System Co-Design and Data Management for Flash Devices

VLDB'2011

Philippe Bonnet,
ITU, Denmark

Luc Bouganim,
INRIA, France

Ioannis Koltsidas
IBM Research, Switzerland

Stratis D. Viglas
University of Edinburgh, United Kingdom

Flash Devices (SSD)

IO don't matter

CPU is the critical resource

Just a SATA drive

I can readily plug in flash devices in my server. What is the big deal?

Why Bother?

Disk is disk

~650 mio units shipped in 2010

PCM is coming

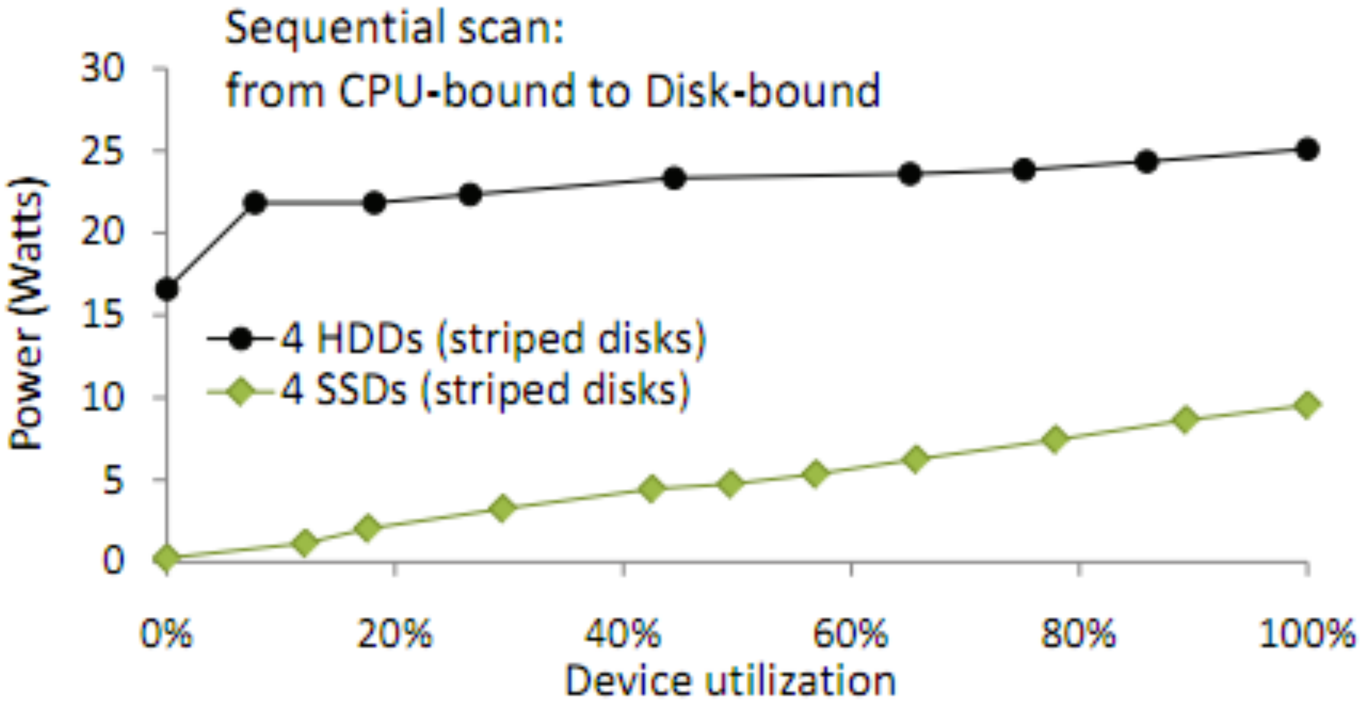
100x faster
10 mio write cycles

[Papandreou et al., IMW 2011]

Some Trends ...

	2000		2010
HDD Capacity	200 GB	**x10**	2 TB
HDD GB/$	0,05	**x600**	30
HDD IOPS	200	**x1**	200
SSD Capacity	14 GB (2001)	**x20**	256 GB
SSD GB/$	3 x10E-4	**x1000**	0,5
SSD IOPS	10E3 (SCSI)	**x1000**	10E6+ (PCIe) 5x10E3+ (SATA)
PCM Capacity			2x10E5 cells, 4 bits/cell
PCM IOPS			10E6+ (1 chip)

… and a Fact



[Tsorigiannis et al. 2010]

Flash-based SSDs do nothing well!
They offer high throughput at low energy consumption.

SSD-based Systems

With more than 1,000 stores, Danish Supermarket group is one of Denmark’s largest retailers.
To help keep up with customer needs, the company manages more than 10 terabytes of **business intelligence** data.

Database Appliances

Neteeza Twin-fin

Oracle Exadata

SSD-based blades

Scaled up

Super Micro 6026

Scaled down

Amdahl blade [Szalay et al., 2009]

IOs matter. Systems are being designed and commercialized for efficient data management for flash devices.

Block Device

SSDs and HDDs provide the same memory abstraction: a block device interface

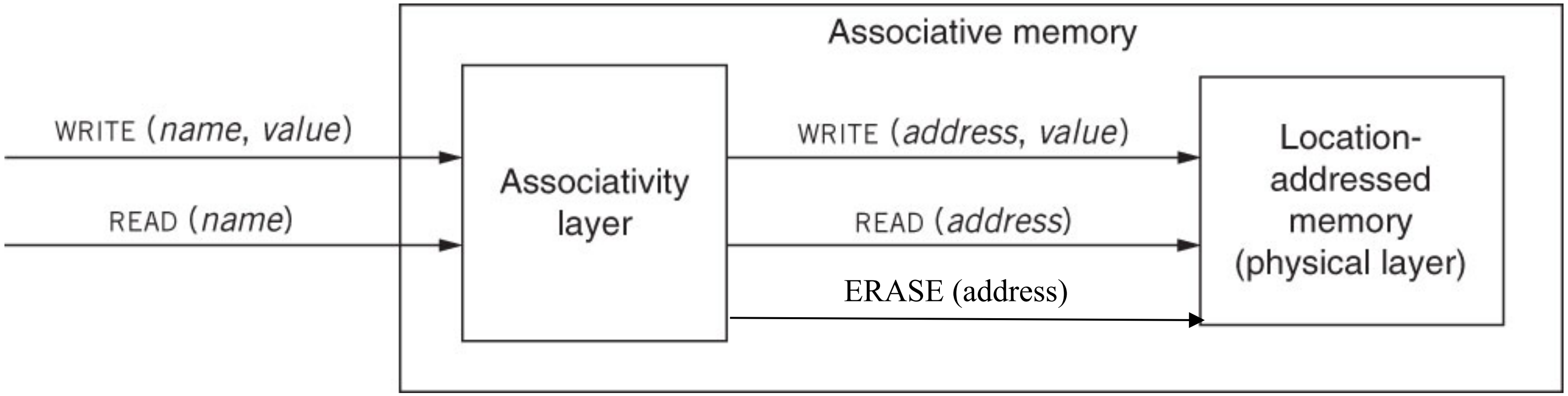
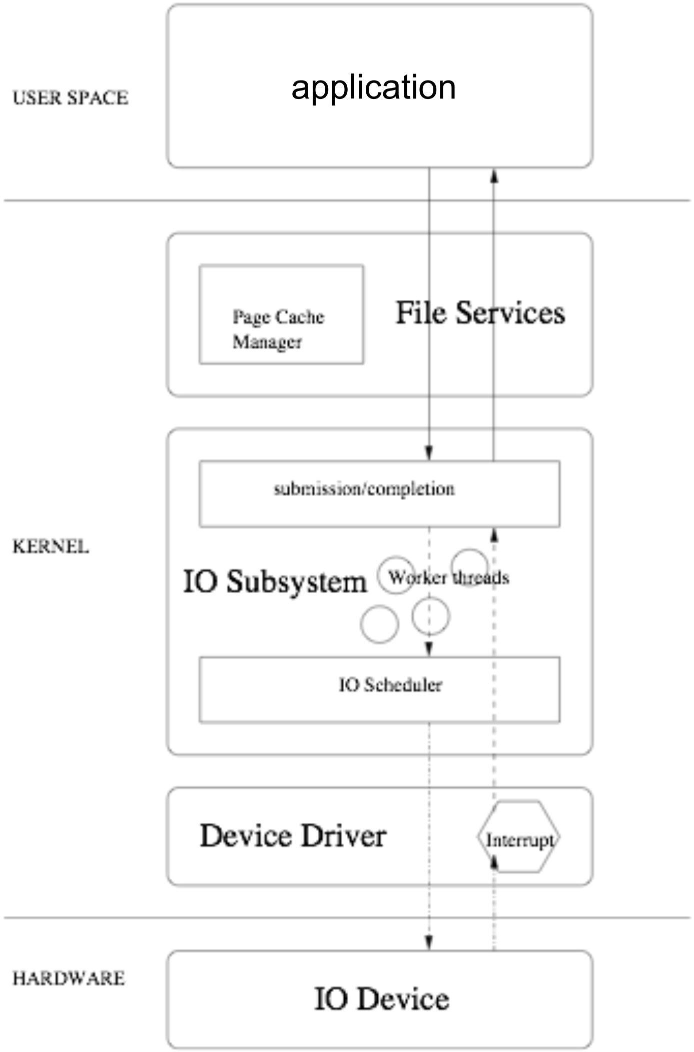


Figure courtesy of Koschaak and Saltzer

Strong Modularity

SSDs and HDDs provide the same memory abstraction: a block device interface

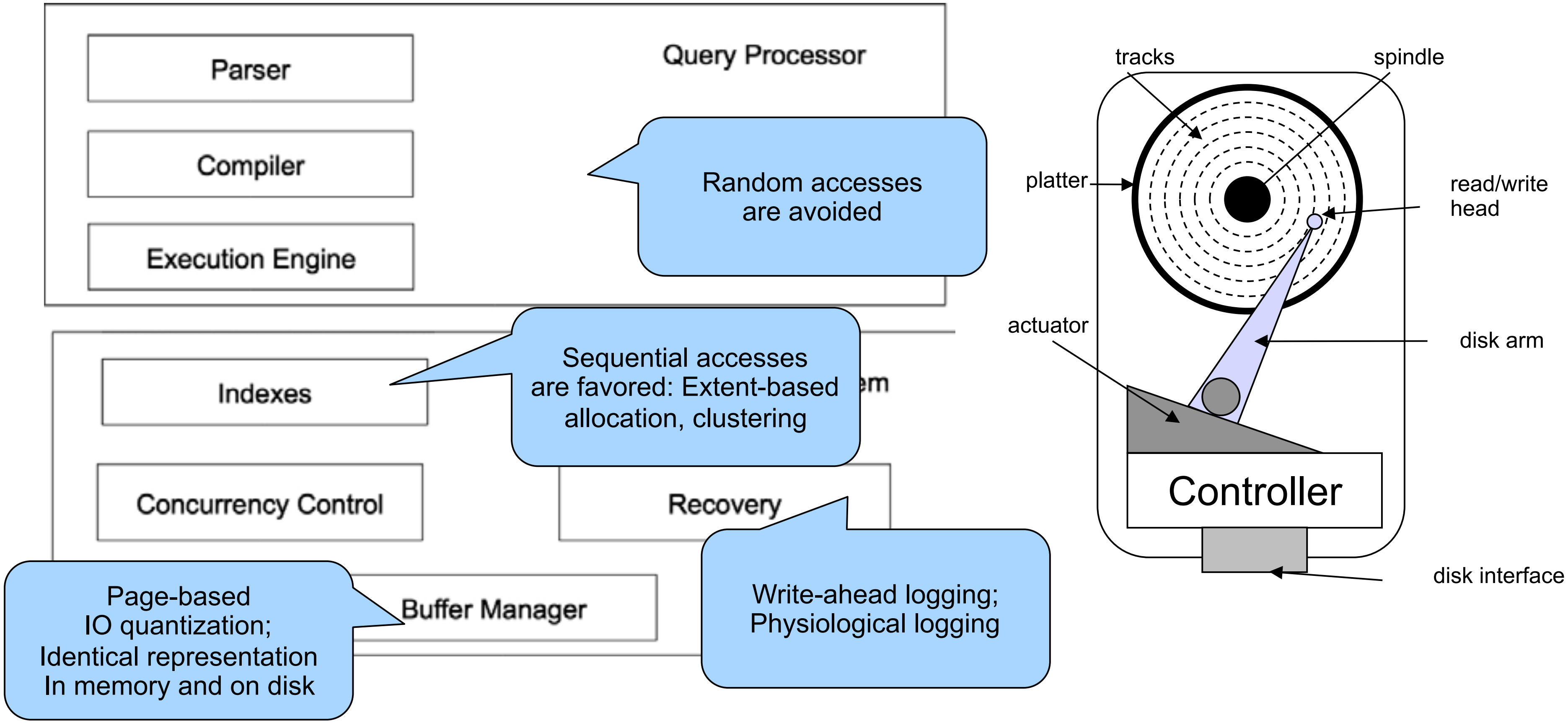


=> There should be no impact on application (e.g., DBMS) ?

Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

Design Assumptions

=> Actually DBMS design very much based on disk characteristics:
(1) locality in the logical space preserved in the physical space,
(2) sequential access is faster than random access.



Random accesses are avoided

Sequential accesses are favored: Extent-based allocation, clustering

Page-based IO quantization; Identical representation In memory and on disk

Write-ahead logging; Physiological logging

How do flash devices impact DBMS design?

(Bottom-up) We need to understand flash devices a bit better.

If they exhibit stable properties

=> Design principles for data management

If they do not exhibit stable properties

=> How to tackle the increased complexity?

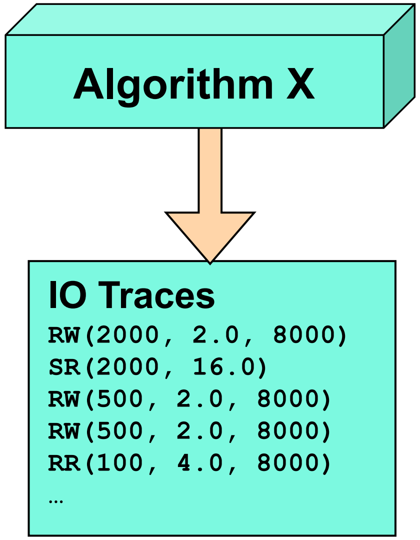
(Top-down) We make assumptions about the behaviour of flash devices, and we design adapted DBMS components. We then need to make sure that (at least some) flash devices actually fit our assumptions.

Tutorial Outline

1. Introduction (Philippe)
2. Flash devices characteristics (Luc)
3. Data management for flash devices (Stratis)
4. Two outlooks (Stratis & Philippe)

A short motivating story (1)

- **Alice**, **Bob**, **Charlie** and **Dave** want to measure the performance of a given data intensive algorithm for flash devices…
- They use different strategies but start from the same IO traces of that algorithm and own an MTRON and 2 identical INTEL X25-M SSDs.



Algorithm X

IO Traces

```
RW(2000, 2.0, 8000)
SR(2000, 16.0)
RW(500, 2.0, 8000)
RW(500, 2.0, 8000)
RR(100, 4.0, 8000)
…
```

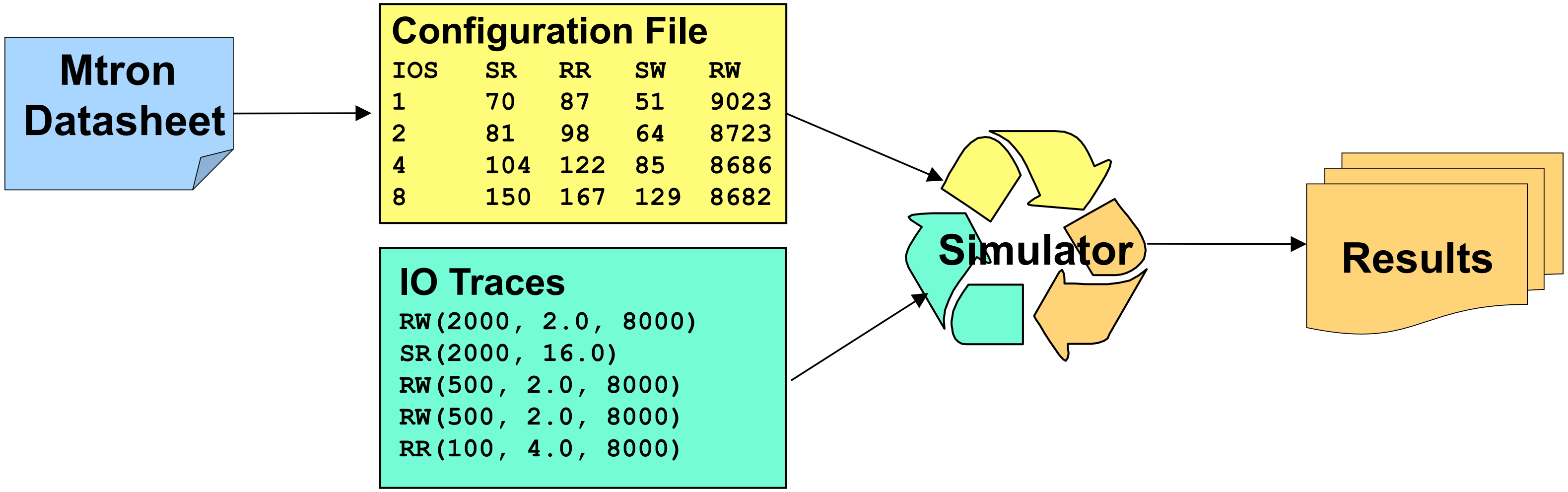
Same model
Same firmware

Never used

Used

A short motivating story (2): Alice & Bob

- **Alice** believes in datasheets. She builds a simple SSD simulator configured with basic SSD performance numbers.
- She takes the SSD performance numbers from the datasheet and runs the simulator using the traces….



Mtron Datasheet

Configuration File

```
IOS     SR    RR    SW    RW
1       70    87    51    9023
2       81    98    64    8723
4       104   122   85    8686
8       150   167   129   8682
```

IO Traces

```
RW(2000, 2.0, 8000)
SR(2000, 16.0)
RW(500, 2.0, 8000)
RW(500, 2.0, 8000)
RR(100, 4.0, 8000)
```

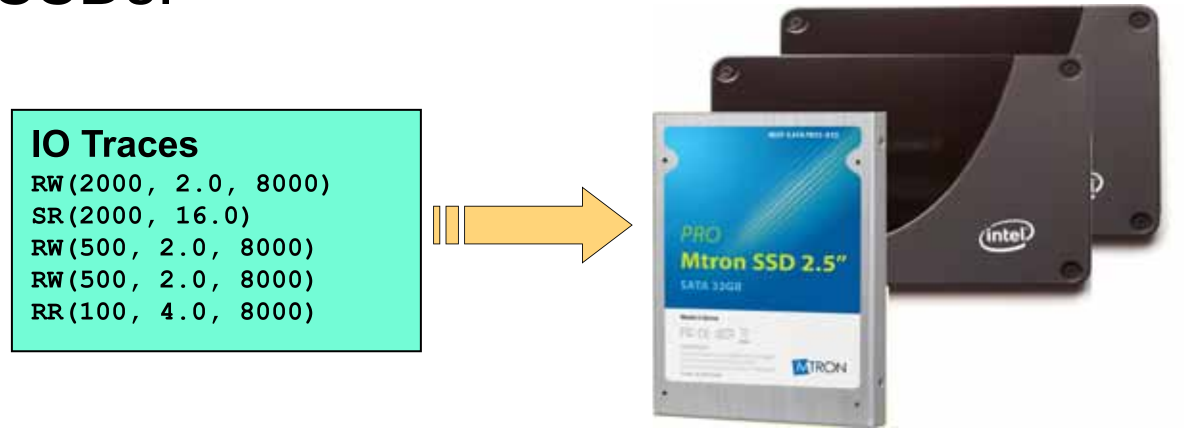
Simulator

Results

- **Bob**, does not believe in datasheets. He runs simple tests on both SSDs to obtain the basic performance numbers…He then runs Alice’s simulator on the traces with his numbers

A short motivating story (3): Charlie & Dave

- **Charlie**, does not believe in Bob! He is more cautious and runs long tests on the same SSDs and obtain his own basic performance numbers. Then, he proceeds as Bob.
- **Dave** does not like simulation and runs the traces directly on the SSDs.

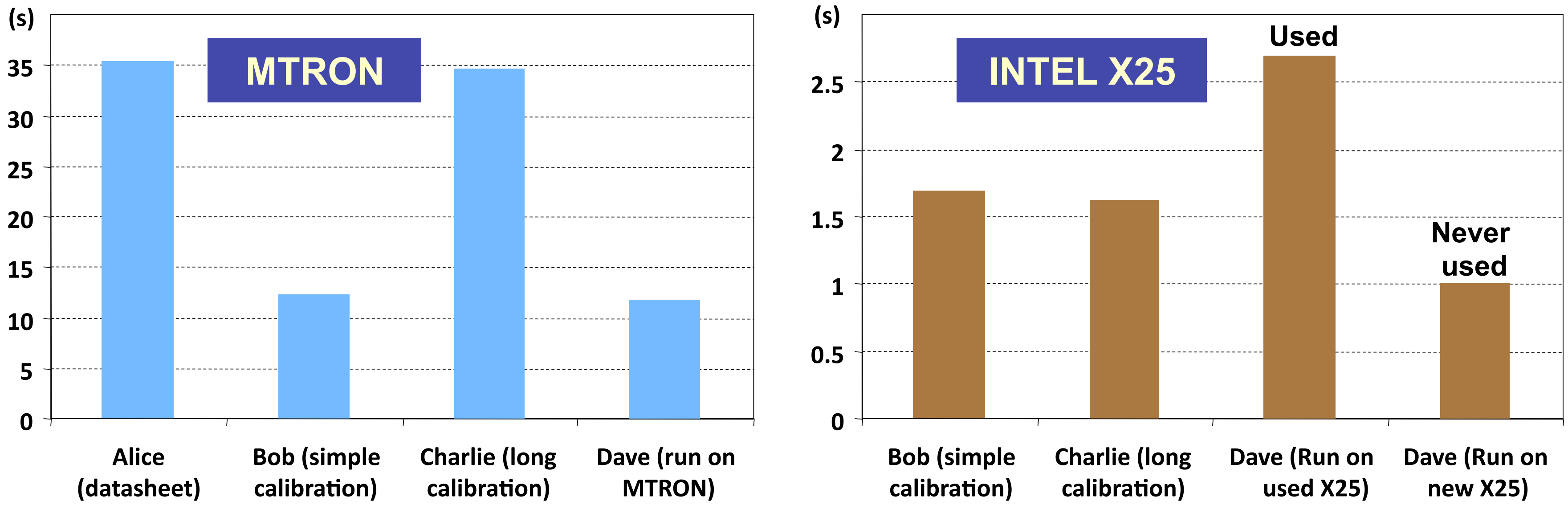


IO Traces

```
RW(2000, 2.0, 8000)
SR(2000, 16.0)
RW(500, 2.0, 8000)
RW(500, 2.0, 8000)
RR(100, 4.0, 8000)
```

What is your take on the resulting measures?

A short motivating story (4): Results



- Mtron and Intel devices **behave** differently
- **Identical Intel devices behave differently**

➔ Confidence in performance measurements is very low!

- **Modeling flash devices seems difficult**
- **What about designing algorithms for flash devices ?**
  - e.g., database systems, operating systems, applications ?

Outline of the first part of this tutorial

Goal: understand the impact of flash memory on software (DBMS) design <u>and vice-versa</u>

- We study flash chips, explaining their constraints and trends
- We then consider flash devices as black boxes and try to understand their performance behavior (uFLIP).
  Goal: Find a simple model, basis for a DBMS design
- We hit a wall with the black box approach → we open the box, i.e., the FTL, and look at FTL techniques.
- Finally, we propose an alternative to complex FTLs, better adapted for DBMS design.

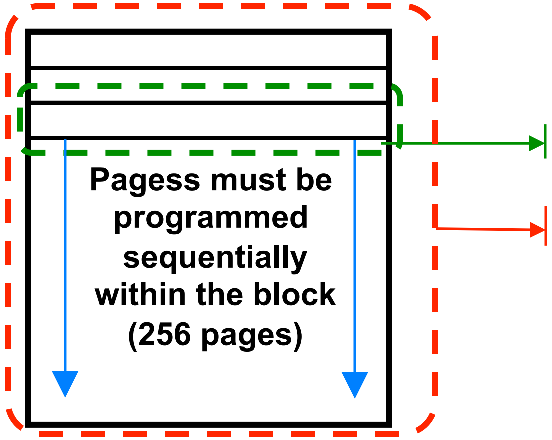
NAND Flash chip performance!

- A **single** flash chip offers **great performance**
  - e.g., 40 MB/s Read, 10 MB/s Program
  - Random access is **as fast as** sequential access
  - Low energy consumption

THE BAD

The severe constraints of NAND flash chips!

- C1: Program granularity:
  - Program must be performed at flash page granularity (2KB-16KB)
- C2: Must erase a block before updating a page (256 KB-1MB)
- C3: Pages must be programmed sequentially within a block
- C4: Limited lifetime (from 10^4 up to 10^5 erase operations)


Pagess must be programmed sequentially within the block (256 pages)

Program granularity: a page (32 KB)

Erase granularity: a block (1 MB)

The facts you need — fast

Flash chips

~~FOR~~ BY

DUMMIES®

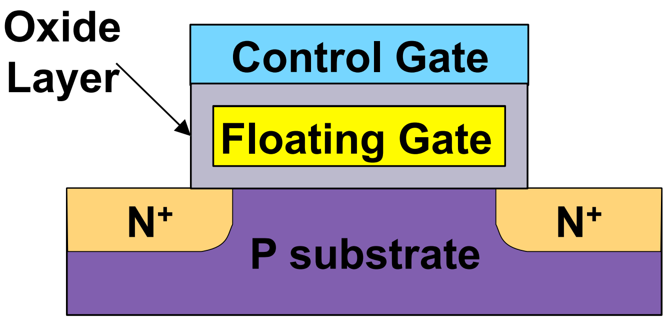
QUICK REFERENCE

A bit of electronic to understand flash chip constraints and trends

Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

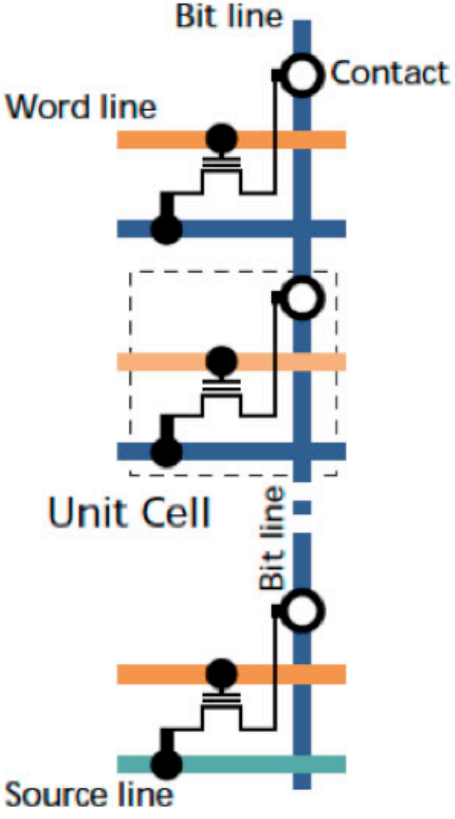
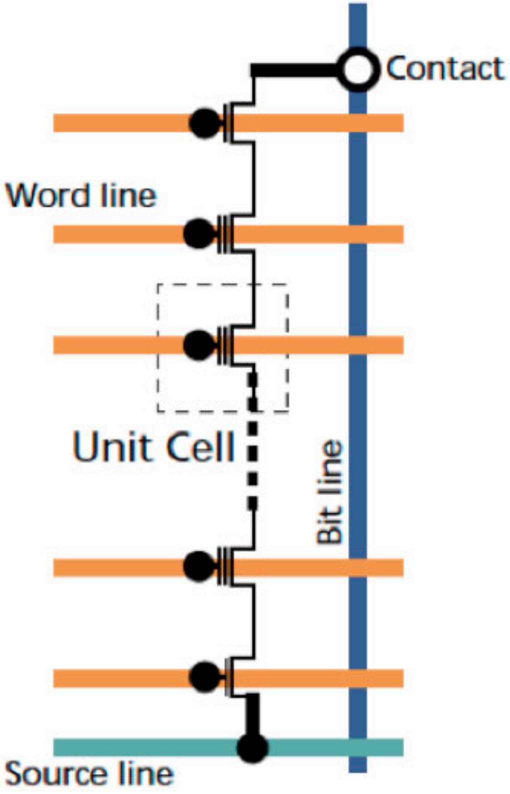
Flash cells

- Flash cell: resembles a semiconductor transistor
  - 2 gates instead of 1
  - Floating gate insulated all around by an oxide layer
- Electrons placed on the floating gate are trapped
- The floating gate will not discharge for many years



Flash cell: a floating gate transistor

Flash cells: NOR vs NAND

	NOR	NAND
Cell Array		
Cell Size	$10F^2$	$4F^2$

NOR

- Quick read (Byte)
- Slow prog. (Byte)
- Slow erase
- XIP → Code

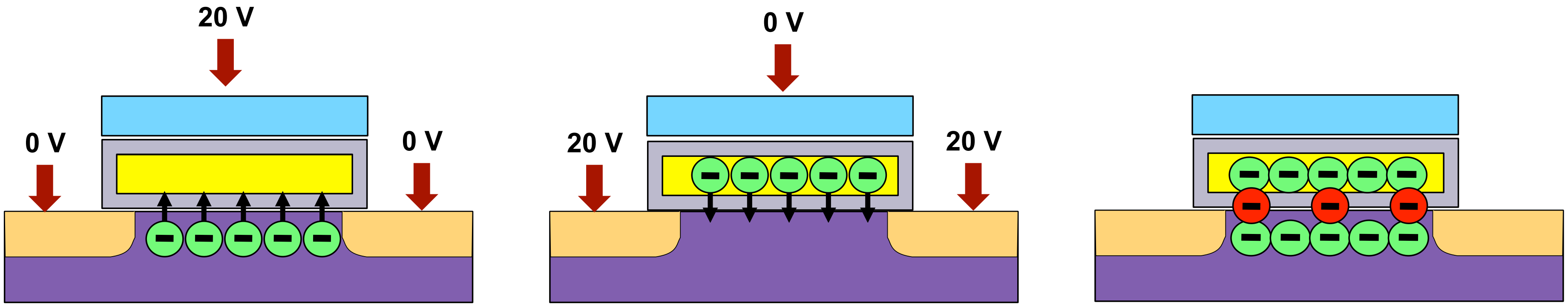
NAND

- Slower read (Page)
- Quicker prog. (Page)
- Quicker erase (Block)
- Files, data

Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

NAND Flash cells mode of operation

- **Programming**: Apply a high voltage to the control gate
  → electrons get trapped in the floating gate
- **Erasing**: Apply a high voltage to the substrate
  → electrons are removed from the floating gate
- **Reading**: the charge changes the threshold voltage of the cell
  - Single level cell (SLC) store one bit per cell: charged = 0, not charged = 1
  - Multi level cell (MLC) store 2 bits per cell (4 levels)
- After a number of program/erase cycle, electrons are getting trapped in the oxyde layer → **End of life of the cell**



Programming

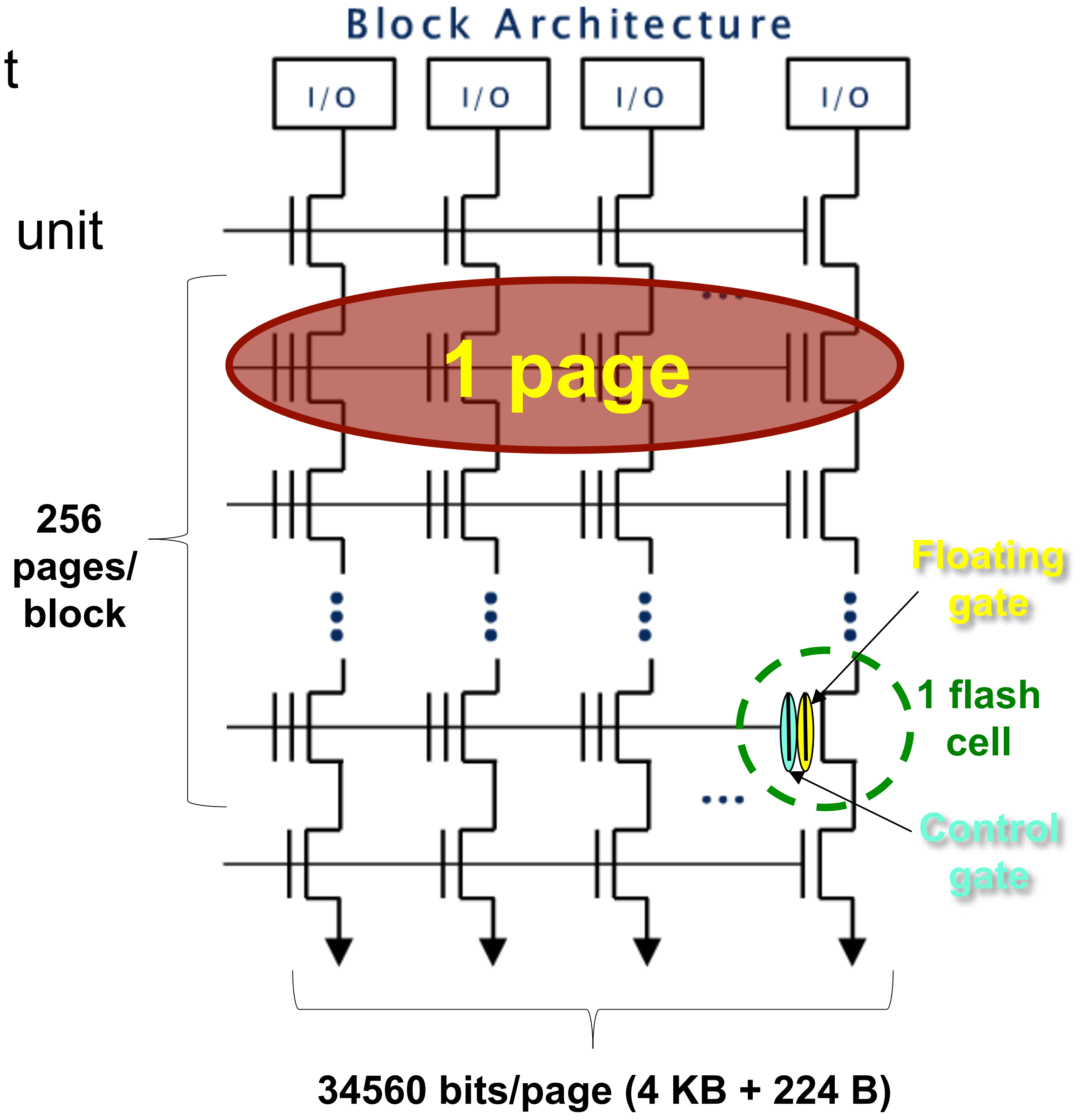
Erasing

Wear out cell

Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

NAND Architecture & timings

- Based upon independent blocks (4 Mio cells here)
- Block: smallest erasable unit
- Page: smallest programmable unit



Geometry & Timings

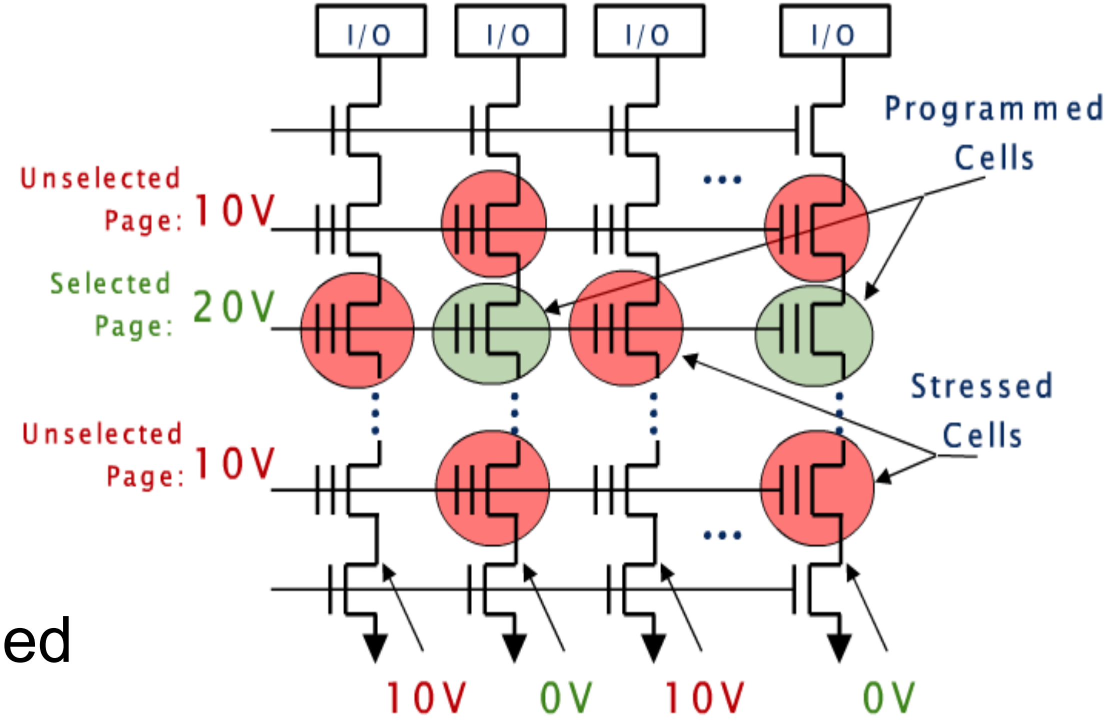
	MLC
Page Size	4 KB
Block Size	1 MB
Chip Size	16 GB
Read Page (µs)	150
Program Page (µs)	1000
Erase Block (µs)	3000

NAND flash MICRON MLC:
MT29F128G08CJABB

Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

Program Disturb

- Some cells not being programmed receive elevated voltage stress (near the cells being programmed)
- Stressed cells can appear weakly programmed



Reducing program disturb:

- Use Error Correction Code to recover errors
- **Program page sequentially within a block**

Cooke (FMS 2007)

Impact on flash chip IOs

- Flash cell technology
  - ➔ Limited lifetime for entire blocks (when a cell wear out, the entire block is marked as failed).
- NAND Layout and structure
  - ➔Block is the smallest erase granularity
- Program Disturb
  - ➔ Page is the smallest program granularity (¼ for SLC)
  - ➔ Pages must me programmed sequentially within a block
  - ➔ Use of ECC is mandatory → ECC unit is the smallest read unit (generally 1 or ¼ page)

Flash chips: trends

- Density increases (price decreases)
  - NAND process migration: faster than Moore's Law (today 20 nm)
  - More bits/cell:
    - SLC (1), MLC (2), TLC (3)
- Flash chip layout and structure: larger, parallel
  - Larger blocks (32 → 256 Pages)
  - Larger pages: 512 B (old SLC) → 16KB (future TLC)
  - Dual plane Flash → parallelism within the flash chip
- Lifetime decreases
  - 100 000 (SLC), 10 000 (MLC), 5000 (TLC)
- ECC size increases
- Basic performance decreases
  - Compensated by parallelism

Abraham (FMS 2011), StorageSearch.com

Outline of the first part of this tutorial

Goal: understand the impact of flash memory on software (DBMS) design and vice-versa

- We study flash chips, explaining their constraints and trends
- We then consider flash devices as black boxes and try to understand their performance behavior (uFLIP)
- We hit a wall with the black box approach → we open the box, i.e., the FTL, and look at FTL techniques
- Finally, we propose an alternative to complex FTLs, better adapted for DBMS design

THE GOOD

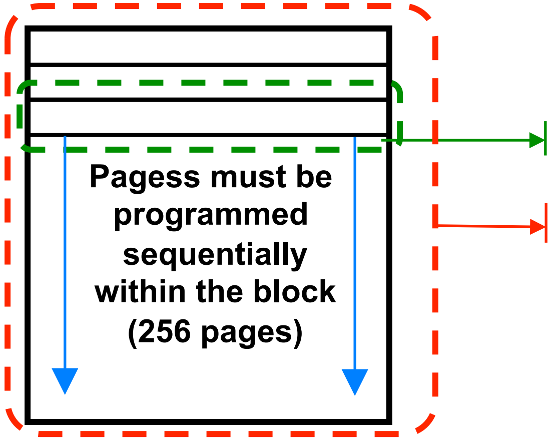
The hardware!

- A **single** flash chip offers **great performance**
  - e.g., 40 MB/s Read, 10 MB/s Program
  - Random access is **as fast as** sequential access
  - Low energy consumption
- **A flash device contains many (e.g., 32, 64) flash chips and provides inter-chips parallelism**
- **Flash devices may include some (power-failure resistant) SRAM**

THE BAD

The severe constraints of flash chips!

- C1: Program granularity:
    - Program must be performed at flash page granularity
- C2: Must erase a block before updating a page
- C3: Pages must be programmed sequentially within a block
- C4: Limited lifetime (from 10^4 up to 10^6 erase operations)



Pagess must be programmed sequentially within the block (256 pages)

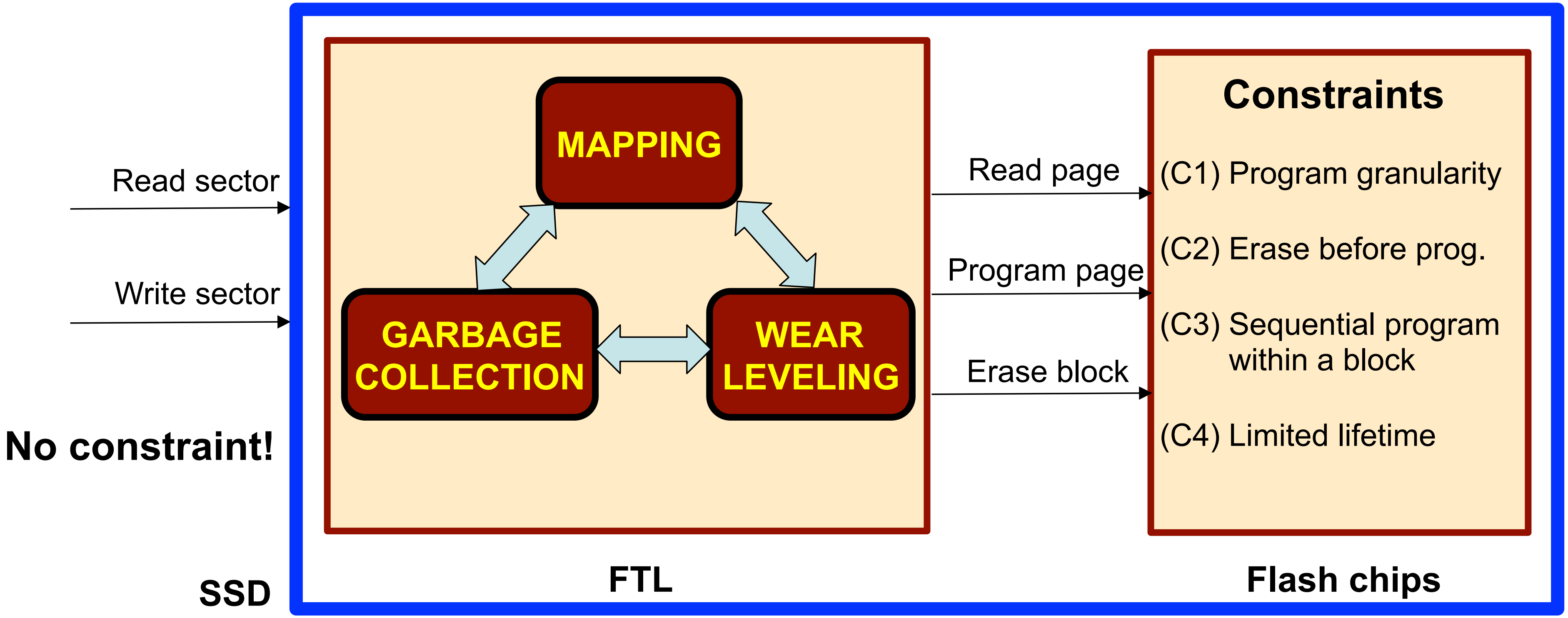
Program granularity: a page (32 KB)

Erase granularity: a block (1 MB)

AND THE
FTL

The software!, the Flash Translation Layer

- emulates a classical block device and handle flash constraints



Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

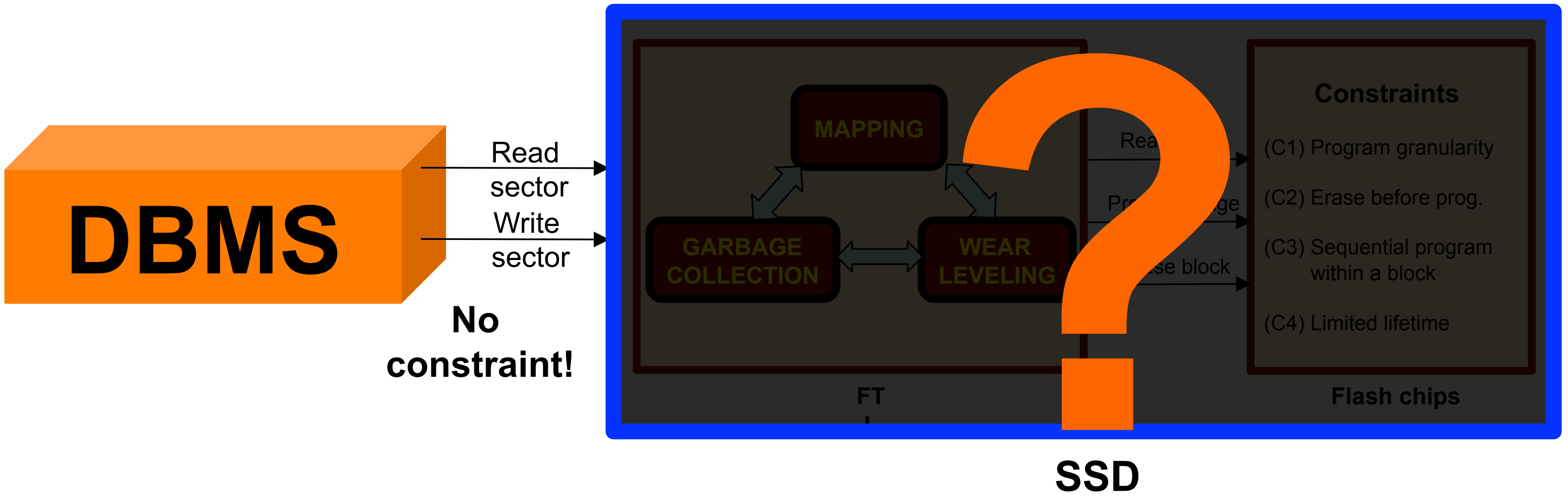
Flash devices are black boxes!

- **Flash devices <u>are not</u> flash chips**
  - Do not behave as the flash chip they contain
  - No access to the flash chip API but only through the device API
  - Complex architecture and software, proprietary and not documented

→ **Flash devices are black boxes !**

→ **DBMS design cannot be based on flash chip behavior!**

We need to understand flash devices behavior!



Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

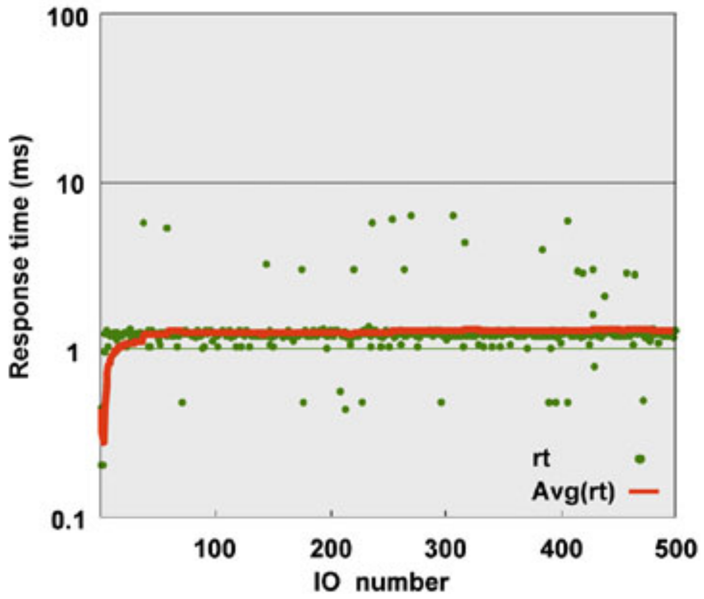
Understanding flash devices behavior

- Define an experimental benchmark which can exhibit the behavior of flash devices.

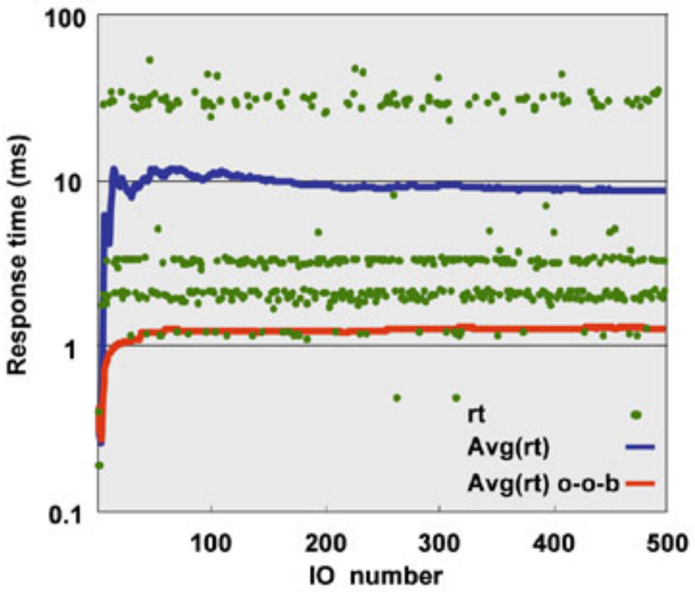
- Define a **broad benchmark**
  - No safe assumption can be made on the device behavior (black box)
    - e.g., Random writes are expensive…
  - No safe assumption on the benchmark usage!

- Design a **sound benchmarking methodology**
  - IO cost is highly variable and depends on the whole device history!

Methodology (1): Device state



Random Writes – Samsung SSD
Out of the box



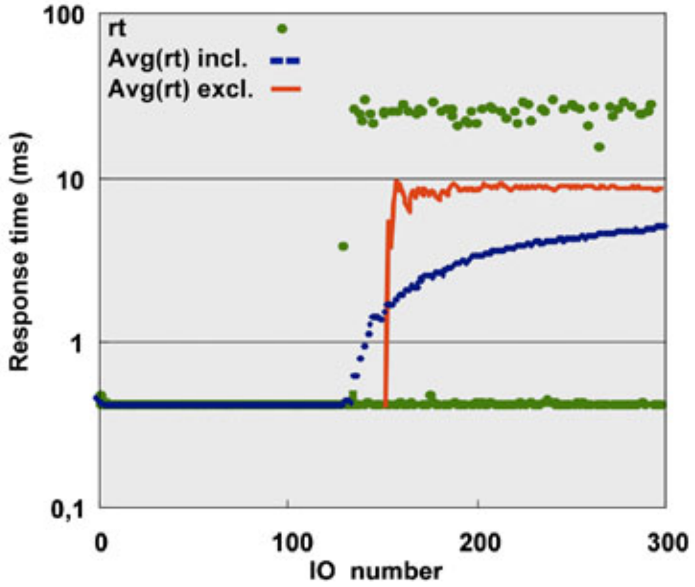
Random Writes – Samsung SSD
After filling the device

➔ Enforce a **well-defined device state**

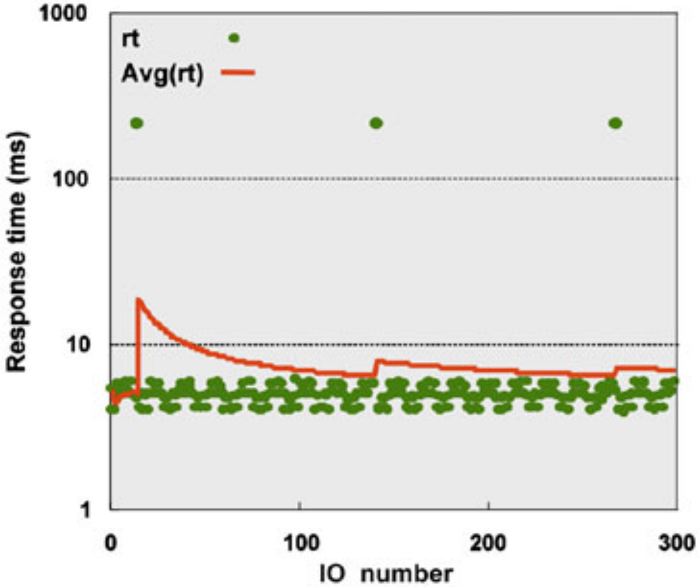
- performing random write IOs of random size on the whole device
- The alternative, sequential IOs, is less stable, thus more difficult to enforce

Methodology (2): Startup and running phases

- When do we reach a steady state? How long to run each test?



Startup and running phases for the Mtron SSD (RW)

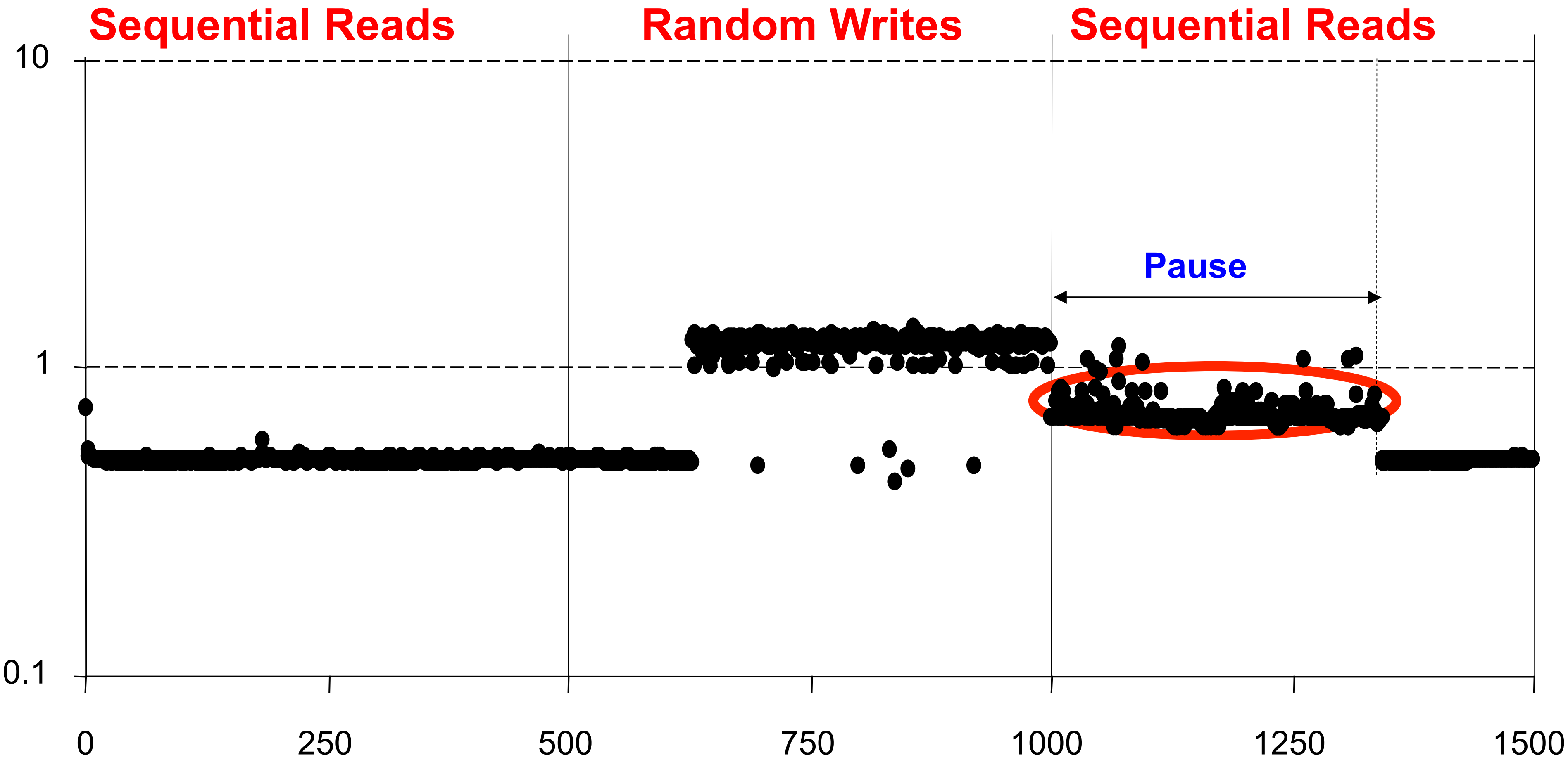


Running phase for the Kingston DTI flash Drive (SW)

➔ **Startup and running phase**: Run experiments to define

- **IOIgnore**: Number of IOs ignored when computing statistics
- **IOCount**: Number of measures to allow for convergence of those statistics.

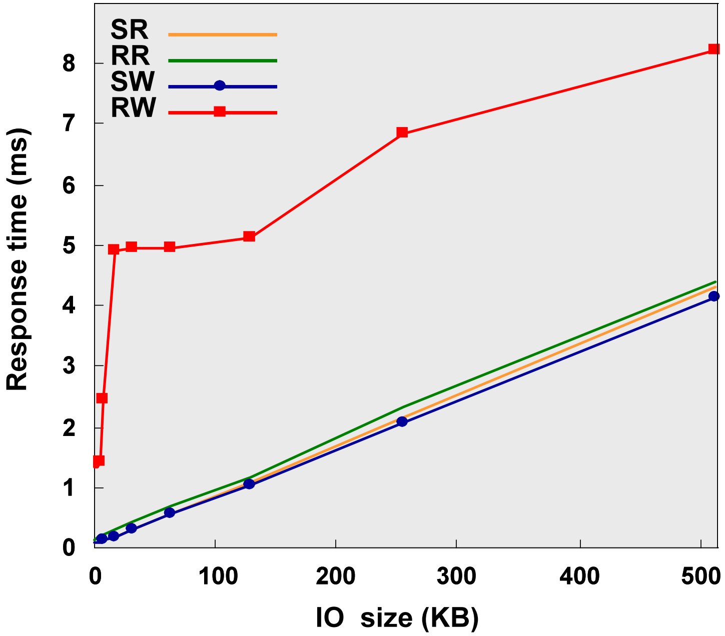
Methodology (3): Interferences



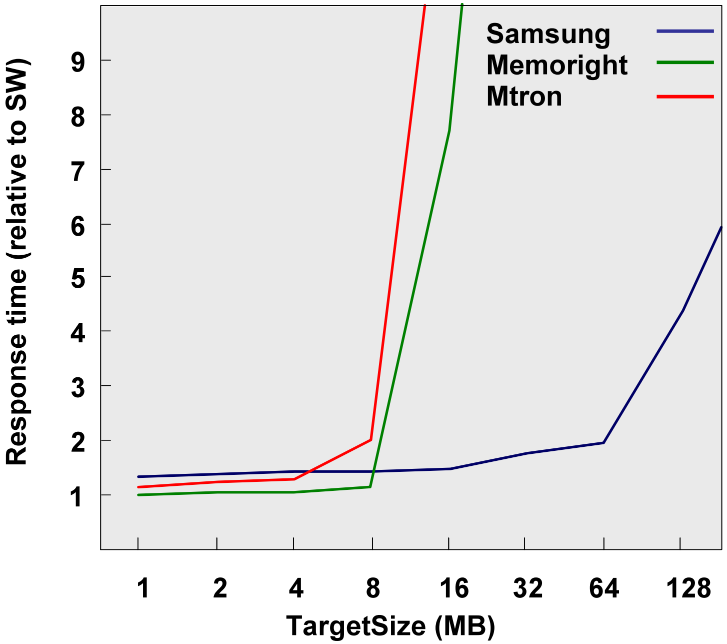
➔ Interferences: **Introduce a pause** between experiments

Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

Results (1): Samsung, memoright, Mtron



Granularity for the Memoright SSD



Locality for the Samsung, Memoright and Mtron SSDs

- For SR, SW and RR,
  - linear behavior, almost no latency
  - good throughputs with large IO Size
- For RW, ≈5ms for a 16KB-128KB IO

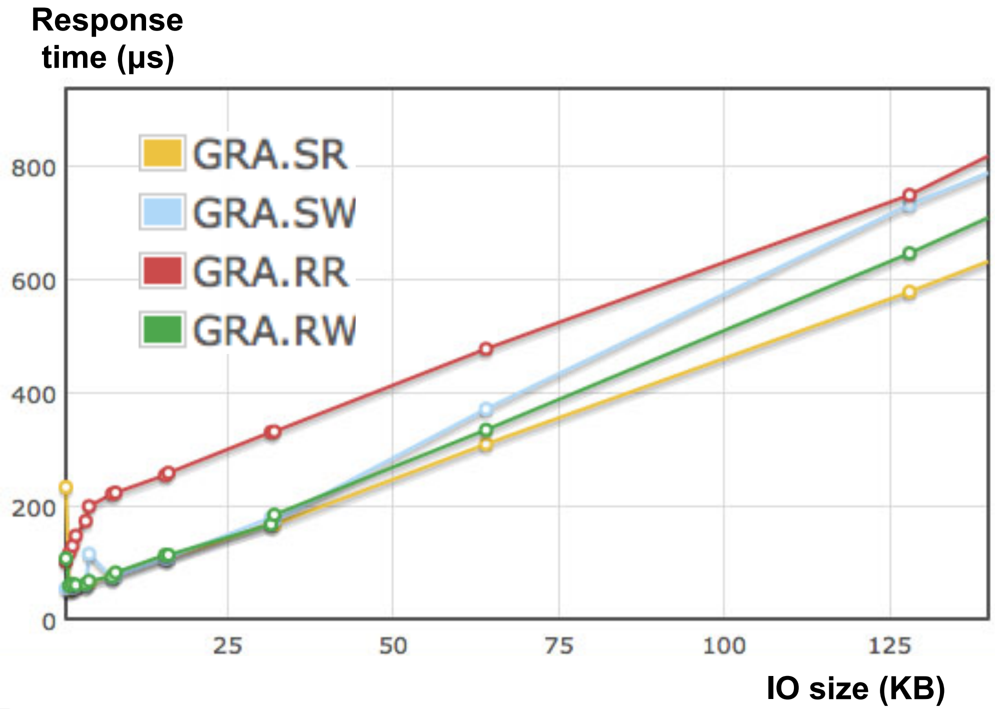
- When limited to a focused area, RW performs very well

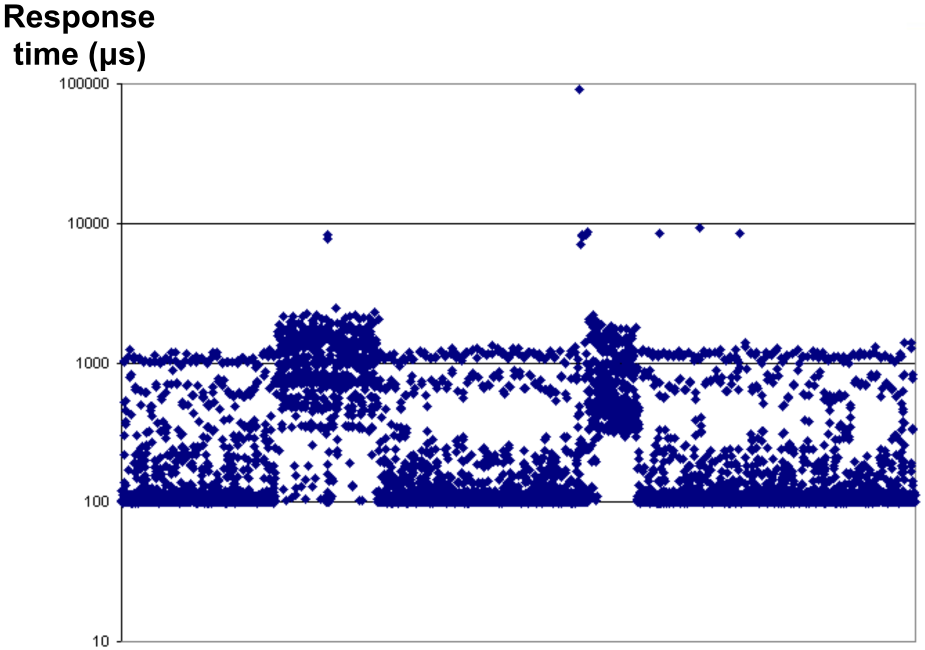
Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

Results (2): Intel X25-E

SR, SW and RW have similar performance.

