比赛题目

1 存储管理

题目描述：

在大赛提供的代码框架的基础上，提供相关接口的实现，完成指定功能。本题目包含若干子任务，每个任务对应不同的测试点，只有通过相应测试点才可以获得相应的分数。

提示：

l 测试代码会调用指定接口来进行测试，参赛队伍不能修改已有的接口，也不能删除已有的数据结构及数据结构中的变量，但是可以增添新的接口、数据结构、变量。

l 大赛提供了项目结构文档来帮助参赛队伍理解代码框架，同时在代码注释中，也对数据结构及接口进行了说明，参赛队伍可以通过阅读代码注释来辅助理解代码框架。

1、磁盘管理器

在本任务中，参赛队伍需要实现磁盘管理器 DiskManager 的相关接口，磁盘管理器负责文件操作、读写页面等。在完成本任务之前，参赛队伍需要阅读项目结构文档中磁盘管理器的相关说明，以及代码框架中 src/errors.h、src/storage/disk_manager.h、src/storage/disk_manager.cpp、src/common/config.h 文件。

参赛队伍需要实现以下接口：

（1）void DiskManager::create_file(const std::string &path);

该接口的参数 path 为文件名，该接口的功能是创建文件，其文件名为 path 参数指定的文件名。

（2）void DiskManager::open_file(const std::string &path);

该接口的参数 path 为文件名，该接口的功能是打开文件名参数 path 指定的文件。

（3）void DiskManager::close_file(const std::string &path);

该接口的参数 path 为文件名，该接口的功能是关闭文件名参数 path 指定的文件。

（4）void DiskManager::destroy_file(const std::string &path);

该接口的参数 path 为文件名，该接口的功能是删除文件名参数 path 指定的文件。

（5）void DiskManager::write_page(int fd, page_id_t page_no, const char *offset, int num_bytes);

该接口负责在文件的指定页面写入指定长度的数据，该接口从指定页面的起始位置开始写入数据。

（6）void DiskManager::read_page(int fd, page_id_t page_no, char *offset, int num_bytes);

该接口需要从文件的指定页面读取指定长度的数据，该接口从指定页面的起始位置开始读取数据。

2、缓冲池管理器

在本任务中，参赛队伍需要实现缓冲池管理器 BufferPoolManager 和缓冲池替换策略 Replacer 相关的接口，缓冲池管理器负责管理缓冲池中的页面在内外存的交换，缓冲池替换策略主要负责缓冲区页面的淘汰和查找。在完成本任务之前，参赛队伍需要首先阅读项目结构

文档中缓冲池管理器的相关说明，以及代码框架中 src/storage 和 src/replacer 文件夹下的代码文件。

对于缓冲池替换策略 Replacer，参赛队伍需要实现一个 Replacer 的子类 LRUReplacer，LRUReplacer 实现了缓冲池页面替换的 LRU 策略，需要实现的接口如下：

(1) bool LRUReplacer::victim(frame_id_t* frame_id);

该接口的功能是选择并淘汰缓冲池中一个页面。如果成功找到要淘汰的页面，则函数返回值为 true；否则，返回值为 false。被淘汰的页面所在的帧由参数 frame_id 返回。

(2) void LRUReplacer::pin(frame_id_t frame_id);

该接口的功能是固定住一个帧中的页面，代表该页面正在使用，不可被换出，参数 frame_id 指定了帧的编号。

(3) void LRUReplacer::unpin(frame_id_t frame_id);

该接口的功能是取消固定一个帧中的页面，当该页面使用完毕时调用 unpin 函数取消对该页面的固定，参数 frame_id 指定了帧的编号。

对于缓冲池管理器 BufferPoolManager，参赛队伍需要管理缓冲池中的页面，并对缓冲池进行并发控制，需要实现的接口如下：

(1) Page *BufferPoolManager::new_page(PageId *page_id);

该成员函数用于在缓冲池中申请创建一个新页面。如果创建新页面成功，则返回指向该页面的指针，同时通过参数 page_id 返回新建页面的编号。

(2) Page *BufferPoolManager::fetch_page(PageId page_id);

该成员函数用于获取缓冲池中的指定页面。待获取页面的编号由参数 page_id 给出。

(3) bool BufferPoolManager::find_victim_page(frame_id_t *frame_id);

当缓冲池中没有可用的空闲帧时，该成员函数用于寻找需要淘汰的页面。如果成功找到要淘汰的页面，则函数返回值为 true；否则，返回值为 false。被淘汰的页面所在的帧由参数 frame_id 返回。在实现这个成员函数时，需要调用 LRUReplacer::victim 函数。

(4) void BufferPoolManager::update_page(Page *page, PageId new_page_id, frame_id_t new_frame_id);

