在levelDB中加入TTL的功能

余明阳 10225501456

包亦晟 10215501451

实验背景

TTL（Time-To-Live）是一种用于控制数据存储时间的机制，以便在特定时间后自动删除或过期不再需要的数据，它的主要目的是节省存储空间和提高查询效率。

TTL 的工作原理

1. **设置过期时间**：在插入数据时，可以为每条记录指定一个过期时间，即数据的有效期。
2. **定期检查**：数据库会定期检查存储的数据，删除那些已经过期的记录。具体的检查频率和策略可能因实现而异。
3. **自动清理**：一旦数据过期，数据库会自动将其标记为无效或直接删除，从而释放存储空间。

实验设计

1. TTL储存方法

为了将TTL记录在数据中，我们设计了一个过期时间字段并把它添加到了每一个kv对中。当我们写入数据的时候，TTL会自动添加到value的后面，并在读取的时候检查数据是否过期。

2. 检查数据是否有效

当我们进行数据查询的时候，我们需要确保返回的数据都没有过期也就是都是有效的。所以我们在`DBImpl::Get`方法中添加了检查校验的部分。如果读取到的数据是有效的那么我们直接返回，但是如果读取到的是无效数据，我们读取的时候是会返回一个未找到的状态。

3. 清除过期数据，触发合并策略

我们在返回有效数据的同时还需要去删除那些已经过期的数据，我们需要在合适的时机去删除这些过期数据，所以我们禁用了LevelDB 的自动合并功能。具体来说，我们在对所有的数据进行写入之后我们会手动调用`db->CompactRange(nullptr, nullptr)`来进行合并，也就能起到删除过期数据的操作。这样做的好处是能够高效的清理大批的数据。

实验内容

基本工具类

```
uint64_t DBImpl::GetTS(const std::string* val) {
    uint64_t expire_time;
    memcpy(&expire_time, val->data() + val->size() - sizeof(uint64_t),
sizeof(uint64_t));
    return expire_time;
}
```

功能概述

- **目的**：从存储的字符串中提取并返回过期时间。
- 参数：
  - `val`：存储有原值和过期时间的字符串。

主要步骤

1. **读取过期时间**：使用 `memcpy` 从字符串的末尾读取最后 8 字节（`sizeof(uint64_t)`），这些字节代表过期时间（`expire_time`）。
2. **返回结果**：返回提取到的过期时间。

Put方法

```
Status DB::Put(const WriteOptions& options, const Slice& key, const Slice& value,
uint64_t ttl) {

  std::string val_time;
  //计算过期时间
  uint64_t expire_time = std::chrono::duration_cast<std::chrono::milliseconds>
(std::chrono::system_clock::now().time_since_epoch()).count() + ttl * 1000; // 毫秒
  // 添加value的值
  val_time.append(value.data(), value.size());
  // 添加过期时间
  val_time.append(reinterpret_cast<const char*>(&expire_time), sizeof(expire_time));
  WriteBatch batch;
  batch.Put(key, Slice(val_time));
  return Write(options, &batch);
}
```

功能分析

参数

- `options`：写入选项。
- `key`：要存储的键。
- `value`：关联的值。
- `ttl`：数据的生存时间。

主要步骤

1. **定义字符串**：创建一个空的 `std::string val_time`，用于存储要写入的数据和过期时间。
2. **计算过期时间**：
   - 使用 `std::chrono::system_clock::now()` 获取当前时间。
   - 计算自纪元以来的毫秒数，并加上指定的 TTL。
   - 结果存储在 `expire_time` 中。
3. **构造存储数据**：
   - 将输入的 `value` 添加到 `val_time` 中。
   - 将计算得到的 `expire_time` 以二进制形式附加到 `val_time` 的末尾。
4. **创建写入批处理**：
   - 创建一个 `WriteBatch` 对象。
   - 使用 `Put` 方法将 `key` 和构造的 `val_time` 存入批处理。
5. **执行写入操作**：
   - 调用 `Write` 方法，将批处理写入数据库，并返回写入的状态。

