The Five-Minute Rule 20 Years Later

作者：Goetz Graefe. 2009
原文：http://www.hpl.hp.com/techreports/tandem/TR-86.1.pdf
译者：phylips@bmy 2012-09-03
译文：http://duanple.com/?p=1058

[

序：

]

摘要

1987 年，Jim Gray 和 Gianfranco Putzolu 提出了他们流行至今的用于权衡内存和 IO 性能的五分钟法则。他们进行了一系列的计算，针对特定价格的 RAM 和磁盘，将一条记录(一个 page)一直保存在内存的开销与每次都执行一次磁盘 IO 的开销进行了对比，并以可以达到平衡的那个访问时间间隔(5 分钟)命名了该法则。如果一条记录(一个 page)访问地足够频繁，那么就应该把它放到内存，否则就应该放到磁盘，在需要的时候再读取。

基于当时 Tandem 设备的价格和性能，Gray 和 Putzolu 发现使用 RAM 去存放 1KB 大小的记录的开销，大概相当于在磁盘上每隔 400 秒访问同样大小记录的开销。随着记录大小的增大，这个时间间隔会变小。Gray 和 Putzolu 指出，对于 100 字节大小的记录来说，这个时间间隔是一小时，对于 4KB 大小的记录来说就变成了 2 分钟。

在 1997 年，也就是该法则提出后的 10 年，Gray 和我回顾并更新了该法则。那时，很多价格和性能参数发生了变化(比如，每 MB RAM 的价格已经从$5,000 下降到$15)。然而针对 4KB page 来说，可以达到平衡的那个访问时间间隔仍然是 5 分钟左右。本文的一个目的就是在经历了另一个 10 年后，再重新审视下 5 分钟法则。

当然，之前的那两篇文章(1986 和 1997 年的那两篇)也已指出，由于技术和设备的不同，价格和性能也会差距比较大(比如用于大型机和用于 PC 的 RAM，SCSI 磁盘与 IDE 磁盘，等等)。感兴趣的读者，可以根据自己的环境和设备列出对应的公式，重新进行计算。表 1 里列出的值都是主要针对 2007 年技术情况的，而不是一个统一值。

Table 1: Prices and performance of flash and disks.

	RAM	Flash disk	SATA disk
Price and capacity	$3 for 8×64Mbit	$999 for 32GB	$80 for 250GB
Access latency		0.1ms	12ms average
Transfer bandwidth		66MB/s API	300MB/s API
Active power		1W	10W
Idle power		0.1W	8W
Sleep power		0.1W	1W

Table 2: Relative costs for flash memory and disks.

	NAND Flash	SATA disk
Price and capacity	$999 for 32GB	$80 for 250GB
Price per GB	$31.20	$0.32
Time to read a 4KB page	0.16ms	12.01ms
4KB reads per second	6,200	83
Price per 4KB read per second	$0.16	$0.96
Time to read a 256KB page	3.98ms	12.85ms
256KB reads per second	250	78
Price per 256KB read per second	$3.99	$1.03

除了在成本和性能方面量上的变化，已出现的一些质的改变也将会影响到服务器以及尤其是数据库系统的软硬件架构。随着这些新技术的出现，数据库软件将会发生根本的改变：虚拟化的软硬件支持，并伴随着对物理机利用率的提高；多核处理器以及在编程环境和硬件层面的 transactional memory 支持；可以容纳数千处理器和 TB 级数据的数据中心；填补了传统 RAM 和磁盘间差距的 flash memory。

在采购成本，访问延时，传输带宽，空间密度，电力消耗和制冷开销等方面上，flash memory 刚好位于传统 RAM 和磁盘之间。通过表 1 和表 2 中的某些参数可以看出这一点(所有参数来源于 dramexchange.com，dvnation.com，buy.com，seagate.com 和 samsung.com 在 2007-11-04 日的网页内容)。

在一次磁盘 IO 的时间内所能执行的 CPU 指令稳步增长的情况下，存储体系中就需要在二者之间有一种中间性的存储设备，flash memory 看起来是一个不错的选择，同时目前为止它也引起了人们的广泛关注。

但是仍然有很多架构细节需要解决。比如，在硬件架构上，flash memory 是通过一个 DIMM slot，还是一个 SATA 磁盘接口，亦或是另一种其他的硬件接口呢？如果需要定义一种新的硬件接口的话，那么就可能需要对现有接口进行调整。

