# 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>

## 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。

## sstable

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**。

## WAL

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。

## 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SequenceNumber，key的排序以及snapshot都要依赖它。

## Version

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。

## FileNumber

DB创建文件时将FileNumber加上特定的后缀作为文件名，FileNumber在内部是一个uint64\_t类型，并且全局递增。不同类型的文件的拓展名不同，例如sstable文件是.sst，wal日志文件是.log。LevelDB有以下文件类型：

enum FileType {

kLogFile,

kDBLockFile,

kTableFile,

kDescriptorFile,

kCurrentFile,

kTempFile,

kInfoLogFile // Either the current one, or an old one

};

## Key

LevelDB对于用户输入的key做了不同的处理，user\_key表示用户输入的key，internal\_key是DB内部使用的key，在uers\_key的基础上添加了sequence number和value type。

## compact

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。

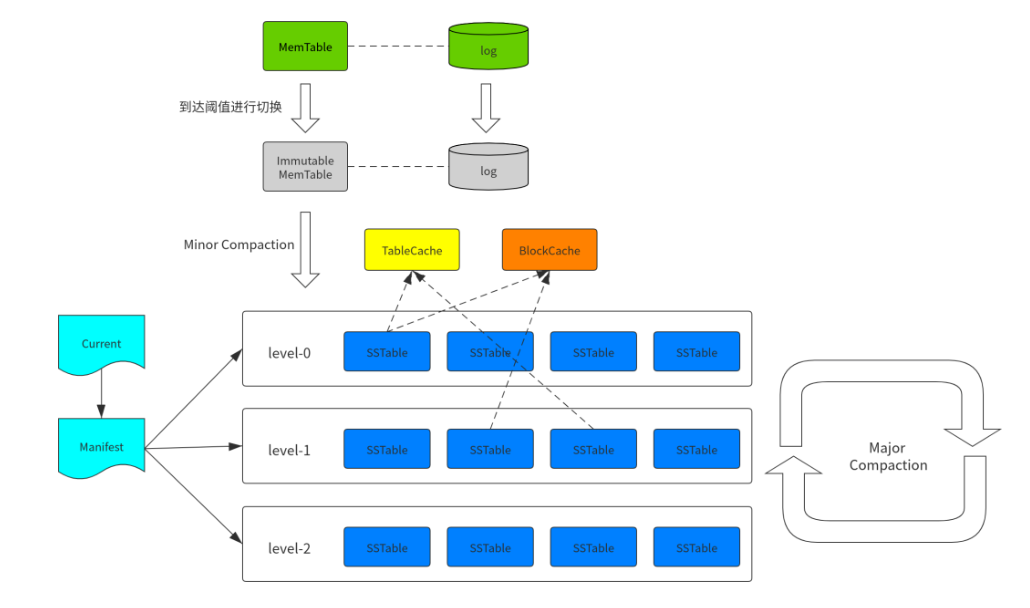
**compact分为minor compaction和major compaction。memtable持久化成sstable称为minor compaction，level(n)和level(n+1)之间某些sstable的merge称为major compaction**。

## 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。

# 架构



上图简单展示了LevelDB的整体架构。

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。

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。

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。

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。

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。

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。

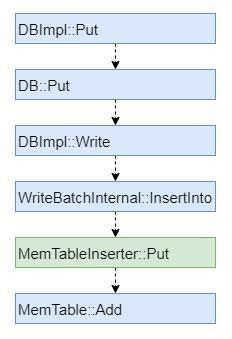
TableCache：TableCache用于缓存SSTable的文件描述符、索引和filter。

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。

# 原理

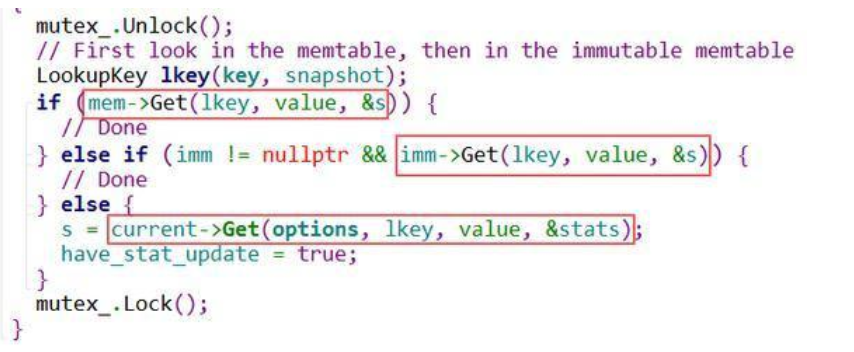
## 写流程

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。