数据库设计与实现文档

1 设计思路

1.1 存储管理

1.1.1 磁盘管理

磁盘管理负责数据库系统中文件的基本操作，提供对磁盘文件的创建、读取、写入等功能。具体功能包括：

1. **创建文件**：创建一个新的数据库文件。
2. **打开文件**：打开现有的数据库文件，以便进行读写操作。
3. **关闭文件**：在文件操作完成后关闭文件，释放资源。
4. **销毁文件**：删除数据库文件及其相关资源。
5. **写入文件**：将数据写入文件中指定的位置，文件以页的方式进行管理。
6. **读取文件**：从文件中指定位置读取数据，按页读取。

为了与缓冲池管理模块协作，所有的读写操作均以页为单位执行。

1.1.2 缓冲池管理

缓冲池管理模块为数据库文件提供分页机制下的内存缓存，以提高数据访问的效率。主要功能包括：

1. **分配新页**：在缓冲池中为文件分配一个新的数据页。
2. **获取页**：从缓冲池中获取指定页，如果页不在缓冲池中，则从磁盘中读取。
3. **淘汰页**：根据缓冲池替换策略（如LRU）淘汰不常用的页，以释放内存。
4. **更新页**：在缓冲池中更新页内容，并标记该页为“脏页”。
5. **取消固定页**：取消对指定页的固定状态，使其可以被替换。
6. **删除页**：从缓冲池中删除指定的页。
7. **将一页写入磁盘**：将缓冲池中的脏页内容写回到磁盘文件。
8. **将所有页写入磁盘**：将缓冲池中的所有脏页内容刷新到磁盘中，确保数据持久化。

在选择淘汰页的过程中，需要考虑缓冲池替换策略，实现以下功能：

1. 淘汰页
2. 固定页
3. 取消固定页

1.1.3 记录管理

记录管理模块负责对数据库表中的记录进行精细化管理，提供以下功能：

1. **获取记录**：从表中读取指定位置的记录。
2. **插入记录**：在表的空闲位置或指定位置插入新记录；如果是事务回滚场景，需要在指定位置插入。
3. **删除记录**：删除表中的指定记录，并更新相关索引。
4. **更新记录**：对表中的指定记录进行更新，同时更新相关索引。

记录管理模块确保数据的插入、删除、更新操作能够正确执行，并维护表的完整性。

1.1.4 记录遍历

为了支持SQL查询执行中的记录遍历操作，记录管理模块还需要提供遍历表中所有记录的接口。这一接口将用于实现查询操作中的记录筛选、投影和连接等功能。

1.2 查询执行

1.2.1 DDL语句

DDL（数据定义语言）语句主要包括以下操作：

1. **创建表**：创建表文件，初始化表的元数据信息，并将表添加到数据库的打开表列表中。
2. **删除表**：关闭表文件，删除关联的索引文件，删除表文件，并更新数据库元数据。

这些操作确保表的创建和删除操作对系统资源进行适当的管理和更新。

1.2.2 DML语句

DML（数据操作语言）语句用于操作表中的数据，包括：

1. **INSERT语句**：构建插入数据的二进制表示，将其写入相应的表和索引中。
2. **DELETE语句**：从表中查询符合条件的记录，并删除记录及其相关索引。
3. **UPDATE语句**：查询符合条件的记录，更新记录及其相关索引。

这些语句在执行过程中需要考虑数据的一致性和完整性。

1.2.3 DQL语句

DQL（数据查询语言）语句主要用于查询操作，`SELECT`语句的实现包括以下算子：

1. **记录遍历算子**：从表中检索符合查询条件的记录。
2. **投影算子**：对检索到的记录进行投影，仅返回用户指定的字段。
3. **连接算子**：当查询涉及多个表时，通过连接算子合并表中的相关记录。

这些算子协同工作，实现复杂的查询功能。

1.3 唯一索引

1.3.1 索引的创建、删除和展示

唯一索引确保表中的某些字段的值在表中是唯一的。相关操作包括：

1. **创建索引**：创建并打开索引文件，将表中的现有数据插入索引中。
2. **删除索引**：关闭并删除索引文件，更新表的索引信息。
3. **展示索引**：遍历并展示表中的所有索引，以便查看索引的状态。