最主要的问题是，flash memory 是应该看做是主存还是持久性存储的一部分呢？也可以这样问：如果某系统有 1GB 的传统 RAM，8GB 的 flash memory 和 250GB 的传统磁盘，那么软件系统是应该把它们看做是 250GB 的持久性存储加上 9GB 的缓存，还是 258GB 的持久性存储加上 1GB 的缓存呢？本文的第二个目的就是要解答下这个问题，实际上，对于文件系统和数据库系统来说，答案并不是一样的。

很多设计决定都会依赖于针对这个问题的答案。比如，如果 flash memory 是缓存的一部分，那么当 pages 的内容与持久性存储中的对应 page 不同时，它们就应该被认为是 dirty 的。在对文件系统进行 sync 或者对数据库系统进行 checkpoint 时，必须将它们强制写入磁盘。而如果把 flash memory 看做是持久性存储，那么这些写入操作就都是不需要的。

操作系统和文件系统的设计者们会把 flash memory 当成一个扩展缓存 (扩展 RAM) 来用，但是数据库系统把 flash memory 当成扩展磁盘(扩展持久性存储)会更好。已经有多种文件系统和数据库系统采用了这两个设计策略，提供一些这方面的实例就是本文的第三个目的。

最后，flash memory 本身的这些特点要求对 B-tree pages 的管理和分配进行实质性的改变。除了优化 page 大小之外，B-tree 本身可以为 flash memory 和磁盘使用不同的 IO 单元。这样两个不同的 page 大小，将会对应两个新的五分钟法则。引入这两个新的法则就是本文的第四个目的。

一些假设

比较具有前瞻性的研究通常都会基于一些假设。这一节就列出导出了本文结论的那些假设。在这些假设中，其中的一些很基本，其他的一些就比较具有推测性。

其中一个假设是文件系统和数据库系统会分配一样的数据给 flash memory。这两个软件系统都是更倾向于将那些未来会有一定概率但不保证会被访问的页面放入 RAM，通常会使用诸如 LRU 这样的方式进行管理。

我们假设管理这种信息所使用的数据结构位于 RAM，即使页面已经从 RAM 删除而放入了 flash memory。比如，一个文件系统的 LRU 链通常会同时包含 RAM 和 flash memory 里的内容，或者是为它们分别建一个 LRU 链。当一个页面被应用需要时，它会被 load 到 RAM 并插入到第一个 LRU 链的头部。当它到达第一个 LRU 链尾部的时候，该页面会被移入 flash memory，同时它对应的描述符也会被移入到第二个 LRU 链的头部。当它到达第二个 LRU 链的尾部的时候，该页面就会被移入磁盘，同时移出 LRU 链。也可以使用其他的替换算法，只是细节上会有所不同。

这种页面级的细粒度 LRU 替换策略，与整个文件，目录，数据库表或者是针对不同存储单元的数据库级别上的不同。看起来，页面级的替换更适合缓存池。此外，现有的经过验证的用于加载和替换缓存池内容的方法都是完全自动的，无需 tuning 工具或者是人工参与。一个使用了 flash memory 进行扩展后的缓存池，也应该使用与传统缓存池相同的方法。从性能和管理开销上看，当 flash memory 被用来扩展磁盘时，使用类似的方法也是不错的选择。

文件系统。我们的研究是基于一种非常传统的文件系统。尽管可能很多文件系统与此模型不同，但是大部分都还基本跟它一致。在我们的传统系统中，每个文件是一个很大的字节流。文件经常会被整个读取，它们的内容是在内存中被操纵的，如果它被更新了，整个文件会被替换掉。存档，版本保留，分层的存储管理，数据移动到可移动介质，等等，看起来都符合该模型。针对这个模型，磁盘上的存储空间分配会尝试为每个文件分配连续的磁盘块。元数据只包括目录，一些标准 tags，比如创建时间，以及用于空间管理的数据结构。

这些磁盘上的数据结构的一致性，主要通过仔细的 write ordering，已更新数据块的快速写回，和非正常关机或设备移除导致的文件系统检查来达到的。话句话说，我们假设没有事务性的保证和日志机制，至少对于文件内容是这样的。如果支持对文件内容比如单个的页面或者是页面里的记录，进行基于日志的恢复，那么本文提出的很多观点都需要重新考虑。

数据库系统。我们针对的是非常传统的数据库系统，使用 B-树索引作为底层存储结构。类似的树结构不仅包含传统的聚簇和非聚簇索引，也包括 bitmap 索引，列式存储，内容索引，XML 索引，元数据和 allocation 数据结构。

