國立臺灣大學管理學院資訊管理學研究所

碩士論文

Graduate Institute of Information Management

College of Management

National Taiwan University

Master Thesis

巨量網路資料之互動式分析系統

An Interactive Security Analysis System of Large Scale NetFlow Data

周振澔

Zhen-Hou Zhou

指導教授：孫雅麗 博士

Advisor: Yeali S. Sun, Ph.D.

中華民國 104 年 8 月

Aug, 2015

國立臺灣大學(碩、博)士學位論文
口試委員會審定書

巨量網路資料之互動式安全分析系統
An Interactive Security Analysis System of Large Scale NetFlow Data

本論文係周振澔君（學號 R02725012）在國立臺灣大學資訊管理學系、所完成之（博、碩）士學位論文，於民國 104 年 7 月 23 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

陳孟彰　　張雅惠

潘音辉　　洪士灝

所　　長：　彭文志

誌謝

從一開始懵懵懂懂的加入實驗室開始，到現今即將畢業，首先要感謝的是父母在背後無怨無悔支持，在疲倦的時候讓我總是有個溫暖的家可以休息、充電；實驗室裡很猛的 Mike 學長在有問題的時後隨時可以罩我；感謝杜承恩學長對於 Proposal 所給予的大方向、戴瑋如和曾雅敏學姊在系統交接時所給予的指導還有吳張民學長對於資料前處理的支援都讓我在研究的路上能夠順利走下去；此外下一屆的學弟妹們：施黛玲、林劭軒、李士暄、李奕德與姜立垣的加入讓實驗室多了許多活力，每一位學弟妹都很給力，老師也都對妳們很滿意，以後實驗室就交給你們了。

還有要感謝的是同一屆的好基友們：Weal Lab 的各位。施舜元、李孟兩個明明很強在那邊裝弱，施舜元的早午餐外送服務讓我可以安心的宅；李孟的楓之谷嚮導與 ARAM 的專業凱瑞服務，讓我在寫論文辛苦的時候能夠安心拿首勝；還要感謝陳世穎長期以來讓我占用位子，可以不用孤獨守在實驗室中；感謝楊智幃的司機與果汁機服務，雅美時光與蘋果牛奶總是讓我難以忘懷；陳靖甯三不五時的貓咪影片，總是令人不禁莞爾一笑。在碩二的時候一路陪伴著我、支持著我的楊子嫻，沒有你的鼓勵我絕對是沒有辦法走到最後的，非常非常感謝你，也感謝上天讓我遇見了你。

一路跌跌撞撞到了現在，也終於要結束學生生涯，最後則是要感謝路上遇見的你與妳，你們的出現讓我的人生歷程中劃上精彩的一筆，我是絕對不會忘記你(妳)們的。

中文摘要

現今網路流量已以往無法想像的速度成長；網路犯罪亦隱身在龐大的網路流量中。為協助資安人員快速且有效率地為在網路流量中找出可做為呈堂供證的通聯記錄，我們提出了將網路流量視覺化的互動式查詢系統－NetActy。在本論文中對 NetActy 的互動性以及視覺化過程進行改進，藉由考慮節點間工作量的平衡以及 Data Locailty，目的為了使計算節點執行時間平衡以達到互動程度的回應時間。本論文將工作量分配制定成一個 Linear Programming 問題，並提出經驗解－Algorithm 1 以期在多項式時間內解決；視覺化部分，我們為每個查詢視圖做快取以及利用 Multicast 技術來加速處理。最後於實驗中，我們衡量 Algorithm 1 的效能確認其能夠在不違背 Data Locality 的情況下平衡節點間工作量；此外在視覺化部分所遇到的問題我們亦參考現行作業系統的做法來解決。

關鍵字：資安犯罪偵查、互動式查詢、大數據、工作分配、資料在地化、流量視覺化

ABSTRACT

As the network volume grows rapidly, network crimes can hide behind the huge network traffic. In order to let IT security people find evidences fastly and effectively from such a huge network traffic, we proposed a interactive, visualable network query system－NetActy. In this thesis, we improve the interactivity and visualization process, by takeing the balance between workload and data locality into consider. We formulate the job assignment problem into a Linear Programming problem and solve it by a heuristic solution－Algorithm 1. In the last, we evaluate the performance of Algorithm 1 and make sure that Algorithm 1 can actually balance the workload without violating data locality. Besides, we solve the problem encountered in visualization part by applying current OS's solution.

Keyword：Network security forensics, Interactive query system, Big data, Job assignment, Data locality, NetFlow records visualization.

目錄

圖目錄

表目錄

第一章　介紹

第一節　研究背景

現今網際網路產生流量總和已遠遠超越以往，在[1]中表示 2013 年每人每年平均產生的流量是 2005 年的約 20 倍，到了 2018 年更是會成長到 2013 年的三倍；在 2016 年底，網際網路的流量會來到 Zeta 等級(1 Giga Trillion)；近年行動裝置的普及，在 2016 年其產生的流量正式超過有線裝置；此外隨著物連網的發展(Internet of Thing, Internet of Everything)，在 2018 年有過半的網路流量由非 PC 裝置所產生。

除了流量的增長之外，自 2004 年以來社群媒體網站的蓬勃發展，使得網路上的內容變異性也大大增加。這樣的變異性讓資安調查人員在分析封包內容時遭遇不小的困境，就算我們能夠紀錄再多流量，無法解讀的內容也派不上用場。

網路資料的價值隨著網路服務的豐富度水漲船高，導致資安攻擊事件層出不窮，駭客盜取伺服器的資料之後於暗網(DarkNet)販售借以獲利。故現今駭客不同以往高調的攻擊作法，反以偷偷摸摸在受害者的機器架設後門程式，慢慢把資料給偷走。而且駭客也不再是漫無目的的攻擊，漸漸的他們會針對那些組織中高層人員進行持續性滲透攻擊(Advanced Persistent Threat)，滲透進入他們的電腦，找出可茲利用的資訊之後再隱匿自己的行蹤。

第二節　研究問題

綜合以上的狀況，資安人員在調查追究資安攻擊的責任歸屬愈發地困難。因為日常流量蓬勃發展導致儲存裝置數量跟不上增加速度，一旦我們要在這些資料中搜尋可疑資訊的話，不僅要面對大量的資料之外還要能夠建構出符合資安偵察具快速處理大量資料的分析應用程式，以萃取出想達到的資料與資訊。故如何”快速”且”有效率”的從”非常大量”的日常網路資料中找出可作為呈堂供證的資料證據是目前資安調查中一個急需解決的問題。

第三節　研究貢獻

在先前的研究中[2, 3]，我們提出一個網路流量視覺化暨互動查詢系統－NetActy，這個系統的設計目的是提供資安調查人員一個可視覺化、相似於傳統資料庫 SQL 語言(SQL-like) 的互動介面，使用者可利用 NetActy 針對其管理網域(Managed Network)與外部網路的通訊資料所收集的 Netflow 下指令，查詢任何想要找尋的通訊資料。透過網頁介面下達包含時間與空間範圍的 Query，NetActy 把網路流量呈現在地圖上，將龐大的網路流量資料以快速的處理找到欲搜尋資料再以視覺化的方式呈現，提供資安調查人員較直觀的方法來辨識可疑流量，以利資安調查與偵防。圖 1 為所設計的 NetActy 查詢介面之一。

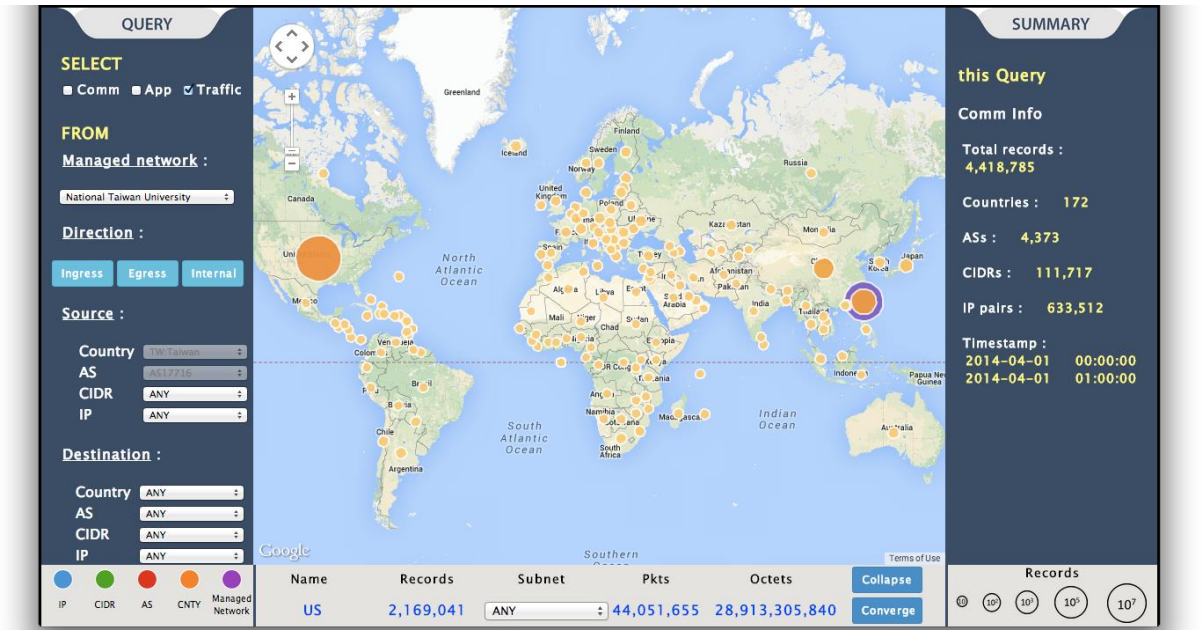


圖 1、NetActy 使用介面圖

在 NetActy 系統查詢介面中，我們將畫面切分為上下兩部分，下半部設計用來呈現說明圖例，例如：各網路層級所代表的顏色，以及關於目前選定目標的流量資訊：該目標包含的 NetFlow Record 數量、全部傳輸封包數量和傳輸總資料量等。這部分資訊會隨著使用者點選不同目標而有所變動，因此我們將這部分放在地圖的正下方，讓使用者明顯的看到資訊變化。

上半部可以分為左、中、右三大塊。根據人們流覽網頁的習慣都是從上到下、左到右，因此我們將下達 Query 的功能放在左上區塊，使用者可以在最快的時間內找到此功能；中間區塊透過保留大部分面積以呈現地圖，讓使用者能對呈現在地圖上的流量資料一目瞭然，而不需要額外的拖曳才能了解全貌；最右邊則呈現關於當前 Query 的統計資訊：像是總共處理幾筆 NetFlow Records、有多少國家、AS、CIDR 與管理網

域有聯繫、在這些聯絡中有多少不重複的 IP Pairs 等。因為這些統計資訊只跟 Query 有觀而不會隨著使用者瀏覽目標變動而改變，所以放在最右邊。

3.1 NetActy 儲存系統

為了儲存大量的 Netflow 網路流量資料，不可能只將資料儲存於一台機器上，我們假設使用 2003 年 Google massively parallel 巨量資料處理的架構，也就是將一群 commodity computers 組成叢集，利用這些分散機器的 local 儲存系統來共同存放這些流量資料。NetActy 假設這些機器叢集的 local 儲存系統建構為一個 HDFS[4] (Hadoop Distributed File System)。其中，資料的儲存政策是不僅能儲存原資料外，還擁有備份、Load Balance 等功能。在資料儲存上，因為考量未來資料檢索與搜尋的快速效能，以及 Netflow 網路流量資料內容可能有些欄位資料多有重複，因此，NetActy 並不直接儲存由路由器 (Router)所記錄下 NetFlow Records，我們對 NetFlow Records 做一連串的前處理 (pre-processing)。資料前處理有以下幾個目標：

(1) NetFlow Records 並沒有地理區域的資訊，像是 CIDR、AS 等等。因此我們利用在[2]中提出的工具－toolNetData，額外為每一筆 Record 補足缺少的資訊。

(2) 為了減少處理 Query 所需要查詢的資料量，NetActy 採用 Column Storage 策略[5]：將一筆 NetFlow Record 依照資訊類型，切分成 Application Segment, Traffic Segment 與 Communication Segment，NetActy 將依照使用者需求查詢不同 Segment，以減少讀取資訊數量來加快本系統，達到互動式 Query 的目的。

(3) NetActy 在加速針對空間與時間範的查詢上，為資料儲存方式做了特別的設計，採用階層式路徑的方法，將網路流量的地理區域與時間作為資料夾路徑 (directory path)，以利 Query 可以快速讀取相對應 Query 所欲查詢的網路流量資料。比如說，現在的 Query 想要查詢 2014 年 4 月 1 號所管理的網域對美國 (US)下某個 AS 的網路流量狀況，系統在接到這個查詢時可以直接在檔案目錄 US/ASxxxx/2014-04-01/CommunicationSegment0.txt 檔案中作搜尋，不需要花費額外的時間搜尋相關檔案所在位置。圖二為前處理的架構圖。

Preprocessing 從路由器收集 NetFlow 原始資料開始，由圖中左上角開始進行一連串處理。路由器每隔 20 分鐘將紀錄的 NetFlow Record 寫入 Local Disk 中，而 Precprocess 一次處理一天的流量，所以總共有 72 個由路由器所儲存的檔案。把這些檔案放進 HDFS 中接著利用 MapReduce 方式進行 Preprocessing，透過大量機器同時處理以冀提早完成。

首先我們補足地理資訊：toolNetData 掃過原始資料內每一筆 Reocrd 為其加入國家、AS 與 CIDR 資訊；接著依欄位把 Record 資訊分別寫入三個 Segments 中，寫入同時進行 Running Length Encoding 為 Segments 做壓縮：對於重複頻率較高的的欄位，諸如：CIDR、Port number 等，我們不直接紀錄該筆 Record 的原始資訊而是紀錄該資訊重複的內容以及重複的次數。以 Port number 為例，若是該筆資料以上一筆的 Port number 同樣是 80，那麼我們就紀錄 ”80 2”表示有兩筆 Recodr 的 Port 為 80。

Segment 會持續寫入 NetFlow Records 直到大小接近設定的門檻 (在 NetActy 中，設定為 63MB)，一但大小接近門檻便寫入 HDFS。Segments 會依據通聯的地理區域放到對應路徑的資料夾內，當所有檔案寫入 HDFS 後就完成 Preprocess 過程。

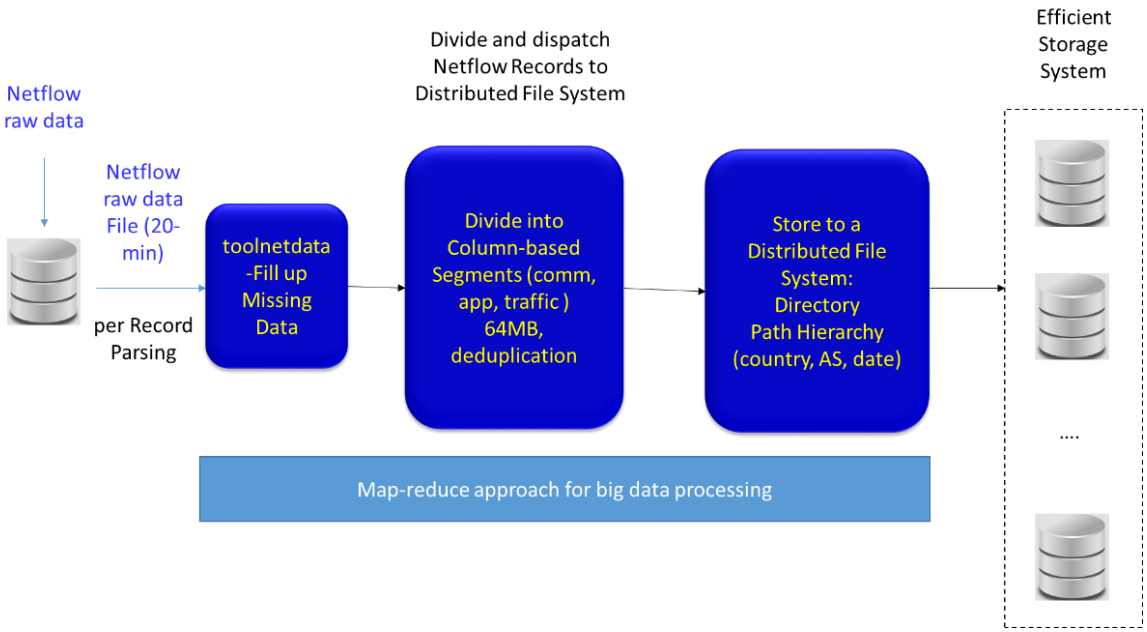


圖 2、NetActy 資料前處理流程架構圖 BigIP Render Tree

在 NetActy 系統中，為方便使用者在視覺化的互動介面可以在這個查詢 (This Query)下任意遍尋 (Traverse)不同地理區域規模的網路通訊與流量資料，NetActy 將每一筆 NetFlow Record 的地理區域劃分為四個等級：國家 (Country)、AS (Autonomous System)、CIDR (Classless Inter-Domain Routing)、IP。每筆 NetFlow Record 一定會屬於某個國家、AS 以及 CIDR 轄下；而每個國家會包含一到多個 AS；每個 AS 也可能包含一到多個 CIDR。這種由上到下的隸屬關係可以用樹狀結構來表示。我們把所有在這次 Query 下所搜尋到的相關 NetFlow Records 資料建立成一顆由許多國家、AS 等所組成的樹結構，我們稱之為 BRT (BigIP Rendering Tree)，如圖 3 所示。

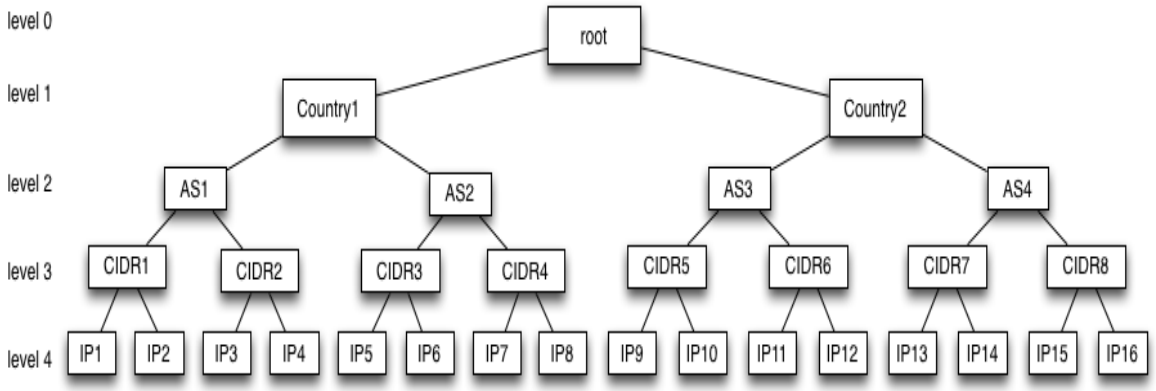


圖 3、BigIP Rendering Tree (BRT) 示意圖

透過 BRT 的建立，我們能夠快速找到相關的網路通訊與流量資料以回應使用者在互動介面上遍尋 (Traverse)不同地理區域規模的需求。當使用者點擊不同地理區域作資料查詢時，NetActy 所要做的就只是找出相對應的子樹(Sub Tree)資料再呈現在介面地圖上即可，只需付出極少的處理成本，因為這個查詢下的一切資訊在一開始就已經都收集整理在 BRT trees。

3.2 NetActy 系統執行架構

為了達到互動查詢 (Interactive Query)的回應時間，NetActy 採取分散式的巨量資料處理架構與方法：利用群的處理叢集在所有儲存節點上同時尋找符合 Query 的網路流量資料檔案並平行處理來能夠減少 Query 處理所需的時間。

NetActy 從收到使用者下達的 Query 到將結果以視覺化地圖方式呈現的執行過程可以分為 5 個步驟：

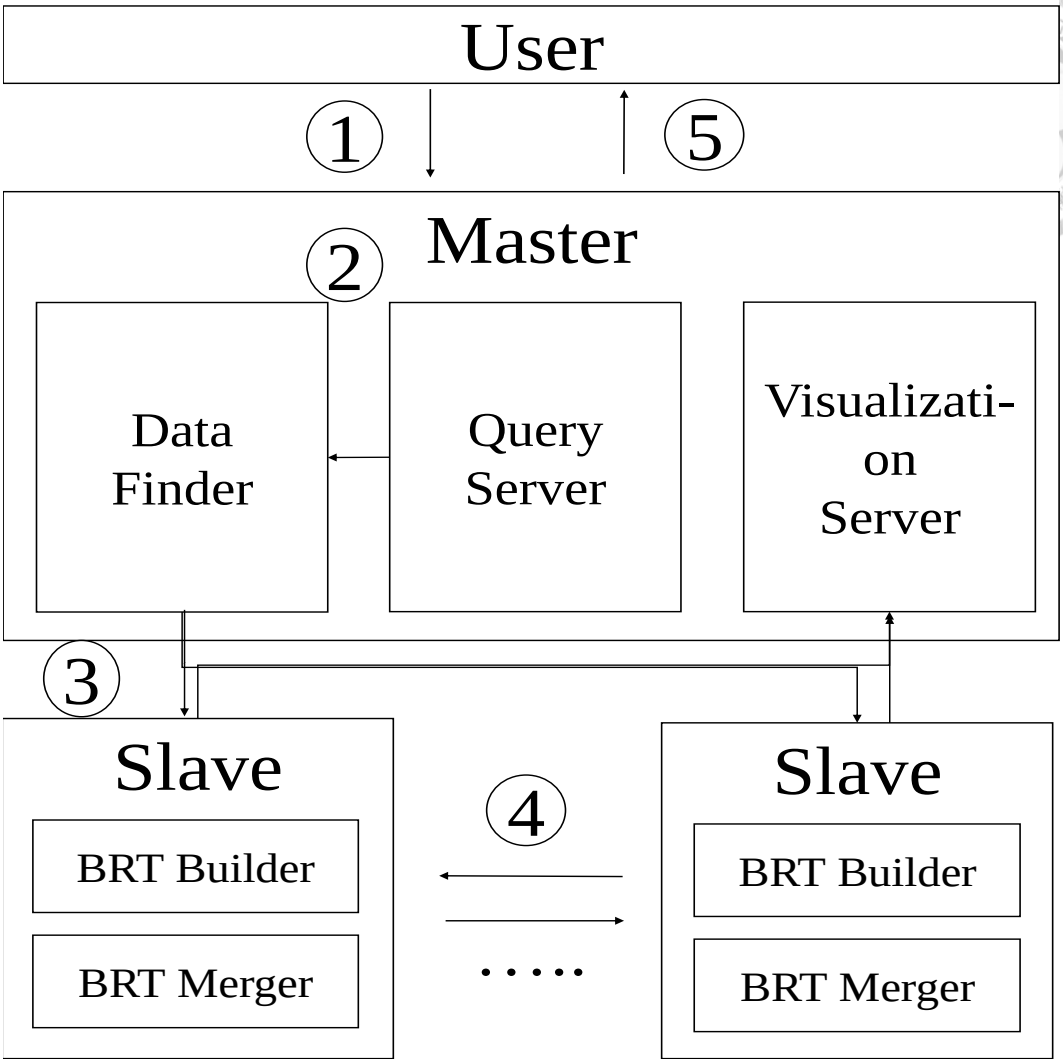


圖 4、NetActy 執行流程圖

(1) Query Server 接收使用者透過網頁所下達的 Query。

(2) Data Finder 根據 Query 從分散式檔案系統中找出需要處理的 Segments。並將這些 Segment 指派給不同節點作處理。

(3) 節點上的 BRT Builder 從 HDFS 當中讀取檔案，並將 Segments 建立成 BRT Partitions。為何稱為 BRT Partition 是因為我們不能保證相同國家的 Segments 都存在相同節點（Segment 檔案儲存位置目前是由 HDFS 系統決定），那些擁有 Segments 的計算節點會建立以國家為單位的 BRT Tree，每一國家所屬的 BRT Tree 稱為一 BRT Partition。這步驟將 Segments 分配給資料所在計算節點處理與 MapReduce 巨量資料處理架構中的 Map 步驟相似，因此我們稱此一部分的處理為 Map Phase。我們之前的設計依循 MapReduce 資料處理架構，因為這是中間處理結果，因此得到這些資料後便回儲至各節點的硬碟。

當各國家的 BRT Partitions 都建立完成時，系統會收集各計算節點結果的資料，以國家為單位，指派擁有該國家下最大 BRT Partition 的計算節點接收儲存在其他計算節點下的 BRT Partitions，利用 BRT Merger 模組把 Partitions 合併成一棵 BRT Tree。這個步驟相似於 Reduce Phase。

(4) 將屬於各個國家的 BRT Trees 轉換 (rendering)為可以被呈現的 HTML code，回傳給 Visualization Server。

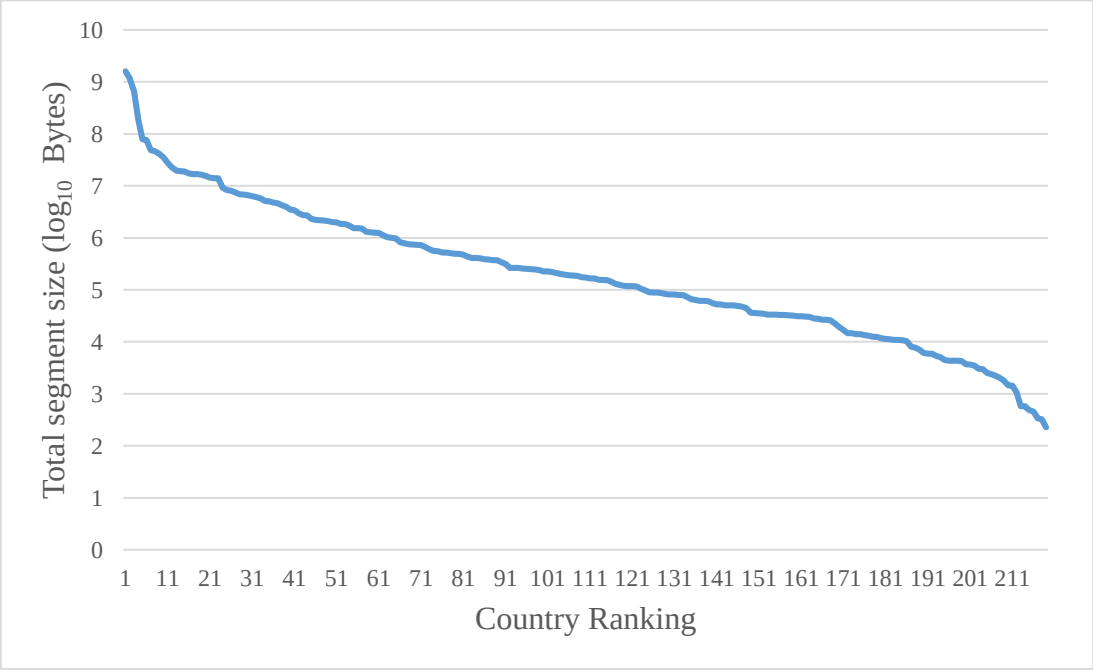
(5) Visualization Server 將各國家 HTML code 合併後，讓資料在網頁地圖上顯示，達到巨量資料分析的視覺化呈現。

3.3 NetActy 的效能問題

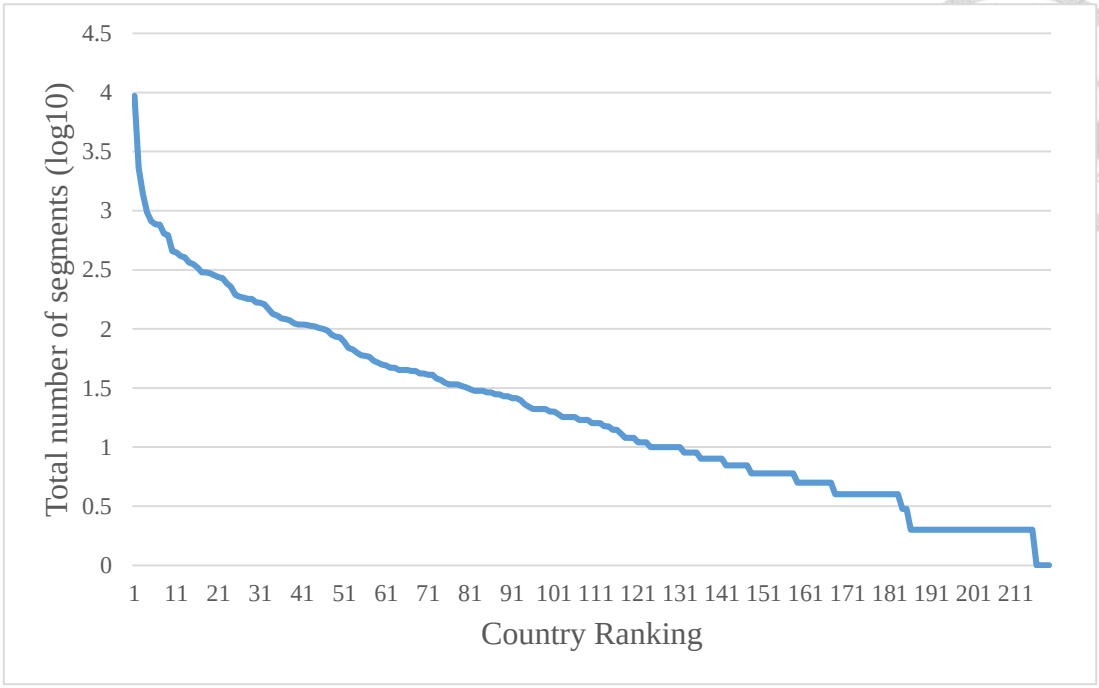
以上所述的 NetActy 系統，在實作上我們發現有幾項缺點必須改進。

第一、在 Map Phase 中間產生的 BRT Partition 資料結果寫回硬碟會花費不少時間，增加 interactive response time。追究其原因為某些國家所擁有的檔案數量過多，造成 BRT Partition 數量過多，導致寫入硬碟時產生額外的 Overhead。

在這個網路流量資料的巨量分析裡，我們發現根據之前所設計的資料儲存方法，有幾個特性。第一、以 Country-AS 為分類條件下的 Segment 檔案數與國家間存在遵循 Zipf distribution。以本實作系統的應用資料為例，對外的通訊國家有 219，以總資料量以及檔案個數的關係如圖 5 所示，大略是集中在前四個國家：美國、台灣、中國、日本 (依名次排序)。



