

## 目 录

<b>第一章 需求分析</b>	<b>1</b>
1.1 总体要求	1
1.2 存储要求	1
1.3 添加要求	1
1.4 搜索要求	1
1.5 索引要求	2
1.6 并发要求	2
<b>第二章 总体设计</b>	<b>3</b>
2.1 整个程序的架构	3
2.2 关键流程分析	3
2.2.1 添加数据	3
2.2.2 建立索引	3
2.2.3 查询数据	4
<b>第三章 详细设计与实现</b>	<b>6</b>
3.1 文件 I/O 与跨平台	6
3.1.1 文件 I/O 设计与实现	7
3.1.2 Env 工厂方法具体实现	7
3.2 表数据操作设计与实现	10
3.2.1 添加数据	11
3.2.2 建立索引	12
3.2.3 查询数据	13
3.2.4 多线程安全	14
3.3 其他	15
3.3.1 Slice 类	15
3.3.2 Status 类	16
3.3.3 数据的编码与解码	16
<b>第四章 测试</b>	<b>17</b>
4.1 测试环境	17
4.2 测试内容	17
4.2.1 测试用例 1：单线程	17

4.2.2 测试用例 2：多线程.....	18
4.3 测试结果.....	19

## 第一章 需求分析

### 1.1 总体要求

- 存储一张表，然后能对该表进行查询、添加等操作。上述功能以 API 的形式提供给应用使用。

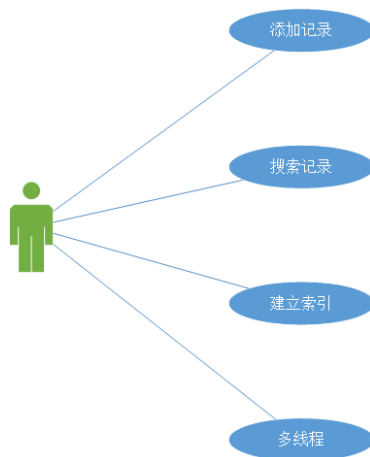


图 1-1 用例图

### 1.2 存储要求

- 存储一张表，然后能对该表进行查询、添加等操作。上述功能以 API 的形式提供给应用使用。
- 该表有 100 个属性，每个属性都是 `int64_t` 类型；
- 需要支持的最大行数为 1 百万行。

### 1.3 添加要求

- 提供 API 函数，实现向表格添加一行的功能（添加到表格的末尾）。

### 1.4 搜索要求

- 提供 API 函数，实现对表格的某一个属性进行范围查找的功能。例如：查找在属性 A 上，大于等于 50，小于等于 100 的所有行
- 用户可以指定在哪一个属性上进行搜索；
- 当搜索结果包含的行数过多时，可以只返回一小部分，如 10 行等。

## 1.5 索引要求

- 提供 API 函数，为表格的某一个属性建立索引结构，以实现快速搜索；
- 自行选择使用哪种数据结构，建立索引结构，比如 B+ 树等；
- 建立的索引结构，需要保存到一个文件中（索引文件）；下次重启应用程序，并执行搜索任务时，应先检查是否已为相应属性建立了索引结构；
- 即，搜索功能实现时，需要查找是否有索引文件存在，若有，则使用该文件加速搜索。

## 1.6 并发要求

- 应用程序可以以多线程的方式，使用我们提供的上述 API；
- 要保证多线程环境下，表、索引结构、索引文件的一致性。

## 第二章 总体设计

### 2.1 整个程序的架构

存储引擎维护三个文件：`table_file`，`index_file`，`manifest_file`。其中，`table_file` 负责用户添加数据的持久化；`index_file` 负责索引结构的持久化；`manifest_file` 负责存储元数据，以便重启程序后进行恢复。

用户在使用该引擎时可以以多线程方式向表中添加数据和查询数据，但建立索引时必须以单线程方式，因为索引结构属于全局结构，即整个表只有一个索引结构。如果以多线程方式建立索引就会存在一个问题：如果线程 A 建立了索引，那么这个索引结构只是针对线程 A 所添加的数据而建立的；如果线程 B 被调度，并且尝试建立索引，那么线程 A 建立的索引就会需要合并上线程 B 添加的数据所对应的索引。当多个线程并发地添加数据并尝试建立索引时，索引结构就会被频繁修改，造成效率的降低。基于上述考虑，该存储引擎在设计时只允许以单线程的方式建立索引，如果多线程并发添加数据，那么必须等到所有线程结束后再给表中的属性建立索引。

另外，索引结构只有一个，也就是说同一时刻只存在表中的某一个属性的索引。当尝试为别的属性建立索引时，旧的索引结构就会被删除。

存储引擎在开发时进行了跨平台处理，支持 Linux 和 Windows 平台。

### 2.2 关键流程分析

#### 2.2.1 添加数据

添加数据的流程图如图2-1所示。因为添加数据要实现多线程安全，所以需要先获得互斥锁，然后判断 `table_file` 文件是否有效、用户输入是否有效等，检查成功后就可以将编码后的数据使用追加写的方式写入 `table_file`。

#### 2.2.2 建立索引

建立索引的流程图如图2-2所示。为标号为 `attr_id` 所对应的属性列建立索引时，如果用户没有显式告知存储引擎，添加数据已经结束，那么程序就主动结束数据的添加，关闭可写的 `table_file` 文件，并重新以只读方式打开 `table_file`。然后从 `table_file` 读入需要建立索引的属性列的全部数据，构造一个 `IndexEntry` 结构的线性表（`IndexEntry` 结构包含两个数据，一个是表中的数据，一个是该数据所在的行号），将其排序后写入 `index_file`。写入成功后重新打开一个只读的 `index_file` 以

