RocksDB

C++ 编写

为快速存储介质而生

嵌入式持久化 key-value 存储

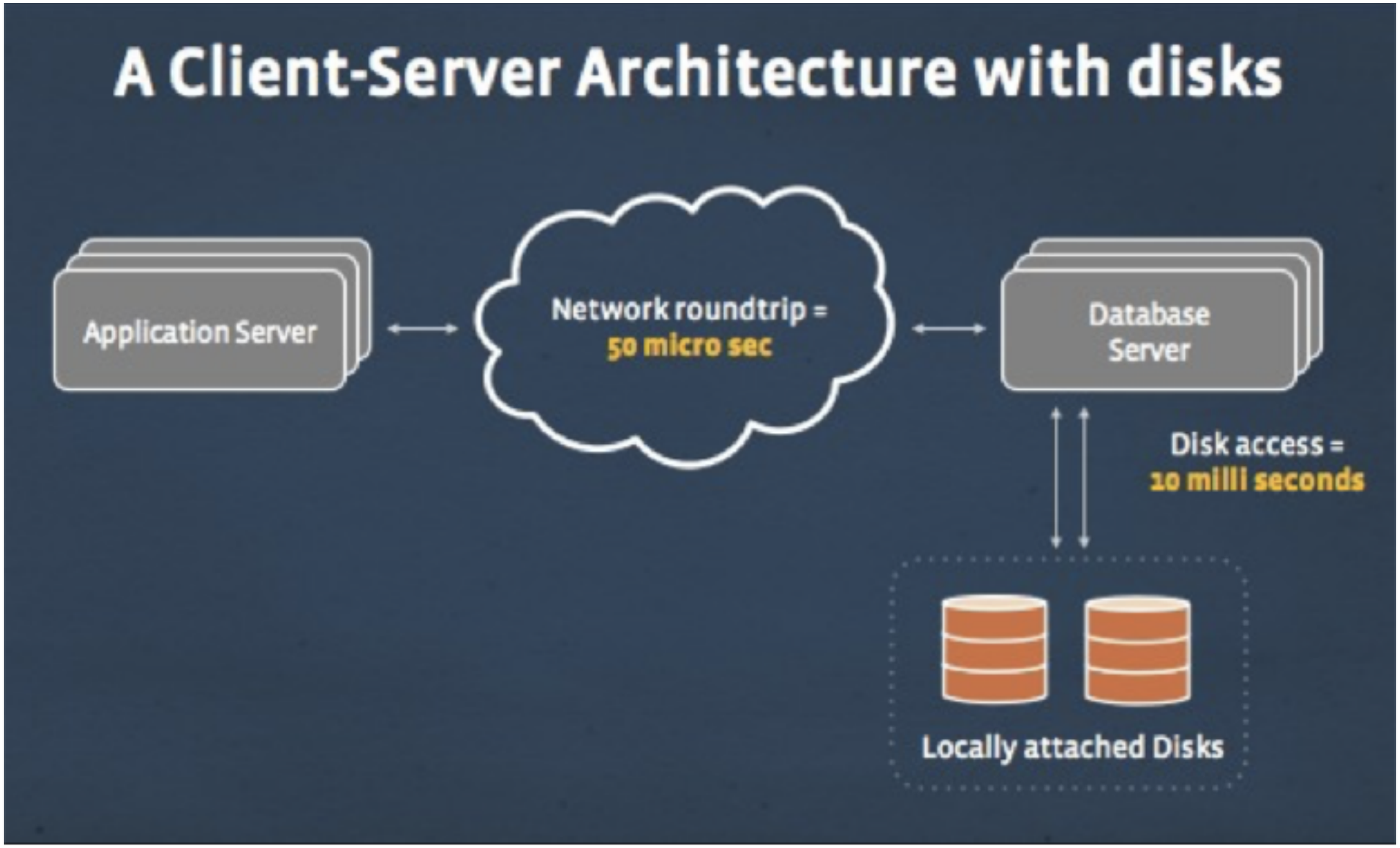
key 按序排列（排序比较方法可自定义）

key-value 可为任意大小的二进制字节流

最初的目的

充分发挥快速存储介质的全部读写潜力

内存（RAM）& 闪存（Flash，包括 SSD）

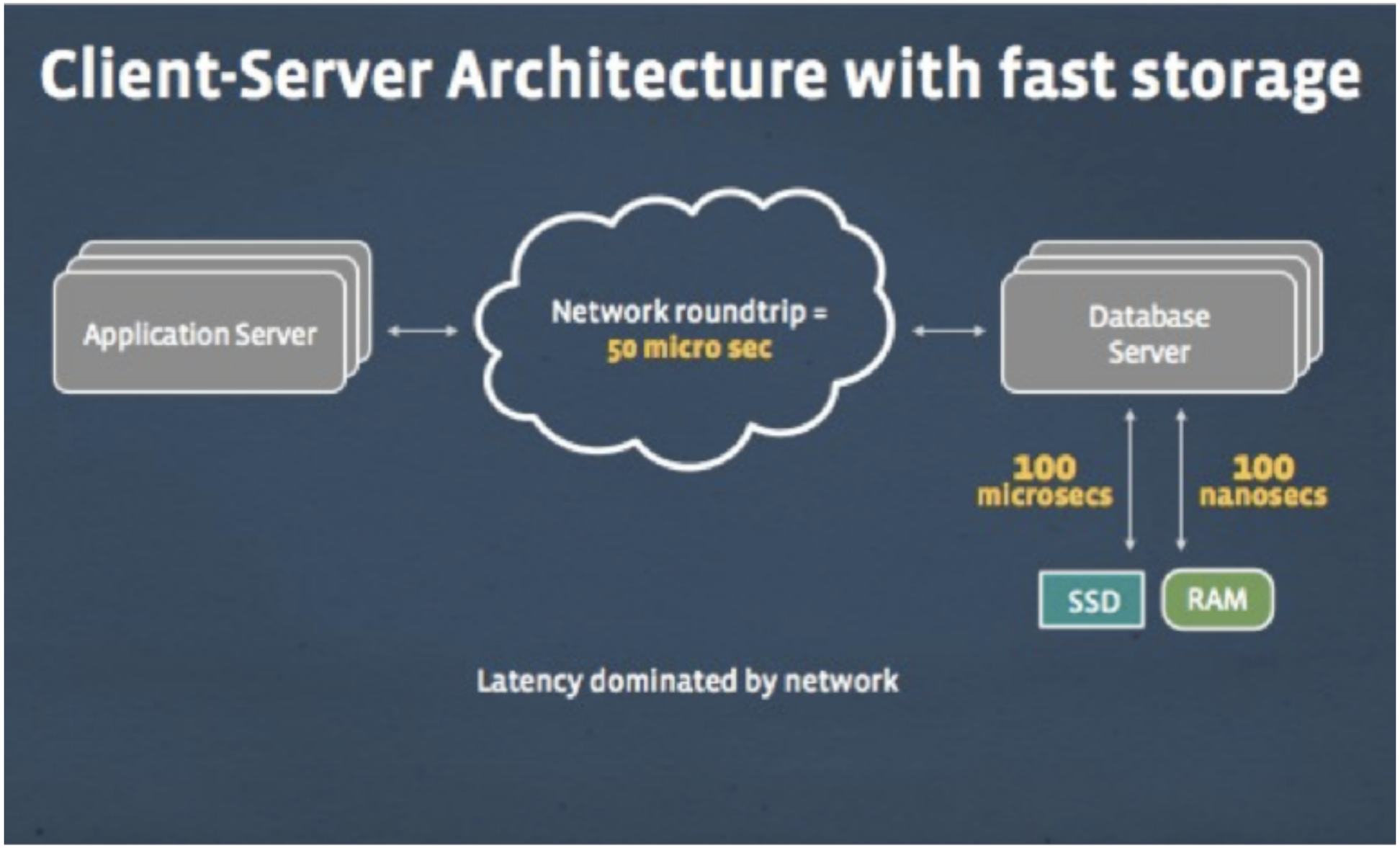


1milli sec = 0.001 sec 毫秒

1micro sec = 0.000,001 sec 微妙

1nano sec = 0.000,000,001 sec 纳秒

RTT / (IO Time + RTT) < 0.5%



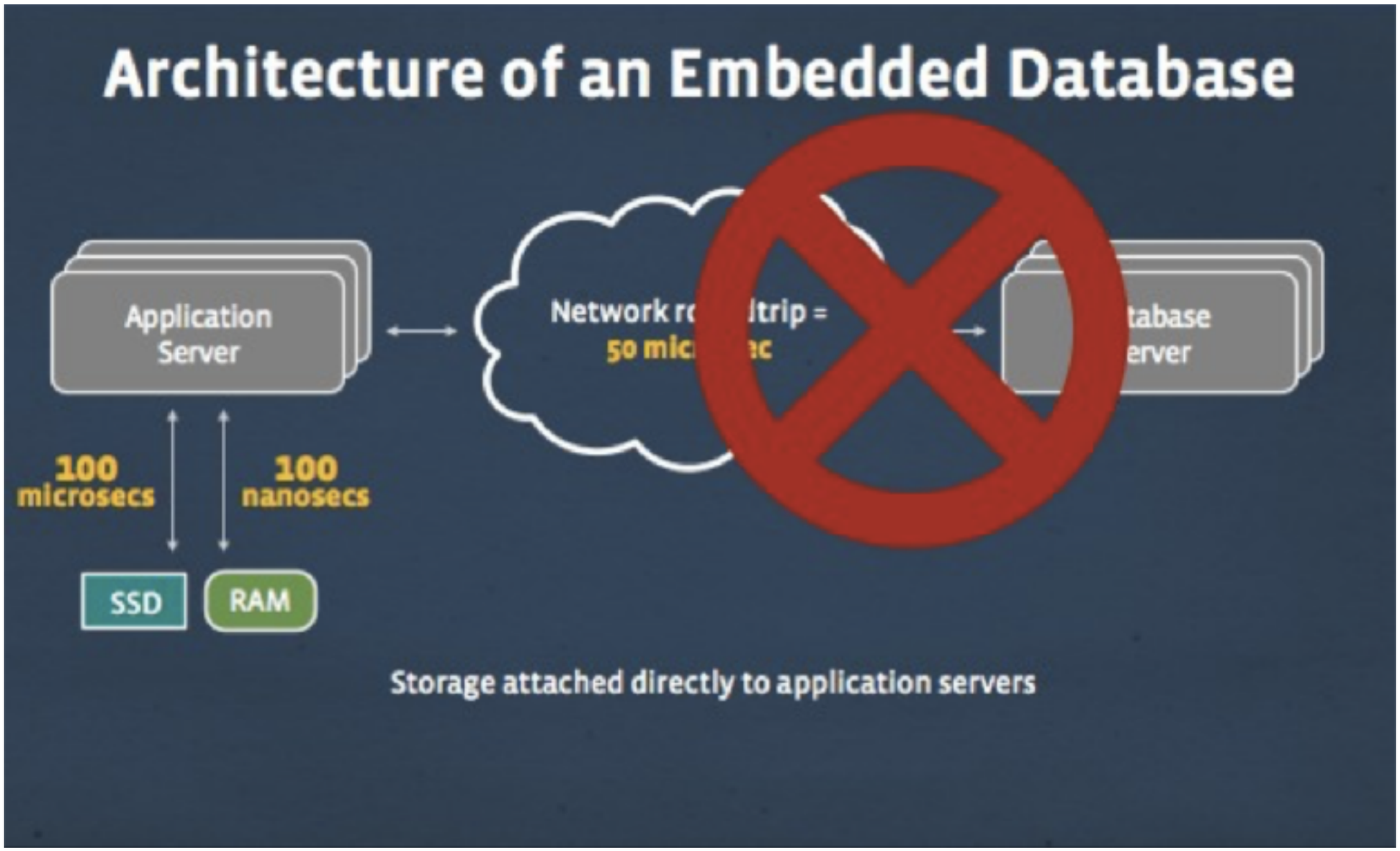
1milli sec = 0.001 sec 毫秒

1micro sec = 0.000,001 sec 微妙

1nano sec = 0.000,000,001 sec 纳秒

需硬盘 IO：RTT / (SSD IO Time + RTT) > 33%

内存命中：RTT / (RAM IO Time + RTT) > 98%



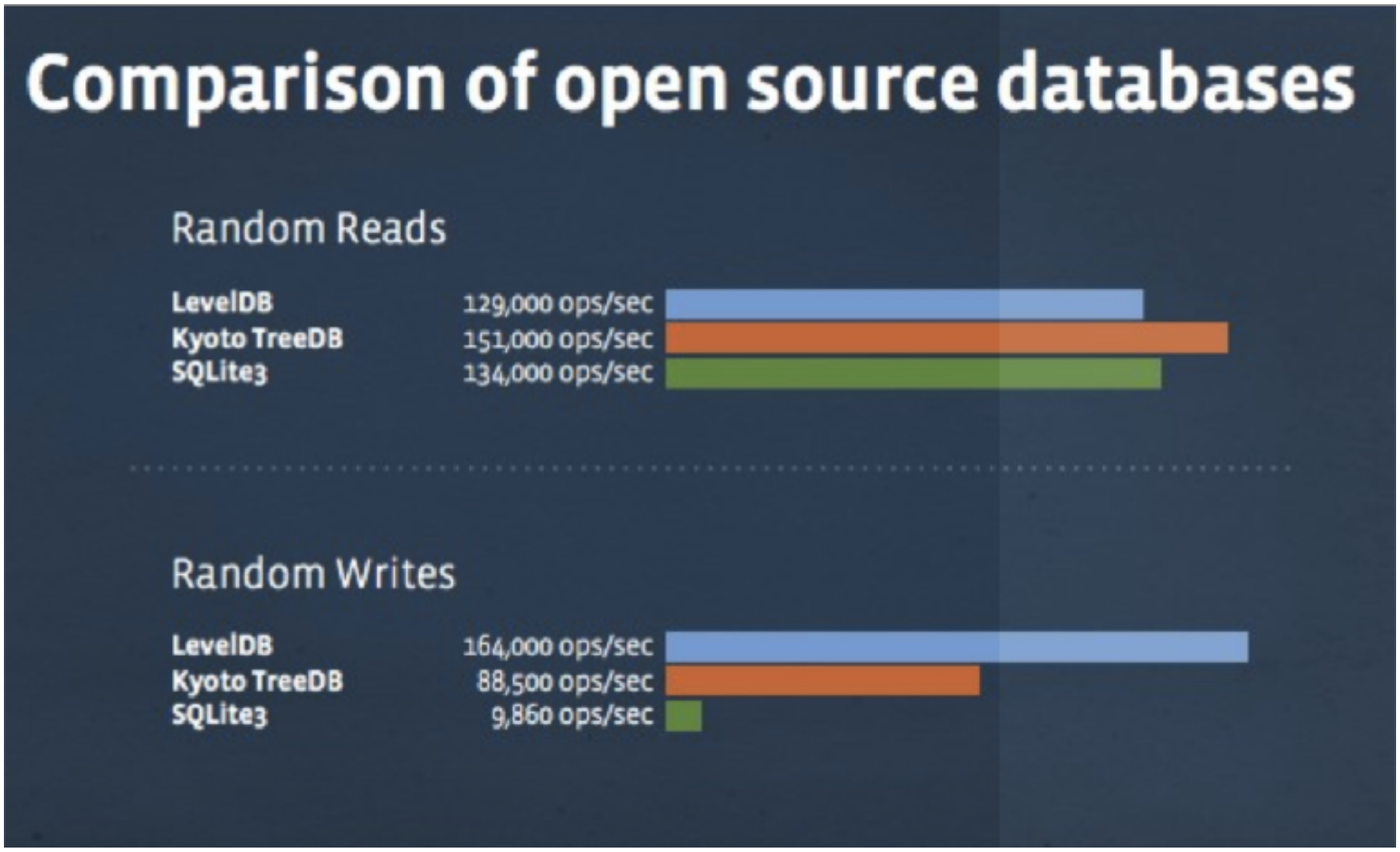
1milli sec = 0.001 sec 毫秒

1micro sec = 0.000,001 sec 微妙

1nano sec = 0.000,000,001 sec 纳秒

需硬盘 IO：减少 33% 时延

内存命中：减少 98% 时延



LevelDB（Google）

基于 **LSM** 树的文件组织方式

“We chose levelDB because of its high write rate, but then we discovered that its write rate was actually pretty low, so we fixed it.”