关于事务支持方面，我们假设会为内容变更(比如记录的插入和删除)和结构变更(比如 B-树节点的分裂)提供传统的 write-ahead logging 机制支持。可以在出错之后，高效地基于日志和 checkpoint 进行恢复，chekpointing 时会将缓存池中的脏数据强制刷到持久性存储中。

另外的一些变种，比如”second-chance”或者是 fuzzy checkpoint 也适用于我们的假设。此外，也允许对某些操作比如索引创建，不记录日志。这些操作需要相应地 write ordering 和”force”缓存池策略。

Flash Memory。假设硬件和设备驱动已经隐藏了很多实现细节，比如针对 flash memory 的特殊硬件接口。比如，flash memory 可能是被嵌入到计算机的主板上，通过 DIMM 槽访问，或者被包装成一个标准磁盘。无论是哪种情况，DMA 传输都是假设在 RAM 和 flash memory 之间进行。此外我们还假设，要么是在 flash memory 和磁盘之间存在一个高效的 DMA 数据传输通道，要么是在 RAM 上有一个传输 buffer。这个传输 buffer 的大小大概应该是传输带宽和磁盘延迟的乘积。另外如果要求磁盘写操作不能影响读操作的话，还需要将写操作带来的延迟影响考虑在内。

另一个假设是，flash memory 和磁盘的传输带宽差不多的。Flash 的写入带宽落

后于读取带宽，有些产品声明两者的差距在 2 倍以内。如果必要，可以通过 array arrangement 来提高传输带宽。

由于当前 NAND flash 可以承受 100000-1000000 次的擦写，我们假设会采用一些机制来进行 wear leveling(穿平，平均抹写磁盘区块)。这些机制会保证所有的页面或者数据块被均匀的写入数据。认清 wear leveling 算法和 log-structured 文件系统间的相似性是非常重要的，比如 wear leveling 算法也会移动那些处于稳定状态的，未发生变更的数据，这样也可以消耗掉它们所处的位置一次擦写 circle。

需要注意的是，由于各方面的原因，传统磁盘所支持的写操作数有限。比如，就算是连续不断地以 100Mbps 的带宽覆写一个 250GB 磁盘，六年的时间里也最多能覆写 80,000 次。换句话说，假设一个 log-structured 文件系统是基于 RAID-5 或 RAID-6 的，如果用当前 flash 来做底层的存储设备，从可靠性上来看也是很有竞争力的。类似的，持续以 30Mbps 的平均带宽去覆写一个 flash 磁盘 100,000 次，大概需要花 3.5 年。

除了 wear leveling 外，我们假设有一个异步的 agent 将数据从 flash memory 移动到磁盘，同时会立刻擦除 flash memory 里释放出来的空间，避免写操作因为需要擦除操作而被 delay。这个行为在 log-structured 文件系统也有一个与之等价的，用来为为了的 log 写入准备空间的清除动作。区别在于，它只需要移动磁盘内容，但是 flash 在下一次写再次写到这个位置上时，必须先擦除它上面的内容。

无论是文件系统还是数据库系统，我们都假设为 page tracking 和 page replacement 提供了各自独立的机制。比如一个提供了这两种机制的传统的缓存池，但是它会为这两个不同的目的使用不同的数据结构。标准做法是会为 page replacement 使用 LRU list，为 page tracking(即，记录 page 是否在缓存池以及在缓存池的哪个部分)使用 hash 表。

假设替换算法所使用的数据结构是很小的，而且经常被访问到，因此它会被放到 RAM 里。我们也假设 page tracking 信息也必须像数据一样被持久化，包括空闲空间信息。因此，缓存池的 hash 表会在系统重启时重新初始化，但是 tracking 信息必须跟数据一块存到持久化存储设备上。page tracking 信息可能会被缓存到 RAM，但是针对它的任何变更必须要记录日志，同时写回到持久化存储中。用来进行页面位置查找的索引可能是只存在于 RAM 中，每次打开的时候重建，同时将 tracking 信息加载到 RAM 中。

如前所述，我们假设 page replacement 是需要的，此外，也可能存在一些针对 prefetch，read-ahead 和 write-behind 的自动化策略和机制。

基于这些考虑，我们假设 flash memory 的内容是非常一致的，无论 flash memory 是作为缓存池还是磁盘的扩展。因此核心问题不是把什么放到 cache 中，而是如何管理 flash memory 的内容及其生命期。

