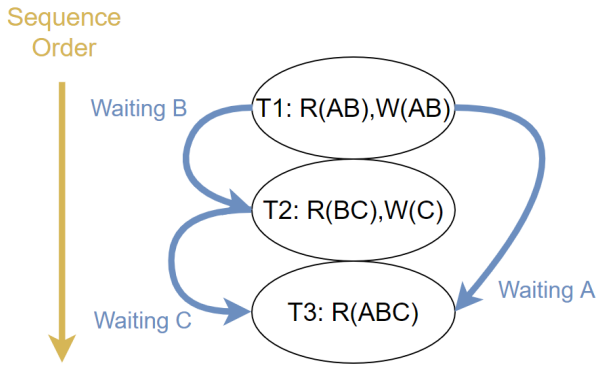
Final Project Report

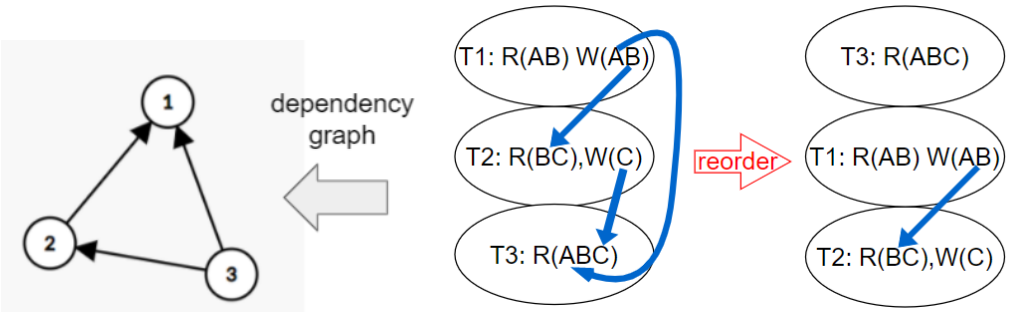
106062137 徐郁閎、106033233 周聖諺、p123786579 王麒銘

1. Background

在 5/28 的 Presentation 中，我們介紹了我們的 Motivation :「盡可能減少因前面的 Txn write record，所導致後面的 Txn read 同個 record 而等待的時間」（如下圖），因此，在希望 Minimize the pending cost 的情況下，我們想出三個 idea，分別是 Read-only First、Topology sort 與 Greedy。



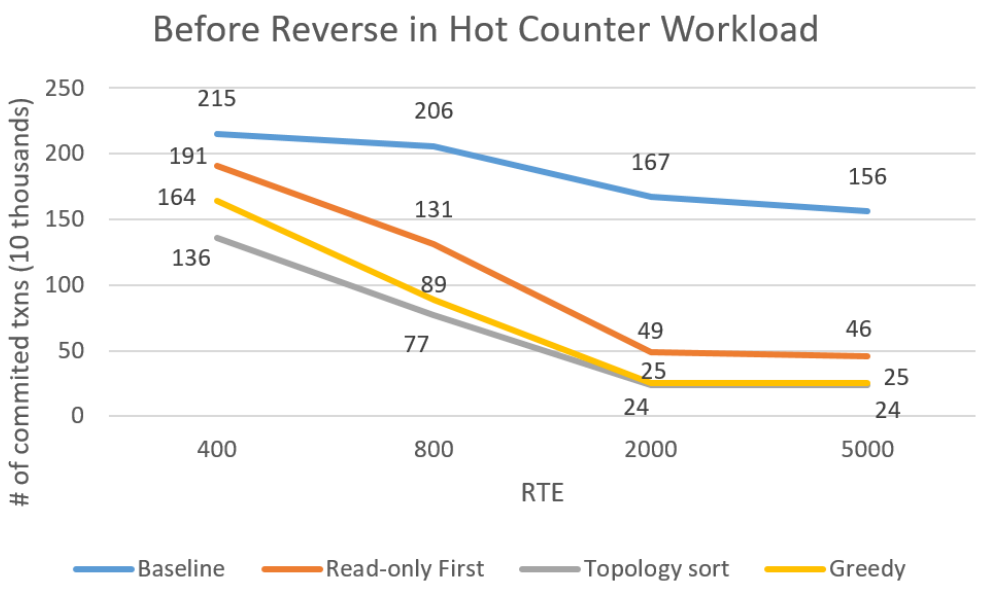
在 Read-only First 的 idea 中，我們簡單地在 hermes alogorithm 分配 partition loading 之前，將 Read-only Txns 搬到 batch 的開頭先執行，以減少 pending edge 的數量（如下右圖）。



