Final Project Opt Report

105062206 蔡哲維
105062306 胡世昕

縮短 CacheMagr 查看 Remote Record Package 的間隔

```
// Check every 1 second (fast enough?)
HoldPackage pack = newPacks.poll(100, TimeUnit.MICROSECONDS);
if (pack != null && !handoverToTransaction(pack.txNum, pack.record)) {
    pending.add(pack);
}
```

在 serer 收到 RecordPackage 型別的 P2PMessage 後，將獲得的 remote record 資訊放進 newPacks List 中等待消化。將查看 newPacks 的頻率由一秒一次增加為每 100 微秒一次，減少 Sp Thread 收集 remote read record 的時間。

於 Conservative Lock Table 中導入 Lock Striping

操作 Lock 時不使用 synchronized 將整個 method 包住，而是讓使用 anchors 縮小鎖定範圍後對其使用 synchronized 同步保護。

```
Object getAnchor(Object obj) {
    int code = obj.hashCode() % anchors.length;
    if (code < 0) {
        code += anchors.length;
    }
    return anchors[code];
```

導入 anchor 讓物件區分為多個 bucket，分別由不同鎖控制。

```
void requestLock(Object obj, long txNum) {
    Object anchor = getAnchor(obj);
    synchronized(anchor) {
        Lockers lockers = prepareLockers(obj);
        lockers.requestQueue.add(txNum);
    }
```

將 synchronized 內移，縮小 critical section。

減少重複 HashCode 運算

Calvin.sql.PrimaryKey 中創建物件時已將運算 HashCode 將使用的 tableName, fields, values 定值且不會更改，因此 HashCode 運算不會改變。計算 HashCode 將花費硬體資源。

```
private void calculateHashCode() {
    this.hashCode = 17;
    this.hashCode = 31 * this.hashCode + tableName.hashCode();
    for (int i = 0; i < fields.length; i++) {
        this.hashCode = 31 * this.hashCode + this.fields[i].hashCode();
        this.hashCode = 31 * this.hashCode + this.values[i].hashCode();
    }
    this.hashCodeIsInit = true;
```

在第一次 HashCode 被呼叫時運算並記錄結果。

```
public int hashCode() {
    if(!this.hashCodeIsInit) {
        this.calculateHashCode();
    }
    return hashCode;
```

往後呼叫僅須返回 this.hashCode。

```
private void calCulateHashCode() {
    this.hashCode = 17;
    this.hashCode = 31 * this.hashCode + this.primaryKey.hashCode();
    this.hashCodeIsInit = true;
```

Calvin.cache.InMemoryRecord 中 primaryKey 物件在創建時已定值且不會更改，因此在第一次 HashCode 被呼叫時運算 primaryKey 的 hashcode 並記錄結果

```
public int hashCode() {
    if(!this.hashCodeIsInit) {
        this.calCulateHashCode();
    }
    return (31 * this.hashCode + this.nonKeyFldVals.hashCode());
```

往後呼叫僅須計算 nonKeyFldVals 的 hashcode。

增加 HEART_BEAT TIMEOUT 間隔

```
private static final int HEARTBEAT_PERIOD = 10_000; // in milliseconds
private static final int HEARTBEAT_TIMEOUT = 600_000; // in milliseconds
```

延長 TcpFailureDetectionSession 中 HEART_BEAT TIMEOUT 間隔時間，容忍 server 與 client 在更差的網路延遲下維持運作，並減少 server 開啟失敗的機率。

使用 public cache 保存 cache 資料

每個 transaction 都會生成 transactionCache 暫存 remote 與 local 讀取的 record 以及對其執行的操作。但在 transaction 結束後會將 cache 中的 record flush 進 stroage 中並刪除該 cache。在 transaction commit 後將該 cache 放入 public cache 中可加速往後 local read 的速度。

```
private Map<PrimaryKey, InMemoryRecord> inMemoryDatas = new ConcurrentHashMap<PrimaryKey, InMemoryRecord>();
```

在 Calvin.cache.CacheMgr 中加入 public cache。

```
protected void afterCommit() {
    Calvin.cacheMgr().moveTxnCacheToInMemoryDatas(this.txNum);
```

```
public void moveTxnCacheToInMemoryDatas(long txNum) {
    this.inMemoryDatas.putAll(caches.get(txNum).getCacheRecords());
```

