# 背景

Log Structured Merge Tree，简称LSM-Tree。2006年，Google发表了 BigTable的论文。这篇论文提到BigTable单机上所使用的数据结构就是LSM-Tree。  
 很多存储产品使用LSM-Tree作为数据结构，比如 Apache HBase，Apache Cassandra，MongoDB的Wired Tiger存储引擎，LevelDB存储引擎，RocksDB存储引擎等。  
 简单地说，**LSM-Tree的设计目标是提供比传统的B-Tree/B+Tree更好的写性能。LSM-Tree通过将磁盘的随机写转化为顺序写来提高写性能 ，而付出的代价就是牺牲部分读性能、写放大（B-Tree/B+Tree 同样有写放大的问题）**。

## 如何优化写性能

如果我们对写性能特别敏感，我们最好怎么做？

Append only：所有写操作都是将数据添加到文件末尾。这样顺序写的性能是最好的，大约等于磁盘的理论速度（无论是SSD还是HDD，顺序写性能都要明显由于随机写性能）。

注：GFS就是采用append only。

但是append only的方式会带来一些问题：

**1、不支持有序遍历。**

**2、需要垃圾回收（清理过期数据）。**

所以，纯粹的append only方式只能适用于一些简单的场景：

1、存储系统的 WAL。

2、能知道明确的offset的查询，比如 Bitcask。

注：HDFS，Kafka都是利用append only提高写效率的。

## 如何优化读性能

如果我们对读性能特别敏感，一般我们有两种方式：

有序存储，比如B-Tree/B+Tree。但是B-Tree/B+Tree会导致随机写。

哈希存储：不支持有序遍历，适用范围有限。

## 读写性能的权衡

如何获得接近append only的写性能，而又能拥有不错的读性能呢？以LevelDB/RocksDB为代表的LSM-Tree存储引擎给出了一个参考答案。  
 LevelDB 的写操作（Put/Delete/Write）主要由两步组成：

1、写日志（WAL，顺序写）。

2、**写MemTable（内存中的SkipList）**。

所以，正常情况下，LevelDB的写速度非常快。  
 内存中的 MemTable写满后，会转换为Immutable MemTable，然后**被后台线程compact成按key有序存储的SSTable（顺序写）**。  
 SSTable按照数据从新到旧被组织成多个层次（上层新下层旧），点查询（Get）的时候从上往下一层层查找，所以LevelDB的读操作可能会有多次磁盘IO（**LevelDB通过table cache、block cache和bloom filter等优化措施来减少读操作的I/O次数**）。  
 后台线程的定期compaction负责回收过期数据和维护每一层数据的有序性。在数据局部有序的基础上，LevelDB实现了数据的（全局）有序遍历。

# 概述

LevelDB是Google的Jeff Dean和Sanjay Ghemawat设计开发的key-value存储引擎。LevelDB底层存储利用了LSM tree的思想，RocksDB是Facebook基于LevelDB开发的存储引擎，针对LevelDB做了很多优化，但是大部分模块的实现机制是一样的。

**LevelDB是一个持久化存储的KV系统，和Redis这种内存型的KV系统不同，LevelDB不会像Redis一样狂吃内存，而是将大部分数据存储到磁盘上。**LevleDB在存储数据时，是根据记录的key值有序存储的，就是说**相邻的key值在存储文件中是依次顺序存储的**，而应用可以自定义key大小比较函数，LevleDB会按照用户定义的比较函数依序存储这些记录。

像大多数KV系统一样，LevelDB的操作接口简单，基本操作包括写记录、读记录以及删除记录。另外，LevelDB支持数据快照（snapshot）功能，使得读取操作不受写操作影响，可以在读操作过程中始终看到一致的数据。除此之外，LevelDB还支持数据压缩等操作，这对于减小存储空间以及增快IO效率都有直接的帮助。

参考：<https://zhuanlan.zhihu.com/p/203578801>

## 特点

作为一个函数库，LevelDB有以下特点：

键和值都是任意的字节数组；

数据以键的顺序存储；

可以使用定制的比较器定义排序的方式；

基本操作是Put(key,value)、 Get(key)、Delete(key)；

一个原子的Batch中可以做多个改变；

支持Snapshot，用户可以建立一个一致性的视图；

可以正向和反向迭代；

支持snappy压缩数据；

操作系统相关操作被抽象，跨平台时可以定制这些接口；

同一时间只能由一个进程访问数据库，但是支持多线程访问。

## 功能特性

## 对比

RocksDB与LevelDB对比：

1. 增加了column family，这样有利于多个不相关的数据集存储在同一个db中，因为不同column family的数据是存储在不同的sst和memtable中，所以一定程度上起到了隔离的作用。
2. 采用了多线程同时进行compaction的方法，优化了compact的速度。
3. 增加了merge operator，优化了modify的效率。
4. 将flush和compaction分开不同的线程池，能有效的加快flush，防止stall。
5. 增加了对write ahead log(WAL)的特殊管理机制，这样就能方便管理WAL文件，因为WAL是binlog文件。

