

國立清華大學資訊工程研究所

碩士論文

利用 MapReduce 實作分散式協同推薦系統

MapReduce Implementations of Distributed
Collaborative Based Recommendation System



學號： 9962637

姓名： 張雅芳

Ya-Fang Chang

指導教授： 李哲榮教授

Dr. Che-Rung Lee

中華民國 101 年七月

摘要

現在的資訊社會有許多應用都是仰賴大量資料而形成，Recommendation System 便是一例。Recommendation System 近年來大量應用在電子商務，主要用在推薦使用者可能會感興趣的物品，希望能藉此提高使用者對該網站有更多的探索，用處極廣，是資料處理相關領域相當重要的一個研究議題。隨著商業規模的擴大，Recommendation System 所要計算的資料大小也伴隨增加，計算所需花費的時間更是呈倍數成長。

Apache Mahout 實作了一個使用 MapReduce Framework 來進行 Large scale data-process 的 Recommendation System 運算，希望能夠藉由 MapReduce 讓運算更有效率並達到更高的精確度。MapReduce 是一個用在大量資料分散式運算的 programming model，能夠架設在 cluster 中，將資料分散到數台機器進行運算，用以進行大規模的資料處理。是 Google 先進行實作且大量運用在大規模資料的處理，後來 Apache Hadoop 跟進且將其 MapReduce Project 發展至更廣大的應用層面。

而我們針對上述 Mahout Distributed Item Based Recommendation System 進行分析後，針對該實作中花費較多執行時間的 Similarity Matrix 計算部分進行修改，改為進行 Stochastic SVD 和配合相關數學推導，實作出兩個 Distributed Collaborative-based Recommendation Systems。

本篇論文使用 Apache Hadoop 架設了兩組 Cluster，對 Mahout Distributed Item Based Recommendation System 和 Distributed SSVD Recommendation System 進行實驗測試，並且進一步比較其整體表現。

Abstract

Recommendation System has been widely used in electronic commerce recently. To promote user's visiting of websites, it recommends objects that users might be interested. Nowadays, large E-commerce sites often have millions of items and users, which increase the computation workload of Recommendation System rapidly.

Apache Mahout, an open source machine learning library, which uses MapReduce framework to implement Collaborative Filtering Recommendation Systems, is designed to make large-scale data process more efficient. MapReduce framework is a distributed computation programming model. It is first proposed by Google and applied to the development of many Google's services. Then Apache Hadoop developed its MapReduce Project which has more widespread applications. MapReduce is mainly used to do large-scale data processing by distributing data and computation to different nodes of Cluster.

In this thesis we present the work of analyzing the processing of a Mahout Distributed Item Based Recommendation System and improving the most time consuming part, which is the computation of similarity matrix. Two new algorithms of Distributed Collaborative-based Recommendation System are proposed and implemented using Stochastic SVD. Moreover, we conducted experiments to compare the performance and accuracy of those algorithms on two different clusters of Apache Hadoop servers. Experimental evaluations showed our algorithms and implementation can improve the performance of Mahout Distributed Item Based Recommendation System 2.5 times and its accuracy by Stochastic SVD features.

目錄

摘要.....	I
Abstract	II
目錄.....	III
圖片目錄.....	VI
表目錄.....	VIII
1. Introduction.....	1
1.1. Recommendation System.....	1
1.2. MapReduce	4
1.3. Apache Mahout.....	5
1.3.1. Mahout Recommendation System	5
1.4. Singular Value Decomposition.....	6
1.5. Contents of this Thesis	7
1.5.1. Collaborative Based Recommendation Systems	7
1.5.2. Contribution	7
1.5.3. Outline.....	8
2. Preliminaries.....	9
2.1. Terminology	9
2.2. Apache Hadoop and MapReduce.....	10
2.2.1. Apache Hadoop.....	10
2.2.2. Hadoop Distributed File System (HDFS)	11
2.2.3. MapReduce	11
2.3. Mahout Item Based Distributed Recommendation.....	13
2.3.1. Algorithm	14
2.4. SVD Recommender	16

2.5.	Evaluating Collaborative Filtering Recommendation System.....	17
3.	Algorithms	20
3.1.	Mahout Stochastic SVD Algorithm	20
3.1.1.	Low-rank Matrix Approximation.....	20
3.1.2.	Randomized SVD	20
3.1.3.	Mahout Stochastic SVD Algorithm	22
3.2.	Distributed SSVD Item-Based Recommendation.....	24
3.2.1.	Derivation.....	24
3.2.2.	Algorithm.....	24
3.3.	Distributed SSVD Recommendation	26
3.3.1.	Derivation.....	26
3.3.2.	Algorithm.....	27
3.4.	Complexity Comparison	28
4.	MapReduce Implementations	30
4.1.	Algorithm 1 : Mahout Item-Based Distributed Recommendation...31	
4.2.	Algorithm 2 : Distributed SSVD Item-Based Recommendation.....36	
4.3.	Algorithm 3 : Distributed SSVD Recommendation	38
4.4.	HDFS Reading and Writing	41
5.	Experiments and Results	43
5.1.	Environment.....	43
5.2.	Evaluation	44
5.2.1.	Data	44
5.2.2.	Quality.....	45
5.3.	Results	47
5.3.1.	Settings.....	47

5.3.2.	Performance	51
5.3.3.	Accuracy	56
5.3.4.	Evaluation	58
5.3.5.	Discussion	59
6.	Conclusion	63
	Future Work	63
	References	65



圖片目錄

圖 1-1：使用者與物品關係示意圖。	1
圖 1-2：由 user 和 item 關係組成之 User/Item matrix。	2
圖 1-3：User-based CF 示意圖。	3
圖 1-4：Item-based CF 示意圖。	4
圖 2-1：Hadoop Cluster 系統架構圖。	10
圖 2-2：MapReduce 簡易流程圖。	13
圖 2-3：Mahout Item-based Distributing Recommendation 示意圖。	15
圖 4-1：Mahot Item-Based Distributed Recommendation 於 Hadoop 實作 之流程圖。	31
圖 4-2：建立 User Vector 的 MapReduce 流程圖。	32
圖 4-3：Pairwise Similarity 示意圖。	33
圖 4-4：PartialMultiply 流程示意圖。	35
圖 4-5：Distributed SSVD Item Based Recommendation 於 Hadoop 實作之 流程圖。	36
圖 4-6：Distributed SSVD Recommendation 於 Hadoop 實作之流程圖。	38
圖 4-7：Mahout Stochastic SVD MapReduce 實作流程圖。	39
圖 4-8：Mapper 資料分佈示意圖。	40
圖 5-1：Evaluation 流程圖。	46
圖 5-2：Mapper,Reducer 個數與 node 個數比例關係圖。	48
圖 5-3：SSVD Singular values 趨勢圖。	49
圖 5-4：Distributed SSVD Recommendation 於不同 threshold 之 P/A ratio 比較圖。	50
圖 5-5：Algorithm1 和 Algorithm3 於 Delta 之執行時間。	51

圖 5-6：Algorithm1 和 Algorithm3 於 Delta 之 Speed up。	52
圖 5-7：Algorithm1 和 Algorithm3 (k=50) 於 Formosa 之執行時間。	53
圖 5-8：Algorithm1 和 Algorithm3 (k=10) 於 Formosa 之執行時間。	53
圖 5-9：Algorithm1 和 Algorithm3 (k=50) 於 Formosa 之 Speed up。	54
圖 5-10：Algorithm1 和 Algorithm3 (k=10) 於 Formosa 之 Speed up。	54
圖 5-11：Algorithm1 和 Algorithm3 (k=10) 整體之 Speed up。	55
圖 5-12：Algorithm1 和 Algorithm3 (k=50) 整體之 Speed up。	55
圖 5-13：Algorithm1 在不同 Cluster 上的 F1 Accuracy。	56
圖 5-14：Algorithm1 與 Algorithm3 (k=10) 的精準度比較圖。	57
圖 5-15：Algorithm1 與 Algorithm3 (k=50) 的精準度比較圖。	57
圖 5-16：Algorithm1 與 Algorithm3 (k=10) 的整體表現。	58
圖 5-17：Algorithm1 與 Algorithm3 (k=50) 的整體表現。	59
圖 5-18：Algorithm 1 在不同 Cluster 上的執行時間。	60
圖 5-19：Algorithm 3 在不同 Cluster 上的執行時間。	60
圖 5-20：Mahout Item-based Distributed Recommendation MapReduce 實作執行時間比例圖。	62
圖 5-21：Distributed SSVD Recommendation MapReduce 實作執行時間比例圖。	62

表目錄

表格 1：Mahout Recommender System 實作種類。	6
表格 2：Precision and Recall 物品分類表。	19
表格 3：各演算法於 MapReduce 實作方法比較表。	30
表格 4：各演算法實作於 HDFS 讀寫次數比較表。	42
表格 5：Environment 配置規格。	43



1.Introduction

1.1. Recommendation System

Recommendation System，或是 Recommendation Engine，屬於資料過濾系統（Information filtering system）的一環。資料過濾指的是在大量資料中擷取出所需要的資訊，指引出使用者所需之最相關資訊，一般是透過電腦來進行自動抽取、分類、摘要等工作；Recommendation System 是目前應用相當廣泛的一種資料過濾系統，目前大多用於電子商務（electronic commerce）的部分，透過自動化和客製化的行銷手法，根據每位客戶的需求推薦適當的產品。以亞馬遜網路書店（Amazon.com）為例，它們根據顧客的購物和瀏覽的行為，推薦該顧客可能感興趣的書籍及相關物品，透過這種方式讓它們成為全球數一數二著名的網路書店。

Recommendation System Model 主要是以“使用者”和“物品”這兩個物件的相關資訊所構成。每位使用者資訊可能包括性別、年齡、種族或是其他個人資料，物品資訊則可能是內容敘述、價錢、重量大小等，在這裡我們稱呼這些資訊為 metadata。除此之外，還會記錄使用者和物品之間的交易關係（transaction）或是互動情形（interaction）。例如，使用者 A 購買電影 B，使用者 X 評比產品 Y 為三等...諸如此類的行為。

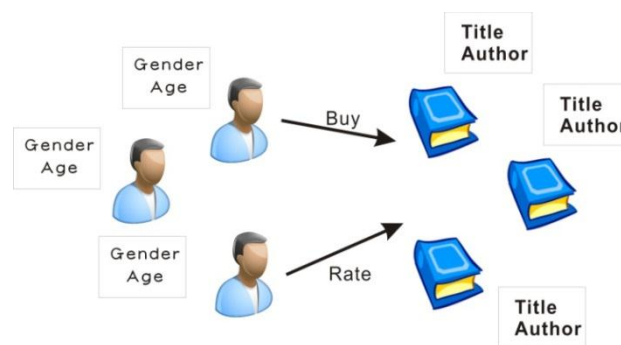


圖 1-1：使用者與物品關係示意圖。

在得到使用者和物品的 metadata 及相關 interaction 資訊後才能夠對其進一步分析，Recommendation System 的實作主要有 Content-based Approach 和 Collaborative Filtering Approach 這兩種方法。

Content-based Approach

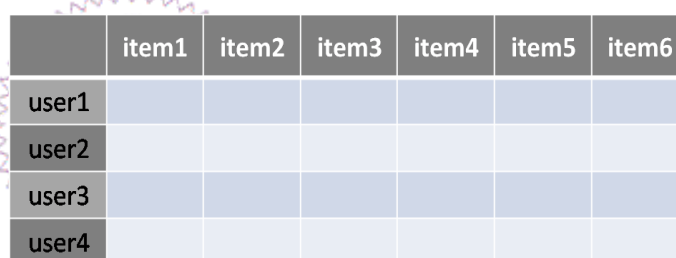
這個方法主要是對使用者和物品個別的 metadata 進行分類，找出和對應 metadata 最相近的項目再進行推薦。例如，使用者 A 購買了電影 B，則系統就會推薦使用者 A 購買一部和電影 B 非常相似的電影 C。

相似程度的計算是根據物品的 metadata 並使用不同的距離計算方法，其目的是要找出和電影 B 最相近的 k 個物品，這種方法稱作 k nearest neighbor(KNN)。利用這種找物品相似度的方法，只需要考慮到物品的 metadata，在實作時比較容易；但是當物品數量很大的時候計算 KNN 需要花費相當多的時間。

Collaborative Filtering Approach

不同於 Content-based Approach 是對 metadata 進行分類，Collaborative Filtering Approach 關注於使用者和物品的交易關係（transaction）或是互動情形（interaction），分析處理這些資料用以推薦適當的物品。

可以將使用者和物品的 interaction 資料表示成矩陣的形式，在本論文中稱該矩陣為 User/Item matrix。如圖 1-2 所示，在 User/Item matrix 中



	item1	item2	item3	item4	item5	item6
user1						
user2						
user3						
user4						

每一個 cell 代表對應 userX 和 itemY 的 interaction；其值可以是使用者給予物品的排名、也可以單純表示使用者和物品間是否曾有過互動或是互動的次數…等。

當 Recommendation System 的規模擴大，系統的使用者和物品數量將會非常龐大。像是亞馬遜網路書店（Amazon.com）中有數百萬種不同的商品，即便是頻繁使用該網站的使用者可能也只會跟其中 1% 的商品有 interactions。此時 User/Item matrix 就會是一個稀疏矩陣（Sparse matrix），表示大部分的 cells 內容將會是沒有意義或是沒有資訊的，而這些沒有意義 cells 會需要特別的方法來處理，在這裏就不多加描述。

使用 Collaborative Filtering Approach 實作 Recommendation System 的過程，就是對這些 Sparse User/Item matrix 進行一系列的矩陣運算，從中找出有用的資訊並進一步整理成使用者所需的訊息。

一般來說，Collaborative Filtering Approach 又會分成以使用者為主或是以物品為主的 model，以下會分別針對這兩種 model 和實作方法進行簡單的介紹。

1. USER-BASED COLLABORATIVE FILTERING

在以使用者為主的 model 下，要先找到和使用者 A 相似的群組，接著找出該群組成員所喜愛且使用者 A 尚未看過或購買的書籍，最後對這些書籍進行排序後再推薦給使用者 A；此方法的概念在於比對 User/Item matrix 中兩個 row vectors 的相似程度，找出前 k 個和使用者 A 相似的 user row，在從這 k 個 row 中找出適當的 item 推薦給使用者 A，如圖 1-3 所示。

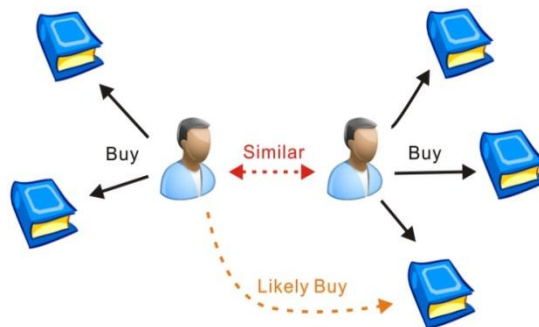


圖 1-3：User-based CF 示意圖。

在找相似 user row 時可以使用 Pearson similarity、Cosine similarity 等常見的 Similarity 計算方法，但隨著每位使用者 metadata 欄位和系統使用者的增加，計算時間也會隨之成長，User-based CF 將會遇到一些潛在的挑戰[7]：

- **Sparsity.** 大部分商業使用的 Recommendation System 通常都會有非常大量的 item sets，即便每位使用者經常和物品有交易或是互動，有關係的物品比例和沒關係的物品比例相較之下依然很低。因此會導致每個 user row 都非常稀疏，在進行一般 similarity 計算時可能會花費太多時間，或因此而遺漏部分資訊。
- **Scalability.** 當有新的使用者或是物品時，皆會影響 similarity 的計算。這對擁有數百萬使用者和物品的 Recommendation System 而言，將會是 scalability 上的重要議題。

目前已有提出改成以 hashing 為主的方法，配合適當演算法來加快整體運算的速度，在這裡並不多加討論。

2. ITEM-BASED COLLABORATIVE FILTERING

將使用者和物品的 interaction 視為關注對象的方法，除了上述以使用者為主的 model，還能以物品的角度來進行分析。首先從 interaction 資料中找出使用者 A 喜愛的書籍群組，接著找出和該書籍群組相似的物品，最後對這些物品進行排序後再推薦給使用者 A，如圖 1-4 所示。

改成以計算物品相似程度有幾項優點，首先物品 metadata 的欄位/種類一般會比使用者還要少，因此需要的計算量也會比較少；其次，不同於使用者的喜好可能會改變，物品的 metadata 較不易變動，一旦求出物品間相似程度後就比較不需要頻繁的更新。目前這個 model 的實作常見的是使用 Singular Value Decomposition (SVD) 來進行，利用 SVD 找出 User/Item matrix 的重要資訊，不僅能降低計算的維度，還能透過 SVD 的各種變形來加快整體的效率。

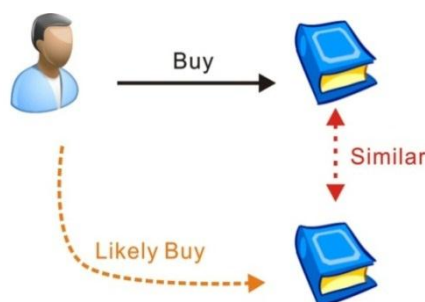


圖 1-4：Item-based CF 示意圖。

1.2. MapReduce

現代資訊社會中存在於網際網路的資訊相當龐大，許多機構或網站的資料儲存量以達到 petabyte (10^{15} byte) 的單位，若想對這些資料進行更進一步的處理則需要相對應的方法，一般稱這些議題為 Large-data problem。在西元 2004 年 Google 首先發展並將 MapReduce 應用於 Web search engine。後來 Apache Hadoop 參考 Google 的方法，也使用 MapReduce Programming Model 作為 Framework 中相當重要的一個部分。

MapReduce 是一種用於分散式資料平行處理的模型，將 Large-data problem 分散到 Cluster 內的各個 nodes，藉此進行大量資料的平行處理。

MapReduce 將處理程序分成兩個階段[2]：Map 和 Reduce 階段，分別的輸入

及輸出皆採用 key-value 的 pair。在 Hadoop MapReduce Framework 中，key 和 value 的型態可由 Programmer 根據 Data type 自行決定該使用的 Data structure，Programmer 也需要自行撰寫 Map 和 Reduce 函數來達成所需的工作。

1.3. Apache Mahout

Apache Mahout 為 Apache Hadoop 相關 project 之一，是個開放原始碼擁有可擴充性（scalable）的機器學習（ML，Machine Learning）資料庫，其中包含了許多機器學習和集體智慧（CI，Collective Intelligence）領域相關的演算法及實作，內容主要包括 Recommender Systems、Clustering、Classification 和 Frequent Pattern Mining 這四大種類。

