RDBMS 課程 v2（第二課）

by

Triton Ho 

前言 1

- ~~為何我需要知道內部結構？~~
  - ~~（絕對誤）因為笨喵在賣弄他的 DBA 知識~~
  - ~~其實真正深層的東西我已經沒有教子~~
- 理想世界的 data abstraction ，當然不用管底層
  - 反正我只是一個 developer~
- 但是在 C10K 環境，資料庫的內部結構會大幅影響效能
  - Web server 可易輕易加裝新電腦， database server 卻不行
- 如果完全不理解資料庫的內部結構，沒法設計出能真正面對 C10K 的良好系統架構的

前言 2

- 大部份公司，都沒有專職的 DBA ，一般性的資料庫管理最終還是由 Developer 負責
  - ~~想對老闆說的話：不用說「辛苦了」，請把誠意反映到年終上~~
- 現實社會中，自稱是 DBA 卻連 backup and recovery 也不會的「 DBA 」（例子：某隻喵）到處都有，系統癱瘓時還是自己最可信任

前言 3

- 第一課的內容是關於兩個 TX 對同一份資料的活動，而引起的 blocking
  - row-level lock
  - 邏輯上相關的
    - Mysql 的 Gap lock…… 唉
  - 相對長時間的（直到 TX 結束）
- 這課的內容，是關於兩個 TX 所改動的資料物理位理相近，而引起的 blocking
  - page level lock
  - 邏輯上不相關的
  - 相對短時間的（直到那個 data page 使用完了）

今天大綱

- 資料表的內部結構
- 淺談 B+ tree
- 淺談 heap table 和 Index-Organized Table(IOT)
- Primary Key 的兩大流派
- 淺談 Primary key 和 Contention

資料表的內部結構

淺談檔案結構

- 大部份的 file system ，都會把硬碟分割成很多小區塊
  - 「區塊」(blocks)，也有人叫「分頁」(pages)
  - 因為硬體有這些限制
- 一個檔案可以佔用多個區塊，但同一區塊只能被一個檔案佔用
- 即使檔案只有 1bytes ，在以 4KB 為區塊大小的檔案系統中，他還是用掉了 4KB

幻想中的儲存方式 1

- RDBMS 為每一 Record 建立檔案
- 大量的空間會被浪費
  - 因為硬體操作是以區塊為單位的，這樣會浪費大量的 IO
- 大量的 file descriptor 會浪費大量系統資源
  - 如果你的系統還沒有當掉

幻想中的儲存方式 2

- RDBMS 建立一個巨型的檔案，並且把這個檔案看成連續性的空間。把同一 table 內的 Record 緊密地（一個接一個地）存到這個連續性的空間內。
- 在初期階段，這的確最節省空間還有 IO 啦
- 在 Delete / Update（ varchar 變短）時，便會留下空洞
- 如果 Record 是需要排序的，每次的 Insert / Update（ varchar 變長）都會引起整個 table 重建

現實中的儲存方式

- RDBMS 建立一個 / 數個巨型的檔案，然後把空間切成 8KB(postgreSQL, Oracle) ， 16KB(MariaDB) 的區塊
- 一個區塊會被多個 record 共用，並且緊密地排列
  - 不管 Create / Update / Delete 也好，要重新整理 8 / 16KB 的數據還是很簡單的
- 如果是需要排序的，區塊會用 double-linked list 的方式連結
  - 即是說：每一個區塊都存有之前區塊和之後區塊的指標 (Pointer)
  - Insert / Create 不夠空間時，插入新區塊在 double-linked list 便好
  - 如果有區塊不再存有數據，從這個 double-linked list 移除便好