RR are more costly!



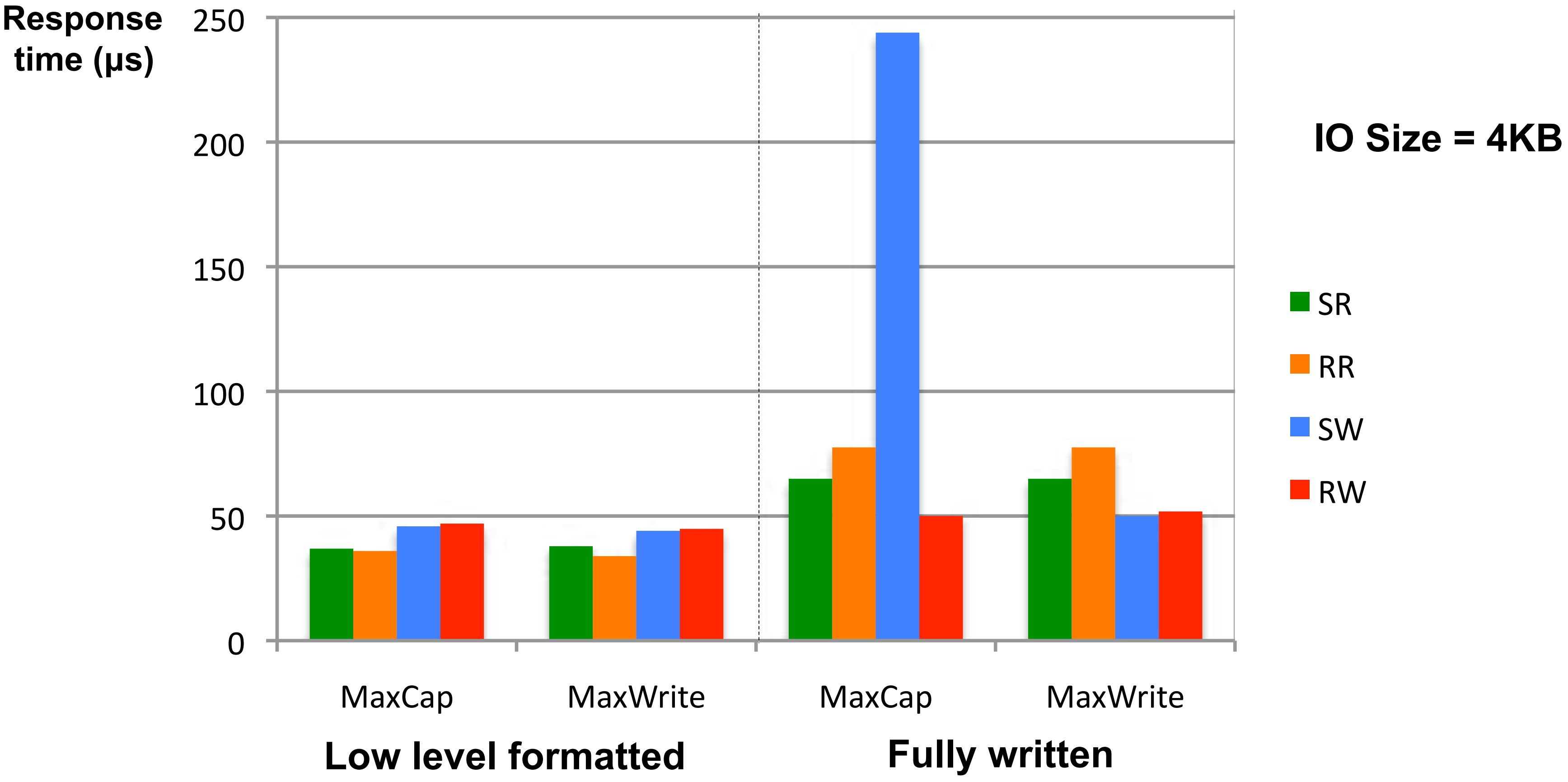


RW (16 KB) performance varies from 100 µs to 100 ms!! (x 1000)

Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

Results (3): Fusion IO

- Capacity vs Performance **tradeoff** (80 GB → 22 GB!)
- Sensitivity to device state



Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

Conclusion: Flash device behavior

Finally, what is the behavior of flash devices?

Common wisdom

- □ Update in place are inefficient?
- □ Random writes are slower than sequential ones?
- □ Better not filling the whole device if we want good performance?

⇨ **Behavior varies across devices and firmware updates**

⇨ **Behavior depends heavily on the device state!**

Is it a problem ?

Conclusion: Flash device behavior (2)

- **Flash devices are difficult (impossible?) to model!**
- **Hard to build DBMS design on such a moving ground!**

Bill Nesheim: **Mythbusting Flash Performance** ORACLE®

- *Substantial performance variability*
  - *Some cases can be even worse than disk*
- *Performance outliers can have significant adverse impact*
- *What’s Needed:*
  - *Predictable scaling & performance over time*
  - *Less asymmetry between reads/writes, random/sequential*
  - ***Predictable response time***

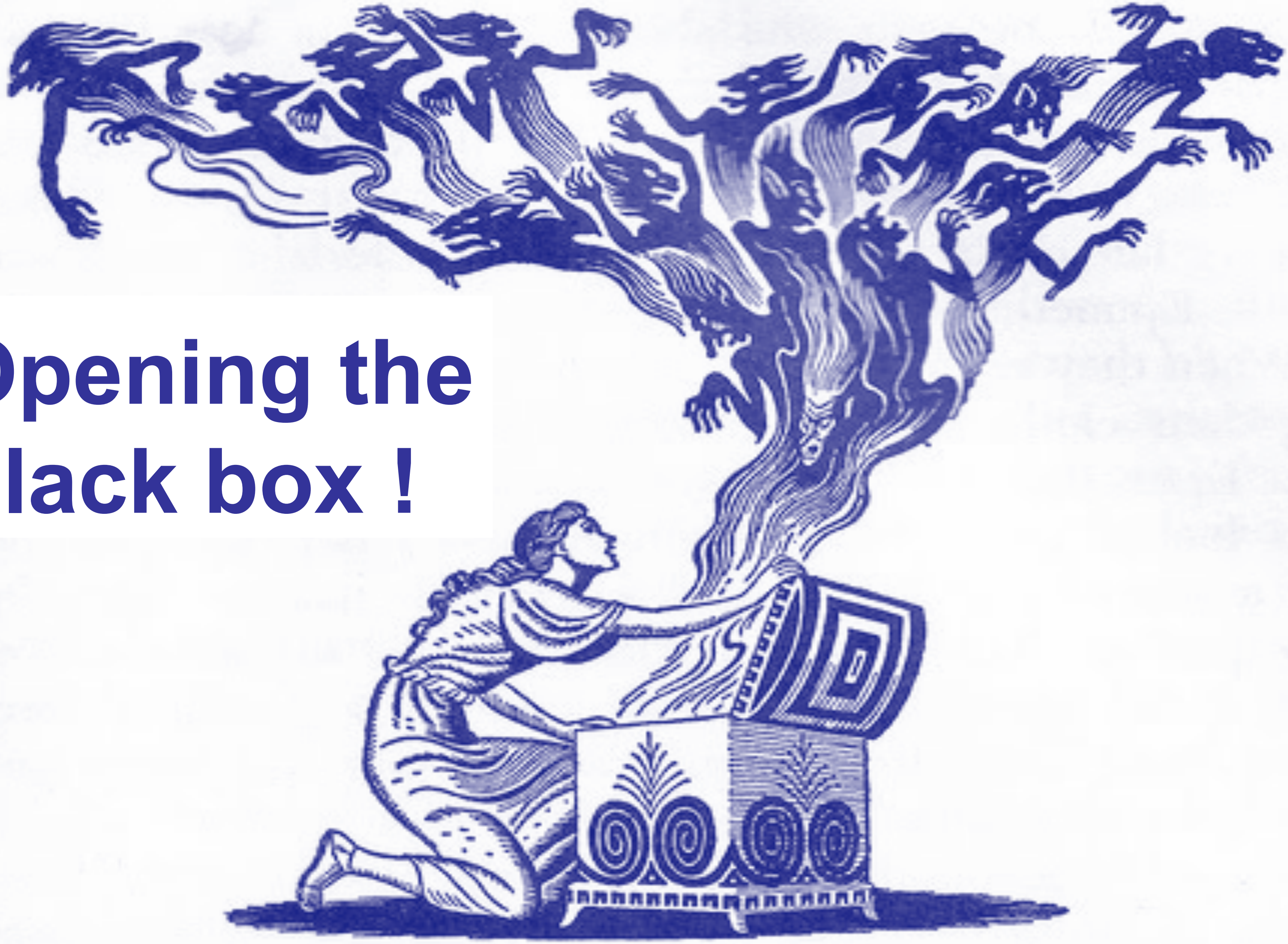
(FMS 2011)

Outline of the first part of this tutorial

Goal: understand the impact of flash memory on software (DBMS) design and vice-versa

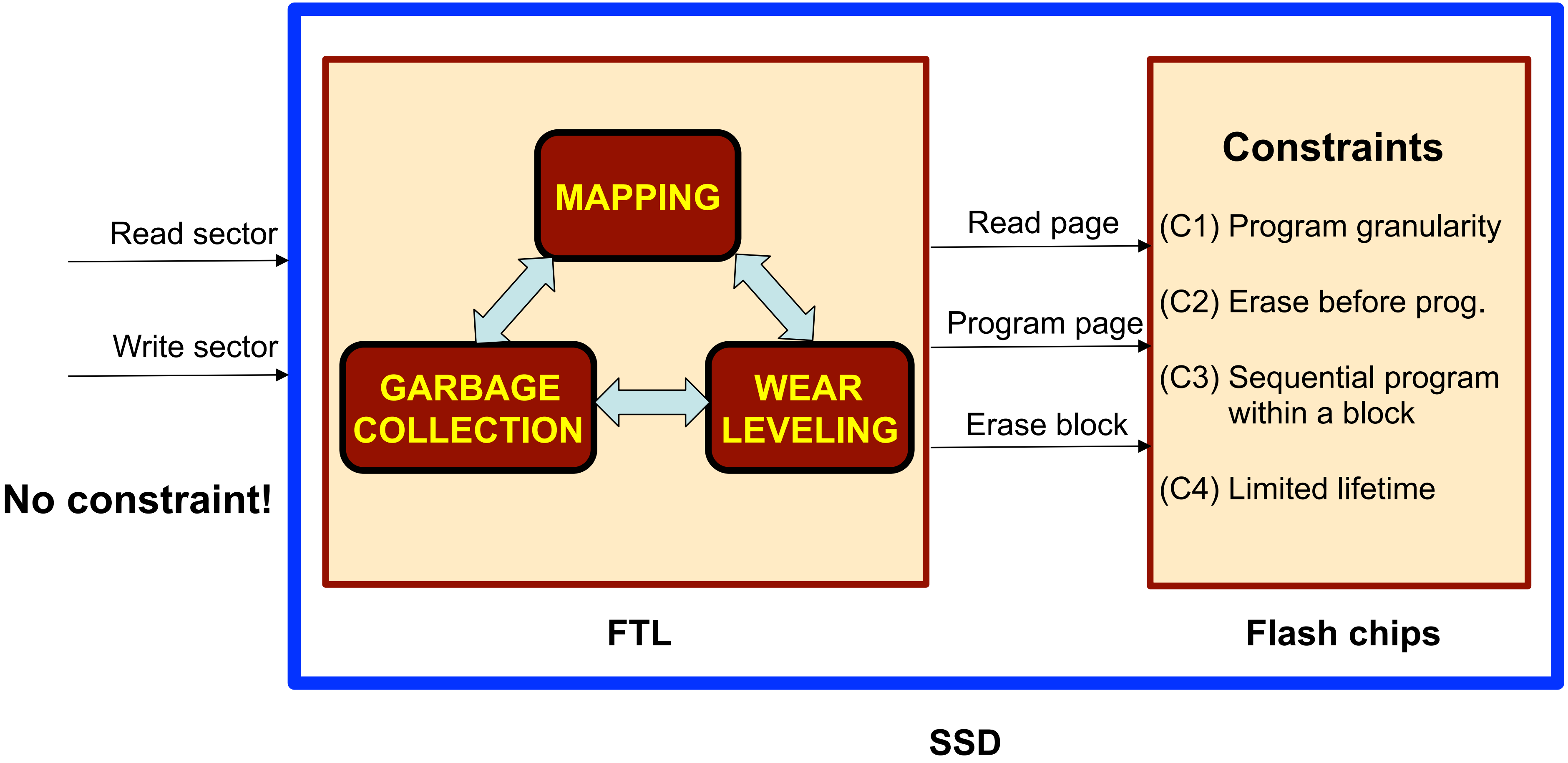
- We study flash chips, explaining their constraints and trends
- We then consider flash devices as black boxes and try to understand their performance behavior (uFLIP)
- We hit a wall with the black box approach → we open the box, i.e., the FTL, and look at FTL techniques
- Finally, we propose an alternative to complex FTLs, better adapted for DBMS design

Opening the black box !



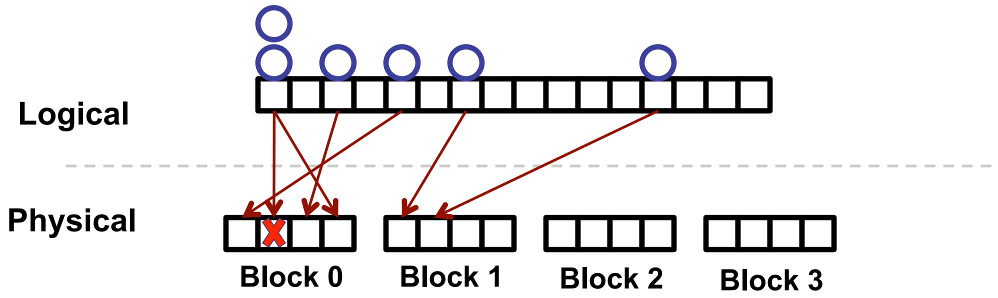
Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

FTL – Basic components



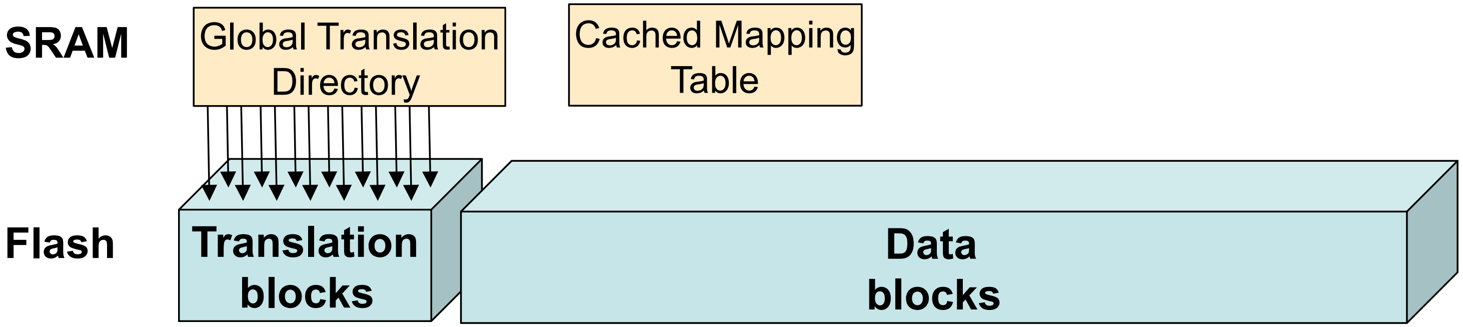
FTL – Page Level Mapping

- Basic page level mapping: translation table stored in SRAM



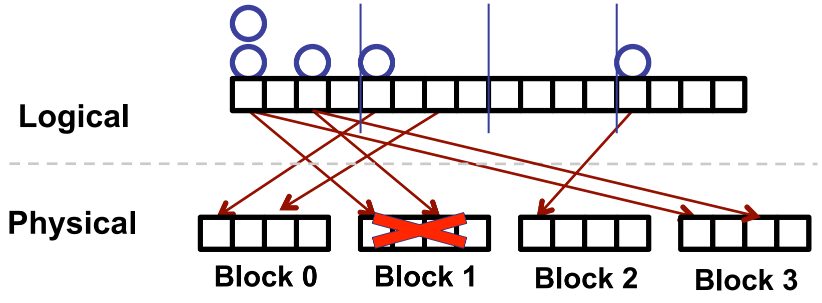
  - Problem: the table is too large ! (1 GB for 1 TB flash) (4KB pages)

- Demand-base FTL: DFTL (Gupta et al. 2009)
  - The translation table is stored in Flash and cached in SRAM



FTL - Mapping: Block Level / Hybrid

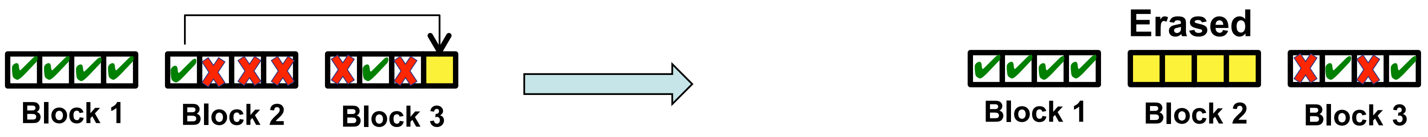
- Pure Block Level Mapping
  - Translation table at block level
  - The page offset remains the same
  - Does not comply with C3!



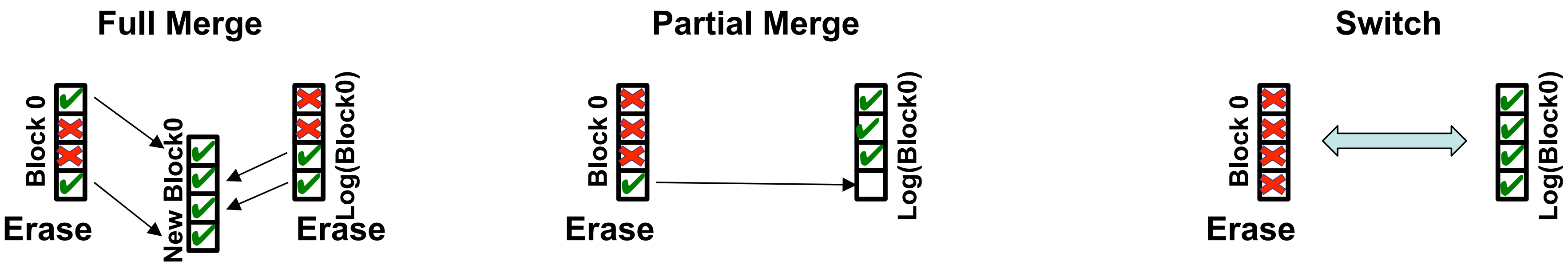
- Hybrid Mapping
  - Updates done out-of-place in log blocks
  - Data blocks → block mapping
  - Log blocks → page mapping
  - Proposals differ in the way log blocks are managed
    - 1 log block for 1 data block → BAST (Kim et al. 2002)
    - n log blocks for all data blocks → FAST (Lee et al. 2007)
    - Exploiting locality → LAST (Lee et al. 2008)
  - Cleaning when log blocks are exhausted → Major costs
  - **Block mapping for data blocks does not either comply with C3!**

FTL – Garbage Collection

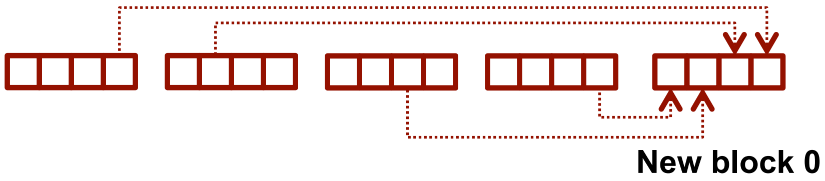
- With page mapping:



- With hybrid mapping: three cases with BAST



- More complex with FAST
  - pages of the same block can be on different log blocks

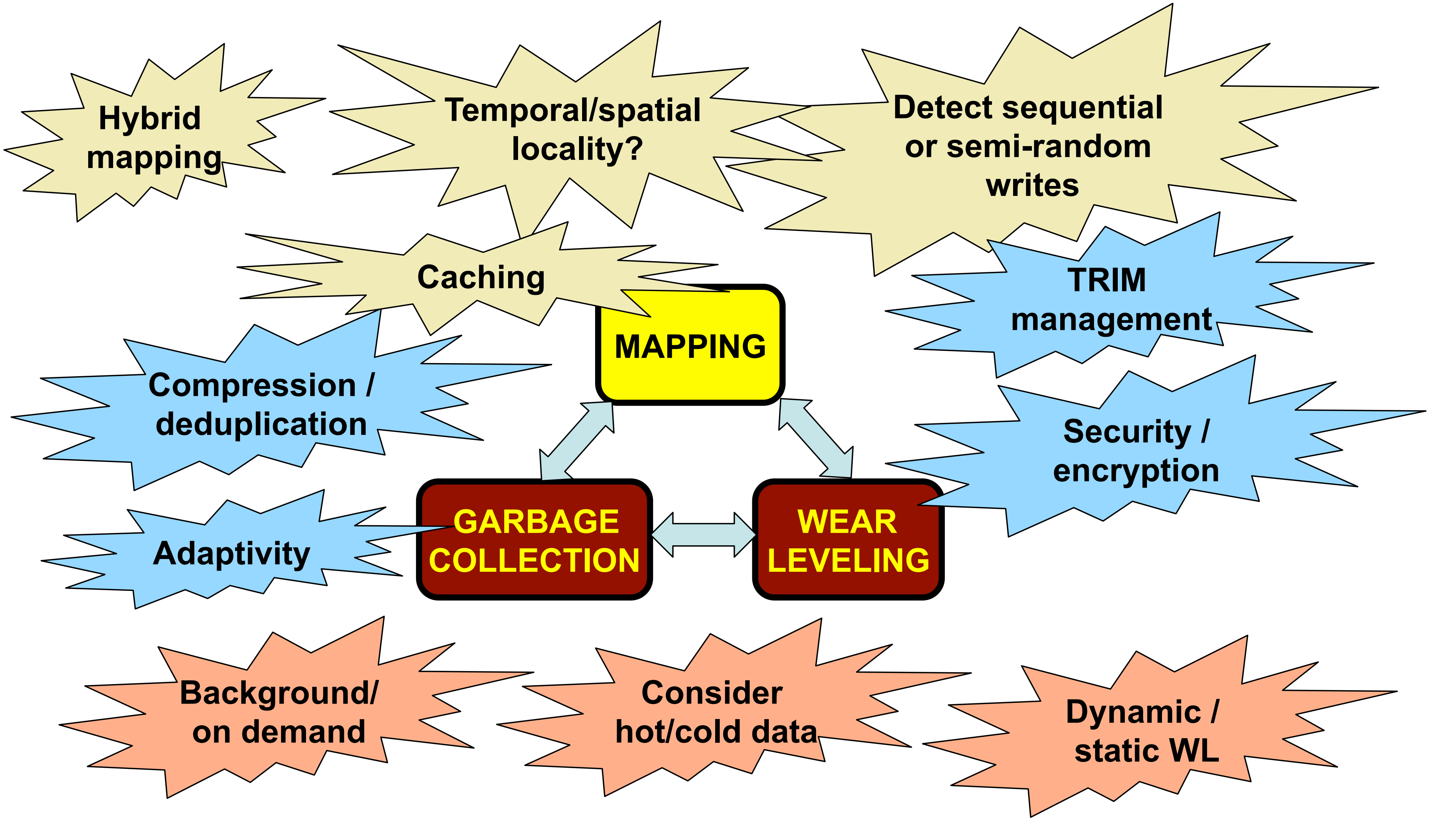


Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

FTL-Wear leveling

- **Goal:** ensure that all blocks of the flash have about the same erase count (i.e., number of program/erase cycle).
- **Basic algorithm: hot-cold swapping** (Jung et al. 2007)
  - Swap the blocks with min and max erase count.
- **Difficulties:**
  (1) When to trigger the WL algorithm
  (2) How to manage erase count, how to select min or max erase count block wrt the limited CPU and memory resources of the flash controler
  (3) What wear leveling strategy?
  (4) Interactions between Garbage Collection and Wear Leveling
- **The same difficulties arise with garbage collection!**

FTL: Trends



Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

FTL designers vs DBMS designers goals

- **Flash device designers goals:**
  - Hide the flash device constraints (usability)
  - Improve the performance for most common workloads
  - Make the device auto-adaptive
  - Mask design decision to protect their advantage (black box approach)
- **DBMS designers goals:**
  - Have a model for IO performance (and behavior)
    - Predictable
    - Clear distinction between efficient and inefficient IO patterns
  - ⇨ To design the storage model and query processing/optimization strategies
  - Reach best performance, even at the price of higher complexity (having a full control on actual IOs)

These goals are conflicting!

Outline of the first part of this tutorial

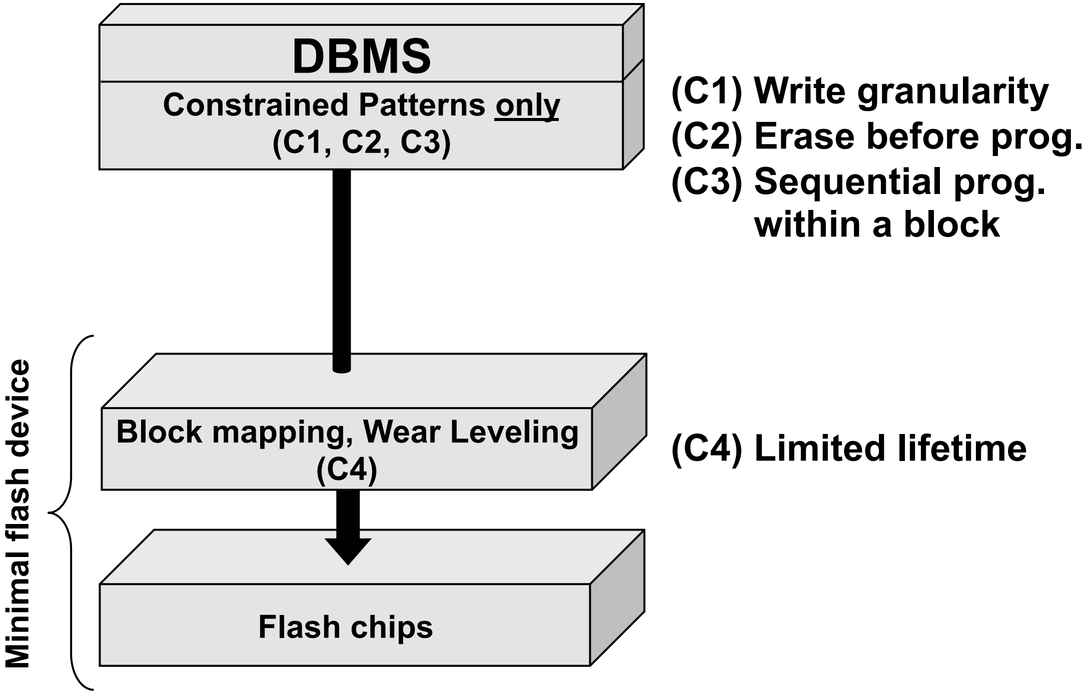
Goal: understand the impact of flash memory on software (DBMS) design and vice-versa

- We study flash chips, explaining their constraints and trends
- We then consider flash devices as black boxes and try to understand their performance behavior (uFLIP)
- We hit a wall with the black box approach → we open the box, i.e., the FTL, and look at FTL techniques
- Finally, we propose an alternative to complex FTLs, better adapted for DBMS design

Minimal FTL: Take the FTL out of equation!

