Final Project Report

Group members

1. 107062321 王劭元
2. 107062214 陳伯瑾
3. 107062228 陳劭愷

Idea

Hermes 原本的 algorithm 是將 transactions 分成一個一個 batch 執行，在每一個 batch 中，利用 transactions 讀/寫的資料判斷，將 transaction 分派給任一個 node 執行，當所需的資料 A 不在當前的節點上時，會等待上一個修改 A 的 transaction 完成操作後，將 A 透過網路傳遞到執行的節點執行。最後完成一個 batch 的 transactions 後，將所有修改過的 records 寫回原本資料所在節點。

當測試的資料有 Hot Count 的性質，也就是每次操作中都會有少數的 hot record 被選取到，這時 Hermes 的算法會導致 hot record 不斷的需要被搬動，而造成效率的低落。

因此我們將原本的算法改寫，透過 data 的 replication 以及 reordering 來降低資料的移動量。

Replication 的部分，我們透過蒐集每一個 transaction 讀寫的資料，統計每個 Primary key(PK) 出現的次數以及總 transactions 數量，因為 hot record 會在每一筆 transaction 中出現，因此當每一個 PK 的出現比率大於我們設定的一個參數 `IS_HOT_RECORD_THRESHOLD` 時，我們會判定此 PK 為 hot record，在一個 batch 執行的最一開始即對所有的 hot records 做 replication 的動作，將它們複製到每一個節點。

在每個 batch 的最初做 replication 後，transaction 中的 hot record 可以在各個節點獲得，但若有修改 hot record 的 transaction 出現，這個 transaction 所做出的修改必須要讓其他後面要讀此 hot record 的節點也做出修改，可以採用複製此 transaction 的方式來解決此問題，但是複製 transaction 也會導致效率降低。因此我們在 v2 與 v3 的版本中使用了此方法，在 v4 的版本中，使用的則是 reordering 的方法。

Version 4 版本中，為了解決修改 hot record 的 transaction 必須被複製到所有讀取此 hot record 的節點的問題，透過將一個 batch 內的 transaction 排序，我們能夠完全的避免掉這個問題的發生。找出有修改操作的 hot record，透過原本的方式將其執行在 replication 之前，即可保證 replication 後所有的 hot record 都不會再被修改。

Implementation

因為每一個 batch 都需要一個 Replication 的 Task，因此我們在 `TPartPartitioner` 的 `processBatch` 中新增一個特別的 stored procedure call，並且轉換成 task 送到 `HermesNodeInserter` 的 `insertBatch` 中。

```
private void processBatch(List<TPartStoredProcedureTask> batchedTasks) {
  StoredProcedureCall call = new StoredProcedureCall(-1, -1, SP_DOING_REPLICATION, new Object[] {});
  TPartStoredProcedureTask task = createStoredProcedureTask(call);

  // Insert the batch of tasks
  inserter.insertBatch(graph, batchedTasks, task);
```

操作 replication 的 stored procedure call 是新增在 `TPartYscbStoredProcFactory` 中，並且在 `TPartStoredProcedureCall` 中新增了一個函數 `isDoingReplication` 來判斷一個 stored procedure call 是否為我們定義的副本操作。

```
private static final int SP_DOING_REPLICATION = -1234;

@Override
public TPartStoredProcedure<?> getStoredProcedure(int pid, long txNum) {
  TPartStoredProcedure<?> sp;
  if (pid == SP_DOING_REPLICATION) {
    sp = new TpartYcsbProc(txNum, true);
    return sp;
  }
```

進入 `HermesNodeInserter` 的 `insertBatch` 後，分為以下幾個步驟：

1. 統計 transactions 中的所有 PK 數量以及總 transactions 數量，作為判斷一筆資料是否為 hot record 的依據。我們使用的數據結構為 `HashMap<PrimaryKey, MutableInteger> readWriteCount`，代表一個 PK 總共出現的次數。`MutableInteger` 為一個 private 的 class 包含一個 integer value，提供 `increment` 與 `get` 的兩種操作，使用 `MutableInteger` 的原因為能夠更快的達到我們想要的計數效果[1]。

   另外，當總 transactions 每超過一定數量，會進行一次重新計算 hot records 有哪些的過程。而我們選擇的大小是1000，這個是有做過實驗才決定的，相關結果在底下會介紹[2]。

```
// Step 1: calculate read write count, update replicaTask if needed
for (TPartStoredProcedureTask task : tasks) {
  totalNumberOfTxs++;
  for (PrimaryKey key : task.getReadSet()) {
    MutableInteger count = readWriteCount.get(key);
    if (count == null) count = new MutableInteger();
    else count.increment();
  }
  for (PrimaryKey key : task.getWriteSet()) {
    MutableInteger count = readWriteCount.get(key);
    if (count == null) count = new MutableInteger();
    else count.increment();
  }
  if (totalNumberOfTxs % DO_REPLICATION_TXS_SIZE == 0) recalculateHotRecordKeys();
}
```

2. （不一定每次執行）重新計算 hot records 有哪些。將上述的 `readWritCount` 中記錄到的 key 取出，若其出現的比例 ≥ `IS_HOT_RECORD_THRESHOLD`，則將其加入 `HashSet<PrimaryKey> hotRecordKeys` 中，數據結構 `hotRecordKeys` 會在接下來的步驟幫助我們判斷哪些 key 是需要/已經被副本到每一個節點的。

```
private void recalculateHotRecordKeys() {
  hotRecordKeys.clear();
  for (PrimaryKey key : readWriteCount.keySet()) {
    if (isHotRecord(key)) {
      hotRecordKeys.add(key);
    }
  }
}
```

```
private boolean isHotRecord(PrimaryKey key) {
  if (totalNumberOfTxs <= 100) {
    return false;
  }
  if (!readWriteCount.containsKey(key)) {
    return false;
  }
  return (readWriteCount.get(key).get() / totalNumberOfTxs) >= IS_HOT_RECORD_THRESHOLD;
}
```