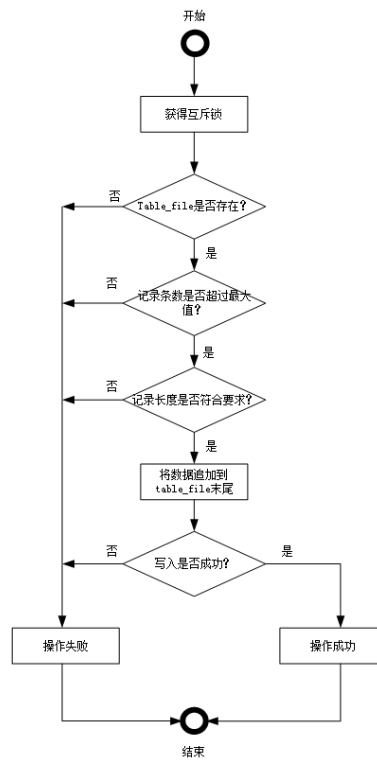


图 2-1 添加数据

便后续的查询操作所使用。

### 2.2.3 查询数据

查询数据时用户需要指定一个属性，以及在该属性上查询的范围。如果索引文件不存在就直接读取表数据文件进行顺序查找，每次读取一行数据（100 个属性），判断给定属性的值是否在用户指定的查询区间内，如果是，就说明将该行数据添加到查询结果中。如果索引文件存在，就可以加速查询。因为索引结构实则是一个 KV 结构的数组，Key 就是相应属性上的值，Value 就是该属性值所属的行号，根据行号可以实现表数据文件 table\_file 的随机读取。查询数据的具体流程如图2-3所示。

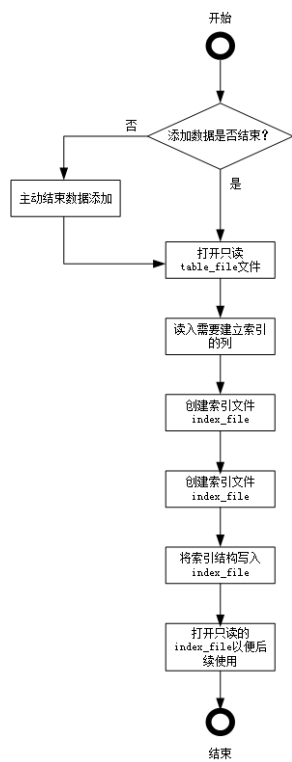


图 2-2 建立索引

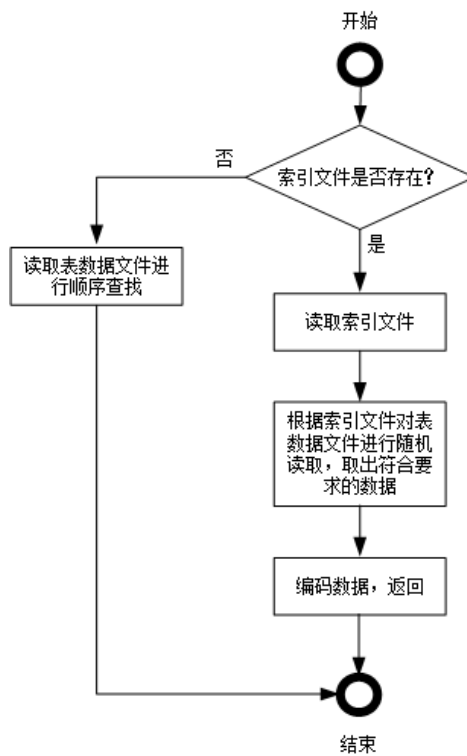


图 2-3 查询数据

## 第三章 详细设计与实现

### 3.1 文件 I/O 与跨平台

存储引擎只支持两种类型的文件：写入方式只能是追加写的可写文件 WritableFile，支持随机读的只读文件 RandomAccessFile。这两个文件抽象类对 Linux 和 Windows 都有具体的实现，跨平台的核心就是底层的文件操作接口。相关类图如图3-1和图3-2所示。



图 3-1 RandomAccessFile 类图

在 PosixRandomAccessFile 中，成员变量 fd\_ 是 int 类型的文件描述符；在 WindowsRandomAccessFile 中，file\_handle\_ 是 HANDLE 类型的文件句柄，二者的实质都是进程的文件指针表的索引（下标）。

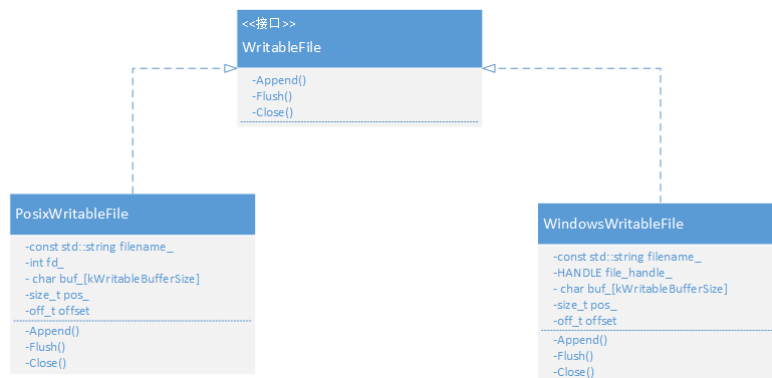


图 3-2 WritableFile 类图

在 WritableFile 的两个子类中，buf\_ 是写缓冲，每次接受到文件写请求时都先将数据写入缓冲区，待缓冲区满后在写入磁盘；pos\_ 表示当前缓冲区内的数据写到哪里了；offset\_ 是文件的磁盘偏移，等价于文件的当前大小，每次 Flush 数据到磁盘时都会更新 offset\_。