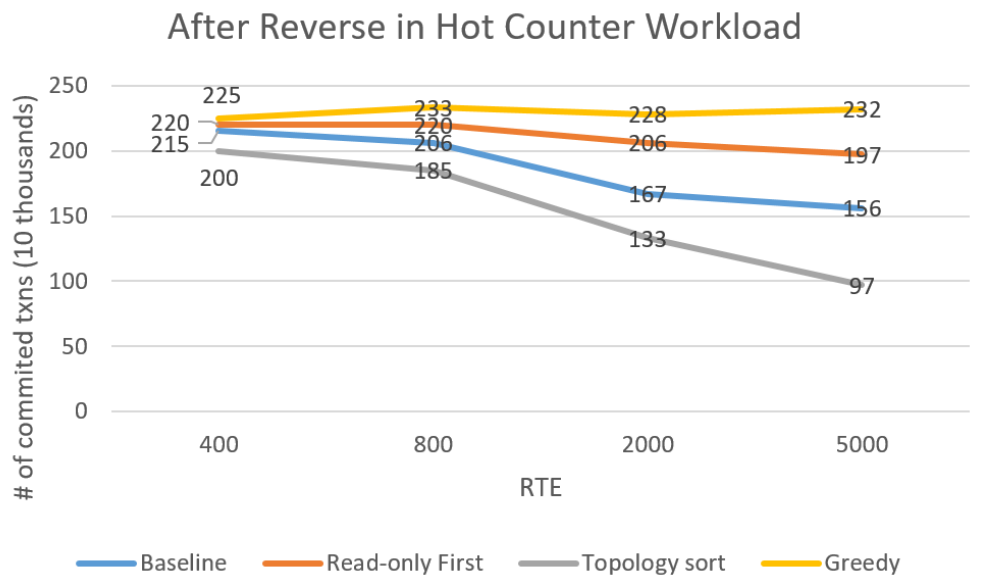
在 Topology sort 中，我們先使用 O(N^2) 的時間，依據任兩個 Txns 之間的 dependency 建立有向圖（read 指向 write，如上左圖），再使用 DFS 的方式排列 Txns 的執行順序。

在 Greedy 中，每一輪我們對於每個尚未排序的 Txns 計算當前的 pending cost，挑出最小的 cost 後加入已排序的 List，並更新當前的 DirtyMap、FutureMap 等當前資訊，直到所有 Txn 都插入已排序的 List，我們總共花費 O((N^2)/2) = O(N^2) 的時間來排序 Txns 的執行順序。

（更多實作細節，可參考後面 Implementation 章節）



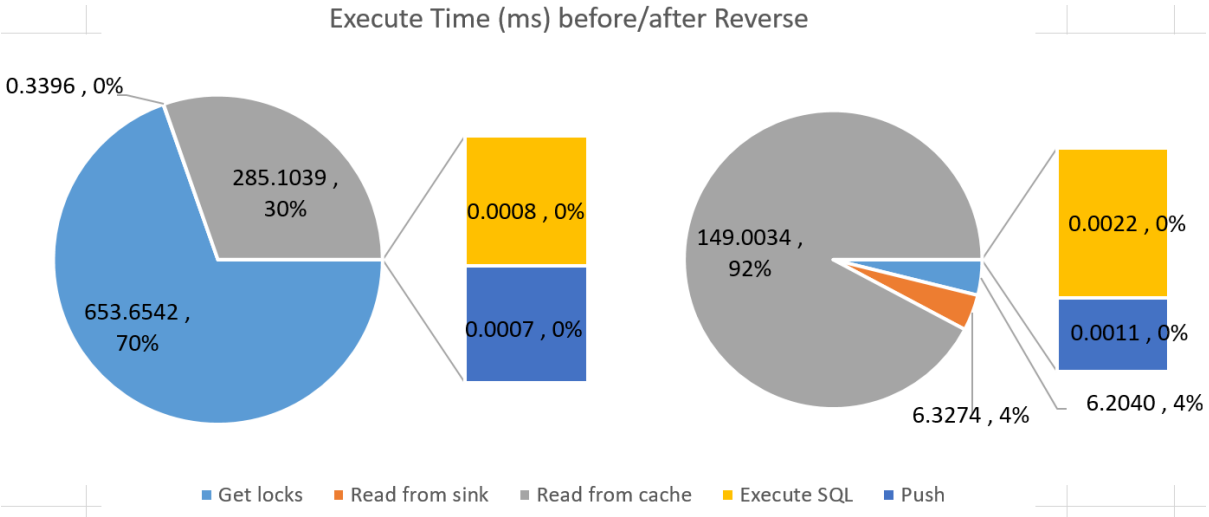
然而，在測試在 Hot Counter Workload 後，卻發現依照我們三個 idea 所 reorder 的 performance，都只剩 baseline 的 1~8 成（如上圖），反而，將我們所排出的結果「再做反排」(reverse order)，所測出來的 performance 高於 baseline 2~5 成（為原本的 reverse 前的 2~5 倍，如下圖）。



