NVM+DRAM 混合内存架构下的连接算法优化

罗永平[1)] 金培权[1),2)]

[1)](中国科学技术大学计算机科学与技术学院 合肥 230027)

[2)](中国科学院电磁空间信息重点实验室 合肥 230027)

摘 要 非易失性内存(Non-Volatile Memory,NVM)具有按字节存取、非易失、存储密度高、能耗低等优点,因此被认为是替代 DRAM 的下一代内存技术.虽然目前 NVM 的存取速度远高于闪存,但还低于 DRAM,并且还存在着读写不均衡等问题.因此,综合内存性能、存储密度、非易失性等因素,构建基于 NVM 和 DRAM 的混合内存系统是未来若干年内的可行方案.本论文以 NVM+DRAM 混合内存架构为基础,研究了混合内存架构下传统数据库磁盘连接算法的优化方法.由于传统的连接算法在混合内存架构和纯 DRAM 架构下的 I/O 代价相同,因此我们的主要目标是优化内存代价.在传统的磁盘连接算法中,中间过程产生的数据结构的读写次数存在着较大差别.如果将连接过程的中间数据结构以合适的策略存放在混合内存中,则有望降低连接算法的内存代价.基于这一思路,论文首先给出了一个形式化的数据结构(映像)部署模型,分析了连接算法内存代价的上下界及其成立条件并给出了证明,进而给出了基于最优部署模型的连接算法优化设计.最后,论文实现了 4 种连接算法,包括嵌套循环连接、排序连接、散列连接等 3 种经典连接算法以及面向内存数据库的虚拟分区连接算法,并对比了最优映像部署模型、最差映像部署模型和随机映像部署模型下各个连接算法的性能.实验结果证明,最优映像部署模型能显著提升 4 种连接算法在混合内存架构下的时间性能,并显著减少了 NVM 写总数.

关键词 非易失性内存;混合内存架构;连接算法;优化

中图法分类号 TP311 **DOI 号** 10.11897/SP.J.1016.2020.01069

Optimizing Join Algorithms for NVM+DRAM-Based Hybrid Memory Architecture

LUO Yong-Ping[1)] JIN Pei-Quan[1),2)]

[1)](*School of Computer Science and Technology, University of Science and Technology of China, Hefei* 230027)

[2)](*Key Laboratory of Electromagnetic Space Information, Chinese Academy of Sciences, Hefei* 230027)

Abstract Non-Volatile Memory (NVM) has been regarded as the main alternative of next main memory technologies to replace DRAM, due to its advantages of byte addressability, non-volatility, high density, and low power consumption. Although NVM is currently faster than flash memory, it has higher access latency than DRAM. In addition, the read/write latency of NVM is unbalanced. Therefore, considering all factors involving memory performance, storage density, and non-volatility, it is a feasible solution to construct a hybrid memory system based on NVM and DRAM. Based on such hybrid memory architecture, this paper investigates the optimization of join algorithms on the hybrid memory. As a specific join algorithm has the same I/O cost when running on the hybrid memory and the DRAM-only architecture, we focus on the reduction of memory cost. We noticed that the intermediate data structures of a join algorithm have different read/write operations. Thus, we can reduce memory cost of join algorithms on the hybrid memory by designing an appropriate way to place intermediate data structures on NVM or DRAM. Following this idea, we propose a formal Data Structure Placement Model (DSPM) to address this problem. We analyze

收稿日期:2019-09-05;在线出版日期:2020-02-15.本课题得到国家自然科学基金面上项目(61672479)资助.**罗永平**,硕士研究生,主要研究方向为面向 NVM 的数据库和强化学习.E-mail:ypluo@mail.ustc.edu.cn.**金培权**(通信作者),博士,副教授,中国计算机学会(CCF)高级会员,数据库专委会委员,主要研究方向为面向新型硬件的数据库技术、移动对象数据库.E-mail:jpq@ustc.edu.cn.

the upper bound and lower bound of the memory cost and corresponding conditions, and present proofs. Then, we propose optimizations of join algorithms based on the optimal DSPM model. Finally, we implement four join algorithms including three traditional join algorithms (nested loops join, sort join, and hash join) and the virtual partition join for in-memory databases. We conduct comparative experiments to verify performance of the optimal DSPM model, the worst DSPM model, and the ransom DSPM model on the four join algorithms. The experimental results show that the optimal DSPM model can significantly improve the time performance and reduce NVM writes when integrated with the four join algorithms on the hybrid memory architecture.

Keywords non-volatile memory; hybrid memory architecture; join algorithm; optimization

1 引 言

近年来涌现了多种颇具潜力的非易失性内存(Non-Volatile Memory,NVM)技术[1-2],如相变存储器(Phase Change Memory,PCM)[3]、自旋矩传输磁性存储器(Shared Transistor Technology Random Access Memory,STT-RAM)[4]和可变电阻式存储器(Resistive Random Access Memory,RRAM)[5]等.NVM 具有类似 DRAM 的按位寻址和高读写速率的优势,且存储密度比 DRAM 更高,掉电后数据不丢失,可以很好地支持全内存数据库系统.然而,NVM 一般都存在着读写不对称性的问题,例如 PCM 的读延迟与 DRAM 相当,但其写延迟却比 DRAM 高了一个数量级.此外,NVM 的内存单元存在刷写次数的限制,而 DRAM 则没有这一问题.表 1 给出了 DRAM、NVM(以 PCM 为例)和 HDD 的相关参数对比.

表 1 DRAM、NVM 和 HDD 的对比[6-7]

	参数						
	非易失性	读延时	写延时	读耗能	写耗能	写次数	存储密度
DRAM	不支持	60 ns/64 B	60 ns/64 B	0.8 J/GB	1 J/GB	$+\infty$	1 x
NVM(以 PCM 为例)	支持	115 ns/64 B	395 ns~1 μs/64 B	1 J/GB	6 J/GB	10^6~10^8	2~4 x
HDD	支持	5 ms/sector	5 ms/sector	65 J/GB	65 J/GB	$+\infty$	N/A

从表 1 可以看到,DRAM 和 NVM 在读写延迟、能耗、寿命、非易失性等方面各有所长.业界普遍认为,在可预见的未来 NVM 还无法完全替代 DRAM.因此更合理的方式是构建基于 DRAM+NVM 的混合内存架构,发挥两类内存技术各自的优点,提升内存系统的综合性能.

如何在内存架构中合理地使用 NVM 是构建支持 NVM 的计算机系统需要首先考虑的问题.目前国内外提出的基于 NVM 的主存系统大致分为三种方案[8-10].第一种用 NVM 完全替代 DRAM[8],如图 1(a)所示,但是由于 NVM 读写性能还达不到 DRAM 的水平,短期内这种架构难以成为主流.第二种是层次型架构[9],如图 1(b)所示.这种架构需要把 DRAM 作为 NVM 的缓存,而 NVM 则作为 DRAM 的第二级内存,例如 Intel 傲腾持久化内存的 Memory Mode* 即为这一架构.这一架构存在两个方面的问题.首先,由于 DRAM 只是作为 NVM 的缓存,因此系统可见的内存仅为 NVM 的空间.假设系统配置了 128 GB 的 DRAM 和 512 GB 的 NVM,实际系统能用的内存只有 NVM 的 512 GB 空间,因此在内存空间使用上不合算.其次,这一架构只是利用了 NVM 比 DRAM 容量大的优点,操作系统的数据访问依然还是通过 DRAM,没有充分利用 NVM 的非易失性特点.第三种架构是平行的混合架构,即 NVM 和 DRAM 同时作为同一层次的主存使用[10],如图 1(c)所示.目前 Intel 的傲腾持久性内存所支持的 App-DirectMode* 即为这一架构.在这种架构下,系统可用的内存空间等于 DRAM 和 NVM 的容量之和,而且操作系统感知两类内存的特性(NVM 非易失和 DRAM 掉电易失),可以充分利用 DRAM 和 NVM 各自的优点.这一架构主要的实现难点是需要操作系统和 DBMS 等在软件层面做相应的适配.本论文的工作主要针对图 1(c)所示的平行混合

* Intel optane DC persistent memory operating modes explained. https://itpeernetwork.intel.com/intel-optane-dc-persistent-memory-operating-modes/#gs.okq9q8 2019,12,30

内存架构，另两种架构也有值得研究的价值，我们将在未来工作中进一步探索.

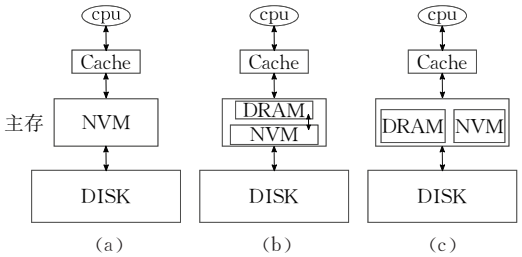


图 1　引入 NVM 后的内存架构

数据库连接算法是数据库物理查询的必要组件，其时间性能对数据库系统，特别是 OLAP 系统至关重要，因此近年来持续涌现出诸多针对新型内存技术 NVM 优化数据库连接算法的工作[6,11-12]. 因为 NVM 写延迟显著高于读延迟，针对 NVM 的连接算法优化无非都是以额外的读操作换取更少的写操作，这种思路在纯 NVM 内存系统下能够显著提升连接时间性能，但是在混合 DRAM 和 NVM 内存架构下，这种思路无法充分利用 DRAM 和 NVM 的特性. 由于 DRAM 读写时延较低，我们考虑对不同读写密集程度的数据对象进行适应性部署. 考虑到连接算法的不同中间关系存在不同的读写频度，如果能找到一种最优的中间数据结构在不同介质上的部署方法，则有望达到混合内存架构下的连接算法代价的下界. 本文的主要目的是优化数据库连接查询算法在混合内存架构上的内存读写代价. 磁盘连接算法的代价包括磁盘 I/O 代价，内存代价和 CPU 代价. 对于特定的连接算法，无论运行在混合内存架构上还是传统 DRAM 的架构上，I/O 代价都是恒定的. 与此同时，由于高密度的 NVM 的引入，混合内存的大小预计会大大超过现有的 DRAM 大小. 在这种情况下，内存代价将成为影响连接算法性能的主导因素[13]. 基于此，本论文将着重优化连接算法在混合内存架构上的内存代价. 由于 NVM 和 DRAM 的读写延迟不同，而且 NVM 存在读写延迟不对称性，理论上通过设计合适的内存分配策略，将算法中具有不同读写比例的数据结构分配到这两种内存设备上，可以有效减少算法的总内存读写代价. 本论文的后续实验也证明了这一点.

