

# Team24 Final Project - 2

---

106034061 曾靖瑜

106062116 黃晨

106062216 馮謙

## A. Environment



## B. Experiment Method

1. Test each case 3 times.
2. Compare the variance (Eliminate outlier)
3. Choose the median (Throughput)
4. Our deviation is about 5%

## C. Improvement Strategy

1. Early Release
2. Cache
3. Delayed Updates
4. Cache Time (minor)
5. Hash Index (minor)

## Improvement of Early Release

- 優化想法：

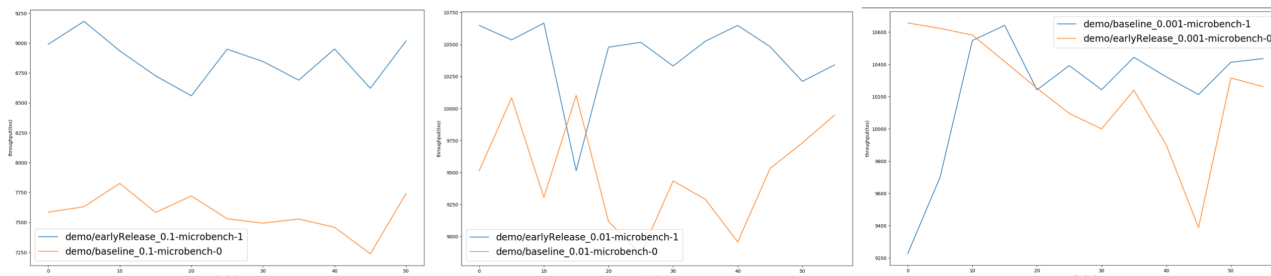
增加 Throughputs 的方法中，除了解決網路頻寬問題，另一個方法就是減少 Contention 以達到平行化。從課堂上我們學到，在傳統的 DBMS 中為了確保 Serialability，會使用 S2PL 的方式以避免 Cascading Abort 所帶來的影響；然而，在 Calvin 論文中有提到，Calvin 預設是只要一個 Tx 已經開始執行，那麼它就必須做到結束以確保 Deterministic Execution，因此在這樣的假設下，Early Release 的想法就變得可行，並能減少 Tx 等待 Lock 的時間，以達到增加 Throughputs 的目標。

- 實作：

執行這個想法其實不難，因為在拿 Lock 的時候，如果一個 Tx 對某個 Record 又需要讀又需要寫，那麼它只會拿 xLock，而如果只是單純讀的話，是不會互相 Block 的，所以我們只需要確保 xLock 的 Release。而最終我們把 Release Lock 的時間點寫在 Record 被寫完 Flush 回 Cache 後。

- 實驗結果：

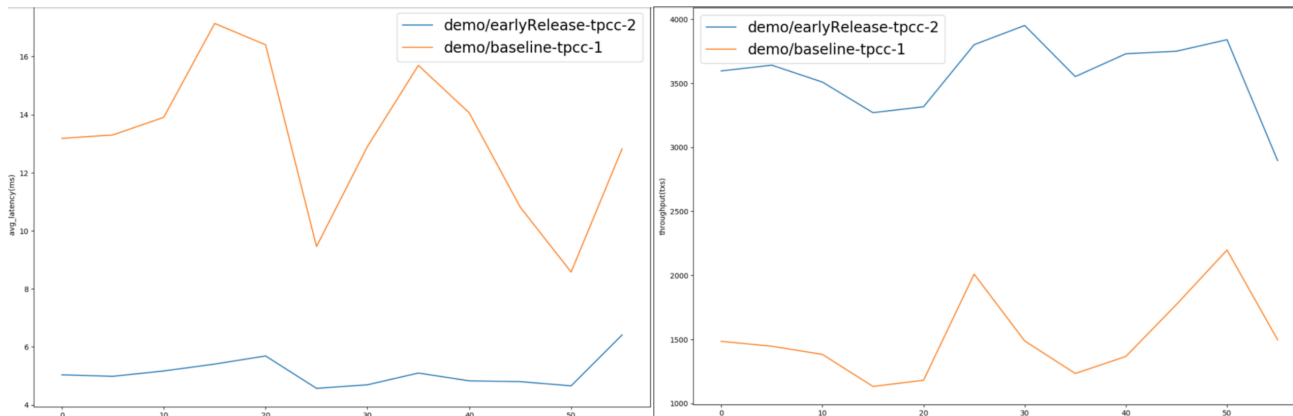
首先我們必須先證明我們對於 Lock 的改動在 Contention 高的時候是有作用的，因此我們設計了一個實驗，在 Micro Benchmark 上用不同的 Conflict Rate 來比較 Early Release 與 Baseline 之間的差別。結果如下：