(a) 將一天內網路流量依各國家擁有待處理檔案總量作排序(s = 2.95)



(b) 將一天內網路流量依各國家擁有待處理檔案總數作排序 (s = 1.77)

圖 5、一天內網路流量各國家擁有資料總量與檔案個數

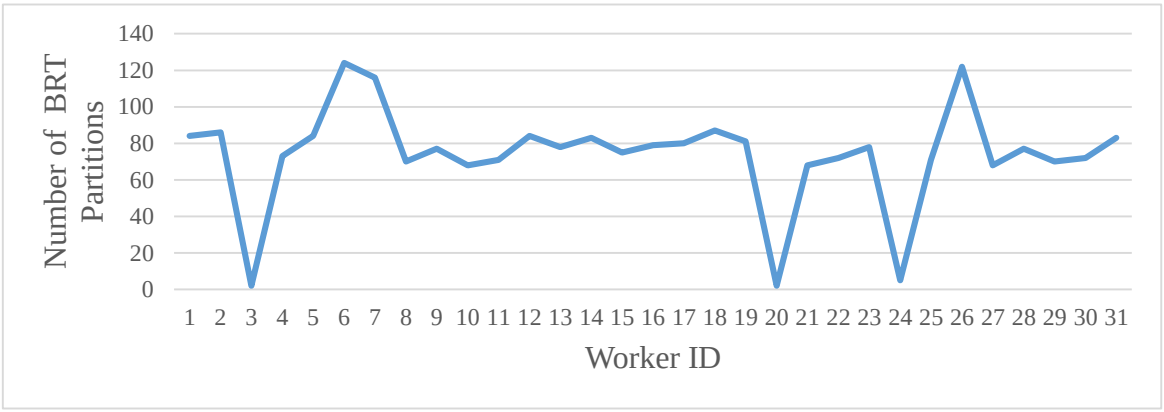


圖 6、對一天網路流量查詢中，各計算節點於 Map task 產生的所有 BRT Partition 數量 (Total : 2290)

此外，如圖 5(a)所示，其餘的超過 200 名的國家，在一整天內的通訊資料量都極少。因此，在 Map phase 處理完後，每一個 Map task 計算節點產生產生的 BRT Partition 數很多如圖 6 所示。當這些檔案全部寫回計算節點的本地硬碟，在 Reduce Phase 透過網路傳送到負責彙整該國家的計算節點，會花費大量資料讀取與傳送的時間。根據[6]，當時也遭遇到在某些應用所處理的資料例如計算資料裡關鍵字出現的次數時，因為關鍵字次數具有 Zipf distribution 的特性，每一個 Map task 幾乎都有成千上萬的資料檔案有相同熱門關鍵字出現，而這些在這個步驟所計算出的次數資料經由網路傳到下一階段負責該關鍵字次數加總的 Reduce task。

為了要避免大量中間所產生資料網路傳輸成本以及不平均的工作量分配於所有的計算節點上，該論文提出一個解決方式是如果 Reduce function 具備 commutative 和 associative 特性，就建議使用一個叫做 Combiner 的功能：它將部分本地的資料先進行彙整，再透過網路傳輸。這個功能的執行是在第一階段的 Map phase 執行，通常是整合到 map task 。利用 Combiner 的功能可以有效加快某些應用的處理時間。在本論文所提出的巨量 Netflow 資料的處理，我們採用的是考量資料在不同國家與網域的資料通訊特性，緊湊(compact)儲存方式的設計以及利用不寫回硬碟而以記憶體儲存的方式來加速處理時間。

第二，在步驟 2 分配 Segment 處理時並沒有考慮到工作量的平衡，尤其是在一個查詢指令下所涵蓋搜尋的資料量大時，會有計算節點間，就是有幾個計算節點的完成時間相當地長，甚至有沒有回應的情形。這個狀況也是本論文的研究議題之一。我們提出一個方法，考量能夠依照每個計算節點不同的處理能力給予不同的工作量分配，以達到所有的計算節點都能在差不多的時間內完成，讓節點工作量平衡來避免一或多個慢 (slow)計算節點，拖累整個工作完成所需的執行時間。

第三，原先的系統設計在分配工作時並沒有考慮到待處理資料所在的位置，這會造成在決定工作分配後，計算處理開始前，資料必須要複製傳送到被分派處理的計算節點。如果僅以工作量作為分配的依據，則導致額外的 I/O 與 Network overheads。在本論文中，我們考慮資料本地化 (data locality)，也就是只考慮任意計算節點一定是處理儲存在其本地硬碟的資料，以避免不必要的 I/O 與 Network overheads。

第四，在步驟 3 中由 NetActy 系統產生的中間結果是以檔案的形式儲存在硬碟中，接著以 FTP 在節點之間交換。這樣的執行過程會在造成可觀的 I/O overhead。若是能將所有中間結果也就是 BRT Partition 甚至是最後結果 (BRT)存放在 Memory，不僅因為不需要再花額外 effort 從硬碟讀取到 Memory 而加速處理過程亦能加快節點之間交換 BRT Partition 的速度。

在第二章我們對 Distributed Parallel Processing 和 Interactive Query Framework 設計收集相關文獻並比較其與本論文所提的方法比較。第三章針對在考量計算節點計算能力下，如何做工作量平均分配以達到所有計算節點都能在最小化完成時間之差異下完成。我們提出了工作分配演算法，解決工作量不平均可能導致執行時間被少數節點拖長的問題，同時利用幾個實驗設計，評估所提方法的效益。第四章針對 Reduce Phase 進行改進，第五章為結論。

第二章　文獻探討

第一節　MapReduce

A. 設計目的：

至 Google 創立以來其搜尋引擎以其快速與精準的結果出名，為了提供高品質搜尋服務每天 Google 得到處爬下網頁後儲存進資料庫，接著做後續 Text Mining 處理。Google 服務地區不僅美國，也在歐洲、亞洲等地擁有大量用戶，也就是說 Google 得從這些地區的網站上爬下當天更新的內容後儲存。這些 TB 等級的更新內容，不僅難以儲存處理上也相當棘手，因為僅僅使用少數的機器無法在短時間內處理完這麼大量的資料，所以 Google 想到利用大量機器共同儲存與平行作 Text Mining 處理就能夠在可接受的時間範圍內完成。

B. 設計目標：

不只是每日網頁資料處理可以利用大量機器共同儲存與加速處理完成，在 Google 內有很多業務也有相同的資源使用模式，因此 Google 就提出一個 General Purpose 的分散式儲存系統與平行化程式的 Programming Framework－GFS [7](Google File System)與 MapReduce[6]，讓 Google 內部同仁能撰寫 MapReduce 程式以共同使用相同 Cluster 完成批量工作進而提高 Cluster Utilization。

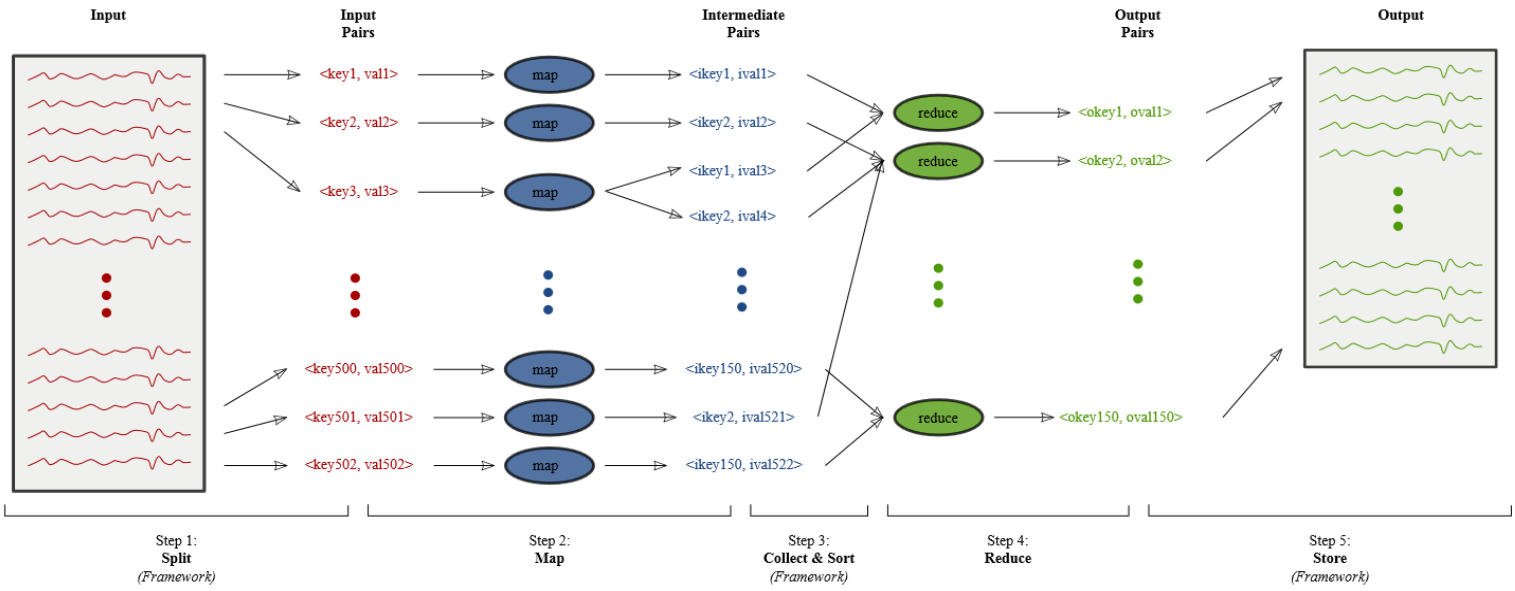


圖 7、MapReduce 執行流程圖

C. 設計細節：

MapReduce 執行過程分為五個步驟：Split、Map、Collect & Sort、Reduce 與 Store。圖中圓形代表著一個 Map 或 Reduce Task，而每台機器會執行不只一個 Tasks 以增加平行化程度 (Parallelism)。

(1) Split：MapReduce 從 GFS 讀取 Input data 傳給 Map Task 處理，這些 Input data 會在儲存進 GFS 時都切割成一塊一塊、相同大小的 Data Block，讓 Map Task 工作量都差不多以期擁有相同的執行時間

(2) Map：在此步驟，Map Task 會將 Input data 的每一行傳進使用者定義的 Map Function 執行。使用者可以自行定義 Map Function 將如何操作 input data 並產生一個或多個 Intermediate Pairs。

(3) Collect & Sort：同一台機器上所有 Map Task 所產生的 Intermediate Pairs 先利用 Key 做排序後接著分配給 Reduce Task 處理。分配的方式有很多種，都是基於 Intermediate pair 的 key 做指派，比如說總共有 10 個 Reduce Tasks 我們可以將 Key 對 10 取餘數後根據結果將該 Pair 傳給負責的 Reduce Task。

(4) Reduce：如同 Map 步驟，每一組 Intermediate pair 都會傳進使用者定義的 Reduce Function 執行。同樣的，Reduce Function 也需要根據 input 產生一組最終的 Output。

(5) Store：同台機器上所有 Reduce Task 產生的 Output 被共同寫到一個檔案中，接著將檔案儲存進 GFS 內。日後使用者可以從 GFS 讀取 MapReduce 的執行結果做使用。

MapReduce 為了讓程式容易撰寫因此設計出只需要五步驟就完成的執行過程，我們稱此特色為：Easy to program。除了這個特色之外，MapReduce 上有下列兩個特色：Fault tolerance 與 Batch Processing。

(1) Easy to program：為了讓其他不擅長撰寫平行化程式的人也可以輕鬆利用大量機器平行處理龐大資料，將執行過程需要人為介入的部分切分為 Map Phase 與 Reduce Phase。這樣做的話，撰寫者就只需要寫 Map Function 與 Reduce Function，而完全不需要知道資料是如何擺放、Cluster 的架設細節等等，就能夠享受 MapReduce 帶來平行化與大量資料處理的好處。

(2) Fault Tolerance：GFS 讓資料可以分散儲存在由大量家用規格機器所組成的 Cluster 上；同時把資料切成固定大小的 Data Block，並將 Block 複製成三份儲存在不同機器上，當有任何機器毀損時還能夠利用其他複本還原資料。在執行時若是有 Task 遺失或處理過程因為軟硬體的緣故而重啟，因為每筆資料都有三份複本，其他機器上的副本便會啟動作為該 Task 的 Backup，並由它繼續完成剩下過程，借由以上機制 MapReduce 能夠擁有 Fault Tolerant 的特性。

(3) Batch Processing：MapReduce 適合處理大量資料、需批次執行工作，因為一旦 Map、Reduce Function 撰寫完成就能不斷重複使用，而且全部執行過程完全不需要人為的介入。比如說：可以設定將每天爬下來的網路資料在固定時間做處理，並將結果自動的寫入 GFS 內，隔天使用者可以在從 GFS 讀取利用。

(4) Outlier Handling：MapReduce 運行在由一群家用規格機器所組成的 Cluster 上，可想而知機器出錯的機率遠遠大於專業等級電腦。當機器有任何的狀況，像是硬碟老舊導致讀取過慢等等往往會造成那台機器當的 Map、Reduce Task 比其他需要更久時間完成，或是幾乎不可能完成。這種狀況平常雖然很少發生但是當 Cluster 內機器數量一多，Outlier 數量也就隨之增加。MapReduce 一旦發現有 Outlier 的存在時，便立刻將那台機器的 Task 在其他擁有複本的機器上重新執行，所以當重新執行的 Task 完成後便可捨棄原本延宕已久的 Task，讓原本進度幾乎停止 MapReduce Job 得以順利完成。

MapReduce 自發表以來，吸引相當大量的關注。他不僅提出了一個簡單的 Programming Framework，讓沒有撰寫平行化程式經驗的使用者也可以使用 Cluster 做平行處理，更重要的是他摒棄以往快速平行處理就需要高效能或是超級電腦的傳統，改用大量的家用規格 PC 組成 Cluster。在 2007 年 Apache 基金會發表開源版本的 MapReduce 與 GFS－Hadoop[8]、HDFS [4](Hadoop Distributed File System)，如此一來一般使用者也可以自行搭建 Cluster 後使用 Hadoop 做平行處理，我們可以說 MapReduce 的發表導致了資料處理的典範轉移，同時也是 Big Data 的濫觴。

第二節　Pregel

A. 設計目的：

隨著社群媒體蓬勃發展，將社群關係以圖像形式處理的需求日漸增高。當圖規模成長到 Billion 等級節點數；Trillion 等級邊數，此時若只靠少數機器不僅難以儲存，就算儲存下來也很難以平行處理。MapReduce 提出作為 General Purpose Programming Framework，雖然可以用以撰寫平行化程式在大規模機器上執行，但圖形處理方面卻是沒辦法擁有最佳的效能。因為圖形計算時節點之間會需要頻繁交換計算訊息，但 MapReduce 僅是將 Map Task 執行結果儲存在 Local Disk 後讓 Reduce Task 讀取並不支援重複的訊息交換。Pregel [9]便是專門設計出來用以分散處理這些大型圖形的 Programming Framework。

B. 設計目標：

首先 Pregel 給予圖形上每個節點一個獨立 ID，接著把這些 ID 丟進 Hash Function 得知該節點應該被分配到哪一台機器上。Hash Function 可以根據使用者的需求自行設計，最簡單的方法是 Modulo Function，對機器數量取餘數將節點分散在不同機器上。節點會連同其 Outgoing Edges 一起分配，機器上所有節點與其 Outgoing Edges 所組成的集合稱為一個 Partition，在接下來運算中每台機器只需要負責他所擁有的 Partition 即可，也就是 Pregel 執行時是有考慮 Data Locality 的。

Pregel 的圖形平行計算是屬於 BSP (Bulk Synchronous Parallel) Model，在這 Model 中整體的計算過程會被切分成許多次的 Supersteps，在每次 Superstep S 中各節點會把上個 Superstep $S-1$所收到的訊息作為這次的 Input State 開始計算，如果有節點滿足終止條件便將狀態改為 Halt。一旦機器上所有節點都計算完成；便將計算結果傳給所有 Outgoing Edges 的 Destination 節點，Supersteps 會在所有節點都處於 Halt 狀態時結束。

當每次 Superstep 結束後，Pregel 要求所有機器將節點的狀態儲存在 Local Disk 中做 Check pointing。當有任何機器當機或 Pregel Process 因故被刪除時就立即停止當前 Superstep；放棄目前節點狀態，讓機器從 Local Disk 中讀取先前記錄的 Check Point，並從該狀態重新執行 Superstep。Pregel 藉由 Check Pointing，來達到 Fault Tolerance。

接著，Pregel 在擁有 300 台機器 Cluster 上用 SSSP (Single Source Shortest Path)演

算法來測量 Pregel 對不同 Worker 數量與 Graph Size 否具有 Scalability，作者進行兩個實驗。首先，在每台機器上執行 50 個 Pregel Worker 慢慢增加到 800 個，執行時間從 174 秒下降到 17.3 秒，使用 16 倍的機器其 Speed Up 為 10 倍。再來，SSSP 的 Input Graph 節點數從 1 Billions 增加到 50 Billions 時，執行時間從 17.3 秒僅增加到 702 秒，雖然 Pregel 執行時間會隨著 Graph 節點數量而增加卻還是能與節點數量保持線性的關係。根據這兩個實驗，Pregel 確實能夠在可接受的時間範圍內處理擁有數十億節點的圖。

C. 與本論文差異：

Pregel 計算過程是屬於 BSP Model：由一次又一次的 Supersteps 所組成，而且在 Superstep 執行完畢後機器之間會有 Message 的交換；相對的，本論文在 Map Phase 與 Reduce Phase 時各只會執行一次，也就是當 Job Assignment 完成後各計算節點便開始執行分配的工作量，然後節點之間在 Map Phase 時並不會交換任何 Message，只有接收來自 Master 的分配結果與向 Master 回報執行結果，所以說本論文不需要考慮計算節點間 Message 交換的需求。

Pregel 在分割圖形時使用者是可以控制節點的分配，此外每台機器在執行時只會執行分配到的 Graph Partition 所以說 Pregel 與本論文相同的地方在於：我們都要求節點只處理本地儲存的資料，如此就能避免 Input Data 需要透過網路讀取所造成 Overhead。

第三節　Spark

A. 設計目的：

在 Pregel 提出之後，許多經典圖形演算法像是：Shortest Path, PageRank 等都已經實際被開發出來也在企業中實際被運用。但是每次 Superstep 執行完畢後都需要將所有節點的狀態寫入 Local Disk 中做 Check Pointing。下一次 Superstep 便從 Local Disk 中讀取上次執行結果，並重覆先前的計算過程。不只是 Pregel，其他 Application 像是 Machine Learning 也需要對同一群資料進行多次 Iteration 的計算才能完成。但若是每次執行結果都需要不斷從 Local Disk 讀取與寫入 Local Disk 是相當花費時間與空間，會對效能產生相當大的影響，Spark[10]便被設計出來用以解決這些的問題。

B. 設計目標：

Spark 是一套分散式程式 Programming Framework，利用 RDD [11] (Resilient Distributed Dataset)資料結構將需要重複讀取資料存放在 Memory 當中，如此利用記憶體的高速以省下不斷從硬碟存取的時間。

RDD 是由一群 Read Only 的物件所組成，RDD 內除了儲存需要重複存取的資料之外，也有每次 Iteration 那些資料的變化過程。這些變化過程會是一群有順序的 Function Calls，代表著一步一步對於資料的操作，透過記錄變化過程以及 Iteration 次數，當需要使用節點狀態時只用變化過程與 Iteration 次數去計算節點當前狀態。Spark 僅紀錄節點狀態的變化資訊不僅能夠減少儲存空間，當任何節點狀態遺失時亦能用快速的重新計算回復；雖然 Pregel 也是能利用 Check Point 重新計算遺失 Partition 的狀態，但是因為 Partition 需要其他 Incoming Edge 的 Message 作為 Input 所以不只從本地端也要從包含 Incoming Edge 的 Partition 讀取 Messages。相比之下 Spark 只需要從本地記憶體中讀取節點變化過程並重複執行數次就可以回復遺失的節點狀態，省去從硬碟中讀取以及透過網路傳輸所造成的 I/O Overhead。

在[7]實驗中，將在 Spark 與 Hadoop 上執行 Logistic Regression，因為其需要不斷重複對相同一群資料作計算，所以算是一種 Iterative Job，藉此作者想要知道 Spark 對於 Iterative Job 能達到多少程度的加速。作者在 20 台 m1.xlarge 規格 EC2 機器所組成的 Cluster 上對 29G 的資料進行 Logistic Regression 運算，我們可以從圖 8 中看到，雖然在只有一次 Iteration 時 Spark 執行時間稍稍大於 Hadoop，但是當 Iteration 數量開始增加時，相較於 Hadoop 上升幅度；Spark 每個 Iteration 只多花了 6 秒。根據這些數據

我們可以得知 Spark 對於 Iterative Jobs 有顯著的加速效果。

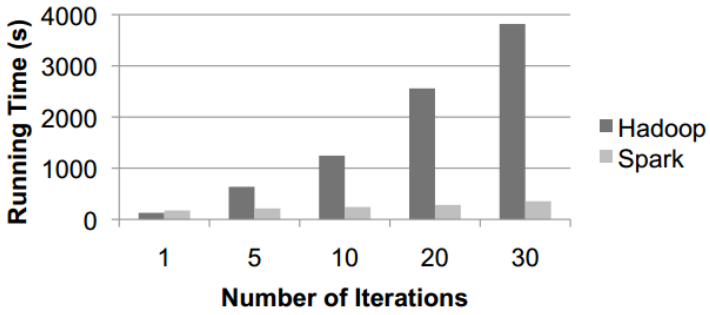


圖 8、Hadoop 與 Spark 執行不同次數下所需執行時間

C. 與本論文差異：

雖然本論文的計算模式並不是 BSP Model，但是本論文期望提供能與使用者互動的系統，使用者可以不斷對相同資料下 Query，因此 Reduce Phase 將 BRT Partitions 合併後應當存放在 Memory 中讓使用者能對同一群資料重複下 Query。像是剛開始可能是比較廣域 (國家或 AS)的 Query，但是找出可疑對象後接下來的 Query 就可能針對個別 CIDR 或 IP，當本論文將 BRT 儲存在記憶體後便能夠與使用者達到更佳的互動性。

第四節　Dremel

A. 設計目的：

Google 在搭建 GFS 後，不只搜尋引擎所需網頁資料需要儲存之外連其他業務資料像是會員資訊或機器 Log 等等，也會共同儲存在 GFS 中。為了分析如此大量資料得利用 MapReduce 才能進行，通常 MapReduce 得執行數十秒到幾分鐘才能完成，當使用者需要重複對資料做操作時就得花費相當大量時間在等待 MapReduce 完成。而且 GFS 內資料並沒有強制規定儲存格式，因為格式不一致而沒有辦法統一使用相同的資料處理方式；而且不支援巢狀 (Nested)格式也讓資料的表達方式不夠豐富。

B. 設計目標：

為了解決因為 MapReduce 導致過長的執行時間與檔案格式的問題，Dremel[12]提出兩項設計：Hierarchical Server Structure 與自行定義 Data Model。

Dremel 將 Cluster 內機器組織成樹狀結構，以 Level 為 3 的樹為例：樹中有唯一的根節點；根節點下有多個中間節點而每個中間節點也有多個葉節點。根節點接收來自使用者的 Query，依據與 Query 相關的待處理資料其所在葉節點，將 Query 分配給負責這些葉節點的中間節點；中間節點則進一步將 Query 切分成各個葉節點所需處理的範圍，將葉節點回傳的結果 Aggregate 後回傳給根節點，最後根節點再一次 Aggregate 中間結果最後才回傳給使用者。藉由將機器組織成樹狀結構，Query 能夠快速的根據資料所在位置切割成許多能夠同時進行的 Sub-Query，隨著 Sub-Query Granularity 的提高，Query 就能在更多機器上運行而在更短的時間內完成。

然後 Dremel 提出的 Data Model 不僅支援巢狀格式以表達資料中更高層語意之外，為了讓使用者可以直接存取 Model 中某個資料而模仿 OO 語言存取 Class 的方式。例如有個 Model 叫 A，它有兩種屬性 B 跟 C；然後 B 又有兩種子屬性 D 跟 E，若是我們想要讀取屬性 E 的話可以使用 A.B.E 的表示法直接存取。

在 Data Model 儲存方面，為了減少針對特定欄位查詢而需要讀取全部 Table 所花費的時間，Dremel 採用 Column Storage 的策略。Column Storage 策略是讓一張 Table 依據 Column 切割為多張子 Table，而每個子 Table 都單獨儲存，如此一來當使用者只需要查詢特定欄位時就可直接讀取相對應的子 Table，這樣就可以省下讀取其他不需要 Table 的時間。Dremel 則是讓不同的屬性儲存在不同的檔案中，讓葉節點直接從硬碟讀取對應屬性的檔案做查詢；以先前的例子來說，A.B.E 與 A.B.D 會各別儲存在兩個

檔案中，若 Query 只有查詢 A.B.E 的欄位時只需讀取 A.B.E 對應的檔案，就不需理會 A.B.D 的檔案。

在[12]中，設計了一個實驗：目的是要看出當逐步採用 Nested Data Model with Column Storage 與 Hierarchical Structure 後，對於相同 Query 其效能的提升。根據圖 9，我們可以很明顯看到當採用 Nested Data Model with Column Storage 後，原本的 MapReduce Job 執行時間下降了快一個 Order，說明了當我們能夠只讀取需要處理的資料就能夠大幅度降低浪費在讀取與處理不必要資料的時間；然後到了 Dremel 又採用 Hierarchical Structure 後，執行時間又再次下降了一個 Order，正說明了把 Query 切割成多個能夠同時執行的 Sub-Query 更能顯現出大量機器同時執行的加速程度。

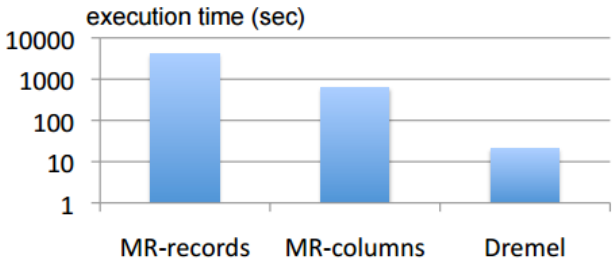


圖 9、Dremel 於相同 Query 在採用不同改善方式下的執行時間

C. 與本論文差異：

與本論文相同的部分在於為了加速查詢時間都採用了 Column Storage，將資料依照欄位儲存在不同檔案中，查詢時僅讀取所需欄位即可。但是，本論文需要完成 Query 的步驟較為複雜，因此除了 Master 機器外都會實際處理 Query 的情況下並沒有階層架構的設計。然後因為本論文的研究對象只有網路流量，比起 Dremel 所需要處理的單純許多，因此就沒有額外設計的 Data Model 來表達不同格式的資料。

第五節　Impala

A. 設計目的：

以往商業資料都儲存在 RDMS (Relational Database Management System)當中，當我們需要分析 RDMS 內的資料企圖找出 Data Insight 時，已有許多現成 OLAP (On-Line Analytical Processing)工具可以使用。但隨著資料量增長，RDMS 已經不足以負荷大量資料的處理，因此當資料便不再儲存於 Relational Database 而是儲存在 HDFS 中；當資料不再以結構化格式儲存時，原先 OLAP 工具便不再適用。

Apache 提出兩套資料分析平台－Hive[13]與 Pig[14]來滿足 HDFS 上資料的分析需求。兩者都有提供 SQL-Like 介面讓使用者能夠重覆下 Query 與資料進行互動，不過不管是 Hive 還是 Pig 都是將 Query 轉換成一連串的 MapReduce Jobs 存取 HDFS 內的資料做查詢，因此執行時間往往得花到數十分鐘到數小時。

B. 設計目標：

Cloudera Impala [15]同樣提供 SQL-Like 介面，Impala 實作了 Hive 支援 SQL 語言的一個子集，因此 Hive 使用者可以很輕鬆的轉換到 Impala 上。為了避免轉換成 MapReduce 工作所造成多階段 (Split、Map、Collect & Sort…等)執行的 Overhead，Impala 捨棄由單一 Master 指派工作的模式，反而採用 MPP (Massively Parallel Processing)模式，在每台機器上運行 Impala Daemon－直接接收使用者所下達的 Query、存取 Local HDFS 資料並回傳 Query 結果給使用者。

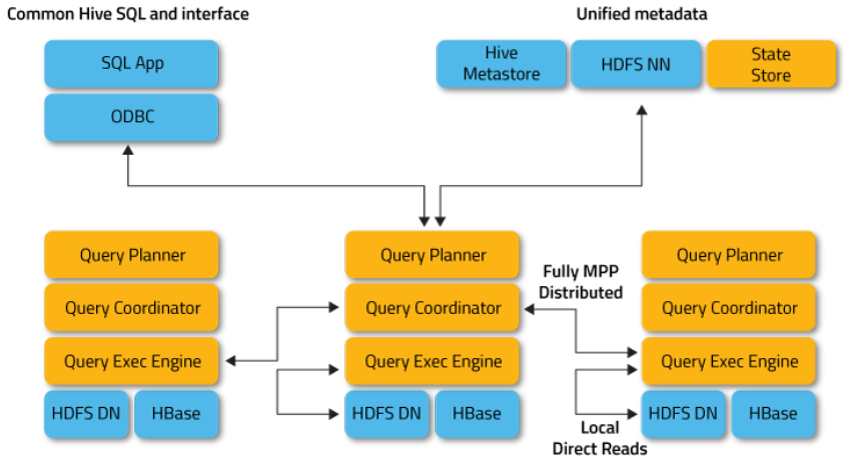


圖 10、Impala 系統架構圖

圖 10 中，Impala 在每個 DataNode 上執行 Impala Daemon，而 Impala Daemon 主要由三個 Module 所組成：Query Planner、Query Coordinator 與 Query Execution Engine。

(1) Query Planner：接收從 SQL 介面或 ODBC (Open Database Connectivity)來的使用者 Query，並向儲存資料 Metadata 的 NameNode 要求資料的位置；找出 Query 需要在哪些機器上執行後把 Query Plan 送給 Query Coordinator 並將結果回傳給使用者。

