存储数据结构之 B-tree

极客时间

扫码试看/订阅

《ZooKeeper实战与源码剖析》视频课程

B-tree

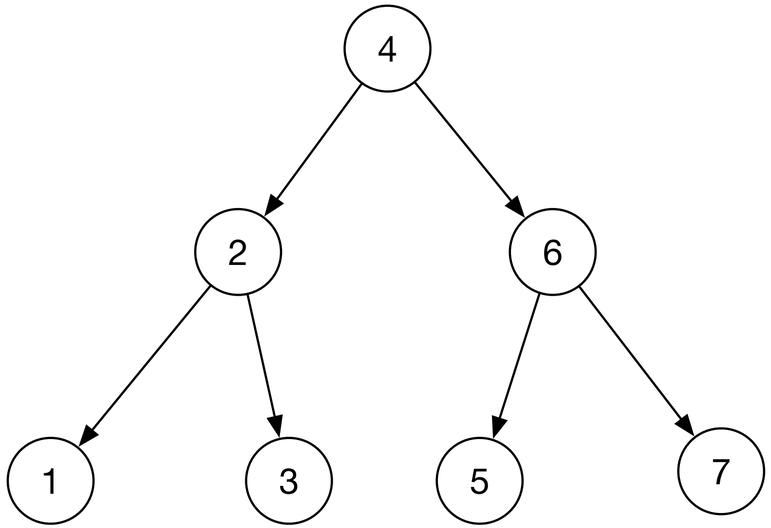
B-tree 的应用十分广泛，尤其是在关系型数据库领域，下面列出了一些知名的 B-tree 存储引擎：

- 关系型数据库系统 Oracle、SQL Server、MySQL 和 PostgreSQL 都支持 B-tree。
- WiredTiger 是 MongoDB 的默认存储引擎，开发语言是 C，支持 B-tree。
- BoltDB：Go 语言开发的 B-tree 存储引擎，etcd 使用 BoltDB 的 fork bbolt。

存储引擎一般用的都是 B+tree，但是存储引擎界不太区分 B-tree 和 B+tree，说 B-tree 的时候其实一般指的是 B+tree。

平衡二叉搜索树

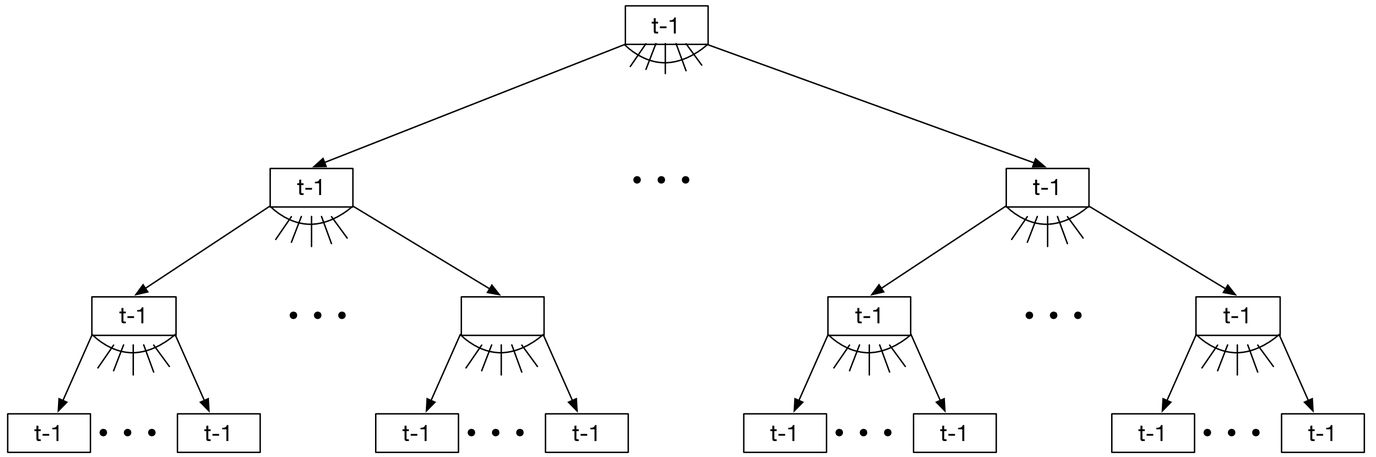
平衡二叉搜索树是用来快速查找 key-value 的有序数据结构。平衡二叉搜索树适用于内存场景，但是不试用于于外部存储。原因在于没访问一个节点都要访问一次外部存储，而访问外部存储是非常耗时的。要减少访问外部存储的次数，就要减少树的高度，要减少树的高度就要增加一个节点保存 key 的个数。B-tree 就是用增加节点中 key 个数的方案来减少对外部存储的访问。



lookup, insertion, and removal:$O(\log n)$

B-tree

B-tree 是一种平衡搜索树。每一个 B-tree 有一个参数 t，叫做 minimum degree。每一个节点的 degree 在 t 和 2t 之间。下图是一个每个节点的 degree 都为 t 的 B-tree。如果 t 为 1024 的话，下面的 B-tree 可以保存 1G 多的 key 值。因为 B-tree 的内部节点通常可以缓存在内存中，访问一个 key 只需要访问一次外部存储。



depth	#node
0	1
1	t
2	t^2
3	t^3

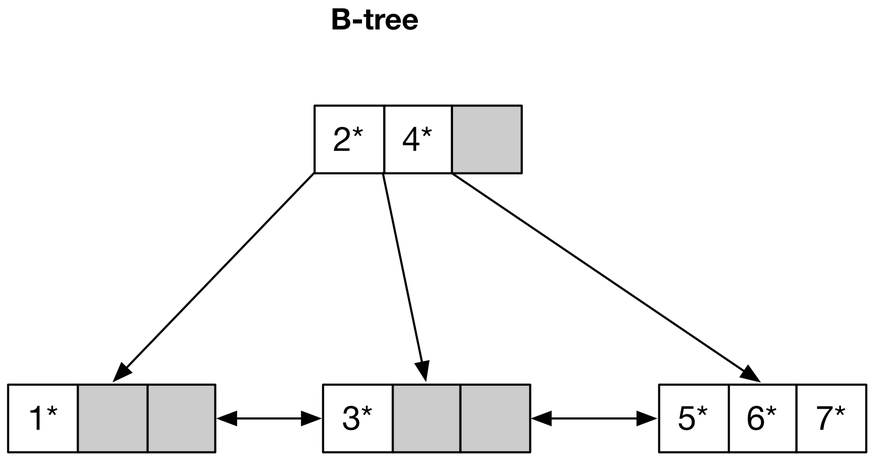
B-tree 和平衡二叉搜索树的算法复杂度一样的，但是减少了对外部存储的访问次数。

