LevelDB 源码分析「九、Compaction」

2019.09.13 SF-Zhou

祝大家中秋快乐！

LevelDB 源码分析系列也步入尾声，本篇将分析 LevelDB 中至关重要的 Compaction 过程，依然从代码的角度出发。建议大家同时阅读参考文献 1 了解 Compaction 的作用和过程的描述。

1. 触发 Compaction

本系列第三篇中描述了内存数据库转为 Sorted Table 的过程，其中会执行 DBImpl::BackgroundCompaction 这一后台任务：

```
void DBImpl::BackgroundCompaction() {
  mutex_.AssertHeld();

  if (imm_ != nullptr) {
    CompactMemTable();
    return;
  }

  Compaction* c;
  bool is_manual = (manual_compaction_ != nullptr);
  InternalKey manual_end;
```

```
  if (is_manual) {
    ...
  } else {
    c = versions_->PickCompaction();
  }


  ...
}
```

现在假设 imm_ 为空，并且不考虑手动 Compaction，那么这里会执行 versions_->PickCompaction 去选择一个 Compaction，其实现位于 db/version_set.cc：

```
Compaction* VersionSet::PickCompaction() {
  Compaction* c;
  int level;

  // We prefer compactions triggered by too much data in a level over
  // the compactions triggered by seeks.
  const bool size_compaction = (current_->compaction_score_ >= 1);
  const bool seek_compaction = (current_->file_to_compact_ != nullptr);
  ...
}
```

这里会有两种需要 Compaction 的情况，一种是某一 Level 的分数超过了 1，一种是某一个文件的无效查询次数超过阈值。分数的计算位于版本更新之后的 VersionSet::Finalize：

```
void VersionSet::Finalize(Version* v) {
  // Precomputed best level for next compaction
  int best_level = -1;
  double best_score = -1;

  for (int level = 0; level < config::kNumLevels - 1; level++) {
    double score;
    if (level == 0) {
      // We treat level-0 specially by bounding the number of files
      // instead of number of bytes for two reasons:
      //
      // (1) With larger write-buffer sizes, it is nice not to do too
      // many level-0 compactions.
      //
      // (2) The files in level-0 are merged on every read and
      // therefore we wish to avoid too many files when the individual
      // file size is small (perhaps because of a small write-buffer
      // setting, or very high compression ratios, or lots of
      // overwrites/deletions).
      score = v->files_[level].size() /
              static_cast<double>(config::kL0_CompactionTrigger);
    } else {
      // Compute the ratio of current size to size limit.
      const uint64_t level_bytes = TotalFileSize(v->files_[level]);
      score =
          static_cast<double>(level_bytes) / MaxBytesForLevel(options_, level);
    }

    if (score > best_score) {
      best_level = level;
      best_score = score;
```

```
    }
  }

  v->compaction_level_ = best_level;


  v->compaction_score_ = best_score;
}
```

对于 0 层文件，当文件数量超过阈值（默认 4）时触发 Compaction；对于其他层的文件，当文件的总大小超过阈值（默认 10^lMB）时触发 Compaction。而一个文件的的查询次数阈值定义于 VersionSet::Builder::Apply：

```
// We arrange to automatically compact this file after
// a certain number of seeks.  Let's assume:
//   (1) One seek costs 10ms
//   (2) Writing or reading 1MB costs 10ms (100MB/s)
//   (3) A compaction of 1MB does 25MB of IO:
//         1MB read from this level
//         10-12MB read from next level (boundaries may be misaligned)
//         10-12MB written to next level
// This implies that 25 seeks cost the same as the compaction
// of 1MB of data.  I.e., one seek costs approximately the
// same as the compaction of 40KB of data.  We are a little
// conservative and allow approximately one seek for every 16KB
// of data before triggering a compaction.
f->allowed_seeks = static_cast<int>((f->file_size / 16384U));
if (f->allowed_seeks < 100) f->allowed_seeks = 100;
```

英文注释写得十分详细。首先假设：

1. 一次查询耗时 10ms；
2. 读/写 1MB 耗时 10ms （假设速度 100MB/s）；

3. 1MB 的 Compaction 需要做 25 MB 的 IO
    1. 本层读 1MB；
    2. 下一层读 10-12 MB
    3. Compaction 后写 10-12 MB

