國立政治大學資訊管理學系

碩士學位論文

指導教授:劉文卿博士

傳統關聯式資料庫暨欄導向資料庫之轉換機制研究–以台灣學術期刊搜尋引擎為例

An Approach to the Translation Mechanism from Relational-based Database to Column-oriented Database - Take Taiwan Academic Journal Search Engine as an Example

研究生：黃勁超

中華民國一百零一年七月

致謝

輾轉兩年之間，從懵懵懂懂的大學生搖身一變成為具備一技之長以及些許實習經驗的碩士生，心中真是百感交集。首先感謝我的媽媽，謝謝妳讓我毫無後顧之憂地完成碩士學業，之後我一定會致力於事業上，另創人生的巔峰。再來感謝總是耐心傾聽我冗長的報告的劉文卿老師，老師不吝指點我研究的方向、在一次又一次的報告中匡正我的架構並給予報告方法上的建議，嚴格卻和藹地在這兩年半中循循善誘，讓我得以有在企業實習的機會。在畢業前提前體會真實社會的環境，使我獲益匪淺，並完成這本曠世鉅作，在此獻上至深的謝意。

感謝老是與我鬥嘴，忍受我白目個性的雅菱以及佳穎學姊，使得我在什麼都不懂的情況下，慢慢地帶領我步入正軌。陪伴我度過兩年時光的大學部應屆畢業生學弟妹，總是很帥氣的維文、實驗室之花祖韵、酷酷的威辰、認真的承翰、愛家的冠緯、總是一直支持我的藝方以及肩膀很寬的冠廷，謝謝你們的鼓勵，我達成了與你們一起畢業的約定。此外，大三的學弟妹們當然也不能忘，最可愛的鈺雯、最愛一起屁話的政瑜、謹維、柏崴、文治、庭芳、貴堯、志騰，謝謝你們一年多來的陪伴與打拼。感謝戴睿這一年來的陪伴與支持，在未來的一年裡你會是kmlab最強大的支柱。感謝凭哲與柏翰，論文最後三個月有你們一起陪伴完成論文，真好。特別感謝鈺雯學妹，謝謝你在遇到挫折的時候給予扶持，用心的督促我論文的進度，以及最後關頭的無私校稿，使得最後論文得以即時的完成，真的非常感謝你。感謝實驗室的大家，像家人一樣一起努力不懈地奮鬥，一起歡笑，一起出遊，我想，這是我一輩子也忘不了的，最美好的回憶，我會常常回來看大家的。

感謝碩士班的同學們支持，太多同學們我無法一一列舉，但真心感謝你們各方面的相助，大家一起互相討論學業、玩樂、慶生，在研究所能認識你們是我最大的榮幸。感謝同學阿葉，為了出國的夢想而打拼，達成了不可能的任務，不管在學業、感情上都蒙受妳很大的幫助，謝謝你，往後也要繼續互相扶持。感謝系

辦助教雨儒以及詩晴，謝謝你們熱心的幫我解決行政上面的問題。

感謝從小一起長大的兩位麻吉，育徵與宏哲，感謝你們在我心情低落時總是給予我支持與鼓勵，在遇到困難時會陪著我打球洩氣，雖然我們各自擁有不同的生活可是還是像親兄弟一樣的麻吉，謝謝你們。

最後感謝兩位口試委員，陳亦光以及楊建民老師，感謝兩位老師撥冗參加我的論文口試，更給予我許多研究方面的建議，使得論文更臻完善。特別是亦光老師在公司的照顧，讓我有機會體驗大公司的體制並與來自各地的好手比拚，對於我來說是相當難得的經驗。

謹以此文，獻給無數個疲倦卻歡愉的夜晚，獻給愛我的師長、父母、同學及學弟妹，獻給被我私自重視的你們。期許在未來人生道路上，不會辜負這份初衷以及這些日子中的所知所學，當然，還有最美的回憶。

黃勁超

中華民國一百年八月一日

摘要

源於資訊量爆炸時代的來臨，企業面臨大量資料所帶來的挑戰：傳統關聯式資料庫無法負荷龐大資料所造成的效能及儲存設備升級等問題。為了解決大量資料所帶來的諸多問題，各界提出不同的理論，而其中最被廣為討論的就是雲端運算。時至今日，許多企業及個體用戶逐漸開始使用雲端運算中，目前最具代表性的分散式架構Hadoop上的資料庫代表－欄導向資料庫HBase來作為底層資料庫。故本研究提出一套傳統關聯式資料庫轉換至欄導向資料庫 HBase 之轉換機制，以台灣學術期刊搜尋引擎為例。

關鍵字：HBase，Hadoop，轉換機制，關聯式資料庫，欄導向資料庫，實體關聯模型，搜尋引擎

目錄

致謝 2

摘要 4

目錄 5

圖目錄 7

表目錄 9

第一章　緒論 10

第一節　研究背景與動機 10

第二節　研究架構及流程 12

第三節　研究目的與貢獻 14

第二章　文獻探討 15

第一節　雲端運算 15

第二節　Google File System 18

第三節　Apache Hadoop 20

第四節　Hadoop Distributed File System(HDFS) 22

第五節　Map/Reduce 23

第六節　HBase 25

第七節　Avro 27

第三章　關聯式暨欄導向轉換機制 30

第一節　欄導向資料庫結構轉換 30

第一段　實體關係轉換 30

第二段　一對一關係轉換 34

第三段　一對多關係轉換 35

第四段　多對多關係轉換 37

第五段　遞迴關係轉換 38
第二節　Avro 序列化資料結構轉換 40
第一段　實體關係轉換 41
第二段　一對一關係轉換 42
第三段　一對多關係轉換 43
第四段　多對多關係轉換 45
第五段　遞迴關係轉換 46
第三節　小結 47
第四章　系統實作 49
第一節　使用技術暨環境 49
第二節　學術期刊資訊搜尋介面 52
第一段　欄導向資料結構轉換 53
第二段　Avro 序列化結構轉換 56
第三段　搜尋流程 61
第三節　Web Service 63
第一段　輸入格式 63
第二段　輸出格式 63
第五章　結論與建議 65
參考文獻 67

圖目錄

圖 1. 查詢 Cloud Computing 一詞之次數折線圖(引用自 Google Trend, 2012)11

圖 2. 研究流程圖 12

圖 3. NIST 美國國家標準局對雲端運算的定義 15

圖 4. Google File System 架構(引用自 Ghemawat, Gobioff et al.，2003)..... 19

圖 5. Hadoop Ecosystem integration by MapR(資料來源：

http://www.mapr.com/products) 21

圖 6. HDFS 架構圖(資料來源：apache.org)..... 22

圖 7. Map/Recduce 運作圖(資料來源：White, T.，2010) 23

圖 8. HBase 架構圖(資料來源：George，2011) 25

圖 9. HBase Table 概念視圖(資料來源：George，2011)..... 26

圖 10. 實體關係轉換示意圖(Entity Relationship Translation)..... 32

圖 11. 一對一關係轉換示意圖 34

圖 12. 一對多關係轉換示意圖 35

圖 13. 多對多關係轉換示意圖 37

圖 14. 遞迴關係(R-R)轉換示意圖 38

圖 15. 欄導向 HTable 資料結構轉換至 HTable 使用 Avro 序列化儲存示意圖40

圖 16. Avro 實體關係轉換示意圖 41

圖 17. Avro 一對一關係轉換示意圖 42

圖 18. Avro 一對多關係轉換示意圖-HA 43

圖 19. Avro 一對多關係轉換示意圖-HB..... 44

圖 20. Avro 多對多關係轉換示意圖 45

圖 21. Avro 遞迴關係轉換示意圖 46

圖 22. Hadoop 叢集環境..... 50

圖 23. TAJ 系統架構圖 51
圖 24. 台灣學術期刊搜尋系統 ERM 圖 52
圖 25. 欄導向資料結構轉換-Paper 54
圖 26. 欄導向資料結構轉換-Periodical-Keyword-Author 55
圖 27. Avro 序列化結構轉換-Paper-1 56
圖 28. Avro 序列化結構轉換-Paper-2 57
圖 29. Avro 序列化結構轉換-Paper-3 57
圖 30. Avro 序列化結構轉換-Paper-4 58
圖 31. Avro 序列化結構轉換-Paper-5 58
圖 32. Avro 序列化結構轉換-Publisher 59
圖 33. N-gram 搜尋目標與搜尋標的 ERM 60
圖 34. Index Table Schema 60
圖 35. 搜尋流程示意圖 61
圖 36. 台灣學術期刊搜尋引擎搜尋畫面 62
圖 37. 搜尋論文全文內容結果 62
圖 38. Web Service JSON 輸入格式 63
圖 39. Web Service 輸出格式 64

表目錄

表 1. 台灣學術期刊搜尋引擎(TAJ)技術表 49

表 2. 搜尋標的資料筆數表 52

表 3. 關聯式資料庫 Table 欄位表 53

第一章　緒論

第一節 研究背景與動機

近年來隨著網路普及率飆高，連帶影響資訊量快速增長。全球最大能源公司雪弗龍(Chevron's)的首席資訊長曾在一篇公開文章提及，該公司一天的資訊量成長約在 17,592,000,000,000bits，相當於是 2TB 的巨大資料量。而在 IDC 的研究報告(Gantz & Reinsel，2010)中更預測 2020 年的資訊成長幅度將高達 2009 年的 44 倍。為了應付巨大資料量所造成的儲存空間問題，企業開始使用 DAS、NAS 與 SAN 等技術動態擴增硬碟及周邊設備的儲存空間。然而，雖然資訊儲存空間隨著時代的演進增加，但資料存取的速度卻始終無法跟上；因此，從龐大資料中存取單一檔案的時間增加，造成系統效能問題。因此，要如何在龐大的資料中快速、精確地搜尋到正確的資料便成為一門相當重要的課題。而之於搜尋引擎，這個課題更是不可或缺。

正當各界正為大量資料所帶來的問題苦惱時，Google 於 2003 年提出了以 Google File System(GFS)為核心的分散式儲存架構(Ghemawat, Gobioff et al. 2003)，奠定了雲端運算的基礎。隨後以 GFS 架構為核心的 Apache 開放原始碼專案 Hadoop 也應聲而出，成為目前最多企業及個體用戶所使用的分散式運算架構。分散式系統使用平行的 Input/Output，使大量資料能夠快速讀取，同時也令儲存空間能以 Scale-out(Greenberg, Hamilton et al.，2008)的方式增加，解決過去增加磁碟空間時，周邊設備也必須重新更換的不便與成本增加。

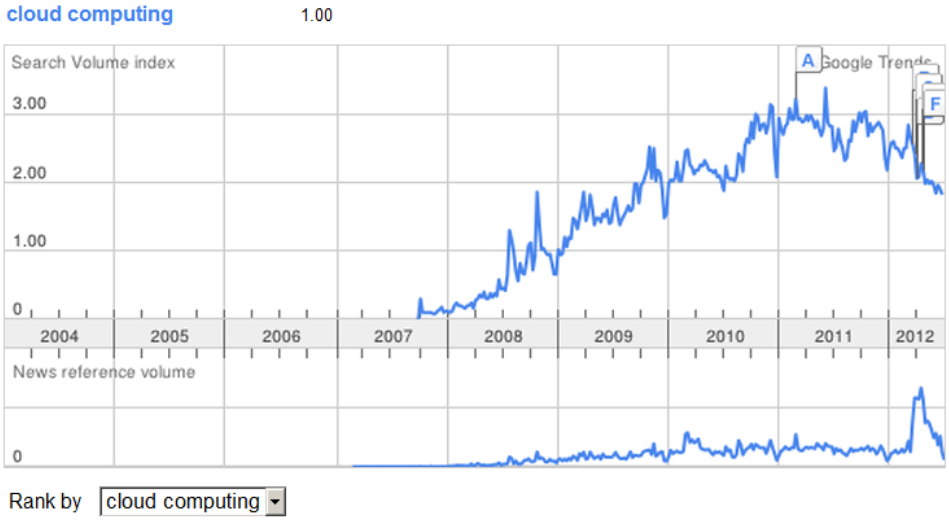


圖 1. 查詢 Cloud Computing 一詞之次數折線圖(引用自 Google Trend, 2012)

圖 1 為 Google Trend 截至目前為止的報告。由圖 1 可得知雲端運算的風潮由 2007 年末開始，直至 2010 年底開始步入穩定期，查詢量也逐漸下降。然 Google 於 2012 年 6 月 28 日發表聲明採用 MapR 公司所營運之 Hadoop 套件，將其整合進 Google Compute Engine 中(McLuckie，2012)，此舉證明 Hadoop 已進入穩定且商業化的時代。

隨著越來越多使用者開始將其原先系統移轉至Hadoop及其周邊開放原始碼專案，資料庫的移轉隨之成為一個重要的課題。過去，使用者最為廣泛使用的資料庫為傳統關聯式資料庫，因此本研究將探討如何將關聯式資料庫移轉至建置於 Hadoop 上的 NoSQL 欄導向資料庫代表－HBase，期望能找出一通用轉換機制。此外，為了印證本研究所提出的轉換機制為有效可行，本研究將建置一使用 HBase 為底層資料庫的台灣學術期刊搜尋系統，此系統實作一基於 Hadoop 的 Map/Reduce 分散式運算的轉換工具，此轉換工具方便使用者進行關聯式資料庫至欄導向資料庫的轉換，以驗證本研究所提轉換機制之可行性及效能改善。同時，也將就使用 Map/Reduce 轉換工具之執行效能進行探討。

第二節 研究架構及流程

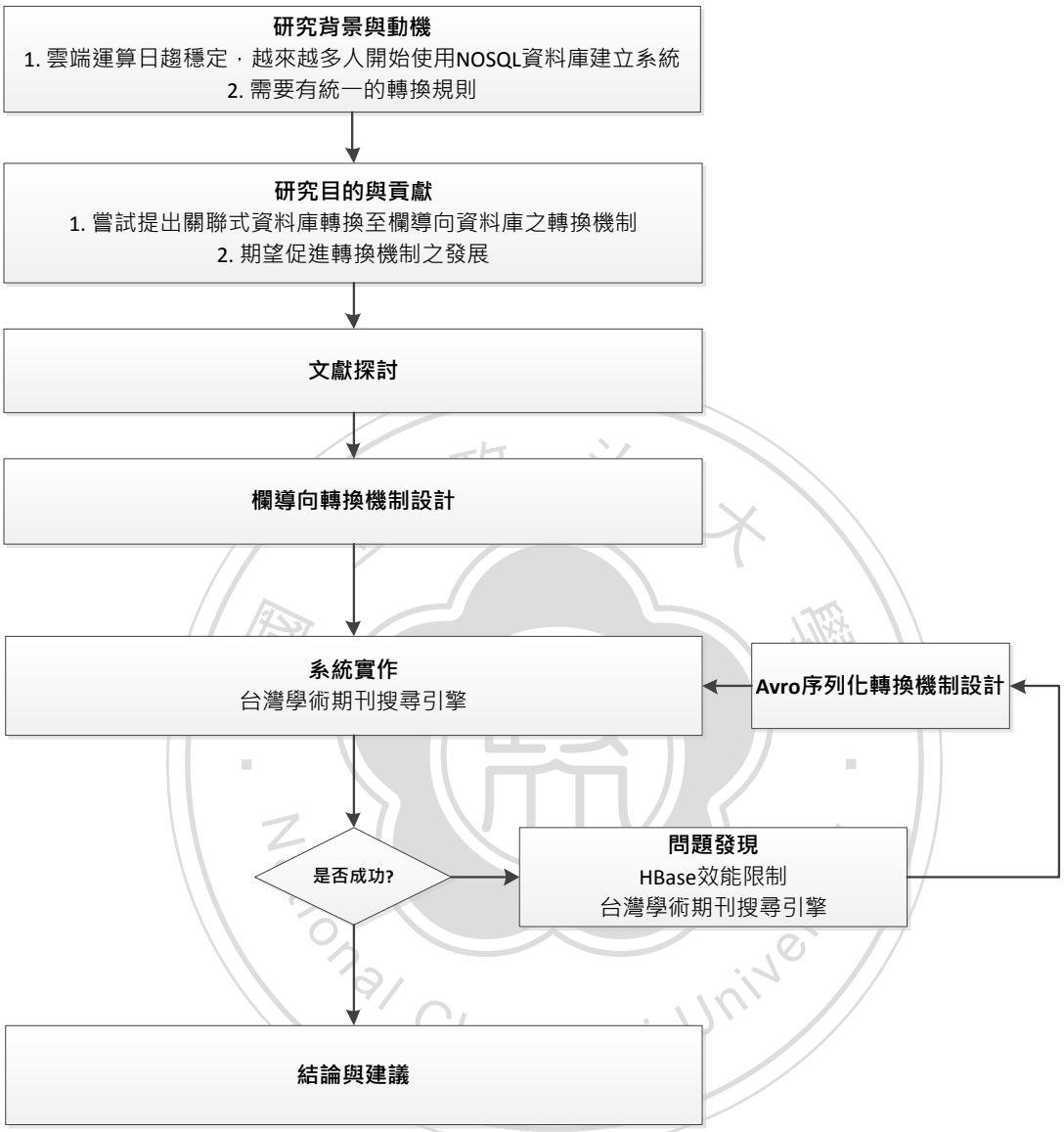


圖 2. 研究流程圖

本研究首先透過兩個現象點出研究背景與動機：

（1）雲端運算日趨穩定，越來越多人開始使用 NOSQL 資料庫建立系統。

（2）相關系統資料庫移轉時需要有統一的轉換規則作為參考設計。

而後提出本研究之目的與貢獻：

（1）提出由傳統關聯式資料庫轉換至欄導向資料庫之轉換規則。

（2）實作出台灣學術搜尋引擎以驗證轉換規則之可行性。

(3) 期望刺激轉換機制的制定與 Object-Relation Mapping(ORM)的軟體實現。

第二章介紹本研究所用到之技術：雲端運算、Google File System、Apache Hadoop、Hadoop Distributed File System、Map/Reduce、HBase、Avro 以及序列化技術 Avro。

