期末大作业简介
- Task1 - Index Concurrency
  - 文档参考
  - 任务赋分
- Task2 :Order By
  - example
    - SQL测试1
    - SQL测试2
    - SQL测试3
    - SQL测试4
  - 帮助
  - 任务赋分
- 实验报告要求
- 截止时间说明

期末大作业简介

> 期末大作业总共40分，包括2个代码任务（15分+15分）和1个实验报告任务（10分）。其中代码任务部分两个任务均包括5分性能分。

Task1 - Index Concurrency

在本任务中，你需要在lab2的基础上，实现B+树的Page级并发控制协议。

文档参考

请自行学习B+树索引并发算法：**蟹行协议（crabbing protocol）**。

蟹行协议实现可参考实验文档 `docs/Rucbase-Lab2[索引管理实验文档].md`

主要需要修改 `IxIndexHandle` 类中以下函数的实现逻辑：

（1） `FindLeafPage()`

此函数十分重要，在B+树的查找/插入/删除操作中均被调用。其基本功能在于根据指定 `key` 从根结点向下查找到含有该 `key` 的叶结点。

引入并发控制算法，以 **蟹行协议（crabbing protocol）** 为例，其使用读写锁来控制对树结点（索引页面）的访问和修改，并规定向下遍历树时 获取/释放 锁的机制：每个线程都是以自上而下的方式获取锁，从根结点开始获取锁，然后向下进入孩子结点并获取锁，再选择是否释放父结点的锁。

参数 `operation` 表示操作类型。对于查找操作，进入树的每一层结点都是先在当前结点获取读锁，然后释放父结点读锁；对于插入和删除操作，进入树的每一层结点都是先在当前结点获取写锁，如果当前结点“安全”才释放所有祖先节点的写锁。“安全”结点的定义是：结点插入一个键值对后仍然未满(size+1<max_size)；或者结点删除一个键值对后仍然超过或等于半满(size-1>=min_size)；注意，根结点的min_size=2，其余结点的min_size=max_size/2。

参数 `transaction` 表示事务，其中有一个数据结构 `page_set_` 用于存储从根结点到当前结点经过的所有祖先结点（索引页面）。实际上，只有插入或删除操作需要记录当前结点的所有祖先结点，然后判断如果当前结点是“安全”的，就遍历 `transaction` 的 `page_set_` 中存放的所有页面，依次释放这些页面的写锁。

函数返回值修改为`std::pair<IxNodeHandle*, bool>`，其两部分分别表示找到的叶结点以及根结点是否被锁住。在`IxIndexHandle`类中设计了一个mutex锁（互斥锁）`root_latch_`用于对根结点进行上锁。对于读操作（查找），不需要对根结点上锁，因为蟹行协议允许多个线程同时读B+树；但对于写操作（插入/删除），则需要上锁，直到确定根结点不会被修改或者已经将根结点修改完毕，才能释放锁，从而防止本线程写操作未完成而其他线程又进行读的错误。最后用一个bool类型的变量表示根结点是否被上锁。

（2）查找函数`GetValue()`

与之前实现不同的是，此处经过`FindLeafPage()`找到的叶结点被加上了读锁，且其祖先结点无任何读锁。最后释放叶结点的读锁即可。

（3）插入函数`insert_entry()`、`Split()`、`InsertIntoParent()`

删除函数`delete_entry()`、`CoalesceOrRedistribute()`、`Coalesce()`、`Redistribute()`、`AdjustRoot()`

当要插入或删除某个键值对时，首先获取根结点的写锁， 在其孩子结点上获取写锁。然后判断孩子结点是否“安全”，只有孩子结点安全才能释放它的所有祖先结点的写锁。不断重复这一过程，直到找到叶结点，最后叶结点获取的是写锁。

注意释放结点写锁的时机：对于每一层结点，都是确定其安全之后，才能释放其上层的写锁。

任务赋分

本任务满分为15分，测试文件对应的任务点及其分值如下：

任务点	测试文件	分值
B+树并发控制-功能测试	src/index/b_plus_tree_concurrent_test.cpp	60
B+树并发控制-性能测试	src/index/b_plus_tree_concurrent_test.cpp	40

- 功能计分

在lab2中你完成了Tree级的粗粒度B+树索引并发，但是实际上是串行执行了查找、插入、删除这三个操作，执行效率低下。在期末大作业中，我们希望你能将其改写成Page级的细粒度B+树索引并发，尽可能地提高操作之间的并发度，并且保证执行结果的正确性。你可以尝试实现蟹行协议（crabbing protocol）或者其他B+树并发控制算法。我们将使用更新后的测试文件`src/index/b_plus_tree_concurrent_test.cpp`测试并发功能的正确性。注意，我们会设置一个时间限制（具体见测试文件），只有你的程序在这个时间限制内通过了所有测试点才认为实现的并发算法有效。

- 性能计分

在保证功能正确的前提下（通过所有测试点），你可以实现任意并发控制算法，例如引入意向锁、修改“悲观估计”为”乐观估计“等。我们将用测试程序运行时间升序进行排名，按排名分档次给分，排名越靠前，你所得到的性能分数越高。

性能分计算规则：耗时保留一位小数，第一名为满分5分，以第一名成绩为参照，其余同学分数以第一名耗时按比例赋分。赋分保留一位小数计入纳入课程成绩计算。

如第一名平均用时为4.5s，A平均用时7.0s，则A在本任务中的性能得分为$Score = 5 \times \frac{4.5}{7.0} = 3.2$

未能通过功能测试，性能分计为0分。

你可以编译生成可执行文件进行测试：

```
cd build

make b_plus_tree_concurrent_test
./bin/b_plus_tree_concurrent_test
```