(2) Query Coordinator：依據 Query Plan 將 Query 同時交給不同機器上 Execution Engine 執行，並將各 Execution Engine 回傳結果整合後傳回 Query Planner。在執行過程中 Coordinator 會確認 Execution Engine 是否持續執行，若是有任何 Execution Engine 失去聯繫則在其他機器上重新執行。

(3) Query Execution Engine：從 HDFS 讀取 Local 檔案並執行 Query。Execution Engine 執行結果並不會儲存回檔案系統中而是直接回傳給 Coordinator，如此就能夠避免 MapReduce 中不同階段需要將 Intermediate Result 儲存到檔案系統的 Overhead。

Impala 的架構下與 Hive 最大不同在於：Impala 並沒有 Master 角色的存在，每個 Query Planner 都可以接收 Query；而 Query Coordinator 都可以接收 Query Plan 執行，如此就能避免 Master/Slave 架構下 Single point of failure 的問題。此外利用 Apache Parquet，Impala 也支援 Column Storage。

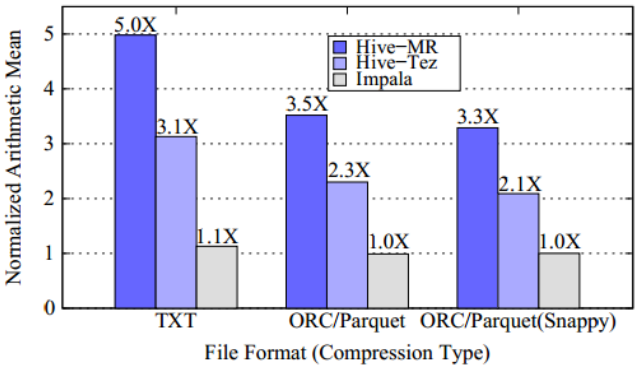


圖 11、Impala 使用不同資料格式(一未壓縮、兩壓縮)下不同 Framework 的平均回應時間

在[16]中，針對 Hive、Impala 做了相同 Query 的效能比較。圖 11 列出不同資料格式下 Query 的平均回應時間。我們可以看到在未壓縮格式下 Impala outperform Hive 5 倍的速度；就算有了 Block level 的壓縮技術 (Snappy)，Impala 依舊 Outperform Hive 3

背的執行速度。與 Dremel 的實驗相同，都說明了 MapReduce 並不適合運用在互動式資料分析，Dremel 與 Impala 同樣採用 Column Storage 來減少 Query 需要讀取資料，但 Dremel 式採用階層式架構將 Query 層層指派給葉節點的機器執行；而 Impala 則是採用 MPP 模式，讓任意的節點都可以接收、指派並執行 Query，以節省層層節點間所需的通訊成本。

C. 與本論文差異：

在本論文當中，同樣也考慮到大部分 Query 只會針對少數欄位作查詢，因此我們也是將每筆 NetFlow Record 依照資訊種類分為 Communication、Traffic 與 Application 三種分別儲存在不同檔案中以加速讀取時間。

雖然本論文也是將 Query 分配給機器同時執行，但是因為 Query 結果與處理過程較為複雜所以不採用 MPP 模式－在每台機器上執行 Query Server、Visualization Server 與其他 Master 的 Functional Component。此外我們不僅僅要求快速的執行時間，亦要求每台機器的執行時間儘量相同，所以本論文在執行 Query 之前額外增加了 Job Assignment 希望藉由平衡執行 Query 所需要的工作量以平衡機器間執行時間。

第六節　Sparrow

A. 設計目的：

從 Hive、Dremel 到 Impala，Scalable Interactive Query System 的改進讓 Query 得以在越來越短得時間內完成。為了要讓 Query 執行時間益發短暫，常見做法就是增加 Cluster 內計算節點數量以減少每台機器工作，但是當機器數量多到一種程度之後系統擔心的不再是執行 Query 太久而是因為機器數量太多不知道該如何分配工作才能讓 Query 以最快時間完成，此時最重的負擔反而是在 Scheduler 身上。

現今 Scheduler 面臨了三種挑戰：第一、當 Query 都需要在一百毫秒內完成時，就算 Scheduler 一次錯誤的指派時只會讓完成時間延後數十毫秒，但機器數量一多；造成錯誤的機率隨之增高；對完成時間的影響便被放大。第二、這些排程的 Tasks 都可以平行執行，因此每次不只 Schedule 一個 Task 而是得做出多個決策，Schedule 多個 Tasks。第三、因為 Scheduler 需要在短時間內做出大量決策，程式負擔重就容易發生錯誤而當掉，一旦發生這種狀況因為沒有人安排工作所以 Cluster 就此停擺、機器都處於閒置狀態。

B. 設計目標：

因此在 Sparrow[17]這篇論文中作者便提出 Scheduler 平行執行架構、能夠滿足 Low-latency、High Throughput 與 High availability 的 Scheduler 與 Node Monitor－他會在每台機器上運行，幫助 Scheduler 蒐集需要的資訊。

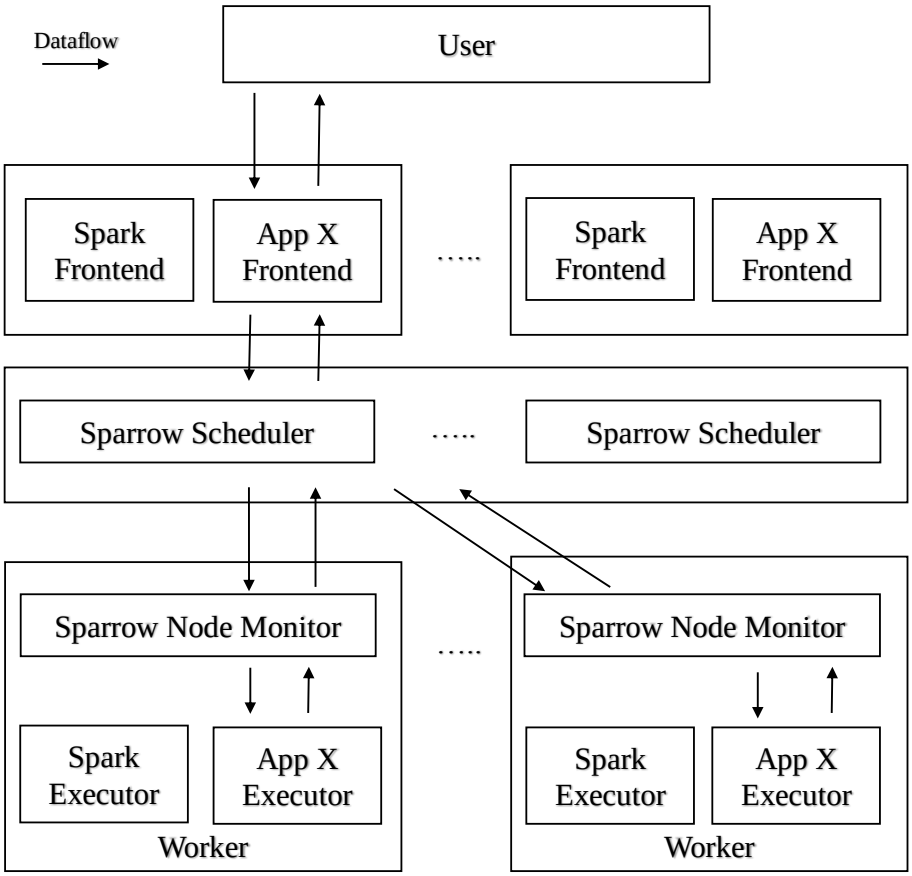


圖 12、Sparrow 架構圖，圖中箭頭代表系統中的資料流通

在平行化的架構下，Sparrow 在許多機器上同時執行 Sparrow Scheduler，另外亦在 Worker 上執行 Node Monitor 以提供 Scheduler 關於 Loading 資訊。整個執行過程從使用者提交(submit) Query 給 Application Frontend 開始，Frontend 會從 Scheduler List 中挑選一台，並將一連串的 Tasks 提交給 Scheduler，請 Scheduler 安排 Tasks 給 Worker 執行。

接著，Scheduler 依據 Node Monitor 回傳資訊將 Task 指派給 Loading 最輕的 Worker。Node Monitor 會利用 Queue 紀錄 Tasks，每次將 Queue Head 的 Task 交由 Executor 執行，當有任何 Task 完成之後便會利用一連串的 RPC (Remote Procedure

Call)回報給 Frontend。Application Frontend 會在所有 Task 執行完畢之後將結果回傳給使用者。

介紹完 Sparrow 的平行架構，在下面我們會介紹 Sparrow 是提供 High Throughput 與 Availability 的 Scheduling Techniques。

(1) Batch scheduling：原先 Scheduler 都是一次 schedule 一個 Task，在機器中隨意挑選 d 台機器並從中選擇 Task Queue 最短的將 Task 分配給他。當 Task 一多時 Scheduler 便 Schedule 不夠快，因此他們認為一次應該 schedule m 個 (m>1) Tasks。首先他們就隨意挑選 md 台機器，同樣的挑選其中前 m 個 Task Queue 中數量最少的機器將 Task 分配給他們。作者認為當 Sampling 數量變多時，Sample 到 Low loading 機器的機率便上升就能盡量把 Task 分配給負擔較輕的機器以達到 Load Balancing 的目的，減少 Task 在 Queue 中等待時間而讓 Query 盡快完成。此外，因為一次 Schedule 的 Task 數量變多也能夠提高 Scheduler 的 Throughput。

(2) Late Binding：在先前的 Scheduler 都只依據 Task Queue 的長度做分配，但事實上我們並不能從 Task Queue 的長度直接估計出剛剛分配的 Task 實際被執行的時間。比如說現有兩個 Task Queue，一個有兩個 Task 另一個只有一個 Task，按照先前的方式我們會挑選只有一個 Task 的機器將 Task 分配給他，但是有可能兩個 Task 的機器只需要 100ms 就能夠執行完畢；另一個卻得花費 300ms。為了解決上述問題，Sparrow 使用 Late Binding 技巧，也就是說我們挑選出 md 個機器時都請他們在 Task Queue 內保留一個空位給當前的 Task，等到這些空位可以立即被執行時再回復 Scheduler，此時 Scheduler 再將這 m 個 Tasks 指派給前 m 個回復的機器，剩下的機器就等到他們回復的時候再指派 no-op (No Operation)的指令，如此一來就解決 Task 數量沒辦法反應等待時間的問題。

(3) Proactive cancellation：在剛剛的情境之下，那些等待的空位若是最後沒有被真正分配 Task 的話，那等於是當初的空位被浪費掉了甚至影響到後續分配的結果。所以現在當 Scheduler 已經收到 m 個可以立刻執行的回覆時，便立刻通知剩下的(d-1)m 個機器把剛剛保留的空位取消，如此一來就能解決前述的問題。

(4) Fault Tolerance：Cluster 執行時，並不會只執行一個 Scheduler 而是執行一群待命，Cluster 會將包含所有 Scheduler 位置的列表傳給所有的 Application Frontend。當 Frontend 目前連接的 Scheduler 因為任何原因而停止運作時，Frontend 便立刻將 scheduling requests 傳給列表中下一個 Scheduler 以便繼續執行。

Sparrow 在相同的 Cluster 上執行 TPC-H Query，用以測試不同的 Scheduling Technique 效能。我們可以從圖 13 中看到，隨著 Scheduling Technique 的演進，Query

從提交給 Application Frontend 到最後 Application Frontend 回傳結果給使用者的回應時間 (Response Time)大幅度的下降：從一開始隨意指派 Task 給 Worker 執行的 Random Scheduling、一次指派一個 Task 的 Per-Task Scheduling 到 Sparrow 的 Batch Scheduling w/ late binding，回應時間的眾數下降了 4 到 8 倍而 95 百分位數至少下降了 10 倍左右。在此實驗中顯現出 Sparrow 確實能夠提供 Low latency、High throughput scheduling，藉由 Batch Sampling 與 Late Binding 讓工作儘量分配給 Low Loading 機器以達到 Load Balancing，減少 Task 等待時間讓 Query 盡快完成。

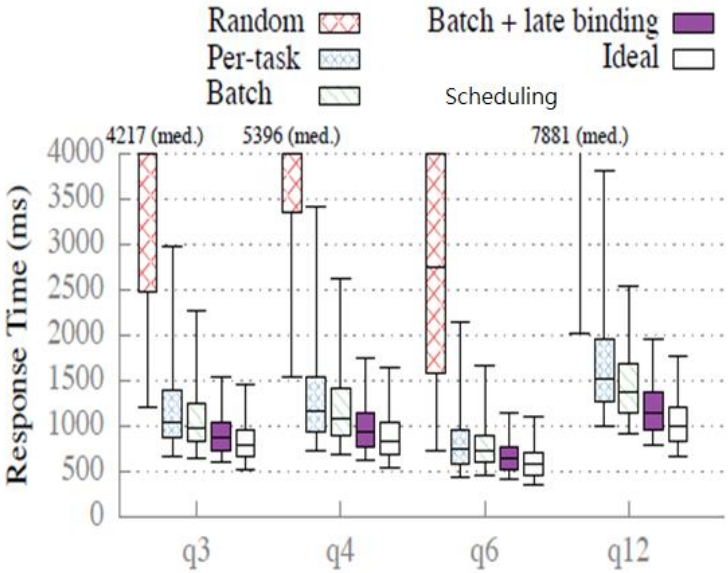


圖 13、Sparrow 執行 4 個 TPC-H Query 在不同 Scheduling Technique 下的回應時間

C. 與本論文差異：

在 Sparrow 這篇論文中，藉由動態調查機器的負載程度來完成不中斷、快速且正確的 Task Scheduling。但是在本論文中，一開始就擁有全部機器以及工作量的資訊，所以我們並不會動態的調整工作分配，當 Algorithm 1 執行完成後便將工作量分配給機器執行。不過實際執行時，可能發生有 Task 執行過久導致 Query 無法完成的狀況，當我們需要動態調整 Algorithm 1 分配結果時就能參考 Sparrow 的作法：依據機器 Loading 與資料所在位置，在其他機器上執行相同的 Task 作為備份，讓 Loading 輕的機器執行尚未完成的 Task 以減少 Query 整體執行時間被拖長的狀況。

同時，本論文尚未考慮 Fault Tolerance，NetActy 在 Master 的 Function Component (諸如：Query Server、Visualization Server 等)都只有一個 Instance 存在，很容易面臨 Single Point of failure 問題。因此藉由參考 Sparrow 的分散式架構，在許多機器上同時運行這些 Instance，當有任何機器掛掉便立即將 Query forward 給其他機器上的 Query Server，這樣就能在使用者不察覺機器發生問題的情形下而繼續執行 Query。

第三章　最小化工作完成時間差異之工作分配演算法

第一節　目標

假設給定 NetFlow Segments 檔案的分佈環境與一群可茲利用的計算節點 (Workers)，在遵循資料在地化 (Data Locality) 政策下，我們希望找出最佳的資料 (Segments)處理分配給計算節點，達到計算節點間執行時間差異最小化，使得整體執行時間不會受到執行效能差的計算節點所拖長。所謂檔案分佈環境指的是 NetFlow Data 原先已分散儲存在這些計算節點的本地磁碟，假設檔案儲存的政策是：一份檔案例如會有三份儲存(Replica)在計算叢集裡。對一個查詢（Query）而言，其關連檔案的所有儲存都會在尋找最佳分配的演算法裡被納入考量。雖然計算節點彼此之間有網路連結，但在資料在地化 (Data Locality) 政策下不希望資料檔案需經由網路傳輸到處理它的計算節點。因此，所有與本查詢相關的檔案都必須儲存在被分配處理該檔案之計算節點的本地硬碟。

我們假設每個計算節點能力有其工作容量 (Capacity)，基本上處理能力低的計算節點會分配處理較少量的資料以避免成為查詢 (Query)執行完成之落後者 (Straggler)。也就是這個問題需同時考慮、計算節點的處理容量以及與本查詢相關的資料所在位置，希望找到一個檔案處理在計算節點分配的配置以達到最小化各節點執行完成時間差異。

第二節　問題定義

根據以上要求，我們將問題制訂成一個作業研究（Operation Research）問題。此問題包含五個假設，如下所示：

(1) 計算節點可以任意地被分配資料處理，並不限定處理某些資料。

(2) 資料可以任意的分配，並不限定在某計算節點上處理。

(3) 資料處理沒有先後順序的關係，彼此之間並不依賴互相的執行結果。

(4) 一個計算節點同時只能處理一個資料，也就是資料是以一條 Queue 的形式在計算節點上等待。

(5) 資料的處理時間只與資料大小(Segment size)成正向線性關係，與計算節點的執行效能沒有關係。

問題所定義的變數如表 1，表中最後三個是 Decision Variables。

表 1、Algorithm 1 定義之變數

Notation	Definition
Q	This query Q is given by user, specifying the time and space domains for the targeted data set.
β	The number of replicas, i.e. ,each segment file has β copies stored on the local disks of β workers.
$\{S_j\}_{j=1...N_s}$	The set of segments which are relevant to Q. $\|S_j\|$ is the size of S_j. N_s is the total number of segments subject to Q.
$\{W_i\}_{i=1...N_w}$	The set of worker machines. N is the number of total workers while N_w is the number of workers in which local disk it has a replica of $\{S_j\}$., $N_w <= N$.
$\{C_i\}$	The initial capacity of W_i.
$\{L_{i,j}\}$	A set of binary varibles indicating the location of Segments subject to Q $L_{i,j} = \begin{cases} 1, \text{ if } S_j \text{ stored on } W_i\text{'s local disk..} \\ 0, otherwise. \end{cases}$
$\{B_{i,j}\}$	The segment assignment. A set of binary variables such that $B_{i,j} = \begin{cases} 1, if\ S_j \text{ is assigned to } W_i. \\ 0, otherwise. \end{cases}$
M	The total number of workers that have non-zero workload.
$\{F_i\}$	The total amount of workload assigned to W_i.

目標函式與限制式定義如下：

$$\min \frac{\sum_{1\le k<l\le M} |F_k - F_l|}{\binom{M}{2}}$$

$$subject\ to \quad \sum_{i=1}^{N} B_{i,j} = 1, \forall\ j \in \{S_j\}_{j=1...N_s} \quad \text{.....(1)}$$

$$L_{i,j} = \begin{cases} 1, \text{ if } S_j \text{ stored on } W_i\text{'s local disk..} \\ 0, otherwise. \end{cases} \quad \text{.....(2)}$$

$$\sum_i L_{i,j} = \beta, \qquad \forall 1 \le j \le N_s \quad \text{.....(3)}$$

$$B_{i,j} \le L_{i,j}, \qquad \forall 1 \le i \le N_w, 1 \le j \le N_s \quad \text{.....(4)}$$

$$F_i = \sum_{j=1}^{N_s} B_{i,j} \times |S_j| \quad \text{.....(5)}$$

$$F_i \le C_i, \qquad \forall 1 \le i \le N_w \quad \text{.....(6)}$$

$$M \le N_w \quad \text{.....(7)}$$

我們所提出的工作分配方法的設計理念是令目標函式比較所有有被分配到不為零工作量(Workload)的計算節點上，任兩個節點的工作量差距絕對值的總和，除上所有任意挑選兩個機器的組合數量為最小化。也就是說，我們希望藉由縮小計算節點間工作量的差距以達到不會被因工作量分配不均造成執行慢的節點拖長整體執行時間。目標函式是 Gini's Mean Difference[18]，所提出的方法不採用 Variance 做為目標函式有以下幾點原因：

(1) 我們希望任兩個計算節點的工作量之間的差異最小化，Variance 描述的是整體工作量的變異程度，計算只有考慮各工作量與平均工作量(Average Workload)的距離，而非工作量間的差距。

(2) 變異數的計算公式中 $\sigma^2 = \frac{\sum (x_i - \bar{x})^2}{M}$，除了有分母之外還有平方存在，此公式的非線性程度比 Mean Difference 大。此外在[19]中也提到因為平方關係對於距離平均工作量較近的工作量會給予較低的權重，反之給予較高權重；而 Mean Difference 的計算方式則同等對待所有的工作量之間的差距。

以下一一說明各限制式：

(1) 限制式 1 表示在 This Query 下的資料檔案一定都會被指派給一個且只有一個計算節點處理，不會有重複指派的情況發生，又稱為 One-Worker-Only Assignment Constraint。

(2) 限制式 3 表示在 Replica Level 為β的時候，也就是任意一個資料檔案 S_j 會有 β 份複本分別儲存在 β 個計算節點上，不多也不少，這些複本的內容皆相同。

(3) 限制式 4 表示每次針對一個資料檔案 S_j 進行分配給哪一個節點要處理該檔案時，會考量哪些計算節點的本地硬碟存有 S_j，我們只會在這些計算節點中挑選，以避免需要透過網路傳輸複本以減少執行時間，又稱為 Data Locality Constraint。

(4) 限制式 6 表示該計算節點被分配到的總工作量不應該超過他本身所可以承受的工作容量（上限值）。我們針對機器的本身運算能力以設定工作上限值，這樣一來可以避免讓效能較差的計算節點負責太多的工作量導致整體執行時間被拉長，又稱為 Worker Capacity Constraint。

(5) 限制式 7 表示有被分配到工作量的計算節點數量不可超過可以被分配工作量的計算節點數量。

第三節　最小化計算節點執行時間之間差距的工作分配演算法

在此章節我們將分析本論文所欲解決的最小化各計算節點工作完成時間問題的計算複雜度，以了解解題所需的計算成本；接著探討現有文獻上解題方式的優缺點；最後則是參考現有的解題方式，納入問題本身的一些有用的知識提出一個較低計算複雜度的經驗演算法(Heuristic Algorithm)。

3.1 複雜度分析

Cook [20] and Karp [21]對於決定性問題 (Decision Problem)提出問題複雜度的分類方式，其將問題分類為 P、NP 或 NP-Complete。所謂一個決定性問題是指不論這些問題內容為何，他們的解答只會是 Yes 或 No；被分類到 P 的問題是指可以使用決定性(deterministic)圖靈機(Turning Machine)在多項式時間 (Polynomial time)內解決的問題；NP 問題是指使用非決定性(non-deterministic)圖靈機在多項式時間內解決的問題。決定性與非決定性圖靈機的差別在計算的每一時刻，根據當前的狀態以及輸入，若機器的行為可唯一確定也就是只有一種執行方式可以做時則為決定性圖靈機，相反的若是有很多行為可供選擇則為非決定性圖靈機。雖然 NP 問題可以使用非決定性圖靈機在多項式時間內解決，但是非決定性圖靈機是在假設計算資源無限的情形下運作，實際上這種機器並不存在，所以在資源有限的情況下，NP 問題並無法在多項式時間內解決。

$P \in NP$ 可以利用驗證解答的方式證明，因為任意屬於 P 類問題的解答可以在非決定性圖靈機上執行相同的多項式演算法 (polynomial algorithm)驗證其正確性。但是 $P = NP$ 依然是個 Open Question，目前尚無能夠證明其成立或是不成立。NP-Complete 是定義為 NP 問題中最難的一群，任意的 NP 問題都可以在多項式時間內約化(reduce)成一個 NP-Complete 問題。因此，一個決定性問題必須滿足其複雜度為 NP 以及可以被任意 NP 問題約化成的兩種條件才被稱呼 NP-Complete。因為目前已被證明為 NP-Complete 的問題大部分有已知解法，所以當我們把欲解決問題轉化成 NP-Complete 問題後就能直接利用現有解法來解決原本的 NP 問題。

而當某個問題只滿足 NP-Complete 問題兩個條件的第二個：可以被任意 NP 問題約化成時，我們就稱為 NP-Hard 問題。那因為該問題有可能約化後依然是 NP 問題，如果是這種狀況則表示這些問題等同 NP-Complete 問題，但約化後若不再是 NP 問題則表示該問題比 NP 問題還難。故稱為 NP-Hard，代表這些問題至少比 NP 問題還難。

這種在多個計算節點間決定工作量的分配能否在各節點計算容量擁有上限值與成

本的情況下，找出可行分配的解答可歸屬於 Knapsack 問題，其複雜度為 NP-Complete。本論文所探討的問題是一個找最佳化解的問題：除了找出可行的工作分配之外，此分配還必須最佳化各節點執行時間差；因此理論上要找到最佳解，我們得嘗試所有可能的工作分配在可能的計算節點組合上，這是一種 Combinatorial Optimization Problem，已經在[21, 22]中證明就算這種問題經過簡化其複雜度依然為 NP-Hard，所以不可能在多項式的時間找出最佳解。

在[23]此篇論文研究中考慮執行節點間工作（Task）執行之先後順序關係以及計算節點間的通訊成本，目標是找出最小整體執行時間(Makespan)的工作規劃(Allocation)與排程(Scheduling)。雖然與我們想要解決的問題並非全然相似，不過作者也是把問題制訂成（Formulate）成一個 OR 問題後透過 MILP(Mixed Integer Linear Programming)方法找到最佳解(Optimal Solution)。

MILP 是指一個 LP(Linear Programming)問題中存在一些整數變數(Integer Variables)或二元變數(Binary Variables)，所以稱為 Mixed Interger。基本上，在問題的描述裡二元變數的值只會是零或一，通常可以用來表示工作有或沒有在某 Worker 上執行、工作可不可以在某 Worker 上執行或是工作之間的先後順序關係。在我們的問題中因為有 $\{B_{i,j}\}$與$\{L_{i,j}\}$這兩個二元變數的存在，所以我們可以利用 MILP 技巧來解決問題。

MILP 擁有幾項優點。首先，它確實可以找出最佳解。此外，現成一般的解題軟體(Solving Tools)亦支援 MILP 方法。但是因為其採用 Branch and Bound 的解題策略，因此 MILP 的解題之指數時間的計算複雜度常導致找最佳解所需時間過長。

以本論文所欲解的最佳化問題為例，在 Branch and Bound 下的解題過程可以被視為一棵深度為 N_s 的 β 元樹：每一層(Level)代表一筆資料的分配，該層代表該筆資料分配狀況的二元變數的所有可行值。因為我們每筆資料有 β 份複本所以會有 β 個節點。解決 MILP 的過程就猶如由上 (Top) 往下 (Bottom) 一層一層地搜尋（Traverse）這棵樹，當抵達葉節點時也就是找到一種分配的完成。因為這棵樹有 N_s 層（每一層代表一筆檔案的分配處理）；每層都有 β 個節點，因此所有可行解的分配方式會在這棵樹的 β^{N_s} 個葉節點(Leaf Node)。理論上我們得走過所有的葉節點然後選擇其中最好的分配方式做為 MILP 的解。由以上我們可以知道可行解的個數與欲分配處理之檔案個數數量成指數關係，因此，MILP 具有指數時間的時間複雜度。

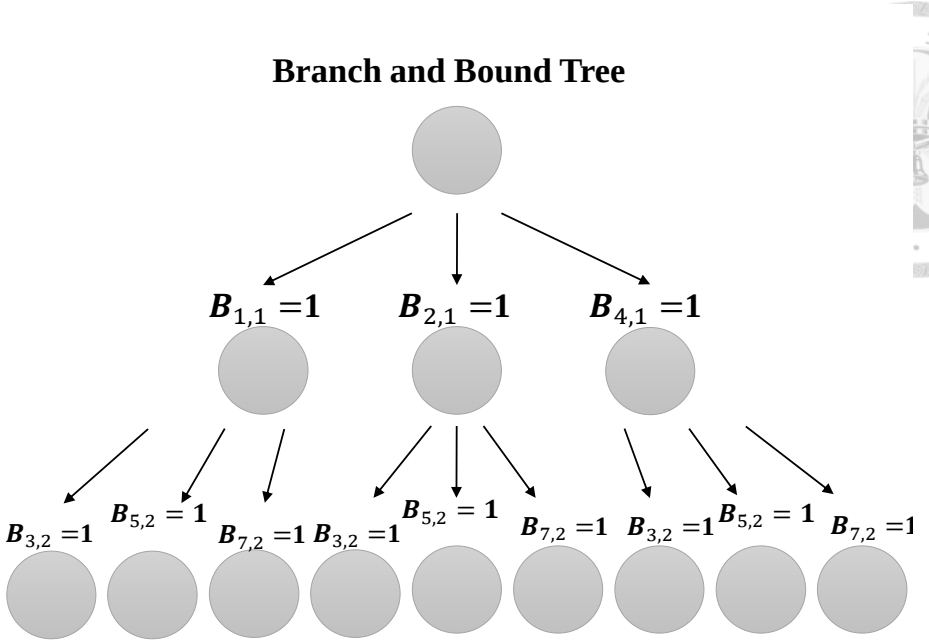


圖 14、對 LP 問題於 MILP 方法中，採用 Branch and Bound 策略解題示意圖，假設β為 3，檔案 1 的三份複本在 1,2,4 三個節點上；檔案 2 的三份複本在 3,5,7 三個節點上。

3.2 Heuristic Solution

前面段落提到一個 Combinatorial Optimization Problem 的計算複雜度為 NP-Hard，就算經過簡化其複雜度依然為 NP-Hard。NP-Hard 問題並沒有辦法在多項式時間內被解決；此外 MILP 方法也具有指數時間的時間複雜度。實務上，為避免過長的執行時間，大部分的研究都採用經驗上逼近(Heuristic Approximation)的技巧在犧牲找最佳解的條件下縮短所需的計算時間。

經驗解(Heuristic Solution)的做法，通常是透過加速移除一個多元變數的過程來減少問題解決時間。一種方式是利用 Diving Heuristic，也就是當 Branch 出一個多元變數的可能值時，並不一定每一個可能值都會引到(Lead)一個可行解，我們希望運用對問題本身的瞭解或是知識將之描述成條件式以快速的篩選出可能有可行解的分枝（Branch）並往下走，盡快的走到葉節點找出一組可行解。因為此種方式一直不斷的往下走，故稱為 Diving Heuristic。

本論文的目標是支援即時互動的網路資安分析系統，為避免採用 MILP 方法之過長的執行時間以提供使用者多項式時間的回應時間，所以設計一個資料分配的啟發式演算法(Heuristic Algorithm)－Algorithm 1，在使用者可接受的時間範圍內找出計算節點工作分配的可行解。在接下來的篇幅，我們會說明 Algorithm 1 的設計理念與執行流程。

