RDBMS 課程 v2 （第一課）

by

Triton Ho 

今天大綱

- Race condition 是什麼
- 溫習知識： RDBMS 的 3 種 read phenomena
- 溫習知識： Shared-exclusive Lock 還有 MVCC
- 溫習知識： RDBMS 的 4 個 Isolation level
- 面對 Race condition 的實務建議

前言： Isolation 很麻煩！

- 因為現實世界，很多時候你的 atomicity 必須涉及多個 record
- 因為現實世界，很多時候 verification 和 update 不能在同一句 statement 內完成

  例子：
  - 客戶想要購買 HKG 去 HKT 的最短路徑的機票
    - Step 1: 紀錄航斑斑次表（只包括還有座位的）的最後改動時間
    - Step 2: 使用 Dijkstra's 演算法，先找出 shortest path 吧（先假設結果是 HKG -->BKK ， BKK-->HKT 吧）
    - Step 3: begin TX
    - Step 4: 再次檢查在航斑斑次表在 step 0 後有否被改動
    - Step 5: 檢查所以客戶是否有足夠金額去支持整個旅程費用
    - Step 6: 購買 HKG-->BKK 機票
    - Step 7: 購買 BKK-->HKT 機票
    - Step 8: Commit TX

為什麼要學習 Isolation

正規化的基礎訓練，能讓你設計系統時能跟蹤前人的智慧，而不是自行發明看起來沒問題的演算法。

- 如果你只用 noSQL ，雖然接下來內容看起來好沒用，但是：
  - noSQL 不是不需要 locking ，而是需要自己手動管理 locking 。你在 RDBMS 面對的困難，在 noSQL 只會加倍的困難。
  - 總結：請不要穿新手裝去越級打怪。

- 抱歉，這部份有大量基礎知識。無論怎麼沉悶都請要堅持下去。
  - ~~反正你放棄了也拿不回學費~~

RDBMS 的 Isolation

- 簡單來說，就是幫助 developer 避免 race condition 的機制
- 為了最大化效能， RDBMS 會同時處理盡量多的 TX 。
  - 今天 CPU 的單一線程效能不再跟蹤 Moore's law
  - Backend 的效能是以吞吐量 (throughput) 來計算的

    Throughput = Concurrency level * (1 / TX process time)
- 但是，如果兩個 TX 正在改動相同的資料， RDBMS 會讓後來的 TX 被 blocking （阻擋），這樣便不會發生 race condition

Race condition 是什麼

Race condition

- 最簡單來說：兩個 process 同時運行 (Concurrent execution) 時，同時進入 Critical section （關鍵部份），引發錯誤。
- 在佔香港 25 %空運流量的貨運站，大約每一到兩個月發生一次。
  - 這只計算被發現到的
  - 每次都引發大量數據錯誤，然後大量的爆肝

Race condition 例子

- 在航班的售票系統中：
  - Step 1A:　客戶 A 查看 BR855@16SEP 的坐位，這時系統顯示 81A 座位是空的
  - Step 1B:　同一時間，客戶 B 查看 BR855@16SEP 的坐位，這時系統顯示 81A 座位是空的
  - Step 2:　客戶 A 和 B 同一時間都決定坐上 81A 座位，同時按下確定
  - Step 3:　客戶 A 比客戶 B 慢了數微秒按下確定，所以他蓋過了客戶 B 的紀錄，然後得到座位，然後客戶 B 到了機場時才發現沒了座位了

Race Condition 的必需條件

- 同一資源會被同時改動
- 「檢查」和「資料改動」不是在同一 Atomic operation 中完成
- 註：這是假設有 atomic WRITE 的情況下

Race condition －安全例子 1

- 讓同一航班所有資料改動都要先拿到該航班 WRITE LOCK ，改動完成後便解除 WRITE LOCK
  - Zookeeper
  - Redis
- 缺點：萬一得到那個 WRITE LOCK 的伺服器當掉，那麼便要等待 WRITE LOCK 自然到期才能更新資料

Race condition －安全例子 2

- 為每一航班建立對應的 Queue
- 對同一航班的改動，必須先把工作放到對應的 Queue ，然後單線程順序執行
- 因為同一時間只有一個線程正在改動同一份資料，所以不會發生 Race condition