整体来看，1MB 的数据做 25 次查询和 Compaction 的时间差不多，1 次查询就相当于做 40KB 数据的 Compaction。LevelDB 将其设为更保守的 16KB，进而一个文件的查询次数阈值设定为 FileSize / 16KB 。当一次查询中读取了多个文件，则将第一个文件的查询次数 +1，直到其超过阈值、触发 Compaction。继续看 VersionSet::PickCompaction：

```
Compaction* VersionSet::PickCompaction() {
  ...

  if (size_compaction) {
    level = current_->compaction_level_;
    assert(level >= 0);
    assert(level + 1 < config::kNumLevels);
    c = new Compaction(options_, level);

    // Pick the first file that comes after compact_pointer_[level]
    for (size_t i = 0; i < current_->files_[level].size(); i++) {
      FileMetaData* f = current_->files_[level][i];
```

```
    if (compact_pointer_[level].empty() ||
        icmp_.Compare(f->largest.Encode(), compact_pointer_[level]) > 0) {
      c->inputs_[0].push_back(f);
      break;
    }
  }
  if (c->inputs_[0].empty()) {
    // Wrap-around to the beginning of the key space
    c->inputs_[0].push_back(current_->files_[level][0]);
  }
} else if (seek_compaction) {
  level = current_->file_to_compact_level_;
  c = new Compaction(options_, level);
  c->inputs_[0].push_back(current_->file_to_compact_);
} else {
  return nullptr;
}

c->input_version_ = current_;
c->input_version_->Ref();

// Files in level 0 may overlap each other, so pick up all overlapping ones
if (level == 0) {
  InternalKey smallest, largest;
  GetRange(c->inputs_[0], &smallest, &largest);
  // Note that the next call will discard the file we placed in
  // c->inputs_[0] earlier and replace it with an overlapping set
  // which will include the picked file.
  current_->GetOverlappingInputs(0, &smallest, &largest, &c->inputs_[0]);
  assert(!c->inputs_[0].empty());
}
```

```
  SetupOtherInputs(c);

  return c;
}
```

对于数据大小触发的 Compaction，会选取 compact_pointer_ 后的第一个文件作为 Compaction 对象，即本层上一次 Compaction 区间之后的文件；而查询次数触发的 Compaction 其本身对应一个文件。对于 0 层文件，因为之间存在 Overlap，需要将存在重叠的文件都加入 Compaction 集合里。至此本层的文件选择完毕。

