

浙江大学

数据库系统实验报告

MINISQL

B+ 树、索引管理器与 GUI 实验报告

徐震 3180105504 18888916826

指导教师

孙建伶

2020 年 6 月 21 日

目录

第一部分 正文	4
第一章 实验目的	5
第二章 系统需求	7
2.1 需求概述	7
2.1.1 数据类型	7
2.1.2 索引的建立和删除	7
2.1.3 查找记录	7
2.1.4 插入和删除记录	7
2.1.5 图形界面	7
2.2 语法说明	7
2.2.1 创建索引语句	8
2.2.2 删除索引语句	8
2.2.3 选择语句	8
2.2.4 插入记录语句	9
2.2.5 删除记录语句	9
2.2.6 退出 MINISQL 系统语句	9
2.2.7 关于返回信息	10
2.2.8 关于图形界面	10
2.2.9 关于代码编辑器	10
2.2.10 关于错误提示	10
第三章 实验环境	12
第四章 模块设计	13
4.1 功能描述	13
4.1.1 建立/删除索引	13
4.1.2 查找/删除索引键	14

4.1.3	插入键值对	15
4.1.4	更新值内容	15
4.1.5	图形界面	16
4.2	主要数据结构	16
4.2.1	B+ 树	16
4.2.2	排序数组	17
4.3	类图与类间关系	17
4.3.1	B+ 树	17
4.3.2	异常类型	18
4.3.3	索引管理器	20
4.3.4	图形界面	20
第五章	模块实现	21
5.1	B+ 树模块具体实现	21
5.1.1	节点操作具体实现	21
5.1.2	辅助函数具体实现	22
5.1.3	查询操作具体实现	26
5.1.4	插入操作具体实现	27
5.1.5	删除操作具体实现	27
5.2	异常模块具体实现	28
5.3	备用数据结构具体实现	29
5.4	索引管理器模块具体实现	29
5.4.1	整体索引操作	30
5.4.2	元素查找，插入，删除具体实现	30
5.4.3	图形界面的设计	30
第六章	遇到的问题及解决方法	35
6.1	更改目录结构后 Python 无法正确引用其他.py 文件	35
6.1.1	问题描述	35
6.1.2	解决方法	36
6.2	使用 pickle 储存二进制形式的索引超出递归层数限制	38
6.2.1	问题描述	38
6.2.2	解决方法	38
第七章	总结	41

目录	3
第二部分 附录	42
第一章 接口说明	43
第二章 插图，表格与列表	45

第一部分

正文

第一章 实验目的

miniSQL 设计并实现一个精简型单用户 SQL 引擎 (DBMS)miniSQL, 允许用户通过字符界面输入 SQL 语句实现表的建立/删除; 索引的建立/删除以及表记录的插入/删除/查找。

索引管理器 我们负责设计的索引管理器模块主要负责数据库系统中的索引管理, 提供基于 B+ 树的索引实现并提高数据库查询/插入/删除效率。并通过提供易用接口与其他模块整合实现有效功能。

设计目的 通过对 miniSQL 的设计与实现, 提高学生的系统编程能力, 加深对数据库系统原理的理解。通过编程设计, 加深对数据库系统的理解并深入了解 B+ 树这一数据结构。以模块化方式构建大型计算机软件, 提高架构抽象能力并重视模块化和解耦合在软件设计中的作用。

功能实现 我们将实现如下的基本索引功能:

- 对单属性索引提供完整支持。
- 通过客制化比较函数可支持简单多属性索引。
- 对于表的主键自动建立 B+ 树索引/排序数组索引。
- 对于声明为 `unique` (唯一值) 的属性可以通过 SQL 语句由用户指定建立/删除 B+ 树索引。
- 支持除 B+ 树以外的数据结构索引 (如排序数组索引), 控制接口。
- 基于 `Exception` (异常) 的错误信息传递路径。
- 基于 `Python` 语言的多种数据类型索引支持。

图形界面 我们通过实现图形界面以环节用户对于命令行界面的恐惧（纵使这个图形界面中用户还是要敲写 SQL 语句），通过 `PyQt` 等易用接口实现一个简单的带有语法高亮的编辑器界面，并用对用户友好的方式输入输出各种信息，并提供对于一般编辑器操作的支持（例如多种快捷键和其他功能）。

第二章 系统需求

2.1 需求概述

2.1.1 数据类型

由于 Python 的动态语言特性，索引管理器支持所有可比较类型作为 key（对于不可比较类型，需用户自定义比较函数）。

2.1.2 索引的建立和删除

对于表的主键自动建立 B+ 树索引，对于声明为 unique 的属性可以通过 SQL 语句由用户指定建立/删除 B+ 树索引（因此，所有的 B+ 树索引都是单属性单值的）。

2.1.3 查找记录

可以通过指定用 and 连接的多个条件进行查询，支持等值查询和区间查询。

2.1.4 插入和删除记录

支持每次一条记录的插入操作；支持批量更新操作；支持每次一条或多条记录的删除操作。（where 条件是范围时删除多条）

2.1.5 图形界面

支持简单 GUI 操作和编辑器操作。

2.2 语法说明

我们需要提供索引管理器相关接口供接口模块调用，并提供图形界面与接口模块交互。索引管理器相关接口请参考附录。

2.2.1 创建索引语句

该语句的语法如下：若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。

```
1  -- 语法说明
2  create index 索引名 on 表名 ( 列名 );
3  -- 示例语句:
4  create index stunameidx on student ( sname );
```

Listing 2.1: Create Index Syntax and Example

2.2.2 删除索引语句

该语句的语法如下：若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。

```
1  -- 语法说明
2  drop index 索引名 ;
3  -- 示例语句
4  drop index stunameidx;
```

Listing 2.2: Drop Index Syntax and Example

2.2.3 选择语句

我们需实现基于索引的单值和范围查询语句。该语句的语法如下：

```
1  -- 语法说明
2  select * from 表名 ;
3  -- 或:
4  select * from 表名 where 条件 ;
5  -- 示例语句
6  select * from student;
7  select * from student where sno = '88888888';
8  select * from student where sage > 20 and sgender = 'F';
```

Listing 2.3: Select Syntax and Example

其中“条件”具有以下格式：*op and op ...and op*。op 是算术比较符：*=<><>=>=*。若该语句执行成功且查询结果不为空，则按行输出查询结果，第一行为属性名，其余每一行表示一条记录；若查询结果为空，则输出信息告诉用户查询结果为空；若失败，必须告诉用户失败的原因。

2.2.4 插入记录语句

我们需实现基于索引的单值插入或批量更新。该语句的语法如下：若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。

```
1  -- 语法说明
2  insert into 表名 values ( 值1 , 值2 , ... , 值n );
3  -- 示例语句
4  insert into student values ('12345678','wy',22,'M');
```

Listing 2.4: Insert Syntax and Example

2.2.5 删除记录语句

我们需实现基于索引的范围或单值删除。该语句的语法如下：

```
1  -- 语法说明
2  delete from 表名 ;
3  -- 或：
4  delete from 表名 where 条件 ;
5  -- 示例语句
6  delete from student;
7  delete from student where sno = '88888888';
```

Listing 2.5: Delete From Syntax and Example

若该语句执行成功，则输出执行成功信息，其中包括删除的记录数；若失败，必须告诉用户失败的原因。

2.2.6 退出 miniSQL 系统语句

我们在图形界面层直接实现了对于退出命令的执行，免于将图形接口暴露给内部 API。该语句的语法如下：

```
1  -- 语法说明
2  quit;
3  exit;
```

Listing 2.6: Quit miniSQL Syntax

2.2.7 关于返回信息

我们通过抛出异常的方式来反馈索引操作的状态，这样可以使接口模块免于处理过于冗杂的返回值，只需要接受并处理相关异常。

2.2.8 关于图形界面

我们希望开发一个图形界面以管理用户的输入输出，提供基本按钮和快捷键。并提供图标等。



图 2.1: GUI Requirements

2.2.9 关于代码编辑器

我们希望提供一个带有代码高亮功能，自动补全功能，自动换行功能，以及各种编辑器快捷键（复制一行，列操作，关键词操作）等的完整小型 sql 语句编辑视窗。同时，我们希望 MINISQL 支持多行语句执行的操作。