3. 將所有修改到 hot record 的 transactions 找出，並利用原 Hermes 的 `insertAccordingRemoteEdges` 函數將其安排在機器上，並插入到 `TGraph` 中，同時紀錄剩下來的 tasks 有哪些。

```java
// Step 2
ArrayList<TPartStoredProcedureTask> restOfTasks = new ArrayList<TPartStoredProcedureTask>();
for (TPartStoredProcedureTask task : tasks) {
  boolean containWriteHotRecord = false;
  for (PrimaryKey key : task.getWriteSet()) {
    if (hotRecordKeys.contains(key)) {
      containWriteHotRecord = true;
      break;
    }
  }
  if (containWriteHotRecord) insertAccordingRemoteEdges(graph, task);
  else restOfTasks.add(task);
}
```

4. 接著將所有的 hot record 進行複製的操作。對於 replication 的 task，將 task 插入到 graph 時，我們會複製這個 task，使其跑在每一個節點上。因為在複製之前，資源可能會有改變，因此從 `getResourcePosition` 中得到最後修改位置，加上 read edge 與 write edge 使得節點可以從最後修改位置獲得到資料。

```java
// Step 3: Insert replica node into graph
insertReplicationNodeAndEdges(graph, replicaTask);
```

```java
private void insertReplicationNodeAndEdges(TGraph graph, TPartStoredProcedureTask task) {
  for (int partId = 0; partId < partMgr.getCurrentNumOfParts(); partId++) {
    graph.insertReplicationNode(task, hotRecordKeys, partId);
  }
}
```

```java
public void insertReplicationNode(TPartStoredProcedureTask task, HashSet<PrimaryKey> replicatedKeys, int assignedPartId) {
  // Assign to current server, since everyone is responsible
  // for doing replication work
  TxNode node = new TxNode(task, assignedPartId, false);
  txNodes.add(node);

  for (PrimaryKey res : replicatedKeys) {
    Node targetNode = getResourcePosition(res);
    node.addReadEdges(new Edge(targetNode, res));
    targetNode.addWriteEdges(new Edge(node, res));
  }
}
```

5. 將剩餘的 transactions 依照順序插入到圖中。

```
// Step 4: Insert nodes to the graph
for (TPartStoredProcedureTask task : restOfTasks) {
  insertAccordingRemoteEdges(graph, task);
}
```

```
private void insertReplicationNodeAndEdges(TGraph graph, TPartStoredProcedureTask task) {
  for (int partId = 0; partId < partMgr.getCurrentNumOfParts(); partId++) {
    graph.insertReplicationNode(task, hotRecordKeys, partId);
  }
}
```

6. Step 5 & 6 與原本相同，進行負載平衡的動作。這邊可以注意一點，為了讓 transactions 的 task 不被 移動，我們在 `TxNode` 中加入一個參數 `allowReroute`，提供一個 task 是否可以被移動到其他節點的資訊，所有 replication 用的 task 所產生的 graph 節點 `allowReroute` 都是 `false`。最後在 `findTxNodesOnOverloadedParts` 時，只有 `allowReroute` 的 `TxNode` 會被選到。

```
// Step 5: Find overloaded machines
overloadedThreshold = (int) Math.ceil(
    ((double) tasks.size() / partMgr.getCurrentNumOfParts()) * (IMBALANCED_TOLERANCE + 1));
if (overloadedThreshold < 1) {
  overloadedThreshold = 1;
}
List<TxNode> candidateTxNodes = findTxNodesOnOverloadedParts(graph, tasks.size());
```

```
// Step 6: Move tx nodes from overloaded machines to underloaded machines
int increaseTolerence = 1;
while (!overloadedParts.isEmpty()) {
//      System.out.println(String.format("Overloaded machines: %s, loads: %s, increaseTolerence: %d", overlo
  candidateTxNodes = rerouteTxNodesToUnderloadedParts(candidateTxNodes, increaseTolerence);
  increaseTolerence++;

  if (increaseTolerence > 100)
    throw new RuntimeException("Something wrong");
}
```

```java
private List<TxNode> findTxNodesOnOverloadedParts(TGraph graph, int batchSize) {
  // ...

  // Find out the tx nodes on these parts
  List<TxNode> nodesOnOverloadedParts = new ArrayList<TxNode>();
  for (TxNode node : graph.getTxNodes()) { // this should be in the order of tx number
    if (node.getAllowReroute()) {
      int homePartId = node.getPartId();
      if (overloadedParts.contains(homePartId)) {
        nodesOnOverloadedParts.add(node);
      }
    }
  }
```

Implementation Before Version 4

在實作到 version 4 的 reordering 之前，我們有先實作出沒有 reordering 的版本，因為不是最終版本，因此這裡只是簡單的介紹我們的 Implementation 方式。

Version 2 與 3 我們實作的是在每個 batch 的最一開始進行 replication 的動作，為了解決 hot record 被修改後，其他 transactions 需要知道此修改的問題，我們會將修改 hot record 的 transactions 複製到其他的節點上運行。

而 Version 2 與 3 的差別在於，Version 2 會將修改操作**複製到每一個節點上**，而 Version 3 只會將修改操作**複製到有讀寫同一個 hot record** 的節點上。雖然因為預設做 benchmark 的機器數量為 2 台 server，所以複製到每一個節點應該與複製到所有節點的花費是一樣的[3]，但是如果有更多 server 的話，Version 3 的版本應該能跑的更好。

