

## 数据库调优课程研究报告之索引研究

|  |  |
| --- | --- |
| 院 系： | 软件学院 |
| 组 长 学 号： | 19212010054 |
| 组 长： | 吴俊 |
| 小 组 成 员： | 吴俊 |
| 完 成 日 期： | 2019年12 月 9 日 |

## 目录

[数据库调优课程研究报告之索引研究 1](#_Toc2773)

[目录 1](#_Toc15158)

[图目录 2](#_Toc17418)

[表目录 3](#_Toc7897)

[一、 系统概述 4](#_Toc9466)

[二、 系统平台 5](#_Toc11162)

[三、 系统数据库设计 6](#_Toc30372)

[四、 系统索引算法 11](#_Toc15094)

[4.1 线性哈希(Linear Hash) 11](#_Toc30498)

[4.2 扩展哈希(Extensible Hash) 13](#_Toc31465)

[4.3 B+树索引(B+ Tree Index) 14](#_Toc2973)

[五、 实验设置 18](#_Toc30354)

[六、 实验过程 19](#_Toc19187)

[七、 结论 25](#_Toc2243)

## 图目录

[图 三-1 总体E-R图 6](#_Toc16153)

[图 三-2 图书E-R图 7](#_Toc24646)

[图 三-3 学生E-R图 7](#_Toc22625)

[图 三-4 教师E-R图 8](#_Toc27553)

[图 四-1 线性哈希算法数据插入流程图 11](#_Toc27140)

[图 四-2 线性哈希算法数据查询流程图 12](#_Toc32298)

[图 四-3 线性哈希算法数据删除流程图 12](#_Toc30730)

[图 四-4 扩展哈希算法数据插入流程图 13](#_Toc6727)

[图 四-5 扩展哈希算法数据查询流程图 14](#_Toc26413)

[图 四-6 扩展哈希算法数据删除流程图 14](#_Toc4770)

[图 四-7 B+树数据插入过程流程图 15](#_Toc10381)

[图 四-8 B+树数据查询过程流程图 16](#_Toc13183)

[图 四-9 B+树数据删除过程流程图 16](#_Toc19877)

[图 六-1 系统首页 19](#_Toc5709)

[图 六-2 实验一首页 19](#_Toc6270)

[图 六-3 实验一数据插入图 20](#_Toc21370)

[图 六-4 实验一数据查询图 21](#_Toc23586)

[图 六-5 实验一数据查询结果 21](#_Toc1676)

[图 六-6 图书csv文件记录图 22](#_Toc16724)

[图 六-7 实验一数据删除图 22](#_Toc22638)

[图 六-8 实验二首页 22](#_Toc17278)

[图 六-9 实验二数据插入图 23](#_Toc1031)

[图 六-10 实验三首页 23](#_Toc31211)

[图 六-11 实验三等值查询图 23](#_Toc2321)

[图 六-12 实验三范围查询图 24](#_Toc310)

## 表目录

[表 三-1 图书信息表 8](#_Toc12383)

[表 三-2 学生信息表 9](#_Toc15486)

[表 三-3 教师信息表 9](#_Toc13979)

[表 三-4 图书借阅表 9](#_Toc25442)

[表 三-5 图书归还表 10](#_Toc14049)

# 系统概述

本文设计实现了一个比较不同索引算法性能特点的系统。该系统是由Java开发的Spring Boot Web应用程序，其中线性哈希算法(Linear Hash)，扩展哈希算法(Extensible Hash)以及B+树索引算法(B+Tree Index)均由Java实现，系统实现了每种索引算法的增加、删除和查询数据的方法；系统模拟生成的数据均以设计的表名作为文件名，以csv格式保存在文件中，每个文件代表一张数据库表，文件中的每一行代表一条数据记录，每条数据记录所在的行代表其在磁盘上的位置，以这种方式模拟一个数据库；系统以图表的形式可视化实验过程中不同索引算法在不同实验条件下的性能，不仅可以看出不同索引算法随实验条件改变而变化的整体趋势，而且可以对比索引算法之间的性能差异。

本文设计了三种不同的实验方案，分别是：探索索引种类与数据量的关系，探索索引种类与索引字段类型的关系以及探索不同索引种类的查询范围。在探索索引种类与数据量的关系实验中，系统保持索引字段不变，观察随着插入或删除的数据量的变化，不同索引算法插入，查询和删除数据的用时变化；在探索索引种类与索引字段类型的关系实验中，系统保持每次插入的数据量不变，观察不同索引算法在索引字段变化时的表现；在探索不同索引种类的查询范围实验中，系统保持索引字段和数据量不变，改变查询要求，观察不同索引算法能否完成任务。

# 系统平台

系统开发语言：Java

系统开发工具：Jetbrain IntelliJ Idea

系统开发框架：Spring Boot，Echarts

系统开发环境：

Windows 10 专业版 Intel(R) Core(TM)i5-8250U CPU @1.60GHz 1.80GHz

16.0GB内存，64位操作系统

系统运行环境：

Windows 10 专业版 Intel(R) Core(TM)i5-8250U CPU @1.60GHz 1.80GHz

16.0GB内存，64位操作系统

# 系统数据库设计

本系统设计的关系数据库应用场景是大学图书馆的图书管理系统。系统共设计了五张数据表，分别是图书信息表(Book)，学生信息表(Student)，教师信息表(Teacher)，图书借阅表(BookBorrow)以及图书归还表(BookReturn)。为了方便比较不同索引算法的性能，系统并未将这些表设计到数据库中，而是保存为多个csv文件，一张表对应一个文件名与表名相同的csv文件，数据保存在文件中，文件中的一行代表一条数据记录，数据记录所在的行代表其磁盘地址，以这种方式模拟一个图书管理系统的数据库。