在 transaction commit 後將 transactionCache 中的資料放入 public cache。

```
private Map<PrimaryKey, InMemoryRecord> performLocalRead(Set<PrimaryKey> readKeys) {
    Map<PrimaryKey, InMemoryRecord> localReadings = new HashMap<PrimaryKey, InMemoryRecord>();

    // Read local records (for both active or passive participants)
    for (PrimaryKey k : readKeys) {
        InMemoryRecord rec = cache.readFromLocal(k);
        if (rec == null)
            throw new RuntimeException("cannot find the record for " + k + " in the local stroage")
        localReadings.put(k, rec);
```

執行 local read 時先於 public cache 中查找，查找失敗後再進入 storage 本身進行查找。

Experiments

Method: 使用 Windows HyperV 開啟四台 VM 模擬實體電腦，每台 VM 擁有實體 IP 可以互相溝通。3 台 VM 為 Sever，1 台為 Client。

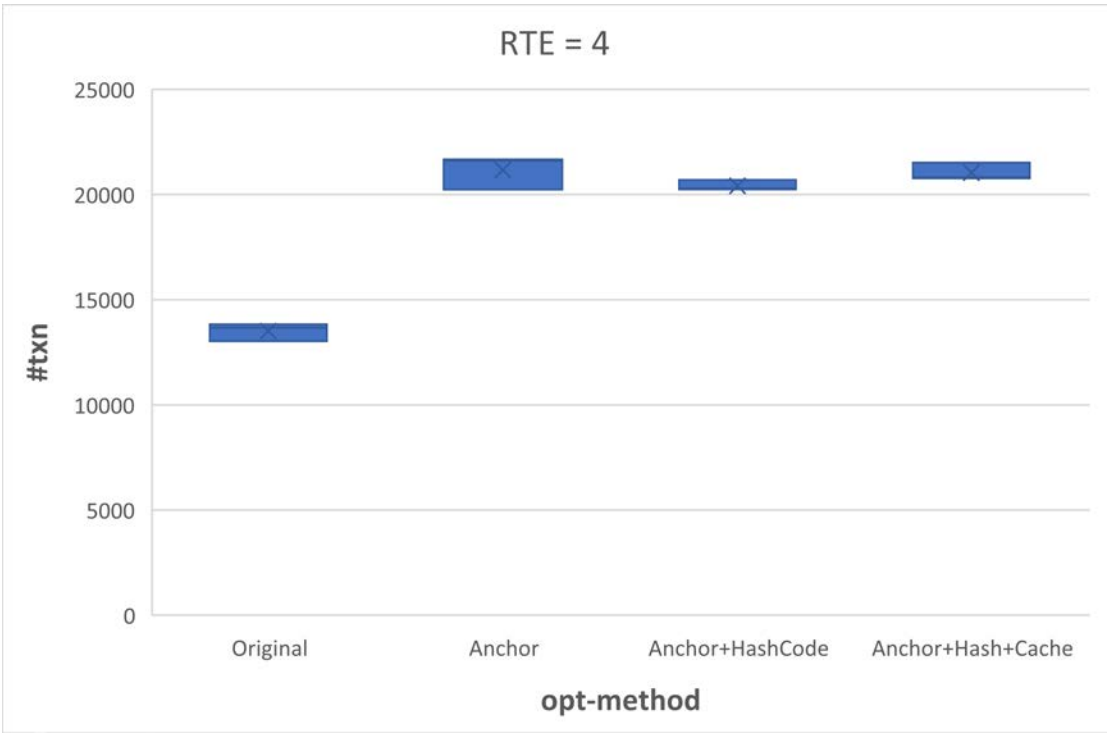
Environment:

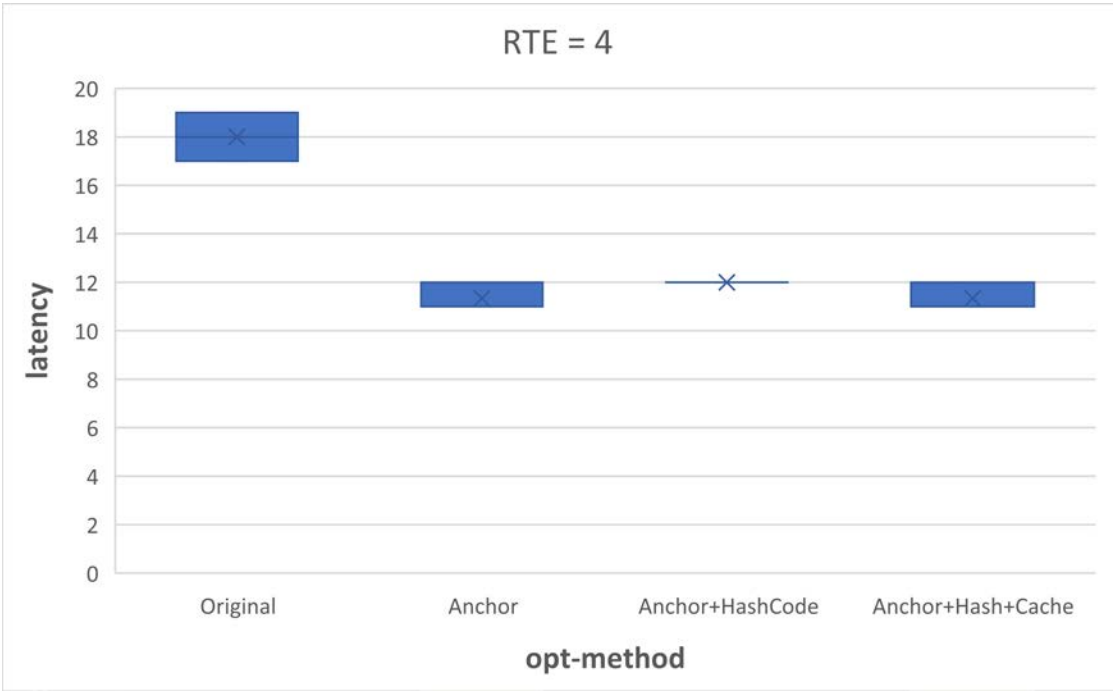
Host: AMD 2990WX 32Core CPU, 128G RAM, Win server 2019

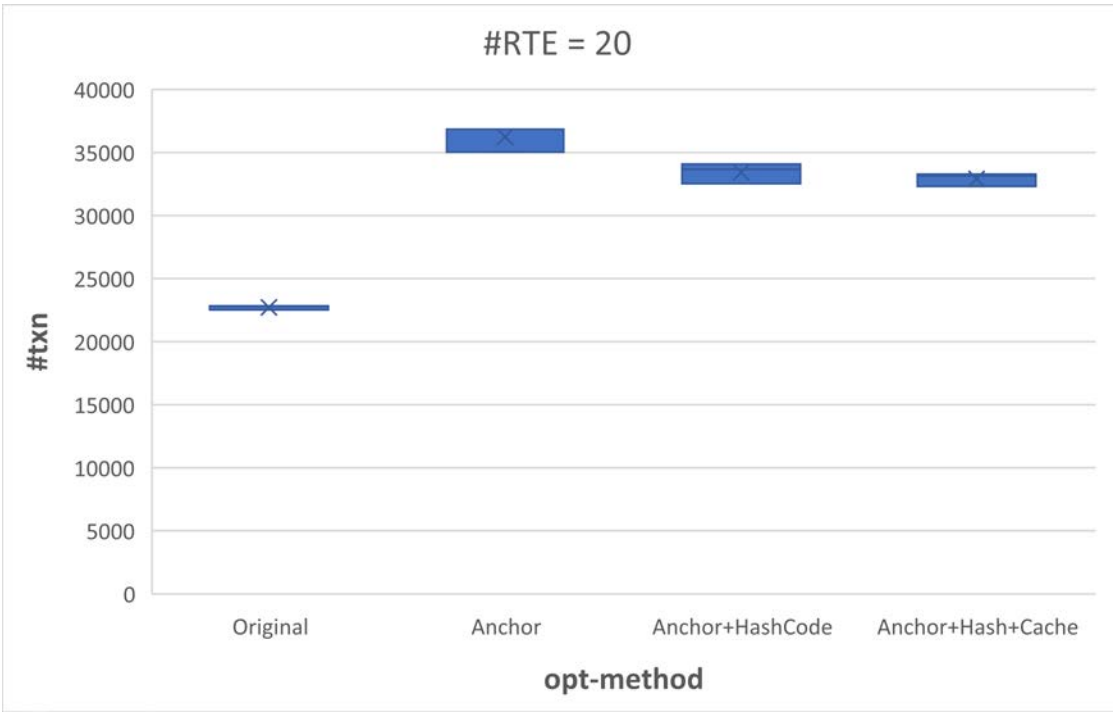
Guest: 16thread virtual CPU, dynamic allocated RAM, 1G RAM Disk, win10