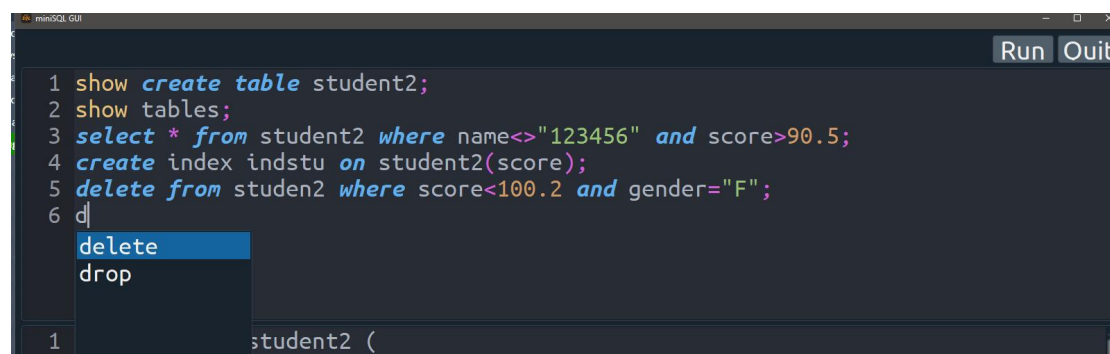
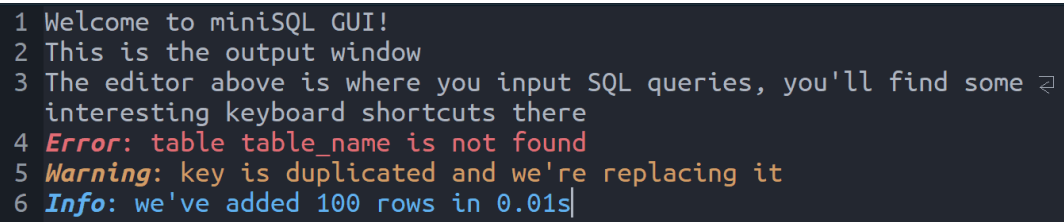


图 2.2: Editor Requirements

2.2.10 关于错误提示

我们希望错误提示以一种用户容易接受的格式呈现到程序上，例如在所有严重错误前加上 Error 标识符。结合 GUI 我们将会对这些格式化的错误信息进行高亮以方便用户识别。



```
1 Welcome to miniSQL GUI!
2 This is the output window
3 The editor above is where you input SQL queries, you'll find some interesting keyboard shortcuts there
4 Error: table table_name is not found
5 Warning: key is duplicated and we're replacing it
6 Info: we've added 100 rows in 0.01s
```

图 2.3: Error Message Requirements

第三章 实验环境

主要开发语言 Python 3.7.*/3.8.*

主要开发环境

- PyCharm 2020.1
- Visual Studio Code 1.45

经过测试的系统环境

- MICROSOFT WINDOWS [VERSION 10.0.18363.836]
- UBUNTU WSL2
- MACOS CATALINA

Python 包要求 QScintilla, QDarkStyle, PyQt5

实验系统环境 MICROSOFT WINDOWS [VERSION 10.0.18363.836]

实验处理器环境 Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 12 Logical Processors

实验内存环境 SODIMM 15.8GB/16.0GB

实验硬盘环境 KBG30ZMS512G NVMe TOSHIBA 512GB

第四章 模块设计

4.1 功能描述

4.1.1 建立/删除索引

实际的数据库应用中，有在建立表时创建索引和表中有数据情况下创建索引的需求。区别在于，前者不需要其他模块为索引管理器提供数据，仅仅需要分配一个新索引所需的内存和磁盘空间；而后者要求发出建立索引请求的模块提供相应的数据块。默认情况下我们以记录的行号作为 B+ 树中键值对上的“值”，因此我们要求相应数据块是按照它们将来被查找的顺序提供的。

建立索引 正如图4.1所示的¹。为了方便演示，让我们用其中的第一列作为建立索引的数据集。我们会按顺序将提供的数据插入索引数据结构，因此在这一个键值对中，键为第一列数据值，如 10101，而值为行号，如 0。

<i>record 0</i>	10101	<i>Srinivasan</i>	<i>Comp. Sci.</i>	<i>65000</i>
<i>record 1</i>	12121	<i>Wu</i>	<i>Finance</i>	<i>90000</i>
<i>record 2</i>	15151	<i>Mozart</i>	<i>Music</i>	<i>40000</i>
<i>record 3</i>	22222	<i>Einstein</i>	<i>Physics</i>	<i>95000</i>
<i>record 4</i>	32343	<i>El Said</i>	<i>History</i>	<i>60000</i>
<i>record 5</i>	33456	<i>Gold</i>	<i>Physics</i>	<i>87000</i>
<i>record 6</i>	45565	<i>Katz</i>	<i>Comp. Sci.</i>	<i>75000</i>
<i>record 7</i>	58583	<i>Califleri</i>	<i>History</i>	<i>62000</i>
<i>record 8</i>	76543	<i>Singh</i>	<i>Finance</i>	<i>80000</i>
<i>record 9</i>	76766	<i>Crick</i>	<i>Biology</i>	<i>72000</i>
<i>record 10</i>	83821	<i>Brandt</i>	<i>Comp. Sci.</i>	<i>92000</i>
<i>record 11</i>	98345	<i>Kim</i>	<i>Elec. Eng.</i>	<i>80000</i>

图 4.1: 线性记录储存方式

¹图4.1来源于 *Database System Concepts 6th Edition Abraham Silberschatz 等*。

这样处理的原因在于，我们希望日后通过索引根据查找键快速找到记录对应的位置，而其在表格中的相对位置是最方便的寻址信息之一。若我们按照索引的自定义值储存插入的键，则索引失去加快搜索的作用；若我们插入的值为记录的绝对磁盘位置信息，则缓存管理器失效，且数据的移动会导致索引失效。

在内存中创建索引（并插入相应数据后），索引管理器会将索引的内存信息叫给缓存管理器，由其决定是否应将内存保留或者存储到磁盘中，同时返回给索引管理器一个唯一的索引标号（索引管理器会将其继续返回给上层模块），日后将根据这一唯一标识符从缓存管理器中取得相应索引（无论是通过读取磁盘文件还是直接获取内存指针）。

删除索引 我们通过上述的唯一标识符给缓存管理器发出删除信号，完成删除操作。

4.1.2 查找/删除索引键

查找和删除操作支持快速范围操作，并且两者在具体实现上有极大相似性，我们通过抽象两者的操作来提高代码复用率。我们首先直接判断用户进行的是范围还是单值查找，并且替前从缓存管理器中取得相应索引内容（内存或硬盘中）。

单值操作 我们调用 B+ 树相应查找接口获得应查找的值，并通过异常来进行错误通讯。若查找成功则直接返回，否则抛出相关异常。

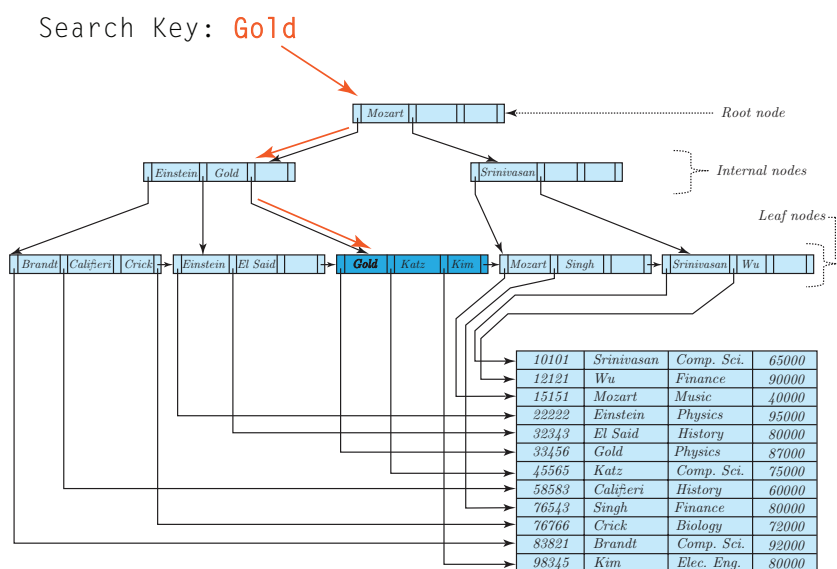


图 4.2: 单值操作

范围操作 我们首先会检查用户给予的范围是否有效²，同样的，我们使用异常来进行错误通讯，这使得接口模块能方便的实现错误处理。接着我们查询范围两端的值³并根据返回的节点情况进行相关操作，对于查找指令，这一操作是返回查询得到的相关信息；对于删除指令，这一操作是操作 B+ 树删除相关的值，并在操作全部完成后将修改后的索引叫给缓存管理器。如图4.3所示（加深部分为需要查找或删除的部分）。

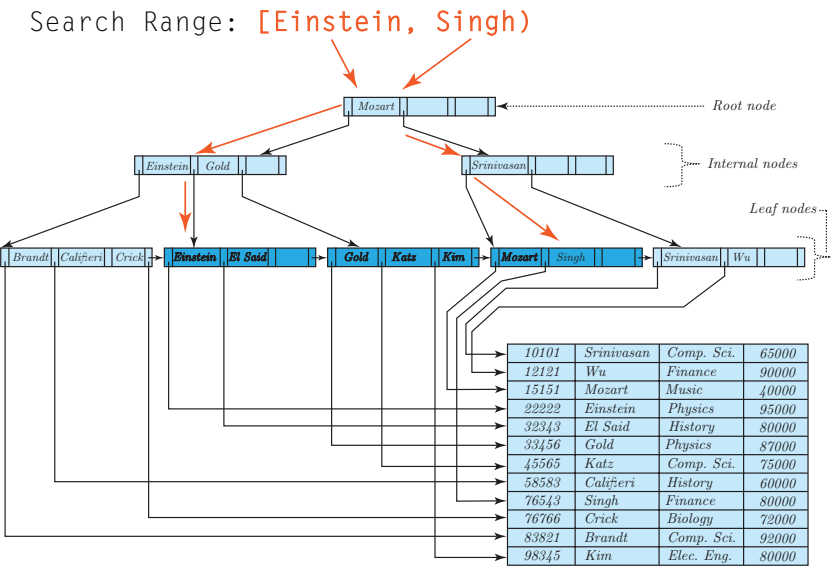


图 4.3: 范围操作

4.1.3 插入键值对

我们单独实现了插入键值对功能，因为其逻辑相对于查询和删除操作都有所不同（需检查重复元素等）。类似的，索引管理器会首先向缓存管理器请求相关索引内容。接着我们调用插入操作，将是否允许替换的信息传递给 B+ 树的相关函数直接进行操作⁴。

