

# Flash-aware Database Management Systems

Hardock, Sergej  
(2020)

DOI (TUprints): <https://doi.org/10.25534/tuprints-00014476>

License:



CC-BY-SA 4.0 International - Creative Commons, Attribution Share-alike

Publication type: Ph.D. Thesis

Division: 20 Department of Computer Science

Original source: <https://tuprints.ulb.tu-darmstadt.de/14476>

---

---

# Flash-aware Database Management Systems

---

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)  
Genehmigte Dissertation von Sergej Hardock aus Kiew, Ukraine  
Tag der Einreichung: 30.7.2020, Tag der Prüfung: 4.11.2020

1. Gutachten: Prof. Dr. Carsten Binnig
2. Gutachten: Prof. Dr.-Ing. Ilia Petrov
3. Gutachten: Alejandro Buchmann, Ph.D.  
Darmstadt



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Computer Science  
Department  
Technical University of  
Darmstadt  
Data Management Lab

Flash-aware Database Management Systems

Accepted doctoral thesis by Sergej Hardock

1. Review: Prof. Dr. Carsten Binnig
2. Review: Prof. Dr.-Ing. Ilia Petrov
3. Review: Alejandro Buchmann, Ph.D.

Date of submission: 30.7.2020

Date of thesis defense: 4.11.2020

Darmstadt

Bitte zitieren Sie dieses Dokument als:

URN: urn:nbn:de:tuda-tuprints-144769

URL: <http://tuprints.ulb.tu-darmstadt.de/id/eprint/14476>

Dieses Dokument wird bereitgestellt von tuprints,

E-Publishing-Service der TU Darmstadt

<http://tuprints.ulb.tu-darmstadt.de>

[tuprints@ulb.tu-darmstadt.de](mailto:tuprints@ulb.tu-darmstadt.de)

Die Veröffentlichung steht unter folgender Creative Commons Lizenz:

Namensnennung - Weitergabe unter gleichen Bedingungen 4.0 International

<https://creativecommons.org/licenses/by-sa/4.0>

This dissertation is dedicated to my parents who encouraged me to follow my dreams.



---

## Erklärungen laut Promotionsordnung

### §8 Abs. 1 lit. c PromO

Ich versichere hiermit, dass die elektronische Version meiner Dissertation mit der schriftlichen Version übereinstimmt.

### §8 Abs. 1 lit. d PromO

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuchs mitzuteilen.

### §9 Abs. 1 PromO

Ich versichere hiermit, dass die vorliegende Dissertation selbstständig und nur unter Verwendung der angegebenen Quellen verfasst wurde.

### §9 Abs. 2 PromO

Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 30.7.2020

---

S. Hardock



---

# Acknowledgments

---

First, I would like to thank my supervisors Ilia Petrov and Alejandro Buchmann. My interest in database management systems, which continuously evolved over the years, began back in year 2010 with your exciting lectures. After different projects and Master thesis done under your supervision, it was a great pleasure to get an offer to work as PhD student in your team. Without your support, help and guidance I would not be able to finish this dissertation.

I wish to thank all other members of defense committee Carsten Binnig, Andreas Koch, Christian Bischof and Max Mühlhäuser for meaningful and fruitful discussions, comments and suggestions.

Many thanks to my former colleagues Robert Gottstein, Daniel Bausch, Dejan Petkov, Alexander Frömmgen and Robert Rehner. Your support has helped me a lot during the research. I really appreciate the work and spare time we spent together at university and on conferences. I like to express my gratitude to Maria Tiedemann for the help in lots of organisational topics.

I had great pleasure of working with undergraduate students Sruthi Parvathy Subramanian, Mohan Kanth Dayanandan and Elena-Madalina Cososchi. Thank you for your contribution to this project.

Special thanks to my family, especially to my dear parents. Your support and belief in me helped me in various difficult moments over these 6 years. You did not let me give up at the final, most difficult steps.



---

# Abstract

---

Flash SSDs are becoming the primary storage technology for single servers and large data centers. In contrast to conventional magnetic disks, which were dominating the storage market for more than 40 years, Flash offers significantly more performance, consumes less energy and has lower cost per IOPS (I/O Operations Per Second). Besides these advantages, an important role in establishment and quick proliferation of Flash storage was played by the black-box design of SSDs, which guaranteed their backwards compatibility with the traditional hard disk drives. This makes the replacement of HDDs seamless as the software stack, including the application, does not require any adjustment. However, such design of SSDs has multiple disadvantages, which become especially critical for database management systems.

The backwards compatibility of SSDs is encapsulated in the so-called Flash translation layer (FTL). FTL is a set of Flash management tasks that typically run on device and mask the native behavior of Flash memory. In other words, FTL creates a black-box over Flash memory and emulates the behavior of HDDs. The fact that the database system has no knowledge about FTL, and has no control over the physical data placement on Flash, results in high I/O overhead, which is caused by suboptimal realization of Flash management tasks and functional redundancy along the critical I/O path. Thus, write-amplification of conventional SSDs used in traditional 'cooked' storage architecture (i.e., with file system indirection) can be as high as 15x, i.e., a single 4KB write request submitted by the DBMS can turn into 60KB being physically written on Flash storage. As a result, the effective I/O throughput and longevity expectations of SSDs are significantly lower than those of Flash memory encapsulated in these SSDs.

In this work we describe our approach - the NoFTL storage architecture - that aims to solve the aforementioned disadvantages of modern Flash SSDs. The basic idea behind the NoFTL is to give the full control over the underlying Flash storage to the database management system, which in turn assumes elimination of all intermediate abstraction layers (file system, block device layer and FTL) between the DBMS and physical storage. NoFTL consists of three main elements - (i) native Flash interface; (ii) integration of Flash management into subsystems of the DBMS; and (iii) the concept of configurable Flash storage. The interplay of them allows us to realize the whole performance potential of

---

Flash memory. Native Flash interface allows the DBMS to control physical data placement on Flash storage, and to utilize the computational power of the SSD to perform near-data processing. Integration of typical Flash management tasks (address translation, garbage collection and wear leveling) into different subsystems of the DBMS leads to an optimization of these tasks and of native DBMS algorithms. The concept of configurable Flash storage is a unique approach to organize and manage data on Flash SSDs. With the help of novel storage abstractions, the database system can perform intelligent data placement by clustering objects into different regions. Moreover, for each such region the DBMS can apply a separate set of Flash management algorithms, which would be optimal for data assigned to that region.

All this reduces the write-amplification of SSDs to a minimum (up to 15x reduction for OLTP workloads), improves the overall system performance, and significantly increases the lifetime of Flash SSDs (up to 30x improvement). We have realized the NoFTL prototype on an open-source database engine and evaluated it under various scenarios and on different testbeds.

---

# Zusammenfassung

---

Flash-SSDs werden zur primären Speichertechnologie für einzelne Server und große Rechenzentren. Im Gegensatz zu herkömmlichen Festplatten, die mehr als 40 Jahre lang den Speichermarkt dominierten, bietet Flash deutlich mehr Leistung, verbraucht weniger Energie und hat geringere Kosten pro IOPS (I/O-Operationen pro Sekunde). Eine wichtige Rolle bei der Etablierung und schnellen Verbreitung von Flash-Speichern spielte neben diesen Vorteilen auch das Black-Box-Design von SSDs, wodurch deren Abwärtskompatibilität mit den herkömmlichen Festplatten garantiert wurde. Dies macht den Austausch von Festplatten nahtlos, da der Software-Stack einschließlich der Anwendung keine Anpassung erfordert. Ein solches Design von SSDs weist jedoch mehrere Nachteile auf, die für die Datenbankverwaltungssysteme besonders kritisch werden.

Die Abwärtskompatibilität von SSDs ist in der sogenannten Flash Translation Layer (FTL) eingekapselt. FTL ist eine Reihe von Flash-Verwaltungsaufgaben, die normalerweise auf dem Gerät ausgeführt werden und das native Verhalten des Flash-Speichers maskieren. Mit anderen Worten, FTL erstellt eine Blackbox über dem Flash-Speicher und emuliert das Verhalten von Festplatten. Die Tatsache, dass das Datenbanksystem keine Kenntnisse über FTL und auch keine Kontrolle über die physische Datenplatzierung auf dem Flash hat, führt zu einem hohen I/O-Overhead, der durch die suboptimale Realisierung von Flash-Verwaltungsaufgaben und funktionale Redundanz entlang des kritischen I/O-Pfades verursacht wird. Somit kann der Schreibfaktor (write amplification) herkömmlicher SSDs, die in der üblichen "cookedSSpeicherarchitektur verwendet werden (d.h. mit Dateisystem-Indirektion), bis zu 15x sein. So kann eine einzelne 4KB Schreibanforderung, die vom DBMS gesendet wird, bis zu 60KB werden, die physisch auf dem Flash-Speicher geschrieben sind. Infolgedessen sind der effektive I/O-Durchsatz und die Langlebigkeitserwartungen von SSDs erheblich niedriger als die von in diesen SSDs eingekapselten Flash-Speichern.

In dieser Arbeit beschreiben wir unseren Ansatz - die NoFTL-Speicherarchitektur, der darauf abzielt, die oben genannten Nachteile moderner Flash-SSDs zu lösen. Die Grundidee hinter der NoFTL besteht darin, dem Datenbankverwaltungssystem die vollständige Kontrolle über den zugrunde liegenden Flash-Speicher zu geben, was wiederum die Eliminierung aller zwischengeschalteten Abstraktionsschichten (Dateisystem, Block-

---

Geräteschnittstelle und FTL) zwischen dem DBMS und dem physischen Speicher voraussetzt. NoFTL besteht aus drei Hauptelementen: (i) native Flash-Schnittstelle; (ii) Integration des Flash-Managements in Subsysteme des DBMS; und (iii) das Konzept des konfigurierbaren Flash-Speichers. Das Zusammenspiel ermöglicht es uns, das gesamte Leistungspotential des Flash-Speichers auszuschöpfen. Mit der nativen Flash-Schnittstelle kann das DBMS die Platzierung physischer Daten auf dem Flash-Speicher steuern und die Rechenleistung der SSD für die datennahe Verarbeitung nutzen. Die Integration typischer Flash-Verwaltungsaufgaben (Adressumsetzung, Garbage Collection und Wear-Leveling) in verschiedene Subsysteme des DBMS führt zu einer Optimierung dieser Aufgaben und als auch Optimierung nativer DBMS-Algorithmen. Das Konzept des konfigurierbaren Flash-Speichers ist ein einzigartiger Ansatz zum Organisieren und Verwalten von Daten auf Flash-SSDs. Mithilfe neuartiger Speicherabstraktionen kann das Datenbanksystem eine intelligente Datenplatzierung durchführen, indem Objekte in verschiedenen Regionen gruppiert werden. Darüber hinaus kann das DBMS für jede dieser Regionen einen separaten Satz von Flash-Verwaltungsalgorithmen anwenden, die für Daten, die dieser Region zugewiesen sind, optimal wären.

All dies reduziert den Schreibfaktor von SSDs auf ein Minimum (bis zu 15-fache Reduzierung für OLTP-Workloads), verbessert die Gesamtsystemleistung und verlängert die Lebensdauer von Flash-SSDs erheblich (bis zu 30-fache Verbesserung). Wir haben den NoFTL-Prototyp auf einer Open-Source-Datenbank-Engine realisiert und unter verschiedenen Szenarien und auf verschiedenen Testumgebungen evaluiert.

---

# Contents

---

<b>Glossary</b>	<b>xvii</b>
<b>List of Figures</b>	<b>xix</b>
<b>List of Tables</b>	<b>xxii</b>
<b>1. Introduction</b>	<b>1</b>
1.1. Scope . . . . .	1
1.2. Problem Statement . . . . .	2
1.3. Proposed Approach . . . . .	8
1.4. Contributions . . . . .	10
1.5. Structure of the Thesis . . . . .	12
<b>2. Background</b>	<b>13</b>
2.1. DBMS . . . . .	13
2.1.1. Evolution . . . . .	13
2.1.2. Architecture of Relational DBMS . . . . .	15
2.2. DBMS Storage Alternatives and I/O Stack . . . . .	19
2.2.1. Cooked Storage . . . . .	20
2.2.2. Raw Storage . . . . .	24
2.2.3. Block-Device Layer . . . . .	26
2.3. Flash . . . . .	27
2.3.1. NAND Flash Memory . . . . .	28
2.3.2. Flash SSD . . . . .	37
<b>3. NoFTL Approach</b>	<b>55</b>
<b>4. Native Flash Interface</b>	<b>71</b>
4.1. Command Set . . . . .	71
4.2. User-Defined Commands . . . . .	72
4.3. Addressing and Granularity . . . . .	73

---

---

<b>5. DBMS Integration</b>	<b>75</b>
5.1. Address Translation . . . . .	76
5.2. Atomicity of Writes . . . . .	77
5.3. Garbage Collection and Wear-Leveling . . . . .	78
5.4. Parallelism . . . . .	79
5.5. Recovery . . . . .	80
5.6. Evaluation Results . . . . .	81
<b>6. Configurable Flash storage</b>	<b>87</b>
6.1. NoFTL Regions . . . . .	90
6.1.1. Data Placement with Regions . . . . .	90
6.1.2. Region-Specific Flash Management . . . . .	93
6.2. NoFTL Groups . . . . .	98
6.3. Evaluation Results . . . . .	101
<b>7. In-Place Appends</b>	<b>111</b>
7.1. Revisiting Erase-Before-Overwrite Principle . . . . .	113
7.2. Design and Implementation Details of IPA . . . . .	116
7.2.1. Page Layout for IPA . . . . .	117
7.2.2. Page Operations under IPA . . . . .	119
7.2.3. WRITE_DELTA Command . . . . .	121
7.2.4. Error Detection and Correction . . . . .	123
7.3. Variations of IPA . . . . .	124
7.3.1. IPA for Indexes . . . . .	124
7.3.2. IPA on Different Flash Types . . . . .	127
7.4. Evaluation . . . . .	134
7.4.1. Basic IPA . . . . .	135
7.4.2. IPA for Indexes . . . . .	142
<b>8. Auto-Tuning</b>	<b>147</b>
8.1. Selection of Multi-Region Configuration . . . . .	147
8.2. Change of Multi-Region Configuration . . . . .	148
8.3. Wear-Leveling in Multi-Region Configuration . . . . .	156
<b>9. Related Work</b>	<b>159</b>
9.1. FTL-based SSDs . . . . .	159
9.2. Native Flash Storage . . . . .	163
9.3. IPA Approach . . . . .	166

---

---

9.4. In-Storage Processing . . . . .	167
<b>10. Conclusions</b>	<b>169</b>
<b>11. Future Work</b>	<b>173</b>
<b>A. Appendix</b>	<b>187</b>
A.1. OpenSSD Jasmine . . . . .	187



---

# Glossary

---

**BBM** bad-block management

**BCH** Bose–Chaudhuri–Hocquenghem codes

**BLM** block-level address mapping

**DBMS** database management system

**DDL** data definition language

**ECC** error-correction codes

**FS** file system

**FTL** Flash translation layer

**GC** garbage collection

**HDD** hard disk drive

**I/O** input/output operation

**ISP** in-storage processing

**ISPP** incremental step-pulse programming

**LDPC** low-density parity-check code

**MLC** multi-level cell

**NDP** near-data processing

---

**NFI** native Flash interface

**NVM** non-volatile memory

**OS** operating system

**PLM** page-level address mapping

**QLC** quad-level cell

**SLC** single-level cell

**SQL** structured query language

**SSD** solid-state drive

**TLC** triple-level cell

**WL** wear-leveling

---

## List of Figures

---

1.1. Pillars of NoFTL approach. . . . .	9
1.2. Performance comparison of DBMS on different storage alternatives under TPC-C benchmark. . . . .	11
2.1. Module-based architecture of the relational DBMS (based on [80]). . . . .	15
2.2. DBMS storage alternatives. . . . .	20
2.3. Floating-gate transistor making a single Flash memory cell. . . . .	29
2.4. Cell-array architecture of the NAND Flash memory. . . . .	30
2.5. Latencies of basic operations for different NAND types. Source: AnandTech [98] . . . . .	32
2.6. Four different types of NAND Flash memory. . . . .	35
2.7. Simplified architecture of SSD. . . . .	38
2.8. Main tasks of FTL. . . . .	41
2.9. I/O bandwidth fluctuations of an enterprise SSD. Source: Petrov et al. [77].	44
2.10. Example of data placement in page-level FTL. . . . .	45
2.11. Example of read operation in block-level FTL. . . . .	47
2.12. Example of write operation in block-level FTL. . . . .	48
2.13. Address translation in hybrid FTL. . . . .	49
2.14. Example of merge operation in hybrid FTL. . . . .	50
3.1. NoFTL as a storage alternative for DBMS. Source: Hardock et al. [34]. . . . .	56
3.2. Integration of Flash management into the modules of the DBMS. . . . .	59
3.3. Example of a region configuration. . . . .	62
3.4. Cumulative distribution of update-sizes in TPC-B benchmark under default eager-eviction strategy with different sizes of DBMS buffer. Source: Hardock et al. [31] . . . . .	66
3.5. Performance comparison of DBMS on different storage alternatives under TPC-C benchmark. Figure is also presented in Chapter 1. . . . .	68
3.6. Comparison of write amplification and intensity of erase operations for different storage alternatives under TPC-C benchmark. . . . .	69

---

5.1. NoFTL architecture and integration of Flash management in the DBMS. Source: Hardock et al. [34]. . . . .	76
5.2. NoFTL with traditional and Flash-aware strategies for background writes under TPC-C benchmark. . . . .	85
5.3. NoFTL with traditional and Flash-aware strategies for background writes under TPC-B benchmark. . . . .	85
6.1. Example of reclaiming a victim block by garbage collection. . . . .	88
6.2. Hot/cold data separation using NoFTL regions. . . . .	91
6.3. Control of SSD parallelism using NoFTL regions. . . . .	92
6.4. Selective Flash management with NoFTL Regions . . . . .	98
6.5. Hot/cold data separation using NoFTL groups . . . . .	99
6.6. Comparison of write amplification and intensity of erase operations for different storage alternatives under TPC-B benchmark. . . . .	109
7.1. Write-amplification under OLTP workloads. Source: Hardock et al. [31]. . . . .	112
7.2. Database page layout for IPA on Flash. Source: Hardock et al. [31] . . . . .	117
7.3. Example of database page with two delta records. Source: Hardock et al. [31] . . . . .	119
7.4. Implementation of write_delta command on OpenSSD Jasmine board. (1) up-to-date version of a page in database buffer pool; (2) "remembered" changes since last fetch from SSD encoded as delta record in delta-record area; (3) storage manager issues write_delta command only for delta record, i.e., only delta record is transmitted to the SSD; (4) controller copies delta record into the original position in an empty page (initialized with "1"s) in DRAM buffer of SSD (write cache); (5) original Flash page on Flash with empty delta-record area; (6) through overwriting the original (PPN=7) page only the delta-record area becomes updated, while the body of the page remains unchanged. . . . .	122
7.5. Modification of an entry in B-Tree. Traditional approach. . . . .	125
7.6. Page layout and format of delta record in IPA-IDX. Source: Hardock et al. [33] . . . . .	127
7.7. Modification of an entry in B-Tree using IPA-IDX. . . . .	128
7.8. Memory organization of MLC Flash. . . . .	130
7.9. Program interference in MLC Flash during PROGRAM operation. Cell $M$ is being programmed; cells $G, H, I, Q, R, S$ can experience program interference errors. . . . .	132

---

---

7.10. Cumulative distribution of update sizes in TPC-C under eager eviction strategy. . . . .	136
7.11. Cumulative distribution of update sizes in TPC-C under non-eager buffer eviction strategy. . . . .	141
7.12. CDF of update sizes for index on NewOrder table in TPC-C benchmark. . .	143
8.1. Excerpt from an example of <i>summary</i> output of NoFTL advisor. . . . .	149
8.2. Example of a migration to target NoFTL configuration. . . . .	151
A.1. OpenSSD Jasmine board. . . . .	187



---

## List of Tables

---

2.1. Comparison of NAND and NOR Flash memory types. . . . .	28
2.2. Variety of SSDs on the storage market (Part 1). . . . .	39
2.3. Variety of SSDs on the storage market (Part 2). . . . .	40
5.1. TPC-C benchmark under different storage alternatives on OpenSSD Jasmine board. . . . .	82
5.2. TPC-B benchmark under different storage alternatives on OpenSSD Jasmine board. . . . .	83
5.3. DFTL-based SSD versus NoFTL under TPC-C/B benchmarks. . . . .	84
6.1. TPC-C benchmark under three different NoFTL configurations on OpenSSD Jasmine board. . . . .	102
6.2. TPC-B benchmark under three different NoFTL configurations on OpenSSD Jasmine board. . . . .	103
6.3. Comparison of NoFTL with and without regions under TPC-C on Flash emulator. . . . .	104
6.4. Data placement configuration for TPC-C. . . . .	105
6.5. Data placement configuration with selective, region-specific Flash management for TPC-C benchmark. . . . .	106
6.6. Comparison of NoFTL approach with a single- and multi-FMS under TPC-C benchmark. . . . .	107
7.1. TPC-C benchmark on OpenSSD: traditional approach vs. IPA in pSLC and odd-MLC modes. Source: Hardock et al. [31] . . . . .	138
7.2. TPC-C: traditional (no IPA [0×0]) vs. [2×48] schemes with large buffer pools (eager eviction). Source: Hardock et al. [31] . . . . .	139
7.3. TPC-C: traditional (no IPA) vs. [2×M] schemes with large buffer pools (non-eager eviction). Source: Hardock et al. [31] . . . . .	142
7.4. Evaluation results of IPA-IDX on TPC-C benchmark. Source: Hardock et al. [33] . . . . .	145

---

7.5. Evaluation results of IPA-IDX on TPC-B benchmark. Source: Hardock et al. [33] . . . . .	146
---	-----

---

# 1. Introduction

---

## 1.1. Scope

Data volume and speed and quality of data processing have become crucial for many domains, such as, communication, security, entertainment, sales, mobility and health care. For more than forty years the hard disk drive was the only feasible solution for the secondary storage of large volumes of data with reasonable access speed. However, portable devices, such as, MP3 players, game consoles, digital cameras, smartphones, and a huge number of small embedded devices required a compact, shock-resistant and energy efficient memory. The solution was Flash memory. It took more than a quarter of a century for Flash memory to evolve from a small capacity storage for portable devices to the mass storage media for data intensive applications and data centers. Questions about moving from HDDs to Flash-based storage, durability, cost, and where Flash memory should be used in the storage architecture have shifted to a debate about what kind of Flash storage is best and how its properties can best be exploited by today's data processing software.

The major reason behind the limited use of Flash in the early 2010s was its price. The price per GB of Flash storage was in average 10-20x higher than the price per GB of HDD. Today this difference is about 5x, which is, however, still a lot. Nevertheless, Flash has won the competition on the storage market by another, more important metric – price/performance (\$/GB/s). In other words, it is not simply about the amount of stored data, but rather about the ability to process this data fast. Besides its better performance, Flash storage consumes significantly less energy (~3x), has typically much smaller form factor per GB, produces less heat and noise, and due to the absence of moving parts is much more robust and shock-resistant than HDDs. This makes it a perfect match for both large data centers, and small portable or embedded devices. Thus, Flash nowadays is the primary technology for non-volatile storage for all kinds of data management systems. The newest developments in Flash technology, as well as Non-Volatile-Memory, lead to the conclusion that in the next five years the market share of Flash will continuously increase from its current level of about 50%, and the dominance of Flash as a mass storage will remain for at least 10 more years.

However, despite all its advantages, the design of modern Flash-based storage faces

---

multiple issues that have kept Flash memory from realizing its full potential in data intensive applications. In this work, we focus on those issues, and propose a novel design along with multiple optimizations based on it, which significantly increase performance and longevity of Flash storage, simultaneously decreasing its overall cost.

## 1.2. Problem Statement

Although both software and hardware evolve continuously, they rarely evolve at the same rate. Software was usually leading, and thus was often hungering for more powerful hardware. However, during the last decade there has been a clear trend towards a role reversal. The power of computational units with high degree of parallelism and the performance of storage in modern systems are typically orders of magnitude higher in their hardware specifications than their real performance in running systems. The reason for this lies typically in legacy software, which is unable to utilize the full potential of hardware.

The common characteristic of most database systems until about the mid-2000s was the limited amount of main memory, which was making them IO-bound<sup>1</sup>. This created the situation where the gap between CPU and HDD became the major performance bottleneck for DBMSs. Not surprisingly, the design of those systems was strongly influenced by the IO infrastructure. In all DBMS subsystems, data structures and processing algorithms it is easy to find design decisions, which are based on the assumption of having HDD as the secondary storage, and are ment to reduce the IO costs. The buffer manager, query optimizer, storage and free space managers, recovery, access paths, page layout and the whole IO stack were strongly influenced by the characteristics of spinning storage media. These “deep roots” of the HDDs inside the DBMS, as well their clear monopoly on the storage market made the deployment of Flash storage for the database systems challenging.

Flash memory differs from magnetic disk significantly. Different physical principles of work and properties of media result also in the behavioral differences of Flash as compared to HDDs. The major of those are the following:

- Data access on Flash memory is **much faster** than on the HDD. Thus, reading a 4KB data block from Flash memory requires typically less than 50 $\mu$ s, while the high-end enterprise HDD would need at least 2ms, resulting therefore in more than 40x access time difference for reading.

---

<sup>1</sup>An IO-bound system is one in which the total time spent for accessing the data is significantly larger than the one for processing it.

- 
- Flash memory is **asymmetric**. Reading a page is usually 10x faster than writing a page, while erasing a block takes usually as much as 10 page writes. The latencies of read and write operations on the HDD are equal.
  - There are **no seek times** on Flash, i.e. the access time to an arbitrary location on Flash does not depend on the location itself and is constant in time<sup>2</sup>. On the HDD, in contrast, positioning of the moving head for a particular data access causes location dependent seek time, making performance of the drive strongly dependant on the workload's access pattern (random vs. sequential). Thus, the access “sequentialization” became the major instrument for DBMS designers to increase the I/O performance on magnetic storage.
  - Flash memory follows a so-called **erase-before-overwrite** principle. Once the data is written to a certain location on Flash, it cannot be updated in place without a preceding erase operation. Furthermore, while the minimal unit of reading and writing on Flash is a Flash page (typically 4-16KB), the erase operation is performed on Flash blocks, which are composed of hundreds of pages. This makes a simple update operation on Flash even more challenging. This is quite different from the overwrite behavior of the HDD, where updated data is simple written “on top” of the old one, and the erase operation as such is not defined at all.
  - Flash blocks **wear** with increasing number of erase cycles performed on them. After a certain number of erases, which might vary from 3 to 100 thousands depending on the Flash type, the block with all its pages cannot store data reliably anymore, and thus they become invalid for further read/write operations. The magnetic surface of the HDD plates is not affected equally by its use<sup>3</sup>.
  - Flash storage typically provides **high level of IO parallelism**, supporting dozens to hundreds different IO operations in parallel. The “built-in” parallelism of Flash is made possible by the multi-level memory hierarchy (see Chapter 2.3.2). The parallelism of the HDD is very limited. Multiple heads and plates of HDDs are still connected to the common axes, allowing therefore parallelism only through splitting large requests into chunks, which are then processed in parallel.

These physical properties of Flash memory influence the access strategy, and require managing functionality for the efficient and reliable usage of Flash storage. The combina-

---

<sup>2</sup>Although the sequential access to the Flash storage is still faster than the random, it is caused by optimized caching and other Flash management algorithms described in more detail in Chapter 2.3.2.

<sup>3</sup>However, the moving parts of spinning disks make this media more susceptible to failures as compared to Flash memory with its wear-outing nature.

---

tion of these three factors: (1) behavioral specifics of Flash memory, which would require significant architectural changes in the HDD-defined design of the DBMS and IO-stack; (2) the “immature” technological level of Flash development in the earlier 2000s; as well as (3) high price difference between Flash and HDD back then - created together a firewall for Flash on the way to become a mass storage for data intensive applications.

To overcome these obstacles and start the competition with HDDs, Flash storage was made to be backwards compatible to spinning drives. Flash Solid-State-Drives (hereafter SSDs) provided the same physical interface and transmission protocols as HDDs, and thus did not require any additional driver, and were recognized by the operating system and application as traditional HDDs. This made the replacement of the HDDs with SSDs seamless, and stimulated establishment and proliferation of Flash on the storage market.

To achieve this compatibility with the traditional hard disk drives, SSDs were designed as black boxes over Flash memory inside them. Through the use of an additional level of indirection, which is encapsulated in the storage device, all native specifics of Flash memory become hidden from the upper I/O layers. This indirection is realized as a software called Flash Translation Layer (FTL) running on the SSD’s internal controller. FTL executes all management functionality for Flash memory transparently to the host system, and thus it provides outward the illusion of operating on typical (although 2-5x faster) HDD, i.e., no erases, no out-of-place updates, no wear-out of Flash, no highly parallel internal architecture (see more about FTL functionality in Chapter 2.3.2).

A black-box design of SSDs is a standard nowadays. While the current storage market provides SSDs in a variety of form factors, with different physical interfaces and transmission protocols, with different “place of residence” of the FTL (device vs. OS), the common feature of probably 99.9%<sup>4</sup> of all Flash storage is still the FTL. Looking back, we can certainly say that this backward compatible design of SSDs did its job well – **Flash SSD is the storage of today, and this storage is FTL-based.**

However, for all its advantages, the backwards compatibility in general and FTL in particular are something of a double-edged sword. Masking the native behavior of Flash behind the additional abstraction layer, and emulating thereby legacy storage, results in two main disadvantages: overhead and underutilization. Those are themselves manifold. The overhead is of computational and IO nature, and it is rooted in many layers along the IO path (DBMS, file system, OS and FTL). The underutilization relates primary to memory and computational resources of Flash SSDs. The most important of those are briefly described below.

---

<sup>4</sup>We speculate on this number, since due to the clear dominance of the FTL-based SSDs on the market, the statistics regarding the popularity of alternative SSD designs is (to the best of our knowledge) not available. The very few FTL-less alternatives are mostly research projects and are covered later in Chapter 9.

- 
- Typically, SSDs are featured with quite **limited amount of volatile DRAM memory**<sup>5</sup>, varying from a few hundred megabytes (e.g., 64-256MB) in consumer SSDs to a maximum of 2GB in the expensive enterprise devices. The lion share of this memory is used by the controller for queuing and caching of incoming write requests (e.g., write-cache), which allows to smooth the negative effects of GC during the relatively low IO loads. The minor part of the on-device DRAM is used for caching FTL metadata. Among others this includes the mapping information required for the logical-to-physical address translation. However, since the remaining DRAM size does not allow to cache the complete mapping table with the desired fine granularity, FTL designers are forced to implement the alternative mapping schemes (e.g., hybrid mapping). As a result, the write-amplification and the insensitivity of erases produced by GC increases manifold (2-3x), which negatively impacts the foreground performance of the SSD and its longevity (see more in Chapters 2.3.2 and 5.6). To overcome those disadvantages, several SSD manufactures offer enterprise SSDs in which FTL is moved from device to host system (implemented as a device driver in the OS). This solves the problem of limited on-device DRAM resources, since now FTL uses main memory of the host.
  - Another typical source of the overhead is the **functional redundancy along the IO path**. In the traditional “cooked” storage alternative for the DBMS (see Figure 2.2), functions like buffering/caching, address mapping, free space management and recovery are performed separately by DBMS, file system and FTL. This results in the computational overhead and significant write-amplification on the SSD. For instance, we have experienced more than 3x write-amplification from the ext4 file system for the OLTP workloads (see Tables 5.1 and 5.2 in Chapter 5.6). Similar and even more dramatic results (up to 11x write-amplification) are presented in [59]. To eliminate the overhead from the file system some popular DBMSs (e.g., Oracle, IBM DB2) offer the so-called “raw” storage alternative. It eliminates the file system as an intermediate layer and gives the DBMS direct control over the storage. However, if the storage is based on Flash SSD, the terms “raw storage” and “direct control” are not valid anymore because of the indirection introduced by FTL.

While the above overheads could be eliminated by using SSDs with the OS-resident FTL and eliminating the file system, the major disadvantage of the FTL is present in all modern Flash SSDs regardless of their design or the configuration of the IO stack. By creating a

---

<sup>5</sup>In modern SSDs the on-device DRAM memory is backed by the large capacitors (big CAPs), which supply the device with enough power to write the data from volatile DRAM to durable Flash memory in case of power outage.

---

black-box abstraction over the Flash memory FTL builds a wall between the DBMS and SSD. This wall is opaque from both sides. On one side, the DBMS has almost no knowledge about the underlying storage, as well as no control over it. The SSD is in the eyes of the DBMS just a storage with a continuous address space, supporting reads and writes on immutable addresses. On the other side of the wall, the Flash management algorithms of the SSD's FTL have no access to rich DBMS knowledge about the data being stored on the Flash. This **lack of information and control** on both sides creates the lose-lose situation.

- The GC is a major influencer of the overall SSD performance. The GC overhead itself is characterized by the amount of performed page migrations (i.e. write-amplification of the SSD) and erases required to reclaim the space occupied by invalidated versions of data. Mapping scheme, as already mentioned, is one of the factors determining the GC overhead. Another, more significant factor, is the proper data placement strategy. The ability to place the data with similar update frequency (data temperature) together is the major knob we can use to reduce the number of page migrations performed by the GC. This is often called hot-cold data separation strategy. Under the FTL-based design of the modern SSDs, however, **efficient data placement is almost impossible**. The DBMS, which has the comprehensive metadata required for the proper hot-cold data separation, has no control over the physical data placement on Flash. Writing a data to a certain logical address (LBA) the DBMS is completely unaware about the resulting physical location of data on Flash. On the other side, the FTL algorithms responsible for placing the data have no access to the metadata of the DBMS. Thus, the only way for the SSD to provide hot-cold data separation is to collect and analyze the device internal statistics about the update frequencies of individual pages. However, the limited on-device DRAM resources prevent this approach to be realized efficiently. As a result, on average, SSDs experience the write-amplification of about 3.5x, i.e., every single incoming write request results in 3.5 physical writes on the Flash. This increases IO response times and shortens the lifetime of the SSD.
- Further, the address indirection provided by the FTL makes the **typical data placement strategies of the DBMS and file system useless**. Extents, tablespaces, files and partitions are becoming, in the case of SSD-based storage, solely logical data structures and have no impact on the physical placement of data on Flash memory. In other words, data supposed to be placed continuously (or separately from each other) by an application, in reality might be distributed randomly over the whole Flash memory. Therefore, a large part of the effort of the DBMS or file system for placing the data continuously (crucial for the HDD) is senseless for SSDs.

- 
- The lack of information about the SSD's internals and the inability to control physical data placement by the DBMS has also a negative impact on the efficient **utilization of the available Flash parallelism**. To utilize the IO parallelism of underlying Flash memory the typical approach of the FTL is to distribute the data evenly (based on its logical address - LBA) over the independent units of the Flash (data channels, Flash chips/dies, planes). Although this strategy typically results in the perfect load balancing and simplifies wear-leveling algorithms, it is usually losing to smart data placement by delivering in the general case 2-3x lower performance of the SSD. Smart data placement uses data semantics and access statistics to distribute the data on Flash in a way that allows minimizing write-amplification (GC overhead) and maximizing the gain from available IO parallelism by managing it in a demand-based manner (e.g. while hot data requires often high level of IO parallelism, cold data does not). Solely the DBMS (not the FS nor the OS) has all the required information to perform smart data placement. However, the FTL-based, black-box design of SSDs provides no means to realize such strategies.
  - The efficiency of separate FTL algorithms (address translation, GC, WL, ECC) was a hot topic for both database and storage communities in recent years. Although many research approaches were proposed, neither of them could be marked as universally optimal, since their efficiency is strongly workload dependent. On the other side, every SSD nowadays utilizes only one (which one is the manufacturer's top secret) predetermined variation of those algorithms. This makes each particular FTL – a “**one size fits all**” solution. This is another consequence of the SSD's black-box design. As a result, the complete performance of the storage becomes **workload dependent**. Not surprisingly, it is common to experience an order of magnitude difference between the numbers in a device's specification and those measured under various real workloads. Beside the workload dependency the SSD's **performance is often unpredictable** for the system, due to the complete unawareness of the host about the internal FTL processes on the SSD.
  - Another disadvantage of the FTL and backwards compatibility of SSDs with HDDs is the **underutilization of the computational power of SSDs**. Modern SSDs are equipped with various powerful processing units (multi-core controllers, FPGAs, GPUs). Currently those are utilized solely for execution of FTL tasks. However, they might be used efficiently for data processing on behalf of the DBMS. This would significantly minimize data transfer, as well as the “pollution” of the host's DRAM and CPU cache. The realization of such near data processing is, however, impossible under the black-box design of SSDs and the use of the traditional block-device

---

---

interface.

This list of drawbacks and disadvantages of modern FTL-based SSDs is not exhaustive. A deeper look into the DBMS subsystems shows many places that might be significantly optimized by having a direct control over Flash memory. Query optimizer, buffer manager, transaction manager and recovery algorithms would be the first candidates for such optimizations.

### 1.3. Proposed Approach

If the disadvantages and unrealized optimizations for Flash storage are caused by the FTL and the compatibility mode supported by modern SSDs, then the solution to those problems lies in eliminating the FTL. **In this work we present a concept called NoFTL, which gives the DBMS full and direct control over the underlying Flash storage by removing typical abstraction layers on top of it, such as file system, block device abstraction and FTL.** The efficient realization of this concept is based upon three basic principles: native Flash interface, integration of Flash management into the DBMS subsystems, and configurable Flash storage (Figure 1.1).

Once the traditional block device abstraction is removed the DBMS is confronted with the first question: how to “speak” to Flash storage without FTL. To answer this question we introduce the concept of native Flash interface. The interface is open and flexible, and thus might be developed further depending on the needs and system resources. Its major purpose is to enable the DBMS to control the physical data placement on Flash, and utilize the on-device computational resources to perform near-data processing tasks.

Since the removal of the FTL does not eliminate the necessity of the Flash management tasks, this raises the second question: who takes the responsibility of them? Under the NoFTL design the main high-level tasks of the Flash management are integrated into the subsystems of the DBMS. This integration creates a win-win situation for both the DBMS and the Flash management. On one side, Flash management algorithms and data structures can be significantly optimized by utilizing comprehensive metadata and statistics of the DBMS, as well as rich memory and (typically underutilized) computational resources of the host system. On the other side, the ability to perform direct physical data placement on Flash and control Flash management tasks leads to optimizations and simplifications in many places inside the DBMS. For instance, the control over the out-of-place update strategy and garbage collection on Flash allows for significant reduction in the overhead required for the support of IO and transactional atomicity by the DBMS. It is worth to note, that the implementation of Flash management inside the DBMS does not

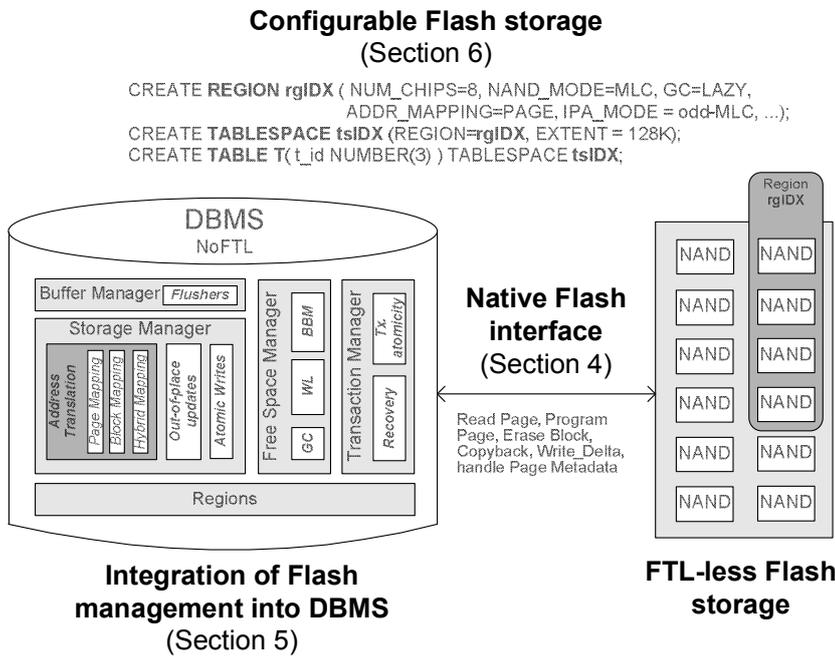


Figure 1.1.: Pillars of NoFTL approach.

add much complexity to the latter. Every DBMS has its own address translation, free space management, and garbage collection; the corresponding functions and data structures of Flash management can be smoothly integrated into them. This results in the reduction of functional redundancy along the IO path. It is important to observe that only the high-level Flash management functionality is integrated into the DBMS, the low-level Flash management remains the responsibility of the on-device controller.

The open native Flash interface can be seen as a toolbox containing different tools allowing the DBMS to operate on the FTL-less Flash storage. The integration of Flash management into the DBMS subsystems is then a set of techniques and methods to work with those tools on Flash. The third guiding principle of this dissertation answers the question: how to apply these techniques efficiently, and what technique is best suited for a particular system and workload. Those questions are answered by the concept of configurable Flash storage. In contrast to the “one size fits all” design of modern SSDs, NoFTL optimizes the usage and management of Flash storage by clustering the data with similar properties physically on Flash, and by applying the most appropriate set of Flash management algorithms for the particular cluster. In order to realize this concept

---

we introduce two novel data structures – region and group. These structures allow us to manage the physical address space of Flash and selectively apply Flash management algorithms. Regions and groups are efficiently coupled with the traditional DBMS concepts of tablespaces, files and extents, thereby returning to the DBMS the power over the physical data placement. Depending on the properties of data assigned to a particular region, the latter is managed by the most appropriate set of Flash management techniques, i.e. variants of address translation, GC and WL. Those local optimizations significantly reduce the overall overhead of Flash management. Moreover, using regions the DBMS can control the utilization of available Flash parallelism, with the purpose to maximize its benefit. Thus, the configurable Flash storage gives the DBMS flexibility in how data is placed on Flash and how the storage is managed.

The native Flash interface, the integration of Flash management functionality into the DBMS subsystems, and configurable Flash storage are the three pillars of the NoFTL concept, which enables the DBMS to obtain maximum performance from Flash storage, avoiding the disadvantages of the FTL-based, black-box SSD design.

To simplify the configuration of Flash storage for the database administrator two solutions called Advisor and Migrator are provided. The NoFTL Advisor monitors the IO statistics of the DBMS, as well as the statistics of Flash management, and based on this information assists the database administrator in selecting the appropriate configuration. The transformation of the storage from one configuration to another is carried out by the Migrator module. The migration process is performed in the background and can be configured by the DBA over tuning knobs.

## 1.4. Contributions

We see the main contribution of this research in describing the novel storage alternative for the DBMS – NoFTL, which assumes operation on FTL-less Flash storage. NoFTL eliminates all significant disadvantages resulting from the black-box design of modern SSDs, and enables the DBMS to utilize the whole performance potential of Flash SSDs. To prove this concept we have implemented the NoFTL prototype in an open-source DBMS and evaluated it under various scenarios on real Flash storage, as well as on a configurable and precise Flash emulator. NoFTL should not be seen as a finished solution. We believe that NoFTL opens the door for many even more significant DBMS optimizations on modern Flash storage. We further believe, that the main idea of NoFTL – DBMS on native storage – is going to prove itself for the next generation of storage based on non-volatile memory technologies. This assessment is based not only on our own results, but is also confirmed by developments in industry and academia.

---

The various optimization approaches we designed, implemented and evaluated in this research under the general NoFTL concept are rooted in different subsystems of the DBMS, mainly buffer, storage, transaction, and recovery managers. Performance gains achieved via those optimizations are primary due to the reduction of IO and computational overheads, as well as better utilization of available Flash parallelism and on-device computational resources.

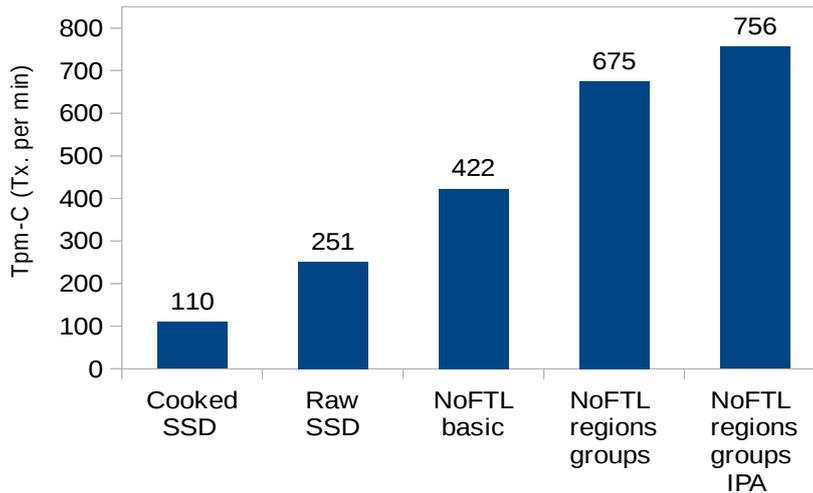


Figure 1.2.: Performance comparison of DBMS on different storage alternatives under TPC-C benchmark.

For instance, by applying smart data placement and Flash management we could reduce the write-amplification of Flash SSD for OLTP workloads by 2-15x depending on the system configuration. Another example is the reduction of performed erase operations for a particular workload by up to 80% thanks to one of the most significant approaches we provide – in-place appends (IPA). These reductions of GC overhead have two main implications. First, the reduction of IO response times and consequently the **increase of the overall performance of the storage**. The extent to which the improved IO performance positively impacts the system performance depends on the IO-boundedness of the latter. Thus, we experienced the improvement of the system’s throughput by up to 7x for IO-bound configurations (see Figure 3.5). The second important implication is the **prolongation of the SSD’s lifetime**. The NoFTL approach can improve the longevity of SSD by twice and even more under the OLTP-like workloads.

Continuing the already mentioned statement that FTL-based Flash SSD is the storage of

---

---

today, we strongly believe that the principles underpinning the proposed NoFTL approach will characterize the Flash storage for data-intensive applications of the future. We dare to forecast that FTL-less Flash SSDs will hold the dominance on the storage market for the next decade.

## **1.5. Structure of the Thesis**

On the whole, the flow of the thesis corresponds the chronological evolution of the research underlying it. As the construction of a building follows a certain order, so the research and realization of the NoFTL concept was following its plan. To start our construction project we first must discover the area, its past and current developments. We do this in the next Chapter by providing necessary background information regarding the DBMS, IO stack and Flash. We continue then in Chapter 3 with general overview of the proposed NoFTL concept – the architectural plan for our building. As already mentioned, the NoFTL is founded on three main pillars – the native Flash interface, the DBMS integration of Flash management, and the concept of configurable Flash storage. Their design, implementation and evaluation details are provided in Chapters 4, 5 and 6 respectively. The sustainable base of the NoFTL concept allows the realization of various optimizations for the Flash management algorithms and in different subsystems of the DBMS. While multiple of those optimizations are described during the building of corresponding pillars, in Chapter 7 we present the approach called IPA. This approach was designed and developed in the final phase of this research and is especially interesting because it shows the whole NoFTL concept with its three pillars “in action”. Chapter 8 describes the idea and example solutions that help to find the proper configuration of the Flash storage under NoFTL. The analysis of the related work is provided in Chapter 9. We conclude and discuss future work in Chapter 10.

---

## 2. Background

---

The following chapter is dedicated to the necessary background knowledge on database management systems and Flash storage. It consists of short surveys for the relevant topics, which allow us to provide quick, in-place references to other chapters. Please note, that related research work is described separately in Chapter 9, which is placed at the end of the thesis to enable better comparison with the proposed approach.

### 2.1. DBMS

#### 2.1.1. Evolution

For more than thirty years since the 1980s the term DBMS was primarily associated with relational database management systems (RDBMSs), having their begin with the pioneering System R and Inges projects based on Codd's seminal paper on the theoretical aspects of relational systems.

RDBMSs were designed for the management of structured data. The data here is organized according to the schema in tabular format and is queried via a query language, Structured Query Language (SQL) being the de facto standard. Through extensions the RDBMSs could successfully cover almost all possible data domains and satisfy the needs of the corresponding data-intensive applications. The database market grew soon to the billion range and is estimated to reach \$50 billion this year. The three major vendors of RDBMS software holding together more than 85% of the total database market's revenue are Oracle (Oracle DBMS), IBM (DB2) and Microsoft (Microsoft SQL Server). There are also multiple open-source alternatives, which are successfully competing with the proprietary software and continuously increase their market share. The most popular examples of open-source RDBMSs are MySQL, PostgreSQL and SQLite.

However, about a decade ago the monopoly of the relational databases finished with the establishment of so-called NoSQL database management systems. The major reason stimulating the (rush) development of those systems was the tremendous increase of the amount and importance of unstructured and semi-structured data. Petabytes of video and audio files, documents in dozens of possible formats, event logs, crawling data and

---

web indexing are the main examples thereof. Regardless of the physical location, variety of formats and purpose the Internet is by far the major place of residence for this data. For YouTube, Facebook, Google, Amazon and many other Internet companies the ability to store and process it is crucial for the business model. The use of RDBMSs for the management of this data is, however, typically impractical. First, the relational model is less suitable for semi-structured data, and almost useless for unstructured data. Second, the amount of the data is typically far beyond the capacities of a single server and often even beyond a single data center. Thus, highly distributed data management systems are required. Although the RDBMSs can scale well by increasing the resources of a single machine (scale-up), they are typically weak in the distributed environments. By horizontal scaling (scale-out) the provision of basic RDBMS guarantees becomes challenging. Especially the transactional model, which provides the atomicity and consistency properties (A and C from basic ACID properties of transactions) is hard to realize efficiently in the environment with hundreds and thousands of machines distributed all over the world. The strict consistency of changes over high (dynamic) number of replicas or shards would cause high locking overhead and increase response times drastically, which in turn will slow down the system performance and throughput.

In contrast to RDBMSs, NoSQL data management systems are specifically designed to manage mixed data (unstructured and structured), they provide lower guarantees regarding data consistency (and sometimes durability), while increasing the availability and processing performance via ease of horizontal scaling (ACID vs. BASE). There are four main types of NoSQL systems: key-value pair systems, document management systems, (wide-)column stores, and graph databases. NoSQL database systems, their comparison and popular examples are well described in [19] and [25].

However, most of the enterprise data is still structured and is also transactional, which means that only the RDBMSs with their strong consistency guarantees, powerful query language and schema-based data model will be the data management systems of choice in the foreseeable future. The revenue numbers prove this statement: while taking into account that the market for commercial DBMSs (primary choice of enterprises) is continuously increasing, the NoSQL market share is estimated to be less than 5% nowadays for all NoSQL products together. The non-revenue market share for NoSQL is, however, significantly larger than that, since currently the majority of NoSQL products are open-source. Saying that, it is important to understand that it is not about “Who wins?”, because SQL and NoSQL data management systems are not really competitors, but rather complementary systems. Both products are good in a certain data domain (structured vs. unstructured) and for certain requirements (consistency vs. scalability). Therefore, both will co-exist in the future. Already today the major vendors of traditional RDBMSs start offering also NoSQL products coupled to their relational systems.

---

The problem of sub-optimal use of modern storage hardware considered in this work is the general problem for both SQL and NoSQL systems. Although, the same is valid also for the basic concept of the proposed solution, the implementation details might differ significantly. In this work we concentrate primary on the traditional relational database management system (referring to them simply as DBMS).

### 2.1.2. Architecture of Relational DBMS

Figure 2.1 presents the module-based architecture of the classical relational DBMS. Despite the variety of today's database systems, their design and implementation differences, the basic functional components are present in all of them. Each of those is complex and consists of multiple sub-modules. Although the detailed description of the DBMS architecture and the interplay of its modules are well described and can be easily found in classical database textbooks ([80], [22], [38]), we decided to provide here a brief recap for the purpose of completeness.

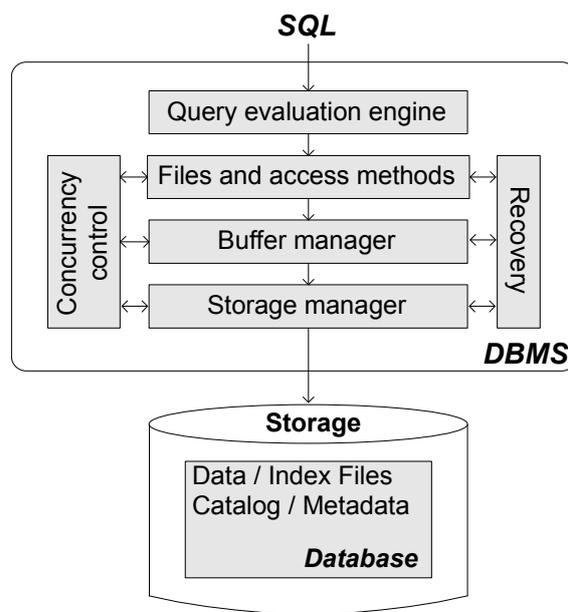


Figure 2.1.: Module-based architecture of the relational DBMS (based on [80]).

Processing of incoming SQL queries begins with the Query Processor. The key competence of this module is an establishment of the query plans and control of their execution.

---

In order to find an optimal execution strategy the DBMS utilizes various metadata, such as catalog information, access statistics, histograms, runtime data, etc. The query plan decisions are made not only regarding the kind of the operators, but also about their execution order. The emerging computational and storage hardware with high level of supported parallelism force the DBMS designers to revisit traditional operators and data structures in order to increase intra-query and intra-operator parallelism. Thus, the efficient query optimizer must be able to recognize possible ways of parallelizing the query execution based on the available hardware resources. Flash storage is the perfect example of hardware development which offers the possibility to significantly speedup the execution of major query operators. Read/write asymmetry and I/O parallelism are the main properties of Flash SSDs allowing these improvements. Further, the rich on-device computational resources of modern SSDs allow for execution of query operators on the storage device itself, increasing thereby the computational parallelism, and saving data transmission costs, as well as reducing CPU cache and buffer pool pollution of the host system. Those optimizations require, however, a re-design of large parts of the logic of DBMS's modules, as well as an additional support from the storage device.

Once the query plan is defined and the query processor starts with the execution of operators (e.g., selection, projection, joins, modification, aggregation, etc.) the second layer of the DBMS comes into the play - Files and Access Methods. This layer is responsible for the organization of data in logical database structures such as files and indexes, as well as internal organization of data records within those structures. It provides, therefore, interfaces for the query operators to locate, access and modify the data. The standard data organization layouts supported by the majority of DBMSs today are heap, sorted and hashed files, as well as B-Tree indexes. Bitmap, UB-Tree or R-Tree indexes are further alternatives supported by some database systems.