Mahout 使用 MapReduce Framework 來幫助大量資料的處理，部分實作可直接用在 Apache Hadoop Cluster 來進行分散式計算。本篇論文即適用該架構來進行 Mahout Recommender System 的改進。

1.3.1. Mahout Recommendation System

根據先前所介紹過的 Recommendation System 的種類，Mahout 的實作主要是使用 Collaborative Filtering 的方法，根據不同的 model 和實作方法又分成多個種類。如表格 1，Non-distributed 類別主要是由一台機器，且使用非 MapReduce 的方式進行 Sequential 的實作；Distributed 類別的實作則是透過 MapReduce Framework，使用不同形式透過 Cluster 進行分散式運算。

如章節 1.1 Item-Based Collaborative Filtering 部分提到，以計算物品相似程度有計算量可能會比較少及不需要頻繁的更新等優點。儘管如此，在資料量越來越大的資訊時代，因為硬體的限制，一般 Non-distributed Recommender 已難以應付大量計算的需求。不同於 Mahout Non-distributed Recommender 每當要推薦給某位使用者時就必須重新多呼叫 function 且考慮其他 ratings，Mahout Distributed Recommender 能夠在一次計算後推薦大量使用者 Top-N recommendation。

本篇論文主要針對 Mahout 中 Distributed Recommender 的部分進行研究，接下來會以 Mahout Item Based Collaborative Filter Recommendation 為基礎，修改演

算法並對其效能改進。

	Non-distributed	Distributed
實作	UserBasedRecommender	Distributed Item-BasedCF
種類	ItemBasedRecommender	Pseudo-distributed Recommender
	SlopeOneRecommender	
	SVDRecommender	
	KnnItemBasedRecommender	
	TreeClusteringRecommender	

表格 1：Mahout Recommendation System 實作種類。

1.4. Singular Value Decomposition

Singular Value Decomposition(SVD)經常用在 Latent Semantic Indexing(LSI) 的 Information Retrieval(IR), 透過矩陣分解直接或間接找出該矩陣的各項特徵。

假設有個 $m \times n$ 的矩陣 A ，對 A 作 Singular Value Decomposition 則：

$$A = U \times \Sigma \times V^T \quad (1-1)$$

其中 U 和 V 分別是 $m \times m$ 和 $n \times n$ 的 Orthogonal 矩陣， Σ 則是 $m \times n$ 的對角矩陣。

Σ 對角線上的數值是 A 的 singular values，可令其為一個向量 \vec{S} ，若 A 的 rank 為 r ，則 S 的前 r 個 singular values $s_1 \geq s_2 \geq \dots \geq s_r > 0$ ，剩下的 $n - r$ 個數則為 0。 U 的 m 個 columns 是 A 的 left singular vectors，亦是 AA^T 的 eigenvectors； V 的 n 個 columns 是 A 的 right singular vectors，則是 $A^T A$ 的 eigenvectors。若我們只專注於前 r 個非零的 singular values，便只需要 U 、 V 矩陣的前 r 個 columns。這麼一來分別只要儲存 $m \times r$ 、 $r \times r$ 和 $n \times r$ 的 U 、 Σ 、 V 。

SVD 常用於解決 Low-rank matrix approximation 問題。若我們令整數 $k \ll r$ ，然後只取向量 \vec{S} 中的 Top k largest singular values 並令其為 Σ_k 對角線上的值，移除 U 後 $r - k$ 個 columns 令其為 U_k ，移除 V 後 $r - k$ 個 rows 令其為 V_k 。則我們可以得到新的矩陣 A_k ：

$$A_k = U_k \times \Sigma_k \times V_k^T \quad (1-2)$$

此時的 A_k 和 A 將會是 rank 為 k 時最相近（兩者間的 Frobenius norm 最小）的矩陣。

1.5. Contents of this Thesis

根據之前介紹 Recommendation system 的種類，在本篇論文只對 Item-Based Collaborative Filtering 進行討論與實作。如先前所述，實作過程就是對 Sparse User/Item matrix 進行一系列的矩陣運算，從中找出有用的資訊並進一步整理成使用者所需的訊息。但隨著商業規模的擴大，Recommendation System 所要計算的 User/Item matrix 也伴隨變大，矩陣運算所需花費的時間更是呈倍數成長。

因此 Apache Mahout 實作了一個使用 MapReduce Framework 以達到分散式運算的 Recommendation System 運算，希望能夠藉由 MapReduce 讓運算來處理這個 Large-data problem。

1.5.1. Collaborative Based Recommendation Systems

在經過對上述 Mahout Distributed Item Based Recommendation System 進行詳細的分析與研究後，以不改變演算法目標為前提，使用 SVD 方法的變化來幫助 Big Sparse User/Item matrix 的矩陣運算和重要資訊的擷取。本篇論文提出並使用 MapReduce 實作了兩個 Distributed Collaborative-based Recommendation systems，分別是 Distributed SSVD Item-based Recommendation 和 Distributed SSVD Recommendation。前者根據既有的 Mahout Distributed Item Based Recommendation System 進行部分修改，透過 Stochastic SVD 找出矩陣的重要資訊，配合公式推導取代部分運算；後者則是結合 Distributed SSVD Item-based Recommendation 和 Distributed SSVD Recommendation，為提升執行效率結合原先演算法的部分步驟，根據複雜度的分析結果將可以有效的改善整體執行的效率。將 Mahout Distributed Item Based Recommendation System 的演算法用 Stochastic SVD 找出的資訊做部分修改，以用來和前兩個方法進行比較。

1.5.2. Contribution

使用 MapReduce 實作兩個 Collaborative-based Recommendation Systems，並以

Distributed SSVD Recommendation 和 Mahout Distributed Item Based Recommendation System 進行效能和準確度的全面性評比。前者在演算法設計中減少有 Low-latency 性質的 HDFS 讀寫數次，從原先的六回減至三回；並省去實作中大部分繁複的資料轉換及運算。單就演算法部分的改進，Distributed SSVD Recommendation 在執行時間方面可以達到 1.5~1.8 倍的加速，若加上硬體環境方面的改善則可達到 2.2~2.5 倍。在準確度方面也有大幅的提高，在 Performance 和 Accuracy 整體表現更有明顯的優勢。

1.5.3.Outline

本篇論文的內容如下：第二章是 Preliminaries，首先在第一節定義本文中使用的名詞，第二節則是介紹與本文相關的背景知識。第三章是 Related Work，在第一節對 Mahout Distributed Item Based Recommendation System 當中所使用的演算法和 MapReduce 實作有仔細的分析，第二節則介紹 Collaborative Filtering Recommendation System 較常使用的 evaluating 方法。第四章是 Method and Implementation，第一節首先對 Stochastic SVD 的背景和推導做詳細的說明，然後在第二、第三節分別對 Distributed SSVD Item-based Recommendation 和 Distributed SSVD Recommendation 的演算法和 MapReduce 實作做詳細的說明。第五章是 Experiment and Result，當中包含了實驗環境設定、效能比較和實驗總結等。第六章是 Conclusion and Future Works。第七章是 Reference，列出本篇論文所參考的書籍、論文和網站資訊。

2. Preliminaries

本章先討論本篇論文所需的背景知識，共分兩大節。第一節對本論文使用的名詞給予明確定義，第二節說明與本文相關的基礎知識。

2.1. Terminology

Definition 1 : node

在 Hadoop 中 node 指的是在同一個 Cluster 中，每一台有架設 Apache Hadoop 環境的機器

Definition 2 : Recommendation

將所有使用者和物品的 metadata 整理後，經過 Recommendation System 的計算後，所列出對應使用者最可能感興趣的物品

Definition 3 : m

在本文中用來代表 Recommendation System 中的使用者個數

Definition 4 : n

在本文中用來代表 Recommendation System 中的物品個數

Definition 5 : key-value pair

是 MapReduce 中用來輸入、輸出資料的 Mapper 和 Reducer 的基本單位，由於 key 和 value 的型態不盡相同，在本文中將 value 的內容用底線標示

Definition 6 : User/Item matrix

在 Collaborative Filtering Approach 中以使用者和物品間 interaction 資料表示成矩陣的形式，大小為 $m \times n$

Definition 7 : User Vector

將 User/Item matrix 以 row 的形式儲存，每個 row_i 即是 $user_i$ 和所有物品間的 interaction 資料，User Vector 大小為 $1 \times n$ ， $i = 1, \dots, m$

Definition 8 : Rating matrix

User/Item matrix 的轉置矩陣，大小為 $n \times m$

Definition 9 : Similarity matrix

在本文中用來代表 item-item 間相似程度的矩陣，大小為 $n \times n$

Definition 10 : N

在本文中用來代表經過 Recommendation System 的計算後，每位使用者最後

被推薦的 recommendation 個數

Definition 11 : k

在本文中用來代表經過 Mahout Stochastic SVD 的分解後，會留下的 Singular value 個數

2.2. Apache Hadoop and MapReduce

2.2.1. Apache Hadoop

Hadoop[2]是 Apache 軟體基金會所研發的開放源碼平行運算編程工具和分散式檔案系統，主要緣起於 Nutch（開放原始碼的網路搜索引擎），其內部架構的概念來自於2003年與2004年Google發表的兩篇GFS和MapReduce相關論文。Apache Hadoop 目前已大量使用在商業用途，用於進行 Big Data 的處理，其中以 Yahoo!和 Amazon 的應用最有代表性。主要有下列三大 subprojects：

- Hadoop Common：提供分散式檔案系統和 I/O 控制存取（序列化、Java RPC 和實體資料結構）的元件及介面套件。
- Hadoop Distributed File System（HDFS）：可以在一般電腦所架設的大型從及環境下執行的一種分散式檔案系統。
- Hadoop MapReduce：可以在一般電腦所架設的大型從及環境下執行的一種分散式資料處理模型。

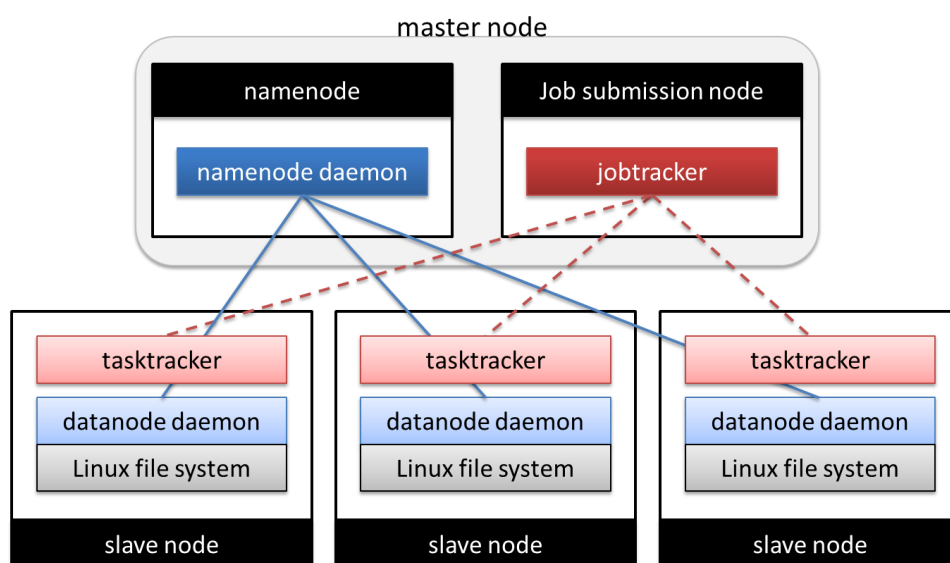


圖 2-1：Hadoop Cluster 系統架構圖。

除此之外還有許多 Hadoop 相關的 Projects，像是用來進行 RPC 和資料序列化處理的 Avro、使用 column-oriented database 的 HBase、或是輔助 HDFS 和 MapReduce 在 Cluster 環境下處理大量資料的高階語言 Pig...等，越來越多的相關專案被開發，提供 Hadoop 擴充性的服務或建立更高階層的核心。

Hadoop 為 Master-Slave 的 Cluster 型態，Master 除了可以和 Slave 一樣執行分配下來的工作，還必須有額外的程式來進行工作的分配與資料的調節。其大致上的系統架構如圖 2-1：Hadoop Cluster 系統架構圖。

2.2.2.Hadoop Distributed File System (HDFS)

Hadoop Distributed File System (HDFS) 是一個採用串流資料存取的檔案系統，設計用在儲存大型檔案，與現存的 Distributed File System 相類似。能透過 POSIX Like 的 Command line 指令直接存取，或是寫在程式中透過 API 進行 HDFS 的操作。

主要特色在於 HDFS 有極高的容錯能力，且適用於低階硬體設備 (Low-end server) 的 Cluster。然而 HDFS 卻有低延遲性 (Low-latency) 的資料存取，意指 HDFS 在存取資料時所需的時間較長，較適合少次大量資料的工作，而非多次少量資料的存取。

2.2.3.MapReduce

MapReduce 是一種用於分散式資料平行處理的模型，主要優勢在於大量資料處理。在[4]中提到，對於大量資料處理常被討論的 “Big ideas” 如下：

- **Scale out, not up.** 此原則表示在進行大量資料處理時，應選擇向外增加 Low-end servers 的個數 (Scale out)，而非提升少數 High-end servers 的硬體規格 (Scale up)。在這裡主要是以價錢 (price) 和效能 (performance) 作為考量，Barroso 和 Hölzle 在[11]中曾針對 Low-end servers 和 High-end servers 的 price/performance 進行比較，結果顯示 Scale “out” 確實明顯優於 Scale “up”。
- **Assume failures are common.** 當 cluster server 個數越多，整個 cluster

發生錯誤的機會也會隨之累加。為保持大量資料處理時的 Reliability，提供良好容錯機制是相當重要的議題。而 MapReduce programming model 能夠透過數個自動化機制的進行 failures 的處理，例如自動重啟不同 cluster node 上的工作。

- **Move process to data.** 在傳統 High Performance Computing (HPC) 常會將整體運算架構分成 *Processings node* 及 *Storage node*，然而在進行大量資料處理時一則不需要 High-end servers，二則可能因 Network bandwidth 問題產生 bottleneck；因此 MapReduce 架構將 *processing* 與 *storage* 放在同一 node 上，藉此提升效率。
- **Hide system-level details from the application developer.** 有別於一般平行分散式運算，MapReduce 提供抽象的使用方式，隱藏大部分 System-level 的細節。例如：程式中 Share memory 的分配、如何使用 Barriers 進行各個 Process 的同步化、或是避免 Deadlock 或 Race Condition 等問題。因此，對大部分的 Programmer 來說，在發展 MapReduce Job 的時候，只需要專注於需要被執行的運算，而不用考慮進行運算的流程和過程中資料的分配。

如先前所述，MapReduce 將處理程序分成兩個階段[2]：Map 和 Reduce 階段，分別的輸入及輸出皆採用 Key-Value 的 pair。

一個 MapReduce 作業 (Job) 是客戶端要執行的工作單位，包含輸入資料、MapReduce 程式和相關組態。通常 Hadoop 在執行一個 Job 時會將它分成數個任務 (Task)，這些 tasks 可分成 map task 或是 reduce task。如圖 2-1 中所標示，每個 MapReduce Job 會有一個 *jobtracker* 和數個 *tasktracker* 來控制工作執行流程。由 master node 控制 *jobtracker*，透過 *jobtracker* 對各 slaves 中的 *tasktracker* 進行任務排程，並協調系統上的所有執行工作。

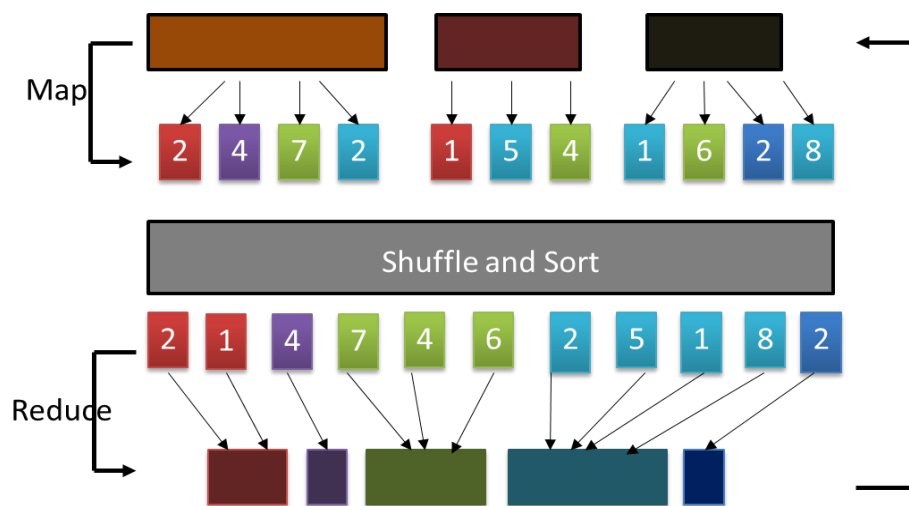


圖 2-2：MapReduce 簡易流程圖。

在開始執行 Job 前會先將存在 HDFS 中的資料分散到不同 nodes，在不同 node 中分別進行 MapReduce Tasks。如圖 2-2 所示，圖中每個小 blocks 用來表示一個 key-value pair，顏色表示 key，block 中的數值表示 value；一般來說，Hadoop 會先將輸入資料分割成特定大小後再交給 MapReduce Job，由 Map 階段將對應的 Input split 內容轉化成 Pairs 後，中間經過 Framework 對資料的 key 值進行 Shuffle 或是 Sort 的整理，再在 Reduce 階段根據相同的 key 對 values 進行處理後將結果整合輸出。

2.3. Mahout Item Based Distributed Recommendation

目前 Mahout 實作了一個完整的 Distributed Recommendation System — Item-Based Distributed Recommendation。主要是參考 B. Sarwar 等人的 Item-Based Collaborative Filtering Recommendation Algorithms [7]，該演算法在對資料進行 preprocessing 後，首先利用常見的 Correlation-Based 方法進行 User/Item matrix 中 Item 間的 Similarity 計算；然後再進行 Prediction Computation，利用求出的 Item Similarity，使用 Weighted Sum 或是 Regression 的方式分別求算每位使用者可能感興趣的物品清單。

2.3.1.Algorithm

如先前所說，此 Item-Based 演算法最主要的兩個步驟是 *Item Similarity Computation* 和 *Weighted Sum Prediction*，但為了要從未整理過的 Input ratings 開始做起，過程中還要經過多次 data 的處理和計算。以下為參考 *Mahout In Action Chapter 6*[1]的內容，大致整理該 Algorithm 使用的方法如下：

首先假設有 m 位 user， n 個 item。

Step 1 Preprocess 使用者和物品間互動情形，建構出一個 $m \times n$ 的 User/Item matrix，User/Item matrix 中的每個 cell 代表使用者對物品的評價；為了接下來的 Similarity matrix 計算，真正儲存在 HDFS 的矩陣是 User/Item matrix 的轉置矩陣，在這裡將其命名為 Rating matrix R ： $n \times m$ 。

Step 2 求算 Similarity matrix，利用之前提到的 Correlation-based 方法，對 Rating matrix 進行 *Item Similarity Computation*，最後求出 $n \times n$ 的 Similarity matrix，cells 紀錄 item-item 間的 similarity，可在圖 2-3 中看到是個對稱矩陣。

Step 3 建立 user vector，在這裡的 user vector _{i} 即是 User/Item matrix 的 row _{i} 。把每位使用者當作 n 維空間中的一個向量 \vec{u} ，而該使用者對每個 item i 的喜好程度對應到 u_i 的值， $i = 1, \dots, n$ 。如圖 2-3 中的 User 3 向量，若 User 3 和 item 沒有互動則將其值設為零。

Step 4 藉由 *Weighted Sum Prediction* 產生 Recommendations，將上述兩個步驟產生的 Similarity matrix 和 user vector 相乘，產生 $n \times 1$ 的 result vector \vec{r} 。除去原先就在 user vector 中有對應值的 items，如圖 2-3 中編號 101、104 和 105 的 items，對 result vector 剩下的 items 對應值由大到小排序，找出前 N 個 items 推薦給該使用者。

	item similarity matrix							user vector	result vector
	101	102	103	104	105	106	107	User 3	Result
101	5	3	4	4	2	2	1	2.0	40.0
102	3	3	3	2	1	1	0	0.0	18.5
103	4	3	4	3	1	2	0	0.0	24.5
104	4	2	3	4	2	2	1	4.0	40.0
105	2	1	1	2	2	1	1	4.5	26.0
106	2	1	2	2	1	2	0	0.0	16.5
107	1	0	0	1	1	0	1	5.0	15.5

圖 2-3：Mahout Item-based Distributing Recommendation 示意圖。

上述的四個步驟為進行 Collaborative Filtering Item-Based Recommendation 的方法，可將其進一步整理成 **Algorithm 1：Mahout Item-Based Distributed Recommendation**，在演算法中的四個部分和上述的步驟大致相同，主要是根據實作的演算法流程呈現。

Mahout Item-Based Distributed Recommendation Algorithm 的計算時間主要在 2. Do Item Similarity Computation. 的部分，根據[7][15]中對 Pairwise Similarity 的描述，這部分的計算的時間複雜度需要 $O(mn^2)$ ，在物品眾多的情形下將需要花

Algorithm 1：Mahout Item-Based Distributed Recommendation

Given an input ratings file and the ratings involve m users and n items. And a user file, which contains the recommended users list. $i = 1, \dots, m$.

