目 录

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。

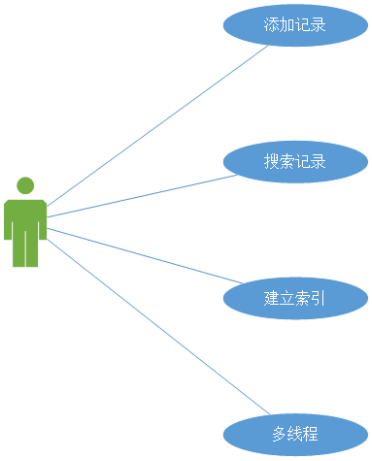

图 1-1 用例图

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

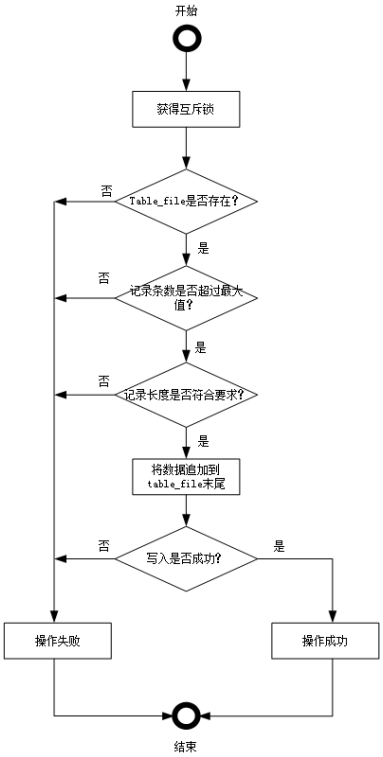

图 2-1 添加数据

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。

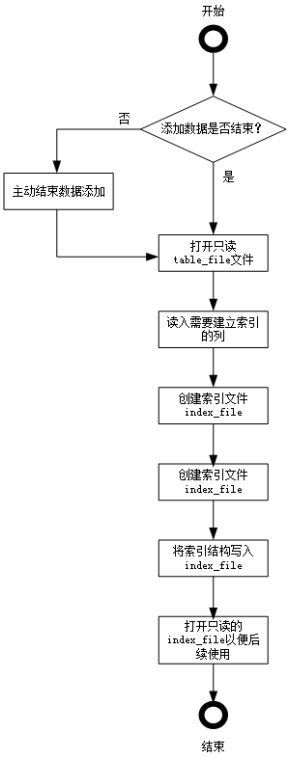

图 2-2 建立索引

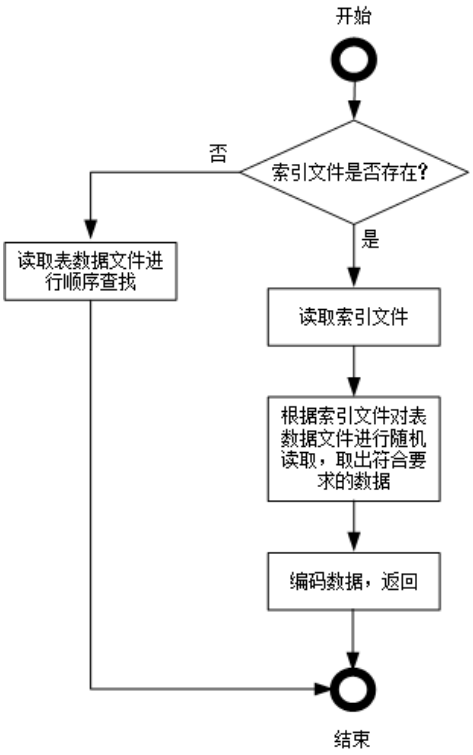

图 2-3 查询数据

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

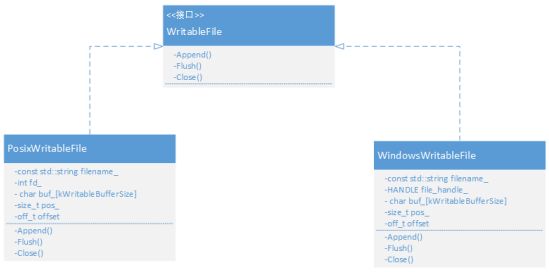

图 3-2 WritableFile 类图

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，使用工厂方法模式。Env 及其子类的类图如图所示。

图 3-3 Env 类图

3.1.1 文件 I/O 具体实现

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。

第四章 测试