The key challenge of introducing a novel data structure or access method is the efficient support of concurrency and recovery. In recent years the database community actively looked into the data access layer in order to better address the properties of Flash SSDs. Prioritization of reads for writes, parallel access and the out-of-place update strategies are common ideas underlying the proposed novel or modified data structures. In this work we touch this layer multiple times while implementing the NoFTL architecture. Especially the approach of In-Place Appends (IPA) (see Chapter 7) required modifications in page layout for table and index data.

The next essential module of the DBMS is the Buffer Manager. Once the data requested by the operators is being located using appropriate access routines, the Buffer Manager is asked to provide access to the corresponding database page. If the look-up in the internal hash table was successful, the reference to the buffer frame containing this page is returned. Otherwise (i.e., cache miss), the Buffer Manager issues an I/O request to

---

---

the underlying Storage Manager to read the page from the storage and place it into the reserved free buffer frame. In the meanwhile, the replacement policy is responsible for the decisions regarding when and what data should be removed from the buffer pool. Thereby, unmodified data is simply evicted from the cache, while modified pages are written back (flushed) to the storage. The flushing API of the Buffer Manager is also used by other database modules, such as Recovery and Transaction Managers, for guaranteeing atomicity and durability properties of the DBMS.

The influence of the buffer's replacement strategy on the database performance correlates with system's dependency on the I/O stack. In systems where the working set cannot be cached completely in the buffer, the improper replacement strategy can easily become a significant bottleneck. Especially, if the workload mixes large table scans with small OLTP-like accesses, an intelligent Buffer Manager is vitally important. The efficient replacement strategy must take into account not only the simple statistics about frequency (e.g., LFU) or recency (e.g., LRU) of page accesses, but also the information about the workload, the query plan and the current operator causing those accesses, as well as the type of database pages (e.g., index or table pages). Also, the type and characteristics of storage have an influence on the Buffer Manager. The clear latency asymmetry of read and write requests of the Flash storage (e.g., 10x and more) encourage to revisit the replacement strategy. The recent approaches addressing those issues are covered in Chapter 9. In this work we have looked at the Buffer Manager considering yet another important property of Flash SSDs - parallelism. We do not change the way the replacement strategy decides about what and when to write out back to storage, but rather how the selected pages are being flushed. We show that (i) by utilizing information about the internal architecture of the Flash storage, and (ii) having a control of the physical data placement on Flash - both of which have first become available in the NoFTL architecture - the DBMS can speed-up the flushing process by about 50%. This is achieved by better utilization of a device's parallelism and reducing the contention for physical resources between DBMS flusher threads (page cleaners). We did further modifications to the Buffer Manager module in connection with the IPA approach. The detailed description of those approaches and the required modifications are presented in Chapters 5.4 and 7, respectively.

The nearest to the physical storage is the Storage Manager module of the DBMS. It is responsible for the space management on the storage, which includes among others storage access routines, allocation and deallocation of space for database structures, and mapping of those structures to physical storage addresses (or files and their offsets in case of using a file system). This module was getting the lowest attention from the DBMS research community and is, therefore, the least modified one over the last couple of decades. The reason for this is an assumption of operating on storage devices which support the block device interface. Having only a hard disk drive as a reasonable storage

---

---

for more than three decades made this assumption an indisputable rule. Flash as a viable storage option was made possible mainly due to the support of a block device interface, i.e., SSDs were made backwards compatible with the traditional HDDs. In contrast to this, the NoFTL architecture removes the backwards compatibility of Flash storage, gives the DBMS full and direct control over the Flash by means of the novel native Flash interface. As a consequence, the Storage Manager of the DBMS has been rewritten to support FTL-less Flash SSD. More information about the changes in this module are provided in Chapters 4 and 5.

The next important module in the simplified DBMS architecture is the Transaction Manager. Its main task is to provide concurrency control. The efficient support for parallel execution of transactions is the most critical and important functionality of modern DBMSs. Not surprising is, therefore, the attention to this module from industry and academia over the last decade, motivated by the continuously growing gap between hardware parallelism and concurrency level of the DBMS. In fact, the majority of today's database systems utilize traditional concurrency models, which often are going back to the pioneer systems in the late 80s. The typical mechanisms to support different levels of transaction's isolation are locking, multi-versioning and tracking of transactional access history (e.g., in an optimistic concurrency control model). Although, compared to other modules, this module is only weakly coupled to the characteristics of physical storage, careful consideration of Flash specifics allows us to optimize traditional approaches. For instance, the implementation of multi-version concurrency control can be significantly improved by considering the out-of-place update principle of Flash SSD. More details about recent research in this area are provided in Chapter 9.

The last module in our simplified architecture of the DBMS is the Recovery Manager, which is, however, definitely not less important than others. Although, its core competence is to guarantee the durability of data after failures or system crashes ("D" in ACID properties of RDBMS transactions), it typically assists also the Transaction Manager to realize atomicity and isolation properties of a transaction's execution ("A" and "I" in ACID). The common principle for all systems to provide the recovery functionality is based on maintaining a certain level of data redundancy. Database systems realize this via logging. Logs are small records describing a single modification of data item(s), which can be used to recover the corresponding data in case of losses or data corruption. The majority of today's relational database systems are implementing a kind of ARIES-like recovery. Its main principle is based on persisting the log records before writing the corresponding modified data items to the stable storage, the so-called Write-Ahead Logging (WAL). Flash storage with its "native" support of data redundancy caused by the out-of-place update strategy allows for significant simplification of these traditional recovery mechanisms. In other words, on writing a modified database page out of the buffer pool to the Flash

---

SSD, there is always a previous unmodified version of the page present on the storage. This "backup" page can be used for recovery purposes. However, modern FTL-based SSDs hide completely the out-of-place update behavior from the host system, making thereby this "free" recovery support of Flash being inaccessible for the DBMS. In contrast, under the NoFTL architecture the DBMS performs direct physical data placement, and thus has full control over the out-of-place update strategy, which allows us to use it for recovery optimizations. We discuss similar approaches proposed in recent years, as well as our ideas in Chapter 9. Our modifications to the Recovery Manager required for the basic realization of NoFTL are provided in Chapter 5.5.

Apart from the mentioned six modules, database systems must implement also other important functionalities, which might be realized as separate modules or be a part of the above. Those include authorization, metadata management, memory management, administration, etc. Those modules are typically completely independent of the underlying storage infrastructure and are, therefore, out of the scope of this work. In our NoFTL prototype implemented on top of an open-source storage engine they are left unchanged.

## 2.2. DBMS Storage Alternatives and I/O Stack

In this chapter we continue to follow the path of user requests submitted to the database, but move our attention now outside the DBMS. As we saw, the last stop of requests within the database system is the Storage Manager, where I/O requests are formed and sent down in the direction of the storage device. However, the way to the destination is typically quite long and full of further stops. Because this way is the major cause of disadvantages and limitations of modern Flash SSDs, we describe it in more detail.

There are two traditional ways for the DBMS to operate on physical storage - with and without a file system on top of a block device. The storage alternative with the file system - also known as "cooked" storage (Figure 2.2-A,B) - was, and continues to be, the most widely used, and thus it is supported by all of today's database systems. The alternative without a file system is typically referenced as "raw storage" (Figure 2.2-C), and is also available as an option by the main DBMSs on the market. Common for both alternatives is, however, the utilization of the block device interface and block I/O infrastructure (encapsulated in the so-called *block-device layer*).

The backwards compatibility of modern Flash SSDs allows (or better said *forces*) them to utilize the very same I/O stack as used by traditional HDDs. That means, that block device interface is common for both SSDs and HDDs, and consequently two mentioned storage alternatives (cooked and raw) are applicable for both types of storage devices as well. In the following we describe major characteristics of these alternatives and of

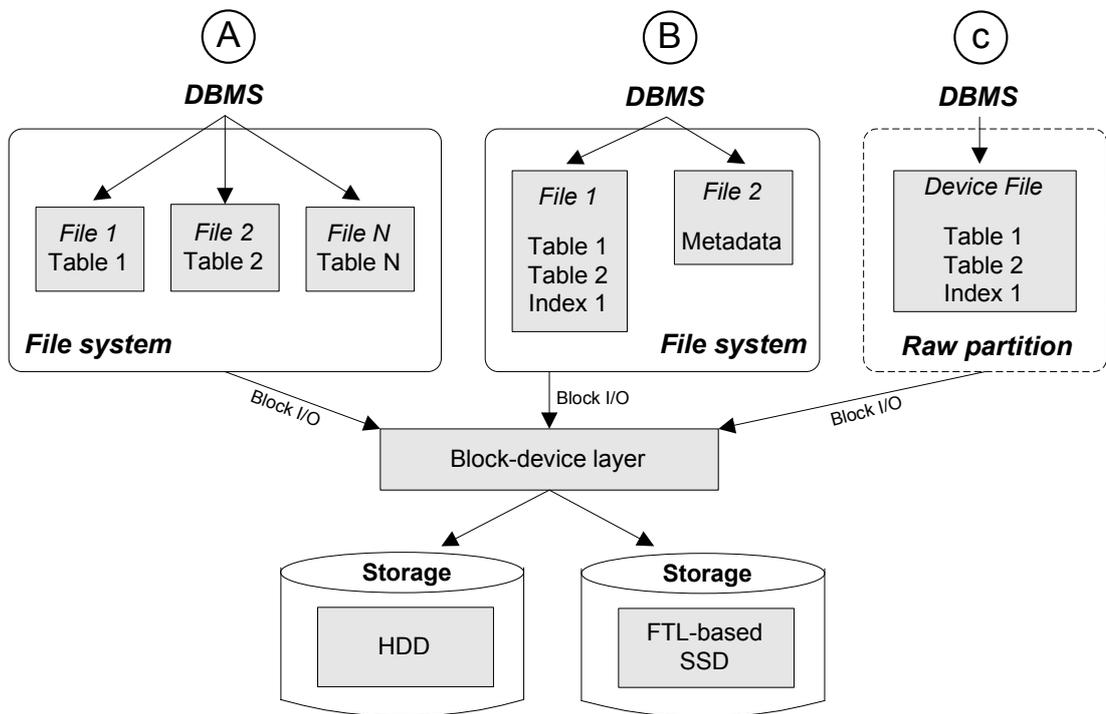


Figure 2.2.: DBMS storage alternatives.

the block device layer from the perspective of the HDDs. In the subsequent chapter we concentrate on the properties of Flash memory and Flash SSDs, and thereafter we come back again to the I/O stack and revisit it now from the perspective of Flash SSDs.

### 2.2.1. Cooked Storage

The use of a file system allows the DBMS to organize whole data as a set of files and directories. The main advantage of this approach is the simplification of the data administration on the DBMS side, achieved by delegating the responsibility for space management (at least the most part of it) to the file system. Further, the file system simplifies the management of data access permissions; makes the DBMS support of portability across multiple OSs easier; and allows the use of external tools to access database data (e.g., external data backup).

However, using a file system has also its down side for the DBMS. The negative influence

---

---

of the FS on the database performance might vary significantly depending on the particular file system, its configuration, storage media and DBMS data placement strategies. The major reasons for the possible FS-related bottlenecks might be classified into the following groups:

- partial or complete loss of control over the physical placement of data on the storage;
- partial or complete loss of control over the time *when* the data becomes physically written on the storage;
- computational, memory and storage overheads due to redundant or unnecessary functionality.

While the detailed discussion of the bottlenecks caused by file systems can be found in [68], [59] and [73], in the following we briefly summarize them based on the above grouping.

As already mentioned in the previous chapter, the traditional DBMS design is strongly influenced by the characteristics of HDDs. Thus, the algorithms to keep I/O requests as sequential as possible are present in almost all modules of the DBMS: query processor, access methods, buffer manager, storage manager and recovery. This is not surprising, since the difference between random and sequential accesses on modern HDDs is typically in a range of one order of magnitude. However, having a file system as an additional abstraction layer between the DBMS and the storage typically has a negative influence on the DBMS's efforts to place the data in a close physical proximity. For instance, depending on the allocation strategy performed by the FS (e.g., contiguous, linked, indexed, extent-based, hybrid) and an alignment of FS-blocks to database pages, FS might map and place contiguous blocks of data on random positions on the storage.

Another bottleneck the file system can produce for the DBMS, is caused by the utilization of write buffer<sup>1</sup>. By delaying the write requests of the DBMS, the file system might potentially violate the durability ("D" in ACID) property guaranteed by the database system. In turn, forcing the FS to perform immediate writes (e.g., via *fsync*) might result in an additional overhead [73].

Another group of disadvantages is caused by performing some functions redundantly by both the DBMS and the FS. The typical examples here are caching (read cache), prefetching and recovery. If the file system (or OS) caches the DBMS data, which has been recently read from the storage, this simultaneously means that there are two copies of the same data kept in the system's memory - one in the DBMS internal buffer and another in the OS cache (e.g., *page cache* in Linux). Apart from the obvious memory overhead, this

---

<sup>1</sup>Although it is typically implemented as a part of OS, the file systems rely on it, or even might extend it.

---

caching redundancy produces also computation overhead resulting from the maintenance of the second cache and copying the data from the FS/OS cache to the DBMS buffer [73]. Moreover, the FS prefetching strategies might in some cases pollute the system's memory with irrelevant data. The typical example of such a situation is performing a sequential scan of linked B-Tree leaf nodes for answering a certain range query [38]. The DBMS can easily predict the sequence of nodes to be scanned, and thus corresponding pages can be efficiently prefetched. In contrast, neither OS nor the file system could correctly estimate the pages going to be read next, because commonly the linked leaves of a B-Tree are not logically sequential, i.e., their LBAs are not contiguous. As a result, the file system will try to prefetch "useless" pages, polluting thereby the system's memory and producing an unnecessary overhead of read I/Os.

Even more significant overhead issues for the DBMS might be caused by a file system's recovery strategy. Modern file systems typically offer multiple recovery modes, allowing the user to trade-off between performance (influenced by additional I/O overhead) and different recovery possibilities in the case of system crashes. Thus, for instance, *ext4* and *ext3* file systems provide three recovery modes, known as *journal*, *ordered* and *writeback*. In the recommended, "safest" *journal* mode the user data and the internal file system's metadata are written first sequentially to the dedicated region (file), called journal, and only after successful "journaling", the data is written again to their corresponding addresses. After a crash, the file system can easily<sup>2</sup> determine inconsistency in its metadata or user data, which occurred due to the uncompleted write sequence. By re-doing these requests with the data from the journal, or by simply ignoring them in case of incomplete writes to the journal, the file system is returned to the consistent state again.

This journaling concept is very similar to a WAL strategy traditionally used in early relational DBMSs (which is clear, since it was inherited from the latter). However, journaling alone (in its most consistent mode) would be insufficient for the DBMS, since although it can guarantee the consistency of a file system, it could not guarantee atomicity, consistency and durability properties for the database transactions, and thus for the database data as a whole. First of all, this is because the file system has no access to the transaction's semantics (e.g., which I/Os belong to the particular transaction), and no control about their execution. As a result, no database system is relying on the FS journaling, but rather "cares" always itself about the consistency of its data (see *Recovery manager* in Chapter 2.1.2). This makes the journaling of database data by the FS redundant, and thus unnecessary, since it does not give the DBMS any further guarantees for data recovery after system crashes. But, this redundancy might produce a significant I/O overhead for the

---

<sup>2</sup>Analyzing a small journal file takes negligible amount of time as compared to performing a scan over a whole file system, e.g., *fsck*.

---

whole system. For instance, if the *ext4* file system is configured in the *journal* mode, every single data update by the DBMS might be written practically four times on the storage - (i) write of the database log record to the file system's journal; (ii) write of the same record from the journal to the end of the database log file; (iii) write of actual modified database data to the journal; (iv) write of this data from the journal to the corresponding database file. Besides at least doubling<sup>3</sup> the amount of data written to storage, FS journaling can provide further performance degradation by influencing the write pattern of I/O requests. Especially, if the I/O requests submitted by the DBMS are sequential, the journaling might turn those into random by introducing additional writes to the journal in between.

Although the largest part of write-amplification produced by the file system can be eliminated by turning off the journaling for the application data (e.g., *ordered* and *writeback* recovery modes in *ext3* or *ext4*), the journaling of the file system's metadata is typically unavoidable. Thus, even in the lightest configuration file systems produce a certain additional write-amplification (e.g., 25% more I/Os) and influence the pattern of I/O requests.

There are two basic approaches of how database systems utilize the file abstraction for storing data. The first practice is to assign every database object to one or multiple files, i.e., one-to-many relationship (Figure 2.2-A). For instance, every table/index is stored initially in a separate file. By growing in size, new files for storing object's data might be allocated, so that large tables or indexes span over multiple files. The alternative strategy is to store multiple or even all database objects (or their parts) in a single file (Figure 2.2-B), while multiple of those files can exist, i.e., many-to-many relation between objects and files. Without going deeper into the discussion of pros and cons, it is worth to mention that in recent years there is a clear trend towards the latter strategy. By storing the data in only few large files the DBMS can reduce certain disadvantages resulting from the use of a file system. For instance, the initial allocation of a large file (especially on a relatively empty storage) increases the probability that logically contiguous blocks of data would also get contiguous device addresses assigned by the file system. If the underlying storage is the hard disk drive, then the data would be placed also continuously on the physical medium, i.e., the DBMS can to a large extent control the physical placement of data. Consequently, the DBMS can better leverage the performance gains of the sequential I/O patterns on HDDs. Thus, this strategy becomes a widely used trade-off for DBMSs on HDDs, which allows to combine some advantages of file system-based and raw storage (described below). Berkeley DB, SQLite and ShoreMT -are just some examples of popular

---

<sup>3</sup>The actual write-amplification of the *ext3* or *ext4* file systems in *journal* mode in terms of the amount of written data might be as high as 5x, and in terms of I/O count be more than 3x for OLTP workloads [68], [59].

---

DBMSs that utilize this strategy and store a complete database in a single file.

### 2.2.2. Raw Storage

Most of the commercial database systems on the market today offer the possibility to operate without the use of file system as storage abstraction. This storage alternative is also known as "raw storage" (Figure 2.2-C). The rationality behind it is many-fold.

First, the DBMS gets more control over the physical data placement. As already mentioned, the performance of HDDs varies drastically dependent on the access patterns, e.g., sequential accesses can be up to 100x faster than random. To leverage this property the data should be placed on the drive in a way, which would "sequentialize" the access pattern for the current workload as much as possible. For this task raw storage has a significant advantage over the cooked storage - the direct mapping from logical to physical block addresses. This mapping requires actually no address translation table, but is rather realized by simple constant offset calculation<sup>4</sup>. This ensures that blocks with contiguous LBAs would have also contiguous physical addresses on the storage<sup>5</sup>. Thus, by managing the logical address space, the DBMS fully controls physical data placement. In contrast, under the cooked storage alternative the file system manages the physical address space on its own, creating therefore an additional level of address translation between the DBMS and the physical storage. As a result, the DBMS can only indirectly and to a certain extent influence the physical data placement.

Another advantage of the raw storage is the lean I/O stack and the minimal memory and computational overhead. As already mentioned, caching (buffering), prefetching and recovery are the typical functions of the file system that become redundant and overwhelmingly expensive for applications like DBMSs. Moreover, the elimination of FS/OS caching gives the DBMS under the raw storage better control over the time when a certain data gets physically persisted, which solves the issues related to the DBMS' guarantees regarding data durability in the cooked storage stack.

A general performance comparison of both storage alternatives is, however, difficult to perform, as there are many factors that have a significant influence on it. The DBMS and its configuration, the file system, the workload, the storage and the amount of RAM are the main parameters, which define a huge space of possible system configurations. For instance, under the workload with completely random access pattern, the direct control over the data placement in the raw storage would not bring any advantage over the cooked alternative; but it will still outperform the latter due to the reduced memory and

---

<sup>4</sup>offset = partition\_offset + (LBA \* block\_size)

<sup>5</sup>With except of cases where due to the "dead" sectors the HDD would internally remap them to reserved sectors.

---

computational overhead. Yet, for DBMSs with ample memory that are CPU-bound the performance advantages from the raw storage would be significantly less than for an I/O-bound system. Some performance numbers can be found in [73], [59], [68], as well as in Chapter 5 of this thesis.

Despite these performance advantages of the raw storage alternative on the HDDs, many database setups today still opt for the cooked I/O stack with the file system between the DBMS and storage. The reasons for this are diverse.

One of them is that the database data stored in files can be easily accessed by applications other than a DBMS. For instance, separate backup applications are working typically only with file abstractions, and not with the raw disk partitions. Also, under the raw storage the whole disk partition must be dedicated solely to the DBMS, which might under certain circumstances result in poor space utilization. Assume, for example, a single HDD with 5TB capacity, and that the initial size of the database is just 100GB. If the database is relatively static or has very slow growing rate, then creating a 200GB raw partition and dedicating this to the DBMS might work pretty well. However, what if the database has a higher growing rate, e.g., we estimate it being in one year approximately 1TB. How large should be the database partition - 1TB, 2TB or the whole drive? If we decide for 2TB, then 1TB of our HDD is "reserved" for the second year, and cannot be used by other applications (even for temporal storage of data). And what if the growing rate of data cannot be estimated in advance? Note, that resizing of non-empty disk partitions is a "no go" option, due to the high risk of data corruption or loss. However, those two issues are typically relevant only for small setups. Large and enterprise database instances are typically running anyhow on separate machines dedicated strictly to the DBMS (e.g., database servers, nodes in data centers). Further, modern DBMSs (commercial and open-source) typically provide themselves enough mechanisms to guarantee high reliability and availability, so that there is no need in third-party backup applications.

Another, often mentioned, reason for choosing cooked over raw storage might be encapsulated in the adjective *simplicity*. By shifting the responsibility of physical space management to the file system, the DBMS storage manager becomes leaner. Moreover, to achieve more efficient data placement on raw storage, the DBMS needs to know some physical characteristics of that storage (e.g., track alignment of HDD [86]). This makes data placement strategy to a certain degree device-specific, which is often seen as a disadvantage compared to the abstraction level given by the file system. However, these reasons are rather subjective. The additional code complexity of the storage manager required for the support of raw storage is usually over-estimated.

Yet, the main reason for the low popularity of raw storage alternative is the widespread use of virtual storage techniques, such as RAID, SAN and LVM. All of them create an additional level of abstraction over the physical storage, which, in turn, makes the control

---

over the physical data placement by the DBMS difficult. Since those issues are relevant also for the proposed approach on Flash SSDs, we will touch them in detail later on in this work.

### 2.2.3. Block-Device Layer

Independent of the storage alternative used by the DBMS, all its I/O requests are passing through the generic block layer of the operating system. This layer consists of a complex set of mechanisms, which together provide a block device interface being common for a variety of storage devices, including physical (HDDs, SSDs) and logical (RAID, LVM) devices. Thus, the block device is an abstraction, which allows to hide specific device characteristics, and gives the upper layers (file system, DBMS, etc.) the generic view of the underlying storage. Every such device is represented as a contiguous logical address space, built of fixed-sized blocks, which are accessed via the immutable logical block addresses (LBAs). The block size is defined by the operating system and its value is (i) a power of two, and (ii) between the sector size<sup>6</sup> and the page size<sup>7</sup>.

Apart from being an abstraction, the block layer typically performs caching (queuing) and, if appropriate, the re-arrangement of incoming I/O requests. Traditionally, these tasks are encapsulated in the *I/O scheduler* module of the block layer. Depending on the strategy utilized by the I/O scheduler requests might be merged (coalesced), sorted and consequently cached in a single queue (e.g., *elevator*, *noop*) or multiple software queues (e.g., 3 queues in *deadline*, per-process queues in *cfq*). By doing so, the scheduler tries to satisfy simultaneously multiple goals: minimizing the I/O latency, maximizing the throughput, while avoiding the starvation of the single requests. Afterwards, the I/O requests are added to the hardware dispatch queue, which is their end station in the block device layer. From here they are passed to the device driver layer (e.g., in Linux the so-called SCSI layer is responsible for all SATA and SAS HDDs and SSDs, as well as RAID systems). More details about I/O scheduler and block-device layer in the Linux kernel can be found in [12], [18], [58].

---

<sup>6</sup>Smallest addressable unit of the storage device

<sup>7</sup>Smallest allocation unit in main memory performed by the operating system

---

## 2.3. Flash

The general principles of the Flash memory were born in the early 1980s in the group of Dr. Fujio Masuoka at Toshiba. After he presented the innovation idea<sup>8</sup> in 1984 in [63], the market immediately realized its potential and started the intensive development of new non-volatile memory. Intel worked on the development of the NOR type of Flash memory, and already in 1988 introduced the first commercial product - the 256K NOR Flash chip. In 1987 the father of Flash memory, Dr. Masuoka, published the paper about the NAND Flash [64], and two years later, in 1988 Toshiba entered the new market with the world's first NAND Flash chip [84]. From this point on, the Flash market exploded.

The invention of Flash memory has basically revolutionized large portions of the IT industry. Many portable devices have been re-designed for Flash memory, or only become possible through the use of Flash memory. Digital cameras, video and audio recorders, MP3-players, simple mobile devices and modern smartphones, tablets, e-books, ultrabooks, desktop PCs and servers, routers, TVs, various embedded devices and sensors in all possible spheres of our life (transportation, medicine, meteorology, manufacturing, etc.), and today even household devices like washers and fridges use Flash memory. It is the main long-term persistent storage nowadays. The reasons for the huge popularity of Flash are many-fold: small form factor, low power consumption, noiseless and shock resistance. While those are also important for the desktop PCs and servers, the dominant advantage here becomes the performance characteristics of Flash storage. This is also the major Flash characteristic considered in this work.

There are two basic types of Flash memory: NAND and NOR Flash<sup>9</sup>. NOR Flash is typically used as code execution storage (e.g., XIP, BIOS ROM, memory inside microcontrollers)<sup>10</sup>, while NAND Flash is designed to be a high-density data storage. The brief comparison of both Flash types is presented in Table 2.1. However, since in this work our focus is the data storage for large databases, we will consider only NAND Flash. The following two chapters are dedicated to provide more details about this NAND Flash

---

<sup>8</sup>The full name of the new memory type back then was Flash EEPROM (electrically erasable programmable read-only memory). The name "Flash" was suggested by the colleague of Dr. Masuoka, who found the block-wise erasure process of the memory cells to be similar to the flash in photography.

<sup>9</sup>The names *NOR* and *NAND* were taken because of the similarity of the internal memory organization with CMOS gates. Thus, in NAND Flash memory cells are organized in series, like in NAND CMOS gates, while in NOR Flash those are connected in parallel similarly to NOR CMOS gates.

<sup>10</sup>As of today, due to the lower prices for RAM (DRAM, SRAM), as well as need in larger storage capacity for firmware, the use of NOR Flash for XIP is typically substituted with the combination of NAND Flash and RAM. The code is read from NAND Flash and cached in RAM, from where it is then executed (e.g., BIOS shadowing). Since RAM is still an order of magnitude faster than Flash, this configuration has also performance advantages over the XIP on NOR Flash.

---

memory and modern storage devices based on it.

	<b>NAND</b>	<b>NOR</b>
<b>Density</b>	High	Low
<b>Read speed</b>	Medium	High
<b>Write speed</b>	High	Low
<b>Erase speed</b>	High	Low
<b>Endurance</b>	Medium	High
<b>Cost per bit</b>	Low	High
<b>Power consumption</b>		
<b>Active</b>	Low	High
<b>Standby</b>	Middle	Low
<b>Access</b>	Page-based	Byte-based
<b>Application</b>	Data storage	Code storage

Table 2.1.: Comparison of NAND and NOR Flash memory types.

### 2.3.1. NAND Flash Memory

#### Physics of Flash Memory

The core of Flash memory is the floating-gate transistor (FGT) (Figure 2.3) making a single Flash cell. FGT differs from the normal transistor (e.g., one used in DRAM cell) by having an additional gate - a floating gate. This is located between the control gate and the channel, and is separated from them via two insulating layers<sup>11</sup>, i.e., it is completely unconnected. This gate is basically what makes the Flash memory being persistent. By manipulating the voltages on the control gate and the channel, the floating gate can be charged to a certain level, or be completely discharged<sup>12</sup>. After removing those applied voltages, the charge in the floating gate will remain, and can be "read" any time later

<sup>11</sup>The insulating layer between the floating gate and the channel of FGT is made of a special tunnel oxid, which allows the electrons to move freely through it in both directions. This is opposite to the blocking oxid insulator, which separates the floating gate from the control gate and does not allow the electrons to flow through it.

<sup>12</sup>The process of moving electrons (i.e., charge) to and from the floating gate is known as Fowler-Nordheim Tunneling.

---

on. The amount of this negative charge inside the floating gate represents the bit-code of stored information on this Flash cell.

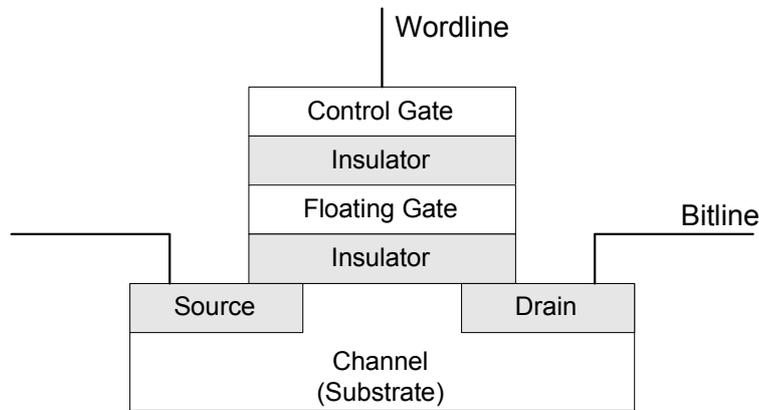


Figure 2.3.: Floating-gate transistor making a single Flash memory cell.

The cells in the NAND Flash are connected in series (Figure 2.4), also called *strings*, such that two adjacent cells share their drain and source connections. This enables to significantly reduce the size of a single cell (by saving on wiring<sup>13</sup>), and thus, to achieve very high memory densities. Each string is connected via special control transistors to the bitline (vertical wire) on the one end, and to the source line on the other end. Row-wise the cells are connected to the corresponding wordlines (horizontal wires), i.e., control gates of every FGT in a certain row are connected to one wordline. The number of cells in a string determines the number of pages in a Flash block<sup>14</sup> (erase unit), while the number of strings crossing each particular wordline corresponds to the size of Flash page in Bits.

The three possible operations - read, program and erase - are performed by manipulating the voltages on the word-, bit- and source-line wires.

- To program a Flash page, the controller applies a high voltage ( $V_{pgm}$ ) to the corresponding wordline, and either no ( $0V$ ) or low voltage ( $V_{cc}$ ) to the bitlines, depending on whether the corresponding cells should receive a charge or not. Applying  $0V$  to the bitline creates a high potential difference between the control gate and the channel of the corresponding FGT, which creates a current flow between them. As a result the floating gate receives a certain amount of charge (electrons

---

<sup>13</sup>Thus, the cell size of NAND Flash is about 60% smaller than the cell size of NOR Flash [93].

<sup>14</sup>The number of pages in a Flash block equals  $K$  times the number of cells in string, where for SLC NAND Flash  $K$  is 1, for MLC NAND Flash  $K$  is 2, and for TLC NAND Flash  $K$  equals 4.

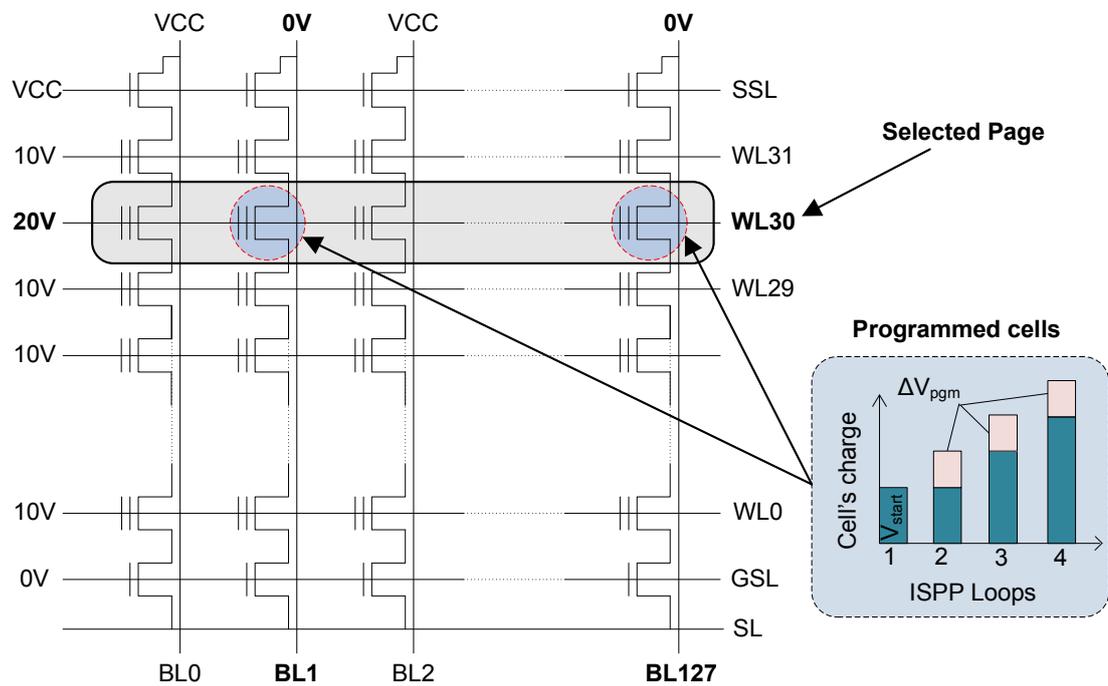


Figure 2.4.: Cell-array architecture of the NAND Flash memory.

are "trapped" on their way from channel to the control gate). Supplying the bitline with the  $V_{cc}$  prevents the current flow between the control gate and the channel<sup>15</sup>, and thus leaves the floating gate of the FGT in the unchanged state. The common approach nowadays is to utilize the so-called Incremental Step Pulse Programming (ISPP) technique to program Flash cells [90, 66]. ISPP programs cells in multiple iterations, increasing every time the cell's charge by a certain amount. After each ISPP iteration the charge in the cells is checked, and if it is lower than the desired charge it is increased in a subsequent iteration. The procedure repeats until each cell in the current wordline has the desired level of charge. In general, ISPP allows to achieve better precision as compared to the alternative "program at once" techniques, and it is more gentle on the Flash, because of lower voltages used during programming.

- In order to read a Flash page, a high voltage ( $V_{read}$ ) is applied to all wordlines in the block, except the one corresponding to the desired page.  $V_{read}$  turns the FGTs

<sup>15</sup>This technique is known as self-booted program inhibit (see Chapter 3.4.1 in [66]).

---

of all irrelevant wordlines into "transfer" mode letting the current flow along the string. At the same time applying no voltage ( $0V$ ) to the selected wordline, causes the corresponding cells to behave differently depending on the charge stored in their floating gates. The cells with no charge allow the current to flow through them, and thus through the whole string. If, however, the cells in the selected wordline contain some charge in their floating gates, the current would not flow through them. Through sensing the current flow at the end of each bitline, the controller can determine the charge (and thus the logical value) of every cell in the selected wordline (page).

- To erase a block, high voltage is applied to all bitlines, and zero voltage is applied to all wordlines in the current block. This creates a high potential difference (similar to the one during the program operation, but with the opposite direction) between the control gate and the channel in every block's cell, which forces the electrons from the floating gates (if they are present there) to move to the channel. The erase operation is often also implemented using the iterative process similar to the ISPP.

### Properties of Flash Memory

The "physics" of the FGT, the organization of NAND Flash, and the performing procedure of three basic operations define altogether the major properties of NAND Flash memory.

The key characteristic of the program operation on Flash is that it can change the cell's charge only upwards, i.e., increase it using ISPP technique. While there is no way to decrease the charge of FGT to a certain level, the only possibility to re-program a cell to lower charge is to perform an erase operation (i.e., remove the cell's charge completely) and consequently program a cell to a desired level. This is known as the **erase-before-rewrite** principle of Flash memory. Since the probability that during an arbitrary update all bits in a Flash page require either no change in the corresponding cell's charges or their increase is negligibly small, the overwrite is never performed without preceding erase operation. This write behavior of Flash memory is fundamentally different from one utilized by HDD, where the modified data can be written simply "on top" of the old ones, and the erase operation is not defined.

As already mentioned, **NAND Flash supports read and program operations on page basis (typically 4-16KB), while the smallest unit to erase is a Flash block (typically 128 or 256 pages, i.e., 512KB-4MB)**. However, even a brief look into the Flash memory architecture and the procedure of basic operations, can easily raise a question about those units. Why is the smallest unit of program operation the whole Flash page? Actually, NAND Flash is capable of programming just a single bit of information (e.g., apply  $V_{pgm}$  to the

corresponding wordline, and  $V_{cc}$  to all bitlines, except the one holding the cell we wish to program). Further, by reading we can sense only the interested string and omit the others. Although, in fact, this works, there are certain reasons why it is almost never utilized. One of those is that the difference between the power consumption by programming the whole page, as compared to programming just one bit or byte would be negligible (same for reading a page vs reading a single bit/byte). Other reasons are management complexity and certain reliability issues (e.g., implementation of ECC, increasing of program disturb errors). In other words, although it is possible, it is considered impracticable for common Flash storage devices. However, the proposed NoFTL architecture allows to revisit both properties (minimal program unit and erase-before-overwrite principle of Flash), and efficiently utilize them to speedup the system performance and to prolong the lifetime of Flash storage (see Chapter 7).

One important property of NAND Flash memory is the **latency asymmetry** of three basic operations. Although there is a significant discrepancy in latencies between different manufactures, the rough estimations for three Flash types are presented in Table 2.5. Thus, the rough average ratio between read, program and erase operations is 1:15:70. The latency asymmetry is the characteristic property not only for Flash memory, but also for the majority of Non-Volatile Memories.

	SLC	MLC	TLC
<b>Bits per Cell</b>	1	2	3
<b>P/E Cycles</b>	100000	3000	1000
<b>Read Time</b>	25us	50us	~75us
<b>Program Time</b>	200-300us	600-900us	~900-1350us
<b>Erase Time</b>	1.5-2ms	3ms	~4.5ms

Figure 2.5.: Latencies of basic operations for different NAND types. Source: AnandTech [98]

Another characteristic of Flash memory is that **all operations require a constant time to locate the unit to be accessed (page, block)**, which is opposite to variable seek times present in HDDs. Because of this property Flash memory is seen as storage with IO latencies being constant, and thus independent from the location of data, making it especially beneficial compared to HDD in workloads with random access patterns. While generally this is true, there are few exceptions and remarks regarding this statement. For MLC and TLC Flash types the latency of a program operation may vary depending on the offset of a page within each block [27], [111]. For instance, on MLC Flash roughly half

---

of pages in every block are the so-called fast pages (LSB-pages), while another half are the slow pages (MSB-pages). The programming latency for slow pages can be up to 6 times higher than for fast pages [27]. More details on this are provided in Chapters 7 and 6.1, where we utilize this behavior for improving the write performance of the database system. Similar effects are often observed also by the erase operation.

One might also doubt the statement of *location independent access times on NAND Flash* by looking at the specifications of SSDs, where the manufacturers clearly distinguish between the *Random Read* and the *Sequential Read* latencies. The difference between two access patterns can be multiple orders of magnitude (e.g., 25 $\mu$ s for random read, and 25ns for sequential read), which seems to be even more significant than for HDDs. However, the meaning of adjectives *random* and *sequential* is different for Flash memory as for spinning drives. While for HDDs they characterize the physical contiguity of data chunks to be accessed, for Flash memory they rather describe the knowledge about chunks to access in advance. This knowledge can be utilized by the Flash controller to: (i) perform read operations in parallel on independent units of Flash memory; (ii) utilize the interleaving techniques; and (iii) optimize usage of cache registers and on-device buffers [16]. This leads to decreasing the overall latency for reading a group of pages, and thus lowers the average latency for a single Flash page from this group. In other words, reading 100 Flash pages with contiguous physical addresses, would take about the same time as reading 100 pages with random physical addresses, if in both cases the controller knows addresses of all requested pages in advance. The latter case might perform even faster, because pages randomly distributed over the Flash memory are more likely to be read in parallel, than those with contiguous addresses. Thus, the adjectives "single" and "bulk" might be more descriptive for Flash memory than "random" and "sequential". Note, that the utilization of optimized read operations is in praxis very complex, due to the legacy software protocols and black-box design of modern Flash storage. We elaborate more on this in the following chapters.

Yet, another important property of Flash memory is its **limited lifetime**, which is typically expressed in the maximal number of program/erase (P/E) cycles every Flash block can undergo until the pages in it cannot store data reliably anymore. The reason behind this limitation lies in the degradation of insulating layer (tunneling oxid) between the floating gate and the channel of FGT, caused by high voltages and current flow through this layer during program and erase operations. The damaged insulating layer can itself trap and accumulate electrons, which gives it a certain electrical field. It, in turn, interferes with the field created to program or erase a cell, and after a certain point the interference does not allow to perform these operations correctly [24, 14]. Thus, the wear of Flash memory is a continuous process, while the limit of P/E cycles is based on aggregated statistics and provides a rough estimation of a block's lifespan. Different Flash types (see

---

below) differ also significantly in their longevity. While there is a difference also between manufacturers, the rough numbers as of today are: SLC Flash - 100.000, MLC Flash - 3.000 and TLC Flash - 1.000 P/E cycles (2.5).

Apart from the program and erase fails caused by wear of blocks, there are three other types of errors that might occur on Flash memory: retention, program interference and read disturb errors. Retention errors appear when the small amount of charge stored in floating gate leaks through the insulating layer. While retention causes the decrease of a cell's charge with time, the program interference and read disturb errors, conversely, result in an unintentional increase of a cell's charge during program and read operations. According to analysis provided in [14], retention errors dominate by frequency, followed by program interference errors, and then read disturb errors. However, common to all types of errors is the correlation of their frequency with the Flash wear - the more P/E cycles a certain block has undergone, the more vulnerable are its cells to those errors. The detailed study of this topic can be found in [13] and [14].

### Types of NAND Flash

Flash memory underwent over the last three decades significant technological changes. The most significant force in its evolution was the demand on reducing the price of memory, which could be achieved mainly by increasing its density. In 2016 the technology was able to store as much as one terabyte of data in a single SD card with the size of a cent coin<sup>16</sup>, while 10 years back the maximum limit of the same card was 4GB<sup>17</sup>, resulting therefore in about 25x capacity increase per year for this product over a decade. This technological progress was achieved by following two different directions in the development of NAND Flash.

First, as common in the whole semiconductor industry, increase of memory density is a result of the so-called "process shrink" - decreasing the size of transistors via better fabrication process or materials, which roughly follows the Moore's law<sup>18</sup>. Today Flash memory chips are primarily fabricated using 1Xnm nodes (the distance between memory cells is in 10 to 19 nanometers range), which back in 2008 seemed technologically impossible. While going beyond this range seems to be technically unreasonable (at least,

<sup>16</sup><https://www.sandisk.com/about/media-center/press-releases/2016/western-digital-demonstrates-prototype-of-the-worlds-first-1terabyte-SDXC-card>

<sup>17</sup><https://www.sandisk.de/about/media-center/press-releases/2006/2006-09-26-sandisk-introduces-the-first-4-gigabyte-sandisk-ultra-ii-sdhc-card%E2%80%94fast-performance-large-capacity-ideal-for-digital-cameras-and-camcorders>

<sup>18</sup>"The number of transistors incorporated in a chip will approximately double every 24 months.", Gordon Moore [69]

as of today), the new 3D memory organization allows to continue the growth of memory density by improving wafer lithography also in the future.

The second NAND Flash development resulting in an increase of density is known as "bit growth". As we already know, the information is stored on Flash by means of electrical charge captured in memory cells. So, the bit growth is about how much information can be encoded in a single cell, and today's market offers four different types of NAND Flash based on these criteria (Figure 2.6). Originally, all Flash memory was **Single-Level Cell Flash (SLC)**, where every cell can encode just a single bit of information. Thus, logical bit 1 is associated with the cell having a charge below certain threshold level (ideally no charge, but due to read disturb and program interference errors those cells might have some parasitic charge), while the one with the charge above a threshold level would indicate the logical 0. In **Multi-Level Flash (MLC)** the controller differentiates between four different thresholds of the cell's charge, thus being able to encode two bits of information in every cell. **Tripple-Level Cell Flash (TLC)** can store three bits of data in a cell, which assumes eight different threshold voltages. The most dense type of Flash memory is Quad-Level Flash (QLC), which was first introduced in 2018 ([91]), and as its name infers, it allows to encode four bits of information in every floating gate transistor. Thus, MLC Flash offers twice as much capacity as SLC Flash, and TLC increases this again by 1.5x, while QLC offers four times more capacity as SLC in the same number of Flash cells.

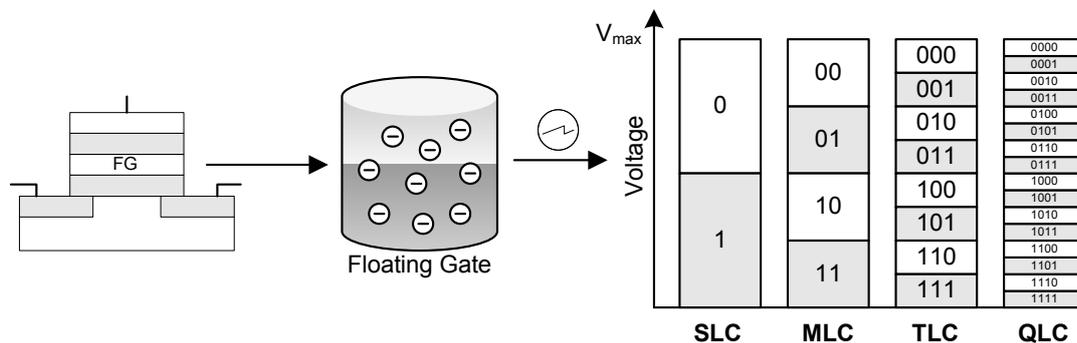


Figure 2.6.: Four different types of NAND Flash memory.

Despite the clear dominance of TLC and QLC Flash in terms of density, all four types of memory are manufactured today. This is because the increase of memory density through bit growth process is not a free beer. The price paid is the decrease in performance and longevity. In general, there is no structural difference between the memory cells of the three Flash types, rather they differ in how those cells are read and programmed. While for SLC Flash the controller must differentiate only between two voltage thresholds, for

---

MLC there are already four different thresholds, for TLC - eight, and for QLC - 16. In other words, the program and read operations on MLC, TLC and QLC Flash are becoming more and more "accurate" and "sensitive" as compared to SLC Flash (e.g., by programming more ISPP cycles performed, which increase the cell's charge each time only by a small delta).

On the other hand, more voltage thresholds means that the distance between two distinct values becomes smaller, and thus even small parasitic injections of electrons (read disturb or program interference errors), or small leakage of them (retention errors) can lead to erroneous interpretation of stored bit values. So, it is not that on MLC and TLC Flash these errors occur more frequently, but it is that the impact of them for these NAND types is larger as compared to SLC Flash. And since the frequency of errors is correlated with the wear of memory, TLC Flash will "give up" earlier than MLC Flash, and MLC Flash again earlier than SLC Flash (compare the P/E cycles in Figure 2.5).

Therefore, the bit growth development created three Flash types trading off three characteristics of memory - density on one axis, and performance with longevity on the other. While each memory type has found its consumer, MLC Flash is leading the storage market today, but TLC Flash might catch up soon thanks to 3D NAND technology.

The latest breakthrough in Flash technology is the **3D Flash**. Along with the process shrink and bit growth, it might be seen as the third direction of NAND development. The basic idea behind 3D Flash is to add a third dimension to the Flash memory organization (Z-dimension), which is done by placing series of memory cells (strings) vertically on the silicon wafer. While in traditional Flash (2D Flash) a string of cells "lies" on the horizontal plane of the die, and thus occupies space proportional to the number of cells in it; in 3D Flash this string is folded in the middle and then put up high occupying therefore minimal space on the horizontal plane of the die. This fabrication process allows the manufacturers to increase memory density without the process shrink (and even go back to 2Xnm or 3Xnm space between Flash cells), i.e., memory is growing in Z-dimension, instead of shrinking in X-Y space. Apart from the clear density benefit, 3D Flash offers significant advantages in terms of reliability and performance. New architecture and materials<sup>19</sup> enable the use of TLC Flash with performance and endurance characteristics close to MLC Flash. A more detailed description of 3D Flash technology can be found in [2].

---

<sup>19</sup>3D Flash often uses cells based on a charge trapping (CT) technology. CT transistors have a special charge trapping layer (silicon nitride film), instead of a floating gate (polycrystalline silicon) used in FGT.

---

### 2.3.2. Flash SSD

The market nowadays offers multiple classes of storage devices based on NAND Flash memory. Those include (i) removable/portable devices, like eMMC Flash<sup>20</sup> cards (e.g., SDHC, miniSD) and USB Flash drives; (ii) embedded storage, e.g., Flash memory chips in smartphones, routers, etc; (iii) Flash Solid State Disks (Flash SSDs, or just SSDs hereafter) - mass storage devices for desktop PCs, laptops and servers; and (iv) NVDIMMs<sup>21</sup> - a kind of non-volatile RAM made via combination of Flash SSD and traditional DRAM in one module. Each class, in turn, is represented by a variety of devices differing by form factor, physical interface, software protocol, hardware architecture and firmware. As this work is dedicated to the optimal use of Flash storage for DBMSs on servers, we are interested only in SSDs. In this chapter we briefly describe the architecture, working principles, characteristics and diversity of modern SSDs.

SSD is a storage device, which uses NAND Flash memory as a persistent medium, but hides internally most of its native behavioral characteristics to provide to the outside the standard block device access interface. In other words, an SSD creates an abstraction level over the Flash memory emulating the behavior of a traditional HDD. Since this abstraction is completely transparent (hidden) to the device users (OS, FS, DBMS, etc.), SSDs are often referred to as black-box devices. The software layer (firmware) responsible for the abstraction is called Flash Translation Layer (FTL), and it is running either on the on-device controller (most of the products), or on the host system as a device driver (very few products). The SSD's controller (Figure 2.7) might vary from the simplest single-core controller (e.g., ARM), to the multi-core controllers combined with FPGA or GPU units. To execute the code and cache frequently accessed metadata SSDs commonly have an SRAM module of small capacity (e.g., 128KB), while for caching other FTL metadata and buffering request data (read and write cache) the controller uses on-device DRAM module (e.g., 1GB in enterprise SSDs).

A common SSD nowadays has multiple memory *chips* (Figure 2.7), where each chip has either two or four NAND *dies* (dual- or quad-die NAND chips). Each die, in turn, has two or four separate *planes* of blocks, with one *page buffer* and one *cache buffer* registers per plane. The fixed-size blocks contain typically 64, 128 or 256 Flash pages. A page is logically divided into main area (e.g., 2-8KB), which is available to the host for storing user data, and a small-sized (e.g., 64-256 bytes) out-of-band area (OOB), which is used only internally by FTL algorithms. Groups of chips are connected via a shared I/O bus (channel) with the SSD controller.

---

<sup>20</sup>Embedded Multi-Media Controller (eMMC) Flash - flash storage, where the NAND Flash memory and the controller are integrated on the same silicon wafer.

<sup>21</sup>NVDIMM - non-volatile dual in-line memory module

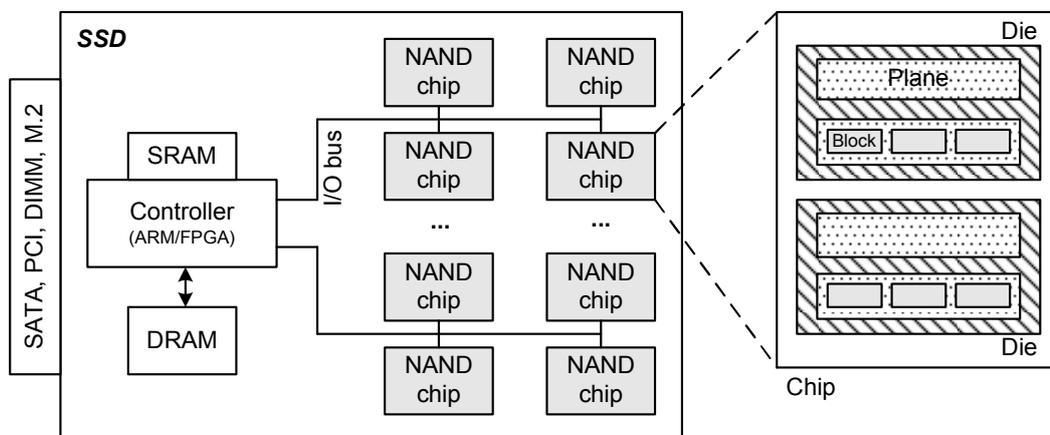


Figure 2.7.: Simplified architecture of SSD.

This architecture supports multiple levels of I/O parallelism. Thus, different operations can be executed simultaneously on chips belonging to different channels. Further, each die can execute I/O operations independently, but must coordinate data transfer through the shared I/O bus with other dies in its channel-group (interleaving of operations). The smallest unit of parallelism is the plane. The operations can be executed on all planes of the die simultaneously, but those operations must be of the same type (e.g., perform read operation on all four planes of a certain die simultaneously), and the addresses of accessed pages must conform to the requirements of multi-plane operations (e.g., have identical offsets within each plane). Altogether, it gives the SSD many possibilities to parallelize execution of incoming I/O requests. For instance, if the SSD consists of 16 quad-die chips (64 dies in total), where each die has two planes, then the controller can theoretically execute as many as 128 operations simultaneously. In practice, however, the available I/O parallelism of SSDs is typically underutilized. The main reasons for this and the proposed solutions will be discussed in Chapters 5, 6.

Today's storage market offers SSDs in a variety of form factors, with almost dozens of possible bus interfaces (connectors), supporting multiple data transport standards and protocols. We summarize the most popular available options in Tables 2.2, 2.3. Because the topic about alternatives of physical and logical interfaces used by SSD manufacturers is rather orthogonal to the current research, as well as due to the complexity and volume a comprehensive analysis would require, we omit the detailed description of those options.

<b>Bus interface (Connector)</b>	SATA	mSATA	SAS	PCIe: x1, x2, x4, x8, x16, x20	SATAe	
<b>Form factor</b>	2.5" 3.5"	PCB	2.5" 3.5"	PCIe cards	2.5" PCB	
<b>Transport*</b>	SATA 3G SATA 6G		SCSI	PCIe: 1.0, 2.0, 3.0	SATA 6G	PCIe
<b>Protocol Driver</b>	AHCI		SCSI	AHCI NVMe	AHCI	AHCI NVMe

Table 2.2.: Variety of SSDs on the storage market (Part 1).