FTL provides only wear leveling, using block mapping to address C4 (limited lifetime)

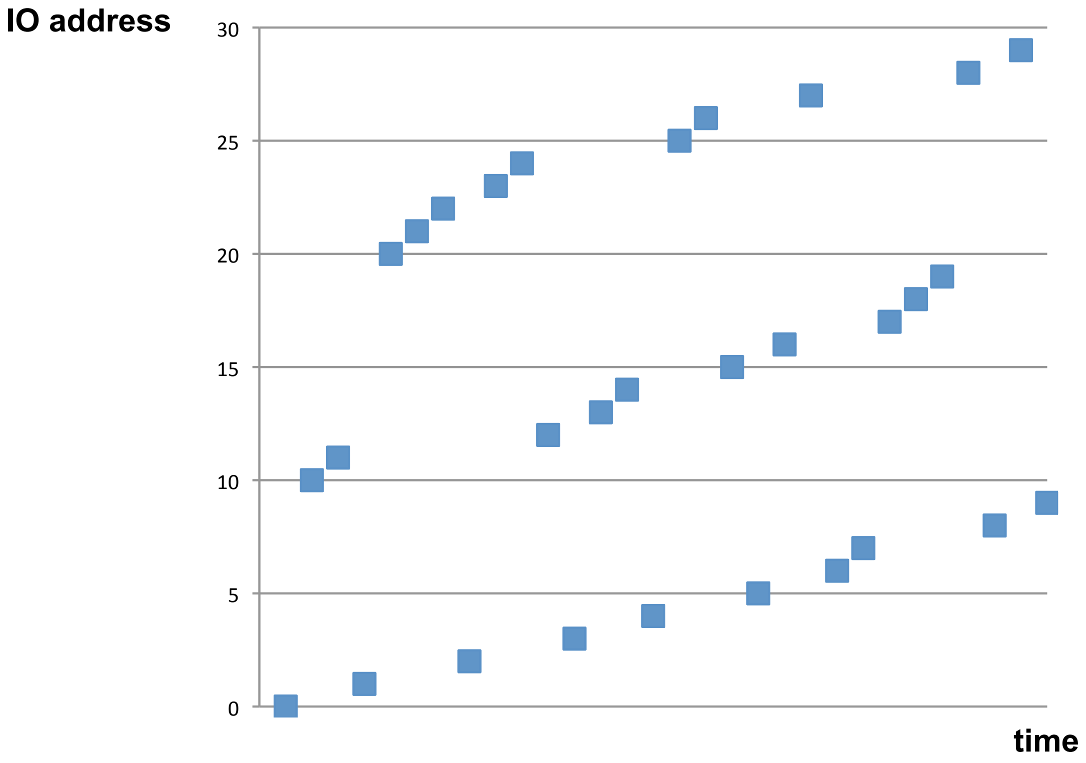
- **Pros**
  - Maximal performance for
    - SR, RR, SW
    - Semi-Random Writes
  - Maximal control for the DBMS
- **Cons**
  - All complexity is handled by the DBMS
  - All IOs must follow C1-C3
    - The whole DBMS must be rewritten
    - The flash device is dedicated



Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

Semi-random writes (uFLIP [CIDR09])

- Inter-blocks : Random
- Intra-block : Sequential
- Example with 3 blocks of 10 pages:



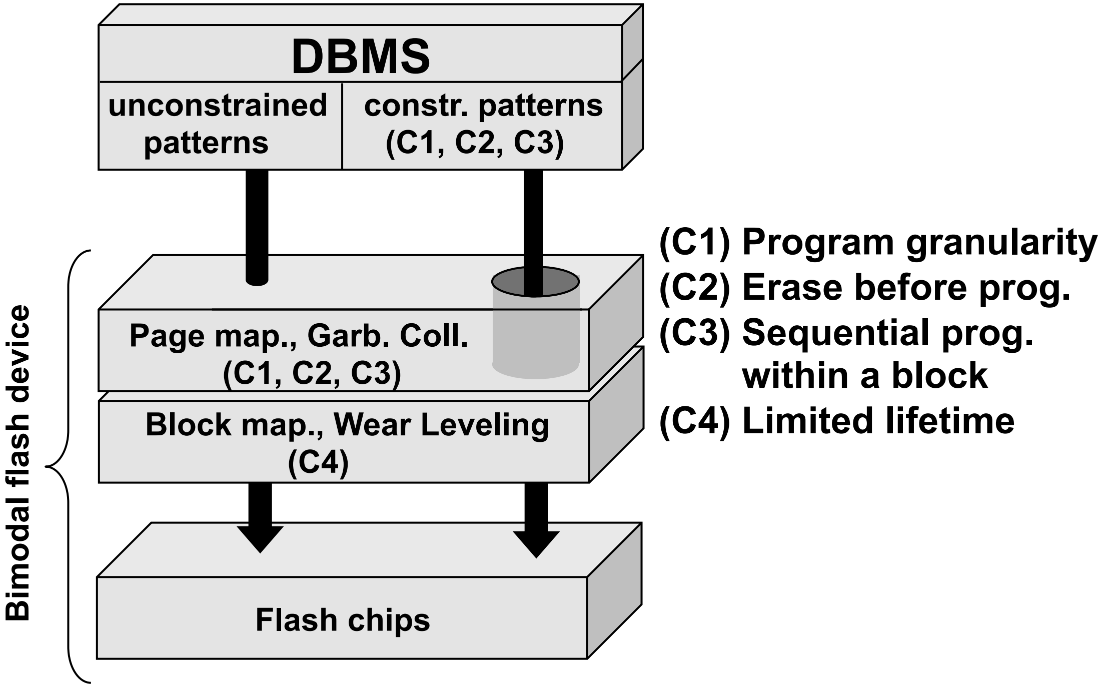
0	10	11	1	20	21	22	2	23	24	12	3	13	14	4	25	26	15	5	16	27	6	7	17	18	19	28	8	29	9

Bimodal FTL: a simple idea …

- **Bimodal Flash Devices**:
  - Provide a ***tunnel*** for those IOs that respect constraints C1-C3 ensuring maximal performance
  - Manage other **unconstrained** IOs in best effort
  - Minimize **interferences** between these two modes of operation

- **Pros**
  - Flexible
  - Maximal performance and control for the DBMS for constrained IOs

- **Cons**
  - No behavior guarantees for unconstrained IOs.

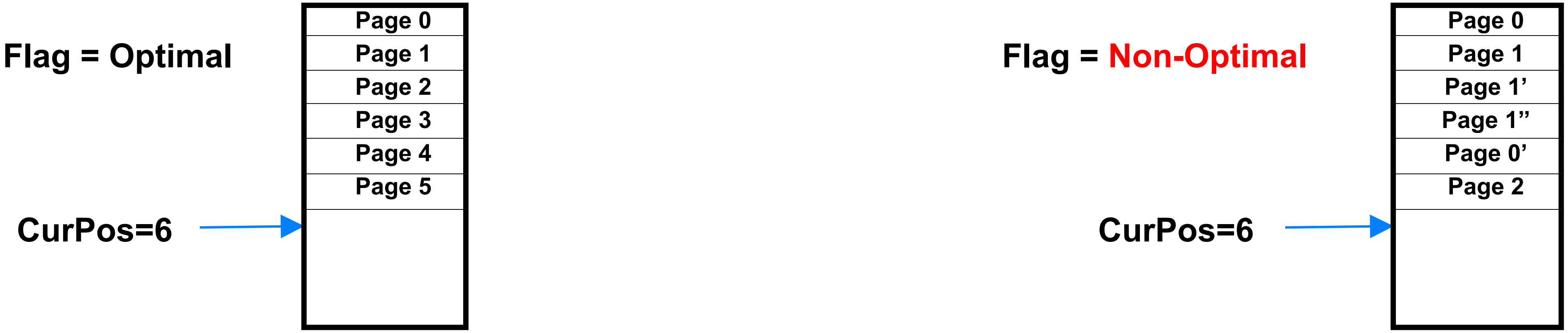

DBMS
unconstrained patterns
constr. patterns (C1, C2, C3)
Bimodal flash device
Page map., Garb. Coll. (C1, C2, C3)
Block map., Wear Leveling (C4)
Flash chips

(C1) Program granularity
(C2) Erase before prog.
(C3) Sequential prog. within a block
(C4) Limited lifetime

Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

Bimodal FTL: easy to implement

- Constrained IOs lead to **optimal blocks**



Flag = Optimal

Page 0
Page 1
Page 2
Page 3
Page 4
Page 5

CurPos=6

Flag = Non-Optimal

Page 0
Page 1
Page 1’
Page 1”
Page 0’
Page 2

CurPos=6

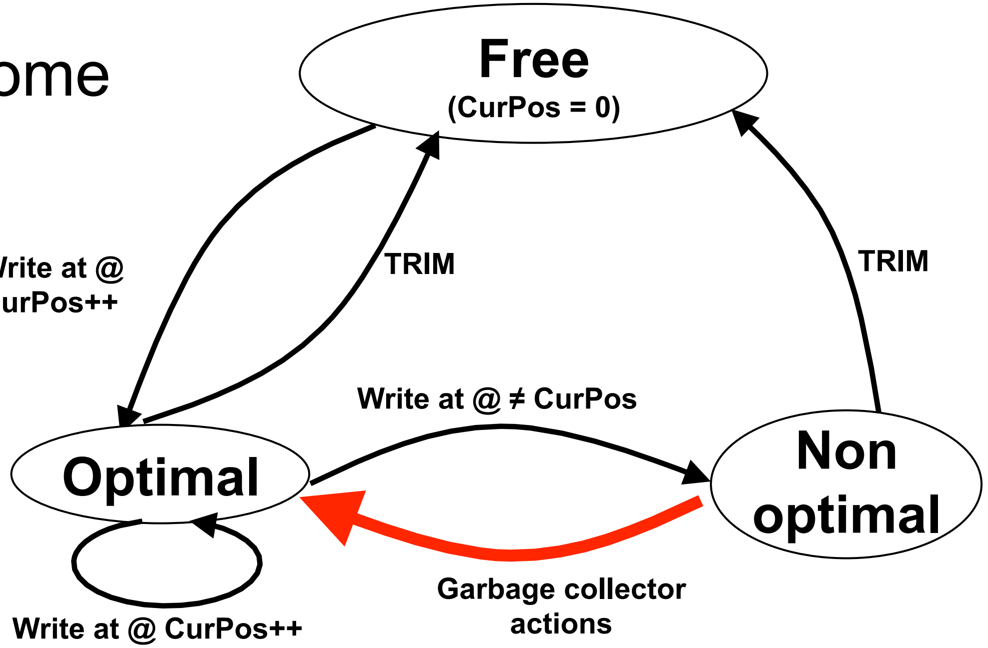
- Optimal blocks can be trivially
  - mapped using a small map table in safe cache
  - detected using a flag and cursor in safe cache

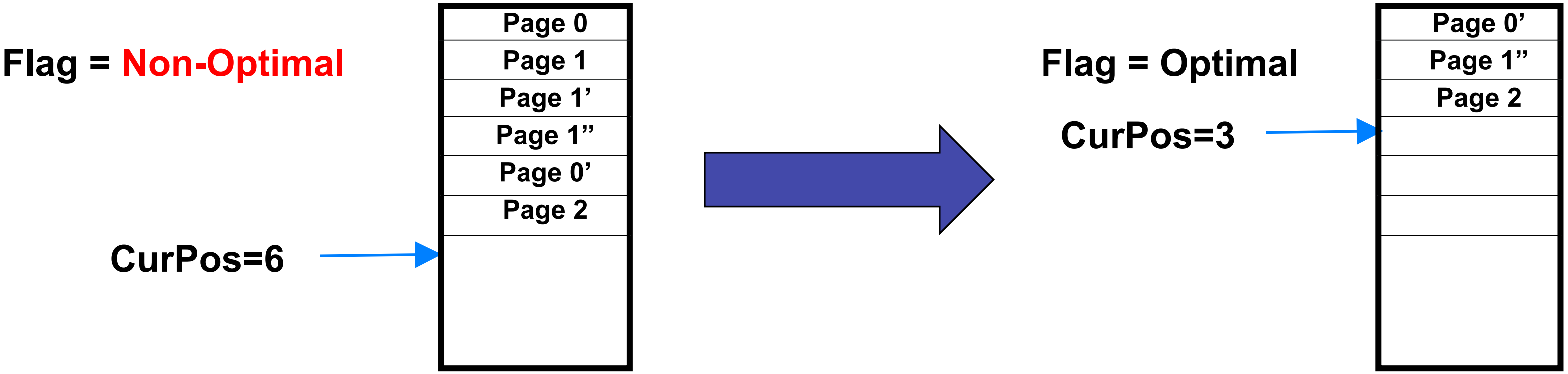
16 MB for a 1TB device

- **No interferences**!
- **No change to the block device interface**:
  - Need to expose two constants: block size and page size

Bimodal FTL: better than Minimal + FTL

- Non-optimal block can become optimal (thanks to GC)





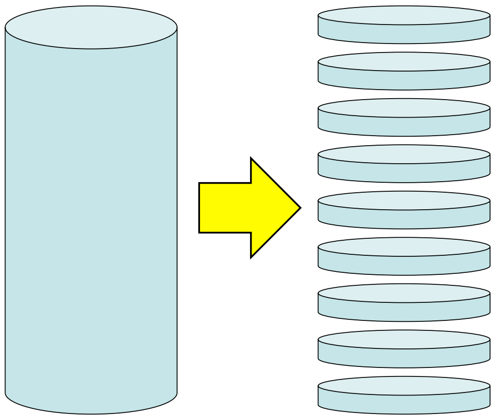
Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

Impact on DBMS Design

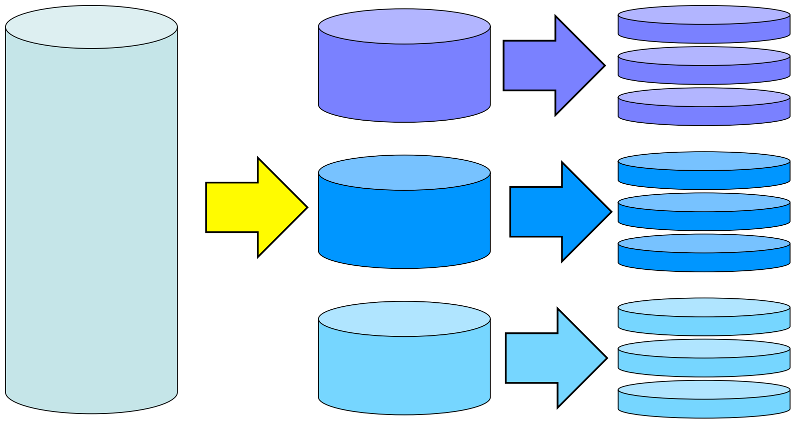
Using bimodal flash devices, we have a solid basis for designing efficient DBMS on flash:

- What IOs should be constrained?
  - i.e., what part of the DBMS should be redesigned?
- How to enforce these constraints? Revisit literature:
  - Solutions based on flash chip behavior enforce C1-C3 constraints
  - Solutions based on existing classes of devices might not.

Example: Hash Join on HDD



One pass partitioning



Multi-pass partitioning (2 passes)

Tradeoff: IOSize vs Memory consumption

- **IOSize should be as large as possible, e.g., 256KB – 1 MB**
  - To minimize IO cost when writing or reading partitions
- **IOSize should be as small as possible**
  - To minimize memory consumption: One pass partitioning needs 2 x IOSize x NbPartitions in RAM
  - Insufficient memory → multi-pass → performance degrades!

Hash join on SSD and on bimodal SSD

- **With non bimodal SSDs**
  - No behavior guarantees but…
  - Choosing IOSize = Block size (256 KB – 1MB) should bring good performance

- **With bimodal SSDs**
  - Maximal performance are guaranteed (constrained patterns)
  - Use semi-random writes
  - IOSize can be reduced up to page size (4 – 16 KB) with no penalty
  - → Memory savings
  - → Performance improvement
  - → Predictability!

Summary

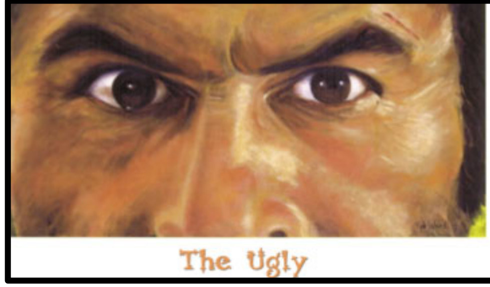
The Good

- Flash chips
  - Performance & Energy consumption
  - Wired in parallel in flash devices

The Bad

- Hardware constraints!

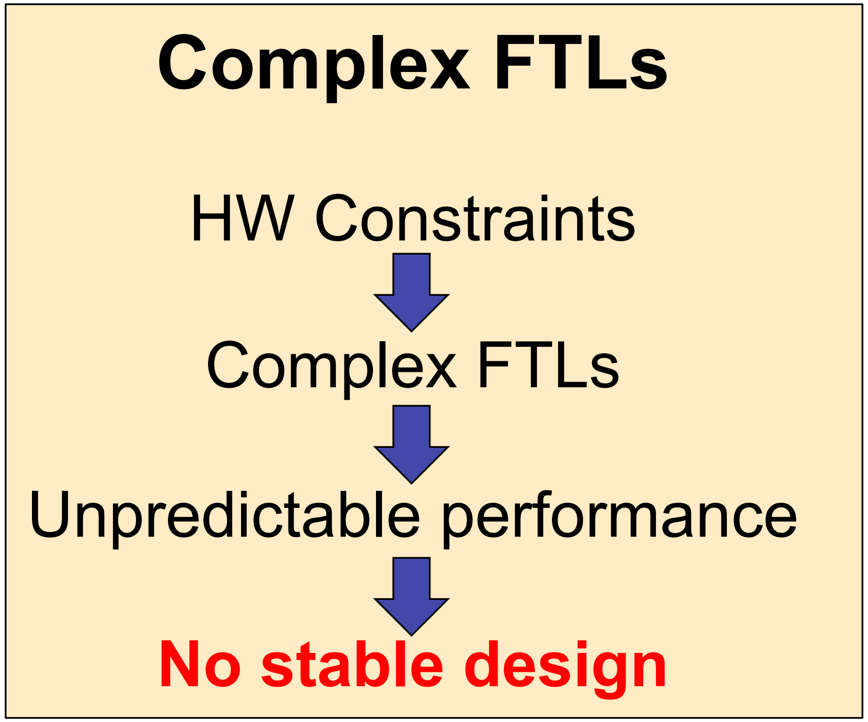
  (C1) Program granularity, (C2) Erase before program,
  (C3) Sequential program within a block,
  (C4) Limited lifetime



The Ugly

- FTL: a complex piece of sofware
  - Constantly evolving, no common behavior
  - Hard to model
  - Hard to build a consistent DBMS design!

Conclusion: DBMS Design ?



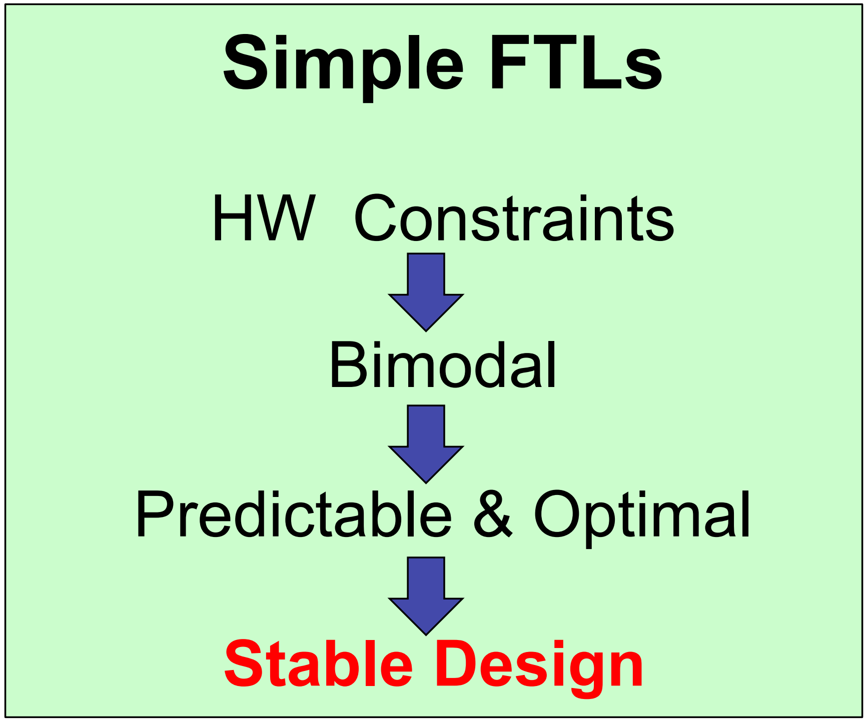
Complex FTLs

HW Constraints

Complex FTLs

Unpredictable performance

No stable design



Simple FTLs

HW Constraints

Bimodal

Predictable & Optimal

Stable Design

- **Adding bimodality does not hinder competition between flash device manufacturers, they can**
  - bring down the cost of constrained IO patterns (e.g., using parallelism)
  - bring down the cost of unconstrained IO patterns without jeopardizing DBMS design

Tutorial Outline

1. Introduction (Philippe)
2. Flash devices characteristics (Luc)
3. Data management for flash devices (Stratis)
4. Two outlooks (Stratis & Philippe)

Flash: a disruptive technology

- ✓ Orders of magnitude better performance than HDD
- ✓ Low power consumption
- ✓ Dropping prices
- Idea: Throw away HDDs and replace everything with Flash SSDs
  - Not enough capacity
  - Not enough money to buy the not-enough-capacity
- However, Flash fits very well between DRAM and HDD
  - DRAM/Flash/HDD price ratio: ~100:10:1 per GB
  - DRAM/Flash/HDD latency ratio: ~1:10:100
- Integrate Flash into the storage hierarchy
  - complement existing DRAM memory and HDDs

Outline

- Flash-based device design
  - Solid state drives
  - Making SSDs database-friendly
- System-level challenges
  - Hybrid systems
  - Storage, buffering and caching
  - Indexing on flash
  - Query and transaction processing

Outline

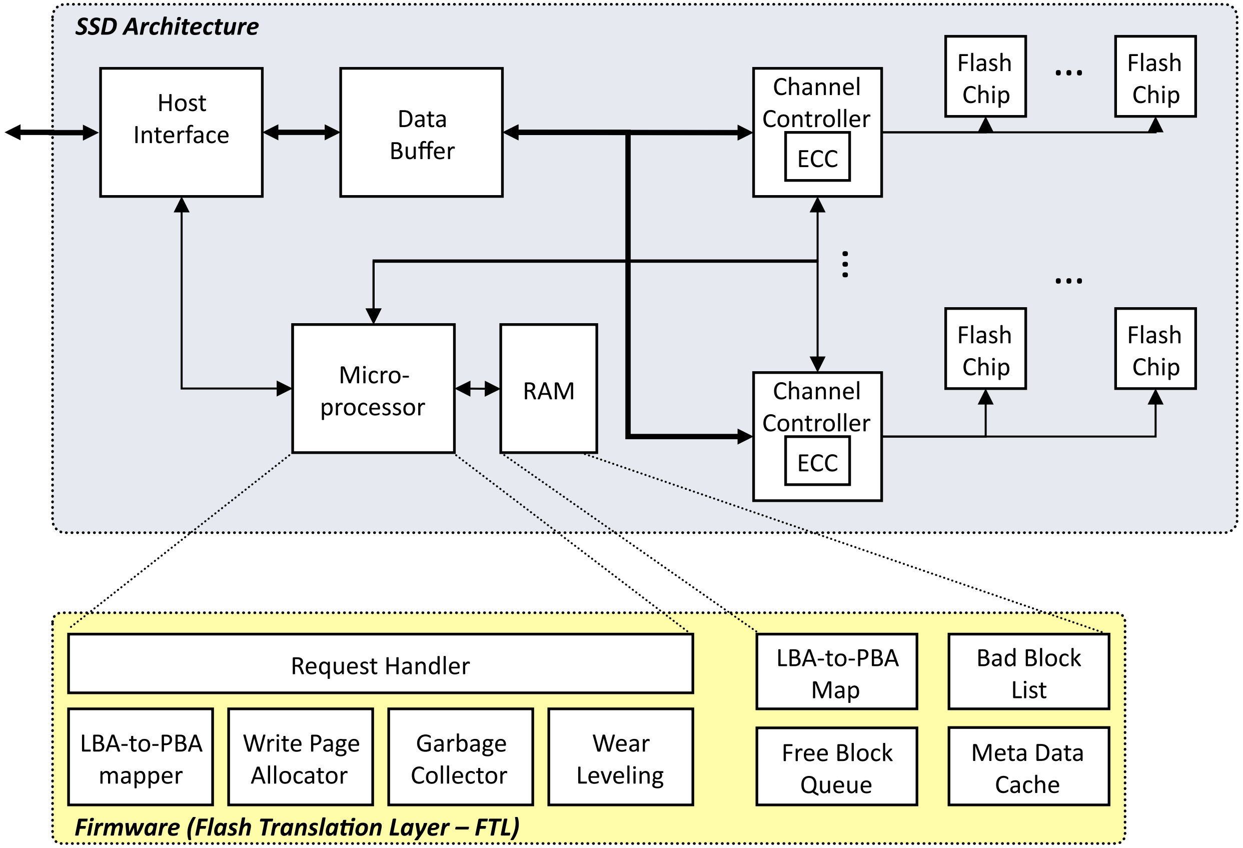
- Flash-based device design
  - Solid state drives
  - Making SSDs database-friendly
- System-level challenges
  - Hybrid systems
  - Storage, buffering and caching
  - Indexing on flash
  - Query and transaction processing

Flash-based Solid State Drives

- Common I/O interface
  - Block-addressable interface
- No mechanical latency
  - Access latency independent of the access pattern
  - 30 to 50 times more efficient in IOPS/$ per GB than HDDs
- Read / Write asymmetry
  - Reads are faster than writes
  - Erase-before-write limitation
- Limited endurance / wear leveling
  - 5 year warranty for enterprise SSDs (assuming 10 complete re-writes per day)
- Energy efficiency
  - 100 – 200 times more efficient than HDDs in IOPS / Watt
- Physical properties
  - Resistance to extreme shock, vibration, temperature, altitude
  - Near-instant start-up time

SSD architecture

- Various form factors
  - PCI-e
  - SAS (1.8” – 3.5”)
  - SATA (1.8” – 3.5”)
- Number of channels
  - 4 to 16 or more
- RAM buffers
  - 1MB up to more than 256MB
- Over-provisioning
  - 10% up to 40%
- Command Parallelism
  - Intra-command
  - Inter-command
  - Reordering / merging (NCQ)



SSD Architecture

Host Interface

Data Buffer

Channel Controller

ECC

Flash Chip

…

Flash Chip

Micro-processor

RAM

Channel Controller

ECC

Flash Chip

Flash Chip

Request Handler

LBA-to-PBA mapper

Write Page Allocator

Garbage Collector

Wear Leveling

LBA-to-PBA Map

Bad Block List

Free Block Queue

Meta Data Cache

Firmware (Flash Translation Layer – FTL)

Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

15k RPM SAS HDD: ~**250-300** IOPS
7.2k RPM SATA HDD: ~**80** IOPS

Off-the-shelf SSDs

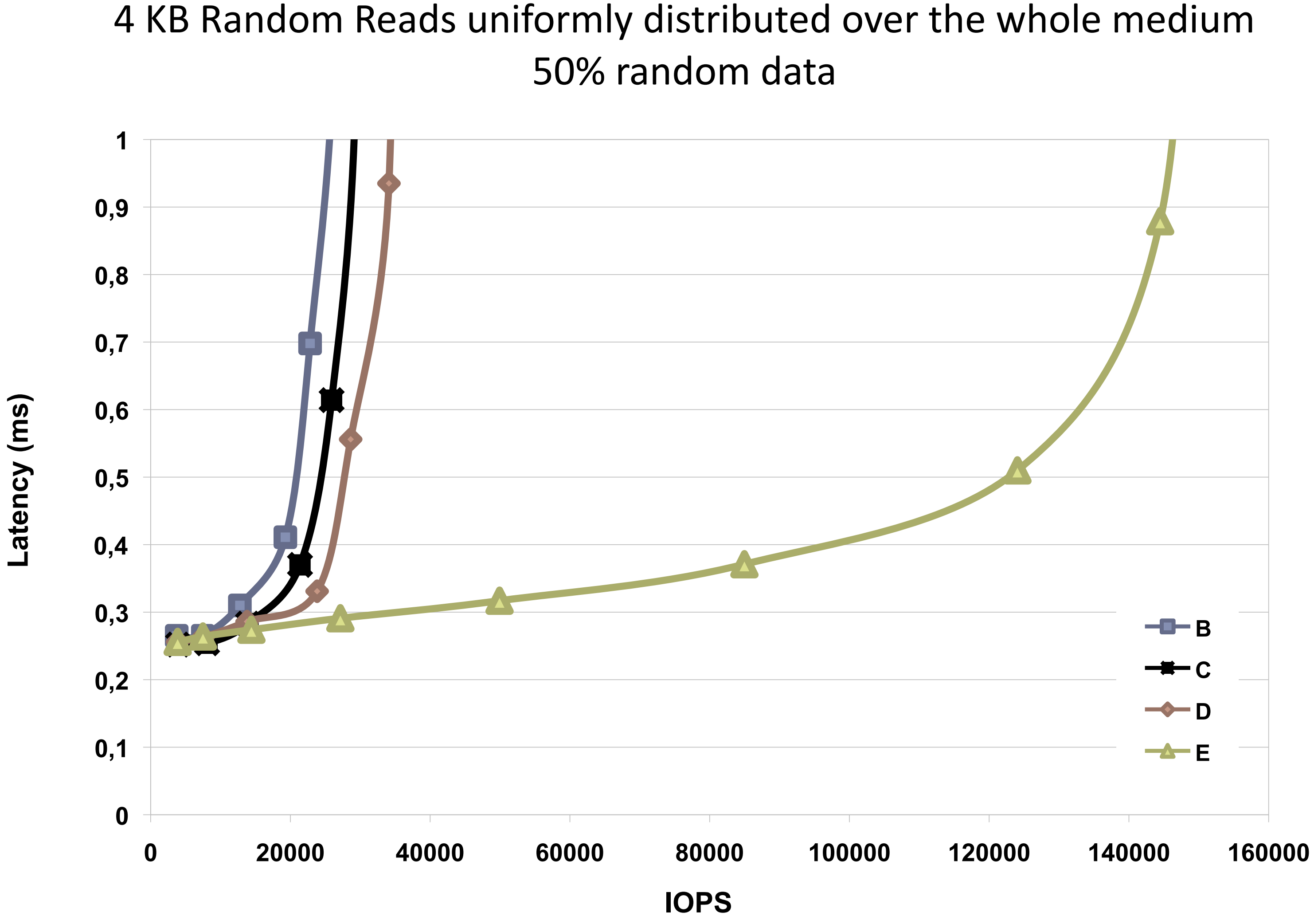
	A	B	C	D	E
Form Factor	PATA Drive Consumer	SATA Drive Consumer	SATA Drive Consumer	SAS Drive Enterprise	PCI-e card Enterprise
Flash Chips	MLC	MLC	MLC	SLC	SLC
Capacity	32 GB	100 GB	160GB	140GB	450 GB
Read Bandwidth	53 MB/s	285 MB/s	250 MB/s	220 MB/s	700 MB/s
Write Bandwidth	28 MB/s	250 MB/s	100 MB/s	115 MB/s	500 MB/s
Random 4kB Read IOPS	3.5k	30k	35k	45k	140k
Random 4kB Write IOPS	0.01k	10k	0.6k	16k	70k
Street Price:	~ 15 $/GB (2007)	~ 4 $/GB (2010)	~ 2.5 $/GB (2010)	~18 $/GB (2011)	~ 38 $/GB (2009)

