数据库专题训练 · Lab2

计01　容逸朗　2020010869

实验目的

1. 理解数据库日志的记录方式以及日志对于故障恢复过程的作用；
2. 初步了解 ARIES 算法的实现方式；
3. 掌握简单的日志记录和管理的方式。

基础实验内容

1. 添加记录 WAL 日志

- 首先利用 `LogManager` 的对应函数在 `table.cpp` 中记录日志，注意需要在更新数据前先写入日志；
  - 利用 `RecordFactory::StoreRecord` 取得当前数据的序列化数据；
  - 直接使用 `PageHandle::GetRaw` 函数取得旧数据记录；
- 同时，在插入记录前需要加上对应的 `Rid` ，否则利用 `LogManager` 的函数做记录时会出现错误；
- 为了得到 `InsertRecord` 时正确的插入位置，需要对 `PageHandle::Next` 函数作如下更改：

```
return Rid{page_->GetPageId().page_no, FirstFree()};
```

2. WAL 数据插入日志

- 主要是利用 `LogFactory` 的对应函数生成日志信息。
- ATT 记录同一个进程下最新的 LSN（即 Last LSN）
  - 每次操作前先从 ATT 表中取出记录，将其写入本次操作对应日志的 Prev LSN 项；
  - 然后更新本次操作的 Prev LSN 为 ATT 中对应进程的 Last LSN；
- DPT 记录了某一页面的 Rec LSN
  - 若记录在 DPT 中出现的话，就不需要改变 Rec LSN 的值
  - 否则需要把该页面的 Rec LSN 记为当前的 LSN。

3. Update 日志镜像设计

- 先存放表名 `table_name_` 的长度，然后分别记录表名、对应页面、对应槽位、操作类型。
- 然后需要根据对应的操作类型作讨论：
  - 插入：只需保留新数据
  - 删除：需要保存删除前的值
  - 更新：既要保存对应数据的旧值，也要保留新的数据。

4. 基于日志镜像的 Redo 和 Undo 操作

- 首先通过 `SystemManager` 取得当前表格，然后根据 `page_id` 取得对应页面；
- 对于 Redo 操作，若日志对应的 LSN < Page LSN 则不需要操作；
- 对于 Redo 和 Undo 过程，只需要调用 `PangHandle` 对应的函数操作即可，注意 Undo 与 Redo 的逻辑是正好相反的；
- **重点：**需要在每一条 Redo 操作后对应的页面更新 LSN，避免页面多次更新。

5. Checkpoint 日志设计

- 先储存 ATT 和 DPT 的大小，然后分别将 ATT 和 DPT 的内容序列化即可。

6. 故障恢复算法

- **Redo**
  1. 首先遍历 DPT ，找出最小的 Rec LSN；
  2. 然后 Redo 所有遇到的 Update Log；

     难点：Redo 过程需要跳过 Checkpoint LSN，否则经过 `ReadLog` 操作后会使其对应的 ATT 和 DPT 表被更新，这是不应该的。
  3. 直至到达当前 LSN `current_lsn_` 的位置则停止 Redo。
- **Undo**
  1. 遍历 ATT ，取得任意的 LSN 进行回滚，若 ATT 为空则结束。
  2. 若当前 log 的操作类型为 `BEGIN` 或 log 为空，则回到 1；
  3. 若为 `UPDATE` 则 Undo ，否则更新 Prev LSN 并回到 2。

高级功能 1: Undo 过程中系统出现异常的恢复

1. 设计方案

- 首先在每次 Undo 操作后，都插入一条新的 CLR Log:
- LSN, Prev LSN 和 XID 与其他日志相同；
  - 数据部分和 Update Log 类似，不过两者的新旧值是恰好相反的；
  - Undo Next LSN 的值为当前 Update Log 的 Prev LSN；
- 需要注意 CLR Log 只需要实现 Redo 函数，因为 CLR 是不需要 Undo 的；
  - Undo 过程中遇到 CLR 时，下一个 undo 的日志应为 CLR 对应的 undo next LSN。

2. 实现效果

- 为测试效果，我增加了新的 SQL 语句： `mycrash <db>`
  - 其作用相当于 `use <db>`，但是会在 undo 过程中随机发生 `crash`，最终返回 FAILURE。
- 对应测例如下：
  - 此测例中，首先进行数次插入、删除和更新操作，然后 `commit`；
  - 接着又进行数次插入、删除和更新操作，刷盘，再调用 `crash`；
  - 然后使用我们自定义的 `mycrash` 操作数据库，应当输出 FAILURE；
  - 最后使用 `use <database>` 恢复数据库，并打印表格中数据作比较。

```
drop database if exists dbtrain_test_lab2_advanced;
create database dbtrain_test_lab2_advanced;
use dbtrain_test_lab2_advanced;
begin;

create table persons(id int, first_name varchar(20), last_name varchar(20), temperature float);

insert into persons values(1, 'apple', 'abc', 36.5), (2, 'bob', 'sadf', 36.4), (3, 'cat', 'kasdlf', 37.2), (4, 'dsdaf', 'sdag', 35.3), (5, 'esdf', 'sdf', 36.8);
insert into persons values(6, 'fsdf', 'sdf', 36.3), (7, 'gsdf', 'sdf', 36.5), (8, 'hsdf', 'sdf', 36.4), (9, 'isdf', 'sdf', 37.2), (10, 'jsdf', 'sdf', 37.3);
insert into persons values(11, 'ksdf', 'sdf', 36.8), (12, 'lsdf', 'sdf', 36.0), (13, 'msdf', 'sdf', 36.7), (14, 'nsdf', 'sdf', 36.9), (15, 'osdf', 'sdf', 37.2);

delete from persons where id = 8;
update persons set temperature = 36.8 where id = 2;
commit;

begin;
insert into persons values(22, 'dsag', 'dsgat', 36.3);
delete from persons where id > 13;
update persons set temperature = 37.3 where id < 5;
flush;
crash;

mycrash dbtrain_test_lab2_advanced;
use dbtrain_test_lab2_advanced;
select * from persons;
```