在数据库系统中，flash memory 也可以用于保存恢复用日志，因为它的访问时间

很短可以比较快地完成事务提交。但是，写入带宽方面的限制使得这种用法并不常见。或许系统可以通过保存双份 log 而同时获取低延迟和高带宽，将一份放在传统磁盘上，一份放到 flash 上，只要写入操作落到 flash 上就认为事务提交完成。

其他硬件。在所有情况中，我们都假设 RAM 是足够大的，尽管它可能要比 flash memory 和磁盘都要小。它们间的相对大小关系需要根据 5 分钟法则进行调整。需要注意的是，尽管 flash memory 与磁盘的传输带宽差不多，但是它相对短的访问延迟可能会导致令人吃惊的数据在 RAM 上的滞留次数。

最后，我们假设现代多核处理器提供了足够高的处理带宽。此外，即将到来的 transactional memory(硬件和运行时系统上的)将能够支持对复杂数据结构进行高并发操作。比如，page replacement 可能会使用优先级队列而不是 bitmap 或链表实现。类似地，高级地锁管理机制也可能通过使用更复杂的数据结构而获益。虽然如此，我们并不假设或需要使用那些比目前普遍用于 page replacement 和 locaticon tracking 的更复杂的数据结构。

5 分钟法则

如果 flash memory 被引入到当前内存体系结构中的话，那么各级存储的大小就需要重新地考虑了。

可以基于购置成本，维护开销，耗电量，平均出错时间，平均数据丢失时间，或者一系列参数来进行 tuning。像 Gray 和 Putzolu 那样，本文会关注在购置成本上。如果考虑其他参数的话，公式都是类似的，很容易能推导出来。

Gray 和 Putzolu 提出了如下公式：
BreakEvenIntervalInSeconds=(PagesPerMBofRAM)/AccessesPerSecondPerDisk*(Price-PerDiskDrive/PricePerMBofRAM)。

该公式是通过将一个 page 放到 buffer pool 里的开销与每当 page 被需要时磁盘执行一次 IO 的开销，令它们相等然后求解，得出的两次访问间的时间间隔。

假设采用现代 RAM，而磁盘驱动器使用 4KB 大小的 page，同时采用表 1 和表 2 中的数据，代入上式得到：
(256 / 83)×($80 / $0.047) = 5,248 seconds ≈ 90 minutes = 1.5 Hours
而 20 年前，这个值是 2 分钟(同样针对 4KB page 来说)。这个变化还是很大的，增加了接近两个数量级。在 1987 年，RAM 大概是 5000$/MB，但是到了 2007 年，低到了大概 0.05$/MB，差了 5 个数量级。另一方面，磁盘价格也在下跌(在 1987 年一个磁盘大概 15000$)，同时磁盘延迟和带宽也有显著提高(从单磁盘每秒访问次数从 15 提升到了 100，高性能企业级磁盘甚至达到了 200)。

对于 RAM 和 32GB 的 flash 磁盘来说，该平衡间隔是：

(256 / 6,200)×($999 / $0.047) = 876 seconds ≈ 15 minutes
如果说 2007 年 flash 磁盘再继续降价，比如降到 400$--那么这个平衡间隔就变成了 351 秒，大概是 6 分钟。

一个重要的结论是，对于一个考虑经济成本的系统来说，如果次级存储设备采用的是 flash 而不是磁盘的话，RAM 的利用率将会加快 15 倍(90 分钟/6 分钟)。需要的 RAM 越少，相应的采购，制冷，供电成本就越少。{!很明显 flash 需要的 RAM 更少，因为只有那些访问频率高于 6 分钟一次的数据才需要放入 RAM，而 disk 则需要将频率高于 90 分钟一次的就要放入 RAM}

或许，更有趣的是如果我们将 flash 和磁盘代入到该公式，可以得到如下结论：
(256 / 83)×($80 / $0.03) = 8,070 seconds ≈ 2.25 hours
因此，所有的活动数据就都会停留到RAM和flash中了。毫无疑问，2小时已经超过任何的checkpoint间隔了，这就意味着flash中的脏页会通过checkpoint而不是换页被强制flush到磁盘中。Pages that are updated frequently must be written much more frequently (because of checkpoints) than is optimal based on Gray and Putzolu's formula.

在 1987 年，Gray 和 Putzolu 就展望过未来 20 年的情况，同时为 RAM 和 disk 做出一个“5-hour rule”的预测。对于 1KB 的记录，根据 2007 年的价格和相关参数，大概是 20978 秒，刚好小于 6 小时。他们当时的预测出乎意料的准确。

对于不同的 page 大小平衡间隔也是不同的。表 3 展示了针对一系列不同 page 大小和不同存储技术的平衡间隔。(“$400”的 32GB NAND flash 驱动器目前还未出现，2007 年的实际价格是 999$，实际上到了 2009 年才出现售价$400 的 32GB SLC SATA 驱动器)

Table 3: Break-even intervals [seconds].

