MapReduce 架構下循序樣本探勘演算法之效能分析

陳世穎　魏秀蕙　蔡岳峻　許珉豪　郭立辰
國立臺中科技大學資訊工程系
sychen@nutc.edu.tw, s24916007@gmail.com, birdtasi@gmail.com,
rickyhsu1942@gmail.com, webbers81411@gmail.com

摘要

由於雲端科技的普及和大量資料的累積，如何有效率地縮短時間處理分析巨量資料，成為一個重要的研究方向，尤其是巨量資料的資料探勘技術。本研究的目的是透過 MapReduce 架構平行化設計並分析兩種不同類型探勘演算法，包括循序探勘模式之 AprioriAll 演算法與 GSP 演算法，有效處理大型資料庫並縮短執行時間。實驗結果顯示， GSP 平行化演算法較 AprioriAll 平行化演算法有較佳的探勘時間。

關鍵詞：資料探勘、循序樣本、關聯式規則、平行運算、MapReduce

1. 前言

雲端科技已經成為現代人生活中不可獲缺的應用。由於雲端技術的提升，資料不斷的累積，進而成為大量的資料，也就是所謂的巨量資料(Big Data)時代。因此更需要有效率的巨量資料分析技術，讓使用者能從巨量資料中，發掘出適合有用性的數據資料，以適合各類應用。

現今已有許多各樣資料探勘技術的應用，所謂的資料探勘是從資料庫中找尋未知卻有用的知識訊息。資料探勘的技術有資料方塊 (Data Cube)、分類分析 (Classification) 、群集分析 (Clustering Analysis)、關聯分析(Association Rule Analysis)、序列樣式相關分析(Sequential Pattern Analysis)、鏈結分析(Link Analysis)、時間序列相似性分析(Time Series Similarity Analysis)。其中，關聯規則(Association Rules, AR) [3]是資料探勘中最早被提出來的重要問題，主要是用來探勘出資料項目之間彼此相互隱藏的關連性。

原來的關聯規則探勘並無法處理具有時間相關特性的資料庫，所以學者們以此為改進的依據，發展出與時序相關的樣本探勘技術，稱為循序樣本探勘 (Sequential Pattern Mining)[1,5]。循序樣本探勘是由資料庫內搜尋出經常出現的樣本，並指出樣本內各項目出現的時序，被視為一種時間性資料探勘，此探勘的複雜度遠高於關聯規則。常見的循序樣本探勘包括AprioriAll演算法 [1, 4]與GSP演算法[5]。

以上探勘方法各有優劣，為了可以有效率的處理巨量資料並量測其效能，本文將以MapReduce的軟體架構來平行化AprioriAll演算法與GSP 演算法，並量測兩個演算法在平行運算下的效率。

MapReduce 是由 Google 發表的一種軟體框架，主要技術是將資料本地化 (data locality)，並將運算節點與資料放置一起加快存取速度，適用大量資料平行運算，處理運行在眾多電腦組成的叢集(clusters)上的資料。MapReduce 執行的過程包括將 Master 主機和其他 Slaves 機器先複製好需使用的 MapReduce 程式，再由 Master 主機指派 Slave 機器那幾台負責執行 Map 而那些負責執行 Reduce。切割好的區塊資料將分派到負責 Map 程式的 Slave 機器來執行並儲存 Map 後的結果，再執行 Reduce 程式遠端讀取每一份 Map 程式處理後的結果，進行排序、合併、彙整；最後，取得運算結果。

本文其餘結構如下。第二節是相關研究，包括循序樣本探勘演算法；第三節是方法設計;第四節是實驗結果;第五節是結論。

2. 相關研究

關聯式規則主要是透過每筆相同交易探勘出交易商品項目彼此之間重要的關聯性，也就是此交易商品項目被購買後連同其他的商品項目也會一同出現被購買。而循序樣本探勘與關聯式規則類似，差異點在於循序樣本探勘使用關聯式規則並加入時間點的屬性來探勘交易商品的先後順序。

Rakesh Agrawal和Ramakrishnan Srikant等學者於提出AprioriAll演算法[1]，是循序樣本探勘中最基礎的演算法，主要的觀念來自於Apriori演算法[2]。