在 Presentation 中，我們針對這個反邏輯的實驗結果提出假說：「在 Tpart 會將修改過後的 record 資料儲存在 VanilliaCore 之上的 cache 的情況下，從 Tpart 的 cache 讀資料的速度可能遠快於從 VanilliaCore Buffer 讀資料，因此，越多的 Dirty read（Read-after-write）會產生越多從 Tpart 的 cache 讀資料的動作，反而使 Txn 執行速度變快」。在 Presentation 後的這幾天，我們多做了以下實驗，對該假說做更仔細的檢核。

2. Experiments

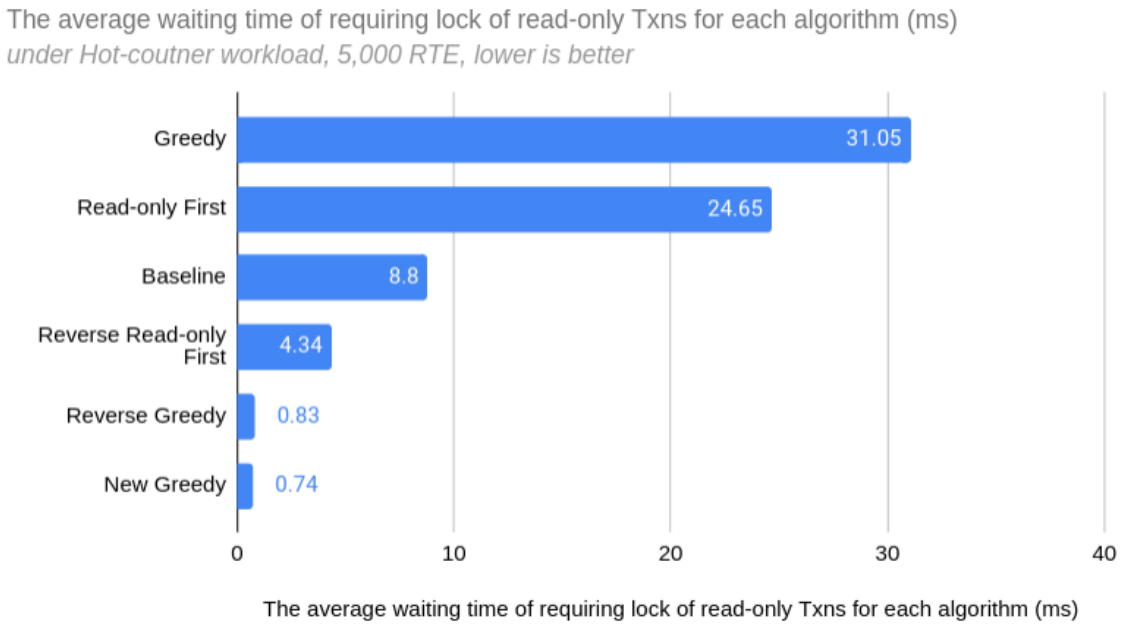
Experiment 1: Txn 執行時間分布