如下为该数据库的E-R图：

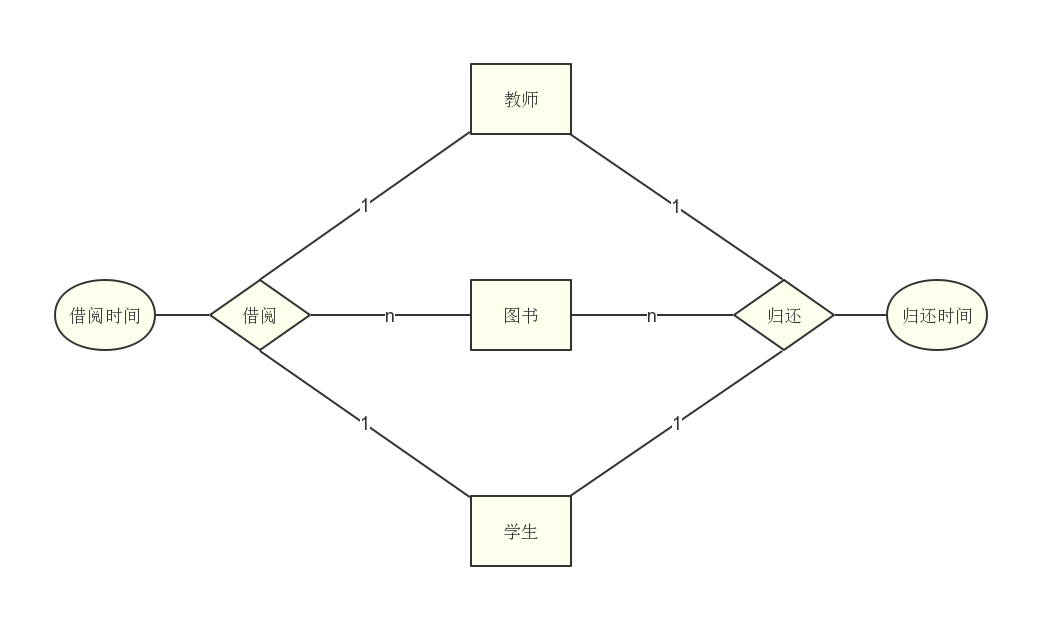


图 三-1总体E-R图

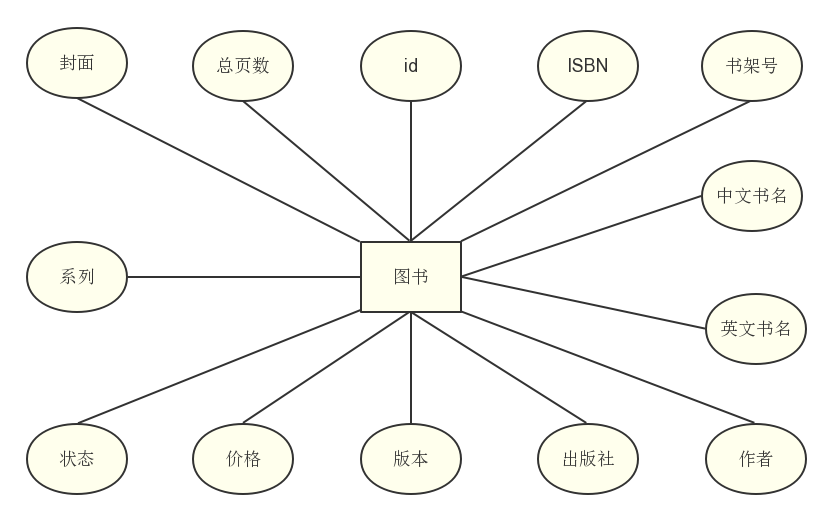


图 三-2图书E-R图

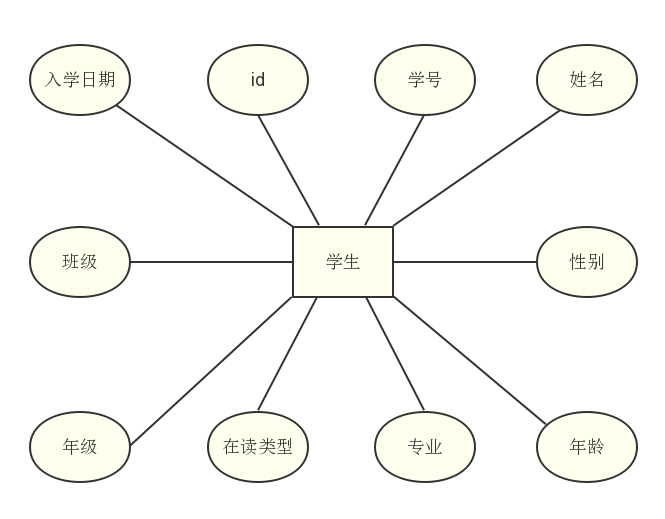


图 三-3学生E-R图

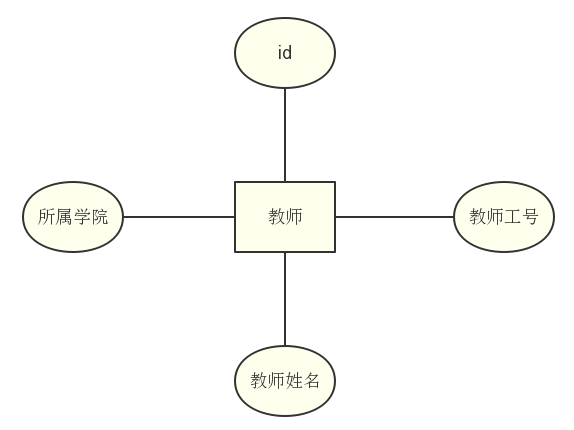


图 三-4教师E-R图

数据表设计如下：

表 三-1图书信息表

|  |  |  |  |
| --- | --- | --- | --- |
| 字段 | 类型 | 约束 | 描述 |
| id | int | primary key | 主键，记录唯一标识 |
| book\_isbn | varchar(20) | not null | 图书ISBN编号 |
| book\_bookcase\_no | varchar(20) | not null | 图书书架号 |
| book\_name\_cn | varchar(30) | not null | 中文图书名 |
| book\_name\_en | varchar(30) | not null | 英文图书名 |
| book\_author | varchar(20) | not null | 图书作者 |
| book\_press | varchar(20) | not null | 出版社 |
| book\_version | int | not null | 图书版本 |
| book\_price | float | not null | 图书单价 |
| book\_series | varchar(30) |  | 图书所属系列 |
| book\_cover | varchar(30) |  | 图书封面 |
| book\_total\_page | int | not null | 图书总页数 |
| book\_status | int | not null | 图书状态，0表示未借出，1表示已借出 |

表 三-2 学生信息表

|  |  |  |  |
| --- | --- | --- | --- |
| 字段 | 类型 | 约束 | 描述 |
| id | int | Primary key | 主键，记录唯一标识 |
| stu\_no | varchar(20) | not null | 学号 |
| stu\_name | varchar(20) | not null | 姓名 |
| stu\_gender | int |  | 性别，0表示未知，1表示男，2表示女 |
| stu\_age | int |  | 年龄 |
| stu\_major | varchar(20) |  | 专业 |
| stu\_type | int |  | 学生类型，0表示本科生，1表示硕士，2表示博士 |
| stu\_grade | int |  | 年级 |
| stu\_class | int |  | 班级 |
| stu\_admission\_date | date |  | 入学日期 |

表 三-3教师信息表

|  |  |  |  |
| --- | --- | --- | --- |
| 字段 | 类型 | 约束 | 描述 |
| id | int | primary key | 主键，记录唯一标识 |
| tea\_no | varchar(20) | not null | 教师工号 |
| tea\_name | varchar(20) | not null | 教师姓名 |
| tea\_college | varchar(30) | not null | 教师所属学院 |

