

RDBMS 課程（先修課）

今天大綱

- 為何選擇 RDBMS
- RDBMS 中的 ACID 分別是什麼？如何幫助我開發程式
- 對 RDBMS 的迷思

為何選擇 RDBMS

為何選擇 RDBMS (誤)

- ~~因為前輩叫我用 (誤)~~
- ~~因為某笨喵整天都在推廣 PostgreSQL (更大誤)~~
- ~~因為可以向外面廠商購買 RDBMS 的專業技術支援，之後自己犯的錯也能推給他們~~

為何選擇 RDBMS

- 歷史悠久，信心之選（無誤）
- 使用 RDBMS 比 noSQL 有更快的開發速度
- 對一般中小型系統，RDBMS 效能足夠了
- 使用 RDBMS 更加安全

RDBMS 歷史悠久

- PostgreSQL 在 1989 發行第一版
- MariaDB 的前身 MySQL ， 1995 年發行第一版
- Oracle 在 1978 年發行第一版

歷史悠久的優點

- 社群支援良好
- 大部份軟體錯誤都已經被發現更正
- 有基於實務環境的優化

快速開發 @RDBMS

- 大部份系統， multiple records atomicity 是不能迴避的需求
 - 把錢從用戶 A 轉到用戶 B
 - 購買虛擬道具
- RDBMS 支援多種資料型態
- 對於小型數據庫， RDBMS 作報表是最簡單的
- 大部份商業系統，其資料流失的容忍度都很低的

多種資料型態 @RDBMS

- RDBMS 支援十進制的 numeric ，在計算金錢數值時特別有用

報表 @RDBMS

- 中小型數據庫 = 數據量 < 100GB (個人定義)
- 單純使用 SQL 便可以，不需要再用別的語言去寫程式
 - JOIN 和 sub query 能滿足大部份需求
- 常用的報表工具都內建了 (像 AVG, SUM, COUNT)

高資料安全性@ RDBMS

- 所有 RDBMS 返回 " 成功 " 結果的資料改動，除非儲存空間受損，否則資料永不流失
 - 在 C10K 環境下，concurrent file IO 是超級地難，真正的困難。
- MongoDB(2.6) 的標準設定是 Acknowledged，意思資料未存到硬碟前便返回 " 成功 "
 - 你肯定這樣是安全的嗎？

ACID @ RDBMS

ACID @ RDBMS

- 每家資料庫都自稱支持 ACID ，但是每家所定義的 ACID 都有所不同.....
 - 這問題在 noSQL 世界特別嚴重
- Atomicity
- Consistency
- Isolation
- Durability

A@RDBMS

- RDBMS 所有運作都是以 Transaction (之後簡稱 TX) 為單位
- 一個 TX 可以自由包含多個 SQL 指令
- 同一 TX 內的所有資料改動必須全部被順序執行，或是同一 TX 內所有資料改動都不被執行
- 在當機時，所有還沒有 committed 的 TX 全被 Rollback
- 當機後，需要修復資料時。RDBMS 必須以 TX 為單位進行修復。

Atomicity 重要性

- 梅菲定律：當機是不能避免的
 - 核心思想：別相信主機永遠不當掉，而是讓系統在當掉後不會有數據錯誤
 - 縱深防禦思維很重要，不懂的去玩一下戰略遊戲
- 簡單四句，便能做好了用戶轉帳

```
START TRANSACTION;  
Update user_balance set balance = balance - amount where username = 'UserA';  
Update user_balance set balance = balance + amount where username = 'UserB';  
COMMIT;
```
- Atomicity 保證系統數據從一個正確狀態 (consistent state) 直接移到下一個正確狀態
- noSQL 世界，是需要 9 個步驟的 2 phase commit

C@RDBMS

- 不同人對 Consistency 定義不完全相同
- RDBMS 內有 unique constraint 和用戶自義 constraint，在 TX commit 時，所有 constraints 都必須被滿足
- 否則，RDBMS 將會自動復原當前 TX 一切資料改動

Consistency 重要性

- 使用 RDBMS 時，任何錯誤時把 TX Rollback 便好，而不用思考怎把改動還原。
 - 在 noSQL 的 2pc，有一半以上的程式碼花在當機後怎來復原資料
- Atomicity 保證系統數據安全地移到下一個正確狀態； Consistency 保證系統在移動失敗時，能安全地回到本來正確狀態。
 - 所以，RDBMS 能保證不會發生付了錢卻搶不到票