1. Preprocess ratings file, create User/Item matrix and its transpose matrix Rating matrix R . Storing User/Item matrix as user vectors for later use, $R \in \mathbb{R}^{n \times m}$.
2. Do *Item Similarity Computation*. Compute Item Similarity matrix S by Pairwise Similarity, $R \in \mathbb{R}^{n \times n}$.
3. Map corresponding item rows of S to user vector $_i$ if the itemID is found inside user vector $_i$.
4. Do *Weighted Sum Prediction*. Multiply the corresponding item rows with user vector $_i$ respectively, and use the summation as result vector $_i$. Select Top-N as recommendations for user i .

費大量的執行時間；1 是進行 input data 的處理，時間根據 input ratings file 而定；3 的部分是將 Similarity matrix 的 row 分散到各個使用者，時間複雜度為 $O(n)$ ；4. Do Weighted Sum Prediction 的部分則根據每個使用者給予 ratings 的物品個數而定，一般來說此數會遠小於 n ，因此在這裡我們可以直接將這個部分的時間複雜度視為 $O(m)$ 。

2.4. SVD Recommender

如先前所言，現在是個資訊發達、資料量暴增的時代。當我們使用 Collaborative Filtering Approach 預測且推薦物品給使用者時，首先遇到的問題是如何處理龐大且稀疏 User/Item matrix。近來 Latent Semantic Indexing (LSI) 領域常使用的 SVD 也開始被大量運用在 Collaborative based Recommendation System。

D. Billsus 和 M. J. Pazzani[8]透過 SVD 將 Collaborative Filtering 的過程轉化成 classification 的問題。他們提到使用一般 correlation-based prediction 方法可能會有的幾項限制：

- Correlation 只能針對每兩位使用者間共同有給予評價的物品進行計算。然而當資料庫中物品數量相當多時，大部分的物品將不會被納入分析，導致計算的結果有失公平性。
- 每位使用者對於評分的标准相當難以界定，或許使用者 A 和使用者 B 同時對物品 X 有興趣，卻因個人因素分別給予高分和低分。如此一來，使用 correlation approach 將可能把這項訊息視為沒有意義
- 若使用者 A 和使用者 B 擁有相似的品味或興趣，卻因為物品數量太多而沒有互相交集的 ratings，那麼最後推薦給使用者 A 和使用者 B 的物品可能會大有不同。

基於以上幾點將很有可能影響推薦的結果。所以 D. Billsus 和 M. J. Pazzani 認為藉由使用 SVD 將 User/Item matrix 中的重要訊息進行分類，可以有效減少類似的 transitive similarity relation 資訊流失。

M. G. Vozalis 和 K. G. Margaritis[9]則是使用 SVD 和 demographic data 的變化，進行一系列 Item-based 及 User-based Collaborative Filtering 實驗。最後他們認為

將 SVD 應用於 Recommendation System 不僅能夠解決 Recommendation System 最常面對的 *scalability* 和 *sparsity* 問題，還能夠有效提升其準確度。

後來 S. Gong 等人[10]，也實作了結合 SVD 和 Item based 的 Collaborative Filtering，並且對於實作的演算法及實驗結果有詳細的說明，主要是將 SVD 用在降低 User/Item matrix 的維度。首先將 $m \times n$ 的矩陣 A 進行 SVD 分解：

$$SVD(A) = U \times \Sigma \times V^T$$

U 、 V 分別是 orthogonal 的 $m \times m$ 和 $n \times n$ 矩陣， Σ 則是 $m \times n$ 的對角矩陣。透過對這些矩陣的 dimension reduction，求得和原矩陣 A 有相近線性關係的 A_k ，其中 U_k 、 V_k 和 Σ_k 降低維度成 $m \times k$ 、 $n \times k$ 和 $k \times k$ ，且 $k \ll n$ ：

$$A_k = U_k \times \Sigma_k \times V_k^T$$

然後再使用 U_k 、 V_k 和 Σ_k 這些 reducing matrix，在低維度的條件下進行 Item-based CF，以達到比較有效率的 Prediction 的過程。

在 Mahout 中也有實作一個 Non-distributed SVD-based Recommender，他們並未直接實作 SVD，而是使用 *Alternating Least Squares* 或是 *Expectation Maximization* 來達到 SVD 的效果。根據[1]中提到，實驗結果有相當好的準確度，然而 SVD 的矩陣分解卻需要相當大的計算量。

2.5. Evaluating Collaborative Filtering

Recommendation System

一般來說，在進行 Recommendation System 的評估時，會根據使用者的目的及使用的 recommender 來分類，使用適當的 dataset 進行測試；並且假設是在使用者可使用或是購買所有物品並對其進行喜好排序的情況。主要在於評估 Recommendation System 給予某使用者的物品喜好排名預測和該使用者實際喜好排名的差異，以及對某物品排名分數預測的精確度。

Recommendation System 精確度的分析問題已經相當多相關的研究，根據 Herlocker[5]等人的討論，可以將 recommendations 精確度度量種類分成 Predictive Accuracy Metrics、Classification Accuracy Metrics 和 Rank Accuracy Metrics。

(1) Predictive Accuracy Metrics

該種類方法主要是在分析 Recommendation System 的預測排名結果和實際使用者排名結果有多接近，在須將預測排名結果呈給使用者的情形下，此方法將會格外重要。

最常見的測量方法是 *Mean Absolute Error and Related Metrics*，用在計算預測和實際結果的平均差(2-1)， p_i 為預測結果， r_i 為實驗結果， N 是所有物品的個數。

$$|\overline{E}| = \frac{\sum_{i=1}^N |p_i - r_i|}{N} \quad (2-1)$$

Mean absolute error 不適合用在只需要少數 top n items 和物品分級不夠精確的情況，例如只有 1（喜歡）、0（不喜歡）兩種選項的情形就不適用。雖然此方法計算量相當大，但計算簡單且利於進行兩個不同系統之間結果的比較。

(2) Classification Accuracy Metrics

Classification metrics 量測 recommender 正確判斷物品的好壞的頻率，物品的好壞是根據使用者的喜好來做決定，不同於 Predictive Accuracy Metrics，在這裡傾向於物品分級只有 1（喜歡）、0（不喜歡）兩種選項。Classification Accuracy Metrics 並不直接測量演算法排名預測的能力，而是從預測的結果正確性來判斷該演算法的優劣。

Precision and Recall 即是用在評估 Information Retrieval System 時相當受歡迎的一種方法，Precision and Recall 首先將排名的成績分成 binary scale，再將結果進行分類：將物品根據之前分好的 binary scale 歸類到 relevant 和 not relevant，再分別判斷該類物品是否有被 recommender 推薦，有被推薦的分到 selected、反則分到 not selected，最後將會分成四個類別，如表格 2 所示。

	Selected	Not Selected	Total
Relevant	N_{rs}	N_{rn}	N_r
Not Relevant	N_{is}	N_{in}	N_i
Total	N_s	N_n	N

表格 2：Precision and Recall 物品分類表。

Precision 定義為 relevant 且 selected 的物品數目除以所有 selected 物品的數目，代表 selected 物品是 relevant 的機率(2-2)；*Recall* 則是 relevant 且 selected 的物品數目除以所有 relevant 物品的數目代表 relevant 物品是 selected 的機率(2-3)。若將 recommend 出來的結果和真實的 data set 比較，當 Precision and Recall 的值越接近，表示所求結果越精確。

$$P = \frac{N_{rs}}{N_s} \quad (2-2)$$

$$R = \frac{N_{rs}}{N_r} \quad (2-3)$$

(3) Rank Accuracy Metrics

這個方法是在量測 Recommendation System Algorithm 產生物品的 recommended ordering 正確性。和前兩個方法有所不同，Rank Accuracy Metrics 較適合用在須將所有 ranking 結果展現給使用者的情形，並不適用在推薦單一物品的 recommender 評估。和此類方法相關的有 *half-life utility metric* 和 *NDPM metric*。

3. Algorithms

3.1. Mahout Stochastic SVD Algorithm

3.1.1. Low-rank Matrix Approximation

Low-rank matrix approximation 是指將特定的 Given matrix A ，在限制的 rank 下轉化成較小的 Approximating matrix A' ，常見的方法有 QR 分解、eigenvalue decomposition 和 SVD。在進行 low-rank matrix approximation 時，一般可將其步驟分成兩個階段[12]：

Stage A. 找出 Given matrix A 的 approximate basis，意思是我們需要一個矩陣 Q ，其中 Q 有 orthonormal columns 並且滿足 $A' \approx QQ^T A$ 。

Stage B. 利用 Stage A 找出的 Q 來幫助 A 的分解，例如：QR、SVD。

當中 Stage A 的步驟能夠藉由 random sampling 的方法快速完成，亦是 3.1.2 將會討論的 Randomized SVD。

3.1.2. Randomized SVD

Randomized SVD 是指利用 randomized matrix 進行 approximation，將原先龐大的矩陣分解成 approximate SVD，亦是屬於 Low-rank matrix approximation 的一種方法。Halko 等人[12]提出了 **Prototype 1 : Prototype for Randomized SVD**，和上一小節提到 Low-rank matrix approximation 的步驟一樣分成兩階段進行，Stage A 利用 Gaussian test matrix Ω 求出 Orthonormal matrix Q ，Stage B 在進一步計算 Randomized SVD。



Prototype 1 : Prototype for Randomized SVD

Given an $m \times n$ matrix A , a target number k of singular vectors, and an exponent q (say $q = 1$ or $q = 2$), this procedure computes an approximate $\text{rank-}2k$ factorization $U\Sigma V^T$, where U and V are orthonormal, and Σ is nonnegative and diagonal.

Stage A.

1. Generate an $n \times 2k$ Gaussian test matrix Ω .
2. Form $Y = (AA^T)^q A\Omega$ by multiplying alternately with A and A^T .
3. Construct a matrix Q whose columns form an orthonormal basis for the range of Y .

Stage B.

4. Form $B = Q^T A$.
5. Compute an SVD of the small matrix: $B = \tilde{U}\Sigma V^T$.
6. Set $U = Q\tilde{U}$.

當 A 是個稀疏矩陣時，實作時多會將矩陣轉成向量的型態以便資料儲存和計算。傳統 Low-rank matrix approximation 常使用 Krylov subspace method 和 SVD 結合進行計算，雖然用 Krylov method 通常能有較高的準確度，但若和 randomized 的方法比較，後者有兩個明顯的優勢。第一，因為 randomized 本身的性質，可以擁有較好的穩定性而不必侷限於特定性質的 A 矩陣；第二，在實作矩陣和向量乘法時一如 **Prototype 1** 中 Stage A. 的 $A\Omega$ 相乘，能夠被平行化。

Oversampling

一般在做 Low-rank matrix approximation 時，不可避免會失去 Given matrix A : $m \times n$ 中的部分資料。但留下的會是 **Top-k** singular values (用 \bar{S} 表示)，意指去掉的資料大多為較不重要的部分，因此最後求出的 approximate matrix A' : $m \times k$ 將會和 A 相當接近。

然而在使用 Randomized matrix 進行 approximation 時，所找出來的 **Top-k** singular values (用 \bar{S}' 表示) 可能會失去原來 \bar{S} 中部分的重要 elements，進而導致 A' 和 A 間的距離被拉遠[14]。為了彌補因這些 elements 所造成的誤差，通常會設

一個用來做額外 sampling 的參數 p ，找出新的 approximate matrix $A'': m \times (k + p)$ 。通常只需要極小的 p 值（如： $p = 10$ ）就能夠提升足夠的精確度。

3.1.3. Mahout Stochastic SVD Algorithm

Mahout Stochastic SVD Algorithm 和 **Prototype 1 : Prototype for Randomized SVD** 相當類似，但為了實作成 MapReduce Framework 而將演算法進行了部分修改。**Algorithm: Modified SSVD Algorithm** 為 Mahout 工作團隊 release 的 working note[13]中所詳述的演算法。該演算法架構大致沿襲 **Prototype 1**，步驟 1.~3.為 Stage A—找出 matrix Y 的 approximate basis Q ，步驟 4.~8.是 Stage B—利用 Stage A 找出的 Q 幫助矩陣分解，並根據需要求算 U 、 V 。

Algorithm : Modified SSVD Algorithm

Given an $m \times n$ matrix A , a target rank k , and an oversampling parameter p , this procedure computes an $m \times (k + p)$ SVD, $A = U\Sigma V^T$.

Stage A.

1. Create seed for random $n \times (k + p)$ matrix Ω . The seed defines matrix Ω using Gaussian unit vectors per one of suggestions in [12].
2. $Y = A\Omega, Y \in \mathbb{R}^{m \times (k+p)}$.
3. Column-Orthonormalize $Y \rightarrow Q$ by computing thin decomposition $Y = QR$. Also, $Q \in \mathbb{R}^{m \times (k+p)}$, $R \in \mathbb{R}^{(k+p) \times (k+p)}$.

Stage B.

4. $B = Q^T A, B \in \mathbb{R}^{(k+p) \times n}$.
5. Compute Eigensolution of a small Hermitian $BB^T = \hat{U}\Lambda\hat{U}$. $BB^T \in \mathbb{R}^{(k+p) \times (k+p)}$.
6. Singular values $\Sigma = \Lambda^{0.5}$, or, in other words, $s_i = \sqrt{\sigma_i}$.
7. If needed, compute $U = Q\hat{U}$.
8. If needed, compute $V = B^T \hat{U} \Sigma^{-1}$, or $V = A^T U \Sigma^{-1}$.