当缓冲池想要把某个帧中的页面置换为新页面或者删除该帧中的页面时，会调用 update_page 函数，该函数把指定帧中的原有页面刷入磁盘中，并将新页面和该帧建立映射，即更新页表。

(5) bool BufferPoolManager::unpin_page(PageId page_id, bool is_dirty);

当某个操作完成某个页面的使用之后，需要调用该函数将取消该操作对该页面的固定。当所有操作都完成该页面的使用之后，需要在 Replacer 中调用 Unpin 函数取消该页面的固定。

(6) bool BufferPoolManager::delete_page(PageId page_id);

用于在缓冲池中删除指定页面，同时将该页面所在的帧置为空闲帧。如果当前页面正在被某个操作使用，则该页面不能被删除。

(7) bool BufferPoolManager::flush_page(PageId page_id);

用于强制刷新缓冲池中的指定页面到磁盘上，无论该页是否正在被使用，或者是否为脏页，都需要把该页面的数据刷入磁盘中。

(8) void BufferPoolManager::flush_all_pages(int fd);

用于将指定文件中的存在于缓冲池的所有页面都刷新到磁盘。

提示：在缓冲池中，需要淘汰某个脏页时，需要将脏页写入磁盘。

3、记录管理器

在本任务中，参赛队伍需要填充记录管理器涉及的 RmFileHandle 类和 RmScan 类，RmFileHandle 类负责对表的记录进行操作，RmScan 类用于遍历表文件中存放的记录。

对于 RmFileHandle 类。在完成本文任务之前，参赛队伍需要阅读项目结构文档中记录管理器的相关说明，以及代码框架中 src/record 文件夹下的代码文件。

参赛队伍需要实现的接口如下：

(1) std::unique_ptr RmFileHandle::get_record(const Rid &rid, Context *context) const;

该函数用于获取表中某一条指定位置的记录，每条记录由 Rid 来进行唯一标识。

(2) Rid RmFileHandle::insert_record(char *buf, Context *context);

该函数负责向表中插入一条新记录，在该函数中未指定记录的插入位置，参赛队伍需要选择一个空闲位置插入记录并同步更新表的元数据信息。

(3) void RmFileHandle::insert_record(const Rid &rid, char *buf);

该函数负责向表中的指定位置插入一条记录，该函数主要用于事务的回滚和系统故障恢复。

(4) void RmFileHandle::delete_record(const Rid &rid, Context ontext);

该函数负责删除表中指定位置的记录。

(5) void RmFileHandle::update_record(const Rid &rid, char *buf, Context *context);

该函数负责把表中指定位置的记录更新为新的值。

对于 RmScan 类，参赛队伍需要实现的接口如下：

(1) RmScan(const RmFileHandle *file_handle);

该函数为 RmScan 类的构造函数，需要初始化相关成员变量。

(2) void RmScan::next() override;

该函数负责找到表文件中下一个存放了合法记录的位置。

(3) bool RmScan::is_end() const override;

该函数负责判断是否已经扫描到文件的末尾位置。

2 查询执行

题目描述：

在实现题目一功能的基础上增加查询执行模块，使得系统支持通过测试需要的元数据管理、DDL 语句、DQL 语句、DML 语句。

提示：

l 本题目提供了空缺的样例代码，参赛队伍可以选择进行补全，也可以选择自行构建。如果选择补全空缺的样例代码，需要实现“元数据管理与 DDL 语句”和“DQL 语句和 DML 语句”。

l 本题目通过 sql 进行测试，分为五个测试点，它们依次是“尝试建表”、“单表插入与条件查询”、“单表更新与条件查询”、“单表删除与条件查询”、“连接查询”。

1、元数据管理与 DDL 语句

本任务要求参赛队伍对数据库的元数据进行管理，并实现基本的 DDL 语句，包括 create table 和 drop table 两种语句。大赛提供的框架中，与元数据管理和 DDL 语句相关的代码文件位于 src/system 文件夹下，代码框架中提供了 create table 语句的实现方式。

2、DQL 语句和 DML 语句

DML 语句要求实现基本的增删改，即 insert、delete、update 语句。DQL 语句要求实现 select 语句。在完成本任务之前，参赛队伍需要首先阅读项目结构文档中查询解析、查询优化、查询执行的相关说明，以及代码框架中 src/analyze、src/optimizer、src/execution 文件夹下的代码文件，参赛队伍需要实现代码框架中未提供的功能，包括语义检查、查询执行计划的生成、执行算子等。

本题目通过 SQL 来进行测试，因此参赛队伍可以对代码进行重构。

测试示例：

测试点 1: 尝试建表（2 分）

测试示例：

create table t1(id int,name char(4));

show tables;

create table t2(id int);

show tables;

drop table t1;

show tables;

drop table t2;

show tables;