AprioriAll 演算法主要包含下列步驟。第一、前排序階段。將原始資料的客戶編號與交易時間進行排序。第二、找出最大的項目集階段。此階段與關聯式規則相同，找出滿足最小支持度之最大項目集。第三、轉換項目集。刪除資料庫中非最大項目集，再將資料的內容轉換為整數值，以便進行下一個步驟計算。第四、再產生大型序列(large sequence)。將轉換好的資料再產生大型序列，一直重覆到無法再產生任何大型序列為止。第五、找出最大序列，由第四步驟的大型的序列中再找出最大的序列。第六、將最後搜尋的最大序列，所對應的交易整數編號，還原為原始的交易項目。

AprioriAll 演算法[1, 3]如圖1所示。其中行1產生大型項目集合，行2~9產生候選序列，並找出滿足最小支持度之最大項目集。

Srikant等學者提出循序樣本探勘演算法GSP (Generalized Sequential Pattern) [5]，在循序樣本探勘演算法下找出購買商品之間的順序。GSP演算法以 AprioriAll 為基礎，改善缺少時間的限制與分層

的類別，且適用實行於分散式的處理。

```
1)   L1 = {large 1-sequences}；
2)   for (K = 2；Lk-1≠Ø；k++){
3)     Ck = New candidates generated from Lk-1
4)     foreach在資料庫中的客戶序列C
5)       Increment the count of all candidates in Ck
6)         that are contained in c.
7)     Lk = Candidates in Ck with minimum support.
8)   }
9)   Answer = Maximal Sequences in∪k Lk；
```

圖1 AprioriAll 演算法[3]

GSP演算法與AprioriAll演算法不同之特性包括增加時間的限制與設定滑動時間視窗。增加時間的限制為具備時效性，從交易項目集中，限制最大時間(max-gap)與最小時間(min-gap)的間隔。設定滑動時間視窗則設定每筆交易時間範圍，能允許不同交易項目增加交易時間彈性。

GSP演算法[6]如圖2所示。行1~行8 LS_1產生大型項目集合，再透過大型項目集合來不斷產生下一個階段的大型項目集合，直到無任何大型序列產生為止，其中行5候選序列需滿足最小支持度。

```
1)   LS1 = 大型1-序列所成的集合；
2)   for (k=2 ; LSk-1 ≠ Ø; k++){
3)     CSk = 候選k-序列所成的集合；
4)     foreach在資料庫中的客戶序列C do
5)       對於CSk中每一個候選序列S，
         若C包含S則將S的支持個數增加1；
6)     LSk =在CSk中滿足最小支持個數的
       候選序列所成的集合；
7)   }
8)   Answer = Maximal Sequences
```

圖2 GSP 演算法[6]

3. 方法設計

本研究設計需先將 AprioriAll 演算法與 GSP 演算法以 JAVA 語法與和 MapReduce 架構設計 AprioriAll 演算法與 GSP 演算法的平行應用程式。

3.1 平行化 AprioriAll

AprioriAll 平行化虛擬碼如圖 3 至圖 8 所示，依據 AprioriAll 單機演算法區分為六個階段。

第一階段是 Sort Phase，如圖 3，將每一筆客戶交易記錄分別在不同的 Mapper，執行相同的 Map function，以客戶交易的 ID，交易的間做為雙索引，把購買項目作為 value 輸出，再透過 Map 與 Reduce 之間的 Partitioner 階段將的資料輸入分別指定到不同的 Reducer 進行 Reduce function，進行最後處理，最終輸出以客戶 ID 為索引的交易資料表(New-transaction)。

```
    input: <key,transaction Tx>
    output:<KeyPair T,Item I>
    Map-class{
1)    Map-function{
2)      for each value T in Tx
3)        key.set(T.客戶 ID , T.交易時間);
4)      value.set(T.交易項目);
5)      output(key,value)
      }
    }

    input:<WritableComparablew1,
           WritableComparable w2>
    output:0,1,-1
    GroupingComparator-class{
1)    GroupingComparator-function{
2)      return w1.客戶 ID;= = w12 客戶 ID; ? 0 :
          (CID1 < CID2? -1 : 1);
      }
    }

    input:<KeyPair T , Item I>
    output:<1,2,3,....,n>
    Partition-class{
1)    Partition-function{
2)      return (Key.客戶 ID) % (reducer 個數);
      }
    }

    input:<KeyPair T,list(I1,I2,I3,....Ix)>
    output:<T.客戶 ID,All-Item of Ix>
    reduce-class{
1)    reduce- function {
2)      Key.set(T.客戶 ID);
3)      for each Item Ii in list
4)        All-Item+=Item;
5)      value.set(All-Item);
6)      output<Key,All-Item>;
      }
    }
```