Facebook

从 LevelDB 继承

- 基本的 DB 操作、双向向迭代器（bi-directional iterators）
- 基于 **LSM 树** 实现 → 优异的写入性能

对 LevelDB 改进

- 更快的写入
- 更少的停顿（stall）
- 更小的写入放大（write amplification）
- 读取操作的优化
- ……

Adaptable 可适应

- DB 引擎：MyRocks（MySQL on RocksDB）
- 应用缓存：EVCache（RocksDB as L2, Netflix）
- 应用存储：TiDB（TiKV on RocksDB, PingCAP）
- 应用存储：pika（for Redis, 360）
- ……

快速写入，减少停顿

- LevelDB：使用单核 CPU 处理，合并压缩数据时无法同时读写
- RocksDB：充分利用多核 CPU，多线程同时读写和合并压缩

LSM Tree

Log-Structured Merge-Tree

1996

Log-Structured Merge-Tree 论文发表

O’Neil et al.

Jeff Dean

- Jeff 出生于1969年12月31日的下午11点48分，然后他花了整整12分钟的时间实现了他的第一个计时器。

- Jeff 发送以太网封包从不会发生冲突，因为其他封包都吓得逃回了网卡的缓冲区里。

2004

BigTable 论文发表，开发并投入使用

Jeff Dean et al.

2011

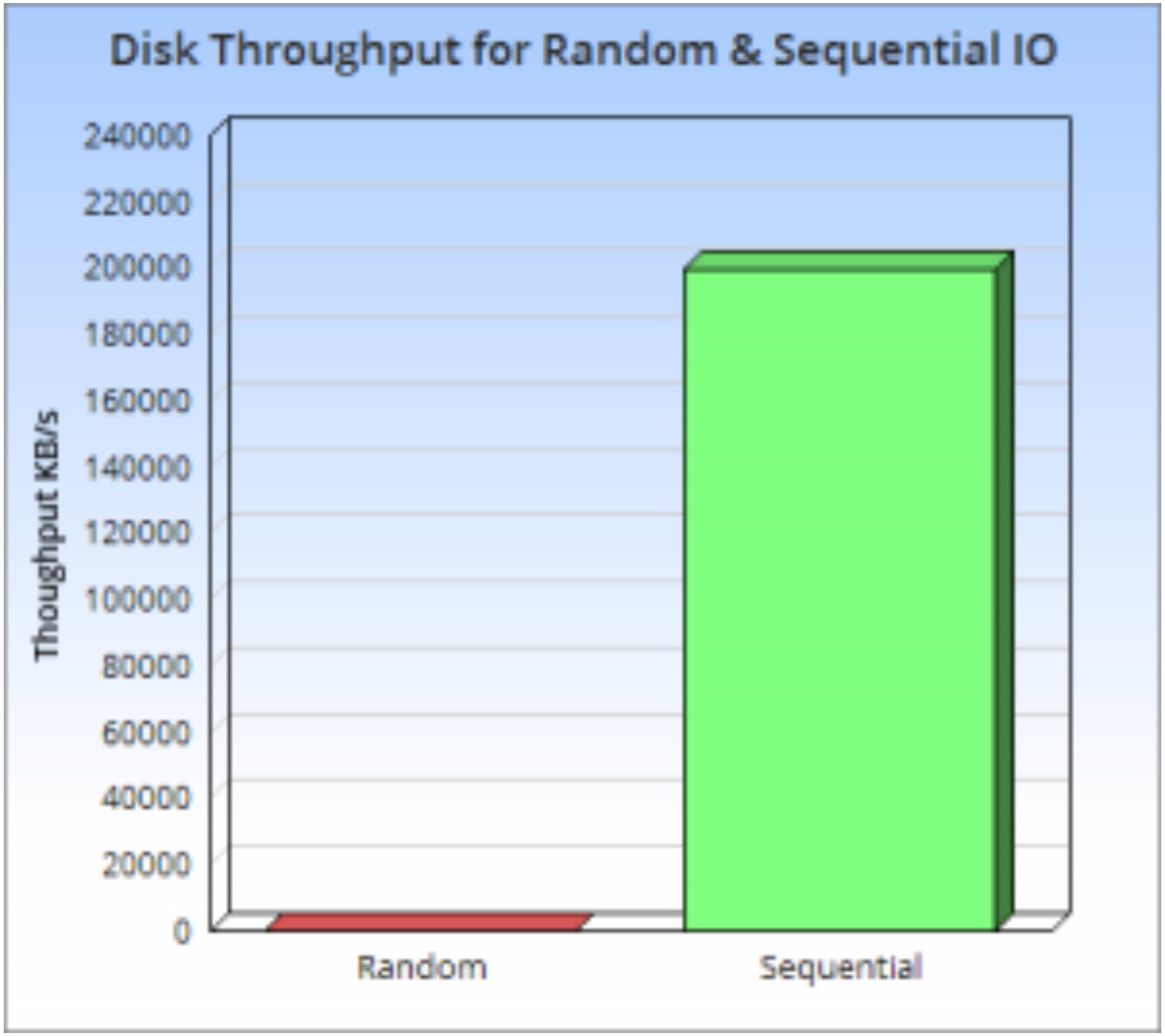
LevelDB 开发

Jeff Dean et al.

基于 LSM 树

2004 BigTable / 2008 HBase / 2009 MongoDB

2011 LevelDB / 2012 RocksDB / SQLite4


Disk Throughput for Random & Sequential IO
Thoughput KB/s
240000
220000
200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
Random
Sequential

随机 I/O << 顺序 I/O

~~Random I/O~~

Sequential I/O

随机读写 → 顺序读写

Results | Messages

	Order...	ProdName	RegionName	FiscalQtr	NetSales
1	111	Printer	East	Q1-2016	7956
2	112	Scanner	West	Q1-2016	5763
3	113	Television	North	Q1-2016	7301
4	114	Laptop	South	Q1-2016	5719
5	115	Printer	East	Q2-2016	5742
6	116	Scanner	West	Q2-2016	5873
7	117	Television	North	Q2-2016	1361
8	118	Laptop	South	Q2-2016	3659
9	119	Printer	East	Q3-2016	6869
10	120	Scanner	West	Q3-2016	1736
11	121	Television	North	Q3-2016	2061
12	122	Laptop	South	Q3-2016	8253
13	123	Printer	East	Q4-2016	6054
14	124	Scanner	West	Q4-2016	8550
15	125	Television	North	Q4-2016	2073
16	126	Laptop	South	Q4-2016	9961

Update-in-place

原地修改

1. .....
2. ~~set A to 123~~ (invalid record)
3. ~~set B to 456~~ (invalid record)
4. set C to 789
5. set A to 666
6. delete B

~~Update-in-place~~

Append-only

原地修改 → 只添加（不修改）

取舍

写入：平均 & 最差复杂度 O(1)

牺牲读性能，提升写性能

倒序扫描

读取：平均 & 最差复杂度 O(N)

数据结构

内存：C0 → MemTable

SSD：C1 → SST

（Sorted String Table）

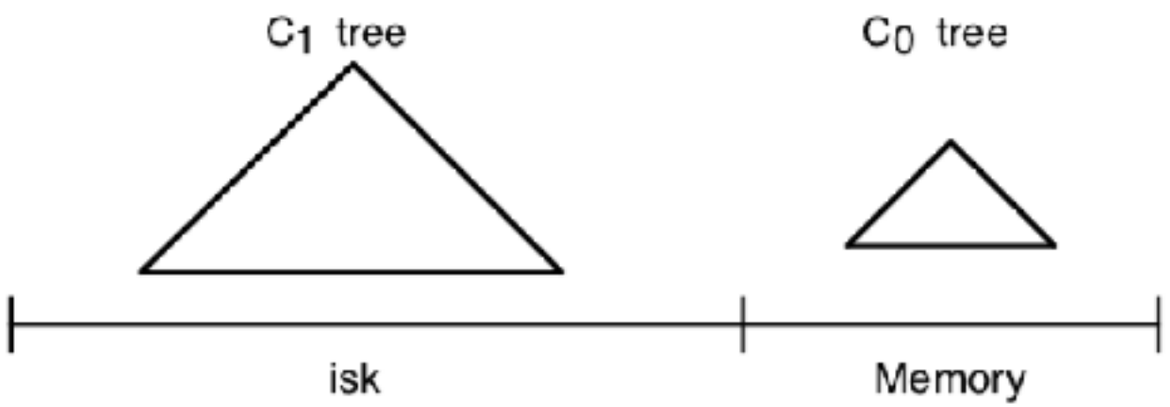
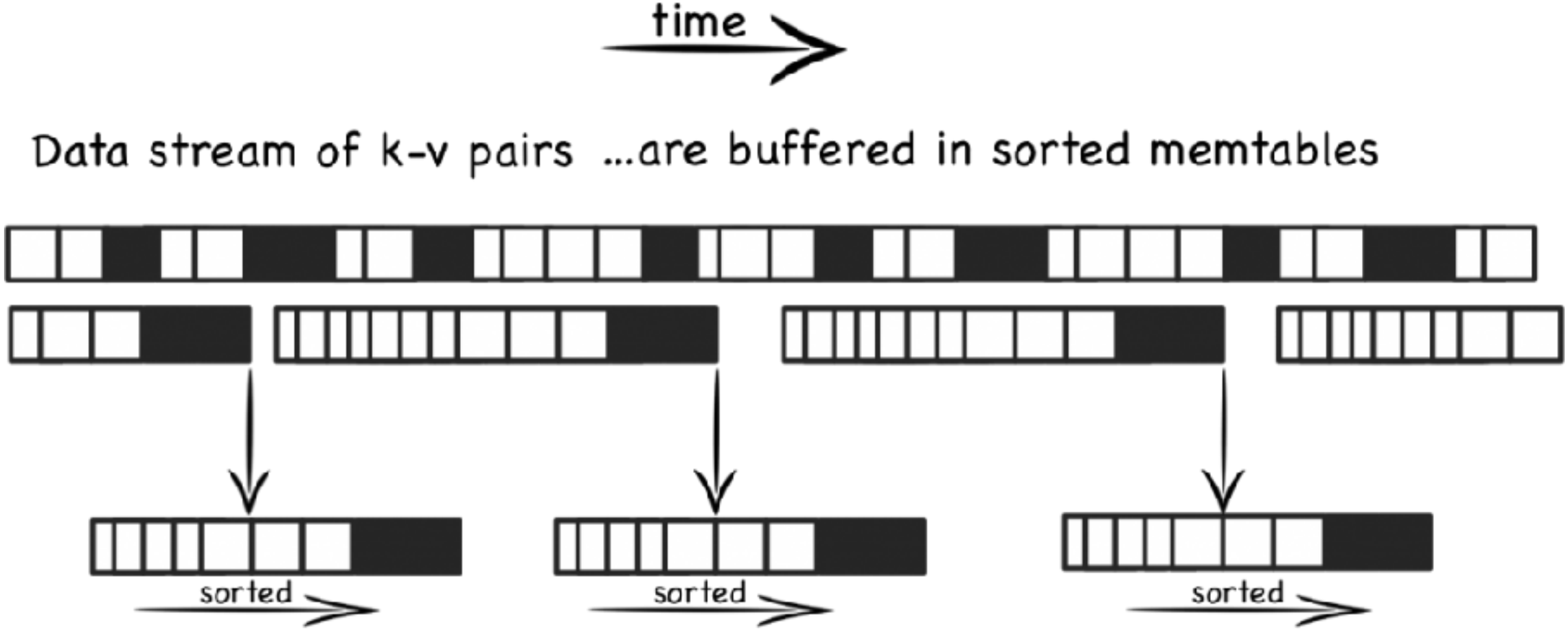
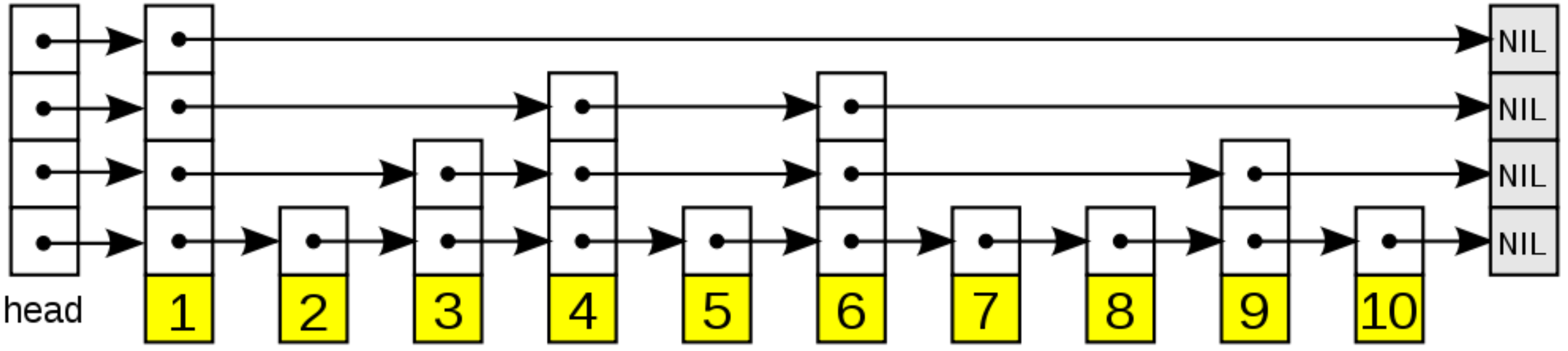


Figure 2.1. Schematic picture of an LSM-tree of two components

C0 C1

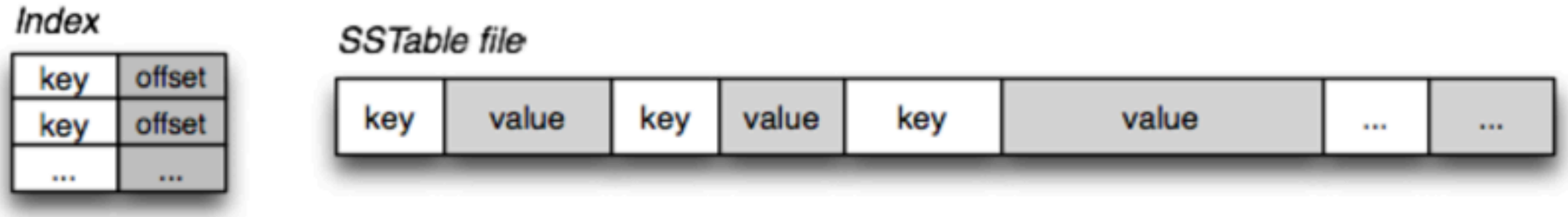


and periodically flushed to disk...forming a set of small, sorted files.