4.1.4 更新值内容

由于我们使索引中键值对中值指向记录在某张表中的位置，而这一位置在数据库运行过程中可能会发生很大变化⁵，我们给外界留出批量修改值的接口。

这一操作类似于范围操作中涉及到的内容，我们需要对 B+ 树中的有效节点数目进行检查，根据节点数目选取不同的处理方式。

²例如，范围左右下标是否为左小右大，或被查找的树是否为空等

³我们默认范围查找的区间是左闭右开的。

⁴原因在于 B+ 树内部实现中也需要调用查找相关功能，与其规定执行流程，不如让 B+ 树具体实现获得最优执行流程

⁵例如闲时的数据库清理和记录重排序等，亦或记录管理器采取了不同的数据储存模型。

4.1.5 图形界面

我们拓展了 API 模块的接口，通过图形界面来提高用户体验。我们设计了一个编辑器界面，加入了语法提示与自动补全。我们通过 PyQt 来实现对图形界面的构建。

4.2 主要数据结构

4.2.1 B+ 树

⁶ B+ 树是一种树数据结构，通常用于数据库和操作系统的文件系统中。B+ 树的特点是能够保持数据稳定有序，其插入与修改拥有较稳定的对数时间复杂度。B+ 树元素自底向上插入，这与二叉树恰好相反。

B+ 树在节点访问时间远远超过节点内部访问时间的时候，比可作为替代的实现有着实在的优势。这通常在多数节点在次级存储比如硬盘中的时候出现。通过最大化在每个内部节点内的子节点的数目减少树的高度，平衡操作不经常发生，而且效率增加了。这种价值得以确立通常需要每个节点在次级存储中占据完整的磁盘块或近似的大小。

B+ 背后的想法是内部节点可以有在预定范围内的可变量目的子节点。因此，B+ 树不需要像其他自平衡二叉查找树那样经常的重新平衡。对于特定的实现在子节点数目上的低和高边界是固定的。例如，在 2-3 B 树（常简称为 2-3 树）中，每个内部节点只可能有 2 或 3 个子节点。如果节点有无效数目的子节点则被当作处于违规状态。

B+ 树的创造者 Rudolf Bayer 没有解释 B 代表什么。最常见的观点是 B 代表平衡 (balanced)，因为所有的叶子节点在树中都在相同的级别上。B 也可能代表 Bayer，或者是波音 (Boeing)，因为他曾经工作于波音科学研究实验室。

查找 查找以典型的方式进行，类似于二叉查找树。起始于根节点，自顶向下遍历树，选择其分离值在要查找值的任意一边的子指针。在节点内部典型的使用是二分查找来确定这个位置。

插入 节点要处于违规状态，它必须包含在可接受范围之外数目的元素。首先，查找要插入其中的节点的位置。接着把值插入这个节点中。如果没有节点处于违规状态则处理结束。如果某个节点有过多元素，则把它分裂为两个节点，每个都有最小数目的元素。在树上递归向上继续这个处理直到到达根节点，如果根节点被分裂，则创建一个新根节点。为了使它工作，元素的最小和最大数目典型的必须选择为使最小数不小于最大数的一半。

删除 首先，查找要删除的值。接着从包含它的节点中删除这个值。如果没有节点处于违规状态则处理结束。如果节点处于违规状态则有两种可能情况：它的兄弟节点，就是同一个父

⁶ 此段内容摘自 [WikiPedia](#)

节点的子节点，可以把一个或多个它的子节点转移到当前节点，而把它返回为合法状态。如果是这样，在更改父节点和两个兄弟节点的分离值之后处理结束。亦或，它的兄弟节点由于处在低边界上而没有额外的子节点。在这种情况下把两个兄弟节点合并到一个单一的节点中，而且我们递归到父节点上，因为它被删除了一个子节点。持续这个处理直到当前节点是合法状态或者到达根节点，在其上根节点的子节点被合并而且合并后的节点成为新的根节点。

4.2.2 排序数组

本数据结构是为了配合主键而实现的，采用最普通的排序数组查找方式，并在数组内部采用二分查找进行相关操作。可从无限子树数目的 B+ 树抽象的到，因此我们可以较为方便的统一两者的接口。