# 编译安装

要使用LevelDB，必须先要编译安装函数库。LevelDB使用了cmake，需要安装cmake，可以根据需要，安装snappy、crc32c和tcmalloc。通过执行以下几个步骤，就可以安装函数库了：

mkdir -p build && cd build

cmake -DCMAKE\_BUILD\_TYPE=Release .. && cmake --build .

make install

# 源码

根目录下面有几个主要的目录：

include: 函数库的头文件

port: 可移植性相关的功能

util: 项目用到的一些功能函数

table: SSTable的实现

db: 数据库实现，版本管理，Compaction，WAL和MemTable实现

LevelDB作为函数库，对外提供的接口文件及功能如下：

cache.h: 缓存接口，提供了默认的LRU缓存，也可以自己实现缓存

comparator.h: 定以数据库比较器的接口，用来比较键，可以使用默认的基于字节的比较，可以定义自己的比较器

dumpfile.h: 以可读文本形式导出一个文件，调试使用

export.h: 可移植性相关

iterator.h: 迭代器接口

slice.h: 实现一个字符串，存储指针和长度，指向字符串

table\_builder.h: 构造一个SSTable

write\_batch.h: 实现批量写入的接口

c.h: 实现C语言相关的接口

db.h: 操作数据库的主要接口

env.h: 定义操作系统相关的功能，如读写文件之类的

filter\_policy.h: 定义布隆过滤器接口

options.h: 配置选项

status.h: 定义数据库操作的返回状态

table.h: SSTable相关的接口

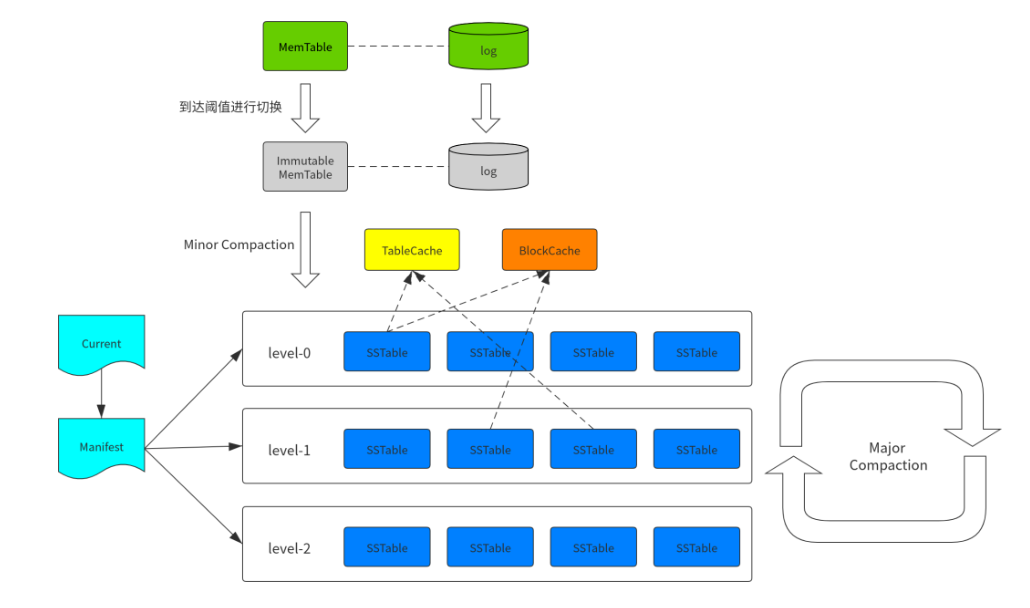
Comparator 定义排序的规则；

Status定义函数执行的结果信息；

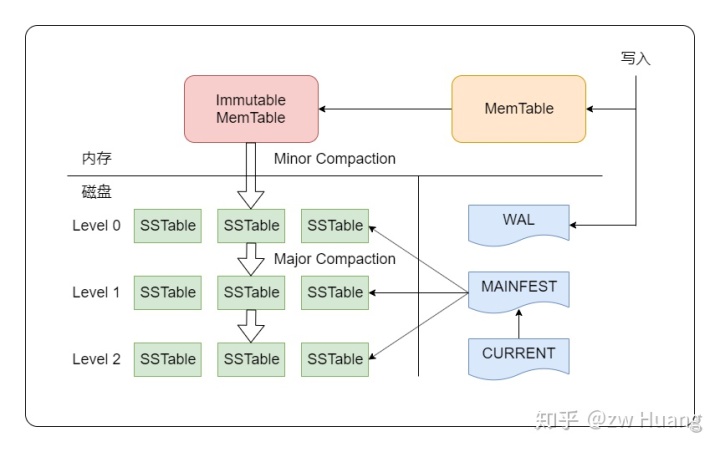
Env封装系统相关的调用，比如文件操作，线程操作；

Options指定数据库选项。

# 架构



LevelDB按照存储来分，可以分为三个组件：MemTable，SSTable，WAL。



MANIFEST是用来管理SSTable的，保存了SSTable的元信息。