Stage A 主要的計算在 3.的 QR decomposition，將 Y 矩陣分成數個 blocks 後使用 Givens Rotation 的方法求出 $Y = QR$ 。過程中經過數次 Mapper 和 Reducer Tasks 的執行，QR Job 主要執行時間取決於 input parameters $(k + p)$ 的大小和 Y

矩陣的 block 個數 z ，根據 Halko[14] 的分析其複雜度約為 $O(z \times (k + p)^3)$ ；Stage B 的部分主要是使用 Q 求出矩陣 $B = Q^T A$ ，有必要的話再進一步由 BB^T 找出 A 的 U 、 Σ 、 V 。以上兩個步驟是在同一個 Job 計算，其複雜度約 $O((k + p)^3)$ 。最後可得到 $A \approx \text{approximate } A' = U \times \Sigma \times V^T$ 。

和一般 SVD 或是 Low-rank approximation 方法比較，Mahout SSVD 有下列幾項優點[16]：

- **Parallelization**：和 Krylov subspace 的方法比較，和 Randomized 方法在計算複雜度上是相同的，但如先前所言 **Prototype 1** 中 Stage A 的 $A\Omega$ 相乘，能夠輕易的被平行化。當實作在 Cluster 時，只需要分別給予每個 Node random seed，便能夠平行進行 A 中 row_i 和 Ω 的相乘。
- **Fixed Iterations**：不論 SSVD 中的所需要的 rank k 值設為多少，所需要做的 Mapper 和 Reducer 次數是固定的，不用擔心過多 iteration 次數影響執行時間。
- **Precision/Speed Balance**：在 SSVD 中主要影響執行速度的參數是 rank k 、oversampling p 和代表 QR Job 中 Y 矩陣 block 個數的 z ，當這三個參數值越大時，所花的計算時間便越長，其中又以 k 、 p 的影響尤甚；同時 k 、 p 也影響 $m \times n$ 矩陣 A 和 approximate A' 間的相似程度，當 k 、 p 越大則 A 和 approximate A' 間的距離會越小。因此在使用 SSVD 時可根據當下的需求而進行 k 、 p 和 z 的適當調整。
- **Special Data Structure**：使用 Mahout Library 中 *DistributedRowMatrix* 這個資料結構進行矩陣的儲存與運算，讓矩陣中沒有對應值的 cell 能夠不給予紀錄。由於在 Big Dataset 時的矩陣大部分皆會非常稀疏，能因此省下不少儲存空間。
- **PCA**：從 Mahout 0.7 版開始將可以直接用 SSVD 求算出 PCA，但這並非本篇論文的研究重點，因此在此不多做討論。

然而 SSVD 的執行確實會減少矩陣的部分資訊，因此和一般 SVD 比較時不可避免會有精確度較差的缺點。

3.2. Distributed SSVD Item-Based Recommendation

Distributed SSVD Item-Based Recommendation 以 Mahout Item Based Distributed Recommendation 結合 Mahout Stochastic SVD Algorithm。在這裡沿襲了前者的程式架構—分別求算 *Item Similarity Computation* 和 *Weighted Sum Prediction*，但在 Similarity 求算的部分改用 SSVD 找出來的 U_R 、 Σ_R 、 V_R 直接計算 approximate similarity $RR^T = U\Sigma^2U^T$ ，這部分的推導公式在下面的 (3-2)。

3.2.1.Derivation

首先假設有 m 位 user， n 個 item，User/Item matrix A 為 $m \times n$ ，Rating matrix $R = A^T$ 為 $n \times m$ 。如之前提到，計算 Rating matrix R 的 Item Similarity matrix 便是在求算 RR^T ： $n \times n$ ，且 Orthogonal 矩陣 U 和 V 滿足

$$\begin{aligned} UU^T &= U^TU = I_m \\ VV^T &= V^TV = I_n \end{aligned} \quad (3-1)$$

所以我們可以進行以下推導：

$$\begin{aligned} R &= U\Sigma V^T, R^T = V\Sigma U^T \\ RR^T &= (U\Sigma V^T)(V\Sigma U^T) \\ &= U\Sigma^2U^T \\ &= \text{similarity matrix} \end{aligned} \quad (3-2)$$

3.2.2.Algorithm

Distributed SSVD Item-Based Recommendation 共分成四個步驟，大部分都和前兩個演算法的步驟重複，首先假設有 m 位 user， n 個 item。

Step 1 Preprocess 使用者和物品間互動情形，建構出一個 $m \times n$ 的 User/Item matrix A 和 $n \times m$ 的 Rating matrix $R = A^T$ ，這個部分跟 Algorithm 1 的 Step 1 是相同的。

Step 2 對 Rating matrix 進行 SSVD 分解，也是直接利用 Mahout

Stochastic SVD 求出

$$R \approx U_R \times \Sigma_R \times V_R^T$$

當中的 U_R 、 Σ_R 、 V_R 分別為 $n \times k$ 、 $k \times k$ 和 $m \times k$ ，在這裡求算 SSVD

過程中的參數 k 、 p 能夠根據需求而自行調整。

Step 3 使用 Step 2 的 U_R 、 Σ_R 、 V_R 計算 approximate similarity RR^T 步驟，建立 user vector，在這裡的 user vector_{*i*} 即是 User/Item matrix 的 row_{*i*}。

Step 4 產生 Recommendations，將上述步驟產生的 similarity matrix 和 user vector 相乘產生 $cn \times 1$ 的 result vector \vec{r} ，並選出 Top-N recommendations。

上述的四個步驟可將其進一步整理成 **Algorithm 2 : Mahout Item-Based SSVD Distributed Recommendation**，在演算法中的四個部分和上述的步驟大致相同，主要是根據實作的演算法呈現。

Algorithm 2 主要沿襲 **Algorithm 1** 的架構，但不做 RowSimilarityJob，改用公式 (3-2) 以 SSVD 的方法計算 Item Similarity matrix。計算時間主要在 2. Do Stochastic SVD. 的部分，如先前說提到，計算所花費的時間根據 k 、 p 、 z 而定，

Algorithm 2 : Mahout Item-Based SSVD Distributed Recommendation

Given an input ratings file, user file, a target rank k , and an oversampling parameter p , $i = 1, \dots, m$.

1. Preprocess ratings file, create User/Item matrix and its transpose matrix Rating matrix R . Storing User/Item matrix as user vectors for later use, $R \in \mathbb{R}^{n \times m}$.
2. Do Stochastic SVD. Use R as input, generating U_R 、 Σ_R 、 V_R , where $U_R \in \mathbb{R}^{n \times k}$, $\Sigma_R \in \mathbb{R}^{n \times n}$, $V_R \in \mathbb{R}^{m \times k}$.
3. Compute Similarity matrix S by RR^T . Then map corresponding item rows of S to user vector_{*i*} if the itemID is found inside user vector_{*i*}.
4. Do *Weighted Sum Prediction*. Multiply the corresponding item rows with user vector_{*i*} respectively, and use the summation as result vector_{*i*}. Select Top-N as recommendations for user i .

時間複雜度約為 $O(z \times (k + p)^3)$ ；1 是進行 input data 的處理，時間根據 input ratings file 而定；3 的部分在做 $U_R \Sigma_R^2 U_R^T$ 計算要 $O(n \times k + k \times k + k \times n) = O(k(2n + k)) = O(nk)$ ，且 $k \ll n$ ；4. Do Weighted Sum Prediction 的部分和 **Algorithm 1** 一樣，時間複雜度視為 $O(m)$ 。

3.3. Distributed SSVD Recommendation

在 2.3Mahout Item Based Distributed Recommendation 提到該 Algorithm 的主要工作在於找出 Rating matrix 的 Item Similarity matrix，接著用此矩陣進行 *Weighted Sum Prediction* 求出 user file 中對應使用者的 Top-N recommendations。在 Distributed SSVD Recommendation 中結合了上述的兩個步驟，使用 SSVD 求算 Rating matrix R 的 approximate U、 Σ 、V，再利用 Orthogonal matrix 的特性省略不必要的矩陣相乘和 HDFS 讀寫。且為了減少過多的篩選步驟，選擇同時求出所有 user 的 Top-N recommendations，達到更好的執行效率和輸出結果。以下為了敘述方便，直接稱 Mahout Item Based Distributed Recommendation 為 **Algorithm 1**。

3.3.1.Derivation

首先假設有 m 位 user，n 個 item，User/Item matrix A 為 $m \times n$ ，Rating matrix $R = A^T$ 為 $n \times m$ ，根據 **Algorithm 2** 的 Derivation 可得：

$$R = U \Sigma V^T, R^T = V \Sigma U^T$$

$$R R^T = U \Sigma^2 U^T = \text{similarity matrix}$$

又 **Algorithm 1** 中用於 *Weighted Sum Prediction* 的 User Vectors 皆是 Rating matrix R 的 Columns，所以我們能夠將 m 個 User Vectors 結合成一個 User matrix $M_u : n \times m$ ，且 $M_u = R = A^T$ ；m 個 Result Vectors 結合成一個 Result matrix $M_r : n \times m$ 。如此一來，可以將 **Algorithm 1** 中 m 回合的矩陣／向量相乘：

$$\text{similarity matrix} \times \text{user vector}_i = \text{result vector}_i, i = 1 \dots m.$$

結合成一回矩陣／矩陣相乘，並且利用上面的推導進行化簡：

$$\text{similarity matrix} \times M_u = M_r$$

$$\begin{aligned} \xRightarrow{\text{轉置}} (\text{similarity matrix} \times M_u)^T &= M_u^T \times \text{similarity matrix} \\ &= R^T \times R R^T \\ &= (V \Sigma U^T) \times (U \Sigma^2 U^T) \\ &= V \Sigma^3 U^T \\ &= M_r^T \end{aligned} \quad (3-3)$$

因為 Result matrix M_r 的 Column $_i$ 為 Result Vector $_i$ ，為了實作上的便利將其轉置，經過轉置後 M_r^T 的 Result Vector $_i$ 只是換成是 row $_i$ ，對結果並沒有影響；同時為了減少資料的儲存，只留下最後 M_r^T 的 Top-N recommendations。

3.3.2. Algorithm

經過上面的推導，重新設計一個 Distributed SSVD Recommendation，並將演算法分成三個步驟：首先假設有 m 位 user， n 個 item。

Step 1 Preprocess 使用者和物品間互動情形，建構出一個 $m \times n$ 的 User/Item matrix A 和 $n \times m$ 的 Rating matrix $R = A^T$ ，這個部分跟 Mahout Item Based Distributed Recommendation 的 Step 1 是相同的。

Step 2 對 Rating matrix 進行 SSVD 分解，利用 Mahout Stochastic SVD 求出

$$R \approx U_R \times \Sigma_R \times V_R^T \quad (3-4)$$

當中的 U_R 、 Σ_R 、 V_R 分別為 $n \times k$ 、 $k \times k$ 和 $m \times k$ ，在這裡求算 SSVD 過程中的參數 k 、 p 能夠根據需求而自行調整。

Step 3 利用 *Item Similarity Computation* 和 *Weighted Sum Prediction* 化簡後的公式 (3-3)，直接計算出所有 users 對應的 Result Matrix M_r^T ，最後只留下 Top-N recommendations。

上述的三個步驟可將其進一步整理成 **Algorithm 3：Distributed SSVD Recommendation**，在演算法中的三個部分和上述的步驟大致相同，主要是根據實作的演算法呈現。

Algorithm 3 : Distributed SSVD Recommendation

Given an input ratings file, a user file, a target rank k , and an oversampling parameter p .

1. Preprocess ratings file, create User/Item matrix and its transpose matrix Rating matrix R . Storing User/Item matrix as user vectors for later use, $R \in \mathbb{R}^{n \times m}$.
2. Compute Stochastic SVD. Use R as input, generating U_R, Σ_R, V_R , where $U_R \in \mathbb{R}^{n \times k}, \Sigma_R \in \mathbb{R}^{n \times n}, V_R \in \mathbb{R}^{m \times k}$.
3. Do *Item Similarity matrix* and *Weighted Sum Prediction* by single multiplication function. Compute Result matrix $M_r^T = V_R \Sigma_R^3 U_R^T$, and select Top-N recommendations for all users.

Algorithm 3 的計算時間主要在 2. Compute Stochastic SVD. 的部分，根據 3.1 介紹的 Mahout Stochastic SVD Algorithm，計算所花費的時間根據 k (rank)、 p (oversampling)、 z (block numbers) 這三個參數而定，這部分的計算的時間複雜度如之前所提到約為 $O(z \times (k + p)^3)$ ；1 是進行 input data 的處理，時間根據 input ratings file 而定；3 的部分在進行 $V \Sigma^3 U^T$ 的矩陣相乘，然後直接輸出每位使用者的 Top-N recommendations。根據上述的實作過程，在這部分只需要 $O(m \times k + k \times k + k \times n) = O(k(m + n + k))$ 的執行， $k \ll n$ 且 $k \ll m$ 。

3.4. Complexity Comparison

將 Algorithm1 相比，Algorithm2 雖然少了 $O(mn^2)$ 的 Pairwise Similarity 計算，而改用 SSVD 的方法求算 approximate similarity RR^T ，來找出大量資料中的重要資訊。但仍然需要經過四個步驟才能完成，且又比 Algorithm1 多做了一回合的乘法，因此較難以判斷其結果的優劣程度。所以為了確實讓整體的計算複雜度降低，又提出了也用 Stochastic SVD. 進行 Recommendation System 實作的 **Algorithm 3 : Distributed SSVD Recommendation**。

Algorithm 3 也沒有使用 $O(mn^2)$ 的 Pairwise Similarity 計算 similarity matrix 而改用 SSVD 的方法，和 Algorithm 2 不同的是直接將 Algorithm1 的四個步驟使用

數學推導的方式結合，計算複雜度上雖然還是三次方，但因為 $k \ll n$ 且 $k \ll m$ 讓執行複雜度有明顯降低且有可調整的彈性。雖然用 SSVD 會使 R 失去部份的資訊，但因為 SVD 的性質且我們主要重視的是 Top-N 的 recommendations，所以可將這步驟視為 *Smoothing*，去掉 noise 後反而能夠使最後的 prediction 結果更好。



4.MapReduce Implementations

在上一章對 Recommendation 演算法做了詳細說明，可以看出大部分的計算在於矩陣的轉換和相乘，而有關矩陣處理平行化的議題已經被大量討論。在本章將會介紹如何用 MapReduce 實作 Distributed Recommendation System，加以 Mahout Library 的輔助，以有效率的完成 Recommendation System。

大部分的 Recommendation System 計算過程都很繁雜，在本篇論文中介紹的每個演算法都需要做三到四個步驟，詳細內容可見表格 3。各 Recommendation System 皆是由 RecommenderJob 這個 main program 開始執行。在這裡將可以直接從 Command line 被執行，且有獨立輸入輸出結果的步驟稱為某 Job，如：PreparePreferenceJob；直接在 RecommenderJob 中設定並執行 Mapper 或 Reducer Task 的步驟，則直接命名不加 Job 於字尾，如：AggregateAndRecommend。

Algorithm 1	Algorithm 2	Algorithm 3
Mahout Item-Based Distributed Recommendation	Mahout Item-Based SSVD Distributed Recommendation	Distributed SSVD Recommendation
PreparePreferenceJob	PreparePreferenceJob	PreparePreferenceJob
RowSimilarityJob	StochasticSVDJob	StochasticSVDJob
PartialMultiply	PartialMultiply2	MultiplyAndRecommend
AggregateAndRecommend	AggregateAndRecommend	

表格 3：各演算法於 MapReduce 實作方法比較表。

在 Mahout Library 中定義了許多幫助 MapReduce 程式執行的 Data Structures，在大部分的 Mahout implementation 中都大量使用這些 Data Structures 來幫助程式的撰寫和執行，而此 Recommendation System 也是如此。在這裡 Recommendation System 的資料傳遞大多都使用<Int,Vector>或是<Long,Vector>的 key-value pair 來儲存矩陣的 row 或 column，有時候則會為了讓 MapReduce 程式更有效率，而另外使用較為特殊的 Data Structures，像是 *RandomAccessSparseVector* 或 *DistributedRowMatrix* 這兩個資料結構進行儲存與運算，讓矩陣中沒有對應值的

cell 能夠不給予紀錄。由於在 Big data set 時的矩陣大部分會非常稀疏，能因此省下不少儲存空間。

以下會分別對每個 Recommendation System Algorithm 的 MapReduce 實作進行各步驟的詳細說明，當 Algorithms 有相同的步驟時，將不會再重複討論。

4.1. Algorithm 1 : Mahout Item-Based Distributed Recommendation

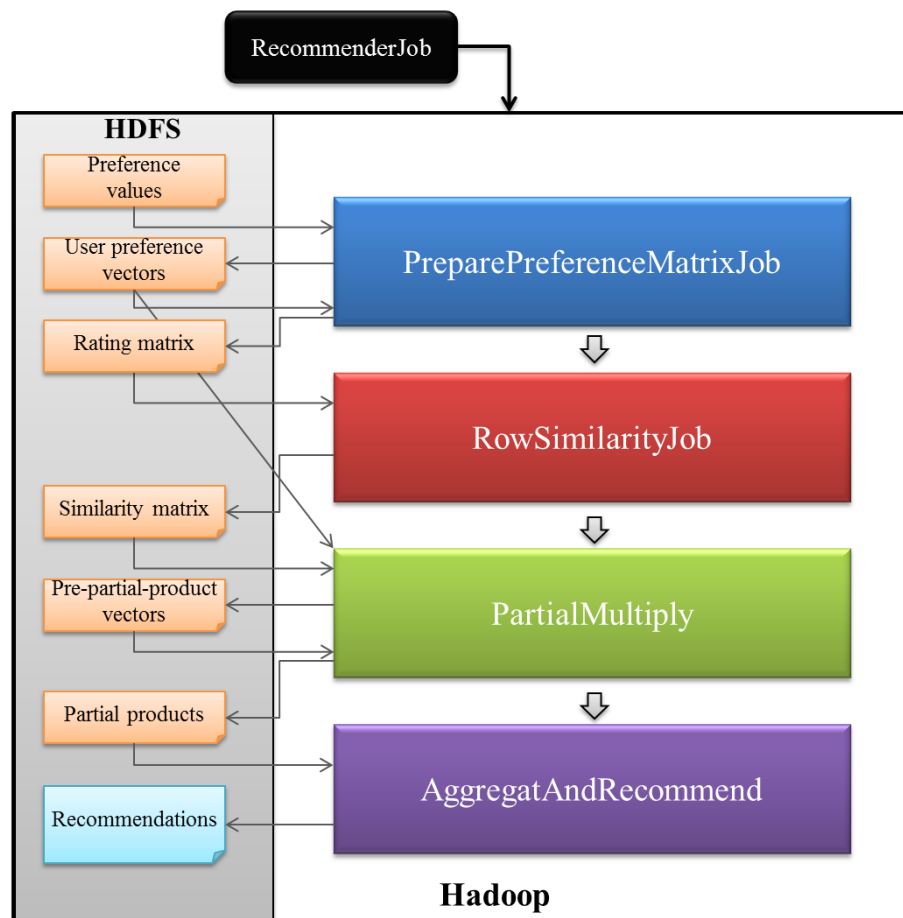


圖 4-1：Mahot Item-Based Distributed Recommendation 於 Hadoop 實作之流程圖。

在進行 MapReduce 實作時，將此 RecommenderJob 分成四個部分，如圖 4-1 的流程圖（參考 *Mahout In Action Chapter 6*[1]）。右半部的四個長方框分別表示四個主要的步驟，每個步驟裡可能又包括了數個 Mapper Reducer Tasks；左邊灰底部分代表 HDFS 的操作，橘色方框為儲存在 HDFS 裡的資料，細箭頭表示資料的讀寫，粗箭頭代表程式的執行流程。各步驟的詳細實作說明如下：

PreparePreferenceMatrixJob

此 Job 的主要目的在處理大量的 input ratings，產生 User Vectors 和 Rating matrix。在 input ratings file 裡的每一行都是 *userID,itemID,prefValue* 格式的 rating，*userID* 和 *itemID* 分別對應到使用者和物品的編號，*prefValue* 則是使用者對該物品的評分，數值從 1~5，越喜歡的物品 *prefValue* 數值越高。在這部分的實作包含了三回合的 MapReduce Process 和 HDFS 讀寫：

- (1) *itemIDIndex*：建立 input ratings 中所有 items 的對應 Index，以利於後面 ID 對照的工作。
- (2) *toUserVectors*：將 input ratings 從一行一行的 ratings，合併成以 *m* 個 *userID* 為 *key*，和以 user rated items list 為 *value* 的 user vectors，並將這些 vectors 以 row vector 的形式儲存在 HDFS，可視為一個 $m \times n$ 的 User matrix。過程中利用 MapReduce 進行大量 input ratings 的處理，圖 4-2 為 MapReduce 流程示意圖，最左邊的 pairs 代表一開始 ratings，*key* 為 input rating file 的行數，*value* 為對應的 rating；經過 Mapper 進行文件的分割和初步處理，得到 $\langle \text{userID}, \text{itemID}:\text{prefValue} \rangle$ 。然後經過 Framework 的簡單排序，再進一步由 Reducer 結合、產生 $\langle \text{userID}, \text{user rated items list} \rangle$ pair 的 user vectors（也是 User/Item matrix 的 rows）。