总体而言，本文的主要工作和贡献可归纳为如下几点：

(1) 基于 NVM 的读写不对称特性以及 NVM 与 DRAM 读写延迟上的差异，提出了混合内存下的中间数据结构的最优部署策略，它解决了中间数据结构如何部署在混合内存中以保证总内存代价最小的问题. 模型还确定了代价最小时的最优映像部署方案以及代价最大时的最差方案的成立条件，并给出了相应的理论证明.

(2) 基于混合内存下的最优映像部署策略，对 3 种经典数据库连接算法(嵌套循环连接算法、散列连接算法以及排序连接算法)以及 1 种内存数据库连接算法(虚拟分区连接算法[6])进行了优化，给出了各个算法的内存代价分析和中间数据结构部署策略，使这些连接算法能够在混合内存架构下取得较高的性能.

(3) 实现了嵌套循环连接算法、散列连接算法、排序连接算法和虚拟分区连接算法，并通过实验对比了最优部署模型、最差部署模型和随机部署模型的时间性能和 NVM 写总数. 实验结果表明，最优映像部署模型能有效提升 4 种连接算法在混合内存架构下的时间性能，并显著减少了 NVM 写总数.

本节第 2 节回顾国内外相关工作；第 3 节讨论面向混合内存架构的映像部署模型；第 4 节给出基于最优映像部署模型的连接算法优化方法；第 5 节论述实验及结果；最后第 6 节总结全文工作并讨论未来研究方向.

2　国内外相关工作

2.1　非易失内存(NVM)

NVM 技术是近年来最具有潜力的内存技术，它既有传统 DRAM 按位寻址，低读写延迟的优点，还能像二级磁盘存储那样保证掉电内容不丢失，因而被称为持久化内存. 主流的 NVM 技术在保证上述优势的同时，还兼具高存储密度，低制作成本的优势，这有利于满足大数据时代日益增长的低成本、低延迟、大容量、持久化的内存系统需求. 目前主要的 NVM 技术有三种，分别是相变存储器(PCM)[3]、自旋矩传输磁性存储器(STT-RAM)[4]和可变电阻式存储器(RRAM)[5]. 在三种技术中，PCM 制作工艺最成熟，性价比最高，业界已经存在基于 PCM 的相关产品(Intel 于 2019 年正式推出了基于 PCM 的傲腾持久化内存 DCPMM*①).

近年来有很多工作与 NVM 相关，包括系统层面的研究例如基于 NVM 的文件系统和操作系

① Intel optane DC persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html 2019,10,24

统[14-18]和数据库系统内核[6,11,19-23],以及算法层面的研究例如排序算法[11]、无尺度网络生成算法[24]等.文献[14]研究了非易失内存下的虚拟文件系统设计,作者认为传统文件系统通过缓存目录项和 Inode 项来加速文件路径查找对于持久性内存而言不再具有优势.因此,该工作提出了忽略元数据预取步骤,直接访问 NVM 上的目录元数据的 ByVFS 虚拟文件系统.ByVFS 能充分利用 NVM 的字节寻址特性,以提高文件系统性能.文献[22]提出了一个基于日志结构的非易失性内存键值存储系统 TinyKV,它针对持久性内存系统诸多问题:数据库的存储分配问题、数据组织模式、持久性和失败原子性,并给出了相应的解决方案.

此外,针对混合 DRAM+NVM 内存架构的算法也有一些相关进展[25-26].文献[25]提出了基于排序的页布置算法(RAPP).RAPP 在程序执行过程中根据读写热度实时在 DRAM 和 NVM 之间迁移数据页,以减少读写热度较高的数据页.文献[26]提出了更细粒度的数据对象布置工具.数据对象布置工具需要对程序进行预执行操作以获得每个数据对象的平均读写次数,算法依据此统计数据对数据结构选择能耗最优的部署位置,因而能提高程序的综合性能.但是,上述技术实现复杂性较高,需要对操作系统内存管理模块做相应的修改,且运行过程中会给负载引入额外的读写代价,甚至大大降低负载的实时性水平.

2.2 传统连接算法优化

给定两个关系 A,B,若 A 有 n 个字段,B 有 m 个字段,则 A 和 B 可分别表示为 $A(a_1,a_2,\cdots,a_n)$、$B(b_1,b_2,\cdots,b_m)$.若连接字段为 A 中字段 a_i 和 B 中字段 b_j,那么对 A 和 B 的连接算法得到以下元组集合:

$$C=\{t(a_1,a_2,\cdots,a_n,b_1,b_2,\cdots,b_m)\mid t\in A\times B \text{ 且 } a_i\ op\ b_j \text{ 为真}\},$$

其中,$A\times B$ 为关系 A 和关系 B 的笛卡尔积.当 A,B 在连接字段上不存在索引时,算法的时间复杂度为 $O(n^2)$.特别地,当连接操作 op 为等值操作时,即为等值连接算法.针对这种场景的等值连接算法主要有 3 类:分块嵌套连接、散列连接和归并排序连接.

一般认为,数据库连接算法的代价由 I/O 代价、内存代价和 CPU 代价组成[27],其中 I/O 代价取决于算法引发的 I/O 操作次数(寻道次数,磁头旋转次数,传输页数量)[28],内存代价取决于算法对物理内存空间的 load/store 次数,CPU 代价则主要取决于算法引发的 Cache 缺失数量和 TLB 缺失数量[29].根据连接算法代价公式,传统的连接算法优化思路主要分为 3 层:

(1)进行磁盘 I/O 时需选择合适的内存分配方案以减少 I/O 总量[28];尽量减少对磁盘的随机 I/O;选择合适读写磁盘请求块大小;采用直接 I/O 的方式减少额外的内存复制.

(2)内存代价优化需要考虑中间数据结构的实现方式,以减少内存访问代价.因此设计内存数据结构时应尽量不使用链表等包含指针的数据结构;利用哈希或索引加速内存访问;增加算法访问数据结构的空间局部性.

(3)在完成内存连接时,会产生大量的内存访问和 Cache 缺失.因此 CPU 代价的主要优化思路包括[29-30]:对关系进行分区时选择合适的分区数以防止 TLB 项目频繁换入换出;设计物理紧凑的散列表结构以减少 Cache 缺失代价;利用 SIMD 技术和多线程技术优化算法执行时间.

值得一提的是,不同层的优化方案是互不相交的,结合每层的优化方法,磁盘连接算法能达到理论上的性能最优.

2.3 顾及 NVM 的连接算法优化

连接算法执行过程中会产生大量中间关系,这些中间关系要么存储在内存中,要么经过 I/O 缓存区写回磁盘.这意味着算法在不同内存架构下都将对 NVM 执行大量的写操作,因此很多相关工作[6,11-12]均希望能显著减少连接算法的这些写操作,以提升连接算法性能.

文献[6]中研究者提出了数据库虚拟分区连接算法,该算法通过虚拟分区的方式,避免了数据库分区时的拷贝过程,大大减少了数据库连接算法执行过程中对 NVM 的写操作.具体而言,该算法通过虚拟分区的方法,即分区时不需要将元组写入对应分区,而是使用〈键,元组地址〉的格式在内存中维持一个分区表,使用分区表进行连接并通过元组地址获取元组内容得到连接结果.实验表明这一方法能有效减少对 NVM 的写操作总数.

文献[11]提出了减少写 NVM 算法的策略,该策略将算法分成写密集部分和写稀疏部分,通过调整两者比例使算法总时间代价最优.这样算法能显式地调控对 NVM 的总写次数,因而可以在保证性能较优的同时减少对 NVM 的总写次数.利用该策略,文章提出了写限制的数据库连接算法.

文献[12]则研究了多关系连接算法在 NVM 内存环境下的优化方法. 其提出的 NVjoin 算法利用抽样估计得到中间结果数最小的连接次序，该连接次序使得对多个关系执行连接时，使用最少的内存空间，并且连接过程中对 NVM 读写总数最少. 为了进一步优化多表连接，文章还提出了轻量中间表结构 LWTab，该结构充分利用了 NVM 可字节寻址的特性，不将元组拷贝到中间表中，而是将其内存地址存储在 LWTab 中，这种方式进一步减少了对 NVM 的写次数.

然而，上述工作均没有考虑 DRAM+NVM 混合内存架构下的连接算法优化. 本论文试图挖掘两种异构内存的不同特性，分析混合内存架构下连接算法的详细代价，进而提出优化的数据结构部署模型并用于优化连接算法，为混合内存架构下的连接算法设计提供参考.

3　混合内存下的映像部署模型

通过分析传统的数据库连接算法，我们发现连接算法通常都会使用多个中间关系. 在基于 DRAM 和 NVM 的混合内存架构下，如果能找到一种最优的中间数据结构在不同介质上的部署方法，则有望降低混合内存架构下的连接算法代价. 基于此思路，本节我们提出了混合内存下的映像部署模型.

传统的数据库连接算法使用了多个内存中间数据结构. 对于给定的连接关系 R 和 S，连接过程中使用的中间数据结构总体上可以分为输入缓存区 I、输出缓存区 O、暂存关系 R 的缓存区 M_R，暂存关系 S 的缓存区 M_S 和排序工作区 WS 等[28]. 我们将这些不同功能内存空间块统称为缓存区. 不同缓存区在算法执行过程中的读写次数相差较大，因而若将每种缓存区部署到合适的物理内存设备上，既能充分利用混合内存空间，也能保证代价最优. 例如，对于两个大小相同的缓存区 A，B，其中 A 每个单位平均读次数为 10，写次数为 1，B 每个单位平均读次数为 1，写次数为 10. 显然，将 B 分配在 DRAM，A 分配在 NVM 中得到的总内存读写代价会优于其它方案，因为 NVM 对写次数较多的数据结构 B 而言并不友好.

但是，对更复杂的缓存区部署问题，通过经验往往难以轻易得出正确结论. 为了能形式化描述并解决缓存区代价最优内存物理存放位置的问题，本论文提出了混合内存模型下的最优映像部署模型，该模型给出了缓存区不同部署方案的内存读写代价上下界及相应成立条件.

3.1　混合内存模型

混合内存模型是对混合内存架构下进程可用内存空间的一种逻辑抽象. 通过传统的内存分页技术，操作系统在混合物理内存空间和进程逻辑地址空间形成映射，进程会被指定一个连续的逻辑地址空间，该地址空间由一定大小的 DRAM 物理页和指定大小的 NVM 物理页组成.