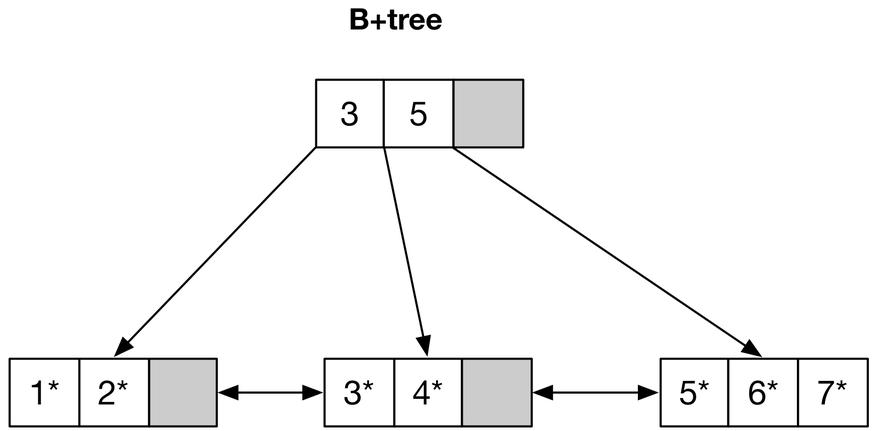
B-tree 特点

- 所有的节点添加都是通过节点分裂完成的。
- 所有的节点删除都是通过节点合并完成。
- 所有的插入都发生在叶子节点。
- B-tree 的节点的大小通常是文件系统 block 大小的倍数，例如 4k，8k 和 16k。

B+tree

为了让 B-tree 的内部节点可以具有更大的 degree，可以规定内部节点只保存 key，不保存 value。这样的 B-tree 叫作 B+tree。另外通常会把叶子节点连成一个双向链表，方便 key-value 升序和降序扫描。