以下簡介 Version 3 的 Implementation，要實作 Version 3 的複製 transaction 操作，我們必須做到兩點：一、找到需要複製的 transaction 以及需要複製到的節點。二、將 transaction 複製到另一個節點執行，也就是插入到 `TGraph` 中，與原本的 task 相同的位置（複製出來的 task 要插入到與原 task 相同的位置才能使得執行結果正確）。

1. 同上的 Step 1、3 ~ 6，再加上一個 Step 7 是用來找出需要複製的 transaction 以及需要複製到的節點。我們使用資料結構 `HashMap<PrimaryKey, HashSet<Integer>> hasRead`，從最後一個 task 到最新的 task，紀錄 PK 在哪些機器（partId）上有讀取的操作。有了 `hasRead` 的資料後，每次遇到有修改到 hot record 的 `TxNode` 時，如果他有在其他的機器上面做讀的動作的話（也就是 `hasRead.get(key)` 中所包含的 partId 們），我們就複製這個 `TxNode`，先將其加入到 `HashSet<Integer> partIds` 中。這裡我們並沒有直接的呼叫 `graph.copyTxNode`，因為直接複製會導致 `txNodes` 的數量改變，可能會連帶的影響到 `findShouldReplicaTxs` 的運作。對於每個 `TxNode`，所有要被複製的 `partIds` 們，會被加到一個 `HashMap<Integer, HashSet<Integer>> copyTxNodes` 中，其中 `HashMap` 的 key 代表要複製的 `TxNode` 的索引值，可以用來幫助我們定位接下來要在哪一個位置插入複製出來的 `TxNode`。最後跑過 `copyTxNodes` 中的所有 key-value pair，呼叫 `graph.copyTxNode` 幫助我們複製新的 `TxNode`。

```java
// Step 7: replica txs
findShouldReplicaTxs(graph)
```

```java
private void findShouldReplicaTxs(TGraph graph) {
  HashMap<PrimaryKey, HashSet<Integer>> hasRead = new HashMap<PrimaryKey, HashSet<Integer>>();
// key: pk, value: set of partIds => key pk has been read by some partIds

  List<TxNode> txNodes = graph.getTxNodes();
  HashMap<Integer, HashSet<Integer>> copyTxNodes = new HashMap<Integer, HashSet<Integer>>();

  for (int i = txNodes.size() - 1; i >= 0; i--) {
    TxNode node = txNodes.get(i);
    if (node.getTask().getProcedure().isDoingReplication()) continue;

    copyTxNodes.put(i, new HashSet<Integer>());
    HashSet<Integer> partIds = copyTxNodes.get(i);
    for (PrimaryKey key : node.getTask().getWriteSet()) {
      if (partMgr.isFullyReplicated(key) && hasRead.containsKey(key)) {
        for (Integer partId : hasRead.get(key)) {
          if (node.getPartId() != partId) {
            partIds.add(partId);
          }
        }
      }
    }

    for (PrimaryKey key : node.getTask().getReadSet()) {
      if (partMgr.isFullyReplicated(key)) {
        if (!hasRead.containsKey(key)) {
          hasRead.put(key, new HashSet<Integer>());
        }
        hasRead.get(key).add(node.getPartId());
      }
    }
  }

  for (Map.Entry<Integer, HashSet<Integer>> entry : copyTxNodes.entrySet()) {
    Integer txNodeId = entry.getKey();
    for (Integer partId : entry.getValue()) {
      graph.copyTxNode(txNodeId, partId);
    }
  }
}
```

2. 我們觀察原本的 `insertTxNode`，發現為了要能做到 `graph.copyTxNode`，必須可以知道以下兩件事情：一、被複製的 `TxNode` 的 readEdges 有哪些，因為新複製出來的同樣也需要這些 readEdges 來獲取最新資料。二、有哪些 writeEdges 連到這個被複製的 `TxNode` 上，因為上一個修改資料的 `TxNode` 必須也將資料寫到這個新複製出來的 `TxNode`。這兩點我們都可以從 `TxNode` 上的 readEdges 得知（writeEdges 是 readEdges 的相反）。

   因此，在 `copyTxNode` 中，我們先 new 出一個新的 `TxNode`，並將此 `TxNode` 插入到原本的位置之前（之後也可以），再將 readEdges 與對應的 writeEdges 都加上即完成。

```java
public void copyTxNode(int templateTxNodeIndex, int partId) {
  TxNode templateNode = txNodes.get(templateTxNodeIndex);
  TPartStoredProcedureTask task = templateNode.getTask();
  TxNode node = new TxNode(task, partId, false);
  txNodes.add(templateTxNodeIndex, node);

  for (Edge e : templateNode.getReadEdges()) {
    node.addReadEdges(new Edge(e.getTarget(), e.getResourceKey()));
  }
  for (Edge e : templateNode.getReadEdges()) {
    e.getTarget().addWriteEdges(new Edge(node, e.getResourceKey()));
  }
}
```

到此 Version 3 的 Implementation 完成。

Experiment Results on Single machine

Notes

我們一共有三台機器，環境分別為

1. Machine A: 2 GHz Quad-Core Intel Core i5, 16GB 3733MHz LPDDR3, 512GB SSD, macOS
2. Machine B: 2.3 GHz Dual-Core Intel Core i5, 16 GB 2133 MHz LPDDR3, 256GB SSD, macOS Big Sur 11.4
3. Machine C: Intel i5-8365 @ 2.4GHz, 16GB RAM, 512GB SSD, macOS Big Sur 11.4

