

MyRocks 入门

Lemonacy

目录

- MyRocks简介
- RocksDB的诞生
- LSM-Tree
- Bigtable
- MyRocks入门

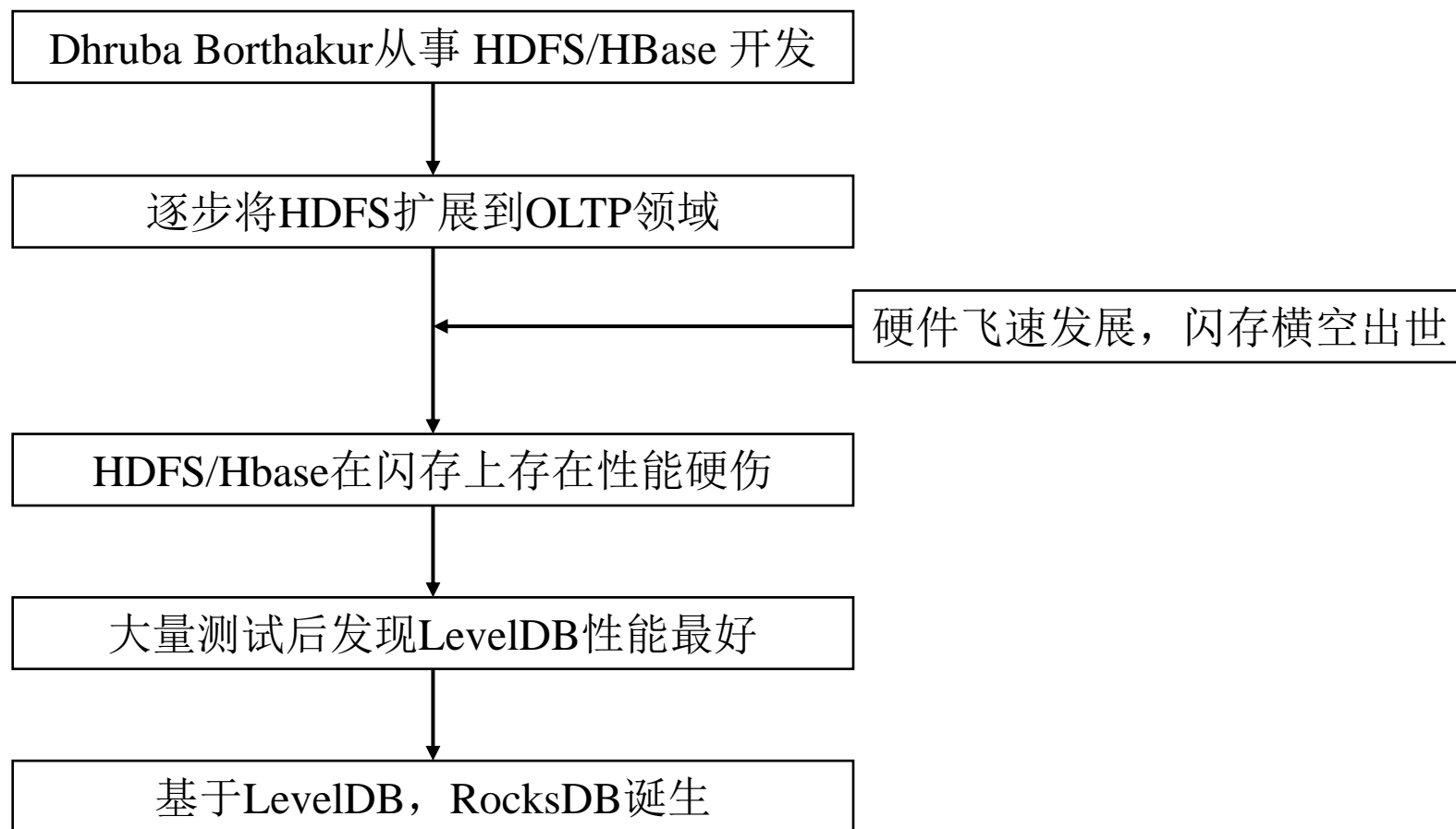
MyRocks简介

MyRocks简介

- MyRocks: MySQL on RocksDB
- RocksDB是Facebook基于LevelDB实现的，目前为Facebook内部大量业务提供服务。经过Facebook大量工作，将RocksDB作为MySQL的一个存储引擎移植到MySQL，称之为MyRocks。
- 相比InnoDB，RocksDB占用更少的存储空间，能够降低存储成本，提高热点缓存效率；具备更小的写放大比，能够更高效利用存储IO带宽；将随机写变为顺序写，提高了写入性能，延长了SSD使用寿命。

RocksDB 的诞生

RocksDB的诞生



The Log-Structured Merge-Tree (LSM-TREE)

O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33: 351-385.

LSM-Tree编年史

1996 LSM Tree

The log-structured merge-tree (LSM-tree)
cited by **1458**

2006 Bigtable

Bigtable: A distributed storage system for structured data
cited by **7825**

2013 RocksDB

Under the Hood: Building and open-sourcing RocksDB

1992 LSF

The design and implementation of a log-structured file system
cited by **2504**

2007 HBase

2010 Cassandra

Cassandra: a decentralized structured storage system
cited by **3914**

2011 LevelDB

LevelDB: A Fast Persistent Key-Value Store

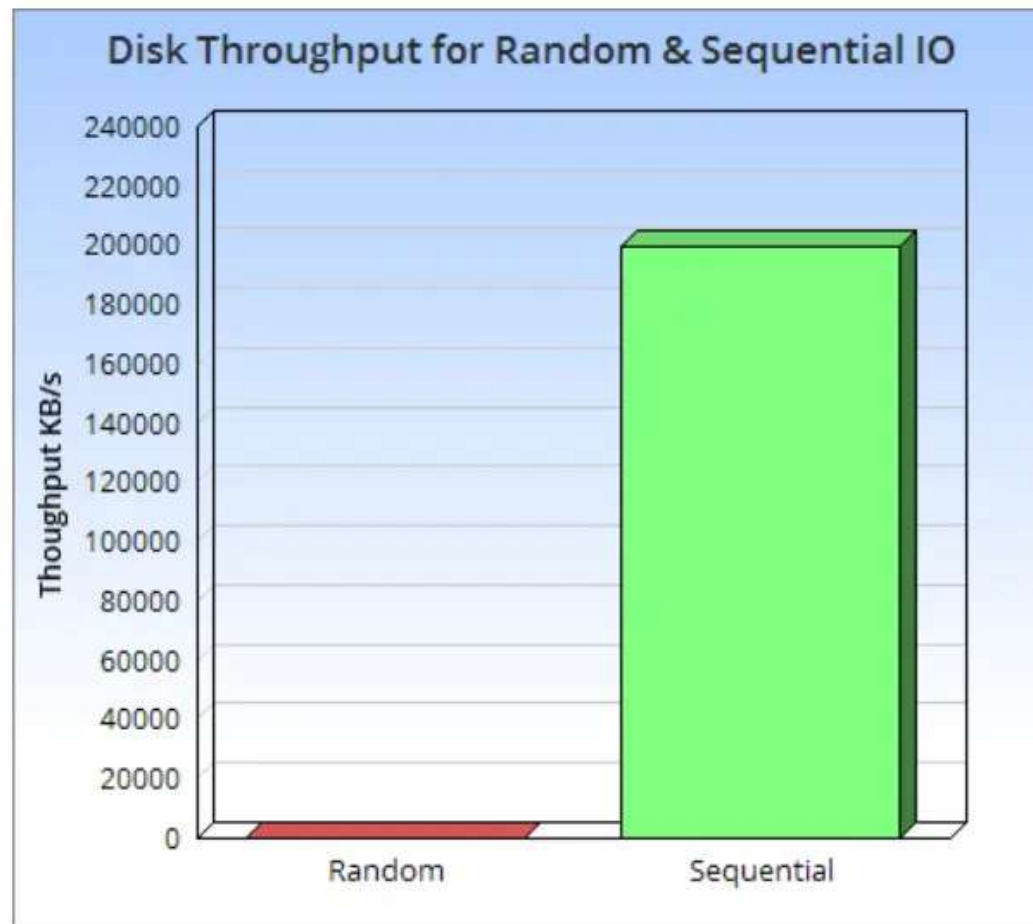
2015 TSM tree

The New InfluxDB Storage Engine: Time Structured Merge Tree

What makes LSM trees interesting is their departure from binary tree style file organisations that have dominated the space for decades. LSM seems almost counter intuitive when you first look at it, only making sense when you closely consider how files work in modern, memory heavy systems.

LSM-Tree诞生背景

- LSM Tree的全称为Log-Structured Merge Tree，是一个分层、有序、针对块存储设备（机械硬盘和SSD）特点而设计的数据存储结构
- 磁盘在随机操作时速度慢，在顺序访问时速度快
- 即使是SSD，由于块擦除和垃圾回收的影响，顺序写速度还是比随机写速度快很多



LSM-Tree诞生背景

- SSD的写放大（Write Amplification）
 - 闪存基本构成：页page（4K）→块block（通常64个page组成一个block，有的是128个）→面plane（多个block组成）→die（plane就是一个die）→闪存片（多个die组成）→SSD（多颗闪存片组成）
 - 当操作系统删除数据时，并不会马上删除，而是标记一个“删”的tag
 - 当机械硬盘要写入新数据时可以直接覆盖那些已经被标记“删”标签的数据
 - 固态硬盘不行，只能先擦除旧的数据才能写入新数据
 - NAND闪存工作原理是以4K页（page）为一个单元写入的，但擦除只能以块block（64个page）为单位
 - 实际写入的物理数据量是写入数据量的多倍

LSM-Tree诞生背景

- 对于大量写入操作的场景下，B-Tree的写入会带来额外的I/O开销
 - 更改数据时多次覆盖整个 Page
 - 可能会导致页分裂
- 顺序写入性能远好于随机写入
- 避免SSD的磨损，延长使用寿命
- ...

LSM-Tree设计理念

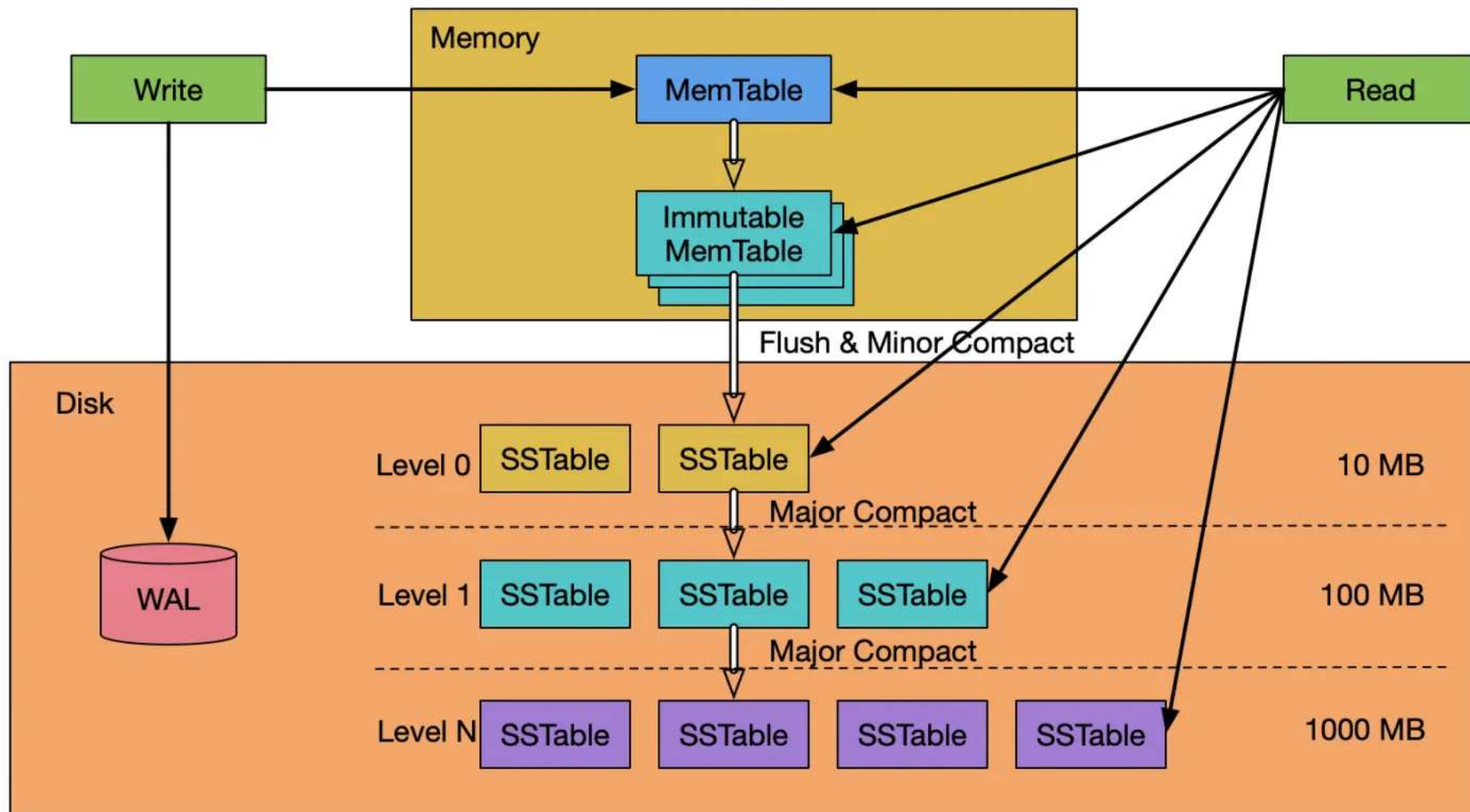
- 将整个磁盘就看做是一个日志，在日志中存放永久性数据及其索引，每次都添加到日志的末尾
- 将写入推迟 (Defer) 并转换为批量 (Batch) 写
- 采用内存+磁盘两级存储，增删改均为append，首先将数据大量写入缓存，当积攒到一定程度后，再批量写入文件
- 对于文件系统的大多数存取都是顺序性的，从而提高磁盘带宽利用率，故障恢复速度快
- 适时合并
- 引入索引和布隆过滤器加快查询
- 尽可能减少数据写入的I/O开销

SSTable

- SSTable: Sorted String Table, 一种可持久化, 有序且不可变的键值存储结构, 一旦写入磁盘中, 就像日志一样不能再修改



LSM-Tree架构



B-Tree与LSM-Tree对比

	B-Tree	LSM-Tree	备注
优势	读取更快	写入更快	
写放大	1. 数据和WAL 2. 更改数据时多次覆盖整个PAGE	1. 数据和WAL 2. Compaction	SSD 不能过多擦除。
写吞吐	相对较低： 1. 大量随机写	相对较高： 1. 较低的写放大（取决于数据和配置） 2. 顺序写入 3. 更为紧凑	
压缩率	存在较多内部碎片	1. 更为紧凑，没有内部碎片 2. 压缩潜力更大	压缩不及时会造成LSM树存在很多garbage
后台流量	1. 更稳定可预测，不会受后台compaction 突发流量影响。	1. 写吞吐过高，compaction 跟不上，会进一步加重读放大。 2. 由于外存总带宽有限，compaction 会影响读写吞吐。 3. 随着数据越来越多，compaction 对正常写影响越来越大。	RocksDB 写入太过快会引起write stall，即限制写入，以期尽快 compaction 将数据下沉。
存储放大	有些 Page 没有用满	同一个 Key 存多遍	
并发控制	1. 同一个 Key 只存在一个地方 2. 树结构容易加范围锁。	同一个 Key 会存多遍，一般使用 MVCC 进行控制。	

Bigtable: A Distributed Storage System for Structured Data

Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2): 1-26.

Bigtable简介

- 谷歌一个分布式的结构化数据存储系统，用于解决GFS无法对结构化数据进行访问与管理
- 一个开源实现版本：Apache HBase
- 谷歌将许多自己提供服务的数据使用Bigtable进行管理，例如Google Earth、Google Finance、Gmail等等
- 数据种类繁多、后端容量巨大、延迟敏感、高容错性、高可用性、可扩展性

数据模型

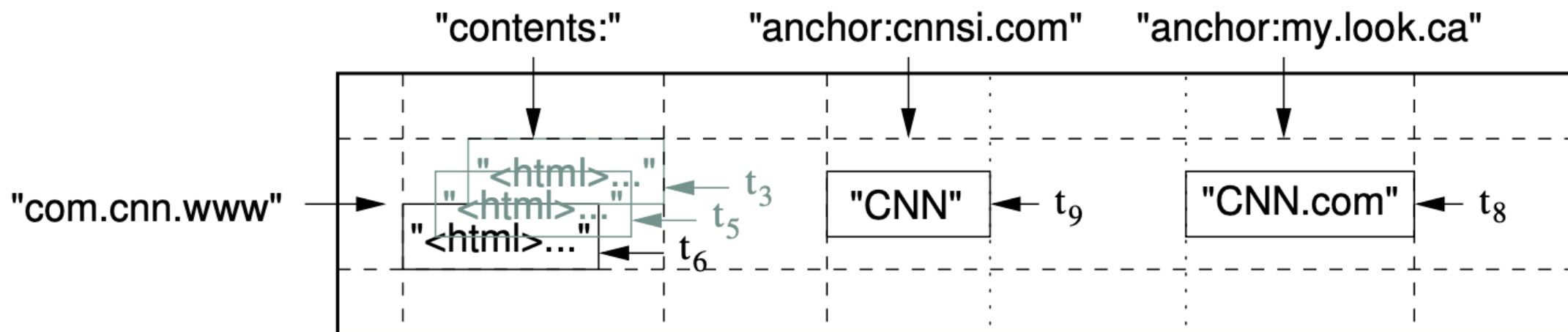
- Bigtable是一个稀疏的、分布式的、持久化的多维度的排序map (sorted map)
- 数据被设计通过三个层次进行索引：行关键字 (row key)，列关键字 (column key) 和时间戳 (timestamp)
- map中的key是这三个参数的复合结构，map中的每个value是一个未经解析的byte数组：

(row: string, column: string, time: int64) → string

- 足够简单灵活，可以满足大多数的数据存储需求
- 足够可靠

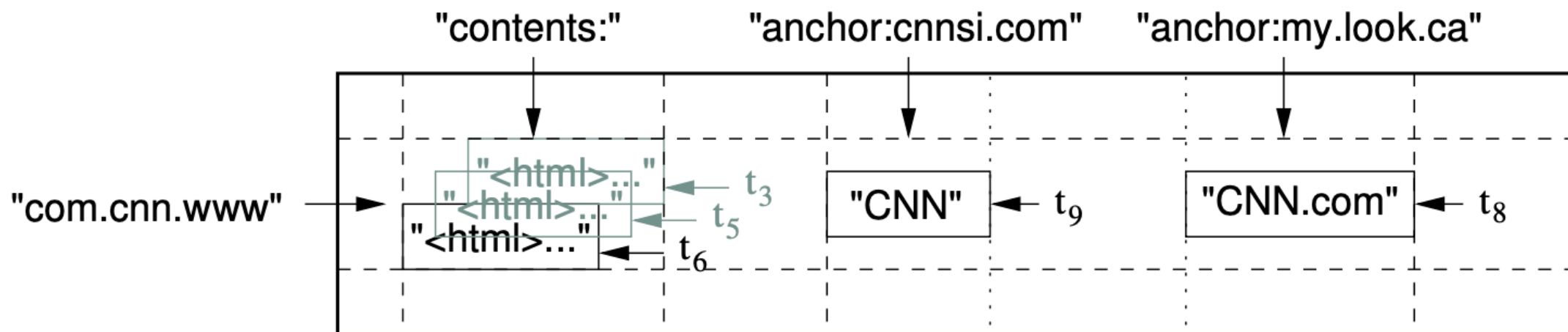
数据模型：Row

- 可以是任意的字符串
- 对同一个行关键字的读或者写操作都是原子的（无论多少个不同列）
- 通过行关键字的字典顺序来组织数据
- 表中的每个行都可以动态分区，每个分区叫一个tablet
- tablet是数据分布和负载均衡调整的最小单位，当操作只读取行中很少几列的数据时访问效率很高



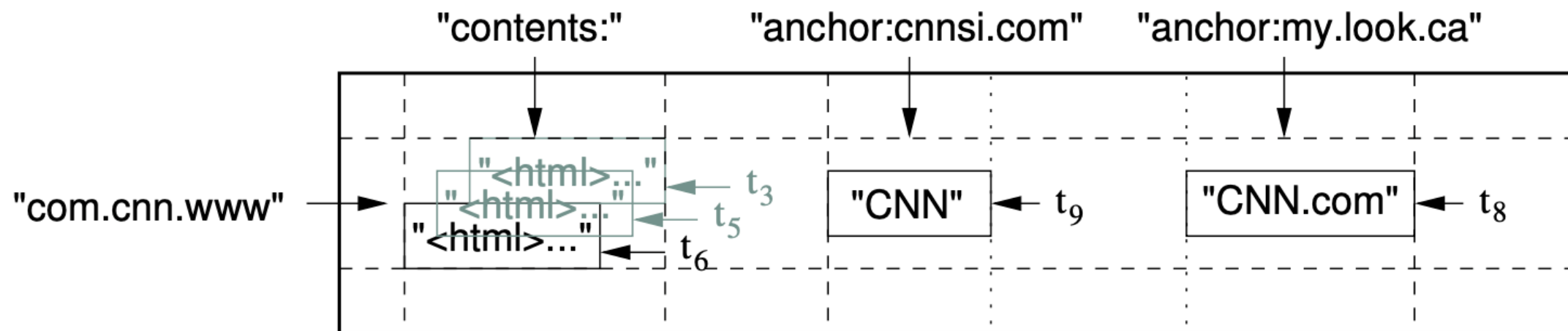
数据模型：Column Family

- 列关键字组成的集合叫做列族（Column Families）
- 列族是访问控制的基本单位
- 存放在同一列族下的所有数据通常都属于一个类型
- 列族在使用之前必须先创建
- 列关键字的命名语法： *family: qualifier* （列族：限定词）