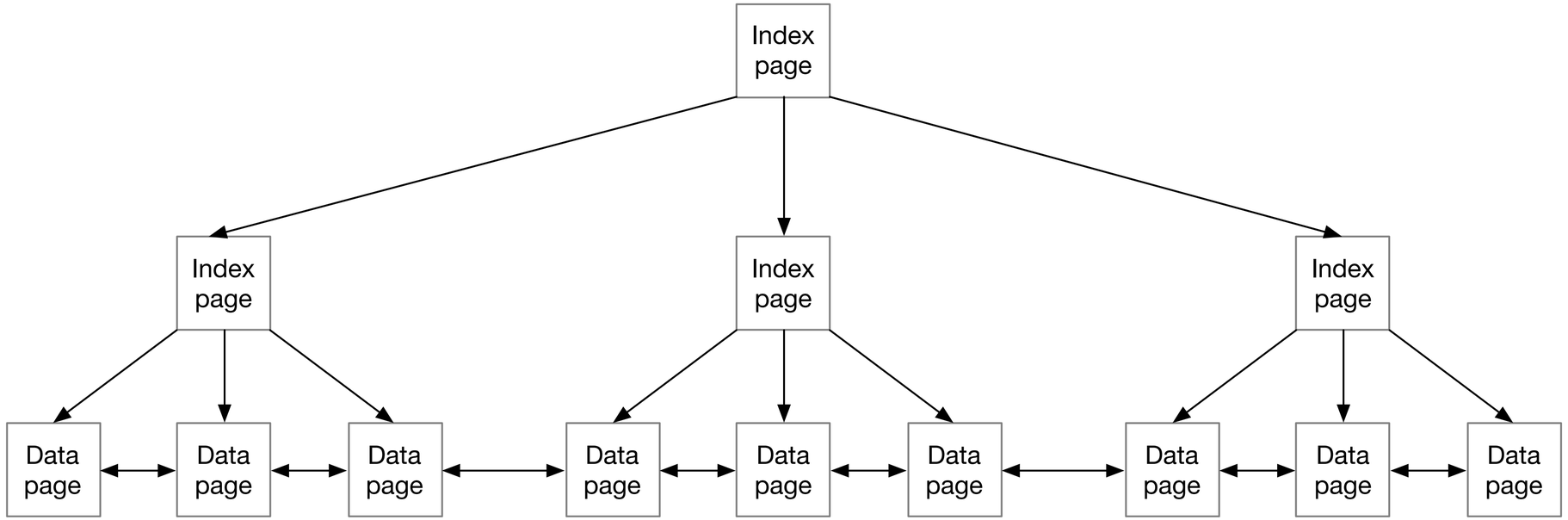




1*代表key为1的key-value

B+tree 索引

大部分关系型数据库表的主索引都是用的 B+tree。B+tree 的叶子节点叫作 data page，内部节点叫作 index page。



存储数据结构之 LSM

Log Structured Merge-tree（LSM）

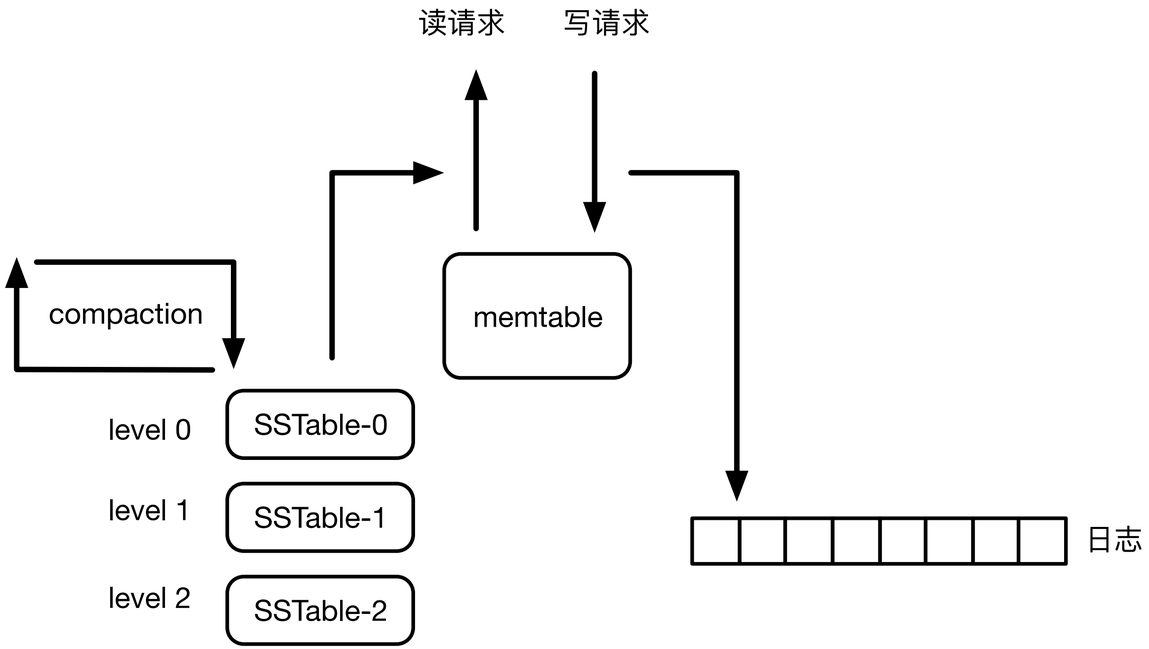
LSM 是另外一种广泛使用的存储引擎数据结构。LSM 是在 1996 发明的，但是到了 2006 年从 Bigtable 开始才受到关注。

LSM 架构

一个基于 LSM 的存储引擎有以下 3 部分组成：

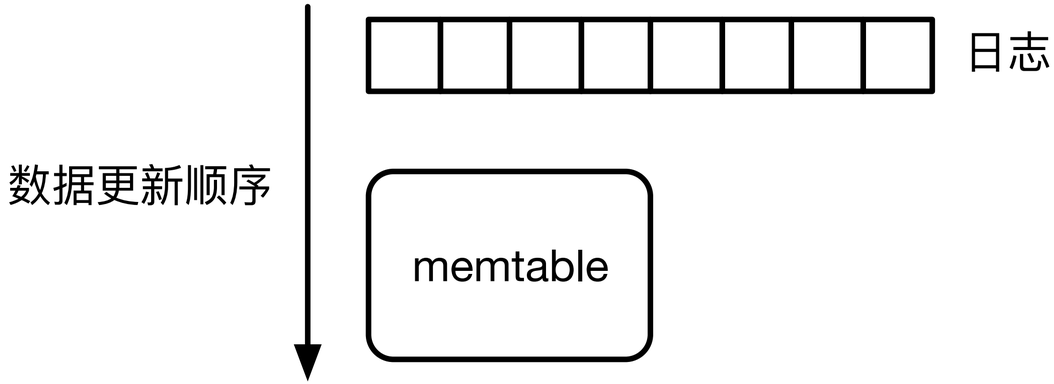
- Memtable：保存有序 KV 对的内存缓冲区。
- 多个 SSTable：保存有序 KV 对的只读文件。
- 日志：事务日志。

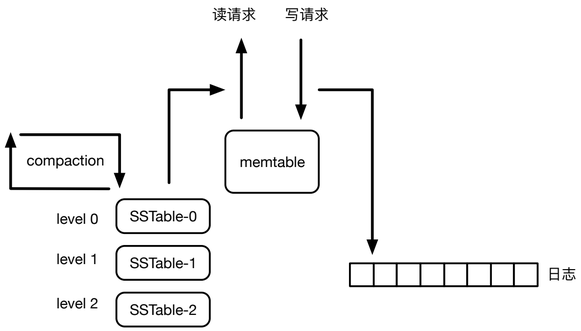
LSM 存储 MVCC 的 key-value。每次更新一个 key-value 都会生成一个新版本，删除一个 key-value 会生成一个 tombstone 的新版本。