抽象基类 `RandomAccessFile/WritableFile` 的子类实例分别由抽象基类 `Env` 的子类 `WindowsEnv/PosixEnv` 实例创建，这里使用的是工厂方法模式。`Env` 及其子类的类图如图3-3所示。



图 3-3 Env 类图

### 3.1.1 文件 I/O 设计与实现

#### 3.1.1.1 文件随机读

Linux 平台上，`PosixRandomAccessFile::Read()` 使用 `pread()` 系统调用根据给定的偏移量随机读；Windows 平台上，`WindowsRandomAccessFile::Read()` 使用 `ReadFile()` 和参数 `OVERLAPPED` 实现随机读。

#### 3.1.1.2 文件追加写

调用 `WritableFile::Append()` 写文件时先将数据写入内存缓冲，当缓冲区满后再 Flush 到磁盘，以此减少磁盘 I/O 次数提高效率。

### 3.1.2 Env 工厂方法具体实现

抽象基类 `Env` 提供了 4 个接口：`GetFileSize()` 获取文件大小；`NewRandomAccessFile()` 创建一个 `RandomAccessFile` 子类实例；`NewWritableFile()` 和 `OpenWritableFile()` 都是创建一个 `WritableFile` 子类实例，是不过 `New*` 会新建一个文件；`Open*` 打开一个已存在的文件，并将文件指针移到文件末尾。

Linux 平台上 `PosixEnv::GetFileSize()` 使用 `stat()` 系统调用，Windows 平台上 `WindowsEnv::GetFileSize()` 先使用 `CreateFile()` 打开文件，获得文件句柄，再使用 `GetFileSize()` 系统调用获得文件大小，最后 `CloseHandle()` 关闭文件句柄。

Linux 平台上 `PosixEnv::NewRandomAccessFile()/NewWritableFile()` 使用 `open()` 系统调用打开文件，Windows 平台上 `WindowsEnv::NewRandomAccessFile()/NewWritableFile()`

使用 `CreateFile()` 系统调用打开文件。这里需要注意，Linux 的 `open()` 打开的文件默认是非阻塞的，即文件打开到关闭期间，可以调用 `open()` 再次打开之；但是 Windows 需要给 `CreateFile()` 指定 `FILE_SHARE_READ` 或 `FILE_SHARE_WRITE` 显式地创建为非阻塞模式，否则在关闭文件前无法再次打开之。

`PosixEnv` 和 `WindowsEnv` 的代码如下：

```

1  class PosixEnv : public Env {
2  public:
3      PosixEnv() = default;
4      PosixEnv(const PosixEnv &) = delete;
5      PosixEnv &operator=(const PosixEnv &)= delete;
6      ~PosixEnv() = default;
7
8      Status GetFileSize(const std::string &filename, size_t *size) override {
9          struct stat buf;
10         if (::stat(filename.c_str(), &buf) >= 0) {
11             *size = static_cast<size_t>(buf.st_size);
12             return Status::OK();
13         }
14         *size = 0;
15         return Status::IOError();
16     }
17
18     Status NewRandomAccessFile(const std::string &filename, RandomAccessFile **result) override {
19         int fd = ::open(filename.c_str(), O_RDONLY);
20         if (fd < 0) {
21             *result = nullptr;
22             return Status::IOError();
23         }
24
25         *result = new PosixRandomAccessFile(filename, fd);
26         return Status::OK();
27     }
28
29     Status NewWritableFile(const std::string &filename, WritableFile **result) override {
30         int fd = ::open(filename.c_str(), O_RDWR | O_CREAT | O_TRUNC, 0664);
31         if (fd < 0) {
32             *result = nullptr;
33             return Status::IOError();
34         }
35
36         *result = new PosixWritableFile(filename, fd);
37         return Status::OK();
38     }
39
40     Status OpenWritableFile(const std::string &filename, WritableFile **result) override {
41         int fd = ::open(filename.c_str(), O_RDWR | O_CREAT | O_APPEND, 0664);
42         if (fd < 0) {
43             *result = nullptr;
44             return Status::IOError();
45         }
46
47         *result = new PosixWritableFile(filename, fd);
48         return Status::OK();
49     }
50 };
51
52 class WindowsEnv : public Env {
53 public:
54     WindowsEnv() = default;
55     WindowsEnv(const WindowsEnv &) = delete;
56     WindowsEnv &operator=(const WindowsEnv &)= delete;
57     ~WindowsEnv() = default;
58
59     Status GetFileSize(const std::string &filename, size_t *size) override {