# 原理

## arena内存管理

数据库内存分配非常重要，尤其是插入一个键值对时，需要分配内存给这个键值对。如果直接使用malloc/free或者new/delete碰到很小的键值对时，每个调用平均的开销比较大，而且会产生很多内存碎片。

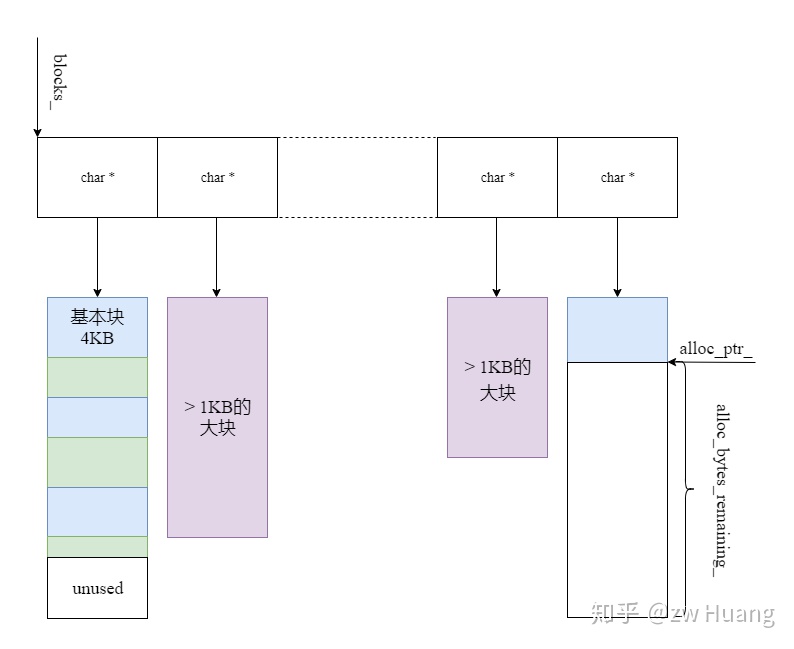
LevelDB只有一处使用了自己的内存管理，就是MemTable，MemTable使用一个Skiplist存储最新插入的键值对。LevelDB为每个MemTable都绑定了一个Arena来管理内存，其它地方则直接使用malloc/free，因为这些地方都使用了比较大块的内存或者新建销毁不频繁。

### 设计思想

内存分配经常采用的一种方式，就是首先使用new预分配一块比较大的内存，需要使用小块内存时，从这块大内存里面继续分配，这时候分配可能只是移动指针或者更新变量的事情了，非常高效。

arena内存管理就使用了这种思想，解决了小块内存频繁调用new的开销和内存碎片的问题，但是却可能浪费一些内存。

arena的内存分配如下图所示：



使用一个char \*的vector保存每个块；

当需要分配一块内存时，查看alloc\_bytes\_remaining\_(就是当前块还有多少内存未分配)是否大于等于所需内存；

如果大于等于，直接分配，这时候只需要移动指针即可；

如果小于，要分两种情况，看所需要分配的内存是否大于1KB；

如果大于1KB，直接分配相应大小的块，并且插入到vector中;

如果小于等于1KB，则分配一个4KB的块，插入到vector中，从4KB的块上分配相应的内存；

上一个块里没有分配的内存就浪费了。

arena使用一个Arena类来定义，将数据存储在std::vector<char\*> blocks\_变量里，依次存储，每个数组项存储一个内存块，使用alloc\_ptr\_和alloc\_bytes\_remaining\_来跟踪当前块分配的状态。

// util/arena.h util/arena.cc

static const int kBlockSize = 4096;

// 首先是Arena的定义

class Arena {

private:

...

char\* alloc\_ptr\_; // 指向当前块第一个free的字节

size\_t alloc\_bytes\_remaining\_; // 当前块还有多少字节free的内存

std::vector<char\*> blocks\_; // 用new分配的内存块的数组

...