LSM 写操作

一个写操作首先在日志中追加事务日志，然后把新的 key-value 更新到 Memtable。LSM 的事务是 WAL 日志。

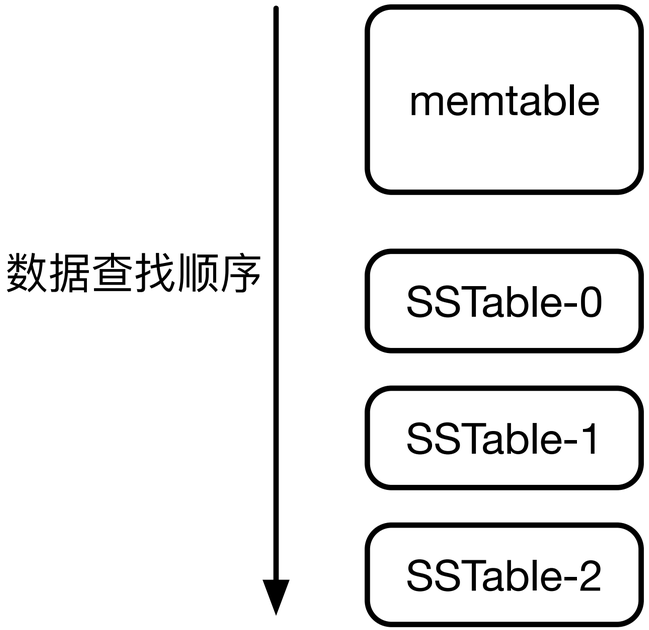


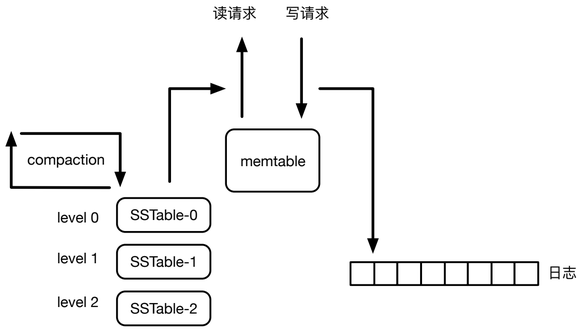


LSM 读操作

在由 Memtable 和 SSTable 合并成的一个有序 KV 视图上进行 Key 值的查找。例如在右图所示的 LSM 中，要查找一个 key a，

1. 在 memtable 中查找，如果查找到，返回。否则继续。
2. 在 SSTable-0 中查找，如果查找到，返回。否则继续。
3. 在 SSTable-1 中查找，如果查找到，返回。否则继续。
4. 在 SSTable-2 中查找，如果查找到，返回。否则返回空值。



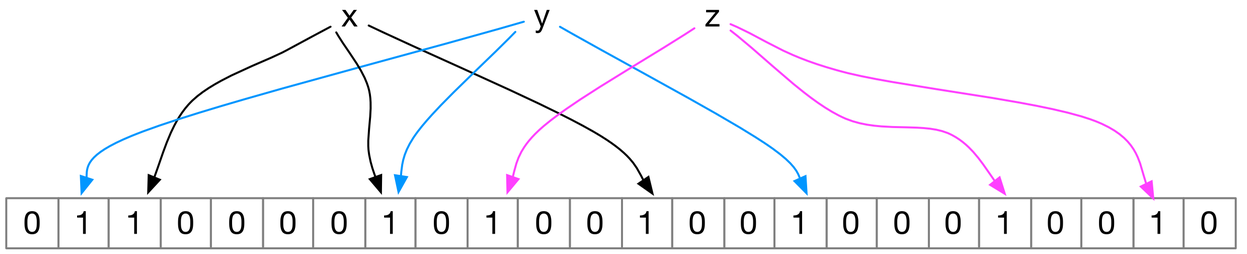


Bloom Filter

使用 Bloom filter 来提升 LSM 数据读取的性能。Bloom filter 是一种随机数据结构，可以在 O(1) 时间内判断一个给定的元素是否在集合中。False positive 是可能的，既 Bloom filter 判断在集合中的元素有可能实际不在集合中，但是 false negative 是不可能的。Bloom filter 由一个 m 位的位向量和k个相互独立的哈希函数 h_1，h_2，...，h_k 构成。这些hash函数的值范围是 {1，...，m}。初始化 Bloom filter 的时候把位向量的所有的位都置为 0。 添加元素 a 到集合的时候，把维向量 $h_1(a)$， $h_2(a)$， $h_k(a)$ 位置上的位置为 1。判断一个元素 b 是否在集合中的时候，检查把维向量 $h_1(b)$， $h_2(b)$， ... ，$h_k(a)$ 位置上的位是否都为 1。如果这些位都为 1，那么认为 b 在集合中；否则认为 b 不在集合之中。下图所示的是一个 m 为 14，k 为 3 的 Bloom filter。下面是计算 False positive 概率的公式（n 是添加过的元素数量）：

$$\text{False positive probability} \approx (1 - (1 - \frac{1}{m})^{kn})^k$$

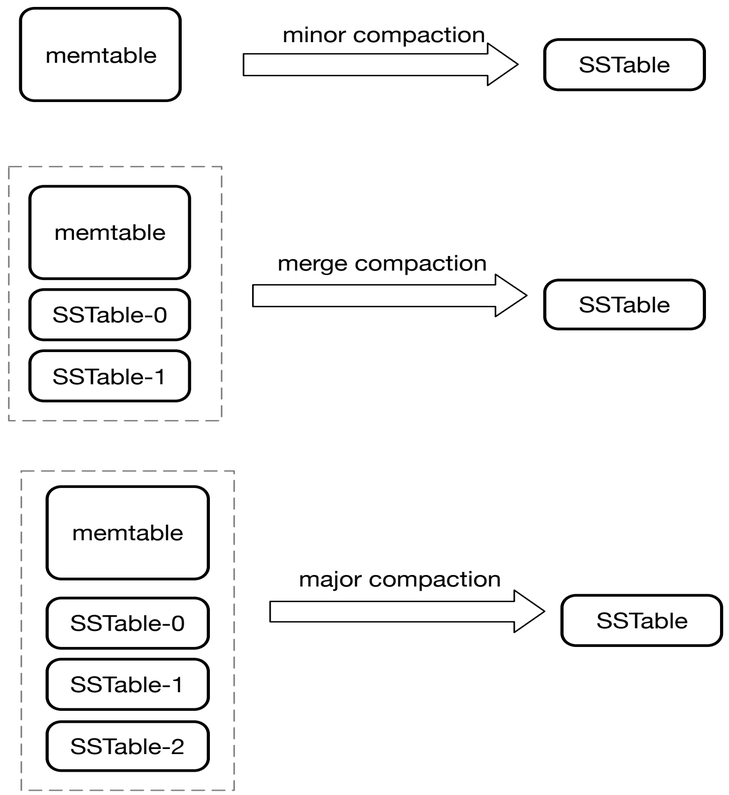
下图所示的是一个 m 为 14，k 为 3 的 Bloom filter。



Compaction

如果我们一直对 memtable 进行写入，memtable 就会一直增大直到超出服务器的内部限制。所以我们需要把 memtable 的内存数据放到 durable storage 上去，生成 SSTable 文件，这叫做 minor compaction。