```

```

60         HANDLE file_handle = ::CreateFileA(filename.c_str(),
61             GENERIC_READ,
62             FILE_SHARE_READ | FILE_SHARE_WRITE,
63             nullptr,
64             OPEN_EXISTING,
65             FILE_ATTRIBUTE_NORMAL,
66             0);
67         if (file_handle == INVALID_HANDLE_VALUE) {
68             *size = 0;
69             return Status::IOError();
70         }
71         *size = ::GetFileSize(file_handle, nullptr);
72         ::CloseHandle(file_handle);
73     return Status::OK();
74 }
75
76 Status NewRandomAccessFile(const std::string &filename, RandomAccessFile **result) override {
77     HANDLE file_handle = ::CreateFileA(filename.c_str(),
78         GENERIC_READ,
79         FILE_SHARE_READ | FILE_SHARE_WRITE,
80         nullptr,
81         OPEN_EXISTING,
82         FILE_ATTRIBUTE_NORMAL,
83         0);
84     if (file_handle == INVALID_HANDLE_VALUE) {
85         *result = nullptr;
86         return Status::IOError();
87     }
88
89     *result = new WindowsRandomAccessFile(filename, file_handle);
90     return Status::OK();
91 }
92
93 Status NewWritableFile(const std::string &filename, WritableFile **result) override {
94     HANDLE file_handle = ::CreateFileA(filename.c_str(),
95         GENERIC_READ | GENERIC_WRITE,
96         FILE_SHARE_READ | FILE_SHARE_WRITE,
97         nullptr,
98         CREATE_ALWAYS,
99         FILE_ATTRIBUTE_NORMAL | FILE_FLAG_WRITE_THROUGH,
100        0);
101     if (file_handle == INVALID_HANDLE_VALUE) {
102         *result = nullptr;
103         return Status::IOError();
104     }
105
106     *result = new WindowsWritableFile(filename, file_handle);
107     return Status::OK();
108 }
109
110 Status OpenWritableFile(const std::string &filename, WritableFile **result) override {
111     HANDLE file_handle = ::CreateFileA(filename.c_str(),
112         GENERIC_READ | GENERIC_WRITE,
113         FILE_SHARE_READ | FILE_SHARE_WRITE,
114         nullptr,
115         OPEN_ALWAYS,
116         FILE_ATTRIBUTE_NORMAL | FILE_FLAG_WRITE_THROUGH,
117         0);
118     if (file_handle == INVALID_HANDLE_VALUE) {
119         *result = nullptr;
120         return Status::IOError();
121     }
122     if (GetLastError() == ERROR_ALREADY_EXISTS) {
123         if (INVALID_SET_FILE_POINTER == ::SetFilePointer(file_handle, 0, nullptr, FILE_END)) {
124             *result = nullptr;
125             return Status::IOError();
126         }
127     }
128 }

```

```

129     *result = new WindowsWritableFile(filename, file_handle);
130     return Status::OK();
131 }
132 };

```

## 3.2 表数据操作设计与实现

表数据的具体操作包括添加数据、建立索引和范围查询，以上操作需要实现多线程安全。Table 类对此进行了封装，类图如图3-4所示。



图 3-4 Table 类图

- 成员变量：

- Env \*env\_ 工厂类，根据不同的平台创建不同子类类型的 RandomAccessFile/WritableFile 子类实例；
- WritableFile \*table\_file\_, \*index\_file\_ 可写的表数据文件和索引文件，在向表中添加数据和建立索引结构时使用之；
- RandomAccessFile \*\*table\_file\_readonly\_, \*index\_file\_readonly\_ 只读的表数据文件和索引文件，在查询数据时使用之；
- std::string table\_file\_name\_, index\_file\_name\_, manifest\_file\_name 表数据文件、索引文件和存储元信息的 MANIFEST 文件名；
- int nr\_entries\_ 表数据文件中存储了多少条数据；
- bool appending\_finished 向表中添加数据的操作是否已经结束；
- int index\_attr\_id\_ 当前索引结构/索引文件是为哪个属性列建立的；
- Mutex mutex\_ 互斥量，向表中添加数据时需要先获得互斥锁。Mutex 类只是对 std::mutex 的轻量级封装，需要配合 MutexLock 类进行使用。

- 成员函数:

- Status Append(std::vector<uint64\_t> &data) 向表中添加数据, data 的长度要求等于 Table::kNumTableAttributes(100);
- Status BuildIndexBlock(int attr\_id) 为属性列 attr\_id 建立索引结构, 并将其保存到文件中;
- Status Lookup(int attr\_id, uint64\_t lower\_bound, uint64\_t upper\_bound, std::vector<std::vector<uint64\_t>> \*results) 在属性列 attr\_id 上查找 [lower\_bound, upper\_bound] 范围内的数据, 将符合要求的行返回到 results;
- Status Finish() 结束向表中添加数据, 并创建 MANIFEST 文件, 向其中写入 nr\_entries\_ 和 index\_attr\_id\_;
- int FindIndexEntryLessOrEqual(std::vector<IndexEntry> &index\_entries, uint64\_t x) const 使用二分查找方式在 index\_entries 中查找 Key 刚好小于或等于 x 的 IndexEntry;
- int FindIndexEntryGreaterOrEqual(std::vector<IndexEntry> &index\_entries, uint64\_t x) const 使用二分查找方式在 index\_entries 中查找 Key 刚好大于或等于 x 的 IndexEntry;
- Status Recover() 每次重启程序时: 如果表数据文件存在就打开之, 用于后续的写入; 读取 MENIFEST 文件, 恢复 nr\_entries\_ 和 index\_attr\_id\_;
- std::vector<uint64\_t> Decode(const Slice &slice) const 将 Slice 对象指向的字符数组解码成 uint64\_t 数组。

### 3.2.1 添加数据

Table::Append() 负责将数据添加到表的末尾。如果表数据文件无效, 或者记录的条数已达最大值, 或者用户输入的数据长度不合法, 就直接返回相应的错误信息。如果没有任何异常, 就可以进行数据的添加操作。首先需要获得互斥锁, 然后调用 PutFixed64() 将用户输入的 100 个 uint64\_t 数据编码成字符数组的形式 (字符数组使用 std::string 来实现), 再调用 WritableFile 实例 table\_file\_ 的 Append() 方法完成文件写入, 最后计数器 nr\_entries\_ 递增。相应代码如下:

```

1 Status Table::Append(std::vector<uint64_t> &data) {
2     MutexLock l(&mutex_);
3     if (table_file_ == nullptr) {
4         return Status::GeneralError("Table::table_file_ has been closed.");
5     }
6     if (nr_entries_ >= kMaxTableEntries) {
7         return Status::GeneralError("too much data");
8     }
9     assert(data.size() == kNumTableAttributes);
10    std::string d;
11    for (int i = 0; i < kNumTableAttributes; i++) {