};

// 分配内存的函数

inline char\* Arena::Allocate(size\_t bytes) {

// 如果当前块剩余的内存足够，更新free指针，返回内存指针

if (bytes <= alloc\_bytes\_remaining\_) {

char\* result = alloc\_ptr\_;

alloc\_ptr\_ += bytes;

alloc\_bytes\_remaining\_ -= bytes;

return result;

}

// 否则退化的方式分配

return AllocateFallback(bytes);

}

char\* Arena::AllocateFallback(size\_t bytes) {

// 如果分配的内存大于1KB，直接分配一个块

if (bytes > kBlockSize / 4) {

char\* result = AllocateNewBlock(bytes);

return result;

}

// 如果内存小于等于1KB，分配一个4KB的块，更新指针

alloc\_ptr\_ = AllocateNewBlock(kBlockSize);

alloc\_bytes\_remaining\_ = kBlockSize;

char\* result = alloc\_ptr\_;

alloc\_ptr\_ += bytes;

alloc\_bytes\_remaining\_ -= bytes;

return result;

}

这样的思路对于大块的内存会直接调用new分配，对于小块的内存会在大块内存的基础上分配。如果需要分配的内存刚好大于当前块剩余的大小，那么当前块剩余的内存空间就浪费了。这里采用简单化的处理，牺牲了内存使用率。大块的内存直接分配一个块，而不是分配一个4KB的块，可以减少内存的浪费。

对于释放内存，arena不支持单独释放某个块，而是只能销毁整个arena。这是和arena的使用场景有关的，arena存储的是内存中的键值对，对于LevelDB来说，只有插入操作，没有实际的删除操作，所以不需要释放一块内存。而当一个arena里的数据dump到SSTable后，只需要释放arena里所有的内存。

### 参考源码

### 总结

LevelDB的内存分配策略非常简单，这和使用场景有关的。如果Kv 数量很多，而且比较小的情况下，采用这种分配方式会非常高效，内存的浪费也可以控制在相对理想的水平，这些浪费的内存在MemTable写满后，就会释放了。

## Cache缓存

## 编码

编码说的是内存里的整数和字符串是怎么存储到磁盘上的。对于整数，主要是有Big Endian 和Little Endian的区分，还有变长整数和定长整数的区别。

对于LevelDB，有以下几点：

整数分为32位整数和64位整数；

整数分为定长整数和变长整数；

整数采用Little Endian的方式存储；

字符串采用长度前缀编码的方式存储，所以字符串里面可以出现任何字符，包括\0。

### 存储方式

#### 定长整数

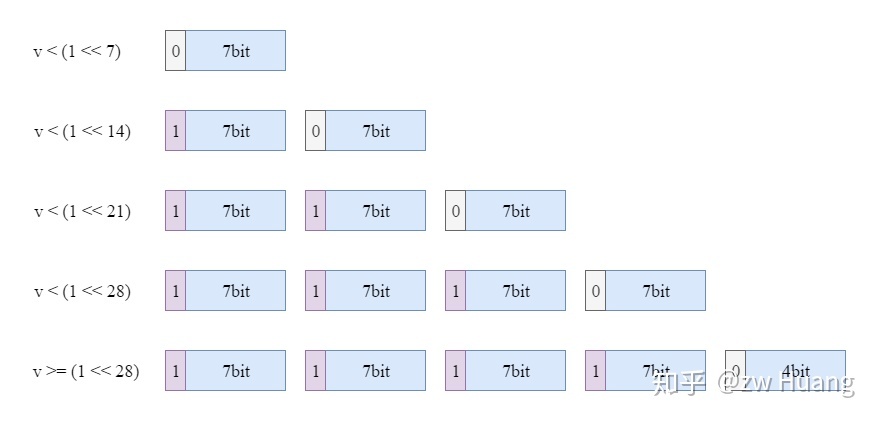
定长整数的存储非常简单，比如一个32位整数，把低8位的字节编码到第一个字节位置，低9-16位的字节放在第二个字节位置，以此类推。

#### 变长整数

一些小的整数，比如可能频繁使用100以内的整数（比如字符串的长度编码)，如果使用定长编码，至少需要4个字节，容易造成空间的浪费，所以LevelDB里面有一种变长整数的编码方式。对于32位整数使用1-5个字节进行编码，而对于64位整数使用1-10个字节进行编码。

这种编码方式的原理就是只使用一个字节的低7位存储数据，而高位用来做标识，高位为1的时候表示需要继续读取下一个字节，高位为0的时候表示当前字节已是最后一个字节。存储也是采用Little Endian的方式，拿出第一个字节的低7位作为数字的低7位，如果最高位为1，则拿出第二个字节的低7位作为数字的8-14位，以此类推。

