RDBMS 課程 v2 （第三課）

by

Triton Ho 

前言 1

- 每次有人告訴我資料庫很慢時 ~
  - （沒有誤）有 40% 的 root cause 是 application tier
    - ~~（沒有誤）你看過用 for loop 計算除法的餘數嗎~~
  - 40% 是 schema 設計有問題
  - 15% 是沒用大腦寫的 SQL query
  - 5% 是資料庫設定出錯
    - 在 16GB RAM 的電腦上只使用 256MB
    - ~~在裝有高度敏感資料的資料庫使用「abc123」作密碼~~

前言 2

- Data schema 是系統的基石， data schema 不正確，會讓其上的 application tier 產生扭曲
  - Data model 是「名詞」， application tier 是「動詞」，只有兩者加在一起，才能組成一個系統的
  - 之後開發時只會事倍功半，甚至事倍功零
  - 爛的 data schema 引起的效能問題，是救不回來的
- 雖然 Data schema 也能使用 refactoring 來修正，但是不容易

今天大綱

- 淺談在 OLTP 的 schema 設計
- 淺談 Secondary Index
- OLTP 的 schema anti-pattern
- 淺談 SQL 通用守則

淺談在 OLTP 的 schema 設計

淺談在 OLTP 的 schema 設計

- Normalization 與 modeling
- 面對 OOP 的 inheritance
- 故意的 data redundancy
- Data Materialization
- Data partitioning

Normalization 與 modeling

- 1NF, 2NF, 3NF, BCNF 很重要，不過各位還是回家看書看維基百科好了
  - 反正單單談那些定義，各位睡覺後肯定把東西完整還給我
- 現實工作中，系統設計時一定會有 data modeling 階段
- 一般來說：讓每一個「class」是對應單一 database table，每一「object」是對應單一 database record
  - Class 和 table 應該是一對一的，別試圖把不同 class 的 object 放到同一 table 上
  - 只要不再額外亂做一些「optimization」，由正常有大腦的人做出來的 schema 已經自動滿足了 3NF

面對 OOP 的 inheritance

- 其實 inheritance 在 modeling 只是偶然發生的
  - 例子：在保育動物團體的系統中，「貓」和「兔子」都是「寵物」的 subclass
    - ~~狗不是寵物啊吼！~~
- 應該建立三個表：
  - 寵物＜寵物 ID ，性別，體重，入住時間，領養時間，預計人道毀滅時間＞
  - 貓＜寵物 ID ，毛色，是否已絕育，乾糧需求，罐罐需求＞
  - 兔子＜寵物 ID ，耳朵長度，草料需求，胡蘿蔔需求＞
- 然後，＜貓＞和＜兔子＞建立對寵物的 foreign key

故意的 data redundancy

- Data Redundancy 會違反 normalization 。但實際環境下，如果涉及 aggregation 時， Redundancy 未必是壞事
- 例子：在售票系統中， flight.sold_tickets 和 flight.total_seat
  - 用戶想知道航次還有沒有空位，只需要拿出 flight 這一個 record
  - 無需要 select count(1) from ticket where flight_id = @flight_id
- 而且 Conflict promotion ， Conflict materialization 有時需要額外建立 Record 的（請溫習第一課內容）

Data Materialization

- 在 OLTP ，「報表」會是不停地產生和使用的。其內容卻不一定需要即時性準確的
  - 例子：在討論區中，「最熱門話題」，「最新話題」
  - 誰告訴你「報表」一定是 PDF / excel 的
- 對於「最熱門話題」，可能是每五分鐘實行一次 Query ，然後把 query result 放到一個 table （放到 Redis / memcache 也可以）
  - 之後的五分鐘，所有「最熱門話題」的 request 都使用這份預先準備好的數據便好
- 這種使用預先計算的數據，便是 Data Materialization
  - 越是大型的系統，便越需要使用 Data Materialization
- Oracle 支持全面自動化的 Materialization ，這樣連 schedule job 也不用寫