```

```

12         PutFixed64(&d, data[i]);
13     }
14
15     Status status = table_file->Append(d);
16     if (!status.ok()) {
17         return status;
18     }
19     nr_entries++;
20     return Status::OK();
21 }

```

### 3.2.2 建立索引

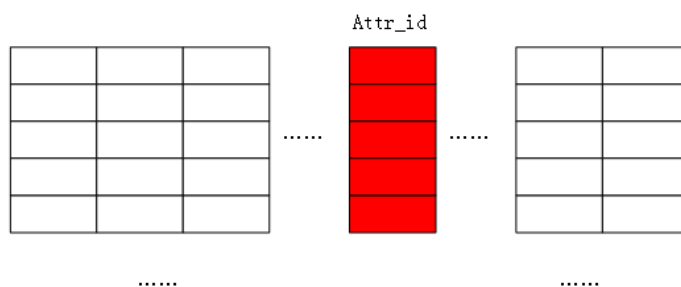


图 3-5 Demo: 为 Attr\_id 属性列建立索引

Table::BuildIndexBlock() 负责建立索引结构并将其保存到文件。首先读入表数据文件的某一属性列 attr\_id (如图3-5所示), 将其作为 Key, 并将所属行号作为 Value, 封装成一个 IndexEntry 的 KV 结构 index\_entries。然后将根据 Key 排序后的 index\_entries 编码成字符数组的形式写入索引文件 index\_file。相应代码如下:

```

1 Status Table::BuildIndexBlock(int attr_id) {
2     Status status;
3     if (!appending_finished_) { /* 添加数据结束后才建立索引 */
4         Finish();
5     }
6     assert(attr_id >= 0 && attr_id < kNumTableAttributes);
7     index_attr_id_ = attr_id;
8
9     if (table_file_readonly_ == nullptr) {
10         status = env_->NewRandomAccessFile(table_file_name_, &table_file_readonly_);
11         if (!status.ok()) {
12             return status;
13         }
14     }
15
16     /* 将attr_id所对应的列读入index_entries */
17     Slice result;
18     char scratch[sizeof(uint64_t)];
19     uint64_t offset = attr_id * sizeof(uint64_t);
20     std::vector<IndexEntry> index_entries;
21     for (int i = 0; i < nr_entries_; i++) {
22         status = table_file_readonly_->Read(offset, sizeof(uint64_t), scratch, &result);
23         if (!status.ok()) {
24             return status;
25         }
26         index_entries.push_back(std::make_pair(
27             DecodeFixed64(result.data()), i));
28         offset += kNumTableAttributes * sizeof(uint64_t);
29     }

```

```

30
31     if (index_file_ == nullptr) {
32         status = env_->NewWritableFile(index_file_name_, &index_file_);
33         if (!status.ok()) {
34             return status;
35         }
36     }
37
38     /* index_entries 排序后写入索引文件 index_file_ */
39     std::sort(index_entries.begin(), index_entries.end());
40     std::string d;
41     for (size_t i = 0; i < index_entries.size(); i++) {
42         PutFixed64(&d, index_entries[i].first);
43         PutFixed32(&d, index_entries[i].second);
44     }
45
46     status = index_file_->Append(d);
47     if (status.ok()) {
48         status = index_file_->Close();
49         if (status.ok()) {
50             /* 打开只读的 index_file_ 以便后续使用 */
51             if (index_file_readonly_ == nullptr) {
52                 status = env_->NewRandomAccessFile(index_file_name_, &index_file_readonly_);
53             }
54         }
55     }
56     return status;
57 }

```

### 3.2.3 查询数据

Table::Lookup() 查询数据时首先检查相应属性对因的索引文件是否存在，如果不存在就直接读取表数据文件进行逐行查找，效率较低。如果索引文件存在，可以在索引结构上进行二分查找，确定符合要求的数据的行号，然后就可以对表数据文件进行随机读，将读入的数据编码后返回给用户。相应代码如下：

```

1  Status Table::Lookup(int attr_id, uint64_t lower_bound, uint64_t upper_bound,
2      std::vector<std::vector<uint64_t>> *results) {
3      Status status;
4      Slice slice;
5      char scratch[kNumTableAttributes * sizeof(uint64_t)];
6
7      if (table_file_ != nullptr) {
8          assert(!appending_finished_);
9          table_file_->Flush();
10     }
11
12     if (index_attr_id != attr_id ||
13         index_file_readonly_ == nullptr) { /* 索引不存在则读取 table_file_ 进行顺序查找 */
14         if (table_file_readonly_ == nullptr) {
15             status = env_->NewRandomAccessFile(table_file_name_, &table_file_readonly_);
16             if (!status.ok()) {
17                 return status;
18             }
19         }
20         uint64_t offset = 0;
21         for (int i = 0; i < nr_entries_; i++) {
22             status = table_file_readonly_->Read(offset, sizeof(scratch), scratch, &slice);
23             if (!status.ok()) {
24                 return status;
25             }
26             std::vector<uint64_t> r = Decode(slice);
27             if (r[attr_id] >= lower_bound && r[attr_id] <= upper_bound) {