表 三-4图书借阅表

|  |  |  |  |
| --- | --- | --- | --- |
| 字段 | 类型 | 约束 | 描述 |
| id | int | primary key | 主键，记录唯一标识 |
| person\_no | varchar(20) | not null | 人员编号，如学号或教师工号 |
| person\_type | int | not null | 人员类型，0表示学生，1表示教师 |
| book\_id | int | not null | 图书记录id |
| borrow\_time | datetime | not null | 借阅时间 |

表 三-5图书归还表

|  |  |  |  |
| --- | --- | --- | --- |
| 字段 | 类型 | 约束 | 描述 |
| id | int | primary key | 主键，记录唯一标识 |
| person\_no | varchar(20) | not null | 人员编号，如学号或教师工号 |
| person\_type | int | not null | 人员类型，0表示学生，1表示教师 |
| book\_id | int | not null | 图书记录id |
| return\_time | datetime | not null | 归还时间 |

# 系统索引算法

本文主要研究了三种数据库索引算法，分别为线性哈希算法，扩展哈希算法以及B+树索引算法。其中B+树索引算法是目前多种数据库中使用的最多的算法之一。前两种算法均属于哈希算法的范畴，并且是动态哈希算法，而B+树算法是由B树算法改进而来，属于树数据结构算法。在本节中，本文主要介绍三种索引算法的插入，查询和删除数据方法的实现，并用流程图的方式显示算法的原理。

## 4.1 线性哈希(Linear Hash)

线性哈希算法也叫线性散列算法，是由Witold Litwin发明并由Paul Larson推广的一种动态哈希算法。在静态哈希算法中，数据桶的个数是固定的，当发生冲突时，要么利用冲突解决策略化解冲突，要么使用开散列，将冲突的数据插入到对应数据桶的后续溢出桶中，缺点显而易见，当哈希表中的数据越来越多时，每次查询都近似于列表查找。动态哈希算法与之不同之处在于，其哈希算法是随着数据量的改变而动态变化的，并且数据桶的个数也不是固定的，而是可以动态增加的。

下面将结合流程图具体介绍线性哈希算法的数据插入，查询和删除过程。

首先是数据插入过程。

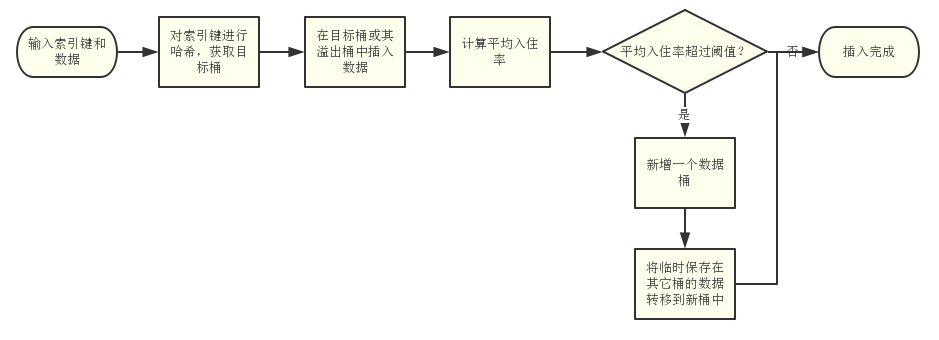


图 四-1线性哈希算法数据插入流程图

在输入索引键和数据后，首先对索引键进行哈希。这里需要注意的是，在将索引键哈希成一个整数后，需要根据目前桶的数量进行取余，取余后的值就是目标桶的下标，然后就可以获得目标桶。

在获得目标桶后，需要找到数据插入的位置，首先判断目标桶是否已经装满，如果没装满则找到一个空位置插入数据，否则在其溢出桶中寻找空位置，然后插入数据。

插入数据后，需要计算平均入住率来判断是否需要新增一个桶。平均入住率的计算方法是：平均入住率=总数据量/(桶的数量 x 桶的容量)。当平均入住率超过设置的阈值后，就需要新增一个桶，然后计算本来应该插入到该新桶的数据被临时保存在哪个桶中，然后遍历该桶，重新哈希该桶中的每个元素，将哈希后应该插入到新桶里的数据转移到新桶里来。插入完毕。

接下来是数据查询过程。

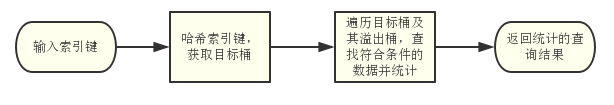


图 四-2线性哈希算法数据查询流程图

在输入索引键后，用哈希算法哈希索引键，获取目标桶的下标，进而获取目标桶。遍历目标桶及其所有溢出桶，统计符合条件的数据，在统计完成后，将统计的数据返回。

最后是数据删除过程。

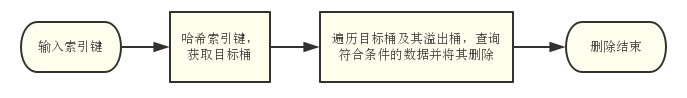


图 四-3线性哈希算法数据删除流程图

数据删除过程和查询的过程类似，只不过在找到符合条件的数据后，会将其删除。

## 4.2 扩展哈希(Extensible Hash)

扩展哈希算法与线性哈希算法类似，但也有区别。相似之处在于两种算法都用数据桶来保存数据，并且哈希函数都会变化，不同之处在于线性哈希算法的哈希函数是随着桶的数量变化而变化，而扩展哈希算法的哈希函数是随着目录的扩展而变化，并且扩展哈希中不存在溢出桶。在扩展哈希算法中，引入了目录的概念，目录中保存了数据桶的地址，不同目录中可能保存着同一个数据桶的地址。目录的每次扩展都是翻倍的，即从一个变为两个，两个变为四个，以此类推。目录的翻倍与桶的分裂都与两个概念有关，即目录的全局深度以及桶的局部深度。因为目录只有一个，因此全局深度也是唯一的，而数据桶的数量可以有很多，因此每一个桶都有其局部深度。全局深度与局部深度的关系决定了是否对目录进行扩展。