Data Materialization 注意事項

- 以下是正確的 on-the-fly data materialization
  - Step1: 如果 precomputed resultset 的 timestamp 在容許範圍內，直接返回 resultset
  - Step2: 拿取 X lock
  - Step3: 再次檢查 resultset 的 timestamp。如果在容許範圍內，釋放 X lock 並且返回 resultset
  - Step4: 建立 resultset
  - Step5: 釋放 X lock
  - Step6: 返回 resultset
- 如果沒有 Step2,3,5，在高流量系統中 precomputed resultset 一旦過期，將會引發多線程同一時間建立 resultset。
  - 小心會當掉！

Data fan-out

- Data Materialization 的再進一步
- 例子： facebook newsfeed
  - 每人的 newsfeed 都是預先計算，而不是即時地查詢的
  - 用簡單的 Range Scan on PK 便可以
- 今天 Disk Volume 不值錢， Disk IO 才值錢

Data partitioning 前言

- 部份資料，擁有固定的「最長生存期限」
  - 例子：在空運貨運站主系統，所有的貨運紀錄只有１４天的生存期（超過１４天的貨運紀錄只用作統計用途，所以不需放在主系統）
- 如果使用 schedule job ，在每天系統相對地清閒的時間把過期紀錄清掉
  - 清理過程時會引發大量 IO ，大幅拖慢系統效能
  - 系統又多一件東西需要監控 =_=

Data partitioning

- Truncate table是把整個table所相關的file刪掉再重建，沒有UNDO也沒有REDO，所以效能超快的
  - （所以除DBA外，普通人不應該有這權限）
- 以空運貨站作例子：
  - 建立cargo_0, cargo_1, cargo_2, …… cargo14共１５個subtable
  - Insert cargo時，根據daydiff('1900-01-01', record_create_date)來決定應該放到那一個subtable內所以同一天的cargo record都會集中在同一subtable中
  - 每天truncate一個subtable，一次性把１４天前的cargo record清掉～
- 在Oracle，這是付費的功能（好像８０萬ＮＴ），在postgreSQL和MySQL， data partitioning需要application tier配合，或是要使用trigger和view
- 補充：data partitioning和data sharding不是同一種東西，而今天在noSQL全面普及下， data sharding on RDBMS已經沒落

淺談 Secondary Index

- Secondary Index 前言
- 要增加 index 前要問自己的問題
- Foreign key 與 index
- Loose index

Secondary Index

- 一般人口中的「為 table 加上 index」便是指 secondary index
  - 又名 non-clustered index
- Clustered index 是指第二課 Index-organized table 的 PK
  - 再說一次： IOT 本身便是一個以 PK 去排序的巨型 B+ tree

Secondary Index 前言

- Oracle 說過：每增加一個 index ，該 table 在 insert / update / delete 時便慢 3 倍
  - index 是犧牲 WRITE 效能去提升 READ 的效能
- 每增加一個 index 時～
  - 便有額外要改動的 index data page （請溫習第二課）
  - 便有額外可能性因為遇上 index page lock 而需要等待／發生 deadlock
- Bitmap index 的改動是單線程的， hash index 是不安全的
  - 如果你堅持在 OLTP 使用，祝君好運

要增加 index 前要問自己的問題

- ~~（無誤）今天有沒有善待貓咪？~~
- 如果你要增加的是 unique index ，你有強烈原因不讓它作為 table 的 Natural key 嗎？
- 你要增加的 index ，是幫忙 OLTP 行動嗎？
- 如果是 non-unique index ，你假設會用上的 query ，它能幫助你把 candidate records 數目變到 100 下嗎？
  - 如果是 enum column （像是 gender）， B+ tree index 幾乎肯定幫不了你
- 你是否真的需要「即時性」的資料？

是否應該用 index?