1.3.2 索引查询

系统实现了基于B+树的索引结构，主要功能包括：

1. **查找大于或等于指定键的叶子节点**。
2. **查找大于指定键的叶子节点**。
3. **插入指定键**：将新键插入索引中。
4. **删除指定键**：从索引中删除指定键。

索引查询优化了数据检索的效率，尤其是在处理大量数据时，能够快速定位目标记录。

1.3.3 索引维护

唯一索引要求插入或更新操作不能违反唯一性约束，相关功能包括：

1. **插入键时**：如果键已存在，抛出异常，避免违反唯一性约束。
2. **更新键时**：如果更新后的键已存在则抛出异常，但如果新旧键值相同则不抛出异常。

1.4 聚合函数与分组统计

1.4.1 聚合函数

数据库系统中实现了多个常用的聚合函数，通过新增聚合算子实现：

1. **MAX**：取出表中的记录，统计指定字段的最大值。
2. **MIN**：取出表中的记录，统计指定字段的最小值。
3. **COUNT**：取出表中的记录，计算指定字段的总记录数。

1.4.2 分组统计

分组统计是聚合函数的一部分，通过在聚合算子中按指定字段进行分组，然后对每组进行统计，计算所需的聚合值。

1.5 不相关子查询

不相关子查询需要修改查询条件的右值，新增子查询和集合类型。在使用到条件判断的地方都需要事先判断查询条件的右值是否为子查询，若是则优先执行子查询。

1.6 事务控制语句

1.6.1 开启事务

开启事务时，需要分配一个新的事务`id`，并将该事务加入全局事务表中，以便系统对事务进行管理。

1.6.2 提交事务

提交事务时，系统需要执行所有挂起的写操作，释放事务持有的锁，并刷新事务日志，确保数据的持久性。

1.6.3 回滚事务

回滚事务需要取消所有未提交的写操作，恢复表和索引文件的原始状态，然后释放所有锁和资源，最后刷新事务日志。

1.7 冲突可串行化

系统实现了基于表级的共享锁和排他锁机制：

1. **查询时**：获取共享锁，确保读操作的并发安全。
2. **插入、删除、更新时**：获取排他锁，防止数据冲突。

系统采用`wait-die`策略来处理死锁问题。如果一个较晚启动的事务在等待一个较早启动的事务释放锁，则较晚的事务将被回滚，以避免死锁。

1.8 基于静态检查点的故障恢复

1.8.1 故障恢复

故障恢复分为三个阶段：

1. **日志分析**：从日志文件中读取并解析日志记录，按照事务划分日志。
2. **REDO**：按照事务开始的顺序重放日志，以确保所有提交的操作都被执行。
3. **UNDO**：回滚所有未提交或已经回滚的事务，确保数据库状态的一致性。

1.8.2 静态检查点

创建静态检查点时，需要先获取事务锁，阻止新事务的分配。然后等待所有现有事务完成，将日志文件和缓冲池中的内容写入磁盘。最后，写入检查点日志并记录检查点在日志文件中的偏移量，以便后续的恢复操作。