期待输出：

| Tables |

| t1 |

| Tables |

| t1 |

| t2 |

| Tables |

| t2 |

| Tables |

测试点 2: 单表插入与条件查询（2 分）

测试示例：

```
create table grade (name char(20),id int,score float);
insert into grade values (‘Data Structure’, 1, 90.5);
insert into grade values (‘Data Structure’, 2, 95.0);
insert into grade values (‘Calculus’, 2, 92.0);
insert into grade values (‘Calculus’, 1, 88.5);
select * from grade;
select score,name,id from grade where score > 90;
select id from grade where name = ‘Data Structure’;
select name from grade where id = 2 and score > 90;
```

期待输出：

```
| name | id | score |
| Data Structure | 1 | 90.500000 |
| Data Structure | 2 | 95.000000 |
| Calculus | 2 | 92.000000 |
| Calculus | 1 | 88.500000 |
| score | name | id |
| 90.500000 | Data Structure | 1 |
| 95.000000 | Data Structure | 2 |
| 92.000000 | Calculus | 2 |
| id |
| 1 |
| 2 |
| name |
| Data Structure |
| Calculus |
```

测试点 3：单表更新与条件查询（2 分）

测试示例：

```
create table grade (name char(20),id int,score float);
insert into grade values (‘Data Structure’, 1, 90.5);
insert into grade values (‘Data Structure’, 2, 95.0);
insert into grade values (‘Calculus’, 2, 92.0);
```

insert into grade values (‘Calculus’, 1, 88.5);

select * from grade;

update grade set score = score + 5 where name = ‘Calculus’;

select * from grade;

update grade set name = ‘Error name’ where name > ‘A’;

select * from grade;

update grade set name = ‘Error’ ,id = −1,score = 0 where name = ‘Error name’ and score > 90;

select * from grade;

期待输出：

| name | id | score |

| Data Structure | 1 | 90.500000 |

| Data Structure | 2 | 95.000000 |

| Calculus | 2 | 92.000000 |

| Calculus | 1 | 88.500000 |

| name | id | score |

| Data Structure | 1 | 90.500000 |

| Data Structure | 2 | 95.000000 |

| Calculus | 2 | 97.000000 |

| Calculus | 1 | 93.500000 |

| name | id | score |

| Error name | 1 | 90.500000 |

| Error name | 2 | 95.000000 |

| Error name | 2 | 97.000000 |

| Error name | 1 | 93.500000 |

| name | id | score |

| Error | −1 | 0.000000 |

| Error | −1 | 0.000000 |

| Error | −1 | 0.000000 |

| Error | −1 | 0.000000 |

测试点 4：单表删除与条件查询（2 分）

测试示例：

create table grade (name char(20),id int,score float);

insert into grade values (‘Data Structure’, 1, 90.5);
select * from grade;
delete from grade where score > 90;
select * from grade;
期待输出：
name	id	score
Data Structure	1	90.500000
name	id	score
测试点 5：连接查询（4 分）
测试示例：
create table t (id int , t_name char (3));
create table d (d_name char(5),id int);
insert into t values (1,’aaa’);
insert into t values (2,’baa’);
insert into t values (3,’bba’);
insert into d values (‘12345’,1);
insert into d values (‘23456’,2);
select * from t, d;
select t.id,t_name,d_name from t,d where t.id = d.id;
select t.id,t_name,d_name from t join d where t.id = d.id;
期待输出：
id	t_name	d_name	id
1	aaa	23456	2
1	aaa	12345	1
2	baa	23456	2
2	baa	12345	1
3	bba	23456	2
3	bba	12345	1
id	t_name	d_name
1	aaa	12345
2	baa	23456
id	t_name	d_name
1	aaa	12345

| 2 | baa | 23456 |

3 唯一索引

题目描述：

在现有系统的基础上增添索引功能，该索引为唯一索引，要求系统能够支持创建索引、删除索引、展示某个表上的索引、单点查询和范围查询，实现索引与基表的同步。

如果一个表上的某个字段建有唯一索引，则该表中任意两个记录中该字段的值不相同。

提示：

（1）功能题目对代码框架没有限制，参赛队伍可以修改、增添、删除数据结构及接口，也可以对框架进行重构。

（2）磁盘数据库中常用的索引为 B+树索引，推荐使用 B+树来实现索引功能，参赛队伍也可以使用其他类型的索引，但需要能够支持题目中要求的功能。

（3）对于测试点 2、3，即“索引的查询”和“维护索引的插入、删除、更新”，会进行单独的是否真正使用了索引的测试——对于数千条查询语句，建立索引后的执行时间应该小于建立索引前的执行时间的 70%，才可以视为真正使用了索引的测试。