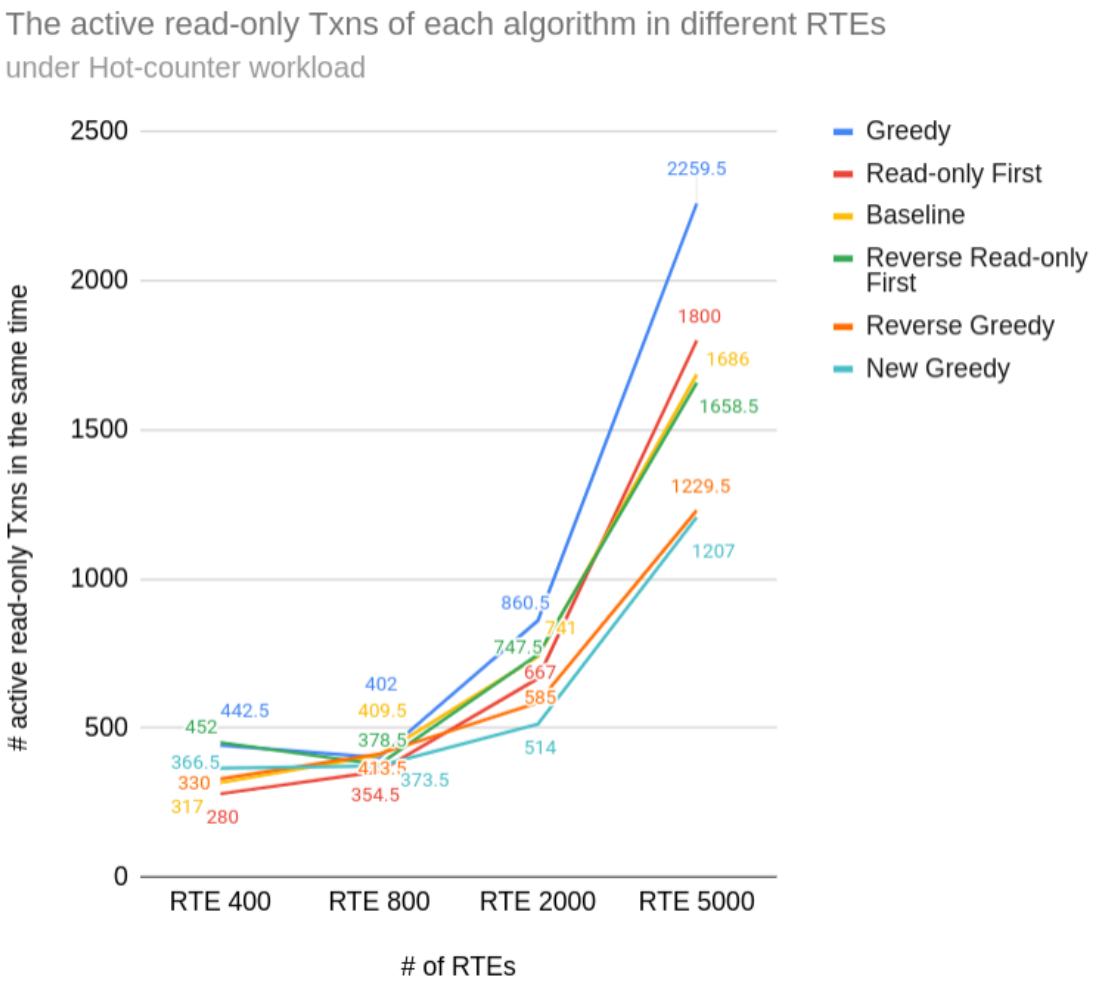
首先，我們印出 Hot Counter Workload 下 Txn 的執行功能時間（如上圖，左邊是 Greedy reverse 前，右邊是 Greedy reverse 後），可以發現 Get locks 與 Read from cache 的時間在 reverse 後大幅減少，特別是 Get locks 的時間，從 reverse 前的 653ms 縮短到 reverse 後的 6ms，因此，我們進一步針對 Get locks 做細部比較。

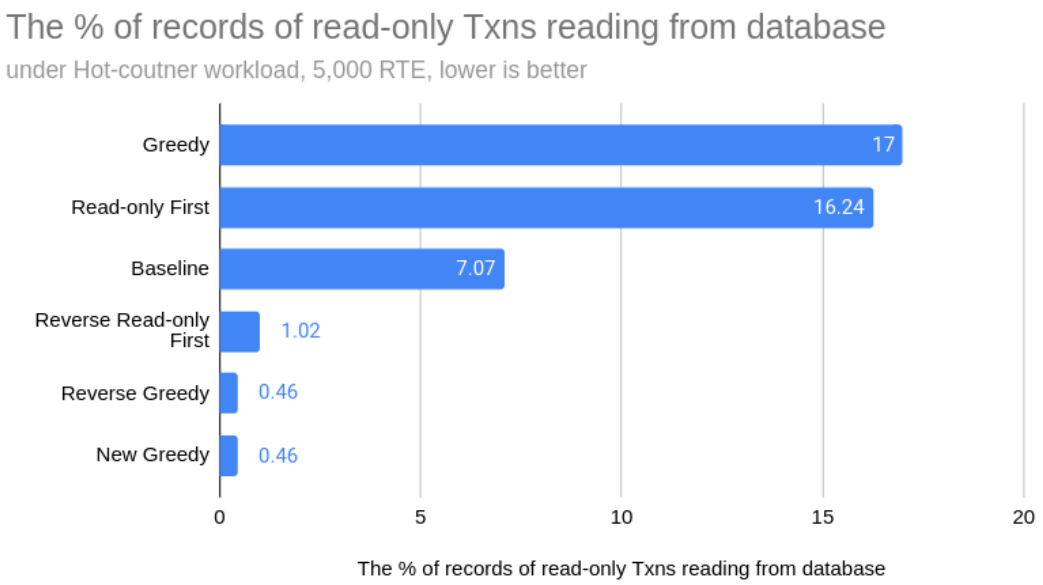
Experiment 2: 不同 RTE 間，各種演算法需要拿 conservative lock 的時間與record 的比例