Get操作，检查数据是否过期

```
Status DBImpl::Get(const ReadOptions& options, const Slice& key,
                   std::string* value) {
  Status s;
  MutexLock l(&mutex_);
  SequenceNumber snapshot;
  if (options.snapshot != nullptr) {
    snapshot =
        static_cast<const SnapshotImpl*>(options.snapshot)->sequence_number();
  } else {
    snapshot = versions_->LastSequence();
  }

  MemTable* mem = mem_;
  MemTable* imm = imm_;
  Version* current = versions_->current();
  mem->Ref();
  if (imm != nullptr) imm->Ref();
  current->Ref();

  bool have_stat_update = false;
  Version::GetStats stats;
  // Unlock while reading from files and memtables
  {
    mutex_.Unlock();
    // First look in the memtable, then in the immutable memtable (if any).
    LookupKey lkey(key, snapshot);
    if (mem->Get(lkey, value, &s)) {
      // Done
    } else if (imm != nullptr && imm->Get(lkey, value, &s)) {
      // Done
    } else {
      s = current->Get(options, lkey, value, &stats);
      have_stat_update = true;
    }
    mutex_.Lock();
  }
  if (s.ok()) {
      auto t1 = env_->GetTime();
      auto t2 = GetTS(value);
      if(t1 >= t2){

        s = Status::Expire("Expire",Slice());
      } else {
        *value = value->substr(0, value->size() - sizeof(uint64_t));
      }
    }

  if (have_stat_update && current->UpdateStats(stats)) {
    MaybeScheduleCompaction();
  }
```

```
  mem->Unref();
  if (imm != nullptr) imm->Unref();
  current->Unref();

  return s;
}
```

功能分析

参数

- `options`: 读取选项。
- `key`: 要读取的键。
- `value`: 指向存储结果的字符串，用于返回找到的值。

主要步骤

1. **锁定和快照处理**:
   - 使用互斥锁 `MutexLock` 确保线程安全。
   - 根据传入的 `options` 确定读取快照的版本。如果没有快照，使用最新的版本序列号。
2. **引用计数**:
   - 获取当前的内存表（`mem_`）和不可变内存表（`imm_`），以及当前版本。
   - 对内存表和版本对象进行引用计数，确保在读取过程中不会被销毁。
3. **读取数据**:
   - 在解锁状态下，首先从内存表（`mem`）中查找值。如果未找到，再检查不可变内存表（`imm`），最后从当前版本中查找。
   - 如果在内存表或不可变内存表中找到值，则直接返回。如果在当前版本中找到，则更新统计信息。
4. **检查过期**:
   - 如果找到值，则获取当前时间（`t1`）和存储的过期时间（`t2`）。
   - 如果当前时间大于或等于过期时间，返回一个过期的状态（`Status::Expire`）。
   - 如果未过期，则从 `value` 中去掉过期时间部分，保留原始值。
5. **更新统计信息**:
   - 如果更新了统计信息，检查是否需要调度压缩操作。
6. **释放引用**:
   - 释放对内存表和版本的引用，确保资源得到管理。
7. **返回状态**:
   - 返回读取操作的状态。

合并时清理

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

.......
      Slice value = input->value();
      if (value.size() >= sizeof(uint64_t)) {
        const char* ptr = value.data();
        std::string temp_str(value.data(), value.size());
        uint64_t expire_time = DBImpl::GetTS(&temp_str);
        uint64_t current_time = env_->GetTime();

        if (current_time > expire_time) {
          drop = true;
        }
      }
.......
```

功能概述

1. **获取值**:
   - `Slice value = input->value();`：从输入的压缩状态中获取当前键的值。
2. **检查值的大小**:
   - `if (value.size() >= sizeof(uint64_t))`：确保值的大小至少为 8 字节（`sizeof(uint64_t)`），因为过期时间以 `uint64_t` 存储在值的末尾。
3. **构造临时字符串**:
   - `std::string temp_str(value.data(), value.size());`：创建一个临时字符串 `temp_str`，以便后续提取过期时间。
4. **获取过期时间**:
   - `uint64_t expire_time = DBImpl::GetTS(&temp_str);`：调用 `GetTS` 方法，从 `temp_str` 中提取过期时间。
5. **获取当前时间**:
   - `uint64_t current_time = env_->GetTime();`：获取当前的时间戳。
6. **检查过期状态**:
   - `if (current_time > expire_time)`：比较当前时间与过期时间。
   - 如果当前时间大于过期时间，则将 `drop` 标志设置为 `true`，表示该值需要被丢弃。

实验中的问题