## FTL

Performance characteristics of SSDs are basically defined by three major aspects - the hardware architecture (e.g., Flash memory, controller, physical interface), the FTL and the communication protocol (e.g., ATA, SCSI, NVMe). The role of FTL in the overall SSD architecture is probably the most important, since it defines to which extent the performance potential of the underlying Flash memory can be utilized. Consider the fact, that there are only few (about 5) manufacturers of Flash memory chips, a dozen alternatives of possible hardware interfaces and protocols, but there are several hundreds manufacturers of SSDs. Thus, the software layer - the FTL - is the key aspect, which allows them to coexist and compete with each other. This is also the reason why SSD manufacturers never publish the details of their FTL and keep those as top secret information. Both aspects - the importance of the FTL and the lack of information from industry - have forced academia to put a lot of effort into the research and development of FTL in the past two decades, and even today this topic is actual and actively researched.

The research community proposed dozens of different variants of FTL (often called *FTL schemes*). It is important to mention that neither of the proposed FTL schemes can be seen as being ultimately better than the others, and even the pair-wise comparison could not clearly define the generally better FTL scheme out of two. There are several reasons for this. First, the FTL scheme is characterized by multiple criteria, such as performance numbers and predicted longevity of device, memory and storage footprints, computational complexity and reliability. Those are usually mutually dependent, which means that optimizing on one parameter (e.g., better longevity guarantees) leads to degradation

<b>Bus interface (Connector)</b>	m.2		u.2			DDR DIMM	FC
<b>Form factor</b>	PCB		2.5" 3.5"			DIMM	2.5" 3.5"
<b>Transport*</b>	SATA 6G	PCIe	SATA	SAS	PCIe	DDR3	FC
<b>Protocol Driver</b>	AHCI	AHCI NVMe	AHCI	SCSI	AHCI NVMe	AHCI	FCP-SCSI FC-NVMe

\* Maximal bandwidth of transport protocols:  
 SATA 3G = 3Gb/s  
 SATA 6G = 6Gb/s  
 PCIe 1.0 = 250 MB/s per lane per direction  
 PCIe 2.0 = 500 MB/s per lane per direction  
 PCIe 3.0 = 1 GB/s per lane per direction  
 SAS-3 = 12Gb/s

Table 2.3.: Variety of SSDs on the storage market (Part 2).

of another (e.g., decreased performance and higher complexity). Since, customers have different priorities of requirements for the storage, a certain FTL scheme might be the best choice for one system, while being less useful for another. The second reason is that the performance and longevity numbers of a certain FTL are highly workload-dependent. The read/write ratio of I/O requests, their frequency, size, locality and data skew are the typical workload parameters, the interplay of which makes one (or several) FTL scheme(s) preferable.

An FTL scheme consists of multiple algorithms (Figure 2.8), each of which covers a certain property of Flash memory. Address translation and garbage collection (GC) are dealing with the erase-before-overwrite property. Wear-leveling (WL) and bad block management (BBM) address the wear property of Flash memory. Error detection and correction techniques (ECC) are needed due to different types of errors present on Flash memory. Queuing and scheduling of requests are required to utilize latency asymmetry and available degree of parallelism. Below we provide a brief description of those FTL algorithms and their main variants.

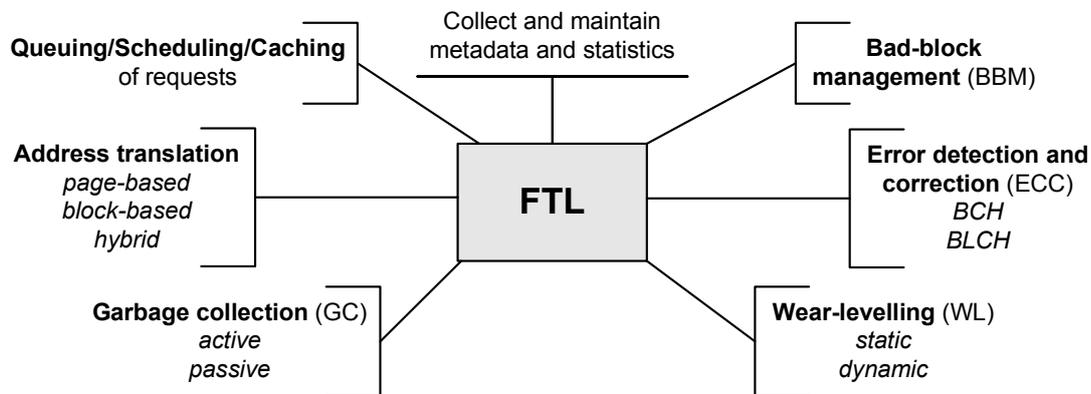


Figure 2.8.: Main tasks of FTL.

## Address Translation and Garbage Collection

One of the main challenges the FTL designers are facing is how to mask the erase-before-overwrite property of Flash memory, so that the SSD can be accessed using the traditional block device interface and behave as an HDD. The naive approach would be: each time when the SSD gets a write request that modifies previously written data, FTL, at first, stealthily executes an erase operation of the corresponding Flash block (i.e., the block where the old data resides), and then writes the modified data in the original place. This would perfectly work if the data is always updated in multiples of whole Flash blocks. For instance, if the block consists of 128 4KB Flash pages (512KB in total), this would mean, that each update overwrites all 128 pages. While there are some application scenarios with such update pattern (e.g., circular DBMS logs written in chunks equal to Flash block), in general however, applications tend to update much smaller chunks of data. Common units of read and write I/Os are an OS page (e.g., 4KB), a DBMS page (e.g., 4KB-32KB) or just a single disk sector (512B). Consider, for instance, the case where just one Flash page needs to be updated. Since the corresponding Flash block includes also many other pages (e.g., 128), the erase operation would lead to loss of their content. Thus, before erasing a block, FTL must read and temporary store (e.g., in on-device DRAM) all block's pages except the modified one, then perform an erase operation, and afterwards write back those pages from the temporal store, as well as the modified page. Thereby, for a single page update FTL would need to perform 127 read, one erase and 128 write operations, which would obviously introduce a huge overhead, and make even the fastest Flash memory being slower than the slowest HDD today in terms of write request latency.

---

Apart from the latency overhead, the naive approach might easily lead to a significant issue with the Flash wear. Typically every workload has a certain locality of requests, which means that some portions of data are accessed and modified more frequently than the others. Thus, often to describe the access patterns of database workloads (especially OLTP workloads) analysts use the Pareto's 80/20 rule, meaning that 80% of all user requests touch only 20% of data. With such a workload under the naive approach, Flash blocks containing pages with hot data would be erased frequently, while the blocks with outdated or static data (cold blocks) might undergo only few erases over the same period. This skew in erase counts would result in an uneven wear out of Flash memory, and soon its parts with hot blocks might become invalid (due to burn-out), leading to premature damage of the whole SSD. Thus, both issues make the naive approach completely unsuitable for SSDs.

The common solution applied in all modern SSDs is to implement a variant of an **out-of-place update strategy**. Its basic idea is to decouple the way the host system (OS, FS, DBMS, etc.) organizes the data (logical data placement), and how the SSD does this (physical placement of data). That is done via an address indirection level, which is typically realized as an **address translation table**. It stores for every logical address (at certain granularity) the corresponding physical address pointing to the place where the data actually resides on Flash memory. When the host submits a write request to the SSD (e.g., write 4K page at address 100), the FTL decides on its own (see later how) where to place the data on Flash memory (e.g., write data to physical page 478), and consequently updates the corresponding mapping information (e.g., 100 -> 478). If the host modifies the same data later on, it would be again written to a new location on Flash (e.g., 100 -> 1987). In other words, the data is written/updated out of its original place (out-of-place). To read a data with certain logical address, FTL consults its translation table to get the physical address of where the data is currently placed.

This address indirection has two main purposes. First, the FTL designers gain complete freedom regarding where and how to place the data on Flash memory, which, in turn, gives them a possibility to mitigate the performance and wear issues resulting from the erase-before-overwrite principle. Second, since address translation is completely invisible for the system outside the SSD, the host loses the control over the physical data placement, and thus it is also freed from the responsibility to deal with the constraints of Flash memory, like erase-before-overwrite principle and memory wear. Thus, address indirection is the key strategy of FTL to mask all native behavioral characteristics of Flash memory and make the SSD behave as a traditional HDD.

By placing the data at every update into a new location on Flash memory, the previous version of the data in its original place becomes outdated, and thus invalid. To delete this data, FTL must perform an erase on the corresponding Flash block. However, for

---

performance reasons FTL typically tries to postpone those erase operations, and thus the invalidated versions of data items are kept stored on Flash memory for a certain time along with their up-to-date versions. In order to avoid a situation of running out of space because of keeping lots of outdated data, FTL periodically runs the process called **garbage collection (GC)**, which selects the blocks containing invalidated data and erases them, while shifting the valid data in those blocks to new locations. Finding victim blocks, performing multiple read/write operations to copy valid pages to new destinations and erasing the blocks requires significant computational, memory and especially I/O overhead, which makes GC the most resource and time consuming process of the FTL. Since every die can perform only one operation at a time (e.g., one multi-plane read operation), the FTL must efficiently coordinate the usage of Flash memory by executing on every available die either a GC operation or a read/write operation submitted by the host. When GC is holding an exclusive control over a certain part of Flash memory, the incoming user requests to those dies are postponed. The interleaving of GC with processing of user requests results in two significant issues all SSD manufacturers are faced with. First, this introduces an additional delay to the user requests, and thus slows down the I/O throughput of the device. For instance, although the latency of a program page operation on Flash memory is typically in a range of 250 $\mu$ s to 750 $\mu$ s, the single write request can take as much as 80ms (Figure 2.9) and more (up to 680ms were measured in [16]), when it is interfered with GC. Second, the amount of work performed by GC varies significantly depending on several factors, like current load, access patterns, state of SSD, etc. Thus, the interference with the user requests is also variable, which often results in significant fluctuations in the I/O throughput provided by SSD.

The granularity of address translation table, the algorithms responsible for finding a new location for modified data, as well as GC algorithms defining which blocks to select for erasing, and where to place the valid data in them are the most significant and fundamental characteristics of all FTL schemes. There are basically three different groups of FTL schemes: page-level, block-level and hybrid FTL schemes. They are named based on the granularity of the address translation table, because this parameter influences also the other algorithms of the FTL.

### **Page-Level FTL Scheme**

In a page-level FTL scheme the address translation table has an entry for every Flash page of the underlying memory. The simplest implementation of such table would be a one-dimensional array with the size equal to the number of Flash pages in the SSD. The offsets in the array would indicate the immutable logical addresses of pages (logical page number - LPN), while the elements at these offsets would store the current physical addresses of those pages (PPN). Because the page is the smallest addressable unit in NAND

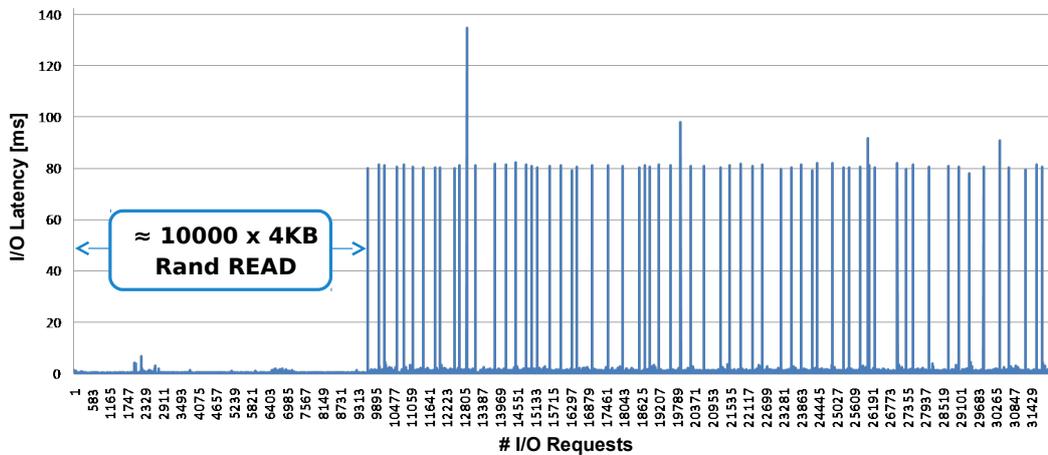


Figure 2.9.: I/O bandwidth fluctuations of an enterprise SSD. Source: Petrov et al. [77].

Flash memory, having a slot in the address translation table for every page, gives this FTL scheme the highest possible flexibility in placing the data. Consider an example in Figure 2.10 assuming a simple implementation of page-level FTL scheme. Here, two consequent write requests are modifying pages with LPNs 578 and 579. Each of those pages can be basically written into an arbitrary free location on the SSD. However, typically, the FTL keeps track of only one *current* block on every die, and appends the incoming pages into it. Once this block gets full, the FTL selects another block and continues. In our example, the first page is written to the current block 7 into the Flash page with PPN 1022. Then, FTL finds a new empty block (PBN = 1), which becomes a current one, and writes the second page into this block (PPN = 190). The mapping table is updated after every request and pages, holding the original versions of modified data, are marked as invalid (pages with PPNs 2026 and 356).

When there are only few empty blocks left on the SSD, the FTL kicks in garbage collection. GC selects a *victim* block (e.g., the one with the largest number of invalid pages), copies its valid pages to the *current* block, and consequently, erases the victim block and adds its address to the pool of free blocks. This process continues until the number of free blocks reaches a certain threshold. It is easy to see that in this FTL scheme the amount of work performed by GC is proportional to the number of valid pages in victim blocks. For instance, for a block with 5 valid pages the garbage collector will perform 5 copybacks and one erase operation, while for a block with 25 valid pages already 25 copybacks and one erase are needed. Thus, the key factor for reducing the overhead of GC and improving thereby performance characteristics of an SSD is reducing the average number of valid

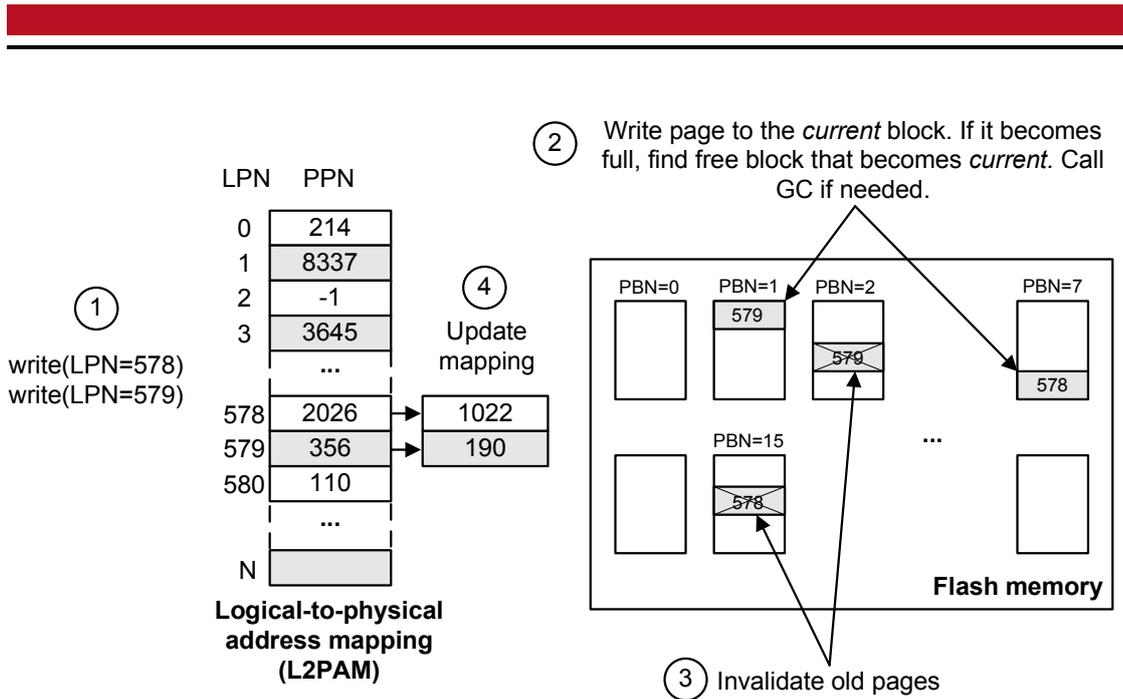


Figure 2.10.: Example of data placement in page-level FTL.

pages in victim blocks chosen by GC. The common way to achieve this is performing a proper data placement strategy, which is based on considering update frequencies of data (*temperature* of data).

Assume a simple example of storing two equally sized files on SSD, where one is update-heavy (*hot* file), while another is relatively *cold* with only rare updates. If pages of both files are evenly mixed over the physical address space, then the average victim block selected by GC would have half of its pages being hot, and another half being cold pages. With high probability the majority of hot pages would be invalid, since the high update frequency of the hot file would probably result in newer versions of those pages existing somewhere else on the Flash memory. In contrast, the cold pages would be rarely invalidated by updates, and thus would be subject to copyback operations performed by GC on this block. Copying half of the block's pages on average in every victim block would result in a significant overhead of GC and low performance of SSD. Consider now the data placement strategy which holds hot pages physically separated from the cold ones. In that case, GC would in most cases select victim blocks from those with hot pages, since these would have much smaller number of valid pages as compared to blocks holding cold pages. With high probability GC would need to perform only few copybacks on average. This would result in lean GC with low I/O overhead, and consequently in a high performance

---

---

SSD.

This technique to reduce the GC overhead is known as *hot-cold data separation*, and it was intensively researched in recent years (e.g., [89], [106]). The page-level mapping scheme is the most suitable for this strategy. The full associativity between logical and physical page addresses, gives this FTL the highest possible flexibility during the data placement process, needed for efficient hot-cold data separation. However, this technique might easily become a double-edge sword - while improving the performance characteristics of the SSD on the one side, it can be disadvantageous for the longevity of the device. In the previous example, the hot-cold data separation would lead to a situation in which mainly the Flash blocks containing hot pages would be erased by GC, while blocks with cold pages would experience only few P/E cycles. As a result, one part of Flash memory would wear-out much faster than the other. To avoid this situation the FTL applies different wear-leveling approaches.

While the flexibility of page-level FTL allows to achieve optimal SSD performance under various workloads, it has its price - the memory requirements. Due to the fine-grained mapping entries, the size of the address translation table becomes relatively large. For instance, for an SSD with 1TB storage capacity the complete mapping table would require about 1GB of memory (assuming 4KB Flash pages). Since, the on-device DRAM is typically limited (e.g., 1TB enterprise SSD with 512MB on-device DRAM), and its major part is dedicated to read/write caches, holding the whole mapping table in the SSD's DRAM is usually impossible (or at best impractical<sup>22</sup>). Several approaches have been proposed [28], [62], to overcome the issue with limited amount of available memory. The basic idea behind them is to cache only a small part of the address translation table in DRAM, and then load missed mapping entries on demand from Flash (where the whole table is stored), replacing thereby some other entries in cache. However, those approaches introduce additional I/O and computational overheads. This makes them suitable only for workloads with certain properties (see more in Chapters 5 and 6).

### **Block-Level FTL Scheme**

As its name reveals, the block-level FTL scheme maintains an address translation table at a granularity of a single Flash block. Because every block consists of multiple pages (e.g., 128-512), the size of total mapping information is small. Assuming a Flash block with 128 pages (4KB each), only 16MB of memory is needed to store the complete address translation table for a 1TB SSD. Such a small table can be easily cached completely in an on-device DRAM (which makes this scheme especially suitable for portable or embedded

---

<sup>22</sup>Enlarging the capacity of SSD's DRAM increases the costs of the device, and introduces some reliability issues.

Flash storage devices). However, while solving the problem on the one side, it introduces a new one on the other side.

The block-grained translation table allows mapping logical block addresses to their current physical locations. However, how to find a physical location of a single Flash page? This is done by keeping the pages within a block being logically and physically contiguous, which allows transforming logical page address to their physical addresses via a simple mathematical formula. Consider an example in Figure 2.11, where we assume a Flash block containing 128 pages. To find a physical location of a page with LPN 1234, at first the FTL derives (i) the logical block number (LBN) - by dividing LPN by the number of pages in a block ( $LBN = 1234/128 = 9$ ); and (ii) the offset of the page within the block - by replacing the division operation with the modulo ( $OFFSET = 1234\%128 = 82$ ). Then, it consults the address translation table to get the current physical block address (LBN-9 -> PBN-389), and finally uses the same offset within a block (82) to get the physical page address ( $PPN = 389*128 + 82$ ).

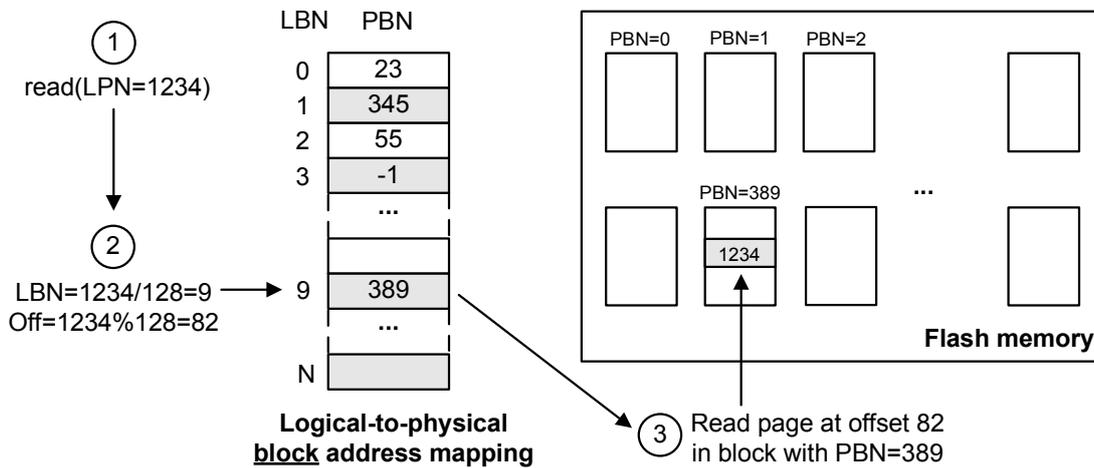


Figure 2.11.: Example of read operation in block-level FTL.

While this address translation introduces no overhead for read operations, it becomes really problematic when it comes to writes. Assume now the SSD gets a write request to update the logical page with LPN 1234 (see Figure 2.12). Since update is performed in an out-of-place manner, the FTL selects a new empty block (using a WL algorithm), where the modified version of the page would be stored (e.g., PBN = 2). However, the page can not be simply placed at the beginning of the block, rather it must be written at exactly the same relative location within the block as the original page to keep the page offset constant

(e.g., at offset  $1234\%128=82$ ). Moreover, all remaining (unchanged) pages of the original block (PBN-389) must be copied to a new block as well, because the mapping of logical block would be updated (LBN-9  $\rightarrow$  PBN-2), and the whole original block (PBN-389) would be invalidated. Finally, the original block is erased and returned to a pool of free blocks. Thus, a simple page update requires in our example 127 copyback operations, 1 write operation and 1 erase. Basically, the performance of block-level mapping is identical to the naive approach, which we have described at the beginning of this chapter. The only difference between both is that in block-level FTL the address translation table allows to use an arbitrary free block for placing modified data, which together with the proper wear-leveling strategy solves the wear issue present in the naive approach. Note however, that although, in the general case, the block-level FTL has very high I/O overhead, under certain conditions it can become the best FTL scheme. For instance, for read-only data, for log-based data structures, or for data which is updated in chunks equal or bigger than the size of a Flash block (e.g., separate tables storing large BLOB fields, like images, audio, video, etc.) - the block-level FTL would introduce no overhead at all, would perform same as page-level FTL (or even slightly better), but would have minimal memory footprint and lean implementation.

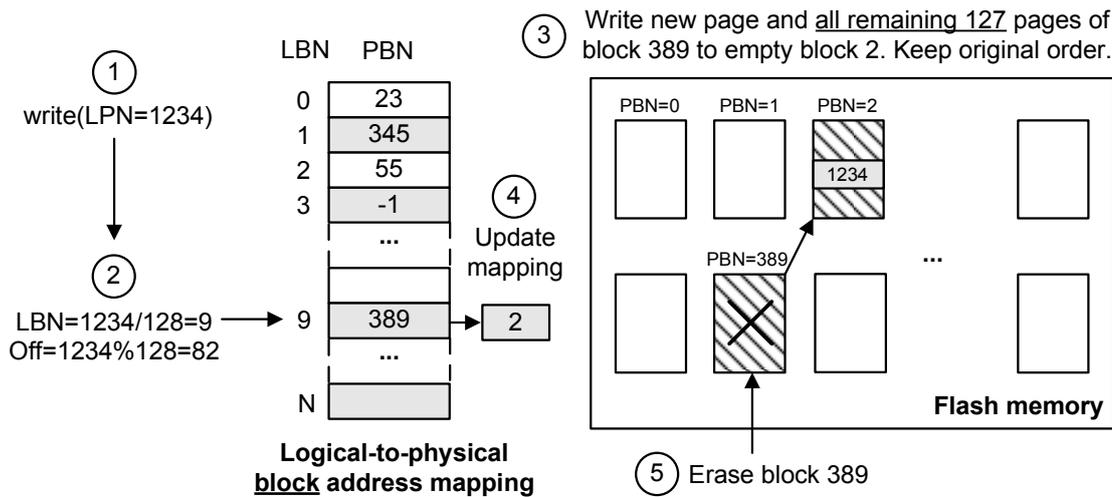


Figure 2.12.: Example of write operation in block-level FTL.

Summarizing the above, the page-level FTL is generally the most efficient FTL scheme, while simultaneously the most memory consuming; the block-level FTL is, in contrast, the

one with the highest I/O overhead, but it requires the least amount of memory. Practically, the disadvantages of each make them being almost unsuitable for general-purpose SSDs. The solution for SSD manufacturers is typically applying a kind of *hybrid FTL scheme*.

### Hybrid FTL Scheme

Although there are many variants of hybrid FTL scheme [48], [52], [56], all of them share some common characteristics. Those schemes virtually divide the physical address space into two segments (Figure 2.13). The larger segment (e.g., typically 85% or more of available storage space) is called *data area*, while the smaller segment (e.g., 15% or less) is referred to as *log area*. Although both areas consist of a constant number of blocks, they are dynamic in their composition, e.g., a block belonging currently to the data area, might be used after its erasure by the log area. For each segment FTL maintains a separate address translation table. Thus, blocks belonging to the data area are referenced through the block-level translation table, while the log area uses a fine-grained page-level mapping table (thus *hybrid FTL*). Since only a small amount of physical space utilizes page-level mapping, the total size occupied by both address translation tables is kept relatively small, and thus can be completely cached in an on-device DRAM. The log area represents an *over-provisioning* of the device, i.e., it is not available to the host system and used only internally by FTL. The host can utilize only the capacity of the data area.

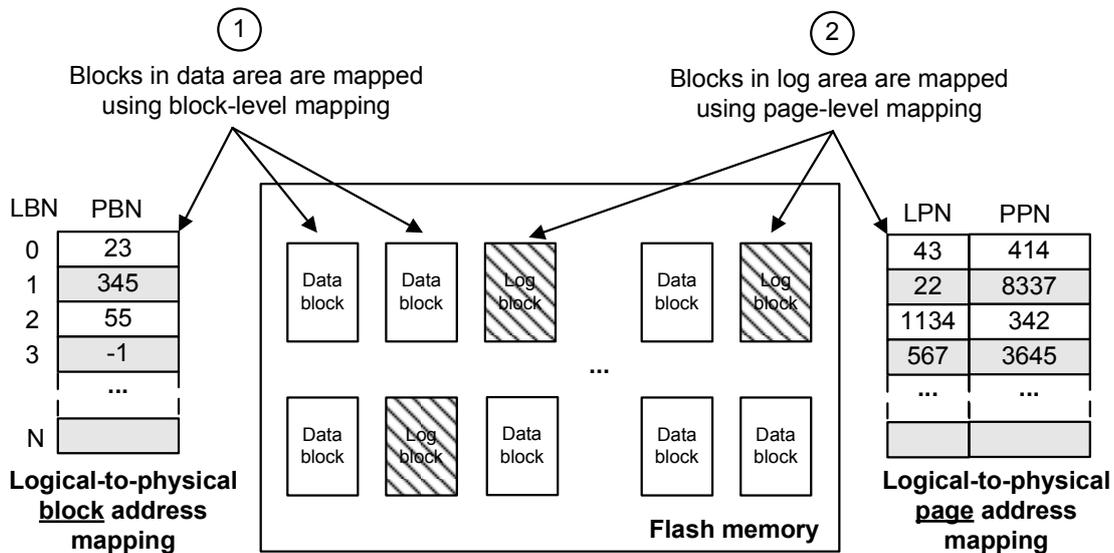


Figure 2.13.: Address translation in hybrid FTL.

The new data is written to empty blocks of the data area according to the rules of block-level mapping (i.e., within each block pages are logically and physically contiguous). Conversely, the write requests, which update previously written data, are firstly absorbed by the log area. How the modified data is written to log area depends on the concrete implementation of hybrid FTL. For instance, in FAST [52] and FASTer [56] FTL schemes the pages of update requests are simply appended to the *current* block in the log area - as it would be in a pure page-level FTL scheme. At a certain point in time, the modified data stored in the log area is *merged* with the unmodified data from the data area. When the log area gets almost full, the GC selects a victim block from this area and merges it with the corresponding blocks in the data area. Assume that GC has chosen the victim block with PBN 478 from the log area. This block contains 90 valid pages (e.g., LPN: 466, 788) and 38 invalid pages (the newer versions of which are stored in other log blocks). The invalid pages are simply ignored, but every valid page must be merged with the corresponding block in the data area. The merge operation is identical to a page update procedure in block-level FTL. Thus, the merge procedure for page with LPN-466 might look like this (Figure 2.14):

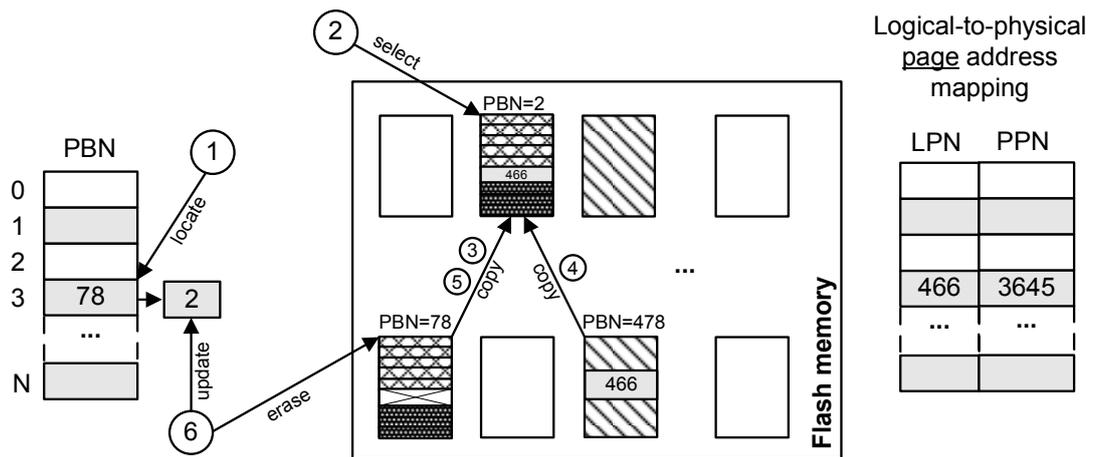


Figure 2.14.: Example of merge operation in hybrid FTL.

1. GC locates a data block, which holds the original version of the page (LBN=466/128=3, PBN=78);
2. A new empty block is taken from the pool of free blocks (e.g., PBN=2);

- 
3. All pages with relative offsets from 0 to 81 (LPNs: 384-465) are copied from original data block (PBN=78) to new data block (PBN=2);
  4. The modified page (LPN=466) is copied from log block (PBN=478) to new data block (PBN=2) at offset 82 (Offset=466  $\text{mod}$  128=82);
  5. Remaining pages with relative offsets 83-127 (LPNs: 467-511) are copied from original data block (PBN=78) to new data block (PBN=2);
  6. Finally, GC erases the original data block, returns its address to a pool of free blocks, and updates the block-level address translation table.

This procedure is then performed for the remaining 89 valid pages in the selected victim block, which is at the end also erased and returned to the pool of free blocks.

The log area acts, therefore, as a persistent buffer for incoming updates. The use of a page-level mapping table for this segment allows to postpone the expensive merge operations for a certain time (e.g., until the log area gets full), and thus execute current write requests in this interval without delays. In other words, until garbage collection starts, the hybrid FTL performs identically to pure page-level FTL scheme. However, when the "buffer" cannot absorb further update requests any more, GC must evict modified data from it to free the space using merge operations. This is where the hybrid FTL starts behaving similarly to block-level FTL with significant overhead and performance issues. However, in most cases the performance of hybrid FTL is a way better than that of block-level FTL (as well as typically much worse than that of page-level FTL). The reason is that the buffering of update requests in the log area is not a simple deferring of the erase and copyback operations, which would be needed in pure block-level FTL. Keeping data in log area for a certain time (especially hot data) allows to save certain amount of those I/O operations. Assume, for instance, that some hot page was kept in log area for 10 minutes. If in this period it was updated 25 times, than at the time of merging it with the corresponding data block only its latest (25th) version is considered. Thus, alone for this page the GC in pure block-level FTL would need to perform approximately 24 times more erases and copybacks than the hybrid FTL needs.

Some additional details about page-level, block-level, DFTL and FASTER FTL schemes, as well as the comparison of SSDs with those FTL schemes with the proposed NoFTL approach will be covered in Chapters 5 and 6. The comprehensive overview of those and other popular FTL schemes can be found in [61].

---

## Wear-Leveling and ECC

In order to hide the wear-out issue of Flash memory from the host system, the FTL uses three main approaches: wear-leveling (WL), bad-block management (BBM) and error-correction codes (ECC).

### Wear-Leveling

As already mentioned, every particular Flash block wears out individually, depending on the number of P/E cycles it is undergoing. The static mapping of logical to physical addresses would create a situation where some parts of Flash memory (storing frequently updated data) would wear out fast and soon become invalid, while other regions would experience only slow (with cold data) or even no (with static data) wear out. As a result, the whole SSD becomes inoperable. Thus, the task of WL is to make the SSD's Flash memory wear out evenly over the whole physical address space. The perfect WL would guarantee at every moment in time that all available Flash blocks have roughly equal number of P/E cycles performed on them. However, the practical value and efficiency of a particular WL algorithm is defined not only on the longevity guarantees, but also on its overhead and influence on the SSD's performance. Various WL algorithms introduce some computational, memory and especially I/O overheads. Additional copyback and erase operations, needed to keep block erasures evenly distributed over the whole Flash memory, are performed either by GC or by a separate WL process.

Different WL strategies can be grouped into two major categories - dynamic and static WL. *Dynamic WL* is generally simpler in implementation and introduces small overhead. The name "dynamic" is used to point that this type of WL is targeting only those regions of Flash memory, which store dynamic data, i.e., data that is frequently modified. The simplest implementation would only influence the algorithm of selecting a next free block to write data (either from incoming user requests or from GC). The FTL picks up the block with the lowest number of P/E cycles from the pool of free blocks. If the updates are not skewed, but rather evenly distributed over the whole address space, the dynamic WL would provide strong longevity guarantees, while keeping the additional overhead being negligible. However, such access pattern is rather artificial, and rarely can be found in real life applications. Typically, the write requests are very skewed, and different data entities (files, tables) or even their parts have different frequencies of updates (e.g., static, cold, warm and hot data regions). In this scenario the effect of having a dynamic WL would be almost equal to having no WL at all, because regions of Flash memory holding static or cold data (e.g., up to 80% in OLTP-like workloads) would not be considered by WL, thus leading to uneven wear-out of Flash memory.

*Static WL* solves the above problem by considering the complete physical address space

---

regardless of the properties of data stored in them. Those WL algorithms keep track of the whole SSD's wear-out picture, and try to level the erase counts of all available blocks being within a current interval. If certain blocks appear to have less erases than the lower edge of the current interval, the FTL picks up those blocks to be the next victim blocks for GC. Even if there are no invalidated pages in those blocks (so that there is actually nothing to garbage collect), GC would copy the block's pages (most probably holding cold data) to another block, while the victim block after its erasure would be used for storing the new data (probably hot one). By doing so, the hot data (as well as cold) would migrate through the complete address space, and make the Flash memory wear out evenly. Clearly, depending on the workload the static WL might introduce a significant I/O overhead, which interferes with the user requests, and thus would slow down the SSD performance.

### **Bad-Block Management**

Despite the even wear out of Flash memory ensured by proper WL, some Flash blocks can (and will) become invalid much earlier than their estimated lifespan defined in P/E cycles. The reason is typically small defects in FGTs made during the manufacturing process. Even the out-of-box Flash chips usually have some few invalid blocks. For instance, in OpenSSD boards we have used in our tests throughout this work, each Flash chip had about a dozen of invalid blocks (out of 4152) before the first SSD use. That is why all FTL schemes keep track of current invalid blocks, so that they are not repeatedly considered for storing new data. This is called a *bad-block management*, and it is typically implemented as a simple bit-vector with one bit per physical Flash block, which indicates its validity.

### **Error-Correction Codes**

Flash memory is vulnerable to different types of errors, such as retention errors, program interference errors and read disturb errors (see Chapter 2.3.1). Since the frequency of their occurrence is directly correlated to the wear of Flash, the proper WL is very important in keeping the overall error rate low. However, since bit errors are inevitable, FTL must provide a mechanism for their detection and correction. This is done via implementation of a certain kind of error-correction code (ECC).

Depending on the bit error rate (BER) common for a certain type of NAND Flash, and computational characteristics of the SSD controller, different ECC algorithms might be more applicable than others. Moreover, a particular ECC algorithm provides different guarantees (strength of ECC) depending on its configuration parameters (e.g., number of parity bits for fixed-sized data chunk). The common ECC algorithms used for MLC and TLC Flash types are BCH and LDPC, with code strength of approximately 64 parity bits per 2KB data chunk. Some additional details about BCH ECC are provided in Chapter 7, where we propose a modified algorithm to support the in-place appends on Flash. The

---

comprehensive analysis of ECC codes used in modern SSDs can be found in [15].

As you can see, FTL is the complex software layer, where different modules are highly correlated with each other, having often opposite influence on characteristics of SSDs. Moreover, all aforementioned FTL modules have multiple variations, each of which might be more efficient than the others under certain types of workloads. However, the SSD manufacturers typically do not know the details of workloads in which their products would be used, and usually have only limited hardware resources of SSD on which the FTL is running. Thus, they try to play it safe and compose the FTL scheme such, that it performs "good" under the majority of workload types. But as we show later in this thesis, this "good" is generally far away from the possible performance the SSD can provide. One of the key contributions of this thesis is a solution to that "one size fits all" problem, which we describe in Chapter 6.

---

## 3. NoFTL Approach

---

In this chapter we provide an end-to-end overview of the NoFTL approach for a better orientation. All aspects and evaluation results will then be discussed in detail in the four following chapters.

Flash SSD is the main storage device for high performance database systems today. Although HDD is, and certainly will be for a while, a part of the majority of storage systems, its role has changed irrevocably. HDD is used nowadays primary for storing cold and archived data, while SSDs are the first choice for operational data. Performance, price per IOPS, energy efficiency, robustness and small form factor - are the main reasons behind the success of SSDs. It took about 30 years for Flash technology to achieve this state, and it is definitely not the endpoint in its development.

"Performance hunger" of data intensive applications today goes often beyond the possibilities of even main memory DBMSs, not to mention the I/O-bound systems dominating the world of big data. This demand forces both industry and academia to continuously look for faster storage. While the new generation of non-volatile memories (NVM) is being actively researched and developed, it probably will take a decade until it becomes economically attractive and will replace Flash. But even then Flash storage will not disappear, but rather take a niche currently occupied by HDD. Thus, further development and improvement of Flash SSDs is a highly relevant topic today.

As we have seen earlier (see Chapter 2.3.2), SSD is a complex device, the performance of which equally depends on hardware and software development. In this work, we concentrate on the software side, and propose an approach called NoFTL, which allows to significantly improve performance ( $>2x$ ) and longevity characteristics ( $>2x$ ) of Flash SSDs used by database systems. Those results are achieved by solving multiple issues and drawbacks present in almost all modern SSDs (see Chapter 1.2). While those problems are rooted in different levels of the I/O stack, have different nature and influence, they all have a common root - the black-box architecture of SSDs and their backwards compatibility with the traditional block-device interface, which are both realized by the FTL (see Chapter 2.3.2). As the name of our approach reveals, we propose to remove FTL from the I/O stack altogether with other abstraction layers between the storage device and the DBMS, such as file system and block-device layer. By doing so, we let the DBMS operate directly on

the raw Flash. All required Flash memory functionality that was previously encapsulated and hidden in the on-device or on-host FTL, becomes now visible and the responsibility of the DBMS (Figure 3.1).

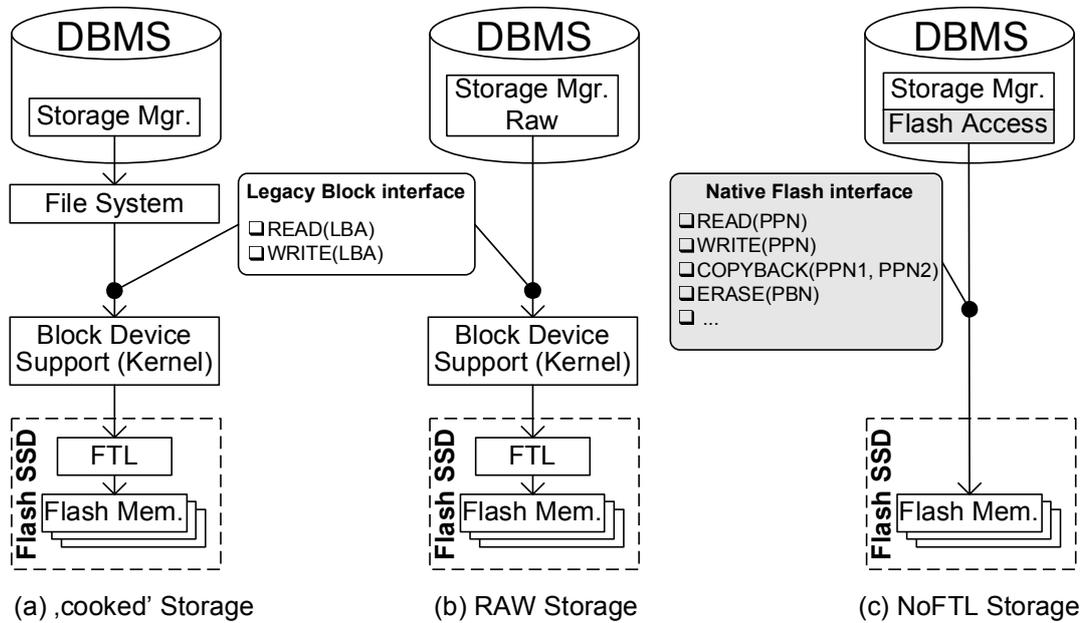


Figure 3.1.: NoFTL as a storage alternative for DBMS. Source: Hardock et al. [34].

It is important to note, that removing intermediate layers from the I/O stack alone is not the solution to issues with SSDs, it is more a prerequisite for various optimizations to be realized. Without FTL, FS and block-device layer an application acquires full control over the Flash memory. However, unlimited control requires the proper knowledge and techniques to exercise it. That is why, in the NoFTL approach, we give this control over the Flash memory to the DBMS - the only application that has all the important knowledge about the data and the workload. Recently, there were a few similar approaches proposed, which also try to reduce the disadvantages of SSD's black-box architecture (see Chapter 9). However, they either provide only very limited control over the storage to the application and/or give this control to the OS or FS. As a result, the lack of control and/or the lack of information do not exploit the available performance of SSDs completely. Only the combination of full control and knowledge about data and workload makes it possible. Having both of those, the only missing part are the proper techniques. In this work we have implemented and evaluated multiple of those, proving thereby the effectiveness of

---

---

the concept by showing significant improvements over the traditional FTL-based SSDs. Further approaches and optimizations under the NoFTL architecture are outlined as future work in Chapter 11.

### **Native Flash Interface**

The first important milestone in the realization of the NoFTL concept is the design and implementation of the native Flash interface. It is utilized by the DBMS to operate on FTL-less SSD directly, i.e., without block-device layer and file system in-between. The traditional block-device interface and its infrastructure are unsuitable for operation on native Flash memory.

The block-device layer manages only two basic I/O commands - read and write of a chunk of data. While this is sufficient for HDD, Flash memory requires additional commands. Thus, the operation on native Flash demands a third basic command - erase of a Flash block. While it is possible to pass management commands from applications directly to the device driver (e.g., `ioctl`), this solution needs an additional synchronization between the queued I/O requests in block-device layer and bypassed erase commands<sup>1</sup>, which will introduce a significant overhead. However, the erase command is only the tip of the iceberg.

Continuously growing importance and popularity of analytical requests, which process huge amounts of data, introduce significant issues for modern DBMSs. It gets especially challenging when OLAP and OLTP workloads are being mixed and carried out by the same database systems and servers, which is often done today in order to reduce the deployment costs and avoid synchronization complexity. The footprint of analytical queries on CPU, memory, storage and network has a significant negative impact on the transactional throughput. As a result, the system demands more hardware power, and the DBMS must do a good job to schedule the requests and ensure fair consumption of available resources. But even in "pure" OLAP servers growing data volumes and more stringent requirements to response times make storage and network the most limiting factors.

On the other side, modern Flash and NVM storage devices are equipped with powerful computational units, such as multi-core ARM controllers, FPGAs and GPUs. While currently this computational power is used solely to deal with the high I/O parallelism of Flash and to perform various management tasks (FTL), it offers also a great potential to be used for data processing. The approach of moving certain data processing tasks down to the

---

<sup>1</sup>The TRIM command introduced in the ATA protocol, which might be submitted by the application does not require synchronization with the I/O requests because it provides just a hint to the SSD about the invalidity of a certain range of LBAs

---

---

storage (aka near-data processing (NDP)) is not new, and it was intensively researched in academia in the past and recently rediscovered (see Chapter 9.4). However, despite the significant performance improvement shown so far, as well as enormous potential of NDP on new generations of Flash and NVM devices, the industry is still lagging behind in development of NDP-capable storage. Thus, to the best of our knowledge there is no device available on the market today which supports data processing functionality. It is, however, clear that NDP will be an essential part of the storage of the future, and the current activity of industry in the NDP research is proof of that.

One of the main obstacles to support NDP lies in the block device interface and impossibility to efficiently extend it with new data processing commands. Therefore, we propose the concept of native Flash interface (NFI) - an open and extendable interface for direct communication between application and FTL-less Flash storage. NFI gives the application a direct control over the underlying Flash memory, as well as the possibility to utilize the computational resources of the SSD. It is important to mention, that although NFI is a significant and essential part of the NoFTL architecture, in this work we provide only the basic description of the interface. We have implemented a simplified version of it in our prototype, which was enough to prove the validity and potential of NoFTL. The extensive analysis and development of NFI (or native storage interface, in general) is the topic of a separate, self-contained and substantive work (more detail on the native storage interface can be found in [101]).

### **DBMS Integration**

The NFI enables the DBMS to operate on FTL-less SSDs. The next step in the realization of the NoFTL concept is integration of the Flash management functionality into the DBMS. Removing FTL (as well as file system and block device layer) from the I/O stack requires the database system to take full responsibility for tasks such as wear-leveling, bad-block management, logical-to-physical address translation and garbage collection. The DBMS becomes the sole decision-maker for physical data placement on Flash storage. It is important to mention, that NoFTL is not about a simple shift of FTL from the on-device controller to the DBMS. Rather, Flash management is integrated into the different subsystems of the database system (Figure 3.2), like buffer manager, storage and free space managers, transaction manager.

From the very beginning of the NoFTL design, we were aware about, and always kept an eye on the fact that this approach would introduce additional code and management overheads for the DBMS, as well as that it would consume extra memory and computational resources of the host system, as compared to the traditional storage architecture on the FTL-based SSDs. However, our NoFTL prototype realized in the open-source DBMS and

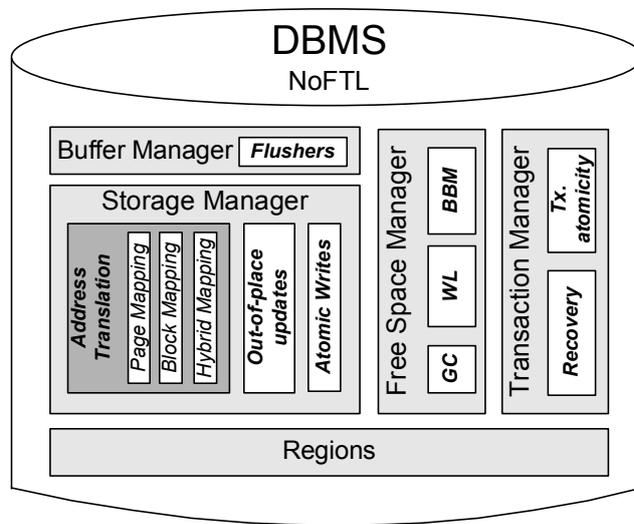


Figure 3.2.: Integration of Flash management into the modules of the DBMS.

evaluated extensively on the emulated and real Flash storage have shown that some of those risks did not materialize, while the effects of others are (negligibly) small and do not become an issue. On the other side, the advantages of the NoFTL architecture are significant and valuable. There are several reasons behind this.

- Integration of the Flash management functionality into the subsystems of the DBMS is generally "light" and "natural". In fact, the typical database system maintains on its own a kind of address translation and garbage collection. Also, some database engines utilize out-of-place update strategies (e.g., log-based storage, copy-on-write technique). Thus, corresponding tasks of Flash management are not added to the DBMS, but rather cause just a modification of the existing modules and techniques.
- Giving the DBMS exclusive control over the data placement (in both logical and physical address spaces) allows to significantly reduce the functional redundancy along the I/O path, which consequently lowers the memory footprint and CPU utilization. Thus, tasks like data placement, address mapping, free space management, garbage collection, caching and recovery are performed redundantly by the DBMS, FS and FTL. Moreover, apart from the savings in resources, removing those redundancies has a clear advantage for reducing the write-amplification on SSDs, which in turn, improves the I/O throughput and longevity characteristics of the device.

- 
- Through the elimination of layers of abstractions valuable cross-layer optimizations become possible under NoFTL. On the one side, Flash management tasks being integrated into the subsystems of the DBMS get access to the rich meta information and statistics maintained in the database system. This data is typically hidden from the FTL, and can not be accessed via common block-device interface. In other words, modern SSDs know only the logical addresses of the pages to be stored, and have no clue if those pages belong to a cold, static relation or to a frequently accessed, dynamic B-Tree index, etc. However, as we will show in our evaluations the utilization of this metadata allows to significantly optimize garbage collection, address translation strategies and wear-leveling algorithms. On the other side, under NoFTL the DBMS gets information about the physical resources of the SSD, i.e., memory layout (boundaries of its parallel units, number of data channels, etc.) and computational resources available for executing near-data processing commands. This information allows the database system through NFI to perform physical data placement so, that the available SSD parallelism is utilized completely; and to delegate certain data processing tasks to the device, reducing thereby the load on host resources (CPU, memory) and network.

### Configurable Storage

Although the evaluation results of our initial NoFTL prototype have shown significant improvements (up to 3x) in system performance and device longevity, which were based on the optimizations achieved by DBMS integration mentioned above, it became clear that this is not enough to achieve the whole potential of the Flash memory.

As we have already mentioned (see Chapter 2.3.2), every Flash management task (i.e., garbage collection, wear-leveling and address translation) has multiple alternative realizations. Which is best depends on the workload properties. Thus, a certain FTL scheme might perform best in terms of performance characteristics, memory and computational footprints, as well as stress on Flash wear under one workload, while being the worst choice for another. And the difference between these "best" and "worst" can be as much as an order of magnitude or even more (e.g., measured in I/O response time or write-amplification). That is why the FTL designers are facing such a hard challenge. All SSDs today are provided with a single, static FTL scheme. At the same time, the manufacturers don't know in advance the details of the workload on the client side, nor do they give an opportunity to change the FTL scheme by (or for) the client. Furthermore, they do not provide any details about the FTL scheme used in their SSDs so that a client can choose the most appropriate product for his system. Thus, the FTL must be chosen to perform reasonably well for a variety of workloads. As a result, a generally good FTL scheme is

---

---

typically much worse than the optimal one for a particular workload.

In contrast to this "*one size fits all*" approach of modern SSDs, NoFTL has an ability to select the most appropriate set of Flash management algorithms. Thus, for instance, based on its own metadata and statistics the DBMS can take a decision to use a page-level address translation (e.g., DFTL [28]) or a variant of a hybrid mapping (e.g., FASTER [56]). Further, it can tune the parameters of the selected scheme to better match the current workload (e.g., vary the size of cached mapping table in DFTL depending on the load and access pattern).

Databases consist of dozens and hundreds of different objects (i.e., relations and indexes), and each of those typically has unique properties. Size of an object (small, large, constant, dynamic), access frequency (hot, warm, cold), read-write access ratio (read-only, read-intensive, write-intensive), distribution of accesses (random, sequential, skewed), and stability of access pattern - are the main characteristics of the database objects. OLTP-like workloads have usually a full range of those objects and access patterns. Thus, a single Flash management schema applied to the whole database would be more suitable for one group of objects, less efficient for another group, and could be an outright poor choice for yet another part of database. As a result, the more diverse a database and the workload are, the more overhead (memory, CPU, I/O) is caused by the Flash management. The NoFTL architecture offers a unique solution to this problem by allowing multiple Flash management schemes to be utilized simultaneously for a single SSD. This becomes possible due to the combination of two main factors: 1) the DBMS has **full control** over the Flash memory and Flash management; and 2) the DBMS has **complete knowledge** about the data and the workload running on it.

The challenge, however, is realizing these concepts. How to use a certain FTL scheme only for a particular group of objects? How to separate objects on Flash memory? Furthermore, the latter is closely related to the following questions: how to control the utilization of available I/O parallelism of SSDs, and how to provide an efficient hot-cold data separation, which is crucial for the GC overhead, and thus for performance and longevity characteristics of the device. To answer these questions we introduced two novel physical storage structures - *regions* and *groups*.