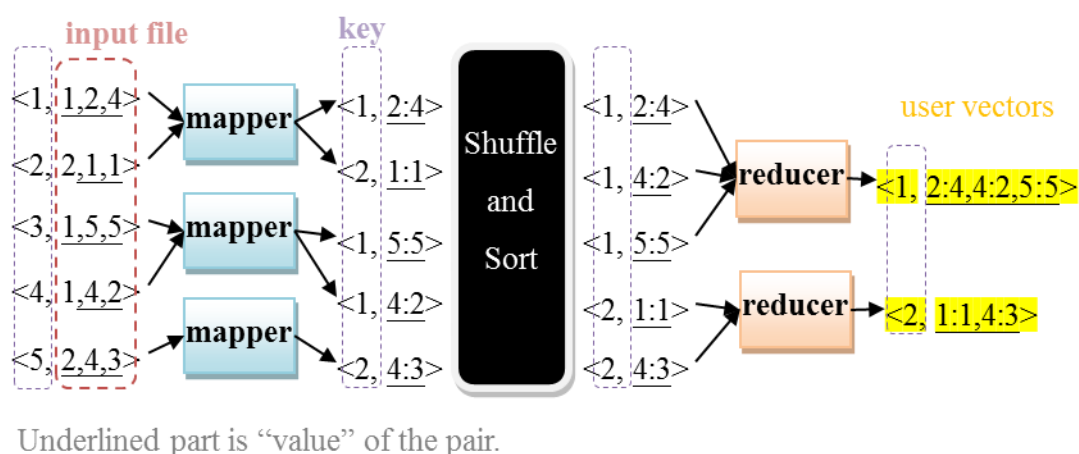


圖 4-2：建立 User Vector 的 MapReduce 流程圖。

- (3) *toItemVectors*：為了建立 $n \times m$ 的 Rating matrix，將上一個步驟得到的 $m \times n$ User/Item matrix 轉置。Rating matrix 的 rows 是 $\langle \text{itemID}, \text{related user list} \rangle$ 的 pairs，related user list 是指所有曾給該 item 評分的 user 和

分數。在 input ratings 數量相當龐大時，不論是 Rating matrix 或是 User/Item matrix 都會是稀疏矩陣。

RowSimilarityJob

這部分直接使用 Mahout Math Library 中的 RowSimilarityJob，用來進行 *Item Similarity Computation*，目的在求算 Rating matrix R 的 Pairwise Similarity RR^T ，在 library 中提供了數種求算 Pairwise Similarity 方法，但在這裡先只對其流程進行介紹，在這當中亦需要 3 回 MapReduce Process 的計算和 HDFS 讀寫[7][18]：

- (1) *normsAndTranspose*：對 Rating matrix 的每個 rows 做 normalization，然後將其轉置，將其 columns 存到 HDFS，以便接下來 pairwiseSimilarity 的計算。
- (2) *pairwiseSimilarity*：主要在這部分進行 Pairwise Similarity 的計算，根據演算法必須對 i 和 j columns 兩兩比較，最基本的比較法是直接將 columns 中對應 element w_{t,Col_i} 或 w_{t,Col_j} 相乘後的數值相加，作為 item i 和 j 的 similarity $sim(Col_i, Col_j)$ ， $i = 1, \dots, n$ 且 $j = 1, \dots, n$ ，複雜度為 $O(n^2)$ 。

$$sim(Col_i, Col_j) = \sum_{t=1}^m w_{t,Col_i} \cdot w_{t,Col_j} \quad (4-1)$$

如圖 4-3 所示，實作時主要是對 Rating Matrix 轉置後的 columns 兩兩比較，只有兩個 column 都有數值才進一步處理。

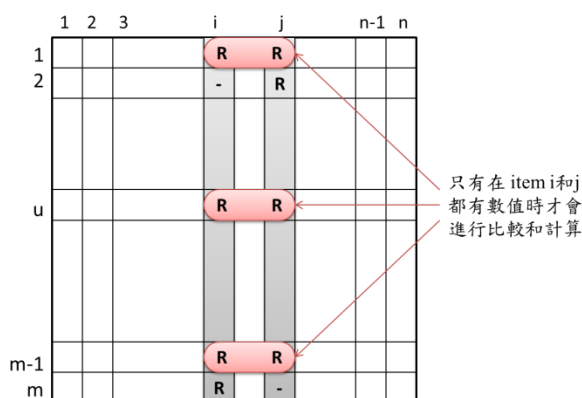


圖 4-3：Pairwise Similarity 示意圖。

實作時先在 Mapper 進行兩兩 columns 對應演算法的比較和計算，然後

在 Reducer 將同 Column 的結果加起來並用根據對應 Similarity 演算法求算 Similarity。

- (3) *asMatrix*: 根據初始設定, 每個 row 只保留 Top-k 個 Similarity 後才將結果寫到 HDFS, Similarity matrix 為 $n \times n$, 但每個 row 中只存有最多 k 個 elements。

PartialMultiply

這部分是為了更有效率的執行進行下個步驟的 MapReduce 才做的 Preprocessing, 結合前兩個 Jobs 寫到 HDFS 的 data, 並且配合 User File (內含有想被推薦物品的使用者 ID 的 input file) 的 userID, 整合至特別定義的 Data Structure 來為下一個部分的 *Prediction* 做準備。

在這裡分別做了兩回合 Mapper 和一回合 Reducer, 但仍然有三次 HDFS 讀寫, 可對照圖 4-4: PartialMultiply 流程示意圖。左上方的 map1 表示 *prePartialMultiply1*, 左下方的 map2 表示 *prePartialMultiply2*, 右邊結合前兩者 pairs 的 reduce 則是 *partialMultiply*:

- (1) *prePartialMultiply1*: 只進行 Mapper Task, 將 Similarity matrix 的 $\langle itemID, similarity\ vector \rangle$ 中的 value 轉換至另一個可記錄 similarity vector 或是 userID:prefValue list 的 Data Structure **VectorOrPref**, 並將結果暫存於 HDFS。
- (2) *prePartialMultiply2*: 同樣只進行 Mapper Task, 在 userID 有出現在 User File 的情形下, 從已儲存在 HDFS 中選出對應此 userID 的 User Vector, 再從這個 User Vector 的 value, 也就是 itemID:prefValue list 中, 選出 Preference value 值較高的 t 個 items, 也用 VectorOrPref 儲存成 t 個 $\langle itemID, userID:prefValue \rangle$, 將結果暫存於 HDFS。
- (3) *partialMultiply*: 實作 Reducer Task, 將已依照 itemID 排序過、從 HDFS 取得來自 (1)(2) 的 pairs 根據相同的 itemID 進行合併變成 $\langle itemID, similarity\ vector\ \&\ [userID:prefValue\ list] \rangle$ 的 pair, 這裡的 value 以 **VectorAndPref** Data Structure 儲存和傳遞, 待下個階段使用。

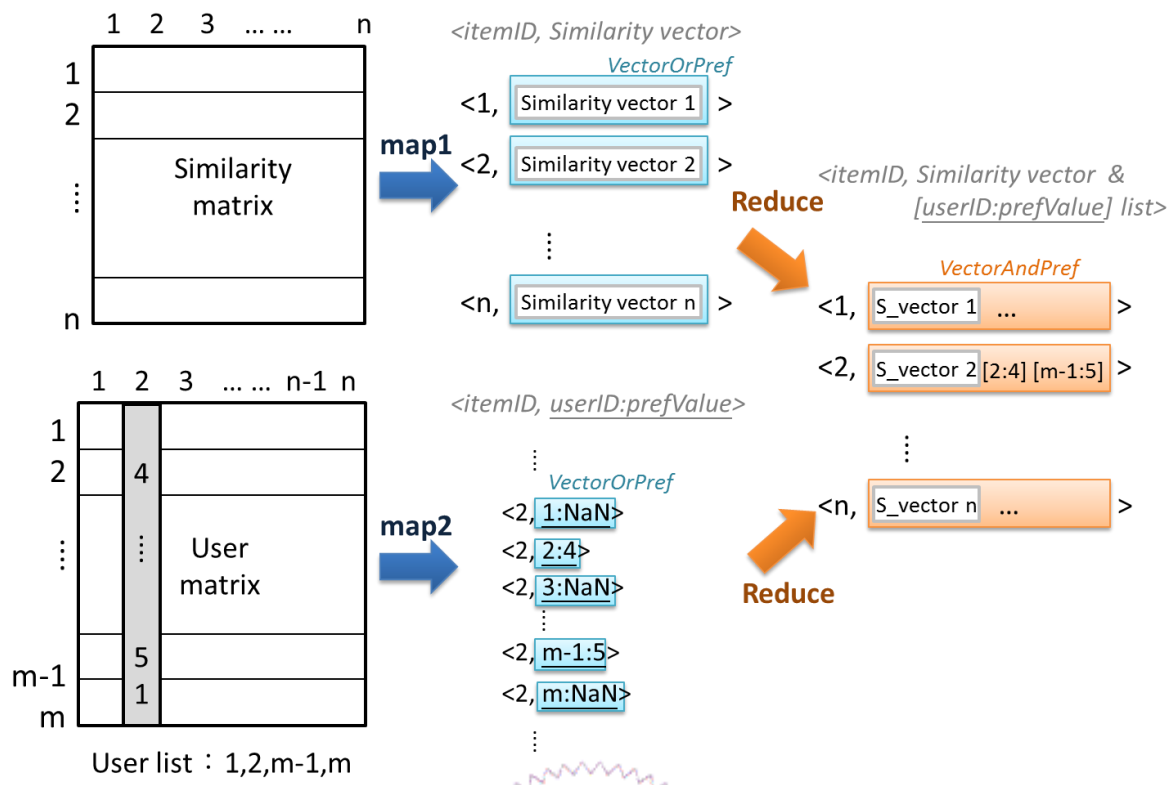


圖 4-4：PartialMultiply 流程示意圖。

AggregateAndRecommend

最後是計算 *Weighted Sum Prediction*，然後找出 Top-N 的結果並根據使用者輸出推薦的 N 個 recommendations。這部分只做了一回合的 MapReduce Process，並在最後將結果寫到 HDFS。

- (1) Mapper：將之前的 $\langle \text{itemID}, \text{VectorAndPref} \rangle$ pair 重新組合 key 和 value 的內容，根據每個 userID:prefValue 中的 userID 分配 item similarity vector 和 prefValue，改成以 userID 為 key 的 pair $\langle \text{userID}, \text{similarity vector \& prefValue} \rangle$ 。
- (2) Reducer：首先計算 weighted sum—把每個 pair 的 similarity vector 乘上 prefValue 倍，再將同一 userID 的 weighted similarity vector 加起來。然後經過 Scaling 取出 Top-N elements 寫到 HDFS，就是該 userID 使用者的 Top-N recommendations。

從上述的 Algorithm 1 實作介紹中可看出，使用 MapReduce 實作

Recommendation System 的過程相當繁複，除了基本演算法的設計外，常會需要數次 Mapper 和 Reducer 間 key-value 的 pairs 轉換與計算，而且還需要藉由多次讀寫 HDFS 來達到資料的傳遞。而 HDFS 的讀寫有低延遲性（Low-latency）的資料存取，過多的資料讀寫可能會造成 Recommendation System 的 Bottleneck。

4.2. Algorithm 2 : Distributed SSVD Item-Based Recommendation

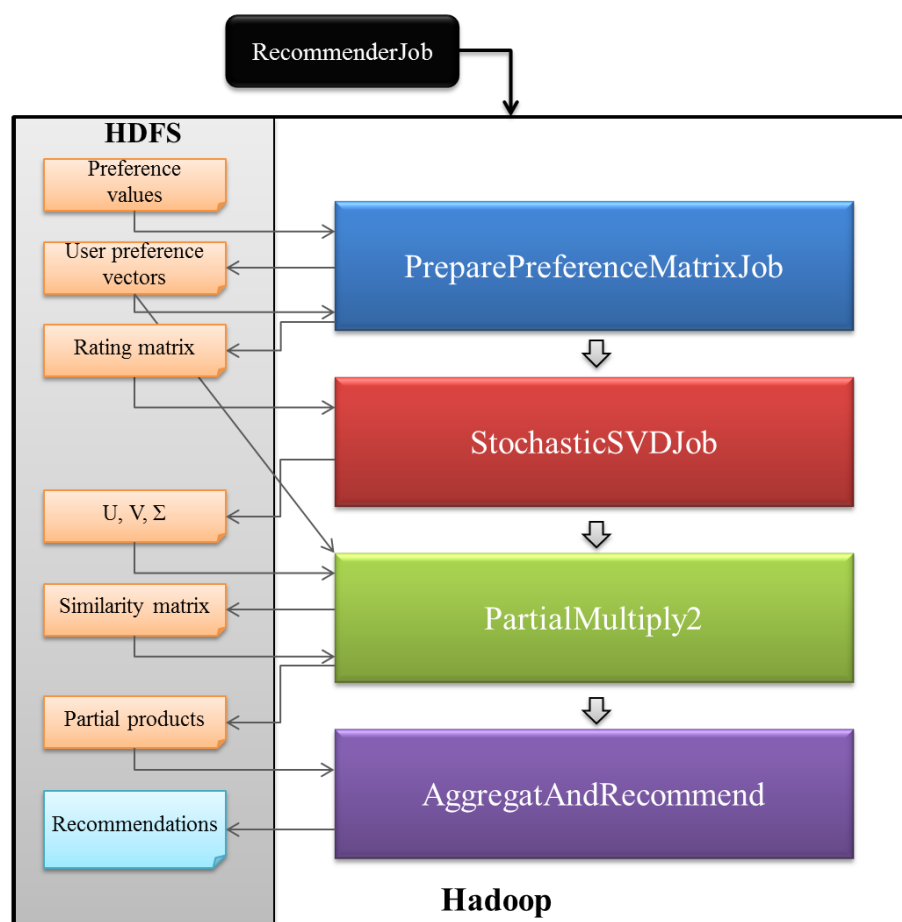


圖 4-5：Distributed SSVD Item Based Recommendation 於 Hadoop 實作之流程圖。

此節在進行 **Algorithm 2 : Mahout Item-Based SSVD Distributed Recommendation** 實作的介紹，將此 RecommenderJob 分成四個部分，如圖 4-5 的流程圖。右半部的四個長方框分別表示四個主要的步驟，每個步驟裡可能又包括了數個 Mapper Reducer Tasks；左邊灰底部分代表 HDFS 的操作，橘色方框為儲存在 HDFS 裡的資料，細箭頭表示資料的讀寫，粗箭頭代表程式的執行流程。

該演算法的是從 Algorithm 1 和 SSVD 結合而成，因此實作和 **Algorithm 1** 的實作也是大致相同。從表格 3 可以看出，Algorithm 2 的 **PreparePreferenceMatrixJob** 和 **AggregateAndRecommend** 部分和 Algorithm 1 的實作完全相同，**StochasticSVDJob** 的部分則和 Algorithm 3 相同；唯有 **PartialMultiply2** 的部分和 Algorithm 1 的 **PartialMultiply** 略有不同。

PartialMultiply2

和 **PartialMultiply** 唯一的不同在於做 *prePartialMultiply1* 的 Mapper Task 時，同時算出 similarity matrix。因為 Algorithm 3 並沒有透過 **RowSimilarityJob** 求算 similarity matrix，而是用 SSVD 計算後得到的 U_R 、 Σ_R 根據公式的推導結果，求算：

$$\text{similarity matrix} = RR^T = U_R \Sigma_R^2 U_R^T$$

乘法的實作則和 Algorithm 3 **MultiplyAndRecommend** 的矩陣乘法類似，這部分將會在稍後進行更詳細的說明。且會在做完矩陣乘法後根據 threshold 值只留下部分用的到的 element，進行最後推薦。



4.3. Algorithm 3 : Distributed SSVD

Recommendation

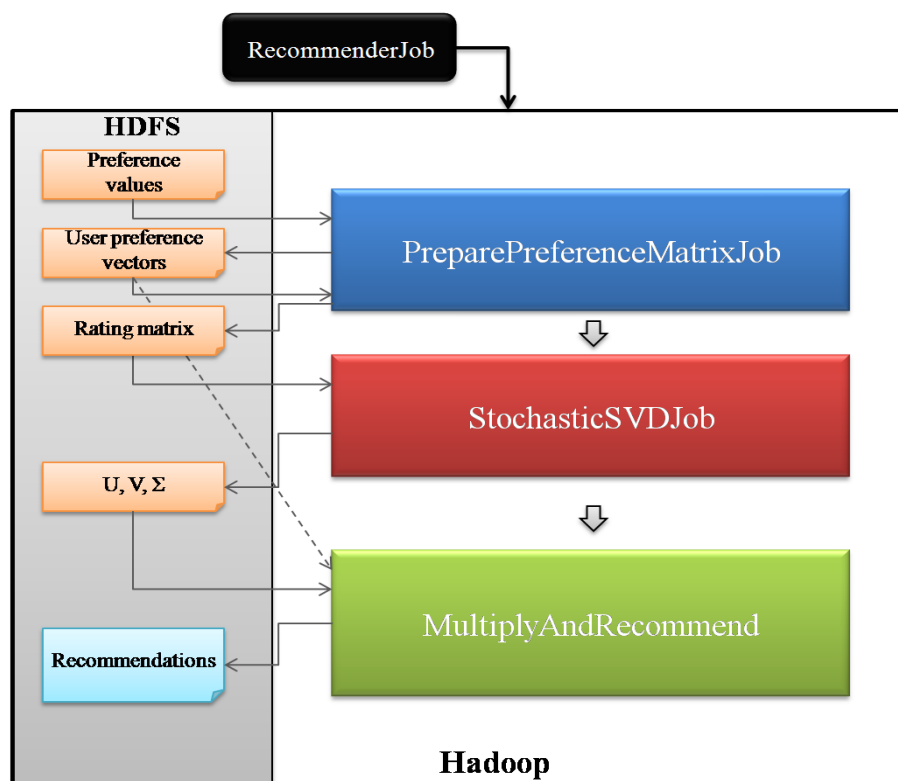


圖 4-6：Distributed SSVD Recommendation 於 Hadoop 實作之流程圖。

在進行 **Algorithm 3 : Distributed SSVD Recommendation** MapReduce 實作時，將此 RecommenderJob 分成三個部分，如圖 4-6 的流程圖。右半部的三個長方框分別表示三個主要的步驟，每個步驟裡可能又包括了數個 Mapper Reducer Tasks；左邊灰底部分代表 HDFS 的操作，橘色方框為儲存在 HDFS 裡的資料，細箭頭表示資料的讀寫，虛線部分為先前有但現在沒有的讀寫，粗箭頭代表程式的執行流程。

