BPLUS_SQL：一个基于 B+ 树存储整数的数据库

摘要

BPLUS_SQL 是一个基于磁盘的数据库引擎实现，使用 B+ 树数据结构来存储和管理整数。本项目作为 2024 年数据结构与算法挑战性课程设计的一部分开发完成。该系统具有完整的类 SQL 命令解析器、通过自定义分页系统实现的持久化存储、用于性能优化的 LRU 缓存，以及包含与红黑树性能比较在内的综合性测试基础设施。

项目由约 1,967 行 C++ 代码组成，实现了核心数据库操作（CREATE、INSERT、QUERY、ERASE、DESTROY）并具备完整的磁盘持久性，确保数据库在程序重启后依然存在。实现采用了现代 C++23 特性，并经过了广泛的测试以确保正确性和性能。

什么是 B+ 树?

B+ 树是一种自平衡的树状数据结构，它维护有序数据，并允许在对数时间内进行搜索、顺序访问、插入和删除操作。它是 B 树的一种扩展，广泛应用于数据库系统和文件系统中。

B+ 树的关键特性：

1. **多路树结构**：与二叉树不同，每个节点可以有多个键（在本实现中最多 128 个）和子节点（最多 129 个），这减少了树的高度并提高了缓存局部性。
2. **所有数据在叶节点**：内部节点仅存储用于引导搜索的路由键，而所有实际数据都驻留在叶节点中。这种设计使范围查询高效。
3. **叶节点链接**：叶节点通过 `next` 指针在顺序链中链接在一起，支持高效顺序遍历和范围查询。
4. **平衡结构**：所有叶节点处于相同深度，保证所有操作在最坏情况下具有 $O(\log n)$ 的性能。
5. **高扇出**：每个节点最多 128 个键、最少 64 个键（根节点除外），即使有数百万条记录，树也能保持浅层，从而最小化磁盘 IO 操作。

为什么数据库使用 B+ 树?

- 由于树高度浅，最小化磁盘访问次数
- 通过链接的叶节点实现高效范围查询
- 一次读取整个页面带来优异的缓存性能
- 可预测的性能特征

- 非常适合基于页的存储系统

我遇到了哪些问题，以及如何解决

在开发 BPLUS_SQL 的过程中，我遇到了几个需要仔细调试和算法思考的重大挑战：

将节点转化到磁盘中存储，而不是在内存中操作

如果节点存储在内存中，读取内存的操作（ `lw` ）要比分支操作（ `slt + bne` ）快得多，此时使用B+树不仅实现困难，性能上也会极大的不如红黑树、Splay树等二叉树。只有把节点存储在磁盘中，让读取的时间消耗远大于分支控制的时间，B+树才会真正体现出它的优势。我使用了 `std::fstream file("data/test.bin", std::ios::in | std::ios::out | std::ios::binary)` 的方式，紧凑高效的存储树的每一个节点，同时使用数组下标而不是指针以存储信息，从而让读取的信息始终稳定准确。

将一部分信息存储在内存中，而不是全部依赖于磁盘操作——LRU缓存算法

靠近根的几个节点经常被访问，如果在内存中存储它们，不仅不会占用大量内存，还会大幅提升访问速度。我采用LRU缓存算法管理节点，如果节点最近被访问则存储在内存中，如果节点很久没被访问就从内存中移除。通过这种算法，B+树的性能得以提升。

B+树的正确性

在一开始的测试中，我尝试对B+树和红黑树进行相同的操作，并检测两棵树是否完全相等，测试结果显示失败。在反复测试中，我发现分裂的逻辑有很大漏洞，导致某些键在插入后立刻消失，导致正确性受到挑战。在重构分裂逻辑后，正确性问题得到解决。

B+树的可持久性

在一开始的测试中，我发现重新打开一棵树时，树的信息不完整，导致系统无法解析它的信息。我在其中添加了 `TreeMetadata` 类，在其中存储了树根等信息，从而提升了树的可持久化性。

如何验证我的数据结构的正确性?

项目采用了全面的多层测试策略来确保正确性：

1. 压力测试

压力测试 `pressure_test.cpp` 使用 C++ 的 `std::set` 作为参考实现：

```
std::set<int> cmp;
bplus_sql::BPlusTree tree;
for(int i = 0; i < 10000000; ++i) {
   int op = randInt(1, 2), value = randInt(1, 100000);
   assert(cmp.contains(value) == tree.contains(value));
   if(op == 1) {
      cmp.insert(value);
      tree.insert(value);
   } else {
      cmp.erase(value);
      tree.erase(value);
   }
}
```

该测试执行 1000 万次随机操作，持续验证 B+ 树的行为是否与经过验证正确的标准库实现相匹配。

2. 连续插入测试

`always_insert.cpp` 执行持续的插入操作：

- 测试树能否处理持续增长的数据集
- 验证连续负载下的分裂行为
- 确保没有内存泄漏或资源耗尽

3. 持久性测试

`test_existing_tree.cpp` 验证数据库是否能在程序重启后重新打开，并正确读取信息：

- 在 `always_insert.cpp` 成功运行并退出后，重新打开同一个文件
- 验证所有先前插入的值仍然存在
- 确保树结构在不同会话间得到正确维护

4. 命令解析器测试

`parse_commands.cpp` 测试类 SQL 命令接口：

- 验证 CREATE、INSERT、QUERY、ERASE 和 DESTROY 命令的解析

- 测试不区分大小写的命令识别
- 确保正确提取表名、操作和键值

5. 性能基准测试

`rb_tree.cpp` 实现了一个完整的基于磁盘的红黑树用于性能比较：

- 在 B+ 树和红黑树上运行相同的工作负载，使用同样的磁盘管理算法和LRU缓存算法
- 测量并比较执行时间
- 验证 B+ 树性能具有竞争力或更优
- 展示 B+ 树在基于磁盘存储方面的优势（更好的缓存局部性，更少的磁盘寻道）

