DB21 Team21 Final Project Report

Abstract

我們在 DEMO 後回去找到 replication 和 reordeing 實作上的問題，並且升級 AWS 上面的設備的規格以及網路速度，重新跑過所有實驗，發現新的做法能比 Hermes 在 HotSpot workload 上面，最好提升 5% 的 total throughputs；針對其他沒有辦法 improve 的 setting，我們也從實作角度提出我們自己的分析。除此之外，我們也證實這次改版後在 Google wrokload 上面可以比 Hermes 提升大約 6% 的 throughputs。

Implementation

increase check frequency: 如下圖所示，原本的TimeUnit為SECONDS，
我們改為MILLISECONDS以增加檢查資料的頻率

```
public class ZipThread extends Thread {
    boolean stop = false;

    @Override
    public void run() {
        while (!stop||!resultSets.isEmpty()) {
            try {
                TxnResultSet resultSet = resultSets.poll(1, TimeUnit.MILLISECONDS);
                if (resultSet!=null) {
                    analyzeResultSet(resultSet);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public void stopRunning() {
        stop = true;
    }
}
```

hashcode: 有些hashcode值我們可以在資料增減時便先計算出來，之後就不需要重複計算。

```
public static CachedRecord newRecordForInsertion(PrimaryKey key,
        Map<String, Constant> fldVals) {
    CachedRecord rec = newRecordWithFldVals(key, fldVals);
    rec.isNewInserted = true;
    rec.isDirty = true;
    rec.hashCode = 31 * rec.hashCode + key.hashCode();
    rec.hashCode = 31 * rec.hashCode + fldVals.hashCode();
    return rec;
}

public static CachedRecord newRecordForDeletion(PrimaryKey key) {
    CachedRecord rec = new CachedRecord(key);
    rec.isDeleted = true;
    rec.isDirty = true;
    rec.hashCode = 31 * rec.hashCode + key.hashCode();
    return rec;
}

public CachedRecord(PrimaryKey primaryKey) {
    this.primaryKey = primaryKey;
    this.hashCode = 31 * this.hashCode + this.hashCode();
}
```

Conservative Lock Early Release: 根據Calvin的論文內容，為了確保Deterministic Execution，只要Transaction一開始執行，就必須執行到結束，根據此情況我們認為可以在Record被Flush時Release xLock以減少其他Transaction的Waiting time以及增加

Throughputs。

```
public void flush() {
    for (PrimaryKey key : writeKeys) {
        CachedRecord rec = cachedRecords.get(key);

        if (rec.isDeleted()) {
            VanillaCoreCrud.delete(key, tx);
            ConservativeOrderedCcMgr ccMgr = (ConservativeOrderedCcMgr) tx.concurrencyMgr();
            ccMgr.releaseWriteLock(key);
        }
        else if (rec.isNewInserted()) {
            VanillaCoreCrud.insert(key, rec, tx);
            ConservativeOrderedCcMgr ccMgr = (ConservativeOrderedCcMgr) tx.concurrencyMgr();
            ccMgr.releaseWriteLock(key);
        }
        else if (rec.isDirty()) {
            VanillaCoreCrud.update(key, rec, tx);
            ConservativeOrderedCcMgr ccMgr = (ConservativeOrderedCcMgr) tx.concurrencyMgr();
            ccMgr.releaseWriteLock(key);
        }
    }

    writeKeys.clear();
}
```

```
public void releaseWriteLock(Object obj) {
    lockTbl.release(obj, txNum, LockType.X_LOCK);
}
```

Reordering & Replication： 主要修改HermesNodeInserter.java以及TPartPartitioner.java，根據助教的提示，我們的實作步驟基本為：

1.calculate read/write numbers, and then find hot records

這邊我們同時考慮 read, write records 的情況，如果有讀或寫到該個 records ，就會用一個 hashMap 紀錄該 record 的 key，並讓 value +1。

```
for (TPartStoredProcedureTask task : tasks) {
    totalNumberOfTxs++;
    for (PrimaryKey key : task.getReadSet()) {
        if(RWCount.containsKey(key)) {
            int count = RWCount.get(key)+1;
            RWCount.put(key, count);
        }else {
            RWCount.put(key, 1);
        }
    }
    for (PrimaryKey key : task.getWriteSet()) {
        if(RWCount.containsKey(key)) {
            int count = RWCount.get(key)+1;
            RWCount.put(key, count);
        }else {
            RWCount.put(key, 1);
        }
    }
    // QMurphy: recalcalateHotRecord 的時機設定
}
```