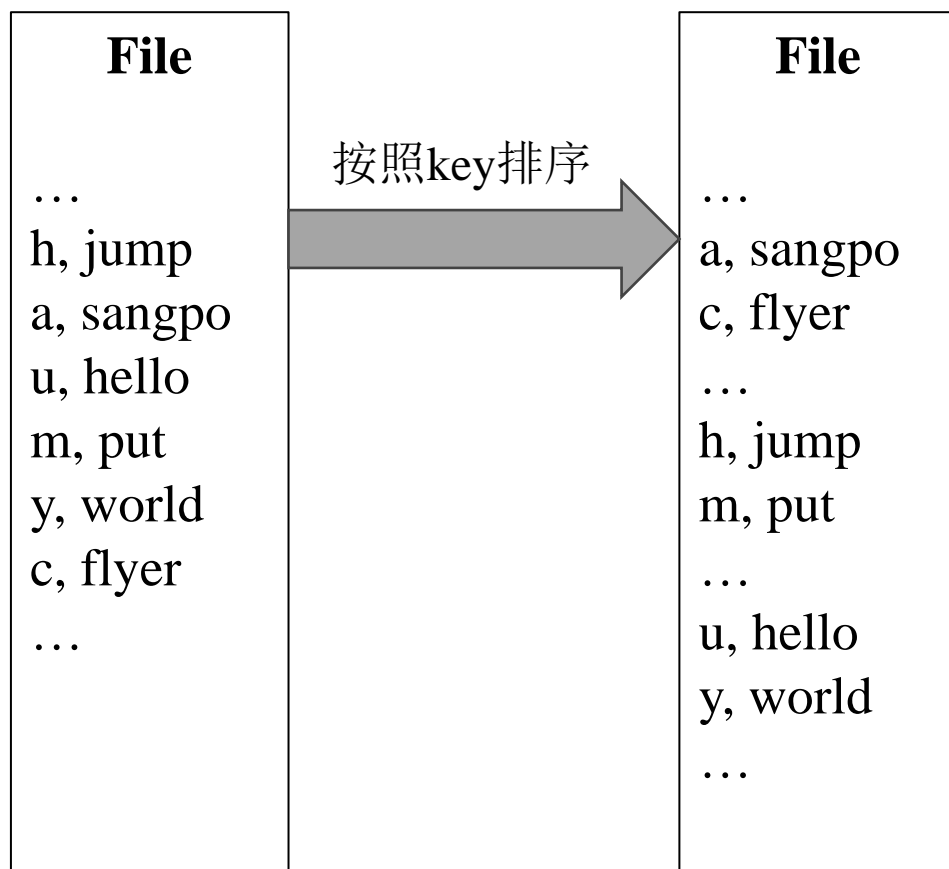


数据模型：Timestamp

- 表的每一个数据项都可以包含同一份数据的不同版本
- 不同版本的数据通过时间戳来索引

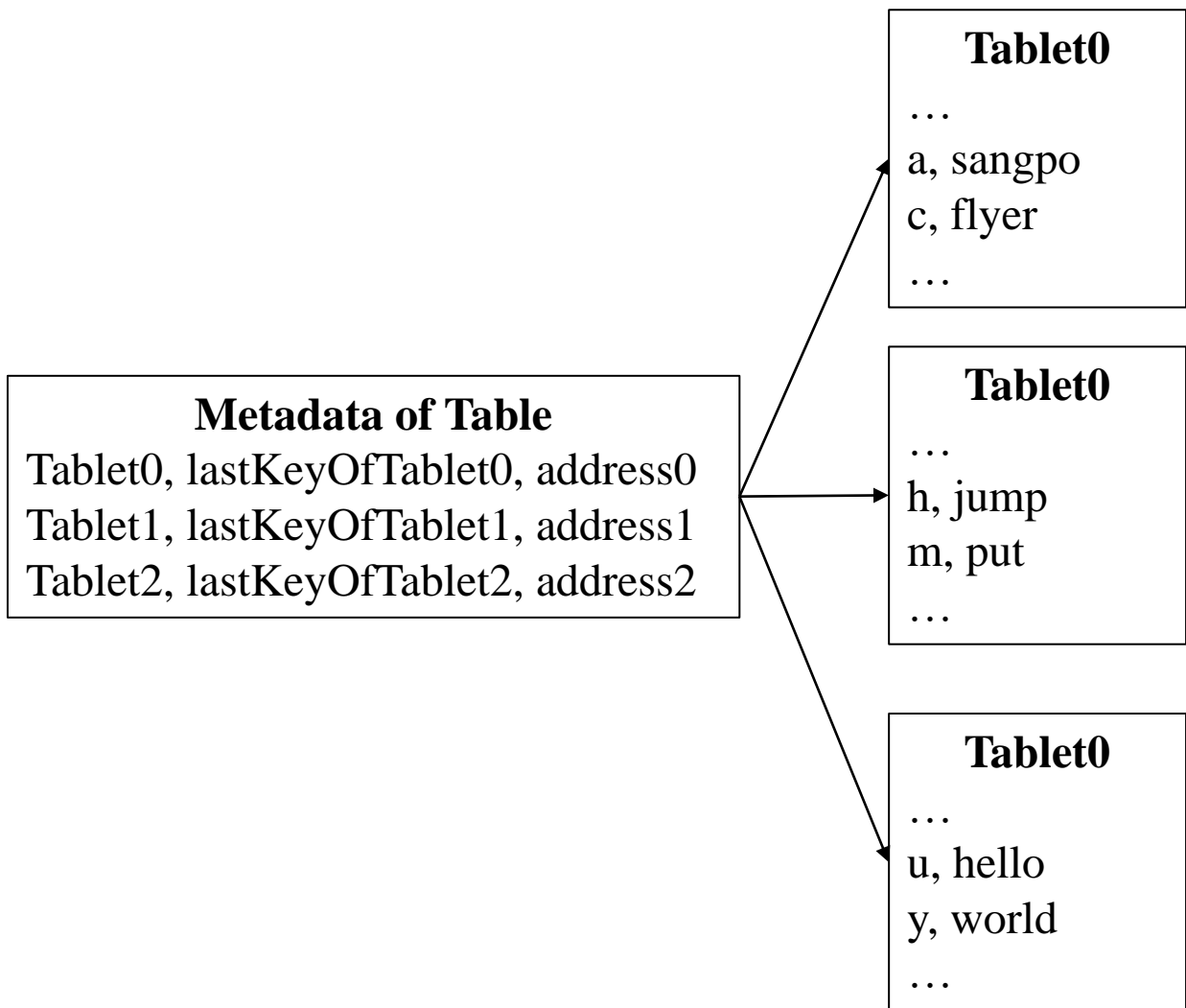


如何在文件内快速查询



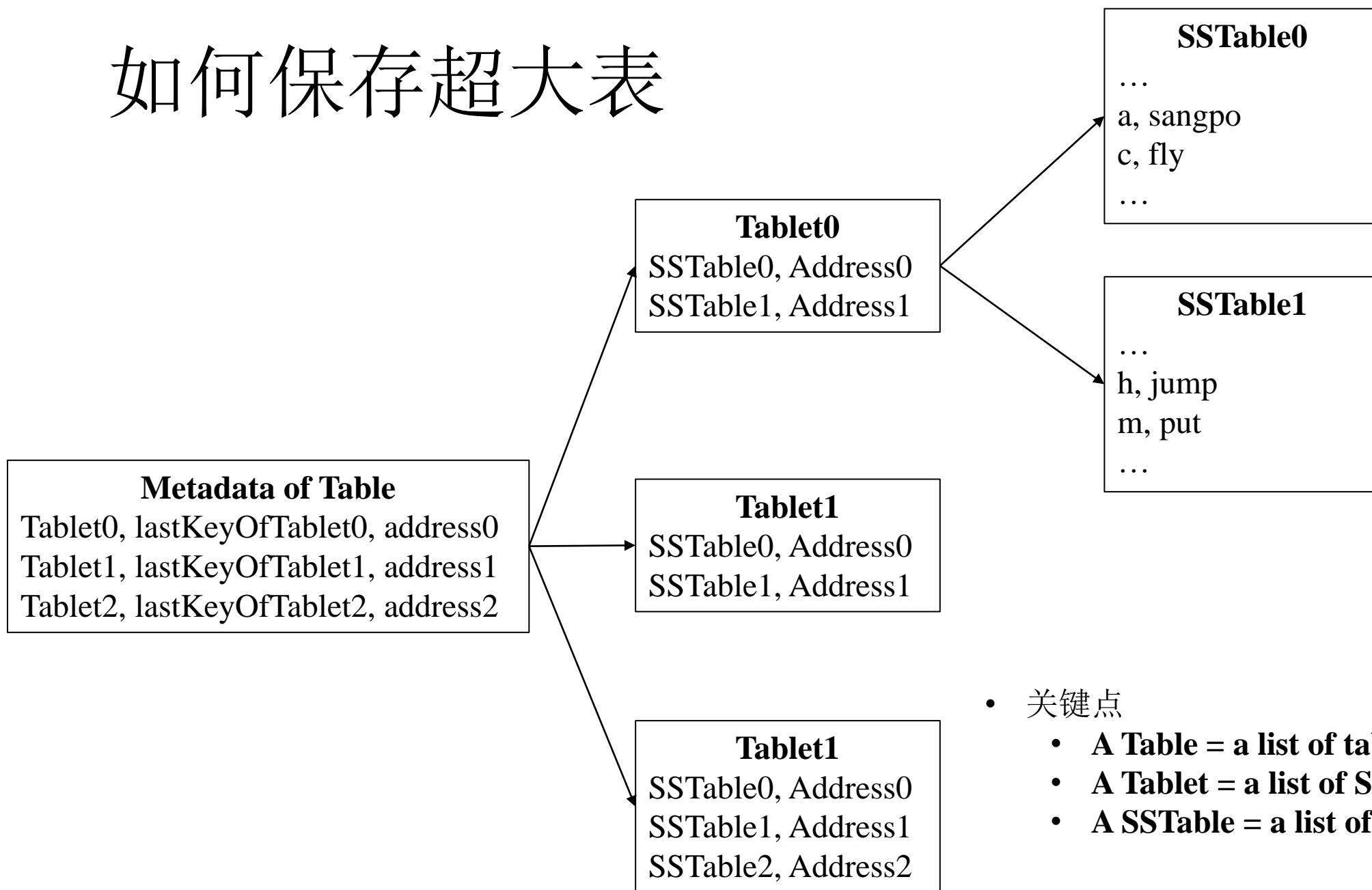
- 关键点
 - **Table = a list of sorted <key, value>**

如何保存大表



- 关键点
 - **A Table = a list of tablets (小表)**
 - **A Tablet = a list of sorted <key, value>**

如何保存超大表

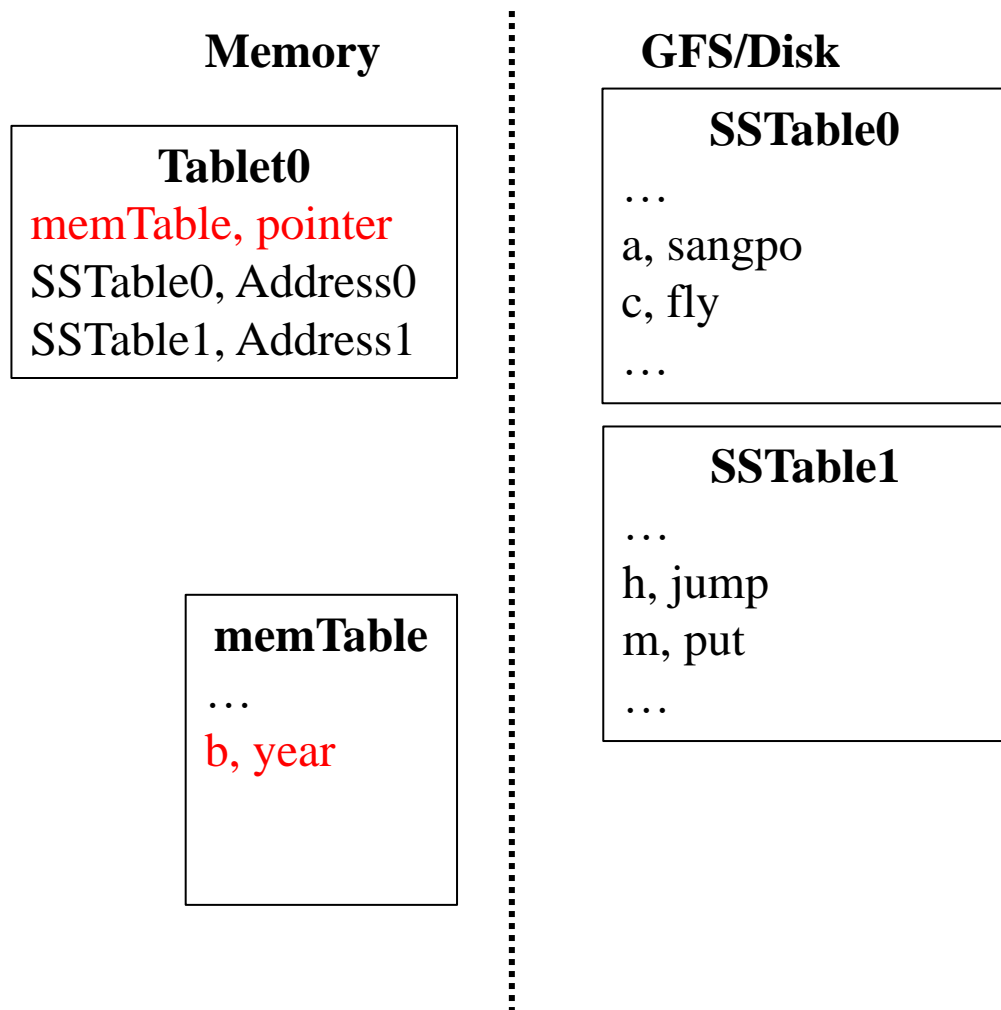


- 关键点

- **A Table = a list of tablets** (小表)
- **A Tablet = a list of SSTables** (小小表)
- **A SSTable = a list of sorted <key, value>**