在混合内存模型下，假设进程被分配了 M 单位的 DRAM 和 P 单位的 NVM，那么进程总的可用内存空间大小为 $M+P$. 设进程读写每个 DRAM 内存单位的代价分别是 r^d 和 w^d，读写每个 NVM 内存单位的代价分布是 r^p 和 w^p（符号含义参见表 2）. 根据表 1 可知 $w^p>r^p$，且 $r^p>r^d$. 进程使用混合内存的最基本的策略有两种：非 NVM 感知的策略和 NVM 感知的策略. 当 NVM 加入到现有的内存系统中时（例如 Intel 的 DCPMM 可以和 DRAM 内存条一样直接插到 DIMM 插槽上），如果操作系统不对 DRAM 和 NVM 的空间进行区分，而是将 NVM 和 DRAM 混合内存空间看成是一块更大的内存，此时操作系统并不能感知到 NVM 的存在，这就是“非 NVM 感知的策略”. 这一策略的好处是现有的操作系统不需要做任何修改即可支持 NVM. 相对地，如果操作系统在使用内存时知道 DRAM 和 NVM 的区别，并且采用不同的内存管理策略对 DRAM 和 NVM 进行控制，则称为“NVM 感知的策略”. 这种策略能够更好地利用 NVM，但是需要软件显式区分两种内存空间. 非 NVM 感知的策略忽略了 NVM 的读写不对称性，将混合内存当成均匀的内存空间使用. 由于 NVM 密度高于 DRAM，因此混合内存系统中我们假设 NVM 容量大于 DRAM，这容易导致较多的读写操作在 NVM 执行，造成系统性能的下降（因为 NVM 的读写延迟高于 DRAM）.

表 2　混合内存模型涉及的主要符号

符号	含义
M	DRAM 大小（单位：Cacheline）
r^d	读一个 DRAM 单位的延迟
w^d	写一个 DRAM 单位的延迟
P	NVM 大小（单位：Cacheline）
r^p	读一个 NVM 单位的延迟
w^p	写一个 NVM 单位的延迟
B_i^d	映像 IMG_i 占用 DRAM 的大小
B_i^p	映像 IMG_i 占用 NVM 的大小
C_i	映像 IMG_i 上的内存读写代价
C_i^g	映像 IMG_i 每个内存单位的代价增益值
C	算法的总内存读写代价

3.2 混合内存读写代价

任何数据结构均需要在物理内存单元上分配存储空间，其在内存空间的表示称为数据结构映像，简称为映像. 映像部署模型(Data Structure Placement Model，DSPM)考察的是数据结构占用的混合内存单元和其内存读写代价之间的关系. 注意，下文采用的内存单位大小均指一个 Cacheline 大小，它是一次主存访问的基本单元，通常为 64 字节大小.

假设某算法中使用了 t 个数据结构，其对应的 t 个映像记为$\{IMG_i \mid i=1,2,\cdots,t\}$，若这 t 个映像以某种方式从混合内存空间中分配空间，我们将这种分配定义为一种映像部署方案，它确定了每个数据对象占有 DRAM 和 NVM 内存空间的大小. 对于任意映像 IMG_i，若已知它每个内存单位的平均读写次数，不妨记读 m_i 次，写 n_i 次，则对于 IMG_i 的每个内存单位：若该单位存储在 DRAM 中，则该单位读写代价 $C_i^d=m_i r^d+n_i w^d$；若该单位存储在 NVM 中，则其读写代价 $C_i^p=m_i r^p+n_i w^p$；将该单位从 NVM 换入 DRAM 造成的读写代价增益为 $C_i^g=C_i^p-C_i^d$. 在某映像部署方案下，若 IMG_i 占用了 DRAM B_i^d 内存单位，占用了 NVM B_i^p 内存单位，则可计算映像 IMG_i 的内存读写代价为 $C_i=C_i^d B_i^d+C_i^p B_i^p$，算法对内存的总内存代价为：$C=\sum_{i=1}^{t}(C_i^d B_i^d+C_i^p B_i^p)$.

3.3 最优和最差映像部署方案

定理 1. 如果在某映像部署方案下，算法的总内存读写代价最小，那么在该部署方案下，若 $\forall x,y\in\{1,2,\cdots,t\}$ 满足 $C_x^g>C_y^g$ 且 DRAM 不能同时容纳 IMG_x 和 IMG_y 时，DRAM 必定优先分配给 IMG_x.

证明. 假设某映像部署方案下内存代价 C 达到最小. 若方案满足 $C_x^g>C_y^g$ 且 DRAM 不能同时容纳 IMG_x 和 IMG_y 时，DRAM 仍可不优先分配给 IMG_x. 由 DRAM 未优先分配给 IMG_x，则必有可分配给 IMG_x 的 DRAM 空间分配给了 IMG_y，即 $B_x^p>0$ 且 $B_y^d>0$. 不妨假设 $B_x^p>B_y^d$，则交换 DRAM 中 B_y^d 单位的 IMG_y 和 NVM 中 B_y^d 单位的 IMG_x，得到一种新的内存部署，该部署的总内存读写代价为

$$\begin{aligned}C'&=\sum_{i=1,\cdots,t \text{ and } i\neq x,y}(C_i^d B_i^d+C_i^p B_i^p)+C_x^d(B_x^d+B_y^d)+\\&C_x^p(B_x^p-B_y^d)+C_y^d(B_y^d-B_y^d)+C_y^p(B_y^p+B_y^d)\\&=\sum_{i=1,\cdots,t}(C_i^d B_i^d+C_i^p B_i^p)+B_y^d(C_x^d-C_x^p+C_y^p-C_y^d)\\&=C+B_y^d(C_y^g-C_x^g).\end{aligned}$$

因为 $B_y^d>0$、$C_x^g>C_y^g$，故 $B_y^d(C_y^g-C_x^g)<0$，故 $C'<C$，这与原内存分配的总内存读写代价最小矛盾，故定理得证. 证毕.

定理 2. 如果在某映像部署方案下，算法的总内存读写代价最大，那么在该部署方案下，若 $\forall x,y\in\{1,2,\cdots,t\}$ 满足 $C_x^g>C_y^g$ 且 DRAM 不能同时容纳 IMG_x 和 IMG_y 时，NVM 必定优先分配给了 IMG_x.

定理 2 的证明与定理 1 类似，此处略.

我们将定理 1 中代价最优的方案称为最优部署方案，定理 2 对应的代价最差的部署方案称为最差部署方案. 由上两定理可知，最优部署方案将 DRAM 优先分配给 C^g 值较大的数据结构，而最差部署方案则将 NVM 优先分配给 C^g 值较大的数据结构. 具体场景中，最优部署方案和最差部署方案的内存代价存在较大差别. 例如，若需将 3 个数据结构 A,B,D 部署在 8 个单位的 NVM 和 4 个单位的 DRAM 组成的混合内存中，假设取 $r^d=1,w^d=1,r^p=2,w^p=20$，A、B、D 读写次数和 C^g 值如表 3 所示.

表 3 数据结构 A,B,D 参数设定

	参数			
	大小(Cacheline 数)	单位读次数	单位写次数	C^g
A	4	2	8	154
B	4	3	3	60
D	4	8	2	46

由表 3 可知 $C_A^g>C_B^g>C_D^g$，故可计算最优和最差方案的内存读写代价：

(1) 最优映像部署方案. A 存放在 DRAM 中，B、D 存放在 NVM 中，总读写代价为 $|A|(2r^d+8w^d)+|B|(3r^p+3w^p)+|C|(8r^p+2w^p)=528$.

(2) 最差映像部署方案. D 存放在 DRAM 中，A、B 存放在 NVM 中，总读写代价为 $|C|(8r^d+2w^d)+|B|(3r^p+3w^p)+|A|(2r^p+8w^p)=960$.

对于上述设定，最差部署方案的内存代价为最优部署方案代价的 1.82 倍. 如果 A 的写次数增大，两者差距将更大.

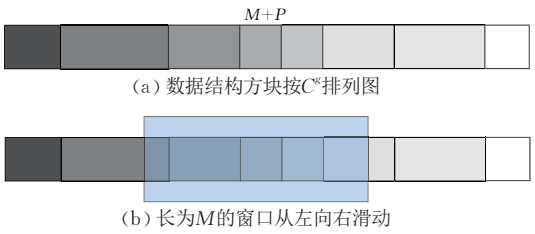
3.4 其它映像部署方案

为了形象地表示其它的映像部署方式，我们做如下抽象：用一个方块表示一个数据结构①，将算法的 t 个数据结构按照其 C^g 值从大到小排序后首尾相连排列成如图 2(a)的形式.

其中方块长度表征数据结构所占内存空间大小；方块越靠左代表其 C^g 越大，颜色越深；如果排列

① 映像用一个连续的方块表示，但是并不代表数据结构占用的物理地址必须连续，而是指同种数据结构分配空内存空间的优先级相同.

后长度不足 $M+P$，在最排列的右边用 $C^p=C^d=C^g=0$ 的空方块补满，它代表空闲不用的内存空间）. 使用一个长度为 M 的窗口，首先将该窗口紧靠排列的最左端，然后将该窗口从左往右滑动直到窗口右边与方块排列最右端重合，如图 2(b). 窗口至多可滑动距离为 P 单位长度.



(a) 数据结构方块按C^g排列图

(b) 长为M的窗口从左向右滑动

图 2　数据结构部署示意图

如果我们规定将长为 M 的窗口内的所有完整方块或部分方块代表的映像存放在 DRAM 中，其它的映像均存放在 NVM 空间中，那么窗口在任何一个固定位置，都能代表一种映像部署方案.

推论 1.　长为 M 的窗口从最左端滑动到最右端时，若数据结构分散存储于 DRAM 和 NVM，则该数据结构对应的部署方案总内存读写代价单调递增.

证明.　由定理 1，当 $x=0$ 时，总内存代价最小. 假设窗口在滑动 $x(0<x\leq P)$ 单位长度后，此时对应部署方案的总内存读写代价为 $C(x)$. 令窗口向右滑动 Δx 单位长度，此时对应的新部署方案的总内存代价为 $C(x+\Delta x)$. 该方案与原部署方案的不同之处为：将位于原窗口内最左端的 Δx 单位的 IMG_a 从 DRAM 换到 NVM，将原窗口外右边 Δx 单位的 IMG_b 从 NVM 换到 DRAM. 故有：$C(x+\Delta x)=C(x)+\Delta x(C_a^p-C_a^d)+\Delta x(C_b^d-C_b^p)=C(x)+\Delta x(C_a^g-C_b^g)$. 又 IMG_a 位于 IMG_b 的左边，故 $C_a^g\geq C_b^g$，所以 $C(x+\Delta x)\geq C(x)$. 由定理 2，当 $x=P$ 时，总内存代价最大. 综上得证.　证毕

4　基于最优映像部署的连接算法优化

如 2.2 节所述，优化传统磁盘连接算法有三个正交的层面：优化 I/O 代价、优化内存代价和优化 CPU 代价. 优化 I/O 代价和纯 CPU 代价的方法同样可以直接应用到传统 DRAM 架构环境中去，因而本节将重点讨论针对混合内存架构优化磁盘连接算法内存读写代价的方法.