2 系统框架

本数据库系统采用模块化设计，将各项功能划分为多个独立的模块。这些模块共同协作，完成从SQL解析到查询执行、事务管理、并发控制、故障恢复等数据库的核心功能。

本数据库系统框架详见框架图。

2.1 系统架构概述

整个系统的架构分为多个主要模块，包括：

- **存储管理模块**
- **索引与并发控制模块**
- **查询处理与执行模块**
- **事务管理与故障恢复模块**

各模块之间通过清晰的接口进行交互，保证系统的高内聚性和低耦合性。

2.2 存储管理模块

存储管理模块负责数据库数据的持久化和内存缓存管理。它包含以下子模块：

- **文件存储组织**：管理数据库的数据文件和页面。包括 `DiskManager` 和 `Page` 类，它们负责数据的物理存储和页面的读取与写入。`DiskManager` 提供了对磁盘文件的低级别操作，而 `Page` 类则抽象出一个逻辑页面的概念，使得数据操作更加直观。
- **缓冲区管理**：实现了对内存中的数据页的管理，通过 `BufferPoolManager` 类维护一个 LRU（最近最少使用）缓存，以提高数据访问的性能。`BufferPoolManager` 负责在内存中缓存热数据，并通过 LRU 策略在缓存空间有限的情况下有效替换不常使用的数据页。
- **记录存储组织**：管理表中的记录操作，为上层模块提供数据的插入、删除、修改和读取接口。该子模块封装了对记录的底层操作，使得上层查询处理模块无需关心数据的物理存储细节。

2.3 索引与并发控制模块

索引与并发控制模块确保在多线程环境下数据库操作的正确性：

- **索引管理**：系统通过实现 B+ 树等索引结构来加速数据检索。索引管理模块不仅提升了数据查询的效率，还为范围查询、排序等操作提供了支持。通过 B+ 树索引结构，系统能够高效地找到符合条件的记录。
- **锁管理**：为事务提供行级、表级等多种粒度的锁，确保并发事务之间的一致性和隔离性。锁管理模块通过 `LockManager` 实现，支持多种锁类型（如共享锁、排它锁），并且能够处理死锁检测和锁升级等复杂的并发控制问题。

2.4 查询处理与执行模块

查询处理与执行模块负责 SQL 查询的解析、优化和执行。该模块的核心流程如下：

- **SQL 解析**：接收 SQL 查询语句后，解析生成抽象语法树（AST）。SQL 解析器将用户输入的 SQL 文本解析为系统可以理解的内部表示形式，即抽象语法树（AST），为后续的查询优化和执行奠定基础。
- **语义分析与查询优化**：通过分析语法树生成查询计划树，并进行查询优化，优化后的查询计划能够以更高效的方式获取数据。查询优化器通过代价模型选择最优的查询执行计划，以减少查询的执行时间和资源消耗。优化策略包括连接顺序的调整、索引的选择以及子查询的优化等。
- **查询执行**：根据生成的执行计划，通过调用存储管理、索引管理和事务管理等模块，完成查询的实际执行，并返回结果集。查询执行模块通过执行器（Executor）将优化后的查询计划转化为具体的操作指令，逐步执行查询，获取数据，并将结果返回给用户。

2.5 事务管理与故障恢复模块

事务管理模块提供对事务的支持，包括开始、提交、回滚操作，确保数据库在出现故障时能够恢复到一致的状态：

- **事务管理**：负责管理事务的生命周期，保证事务的ACID特性。通过 `TransactionManager` 实现事务的开始、提交和回滚操作，确保事务的原子性、一致性、隔离性和持久性。
- **日志管理与故障恢复**：通过 `LogManager` 管理事务日志，使用 WAL（Write-Ahead Logging）技术记录事务操作，以支持系统的故障恢复。在事务提交前，系统会先将事务的操作记录写入日志，确保即使在系统故障时，也能通过重放日志恢复数据库的一致状态。

该模块在系统出现崩溃或其他故障时，可以通过分析日志文件，恢复未完成的事务并回滚未提交的修改，从而保障数据的一致性和完整性。

3 实现重点

3.1 `Value`

在框架给定的 `Value` 结构中，我们需要实现运算符重载以支持 `Value` 的比较操作。以下是判断两个 `Value` 是否相等的实现：

```
bool operator==(const Value &rhs) const {
    if (type != rhs.type) throw IncompatibleTypeError(coltype2str(type), coltype2str(rhs.type));
    switch (type) {
        case TYPE_INT:
            return int_val == rhs.int_val;
        case TYPE_FLOAT:
            return float_val == rhs.float_val;
        case TYPE_STRING:
            return strncmp(str_val.c_str(), rhs.str_val.c_str(), str_len) == 0;
        default:
            throw InternalError("Unexpected value type");
    }
}
```