等到全部都計算完畢後，我們會遍歷 (xxx) 判斷如果 `count / all txn counts` > 0.25 的話，我們就把它視作 hot records。 至於為什麼是 0.25 我們會在下面的實驗說明。

```
hotKeys.clear();
double maxRatio = 0.0;
for (PrimaryKey key : RWCount.keySet()) {
    double ratio = RWCount.get(key) * 1.0 / totalNumberOfTxs;
    if(ratio > maxRatio) {
        maxRatio = ratio;
    }
    if ((totalNumberOfTxs > 1000) && (RWCount.containsKey(key)) && (RWCount.get(key) * 1.0 / totalNumberOfTxs
        hotKeys.add(key);
    }
}
```

(助教！我們這邊和 DEMO 不同！)

在 DEMO 後我們發現這個 threshold 如果設定的太高（我們之前設定的太高了），`hotKeys` 根

本就不會有東西進去，也就不會觸發後面的插入 replication node 的部分，這樣 replication 的機制就等於沒有做(有很多潛在的 bug 也就不會觸發)，這應該就是之前我們做出來跟原本 Hermes 一樣甚至更差的原因。

2.Find hot records which will be updated and insert them to the graph

接下來我們把有要寫入 data 到這些 hot racords 裡面的 transaction node 先利用原本助教 code 就已經實作好的 `insertAccordingRemoteEdges()` 插到 graph 中。

3.Insert replica to the graph

因為每個 transaction node 都需要一個 stored procedure task，裡面會紀錄 pid, client id, txnNumber 等等，但是 replication node 是我們產生的，所以我們需要手動在這邊幫每個 replication node new 一個新的 stored procedure task 出來，再插入到所有的 partition server 上。

```
for (int partId = 0; partId < partMgr.getCurrentNumOfParts(); partId++) {
    StoredProcedureCall call = new StoredProcedureCall(-1, -1, -1234, new Object[] {});
    TPartStoredProcedureTask task_ = createStoredProcedureTask(call);
    graph.insertReplicationNode(task_, hotKeys, partId);
}
```

(助教！我們這裡跟 DEMO 不同！)

如果不幫每個 replication node 都插入一個新的 task 的話，在 `decideExceutionPlan()` 那邊會因為下面這個判斷式被卡住(這是之前我們沒有注意到的部分)：

```
if (plan != null) {
        throw new RuntimeException("The execution plan has been set");
    }
```

除此之外，在 `executeTransactionLogic()` 這邊，因為 replication node 不需要執行 `executeSql(readings)` 所以我們會在這邊做判斷。

```
if(!plan.getIsDoingReplication()) {
    executeSql(readings);
}
```

插入 replication node 的邏輯如同其他 node，都是加入 read write edge。

4.Insert remaining nodes to the graph

剩餘其他沒有要寫入資料到 hot records 的 transaction，我們一樣是利用 `insertAccordingRemoteEdges()` 把 txn node 插到 graph 中。

5.Balance machine...etc

這部分的邏輯大部分沒有做修改，只有像是 `countRemoteReadWriteEdges()` 或是 `countRemoteReadEdge()` 這兩個 function，如果輸入的 record key 屬於 hot record 的話，就會 skip 掉。

Experiments and Discussion

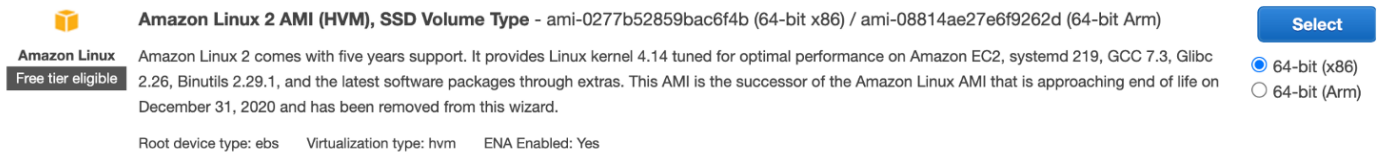
(助教！下面的實驗全部都重新做了，跟 demo 時的呈現不一樣)