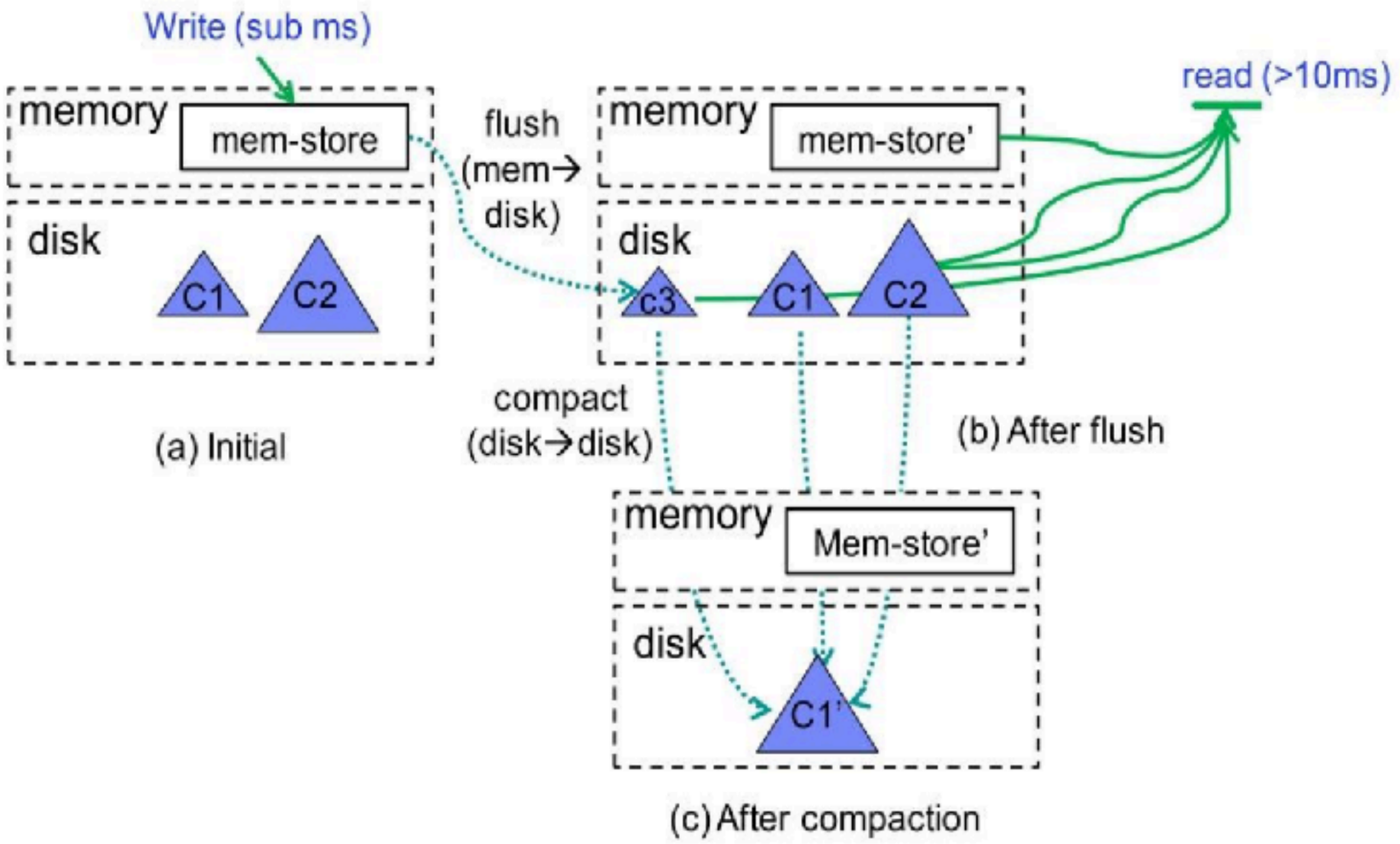
内存 MemTable：Skiplist 跳跃表

查询 O(log(N))

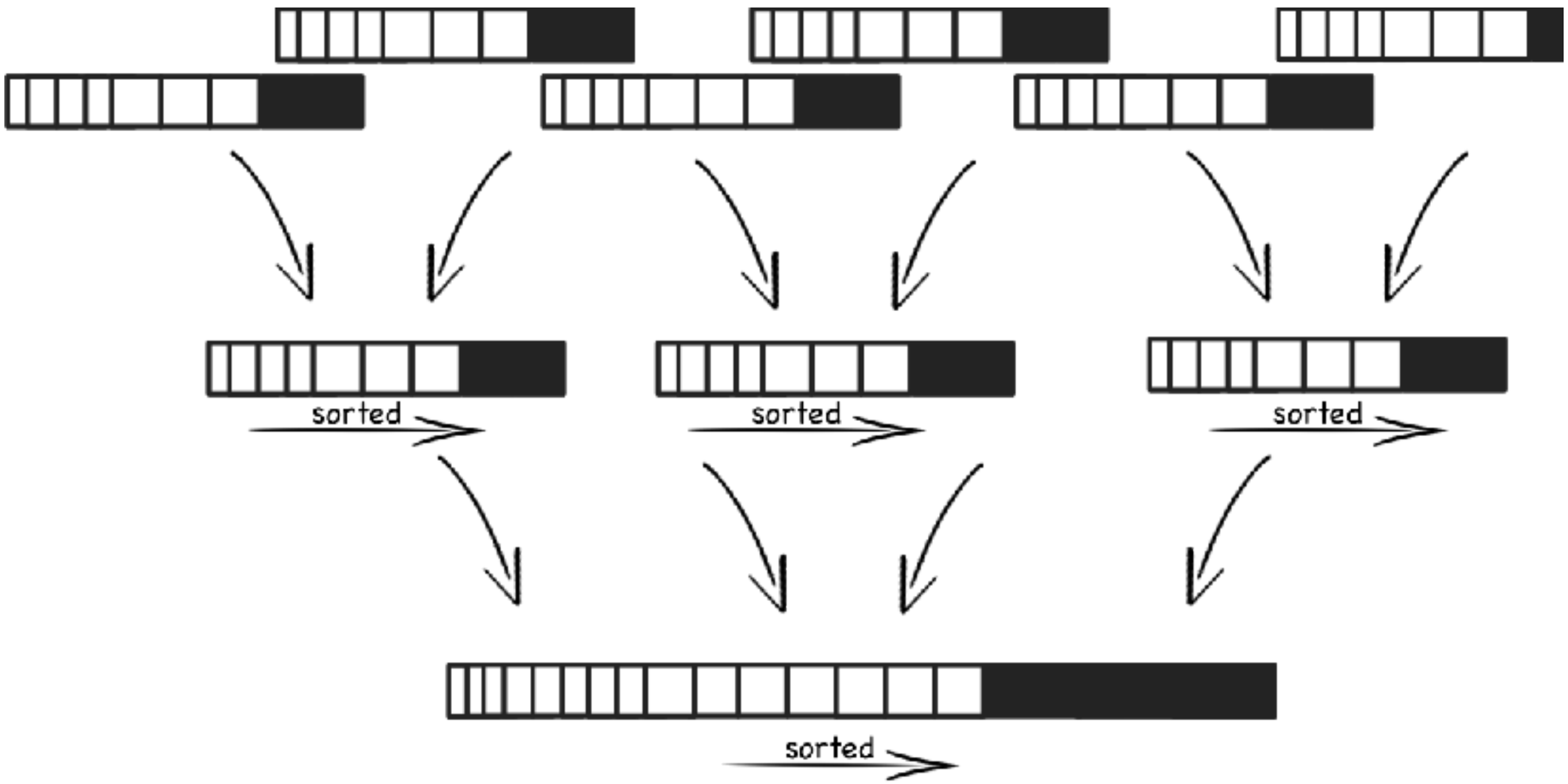


A "Sorted String Table" then is exactly what it sounds like, it is a file which contains a set of arbitrary, sorted key-value pairs inside. Duplicate keys are fine, there is no need for "padding" for keys or values, and keys and values are arbitrary blobs. Read in the entire file sequentially and you have a sorted index. Optionally, if the file is very large, we can also prepend, or create a standalone `key:offset` index for fast access. **That's all an SSTable is: very simple, but also a very useful way to exchange large, sorted data segments.**

磁盘： Sorted String Table



MemTable & SST

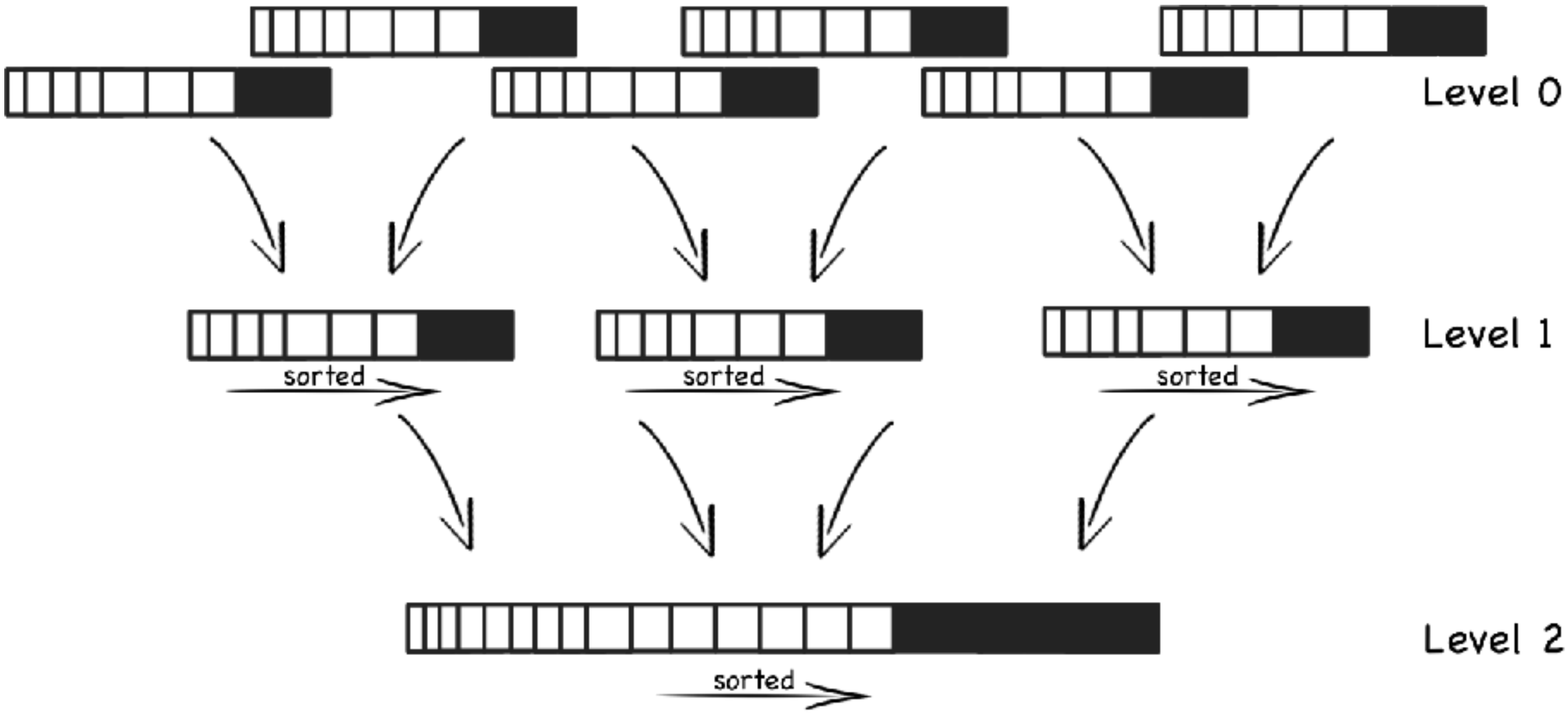


Compaction continues creating fewer, larger and larger files

二分法

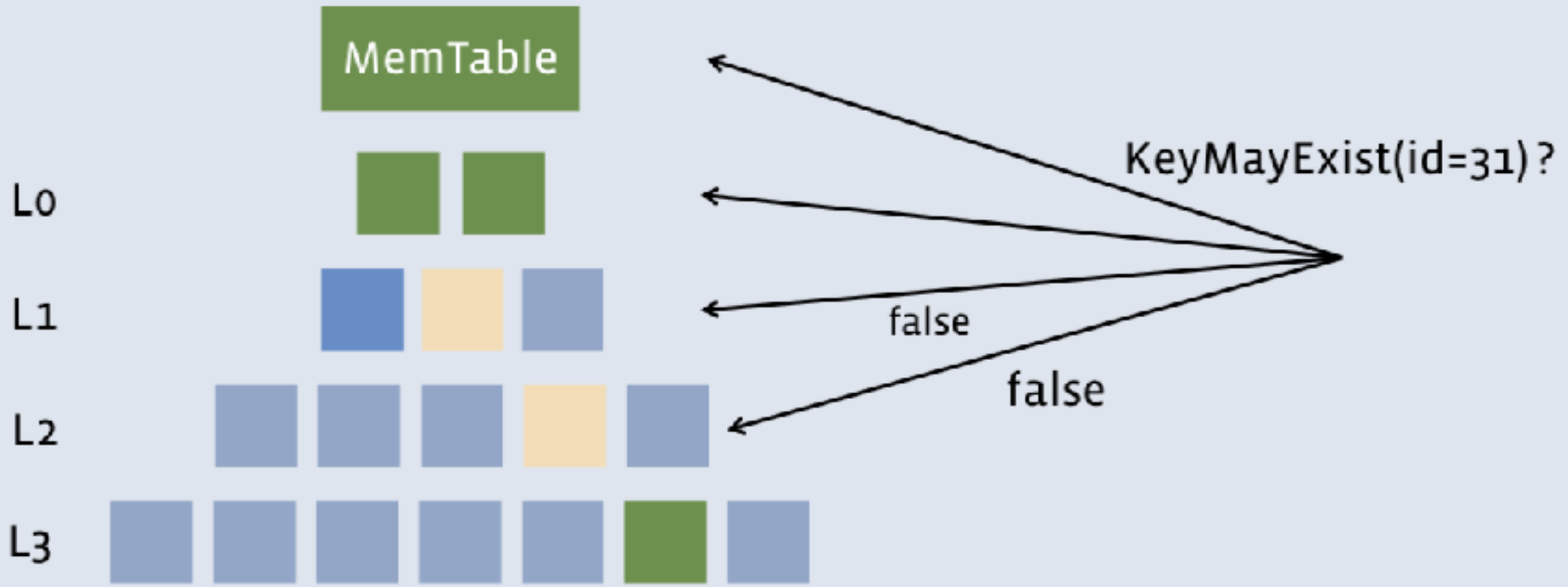
K 个文件

读取：平均 & 最差复杂度 O(K * log(N))