Algorithm 3 的 **PreparePreferenceMatrixJob** 和 Algorithm 1 的實作完全相同，因此本節不多做說明。**StochasticSVDJob** 為 Mahout Stochastic SVD Algorithm 在 Mahout Math Library 的實作，由於該 Job 的步驟相當複雜且包含與多 Mapper 和 Reducer Tasks，這裡只會對內部流程做大致說明。不過將會對 **MultiplyAndRecommend** 的 MapReduce 實作做詳細說明。

StochasticSVDJob

這個步驟為 3.1.3 Mahout Stochastic SVD Algorithm 的實作，該實作是 Mahout Library 中可獨立被執行的 Job。在用 MapReduce 實作的過程，又分成多個步驟、執行數個 Mapper 和 Reducer Tasks，在這裡將只會大致介紹該實作的順序和 MapReduce 流程[13]。流程圖如圖 4-7 所示，黑色框為可獨立執行的 Job，灰色框是執行中用來作特別處理的物件，藍色框是 StochasticSVDJob 中的各個步驟，藍框中的 M 和 R 分別代表該步驟是執行 Mapper、Reducer 或都有。

這裡的實作大致根據 **Modified SSVD Algorithm** 的步驟順序進行，接下來根據圖 4-7 的數字順序進行說明。首先，步驟①從 RecommenderJob 提交 StochasticSVDJob，步驟②在 SSVDJob 中的 Q-Job 先進行 Random sampling；然後配合步驟③Bt-Job 分別經過步驟①②執行另一個分兩階段實作 QR Gram-schmidt 的程序，如有需要可在過程中求算 BB^T ；再在步驟④的 ABt-Job 完成演算法中的 $B = Q^T A$ ；接著用步驟⑤的 UpperTriangular 和步驟⑥的 Eigenwrapper 物件從上面得到的結果中找出 Σ_R ；最後步驟⑦⑧再根據需求找出 U_R 、 V_R 。

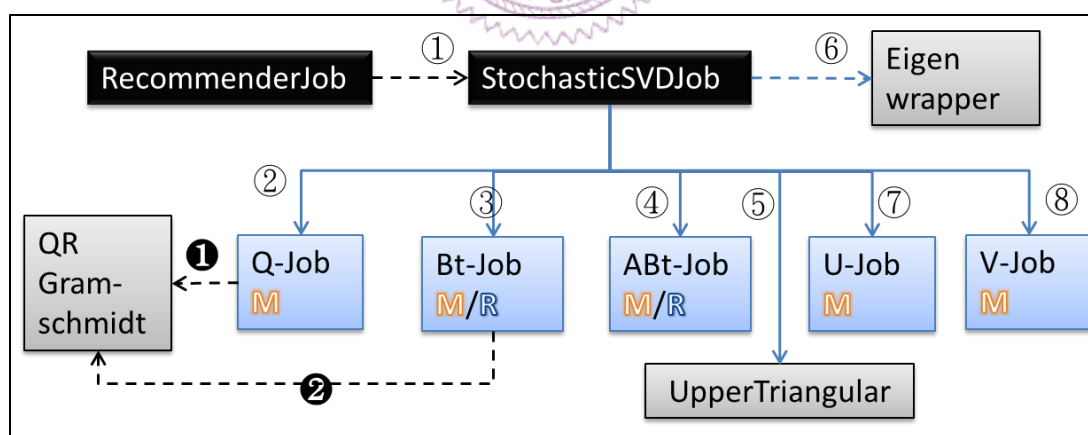


圖 4-7：Mahout Stochastic SVD MapReduce 實作流程圖。

MultiplyAndRecommend

根據之前的推導，這個部分所要做的就是 $V_k \Sigma_k^3 U_k^T$ 的計算。並在最後取出每個 row 的 Top-N elements，就是對應 userID 使用者的 Top-N recommendations。為了減少繁雜的 key-value 轉換和避免大量資料對 HDFS 的讀寫，這裡我們只有一個 Map Task 來進行矩陣計算和預測 Top-N recommendations：

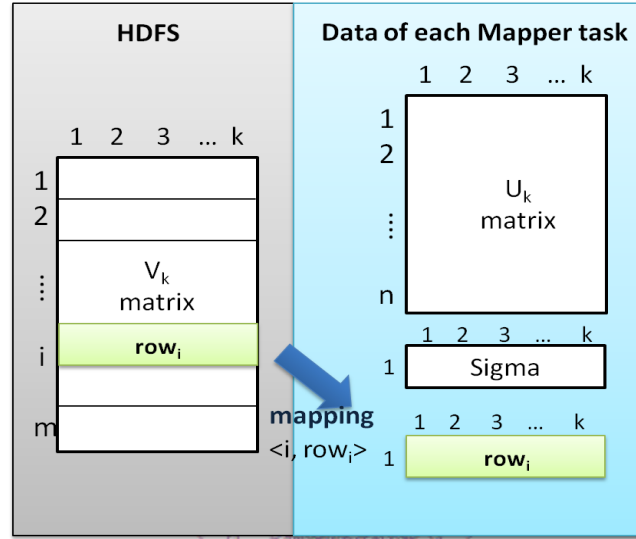


圖 4-8：Mapper 資料分佈示意圖。

- *Mapper* 在這裡將 $V_k \Sigma_k^3 U_k^T$ 的矩陣相乘分成三個部分，分別求算 Σ_k^3 、 $V_k \Sigma_k^3$ 和最後的 $V_k \Sigma_k^3 U_k^T$ 。在進行計算前，為了讓同一 Mapper Task 中同時有 U_R 、 Σ_R 、 V_R 的 data，除了 Mapper 的 Input pairs 外，另外用 Mapper Class 中的 private variable 直接從 HDFS 載入 U_R 和 Σ_R 到 Mapper Task 中， V_R 的部分則以 pairs 的方式 mapping 到對應的 Mapper，如圖 4-8 表示。

- I. Σ_k^3 ：SSVD 分解完成後的 Σ_k 是個對角矩陣，只將其對角線值存一向量 $\overrightarrow{\text{Sigma}}$ ，求算 Σ^3 也只需要讓 $\overrightarrow{\text{Sigma}}$ 的每個 element 自己乘三次方即可。

$$\vec{S}(j) = \overrightarrow{\text{Sigma}}(j)^3, j = 1, \dots, k. \quad (4-2)$$

- II. $V_k \Sigma_k^3$ ：因為在這裡使用 \vec{S} 表示矩陣 Σ_k^3 的結果，所以必須要讓所有代表 V_k rows 的 vector $\overrightarrow{\text{row}}_i: 1 \times k$ 乘以對應 \vec{S} ，用 $O(k)$ 的向量相乘取代 $O(nk)$ 的矩陣相乘。

$$\vec{S}(j) \times \overrightarrow{\text{row}}_i(j) = \overrightarrow{v}_i, i = 1, \dots, m, j = 1, \dots, k. \quad (4-3)$$

III. $V_k \Sigma_k^3 U_k^T$ ：最後將 II 求出的 \overline{v}_i 乘以矩陣 U_k ，求出來的 \overline{r}_i 就是給對應 userID 為 i 的使用者的 result vector。由於共有 m 位使用者，所以共會進行 m 次。

$$U_k \times \overline{v}_i = \overline{r}_i, i = 1, \dots, m \quad (4-4)$$

在得到每位使用者的 result vector 後，根據使用者設定的 threshold 值，只留下 \overline{r}_i 中足夠大的 elements，再對結果根據數值排序，選出 Top-N recommendations 推薦給使用者。

4.4. HDFS Reading and Writing

由於 Recommendation System 是個步驟相當繁雜的系統，在實作過程中需要多次對資料進行全部或部分的讀寫。在使用 MapReduce 實作時，須經過多次不同的 Mapper 和 Reducer Tasks，且 MapReduce 的分散式架構不利於過多的參數傳遞，因此過程中的大量資料傳遞就必須靠著 HDFS 的讀寫來進行。然而 HDFS 的讀寫卻有著難以解決的低延遲性 (Low-latency) 問題，因此在設計演算法時最好盡可能減少 HDFS 的讀寫次數和資料量。

以 **Algorithm 1: Mahout Item-Based Distributed Recommendation** 的實作為例，若將 **PartialMultiply** 的 *prePartialMultiply1* 和 *partialMultiply* 合併，減少一回合 HDFS 讀寫，則可以減少約 5% 的程式執行時間。因此在 **Algorithm 3: Distributed SSVD Recommendation** 省略了許多繁雜的步驟，其中一個目的就是減少 HDFS 讀寫，以達到較好的執行效率。

由於 PreparePreferenceJob、RowSimilarityJob 和 StochasticSVDJob 都是可單獨被執行的完整 Jobs，因此在這裡的討論將該部分的 HDFS 的讀寫皆視為一回。就整個 RecommenderJob 中的 HDFS 讀寫來討論，如表格 4 的標示，Algorithm 1 和 Algorithm 2 的 HDFS 讀寫次數各是六回，Algorithm 3 則是三回。

可看出 Algorithm 3 在 HDFS 的讀寫次數的部分有明顯改進，在測試結果也顯示 Algorithm 3 的執行效率最好，這部分將在下一章進行完整的討論。

Algorithm 1 : Mahout Item-Based Distributed Recommendation		Algorithm 2		Algorithm 3	
Mahout Item-Based Distributed Recommendation		Mahout Item-Based SSVD Distributed Recommendation		Distributed SSVD Recommendation	
Steps	HDFS	Steps	HDFS	Steps	HDFS
P.P.Job	1	P.P.Job	1	P.P.Job	1
R.Sim.Job	1	SSVDJob	1	SSVDJob	1
PartialMultiply	3	PartialMultiply2	3	Mult.&Reco.	1
Agg.&Reco.	1	Agg.&Reco.	1		
6 times		6 times		3 times	

表格 4：各演算法實作於 HDFS 讀寫次數比較表。



5.Experiments and Results

5.1. Environment

我們用了兩組 Cluster 來進行測試，第一組為台達電提供的 12 台實體機器 Delta，每台機器分別有四個 cores，共有 48 cores；第二組是國家高速網路與計算中心提供的 Virtual Machines Formosa，根據既有的 32 core 個數我們又分成四種 node \times core 的 Clusters，分別是 4×8 、 8×4 、 16×2 、 32×1 以進行 MapReduce Cluster Scalability 的比較。詳細規格如表格 5:Environment 配置規格。

	Delta	Formosa
Architecture	Real Machines	Virtual Machines Red Hat KVM RHEL 5.5.0 PC64 bits
Cluster size	12 \times 4 (48 cores)	32 cores (4×8 、 8×4 、 16×2 、 32×1)
Hardware	Per node	Per node
CPU	Dual Core AMD Opteron(tm) Processor 270 \times 2 1GHz 64 bits	QEMU Virtual CPU version (cpu64-rhel5) 2GHz 64bits
Memory	1024MB \times 16 (Memory array in 8)	4GB per core
Hard Disk	400GB \times 2	40GB (32 \times 1 Cluster is 20GB)
Operating System	Ubuntu Server 11.10 (GNU/Linux 2.6.38-13-generic x86_64)	Ubuntu Server 11.10 (GNU/Linux 3.0.0-12-server x86_64)
Libraries	Apache Hadoop 0.20.204 Apache Mahout 0.6(master only) Apache Maven 2.2.1 Java 6	Apache Hadoop 0.20.204 Apache Mahout 0.6(master only) Apache Maven 2.2.1 Java 6

表格 5：Environment 配置規格。

國家高速網路與計算中心提供的 NCHC GPU Cluster 是由自行研發建置的科技研發雲端運算平台主機，此主機採用 CPU 加 GPU 混合的運算架構，目前擁有 88 台計算伺服器，每台計算伺服器配備 2 顆六核心 Xeon 5670 2.93GHz 處理器、InfiniBand 40Gb/s 高速網路、3 張 Nvidia Tesla M2070 GPU、96GB 記憶體，搭配儲存節點共 42TB 硬碟空間。同時提供 IaaS 雲端服務和 GPU 計算服務，在這裡我們使用的是 IaaS 雲端服務。

Hadoop Configuration

有鑑於測試資料的大小及大量的計算過程，在這裡除了 Hadoop 的基本安裝外，另外針對我們的需求對 Hadoop 進行 configuration 設定。

- *hadoop-env.sh*：Hadoop heap size 設為 2GB。
- *core-site.xml*：為避免執行過程中因為過多 log 檔占用 disk space，限制 log 檔的最大 size 為 10MB 和最多 100 個檔案。
- *mapred-site.xml*：為了讓 Job 有效使用每個 node 的 Memory，根據[2]的建議，設定每個 tasktracker 上可執行的最大 map 或 reducer 任務數量為 core 個數減一，不然就使用預設值 2。並且配置更大的 JVM 記憶體給這些 tasks。
- *hdfs-site.xml*：根據 Cluster 的大小設定 3—8 不等的複製 blocks。

再來則是針對不同的 Cluster 設定對應的 masters 和 slaves IP。

而 Job 的 Mapper 和 Reducer 個數部分，直接在執行的 Command line 中加上 Streaming control command，在這裡都將 Mapper 和 Reducer 設為相同個數。

5.2. Evaluation

5.2.1.Data

Input Ratings File

資料來自於 GroupLens Research 的 MovieLens Data Sets [3]，MovieLens 是一個 web-based 研究用的 Recommendation System，在這裡我們使用該團隊隨機選取的 Data Sets—MovieLens 10M。該 Data Sets 中有 10000 個物品，取用 69878

位使用者的紀錄，共有 10000054 (10 million) 筆資料。每筆資料代表某位使用者對個物品的評分，每位使用者至少會有 20 筆以上的評分，分數範圍為 1~5。

首先我們將 Data Set 分成 training set 和 testing set，並用變數 x 用來表示資料比例。例如： $x = 0.8$ ，代表 Data Set 有 80% 為 training set，20% 為 testing set。如圖 5-1 所表示，training set 將會送進 recommender 進行推薦，testing set 則是在經過 data preprocessing 後用來和 recommendations 進行比較。

User File

在 Mahout 中為使 RecommenderJob 能給予特定使用者 Top-N recommendations，可在 User File 中列出欲被推薦的 userID list，讓 RecommenderJob 能根據對應使用者給予相對的結果。在本論文的實驗中，皆以推薦最多使用者為目標。因此 User File 內的 userID list 為 1~69878。

5.2.2.Quality

先前於 2.5 Evaluating Collaborative Filtering Recommendation System 提到三種針對不同目的及不同 Recommendation Systems，常見的 Recommender Quality Evaluation 方法，在這裡主要使用 Classification Accuracy Metrics 對推薦的結果進行 Quality 的判斷分析。

Classification Accuracy Metrics 較為重視演算法預測的結果正確性，而不是演算法排名預測的能力。但當物品種類相當多的時候，正確找出使用者可能有興趣的 Top-N 物品可能比找出使用者對“所有”物品的喜好排名來的重要。且本篇論文介紹的演算法皆以提供 Top-N recommendations 進行。因此在這裡選用 Classification Accuracy Metrics 而非 Predictive Accuracy Metrics 中常見的 MAE 來進行演算法 Quality 的比較。

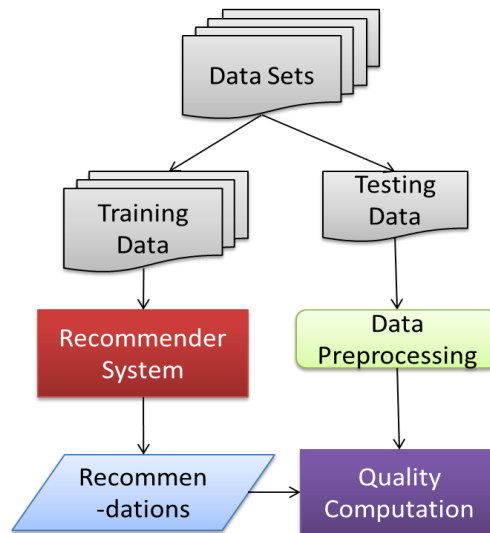


圖 5-1：Evaluation 流程圖。

在這裡我們使用 Sarwal et al.[6]提出 **Precision and Recall** 的改進方法來對結果進行分類及進一步的分析比較。如圖 5-1 所示，首先將 training data 丟進 recommender 得到 Top-N 個最佳的 recommendations，稱之為 *top-N set*。將其和 testing data 進行 preprocessing 後得到的 *test set* 進行比對，若某 item 同時出現在 *top-N set* 和 *test set*，則將其納入 *hit set* 當中。然後進一步定義 Recall and Precision：

- *Recall*. 在這裡定義 recall 為 test set 中的 item 有出現在 top-N set 中的比例值，也就是 $\text{recall} = \frac{\text{size of hit set}}{\text{size of test set}}$ ，可以將其寫成：

$$\text{recall} = \frac{|\text{test set} \cap \text{top-N set}|}{|\text{test set}|}$$

- *Precision*. 和 2.5 的 recommendation system evaluation 定義相同，視 precision 為 top-N set 個數中出現 hit set 個數的比例值，也就是 $\text{precision} = \frac{\text{size of hit set}}{\text{size of top-N set}}$ ，可以將其寫成：

$$\text{precision} = \frac{|\text{test set} \cap \text{top-N set}|}{N}$$

然而 Recall 和 Precision 在數值大小上是相反的，例如：當 N 越大時，因為 top-N set 的個數多， $|\text{test} \cap \text{top-N}|$ 也會隨之增加，因此 recall 的數值明顯會變大；但 precision 卻因為 N 變大而不盡然如此。因此，Sarwal 等人[6]使用 *F1 metric*，讓 recall 和 precision 在公平的情形下進行比較：

$$F1 = \frac{2 \times \text{recall} \times \text{precision}}{\text{recall} + \text{precision}}$$

當 F1 的值越大，代表其 Quality 越好。我們將會分別計算每位使用者的 F1，最後取平均值後進行比較。

最後，為了同時比較 Performance 和 Accuracy 的優劣，我們自己定義了一個 *P/A ratio*，用來當作另一個評比的標準。

$$P/A \text{ ratio} = \frac{F1 \times 1000}{\text{Execution Time (sec.)}}$$

當 P/A ratio 值越大代表 F1 的值越大或是執行時間越小，表示 Performance 和 Accuracy 的整體表現也越好。

5.3. Results

本篇論文的實驗分成測試環境和演算法的比較，測試環境方面分成不同的 Cluster 種類和 Cluster 的大小，演算法方面針對 Algorithm 1 : Mahout Item-Based Distributed Recommendation 和 Algorithm 3 : Distributed SSVD Recommendation 作比較，在這裡以 Algorithm 1 和 Algorithm 3 作為簡稱。分別皆會對執行時間、精準度和 P/A 的整體表現進行比較和討論。