使用區塊的好處

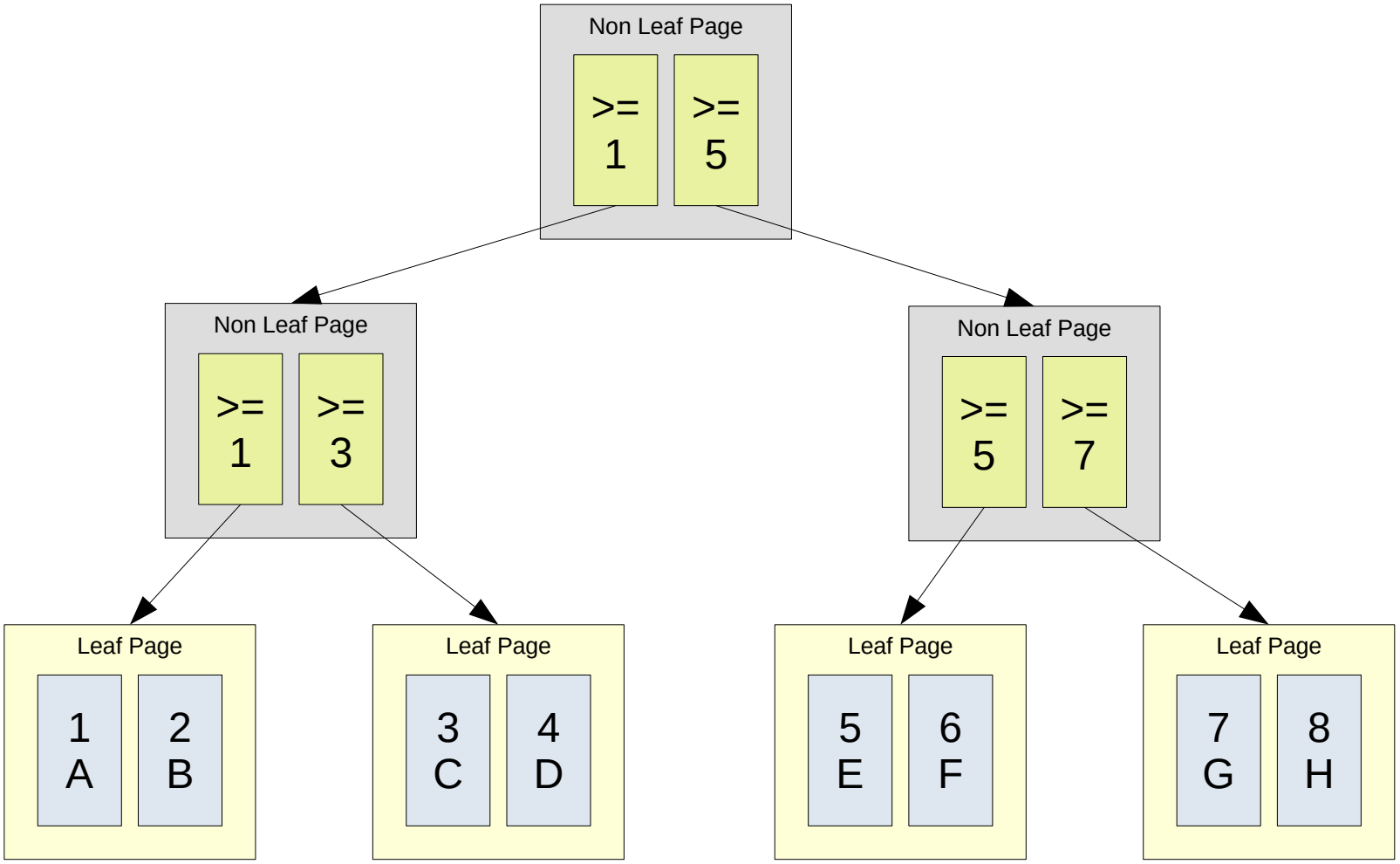
- 能讓 records 緊密地儲存（雖然區塊末端還是有剩餘空間）
- 因為空間是集中在區塊末端而不是在每一 records 之間
  - 所以空間碎片化會大幅減少
- 要找到特定 Record ，只需要知道其所在區塊便好
  - index 體積會比較小
  - 在管理 IO 時，只需要維持（相對）簡單的 page SX lock ，而不用 range lock

這些和 developer 有關係嗎？

- 每個 page 在同一時間只能被一個 TX 改動
  - 因為 page 在被改動的短時間，整個 page 會被加上 page X lock ，暫時沒法被讀寫
  - 即使這個 page 在 RAM ，即使改動這個 TX 只使用 1 ms ，那麼這個 page 每秒只有 1000 WRITE
  - 現在 cpu 成長開始停下來，不能再靠換上更快 cpu 去解決
- 把性質相近的 Record 物理性放到一起會提高 cache hit rate ，但是更容易發生 Contention
  - 就是說：單一 page 能儲存越多的東西， READ 的效能會越好，但是 WRITE 效能會越差
- 部份 RDBMS 會因為 page lock 而發生「特殊現象」
  - Oracle 的 ITL_waits
  - MySQL 5.6 的 index lock