以32位整数为例，如图:



如果编码值 v < 1 << 7，只需要7位即可编码，可使用0 + v的方式；

如果编码值 1 << 7 <= v < 1 << 14，需要两个字节编码，第一个字节使用1 + v的低7位，表示需要查看下一个字节，下一个字节使用0 + v的高7位，表示不需要查看下一个字节；

以此类推。

// util/coding.cc

char\* EncodeVarint32(char\* dst, uint32\_t v) {

uint8\_t\* ptr = reinterpret\_cast<uint8\_t\*>(dst);

// 设置字节最高位的掩码

static const int B = 128;

// 只需要1字节

if (v < (1 << 7)) {

\*(ptr++) = v;

} else if (v < (1 << 14)) {

// 低7位放在低字节，最高位置1

\*(ptr++) = v | B;

// 7-14位放在第二个字节，最高位置0

\*(ptr++) = v >> 7;

} else if (v < (1 << 21)) {

\*(ptr++) = v | B;

\*(ptr++) = (v >> 7) | B;

\*(ptr++) = v >> 14;

} else if (v < (1 << 28)) {

...

} else {

...

}

return reinterpret\_cast<char\*>(ptr);

}

#### 字符串

字符串的编码使用了前面32位变长整数来编码字符串长度，编码长度后跟字符串的实际值。因为采用长度前缀编码，所以不需要根据字符串里的\0来判断字符串的结束，字符串里面可以是任何值。

// util/coding.cc

void PutLengthPrefixedSlice(std::string\* dst, const Slice& value) {

PutVarint32(dst, value.size()); // 首先以变长的方式编码长度

dst->append(value.data(), value.size()); // 添加内容到长度后面

}

​

void PutVarint32(std::string\* dst, uint32\_t v) {

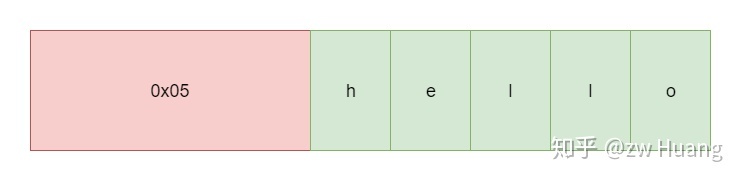
char buf[5];

char\* ptr = EncodeVarint32(buf, v);

dst->append(buf, ptr - buf);

}

假如需要编码字符串hello，因为长度是5个字节，所以长度的编码就是0x05，后面跟字符串实际值，所以最终的编码就是 05 48 45 4C 4C 4F，需要6个字节。



这种编码有以下好处：

字符串里面可以包含任何字符；

字符串的长度可以预先知道，读取文件的时候更加方便；

采用变长编码字符串长度，对于大多数的小字符串只需要1个字节。

### 参考源码

util/coding.h util/coding.cc

### 小结

LevelDB里面整数和字符串编码的方式很常见，我们经常可以在其它地方看到，这种方式简单高效紧凑。

## 字符串Slice

## Key

### User Key

### Internal Key

### Lookup Key

## Iterator迭代器

## Memtable

DB数据在内存中的存储方式，写操作会先写入memtable，memtable有最大限制(write\_buffer\_size)。**LevelDB/RocksDB的memtable的默认实现是skiplist**。**当memtable的size达到阈值，会变成只读的memtable(immutable memtable)。后台compaction线程负责把immutable memtable dump成sstable文件**。

RocksDB增加了column family的概念，不同的column family不共享memtable，其他memtable机制与LevelDB一样。

### MemTable

MemTable：内存数据结构，具体实现是SkipList。接受用户的读写请求，新的数据会先在这里写入。

MemTable是一个内存数据结构，简单说它就是一个SortedMap：

MemTable是一个Map，提供了Get接口，也就是可以快速地根据键查找到值，也可以使用Put接口插入KV；

MemTable是Sorted，也就是里面存储的键都是有序的，可以按照键的顺序迭代Map，或者做范围查询。

有了MemTable提供的Get和Put，就有了一个简单的内存KV数据库，可以实现KV的查询，插入和范围查询。

因为MemTable是内存数据结构，不需要磁盘IO，所以读写的速度非常快，所以就达到了场景需求：高效的写性能。然而这会有个问题，当数据库实例崩溃、宕机或者停机维护的时候，存储的数据就会丢失，这时候需要持久化数据。