值得注意的是，为了配合主键和记录管理器中数据的储存方式，我们往往使用一种特殊的类作为排序数组的内部容器：一种返回当前下标的特殊数组⁷。

4.3 类图与类间关系

4.3.1 B+ 树

B+ 树的类图实现如图4.4所示⁸⁹。

⁷我们可以利用这一特性而使得这种储存不占用任何空间，而同时保证接口的一致性。

⁸为了类图完整性，我们也列举了除用户自定义类型以外的类型。

⁹我们使用矢量图渲染了字体，若图表过小请放大查看。

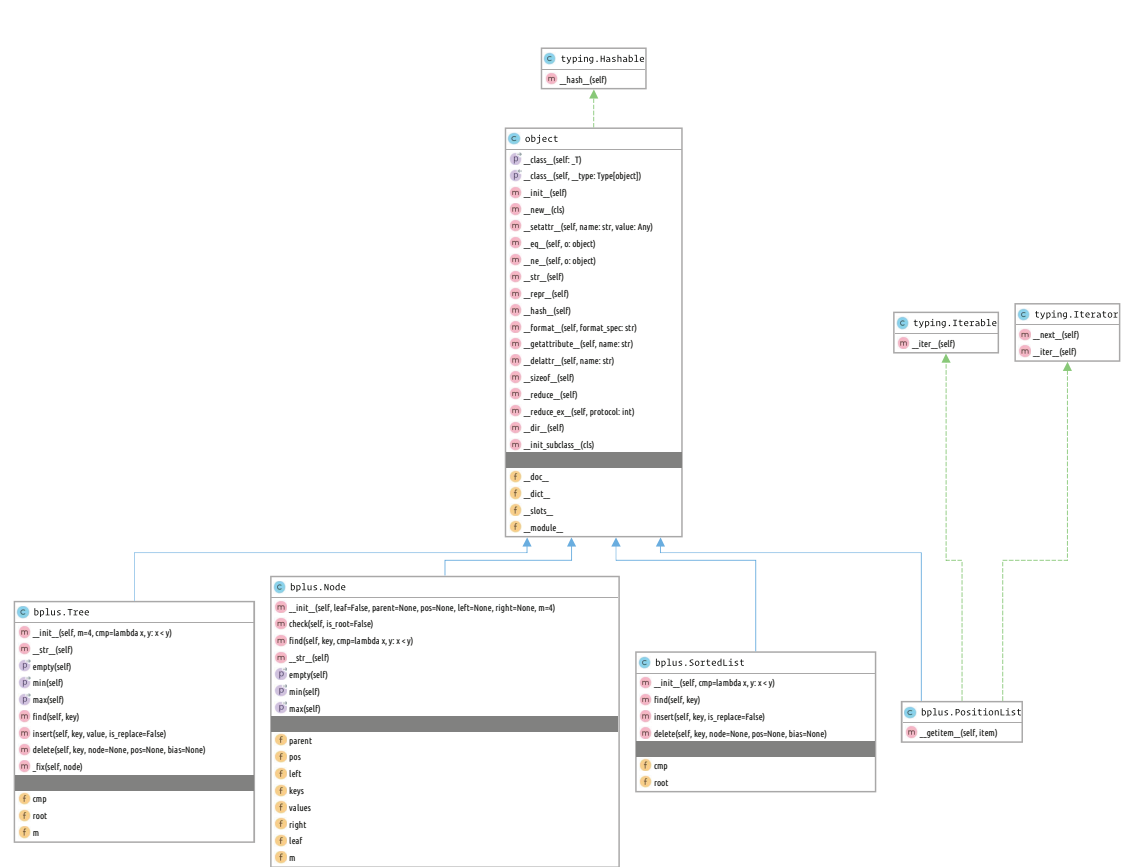


图 4.4: B+ 树的类图与类间关系

4.3.2 异常类型

程序使用的异常类图如4.5所示。

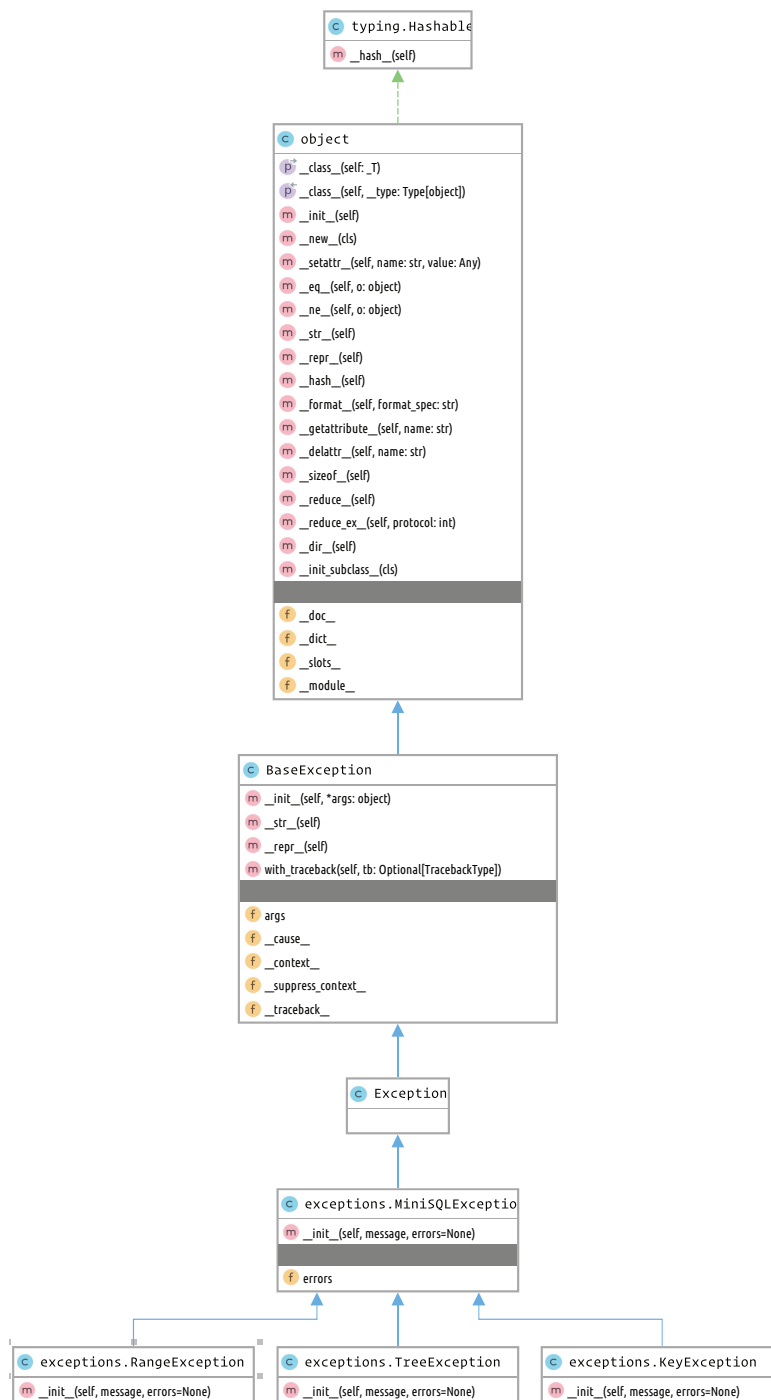


图 4.5: 异常的类型图与类间关系

4.3.3 索引管理器

在具体的索引管理器实现上，我们采用了模块层面上的抽象而非类层面的，这更贴合 Python 语言的风格。这样能保证尽量大的抽象层次与代码复用率。值得注意的是，我们在实现 B+ 树相关操作的时候也使用了静态函数来抽象部分内容¹⁰。

4.3.4 图形界面

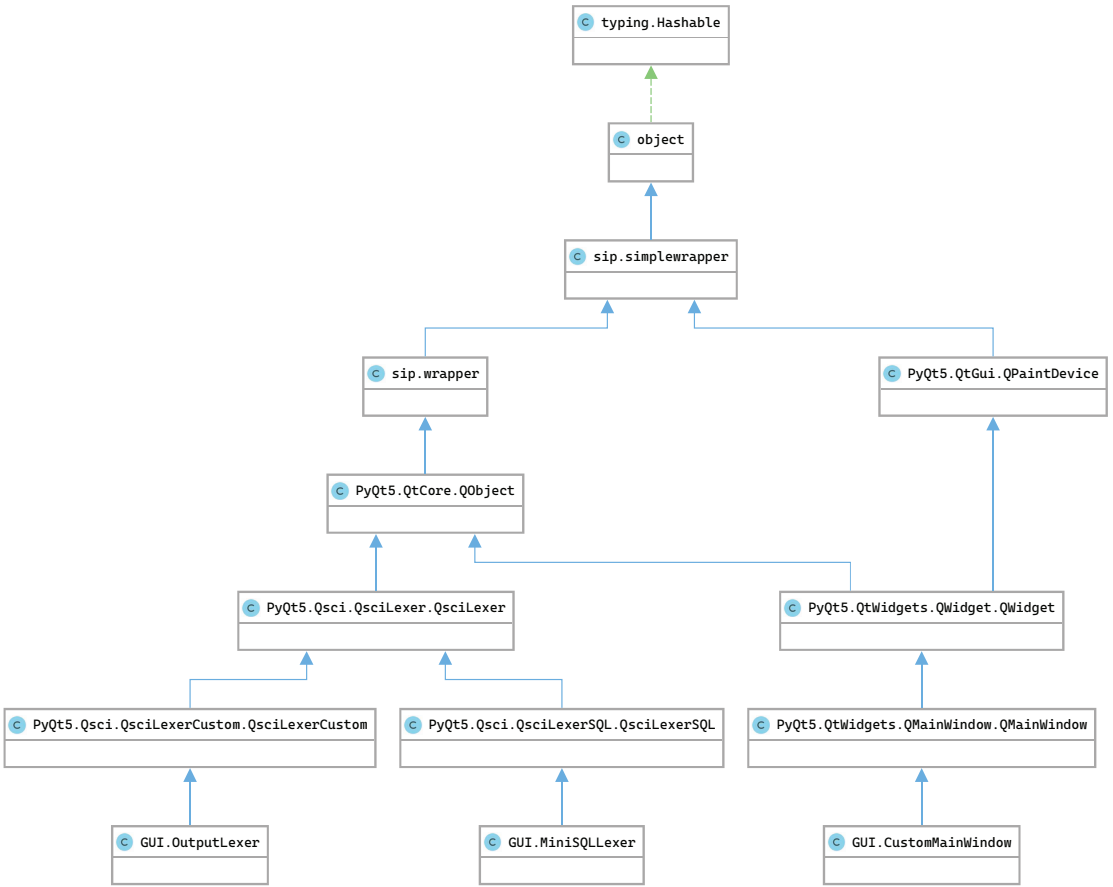


图 4.6: 图形界面类图

¹⁰我们将在下一部分详细阐述实现细节

第五章 模块实现

我们通过 Python 语言来实现各个模块和类。下面我们会通过

- 伪代码。
- 图表。
- 简短的源码。

等方式来阐述各个模块的具体实现。

5.1 B+ 树模块具体实现

5.1.1 节点操作具体实现

节点是 B+ 树中的数据储存元素，每个节点根据要求存储一定数量的排序数据。我们需首先实现节点相关操作。算法1实现了一种 $O(\log n)$ 级别的二分下届查找，用于在某个节点中寻找关键元素。在实际实现中，我们会在 C++ 项目中采用模板，但 Python 语言的动态类型特性使得这类普适算法很容易应用到具体的数据类型上。

Algorithm 1 General Binary Search

Input: The *items* in iterable list and the *key* to be searched, optionally the *comparator*

Output: The *lower_bound* of the *key* in *items*, as index

```
1: function BINARYSEARCH(items, key, comparator = lambda)
2:   left  $\leftarrow$  0
3:   right  $\leftarrow$  LENGTH(items)
4:   while True do
5:     mid  $\leftarrow$  (left + right)/2
6:     if left  $\geq$  right - 1 then
7:       Break
8:     end if
9:     if key < items[mid] then                                 $\triangleright$  Use custom comparator if present
```

```

10:          $right \leftarrow mid$ 
11:     else
12:          $left \leftarrow mid$ 
13:     end if
14: end while
15: return  $mid$ 
16: end function

```

B+ 数的查找操作实现如算法2所示，虽然我们正在节点模块介绍这一算法，但实际使用中我们往往会通过一颗 B+ 树的根来调用此算法¹²。

Algorithm 2 Find Operation on Node

Input: Current *node* and the *key* to be found, optionally the customer *comparator*

Output: The leaf *node* and the *position* of the found element, plus a bias to indicate whether the key is smaller than the smallest element

```

1: function FIND(node, key, comparator = lambda)
2:      $position \leftarrow \text{BINARYSEARCH}(node.items, key, comparator)$ 
3:     if  $position = 0 \cap key < node.keys[0]$  then
4:          $bias \leftarrow 0$ 
5:     else
6:          $bias \leftarrow 1$ 
7:     end if
8:     if ISLEAF(node) then
9:         return GROUP(node, position, bias)
10:    else
11:        return FIND(node.children[position + bias], key, comparator)
12:    end if
13: end function