2. 扩大 Compaction 文件集合

VersionSet::PickCompaction 随后执行 SetupOtherInputs 以扩大 Compaction 文件集合：

```
// Finds the largest key in a vector of files. Returns true if files it not
// empty.
bool FindLargestKey(const InternalKeyComparator& icmp,
                    const std::vector<FileMetaData*>& files,
                    InternalKey* largest_key) {
  if (files.empty()) {
    return false;
  }
  *largest_key = files[0]->largest;
  for (size_t i = 1; i < files.size(); ++i) {
    FileMetaData* f = files[i];
    if (icmp.Compare(f->largest, *largest_key) > 0) {
      *largest_key = f->largest;
```

```
    }
  }
  return true;
}


// Finds minimum file b2=(l2, u2) in level file for which l2 > u1 and
// user_key(l2) = user_key(u1)
FileMetaData* FindSmallestBoundaryFile(
    const InternalKeyComparator& icmp,
    const std::vector<FileMetaData*>& level_files,
    const InternalKey& largest_key) {
  const Comparator* user_cmp = icmp.user_comparator();
  FileMetaData* smallest_boundary_file = nullptr;
  for (size_t i = 0; i < level_files.size(); ++i) {
    FileMetaData* f = level_files[i];
    if (icmp.Compare(f->smallest, largest_key) > 0 &&
        user_cmp->Compare(f->smallest.user_key(), largest_key.user_key()) ==
            0) {
      if (smallest_boundary_file == nullptr ||
          icmp.Compare(f->smallest, smallest_boundary_file->smallest) < 0) {
        smallest_boundary_file = f;
      }
    }
  }
  return smallest_boundary_file;
}

// Extracts the largest file b1 from |compaction_files| and then searches for a
// b2 in |level_files| for which user_key(u1) = user_key(l2). If it finds such a
// file b2 (known as a boundary file) it adds it to |compaction_files| and then
```

```
// searches again using this new upper bound.
//
// If there are two blocks, b1=(l1, u1) and b2=(l2, u2) and
// user_key(u1) = user_key(l2), and if we compact b1 but not b2 then a
// subsequent get operation will yield an incorrect result because it will

// return the record from b2 in level i rather than from b1 because it searches
// level by level for records matching the supplied user key.
//
// parameters:
//   in     level_files:      List of files to search for boundary files.
//   in/out compaction_files: List of files to extend by adding boundary files.
void AddBoundaryInputs(const InternalKeyComparator& icmp,
                       const std::vector<FileMetaData*>& level_files,
                       std::vector<FileMetaData*>* compaction_files) {
  InternalKey largest_key;

  // Quick return if compaction_files is empty.
  if (!FindLargestKey(icmp, *compaction_files, &largest_key)) {
    return;
  }

  bool continue_searching = true;
  while (continue_searching) {
    FileMetaData* smallest_boundary_file =
        FindSmallestBoundaryFile(icmp, level_files, largest_key);

    // If a boundary file was found advance largest_key, otherwise we're done.
    if (smallest_boundary_file != NULL) {
      compaction_files->push_back(smallest_boundary_file);
      largest_key = smallest_boundary_file->largest;
```

```
    } else {
      continue_searching = false;
    }
  }
}


void VersionSet::SetupOtherInputs(Compaction* c) {
  const int level = c->level();
  InternalKey smallest, largest;

  AddBoundaryInputs(icmp_, current_->files_[level], &c->inputs_[0]);
  ...
}
```

首先执行的是 AddBoundaryInputs 。其英文注释中解释地非常详细：当 Compaction 的范围为 $[l1, u1]$ 时，该范围的数据将会被移动到 Level+1。如果当前 Level 存在文件 $[l2, u2]$，并且 user_key(u1) = user_key(l2)，那么下一次查询 user_key(u1) 时会在 Level 层提前返回旧的数据！故需要将受影响的文件全部加到 Compaction 文件范围中。继续看 VersionSet::SetupOtherInputs：

```
void VersionSet::SetupOtherInputs(Compaction* c) {
  ...
  GetRange(c->inputs_[0], &smallest, &largest);

  current_->GetOverlappingInputs(level + 1, &smallest, &largest,
                                 &c->inputs_[1]);

  // Get entire range covered by compaction
```

```
InternalKey all_start, all_limit;
GetRange2(c->inputs_[0], c->inputs_[1], &all_start, &all_limit);

// See if we can grow the number of inputs in "level" without
// changing the number of "level+1" files we pick up.

if (!c->inputs_[1].empty()) {
  std::vector<FileMetaData*> expanded0;
  current_->GetOverlappingInputs(level, &all_start, &all_limit, &expanded0);
  AddBoundaryInputs(icmp_, current_->files_[level], &expanded0);
  const int64_t inputs0_size = TotalFileSize(c->inputs_[0]);
  const int64_t inputs1_size = TotalFileSize(c->inputs_[1]);
  const int64_t expanded0_size = TotalFileSize(expanded0);
  if (expanded0.size() > c->inputs_[0].size() &&
      inputs1_size + expanded0_size <
          ExpandedCompactionByteSizeLimit(options_)) {
    InternalKey new_start, new_limit;
    GetRange(expanded0, &new_start, &new_limit);
    std::vector<FileMetaData*> expanded1;
    current_->GetOverlappingInputs(level + 1, &new_start, &new_limit,
                                   &expanded1);
    if (expanded1.size() == c->inputs_[1].size()) {
      Log(options_->info_log,
          "Expanding@%d %d+%d (%ld+%ld bytes) to %d+%d (%ld+%ld bytes)\n",
          level, int(c->inputs_[0].size()), int(c->inputs_[1].size()),
          long(inputs0_size), long(inputs1_size), int(expanded0.size()),
          int(expanded1.size()), long(expanded0_size), long(inputs1_size));
      smallest = new_start;
      largest = new_limit;
      c->inputs_[0] = expanded0;
      c->inputs_[1] = expanded1;
```

```
        GetRange2(c->inputs_[0], c->inputs_[1], &all_start, &all_limit);
      }
    }
  }


  // Compute the set of grandparent files that overlap this compaction
  // (parent == level+1; grandparent == level+2)
  if (level + 2 < config::kNumLevels) {
    current_->GetOverlappingInputs(level + 2, &all_start, &all_limit,
                                   &c->grandparents_);
  }

  // Update the place where we will do the next compaction for this level.
  // We update this immediately instead of waiting for the VersionEdit
  // to be applied so that if the compaction fails, we will try a different
  // key range next time.
  compact_pointer_[level] = largest.Encode().ToString();
  c->edit_.SetCompactPointer(level, largest);
}
```