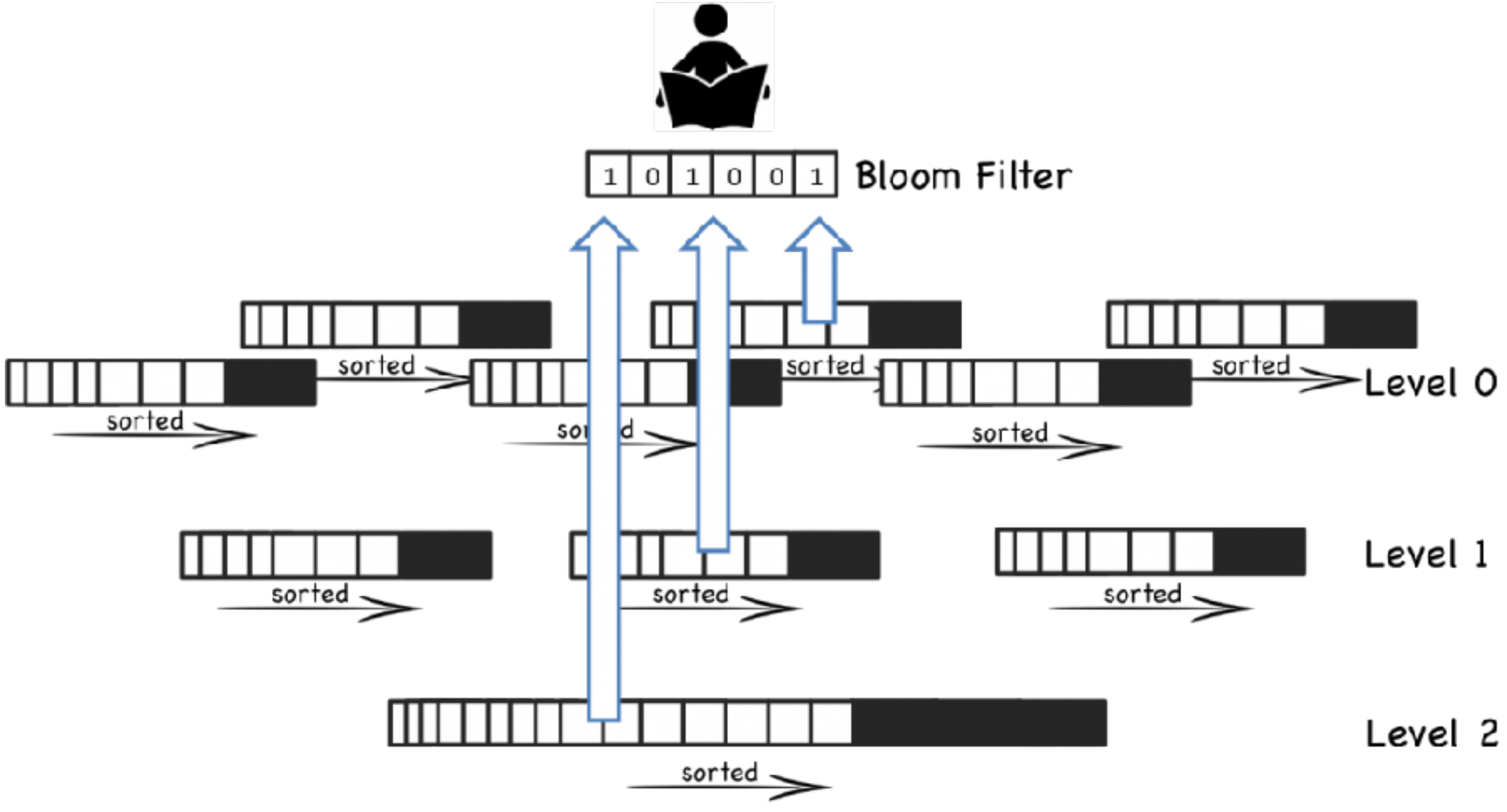
Compaction continues creating fewer, larger and larger files

Checking key may exist or not without reading data,
and skipping read i/o if it definitely does not exist



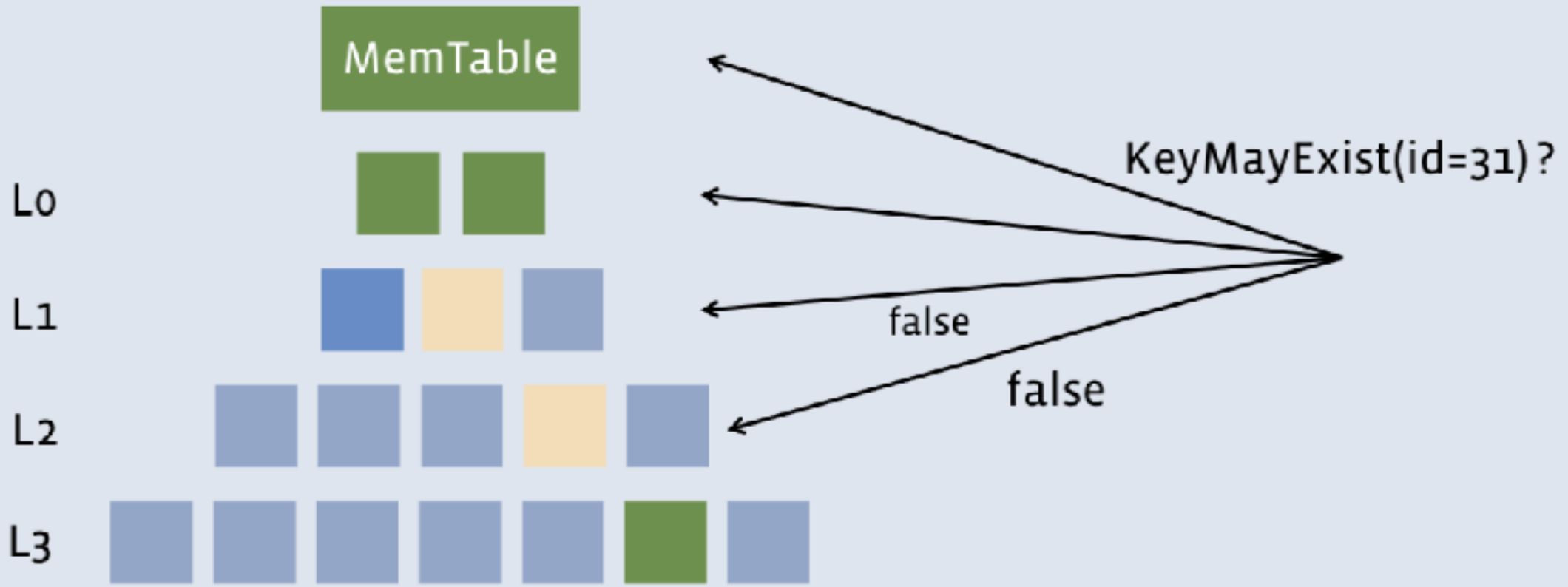
布隆过滤器

As elements of a record could be in any level all levels must be consulted. Thus bloom Filters are used to avoid files unnecessary reads.



布隆过滤器

Checking key may exist or not without reading data,
and skipping read i/o if it definitely does not exist