- 如果你的 table 已經做了 time-based partition
  - 你的 time-based query 只會用上數個 sub-table
- 偶然要跑的報表， Full-table scan 其實沒大家想像中可怕
  - 一個擁有 1M 筆紀錄，每筆紀錄大約 100bytes 的 table ，其物理空間不會超過 500MB
  - 利用 Sequential Read ，所需時間不應該超過 5 秒（如果超過了，你是時候考慮換工作了）
  - 那些＞ 100GB 級的，很多都是用作數據分析的。
  - 人們說自己的 table 很大，很多時候都是因為沒有把過期資料清掉！
    （掌握香港 25％空運貨物的系統，主資料庫才＜ 50GB）
- 如果用戶 95% 時候只關心「活躍」的 Record ，你可以考慮把 table 分割
  - 例子：在工作管理系統中， 95% 時間人們只關心還未完成的工作
  - 所以不是單一的 <Task> table ，而是會分割成 <unfinished_task> 和 <finished_task>
  - 當工作完成時，便會從 <unfinished_task> 刪除，然後增加到 <finished_task>

index 與 Good schema

- 良好的 schema ，能讓一些 secondary index 不用建立的
- 例子：在某遊戲系統中，用戶可以使用 user_name+password 或是 facebook 身份登入，而 95% 用戶都只會以單一方法登入，不會 2 者混用
- 如果只有一個 <user> table ，其內有 user_name, hashed_password, facebook_id 這些 columns ，便需要為 user_name 和 facebook_id 分別建立 secondary index

index 與 Good schema （續）

- 如果是 3 個 table: user, user_name_login, user_facebook_login ，便不再需要建立 secondary index
- 要以 facebook 身份登入時：

```
select user.* from user u
inner join user_facebook_login fb on u.id = fb.user_id
where fb.facebook_id = @facebook_id
```

- 即使最終還是有 3 個 index （ PK 背後也有 index 的），可是 username 和 facebook_id 這 2 個 index 卻不再包括無關的 null value
  - 讀取時快了（因為 index 體積小了）
  - 建立新用戶時也快了（因為不用再把 null value 放進 index ）

Foreign key 與 index

- 這是 parent and children
  - 你經常性利用 parent 的 PK ，去找出全部 child record
    - 例子：你需要找出航次 'BR827' 的座位表
  - 不會有大量的 child FK 指向同一 parent record
- 如果同時滿足以上兩點，你才需要考慮為 FK 建立 index

Loose index

- 對於 composite key<colA, colB, colC>
  即使你 Query 中只有 colA / (colA + colB) 的值， RDBMS 還是有機會用上 loose index
  - Oracle 全面支援
  - MySQL 很有限度支援
  - postgreSQL 不支援 =_=
- 在支援 loose index 的 RDBMS 要建立 composite PK 時，一般守則：最有機會在 where-clause 的 column 會放在前面
- 如果你在能支援 loose index 的 RDBMS ，使用 natural key 有機會能節省 Foreign key 的 index
  - 如果使用 natural key ， child table 的 composite key 內很可能包含了對 parent 的 FK
  - 例子：「座位表」的 PK 是 < 航次編號，座位編號 > ，而航次編號正正是「航次」的 PK

OLTP 的 schema 設計常犯錯誤

anti-pattern 前言

- 接下來的 anti-pattern ，大家總會聽得非常歡樂
- 哈哈哈哈！我才不會犯這種錯誤啦！
- 可是，我接觸過 schema 有問題的系統中，幾乎全部都不同程度地犯上這些 anti-pattern
- 除了 smart column 的影響相對上輕微—點點，其他的都是極度致命的（長期爆肝會死的）

OLTP 的 schema anti-pattern

- “smart” column
- denormalization
- 多用途 column
- 多用途 table

“smart” column

- 1NF 的定義：所有 column 的 value 都必須是 atomic value
- 就是說： xml, json, array 這些東西全都違反 1NF
  - ~~這個 postgreSQL 傳教士居然建議人們不要使用 postgreSQL 的專屬功能~~
- xml, json, array 是某一 RDBMS 的專屬功能，不一定所有 db connector 和 OR/M 支持
- Xml, json 內有過份高的自由性，未把 column value 內容拿出來前，永遠猜不到 column 內具體存了什麼東西
  - 請溫習 flexible schema 的缺點
- “smart column” 內部資料不能再建立 index ，當報表要用上 smart column 內部資料時作 filtering ／ joining 時，不但 Query 難於編寫，其效能也會低下