圖 3 AprioriAll 平行化虛擬碼: Sort Phase

第二階段是 Itemsets Phase，如圖 4，以 New-transaction 作為輸入，分別把各客戶交易資料分配到不同的 Mapper，分開搜尋客戶交易順序，輸出客戶交易序列資料表後，再透過 Reducer 加總計算支持個數，找出滿足最小支持度的大型項目集，並將大型項目集對應到連續的整數。

第三階段是 Transformation Phase，如圖 5，把 Mapped-Table 設為此次的 DistributedCache-file，且載入 New-transaction 轉換階段移除交易不包含任何大型項目集，完成後直接輸出產生的 Mapped-table 作為後面各階段的輸入資料。

第四階段是 Sequence Phase，如圖 6，載入 Mapped-transaction，各客戶已轉換的交易資料分散到各台 Mapper 中，並找出客戶交易出現的 Candidates-1- sequence 並輸入 Reduce function 判斷是否符合 min-support。

```
    input:<key,New-transaction Nx>
    output:<Item-Set Ai ,1>
    Map-class{
1)      Map-function{
2)          for each Item-Sequence Ai in Nx
3)              for each Item Iy in Ai
4)                  Tree.add(Iy);
5)              for(Tree : subset){
6)                  Key.set(subset);
7)                  Value.set(1);
8)                  output(Key,value)
            }
        }
    }

    input:<Item-Set Ai,list(1,1,1,...,1) >
    Output:<Item-Set Ai,suppot of Ai>
    Reduce-class{
1)      Reduce-function{
2)          for each value 1 in list
3)              support +=1;
4)          if(support ≥  minimum_support)
5)              output <Ai,support>
        }
    }
```

圖 4 AprioriAll 平行化虛擬碼：Itemsets Phase

```
    input:<Key,New-transaction Nx>
    output:<客戶 ID,Mapped-transaction Mx>
    DistributedCache-file: Mapped-Table MT
    Map-class{
1)      Map-function{
2)          for each Item Ai in Nx
3)              for each Number N in MT
4)                  if(Ai in MT){
5)                      Mapped-transaction+=N;
                    }
6)              Key.set(客戶 ID);
7)              Value.set(Mapped-transaction);
8)              output(Key,value) ;
        }
    }
    input:  <客戶 ID,Mapped-transaction Mx>
    output:<客戶 ID,Mapped-transaction Mx>
    Reduce-class{
1)      Reduce-function{
2)          output(Key,value)
        }
    }
```

圖 5 AprioriAll 平行化虛擬碼：Transformation Phase

```
    input:  <key,Mapped-transaction Mx>
    output:<Candidates-1-sequence Ai ,1>
    Map-class{
1)      Map-function{
2)          foreach item Ai in Mx
3)              Key.set(Ai);
4)              Value.set(1);
5)              output(Key,value)
        }
    }
```

```
    input:<Candidates-1-sequence Ai,list(1,1,1...,1)>
    Output:<Large-1-sequence Ai,suppot of Ai>
    Reduce-class{
1)      Reduce-function{
2)          for each value 1 in list
3)              support +=1;
4)          if(support ≥ minimum_support)
5)              output <Ai,support>
        }
    }
```