下圖，我們單獨比較 Read-only Txn 拿 Locks 的花費的平均時間，由於在 Read-only reverse 和 Greedy reverse 前，演算法會將 Read-only Txns 放在最一開始執行，即在 recordCache 仍是空的狀態執行 Read-only Txns，因此，此時的 Read-only Txns 幾乎都需要拿取 Share locks 以進入 VanilliaCore 讀取資料，導致拿取 Locks 的數量與等待時間都相對 Baseline 大；對比之下，在 reverse 後，Read-only Txns 在 recordCache 已經暫存相當多資料的情況下執行，使得拿取 Locks 的數量與等待時間都相對 Baseline 小。



以上解釋了為何 Read-only reverse 和 Greedy reverse 會比 Baseline 好的原因。但實際上，可以觀察到 Greedy reverse 比 Read-only reverse 的 committed Txn 還要再多 20% 左右，原因應是 Greedy reverse 將 pending cost 最大化，導致執行 read-only Txn 時 Dirty Read 的比例大過於 Read-only reverse，而這樣的好處就是當執行 read 時，Txn 只需要去 cache 讀取 record，可以省下拿 share lock 的時間；且實際上，write record 是在 cache 新增新版本的 record，其實 racing 的問題不大，時間成本相當低，而 RW Txns 也可以同樣理由說明。從另一方面來看（下圖），在 RTE = 400 時，Read-only first 和 Greedy 還能有不錯效能，然而隨著 RTE 的上升和同一時間 active read-only Txns 的增加，就會放大 record 需要 share lock 的劣勢，使得效能大幅下降。

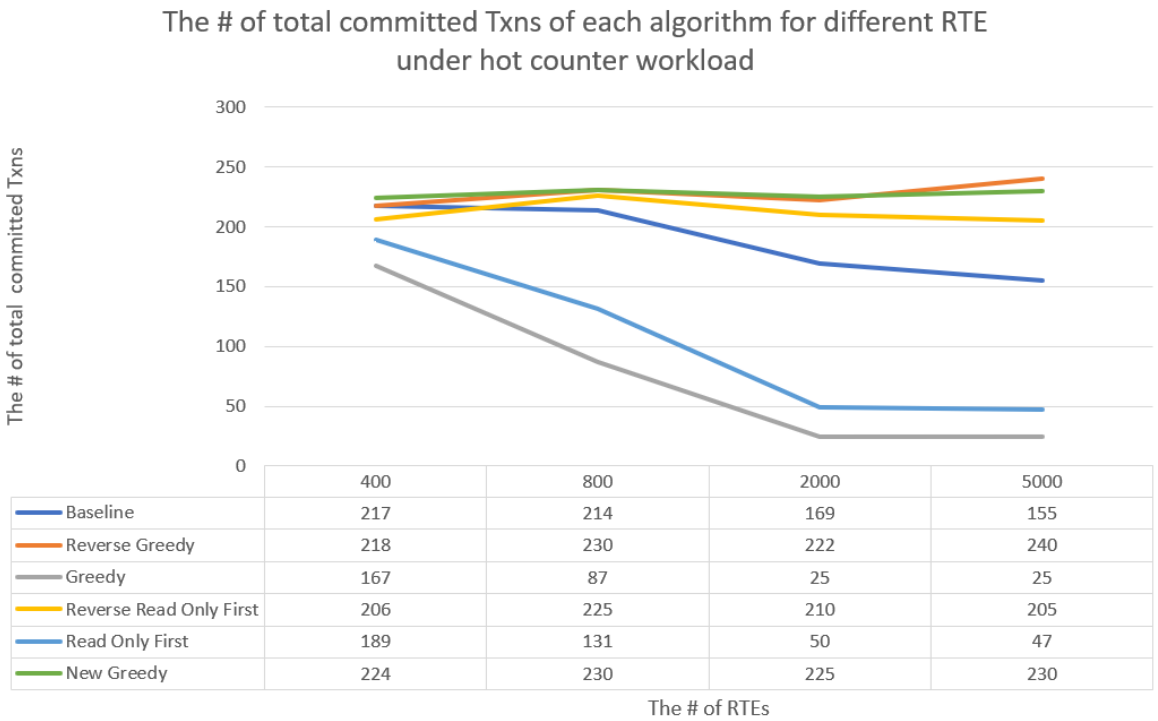




而上圖是所有 read-only Txns 所需要的 record 中，需要從 database 讀取 record 的比例（也就是在所有 read-only Txns 所需要的 record 中，需要拿 share lock 的