I@RDBMS

- 同一筆資料， RDBMS 保障不會同時被兩個 TX 改動
- 多層級的 Isolation，能讓 application 只看到應該看到的資料。
- 要避免 race condition， LOCK（上鎖）是不能避免的，而 RDBMS 能自動管理 LOCK

沒有 Isolation 的世界

- 在沒有 isolation 下的提款程式

Step 1: Lock User record in DB

Step 2: GET @balance from DB

Step 3: @newBalance = @balance - @ withdrawalAmount

Step 4: if @newBalance <= 0, return error and exit

Step 5: Set @newBalance to DB

Step 6: Unlock User record in DB

Step 7: 把錢吐出來

- 如果沒有 RDBMS 的 Isolation，便需求額外的 locking 系統
 - 所以 Cassandra 2.0 之前是需要 Zookeeper
- 如果沒有 Isolation 也沒有手動管理的 locking，用戶可以同秒在多個 ATM 提錢去偷錢 (Race condition)

I@RDBMS, 背後的 locking

- update user set balance = balance - \$amount
where user_id = \$user_id and balance >= \$amount
- 在 RDBMS 世界，update 會自動為受影響的數據加上 WRITE lock，並且自動在 TX 結束時釋放，保證一份資料不能被同時改動。
- RDBMS 支持 atomic check-and-set 模式
所以檢查用戶是否有足夠錢，和改動戶口餘額是在同一 statement 內完成

Isolation 重要性 1

- 這是正確（而且簡單明快的購買機票的程式）
 - Step 1: Select price, vacancy from flight where flight_id = @flight_id for update
 - Step 2: If vacancy = 0, rollback TX and return 賣光了
 - Step 3: update flight set vacancy = vacancy – 1 where flight_id = @flight_id
 - Step 4: update user set balance = balance - \$price where user_id = \$user_id and balance >= \$price
 - Step 5: if @@recordAffected = 0, rollback TX and return 用戶餘額不足
 - Step 6: create ticket record
 - Step 7: commit
- RDBMS 的 Isolation 支持全自動化的 lock management，也支持自動化的 deadlock detection。
- 開發者不用自行管理 locking，也不用自行做 deadlock resolution

簡單的戲院售票系統

- 用戶 A 先選擇他想要的座位
 - update seats
set user_id = 'userA' and book_time = '23:14'
where position = '31A' and status != 'sold'
and book_time - now > '30 minutes'
- 用戶 A 以信用卡付款後確定座位
 - update seats
set status = 'sold'
where position = '31A' and status != 'sold'
and user_id = 'userA'

D@RDBMS

- 一旦 Committed 的資料改動，除非存儲空間受損，否則永不流失
 - 前題：你沒有亂改 Linux kernel 和 RDBMS 的標準設定
- 即使在資料寫入時斷電，引發正在寫入的區塊資料流失，RDBMS 也有機制在斷電後復原資料
- High performance + high concurrency + high durability 的系統是非常困難設計的，超級超級超級困難
 - 請不要忽視 RDBMS 超過二十年的歷史
- MongoDB 標準設定是沒有等待 Disk WRITE 的成功寫入，對資料庫知識不足的使用者會造成當機時的數據流失。

淺談 REDO Log

- 對大部份系統，其數據改動是 Random 的
 - 即是說：這一秒用戶 13947 來提錢，不代表下一秒 13946/13958 會走近銀行
- 所以，其 Disk WRITE 也會是 Random IO
- 如果每一個 TX 都要等待其 Random IO 完成，其效能會是慢得不能接受的

REDO Log 圖解 1 (大幅簡化版)

Page 1389

TX-1 對 page 1389 內的 data 作出了改動，在這時候發出了 Commit 指令

Page 4665

TX-2 對 page 4665 內的 data 作出了改動，在這時候發出了 Commit 指令

Step1:

用戶改動資料後，發出了 Commit 指令。這時改動了的資料還停留在 Memory

Step4:

這時候，Page 1389 和 Page 4665 已經在硬碟中，即使斷電了也不會流失。RDBMS 對 TX-1 和 TX-2 發出 Commit 成功的訊息

REDO log1

Step2:

RDBMS 把改動過的 Page 1389 和 Page 4665 以 append 的方式寫到現在正使用中的 Redo log1 檔案的最末端

Page 1389

Page 4665

fsync()

Step3:

RDBMS 對 Linux 發出 fsync()，保證寫入是真的存到硬碟中，而不是還停在檔案系統的 cache 中

REDO Log 圖解 2 (大幅簡化版)