Page size	1KB	4KB	16KB	65KB	256KB
RAM-SATA	20,978	5,248	1,316	334	88
RAM-flash	2,513	876	467	365	339
Flash-SATA	32,253	8,070	2,024	513	135
RAM-$400	1,006	***351***	187	146	136
$400-SATA	80,553	20,155	5,056	1,281	***337***

关于 RAM 和磁盘的老 5 分钟法则，目前可以应用到 64KB 的 page 大小上(334 秒)。在 1987 年，5 分钟这个平衡间隔对应的是 1KB 的，1997 年则是 8KB 的。这种趋势反映了磁盘访问次数和传输带宽之间的不均衡发展。

5 分钟法则也可以应用到 RAM 和 page 大小为 64KB 及更大的 2007 年的昂贵 flash memory 上(365 秒和 339 秒)。随着 flash memory 价格的下降，平衡间隔也在下降(146 秒和 136 秒)。两个新的 5 分钟法则在表 3 中用黑体标示出来了，后面在

讨论 B-树索引的最优节点大小时我们还会再来看下表 3。

Page Movement

除了进出 RAM 的 I/O 外，在 flash，内存和磁盘之间还需要一个三级内存体系结构。

用于页面移动的机制可以应用在硬件领域(比如，DMA transfer)，或者它可能需要通过 RAM 进行中转。前者是为了获得更好的性能，但是后面的设计则完全可以在软件层面实现。另一方面，混合的磁盘制造方式还可能可以降低硬件成本。

页面移动策略是由demand-paging和LRU替换导出或控制的。如前所述，文件系统和数据库系统中的替换策略可能基于LRU，同时也可以通过采用RAM里合适的数据结构来实现。就像RAM里的buffer管理，可能会由于prefetch，read-ahead和write-behind等的存在而存在很多不同。在数据库系统中，这些可能都是由查询执行层决定的，而文件系统则必须要检测page访问模式，worthwhile read-ahead actions without the benefit of such hint。

如果flash memory是持久化存储的一部分，那么flash memory与磁盘之间的页面移动就类似于defragmentation时的页面移动，对于文件系统和数据库系统都是这样。在这两种系统中，最重要的不同是页面如何移动和当前的页面如何被追踪的。

Tracking Page Locations

文件系统与数据库系统中的页面追踪机制非常不同。在文件系统中，由指针页面负责追踪数据页面，或者使用的是邻接的数据页面。页面的移动可能需要中断一下正常运行。通常需要对它进行更新，并修改指针页面。

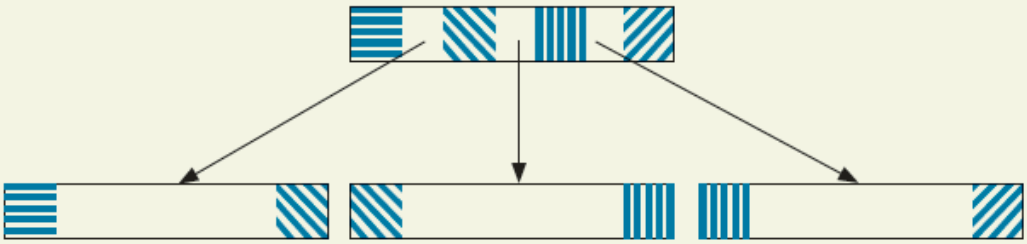
在数据库系统中，大多数数据存储在B-树索引中，包括表的聚簇及非聚簇索引，物化视图和数据库元数据。位图索引，列存储和master-detail clustering可以方便高效地利用B-树进行表示。也可以用一些源于B-树的树形结构来存储blobs(二进制大对象)，有些类似于某些文件系统中的存储结构[5,25]。

对于B树来说，移动一个页面可能非常昂贵也可能非常简单。最高效的机制通常可以在用于defragmentation或reorganization的工具中找到。B-树实现的成本和效率主要取决于两点—邻居指针的维护和用于故障恢复的日志机制。

首先，如果每个B-树页面都维护着物理邻居指针，单个页面的移动，除了父节点外还需要对两个邻居进行更新。如果邻居指针是采用fence keys的，那么页面移动时只需要对父页面进行更新[10]，图2就展示了这样一种B-树。父页面中的指针变更会被记入日志，但是不需要立即将日志强制写出，因为这只是改变了内部结

构并未改变数据库内容。

Figure 2: A write-optimized B-tree with fence keys instead of neighbor pointers.