布隆过滤器

RocksDB 存储架构

内存： MemTable

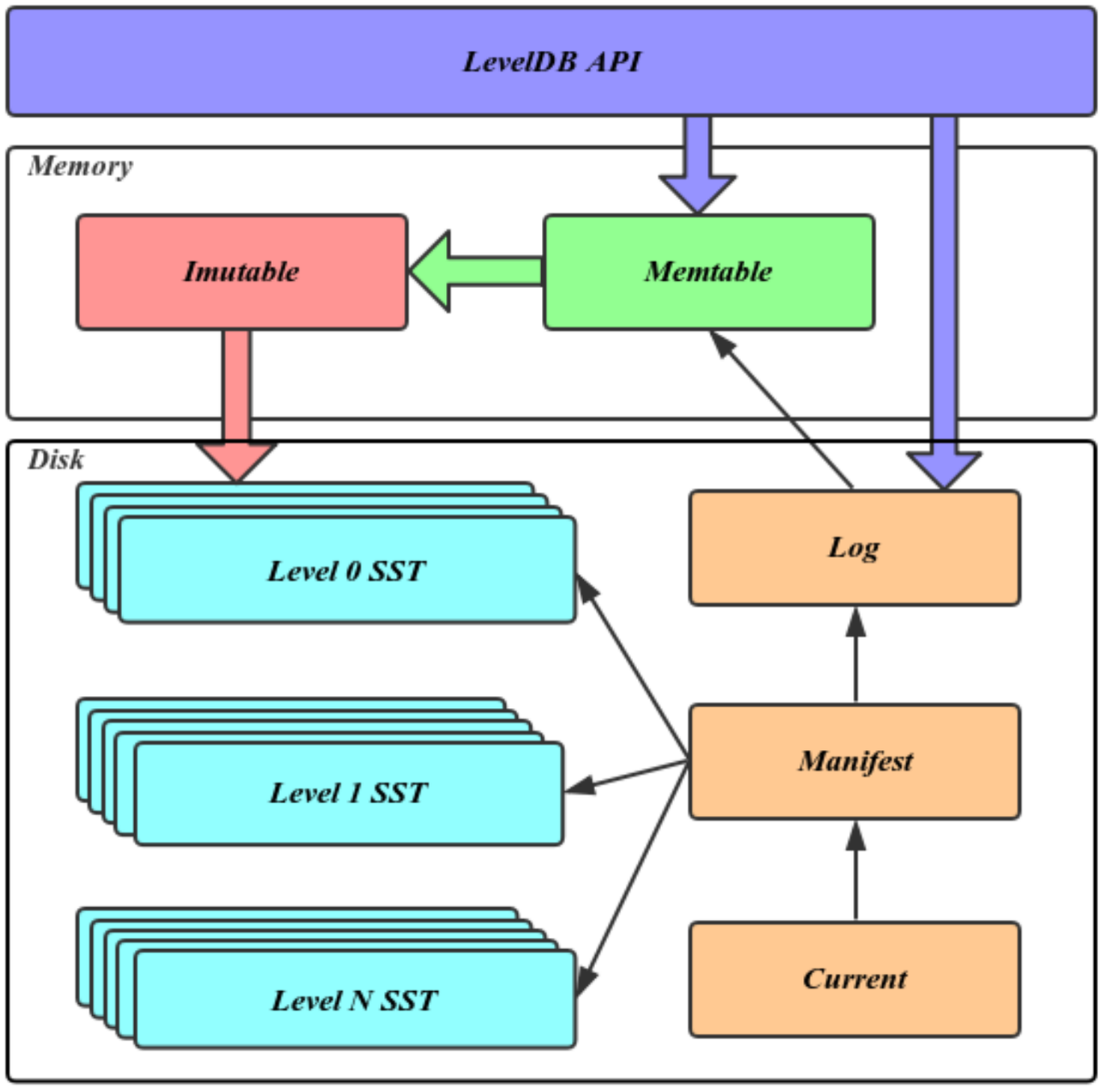
闪存： SST（String Sorted Table）

MemTable

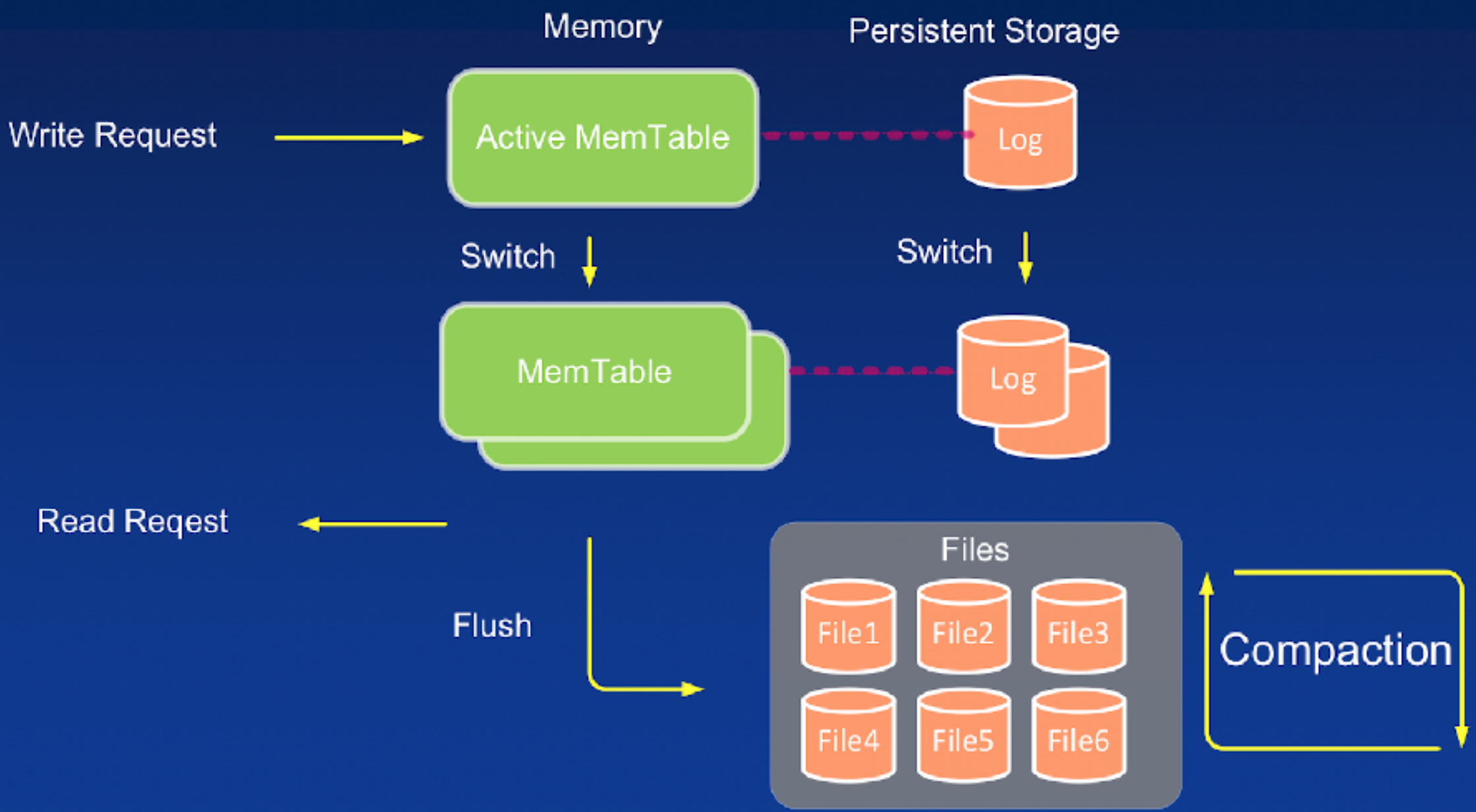
- Active
- Immutable

文件

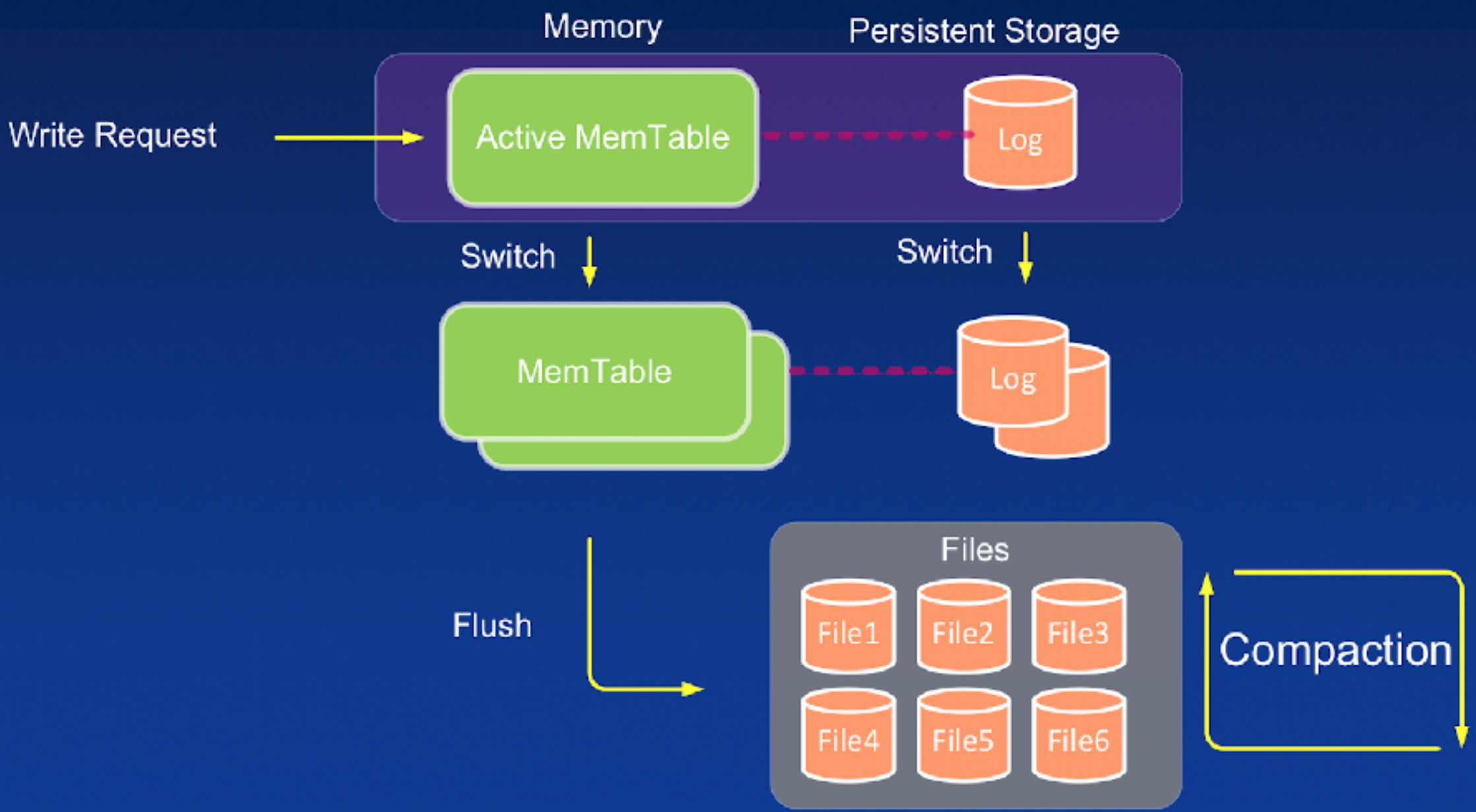
- WAL (Write Ahead Log)：持久化 & 错误恢复
- SST (String Sorted Table)


LevelDB API
Memory
Imutable
Memtable
Disk
Level 0 SST
Level 1 SST
Level N SST
Log
Manifest
Current

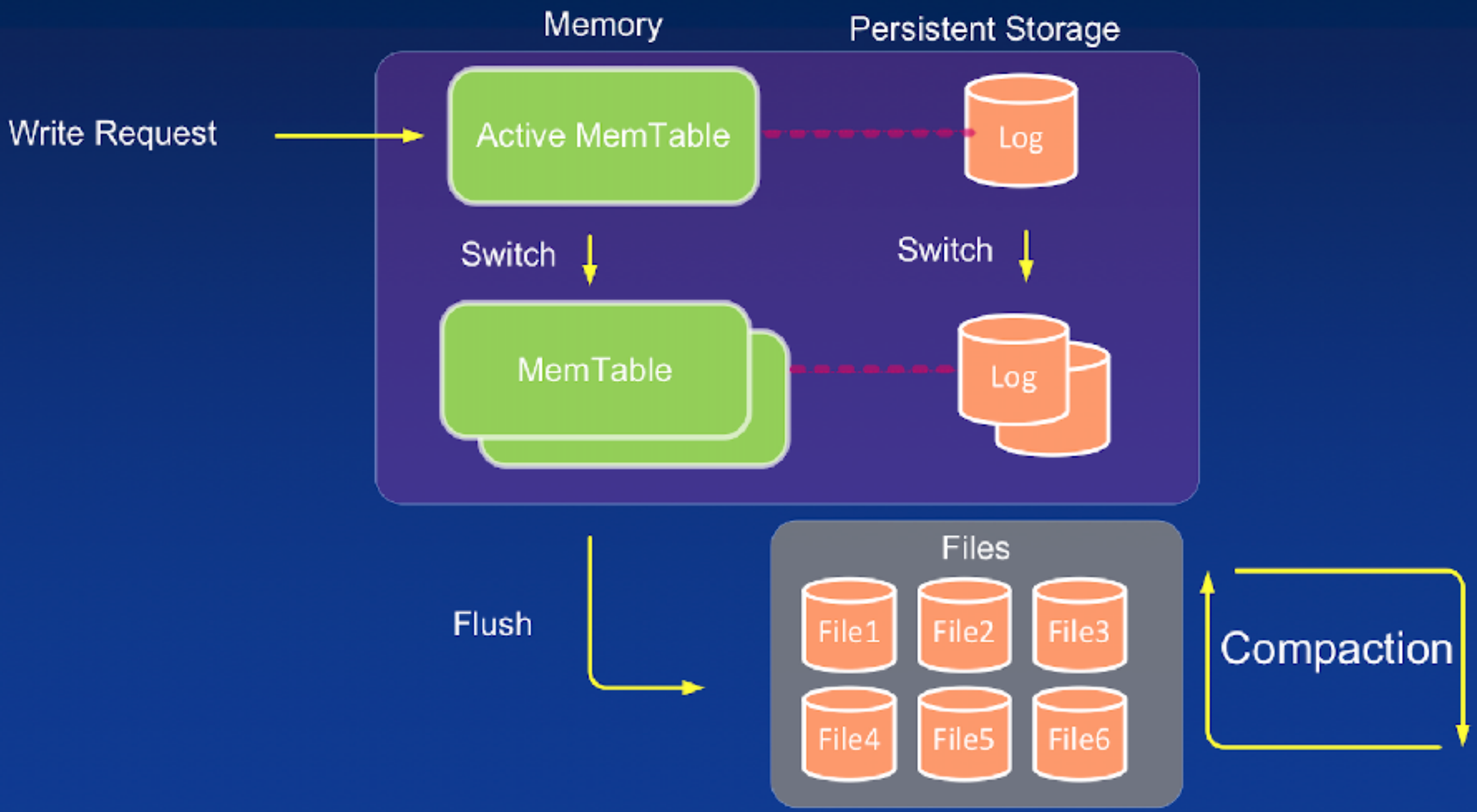
RocksDB Architecture



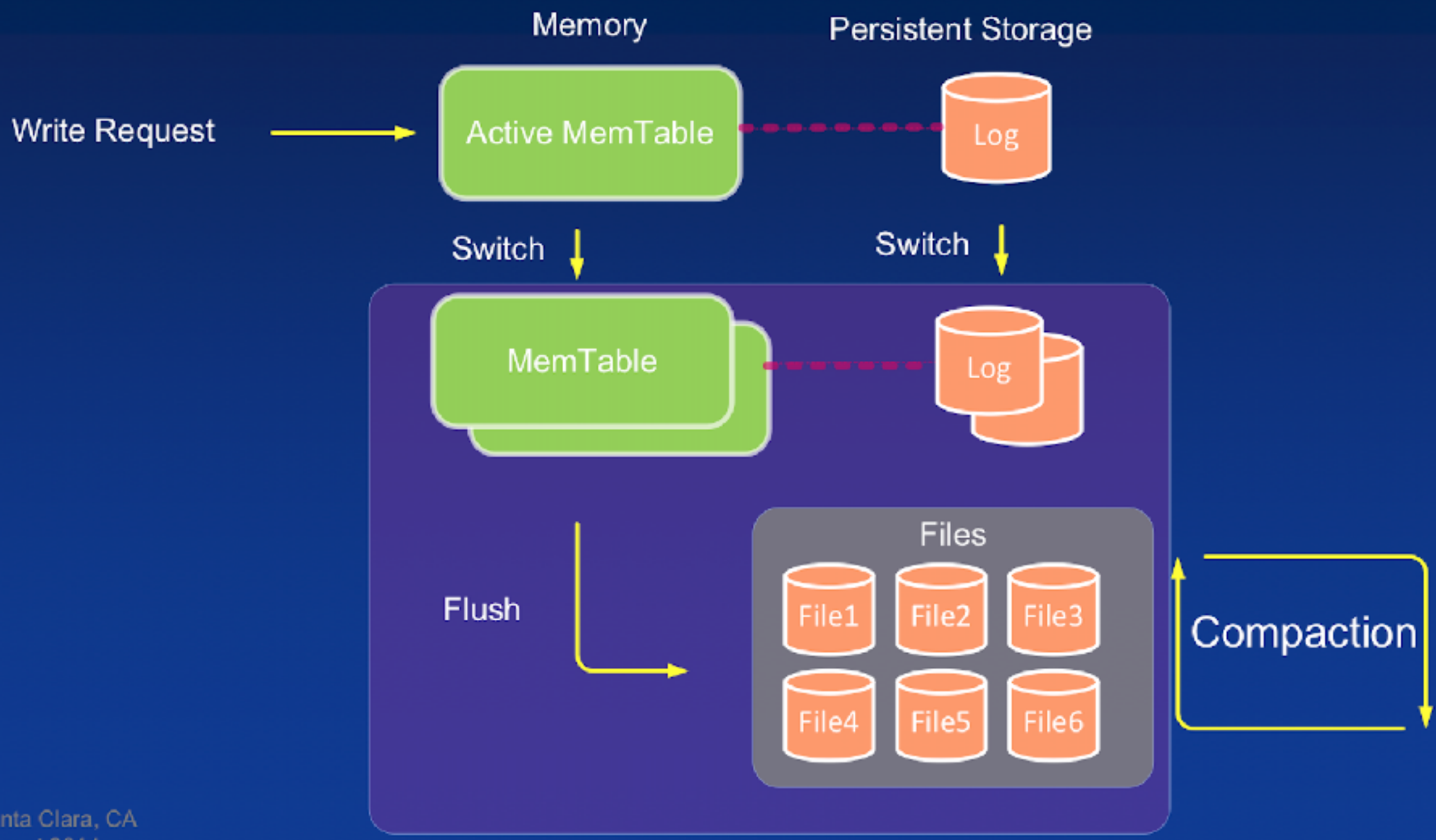
Write Path (1)


Memory
Persistent Storage
Write Request
Active MemTable
Log
Switch
Switch
MemTable
Log
Flush
Files
File1
File2
File3
File4
File5
File6
Compaction