Race condition 一安全例子 2（續）

- 缺點：非常麻煩，超級麻煩！
- 這方法需要使用 MQ Server ， worker ， server push ，這些額外的組件會增加系統複雜度
- 不是所有 race condition 都能用這方法：
  - 像轉帳 A→B 的工作，應該放到 A 的 Queue 還是 B 的 Queue ？
    - 別告訴我把工作拆成 2 部份分別放到 Queue A 和 Queue B

Race condition －安全例子 3

- 善用 RDBMS 的 Atomicity 還有 Isolation
  - Update seats set user = $user
    where flight_no='BR855' and flight_date = '16SEP'
    and user = null
  - 使用 AffectedRowCount 便知道是否更新成功
- 關鍵字： atomic check-and-set
- 缺點：呃，中小型系統沒什麼缺點的。

RDBMS 的 3 種 read phenomena

RDBMS 的 read phenomena

- Dirty Read
  - 能讀取其他還未 committed 的 TX 的資料改動
- Non-repeatable read
  - 在同一個 TX，同一筆資料在第一次讀取和第二次讀取出現不同結果
  - 另一個說法：讀取的資料包含其他已經 committed TX 的 update，而這些 TX 的 commit 時間發生在本 TX 的開始之後
- Phantom read
  - 在同一 TX，同一 Query 在第一次和第二次執行時，出現不同結果
  - 另一個說法：讀取的資料包含其他已經 committed TX 的 insert/update，而這些 TX 的 commit 時間發生在本 TX 的開始之後
- http://en.wikipedia.org/wiki/Isolation_(database_systems)

RDBMS 的 read phenomena （續）

- 別分不清 Non-repeatable read 和 Phantom Read 耶～
- Non-repeatable read(NRR)
  - 第一次 Query 和第二次返回是同一批的 Record ，但是 Record 的內容不同了
- Phantom read
  - 第二次返回的結果比第一次多了 Record

Dirty Read 例子

- 如果股票交易系統容許 Dirty Read ，你不用花 1NT 便能操縱股價了

奸商：

```
Start transaction read uncommitted;

Update stock set last_price = '0USD'
where stock_name = 'MSFT';

Rollback;
```

不知情的外人：

```
Start transaction read uncommitted;

Select last_price from stock
where stock_name = 'MSFT';
/* 顯示 0USD ，因為看到了奸商還未交易完成的數據 */

Commit;
```

只要奸商不停地發起 0USD 的虛假交易然後 rollback ，沒有防範 Dirty Read 的用戶便會看到其偽造的 last_price

Dirty Read 小結

- Dirty Read 的數據都是還未 Committed 的數據
  即是說：有機會因為 Rollback 而放棄掉的數據
- 在還未 Commit 之前， database 是處於 inconsistent state （不正確是指商業邏輯上），這些「不正確的數據」是不應該被其他用戶看到的
- 任何情況下， Dirty Read 在 RDBMS 都應該避免的
  - 所以 MySQL, Oracle, PostgreSQL 標準設定都不會發生 Dirty Read

Non-repeatable read 例子

普通人：
Start transaction READ COMMITTED;

Select price from flight
where flight_name = 'HKG-->BKK';
/* 顯示 200USD ，用戶覺得價格合理，並且
按下「購買」按鍵 */

Update user set
balance = balance - (Select price from flight
where flight_name = 'HKG-->BKK')
Where user.id = $userId;
/* 這時用戶會被扣除 300USD ，之後請向消保
官解釋吧 */

Commit;

航空公司同一時間卻決定加價：
Start transaction read committed;

Update flight set price = '300USD'
where flight_name = 'HKG-->BKK';

Commit;

Non-repeatable read 小結

- NRR 會讓同一 Record 在不同的 select 中顯示不同的數值
- 一般來說，同一數據的第一次 select 是用作 data verification ，第二次是用作 data processing
- 如果 verification 和 processing 是使用不同的數值，那麼 verification 根本沒意義

Phantom read 例子

普通人：

Start transaction REPEATABLE READ ；

Select sum(cost) from flight_misc_cost
where flight_name = 'HKG-->BKK';
/* 機場費，稅金加起來顯示 50USD ，用戶覺得價格合理，並且按下「購買」按鍵 */