Conflict Rate	EarlyRelease(commits )	Baseline(commits)	Improvement(commits)
0.1	106315	90144	106315/90144=117.9%
0.01	124907	113888	124907/113888=109.6%
0.001	122736	121826	122736/121826=107.4%

可以看到，我們在 Conflict Rate 較高時，結果明顯優於 Baseline，而 Conflict Rate 低時，與 Baseline 差別不大，因此證明我們的改動是有效的。

最後我們在 TPCC 上的做測試，實驗結果也有不錯的表現，結果如下（左圖為 Throughputs、右圖為 Latency）：





Early Release	Improvement(Commits)
Throughput	$42857/18185=235.6\%$
Latency	$6(\text{ms})/12(\text{ms}) = 50\%$

## Improvement of Cache

- 優化想法：

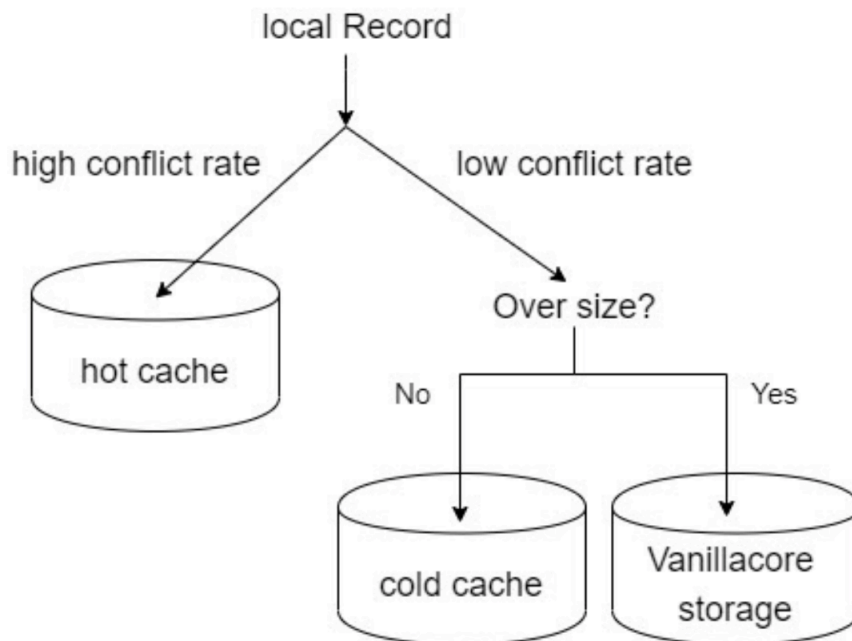
在做 Cache 時，我們必須先觀察資料的 Locality。而經過我們對 Benchmark 的分析後，我們發現，Micro Benchmark 裡可以透過調整 Conflict Rate 來增加資料被 Hit 的機率，然而 TPCC 裡就沒有明顯的趨勢。因此如果要讓 Cache 在 TPCC 上的優化有效，我們必須更進一步的分析 TPCC。

TableName	Size
warehouse 	W (property)
district 	w*10
stock、item	100000
customer、history、order	w*10*3000

從上圖的表個我們可以發現，Warehouse 與 District 這兩個 Table 是相對容易被 Hit 的，而其他的 Table 由於 Size 問題，被 Hit 到的機率極低，因此在下一步的實作上，我們會針對這兩個 Table 去進行 Cache。