```

```

28         results->push_back(r);
29     }
30     offset += kNumTableAttributes * sizeof(uint64_t);
31 }
32 return status;
33 }
34
35 assert(table_file_readonly_ != nullptr);
36 assert(index_file_readonly_ != nullptr);
37
38 std::vector<IndexEntry> index_entries;
39 uint64_t offset = 0;
40 for (int i = 0; i < nr_entries_; i++) {
41     status = index_file_readonly_->Read(offset, sizeof(uint64_t), scratch, &slice);
42     if (!status.ok()) {
43         return status;
44     }
45     uint64_t var = DecodeFixed64(slice.data());
46     status = index_file_readonly_->Read(offset + sizeof(uint64_t), sizeof(uint32_t), scratch, &slice);
47     if (!status.ok()) {
48         return status;
49     }
50     uint32_t index = DecodeFixed32(slice.data());
51     index_entries.push_back(std::make_pair(var, index));
52     offset += sizeof(uint64_t) + sizeof(uint32_t);
53 }
54
55 int lower_bound_idx = FindIndexEntryGreaterOrEqual(index_entries, lower_bound);
56 int upper_bound_idx = FindIndexEntryLessOrEqual(index_entries, upper_bound);
57 if (lower_bound_idx == -1 || upper_bound_idx == -1) {
58     return Status::NotFound();
59 }
60
61 for (int i = lower_bound_idx; i <= upper_bound_idx; i++) {
62     const size_t nbytes_per_record = kNumTableAttributes * sizeof(uint64_t);
63     status = table_file_readonly_->Read(index_entries[i].second * nbytes_per_record,
64     nbytes_per_record, scratch, &slice);
65     if (!status.ok()) {
66         return status;
67     }
68     std::vector<uint64_t> r = Decode(slice);
69     results->push_back(r);
70 }
71 return status;
72 }

```

### 3.2.4 多线程安全

多线程安全主要由两个类实现：Mutex 和 MutexLock，类图如图3-6所示。



图 3-6 MutexLock 类图

Mutex 只是对 std::mutex 的轻量级封装，需要配合 MutexLock 使用。创建 MutexLock 对象是，构造函数即完成 Mutex 的 Lock() 操作；MutexLock 对象析构时完成 Mutex 的 Unlock() 操作。Mutex/MutexLock 的设计参考了 LevelDB，相应代



码如下：

```

1  class Mutex {
2  public:
3      Mutex() = default;
4      ~Mutex() = default;
5
6      Mutex(const Mutex&) = delete;
7      Mutex &operator=(const Mutex&) = delete;
8
9      void Lock() { mu_.lock(); }
10     void Unlock() { mu_.unlock(); }
11 private:
12     std::mutex mu_;
13 };
14
15 class MutexLock {
16 public:
17     MutexLock(Mutex *mu):mu_(mu) { mu_->Lock(); }
18     ~MutexLock() { mu_->Unlock(); }
19
20     MutexLock(const MutexLock&) = delete;
21     MutexLock &operator=(const MutexLock&) = delete;
22 private:
23     Mutex *mu_;
24 };

```

## 3.3 其他

### 3.3.1 Slice 类

LevelDB 中的数据都是用 Slice 类封装，因为只是保存了指针，不存在数据的深拷贝，所以在传递参数时开销非常小。类图如图3-7所示。

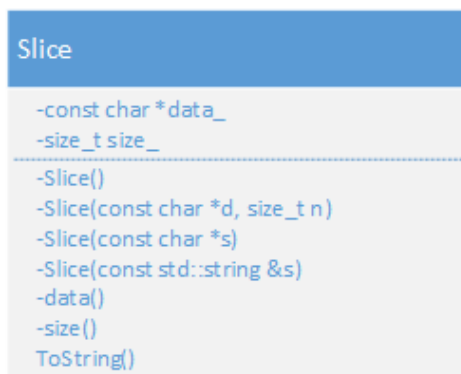


图 3-7 Slice 类图

Slice 类提供了 4 个构造函数，用户可以创建一个不含任何数据的 Slice 对象，或者使用给定长度的字符数组、字符串或 std::string 对象创建 Slice 对象。指针 data\_ 指向具体的数据。

### 3.3.2 Status 类

程序中几乎所有函数的返回值都是 Status 对象，表示操作是否成功，以及错误类型。类图如图3-8所示。

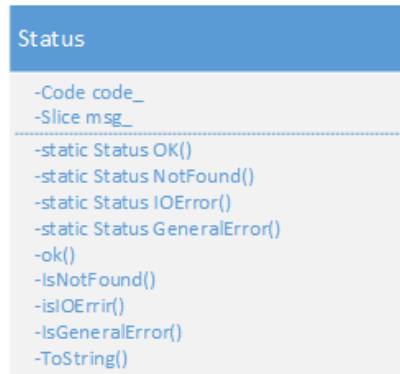


图 3-8 Status 类图

Status 定义了四种操作状态：成功、未找到（主要用于 Table 的数据查询操作）、I/O 错误（文件操作失败）和常规错误（对于常规错误，需要用户自行制定错误消息）。Status 类提供了 4 个 static 方法用于创建上述 4 中操作状态对应的 Status 对象，并提供了对应的非 static 方法用于判断当前 Status 对象的状态。同样地，Status 类的设计也参考了 LevelDB。

### 3.3.3 数据的编码与解码

在进行数据的存取——写入到文件中、从文件中读取——时，需要进行必要的编码与解码。因此，我参考 LevelDB 设计以一组 32 位和 64 位整数和字符数组之间的编解码函数，如下所示：

- void EncodeFixed64(char \*dst, uint64\_t value) 将 64 位无符号整数编码成字符数组，保存到 dst 指向的内存
- void EncodeFixed32(char \*dst, uint32\_t value) 将 32 位无符号整数编码成字符数组，保存到 dst 指向的内存
- void PutFixed64(std::string \*dst, uint64\_t value) 将 64 位无符号整数编码成字符数组，附加到 dst 的末尾 (通过 dst->append() 实现，下同)
- void PutFixed32(std::string \*dst, uint32\_t value) 将 32 位无符号整数编码成字符数组，附加到 dst 的末尾
- uint64\_t DecodeFixed64(const char \*ptr) 从 ptr 解码出一个 uint64\_t 类型的数据
- uint32\_t DecodeFixed32(const char \*ptr) 从 ptr 解码出一个 uint32\_t 类型的数据