每次實驗我們只會調整部分參數，其餘參數部分皆與下面相同：

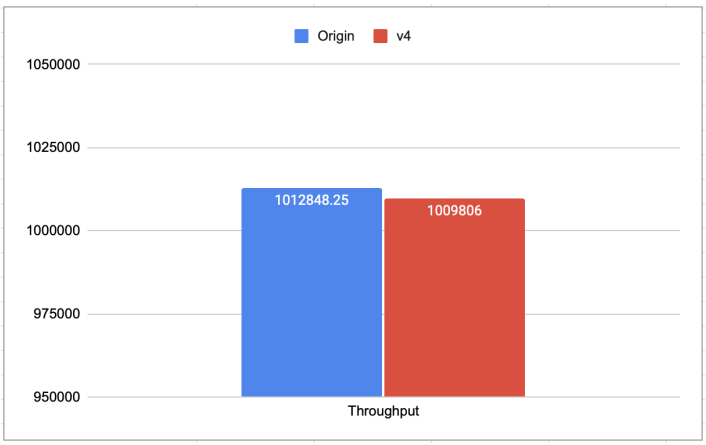
workload若沒有特別說明都是在HotCounter下

```
org.vanilladb.bench.BenchmarkerParameters.BENCHMARK_INTERVAL=180000
org.vanilladb.bench.BenchmarkerParameters.NUM_RTES=200
org.vanilladb.bench.StatisticMgr.GRANULARITY=10000
org.elasql.schedule.tpart.hermes.FusionTable.EXPECTED_MAX_SIZE=1000000
org.elasql.schedule.tpart.hermes.HermesNodeInserter.IMBALANCED_TOLERANCE=0.1
org.elasql.schedule.tpart.TPartPartitioner.ROUTING_BATCH=100
org.elasql.remote.groupcomm.client.BatchSpcSender.BATCH_SIZE=20
org.elasql.bench.benchmarks.ycsb.ElasqlYcsbConstants.DATABASE_MODE=1
org.elasql.bench.benchmarks.ycsb.ElasqlYcsbConstants.TX_RECORD_COUNT=2org.elasql.bench.benchmarks.ycsb.ElasqlYcsbConstants.TX_RECORD_COUNT=
org.elasql.bench.benchmarks.ycsb.ElasqlYcsbConstants.HOT_COUNT_PER_PART=1
org.elasql.bench.benchmarks.ycsb.ElasqlYcsbConstants.HOT_UPDATE_RATE_IN_RW_TX=0.1
```

Machine A

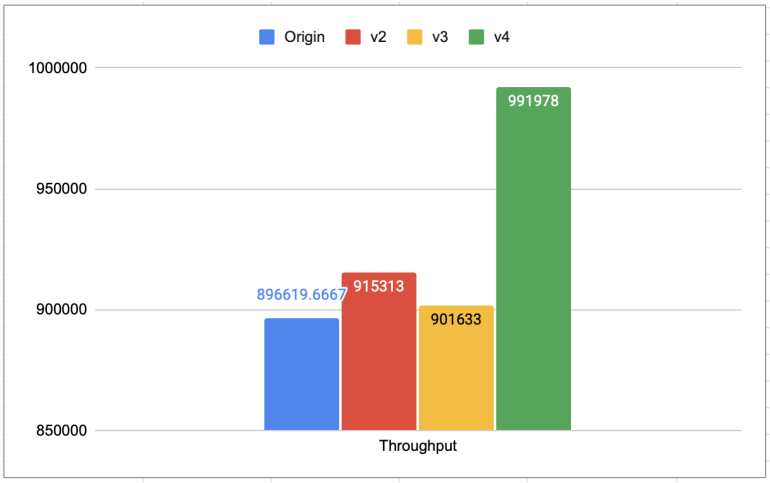
Default parameter, Workload=HotSpot

助教有說過可以跑Hotspot的workload來驗證自己有沒有寫壞，因為Hermes在Hotspot下跑得很好，很難優化。以下我們就跑了五次實驗，去除掉一次不太穩定的數值後的結果



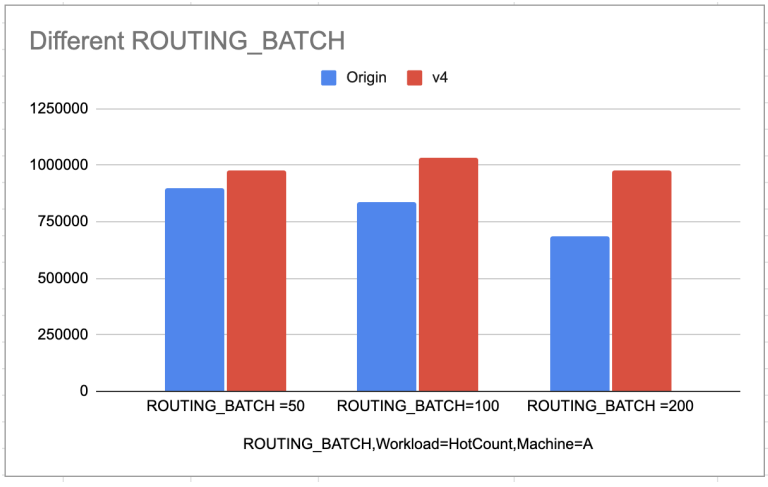
1. 可以看出來我們沒有把Hermes改壞掉!