比例)，可以看到 Greedy、Read-Only First 需拿 lock 的比例都超過 10%，而 baseline 則約是 7.07%，Reverse Greedy、Reverse Read-Only First 及 New Greedy 的比例都只有 1% 左右或甚至更低。

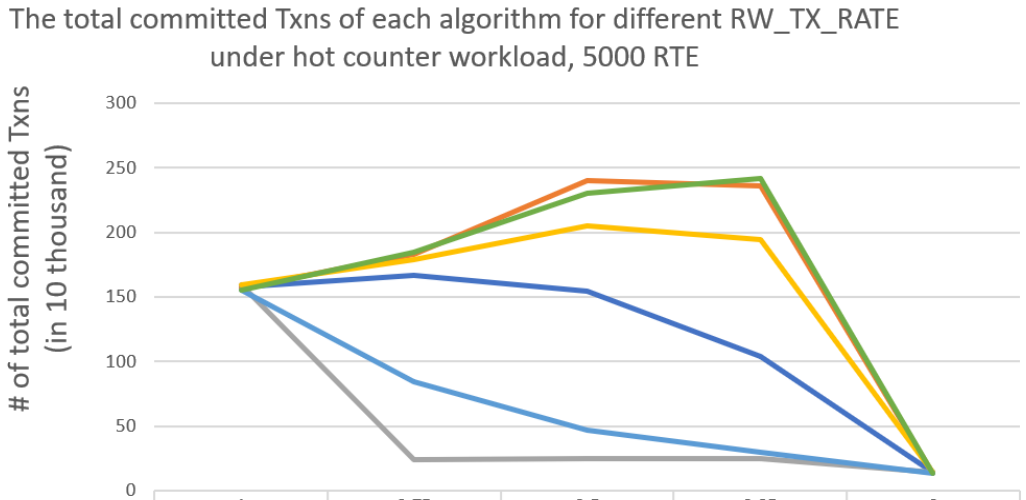
而下一張圖則顯示同一時間 active read-only Txn 的數量確實隨著 RTE 數量而增長，且在高 RTE 時，以 Greedy 與 Read-Only First racing 的情況最為嚴重，足以佐證上一段論。



	400	800	2000	5000
Baseline	217	214	169	155
Reverse Greedy	218	230	222	240
Greedy	167	87	25	25
Reverse Read Only First	206	225	210	205
Read Only First	189	131	50	47
New Greedy	224	230	225	230

綜合以上敘述，可以知道，其實決定效能的關鍵因素是 conservative lock 的時間成本，若能有效降低拿 lock 所需等待的時間，就可以大幅提高效能，而若要減少拿 lock 的時間，最好的解決辦法就是將 read / write 都在 cache 內完成， reorder的目標應變成:最大化 pending cost，依據此一目標，我們新實作了 New Greedy 演算法，每次都挑選 pending cost 最大的 Txn 並把 Read-only Txns 排到 batch 的後面，在以上四張圖表中，可以觀察到其各指標與 Reverse Greedy 相當接近，包括在不同 RTE 中的 total committed Txns。

Experiment 3: 不同 RW_TX_RATE 間，各演算法的 total committed Txns



	1	0.75	0.5	0.25	0
Baseline	158	166	155	104	14
Reverse Greedy	157	183	240	236	13
Greedy	159	24	25	25	14
Reverse Read-only First	159	179	205	194	14
Read-only First	155	84	47	30	14
New Greedy	155	185	230	241	14

RW_TX_RATE

從以上圖表中，可以看出當 Read-only Txn 的比例為 100% 時(RW_TX_RATE = 0)，total committed Txns 崩跌，符合在上一個實驗中得出的結論，Reverse greedy 和 Reverse read-only first 都是依靠將 read-only Txn 排至後面並在執行 RW Txns 時將 record 寫至 cache 以增加執行速度的，所以當沒有 RW Txns 時，就沒辦法先將 record 做 cache，導致需要拿 share lock 讀取 database 內的資料，最後使得效能大幅降低。但另一方面，當 read-only Txn 的比例為 100% 時(RW_TX_RATE = 1)，所有 reordering 的作法似乎都差不多，也說明最主要的效能提升是在 RW Txns 和 read-only Txns 混合的時候。