## 第四章 测试

### 4.1 测试环境

- 操作系统：Ubuntu 16.04 64 位
- 硬件：内存 4GB
- 测试框架：GooglgTest

### 4.2 测试内容

#### 4.2.1 测试用例 1：单线程

每次添加 1000 条记录，每条记录中的 100 个属性值随机生成；在第 1 个属性上查询，查询范围是 [1000,100000000]。每添加一个数据就执行一次查询，此时的查询操作需要直接读取表数据文件进行，因为索引文件尚未建立。等到所有数据添加结束后，建立索引结构，创建索引文件再次查询，并将查询结果打印出来（最多打印 10 条记录）

测试需要多次进行，目的是测试程序是否能在重启后恢复之前的状态，继续接受添加/查询/建立索引的操作。测试代码如下：

```

1  TEST(table_storage, single_thread1) {
2      Table table("table1", "index1", "MANIFEST1");
3      TestHelper testhlp;
4      Random rnd;
5      Status status;
6      const int n = 1000;
7      const uint64_t lower_bound = 1000;
8      const uint64_t upper_bound = 100000000;
9      int query_attr_id = 0;
10     size_t nr_expected_query_results = 0;
11     std::vector<std::vector<uint64_t>> query_results;
12     clock_t start, end;
13
14     /* 从data文件读取上次的查询结果数，如果data文件存在的话 */
15     testhlp.LoadLastQueryResultsFromFile("data1", reinterpret_cast<int*>(&nr_expected_query_results));
16
17     /* 向表中添加数据 */
18     start = clock();
19     std::cout << "Appending data...\n";
20     for (int i = 0; i < n; i++) {
21         /* 添加记录 */
22         std::vector<uint64_t> nums = rnd.GenerateRandomNumbers(Table::kNumTableAttributes);
23         status = table.Append(nums);
24         ASSERT_TRUE(status.ok());
25
26         /* 添加的记录是否应该在后续的查询结果中出现？ */
27         if (nums[query_attr_id] >= lower_bound && nums[query_attr_id] <= upper_bound) {
28             nr_expected_query_results++;
29         }
30
31         /* 在query_attr_id指定的属性上进行范围查找。由于索引未建立，查询时需要读取table_file */
32         query_results.clear();

```

```

33     status = table.Lookup(query_attr_id, lower_bound, upper_bound, &query_results);
34     if (!status.ok()) {
35         ASSERT_TRUE(status.IsNotFound());
36     }
37
38     /* 校验查询结果 */
39     ASSERT_EQ(query_results.size(), nr_expected_query_results);
40     for (size_t i = 0; i < query_results.size(); i++) {
41         ASSERT_EQ(query_results[i].size(), Table::kNumTableAttributes);
42         ASSERT_TRUE(query_results[i][query_attr_id] >= lower_bound &&
43             query_results[i][query_attr_id] <= upper_bound);
44     }
45     end = clock();
46     printf("Done. Time elapsed: %.5fs\n", (double) (end - start) / CLOCKS_PER_SEC);
47
48     /* 将新的查询结果数写入data文件 */
49     status = testhlp.SaveLastQueryResultsToFile("data1", static_cast<int>(nr_expected_query_results));
50     ASSERT_TRUE(status.ok());
51
52     /* 建立索引 */
53     status = table.Finish();
54     ASSERT_TRUE(status.ok());
55     status = table.BuildIndexBlock(query_attr_id);
56     ASSERT_TRUE(status.ok());
57
58     /* 在query_attr_id指定的属性上进行范围查找。索引已建立，查询时使用索引加速查找 */
59     query_results.clear();
60     status = table.Lookup(query_attr_id, lower_bound, upper_bound, &query_results);
61     if (!status.ok()) {
62         ASSERT_TRUE(status.IsNotFound());
63     }
64
65     /* 校验查询结果 */
66     ASSERT_EQ(query_results.size(), nr_expected_query_results);
67     for (size_t i = 0; i < query_results.size(); i++) {
68         ASSERT_EQ(query_results[i].size(), Table::kNumTableAttributes);
69         ASSERT_TRUE(query_results[i][query_attr_id] >= lower_bound &&
70             query_results[i][query_attr_id] <= upper_bound);
71     }
72
73     /* 打印查询结果。最多打印10条记录 */
74     ...
75 }

```

## 4.2.2 测试用例 2：多线程

创建 8 个线程并发地向表中添加数据，每个线程添加  $1000/8=125$  条记录，待线程结束后执行查询（属性列和查询范围和单线程的测试相同）。

和单线程测试一样，测试需要多次进行，目的是测试程序是否能在重启后恢复之前的状态，继续接受添加/查询/建立索引的操作。测试代码如下：

```

1 TEST(table_storage, multi_thread) {
2     TestHelper testhlp;
3     Status status;
4     const int nr_thds = 8;
5     const int n = 1000 / nr_thds;
6
7     /* 从data文件读取上次的查询结果数，如果data文件存在的话 */
8     testhlp.LoadLastQueryResultsFromFile("data0", reinterpret_cast<int*>(&nr_expected_query_results));
9
10    /* 创建多个线程 */
11    std::thread **thds = new std::thread[nr_thds];
12    for (int i = 0; i < nr_thds; i++) {