（4）改题目不要求使用 B+树作为索引，所以对于参赛队伍测试点 2、3 的输出结果不要求行的顺序与期待输出的行的顺序完全一致，但列的顺序要求完全一致。

1、创建、删除、展示索引

（1）支持单个字段索引的创建和删除；

（2）支持多个字段索引的创建和删除；

（3）查看某个表上的索引信息；

（4）show index from table_name 的输出格式为 | table_name | unique | (column_name, column_name) |。参考下方测试用例和期待输出

测试示例：

create table warehouse (id int, name char(8));

create index warehouse (id);

show index from warehouse;

create index warehouse (id,name);

show index from warehouse;

drop index warehouse (id);

drop index warehouse (id,name);

show index from warehouse;

期望输出：

| warehouse | unique | (id) |

| warehouse | unique | (id) |

| warehouse | unique | (id,name) |

2、索引查询

在创建索引后，能够使用索引进行单点查询和范围查询。

提示：

在大赛提供的框架中，只有查询条件与索引完全一致，并且是单点查询时，才使用索引来进行查询，你需要对索引匹配规则进行修改，要求使用最左匹配原则。例如，表 A 的(id, name, score)三个属性上有一个联合索引，对于以下几种查询，都需要使用索引来进行查询：

l select * from A where id = 1 and name = ‘abcd’ and score = 99.0;

l select * from A where id = 1 and name = ‘abcd’ and score > 90.0;

l select * from A where id = 1 and name = ‘abcd’;

l select * from A where name = ‘abcd’ and id = 1;

l select * from A where id = 1;

l select * from A where id > 1;

测试示例：

create table warehouse (w_id int, name char(8));

insert into warehouse values (10 , ‘qweruiop’);

insert into warehouse values (534, ‘asdfhjkl’);

insert into warehouse values (100,’ qwerghjk’);

insert into warehouse values (500,’ bgtyhnmj’);

create index warehouse(w_id);

select * from warehouse where w_id = 10;

select * from warehouse where w_id < 534 and w_id > 100;

drop index warehouse(w_id);

create index warehouse(name);

select * from warehouse where name = ‘qweruiop’;

select * from warehouse where name > ‘qwerghjk’;

select * from warehouse where name > ‘aszdefgh’ and name < ‘qweraaaa’;

drop index warehouse(name);

create index warehouse(w_id,name);

select * from warehouse where w_id = 100 and name = ‘qwerghjk’;

select * from warehouse where w_id < 600 and name > ‘bztyhnmj’;

期待输出：

| w_id | name |

| 10 | qweruiop |

| w_id | name |

| 500 | bgtyhnmj |

| w_id | name |

| 10 | qweruiop |

| w_id | name |

| 10 | qweruiop |

| w_id | name |

| 500 | bgtyhnmj |

| w_id | name |

| 100 | qwerghjk |

| w_id | name |

| 10 | qweruiop |

| 100 | qwerghjk |

3、索引维护

在建有索引的表中插入、删除、更新数据时，能够根据表数据的变化同步对表上对索引进行更新，保证索引的正确性。同时，在索引被更新时，需要检查唯一性约束。

测试示例：

create table warehouse (w_id int, name char(8));

insert into warehouse values (10 , ‘qweruiop’);

insert into warehouse values (534, ‘asdfhjkl’);

select * from warehouse where w_id = 10;

select * from warehouse where w_id < 534 and w_id > 100;

create index warehouse(w_id);

insert into warehouse values (500, ‘lastdanc’);

update warehouse set w_id = 507 where w_id = 534;

select * from warehouse where w_id = 10;

select * from warehouse where w_id < 534 and w_id > 100;

期待输出：

| w_id | name |

| 10 | qweruiop |

| w_id | name |

| w_id | name |

| 10 | qweruiop |

| w_id | name |

| 500 | lastdanc |

| 507 | asdfhjkl |

4 聚合函数与分组统计

题目描述:

本题目要求实现聚合函数与分组统计。聚合函数：对一组值进行计算并返回单一的值，通过使用 SQL 聚合函数，可以确定数值集合的各种统计值。分组统计：将查询结果按照 1 个或者多个字段进行分组，字段值相同的为同一组。

要求：

l 聚合函数 COUNT(expr)：其中 expr 可以是列名、"*"。count(*)统计元组个数，count(列名)统计一列中值的个数。该函数仅需支持 int、float、char 类型的字段。

l 聚合函数 MAX(expr)：返回一列中的最大值。该函数仅需支持 int、float 类型的字段。

l 聚合函数 MIN(expr)：返回一列中的最小值。该函数仅需支持 int、float 类型的字段。

l 聚合函数 SUM(expr)：返回数值列的总数。该函数仅需支持 int、float 类型的字段。

l 分组统计(group by、having)：group by 子句根据一个或多个列对结果集进行分组，可以使用 having 子句对每一个分组按条件进行过滤。分组属性仅需支持 int、float、char 类型的字段。