第三章提出兩種轉換的機制－欄導向資料結構轉換以及 Avro 序列化結構轉換。但於第四章系統實作時發現，HBase 目前對於多 CF 的效能較差，因此重新思考提出以 Avro 序列化結構為基礎的轉換機制;並於第四章系統實作中，重新規劃架構。第五章針對本研究所提出的轉換機制與系統實作提出結論與日後改進之建議。

第三節 研究目的與貢獻

根據研究背景與動機可得知，雲端運算的發展已日漸穩健，使用雲端運算最具代表性的分散式架構 Hadoop 建置系統的企業也越來越多，而企業不只使用 Map/Reduce 處理大量資料，也使用分散式資料庫儲存大量資料，如此一來可降低企業所需之成本與人力。Hadoop 與基於 Hadoop 的 NoSQL 資料庫 Hbase 也因而蓬勃發展。根據 Apache power wiki 資料顯示，有越來越多的公司開始使用 HBase 來建構資料庫(Apache wiki，2012)，如 Facebook、Adobe、Trend Micro、Twitter 等知名公司皆採用 Hbase 以達到彈性、高速讀取寫入、隨機從硬碟讀取資料以及低延遲的優點。

這樣的特性對搜尋引擎而言尤為重要。Google 工程師發現，就算是 4 毫秒（大約是眨一次眼）的時間都太久了，即便是這一點點的耽擱也會導致使用者減少搜尋量。因此這也是本研究選擇實作搜尋引擎的動機與目的，期望幫助解決現有關聯式資料庫在應付龐大資料量搜尋時，速度下降的問題。

遠流出版公司的資訊長表示，以關聯式資料庫建立的搜尋引擎進行搜尋，其所花費的時間約為 6 秒。這樣的速度在現在的網路環境中已不敷需求，使用者需要的是更快、更精確的搜尋引擎，即使是一秒鐘的等待對使用者而言仍舊太久。目前的搜尋引擎龍頭 Google，其打敗眾人的關鍵也在於它迅速的回應，以及其精準的搜尋排序。

因此，本研究之貢獻有二，其一為利用 Hadoop 及 HBase 實作一分散式搜尋引擎，除提供以 Hadoop 建構學術期刊搜尋引擎之系統架構，也提升原先關聯式資料庫所無法達到的快速搜尋以及精確度。其二為提供一通用關聯式至欄導向之轉換機制，促進組織雲端化發展，也期望日後能發展出一套標準的轉換甚或是 Object Oriented Mapping(ORM)機制。

第二章　文獻探討

本章節首先就目前最廣為人知的雲端運算、雲端運算分散式架構 Hadoop、分散式儲存及 Map/Reduce 作概念性介紹。再細說本研究使用、運行於 Hadoop 上方的欄導向資料庫 HBase，包含其 Table 儲存格式、應用及其限制。

第一節 雲端運算

雲端運算為近年來學術界及業界相當重視的議題，市場甚至預期在未來五至十年內會有許多新的應用跟技術改變人們使用資訊科技的方式。雲端運算自分散式平行運算與網格運算發展出來，專注於大量資料的密集處理，需要充足的運算資源，並將複雜的運算與儲存工作分散到網路雲端並隱藏起來。維基百科表示，雲端運算是一種基於網際網路的運算方式，透過這種方式，共享的軟硬體資源和信息可以按需求提供給運算機和其他設備。而雲端運算應提供基於虛擬化技術的服務，使使用者能快速部署資源，並按需求及其資源使用量付費。除此之外，雲端運算使用戶可方便地通過網際網路獲取海量信息處理之服務，並降低用戶對於 IT 專業知識的依賴(Wikipedia，2012)。

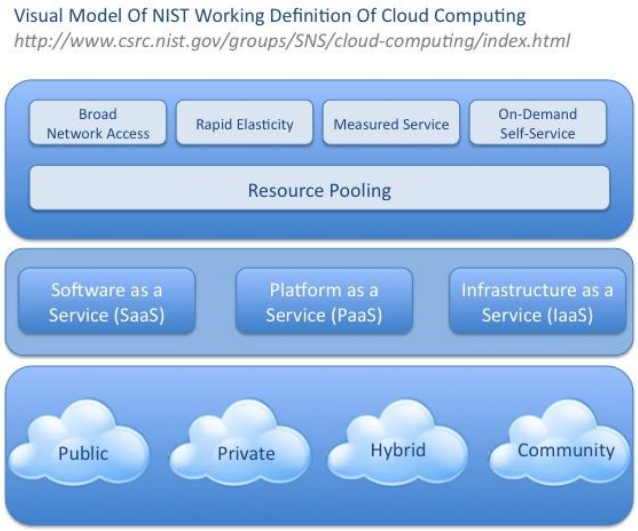


圖 3. NIST 美國國家標準局對雲端運算的定義

根據 National Institute of Standards and Technology (NIST)定義，雲端運算為使

用無所不在、便利、隨需應變的網路，共享廣大的運算資源，如網絡、伺服器、儲存、應用程式以及服務等，可透過最少的管理工作與服務供應者互動，快速提供各項服務(Mell and Grance ，2011)。

這些服務可以包含五大基本特徵(Essential Characteristic)及四個佈署模型(Deployment Models)與三種服務模式(Service Model)。

五項基本特徵分別為：

1. On-demand self-service：使用者可依自己的需求直接於網路上取得所需之雲端服務，如網路硬碟或虛擬伺服器等服務，而不需經過人工作業的方式。
2. Broad network access：使用者可以使用電腦、手機或是更小的部件，以標準的溝通機制透過網路取得服務。
3. Resource Pooling：多人共享資源，如頻寬、儲存空間、運算資源以及記憶體。
4. Rapid elasticity：使用者能夠彈性且快速地重新佈署他們所需的服務。
5. Measured Service：服務是能夠被監控與測量其狀態的。

四種佈署模型分別為：

1. Private Cloud：意指企業自行建置雲端運算平台，其建置成本較為昂貴，但因為企業擁有伺服器控管的權限，所以在安全性及隱私權上的防護較佳。大型企業通常會建置企業本身的私有雲。
2. Public Cloud：意指建置於遠方租賃的伺服器或是虛擬服務平台，甚至是服務本身，使企業不用做資源或是伺服器的控管，並且可以彈性的調整租用量。但因為企業所有的資訊應用資料皆放置於遠端的公有雲上，因此在安全性與隱私權的威脅相對而言較高。
3. Community Cloud：意指多個組織間互相友善，合作建置共有的社群雲，使得組織間可以共享其他組織所釋出的資源以及分攤雲端的維護費用。

4. Hybrid Cloud：將以上三種雲混和即為混和雲，是較為複雜的結構，會出現這種現象通常為私有雲加公有雲。因為某些大型企業會有極大的資源處理需求，然私有雲的建置費用極其昂貴，因此會動態調用遠方的服務幫助其運算。

而三種服務模式則分別為：

1. Infrastructure as a service, IaaS：提供運算、儲存以及網路等基礎設備的服務，以提供內外部使用者存取之用。為了幫助內外部使用者存取使用，IaaS 通常透過虛擬化技術(Virtualization)來完成伺服器整合的基本作業。目前市面上的 IaaS 以 Amazon EC2(Amazon Elastic Cloud 2), Google Compute Engine 以及 IBM Smart Cloud 最廣為人知。
2. Platform as a Service, PaaS：服務提供商提供運算平台給外部開發人員或使用者，並提供整合的 API 及相關管理套件方便開發人員構建、開發以及佈署其系統，但平台管理成本相對昂貴。目前最有名的為 Google 所推出的 Google App Engine(GAE), Windows 推出的 Azure 以及 Amazon 的 S3。
3. Software as a Service, SaaS：用戶向服務提供商租用雲端應用服務，使用者透過多種溝通協定對其所租用的軟體進行操作或取得運算結果。所有軟體的管理以及運轉皆由服務提供商負責，對使用者管理負擔以及成本的降低有不小的助益。

更深入而言，雲端運算是一種模式，其依照需求方便地存取網路上所提供的電腦資源，這些電腦資源包括網路、伺服器、儲存空間、應用程式、以及服務等，可以快速地被供應，同時減少管理的工作，降低成本並提昇效能。

第二節 Google File System

第一章中提到，由 Google 所提出的 Google File System(GFS)分散式檔案系統，對於分散式計算非常重要；因為當資料被放置在不同的機器上時，需要檔案系統來做適當的管理以及備份。

Google 於 2003 年所提出的 GFS 是一種相當容易擴大檔案系統容量的架構 (Ghemawat, Gobioff et al. 2003)，主要應用在大規模、分散式以及需要對大量資料進行運算的應用。GFS 的運行並不需要企業等級的高階伺服器，它的特點在於能夠運行在一般使用者的 Personal Computer(PC)上，雖然一般 PC 在損壞率上較伺服器來得差，但 GFS 提供了相當完整的容錯備援機制，使得即使在較差的環境下依然能夠快速地運行。這樣的一個特性使得企業或一般使用者在不需要花費大量資金的情況下也能夠獲得企業級伺服器的運算能量，節省企業資源也使得原本應該要廢棄的老舊機器得以有再被利用的機會。

然而 GFS 與過往的檔案系統最大的不同點在哪呢？

1. 備援機制：硬體的故障是有可能經常性的發生，因為資料會經常性的存取，雖然在企業級伺服器上故障的機率較低，但是仍然是一定機率發生的。一旦故障，企業即會損失相當多的檔案以及花費高昂的價格來維修。然 GFS 建置於分散的系統上，提供了資料備份的機制，也使用 Heart Beat 的機制來隨時監控硬體狀況，當有問題時，會立刻提供預先儲存的備份資料來即時回應。
2. Big Data 資料儲存：GFS 的誕生是由於資訊量的爆炸產生，使得系統的儲存空間得以 Scale out 的方式增加，意即購置額外的 PC 或者硬碟即可增加儲存容量，不需更換既有的設備。同時 GFS 能對大型檔案做有效的管理，在處理大量資料時，能夠即時的回應。
3. 大部分對於存在 GFS 的處理模式為在檔案結尾處增加資料，比較少的情況是修改既有的資料。因為存在 GFS 檔案系統的資料通常都是很大的單一檔案，所以對一個檔案的隨機寫入操作通常是不存在的。取而代之的是大量的程式對

單一檔案進行依序的讀取或寫入動作，因此必須對此動作進行最佳化以及保證大量的程式同時寫入或是讀取時彼此之間不能混淆。

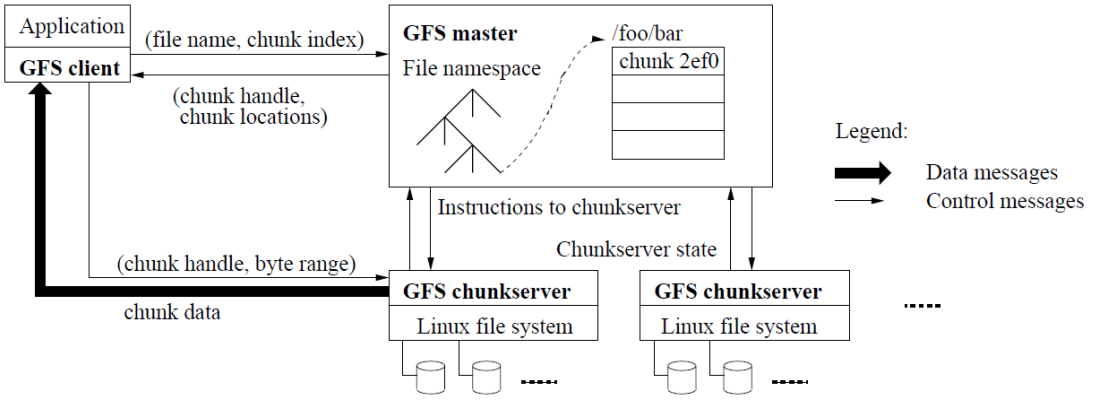


Figure 1: GFS Architecture

圖 4. Google File System 架構(引用自 Ghemawat, Gobioff et al.，2003)

如圖 4 所示，一個典型的 GFS 叢集是由一個 master 和多個 chunkserver 所組成，並且可由多個用戶端程式所存取。每一個節點都是在 Linux 的環境下，而 GFS 系統則是運行在該 Linux 系統下的程式。由圖 4 中可以知道 GFS 透過 GFS Master 來集中管理資料，其下的 chunkservers 可以是不同台的 PC，提供儲存的空間。GFS Master 也會隨時隨地的去偵測 chunkservers 的狀態，Hadoop 的 Hadoop Distributed File System(HDFS)即是從 GFS 衍生而來。

第三節 Apache Hadoop

Hadoop 是 Apache 軟體基金會(Apache Software Foundation)底下的開放原始碼計畫。Hadoop 以 java 為基礎，提供大量資料在分散式架構的環境下做運算以及儲存。Hadoop 的核心主要由 Hadoop Distributed File System(HDFS)以及 Map/Reduce 組成。

Hadoop 是由 Apache Lucene 專案中的搜尋引擎 Nutch 衍生而來。Nutch 為能夠爬網頁資料並提供搜尋功能的系統，不過 Nutch 的開發團隊逐漸發現建構一個網路搜尋引擎是一個雄心勃勃的目標，不僅是要撰寫一個可以處理複雜網站的資料、抓取以及建立索引的軟體，還需要團隊的大量財力支持。根據 Mike Cafarella 以及 Doug Cutting 的估計(White，2011)，一個提供一憶個網頁索引的硬體需要花費將近一百萬美元，每月的運行費用還需要三萬元的驚人數字。不過他們相信這是個值得開發的目標。很快地在 Nutch 營運一年後，也就是西元 2003 年，正當 Nutch 營運團隊一籌莫展之際，Google 發表了一篇關於 Google File System 的論文，Nutch 的創辦人 Doug Cutting 很快地發現這是一個幫助他們搜尋引擎變革的一個契機。

GFS 檔案系統可以解決抓取大量網頁和建立索引時所產生的檔案，並節省對檔案管理上的時間，這對 Nutch 團隊來說相當有幫助，他們便很快速地採用 GFS 的論文於系統中實作，產生 Nutch Distributed File System(NDFS)，也就是 HDFS 的前身。然而，Google 於 2004 年又提出了 Map/Reduce 的分散式運算架構，Nutch 於 2005 年中便將 Nutch 所使用的演算法移植到 Map/Reduce 的環境上面。

在 Nutch 計畫中 NDFS 與 Map/Reduce 的應用已經超過搜尋領域，於是 Nutch 團隊便從 Nutch 中抽離出一個 Lucene 的子計畫並命名為 Hadoop。直至現在，Hadoop 已經成為 Apache 的頂級計畫，成為一個多樣性且熱絡的社群。使用 Hadoop 的公司相當多，如 Yahoo!、Last.fm、Facebook，甚至是到現在 Google 也宣布與 MapR 合作採用 Hadoop 作為基礎運算架構。(Mell and Grance， 2011)

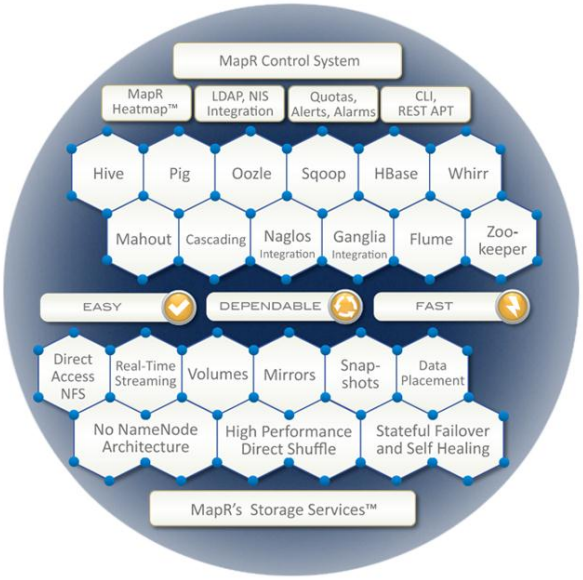


圖 5. Hadoop Ecosystem integration by MapR(資料來源：http://www.mapr.com/products)

Hadoop 的核心是由 HDFS 以及 Map/Reduce 組成，因為開放社群的關係，許多程式設計師開始針對 Hadoop 進行周邊套件的研發，並以動物的名稱來命名周邊的套件，如 Hive、Zookeeper、Sqoop 等套件。ZooKeeper 提供散式且高可用性的協調服務，為建置分散式系統提供分散式鎖定等原始鎖定功能。Pig 是超大資料集的資料流語言以及執行環境，可在 HDFS 和 MapReduce 叢集環境中執行。Hive 為分散式資料倉儲，透過 Hive 可管理存放於 HDFS 的資料，並提供根據 SQL 發展的查詢語言來查詢資料。Sqoop 提供 RDBMS 資料導入。Avro 提供高效能、跨語言以及可保存資料的 RPC 資料序列化系統。

如圖 5 所示，這些周邊的套件組合成為了 Hadoop 的生態系統(Hadoop Ecosystem)，但也因為各項套件獨立開發，在版本上面會有相容性的問題，所以許多商業化軟體的公司便開始整合 Hadoop 的生態系統套件，調整效能與佈署機制，如 Cloudera、MapR、Hortonworks 等。這樣的整合使得 Hadoop 更蓬勃發展，也日趨穩定，成為了雲端分散式運算的翹楚。

第四節 Hadoop Distributed File System(HDFS)

HDFS 由 NDFS 衍生而來，唯一分散式的檔案系統，採用主從(Master/Slave)結構的模型，一個 HDFS 叢集是由一個 Namenode 和許多個 Datanode 組合而成的。其中以 Namenode 作為主要的伺服器，管理檔案系統的命名空間和使用者端對檔案的讀取操作；而叢集中的 Datanode 則負責管理儲存的資料。