一、Experiment Environment:

(助教！我們租用了更好的設備，與 DEMO 不同了！)

我們在 Amazon AWS EC2 上面註冊四個 instances，分別來跑我們的兩個 server, 一個 sequencer，和一個 client，他們的規格都相同如下：

Intel® Xeon® CPU E5-2666 v3 @ 2.90GHz with 16 processors, 30 GB RAM, 2,000 Mbps for data transfer rate.



二、Experiment Result:

實驗的 default setting 如下：

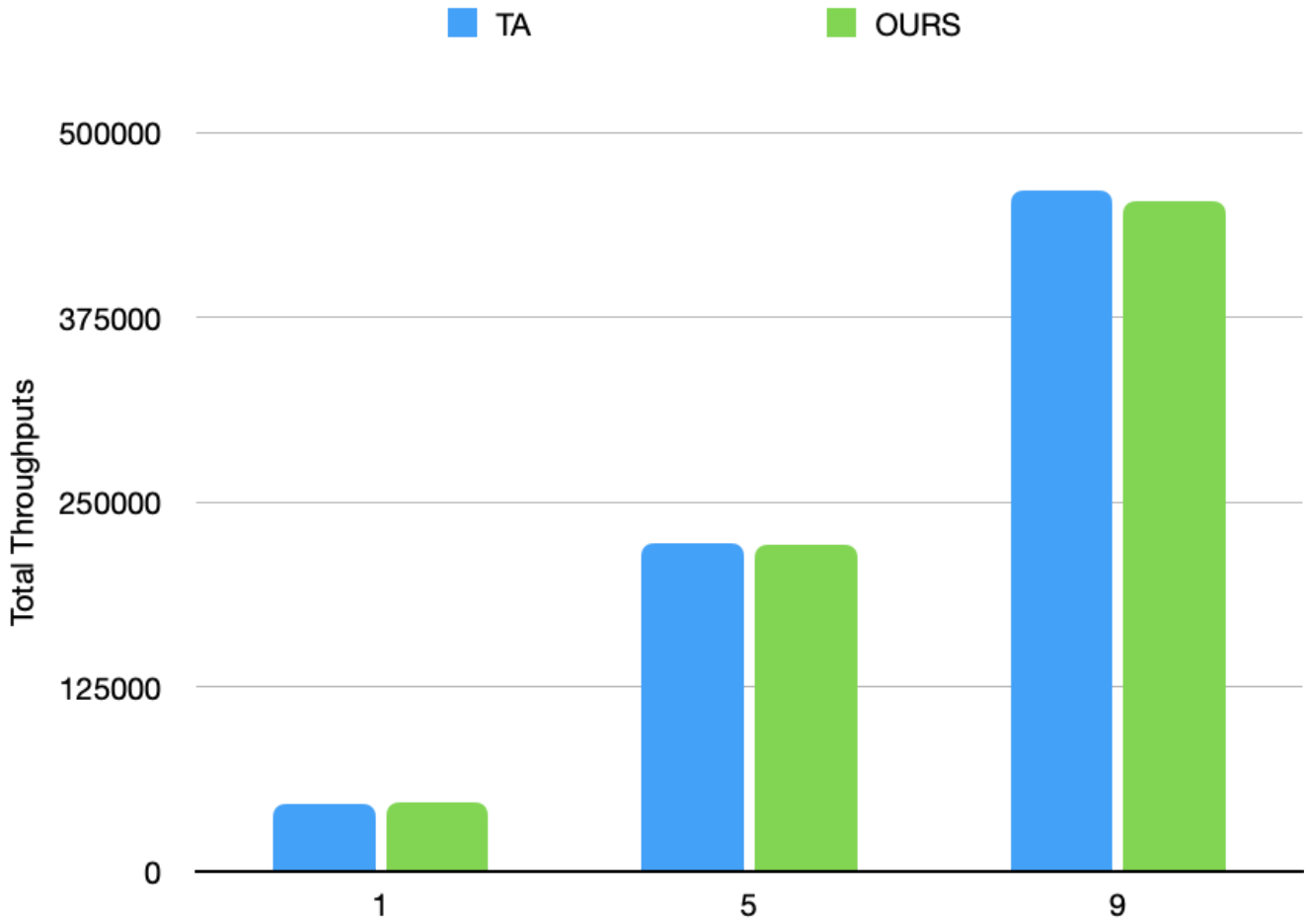
```
org.vanilladb.bench.BenchmarkerParameters.BENCHMARK_INTERVAL=250000
org.vanilladb.bench.BenchmarkerParameters.NUM_RTES=1000
org.vanilladb.bench.StatisticMgr.GRANULARITY=3000
org.elasql.schedule.tpart.hermes.FusionTable.EXPECTED_MAX_SIZE=1000000
org.elasql.schedule.tpart.hermes.HermesNodeInserter.IMBALANCED_TOLERANCE=0.1
org.elasql.schedule.tpart.TPartPartitioner.ROUTING_BATCH=500
org.elasql.remote.groupcomm.client.BatchSpcSender.BATCH_SIZE=25
org.elasql.bench.benchmarks.ycsb.ElasqlYcsbConstants.DATABASE_MODE=1
org.elasql.bench.benchmarks.ycsb.ElasqlYcsbConstants.TX_RECORD_COUNT=2
org.elasql.bench.benchmarks.ycsb.ElasqlYcsbConstants.HOT_COUNT_PER_PART=1
org.elasql.bench.benchmarks.ycsb.ElasqlYcsbConstants.HOT_UPDATE_RATE_IN_RW_TX=0
```