### Immutable MemTable

Immutable MemTable：当MemTable的大小达到设定的阈值后，会被转换成 Immutable MemTable，只接受读操作，不再接受写操作，然后由后台线程flush到磁盘上—这个过程称为minor compaction。

## WAL

Log：数据写入MemTable之前会先写日志，用于防止宕机导致MemTable的数据丢失。一个日志文件对应到一个MemTable。

为了不丢失数据，需要持久化的功能。把数据持久化到磁盘上，最常用的技术就是日志技术，也就是当修改数据时，先把对数据的修改写到磁盘上，然后在MemTable里做修改。因为日志记录了每个操作，只要对日志进行重放便可以恢复MemTable，这样就做到了数据库实例崩溃、宕机或者停机维护的时候数据不丢失。

这种日志技术在数据库里面很常用（Redis里的AOF，Innodb里的Redo Log都是这样的技术），一般称为WAL （Write Ahead Log），正如名字一样，就是在写入前先写入日志的意思。

因为日志的写入都是Append的，也就是顺序写，所以写磁盘也是顺序写，虽然磁盘的随机写效率比较低，但是顺序写效率还是很高的，所以就算加入了日志，写效率还是很高，依然可以满足写多的场景。

另外可以通过控制日志写同步的策略在性能和可靠性之间做折中：

1、每次写入都做一次sync，可靠性最高，不会丢数据，但是性能最低；

2、每次写入，但是不sync，数据库崩溃不会丢数据，但是机器崩溃会丢数据，性能高；

3、每次写入，不sync，但是每1s做一次sync，数据库崩溃不会丢数据，但是机器崩溃丢最多1s的数据，性能较高。

有了MemTable和WAL，就有了一个功能比较完备的数据库了，类似于一个简化版的Redis。然而这又引入了新的问题：

1、如果日志量很大，每次重启数据库时，恢复的时间非常长；

2、内存的容量是有限的，所以当数据量太大，MemTable的大小超过内存容量时，就没法写入数据了。

这时候不仅仅需要将日志写入到磁盘，也需要定时将MemTable的镜像写入磁盘，并且清空MemTable。

说明：Redis是完全基于内存的KV数据库，LevelDB是基于内存和磁盘的KV数据库，比Redis多了一个SSTable，即从内存刷新到磁盘的动作。

## SSTable

SSTable：Sorted String Table。分为level-0到level-n多层，每一层包含多个SSTable，文件内数据有序。除了level-0之外，每一层内部的SSTable的key范围都不相交。

DB数据持久化文件，内部key是有序的，文件内部前面是数据，后面是索引元数据。

sstable文件之间逻辑上是分层的，**LevelDB最大支持7层**。

将MemTable的镜像写入到磁盘有一个要求：需要具有快速地在磁盘上查询一个键的功能。

最先想到的就是B+ Tree，这是一种古老的磁盘数据结构，现在很多数据库依然在采用，具有很好的读取性能。B+ Tree其实是一种多级索引，设计成这样是为了支持快速读取，同时也支持更新，但是B+ Tree更新的开销会比较大。如果将一个MemTable写入磁盘的方式是将数据原地更新磁盘上已存在的镜像，那么其实就是使用了类似于B+ Tree，这样会有很大的更新开销。所以LevelDB不更新磁盘镜像，每次都将MemTable写入一个新镜像。

对于不更新的磁盘数据结构，并不需要使用B+ Tree。LevelDB使用了SSTable（Sorted String Table），意思是数据按照键的顺序存储在磁盘上。键是有序的，查找键可以采用二分搜索。但是磁盘的二分搜索会读取多个磁盘块。这时候只需要给每个磁盘块一个索引，告诉这个磁盘块里面存储键的范围，那么在查找时，可以先通过对索引进行二分搜索找到键所在的磁盘块，只需要读取这个磁盘块，便可以找到这个键。索引是一个稀疏索引，比较小，可以放在内存中缓存。

定时将MemTable的镜像写入磁盘中的一个SSTable，数据就会分为两个部分：一部分是内存中的MemTable里的数据，一部分是磁盘中SSTable的数据。当写入数据时，还是写入MemTable和日志，而后台线程定时将MemTable的镜像写入磁盘的SSTable中，如果很好的控制后台线程的写入频率，或者SSTable和日志不在同一个磁盘，那么写入依然是比较符合顺序写的，可以有很高的写性能。写入SSTable后，之前的日志就可以被淘汰了，因为之前的数据已经持久化到磁盘上了。这就同时解决了上面的两个问题，恢复时间和数据库容量。

使用定期写入MemTable镜像的方式，解决了恢复和数据库容量的问题，并且数据库具有持久化的功能，在满足这些条件的情况下，数据库依然有很高的写性能，符合LSM Tree的写多的场景。

