How Flash Memory Changes the DBMS[1] World
An introduction

Hans Olav Norheim

Department of Computer and Information Science
Norwegian University of Science and Technology (NTNU)
Trondheim, Norway

hansolno -at- stud.ntnu.no

April 22nd, 2008

Abstract

Most of today's DBMSs rely heavily on disk subsystems for persistent storage of data and transaction logs. Both in terms of price and performance, the gap in the memory hierarchy between main memory and disk storage is a rather huge one. In the latest years, flash memory – which appears to fill this gap – has started to gain market. This paper looks at what the flash technology offers, how it affects database applications, as well as how to exploit it in the best possible way.

[1] Database Management System

Table of Contents

1 Introduction ... 3
2 Characteristics of NAND Flash Memory ... 4
2.1 No In-Place Updates ... 4
2.2 No Mechanical Latency ... 4
2.3 Limited Number of Writes ... 4
2.4 Asymmetric Speed of Read/Write ... 4
3 The Role of Flash Memory in the Memory Hierarchy ... 5
3.1 Extension of Buffer Pool or Persistent Storage? ... 5
3.2 Replace or Supplement Hard Drives? ... 5
4 The Five-Minute Rule Before and After Flash Memory ... 6
5 Data Structures Reconsidered ... 7
5.1 Log-Structured Approach ... 7
5.2 B-trees and Flash Memory ... 8
5.3 R-Trees and Flash Memory ... 9
6 Examples of two DBMSs optimized for Flash ... 9
6.1 LGeDBMS ... 9
6.2 FlashDB ... 10
7 Conclusion ... 10
Bibliography ... 11
Appendix A – Main References ... 12

1 Introduction

Today's DBMSs use magnetic disk drives as their primary means of storage. This comes of several reasons: First and foremost, they offer persistent storage which is crucial to support critical data applications in case of system or power failure. Second, disk drives are acceptably quick – descent throughput rate and acceptable seek times, and the cost of storing one gigabyte is relatively low. Moving up the traditional memory hierarchy, we get to main memory, which is a huge jump upwards both in price and performance, in addition to the volatile nature of DRAM.

Today's DBMSs therefore focus on how to take advantage of both, by using main memory as a smaller buffer and disk as permanent storage of large data volumes as well as stable storage of transaction logs, in order to support ACID[2] properly.

It is in this gap in the memory hierarchy flash memory comes into the picture, falling in between traditional RAM and persistent mass storage based on rotating disks in terms of cost of ownership (including power), access latency and transfer bandwidth (RAIDed) [1]. In addition to that, flash memory is persistent storage. We can summarize the most important property, latency, with the following relation:

Latency: Disk = 100 x Flash memory = 100 000 x RAM

There are several ways to introduce flash memory into today's computer systems. It can be connected via a DIMM memory slot, via an S-ATA disk interface, or an entirely new interface. The trend today is the disk interface approach. Another major question is whether flash memory should be considered a special part of main memory, or as a special part of persistent storage, mimicking a hard disk drive. A third question is if hard disks will be completely replaced by flash memory or if they will still be an active part in the memory hierarchy. Section 3 aims to give insight into these questions.

Media	**Price**	**Access time**			**Transfer bandwidth**	**Power Cons.** Act/Idle/Slp
		Read	Write	Erase		
Magnetic Disk	$0.3/GB	12.7 ms (2 KB)	13.7 ms (2 KB)	N/A	85 MB/s	13W/8W/1W
NAND Flash	$30/GB	80 µs (2 KB)	200 µs (2 KB)	1.5 ms (128 KB)	25 MB/s	1W/0.1W/ 0.1W
DDR2-533 RAM	$25/GB	22.5 ns	22.5 ns	N/A	4266 MB/s	12W/6W/NA per GB

Table 1 – Characteristics of Magnetic disks vs. NAND Flash vs. RAM (ca.) [2], [3]

It is clear that such a new level in the memory hierarchy can be of great benefit to DBMSs. If we look at typical database queries, pending I/O[3] operations are often the main contributor to latency, either it is page read requests, or transaction log page flushes on transaction commits. If we can speed up the I/O by a factor of 100, this can clearly boost performance.

That said, it is not given that all types of queries will run equally fast or faster on flash memory, as we will see in the following sections. We will also possibly need to reconsider the data structures we are using, such as B-trees, to exploit the full opportunities of flash memory.