**Regions.** A region is a set of NAND chips, which is constant in size, but dynamic in its composition, i.e., chips might change their owner-region. One or multiple database objects are assigned to a region, which makes them to be physically stored in the corresponding chips. Thus, every object (or its declared partition) is assigned to exactly one region, while every region can store multiple objects. Based on the properties of objects assigned to a region, the DBMS selects the most appropriate set of Flash management algorithms for that particular region. For instance, the database system might decide to put multiple frequently accessed, dynamic indexes together in one region, and maintain it with the most

flexible page-level mapping scheme. Furthermore, the parameters of garbage collection and a variant of local (per-region) wear-leveling would be chosen appropriately. Yet, another region can be dedicated to objects that are updated in an append manner or in large logically sequential chunks. For this region, the block-level or the hybrid mapping schemes would be the perfect choice (Figure 3.3).

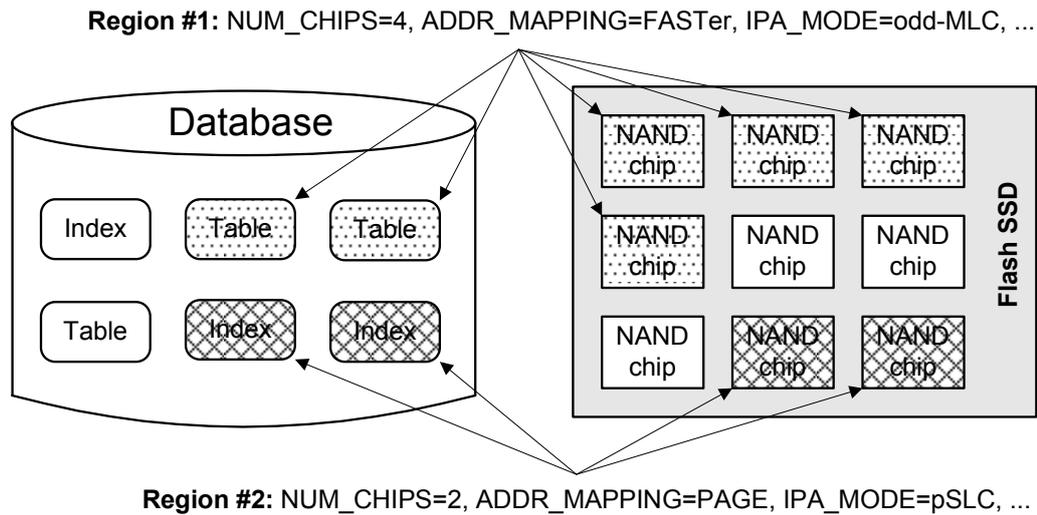


Figure 3.3.: Example of a region configuration.

As we will see in Chapter 6, apart from choosing and configuring general Flash management algorithms (address translation, GC, WL), regions allow us to select the appropriate

1. NAND mode for the corresponding chips (e.g., MLC Flash can be used in pseudo-SLC mode or in MLC mode);
2. update strategy (e.g., out-of-place updates or in-place appends, see Chapter 7); and
3. error-correction scheme.

Regions serve also two other important functions. First, they allow us to control the utilization of available I/O parallelism of Flash storage, and second, they can be used as means to perform hot-cold data separation. The common approach applied in modern SSDs is to distribute the logical address space evenly over the whole Flash memory. Typically, a simple modulo function

$$Chip\_Id = LPN \bmod Total\_Num\_Chips$$

---

is used to assign logical addresses to a certain chip (or die). This strategy has multiple advantages. It ensures very good load balancing, because all data (e.g., all database objects) will be evenly distributed over the whole storage. Consequently, every object can theoretically utilize the maximum access parallelism. Moreover, such data placement can simplify wear-leveling algorithms, due to the fact that data with different update frequencies is mixed in Flash blocks. However, this simple approach has also multiple disadvantages, which usually dominate. Mixing of data with different properties physically on Flash memory inevitably leads to increased overhead of garbage collection. As a result, the on-device write amplification rises significantly, which in turn, negatively impacts I/O response times. Apart from the slowdown in performance, this data placement strategy causes faster wear of an SSD.

In contrast, regions enable us to physically separate database objects with different access characteristics, i.e., perform a hot-cold data separation. Depending on the requirements of objects the DBMS can decide on the proper number of chips (dies) composing a region. Cold objects or hot but small objects that are normally completely cached in the DBMS buffer do not require much parallelism, while frequently accessed large indexes and tables demand a high level of parallel access. The evaluation results presented in Chapter 6.1 show that the intelligent data placement using regions allows us to utilize the available resources of Flash memory much more efficiently than the traditional approach with perfect load balancing. It results in better performance and longevity characteristics of the device.

**Groups.** Summarizing the above, a region is a physical storage structure, which enables selective Flash management, hot-cold data separation and control over the Flash parallelism. However, it is not always possible (due to size restrictions), and typically even not reasonable (due to potentially low access parallelism) to place every database object in a separate region. Thus, multiple objects commonly share a region. While the DBMS tries to group objects with similar access properties into regions, every object has a unique access pattern. This motivated us to look for the possibility to perform finer hot-cold data separation within a region. We solved the problem with another physical storage structure - *group*. Every region has one or multiple groups, and each database object within this region is assigned to a certain group. The idea behind groups is very simple - every Flash block contains pages only from one group. For instance, if there are five relations assigned to a certain region, and further each of those relations is assigned to a separate group, then every utilized Flash block in this region will contain data only from one relation. Whenever the GC selects a victim block for space reclamation, with very high probability it will contain pages with similar update frequency. This will lower the amount of work needed to copy the valid data within the block. As a result, the GC overhead is minimized leading to lower response times and increasing the device longevity. More details about

---

*groups*, their implementation, influence on GC and WL, as well as evaluation results are presented in Chapter 6.2.

**Advisor and Migrator.** While regions and groups are means for realizing the concept of configurable storage and solving the *one size fits all* issue of FTL-based SSDs, one question still remains - how to select an appropriate configuration for the database. This question is of particular relevance in systems where the workload is dynamic. Even the optimal data placement configuration will become potentially suboptimal once the workload changes. We address this issue by proposing two modules, *Advisor* and *Migrator*, as part of the general NoFTL concept.

*Advisor* is responsible for suggesting multiple preferred data placement configurations based on the DBMS statistics and metadata. *Advisor* might be run on demand (e.g., in systems with relatively static workload characteristics) or automatically at a selected interval. Since finding an optimal NoFTL configuration is similar to the data placement problem in distributed systems, which is known to be NP-hard [8], the core of the *Advisor* is an algorithm based on heuristics. Research on these heuristics is a complex and vast research topic. To prove the concept of the *Advisor* module, we limited ourselves to a simple heuristic algorithm in this thesis. Despite its simplicity the *Advisor* was able to suggest multiple data placement configurations, which were significantly better than the naive strategy applied in SSDs. Research into more advanced heuristics is left as future research.

For certain systems (e.g., databases with several dozens of objects and simple transactional profile) choosing an appropriate data placement configuration might be easily done manually based on the visualized statistics provided by the *Advisor*. However, for databases with complex workload and hundreds of objects the role of the *Advisor* in supporting the database administrator (DBA) becomes critical. In those cases the DBA would solely select one of several configurations proposed by the *Advisor*.

It is worth noting, that the introduction of two novel storage structures does not overcomplicate the work of the DBA. This is because regions and groups perfectly match with the existing structures utilized by the DBMS. Thus, regions are coupled to tablespaces, while groups are linked to extents. Operating on traditional HDDs, tablespaces and extents have a direct (raw storage) or indirect (cooked storage) impact on the physical data placement. Extents serve as a means to provide a trade-off between efficient space utilization and advantageous sequential accesses, while tablespaces allow the separation of different groups of objects for better logical and physical manageability. However, on FTL-based SSDs these are becoming purely logical storage structures, which do not have any impact on the physical data placement on the underlying Flash memory. This makes extents useless, and lessens the value of tablespaces. By coupling those structures to regions and groups under the NoFTL architecture we basically restore their power to influence and

---

---

control the physical placement of data.

While the DBA, assisted by the *Advisor*, selects the data placement policy, these policies must be executed. The *Migrator* executes and enforces the desired Flash management and data placement configurations. It is a module integrated into the DBMS that takes care of the transitions from one configuration to another. The key characteristic of the *Migrator* is the ability to perform configuration changes in parallel to the normal DBMS work with minimal interference on the user load. *Migrator* is also used by the global wear-leveling strategy to exchange chips between different regions in order to prolong the lifetime of the SSD. More details about *Advisor* and *Migrator* are presented in Chapter 8.

### In-Place Appends

One of the optimizations that we have realized within the NoFTL architecture is the approach of in-place appends (IPA). The charm of IPA (apart from its results), and the reason why we dedicated a separate chapter at the end of this work solely to this approach (Chapter 7), is its ability to perfectly show the complete NoFTL concept in action. The basic idea of IPA is based on the fact that, although NAND Flash memory is well known to be addressable in Flash pages and to follow the erase-before-overwrite principle, under certain conditions it is possible to update an already written page physically in-place without foregoing erase operation. For instance, when a write request does not change the previously written data on the page, but rather appends new data to the page's free space, then such an update is a good candidate to be performed in-place. While in traditional FTL-based SSDs the utilization of this property is hard to realize and is associated with an extra overhead, the NoFTL architecture makes the realization easy and efficient.

Our analysis of typical OLTP workloads has shown that the lion's share of updates performed in those systems modify only few, at most several dozens of bytes in tuple' data on a page (see Figure 3.4). The dominant operations on records performed by transactions are as simple as changing a single numeric field (e.g., increasing a customer's balance), updating a date or a status attribute. However, despite the small size of changed net data, the DBMS submits the whole page (e.g., 4-16K) to the storage. If the file system is used on top of the SSD, then it typically amplifies the volume of written data further. Yet, as if this wasn't enough, the SSD's FTL contributes on its own to the final volume of data physically written on Flash memory. Thus, these three types of write-amplification along the I/O path turn 100 changed bytes into up to 60 thousand bytes written on Flash. This results in significant stress on the network and storage, and consequently negatively impacts system performance and longevity characteristics of the Flash.

The desire to modify Flash pages without needing to erase the block, and avoiding the huge write amplification of OLTP workloads lead us to propose an IPA approach. Small

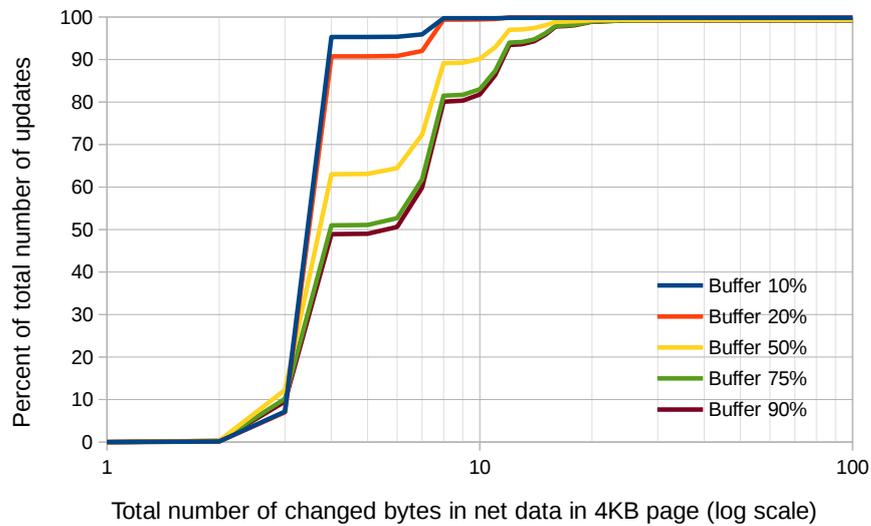


Figure 3.4.: Cumulative distribution of update-sizes in TPC-B benchmark under default eager-eviction strategy with different sizes of DBMS buffer. Source: Hardock et al. [31]

changes to a database page are captured in a special delta record, which is then appended to the very same physical Flash page the original data is placed in. To ensure that the Flash page has enough space to accommodate certain number of delta-records, we initially reserve a small amount of space at the end of the page (e.g., 1-5%). The technique to transform updates within a page into appends is not new, has many variations, and is widely applied by database systems (e.g., [71], [50]). However, the unique benefit of IPA is application of this technique for modifying the original Flash pages without performing an erase operation on the corresponding Flash block. When using IPA we re-use original Flash pages and save on multiple operations, which are typically performed in combination with an update: (i) invalidation of original Flash page; (ii) out-of-place write of modified page; (iii) multiple page migrations and erase operations performed later by garbage collection to reclaim the space occupied by invalidated pages. As a result, applying IPA can reduce write amplification and number of erases on Flash under OLTP workloads up to 80%. These savings effect, in turn, I/O response times (e.g., -50%) and expected longevity of SSD (e.g., +2x).

Although the IPA approach can be applied also for traditional black-box SSDs, its realization under the NoFTL architecture is easier and more efficient. There are several reasons for this.

- 
1. Using extensibility of the native Flash interface we added a new command to support IPA, *delta\_write*, which allows the DBMS to submit only small delta records to the SSD, instead of whole pages. This helps to further reduce the write amplification and relieve the pressure on the storage bus. In contrast, the rigidity of a block device interface would prohibit to introduce a modified write command, which can transfer data chunks of arbitrary size (e.g., 100 bytes) to the SSD.
  2. Direct control over the physical data placement and integration of Flash management into the DBMS enable the DBMS to recognize if the page can be updated using IPA, and then decide on the type of delta record used depending on the object type and its modification pattern (e.g., offset-value pairs for relations, logical operations for B-Tree indexes, etc.).
  3. Using the notion of regions we can enable and configure application of IPA for every region independently based on objects stored in it. The decision regarding if and what type of IPA should be utilized for any particular region is easily guided by suggestions of the *Advisor* module.

### Summary

Native Flash interface, integration of Flash management functionality into the subsystems of the DBMS, as well as the concept of configurable Flash storage are the main elements building the core of the NoFTL architecture. The combination of them enables various optimizations in different modules of the DBMS (buffer manager, storage manager, recovery, query optimizer and query processing engine). In this thesis we realized and evaluated multiple of those, resulting in significant performance speed-ups and improved longevity of the Flash storage. We want to emphasize that the set of optimizations described in the following chapters is neither complete nor exclusive. Our main goal was to provide the conceptual description of the NoFTL, design and implement its prototype, and prove its advantage over the traditional, FTL-based, black-box and backwards compatible design of modern SSDs. Thus, further improvements under the NoFTL architecture are possible, and some of them are outlined in Chapter 11.

To give a first impression of the performance improvements achieved under the NoFTL architecture consider the following evaluation results. Figure 3.5 shows the transactional throughput for the tests performed on real Flash SSD<sup>2</sup>, where the DBMS was running the TPC-C benchmark [96] under an I/O-bound configuration. Figure 3.6 depicts write amplification and intensity of erase operations for the same tests.

---

<sup>2</sup>Jasmine OpenSSD platform, see Appendix A.1

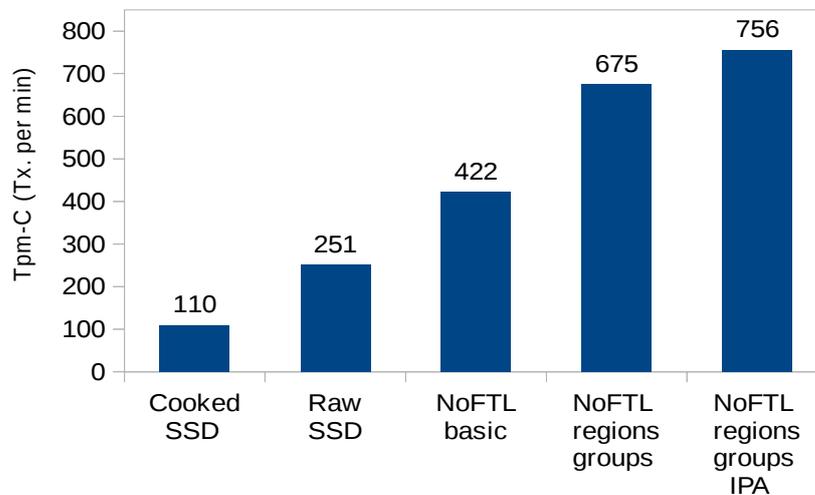


Figure 3.5.: Performance comparison of DBMS on different storage alternatives under TPC-C benchmark. Figure is also presented in Chapter 1.

The first two bars on the left in both figures show the results for FTL-based SSD under traditional storage alternatives - cooked storage (with ext4 in journaling mode) and raw storage (see Chapter 2.2). As one can see, alone the file system can almost triple the overall write amplification, which has a direct impact on the longevity of Flash storage, as well as on the database performance, especially in I/O-bound systems. While Youyou Lu et al. in [59] have reported similar results for ext2 and ext3 file systems running TPC-C benchmark, they have also shown that under certain workloads FS write-amplification might be as high as 11, meaning that on average a single write I/O submitted by the DBMS is turned into eleven write requests performed by the file system.

Yet, a simple elimination of a file system and utilizing the raw storage alternative on the FTL-based SSD does not solve the problem. Backwards compatibility and black-box design of modern Flash SSDs cause significant overhead, and do not allow us to utilize the full potential of Flash memory. This becomes visible by comparing the bars *Raw SSD* and *NoFTL basic*. The latter corresponds to the earlier stage of our NoFTL prototype with only few optimizations, which were realized through integration of Flash management into the DBMS. But even this implementation allows to further reduce ( $>2x$ ) the system's write amplification and intensity of erase operations. The next bar shows the effect of applying the concept of configurable Flash storage. Here, through intelligent data placement and selective Flash management achieved by applying regions and groups, the write

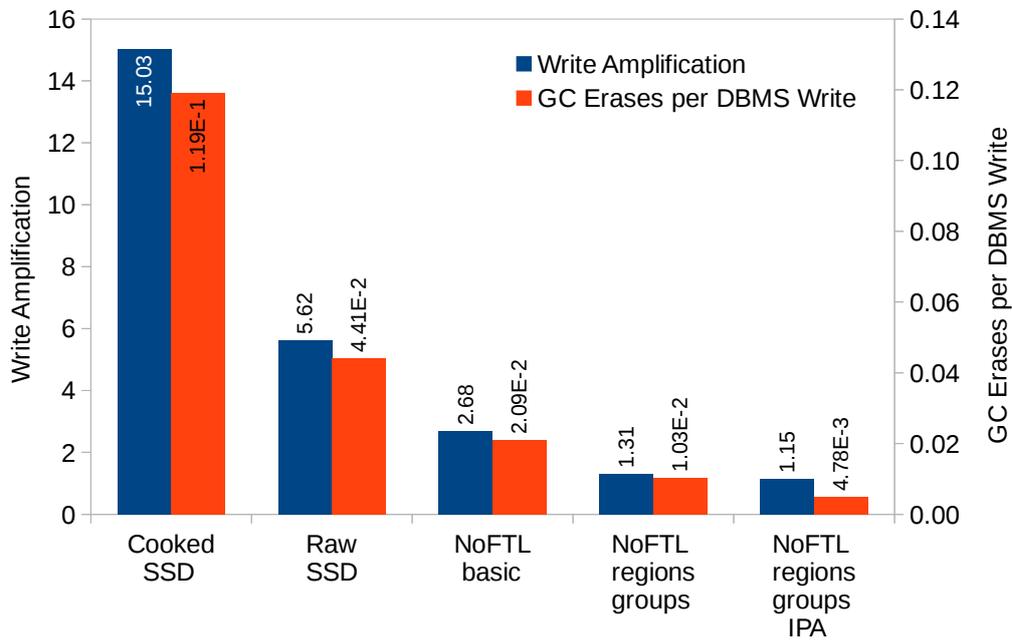


Figure 3.6.: Comparison of write amplification and intensity of erase operations for different storage alternatives under TPC-C benchmark.

amplification of SSD can be reduced to 1.31. In the last configuration, the GC overhead is further reduced by applying an IPA approach for certain database objects, which in turn, positively impacts the device longevity and the transactional throughput. More details and analysis on these tests and further evaluation results are presented in the following chapters.



---

## 4. Native Flash Interface

---

One cornerstone of the NoFTL architecture is the native Flash interface (NFI). Through elimination of multiple abstraction layers along the I/O path (FTL, block-device interface and file system) the DBMS gets direct access to the Flash storage. Novel storage structures (Flash pages, blocks, dies, etc.), behavior of Flash memory that is different from HDDs (erase-before-overwrite principle), as well as access to computational resources of the SSD (FPGA, GPU) require also a new communication interface with the corresponding set of commands. This interface differs from the traditional block device interface in the set of I/O operations, the request granularity, the addressing paradigm, extensibility and pushdown support.

### 4.1. Command Set

The block device interface relies on a basic set of SEEK, READ and WRITE commands. Protocols such as SATA introduce additionally the TRIM command. The native Flash interface defines READ, PROGRAM, ERASE, COPYBACK, WRITE\_DELTA, GET\_ADDR\_TABLE, as well as a set of management commands and near-data processing (NDP) commands.

The basic set of commands used for evaluation throughout this work are:

- PAGE\_READ – reads a physical Flash page from a specified physical address and transfers it back to the host.
- PAGE\_PROGRAM – transfers a page to the Flash device and writes it to a specified physical Flash address.
- BLOCK\_ERASE – erases a block of Flash pages.
- COPYBACK – copies data within the SSD without involving a host. Basically, there are two variations of how this command is executed on SSD. In the first case, when the data needs to be moved within the same Flash plane, SSD uses the native NAND operation *copyback*. This uses only the plane's page register as temporal storage (NAND page → plane's page register → NAND page), and is the fastest way to

---

copy a NAND page to a new location. In the second case, when data is copied from one plane/die/chip to another, SSD's DRAM must be used as additional temporal storage (NAND page → plane's page register → DRAM → plane's page register → NAND page). Despite a certain latency difference of both these variations, we will use the term COPYBACK command without further differentiating (if not clearly stated), generalizing a data move within SSD without data transfer to/from the host. The typical use case for the COPYBACK command is page migrations performed by garbage collection (see Chapter 2.3.2).

Erase and Copyback commands do not transfer any data to or from the storage device, rather they issue analogous commands on the corresponding Flash chip.

- GET\_GEO – an identification command (similar to HDIO\_GETGEO for HDDs), which allows the DBMS to receive detailed information about the architecture and geometry of the Flash device (channels, LUNs, Flash type, etc.)
- GET\_ADDR\_TABLE – retrieves a part of the latest consistent address mapping table and transfers it to the host (see *Recovery in NoFTL* in Chapter 5).
- WRITE\_DELTA – performs an append to a specified physical page (see Chapter 7 for more detail).

Meaningful are also the variants of the PAGE\_READ and PAGE\_PROGRAM commands to support reading or writing a series of pages (not necessarily physically adjacent). Such variants are for example READ\_CACHE\_RANDOM, READ\_CACHE\_SEQUENTIAL, PROGRAM\_CACHE\_RANDOM, PROGRAM\_CACHE\_SEQUENTIAL. These commands will be mapped into appropriate optimized commands according to the Flash specification (e.g., ONFI NAND).

Under NoFTL the database page size is a multiple of a physical Flash page. The Flash page metadata is used by the DBMS and the Flash controller for storing Flash maintenance data (ECC, write count, LPN, etc.).

## 4.2. User-Defined Commands

Another operation subset of the native Flash interface are the so-called *push-down* commands, aiming to (i) leverage the computational power of the Flash storage device, and (ii) reduce the amount of data transferred to the host by performing the data processing tasks (e.g. selection, join) and certain Flash management tasks on the Flash device itself. The near-data processing under the NoFTL architecture is actively researched at the moment

---

in our group. The detailed description of the push-down commands, their usage by the DBMS and the evaluation results are the scope of another thesis.

### **4.3. Addressing and Granularity**

One key principle of the block device interface is that it supports stable host/logical addresses – the so called LBA. These are defined once a device is connected and its geometry retrieved and remain immutable. Moreover, the granularity of the addressed units (so called sectors) is uniform. All operations are performed on those stable LBAs, and have uniform granularity.

All NFI commands address Flash pages or blocks with their physical addresses. In a prototype we have demonstrated that byte-granularity is also possible (see Chapter 7). This gives the DBMS full control over the physical data placement on the Flash storage. It is important to note, however, that although the physical addresses of the database units change continuously on the Flash memory (out-of-place updates, wear-leveling) their logical addresses remain always stable. This is achieved by integrating the logical-to-physical address indirection into the existing mapping tables of the storage manager (see Chapter 5).

The native Flash interface reveals for the DBMS also the new structures describing the physical architecture of the Flash storage, such as chips, dies, planes, blocks, channels, on-device compute units (e.g. SoC ARM controllers, FPGAs, GPUs) and the on-device cache. How those are efficiently managed is explained in Chapter 6.



---

## 5. DBMS Integration

---

The second pillar of the NoFTL architecture is integration of Flash management into the DBMS. Native Flash interface gives the DBMS full control over the Flash storage underneath. However, in order to utilize this control efficiently and realize the performance potential of Flash, the DBMS must take the responsibility for multiple Flash management tasks. The four major tasks are logical-to-physical address translation, garbage collection, wear-leveling, and bad-block management. In traditional Flash SSDs these tasks are encapsulated in the FTL scheme, which is executed internally on the SSD and is completely transparent (hidden) for the host system. However, it is important to note, that the NoFTL architecture does not assume simply moving this functionality from the SSD's controller to the DBMS, but rather a deep integration of these tasks into subsystems of the DBMS. This integration is based on two important observations:

- Implementation of Flash management in the DBMS will create a win-win situation for both parties. On the one side, through access to the database system's metadata, Flash management tasks can be significantly optimized. On the other side, by giving the DBMS control over Flash memory, various traditional algorithms of database systems can be improved and simplified.
- Integration of Flash management into the DBMS is relatively light and does not over-complicate the system. In most cases, the corresponding sub-systems of the DBMS themselves implement similar functionality. Thus, integration typically requires small changes of existing algorithms and data structures, and removes functional redundancy along the I/O path.

In this chapter, we look into the integration details of main Flash management functionality, and at the end of the chapter present a performance evaluation that shows the potential of this approach.

The general integration of Flash management into different sub-systems of the DBMS is presented in Figure 5.1. Depending on the properties of a concrete database system the NoFTL integration might also vary. Thus, the type of storage and free-space managers, concurrency control and buffer realization are the main determinants. It is important to

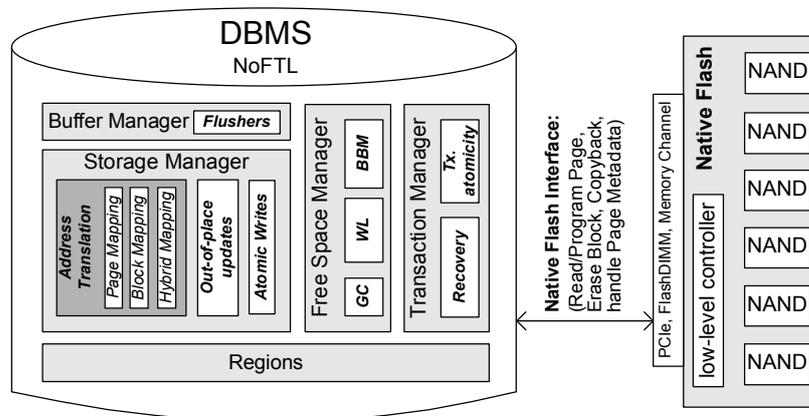


Figure 5.1.: NoFTL architecture and integration of Flash management in the DBMS. Source: Hardock et al. [34].

note, that although Figure 5.1 depicts a clear association of Flash management functions to database modules, it does not show a comprehensive picture of the integration. In practice, even a single Flash management task (e.g., address mapping, GC) is typically spread over (or influenced by) multiple DBMS modules. Thus, Figure 5.1 provides only a rough architecture, showing rather the conceptual place of residence for each functionality that was integrated.

## 5.1. Address Translation

The key component of all FTL schemes is the logical-to-physical address translation. Its realization determines to a large extent also the implementation of garbage collection and wear-leveling algorithms. Thus, the performance characteristics of Flash SSDs are highly dependent on this functionality. The biggest challenge in the realization of address mapping for modern SSDs is the limited capacity of on-device DRAM, which is needed to cache mapping entries. In order to reduce the memory footprint, the SSD designers typically use hybrid mapping schemes (e.g., FASTER [56]) or modified versions of page-level mapping (e.g., DFTL [28]). These schemes, however, cause significantly larger write amplification and negatively impact device longevity (see more in Chapter 2.3.2).

Address translation is also an inevitable part of the NoFTL architecture. On one hand, out-of-place updates, wear-leveling and garbage collection result in mutable physical addresses. On the other hand, the DBMS assumes and assigns immutable *record IDs (RID)*

---

---

or *Tuple IDs (TID)* for the lifetime of the data records. Thus, logical-to-physical address translation is essential.

However, the fact that under NoFTL address translation becomes the responsibility of the DBMS alone solves many of the issues present in modern Flash SSDs. As the host system operates with much larger resources of memory, it becomes possible to utilize the most effective and flexible mapping scheme - pure page-level mapping - for the whole Flash storage. For instance, for a 1TB Flash storage such mapping table would require about 1GB of DRAM. While this amount of memory is almost infeasible for modern SSDs, it is typically not an issue for large data centers.

However, as we will see in Chapter 6, NoFTL does not require even that amount of memory to deliver the best performance. In contrast to the common strategy "one size fits all" applied by traditional SSDs, where a single mapping scheme is used for the whole Flash storage, NoFTL follows the concept of configurable Flash storage. The latter allows to support multiple different address translation schemes simultaneously, and apply those selectively to the data that optimally matches this scheme. With this the total memory consumption needed for address translation is significantly lower than it would be for pure page-level mapping applied to the whole Flash SSD. This is a clear example of how the DBMS information about data can be utilized to optimize Flash management tasks (see more in Chapter 6).

Another advantage of address translation being integrated into the DBMS is the ability to leverage existing data structures. The majority of modern DBMSs already implement some kind of address translation from logical-to-physical addresses. For instance, different variants of Log-based Storage Managers (LbSM, also known as append-based storage), which are especially popular for use with Flash storage, contain high-resolution mapping of logical page numbers to physical block addresses to ensure append I/O behavior. In these systems the integration of address translation required for Flash management is basically a noop at no additional memory cost. NoFTL can simply rely on the existing mapping table.

Giving the DBMS control over the address translation, and thus over the physical data placement, has yet another advantage for such sub-systems as storage and transaction managers, namely the atomicity of write requests.

## 5.2. Atomicity of Writes

Atomicity of write requests is very important for all relational DBMSs to guarantee physical data integrity. The general approach is based on redundancy (e.g., doublewrite buffer

---

---

in MySQL<sup>1</sup>). However, Flash memory has itself a native support for write atomicity. The common out-of-place update strategy designed to handle overwrites on Flash, allows to maintain the original unmodified copy of data pages on persistent storage, while the modified versions of such pages are written to a new location. However, the black-box architecture of modern SSDs hides all details of out-of-place updates from the DBMS and makes it impossible to make use of it. Therefore, running a DBMS on Flash SSDs in a traditional architecture would result in doubling the real data redundancy (as well as the corresponding overhead) without improving the physical data integrity guarantees.

Both academia [46] and industry [74] have proposed solutions for utilizing the native redundancy of Flash SSDs in a DBMS. Kang et al. [46] offloads the responsibility for providing atomic writes to the drive's FTL. This puts additional load on the limited on-device resources of Flash SSDs and makes this approach suitable only for small databases with low-level of concurrency. The approach used in [74] manages atomic writes comprising multiple data pages in FTL that executes on the host system. This approach does not suffer from limited on-device resources, but due to the black-box design and legacy protocols the storage device cannot differentiate between different atomic writes performed simultaneously, and therefore supports only one atomic write of a set of pages at a time. With increasing level of transactional concurrency in modern DBMSs and supported I/O parallelism of Flash SSDs this drawback becomes more prominent.

NoFTL combines the advantages of both approaches, leveraging atomic writes, shadow paging, wear-leveling and physical-to-logical address mapping. It avoids resource limitations, while maintaining the complete information about atomic transaction write I/Os. In contrast to [46] such transaction information must not be passed down the I/O stack. The direct control of the out-of-place updates allows performing atomic writes with a high level of transactional concurrency.

### 5.3. Garbage Collection and Wear-Leveling

Every database management system executes on its own one or several garbage collection tasks. It might be a defragmentation job that coalesces free space on database pages; or a GC job in multi-version DBMSs to purge old, invisible versions of tuples; or garbage collection in log-based storage managers that reclaims space occupied by old block/page versions; or compaction jobs in storage engines based on LSM trees (e.g., LevelDB or RocksDB). Based on the current implementations of storage and free-space managers in the particular DBMS, the garbage collection as part of Flash management might be coupled to the existing cleaning tasks.

---

<sup>1</sup><https://dev.mysql.com/doc/refman/5.7/en/innodb-doublewrite-buffer.html>

---

---

One advantage of integrating garbage collection and wear-leveling into the DBMS (as part of free-space or storage managers) is the ability to provide proper scheduling of these jobs. GC and WL are the most resource and time consuming tasks of Flash management, and thus they are the major source of delay for incoming I/O requests. Consider a measurement of I/O throughput of one enterprise-class SSD provided in Figure 2.9. The clearly seen periodic outliers are a good example of how the GC and WL can influence the I/O performance of SSDs (see more about GC overhead and its influence in Chapter 2.3.2). The DBMS has a better knowledge about the current workload, which might be used to determine intervals when intensive background GC or WL jobs would not significantly interfere with the foreground I/O load. For instance, many database systems utilize some kind of synchronization points (e.g., checkpoints in traditional DBMSs, savepoints in SAP HANA<sup>2</sup>). The duration of these synchronizations is critical as during this operation typically no modification on data is allowed. On the other side, these intervals often correspond to the highest I/O load of the system (e.g., flushing of dirty data and metadata). Under NoFTL the DBMS can postpone the expensive GC and WL jobs during this synchronization work, and thus guarantee stable and high I/O throughput of the Flash storage. Further examples for proper GC/WL scheduling (region-specific) are discussed in Chapter 6.1.2.

The overhead of the GC depends on two main factors: address translation scheme and data placement. For instance, using a hybrid mapping scheme (e.g., FASTER) for hot data with random update pattern would result in a large number of full merges performed by GC, and thus significantly increase write amplification (see more in Chapter 2.3.2). A similar effect on GC overhead is caused by the mixture of hot and cold data in Flash blocks. NoFTL achieves low overhead of garbage collection through optimizing on both these factors. With the concept of configurable Flash storage (see Chapter 6) the DBMS can cluster the data based on its properties into regions and groups, which ensures proper hot/cold data separation. Moreover, for each such region the DBMS selects the most appropriate set of Flash management algorithms (address translation, GC, WL), thus keeping the GC overhead at its minimal.

## 5.4. Parallelism

Many DBMS algorithms can be optimized based on the architecture of underlying Flash SSDs. Direct control over physical data placement allows to efficiently exploit native Flash parallelism. For instance, SATA2 allows for at most 32 concurrent I/O commands; whereas a commodity Flash SSD with 8 to 10 chips is able to execute up to 160 concurrent I/Os

---

<sup>2</sup><https://blogs.sap.com/2017/12/04/sap-hana-savepoint-mechanism-internal-stages/>

---

(each chip runs 8 to 16 commands). NoFTL addresses this by creating regions depending on the desired level of I/O concurrency by varying the number of NAND chips and data channels in the region (see Chapter 6.1). Within a region the parallelism is utilized by applying appropriate striping and data placement techniques. For example, for database objects with highly skewed accesses, larger striping units reduce the garbage collection overhead. In contrast, hot uniformly accessed objects might benefit from smaller striping units, due to higher parallelism [88].

Another technique to utilize the available Flash parallelism is to incorporate the knowledge about Flash architecture into the logic of database writer processes (db-writers). The basic idea is to remove the contention for physical resources among db-writers. Instead of having multiple db-writers, where each is responsible for a subset of dirty pages from the whole address space, we have assigned each db-writer to a NoFTL region. Therefore, each db-writer receives a distinct subset of dirty pages that belongs to a certain region, and does not compete for physical storage with db-writers assigned to other regions. Depending on the workload and the size of the regions it is also possible to assign several db-writers to a single region.

## 5.5. Recovery

Multiple critical Flash-maintenance structures are memory resident under NoFTL: logical-to-physical address mapping tables, block erase counts and other wear-leveling statistics, bad-block tables, etc. *How are these protected against loss in the event of a system crash?* NoFTL guarantees complete recovery from system crashes at negligible costs and runtime overhead.

Each Flash page consists of two parts: data area for storing page data, and out-of-band area (OOB) - for FTL metadata, write counts and error-correction codes. The OOB area can be accessed in two different ways. First, it is always accessed as a part of the regular page access, i.e. whenever a page is read/programmed the OOB area is read/programmed accordingly. Second, the OOB area can be accessed independently, without touching the data area. For the latter type of operations controllers use a special addressing strategy, providing very fast low-level access. Under NoFTL a small part of OOB area of each page (4-8 bytes) is reserved to store the logical page number (LPN), the block's erase count (only in the first block's page) and the page invalidation bit. These data are written back on Flash, together with the page data, piggybacking the regular programming mode. Note that storing this information does not require any additional I/O and does not influence the programming latency. The space requirement of 4-8 bytes of each OOB is insignificant and will not reduce the size of ECC, since there is usually enough space in OOB reserved

---

---

for FTL-specific data. Also note that the process of invalidating an old page version on flash is unchanged and atomic, i.e. it can never happen that a partial write of a new page version invalidates the old version of that page.

In the event of a system crash, prior to initiating recovery, the DBMS under NoFTL issues a special mapping table recovery command *GET\_ADDR\_TABLE* to the Flash controller. The controller then performs a full scan of all the OOB areas in parallel using the special OOB access mode, i.e. all Flash dies are scanned in parallel, and within a die the OOB areas of all pages are read sequentially utilizing an optimized NAND command for serial reads. For further processing only a few bytes out of each OOB area (64-512 bytes) are retrieved. This data is then packed together on units the size of a Flash page by the controller; these are placed in the read cache of the device and subsequently transferred to the server (e.g. using DMA). Hence, the server can easily recover the mapping tables, wear-leveling statistics and bad-block tables. Depending on the mapping scheme used in a particular region (e.g. page-/block-level mapping, hybrid mapping) different Flash metadata might need to be recovered. In no cases, however, more than 12 bytes of a page's OOB is required.

In all our experiments the time to recover NoFTL metadata for the 64GB drive was about 0.65 seconds, while the complete DBMS recovery cycle (Shore-MT uses ARIES recovery) took 100 to 200 seconds. Thus, NoFTL recovery adds less than 1% to the traditional DBMS recovery.

## 5.6. Evaluation Results

The following experiments investigate the performance of traditional FTL-based SSDs compared to the basic NoFTL approach. First, we analyze the overhead caused by modern file system (ext4 with journaling enabled) in traditional cooked storage architecture. Then, we eliminate the file system from the I/O stack in order to analyze in isolation the overhead caused by different FTL schemes used in common black-box SSDs. We compare these results to the basic NoFTL architecture, which does not utilize the concept of configurable Flash storage (discussed in detail in Chapter 6). Thus, the NoFTL scenario uses a simple single-region, single-group configuration with pure page-level address translation table.

Tables 5.1 and 5.2 show the performance results for TPC-C (SF=400) and TPC-B (SF=5500) benchmarks, respectively. After ramp-up phase (after which statistics were reset) both benchmarks were running for 2 hours, and in each scenario three tests were performed, based on which the aggregated values were calculated. All these tests were performed on a real Flash storage, the OpenSSD Jasmine board A.1. The relatively low values for transaction throughput across all scenarios are due to the limited I/O parallelism

of OpenSSD and I/O-bound workload.

	TPC-C		
	Cooked SSD	Raw SSD	NoFTL basic
<b>Tpm-C (Tx. per min)</b>	110	251	422
<b>Tx. Completed</b>	29,940	67,577	113,151
<b>DBMS Reads (16KB)</b>	821,056	1,883,277	3,236,547
<b>DBMS Writes (16KB)</b>	215,491	489,834	795,378
<b>File System Reads</b>	840,302		
<b>File System Writes</b>	917,153		
<b>GC Page Migrations</b>	2,321,498	2,264,782	1,332,795
<b>... per DBMS Write</b>	10.77	4.62	1.68
<b>GC Erases</b>	25,641	21,588	16,628
<b>... per DBMS Write</b>	0.119	0.044	0.021
<b>Write Amplification</b>	15.03	5.62	2.68

Table 5.1.: TPC-C benchmark under different storage alternatives on OpenSSD Jasmine board.

**File System Overhead** The I/O overhead caused by the ext4 file system becomes evident by comparing the numbers for the first two scenarios [*Cooked SSD*] and [*RAW SSD*] in Tables 5.1, 5.2. In an ext4-based DBMS storage the garbage collector performs 2.7x (TPC-C) and 2x (TPC-B) more erases per DBMS write and approx. twice as many page migrations. The increase of GC overhead is caused by two main factors. First, the file system issues additional write I/Os (e.g. metadata, journaling, etc.) resulting in higher write-amplification. Thus, under TPC-C the ext4 write-amplification is about 4.2 (similar results but for ext3 were reported in [59]). Second, the update pattern becomes more random, which causes the clear dominance of *full merges* over *partial merges* in FASTER FTL. For instance, on Raw SSD storage the fraction of full merges for TPC-C is about 55%, while on ext4 cooked storage it raises to more than 90%.

**NoFTL versus FTL-based SSD** Basic NoFTL without regions (*NoFTL basic*) outperforms the FTL-based SSD used as raw storage (*RAW SSD*) delivering 68% higher transactional throughput under TPC-C, and more than 2x higher throughput under TPC-B. Moreover,

	TPC-B		
	Cooked SSD	Raw SSD	NoFTL basic
TPS (Tx. per sec)	18.9	49.0	101.8
Tx. Completed	135,813	352,918	733,055
DBMS Reads (16KB)	1,311,241	873,485	1,566,834
DBMS Writes (16KB)	159,388	351,919	757,797
File System Reads	1,324,520		
File System Writes	678,365		
GC Page Migrations	1,962,340	2,460,320	775,865
... per DBMS Write	12.31	6.99	1.02
GC Erases	20,714	21,939	11,978
... per DBMS Write	0.130	0.062	0.016
Write Amplification	16.57	7.99	2.02

Table 5.2.: TPC-B benchmark under different storage alternatives on OpenSSD Jasmine board.

the garbage collection under NoFTL performs 2.75x less page-migrations per host write resulting in 2x lower write-amplification under TPC-C. For TPC-B benchmark the reduction in GC overhead is even more drastic: 6.8x less page-migrations and 4x lower write-amplification. The FTL-based SSD without file system (*RAW SSD*) performs also 2.1x (TPC-C) and 3.9x (TPC-B) more erase operations per host write than (*NoFTL basic*), which negatively impacts the longevity of the Flash SSD.

The main cause for the I/O overhead of the FTL-based SSD (*RAW SSD*) are the full merges under FASTer, which are also typical for all hybrid FTLs. In hybrid FTLs like FASTer the larger part of Flash memory is mapped at block-level granularity (data block area), while only a small part (log-block area) uses page-level address translation. Throughout our experiments the size of the log block area was set to 10% of the total storage capacity. All updates and write requests are first performed in the log-block area, and as soon as free space in that region runs out those updates are merged with the corresponding blocks in the data block area. The more random the workload, the higher the GC overhead and the write-amplification. Each full merge requires at least two erase operations and many page-migrations.

However, the I/O overhead in [*RAW SSD*] is only partly due to hybrid FTL schemes

like FASTER alone. Also FTL schemes based on the page-level mapping like DFTL [28] have higher write-amplification compared to the NoFTL. In DFTL additional I/O load is caused by fetching and eviction of mapping pages to/from the mapping cache. We have recorded TPC-C and TPC-B traces in Shore-MT under NoFTL and replayed them on the DFTL simulator<sup>3</sup> (mapping cache size is 15% of all mappings). The results in Table 5.3 show that under NoFTL the GC performs about 3x less page-migrations and 2.3x less erases. Moreover, NoFTL is free from the regular fetching and eviction of mapping pages. In NoFTL the complete mapping table (or tables in case of regions) is loaded during the device initialization, whereas to write-out modified mapping entries we piggyback the regular page writes (see Chapter 5).

	TPC-C		TPC-B	
	DFTL	NoFTL	DFTL	NoFTL
<b>Trace Reads</b>	15,609,311		17,384,094	
<b>Trace Writes</b>	1,379,696		1,334,973	
<b>Mapping Table Reads</b>	2,123,957	-	2,725,963	-
<b>Mapping Table Writes</b>	850,258	-	918,259	-
<b>GC Page Migrations</b>	3,888,166	1,248,777	3,189,524	1,012,418
<b>GC Erases</b>	95,596	41,071	85,043	36,678
<b>Write Amplification</b>	4.43	1.91	4.08	1.76

Table 5.3.: DFTL-based SSD versus NoFTL under TPC-C/B benchmarks.

**Buffer Manager** The following experiments are designed to analyze the optimization of background writer processes of the DBMS buffer manager. The tests have been performed on the Flash emulator under TPC-C and TPC-B benchmarks varying the number of Flash chips 2-32 [34]. The average results for the NoFTL with the traditional strategy and the NoFTL with the Flash-aware strategy of db-writers are presented in Figures 5.2, 5.3.

The Flash-aware strategy of db-writers improves the throughput by up to 1.5x under TPC-C benchmark. With an increasing amount of Flash parallelism and more db-writers leveraging the parallelism, the difference in the transactional throughput increases. Under the standard approach the response time for each single db-writer increases, due to the higher contention on Flash chips.

<sup>3</sup><http://csl.cse.psu.edu/a-simulator-for-various-ftl-schemes/>

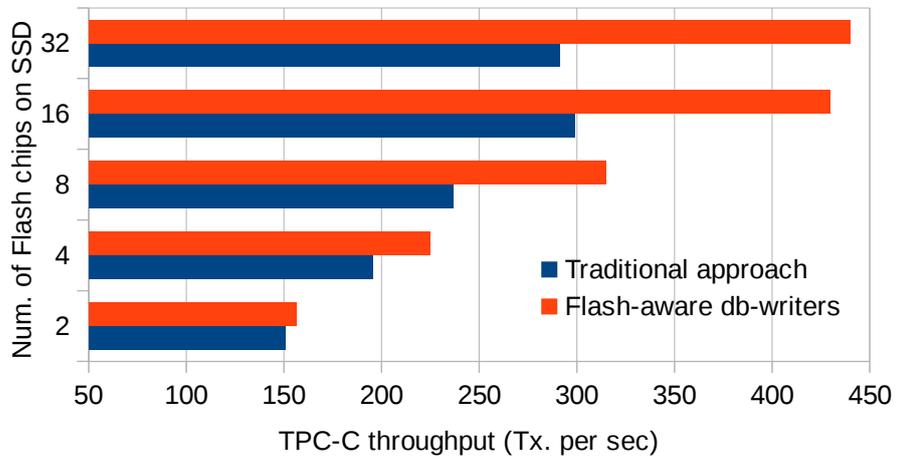


Figure 5.2.: NoFTL with traditional and Flash-aware strategies for background writes under TPC-C benchmark.

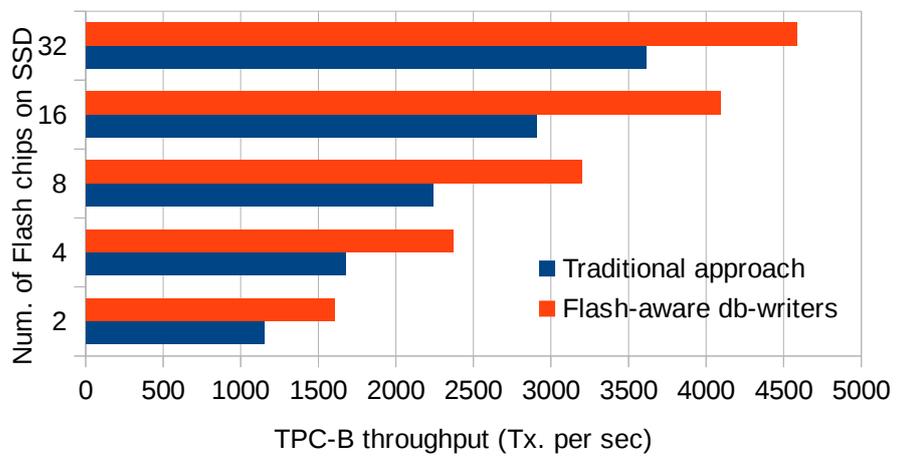


Figure 5.3.: NoFTL with traditional and Flash-aware strategies for background writes under TPC-B benchmark.



---

## 6. Configurable Flash storage

---

This chapter is dedicated to the third pillar of the NoFTL architecture - the concept of configurable storage. Its realization becomes possible only with the combination of the other two NoFTL components, the native Flash interface and integration of Flash management into the DBMS. By giving the DBMS a direct control over the Flash memory we can answer two important questions: (1) how to place data efficiently on raw Flash memory, and (2) how to manage Flash storage efficiently considering data placement. The emphasis on *efficiency* becomes obvious by looking at the solutions applied in modern Flash storage.

**Data Placement on Modern SSDs** In contrast to magnetic drives, Flash SSDs have a complex hierarchical memory organization, which comprises data channels, chips, dies, planes and blocks (see Chapter 2.3.1). Those units are responsible for different levels of I/O parallelism, which in turn give the SSD very high performance potential. However, the black box design of modern SSDs does not utilize this potential efficiently. Due to the lack of information about the data stored on the SSD, FTL simply distributes all the data evenly across the whole physical address space. Although this naive data placement strategy achieves ideal load balancing over all memory units of the SSD, it simultaneously results in a huge I/O overhead produced by the garbage collection of FTL.

To illustrate this problem let us consider an example. The most common naive data placement strategy applied in SSDs is based on a simple even distribution of logical addresses. For instance, if the Flash SSD consists of 32 dies (16 dual-die chips), the formula to determine the target die for the new data might be as simple as  $die\ index = logical\ page\ address \% 32$ . Within the die the new page is then typically written into the *current write block*. Under this approach every logical address is always assigned to a specific physical Flash die. Within the die, however, its location is mutable and might be changed by garbage collection, wear-leveling or simply by a new update to a page. It is clear that under this strategy the whole data of the database (and even of the whole system, i.e., from various applications) is mixed physically on the Flash storage. Therefore, an average Flash block contains pages with different access properties. Frequently modified data (hot data) is mixed with cold, read-only data. Although, the concrete "thermal image"

of an average Flash block strongly depends on the workload and a particular FTL scheme, we might assume that for write-intensive OLTP workloads the portion of cold data within a block is typically higher than that of hot data. This assumption can be reasoned as follows. The common 80/20 rule for OLTP workloads says that 80% of requests touch 20% of data, i.e. only 20% of the whole database data is hot. A typical wear-leveling and garbage collection strategy of FTL would ensure that this hot data is roughly evenly distributed over the whole address space. Assume that the Flash block depicted on the left in Figure 6.1 is selected by the garbage collection as a victim block (one with large number of invalid pages) to be reclaimed next. Since pages containing hot data are modified frequently, with high probability they will dominate among invalid pages. Pages containing cold data are rarely updated and thus mainly remain valid in such victim blocks. To reclaim this block the garbage collector must first copy all valid pages to a new empty block, which in our example results in 37 page migrations. Afterwards, the victim block is erased and returned to the pool of empty blocks. These page migrations make the main part of the I/O overhead produced by FTL, and thus they are one of the key factors determining the performance of the whole SSD.

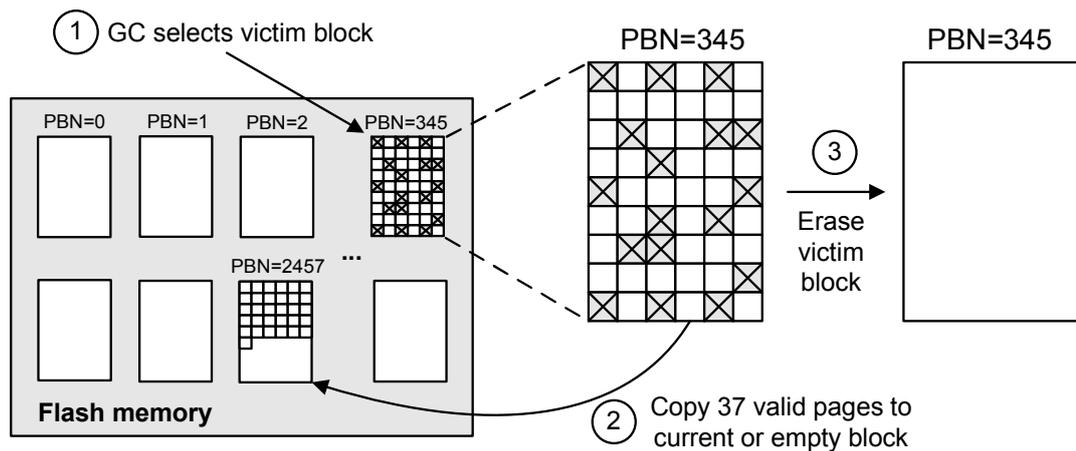


Figure 6.1.: Example of reclaiming a victim block by garbage collection.

The main technique to reduce the GC overhead is known as *hot/cold data separation*. Its basic idea is to avoid that data with different update frequencies is mixed physically on Flash storage. If we can achieve that data on an average Flash block has roughly the same temperature (e.g., only pages storing hot data), then upon selecting such a block as a victim by GC with high probability it will contain only few valid pages, and thus only few page migrations would be needed in order to reclaim that block. In recent

---

---

years researchers have proposed multiple approaches attempting to improve the hot/cold data separation on Flash SSDs [89], [56], [106]. Although, their results show significant reduction of GC overhead and thereby improvement of an SSD's performance, often their practical application in modern SSDs is difficult. Some of those approaches (e.g., [89]) require the FTL to maintain additional statistics regarding the update frequencies of stored data. However, memory footprint and computational overhead of keeping those statistics are usually critical for the limited on-device resources of Flash SSDs. The efficiency of other approaches is typically highly workload dependent.

One example of applying an explicit hot/cold data separation policy by an industrial product is the Multi-Stream Write SSD from Samsung [17], [81]. This SSD allows the application to provide a hint to the SSD regarding the temperature of written data. Thus, by specifying a certain stream number in write I/O an application can tell FTL that this data has different access characteristics than data submitted to other streams. Although we cannot know the implementation details of Multi-Stream SSD, nor were we able to perform experiments with this product, we assume that this approach might have several bottlenecks. One of them is its application in dynamic workloads, where the access properties of data change with time. Furthermore, although this approach addresses one important problem (efficient data placement through hot/cold data separation), it does not solve other problems of modern SSDs, which are caused by their black-box architecture.

The main problem of the proposed approaches for hot/cold data separation is rooted in the fact that they are targeting traditional FTL-based SSDs. The black-box design of SSDs separates and hides the knowledge of the DBMS about data properties, which is required for efficient hot/cold data separation, from the control over the physical data placement, which is done exclusively by the FTL. Returning this control to the database system under NoFTL allows to realize hot/cold data separation to its full extent and minimize thereby the overhead of garbage collection.