- Minor compaction：把 memtable 的内容写到一个 SSTable。目的是减少内存消耗，另外减少数据恢复时需要从日志读取的数据量。
- Merge compaction：把几个连续 level 的 SSTable 和 memtable 合并成一个 SSTable。目的是减少读操作要读取的 SSTable 数量。
- Major compaction：合并所有 level 上的 SSTable 的 merge compaction。目的在于彻底删除 tombstone 数据，并释放存储空间。



基于 LSM 的存储引擎

下面列出了几个知名的基于 LSM 的存储引擎：

- LevelDB：开发语言是 C++，Chrome 的 IndexedDB 使用的是 LevelDB。
- RocksDB：开发语言是 C++，RocksDB 功能丰富，应用十分广泛，例如 CockroachDB、TiKV 和 Kafka Streams 都使用了它。
- Pebble：开发语言是 Go，应用于 CockroachDB。
- BadgerDB：一种分离存储 key 和 value 的 LSM 存储引擎。
- WiredTiger：WiredTiger 除了支持 B-tree 以外，还支持 LSM。

存储引擎的放大指标（Amplification Factors）

- 读放大（read amplification）：一个查询涉及的外部存储读操作次数。如果我们查询一个数据需要做 3 次外部存储读取，那么读放大就是 3。
- 写放大（write amplification）：写入外部存储设备的数据量和写入数据库的数据量的比率。如果我们对数据库写入了 10MB 数据，但是对外部存储设备写入了 20BM 数据，写放大就是 2。
- 空间放大（space amplification）：数据库占用的外部存储量和数据库本身的数据量的比率。如果一个 10MB 的数据库占用了 100MB，那么空间放大就是 10。

比较 B-tree 和 LSM

LSM 和 B-tree 在 Read amplification（读放大），Write amplification（写放大）和 Space amplification（空间放大）这个三个指标上的区别：

	LSM	B+-Tree
读放大	一个读操作要对多个 level 上的 SSTable 进行读操作。	一个 key-value 的写操作涉及一个数据页的读操作，若干个索引页的读操作。
写放大	一个 key-value 值的写操作要在多级的 SSTable 上进行。	一个 key-value 的写操作涉及数据页的写操作，若干个索引页的写操作。
空间放大	在 SSTable 中存储一个 key-value 的多个版本。	索引页和页 fragmentation。

LSM 和 B+-Tree 在性能上的比较：

- 写操作：LSM 上的一个写操作涉及对日志的追加操作和对 memtable 的更新。但是在 B+-Tree 上面，一个写操作对若干个索引页和一个数据页进行读写操作，可能导致多次的随机 IO。所以 LSM 的写操作性能一般要比 B+-Tree 的写操作性能好。

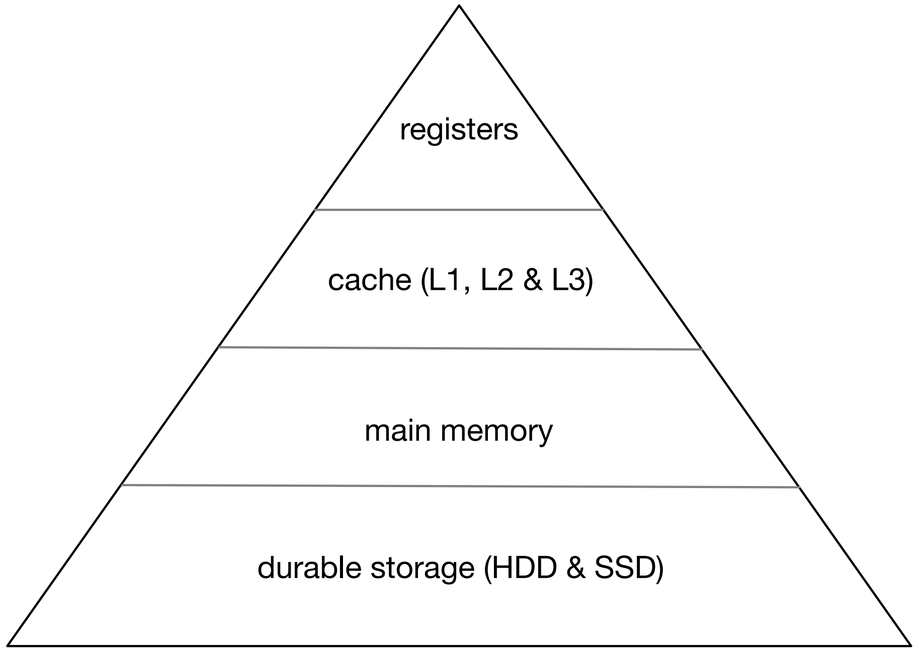
- 读操作：LSM 上的一个读操作需要对所有 SSTable 的内容和 memtable 的内容进行合并 。但是在 B+-Tree 上面，一个读操作对若干个索引页和一个数据页进行读操作 。所以 B+-Tree 的读操作性能一般要比 LSM 的读操作性能好。

本地存储技术总结

数据的随机读写 vs 顺序读写

在右图的 memory hierarchy，越往下的存储方式容量越大延迟越大，越往上容量越小延迟越小。对 main memory 和 durable storage 的数据访问，顺序读写的效率都要比随机读写高。例如 HDD 的 seek time 通常在 3 到 9ms 之前，所以一个 HDD 一秒最多支持 300 多次随机读写。虽然 SSD 和 main memory 的随机读写效率要比 HDD 好的多，顺序读写的效率仍然要比随机读写高。