非 NVM 感知的内存使用策略无法保证将写密集的数据结构分配到 DRAM 中，甚至很有可能得到较差的部署方案，这将成倍地增加内存读写代价. 因此我们需要对磁盘连接算法缓存区采用最优部署方案，以达到内存层代价的最优. 为了确定算法各个内存缓存区的最优的内存部署方案，映像部署模型需计算各缓存区的平均读写次数，据此计算每个存储单元的读写代价，即存放在 DRAM 中的读写代价 C^d，存放在 NVM 中的读写代价 C^p 以及从 NVM 中换入到 DRAM 产生的代价增益 C^g. 在对缓存区单元的 C^g 值从大到小排序之后，只要将排序越靠前的缓存区的优先存放到 DRAM 内存中，算法的内存读写代价便最小.

下面我们分析不同磁盘连接算法实现，考察每个算法及其缓存区平均内存读写次数，并给出它们的最优部署方案.

4.1　嵌套循环连接算法(BNJ)优化

DBMS 中常用的嵌套循环连接算法是分块嵌套连接算法(Blocked Nested Loop Join，在本论文中用缩写 BNJ 表示). BNJ 算法最大的优势在于：其磁盘 I/O 均为顺序 I/O，而顺序 I/O 由于不需要磁盘寻道和过多的磁头旋转，代价远小于磁盘随机 I/O. 当 R 和 S 中有一个关系相对较小时，BNJ 是一个不错的选择. BNJ 算法的流程如算法 1 所示，算法使用了 3 个缓存区：输入缓存区 I、关系 R 的缓存区 M_R 和关系 S 的缓存区 M_S.

算法 1.　BNJ 算法伪代码.

```
输入：relation R,S, buffer I, M_R, M_S
输出：result of R joins S
FOR i=1 to |R|/|M_R| DO
  Read R's tuples to fill M_R;
  FOR j=1 to |S|/|M_S| DO
    Read S's tuples to fill M_S;
    Perform In-memory join on M_R and M_S;
  END FOR
END FOR
```

对内存关系进行分区是利用额外 CPU 计算减少内存代价的常见方法[30]. 我们也对缓存区 M_R 采用分区结构，即从输入缓存区解析出 R 的元组后，将元组存入 M_R 内对应分区中. 如果缓存区 M_R 和 M_S 均采用分区结构，且保证 R 和 S 的两个子分区能够完全放入最底层 Cache，这样内存连接算法完成连接操作只需要将内存中的 M_R 和 M_S 加载一次进入最底层 Cache(Last Level Cache)，这将能大大减少对混合内存读写代价. 使用了如上分区结构的缓存区后，其平均读写次数如表 4 所示.

表 4 BNJ 算法缓存区每个单位平均读写次数

	读写次数	
	读次数	写次数
I	$\frac{\left(\lvert R\rvert+\frac{\lvert R\rvert}{\lvert M_R\rvert}\lvert S\rvert\right)}{\lvert I\rvert}$	$\frac{\left(\lvert R\rvert+\frac{\lvert R\rvert}{\lvert M_R\rvert}\lvert S\rvert\right)}{\lvert I\rvert}$
M_R	$\frac{\lvert R\rvert}{\lvert M_R\rvert}\frac{\lvert S\rvert}{\lvert M_S\rvert}$	$\frac{\lvert R\rvert}{\lvert M_R\rvert}$
M_S	$\frac{\lvert R\rvert}{\lvert M_R\rvert}\frac{\lvert S\rvert}{\lvert M_S\rvert}$	$\frac{\lvert R\rvert}{\lvert M_R\rvert}\frac{\lvert S\rvert}{\lvert M_S\rvert}$

表 4 所示的读写次数详细分析如下：

(1)输入缓存区 I. 关系 R 和 S 的所有磁盘读取均先经过该缓存区.每次填满 I 并完成解析操作的过程中,该缓存区的每个单位平均读/写各 1 次.算法总共需要遍历 1 次关系 R,且遍历($|R|/|M_R|$)次关系 S,因而经过该缓存区的单位总量为 $|R|+(|R|\cdot|S|/|M_R|)$,I 总共被填满的次数为:$|R|+(|R|\cdot|S|/|M_R|)/|I|$,故该缓存区平均读/写次数为:$(|R|+(|R|\cdot|S|/|M_R|))/|I|$.

(2)缓存区 M_R. 每次填满 M_R 则写每个单元一次,故 M_R 的写次数即为嵌套循环的外层循环数,即($|R|/|M_R|$);其读次数与内层循环有关,每当 M_S 被关系 S 填满,即需要对 M_R 和 M_S 执行内存连接操作,即读 M_R 的每个单元一次,因此共需要读($|R|/|M_R|$)·($|S|/|M_S|$)次.

(3)缓存区 M_S. 每次对 M_S 填满并连接过程均发生在内层循环中,故其读/写次数均为内层循环次数,即平均每个单位读/写($|R|/|M_R|$)·($|S|/|M_S|$)次.

三个缓存区可以视为 3 个内存数据结构,根据它们的平均读写次数,计算其对应的 C^g 值如式(1)~(3)所示.

$$C_I^g=\frac{|R|+\frac{|R|}{|M_R|}|S|}{|I|}(r_p-r_d)+\frac{|R|+\frac{|R|}{|M_R|}|S|}{|I|}(w_p-w_d) \tag{1}$$

$$C_{M_R}^g=\frac{|R|}{|M_R|}\frac{|S|}{|M_S|}(r_p-r_d)+\frac{|R|}{|M_R|}(w_p-w_d) \tag{2}$$

$$C_{M_S}^g=\frac{|R|}{|M_R|}\frac{|S|}{|M_S|}(r_p-r_d)+\frac{|R|}{|M_R|}\frac{|S|}{|M_S|}(w_p-w_d) \tag{3}$$

一般地,我们可以假设输入缓存区 I 远小于关系 R 和 S 的大小,故易证:$C_I^g>C_{M_S}^g>C_{M_R}^g$.因此 BNJ 算法在执行最优内存分配时应按 $C_I^g>C_{M_S}^g>C_{M_R}^g$ 的顺序从 DRAM 分配空间给这三个缓存区数据结构,这样总内存代价最优.

4.2 散列连接算法(HHJ)优化

散列连接算法在实际实现时存在着多种方式[27],本论文主要考查混合散列连接算法(Hybrid Hash Join,在本文中以缩写 HHJ 表示).HHJ 算法理论上性能优于其它散列连接,它通过在内存中保留部分分区以减少写回磁盘的元组总数,因此 HHJ 算法的优势在于它具有最少的 I/O 代价.

HHJ 算法的流程如算法 2 所示.由算法 2 可知,混合散列连接算法共分为两个阶段.在第一阶段,算法将 R 和 S 分区,对存储在内存中的分区 M_R,M_S 执行内存连接算法,将其它分区写回磁盘.此阶段算法使用了 4 个内存缓存区,分别是缓存区 I、输出缓存区 O、暂存关系 R 的缓存区 M_R 和暂存关系 S 的缓存区 M_S.为了减少内存读写代价,同样对 M_R 和 M_S 采用分区结构,保证子分区能够完全放入底层 Cache,分区大小保证 M_R 能容纳下 R 的一个分区,每当 M_S 缓存区满后,对 M_R 与 M_S 执行一次 Cache 友好的内存连接并清空 M_S.表 5 给出了 HHJ 算法读写次数.

算法 2. HHJ 算法伪代码.

```
输入:relation R,S, buffer I,O,M_R,M_S
输出:result of R joins S
FOR EACH r in R's tuple input buffer DO
  pid = partition ID of r;
  IF pid==0 THEN
    Insert r into M_R;
  ELSE
    Output r into the pid-th partition;
  END IF
END FOR
FOR EACH s in S's tuple input buffer DO
  pid = partition ID of s;
  IF pid==0 THEN
    Insert s into M_S;
    IF M_S is full THEN
      Perform in-memory join for M_R,M_S;
      Clear M_S;
    END IF
  ELSE
    Output s into the pid-th partition;
  END IF
END FOR
Perform in-memory join for M_R,M_S;
FOR i=1 to Max partition ID DO
  Run BNJ on the i-th partition of R and S;
END FOR
```

表 5　HHJ 算法缓存区每个单位平均读写次数

	读写次数	
	读次数	写次数
I	$\frac{(\lvert R\rvert+\lvert S\rvert)}{\lvert I\rvert}$	$\frac{(\lvert R\rvert+\lvert S\rvert)}{\lvert I\rvert}$
O	$\frac{\left(\lvert R\rvert+\lvert S\rvert-\lvert M_R\rvert-\lvert S\rvert\frac{\lvert M_R\rvert}{\lvert R\rvert}\right)}{\lvert O\rvert}$	$\frac{\left(\lvert R\rvert+\lvert S\rvert-\lvert M_R\rvert-\lvert S\rvert\frac{\lvert M_R\rvert}{\lvert R\rvert}\right)}{\lvert O\rvert}$
M_R	$\frac{\lvert S\rvert}{\lvert R\rvert}\frac{\lvert M_R\rvert}{\lvert M_S\rvert}$	1
M_S	$\frac{\lvert S\rvert}{\lvert R\rvert}\frac{\lvert M_R\rvert}{\lvert M_S\rvert}$	$\frac{\lvert S\rvert}{\lvert R\rvert}\frac{\lvert M_R\rvert}{\lvert M_S\rvert}$

表 5 所示的读写次数详细分析如下：

(1) 缓存区 I. 每次 I 被填满并完成元组解析，平均每个单位读写各 1 次. 第一阶段只需要对 R 和 S 遍历一次，故途经该缓存区的元组总数为 $|R|+|S|$，每单位平均读写次数为 $(|R|+|S|)/|I|$.