Algorithm 1 的目的並不是要找出最佳解，而是最快得找出可行解同時希望此可行解能越接近最佳解越好。為此我們參考 Diving Heuristic 的想法，在每次分配資料時只選擇局部最佳(Local Optimal)的 Branch 往下走希望能夠走到一個葉節點也就是我們找到的局部最佳(Local Optimum)，而不是 Traverse 所有的葉節點找全域最佳解(Global Optimum)，藉以省去 Tree Traversal 的時間。

在問題定義之限制式(1)、(4)、(6)是我們問題的核心，MILP 找出的解要滿足這三條限制式的同時也擁有最小的 Mean Difference。我們提出的 Algorithm 1 在分枝的選擇上，會先滿足 One-Worker-Only Assignment (1)式與 Data Locality Constraint (4)式，再滿足 Worker Capacity Constraint (6)式。因為我們認為確保資料能夠在本地執行是最重要的，以避免大量的資料透過網路傳輸：一來傳輸時間會受到網路狀況影響而讓整體執行時間沒辦法與 Algorithm 1 執行結果相符；二來萬一傳輸過程中有出錯，我們得花額外的功夫(Efforts)重傳資料，讓每個計算節點開始處理資料的時間不盡相同進而影響 Algorithm 1 執行結果。

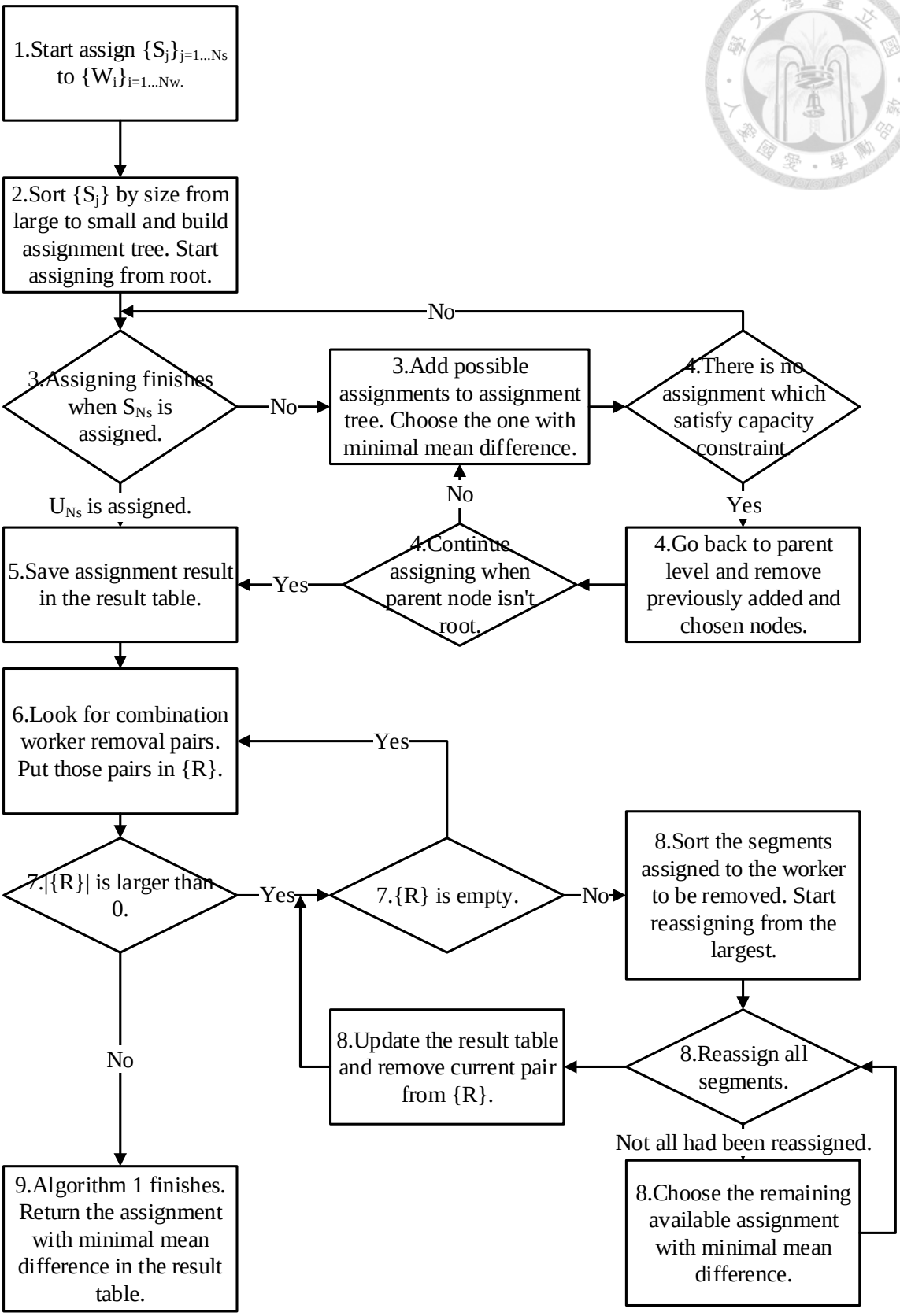


圖 15、 Algorithm 1 流程圖

以下，詳細說明所提的演算法的設計邏輯與方法。圖 15 為演算法之流程圖。

1. 為解決目標函式的非線性，我們得給定 K 值（固定所欲分配工作的計算節點數）讓目標函式分母成為常數。我們先嘗試將$\{S_j\}_{j=1...N_s}$分配給$\{W_i\}_{i=1...N_w}$。因為這$\{W_i\}_{i=1...N_w}$計算節點集合所儲存的資料集合是最完整的，理論上最有可能在這計算節點集合中找出可行分配方式，所以我們從$K = N_w$開始執行 Algorithm 1。

2. 我們假設大容量檔案消耗計算資源的量較小容量檔案多，所以先從大容量檔案開始分配。首先，我們將所有檔案$\{S_j\}_{j=1...N_s}$按照大小做排序從大到小，令$|S_1| \geq |S_2| \geq \ldots \geq |S_{N_s}|$。接著我們將每筆資料的分配決策利用樹狀結構來表達稱為分配樹(Assignment Tree)：樹的節點都代表一種分配方式；樹每層都是由一筆資料可能的分配方式所組成；節點與節點之間的連結代表分配決策。

一開始建立的分配樹只有一個根節點(Root Node) 表示分配開始，隨著資料分配的進行會有越來越多節點新增到分配樹中，越先被分配的資料會越接近根節點。分配樹建立的過程如下：首先分配的是S_1，S_1有β份複本分別儲存在β個計算節點上，分配給不同計算節點會產生相對應在此分配下各計算節點的應執行工作量，我們就可以計算各節點的工作完成時間之 Mean Difference。把這β種不同分配方式的節點加入分配樹後與根節點相連接就形成的一顆深度為 1、具有β個葉節點的樹，接著我們從這β個葉節點中選擇擁有最小 Mean Difference 且滿足 Capacity Constrain 的分配方式，這樣便完成S_1分配。基於先前的分配結果再繼續分配S_2，將S_2的β種分配方式的節點加入分配樹與S_1選擇的節點連接後再從中挑選節點，如此不斷重複新增節點與挑選的過程就完成分配樹的建立。

為什麼我們使用分配樹的理由在於當遇到沒辦法繼續分配的情況－當前分配方式皆無法滿足 Capacity Constraint 時，得重新分配上一筆資料，因為樹上的節點都有紀錄 Mean Difference 所以不需要重新計算，只要選擇除了剛剛的節點之外，那個符合 Capacity Constraint 且擁有最小 Mean Difference 的分配方法，然後繼續分配下去。所以每當我們需要調整先前的分配方式時，分配樹能省去重新計算 Mean Difference 的成本以加速回溯 (back track)的過程。

3. 從S_1開始分配到S_{N_s}結束。對當前S_j的分配選擇上，對擁有S_j的計算節點個別計算如果將S_j分配給這個節點則計算目前整體分配之 Mean Difference，在所有可能的選擇上挑選計算節點其結果 Mean Difference 是最小且不超過其工作上限值的。因為在每一次資料分配選擇是選當時 Mean Difference 最小的分配方式，就等同於在找局部最佳解，這是一種貪婪演算法(Greedy Algorithm)的精神。

在分配過程中我們會不斷的紀錄每筆資料的分配結果。在演算法中，以γ表示該分配方式的 Mean Difference；Ψ為一個長度為N_s的向量，裡面的每個維度分別記錄每一個檔案分配給的計算節點的ID；Φ為一個長度為N_w的向量，記錄每一個計算節點被分配到的檔案 ID(s)。因此當一筆資料S_j分配給計算節點W_i後我

們便更新γ成當前分配方式的 Mean Difference；$\Psi[j]=i$表示S_j是由W_i負責處理；$\Phi[i].add(j)$把S_j加入W_i分配到的資料列表中。藉由從資料與計算節點兩個角度紀錄分配結果，我們可以快速得知負責該筆資料的計算節點同時了解計算節點間工作量關係。

接著我們想要知道分配完成之後是否能夠滿足三條限制式。在整個分配過程中我們只會將S_j分配給$L_{i,j}$=的W_i，所以(4)式一定滿足；而且我們會遍尋完 $\{S_j'\}$，每次只選擇一個計算節點，故每個S_j至少會而且只會分配給一個計算節點，如此一來(1)式就滿足；同時，每次挑選計算節點的時候會考慮到：如果我們將S_j分配給W_i時，資料大小是否大於剩餘工作上限值，若不會超過才分配給W_i，否則會挑選其他計算節點負責處理S_j。也就是說，每次成功的分配都會滿足(6)式。

4. 若是遇到當前S_j沒辦法被分配，也就是不管分配給哪一個計算節點W_i都會超過其工作上限值 (C_i)的情況下 ($F_i+|S_j|>C_i$)便將剛加入的節點與分配樹上的父節點刪除，然後回到父節點那層重新選擇可行的分配方式。每當我們遇到不能分配的狀況便會回到上一層，最後如果回溯到根節點表示所有分配方式都嘗試過也都不可行，我們就紀錄此問題無解。
5. 分配完成的結果我們會記錄在 Result table，這張表有 K 列分別記錄每一個 K 值找到的可行分配方式。以$\{\gamma_i(K),\Psi_i(K),\Phi_i(K)\}$表示找到的可行解，又$A=C_K^{N_w}$、$i=1\cdots I_A$、$I_A\le A$，代表該 K 值所有可能的分配方式。裏頭包含了分配方式 Mean Difference 也就是γ以及從資料與計算節點兩個角度所觀察的分配結果就是我們一直紀錄的Ψ與Φ。若是不可行的分配方式我們便在對應的 K 值留下$\{\gamma_i(K)=\infty,\Psi_i(K)=\phi,\Phi_i(K)=\phi\}$的紀錄。

當$K=N_w$開始執行時，若是每一個計算節點的工作上現值 (C_i)都超過該節點上所儲存的資料大小總合 ($\sum_{j=1}^{N_s}|S_{i,j}|\times L_{i,j}$)則勢必能找到一個可行分配方式；反之則不保證能夠找到可行解。要是連最充裕的計算節點集合$\{W_i\}_{i=1...N_w}$都無法找到可行分配方式時，則此問題就無解，Algorithm 1 就此結束。

當找到第一個分配方式時，藉由參考向量$\Phi[i]_{i=1\cdots N_w}$我們可以統計有多少個計算節點有分配到工作量，也就是假設Φ有 M 個維度大於 0 時我們便在 Result Table 的 M 列紀錄此分配結果－$\{\gamma_1(M),\Psi_1(M),\Phi_1(M)\}$。

6. 為了解決最佳化問題，理論上我們需要嘗試所有$C_K^{N_w},K=1\cdots N_w$種計算節點組合，找出可行分配方式記錄在 Result table 中最後從 Result table 內選擇 Mean Difference 最小的分配方式。但是我們可以觀察到在$K=N_w$，必須分配給N_w個計算節點時只有一種計算節點組合，而當分配給N_w-1個計算節點時，得選擇一個計算節點從當前組合中捨棄因此有N_w種可能組合：也就是當前 K 值的計算節點組合等於上個

K 值的某些組合再移除掉一個計算節點。根據以上觀察，K 從 $N_w - \beta$ $(\beta = 3)$ 開始我們並不會將全部 $C_K^{N_w}$ 組合納入分配的考量中，而是檢視上個 K 值的每一組可行分配方式試著從中拿掉一個計算節點，將該計算節點上的資料都分配給其他有工作量的計算節點後做為新的分配方式記錄下來。藉由上述過程以避免嘗試所有組合所需要的指數時間。

同時，為了讓新的分配方式仍舊滿足 Data Locality Constraint，我們會計算當前分配方式中每一個工作量大於 0 節點的可替代率(Substitution rate)。可替代率定義為該節點所分配到的資料中仍有其他複本儲存在具有工作量節點的資料數量所占全部資料的比例，而當可替代率為 1 時，代表該計算節點的工作量可以完全分配給其他擁有工作量的計算節點負責，所以我們會選擇可替代率為 1 的節點進行移除以免將資料重新分配給原本不具有工作量的計算節點而導致 K 值的上升。

可替代率的計算過程如下：對於當前計算節點，我們會檢查每一筆分配給該節點的資料剩餘的複本儲存在哪些節點上，而那些節點是否具有工作量，如果至少有一個儲存複本的節點具有工作量的話，我們便用一個計數器計算這樣的資料有多少筆。最後將計數器除上當前計算節點分配到的資料總數就可以得到節點的可替代率。

在開始執行這步驟時會看看 Result Table 內 M 列有沒有分配方式，如果有的話便開始尋找 Combination－Worker Removal Pair，從步驟五找到的第一個分配方式的 K 值－M 開始尋找 Removal Pair。每個分配方式先各別計算節點的可替代率，在一個分配方式內可能有不只一個可替代率為 1 的節點，在 K 大於 M - 3 時這些節點都需要被嘗試移除所以將每個可替代率為 1 的節點與當前的分配方式作為一個 Removal Pair 紀錄在{R}內；我們使用集合{R}儲存 Removal Pairs。但在 K 小於等於 M - 3 後每個分配方式只會移除一個可替代率為 1 的節點，因此每個分配方式只產生一組 Pair 紀錄在{R}內。最後當找出所有可能 Pair 後將 K 值減 1，以便再次步驟 6。

這步驟當 K 列沒有任何分配方式便停止，代表前個 K 值的分配方式已經沒有任何可替代率為 1 的節點，移除任意的節點都會導致某些資料無法被分配到，因此 Algorithm 1 即可結束。

下面我們證明當前做法就算是 Worst Case 也能夠降低到多項式時間的計算複雜度，假設每個組合都至少能找到移除一個計算節點後的新組合。

Method / K value	Combinational (列舉所有分配方式)	Heuristic (根據上次組合推導)
N_w	$C_{N_w}^{N_w}$	1
$N_w - 1$	$C_{N_w-1}^{N_w}$	N_w
$N_w - 2$	$C_{N_w-2}^{N_w}$	$\frac{N_w \times (N_w - 1)}{2}$
$N_w - 3$	$C_{N_w-3}^{N_w}$	$\frac{N_w \times (N_w - 1)}{2} - (N_w - 3)$
⋮	⋮	⋮
1	$C_1^{N_w}$	$\frac{N_w \times (N_w - 1)}{2} - 1$
Total combinations	$\sum_{K=1}^{N_w} C_K^{N_w} = 2^{N_w} - 1$ （根據二項式定理）	$(\frac{N_w \times (N_w - 1)}{2}) \times (N_w - 2) - \sum_{i=1}^{N_w - 3} i + N_w + 1$

表 2、在採用與未採用 Algorithm 1 所提出經驗解於 Worst Case 需要嘗試的計算節點組合數量比較。

根據表 2，Combinational 方式其計算複雜度為 $O(2^{N_w} - 1) = O(2^n)$ 而 Heuristic 方法的則是 $O(n^3)$。正如同先前的敘述，隨著計算複雜度的降低能大大減少執行所花費的時間。

7. 接著將{R}內的每個 Pair 需要移除的計算節點上的資料重新分配，當所有 Pair 都重分配完後便重新執行步驟 6，若是{R}為空集合代表沒有任何分配方式擁有可被移除的計算節點因此 Algorithm 1 就此結束。
8. 每當我們開始處理一組 Removal Pair，先將需要被移除的計算節點 W_i 上被分配到的資料如同步驟 2 的理由也依照大小從大排序到小，接著每筆資料 S_j 看看那些除了當前節點外亦擁有資料複本且工作量不為零、重新分配後不超過工作上限值的計算節點 W_u，分配給哪一個 W_u 所產生的分配方式 Mean Difference 最小便將 S_j 分配給它，但若因為工作上限值的原因而找不到 W_u 便捨棄當前的 Removal Pair。如此不斷重複上述步驟直到需要重新分配的資料都已經被分配出去，將新的分配方式記錄在 Result table 後繼續下一組 Removal Pair。
9. 當 Algorithm 1 結束時會回傳 Result table 內擁有最小 Mean Difference 的分配方式作為原先 LP 問題的 Heuristic Solution。

Algorithm 1: Heuristic Solution of LP.

Input: $N_w, N_s, \{W_i\}_{1\le i\le N_w}, \{C_i\}_{1\le i\le N_w}, \{S_j\}_{1\le j\le N_s}, \{L_{i,j}\}_{1\le i\le N_w, 1\le j\le N_s}$

Output: $B_{i,j}$

sort($\{S_j\}$) /* from large to small

/* Build Assignment tree.
$AT \leftarrow \phi$
$AT.add(T_0)$ /* add the level 0 node a.k.a. root to T.
$\Psi[0] = 0$
ResultTable $\leftarrow \phi$

/* Assign segments from U_1 to U_{N_s}.
$Traceback \leftarrow false$
for P **from** 1 **to** $N_s + 1$ **do**
  $\{W_{i,P}\} \leftarrow$ the workers with $L_{i,P} = 1$.
  $\{T_{i,P}\} \leftarrow$ the nodes in the AT if we assign S_P to $W_{i,P}$.
  $TN \leftarrow$ the temporary tree node.
  $TN.\text{md} \leftarrow \infty$ *and* $TN.\text{id} \leftarrow 0$
  if $P == N_s + 1$ **do**
    /* When only M workers are assigned workload.
    $\gamma_1(M) \leftarrow T_{\Psi_1[N_s],N_s}.\text{md}$
    ResultTable(N_w).add($\{\gamma_1(M), \Psi_1(M), \Phi_1(M)\}$)
  else if $Traceback == false$ **do**
    for each *node* $T_{i,P}$ **in** $\{T_P\}$ **do**
      F_i+= $|S_P|$
      $T_{i,P}.\text{md} = \text{computeMeanDifference}(\{F_i\})$
      $AT.link(T_{i,P}, T_{\Psi_1[P-1],P-1})$
      if $F_i \le C_i$ *and* $T_{i,P}.\text{md} < TN.\text{md}$ **do**
        $TN.\text{md} = T_{i,P}.\text{md}$
        $TN.\text{id} = \text{i}$
      end
      F_i−= $|S_P|$
    end
  else do
    for each *node* $T_{i,P}$ **in** $\{T_P\}$ **do**
      if $F_i + |S_P| \le C_i$ *and* $T_{i,P}.\text{md} < TN.\text{md}$ **do**
        $TN.\text{md} = T_{i,P}.\text{md}$
        $TN.\text{id} = \text{i}$
      end
    end
  end
  if $TN.\text{md} == \infty$ do
    if $P == 0$ **do**
      ResultTable(N_w).add($\{\gamma_1(N_w) = \infty, \Psi_1(N_w) = \phi, \Phi_1(N_w) = \phi\}$)
    else do
      $AT.remove(\{T_P\})$

$AT.remove(T_{\Psi[P-1],P-1})$
$\Phi_1[\Psi_1[P-1]].\text{remove(P)}$
$\Psi_1[P-1] = null$
P−= 2
$Traceback = true$
end
else
$F_{TN.\text{id}}$+= $|S_P|$
$\Phi_1[\text{id}].\text{add(P)}$
$\Psi_1[P] = TN.\text{id}$
$Traceback = false$
end
end
/* Start remove worker from the assignment in row M of result table.
K ← M
while K > 0 **do**
{R} ← ϕ
for each *assignment* AS = $\{\gamma(K), \Psi(K), \Phi(K)\}$ **do**
/* {SR} is the substitution rate set and SR_i is the substitution rate of W_i.
{SR} ← computeSub$stitutionRate(\Phi(K))$
for each SR_j *in* {SR} **do**
if $SR_j == 1$ **do**
{R}.$add(\{AS, j\})$
if K < M − 2 **do**
break
end
end
end
end
if |{R}| > 0 **do**
escape ← false
while {R} is not empty **do**
/* $\{AS, i\}$ is the current processing pair.
sort($AS.\Phi$[i]) /* from large to small
for each S_j *in* $AS.\Phi$[i] **do**
W_t ← the worker with minimal mean difference while $L_{t,j} = 1$,
$F_t > 0, F_t + |S_j| < C_t$ *and* $t \neq \Psi$[j].
if $W_t \neq null$ **do**
F_t+= $|S_j|$
$F_{\Psi[j]}$−= $|S_j|$
$AS.\Psi[j] = t$
$AS.\Phi[t].add(j)$
$AS.\Phi[i].remove(j)$
$AS.\gamma = T_{t,j}.md$
else do
escape = true
break
end

```
                end
                {R}.remove({AS, i})
                if escape == true do
                    break
                else do
                    ResultTable(K − 1).add({AS.γ, AS.Ψ, AS.Φ})
                end
            end
            K−= 1
        else do
            Break
        end
end
/* Return the assignment − B_{i,j} with minimal mean difference among all
/* assignments in the result table.
return B_{i,j}
```

3.4 實驗設計

借此為瞭解所提出的 Algorithm 1 是否能夠確實將工作量平均分配以達到各計算節點儘量相同的執行時間，我們設計了一個實驗：執行四個 Query 後紀錄各計算節點工作量與執行時間。

A. 實驗環境介紹

實驗在 4 台實體機器，每台實體機器上開啟 8 台 VM 共 32 台 VM 所組成的 Cluster 上完成，4 台實體機器以一台 1G Switch 連結而成為 Private Network，此 Cluster 並沒有連上網際網路以避免外部流量影響執行過程。在這 32 台 VM 中因為有一台將作為 Master 所以會被分配工作量與實際儲存資料(Segment)的 VM 只會有 31 台。β 設定為 3，也就是每筆資料都會有三份複本個別儲存在不重複的 VM 上，在後面論文中我們將實際執行工作的 VM 稱呼為計算節點(Worker)。

OS	CentOS 6.5 Final
CPU	i7-3770s, 8 Cores.
Memory	16G
Disk	1T

表 3、實體機器規格

OS	CentOS 6.5 Final
CPU	i7-3770s, Pinned on 1 core.
Memory	2G
Disk	100G
Hypervisor	KVM

表 4、VM 規格

B. Query

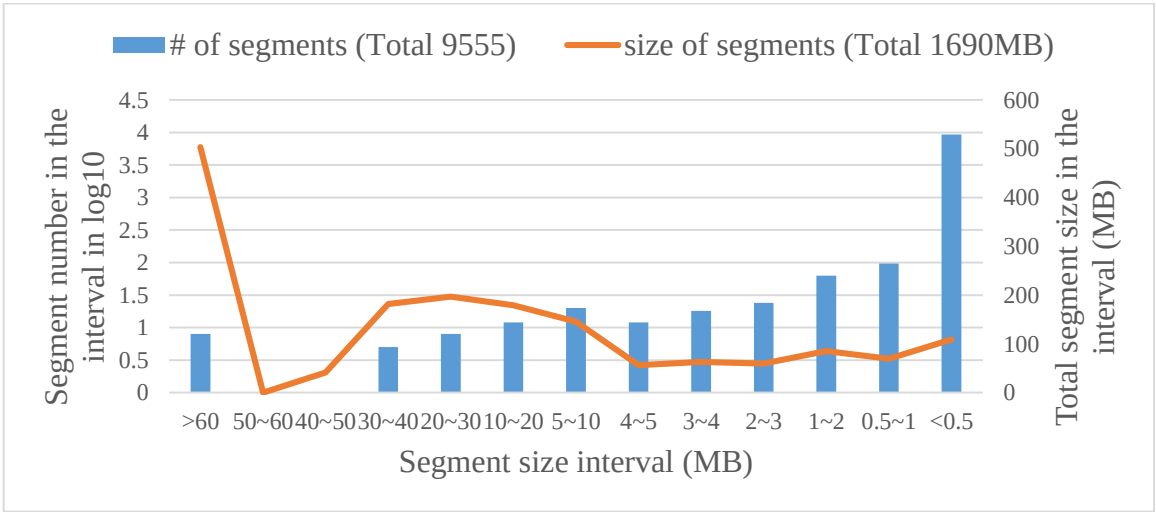
Query Id	Direction	Time	Total volume of segments	Capacity per worker
1	in-out	0:00-12:00	1690.41MB	100MB
2	out-in	0:00-12:00	1779.01MB	100MB
3	in-out	0:00-24:00	2164.85MB	100MB
4	out-in	0:00-24:00	2186.10MB	100MB

表 5、Q1 至 Q4 各 Query 之細節。

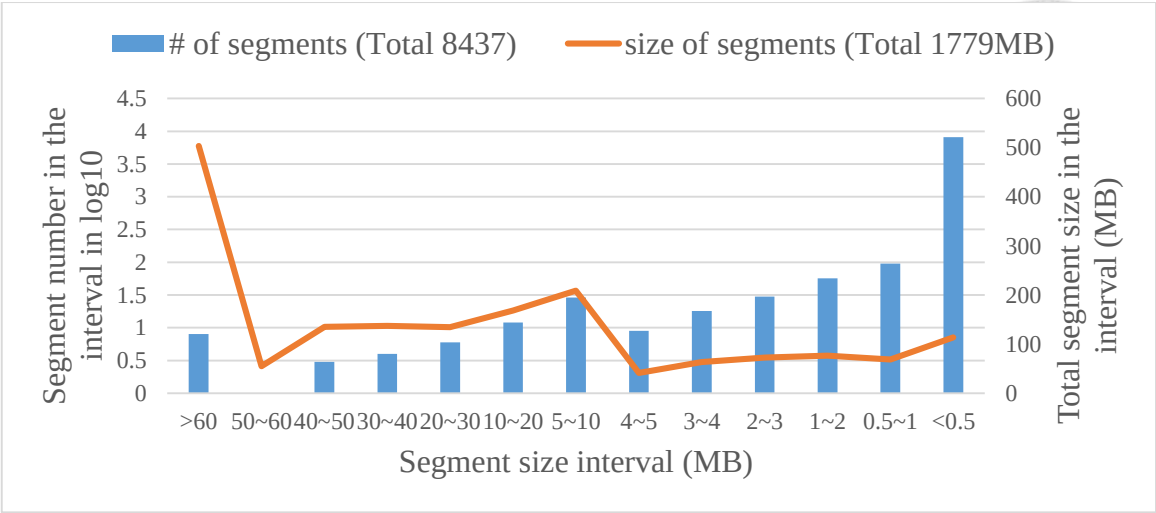
這四個 Query 都針對同一天網路流量，計算節點也設定為相同的工作量上限。它們之間差別只在於 NetFlow Record 流量方向、Query 時間長短與相關資料量多寡。表 6 是各 Query 下，待處理之檔案大小的分布與屬於該分類的資料檔案數與資料量總和。我們可以知道小於 0.5MB 的檔案數佔絕大部分。

(file #,size (MB)) size	Q1	Q2	Q3	Q4
>60 (MB)	(8, 503.32)	(8, 503.3)	(10, 629.15)	(12, 755.0)
50~60	(0, 0)	(1, 55.08)	(0, 0)	(1, 55.08)
40~50	(1, 41.1)	(3, 135.3)	(5, 230.91)	(3, 135.3)
30~40	(5, 181.83)	(4, 137.4)	(6, 215.6)	(5, 171.73)
20~30	(8, 196.78)	(6, 134.33)	(9, 222.05)	(6, 134.33)
10~20	(12, 179)	(12, 168.49)	(13, 198.65)	(13, 183.69)
5~100	(20, 145.55)	(29, 208.61)	(20, 145.55)	(31, 221.26)
4~5	(12, 56.17)	(9, 41.25)	(12, 56.17)	(10, 45.52)
3~4	(18, 62.77)	(18, 63.54)	(18, 62.77)	(18, 63.54)
2~3	(24, 59.86)	(30, 72.74)	(24, 59.86)	(30, 72.74)
1~2	(63, 85.38)	(57, 76.56)	(63, 85.38)	(62, 83.56)
0.5~1	(97, 69.72)	(95, 69.01)	(105, 74.96)	(109, 78.78)
<0.5	(9287, 108.98)	(8165, 113.45)	(14881, 183.85)	(13284, 185.53)
Total	(9555, 1690.4)	(8437, 1779)	(15166, 2164.9)	(13584, 2186.1)

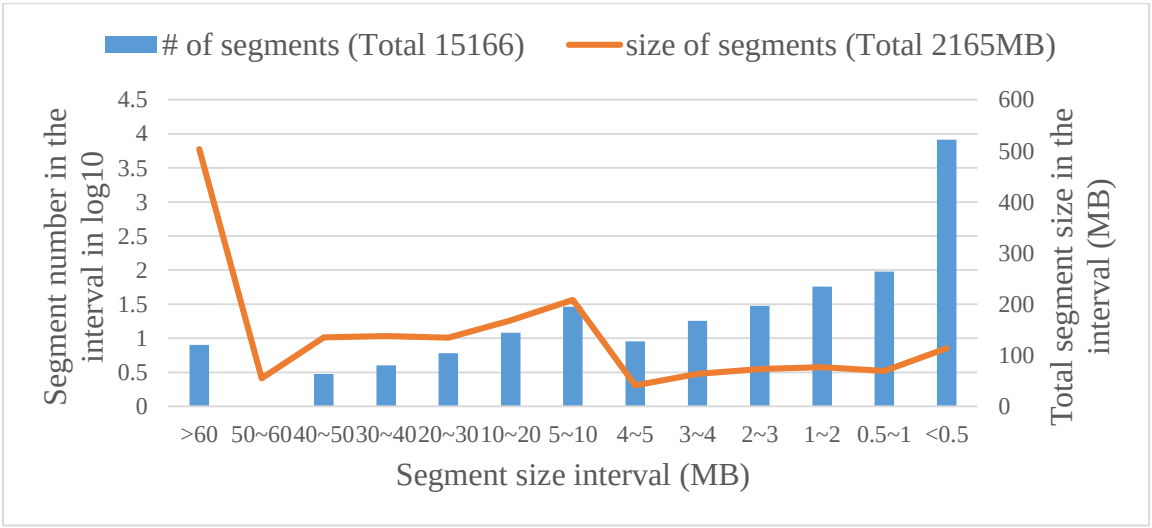
表 6、Q1 至 Q4 其相關資料在不同大小區間的數量與該區間內所有資料總和大小。



