Sstables的文件管理

每个Sstable都是一个文件

- 文件信息FileMetaData

```
struct FileMetaData {
  int refs;
  int allowed_seeks;          // Seeks allowed until compaction
  uint64_t number;
  uint64_t file_size;         // File size in bytes
  InternalKey smallest;       // Smallest internal key served by table
  InternalKey largest;        // Largest internal key served by table

  FileMetaData() : refs(0), allowed_seeks(1 << 30), file_size(0) { }
};
```

- 文件管理模块
  - Version Set

Version，Version Edit，Version Set

- Version
  - 每次Compaction都会使得Sstables变化，每个Sstables的状态称之为一个Version。
- 从旧到新所有Version都存储在Version Set中，所以Version Set就是Version的一个链表。
- Version之间是通过增量的方式演进的，即
  - Version0 + VersionEdit = Version1
  - 两个Version之间的差异称之为一个Version Edit

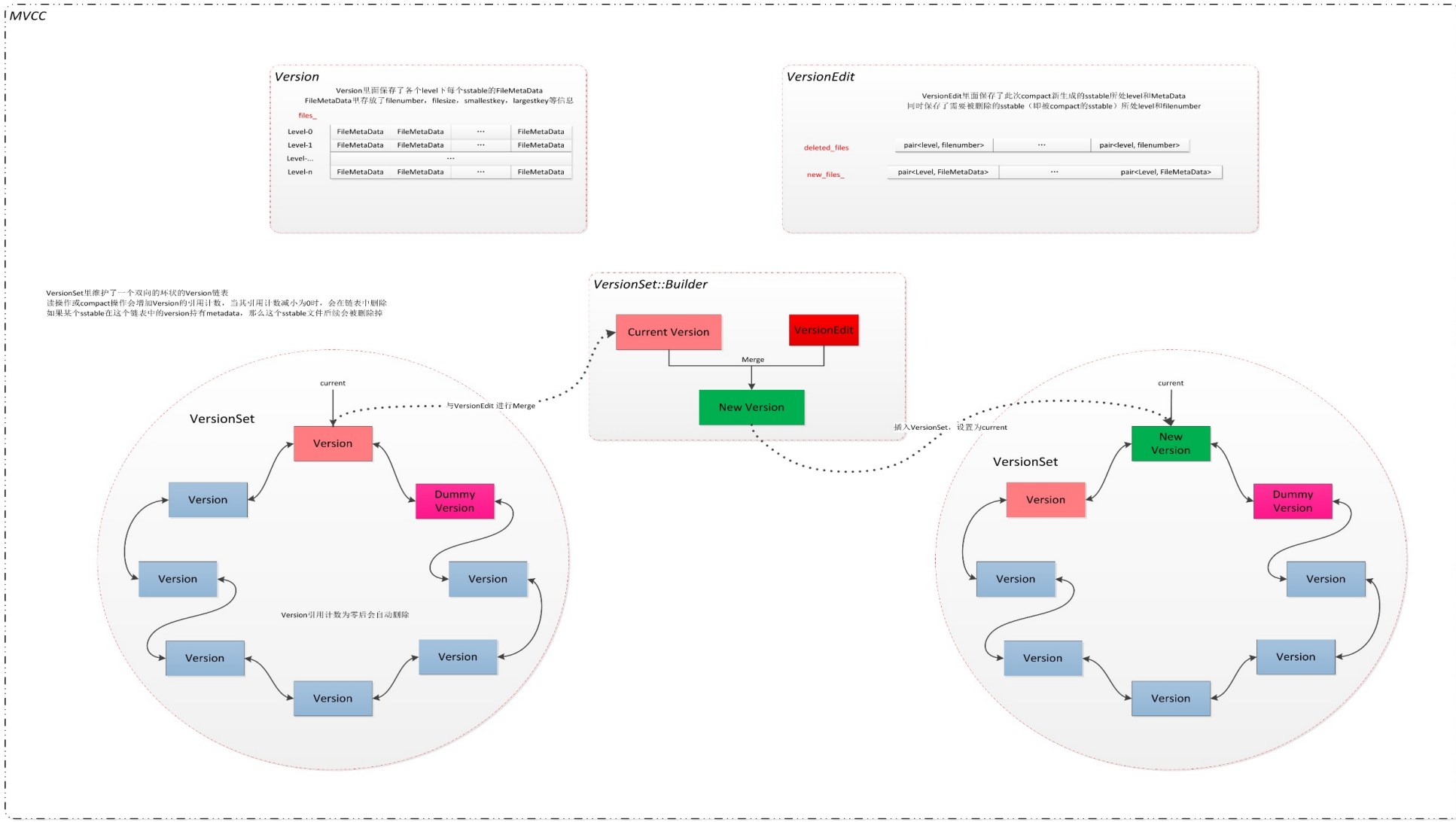
Version

- /*该version下的所有level的所有sstable文件，每个文件由FileMetaData表示*/
  std::vector<FileMetaData*>
  files_[config::kNumLevels];
- Version就是文件信息的一个二维数组

Version Edit

- 主要元素
  - DeletedFileSet deleted_files_; //删除的文件
  - std::vector< std::pair<int, FileMetaData> > new_files_; //新产生的文件