下面结合流程图来具体介绍扩展哈希的数据插入，插叙和删除过程。

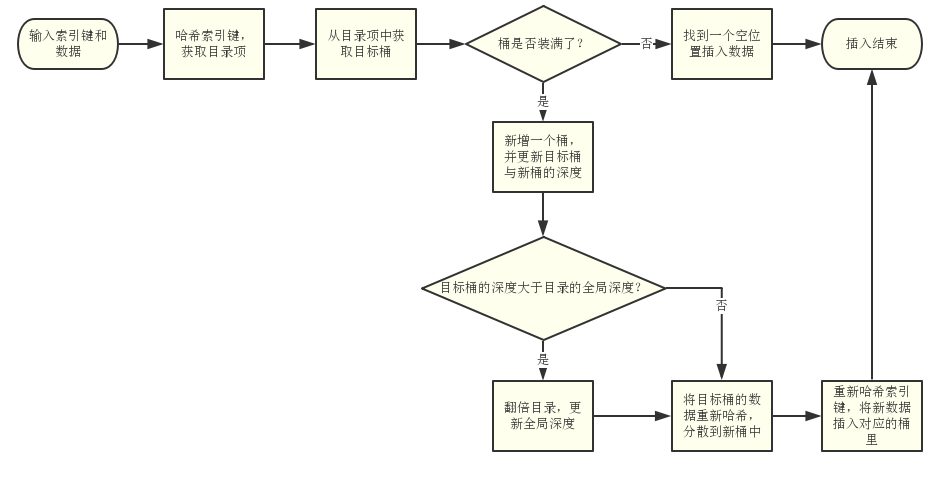


图 四-4扩展哈希算法数据插入流程图

在输入索引键和数据后，根据当前目录的全局深度对索引键进行哈希，获取目录项的下标，然后从目录项中获取目标桶的地址，进而获取目标桶。

获取目标桶后，判断桶是否已经装满，如果没有装满，在找到一个空位置插入数据，否则新建一个桶，将目标桶的局部深度加一，并赋值给新桶。然后判断更新后的桶的局部深度与目录的全局深度的大小关系，如果桶的局部深度大于目录的全局深度，则需要对目录进行翻倍。然后遍历目标桶中的数据，重新哈希，将数据分散到目标桶与新桶中，最后将新数据插入对应的数据桶。插入结束。

接下来是数据查询过程。



图 四-5扩展哈希算法数据查询流程图

在输入索引键后，对其进行哈希，获取目录项，然后从目录项中获取目标桶的地址，进而获取目标桶，最后遍历目标桶，统计符合条件的数据，遍历完成后返回统计的结果。

最后是数据删除的过程。

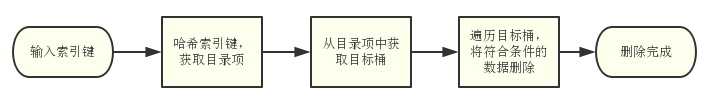


图 四-6扩展哈希算法数据删除流程图

扩展哈希算法的数据删除过程与查询类似，只不过该遍历目标桶时，如果找到了符合条件的数据，则将其删除。这里的删除过程是不完整的，完整的数据删除过程还应包含目录的缩小与桶的合并，本文考虑到目录的缩小与桶的合并需要消耗大量的计算资源，不合并的话只会对查询性能有较小的影响，因此省略了该过程。

## 4.3 B+树索引(B+ Tree Index)

B+树索引算法是由B树索引算法优化改进而来的，与B树的主要不同点在于：B+树的非叶子节点只保存键值信息，不再保存数据；数据记录都保存在叶子节点中；所有叶子节点之间都有一个链指针。B+树基于树数据结构，通常用于数据库和文件系统中。B+树的特点是能够保持数据稳定有序，其插入和修改具有较稳定的对数时间复杂度。B+树与一般二叉树的不同之处在于，B+树的元素是自底向上插入的。一颗m阶B+树的定义如下：

⑴每个节点至多有m-1个关键字和m个子女。

⑵除根节点外，每个节点至少有m/2个子女。

⑶叶子节点按照关键字大小从小到大顺序链接。

B+树中保存了最小的叶子节点的地址，又因为叶子节点是按关键字大小从小到大顺序链接的，因此很容易进行范围查询。

下面结合流程图来讲解B+树算法的数据插入，查询和删除过程。

首先是数据插入过程。

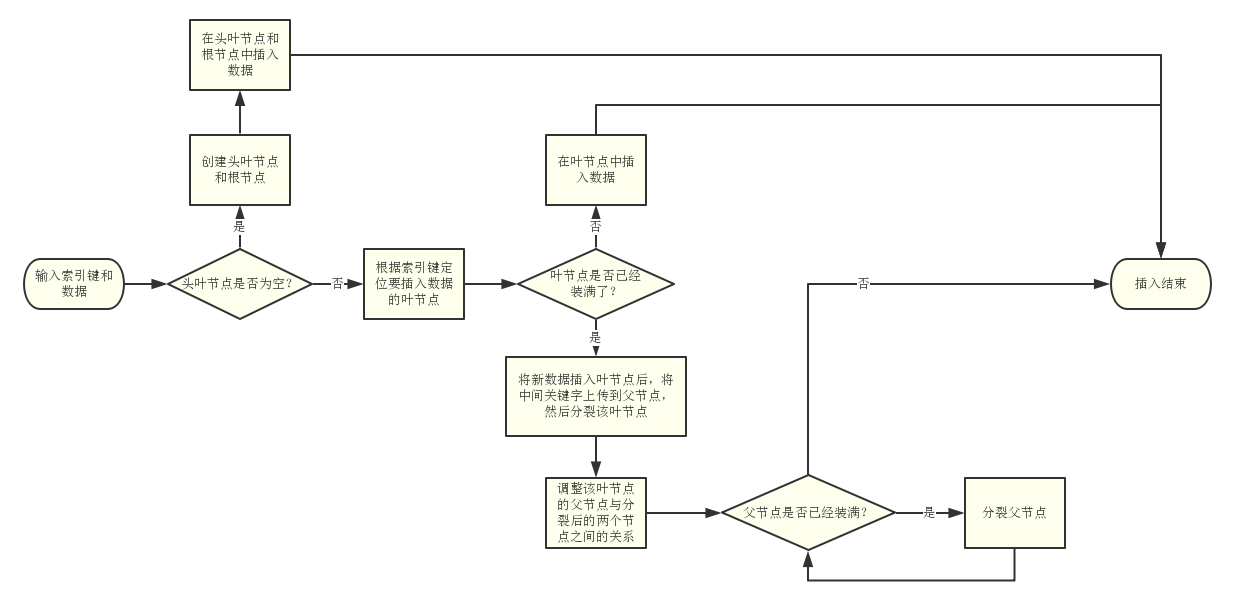


图 四-7 B+树数据插入过程流程图

在输入索引键和数据后，首先判断头叶节点是否是空的，如果是空的说明树也是空的，那么只需要创建头叶节点和根节点，然后把新数据插入到两个节点中就行了，否则需要根据输入的索引键大小关系找到适合插入数据的叶节点。

定位到目标叶节点后，判断该叶节点是否已经装满了，如果没有装满，则直接把新数据插入该叶节点，否则需要对该叶节点进行分裂。

在对叶节点进行分裂之前，先把新数据插入到叶节点中，排序，然后将位于中间的关键字上传到父节点，然后以此中间关键字为边界进行分裂，将中间关键字左边的关键字划分到左节点，将中间关键字右边（包含中间关键字）的关键字划分到右节点。

