Record Manager 设计报告

计算机科学与技术学院 计科1405 於航 3140102234

1. 模块概述

MiniSQL中Record Manager主要针对API的一些请求对Buffer中的Record进行管理，同时返回API所预设的Data类型。

Record Manager主要调用Buffer Manager的函数，通过Table类参数对Buffer中指定Block的指定区间进行修改。本模块实现了条件比较，对于int、float、char这3中变量类型的6中比较条件进行定义，并实现无条件和有条件的查找、插入、删除。

1. 主要功能
   1. 条件比较（cmp）

对于指定的一行记录和给定的条件，返回该记录是否满足条件。

* 1. 查找记录（select）

对于指定的Table以及给定的条件，返回符合条件的所有记录集合的Data类。

* 1. 插入记录（insertRecord）

将一行记录插入目标Table。

* 1. 删除记录（deleteRecords）

对于指定的Table以及给定的条件，将BufferBlock中相应区段置为空，同时返回被删除的记录条数。这些记录已被删除，但从buffer和后面磁盘写回的角度来看，Table所占空间不会变化。

* 1. 删除表（dropTable）

删除指定Table在磁盘、以及Buffer中的所有数据。

* 1. 新建表（createTable）

仅新建磁盘文件。

1. 对外接口

无条件操作和有条件操作进行了重定义，这里主要介绍有条件操作。

* 1. 查找记录

Data select(const Table table, vector<Condition> conditions)

* 1. 插入记录

void insertRecord(Table & table, Row & splitRow)

* 1. 删除记录

int deleteRecords(Table table, vector<Condition> conditions)

* 1. 删除表

void dropTable(Table table)

* 1. 新建表

void createTable(Table table)

1. 设计思路

Record Manager主要实现的是给出条件，在Buffer中管理Record。因此需要实现记录的条件判断，同时一个设计难点在于要通过表信息，获取块号，然后在Block的相应区间内获取一条条Record的数据，最后再将一行（Row）Record根据表上的属性（Attributes）分成小段的string，便于条件判断。

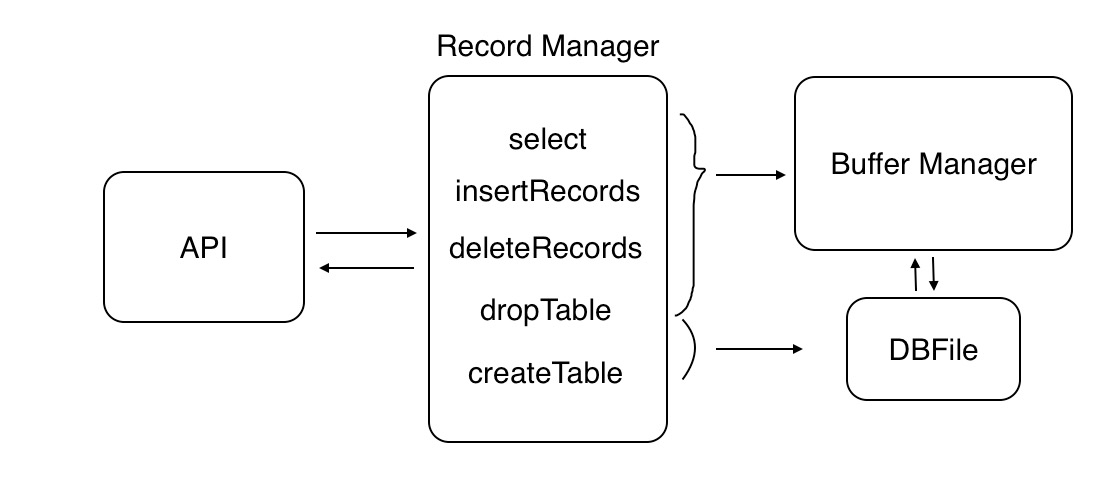
为了优化各个操作的效率，对于每一条记录增加一位的校验位，放在记录的最前面，设置为NIL或者NOTNIL代表这段区间是否存在有效记录。

其他的记录管理函数，可调用条件判断，将记录压入指定的Data类。

无条件删除记录（deleteRecords）与删除表（dropTable）的区别在于，前者是lazy delete，只会把所有记录的首位置为NIL。后者是删除所有文件，包括磁盘文件和buffer中相应block。前者的一种优化思路是将block块数缩减为1，但实际上还要配合文件内存的删除。

1. 整体架构

设计类RecordManager类，API等模块可以对其类函数进行调用。



1. 关键代码
   1. RecordManager的查找记录函数

|  |
| --- |
| Data select(const Table table, vector<Condition> conditions) {  string fileName = table.name + ".table";  string stringRow;  Row splitRow;  Data datas;  const int memsize = table.recordLength + 1;  const int recordNum = BLOCKSIZE / memsize;    for (int blockOffset = 0; blockOffset < table.blockNum; blockOffset++) {  int blockNum = buf.getBlockNum(fileName, blockOffset);  for (int recordOffset = 0; recordOffset < recordNum; recordOffset++) {  int position = recordOffset \* memsize;  stringRow = buf.bufferBlock[blockNum].getValues(position, memsize);  if (stringRow[0] == NIL) continue;  stringRow.erase(stringRow.begin());  splitRow = getSplitRow(table, stringRow);  if (cmp(table, splitRow, conditions))  datas.rows.push\_back(splitRow);  }  }  return datas;  } // Select all the data by conditions. |

* 1. RecordManager的插入记录函数

|  |
| --- |
| void insertRecord(Table & table, Row & splitRow) {  string stringRow = connectRow(table, splitRow);  string record = NIL + stringRow;  InsertPos ipos = buf.getInsertPosition(table);  buf.bufferBlock[ipos.BlockNum].setValues(ipos.Position, record);  } // Insert a record to the table |

* 1. RecordManager的删除记录函数

|  |
| --- |
| int deleteRecords(Table table, vector<Condition> conditions) {  string fileName = table.name + ".table";  int cnt = 0;  const int memsize = table.recordLength + 1;  const int recordNum = BLOCKSIZE / memsize;    for (int blockOffset = 0; blockOffset < table.blockNum; blockOffset++) {  int blockNum = buf.getBlockNum(fileName, blockOffset);  for (int recordOffset = 0; recordOffset < recordNum; recordOffset++) {  int position = recordOffset \* memsize;  if (buf.bufferBlock[blockNum].getValue(position) == NOTNIL) {  string stringRow = buf.bufferBlock[blockNum].getValues(position, memsize);  stringRow.erase(stringRow.begin());  Row splitRow = getSplitRow(table, stringRow);  if (cmp(table, splitRow, conditions)) {  buf.bufferBlock[blockNum].setValue(position, NIL);  cnt++;  }  }  }  }  return cnt;  } // Delete the records satisfy the conditions in the table and return the number of records. |

* 1. RecordManager的删除表函数

|  |
| --- |
| void dropTable(Table table) {  string fileName = table.name + ".table";  if (remove(fileName.c\_str()) != 0)  perror("Error in deleting file\n");  else buf.deleteFile(fileName);  } // delete the file and the blocks in buffer |

* 1. RecordManager的新建表函数

|  |
| --- |
| void createTable(Table table) {  string fileName = table.name + ".table";  ofstream fout(fileName.c\_str());  fout.close();  } // new a file |