圖 6 AprioriAll 平行化虛擬碼：Sequence Phase

第五階段找出 Maximal Sequences，如圖 7，載入 Mapped-transaction，各客戶轉換交易資料分散到各台 Mapper 中，利用圖 6 輸出的 Large-(N-1)-sequence 作為這次的 DistributedCache-file 來產生 Candidates- N-item，並且把出現在此客戶 Candidates-N- sequence 輸出到 reducer 統計並且刪除不符合 min-support 的 Candidates-N-sequence，則符合 min-support 的為 Large-N-sequence，持續重複直到下一階段無任何大型序列為止。

```
    input:<key,Mapped-transaction Mx>
    output:<Candidates-2-sequence Ai ,1>
    DistributedCache-file: Large 1-Sequence
    Map-class{
1)      Map-function{
2)          for each item Xi in Large 1-Sequence
3)              for each item Yi in Large 1-Sequence
4)                  if(比對<Xi,Yi>在 Mx 出現&Xi≠Yi){
5)                      Key.set(<Xi,Yi>);
6)                      Value.set(1);
7)                      output(Key,value);
                    }
        }
    }

    input:<Candidates-2-sequence Ai,1>
    Output:<Large-2-sequence Ai,2 >
    Reduce-class{
1)      Reduce-function{
2)          for each value 1 in list
3)              support +=1;
4)              if(support ≥  minimum_support)
5)                  output <Ai,2>
        }
    }
```

圖 7 AprioriAll 平行化虛擬碼：Maximal Sequences

第六階段進行 Reduction Large Itemsets，如圖 8，分別把 Large-N-sequence，資料載入 Mapper，並設 DistributedCache-file 為 Mapped-Table 做為參考，並將對應編號還原成 ItemSet 輸出。

```
    input:  <Key,Large-N-sequence LN>
    Output:<N,Back-transaction>
    DistributedCache-file: Mapped-Table MT
    Map-class{
1)      Map-function{
2)          for each ItemSet-Number Ii in LN
```

```
3)                foreach Mapped-item MIT in MT
4)                  if(I_i in MT){
5)                      Back-transaction+=MIT;
                  }
6)                Key.set(N);
7)                Value.set(Back-transaction);
8)                output(Key,value);
        }
    }
    input:<N,Back-transaction>
    output:<N,Back-transaction>
    Reduce-class{
1)      Reduce-function{
2)          output(Key,value);
        }
    }
```

圖 8 AprioriAll 平行化虛擬碼：Reduction Large Itemsets

3.2 平行化 GSP

GSP平行化虛擬碼如圖9至圖12所示，我們依據GSP的特性，設計GSP平行化演算法。

```
    input:<key,transaction T_x>
    output:<IntPair T,Item I>
    Map-class{
1)      Map-function{
2)          foreach 一筆交易資料 T in T_x
3)              key.set(T.客戶 ID , T.交易時間);
4)              value.set(T.交易項目);
5)              output(key,value);
        }
    }
    input:< WritableComparable w1,
            WritableComparable w2>
    output:0,1,-1
    GroupingComparator-class{
1)      GroupingComparator-function{
2)          return w1.客戶 ID = = w2.客戶 ID ? 0 :
                (w1.客戶 ID < w2.客戶 ID? -1 : 1);
        }
    }
    input:<IntPair T,Item I>
    output:<1,2,3,....,n>
    Partition-class{
1)      Partition-function{
2)          return (Key.客戶 ID )%(reducer 個數);
        }
    }

    input:<IntPair T,list(I1,I2,I3,....Ix)>
    output:<T.客戶 ID,All-Item of Ix>
    reduce-class{
1)      reduce- function {
2)          Key.set(T.客戶 ID);
3)              for each Item Ii in list
4)                  All-Item+=Item;
5)              value.set(All-Item);
6)              output<Key,All-Item>;
        }
    }
```