~ 1 order of magnitude

> 2 orders of magnitude

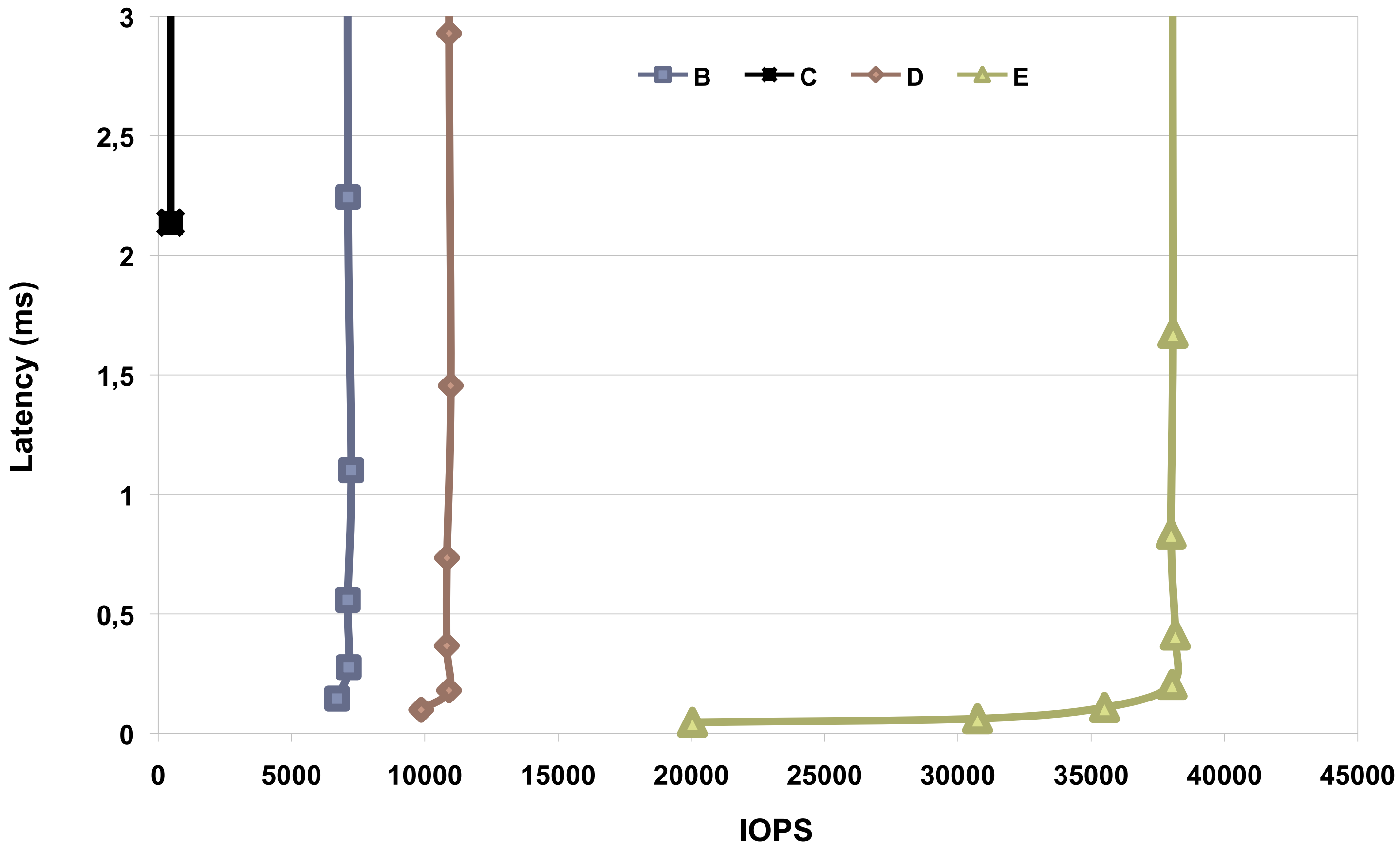
Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

Read latency



Write latency

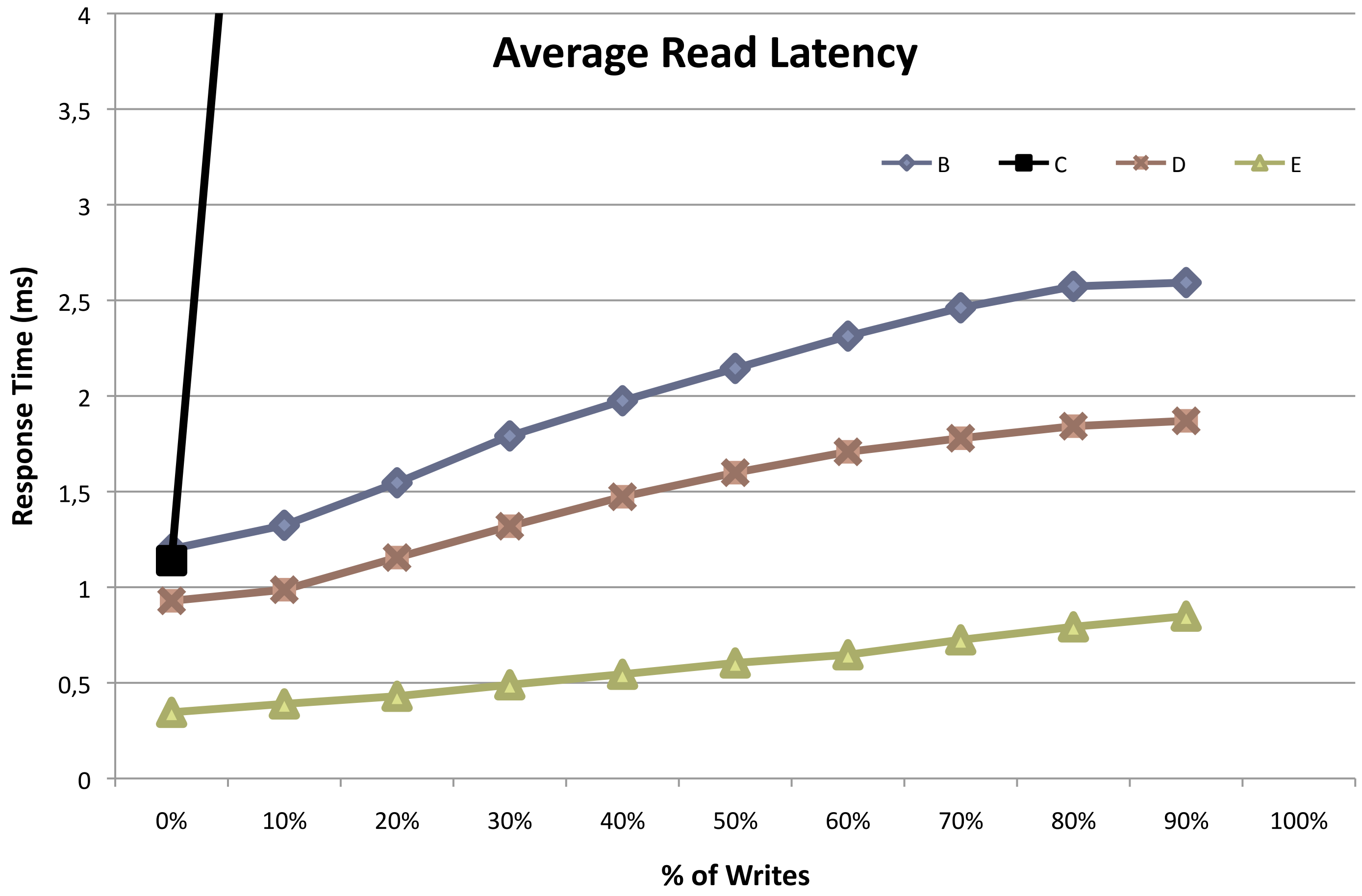
4 KB Random Writes uniformly distributed over the whole medium
50% random data



Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

Mixed workload – Read latency

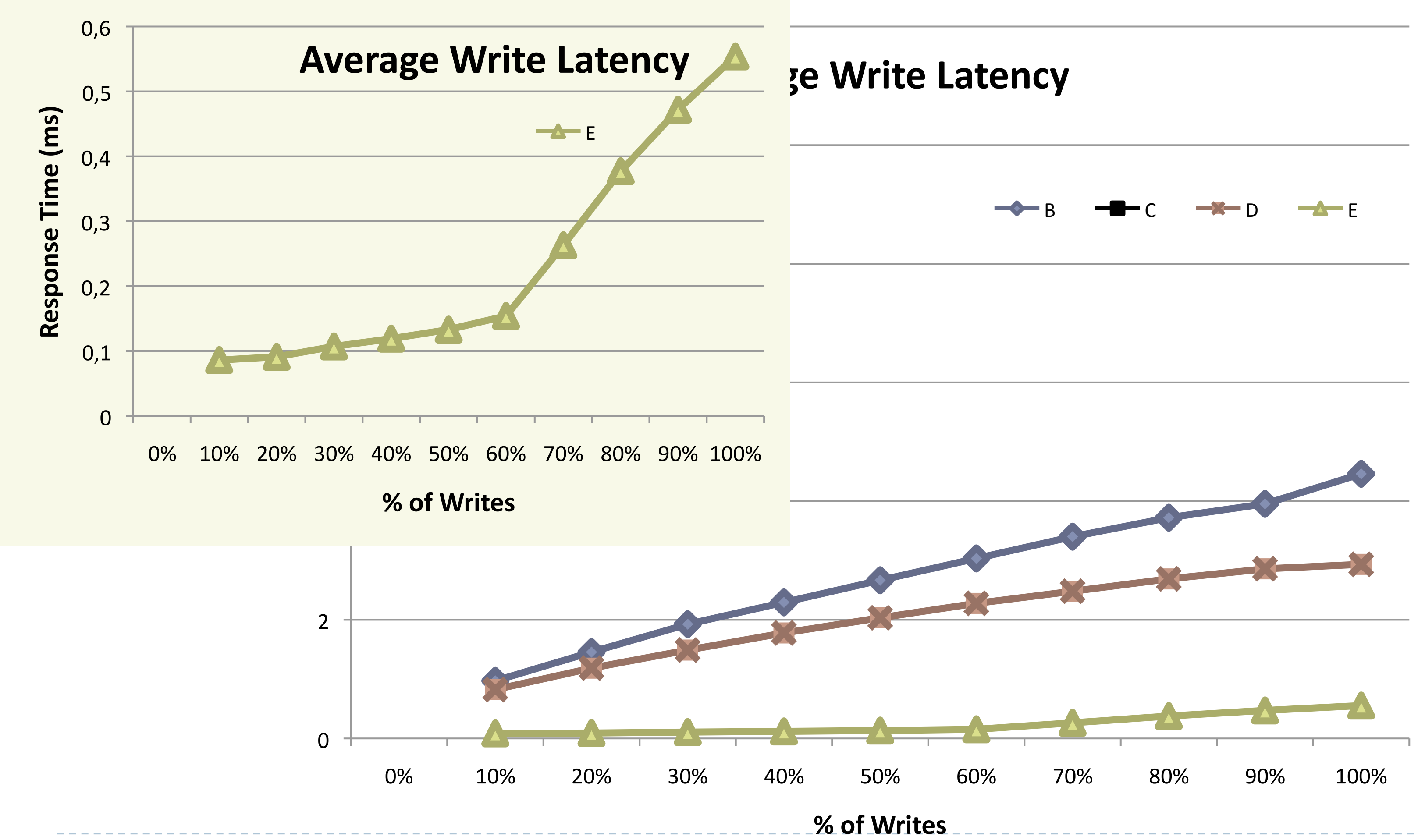
4 KB I/O operations uniformly distributed over the whole medium
50% random data, Queue depth = 32



Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

Mixed workload – Write latency

4 KB I/O operations uniformly distributed over the whole medium
50% random data, Queue depth = 32



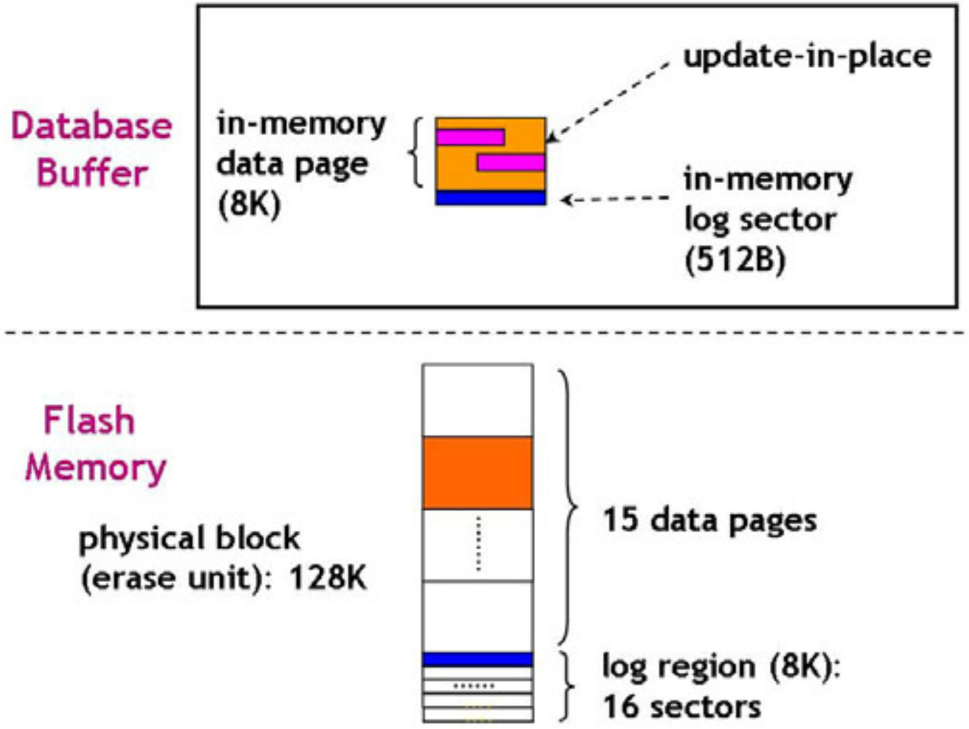
Outline

- Flash-based device design
  - Solid state drives
  - Making SSDs database-friendly
- System-level challenges
  - Hybrid systems
  - Storage, buffering and caching
  - Indexing on flash
  - Query and transaction processing

[Lee & Moon, SIGMOD 2007]

In-page logging

- Page updates are logged
- Log sector for each DB page
  - Allocated when page become dirty
- Log region in each flash block
- Page write-backs only involve log-sector writes
  - Until a merge is required
- Upon read:
  - Fetch log records from flash
  - Apply them to the in-memory page
- Same or more number of writes
  - **But**, significant reduction of erasures

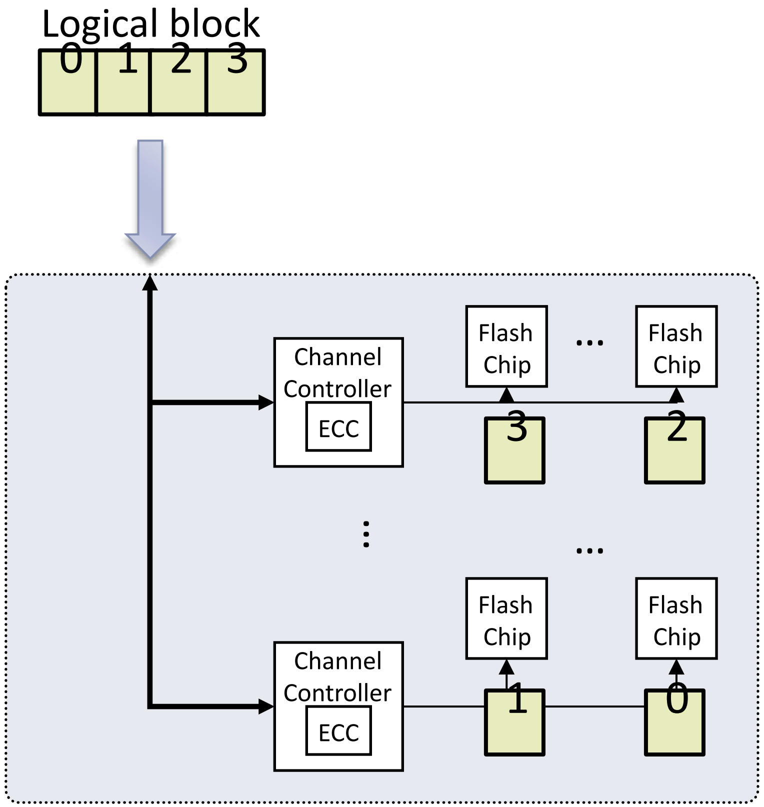

Database Buffer
in-memory data page (8K)
update-in-place
in-memory log sector (512B)
Flash Memory
physical block (erase unit): 128K
15 data pages
log region (8K): 16 sectors

However:

- The DBMS needs to control physical placement
- Partial flash page writes are involved

Parallelism?

- Still, some operations are more efficient on hardware
  - Mapping of the address space to flash planes, dies and channels
  - ECC, encryption etc.
  - Wear-leveling still needs to be done by the device firmware
- The internal device geometry is critical to achieve maximum parallelism
- The DBMS needs to be aware of the geometry to some degree



Logical block size ≠ Flash block size

Logical block size relevant to number of channels, pipelining capabilities on each channel, etc.

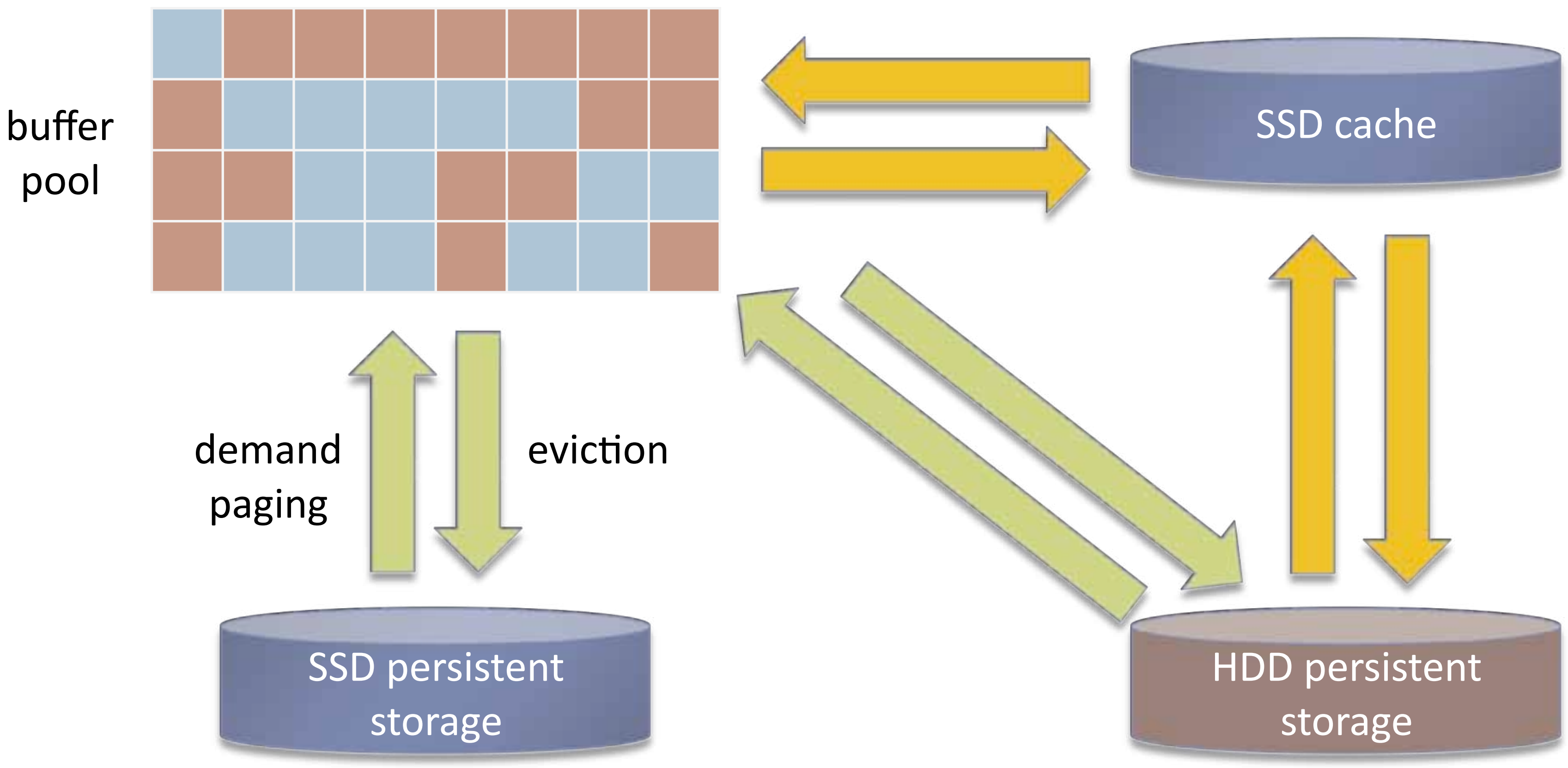
SSDs – summary

- Flash memory has the potential to remove the I/O bottleneck
  - Especially for read-dominated workloads
- "SSD": multiple classes of devices
- Excellent random read latency is universal
- Read and write bandwidth varies widely
- Dramatic difference across random write latencies
- Dramatic differences in terms of economics: $/GB cost, power consumption, expected lifetime, reliability
- A lot of research to be done towards defining DBMS-specific interfaces

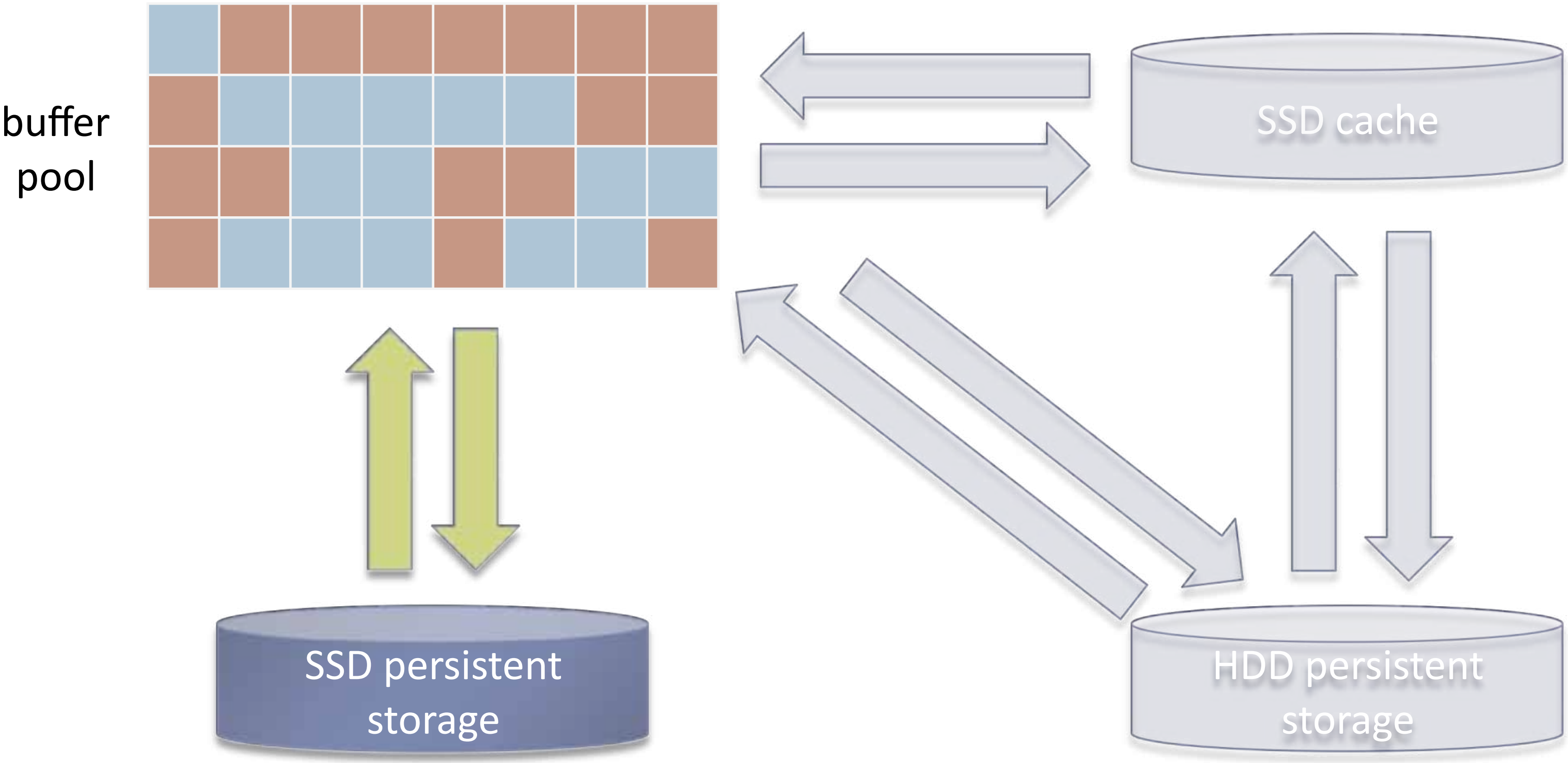
Outline

- Flash-based device design
  - Solid state drives
  - Making SSDs database-friendly
- System-level challenges
  - Hybrid systems
  - Storage, buffering and caching
  - Indexing on flash
  - Query and transaction processing