Version与Version Edit



将Version Edit Append到当前Version

- VersionSet::LogAndApply(VersionEdit* edit, port::Mutex* mu){

  //创建一个新的Version v，并把新的edit变动保存到v中。Builder很重要，我们后面进行分析 Version* v = new Version(this);

  { *//这边是函数名字apply的含义，应用versionedit生成新版本v*

  Builder builder(this, current_);

  builder.Apply(edit); // 解析edit, 生成一个临时结构

  builder.SaveTo(v); //将edit append

  }

- }

Version set的持久化：manifest

- 理论上数据库重新启动或者恢复，当前的version也需要回复。可以对version set 做持久化
  - Meta信息
- 每次Append VersionEdit之后将 edit 持久化

```
VersionSet::LogAndApply{
        edit->EncodeTo(&record); //将version edit编码成一个log record
        s = descriptor_log_->AddRecord(record);
       s = descriptor_file_->Sync();   //将日志刷盘
}
```

Version Set的持久化

- 如果不做version的持久化，系统是否也能恢复？
  - 可以，因为可以扫描所有sstable，重新生成version，但效率较低

Compaction的流程

总体流程

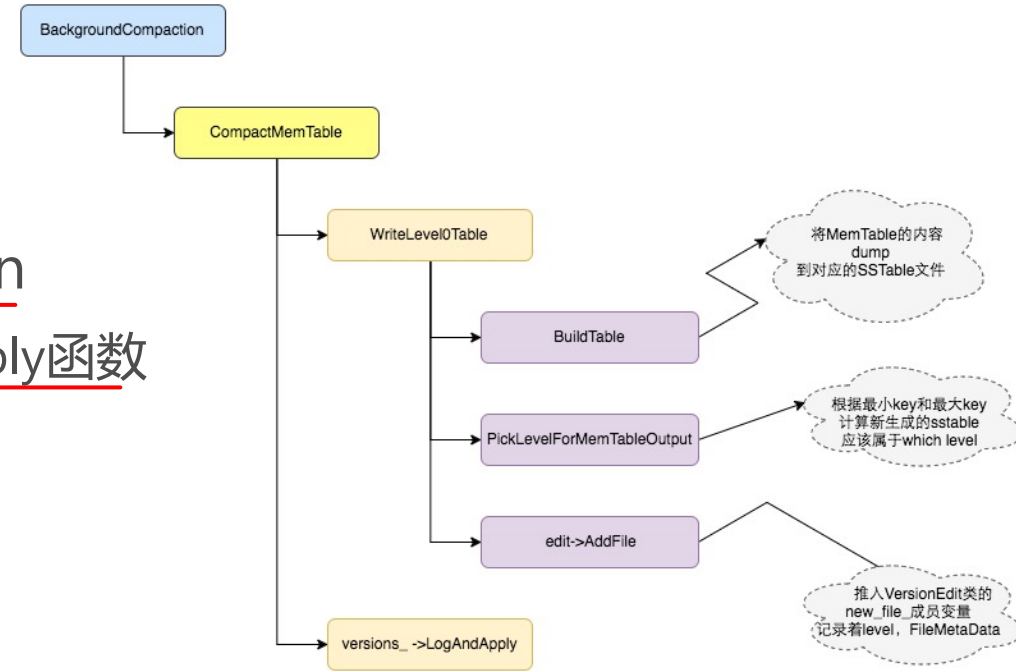
- DBImpl::MaybeScheduleCompaction()->BackGroundCompaction()->MaybeScheduleCompaction
  - 调用上述函数的地方都可能触发compaction
  - 这是一个循环调用
  - 主要有两个地方 PPT P17, P20