```

5.1.2 辅助函数具体实现

为了方便调用/实现 B+ 树相关功能，并提供代码的抽象程度和复用率，我们使用了一些普适性的函数。例如算法3所示的操作会在节点的头元素修改后修改父节点的相关头元素，如图5.1所示。

¹本算法会从当前节点开始，一直查询到叶节点。

²要查找某棵树上的元素，我们只需要从根节点调用此方法。

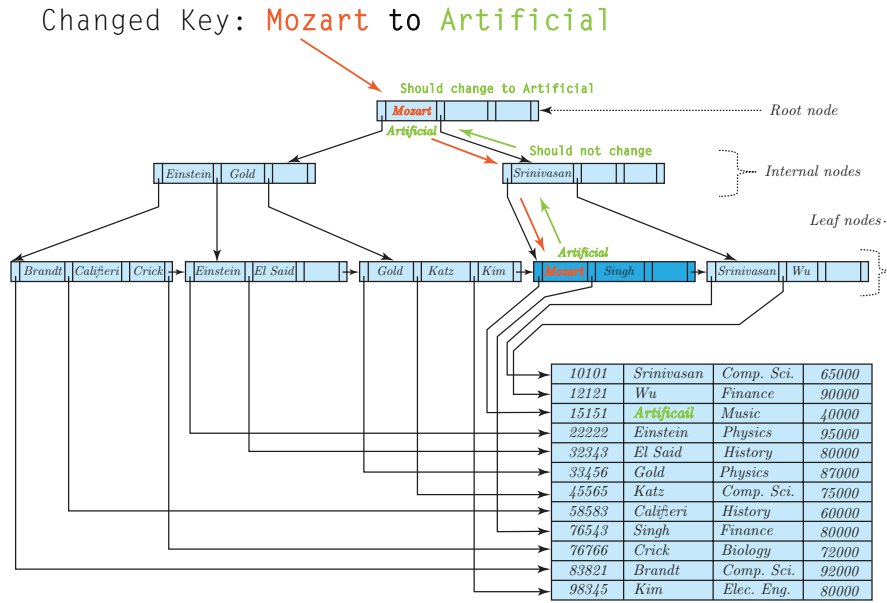


图 5.1: FixParent

Algorithm 3 Fix a Node's Parent After the Left Most Value Is Updated**Input:** The *node* whose parent is to be fixed

```

1: function FIXPARENT(node)
2:   key ← node.keys.front
3:   while EXIST(node.parent) ∩ node.position = -1 do           ▷ -1 (position in parent)
       indicates the node to be the first of parent's children
4:     node ← node.parent
5:   end while
6:   if EXIST(node.parent) then
7:     node.parent.keys[node.position] ← key
8:   end if
9: end function

```

B+ 树中节点出现异常³后，我们可以直接通过合并节点或拆分节点进行修复操作，也可以首先检查问题节点的姊妹节点是否有修复余地⁴。本实验中我们采用了首先检查姊妹节点的方式。我们会在进行分裂或合并操作前，检查姊妹是否能容纳过饱和的冗余或是否有足够多元素为当前问题节点作出补充。若仍有空余/节点数目足够，则进行修复操作⁵。此函

³过饱和，欠饱和等。⁴有些 B+ 树不会实现这一功能，以降低开发难度。⁵对于插入造成的过饱和，删除造成的欠饱和，我们仅仅需要实现一份代码，将问题节点和姊妹节点互换即可复用这一份代码

数的伪代码如算法4所示，图5.2演示了这一过程。

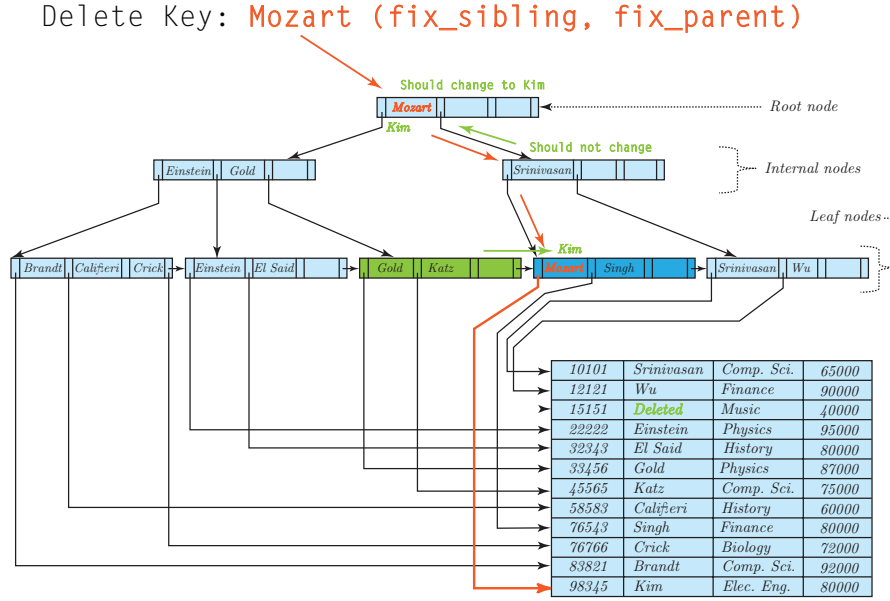


图 5.2: FixSibling

Algorithm 4 Fix a Node by Borrowing From a Sibling

Input: The *node* to be fixed

Output: *True* if fixed, and *False* if otherwise

```

1: function FIXSIBLING(node)
2:   sibling  $\leftarrow$  node.sibling  $\triangleright$  next lines will go through both left and right sibling, we'll
   illustrate the procedure when sibling is left
3:   if EXIST(sibling)  $\cap$  CHECK(node) then  $\triangleright$  Check whether node overflows or
   underflows
4:     MOVEVALUE(node.keys.front, sibling.keys.back)  $\triangleright$  Move front of node's keys
   to end of sibling's keys
5:     MOVEVALUE(node.values.front, sibling.values.back)
6:     FIXPARENT(node)
7:     return True
8:   else
9:     return False
10:  end if
11: end function

```

算法5列举了修改在插入或删除操作后出现问题节点的 B+ 树的完整过程⁶。它调用了上述的 `FixSibling` 和 `FixParent` 方法。

Algorithm 5 Fix a B+ Tree After Insertion/Deletion

Input: The *tree* to be fixed, and the *node* we've inserted new value on

```

1: function FIXTREE(tree, node)
2:   while True do
3:     flow  $\leftarrow$  CHECK(node)      ▷ 1 for overflow and -1 for underflow, 0 if node is ok
4:     if flow = 0 then
5:       Break
6:     end if
7:     if FIXSIBLING(node) then
8:       Break
9:     end if
10:    if flow = 1 then                                     ▷ overflow, should split up
11:      if node = tree.root then
12:        tree.root  $\leftarrow$  NEWNODE
13:        APPEND(tree.children, node)
14:        node.parent  $\leftarrow$  tree.root
15:      end if
16:      new  $\leftarrow$  NEWNODE
17:      if EXIST(new.right) then
18:        new.right.left  $\leftarrow$  new
19:        right  $\leftarrow$  new.right
20:        while EXIST(right) do
21:          INCREMENT(right.position)
22:          right  $\leftarrow$  right.right
23:        end while
24:      end if
25:      node.right  $\leftarrow$  new
26:      new.keys  $\leftarrow$  LEFTHALF(node.keys)
27:      node.keys  $\leftarrow$  RIGHTHALF(node.keys)
28:      new.values  $\leftarrow$  LEFTHALF(node.values)
29:      node.values  $\leftarrow$  RIGHTHALF(node.values)
30:      INSERT(new.parent.keys, new.position, new.keys.front)
31:      INSERT(new.parent.children, new.position + 1, new)
  
```

⁶包含分裂节点与合并节点的具体内容

```

32:         if  $\neg$ ISLEAF(node) then
33:             DELETE(new.keys.front)    ▷ Delete is to delete given element from its
container like a Python list
34:         end if
35:         if  $\neg$ ISLEAF(new) then
36:             for child  $\leftarrow$  new.children.front do new.children.back
37:                 child.parent  $\leftarrow$  new
38:                 child.posion  $\leftarrow$  index
39:             end for
40:         end if
41:         node  $\leftarrow$  new.parent    ▷ Prepare for the next loop, peculate up one level
42:     else                                ▷ underflow, should merge
43:         if EXIST(node.right) then    ▷ By default we'd merge current node and its
right sibling
44:             node  $\leftarrow$  node.left
45:         end if
46:         node.keys  $\leftarrow$  CONCATENATE(node.keys, node.right.keys)
47:         node.children  $\leftarrow$  CONCATENATE(node.children, node.right.children)
48:         DELETE(node.parent.keys, node.position + 1)
49:         DELETE(node.parent.children, node.position + 2)
50:         temp  $\leftarrow$  node.right
51:         node.right  $\leftarrow$  temp.right
52:         DELETE(temp)
53:         node  $\leftarrow$  node.parent    ▷ Peculate up one level, prepare for next loop
54:         if node = tree.root then
55:             tree.root = node.children.front
56:             DELETE(node)
57:         end if
58:     end if
59: end while
60: end function