**Flash Management on Modern SSDs** As mentioned in Chapter 2.3.2 FTL consists of multiple Flash management tasks, such as address translation, garbage collection, wear-leveling, error-correction and bad-block management. Since performance of the whole SSD depends directly on the efficiency of Flash management, research and industry have invested a large effort into analysis and development of efficient algorithms for FTL. This evolution has provided for each of the Flash management tasks multiple implementation alternatives. However, there is no ultimate winner. Each FTL scheme has its advantages and disadvantages in a concrete environment. This environment is a combination of the workload, available hardware resources of the Flash SSD and requirements on I/O

---

characteristics of the storage. An overview of the available FTL schemes is provided in Chapter 2.3.2, while the research analysis of the FTL performance under different workloads can be found in [88].

SSD manufacturers are facing a difficult problem when selecting an FTL scheme for their product. With no prior knowledge about the system and the workload in which their SSD will be used, they need to choose an FTL that would perform reasonably well under the whole range of workloads. The performance of such a generic FTL scheme under a certain workload is usually significantly lower than the performance of the optimal FTL for a particular case. That difference might easily go beyond an order of magnitude (see [88] for FTL comparison under different workloads). To compensate these performance gaps SSD designers typically increase the size of an SSD's write-cache (dedicating a lion's share of the on-device DRAM for it) and use larger values for over-provisioning (e.g., up to 40% in enterprise devices), which dampens the influence of the GC overhead on the throughput. However, as soon as these "buffers" become full (e.g., under workloads with long write-intensive phases) the SSD's performance decreases. The result is unstable performance, the need for large capacitors to protect the write-cache in case of power failure, and inefficient utilization of available Flash memory due to higher over-provisioning.

The NoFTL architecture allows us to realize new approaches of efficient data placement and Flash management. Our solution concept, configurable storage, is based on two new storage structures: *regions* and *groups*.

## 6.1. NoFTL Regions

Region is a set of Flash chips that is constant in size but can change in its composition. Regions are means to perform data placement with respect to hot/cold data separation, and also they are means to utilize selective Flash management. Regions are coupled to the existing DBMS logical structure *tablespaces* (sometimes known as segments, e.g., in Oracle), which reduces the complexity overhead of the new structure to a minimum.

### 6.1.1. Data Placement with Regions

Every database object (e.g., table, index, or a partition of those) is assigned to one region, while every region might keep one or many database objects. Assigning an object to a region means that all data belonging to that object is stored within Flash chips composing that region. Assignment of all database objects to the set of regions is called *configuration*.

Many large productive databases consist of hundreds and thousands of database objects. Assignment of each of those to a separate "individual" region would not be efficient and

often even not feasible due to an alignment of regions to Flash chips (modern Flash SSDs have only about a dozen of chips). Instead, typically, database objects with similar access patterns are assigned to a single common region. Thus, regions allow us to separate data with different update temperature physically from each other (see Figure 6.2). As a result, the overhead of garbage collection in each particular region, as well as the overall overhead are reduced. Depending on the database and workload characteristics (e.g., heterogeneity of database objects) this reduction can vary from a few percent to several times in terms of write-amplification, number of erases and I/O throughput (see Chapter 6.3).

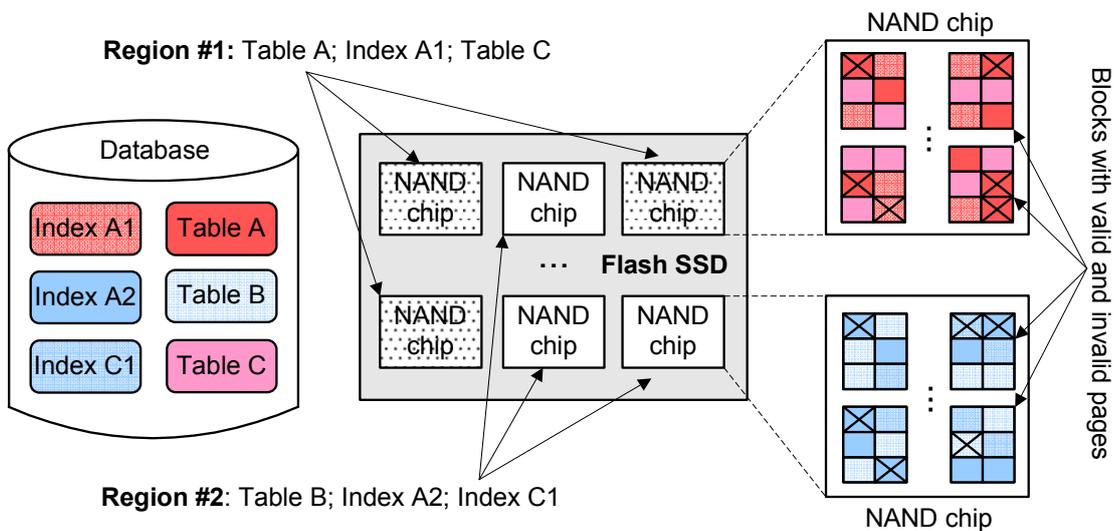


Figure 6.2.: Hot/cold data separation using NoFTL regions.

Another important function of regions is better utilization of available I/O parallelism of Flash storage. Naive data placement, applied by modern SSDs, where all data is distributed evenly over the whole physical address space, results in a perfect load balancing among all independent units of Flash memory. Since, every database object is uniformly striped over all available Flash chips, it can be accessed by using the maximal level of parallelism. However, the inevitable disadvantage of naive data placement, increased overhead of garbage collection, significantly prevails over the positive effect of load balancing.

We will look at parallelism in case of intelligent data placement by means of regions. At first sight, placing data objects within a certain set of Flash chips, which compose the corresponding region, limits the level of parallelism with which data might be accessed,

as compared to distributing those objects evenly over all available chips. However, besides significant reduction of GC overhead achieved through proper hot/cold data separation, regions allow us to control the utilization of available Flash parallelism. This is done *on-demand*. Database objects that demand higher level of I/O parallelism (hot objects) should be assigned to regions with higher number of chips (see Figure 6.3). Cold objects, that are accessed seldomly, might be assigned to regions with a number of chips that is defined based on tightly estimated space required for those objects. Together, reduced overhead of the garbage collection, reduced contention for physical resources (Flash chips) between different database objects, and on-demand distribution of available Flash parallelism result, for a particular region with limited set of Flash chips, in significantly better I/O throughput as compared to the throughput for the same set of database objects evenly distributed over the whole SSD (see Chapter 6.3).

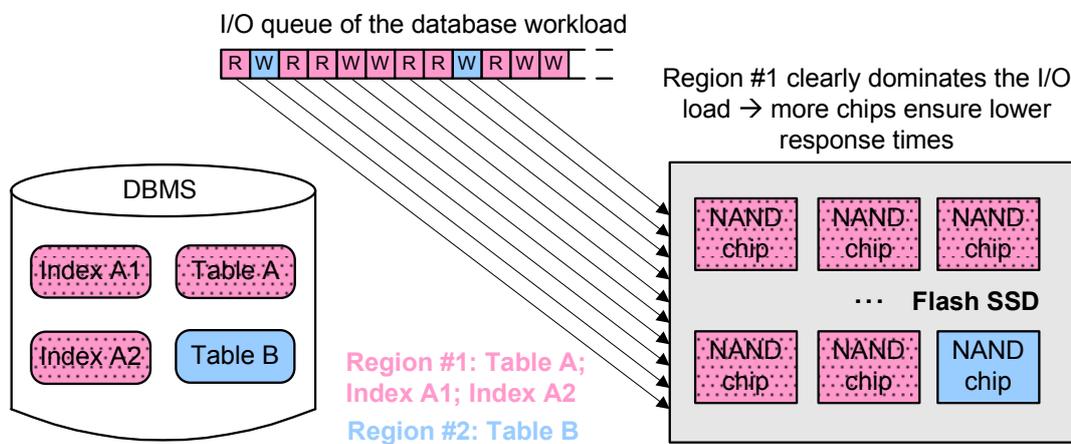


Figure 6.3.: Control of SSD parallelism using NoFTL regions.

Another aspect regarding the intelligent data placement is that within a region data can be distributed using different striping sizes. Depending on the properties of database objects assigned to a certain region (e.g., temperature of objects, type of access pattern) striping size might be selected to be the size of a Flash page, or size of a database page, or any other size which is a multiple of a Flash page. An evaluation analysis of striping on Flash storage can be found in [88]. In brief, hot database objects with rather random access pattern would benefit from smaller stripe sizes, i.e. size of a single Flash page. Objects with skewed access pattern should utilize rather large striping sizes, as this would lower the overhead of garbage collection. See more on implementation details regarding the striping size in the following Chapter 6.1.2.

---

Details about how to select the proper data placement configuration, what statistical data to use for this, as well as how this process can be automated are discussed in Chapters 6.1.2 and 8.

### 6.1.2. Region-Specific Flash Management

There are dozens of different Flash Management schemes (FMS)<sup>1</sup> proposed in recent years. However, our NoFTL prototype and experiments have shown that it is typically enough to be able to choose among only a few of them to achieve better performance results, as compared to *one size fits all* approach used by the FTL. For instance, in our NoFTL prototype we have integrated into the DBMS four different FMS: pure block-level, pure page-level, DFTL [28], and FASTer [56] schemes. Thus, all major types of FMS are present (see Chapter 2.3.2), which allows us to cover the majority of different workloads.

We realized that selecting a *single* Flash management scheme is not sufficient. Although every workload is characterized by its dominant objects and their access patterns, at the same time, it is very diverse and consists of hundreds to thousands of different objects, which typically might be grouped based on their access patterns into dozens of groups. Thus, considering the workload as a whole and choosing only one FMS based on its dominant characteristics is still suboptimal. Therefore, we propose under the NoFTL architecture to select the most appropriate FMS for each *region* individually. Thus, a NoFTL region becomes not only a means of performing data placement and controlling available Flash parallelism, but is also a unit for applying *selective Flash management*. Fortunately, the criteria for object clustering and assignation to regions are same for both effective data placement and optimal Flash management. In both cases database objects are grouped based on their access characteristics, such as, read/write ratio, locality of accesses, access frequency and access pattern (sequential, random, append-based), level of required parallelism, *role* for the workload (e.g., logs vs. background/lazy writes), etc. This information is typically either already available in the statistics and metadata of the DBMS, or can be easily collected.

In the following, we describe the major aspects and algorithms of Flash management that might be configured individually for every region.

---

<sup>1</sup>We use the term *Flash management scheme (FMS)* because usage of term *FTL scheme* under NoFTL architecture is incorrect. FMS is a general term, which means a set of Flash management algorithms (address translation, garbage collection, wear-leveling, bad-block management), while FTL scheme is a special case of FMS. FTL scheme assumes that Flash management algorithms are implemented as an isolated (black-box), intermediate layer between Flash memory and system/application. In contrast, FMS used in NoFTL architecture, assumes deep integration of the same Flash management algorithms into the different subsystems of the DBMS (see more in Chapter 5).

- 
- **Address translation.** Translation of logical addresses (LBAs) to physical addresses on Flash memory is the key task of every FMS. The address translation scheme is also decisive for other FMS algorithms, such as garbage collection and wear-leveling. As described in Chapter 2.3.2, there are basically three main types of address translation applied in Flash SSDs: page-level, block-level and hybrid. Each of those has, in turn, many variations, which under a certain workload might be more appropriate than others due to reduced GC overhead or memory footprint. NoFTL allows us to use the type of address translation that is most suitable for objects assigned to that region. Thus, for instance, a region containing hot, write-intensive database objects with rather random access pattern (e.g., B-Tree indexes) is typically a good candidate to use pure page-level address mapping. Hot, write-intensive data characterized by high level of access locality can use the mapping scheme proposed in a DFTL [28] approach, which would provide the same level of placement flexibility as pure page-level mapping, but will save on DRAM memory needed to cache mapping entries. Hybrid mapping schemes (such as FASTer [56]) are generally a good choice for database objects for which large sequential access dominates (e.g., database objects storing LOBs, where every insert or modification causes multiple logically sequential pages to be updated). Especially interesting are the use cases for block-level mapping under NoFTL. Typically, pure block-level mapping is not even considered as a choice for modern Flash SSDs. The reason is that for random write accesses (which are unavoidable in almost any workload) it produces the highest overhead of garbage collection. However, selective Flash management enables the use of block-level mapping in an advantageous manner, i.e., for regions where this issue with GC overhead does not come into play but the whole system can benefit from the least memory footprint needed to store address mappings. Typical examples for application of block-level mappings are: (i) read-only and read-mostly data; (ii) data that is modified in an append-based manner (e.g., tables storing historical data, objects that are modified under the MVCC policy, objects in log-based storage engines, etc.); (iii) data that is modified in multiples of Flash block sizes (e.g., tables storing LOB data with the fixed-size records of multiples of Flash block size). Even objects with a high rate of random modifications can be effectively maintained with block-level mapping if the total size of modified data per Flash page is small (e.g., data where only one numeric, enumerate or boolean attribute changed per record). In this case block-level mapping is applied in combination with the IPA approach (see Chapter 7).

Although due to access to memory resources of the host system, NoFTL makes it possible to apply the most flexible pure page-level mapping for the whole Flash SSD,

---

in practice this is not necessary. Address translation applied selectively for each region makes it possible to achieve performance which is comparable to (and in some cases even better than) the pure page-level mapping for the whole storage. At the same time, the overall memory consumption by address mappings of all regions becomes comparable to the consumption of a single hybrid mapping scheme used in modern SSDs. This is achieved since typically regions with higher memory consumption (e.g., page-level mapping) are compensated by regions with lower memory footprint (e.g., block-level mapping)).

The selective approach allows us not only to choose the most appropriate address translation scheme per region, but also to configure the selected scheme optimally. Many translation schemes have configurable parameters that have direct influence on their efficiency. For instance, main parameters for the DFTL scheme are the sizes of two cache buffers that determine the maximal number of mapping entries stored in DRAM. Depending on the size of the working set, the sizes of the two caches might have to be increased in order to avoid additional I/O operations needed to fetch mapping entries from Flash. Similarly, the level of request locality is important to correctly adjust the ratio between both caches. For hybrid mapping schemes, like FASTer, important parameters are the size of over-provisioning (size of log-block area, which uses page-level mapping), as well as the size of the isolation area.

- **Garbage collection and wear-leveling.** Although garbage collection and wear-leveling are defined by the selection of address translation, still some behavioral aspects and parameters of those algorithms can be adjusted based on specific properties of database objects assigned to that region. For instance, garbage collection can be configured to be triggered only on-demand (i.e., when the remaining free space is less than a threshold), or work "proactively", when GC is triggered in constant time intervals or depending on the amount of write requests submitted to that region. Configurable is also the amount of work that GC should perform on every start. For the wear-leveling strategy the most important configuration parameters are: (i) its type (static, dynamic or mixed); (ii) the area of responsibility (single chip or region as a whole); (iii) type of execution (coupled to GC function for selecting a victim block or as a separate process).

Furthermore, if wear-leveling is configured to work as a separate process it can have different ways to be triggered. For instance, it can be started by a monitoring process when a predefined threshold for the difference between minimum and maximum erase counts is achieved. NoFTL allows for another, interesting alternative for triggering wear-leveling and garbage collection. As NoFTL gives the database

---

system the whole responsibility to perform Flash management, the DBMS can choose the most appropriate time to trigger these functions (see Chapter 5). Thus, during heavy I/O loads regions might use a "light" version of wear-leveling, while during the times when the intensity of the workload becomes minimal the DBMS can explicitly trigger global, static WL (see Chapter 8). A good candidate for such time-scheduled WL and GC is a database system, where writes to storage are performed periodically. For instance, although productive systems typically run in a 24/7 mode, in many cases there are daily periods when the intensity of workload becomes minimal, e.g., for online shopping platforms it would be night hours if most accesses exhibit geographic locality.

The potential issue of uneven wear between regions due to data separation is solved by the utilization of one important property of NoFTL regions, namely, the set of physical Flash units (e.g., dies) comprising a region is dynamic. Thus, special wear-leveling policies in NoFTL ensure even wear-out of Flash dies across all regions. The switch of dies from one region to another (i) does not require additional space reservation; (ii) is performed in the background, internally on-device; and (iii) is highly configurable by the DBMS, e.g., depending on the current I/O load it might be configured trading transition speed versus additional I/O overhead. More details on this global wear-leveling strategy are presented in Chapter 8.

- **NAND mode.** Nowadays, there is Flash memory with either single-level (SLC), multi-level (MLC) or triple-level cells (TLC), which can store 1, 2 or 3 bits per cell, respectively. While more bits per physical cell increase the capacity of the drive, the performance characteristics and longevity deteriorate significantly. Due to the increasing demand for high-volume Flash SSDs, there is a trend towards MLC or TLC Flash. It is noteworthy that MLC can be dynamically configured in a pseudo SLC mode (pSLC), while TLC can be configured in both MLC and SLC modes. In such cases the performance and endurance characteristics would be similar to real SLC (MLC), but the capacity is reduced accordingly. Current research focuses on maintaining a small partition of MLC or TLC in pseudo SLC mode, to accumulate pending update requests [40], [41], [103]. NoFTL utilizes the DBMS run-time information and statistics to directly control data placement, and builds upon the above property. Depending on the access patterns of different database objects, the DBMS under NoFTL might use the most appropriate NAND cell mode for the corresponding logical regions. The pSLC mode might be used for frequently accessed (*hot*) database objects, due to the best write/read performance. MLC or TLC modes would be more appropriate for *colder* less frequently accessed objects. The NoFTL transitions between different modes can be performed dynamically.

- 
- **Update strategy.** It is a well known fact that Flash memory follows the so-called erase-before-overwrite principle, which requires an erase operation to be performed on a Flash block before pages within it can be overwritten. The common approach to mitigate these erases on every single update, and thus reduce the overhead, is to utilize an out-of-place update strategy, where the updated data is always written to a new, free location on Flash, while a separate garbage collection process cares about space reclamation occupied by outdated versions of data. However, it is a commonly ignored fact that under certain conditions, Flash memory does support in-place modification of written pages without preceding erase operation. The obscurity of this property is not surprising, since the traditional, black-box FTL-based design of SSDs makes the use of this approach almost impossible. In contrast, NoFTL allows the DBMS to effectively utilize it, and gain in that way significant decrease in the garbage collection overhead, which further results in the increased performance and longevity of Flash storage. The detailed description of this approach - in-place appends (IPA) - is provided in Chapter 7. It is important that the DBMS can apply IPA selectively, i.e., only to regions, which contain database objects with appropriate update behavior (the dominant part of updates change only a small portion of a page's data). Moreover, the choice of applying IPA or not, and if so, what IPA mode, type and configuration would be appropriate, can be well automated via the IPA-advisor (see Chapter 8), thus introducing negligible additional overhead for the database administrator.

Consider an example of the DDL statement shown below, which defines a region with certain configuration parameters. Here, the region with the name *rgIndex* comprises 8 NAND chips, which are used in MLC mode, and managed with the page-level address translation scheme and *lazy* variant of garbage collection. Furthermore, the region utilizes the IPA approach for B-Tree indexes (see Chapter 7.3.1).

```
CREATE REGION rgIndex (  
    NUM_CHIPS=8, NAND_MODE=MLC, ADDR_MAPPING=PAGE, GC=LAZY,  
    IPA_MODE=odd-MLC, IPA_TYPE=IPA-IDX, IPA_CONF=2x270);
```

Regions can be naturally coupled to the established logical storage structure tablespace (Figure 6.4). Thus, by assigning a tablespace to a certain region, all database objects that belong to that tablespace will be physically stored in Flash chips of the region. In other words, regions give tablespaces the ability to control physical data placement and Flash management for enclosed data.

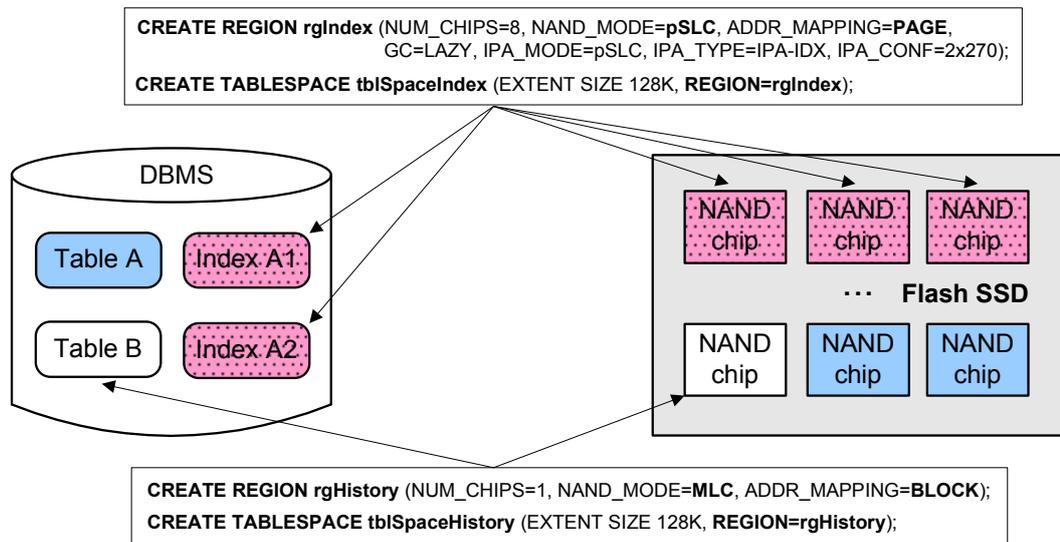


Figure 6.4.: Selective Flash management with NoFTL Regions

## 6.2. NoFTL Groups

As we have already seen, *regions* are means of (i) performing data placement, (ii) applying selective Flash management, and (iii) controlling available I/O parallelism. Regions store data with similar access properties, which ensures proper hot/cold data separation and allows to select the most appropriate set of Flash management algorithms for each particular region. However, typically, database objects assigned to one region are not completely homogeneous, and still differ in their properties. Especially in large systems with hundreds and thousands of database objects the diversity between objects in a region might be significant. One of the most important properties for Flash management is the update frequency of data. The ability to physically separate data with different update temperatures is the key factor in minimizing the overhead of garbage collection and prolonging the lifetime of an SSD. Although *regions* help to separate database objects with different classes of access temperatures (e.g., hot and cold objects), they are typically not suitable to provide more fine-grained separation.

In order to further improve hot/cold data separation we introduce another storage structure - *group*. A group is a mutual subset of database objects within a certain region. The number of groups in each region is individual and varies from 1 to  $N$ , where  $N$  is the number of objects in that region. Data that belongs to different groups is not mixed in any

particular Flash block. In other words, every Flash block contains pages from only one group (see Figure 6.5). Note, that in contrast to regions, groups are not used for selective Flash management, i.e., all region-specific settings (e.g., address translation, garbage collection, wear-leveling, NAND mode, etc.) are equal for all groups within that region.

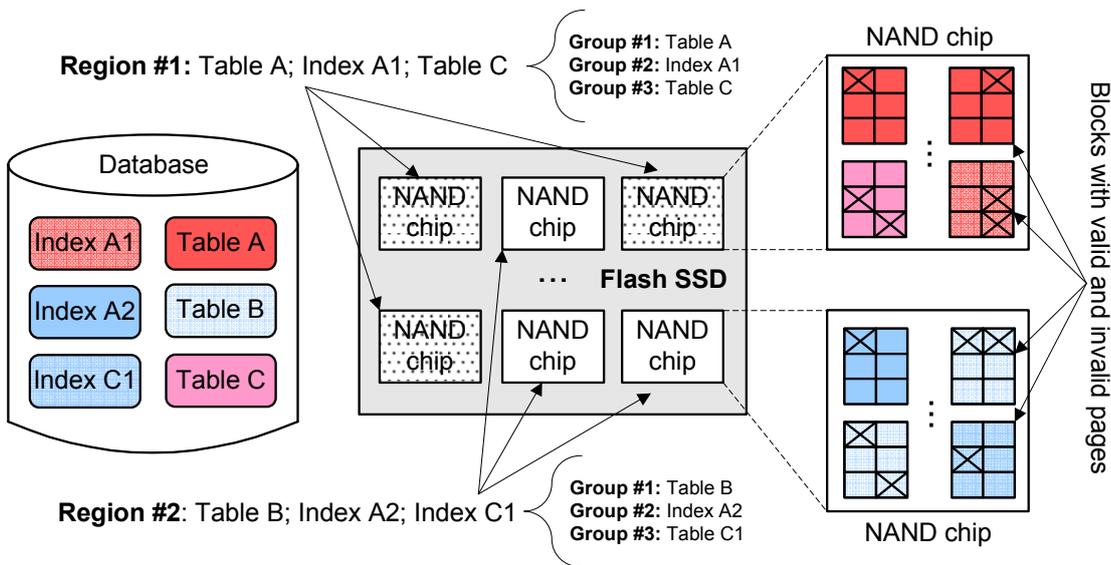


Figure 6.5.: Hot/cold data separation using NoFTL groups

Thus, by *clustering* database objects within one region into different groups based on their update frequencies, we ensure that every arbitrary Flash block in that region would have roughly homogeneous temperature of its data. As a result, the average amount of valid pages within victim blocks selected by GC would be significantly lower, as compared to blocks containing heterogeneous data. This, in turn, will reduce write amplification (less page migrations), and slow-down the wear-out of Flash storage (less erase operations).

Application of groups does not produce neither management nor computational overheads. The memory overhead is typically negligibly small. It is enough for each independent unit of Flash memory (chip/die) to keep track only of the currently written Flash block for every group. For instance, if we assume a Flash SSD with 32 dual-die chips, and that on average every region has 10 groups, then the overall memory footprint of applying groups would be less than 3KB of memory. A simple one-dimensional array with one element per group would be a sufficient data structure to hold information about current Flash blocks for all groups, e.g., group identifiers are indexes in that array, and

---

block numbers are elements. An example of DDL statements defining database objects belonging to a certain group in a region can look as follows:

```
CREATE REGION rgPage (  
    NUM_CHIPS=8, NAND_MODE=MLC, ADDR_MAPPING=PAGE, GC=LAZY,  
    IPA_MODE=odd-MLC, IPA_TYPE=IPA-IDX, IPA_CONF=2x270);  
  
CREATE TABLESPACE tblPage (  
    REGION=rgPage, EXTENT SIZE 128K);  
  
CREATE TABLE A ( t_id NUMBER(3) )  
    TABLESPACE tblPage ( GROUP #1 );  
  
CREATE INDEX A1 ON A( t_id )  
    TABLESPACE tblPage ( GROUP #2 );
```

There is no pre-assignment of Flash blocks to groups. Whenever a region gets a write request, it checks to which group the incoming data belongs. This is done via a quick lookup in a small hash table, which maps database objects to groups. If the current Flash block (array access) in the corresponding chip/die<sup>2</sup> has enough space, the data is written there. Otherwise, an empty Flash block is taken, while the corresponding element of the array is updated with the number of the new Flash block. Blocks reclaimed by garbage collection can be used for data of any group in that region. As the blocks can be re-used for any group, wear-leveling strategy does not require any additional techniques to ensure even wear-out across one particular chip/die, as well as across a whole region.

The idea of *groups* is quite similar to the established and common concept of *extents* applied in database systems. Extents are equally sized chunks of data (e.g., 8 database pages) with sequential logical addresses. For instance, database pages with logical page numbers from 8 to 15 are stored in their logical order in one extent (e.g., extent number 47), while the next extent (#48) might contain db-pages with numbers 800 - 815. A common practice is to use extents as a growth unit for database objects. Whenever a new db-page should be appended to a certain object, the DBMS checks if there is a free space in the last extent used for that object, if so, the page is written there, otherwise a new extent is allocated, e.g., 8 continuous pages are reserved for that db-object. Thus, within every extent all pages belong to the same database object, however, extents comprising one database object are not necessarily sequential. That means, that each database object

---

<sup>2</sup>The target chip/die for new data is identified based on data striping technique within a particular region.

---

---

can be seen as a set of extents (e.g., Table #1 consists of extents with numbers 34, 56, 84, 232, ...).

Extents are used by database systems for multiple purposes. Their primary goal is to improve I/O performance of storage systems based on spinning drives. As the data within extents are logically contiguous it can be accessed with sequential I/Os, which can be up to 100x faster than random accesses to disk storage. For instance, assume a small database object that consists of 1000 extents, and each extent contains 8 db-pages. A scan operation over that object would in the worst case (extents are randomly distributed) result in 1000 random accesses, while without extents the number of random I/Os can be as much as 8000. However, this advantage of extents is often lowered and even completely brought to naught in modern systems. This is because of software and hardware layers between the DBMS and physical storage, which introduce additional address indirection. For instance, if the DBMS uses HDDs as secondary storage, file systems typically provide their own address translation, which might sometimes turn sequential I/Os (within extents) submitted by the DBMS into random accesses on disk. In cases when the storage is based on modern Flash SSDs, there is no advantage of extents with regards to I/O pattern at all. FTL and an out-of-place update strategy will turn all incoming sequential accesses into random requests on Flash memory.

### 6.3. Evaluation Results

The concept of configurable Flash storage based on utilization of novel storage structures *regions* and *groups* was realized in an open-source database engine Shore-MT [42]. The experimental evaluation is performed on real Flash hardware (the OpenSSD Jasmine Flash research platform [94]) and a data-driven Flash emulator [30].

The original implementation of Shore-MT organizes all database data in one file, and thus provides a single address space for all database objects. In order to introduce the concept of regions we had to implement the support for the concept of tablespaces, which is common in multi-file DBMSs. This allows to partition the logical address space and assign database objects to appropriate partitions. Consequently we coupled the novel storage structure *regions* to tablespaces.

The NoFTL architecture allows for flexible configuration of regions with the most appropriate Flash management algorithms. We integrated into the storage and free space management subsystems of the DBMS multiple popular variations of those algorithms. We provide address mapping with page and block granularities, as well as a hybrid one; multiple garbage collection and wear-leveling algorithms enable us to study their effects (see more in Chapter 5).

We extend the results presented in Chapter 5.6 in Tables 5.1 and 5.2 with new experiments for configurable Flash storage. Thus, Tables 6.1 and 6.2 present now the results for TPC-C and TPC-B benchmarks tested in three different scenarios: (i) *[NoFTL basic]* – NoFTL storage with a single region; (ii) *[NoFTL regions groups]* – configuration with multiple regions and groups for controlling data placement and Flash management; and (iii) *[NoFTL regions groups IPA]* – which enhances the previous configuration with the usage of IPA approach for certain regions. All these tests are performed on the real Flash memory - OpenSSD Jasmine board (see Chapter A.1).

	TPC-C		
	NoFTL <i>basic</i>	NoFTL <i>regions groups</i>	NoFTL <i>regions groups IPA</i>
<b>TPS</b>	15.6	25.1	25.1
<b>Tpm-C (Tx. per min)</b>	422	675	756
<b>Tx. Completed</b>	113,151	181,190	203,476
<b>DBMS Reads (16KB)</b>	3,236,547	5,189,878	5,937,220
<b>DBMS Writes (16KB)</b>	795,378	1,270,529	1,444,591
<b>GC Page Migrations</b>	1,332,795	397,776	219,970
<b>... per DBMS Write</b>	1.68	0.31	0.15
<b>GC Erases</b>	16,628	13,036	6,909
<b>... per DBMS Write</b>	0.021	0.010	0.005
<b>Write Amplification</b>	2.68	1.31	1.15

Table 6.1.: TPC-C benchmark under three different NoFTL configurations on OpenSSD Jasmine board.

**Regions & Groups, Data Placement and Parallelism** In this chapter we investigate the performance impact of novel storage structures *regions* and *groups*. We focus on the question: would an even distribution of all data over all Flash chips (regardless of the data properties) not be better in terms of I/O parallelism and wear-out? Based on the experimental results we show in this chapter, in most cases the use of regions significantly increases the I/O concurrency and almost doubles the longevity of the Flash SSD.

We compare the basic NoFTL scenario with a single region (columns *[NoFTL basic]* in

	TPC-B		
	NoFTL <i>basic</i>	NoFTL <i>regions groups</i>	NoFTL <i>regions groups IPA</i>
TPS (Tx. per sec)	101.8	124.8	151.4
Tx. Completed	733,055	898,243	1,090,467
DBMS Reads (16KB)	1,566,834	1,919,821	2,476,569
DBMS Writes (16KB)	757,797	927,590	1,085,280
GC Page Migrations	775,865	259,840	137,715
... per DBMS Write	1.02	0.28	0.13
GC Erases	11,978	9,271	4,635
... per DBMS Write	0.016	0.010	0.004
Write Amplification	2.0	1.3	1.1

Table 6.2.: TPC-B benchmark under three different NoFTL configurations on OpenSSD Jasmine board.

Tables 6.1, 6.2, and 6.3) to a scenario involving data placement according to multi-region, multi-grouping configurations (columns [*NoFTL regions groups*]). In both these scenarios we do not apply region specific Flash management, i.e., all regions utilize the same Flash management scheme (e.g., page-level address translation). Thus, we can evaluate the effect of smart data placement based on applying two main functions of regions and groups - hot/cold data separation and control of I/O parallelism. The evaluation of selective Flash management is done in Chapter 6.3.

The columns [*NoFTL basic*] and [*NoFTL regions groups*] in Tables 6.1, 6.2 show the corresponding performance results under TPC-C and TPC-B on OpenSSD, while Table 6.3 contains the results for TPC-C on the Flash emulator. On OpenSSD the use of regions and groups improves the transactional throughput by 60% for TPC-C, and by 22% for TPC-B. As OpenSSD Jasmine Board supports only very limited I/O parallelism, the achieved performance improvement is primary the result of hot/cold data separation. By clustering database objects with similar access properties the overhead of the garbage collection is reduced significantly. Thus, the average amount of page-migrations per single host write is reduced 5x under TPC-C, and 3.6x under TPC-B. The number of erase operation per host write, as well as overall write-amplification are decreased 2x under TPC-C, and by

about 37% under TPC-B benchmark.

Columns *[NoFTL regions groups IPA]* in Tables 6.1, 6.2 show the results under the same multi-region, multi-grouping configurations as used for tests in *[NoFTL regions groups]*, but in addition, one region was utilizing IPA. In TPC-C IPA was enabled for the region containing Stock table, while in TPC-B it was a region with Account table. IPA allows updating the original Flash page without a preceding erase operation. Consequently, the number of page invalidations decreases, and thus the garbage collector needs to perform approximately twice as few page migrations and erases (see more about IPA approach in Chapter 7).

	TPC-C		
	NoFTL basic	NoFTL regions	$\Delta$ [%]
<b>TPS (Tx. per sec)</b>	769.2	1,140.0	48.2
<b>Tx. Completed</b>	6,958,868	10,315,234	48.2
<b>DBMS Reads (16KB)</b>	210,025,055	312,005,523	48.6
<b>DBMS Writes (16KB)</b>	48,304,777	71,016,177	47.0
<b>GC Page Migrations</b>	49,423,253	29,798,578	-39.7
<b>... per DBMS Write</b>	1.02	0.42	-59.0
<b>GC Erases</b>	1,526,977	1,575,255	3.2
<b>... per DBMS Write</b>	0.0316	0.0222	-29.8
<b>Write Amplification</b>	2.0	1.4	-29.8

Table 6.3.: Comparison of NoFTL with and without regions under TPC-C on Flash emulator.

In addition to lower GC overhead, the experiments on the Flash emulator allow us to analyze the effect of regions on utilization of I/O parallelism and reduced contention for physical resources. In the next experiment (Table 6.3) we have emulated a storage device with 128GB of Flash memory comprising 64 dies. As in all experiments after the ramp-up phase, we run the TPC-C for two hours. Under scenario *[NoFTL regions]* with 4 regions (Table 6.4) we observe a 48% better transactional throughput than under *[NoFTL basic]*. The GC overhead is reduced by 59% in terms of page-migrations per host write and by approx. 30% for erases per host write and the total system write-amplification.

Storage Structures	Database objects	NAND chips	Address Mapping
Tablespace 1 Region 1	Warehouse, District, History, NewOrder, NewOrder-IDX, Order, Order-IDX, Order-Cust-IDX	6	Page-Level
Tablespace 2 Region 2	Warehouse-IDX, District-IDX, Item, Item-IDX, Stock-IDX, Customer-IDX, Cust-Name-IDX	5	Block-Level
Tablespace 3 Region 3	OrderLine, Stock	33	Page-Level
Tablespace 4 Region 4	Customer, OrderLine-IDX	20	Page-Level

Table 6.4.: Data placement configuration for TPC-C.

**Region-Specific Flash Management** This experiment investigates the performance impact of assigning a specific Flash management scheme to a NoFTL region, depending on the properties of db-objects placed in it. By doing so we relax the current SSD vendor strategy of employing a single set of Flash management algorithms for the whole storage device ("one size fits all").

NoFTL currently contains page-, block-level and hybrid (FASTer and DFTL) address mapping schemes. While page-level mapping (PLM) offers maximum flexibility for data placement, and is best choice for update-intensive data, it also has the largest memory footprint. Block-level mapping (BLM) consumes negligible amount of memory, has excellent performance for db-objects with read-only or append-based patterns, but is ill-suited for randomly updated db-objects, due to its high maintenance overhead. Hybrid schemes offer a resource and performance trade-off between PLM and BLM and are suitable for read-mostly db-objects with some degree of write-skew. Currently two hybrid schemes are implemented DFTL and FASTer: DFTL handles strong write-skew better, while FASTer excels with general I/O patterns (see more about Flash management schemes in Chapter 2.3.2).

Standard TPC-C is ill-suited for demonstrating the advantages of multi-FMS since the size and write-behavior of the STOCK table practically dominates the I/O behavior of the whole benchmark. Therefore, we extend TPC-C by increasing the size of the HISTORY

table, as a counterweight to the STOCK table with read-mostly and append-write I/O patterns. As a result TPC-C runs on a dataset that would have been produced running the original TPC-C transaction mix after running for two days instead of two hours.

Based on their I/O properties TPC-C db-objects have been placed in four regions as shown in Table 6.5. Tables like *Warehouse* or *District* are small and thus, typically fit completely in the DBMS buffer. This results in lower I/O activity making them being relatively cold compared to other db-objects (consider Table 6.5 "% Write" and "% Read"). Therefore, they were placed in a hybrid FMS region managed by FASTer. DB-objects like the *Stock* and *Customer* tables or the *o\_cust\_idx* index take the lion's I/O share with random updates, hence they are placed in a PLM-based region (*HOT*). Furthermore, these objects require a lot of parallelism, hence *HOT* comprises more chips/dies than the aggregate object sizes. The new *History* exhibits read-mostly and append-based I/O patterns, making it an ideal candidate for a region, which utilizes block-level address translation. Objects like *no\_idx* or *o\_idx*, which are primary key indexes in *NewOrder* and *Order* tables, exhibit relatively frequent updates (e.g., insertion of new records) and can be placed in a DFTL-managed region (*WARM*). The comparison of performance results for this configuration against different single-region configurations is provided in Table 6.6.

Region	Min. chips required	Assigned chips	Mapping	Groups	% Writes	% Reads
<b>Read-only/mostly, Append-based</b> History, Item, ...	21.81	23	BLM	1	2.3	30.2
<b>COLD</b> Metadata, District, Warehouse	0.06	1	FASTer	1	0.1	0.1
<b>WARM</b> NO_IDX, O_IDX, OL_IDX	11.90	14	DFTL	1	8.6	1.4
<b>HOT</b> Stock, Customer, ...	12.66	26	PLM	3	89.1	68.2

Table 6.5.: Data placement configuration with selective, region-specific Flash management for TPC-C benchmark.

Columns [*Single-FSM BLM/FASTer/DFTL/PLM*] in Table 6.6 show results for tests with

Flash SSD 50GB TPC-C, SF = 150 2 hours		Single-FSM BLM 1 region 1 group No IPA	Single-FSM FASTer 1 region 1 group No IPA	Single-FSM DFTL 1 region 1 group No IPA	Single-FSM PLM 1 region 1 group No IPA	Multi-FMS 4 regions 3 groups No IPA	Multi-FMS 4 regions 3 groups IPA
Host Reads (4KB)		6,998,704	30,264,560	33,833,853	57,634,830	66,890,004	74,345,031
Host Writes (4KB)		3,327,108	24,177,351	28,590,405	48,224,391	56,432,211	60,388,800
GC Page Migrations		197,686,609	87,263,511	32,894,397	14,285,105	9,583,656	5,507,492
GC Erases		3,138,778	1,741,134	940,719	869,957	952,206	440,659
GC Page Migrations per Host Write		59.4169	3.6093	1.1505	0.2962	0.1698	0.0912
GC Erases Per Host Write		0.9434	0.0720	0.0329	0.0180	0.0169	0.0073
I/O Response Time [ $\mu$ s]	READ	4,522	808	734	244	179	140
	WRITE	21,050	1,507	472	391	354	324
Trx. throughput		116	600	677	1,146	1,330	1,414
Memory (MB)		0.78	5.78	2.50	50.00	21.23	21.23

Table 6.6.: Comparison of NoFTL approach with a single- and multi-FMS under TPC-C benchmark.

single-region configuration, where that region was using (i) pure block-level mapping (BLM), (ii) hybrid address translation scheme FASTer [56], (iii) variant of page-level mapping - DFTL [28], and (iv) pure page-level address translation (PLM), respectively. Columns *[Multi-FMS No-IPA/IPA]* correspond to tests with the multi-region configuration shown in Table 6.5; the only difference between these two tests is that in *[Multi-FMS IPA]* case the *HOT* region was additionally configured with the IPA support (see more about IPA in Chapter 7).

Among four uni-region configurations, the scheme with pure page-level address translation (PLM) can achieve the highest transactional throughput (1146 TPS), which is almost 70% higher than the scheme with DFTL, 90% higher than the configuration with hybrid mapping (FASTer), and almost 10x higher than the throughput of region with block-level mapping (BLM). The major factor for the performance differences between different schemes is the overhead of the garbage collection. Thus, under hybrid FASTer scheme GC has performed 12x more page migrations and 4x more erase operations as compared to the scheme with pure page-level mapping. Although the DFTL scheme has significantly lower GC overhead than FASTer (3x less page migrations and 2x less erases),

---

it has additional CPU overhead which is caused by lookups in cached mapping tables (see Chapter 2.3.2), which also has a negative impact on system performance.

On the other side, the uni-region PLM case has the highest memory consumption (see row *[Memory (MB)]*) among all other schemes. It requires 20x as much memory for mapping table as FASTER scheme, and 8,6x more memory than DFTL scheme. This is the main reason why modern FTL-based Flash SSDs cannot utilize pure page-level address translation, although it produces the lowest GC overhead. As on-device DRAM is typically quite limited, its capacity is not sufficient to cache the complete mapping table on page-level granularity, and simultaneously support reasonable sizes of read/write I/O caches. That is why modern SSDs typically use FTLs based on hybrid mapping schemes. Under the NoFTL architecture the issue with insufficient amount of DRAM needed for Flash management is solved through use of host memory for this purpose. Thus, pure page-level address translation can be typically applied for NoFTL-based Flash storage without sacrificing other memory resident data structures (e.g., DBMS buffer). However, in practice, NoFTL does not require even that amount of memory. With the proper multi-region configuration the total memory footprint of NoFTL is much less than it would be for a single-region configuration with pure page-level address translation.

Column *[Multi-FMS 4 regions 3 groups No IPA]* in Table 6.6 shows the results for a data placement configuration presented in Table 6.5. The total memory consumption for Flash management data (e.g., address translation table) in all four regions is 2.4x smaller than in case of uni-region configuration with PLM. Note that in this case the reduction of memory footprint does not sacrifice the performance. In contrast, the transactional throughput is 16% higher as compared to *[Single-FSM PLM]*, and the write-amplification (GC page migrations per host write) is 43% lower. Under workloads that are more realistic than TPC-C, with broader range of I/O patterns and diversified DB-schema, the memory savings and performance improvement via the flexible Flash management would be significantly higher.

The use of IPA approach for *HOT* region allows to further decrease the GC overhead (column *[Multi-FMS 4 regions 3 groups No IPA]*): 46% less page migrations and 57% less erase operations as compared to configuration without IPA support (more about IPA in Chapter 7).

**Write-Amplification** By reducing the GC overhead, employing regions and all other factors already discussed, NoFTL performs less writes and erases per DBMS write. Low write-amplification leads to better I/O performance and energy efficiency, but also to higher Flash longevity and performance stability over time. Figures 3.6 and 6.6 summarize some of the numbers already discussed, to highlight the impact of NoFTL on write-amplification.

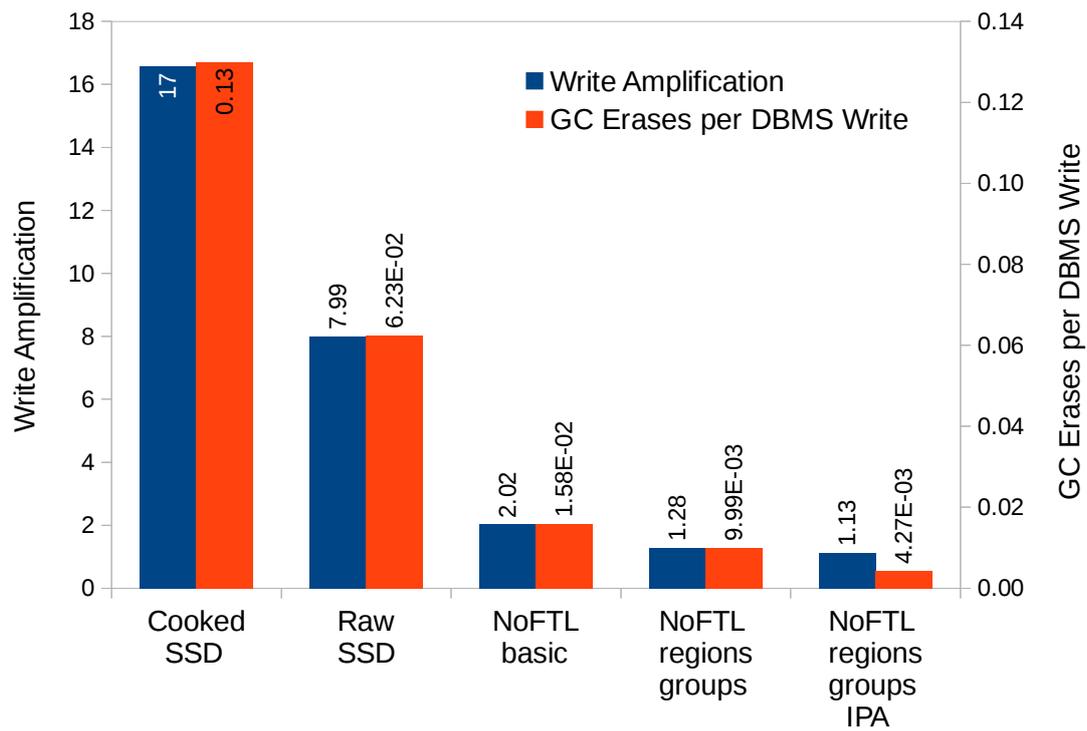


Figure 6.6.: Comparison of write amplification and intensity of erase operations for different storage alternatives under TPC-B benchmark.



---

## 7. In-Place Appends

---

This chapter is dedicated to an approach of *in-place appends* (IPA), which we have designed and implemented within the NoFTL architecture. IPA allows us to significantly reduce the write-amplification under OLTP workloads (up to 70%), as well as improve the longevity of Flash storage devices (>2x). IPA and its results were presented in [31], [36] and [32]. Below we provide a more detailed description of the approach, its implementation and evaluation results, which partially go beyond the scope of the aforementioned publications.

The motivation for the IPA approach is based on an observation of huge write-amplification, which is common for the majority of OLTP workloads running on modern Flash storage. The reasons behind such write-amplification are manifold: 1) outdated assumptions regarding the storage rooted deep in multiple subsystems of the DBMS; 2) cooked storage alternative; 3) backwards compatibility of SSDs; and 4) their black-box architecture. Consider an example depicted in Figure 7.1, where a transaction modifies a single field of a tuple (e.g., withdrawing a certain amount from a user's bank account). Although the modification is limited actually just to few bytes on a page (Figure 7.1.a), many DBMSs overwrite the whole tuple (b) and consequently modify the page's metadata (c). Thus, an update of 4-10 bytes already at this stage has turned into a modification of hundreds of bytes, i.e., a write-amplification of at least one order of magnitude.

Further, due to block-device interface and common page-based data management of DBMS, the whole database page (e.g., 4KB) is submitted to the storage on eviction from the buffer pool (Figure 7.1.d). If the cooked storage alternative is used, then the file system, depending on its type and configuration, might add its own contribution to the overall write-amplification. Previous studies have reported write-amplification of common file systems being in a range of 2.5-5 for OLTP workloads (e.g., [59]). Our own measurements for the TPC-C and TPC-B benchmarks have verified those numbers (see Tables 5.1, 5.2). At this stage, the write-amplification in written bytes has reached two to four orders of magnitude. For instance, four changed bytes of a numeric attribute being changed by a transaction have turned into 4K to 20K bytes submitted to the SSD.

As if the story were not dramatic enough at this point, the FTL-based SSD adds its own part to the general picture. Backwards compatibility of modern SSDs creates an illusion of "free" in-place updates on Flash, simulating thereby a traditional HDD. In turn,

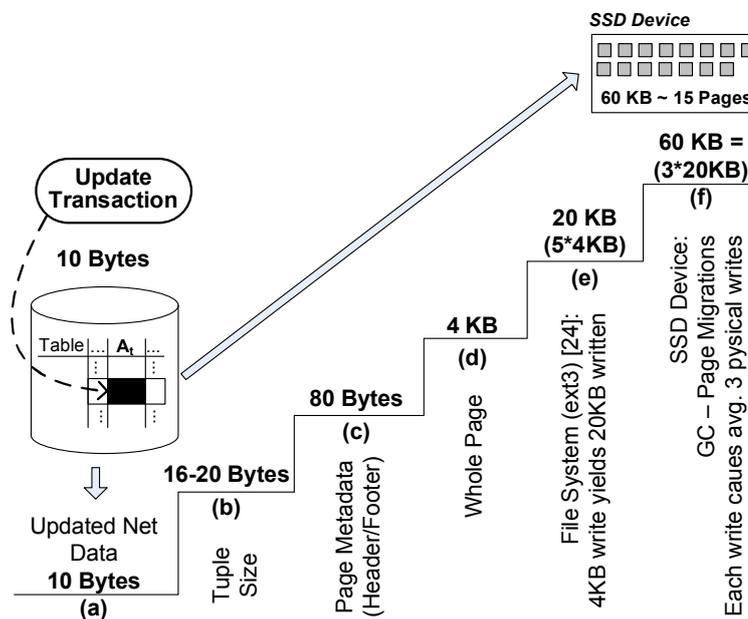


Figure 7.1.: Write-amplification under OLTP workloads. Source: Hardock et al. [31].

the DBMS and the file system take this behavior of the storage for granted and submit modified pages to the same logical addresses. Consequently, the FTL invalidates the old versions of pages and writes updates to new locations on the Flash. In the meanwhile, the garbage collection running in the background reclaims the space occupied by the invalidated pages. Invalidation of pages and page migrations performed by GC cause an additional write-amplification. Depending on the state of the SSD and the applied FTL scheme, the on-device write amplification can vary between 1.5 and 3, i.e., one page arrived at the SSD might result in up to 3 physical page writes. Thus, while in the best case few modified bytes result in 4K bytes being physically written and another 4K bytes invalidated, in the worst case up to 60K bytes (15 pages) are written on Flash. This sums up to write-amplification of 1000 to 15000 in terms of bytes (I/O size), and 1 to 15 in terms of written pages (I/O count).

In the example above we have considered a single case of write-amplification, where only few bytes of data on a page are modified upon its eviction from the buffer pool. This case is, however, representative for the majority of update-intensive workloads. Updating a user balance or product count in stock, setting a flag or changing a status of an online transaction, as well as other simple operations, dominate OLTP workloads, and typically

---

---

change only one or two fields of a single tuple on a page. Moreover, the randomness of requests and large sizes of relations often cause modified pages to be evicted from the buffer before the next change in this page occurs. Even in database systems, where the working set fits completely in main memory, cold pages with only few changed bytes are regularly flushed to storage in order to reduce recovery time in case of failure. Thus, our analysis of typical OLTP benchmarks (TPC-B, TPC-C) has shown that up to 70% of all updates modify 10 or less bytes of a tuple's data on a page (excluding modifications in the page's metadata). The detailed analysis on these and other benchmarks is provided in Chapter 7.4.

This write amplification in OLTP workloads obviously has a significant influence on the whole system. Two major parameters mirroring this impact are I/O response times and longevity of Flash storage. Even if the former factor can be more or less important depending on the system configuration (e.g., in an I/O-bound system I/O response time is directly proportional to the transactional throughput, while in main memory DBMSs this correlation is weak), the SSD's longevity is critical in all systems because of cost and longevity. Apart from this, high write-amplification has a negative impact on available storage link (or network) bandwidth, stability of I/O throughput and power consumption by the storage.

The IPA approach can effectively reduce write-amplification in OLTP workloads, prolonging thereby the lifetime of SSDs and improving performance characteristics of the system. But before we describe the approach in detail let us look into one interesting (and commonly ignored) property of Flash memory, which lays at the core of IPA.

## 7.1. Revisiting Erase-Before-Overwrite Principle

One of the fundamental properties of Flash memory is the *erase-before-overwrite* principle (see Chapter 2.3.1), which means that once a Flash page is written (programmed) it cannot be overwritten until the corresponding Flash block is erased. However, it is worth to look a bit deeper into the write process on Flash, and check this property more closely.

Modern SSDs utilize the so-called Incremental-Step-Pulse-Programming (ISPP) technique [90], [66], [3] to perform a program operation on Flash. Herewith, the charge of every Flash cell within the selected page (i.e., wordline) is incrementally increased to its desired value in multiple iterations (see Figure 2.4). In the first iteration of ISPP programming some initial value of charge is stored in the selected cells<sup>1</sup>. After this, each cell in the wordline is sensed (read), and based on the difference between its actual charge

---

<sup>1</sup>Flash cells are selected during programming through manipulation of voltages on corresponding bitlines and wordlines (see Chapter 2.3.1).

---

and the target value, the next iteration of ISPP is performed, which puts an additional portion of charge into the cells. This process repeats until all cells in the wordline are programmed to their target values. Note, that some cells might need more iterations than others. This is especially the case in MLC or TLC Flash types, where cells can have different target values of charge, depending on their initial state<sup>2</sup>.