Compaction的分类

- BackGroundCompaction()包含
  - Minor Compaction
    - 内存到Level 0
  - Major Compaction
    - Sstables之间的合并
  - Manual Compaction
    - 手工对一个范围内数据做Compaction

Minor Compaction流程

- 系统调用
  - DBImpl::CompactMemTable()->WriteLevel0Table()
- 流程
  - 构建Sstable
  - 寻找合适的Level
  - 生成version edit
  - Apply到当前version
    - 前面LogAndApply函数



Minor Compaction触发时机

```
Status DBImpl::Write(const WriteOptions&
options, WriteBatch* my_batch) {
        Status status = MakeRoomForWrite(my_batch ==
        nullptr);
        // 每次插入操作都会检查是否memtable是否超过限制
        ，触发MaybeScheduleCompaction()函数
}
```

产生Version Edit

- 在BuildTable之后
  - Recall构建(刷盘)一个Sstable之后其实我们已经知道了文件的meta信息
  - 将传递给meta参数(指针)，并产生一个version edit

```
s = BuildTable(dbname_, env_, options_, table_cache_, iter, &meta);

edit->AddFile(level, meta.number, meta.file_size,  meta.smallest, meta.largest);
```

Level的选择

- int version::PickLevelForMemTableOutput
  - 从Level 0到Level 2之间选择一层
  - 一定程度上减少写放大
  - 如果与某个Level 0的Sstable有交集，放入Level 0
  - 如果与某个Level 1的Sstable有交集，放入Level 0
  - 如果与某个Level 2的Sstable有交集，放入Level 1
  - 否则放入 Level 2

Major Compaction流程

- 确定参与compaction的文件
  - 确定level n层
  - 确定level n+1层
- 合并文件（主要是归并排序生成文件）
  - DoCompactionWork
- 更新version
  - DBImpl::InstallCompactionResults

    ->LogAndApply函数

Major Compaction

- Compaction的条件
  - 条件1： 某个level sstables太多
    - 选择该层部分sstable加入compaction,防止瞬时写入过大
    - Versionset中记有每层上次compaction到的最大的key
      - std::vector< std::pair<int, InternalKey> > compact_pointers_;
      - 选择比该key大的下一个文件
  - 条件2：某个level的某个sstable 不成功seek 次数太多
    - 代码中用allowed_seeks标识

条件一

```
void VersionSet::Finalize(Version* v) {
//每次Compacktion后预计算下次compaction的内容
  for (int level = 0; level < config::kNumLevels-1; level++) {
   if (level == 0) {
          score = v->files_[level].size() /static_cast<double>(config::kL0_CompactionTrigger);
           //如果是Level0， 主要是文件数目不能超过配置值
    } else {
       const uint64_t level_bytes = TotalFileSize(v->files_[level]);
      score = static_cast<double>(level_bytes) / MaxBytesForLevel(options_, level);
      //如果是Level1, 文件大小之和不能超过配置值
    }
    if (score > best_score) {
      best_level = level;
      best_score = score;
    }
  }
  v->compaction_level_ = best_level;  //预存最需要Compaction的level
  v->compaction_score_ = best_score;
}
```