workload=HotCounter



1. 可以看出我們一共做了三次實驗後，平均優化後的版本跑在單點node上有進步，並且無論哪個版本都有稍微贏過origin版本。

ROUTING_BATCH

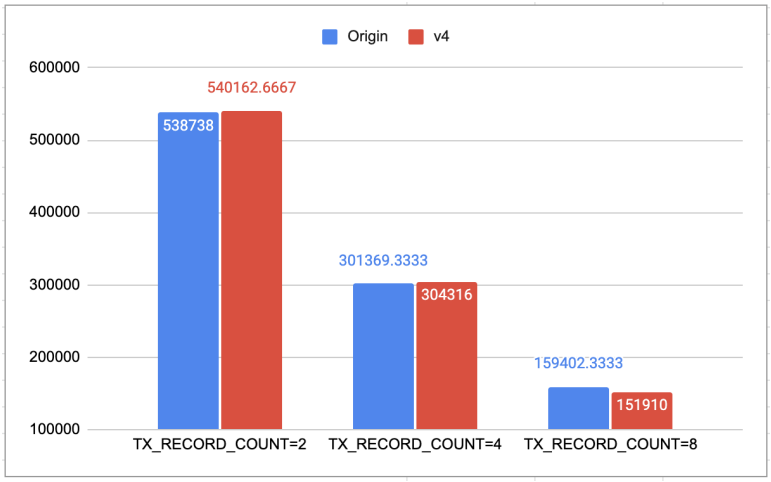


1. 我們可以看到無論在v4還是origin版本，當今天Routing batch很接近，#RTE*clientNum時，throughput都會下降

Machine B

Different TX_RECORD_CONT

我們這邊調整一個 transaction 會用到幾個 record，我們預期是不會有什麼影響

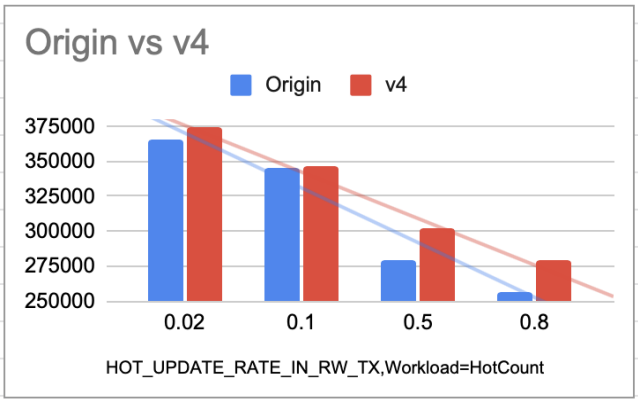


1. 雖然在 Record_Count=8的時候，我們的v4版本的 throughput 有一點明顯地比 origin 少，但若去看每次的實驗數據我們有發現，在這三次中，我們 v4 的版本確實有不穩的變化，所以我們認為這個 TX_RECORD_COUNT 跟我們這次的 throughput 不會有太大關係!

Machine C

Different HOT_UPDATE_RATE_IN_RW_TX

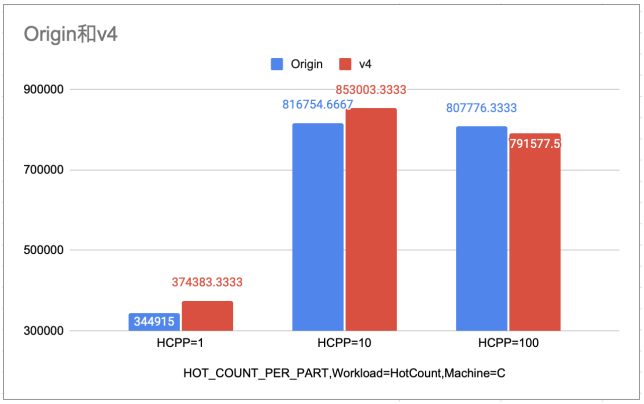
我們有嘗試調整這個rate來改變hot record被更改的機會，然後除了跑v4的版本，也有跑origin的版本，想看看我們更改後的版本能不能在不同參數下有更好或更不好的表現，以下是實驗結果，為我們跑了三次平均的結果。



1. 我們可以看到當Hot record要一直被更改的話，很直觀的當然他的throughput會下降。
2. 但這邊比較特別的是，可以看出，upadate rate越高，我們修改後的v4跑的相對於origin也是更好，並且經過計算，相對下降的比例，origin掉了29.9%，而我們的v4只掉了0.256%，也可以從趨勢線的斜率看出。推測因為我們有做reorder，所以可以handle即使有很多人要改hot record的情況

Different HOT_COUNT_PER_PART

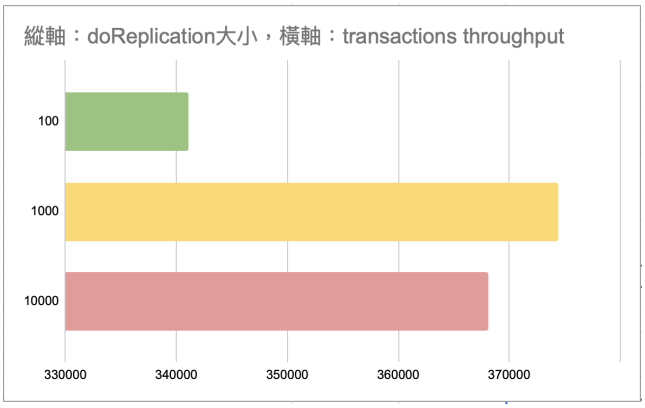
我們有試著調整每個partition的hotcount來看看實驗結果，也有跑在Origin和v4不同版本上，來看看有沒有什麼有趣的結果



1. 可以看到如果每個partition都有更多hot record的話，throughput會變得比較多，推測是因為不容易都要拿某個record而被lock擋住
2. 但也可以看到當hot recor又太多的話，會導致反而又開始下降，推測因為若每個partition的hot record變多，那整體來說就會有更多tx都會需要被reorder和replicate，這樣反而又會讓throughput卡住