还需要实现值的类型转换，以支持从表中加载数据。下面是字符串类型转换为整型或浮点型的实现：

```
void value_cast(ColType new_type) {
    if (type == new_type) return;
    if (type == TYPE_INT && new_type == TYPE_FLOAT) {
        float_val = int_val;
    } else if (type == TYPE_FLOAT && new_type == TYPE_INT) {
        int_val = float_val;
    } else if(type == TYPE_STRING && new_type == TYPE_INT) {
        sscanf(str_val.c_str(), "%d", &int_val);
    } else if(type == TYPE_STRING && new_type == TYPE_FLOAT) {
        sscanf(str_val.c_str(), "%f", &float_val);
    } else {
        throw IncompatibleTypeError(coltype2str(type), coltype2str(new_type));
    }
    type = new_type;
}
```

3.2 `Condition`

在 `Condition` 类中，我们给定了以下属性：

```
struct Condition {
    TabCol lhs_col;   // left-hand side column
    CompOp op;        // comparison operator
    TabCol rhs_col;   // right-hand side column
    Value rhs_val;    // right-hand side value
    std::shared_ptr<Query> rhs_query;
    std::shared_ptr<Plan> rhs_plan;
    std::shared_ptr<PortalStmt> rhs_portal;
    std::set<Value> rhs_set;
    CondRhsType rhs_type;
    std::vector<ColMeta>::const_iterator lhs_match_col;
    std::vector<ColMeta>::const_iterator rhs_match_col;
}
```

为了适配子查询，我们增加了中间结果的存放位置`rhs_query`、`rhs_plan`、`rhs_portal`。在子查询执行完毕后，该条件的右值会退化为一个具体的`Value`或`Value`的集合。

对于条件查询我们做了以下两个优化：

1. **减少类型转换次数**：在开始检索记录之前，首先检查条件的两侧类型是否相同。如果不同但类型转换合法，则将右值转换为与左值相同的类型。
2. **减少查询过程中对列的获取**：在条件中增加`lhs_match_col`和`rhs_match_col`，用于记录左值和右值在表中的列的迭代器。

3.3 B+树索引

在我们实现的 B+ 树中，内部节点的键和值的个数相等。

B+ 树索引涉及到的方法比较多，在这里列出比较重要的两种方法。

3.3.1 插入数据

插入数据的方法实现如下：

1. 获取根节点的锁。
2. 查找应插入的叶节点。
3. 将给定键值对插入叶节点。
4. 如果插入的位置是第一个位置，则需要循环向上更新父节点的第一个键值。
5. 如果插入后叶节点已满，则分裂该节点。

3.3.2 删除数据

删除数据的方法实现如下：

1. 获取根节点的锁。
2. 查找删除的键所在叶节点。
3. 删除键值对。

4. 如果删除的是第一个键，则需要循环向上更新父节点的第一个键值。
5. 对叶节点进行合并或再分配操作。

3.4 事务控制

在事务控制中，事务回滚尤为重要。为了实现事务回滚，我们需要在插入、删除、更新数据之前记录操作，并将其加入到事务的写操作集合中。

写操作记录的内容包括：

1. **插入数据**：记录插入的表名和插入位置的`rid`。
2. **删除数据**：记录删除的表名、删除位置的`rid`及删除的记录内容。
3. **更新数据**：记录更新的表名、更新位置的`rid`及更新前的记录内容。

事务回滚分为两个步骤：回滚表中的记录和回滚索引中的记录。

为正确恢复索引数据，索引恢复前需要将删除的记录插入到表中的原位置；索引恢复后，针对插入操作需删除该记录，针对更新操作需更新该记录。

3.5 冲突可串行化

3.5.1 表级锁

实现表级排他锁的步骤如下：

1. 检查当前事务是否持有该表的共享锁；如果有则等待其他事务释放锁，如果只有当前事务持有共享锁则升级为排他锁。
2. 如果当前事务没有持有该表的锁，则检查是否有其他事务持有该表的锁；如果没有，则获取锁，否则进入等待队列。
3. 如果等待队列中有比当前事务更早开始的事务，则抛出事务回滚异常。
4. 获取锁后，将其加入事务的锁集合中。