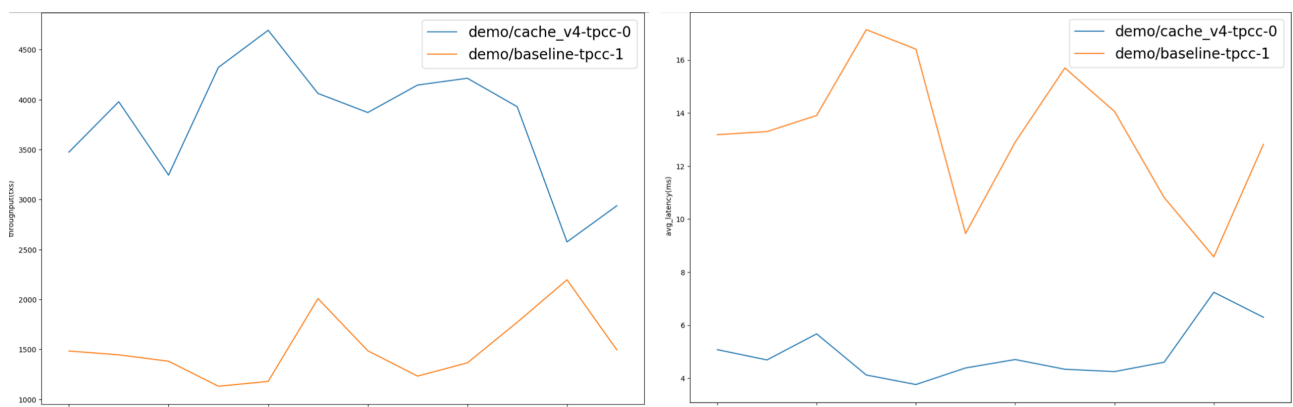
- 實作：

在 CacheMgr 中，我們新增了兩個 Object，分別為：hot cache、cold cache。Hot Cache 用來存放那些我們上述分析中所提到的會高機率被存取的 Records，而 Cold Cache 的容量是有限的，如果超過大小就會被丟到 Vanilla Core 裡，這是因為 Cold 相對而言比較難被存取到，所以並不需要花太多資源在上面。（上述這些處理都是對 Local Records 做的，因為 Remote Record 我們並不知道有沒有被改過。）



- 實驗結果：

如圖所示，挺令人訝異的。（左圖為 Throughputs、右圖為 Latency）



Cache	Improvement(Commits)
Throughput	45453/18185=249.9%
Latency	5(ms)/12(ms) = 41.6%

## Implementation of Delayed-update

- 參考：[Lazy Evaluation of Transactions in Database Systems](#)

- 原理：

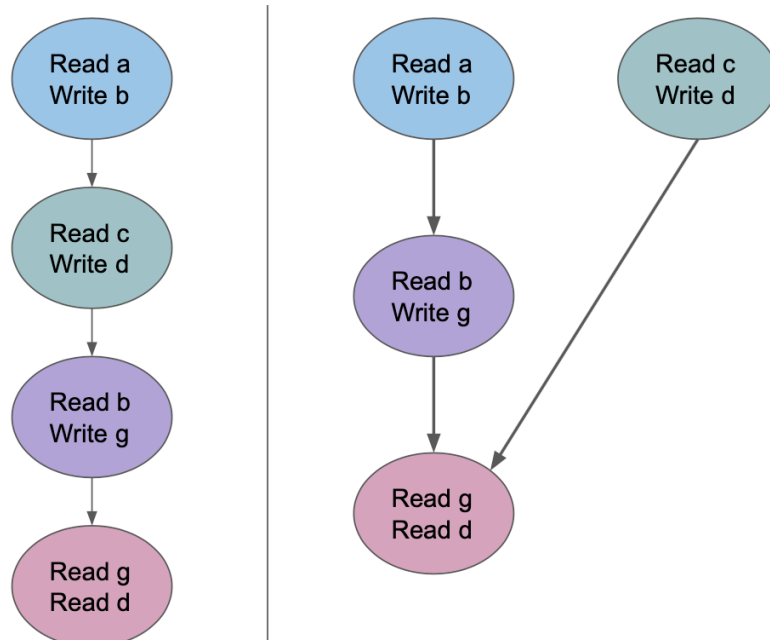
傳統 DBMS 的做法是會將收到的 Tx 從頭到尾做完後才 Commit，然而這之間有些東西其實是沒有必要那麼快做完。舉例來說，如果某個 Update 或 Insertion 跟接下來的 Tx 並沒有 Dependency，那麼我們就可以把這個改動留到之後再寫，然後先 Commit。這樣就可以更快拿到下一個 Request，進而增加 Throughputs。然而，如果 Tx 裡有許多的 Read 指令，那麼這個方法將不適用，且有可能會增加工作量，因為要花更多時間去檢查彼此之間的 Dependency。