首先在 Level+1 层将所有存在重叠的文件加入 Compaction 文件集合里，更新 Compaction 的区间 [all_start, all_limit] 。再回过头来使用新区间获得 Level 层重叠的文件 expanded0 ，如果新的数据大小在阈值以内且不会改变 Level+1 层选择的文件，那么则将 Level 层的文件集合更新为 expanded0 。最后将当前 Level 的 compact_pointer_ 设为当前 Compaction 的最大键。至此扩大 Compaction 文件集合结束，VersionSet::PickCompaction 也返回了 Compaction 对象。

3. 执行 Compaction

回到 DBImpl::BackgroundCompaction：

```
struct DBImpl::CompactionState {
  // Files produced by compaction
  struct Output {
    uint64_t number;
    uint64_t file_size;
    InternalKey smallest, largest;
  };

  Output* current_output() { return &outputs[outputs.size() - 1]; }

  explicit CompactionState(Compaction* c)
      : compaction(c),
        smallest_snapshot(0),
        outfile(nullptr),
        builder(nullptr),
        total_bytes(0) {}

  Compaction* const compaction;

  // Sequence numbers < smallest_snapshot are not significant since we
  // will never have to service a snapshot below smallest_snapshot.
  // Therefore if we have seen a sequence number S <= smallest_snapshot,
  // we can drop all entries for the same key with sequence numbers < S.
  SequenceNumber smallest_snapshot;
```

```
  std::vector<Output> outputs;

  // State kept for output being generated
  WritableFile* outfile;
  TableBuilder* builder;


  uint64_t total_bytes;
};

void DBImpl::BackgroundCompaction() {
  ...

  Status status;
  if (c == nullptr) {
    // Nothing to do
  } else if (!is_manual && c->IsTrivialMove()) {
    ...
  } else {
    CompactionState* compact = new CompactionState(c);
    status = DoCompactionWork(compact);
    if (!status.ok()) {
      RecordBackgroundError(status);
    }
    CleanupCompaction(compact);
    c->ReleaseInputs();
    DeleteObsoleteFiles();
  }
  delete c;

  if (status.ok()) {
```

```
    // Done
  } else if (shutting_down_.load(std::memory_order_acquire)) {
    // Ignore compaction errors found during shutting down
  } else {
    Log(options_.info_log, "Compaction error: %s", status.ToString().c_str());

  }

  if (is_manual) {
    ...
  }
}
```

不考虑手动模式和 TrivialMove，接下来会根据 Compaction 对象构建 CompactionState，并执行 DBImpl::DoCompactionWork：

```
Status DBImpl::DoCompactionWork(CompactionState* compact) {
  const uint64_t start_micros = env_->NowMicros();
  int64_t imm_micros = 0;  // Micros spent doing imm_ compactions

  Log(options_.info_log, "Compacting %d@%d + %d@%d files",
      compact->compaction->num_input_files(0), compact->compaction->level(),
      compact->compaction->num_input_files(1),
      compact->compaction->level() + 1);

  assert(versions_->NumLevelFiles(compact->compaction->level()) > 0);
  assert(compact->builder == nullptr);
  assert(compact->outfile == nullptr);
  if (snapshots_.empty()) {
    compact->smallest_snapshot = versions_->LastSequence();
```

```
  } else {
    compact->smallest_snapshot = snapshots_.oldest()->sequence_number();
  }

  Iterator* input = versions_->MakeInputIterator(compact->compaction);

  ...
}
```