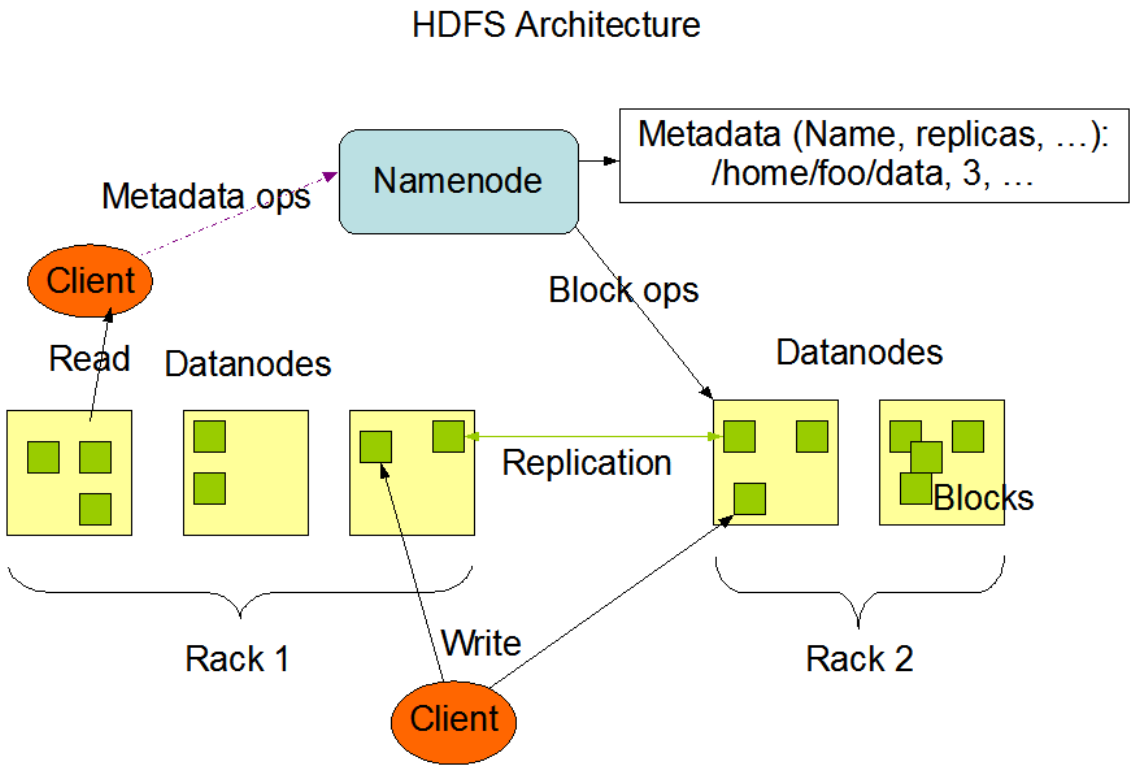


圖 6. HDFS 架構圖(資料來源：apache.org)

圖 6 為 HDFS 的架構圖，由圖中可得知 HDFS 會將資料以檔案的形式儲存，並切割或合併成多個檔案區塊，每個檔案區塊預設為 64MB。HDFS 以檔案區塊作為讀取的單位，每個檔案區塊的大小是固定的，並會複製數份，除了拿來做資料的容錯機制和可靠性外，也增加了執行 Map/Reduce 運算時取得資料的速度。讀取資料時，使用者會先訪問 Namenode 後，由 Namenode 告知資料所在的 Datanode，再由使用者直接連線到該 Datanode 取得資料。

第五節 Map/Reduce

Map/Reduce 是一種平行程式設計模式，這種模式使得軟體開發者可以很輕易的撰寫出分散式平行程式。Map/Reduce 是一個簡單易用的軟體框架，由一個運行於主節點上的 Jobtracker 和執行在每個叢集節點的 Tasktracker 所組成。Jobtracker 負責分配工作給底下的 Tasktracker 執行，Jobtracker 在接收到 Job 的傳送作業以及設定資訊後，就會傳送 Job 的設定資訊到底下的各 Tasktracker，並指定運算的作業內容。此外，Jobtracker 負責監控排程 Tasktracker 的執行狀況，使用心跳(Heart Beat)的方式間隔地詢問 Tasktracker 的執行情況，如果某節點的 Tasktracker 任務執行失敗就會重新將該任務分派給其他節點上的 Tasktracker。

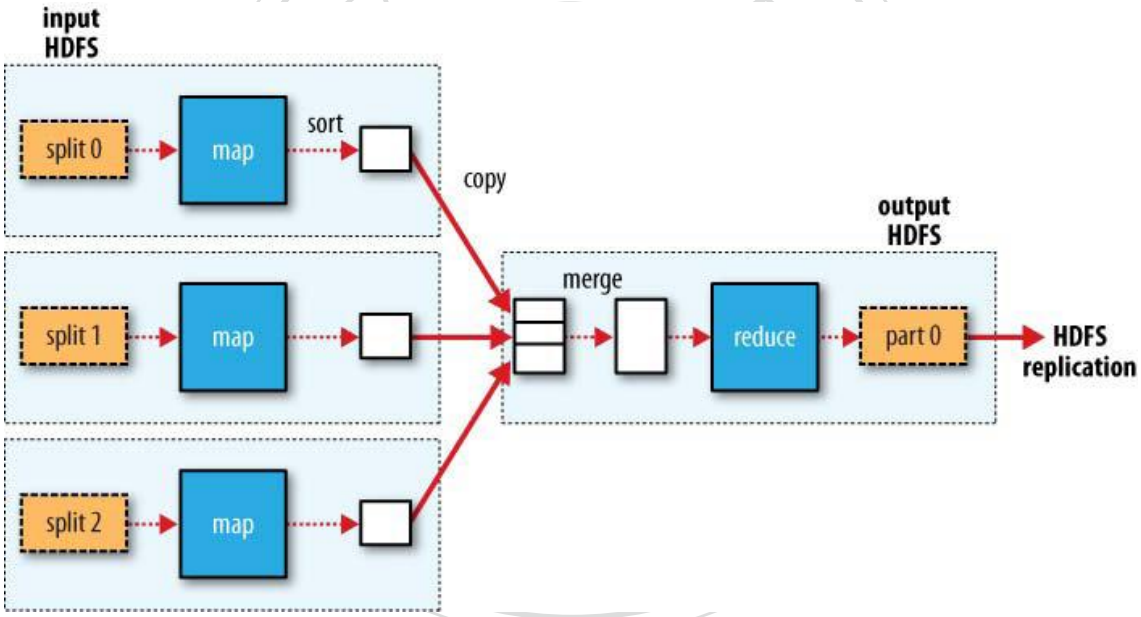


圖 7. Map/Recduce 運作圖(資料來源：White, T.，2010)

Map/Reduce 的任務分別由 Map 跟 Reduce 組成，在執行 Map/Reduce 任務時需先定義 input，也就是運算的資料來源；透過 Jobtracker 會將資料來源根據 Map 數量進行資料切割成為資料片段 Splits，並將各 Splits 分別傳送到執行 Map 動作的 Tasktracker 上。經過 Map 分散運算後的資料會以 Key Value 的方式丟出，並依據輸出的 Key 值進行洗牌(Shuffle)跟排序(Sort)；洗牌是 Map 任務與 Reduce 任務之間的資料流，而排序則是依據 Key 值來進行組合的動作。洗牌將 Map 任務輸出的 Record

複製到 Reduce 任務執行的機器上，當 Map 輸出的 Record 已經複製完成後，會針對檔案進行排序的動作；更正確來說，應該是合併的程序，用來合併 map 輸出且維持排序的順序。

第六節 HBase

HBase 是一個以 HDFS 為基礎的高可靠性、高性能、欄導向、可彈性伸縮的分散式資料庫，適用於使用者需要即時讀取或隨機存取大量資料時。HBase 專案開始於 2006 年，其參考了 Google 所提出的 Bigtable 概念並加以實作。其原先是 Hadoop 的子專案，後來獨立出來成為 Apache 的頂層專案。

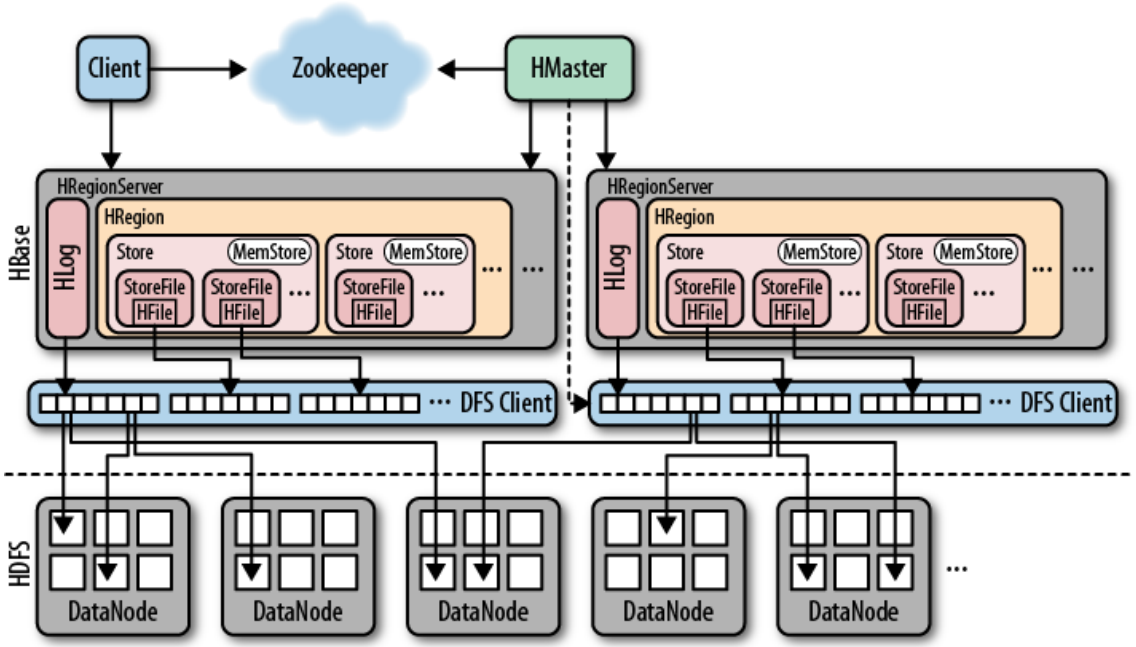


圖 8. HBase 架構圖(資料來源：George，2011)

由圖 8 的架構圖可以看出，HBase 也是類似 HDFS 的主從式(Master/Slave)架構，由一個 HMaster 以及多個 HRegionServer 組成。一個 HBase 可以佈署多個 Master，但同時間只有一台 Master 是活躍的，其透過領袖選取演算法(Leader Election Algorithm)來確保只有一台 Master 在工作。HMaster 負責初始化叢集、紀錄 Table 的區塊(Region)位置、監控 HRegionserver 的運行狀況。區塊(Region)指的是 Table 水平切割的區塊，HBase 會依照 Table 的大小，將其切割為數個區塊儲存，並將之分散在 HRegionserver 中。HRegionserver 則是負責儲存這些被切割的 Table 區塊。

HBase 的 table 是一個稀疏的、長期存儲的、多維度的、排序的映射表。Table 由 Column Family(CF)、Column Qualifier(CQ)、Row Key(RK)、Time Stamp 及 Cell

所組成。每個 Cell 可以視為是關聯式資料庫的一個值，Column Qualifier 可以視為關聯式資料庫的一個屬性，在 HBase 中將同樣類型的屬性歸類到同一 Column Family 底下。每個 CF 與 CQ、RK 以及 Timestamp 的組合會搭配一個 Cell 值，如此的一個搭配便形成了一張可儲存的大表。邏輯上，HBase 是一張大表，但實際上儲存的單位卻是以 Key Value 的方式儲存，Key 的組成是 CF 與 CQ、RK 以及 Timestamp 的組合，而 Cell 則為 Value。這樣的特性使得資料容易被切割且分散式的儲存在多個位置，適合於分散式環境運作。

當使用者想取得資料時，會先向 Zookeeper 詢問 HMaster 的狀況並把請求傳送給運行中的 HMaster，再由 HMaster 將資料區塊所在的 HRegionServer 位置傳回給使用者；使用者得知資料位置後再直接去該 HRegionServer 取得所需資料。整體而言，Zookeeper 扮演的角色為管理整個 HBase 系統運行情況的大總管，如果有任何一台 HMaster 故障，Zookeeper 即會馬上遴選出合適的 HMaster 來提供服務。

Row Key	Time Stamp	Column "data:"	Column "meta:" "mimetype"	Column "meta:" "size"	Column "counters:" "updates"
"row1"	t_3	"{"name":"lars","address":...}"		"2323"	"1"
	t_6	"{"name":"lars","address":...}"			"2"
	t_8		"application/json"		
	t_9	"{"name":"lars","address":...}"			"3"

圖 9. HBase Table 概念視圖(資料來源：George，2011)

如圖 9 所示，一個表可以想像成一組映射關係，透過 Row key，或者是 Row key 加 Timestamp 就可以定位一行的資料。由於是稀疏資料，所以邏輯上某些列可以是空白的。這樣的一個結構就形成了欄導向的資料庫。

第七節 Avro

Avro 是 Hadoop 的一個子項目，由 Hadoop 的創始人 Doug Cutting 領頭開發，當前最新版本為 1.7.0。Avro 是一個資料序列化(Data Serialization)系統，設計用於支援大批資料交換的應用。它的主要特點為：支援二進位序列化方式，可以便捷、快速地處理大量資料；支援動態語言(Dynamic Language)，使動態語言(如：Java, C#)可以更方便地處理序列化資料。

當前市場上有很多類似的序列化系統，如 Google 的 Protocol Buffers， Facebook 的 Thrift。這些序列化系統運行良好，完全可以滿足資料序列化應用的需求。因此 Avro 的開發，使許多人產生疑問：為何要重複開發一個全新資料序列化的系統？ Doug Cutting 撰文解釋道：Hadoop 現存的 RPC 系統遇到一些問題，如性能瓶頸(當前採用 IPC 系統，它使用 Java 自帶的 DataOutputStream 和 DataInputStream)、伺服器端和用戶端必須運行相同版本的 Hadoop、只能使用 Java 開發等問題。且現存的序列化系統自身也存在問題；以 Protocol Buffers 為例，它需要使用者先定義資料結構，然後根據這個資料結構生成代碼，再組裝資料。如果需要操作多個資料來源的資料集，那麼需要定義多套資料結構並重複執行多次上面的流程，如此一來就不能對任意資料集做統一處理。其次，對於 Hadoop 中 Hive 和 Pig 這樣的腳本系統來說，使用代碼生成是不合理的。並且，Protocol Buffers 在序列化時考慮到資料定義與資料可能不完全匹配，允許在資料中添加註解，這會讓資料變得龐大並拖慢處理速度。而其它序列化系統也存在如 Protocol Buffers 類似的問題。所以為了 Hadoop 未來的穩定性考慮，Doug Cutting 主導開發一套全新的序列化系統，也就是 Avro。Avro 於 2009 年加入 Hadoop 專案家族中。

以上透過與 Protocol Buffers 的對比，大致了解 Avro 被設計出來的原因及其試圖改善的部分；以下則著重於 Avro 設計細節。

Avro 依賴 Schema 來實現資料結構定義。Schema 定義每個 datum (Avro 支援的資料型態) 物件結構可以包含哪些屬性，因此可以根據 Schema 來產生任意多個

datum 物件。對 datum 物件序列化/反序列化操作時，Avro 都需要知道 Schema 的具體結構。因此，在應用 Avro 作為資料交換的一些場景下，如檔案儲存或網路通訊，都需要 Schema 與資料同時存在。Avro 資料以 Schema 來決定如何讀、寫(Apache, 2012)，且寫入的資料不需要再加入其它標識，使序列化時速度快且結果內容長度也相對較小，解決上述其他序列化系統序列化速度慢的問題。

Avro 的 Schema 主要由 JSON 物件表示，包含一些特定的屬性，用以描述某種資料類型(Type)的不同形式。Avro 支援八種基本類型(Primitive Type)和六種複雜類型(Complex Type)。

Primitive Type:

[1] Null： no value

[2] Boolean： a binary value

[3] Int：32-bit signed integer

[4] Long：64-bit signed integer

[5] Float：single precision (32-bit) IEEE 754 floating-point number

[6] Double：double precision (64-bit) IEEE 754 floating-point number

[7] Bytes：sequence of 8-bit unsigned bytes

[8] String：unicode character sequence

Complex Type:

[1] Records：紀錄，支援五種屬性。

A. Name：提供 Record 之名稱。

B. Namespace：用以識別名稱的域名。

C. Aliases：Name 的別名，為一陣列。

D. Doc：其餘資訊。

E. Fields：為一陣列型態，儲存多個 Field，每個 Field 可包覆一 Avro 複雜型態。

[2] Enums：列舉，提供 Name、Namespace、Aliases、Doc 及 Symbol 屬性。Symbol 為一 JSON 物件陣列，儲存多個常數。

[3] Arrays：陣列，提供 Items 屬性放置陣列內的元素。元素可以是 Avro 之複雜型態。

[4] Maps：地圖，提供一組 Key Value 的 Collection，Key 的資料型態必須為 String，Value 可以是 Avro 之複雜型態，使用 Values 作為其參數。

[5] Unions：提供 Fields 中型態之選擇，使 Field 可包含的型態不只一種，也可為 null。

[6] Fixed：用以限制型態內的長度，提供 Name、Alias 兩種參數。另提供 Size 作為 bytes 數的限制參數。

第三章　關聯式暨欄導向轉換機制

大部分的傳統關聯式資料庫(Relational Database, RDB)皆以 Entity-Relationship Model(ERM)設計 Table 的概念模型，因此本研究將從 ERM 的觀點來說明關聯式資料庫(RDB)暨欄導向資料庫(Column-oriented database, CDB)的轉換機制。ERM 實體之間存在三種關係，分別為一對一(1-1)關係、一對多(1-m)關係、及多對多(m-n)關係。假設一個實體 A 跟實體 B 發生關係，實體 B 又與實體 C 發生關係，這樣的一個關係鏈，我們稱之為實體 A 與實體 C 擁有遞迴關係(Recursive Relation, R-R)。除此之外，我們也定義兩實體間的 R-R 距離(R-R Length)為中間的實體數加 1。以上述之例而言，實體 A 與實體 C 的 R-R 距離為 2(中間的實體 B+1)。

本章節原先計畫依上述五種關係進行關聯式資料結構轉換至欄導向資料結構之探討，但因研究過程中發現 HBase 之先天效能限制，轉換過程中的資料結構表現不佳；因此另尋解決辦法，最後決定以 Avro 序列化之結構來進行轉換機制的探討。因此本章分為兩節：第一節介紹欄導向資料庫的結構轉換，第二節介紹 Avro 序列化資料結構轉換。

第一節 欄導向資料庫結構轉換

本研究提出五種標準的轉換規則幫助使用者從 ER 模型轉換至 CDB 的資料模型。包含：(1)實體關係轉換(2)一對一關係轉換(2)一對多關係轉換(3)多對多關係轉換。

第一段　實體關係轉換

ERM 中，一個實體代表一個關聯 Table。一個關聯 Table 包含一個主鍵(Primary Key)以及數個非主鍵的屬性(Field)；主鍵可能是由數個 Field 組成。在欄導向資料庫中，一個 CF 包含了多個擁有相同特性的 CQ，每個 CQ 在 CF 中都是唯一的。在關聯 Table 中，一個主鍵的值識別了一個特定的列(Row)。同樣的，在欄導向資料庫中，一個 RK 也識別了 HBase 中一張 Table 的特定列。因此，關聯式資料表的

中的主鍵值可以被視為是 HTable 中的 RK。而對於主鍵是由多個 Field 組成的實體 Table，我們將該主鍵列中的每個 Field 中的值以「：」作為區隔符號合併為單一的值。轉換方法如下所述：

假設：

- Let *R* be a relational database, that is $R = \{T_1, T_2 \cdots, T_n\}$ is a set of relational tables.
- Let HR be a column-based database, that is $HR = \{HT_1, HT_2, \cdots, HT_n\}$ is a set of HTables, where HT_i is the mapping table of relational table T_i, HT_i might be null when T_i is a relational table for a weak entity.
- For a table $T_i \in R$, we define that T_i has
  - Let pk^i be the primary key of T_i, pk^i may consists of one or more fields where are called primary key fields.
  - Assumes that $pk^i = \{pk_e^i\}, 1 \leq e \leq k$, be the set of primary key fields, where k is the number of primary key fields in T_i.
  - A set of non-primary key fields $f^i = \{f_d^i\}$, $k < d \leq q + k$, where q is the number of non-primary key fields in T_i. We have $pk^i \cap f^i = \emptyset$.
  - T_i has a set of tuples t_l^i. For any t_l^i, t_l^i has primary values $pk_l^i = \{pk_{le}^i\}$ and field values $f_l^i = \{f_{ld}^i\}$.

