Database Management
Homework 5

B11705044
Yen-Hung, Chiang

1

下面的 Python 程式碼會先連接資料庫，如果連接失敗會丟出 Failed to connect to the database 的訊息。接著進到第二個 try 區段，先去把 `account_id` = 1 的人扣掉 100 元，然後檢查他被扣款後的餘額是否大於 0，如果餘額小於 0 則 rollback，並且拋出 Insufficient funds 這段訊息；如果餘額足夠則把 `account_id` = 2 的人餘額加上 100，接著 commit 這段 transactions。

```python
import psycopg2

db_params = {
    "dbname": "******",
    "user": "******",
    "password": "******",
    "host": "******",
    "port": "****"
}

try:
    conn = psycopg2.connect(**db_params)
    conn.autocommit = False
    cur = conn.cursor()
except Exception as e:
    print(f"Failed to connect to the database: {e}")
    exit(1)

try:
    sender_account_id = 1
    receiver_account_id = 2
    transfer_amount = 100

    cur.execute("UPDATE accounts SET balance = balance - %s WHERE account_id = %s",
```

```
                (transfer_amount, sender_account_id))

    cur.execute("SELECT balance FROM accounts WHERE account_id = %s",
    ↪  (sender_account_id,))
    sender_balance = cur.fetchone()[0]

    if sender_balance < 0:
        conn.rollback()
        raise Exception("Insufficient funds")

    cur.execute("UPDATE accounts SET balance = balance + %s WHERE account_id = %s",
                (transfer_amount, receiver_account_id))

    conn.commit()

except Exception as e:
    conn.rollback()
    print(f"Transaction failed: {e}")

finally:
    cur.close()
    conn.close()
```

2

底下分別是兩個 trasactions，他們分別會對資料表的同一筆資料做交易（圖 1），Trasaction 1 會去把資料表 id = 1 的 value 更新為 200（圖 2），而 Trasaction 2 則是在 Trasaction 1 還沒 commit 之前去查詢同一筆資料圖（3），接著讓 Trasaction 1 commit 之後 Trasaction 2 再去查一次資料圖（4）。根據實際執行的結果我們可以看到 PostgreSQL 的確會防止 dirty read 的情況產生。

```
-- Transaction 1
BEGIN;
UPDATE test_table SET value = 200 WHERE id = 1;
-- 停留在這等待 Transaction 2 的操作
COMMIT;
```

```
-- Transaction 2
BEGIN;
SELECT * FROM test_table WHERE id = 1; -- 讀取 Transaction 1 正在修改的資料
```

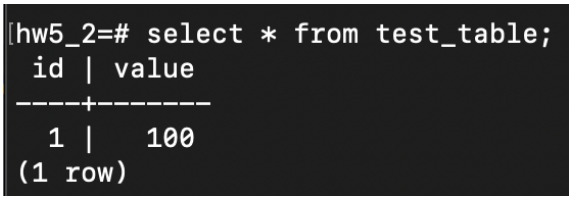

圖 1: 原始資料表

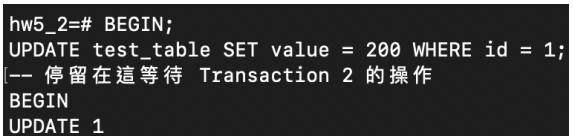

圖 2: Transaction 1 執行到一半

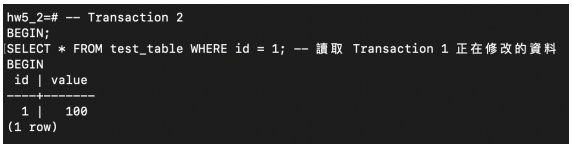

圖 3: Transaction 2 嘗試查詢

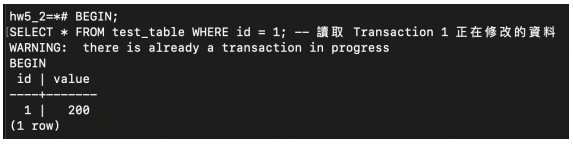

圖 4: Transaction 2 在 Transaction 1 commit 之後查詢

3

(a)

拿掉所有跟鎖有關的兩個原始交易如表 1 所示。

$\mathbf{T}_1$	$\mathbf{T}_2$
read_item(Y);	read_item(X);
read_item(X);	read_item(Y);
X:= X + Y;	Y:= X + Y;
write_item(X);	write_item(Y);