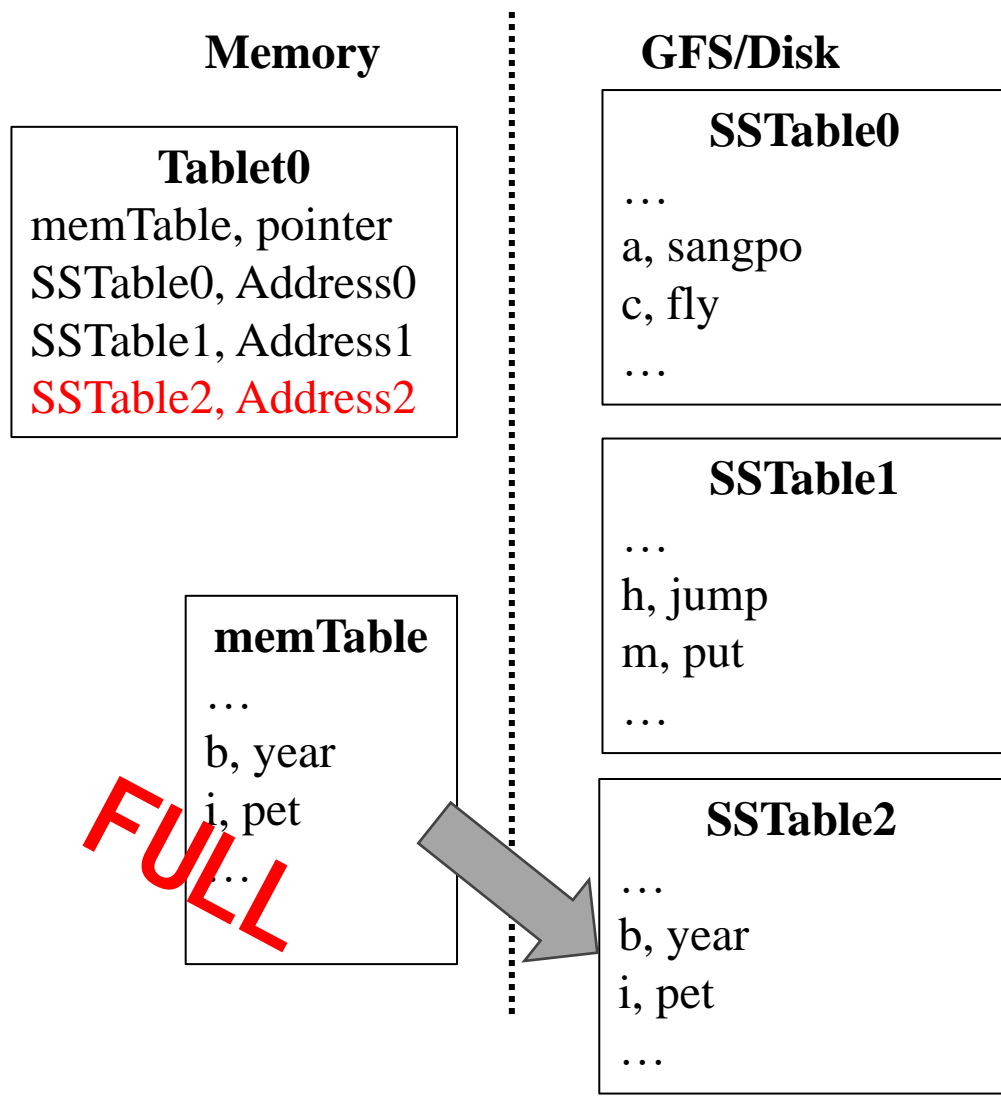
如何写数据

task: 添加<b, year>



- 关键点
 - 通过写入memtable（内存表）加速
 - A Tablet = memtable + a list of SSTables

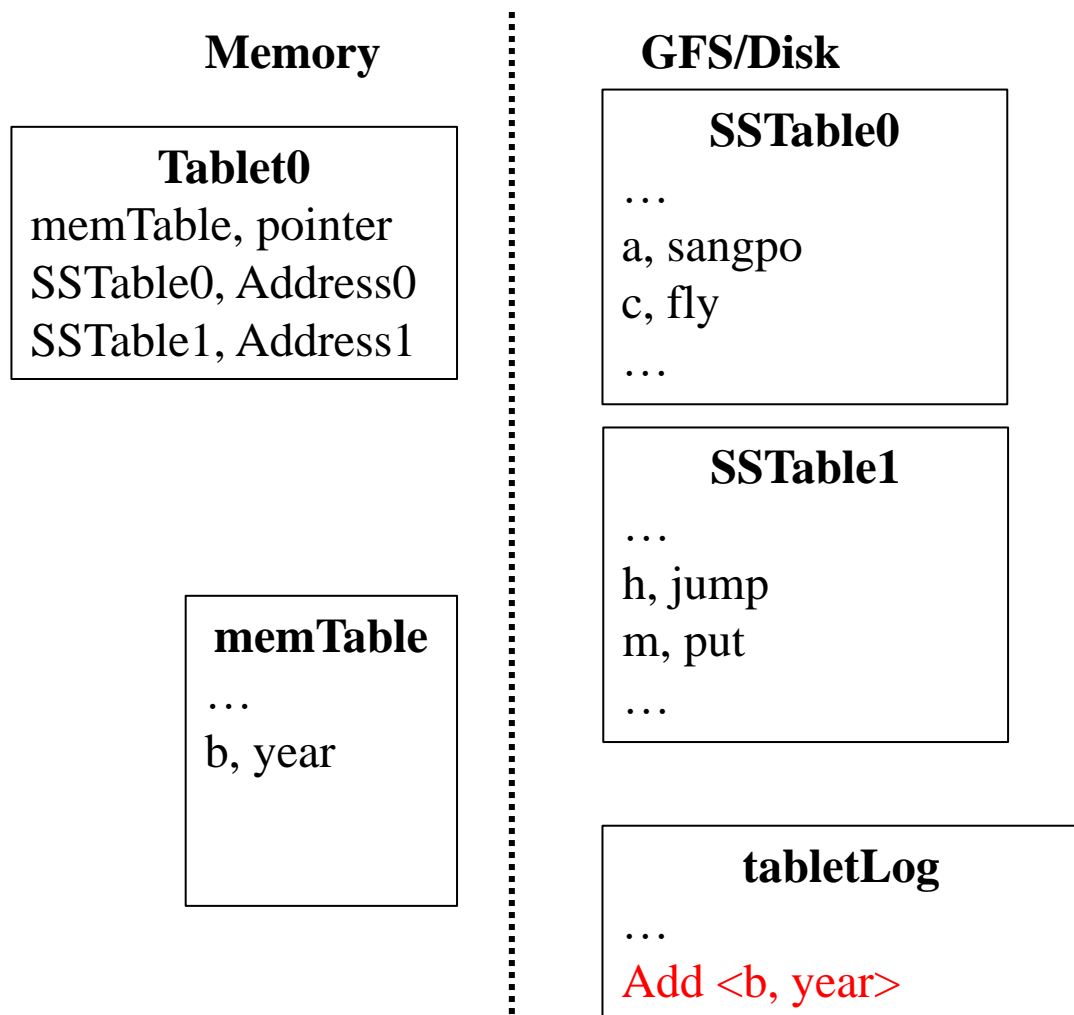
内存表过大



- 关键点
 - 将memTable导入硬盘，成为一个新的SSTable

如何避免内存数据丢失

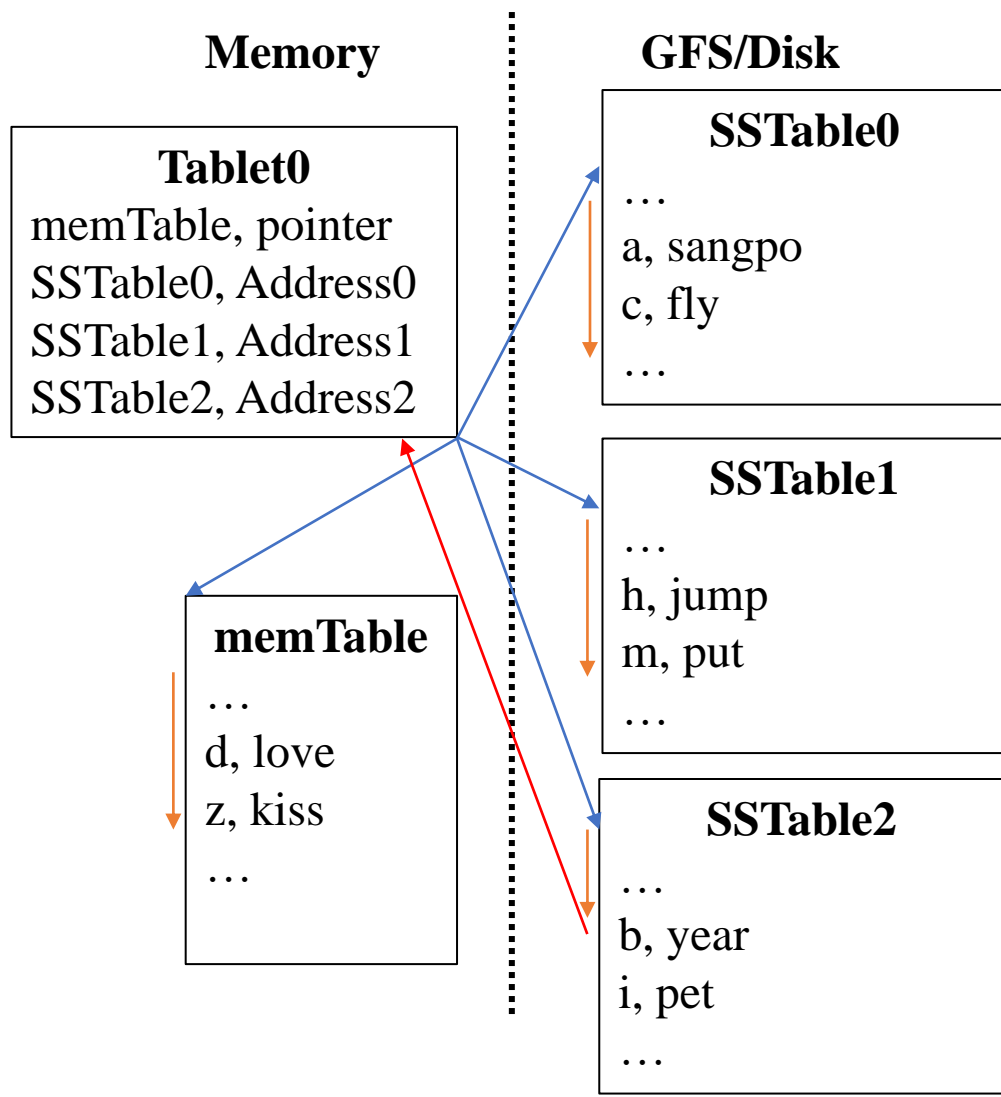
task: 添加<b, year>



- 关键点
 - **A Tablet = memtable + a list of SSTables + log**

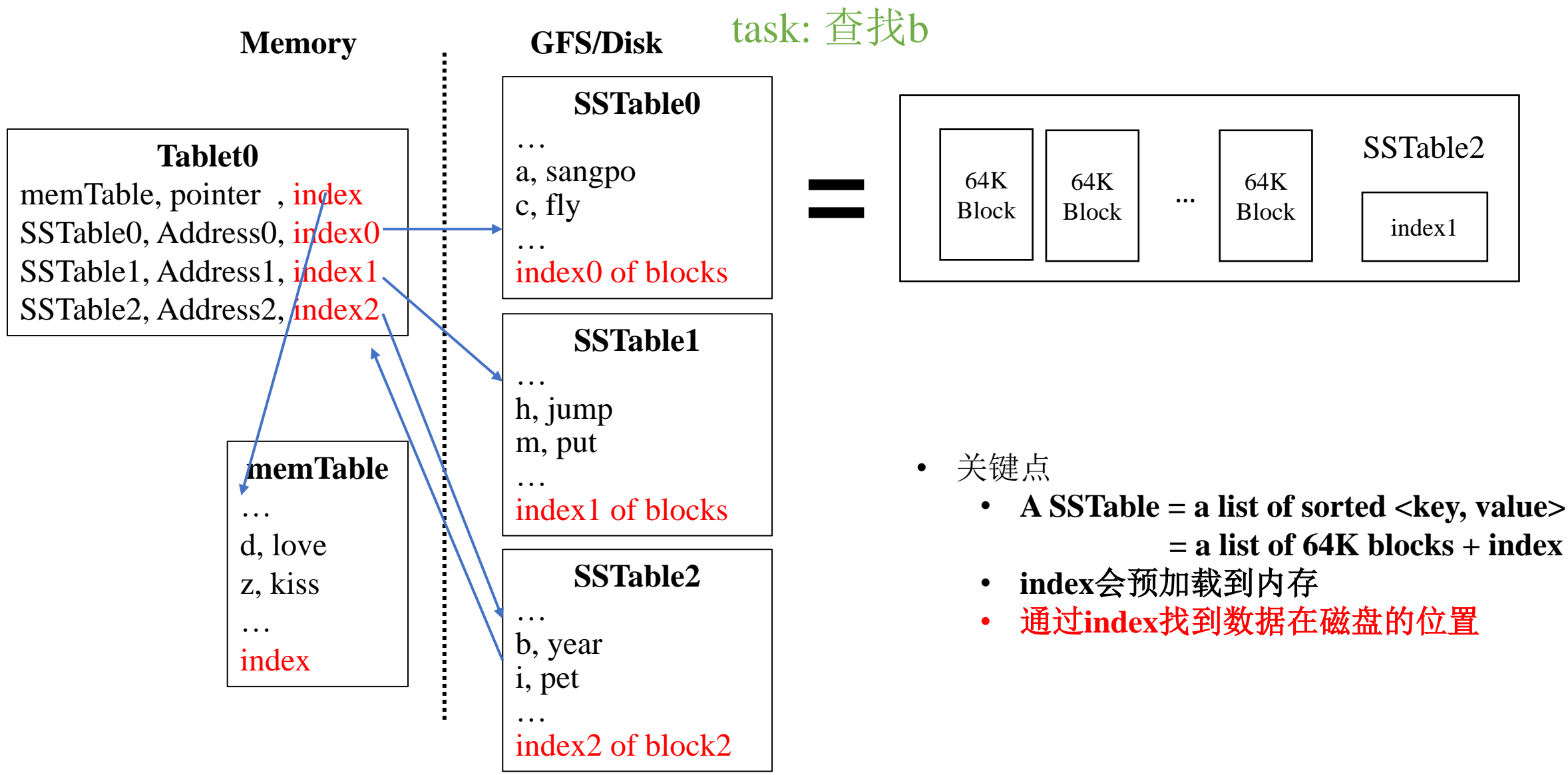
如何读数据

task: 查找b



- 关键点
 - **SSTable**内部数据有序
 - **SSTable**之间数据无序
 - 需要查找所有的**SSTable**和**memTable**
 - 需要在硬盘中的**SSTable**查找该元素

如何加速读数据



如何进一步加速读数据

Memory

GFS/Disk

task: 查找b

Tablet0

memTable, pointer , bloomfilter **X**, index
SSTable0, Address0, bloomfilter1 **X**, index0
SSTable1, Address1, bloomfilter2 **X**, index1
SSTable2, Address2, bloomfilter3 **✓**, index2

memTable

...
d, love
z, kiss
...
index
bloomfilter

SSTable0

...
a, sangpo
c, fly
...
index0
bloomfilter0

SSTable1

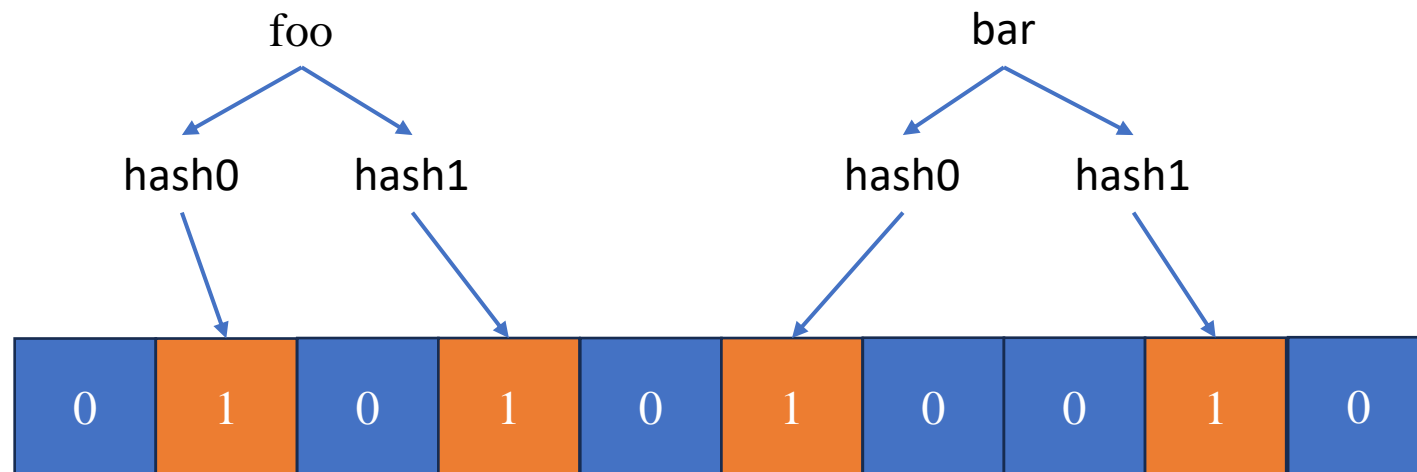
...
h, jump
m, put
...
index1
bloomfilter1

SSTable2

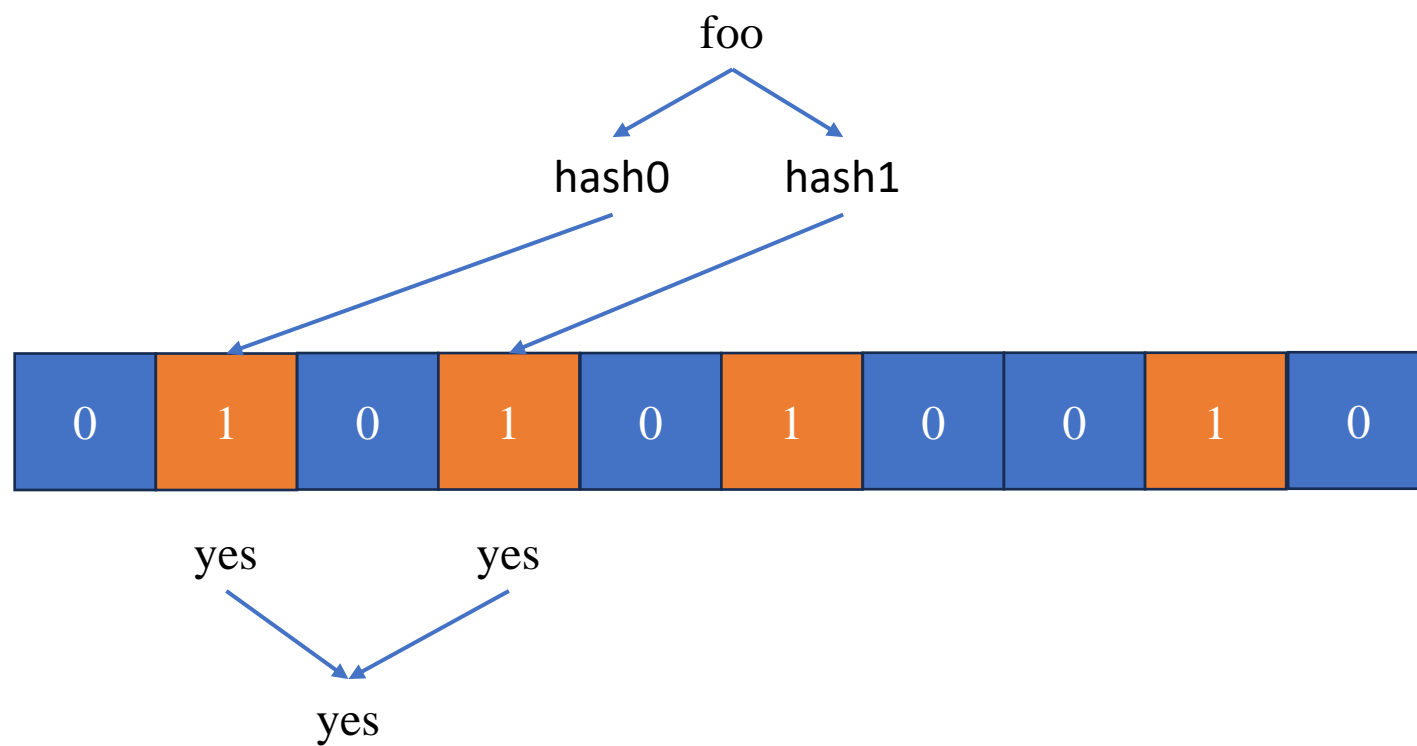
...
b, year
i, pet
...
index2
bloomfilter2

- 关键点
 - A SSTable = a list of sorted <key, value>
= a list of 64K blocks + index
+ bloomfilter
 - Bloomfilters会预加载到内存
 - 通过bloomfilter判断元素是否存在

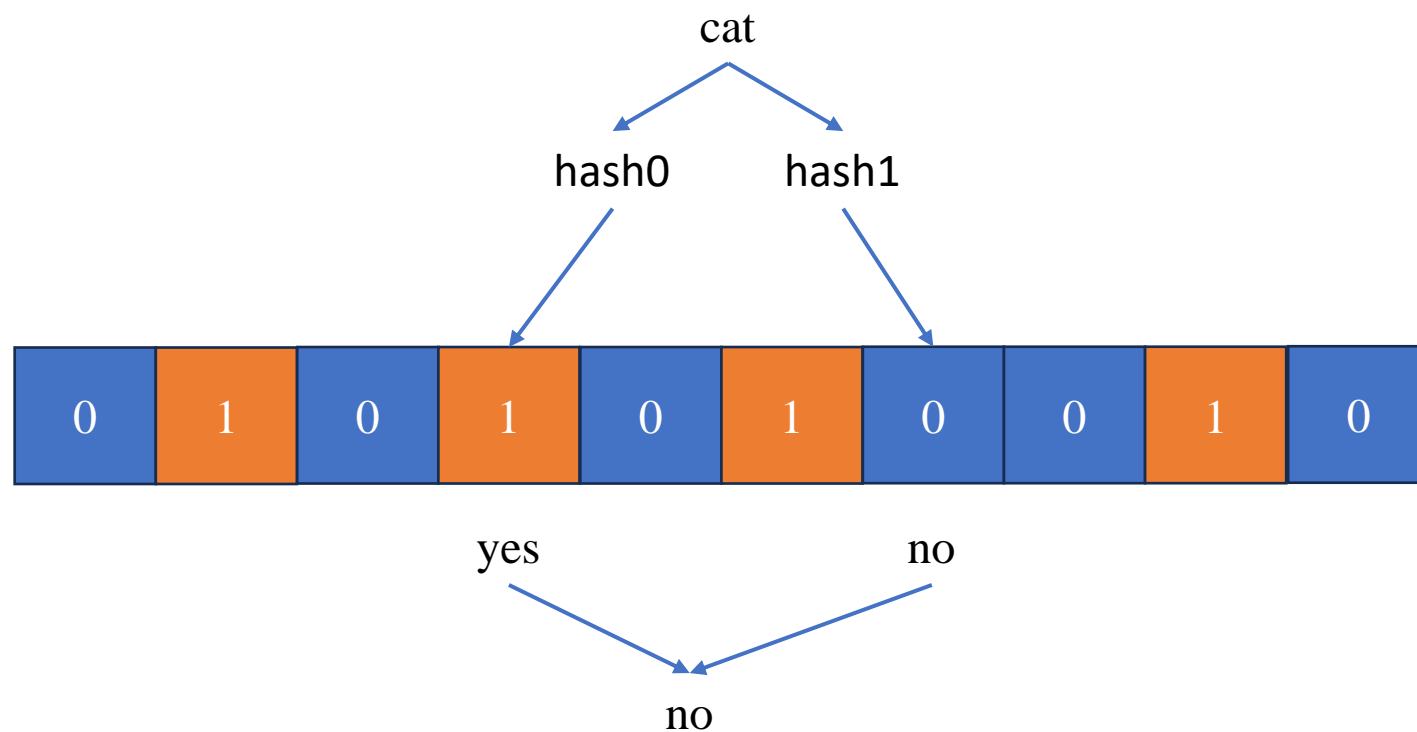
如何构建bloomfilter



bloomfilter查找




bloomfilter查找




bloomfilter: 可以判断数据一定不在表中

表的逻辑视图

		Career	
Name	Photo	Album	Masterpiece
Hanpo			
Lorde	<div>2021</div>  <div>2018</div> <div>2014</div> <div>2010</div>	Solar Power, 2021 Melodrama, 2017 Pure Heroine, 2013	Solar Power, 2021 Green Light, 2017 Yellow Flicker Beat, 2014 Royals, 2013
Tiger			

将逻辑视图转换为物理存储

表的逻辑视图

Name	Photo	Career	
		Album	Masterpiece
Sango			
Lorde		Solar Power, 2021 Melodrama, 2017 Pure Heroine, 2013	Solar Power, 2021 Green Light, 2017 Yellow Flicker Beat, 2014 Royals, 2013
Tiger			

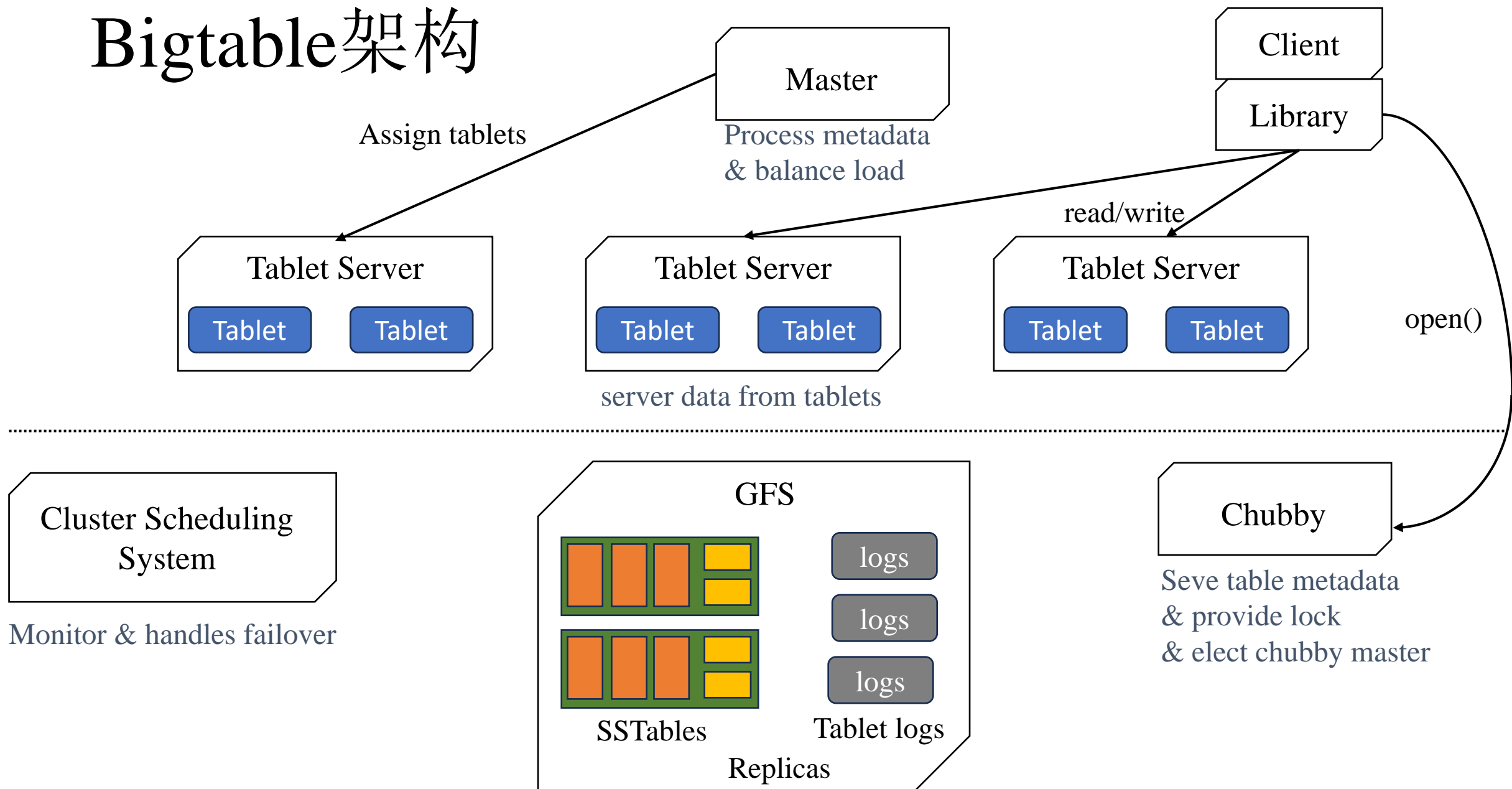
key = string(row, column, time)