所以我们设计存储系统的时候，要尽量避免随机读写多使用顺序读写。

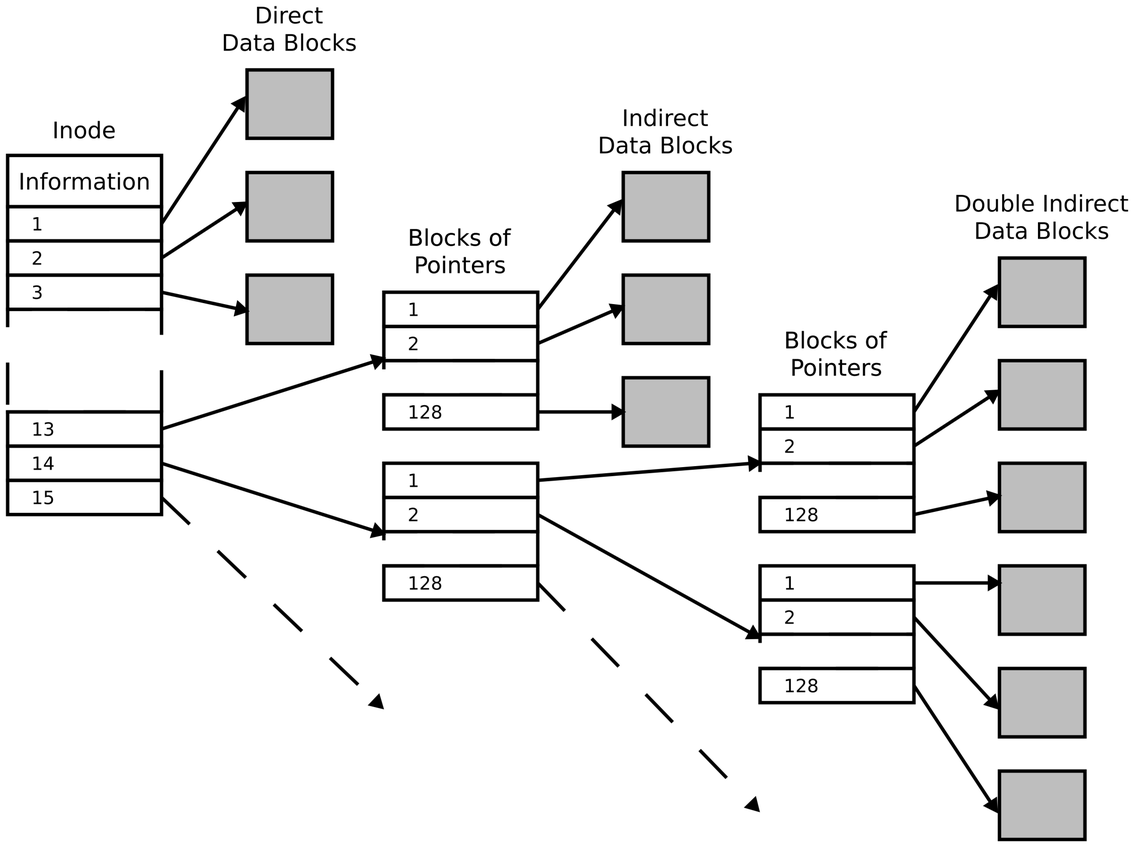


The memory hierarchy

文件系统基础知识

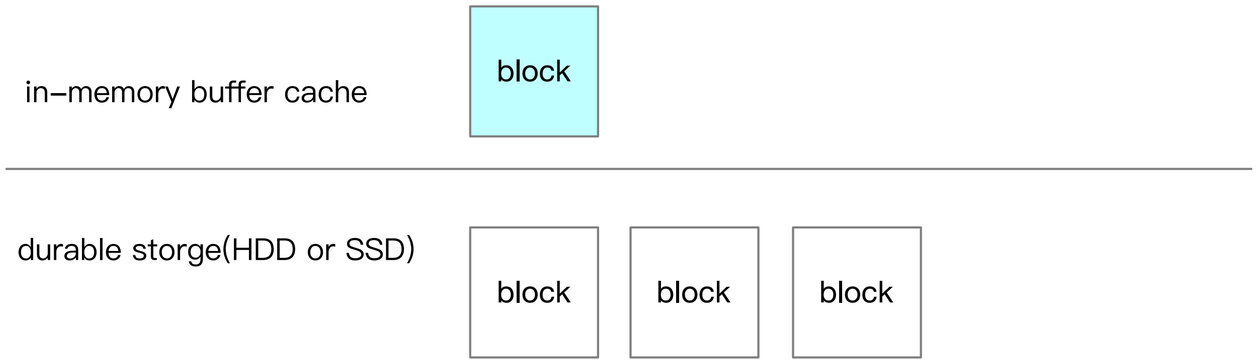
ext4 文件系统

ext4 是 Linux 系统上广泛使用的文件系统。下图列的是 ext4 文件系统 inode 的结构。其中 information 包括文件的 size，last access time 和 last modification time 等。文件的 inode 和 data block 存储在存储设备的不同位置。



文件系统 API

访问文件内容是 read 和 write 两个系统调用。除非使用了 O_DIRECT 选项，read 和 write 操作的都是 block 的 buffer cache，Linux OS 会定期把 dirty block 刷新到 durable storage 上去。



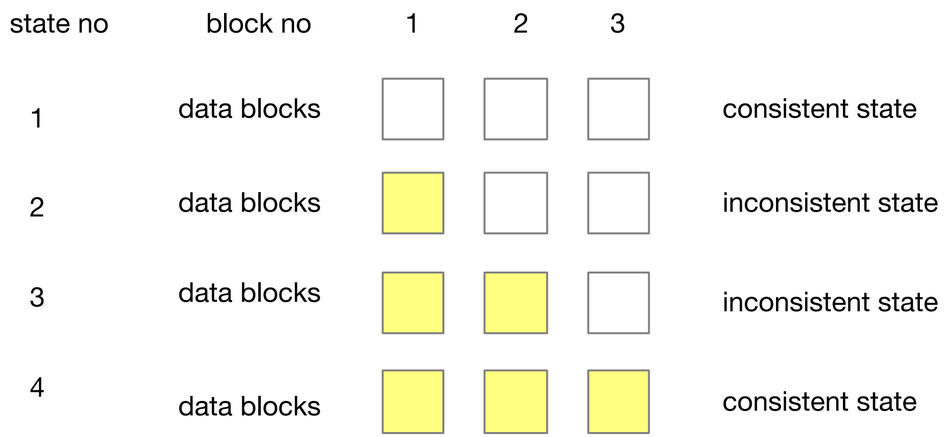
设计可靠的存储系统要求把内容实际写到 durable storage 上去。下面的这两个系统调用提供了把 buffer cache 的内容手动刷新到 durable storage 的机制：