Storage, buffering and caching

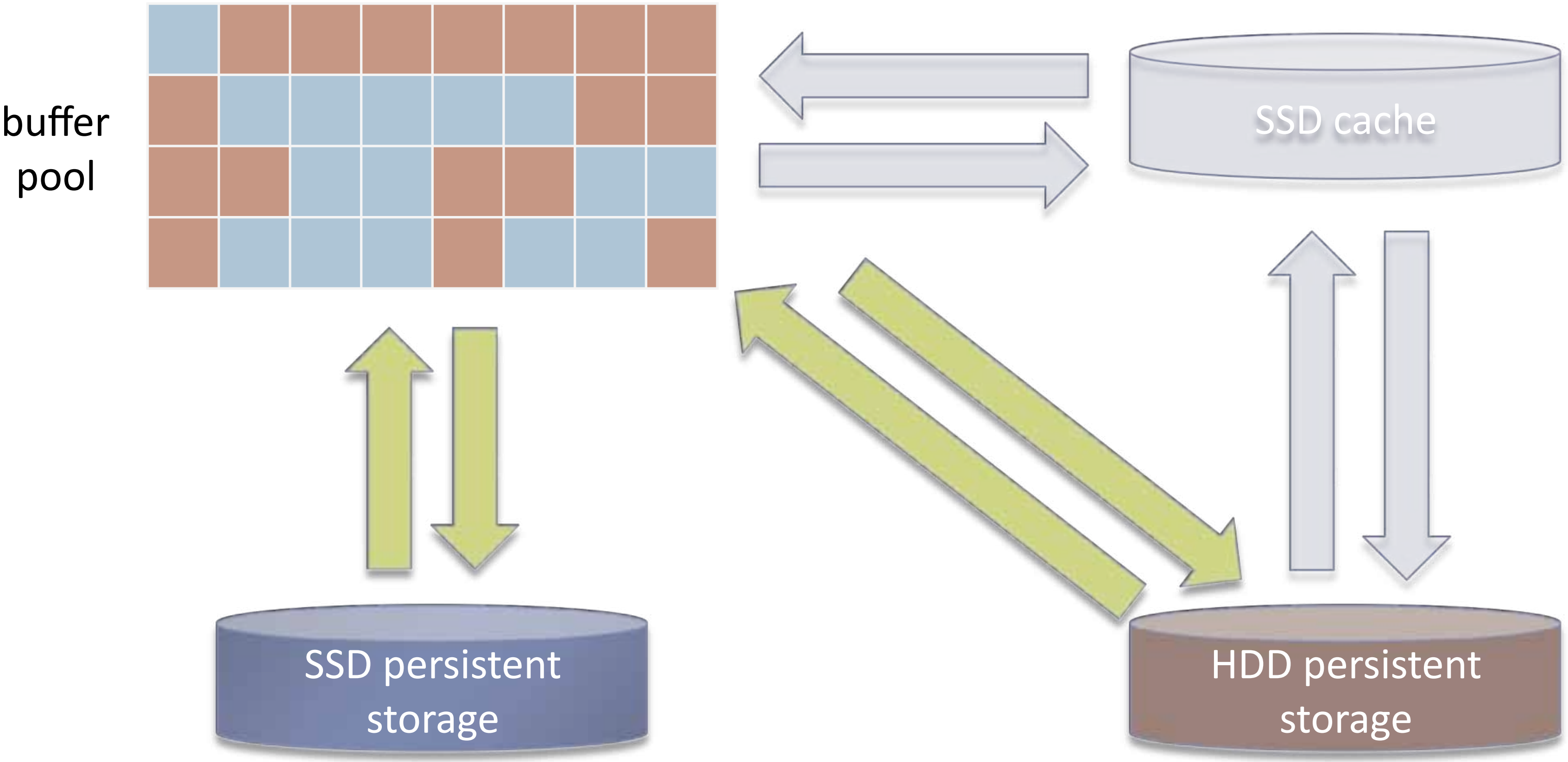


Flash memory for persistent storage

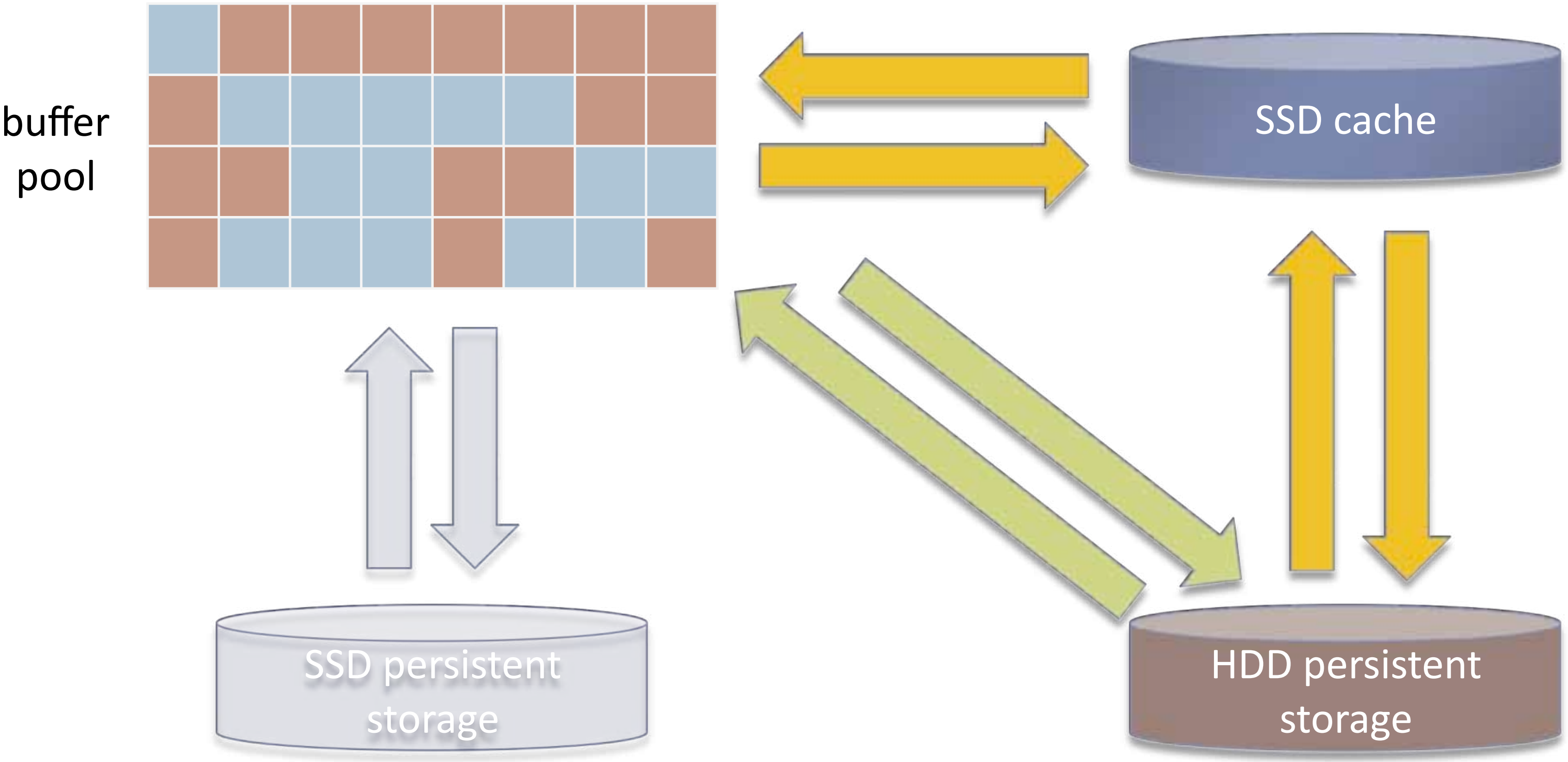


Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

Hybrid storage layer



Flash memory as cache



Outline

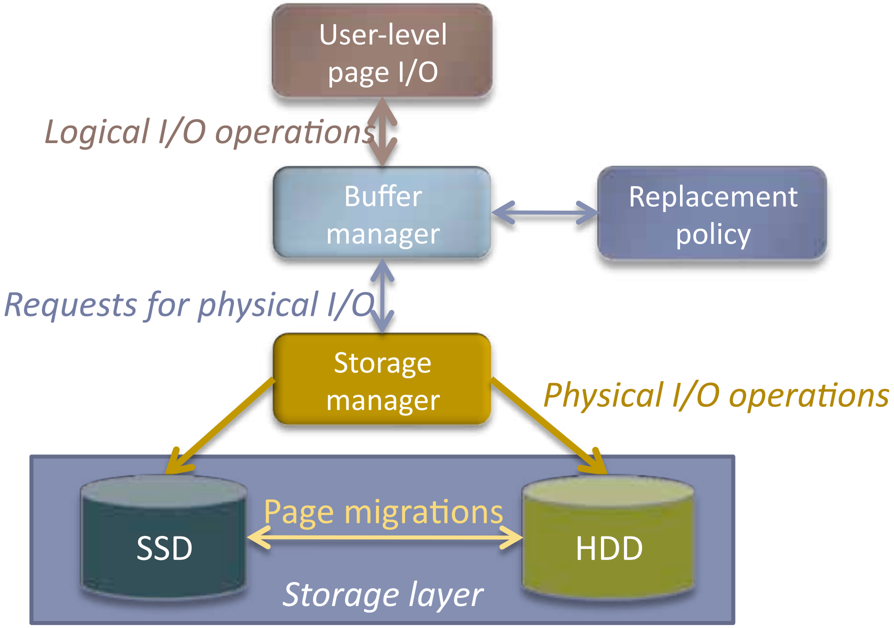
- Flash-based device design
  - Solid state drives
  - Making SSDs database-friendly
- System-level challenges
  - Hybrid systems
  - Storage, buffering and caching
  - Indexing on flash
  - Query and transaction processing

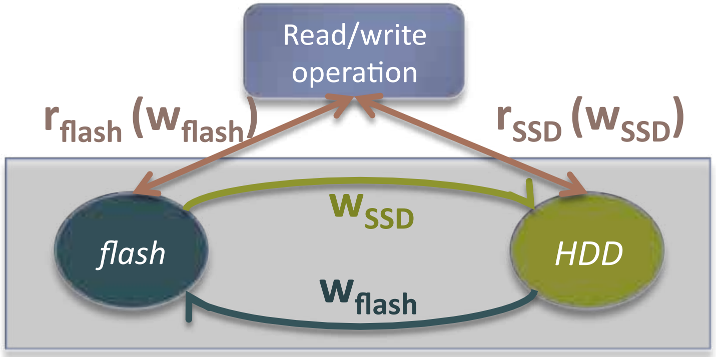
Hybrid systems

- Problem setup
  - SSDs are becoming cost-effective, but still not ready to replace HDDs in the enterprise
  - Certainly conceivable to have both SSDs and HDDs at the same level of the storage hierarchy
- Research questions
  - How can we take advantage of the SSD characteristics when designing a database?
  - How can we optimally place data across both types of medium?
- Methodologies
  - Workload detection for data placement
  - Load balancing to minimize response time
  - Caching data between disks

[K & V, VLDB 2008]

Workload-driven page placement





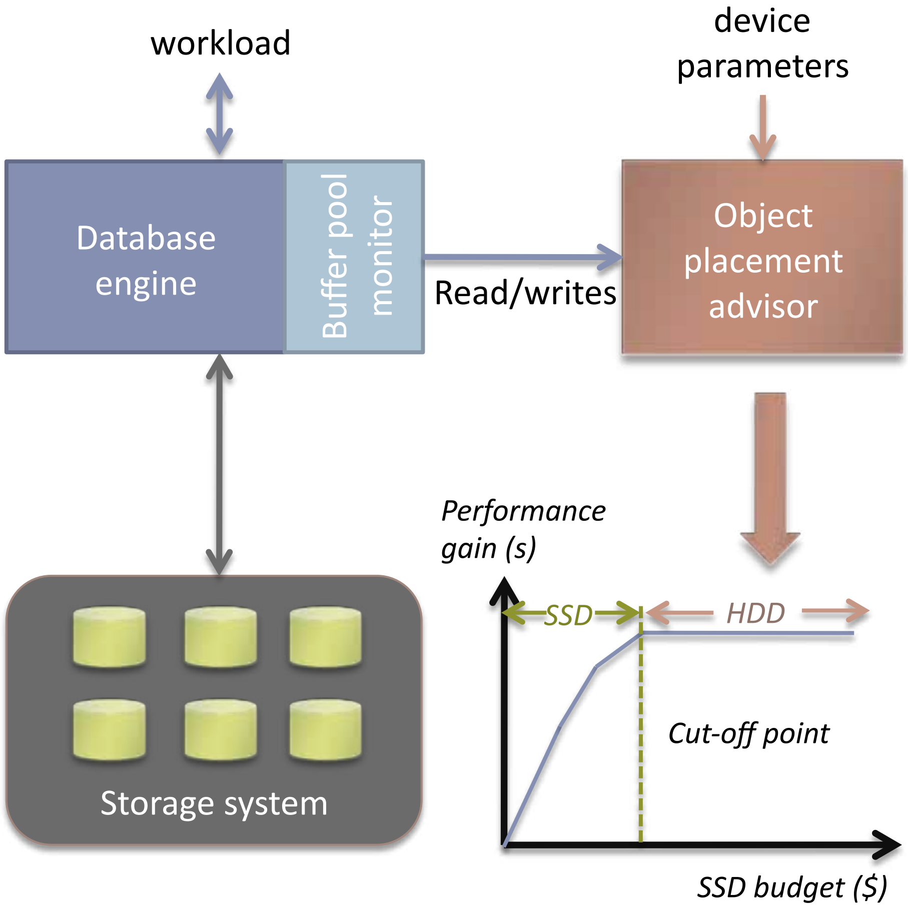
2-state task system

- Flash memory and HDD at the same level of the storage hierarchy
- Monitor page use by keeping track of reads and writes
  - Logical operations (i.e., references only)
  - Physical operations (actually touching the disk)
  - Hybrid model (logical operations manifested as physical ones)
- Identify the workload of a page and appropriately place it
  - Read-intensive pages on flash
  - Write-intensive pages on HDD
  - Migrate pages when they have expensed their cost if erroneously placed

[Canim, Mihaila, Bhattacharjee, Ross & Lang, VLDB 2009]

Object placement

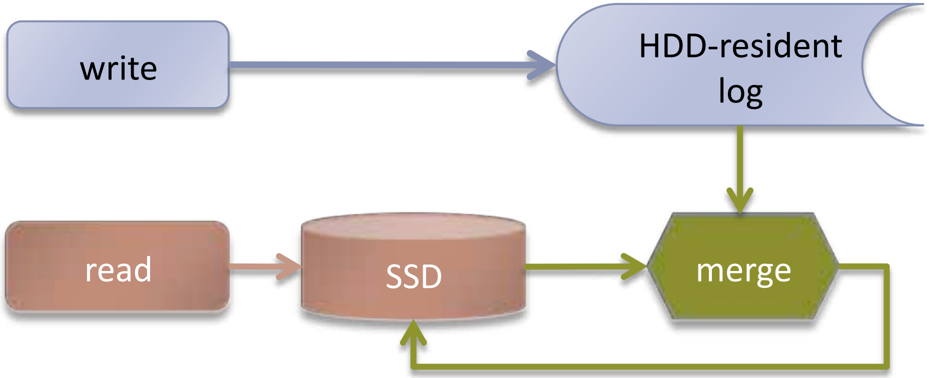
- Hybrid disk setup
- Offline tool
  - Optimal object allocation across the two types of disk
- Two phases
  - Profiling: start with all objects on the HDD and monitor system use
  - Decision: based on profiling statistics estimate performance gained from moving each object from the HDD to the SSD
- Reduce the decision to a knapsack problem and apply greedy heuristics
- Implemented in DB2



[Soundararajan, Prabhakaran, Balakrishnan, & Wobber, FAST 2010]
[Holloway PhD Thesis, UW-Madison, 2009]

Write caching

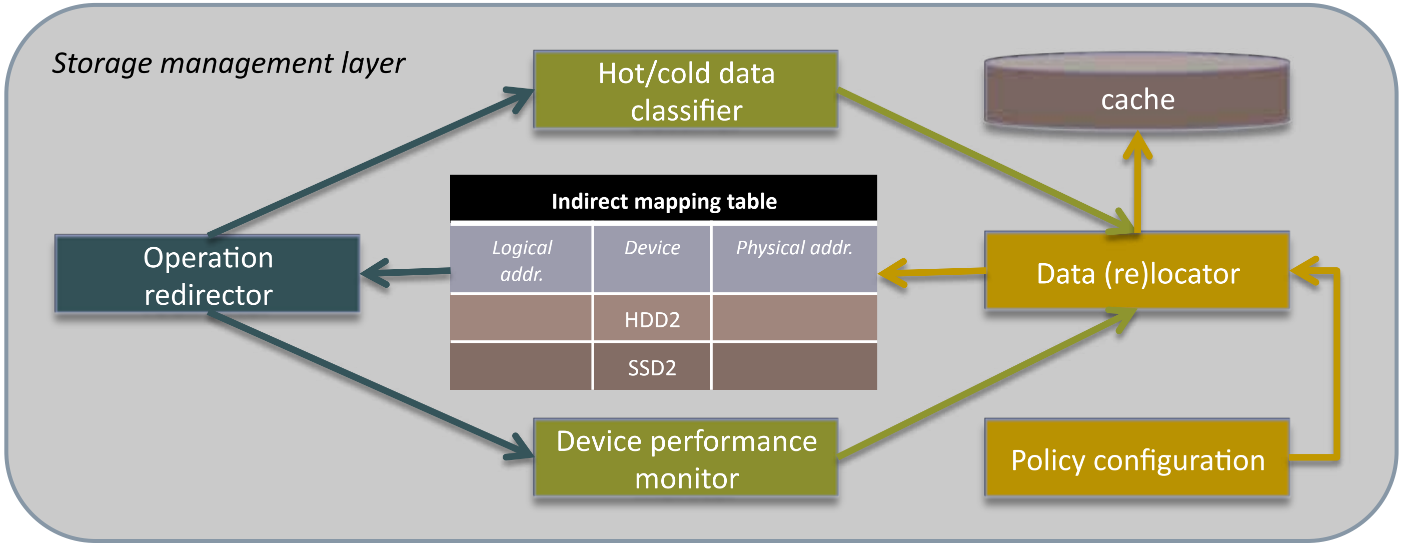
- SSD for primary storage, auxiliary HDD
- Take advantage of better HDD write performance to extend SSD lifetime and improve write throughput
- Writes are pushed to the HDD
  - Log structure ensures sequential writes
  - Fixed log size
  - Once log is full merge writes back to the SSD



[Wu & Reddy, MASCOTS 2010]

Load balancing to maximize throughput

- Setting consists of a transaction processing system with both types of disk
- Objective is to balance the load across media
  - Achieved when the response times across media are equal, i.e., a Wardrop equilibrium
  - Algorithms to achieve this equilibrium
    - Page classification (hot or cold)
    - Page allocation and migration



Indirect mapping table

Logical addr.	*Device*	*Physical addr.*
	HDD2	
	SSD2	

Outline

- Flash-based device design
  - Solid state drives
  - Making SSDs database-friendly
- System-level challenges
  - Hybrid systems
  - Storage, buffering and caching
  - Indexing on flash
  - Query and transaction processing

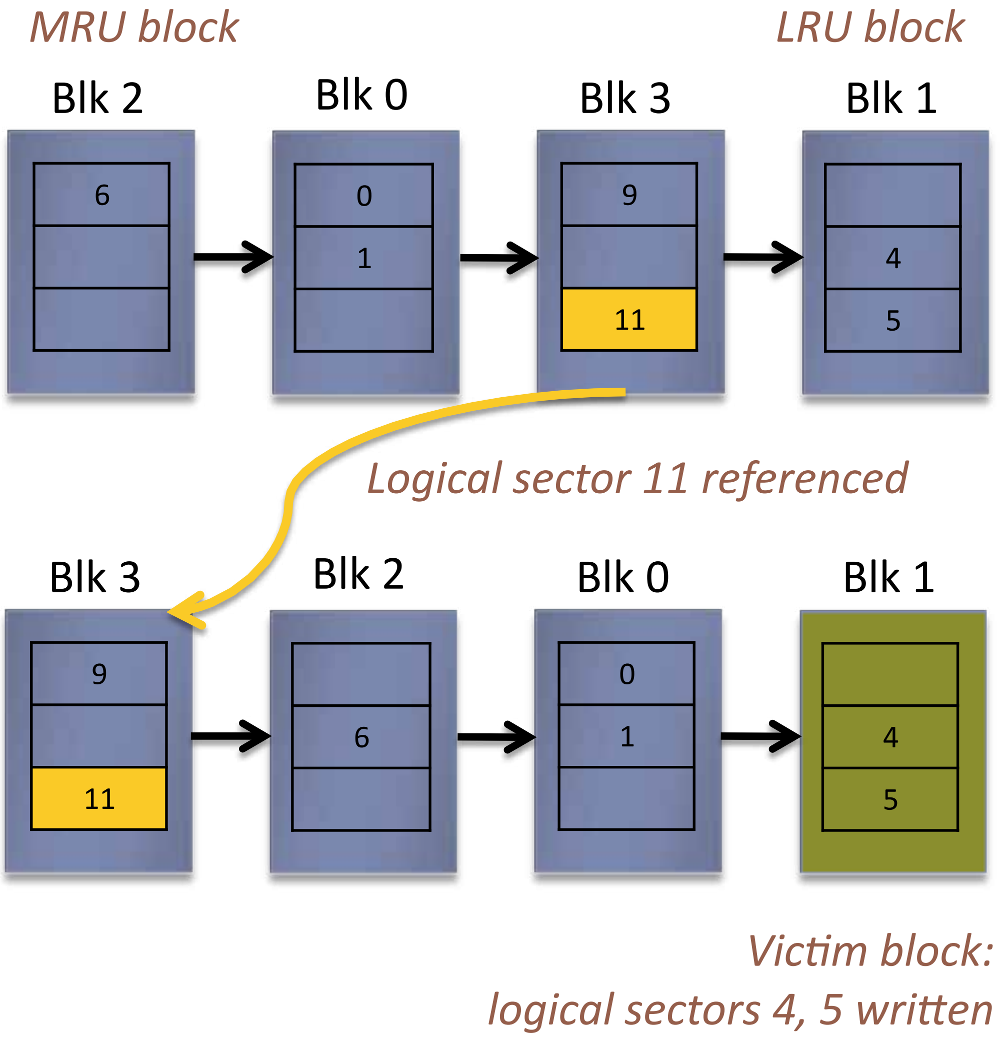
Buffering in main memory

- Problem setup
  - Flash memory is used for persistent storage
  - Typical on-demand paging
- Research questions
  - Which pages do we buffer?
  - Which pages do we evict and when?
- Methodologies
  - Flash memory size alignment
  - Cost-based replacement
  - Write scheduling

[Kim & Ahn, FAST 2008]

Block padding LRU (BPLRU)

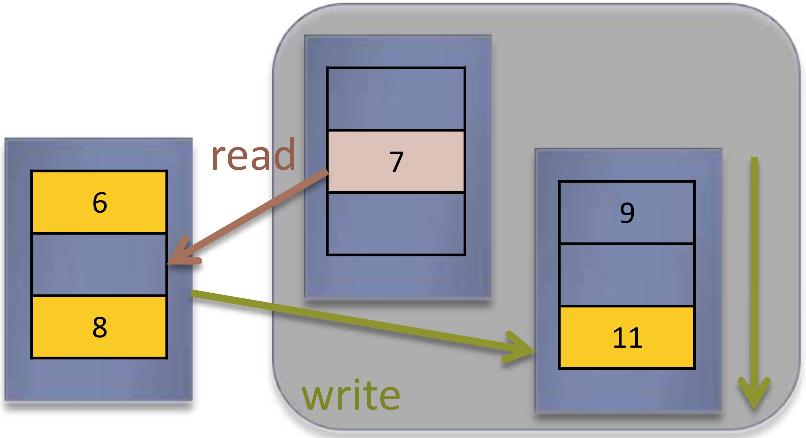
- Manages the on disk RAM buffer
- Data blocks are organized at erase-unit granularity
    - LRU queue is on data blocks
- On reference, move the entire block to the head of the queue
- On eviction, sequentially write the entire block



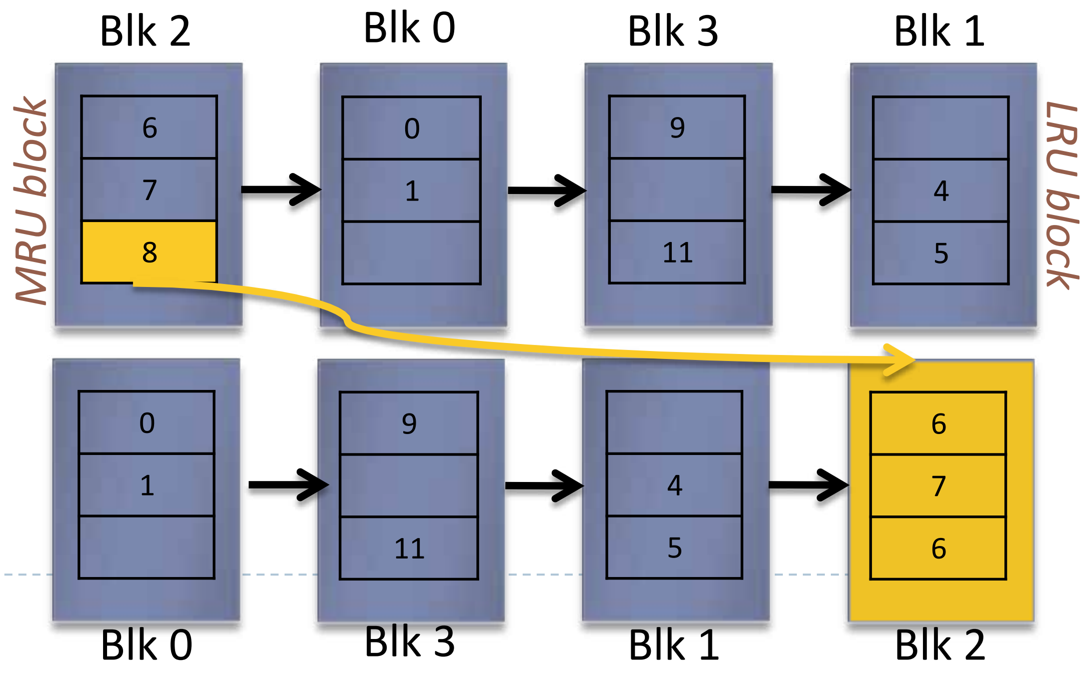
[Kim & Ahn, FAST 2008]

BPLRU: Further optimizations

- Use of padding
  - If a data block to be written has not been fully read, read what's missing and write sequentially
- LRU compensation
  - Sequentially written blocks are moved to the end of the LRU queue
  - Least likely to be written in the future



FTL reads missing sectors and replaces data block in one sequential write



Cost-based replacement

- Choice of victim depends on probability of reference (as usual)
- But the eviction cost is not uniform
  - Clean pages bear no write cost, dirty pages result in a write
  - I/O asymmetry: writes more expensive than reads
- It doesn’t hurt if we misestimate the heat of a page
  - So long as we save (expensive) writes
- Key idea: combine LRU-based replacement with cost-based algorithms
  - Applicable both in SSD-only as well as hybrid systems

[Park, Yung, Kang, Kim & Lee, CASES 2006]

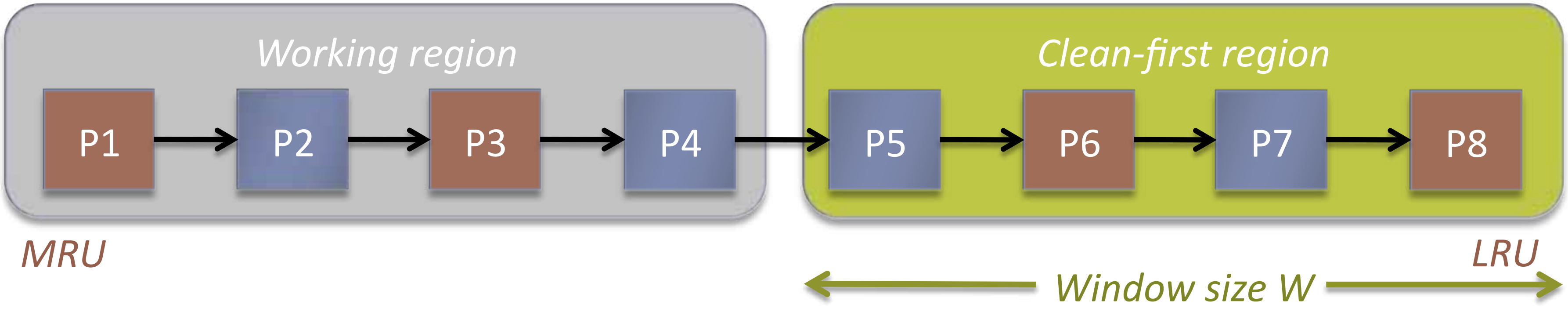
Clean first LRU (CFLRU)

- Buffer pool divided into two regions
  - Working region: business as usual
  - Clean-first region: candidates for eviction
  - Number of candidates is called the window size W
- Always evict from clean-first region
- Evict clean pages before dirty ones to save write cost
- Improvement: Clean-First Dirty-Clustered [Ou, Harder & Jin, DAMON 2009]
  - Cluster dirty pages of the clean-first region based on spatial proximity

🟫 dirty page 🟦 clean page

LRU order: P8, P7, P6, P5

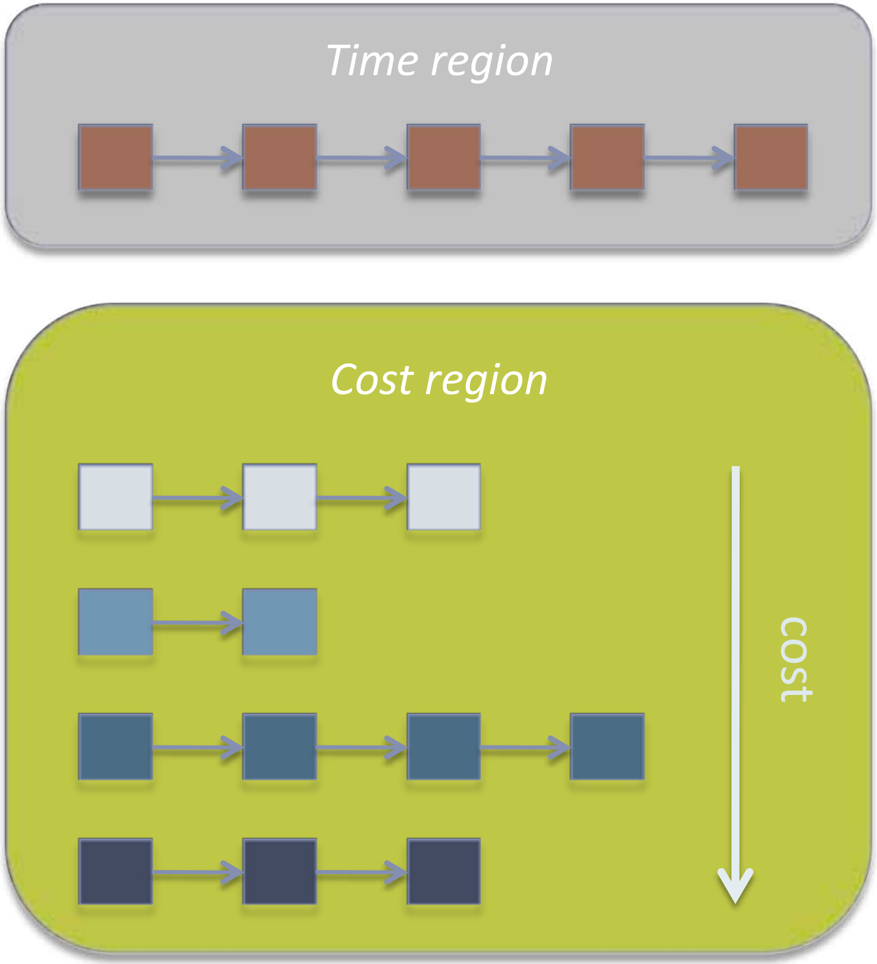
CFLRU order: P7, P5, P8, P6



[K & V, VLDB 2008]

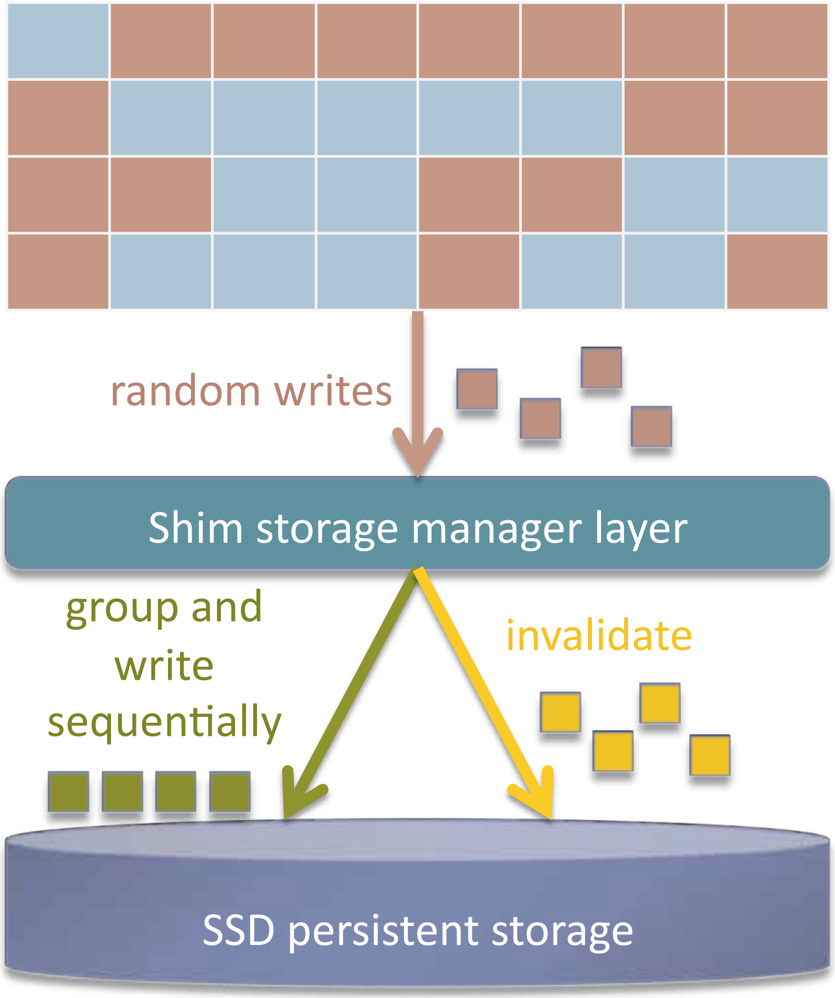
Cost-based replacement in hybrid systems

- Similar to the previous idea, but for hybrid setups
  - SSD and HDD for persistent storage
- Divide the buffer pool into two regions
  - Time region: typical LRU
  - Cost region: four LRU queues, one per cost class
    - Clean flash
    - Clean magnetic
    - Dirty flash
    - Dirty magnetic
  - Order queues based on cost
- Evict from time region to cost region
- Final victim is always from the cost region



[Stoica, Athanassoulis, Johnson & Ailamaki, DAMON 2009]

Append and pack



- Convert random writes to sequential ones
  - Shim layer between storage manager and SSD
  - On eviction, group dirty pages, in blocks that are multiples of the erase unit
  - Do not overwrite old versions, instead write block sequentially
  - Invalidate old versions
- Pay the price of a few extra reads but save the cost of random writes