Table

```
...
Lorde;Career:Album;2013 -> Pure Heroine
Lorde;Career:Masterpiece;2021 -> Solar Power
Lorde;Career:Masterpiece;2017 -> Green Light
Lorde;Career:Masterpiece;2014 -> Yellow Flicker Beat
Lorde;Career:Masterpiece;2012 -> Royals
Lorde; Photo;2021 -> 2021.jpg
Lorde; Photo;2018 -> 2018.jpg
...
```

Bigtable不是关系型数据库，但沿用了关系型数据库的行列表概念

Bigtable架构



MyRocks入门

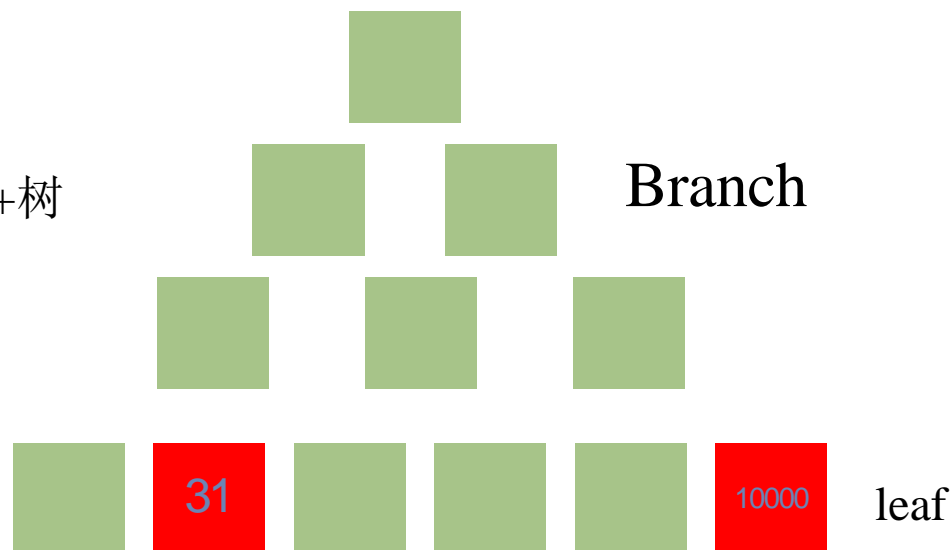
MyRocks诞生背景

- 存储成本的缩减是基础软件的核心
- SSD价格高昂，存储容量有限
- B+树碎片严重，压缩效率低下，磁盘成本高
- InnoDB写放大过高，导致写性能较差
- 考虑引入LSM-Tree结构存储引擎
- 移植RocksDB比重新开发一个新的存储引擎更合适

InnoDB的问题：随机写

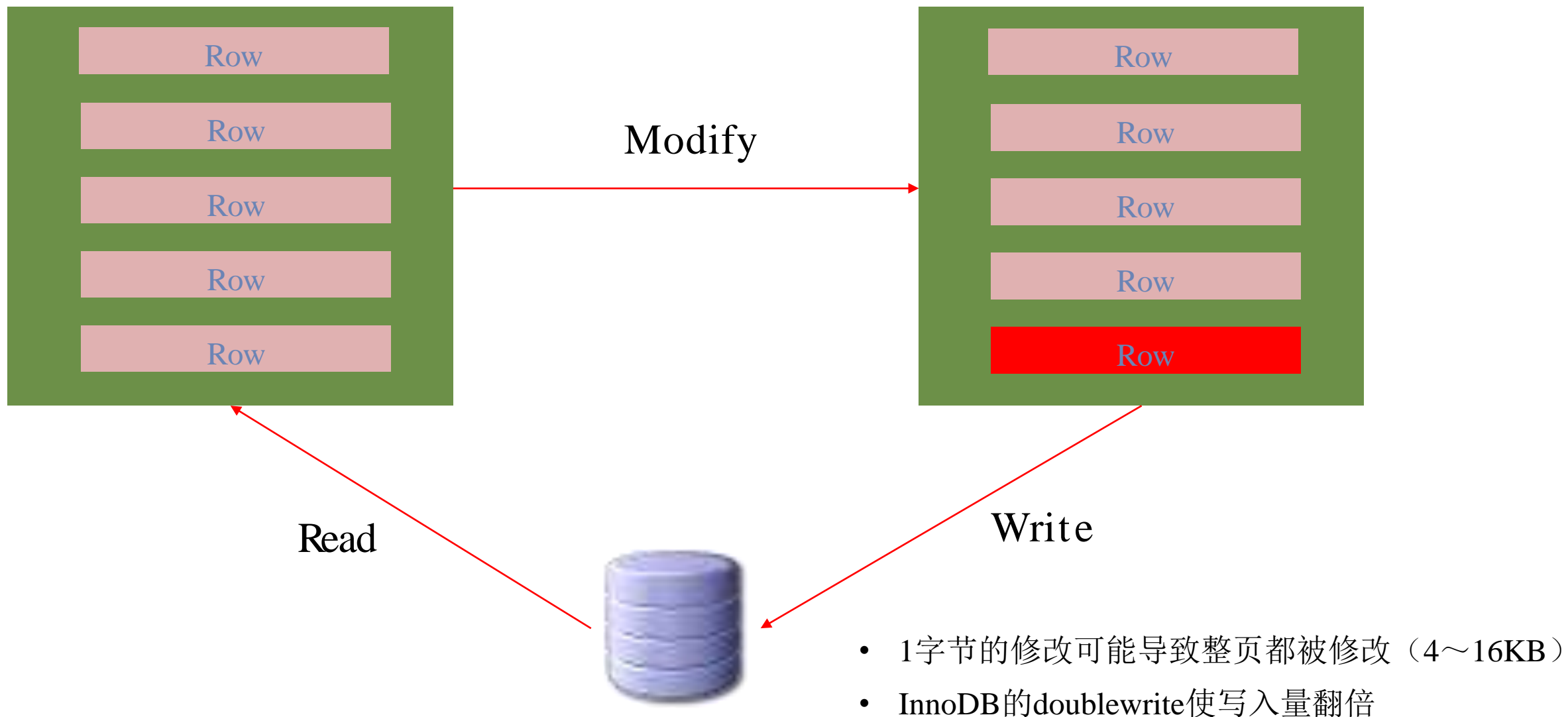
```
INSERT INTO message(user_id) VALUES(31);  
INSERT INTO message(user_id) VALUES(10000);  
...
```

以user_id为索引的B+树



- B+树索引的叶节点的page大小非常小（InnoDB通常为16KB）
- 随机修改 => 随机写，以及在没有缓存的情况下产生随机读
- N行的修改 => 最坏的情况下在每个索引上会有N个不同page的读和写

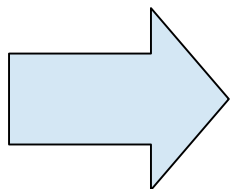
InnoDB的问题：写放大



InnoDB的问题：碎片

```
INSERT INTO message_table (user_id) VALUES (31)
```

Leaf Block 1	
user_id	RowID
1	10000
2	5
3	15321
...	
60	431



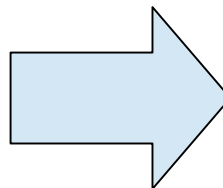
Leaf Block 1	
user_id	RowID
1	10000
...	
30	333
Empty	

Leaf Block 2	
user_id	RowID
31	345
...	
60	431
Empty	

InnoDB的问题：碎片

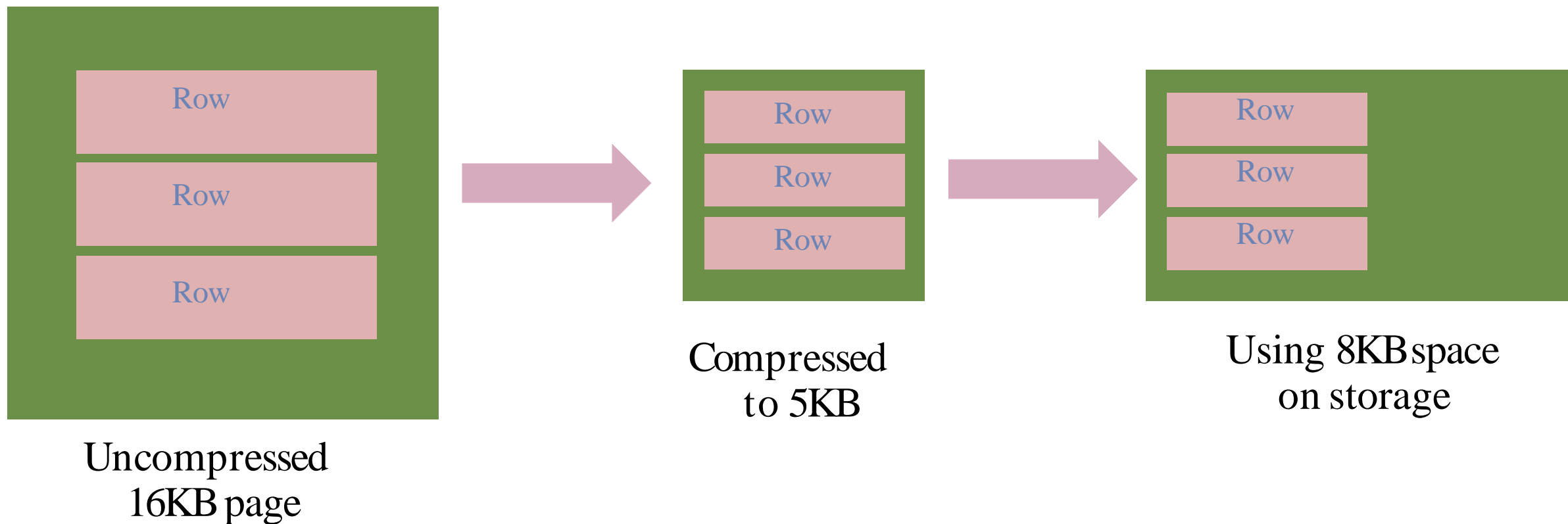
```
DELETE FROM message_table WHERE user_id=2;
```

Leaf Block 1	
user_id	RowID
1	10000
2	5
3	15321
...	
60	431



Leaf Block 1	
user_id	RowID
1	10000
3	15321
...	
60	431

InnoDB的问题：压缩



- InnoDB按页压缩。在MySQL5.7之前，page大小需要与4KB对齐，MySQL5.7之后，也要与OS/设备扇区对齐。

RocksDB

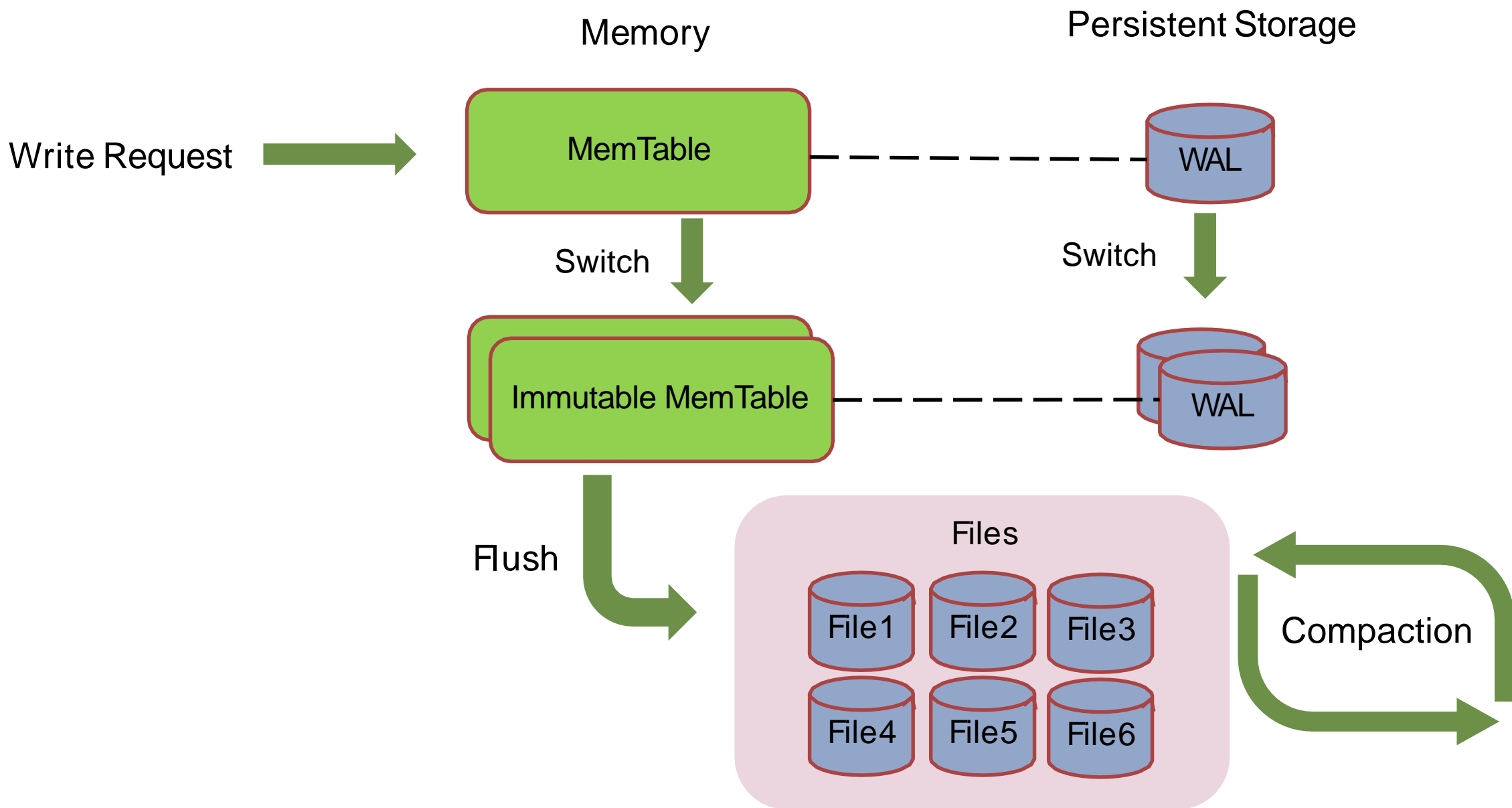
- <http://rocksdb.org/>
- Forked from LevelDB
 - Key-Value LSM (Log Structured Merge) persistent store
 - Embedded
 - Data stored locally
 - Optimized for fast storage
- LevelDB was created by Google
- Facebook forked and developed RocksDB
- Used at many backend services at Facebook, and many external large services
- Needs to write C++ or Java code to access RocksDB



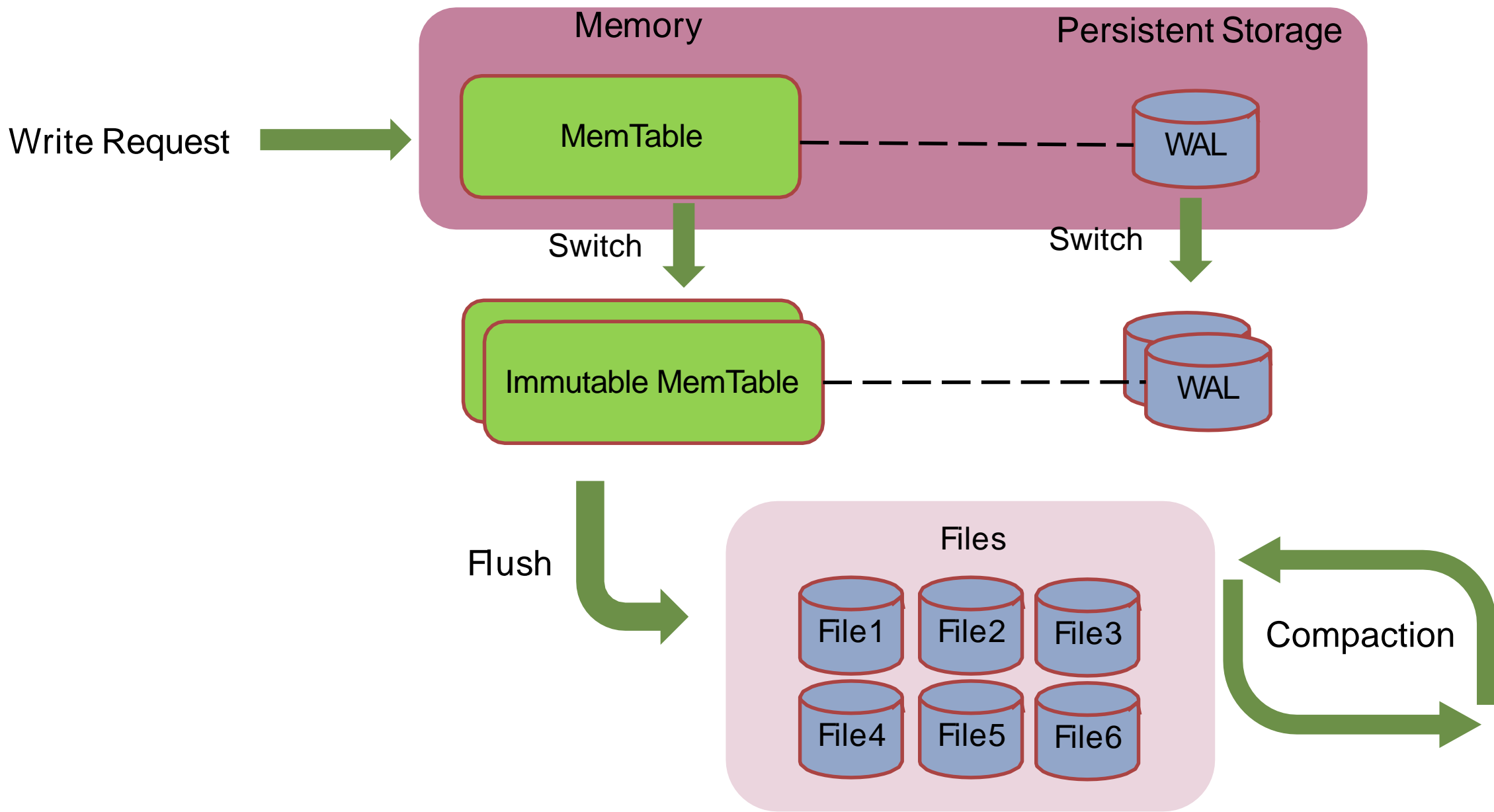
RocksDB体系结构

- 层级LSM树结构
- Memtable
- WAL（Write Ahead Log）
- 压缩（compaction）
- 列族（Column Family）