淺談 B+ tree

B+ tree 例子


Non Leaf Page
>= 1
>= 5
Non Leaf Page
>= 1
>= 3
Non Leaf Page
>= 5
>= 7
Leaf Page
1
A
2
B
Leaf Page
3
C
4
D
Leaf Page
5
E
6
F
Leaf Page
7
G
8
H

淺談 B+ tree

- 主流的 RDBMS(PostgreSQL, MariaDB, Oracle) 用來實現 Indexing 的方式
- Primary Key(PK) 和 Unique Constraint 背後也是 index
  - 即使用戶不提供 PK ， RDBMS 還是會偷偷地建立的

為何 B+ tree

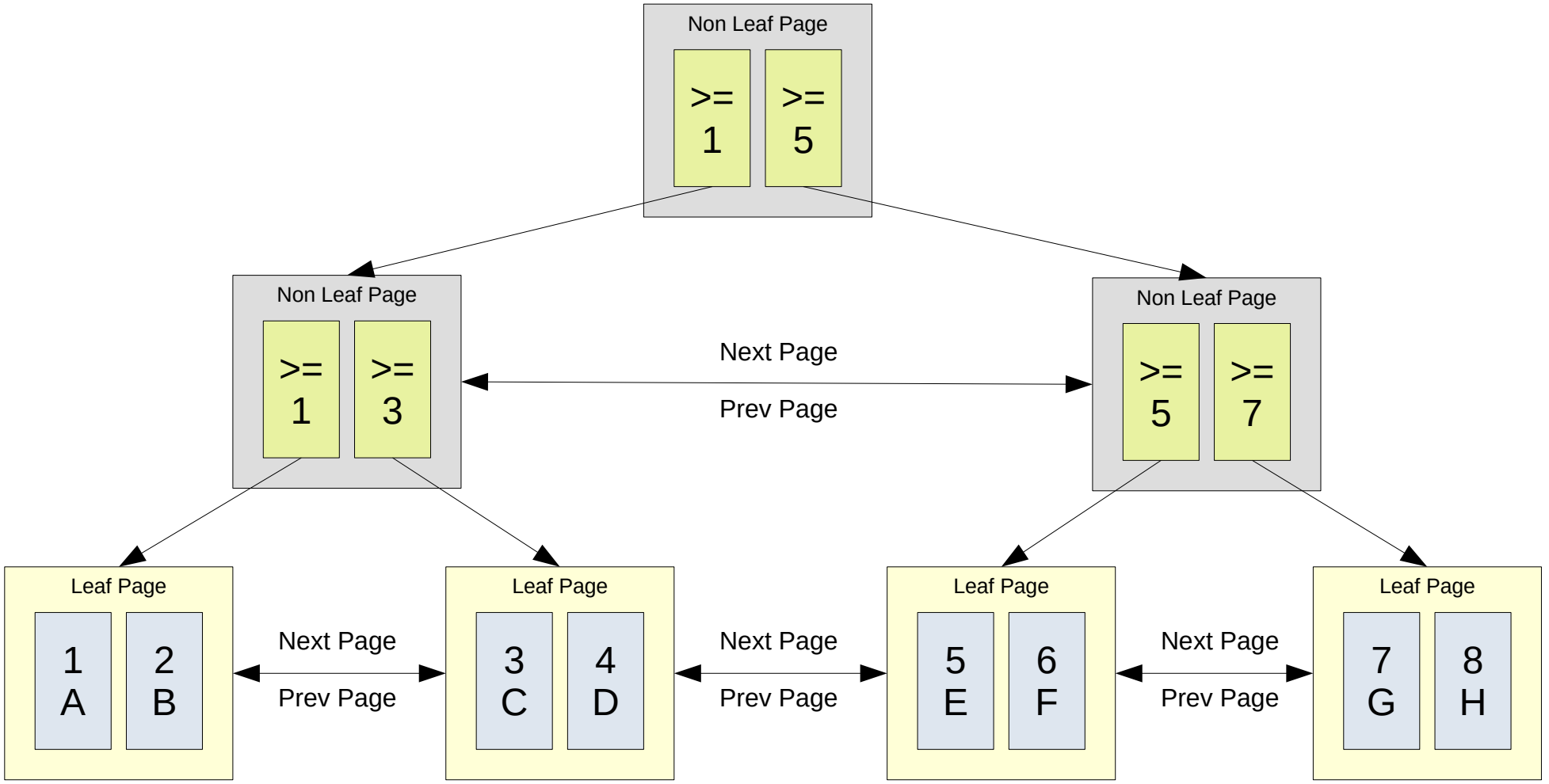
- B+ tree 的高度是 O(log n) 的，一般來說都 <=4
  - 所以，只用 4 個 disk READ 便可以找到記錄
  - 現實世界， B+ tree 的 non leaf nodes 因為經常被使用，長期會留在 cache 中，所以真實的 disk READ ~= 1
- 如果 2 個 TX 改動的 record 不在同一 data page ，他們便能同時更動 B+ tree
- B+ tree 是 auto balancing 。（相對上）不太需要重組資料來維持其效能
  - balanced 和 even distributed 不是同一概念，請注意