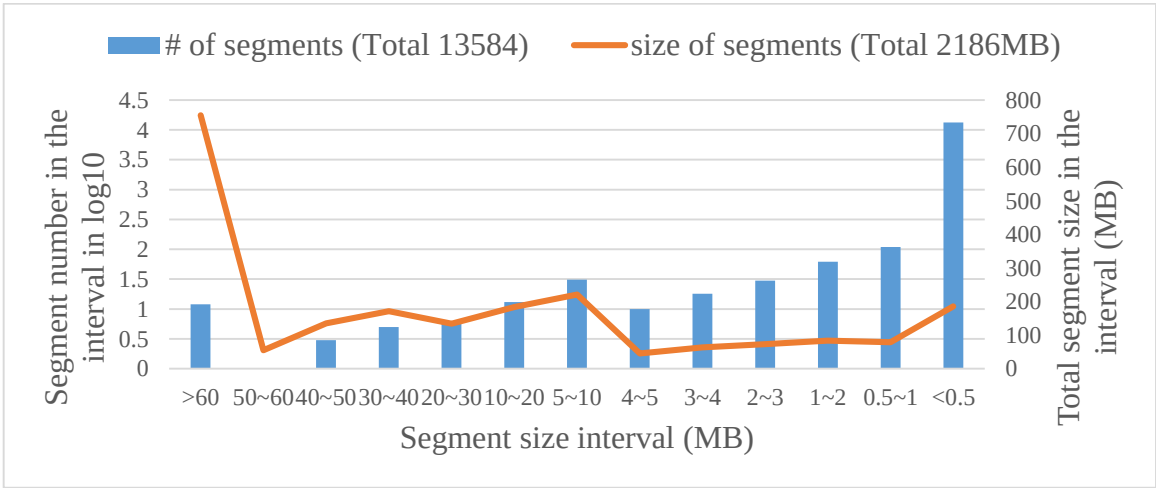
(a) Q1



(b) Q2



(c) Q3

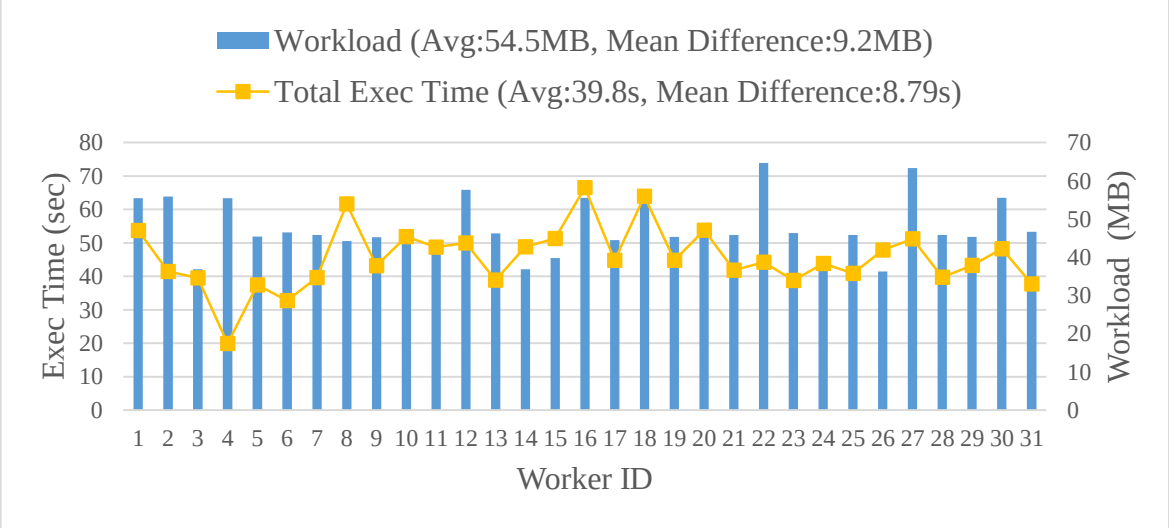


(d) Q4

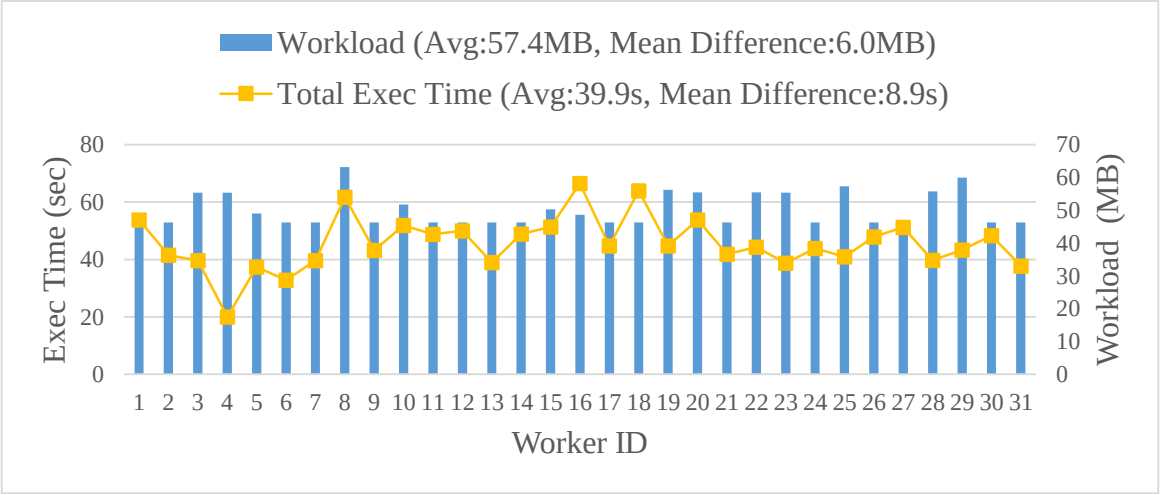
圖 16、Q1 至 Q4 待處理資料大小區間分布圖

C. 結果討論

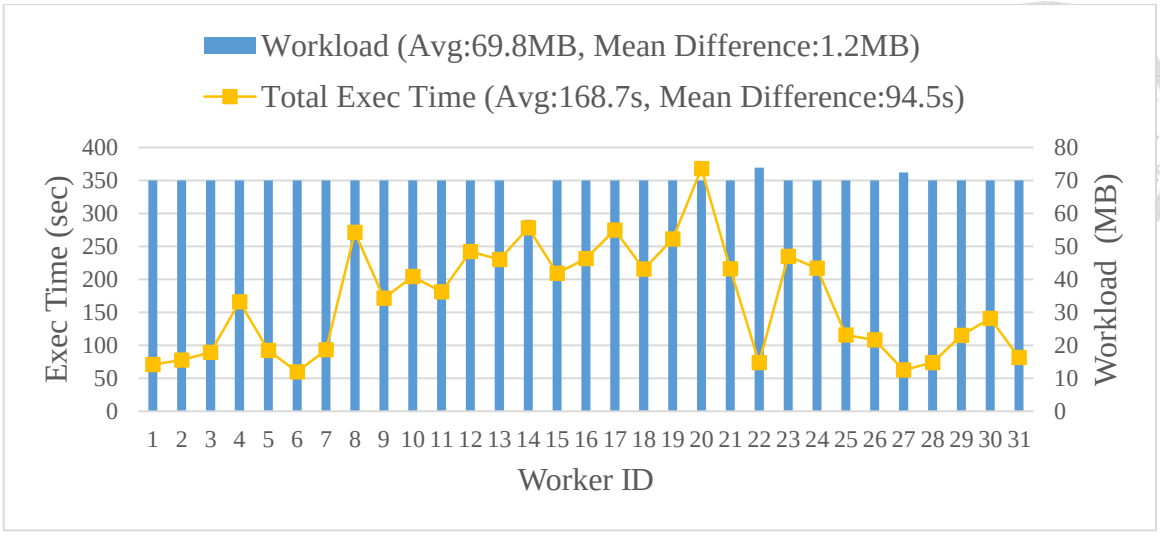
四個 Query 執行結果如所示。圖中列出 31 個計算節點各自所分配到的工作量總量、完成處理的執行時間。結果顯示所提出的 Algorithm 1 並不如我們預期能夠在企圖平均分攤資料總量上達到平衡計算節點間執行時間。我們可以在 Q3 的執行結果中看到，就算 Algorithm 1 能夠找到每個計算節點幾乎相同工作量的分配方式，但是執行時間卻是無法平衡，這說明只考慮分配的資料總量並不能單一影響執行時間，可能有其他因素共同影響執行時間。



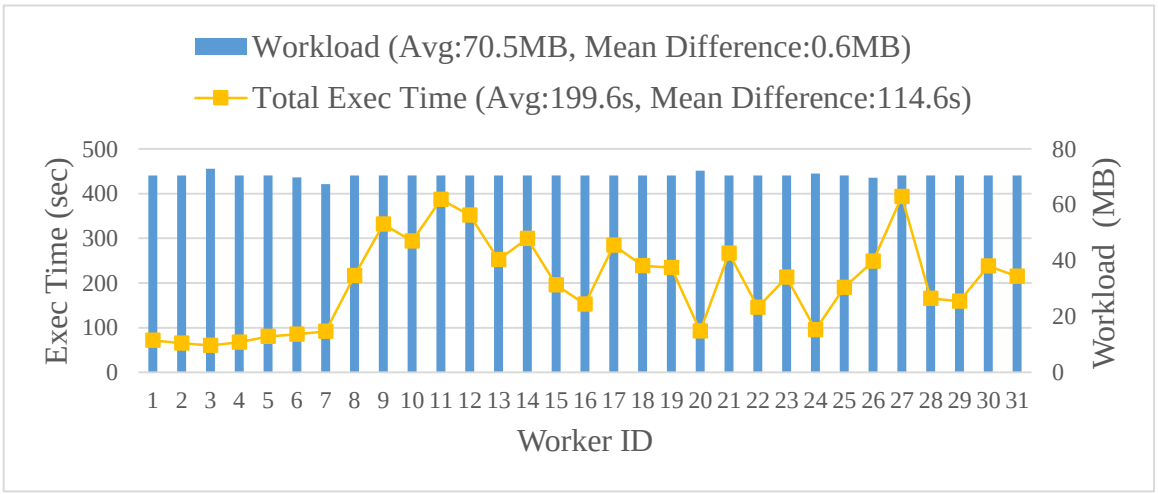
(a) Q1



(b) Q2



(c) Q3



(d) Q4

圖 17、Q1 至 Q4 各計算節點分配之待處理資料量與執行時間

3.5 檔案資料特性

(file #,size (MB)) size	Q1	Q2	Q3	Q4
<0.5	(9287, 108.98)	(8165, 113.45)	(14881, 183.85)	(13284, 185.53)
Total	(9555, 1690.4)	(8437, 1779.0)	(15166, 2164.9)	(13584, 2186.1)
Percentage	(97%, 6.4%)	(96%, 6.4%)	(98%, 8.5%)	(97%, 8.5%)

表 7、Q1 至 Q4 之待處理資料其大小小於 0.5MB 的數量與資料量總合佔所有檔案的比例

在實際執行過程中，並不是一個資料檔案內所有的 NetFlow Record 都會加入所欲建立的 BRT 樹，只有那些跟 Query 有關的才需要加入 BRT。表 7 顯示在這四個 Queries

小於 0.5MB 的資料佔了超過 95%。這些小容量的資料檔案，因為目前資料存放的設計是以國家為第一層分類以下是 AS，再來是日期，因此每一個小檔案都會屬於某一個國家下某一個 AS。這些小容量的資料檔案儲存該 AS 一天中全部流量，像 Q1 與 Q3 時間範圍並不是一整天 24 小時，使用完整資料大小作為分配依據勢必會造成分配結果與執行時間的誤差。

因此，我們考慮資料所涵蓋時間範圍與 Query 時間範圍的交集，以及網路在不同時間會有不同的需求，依照網路流量的 CDF 趨勢方程式來納入工作量計算與分配的考量。

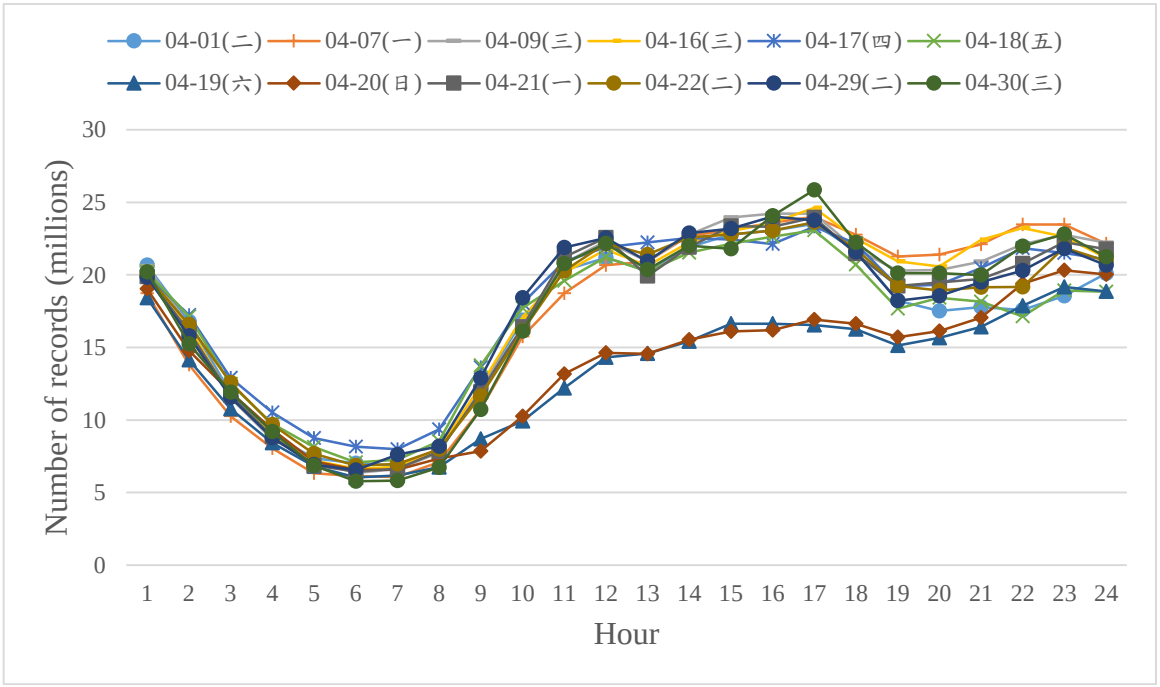


圖 18、12 天網路流量資料在 24 小時內 NetFlow Record 數量

圖 18 畫出蒐集到 12 天的 24 小時網路流量，圖中每一條線呈現某一天內的流量變化，我們可以觀察到網路流量的兩種行為模式 (Pattern)，可被大略分類為平日與假日：白天（上午 8 點到下午 6 點）平日和假日的流量表現不同，但其餘時間差不多；此外 6 點後的網路流量回升幅度略有所不同，大致趨勢相同。

我們對兩種行為模式內網路流量取平均以計算在該小時內平均有多少網路流量，找出該類網路流量的大致表現，接著以平均流量的 CDF 求出 CDF 的趨勢方程式，目的是利用此方程式重新估計待處理資料的大小。估計方式如下：將資料涵蓋時間範圍依據 CDF 趨勢方程式計算在這段時間內紀錄了多少比例的網路流量。同樣的，將 Query 時間範圍與資料涵蓋時間範圍取交集後依據 CDF 趨勢方程式計算該時間內累積網路流量比例，接著除上資料流量比例就可以知道大概有多少部份的資料實際需要處理，接著我們利用估計資料大小再進行一次實驗。

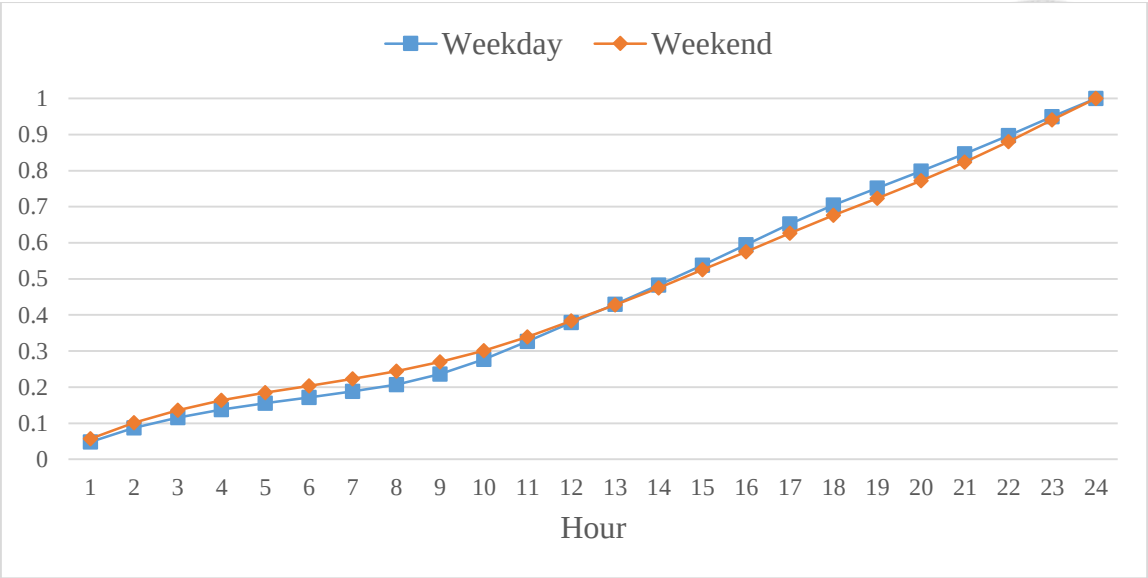


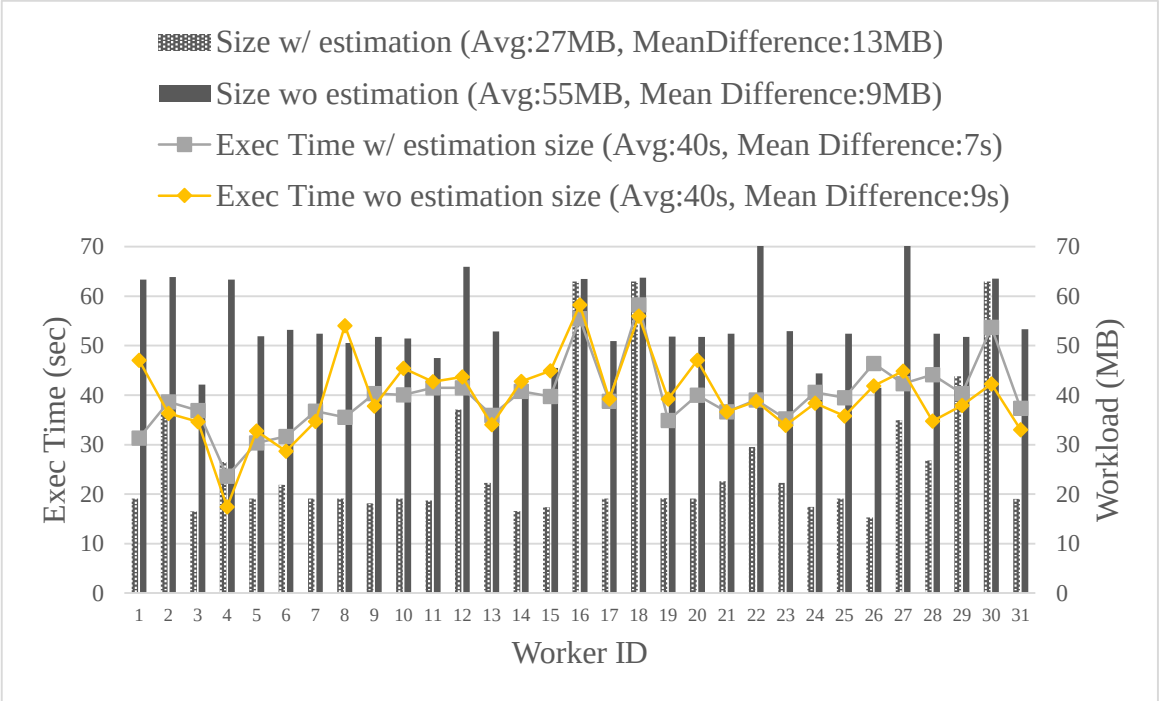
圖 19、屬於假日與平日的網路流量，兩種行為模式的平均流量 CDF

表 8 列出 Q1 與 Q2 所有計算節點待處理資料量依據未估計與有估計的資料量所完成的分配，我們可以觀察到這兩個 Queries 所囊括時間範圍只有檔案資料包含時間範圍的一半，這代表許多原先估計的資料有很大部份並不需要處理。在圖 20 顯示 Q1 與 Q2 使用估計後資料大小進行工作分配的執行時間，大部分計算節點待處理資料量有因著 Query 時間範圍而下降，不過仍有少數大國家因其資料量接近 63MB 導致處理該國家資料的計算節點工作量幾乎沒有變動，例如 Q1 中的 W_{16} 與 W_{18}。

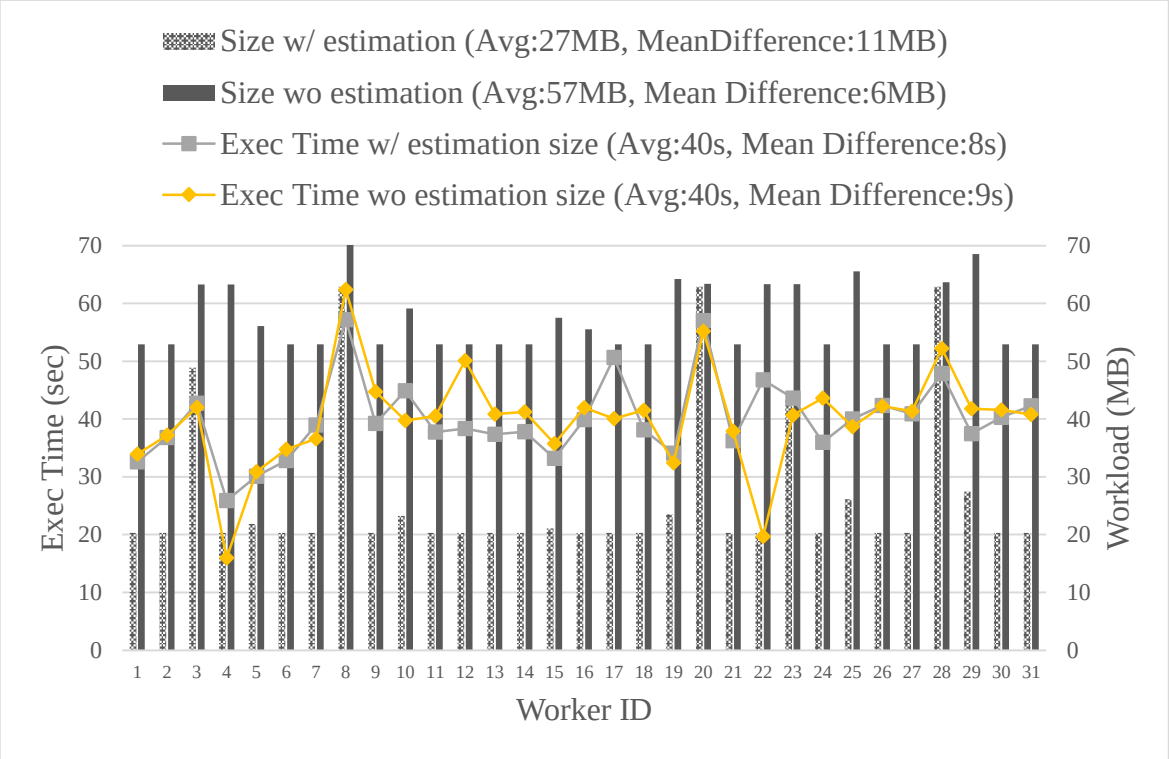
藉由更準確的估計資料大小，Q1 的 Mean Difference 從 8.79 秒下降到 7.3 秒而 Q2 的 Mean Difference 從 8.93 秒下降到 7.52 秒，不過其下降幅度有限。同時在圖中，我們仍然觀察到各計算節點執行時間與待處理資料量並沒有直接關係，也就是尚有其他因素影響執行時間。

(MB) Worker ID	Q1		Q2		(MB) Worker ID	Q1		Q2	
Estimation	w/	wo	w/	wo		w/	wo	w/	wo
W_1	63.36	19.08	52.94	20.29	W_{17}	50.89	19.08	52.94	20.29
W_2	63.86	36.54	52.94	20.29	W_{18}	63.70	62.91	52.94	20.29
W_3	42.12	16.51	63.28	48.90	W_{19}	51.85	19.17	64.25	23.54
W_4	63.35	26.50	63.29	20.29	W_{20}	51.79	19.08	63.42	62.91
W_5	51.91	19.09	56.07	21.83	W_{21}	52.40	22.58	52.94	20.29
W_6	53.16	21.90	52.94	20.29	W_{22}	73.91	29.52	63.33	20.29
W_7	52.42	19.08	52.94	20.29	W_{23}	52.96	22.24	63.32	43.37
W_8	50.55	19.08	72.23	62.91	W_{24}	44.37	7.40	52.94	20.29
W_9	51.76	18.16	52.94	20.29	W_{25}	52.42	19.08	65.54	26.11
W_{10}	51.42	19.09	59.15	23.22	W_{26}	41.48	15.27	52.94	20.29
W_{11}	47.48	18.71	52.94	20.29	W_{27}	72.43	34.96	52.94	20.29
W_{12}	65.89	37.07	52.94	20.29	W_{28}	52.42	26.78	63.68	62.91
W_{13}	52.85	22.25	52.94	20.29	W_{29}	51.78	43.80	68.55	27.43
W_{14}	42.14	16.55	52.94	20.29	W_{30}	63.51	60.19	52.94	20.29
W_{15}	45.46	17.37	57.51	21.09	W_{31}	53.30	19.05	52.94	20.29
W_{16}	63.45	62.91	55.52	20.29	Total	1690	824	1779	830

表 8、各計算節點依據未估算與有估算的資料大小所分配到待處理資料量



(a) Q1



(b) Q2

圖 20、Q1 與 Q2 中各計算節點依據未估計與估計後資料大小所分配待處理資料量與執行時間

3.6 計算節點之實體結構

在實際的實驗環境，我們觀察到可能造成執行時間不平衡的另一可能因素是因為虛擬機器間資源競爭所導致。實驗裡每台實體機器設定有八個計算節點，每個節點的 vCPU 與 vMemory 都有給予專屬使用量，且令此資源的分配在處理資料時都不會用完，也就是說這兩項資源不會是 Bottleneck。但是我們目前採用 commodity PC 的架構下一台實體機器只有一顆硬碟、一個讀寫頭，當一台實體上的所有 VM 們皆需要同時讀取硬碟資料時就會讓 Disk I/O 變成是一個瓶頸，導致同一台實體機器上的計算節點就算擁有相同待處理資料量也會有完成時間的差異。

表 9 中列出 Q1 與 Q2 中各實體機器上計算節點執行完成順序。我們可以在 Q1 的實體機器 1 上發現到就算計算節點 5 與計算節點 7 被分配到相同的待處理資料量但其執行時間卻相差約 23%；而 Q2 的實體機器 1 上計算節點 4 與計算節點 2 也有同樣的狀況發生。這狀況正說明了就算各計算節點擁有相同計算資源與工作量的情況下，但因 Disk I/O 的競爭導致各計算節點有先後執行完成的狀況。

Phy ID	Earliest							Latest
1	W_4(27,24)	W_5(19,30)	W_1(19,31)	W_6(22,32)	W_7(19,37)	W_3(17,37)	W_2(37,39)	
2	W_8(19,36)	W_{13}(22,36)	W_{15}(17,40)	W_{10}(19,40)	W_9(18,40)	W_{14}(17,41)	W_{12}(37,41)	W_{11}(19,41)
3	W_{19}(19,35)	W_{23}(22,35)	W_{21}(23,37)	W_{17}(19,39)	W_{22}(30,39)	W_{17}(19,39)	W_{20}(19,40)	W_{16}(63,55)
4	W_{31}(19,37)	W_{25}(19,39)	W_{29}(44,40)	W_{24}(17,41)	W_{27}(35,42)	W_{28}(27,44)	W_{26}(15,46)	W_{30}(63,54)