Update user set
balance = balance - (Select sum(cost) from flight_misc_cost
where flight_name = 'HKG-->BKK')
/* 這時用戶會被扣除 60USD ，還是向消保官解釋好了 */

Commit;

政府決定讓航空公司徵收燃油附加費，航空公司當然立即跟隨：

Start transaction REPEATABLE READ;

Insert into flight_misc_cost(flight_name, item_name, cost) values
('HKG-->BKK', ' 燃油附加費 ', '10USD')

Commit;

Phantom read 小結

- Phantom read 會讓第二次的 query 出現了新的 Record
  - 正如其名，這些 Records 就像阿飄一樣浮出來
- Phantom read 引起的問題跟 Non-repeatable read 相近
- 但是 PR 比 NRR 更難防範

溫習知識： Shared-exclusive Lock 還有 MVCC

兩種 isolation 流派

- 在 RDBMS 中， SX lock 和 MVCC 都能達到 Isolation 目的，避免 Race Condition
- Oracle 和 PostgreSQL 使用 MVCC ，而 MySQL 和 MSSQL 使用 SX lock
- 因為其底層機制不同，所以雖然大家都說自己支持 ACID ，支持４個 isolation level ，其 isolation 的行為卻有所不同
- MySQL 和 MSSQL 近年試圖在其 SX lock 架構上再加上 MVCC ，所以引入了大量「 feature 」
  - ~~用戶能接受的便不是 bug ，是 feature~~

Shared-exclusive lock

- 在資料庫中，每一個 Record 都有其 SX lock
  - 別名 Readers–writer lock(RW lock)
- 所有由 TX 擁有的 lock ，會在 TX 結束時自動歸還
  - （S Lock 視 Isolation level 決定歸還時間）
- S lock 是對應資料讀取的， X lock 是對應資料改動
- 因為 S lock 能發給多個 TX ，所以同一時間能有多個 TX 讀取同一塊資料
- 發行 X lock 時， Record 必須沒有其他的 lock （不管是 S 還是 X ）。直到該 X lock 結束之前，該 Record 都不能發行其他的 lock
  - 所以直到該 TX 結束，該 Record 都不能被其他 TX 讀取／改動
- http://en.wikipedia.org/wiki/Readers%E2%80%93writer_lock

MVCC

- Multiversion concurrency control
  - 最簡單來說，每次的 insert / update 會為該 Record 增加額外的版本
  - 所以， Record 的每個版本都儲有時間（或相似東西），讓 RDBMS 知道那份版本才是最新的
  - Record 太舊的版本會被 RDBMS 自動刪除
- MVCC 的資料庫的 Record 只有 X lock ，而沒有 S lock 的
- MVCC 在讀取時，會讀最該 Record 「最新」的 Committed 版本，所以自然地沒有了 Dirty Read
- 因為只有 WRITE-WRITE conflict ，所以 MVCC 的 Read 永遠不會被阻擋

SX lock 還是 MVCC?

- 只考慮一個 TX ， MVCC 使用比較多的 IO 還有 CPU ，所以比較花時間
  - 因為要決定那個版本才是「最新」
  - 因為要管理舊的版本的刪除
- 在高流量環境， MVCC 比較高效能。
  - 因為 MVCC 的 READ 永遠不被阻擋，同一時間資料庫能處理更多的 TX
  - MVCC 只有 X lock 而沒有 S lock ，其 Deadlock detector 要管理的 lock 的數目一定更少，所以一定比較快

溫習知識： RDBMS 的 4 個 Isolation level

RDBMS 的 Isolation level

- Read Uncommitted
- Read Committed
  - 能防止 Dirty Read
- Repeatable Reads
  - 能防止 Dirty Read, Non-repeatable read
- Serializable
  - 能防止 Dirty Read, Non-repeatable read, Phantom read

基於 SX lock 的 Read Committed

- 改動前，為 Record 加上 X LOCK ，直到 TX 完結
- 讀取前，為 Record 加上 S LOCK ，在 statement 結束後立即歸還
- 其他試圖讀取／改動這些 rows 的 TX 將會等待（blocked），直到本 TX 完成
- 改動會阻擋讀寫

Dirty Read （SX lock 回顧）

奸商：
Start transaction read committed;