還是傳統的 B+ tree 最好 ~

- tokuDB 的 Fractal Tree Index 只是看起來很快，在現實環境中使用是惡夢
  - Fractal Tree Index 和 Fractal（碎形）無關，是 B+ tree 的特殊變種
  - 現實環境不是單線程的， tokuDB 的所有 WRITE 都集中到 root node ，極容易發生 contention
  - 經常發生頂層的 Flushing ，讓頂層的 page 被加上 X lock ，讓整棵 tree 在 flushing 期間沒法被使用

淺談 heap table 和 Index-Organized Table(IOT)

淺談 Index-Organized Table



IOT 優點

- 因為 Records 全都順序整理好了，在 range scan on Primary Key（之後簡稱 PK ）時非常快
  - 例子：在 facebook ，你是需要拿同一討論串下的所有留言，會用上 range scan on PK 的
- 能省下 sorted by PK 的時間
- 只需要一次（邏輯上） READ 便能拿到資料，需要較少 Storage IO

IOT 缺點

- 因為整個 Records 都存放於 B+ tree 的 leaf node ，每個 leaf node 能存的 rows 有限
  - 因此，經常會發生 leaf node 的 splitting / merging
  - 因為 leaf node 的 splitting / merging ，引發 rows 要物理上移動位置
- 如果 PK 是有規律的（例子： auto-increment ），讀取 / 寫入很容易集中在少量的 leaf-node 中，引發 Contention 問題
  - 因為 Contention 問題而發生意想不到 blocking ，常常接下來引發更多的 blocking
  - 同一時期產生的 rows 常常會在相近的時間 delete ，這樣會 rows 不平均地放到 leaf node 中，進而需要 leaf node merging