注意：在本实验中的所有测试只调用`GetValue()`、`insert_entry()`、`delete_entry()`这三个函数。学生可以自行添加和修改辅助函数，但不能修改以上三个函数的声明。

Task2 :Order By

在本任务中，你需要实现SQL查询的order by子句。这个功能要求从parser模块开始修改。你需要添加以下关键字(大小写均需要支持)：

```
order|ORDER
by|BY
asc|ASC
desc|DESC
limit|LIMIT
```

其中，ASC为升序排序，DESC为降序排序。Order By不指定时你应该默认采用**升序排序**。

对于数字，你应该采用数值进行比较。对于字符串或者字符，使用字典序进行比较。本任务不要求你实现数字与字符串进行的比较。

example

表示例：

Company	OrderNumber
IBM	3532
Microsoft	2356
Apple	4698
Microsoft	6953

自测建表语句:

```
create table test (Company char(10), OrderNumber int);
insert into test values('IBM',3532);
insert into test values('Microsoft',2356);
insert into test values('Apple',4698);
insert into test values('Microsoft',6953);
```

SQL测试1

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company;
```

输出:

Company	OrderNumber
Apple	4698
IBM	3532
Microsoft	2356
Microsoft	6953

或者

Company	OrderNumber
Apple	4698
IBM	3532
Microsoft	6953
Microsoft	2356

SQL测试2

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company, OrderNumber;
```

输出

Company	OrderNumber
Apple	4698
IBM	3532
Microsoft	2356
Microsoft	6953

SQL测试3

```
SELECT Company, OrderNumber FROM Orders ORDER BY Company DESC, OrderNumber
ASC;
```

输出

Company	OrderNumber
Microsoft	2356
Microsoft	6953
IBM	3532
Apple	4698

SQL测试4

```
SELECT Company, OrderNumber FROM Orders ORDER BY OrderNumber ASC LIMIT 2;
```

输出

Company	OrderNumber
Microsoft	2356
IBM	3532

其中， LIMIT关键的语法只需要支持TOP N，不需要实现 TOP N , M

帮助

你需要从Lexer,Parser开始修改关键词和语法语义分析。本系统使用的工具是flex和bison,你需要修改`lex.l`和`yacc.y`文件。并生成所使用的`.c .cpp`文件。

命令为:

```
flex lex.l
bison -d yacc.y
```

你可以以函数`void interp_sql(const std::shared_ptr<ast::TreeNode> &root, Context *context)`为入口点，跟踪sql语句AST的执行流程，修改要增加功能的地方。

由于本任务聚焦的SQL为SELECT语句，你应该跟踪`SelectStmt`结构体

```
struct SelectStmt : public TreeNode {
    std::vector<std::shared_ptr<Col>> cols;
    std::vector<std::string> tabs;
    std::vector<std::shared_ptr<BinaryExpr>> conds;

    SelectStmt(std::vector<std::shared_ptr<Col>> cols_,
               std::vector<std::string> tabs_,
               std::vector<std::shared_ptr<BinaryExpr>> conds_) :
            cols(std::move(cols_)), tabs(std::move(tabs_)),
conds(std::move(conds_)) {}
};
```

你需要对其进行修改。

排序部分是性能优化的重点，本任务的性能测试聚焦于单表，因此你不必考虑多表连接问题，你可以考虑的优化点有：

- 表扫描性能
- 排序性能（索引优化，排序算法等）
- LIMIT排序优化
- 优化`exec_sql.cpp`或者`rucbase_client`，如果你提供自己的自行sql 的程序，我们会使用你提供的程序来执行测试sql语句。你可以选择以`exec_sql`的方式本地运行，也可以选择使用`rucbase_client`的方式以C/S运行(统一测试，不会分开比较性能)。当前的`exec_sql`是每一条sql语句都会初始化一个执行环境，处理大量sql语句时这样会造成巨量的性能损失。你应该编写一个程序使得执行所有sql语句都在一个执行环境context 中。如不提供，我们会使用`exec_sql`的方式单条逐次统计耗时求和作为你的最终性能数据。

任务赋分

本任务分值分为功能分和性能分。功能分要求你正确实现功能并能够正确执行输出example一节中的测试示例。性能分部分会根据给定的大表（之后提供）进行多条Order By查询，最终根据你的性能排名赋予对应层级的分值。

- 功能分：10分
- 性能分：5分

性能分部分，我们允许你在给定大表上预先按照你给出的SQL语句建立你想要的索引，索引数目不限，建立索引的时间不纳入性能计算范围内。此外，为降低难度，本实验测试数据暂时只需要考虑**内排序**。

性能分计算规则：第一名为满分5分，以第一名成绩为参照，其余同学分数以第一名耗时按比例赋分。赋分保留一位小数计入纳入课程成绩计算。

如第一名平均用时为4.5s，A平均用时7.0s，则A在本任务中的性能得分为$Score = 5 \times \frac{4.5}{7.0} = 3.2$

未能通过功能测试，性能分计为0分。

实验报告要求

实验报告共10分，你需要完成的内容有：

- 详细介绍你实现的期末大作业task1和task2。
  - 功能部分的实现思路
  - 性能部分的创新点
- 讲述你完成平时作业和期末作业的心得体会。
- 不少于3000字

截止时间说明

整个实验的截止时间为开学第一周最后一天，日期依据以微人大系统日历为准。

因此，本课程期末大作业截止时间为:**2023年2月26日 23：59**，请注意合理分配时间