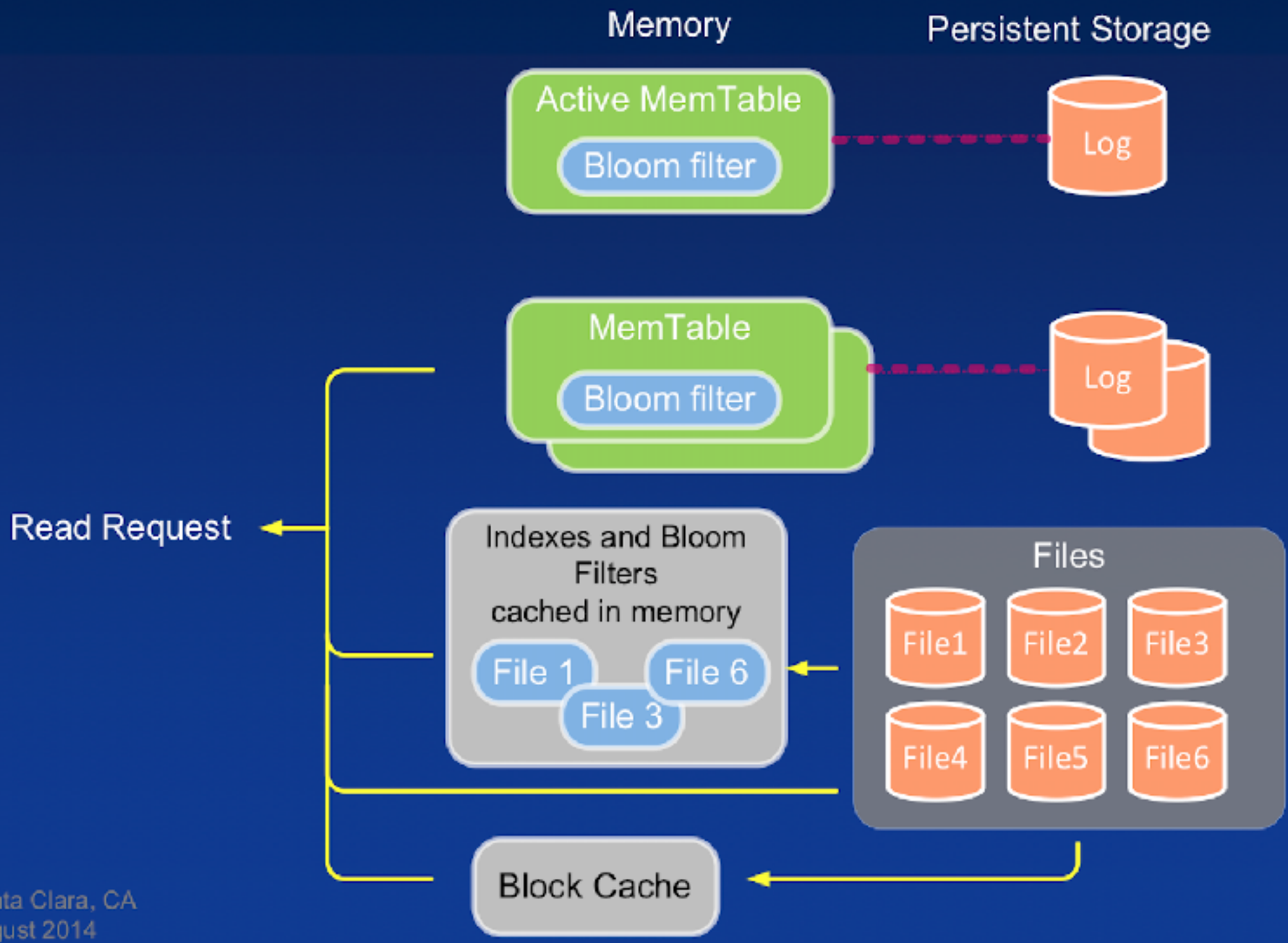
Write Path (2)


Memory
Persistent Storage
Write Request
Active MemTable
Log
Switch
Switch
MemTable
Log
Flush
Files
File1
File2
File3
File4
File5
File6
Compaction

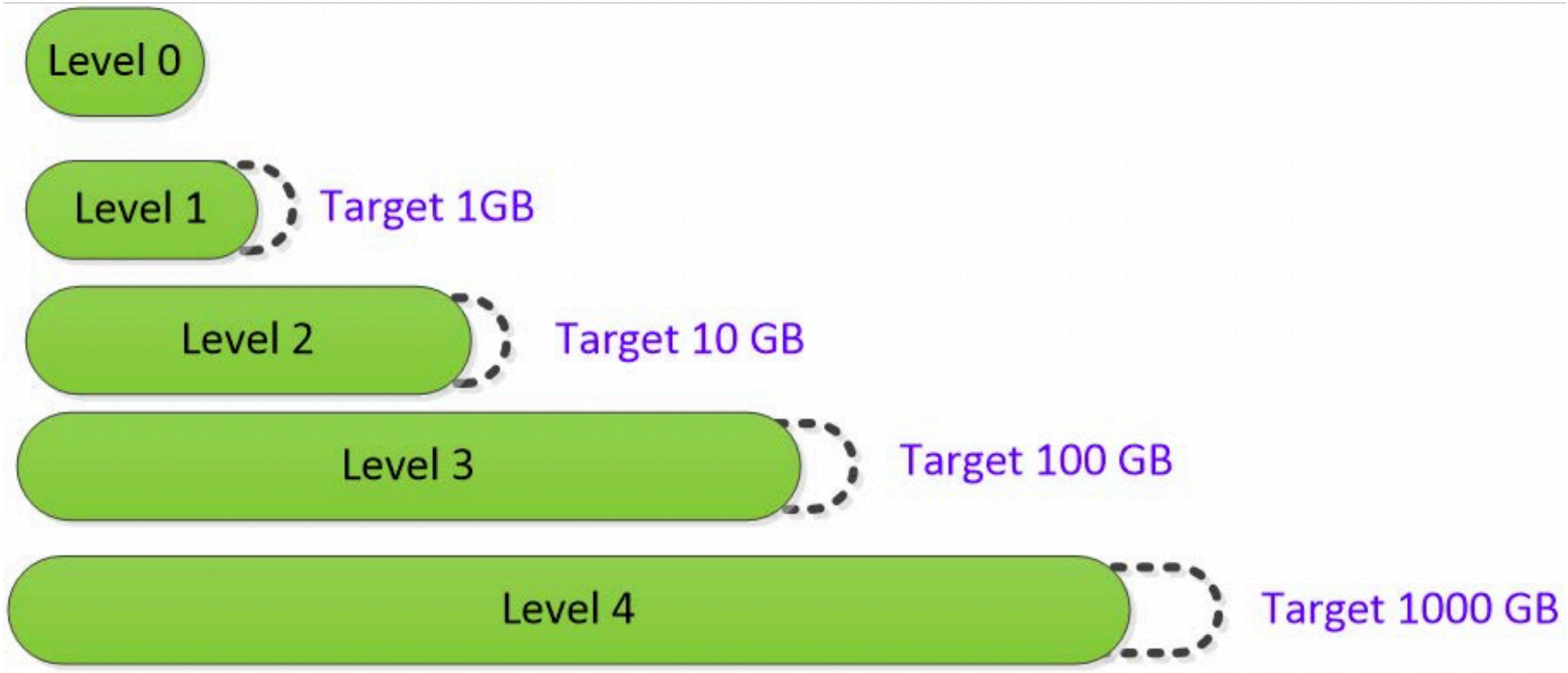
Write Path (3)


Memory
Persistent Storage
Write Request
Active MemTable
Log
Switch
Switch
MemTable
Log
Files
Flush
File1
File2
File3
File4
File5
File6
Compaction

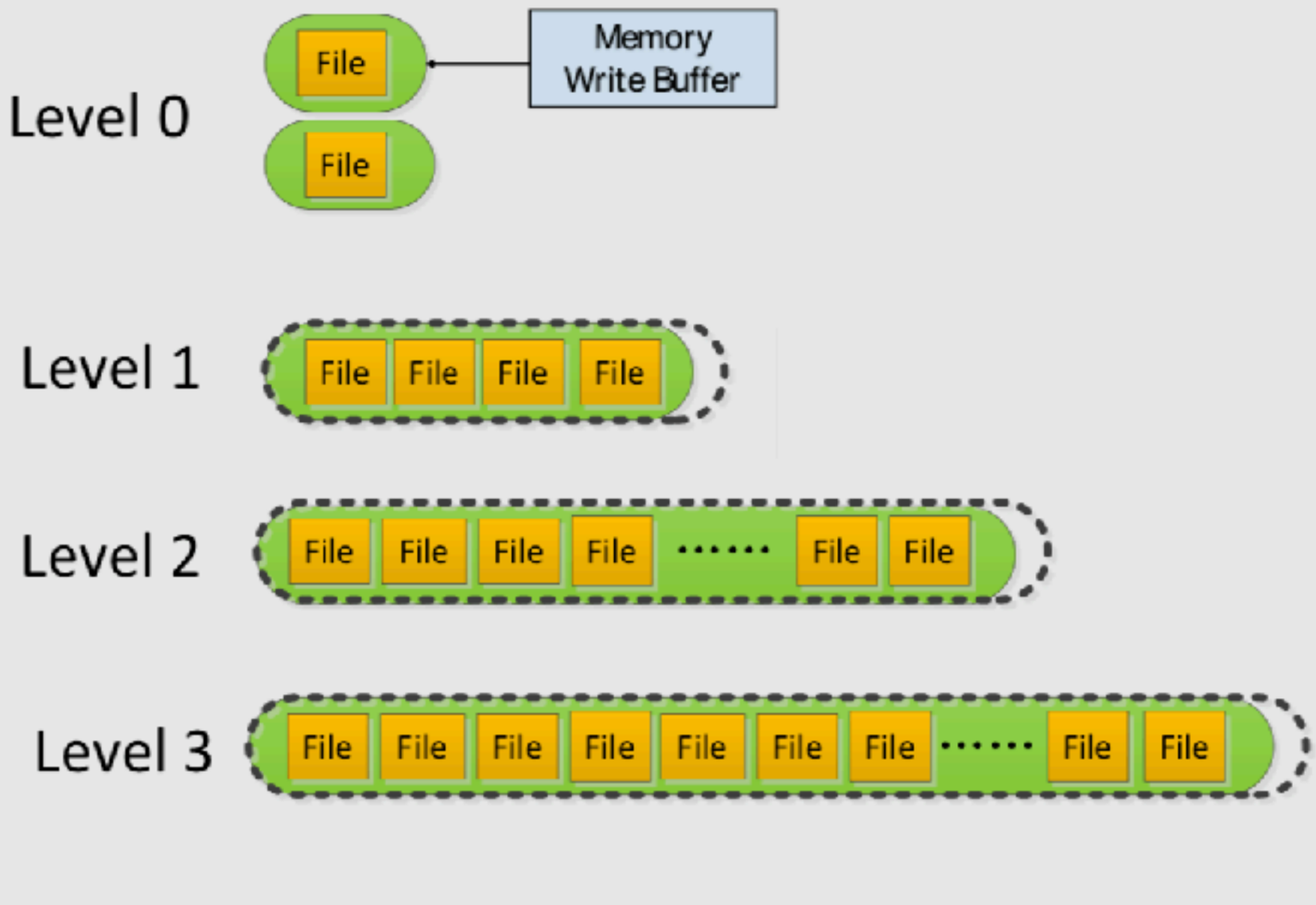
Read Path


Memory
Persistent Storage
Active MemTable
Bloom filter
Log
MemTable
Bloom filter
Log
Read Request
Indexes and Bloom Filters cached in memory
File 1
File 6
File 3
Files
File1
File2
File3
File4
File5
File6
Block Cache

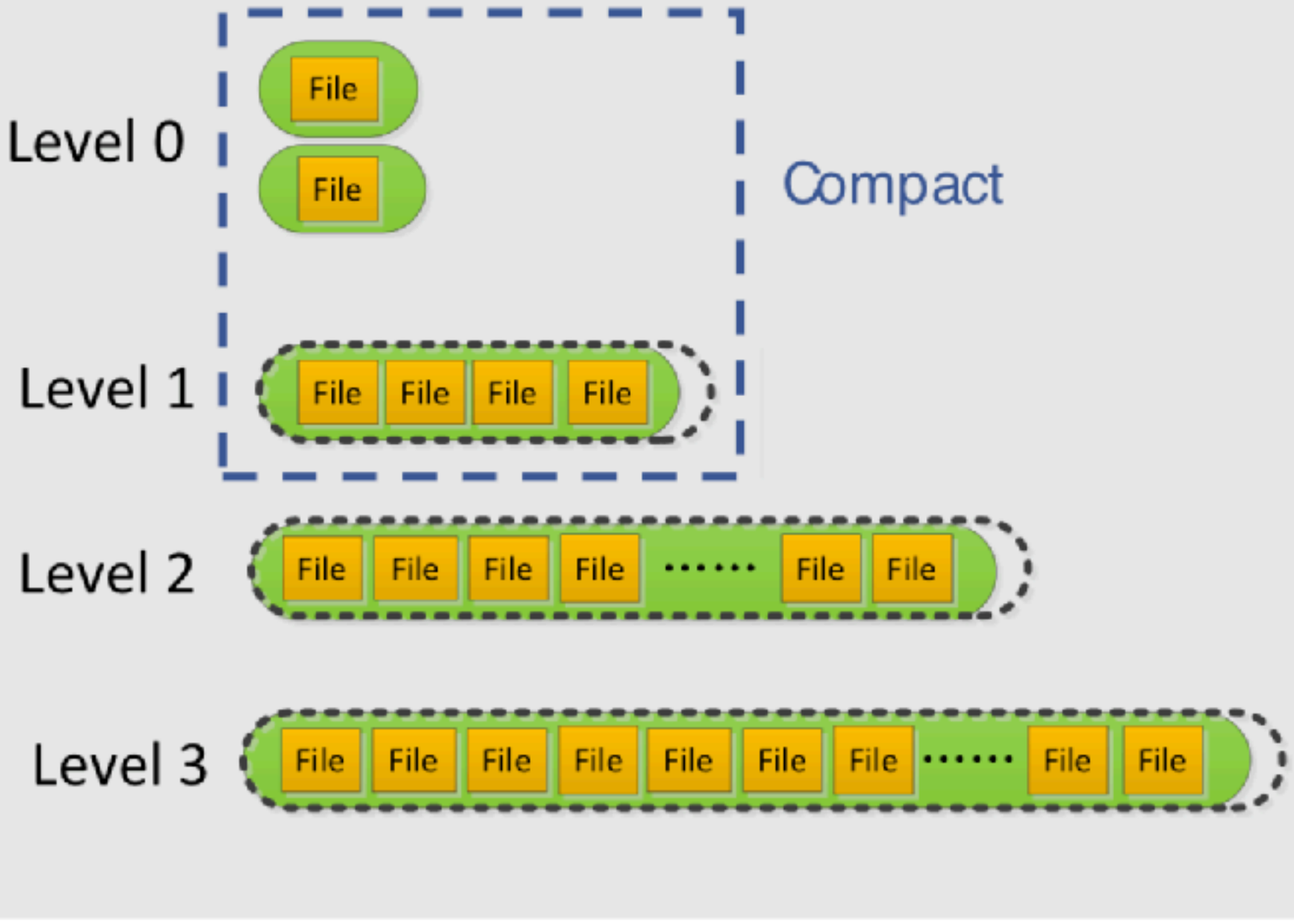
Level 压缩


Level 0
Level 1
Target 1GB
Level 2
Target 10 GB
Level 3
Target 100 GB
Level 4
Target 1000 GB