参考文献

1. Ailamaki, A., DeWitt, D.J. and Hill, M.D. Data page layouts for relational databases on deep memory hierarchies. VLDB Journal 11, 3 (2002), 198 – 215.
2. Bender, M.A. Demaine, E.D. and Farach-Colton, M.Cache-oblivious B-trees. SIAM Journal on Computing 35, 2 (2005), 341 – 358.
3. Bayer, R. and McCreight, E.M. Organization and maintenance of large ordered indexes. SIGFI-DET Workshop (1970), 107 – 141.
4. Bayer, R. and Unterauer, K. Prefix B-trees. ACM Transactions on Database Systems 2, 1 (1977), 11 – 26.
5. Carey, M.J., DeWitt, D.J., Richardson, J.E. and Shekita,E.J. Storage management in EXODUS. In Object-Oriented Concepts, Databases, and Applications. W.Kim and F. Lochovsky, Eds. ACM, N.Y., 1989, 341 – 369.
6. Chen, P.M., Lee, E.L. Gibson, G.A., Katz, R.H. and Patterson,D.A. 1994. RAID: high-performance, reliable secondary storage. ACM Computing Surveys 26(2): 145 – 185.
7. DeWitt, D.J., Naughton, J.F. and Burger, J. Nested loops revisited. Parallel and Distributed Information
Systems (1993), 230 – 242.
8. Graefe, G. Query evaluation techniques for large databases. ACM Computing Surveys 25, 2 (1993), 73 – 170.
9. Graefe, G. Executing nested queries. Database Systems for Business, Technology and Web (2003), 58 – 77.
10. Graefe, G. Write-optimized B-trees. VLDB Journal (2004), 672 – 683.
11. Graefe, G. Implementing sorting in database systems.ACM Computing Surveys 38, 3 (2006), 69 – 106.
12. Graefe, G. Master-detail clustering using merged indexes. Informatik – Forschung und Entwicklun, 2007.

13. Gray, J. and Fitzgerald, B. 2007. Flash disk opportunity for server-applications; http://research.microsoft.com/~gray/papers/FlashDiskPublic.doc.
14. Gray, J., Graefe, G. 1997. The five-minute rule ten years later, and other computer storage rules of thumb. SIGMOD Record 26, 4 (1997), 63–68.
15. Gray, J. and Putzolu, G.R. The 5-minute rule for trading memory for disk accesses and the 10-byte rule for trading memory for CPU time. SIGMOD Journal (1987), 395–398.
16. H arder, T. Implementing a generalized access path structure for a relational database system. ACM Transactions on Database Systems 3, 3 (1978), 285–298.
17. H amilton, J. An architecture for modular data centers.In Proceedings of the Conference on Innovative Data Systems Research, 2007.
18. H arder, T. and Reuter, A. Principles of transactionoriented database recovery. ACM Computing Surveys 15, 4 (1983), 287–317.
19. Lomet, D.B. The evolution of effective B-tree page organization and techniques: a personal account.SIGMOD Record 30, 3, 64–69.
20. Larus, J.R. and Rajwar, R. Transactional Memory.Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.
21. Nyberg, C., Barclay, T., Cvetanovic, Z., Gray, J. and Lomet, D.B. AlphaSort: A cache-sensitive parallel external sort. VLDB Journal (1995), 603–627.
22. Ousterhout, J.K. and Douglis, F. Beating the I/O bottleneck: A case for log-structured file systems.Operating Systems Review 23, 1 (1989), 11–28.
23. O’ Neil, P.W. The SB-tree: An index-sequential structure for high-performance sequential access. Acta Informatica 29, 3 (1992), 241–265.
24. Rivoire, S., Shah, M., Ranganathan, P. and Kozyrakis, C.JouleSort: A balanced energy-efficiency benchmark.SIGMOD Record, 2007.
25. Stonebraker, M. Operating system support for database management. Commun. ACM 24, 7 (July 1981), 412–418.
26. Shatdal, A., Kant, C. and Naughton, J.F. Cacheconscious algorithms for relational query processing.VLDB Journal (1994), 510–521.
27. Woodhouse, D. JFFS: The Journaling Flash File System. Ottawa Linux Symposium, Red Hat Inc., 2001.