(2) 缓存区 O. 注意到除了保存在内存中的关系 R 的首个分区(大小为 $|M_R|$)以及关系 S 的首个分区(大小为 $|S|\cdot|M_R|/|R|$)不需要写回磁盘外，其余分区均需经过缓存区 O 写回磁盘，所以经过该缓存区的元组总数为 $|R|+|S|-|M_R|-(|S|\cdot|M_R|)/|R|$. 因此，我们可以计算得到每个单位的平均读/写次数：$(|R|+|S|-|M_R|-(|S|\cdot|M_R|)/|R|)/|O|$.

(3) 缓存区 M_R 和 M_S：M_R 保存的是关系 R 的首个分区. 根据算法 2，每当缓存区 M_S 写满后，便对 M_R，M_S 执行内存连接，这期间读取 M_R，M_S 的每个单位各 1 次. 关系 S 的首个分区大小为 $|S|\cdot|M_R|/|R|$，故缓存区 M_S 总共会被填满 $(|S|\cdot|M_R|)/(|R|\cdot|M_S|)$，因此 M_S 的每个单位平均读/写次数为：$(|S|\cdot|M_R|)/(|R|\cdot|M_S|)$，同样 M_R 每个单位读的次数也为：$(|S|\cdot|M_R|)/(|R|\cdot|M_S|)$. 而 M_R 只会被填满一次，故缓存区 M_R 每个单位平均只写 1 次.

第一阶段使用了多个输出缓存区用于暂存多个需要写回磁盘的分区，故 $|I|<|O|$；当输入和输出缓存区较小时，易证 $C_I^g>C_O^g>C_{M_S}^g>C_{M_R}^g$，因此最优缓存区部署方案需要按照该次序将 DRAM 空间分配给上述缓存区. 在第二阶段，对磁盘上关系 R，S 的分区对执行 BNJ，因此该阶段缓存区的读写次数与 BNJ 算法中列出的一致.

4.3　排序连接算法(SMJ)优化

排序连接算法(Sort Merge Join，本论文以缩写 SMJ 表示)需要先对关系 R 和 S 进行外部归并排序，然后对两个有序的关系进行连接操作. 归并连接算法对较大的关系性能较好，即 SMJ 算法适合可用内存相对较小的场景[27]. 此外，如果将最后一次的归并操作和关系连接操作合并，则可以省去 $2(|R|+|S|)$ 的 I/O 传输代价. 因此 SMJ 算法包括两个阶段：阶段一生成多个外部有序子关系(算法 3)，阶段二进行归并的同时完成连接(算法 4).

算法 3.　SMJ 排序阶段伪代码.

```
输入：relation R,S, buffer I,O,WS
输出：sorted runs of R and S
FOR i=1 to |R|/|WS| DO
  Read R's tuples to fill WS and Sort WS;
  Output WS as a sorted run;
END FOR
FOR i=1 to |S|/|WS| DO
  Read S's tuples to fill WS and Sort WS;
  Output WS as a sorted run;
END FOR
```

算法 4.　SMJ 归并阶段伪代码.

```
输入：sorted runs of R and S
输出：result of R joins S
Merge on R's runs in Priority Queue QR;
r=pop(QR);
Merge on S's runs in Priority Queue QS;
s=pop(QS);
WHILE not Empty(QR) and not Empty(QS) DO
  IF r.key<s.key THEN
    r=pop(QR);
  ELSEIF r.key>s.key THEN
    s=pop(QS);
  ELSE
    b=s.key; B={};
  WHILE not empty(QS) and s.key=b DO
    B=B∪{s}; s=pop(QS);
  END WHILE
  WHILE not empty(QR) and r.key=b DO
    Join(r,B); r=pop(QR);
  ENDWHILE
 END IF
END WHILE
```

SMJ 算法第一阶段共使用了三个内存缓存区：缓存区 I，输出缓存区 O 和排序使用的工作空间 WS. WS 缓存区的读写次数即排序区进行快速排序所产生的总读写次数. 在 SMJ 算法中，三个缓存区的平均读写次数如表 6 所示. 以缓存区 WS 为例，每次内存排序需要对该区域内存读 $c_1\log(|WS|)$ 次，

写 $c_2\log(|WS|)$次.因为 SMJ 算法需要对两个关系 R 和 S 分别进行排序,故共执行了$(|R|+|S|)/|WS|$次内存排序,因而该缓存区平均每个单位读的次数为:$c_1(|R|+|S|)\log(|WS|)/|WS|$,写操作的次数为:$c_2(|R|+|S|)\log(|WS|)/|WS|$.根据统计实验可得出常数 $c_1=1.8$, $c_2=1.2$,假设缓存区 I 与 O 设为同样大,可计算得三个缓存区的读写次数为 $C^g_{WS}>C^g_I=C^g_O$.

表 6 SMJ 算法缓存区每个单位平均读写次数

	读写次数	
	读次数	写次数
I	$\frac{(\|R\|+\|S\|)}{\|I\|}$	$\frac{(\|R\|+\|S\|)}{\|I\|}$
O	$\frac{\|R\|+\|S\|}{\|O\|}$	$\frac{\|R\|+\|S\|}{\|O\|}$
WS	$\frac{c_1(\|R\|+\|S\|)}{\|WS\|}\log(\|WS\|)$	$\frac{c_2(\|R\|+\|S\|)}{\|WS\|}\log(\|WS\|)$

我们知道,内存排序会产生 $O(n\log(n))$的写,如果这些写操作应用在 NVM 上,排序过程将会产生难以容忍的代价.因此我们的排序算法对混合内存架构做了相应的优化,它能将大量的写偏向到 DRAM 空间中,该工作已经超出了本文范围,此处不做详细阐述.

算法第二阶段使用了数个等大的输入缓存区和两个优先队列 QR 和 QS.输入缓存区用于读入 R 和 S 的多个有序子关系,由于每个磁盘有序子关系大小相同,故输入缓存区的读写次数都相同;优先队列 QR 和 QS 用于归并 R 和 S.由于优先队列大小等于磁盘子关系总数量,故队列均可以完全装入底层 Cache,对优先队列的读写操作基本可以在 Cache 中命中.综上该阶段的内存需求较小,可完全放入 DRAM 中,故不列出各缓存区读写次数.

4.4 虚拟分区连接算法(VRTPJ)优化

虚拟分区连接(Virtual Partition Join,在本论文中用缩写 VRTPJ 表示)算法是针对 NVM 内存环境设计的一种数据库连接算法[6].其基本思路是:当对内存中 R 和 S 的关系分块执行分区连接算法的分区步骤时,不将元组拷贝到对应分区中,而是将元组的内存地址存放在分区表(Partition Table,缩写为 PT)中,执行连接时根据分区表便可寻址到对应元组,完成连接操作.因为分区表的大小远小于关系大小,该思路避免了分区时的拷贝操作,能显著减少了对 NVM 内存空间的写次数,对于 NVM 较为友好.磁盘虚拟分区连接算法伪代码算法 5 所示.

算法 5. VRTPJ 算法伪代码.

```
输入:relation R,S, buffer M_R,M_S,PT_R,PT_S
输出:result of R joins S
FOR i=1 to |R|/|M_R| DO
  Read R′s tuples to fill M_R;
  VPartition(M_R,PT_R) //virtually partition M_R
  FOR i=1 to |S|/|M_S| DO
    Read S′s tuples to fill M_S;
    VPartition(M_S,PT_S) //virtually partition M_S
    //join each partition pair
    VPartitionJoin(M_R,M_S,PT_R,PT_S);
  END FOR
END FOR
```

算法的 VPartition 函数将一个位于内存的关系执行虚拟分区,并将分区结果记录在分区表中,随后的 VPartitionJoin 函数对虚拟分区后的关系执行 Cache 友好的连接,参见文献[6].文献[6]还研究了执行内存连接操作时的 Cache 缺失数量.

算法共有 4 个缓存区,M_R,M_S,PT_R(R 内存分块的分区表),PT_S(S 内存分块的分区表).与 BNJ 算法类似,我们可以分析出不同缓存区的平均读写次数,见表 7.这样我们便可以计算出不同缓存区的 C^g 值,从而不难得出:

$$C^g_{M_S}=C^g_{PT_S}>C^g_{M_R}=C^g_{PT_R}.$$

因此,最优部署方案将有限的内存优先分配给 M_S 和 PT_S,以期减少总内存读写代价.

表 7 VRTPJ 算法缓存区每个单位平均读写次数

	读写次数	
	读次数	写次数
M_R	$\frac{\|R\|}{\|M_R\|}$	$\frac{\|R\|}{\|M_R\|}$
M_S	$\frac{\|R\|\ \|S\|}{\|M_R\|\|M_S\|}$	$\frac{\|R\|\ \|S\|}{\|M_R\|\|M_S\|}$
PT_R	$\frac{\|R\|}{\|M_R\|}$	$\frac{\|R\|}{\|M_R\|}$
PT_S	$\frac{\|R\|\ \|S\|}{\|M_R\|\|M_S\|}$	$\frac{\|R\|\ \|S\|}{\|M_R\|\|M_S\|}$

5 性能评测

本节我们通过实验来验证最优部署方案及相应连接算法的性能.实验中我们对相同算法使用相同的配置、相同缓存区大小分配方案和 Cache 友好的内存连接算法(采用 Non-Partition Join 算法[27]),以保证对比的公平性.实验评测的方法为对比不同连接算法在最优映像部署方案(记为 Best)、完全不考虑两种异构内存的随机映像部署方案(我们假定

DRAM 和 NVM 页在逻辑内存空间中随机分布，因此每个缓存区占用同样比例的 DRAM，该比例即为混合内存中 DRAM 的占比，这种方式记为 Random）和最差映像部署方案（记为 Worst）三种不同数据结构部署方案下的算法运行时间和总的 NVM 写次数.

5.1　实验设置

我们使用 C++实现了第 4 节描述的 4 种磁盘连接算法. 程序使用 g++进行编译，并且开启 O3 优化选项以优化汇编代码质量，所有较小函数均采用内联的方式，以最大程度减少额外函数调用代价. 实验运行在一个主频为 3.0 GHz 的 6 核 Intel Core i5-8500 CPU 上. CPU 具有一个 9 MB 的 L3 Cache，每个核心拥有一个私有的 256 KB 的 L2 Cache 和一个 32 KB 的数据 Cache 和指令 Cache. 实验采用了 DDR4 的内存条，且连接负载均存储在 SSD 外存设备（设备型号为 Intel SSDPEKKW128G8）中，关系数据采用直接 I/O 的方式读入内存缓存区.

为了模拟出混合内存架构中的 NVM 内存空间，我们采用了与已有工作[11,31-32]相同的方法：在读写操作后面添加人工延时，延迟的粒度为 Cacheline，延时方法是空转 CPU 以模拟内存访问时的 CPU 停顿.