The rest of this paper is organized as follows. Section 2 describes the characteristics of flash memory. Section 3 discusses how flash memory fits into the memory hierarchy; section 4

---

[2] Atomicity, Consistency, Isolation, Durability
[3] Input/Output

illustrates flash memory impact with the five-minute rule, while section 5 presents some examples of what we need to do to exploit the full potential of flash memory. Finally, section 6 presents a few selected DBMSs written especially with flash memory in mind.

2 Characteristics of NAND Flash Memory

Before we can discuss what flash memory means for the DBMS world, we need to look at what makes flash memory different from traditional permanent storage – magnetic disk drives. There exists multiple types of flash memory, but this paper is about NAND flash memory, the type used in most flash memory disk devices today.

2.1 No In-Place Updates

While flash memory can write to clean (empty) sectors within a matter of microseconds, it cannot *overwrite* sectors. Before a sector can be overwritten, it has to be *erased*, which is a time-consuming process that takes around 1.5 ms. To make matters worse, erasing is done to whole *erase units*, which usually spans many sectors (usually 16 KB – 128 KB). Overwriting a single sector means reading out all the other sectors, erasing the unit and writing all of it back.

This can actually turn out to be even slower than magnetic hard drives, which can overwrite blocks with the usual latency. Considering that page overwrites is a common access pattern for modern DBMSs, this becomes an important issue, as we will see.

2.2 No Mechanical Latency

Unlike magnetic disk drives, flash memory is purely electronic and has no mechanically moving parts. For random accesses, seek and rotation time becomes the dominant factor for magnetic drives, while for flash memory, the latency is almost linearly proportional to the amount of data transferred, regardless of their physical location in memory.

This is the major selling point for flash memory – it can read or write (clean) sectors with close to constant time, making random reads/writes orders of magnitude faster than magnetic hard drives.

2.3 Limited Number of Writes

Flash memory has the property that it wears out and becomes statistically unreliable after a finite number of erase/write cycles, typically around 100 000. However, most flash memory devices or host systems adopt a process called *wear leveling* to evenly distribute erase cycles across the entire memory segment, prolonging its life to effectively eliminate the problem. [2], sec. 2.3.

2.4 Asymmetric Speed of Read/Write

While symmetric for most magnetic disk drives, the read speed of flash memory is typically at least twice as fast as the write speed. This is because it takes longer to inject a charge into a memory cell than reading its status. Most software systems today, including DBMSs, assume symmetric I/O speed, and are therefore not optimally tuned to be used with flash memory.

When using flash memory, we want to find ways to reduce writes, even if it increases reads a little.

3 The Role of Flash Memory in the Memory Hierarchy

3.1 Extension of Buffer Pool or Persistent Storage?

As mentioned in the introduction, there are several ways to introduce flash memory into the memory hierarchy. How to connect it to the system is not so interesting, but the question concerning if flash memory should be just an extended buffer pool or an extension of the persistent storage is a more important one.

[2] claims that database systems will benefit from the latter solution. (For reasons we not will look at here, the situation is a different one for file systems than for databases. See [2] for details). One of the reasons for that is due to **recovery** – ensuring durability of committed transactions at system or power failure. If flash memory is considered non-persistent storage, at least log writes would have to be flushed to disk at transaction commit, still leaving us with the disk as the slowest component involved in a transaction commit. If flash memory is considered persistent storage on the other hand, writing log records to flash memory is enough to satisfy the durability requirements and could possibly result in a speedup of transaction commit latency.

However, the low write bandwidth of flash memory speaks against using it as the sole log media. One possible solution is have dual logs, one on disk and one in flash memory, to combine low latency and high bandwidth.

To ensure fast recovery after a system or power failure, database systems employ **checkpoints**. During recovery, only activity since the last checkpoint plus some limited activity indicated in the checkpoint information is required to be examined. This is effectively achieved by writing dirty pages from the buffer pool to persistent storage during the checkpoint process, which usually runs at regular intervals, given in seconds or minutes.

If flash is considered non-persistent storage, dirty pages would need to be written to disk, with the increased latency this gives. If there is a lot of write activity against flash memory, this would end up in frequent checkpoints, meaning frequent writes to disk, failing the goal of having flash memory absorb write activity.

However, if flash memory is considered persistent, the checkpoint processing would only need to write dirty pages in main memory to flash memory, possibly speeding up the checkpoint process as well.

3.2 Replace or Supplement Hard Drives?

The third question mentioned was if flash memory will replace hard drives completely as an active participant in a running system (leaving them as backup devices, for instance), or if they will supplement hard drives as an additional level in the memory hierarchy, having hard drives store less frequently accessed data.