划分完成后，需要调整原叶节点的父节点与新分裂的左右节点之间的关系，包括将左右节点的父节点指向原叶节点的父节点，以及将左节点的下一个节点指向右节点等等。

完成关系调整后，需要判断父节点是否因为子节点上传了关键字而超载了，如果没有超载，则插入完毕，否则需要递归分裂父节点，直到树结构满足要求为止。

接下来是B+树的数据查询过程。

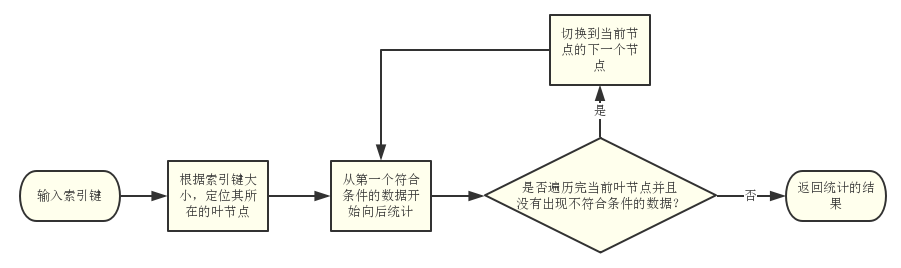


图 四-8 B+树数据查询过程流程图

在输入索引键后，首先根据索引键的大小，从头叶节点开始，找到第一个最大关键字比输入的索引键大的叶节点，因为索引键最有可能第一次出现在该叶节点中。从该叶节点的第一个关键字开始向后搜索索引键的位置，如果是等值查询，则找到第一个相等的关键字后，记录，顺序向后比较，如果遍历完当前叶节点，则进入下一个叶节点，直到出现不相等的关键字为止，返回记录的结果；如果是范围查询，同样在找到第一个比输入的最小关键字大的关键字开始，顺序向后比较并记录，如果遍历完当前叶节点，则进入下一个叶节点，直到关键字的大小比输入的最大关键字还大时结束搜索，返回记录的结果。

最后时B+树数据删除的过程。

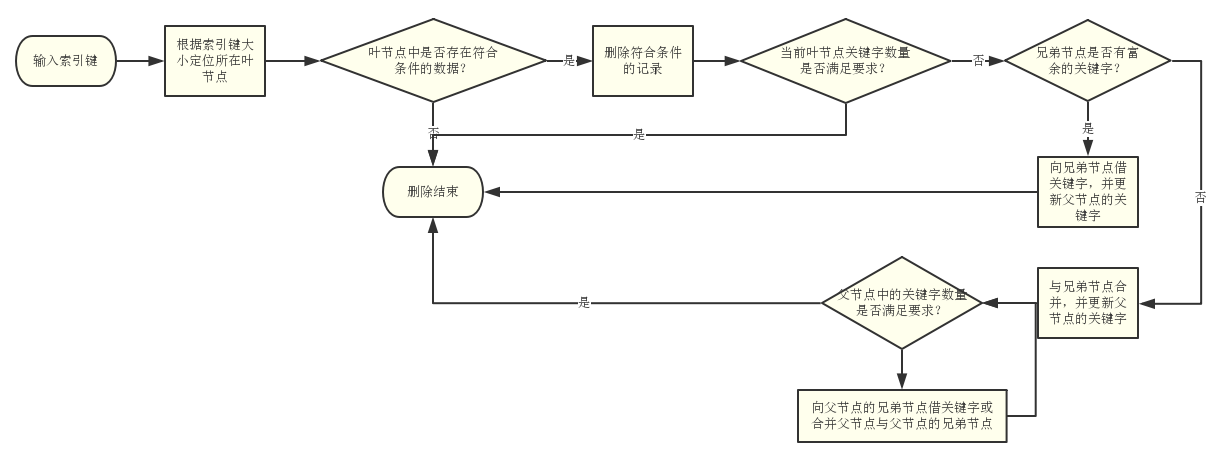


图 四-9 B+树数据删除过程流程图

在输入索引键后，首先找到索引键可能会出现的第一个叶节点，在叶节点中搜索是否存在符合要求的数据，如果没有找到符合要求的数据，则删除结束。如果找到了符合要求的数据，则将其删除，然后判断当前叶节点关键字的数量是否满足最低要求，如果满足，则删除结束，否则需要进一步操作。

如果删除数据后叶节点的关键字数量低于最低要求，则先查看其兄弟节点中是否存在富余的关键字，有富余的关键字的意思是即使从该节点中转移一部分数据，该节点的关键字数量仍然满足最低要求，如果存在富余的关键字，则向其兄弟节点借数据使当前节点达到要求，并更新父节点的关键字。如果兄弟节点没有富余的关键字，则可以将当前节点与其兄弟节点进行合并，并更新父节点的关键字。

在更新父节点的关键字后，父节点的关键字数量可能小于最低要求，这时的做法与叶节点类似，要么从其兄弟节点中借关键字，要么与其兄弟节点合并，然后向上递归，直到树结构满足要求。

# 实验设置

本文设计了三个实验来对比不同索引算法的性能，分别是：⑴探索索引种类与数据量的关系；⑵探索索引种类与索引字段类型的关系；⑶探索不同索引种类的查询范围。在本实验中，索引算法的性能包括索引执行单次操作的用时，以及索引算法能够完成的任务范围。

在探索索引种类与数据量的关系实验中，本文设计了三种查询，分别是⑴向索引表中插入一定量的数据；⑵查找书名为“图书10000”的数据记录；⑶在索引表中删除一定量的数据。因为我们探索的是与数据量的关系，因此我们保持索引字段始终为“图书名”。本文系统首先在三种索引表中初始化一千万条相同的数据，然后通过每次插入不同量级的数据，观察索引性能的整体变化趋势，并且对比不同索引种类之间变化趋势的不同。在每次插入数据后，系统会从三种索引中查询书名为“图书10000”的记录，并显示三种索引的查询用时以及查询结果。最后，系统通过每次删除不同量级的数据来对比不同索引算法的性能。

在探索索引种类与索引字段类型的关系实验中，本文设计了一种查询，就是简单地向索引表中插入数据。因为本实验探究与索引字段类型的关系，因此在实验中保证索引表内的数据以及每次插入的数据都是相同的。索引字段类型指的是某张表的某个字段是值稠密型的，还是值稀疏型的。值稠密型字段指的是该字段的值是枚举类型的，是可数的；值稀疏型字段与值稠密型字段正好相反，可以出现无限多种值。系统分别对图书表中的“图书名”字段和“图书系列”字段建立索引，其中“图书名”字段是值稀疏型的，“图书系列”字段是值稠密型的。系统通过每次插入相同的数据，观察索引整体在不同类型字段上的表现以及索引种类之间的差异。

在探索不同索引种类的查询范围实验中，系统保持索引表中的数据相同，索引字段均为图书表中的“图书价格”字段，设计了两种查询：⑴查询价格为50.0元的所有图书；⑵查询价格在50.0元到60.0元之间的所有图书。系统通过对比不同索引返回的结果，显示出不同索引种类能够完成的任务范围。