“smart” column （續）

- 除了 xml, json, array ，一般人偶然會不知不覺犯下 smart column 錯誤
- 我坦白地說，空運貨物管制系統中， flight 這個 table 的 Primary Key 不是 flight_no ，而是 <airline_code, flight_code>
  - 即是說：不是以 <”BR892”> 作為 PK ，而是 <”BR”, “892”> 作為 PK
  - 之前投影片空間不足，所以我才使用 flight_no 方便解說
- 系統想要找到長榮航空的所有航次時
  - 只需要 where airline_code = 'BR' 便好
  - 而不是 where flight_no like 'BR%'

denormalization

- 為了迴避 Subquery 和 Joining，有人想進行 denormalization，故意地在一個 table 放進 multiple data model 的數據
  - 例子：因為用戶在檢查自己所買的機票時，頁面也會顯示航次出發時間／航次到達時間的資訊，有人做了 denormalization，在 <ticket> 這一個 table 中也存放了航次資訊
- OLTP 的 denormalization 一般無助提升系統效能，反而大幅增加改動資料時的麻煩／出錯可能性
- denormalization 讓 table schema 變得不再 Intuitive（直覺化）
  - Documentation？你跟我認真的嗎？
  - 三個月後，有多少人還記得自己前做過的東西（特別是長期爆肝下）
- 在 OLTP 使用 denormalization 的人，會被詛咒的（無誤）

denormalization 和 rare condition

- 前情提要，有混蛋使用 denormalization ，把航次資料也存到＜機票＞ table ，所以改動航次資料時要改動所有人的機票 Record

普通人購買機票：
Start transaction Read committed;

Select depart_time from flight
where flight_name = 'HKG-->BKK';
/* 因為航空公司還沒有 commit ，所以顯示舊的 1800 */

Insert into ticket blablabla.....
/* 把舊的 depart_timey 放到 ticket*/

Commit;

航空公司同一時間改動起飛時間：
Start transaction Read committed;

Update flight set depart_time = '1815'
where flight_name = 'HKG-->BKK';

Update ticket set depart_time = '1815'
where flight_name = 'HKG-->BKK';
/* 另一邊正在出售的 ticket 還沒有 commit ，所以沒有被 update*/

Commit;

結論：這次的 rare condition 讓其中一個乘客的機票資訊頁面顯示錯誤的舊資訊。在 denormalization 下，更容易因為 developer 不小心而發生 rare condition

多用途 column

- ~~你的系統是用來生產瑞士軍刀的嗎？~~
- 分辨方法：系統需要知道這個 column ／其他 column 的 value ，才知道具體怎去使用這個 column 的內容

多用途 column 例子

- 例子：之前能讓用戶以 username/password 或是 facebook 身份登入的系統
  - 笨蛋建議反正用戶只會以單一的方式登入，所以 user 這個 table 只有一個 auth 的 column
    - 以 username 登入的用戶， auth 的內容是： username:<username>/<hashed_password>
    - 以 facebook 身份登入的用戶， auth 的內容是： facebook_id:<facebook_id>
  - 笨蛋聲稱，這樣的做法可以只使用一個 column ，也只有一個 secondary index
- 致命缺點
  - 這設計只能讓用戶以單一方法登入
  - 因為 hashed_password 是以 salt 和 password 來計算的，這設計讓 index 失效
  - 如果要知道系統有多少 facebook 用戶，便需要 where auth like 'facebook_id%'

多用途 table

- 多用途 column 的進階版本
  - 此 anti-pattern 一旦發生，系統必亡
- 分辨方法：
  - 某一 table 內，大量 column 的 value 是 null
  - 這個 table 內有一個叫「 type 」或是「 kind 」的 column ， application 需要先知道這個 column 的值才能決定這個 Record 怎麼使用