- fsync：把文件的数据 block 和 inode 的 metadata 刷新到 durable storage。
- fdatasync：把把文件的数据 block 刷新到 durable storage。只有修改过的 metadata 影响后面的操作才把 metadata 也刷新到 durable storage。

Write Ahead Logging

如何保证 durable storage 写入的原子性

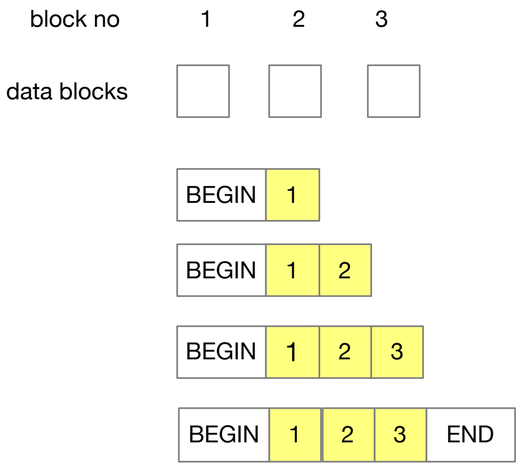
我们在 write 调用之后调用 fsync/fdatasync，文件系统通常可以保证对一个 block 写入的原子性。如果我们的一个数据写入包含对多个 block 的写入。要保证这样整个写入的原子性，就需要另外的机制。



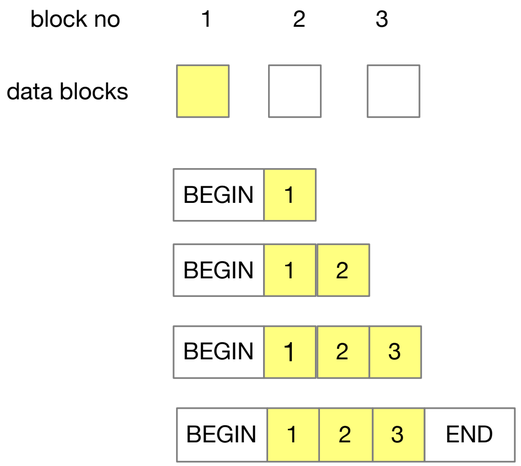
例如在上图中，我们要对 3 个 data block 进行写入。如果我们依次对这些 block 写入，如果在写入 block1 之后发生 crash，数据就会处于状态 2。在状态 2中，block1 保存旧数据、block2 和 block3 保存旧数据，数据是不一致的。

Write Ahead Logging（WAL）

WAL 是广泛使用的保证多 block 数据写入原子性的技术。WAL 就是在对 block 进行写入之前，先把新的数据写到一个日志。只有在写入 END 日志并调用 sync API，才开始对 block 进行写入。如果在对 block 进行写入的任何时候发生 crash，都可以在重启的使用 WAL 里面的数据完成 block 的写入。