本節將測試結果分成數小節進行討論，在 5.3.1 Settings 就演算法的參數設定進行討論，在 5.3.2 Performance 進行執行效率的比較，在 5.3.3 Accuracy 利用 F1 進行準確度的比較，於 5.3.4 Evaluation 透過 P/A ratio 評比 Performance 和 Accuracy 的整體表現，最後在 5.3.5 Discussion 對測試結果作進一步的討論。

5.3.1.Settings

本小節將對 RecommenderJob 執行時該設定的參數進行測試，找出最適當的設定值以進行接下來的實驗。

Mapper/Reducer Number

如之前所說，在執行每個 MapReduce Job 時，將 Mapper 和 Reducer 設為相同個數。為了找出最佳的 Mapper,Reducer 個數，以 Algorithm 1 : Mahout Item Based Distributed Recommendation 當作基本測試程式，根據不同 Mapper,Reducer 個數和 node 個數的比值進行測量與比較。在這裡的測試使用 Delta 的 Cluster，每個 node 的條件皆相同。

結果如圖 5-2 所示，當 Mapper,Reducer 個數和 node 個數的比值為 2 時有最快的執行時間，因此在接下來的實驗都將以此作為設定 Mapper,Reducer 個數的依據。除此之外，從圖 5-2 可以明顯看出，不論 Mapper,Reducer 個數和 node 個數的比值為何，當 node number 越多，Algorithm 1 所花費的執行時間越短。這表示 Algorithm 1 確實可透過增加 Cluster node 個數，改善執行的效率。

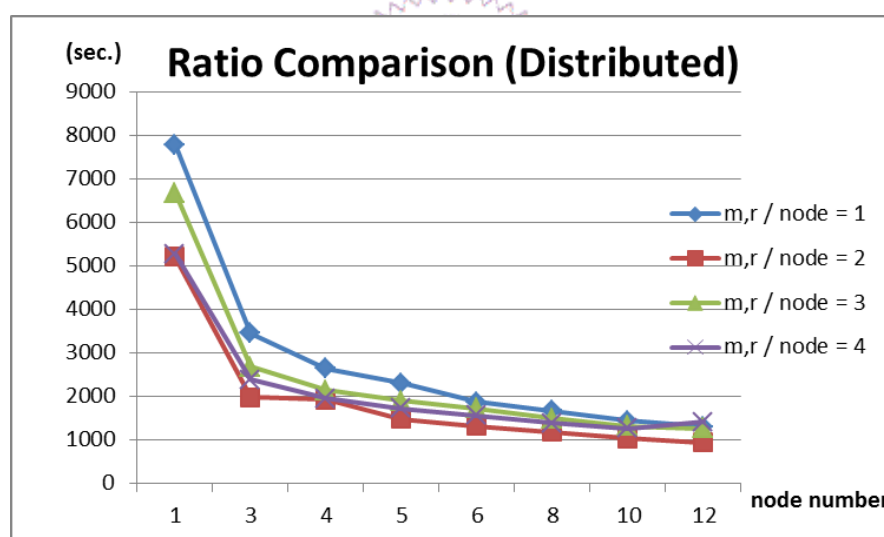


圖 5-2：Mapper,Reducer 個數與 node 個數比例關係圖。

SSVD value of k

為了決定計算 SSVD 時的 k 值，在這裡對 SSVD 的 Singular values 進行分析。在 k 時沒有設限制的情形下，Rating matrix 做完 SSVD 後，將 Σ 的所有對角線值取出來放到向量 \vec{S} 。由於 \vec{S} 本身即是一個由大排到小的向量，便直接將其值對應 s_i ， $i = 1, \dots, 329$ ，畫成圖 5-3。從圖中可看出 Singular values 的數值下降是非常劇烈且快速的，而此現象對 SSVD 非常有利，表示在計算 SSVD 時可以使用 $k \ll n$ 來計算，不僅能夠加快執行速度還不用擔心失去太多 Rating matrix 中的重要訊息。因此接下來的實驗將根據 圖 5-3 以 $k = 10$ 和 $k = 50$ 這兩種 rank 值來進行實作。

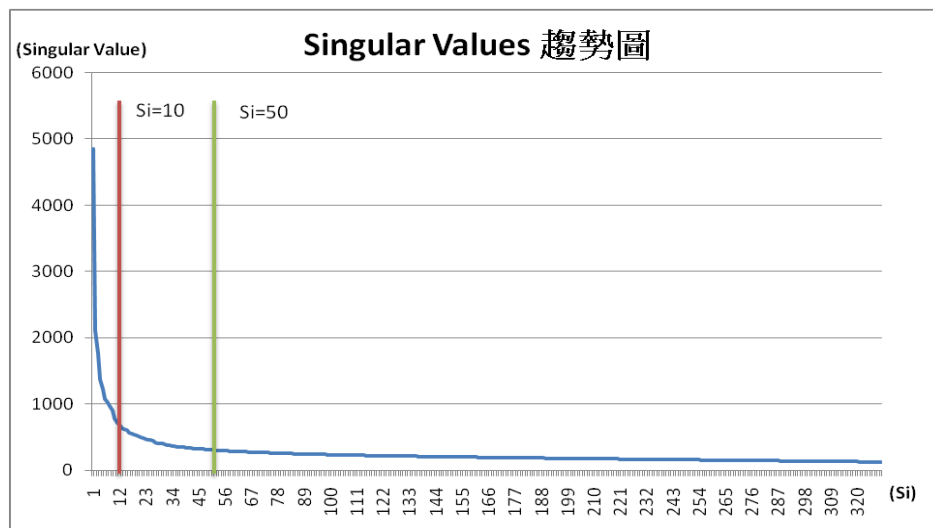


圖 5-3：SSVD Singular values 趨勢圖。

SSVD value of threshold

最後，因為 Algorithm3 中設有一可調整的 Similarity matrix threshold，又 threshold 的可設範圍相當廣，因此為便於接下來的各項評估，在這裡先對該 recommender 在不同 threshold 範圍下的表現進行比較。在圖 5-4 中，藍色曲線為 SSVD k 值設為 10 時，於不同 Cluster、對應不同 ratio training sets 的所有測量結果，根據不同 threshold 值執行結果取平均；紅色曲線是當 k 值設為 50 的結果，綠色的 average 則是將上述兩者相加平均。可以從圖中看出，不論 k 值多少，當 threshold 值設為 10^6 時，Algorithm 3 可以在 Performance 和 Accuracy 這兩部分有最好的表現，因此在接下來的各項實驗中有關 Algorithm 3 : Distributed SSVD Recommendation 的各項評比皆由 threshold 值設為 10^6 時數據進行比較。

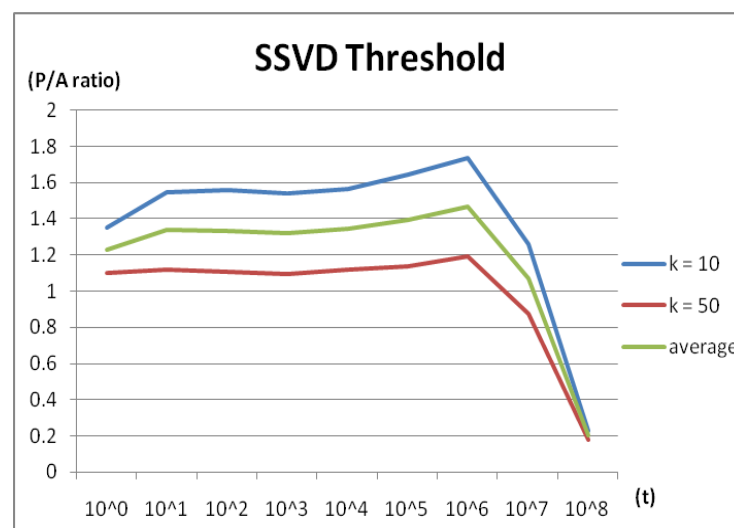


圖 5-4：Distributed SSVD Recommendation 於不同 threshold 之 P/A ratio 比較圖。

5.3.2.Performance

此小節討論 Algorithm 1 和 3 使用不同 ratio 的 training set 的執行時間，先分別比較在不同 Cluster 環境的執行，最後再就整體的執行時間進行比較。

這部分是在 Delta Cluster 架設環境下執行 Algorithm 1 和 Algorithm 3 的結果。圖 5-5 為 Algorithm 1 和 Algorithm 3 (k=10) 的執行時間，從圖中可看出在計算量不大時 (k=10)，Algorithm 3 的執行時間明顯比 Algorithm 1 快。但在 k=50 時 Algorithm 3 於 Delta 上的執行時間卻比 Algorithm 1 慢；圖 5-6 則是以速度最快的 Algorithm 3 (k=10) 為標準計算 Algorithm 1 和 Algorithm 3 (k=50) 相對應的 Speed Up，用 Algorithm 3 (k=10) 比 Algorithm 1 快 1.3~1.4 倍，比 Algorithm 3 (k=50) 快 1.7~1.9 倍。

從圖 5-5 data ratio 變化看出，基本上 ratio 變小，也就是 training set 變小、計算量減少時執行速度也越快。且圖 5-6 中計算量減少也可得到較好的 Speed up。由以上可推斷在 Delta Cluster 環境下，除了可使用計算量較小的 Algorithm 3 加快執行速度，data ratio 的改變和 Algorithm3 的計算量對於執行時間亦有影響。

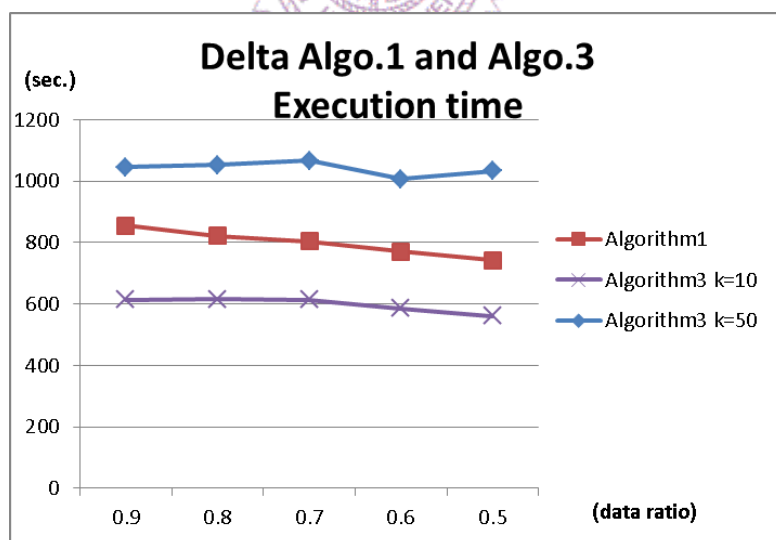


圖 5-5：Algorithm1 和 Algorithm3 於 Delta 之執行時間。

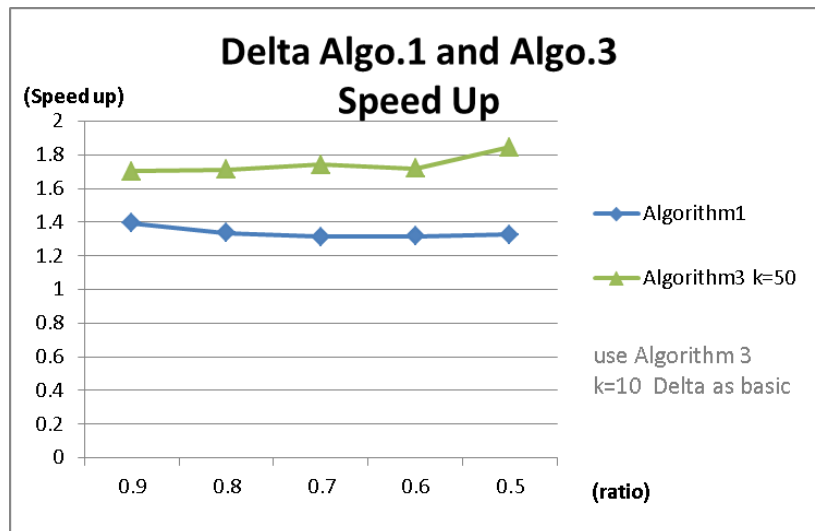


圖 5-6：Algorithm1 和 Algorithm3 於 Delta 之 Speed up。

接下來根據在 Formosa Cluster 架設環境下執行 Algorithm 1 和 Algorithm 3 的結果進行討論。Formosa 根據 node 和 core 個數分配有 c4 (4×8)、c8 (8×4)、c16 (16×2)、和 c32 (32×1) 的 configuration，在 Algorithm 1 的部分只用結果最佳的 c16 和 Algorithm 3 的四個結果比較。

從圖 5-7 的執行時間圖可看出 Algorithm 3 (k=10) 在 Formosa 上的速度都比 Algorithm 1 還要快，而且執行時間較穩定不會根據 data size 和 cluster size 而有明顯變動；圖 5-8 中的結果顯示 Algorithm 3 (k=50) 的執行時間較不穩定，且和 data size 的大小沒有相當一致的變化。尤其是 Algorithm3 c32 的執行速度幾乎都比 Algorithm 1 c16 慢，這部分將會在稍後的 Cluster Discussion 再做討論。

在圖 5-9 中可以看到光是用 Algorithm3 (k=10) 改善，就可以比 Algorithm1 約加速 1.5~1.8 倍。但在圖 5-10 Algorithm3 (k=50) 的結果卻非常不穩且加速效果明顯較差。由以上可推斷在 Formosa Cluster 環境下，除了可使用計算量較小的 Algorithm 3 加快執行速度，Algorithm3 的計算量變大時的加速效果和整體穩定度對於執行時間並沒有幫助。

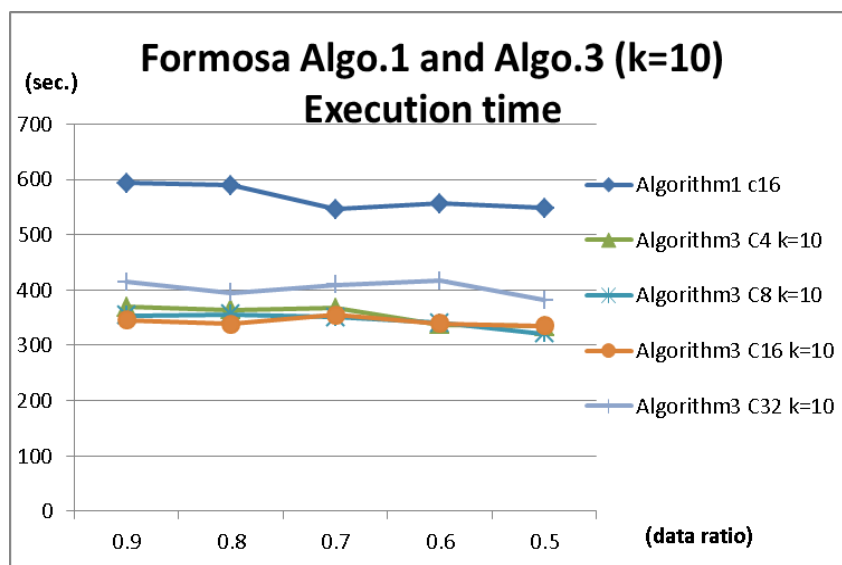


圖 5-7：Algorithm1 和 Algorithm3 (k=10) 於 Formosa 之執行時間。

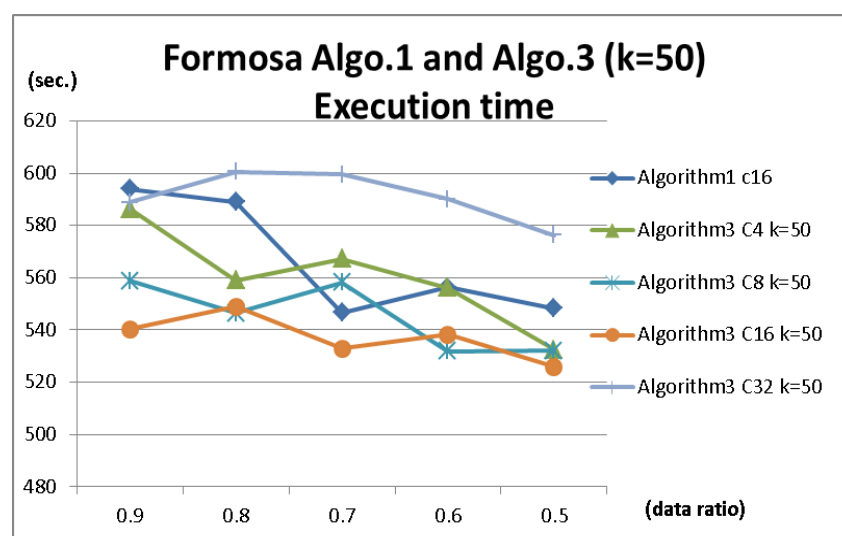


圖 5-8：Algorithm1 和 Algorithm3 (k=50) 於 Formosa 之執行時間。

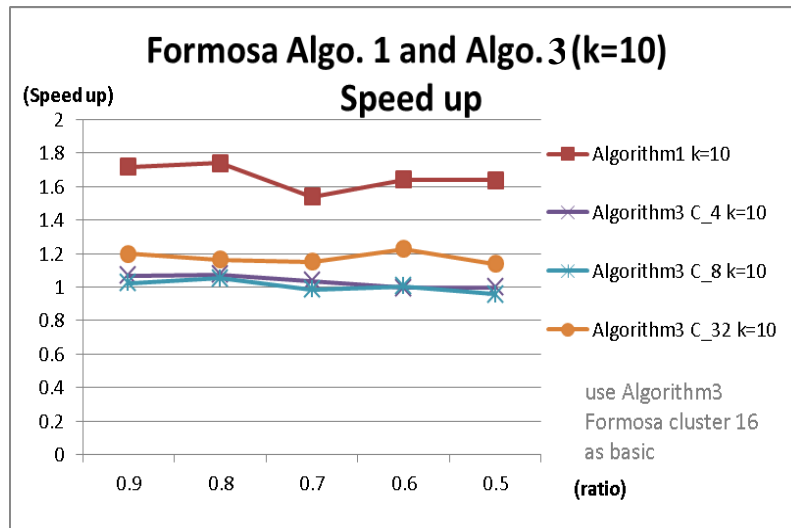


圖 5-9：Algorithm1 和 Algorithm3 (k=10) 於 Formosa 之 Speed up。

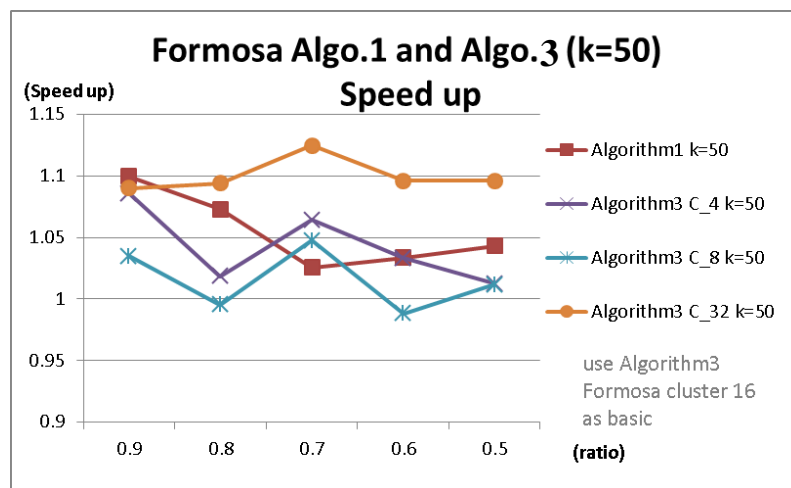


圖 5-10：Algorithm1 和 Algorithm3 (k=50) 於 Formosa 之 Speed up。

就整體的表現來看，若同時用 Algorithm3 和改變 Cluster 架設環境，以 Algorithm3 在 Formosa c16 的執行結果為標準求算其他的 Speed up。在 Algorithm3 (k=10) 的時候，從圖 5-11 可以看到能比 Algorithm1 在 Delta Cluster 執行快 2.2~2.5 倍；若只改變演算法或是 Cluster 環境，還是可以達到 1.5~1.8 倍加速；但若是 Algorithm3 只有改變 Formosa configuration，則沒有明顯的改善。