- 实验结果如下：（这里只显示重要的部分）

```
-- 17.crash;
CRASH

-- 18.mycrash dbtrain_test_lab2_advanced;
FAILURE

-- 19.use dbtrain_test_lab2_advanced;
SUCCESS

-- 20.select * from persons;
id | first_name | last_name | temperature
1 | apple | abc | 36.5
2 | bob | sadf | 36.8
3 | cat | kasdlf | 37.2
4 | dsdaf | sdag | 35.3
5 | esdf | sdf | 36.8
6 | fsdf | sdf | 36.3
7 | gsdf | sdf | 36.5
9 | isdf | sdf | 37.2
10 | jsdf | sdf | 37.3
11 | ksdf | sdf | 36.8
12 | lsdf | sdf | 36
13 | msdf | sdf | 36.7
14 | nsdf | sdf | 36.9
15 | osdf | sdf | 37.2
```

- 多次验证下仍能输出正确结果，因此认为提高实验成功完成。

高级功能 2: 非阻塞的 Checkpoint 日志记录

1. 设计方案

- 原先的 Checkpoint Log 被分为 CheckpointBegin Log 和 CheckpointEnd Log 两部分：
  - 前者代表原来 Checkpoint 的位置，无特殊作用；
  - 后者负责 ATT 和 DPT 数据的写盘，以及保存 CheckpointBegin Log 对应的 LSN （也就是 Checkpoint LSN）；
- 每次调用 `LogManager::Checkpoint` 函数时：
  - 首先记录一条 CheckpointBegin Log；
  - 然后复制当前的 ATT 和 DPT 表；
  - 开一个新进程，并把 Checkpoint LSN 以及两个表对应的指针传入日志写回过程中；
  - 进入新进程后，先等待 400μs，然后构造一个 CheckpointEnd Log 并写回日志；
  - 最后保存 CheckpointEnd Log 对应的 LSN 为 MasterRecord；

- 注：由于现在 MasterRecord 记录的是 Checkpoint End 对应的 LSN，因此在分析阶段前要先从 CheckpointEnd Log 中取出 Checkpoint 真正的 LSN，再进行恢复。

2. 实现效果

- 为测试效果，设计测例如下：
  - 此测例中，首先进行数次插入、删除和更新操作，然后 `commit`；
  - 紧接着进行一次 `checkpoint`；
  - 接着又进行数次插入、删除和更新操作，刷盘，再调用 `crash`；
  - 最后使用 `use <database>` 恢复数据库，并打印表格中数据作比较。

```
drop database if exists dbtrain_test_lab2_advanced;
create database dbtrain_test_lab2_advanced;
use dbtrain_test_lab2_advanced;
begin;

create table persons(id int, first_name varchar(20), last_name varchar(20), temperature float);

insert into persons values(1, 'apple', 'abc', 36.5), (2, 'bob', 'sadf', 36.4), (3, 'cat', 'kasdlf', 37.2), (4, 'dsdaf', 'sdag', 35.3), (5, 'esdf', 'sdf', 36.8);
insert into persons values(6, 'fsdf', 'sdf', 36.3), (7, 'gsdf', 'sdf', 36.5), (8, 'hsdf', 'sdf', 36.4), (9, 'isdf', 'sdf', 37.2), (10, 'jsdf', 'sdf', 37.3);
insert into persons values(11, 'ksdf', 'sdf', 36.8), (12, 'lsdf', 'sdf', 36.0), (13, 'msdf', 'sdf', 36.7), (14, 'nsdf', 'sdf', 36.9), (15, 'osdf', 'sdf', 37.2);

delete from persons where id = 8;
update persons set temperature = 36.8 where id = 2;
commit;

begin;
checkpoint;

insert into persons values(22, 'dsag', 'dsgat', 36.3);
delete from persons where id > 13;
update persons set temperature = 37.3 where id < 5;
insert into persons values(100, 'a', 'a', 38.0), (101, 'b', 'b', 38.1), (102, 'c', 'c', 38.2), (103, 'd', 'd', 38.3), (104, 'e', 'e', 38.4);

```

```
flush;
crash;

use dbtrain_test_lab2_advanced;
select * from persons;
```

- 实验结果如下：

  可以看见，在 `Checkpoint_Begin` 和 `Checkpoint_End` 中间出现了一条 `UPDATE` 语句，说明 Checkpoint 的记录是非阻塞的。

```
dbtrain>
Write Log [21] Type: CHECKPOINT_BEGIN
+---------+
| SUCCESS |
+---------+

dbtrain>
Write Log [22] Type: UPDATE
+---------+
| SUCCESS |
+---------+

dbtrain>
Write Log [23] Type: CHECKPOINT_END

Write Log [24] Type: UPDATE

Write Log [25] Type: UPDATE

Write Log [26] Type: UPDATE
+---------+
| SUCCESS |
+---------+
```

总结

- 高级功能 Commit ID 如下：
  - Undo 过程中系统出现异常的恢复（位于 `ch2a` 分支上）

    Commit ID：`bcb9166db90ec472cd95eff4e5ed90b6dec410c5`
  - 非阻塞的 Checkpoint 日志记录（位于 `ch2b` 分支上）

    Commit ID：`ea2d9f9be4acfb6595de68bb0a26628f9f2d2884`
- 用时：
  - 基础功能用时 8 小时；
  - Undo 异常恢复用时 10 小时；
  - Fuzzy Checkpoint 用时 6 小时；
  - 合计 24 小时。