轉換步驟：

- For each table $T_i \in R$, an HTable HT_i with a set of row keys rk_l^i is constructed. The construction of HT_i includes two steps, the first step is to define the row key for HT_i and the second step is to define the column family for HT_i, which are described as follows.
  - Step 1: For each primary key $pk_l^i = \{pk_{le}^i\}$ in T^i, take $pk_{l1}^i : pk_{l2}^i : \ldots : pk_{lk}^i$ as the value of the row key rk_l^i in HT_i.
  - Step 2: A column family cf^i is constructed for f^i in HT_i, such that for every field $f_d^i \in f^i$, there is a column qualifier cq_d^i in cf^i.
  - For each tuple $t_l^i \in T^i$, a key value pair KVP_{ld}^i will be constructed for each f_{ld}^i of t_l^i, in which row key equals to t_l^i, f_d^i is the column qualifier and f_{ld}^i is the value.

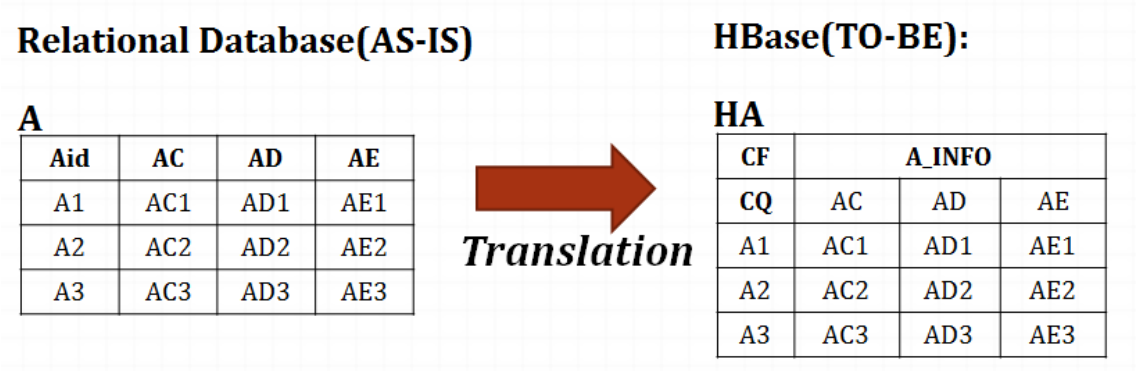


圖 10. 實體關係轉換示意圖(Entity Relationship Translation)

圖 10 中，創建了與實體 A 對應的 Htable－HA，HA 包含一個名為 A_INFO(使用者可以自行定義)的 CF，用以群組實體 A 的 Field，並將每個 Field 視為 A_INFO 中的一個 CQ。

為了方便定義轉換規則，我們預先定義 Table 的參數數學表示式如下：

- For the relational table A, we define that A has
  - Let pk^A be the primary key of A, pk^A may consists of one or more fields where are called primary key fields.
  - Assumes that $pk^A = \{pk_e^A\}, 1 \le e \le k$, be the set of primary key fields, where k is the number of primary key fields in A.
  - A set of non-primary key fields $f^A = \{f_d^A\}$, $k < d \le q + k$, where q is the number of non-primary key fields in A. We have $pk^A \cap f^A = \emptyset$.
  - A has a set of tuples t_l^A. For any t_l^A, t_l^A has primary values $pk_l^A = \{pk_{le}^A\}$ and field values $f_l^A = \{f_{ld}^A\}$.
- For relational table B, we define that B has
  - Let pk^B be the primary key of B, pk^B may consists of one or more fields where are called primary key fields.
  - A set of non-primary key fields $f^B = \{f_c^B\}$ in B, $z < c \le y + z$, where z is the number of non-primary key fields in B. We have $pk^B \cap f^B = \emptyset$.
  - A set of primary fields, $pk^B = \{pk_h^B\}, 1 \le h \le z$, be the set of primary key fields, where z is the number of primary key fields in B.
  - B has a set of tuples t_k^B. For any t_k^B, t_k^B has primary values $pk_k^B = \{pk_{kh}^B\}$ and field values $f_k^B = \{f_{kc}^B\}$.

根據以上定義的 Table 參數表示式，本研究定義單一實體的轉換規則如下：

entity(A)

假設:

- Let A be a relational table for translation.

轉換步驟:

- For table A, an HTable HA with a set of row keys rk_l^A is constructed. The construction for HA includes two steps, the first step is to define the row key for HA and the second step is to define the column family for HA, which are described as follows.
  - Step 1: For each primary key $pk_l^A = \{pk_{le}^A\}$ in A, take $pk_{l1}^A : pk_{l2}^A : \ldots : pk_{lk}^A$ as the value of row key rk_l^A in HA.
  - Step 2: A column family cf^A is constructed for f^A in HA, such that for every field $f_d^A \in f^A$, there is a column qualifier cq_d^A in cf^A.
  - For each tuple $t_l^A \in$ A, a key value pair KVP_{ld}^A will be constructed for each f_{ld}^A of t_l^A, in which row key equals to t_l^A, f_d^A is the column qualifier and f_{ld}^A is the value.

HBase 中，唯一識別資料列的只有 RK，因此使用者只能透過 RK 來取得資料，並不能做跨 table 間的 Join 或者是其他 SQL 查詢。在取得資料時，可以使用掃描(scan)每個 RK 的方式取得資料，並依需求加入針對 RK、CF 以及 CQ 的 Filter 來過濾資料；但因為是使用掃描的方式來取得資料，在資料量相當大的情況下，必須走訪每一列的資料，需花費相當長的時間來回應，因此效能相當的差。也因此，為了方便進行快速存取我們必須針對資料的欄位建立索引。

第二段　一對一關係轉換

除了上一段所闡述的實體關係轉換之外，實體間也存在著一對一的關係，因此本段將針對一對一的關係轉換來做探討。

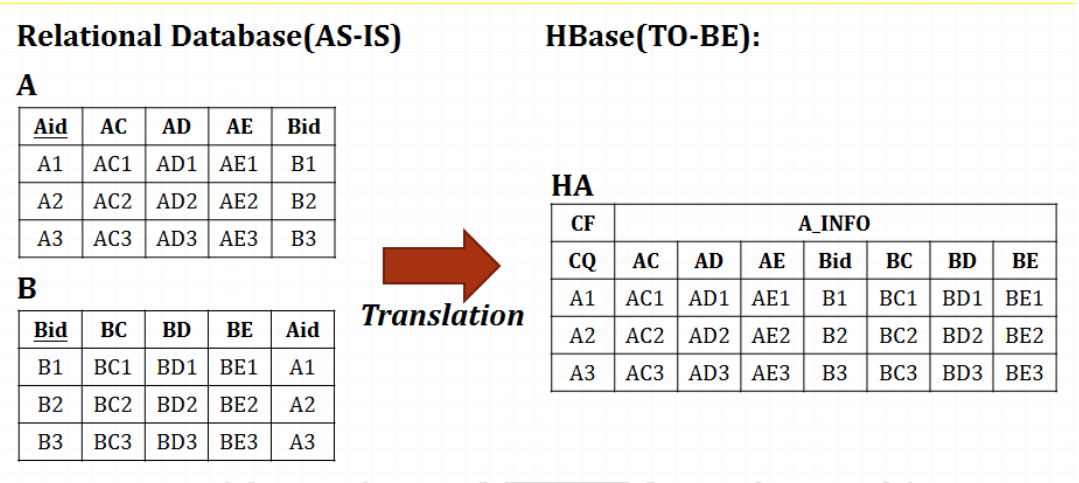


圖 11. 一對一關係轉換示意圖

一對一關係的兩個實體使用 Foriegn Key(FK)紀錄兩實體之間的關係。圖 11 中，A table 使用 Bid 作為 FK 紀錄 A 與 B 之間的關係。兩個擁有一對一關係的實體在關聯式資料庫中，會使用 Full outer join 儲存在一張 table 中，也就是使用其中一個實體的主鍵來當作該 table 的主鍵。因此，轉換規則就變得相當簡單，只要把由 full outer join 產生的 table 做實體轉換至一張 HTable。是否要對兩邊做 Full outer join 則依照使用者的需求。如圖 10 所示，將 A 與 B 做 full outer join 然後將其產生之 table 做實體關係轉換即可。一對一的關係又可以分為 1-0 以及 1-1，如果為 1-0 則以 1 方為主鍵；如果是 1-1 則使用者可以依照使用者需求選擇任一邊做為主鍵。

1-1(A,B):

假設:

- Assume A entity and B entity has 1-1 relationship, C is the actual table stored in the relational database taking A's primary key as C's primary key.

轉換步驟:

- The translation includes two steps as follows:
  - Step 1:

Generate C table by A table full outer join B table on A's foreign key equals to B's primary key.

- Step 2:

Execute entity(C) which accept C table as input and produce an HTable HC.

從以上的轉換規則我們得到 HA 如圖 11 所示，HA 拿 A 的主鍵當 RK 然後保留 B 的所有 Field 在 HA 的 A_INFO 中。

第三段　一對多關係轉換

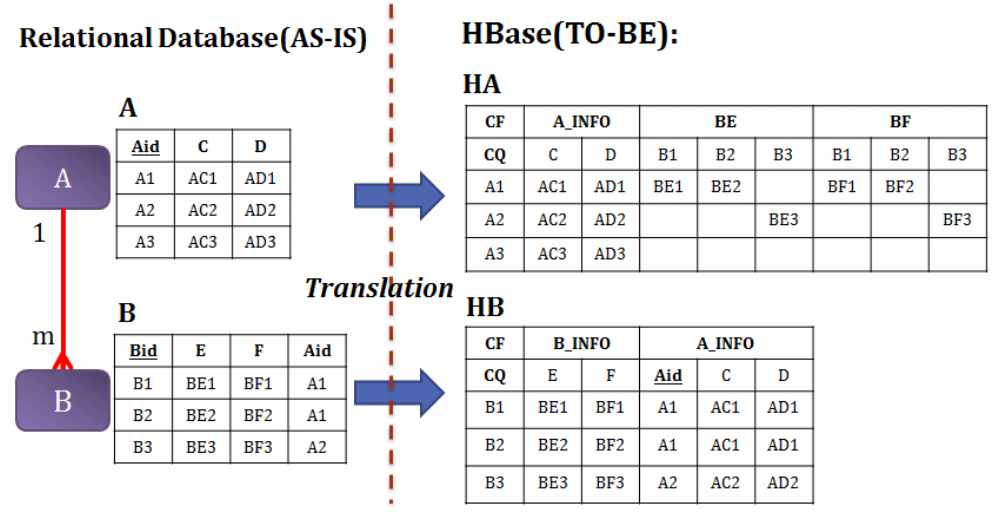


圖 12. 一對多關係轉換示意圖

在一對多的關係中，一方對於多方來說是一對一的關係。因此，對於這樣的關係我們可以先對多方進行一對一的轉換，也就是 1-1(B, A)，而得到 HB 如圖 12 所示。HB 紀錄了 B 連結 A 的外鍵所對應的資料，並使用 CF(A_INFO)來群組 A 的所有 Fields。

然而，在一方的觀點中我們必須去儲存多方的所有 Field，根據 HBase 的特性，CQ 是可以彈性延伸的，適合用以處理一對多的關係。因此我們創建一個 CF 給每個多方的 Field，然後以多方的 RK 來當作 CQ，儲存與多方的關係。這樣的轉換，稱之為一對多關係轉換。然而，多方的 RK 可能會由多個 Fields 組成，且由於我們使用 CQ 值來識別每筆多方資料的關係，CQ 值必須是唯一的。在這樣的情況之下，就必須合併多方 RK Fields 的值而形成單一值，這與一對一中 RK 值的合併是相同的。為了定義轉換的方便起見，我們首先定義一對多關係中一方的轉換規則－

1-m-single(A, B)，其中 A 為一方而 B 為多方。

1-m-single(A,B):

假設：

- Let A and B to be the 1-m relationship table in RDB, A is at the 1-side and B is at the m-side.
- HA and HB is constructed for the mapping HTables in CDB.

轉換步驟：

- Execute entity(A) to have HA.
- In HA, a column family cf_{Bc}^{A} is constructed for each B's non-primary key field f_c^B in f^B.
- Base on the 1-m relationship, a tuple t_l^A with primary key $pk_l^A = \{pk_{le}^A\}$ in A has a group of tuples $G_l^B \subset B$, such that for any tuple $t_{lk}^B \in G_l^B$, t_{lk}^B has pk_l^A as its foreign key.
  - For each non-primary key field of t_{lk}^B, a key value pair is constructed with pk_{le}^A as its row key, the primary key $pk_{l1}^B : pk_{l2}^B : \ldots : pk_{lz}^B$ as column qualifier in cf_{Bc}^A and take the value of f_c^B as the value.

由上述的一對多轉換規則中可以得知一對多的轉換包含一方的轉換與多方的轉換，一方的轉換與一對一關係的轉換規則相同。因此我們定義整個一對多的轉換為 1-m-multi(A,B)，轉換規則如下：

1-m -multi(A,B):

假設：

- Assume A entity and B entity has 1-m relationship, A is the 1-side and B is the m-side.

轉換步驟

- The translation includes two steps as follows:
  - Step 1:

    Execute 1-1(B, A), which takes B's primary key as row key generating an HTable HB.
  - Step 2:

    Execute 1-m-single(A,B), which generates an HTable HA preserving B's data related to A.

第四段　多對多關係轉換

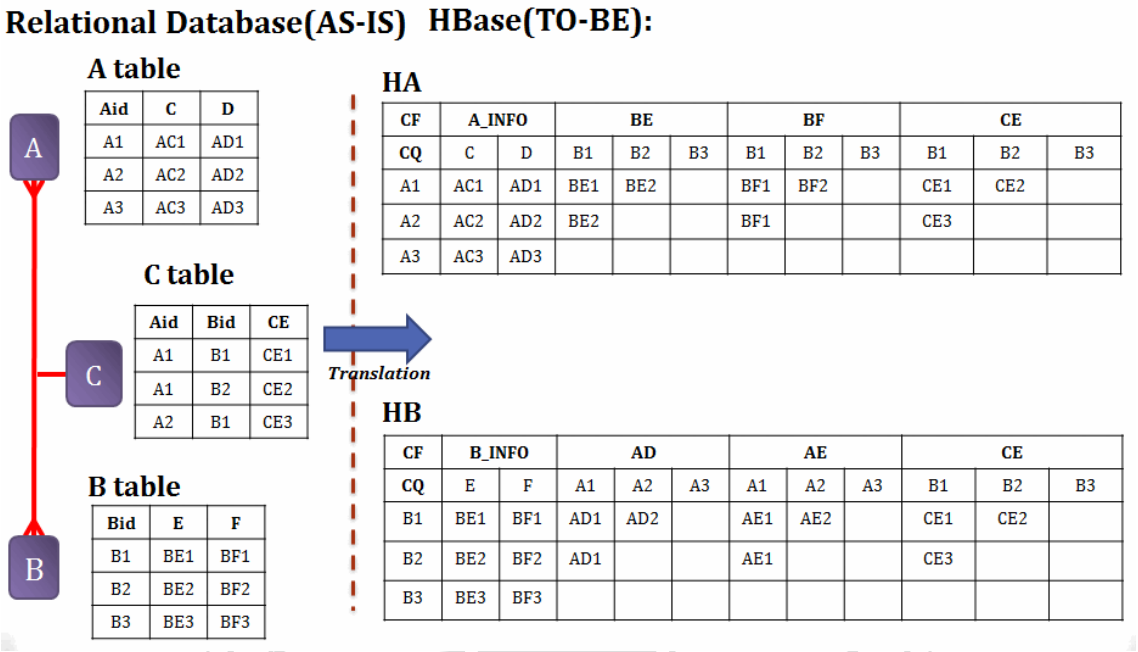


圖 13. 多對多關係轉換示意圖

多對多的關係可以視為兩個一對多的關係。因此對 A 與 B 分別進行兩個一對多的轉換。圖 13 中，A 與 B 擁有多對多的關係，其關係利用 table C 來記錄。經過兩個一對多的轉換後，HA 利用兩個 CF：BE、BF 來保留 B 的 Fields 資訊，HB 則是利用 AD 與 AE 來保留 A 的 Fields 資訊。然而，我們仍然必須儲存 A 與 B 之間關聯 table C 的額外資訊，如圖中的 C 的 Field CE。對於 table C 來說 A 與 C、B 與 C 也是一對多的關係，因此也必須針對 A 與 C、B 與 C 做一對多的轉換。在此我們定義多對多轉換規則為 m-n(B,C)，其規則如下：

m-n(A, B, C):

假設：

- Assume A entity and B entity has m-n relationship, C is relational table generated by A and B.