# 实验过程

因为本系统是通过web页面进行展示的，因此首先需要进入首页。

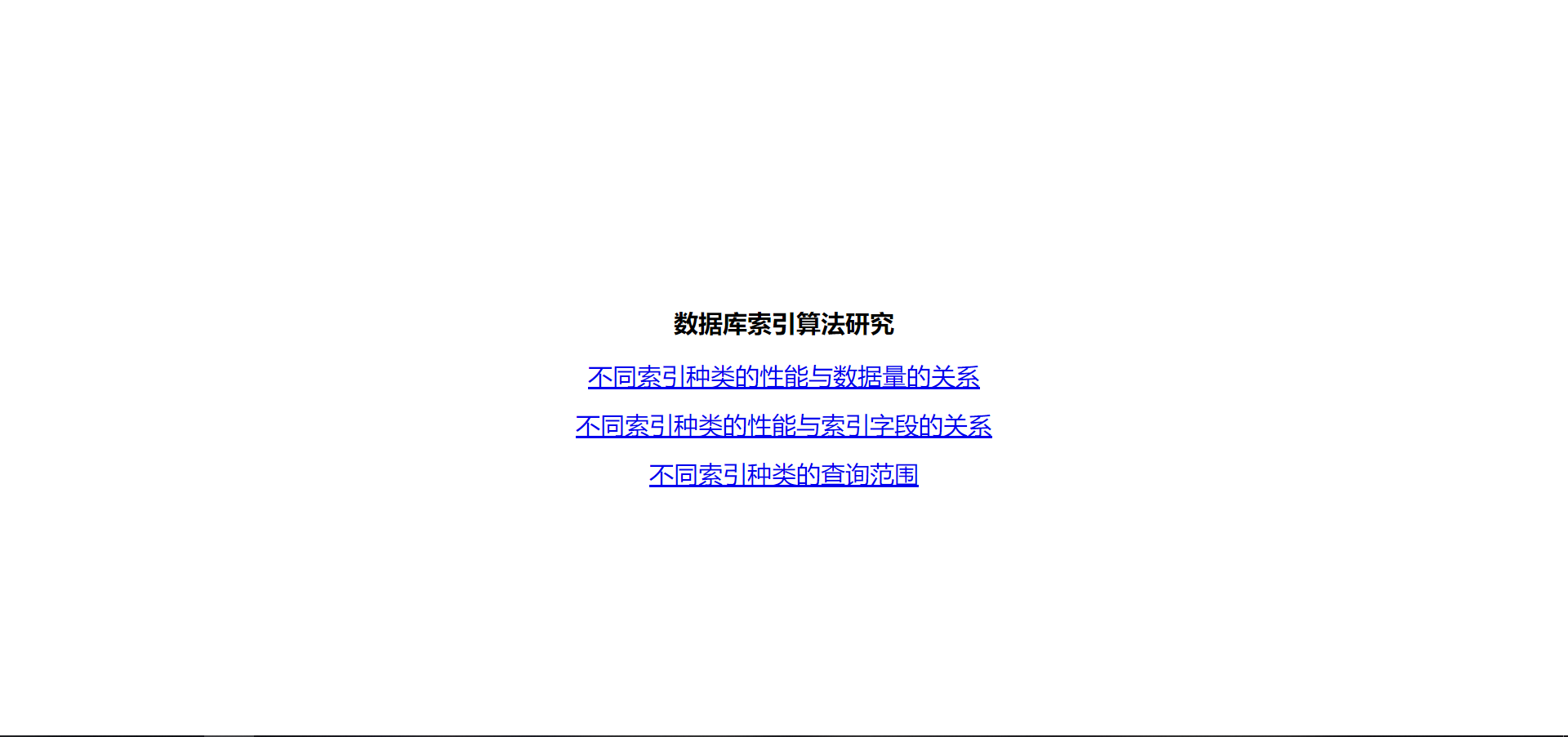


图 六-1 系统首页

这里列出了三个实验的链接，我们先进入第一个实验的页面。

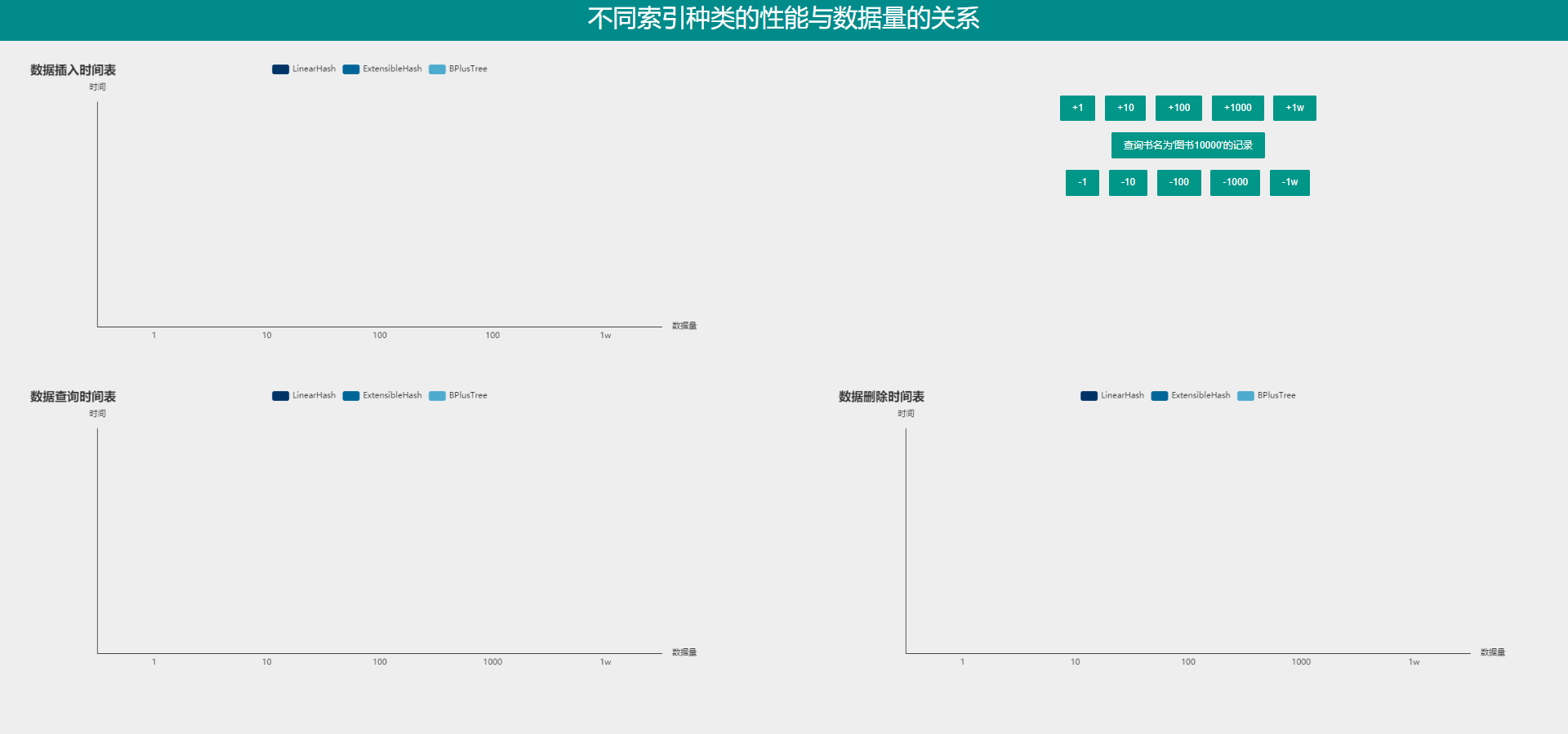


图 六-2 实验一首页

在实验一首页中可以看到三张表和一组按钮。三张表分别为数据插入时间表，数据查询时间表以及数据删除时间表，纵轴表示消耗的时间，计算方法是单次操作所消耗的纳秒取自然对数，横轴表示数据量。需要注意的是，三张索引表里都预先保存有一千万条相同的数据。在本实验中，实验步骤如下：