因為這次的優化主要都是針對 HotSpot workload，所以大部分的實驗都會集中在這個 workload 上；並且我們主要關注不同做法對 throughputs 的影響。

另外以下實驗結果都是採用 warm up(90s)之後較為穩定的數據。

- **The Hotspot Workload**

1. 探討 HOT_COUNT_PER_PART 的影響
   我們觀察 1, 5, 10 這三個不同 hot count per partition 帶來的改變，

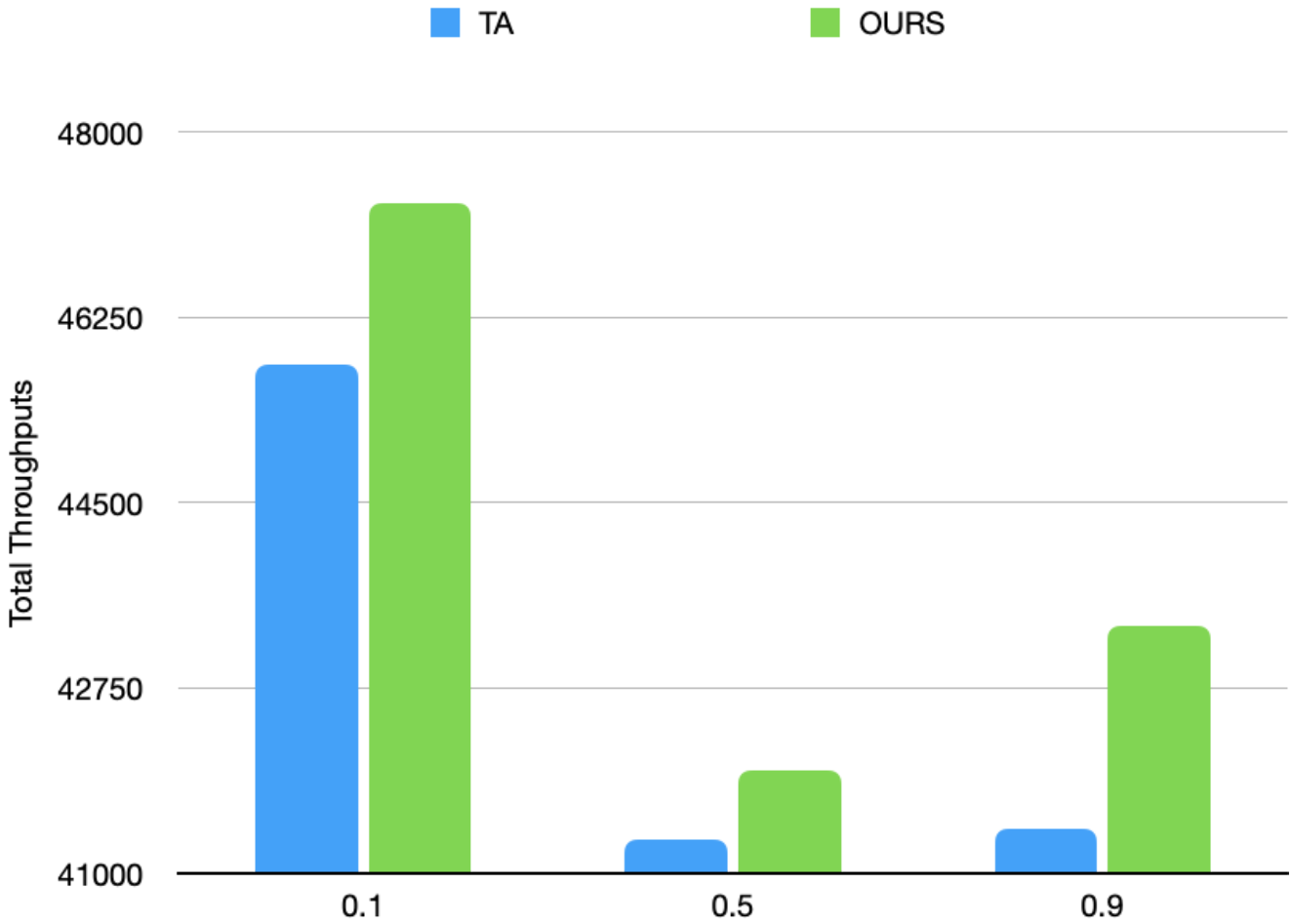


以下是我們實作的版本在不同的 HOT_COUNT_PER_PART 下，相較 Hermes throughputs 提升的比例，可以發現除了參數設為 1 時，我們會有大約 3% 的提升，在設為 5 或 10 時，throughput 都是下降的。

不過這個結果是符合我們的預期的。我們在過程中曾經把存入 `HotRecordKeys` 的 records 數量記錄下來，發現當 HOT_COUNT_PER_PART 的數量越高，會被存入 `HotRecordKeys` 的數量就愈少，這是因為每份 partition 被設定為 hot data 的數量一多，就會分散掉彼此被 read/write 的機率。舉例來說，在HOT_COUNT_PER_PART=1時，**我們發現 records 被access最多次的次數佔總體 transaction 數量的比例大約是 0.524；然而當HOT_COUNT_PER_PART=1時，馬上就會降到 0.1左右，甚至連我們設定的 IS_HOT_RECORD_THRESHOLD 都沒有達到，這種情況就不會 trigger replication 的機制了**。