条件二

```
Status DBImpl::Get(const ReadOptions& options,  const Slice& key,  std::string* value) {
        if (have_stat_update && current->UpdateStats(stats)) {
                MaybeScheduleCompaction();
              //红色函数中会将某个Sstable的allowed_seeks次数减一
        }
}
```

思路: 在查找某个key的过程中，如果level n的某个sstable(范围包含这个key)，但总是查找不成功，但在level n+1中找到，意味着当前sstable产生了一次无效读。 对经常无效读的sstable, 将其compaction到下一层。

Compaction参与文件

- 确定参与Compaction的参与文件，分成两部分
  - 第一部分确定level n的参与文件
    - VersionSet::PickCompaction()
  - 第二部分确定level n + 1的参与文件，当然也可能没有 level n + 1的文件需要参与
    - VersionSet::SetupOtherInputs(Compaction* c)

PickCompaction

- 选择合并的level及Sstable(n层)

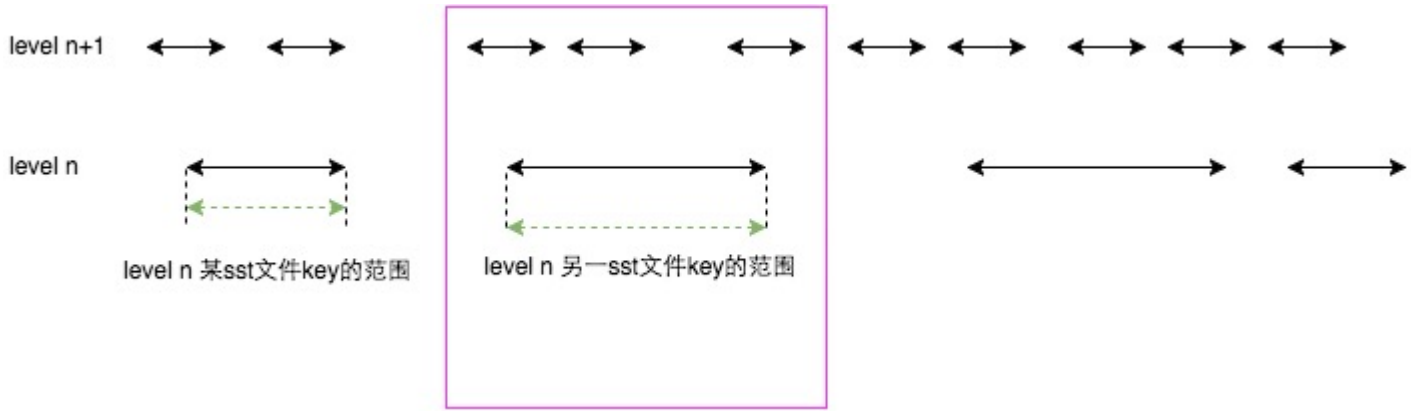
```
Compaction* VersionSet::PickCompaction(){
        const bool size_compaction = (current_->compaction_score_ >= 1);
        const bool seek_compaction = (current_->file_to_compact_ != NULL);
        if (size_compaction) {
                for (size_t i = 0; i < current_->files_[level].size(); i++) {
                        if (compact_pointer_[level].empty() || icmp_.Compare(f->largest.Encode(), compact_pointer_[level]) > 0)
        }//部分sstable进入compaction
        else if (seek_compaction) {
        c->inputs_[0].push_back(current_->file_to_compact_);
          //allow_seek的sstable进行compaction
}
```

compaction Level 0的特殊处理

- 代码Status Version::Get(const ReadOptions& options, const LookupKey& k, std::string* value, GetStats* stats) 中
  - 若某个level 0的 Sstable通过条件二触发了合并， 将其合并至level 1, 查询将会出现不一致问题（未读到最近的更新）
  - 在PickCompaction中与该sstable有重合的sstable都加入到compaction文件中来