轉換步驟：

- Execute 1-m(A, B), 1-m(B, A).
- Execute 1-m(A, C), 1-m(B, C).

第五段　遞迴關係轉換

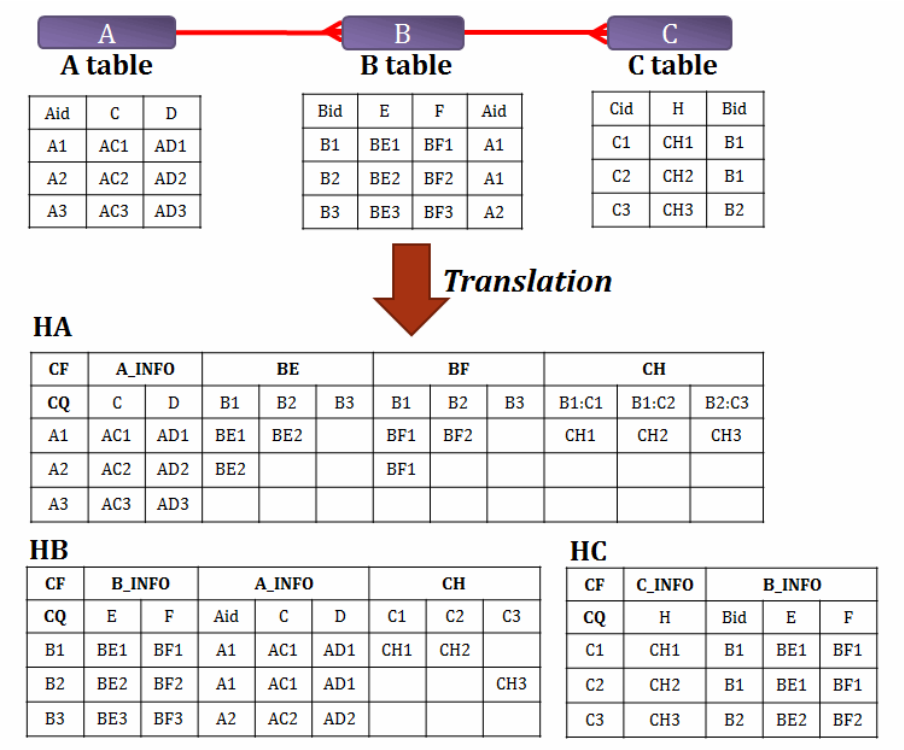


圖 14. 遞迴關係(R-R)轉換示意圖

第一段至第四段的轉換機制幾乎已經可以囊括所有的關係轉換，但是在 Google Bigtable 中提到，一個 table 應該要可以包含所有的資訊，為了增加效能，使用者必須從一次性的讀取中取得所需的資料。

也就是說假如有 A、B、C 三個 table，A 與 B 有一對多的關係，B 對 C 也有一對多的關係，我們希望透過 A 來取得 C 的資訊。雖然在上述的 4 項轉換規則中已經將所有的關係描述定義了，但為了增加效能，有別於過去的多個 table join，我們希望能夠透過一次性的讀取取得需要的資料。意即，能夠透過一次性的讀取 A 就取得 C 的資料，這樣一來 A 就必須保留 C 的資訊。上述的 ABC 關係，稱之為遞迴關係。

我們可以將遞迴關係解釋為多個一對多關係的連結，首先處理所有 table 一對多關係的轉換。如圖 14 所示，我們可以發現 HTable HB 儲存了對應的 C 的所有資訊。對於 A 來說識別 C 的資訊必須透過 B，因此透過 B 與 C 主鍵的結合可以

用來識別相對應的 C 關係。又 C 對於 A 來說也是多方的關係，因此可以拿 C 的 Fields 來當作 A 的 CF，B 與 C 主鍵的結合來當作 CQ。以上的動作可以透過遞迴地 Full outer join 的所有子 table 的方式來達成。以上述的 ABC table 為例，首先創建一個 table D 來暫時儲存 full outer join 的結果。再遞迴地把 D full outer join 子 tables(table B、table C)，如此一來 table D 便會是一張以 A 為主鍵並包含 C Fields 的大表，只要透過存取這張大表即可取得 C 之資訊。最後再對這張大表進行實體關係的轉換即完成遞迴關係轉換。以數學式定義遞迴關係的轉換規則(r-r)如下：

r-r translation(A, C):

假設：:

- Assume that A and C are two relational tables having the R-R relationship with R-R length n. Let A be the 1-side and C be the m-side.
- Let $RR_C^A = \{T_r\}, 1 \leq r \leq n-1$ be the set of relational tables in the R-R relationship between A and C, excluding A and C.

轉換步驟：

Construct a temporary relational table D = C.

- for (i = n-1; i < 1; i--){

  D = full_outer_join(T_i. D)

}

return 1-m(A, D)

第二節 Avro 序列化資料結構轉換

前一節中介紹了關聯式資料結構轉換至欄導向資料結構之機制。然而，HBase 官方文件中指出：「HBase currently does not do well with anything above two or three column families so keep the number of column families in your schema low.」(The Apache HBase Book，2012)。且於系統實作中發現，資料的寫入及掃描速度相差十倍之多，因此希望將 Column Family(CF)以及 Column Qualifier(CQ)的數量減少，以增加存取的速度。

而最直覺的做法就是將所有欲紀錄的資訊壓縮到一個 CQ 中。因此本研究期望透過序列化的技術將資料壓縮，儲存於單一 CQ 中。在文獻探討中，可以得知序列化的技術相當多，序列化技術 Avro 是 Hadoop 家族中專屬的序列化技術，設計用於支援大批量資料交換的應用。因此，本研究選用之於 Hadoop 整合性較佳的 Avro 序列化技術。

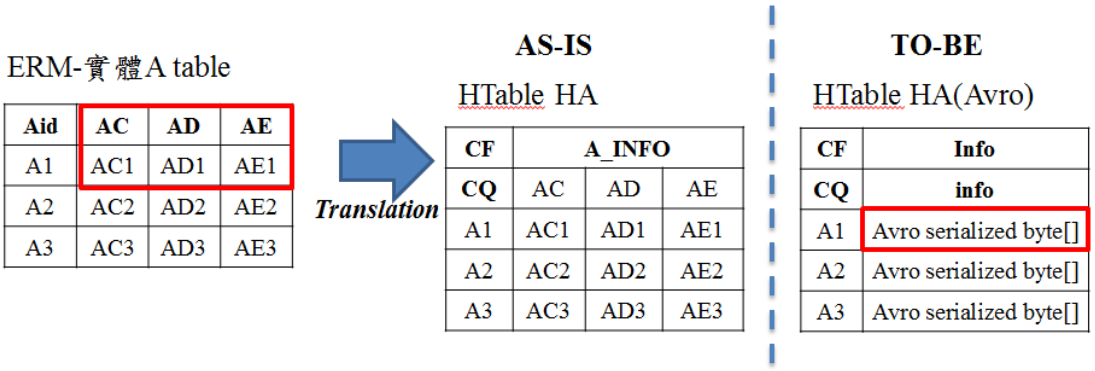


圖 15. 欄導向 HTable 資料結構轉換至 HTable 使用 Avro 序列化儲存示意圖

圖 15 是相較於上一節的資料庫結構設計的對照圖。AS-IS 是上一節中提到之實體關係轉換後的資料結構。但因 HBase 效能的關係，本研究期望將 CF 與 CQ 數量減少，因此利用 Avro 將圖 15 紅色框格的一列資料序列化為 TO-BE 中的 byte array，CF 與 CQ 的數量都降為 1，期望藉由此種作法增加讀取寫入之效能。

要將資料結構序列化，必須於 Avro 定義 Schema 結構，於文獻探討中，我們

得知 Avro 的資料型態支援八種基本類型(Primitive Type)和六種複雜類型(Complex Type)，因此本研究將依照 ERM 實體間的五種關係，並利用 Avro 所提供的型態來設計複雜的資料結構。

第一段　實體關係轉換

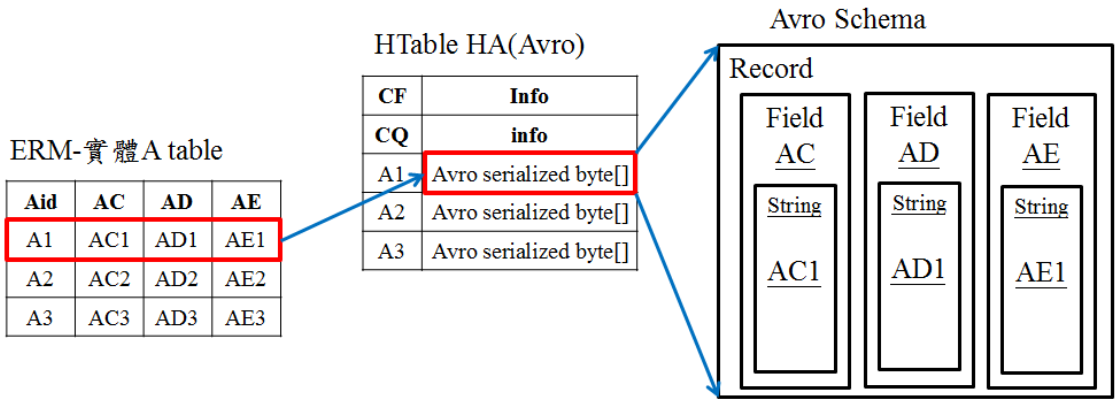


圖 16. Avro 實體關係轉換示意圖

圖 16 中最左邊為 ERM 的實體 Table A，擁有 AC、AD 與 AE 三個 Fields，紅色框格的一列會被序列化成一條 byte array 儲存至 HTable 中相對應的 RK 的 cell 中。如圖 16 所示，一列的資料會需要儲存 AC、AD 與 AE 三個 Field 的值，因此在 Avro Schema 中我們選擇使用 Record 來代表一個物件，而 Record 下的欄位則為關聯式 table 中的 Fields。因此，我們將得到三個 Field，分別為 AC、AD 與 AE，且其底下的資料型態設定為 String；此處的資料型態可以與使用者欲儲存的資料型態相對應。以圖片為例，其 Field 可能會有照相地點、曝光度、光圈等屬性；因此，一張圖片會是 Avro 的一筆 Record，而照相地點、曝光度、光圈為其底下的 Fields。又資料型態可依 Field 的類型設定，如照相地點可以設定為 String 而焦距則可以設定為 int。

第二段　一對一關係轉換

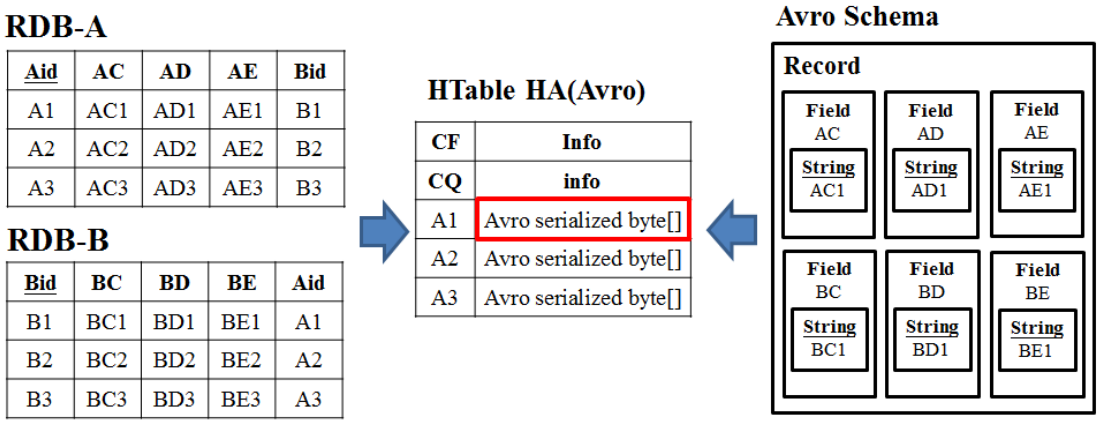


圖 17. Avro 一對一關係轉換示意圖

本段將探討 ERM 中 table 之轉換機制。圖 17 中左方為擁有 1-1 關係的兩個實體 A 與 B，於前一節中可以知道一對一的關係可以合併為一張 HTable 並由其中一實體的主鍵當作其 RK，至於哪個實體被當作 RK 則依使用者之需求來決定。為了儲存一對一的關係，B 的資訊也必須被儲存到 A 的 table 中。本轉換機制中，將其定義為三個步驟：

轉換步驟：

1. 假設 A 與 B 為 ERM 中具有一對一關係的兩個實體 Table，並欲以 A 為主要存取 table。
2. 對 A 做 Entity(A)之轉換，對於每個列資料可以得到一 Avro Record－R(A)。
3. 將實體 table A 中，外鍵所對應的 RK(B)擁有的所有屬性轉換為 R(A)中之 Fields，並將其值賦予給該 Field。

第三段　一對多關係轉換

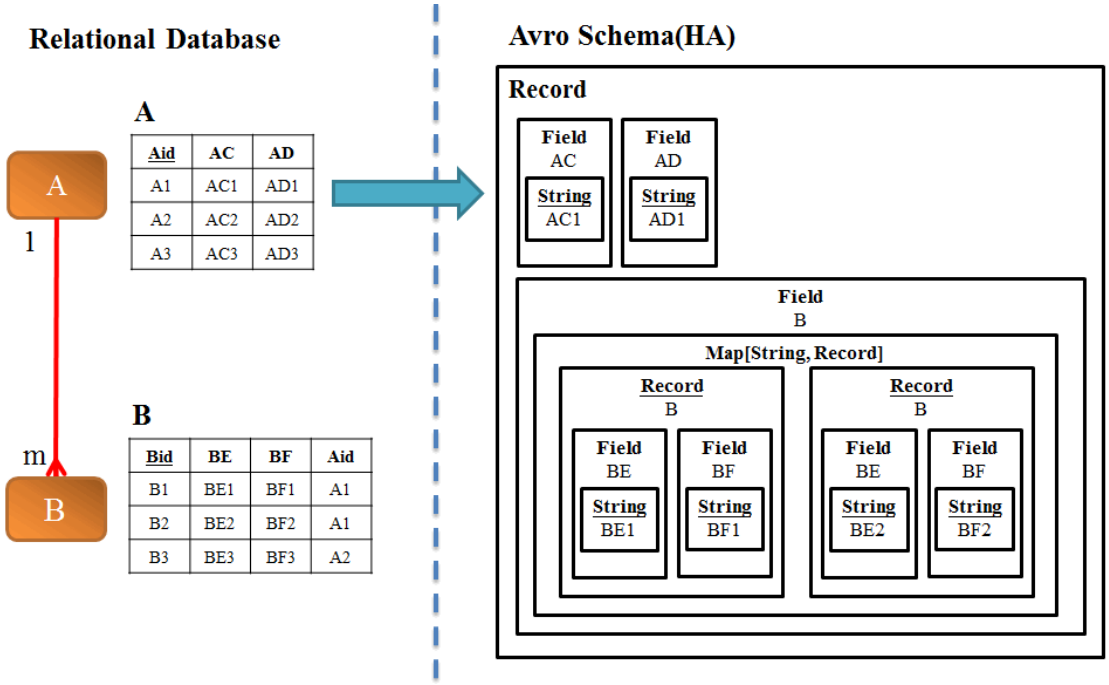


圖 18. Avro 一對多關係轉換示意圖-HA

圖 18 之兩實體 A 與 B 擁有一對多之關係，A 為一方而 B 為多方。轉換規則中，定義每個實體為 HTable 中之一 table，因此為了保存 A 與 B 之一對多關係，必須將關係分別儲存至 A 與 B 之 table 中。

對於 A 方來說，會擁有多個實體 B。簡單的來想，必須將多個實體 B 資訊儲存至 A 中，因此在 Avro Schema 中會先使用 Entity(A)的方式得一保存 A 資訊之 Record；而為了保留 B 資訊，本研究選擇 Avro 複雜形態中的 Map 來儲存多個 B 實體。因此在 A 中增加一 Field 並以 Map 複雜型態儲存 B 資訊。Map key 儲存對應 B 之 RK，Value 儲存 B 之 Record。如圖所示，Field B 儲存兩筆 B Record。

從上述邏輯推演，本研究定義一對多之單方轉換規則如下：

1-m-single(A, B)

假設：

1. 假設實體 A 與實體 B 擁有一對多關係，A 為一方而 B 為多方。

轉換步驟：

1. 對 A 進行 Entity(A)轉換
2. 進行 Entity(B)轉換，得到 B Record 並將此 Record 塞入對應 A 之 Map 中，B 之 RK 為 Map Key。
3. 回傳 A Record。

然而在一對多的轉換中，多方也必須轉換為一張 HTable 供使用者進行存取。根據 Google Bigtable 的觀念，應該於 table 中盡量儲存大量資訊，因此多方必須儲存一方之資訊。對於多方來說與一方是一對一的關係，因此第一步必須進行實體關係的轉換。對於多方來說，一的多方會擁有一個一方的 Record；為了保留與一方間的關係，另創一 Field 來儲存對應之 A 方 Record。

