceedings of the 2011 IEEE 27th International Conference on Data Engineering. Hannover, Germany, 2011: 195-206

[8] Liu T, Curtsinger C, Berger E D. Dthreads: Efficient deterministic multithreading//Proceedings of the 23rd ACM Symposium on Operating Systems Principles. Cascais, Portugal, 2011: 327-336

[9] Merrifield T, Eriksson J. Conversion: Multi-version concurrency control for main memory segments//Proceedings of the 8th ACM European Conference on Computer Systems. Prague, Czech Republic, 2013: 127-139

[10] Aviram A, Weng S C, Hu S, et al. Efficient system-enforced deterministic parallelism. Communications of the ACM, 2012, 55(5): 111-119

[11] Beck M. Linux Kernel Internals. Boston, USA: Addison-Wesley, 1998

[12] Beck F, Diehl S. On the congruence of modularity and code coupling//Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. Szeged, Hungary, 2011: 354-364

[13] Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 1 Basic Architecture. Santa Clara, USA: Intel Corporation, 2012

[14] McCracken D. Sharing page tables in the Linux kernel//Proceedings of the Ottawa Linux Symposium. Ottawa, Canada, 2003: 315-320

[15] Bovet D P, Cesati M. Understanding the Linux Kernel. Sebastopol, California, USA: O'Reilly Media, Inc., 2005

[16] Love R. Linux Kernel Development. 3rd Edition. Boston, USA: Addison-Wesley, 2010

[17] Mauerer W. Professional Linux Kernel Architecture. Birmingham, UK: Wrox, 2008

[18] Yu J, Buyya R. A taxonomy of scientific workflow systems for grid computing. ACM Sigmod Record, 2005, 34(3): 44-49

[19] Curcin V, Ghanem M. Scientific workflow systems—can one size fit all?//Proceedings of the 2008 Cairo International Biomedical Engineering Conference. Cairo, Egypt, 2008: 1-9

[20] Leipzig J. A review of bioinformatic pipeline frameworks. Briefings in Bioinformatics, 2017, 18(3): 530-536

[21] Zaharia M, Chowdhury M, Das T, et al. Fast and interactive analytics over Hadoop data with Spark. USENIX Login, 2012, 37(4): 45-51

[22] Cooper B F, Silberstein A, Tam E, et al. Benchmarking cloud serving systems with YCSB//Proceedings of the 1st ACM Symposium on Cloud Computing. Indianapolis, USA, 2010: 143-154

[23] Tu S, Zheng W, Kohler E, et al. Speedy transactions in multicore in-memory databases//Proceedings of the 24th ACM Symposium on Operating Systems Principles. Farmington, USA, 2013: 18-32

《计算机学报》

MENG Qing-Zhong, born in 1988, Ph. D. His current research interests include memory databases and operating systems.

ZHOU Xuan, born in 1979, Ph. D., professor, Ph. D. supervisor. His current research interests include high performance database and information retrieval.

WANG Shan, born in 1944, professor, Ph. D. supervisor. Her current research interests include high performance database, knowledge engineering and big data.

Background

We always need to transmit large volumes of data between DBMS and data analytical applications, especially when conducting large scale statistical analysis. Data transmission is a time-consuming operation if we use traditional IPC methods. As the capacity of memory keeps growing, memory becomes a powerful tool for big data processing. We prefer to place data in memory as much as possible, because big memory can accelerate data processing drastically. Data processing always consists of several steps, large size of data need to be transmitted between these steps. Traditional IPC methods, such as pipe, FIFO, socket and shared memory are not suit to these applications in the large memory environment. Pipe, FIFO and socket are slow, because these methods need to copy the data multiple times. Synchronization is needed when using shared memory. Some approaches, such as SAP HANA, try to avoid inter-process data transmission by integrating data analysis programs into database systems. We believe that this tight coupling method does meet the needs of all applications, so we solve this problem in a different way: we place the database into every related process instead of placing programs into the database.

We implemented a new IPC in the Linux kernel called SWING. It enables any processes sharing physical memory between each other with copy-on-write technology. SWING works as a combination of shared memory and fork, but it is powerful than fork because it supports any processes to share data and fork does not. It has another great strength: programs are loosely coupled when using SWING. Based on SWING, we developed a memory allocator in user space name SwingMalloc, which makes SWING easier to use. SwingMalloc adopts both buddy memory allocation algorithm and a pre-allocation strategy. So it can use memory efficiently whatever sizes of memory we need to allocate. We implemented SwingDB, with SwingMalloc and supersonic, which is an in-memory embedded database and suitable for multi-steps data analysis applications.

This research is supported by the National High Technology Research and Development Program (863 Program) of China (No. 2015AA015307), and the National Natural Science Foundation of China (No. 61772202).

《计算机学报》