数据库系统project报告

2023-2024学年第2学期（CST21118）

|  |  |
| --- | --- |
| 数据库系统project任务书 | |
| 名称 | Update设计与实现 |
| 类型 | □验证性 □设计性 ☑ 综合性 |
| 内容 | 1. 基于MiniOB理解数据库系统的SQL引擎执行原理。 2. 设计并实现MiniOB的update功能。 |
| 要求 | 1. 设计方案要合理； 2. 能基于该方案完成系统要求的功能； 3. 设计方案有一定的合理性分析。 |
| 任务时间 | 2024年5月8日至2024年5月29日 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 小组成员 | | | | | | |
|  | |  | |  | |  |
|  | |  | |  | |  |
| 项目评分表 | | | | | | |
| 序号 | 评分项 | | 比例 | | 得分 | |
| 1 | Project内容完成情况 | | 50% | |  | |
| 2 | 工具熟练度 | | 30% | |  | |
| 3 | 团队协作 | | 20% | |  | |
| 项目总得分： | | | | | | |

课程项目评分标准（总分10分）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 考核内容（权重） | 评分标准 | | | |
| 10 | 8-9 | 6-7 | 0-5（不合格） |
| 1. 完成project的实现（50%） | 按时完成project分析、设计、实现等核心内容。 | 完成project分析、设计、实现等核心内容，但存在少量错误。 | 基本project分析、设计、实现等核心内容，但存在较多错误。 | 未完成project内容。 |
| 2.熟练使用设计工具（30%） | 熟练掌握实验设计相关的软件工具，完成数据库设计、编码等工作。 | 较为掌握实验设计相关的软件工具，完成数据库设计、编码等工作。 | 能够使用实验设计相关的软件工具，完成数据库设计、编码等工作。 | 对于实验设计软件和工具使用不熟练。 |
| 3.团队协作（20%） | 有团队，分工合理，密切协作。 | 有团队，分工合理，有一定协作。 | 有团队，分工不合理，无协作。 | 无团队，无协作。 |