测试示例：

测试点 1：单独使用聚合函数

测试示例:

create table grade (course char(20),id int,score float);

insert into grade values（‘DataStructure’,1,95);

insert into grade values（‘DataStructure’,2,93.5);

insert into grade values（‘DataStructure’,4,87);

insert into grade values（‘DataStructure’,3,85);

insert into grade values（‘DB’,1,94);

insert into grade values（‘DB’,2,74.5);

insert into grade values（‘DB’,4,83);

insert into grade values（‘DB’,3,87);

select MAX(id) as max_id from grade;

select MIN(score) as min_score from grade where course = ‘DB’;

select COUNT(course) as course_num from grade;

select COUNT(*) as row_num from grade;

select SUM(score) as sum_score from grade where id = 1;

drop table grade;

期待输出：

| max_id |

| 4 |

| min_score |

| 74.500000 |

| course_num |

| 8 |

| row_num |

| 8 |

| sum_score |

| 189.000000 |

测试点 2：聚合函数加分组统计

测试示例:

```
create table grade (course char(20),id int,score float);
insert into grade values(‘DataStructure’,1,95);
insert into grade values(‘DataStructure’,2,93.5);
insert into grade values(‘DataStructure’,3,94.5);
insert into grade values(‘ComputerNetworks’,1,99);
insert into grade values(‘ComputerNetworks’,2,88.5);
insert into grade values(‘ComputerNetworks’,3,92.5);
insert into grade values(‘C++’,1,92);
insert into grade values(‘C++’,2,89);
insert into grade values(‘C++’,3,89.5);
select id,MAX(score) as max_score,MIN(score) as min_score,SUM(score) as sum_score from grade group by id;
select id,MAX(score) as max_score from grade group by id having COUNT(*) > 3;
insert into grade values (‘ParallelCompute’,1,100);
select id,MAX(score) as max_score from grade group by id having COUNT(*) > 3;
select id,MAX(score) as max_score,MIN(score) as min_score from grade group by id having COUNT(*) > 1 and MIN(score) > 88;
select course ,COUNT(*) as row_num , COUNT(id) as student_num , MAX(score) as top_score, MIN(score) as lowest_score from grade group by course;
drop table grade;
```

期待输出：

| id | max_score | min_score | sum_score |

| 1 | 99.000000 | 92.000000 | 286.000000 |

| 2 | 93.500000 | 88.500000 | 271.000000 |

| 3 | 94.500000 | 89.500000 | 276.500000 |

| id | max_score |

| 1 | 100.000000 |

| id | max_score | min_score |

| 1 | 100.000000 | 92.000000 |

| 2 | 93.500000 | 88.500000 |

| 3 | 94.500000 | 89.500000 |

| course | row_num | student_num | top_score | lowest_score |

| DataStructure | 3 | 3 | 95.000000 | 93.500000 |

| ComputerNetworks | 3 | 3 | 99.000000 | 88.500000 |

| C++ | 3 | 3 | 92.000000 | 89.000000 |

| ParallelCompute | 1 | 1 | 100.000000 | 100.000000 |

测试点 3：健壮性测试

测试示例:

```
create table grade (course char(20),id int,score float);
insert into grade values(‘DataStructure’,1,95);
insert into grade values(‘DataStructure’,2,93.5);
insert into grade values(‘DataStructure’,3,94.5);
insert into grade values(‘ComputerNetworks’,1,99);
insert into grade values(‘ComputerNetworks’,2,88.5);
insert into grade values(‘ComputerNetworks’,3,92.5);
– SELECT 列表中不能出现没有在 GROUP BY 子句中的非聚集列
select id , score from grade group by course;
– WHERE 子句中不能用聚集函数作为条件表达式
select id, MAX(score) as max_score where MAX(score) > 90 from grade group by id;
```

期待输出：

failure

failure

5 不相关子查询

题目描述

不相关子查询指：在一个带有子查询的查询语句中，子查询的查询条件不依赖于父查询。不相关子查询是一类较简单的子查询，可以先执行子查询，然后执行父查询，使用递归的方式处理。本任务要求参赛队伍实现不相关子查询并支持以下两类操作：

1.带有比较运算符的标量子查询

带有比较运算符的子查询是指父查询与子查询之间用比较运算符进行连接，支持>、<、=、>=、<=、!=六种比较运算符。标量子查询(Scalar Subquery)是指查询返回结果为单列单行的子查询。注意需要对如下情况进行处理：

①子查询结果为多行或多列；

②子查询结果与比较对象类型不一致。

2.带有 IN 谓词的子查询

子查询的结果是一个集合，本题目不需要考虑 IN 谓词后的子查询返回结果过大而需要将中间结果进行持久化的情况。

测试示例：

测试点 1：与子查询结果进行比较

测试示例：

create table grade (name char(20),id int,score float);

insert into grade values(‘tom’,1,92);

insert into grade values(‘jack’,2,89);

insert into grade values(‘mary’,3,89.5);

select id from grade where score = (select MAX(score) from grade);

select id from grade where score < (select MAX(score) from grade);

select id from grade where score > (select MIN(score) from grade);