3. Implementation

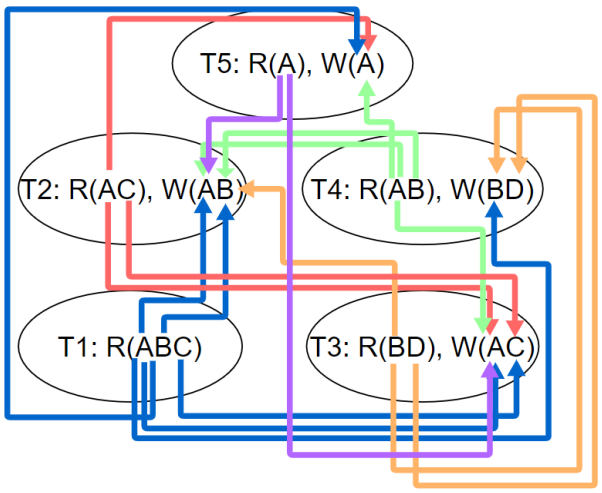
(1) Read-only First

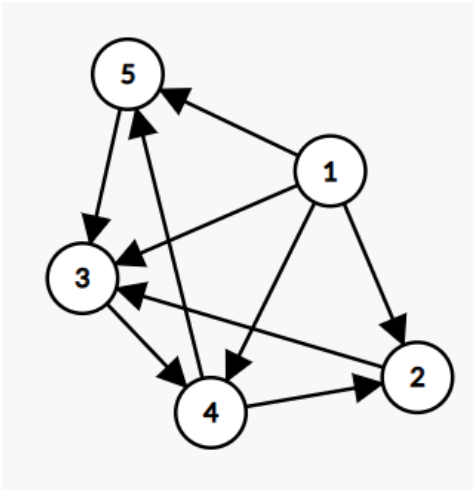
在 HermesNodeInserter.java 的 insertBatch() 中，我們在「Step 1: Insert nodes to the graph」之前，對於傳入的 tasks 檢查每個 task 是否為 Read-only，並將 Read-only 的 Txn 往前排，其餘順序照舊。(實做於 Greedy.java 中的 sortReadOnly())

(2) Topology sort

在 HermesNodeInserter.java 中我們新增了 Graph class，可分成建立 graph 與依據 graph 排序 Txn 兩個部分。首先，對於所有 tasks 呼叫 bulidGraph() 以建立

dependency graph，其中 dependency 是根據「Read 指向 Write」的數量決定，假設今天有數個 Txns 如下左圖所示：





由於 T1 指向 T2 的數量是2(AB)，而 T2 指向 T1 的數量是0，因此，T1 與 T2 之間的 dependency 是 T1 指向 T2；又好比 T3 指向 T4 的數量是2(BD)，而 T4 指向 T3 的數量是1(A)，因此，T3 與 T4 之間的 dependency 是 T3 指向 T4。值得注意的是，如果今天兩邊的數量相等，如 T2 指向 T5 與 T5 指向 T2 的數量皆為1(A)，代表兩者之間相同程度地互相 depend，因此兩個 Txn 之間便沒有 direct edge。(此部分 Txn 的關係比較，實作於 Graph class 的 compare())

在建立完 dependency graph 後，我們呼叫 Graph class 的 sort()，其實作以 DFS 遞迴呼叫 topologicalSortUtil() 的方式，將每個走過的 node 標記，並在走到無路可走時(無向外的 edge 或向外 edge 所連接的 node 都已標記)一層層return 並加入 stack 中，當所有 node 都加入 stack 時，便完成 Topology sort 的排序。