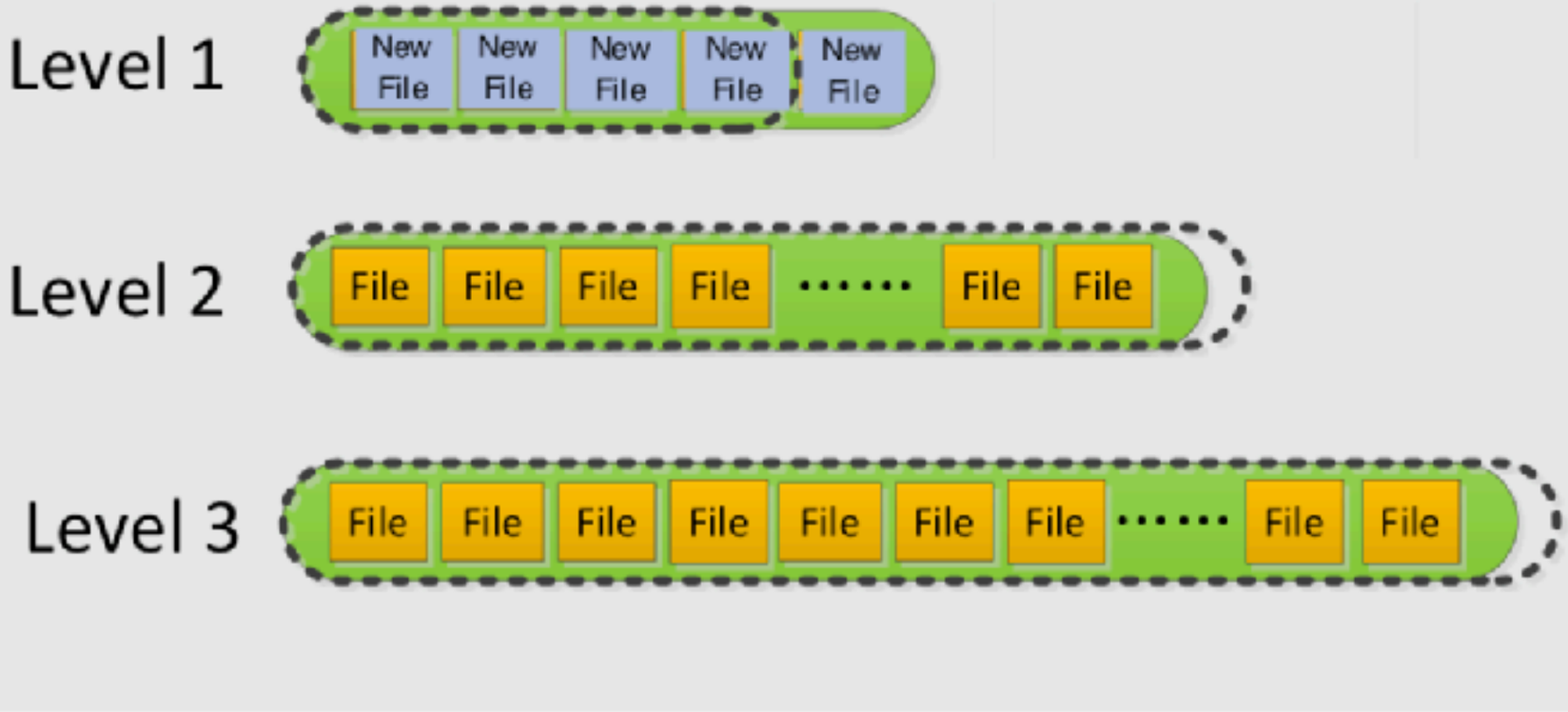
Level 压缩


Level 0
File
File
Memory
Write Buffer
Level 1
File
File
File
File
Level 2
File
File
File
File
.....
File
File
Level 3
File
File
File
File
File
File
File
.....
File
File

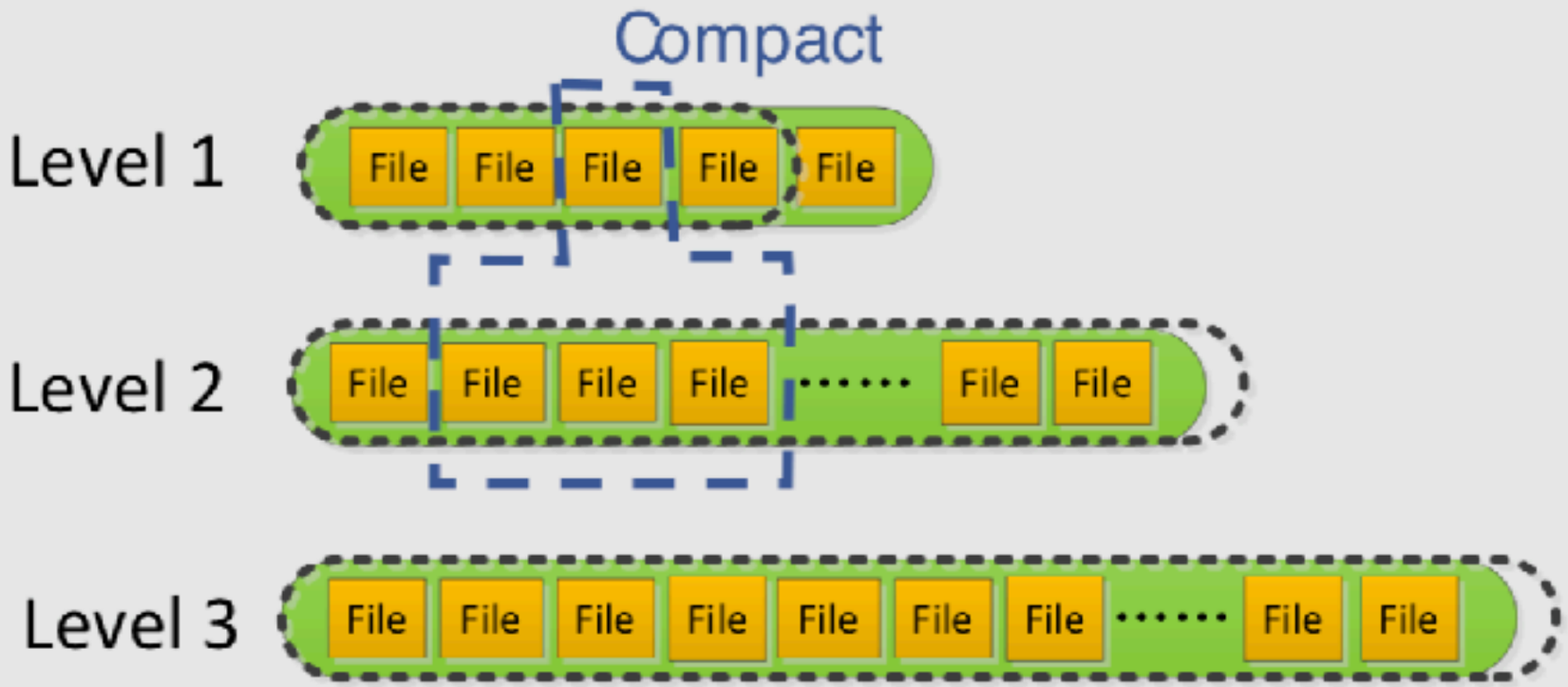
Level 压缩


Level 0
File
File
Compact
Level 1
File
File
File
File
Level 2
File
File
File
File
.....
File
File
Level 3
File
File
File
File
File
File
File
.....
File
File

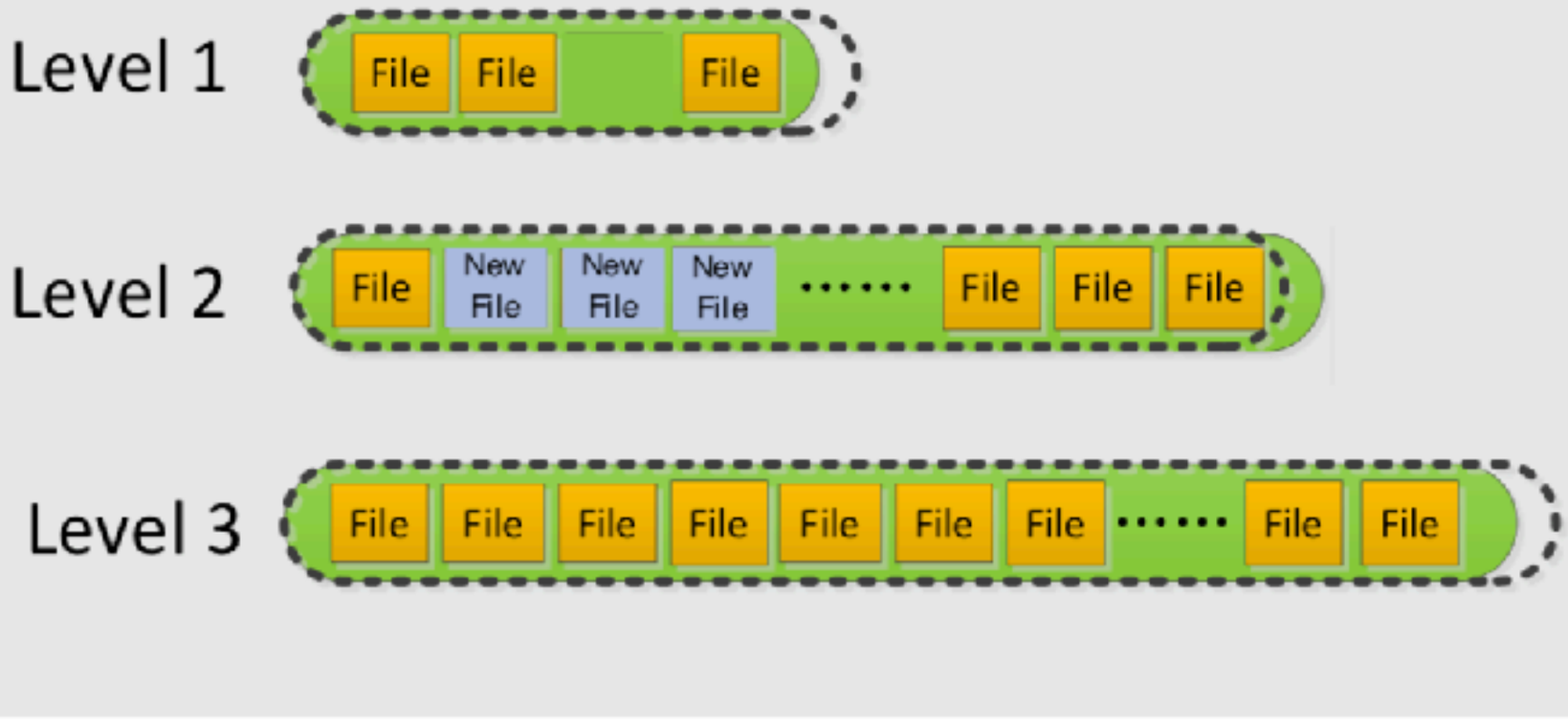
Level 压缩


Level 1
New File
New File
New File
New File
New File
Level 2
File
File
File
File
.....
File
File
Level 3
File
File
File
File
File
File
File
.....
File
File

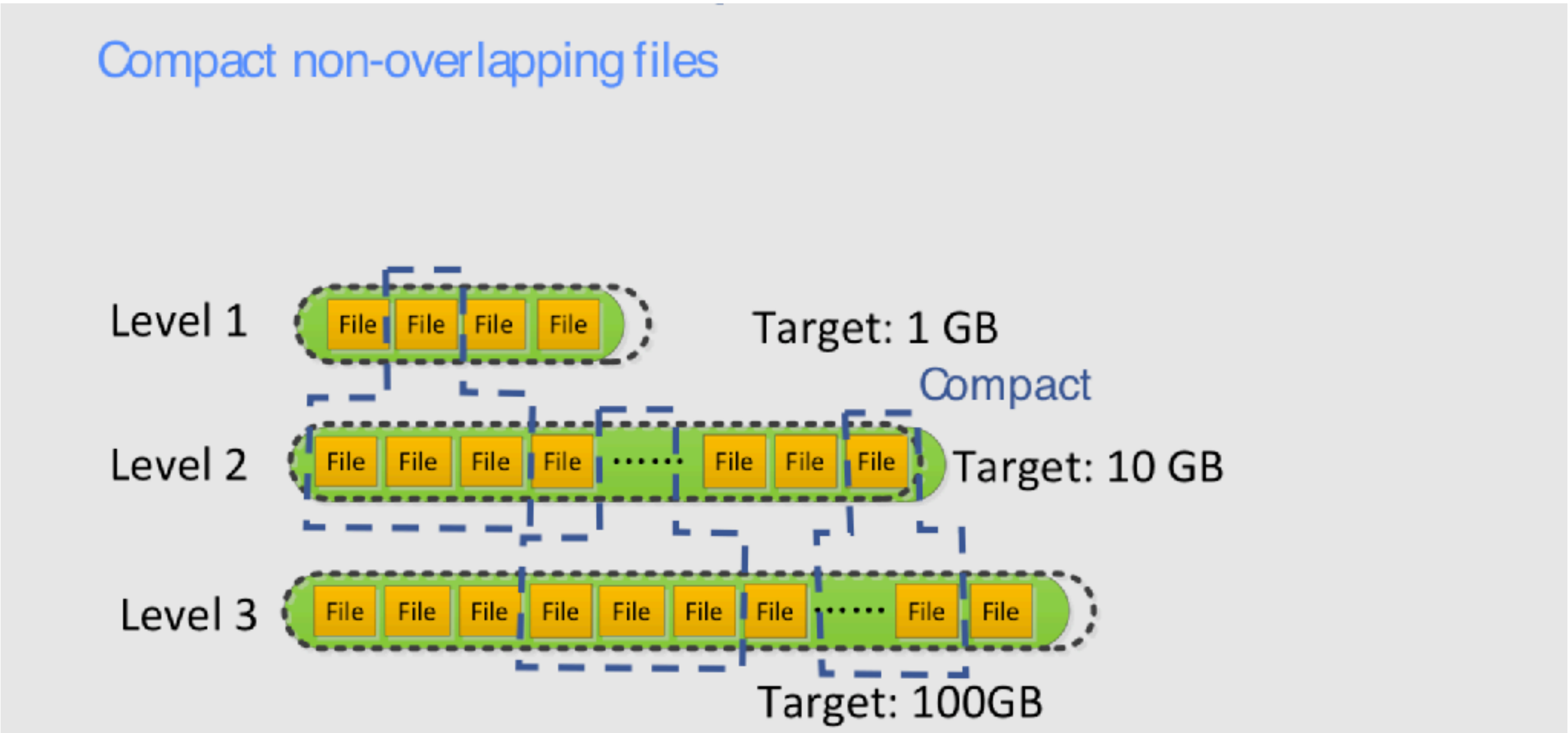
Level 压缩


Compact
Level 1
File
File
File
File
File
Level 2
File
File
File
File
File
File
Level 3
File
File
File
File
File
File
File
File
File