多用途 table 例子

- 某空運系統中（如有雷同，實屬不幸）
  - 出境貨物和入境貨物都放到 <freight> 這個 table ，然後 freight_type 這個 column ，I 代表入境貨物， E 代表出境貨物
- 結果某天某個 procedure 出錯， type 由 I 改成 E ，然後引起全面性數據錯誤
- 對於出境的系統流程， freight 當然只跟出境相關的 table joining ；相反，入境時只跟出境相關的 table joining
- 而且，出境貨物的 record stat 跟入境貨物的 record stat 是不同的
- 結果， optimizer 對於這種擁有雙重性格的 table ，常常使用了非最佳化的 execution plan

淺談 SQL 通用守則

淺談 SQL 通用守則

- SQL 是宣告式語言
- SQL 是 set-based language
- SQL 支持 check-and-set
- SQL 支持 stored procedure
- SQL 應該很短的
- 善用 Temporary table 寫報表
- 對 subquery 的迷思
- 對 joining 的迷思

SQL 是宣告式語言

- 一般來說： schema ＋ query 都正確， execution plan optimizer 應該自動地找到最好的 execution plan ，不應該由 developer 人工控制 execution plan
  - Execution plan hints 例子： MySQL 的 using indexXXX for join ， Oracle 的 /*+ USE_NL (glcc glf) */
- 一般來說： execution plan optimizer 出錯都是因為不合理的 table schema(80%) ／亂來的 SQL(20%) ，請把氣力花到正確地方上
- 今天所使用的 hints ，很可能只針對當前的 data ／運行環境，一旦 data ／運行環境不同了，有可能比 optimizer 的 execution plan 更慢

SQL 是 set-based language

- 不應該輕易使用 looping 的
- 錯誤示範：
  - flight_no_array := []string{“BR892”, “BR893”}
    for _, flight_no in range flight_no_array {
    executeStatement(`Update flight set enabled = false where flight_no = $1`, flight_no)
- 正確示範：

  Update flight set enabled = false where flight_no in ('BR892', 'BR893')
- 如果使用 looping 一個接一個地發出指令，每個 record 都需要等待 network IO 的時間
  - 請記住：TX 的時間越長，他所持有的 resource 便越有可能被其他 TX 需要，進而阻礙到其他 TX 的運行

SQL 支持 check-and-set

- 現在主流的 RDBMS 還有 connection driver ，在 insert/update/delete 時都會返回有多少 record 受到影響的
  - 都 2018 年了，還不支持的丟了它吧
- 所以對於簡單的 checking ，可以跟 data processing 合成一句 statement
- 例子：檢查航次是否還有空位，如有便把空位數目減一

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

  如果受影響的 record 數目＝ 0 ，直接告知用戶交易失敗便好

SQL 支持 stored procedure

- 對於 data-intensive 的演算法，可以考慮使用 stored procedure
- 例子：使用 Dijkstra's algorithm 去找最近的飛行路徑
  - 如果不使用 stored procedure ，每一次演算法的 iteration 便需要查詢資料庫一次
    - 即使說：要等待 network IO 一次
- 不過， stored procedure 沒有 OOP 也沒有好用的 library ，請再三思考是否使用 stored procedure 還是把演算法放在 application tier

SQL 應該很短的

- SQL 是宣告式語言，每一行都能做到很多事的
- 個人看法：非報表類的 SQL statement 應該 10 行內，報表類的 SQL 是 50 行內
- 太長的 SQL 不代表你很神，只代表你這個神經病人不懂使用 transaction 還有暫時性報表，製造了難以維護的程式碼
  - ~~別問我寫過的最長 query，我也曾經年輕過的~~
- 越長的 statement， execution plan optimizer 需要的時間會幾何級地上升 (exponential increase~）
- 越長的 statement，便越有可能讓 optimizer 神經病，使用了 sub-optimum 的 plan

善用 Temporary table 寫報表

- 所有寫進 Temporary table 的資料在 Connection Close / TX commit 時都會被刪除
  - 所以不會使用正常的 tablespace ，速度比一般 table 更快
- Temp table 是每個 Connection 專屬的，不會互相干涉
  - 所以不會有 locking ／ blocking ，速度很快的
- 所以如果你的 SQL 太長，可以考慮把 SQL 變成數個 statement ，然後把中間的結果放進 temp table 內