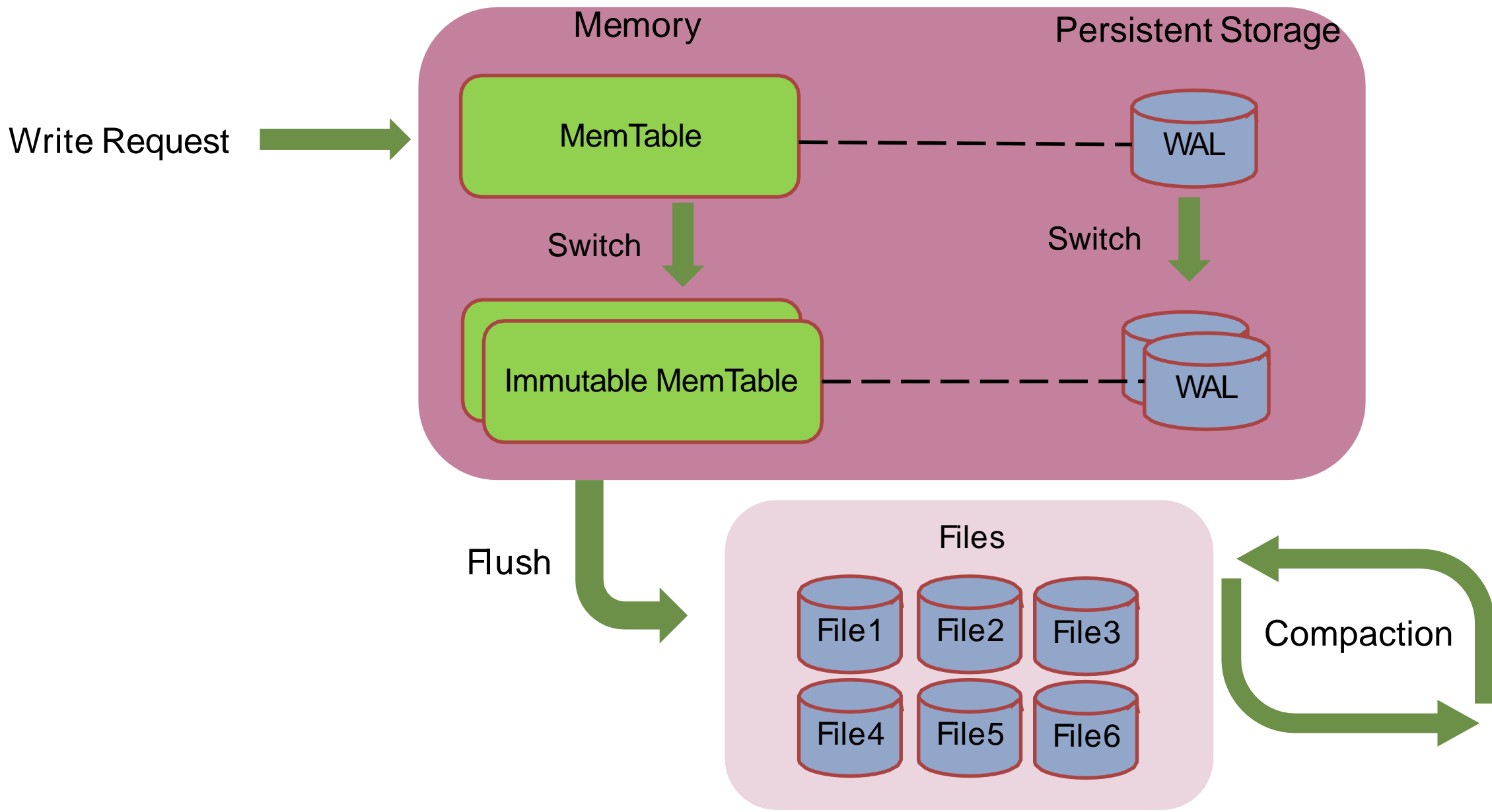
RocksDB的写入



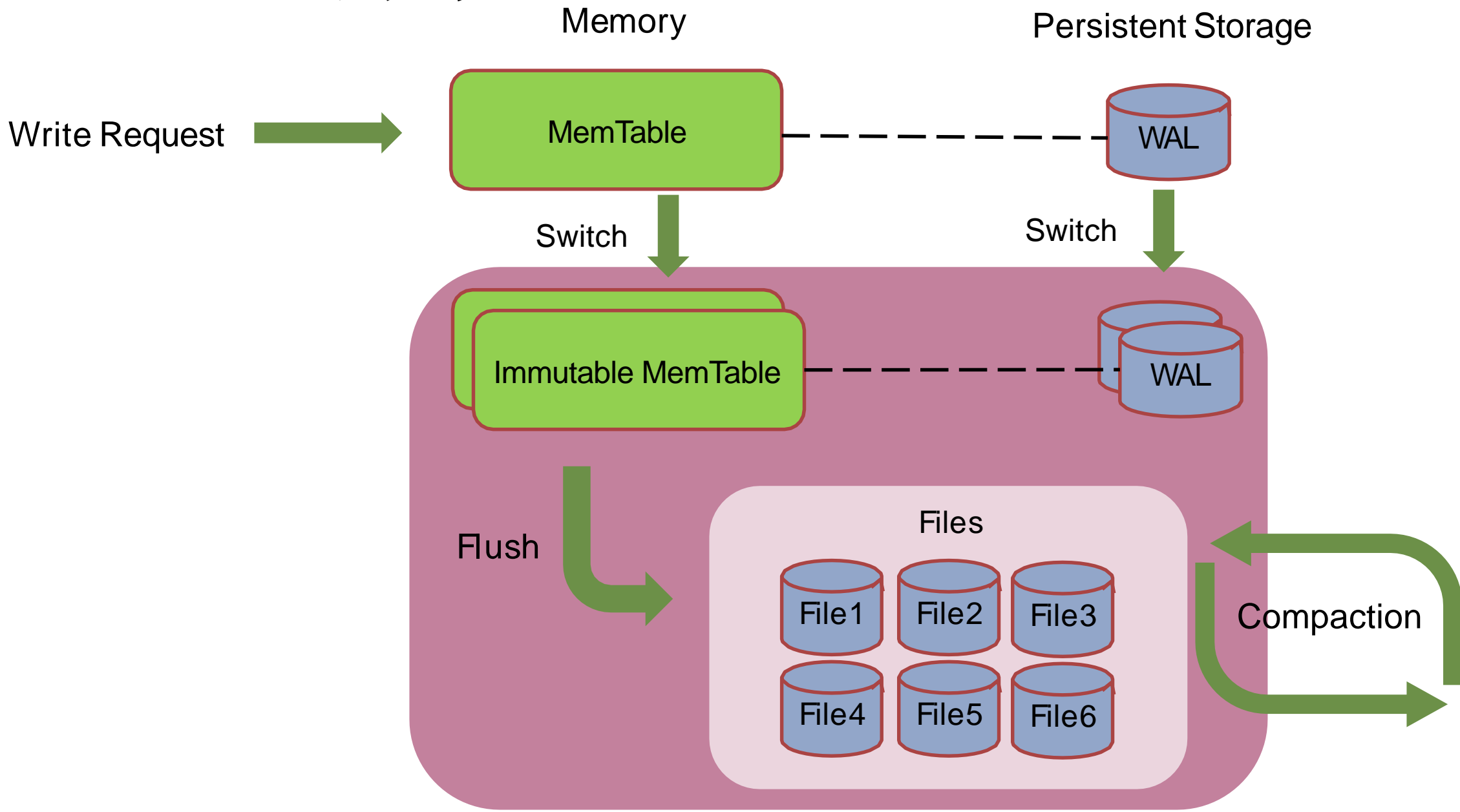
RocksDB的写入



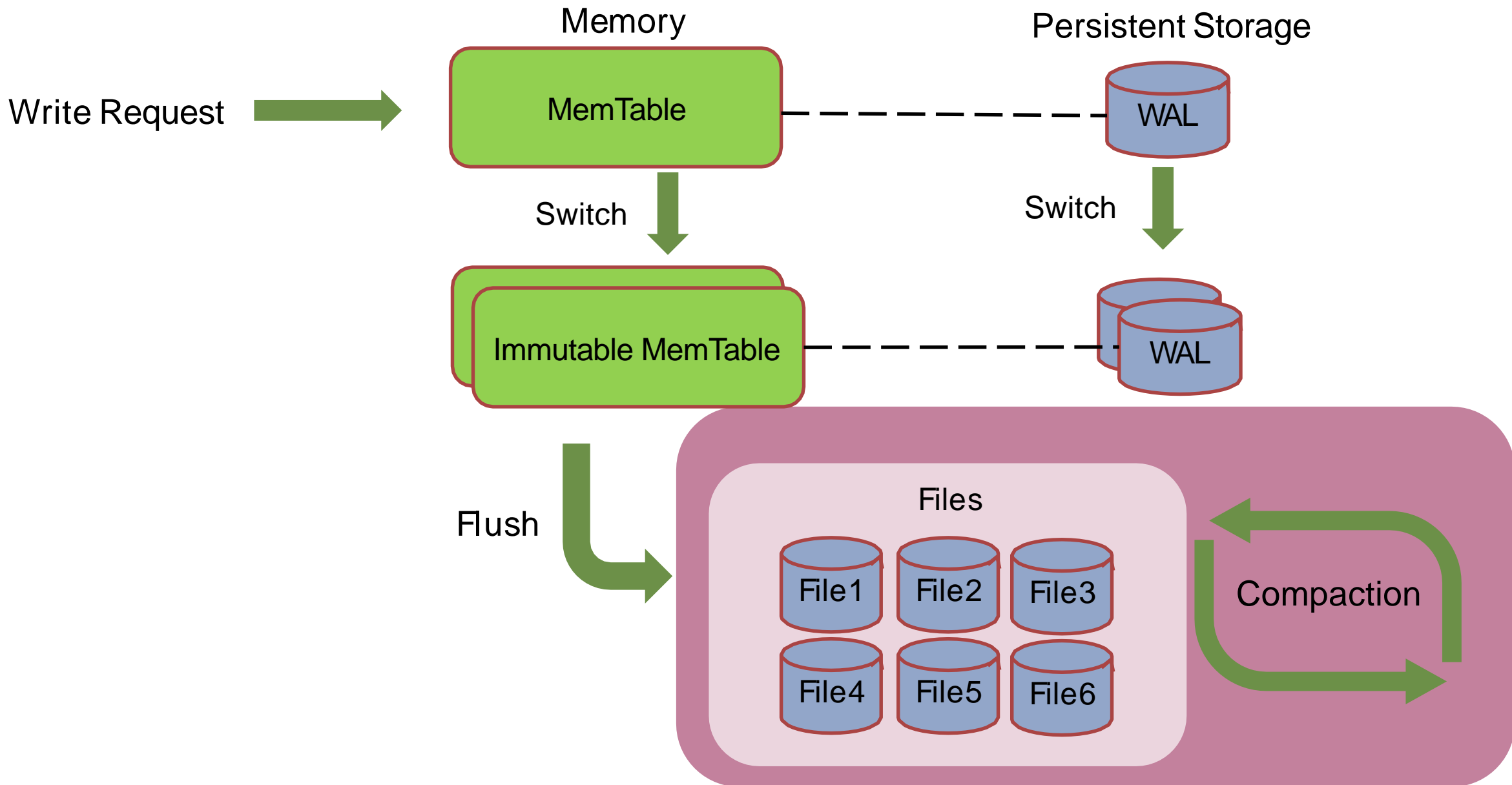
RocksDB的写入



RocksDB的写入

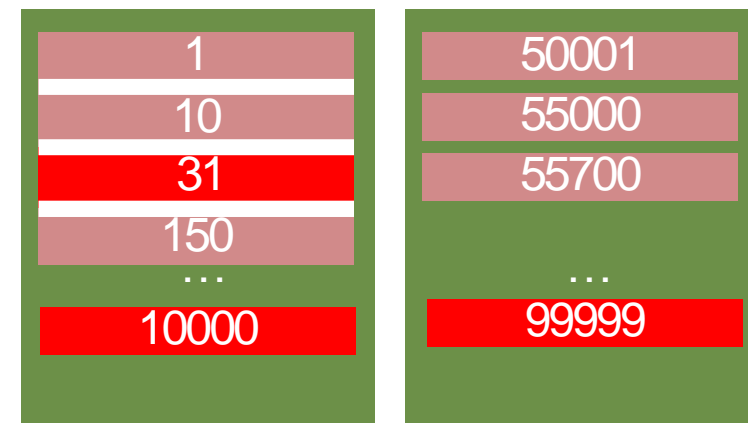
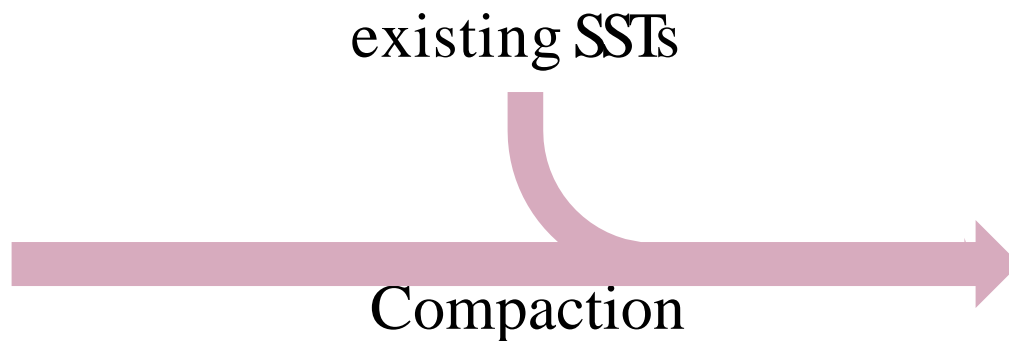
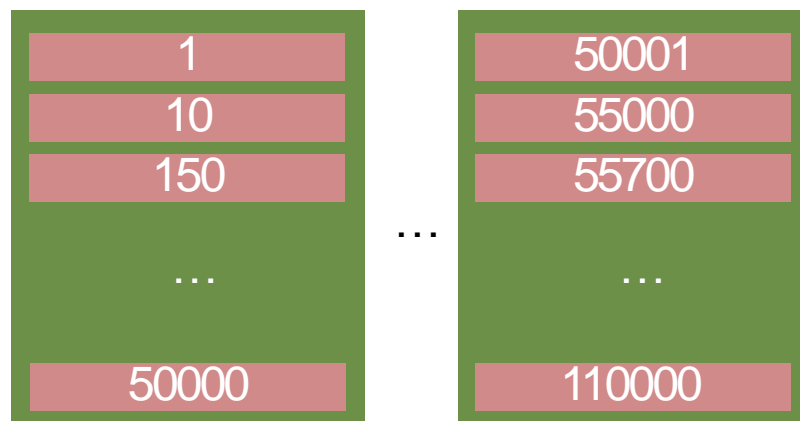
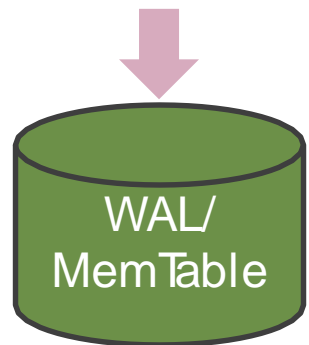


RocksDB的写入



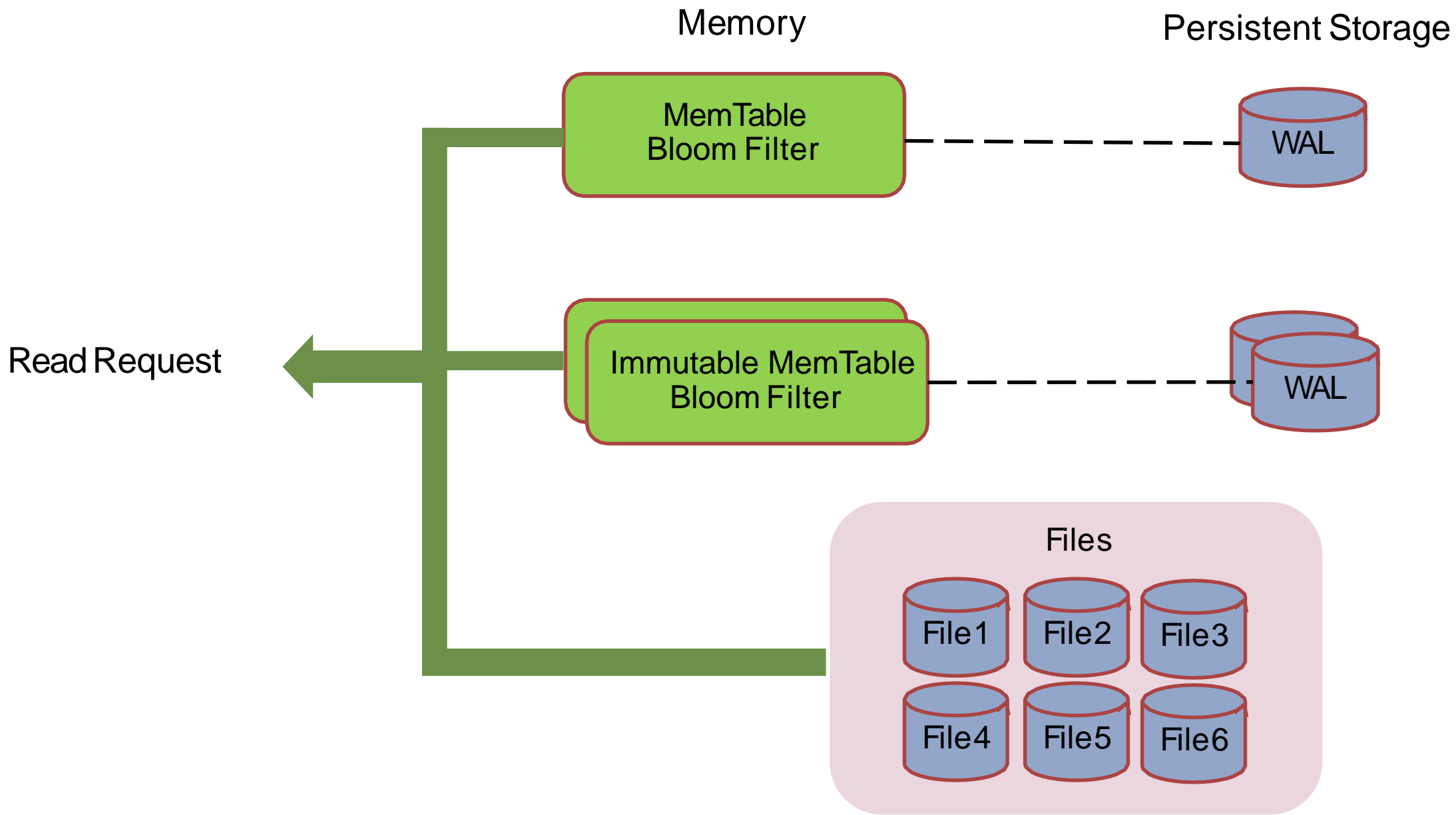
RocksDB的写入

```
INSERT INTO message (user_id) VALUES (31);  
INSERT INTO message (user_id) VALUES (99999);  
INSERT INTO message (user_id) VALUES (10000);
```