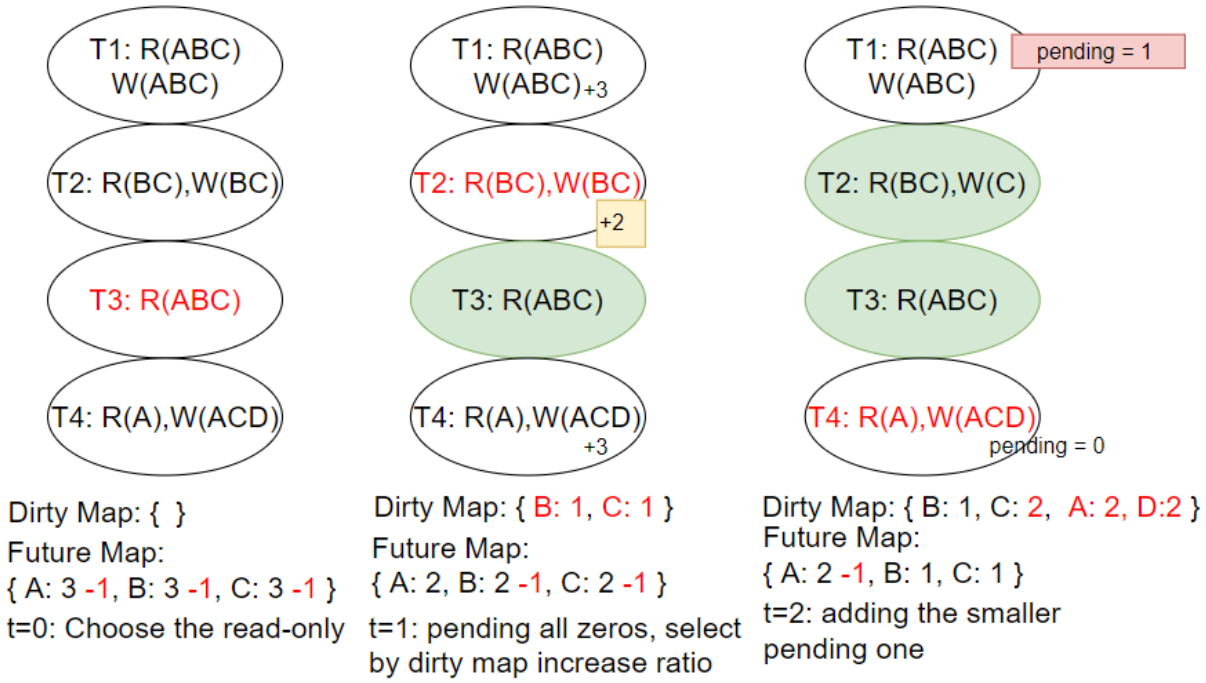
另外，為了使 Topology sort 是 deterministic，當遇到多個選擇(多個 edge 可走)時，我們統一以 Txn 原本的順序決定，例如：皆以 T1 為起始擴展點；當 T4 可同時走 T2 或 T5 時，選擇先走順序較前的 T2。

(3) Greedy

這部分實作於 Greedy class 的 sort() 中，每一輪從尚未排序的 Txn List 中挑選出 cost 最小的 Txn 加入已排序的 List，並根據新加入的 Txn 更新計算 cost 時需要的 dirtyMap 與 futureMap，直到所有 Txns 都被依序加入已排序的 List。其中，cost 的比較可分成3個階段，分別是 pendingCost、dirtySetIncrement、conflictInFuture。

首先比較的是 pendingCost，即對於自己所有的 Read record 中，上個修改該 record 距離最近的時間點，如下圖紅底紅框處的 pending=1，由於 t=2 時的 T1 需

要讀的 B、C，都在 t=1 時選到的 T2 有修改過(Dirty Map 紀錄該 record 上個修改的時間點)，因此，t=2 時 T1 的 pendingCost 為 1，而同一時間的 T4 的 pendingCost 則為 0(代表自己要讀的 record 之前還沒其他 Txn 修改過)。



再來是 dirtySetIncrement，如其變數名，是計算執行該 Txn 會造成的 Dirty Map 增長量，由於 Dirty Map 的快速增長，會使未來 Txn 選擇時更容易產生較大的 pendingCost，因此，我們應該選擇較小的 Dirty Map 增長量，如上圖黃框的 +2，代表在 t=1 時執行 T2 會造成 Dirty Map 增加兩個 entry，而這也是 T1、T2、T4 中最小的 dirtySetIncrement。

最後是 conflictInFuture，我們維護一個 futureMap 紀錄該個 batch 裡所有 read records 剩餘的 read 次數，如上圖 t=0 時，futureMap 中 A、B、C 的剩餘 read 次數，都在執行完 T3 後減 1，又好比 t=1 時，B、C 在選定排入 T2 後，由原本的 2 更新為 1。由於我們盡量要避免 write 到未來許多 Txn 都要讀得值，因此我們定 conflictInFuture 為所有 wirte record 在 futureMap 中的總和，例如，在 t=1 時，T1 的 conflictInFuture 為 6（更新前 A: 2、B: 2、C: 2）、T2 與 T4 的 conflictInFuture 都為 4。

同樣地，為使 Greedy 演算法為 deterministic，因此，當遇到 3 個 cost 依序比較下來都相同時，我們最終以 Txn 原本的順序決定該輪要插入哪個 Txn。

4. Environment

- CPU: Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz (8C16T)

- RAM: 32 GB
- Operating Systems: Ubuntu Server 20.04
    - Linux 5.8.0-55-generic x86_64
- Disk: Intel(R) SSD 665p 1 TB
- JVM: OpenJDK 1.8.0_292
- Virtualization: KVM (Full virtualization), 4 VM
    - Average response time 0.8ms between machines
    - 1 Client, 1 Sequencer, 2 Slave
    - 4 Core, RAM 6GB, Disk 32 GB
- Batch Size: 200