- 實作：

在實作上，我們先建立一個 Directed Graph，用來找出 Data dependency。然後再根據這個 Graph，去找出可執行的 Vertex 來執行。每個 Vertex (tx) 在執行時，除了做自己的 Read 以外，還會有一個 WriteSet，這個 Set 裡面會有真正要執行的 Update 的 PrimaryKey，然後在 Execute Sql 時就可以知道哪些要做哪些不用。這裡有個要注意的地方是，每個 Tx 都可能與前面的 Batch 有 Dependency，那我們解決的方法是建立一個 Global 的 Map，當 Update 沒有在當下被執行時，就會和 PrimaryKey 與當下的 Record 一起被 put 進這個 Map。所以每個 tx 只要和這個 Global map 去做比較就可以得知結果了。

## Implementation

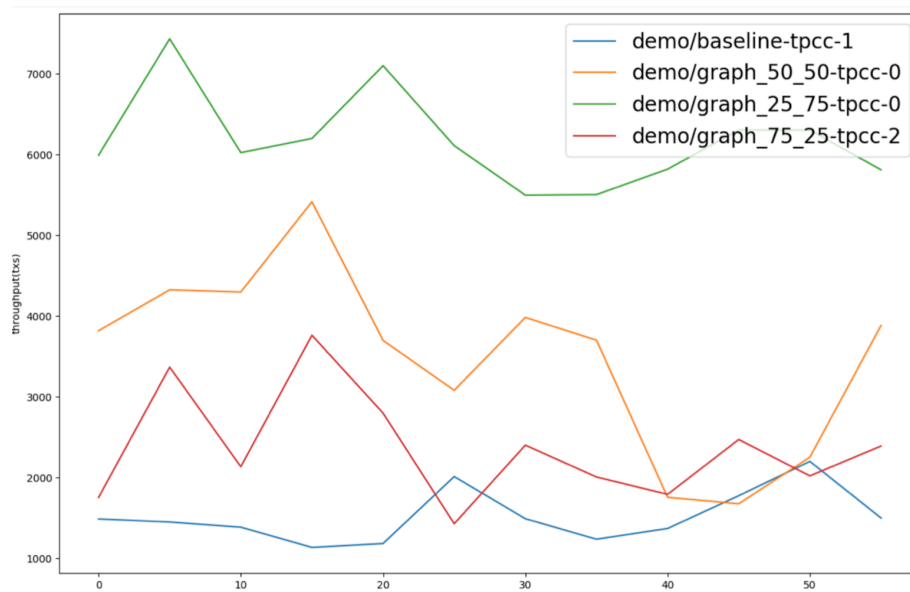
1. Build a directed graph
2. Execute the vertex whose indegree is zero
3. Remove data dependency
4. Back to 2.



- 實驗結果：

這裡我們設計了一個有趣的實驗，我們藉由調整 TPCC 中 NewOrder 與 Payment 的比例，來製造 Read 多與 Write 多的差別。由於 NewOrder 裡的 Select 較多，而 Payment 裡的 Write 較多，所以只要調整這兩個的比例就可以製造我們想要的效果。

實驗結果如圖所示：



New Order : Payment	Throughput(Commits)	Latency(ms)
75:25	28301/18185 = 155.6%	8/12 = 66.6%
50:50	41887/18185 = 230.3%	6/12 = 50%
25:75	74090/18185 = 407.4%	3/12 = 25%