Update stock set last_price = '0USD'
where stock_name = 'MSFT';
/* 資料被加上 X lock*/

Rollback;
/* 歸還 X lock*/

不知情的外人：
Start transaction read committed;

Select last_price from stock
where stock_name = 'MSFT';
/* 拿取 S lock 失敗，所以本 TX 被 blocking*/

/* 成功拿取 S lock ，顯示正確股價 */
Commit;

- 即使奸商不停地發起 0USD 的虛假交易然後 rollback ，在 Read Committed Isolation 下用戶只是需要額外的等待時間，不會收到奸商沒有 commit 的 dirty data

基於 MVCC 的 Read Committed

- 進行 insert/update/delete 時，會先為 Record 其加上 X LOCK ，直到 TX 完成
- 讀取 Record 時，只會考慮已經 committed 的最新版本
  - 即使 Record 已被加上 X lock ， Read 也不會被阻擋
- 當兩個 TX 想改動同一 Record 時，其中一個才被阻擋

Dirty Read （MVCC 回顧）

奸商：
Start transaction read committed;

Update stock set last_price = '0USD'
where stock_name = 'MSFT';
/* 資料被加上 X lock ，並且建立還未 Commit 的 ver1 副本 */

Rollback;
/* 歸還 X lock*/

不知情的外人：
Start transaction read committed;

Select last_price from stock
where stock_name = 'MSFT';
/* 資料庫無視還未 Committed 的 ver1 副本，並且返回最新的 ver0 副本 */

Commit;

即使奸商不停地發起 0USD 的虛假交易然後 rollback ，在 Read Committed Isolation 下未 Committed 的資料會被自動無視，用戶不會收到奸商沒有 commit 的 dirty data

基於 SX lock 的 Repeatable Reads

- 跟 Read Committed 相似，只是讀取時的 S lock 要在 TX 結束時才歸還
  - 所以讀過了的 Record 便不能被改動，哇哈哈哈
  - 所以很容易引起 deadlock，哇哈哈哈（這沒有刪除線的）
- 改動會阻擋讀寫　＋　讀寫會阻擋改動
- 任何本質上是 SX lock，但又自稱自己支持 MVCC 的 RDBMS，請自求多福
- 這世界有「基於 SX lock 也合乎 ISO 標準的 Repeatable Read」，然後還有……
  - 「MySQL 5.6 的 Repeatable Read」
  - 「MySQL 5.7 的 Repeatable Read」

NRR （SX lock 回顧）

普通人：
Start transaction REPEATABLE READ;

Select price from flight
where flight_name = 'HKG-->BKK';
/* 資料被加上 S lock */

Update user set
balance = balance - (Select price from flight
where flight_name = 'HKG-->BKK';)
/* 資料的 S lock 提升為 X lock ，這時用戶會被扣除 200USD */

Commit;
/* 返回所有鎖 */

航空公司同一時間卻決定加價：
Start transaction REPEATABLE READ;

Update flight set price = '300USD'
where flight_name = 'HKG-->BKK';
/* 因為無法拿到 X lock ，所以被 blocking */

/* 終於拿到了 X lock ，執行資料改動 */
Commit;
/* 返回所有鎖 */

基於 MVCC 的 Repeatable Read

- 讀取資料時，只考慮在 TX 開始前已經 committed 的版本
  - 別名 Snapshot （快照） Isolation
- 改動資料時，除了拿取 X lock ，還檢查 Record 是否存在 TX 開始後的 Committed 版本。如果存在，便 raise exception 並強制 Rollback 目前 TX
  - PostgreSQL: could not serialize access due to concurrent update
  - 註： Oracle 把檢查拖延到 TX commit 時才進行，但影響不大
- 雖然不像 SX lock 一般容易產生 deadlock ，但是 Repeatable Read TX 還是很容易需要 Rollback 重來
  - 呃，請做好心理準備
- Oracle 聲稱的 Serializable Isolation ，真面目只是 Repeatable Read

NRR （MVCC 回顧１）

普通人：
Start transaction REPEATABLE READ;

Select price from flight
where flight_name = 'HKG-->BKK';
/* ver0 的 price 是 200USD */

航空公司同一時間卻決定加價：
Start transaction REPEATABLE READ;

Update flight set price = '300USD'
where flight_name = 'HKG-->BKK';