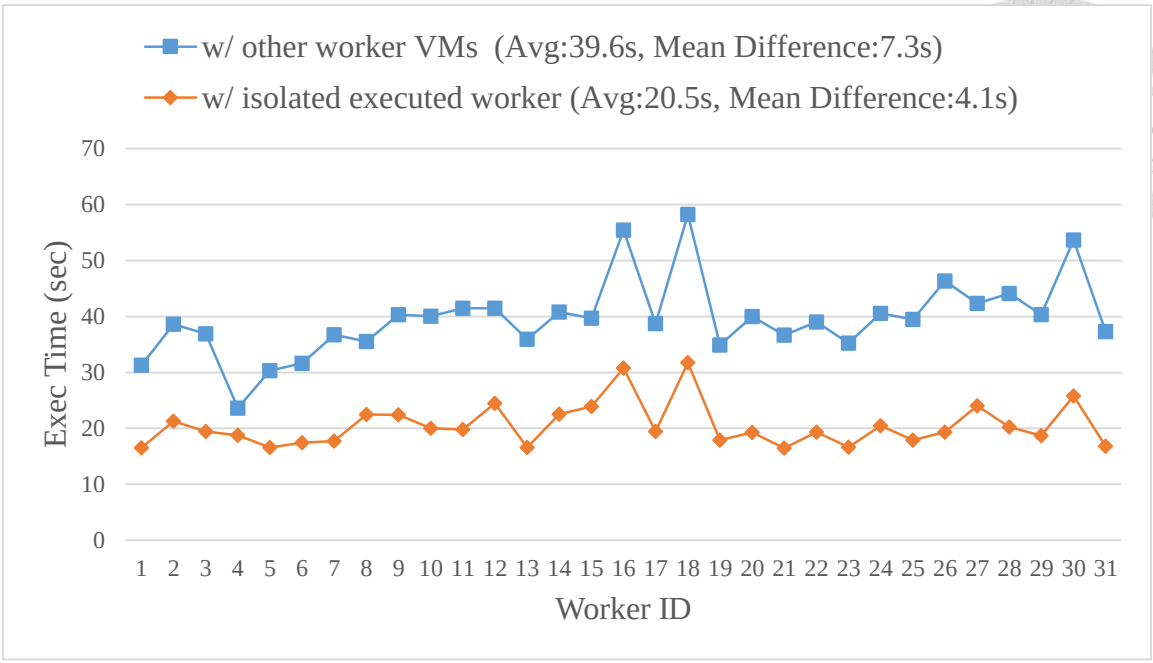
(a) Q1

Phy ID	Earliest							Latest
1 (1-7)	W_4(20,26)	W_5(22,30)	W_1(20,33)	W_6(20,33)	W_2(20,37)	W_7(20,39)	W_3(49,43)	
2 (8-15)	W_{15}(21,33)	W_{13}(20,37)	W_{11}(20,38)	W_{14}(20,38)	W_{12}(20,38)	W_9(20,39)	W_{10}(23,45)	W_8(63,57)
3 (16-23)	W_{19}(24,34)	W_{21}(20,36)	W_{18}(20,38)	W_{16}(20,40)	W_{23}(43,44)	W_{22}(20,47)	W_{17}(20,51)	W_{20}(63,57)
4 (24-31)	W_{24}(20,36)	W_{29}(27,37)	W_{25}(26,40)	W_{30}(20,40)	W_{27}(20,41)	W_{31}(20,42)	W_{26}(20,42)	W_{28}(63,48)

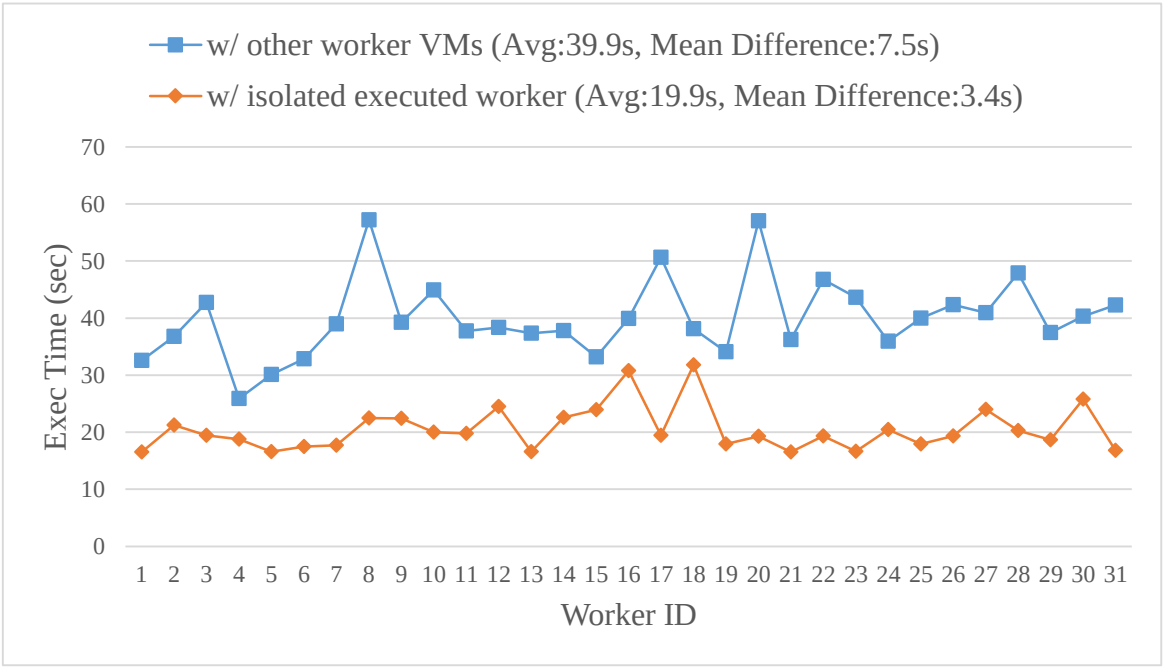
(b) Q2

表 9、Q1 與 Q2 其實體機器上各計算節點執行完成順序，從最早排到最後 (Workload in MB, Exec Time in Sec)

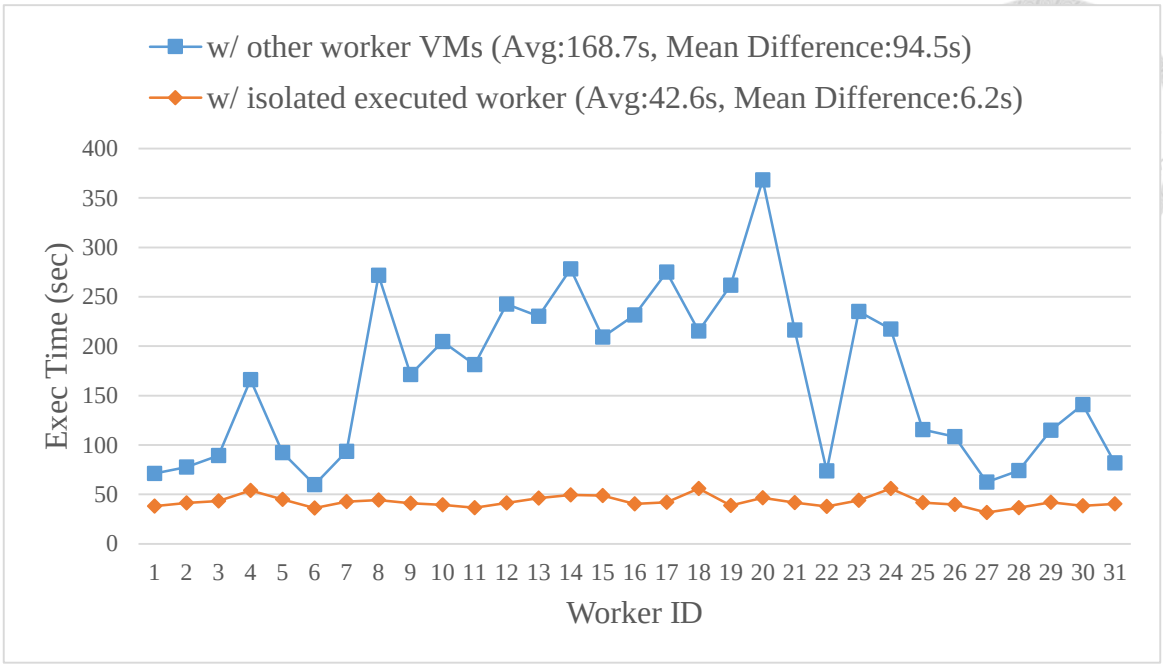
為了了解實體機器上各虛擬機間 Disk I/O 存取競爭所造成影響，我們設計了一個實驗，讓每一台虛擬機的計算節點在沒有其他虛擬機的情形下獨立執行，以測量得該計算節點於隔離環境下的執行時間，視為最基本（最小）的執行時間，作為 Baseline 參考，以釐清 Disk I/O 競爭影響。每台計算節點的工作量依然不變，只是現在每台實體機器一次只會有一個計算節點執行，比較結果顯示在圖 21。


w/ other worker VMs (Avg:39.6s, Mean Difference:7.3s)
w/ isolated executed worker (Avg:20.5s, Mean Difference:4.1s)
Exec Time (sec)
0
10
20
30
40
50
60
70
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Worker ID

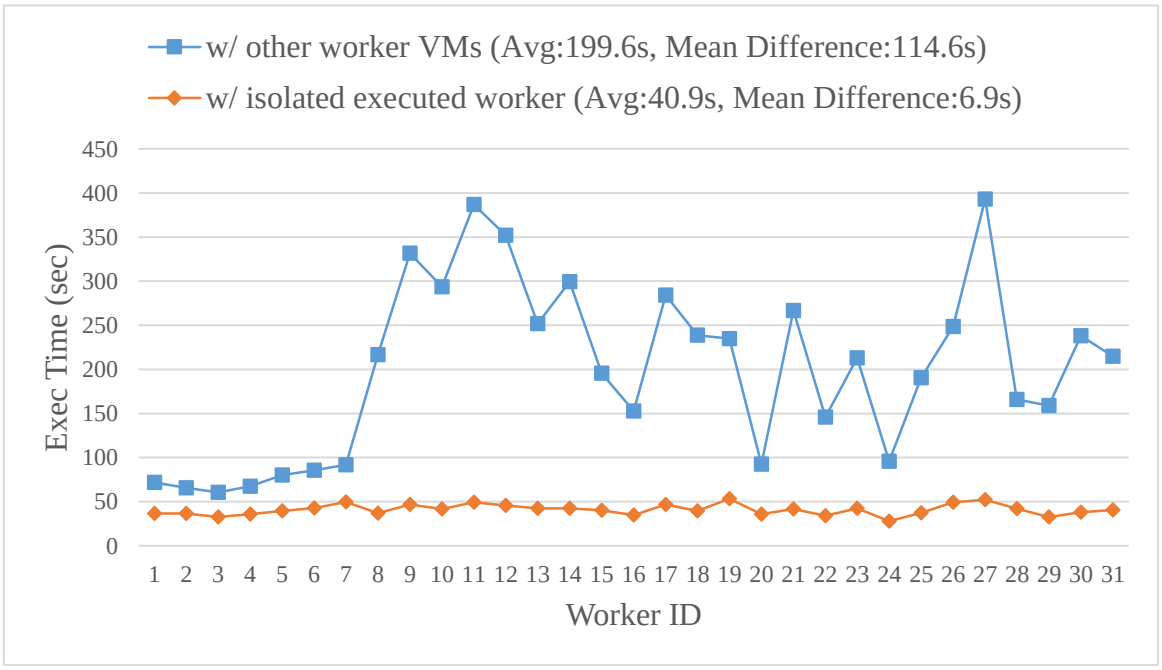
(a) Q1


w/ other worker VMs (Avg:39.9s, Mean Difference:7.5s)
w/ isolated executed worker (Avg:19.9s, Mean Difference:3.4s)
Exec Time (sec)
0
10
20
30
40
50
60
70
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Worker ID

(b) Q2



(c) Q3



(d) Q4

圖 21、Q1 到 Q4 中，各計算節點在共同執行與獨立執行時其執行時間之比較

Compution time(sec) / Query ID	Based on size (MB), multi-worker				Based on Estimated number of Records, multi-worker				Based on Estimated number of Records, isolated execution (baseline)			
	Avg	Std	Max	**MD**	Avg	Std	Max	**MD**	Avg	Std	Max	**MD**
Q1	39.8	7.9	58.2	**8.8**	39.6	6.9	58.2	**7.3**	20.5	3.8	31.8	**4.1**
Q2	40.0	8.6	62.4	**8.9**	40.0	6.8	57.2	**7.5**	19.9	3.4	29.6	**3.4**
Q3	168.7	81.2	368.3	**94.5**	N/A				42.6	5.6	56.1	**6.2**
Q4	199.6	97.4	393.2	**114.7**	N/A				40.9	6.0	53.4	**7.0**

表 10、Q1 至 Q4 在不同資料處理量計算方式作為工作分配依據下各計算節點的執行時間比較

表 10 整理了個計算節點執行時間在資料工作量分配方式調整前與調整後以及各自在單獨隔離(isolated execution environment)執行下（也就是仍然是 virtualized environment 但是只有一個ＶＭ）的執行時間以及 Mean Difference 的統計資料。我們可以看到在移除 I/O 競爭的因素後，計算節點的執行時間不僅大幅度下降也更加的平衡（尤其是當待處理資料量從 12 小時增加到 24 小時愈多時）：Q1 利用調整後大小執行時間的 Mean Difference 從 7.3 秒下降到 4.08 秒；Q2 利用調整後大小執行時間的從 7.52 秒下降到 3.4 秒；Q3 的從 94.53 秒下降到 6.19 秒；Q4 的從 115 秒下降到 6.96 秒。從最大執行時間的結果看來，I/O 競爭因素的影響很大。總結而言，本論文所提出的 Algorithm 1 能透過平均分配工作量來平衡執行時間，但是如何把可能的 I/O 競爭考慮進來以達相似執行時間是接下來要解決的議題。

文獻上，解決 VM 同時執行對硬體所帶來的影響 (Impact)，提出兩種可能的方式：第一、Changing Data Storage System：在[24]此篇論文中， 對於如何在 HDFS 下有效處理小容量檔案，作者認為 MapReduce 等 Parallel Programming Framework 的下層資料儲存系統應該混合 HDFS 與 Parallel DBMS (Database Management System)，讓它們負責儲存各自擅長的檔案類型：大容量的資料交由 HDFS 處理；小容量的就儲存在 Parallel DBMS 中。Parallel DBMS[23] 與傳統 DBMS 的差異在於在前者的架構下資料由一群位在相同地理區域的機器所組成的 Cluster 共同儲存；除了共同儲存外，當接收 Query 後所有的機器會同時、平行的處理。

與 Parallel DBMS 相似的架構還有 Distributed DBMS，但因為其機器的分佈位置並沒有位於相同地理區域，所以彼此尚保有高度自主性以自行處理 Query。Parallel DBMS 為解決由 NN 處理過多小容量檔案的 Request 所造成的 Overhead 問題，作者提出 Fat-Btree [25]－一種自 Parallel B-tree 改良而來的資料結構來儲存 DB index。在 Fat B-tree 中所有檔案會被組織成一顆 Level 3 的樹狀結構，一個根節點、數個中間節點以及葉節點。葉節點與中間節點相連並記錄對應資料的相關資訊；一台機器可以擁有多

個中間節點端看其儲存的資料數量。Fat-Btree 稱為 Parallel 的原因在於這棵 Index Tree 並不會只由一台機器來維護，每台機器都會擁有 Fat-Btree 的一個子樹－包含它所儲存資料的葉節點和跟葉節點連結的中間節點。這顆子樹就如同 Local Cache 一般，當機器需要存取檔案時可以先查詢子樹，以避免每次向 NN 查詢的 Overhead。把檔案資訊組織成樹狀結構有利於資料搜尋，當使用二元搜尋 (Binary Search)演算法時，搜尋檔案的複雜度可從原先循序搜尋的 $O(n)$ 下降到 $O(\log(n))$，進而加快存取檔案的速度。改善後的系統不僅提升 HDFS 在處理小容量資料的效率，也讓 NN 負擔大為下降進而減少 Single point of failure 發生的機會。

第二、Merge files and Redesign the metadata in NN (NameNode)：在[26]此篇論文中，作者指出 HDFS 原本設計用以儲存、管理大容量資料，因此當需要從 HDFS 內讀取太多小容量檔案時就會遇到兩種問題。一是、同時有太多向 NN 索取 Metadata (檔案位置等)的 Request 而讓 NN 的網路頻寬大量消耗，也會導致 NN 效率下降；整體執行時間延長。其解決問題的方法是讓 Data Node 擁有關於本地 (Local)儲存 Blocks 的 Metadata，當 Worker 需要讀取檔案時可以先從 Local Metadata 內尋找，若是沒有再跟 NN 做查詢。如此就不需要每次都發 Request 向 NN 查詢，減少 NN Overhead 同時也避免網路傳輸導致的時間延遲。

其二為 DataNode 硬碟會因為檔案位置不同而需要不斷移動讀寫頭導致 I/O Overhead。其改進方法是改善 HDFS 內儲存 Block 方式，使檔案數目減少以降低移動次數。一般而言，檔案儲存進 HDFS 時會切割成一連串固定大小的 Blocks 後儲存，但是通常最後一個 Block 會放不滿卻依然單獨儲存為一個 Block。為使檔案數目減少，作者提出把所有不足固定大小的 Blocks 合併，同時修改 Metadata 格式，讓 Metadata 紀錄合併後 Block 中原本 Block 的檔名與長度，Clients 能透過 Offset 直接存取所需檔案。如此在 Client 不需改動太多存取方法同時減少 Blocks 數量，讓 HDFS 更加 Consolidated。

作者認為合併時不應該是隨意合併而應從檔案結構或是邏輯上相關性下手。以本論文而言，因為一個目標網路 (target network domain)其通聯的網路對象其實是不平均的，事實上以我們的實驗對象而言，超過 200 個國家中只有前幾個有較多的通聯紀錄。

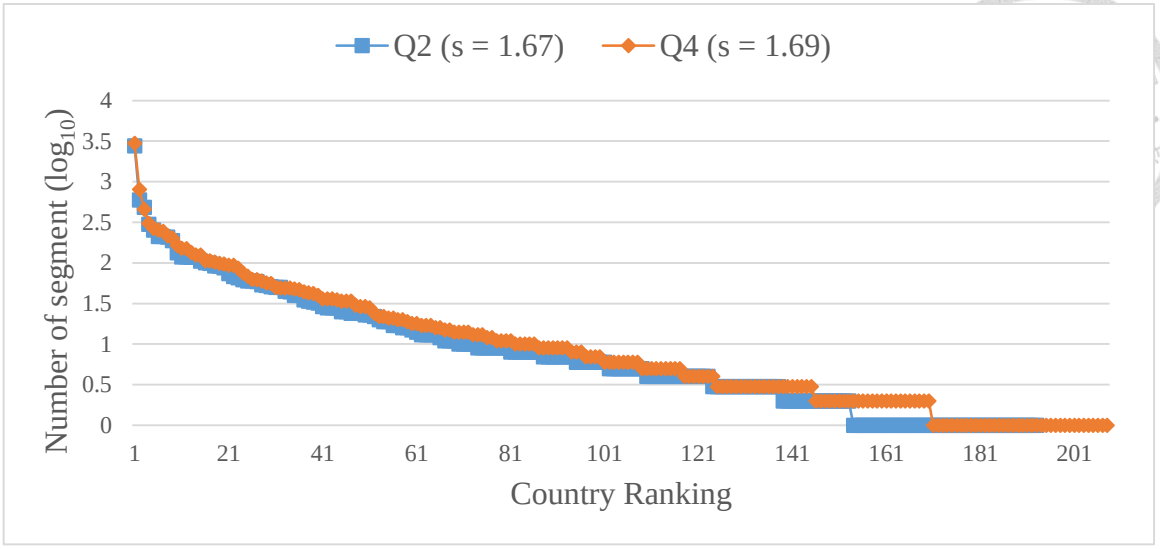


圖 22、Q2 與 Q4 中各國家擁有資料檔案數量，由大到小排序

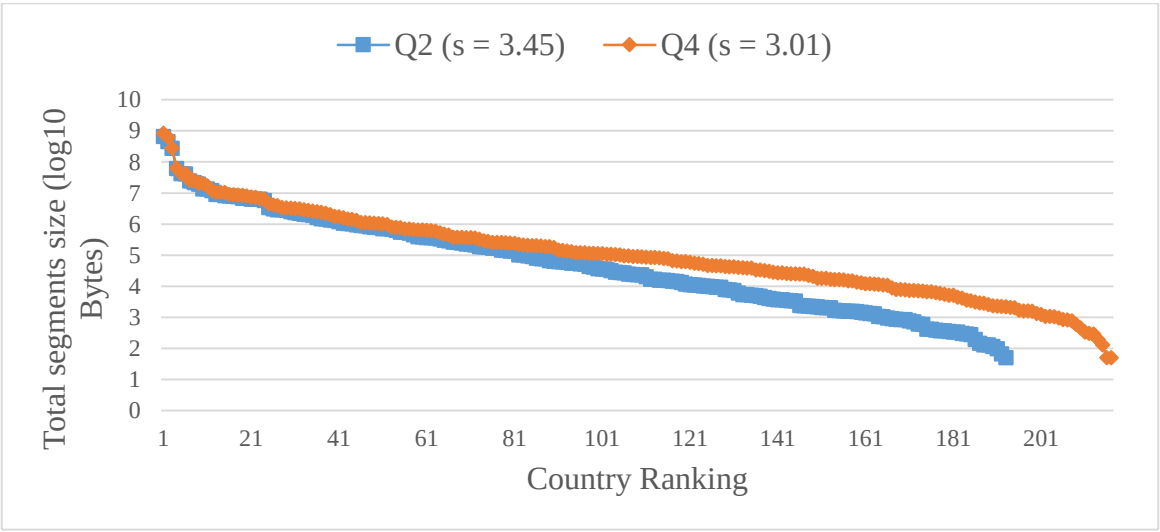


圖 23、Q2 與 Q4 中各國家擁有資料檔案大小總量，由大到小排序

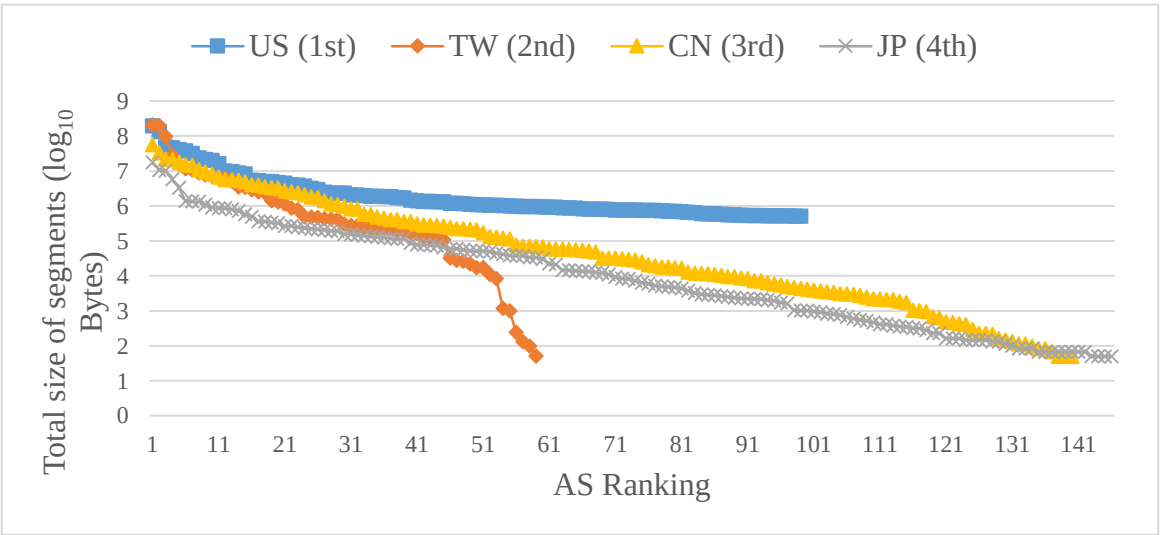


圖 24、Q4 中，對待處理資料總量前四名國家，以各國下各 AS 待處理資料量的排序

在本論文所討論的議題，就如表 7 所示各 Query 下待處理資料的檔案大小小於 0.5MB 的檔案個數至少佔了全部數量 95%。因此我們思考將小容量國家內不同 AS 的流量資料合併為一個檔案，藉由大幅減少檔案數量以降低 disk I/O 讀取競爭次數，以改善執行時間。在本論文中 HDFS 預設 Block 大小為 63MB，所以當一個國家內 AS 大小的加總小於 63MB 時，我們就視為小容量國家。圖 22 與圖 23 列出將排序後的資料數與資料量 fit Zipf Distribution 所得參數 s，當 s 越接近 1 時越 fit Zipf Distribution。在圖中我們可以看到，不管是以數量或是總量的角度而言，資料都集中在少數國家，這呼應了網路流量的長尾現象。因此我們根據資料總量設定一個門檻 (63MB)，對總量超過與不足的國家進行不同的處理。

所以，我們對流量資料少的國家的 AS 合併，當 Query 並非針對特定 AS 時如此合併可以讓該國家檔案的讀取次數從原先 AS 數量降為只要一次即可，就算是針對 AS 的 Query 也能利用 Offset 直接讀取。此外，從圖 24 也發現到那些流量資料多國家內各 AS 的資料總合也只有少數 AS 擁有較多的通聯紀錄，因此對於流量資料多國家我們也可以採用相同的作法進行 AS 合併，把小於 63MB 的 AS 盡量合併成接近 63MB 的 Block。這樣一來又近一步大大降低資料數量，以減少計算節點間 Disk I/O 讀取競爭頻率。

# of total segments	<63MB			>63MB			Total		
	Before merging	After merging	Reduction Ratio	Before merging	After merging	Reduction Ratio	Before merging	After merging	Reduction Ratio
Q1	5860	194	97%	3696	33	99%	9556	217	97%
Q2	5429	190	97%	3008	25	99%	8437	215	97%
Q3	9606	213	98%	5561	32	99%	15167	245	97%
Q4	9065	213	97%	4519	35	99%	13584	248	98%

表 11、Q1 至 Q4 之待處理資料在合併前後 AS 的數量與減少比例

# of countries	<63MB	>63MB	Total
Q1	194	4	198
Q2	190	3	193
Q3	213	4	217
Q4	213	4	217

表 12、Q1 至 Q4 之待處理資料總量大於 63MB、小於 63MB 與全部的國家數

segment size (GB)	<63MB	>63MB	Total
Q1	0.28911	1.40129	1.69041
Q2	0.33246	1.44656	1.77901
Q3	0.33985	1.82499	2.16485
Q4	0.38940	1.79671	2.18611

表 13、Q1 至 Q4 之待處理資料總量

表 11 中可以看到完成合併後之檔案總數大約只剩下原本的 3%，而針對流量較大的前四名國家 (分別是美國、台灣、中國、日本，依名次排序)其數量更下降至 1%。我們對資料進行合併處理後，其大小作為分配依據的執行結果以 Q4 為例呈現在圖 25，大部分計算節點執行時間因此而下降，整體執行時間更為平衡。平均執行時間由 119.58 秒下降到 67.64 秒；Mean Difference 從 114.65 秒下降到 15.3 秒，約下降了 85%。藉由以上說明，我們確實了解到透過降低檔案數量能夠減少計算節點間 Disk I/O 讀取競爭，進而平衡節點間執行時間，達到 Algorithm 1 的最終目的：藉由分配相同工作量，以期各計算節點能在差不多的時間內完成。不只是在一台實體機上多個虛擬計算節點的架構下，計算節點間的 disk I/O 競爭減少，有效降低各節點的執行完成時間；並且，大部分計算節點在單獨隔離的執行環境下的執行完成時間也減少。

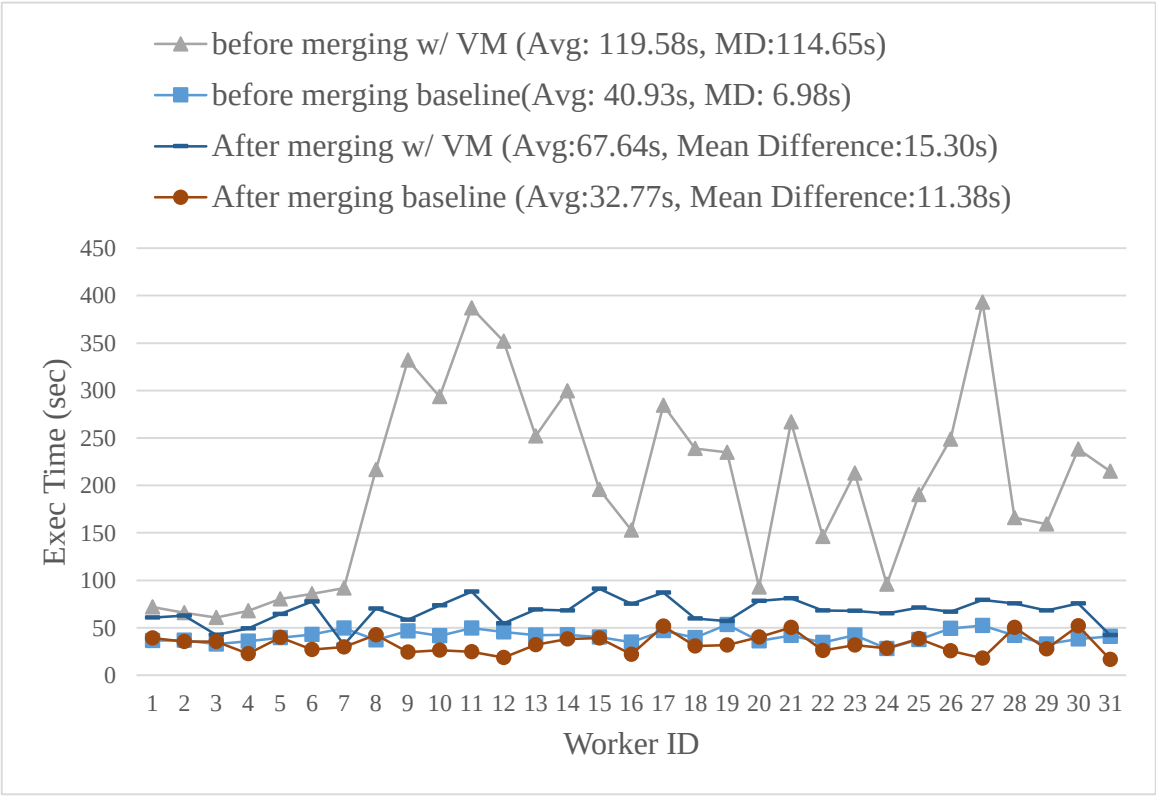


圖 25、Q4 在檔案合併前後其執行時間與 Baseline 之比較

(sec) / Query ID	Multi-worker, based on size (MB)				Baseline, based on Estimated number of Records				Multi-worker, based on size with file merging				Baseline, based on size with file merging			
	Avg	Std	Max	**MD**	Avg	Std	Max	**MD**	Avg	Std	Max	**MD**	Avg	Std	Max	**MD**
Q4	199	97.4	393	**114**	40.9	6.0	53.4	**7.01**	67.6	13.1	91.3	**15.0**	32.8	9.75	51.9	**11.4**

表 14、Q4 在合併小檔案前後之工作完成時間以及與 Baseline 之比較

合併小檔案的做法會讓資訊的顆粒度 (Granularity)下降：原先能夠透過路徑得知該筆資料隸屬的國家與 AS，但是合併後可能需要透過 Metadata 以辨識找出在這查詢下待處理的資料檔案藉以換取處理的效率。在下一個小節中我們將說明如何在檔案合併後依然保有原先的資訊。