5.2　工作负载

实验中我们使用了与已有研究[33-35]相同的两种负载设定：

（1）关系 S 的大小是关系 R 的 10 倍. 该比例是 TPC-H 基准中的常见比例；在一般 OLAP 应用系统中，星型模式数据库的纬度表的规模会远小于事实表. 综上可知该负载可作为一般的连接算法负载设定.

（2）关系 R 的大小和关系 S 大小相近. 这种比例通常是散列连接算法中的最坏情况，因为该负载下构建散列表的代价相比较大，且算法的总 I/O 次数会相比前一种负载大大提高. 因此该负载可以作为最坏情况下的连接负载设定.

我们使用了一个 8 字节的键和 8 字节的值构成一个〈key，value〉元组配置，该配置是类似工作的[33-35]常用负载. 不失普遍性，我们假设关系 R 是两个关系中较小的，其主键作为连接键，关系 S 中包含了引用了 R 主键的一个外键，为了忽略次要因素，我们对关系键值采用均匀分布. 最后，我们在后续实验中使用了以下两种负载规模：

负载 1. 16 MB 的关系 R 与 160 MB 的关系 S.

负载 2. 64 MB 的关系 R 与 64 MB 的关系 S.

5.3　实验结果

实验主要考察以下两个指标：

（1）算法执行时间（Run Time）. 算法执行时间即为算法完成磁盘上两个关系的连接所花费的总时间（实验中具体数值为 10 次执行连接算法的平均耗时）. 数据库最耗时的操作便是连接算法，低延迟的连接算法是数据库系统的首要目标.

（2）NVM 写总数（NVM Writes）. 算法完成连接算法对混合内存中 NVM 的写总数. 由于 NVM 写代价较高，且具有写磨损的问题，减少对 NVM 写总数也同样是 NVM 内存算法研究的重要目标.

5.3.1　总体性能对比

我们统计了 4 种连接算法在 3 种不同部署方案下的执行时间和 NVM 写总数. 负载 1 和负载 2 上的实验结果分别如图 3 和图 4 所示. 实验中我们固定了可用内存大小为 16 MB，其中 DRAM 占 20%（即 DRAM 3.2 MB，NVM 12.8 MB），并且保证 NVM 写延迟为 300 ns，上述参数配置是出于以下几点考虑：

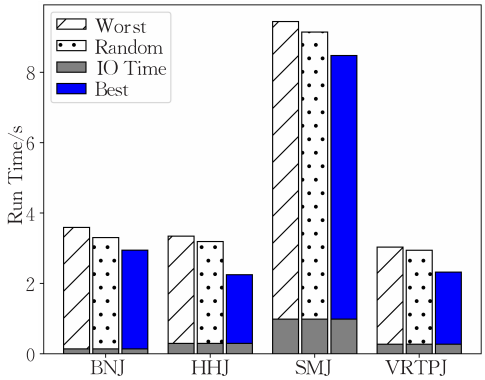
（1）保证磁盘关系不能被内存完全容纳，且连接算法运行时间不至于过长；

（2）算法执行时间与内存大小和关系大小的比例直接相关，与可用内存绝对大小不直接相关；

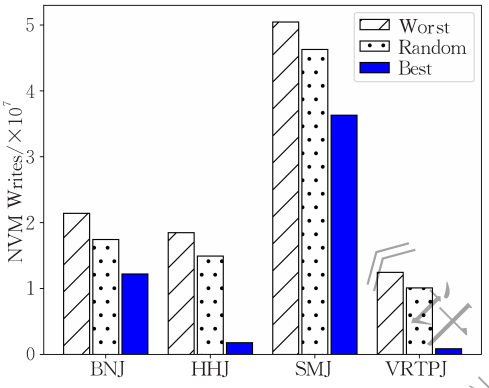
（3）主流 NVM 介质密度大于 DRAM，约为 DRAM 的 4 倍（见表 1）；

（4）持久性内存实验通常采用 300 ns 延时作为常用默认设置[32]；在后续的敏感性实验中，我们还分别研究了这些参数对实验结果的影响.

图 3(a)为四个算法在连接负载 1 过程中算法的总执行时间，图 3(b)为四个算法在连接负载 1 过程中对 NVM 的总写数量. 在执行时间上，我们额外统计了算法对 SSD 的 I/O 时间，该部分时间为算法在系统 I/O 接口 read 和 write 函数上花费的时间. 从图 3(a)我们可以看出，在执行时间上，最差部署方案＜随机部署方案＜最优部署方案，并且随机部署方案与最差方案结果较为相近. 实验数据指出，最优部署方案时间性能平均 1.28 x/1.22 x 于最差和随机部署方案. 从图 3(b)我们可以看出在 NVM 写总数上，最差部署方案＜随机部署方案＜最优部署方案. 相比最差最优部署方案/随机部署方案，最优部署方案至少能减少30%/22%的 NVM写总数.

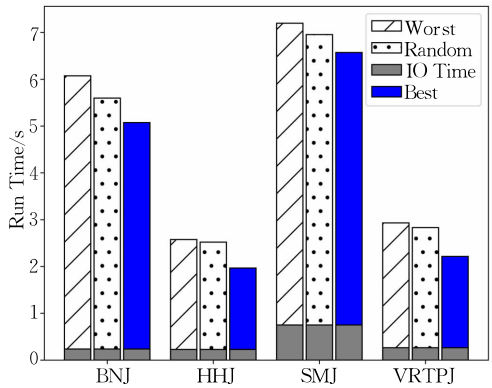


(a) 算法执行时间

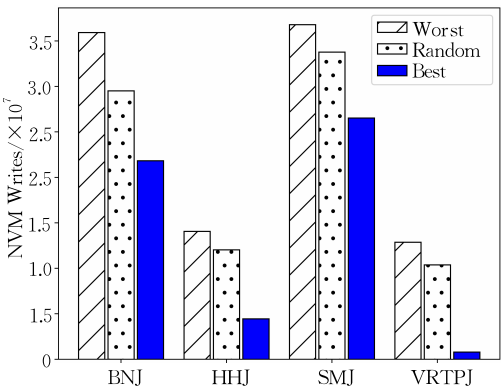


(b) NVM总写次数

图 3 4 种连接算法在负载 1 上的实验结果



(a) 算法执行时间



(b) NVM总写次数

图 4 4 种连接算法在负载 2 上的实验结果

值得一提的是,在虚拟分区算法(VRTPJ)下,NVM 写总数能减少高达94%/90%的NVM写数量. 综合(a)、(b)我们可以发现,最优方案下 NVM 写总数减少得越多,算法的时间性能提速更明显. 对比 I/O 时间可以发现,SMJ 算法 I/O 代价较高,这是因为其它算法的 I/O 基本均为顺序 I/O,而 SMJ 算法在归并过程中会引发大量的随机 I/O,因此 I/O 时间占比相对更高.

图 4(a)为四个算法在连接负载 2 过程中的总执行时间,图 4(b)为四个算法在连接负载 1 过程中对 NVM 的写总数. 我们可以看出,无论在执行时间还是 NVM 写总数上看,最优部署方案均一致优于随机部署方案,后者一致优于最差部署方案. 这不仅一定程度上证明了我们定理中最优最差部署方案的正确性,也表明了最优部署方案在不同负载下、不同算法下的有效性. 同样地,在负载 2 中,最优部署方案下的平均时间性能 1.28 x/1.18 x 于最差部署方案/随机部署方案,最优部署方案下平均能减少最优部署方案下 60%的 NVM 的写,平均能减少随机部署方案下 57%的 NVM 写. 此外,我们对比图 3、图 4 也能发现 BNJ 算法更适合一大一小的连接负载,而 HHJ 和 VRTPJ 算法则能够灵活应对不同负载.

我们在表 8 中记录了算法在连接负载 1 过程中,对 DRAM/和 NVM 的读写总次数. 通过对读写总数加和比较,我们发现,算法在不同部署方案下,读/写总次数是完全一致的. 这符合直观,因为映像部署方案并不改变算法是否读写内存,而是将算法对 DRAM 和 NVM 的读写做了一些重分配,使得内存访问代价最优.

表 8 4 种算法对负载 1 执行连接时对 DRAM 和 NVM 的读写次数统计表

		DRAM(×10^4)		NVM (×10^4)	
		Reads	Writes	Reads	Writes
BNJ	Worst	3125.2	40.1	3585.6	2142.1
	Random	1355.0	437.5	5355.9	1744.8
	Best	960.9	960.9	5750.0	1221.3
HHJ	Worst	2476.0	99.2	1067.6	1848.2
	Random	820.4	454.8	2723.3	1492.6
	Best	975.2	1769.1	2568.5	178.3
SMJ	Worst	119910.6	40462.7	10151.1	5046.4
	Random	119559.8	40879.6	10501.9	4629.5
	Best	119527.3	41878.1	10534.4	3630.9
VRTPJ	Worst	1665.4	12.4	3254.4	1245.5
	Random	1027.7	248.4	3892.1	1009.6
	Best	1385.1	1171.4	3534.6	86.5

5.3.2　混合内存总大小对性能的影响

在这一节中，我们研究了混合内存总大小发生变化时不同连接算法的性能，从而证明最优部署方案在总可用内存大小变化时的适应性. 在本次实验中，我们采用了控制变量的方法，控制其它敏感参数（DRAM 占比，NVM 写延迟，负载）保持不变，与 5.3.1 节一致，我们让总可用内存大小从 8 MB 变化到 20 MB. 相对于执行时间，NVM 总写次数更有说服力，因此我们本次实验考察指标为算法执行过程中对 NVM 的写总数，实验结果如图 5 所示.