Commit;
/* 返回所有鎖，並且把 ver1 副本狀態改成 Committed*/

Update user set
balance = balance - (Select price from flight
where flight_name = 'HKG-->BKK';)
/* 忽略 TX 開始後才建立的 ver1 ，用戶被扣掉 200USD */

Commit;

即使航空公司的 TX 從中間開始並且先行完成，普通人的 TX 還是只使用 TX 開始時的數據。
現實效果的排序便是：（普通人的 TX），（航空公司的 TX）

NRR （MVCC 回顧 2）

普通人：

Start transaction REPEATABLE READ;

Select price from flight
where flight_name = 'HKG-->BKK';
/* ver0 的 price 是 200USD */

Update user set
balance = balance - (Select price from flight
where flight_name = 'HKG-->BKK';)
/* 忽略 TX 開始後才建立的 ver1 ，用戶被扣掉 200USD */

Commit;

使用 Intel 8086 CPU 的航空公司決定加價：
Start transaction REPEATABLE READ;

Update flight set price = '300USD'
where flight_name = 'HKG-->BKK';

/* 因為 CPU 太慢， 10 分鐘後才發出 Commit*/
Commit;
/* 返回所有鎖，並且把 ver1 副本狀態改成 Committed*/

必須注意的：即使航空公司 TX 開始時間在普通人 TX 之前，邏輯排序卻是：（普通人的 TX），（航空公司的 TX）。

基於 SX lock 的 Serializable

- 先說一下：RDBMS 的 Serializable 不等於數學上的 Serializable
  - Auto increment, sequence 不被包括到 TX
- predicate lock：像是 where age between 20 and 35
- 在 Repeatable Read 的基礎上，在執行 query 時，除了為會被讀取的 rows 加上 READ_LOCK 之外，還加上 predicate lock
- 其他 TX 的 insert/update，只要影響到的 rows 滿足 predicate lock 的範圍，那個 TX 也會上鎖
- MySQL 的 Serializable 不遵守 ANSI SQL 定義
  - ~~呃，MySQL 5.7 的 Repeatable Read，用戶請自求多福~~
  - MySQL 用了 gap lock 去防止 Phantom Read，用戶請自求多福
- 極度吃 CPU 也極容易引起 deadlock，沒特別原因別使用這模式

Phantom read （SX lock 回顧）

普通人：
Start transaction SERIALIZABLE;

Select sum(cost) from flight_misc_cost
where flight_name = 'HKG-->BKK';
/* 建立 where flight_name = 'HKG-->BKK' 的 predicate lock */

Update user set
balance = balance - (Select sum(cost) from flight_misc_cost
where flight_name = 'HKG-->BKK')
/* 這時用戶會被扣除 50USD */

Commit;
/* 返回所有鎖 */

航空公司新增燃油附加費：
Start transaction SERIALIZABLE;

Insert into flight_misc_cost(flight_name, item_name, cost) values
('HKG-->BKK', ' 燃油附加費 ', '10USD')
/* 因為資料符合 predicate lock ，所以 insert 被阻擋 */

/* 沒有了 predicate lock ， TX 可以繼續執行 */
Commit;

基於 MVCC 的 Serializable

- 如果沒有使用 sequence/ auto increment ， RDBMS 的 Serializable(MVCC) 等於數學上的 Serializable
- 在 Repeatable Read 的基礎上，為每個 Query 的 Predicate 加上 Predicate monitoring
- 當有新版本的 committed rows 滿足 predicate ，而其版本是在本 TX 開始的時間點後，則本 TX raise exception 並且 rollback
- 高 CPU 要求，沒特別原因不建議使用
- Oracle 沒有這個級別

Phantom read （MVCC 回顧）

普通人：
Start transaction SERIALIZABLE;

Select sum(cost) from flight_misc_cost
where flight_name = 'HKG-->BKK';
/* 建立 where flight_name = 'HKG-->BKK' 的 predicate monitoring */

/* 接收到 predicate monitoring 的 exception ，引起強制性 Rollback*/

rollback;
/* 返回所有鎖和 monitoring*/

航空公司新增燃油附加費：
Start transaction SERIALIZABLE;

Insert into flight_misc_cost(flight_name, item_name, cost) values
('HKG-->BKK', ' 燃油附加費 ', '10USD')