⑴点击“+1”按钮，表示向三张索引表中插入一条相同的数据；

⑵点击“查询书名为图书10000的记录”按钮，查询记录；

⑶点击“+10”按钮，表示向三张索引表中插入十条相同的数据；

⑷点击“查询书名为图书10000的记录”按钮，查询记录；

⑸点击“+100”按钮，表示向三张索引表中插入一百条相同的数据；

⑹点击“查询书名为图书10000的记录”按钮，查询记录；

⑺点击“+1000”按钮，表示向三张索引表中插入一千条相同的数据；

⑻点击“查询书名为图书10000的记录”按钮，查询记录；

⑼点击“+1w”按钮，表示向三张索引表中插入一万条相同的数据；

⑽点击“查询书名为图书10000的记录”按钮，查询记录；

⑾点击“-1”按钮，表示删除一条记录；

⑿点击“-10”按钮，表示删除十条记录；

⒀点击“-100”按钮，表示删除一百条记录；

⒁点击“-1000”按钮，表示删除一千条记录；

⒂点击“-1w”按钮，表示删除一万条记录；

在完成上述操作后，数据插入图表如下：

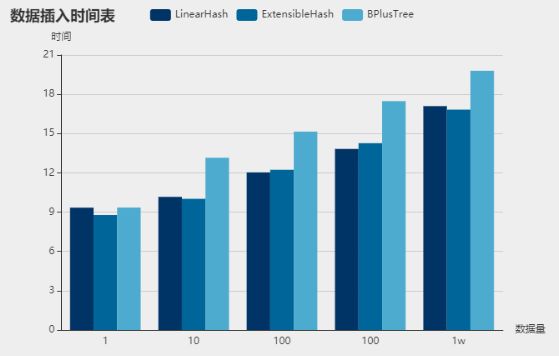


图 六-3 实验一数据插入图

从图六-3中可以看出，随着插入数据量的增大，三种索引的时间消耗都有上升的趋势，并且在三种索引之间，可以看到线性哈希和扩展哈希的性能差异不大，但是B+树索引相对于前两种哈希索引，其插入数据耗时更多，性能较低。

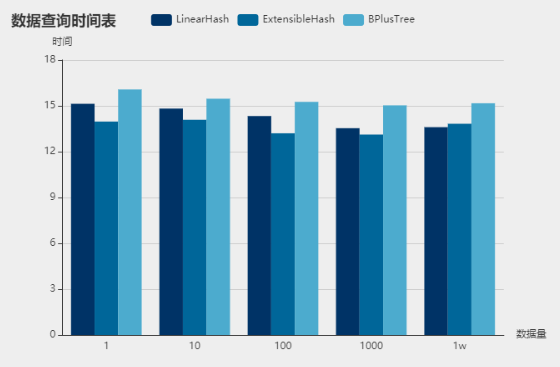


图 六-4 实验一数据查询图

从图六-4中可以看出，随着数据量的增大，三种索引的查询时间消耗变化不大，比较稳定。但在三种索引之间，可以看到前两种哈希索引的用时一直比B+树索引的用时低，这得益于哈希索引独特的机制。

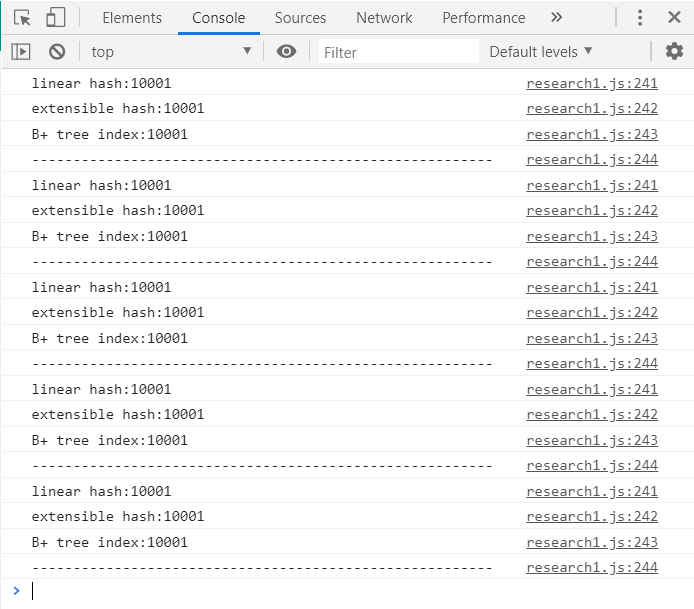


图 六-5 实验一数据查询结果

从图六-5中可以看到，在5次查询中，三种索引返回的结果都是一样的，10001是指数据记录在csv文件中的行数：

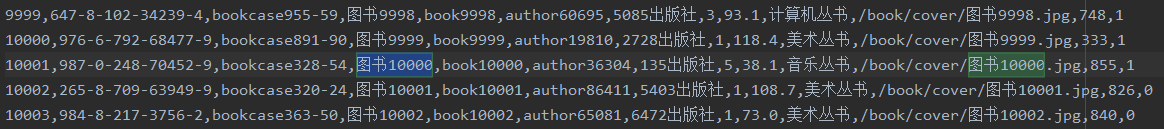


图 六-6 图书csv文件记录图

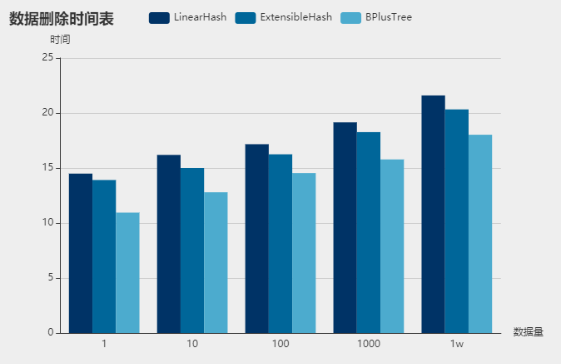


图 六-7 实验一数据删除图

从图六-7中可以看出，随着删除数据量的增大，三种索引的用时都在上升，并且在三种索引之间，线性哈希的用时始终最高，B+树用时始终最低，可见B+树索引的删除数据的性能优于前两种哈希索引。