期待输出：

| id |

| 1 |

| id |

| 2 |

| 3 |

| id |

| 1 |

| 3 |

测试点 2：带有 IN 谓词的子查询

测试示例：

create table grade (name char(20),id int,score float);

insert into grade values(‘tom’,1,92);

insert into grade values(‘jack’,2,89);

insert into grade values(‘mary’,3,89.5);

select id from grade where name in (select name from grade);

期待输出：

| id |

| 1 |

| 2 |

| 3 |

测试点 3：健壮性测试

测试示例：

create table grade (name char(20),id int,score float);

insert into grade values(‘tom’,1,92);

insert into grade values(‘jack’,2,89);

insert into grade values(‘mary’,3,89.5);

select id from grade where score = (select score from grade);

select id from grade where name = (select MAX(score) from grade);

期待输出：

failure

failure

6 归并连接算法

题目描述

归并连接算法是一种经典的连接算法，在没有建立索引的情况下，归并连接首先将两张表按连接属性排序，然后用一次合并操作将两个有序表合并，从而完成连接操作。进一步，如果两张表都已经在连接属性上建立索引，则可以省略排序步骤，直接按索引序合并两个表。

在本题目中，你需要实现归并连接算法，同时考虑未建立索引和已经建立索引的情况，具体的说：

在未建立索引时，归并连接的步骤如下：

排序阶段：将需要连接的两个关系表分别读入内存，并分别在内存中根据连接属性做排序，分别得到有序的关系表。

合并结果：合并两个有序的关系表。分别按连接属性扫描两个有序关系表，合并输出结果。

提示：在排序阶段，如果单个关系表过大，而无法全量放入内存，则需要使用外部排序的方式。关于外部排序的详细内容可以参考教材《数据库管理系统原理与实现》第七章查询处理。在连接字段上建立索引的情况下，可以省略排序阶段，直接按索引序合并两个关系表。

测试说明：

本题包含两个测试点：

sort_merge_join_small_data_test：数据量较少的情况，需要连接的两张表都可以放入内存中。

sort_merge_join_big_data_test：数据量较大的情况，需要连接的表可能不能全量放入内存中，参赛队伍需要实现外部排序算法。

本题的测试样例仅涉及两张表的 join，在每个测试点中，我们会分别测试三种情况下的数据库连接时间，在归并连接时（即后两种情况下），参赛队伍应将排序结果写入 sorted_results.txt 中，顺序从小到大

（注意：我们对参赛队伍的有序关系子表的实现方式不做要求，但是需要将有序子表按统一输出格式输出到 sorted_results.txt 中，sorted_results.txt 仅做测试使用，输出格式见测试方案）

1. 未建立索引时的嵌套循环连接时间 t1

2. 未建立索引时的归并连接时间 t2

3. 已经建立索引时的归并连接时间 t3

要通过测试点，需要参赛队伍的连接结果正确，排序的中间结果正确，并且已建立索引时的归并连接时间 t3 不超过 t2 和 t1 的 70%

```
# 创建表
create table item (i_id int, i_im_id int, i_name char(24), i_price float, i_data char(50));
create table stock (s_i_id int, s_w_id int, s_quantity int, s_dist_01 char(24), s_dist_02 char(24), s_dist_03 char(24), s_dist_04 char(24), s_dist_05 char(24), s_dist_06 char(24), s_dist_07 char(24), s_dist_08 char(24), s_dist_09 char(24), s_dist_10 char(24), s_ytd float, s_order_cnt int, s_remote_cnt int, s_data char(50));
# 插入数据
insert …
# 嵌套循环连接
SET enable_nestloop = true;
SET enable_sortmerge = false;
select * from item, stock where s_i_id = i_id order by i_id;
# 未建立索引时的归并连接
SET enable_nestloop = false;
SET enable_sortmerge = true;
select * from item, stock where s_i_id = i_id order by i_id;
# 建立索引之后的归并连接
create index item(i_id);
create index stock(s_i_id);
select * from item, stock where s_i_id = i_id order by i_id;
```

7 事务控制语句

题目描述：

系统需要支持显示开启事务（begin）、提交事务（commit）、回滚事务（abort）三条事务控制语句，显式事务中只包含增删改查四种语句，不包含 DDL 语句。在大赛提供的框架中，单条语句被包装成一个单独的事务进行执行，参赛队伍可以自行更改。在本任务中，不需要考虑并发事务的执行，测试数据中不包含并发事务。