对于表级共享锁的实现步骤如下：

1. 如果已持有共享锁或排他锁，则无需加锁。
2. 如果表只有共享锁，则直接获取锁。
3. 如果表被加了排他锁，则进入等待队列。
4. 如果等待队列中有比当前事务更早开始的事务，则抛出事务回滚异常。
5. 获取锁后，将其加入事务的锁集合中。

为了简化代码复杂度，我们设计了**锁的相容性矩阵**，每次判断是否能够加锁时，按照所需的锁与当前已加的锁在矩阵中查询其相容性。

3.5.2 释放锁

释放锁的流程如下：

1. 获取该锁的等待队列。
2. 将当前项从等待队列中移除。
3. 如果还有其他事务持有锁，则直接返回。
4. 如果没有其他事务持有锁，为等待队列中的第一个事务赋予锁，其他持有与该事务相容的锁的事务均被赋予锁，唤醒这些线程，更新队列的加锁模式。

3.6 故障恢复

3.6.1 日志分析

我们设计了如下成员变量：

```
std::unordered_map<PageId, RedoLogsInPage> redo_logs_map_;    // 记录每个page上需要redo的操作
std::map<lsn_t, std::shared_ptr<LogRecord>> log_records_;                    // 记录每个lsn对应的日志记录
std::unordered_map<txn_id_t, lsn_t> redo_txn_;                // 需要redo的事务，最后一条redo的日志的lsn
std::unordered_map<txn_id_t, lsn_t> undo_txn_;                          // 需要undo的事务
int last_checkpoint_;                              // 最后一个checkpoint的lsn
std::vector<txn_id_t> txn_list_;                        // 事务列表
```

日志分析的流程如下：

1. 从日志文件起始位置开始，反序列化出日志头的信息。
2. 根据日志头信息判断日志类型，根据日志类型执行不同操作。其中比较重要的有：
   (a) `begin`：将事务加入`undo`事务列表中。
   (b) `commit`：将事务从`undo`事务列表中删除，加入`redo`事务列表中。
3. 将事务执行的操作按照日志顺序分配到各个页面上。

3.6.2 重做事务

重做每个页面上的插入、删除、更新操作。

3.6.3 回滚事务

对于回滚事务表中的事务，按照事务开始顺序的逆序回滚事务，更新对应的记录和索引。

4 代码注释

我们在重要API对应的方法的实现前会添加该方法的注释，格式如下：

```
/**
 * @brief 方法功能介绍
 *
 * @param 参数介绍
 * @return 返回值介绍
 * @note 其他需要注意的点
 */
```

5 遇到问题

5.1 内存泄漏

在不断接受事务的过程中，由于部分使用`new`分配出去的空间没有释放，会导致数据库内存占用不断升高，经过排查后找出比较严重的问题：

1. 事务修改记录的操作没有释放
2. 在索引查询中部分结点管理没有释放
3. 在日志管理中没有释放表名所占空间
4. 事务结束后也一直存放在事务管理表中没有释放

在后续的过程中我们对以上问题涉及的对象在适当的位置进行释放。

5.2 删除表后新建表访问内容不正确

由于删除表会关闭文件，新建表会打开文件，这两个文件对应的`fd`可能相同，导致在缓冲池中可能会访问到错误的内容。因此在删除表后，需要在缓冲池中删除对应的文件页。

6 性能优化

6.1 `count(*)` 优化

我们对全表的记录条数统计进行优化，直接对表中所有页面的记录数进行求和，省去遍历表中记录的步骤。

6.2 锁的优化

我们在每个锁都设计了自己独有的互斥量，在通过锁的全局互斥量获取到锁的等待队列后，获取该锁的互斥量，释放锁的全局的互斥量。

6.3 其他优化

`update`更新索引时，如果出现重复键值，则抛出异常，执行事务回滚过程，省去在插入记录之前检查键在索引中是否存在的过程。