圖 9 GSP 平行化虛擬碼：Sort Phase

第二階段是Itemsets Phase，如圖10，以New-transaction作為輸入，把資料分配到不同的Mapper，再分開搜尋客戶交易順序，輸出客戶交易序列資料表後，再透過Reducer加總計算支持個數，找出滿足最小支持度的大型項目集(Large-1-itemSet)。

```
    input:<key,New-transaction N_x>
    output:<Candidates-1-itemSet A_i ,1>
    Map-class{
1)      Map-function{
2)          foreach 1-itemSet A_i in N_x
3)              Key.set(A_i);
4)              Value.set(1);
5)              output(Key,value)
        }
    }

    input:<Candidates-1-itemSet,A_i list(1,1,1,...,1)>
    Output:<Large-1-itemSet A_i,suppot of A_i>
    Reduce-class{
1)      Reduce-function{
2)          for each value 1 in list
3)              support+=1;
4)          if(support>=minimum_support)
5)              output <A_i,support>
        }
    }
```

圖 10 GSP 平行化虛擬碼：Itemsets Phase

第三階段是 large 2-Sequence，如圖 11，以New-transtion 作為輸入，Large 1-itemSet 設為DistributedCache-file，mapper function 搜尋並比較判斷客戶交易時間 Max-Gap、Min-Gap、最大時間限制，輸出 Candidates-2-ItemSet，再透過 Reducer 加總計算支持個數，找出滿足最小支持度的Candidates-2-ItemSet 為 large 2-Sequence。

第四階段是 large 3-Sequence，如圖 12，以New-transaction 作為輸入，分散到各 mapper 之後，並把上一次輸出的結果作為此次的DistributedCache-file，找尋 Candidates-M-ItemSet，輸出至 reducer function 做 join & pruning 階段，符合 min-support 的 Candidates-M-ItemSet 做為lager-M-ItemSet，並為下次的 DistributedCache-file。

```
    input:<key,New-transaction N_x>
    output:<Candidates-2-ItemSet A_i ,1>
    DistributedCache-file: Large 1-itemSet
    global variables 最大時間限制;
    global variables Max-Gap;
    global variables Min-Gap;
    Map-class{
1)      Map-function{
2)          for each item X_i in Large 1-Sequence
3)              for each item Y_i in Large 1-Sequence
4)                  if(N_x 資料比對 X_i 和 Y_i 存在){
5)                      if(Min-Gap ≤ Y_i 與 X_i 交易時間差
                            ≤Max-Gap){
6)                          if(Y_i. 與 X_i.交易時間差
```

```
                    ≤最大時間限制){
7)                      Key.set([Xi,Yi]);
                    }
8)                  else{
9)                      Key.set([Xi],[Yi]);
                    }
10)                 Value.set(1);
11)                 output(Key,value);
                }
            }
        }
    }
    input:<Candidates-2-ItemSet Ai, list(1,1,1,...,1)>
    Output:<Large-2-ItemSet Ai,suppot of Ai>
    Reduce-class{
1)      Reduce-function{
2)          for each value 1 in list
3)              support +=1;
4)          if(support ≥ minimum_support)
5)              output <Ai,support>
        }
    }
```

圖 11 GSP 平行化虛擬碼：large 2-Sequence

```
    input:<key,New-transaction Nx>
    output:<Candidates-3-ItemSet Ai ,1>
    DistributedCache-file: Large-2-itemSet
    global variables 最大時間限制;
    global variables Max-Gap;
    global variables Min-Gap;
    Map-class{
1)      Map-function{
2)          for each item Xi in Large 1-Sequence
3)              for each item Yi in Large 1-Sequence
4)                  if(Nx 資料比對 Xi 和 Yi 存在
                    &Xi.last = =Yi.first){
5)                      if(Yi.last.與 Xi.first.交易時間差
                        ≤Max-Gap&Yi與 Xi 交易時間差
                        ≥Min-Gap){
6)                          if(Xi 獨立項目集&Yi 獨立項目集){
7)                              Key.set("[Xi.first],[Yi.first]
                                ,[Yi.last]");
                            }
8)                          else if (Xi 為獨立項目集&
                                Yi 為非獨立項目集){
9)                              Key.set("[Xi.first],[Yi.first
                                ,Yi.last]");
                            }
10)                         else if(Xi 為非獨立項目集&
                                Yi 為獨立項目集){
11)                             Key.set("[Xi.first,Yi.first]
                                ,[Yi.last]");
                            }
12)                         else if(Xi 為非獨立項目集&
                                Yi 為非獨立項目集&
                                Yi.last 與 Xi.first 交易時間差≤
                                最大時間限制){
13)                             Key.set("[Xi.first,Yi.first
                                ,Yi.last]");
                            }
14)                         Value.set(1);
15)                         output(Key,value);
                    }
```

```
                }
            }
        }
        input:<Candidates-3-ItemSet Ai, list(1,1,1,...,1)>
        Output:<Large-3-ItemSet Ai,suppot of Ai>
        Reduce-class{
1)          Reduce-function{
2)              for each value 1 in list
3)                  support +=1;
4)              if(support ≥ minimum_support)
5)                  output <Ai,support>
            }
        }
```