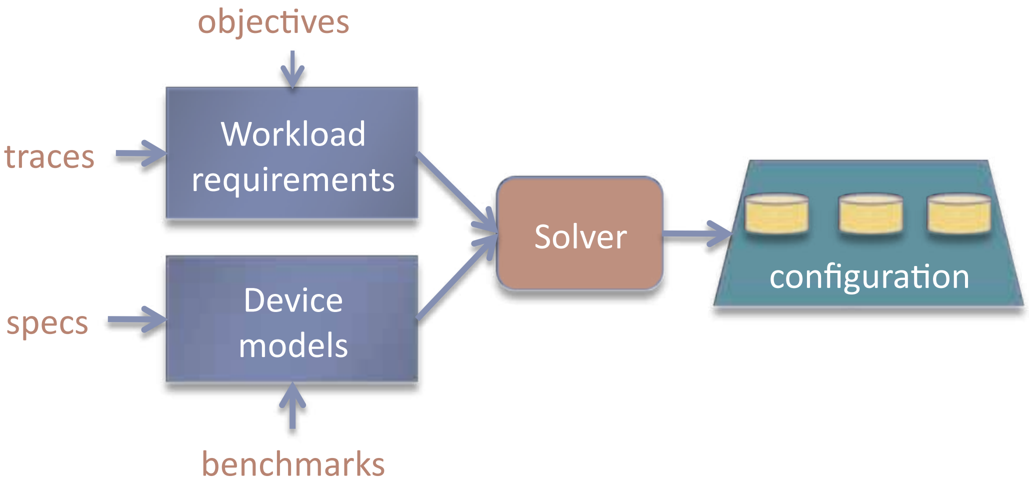
Caching in flash memory

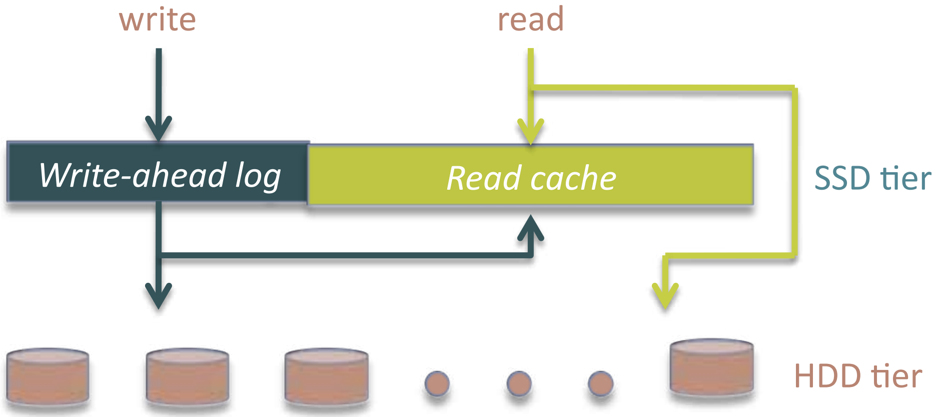
- Problem setup
  - SSD and HDD at different levels of the storage hierarchy
  - Flash memory used as a cache for HDD pages
- Research questions
  - When and how to use the SSD as a cache?
  - Which pages to bring into the cache?
  - How to choose victim pages?
- Methodologies
  - Optimal choice of hardware configuration
  - SSD as a read cache
  - Flash-resident extended buffer pools

[Narayanan, Thereska, Donnelly, Elniketi & Rowstron, EuroSys 2009]

Incorporating SSDs into the enterprise

- Disciplined way of introducing SSD storage
- Migration from HDDs to SSDs
  - Requirements and models analytically to solve the configuration problem
- Simply replacing HDDs with SSDs is not cost-effective
- SSDs are best used as a cache
  - 2-tiered architecture
  - Log and read cache on the SSDs, data on the HDDs
  - But even then the benefits are limited



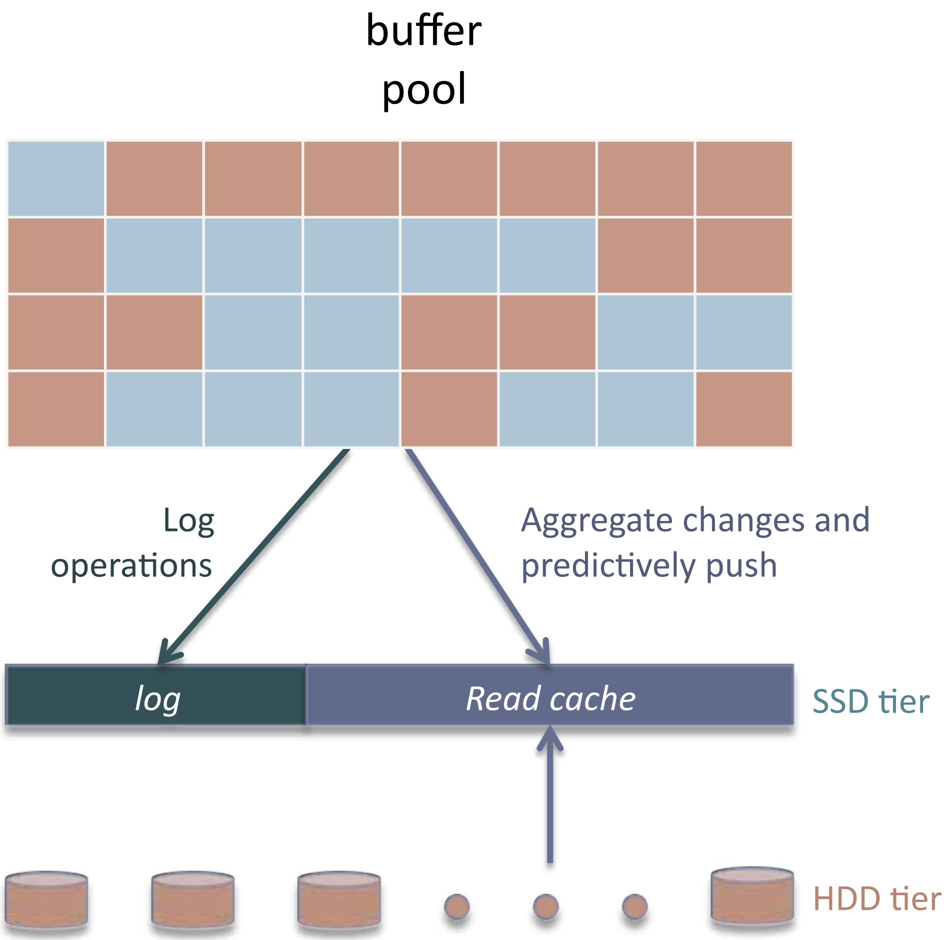


Bonnet, Bouganim, Koltsidas, Viglas, VLDB 2011

[Leventhal, Commun. ACM, 51(7) 2008]

The ZFS perspective

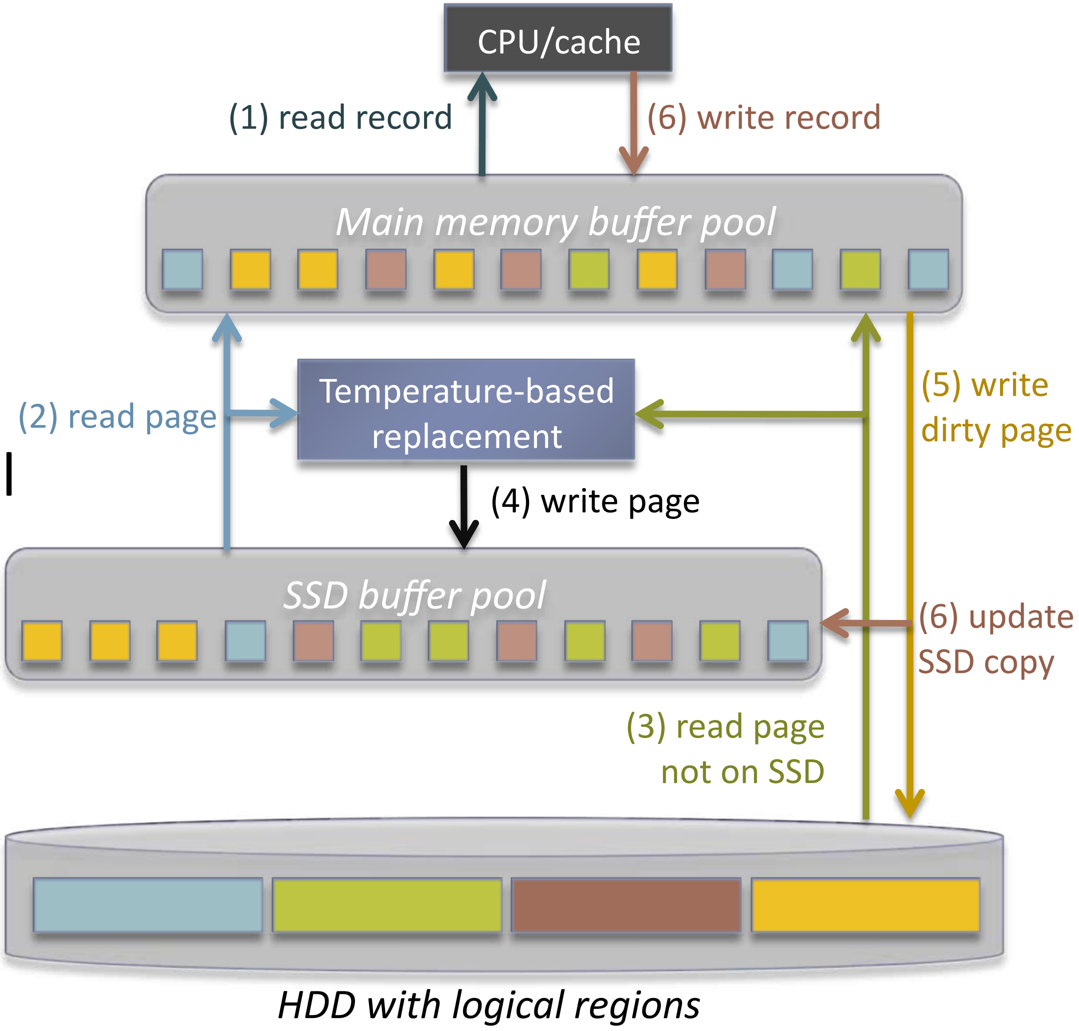
- Multi-tiered architecture
- Combination of logging and read caching
  - Flash memory is good for large sequential writes in an append-only fashion (no updates)
  - Also good as a read cache for HDD data
- Evict-ahead policy
  - Aggregate changes from memory and predictively push them to flash to amortize high write cost



[Canim, Mihaila, Bhattacharjee, Ross & Lang, PVLDB 2010]
[Holloway PhD Thesis, UW-Madison, 2009]

DBMS buffer pool extensions

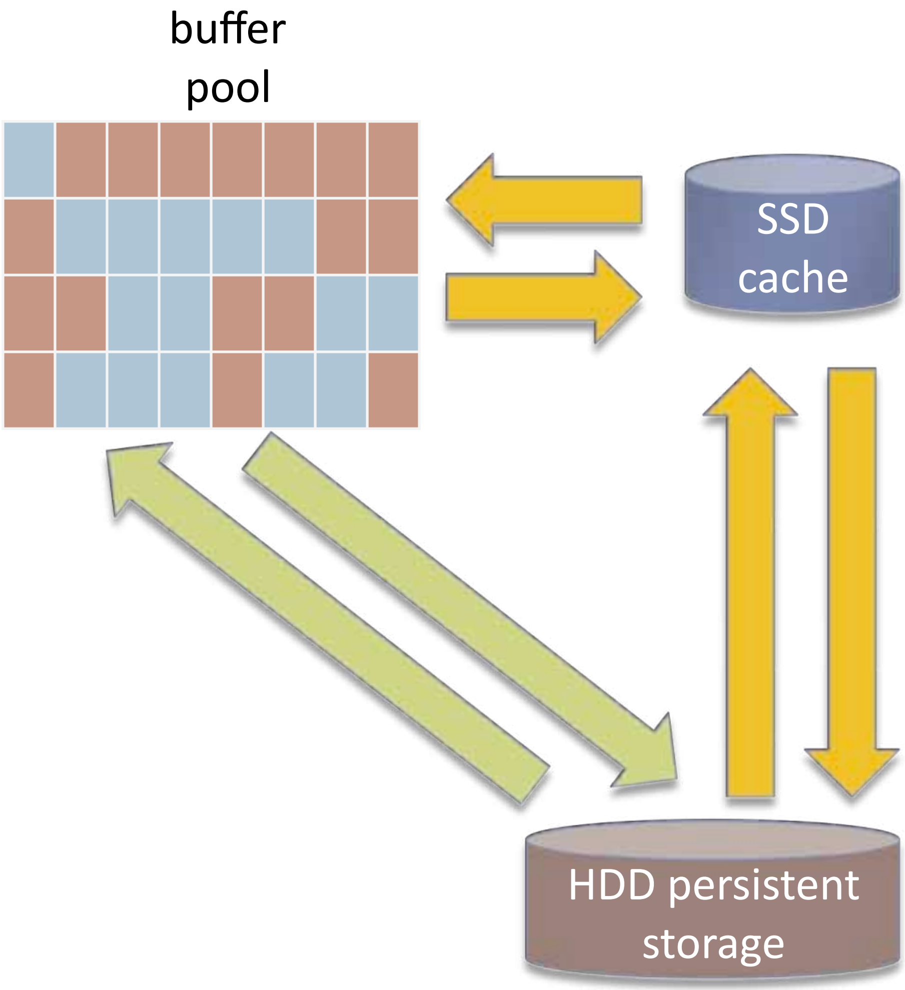
- Again a multi-tiered approach
- Policies and algorithms for caching in flash-resident buffer pools
- SSD as secondary buffer pool
  - Temperature-based eviction policy from memory to SSD
  - Pages are cached only if hot enough
- Algorithms for syncing data across the caches



[K & V, ADBIS 2011]

Putting it all together

- Extensive study of using flash memory as cache
- Page flow schemes dictate how data migrates across the tiers
  - Inclusive: data in memory is also on flash
  - Exclusive: no page is both in memory and on flash
  - Lazy: an in-memory page may or may not be on flash depending on external criteria
- Cost model predicts how a combination of workload and scheme will behave on configuration
- No magic combination; different schemes for different workloads and different HHDs and SSDs



Outline

- Flash-based device design
  - Solid state drives
  - Making SSDs database-friendly
- System-level challenges
  - Hybrid systems
  - Storage, buffering and caching
  - Indexing on flash
  - Query and transaction processing

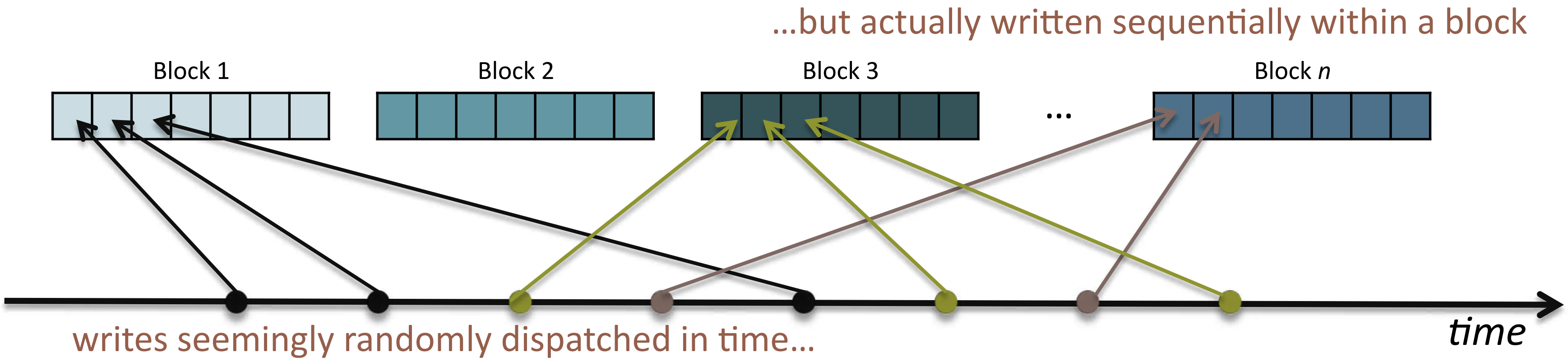
Indexing

- Problem setup
  - While presenting the same I/O interface as HDDs, flash memory has radically different characteristics
  - I/O asymmetry, erase-before-write limitation
- Research questions
  - How should we adapt existing indexing approaches?
  - How can we design efficient secondary storage indexes – potentially for more than one metric?
- Methodologies
  - Avoid expensive operations when updating the index
  - Self-tuning indexing, catering for flash-resident data
  - Combine SSDs and HDDs for increased throughput

[Nath & Gibbons, VLDB 2008]

Semi-random writes

- Starting point is studying typical write access patterns in the context of sampling
- Fact: random writes hurt performance
- But careful analysis of a typical workload shows that writes are rarely completely random
- Rather, they are semi-random
  - Randomly dispatched across blocks, sequentially written within a block
  - Similar to the locality principles of memory access
  - Take advantage of this at the structure design level and when issuing writes
  - Bulk writes to amortize write cost


…but actually written sequentially within a block
Block 1
Block 2
Block 3
…
Block *n*
writes seemingly randomly dispatched in time…
time

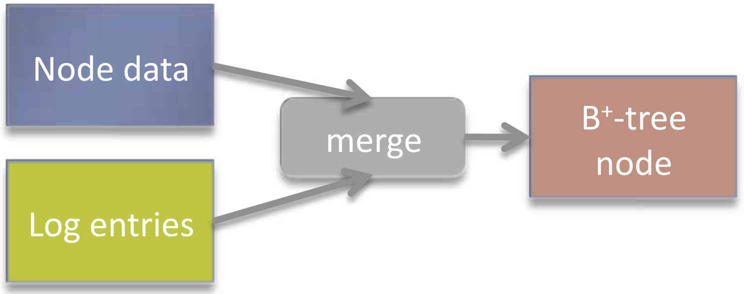
[Nath & Kansal, ISPN 2007]

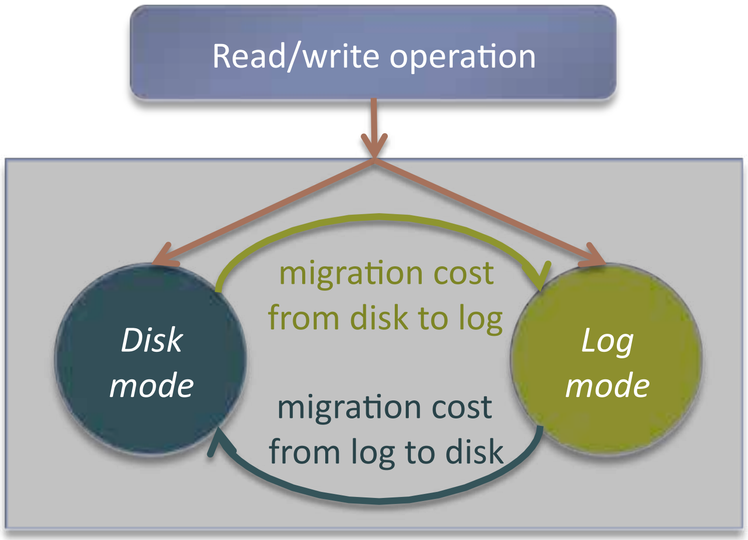
FlashDB: self-tuning B^+-tree

- Reads are cheap, writes are potentially expensive if random
- Two modes for B^+-tree nodes
  - Disk mode: node is primarily read
  - Log mode: node is primarily updated; instead of overwriting, maintain log entries for the node and reconstruct on demand
- Translation layer presents uniform interface for both modes
- System switches between modes by monitoring use

- Similar logging approach in [Wu, Kuo & Chang, ACM Trans. On Embedded Systems, 6(3), 2007]
  - Difference is in when and how writes are applied
  - Buffered first, then batched and applied by the B+-tree FTL

log mode



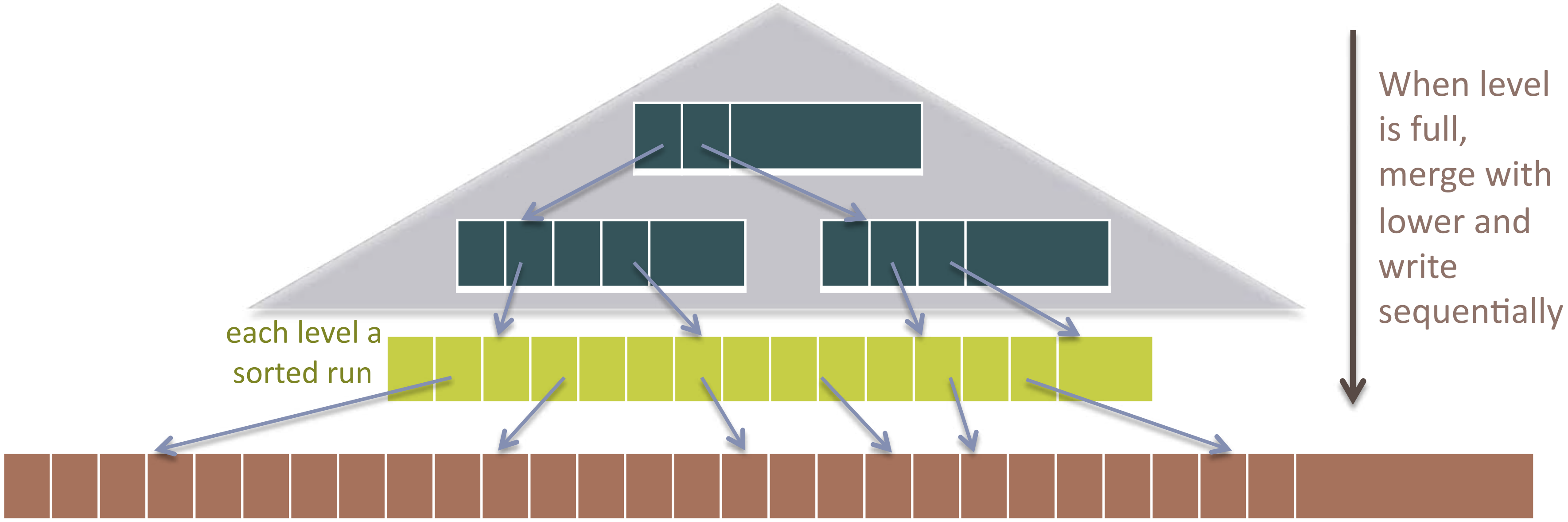

Read/write operation
Disk mode
Log mode
migration cost from disk to log
migration cost from log to disk

2-state task system

[Li, He, Luo & Yi, ICDE 2009]

Making B$^+$-trees flash-friendly (the FD-tree)

- Multiple levels in the index
  - Each progressive level of double size
  - So long as we are not updating in place, and also performing large sequential writes to amortize the cost, it's all good
- FD-tree levels are sorted runs
  - Head-tree (first levels) in main memory
  - Once a lower level exceeds its capacity it is merged with the next one
  - Special entries (fences) are used to maintain the structure and deal with potential skew

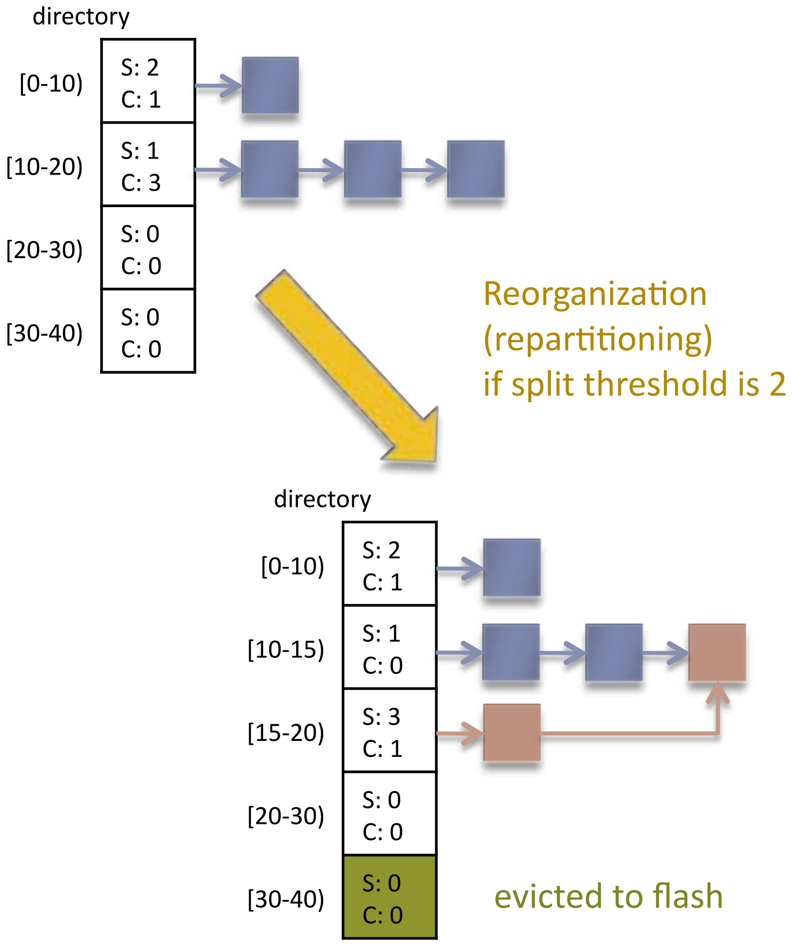


Spatial indexing

- Similar observations as with B^+-trees can be made on R-trees
  - They're tree indexes after all!
  - Lesson is largely the same: one needs to carefully craft the structure and its algorithms for the new medium
  - Batch updates to amortize write costs
    - [Wu, Chang & Kuo, GIS 2003]
  - Trade cheap reads for expensive writes by introducing imbalance
    - [K & V, SSTD 2011]
- Systematic study on performance of R-trees on SSDs in [Emrich, Graf, Kriegel, Schubert & Thoma, DAMON 2010]
  - SSDs not as sensitive as HDDs to page size
  - Capable of addressing higher dimensional data in less time

[Zeinalipour-Yazti, Lin, Kalogeraki, Gunopulos & Najjar, FAST 2005]

Hash indexing based on MicroHash



directory

[0-10)	S: 2 C: 1
[10-20)	S: 1 C: 3
[20-30)	S: 0 C: 0
[30-40)	S: 0 C: 0

Reorganization (repartitioning) if split threshold is 2

directory

[0-10)	S: 2 C: 1
[10-15)	S: 1 C: 0
[15-20)	S: 3 C: 1
[20-30)	S: 0 C: 0
[30-40)	S: 0 C: 0

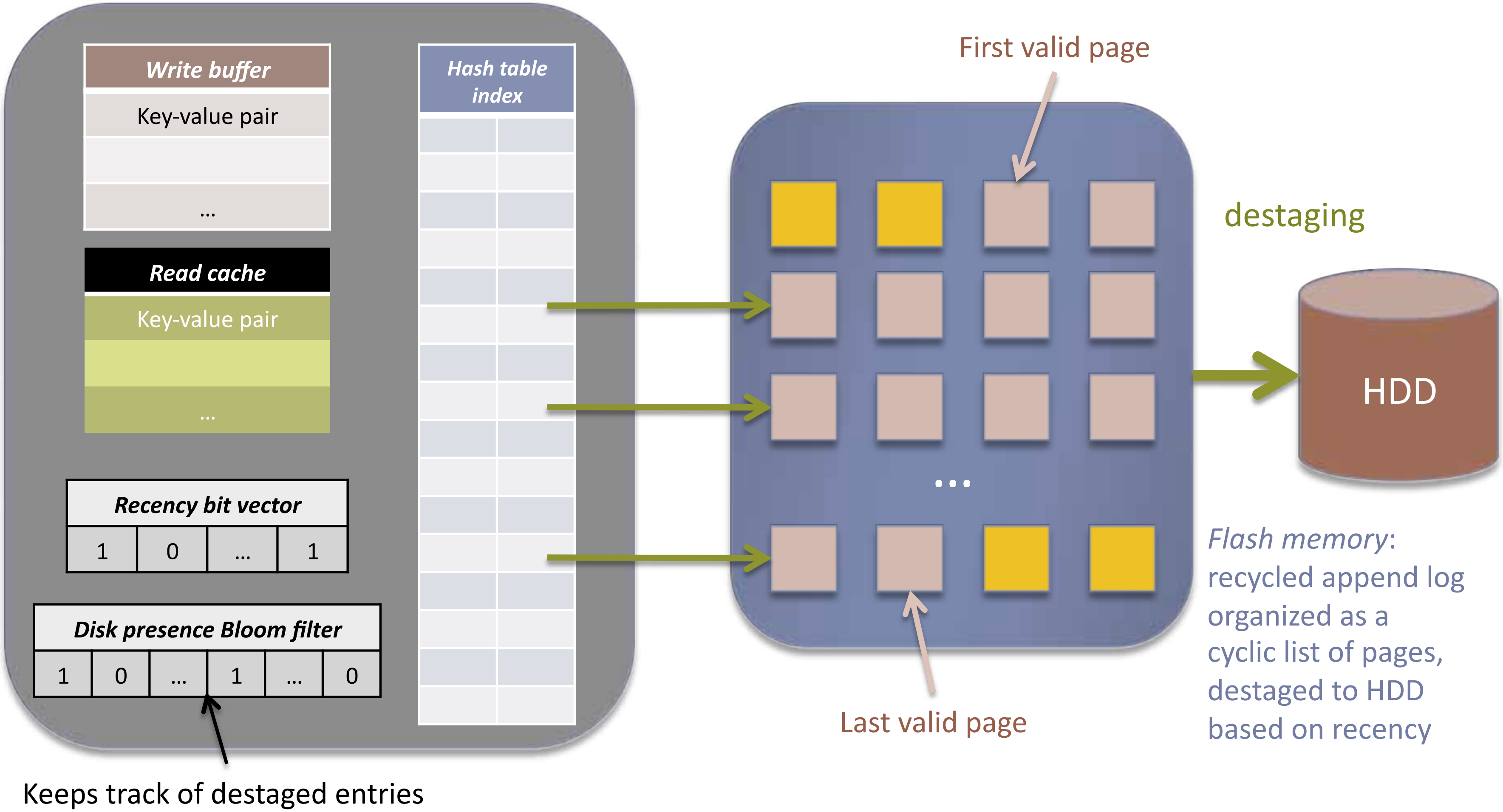
evicted to flash

- Setup is sensor nodes with limited memory and processing capabilities
- Similar to extendible hashing techniques
  - Directory keeps track of bucket boundaries
    - For each bucket maintain the last time it was used S and the number of times it has been split C
  - Progressive expansion based on equi-width splitting
    - Expansion triggered when the number of splits exceeds some threshold
    - Infrequently used buckets are flushed to SSD
  - Deletion through garbage collection and reorganization
    - Batch updates are helpful
- Generalization of linear hashing by lazy splitting in [Xiang, Zhou & Meng, WAIM 2008]
  - Takes advantage of batch updates

[Debnath, Sengupta & Li, VLDB 2010]

A design for hybrid systems (FlashStore)