compact->smallest_snapshot 是为了让当前的 Snapshot 的数据在 Compaction 过程中不丢失。 versions_->MakeInputIterator 返回 Compaction 文件集合的合并迭代器：

```
Iterator* VersionSet::MakeInputIterator(Compaction* c) {
  ReadOptions options;
  options.verify_checksums = options_->paranoid_checks;
  options.fill_cache = false;

  // Level-0 files have to be merged together.  For other levels,
  // we will make a concatenating iterator per level.
  // TODO(opt): use concatenating iterator for level-0 if there is no overlap
  const int space = (c->level() == 0 ? c->inputs_[0].size() + 1 : 2);
  Iterator** list = new Iterator*[space];
  int num = 0;
  for (int which = 0; which < 2; which++) {
    if (!c->inputs_[which].empty()) {
      if (c->level() + which == 0) {
        const std::vector<FileMetaData*>& files = c->inputs_[which];
        for (size_t i = 0; i < files.size(); i++) {
          list[num++] = table_cache_->NewIterator(options, files[i]->number,
                                                  files[i]->file size);
```

```
        }
      } else {
        // Create concatenating iterator for the files from this level
        list[num++] = NewTwoLevelIterator(
            new Version::LevelFileNumIterator(icmp_, &c->inputs_[which]),
            &GetFileIterator, table_cache_, options);
      }
    }
  }
  assert(num <= space);
  Iterator* result = NewMergingIterator(&icmp_, list, num);
  delete[] list;
  return result;
}
```

继续看 DBImpl::DoCompactionWork：

```
Status DBImpl::DoCompactionWork(CompactionState* compact) {
  ...
  // Release mutex while we're actually doing the compaction work
  mutex_.Unlock();

  input->SeekToFirst();
  Status status;
  ParsedInternalKey ikey;
  std::string current_user_key;
  bool has_current_user_key = false;
  SequenceNumber last_sequence_for_key = kMaxSequenceNumber;
  while (input->Valid() && !shutting_down_.load(std::memory_order_acquire)) {
```

```
// Prioritize immutable compaction work
if (has_imm_.load(std::memory_order_relaxed)) {
  const uint64_t imm_start = env_->NowMicros();
  mutex_.Lock();
  if (imm_ != nullptr) {

    CompactMemTable();
    // Wake up MakeRoomForWrite() if necessary.
    background_work_finished_signal_.SignalAll();
  }
  mutex_.Unlock();
  imm_micros += (env_->NowMicros() - imm_start);
}

Slice key = input->key();
if (compact->compaction->ShouldStopBefore(key) &&
    compact->builder != nullptr) {
  status = FinishCompactionOutputFile(compact, input);
  if (!status.ok()) {
    break;
  }
}

// Handle key/value, add to state, etc.
bool drop = false;
if (!ParseInternalKey(key, &ikey)) {
  // Do not hide error keys
  current_user_key.clear();
  has_current_user_key = false;
  last_sequence_for_key = kMaxSequenceNumber;
} else {
```

```
  if (!has_current_user_key ||
      user_comparator()->Compare(ikey.user_key, Slice(current_user_key)) !=
          0) {
    // First occurrence of this user key
    current_user_key.assign(ikey.user_key.data(), ikey.user_key.size());

    has_current_user_key = true;
    last_sequence_for_key = kMaxSequenceNumber;
  }

  if (last_sequence_for_key <= compact->smallest_snapshot) {
    // Hidden by an newer entry for same user key
    drop = true;  // (A)
  } else if (ikey.type == kTypeDeletion &&
             ikey.sequence <= compact->smallest_snapshot &&
             compact->compaction->IsBaseLevelForKey(ikey.user_key)) {
    // For this user key:
    // (1) there is no data in higher levels
    // (2) data in lower levels will have larger sequence numbers
    // (3) data in layers that are being compacted here and have
    //     smaller sequence numbers will be dropped in the next
    //     few iterations of this loop (by rule (A) above).
    // Therefore this deletion marker is obsolete and can be dropped.
    drop = true;
  }

  last_sequence_for_key = ikey.sequence;
}

if (!drop) {
  // Open output file if necessary
```

```
      if (compact->builder == nullptr) {
        status = OpenCompactionOutputFile(compact);
        if (!status.ok()) {
          break;
        }

      }
      if (compact->builder->NumEntries() == 0) {
        compact->current_output()->smallest.DecodeFrom(key);
      }
      compact->current_output()->largest.DecodeFrom(key);
      compact->builder->Add(key, input->value());

      // Close output file if it is big enough
      if (compact->builder->FileSize() >=
          compact->compaction->MaxOutputFileSize()) {
        status = FinishCompactionOutputFile(compact, input);
        if (!status.ok()) {
          break;
        }
      }
    }

    input->Next();
  }
```