圖 12. GSP 平行化虛擬碼：large 3-Sequence

4.實驗結果

本研究針對 AprioriAll 演算法與 GSP 演算法，量測二者演算法於平行化下，執行效能實驗測試。設計方式針對交易資料筆數多寡與最低支持度數據設定進行效能實驗。實際測驗環境與設備，硬體設備結合4台電腦進行效能測試，其中1台為Master主機，其餘3台為 Slaves 機器(如表 1)。

表 1 實驗環境與硬體設備

Server 型號	Dell PowerEdge 1950
CPU	Intel® Xeon® Processor E5420 2.5 GHz *2
Cores	4
RAM	8G DDR2 1066
Host OS	VMWare ESXI 5.1
Guest OS	Ubuntu 12.04.2 32bit
HDD	146G SAS 15K 轉
數量	4(1 台 Master 3 台 Slaves)
Hadoop 版本	0.20.203.0rc1

本實驗為排除 GSP 演算法的時間限制，方能與 AprioriAll 演算法一同進行實驗，所以將 GSP 演算法參數設定為 ws=0，min-gap=0 且 max-gsp=∞，進行實驗效能比較。

4.1 最低支持度數據效能測試

本實驗目的在於實測 MapReduce 資料平行應用架構下，AprioriAll 演算法與 GSP 演算法各別設定下的最低支持度所需的執行時間，資料庫數據約為 50,000 筆交易量，平均長度約 10 個交易項目，X 軸代表最低支持度(%)，Y 軸代表總執行時間(s)，最低支持度數據分別為 5%、10%、15%、20%進行實驗測試之效能分析。

Apriori All 於最低支持度 5%執行時間約 2023sec，最低支持度 10%執行時間約 1777sec，最低支持度 15%執行時間約 1799sec，最低支持度 20%執行時間約 1813sec。

GSP 於最低支持度 5%執行時間約 360sec，最低支持度 10%執行時間約 334sec、最低支持度 15%執行時間約 213sec，最低支持度 20%執行時間約 217sec，如圖 13。

經實驗測試 GSP 演算法執行效能優於 Apriori

All 演算法，最顯著差異在於設定最低支持度為 5% 時，GSP 演算法所執行的時間效能最明顯優於 Apriori All 演算法。

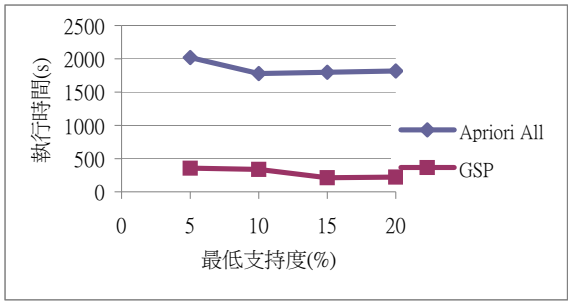

圖 13 交易量 50,000 筆最低支持度執行時間

4.2 資料庫交易量效能測試

本實驗目的測試資料庫交易量分別為 5 萬筆、10 萬筆、50 萬筆、100 萬筆進行實驗測試之效能分析所需的執行時間，資料平均長度約 10 個交易項目，X 軸代表交易資料量，Y 軸代表總執行時間(s)。

Apriori All 於資料量 5 萬筆執行時間約 1278sec，資料量 10 萬筆執行時間約 1638sec，資料量 50 萬筆執行時間約 3899sec，資料量 100 萬筆執行時間約 7189sec。