3.7 Hierarchical Path 改進

當我們將國家內 AS 合併之後，原本設計的 Hierarchical Path 就不再適用。以前，依照國家－AS－日期的規則我們可以根據 Query 拼湊出檔案的路徑，以快速讀取特定的 AS 檔案。但現在將 AS 合併後，原有的路徑規則修改為國家－日期(－AS)：若是有 AS 的網路流量總合超過 63MB，則並不會做合併且保留該 AS 的資料夾。

修改路徑規則後，我們需要額外的 Metadata 來紀錄哪些 AS 有被合併。若該 AS 有被合併則需紀錄合併在第幾個合併檔案中；此外 AS 合併後也需額外 Metadata 紀錄在合併檔案中位置以便利用 Offset 快速讀取。第一種紀錄哪些 AS 有被合併的 Metadata 稱為 MergeMeta；另一種 Metadata 則是每個合併檔案都會有，裡頭紀錄每個 AS 資料在合併後檔案中的 Offset 與原本資料其 Metadata 內容。

現在的查詢流程為，首先從 Merge Metadata 中查詢 Query AS 是否有被合併，若沒有則依照國家－日期－AS 的規則在資料夾中尋找滿足 Query、需要處理的資料；若被合併，根據 Merge Metadata 內紀錄合併到的檔案，再去其對應的 Metadata 中找出該 AS 的 Offset。相較修改前可以直接拼湊出資料路徑，修改後需要最多兩次的 Table Lookup 才能找出資料位置。雖然需要額外查詢次數，但是我們相信藉由合併小檔案；減少檔案數量所降低的 I/O Overhead，會大於額外查詢的 Overhead。

3.8 資料執行時間過久處理方式

當完成一天的流量之後，接著我們想瞭解當待處理資料量的規模擴大時其處理時間會是如何。因為本研究的目的是想探討規模化資料量與計算節點以及執行時間的關係，以達到互動式查詢的合理回應時間。在此目的下，有兩個議題。第一個議題是在一定的計算節點下，規模化資料量與執行時間的關係。第二個議題是給定最大可接受

的執行時間，規模化資料量與計算節點需求的關係。因此，接下來我們將 Query 的時間範圍擴大至兩天與三天。我們定義兩天與三天的 Query 為 Q5 與 Q6 並以表格比較不同天數的 Query 之檔案數量與資料總量。

Query ID	Direction	Time	Total segments size	Workload capacity per worker
Q4	out-in	0:00-24:00	2.18611GB	200MB
Q5	out-in	24:00-72:00	4.42069GB	200MB
Q6	out-in	0:00-72:00	6.60680GB	300MB

表 15、Q4 至 Q6 之資料

Query ID	Q4			Q5			Q6		
Country	Top 4	Others	Total	Top 4	Others	Total	Top 4	Others	Total
# of total segments	35	213	248	67	427	494	102	640	742
Size of segments (GB)	1.796	0.389	2.186	3.593	0.827	4.420	5.390	1.216	6.606

表 16、Q4 至 Q6 之待處理資料檔案數與總量前四大之國家與其他國家

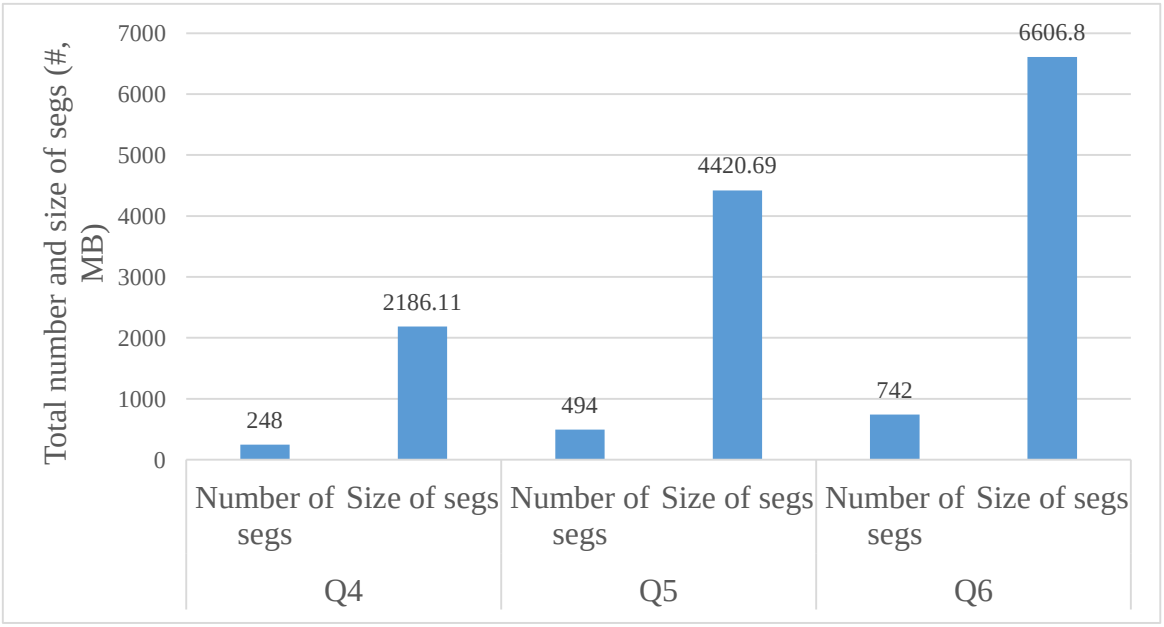


圖 26、Q4 至 Q6 之待處理資料數量與總量

在實驗過程中，無論是一天、二天或是三天，我們遇到有時候某個計算節點會遇到在處理某一個檔案時，不明原因地執行停滯 (stuck)，在其他大部分的計算節點都已

經完成後，這些節點仍在停滯中，無法完成工作，我們稱這些計算節點為落後者(Straggler)。這種計算節點的執行停滯是在處理某個資料檔案 (segments)時會停在程式中某個區段無法繼續往下執行。我們在實驗中發現若我們將那些資料跳過不處理時，落後者就能繼續往下執行並執行完成。此外，這些會卡住的資料只要透過 HDFS 讀取，就算在其他計算節點執行同樣會發生停滯的狀況，但如果直接從本地硬碟讀取的話就不會有停滯的狀況。

	# of stragglers	# of unfinished segments	Size of unfinished segments (MB)
Q4	4	5	252.48
Q5	9	10	354.15
Q6	12	15	606.64

表 17、Q4 至 Q6 未完成資料數量與總量

Average workload (MB)	Stragllers	Non-stragglers	All
Q4	69.75	70.63	70.51
Q5	138.36	144.33	142.60
Q6	211.76	213.98	213.12

表 18、Q4 至 Q6 全部計算節點，落後者與非落後者的平均工作量

Worker ID	6	8	13	29
	3/4	2/8	7/8	10/11

(a) Q4

Worker ID	8	15	16	17	21	26	28	29	30
	2/14	5/8	26/42	1/3	1/12	22/22	18/18	26/26	9/9

(b) Q5

Worker ID	2	4	5	6	8	13	14	17	18	20	23	27
	2/7	8/8	60/62	14/14	27/28	1/12	51/53	17/18	15/38	34/36	21/22	5/35

(c) Q6

表 19、Q4 至 Q6 中，執行停滯時正在處理的資料檔案於各計算節點全部待處理檔案資料之順序（斜線前為未完成資料之名次；斜線後為全部資料數量）。

# of unfinished segments	Top 4 countries	Other countries
Q4	5	0
Q5	5	4
Q6	10	5

表 20、Q4 至 Q6 中未完成資料屬於前四大國家與其他國家的數量

# of unfinished segments	Phy 1	Phy 2	Phy 3	Phy 4
Q4	1	2	0	1
Q5	0	2	3	4
Q6	4	3	4	1

表 21、Q4 至 Q6 中未完成資料在各實體機器的數量

我們列出 Q4 到 Q6 中所有的落後者：在 Q4 中，執行停滯的計算節點有在第一台實體機的 6 號計算節點，第二台實體機的 8、13 號計算節點，第四台實體機的 29 號計算節點共 4 個計算節點；Q5 中，執行停滯的計算節點有在第二台實體機的 8、15，在第三台實體機的 16、17、21，在第四台實體機的 26、28、29、30 共 9 個計算節點；Q6 中，執行停滯的計算節點有第 2、4、5、6、8、13、14、17、18、20、23、27 共 12 個計算節點。

表 18 針對這些計算節點，我們列出落後者與正常執行者的平均工作量，我們可以觀察到以工作量的角度來看落後者與正常執行者是沒有差別的。此外，從表 19 中顯示執行停滯的時間點，有些在剛開始，有些在中間或是執行快完成時都有執行停滯的狀況發生，不過大部分都發生在計算節點執行快結束的時候。我們必須處理執行發生停滯狀況的計算節點，以免導致整體執行時間被大幅度拉長或是根本無法完成，違背 NetActy 的互動性目標。表 20 中，大部分未完成的資料是屬於前四大國家的，但依然難以預測說哪些資料有可能會發卡住的狀況。同時在表 21 中觀察到各實體機器上執行停滯的狀況沒有一定的規律存在，也就是說資料的真實執行狀況是隨機且難以預測，而且整個執行環境是複雜的。文獻[6]，遇到相同問題：MapReduce 在執行時有時候會出現一些執行時間過久的落後者 (Straggler)，而作者為避免解決各式各樣不同的原因而採用當遇到有落後者出現時便在其他擁有資料複本的機器上重新執行該落後者所負責的 Tasks 並捨棄原本落後者負責的那份。

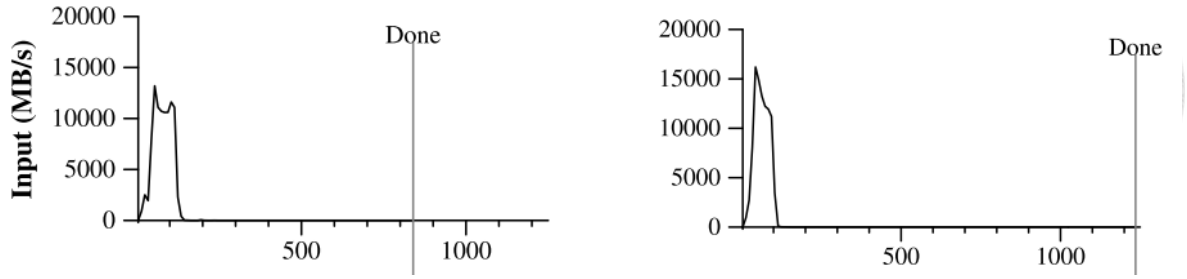


圖 27、在[6]中，兩次 MapReduce Job 中 Map Phase 有處理落後者與沒有處理的處理時間比較

[6]中設計了一個實驗來量測上述處理方式的效能，該實驗中執行兩次相同的 MapReduce Job，但是一次在 Map Phase 加入落後者處理；另一次則沒有。圖 27 是該實驗 Map Phase 的執行時間，Y 軸為不同時間點的處理速率，我們可以發現兩次實驗在接近完成的時候處理速率幾乎是躺平在座標軸上，也就是說只剩下非常少量的資料尚未完成。左圖是有在其他節點上重新執行的狀況，整體執行過程約花費 830 秒左右；右圖是未啟動重新執行的執行過程，全部約經過了 1200 秒，幾乎是原本的 1.5 倍。因此我們可以推論說這樣的重新執行的方式能夠有效管理、降低總執行所需時間。

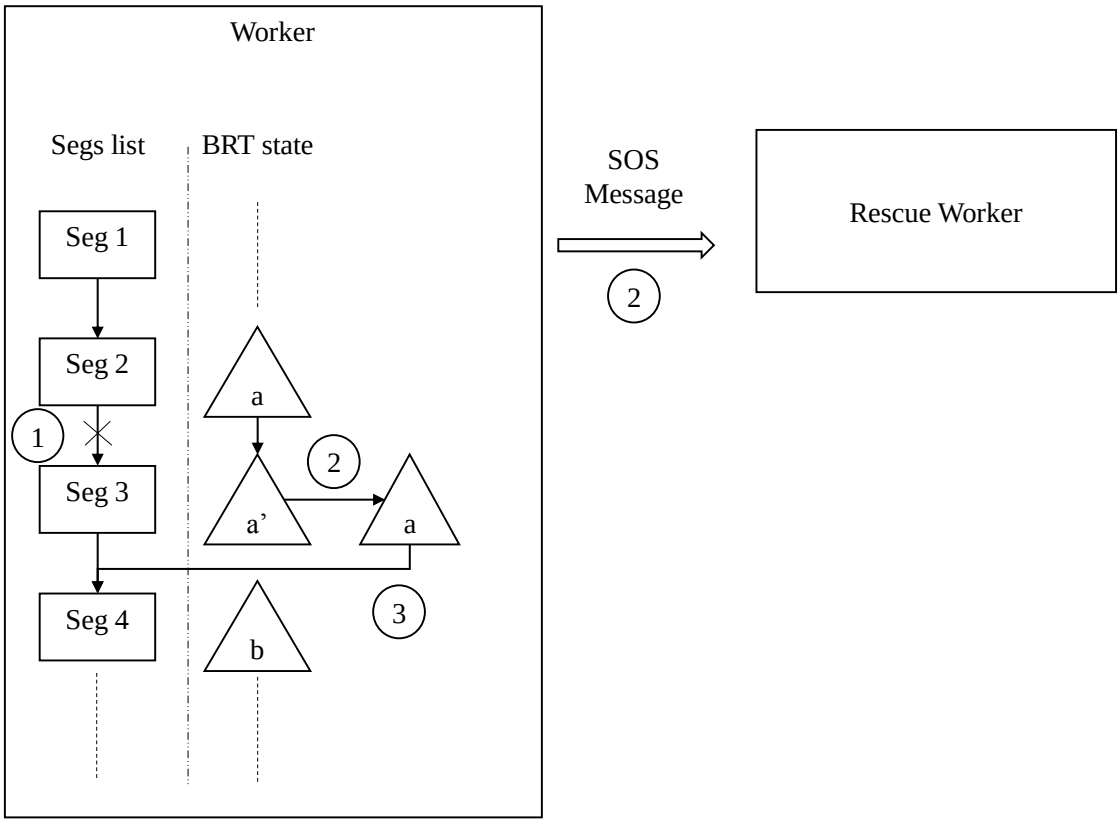


圖 28、NetActy 中 Straggler 處理流程圖

本論文參考[6]的方法設計了處理落後者機制，其運作流程有三個步驟，詳述如下：

1. NetActy 執行過程遇到資料處理停滯時，等於隱含其執行完成時間是無窮大。我們提出的處理方式是，根據先前實驗，我們統計在實驗的計算環境下每 1MB 資料的平均處理時間，在每筆資料檔案被處理時根據其大小計算期待(expected)執行完成時間，當該資料檔案已經執行時間超過期待完成時間時，該計算節點便會自動停止執行該資料檔案，而跳到下一個待處理檔案。

   在我們的實作中，當造成執行停滯的檔案如果透過 HDFS 讀取資料也是會有問題，但是如果直接從本地硬碟讀取處理的話，該檔案是能夠正常的執行。所以我們在 NetActy 系統加入一台新機器做為落後者處理的救援機器，目前是假設在該機器上儲存所有完成前處理的資料，此外該機器上並沒有額外計算節點存在，也就是沒有資源競爭的狀況發生。每當有計算節點遇到停滯的資料時就立刻傳 SOS 訊息給救援機器，由其負責建立問題檔案的 BRT 的工作。

   當資料交給救援機器處理後原先計算節點若要將 NetFlow Record 從原先 BRT 中移除是相當花費時間，所以我們在每個 Segment 執行完成時會為 BRT 做備份（snapshot），當遇到停滯資料時我們會先傳 SOS 訊息，接著利用備份把 BRT 回覆到加入該停滯資料前的狀態。如此一來就不需要額外紀錄該資料中已經處理過的 Netflow Record 數量，甚至是從 BRT 中移除了。

2. 當我們還原把 BRT 到先前狀態後，計算節點就可以繼續往下執行，處理尚未完成的資料。

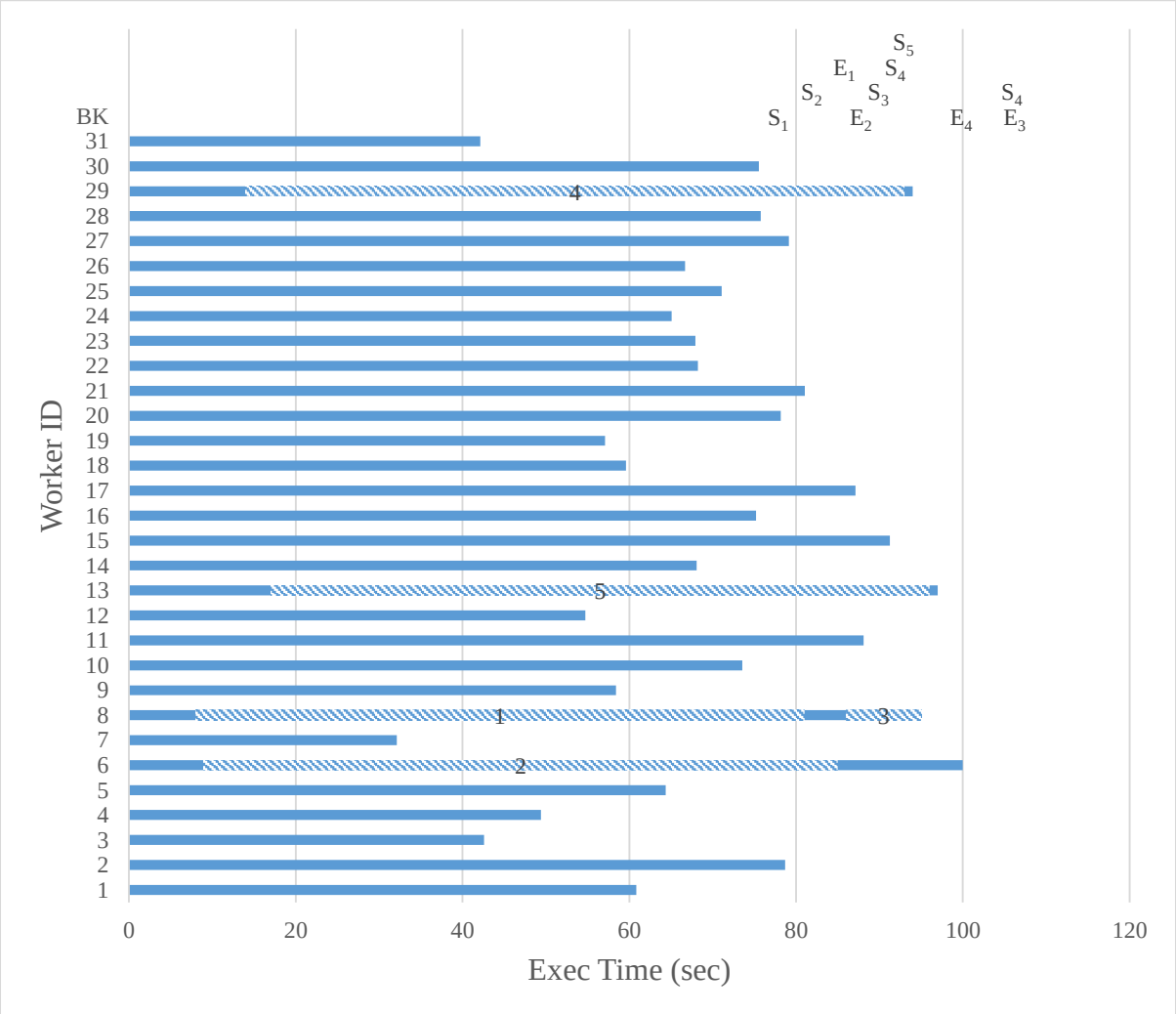
圖 29 顯示採用前述對落後者處理後的執行時間。圖中虛線為未完成資料之已經執行加上起動落後者處理機制所花費時間，而救援機器則是以 S_i 表達第 i 個未完成資料的重新執行時間，以 E_i 表達第 i 個未完成資料的執行完成時間。我們可以看到救援機器執行時間都比其他計算節點長，因為備援用機器只在計算節點查覺有資料執行超過最長執行時間之後才會開始執行。我們在比較圖 1 圖 29(a)與圖 29(b)，可以發現到兩個 Query 的 Map phase 最終完成時間約在 109 與 211 秒秒左右。雖然 Q4 較晚才察覺執行停滯並起動落後者機制，約在 80 秒左右才開始重新執行，但因救援機器擁有 12 核心 CPU 並採用多執行緒同時處理未完成資料，所以只額外花了 9 秒完成 Q4。Q5 與 Q6 隨著未完成資料數量增加，造成救援機器的工作量上升，不過總體而言只需額外 32 與 27 秒就能完成。

(s)	Q4			Q5			Q6		
	Avg	Max	Mean Diff.	Avg	Max	Mean Diff.	Avg	Max	Mean Diff.
	75	109	21	134	211	27	206	321	50

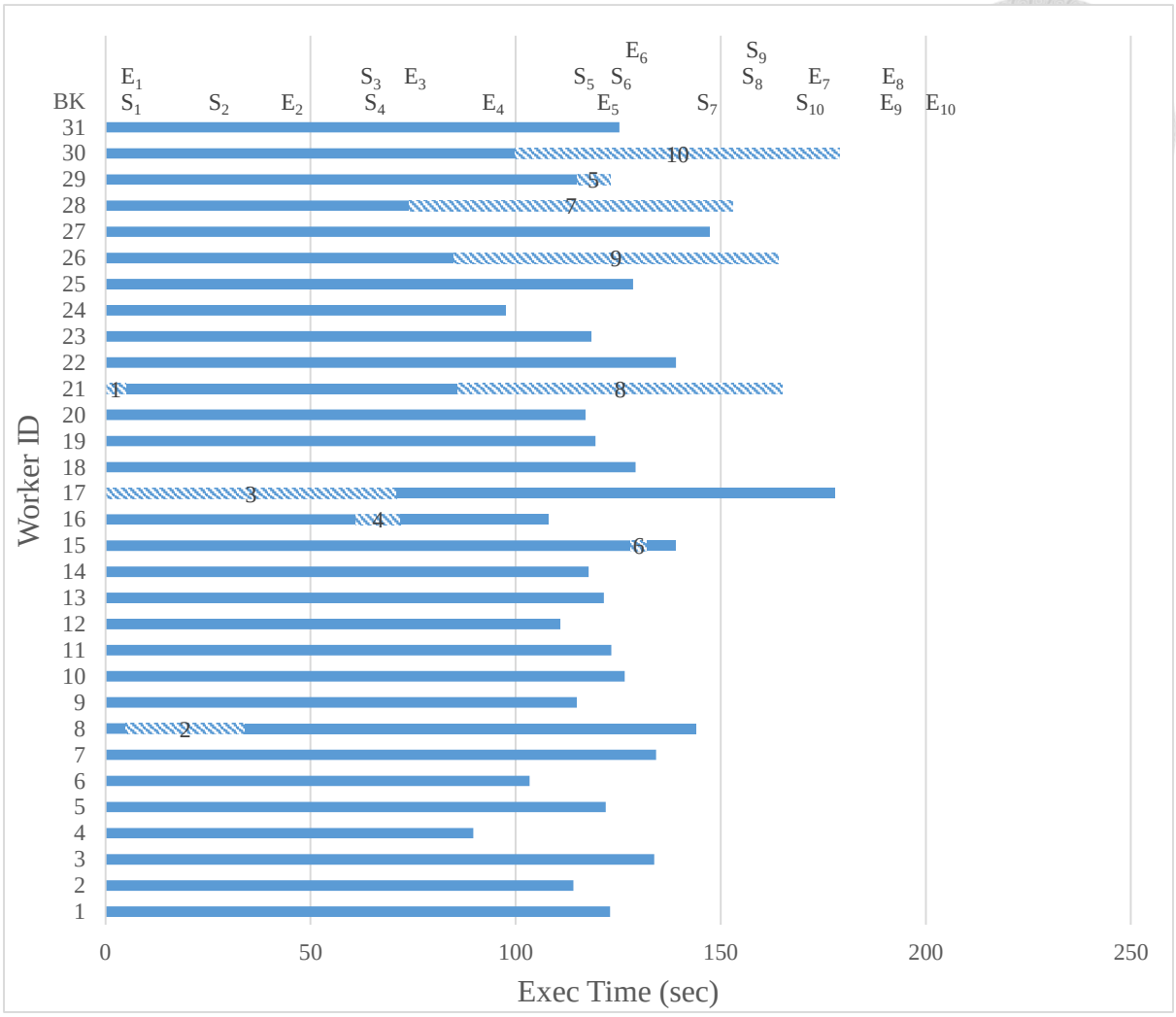
表 22、Q4 至 Q6 在處理落後者之平均、最大執行時間與 Mean Difference 比較

(s)	Max exec time wo backup worker.	Max exec time w/ backup worker.	Start exec time of backup worker
Q4	100	109	81
Q5	179	211	5
Q6	293	321	28

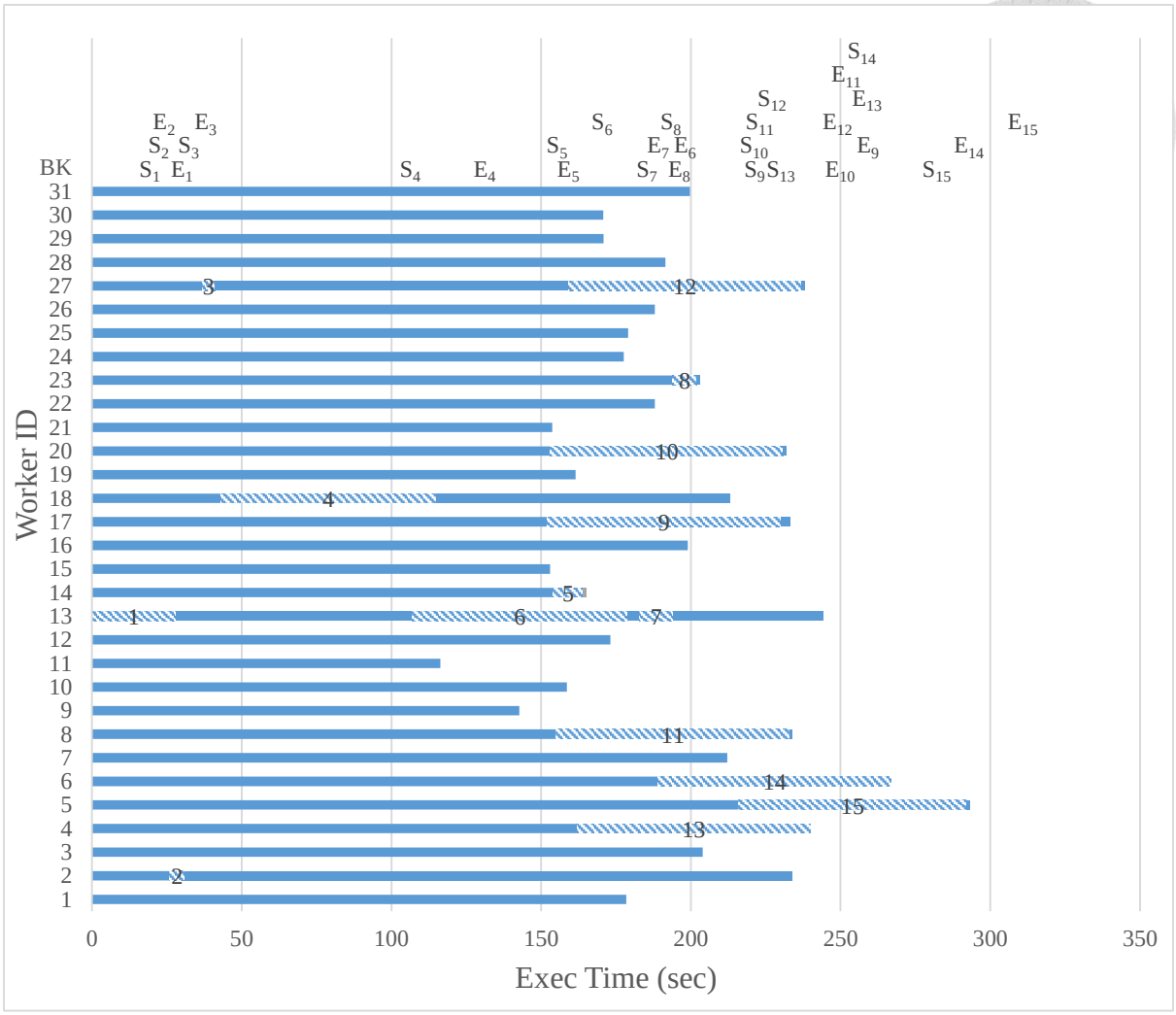
表 23、Q4 至 Q6 在有無包含救援機器之最大執行時間與救援機器之啟動與完成時間



(a) Q4


E1
S3
E3
E6
S9
S5
S6
S8
E7
E8
BK
S1
S2
E2
S4
E4
E5
S7
S10
E9
E10
Worker ID
Exec Time (sec)
0
50
100
150
200
250

(b) Q5



(c) Q6

圖 29、Q4 至 Q6 在處理落後者後其各計算節點與 Backup 節點之執行時間比較

第四章　搜尋結果視覺化之運算

在完成 Map Phase 之 (Job Assignment 及 BRT Building)的運算後，接下來在此章節我們針對 NetActy 系統的一個互動式查詢的第二階段執行，也就是將所搜尋到的資料以視覺化的方式呈現給使用者。這個階段是對應到所謂 Reduce Phase，運算內容包含分散於個計算節點下各通訊連結國家的 BRT partitions 的合併（Merge）以及 Visualization 運算。這個階段的運算時間仍須最小化，以達到互動模式下的執行回應時間之需求。

第一節　方法一

根據最基本的 MapReduce 運算架構，Map Phase 完成的結果，各 Map 計算節點將 Segments 建立成一或多個 BRT Partitions 後。一種做法是仿效[6]利用 Shuffle 程序將屬於同一國家的 BRT Partitions 利用 FTP 的方式傳送到某一個負責合併的計算節點合併成一棵 BRT 並在該節點上執行 Reduce task。這種作法有涉入一次的網路傳輸以及兩次硬碟檔案讀寫。

Reduce 計算節點負責將合併後各國的 BRT 轉換成可供視覺化呈現的 HTML Code，再回傳給 Visualization Server 將每個國家的 Code 串連成一份 HTML Code 顯示在網頁地圖上。