```

5.1.3 查询操作具体实现

详见节点查询部分。

5.1.4 插入操作具体实现

我们通过调用上述的 `Find` 与 `FixTree` 函数来实现查询操作。具体实现如伪代码6所示。插入等操作具有 $\lceil O(\log_{\lceil \frac{m}{2} \rceil} N) \rceil$ 的时间复杂度。

Algorithm 6 Insertion in B+ Tree

Input: The *tree* object to be inserted into, and the *key-value* pair to be inserted

```

1: function INSERT(tree, key, value)
2:   if ISEMPY(tree) then
3:     APPEND(tree.keys, key)
4:     APPEND(tree.values, value)
5:     return
6:   end if
7:   GROUP(node, position, bias)  $\leftarrow$  FIND(tree.root, key, tree.comparator)
8:   if node.keys[pos] = key then
9:     Raise KEYEXCEPTION(duplicated key)
10:  end if
11:  INSERT(node.keys, position + bias, key)
12:  INSERT(node.values, position + bias, value)
13:  FIX(node)
14:  return
15: end function

```

5.1.5 删除操作具体实现

类似插入操纵，我们也会调用上述的 `Find` 与 `FixTree` 函数来实现 B+ 树中的删除操作。算法7描述了这一过程。

Algorithm 7 Deletion in B+ Tree

Input: The *tree* whose key-value pair is to be deleted, and the *key* to be deleted

```

function DELETE(tree, key)
  if EMPTY(tree) then
    raise TREEEXCEPTION(Empty tree)
  end if
  GROUP(node, position, bias)  $\leftarrow$  FIND(tree, key)
  if node.keys[position]  $\neq$  key then
    raise KEYEXCEPTION(Cannot find key)
  end if

```

```

DELETE(node.keys[position])
DELETE(node.children[position])    ▷ On leaf values and keys have the same indices
flow ← CHECK(node)
if flow ≠ 0 then
    FIXPARENT(node)
end if
FIXTREE(tree, node)
end function

```

5.2 异常模块具体实现

我们通过异常机制来传递错误信息，省去了检测错误码的冗长代码。实验中 B+ 树和索引管理器模块用到的异常如列表5.1所示。

```

1  class MiniSQLException(Exception):
2      """
3      raised when we cannot find a particular key
4      or duplication occurs
5      """
6
7  def __init__(self, message, errors=None):
8      # Call the base class constructor with the parameters it needs
9      super().__init__(message)
10
11     # Now for your custom code...
12     self.errors = errors
13
14
15  class KeyException(MiniSQLException):
16     def __init__(self, message, errors=None):
17         super().__init__(message, errors)
18
19
20  class RangeException(MiniSQLException):
21     def __init__(self, message, errors=None):
22         super().__init__(message, errors)
23
24
25  class TreeException(MiniSQLException):
26     def __init__(self, message, errors=None):

```

27

```
super().__init__(message, errors)
```

Listing 5.1: 异常类型的具体实现

5.3 备用数据结构具体实现

为了满足特殊操作⁷我们实现了一种特殊的数据结构：排序数组索引结构，其具备了如下特点：

- 查询操作时间同样为 $O(\log N)$ 。
- 进行各项其他操作表现如同数组，没有冗杂的数据结构与算法支撑。
- 提供与 B+ 树完全相同的接口。

其具体实现和算法1, 2以及6描述的很相似，只不过去除了复杂的修复部分。我们通过图5.3来进一步描述这一过程⁸。

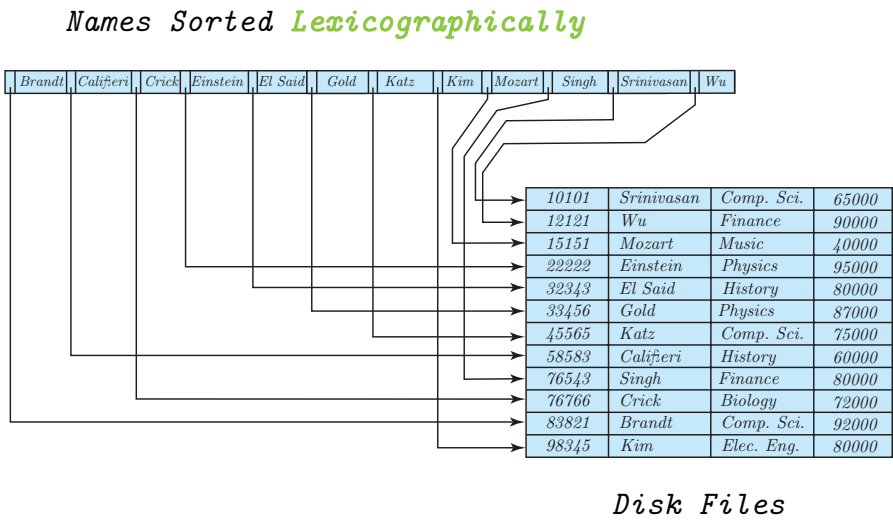


图 5.3: 排序数组具体实现

5.4 索引管理器模块具体实现

我们根据索引管理器系统的要求实现了逻辑管理器模块，并进行了一定程度上的逻辑抽象和代码复用。

⁷如，要求插入的键值对值永远是当前的下标。
⁸若插图过小请放大，我们使用了矢量图。

5.4.1 整体索引操作

对完整索引进行的操作如算法8所示。

Algorithm 8 索引的插入删除

```

function CREATEINDEX(data_list, comparator)
    tree  $\leftarrow$  NEWTREE(comparator)
    for all data  $\in$  data_list do
        INSERTTREE(tree, data[i])
    end for
    SAVEBUFFERINDEX(tree)
end function

function DROPINDEX(id)
    DROPBUFFERINDEX(id)
end function

```

5.4.2 元素查找，插入，删除具体实现

在索引管理器中，我们支持如下元素操作：

- 根据键查找某个元素的内容（返回行号或某种记录指针）。
- 根据两个键查找区间范围内元素的内容（返回行号或记录指针的列表）。
- 通过键值对进行插入操作。
- 根据某个键删除某个元素。
- 根据两个键删除区间元素。

其中插入操作逻辑最为简单⁹，单独实现。而查询、删除操作的逻辑几乎完全相同，我们在实际实现中使用了泛化的 OPERATESINGLE 与 OPERATERANGE 来减少重复代码。在算法9所示的伪代码中，我们通过**查找操作**描述了这一过程以方便读者阅读。

5.4.3 图形界面的设计

我们使用：

- PyQT5

⁹原因在于我们已经实现了 B+ 树并提供了相应接口。

Algorithm 9 元素的查找，插入，和删除

```

function INSERTELEMENT(id, key, value)
    tree  $\leftarrow$  GETBUFFERINDEX(id)
    INSERTTREE(tree, key, value)
    SAVEBUFFERINDEX(tree, ind)  $\triangleright$  Specify the id, so that buffer manager can replace
end function

function FINDSINGLE(id, key)
    tree  $\leftarrow$  GETBUFFERINDEX(id)
    GROUP(node, position, bias)  $\leftarrow$  FINDTREE(tree, key)
    if node.keys[position]  $\neq$  key then
        raise KEYEXCEPTION(Cannot find key)
    else
        return node.values[position]
    end if
end function

function FINDRANGE(id, begin, end)
    CHECKRANGE(begin, end)  $\triangleright$  Would raise certain error if not valid
    tree  $\leftarrow$  GETBUFFERINDEX(id)
    values  $\leftarrow$  NEWLIST
    GROUP(nodea, postiona, biasa)  $\leftarrow$  FINDTREE(tree, begin)
    GROUP(nodez, postionz, biasz)  $\leftarrow$  FINDTREE(tree, end)
    if nodea = nodez then  $\triangleright$  For simplicity we'd omit the part to check for values within
a node for pseudocode (implemented in actual code)
        for all value  $\in$  nodea.values do
            APPEND(values, value)
        end for
    else
        for all value  $\in$  nodea.values do
            APPEND(values, value)
        end for
        while nodea.right  $\neq$  nodez do
            nodea  $\leftarrow$  nodea.right
            for all value  $\in$  nodea.values do
                APPEND(values, value)
            end for
        end while
        for all value  $\in$  nodez.values do
            APPEND(values, value)
        end for
    end if
    return values
end function