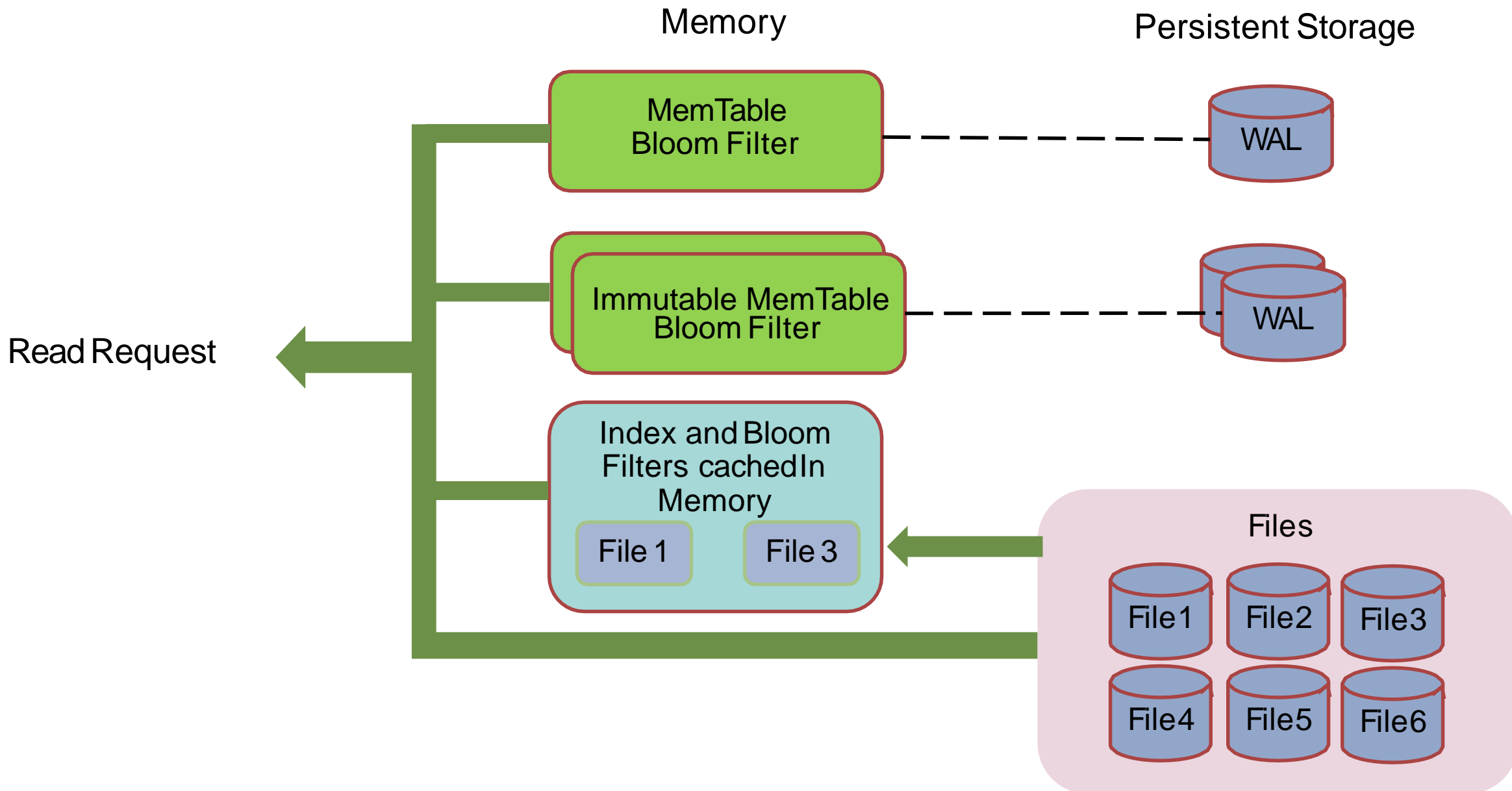


N random row modifications => A few sequential reads & writes

RocksDB的读取



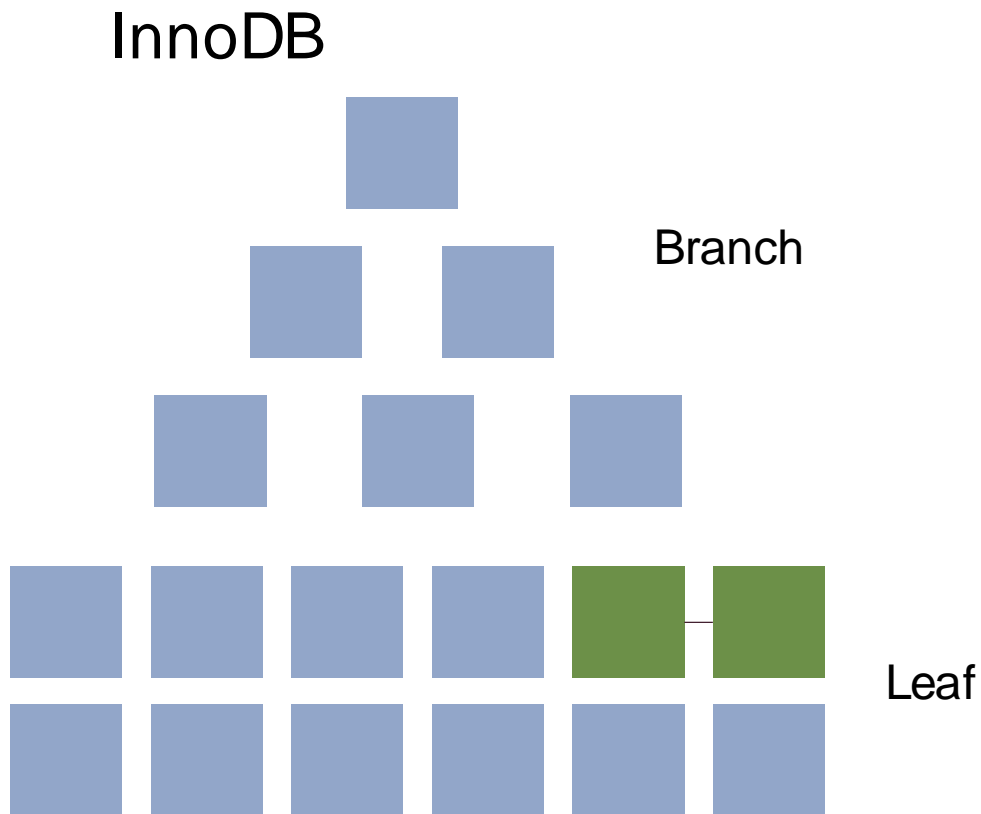
RocksDB的读取



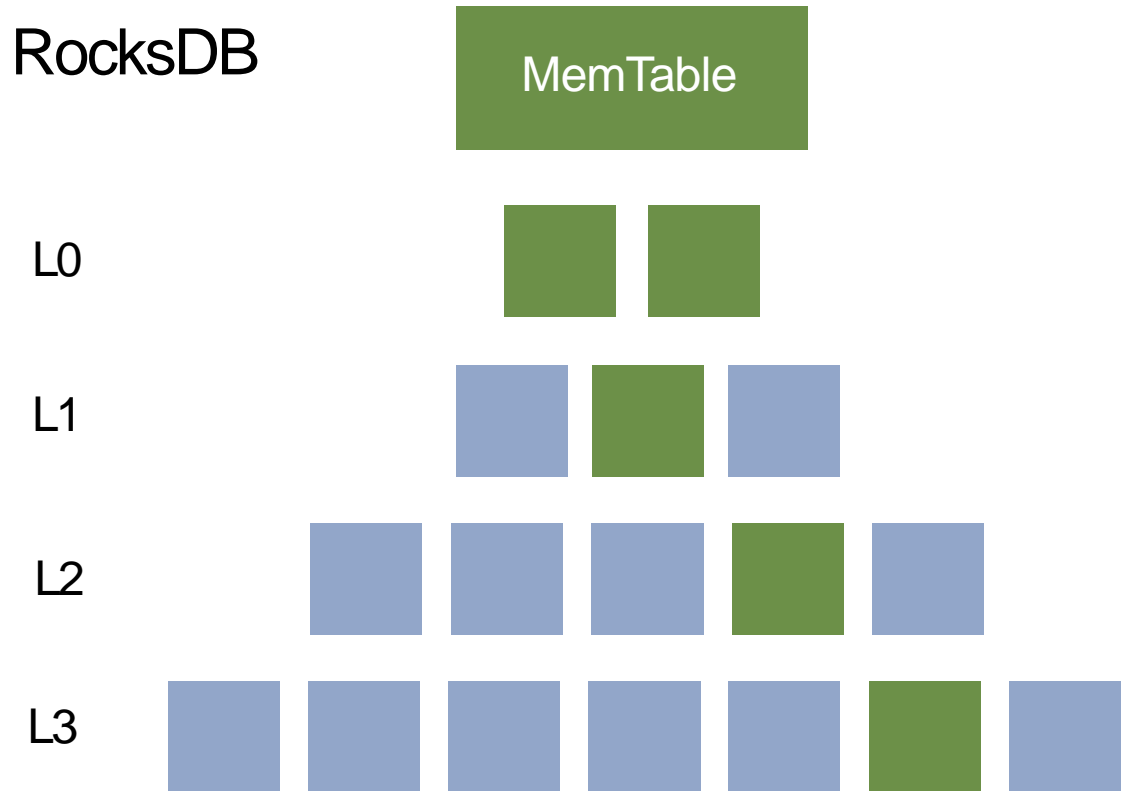
RocksDB的读代价

```
SELECT id1, id2, time FROM t WHERE id1=100 AND id2=100 ORDER BY time DESC LIMIT 1000;
```

Index on (id1, id2, time)



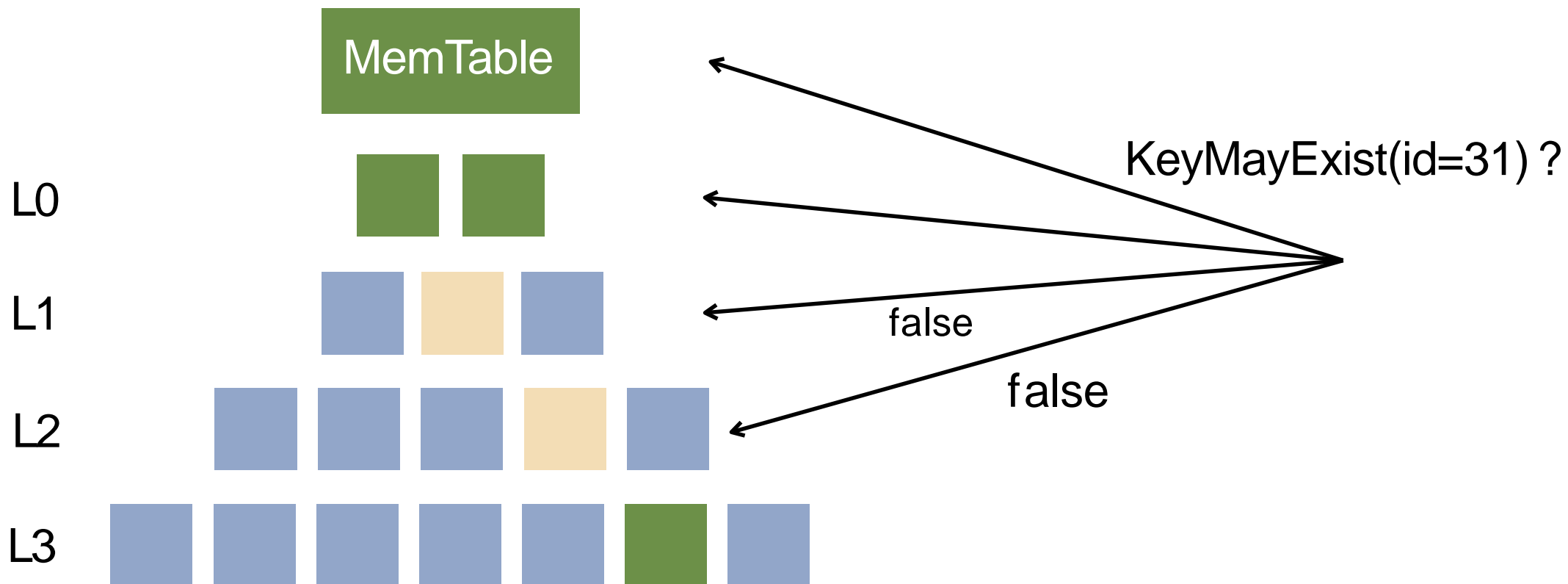
范围扫描+覆盖索引
只需要顺序读取叶子结点
且能保证顺序（效率极高）



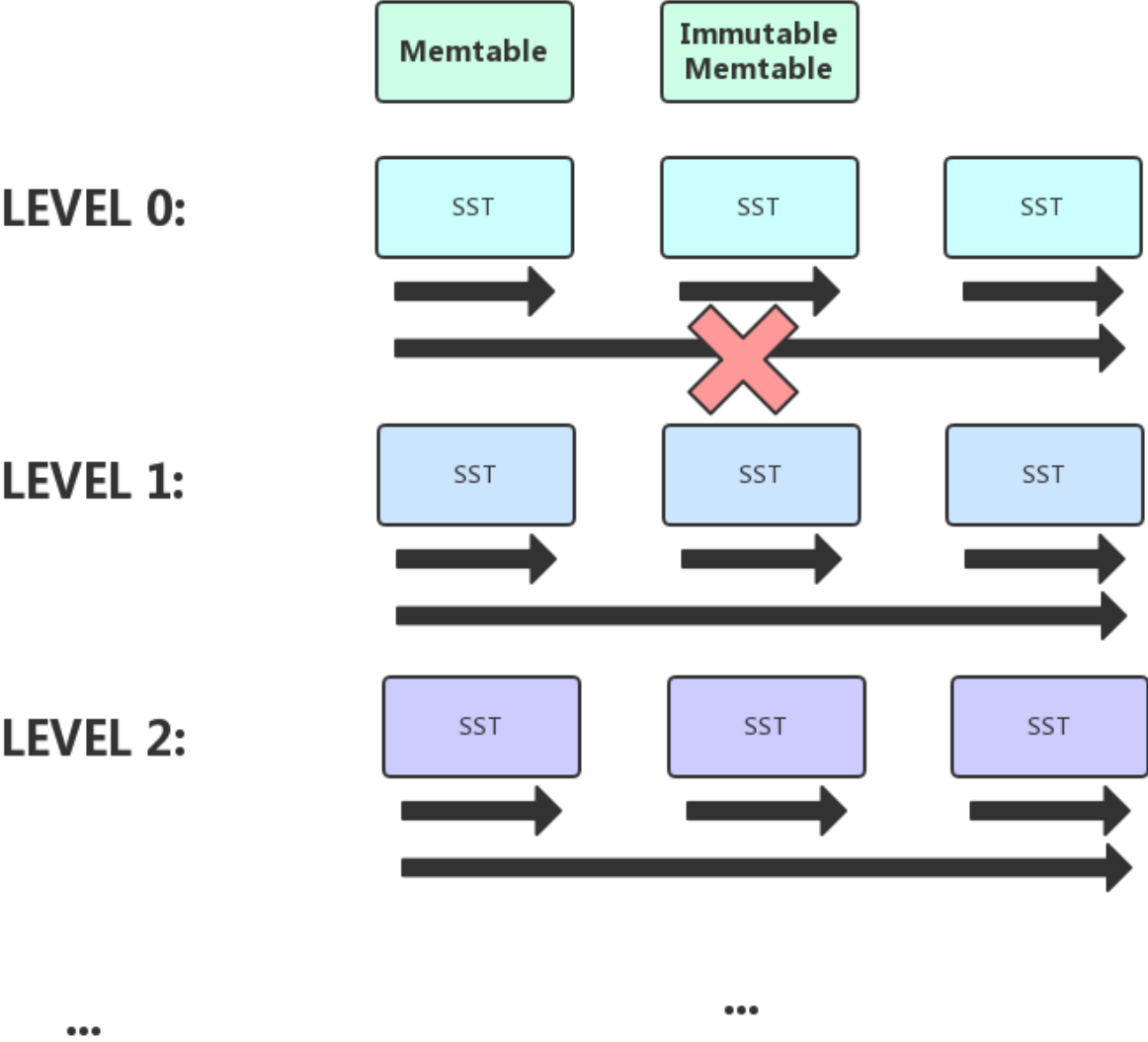
范围扫描+ORDER BY需要Merge
需要遍历所有的level
需要占用更多的CPU

Bloom Filter

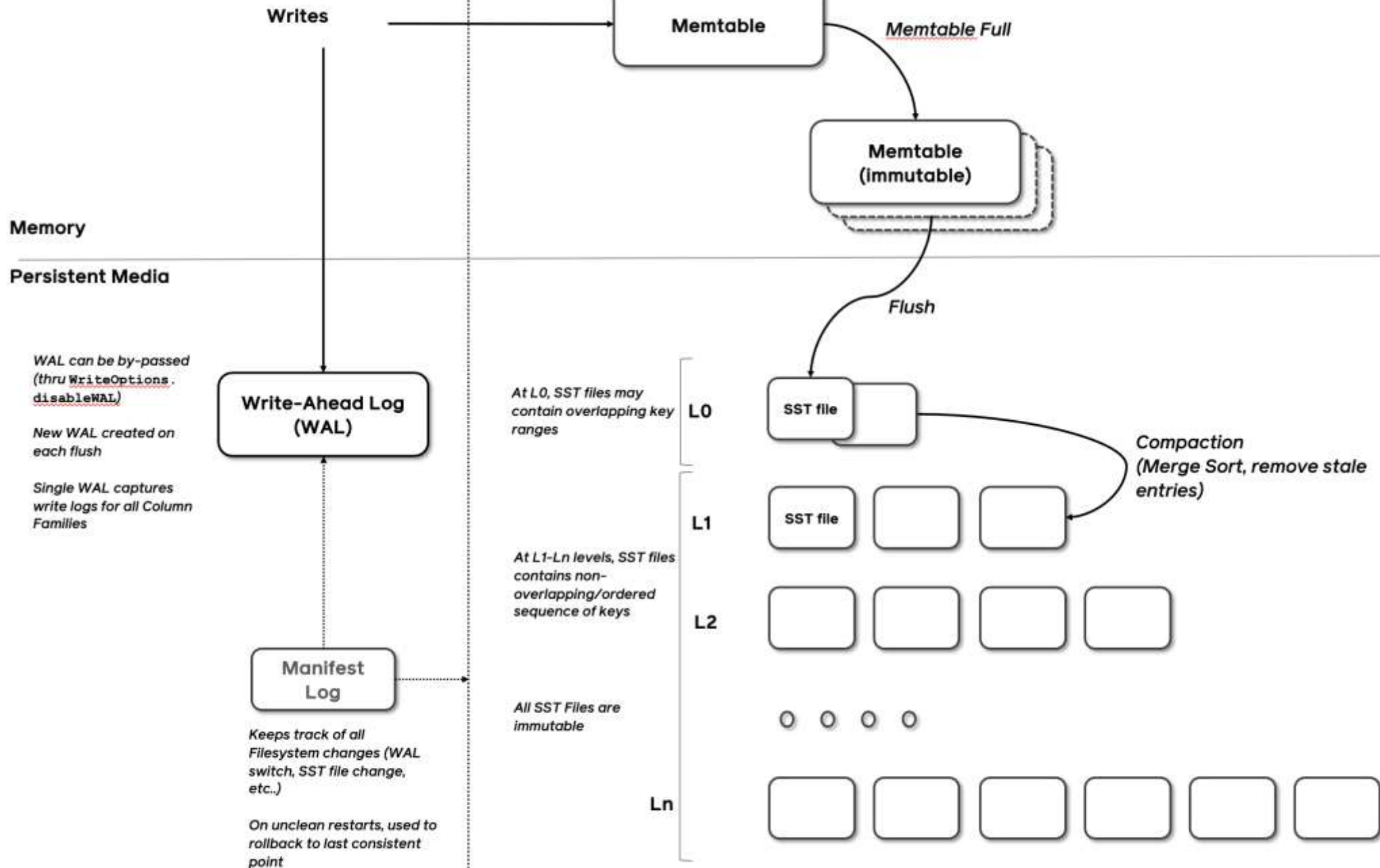
无需读取数据来判断数据是否 **【不存在】**
跳过读IO



RocksDB层级图

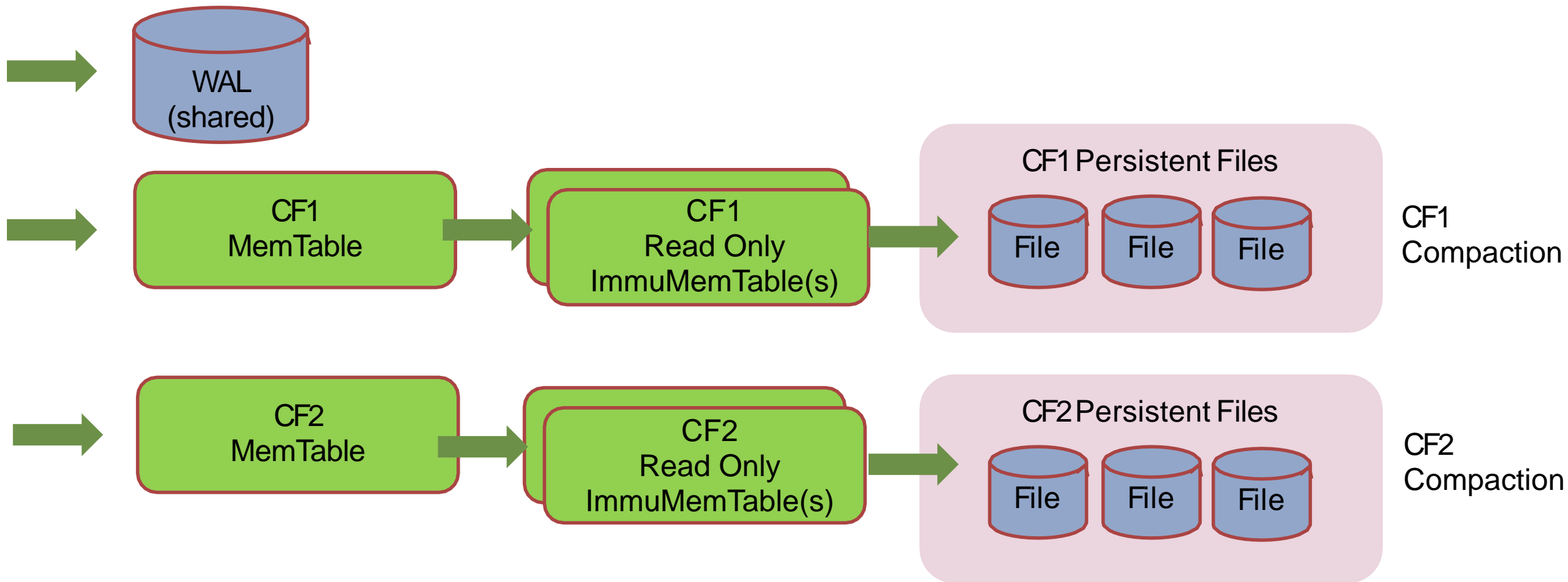


For each Column Family

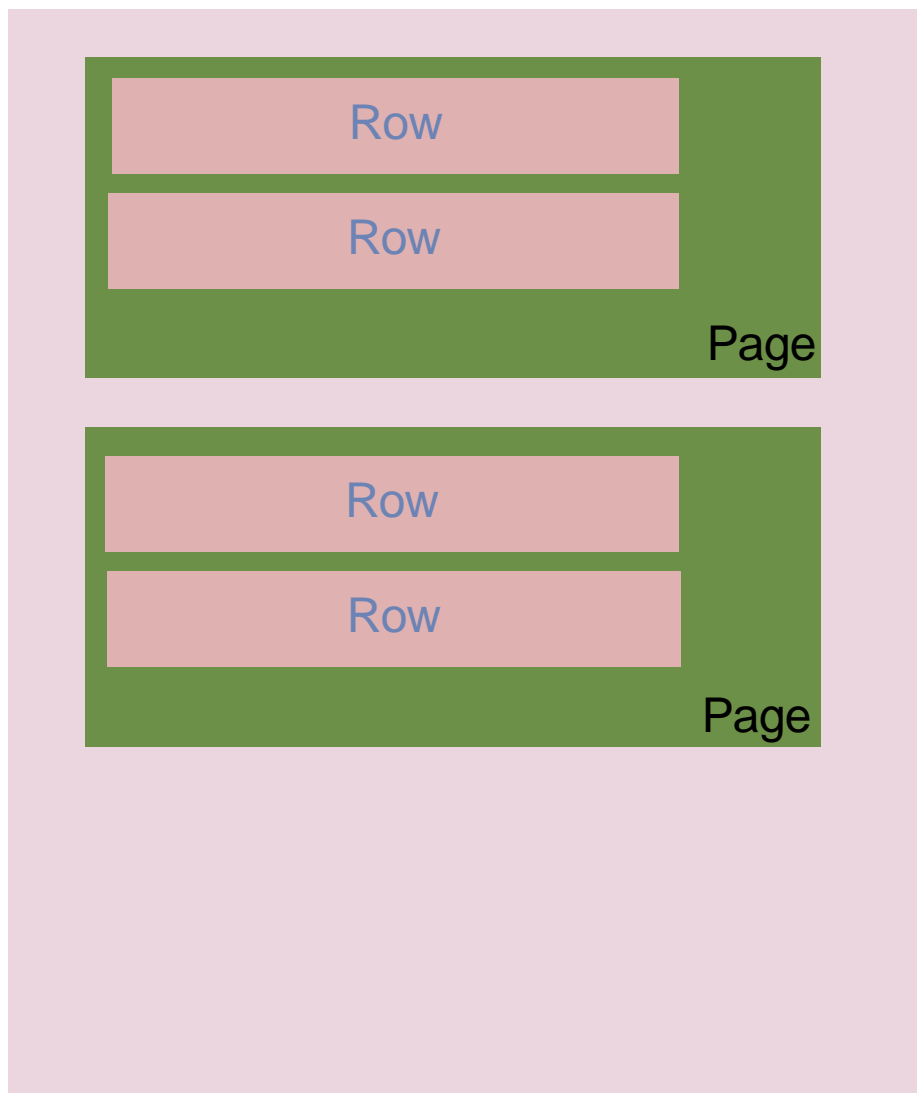


列族Column Family

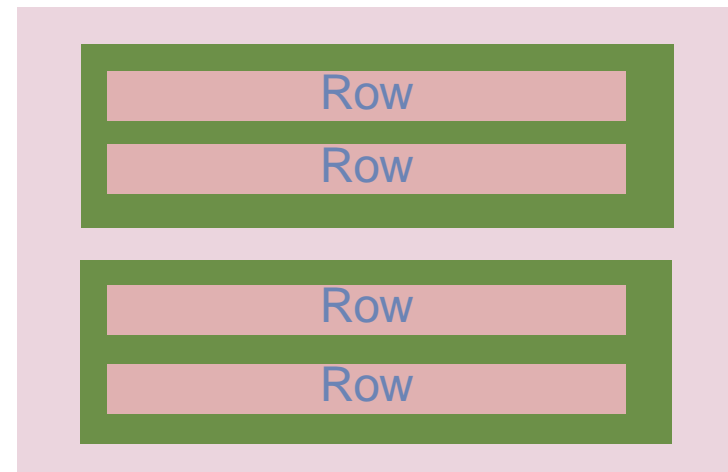
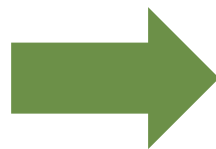
- 单独的MemTables与SST文件
- 共享的WAL