Different DoReplication

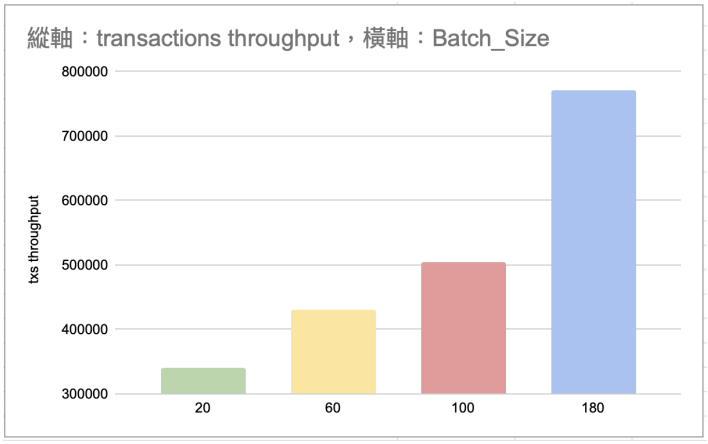
這是我們實作上的一個參數，是在調整看過幾個replication後，才會去重新看有哪些record會是hot record，底下有三個大小分別是100, 1000, 10000然後每個都個跑三次後平均。



1. 可以看出來在 1000 的時候會的時候跑出來的 throughput 會是最好的，太小的話會因為太常去檢查有沒有新的 hot record 而導致效率變差，太大則會因為檢查次數太少導致 hot record 不準確。

Different BATCH_SIZE

我們也試著調整不同Batch_Size，來看看會有什麼實驗結果



1. 可以發現throughput隨著batch_size變大而越來越多，這邊我們不是很理解batch_size代表的意思，我們認為是server會在收到多少個transactions後再做溝通，所以推測應該是server之間不用一直透過網路溝通，可以等多一點tx再一起傳這樣會比較有效率。

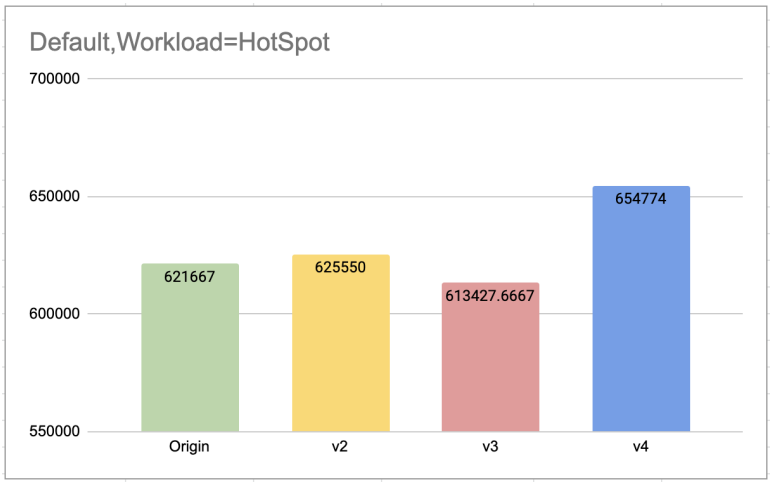
Experiment results on 4 AWS t2.large machines

- 每個實驗的每個數據都是跑了三次後平均的結果

Experiment Environment

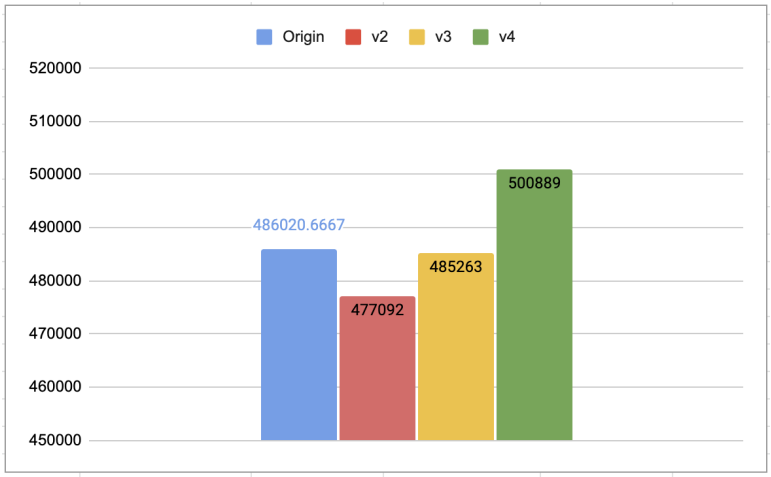
- (2 vcpu(Intel Xeon) + 8GB RAM) * 4

Default, workload=Hotspot



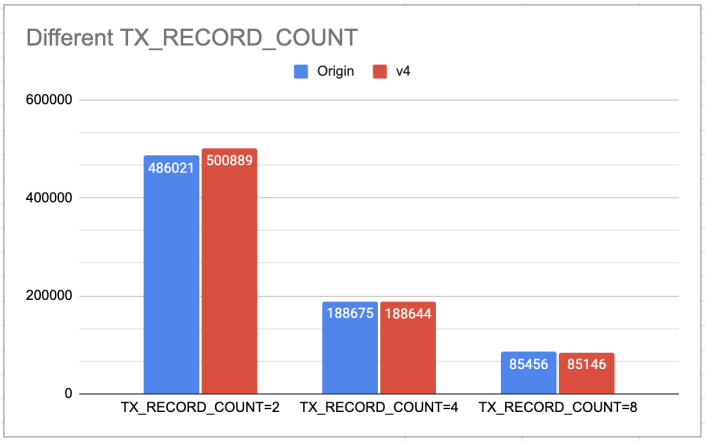
1. 可以看出，我們改的code應該是真的沒有變差，因為在Hotspot上跑了三次平均，甚至還比origin還高
2. 不過如果看每次實驗，會發現 v4 有一次特別高，origin有一次稍微低一些。不過如果剔除掉這些極端值的話，兩者會大概都落在63萬，還是證明了我們不會跑得比較差。