一个巨大的循环。首先判断是否已经 shutting_down_，如果已经关闭了，则终止当前的 Compaction 过程；随后判断当前是否有 imm_，如果存在的话则也先执行 CompactMemTable；再来判断当前输出的文件是否可以结束了，如果是的话就执行 FinishCompactionOutputFile 完成当前文件

FinishCompactionOutputFile 完成当前文件。

接下来是是否丢弃键值对的判定。如果某个 user_key 的非最新版本小于快照版本，则可以直接丢弃，因为读最新的版本就足够了；如果某个删除操作的版本小于快照版本，并且在更高层没有相同的 user_key，那么这个删除操作及其之前更早的插入操作可以同时丢弃了。

对于没有丢弃的键值对，将其写入当前的 Table Builder。当输出的大小超过阈值，同样执行 FinishCompactionOutputFile：

```
Status DBImpl::OpenCompactionOutputFile(CompactionState* compact) {
  assert(compact != nullptr);
  assert(compact->builder == nullptr);
  uint64_t file_number;
  {
    mutex_.Lock();
    file_number = versions_->NewFileNumber();
    pending_outputs_.insert(file_number);
    CompactionState::Output out;
    out.number = file_number;
    out.smallest.Clear();
    out.largest.Clear();
    compact->outputs.push_back(out);
    mutex_.Unlock();
  }

  // Make the output file
  std::string fname = TableFileName(dbname_, file_number);
```

```
  std::string fname = TableFileName(dbname_, file_number);
  Status s = env_->NewWritableFile(fname, &compact->outfile);
  if (s.ok()) {
    compact->builder = new TableBuilder(options_, compact->outfile);
  }
  return s;
}

Status DBImpl::FinishCompactionOutputFile(CompactionState* compact,
                                          Iterator* input) {
  assert(compact != nullptr);
  assert(compact->outfile != nullptr);
  assert(compact->builder != nullptr);

  const uint64_t output_number = compact->current_output()->number;
  assert(output_number != 0);

  // Check for iterator errors
  Status s = input->status();
  const uint64_t current_entries = compact->builder->NumEntries();
  if (s.ok()) {
    s = compact->builder->Finish();
  } else {
    compact->builder->Abandon();
  }
  const uint64_t current_bytes = compact->builder->FileSize();
  compact->current_output()->file_size = current_bytes;
  compact->total_bytes += current_bytes;
  delete compact->builder;
  compact->builder = nullptr;
```

```
  // Finish and check for file errors
  if (s.ok()) {
    s = compact->outfile->Sync();
  }
  if (s.ok()) {

    s = compact->outfile->Close();
  }
  delete compact->outfile;
  compact->outfile = nullptr;

  if (s.ok() && current_entries > 0) {
    // Verify that the table is usable
    Iterator* iter =
        table_cache_->NewIterator(ReadOptions(), output_number, current_bytes);
    s = iter->status();
    delete iter;
    if (s.ok()) {
      Log(options_.info_log, "Generated table #%llu@%d: %lld keys, %lld bytes",
          (unsigned long long)output_number, compact->compaction->level(),
          (unsigned long long)current_entries,
          (unsigned long long)current_bytes);
    }
  }
  return s;
}
```