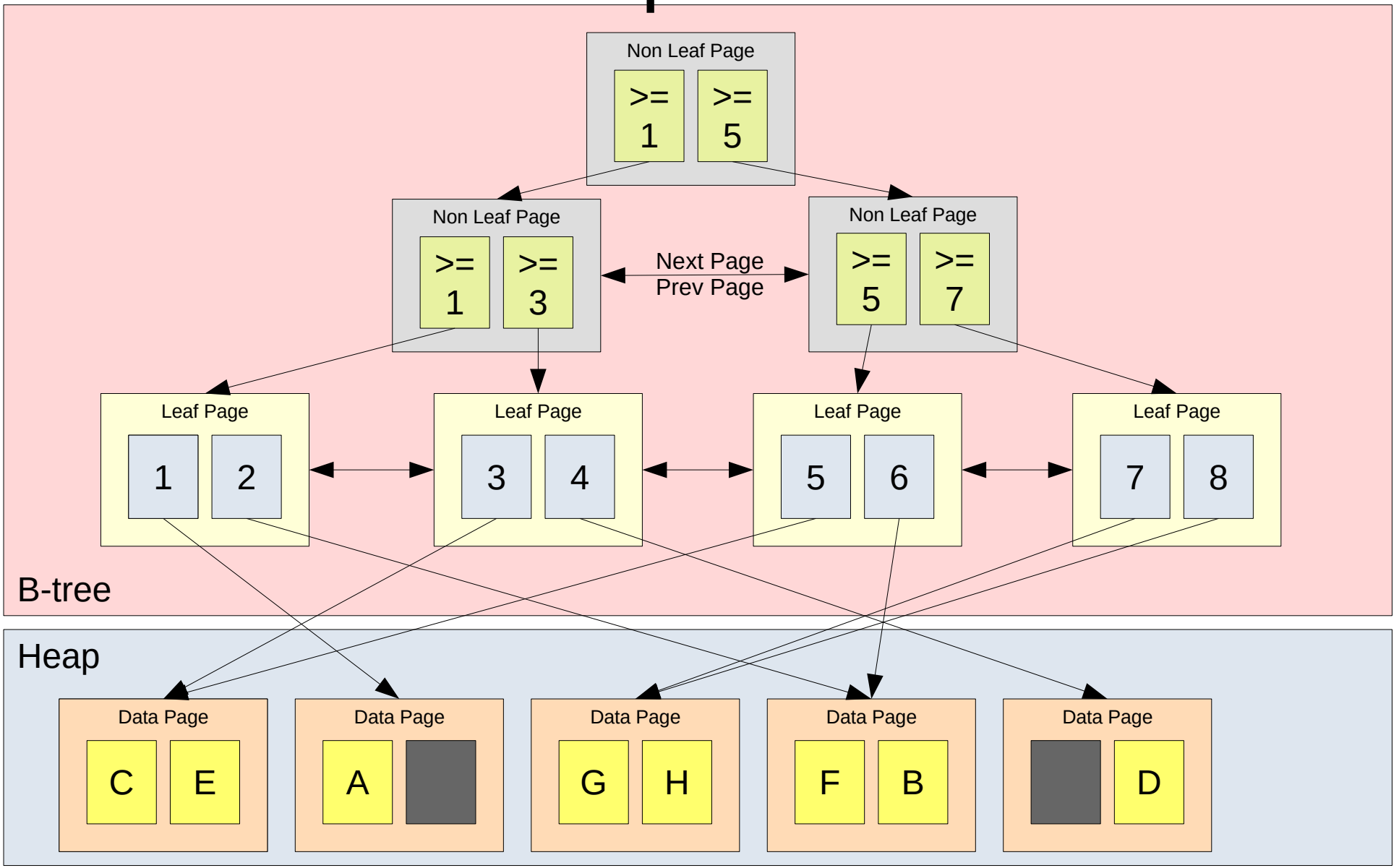
IOT 缺點（續）

- 正正因為 row data 會因為 page merge/split 而改動其物理位置。所以， secondary index 只能儲存那個 record 的 PK ，而不是其物理位置
  - 對使用 natural key 作 PK 的 table ，其 secondary index 會很肥大
  - 使用 secondary index 去找資料時，需要額外的 disk READ 在 IOT 上

淺談 Heap Table

- Heap 不是指 priority list 。應該說：這是指一個沒特定排序的空間。
- 最簡單來說， record data 「隨意」找一個 data page 存放。 PK 獨立放在一個 B+ tree index ，並且在 index leaf node 儲存指向 data page 的 pointer

Heap Table


Non Leaf Page
>= 1
>= 5
Non Leaf Page
>= 1
>= 3
Next Page
Prev Page
Non Leaf Page
>= 5
>= 7
Leaf Page
1
2
Leaf Page
3
4
Leaf Page
5
6
Leaf Page
7
8
B-tree
Heap
Data Page
C
E
Data Page
A
Data Page
G
H
Data Page
F
B
Data Page
D

Heap Table 優點

- 因為 index leaf node 只存放 (PK + pointer)
  - 一個 index leaf node 可以存放更多的 rows， leaf node splitting/merging 自然大減
  - 即使發生 index leaf node splitting/merging ，也不會令 row data 需要移動位置
- Record data 能存放到 heap 中任何一個 data page ，沒有指定位置
  - 即使 PK 是用上 auto-increment ，相近 PK 的 row 會散落到整個 heap 之內，先天性不容易發生 data page contention
  - 在 insert new rows 時， Record data 輕易能找一個沒有正被改動中的 data page 來寫入。不容易發生 blocking
- 現實中的 RDBMS 會有 Free Space Management ，近期被寫進資料的 data page 會優先被重用
  - 我的 laptop SSD 4K Random WRITE 只有可悲的 17 Mib/sec ，但是 postgreSQL 能每秒處理 8000 個 insert
  - 別再相信 heap table 會引發大量 Random IO 很大這種謠言了

Heap Table 缺點

- Range Scan on Pk 一般需要整個 table 都作一次 scaning ，極吃 IO
  - 一般商業系統（ OLTP ）極少使用 PK 作 Range Scan ，問題不太大
  - 但是，如果是專門用作數據分析 / 產生報表的 OLAP ， Range Scan on PK 倒是常常發生
- 需要兩次（邏輯上） READ 才能拿到資料，需要較多 Storage IO
- 萬一發生 index page contention ，後果更加致命