As of now (and probably a few years from now), flash drives are probably too expensive to replace hard drives, at least for bigger systems. For smaller systems it is achievable, but for the largest of today's OLTP[4] systems, towering at over 18 terabytes and over 50 thousand tps[5][4], it becomes very expensive. Still, as flash memory becomes cheaper, it is likely to replace hard drives in the long run [5].

[4] Online Transaction Processing
[5] transactions per second

Most of this paper looks at the complete replacement scenario, but using flash memory as a layer in between main memory and disk is also a very interesting prospect. It can be used as a level 2 buffer, so that pages falling out of main memory buffer can reside in flash memory buffer another while. Such a solution could be implemented using two separate LRU[6] queues in main memory. As another example, flash memory is also interesting when processing large queries which require more memory than available main memory. One such example is sorting of large data volumes, using external merge sort [6]. The traditional way is to merge runs on disk, but with flash, we could first merge runs in flash memory and then merge those runs on disk to achieve increased sort efficiency.

4 The Five-Minute Rule Before and After Flash Memory

When dimensioning a computer database system, there is always a question of how much RAM one should add. Frequently accessed pages should be kept in main memory, while less frequently accessed pages should reside on disk. In some situations, response time requirements make it necessary to keep certain pages in memory, but more often, keeping data in main memory is purely an economic issue. If we add too much main memory it becomes very expensive, but if we add too little, we will need a very expensive disk system to handle the increased disk load due to little buffer space.

The five-minute rule was presented by Gray and Putzolo in 1987, and describes the break-even point in access frequency when trading off acquisition cost for memory and I/O capacity. The question they asked themselves was “When is it cheaper to keep a record in main memory rather than access it on disc?” [7]. At that time, they concluded that for a page size of 1 KB, 400 seconds (~5 minutes) was the break-even interval. They also concluded that the break-even interval is inversely proportional to the page size, and gave 1 hour for 100 byte pages and 2 minutes for 4 KB. This means that at 1 KB page was, at that time, cheaper to keep in memory if was accessed more frequently than every 5 minutes.

$$\text{Break-even interval} = \frac{\text{Pages per MB of RAM}}{\text{Accesses Per Second Per Disk}} \times \frac{\text{Price per disk drive}}{\text{Price per MB of RAM}}$$

Equation 1 - Formula forming the basis of the five-minute rule

Ten years later, in 1997, the rule was revised and updated. Prices had dropped and performance gone up, but still: The break even-interval for 4 KB pages was still around 5 minutes (compared to 2 minutes in 1987) [8].

Twenty years later, in 2007, the rule was reviewed again, this time taking flash memory into account [2]. The results are shown in Table 2. SATA means a regular S-ATA disk drive; flash means flash memory with today’s prices, while $400 means a 32 GB NAND flash drive available for $400 in the future rather than $999 today.

Page size	1KB	4KB	16KB	64KB	256KB
RAM-SATA	350	87	22	6	1
RAM-flash	42	15	8	6	6
Flash-SATA	538	135	34	9	2
RAM-$400	17	6	3	2	2
$400-SATA	1343	336	84	21	6

Table 2 - Break-even intervals [minutes] [2].

[6] Last recently used

We can see that for 4 KB pages, the break-even interval between RAM and disk is now close to *1½ hours*, because of dramatic fall in RAM costs. If we look at RAM vs. flash, we get that break-even is *15 minutes*, because flash can deliver cheaper accesses than disk drives. If we assume $400, we get *6 minutes*. We can see that turn-over in RAM is about 15 times faster (1½ hours / 6 minutes) if flash is secondary storage rather than disk.

If we use the rule on flash vs. disk, we get (from row three) that break-even is 135 minutes or *2¼ hours*. This suggests that all data touched more frequently than 2¼ hours, which is effectively all active data, should stay in RAM and flash memory [2]. This leaves disks with storing archive data, such as transaction history and other data not frequently used.

5 Data Structures Reconsidered

Most DBMS data structures and algorithms are based on the characteristics of the underlying storage subsystems – their latency, transfer rate and block size. For example, one of the reasons to why all major DBMSs use B-trees (or B-link trees) and not binary search trees, is high read latency for disk drives. When we introduce flash memory, with radically different characteristics, we have to review the data structures and algorithms in use. As an example, this chapter looks closer at B and R-trees and how to perform out-of-place updates.

5.1 Log-Structured Approach

Considering that typical OLTP workloads often consists of a large portion random writes, the no in-place update property of flash memory is bad news. Table 3 shows typical numbers we can expect from running queries against a magnetic disk and NAND flash.