Page 1389

Page 4665

Step5:

然後，RDBMS 的 background writer 把 Page 1389 和 Page 4665 抄回本來的位置。

Page 1389

Page 4665

HDD

REDO log1

Page 1389

Page 4665

REDO log2

Page 9812

Step6:

RDBMS 持續觀察 REDO log1，一旦發現其所有 data page 都儲存好了。便會把 REDO log1 刪掉

Durability 重要性

- 一般商用系統，大部份的數據都是不能流失的
 - 如果銀行告訴大家昨天存入金錢的數據丟失了，不暴動才怪。
~~（你欠銀行的借款數據倒是永遠不會流失的）~~
- 一旦斷電了，RDBMS 將會以 TX 為單位進行 Data Recovery，保證資料庫不會停留在不正確的狀態
 - 所以 RDBMS 使用者不需要（也不應該）處理因為當機而發生的數據錯誤
- 在高流量高同時下的檔案讀寫是很難，真真正正的困難。沒有絕對必要請別自己來做

對 RDBMS 的迷思

對 RDBMS 的迷思

- RDBMS 無法支持 flexible schema
- RDBMS 很慢
- RDBMS 沒有 scalability
- RDBMS 會被 noSQL 替代
- facebook 沒有使用 RDBMS

別誤解 24x7

- 24x7 是指沒有計劃外的 downtime，不是沒有 downtime
- 魔獸世界也是每星期 planned downtime 一次
- 銀行 ATM 也會有 planned maintenance
 - 謎之聲：你肯定全都是計劃中的？？？
- 股票交易所不是 24x7 的

RDBMS 也能 flexible schema ~

- 如果你真的跟隨這頁投影片來做，某隻喵一定對你兇猛撕咬！
- RDBMS 可以建立 table 時，預先建立 col1,col2,col3...col200 的 string column，想要增加 record 的 column 時，隨便挑一個來用便好～
- RDBMS 不是不能 flexible schema，而是中小型資料庫不應該使用 flexible schema

flexible schema

- 這會是使用上 flexible schema 的典型情況：

```
GET object from db
switch (object.version) {
    case "v1" : v1_logic(); break;
    case "v2" : v2_logic(); break;
    case "v3" : v3_logic(); break;
    .....
}
```

- 每多一種 data version，你的 switch 便多個 condition
 - OOP 世界配上 ORM 時也許不會使用 switch，而是每一種 version 有所屬的 Class
- 除非把所有 v1 的 records 轉換成最新的版本，否則 v1_logic() 永遠不能刪除
 - 在「每天都在忙」的公司，v1_logic 很快便會變成「傳家之寶」
- 簡單來說：如果你是在面對 TB 級數據，或是你真真正正需要 zero downtime deployment，你才需要 flexible schema
- 一般中小型系統（總數據量 < 100GB），允許 15 分鐘 downtime，在 deployment downtime 內改動 table schema 更加務實

如果不用 flexible schema

- 在 2018 年今天，主流 RDBMS 一定程度上支持 online add / drop column
 - 會有一定限制，請小心
- 邪惡手段：CTAS
 - Create table as select
 - `create tableX_new as select col1, col2 from tableX` 然後刪掉 `table`，再把 `tableX_new` 改名成 `tableX`
 - 要小心 foreign key，table privilege，table setting

RDBMS 很慢嗎？

- SQL 是宣告式 (declarative) 語言，其支持的功能非常多
 - Joining, subquery, window function, stored procedure, recursive query
 - parsing 還有 execution plan optimization 也是需要時間的
- RDBMS 支持 multi-row, multi-statement atomicity
 - 這需要 lock management 的
- RDBMS 支持 unique constraint 還有 secondary index
- RDBMS 比 noSQL 做這麼多的事。而你卻只說他比 noSQL 慢，你應該看看：That still only counts as one~

RDBMS 真的很慢嗎？

- MongoDB(2.6) 標準設定不等待資料真正寫入 REDO log 便返回「成功」，而正常的 RDBMS 一定會先寫進 REDO log 才返回「成功」的
 - 你的資料庫是用來跑分還是貯存重要數據的？
- Cassandra (1.X)的counter沒有把本來的值存到 REDO log，而只存放 delta，在當機後數值有機會出錯。
- Cassandra 標準設定是 ONE 而不是 QUOTUM，那代表 Cassandra 的 READ 可能返回舊版本的記錄。
- noSQL 需要使用 2 phase commit 去手動處理 multiple record atomicity。
 - 現實上的 2pc 需求多個步驟，很慢的
 - 坊間很多使用 noSQL 的人，都沒有提及 2pc
- noSQL 的「快」，是建立犧牲數據的安全性和正確性上。