在完成本任务之前，参赛队伍可以首先阅读项目结构文档中事务管理器的相关说明，以及代码框架中 src/transaction 文件夹下的代码文件。

提示：功能题目对代码框架没有限制，参赛队伍可以在原有框架的基础上进行实现，也可以修改、增添、删除数据结构及接口，或者对框架进行重构。

测试示例：

本测试通过包含四个测试点，分别测试有索引和无索引情况下事务的提交与回滚，其中有索引的测试数据中包含时间类型，但不包含不合法的时间类型，测试数据为 TPC-C 中的 NewOrder 事务。测试语句格式如下：

create table student (id int, name char(8), score float);

insert into student values (1, ‘xiaohong’, 90.0);

begin;

insert into student values (2, ‘xiaoming’, 99.0);

delete from student where id = 2;

abort;

select * from student;

期待输出：

| id | name | score |

| 1 | xiaohong | 90.000000 |

8 冲突可串行化

题目描述 系统需要支持并发事务，参赛队伍需要实现强严格两阶段封锁协议（Strong Strict Two-Phase Locking,SS2PL）来保证可串行化（Serializable）隔离级别。

使用 SS2PL 协议进行并发控制可能会导致并发事务死锁（deadlock）。为了避免出现死锁，参赛队伍需要按照 wait-die 策略实现死锁预防（deadlock prevention）。wait-die 策略的具体描述如下：系统为每个事务设置一个固定的优先级，对于任意两个事务，较早开始的事务优先级更高。在 wait-die 策略中，当一个事务 T1 申请某一个数据项上的锁时，如果持有该锁或已经在等待该锁的事务 T2 优先级比事务 T1 低，则事务 T1 等待事务 T2 释放该锁；否则，事务 T1 立刻回滚。

注：为避免幻读数据异常，对于插入操作，在本题中允许使用表级排他锁。关于冲突可串行化实现技术的更多描述，请参见教材《数据库管理系统原理与实现》。

测试示例：

本测试判断系统是否会出现五种数据异常，包括脏写（dirty write)、脏读(dirty read)、丢失更新(lost update)、不可重复读(unrepeatable read)、幻读(phantom)。

例如，对脏读数据异常进行如下测试：

create table concurrency_test (id int, name char(8), score float);

insert into concurrency_test values (1, ‘xiaohong’, 90.0);

insert into concurrency_test values (2, ‘xiaoming’, 95.0);

insert into concurrency_test values (3, ‘zhanghua’, 88.5);

事务 1:

t1a begin;

t1b update concurrency_test set score = 100.0 where id = 2;

t1c abort;

事务 2:

t2a begin;

t2b select * from concurrency_test where id = 2;

t2c commit;

事务 3：

t3a select * from concurrency_test where id = 2;

输入顺序：t2a t1a t1b t2b t1c t2c t3a

注意：为保证操作序列按照指定顺序调度，参赛队伍需要保证每条 SQL(除因 wait-die 策略等待之外)的执行时间小于 300ms.

期望结果：

+————+————+————+

| id | name | score |

+————+————+————+

| 2 | xiaoming | 95.000000 |

+————+————+————+

Total record(s): 1

+————+————+————+

| id | name | score |

+————+————+————+

| 2 | xiaoming | 95.000000 |

+————+————+————+

Total record(s): 1

9 细粒度锁-间隙锁

题目描述

在冲突可串行化题目中，使用表级锁会降低系统并发度，影响系统性能；引入行锁可以提高事务并发能力，但是可能会出现幻读的问题。引入间隙锁（gap lock）可以对查询中指定范围的已存在或待插入记录进行加锁，因此可以避免幻读问题；此外，引入间隙锁不影响范围之外数据的加减锁。这种细粒度的锁管理机制可以缓解表锁带来的性能下降问题。

间隙锁是一种加在两个索引键之间的锁，或者加在第一个索引键之前，或最后一个索引键之后的间隙。关于间隙锁的描述可参考 MySQL 的间隙锁说明。https://dev.mysql.com/doc/refman/8.0/en/innodb-locking.html

测试示例：

create table gap_lock_test (id int, name char(8), score float);

create index gap_lock_test (id);

insert into gap_lock_test values (1, ‘xiaohong’, 90.0);

insert into gap_lock_test values (3, ‘xiaoming’, 95.0);

insert into gap_lock_test values (5, ‘zhanghua’, 88.5);

insert into gap_lock_test values (7, ‘zhanglin’, 81.0);

insert into gap_lock_test values (9, ‘huangdun’, 92.0);

事务 1:

t1a begin;

t1b Select * from gap_lock_test where id > 2 and id < 6；

t1c Select * from gap_lock_test where id > 2 and id < 6；# 检查是否有幻读现象

t1d Select * from gap_lock_test; # 检查事务 3 是否插入成功

t1e commit;

t1f Select * from gap_lock_test; # 检查一致性

事务 2:

t2a begin;

t2b insert into gap_lock_test values (4, ‘yangming’, 90.0);

t2c commit;

事务 3:

t3a begin;

t3b insert into gap_lock_test values (0, ‘lihuahua’, 75.0);

t3c commit;

输入顺序：t2a t3a t1a t1b t2b t3b t3c t1c t1d t1e t2c t1f

注意：为保证操作序列按照指定顺序调度，参赛队伍需要保证每条 SQL(除因 wait-die 策略等待之外)的执行时间小于 300ms.