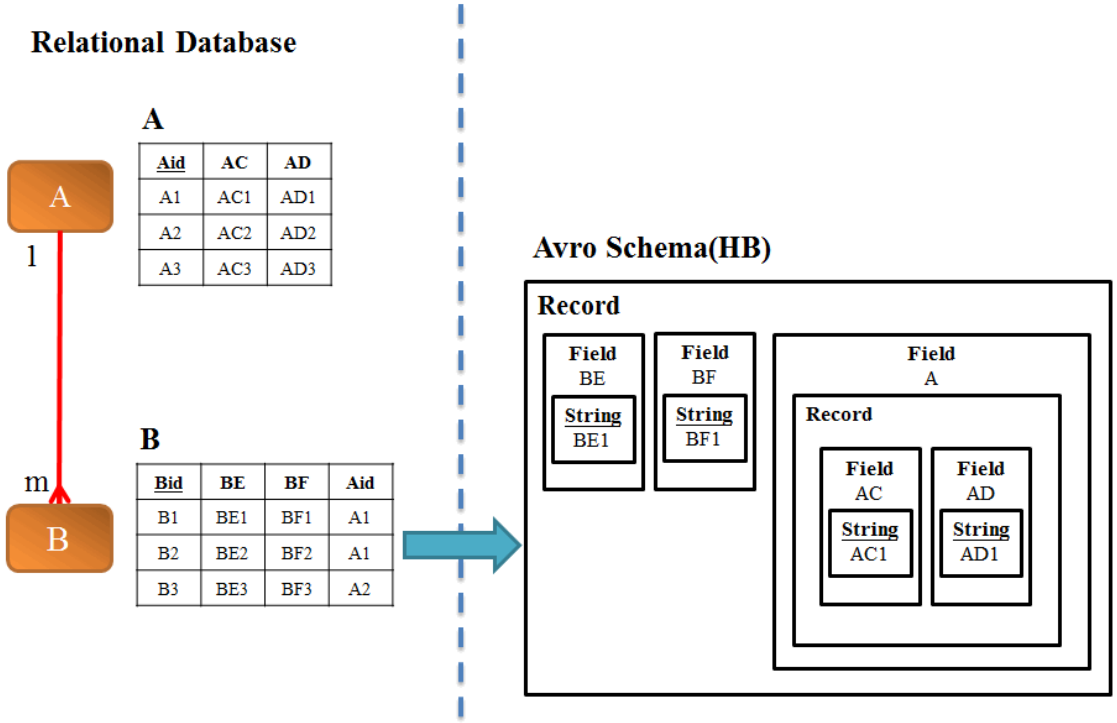


圖 19. Avro 一對多關係轉換示意圖-HB

如圖 19 所示，HB 擁有三個 Fields。BE 與 BF 的資料型態為 String，儲存原 B 之資訊，而 Field A 的資料型態為 Record，儲存經 Entity(A)轉換後之 A Record。A Record 為與 B Record 外鍵所對應之紀錄。

本研究定義轉換規則如下：

1-m-multi(A, B)

假設：

1. 假設實體 A 與實體 B 擁有一對多關係，A 為一方而 B 為多方。

轉換步驟：

1. 對 A 進行 1-m-single(A, B)轉換得到 table HA。
2. 進行 Entity(B)，得到 table HB，另創一 Field 儲存與 B 對應之 A Record。
3. 回傳 table HA、HB。

第四段　多對多關係轉換

多對多可以視為兩個一對多關係的轉換，因此可以視為兩個一對多關係的轉換，如果兩個實體間有一方為弱實體則不需進行該方之轉換。

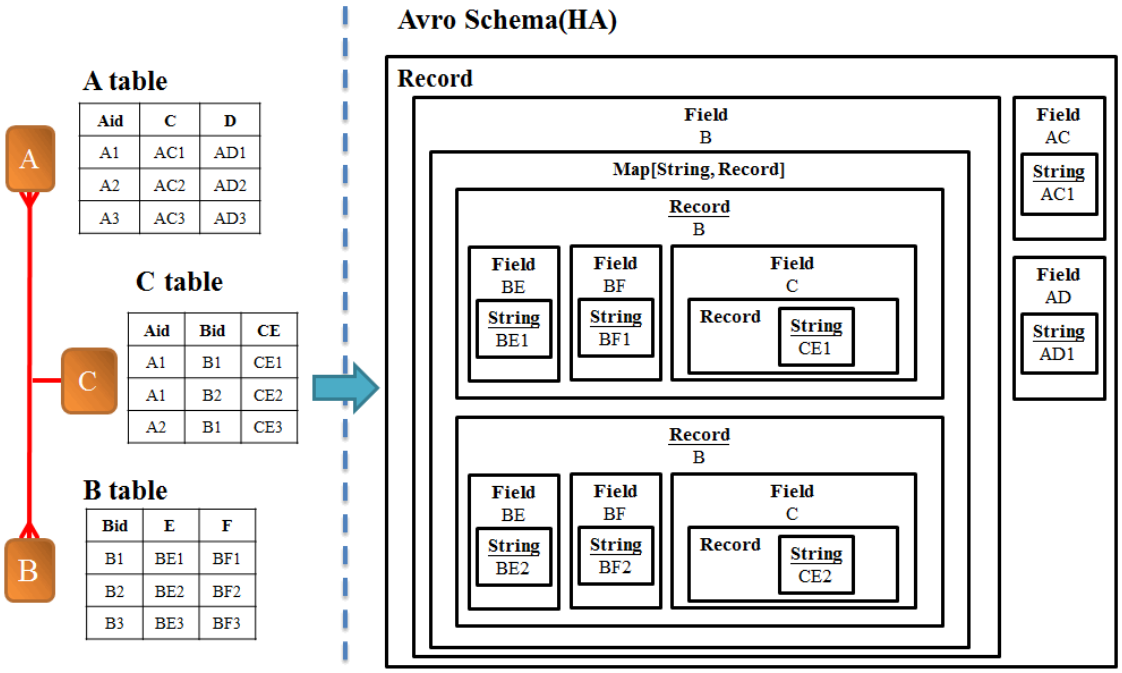


圖 20. Avro 多對多關係轉換示意圖

圖 20 為經 m-n 轉換後的 HA table。可以由圖中看出 Record A 利用 Map 形式來儲存與 B 一對多的關係。另外，由於 table C 仍有依賴於 A、B 的 C 資料，因此也必須記錄於 HA 與 HB 兩 table 中。對於 HA 來說 C 的資訊依靠 B 之 RK 而來，所以可以視為處理 B 與 C 之一對多關係。對於 B 來說 C 為一方，因此 C 將成為 B

下之一 Field。

本研究定義轉換規則如下：

m-n(A, B)

假設：

1. 假設實體 A 與實體 B 擁有多對多關係，兩實體皆不為弱實體。

轉換規則

1. 進行 1-m(A, 1-m(B, C))。
2. 進行 1-m(B, 1-m(A, C))。

第五段　遞迴關係轉換

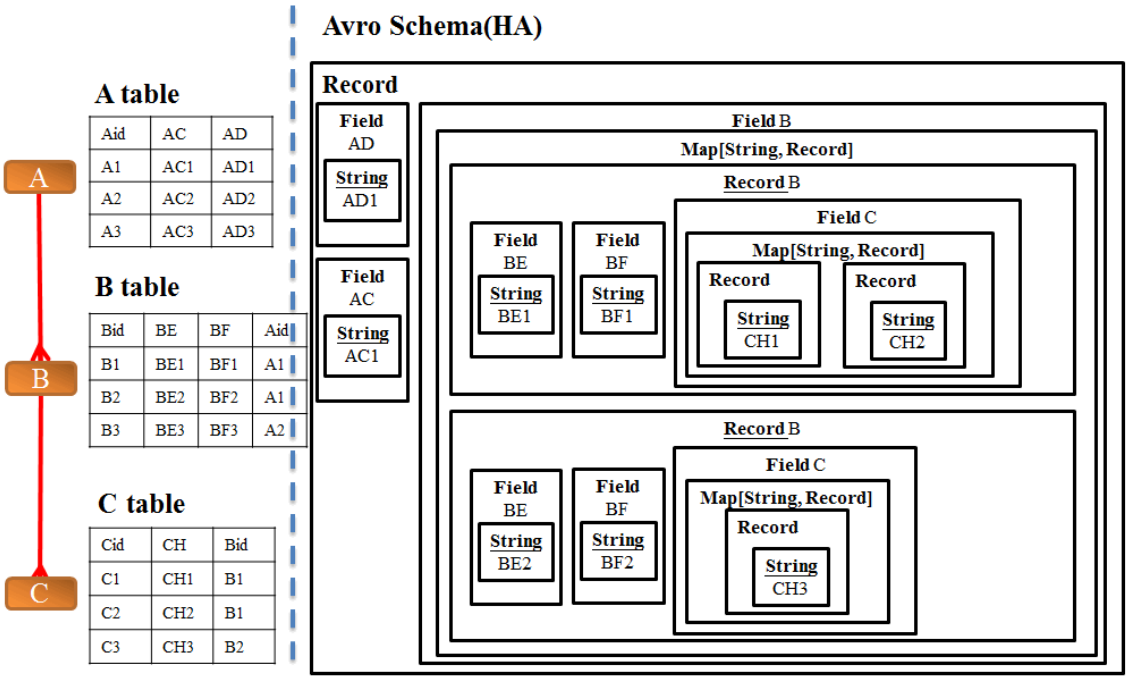


圖 21. Avro 遞迴關係轉換示意圖

如圖 21 所示，遞迴關係可以視為多個一對多關係轉換，以 ABC 三個具有遞迴關係的實體 Table 為例假設 Record T 為暫時儲存的 Record。先處理 B 與 C 的一對多關係，並將結果產生之 Record 賦予 T。由於遞迴之關係，T 具有與 A 一對多之關係，因此再對 T 與 A 進行一對多關係之轉換。具體轉換規則如下：

R-R(A, C)

假設：

1. 假設實體 A 與實體 C 擁有遞迴關係，實體 AC 之間具有多個一對多關係之節點。
2. 使得 $RR_C^A = \{T_r\}, 1 \le r \le n-1$ 為 AC 遞迴關係中之一組關聯 tables，不包含 A table 與 C table。

轉換機制：

1. 創建一暫存之 Record K。
2. for (i = n-1; i < 1; i--){

   T = 1-m(T_i. K)

   }

   return 1-m(A, K)

第三節 小結

HBase 屬於 NoSQL 資料庫的一種。NoSQL 資料庫的興起是源於現代典型的關聯式資料庫在一些必須快速存取大量資料的應用中表現非常差，例如巨量文檔建立索引、高流量網站的網頁服務、以及發送流式媒體等應用。關聯式資料庫的實現主要被調整用於執行規模小而讀寫頻繁，或者大批量極少寫入訪問的事務。NoSQL 的結構通常提供弱一致性的保證，如最終一致性，或交易(transaction)僅限於單個的數據項。

又因為 NoSQL 常被用來處理大量資料的存取，因此為了增加資料存取的便利性，本研究忽略了原先關聯式資料庫的避免資料重複存放(Data Redundancy)與資料一致性(Data Consistency)原則，使得各 table 可以獲取的資訊量增加。這樣的處理方式，能使在需要快速的回應或大量資料處理分析的系統，減少與磁碟 Input/Output (I/O)的次數，增加處理的效能。舉例來說，於欄導向資料結構轉換中，一對多關

係需將多方的資訊「完全」儲存在一方，這對於一方原本只需保存多方的 RK 來說是一種額外的資訊儲存，最明顯的錯誤就是資料重複存放，另一個錯誤就是資料的不一致。

但是在需要快速的回應或大量資料處理分析的系統，抑或資料不需經常性修改的系統中，資料的重複存放可以增加資料的存取速度，使得回應時間減少。舉例來說－搜尋引擎是需要快速回應的一種系統，其底下資料不需要經常性的變更以及修改。又於 GFS 的論文中提到，現在大量的系統多在 Appending 資料，而極少在做寫入的操作。因此在這種情況下，這兩種錯誤可以被有條件性的忽略，而交給使用者的應用程式來維護資料一致性。由此可得出一結論：少量、經常性修改的資料可以儲存在傳統關聯式資料上面；而不需經常性修改、被大量存取進行分析的資料可以儲存在 NoSQL 的資料庫中。

於本章所提出之轉換關係中，一對多、多對多及遞迴關係，皆存放各自關係中的實體，而於原先關聯資料庫中僅需存放 RK 即可。這樣的情形會造成上述提及之重複存放的問題，但因為所取之資料為不經常變動之資料，為了方便快速存取，依照使用存取需求的不同可以犧牲此兩種問題，或只進行 RK 的儲存。

第四章　系統實作

為了驗證轉換規則之有效性，本研究實作一台灣學術期刊搜尋引擎(Taiwan Academic Journal Search Engine)。本章的開端先介紹搜尋引擎所使用之技術。

第一節 使用技術暨環境

台灣學術期刊搜尋引擎所使用系統技術如下表：

技術類別	使用技術
前端畫面	JSP、Java Servlet
分散式環境	Hadoop
分散式演算法	Map/Reduce
資料庫	HBase
資料交換	Memcached
序列化技術	Avro

表 1. 台灣學術期刊搜尋引擎(TAJ)技術表

本搜尋引擎使用 JSP 做為前端畫面的呈現，底層使用 Hadoop 分散式環境佈署，並使用 Memcached 作為資料交換快速存取之中介。資料庫則是使用 HBase，並搭配 Avro 序列化轉換後之資料結構。

本系統所使用之 Hadoop 叢集環境如下：

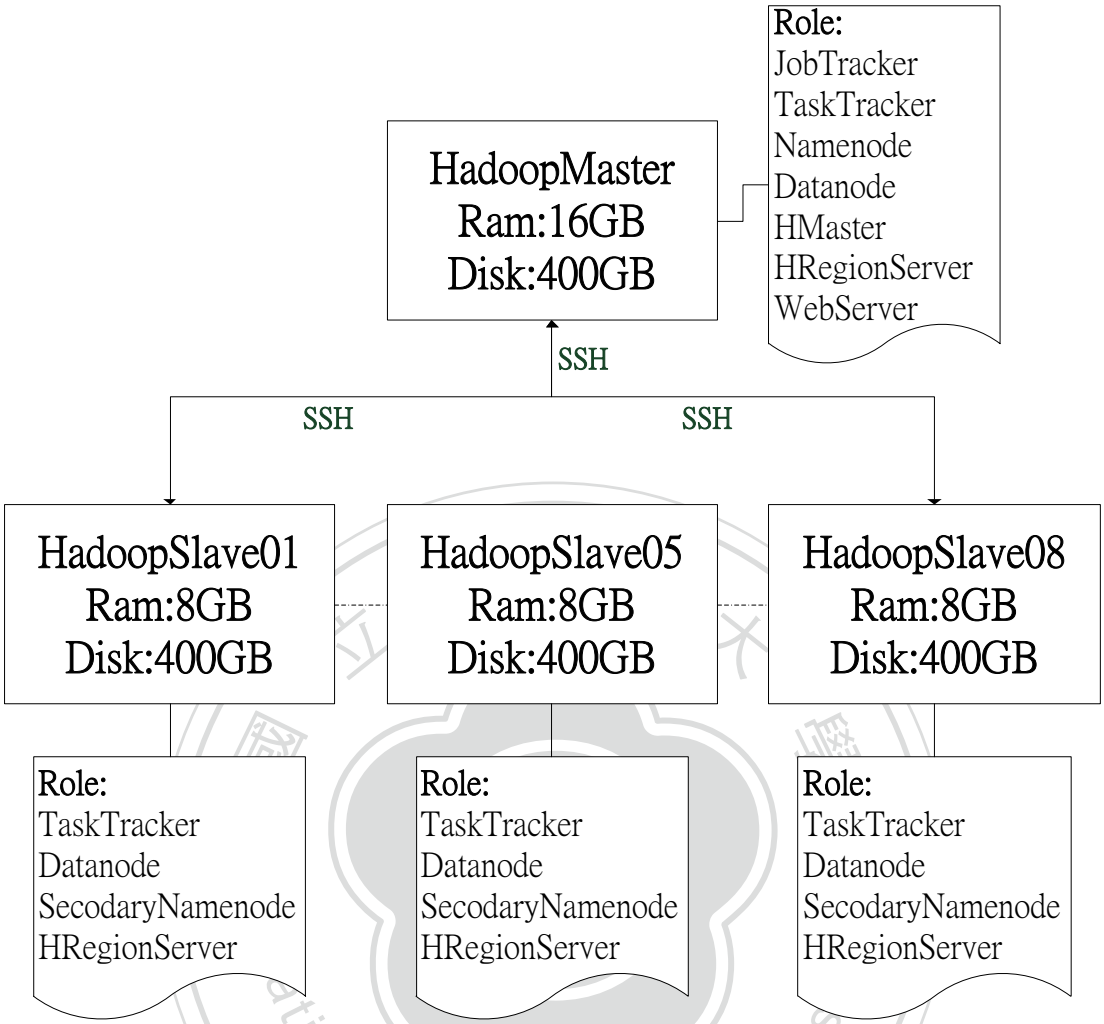


圖 22. Hadoop 叢集環境

本搜尋引擎使用 9 台 VM 作為 Hadoop 叢集之節點，主節點 HadoopMaster 擁有 16GB 之記憶體與 400GB 硬碟，8 台子節點 HadoopSlave 各擁有 8GB 記憶體與 400GB 硬碟。為充分運用硬體資源，本研究令 HadoopMaster 擁有 JobTracker、Tasktracker、Namenode、Datanode、HMaster、HRegionServer 與 WebServer 等 6 種任務角色，其餘 Slaves 則擁有 Tasktracker、Datanode、SecondaryNamenode 與 HRegionServer4 種任務角色。