Compression In RocksDB



16MB SSTFile

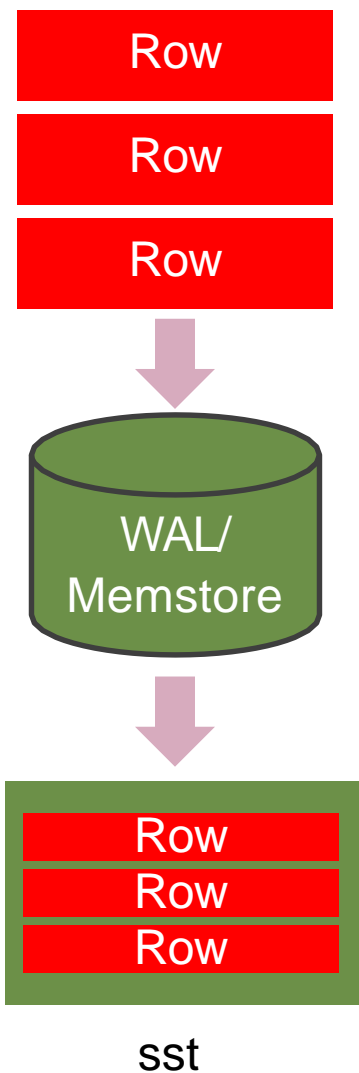


5MB SST File

- 压缩后的page无需与OS扇区对齐
- SST File大小需要与扇区对其，但SST File大小会大得多，所以对齐的开销可以忽略


减少写放大与空间放大

Append Only



Prefix Key Encoding


id1	id2	id3
100	200	1
100	200	2
100	200	3
100	200	4



id1	id2	id3
100	200	1
		2
		3
		4

Zero-Filling Metadata

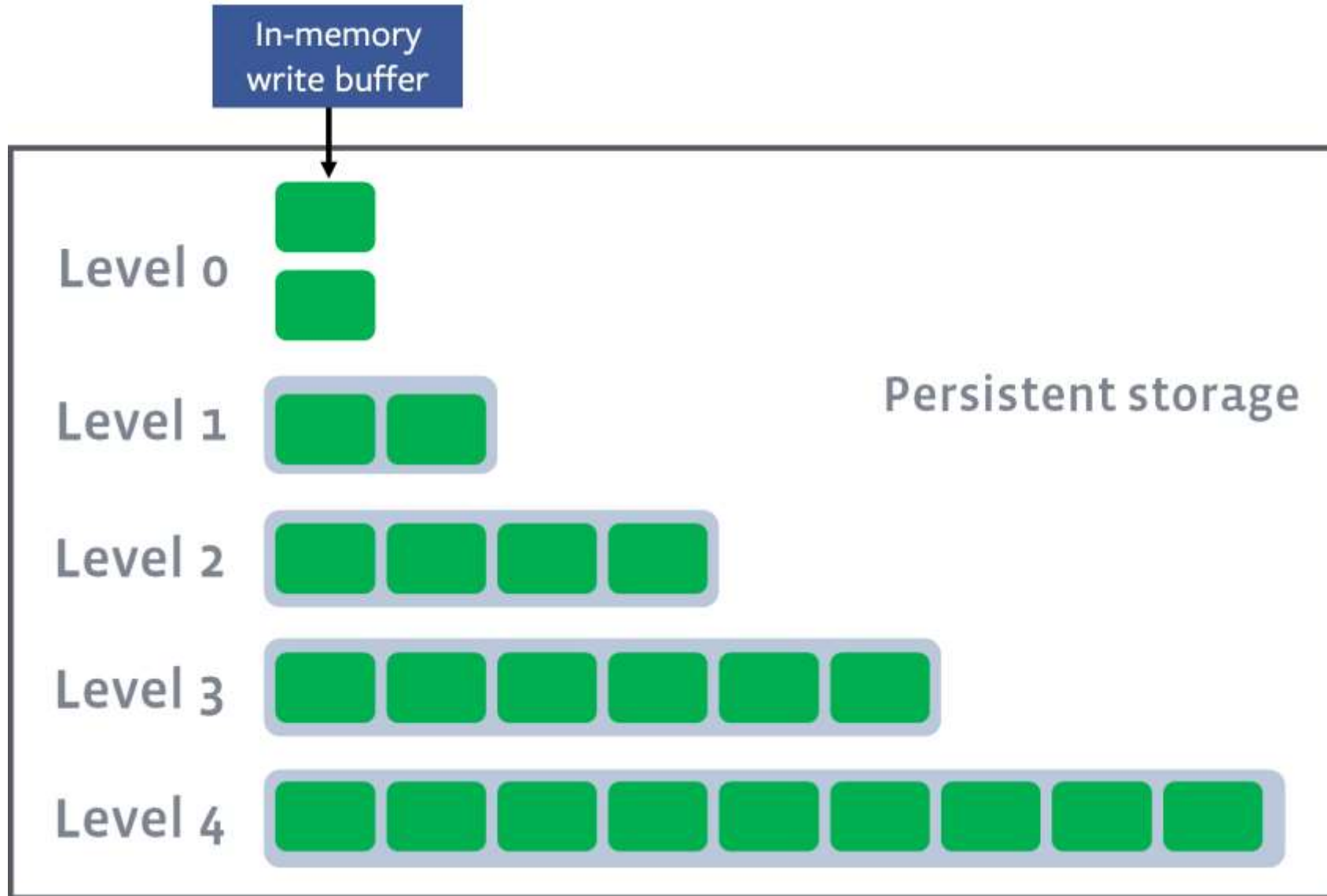
key	value	seq id	flag
k1	v1	1234561	W
k2	v2	1234562	W
k3	v3	1234563	W
k4	v4	1234564	W



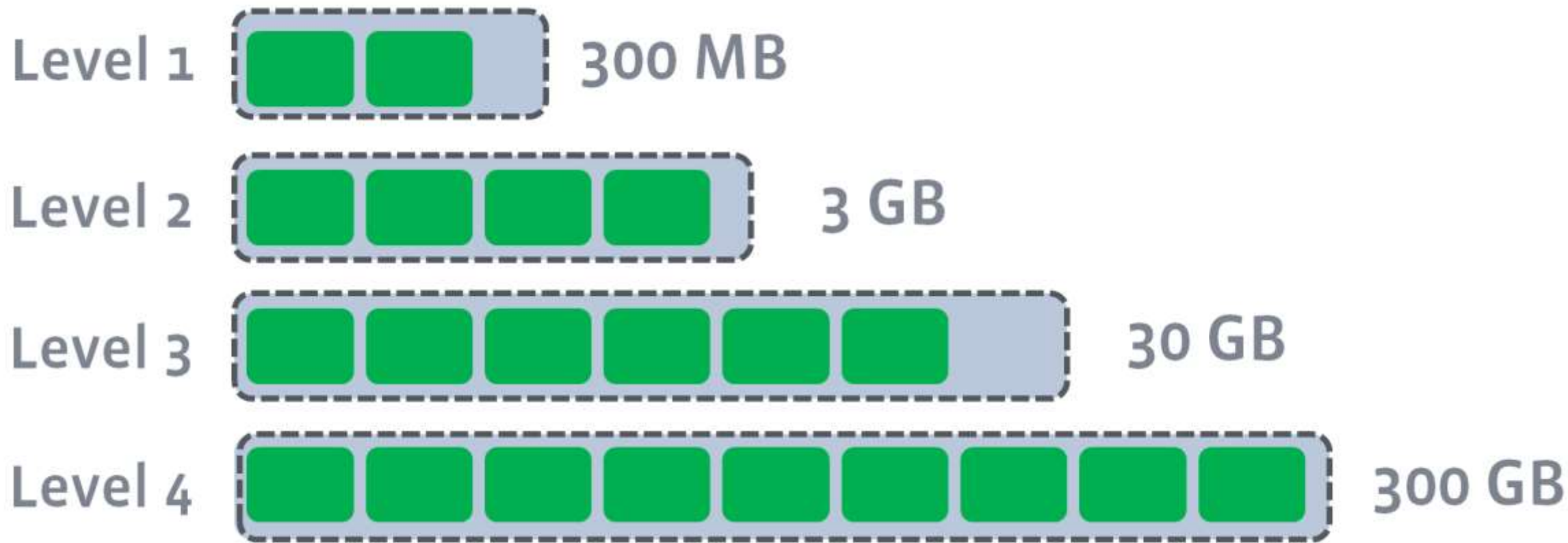
key	value	seq id	flag
k1	v1	0	W
k2	v2	0	W
k3	v3	0	W
k4	v4	0	W