下面进入实验二的页面：



图 六-8 实验二首页

实验二页面中有一张数据插入时间表以及一个“+100”按钮，实验步骤很简单，就是不断点击“+100”按钮，表示不断添加100条记录，观察图表上不同索引字段上索引的性能。在添加了2000条记录后，图表如下所示：

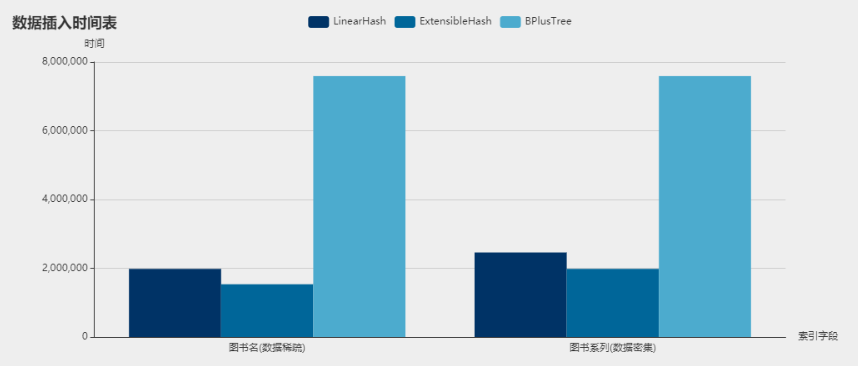


图 六-9 实验二数据插入图

从图六-9中可以看到，对于哈希索引，在插入数据时，建立在值密集型字段上的索引的用时要比建立在值稀疏型字段上的索引用时更多，对于B+树索引的影响不大。

下面进入实验三的页面。



图 六-10 实验三首页

在实验三中，系统在三种索引表中各初始化一千万条数据，并且三张索引表中的数据是相同的。实验步骤是：

⑴在等值查询输入框中输入50.0，点击查询按钮；

⑵在范围查询输入框中分别输入50.0和60.0，点击查询按钮。

操作完成后结果如图所示：

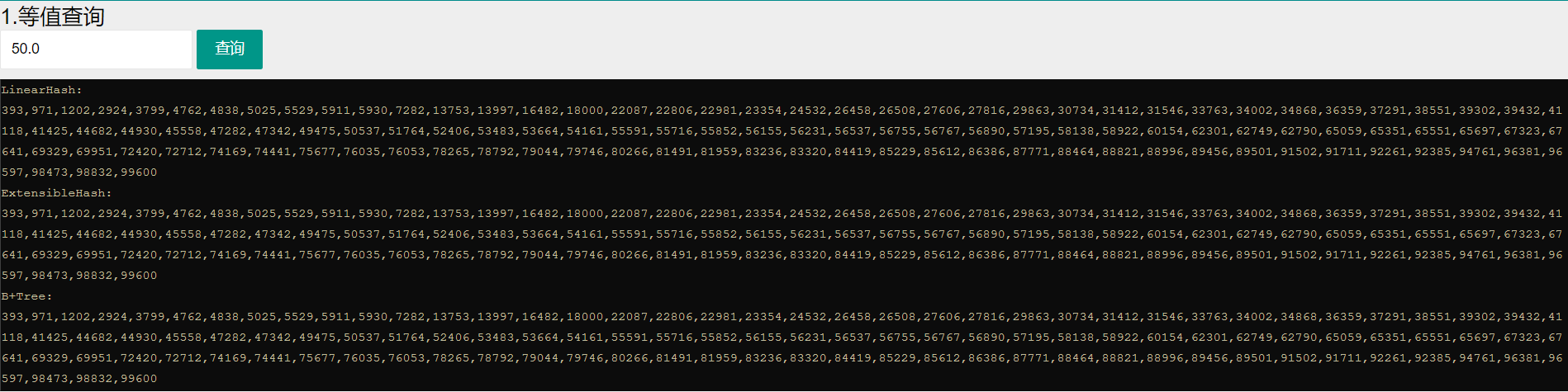


图 六-11 实验三等值查询图

从图六-11中可以看到，线性哈希，扩展哈希和B+树索引均能完成等值查询，查询结果完全相同。

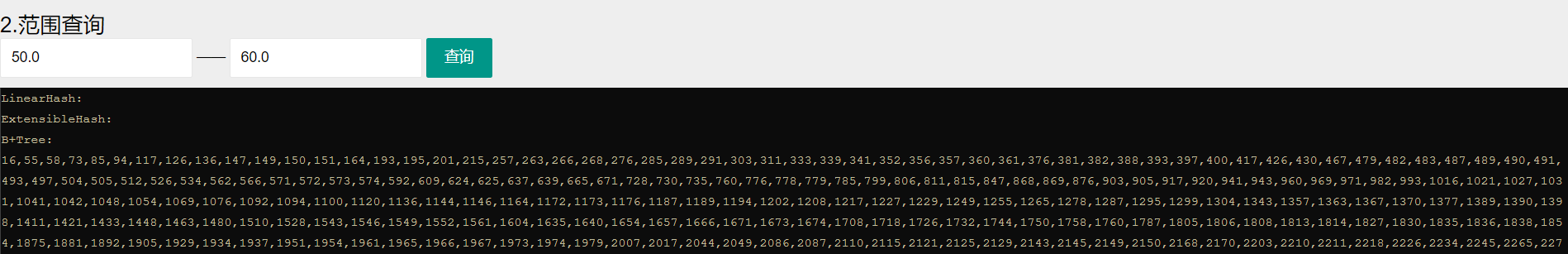


图 六-12 实验三范围查询图

从图六-12中可以看到，线性哈希和扩展哈希均无返回结果，而B+树索引有返回结果，这是由于哈希算法的性质导致的，因为在哈希索引中，第一步就是将索引键进行哈希，然而哈希算法不能保证哈希前后索引键的大小关系，也就是说，如果在哈希前索引键A“小于”索引键B，在分别哈希为HA和HB后，并不能保证HA一定小于HB。正是由于这种特性，哈希索引无法进行范围查询，而B+树索引因为没有修改索引键，并且在叶节点中是按照索引键的大小关系进行排序的，因此很容易进行范围查询。

# 结论

通过上述的三个实验，我们可以得出以下结论：

⑴在插入相同量的数据时，哈希索引的性能要优于B+树索引；

⑵在数据量相差不大的条件下进行等值查询时，哈希索引的查询性能始终优于B+树索引；

⑶在删除相同量的数据时，B+树索引的性能要优于哈希索引；

⑷在哈希索引中，在值稀疏型字段上建立索引是较好的选择；

⑸B+树索引支持范围查询，而哈希索引不支持范围查询，B+树索引的查询功能更强大。