期望结果：

+————+————+————+

| id | name | score |

+————+————+————+

| 3 | xiaoming | 95.000000 |

| 5 | zhanghua | 88.500000 |

+————+————+————+

Total record(s): 2

+————+————+————+

| id | name | score |

+————+————+————+

| 3 | xiaoming | 95.000000 |

| 5 | zhanghua | 88.500000 |

+————+————+————+

Total record(s): 2

+————+————+————+

| id | name | score |

+————+————+————+

| 0 | lihuahua | 75.000000 |

| 1 | xiaohong | 90.000000 |

| 3 | xiaoming | 95.000000 |

| 5 | zhanghua | 88.500000 |

| 7 | zhanglin | 81.000000 |

| 9 | huangdun | 92.000000 |

+————+————+————+

Total record(s): 6

+————+————+————+

| id | name | score |

+————+————+————+

| 0 | lihuahua | 75.000000 |

| 1 | xiaohong | 90.000000 |

| 3 | xiaoming | 95.000000 |

| 4 | yangming | 90.000000 |

| 5 | zhanghua | 88.500000 |

| 7 | zhanglin | 81.000000 |

| 9 | huangdun | 92.000000 |

+———————+———————+———————+

Total record(s): 7

10 基于静态检查点的故障恢复

题目描述：

本题目要求实现基于静态检查点的系统故障恢复。包含以下功能：

1. 日志管理器

2. WAL 机制

3. 基础故障恢复

4. 语法功能：create static_checkpoint;命令创建静态检查点

5. 基于静态检查点的系统故障恢复

基础故障恢复功能需要实现日志管理器，在系统运行过程中把写操作的日志通过日志缓冲区写入磁盘中，参赛队伍需要实现 WAL 机制（Wirte-Ahead Log，即先写日志后写数据）和 REDO/UNDO 日志。在系统发生故障并重启之后，系统可以通过 REDO/UNDO 日志对系统进行故障恢复，让数据库系统恢复到一致性状态，基础故障恢复在系统重启恢复时从磁盘中的第一条日志开始扫描。

基于检查点的故障恢复是在基础故障恢复的一个优化，在创建检查点之后，系统故障恢复时从最新的一个检查点开始扫描日志。

具体的说，检查点技术是将数据库中的脏数据（即缓冲区中已经被修改但还没有写入磁盘的数据）写入磁盘，同时更新数据库的控制文件和日志文件，以便在数据库恢复时能够正确地恢复数据。检查点有多种实现方法，在本题目中，要求参赛队伍实现静态检查点。创建静态检查点时，系统需要停止接收新事务、停止正在运行的事务，并将当前数据库缓冲区中的脏页和日志刷入磁盘。当系统崩溃重启后，能够找到最近的检查点，并从该检查点开始将数据库恢复到一致性状态。

创建静态检查点的具体步骤如下：

（1）停止接收新事务和正在运行事务

（2）将仍保留在日志缓冲区中的内容写到日志文件中；

（3）在日志文件中写入一个“检查点记录”；

（4）将当前数据库缓冲区中的内容写到数据库中；

（5）把日志文件中检查点记录的地址写到“重新启动文件”中。

使用静态检查点做故障恢复的步骤如下：

（1）在重新开始文件中找到最后一个检查点在日志文件中的地址。将创建检查点时刻的所有事务加入 undo list，而 redo list 暂时为空

（2）从检查点开始扫描各个日志，如果有新开始的事务则加入 undo list，如果有事务提交，则从 undo list 删除，并加入 redo list

（3）对于 undo list 中的所有事务做 undo 操作，对于 redo list 中的事务做 redo 操作

关于静态检查点的详细内容可以参考教材《数据库管理系统原理与实现》第 12 章故障恢复

测试示例（详细测试内容见本题目测试方案）：

不带检查点的系统故障恢复：

```
create table t1 (id int, num int);
begin;
insert into t1 values(1, 1);
commit;
begin;
insert into t1 values(2, 2);
crash
…
重启系统
select * from t1;
```

带检查点的系统故障恢复：

```
create table t1 (id int, num int);
begin;
insert into t1 values(1, 1);
commit;
create static_checkpoint;
begin;
insert into t1 values(2, 2);
crash
…
重启系统
select * from t1;
```