Default, workload=HotCounter



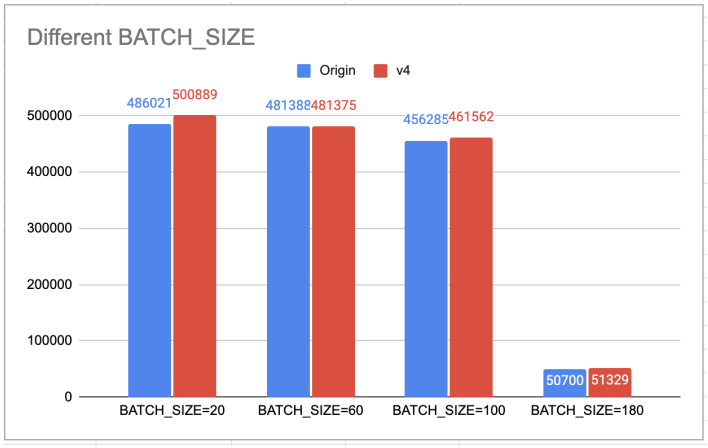
1. 可以看到我們在HotCounter上想做到的優化，在我們做的三次實驗平均後，成功的優化了大約15000的throughput
2. 而我們可以看到v2在single machine上跑的時候，還會比origin好，但放到多台server上時，就會變差不少，我們推測是因為在單台機器上，網路的情況都會比較穩定，所有replication到所有機器上的成本不會太大，但若分散到各處的話，要透過網路溝通就變得不穩定，導致會有更多的overhead，並且 v2 本來就會多複製一些不必要的 tx 因此跑得比較慢也是符合我們預期

Different TX_RECORD_COUNT



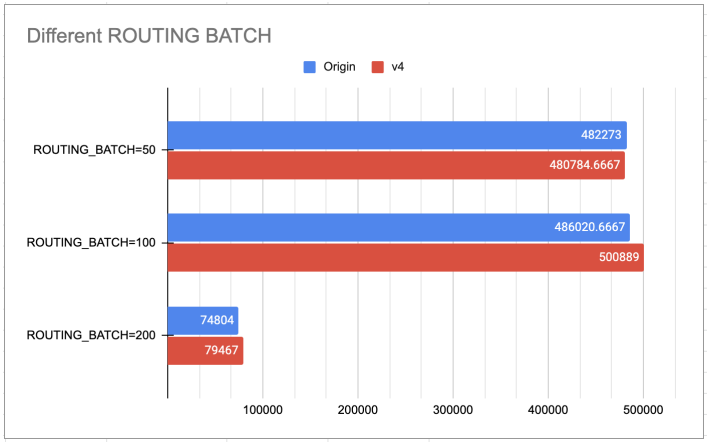
1. TX_RECORD_COUNT代表的是一個transaction中所需要access的record數量，而當每個transaction需要access的數量變多了，要完成該transaction所要花的時間也會增加，因此在定時間內，可以完成的transaction數量也會減少。
2. 另外相較於原版(原本尚未修改的code)，我們修改的最終版本雖然在defualt的TX_RECORD_COUNT=2時，表現好了一些，但主要是在多次實驗上，數據變化較穩定些，但數值比對上，並沒有呈現全面性優化，因此我們認為TX_RECORD_COUNT對於兩個版本間的優化，沒有太大的影響。

Different BATCH_SIZE



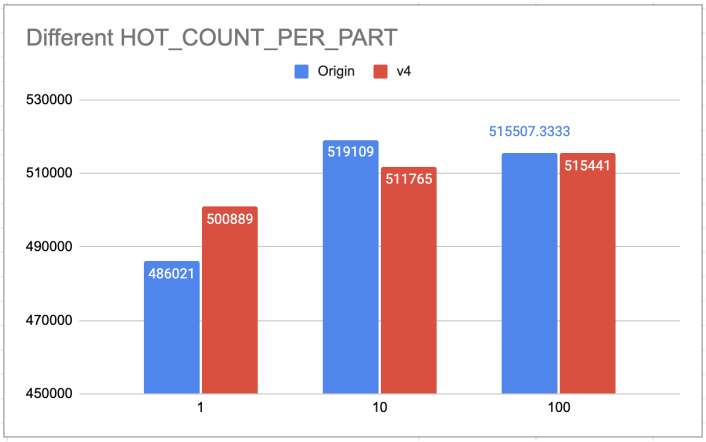
1. 這裡做出來的實驗結果和 single machine 完全相反，我們這裡不是很確定跟 single machine為什麼會產生完全不一樣的走勢，因為在single machine 上我們測出來的結果為，Batch_size越大，throughput也會越好，但我們有猜測兩個原因
    1. 因為180十分貼近我們 default 的RTE數(=200)，如果 client 的 request 一次累積這麼多才送出的話，可能是 server 一下子無法 handle 那麼多 request，導致資源都一直互相卡著 resource，所以 avg latency 都會變很高，throughput也驟降。
    2. 在 AWS 機器上的 server，memory空間不夠大，導致一下來了大量的 request 會塞爆他的記憶體，所以才會有這樣的結果。