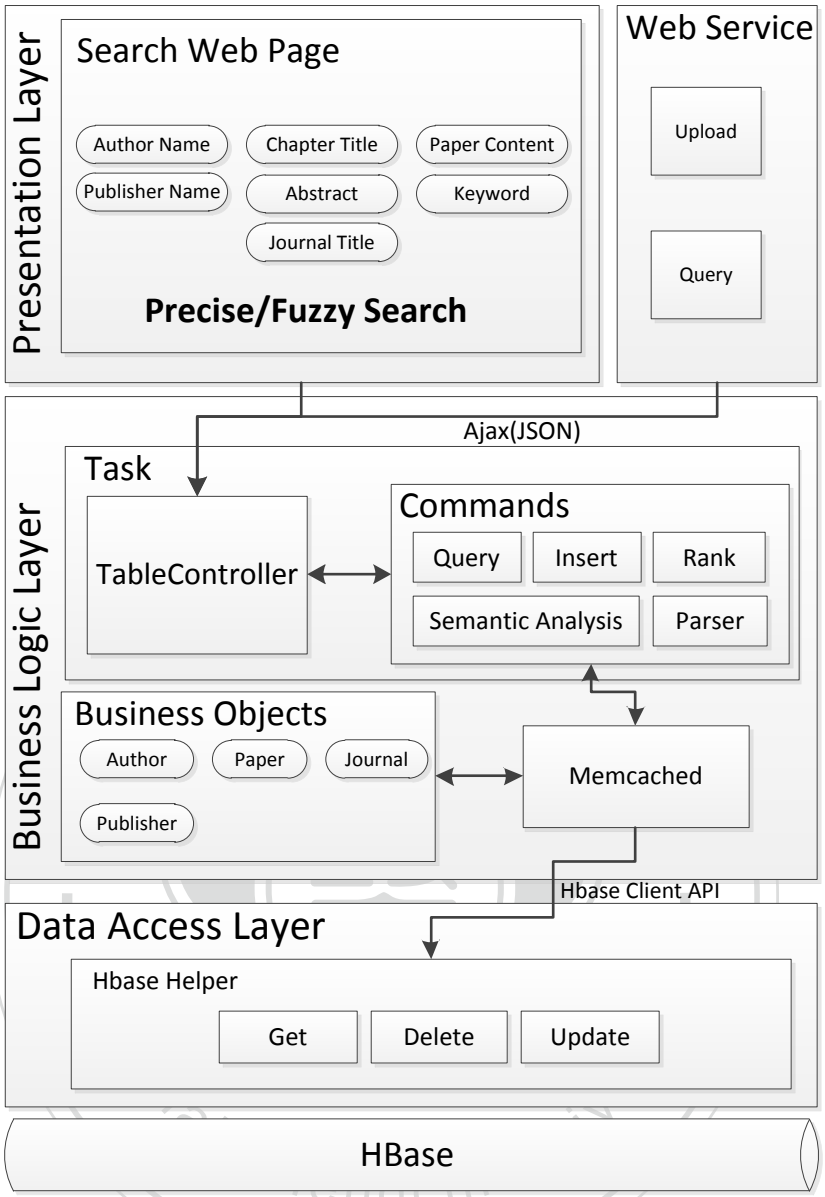


圖 23. TAJ 系統架構圖

本搜尋引擎系統架構圖如圖 23 所示，於前端提供兩種服務(一)學術期刊資訊搜尋介面(二)期刊資訊之增刪改查 Web Service；其搜尋之工作由 Task Controller 掌控並分派給其下之 Command 進行搜尋之動作。底層以 HBase 為基礎提供快速讀取寫入服務；此外，以 Memcached 做為資料交換區域，凡欲存取 HBase 之資料會先詢問 Memcached 該資料是否存在，如不存在則進入 HBase 進行搜尋。搜尋後之結果會存入 Memcached 以供下次同樣之搜尋。以下篇章將以這兩項服務作為主軸，詳細說明之。

第二節 學術期刊資訊搜尋介面

本搜尋引擎針對期刊論文內容(Paper)、期刊論文摘要(Abstract)、期刊論文抬頭(Paper title)、關鍵字(Keyword)、作者名稱(Author name)、期刊名稱(Periodical title)、與出版商名稱(Publisher name)等七項標的進行搜尋。各自擁有資料量如下表所示：

資料類型	筆數
Paper content	131,478
Paper title	131,478
Author name	118,642
Periodical title	2,268
Keyword	387,281
Publisher name	3,250
Abstract	131,478

表 2. 搜尋標的資料筆數表

表 2 為七項搜尋標的之資料筆數表，針對 13 萬餘筆台灣學術期刊論文資料進行檢索，各標的資料間關係如下圖所示：

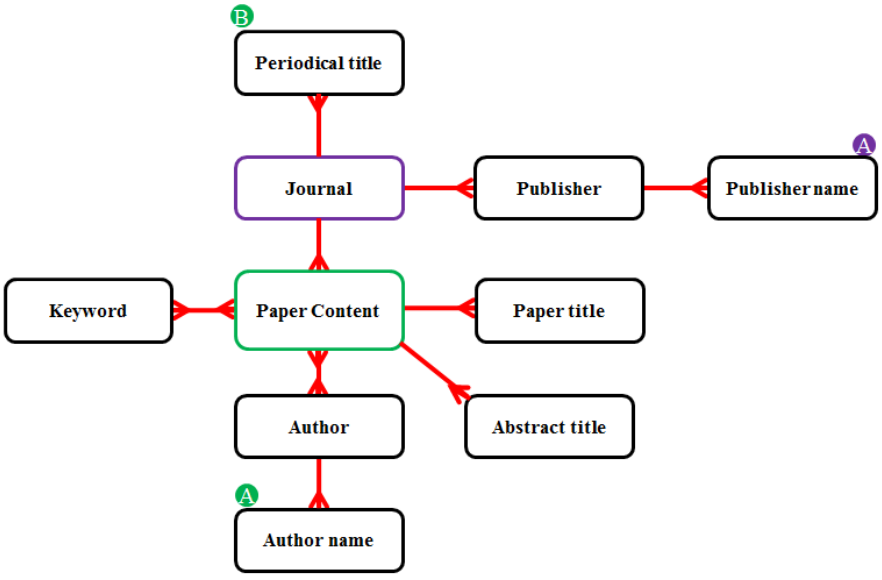


圖 24. 台灣學術期刊搜尋系統 ERM 圖

圖 24 中，綠色標號之節點，為與 Paper 具有 R-R 關係之節點；紫色標號之節點，為與 Periodical 具有 R-R 遞迴關係之節點。為了減少搜尋結果之時間，本研究期望於存取 Paper Content 實體時，也能同時存取此作者名稱與期刊名稱之資訊，而不需透過 Join 或者是多次的 Table I/O 存取。因此本研究欲將此三節點進行 R-R 關係之轉換。以下將依第三節所介紹之欄導向資料結構轉換，與 Avro 資料結構轉換分別作介紹。

第一段　欄導向資料結構轉換

Table 名稱	Table 欄位					
Object_title	object_id	object_title_language	object_title			
Object	object_id					
Paper	paper_id	text_id	chapter_id	content_id	text_language	content_text
Abstract	chapter_id	abstract_language	abstract_title			
Chapter_title	chapter_id	chapter_title_language	chapter_title			
Author_Title	object_id	author_title_language				
Keyword	paper_id	keyword_title				
Publisher_title	content_id	publisher_language	publisher_title			
Periodical	content_id					
Periodical_title	content_id	periodical_langauge	periodical_title			

表 3. 關聯式資料庫 Table 欄位表

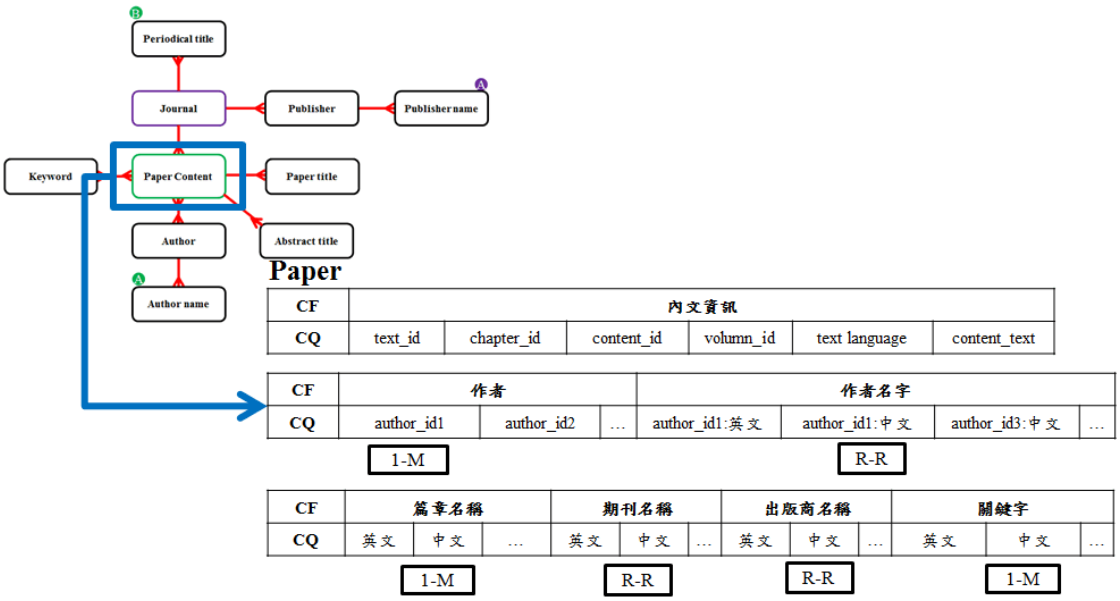


圖 25. 欄導向資料結構轉換-Paper

依圖 25 所示之 ERM，Paper 為主要存取之 table，亦即所有搜尋結果將會以 Paper 之內容作為最後呈現。因此根據第三節所闡述之概念先進行一對一與一對多之轉換：m-n(Author, Paper Content)、1-m(Periodical, Paper Content)、1-m(Keyword, Paper_content)、1-m(Paper Content, Paper Title)、1-m(Author, Author name)、1-m(Periodical, Periodical Title)。至於 R-R 關係之轉換，本研究依照需求(以 paper 為主要存取 table)選擇進行 R-R(Paper_Content, Author_name)、R-R(Paper_Content, Periodical Title)以及 R-R(Paper_Content, Publisher_Title)轉換。轉換結果如圖 26 所示。

而除了 Table Paper 以外之其餘 Table，則依照本研究之需求，保留三個基礎 table，分別為：(1)Periodical (2) Author 與 (3)Keyword。

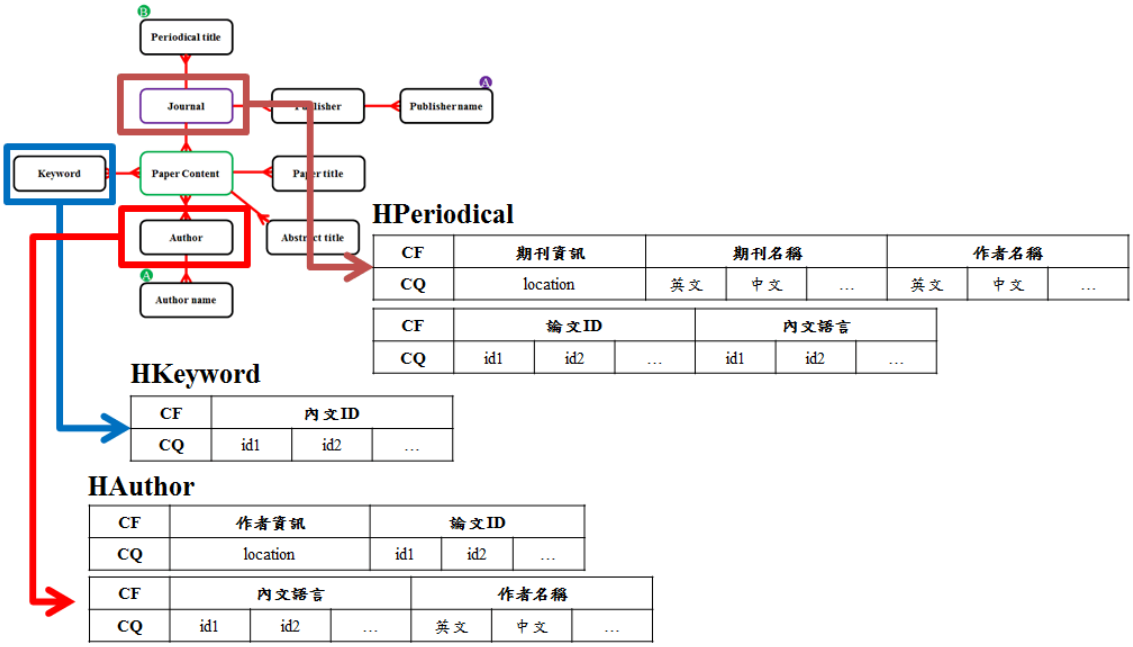


圖 26. 欄導向資料結構轉換-Periodical-Keyword-Author

圖 26 為此三基礎 table 之轉換結果。需注意的是，為避免資料量存放過大，此地只保留 Paper Table 之 id 於此三 table 中，而不進行一對多關係中，除主鍵之外的多方轉換。

第二段　Avro 序列化結構轉換

上節中闡述了兩種轉換機制，一為欄導向資料結構轉換，另一為 Avro 序列化資料結構轉換。本研究首先進行欄導向資料結構轉換後，發現效能不佳，後又於 HBase 官方文件中查到，HBase 目前在多個 CF 下的寫入及讀取速度不佳，因此必須使 CF 的數量維持在很少的數量。目前 HBase 在刷新與壓縮是以 Region 為單位做操作；也就是說，假如一個 HTable 具有兩個以上的 CF，其中有一個 CF 的資料量相當大，數筆資料就會被切割分散在不同台的 Regionserver 上；而另一個資料量較小的 CF 也會隨之被切割，分散在不同的機器上，如此一來會造成不必要的 I/O 刷新。因此如果在掃描資料量較小之 CF，或同時讀取大量小資料量的 CF 時，就會造成效率不佳的問題產生。因此本研究決定將 CF 與 CQ 數量縮小成各為 1，並採用 Avro 序列化 Schema 進行資料結構之描述。首先對 Table Paper 進行本身資料的實體關係轉換。如圖 27 所示，可以得知原先 paper 欄位皆成為 Avro 結構下之一 field。

```
{ "type" : "record",
  "name" : "Paper",
  "fields" : [
    {"name":"content_text", "type":"string"},
    {"name":"database_id", "type":"string"},
    {"name":"content_id", "type":"string"},
    {"name":"volumn_id", "type":"string"},
    {"name":"index_id", "type":"string"},
    {"name":"chapter_id", "type":"string"},
    {"name":"text_id", "type":"string"},
    {"name":"text_language", "type":"string"},
    {"name":"chapter_text", "type":"string"},
```

圖 27. Avro 序列化結構轉換-Paper-1

由於本搜尋引擎之 ERM 並無 1-1 之關係，因此便直接針對一對多，及 R-R 進行轉換。首先由 Author 與 Paper 開始，進行 m-n(Paper)，如圖 28 所示。

```
{"name":"Author","type" : "map",
        "values" : {
            "type" : "record",
            "name" : "AuthorRecord",
            "fields" : [
                {"name" : "AuthorNameLanguageRecordList",
                    "type":{
                        "type" : "map",
                        "items" : {
                            "type" : "record",
                            "name" : "AuthorNameLanguageRecord",
                            "fields" : [
                                {"name" : "language", "type" : "string"},
                                {"name" : "title", "type" : "string"}
                            ]
                        }
                    }
                }
            ]
        }
},
```

圖 28. Avro 序列化結構轉換-Paper-2

接著進行 Author Title 與 Paper 的 R-R 轉換－R-R(Paper, Author_title)，由圖 29 可得知 Author 下有 Author Record 物件，以此物件紀錄與 Paper 間的一對多關係；而 Author Record 下也有一 map field，以此紀錄 Author 與 Author Title 間的一對多關係。以下各圖為 ERM 轉換至 Avro 序列化資料結構後之 schema 結構。

```
{"name" : "JournalRecord",
    "type" : "record",
    "fields" : [
        {"name":"PublisherList",
            "type" : "map",
            "values":{
                {"type" : "record",
                    "name" : "PublisherRecord",
                    "fields" : [
                        {
                            "name" : "PublisherNameRecordList",
                            "type":{
                                "type" : "map",
                                "values" : {
                                    "type" : "record",
                                    "name" : "PublisherNameRecord",
                                    "fields" : [
                                        {"name" : "language", "type" : "string"},
                                        {"name" : "title", "type" : "string"}
                                    ]
                                }
                            }
                        }
                    ]
                }
            }
        }
    ]
},
```

圖 29. Avro 序列化結構轉換-Paper-3

```
    {"name":"Abstract_Title",
                "type" : "map",
                "values" : {
                    "type" : "record",
                    "name" : "AbstractRecordTitleLanguageRecord",
                    "fields" : [
                        {"name" : "language", "type" : "string"},
                        {"name" : "title", "type" : "string"}
                    ]
                }
    },
```

圖 30. Avro 序列化結構轉換-Paper-4

```
    {"name":"Keyword_Title",
                "type" : "map",
                    "values" : {
                        "type" : "record",
                        "name" : "KeywordTitleLanguageRecord",
                        "fields" : [
                            {"name" : "language", "type" : "string"},
                            {"name" : "title", "type" : "string"}
                        ]
                    }

    }
  ]
}
```

圖 31. Avro 序列化結構轉換-Paper-5

```
{"type" : "record",
 "name" : "PublisherRecord",
 "fields" : [
    {
        "name" : "PublisherNameRecordList",
        "type":{
            "type" : "map",
            "values" : {
                "type" : "record",
                "name" : "PublisherNameRecord",
                "fields" : [
                    {"name" : "language", "type" : "string"},
                    {"name" : "title", "type" : "string"}
                ]
            }
        }
    },
    {
        "name" : "JournalRecordList",
        "type":{
            "type" : "map",
            "values":{
                "name" : "JournalRecord",
                "type" : "record"
                "fields" : [
                    {"name":"content_text", "type":"string"}
                ]
            }
        }
    }
 ]
}
```