RDBMS 沒有 scalability

- 這點我同意的
- 今天 (2018) ，在 amazon 能租上 3904 GB RAM 的主機，你的系統真的需要 scale out 嗎？
- RDBMS 支持 replication ， READONLY 的流量能使用 slave 主機
- noSQL 所需要的額外氣力，是 RDBMS 的 3 倍以上
- 每個老闆都想系統容量無限大的，但是最終需要向開發時間妥協
~~(正如男人都愛年輕正妹，女人都愛高富帥，但最後還是跟普通人結婚)~~
- 先把系統正確地做好了，等到 RDBMS 沒法支持的那一天，你絕對有錢找高手用 noSQL 重寫系統

RDBMS 真的沒有 scalability

- 香港交易所用 DB2 來處理每天數百萬筆股票成交
- 亞洲空運中心（佔 25% 香港空中物流），是使用 Oracle
- 香港海關是使用 Oracle

RDBMS 沒有 scalability，但閣下的系統比這些大嗎？

RDBMS 會被 noSQL 替代

- 在超高容量，超高流量的領域：對的。
 - noSQL 正是發明來解決像 facebook 這種巨型系統的東西
- 今天不會再有人用 RDBMS 作 sharding，而是直接使用 noSQL(HBase / Cassandra) 好了。
 - Line 是用 HBase
 - Facebook / uber 是把 mysql 當成 nosql 的 data node 來使用，這場景下的 mysql 不能看成 RDBMS
- 在 hot data（只有短生存期，流失了也無太大問題的，例子：web 的 session data），像 Redis 這些 key/value DB 一直在大量使用。
- 在中小型資料庫 / 只面對地區性用戶的系統，RDBMS 的可信度，功能性，成熟度不是 noSQL 能輕易替代的。

facebook 沒有使用 RDBMS

- ~~每次在推廣 noSQL 的人總會丟的大絕~~
- 有醫生使用砒霜治療肝癌
 - 連肝癌這麼嚴重的病都能治 - > 砒霜一定是好東西 - > 下次輕感冒時也吃砒霜
- ~~聽說某笨喵終於有老婆了~~~~~ ~~關我屁事!~~
- facebook 和 Line 初期用 MySQL，instagram 初期用 postgresSQL + Redis
- facebook，amazon 正在面對的每天流量，一般系統每年加起來也不會面對
- 這些公司有的錢，一般公司絕不會有
 - 沒錢 = 沒有高手，沒錢 = 沒有開發時間
 - 說服老闆別作白日夢更務實

完

附錄：2 phase commit

- 請參考：
<http://docs.mongodb.org/manual/tutorial/perform-two-phase-commits/>
- 假設我需要從用戶 A 轉錢到用戶 B，邏輯上會有涉及 2 個 record 的 2 個步驟：
 - $\text{userA.balance} = \text{userA.balance} - \text{amount}$
 - $\text{userB.balance} = \text{userB.balance} + \text{amount}$
- 在 noSQL 世界，這需要 2pc 才能維持 atomicity

2 phase commit (續 1)

- 這是便使用 2pc 的轉帳：

- Step 1: 建立 tx 紀錄

- { id: \$id, source: "A", destination: "B", value: 50, state: "initial", lastModified: now() }

- Step 2: 把 tx.id 放到 schedule job，讓 schedule job 在 5 分鐘後檢查這個 2pc tx

- Step 3: 把 tx.status 從 "initial" 改成 "pending"

- Step 4: 檢查 userA.pendingTx 為 null。

- 如是，則 set userA.pendingTx = tx.id, userA.balance -= 50

- Step 5: 檢查 userB.pendingTx 為 null。

- 如是，則 set userB.pendingTx = tx.id, userB.balance += 50

- Step 6: 把 tx.state 改成 "applied"

- Step 7: 把 userA.pendingTx 改為 null

- Step 8: 把 userB.pendingTx 改為 null

- Step 9: 把 tx.status 改為 "done"

2 phase commit (續 2)