GSP 於資料量 5 萬筆執行時間約 276sec，資料量 10 萬筆執行時間約 340sec，資料量 50 萬筆執行時間約 906sec，資料量 100 萬筆執行時間約 1637sec，如圖 14。

經不同交易資料量筆數實驗測試，GSP 演算法效能執行上也優於 Apriori All 演算法，最顯著差異在於設定交易筆數為 100 萬筆時，GSP 演算法所執行的時間效能最明顯優於 Apriori All 演算法。

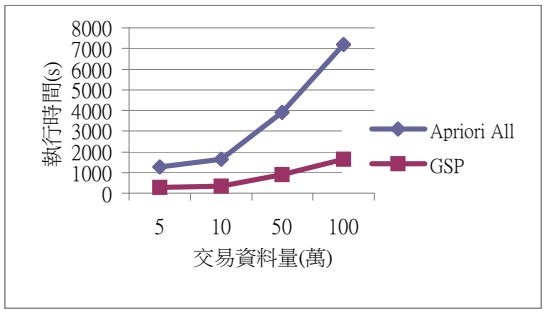

圖 14 交易量執行時間

4.3 GSP 參數設定效能測試

本實驗目的在於實測 GSP 演算法交易時間參數所需的執行時間，資料平均長度約 10 個交易項目，X 軸代表交易單位時間，Y 軸代表總執行時間(s)，測試交易單位時間分別為 100sec、200sec、500sec、1000sec、10000sec 進行分析。

GSP 於 100sec 執行時間，ws 約 343sec、mas 約 205sec、min 約 213sec，200sec 執行時間，ws 約 342sec、mas 約 214sec、min 約 209sec，500sec 執行時間，ws 約 351sec、mas 約 204 sec、min 約 215 sec，1000sec 執行時間，ws 約 358sec、mas 約 210sec、min 約 208 sec，10000sec 執行時間，ws 約 1030sec、mas 約 340 sec、min 約 209sec，如圖 15。

經實驗測試 GSP 演算法設定交易時間長短，當交易時間越長相對交易筆數量也增加的情況下，GSP 演算法所執行的時間效能也會明顯增加。

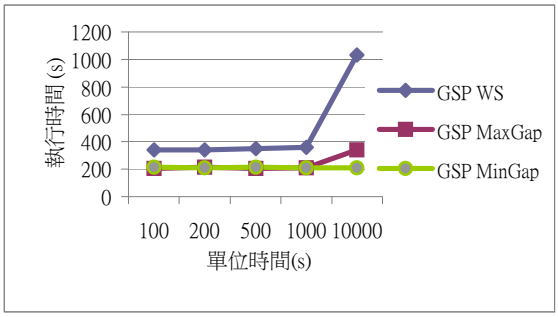

圖 15 GSP 參數執行時間

5. 結論

本研究的目的分析不同類型探勘演算法之關係，採用循序探勘模式中 AprioriAll 演算法與 GSP 演算法，透過 MapReduce 資料平行應用架構，有效處理大型資料庫並縮短執行時間。實驗結果顯示於 MapReduce 資料平行應用架構下 GSP 演算法處理大型資料效能上優於 AprioriAll 演算法。

參考文獻

[1] Agrawal, R. and Srikant, R. “Mining Sequential Patterns,” In Proc. *Conf. Data Engineering (ICDE’95)*, Taipei, Taiwan, Mar. 1995, pp. 3–145.

[2] R. Agrawal and R. Srikant, “Fast Algorithm for Mining Association Rules in Large Database,” In Proceeding of the 20th International Conference on VLDB, Pages 487-499, 1994.

[3] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami, “ Mining association rules between sets of items in large database,”Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993.

[4] Rakesh Agrawal, Ramakrishnan Srikant, “ Mining Sequential Patterns,” Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan.

[5] Srikant, R. and Agrawal, R., “Mining sequential patterns: Generalizations and performance improvements,” *In 5th Intl. Conf. Extending Database Technology*, March 1996,pp.3-17.

[6] 曾憲雄，蔡秀滿，蘇東興，曾秋蓉和王慶堯著(2005)，資料探勘 Data Mining，台北：旗標出版股份有限公司。