我們認為利用 FTP 交換 BRT Partition 以合併各國家所屬單一 BRT 需將 BRT Partition 寫回本地硬碟，再從本地硬碟讀出經網路傳送到目的硬碟以供 Reduce Task 處理，這步驟需耗費時間與網路頻寬，以及大量 Disk I/O 存與取，應該要盡量避免。在此思維下，FTP 步驟可利用 Message Passing 方式改善，直接把 Partitions 透過 Socket 傳輸；負責合併的計算節點在接收合併後直接儲存在 Memory，免除需要額外 effort 在從硬碟中讀進 Memory。

接下來我們考慮是否需要"將 Partitions 合併後轉換成 HTML Code 再將 HTML Code 串聯"？ 如果可以不要，將可節省網路傳輸的成本。也就是另一種做法是" 不合併 Partitions，先各自轉換成 HTML Code 後，交由 Visualization Server 將同一國家的 HTML Code 合併並串聯各國家的 HTML Code"會是相同的結果，我們先定義以下變數與符號以利說明：

A. $BRT_{i,j}$ 為第 i 個擁有流量國家的第 j 個 BRT Partition、

B. BRT_i 為第 i 個擁有流量國家合併後的 BRT、

C. M 為將 BRT 轉換為 HTML Code 的 Function、

D. $hc_{i,j}$ 為由 $BRT_{i,j}$ 經由 M 所產生的 HTML Code、

E. HC_i 為由 BRT_i 經由 M 所產生的 HTML Code，另外定義

F. $+$ 為 BRT 或 HTML Code 合併的運算符號、

G. $\cap$ 為將 HTML Code 串聯的運算符號、

H. $\times$ 為將 BRT 轉換為 HTML Code 的運算符號。

因此我們可以將改進前的過程以符號改寫如下：

$$(BRT_{1,1}+BRT_{1,2}+\ldots)\times M \cap (BRT_{2,1}+BRT_{2,2}+\ldots)\times M \cap \ldots = HC_1 \cap HC_2 \cap \ldots$$

先說明 HTML Code 的特定格式：

```
{
    center:{x,y},
    radius:219067.24394179948,
    …
    click:function(e) {
        $('#title').text('CN-Code');
        $('#nor').text('545,575');
        $('#nop').text('0');
        $('#fs').text('0');
        …
        $('#sublist').append($('<option></option>').attr('value', 'AS-Name').text('AS-Name'));
        …
    }
}
```

以上代表某國家的 HTML Code，每個國家都相同部分就省略僅列出會變動的內容。首先 center 意指該國家劃在地圖上的經緯度、radius 則是當我們以圓圈表示該國家流量多寡時所應該呈現的半徑，此數值會依 NetFlow Record 數量改變、title 是該國家的 ccTLD (Country code top-level domain)、nor (number of record)為該國家所擁有的 NetFlow Record 數量、nop (number of packet)為該國家所傳輸的封包數量、fs 為該國家傳輸資料量，最後的 sublist 為該國家內有流量的 AS。

而 HTML Code 在合併時僅僅將不同國家的 HTML Code 以逗號串聯起來，串聯後將 Code 回傳並不會更動到 Code 內容，所以接下來我們可以針對個別國家的 Code 做分析。我們想要知道對每個國家而言，

$(BRT_{i,1}+BRT_{i,2}+\ldots)\times M = (BRT_{i,1}\times M + BRT_{i,2}\times M + \ldots)$ 是否成立，也就是

$hc_{i,1} + hc_{i,2} + \ldots = HC_i$ 是否成立。

首先對任意 $hc_{i,j}$ 而言，都是由相同國家的 BRT Partitions 所產生的，因此 $hc_{i,j}$ 間的 center 與 title 就沒有差異。接著不同 Partition 間因為由不同的 Segment 所產生，而這些 Segment 在時間上不會重疊，所以我們可以將 $hc_{i,j}$ 之間的 nor、nop 與 fs 直接相加而不會造成重複計算。那因為 radius 根據 nor 做計算，所以當 nor 相加後就能重新計算正確半徑。sublist 紀錄該國家下所有具有流量的 AS，同樣的因為不同 Partition 是由不同的 Segment 所產生故可能造成不同 Partition 有重複的 AS 存在，所以當我們要合併 $hc_{i,j}$ 間的 sublist 就只要紀錄所有不重複的 AS 就可以。

根據上述方法當我們把所有 $hc_{i,j}$ 合併之後就能夠產生 HC_i ，所以 $hc_{i,1} + hc_{i,2} + \ldots = HC_i$ 成立。也因此我們就能不做 BRT Partitions 合併的情狀下直接轉換成 HTML Code 後再對 HTML 合併與串連。

綜合以上說明，我們可以省略 BRT Partition 的合併步驟，比起原先做法能有效減少時間與硬碟讀寫及網路資料傳輸的成本。

第二節　視覺化介面的資料瀏覽：BRT Traversal

當使用者在視覺化地圖介面提出查詢指令(Query)後，系統傳回資料，使用者接下來可以在此查詢指令做資料瀏覽，例如可針對特定網際網路區域的通訊流量作更加詳細的閱覽或是回到前一次瀏覽畫面，而這些國家、AS、CIDR、IP 的上下瀏覽 ，系統之 Visualization Server 需要與參與此查詢指令的計算節點間進行溝通，以找出擁有該區域的 BRT Partition，請節點將資料回傳或是再次顯示先前瀏覽過的畫面。

為此，我們設計兩個加速使用者上下流覽的技術，透過降低回應時間讓 NetActy 達到較高的可互動性。在下面章節我們會介紹 Visualization Server 與計算節點之間的通訊模式以及加速資料索取的技巧。

2.1 通訊模式

當使用者透過視覺化地圖介面選取特定區域的流覽需求後，Visualization Server 採

用輪詢 (Polling)的方式向各個計算節點詢問是否擁有該區域的 BRT Partition，如果有，就將對應的資料回覆，否則就回傳空資訊表示該計算節點並沒有該區域的 BRT Partition。當 Visualization Server 完成一輪的詢問收到所有節點回覆，將結果給網站介面呈現，完成此一流覽。

我們定義 Visualization Server 向各計算節點發送的資料需求訊息的內容格式如下：

(Query Id_CN _AS _CIDR _IP_Display Level)

A. Query Id：當使用者下達有著時間、空間範圍與流量方向的 Query 指令時，系統會給予一個獨立唯一的 Id 作為識別，並建立 BRT Partition。在一 Query 下的所有流覽都使用相同 Id 允許在此 Query 的 BRT Partition 內作閱覽，若 Id 改變則代表新的 Query 指令產生。
B. CN：欲查詢區域的國家名稱，當填入 ANY 則代表全部國家而沒有特定對象。
C. AS：欲查詢區域的 AS 名稱，當填入 ANY 則代表全部 AS 而沒有特定對象。
D. CIDR：欲查詢區域的 CIDR 名稱，當填入 ANY 則代表全部 CIDR 而沒有特定對象。
E. IP：欲查詢區域的 IP 名稱，當填入 ANY 則代表全部 IP 而沒有特定對象。
F. Display Level：表示使用者想要呈現的網路層級，從 0 代表顯示國家、1 為顯示特定或全部 AS、2 為特定或全部 CIDR 而 3 為特定或全部 IP。

所有可能的資料要求訊息我們以圖 30 表示。

Show CN	Query Id_ANY_ANY_ANY_ANY_0	
Show AS	Query Id_CN_ANY_ANY_ANY_1	Query Id_CN_AS_ANY_ANY_1
Show CIDR	Query Id_CN_AS_ANY_ANY_2	Query Id_CN_AS_CIDR_ANY_2
Show IP	Query Id_CN_AS_CIDR_ANY_3	Query Id_CN_AS_CIDR_IP_3

圖 30、NetActy 在不同 Display Level 下之資料要求訊息內容

當計算節點收到資料需求訊息時，首先依據 Display Level 決定資料呈現層級。當 Display Level 為 0 時就顯示所有國家的流量；當 Display Level 為 1 時則顯示 AS 層級的流量，有兩種選擇：當 AS 為 ANY 時則代表呈現國家 CN 下所有的 AS 否則只呈現一個 AS 的流量。Level 2、Level 3 與 Level 1 也是相同的規則。

當使用者想要對目前所瀏覽的區域，往下一層瀏覽時，根據選定特定區域與否有兩種可能的資料要求訊息傳送給所有節點，資料要求訊息內容的變化過程如圖 31 所示。當使用者想要往上一層瀏覽時，為避免使用者還要選擇呈現全部或特定區域所以我們按照圖 32 中順序找出對應資料要求訊息查詢。

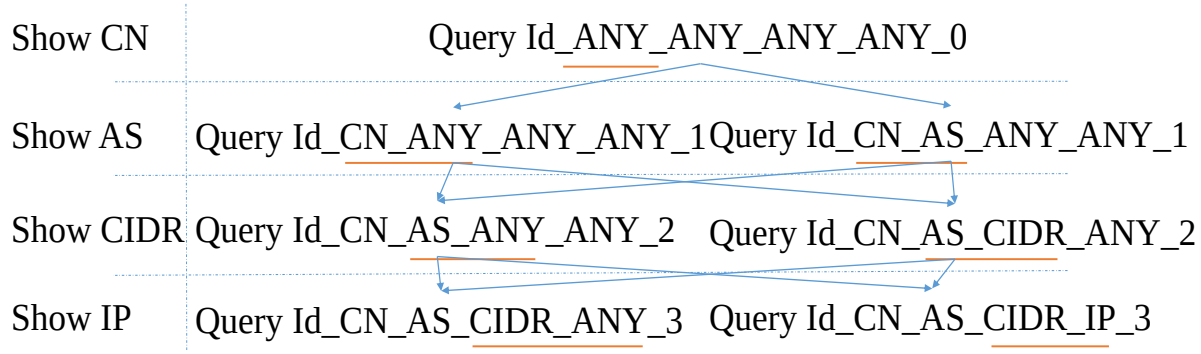


圖 31、當使用者往下一層瀏覽 BRT 時，資料要求訊息內容變化順序

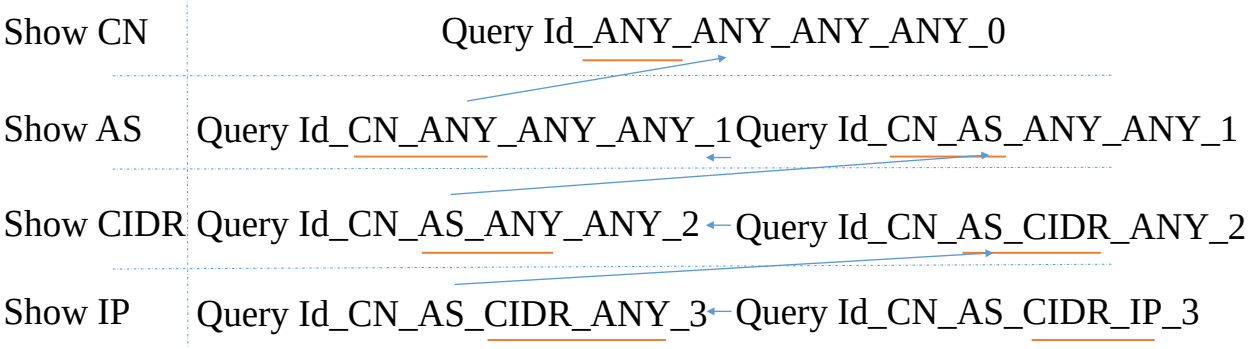


圖 32、當使用者往上一層瀏覽 BRT 時，資料要求訊息內容變化順序

2.2 加速瀏覽回應時間

在 Reduce Phase 執行過程中，我們提出兩種處理方式以利使用者反覆瀏覽時的回應時間。第一，為 HTML Code 的快取機制。在使用者在此 Query 下不同層級的資料瀏覽過程中，Visualization Server 必須依據所欲瀏覽的目標提出不同的資料要求訊息產生對應的 HTML Code。為便利在來回反覆瀏覽中，回到之前曾經瀏覽過的目標內容的呈現，所以我們把資料要求訊息的回覆內容和其對應的 HTML Code 儲存在 Visualization Server 的 Master Memory 中。當 Visualization Server 收到新的資料要求訊息時，我們先查詢新訊息與其對應的 HTML Code 是否已經儲存在 Memory 中，若有則直接回傳，否則就回到先前的步驟，將資料要求訊息送給各節點並等待回復。

進一步，我們利用 Multicast 通訊以利縮短使用 Polling 方式詢問每一個計算節點所需花費的時間。

2.3 記憶體消耗量

在 Reduce Phase 的運算裡， BRT Partitions 是分散地儲存在各計算節點的記憶體，也就是說一個國家可能同時有多個 Partitions 儲存在不同的 Reduce 計算節點的記憶體。我們想要瞭解隨著 Query Time Domain 增加，BRT 大小的增加幅度。

在假設每天受管理網域的通聯對象沒有太大變化的情況下，BRT 內儲存的節點與連線也是差不多而會改變的只有連線上的數量多寡，像是 NetFlow Record 數量、封包數量以及傳輸資料量所以 BRT 大小的總合應該是差不多的。但是從圖 33 中看到，當 Query 天數增加時對每一個國家將屬於該國的 BRT partitions 合併後的，所有 BRT 大小呈現線性成長，這說明不同天的通聯對象不盡相同。此外，不合併所需的記憶體容量比合併所需的記憶體容量要大，這是因為通聯雙方資料分散於不同計算節點。這也是節省網路頻寬傳輸所付出的成本。

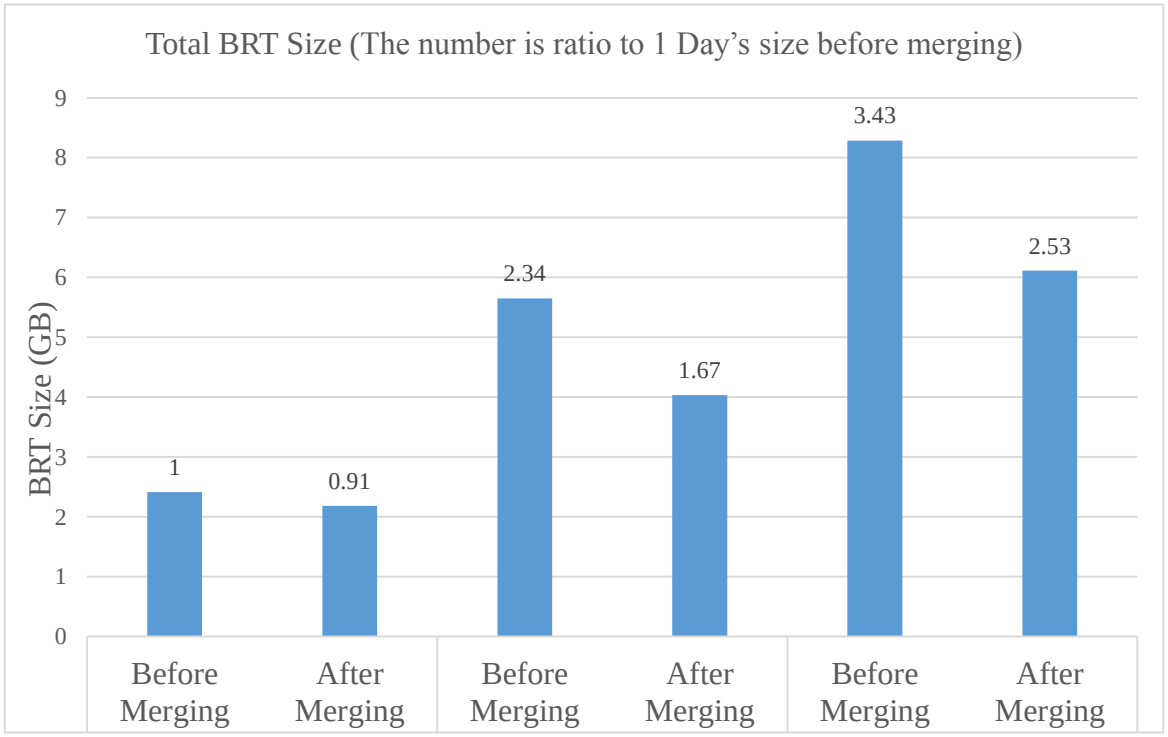


圖 33、NetActy 在不同天數之 Query 其 BRT 合併前後的大小總合

在圖 34 中看到任兩天重複的通聯對象數量非常的低，連 10%都不到。當我們想找出三天內重複的通聯對象時，甚至只有 2.2%是重複的。也正是因為如此 BRT 大小才會隨著天數而成長。而且因為 BRT 大小會隨著 Query 天數線性成長實作上計算環境

的分配給各計算節點的記憶體空間就必須要足夠儲存運算完成的 BRT Partitions。

在現實狀況中記憶體空間不能夠無限制增加，一種可能的做法是當實體機器記憶體遇到空間不足的狀況，將目前記憶體內容儲存進硬碟中做 Swaping。例如每當 NetActy 遇到超過三天的 Query 時就分次進行處理，每次執行三天的 Query，就把目前 BRT Partitions 的視覺化資訊回傳給 Visualization Server 後將各國家的 BRT Partitions 寫入硬碟中，確定所有 Partitions 都寫到硬碟之後再將 BRT 從記憶體中移除並繼續進行下一個三天 Query 的處理。如此重複直到所有視覺化資訊都回傳就執行完成。當使用者需要往下 Traverse 時就需要從硬碟中讀取對應的 BRT Partition 再將視覺化資訊回傳，經由硬碟空間保存原先儲存在記憶體當中的 BRT，再根據使用者的需求切換 BRT。

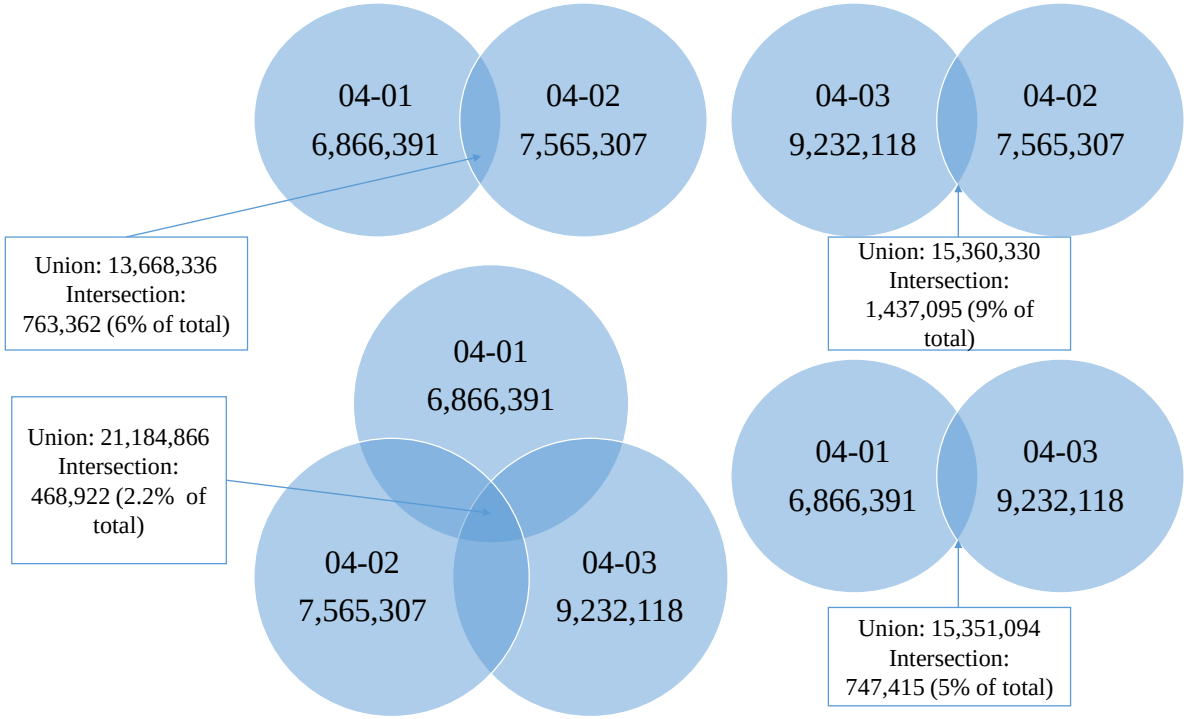


圖 34、NetActy 對不同天的通聯對象取交集情形

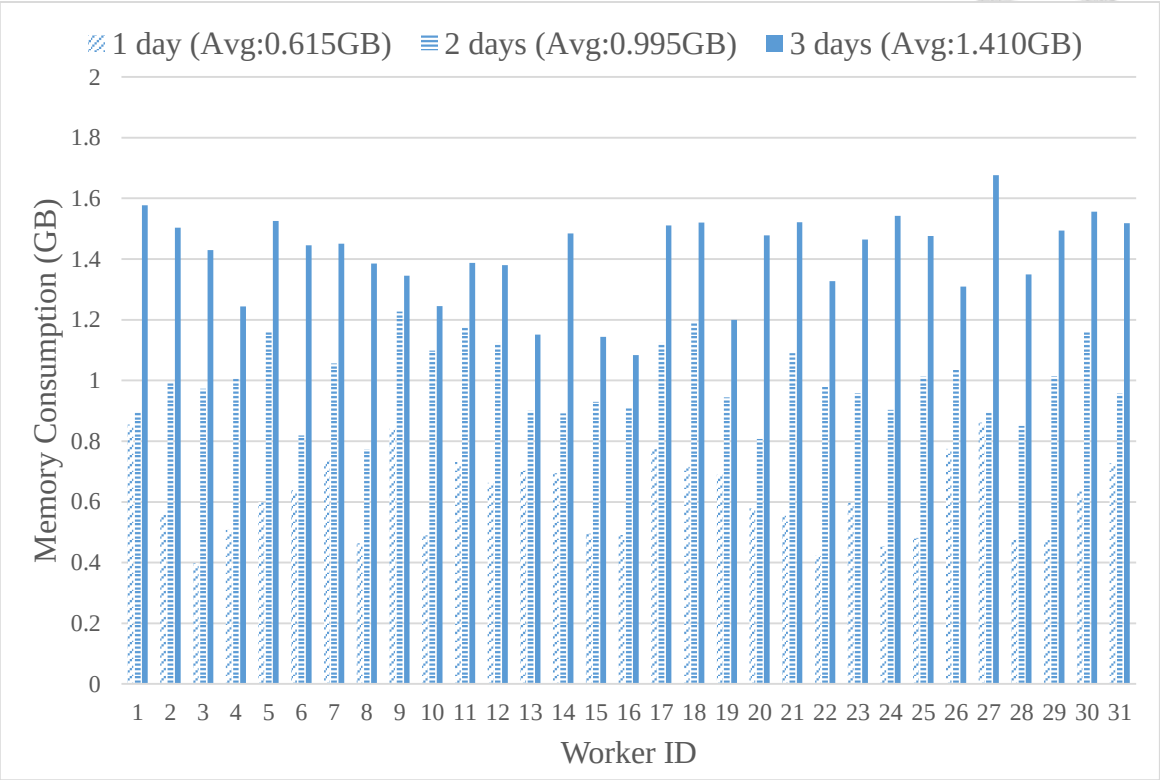


圖 35、NetActy 在不同天數下各計算節點之記憶體使用量

(MB)	Avg	Std	Max	Min
Q4	615.40	133.45	861.13	397.67
Q5	995.71	119.27	1227.56	773.45
Q6	1410.47	139.69	1676.60	1083.60

表 24、NetActy 在不同天數下計算節點記憶體使用量的平均、標準差與最大最小值

第五章　結論與建議

現今網路流量已以往無法想像的速度成長；網路犯罪亦隱身在龐大的網路流量中。攻擊者不如以往高調的行為，轉為偷偷摸摸值入後門藉此竊取有價值的資料，正因為低調的攻擊行為造成資安調查人員只能在事件發生之後開始做調查，但是事後的資安犯罪偵查會面臨到兩大困境。

首先傳統資料儲存方式無法有效存放每天所產生的網路流量；再來面對如此龐大網路流量時，資安調查人員很難有效率的從中過濾出有用的資訊。為協助資安人員快速且有效率地為在網路流量中找出可做為呈堂供證的通聯記錄，在先前研究中提出了將網路流量視覺化的互動式查詢系統－NetActy。

NetActy 採用 HDFS 作為儲存系統，提供分散式儲存、備份與負載平衡等功能，以儲存大量網路流量資料。此外為加速資料處理，我們針對網路流量資料作前處理，其主要目的有三：一是利用 Columnar Storage 減少每次系統所需處理資料量；二是利用 Running Length Encoding 為資料作壓縮；三則是設計 Hierarchical Path 讓 NetActy 得以快速找到待處理資料的位置。為表達流量資料間上下隸屬關係，我們設計了樹狀資料結構－BigIP Rendering Tree 來紀錄不同階層中，各地理區域間的通連狀況。NetActy 透過分散式處理的方式，讓許多計算節點共同平行化處理以加速 Query 完成時間，此外，NetActy 亦提供網頁介面，使用者透過介面下達包含時間與空間範圍的 Query，藉由將結果視覺化呈現在地圖上，資安人員能夠一目瞭然的觀察到流量之間的差異，以找出可疑的網路流量。

本論文為改進 NetActy 的互動性，我們考慮節點間工作量的平衡以及 Data Locailty，目的希望讓各計算節點的執行時間趨近平衡以達到互動程度的回應時間，並將各計算節點的計算能力納入考量以避免計算節點過長執行時間而影響整體系統的互動性。本論文將工作量分配問題制定成一個 Linear Programming 問題，並提出經驗解－Algorithm 1 在多項式時間內解決。

在計算節點執行過程中，會遇到執行停滯的狀況發生，對於這些計算節點我們稱為落後者 (Sraggler)。參考文獻作法，我們為每筆資料設定一個期望執行完成時間，當執行超過期望時間時就啟動落後者處理機制：落後者將立即停止處理並交由救援機器處理，接著跳過該筆資料繼續往下執行。藉由以上處理方式，就能避免執行停滯的狀況發生。在網路流量視覺化部分，本論文避免交換分散處理所產生的中間結果以減少執行時間，並為每個查詢視圖做快取以及利用 Multicast 技術來加速處理。當 Query 隨著天數增加時，BRT 在記憶體中的大小亦隨之增加，當計算節點無法完整儲存時我們參考現行作業系統的解決方法，透過 Swapping 來解決。首先將 Query 切割成 sub-query，

而每次 sub-query 執行完成後就將 BRT 寫入硬碟中，每當需要查詢該 BRT 內資料時再讀進記憶體中。如此一來就解決記憶體空間的問題。

總結而言，本論文提出一套網路流量互動式查詢與視覺化系統－NetActy，利用分散式處理以提供互動式回應時間；將資料儲存再 HDFS 中，以獲得備份與負載平衡的好處；設計了樹狀資料結構 BigIP Rendering Tree 方便紀錄各地理區域間的通聯資訊。此外，我們亦將工作量平衡、資料在地化與各計算節點的處理能力納入工作量分配的考量，提出 Algorithm1 來解決，同時 NetActy 也設計了處理落後者的機器以避免執行過久的計算節點影響整體執行時間。最後我們避免計算節點的中間結果交換以減少執行時間；面對記憶體空間的問題，我們參考現行作業系統的方法用 Swapping 解決。藉由以上方法，NetActy 提供查詢龐大網路流量的能力並在互動時間內視覺化呈現在地圖上，相信透過 NetActy 資安人員能夠以快速、輕鬆、直覺的方式來完成事後犯罪偵查。

參考文獻

[1] Cisco, "Cisco Visual Networking Index: Forecast and Methodology, 2013–2018," 2014.

[2] Y. S. S. Chang-Ming Wu, "Visually Interactive Security Analysis of BigIP," 2014.

[3] Y. S. S. Wei-Ru Dai, "Interactive Visualized Security Analysis System of Large Distributed Network Flow Data.," 2014

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, 2010, pp. 1-10.

[5] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column-oriented database systems," *Proceedings of the VLDB Endowment*, vol. 2, pp. 1664-1665, 2009.

[6] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, pp. 107-113, 2008.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *ACM SIGOPS operating systems review*, 2003, pp. 29-43.

[8] A. Bialecki, M. Cafarella, D. Cutting, and O. O'MALLEY, "Hadoop: a framework for running applications on large clusters built of commodity hardware," *Wiki at http://lucene. apache. org/hadoop*, vol. 11, 2005.

[9] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, *et al.*, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135-146.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 10-10.

[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012, pp. 2-2.

[12] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, *et al.*, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, pp. 330-339, 2010.

[13] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, *et al.*, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, pp. 1626-1629, 2009.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD*

international conference on Management of data, 2008, pp. 1099-1110.

[15] M. Kornacker and J. Erickson, "Cloudera Impala: Real Time Queries in Apache Hadoop, For Real," *ht tp://blog. cloudera. com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real*, 2012.

[16] A. Floratou, U. F. Minhas, and F. Ozcan, "Sql-on-hadoop: Full circle back to shared-nothing database architectures," *Proceedings of the VLDB Endowment*, vol. 7, 2014.

[17] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 69-84.

[18] C. Gini, "Measurement of inequality of incomes," *The Economic Journal*, pp. 124-126, 1921.

[19] S. Yitzhaki, "Gini's mean difference: A superior measure of variability for non-normal distributions," *Metron*, vol. 61, pp. 285-316, 2003.

[20] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, 1971, pp. 151-158.

[21] R. M. Karp, *Reducibility among combinatorial problems*: Springer, 1972.

[22] M. E. McDowell, "Multiprocessor Scheduling in the Presence of Communication Delay," *Master of Science Thesis. MIT, Dept. of Elec. Engineering and Comp. Science, Boston*, 1989.

[23] M. F. Tompkins, "Optimization techniques for task allocation and scheduling in distributed multi-agent operations," Massachusetts Institute of Technology, 2003.

[24] M. Luo and H. Yokota, "Comparing Hadoop and Fat-Btree based access method for small file I/O applications," in *Web-Age Information Management*, ed: Springer, 2010, pp. 182-193.

[25] H. Yokota, Y. Kanemasa, and J. Miyazaki, "Fat-Btree: An update-conscious parallel directory structure," in *Data Engineering, 1999. Proceedings., 15th International Conference on*, 1999, pp. 448-457.

[26] B. Dong, Q. Zheng, F. Tian, K.-M. Chao, R. Ma, and R. Anane, "An optimized approach for storing and accessing small files on cloud storage," *Journal of Network and Computer Applications*, vol. 35, pp. 1847-1862, 2012.