但是，这又引入了一个新的问题：读变慢了。以前读的时候，只需要读取MemTable，现在还需要读取SSTable。随着SSTable不断地写入，SSTable会越来越多，当查找一个键时，可能需要读取多个SSTable，这就涉及了多次随机读，读取效率会很低。

如果把多个SSTable合并成一个大的SSTable，那么查找时，就不需要读取多个小的SSTable，而只需要读取一个大的SSTable，磁盘IO就会减少。

注：SSTable的合并本质上是去除无用的数据，加速读取的效率。

### Compaction

DB有一个后台线程负责将memtable持久化成sstable，以及均衡整个DB各个level层的sstable。

**compact分为minor compaction和major compaction。memtable持久化成sstable称为minor compaction，level(n)和level(n+1)之间某些sstable的merge称为major compaction**。

将多个小SSTable合并成一个大SSTable，可以解决查找的效率问题。但是，如果不断地有新的小SSTable进来，这些小SSTable都需要和这个大的SSTable进行合并，不管多大的SSTable来合并，都需要读取所有的磁盘数据，并且写入所有的磁盘数据。

大的SSTable是按键顺序存储的，可以将大SSTable进行分割，分割成多个小SSTable，每个SSTable都包含一段键的范围，而每个SSTable键的范围是不重复的，并且是按顺序排列的。这时候物理上存在多个SSTable文件，但是逻辑上依然是有一个大的SSTable。在查找的时候只需要多做一步，根据每个小的SSTable包含键的范围，可以做一个二分搜索，就可以找到实际的键在哪个小SSTable文件里。这样依然只需要读取一个SSTable，但是却使用了多个小文件代替一个大文件。这样的好处就是当有新SSTable需要合并到这个逻辑上的大SSTable时，只需要找到和新SSTable的键范围有重合的小SSTable的物理文件进行合并，这样就可以降低合并需要读取的数据量。

这样磁盘文件实际上分为两部分：一部分是MemTable直接写入的SSTable，另一部分是逻辑上的大SSTable。这实际上就是LevelDB里面的Level 0和Level 1。Level 0文件的键范围可能有重叠，而Level 1不会有重叠。而读取的时候，要读取Level 0和Level 1的数据，如果限制Level 0的文件数量，磁盘的读IO依然可以控制在一个常数范围内。

随着数据越写越多，Level 1越来越大，此时就算Level 0的SSTable很小，依然可能会和Level 1很多的SSTable文件重合，那么读写量依然很大。

这时候还需要改进，需要控制Level 1里所有文件的大小，那么多出的文件该怎么办呢？可以将它们再推到更高的一层Level 2中。Level 2里总文件大小设置为Level 1的10倍，在生成Level 1的SSTable时，控制大小使得最多与10个Level 2的SSTable重叠。如果需要将Level 1的一个SSTable合并到Level 2，需要读取的SSTable的数据量依然是一个常数级，是可控的。如果Level 2满了，可以将SSTable继续推向Level 3。因为每一Level的大小都是指数增长的，所以不需要几层，就可以放大量数据了。

这就是LevelDB名字的由来了，SSTable是分层存储的，Level 0的SSTable之间是重叠的，而Level 0以上的SSTable是有序不重叠的。SSTable在各Level之间移动的过程叫做Compaction，意思是让文件变得更加紧凑，易于查询。Level 0的SSTable会Compaction到Level 1，而Level 1的SSTable会Compaction到Level 2，随着Level的增大，每一层的文件总大小会以10倍增大，这样不但可以有大量的存储空间，而且每一次Compaction涉及的SSTable的数量都是可控的。Compaction实际上就是对输入的多个SSTable进行多路归并的过程。

随着Level的增多，读取更复杂了。要先读取MemTable，再读取Level 0的文件，Level 0可能有多个文件的键范围包括这个查找键，还需要读取Level 0以上的文件，每一Level最多有一个文件的键范围包括查找键。不过Level的数量有限，Level 0的文件数量也有限，所以需要读取的SSTable的数量依然是常数级，配合缓存、布隆过滤器等优化技术，可以提高读的性能。这是在读取性能和后台操作性能之间的折中，为了让写操作成为顺序写，而做的牺牲。

磁盘上有多个SSTable，需要知道每个SSTable属于哪一层，每一个SSTable的键范围，这些信息都存储在内存中。但是如果数据库重启了，就丢失了这些元信息，所以需要将它们持久化到磁盘。

### Manifest

Manifest：Manifest文件中记录SSTable在不同level的信息，包括每一层由哪些SSTable，每个SSTable的文件大小、最大key、最小key等信息。