表 1: T_1 and T_2 transactions

(b)

當有兩個交易在同一個資料上 read 和 write 時，如果至少有一個交易在 write 則會衝突，衝突配對如表 2 所示。

$\mathbf{T}_1$	$\mathbf{T}_2$
read_item(Y);	write_item(Y);
write_item(X);	read_item(X);

表 2: T_1 and T_2 的衝突作業配對

(c)

如果是 T_1 先發生然後 T_2 才發生的話可以排出下面的 Serial schedule，如表 3 所示。

$\mathbf{T}_1$	$\mathbf{T}_2$
read_item(Y);	
read_item(X);	
X:= X + Y;	
write_item(X);	
	read_item(X);
	read_item(Y);
	Y:= X + Y;
	write_item(Y);

表 3: Serial schedule of T_1 and T_2 transactions

(d)

等價的 non-serial schedule 的結果如表 4 所示，由於原來是 T_1 先做，所以 T_2 在讀 X 的時候會讀到已經被 T_1 修改過的資料，當把它改成 non-serial schedule 時要讓 T_2 在 T_1 修改過並寫入新的 X 後再去讀 X 的資料，這樣就可以保持和 Serial schedule 一樣的效果。

$\mathbf{T}_1$	$\mathbf{T}_2$
read_item(Y);	
read_item(X);	
	read_item(Y);
X:= X + Y;	
write_item(X);	
	read_item(X);
	Y:= X + Y;
	write_item(Y);

表 4: Non-serial schedule of T_1 and T_2 transactions

(e)

不等價的 non-serial schedule 如表 5 所示，由於 T_2 沒有在 T_1 寫入新的 X 之後才讀 X，所以執行的結果就會不一樣。

$\mathbf{T}_1$	$\mathbf{T}_2$
read_item(Y);	
	read_item(X);
read_item(X);	
	read_item(Y);
X:= X + Y;	
	Y:= X + Y;
write_item(X);	
	write_item(Y);

表 5: Non-serial schedule of T_1 and T_2 transactions

(f)

幫兩段交易都加上兩階段鎖定的機制後如表 6 所示。

$\mathbf{T}_1$	$\mathbf{T}_2$
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
read_lock(X);	read_lock(Y);
read_item(X);	read_item(Y);
upgrade_lock(X);	upgrade_lock(Y);
X:= X + Y;	Y:= X + Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);
unlock(Y);	unlock(X);

表 6: T_1 and T_2 transactions after adding lock

(g)

下面的加上鎖的 non-serial schedule 如表 7 所示，因為都按照一樣的順序加上鎖，並在操作結束後立即釋放鎖，所以不會造成死鎖。

$\mathbf{T}_1$	$\mathbf{T}_2$
read_lock(Y);	
read_item(Y);	
read_lock(X);	
read_item(X);	
	read_lock(Y);
	read_item(Y);
upgrade_lock(X);	
X:= X + Y;	
write_item(X);	
unlock(Y);	
unlock(X);	
	read_lock(X);
	read_item(X);
	upgrade_lock(Y);
	Y:= X + Y;
	write_item(Y);
	unlock(Y);
	unlock(X);

表 7: Non-serial schedule of T_1 and T_2 transactions with no deadlock

(h)

會造成死鎖的 non-serial schedule 如表 8 所示，由於兩個交易申請鎖的順序不一樣，會造成 T_1 在等 T_2 釋放鎖，而 T_2 也在等 T_1 釋放鎖的情形，進而造成死鎖。

T_1	T_2
read_lock(X);	
read_item(X);	
	read_lock(Y);
	read_item(Y);
read_lock(Y);	
read_item(Y);	
upgrade_lock(X);	
X:= X + Y;	
write_item(X);	
unlock(Y);	
unlock(X);	
	read_lock(X);
	read_item(X);
	upgrade_lock(Y);
	Y:= X + Y;
	write_item(Y);
	unlock(Y);
	unlock(X);

表 8: Non-serial schedule of T_1 and T_2 transactions with deadlock

4

(a)

步驟一的查詢：

```sql
SELECT remaining_ticket_qty
FROM REMAINING_TICKET
WHERE travel_date = '2024-12-04' AND train_id = 123;
```

步驟三的再次查詢：

```sql
SELECT remaining_ticket_qty
FROM REMAINING_TICKET
WHERE travel_date = '2024-12-04' AND train_id = 123 FOR UPDATE;
```