seq id在RocksDB中占用七个字节，压缩后，“0”占用很少的空间

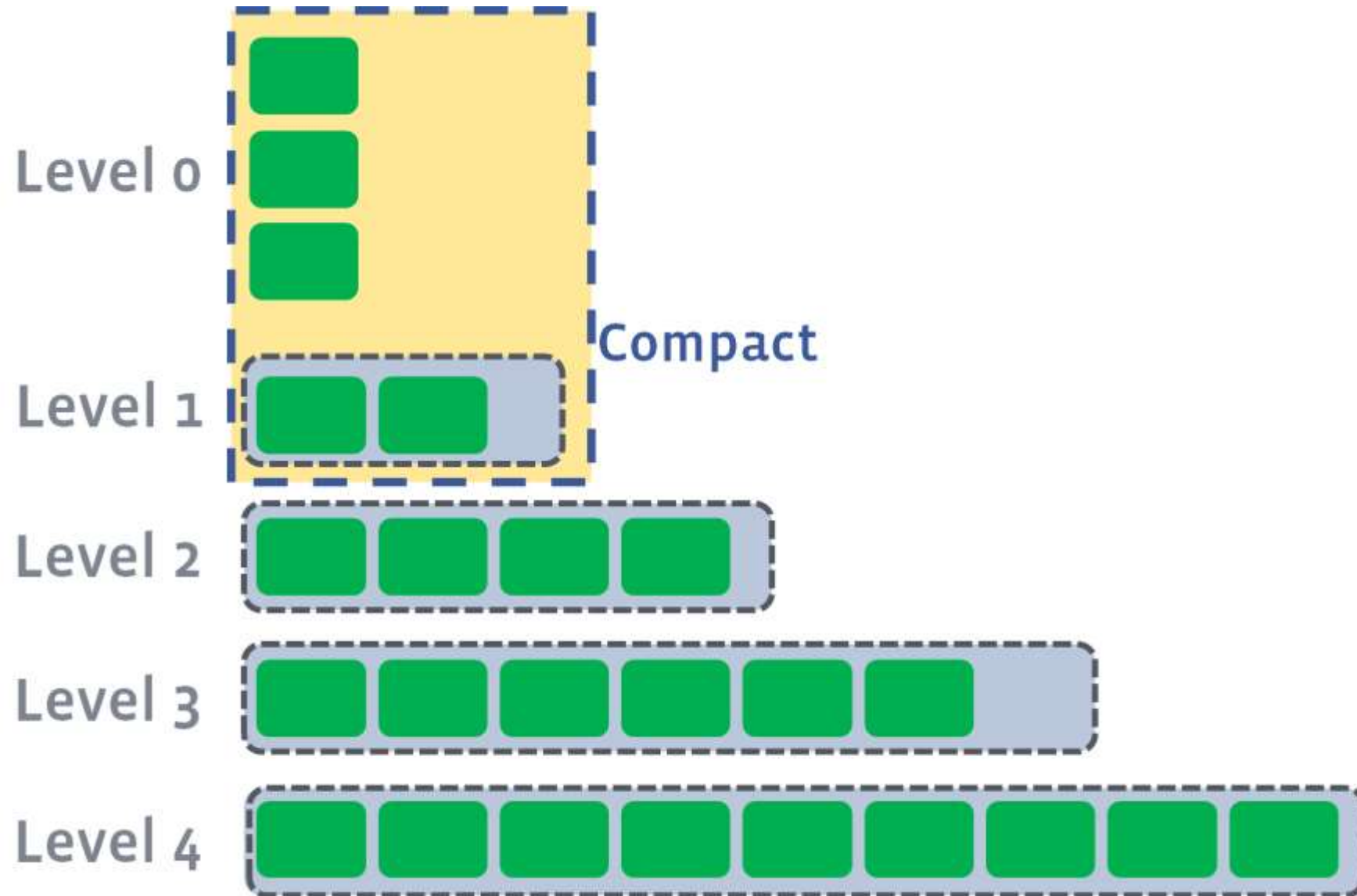
Compaction



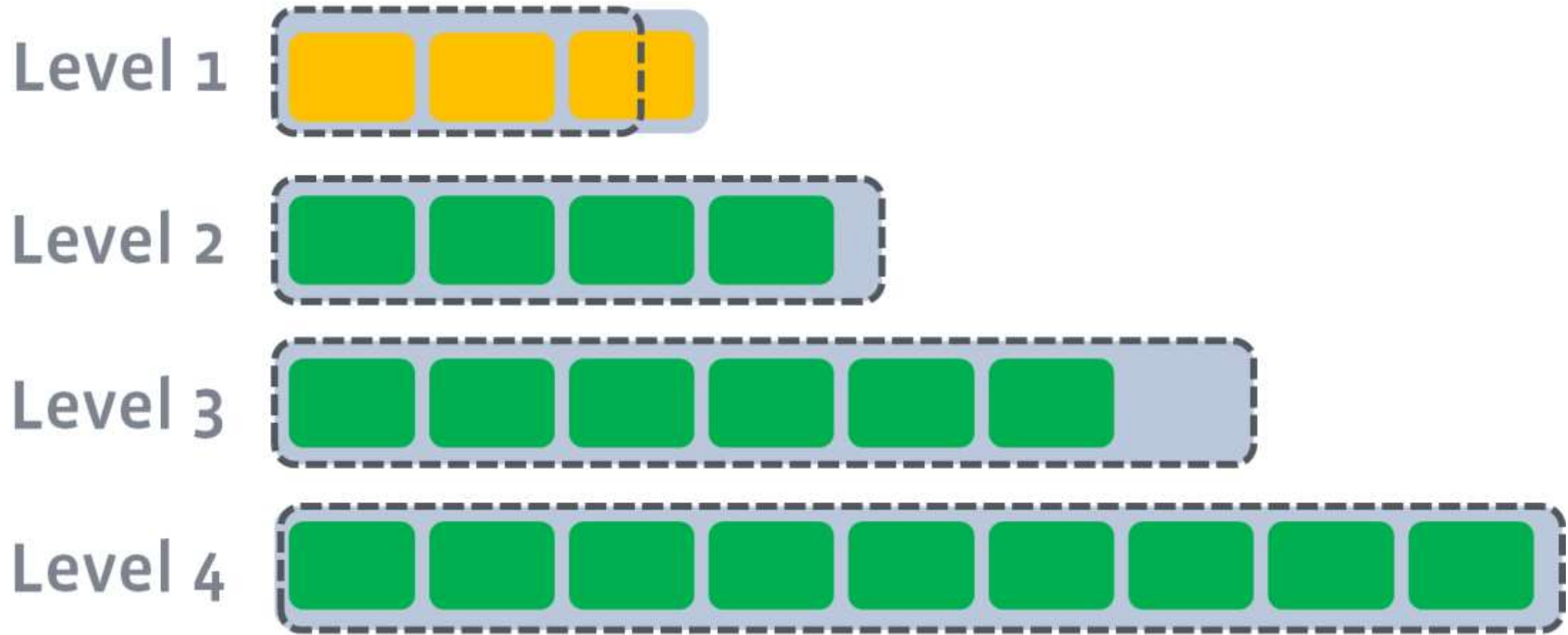
Compaction



Compaction

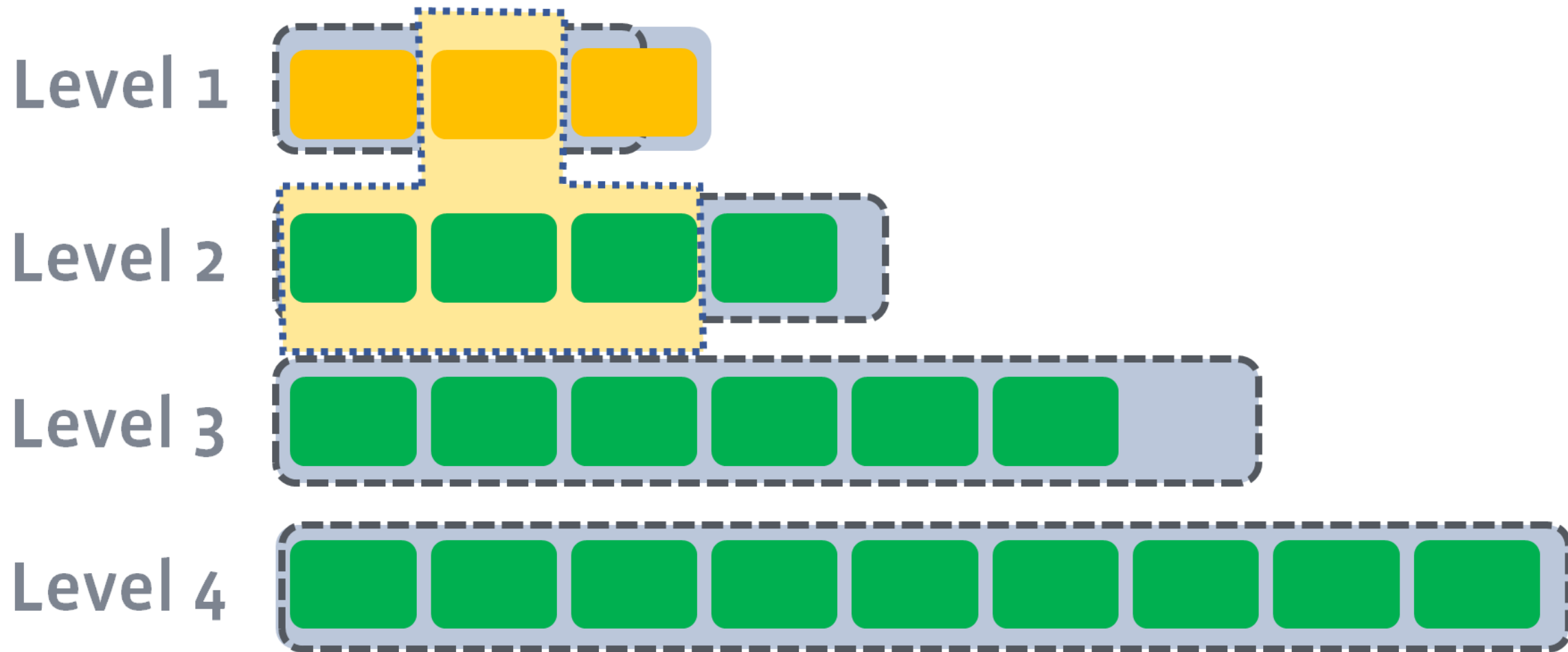


Compaction

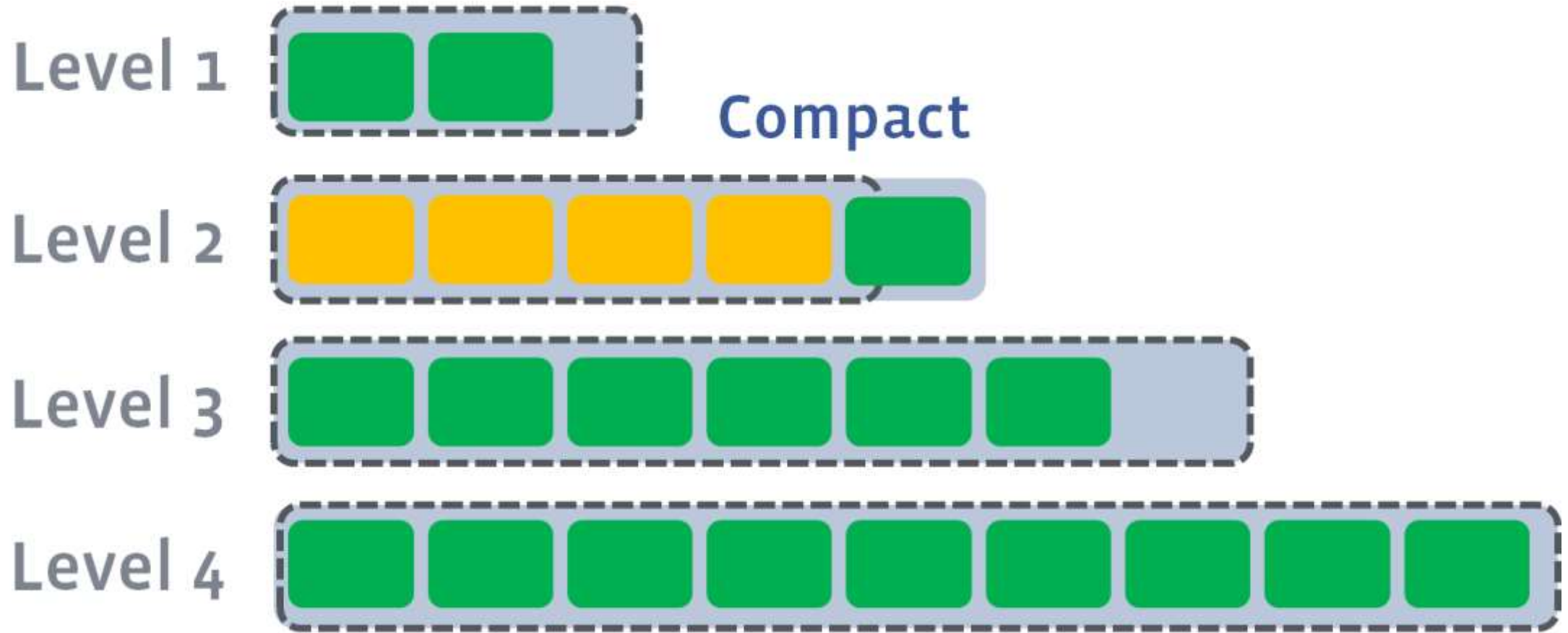


Compaction

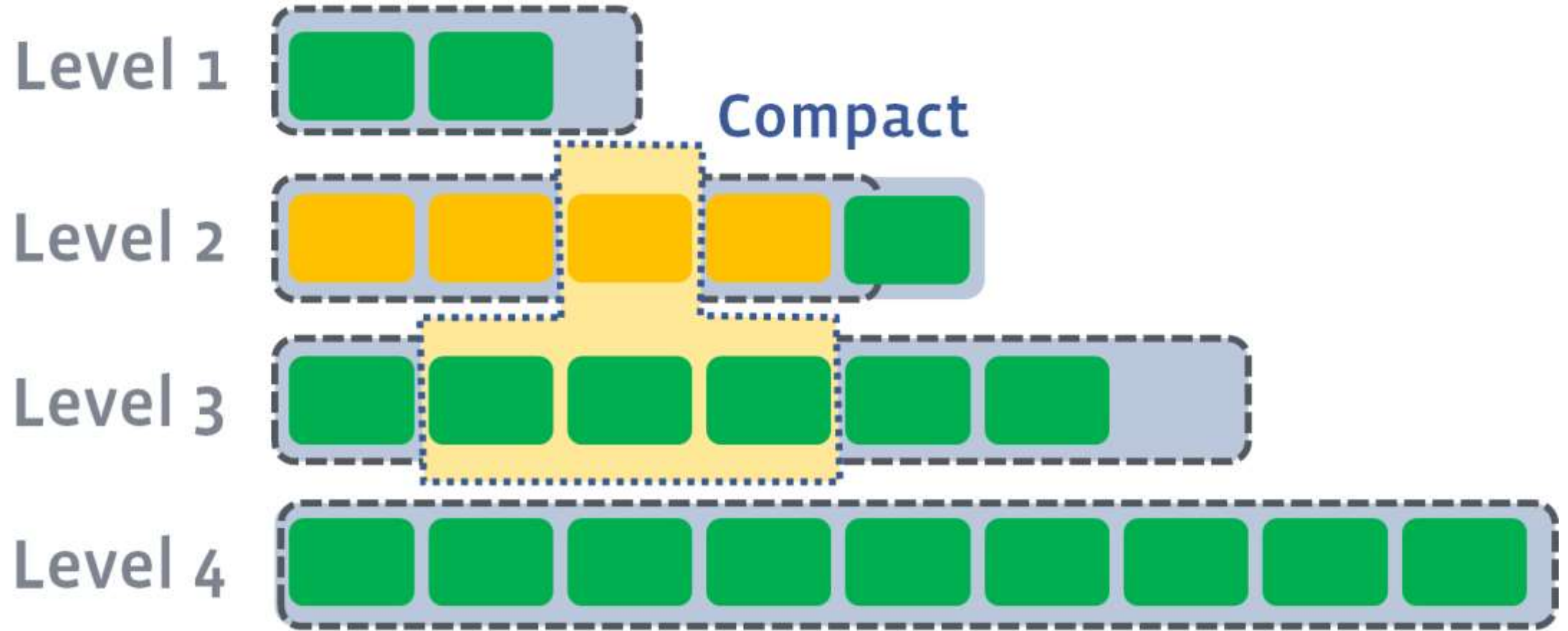
Compact



Compaction

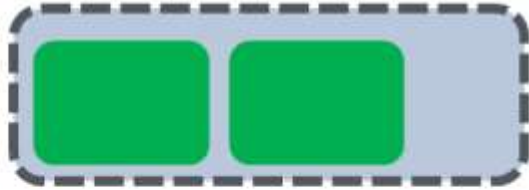


Compaction

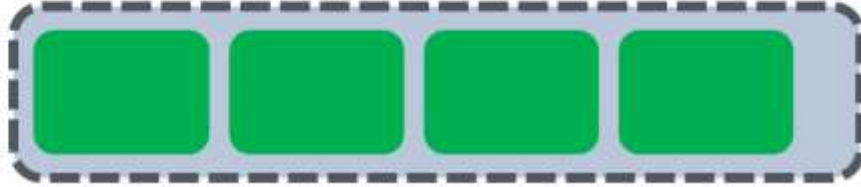


Compaction

Level 1



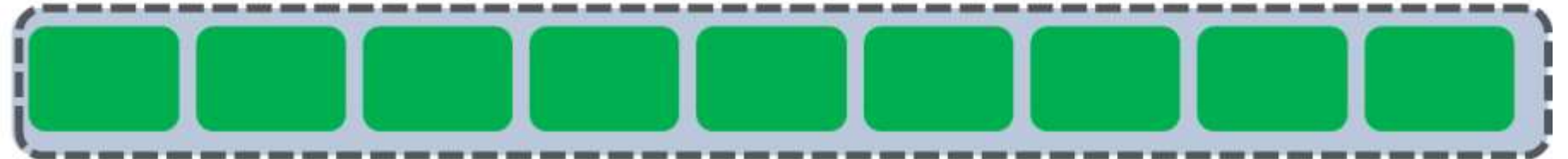
Level 2



Level 3



Level 4

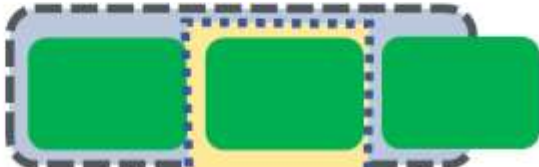


Compaction

Level 0



Level 1



Level 2

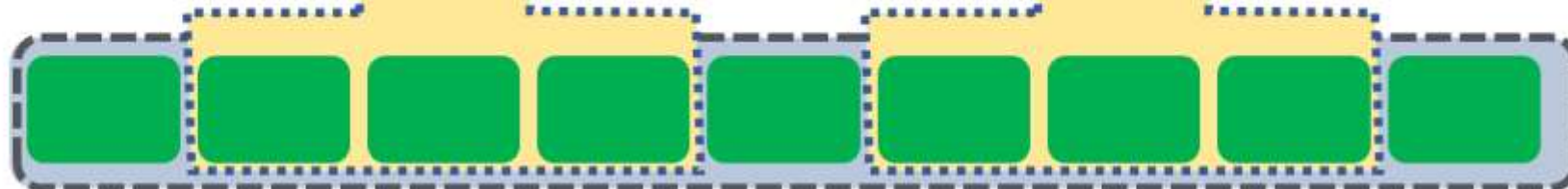


Compact

Level 3



Level 4



MyRocks数据结构和数据库设计

- 支持主键和二级索引键
- 主键索引为聚簇索引
 - 类似于InnoDB
- 索引的COMMENT定义了列族
- 不支持全文索引、外键和空间索引
- 不支持表空间
- Online DDL尚未支持

MyRocks K-V格式

- 主键索引
 - Key
 - 4byte Internal Index ID（自动分配）
 - 主键列打包在一起
 - 如果未定义，会生成隐藏的主键
 - Value
 - 其它列打包在一起
 - 记录的checksum（可选）
- 二级索引
 - Key
 - 4byte Internal Index ID（自动分配）
 - 二级索引列打包在一起
 - 主键索引打包在一起（可能存在重复的键）
 - Value
 - 记录的checksum（可选）

Primary Key:

RocksDB Key		RocksDB Value		Rocks Metadata
Internal IndexID	Primary Key	The rest columns	Checksum	SeqID, Flag

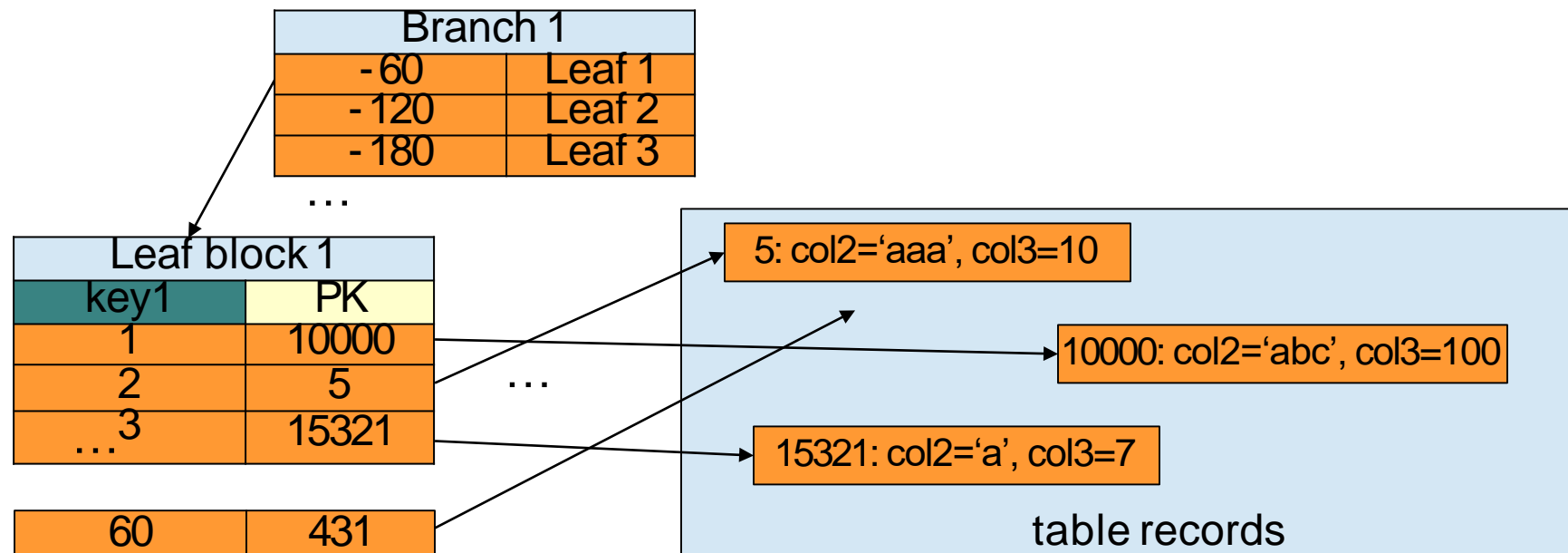
Secondary Key:

RocksDB Key			Rocks Value	Rocks Metadata
Internal IndexID	Secondary Key	Primary Key	Checksum	SeqID, Flag

索引查找/扫描的效率

- 与InnoDB十分类似
 - 如果走主键索引，那么一步就能得到所需数据
 - 覆盖索引：如果使用辅助键的查询仅涉及辅助键+主键字段，则它们不会读取主键（无需额外查找）
 - InnoDB 和 MyRocks 都支持覆盖索引
- 需要避免非覆盖的二级索引查找

```
SELECT* FROMtblWHERE key1 <2000000
```

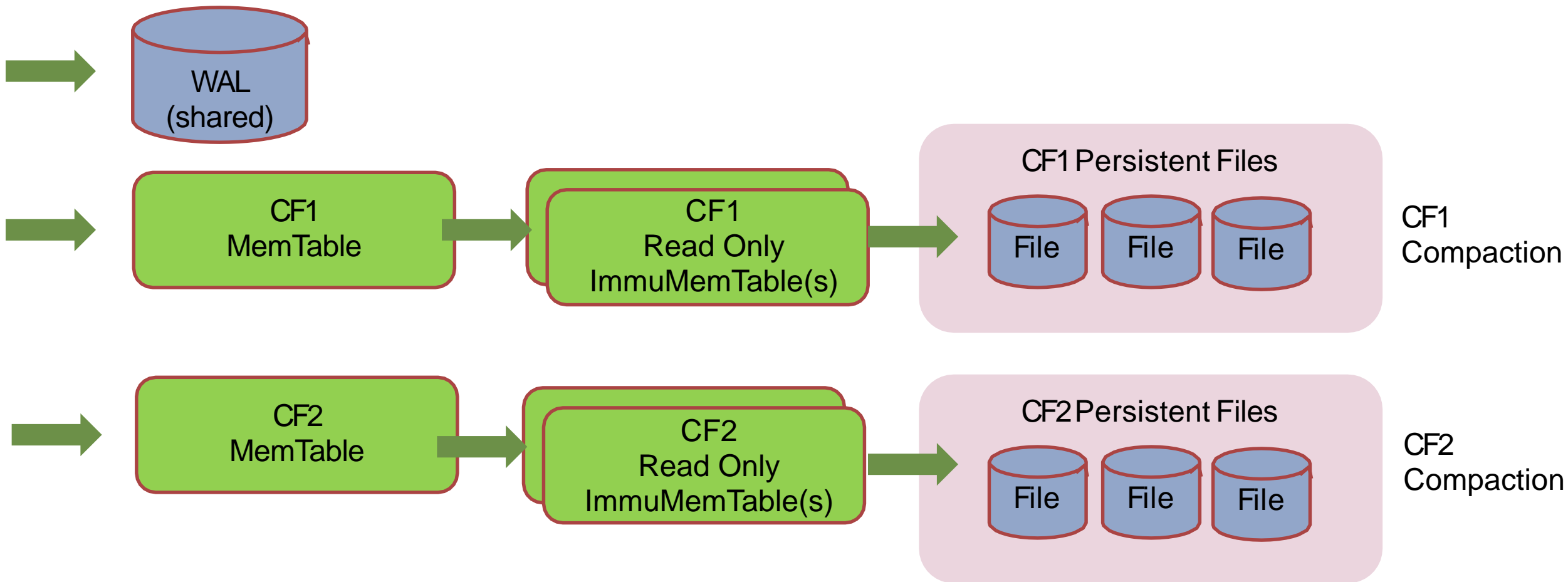


索引与列族

- Column Family 和 MyRocks Index 的映射关系是 1:N
 - 每个 MyRocks 索引属于一个列族
 - 多个索引可以属于同一个列族
 - 如果在 DDL 中没有指定，则该索引属于“默认”列族
- 大多数 RocksDB 配置参数是按列族的
 - Memtable、布隆过滤器等
- 不同类型的索引应该分配给不同的 Column Family
- 不要创建太多列族
 - ~20 足够
- INDEX COMMENT 指定关联的列族

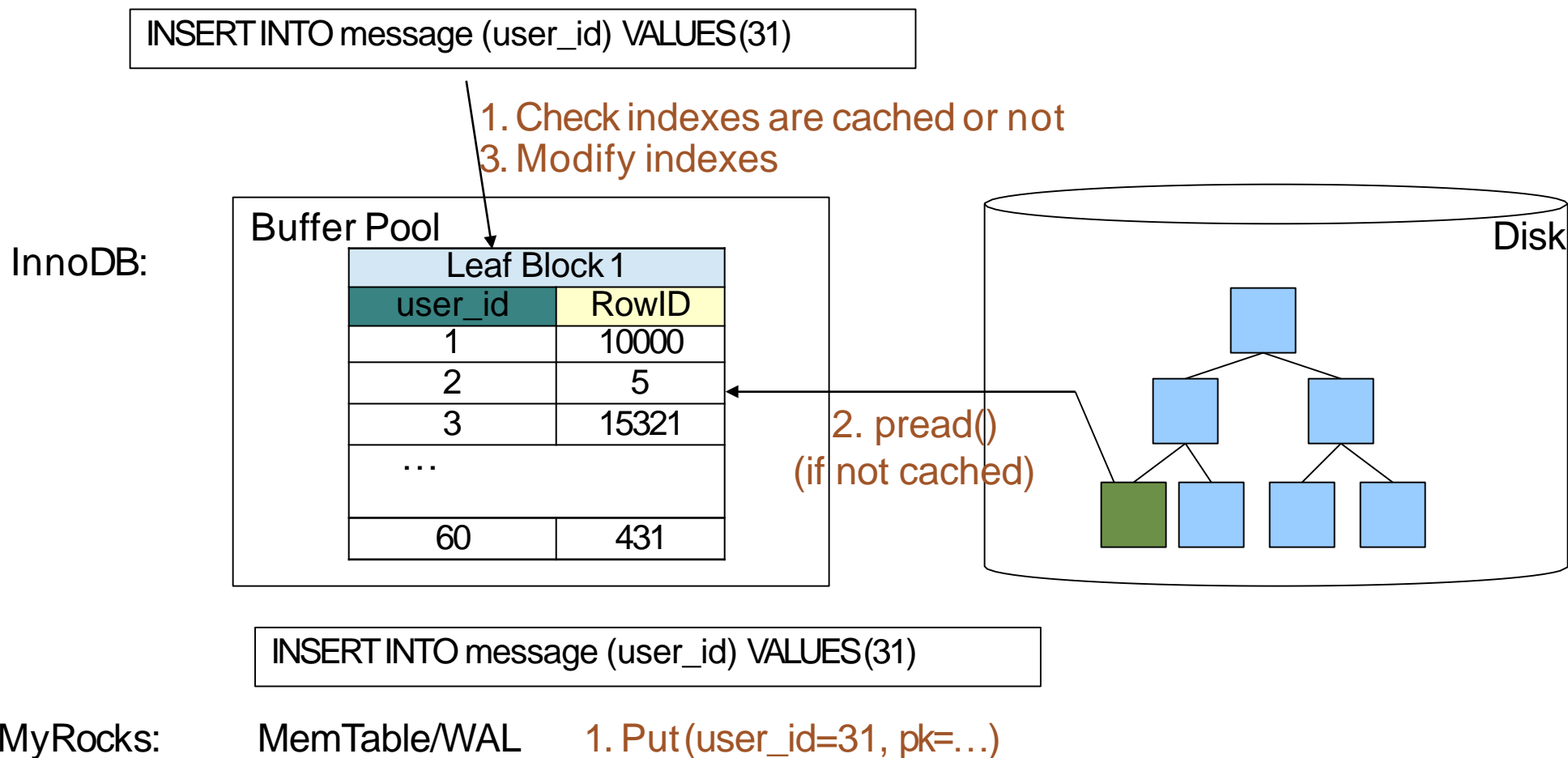
索引与列族

- 单独的MemTables与SST文件
- 共享的WAL



索引维护的效率

- 在 MyRocks 中，非唯一索引不需要在索引写入时读取
- 写QPS更好，尤其是索引不在内存中时



数据字典

- MyRocks 将一些元数据信息存储到 RocksDB 的专用列族__system__中
- Dictionary Examples
 - Table name => internal index id 的映射
 - Internal index id => index metadata, Column Family id
 - Column Family id => Column Family flags (i.e. reverse order or not)
 - Binlog信息
 - Binlog name, position, and GTID
 - Written at transaction commit
 - Index statistics
- 可以通过information_schema读取

THANKS!