MANIFEST是用来管理SSTable的，保存了SSTable的元信息。

对于当前数据有哪些SSTable，这些SSTable属于哪一层，每一个SSTable的键范围和文件大小等信息，需要持久化到磁盘上，下一次打开数据库的时候，就可以从磁盘上读取到这些元数据，恢复内存里的数据结构，这个持久化数据就存储在MANIFEST文件中。

随着Compaction的进行，元数据会改变，所以每次还需要将改变的元数据写到MANIFEST中。恢复元数据时，使用初始的元数据和各个改变恢复出最终的元数据。但是如果改变太多，MANIFEST太大，恢复就会太耗时，这时可以将当前的元数据写入到有一个新的MANIFEST中，而舍弃旧的MANIFEST。而CURRENT文件则存储了当前使用的MANIFEST文件是哪一个，写完MANIFEST后，需要将CURRENT指向新的MANIFEST。

### Current

Current：重启时，LevelDB会重新生成Manifest，所以Manifest文件可能同时存在多个，Current记录的是当前使用的Manifest文件名。

### TableCache

TableCache：TableCache用于缓存SSTable的文件描述符、索引和filter。

### BlockCache

BlockCache：SSTable的数据是被组织成一个个block。BlockCache用于缓存这些block（解压后）的数据。

### SequenceNumber

LevelDB中每次写操作(put/delete)都有一个版本，由sequence number来标识，整个DB有一个全局值保存当前使用的Sequence Number，key的排序以及snapshot都要依赖它。

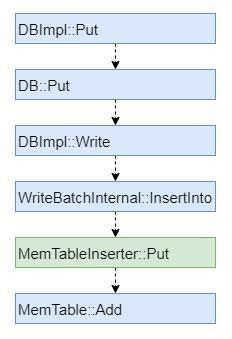
### 版本管理

将每次compact后的最新数据状态定义为一个version，也就是当前DB的元信息以及每层level的sstable的集合。**跟version有关的一个数据结构是VersionEdit，记录了一次version的变化，包括删除了哪些sstable，新增了哪些sstable**。old version + versionedit= new version。整个DB存在的所有version被VersionSet数据结构保存，这个数据结构包含：全局sequencenumber、filenumber、tablecache、每个level中下一次compact要选取的start\_key。

## 操作接口

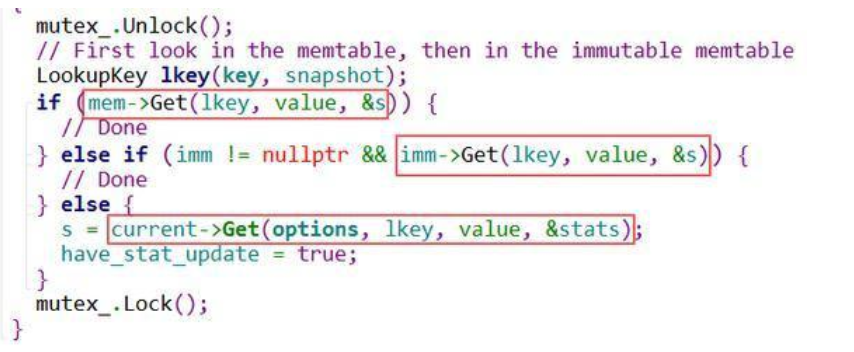
### 写流程

LevelDB的写操作包括设置key-value和删除key两种。需要指出的是这两种情况在LevelDB的处理上是一致的，删除操作其实是向LevelDB插入一条标识为删除的数据。下面我们先看一下LevelDB插入值的整体流程，具体如图所示。



### 读流程

读流程要比写流程简单一些，核心代码逻辑如图所示。



首先，生成内部查询所用的Key，该Key是由用户请求的UserKey拼接上Sequence生成的。其中Sequence可以用户提供或使用当前最新的Sequence，LevelDB可以保证仅查询在这个Sequence之前的写入。然后，用生成的Key，依次尝试从 Memtable，Immtable以及SST文件中读取，直到找到。

从SST文件中查找需要依次尝试在每一层中读取，得益于Manifest中记录的每个文件的key区间，我们可以很方便的知道某个key是否在文件中。Level0的文件由于直接由Immutable Dump 产生，不可避免的会相互重叠，所以需要对每个文件依次查找。对于其他层次，由于归并过程保证了其互相不重叠且有序，二分查找的方式提供了更好的查询效率。

可以看出同一个Key出现在上层的操作会屏蔽下层的。也因此删除Key时只需要在Memtable压入一条标记为删除的条目即可。被其屏蔽的所有条目会在之后的归并过程中清除。

# 应用场景

Level适用于写多读少的场景。