Different ROUTING_BATCH



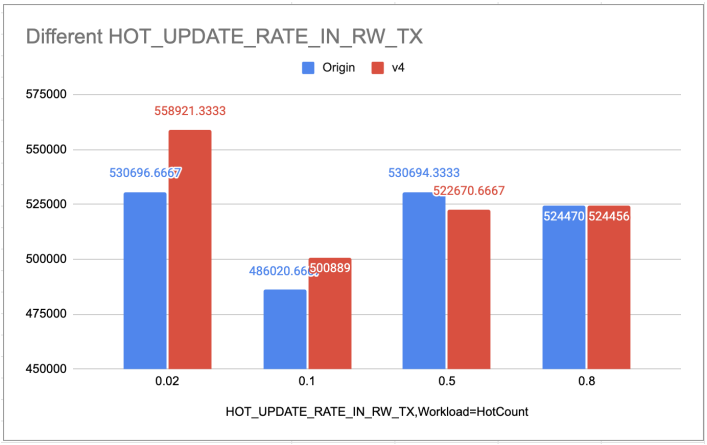
1. ROUTING_BATCH代表的是server一次會囤積多少從各個clients送來的request，再一次做處理，分配給其他servers執行，而我們的實驗是跑了一個client，也就是說，倘若我們的RTE數設為200，則ROUTING_BATCH就不能超過200*1=200。
2. 從上面的實驗可以看出，當ROUTING_BATCH為50或100時，表現都不錯，但到了200，throughput出現了驟降，在此推測造成此狀況的原因應該與上述BATCH_SIZE過大類似，即若ROUTING_BATHC過大，囤積大多request才執行，有可能導致server應付不來，進而導致throughtput下降。

Different HOT_COUNT_PER_PART



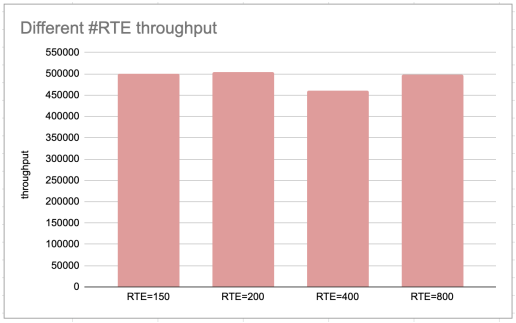
1. HOT_COUNT_PER_PART代表的是每個partition有多少hot record。而先就不同的HOT_COUNT_PER_PART來說，如同在single machine那般，若每個partition的hot record數變多，transaction在拿取hot record時就相對較不易發生conflict，彼此都想access的同一個record的狀況會減低，較不用互等record的lock release，因此throughput也有所提升。
2. 就不同版本來說，我們的修改版會將hot record做replication，因此若每個partition的hot record數量變多，在做replication的overhead就會提升，因此在優化上，可以看出我們在HOT_COUNT_PER_PART變多的狀況，優化反而變差了。

Different HOT_UPDATE_RATE_IN_RW_TX

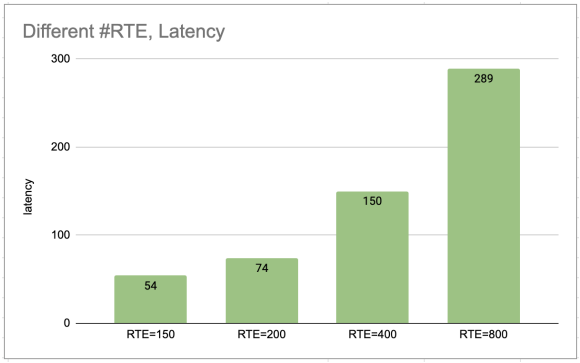


1. 這個實驗是我們比較意外的部分，因為我們原本以為如果要對 hot record 更多的 update 的話，應該會跟 single machine 的結果差不多，就是會隨著比例上升，整體的 throughput 下降，但這邊發現其實差不多。
2. 我們討論後覺得原因可能是因為，跑在雲端的機器上，網路也會是卡著 throughput 的因素，因此有可能是我們網路的速度才產生這樣的結果。

Different #RTE



1. 我們可以看到，在這邊 RTE 數量的多寡跟最終的 throughput 關係不大，只要兩個batch_size 和 RTE 的 restriction 都有符合，這樣就影響不大

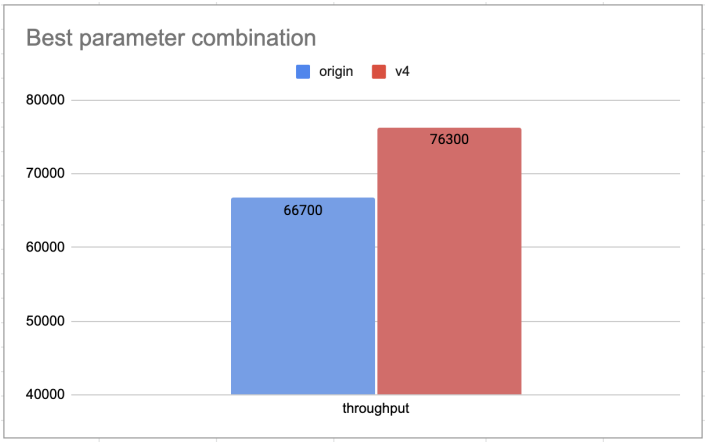


2. 但相對的，如果RTE越多，越有可能會互相搶 resource，因此 avg latency 也會上升，所以才會導致，RTE 數量變多，但相對的 throughput 也沒有變好的狀況

Best Parameter combinations

我們在看完我們的所有實驗數據後，選出了一組我們認為可以造成進步幅度最大的一組parameter

```
org.elasql.schedule.tpart.TPartPartitioner.ROUTING_BATCH=200
org.elasql.remote.groupcomm.client.BatchSpcSender.BATCH_SIZE=20
org.elasql.bench.benchmarks.ycsb.ElasqlYcsbConstants.HOT_COUNT_PER_PART=1
org.elasql.bench.benchmarks.ycsb.ElasqlYcsbConstants.HOT_UPDATE_RATE_IN_RW_TX=0.02
org.elasql.bench.benchmarks.ycsb.ElasqlYcsbConstants.TX_RECORD_COUNT=2
```



1. 以上是我們跑出來的結果，可以看到v4跑得比origin多了約一萬的throughput。