6. 自动化测试套件

脚本（`run_all_tests.sh` 和 `run_all_tests.ps1`）自动化整个测试过程：

- 编译所有测试可执行文件
- 依次运行每个测试
- 报告通过/失败状态
- 支持跨平台测试（Linux/macOS 通过 Bash，Windows 通过 PowerShell）

这种全面的测试方法确保了 B+ 树实现是：

- **正确的**：与经过验证的标准库容器的行为相匹配
- **持久的**：在程序重启间保持数据
- **高性能的**：性能远远超过替代数据结构
- **不退化的**：高效处理边缘情况和压力条件

使用方法

构建项目

项目使用 CMake 进行跨平台构建。要求：

- C++23 兼容的编译器（GCC 12+、Clang 16+、MSVC 2022+）
- CMake 3.10 或更高版本

```
# 克隆仓库
git clone https://github.com/Lumine2024/BPLUS_SQL.git
cd BPLUS_SQL

# 创建构建目录并编译
mkdir build
cd build
cmake ..
cmake --build .
```

运行数据库

主程序以交互方式或从文件接受类 SQL 命令：

```
# 交互模式
./main

# 从文件的批处理模式
./main commands.txt
```

命令语法

BPLUS_SQL是一个简化的SQL数据库，它的指令与SQL类似，但略有不同。数据库支持以下操作：

1. **CREATE TABLE** - 初始化新的数据库表

   ```
   CREATE TABLE users
   ```

2. **INSERT** - 向表中添加整数键

   ```
   INSERT INTO users KEY 42
   INSERT INTO users KEY 123
   ```

3. **QUERY** - 搜索键（找到返回 1，未找到返回 0）

   ```
   QUERY FROM users KEY 42
   ```

4. **ERASE** - 从表中删除键

   ```
   ERASE FROM users KEY 42
   ```

5. **DESTROY TABLE** - 删除整个表及其磁盘文件

```
DESTROY TABLE users
```

6. **EXIT** - 关闭数据库

```
EXIT
```

注意：

- 命令不区分大小写
- 数据库文件以 `<表名>.bin` 格式存储在 `data/` 目录中
- 每个表都是一个单独的 B+ 树，拥有自己的文件
- 表自动持久化到磁盘，并能在程序重启后恢复

运行测试

```
# 在 Linux/macOS 上
cd tests
./run_all_tests.sh

# 在 Windows 上
cd tests
.\run_all_tests.ps1

# 运行单个测试
./pressure_test      # 1000万次随机操作验证
./test_existing_tree # 持久性测试
./rb_tree            # 性能比较
```

哪些方面可以进一步增强？

虽然 BPLUS_SQL 成功实现了 B+ 树的核心功能，但有几项增强可以显著改进系统：

1. 支持范围查询

目前系统仅支持精确键搜索。实现范围查询将利用链接的叶节点结构：

```
std::vector<int> rangeQuery(int start, int end);
```

这将支持诸如“查找 100 到 200 之间的所有键”之类的查询。

2. 带锁的并发访问

当前实现是单线程的。增加对并发操作的支持将使其实用化：

- 实现 B-link 树变体以支持无锁读取
- 添加页面级或节点级锁定
- 支持 MVCC（多版本并发控制）以获得更好的并发性

3. 支持多种数据类型

目前仅限于整数。可以扩展到：

- 可变长度字符串（需要对页面布局进行特殊处理）
- 浮点数
- 自定义复合键
- 基于模板的通用实现

4. 二级索引

允许多个 B+ 树从不同角度索引相同数据，支持复杂的查询模式。

结论

BPLUS_SQL 成功演示了一个用于整数存储的基于磁盘的 B+ 树数据库系统的完整实现。该项目展示了几个重要的系统编程概念：

技术成就：

- 具有插入、搜索和删除操作的全功能 B+ 树
- 带有自定义分页实现的持久化存储系统
- 用于性能优化的 LRU 缓存
- 用于树状态持久化的综合性元数据管理
- 用于直观交互的类 SQL 命令接口
- 包括正确性验证和性能基准测试在内的广泛测试

经验教训：
开发过程突显了构建数据库系统的挑战，包括正确的内存管理、高效处理磁盘 I/O、在复杂操作期间保

持数据结构的不变性，以及确保数据在程序重启间的持久性。遇到的每个问题——从堆损坏到缓存管理——都为深入理解低级系统编程和算法实现提供了宝贵的见解。

项目影响：
此实现可作为理解真实数据库系统内部工作原理的教育参考。它证明了 B+ 树不仅是理论构造，而且是构建高效、持久化数据存储系统的实用工具。代码文档齐全，经过彻底测试，并在 MIT 许可证下可供他人学习和构建。

未来展望：
虽然当前实现提供了坚实的基础，但建议的增强功能描绘了通向生产就绪数据库系统的路径。模块化架构（分离的分页器、缓存、节点管理器和树逻辑）使得未来的扩展可行，无需大规模重构。

BPLUS_SQL 证明了经典数据结构应用于现代存储系统的力量，并展示了通过精心设计和彻底测试，可以可靠且高效地构建复杂系统。

感谢

仓库： https://github.com/Lumine2024/BPLUS_SQL