- 如果 application server 當掉，需要由 scheduler 事後清理：
- 如果 tx.state = initial：
 - 直接把 tx.state 改成 rollback 便好
- 如果 tx.state = pending：
 - Step 1: 檢查 userA, 如果 userA.tx 符合，則把 userA.tx 改回 null，並且 userA.balance += 50
 - Step 2: 檢查 userB, 如果 userB.tx 符合，則把 userB.tx 改回 null，並且 userB.balance -= 50
 - Step 3: 把 tx.state 改成 rollback
- 如果 tx.state = applied：
 - Step 1: 檢查 userA, 如果 userA.tx 符合，則把 userA.tx 改回 null
 - Step 2: 檢查 userB, 如果 userB.tx 符合，則把 userB.tx 改回 null
 - Step 3: 把 tx.state 改成 done

2 phase commit (續 3)

- 為了能在當機後能清除進行到中間的 2pc，你需要額外建立 scheduler 在未來檢查進度
- 正在改動中紀錄 (pendingTx != null 的 userA 和 userB)，不能被其他 TX 改動
 - 小心 deadlock 唷，你有可能需要自行寫 deadlock detector 的
- 在 noSQL 使用 2pc 會涉及 9 個 logical disk IO，而 RDBMS 的 TX 只涉及 2 個 logical disk IO
- 現實世界的 2pc 很慢，超級慢的
 - noSQL 只是同時能處理更多更多的 TX，但是每一 TX 卻要用上更多的時間

威力加強版

淺談 RDBMS 常犯錯誤

Database-as-IPC

- IPC = inter-process channel
- 特徵
 - 數據的生命週期超級短的
 - 你會找到數據的誕生者，還有數據的使用者
 - 一旦數據使用了，這些數據便不再有價值
 - （不一定）會有不停查詢資料庫的 Request，去看看是否有資料更新
- 例子：
 - 把聊天軟體的訊息全放進 RDBMS，而接收方需要每數秒便檢查一下 RDBMS，看看有沒有新計息
- 解決方案：使用像 rabbitMQ 的軟體來傳播訊息

Database 存放 log

- log 的一般定義：
 - 建立後便不再改動的數據
 - 記錄不會被單一使用的，只會經過 aggregation 後在報表中顯示
- 例子：
 - Web server 的 access log
 - user 的登入和登出時間
- 解決方案：小型系統使用 txt 檔案，大型系統使用專門作 logging 的軟體（例子：Amazon Kinesis）

沒有數據生命週期管理

- 老闆 / 業務人員：我們所有數據都要存起來，全部都不能丟失的
- 硬碟很便宜不值錢沒錯，但是管理數據卻要人力成本的
- 老舊了，沒有商業價值 / 法規需求的數據便應該從主資料庫刪掉
 - 分不清什麼數據才有價值的人員，更應該從公司中刪掉
- 不少報表都需要 full-table scan，越來越多的數據只會讓報表越來越慢
 - 使用 data partitioning 也是一種方案。
 - 但是 data partitioning 讓 table schema 需要砍掉重練，或是付 Oracle 80 萬 NT
(註：data partitioning 為額外付費功能，不包括在 oracle 200 萬 NT 的標準價格中)

威力加強版 淺談 ORM

淺談 ORM

- 在一般 OLTP 系統，有不少的場合都是讀取 / 建立 / 改動 / 刪除單一的物件 (Object)
- 而良好的資料庫設計下，大部份 model class 都是對應單一的資料表的
- 所以，只要你好好設定：
 - 這個 class 所對應的 table
 - 這個 class 內每一 attribute 對應的 column
 - 這個 class 的 primary key

ORM 便能自動幫你產生 select / insert/ update / delete 的 SQL，節省開發時間，也減少人手打字時出錯的機會

- 好的系統，其中 80-95% Query 應該由 ORM 自動產生的

適當使用 ORM

- SQL 本身是宣告式語言，單一 SQL 指令能做到非常複雜的行動
 - 例子：把 id = 89072 的 user 的 is_terminated 改為 true，並且返回他的 account_balance

Update users set is_terminated = true
where id = 89072
returning account_balance
- 如果你要求 ORM 做到 SQL 的一切功能，最終你只會發明另一個 SQL
 - 例子：HQL(Hibernate Query Language)

善用 ORM (續)

- 剛才例子，可以把一個複雜命令切碎成兩個簡單命令：
 - Start TX
 - UPDATE User with id = 89072
 - GET User with id = 89072
 - Commit TX
- 不過，這會引起輕微效能問題
 - 如果不是系統最繁忙的關鍵地方，以少量效能去換取開發時間是可以接受的

這次真的完了~