|  |
| --- |
| 1. 小组分工   ：分析MiniOB的SQL引擎执行原理和画出对应过程及描述。  ：设计update的实现方案和设计方案图。  ：实现update的代码并进行功能测试。  ：总结并编写实验报告。 |
| 1. 分析MiniOB的SQL引擎执行原理     图 1 SQL执行原理图  当一条 SQL 请求到达 Server 端后，经过Fast-parser和Plan cash模块，这部分的主要作用是当接收到 SQL 请求时，如果能够直接查询到其对应的 Plan cash，那么就将这个查询计划返回给 Executor，避免经过长时间的 SQL 硬解析。如果不能就要经过 Parser 模块对SQL语句进行解析。  Parser 模块主要作用是根据定义的语法规则判断一条 SQL 语句的格式是否符合对应的语法结构，并将一条 SQL 语句转化为一个语法树的结构。之后经过Resolve模块，Resolver 模块主要作用是对 Parser 模块所生成的符合语法规则的树状结构进行进一步的约束检查，还可能会提取表达式的属性。数据结构在 Resolver 层实际上是一个 Statement，这也是后续进行 Transformer 和 optimizer 改写的基础，在 Transformer 和 optimizer 层，需要关注的是火山模型的结构来优化执行计划。  随后将执行计划送达Excutor模块进行查找。Executor 模块中最经典的模型是 Volcano Model（火山模型），它是一种基于行的流式迭代模型。在火山模型中，所有的代数运算符（operator）都被看成是一个迭代器，它们都提供一组简单的接口：open()—next()—close()，查询计划树由一个个这样的关系运算符组成，每一次的 next() 调用，运算符就返回一行（Row），每一个运算符的 next() 都有自己的流控逻辑，数据通过运算符自上而下的 next() 嵌套调用而被动的进行拉取，最终找到数据。 |
| 1. 设计update的实现方案   update  图 2 update设计方案图  Update设计方案   1. SQL 语句解析:   解析SQL语句确保语句合法性，并生成语法树。对于单字段更新，解析器需要从SQL语句中提取表名、字段名、更新值以及更新条件。   1. 更新声明（Update Statement）:   生成一个表示更新操作的数据结构，它包含表名、字段名、新值以及更新条件。   1. 条件评估：   对符合更新条件的记录进行评估，确保更新操作只应用于满足条件的记录。  步骤：1解析条件：从Update语句中提取条件表达式。  2遍历记录：遍历表中的所有记录。  3评估条件：对于每条记录，使用条件表达式进行评估，确定是否满足条件。   1. 事务支持：   事务支持用于确保更新操作的原子性和一致性，并在更新执行器中使用事务。事务可以确保所有更新操作要么全部成功，要么全部失败，避免部分更新导致数据不一致。  步骤：1.开始事务：在执行更新操作之前，开启一个事务。  2.执行更新：在事务范围内执行所有更新操作。  3.提交事务：如果所有更新操作成功，提交事务。  4.回滚事务：如果在更新过程中发生错误，回滚事务，撤销已经完成的更新操作。   1. 执行器:   执行更新操作。它从存储层读取满足条件的记录，并进行更新操作，将指定字段的值替换为更新值。   1. 索引更新：   索引更新用于确保索引在记录更新后保持一致性，并在更新执行器中使用索引更新。更新记录时，如果更新的字段是索引字段，需要同时更新索引。  步骤：1.查找索引：查找与更新字段相关的索引。  2.删除旧索引：从索引中删除旧值对应的索引项。  3.插入新索引：将新值插入索引。   1. 存储层:   实际存储数据的地方，包括表、记录和索引。需要提供接口以支持记录的读取和写入。将更新后的记录写入到持久化存储，确保数据的持久性。  具体实现方案   1. SQL 解析：解析器需要解析类似如下的SQL语句：   UPDATE 表名 SET 列名 = 更新值 WHERE 条件;并输出一个UpdateStatement对象。   1. 更新声明（Update Statement）：UpdateStatement类用来保存解析后的更新操作信息。   class UpdateStatement {  public:  std::string table\_name;  std::string column\_name;  Value new\_value;  Condition condition;  // 构造函数和其他方法  };   1. 条件评估：用于确定哪些记录需要更新。   class Condition {  public:  bool evaluate(const Record& record) const {  // 假设条件是field\_name == value  Value field\_value = record.getFieldValue(field\_name);  return field\_value == value;  }  private:  std::string field\_name;  Value value;  // 其他成员变量和方法  };   1. 事务支持：用于确保更新操作的原子性和一致性，并在更新执行器中使用事务。   class Transaction {  public:  void begin() {  // 开始事务  }  void commit() {  // 提交事务  }  void rollback() {  // 回滚事务  }  };   1. 更新执行器：Update Executor负责执行更新操作。   class UpdateExecutor {  public:  UpdateExecutor(StorageManager& storage\_manager) : storage\_manager\_(storage\_manager) {}  void execute(const UpdateStatement& stmt) {  Table\* table = storage\_manager\_.getTable(stmt.table\_name);  if (!table) {  throw std::runtime\_error("Table not found");  }  for (Record& record : table->getRecords()) {  if (stmt.condition.evaluate(record)) {  record.setFieldValue(stmt.column\_name, stmt.new\_value);  table->updateRecord(record);  }  }  }  private:  StorageManager& storage\_manager\_;  };  在更新执行器中使用事务：  void UpdateExecutor::execute(const UpdateStatement& stmt) {  Transaction transaction;  transaction.begin();  try {  Table\* table = storage\_manager\_.getTable(stmt.table\_name);  if (!table) {  throw std::runtime\_error("Table not found");  }  for (Record& record : table->getRecords()) {  if (stmt.condition.evaluate(record)) {  record.setFieldValue(stmt.column\_name, stmt.new\_value);  table->updateRecord(record);  }  }  transaction.commit();  } catch (...) {  transaction.rollback();  throw;  }  }  在更新执行器中使用索引更新：  void UpdateExecutor::execute(const UpdateStatement& stmt) {  Transaction transaction;  transaction.begin();  try {  Table\* table = storage\_manager\_.getTable(stmt.table\_name);  if (!table) {  throw std::runtime\_error("Table not found");  }  Index\* index = table->getIndex(stmt.column\_name);    for (Record& record : table->getRecords()) {  if (stmt.condition.evaluate(record)) {  if (index) {  index->remove(record.getFieldValue(stmt.column\_name), record);  }  record.setFieldValue(stmt.column\_name, stmt.new\_value);  table->updateRecord(record);  if (index) {  index->insert(stmt.new\_value, record);  }  }  }  transaction.commit();  } catch (...) {  transaction.rollback();  throw;  }  }   1. 索引更新：用于确保索引在记录更新后保持一致性，并在更新执行器中使用索引更新。   class Index {  public:  void remove(const Value& value, const Record& record) {  // 从索引中删除旧值  }  void insert(const Value& value, const Record& record) {  // 向索引中插入新值  }  };   1. 存储层：需要支持记录的读取和更新，将更新后的记录写入到持久化存储，确保数据的持久性。   class Table {  public:  std::vector<Record>& getRecords() { return records\_; }  void updateRecord(const Record& record) {  // 更新存储中的记录  }  private:  std::vector<Record> records\_;  };  class Record {  public:  void setFieldValue(const std::string& field\_name, const Value& value) {  //设置字段值  }  // 其他  };  class StorageManager {  public:  Table\* getTable(const std::string& table\_name) {  // 按名称检索表  }  // 其他  };  整个设计方案中的关键点是：  1.条件评估：用于确定哪些记录需要更新。  2.事务支持：更新操作通常需要事务支持，以确保操作的原子性和一致性。  3.索引更新：如果有索引，需要确保更新记录时索引的一致性。 |
| 1. Update实现代码 |
| 1. update功能测试 |
| 1. 总结   本实验报告深入探讨了数据库系统中update功能的设计与实现。我们团队在规定的时间内，不仅完成了项目任务，还深入分析了SQL引擎的执行原理，提出了一个既合理又高效的设计方案，并成功实现了update功能。实验结果证实了设计方案的可行性和有效性，展现了团队成员的专业性和协作精神。  在技术实现方面，我们构建了一个完整的update功能框架，包括了SQL解析器、更新执行器、事务管理器和索引更新机制等关键组件。这些组件的设计和实现遵循了模块化和可扩展性原则，使得整个系统易于理解和维护。代码结构清晰，逻辑性强，为后续的优化和扩展提供了便利。  通过本次实验，团队成员不仅提升了编程技能，也加深了对数据库系统工作原理的理解。实验经验的积累，将有助于在未来面对更复杂的问题时，能够更加从容应对。 |