So, as we can see, Flash cells are actually modified multiple times during a single program operation (SLC NAND) or even during multiple separate operations (MLC and TLC NAND). And those modifications do not require any erase operation being performed in-between. Then, why is there a need to erase? The key point here is that the erase-free modification of Flash cells is possible only if their charges are **increased** during the subsequent programming. It is, however, not possible to decrease the cell's charge by a certain value, it can only be removed completely during an erase operation. Thus, if the current level of charge in a cell is higher than the target value, the only way to modify it would be to erase the whole Flash block containing this cell, and consequently re-program it with the desired value. This constraint is the main reason, why the erase-free modification of already written Flash pages is almost never utilized by SSD manufacturers. The probability that an arbitrary page update would result in the cells' charges being only increased is negligibly small in the general case. Assume for instance, that an application has updated only one numeric value on a certain page, e.g., from original value 357 to value 2372 (in binary, from 0000 0001 0110 0101 to 0000 1001 0100 0100). Thus, in the whole page (e.g., 4K) solely three bits of data are changed. However, because there is one bit that is flipped from 0 (SLC cell with a charge) to 1 (SLC cell with no charge) such an update cannot be performed on an original Flash page without the need of a preceding erase operation. Clearly, when a larger amount of data is modified on a page (e.g., 50 bytes of a 4K page), the probability of erase-free overwrites becomes minimal.

In some scenarios, however, the above constraint of erase-free overwrites is easily satisfied. For instance, in case of appends to the page. When data is inserted into the page, cells storing the original content remain unchanged, while all Flash cells that correspond to newly appended data would either get their charges increased, or left in an initial state without a charge<sup>3</sup>. Thus, whenever modifications on a page are limited just to append, SSD could perform such an update in an erase-free manner through direct overwriting of the original Flash page<sup>4</sup>, i.e., perform an **in-place update**. In contrast to the common

---

<sup>2</sup>In MLC (TLC) Flash type each cell stores two (three) bits of data, which belong to **different** Flash pages, which are programmed individually at different times

<sup>3</sup>Assuming that during the initial program operation free space on a page is filled with logical 1s, so that Flash cells that correspond to that area are not programmed

<sup>4</sup>On MLC and TLC Flash there is an additional constraint, which need to be satisfied for in-place updates (see Chapter 7.3.2).

---

out-of-place update (see Chapter 2.3.2), this would save on the need (1) to find a new location for the modified page; (2) to invalidate the original Flash page; (3) to reclaim the invalidated space by GC, which in turn causes multiple page migrations and erase operations.

However, the possibility of in-place updates is almost never utilized by modern SSDs. There are multiple reasons for this, which are again rooted in the block-device interface and black-box design of Flash storage. Thus, to realize the support of in-place updates a common SSD needs to solve the following challenges. First, on every write request SSD needs to compare a submitted page with its original version stored on Flash in order to determine if it can be overwritten in-place. This introduces computational (comparing two 4K pages) and IO (reading an original version of a page from Flash) overheads, which can easily wipe out the benefits of in-place updates. Second, this approach requires modification of error-correction techniques, which are hard to implement under the black-box architecture of SSD (see Chapter 7.2, ECC paragraph).

We are aware only about one case when in-place updates are utilized by commercial SSDs, and about one another case of its use proposed by academia. The former is known as *partial writes* supported by some SSDs, which are based on SLC Flash type. The idea here is to split a physical Flash page into  $N$  equal parts, and allow each part to be written separately. So, it is possible to perform up to  $N$  write requests on the same physical Flash page without the need to erase a block in-between. The only constraint here is that those parts can be written only in order, i.e., each next write to the same Flash page should be an append. On one side, through fixing the granularity of partial writes (e.g., 512 bytes) and aligning it to the sector size in the block device interface, the two main challenges in the realization of in-place updates mentioned above can be easily solved. However, on the other side, this significantly reduces the flexibility of in-place updates. Thus, in order to append a small record (e.g., 64 bytes) to the end of an existing Flash page, still the whole unit of partial write (512 bytes or more) must be written. Moreover, due to the specifics of MLC and TLC Flash (Chapter 7.3.2) the option of partial writes is only available on SLC Flash, which is rarely used nowadays. Multiple research approaches, like [70] and [51], utilize partial writes in order to reduce the overhead of garbage collection.

The second application case of erase-free in-place updates on Flash memory was proposed by Cai et al. in [109]. In their approach "Correct-and-Refresh" they apply in-place updates to mitigate retention errors on MLC Flash. Basically, they suggest to periodically read Flash pages, correct on-the-fly bit errors caused by retention (i.e., leakage of cells' charges with time), and consequently re-write the same page with correct data. Because in this correction the charges of cells can only increase, such an overwrite is possible without preceding erase operation.

---

## 7.2. Design and Implementation Details of IPA

Motivated by the observation of massive write-amplification, and exploiting the property of Flash memory to perform erase-free in-place updates provided the charge must only be increased, we propose the IPA approach. The basic idea of IPA is the following. A small space at the end of a database page is reserved and left unprogrammed during the initial write operation. This area is called *delta-record area* and its size is typically less than 5% of the page size, e.g., 100 bytes on 4K or 8K database pages. Later on, when the page is modified and gets evicted from the buffer pool, the changes that were performed are extracted and accumulated together in a special *delta record*. Further, through the novel *write\_delta* command only this delta record is transferred to the SSD, where it is written (appended) into the delta-record area of the very same physical Flash page the original data remains in. In other words, by transforming the changes on a page into appends, IPA can utilize the mentioned above property of Flash memory and "reuse" (overwrite) the original Flash pages, saving thereby on the need to perform an out-of-place update with all its disadvantages. When the database page, which was previously updated through IPA, is fetched from the SSD, the delta records are applied on the fly to the body of the page, and this way the up-to-date version of the page is placed into the buffer frame. It is worth to note that the DBMS operates on buffered pages in a traditional manner, i.e., all changes are performed as usual in-place.

The IPA approach is flexible and can be optimally configured depending on the properties of the Flash memory and characteristics of an update pattern. Size of delta-record area, form of delta-record and the maximum number of consequent in-place appends are the main configuration parameters. The combination of IPA with the concept of configurable storage in NoFTL allows us to achieve the maximum benefit. Thus, each *region* can have its own specific IPA configuration (including the choice of utilizing IPA or not), which is selected based on properties of database objects placed in that region. The *NoFTL Advisor* assists the database administrator in choosing the optimal region-specific IPA configuration. IPA is of special benefit for OLTP-like workloads, which are characterized by high update rate and small sizes of performed changes (e.g., changing a numeric field or setting a flag). In those scenarios, by sacrificing a very small percentage of page space for the delta-record area, it is possible to reduce the write-amplification and the number of performed erase operations more than twofold, which consequently has a positive impact on the performance of the DBMS and the longevity characteristics of the SSDs. But before we look closely into the evaluation results, let us first dive deeper into the important details of the IPA approach.

### 7.2.1. Page Layout for IPA

Figure 7.2 shows the database page layout modified for supporting the IPA approach. The new page format here is based on the traditional NSM layout (slotted page), since it is the primary choice for DBMSs serving OLTP workloads. However, it is worth to mention that IPA can be applied in a similar manner for other page formats, like DSM and PAX [1].

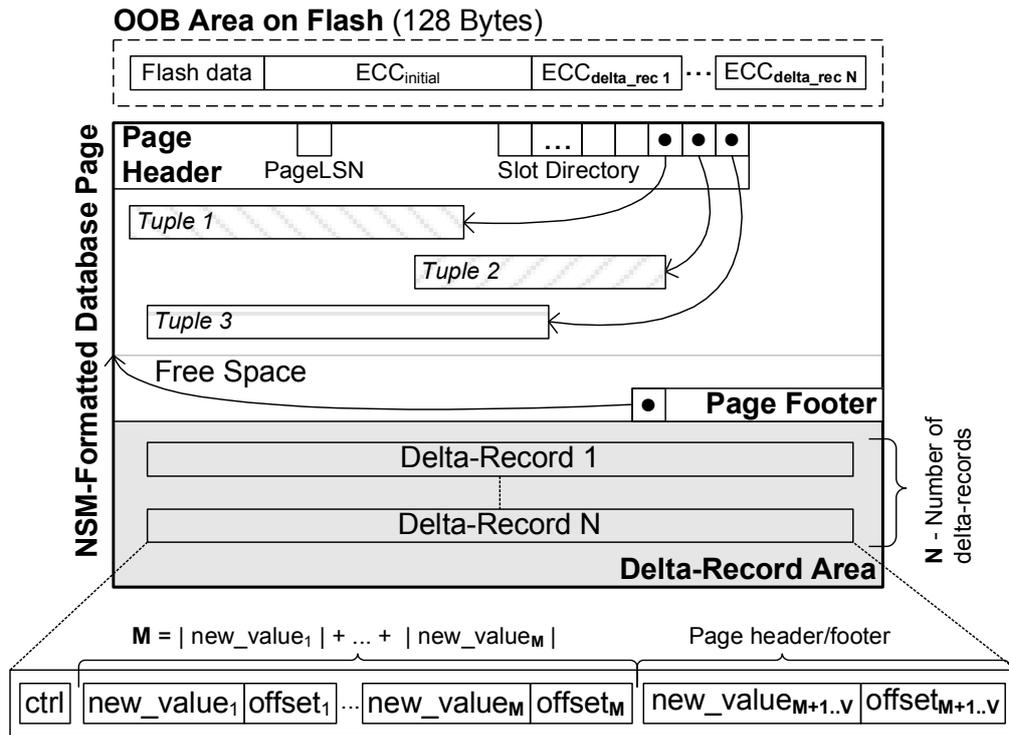


Figure 7.2.: Database page layout for IPA on Flash. Source: Hardock et al. [31]

As already mentioned, under IPA we reserve a fixed size *delta-record area* at the end of free space of each database page (if that page belongs to a region with enabled IPA). The size and structure of this area is determined by three major configuration parameters.

1. Parameter  $M$  specifies the size in bytes of a single delta record<sup>5</sup>. This parameter primarily depends on access properties of database objects in a NoFTL region, to

<sup>5</sup>We use here another definition of parameter  $M$  as compared to the one in [31]. In [31] IPA parameter  $M$  describes the maximum number of changed bytes in tuple data (net data) of database page that can be encoded in one delta record.

---

which IPA is applied. Properties such as size, frequency, and locality of updates are the most relevant for the choice of  $M$ . In most cases a simple analysis of distribution of update sizes in a region is enough to select an appropriate value for  $M$ . In Chapter 7.4 we explain in more detail such analysis and provide multiple examples.

2. Format of delta-record. It determines how changes within a database page are encoded inside a delta record. In our prototype we have realized and evaluated two basic formats. The *offset-value pair* format stores modifications on a page as a list of tuples, where each tuple consists of a modified byte and its offset within a page. For instance, tuple [1024, 77] means that the field starting at byte offset 1024 has value 77 (Figure 7.3). Depending on the maximal supported size of database page, the offset can be two or three bytes in size. Hereafter, we assume that two bytes would be enough, as pages larger than 64KB are rarely used for OLTP workloads. Thus, each single byte, which was changed in a page, requires three bytes in delta record. In other words, the  $M$  bytes large delta record can store up to  $M/3$  modified bytes of net data. This simple format is especially applicable for database objects dominated by random, small-sized updates. However, for such database objects as B-Tree indices offset-value format is sub-optimal, as it would result in a significant space overhead produced by IPA. Therefore, we designed the *log-based* format of delta-record for B-Tree indices. The detailed description of that format is provided in Chapter 7.3.1.
3. Parameter  $N$  specifies the maximal number of delta records that can be stored in delta-record area. The choice of  $N$  depends primarily on physical characteristics of Flash memory (e.g., Flash type, resistance to program interference errors). The detailed description of those characteristics and their influence on parameter  $N$  is provided in Chapter 7.3.2. Another factor that impacts the selection of  $N$  is the tradeoff between the space overhead of IPA (and its implications) and expected advantages of applying IPA (I/O performance and longevity of Flash SSDs).

Therefore, the total size occupied by the delta-record area is a product of two main parameters  $N$  and  $M$ . To describe a certain IPA configuration we use the notion of  $[NxM]$  *scheme*. Thus, IPA [2x48] assumes an IPA configuration, where delta-record area can contain maximal two delta records with offset-value pair format and maximal 48 bytes per record. Similarly, IPA-IDX [2x48] corresponds to the configuration, where delta records use log-based format.

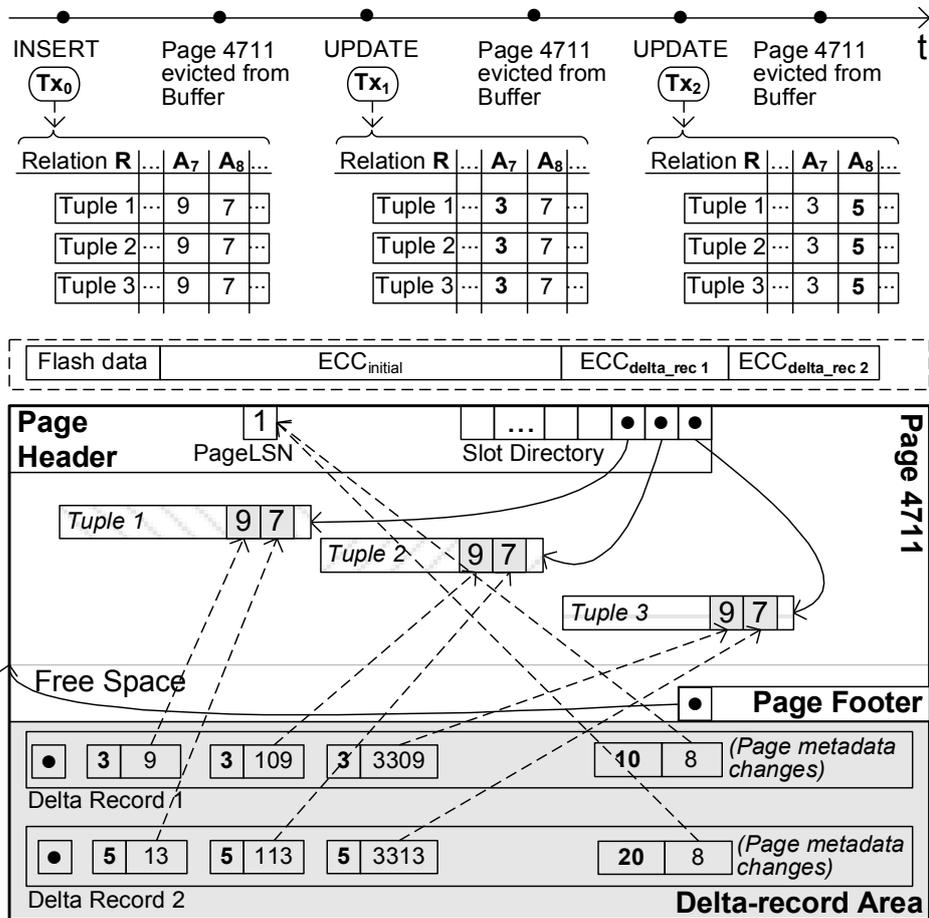


Figure 7.3.: Example of database page with two delta records. Source: Hardock et al. [31]

### 7.2.2. Page Operations under IPA

In order to realize the IPA approach three main operations on a database page must be modified.

- *Fetching of a page from Flash storage into database buffer.* Whenever a page that belongs to a region with enabled IPA is read from storage, the storage manager checks if there are delta records in the delta-record area. If so, those delta records are applied in FIFO order to the body of the page. For instance, if offset-value pair

---

format is used, a simple algorithm iterates over all such pairs stored in delta records, and substitutes bytes at corresponding offsets with their new values. Herewith, the up-to-date version of the page in the database buffer is restored, and it can be accessed by transactions. This additional step during a fetch operation does not produce any memory overhead and very low to negligible computational overhead. Thus, in all our experiments we could not measure any increased CPU overhead related to the process of applying delta records (e.g., replacing 30-50 bytes in a 4KB page). If delta-record area is already full, the page is marked as "No-IPA" (a bit in a page header), which means that next time it will be written out to storage in an out-of-place manner.

- *Modifications on a page.* If the page belongs to a region with enabled IPA, and delta-record area has free space (not marked as "No-IPA") the buffer manager performs tracking of changes on that page. On every update the buffer manager checks if the modified content can fit into the remaining space on a delta-record area. If so, it encodes those changes in selected format and appends them to the current delta record. Note that the content of the page is modified as usual, i.e., in-place. In other words, IPA solely "remembers" what has been changed on a page while it is being cached in the buffer, and it keeps those "notes" in a small area at the end of the page. Delta-record area of the page is completely transparent to the database operations, which work with the up-to-date content stored in the buffer frame. If the current changes cannot fit into the delta-record area, the page is marked as "No-IPA", the content of the delta-record area is ignored, and no further tracking of modifications on that page is performed.
- *Flushing a page to Flash storage.* When the database system decides to flush a modified page to the storage (e.g., due to the work of a background write process), we check if the page can be written out using the IPA approach. If IPA write is not possible ("No-IPA" flag is set) the page is written out in a traditional out-of-place manner on a new location on the Flash SSD. The delta-record area is left empty. Otherwise, if IPA write is possible, the storage manager issues a **write\_delta** command, which transfers to the Flash storage only relevant delta record(s) (those added since the last fetch), which in turn are appended to the very same Flash page the original data is stored in. Remember that appending of delta records to the original Flash page is done without foregoing erase operation. If after flush the page is not evicted from the buffer, we check if also future changes on that page might be saved on the next flush operation using IPA. This is the case when delta-record area has free space and the number of already performed consequent writes on that

---

page using IPA is less than parameter  $N$ . If both conditions are fulfilled the buffer manager simply continues tracking new changes. When consequent IPA write is not possible, the "No-IPA" flag is set, and thus further changes are not tracked.

It is important to note **that apart from the modifications in storage and buffer managers described above, the IPA approach does not change any further database functionality**. Thus, logging, recovery, commit protocol, buffer replacement strategy, etc. are not influenced by the use of IPA. Moreover, IPA does not influence the database decisions regarding the times to fetch and flush pages. IPA does not cause any additional I/O requests. When the delta-record area is full, no I/O is triggered; the content of this area ("notes" about performed changes) is simply ignored.

### 7.2.3. WRITE\_DELTA Command

To enable the database system to perform in-place appends on Flash pages we introduce a novel I/O command under NoFTL:

**write\_delta{pno, offset, size, data}**

With this command *data* of certain *size* is appended to the Flash page with the physical page number *pno* starting at specified *offset*. For an IPA region with certain [NxM] scheme *data* represents one or multiple delta records of total *size*  $k * M$ , where  $k$  is the number of submitted delta records, while *offset* is an offset within delta-record area, at which new delta records should be appended.

Correspondingly, the Flash storage device must also provide the support for the new command. The physical details of implementation of *write\_delta* on the side of a Flash SSD might differ depending on the type of Flash memory and controller specifics. As an example of one possible implementation, let us look at how we implemented this command on OpenSSD Jasmine board [94], which we used for the evaluation of IPA in our NoFTL prototype.

The challenge with the OpenSSD Jasmine board is that the on-device ARM controller allows to reprogram only the FTL logic, while the large part of the firmware, including Flash controller and on-device DRAM controller, cannot be changed. Because of this in our implementation of *write\_delta* we were forced to utilize the standard PROGRAM\_PAGE command of the Flash controller, which accepts and writes only whole Flash pages. However, we implemented an effective workaround solution. By issuing a *write\_delta* command the database system submits only relevant delta records. As this chunk of data arrives at OpenSSD Jasmine board we copy it to the empty buffer frame at an offset specified in the command. The remaining empty space in the frame we fill with logical "1"s (see Figure 7.4). This frame is then written using the standard PROGRAM\_PAGE

command to the original physical address (*PPN*). In other words, the page consisting almost completely of "1"s and only small portion (typically <5%) of delta records is written "on top" of the existing original page. As logical "1"s do not cause any increase of voltage thresholds in the corresponding cells, the already existing data in the Flash page remains unchanged during this program operation. At the same time, the region of Flash page that corresponds to new delta records is programmed appropriately (see Chapters 7.1 and 2.3.1). In this way, we achieve the desired result of the *write\_delta* command - in-place modification of Flash page without a previous erase operation.

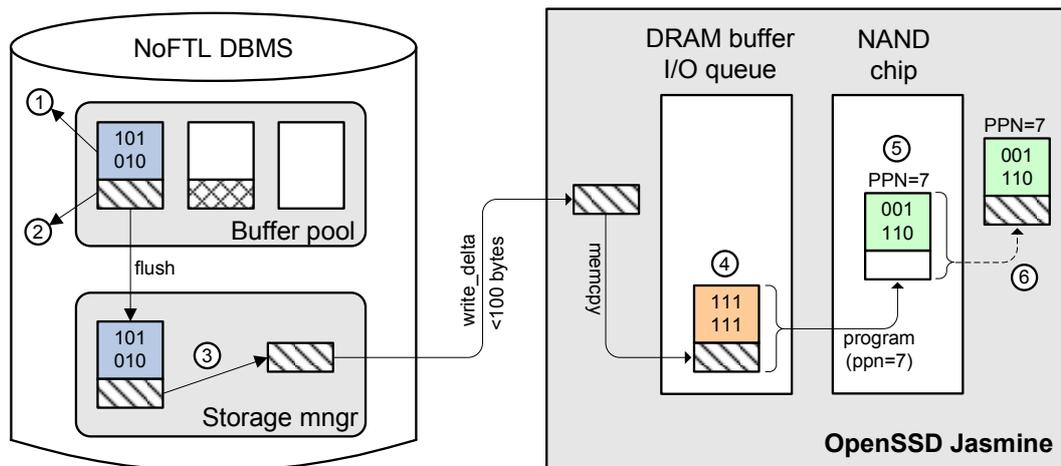


Figure 7.4.: Implementation of *write\_delta* command on OpenSSD Jasmine board. (1) up-to-date version of a page in database buffer pool; (2) "remembered" changes since last fetch from SSD encoded as delta record in delta-record area; (3) storage manager issues *write\_delta* command only for delta record, i.e., only delta record is transmitted to the SSD; (4) controller copies delta record into the original position in an empty page (initialized with "1"s) in DRAM buffer of SSD (write cache); (5) original Flash page on Flash with empty delta-record area; (6) through overwriting the original (PPN=7) page only the delta-record area becomes updated, while the body of the page remains unchanged.

The only overhead of this implementation is caused by the initialization of an empty buffer frame with logical "1"s. In an appropriate implementation of *write\_delta*, which is natively supported by the Flash controller, this can be eliminated or optimized by hardware implementation. Additional details regarding the implementation of *write\_delta*

---

command in OpenSSD Jasmine board are provided in 7.3.2.

It is important to mention the difference between the novel *write\_delta* command and the so-called *partial\_write* command available on some types of SLC Flash memory. Partial writes allow us to perform multiple program operations on the same Flash page without performing an erase operation in between. The Flash page is split into equal fixed-size chunks (four 2KB chunks in a 8KB Flash page), and each such partial write appends the next chunk to the Flash page. Both *write\_delta* and *partial\_write* commands are based on the same physical properties of Flash memory and utilize ISPP technique. However, *write\_delta* is more general and flexible: (i) it is available for all types of Flash memory, such as SLC, MLC and TLC (see more in Chapter 7.3.2; and (ii) the size of appended data is not fixed and may change over time.

#### 7.2.4. Error Detection and Correction

The introduction of the *write\_delta* command requires also an adjustment of error detection and correction techniques used by the Flash controller. There are several approaches used by SSD manufacturers to calculate and store error correction codes (ECC). Thus, there might be a single ECC calculated for the whole Flash page, or the page can be split into equal parts with each having its own ECC. Correction codes might be stored in a single place in the OOB area, or distributed over several locations in a Flash page.

To support IPA we need several error-correction codes: one for the initial content of the page excluding delta-record area, and a separate ECC for each delta record appended via the *write\_delta* command. Those codes might also be stored in a single dedicated area in OOB (see Figure 7.2), or for instance, at the beginning of the corresponding sections. However, in contrast to equally sized parts, the size of the main part of the page (with initial content), and the size of delta records are not fixed over the whole Flash SSD, as they are determined by the concrete [NxM] scheme applied for a particular region. Thus, in order to properly read and apply correction codes the Flash controller must be aware about the sizes and placement of those parts. There are several options how this can be realized. The most simple one is to store the relevant [NxM] scheme directly in every page. Thus, solely two bytes are enough to encode parameters *N* and *M*, which are written into a dedicated place in the page OOB area during the initial program operation.

As in OpenSSD Jasmine board there is no opportunity to change the hardware implementation of ECC, and we cannot access the page OOB area, we shifted this task to the server side. Thus, in our prototype calculation of error correction code for a Flash page, ECC storage, detection of errors and their correction are done by the database system. Fortunately for us, *the default implementation of ECC in OpenSSD board performs only detection of errors, while correction is not done. This fact is very important, since it has*

---

allowed us to realize and evaluate IPA to its full extent on the real Flash storage. Whenever we modify the page using the IPA approach, the original ECC code generated by the Flash controller for the whole page becomes invalid. On reading such a page the controller reports the bit errors it finds, but does not try to correct them. Thus, we simply ignore these errors and perform our own ECC check and correction in the storage manager of the DBMS.

## 7.3. Variations of IPA

### 7.3.1. IPA for Indexes

The basic IPA approach, which stores delta-records in form of offset-value pairs, is the perfect choice for database objects dominated by random updates of small sizes. Good examples are tables in which transactions modify one or multiple numeric, enumeration, boolean or timestamp attributes per record (e.g., STOCK table in TPC-C benchmark, ACCOUNT table in TPC-B benchmark). In those cases, few to several dozens of changed bytes per database page turn into a delta-record, which is about 20 to 100 bytes in size (offset-value pair requires 3 bytes per each modified byte on a page). Thus, total size of delta-record area comprising two or three delta-records would occupy less than 5% of a 4KB database page. Taking into consideration that IPA is typically applied selectively using *regions*, the overall space overhead caused by IPA becomes negligible (typically below 1% of database size).

However, the offset-value format of a delta-record is poorly suited for such database objects as B-Tree indices. To illustrate this, let us consider a simple example. Assume a B-Tree index *Idx1* on a numeric field *Attr1* of database table *Tbl1*. *Idx1* is declared as unique. Transaction *Tx1* modifies a single record of *Tbl1* by changing the value of *Attr1* from 10 to 100. As a result, the two-step update operation on *Idx1* is triggered (Figure 7.5).

1. *Index entry with the key 10 is deleted from the leaf page 456.* There are multiple techniques to realize the deletion of an index record. One of the most common approaches is to remove only the corresponding entry in the slot directory of the page, while the record itself is left unchanged until its space is either reused by the following insert operation or reclaimed during page compaction. The slot directory is always kept sorted and often also compacted, which is needed to support efficient binary search during lookup within an index page. Thus, by removing a single entry the remaining slots on the right side are shifted by one. On average, half of the entries in the slot directory must be shifted.

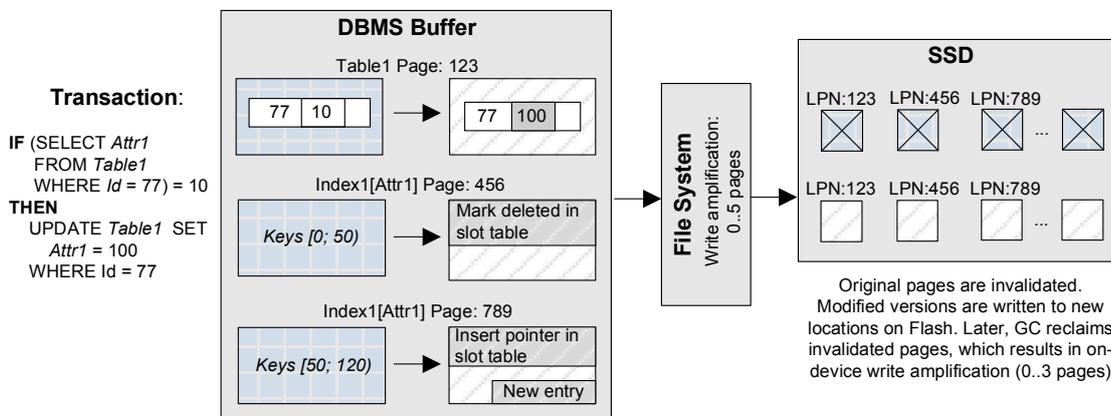


Figure 7.5.: Modification of an entry in B-Tree. Traditional approach.

2. *New index entry with the key 100 is inserted into the leaf page 789.* Insertion requires modification of both the slot directory and the page body. The new entry is inserted into the slot directory according to the sorted order of keys, which on average requires half of the slots to be shifted by one position. The record itself is typically placed without any special ordering; it is either appended to the end of the page body, or inserted into the hole left after the previous delete operation.

Thus, the one-field modification of a single record in *Tbl1* causes three database pages to be updated (e.g., pages 123, 456 and 789). The changes in page 123 of a heap file *Tbl1* are small, i.e., few dozens of changed bytes in page body and metadata. If such changes are dominant for this database object, it is a good candidate for application of the IPA approach with the standard offset-value format for delta records. In this case, the delta-record area with multiple records occupies only a few percent (e.g., 1-5%) of the index page, resulting therefore in negligible space overhead of the IPA approach.

However, due to reorganization of the slot directory, the changes to index pages 456 and 789 are more complex, and typically cause several hundreds of bytes on each page to be changed (see Figure 7.12). Applying an IPA approach with offset-value format would result in delta-records being 500 and more bytes in size. Thus, the total space occupied by the delta-record area with just two delta-records would already be more than 1KB, which is about 25% of a 4KB page. The increased storage consumption itself is typically not a big problem. Especially, taking into account that through the use of selective IPA the overall overhead for a database would be much lower, as only relevant database objects

---

will grow in size. However, more important is the impact of a larger delta-record area on a database buffer. "Stealing" a large portion of an index page for the IPA approach means that the amount of index records per page is reduced correspondingly. This leads to a higher ratio of cache misses and more frequent I/O requests, which would compete with the reduced write amplification and number of erases achieved through IPA.

On the other side, indexes are typically the hottest and most write-intensive objects in the database, and therefore they would especially benefit from an IPA approach. That is why we propose the modified version of IPA for B-Tree indexes - *IPA-IDX*. The main difference to the traditional IPA approach here is the format of a delta-record. Instead of storing modified bytes (offset-value pairs), *IPA-IDX* creates the delta records using physiological log records (see Figure 7.6)<sup>6</sup>. Usage of log records allows *IPA-IDX* to reduce the space requirements for delta-record area by factor 2 and more, as compared to the traditional IPA approach.

The behavior of *IPA-IDX* is similar to the general IPA approach (see Figure 7.7). On each modification of an index page in the buffer, the buffer manager checks if the current change should be tracked by *IPA-IDX*. For this two conditions are required: (i) page can be overwritten in-place, i.e., number of already performed in-place overwrites is less than  $N$ ; (ii) corresponding log record for this change fits into the remaining space in delta-record area. If both requirements are fulfilled, the log record is copied to the delta-record area of the buffered page. This process is repeated for the subsequent changes on that page. If one of the above conditions is not satisfied, the page is marked as one that will be written out using an out-of-place strategy, and no further *IPA-IDX* tracking is performed until the page is flushed. When the database decides to evict the page from the buffer, the storage manager checks if the page should be written using an in-place or out-of-place approach. If in-place update is possible the page is written using the *write\_delta* command, otherwise the normal write to a new Flash address is triggered. Note, like the general IPA, *IPA-IDX* does not trigger any additional I/O requests, nor does it influence the time an index page is cached by the database buffer, nor the point at which it is flushed to the storage. *IPA-IDX* changes the traditional behavior of the database buffer only to the extent that allows to remember the corresponding log records in a page's delta-record area as the page is being modified.

When the page is fetched from storage, the storage manager checks if the page contains delta records. If so, it applies them in FIFO order by performing logical operations of log records stored in those delta records. Applying delta records in *IPA-IDX* is similar to

---

<sup>6</sup>Physiological logging is the common type of logging applied to B-Tree indexes in most database systems. Log record of this type captures the logical operation performed within a certain database page. This allows to keep the flexibility of physical logging and space efficiency of logical logging.

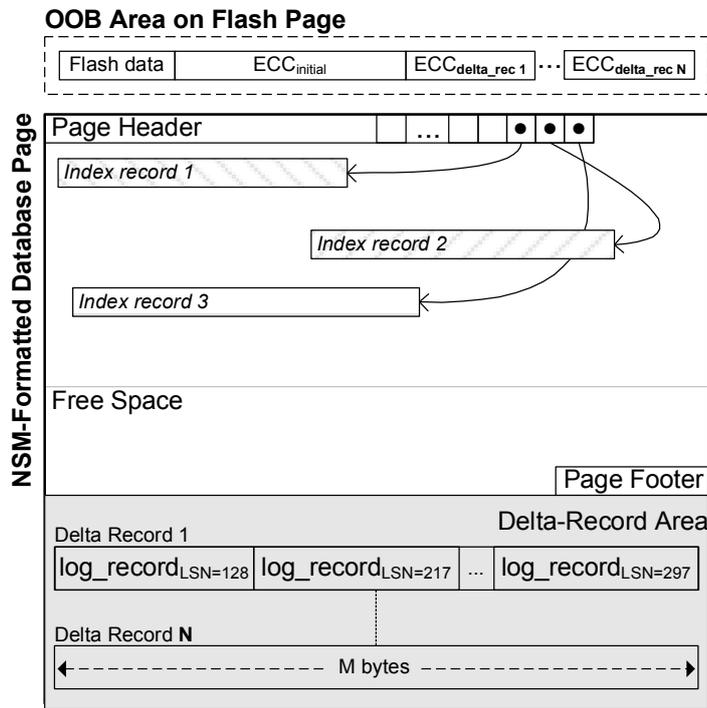


Figure 7.6.: Page layout and format of delta record in IPA-IDX. Source: Hardock et al. [33]

executing the REDO phase of recovery for one particular page.

### 7.3.2. IPA on Different Flash Types

Vulnerability of Flash memory to program interference errors requires also a special attention during realization of an IPA approach. As different types of Flash memory differ in their reliability guarantees (see Section 2.3.1), we propose several modes of IPA approach that allow safe application of IPA on each particular Flash type. Region-specific Flash management (see Chapter 6.1.2) makes it possible to use multiple IPA modes (depending on the NAND mode of the region) simultaneously.

#### IPA on SLC Flash

SLC Flash is the most durable type of Flash memory, which is aligned with its property to be the least sensitive to program interference and read disturb errors, as compared to

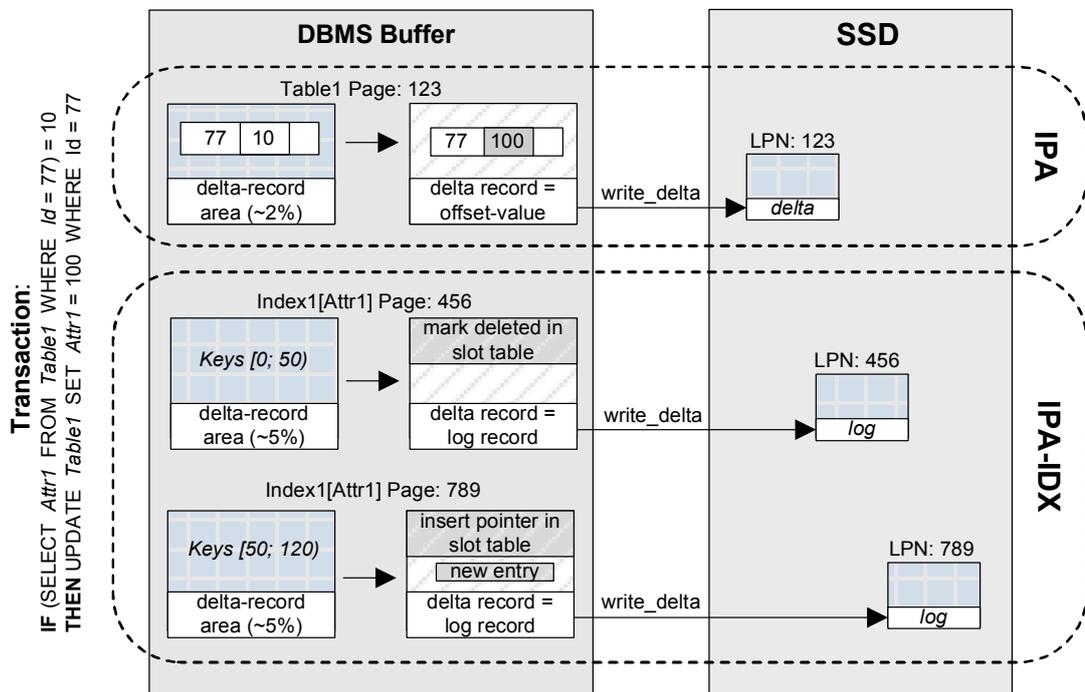


Figure 7.7.: Modification of an entry in B-Tree using IPA-IDX.

other Flash types. This immunity to bit errors is a result of the large distance between two threshold voltages which indicate different bit codes of a page (logical "1" or "0"). In other words, Flash cells can accumulate many small shifts in a cell's voltage, which happen occasionally during programming or sensing operations on neighboring cells, without a risk of leaving a voltage interval that belongs to the corresponding bit value. That is why IPA can be applied on SLC Flash without restrictions. Parameter  $N$ , which defines maximal number of delta records (i.e., maximal number of consequent IPA writes), is limited for SLC Flash primary based on the space utilization factor, and not based on vulnerability of Flash to bit errors.

### IPA on MLC/eMLC Flash

MLC is nowadays the most used type of Flash memory in SSDs (Chapter 2.3.1). Although its durability and performance characteristics are worse than those of SLC Flash, it has better cost/performance tradeoff as it doubles the density. Due to higher vulnerability

---

---

to bit errors, application of IPA on MLC Flash has some type-specific implementation solutions. Before we describe two different modes of using IPA on MLC Flash, let us look into some physical details of this Flash type.

MLC Flash can store twice as much information in the same amount of Flash cells as SLC Flash due to the introduction of four different threshold voltages (instead of two in SLC). Thus, the main difference between SLC and MLC lies in the Flash controller, which performs programming and sensing operations, and not in the physical structure of the cells (they can be absolutely the same). The fact that the controller in MLC Flash must differentiate four different values between states  $V_0$  (no charge) and  $V_{max}$  (max level of a cell's charge) is the main reason for performance degradation of I/O operations and increased sensitivity to interference between neighboring cells. To minimize the effect of those interferences manufactures apply several techniques.

Two bits that are encoded in a MLC Flash cell belong to two different physical pages: LSB (least significant bit) page, and MSB (most significant bit) page. Moreover, those pages do not have consequent addresses. Thus, LSB pages have odd numbers ( $2 * N - 1$ ), while MSB pages follow even numbers ( $2 * N + 2$ ), where  $N$  is an index of corresponding wordline (see Figure 7.8)<sup>7</sup>.

Within a block Flash pages must be written in order of their physical numbers. That is, the sequence of programming operations is the following:

WL0(#0-LSB) → WL1(#1-LSB) → WL0(#2-MSB) → WL2(#3-LSB) → WL1(#4-MSB) → WL3(#5-LSB) → WL2(#6-MSB) → ...

As you can see, Flash cells on a certain wordline are modified twice, having two program operations on neighboring wordlines in between. This order ensures that after the last programming operation on a certain wordline (e.g., #4-MSB on WL1), a cell on this wordline can shift in its voltages only once by programming MSB page on the next wordline (#6-MSB on WL2).

Taking into account these specifics we offer two different modes for IPA application on MLC Flash: pSLC and odd-MLC.

**pSLC.** Every MLC Flash can also be used in the so-called **pseudo-SLC (pSLC)** mode. In this mode only LSB pages are utilized, while MSB pages are simply ignored. In this mode MLC becomes similar to the traditional SLC Flash, while the controller must differentiate

---

<sup>7</sup>The formulas for page numbers of LSB and MSB pages might differ for MLC Flash memory of different manufacturers. Thus, for instance, some manufactures produce MLC Flash chips with four Flash pages per wordline: two LSB pages (LSB-odd, LSB-even) and two MSB pages (MSB-odd, MSB-even). Each cell encodes further on two bits of data, which correspond to two different pages (LSB and MSB). Therefore, there are double as many cells per wordline which results in having four physical pages instead of the traditional two.

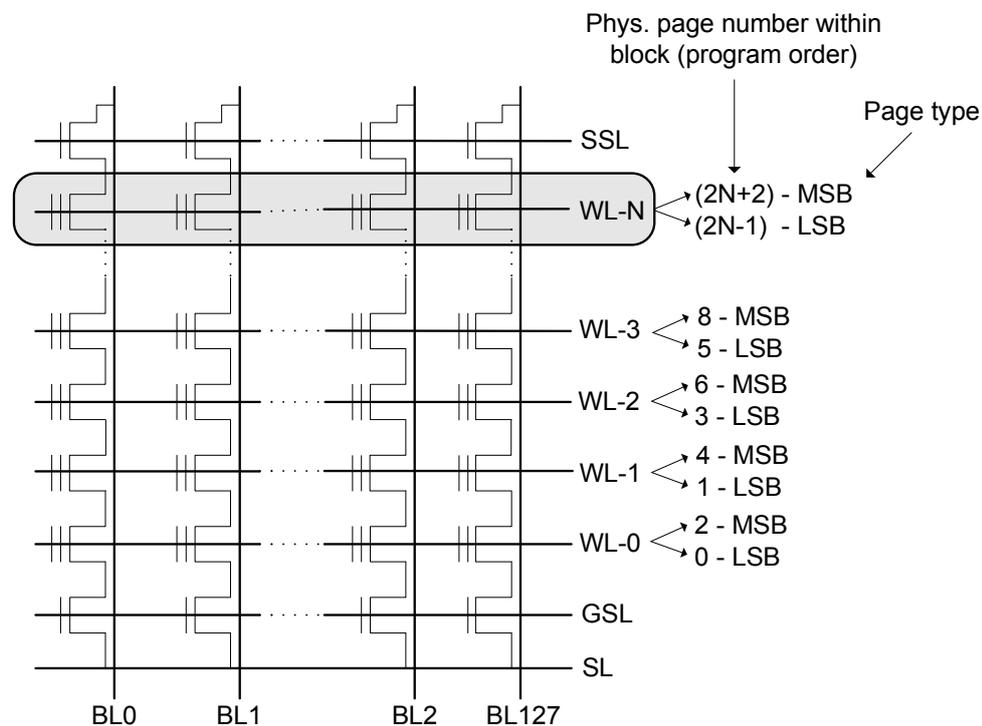


Figure 7.8.: Memory organization of MLC Flash.

only between two different threshold voltages of the cell, which have large distance between their values. Application of IPA in this case is similar to the SLC Flash - parameter  $N$  is defined not based on the vulnerability of Flash to bit errors, but rather by the tradeoff between space overhead of IPA and performance benefits from its application. Thus, in our tests on OpenSSD Jasmine board with MLC Flash modules used in pSLC mode we tried extremely high values of  $N = 10$  ([10x48] IPA scheme), i.e., we allowed up to 10 consequent in-place appends on pages, and there was no meaningful increase in bit errors. However, based on the space overhead of IPA, access patterns of workloads, and performance benefits of IPA, we found the value  $N = 2$  to be a good tradeoff across all OLTP benchmarks. Another advantage of pSLC mode, is that the performance of programming and sensing operations for LSB pages is similar to their performance on SLC Flash.

On the other side, the disadvantage of this mode is the reduction of useful capacity of Flash memory by a factor of two. So, why do we propose a pSLC mode, if it will cut half

---

of the storage space, which was basically the main reason for choosing SSD based on MLC Flash? The answer is - region specific utilization of IPA. pSLC mode of MLC Flash can be utilized selectively only for certain memory regions, i.e., some Flash chips might be used in pSLC mode, while other chips in normal MLC mode. In its simplest implementation pSLC does not require any change or reconfiguration of the Flash controller. In NoFTL, where we have a direct control over the physical address space on Flash memory, we simply ignore the even numbered pages (MSB pages) and use only LSB pages for storage. The manufacturers of Flash SSD can, however, provide through specific I/O or maintenance commands further optimizations for chips used in pSLC mode. For instance, for those chips the complexity of ECC can be significantly reduced.

Thus, the pSLC mode of IPA is especially attractive for very write intensive database objects with small sized updates. Those objects can be then grouped into one or several regions, for which MLC Flash is used in pSLC mode and IPA is configured respectively. In this case the performance advantage is twofold: (i) faster I/O commands due to the use of LSB pages<sup>8</sup>; and (ii) reduced GC overhead through the IPA approach. As such hot database objects represent usually only a small part of a database (e.g., 80/20 rule), the space overheads in these regions caused by pSLC mode (half the capacity) and IPA approach (<5% of pages for delta-record area) will still be acceptable for the whole database or Flash SSD.

**odd-MLC.** In this mode the full capacity of MLC Flash memory is utilized, however, IPA is applied only for LSB pages, while MSB pages are always written in a traditional out-of-place manner. Space reservation for delta-record area (IPA page layout) is applied for both LSB and MSB pages. When the database system fetches a MSB page<sup>9</sup>, the storage manager sets the flag "No-IPA", which turns off tracking of modifications on this page, and signals that when the page will be flushed it should be written into a new location on Flash.

Programming of a MLC Flash cell can lead to program interference errors on surrounding (neighboring) cells on the adjacent wordlines. Thus, programming of cell *M* on wordline WL2 shown in Figure 7.9 might lead to small voltage shifts in cells: *G*, *H*, *I* on WL1; and *Q*, *R*, *S* on WL3. The probability that cells apart from those (e.g., cells *A*, *B*, *C*, *D*, *E*, *J*, *K*, *L*, *N*, *O*, *P*, *T*, *U*, *V*, *W*, etc.) are influenced by programming of *M* is negligibly small (see [109], [13]). Therefore, as the *write\_delta* command programs only the cells within the delta-record area, the voltage shifts can correspondingly happen only

---

<sup>8</sup>Programming of LSB pages can be as much as 6x faster than programming of MSB pages [27]

<sup>9</sup>For instance, in Flash organizations with two pages per wordline, MSB pages are pages with even physical numbers, except first and last pages in each Flash block.

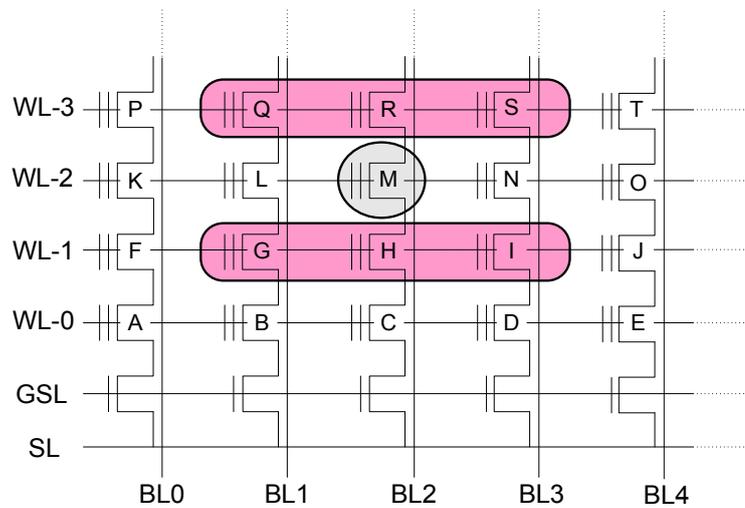


Figure 7.9.: Program interference in MLC Flash during PROGRAM operation. Cell *M* is being programmed; cells *G*, *H*, *I*, *Q*, *R*, *S* can experience program interference errors.

in cells belonging to the delta-record areas of pages on previous and next wordlines. For instance, appending a delta record to LSB page #3 on WL2 might produce interference errors on pages LSB#1, MSB#4, LSB#5 and MSB#8 in the corresponding sections in their delta-record areas. It is important to emphasize that since during IPA appends the cells with original data (main area of the page) are not programmed, no voltage shifts occur in data areas on neighboring wordlines.

But how to solve the problem with program interference errors that might occur in delta record areas of surrounding Flash pages during IPA appends? Because in *odd-MLC* mode we allow *write\_delta* commands to be performed only on LSB pages, delta-record areas of only LSB pages are relevant, while delta-record areas of all MSB pages are completely ignored. At the same time, as for LSB pages the Flash controller differentiates only between two threshold voltages, which have large distance between them, small voltage shifts there typically do not lead to bit errors. This immunity of LSB pages to small voltage shifts is similar to that of Flash cells in pure SLC Flash or MLC Flash used in pSLC mode.

Note that IPA appends are performed in both *pSLC* and *odd-MLC* modes on LSB pages in an arbitrary order. Thus, initially pages are written using an in-order programming strategy, but for the *write\_delta* command there is no restriction regarding the order of modified pages.

---

---

Summarizing the important points of *odd-MLC* mode:

1. The delta-record area is reserved on all Flash pages in a particular NoFTL region;
2. IPA appends are performed only on LSB pages, in any order (no in-order programming is needed);
3. these appends can cause small voltage shifts in delta-record areas of neighboring pages (LSB and MSB pages);
4. as delta-record areas of MSB pages are not utilized, possible bit errors there are irrelevant;
5. program interference in LSB pages generally does not lead to bit errors, because LSB pages have much higher immunity to small voltage shifts (similar to SLC Flash and pSLC mode).

While in pSLC mode the capacity of the corresponding NoFTL region is reduced by half, odd-MLC utilizes almost the full capacity of MLC Flash, except the space occupied by delta-record areas of MSB pages. On the other side, in odd-MLC mode the IPA approach can be applied only to half Flash pages. However, this does not necessarily reduce the performance and longevity benefits of IPA. *If the size of a database page is a multiple of the size of a Flash page, then even in odd-MLC mode the IPA approach can be applied for every database page.* As every database page will comprise multiple Flash pages, we just need to ensure that the Flash page that stores the last part of a database page is IPA capable, i.e., that it is a LSB page. Under the NoFTL architecture it is quite easy to fulfill this constraint as we have full control over the logical-to-physical address translation. For instance, assume that a database page is 8KB in size, and a Flash page is 4KB. Then storing every db-page in two physically consequent Flash pages would mean that the first half of a db-page is stored in an MSB page, while the second part in an LSB page. In turn, the LSB page can be updated in-place via a *write\_delta* command. Note that such a placement policy can be selected for each particular NoFTL region individually.

### **IPA on TLC Flash**

Nowadays, TLC Flash is mostly used in 3D NAND architectures. 3D NAND is a very promising technology, which allows to further increase density of memory without sacrificing performance and endurance characteristics of Flash. Firstly, this is achieved through adding multiple layers of Flash cells vertically on top of each other. It eliminates the need to shrink the size of individual cells, and thus avoid a negative impact on robustness and

---

endurance of Flash. Second, most manufacturers use in 3D TLC Flash the so-called Charge Trap Flash (CTF) technology. In contrast to the Floating-Gate Transistor (FGT), which is a common technology for SLC and MLC Flash, CTF cells have higher reliability and immunity to interference errors [83]. Unfortunately, we did not have the opportunity to evaluate IPA on TLC Flash, as we could not find any TLC-based SSD, which would allow us to re-program the Flash controller in order to implement NoFTL. However, based on the above characteristics, as well as on the statements of SSD manufacturers that 3D Flash is: "Bitline Interference Free" and "Wordline Interference Almost Free" [82], we assume that IPA can be applied on TLC Flash with 3D organization in both pSLC and odd-MLC modes without any limitations.

## 7.4. Evaluation

The detailed evaluation results of IPA were presented in [31], [36], [32] and [33]. The tests were performed with the common OLTP benchmarks TPC-C [96], TPC-B [95] and TATP [92], as well as with the popular web-oriented social graph benchmark LinkBench [4]. We run these benchmarks with different scenarios, characterized by the following factors:

- **Flash SSD.** As the underlying storage we used both the real Flash SSD - OpenSSD Jasmine board [94] (see A.1), and the Flash emulator [30]. Although OpenSSD Jasmine board allowed us to prove and evaluate the concept of IPA on real hardware, it was unsuitable for the test scenarios with high level of I/O parallelism. This is because the firmware of OpenSSD does not support Native Command Queuing (NCQ) technology, which is essential for effective processing of parallel I/O requests. That is why, in tests with high level of I/O parallelism, we used a Flash emulator.
- **Level of I/O boundedness.** To investigate the influence of reduced write-amplification and GC overhead on the system performance, we vary the configuration parameters of the database system to achieve workloads with different level of I/O boundedness. Thus, by selecting the size of the database buffer between 10% and 90% of the initial data size, we could cover the whole spectrum from strong I/O-bound to CPU-bound (in-memory database) system. Apart from the size of the database buffer, the amount of I/O requests depends also on its eviction strategy. We performed test with both eager and non-eager eviction strategies.
- **IPA mode.** We tested IPA in three available modes: SLC (Flash emulator), pSLC (OpenSSD), odd-MLC (OpenSSD).

- 
- **IPA scheme and region configuration.** After analysis of a particular workload we selected an appropriate configuration of NoFTL regions, and for relevant regions we tried different IPA schemes. With this it was possible to investigate the influence of IPA configuration parameters ( $N$  and  $M$ ) on the system performance and longevity characteristics of the Flash SSD.
  - **Benchmark parameters.** We further vary such configuration parameters of the benchmarks as scaling factor and duration.

In order to provide a short summary of this evaluation we will use below some of the results published in [31], [36], [32] and [33]. As the numbers for all OLTP benchmarks (TPC-C, TPC-B, TATP) were relatively similar, we will consider here mostly the tests with the TPC-C benchmark. The first part is dedicated to the evaluation of the IPA approach applied for tables, i.e., IPA with the offset-value format of delta records. The second part describes the results for IPA-IDX.

Please note that in this work we use another notion of IPA scheme as compared to the above publications. In [31], [36] and [32] IPA parameter  $M$  describes the maximum number of changed bytes in tuple data (net data) of database page that can be encoded in one delta record. In contrast, in this work  $M$  corresponds to maximum size of delta record itself. Thus, for instance, the notion of IPA scheme [2x3] used in [31] would correspond to the scheme [2x48] in this work (16 offset-value pairs each 3 bytes in size: 3 pairs to encode changes on tuple data, and 13 pairs to encode changes in page metadata). We have changed the notion of  $M$  in order to unify it for both IPA variants: IPA-basic and IPA-IDX.

#### 7.4.1. Basic IPA

To figure out if application of an IPA approach is reasonable for a particular workload, which type and scheme of IPA, and what region configuration would be optimal, an analysis of this workload must be performed. This analysis can be performed either offline or online. We have implemented a tool for an offline analysis of a workload, the *IPA advisor*, which was presented in [36]. The discussion and initial implementation of workload analysis is presented in Chapter 8.