```
/home/byc/design/leveldb/test/ttl_test.cc:67: Failure
Value of: status.ok()
  Actual: false
Expected: true
```

每次测试的时候总是出现上述的问题，我后来发现问题在于下面的三个常量定义的太小了。

这段代码定义了LevelDB中Level-0层文件数量的三个重要阈值常量：

- kL0_CompactionTrigger

这是触发Level-0压缩的文件数阈值。当Level-0的文件数达到它时，LevelDB会启动压缩操作。Level-0比较特殊，因为这一层的文件可能有重叠的key范围，所以需要及时进行压缩以提高读取性能。

- kL0_SlowdownWritesTrigger

这是一个软限制阈值。当Level-0的文件数超过它时，LevelDB会主动降低写入速度。这是一个预防机制，通过降低写入速度来给压缩操作留出更多处理时间，避免系统被大量的未压缩文件压垮。

- kL0_StopWritesTrigger

这是最严格的限制，当Level-0的文件数达到它时，LevelDB会完全停止写入操作。这是一个安全机制，防止系统因为过多的未压缩文件而崩溃。

Level-0是很特殊的，Level-0的文件之间可能有重叠的key范围，而且每个新的写操作都会直接创建新的Level-0文件。当Level-0文件数量达到kL0_SlowdownWritesTrigger（原为4）时，写入速度会降低；达到kL0_StopWritesTrigger（原为12）时，写入会完全停止。

在InsertData函数中，我们快速写入大量数据：

```
 for (int i = 0; i < key_num; i++) {
    db->Put(writeOptions, key, value, ttl);
}
```

调高之后阈值允许了更多的Level-0文件同时存在，写入操作不会被限制或停止，从而使得我们有足够时间完成所有写入，而不是因为触发限制而失败。

```
open db failed
```

因为两个测试用例使用了相同的数据库路径 "testdb"，所以可能存在资源清理或权限问题。我通过 `delete db` 和 `DestroyDB("testdb", Options());` 确保在每个测试用例结束时正确关闭数据库。之后就可以成功了。

实际测试

```
TEST(TestTTL, ReadTTL) {
    DB *db;
    if(OpenDB("testdb", &db).ok() == false) {
        std::cerr << "open db failed" << std::endl;
        abort();
    }

    uint64_t ttl = 20;

    InsertData(db, ttl);

    ReadOptions readOptions;
    Status status;
    int key_num = data_size / value_size;
    srand(0);
    for (int i = 0; i < 100; i++) {
        int key_ = rand() % key_num+1;
        std::string key = std::to_string(key_);
        std::string value;
        status = db->Get(readOptions, key, &value);
        ASSERT_TRUE(status.ok());
    }

    Env::Default()->SleepForMicroseconds(ttl * 1000000);

    for (int i = 0; i < 100; i++) {
        int key_ = rand() % key_num+1;
        std::string key = std::to_string(key_);
        std::string value;
        status = db->Get(readOptions, key, &value);
        ASSERT_FALSE(status.ok());
    }
    delete db;
}

TEST(TestTTL, CompactionTTL) {
    DB *db;
    DestroyDB("testdb", Options());
    if(OpenDB("testdb", &db).ok() == false) {
        std::cerr << "open db failed" << std::endl;
        abort();
    }

    uint64_t ttl = 20;
    InsertData(db, ttl);

    leveldb::Range ranges[1];
    ranges[0] = leveldb::Range("-", "A");
    uint64_t sizes[1];
    db->GetApproximateSizes(ranges, 1, sizes);
```

```
    ASSERT_GT(sizes[0], 0);

    Env::Default()->SleepForMicroseconds(ttl * 1000000);

    db->CompactRange(nullptr, nullptr);

    leveldb::Range ranges1[1];
    ranges[0] = leveldb::Range("-", "A");
    uint64_t sizes1[1];
    db->GetApproximateSizes(ranges1, 1, sizes1);
    ASSERT_EQ(sizes1[0], 0);
    delete db;
}
```

```
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from TestTTL
[ RUN      ] TestTTL.ReadTTL
[       OK ] TestTTL.ReadTTL (20803 ms)
[ RUN      ] TestTTL.CompactionTTL
[       OK ] TestTTL.CompactionTTL (20989 ms)
[----------] 2 tests from TestTTL (41792 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (41792 ms total)
[  PASSED  ] 2 tests.
```