继续来看 DBImpl::DoCompactionWork：

```
Status DBImpl::InstallCompactionResults(CompactionState* compact) {
```

```
  mutex_.AssertHeld();
  Log(options_.info_log, "Compacted %d@%d + %d@%d files => %lld bytes",
      compact->compaction->num_input_files(0), compact->compaction->level(),
      compact->compaction->num_input_files(1), compact->compaction->level() + 1,
      static_cast<long long>(compact->total_bytes));


  // Add compaction outputs
  compact->compaction->AddInputDeletions(compact->compaction->edit());
  const int level = compact->compaction->level();
  for (size_t i = 0; i < compact->outputs.size(); i++) {
    const CompactionState::Output& out = compact->outputs[i];
    compact->compaction->edit()->AddFile(level + 1, out.number, out.file_size,
                                         out.smallest, out.largest);
  }
  return versions_->LogAndApply(compact->compaction->edit(), &mutex_);
}

Status DBImpl::DoCompactionWork(CompactionState* compact) {
  ...
  if (status.ok() && shutting_down_.load(std::memory_order_acquire)) {
    status = Status::IOError("Deleting DB during compaction");
  }
  if (status.ok() && compact->builder != nullptr) {
    status = FinishCompactionOutputFile(compact, input);
  }
  if (status.ok()) {
    status = input->status();
  }
  delete input;
  input = nullptr;
```

```
  CompactionStats stats;
  stats.micros = env_->NowMicros() - start_micros - imm_micros;
  for (int which = 0; which < 2; which++) {
    for (int i = 0; i < compact->compaction->num_input_files(which); i++) {

      stats.bytes_read += compact->compaction->input(which, i)->file_size;
    }
  }
  for (size_t i = 0; i < compact->outputs.size(); i++) {
    stats.bytes_written += compact->outputs[i].file_size;
  }

  mutex_.Lock();
  stats_[compact->compaction->level() + 1].Add(stats);

  if (status.ok()) {
    status = InstallCompactionResults(compact);
  }
  if (!status.ok()) {
    RecordBackgroundError(status);
  }
  VersionSet::LevelSummaryStorage tmp;
  Log(options_.info_log, "compacted to: %s", versions_->LevelSummary(&tmp));
  return status;
}
```

执行 InstallCompactionResults 时将 Compaction 的文件集合加入到 VersionEdit 的删除列表中，并将新生成的文件加入到新文件列表里，随后执行 versions_->LogAndApply 更新版本。最后再执行一些清理操作，Compaction 过程就结束了。

题外话

看 VersionSet::AddBoundaryInputs 部分的代码时，VS Code 上显示提交于 4 年前，而大部分的 LevelDB 代码提交于 8 年前。这引起了我的警觉：这个 Bug 竟然影响了 4 年。随即用 VS Code 的 Git Blame 插件查看修复该 Bug 对应的 Commit 及对应的 Pull Request，发现了不得了的事情：这个提交是 16 年初的，但 19 年 4 月才合并进去。

该 Bug 最早报告于 2015 年的 Issue 320，当时 richcole 就给出了 Bug 的分析，并在 16 年初提交了该 Bug 的修复，但一直无人理会。直到 19 年 3 月 vonnyfly 发现了这个严重问题，这才引起了官方的重视，之后在大家的协作下终于将修复 patch 合并到主分支。

而基于 LevelDB 开发的 RocksDB 则在该问题上做出了快速响应。16 年 Issue 993 中有人询问 RocksDB 是否同样受到该 Bug 影响，RocksDB 的主要贡献者 igorcanadi 在 12 小时内及时回复，表示 RocksDB 不受该 Bug 影响：

I just read the issue more throughly. RocksDB doesn't have the same bug. Looks like we actually found and fixed that bug years ago. I don't know why we didn't contribute back to LevelDB :(

笔者只是通过 GitHub 的 Issue 和 PR 恢复了该事件的发展过程，不对此作出任何评价。不过这个严重的 Bug 应该仍然影响着很多项目，毕竟小概率触发比 100% 触发更可怕，希望能引起大家的重视，检查下自己使用的 LevelDB 是不是 Release 1.22 之前的版

本。

References

1. "Compaction", *leveldb-handbook*
2. "Fix snapshot compaction bug", *leveldb#339*
3. "Compaction causes data inconsistency when using snapshots", *leveldb#320*
4. "Dose rocksdb have the bug found in leveldb?", *rocksdb#993*