```

```

13         thds[i] = new std::thread(thd_routine, n);
14     }
15     for (int i = 0; i < nr_thds; i++) {
16         thds[i] -> join();
17     }
18     for (int i = 0; i < nr_thds; i++) {
19         delete thds[i];
20     }
21     delete[] thds;
22
23     /* 将新的查询结果数写入data文件 */
24     status = testhlp.SaveLastQueryResultsToFile("data0", static_cast<int>(g_nr_expected_query_results));
25     ASSERT_TRUE(status.ok());
26
27     /* 在query_attr_id指定的属性上进行范围查找。由于索引未建立，查询时需要读取table_file */
28     std::vector<std::vector<uint64_t>>> query_results;
29     status = g_table.Lookup(g_query_attr_id, g_lower_bound, g_upper_bound, &query_results);
30     if (!status.ok()) {
31         ASSERT_TRUE(status.IsNotFound());
32     }
33
34     /* 校验查询结果 */
35     ASSERT_EQ(query_results.size(), g_nr_expected_query_results);
36     for (size_t i = 0; i < query_results.size(); i++) {
37         ASSERT_EQ(query_results[i].size(), Table::kNumTableAttributes);
38         ASSERT_TRUE(query_results[i][g_query_attr_id] >= g_lower_bound &&
39             query_results[i][g_query_attr_id] <= g_upper_bound);
40     }
41
42     /* 建立索引后在次测试查询 */
43     g_table.BuildIndexBlock(g_query_attr_id);
44     query_results.clear();
45
46     status = g_table.Lookup(g_query_attr_id, g_lower_bound, g_upper_bound, &query_results);
47     if (!status.ok()) {
48         ASSERT_TRUE(status.IsNotFound());
49     }
50
51     /* 校验查询结果 */
52     ASSERT_EQ(query_results.size(), g_nr_expected_query_results);
53     for (size_t i = 0; i < query_results.size(); i++) {
54         ASSERT_EQ(query_results[i].size(), Table::kNumTableAttributes);
55         ASSERT_TRUE(query_results[i][g_query_attr_id] >= g_lower_bound &&
56             query_results[i][g_query_attr_id] <= g_upper_bound);
57     }
58
59     /* 打印查询结果 */
60     ...
61 }

```

### 4.3 测试结果

上述两个测试用例的测试结果如下：

如图4-1所示，单线程和多线程两个测试用例均测试通过。创建的表数据文件、索引文件以及存储元数据的 MANIFEST 文件如图4-2所示。

```

zhuxiaoxiang@ubuntu:~/github/linux_homework/code/build$ ./table_storage
[=====] Running 2 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 2 tests from table_storage
[ RUN      ] table_storage.multi_thread
47 query query_results with range [0x3E8, 0x5F5E100] on attr 0:
[0] ..... 0000000000060E1B .....
[1] ..... 0000000000216949 .....
[2] ..... 000000000042CEC3 .....
[3] ..... 00000000004D72A9 .....
[4] ..... 0000000000520804 .....
[5] ..... 0000000000560023 .....
[6] ..... 00000000006C7323 .....
[7] ..... 0000000000757011 .....
[8] ..... 0000000000AED22A .....
[9] ..... 0000000000C9B115 .....
[ OK      ] table_storage.multi_thread (33 ms)
[ RUN      ] table_storage.single_thread1
Appending data...
Done. Time elapsed: 2.06173s
37 query query_results with range [0x3E8, 0x5F5E100] on attr 0:
[0] ..... 000000000042CEC3 .....
[1] ..... 0000000000520804 .....
[2] ..... 0000000000757011 .....
[3] ..... 000000000088BC6C .....
[4] ..... 00000000008E7A7F .....
[5] ..... 0000000000C9B115 .....
[6] ..... 0000000000DBF4EE .....
[7] ..... 00000000012965CC .....
[8] ..... 00000000012FB4E6 .....
[9] ..... 00000000019D9FB7 .....
[ OK      ] table_storage.single_thread1 (2100 ms)
[-----] 2 tests from table_storage (2134 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (2134 ms total)
[ PASSED ] 2 tests.

```

图 4-1 测试结果

```

zhuxiaoxiang@ubuntu:~/github/linux_homework/code/build$ ls -l
total 4180
-rw-rw-r-- 1 zhuxiaoxiang zhuxiaoxiang 12701 Oct 21 07:13 CMakeCache.txt
drwxrwxr-x 5 zhuxiaoxiang zhuxiaoxiang 4096 Oct 26 02:03 CMakeFiles
-rw-rw-r-- 1 zhuxiaoxiang zhuxiaoxiang 1407 Oct 21 07:13 cmake_install.cmake
-rw-rw-r-- 1 zhuxiaoxiang zhuxiaoxiang 4 Oct 26 02:04 data0
-rw-rw-r-- 1 zhuxiaoxiang zhuxiaoxiang 4 Oct 26 02:04 data1
-rw-rw-r-- 1 zhuxiaoxiang zhuxiaoxiang 24000 Oct 26 02:04 index0
-rw-rw-r-- 1 zhuxiaoxiang zhuxiaoxiang 24000 Oct 26 02:04 index1
-rw-rw-r-- 1 zhuxiaoxiang zhuxiaoxiang 6954 Oct 26 01:59 Makefile
-rw-rw-r-- 1 zhuxiaoxiang zhuxiaoxiang 8 Oct 26 02:04 MANIFEST0
-rw-rw-r-- 1 zhuxiaoxiang zhuxiaoxiang 8 Oct 26 02:04 MANIFEST1
-rw-rw-r-- 1 zhuxiaoxiang zhuxiaoxiang 1600000 Oct 26 02:04 table0
-rw-rw-r-- 1 zhuxiaoxiang zhuxiaoxiang 1600000 Oct 26 02:04 table1
-rwxrwxr-x 1 zhuxiaoxiang zhuxiaoxiang 976144 Oct 26 02:03 table_storage
zhuxiaoxiang@ubuntu:~/github/linux_homework/code/build$

```

图 4-2 输出文件