圖 32. Avro 序列化結構轉換-Publisher

圖 32 為另一範例轉換－Publisher。Publisher 如圖 25 所示，與 Journal 之間有一對多的關係，因此進行 1-m(Journal, Publisher)轉換；又與 Publisher Name 具有一對多之關係，因此再進行 1-m(Publisher, Publisher Name)轉換。轉換後之 Journal 便擁有兩 Field，一保存多個作者名稱，另一保留多個期刊間的關係。

本段以兩個轉換圖為基礎，顯示各實體關係、一對一關係、多對多關係以及遞迴關係。轉換後之 table 便可供搜尋系統進行資料之提取。然而經過轉換後之系統僅為搜尋時取得相關資料之用，無法針對內容進行搜尋。SQL 資料庫中可以透

過SQL-Like來進行資料比對，但HBase中所有存取皆需依賴RK的存取以及Scan，即使加上 Filter 來進行資料過濾，每次搜尋仍需搜讀全部 table 資料，因此在速度上是無法符合搜尋引擎之要求的。當然，序列化的資料更是不可能進行搜尋的。

因此，本研究提出了 Index Table 之建立方式：對各搜尋標的之資料進行斷詞，並將斷詞後之 Term 作為 Index Table 之 RK，供搜尋引擎進行搜尋。

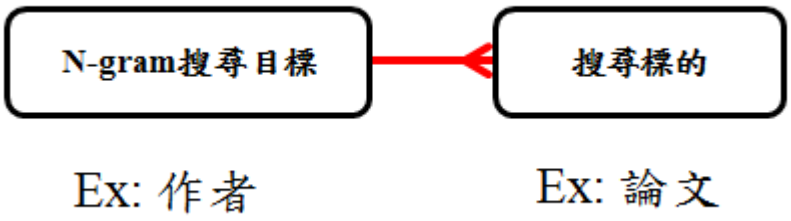


圖 33. N-gram 搜尋目標與搜尋標的 ERM

如圖 33 所示，Index table 之 Schema 設計，是以搜尋目標(Goal)及搜尋目標(Target)共同組成。以本搜尋引擎而言，搜尋目標即為 paper，而搜尋目標則為本章節所提出之 7 種搜尋標的。N-gram 搜尋目標與搜尋標的間的關係，為一對多，因此可根據第三章所描述之轉換規則，將此一對多關係轉換至如圖 34 所示之資料結構。轉換後之 Schema 設計如圖 34 右方所示。

Index Table For Search Target (Avro)

CF/CQ / RK (N-gram Terms)	Info
	info
Tammy	Avro serialized byte[]
Eric	Avro serialized byte[]
Eric(Huang)	Avro serialized byte[]

Avro Schema(Index)

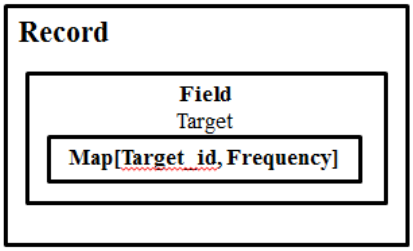


圖 34. Index Table Schema

本研究針對搜尋目標之內容進行斷詞，採用目前被廣泛使用的 Lucene 斷詞工具。為了搜尋之精確度，本研究採用 N-gram 斷詞，並以斷詞後之結果作為該搜尋

目標之 RK，其內容則記錄了該搜尋目標之搜尋標的及出現頻率。圖 34 中紀錄了 N-gram 搜尋目標所對應的多個搜尋標的。為了增加回應速度，此處只儲存搜尋標的之 RK，而不儲存完整資訊。

因此，本搜尋引擎建立下列 7 張索引 table，以供搜尋之用。

1. Paper_Content_Index
2. Author_Name_Index
3. Paper_Title_Index
4. Publisher_Name_Index
5. Keyword_Title_Index
6. Journal_Title_Index
7. Abstract_Index

第三段　搜尋流程

本搜尋引擎之搜尋流程如圖 35 所示：

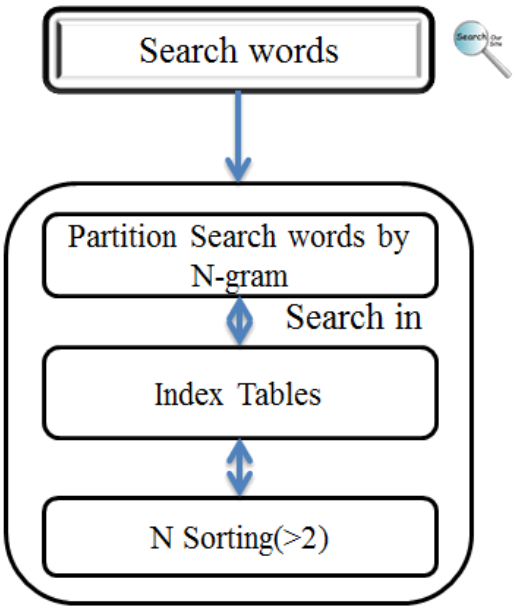
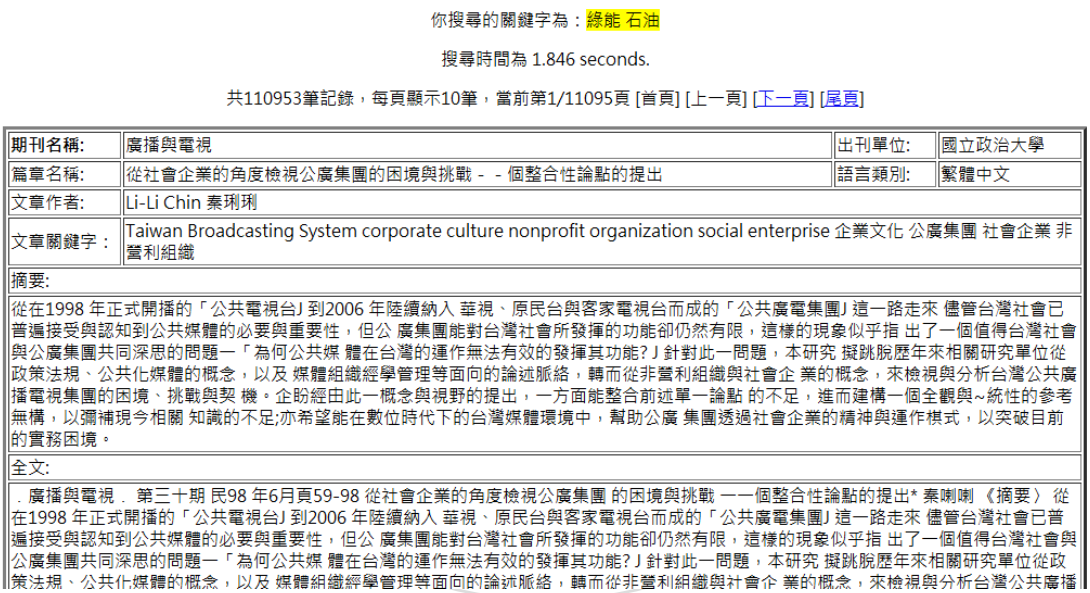


圖 35. 搜尋流程示意圖

本搜尋引擎會將搜尋文字進行 N-gram 斷詞，並將斷詞後所得到的 Term，列

入該搜尋目標之索引表，提供查詢。最後將搜尋結果以符合的字元數量(N)來進行排序，使用者可以依照需求指定符合之字數。搜尋畫面如下圖所示：

圖 36. 台灣學術期刊搜尋引擎搜尋畫面



你搜尋的關鍵字為：綠能 石油

搜尋時間為 1.846 seconds.

共110953筆記錄，每頁顯示10筆，當前第1/11095頁 [首頁] [上一頁] [下一頁] [尾頁]

期刊名稱:	廣播與電視	出刊單位:	國立政治大學
篇章名稱:	從社會企業的角度檢視公廣集團的困境與挑戰 - - 個整合性論點的提出	語言類別:	繁體中文
文章作者:	Li-Li Chin 秦琍琍		
文章關鍵字：	Taiwan Broadcasting System corporate culture nonprofit organization social enterprise 企業文化 公廣集團 社會企業 非營利組織		
摘要:			
從在1998 年正式開播的「公共電視台」到2006 年陸續納入 華視、原民台與客家電視台而成的「公共廣電集團」這一路走來 儘管台灣社會已普遍接受與認知到公共媒體的必要與重要性，但公 廣集團能對台灣社會所發揮的功能卻仍然有限，這樣的現象似乎指 出了一個值得台灣社會與公廣集團共同深思的問題—「為何公共媒 體在台灣的運作無法有效的發揮其功能?」針對此一問題，本研究 擬跳脫歷年來相關研究單位從政策法規、公共化媒體的概念，以及 媒體組織經學管理等面向的論述脈絡，轉而從非營利組織與社會企 業的概念，來檢視與分析台灣公共廣播電視集團的困境、挑戰與契 機。企盼經由此一概念與視野的提出，一方面能整合前述單一論點 的不足，進而建構一個全觀與~統性的參考無構，以彌補現今相關 知識的不足;亦希望能在數位時代下的台灣媒體環境中，幫助公廣 集團透過社會企業的精神與運作棋式，以突破目前的實務困境。		
全文:			
．廣播與電視 ． 第三十期 民98 年6月頁59-98 從社會企業的角度檢視公廣集團 的困境與挑戰 一一個整合性論點的提出* 秦喇喇 《摘要》 從在1998 年正式開播的「公共電視台」到2006 年陸續納入 華視、原民台與客家電視台而成的「公共廣電集團」這一路走來 儘管台灣社會已普遍接受與認知到公共媒體的必要與重要性，但公 廣集團能對台灣社會所發揮的功能卻仍然有限，這樣的現象似乎指 出了一個值得台灣社會與公廣集團共同深思的問題—「為何公共媒 體在台灣的運作無法有效的發揮其功能?」針對此一問題，本研究 擬跳脫歷年來相關研究單位從政策法規、公共化媒體的概念，以及 媒體組織經學管理等面向的論述脈絡，轉而從非營利組織與社會企 業的概念，來檢視與分析台灣公共廣播		

圖 37. 搜尋論文全文內容結果

由圖 37 可見，搜尋速度由 6 秒降低至 1-2 秒以內，可見雲端分散式搜尋引擎有效地提升搜尋的效能。

第三節 Web Service

除搜尋介面外，本搜尋引擎尚提供 Web Service 之功能，方便使用者進行學術期刊文章之增加及查詢。提供 Web Service 之服務，可讓使用者將搜尋服務介接至自行開發之系統，透過統一的 JSON 格式，使用者可以輕易地獲取搜尋的結果，也可以動態的新增文章。目前本搜尋引擎僅提供增加文章資訊之功能，此功能含輸入及輸出兩種格式。

第一段　輸入格式

```
{
    "ServiceType":Query|Upload
    "SearchGoal":Keyword|Paper_title|Publisher|Author_name|Periodical_name|Abstract|Paper_content
    "UploadType":Paper|Author|Publisher|Journal
    "UploadObject":
}
```

圖 38. Web Service JSON 輸入格式

使用者僅需提供下列 4 項參數，即可完成介接。

1. 服務型態(Service Type)

   分為 Query 及 Upload 兩種型態。如為 Query 則毋須提供上傳物件(Upload Object)參數。

2. 搜尋目標(Search Goal)

   使用者於此參數輸入搜尋所需之目標。

3. 上傳型態(Upload Type)

   使用者於此參數輸入上傳所需之型態。

4. 上傳物件(Upload Object)

   使用者輸入上傳物件之 Avro JSON 格式。

上述 Input JSON 中，使用者必須先於自身系統中，取得本搜尋引擎所定義之七種搜尋目標，及上傳型態之 Schema 定義。

第二段　輸出格式

```
{
    "ServiceType":Query|Upload
    "SearchGoal":Keyword|Paper_title|Publisher|Author_name|Periodical_name|Abstract|Paper_content
    "SearchResult":
    "UploadType":Paper|Author|Publisher|Journal
    "UploadStatus":Success|Failed
}
```

圖 39. Web Service 輸出格式

系統會回傳以下資訊，如圖 39 所示。

1. 服務型態(Service Type)
2. 搜尋目標(Search Goal)
3. 搜尋結果(Search Result)

   搜尋結果會以 「/」符號分隔各結果；搜尋結果為排序好之物件。
4. 上傳型態(Upload Type)
5. 上傳狀態(Upload Status)

   上傳狀態分為 Success 與 Failed。

第五章　結論與建議

資訊爆炸的挑戰，間接成就了雲端的發展，Hadoop 與基於 Hadoop 的 NoSQL 資料庫 HBase 也因而蓬勃發展。多數知名公司採用 HBase 以達到彈性、高速讀取寫入、隨機從硬碟讀取資料以及低延遲的優點。然從原先的關聯式資料庫轉換至欄導向資料庫的過程中，需要一套標準的轉換機制。故本研究嘗試提供關聯式資料庫轉換至欄導向資料庫(HBase)之轉換機制，並透過實作系統來印證其可行性。期望藉此研究刺激學術界對於欄導向資料庫的 ORM 及相關技術的發展。

速度及效能，是評斷搜尋引擎好壞的關鍵，即使是一毫秒的延遲也會造成使用者的不悅。遠流出版事業股份有限公司資訊長曾說過，其旗下之搜尋引擎對於 13 萬篇的學術期刊內文搜尋約需 6 秒的搜尋時間。即使建立索引其速度依然不甚理想。基於搜尋引擎對速度及效能的要求，本研究選擇了以 HBase 建構台灣學術期刊搜尋引擎以提升搜尋速度。

本研究於實作中發現 Hbase 官方文件中記載，Hbase 目前對於多 CF 的支援不甚理想，CF 數量能減則減；尤其是當 CF 間資料量差異過大時，會造成不必要的 I/O 負載。於實作中也發現，其寫入速度約差 10 倍之多。因此本研究另尋解決方式，透過序列化來轉換資料結構並定義其轉換規則。

本研究透過嘗試錯誤(Try & Error)的過程來實作並建置搜尋引擎，成功地使原本需要 6 秒的回應速度降低至 1 秒以內，減少使用者的等待時間。同時，建置 Web Service 及轉換工具使系統更具擴充性及跨平台性。轉換工具使用了 Map/Reduce 來降低大量轉換所需時間。

Hbase、Caasandra 等 NoSQL 資料庫適合拿來作為大量資料存取用之資料庫，使用者必須斟酌系統中那些資料區塊是小量且須經常性變動的，例如帳號登入、使用者基本資料修改等等，這類的資料存取就適合使用傳統關聯式資料庫，利用 SQL 來達成複雜條件的新增、刪除以及修改。本研究所提出之轉換機制適用於 ERM

已經有明確的制定下且資料不需經常性變動的系統，並且於商業邏輯上處理上述所提出之資料重複性以及資料一致性的問題。

本研究試圖以 ERM 轉換來貫穿整篇論文，並且成功的制定轉換機制以及完成實作；基於本研究之限制下仍有幾點需要討論，分述如下：

1. ERM 轉換機制是否真的需要存在，是否可以使用更簡單的方式來設計欄導向資料庫的結構？

   在 ERM 的正規化過程中常常會使得資料結構變得複雜、龐大、難懂，這時不用切分得太細，使用幾張簡單的 Table 來描述複雜且會使用到的資料結構，也許可以使得資料處理上更加簡單也更加有效率。在某些程度上，ERM 的轉換機制是需要存在的，如何去從現有的系統中移轉經過使用者考量需要轉換的區域、在甚麼樣的情況下需要存在，抑或可以使用更簡單的方式來實作是未來仍需努力的方向。

2. 如何去避免 ERM 設計上面原本的錯誤造成轉換後的誤差？

   透過本研究的轉換機制，可以使原本傳統關聯式資料庫的 ERM 模型成功轉換至欄導向資料結構以及 Avro 序列化結構。但是原先系統中的 ERM 與 table 之間的對應可能是錯誤的，在複雜、大型的資料庫系統中，更是常常發生。系統過於老舊造成文件的不周全、人為設計上的疏失等等，是造成 ERM 不正確的首要原因，這樣的錯誤會導致在使用本研究所提出之轉換機制上的困難點，轉換出來的資料庫 Table 有可能會是錯誤或者是不能夠被使用的。

基於以上兩點的未來探討與改進方向，本研究會以更有效率、穩定的轉換工具，更廣泛使用範圍、更嚴謹的轉換規則為目標，提供學術界及一般使用者作為參考，發展相關技術以及規範。

參考文獻

1. Gantz, J., & Reinsel, D. (2010). The Digital Universe Decade, Are You Ready?
2. Ghemawat, S., Gobioff, H., & Leung, S. T. (2003). The Google file system. ACM SIGOPS Operating Systems Review, 37, 29-43.
3. Greenberg, A., Hamilton, J., Maltz, D. A., & Patel, P. (2008). The cost of a cloud: research problems in data center networks. ACM SIGCOMM Computer Communication Review, 39(1), 68-73.
4. McLuckie, C. Google Compute Engine: Computing without limits Retrieved 16 July 2012, from http://googleenterprise.blogspot.tw/2012/06/google-compute-engine-computing-without.html
5. Apache wiki. Hbase/PoweredBy Retrieved 16 July, 2012, from http://wiki.apache.org/hadoop/Hbase/PoweredBy
6. NoSQL. (2012, July 10). In Wikipedia, The Free Encyclopedia. Retrieved 02:44, July 16, 2012, from http://en.wikipedia.org/w/index.php?title=NoSQL&oldid=501561398
7. Cloud computing. (2012, July 13). In Wikipedia, The Free Encyclopedia. Retrieved 02:39, July 16, 2012, from http://en.wikipedia.org/w/index.php?title=Cloud_computing&oldid=502099373
8. Mell, P., & Grance, T. (2011). The NIST Definition of Cloud Computing (Draft) Recommendations of the National Institute of Standards and Technology. NIST Special Publication, 145(6), 1-2.
9. White, T. (2011). Hadoop: The definitive guide, Second Edition: Yahoo Press.
10. The Apache HBase Book. Retrieved from http://hbase.apache.org/book.html