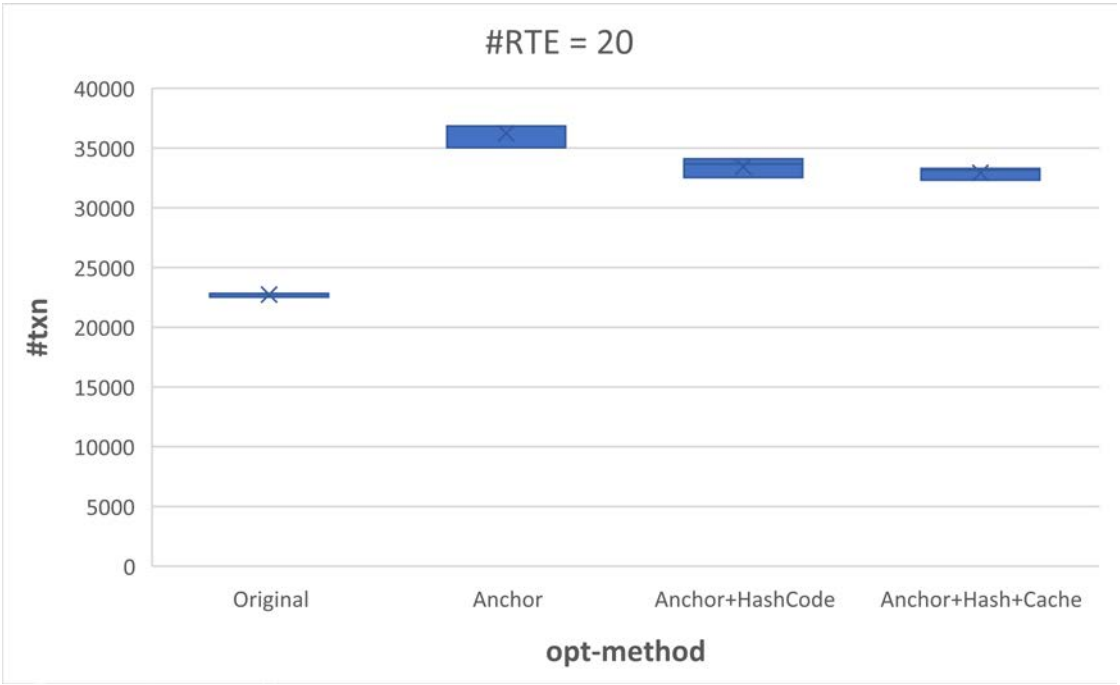
Experiment parameters: TPC-C Benchmark, server partition = 3, NUM_WAREHOUSES=2, FREQUENCY_TOTAL=100, FREQUENCY_NEW_ORDER=50, FREQUENCY_PAYMENT=50。

Results:


RTE = 4
25000
20000
15000
10000
5000
0
#txn
Original
Anchor
Anchor+HashCode
Anchor+Hash+Cache
opt-method


RTE = 4
20
18
16
14
12
10
8
6
4
2
0
latency
Original
Anchor
Anchor+HashCode
Anchor+Hash+Cache
opt-method





Analyze:

就實驗結果而言，對 Conservative Lock Table 做的優化使 throughput 增加了 50%左右。使用 anchors 將傳入物件區分為可各自上鎖的小區塊，更精確地上鎖可減少對 object 的獨佔，讓該 object 可以被更多 thread 同時使用，減少等待時間，導致 throughput 大幅增加。

減少重複計算 HashCode 並沒有對 throuput 提供明顯優化結果，推測原因為實驗結果受網路穩定性因素影響，導致實驗結果誤差標準差很大，HashCode

優化成果可能被誤差雜訊所覆蓋而無法觀察出其效果，且 Calvin.cache.InMemoryRecord 中僅有 primaryKey 的 HashCode 可被預先計算，每次呼叫 HashCode()都必須再次計算 nonKeyFldVal 的 HashCode，使得優化 HashCode 的效果更加細微。

我認為加入 public cache 優化的效果不明顯和 HashCode 優化不明顯的原因相同，都是被網路誤差因素覆蓋。TPC-C benchmark 中 New Order 與 Payment 會執行大量的資料修改，即代表大量的寫入。Public cache 加速 local read，大量的寫入又進一步削弱優化的成果。