One of the basic criteria for efficiency of the IPA approach for a particular workload (or a separate database object) is the dominance of small sized updates. For this, we typically collect and analyze cumulative distribution functions for update sizes of database pages. Such example for a TPC-C workload is presented in Figure 7.10. It is important to note, that under *update size* of a page we assume total number of changed bytes on a page upon

---

its flushing to storage, i.e. cumulative size of all changes performed on a page while it was in the database buffer.

Figure 7.10 shows the cumulative distribution of update sizes in the TPC-C benchmark, which was running for two hours on the Shore-MT storage engine<sup>10</sup> [42]. The buffer size varies between 10% and 90% of the initial data size. As we can see, across all configurations 75% of all update I/Os modify 16 or less bytes on a 4KB page. These numbers can be explained by simple analysis of the TPC-C transaction profile and details about default behavior of buffer and log managers in the Shore-MT storage engine.

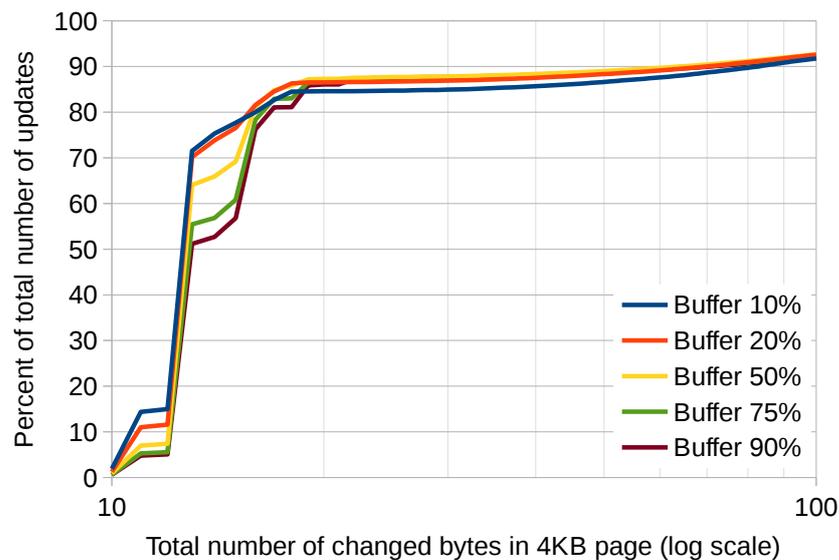


Figure 7.10.: Cumulative distribution of update sizes in TPC-C under eager eviction strategy<sup>11</sup>.

The most write-intensive database object in TPC-C is the *Stock* table (see analysis in [55]). This table is modified by *New Order* transaction in such a way, that per transaction, on average, ten records are changed. Each such record update assumes modification of three numeric attributes. These attributes are modified through adding a small delta (typically  $\leq 10$ ) to the previous value, which in the general case results in only 3 bytes per record being changed. Additionally, each such change requires update of a page's metadata (e.g., LogSN), which in Shore-MT results on average in other 12 bytes being modified. Records to be modified are selected randomly, and thus are usually located in

<sup>10</sup><https://sites.google.com/site/shoremt/>

<sup>11</sup>CDF of update sizes only in net data (tuple data) is provided in [31].

---

---

different database pages. Therefore, each *NewOrder* transaction changes on average ten database pages of *Stock*, and within each page 16 or less bytes. Such small-sized updates are also typical for *Customer*, *Warehouse* and *District* tables in this benchmark.

But why do those changes not accumulate before the pages are written out to storage? The reason for this lies in the configuration of buffer and log managers in Shore-MT. Buffer manager in Shore-MT uses by default eager eviction strategy. It assumes that background flusher threads start to write out dirty pages from the buffer pool when their amount is larger than 12,5% of the buffer size. Moreover, Shore-MT utilizes the so-called eager log space reclamation. Under this policy, when log space is filled to 25% and more, the background flushers start to write out dirty pages, which correspond to the oldest log segments. These strategies are primarily aimed at reducing the time needed to recover the database after a crash [85].

Both these strategies determine what pages are flushed to stable storage and when. It is clear that under these default configurations of buffer and log managers the I/O load produced by the database will not decrease with increasing buffer size. In contrast, buffer sizes of 75% and 90% will make the workload to be CPU-bound, which will significantly increase transactional throughput. This will result in high rate of modifications on buffered pages and high rate of new log entries. Thus, eager buffer eviction and eager log space reclamation will flush more pages to storage as compared to configurations with lower buffer sizes (see line [*Host Write I/O*] in Table 7.2). It also explains why modifications on *STOCK* table rarely can be accumulated. As this table is the largest in the database, and modification on it touch random pages, most of these pages are seen as "cold" by the buffer manager and flushed (not necessarily evicted) to storage.

Such distribution of update sizes as shown in Figure 7.10 makes the decision regarding the proper IPA configuration quite easy. Remember that every changed byte in the page is encoded with three bytes in a delta record with offset-value pair format. Thus, by choosing  $M$  parameter of IPA scheme (maximal size of delta record) to be 48 bytes ( $16 \cdot 3$ ), more than 75% of all updates can fit into a single delta record. This, however, does not mean that 75% of all update I/Os can be performed using in-place appends. The fraction of *delta\_writes* is further influenced by parameter  $N$ , write pattern of the workload, and selected IPA mode (e.g., pSLC or odd-MLC). In the evaluation tests presented below parameter  $N$  equals 2. Our experiments [31] on the real MLC NAND Flash have shown that higher values of  $N$  are also meaningful and do not cause reliability issues. But as this parameter depends on the physical characteristics of Flash memory, and we were not able to perform evaluations on different types of Flash memory, we have chosen the lower value of  $N$ , which should be applicable for all Flash types.

Table 7.1 presents the evaluation results of the IPA approach on OpenSSD Jasmine board running the TPC-C benchmark for two hours. The column [*Traditional Absolute*]

	0x0 Absolute	2x48 Absolute pSLC	2x48 Relative pSLC [%]	2x48 Absolute odd-MLC	2x48 Relative odd-MLC [%]
Out-of-Place Writes vs. In-Place Appends			49/51		70/30
Host Reads	4,977,335	6,390,032	+28	5,671,727	+14
Host Writes	1,347,515	1,768,552	+31	1,524,552	+13
GC Page Migrations	422,753	79,718	-81	230,497	-45
GC Erases	7,151	2,862	-60	3,819	-47
GC Page Migrations per Host Write	0.3137	0.0451	-86	0.1512	-52
GC Erases per Host Write	0.0053	0.0016	-70	0.0025	-53
Tx. Throughput	25	37	+46	28	+11

Table 7.1.: TPC-C benchmark on OpenSSD: traditional approach vs. IPA in pSLC and odd-MLC modes. Source: Hardock et al. [31]

shows the absolute numbers of performed operations and transactional throughput for the configuration, where IPA approach is not utilized, and thus every host write results in an out-of-place update on Flash storage with the potential overhead of the garbage collection (page migrations and block erases). Columns *[IPA [2x48] ...]* correspond to the tests with enabled IPA approach, which is configured with the *[2x48]* scheme either in *pSLC* or *odd-MLC* mode. Here, the *[... Relative]* column shows the relative difference (in %) compared to the configuration without IPA approach (*[Traditional Absolute]*).

As we can see from Figure 7.1 through utilization of IPA in pSLC mode 51% of all database writes can be performed using in-place appends, while in odd-MLC mode this value decreases to 30%. Note that here we consider the fraction of all database writes, which include both update writes and writes of new pages; while the CDF in Figure 7.10 considers only update writes. To clearly evaluate the difference between two IPA modes, we have selected the size of database page to be equal to the Flash page size. That is why, the lower fraction of IPA writes in odd-MLC mode is expected, as in that case application of IPA is possible only for database pages that are stored in LSB Flash pages (every second Flash page).

The direct implication of performing this percentage of writes using IPA is the reduced overhead of the garbage collection. Thus, the number of page migrations per host write was reduced by 86% in pSLC and by 52% in odd-MLC mode as compared to the traditional approach. Consequently, reduced GC overhead results in lower I/O response times, which

in turn, depending on the level of I/O boundedness of the system, has an impact on the performance. Thus, due to the hardware limitations the tests on OpenSSD had small database buffer (about 1.5% of the total database size), which made the system I/O-bound. That is why, in pSLC mode the transactional throughput has increased in case of IPA by 46%, and in odd-MLC mode by 11%. The higher increase of throughput in pSLC mode has two reasons. First, as more writes could be performed via *write\_delta* command, the GC overhead and I/O response times are lower as compared to odd-MLC mode. Second, the average response time of write I/Os is further decreased as in pSLC mode only LSB Flash pages are utilized, which have significantly lower program latency.

Another important advantage of the IPA approach is the improved longevity of the Flash SSD. Thus, as wear of SSD is directly correlated with the number of performed program-erase cycles (P/E cycles), the reduction of erase operations per host write by 70% and 53% for pSLC and odd-MLC modes would at least result in doubling the lifetime of the Flash storage.

	Buffer 20%		Buffer 50%		Buffer 75%		Buffer 90%		
	0x0 Abs.	2x48 Rel.[%]							
Out-of-Place Writes vs. IPAs		51/49		54/46		56/44		56/44	
Host Reads (4KB)	25,120,600	25.89	3,275,550	11.44	614,639	4.43	279,258	0.44	
Host Writes (4KB)	39,530,323	16.25	51,570,434	9.41	62,627,983	9.81	64,345,377	0.54	
GC Page Migrations	27,886,888	-36.00	37,521,497	-31.74	46,874,908	-29.08	47,558,375	-28.51	
GC Erases	1,018,624	-39.51	1,357,349	-37.67	1,676,376	-34.83	1,713,844	-33.77	
GC Page Migrations Per Host Write	0.7055	-44.95	0.7276	-37.61	0.7485	-35.42	0.7391	-28.89	
GC Erases per Host Write	0.0258	-47.97	0.0263	-43.03	0.0268	-40.65	0.0266	-34.13	
Response Time [ms]	READ I/O	0.77	-31.60	3.90	-31.07	8.44	-21.34	9.10	-2.89
	WRITE I/O	0.53	-21.36	0.53	-19.17	0.54	-17.88	0.53	-15.38
Tx. Throughput	1,001	15.42	1,480	6.28	1,984	1.22	2,191	0.21	

Table 7.2.: TPC-C: traditional (no IPA [0x0]) vs. [2x48] schemes with large buffer pools (eager eviction). Source: Hardock et al. [31]

To analyze the effect of IPA on systems with high level of I/O parallelism and larger sizes of database buffer we have also performed tests using a real-time Flash emulator

---

---

[30] on a server<sup>12</sup> with 128GB RAM. Flash Emulator was configured to comprise 16 SLC NAND chips making in total 50GB Flash SSD. Table 7.2 shows aggregated evaluation results for the TPC-C benchmark with scaling factor 150 (roughly 30GB) running for two hours on this testbed. We have varied the database buffer size from 20% to 90% of the initial database size, and for each such configuration performed three tests without IPA approach and three tests with enabled IPA. IPA was in all these tests configured with scheme [2x48]. Note that in all tests I/O measurements are related only to the database data, and do not include the I/O part that corresponds to database logs. Moreover, we store database logs in an in-memory partition, which further helps to separate database performance from the characteristics of the storage device used for logs.

With buffer sizes of 75% and 90% we can evaluate scenarios close to in-memory database systems. In these tests transactional performance becomes CPU-bound, and almost independent from the performance of the I/O subsystem. That is why, the reduced garbage collection overhead (35% and 29% less page migrations), and as a consequence of this lower response times for write requests (18% and 15%) have almost no impact on the system performance. For smaller buffer sizes the improvement of transactional throughput was varying between 6% and 15%.

In these tests we have used the default *eager* eviction strategy of the buffer manager, as well as the default *eager* space reclamation policy of the log manager. As already mentioned before, these strategies cause the I/O rate of write request to increase with the growth of buffer size, and consequent increase of transactional throughput. This can be clearly seen by comparing the absolute number of performed host writes (row [*Host Write I/Os*]) for configurations with different buffer sizes. These default policies of log and buffer managers explain also why even with 90% database buffer 44% of all write requests can be performed using IPA approach with [2x48] scheme. This is because before pages can accumulate multiple small updates, they are flushed to the stable storage (but not evicted from the buffer).

Although the positive impact of IPA on system's performance in CPU-bound scenarios is low, the reduction of performed erase operations by 34% even with 90% buffer would significantly improve the longevity characteristics of Flash SSDs.

To provide comprehensive evaluation results we also tested the behavior of IPA approach with non-eager eviction strategy of the buffer and turned-off default space reclamation of the log manager. By doing so, modified pages will rarely be flushed to storage, and thus they will be able to accumulate multiple updates. Obviously, under these conditions we need to re-evaluate the appropriate configuration scheme for the IPA approach. For this let us look at the corresponding distribution of update sizes presented in Figure 7.11.

---

<sup>12</sup>Intel Xeon server with 32 E7-4830 CPU core, each 2.13 GHz

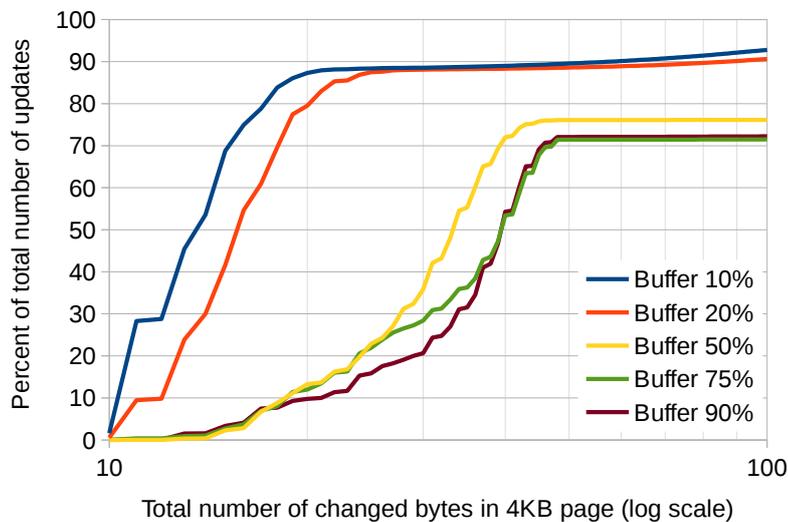


Figure 7.11.: Cumulative distribution of update sizes in TPC-C under non-eager eviction strategy<sup>13</sup>.

For smaller buffer size (20%) the accumulation effect is less visible. Thus, about 85% of all update writes modify 22 or less bytes on a 4KB page. Having a buffer capable of caching about 50% of data would require a delta record that can encode at least 42 modified bytes in order to cover 75% of all updates. Large buffers (75% and 90%) need a delta record sized for about 52 bytes of changed data to account for about 70% of updates. As every changed byte requires 3 bytes to be encoded in delta record with offset-value format, the IPA configurations we selected for these tests are: [2x66] for buffer size 20%; [2x126] for 50% buffer; and [2x156] for buffers 75% and 90%. Thus, for the largest buffer the total reserved size per page for delta-record area is almost 8% of a 4KB page.

Table 7.3 presents the evaluation results for the TPC-C benchmark with turned-off default eviction and log-space reclamation policies and varying buffer sizes. As we can see, for buffer sizes of 75% and 90% the total number of writes performed by the database within two hours is almost 8 times less as compared to the configuration with default eager eviction and log-space reclamation policies. With the [2x156] IPA scheme more than 33% of these writes have been performed using *write\_delta* command. This allowed us to reduce GC overhead by more than 22% and response times of write requests by 8%. The impact of these savings on the transactional throughput is low (1-3%) due to the

<sup>13</sup>CDF of update sizes only in net data (tuple data) is provided in [31].

	Buffer 20%		Buffer 50%		Buffer 75%		Buffer 90%		
	0x0 Abs.	2x66 Rel.[%]	0x0 Abs.	2x126 Rel.[%]	0x0 Abs.	2x156 Rel.[%]	0x0 Abs.	2x156 Rel.[%]	
Out-of-Place Writes vs. IPAs		44/56		51/49		63/37		67/33	
Host Reads (4KB)	39,383,139	16.29	4,462,332	5.33	676,580	0.50	265,543	3.19	
Host Writes (4KB)	28,591,074	20.16	11,767,036	4.43	8,486,996	3.46	8,802,867	3.25	
GC Page Migrations	16,595,099	-40.27	5,027,818	-30.98	2,877,442	-20.07	2,982,080	-19.52	
GC Erases	670,807	-46.07	227,215	-36.13	142,339	-21.63	148,312	-19.10	
GC Page Migrations per Host Write	0.5804	-50.29	0.4273	-33.91	0.3390	-22.75	0.3388	-22.06	
GC Erases per Host Write	0.0235	-55.12	0.0193	-38.84	0.0168	-24.25	0.0168	-21.65	
Response Time [ms]	READ I/O	0.30	-19.46	0.41	-16.95	0.43	-19.30	0.64	-11.53
	WRITE I/O	0.49	-21.56	0.43	-12.33	0.43	-7.87	0.56	-8.21
Tx. Throughput	1,291	6.96	2,220	3.26	2,360	1.06	2,259	3.67	

Table 7.3.: TPC-C: traditional (no IPA) vs.  $[2 \times M]$  schemes with large buffer pools (non-eager eviction). Source: Hardock et al. [31]

CPU-boundedness of the workload. However, being able to save more than 20% - 61% of erase operations in these tests shows a clear benefit of IPA for the longevity characteristics of Flash SSD. It is worth to note, that in these tests with turned-off default eviction and log-space reclamation policies the total recovery time of the database after simulated crash has increased by about 4 times as compared to the corresponding tests with default configuration of buffer and log managers.

More IPA evaluation results for TPC-C and other popular benchmarks were presented in [31, 36, 32].

#### 7.4.2. IPA for Indexes

Figure 7.12 shows the distribution of update sizes for a typical B-Tree index in the TPC-C benchmark. It is a composite index on three attributes of *NewOrder* table, namely, on foreign keys *no\_w\_id*, *no\_d\_id* and *no\_o\_id*. The index is modified only in an append-based manner, i.e., adding a new record to *NewOrder* relation causes also a new index entry to be inserted.

Applying the basic IPA approach for that database object would require a single delta-

record be at least 450 bytes in size ( $[150 \text{ bytes}] * [3 \text{ bytes per offset-value pair}]$ ), and this schema would theoretically catch at most 50% of all updates. In order to cover 75% of index updates, the size of delta record must be increased to about 840 bytes ( $280 * 3$ ). Thus, for storing two delta-records, IPA must reserve about 22% or 41% of a 4KB page for 50% and 75% coverage of index updates, respectively. This example clearly shows that basic IPA approach with delta records in offset-value format is suboptimal for B-Tree indexes due to insufficient coverage and/or extremely large size of delta-record area.

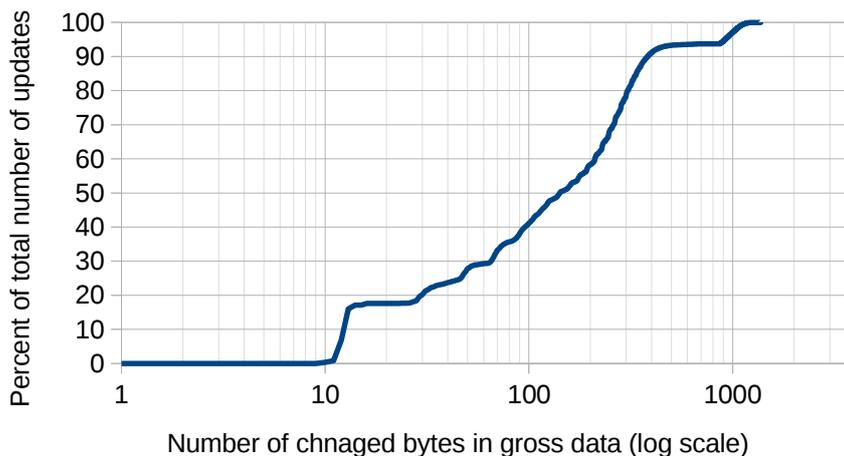


Figure 7.12.: CDF of update sizes for index on NewOrder table in TPC-C benchmark.

The theoretical coverage of index updates mentioned above serves only as a first, narrow guideline. For instance, coverage of 75% does not mean that 75% of all write I/Os to that particular database object will be performed using an IPA approach. In reality, the fraction of in-place writes will typically vary between 40% and 50%. There are several reasons for this. First, due to the accumulation of updates during the time an index page stays in the database buffer, the total size of corresponding delta-records might overflow the remaining space in delta-record area. In this case, the page will be written to the Flash storage in an out-of-place manner. Second, the limited amount of times the page can be written consequently using an IPA approach (e.g.,  $N=2$ ) further reduces the total fraction of delta writes.

We have implemented IPA-IDX as an extension to the basic IPA approach in Shore-MT database engine. IPA-IDX supports selective application, i.e., it can be applied for selected regions only, and within those regions specifically for selected database objects.

The first step in evaluation of IPA-IDX is to determine the right size of delta-record

---

area. In Shore-MT, log records for insert and delete operations on a B-Tree have identical structure. Each such log record consists of metadata, key and element (value). Metadata is 56 bytes in size and comprises the following fields: total length, type, category, page id, transaction id, volume id, page tag, object id, previous lsn, lsn, page id of root, slot index, key length and element length. Element is a tuple identifier and is 16 bytes in size. Size of the key depends on a concrete B-Tree index. For instance, for the index on *New Order* table in Figure 7.12 the key size is 12 bytes, because the key consists of three numeric values. Since the majority of indexes in TPC-C and TPC-B benchmarks that we used for evaluation are composite indexes over 2 to 4 numeric fields, the corresponding log records are 80 to 88 bytes. Based on these statistics we took for all our experiments with IPA-IDX the size of a delta-record equal to 270 bytes, which is enough to accumulate 3 log records (i.e., 3 changes on an index page). Further, we limit the amount of allowed consequent IPA writes to 2, which results in a  $[NxM]=[2x270]$  scheme for selected B-Tree indexes. Therefore, delta record area can accumulate in total up to 6 delta records (i.e., log records). In case of 4KB database pages, the space overhead for each particular index that uses IPA-IDX with this scheme is 13%. This is about 2 to 3 times less than the space overhead in case of delta records with offset-value format. Furthermore, this scheme offers 100% theoretical coverage for all possible changes on an index page, as absolutely all log records can be potentially stored in a delta record.

Table 7.4 presents the performance results of IPA-IDX for the TPC-C benchmark. In this test we enabled IPA-IDX approach for five B-Tree indexes: NO\_IDX(no\_w\_id, no\_d\_id, no\_o\_id), O\_IDX(o\_w\_id, o\_d\_id, o\_id), O\_CUST\_IDX(o\_w\_id, o\_d\_id, o\_c\_id, o\_id), OL\_IDX(ol\_w\_id, ol\_d\_id, ol\_o\_id, ol\_number), S\_QUANTITY\_IDX(s\_w\_id, s\_i\_id, s\_quantity). As the first four indexes are modified only in an append-only manner, we added an additional S\_QUANTITY\_IDX index to have an example of B-Tree in which records are changed (i.e., removed and added). All these indexes together occupy about 30% of database space. Applying  $[2x270]$  scheme for them results in total space overhead of IPA-IDX of less than 4% (13% increase in size for 30% of database size).

This configuration allowed us to perform 40% of all database writes using IPA-IDX approach, while within five selected indexes the fraction of *delta\_writes* is about 65%. Although the improvement on transactional throughput was only 9% as compared to the scenario without IPA-IDX, a noticeably larger benefit of our new approach is seen in reduction of GC overhead. The average amount of page migrations per one 4KB host write was reduced by 40%, while the number of erases by 44%.

An example of simultaneous use of basic IPA and IPA-IDX is shown in Figure 7.5. Here, we first applied the basic IPA with  $[NxM]=[2x48]$  scheme to the *Account* table in TPC-B benchmark. This has improved transactional throughput by 12%, and decreased garbage collection overhead by 37%. In this test scenario we have introduced an additional index

Flash SSD 50GB TPC-C, SF = 150 2 hours		Baseline No IPA	IPA-IDX [2x270]	Relative [%]
Out-of-Place Writes vs. In-Place Appends				60/40
Host Reads (4KB)		56,284,772	64,457,410	+15
Host Writes (4KB)		56,977,326	62,821,498	+10
GC Page Migrations		18,117,128	12,000,279	-34
GC Erases		1,173,341	718,641	-39
GC Page Migrations per Host Write		<b>0.318</b>	<b>0.191</b>	<b>-40</b>
GC Erases per Host Write		<b>0.021</b>	<b>0.011</b>	<b>-44</b>
I/O Response Time [ms]	READ	0.37	0.30	-19
	WRITE	0.44	0.38	-14
Tx. Throughput		<b>557</b>	<b>610</b>	<b>+9</b>
Space overhead				+4.0

Table 7.4.: Evaluation results of IPA-IDX on TPC-C benchmark. Source: Hardock et al. [33]

on *History* table ( $H\_IDX[h\_a\_id, h\_date]$ ). Since on every transaction a new entry is added to the *History* table, this index becomes write-intensive, and accounts for about 45% of database writes (another 54% of write I/Os target pages of *Account* table). Thus, adding this index almost doubles the write load of the benchmark.

In the next test scenario, in addition to basic IPA, we enabled IPA-IDX for new index, and used the same  $[NxM]=[2x270]$  configuration scheme as in case of the TPC-C benchmark. With this, the total fraction of write I/Os performed using in-place appends (*delta\_writes*) has increased to 64%. Consequently, the GC overhead decreases by further 30%, and is in total 66%-68% less as compared to the baseline where IPA is not applied. Transactional throughput improves with the basic IPA by 12%, and with both IPA variants by 28%. The total space overhead of IPA and IPA-IDX together is about 2%.

As we can see, IPA-IDX provides similar performance and longevity advantages for indexes as the basic IPA does for tables. Ability to use IPA-IDX for certain regions and objects, while simultaneously applying basic IPA for other regions allows to lower the

Flash SSD 50GB TPC-B, SF = 2000 2 hours		Baseline No IPA	IPA [2x48]	Relative [%]	IPA [2x48] IPA-IDX [2x270]	Relative [%]
Out-of-Place Writes vs. In-Place Appends				62/38		36/64
Host Reads (4KB)		41,901,849	47,943,305	+14	55,618,681	+33
Host Writes (4KB)		43,523,639	49,535,369	+14	57,262,170	+32
GC Page Migrations		23,767,131	17,139,595	-28	10,422,430	-56
GC Erases		1,030,365	726,637	-29	460,509	-55
<b>GC Page Migrations per Host Write</b>		<b>0.546</b>	<b>0.346</b>	<b>-37</b>	<b>0.182</b>	<b>-67</b>
<b>GC Erases per Host Write</b>		<b>0.024</b>	<b>0.015</b>	<b>-38</b>	<b>0.008</b>	<b>-66</b>
I/O Response Time [ms]	READ	0.52	0.42	-19	0.32	-39
	WRITE	0.52	0.43	-16	0.37	-29
<b>Tx. Throughput</b>		<b>3668</b>	<b>4119</b>	<b>+12</b>	<b>4677</b>	<b>+28</b>
Space overhead				+0.6		+2.1

Table 7.5.: Evaluation results of IPA-IDX on TPC-B benchmark. Source: Hardock et al. [33]

overall GC overhead by over 60%, while keeping the total space overhead of the IPA approach below 5%.

It is important to mention, that in the current IPA-IDX implementation we store in delta-records complete and unchanged log entries used in Shore-MT. We did this because of simplicity in implementation. However, most of the metadata stored in a log record is irrelevant for IPA (e.g., category, transaction id, page id, root id, etc.). Removing this data from delta records would save 40 bytes per each log entry, e.g., 50 bytes per "lean" log record would be needed instead of 90 bytes). Thus, the applied above [2x270] scheme might be replaced in case of this optimization with the scheme [2x150]. This would reduce the space overhead of IPA-IDX by 44%.

---

## 8. Auto-Tuning

---

Chapter 6 discusses one of the most important and fundamental features of the NoFTL architecture - the concept of configurable Flash storage. The evaluation results at the end of the chapter, and in our previous work (e.g., [34], [35], [29]) show the efficiency of this approach with regards to performance and longevity characteristics of Flash storage. However, there are three important questions about utilization of multi-region configurations, which were left untouched in Chapter 6. These questions are:

1. How to find the optimal data placement configuration scheme for a particular workload and system?
2. How to transform a database from one configuration scheme to another?
3. How to guarantee even wear-out of the whole Flash storage in a multi-region configuration?

### 8.1. Selection of Multi-Region Configuration

The process of selecting a proper NoFTL configuration is based on an analysis of database data, and consequently finding an optimal match between this data, units of Flash storage (e.g., dies/chips) and different Flash management schemes. This task is to a certain degree similar to the general problem of finding an optimal data placement for given database objects on a set of distributed storage units. It is known to be a NP-hard [8], and the typical approach of solving such problems is based on heuristics, e.g., [65], [5]. The search for a data placement configuration in NoFTL is even more complex, as the properties of storage units (in our case regions) are not known in advance. Thus, number of chips (i.e., capacity, parallelism), Flash management scheme and NAND mode for every region should be also determined.

As a small part of this research work we also have tried to develop a heuristic for finding an optimal data placement configuration. Although the algorithm is demonstrated to provide good results for TPC benchmarks, we consider its scope limited, and see the

---

development of more advanced approaches as a matter of future research in the context of self-driving DBMS.

In this work and in the current implementation of the NoFTL prototype the data placement configuration must be selected manually by the database administrator. However, this process is supported by the proposed *NoFTL Advisor*. The advisor performs two major tasks: (i) it collects all relevant information about the workload; and (ii) it aggregates and presents this information to a database administrator in a suitable form. There are two main views in the advisor. The *summary* view gives an outlook about the whole database and workload. It allows the DBA to see what database objects dominate the workload based on their sizes, I/O read and write loads (Figure 8.1). Moreover, this view contains information about the transactional profile of the workload. The *per-object* view provides the detailed information about the selected database object. The typical information here is: (i) type of database object; (ii) I/O counts, sizes, response times; (iii) distribution of I/O requests over the address space occupied by this object; and (iv) distribution of update sizes. One part of this information was already available in the DBMS; for another part we integrated additional statistics, which are updated online<sup>1</sup>; other information (e.g., access skew, update sizes) is collected by the offline analysis of database log files.

The NoFTL advisor can also provide certain suggestions regarding some options of Flash management for each particular database object. For instance, based on the clear dominance of a certain access pattern (e.g., read-mostly, append-only) the framework suggests an appropriate address translation scheme (e.g., BLM, FASTer, etc.). Moreover, the advisor can provide hints about suitable IPA configurations (see our demonstration of IPA advisor in [36]). From our practical experience based on choosing an appropriate configuration for several common benchmarks (TPC-C, TPC-B and TATP), we can say that this process of manual search assisted by the advisor is quite easy and straightforward. The summary view gives a good initial picture about the possible regions, while the detailed information for each object allows us to select the appropriate set of Flash management parameters (see Chapter 6.1.2).

## 8.2. Change of Multi-Region Configuration

Another important question is how to change the database from one NoFTL configuration (be it a single or multi-region configuration) to another? Apart from applying the initial configuration, in some cases it might be necessary to change the existing configuration later on. One such reason is the evolution of the workload. If for instance, access patterns or temperature of large database objects significantly changes, it might become reasonable

---

<sup>1</sup>Integrated statistical counters have been striped by core to avoid CPU contention during their update.

```
IO STATISTICS OF STORES:
```

ST-ID	WTR_US	WTW_US	ETR_US	ETW_US	RTR_US	RTW_US
0	334	51	81	436	415	487
3	0	21	74	455	75	477
5	3	0	179	0	183	0
6	18	14	72	472	91	487
8	118	0	78	0	196	0
9	453	16	82	468	536	484
11	153	123	91	276	245	399
12	155	111	89	279	244	390
13	0	15	0	448	0	463
15	312	17	337	493	650	510
17	561	10	89	447	650	458
18	454	17	86	469	541	487
20	693	10	90	447	784	457
21	671	10	91	611	763	622
22	405	15	87	446	492	462
24	657	10	92	443	750	453
25	159	0	88	0	248	0
27	141	0	85	0	226	0
28	412	16	83	465	496	481
30	134	89	84	267	219	357

```
*****
***** Region statistics *****
*****
```

RG-ID	WTR_US	WTW_US	ETR_US	ETW_US	RTR_US	RTW_US
0	414	16	83	465	498	481
1	137	108	85	276	222	384
2	655	10	91	488	746	499

```
*****
***** Transaction statistics *****
*****
```

General MEAN, VARIANCE, STDEV, SAMPLE\_VARIANCE, SAMPLE\_STDEV for each transaction type

```
TRX #1:
MS: μ=      25.95; VAR=      25981.46; STDEV=      161.19; SAMPLE_VAR=      25981.64; SAMPLE_STDEV=
US: μ=      25948.61; VAR= 25981464672.42; STDEV=      161187.67; SAMPLE_VAR= 25981635167.89; SAMPLE_STDEV=
```

Figure 8.1.: Excerpt from an example of *summary* output of NoFTL advisor.

to adapt the region configuration to reflect these changes. Another reason for switching the NoFTL configuration might be the changes in the database schema (e.g., deletion or creation of database objects).

Our solution for this is encapsulated in a special database module called *migrator*. The main characteristic properties of the migrator are the following.

- Migration from one NoFTL configuration to another happens in parallel to the normal database work, i.e., online, without stopping or pausing the workload. Obviously, the prerequisite for this property is that at any point in time during the migration the database remains in a consistent state. Although, for some systems it would be possible to turn the database for short time offline, and then switch to a new

---

configuration. For the majority of production systems even few minutes of downtime are unacceptable. That is why we developed only the online version of the migrator.

- Migration itself can be restarted. For instance, if during the transformation to another configuration the system crashes, after the start and normal recovery the migration process is resumed from the point where it was when the crash occurred. Moreover, it is possible to pause and resume the migration job manually.
- Duration of the migration process can be planned and configured. Migration is based on copying relevant data from one place on Flash memory to another. Although this process happens in the background, it still influences the I/O throughput of the storage, and thus (depending on the I/O-boundedness of the system) might negatively impact the overall performance of the system (e.g., transactional throughput). By configuring the number of background threads that perform migration in parallel, it is possible to select the most appropriate tradeoff between the migration time and I/O overhead during that time. For instance, if the migration task is scheduled for the time when the database is under low load (e.g., service times, during the night, etc.) the number of migration threads can be increased to speed-up the change of configuration. If, however, there is no such opportunity, the migration process should be configured with only few threads, which would result in very small background I/O overhead. It is always possible to pause the migration process and resume it with the modified number of background threads.
- Migration touches only relevant database objects. Upon triggering a background jobs, the migrator performs an analysis of both configurations (current and target). Based on this it determines which database objects need to be migrated to new configuration. For instance, if a certain object is deleted from the database scheme, but the overall region configuration remains, the migration process is a simple modification of metadata describing the configuration (no data movement is needed). Similarly, if a newly created db-object should be added to the existing region without changing its properties (e.g., size, mapping scheme, etc.), no data transfer is necessary. Thus, data that belongs to regions which remain unchanged in the target configuration is not touched during the migration process.

Let us now look deeper into the migration process. Consider a simplified example in Figure 8.2. Here 5 database objects and 8 available Flash chips are clustered into two regions in the current NoFTL configuration. Region #1 should be split into two regions in the target configuration (regions #1 and #3), while region #2 remains unchanged.

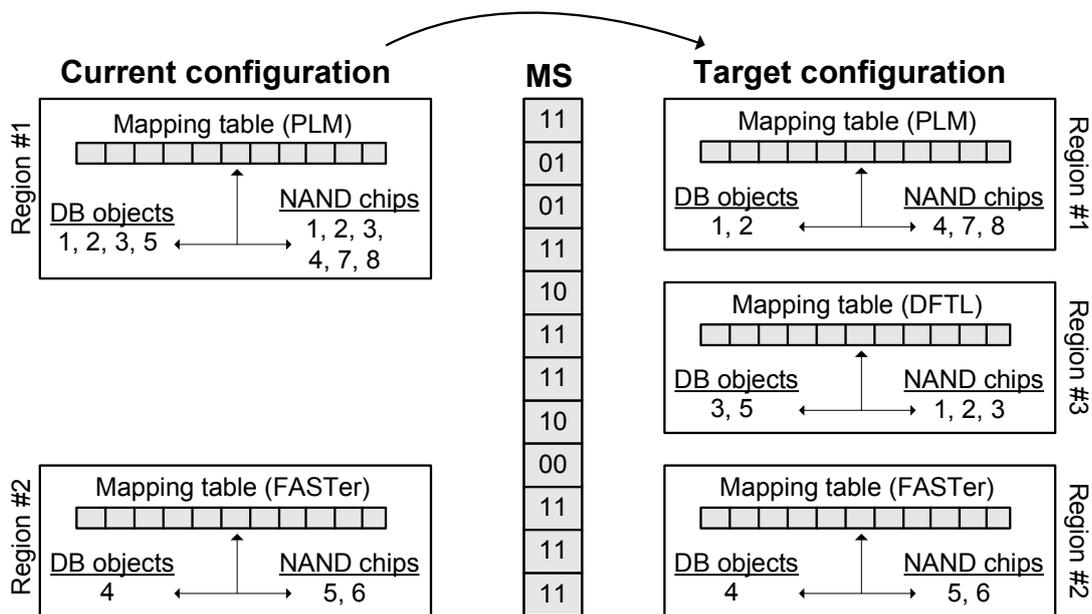


Figure 8.2.: Example of a migration to target NoFTL configuration.

At the beginning, Migrator creates an auxiliary, transient data structure to track the migration process - *MS* (migration status) vector with  $N$  elements, where  $N$  is the total number of database pages. Each element has size of two bits (e.g., 1TB database with 8KB pages would produce a 32MB *MS* vector), which determine the current state of the corresponding database page (LPN = offset in the vector). The states are the following: (i) 00 - page is referenced by the current configuration and is neither in read I/O, nor it is currently migrating; (ii) 01 - page is referenced by the current configuration and it is currently in read I/O; (iii) 10 - page is currently migrating to target configuration; (iv) 11 - page is referenced by target configuration, i.e., page was migrated to target NoFTL configuration. The necessity of two bits per page ensures that pages can always be correctly located by the database during the migration process.

**Read I/O during Migration.** When DBMS issues a fetch request for a certain page (see Algorithm 8.1) it first checks its current status. If the status is 11, the I/O is processed by consulting the target configuration. Otherwise, the DBMS tries to atomically change (compare-and-swap operation (*cas*)) the status to 01 assuming the previous value being 00. If this *cas* succeeds, the I/O is processed by consulting the current configuration, and

at the end of I/O the status is atomically set to 00. If, however, the *cas* did not succeed, then the page had another (other than 00) status upon comparison. It might be that it is either in migration (10), or it was already migrated (11), or another thread is currently performing a read I/O on this page. In all these cases we simply try to repeat the whole procedure. Thus, in rare cases when the DBMS tries to fetch the page that is currently migrating, the system will wait until the migration is finished (typically a delay of one COPYBACK operation on NAND, e.g., 250µs), and then fetch the page by consulting the target configuration.

---

**Algorithm 8.1** Reading a page during migration process

---

**Input:** *lpn* - logical number of page to be read; *buffer* - buffer to place read page

**Output:** return code of read operation

```

1: function READPAGE DURING MIGRATION(lpn, buffer)
2:   curStatusRef ← GETPAGE MIGRATION STATUS REF(lpn)
3:   if curStatusRef = 11 then                                     ▷ already migrated
4:     return READPAGE FROM CONF(targetConf, lpn, buffer)
5:   else
6:     newStatus ← 01                                             ▷ in READ I/O
7:     expStatus ← 00                                             ▷ neither in READ nor in WRITE I/O
8:     statusChanged ← CAS(curStatusRef, newStatus, expStatus)
9:     if statusChanged = True then
10:      ret ← READPAGE FROM CONF(curConf, lpn, buffer)
11:      newStatus ← 00
12:      expStatus ← 01
13:      statusChanged ← CAS(curStatusRef, newStatus, expStatus)
14:      if statusChanged = True then
15:        return ret
16:      else
17:        return ERROR_WRONG_STATUS
18:      end if
19:    else
20:      return ERROR_PAGE_IN_USE
21:    end if
22:  end if
23: end function

```

---

---

**Write I/O during Migration.** Upon a write request the DBMS also first consults the *MS* vector (see Algorithm 8.2). If the current status of the corresponding logical address is 11, the write request is further delegated to the target configuration. Otherwise, the DBMS tries to perform a *cas* operation to set the element to 10 assuming the current value being 00. If this operation succeeds (i.e., the address becomes marked as "in migration"), the page is written to a physical location determined by the target configuration, then the status is changed to 11 ("migrated"), and afterwards (if necessary) the old page is invalidated (physical address is taken by consulting the current configuration). If the *cas* operation failed (i.e., the status of the address was other than 00), then we repeat the above procedure.

One important point in the procedure of write requests is that they also contribute to the migration process, which is happening in the background. All pages flushed by the DBMS (e.g., due to buffer eviction, or checkpoint) are written to storage with respect to the target configuration, thus they are automatically migrated at no additional cost. For instance, if the workload tends to overwrite large portions of database data within a short time, then it makes sense to use only few migration threads. In this case, the migration will be performed primarily by normal DBMS writes (with no additional migration costs), while few background threads will migrate the remaining parts of data.

**Migration Threads.** Before a background migration process begins, the Migrator checks for pages that do not require data movement. For instance, these are pages of database objects that belong to regions, which remain unchanged between current and target configurations (e.g., region #2 in Figure 8.2). In this case, the relevant, transient data structures that correspond to such regions are copied (or moved) to the target configuration. There are also other cases when no physical data migration is needed, and only mapping entries are copied between transient data structures (e.g., when region composition is only partially changed). For all these pages, the corresponding entries in the *MS* vector are marked as 11. Afterwards, the remaining entries are divided equally between  $K$  background threads of Migrator. Each migration thread iterates over its set of logical page addresses and (if not already done) performs a migration of corresponding pages from the current configuration to the target. For this, it follows the same procedure as the normal write requests (see Algorithm 8.2). The only difference is that no data transfer to/from the host is needed. Flash pages are copied to the new physical location via COPYBACK command (see Chapter 4).

After checking all pages in its set, the thread either normally terminates if all pages have been migrated, or it repeats the check again - if there were skipped pages during the last iteration (e.g., pages that were in read I/Os). When all migration threads normally

---

**Algorithm 8.2** Writing a page during migration process

---

**Input:**  $lpn$  - logical number of page to be written;  $buffer$  - buffer holding page's content

**Output:** return code of write operation

```
1: function WRITEPAGE DURING MIGRATION( $lpn, buffer$ )
2:    $curStatusRef \leftarrow$  GETPAGE MIGRATION STATUS REF( $lpn$ )
3:   if  $curStatus = 11$  then                                     ▷ already migrated
4:     return WRITEPAGE FROM CONF( $targetConf, lpn, buffer$ )
5:   else
6:      $newStatus \leftarrow 10$                                      ▷ in migration
7:      $expStatus \leftarrow 00$                                    ▷ neither in READ nor in WRITE I/O
8:      $statusChanged \leftarrow$  CAS( $curStatusRef, newStatus, expStatus$ )
9:     if  $statusChanged = True$  then
10:       $ret \leftarrow$  WRITEPAGE FROM CONF( $tarConf, lpn, buffer$ )
11:      if  $ret = OK$  then                                       ▷ written successfully  $\implies$  mark "migrated" &
      invalidate old
12:         $newStatus \leftarrow 11$ 
13:         $expStatus \leftarrow 10$ 
14:         $statusChanged \leftarrow$  CAS( $curStatusRef, newStatus, expStatus$ )
15:        if  $statusChanged = True$  then
16:           $ret \leftarrow$  INVALIDATE PAGE( $curConf, lpn$ )
17:          return  $ret$ 
18:        else
19:          return ERROR_WRONG_STATUS
20:        end if
21:      else                                                     ▷ write failed  $\implies$  remove "in migration"
22:         $newStatus \leftarrow 00$ 
23:         $expStatus \leftarrow 10$ 
24:         $statusChanged \leftarrow$  CAS( $curStatusRef, newStatus, expStatus$ )
25:        if  $statusChanged = True$  then
26:          return  $ret$                                            ▷ ERROR_WRITE_FAILED
27:        else
28:          return ERROR_WRONG_STATUS
29:        end if
30:      end if
31:    else                                                       ▷ cannot set "in migration"
32:      return ERROR_PAGE_IN_USE
33:    end if
34:  end if
35: end function
```

---

---

---

terminate the migration process is finished, the auxiliary data structures are removed (e.g., *MS* vector), and the target configuration becomes the current one.

**Resuming Migration Job.** The migration process can be paused and resumed at any point in time. It is also possible upon resuming to change the number of background threads.

The migration is also safe against a system crash, after which it continues from the last state before the crash. To ensure this property two implementation details were necessary. First, the flag determining that the system is in migration, as well as the description of the target configuration (in form of DDL statements defining every region) are persisted (e.g., as part of the database catalog) before the migration starts. With this after system restart the DBMS can identify if there is unfinished migration.

Second, in order to correctly reconstruct the address translation tables in both configurations, as well as the *MS* vector, we reserve one bit in an OOB area (so-called *configuration bit*) of every Flash page, that allows to determine under which NoFTL configuration this page was written to Flash storage last time. As only two concurrent configurations are possible, a single bit is enough to differentiate between them. For instance, the logical zero in this bit might indicate the current configuration, while logical one the target configuration. In the next migration, this will be vice versa, i.e., logical one will indicate current configuration. Thus, when system starts and DBMS has identified that migration process was interrupted and must be resumed, it starts the normal NoFTL recovery (see Chapter 5.5) with the only difference that depending on the configuration bit either transient mapping table of the current configuration is updated, or one of the target configurations. Consequently, the status of the page in *MS* vector is either set to 00 (pages not migrated before crash) or to 11 (pages already migrated before crash). Once the NoFTL recovery and DBMS recovery finish, the DBMS goes online and the background migration is resumed.

**Memory and Storage Overheads of Migration.** The memory overhead of the migration process consists of two main parts: (i) *MS* vector holding the migration status for every database page (two bits per page); and (ii) memory consumed by the transient data structures of the current configuration (e.g., address translation tables for every region). In the worst case, if in the current configuration all regions use pure page-level address translation, then every terabyte of Flash storage would require additional 512MB for mapping tables and 32MB for *MS* vector during migration (assuming 8KB Flash pages). After migration all transient data structures associated with the current (old) configuration are deleted. Note, however, that typically, configurations require much less memory due

---

to utilization of the configurable Flash storage concept (see Chapter 6.1.2).

The migration process does not require any additional or reserved space on Flash storage. Every Flash page is referenced only by one configuration (current or target). Only during a short time when the page is being migrated to the target configuration, and its copy in the current configuration is not yet reclaimed by garbage collection, two copies of that particular page exist on Flash storage. But this situation is natural for Flash memory due to the out-of-place update strategy. Every normal GC write in any Flash SSD has the same behavior - two identical copies of the page exist on Flash memory for very short time. With this, one can see the whole migration process as a global garbage collection that touches all regions changed in the target configuration.

The migration process can be properly configured (number of background migration threads) and scheduled by the database administrator based on the following information: (i) total I/O overhead of the migration, which can be easily estimated by comparing current and target configurations; (ii) the performance characteristics of the Flash storage; (iii) typical I/O load produced by the databases over time. It is important to remember, that migration process causes no data transfer between host (DBMS) and Flash storage. Database pages are migrated to new physical locations via Flash internal COPYBACK commands. Thus, neither the database buffer, nor the I/O link are influenced by the migration.

### **8.3. Wear-Leveling in Multi-Region Configuration**

Chapters 5.6, 6.3 and 7.4 show that integration of Flash management into the DBMS, applying the concept of configurable Flash storage, and utilization of an IPA approach allow together to decrease the overall number of performed erase operations by an order of magnitude as compared to the traditional FTL-based Flash SSDs (e.g., 9x-24x less erases under TPC-C benchmark as reported in Table 6.1). As the wear-out of Flash memory directly depends on the number of performed program/erase (P/E) cycles (Chapter 2.3.1) we can clearly state that the NoFTL architecture significantly improves on longevity characteristics of Flash storage. However, the proper wear-leveling is still essential. A region-specific (local) wear-leveling and data placement strategies ensure even wear out of Flash blocks in all chips in that region (see Chapter 6.1.2). However, as regions typically cluster database data based on its access properties, the wear-out speed in every region (average number of P/E cycles across chips in this region within a certain time) might differ from the others. Although these differences are typically small, we need to provide a mechanism to ensure roughly even wear-out of the whole Flash storage. In this chapter we discuss details of a cross-region (global) wear-leveling strategy.

---

Similar to local WL, global WL can also be realized in different ways. Analysis and evaluation of WL strategies is a separate, self-contained topic. In the following we concentrate on one approach of global WL that is based on a slight modification of existing garbage collection algorithms. Remember, one of the main characteristics of a NoFTL region is that it is defined as a flexible composition of NAND chips, i.e., although the number of chips in a region is fixed, the composition of chips can change over time (see Chapter 6.1). Thus, by monitoring the average erase counts in every chip, global WL can decide to switch two chips of different regions. For instance, one chip from a region that contains cold data might be replaced with a chip from a region containing hot data.

The simplified scheme for switching two chips between different regions is following. To track the exchange procedure we use two state bit-vectors (*BS*) - one per chip. Each vector stores the state of all blocks in the corresponding chip. As only one bit per block is required the total size of both vectors is very small (e.g., for NAND chips containing 4096 blocks both vectors together will require 1KB of memory). Logical zero means that corresponding block contains data that need to be moved to another chip, while logical one means that this block was already moved to the target chip. Initially all entries in vectors corresponding to non-empty blocks are marked with zeros, while empty blocks are marked with ones.

The core of the exchange procedure is based on a simple modification of a function that returns the next empty block in a chip (*get\_next\_free\_block*). Typically, for every chip there is a list of free blocks. Whenever the current block for a particular group is full, this function returns (based on the local WL) the next block from this list. However, in case the chip is marked to be exchanged with another one, this function returns the empty block from the free list of that other chip. This function is used for both - normal DBMS writes (host writes), and page migrations of garbage collection.

Thus, all DBMS writes that correspond to chips that are currently in exchange are placed on a correct (target) chip. Whenever some of these chips runs out of free blocks the garbage collection on this chip is triggered (or it is triggered proactively as a background process). Now, in contrast to the normal algorithm, GC selects a victim block only from those that have status of logical zero (i.e., need to be moved) in the *BS* vector. All valid pages from this block are written to the desired target chip. Afterwards, the block is erased and marked with logical one in the *BS* vector. Once both *BS* vectors contain no zeros the exchange procedure is finished, and the auxiliary data structures are removed.

The key advantage of this algorithm is that normal host writes help to exchange data between chips. Moreover, the data being copied by garbage collection contains only valid Flash pages, thus saving the GC overhead that would be needed anyway to reclaim the space occupied by invalidated pages.



---

## 9. Related Work

---

The main purpose of this research work is optimizing the usage of Flash storage for database management systems. For this, we propose the NoFTL architecture that allows to significantly increase performance and longevity characteristics of Flash SSDs. Different aspects of this approach have been partially presented in our earlier publications [30], [34], [35], [29], [32], [36], [31], [33], as well as in tutorials [77] and [78]. Separate research, which took the NoFTL architecture as a foundation and a baseline, is now in progress in our group. It concentrates on a direction of native storage processing for Flash and NVM-based storage, and has already shown significant results [101].

Since the topic of effective use of Flash storage has been a hot trend in research for some time there is a considerable body of related work done by academia and industry. Various optimization solutions show significant performance improvements of Flash-based storage in different scenarios. That work was both the base and the motivation for our own research in this field. The largest part of related work concentrates on improving the performance of traditional, FTL-based SSDs (e.g., design of new FTLs, proper tiering of Flash in the storage hierarchy, optimization of DBMS algorithms to leverage fast random reads, etc). Others recognized the problems associated with the black-box architecture for SSDs and tried to solve them. In most cases, they extend the traditional interface with new primitives, which allow the DBMS to leverage a certain property of native Flash or on-device FTL. Only recently solutions were proposed, which similarly to NoFTL, completely revisit the I/O stack and propose radical changes in its design. In the following we describe several approaches from each group and discuss their relation to the NoFTL architecture.

### 9.1. FTL-based SSDs

**FTLs** In the past numerous designs of FTLs have been proposed. These approaches primarily address the type and behavior of logical-to-physical address translation, as this is the key functionality of every FTL scheme, which to a large extent determines also other management tasks (garbage collection and wear-leveling). Based on the granularity of

---

the mapping table all solutions can be classified into three groups: (i) block-level mapping ("constrained" mode in [11], AFTL in [54], [47]); (ii) page-level mapping ([28], [39], [62], [46], [113]); and (iii) hybrid mapping solutions ([48], [45], [52], [75], [56], [37]). A brief discussion about major characteristics, advantages and bottlenecks of all three groups, as well as analysis of some of the above schemes are provided in Chapter 2.3.2. Further evaluation comparison of different FTLs is provided in [16] and [61].

Interesting is also the work of Bonnet et al. in [11], where they proposed the design of a bimodal SSD. It should be able to operate in two modes: (i) if the DBMS issues "constrained" I/O patterns (i.e. no in-place updates, no random writes) the SSD uses minimal FTL, which provides only block-level mapping and wear-leveling; (ii) for all "unconstrained" I/O patterns the SSD switches to traditional FTL. While for decision support systems with mostly read-only workloads and batch updates such bimodal SSDs would be beneficial, in OLTP systems most of the time the DBMS would issue unconstrained I/O patterns.

Similar to the "constrained" mode of bimodal SSDs [11] is also the idea of application managed Flash (AMF) presented by Lee et al. in [54]. They suggest that applications should issue only append-only write requests, and allow to re-use logical addresses only after explicit triggering of erases on those addresses (via a TRIM command). With these constraints the on-device FTL scheme (called AFTL) can be significantly relaxed. AFTL requires only a very small segment-based address translation table (similar to block-level mapping), simplified wear-leveling and bad-block management. As erases are directly controlled by applications, no garbage collection on the device is needed. This approach is a good match for applications that rely on the log-structured approach. For instance, log-structured file systems and storage engines based on LSM-Trees.

Although FTL-schemes might differ significantly from each other, they all share one common property - their performance and efficiency are workload-dependent [88]. There are FTLs that are optimized for read-mostly workloads; or write-intensive with random or sequential update patterns; for skewed or uniform workloads; with strong and weak response time guarantees, etc. Consequently, also the overall performance of SSDs becomes workload-dependent with frequent and unpredictable performance outliers due to background FTL processes [16].