Commit;
/* 因為資料符合 predicate ，提醒 predicate 相關的 TX*/

write skew

- Phantom Read 的一種
- 二個同時執行的 TX ，都在 insert record ，而這些 Record 是合乎對方的 Select 條件
- 在 Reportable-Read Isolation 下， T1 和 T2 都看不見對方的 insert
- Consistency 是指能為 T1 和 T2 分出先後次序，而在 write skew 時不能
- 在 MVCC 世界中的 Serializable Isolation ，便是專門用來防範 write skew 的

write skew 例子

- 某銀行，他只要求你的總負債不超過 100NT
  - Select sum(amount) from debts
    where user_id = ? and status = 'unpaid'
- 在 Repeatable Read 時， TX 看不到開始時後的 Insert
- 所以只要同一秒在網上理財按下借錢，便能突破 100NT 限制

write skew 例子（續）

Start transaction REPEATABLE READ;

Select sum(amount) from debts
where user_id = X and status = 'unpaid';
/* 返回 70NT ，通過檢查 */

/* 執行借錢 20NT*/
Insert into debts(user_id, amount, status)
values (X, 20NT, 'unpaid');

commit;

Start transaction REPEATABLE READ;

Select sum(amount) from debts
where user_id = X and status = 'unpaid';
/* 返回 70NT ，通過檢查 */

/* 執行借錢 20NT*/
Insert into debts(user_id, amount, status)
values (X, 20NT, 'unpaid');

commit;

結果是：借款人在二個 TX 都同時 commit 後，欠債到達 110NT ，是 race condition

面對 Race condition 的實務建議

實務建議 1

- 請不要輕視 Race condition ，任何可以發生 Race condition 的地方都會發生 Race condition （ Murphy's law ）
  - 還沒有發生 Race condition ，只代表這個有問題的系統沒有用戶。
- 現實數據改動不是隨機的，而是集中的 (Clustered)
  - 如果只剩下最後一個位子，大家都會搶那個的
  - LOL 中，大家都會搶尾刀的~~（我不是玩奈德麗的）~~

實務建議 2

- 在 Read Committed 中使用 Conflict promotion 來預防 Non Repeatable Read ，而不使用 RDBMS 內建的 Repeatable Read Isolation level
- Conflict promotion 意思，就是在 checking 時，在 select 的指令尾部加入 for share ，讓所有讀取過的 Record 被加上 S lock ，直到 TX 結束
  - 這跟 Repeatable Read 效果相同，只是我們人手操作，讓 RDBMS 不會把非關鍵的資料也加上 S lock
  - MVCC 的資料庫不一定有 for share ，改用 for update 去拿 X lock 也有相同效果

Conflict promotion 例子

普通人：
Start transaction Read committed;

Select price from flight
where flight_name = 'HKG-->BKK' for share;
/* record 被加上 S lock */

Update user set
balance = balance - (Select price from flight
where flight_name = 'HKG-->BKK';)

Commit;
/* 返回所有鎖 */

航空公司同一時間卻決定加價：
Start transaction Read committed;

Update flight set price = '300USD'
where flight_name = 'HKG-->BKK';
/* 因為沒法拿到 X lock ，所以被 blocking*/

/* 拿到了 X lock 繼續執行 */

Commit;

實務建議 3

- 在 Read Committed 中使用 Conflict promotion + Conflict materialization 來預防 Phantom Read，而不使用 RDBMS 內建的 Serializable Isolation level
- Conflict materialization 意思：
  - 如果兩個 table 存在 parent-child 關係（例子：flight 和 flight_misc_cost）
  - 所有對 child record 的 Read 都要為其 parent record 加上 S lock
  - 所有對 child record 的 Write 都要為其 parent record 加上 X lock
  - 所以，對 child table 加入新 record 時，便會跟其他正在讀取該範圍（指同一 parent）的 TX 的 S lock 發生碰撞，讓其 insert 被阻擋
- 註：極少數的 Phantom Read 發生於沒有 parent table，或是 select predicate 不包含 parent id，這時便需要故意創造一個人工的 parent table 去防範 Phantom Read
- 其實，大部份系統，只用上 Read Committed 中使用 Conflict promotion + Conflict materialization 便足夠了，不建議用更高階的 isolation
  - ~~學費不能退回啦，不要丟雞蛋和蕃茄啦~~