步驟三的更新：

```sql
UPDATE REMAINING_TICKET
SET remaining_ticket_qty = remaining_ticket_qty - 1
WHERE travel_date = '2024-12-04' AND train_id = 123;
```

步驟三的新增：

```sql
INSERT INTO PURCHASE (customer_id, travel_date, train_id, purchase_datetime)
VALUES ('customer_001', '2024-12-04', 123, CURRENT_TIMESTAMP);
```

(b)

不需要防止 unrepeatable read，因為第一次查詢時有票但第二次查詢時發現沒有票的狀況是業務邏輯所允許的情況，所以不必防止。

(c)

應該要在步驟三的再次查詢加上 FOR UPDATE，避免在後續要更新時別的交易把票搶走。

```sql
SELECT remaining_ticket_qty
FROM REMAINING_TICKET
WHERE travel_date = '2024-12-04' AND train_id = 123 FOR UPDATE;
```

(d)

步驟一的查詢：

```sql
SELECT remaining_ticket_qty
FROM REMAINING_TICKET
WHERE travel_date = '2024-12-04' AND train_id = 123 FOR UPDATE;
```

步驟三的更新：

```sql
UPDATE REMAINING_TICKET
SET remaining_ticket_qty = remaining_ticket_qty - 1
WHERE travel_date = '2024-12-04' AND train_id = 123;
```

步驟三的新增：

```sql
INSERT INTO PURCHASE (customer_id, travel_date, train_id, purchase_datetime)
VALUES ('customer_001', '2024-12-04', 123, CURRENT_TIMESTAMP);
```

這次應該要改成在步驟一的查詢就加上 FOR UPDATE 防止在臨櫃消費者考慮要不要購票時票就被其他的交易搶走。

5

關聯式資料庫在分散化上效益不彰，主要是因為他對強一致性和複雜查詢的支持需要大量跨 node 的查詢，而這與分散式系統的低延遲和高容錯性目標相衝突。相較之下，NoSQL 資料庫對一致性的要求較低，更適合分散式環境，因此能更好地應對分散式應用的需求。

6

(a)

合理，因為資料的結構不固定，關聯式資料庫需要有固定格式的資料，而 NoSQL 則不用，另外資料量很大，NoSQL 支援分散式系統，在處理大量的資料時可以更有效率，相較起來關聯式資料庫就不適合分散式系統。因此選用 NoSQL 紀錄是合理的。

(b)

建議按照時間分片，因為他要進行趨勢分析，照理來說會需要特定時間的資料，所以按照時間分片可以針對某段時間的數據查詢，還可將相關數據集中在同一分片，降低查詢範圍。

(c)

對 TICKET 表做分片，按 `EVENT_ID` 分片。因為搶票場景中，票券資料的查詢集中在特定的活動上。按活動分片可以讓特定活動的請求集中在少數節點上，減少跨節點操作的延遲。

7

(a)

好處：

- 可以減少對營業用資料庫的干擾，在分析資料時不會操作營業用資料庫，營業用資料庫可以專注於高效處理日常交易操作。
- 分析師只能訪問資料倉儲，避免誤更改到原始資料庫的內容。
- 資料倉儲的數據會經過清洗與轉換，所以資料會有更高的一致性和可用性。

壞處：

- 與營業用資料庫的資料之間會有延遲，因為存資料需要時間。
- 使用資料倉儲會需要額外的成本。
- 營業用資料庫與資料倉儲之間需要定期同步，較麻煩。

(b)

使用關聯式資料庫，因為資料倉儲的數據大多是結構化的資料，並且需要大量的 join 與其他分析，資料倉儲的數據需保證一致性，例如 ACID，所以關聯式資料庫較為適合。

(c)

好處：

- 在進行 SUM、AVG 的查詢時，只需要讀取相關的列，避免了讀取不必要的數據，提高查詢效率。

- 同一列的數據類型通常會一致，壓縮效率比列式儲存更高。
- 行式儲存會更適合 OLAP 查詢，因為這些查詢通常只涉及部分列。

壞處：

- 行式儲存需要把資料拆分並寫入多個列中，會導致寫入速度較慢。
- 不適合 OLTP 工作，因為資料寫入速度較慢。

若想計算某年度每月銷售額的總和，只需要讀取銷售額和日期列，如果是列式儲存則需要讀取整行資料，增加不必要的 I/O。