Media	Random-to-sequential Ratio	
	Read workload	Write workload
Magnetic Disk	4.3 ~ 12.3	4.5 ~ 10.0
NAND Flash	1.1 ~1.2	2.4 ~ 14.2

Table 3 - DBMS Performance: Sequential vs. random [3]

For disk, the ratio is fairly high both for read and write. Random access is significantly slower than sequential, but in practice equal for read and write. For flash, the ratio is much lower for read, indicating almost no higher latency for random reads. For write, however, the ratio is even more sensitive to access pattern than the disk – this is caused by the pattern triggering several flash erase operations.

To overcome this problem, we must ensure that we perform as few in-place updates as possible. One approach to this is to employ a log-structured system [9], where we log the changes made to the data instead of overwriting it. This way we avoid the costly erase operations, at the cost of an increased number of reads (which are fairly quick for flash memory anyway).

Traditional log structured systems usually have a sequential, common log for all pages. When using flash memory, there is no compelling reason to have a sequential log, as random writes (with no erase) are just as quick. Therefore, [3] suggests that pages are split in two – one section for data and one for log records. Log records describing changes to the data in the page are logged to the log section in the same page, effectively co-locating log and data, making it faster to reapply the log records to the data later to get the current version of a data segment. [3] also describes merge procedures to be invoked when pages run out of log space.

5.2 B-trees and Flash Memory

B-trees are the "work horses" of most of today's DBMSs, and it is therefore paramount that they work well on the storage subsystem in use. B-trees are balanced, rooted search trees, with nodes that have pointers to two or more children in the tree, possibly several thousands, sorted on key value. B^+-trees have data records stored in the leaf nodes.

There is primarily one big issue with regular B-trees and flash memory, with the no in-place update property of flash memory as the cause. Consider a regular B^+-tree stored in flash memory. Whenever a key value is updated in a data record in a leaf node, this means that the node will have to be updated. Since it cannot be updated in-place, it will have to be written to a new location. Since this changes the physical address of the node, the pointer in the parent node will have to be updated too. This causes the parent node to be written to a new location, and so on. This process continues all the way to the root, causing H writes, where H is the height of the tree. This method of updating a tree is called a "*wandering tree*" and is shown in Figure 1 [10]. It is clearly not very optimal.

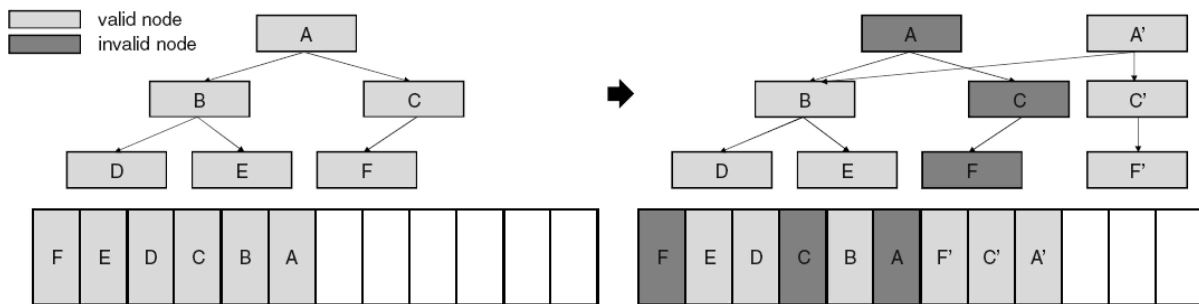

Figure 1 - A "wandering tree" update example [10]

There are in general at least two ways to solve this problem. One of them is to create a mapping layer between the B-tree implementation and the flash memory device, mapping logical page addresses to physical addresses. This layer helps avoiding in-place updates and speeds up the B-tree implementation. The advantage of this approach is that requires minimal changes in the application implementing the B-tree. See [11] for an example of such a solution.

Another solution is to revise the whole B-tree implementation, optimizing it for no in-place updates. One such implementation is the *μ-Tree* [10], which is a balanced search tree, similar to a B^+-tree. It is implemented so that as much as possible of the path from the root node to a leaf node is within a single page in order to minimize the number of flash write operations required when a leaf node is updated. [10] claims that the μ-Tree outperforms B^+-trees by up to 28% for real workload traces. Figure 2 shows an update of the page F in a μ-Tree.