RAM memory: write and read buffers and metadata



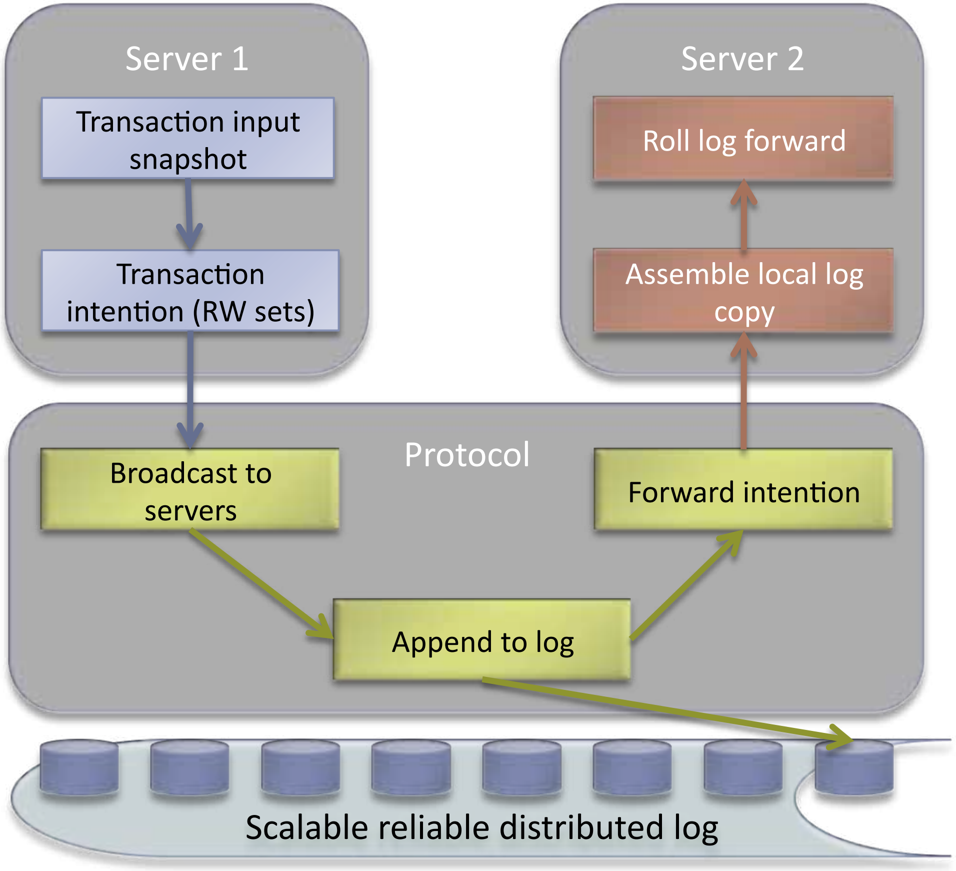
Keeps track of destaged entries

Flash memory: recycled append log organized as a cyclic list of pages, destaged to HDD based on recency

[Bernstein, Reid & Das, CIDR 2010]

Transaction management on flash: Hyder

- A different architecture for transaction management
  - The target is data centers
    - Shared data, multi-core nodes
  - Need for scale-out
  - Log-structured multi-versioned database
    - "The log is the database"
  - No partitioning
- Raw flash chips used for storage
  - Though SSDs or even HDDs may be used as well
- Three-layered architecture
  - Storage layer maintains shared log
  - Index layer supports lookup and versioning
  - Transaction layer provides isolation and continuously refreshes the database cache by running the "meld" algorithm



Outline

- Flash-based device design
  - Solid state drives
  - Making SSDs database-friendly
- System-level challenges
  - Hybrid systems
  - Storage, buffering and caching
  - Indexing on flash
  - Query and transaction processing

Query and transaction processing

- Problem setup
  - Same types of query and transactional workload
  - Different medium; not what existing approaches have been optimized for
- Research questions
  - Are there problems that best fit to SSDs?
  - Does one need radically different approaches, or slight adaptations?
  - Where in the storage hierarchy should we use SSDs and how?
- Methodologies
  - Flash-aware algorithms either by design or through adaptation
  - Offload parts of the computation to flash memory
  - Economies of scale

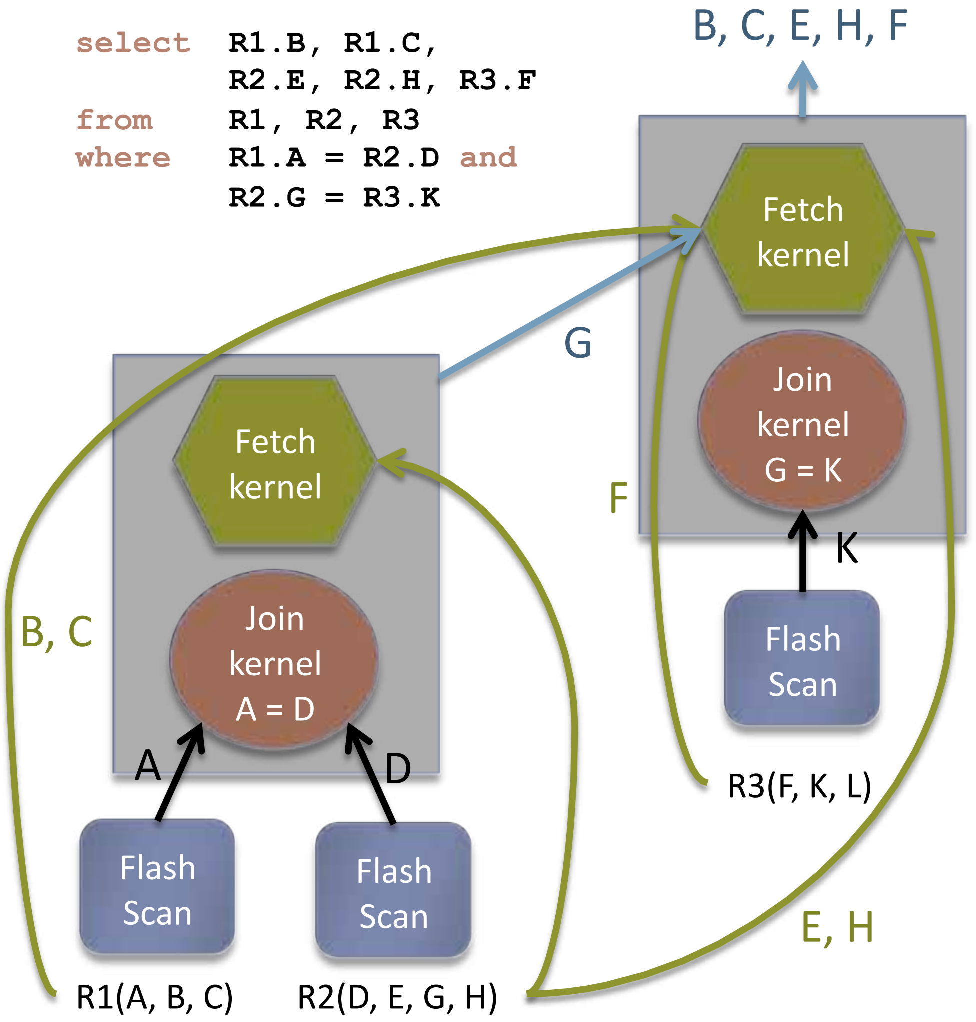
Old stories, new toys

- Impact of selectivity on predicate evaluation [Myers, MSc Thesis, MIT, 2007]
  - Overall, as selectivity factor increases performance degrades (needle in haystack queries)
  - At times HDDs might outperform SSDs
- Join processing on SSDs using algorithms designed for HDDs [Do & Patel, DAMON 2009]
  - SSD joins may well become CPU-bound, so ways to improve the CPU performance become salient
  - Trading random reads for random writes pays off
  - Random writes result in varying I/O and unpredictable performance
  - Blocked I/O still improves performance
  - Block size should be a multiple of the page size

Impact of storage layout: the FlashJoin

- Storage layout based on PAX
- No need to retrieve what's not necessary
  - Use a specialized operator (FlashScan) for on-the-fly projections over PAX
- Delegate join computation to two steps
  - Fetching data by projecting relevant (the fetch kernel)
  - Evaluate the join predicate (through the join kernel) and materialize the result in a join index

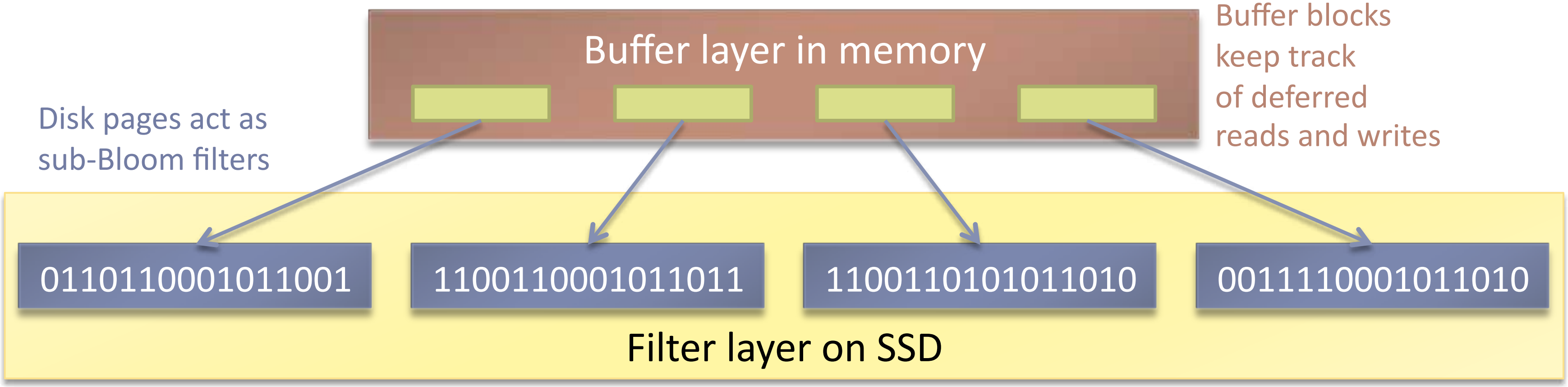
```
select  R1.B, R1.C,
        R2.E, R2.H, R3.F
from    R1, R2, R3
where   R1.A = R2.D and
        R2.G = R3.K
```



[Canim, Mihaila, Bhattacharjee, Lang & Ross, ADMS 2010]

Membership queries

- Motivation: poor locality of Bloom filters
  - Hurts cache performance in CPU-intensive applications
  - Good candidate for offloading to flash memory
  - Good random read performance compared to HDDs, but we still need to cross the memory-disk boundary
- Solution: being lazy pays off
  - Defer reads and writes and through buffering
  - Introduce hierarchical structure to account for disk-level paging



Disk pages act as sub-Bloom filters

Buffer blocks keep track of deferred reads and writes

[Lee , Moon, Park, Kim & Kim, SIGMOD 2008]

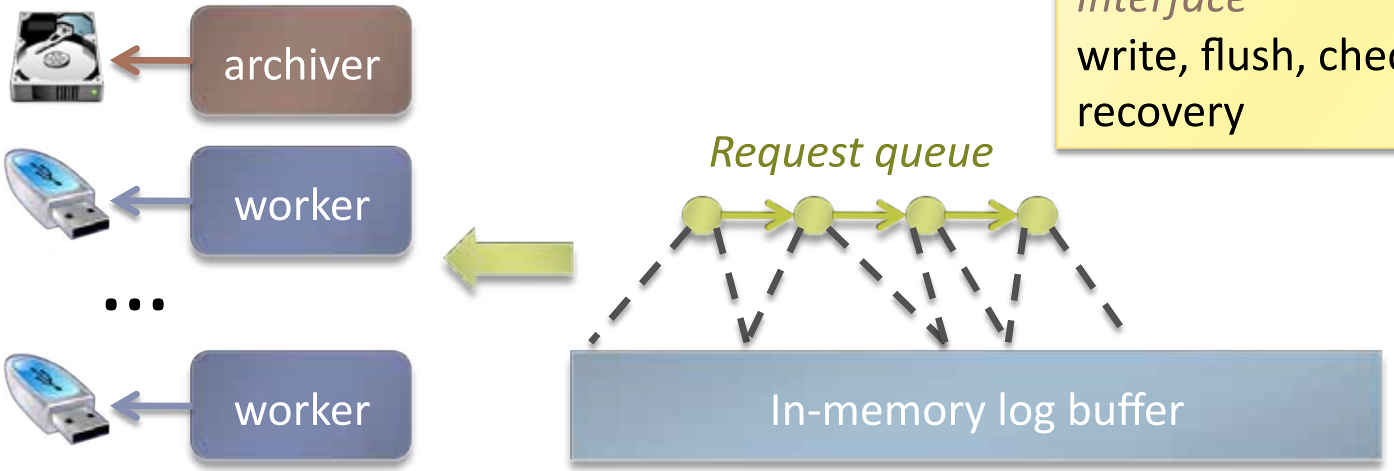
Database operations suited for SSDs

- Given transactional and query processing workloads, which operations are SSDs better for?
- Study of the I/O patterns
- Identify server storage spaces that exhibit SSD-friendly I/O patterns
  - Tables, indexes, temporary storage, log, rollback segments
- Secondary structures are better suited for SSDs
  - Long sequential writes (no updates) and random reads
  - Performance improvement more than one order of magnitude when logging and rollback segments are delegated to SSDs
  - Factor of two improvement when temporary storage is on SSDs

[Chen, SIGMOD 2009]

Inexpensive logging in flash memory

- As mentioned, logging is one of the best fits for SSDs
  - Typically, writes are appended to the log
- The online version of the log is usually small
- USB flash memory is cheap and USB ports are abundant
- Intuition: spread the log across multiple cheap USB flash disks
  - Provide simple logging interface



Interface
write, flush, checkpoint; recovery

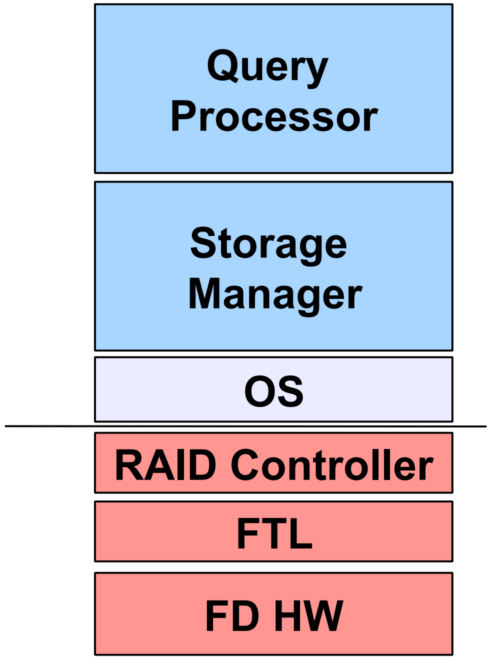
Tutorial Outline

1. Introduction (Philippe)
2. Flash devices characteristics (Luc)
3. Data management for flash devices (Stratis)
4. Two outlooks (Stratis & Philippe)

Outlook

- SSD is a diverse class of devices
  - The only common characteristic of all members of the class is the excellent random read performance
  - Underlying technology affects performance in other operations
  - SSDs do not completely dominate HDDs – not yet
    - Some types of SSD may be an order of magnitude slower than HDDs in random writes
- Where do they fit in the database stack?
  - Persistent storage – maybe in combination with HDDs
  - Read cache of HDD data
  - Transactional logging
  - Using the HDD as a log-structured write-cache for the SSD
  - Temporary storage and staging area
  - Any of the above
- More research necessary at the SSD/DB interfaces

Design Space



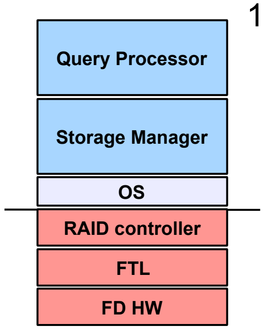
Sources of increased complexity:
Improve Performance AND predictability,
no stable performance contract at interfaces,
high utilization,
d(techno)/dt

Which IOs are issued?

How are IOs scheduled?

How are IOs interpreted?

Cross-layer issues:
- Avoid duplicating work
- Split work most effectively
- Schedule work most effectively
- Avoid arbitrary limitations



Performance Contract

Flash Devices Characteristics:

- Predictable, unconstrained and <u>inefficient</u> : USB key or low-end SSD

⇨ **Existing DBMS are probably good enough**

- Predictable, <u>constrained</u> and efficient : mininal FTL

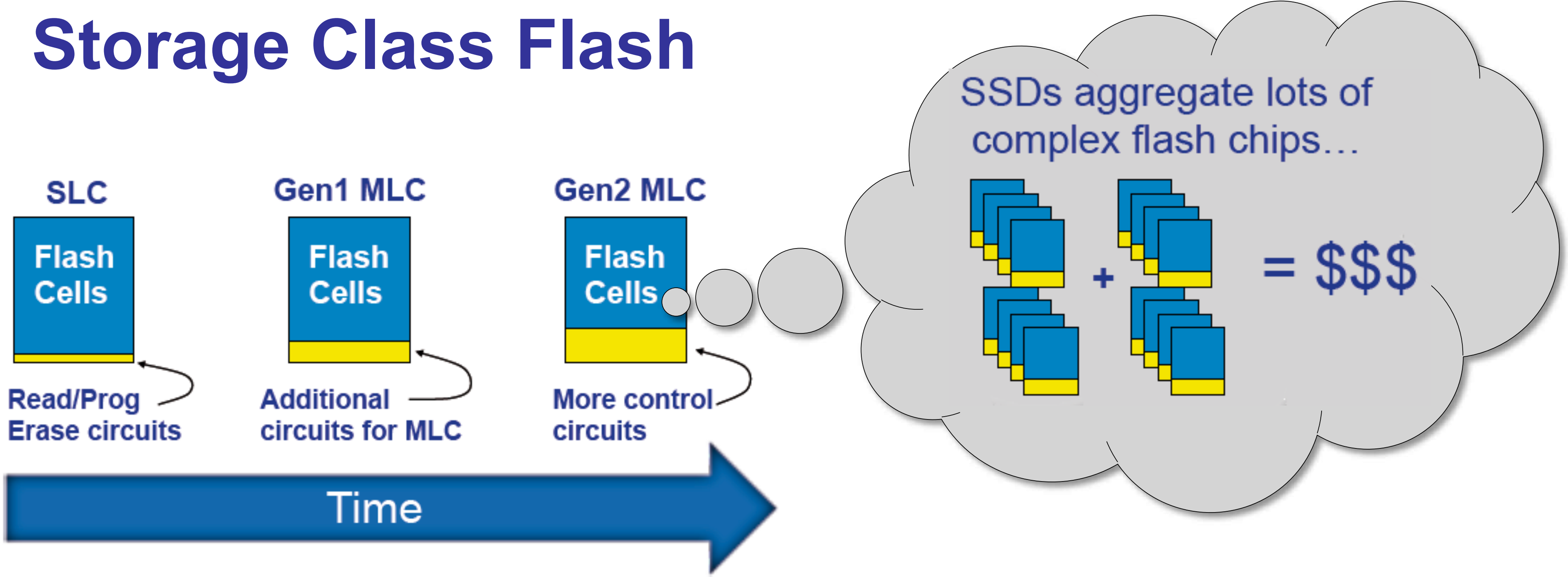
⇨

- **<u>Which DBMS functions can efficiently enforce constraints? How?</u>**
- **<u>Performance Modelling.</u>**

- Unpredictable and unconstrainted

⇨ **Derive constraints for efficient regimes (ad-hoc)**

Flash chips trend: Less into the chip: Storage Class Flash



SLC

Flash Cells

Read/Prog Erase circuits

Gen1 MLC

Flash Cells

Additional circuits for MLC

Gen2 MLC

Flash Cells

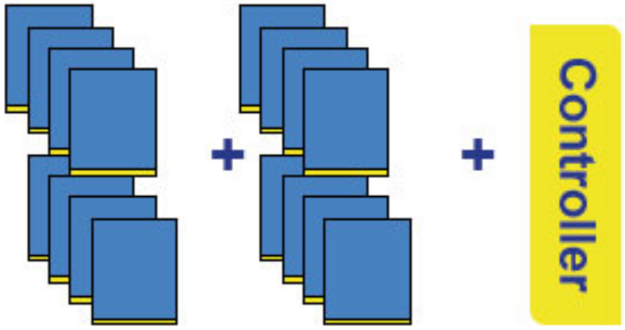
More control circuits

Time

SSDs aggregate lots of complex flash chips…

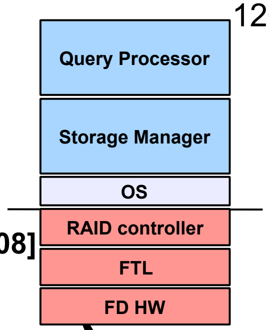
+ = $$$

SSD Solution:
Simplify the flash and Use intelligent controllers



+ + Controller

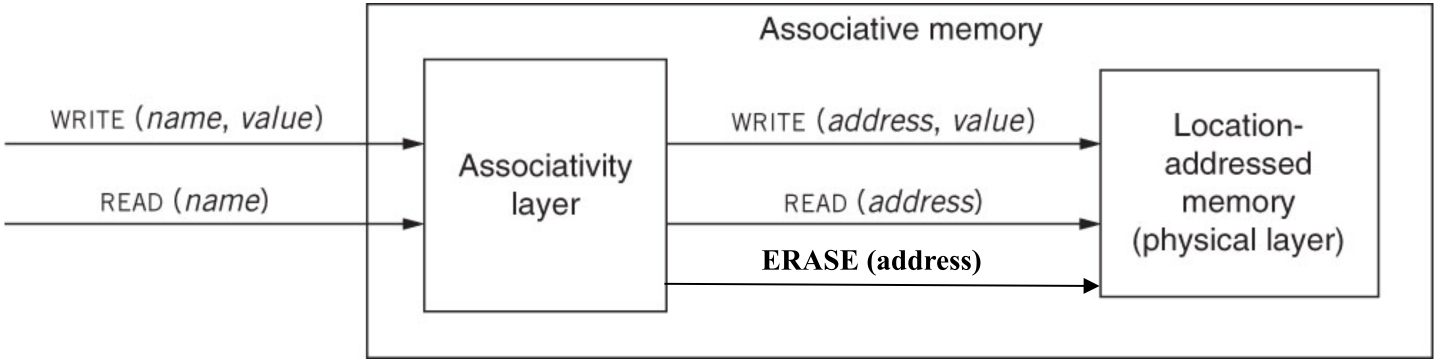
Scaramuzzo (FMS 2010)



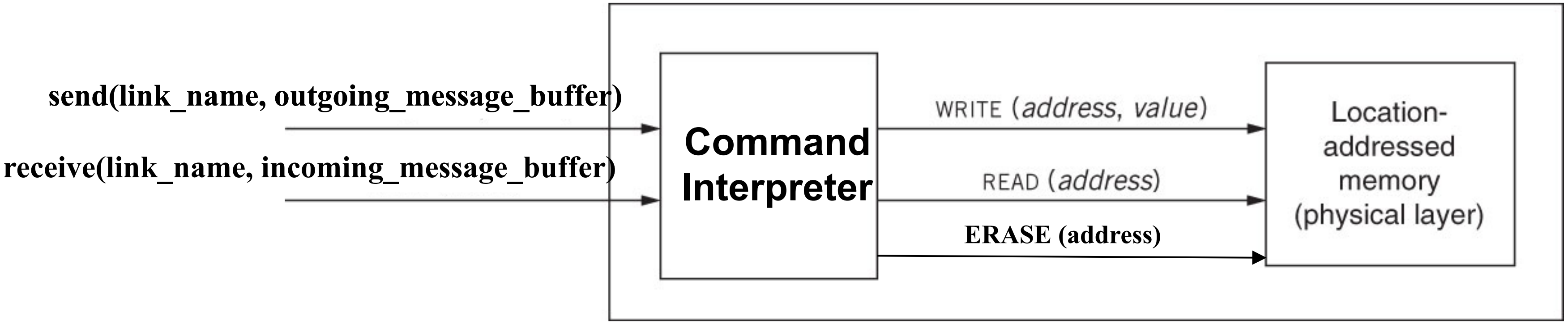
Dealing with complexity

[Schloser et al, CMU tech report 2003; Schloser et al, FAST 2004; Prabakharan et al., OSDI 2008]

From a memory abstraction (*block device*)...



... to a communication abstraction (*rich interface*)


send(link_name, outgoing_message_buffer)
receive(link_name, incoming_message_buffer)
Command Interpreter
WRITE (address, value)
READ (address)
ERASE (address)
Location-addressed memory (physical layer)

Figures courtesy of Koschaak and Saltzer

TRIM command

- ATA/ATAPI Command set

  Data Set Management command (TRIM)

  TRIM(LBA) is a <u>hint</u> to FTL to unmap LBA-PBA

    - Unmapping is asynchronous, i.e., fast (if at all executed)
    - Read after TRIM is unspecified

- Pushed by file systems community

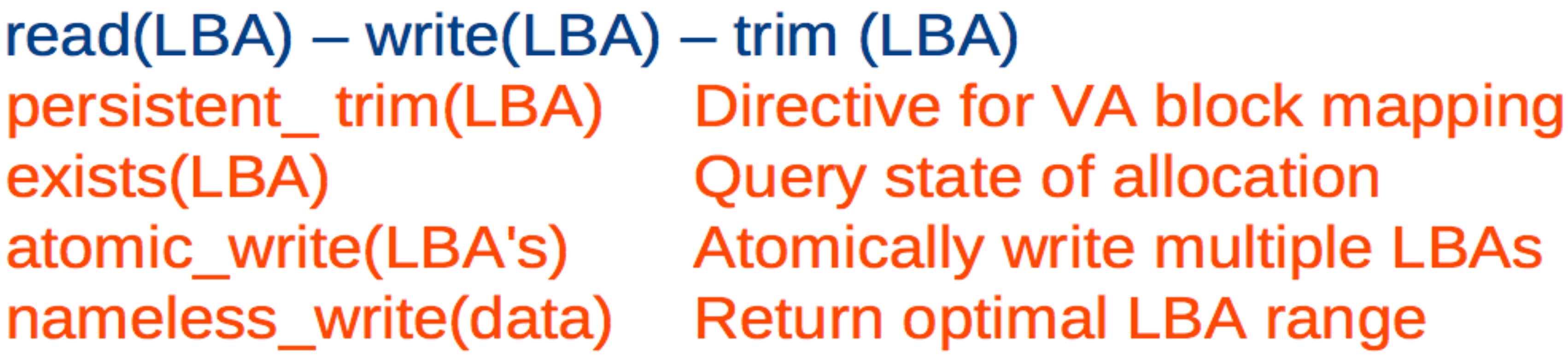
  Supported in Linux kernel 2.6.33+ and Windows 7/2008R2

  Implemented in X25

    - X25-M has 80GB capacity, but provides LBA for 74,4 GB
    - Trimming a whole disk does not take it back to factory settings.

Beyond TRIM

[Nellans et al. FusionIO 2010, Arpaci-Dusseau et al, HotStorage 2010]



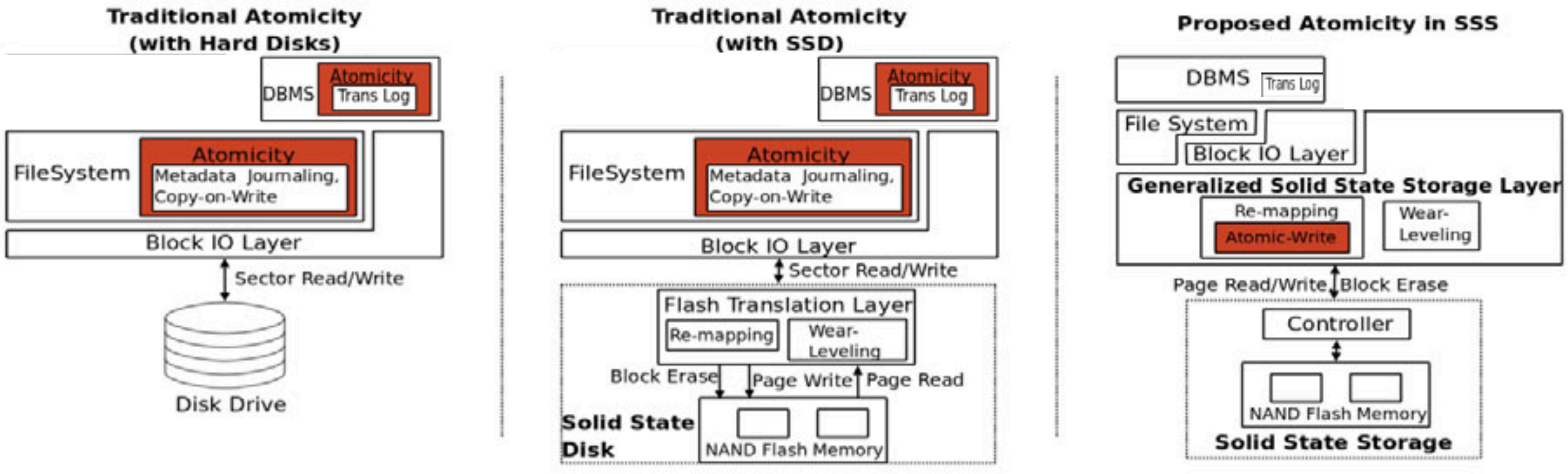
read(LBA) – write(LBA) – trim (LBA)

persistent_ trim(LBA)	Directive for VA block mapping
exists(LBA)	Query state of allocation
atomic_write(LBA's)	Atomically write multiple LBAs
nameless_write(data)	Return optimal LBA range

Slide courtesy of David Nellans, FusionIO, FMS'11

Atomic Writes

[Prabakharan et al, OSDI 2008; Ouyang et al, HPCA'11]



Problem: partial writes due to system failure during an in-place update
Solution: copy on write (InnoDB physiological logging + double write buffer);
atomic write at FTL level improve performance significantly (single write) and reduce DBMS complexity; it also limits concurrency.

Slide courtesy of , Gary Orenstein Fusion IO

Co-design: What's next?

- No in-place updates, do we still need WAL?
- If we reconsider physiological logging, do we still need page-based IOs? Do we still need the same representation in memory and on disk?
- Can we leverage prioritized IOs to improve a form of predictability?
- What does extent-based data allocation buy us?
- How to efficiently deal with arrays of flash devices?

Take-away Point # 1

- Flash devices are here to stay

  Towards high-performance, energy proportionality

- The key issue is to improve predictability AND performance

As long as flash devices hide flash chip constraints to support any types of IOs, performance characteristics will remain opaque.

Take-away Point # 2

- Need to revisit DBMS design decisions stemming from hard drive characteristics
- Need to revisit strict layering between DBMS, OS and FTL

The complexity of flash devices should not be abstracted away if it results in unpredictable and suboptimal performance.

Take-away Point # 3

- Lot of work in DB community based on FD assumptions
- The co-design train has left the station. FusionIO and Oracle are leading the way.

There is an opportunity for the DB community to stop running after the technology, and influence the upcoming developments of flash devices