```

- QDARKSTYLE
- QSCINTILLA

来搭建了 GUI 的整体框架，并通过添加：

- 编辑器相关操作。
- 语法解析与代码高亮。
- 错误信息高亮。
- 语法自动补全。
- 快捷键。

等功能来完善用户体验。由于这部分代码与数据库系统设计的核心没有太大关联，我们仅仅列出一部分核心代码，如列表5.2所示。

```
1  # Import modules
2  ...
3
4  # Define colors and fonts and geometry constants
5  ...
6
7  class OutputLexer(QSCIEXERCUSTOM):
8      def __INIT__(SELF, PARENT):
9          # Set up lexer font
10         # Set up lexer highlighting color
11         # Set up lexer highlighting format
12
13     def STYLETEXT(SELF, START, END):
14         # 1. Initialize the styling procedure
15         # -----
16         SELF.STARTSTYLING(START)
17
18         # 2. Slice out a part from the text
19         # -----
20         TEXT = SELF.PARENT().TEXT()[START:END]
21
22         ...
23
24     def DESCRIPTION(SELF, STYLE):
25         ...
26
```

```

27 class MiniSQLLexer(QScintilla):
28     def __init__(self, parent):
29         # Set up lexer font
30         # Set up lexer highlighting color
31         # Set up lexer highlighting format
32         ...
33
34 class CustomMainWindow(QMainWindow):
35     def __init__(self):
36         super(CustomMainWindow, self).__init__()
37         init() # the API setup
38         # 1. Define the geometry of the main window
39         init_geometry = (300, 300, 800*WINDOW_FONT_SCALE, 400*WINDOW_FONT_SCALE)
40
41         self.setGeometry(*init_geometry)
42         self.setWindowTitle("MINISQL GUI")
43         # Create frame and layout
44         ...
45         # Place buttons
46         ...
47         # ! Make instance of QsciScintilla class!
48         # Add autocompletion and lexer
49         self.editor = QsciScintilla()
50         # adding the SQL lexer to the editor
51         self.sql_lexer = MiniSQLLexer(self.editor)
52         self.api = QscisAPIs(self.sql_lexer)
53         ...
54         self.output.append("ERROR: TABLE TABLE_NAME IS NOT FOUND\n")
55         self.output.append("WARNING: KEY IS DUPLICATED AND WE'RE REPLACING IT\n")
56         self.output.append("INFO: WE'VE ADDED 100 ROWS IN 0.01s")
57         ...
58         self.editor.setFocus()
59         self.show()
60
61     def setup_editor_style(self, editor):
62         ...
63         !!!!!
64     def run_sql(self):
65         content = self.editor.text()
66         stripped = content.strip().lower()
67         if stripped in ["QUIT", "QUIT;", "EXIT", "EXIT;"]:
68             self.exit_sql()

```

```
69         else:
70             SELF.OUTPUT.SETTEXT(STR_MAIN(CONTENT))
71     def EXIT_SQL(SELF):
72         SQL_EXIT()
73         SELF.CLOSE()
74         '''
75     ''' End Class '''
76 def MAIN():
77     APP = QApplication(SYS.ARGV)
78     # Load local font
79     ...
80     APP.SETWINDOWICON(QIcon('FIGURE/MINISQL.SVG'))
81     STYLE_SHEET = QDarkStyle.LOAD_STYLESHEET(QT_API='PYQT5')
82     APP.SETSTYLE_SHEET(STYLE_SHEET)
83     MYGUI = CUSTOMMAINWINDOW()
84     SYS.EXIT(APP.EXEC_())
85
86     '''
```

Listing 5.2: GUI 具体实现

第六章 遇到的问题及解决方法

6.1 更改目录结构后 Python 无法正确引用其他.py 文件

6.1.1 问题描述

为了使文件结构更为整洁方便，我们尝试了使用模块来封装各个.py 文件与直接调用.py 文件或相对引用等多种方式，最初的文件目录如列表6.1所示：

```
1      ./
2      |-- SOME_DATA_FOLDERS
3      |-- SOME_DATA_FILES
4      |-- BPLUS.PY
5      |-- BUFFER.PY
6      |-- EXCEPTIONS.PY
7      |-- ...
8      |-- INDEX.PY
9      |-- README.MD
```

Listing 6.1: 可能的文件目录

这种方式下 PYTHON 脚本只需要使用普通的 `IMPORT` 或 `FROM ... IMPORT ...` 即可正确引用相关文件。

且得益于 IDE 特性¹，当前目录会被自动设置为程序的运行时的根目录，也就意味着程序中对于文件的操作也可正常进行（即使 PYTHON 脚本与数据文件不在同一文件夹下）。因此我们采用了如列表6.2所示的文件组织结构。

```
1      ./
2      |-- SOME_DATA_FOLDERS
3      |-- SOME_DATA_FILES
4      |-- CODE/
5          |-- BPLUS.PY
```

¹Visual Studio Code, PyCharm 等

```
6      |-- BUFFER.PY
7      |-- EXCEPTIONS.PY
8      |-- ...
9      |-- INDEX.PY
10     |-- README.MD
11     |-- LICENSE
```

Listing 6.2: 可能的文件目录

这种文件组织结构下，CODE 中的代码可以正常访问根目录中的数据文件和数据文件夹，但无法引用同在 CODE 文件夹下的其他.PY 文件。如列表6.3所示。

```
1  IN [1]: IMPORT BLABLABLA
2  -----
3  MODULENOTFOUNDERROR                                TRACEBACK (MOST RECENT CALL LAST)
4  <IPYTHON-INPUT-1-7E2F26FA0A8B> IN <MODULE>
5  ----> 1 IMPORT BLABLABLA
6
7  MODULENOTFOUNDERROR: NO MODULE NAMED 'BLABLABLA'
8
9  IN [2]:
```

Listing 6.3: 无法引用同一文件下的模块

6.1.2 解决方法

我们注意到如下事实：

- PYTHON 根据 PYTHONPATH 设定的目录进行文件，模块等的搜索。
- PYCHARM 会将当前工作目录的根目录当作 PYTHON 工作目录之一，即添加到 PYTHONPATH 中，如列表6.4所示。
- VISUAL STUDIO CODE 会根据当前工作目录下的环境配置文件（.ENV）而进行目录与模块搜索。

因此，在 PYCHARM 中我们可以通过修改目录等级，对目录做标记来手动使 IDE 搜索相关内容，如图6.1所示。在 VISUAL STUDIO CODE 中我们可以通过编辑如列表6.5所示的环境配置文件来添加相关内容。

由于上述两种方法都是在间接修改运行时的 PYTHONPATH，在只用 PYTHON 执行相关脚本时我们可以通过如列表6.4所示的语句执行等价操作。

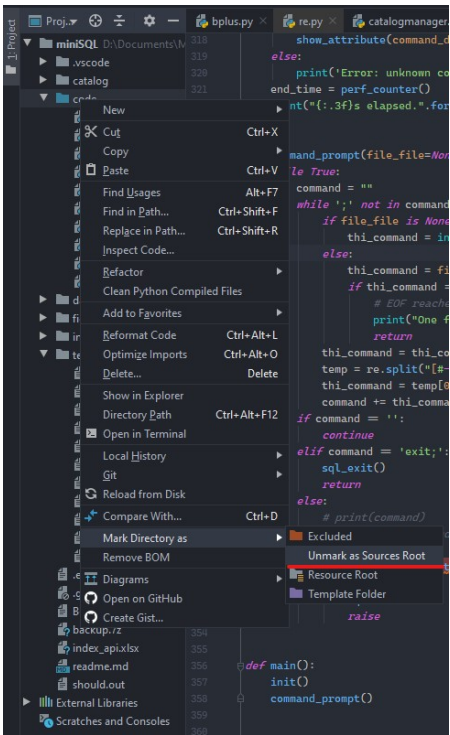


图 6.1: PyCharm 中的文件夹标记

```
1 D:\CONDAENV\ANACONDA\PYTHON.EXE "C:\PROGRAM FILES\JETBRAINS\PYCHARM
  ↳ 2019.3.4\PLUGINS\PYTHON\HELPERS\PYDEV\PYDEVCONSOLE.PY" --MODE=CLIENT --PORT=4243
2 import sys; print('PYTHON %s ON %s' % (SYS.VERSION, SYS.PLATFORM))
3 SYS.PATH.EXTEND(['D:\\DOCUMENTS\\MATERIALS\\DENDE\\PROJ\\DATABASE\\MINISQL',
  ↳ 'D:\\DOCUMENTS\\MATERIALS\\DENDE\\PROJ\\DATABASE\\MINISQL\\CODE',
  ↳ 'D:/DOCUMENTS/MATERIALS/DENDE/PROJ/DATABASE/MINISQL'])
```

Listing 6.4: PyCharm 中的搜索目录自动修改

```
1 PYTHONPATH=CODE
```

Listing 6.5: PyCharm 中的搜索目录自动修改

6.2 使用 pickle 储存二进制形式的索引超出递归层数限制

6.2.1 问题描述

树是一个递归实现的数据结构，PYTHON 提供的二进制储存包 PICKLE 需要通过递归调用树中的内容来储存其所包含的信息。因此树高过大会导致超出 PYTHON 递归限制，如图6.2所示。