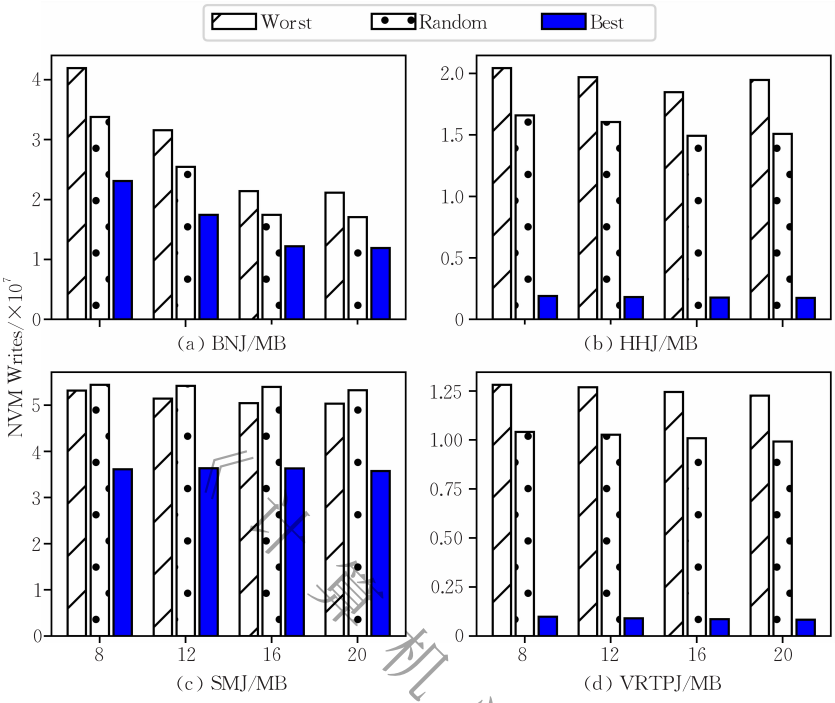


图 5　不同混合内存大小下 NVM 写次数对比

图 5 表明，最优部署方案在不同总可用内存条件下，均能大大降低对 NVM 的写总数. 特别地，在 HHJ 算法和 VRTPJ 算法中，NVM 写总数减少的比例更大. 观察每个子图中我们能发现，随着可用内存大小的增大，不同方案的 NVM 写总数均能一定程度地减少. 实验数据表明，随着可用内存从 8 MB 变化到 20 MB 时，BNJ/HHJ/SMJ/VRTPJ 算法下 NVM 写总数平均减少了 49%/7%/6%/5%. 此外，除 BNJ 算法以外，其它连接算法最优部署方案下的 NVM 总写次数随可用内存变化不显著，这是因为最优部署方案已将 C^g 值较大的缓存区存放在 DRAM 中，系统给其提供更多的内存只能减少 C^g 值稍小的缓存区的读写，因而增益不太明显. 综合来看，最优部署方案并不过分依赖于大内存来保证其高效性.

5.3.3　DRAM 占比对性能的影响

前述实验中，我们均假设 NVM 空间与 DRAM 的内存空间占比为 4∶1，这是因为 NVM 主流产品的密度通常是 DRAM 的 4 倍. 但是真实内存架构中，NVM 空间和 DRAM 空间的比例是可变的硬件配置参数，我们也希望操作系统能够根据不同进程的特点为不同进程分配不同比例的 NVM 和 DRAM 空间. 因此我们让 DRAM 占比可变，控制其它变量不变且与 5.3.1 节相同，并且同样采用负载 1 作为实验负载. 我们让 DRAM 占比从 0.1 变化到 0.4，统计不同占比下 NVM 总写次数，实验结果如图 6 所示.

从图 6 我们可以看出，在不同 DRAM 占比下，最优部署方案仍然给出了三种方案中最少的 NVM 写总数. 当 DRAM 占比从 0.1 变化到 0.4 时，最差部署方案的 NVM 总写次数基本不发生变化，但 SMJ 算法的 NVM 总写次数随着 DRAM 比例提高而显著减少. 此外，随机部署方案和最优部署方案的总 NVM 写次数在四种算法下都得到了显著的削减，其中 BNJ 算法效果最为明显，HHJ 算法的增益效果最差. 当 DRAM 比例为 0.1 时，最优方案能减少最差方案下算法 BNJ/HHJ/SMJ/VRTPJ 21%/89%/23%/88%的 NVM 写总数；当 DRAM 比例达到 0.4 时，最优方案下能减少最差方案下算法 BNJ/HHJ/SMJ/VRTPJ 89%/94%/44%/98%的 NVM

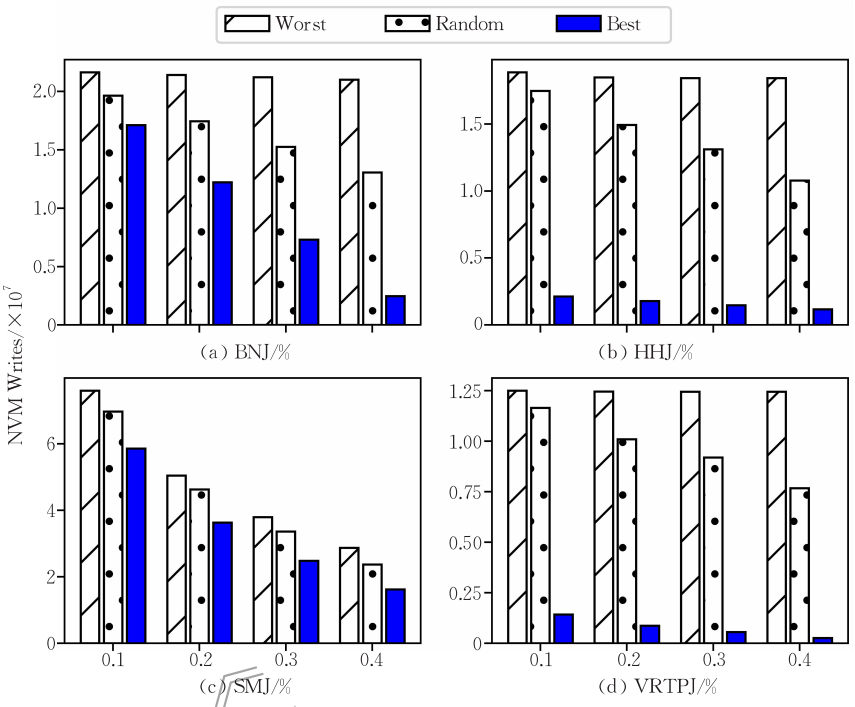


图 6 不同 DRAM 占比下 NVM 写次数对比

写总数.

5.3.4 NVM 写延迟对性能的影响

最后,我们探讨当 NVM 写延迟变化时连接算法的性能. 在本次实验中我们控制其它变量与 5.3.1 节完全相同,实验同样采用负载 1 作为连接负载. 我们让 NVM 写延迟从 300 ns 变化到 900 ns,同时考察最优部署方案带来的连接算法增益. 当 NVM 写延迟变化时,NVM 写总数并不会发生改变,因此本次实验我们使用算法完成连接的执行时间作为结果指标.

图 7 给出了实验结果. 在不同写延迟下,不同算法中,最优部署方案均保持执行时间指标上的最优,最差部署方案下执行时间仍然在三种方案中的最差,这进一步表明了映像部署模型的正确性.

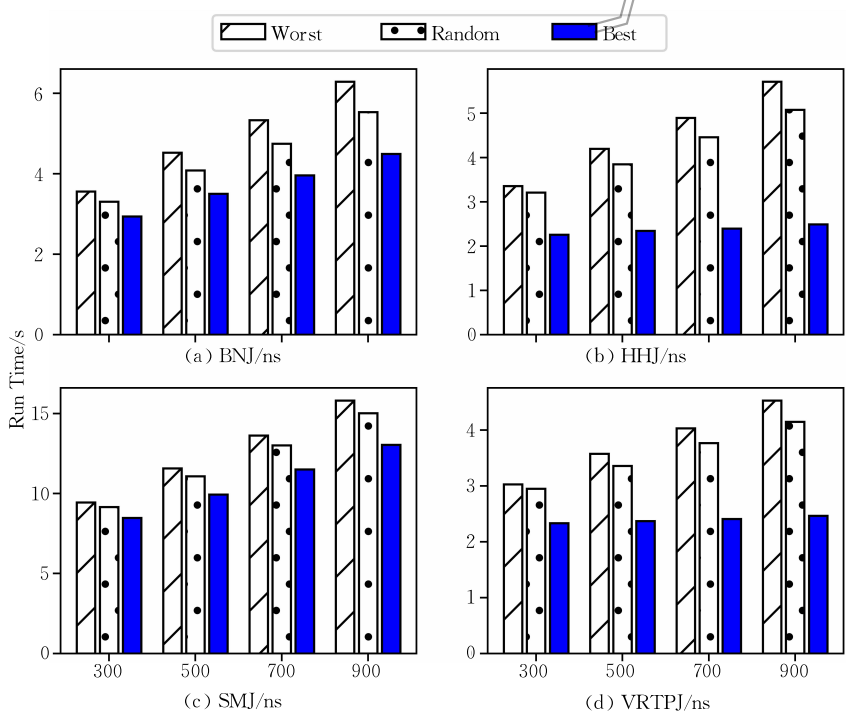


图 7 不同 NVM 写延迟下的运行时间对比

随着 NVM 写延迟从 300 ns 变到 900 ns，4 种算法在 3 种部署方案下的执行时间均变大，其中 BNJ/HHJ/SMJ/VRTPJ 算法执行时间平均增加了 66%/49%/59%/34%，这表明增加写延迟对于算法性能会有较大的影响. 但是对比最优部署方案下，BNJ/HHJ/SMJ/VRTPJ 算法的执行时间只增加了 52%/10%/53%/5%，一致低于平均值，这说明最优部署方案对于 NVM 写延迟容忍度较高. 最优部署方案不仅让算法执行时间最优，还吸收了较多的 NVM 写次数，使得算法能够在更高延迟的 NVM 设备中展现出更强的优势.

6　结束语

NVM 的出现使得基于 NVM 和 DRAM 的混合内存架构成为当前的研究热点. 混合内存架构给传统的数据库连接算法带来了很多挑战. 如何使得传统的数据库连接算法适合于混合内存架构？本论文针对这一问题研究了磁盘连接算法在混合内存架构下的优化思路. 论文提出了一个针对连接算法中间数据结构的映像部署模型，给出了不同读写比例的中间数据结构在不同部署方案下的内存代价理论分析，并给出了内存代价的上界和下界以及理论证明. 基于最优部署方案，我们提出了对嵌套循环连接算法(BNJ)、排序连接算法(SMJ)、散列连接算法(HHJ)和针对 NVM 架构的虚拟分区连接算法(VRTPJ)在混合内存架构下的内存代价优化策略. 实验表明，对于四种连接算法，本论文所提出的最优部署方案的能使时间性能平均提升 28%，平均减少 57%的 NVM 写总数.

论文还通过实验研究了不同敏感变量与算法时间代价、NVM 写总数的关系，该实验进一步证实了最优部署方案的有效性，且启发我们不同敏感变量对最优部署方案的优化：混合可用内存大小对于最优部署方案而言影响较小，但 DRAM 占比能显著提升最优部署方案减少 NVM 写总数的比例；NVM 写延迟会大大增加一般算法的延迟，但是最优部署方案对 NVM 写延时具有较好的适应性.