Level 压缩

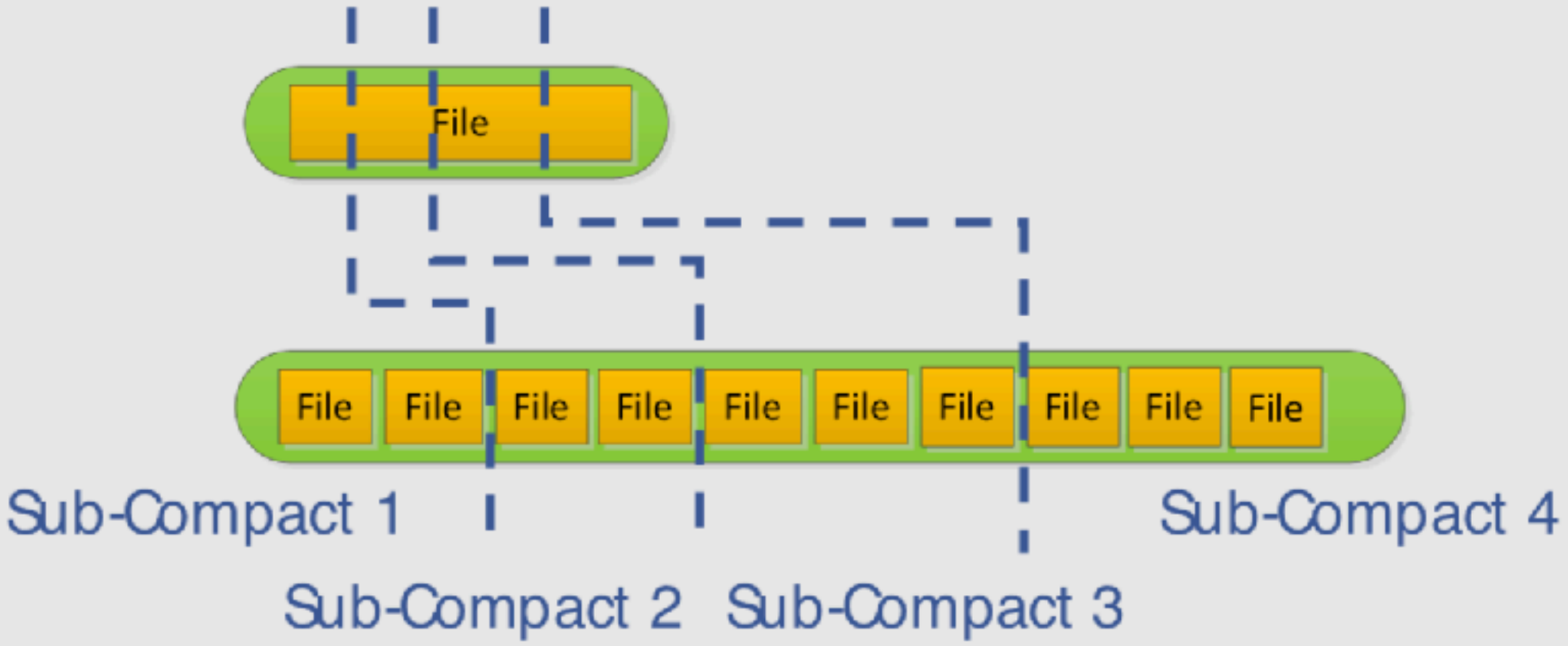

Level 1
File
File
File
Level 2
File
New File
New File
New File
File
File
File
Level 3
File
File
File
File
File
File
File
File
File

多线程压缩


Compact non-overlapping files
Level 1
File
File
File
File
Target: 1 GB
Compact
Level 2
File
File
File
File
File
File
File
Target: 10 GB
Level 3
File
File
File
File
File
File
File
File
File
Target: 100GB

多线程压缩

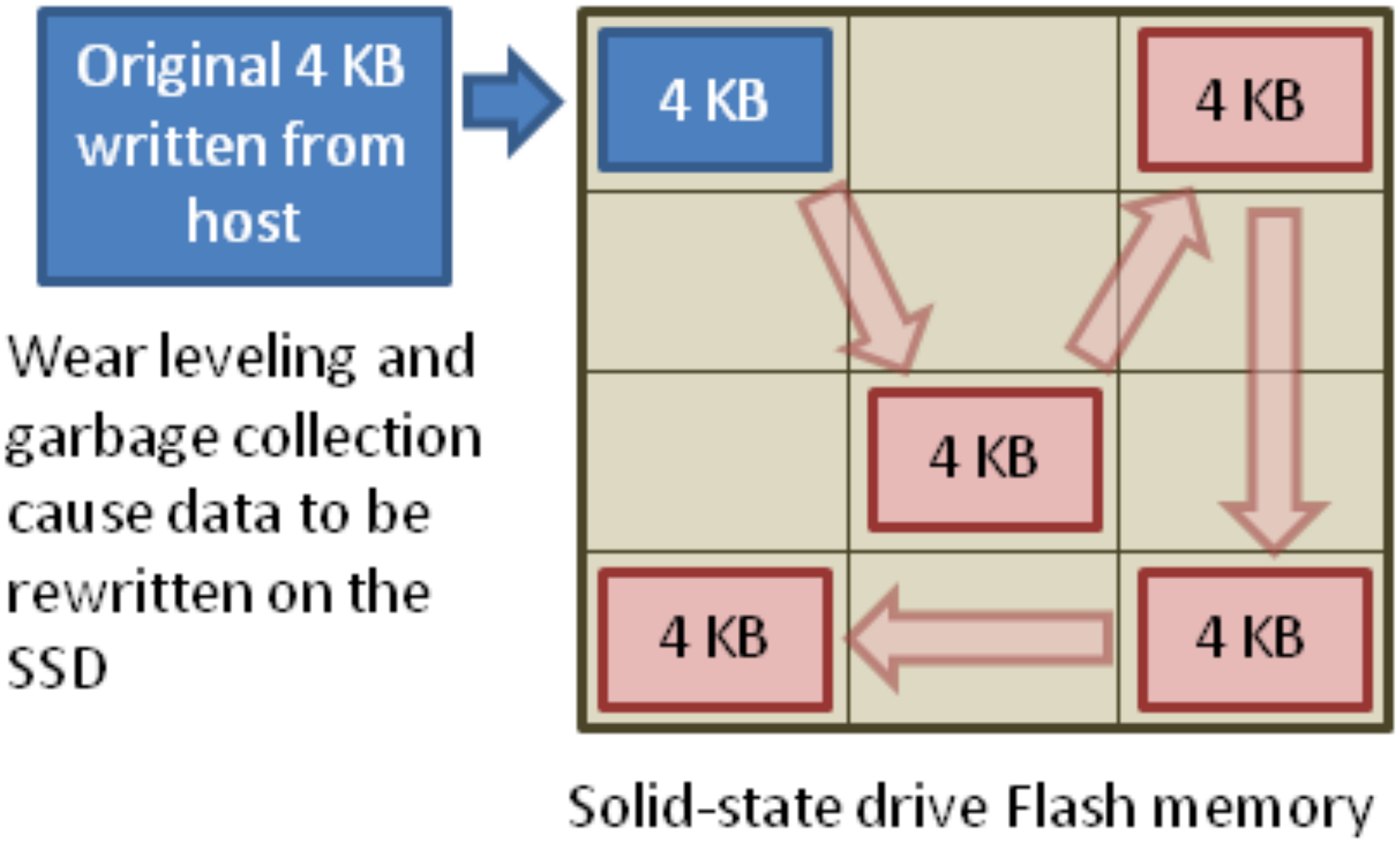
Divide One Compaction to sub-compactions



写入放大

Write Amplification

实际写入的操作量，是写入具体数据量所需的多倍！



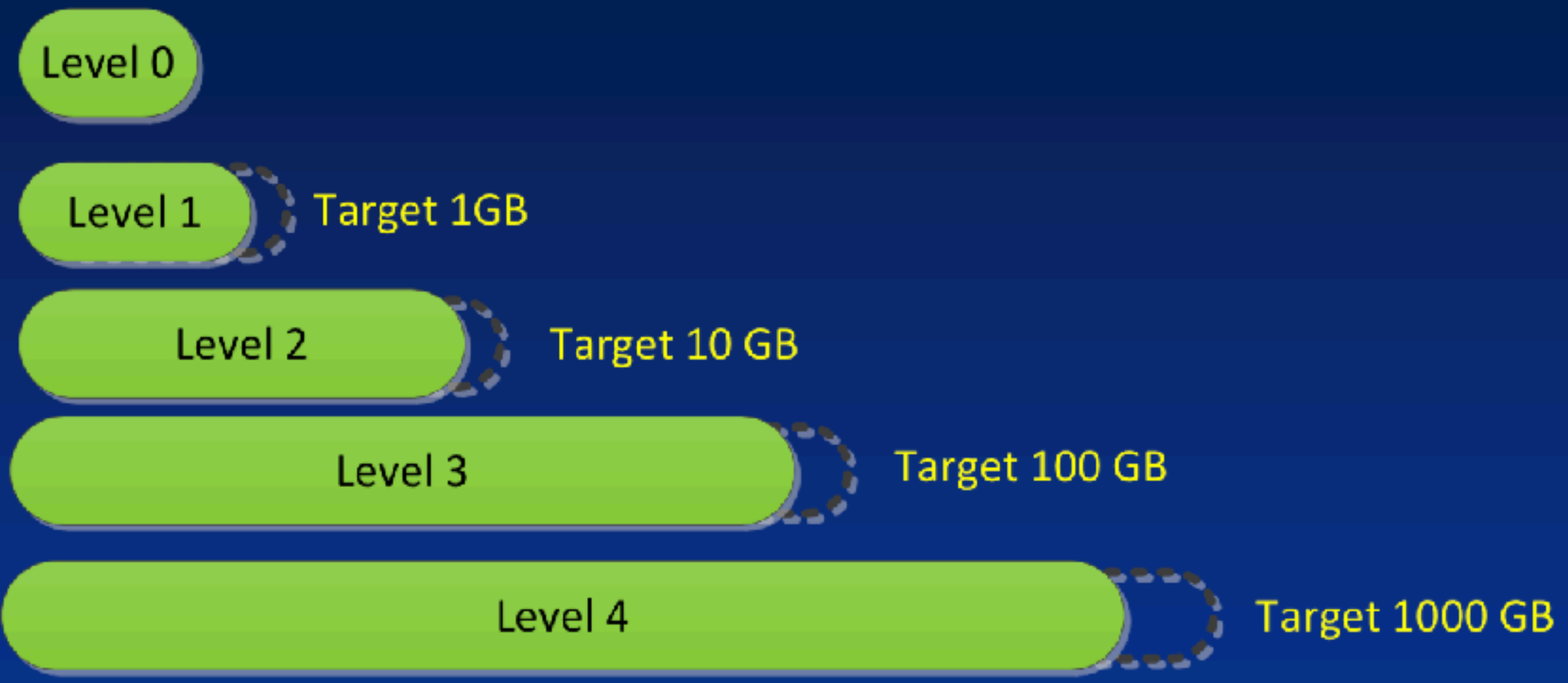
写入放大

是垃圾回收和损耗平衡导致的

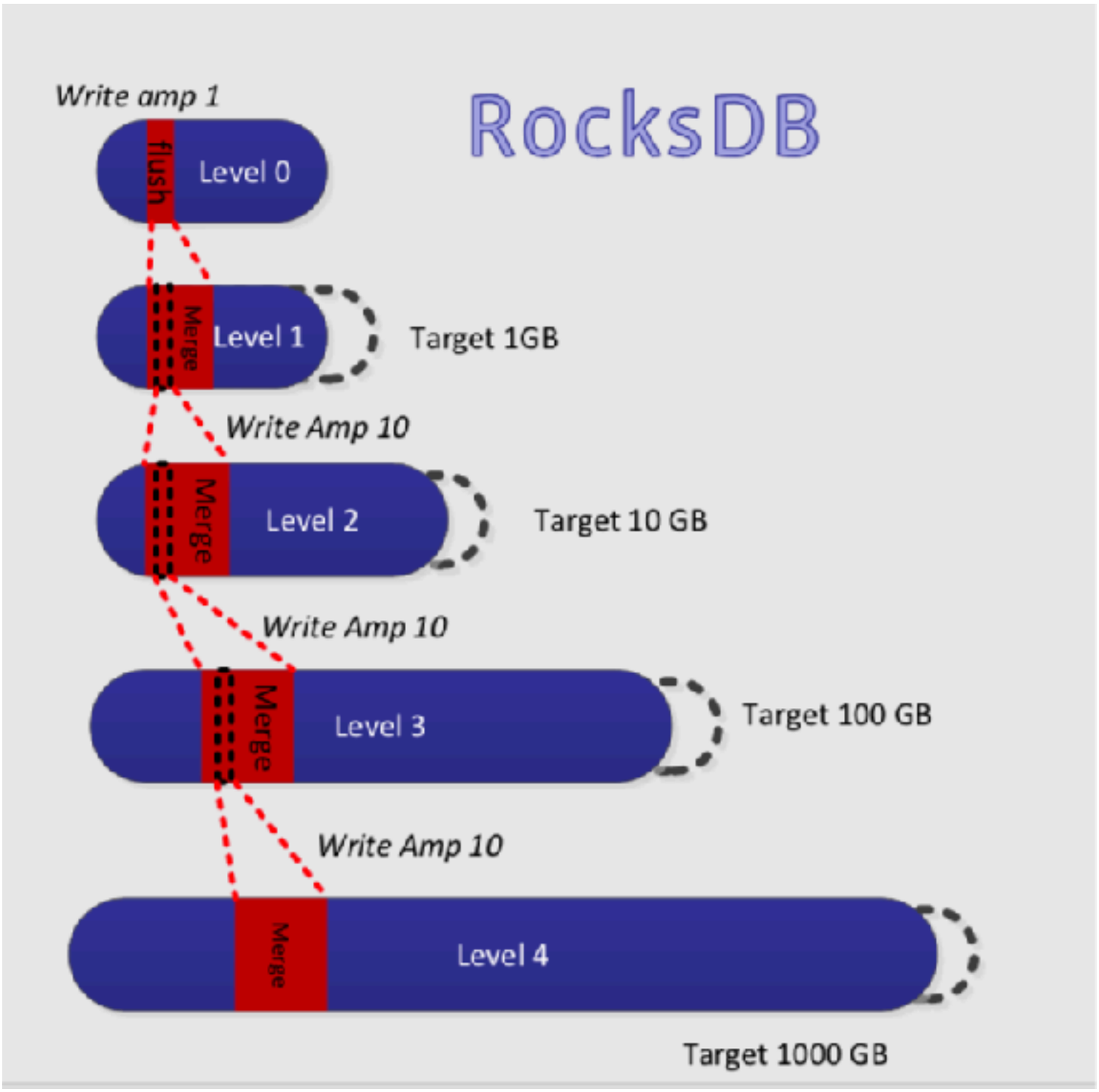
Compaction 1: compact to one file

- Write Amplification = 1000
- Read Amplification = 2 or 1 using bloom
- Space Amplification = 1.001
- Need Double Space for compaction

Compaction 2: Leveled-Compaction



- Read Amplification: number of levels or 1 (using bloom)
- Write Amplification: 10 * number of levels
- Space Amplification: 1.1


RocksDB
Write amp 1
flush
Level 0
Merge
Level 1
Target 1GB
Write Amp 10
Merge
Level 2
Target 10 GB
Write Amp 10
Merge
Level 3
Target 100 GB
Write Amp 10
Merge
Level 4
Target 1000 GB

RocksDB 读取优化

- 缩小存储占用
  - 与 OS 扇区对齐（4KB 单元）
  - Key Prefix Encoding 压缩
    - Compaction
- 写入压缩：Single Delete
- 读取加速：Bloom Filter
- ……

Thx