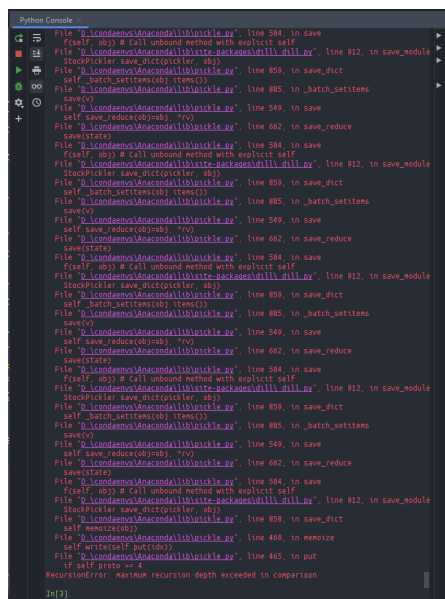


图 6.2: 超出递归限制

6.2.2 解决方法

当然我们可以使用列表6.6所示的代码修改 PYTHON 对于递归深度的限制。但修改词限制之后系统堆栈同样有机会溢出，在这种情况下系统会直接杀死我们的 PYTHON 进程，如图6.3所示。

```

insert into student2 values(1080102102,'name2102',68),
0.045s elapsed
insert into student2 values(1080102103,'name2103',89.5),
0.043s elapsed
insert into student2 values(1080102104,'name2104',93),
0.043s elapsed
insert into student2 values(1080102105,'name2105',63.5),
0.046s elapsed
insert into student2 values(1080102106,'name2106',50.5),
0.044s elapsed
insert into student2 values(1080102107,'name2107',85.5),
0.006s elapsed
insert into student2 values(1080102108,'name2108',88.5),
0.041s elapsed
insert into student2 values(1080102109,'name2109',75.5),
0.042s elapsed
insert into student2 values(1080102110,'name2110',99.5),
0.051s elapsed
insert into student2 values(1080102111,'name2111',61.5),
0.042s elapsed
insert into student2 values(1080102112,'name2112',61),
0.044s elapsed
insert into student2 values(1080102113,'name2113',65.5),
0.044s elapsed
insert into student2 values(1080102114,'name2114',86.5),
0.001s elapsed
insert into student2 values(1080102115,'name2115',66.5),
0.042s elapsed
insert into student2 values(1080102116,'name2116',64.5),
0.043s elapsed
insert into student2 values(1080102117,'name2117',85.5),
0.042s elapsed
insert into student2 values(1080102118,'name2118',77.5),
0.038s elapsed
insert into student2 values(1080102119,'name2119',97.5),
0.039s elapsed
insert into student2 values(1080102120,'name2120',82.5),
0.039s elapsed
insert into student2 values(1080102121,'name2121',84.5),
0.042s elapsed
insert into student2 values(1080102122,'name2122',61),
0.066s elapsed
insert into student2 values(1080102123,'name2123',72.5),
0.045s elapsed
insert into student2 values(1080102124,'name2124',83),
0.039s elapsed
insert into student2 values(1080102125,'name2125',78.5),
0.041s elapsed
insert into student2 values(1080102126,'name2126',72.5),
0.045s elapsed
insert into student2 values(1080102127,'name2127',59),
0.044s elapsed
insert into student2 values(1080102128,'name2128',79.5),
0.044s elapsed
insert into student2 values(1080102129,'name2129',75.5),
0.065s elapsed
insert into student2 values(1080102130,'name2130',64.5),
0.046s elapsed
insert into student2 values(1080102131,'name2131',72.5),
Process finished with exit code -1073741571 (0xC00000FD)

```

图 6.3: 系统杀死当前进程

```

1 import sys
2 SYS.SETRECURSIONLIMIT(10000000)

```

Listing 6.6: 修改递归限制

最终，我们通过修改 B+ 树的最大子树数量避免了这一问题，在调用 `CREATE_INDEX` 方法时直接设置 `M` 值，如列表6.7所示。

```

1 def CREATE_INDEX(IND, DATA_LIST, CMP=DUMMY_CMP, IS_PRIMARY=FALSE):
2     """
3     :param ind: the id of the index to be saved to file
4     :param data_list: the data, as list, to create index on
5     :param cmp: the comparator of the index, defaults to operator<
6     :param is_primary: whether we're dealing with primary key, using sorted list
7     :return: index of the newly created table
8     """
9     if IS_PRIMARY:
10         T = SortedList()

```



```
11     else:
12         # TODO: dynamically compute the M value of the B+ tree
13         # TODO: what if you're out of memory
14         T = TREE(M=50, CMP=cmp)
15     for INDEX, DATA in ENUMERATE(DATA_LIST):
16         # TODO: what happens if you get an error from the B+ tree
17         T.INSERT(DATA, INDEX) # insert data as key and line number as value
```

Listing 6.7: 调整 M 值, 降低深度

第七章 总结

在本实验中，我们从零实现了 B+ 树模块。借助 PYTHON 的动态类型语言特性，我们的 B+ 树可以有效对各种 PYTHON 支持的数据类型进行操作。通过进行单元测试，我们在 MINISQL 项目整体进行的前期就保证了 B+ 树模块的正确性，为后续调试整合过程带来了极大方便。

接着，我们实现了 B+ 树中的索引相关操作。并封装成了一个易用的模块，使得 API 可以方便的调用。

此次实验中我们深刻感受到了小组合作与提前定义接口的重要性，并且在 B+ 树具体实现过程中，我们对单元测试的作用有了更进一步的了解。

我们通过设计 GUI 进一步了解了程序设计过程中与人类交互方式的重要性。我们发现图形界面设计与命令行程序各有优劣，经过研究发现很多程序都采用命令行程序提供核心功能，接着通过图形界面将这些核心功能包裹。例如 VISUAL STUDIO 的 CL.EXE 编译器

第二部分

附录

第一章 接口说明

```
1  def CREATE_INDEX(IND, DATA_LIST, CMP=DUMMY_CMP, IS_PRIMARY=FALSE):
2      """
3      :param ind: the id of the index to be saved to file
4      :param data_list: the data, as list, to create index on
5      :param cmp: the comparator of the index, defaults to operator<
6      :param is_primary: whether we're dealing with primary key, using sorted list
7      :return: index of the newly created table
8      """
9
10
11  def DROP_INDEX(IND):
12      """
13      :param ind: the id of the index
14      :return: currently nothing is returned
15      """
16
17
18  def INSERT(IND, KEY, VALUE, IS_REPLACE=FALSE):
19      """
20      :param ind: the id of the index
21      :param key: the key to insert into the index
22      :param value: the value of the B+ tree, probably the line number of the inserted
    ↪ item
23      :param is_replace: whether we should replace on duplication
24      :raise KeyException: duplication
25      """
26
27
28  def SEARCH(IND, KEY, IS_GREATER=NONE, IS_CURRENT=NONE, IS_RANGE=FALSE,
    ↪ IS_NOT_EQUAL=FALSE):
29      """
30      A thin wrapper around _operate
```

```

31      :param is_current: whether we want a single value range search with current node
32      :param is_greater: whether we want a single value range search of greater than
33      :param ind: the id of the index to be deleted on
34      :param key: the key/keys to be deleted (single or range)
35      :param is_range: are we searching in range?
36      :return: currently nothing is returned
37      """
38
39
40  def DELETE(IND, KEY, IS_GREATER=NONE, IS_CURRENT=NONE, IS_RANGE=FALSE,
41  ↪ IS_NOT_EQUAL=FALSE):
42      """
43      A thin wrapper around _operate
44      :param is_current: whether we want a single value range search with current node
45      :param is_greater: whether we want a single value range search of greater than
46      :param ind: the id of the index to be searched on
47      :param key: the key/keys to be searched (single or range)
48      :return: currently nothing is returned
49      """
50
51  def GET_VALUES(IND):
52      """
53      get every value from the bplus tree for printing all information fast enough
54      :param ind: id of the index whose value is to be updated
55      """
56
57
58  def UPDATE_VALUES(IND, VALUES):
59      """
60      updates information about an index
61      :param ind: id of the index whose value is to be updated
62      :param values: the new values to be set to the index
63      :return:
64      """

```

Listing A.1: 接口说明

第二章 插图，表格与列表

插图

2.1	GUI REQUIREMENTS	10
2.2	EDITOR REQUIREMENTS	10
2.3	ERROR MESSAGE REQUIREMENTS	11
4.1	线性记录储存方式	13
4.2	单值操作	14
4.3	范围操作	15
4.4	B+ 树的类图与类间关系	18
4.5	异常类图与类间关系	19
4.6	图形界面类图	20
5.1	FIXPARENT	23
5.2	FIXSIBLING	24
5.3	排序数组具体实现	29
6.1	PYCHARM 中的文件夹标记	37
6.2	超出递归限制	38
6.3	系统杀死当前进程	39

List of Listings

2.1	CREATE INDEX SYNTAX AND EXAMPLE	8
2.2	DROP INDEX SYNTAX AND EXAMPLE	8
2.3	SELECT SYNTAX AND EXAMPLE	8
2.4	INSERT SYNTAX AND EXAMPLE	9
2.5	DELETE FROM SYNTAX AND EXAMPLE	9
2.6	QUIT MINISQL SYNTAX	9
5.1	异常类型的具体实现	29
5.2	GUI 具体实现	34
6.1	可能的文件目录	35
6.2	可能的文件目录	36
6.3	无法引用同一文件下的模块	36
6.4	PYCHARM 中的搜索目录自动修改	37
6.5	PYCHARM 中的搜索目录自动修改	37
6.6	修改递归限制	39
6.7	调整 M 值, 降低深度	40
A.1	接口说明	44