未来工作将主要集中在以下几个方向. 首先，本文的工作主要基于 DRAM+NVM 的平行架构，但理论上还存在着其它架构，例如 DRAM 作为 NVM 的缓存. 如果采用这些架构，连接算法应该如何优化？在未来工作中我们将围绕这一方向开展进一步地研究. 其次，本文研究的连接策略主要考虑了 NVM 的读写不对称性，没有考虑 NVM 的非易失性. 如何在算法层面将 NVM 的非易失性结合进来是未来值得探索的一个问题. 最后，本文提出的映像部署模型并没有考虑到负载和数据热度随时间而变化的问题. 自适应的查询优化机制一直是数据库查询优化研究中的挑战性问题，如何优化映像部署模型增强其对于负载和数据热度的自适应性将是我们下一步的一个主要研究方向.

参　考　文　献

[1] Boukhobza J, Rubini S, Chen R, et al. Emerging NVM: A survey on architectural integration and research challenges. ACM Transactions on Design Automation of Electronic Systems, 2018, 23(2): 14:1-14:32

[2] Shu Ji-Wu, Lu You-You, Zhang Jia-Cheng, Zheng Wei-Min. Research progress on non-volatile memory based storage system. Science & Technology Review, 2016, 34(14): 86-94 (in Chinese)
(舒继武，陆游游，张佳程，郑纬民. 基于非易失性存储器的存储系统技术研究进展. 科技导报，2016，34(14)：86-94)

[3] Chen K, Jin P, Yue L. A novel page replacement algorithm for the hybrid memory architecture involving PCM and DRAM //Proceedings of the 11th IFIP WG 10.3 International Conference (NPC 2014). Ilan, China, 2014: 108-119

[4] Driskill-Smith A. Latest advances and future prospects of STT-RAM//Proceedings of the 2010 Non-Volatile Memories Workshop. San Diego, USA, 2010: 11-13

[5] Gupta V, Kapur S, Saurabh S, et al. Resistive random access memory: A review of device challenges. IETE Technical Review, 2019, DOI: 10.1080/02564602.2019.1629341

[6] Chen S, Gibbons P B, Nath S. Rethinking database algorithms for phase change memory//Proceedings of the 5th Biennial Conference on Innovative Data Systems Research. Asilomar, USA, 2011: 21-31

[7] Dong X, Xu C, Xie Y, et al. NVSIM: A circuit-level performance, energy, and area model for emerging non-volatile memory. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2012, 31(7): 994-1007

[8] Condit J, Nightingale E B, Frost C, et al. Better I/O through byte-addressable, persistent memory//Proceedings of the 22nd Symposium on Operating Systems Principles. Big Sky Montana, USA, 2009: 133-146

[9] Wu Z, Jin P, Yang C, Yue L. APP-LRU: A new page replacement method for PCM/DRAM-based hybrid memory systems//Proceedings of the 11th IFIP WG 10.3 International Conference (NPC 2014). Ilan, China, 2014: 84-95

[10] Liu R, Jin P, Wu Z, et al. Efficient wear leveling for PCM/DRAM-based hybrid memory//Proceedings of the 2019 IEEE 21st International Conference on High Performance

Computing and Communications. Zhangjiajie, China, 2019: 1979-1986

[11] Viglas S. Write-limited sorts and joins for persistent memory. Proceedings of the VLDB Endowment, 2014, 7(5): 413-424

[12] Ma Zhu-Lin, Li Xin-Chi, Zhuge Qing-Feng, et al. A research of reducing write activities in multi-table join for non-volatile memories. Chinese Journal of Computers, 2019, 42(11): 2417-2428(in Chinese)
(马竹琳, 李心池, 诸葛晴风等. 面向非易失性存储器的多表连接写操作的优化研究. 计算机学报, 2019, 42(11): 2417-2428)

[13] Do J, Patel J. Join processing for flash SSDs: Remembering past lessons//Proceedings of the 2009 International Workshop on Data Management on New Hardware. Providence, Rhode Island, 2009: 1-8

[14] Wang Y, Jiang D, Xiong J. Caching or not: Rethinking virtual file system for non-volatile main memory//Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems. Boston, USA, 2018: 23-31

[15] Kwon Y, Fingler H, Hunt T, et al. Strata: A cross media file system//Proceedings of the 26th Symposium on Operating Systems Principles. Shanghai, China, 2017: 460-477

[16] Jiang De-Jun, Wang Ying, Xiong Jin. Storage system design and optimization towards persistent memory. Communications of the CCF, 2019, 15(1): 21-26 (in Chinese)
(蒋德钧, 王盈, 熊劲. 面向持久性内存的存储系统设计与优化. 中国计算机学会通讯, 2019, 15(1): 21-26)

[17] Wu Z, Jin P, Yue L. Efficient space management and wear leveling for PCM-based storage systems//Proceedings of the 15th International Conference on Algorithms and Architectures for Parallel Processing. Zhangjiajie, China, 2015: 784-798

[18] Lu You-You, Shu Ji-Wu. Persistent memory: From the system software perspective. Communications of the CCF, 2019, 15(1): 15-20(in Chinese)
(陆游游, 舒继武. 持久性内存: 从系统软件的角度. 中国计算机学会通讯, 2019, 15(1): 15-20)

[19] Liu R, Jin P, Wang X, et al. NVLevel: A high performance key-value store for non-volatile memory//Proceedings of the 2019 IEEE 21st International Conference on High Performance Computing and Communications. Zhangjiajie, China, 2019: 1020-1027

[20] Chen K, Jin P, Yue L. Efficient buffer management for PCM-enhanced hybrid memory architecture//Proceedings of the 17th Asia-Pacific Web Conference on Web Technologies and Applications. Guangzhou, China, 2015: 29-40

[21] Li L, Jin P, Yang C, et al. XB+-tree: A novel index for PCM/DRAM-based hybrid memory//Proceedings of the 27th Australasian Database Conference on Databases Theory and Applications. Sydney, Australia, 2016: 357-368

[22] You Li-Tong, Wang Zhen-Jie, Huang Lin-Peng. A log-sructured key-value store based on non-volatile memory. Journal of Computer Research and Development, 2018, 55(9): 2038-2049(in Chinese)
(游理通, 王振杰, 黄林鹏. 一个基于日志结构的非易失性内存键值存储系统. 计算机研究与发展, 2018, 55(9): 2038-2049)

[23] Li L, Jin P, Yang C, et al. Optimizing B+-tree for PCM-based hybrid memory//Proceedings of the 19th International Conference on Extending Database Technology. Bordeaux, France, 2016: 662-663

[24] Tu C, Yeh M, Kuo T. A fast non-volatile memory aware algorithm for generating random scale-free networks//Proceedings of the 2017 IEEE International Conference on Big Data. Boston, USA, 2017: 787-796

[25] Ramos L, Gorbatov E, Bianchini R. Page placement in hybrid memory systems//Proceedings of the 2011 International Conference on Supercomputing. New York, USA, 2011: 85-95

[26] Hassan A, Vandierendonck H, Nikolopoulos D. Software-managed energy-efficient hybrid DRAM/NVM main memory //Proceedings of the 12th ACM International Conference on Computing Frontiers. New York, USA, 2015: 23:1-23:8

[27] Garcia-Molina H, Ullman J, Widom J. Database System Implementation. NJ: Prentice Hall, 2000

[28] Haas L, Carey M, Livny M, et al. Sing the truth about ad hoc join costs. The International Journal on Very Large Data Bases, 1997, 6(3): 241-256

[29] Manegold S, Boncz P, Kersten M. What happens during a join? Dissecting CPU and memory optimization effects//Proceedings of the 26th International Conference on Very Large Data Bases. Cairo, Egypt, 2000: 339-350

[30] Balkesen C, Teubner J, Alonso G, et al. Main-memory hash joins on multi-Core CPUs: Tuning to the underlying hardware //Proceedings of the 2013 IEEE 29th International Conference on Data Engineering. Brisbane, Australia, 2013: 362-373

[31] Zhang D, Jin P, Wang X, et al. DPHSim: A flexible simulator for DRAM/PCM-based hybrid memory//Proceedings of the 1st International Joint Conference on Web and Big Data. Beijing, China, 2017: 319-323

[32] Hwang D, Kim W, Won Y, et al. Endurable transient inconsistency in byte-addressable persistent B+-tree//Proceedings of the 16th USENIX Conference on File and Storage Technologies. Oakland, USA, 2018: 187-200

[33] Lang H, Leis V, Albutiu M C, et al. Massively parallel NUMA-aware hash joins//Proceedings of the 2013 International Workshop on In-Memory Data Management and Analytics. Rivadel Garda, Italy, 2013: 1-12

[34] Balkesen C, Alonso G, Teubner J, et al. Multi-core, main-memory joins: Sort vs. hash revisited. Proceedings of the VLDB Endowment, 2013, 7(1): 85-96

[35] Schuh X, Dittrich J. An experimental comparison of thirteen relational equi-joins in main memory//Proceedings of the 2016 International Conference on Management of Data. New York, USA, 2016: 1961-1976

LUO Yong-Ping, M. S. candidate. His research interests include NVM-based database systems and reinforcement learning.

JIN Pei-Quan, Ph. D., associate professor. His research interests include database systems on new hardware and moving objects databases.

Background

This paper is mainly focused on a new field in the research on database systems, namely NVM-based database systems. Non-Volatile Memory (NVM) has been regarded as the main alternative of next main memory technologies to replace DRAM, due to its advantages of byte addressability, non-volatility, high density, and low power consumption. The integration of NVM in the architecture of modern computer systems introduces many challenging issues, because current operating systems as well as database management systems are designed towards DRAM and hard disks.

Although NVM is currently faster than flash memory, it has higher access latency than DRAM. In addition, the read/write latency of NVM is unbalanced. Therefore, considering all factors involving memory performance, storage density, and non-volatility, it is a feasible solution to construct a hybrid memory system based on NVM and DRAM. Based on such hybrid memory architecture, this paper investigates the optimization of join algorithms on the hybrid memory. We noticed that the intermediate data structures of a join algorithm have different read/write operations. Thus, we can reduce memory cost of join algorithms on the hybrid memory by designing an appropriate way to place intermediate data structures on NVM or DRAM. Following this idea, we propose a formal Data Structure Placement Model (DSPM) to address this problem. Then, we implemented four join algorithms based on the optimal DSPM model and conduct comparative experiments to verify performance of our proposal. The experimental results suggest the efficiency and effectiveness of the proposed approach.

This work is supported by the National Science Foundation of China under Grant No. 61672479.