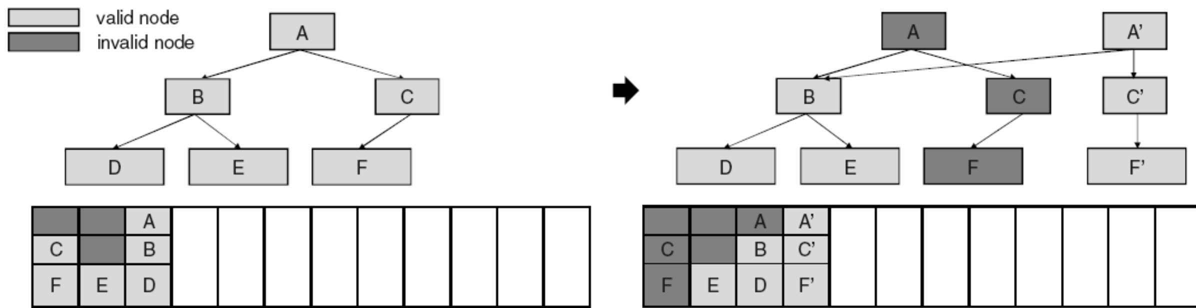

Figure 2 - A μ-Tree update example [10]

Another thing to consider for B-trees on flash memory is page size. The optimal page size combines a short access time with a high reduction in remaining search space (called the *utility*). For hard disks, the access time dominates for all small page sizes, such that additional byte transfer and thus additional utility is almost free [2]. Table 4 and Table 5 shows utility vs. time for B-tree nodes on disk and on flash memory.

Page Size	Records / page	Node utility	Access time	Utility / time
4 KB	140	7	10.0	0.58
16 KB	560	9	12.1	0.75
64 KB	2,240	11	12.2	0.90
128 KB	4,480	12	12.4	0.97
256 KB	8,960	13	12.9	1.01
512 KB	17,920	14	13.7	1.02
1 MB	35,840	15	15.4	0.97

Table 4 - Page utility for B-tree nodes on disk [2]

Page Size	Records / page	Node utility	Access time	Utility / time
1 KB	35	5	0.11	43.4
2 KB	70	6	0.13	46.1
4 KB	140	7	0.16	43.6
8 KB	280	8	0.22	36.2
16 KB	560	9	0.34	26.3
64 KB	2,240	11	1.07	10.3

Table 5 - Page utility for B-tree nodes on flash memory [2]

For disk, 256 KB is very near the optimal page size. For flash memory, the transfer time dominates even for small pages, so the optimal page size is just 2 KB [2].

5.3 R-Trees and Flash Memory

R-trees are similar to B-trees, but are used for spatial access methods, i.e. indexing multidimensional data, such as geographic locations [12]. They suffer from the same problems regarding in-place updates as B-trees. [13] describes an implementation which uses a reservation buffer, which is an in-memory write buffer, to queue up newly created objects before writing them to disk, minimizing the number of overwrites. They claim to have achieved significantly better performance than for regular R-trees.

6 Examples of two DBMSs optimized for Flash

Quite some research has been done in this area, and the goal of this section is to provide some insight into two research databases, built especially with flash memory in mind.

6.1 LGeDBMS

Flash memory is not only relevant in big database systems, but also smaller systems like mobile phones and embedded devices, which typically have flash memory as their only means of storage. *LGeDBMS* is a relational database system for such applications, “designed for efficient data management and easy data access in embedded mobile systems”[14].

LGeDBMS employs a log-structured approach as described in [9], doing out-of-place updates instead of in-place to avoid the costly erase operations. It also includes a PID

mapping table, mapping logical pages to physical pages. This makes it possible to exploit certain properties of the underlying file system, or control *wear leveling* if it is run directly against the flash memory at the device driver level. Basic, atomic transactions are supported. More information about LGeDBMS can be found in [14].

6.2 FlashDB

FlashDB is a self-tuning database from Microsoft Research, optimized for sensor networks using NAND flash storage. As they write, in practical systems, flash memory is used in different packages such as on-board flash chips, compact flash cards, secure digital cards and so on. MSR's experiments reveal non-trivial differences in their access costs. Furthermore, databases may be subject to different types of workloads, resulting in that existing databases are not optimized for all combinations of hardware and workloads [15].