小結

- 有沒有 range scan on PK，決定了你應該用那一種 table structure
- Cache hit rate 是應該被重視的，但不應被過份追求
- Contention 需要更快的 CPU single thread 效能，這方面成長的是越來越慢的
- IO throughput 可以用 RAID 10 來解決
  - 而且 SSD 的 Random IO throughput 也越來越好

Primary Key 的兩大流派

Primary Key 的兩大流派

- Natural Key
  - 使用自然存在的 unique key 作 PK
  - 例子 : 如果用戶以 email 登入系統的，使用 email 作 PK 便好
  - 例子：貨運站和香港海關都是以 (航次日期 + 航次編號) 作 PK
- Surrogate key
  - 人工合成的 Key ，在商業邏輯上一般上沒有任何意義
  - 一般來說： auto increment 或是 UUID
    - 補充： auto increment 不保證是順序和連續的
  - 除非商業邏輯有特殊需求，請不要自行創作奇怪東西 > < "

Natural Key 優點

- 不用再額外建立 Secondary index
  - 在以 email 作登入的系統，即使不用 email 作為 PK ，還是要為 email 建立 index 吧
- 一般來說，有足夠的隨機性來避免 Contention
- 對於支援 loose index （第三課內容）的 RDBMS ， Natural key 能減少 Secondary index

Natural Key 缺點

- 一般來說，需要比較大的空間
  - 例子：用戶 email 總會比 UUID(4 bytes) 更大吧
  - 這會影響到 PK index 還有 Foreign Key 的大小
    - 以今天 SSD 正在持續暴跌的價錢來看，其實問題不太大的
- Natural Key 有機會失效
  - 某一天，老闆說：我想現在讓用戶也能用 facebook 戶口來登入系統耶 ~
  - 萬一 Natural key 失效，後果會是災難性的
- 部份 ORM 對 composite key 的支援不好，開發時很麻煩

Surrogate key 優點

- 即使老闆~~又發瘋~~改動系統邏輯， Surrogate key 幾乎都不受影響的
- Surrogate Key 的 PK Index 還有 FK 都會比較小體積

Surrogate key 缺點

- Auto increment 本身是一個潛在的 Contention 位置
- 單純地使用 Auto increment 會讓所有的 insert 集中在 B+ tree 的其中一邊
  - Insert 時造成 Contention
  - Delete 時造成 B+ tree uneven distribution
- UUID 不能保證 100 %的 uniqueness ，只是相撞可能性很很低
  - 額外的思考方向： birthday problem
    在７０人之中，有２人是同日出世的可能性是 99.9%
    在 1T 個 UUIDv4 之中，出現相撞的可能性大約是 9.4e-14
  - 額外的思考：什麼是 random ？

淺談 Primary key 和 Contention

PK 與 Contention 例子

- 以下是某隻喵的真實例子：
  - 某天笨喵在快要下班的時段~~正在暗處玩沙~~時，牠突然收到了客戶投訴系統異常緩慢，幾乎所有數據都沒法改動
  - 可是系統的 CPU / IO / Network 都正常，而且使用率很低
  - 這是關鍵系統，不能立即重新啟動伺服器
    （事後研判：重啟也不會有幫助）
  - 從 application log 發現，對航次這個 table 的改動是一個接著一個來進行，而不是並行工作
    （註：不同用戶正在改動不同的航次資料，互相沒有關聯的）
  - 因為航次資料是和大量商業邏輯有關，幾乎所有用戶都需要用上航次資料的，所以幾乎所有用戶都受到影響
  - 笨喵還沒清楚正在發生什麼事的時候，系統回復正常
  - 還好笨喵事後找出了原因並且讓問題不再重現，~~否則便被拿去人道毀滅子~~

PK 與 Contention 例子解說

- 剛才例子，事後研判是 Oracle 10g database 發生了 ITL wait 這種 Contention
  - 發生 Contention 的是在 PK index 的一個 data page
  - 因為 DBA（不是某隻喵！）設定錯誤，讓那個 data pages 剛好只剩一個 ITL slot，所以 TX 只能一個接一個地改動這個這個 page
  - 再加上 table 的 PK 設計不良，讓幾乎當天的航次的 PK 都集中到那個 data page，進而引發 Contention
- 雖然 ITL wait 只發生在 Oracle，但是 Contention 卻是所有資料庫共同面對的真正困難
  - ~~（絕對誤）CPU / IO / Network 到了極限還能推說是老闆不給錢的問題，Contention 不能抵賴啊~~