可以看到當 NewOrder 比例高時，所增加的 Throughputs 較少，而 Payment 比例增加時，效果就非常好，與我們的預期相符。

## Improvement of Cache Time and Hash Index

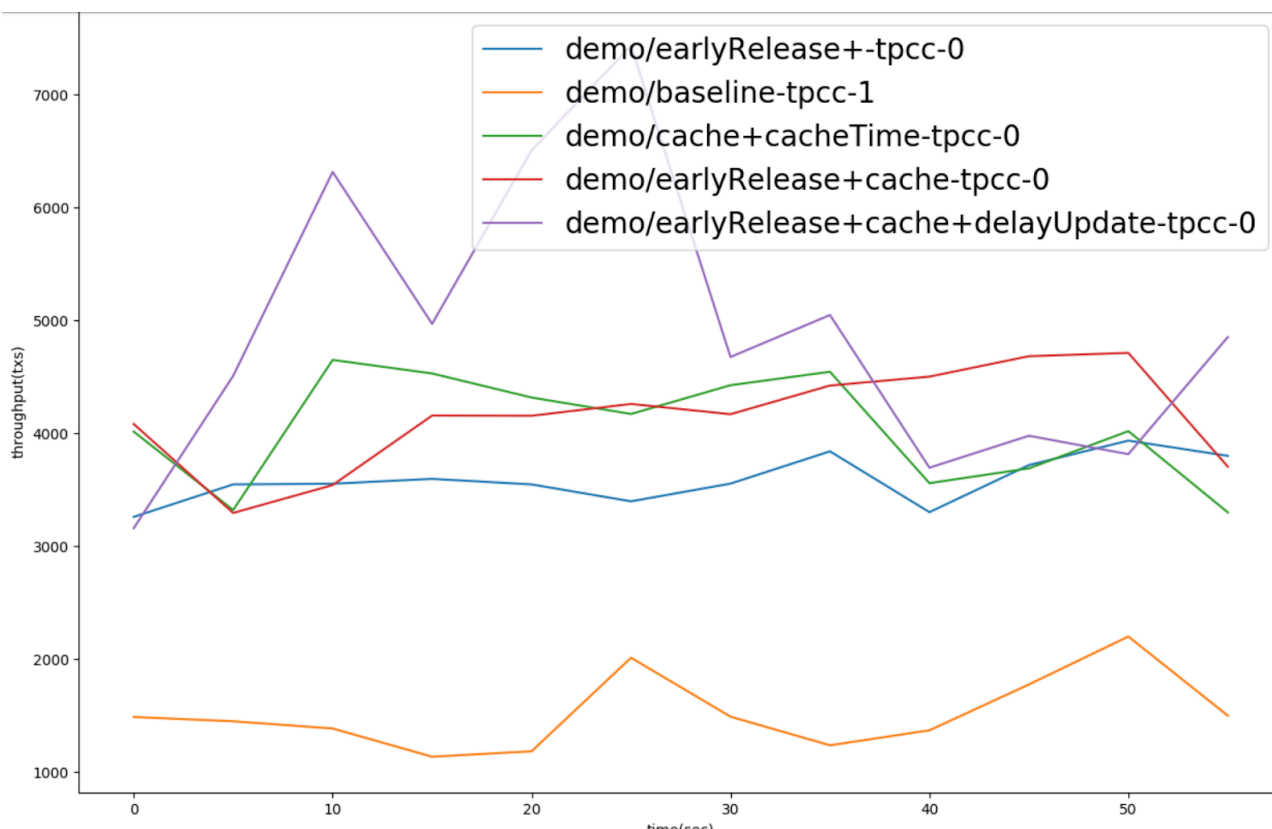
從 Cache 的 code 裡我們可以看到，原版的更新速度似乎有點慢，因此我們這裡稍微更改了一下參數。

```
// Check every 1 second (fast enough?)
HoldPackage pack = newPacks.poll(1, TimeUnit.SECONDS);
if (pack != null && !handoverToTransaction(pack.txNum, pack.reco
    pending.add(pack);
}
```

而我們發現，原本的 Hash Index 的 Bucket size 似乎可以在加大，這樣就可以更接近 O(1) 的速度。上述這兩項是比較微小的調整。

## Final Results

以下是我們最終的結果，有點類似一般 Paper 裡的 Ablation Study，至於為什麼會有這麼誇張的效果我們也沒有頭緒。



Version	Throughput(Commits)	Latency(ms)
Cache+CacheTime	$48531/18185 = 266.8\%$	$5/12 = 41.6\%$
EarlyRelease	$43029/18185 = 236.6\%$	$6/12 = 50\%$
EarlyRelease+Cache	$49665/18185 = 273.1\%$	$5/12 = 41.6\%$
EarlyRelease+Cache+Delay-Updated	$58952/18185 = 324.1\%$	$4/12 = 33.3\%$