The motivation behind FlashDB is to, in addition to performing the usual out-of-place updates; utilize self-tuning indexes that dynamically adapts their storage structure to workload and underlying storage device. They key point is that they use B^+-trees in two modes – *log mode* and *disk mode*. *Disk mode* is much like regular B^+-trees, while *log mode* employs a log-structured approach much like [9]. They each have their pros and cons. Even individual pages in the same tree can be in either mode. The storage manager chooses between the two modes, along with adjusting the page size for optimal performance. See [15] for more information about *FlashDB.*

7 Conclusion

It is clear that flash memory has come to stay, and that it will become cheaper and cheaper. Given a lower price in the future, its low access times and future larger memory sizes; it is likely to replace hard disk drives in the long run.

In the DBMS world, flash memory has some very interesting properties, namely low latency and persistent storage, which enables us to retrieve and update data faster and with lower latency than before. However, there are some issues concerning no in-place updates for flash memory that we will need to address first.

We have also seen that the five-minute rule still holds with 4 KB pages and a future $400 flash drive, but with a larger page size for disks – 64 KB. For 4 KB pages it also predicts that all active data will stay in RAM and flash memory.

Bibliography

[1] **Gray, Jim and Fitzgerald, Bob.** *FLASH Disk Opportunity for Server-Applications.* s.l. : Microsoft Research, 2007.

[2] *The five-minute rule twenty years later, and how flash memory changes the rules.* **Graefe, Goetz.** Palo Alto, CA : ACM, 2007. DaMoN'07.

[3] *Design of Flash-Based DBMS: An In-Page Logging Approach.* **Lee, Sang-Won and Moon, Bongki.** Suwon, Korea; Tucson, AZ : ACM, 2007. SIGMOD'07.

[4] **Winter Corporation.** Top Ten Program Results - All Winners. [Online] 2003. [Cited: April 21, 2008.] http://www.wintercorp.com/vldb/2003_TopTen_Survey/All%20Winners.pdf.

[5] **Slocombe, Mike.** Samsung CEO: NAND Flash Will Replace Hard Drives. [Online] September 2005. [Cited: April 21, 2008.] http://digital-lifestyles.info/2005/09/14/samsung-ceo-nand-flash-will-replace-hard-drives/.

[6] **Graefe, Goetz.** *Implementing Sorting in Database Systems.* s.l. : ACM Computer Surv. 38(3), 2006.

[7] **Gray, Jim and Putzolu, Franco.** *The 5 Minute Rule for Trading Memory for Disc Accesses and The 5 Byte Rule for Trading Memory for CPU Time.* Cupertino, CA : s.n., 1986.

[8] **Gray, Jim and Graefe, Goetz.** *The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb.* Redmond : Microsoft Research, 1997.

[9] *The Design and Implementation of a Log-Structured File System.* **Rosenblum, Mendel and Ousterhout, John K.** Berkeley, CA : ACM, 1991.

[10] *μ-Tree: An Ordered Index Structure for NAND Flash Memory.* **Kang, Dongwon, et al.** Dejeon, Korea : ACM, 2007. EMSOFT'07.

[11] **Wu, Chin-Hsien, Chang, Li-Pin and Kuo, Tei-Wei.** An Efficient B-Tree Layer for Flash-Memory Storage Systems. *Lecture Notes in Computer Science.* Heidelberg : Springer Berlin, 2004.

[12] R-tree. *Wikipedia.* [Online] [Cited: April 21, 2008.] http://en.wikipedia.org/w/index.php?title=R-tree&oldid=206564747.

[13] *An Efficient R-Tree Implementation over Flash-Memory Storage Systems.* **Wu, Chin-Hsien, Chang, Li-Pin and Kuo, Tei-Wei.** Taipei, Taiwan : ACM, 2003. GIS'03.

[14] *LGeDBMS: a Small DBMS for Embedded System with.* **Kim, Gye-Jeong, et al.** Seoul, Korea : ACM, 2006. VLDB'06.

[15] *FlashDB: Dynamic Self-tuning Database for NAND Flash.* **Nath, Suman and Kansal, Aman.** s.l. : ACM, 2007. IPSN'07.

Appendix A – Main References

This appendix lists the three papers I have chosen as the main references.

[2] *The five-minute rule twenty years later, and how flash memory changes the rules.* **Graefe, Goetz.** Palo Alto, CA : ACM, 2007. DaMoN'07.

[3] *Design of Flash-Based DBMS: An In-Page Logging Approach.* **Lee, Sang-Won and Moon, Bongki.** Suwon, Korea; Tucson, AZ : ACM, 2007. SIGMOD'07.

[10] *μ-Tree: An Ordered Index Structure for NAND Flash Memory.* **Kang, Dongwon, et al.** Dejeon, Korea : ACM, 2007. EMSOFT'07.