Conflict materialization 例子

普通人：

Start transaction Read committed;

Select 1 from flight where flight_name = 'HKG-->BKK' for share;

/* 為 flight 加上 S lock */

Select sum(cost) from flight_misc_cost where flight_name = 'HKG-->BKK';

Commit;

/* 返回所有鎖 */

航空公司新增燃油附加費：

Start transaction read committed;

Select 1 from flight where flight_name = 'HKG-->BKK' for update;

/* 試圖拿取 X lock 失敗，所以被 blocking*/

/* 拿到了 flight record 的 X lock ，繼續執行 */

Insert into flight_misc_cost(flight_name, item_name, cost) values

('HKG-->BKK', ' 燃油附加費 ', '10USD')

Commit;

註：部份 MVCC database 沒有 S lock ，改用 X lock 也行

實務建議 4

- 不是每種 Race condition 都是需要迴避的
  - 如果你在賣遊戲中的虛寶，賣 1 萬件和賣 1 萬＋ 1 件沒分別~~（賣模型的奸商們：我們賣完即止，絕不再販）~~
  - 如果你的系統在賣火車票，當然絕對不能超賣~~（呃，印度除外，超賣了便讓人們坐火車頂吧）~~
- 最重要一點：讓 Race condition 發生後不出現「致命錯誤」，讓系統向不致命的方向發生錯誤
- 在一個使用了 noSQL ，卻沒有使用 2pc 的售票系統：
  - 先 (檢查＋改動)user_balance ，然後改動 flight ，這樣便會有可能賣多了
  - 先 (檢查＋改動)flight ，然後改動 user_balance ，這樣便會有可能讓 user_balance 變成負值
- 容錯度越高的商業邏輯，其效能一定越快

實務建議 5

- 盡可能把 checking 和 data processing 放在同一句的 statement 內

  例子：
  update flight set vacancy = vacancy – 1
  where vacancy > 0 and flight_no = @flight_no

- 但是別走火入魔，試圖一句 statement 內實作 Dijkstra's algorithm

實務建議 6

- 善用 Optimistic concurrency control （又名： optimistic lock）
  - 雖然大家都習慣叫 optimistic lock ，但是這沒有真正的 locking 的
- 讀取資料時，也一併讀取這份資料的 timestamp
- update 時檢查這份資料的 timestamp ，便知道這份資料有否在期間被其他人改動
  - 這需要 atomic 的 check-and-set 指令， Cassandra 1.X 無法支援
- 如果讀取資料和改動資料的時間相距很長（大於 5 秒），一般來說 optimistic lock 更好

Optimistic concurrency control

例子：簡易的網上文件系統

- Step1 ：用戶想要改動 docX ，伺服器把文件內容還有現在文件的 timestamp 傳到用戶
  - Select content, last_update_time from documents where id = 'docX'
- Step2 ： 5 分鐘後，用戶完成修改，按下「儲存」
  - 用戶會把 @new_content ， @original_timestamp 回傳伺服器
  - Update documents set content = @new_content
    where id = 'docX' and last_update_time = @original_timestamp
- Step3 ：如果 @AffectedRecordCount = 0 ，告知用戶文件已經被其他人改動過了

實務建議 7

- 如果單一 record 有數千人嘗試同時改動，小心系統當掉
  - Coscup, ODSC 之前不是搶票搶到當掉的嗎
  - 系統繁忙頁面沒出來－＞ F5 －＞系統更繁忙－＞ F5
- Sharding / noSQL 在這種情況下沒有幫忙
  - 單一的 record 只能由單一主機負責
  - 每台主機的 CPU / IO 有其物理極限的

實務建議 8

- 在做超高流量的搶票系統，別用 insert 做 conflict collision
- Bad idea:
  - Insert into ticket…… 然後依靠 ticket 的 PK / UK 來預防相同座位被 2 人買到
- Good idea:
  - Update seat set user_id = ?, sold = true
    where position = ? and user_id = ? and sold = false
- 在搶票時，跑 insert 時的 page split 會拖低系統 throughout ， update 不會
- 沒賣出的票（sold = false）在售票期結束後刪掉便好

完