To overcome the issues derived from the workload-dependent efficiency of the FTL and unknown workloads in advance, NoFTL allows the dynamic selection of the most appropriate set of Flash management algorithms. Thus, Flash management can evolve over time in response to changes in the workload. Furthermore, the utilization of *regions* allows us to provide multiple such algorithms simultaneously, each being most suitable for a particular group of database objects.

---

---

**Leveraging Out-of-Place Updates** Another group of proposed approaches is based on leveraging an out-of-place update strategy, which is applied in all Flash SSDs. The fact that modified data is always written to new physical location on Flash memory, while the old version of data is kept for certain time, can be efficiently used to provide I/O and transactional atomicity, as well as the widely used concept of snapshots.

Kang et al. in [46] have suggested to enrich the responsibility of on-device FTL in order to utilize internal out-of-place updates for providing DBMS transactional atomicity. Through a modified I/O interface the DBMS notifies the SSD (i) on every read and write I/O request to which transaction it belongs, and (ii) about end or abort of every transaction. The latter maintains an internal transaction table and keeps obsolete versions of modified database pages belonging to running transactions for recovery purpose. The authors have integrated X-FTL into the SQLite database engine<sup>1</sup> and evaluated it on multiple benchmarks using the OpenSSD Jasmine board. The tests have shown significant performance improvement, which is the result of reduced number of write requests performed by the DBMS (no write-ahead logging is needed any more). However, the approach has several drawbacks. First, due to the additional memory (transaction table, page-level mapping) and computational overheads on on-board SSD resources they are suitable only for systems with quite low level of concurrency. Second, disadvantages of block device interface and the black-box architecture of SSDs (see Chapter 1.2) are not considered by this solution.

A similar approach was also proposed by Prabhakaran et al. in [79]. They integrate into the SSD additional functionality that takes the responsibility for atomicity, isolation and recovery properties provided by file systems. The interface of such SSD (*TxFIash* SSD) is enhanced with two new commands *WriteAtomic* and *Abort*, through which the file system notifies the storage about single transactions. This approach, however, was designed specifically for file systems, and is not suitable in that form for common database systems because of an insufficient interface, locking-based isolation, and no deadlock detection.

Ouyang et al. in [74] introduced a new I/O primitive "atomic-write", which is utilized by the MySQL InnoDB storage engine to eliminate redundant writes (DoubleWrite buffer in MySQL) for recovery purposes. The approach, however, does not allow to execute several atomic-writes simultaneously, which might be a problem in highly concurrent OLTP systems.

Weiss et al. in [102] propose the general virtualization layer (ANViL) for Flash SSDs. This layer as a part of the FTL scheme, enriches the SSD with the functionality that allows applications to realize (i) snapshot techniques, (ii) atomic writes, and (iii) transactional atomicity without additional I/O overhead. For this, they extend the normal block device

---

<sup>1</sup><https://www.sqlite.org/index.html>

---

interface with three additional commands: *clone*, *move* and *delete*, which are defined on a range of logical addresses.

For instance, let us consider the behavior of a *clone* operation [102], which is the major one for providing the mentioned above functionality. It takes as arguments two logical address ranges and their size. One of these ranges (source) is assumed to be already mapped to a physical range, i.e., this range corresponds to the valid data on Flash SSD. Another range (destination) is a newly allocated address range, which is not mapped to any data on storage. The implementation of *clone* simply assigns (copies) physical mappings, which correspond to the source range, to the destination range. In other words, after calling this operation the application has two logical address ranges both pointing to the very same data on Flash SSD (i.e., both are mapped by SSD to the same physical addresses). This is the basic behavior of creating snapshots. Assume a small file spanning contiguous logical addresses from 0 to 99. An application (e.g., the file system) creates a snapshot of this file by issuing *clone(0..99, size, 400..499)*. This operation will not cause any I/O for copying data of that file, only the mapping entries will be copied. If later on, the application issues a write request that updates a certain piece of data in destination address range (400..499), the SSD will write this updated data as usual in an out-of-place manner, but it will update the mappings of only this destination range, while the mappings of source address range will remain unchanged. Thus, the data corresponding to logical addresses 0..99 will remain unchanged. Similarly, the atomicity of write requests and transactional atomicity are also implemented with the help of *clone* operation.

Also Oh et al. in [72] have proposed to extend the interface of Flash SSDs with a novel *share* command, which has basically the same signature, purpose and implementation as the *clone* command mentioned above.

Although leveraging out-of-place updates on Flash storage can significantly improve performance by removing redundant I/Os, all the approaches mentioned above have some common bottlenecks. First, by shifting the responsibility for providing atomicity and snapshots to the FTL scheme, they increase the pressure on limited on-device resources (memory, controller). For instance, some of these approaches assume a fine-grained address translation table (e.g., pure page-level mapping), which is typically impractical for common SSDs due to its memory requirements. Second, the problems resulting from the FTL-based, black-box architecture of modern SSDs (see Chapter 1.2) are not addressed by these solutions.

In contrast, in the NoFTL architecture the control over the out-of-place updates lies completely with the DBMS, and that allows it to leverage them for guaranteeing atomicity and providing snapshots without the need for extra interface commands (e.g., *WriteAtomic*, *clone*, *share*), and without the aforementioned bottlenecks (see Chapter 5).

---

**Other Optimizations** Research work was also done for optimizing the applications, and DBMSs in particular, which are running on traditional Flash SSDs.

For instance, Gottstein et al. in [26] have proposed the SIAS approach that allows to significantly reduce the number of write requests performed in database systems with multi-version concurrency control. The basic idea is based on avoiding explicit invalidation of old tuple versions. Instead of setting invalidation timestamps on previous versions, SIAS suggests to implement an implicit invalidation by storing a pointer to the previous version in the current version of the tuple. This approach allows SIAS to eliminate all write requests that were previously done for explicit invalidation of tuple versions.

Various suggestions regarding optimizing the database buffer eviction strategy for Flash SSDs were presented in [60], [21] and [112]. The Flash-aware cost model for the DBMS optimizer, which is based on awareness of Flash I/O asymmetry, was proposed in [7].

The performance of the storage interface for Flash SSDs is another important topic in research. [97] argues for the use of RDMA as general high performance storage interface. IBM describes the use of OFED RDMA interfaces under their BlueGene Active Storage initiative [23].

All these approaches, however, are based on using the traditional, FTL-based SSDs, and thus do not address issues resulting from their black-box architecture.

## 9.2. Native Flash Storage

This group of related research work looks deeper into the disadvantages and limitations of modern Flash SSDs, and seeks the root causes.

**LightNVM** Bjorling et al. in [9] discuss the unsuitability of the block device interface for Flash SSDs. Four years later, in [10] they described their solution - the open-channel SSD subsystem called LightNVM. In order to overcome the aforementioned disadvantages of today's SSDs they propose to shift Flash management to the host. The physical page address (PPA) I/O interface of LightNVM resembles our native Flash interface (NFI), as both allow the application to execute read, write and erase commands on physical addresses on Flash memory. However, NFI, which was first introduced in [30], is not limited to only those commands. Our current prototype of the interface offers the DBMS further commands like: *copyback*, *write\_delta*, *get\_addr\_table*, etc, which allow for significant reduction of GC overhead and amount of transmitted data to/from storage. Further, NFI contains a set of commands enabling near-data processing by the DBMS under NoFTL architecture (see Chapter 4).

---

Although [10] describes the LightNVM subsystem and PPA I/O interface, it does not show how this can be utilized by the DBMS to increase its performance. In this sense, the NoFTL approach can be seen as what the authors of [10] described as "future work" in regard of "tuning SQL databases for performance on open channel SSDs" and "characterizing the potential of application specific FTLs with open-channel SSDs". NoFTL did this "tuning" of the database system. It shows how Flash management can be effectively integrated into the DBMS subsystems; and how a DBMS can efficiently manage physical Flash space by means of novel storage structures. Thus, *regions* allow us to control available I/O parallelism of Flash storage, and apply different Flash management algorithms simultaneously depending on characteristics of database objects, while *groups* can further reduce the overhead of garbage collection (see Chapter 6.1).

**BlueDBM** Jun et al. in [43] described the architecture of the distributed Flash storage - BlueDBM. It is a cluster of nodes, each containing a Flash storage device, which is accessed by the host as raw Flash memory. For this, the Flash controller provides an interface that is similar to NFI (see Chapter 4) and PPA I/O in [10] as it defines *read*, *write* and *erase* commands on physical addresses. However, in contrast to the NoFTL architecture, the responsibility for Flash management is shifted to the file system (log-structured, flash-aware file system RFS [53]). Although removing the on-device FTL is an important step in optimizing Flash storage, we argue that it is more beneficial to integrate the Flash management into the DBMS, and thus let the DBMS control the physical placement on Flash. Only the DBMS has a comprehensive knowledge about the data, which allows it to optimize both the Flash management and the traditional DBMS algorithms (see Chapters 5, 6).

Another important concept of BlueDBM is that every storage device is equipped with a hardware accelerator, which can perform in-storage processing. However, as the queries for the on-device FPGA must be defined on physical addresses, before sending such a query to storage, the application (DBMS) must consult the file system about the physical address for each logical address touched in the query. These calls to the file system produce extra overhead, which is not present in the NoFTL architecture.

**CORFU** Balakrishnan et al. in [6] presented a design of a shared log called CORFU. The log is implemented on top of a cluster of Flash SSDs. The distribution of the log over multiple SSDs allows to efficiently scale the I/O bandwidth of the system. Every client in the system maintains a replica of a small map (called projection), which maps log positions to addresses on Flash SSDs in the cluster. In order to read from a certain log position, a client consults this map, and then issues I/O requests to the relevant SSDs.

---

To append to the log, the client first requests the position from a separate central node (called sequencer). The sequencer returns the synchronized last log position (log tail), which is then used by the client to write the log entry.

Although the abbreviation CORFU comes from *Cluster of Raw Flash Units*, in fact, CORFU SSDs are not FTL-less (what we assume under *raw Flash*) in general case. CORFU SSDs differ from conventional SSDs in the following points. First, they provide to the application *write-once* semantic, which is exposed through an API consisting of *append*, *read*, *fill* and *trim* commands. Second, these SSDs offer to the host an infinite address space. The latter requirement results in a special FTL-scheme, which is based on page-level mapping organized as a hash table (needed to map infinite logical addresses to limited physical addresses). This, however, might become an issue as maintaining a pure page-level mapping is typically impractical for conventional SSDs due to limited on-device DRAM resources. Implementing a kind of partially cached page-level mapping (e.g., DFTL [28]) results in significant I/O overhead (see Table 5.3 in Chapter 5.6). Similar to conventional SSDs, CORFU SSDs also include garbage collection, bad-block management, and most probably wear-leveling in their FTL-scheme. With this, CORFU does not change the black-box design of SSDs, and thus it does not address their drawbacks.

**File Systems for Flash** Several file systems have been proposed for Flash-based storage. Some of them, such as [67], [49], operate on conventional FTL-based Flash SSD. By addressing main properties of Flash (e.g., asymmetric latencies of I/O operations, out-of-place update strategy) they can reduce the overhead caused by traditional file systems (see Chapter 5.6). However, the problems and limitations resulting from the black-box design of modern SSDs are not solved in these file systems.

Another group of proposed file systems targets the raw (native) Flash storage. Some examples of them are [107], [57], [44], [76], [110], [108]. These file systems typically apply a log-structured design, and assume the responsibility for all or most of Flash management tasks (e.g., garbage collection, wear-leveling). Exposing the native Flash to the host, and integration of Flash management into the file system allows them to solve multiple problems of conventional SSDs. Thus, (i) the redundant functionalities between file system and FTL are removed (e.g., garbage collection, mapping); (ii) support of journaling does not require additional I/O overhead; and (iii) richer resources of host systems allow them to use fine-grained address translation tables.

However, all these approaches have one important bottleneck - for database systems the Flash storage remains a black-box. As we have shown in this work, much more significant improvements and optimizations can be achieved by giving the DBMS (and not the file system) full control over the Flash storage. Only the DBMS has comprehensive

---

knowledge about the data and the workload, which is necessary to effectively manage Flash storage. Based on this knowledge the DBMS can utilize multiple Flash management schemes simultaneously, depending on the properties of data (see Chapter 6). This solves the so-called "one size fits all" problem, which is typical for all modern SSDs, but also for all Flash-aware file systems. Moreover, the DBMS can better control available Flash parallelism (see Chapters 5 and 6.1), and further eliminate redundancies along the I/O path.

### 9.3. IPA Approach

The detailed analysis of the previous research work related to the approach of in-place appends (IPA), which is discussed in Chapter 7, can be found in our earlier publication [31]. In brief, Lee et al. in [50] proposed an approach called In-Page Logging (IPL) that has a similar idea for reducing write-amplification on Flash storage. Both solutions (IPL and IPA) allow to a certain extent to avoid expensive out-of-place writes by writing out not the complete Flash pages, but only data which is actually modified (delta). However, the key difference between IPL and IPA is *where* and *how* the delta records are written on Flash storage.

IPL tries to accumulate the delta records separately for every page, and when this per-page, in-memory delta buffer is full, it is written to a *separate* Flash page within the Flash block that contains the original page. When the database issues a fetch of a single database page, it must read from the Flash SSD multiple Flash pages: (i) Flash page(s) that contain the original version of db-page; (ii) Flash page(s) that contain delta-records for that particular db-page. If we assume that db-page equals the Flash page, then IPL approach will result in at least doubling the number of performed read operations. Under the read-intensive workloads this read overhead on storage and storage link will cause significant performance bottleneck. In contrast, IPA utilizes the *ISPP* technique to append the delta records directly to the *very same* physical Flash page that contains original data (see more in Chapter 7). With this, there is no additional read overhead produced by IPA. Both approaches also have significant differences with regard to the overhead of garbage collection, and space reservation. Please refer to [31] for more analysis and evaluation comparison of IPL and IPA. IPL B+-tree [70] and IPA-IDX (see Chapter 7.3.1 and [33]) - are variations of the basic approaches (IPL and IPA) designed for database indexes. The major similarities and differences between them are similar to those mentioned above.

Cai et al. in [109] proposed an approach called "Correct-and-Refresh", which utilizes ISPP technique to correct retention errors on Flash memory. Basically, they suggest to periodically overwrite Flash pages with the same data without performing erase operations.

---

Thus, if some Flash cells in that page have small capacity leakage (retention errors) it will be corrected with this overwrite operation. Cells without leakage would remain unchanged during this overwrite operation. More information about ISPP can be found in [3].

## 9.4. In-Storage Processing

The significant increase of computational power in modern Flash storage devices naturally raised the interest for in-storage processing in the research community. For instance, in [20] Do et al. presented the prototype of "Smart SSD", which is capable of executing certain simple queries of analytical data processing. The authors provide in detail both the communication protocol (using sessions) and the API to be used by the DBMS in order to issue the execution and control of queries on SSD. The evaluation results show the improvement of read bandwidth of more than 2.5 times.

Similarly, Seshadri et al. in [87] proposed the prototype of the user-programmable SSD called Willow, which allows the users to augment the storage device with the application specific logic. In order to provide the new functionality of the SSD for a certain application, three subsystems must be appropriately modified to support a new set of RPC commands: the application, the kernel driver and the operating system running on each storage processing unit inside the Flash SSD.

Woods et al. in [105] and [104] presented an intelligent storage engine called *Ibex*. It can offload to the on-device FPGA complex multi-predicate filtering, as well as aggregation tasks. The evaluation results against multiple storage engines of MySQL have shown significant performance improvement, and decrease in energy consumed by the system.

Another example is BlueDBM [43], which allows for in-storage processing in the distributed Flash storage with the efficient link between storage devices. We discuss some aspects of BlueDBM in Chapter 9.2.

In-storage processing allows the DBMS to utilize the processing resources of the storage device. Furthermore, performing certain types of queries (e.g. selection, projection, join) on the device itself significantly reduces the amount of data transferred to the host, which unloads the bus, improves required bandwidth and cache hits. However, the proposed protocols for in-storage processing enable only the partial access to the on-device processing units, while the general black-box architecture of SSD with its disadvantages remains. In such a smart SSD the DBMS still does not have any control over the physical data placement nor over Flash management. Moreover, the on-device FTL (or Flash management in file system) cannot access the rich DBMS status information (statistics, metadata) needed to optimize its algorithms for a particular workload and its data.

---

NoFTL attempts to solve all the main problems resulting from the black-box architecture of modern SSDs by giving the DBMS full and exclusive control over the Flash storage (native Flash memory and on-device processing resources). This architecture provides a uniform solution for all workloads and database objects with different characteristics. Vincon et al. in [99] presented recently the first results of the *nativeNDP* concept - native data processing for storage under the NoFTL architecture. This is an on-going follow-up research project.

---

## 10. Conclusions

---

Flash SSDs are becoming ever more popular as storage devices, and their market share is going to continuously increase in the next years. High performance, low energy consumption, and the price measured in terms of Price/IOPS - are the main advantages of Flash-based SSDs over the conventional magnetic disks, which were dominating the storage market for more than forty years. However, while all-Flash and Flash-dominant data centers are becoming common nowadays, the increasing requirements on storage put also the development of Flash SSDs under high pressure. Thus, the demand for higher density lead Flash technology in the past years from planar NAND to 3D-NAND. By introducing more and more vertical layers of Flash cells on a single chip, 3D-NAND enables higher memory density without the necessity to shrink individual cells, and thus without performance and reliability disadvantages. However, far more challenging is tackling the performance requirements of today's applications. The usage of high performance hardware interfaces (e.g., PCIe, Fibre Channel) and protocols (e.g., NVMe) significantly improves the I/O bandwidth of SSDs, but they do not solve the major barrier on the way of lifting the whole potential of Flash memory - the software stack.

Almost all Flash SSDs nowadays are designed to be backwards compatible with magnetic disks. This makes the replacement of HDDs with SSDs seamless, and together with the advantageous properties of Flash storage it has favored its proliferation. However, masking the native behavior of Flash memory behind the compatibility layer, and the use of a legacy I/O stack negatively impact performance and longevity characteristics of Flash. Thus, one of the major disadvantages of modern Flash SSDs is the high overhead of management algorithms encapsulated in the Flash translation layer (FTL). Tasks of the FTL such as address translation and garbage collection, which basically hide the erase-before-overwrite principle of Flash memory from the host, produce significant write-amplification (e.g., 5x). In combination with the traditional 'cooked' storage architecture (using a file system) write-amplification is further increased up to 15x. That means, that a single write request of a 4KB page issued by an application can turn into 20 to 60KB being written on Flash SSD. As a result, the effective I/O throughput and the lifetime of storage dramatically decrease. The reasons for this overhead can be clustered into several groups: limited computational and DRAM memory resources of the SSD; lack of information on the side

---

of FTL about properties of stored data; lack of information on the side of applications about FTL and physical organization of Flash; and redundant functionality along I/O path (application, file system, FTL).

These drawbacks of modern Flash SSDs resulting from their FTL-based, black-box architecture have motivated this work. Our goal was to optimize the use of Flash for one of the most important 'customers' of storage that demands ever higher I/O performance - the database management system. To achieve this goal we propose the novel NoFTL storage architecture. The basic idea of NoFTL is to give the database system full control over the underlying Flash SSD. To achieve this, all intermediate layers between the DBMS and physical storage, such as file system, blocked I/O layer and FTL, are eliminated. By exposing the native behavior and properties of Flash memory to the DBMS, the latter can provide significant optimizations for multiple internal subsystems. Thus, many algorithms and data structures, that were originally designed for conventional magnetic disks, and remain unchanged due to backwards compatibility of modern Flash SSDs, can be completely revisited under the NoFTL architecture. On the other side, information about data and workload available in the DBMS, as well as richer resources of the host system, allow for optimization of typical Flash management tasks. Moreover, NoFTL removes all functional redundancy, which is present in different layers of traditional 'cooked' storage architecture.

The topic of effective use of Flash storage has found large and widespread interest in recent years. Both academia and industry have proposed various approaches to tackle disadvantages of modern Flash SSDs. However, most of them address only one part of existing problems, while leaving other issues uncovered. Though there are a few approaches that similarly to NoFTL propose to completely redesign the I/O stack, to the best of our knowledge the NoFTL architecture is the first approach that not only opens the SSD by removing legacy and compatibility layers, but also provides the complete design of a Flash-aware database system, which allows to unveil and fully utilize the performance potential of Flash-based storage. Another important contribution of this work is the complete functional prototype of a database storage engine following the NoFTL architecture based on an open-source database engine. Different aspects of the NoFTL architecture, and live demonstrations of the NoFTL prototype have been presented to the database community in [30], [34], [35], [29], [32], [36], [31], [33], as well as in tutorials [77] and [78].

The NoFTL storage architecture consists of three main integral parts. The first is the native Flash interface (NFI), which exposes full control over the Flash memory and computational resources of the SSD to the DBMS (see Chapter 4). In order to operate via NFI on an FTL-less SSD the DBMS must take responsibility for the Flash management functionality. Thus, integration of typical Flash management tasks into different subsystems of

---

the DBMS, as well as mutual optimizations of these tasks and DBMS algorithms represent the second part of the NoFTL architecture (see Chapter 5). These two parts which we call *basic* NoFTL allowed us to decrease the write-amplification of Flash SSD by more than 5.5x under OLTP workloads as compared to the traditional 'cooked' storage architecture.

Lower write-amplification has two major implications. First, it results in smaller response times for read and write operations, which in turn, has a direct impact on the transactional throughput. Thus, on I/O-bound systems this basic NoFTL approach can increase the transactional throughput of OLTP workloads by more than 3.5x. Obviously, for in-memory systems this advantage on I/O layer has significantly less impact on database performance.

The second advantage of smaller write-amplification is the improved longevity of Flash storage. The wear-out of Flash SSDs is primary coupled with the number of program-erase cycles performed on Flash memory. Thus, for MLC Flash each particular Flash block can "survive" only a few thousand P/E cycles (typically 3000-5000 P/E cycles). The basic NoFTL approach allows us to reduce the number of performed P/E cycles by more than 5x for OLTP workloads as compared to traditional 'cooked' storage architecture. As a result, the lifetime of an SSD for the same database load is increased correspondingly. Last but not least, improved performance and longevity of Flash storage enables us to decrease the overall cost of the storage infrastructure.

To further improve performance and longevity of SSDs we extended the *basic* NoFTL architecture with the third component - the concept of configurable Flash storage. With the help of two novel storage abstractions, regions and groups, the DBMS under NoFTL can (i) perform intelligent physical data placement; (ii) apply multiple Flash management schemes simultaneously; and (iii) control utilization of available I/O parallelism of SSD (see Chapter 6). Thus, the proper NoFTL configuration of SSD using multiple regions with different management schemes can outperform the *basic* NoFTL approach by more than 2x, resulting in total in up to 12x less write-amplification for OLTP workloads as compared to cooked storage. The overall reduction in number of performed P/E cycles is also about 10x compared to a traditional approach with FTL-based SSD and file system.

Although the three pillars of the NoFTL architecture mentioned above provide a complete concept of Flash-aware DBMS on FTL-less SSDs, it should be seen rather as a foundation for further optimizations. A good example for this is the approach of in-place appends (IPA) presented in Chapter 7, as it perfectly shows the interplay of NoFTL components. The IPA approach is the optimization of write behavior on Flash storage for small updates, which typically dominate in OLTP workloads. It completely revisits the traditional erase-before-overwrite principle of Flash memory, by allowing already written Flash pages to be updated by small appends without the need to perform an erase operation. IPA becomes possible only because the NoFTL gives the DBMS full control over the Flash storage. All three components of NoFTL contribute to the realization of IPA: native Flash interface

---

offers a special command for in-place appends; integration into the Flash management tasks, as well as support by buffer and storage managers of the DBMS ensure the proper logic of IPA; and the concept of configurable Flash storage allows selective application of IPA only for relevant database objects. IPA allows to further decrease the number of erase operations by about 2x as compared to a configuration without IPA, which positively impacts performance and longevity characteristics of the Flash storage.

A number of other significant optimizations based on the NoFTL storage architecture are possible, and some of them are currently investigated in the ongoing projects. However, these are separate, complex and self-contained topics, which are beyond the scope of this work.

---

## 11. Future Work

---

In this work we have focused on effective use of Flash storage for relational database management systems. However, the NoFTL storage architecture would be beneficial also for NoSQL database systems, such as key-value stores, document stores and graph databases. While the basic principles and various optimizations described in this work would remain unchanged, each particular type of DBMS would require its own design and implementation of NoFTL. Thus, for instance, a separate on-going project presented in [101] currently investigates the realization of NoFTL for key-value stores.

Another interesting direction of future research is the near-data processing (NDP) under NoFTL architecture. The topic of NDP for Flash SSDs is very popular nowadays, and recently multiple approaches have been proposed by academia and industry (see Chapter 9.4). NDP uses the computational power of SSDs (e.g., ARM controller, FPGA) by delegating certain database tasks and simple queries to the device. This reduces significantly the amount of transmitted data between host and storage, speeds up data processing and decreases power consumption of the system. However, most of the proposed approaches are designed for FTL-based SSDs, which are suffering from significant performance and longevity bottlenecks described in this work. Combining NoFTL architecture with extensive NDP support would allow us to use both memory and computational resources of SSDs in the most effective way. For instance, the fact that under NoFTL the database system has full control over the physical data placement can significantly simplify the process of specifying the tasks for SSDs, and enlarge their spectrum. Vincon et al. in [99] and [100] present the first promising results of realizing NDP support for key-value stores under NoFTL architecture.

As a follow-up future work we also see the realization of an idea to find an optimal NoFTL data placement configuration in automated or semi-automated modes (see Chapter 8). This can be realized with the help of appropriate heuristics. Developing these is a complex and self-contained research topic.



---

## Bibliography

---

- [1] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. “Data Page Layouts for Relational Databases on Deep Memory Hierarchies”. In: *The VLDB Journal* 11.3 (Nov. 2002), pp. 198–215. ISSN: 1066-8888. DOI: 10.1007/s00778-002-0074-9. URL: <http://dx.doi.org/10.1007/s00778-002-0074-9>.
- [2] S. Aritome. “NAND Flash Memory Revolution”. In: *2016 IEEE 8th International Memory Workshop (IMW)*. May 2016, pp. 1–4. DOI: 10.1109/IMW.2016.7495285.
- [3] Seiichi Aritome. *NAND flash memory technologies*. IEEE Press series on microelectronic systems. 2016.
- [4] Timothy G. Armstrong et al. “LinkBench: A Database Benchmark Based on the Facebook Social Graph”. In: *Proc. SIGMOD’13*.
- [5] Ivan Baev, Rajmohan Rajaraman, and Chaitanya Swamy. “Approximation Algorithms for Data Placement Problems”. In: *SIAM Journal on Computing* 38.4 (Aug. 2008), pp. 1411–1429. ISSN: 0097-5397. DOI: 10.1137/080715421. URL: <http://dx.doi.org/10.1137/080715421>.
- [6] Mahesh Balakrishnan et al. “CORFU: A Shared Log Design for Flash Clusters”. In: *Proc. USENIX NSDI’12*. 2012.
- [7] Daniel Bausch, Ilia Petrov, and Alejandro Buchmann. “Making Cost-based Query Optimization Asymmetry-aware”. In: *Proceedings of the Eighth International Workshop on Data Management on New Hardware*. DaMoN ’12. Scottsdale, Arizona: ACM, 2012, pp. 24–32. ISBN: 978-1-4503-1445-9. DOI: 10.1145/2236584.2236588. URL: <http://doi.acm.org/10.1145/2236584.2236588>.
- [8] David Bell. “Difficult Data Placement Problems”. In: *The Computer Journal* 27 (Apr. 1984), pp. 315–320. DOI: 10.1093/comjnl/27.4.315.
- [9] Mathias Bjorling et al. “The Necessary Death of the Block Device Interface”. In: *6th Biennial Conference on Innovative Database Research (CIDR)*. 2013.
- [10] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. “LightNVM: The Linux Open-Channel SSD Subsystem”. In: *Proc. FAST’17*. 2017.

- 
- 
- [11] Philippe Bonnet and Luc Bouganim. “Flash Device Support for Database Management”. In: *Proc. CIDR*. 2011.
- [12] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. O’Reilly & Associates Inc, 2005. ISBN: 0596005652.
- [13] Y. Cai et al. “Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques”. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2017, pp. 49–60. DOI: 10.1109/HPCA.2017.61.
- [14] Yu Cai et al. “Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis”. In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE ’12. Dresden, Germany: EDA Consortium, 2012, pp. 521–526. ISBN: 978-3-9810801-8-6. URL: <http://dl.acm.org/citation.cfm?id=2492708.2492838>.
- [15] Yu Cai et al. “Errors in Flash-Memory-Based Solid-State Drives: Analysis, Mitigation, and Recovery”. In: *CoRR abs/1711.11427* (2017). arXiv: 1711.11427. URL: <http://arxiv.org/abs/1711.11427>.
- [16] Feng Chen, David A. Koufaty, and Xiaodong Zhang. “Understanding intrinsic characteristics and system implications of flash memory based SSDs”. In: *Proc. SIGMETRICS’09*. 2009.
- [17] Changho Choi. *Multi-Stream Write SSD*. [https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2016/20160809\\_FC12\\_Choi.pdf](https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2016/20160809_FC12_Choi.pdf). 2016.
- [18] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O’Reilly Media, Inc., 2005. ISBN: 0596005903.
- [19] Ali Davoudian, Liu Chen, and Mengchi Liu. “A survey on NoSQL stores”. In: *ACM Computing Surveys* 51 (Apr. 2018), pp. 1–43. DOI: 10.1145/3158661.
- [20] Jaeyoung Do et al. “Query Processing on Smart SSDs: Opportunities and Challenges”. In: *Proc. SIGMOD’13*. 2013.
- [21] Paul Dubs et al. “FBARC: I/O Asymmetry Aware Buffer Replacement Strategy”. In: *Proceedings of International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. ADMS ’13. Jan. 2013, pp. 58–69.
- [22] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. 6th. USA: Addison-Wesley Publishing Company, 2010. ISBN: 9780136086208.

- 
- 
- [23] Blake G. Fitch et al. “Exploring the Capabilities of a Massively Scalable, Compute-in-Storage Architecture.” In: *Proc. HPDC’13*. 2013.
- [24] *Flash Memory Wear, Reliability & Life*. Radio-Electronics.com. URL: <http://www.radio-electronics.com/info/data/semicond/memory/flash-wear-reliability-lifetime.php> (visited on 07/02/2020).
- [25] Felix Gessert et al. “NoSQL Database Systems: A Survey and Decision Guidance”. In: *Comput. Sci.* 32.3-4 (July 2017), pp. 353–365. ISSN: 1865-2034. DOI: 10.1007/s00450-016-0334-3. URL: <https://doi.org/10.1007/s00450-016-0334-3>.
- [26] Robert Gottstein, Ilia Petrov, and Alejandro Buchmann. “Append Storage in Multi-Version Databases on Flash”. In: *Proc. BNCOD’13*.
- [27] Laura M. Grupp, John D. Davis, and Steven Swanson. “The Harey Tortoise: Managing Heterogeneous Write Performance in SSDs”. In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC’13. San Jose, CA: USENIX Association, 2013, pp. 79–90. URL: <http://dl.acm.org/citation.cfm?id=2535461.2535472>.
- [28] Aayush Gupta, Youngjae Kim, and Bhuvan Uргаonkar. “DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings”. In: *Proc. ASPLOS*. 2009.
- [29] Sergej Hardock et al. “Effective DBMS space management on native Flash”. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*. Ed. by Bernhard Mitschang et al. Gesellschaft für Informatik, Bonn, 2017, pp. 605–608.
- [30] Sergej Hardock et al. “NoFTL: Database Systems on FTL-less Flash Storage”. In: *Proceedings of the VLDB Endowment* 6.12 (Aug. 2013), pp. 1278–1281. ISSN: 2150-8097. DOI: 10.14778/2536274.2536295. URL: <http://dx.doi.org/10.14778/2536274.2536295>.
- [31] Sergej Hardock et al. “From In-Place Updates to In-Place Appends: Revisiting Out-of-Place Updates on Flash”. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD ’17. Chicago, Illinois, USA: ACM, 2017, pp. 1571–1586. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3035958. URL: <http://doi.acm.org/10.1145/3035918.3035958>.
- [32] Sergej Hardock et al. “In-Place Appends for Real: DBMS Overwrites on Flash without Erase”. In: *Proc. EDBT’17*.

- 
- [33] Sergey Hardock et al. “IPA-IDX: In-Place Appends for B-Tree Indices”. In: *Proceedings of the 15th International Workshop on Data Management on New Hardware*. DaMoN’19. Amsterdam, Netherlands: ACM, 2019, 18:1–18:3. ISBN: 978-1-4503-6801-8. DOI: 10.1145/3329785.3329929. URL: <http://doi.acm.org/10.1145/3329785.3329929>.
- [34] Sergey Hardock et al. “NoFTL for Real: Databases on Real Native Flash Storage”. In: *Proc. EDBT*. 2015, pp. 517–520.
- [35] Sergey Hardock et al. “Revisiting DBMS Space Management for Native Flash”. In: *Proc. EDBT*. 2016.
- [36] Sergey Hardock et al. “Selective In-Place Appends for Real: Reducing Erases on Wear-prone DBMS Storage”. In: *Proc. ICDE’17*.
- [37] Dan He et al. “Improving Hybrid FTL by Fully Exploiting Internal SSD Parallelism with Virtual Blocks”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 11.4 (Dec. 2014), 43:1–43:19. ISSN: 1544-3566. DOI: 10.1145/2677160. URL: <http://doi.acm.org/10.1145/2677160>.
- [38] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. “Architecture of a Database System”. In: *Found. Trends databases* 1.2 (Feb. 2007), pp. 141–259. ISSN: 1931-7883. DOI: 10.1561/1900000002. URL: <http://dx.doi.org/10.1561/1900000002>.
- [39] Yang Hu et al. “Achieving Page-Mapping FTL Performance at Block-Mapping FTL Cost by Hiding Address Translation”. In: June 2010, pp. 1–12. DOI: 10.1109/MSST.2010.5496970.
- [40] Soojun Im and Dongkun Shin. “ComboFTL: Improving Performance and Lifespan of MLC Flash Memory Using SLC Flash Buffer”. In: *Journal of Systems Architecture* 56.12 (Dec. 2010), pp. 641–653. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2010.09.005. URL: <http://dx.doi.org/10.1016/j.sysarc.2010.09.005>.
- [41] Xavier Jimenez, David Novo, and Paolo Ienne. “Software Controlled Cell Bit-density to Improve NAND Flash Lifetime”. In: *Proc. DAC’12*.
- [42] Ryan Johnson et al. “Shore-MT: A Scalable Storage Manager for the Multicore Era”. In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT ’09. Saint Petersburg, Russia: ACM, 2009, pp. 24–35. ISBN: 978-1-60558-422-5. DOI: 10.1145/1516360.1516365. URL: <http://doi.acm.org/10.1145/1516360.1516365>.

- 
- 
- [43] Sang-Woo Jun et al. “BlueDBM: Distributed Flash Storage for Big Data Analytics”. In: *ACM Transactions on Computer Systems (TOCS)* 34.3 (June 2016), 7:1–7:31. ISSN: 0734-2071. DOI: 10.1145/2898996. URL: <http://doi.acm.org/10.1145/2898996>.
- [44] Dawoon Jung et al. “ScaleFFS: A scalable log-structured flash file system for mobile multimedia systems.” In: *TOMCCAP* 5 (Jan. 2008).
- [45] Jeong-Uk Kang et al. “A Superblock-based Flash Translation Layer for NAND Flash Memory”. In: *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*. EMSOFT ’06. Seoul, Korea: ACM, 2006, pp. 161–170. ISBN: 1-59593-542-8. DOI: 10.1145/1176887.1176911. URL: <http://doi.acm.org/10.1145/1176887.1176911>.
- [46] Woon-Hak Kang et al. “X-FTL: Transactional FTL for SQLite Databases”. In: *Proc. SIGMOD*. 2013.
- [47] Hyukjoong Kim, Kyuhwa Han, and Dongkun Shin. “StreamFTL: Stream-level address translation scheme for memory constrained flash storage”. In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2018), pp. 617–620.
- [48] Jesung Kim et al. “A Space-efficient Flash Translation Layer for CompactFlash Systems”. In: *IEEE Trans. on Consum. Electron.* 48.2 (May 2002), pp. 366–375. ISSN: 0098-3063. DOI: 10.1109/TCE.2002.1010143. URL: <http://dx.doi.org/10.1109/TCE.2002.1010143>.
- [49] Changman Lee et al. “F2FS: A New File System for Flash Storage”. In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. FAST’15. Santa Clara, CA: USENIX Association, 2015, pp. 273–286. ISBN: 978-1-931971-201. URL: <http://dl.acm.org/citation.cfm?id=2750482.2750503>.
- [50] Sang-Won Lee and Bongki Moon. “Design of Flash-based DBMS: An In-page Logging Approach”. In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’07. Beijing, China: ACM, 2007, pp. 55–66. ISBN: 978-1-59593-686-8. DOI: 10.1145/1247480.1247488. URL: <http://doi.acm.org/10.1145/1247480.1247488>.
- [51] Sang-Won Lee and Bongki Moon. “Design of Flash-based DBMS: An In-page Logging Approach”. In: *Proc. SIGMOD’07*.
- [52] Sang-Won Lee et al. “A log buffer-based flash translation layer using fully-associative sector translation”. In: *TECS* (July 2007).

- 
- [53] Sungjin Lee, Jihong Kim, and Arvind Mithal. “Refactored Design of I/O Architecture for Flash Storage”. In: *IEEE Computer Architecture Letters* 14 (2015), pp. 70–74.
- [54] Sungjin Lee et al. “Application-managed Flash”. In: *Proceedings of the 14th Usenix Conference on File and Storage Technologies. FAST’16*. Santa Clara, CA: USENIX Association, 2016, pp. 339–353. ISBN: 978-1-931971-28-7. URL: <http://dl.acm.org/citation.cfm?id=2930583.2930609>.
- [55] Scott T. Leutenegger and Daniel Dias. “A Modeling Study of the TPC-C Benchmark”. In: *Proc. SIGMOD’93*.
- [56] Sang-Phil Lim, Sang-Won Lee, and Bongki Moon. “FASTER FTL for Enterprise-Class Flash Memory SSDs”. In: *Proc. SNAPI*. 2010.
- [57] Seung-Ho Lim and Kyu Park. “An efficient NAND flash file system for flash memory storage”. In: *IEEE Transactions on Computers* 55 (Aug. 2006), pp. 906–912. DOI: 10.1109/TC.2006.96.
- [58] Robert Love. *Linux System Programming: Talking Directly to the Kernel and C Library*. O’Reilly Media, Inc., 2007. ISBN: 0596009585.
- [59] Youyou Lu, Jiwu Shu, and Weimin Zheng. “Extending the Lifetime of Flash-based Storage Through Reducing Write Amplification from File Systems”. In: *Proc. FAST’13*.
- [60] Yanfei Lv et al. “Operation-aware Buffer Management in Flash-based Systems”. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. SIGMOD ’11*. Athens, Greece: ACM, 2011, pp. 13–24. ISBN: 978-1-4503-0661-4. DOI: 10.1145/1989323.1989326. URL: <http://doi.acm.org/10.1145/1989323.1989326>.
- [61] Dongzhe Ma, Jianhua Feng, and Guoliang Li. “A Survey of Address Translation Technologies for Flash Memories”. In: *ACM Comput. Surv.* 46.3 (2014), 36:1–36:39.
- [62] Dongzhe Ma, Jianhua Feng, and Guoliang Li. “LazyFTL: A Page-level Flash Translation Layer Optimized for NAND Flash Memory”. In: *Proc. SIGMOD*. 2011.
- [63] F. Masuoka et al. “A new flash E2PROM cell using triple polysilicon technology”. In: *International Electron Devices Meeting*. Vol. 30. 1984, pp. 464–467.
- [64] F. Masuoka et al. “New ultra high density EPROM and flash EEPROM with NAND structure cell”. In: *1987 International Electron Devices Meeting*. Vol. 33. 1987, pp. 552–555. DOI: 10.1109/IEDM.1987.191485.

- 
- 
- [65] S. I. McClean, D. A. Bell, and F. J. McErlean. “Heuristic Methods for the Data Placement Problem”. In: *Journal of the Operational Research Society* 42.9 (Sept. 1991), pp. 767–774. ISSN: 1476-9360. DOI: 10.1057/jors.1991.147. URL: <https://doi.org/10.1057/jors.1991.147>.
- [66] Rino Micheloni, L Crippa, and Alessia Marelli. *Inside NAND flash memories*. Springer Publishing Company, Incorporated, Jan. 2010. DOI: 10.1007/978-90-481-9431-5.
- [67] Changwoo Min et al. “SFS: random write considered harmful in solid state drives”. In: *FAST*. USENIX Association, 2012, p. 12.
- [68] Jayashree Mohan, Rohan Kadekodi, and Vijay Chidambaram. “Analyzing IO Amplification in Linux File Systems”. In: *ArXiv abs/1707.08514* (2017).
- [69] G. E. Moore. “Cramming More Components Onto Integrated Circuits”. In: *Proceedings of the IEEE* 86.1 (Jan. 1998), pp. 82–85. ISSN: 0018-9219. DOI: 10.1109/JPROC.1998.658762.
- [70] Gap-Joo Na, Bongki Moon, and Sang-Won Lee. “In-Page Logging B-Tree for Flash Memory”. In: *Proc. DASFAA’09*.
- [71] Patrick O’Neil et al. “The Log-structured Merge-tree (LSM-tree)”. In: *Acta Informatica* 33.4 (June 1996), pp. 351–385. ISSN: 0001-5903. DOI: 10.1007/s002360050048. URL: <http://dx.doi.org/10.1007/s002360050048>.
- [72] Gihwan Oh et al. “SHARE Interface in Flash Storage for Relational and NoSQL Databases”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. San Francisco, California, USA: ACM, 2016, pp. 343–354. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882910. URL: <http://doi.acm.org/10.1145/2882903.2882910>.
- [73] Oracle. *A Quantitative Comparison between Raw Devices and File Systems for implementing Oracle Databases*. 2004. URL: <https://www.oracle.com/technetwork/database/performance/twp-oracle-hp-files-130020.pdf>.
- [74] Xiangyong Ouyang et al. “Beyond block I/O: Rethinking traditional storage primitives”. In: *Proc. HPCA*. 2011.
- [75] Chanik Park et al. “A reconfigurable FTL architecture for NAND flash-based applications”. In: *TECS* 7.4 (2008), 38:1–38:23.

- 
- [76] Youngwoo Park et al. “PFFS: A Scalable Flash Memory File System for the Hybrid Architecture of Phase-change RAM and NAND Flash”. In: *Proceedings of the 2008 ACM Symposium on Applied Computing*. SAC '08. Fortaleza, Ceara, Brazil: ACM, 2008, pp. 1498–1503. ISBN: 978-1-59593-753-7. DOI: 10.1145/1363686.1364038. URL: <http://doi.acm.org/10.1145/1363686.1364038>.
- [77] Ilia Petrov, Robert Gottstein, and Sergey Hardock. “DBMS on Modern Storage Hardware”. In: *Proc. ICDE*. 2015.
- [78] Ilia Petrov et al. “Native Storage Techniques for Data Management”. In: *Proc. ICDE*. 2019.
- [79] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. “Transactional Flash”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 147–160. URL: <http://dl.acm.org/citation.cfm?id=1855741.1855752>.
- [80] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 3rd ed. New York, NY, USA: McGraw-Hill, Inc., 2003. ISBN: 9780072465631.
- [81] *Samsung Multi-stream Technology*. [https://www.samsung.com/semiconductor/global.semi.static/Samsung\\_Multistream\\_Technology-1.pdf](https://www.samsung.com/semiconductor/global.semi.static/Samsung_Multistream_Technology-1.pdf). 2017.
- [82] *Samsung V-NAND technology*. [http://www.samsung.com/us/business/oem-solutions/pdfs/V-NAND\\_technology\\_WP.pdf](http://www.samsung.com/us/business/oem-solutions/pdfs/V-NAND_technology_WP.pdf). 2014.
- [83] C. Sandhya et al. “Effect of SiN on Performance and Reliability of Charge Trap Flash (CTF) Under Fowler–Nordheim Tunneling Program/Erase Operation”. In: *IEEE Electron Device Letters* 30.2 (Feb. 2009), pp. 171–173. ISSN: 0741-3106.
- [84] Brian R. Santo. “25 Microchips That Shook the World”. In: *IEEE Spectrum* (May 2009).
- [85] Caetano Sauer, Goetz Graefe, and Theo Härder. “An empirical analysis of database recovery costs”. In: *RDSS*. 2014.
- [86] Jiri Schindler, Anastassia Ailamaki, and Gregory R. Ganger. “Lachesis: Robust Database Storage Management Based on Device-specific Performance Characteristics”. In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*. VLDB '03. Berlin, Germany: VLDB Endowment, 2003, pp. 706–717. ISBN: 0-12-722442-4. URL: <http://dl.acm.org/citation.cfm?id=1315451.1315512>.

- 
- 
- [87] Sudharsan Seshadri et al. “Willow: A User-programmable SSD”. In: *Proc. OSDI’14*. 2014.
- [88] Ji-Yong Shin et al. “FTL Design Exploration in Reconfigurable High-performance SSD for Server Applications”. In: *Proc. ICS’09*. 2009.
- [89] Radu Stoica and Anastasia Ailamaki. “Improving Flash Write Performance by Using Update Frequency”. In: *Proceedings of the VLDB Endowment* 6.9 (July 2013), pp. 733–744. ISSN: 2150-8097. DOI: 10 . 14778 / 2536360 . 2536372. URL: <http://dx.doi.org/10.14778/2536360.2536372>.
- [90] Kang-Deog et al. Suh. “A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme”. In: *Proc. ISSCC’95*.
- [91] Billy Tallis. *Intel And Micron Launch First QLC NAND: Micron 5210 ION Enterprise SATA SSD*. <https://www.anandtech.com/show/12744/micron-launches-first-qlc-nand-micron-5210-ion-enterprise-sata-ssd>. 2018.
- [92] *TATP Benchmark Description (Version 1.0)*. [http://tatpbenchmark.sourceforge.net/TATP\\_Description.pdf](http://tatpbenchmark.sourceforge.net/TATP_Description.pdf). 2009.
- [93] *Technical Note. NAND Flash 101: An Introduction to NAND Flash and How to Design It Into Your Next Product*. Micron Technology, Inc. Apr. 1, 2010. URL: <https://www.ece.umd.edu/~blj/CS-590.26/micron-tn2919.pdf> (visited on 07/02/2020).
- [94] *The OpenSSD Project*. [http://www.openssd-project.org/wiki/The\\_OpenSSD\\_Project](http://www.openssd-project.org/wiki/The_OpenSSD_Project). 2014.
- [95] *TPC BENCHMARK™ B Specification*. [http://www.tpc.org/tpcb/spec/tpcb\\_current.pdf](http://www.tpc.org/tpcb/spec/tpcb_current.pdf). 1994.
- [96] *TPC BENCHMARK™ C Specification*. [www.tpc.org/tpcc/spec/tpcc\\_current.pdf](http://www.tpc.org/tpcc/spec/tpcc_current.pdf). 2010.
- [97] Animesh Trivedi et al. “Unified High-performance I/O: One Stack to Rule Them All”. In: *Proc. HotOS’13*. 2013.
- [98] Kristian Vättö. *Samsung SSD 840 (250GB) Review*. <https://www.anandtech.com/show/6337/samsung-ssd-840-250gb-review/2>. 2012.
- [99] Tobias Vincon et al. “nativeNDP: Processing Big Data Analytics on Native Storage Nodes”. In: *Proc. ADBIS*. 2019.
- [100] Tobias Vincon et al. “nKV in Action: Accelerating KV-Stores on Native Computational Storage with Near-Data Processing”. In: *Proc. PVLDB*. 2020.

- 
- 
- [101] Tobias Vincon et al. “NoFTL-KV: Tackling Write-Amplification on KV-Stores with Native Storage Management”. In: *Proc. EDBT*. 2018.
- [102] Zev Weiss et al. “ANViL: Advanced Virtualization for Modern Non-volatile Memory Devices”. In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. FAST’15. Santa Clara, CA: USENIX Association, 2015, pp. 111–118. ISBN: 978-1-931971-201. URL: <http://dl.acm.org/citation.cfm?id=2750482.2750491>.
- [103] Bill Wong. *Pseudo-SLC Flash Provides Design Flexibility*. 2013. URL: <http://electronicdesign.com/site-files/electronicdesign.com/files/uploads/2013/09/FAQs-Toshiba-September.pdf>.
- [104] Louis Woods, Zsolt István, and Gustavo Alonso. “Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading”. In: *Proc. VLDB Endow*. 7.11 (July 2014), pp. 963–974. ISSN: 2150-8097. DOI: 10.14778/2732967.2732972. URL: <http://dx.doi.org/10.14778/2732967.2732972>.
- [105] Louis Woods, Jens Teubner, and Gustavo Alonso. “Less Watts, More Performance: An Intelligent Storage Engine for Data Appliances”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’13. New York, New York, USA: ACM, 2013, pp. 1073–1076. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2463685. URL: <http://doi.acm.org/10.1145/2463676.2463685>.
- [106] Wei Xie and Yong Chen. “An Adaptive Separation-Aware FTL for Improving the Efficiency of Garbage Collection in SSDs”. In: May 2014, pp. 552–553. ISBN: 978-1-4799-2784-5. DOI: 10.1109/CCGrid.2014.52.
- [107] *YAFFS - A NAND flash filesystem*. <https://yaffs.net/sites/yaffs.net/files/yaffs.pdf>. 2007.
- [108] Jinsoo Yoo et al. “OrcFS: Orchestrated File System for Flash Storage”. In: *ACM Transactions on Storage (TOS)* 14.2 (Apr. 2018), 17:1–17:26. ISSN: 1553-3077. DOI: 10.1145/3162614. URL: <http://doi.acm.org/10.1145/3162614>.
- [109] Cai Yu et al. “Error Analysis and Retention-aware Error Management for NAND Flash Memory”. In: *Intel Tech. Journal* 17.1 (2013), pp. 140–164.
- [110] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. “ParaFS: A Log-structured File System to Exploit the Internal Parallelism of Flash Devices”. In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC ’16. Denver, CO, USA: USENIX Association, 2016, pp. 87–100. ISBN: 978-1-931971-30-0. URL: <http://dl.acm.org/citation.cfm?id=3026959.3026968>.

- 
- [111] Wenhui Zhang et al. “PA-SSD: A Page-Type Aware TLC SSD for Improved Write/Read Performance and Storage Efficiency”. In: *Proceedings of the 2018 International Conference on Supercomputing*. ICS ’18. Beijing, China: ACM, 2018, pp. 22–32. ISBN: 978-1-4503-5783-8. DOI: 10.1145/3205289.3205319. URL: <http://doi.acm.org/10.1145/3205289.3205319>.
- [112] Zhengdong Xia and Tianming Bu. “The implementation of flash-aware buffer replacement algorithms in PostgreSQL”. In: *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*. Aug. 2015, pp. 1215–1219. DOI: 10.1109/FSKD.2015.7382115.
- [113] You Zhou et al. “An Efficient Page-level FTL to Optimize Address Translation in Flash Memory”. In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys ’15. Bordeaux, France: ACM, 2015, 12:1–12:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741949. URL: <http://doi.acm.org/10.1145/2741948.2741949>.



---

## A. Appendix

---

### A.1. OpenSSD Jasmine

The Jasmine Board (see Figure A.1) was developed under the OpenSSD project [94]. The board was designed to enable Flash research and evaluate different Flash management schemes and algorithms on the real Flash hardware. It comprises the following components: two Flash memory modules each with four dual-die K9LCG08U1M 8GB MLC Flash packages, 64MB DRAM memory, 96KB SRAM memory and an Indilinx ARM controller with 87.5MHz. The board connects to the host via SATA 2.0 (3 Gbps) interface. The firmware contains several popular FTL schemes and allows for implementing custom ones.

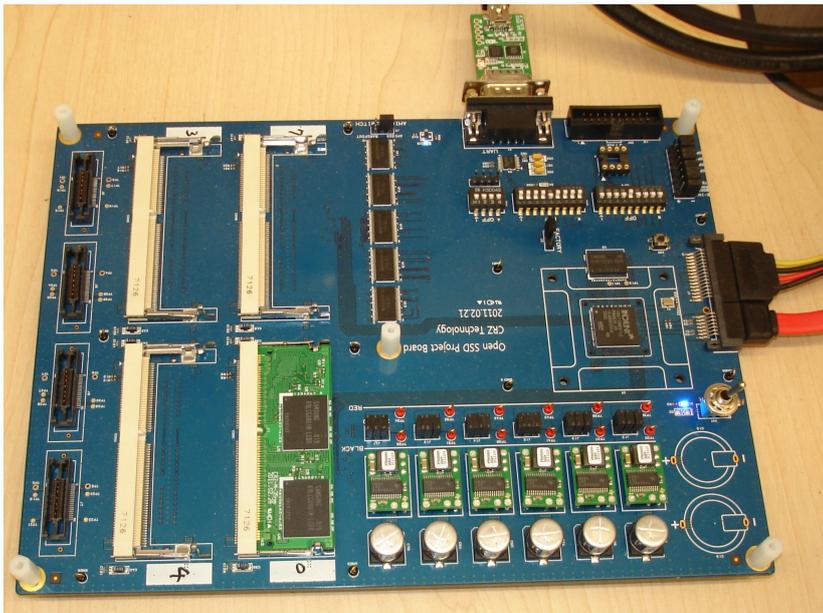


Figure A.1.: OpenSSD Jasmine board.

---

The implementation of the native Flash interface (see Chapter 4) on the OpenSSD board led to several challenges since underlying SATA protocol is naturally block device oriented and is hard to extend. To implement the new native Flash commands we partly used the reserved command codes, partly "overloaded" some of the original SATA commands.

Although the internal architecture of on-board Flash supports I/O parallelism, the current version of the firmware does not allow for utilizing it. According to the specification the on-board SATA controller supports NCQ, but the firmware does not implement it. Furthermore, the absence of additional documentation about the SATA controller overcomplicates a custom NCQ implementation. In other words, the board only supports serial synchronous execution of SATA commands. The workaround provided by the OpenSSD developers comprises a combination of special Write Cache use and two internal I/O queues for READ and WRITE I/Os respectively does not really solve the problem since the READ I/Os (which are dominated in OLTP workloads) are still executed synchronously.

Two other limitations of OpenSSD Jasmine board are: (1) inability to access OOB area of Flash pages; and (2) inability to modify the ECC engine of Flash controller. Fortunately for us, the default implementation of ECC has only error-detection mechanism, and does not perform correction of detected bit-errors. This fact has allowed us to implement and evaluate the IPA approach (see Chapter 7), while modified BCH-ECC needed for IPA has been implemented on server side within the DBMS.