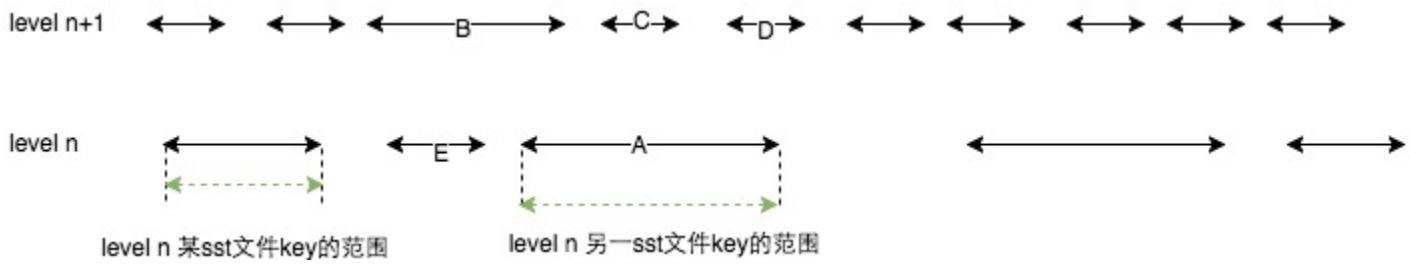
n+1层的参与文件

- n+1层有交集的范围都要参与compaction



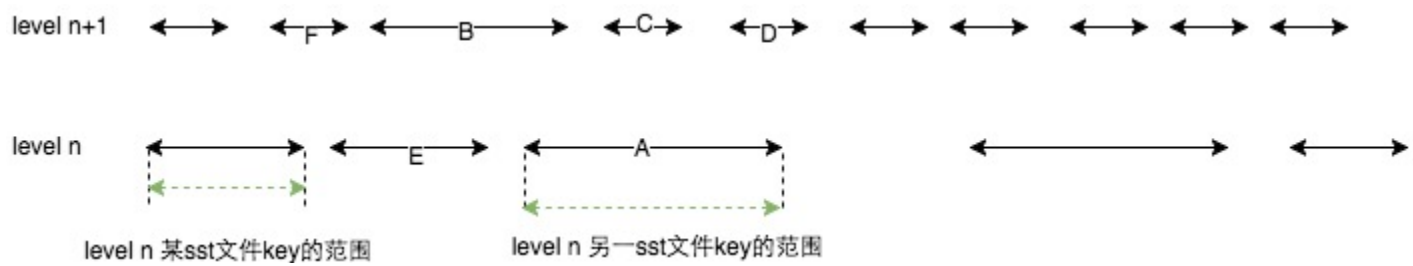
参与文件

- 对A, 选择B, C, D进行参与，但进一步的发现n层的E也可以Compaction



参与文件

- 对A, 选择B, C, D进行参与，但E不完全包含于BCD, 进一步参与会引起F也要参与，这种情况只对A, BCD进行Compaction



Compaction的参与文件

- VersionSet::SetupOtherInputs(Compaction* c){

```
/获取level n所有参战文件的最小key和最大key/
GetRange(c->inputs_[0], &smallest, &largest);
/*根据最小key和最大 key，计算level n + 1的文件中于该范围有重叠的文件，放入c->inputs_[1]中*/
current_->GetOverlappingInputs(level+1, &smallest, &largest, &c->inputs_[1]);
// 通过P29页的方式观察Level n+1的参于文件
 if (!c->inputs_[1].empty()) {
```