善用 temp table(1)

- 我想對多筆資料的 colX 改動， RowA 的改成 10, RowB 的改成 11, RowC 的改成 12
- Insert into tempTable (id, newValue) values ('RowA', 10), ('RowB', 11), ('RowC', 12);
- Update TableZ
  set colX = tempTable.newValue
  from tempTable
  where TableZ.id = tempTable.id

善用 temp table(2)

- 如果我要從 Table1 拿 colX = 'A' or 'C' or 'E' 的資料
  - Select * from Table1 where colX in ('A', 'C', 'E')
- 如果我要從 Table1 拿 colX = 'A' or 'C' or 'E' or…… （超過１０００個值）
  - Insert into tempTable(colX)
    values ('A'), ('C'), ('E').....
  - Select * from Table1
    inner join tempTable on Table1.colX = tempTable.colX

對 subquery 的迷思

- Oracle 和 postgreSQL 的 execution plan optimizer 都懂得 subquery unnesting
- 在 oracle 和 postgreSQL 中，這三句 query 沒有任何分別
  - Select child.id from child
    inner join parent on child.parent_id = parent.id
    where parent.name = 'TritonHo'

  - Select child.id from child
    where child.parent_id in
    (select id from parent where parent.name = 'TritonHo')

  - Select child.id from child
    where exists
    (
        select 1 from parent
        where child.parent_id = parent.id and parent.name = 'TritonHo'
)

對 subquery 的迷思（續）

- 在 Oracle / postgreSQL ，大部份 subquery 會被 optimizer 自動轉成 joining
  - 所以對 Oracle / postgreSQL 用戶， subquery 一點也不可怕～
- 以下的 correlated subquery 沒法被 unnest 的，請注意：
  - 含有 aggregation function(count / sum / group by)
  - 含有 window function(rank, limits)

對 joining 的迷思

- 50% 對我說 joining 很慢的人，其 schema 設計有問題
- 40% 對我說 joining 很慢的人，用上了很「天才」而且很長的 Query 去做很簡單的事
- 10% 對我說 joining 很慢的人，是用來跑真的很慢的報表
- 真正會慢的，是 anti-join

對 joining 的迷思（續 1）

- 例子：顯示機票時，連帶顯示航次的資料
- 正確寫法（這例子用了 Surrogate key）
  - select * from flight_ticket ft
    inner join flight f on ft.flight_id = f.id
    where ft.id = @ticket_id
- 因為有 where ft.id = @ticket_id 這個基於 PK 的 filtering， flight_ticket 只有一個 record 會參加 joining
- 因為 f.id 是 Primary key，所以會有 index，所以在 nested loop join 中，這會是內層的 loop
  - 這種 joining 正確名字叫 index nested loop join， OLTP 經常用上這種 joining

對 joining 的迷思（續 2）

- 剛才例子， RDBMS 的內部運作應該是：
  - Step1: 用 where ft.id = @ticket_id 過濾 flight_ticket
  - Step2:
    - for each candidate record ft in flight_ticket {
      f := flight.getRecordByIndex(ft.flight_id)
      如果有相配資料，便把 <f, ft> 加入結果中
      }
- 這個例子，即使兩個 table 各有 10M 個資料也好，也只有 2 次 getRecordByIndex 便能找到結果
  - 這個例子，使用 joining 比分拆成 2 個 query 更快
  - 在 OLTP 中，為了迴避 joining 而作 denormalization 的人，去死好了

避免使用 anti-join

- 我想找出所有飛機票，飛機其不是在 T1 航廈起飛
- anti-join 寫法
  - Select * from tickets
    where flight_no not in (select flight_no from flights where terminal = 'T1')
- 不用 anti-join 寫法
  - Select * from tickets
    EXCEPT
    Select * from tickets
    where flight_no in (select flight_no from flights where terminal = 'T1')

完