HOT_COUNT_PER_PART	**1**	**5**	**10**
improve ratio	3%	-1%	-1.6%

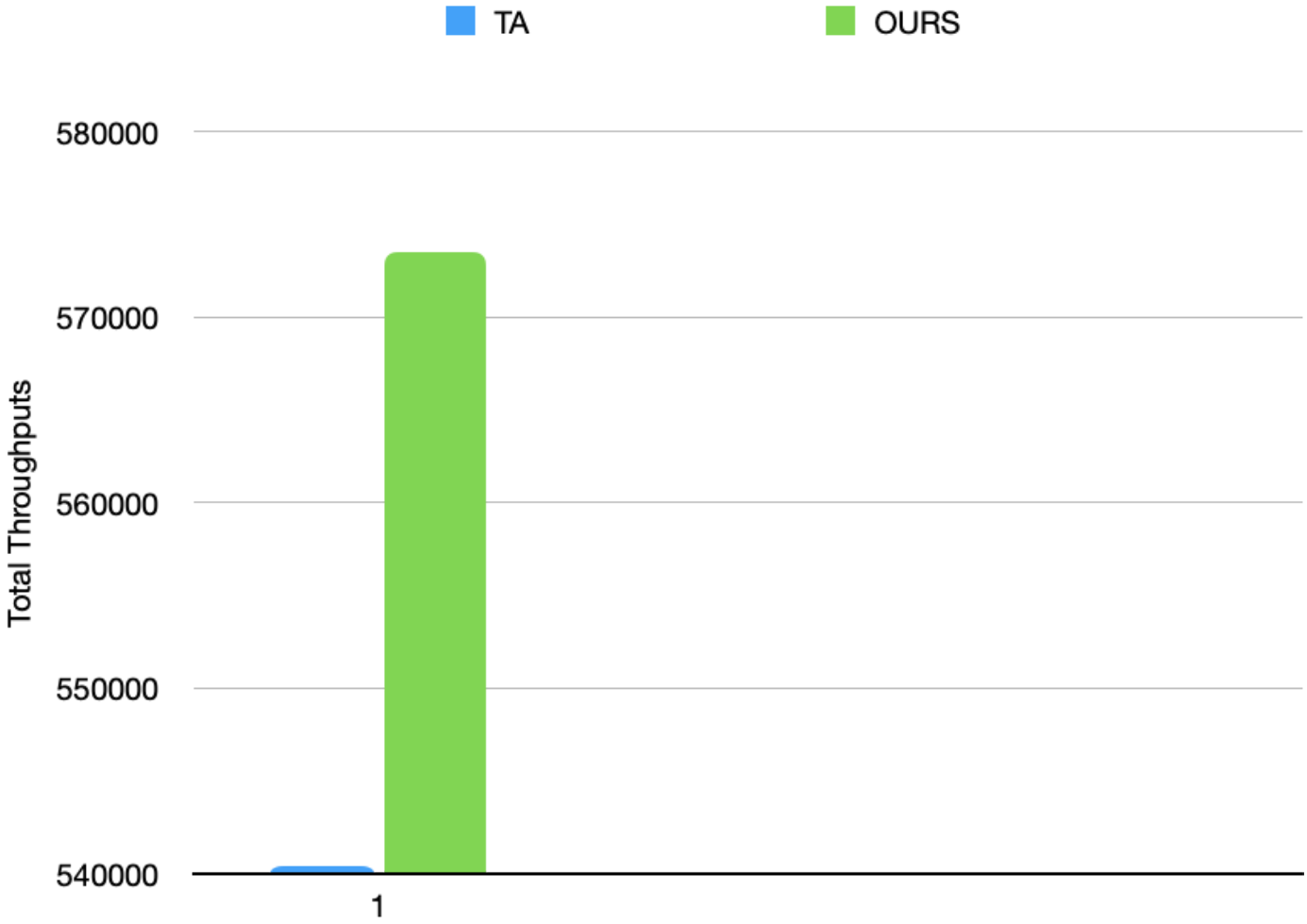
2. 探討 HOT_UPDATE_RATE_IN_RW_TX 的影響
   我們觀察 0.1, 0.5, 0.9 這三個不同 ratio 帶來的改變，值得注意的是，其中 improvement 最多的，是在 update ratio 最高的 0.9 時，我們推測這是因為最初在插入第一批 txn nodes 的時候，我們就是看哪些 transaction 有 write 到 hot records 的，就先把他插入，之後再繼續插入 replication node。可能是因為這種 reorder 機制，使得我們實作的版本對於 update ratio 高的情況，能輸出更高的 throughputs。



以下是我們實作的版本在不同的 update rate 下，相較 Hermes throughputs 提升的比例。可以發現在不同的 update ratio 下，我們也能夠比 hermes 產生更多一些的 throughputs。

update ratio	0.1	0.5	0.9
improve ratio	3%	2%	5%

- **The Google Workload**
  在 Google workload 上面我們的 total throughputs 相較 hermes 大約提升了 6%，顯示對於這種 workload 變化具有相當 unpredictable 的 pattern 特性的情況，我們實作的版本也可以 handle 的很好。


TA
OURS
580000
570000
560000
550000
540000
Total Throughputs
1