PK 與 Contention

- 一般商業系統（不考慮報表的部份），９０%的資料讀取 / 改動都是基於 PK 的
- 同一時期建立的 Record ，它們相近時間被改動 / 刪除的可能性很高
  - 所以為了避免發生 Contention ，不應該把同一時期建立的 Record 集中儲存在一起
  - 但是，為了提升 Cache hit rate ，同一時間的 Record 不應該過份分散

Monotonic Increasing PK

- Monotonic Increasing ：就是說（大致上）持續上升的數值
  - 例子： auto increment ， timestamp ，日期
- MI PK 問題：同一時期的 record ，集中在 PK index 的最右邊
- PK index 左邊的 Record 隨時間而被刪除，所以越左邊的 index leaf page 的 data 密度越低
  - 這就是 index uneven distribution 問題啦
- 小結： MI PK 會造成 PK index 右邊不停 splitting ，左邊不停的 merging

MI PK 與 heap table

- 近期新增加的 Record 的 PK 全集中在 PK index 的最右邊，引發 index page contention
  - 在 Oracle ，這問題非常致命 (呃，在 postgreSQL 也很致命的)
- 不過， heap space 的 Data 不受影響
- 因為 B+ tree index 只存放 PK ，單一 index page 能存放近千個 PK
  - 所以 MI PK 引起的 index page splitting / merging 還不算嚴重
  - 而且 page splitting / merging 只有 PK 需要移動物理位置
  - 但是 MI PK 引起的 Contention 時，受災範圍卻很大

MI PK 與 IOT

- 因為 index leaf page 存放 Record PK + data ，所以每一 index leaf page 不能存放太多的 Record
  - 所以即使發生 Contention ，受災範圍卻不太大
  - 所以會引起高頻度的 page splitting / merging
  - 在 MySQL 5.6 ， page splitting 和 page merging 都會引起 index X lock 的，祝君好運
- 因為 Record data 放在 index leaf node, page splitting 將會引發 Row data 移動物理位置

避開 MI PK 的方法

- 決定用 Surrogate Key 時，使用 UUID ，別用 auto-increment
- 如果需要使用 Natural Key ，而那 Natural Key 卻是帶有 MI 特性
  - 例子 1 ：航次管理系統中，航次的 PK 是 < 出發日期，航次編號 > ，例子： < 15AUG2015, BR827 >
  - 例子 2 ： PK 是由第三方系統提供的，像是 icq_id
- 這時候，可以使用簡單方法進行轉換：
  - 方法 1 ：把 icq_id 倒過來儲存，像 123456 的，在系統中以 654321 作為 PK
  - 方法 2 ：加上像是 MD5 這樣的 hashing ，
    像是以 < MD5(出發日期 + 航次編號) ，出發日期，航次編號 > 作為 composite key
- 建議使用 RDBMS 內建的 reverse key index ，把轉換過程由 RDBMS 自動化進行

Random PK 的迷思 1

- 迷思： Random PK 會引發大量 Random IO
- 如果是 heap table ， Random PK 引起的 Random IO 只集中在 PK Index 部份，而 PK index 體積很少的
  - Free space manager 會優先使用 dirty page 附近的 free space 來存放 new records ，讓 DiskIO 不是完全地 Random
- 為了防止斷電，所有 TX 在 Commit 的時間，會把其所做的改動抄到 Redo log ，這只是 Sequential IO
- 這些 dirty pages ，會之後由 background writer 順序地寫回硬碟
  - 延伸學習： Redo log, Undo log, checkpoint, background writer

Random PK 的迷思 2

- 迷思： Random PK 沒法進行 range scan on PK
- 事實上， OLTP 極少使用 range scan on PK ， range scan on PK 一般都是因為要產生報表才需要
  - 除非你要做 facebook-like / whatsapp-like system
- 在網上商店，客戶購買商品的 TX 是最關鍵的，慢 1 秒便流失一筆生意。給內部分析用的報表是可以慢慢等的
  - 只要告訴老闆會讓系統變慢流失生意，老闆便不會再堅持「即時性」的報表
- 報表能在 READ ONLY 的 slave database 慢慢來做，不會影響到 Master database 的

完