完成END的日志写入是整个数据写入的commit point

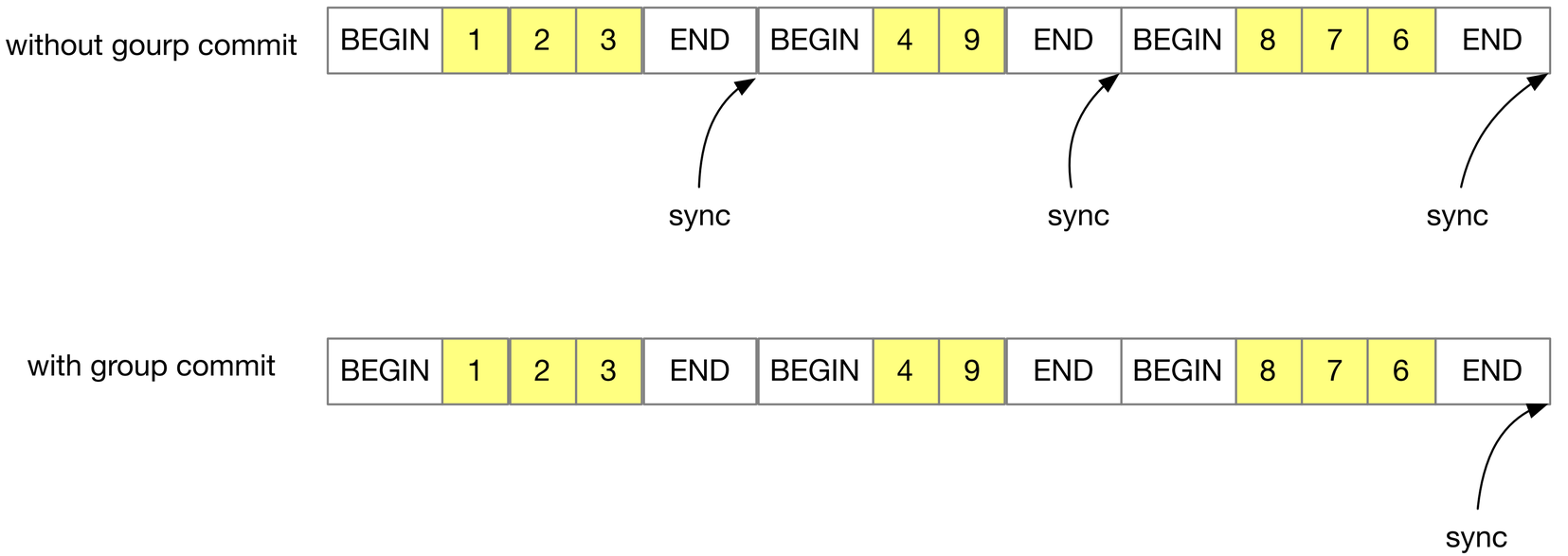


完成block 1写入之后，发生crash

另外通过使用 WAL，我们在提交一个操作之前只需要进行文件的顺序写入，从而减少了包含多 block 文件操作的数据写入时延。

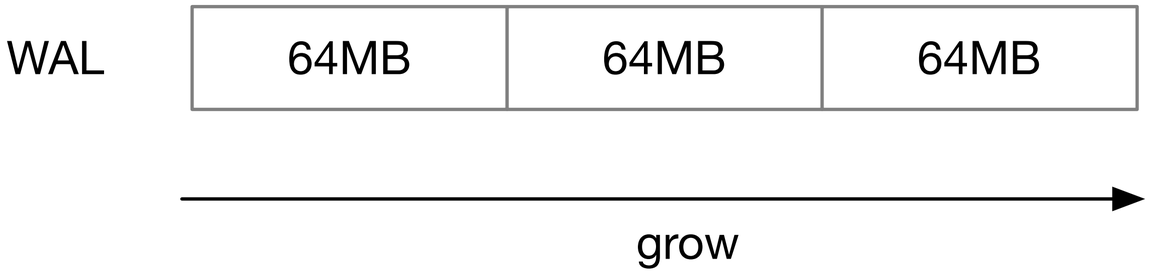
WAL 优化1：Group Commit

上面的 WAL 方案中每次写入完 END 日志都要调用一次耗时的 sync API，会影响系统的性能。为了解决这个问题，我们可以使用 group commit。group commit 就是一次提交多个数据写入，只有在写入最后一个数据写入的 END 日志之后，才调用一次 sync API。



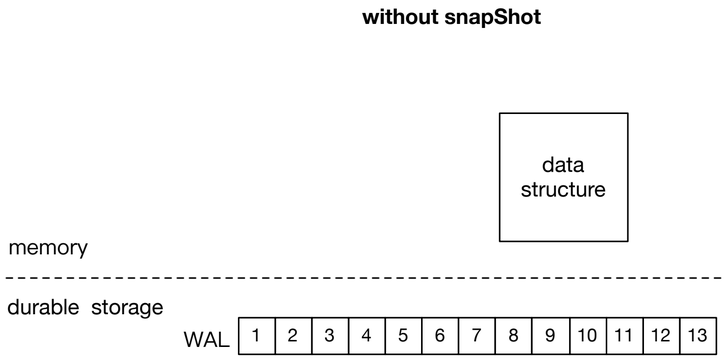
WAL 优化2：File Padding

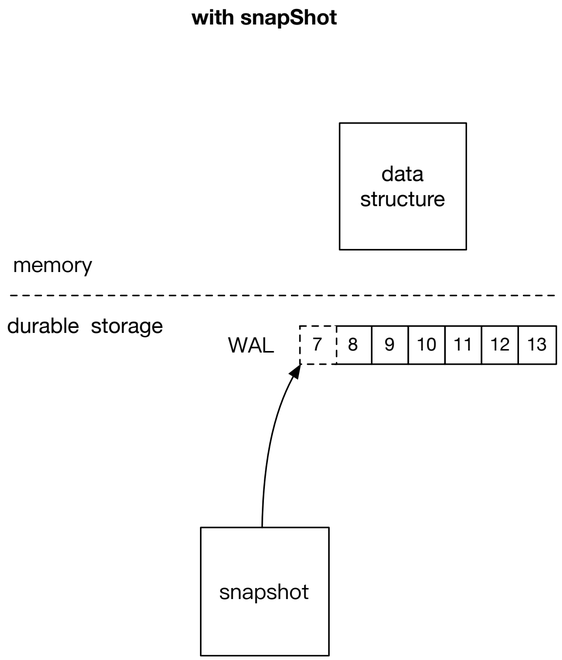
在往 WAL 里面追加日志的时候，如果当前的文件 block 不能保存新添加的日志，就要为文件分配新的 block，这要更新文件 inode 里面的信息（例如 size）。如果我们使用的是 HHD 的话，就要先 seek 到 inode 所在的位置，然后回到新添加 block 的位置进行日志追加。为了减少这些 seek，我们可以预先为 WAL 分配 block。例如 ZooKeeper 就是每次为 WAL 分配 64MB 的 block。



WAL 优化3：快照

如果我们使用一个内存数据结构加 WAL 的存储方案，WAL 就会一直增长。这样在存储系统启动的时候，就要读取大量的 WAL 日志数据来重建内存数据。快照可以解决这个问题。快照是应用 WAL 中从头到某一个日志条目产生的内存数据结构的序列化，例如下图中的快照就是应用从 1 到 7 日志条目产生的。





除了解决启动时间过长的问题之外，快照还可以减少存储空间的使用。WAL 的多个日志条目有可能是对同一个数据的改动，通过快照，就可以只保留最新的数据改动。

数据序列化

现在有众多的数据序列化方案，下面列出一些比较有影响力的序列化方案：

- JSON：基于文本的序列化方案，方便易用，没有 schema，但是序列化的效率低，广泛应用于 HTTP API 中。
- BSON：二进制的 JSON 序列化方案，应用于 MongoDB。
- Protobuf：Google 研发的二进制的序列化方案，有 schema，广泛应用于 Google 内部，在开源界也有广泛的应用（例如 gRPC）。
- Thrift：Facebook 研发的和 Protobuf 类似的一种二进制序列化方案，是 Apache 的项目。
- Avro：二进制的序列化方案，Apache 项目，在大数据领域用的比较多。

如何研发本地存储

研发一个高效的本地存储引擎需要该领域的专家级技术，所以不建议自己从 0 开始研发。目前开源界有众多的本地存储引擎，建议直接使用现有的方案。如果现有的开源方案不能满足要求，可以在这些方案的基础之上进行二次开发。

极客时间

扫码试看/订阅

《ZooKeeper实战与源码剖析》视频课程