在圖 5-12 看出當計算量變大時，在只用 Algorithm3 代替 Algorithm1 沒有明顯改善，反而是透過只改變 Cluster 環境可讓 Algorithm3 加快 1.8~2 倍、讓 Algorithm1 加快 1.4~1.6 倍的執行。

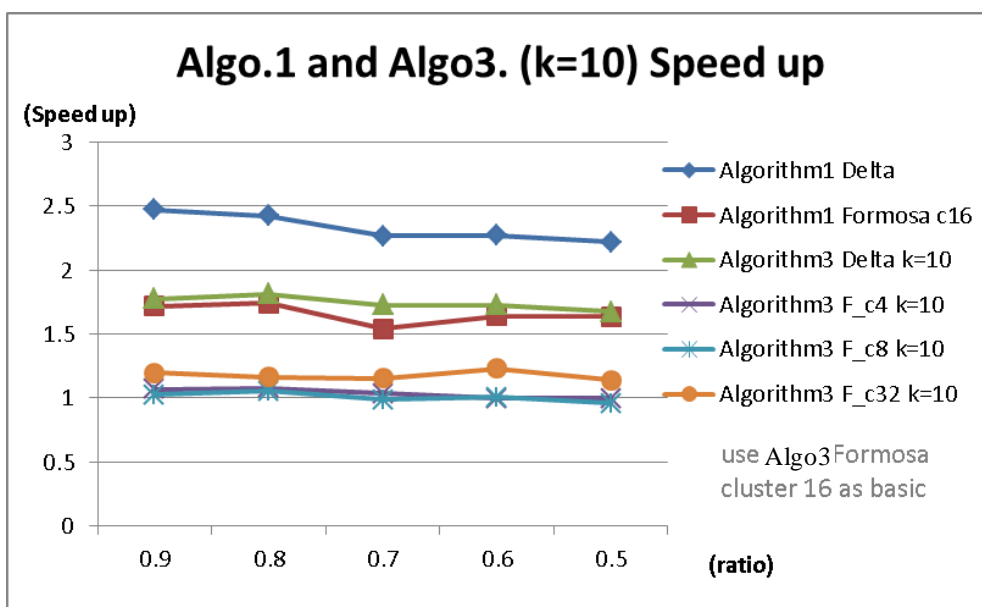


圖 5-11：Algorithm1 和 Algorithm3 (k=10) 整體之 Speed up。

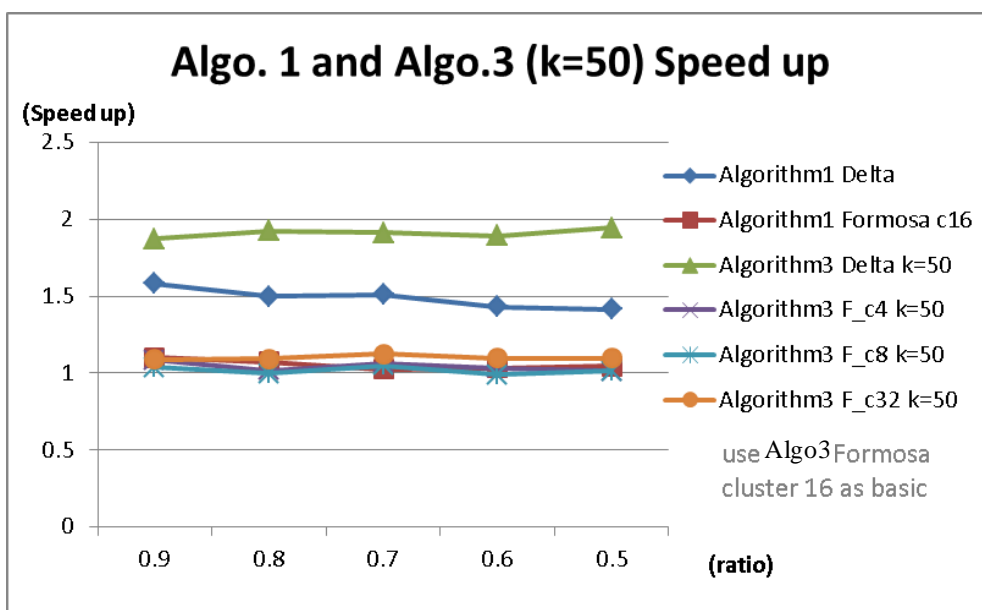


圖 5-12：Algorithm1 和 Algorithm3 (k=50) 整體之 Speed up。

5.3.3.Accuracy

此小節討論不同 Recommendation System 演算法使用不同 ratio 的 training set 和在不同 cluster 環境的執行輸出結果的精準度，利用之前提到 Precision and Recall 修改後的 F1 做為比較標準，一般來說當計算量越大也就是資訊越多時的精確度會越高。

圖 5-13 是 Algorithm1 在不同 Cluster 執行結果的 Accuracy 比較，因為是相同的演算法，所以不同 Cluster 做出來的結果於 F1 Accuracy 值也會差不多，所以接下來的部分將不會再對此多做討論。且可以明顯看出使用 Algorithm1，當計算量越大時（data ratio 越大）精確度也較高。

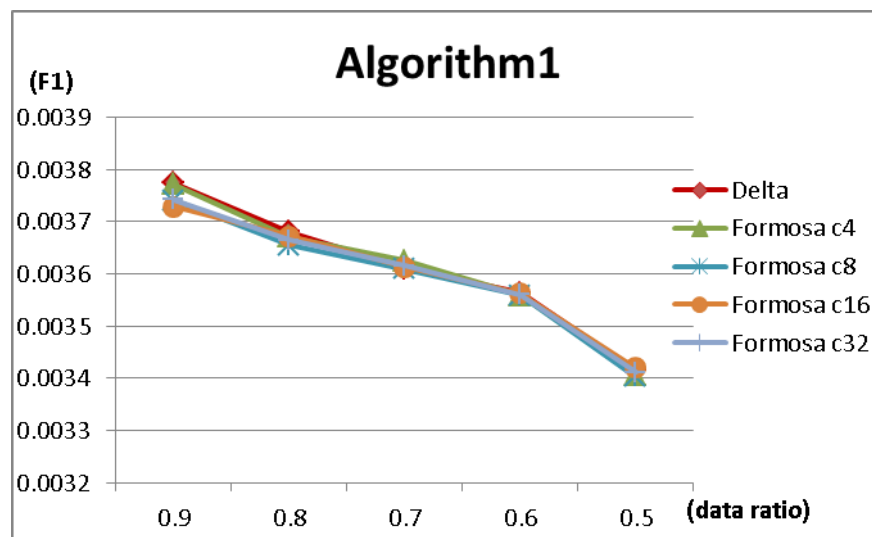


圖 5-13：Algorithm1 在不同 Cluster 上的 F1 Accuracy。

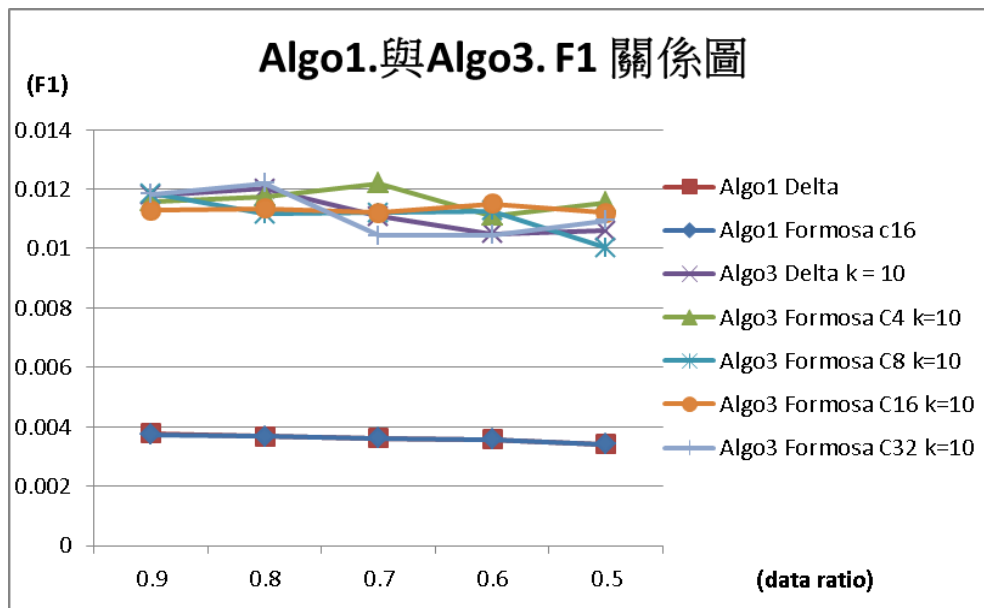


圖 5-14：Algorithm1 與 Algorithm3 (k=10) 的精準度比較圖。

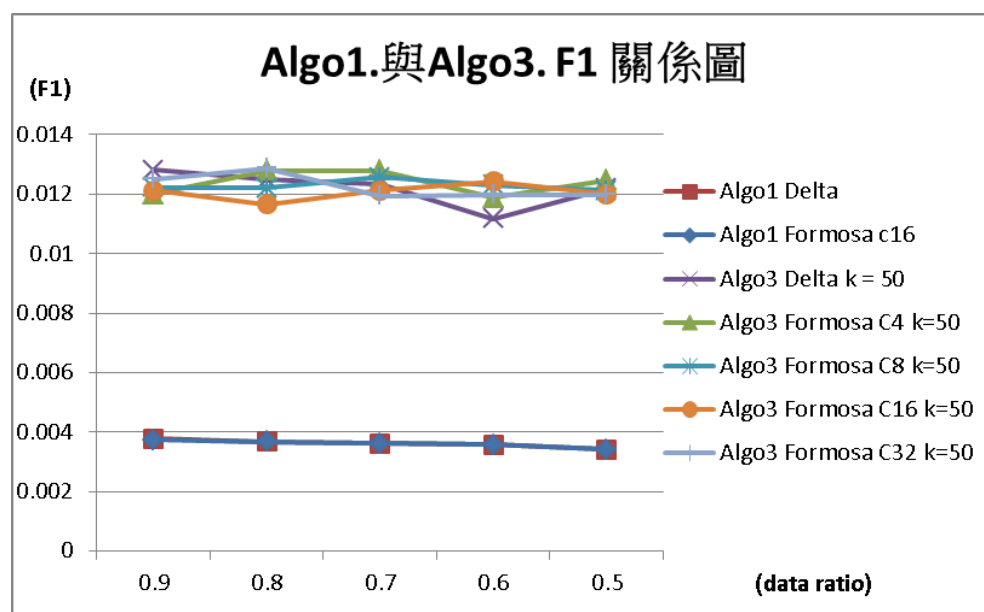


圖 5-15：Algorithm1 與 Algorithm3 (k=50) 的精準度比較圖。

從圖 5-14 可以發現 Algorithm3 的整體精準度明顯優於 Algorithm1，而且演算法的精準度優劣和 Cluster 並無太大的關聯。另外可以從圖 5-15 發現，當 Algorithm3 k=50 時的精準度並沒有因為 SSVD 的 rank 比 Algorithm3 k=10 較多而比較好，也沒有因為計算量較大而有明顯的提高，穩定性方面 Algorithm3 則比 Algorithm1 差。

5.3.4.Evaluation

此小節根據之前提到的 Performance 和 Accuracy，利用本篇論文中定義的 P/A ratio 作為評比與討論。P/A ratio 反應的數值反應了執行速度和最後的 Top-N recommendations 是否有確實符合使用者的喜好。

$$P/A \text{ ratio} = \frac{F1 \times 1000}{\text{Execution Time (sec.)}}$$

可將圖 5-16 的結果分成三個部分，第一個部分是因為演算法而導致結果較差的 Algorithm1 兩項，第二是受限於測試環境的而表現普通 Algorithm3 Delta，第三部分則是其他表現較佳的部分。從先前的圖 5-11 和圖 5-14 都可以看出和 Algorithm1 相比，Algorithm3 (k=10) 在執行時間和準確度兩方面都有明顯的改進，也因此 P/A ratio 的部分才会有第一和第三部分的明顯差異；另外也能從第二和第三部分看出在測試環境方面，Formosa Clusters 執行以 MapReduce 實作 Algorithm3 的表現幾乎都優於使用 Delta Cluster。

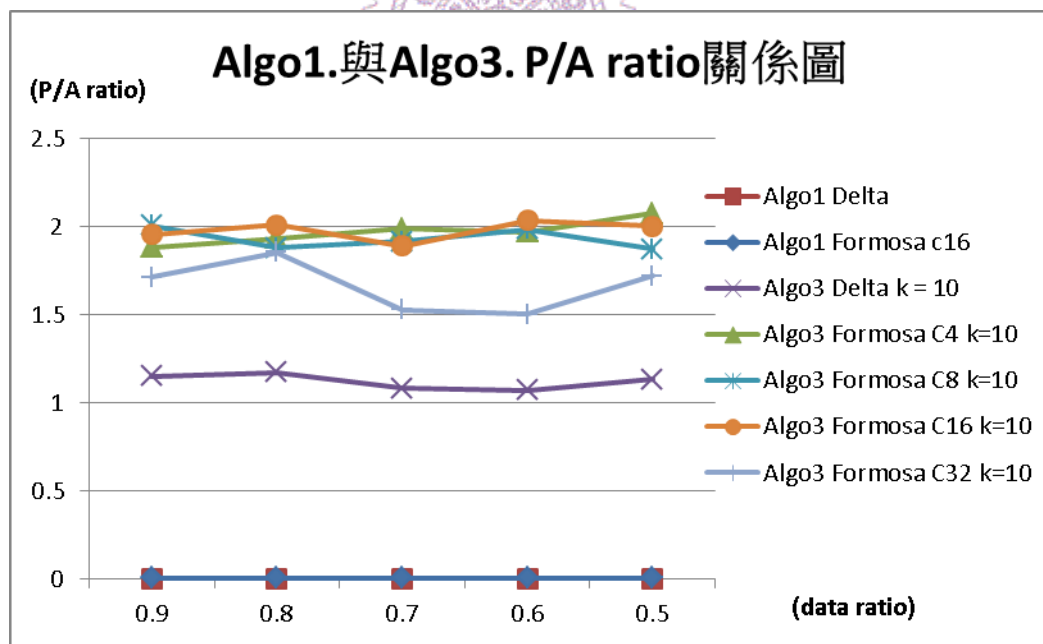


圖 5-16：Algorithm1 與 Algorithm3 (k=10) 的整體表現。

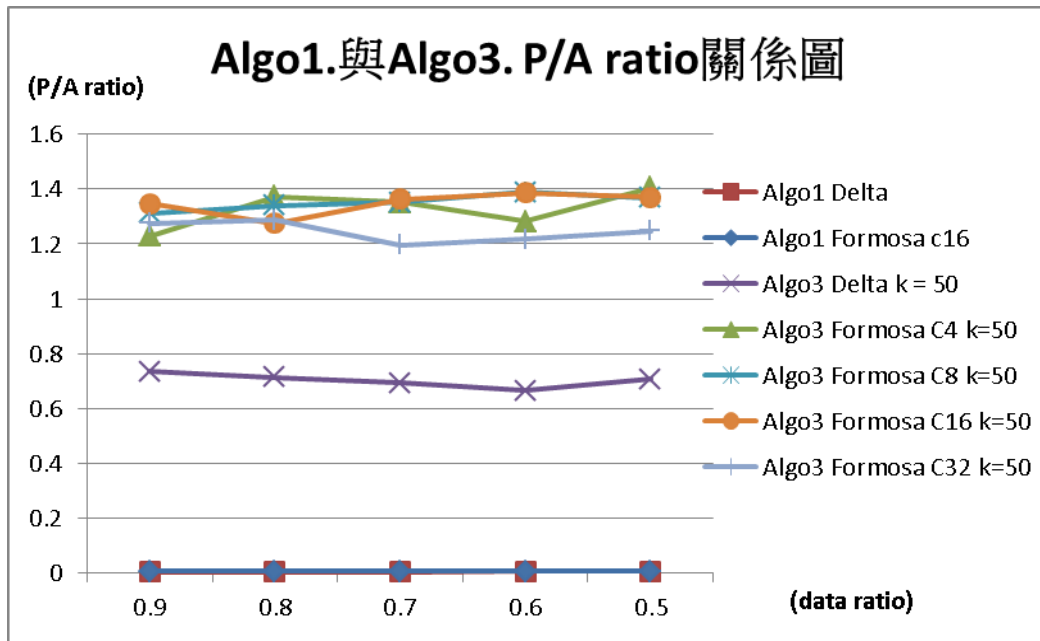


圖 5-17：Algorithm1 與 Algorithm3 (k=50) 的整體表現。

若只看 Algorithm3 的部分，從圖 5-16 (k=10) 和圖 5-17 (k=50) 中 P/A ratio 的數值即可看出，在測試 Algorithm3 時使用 k=10，比用 k=50 得到的整體執行結果更佳。表示在本實驗中圖 5-3 裡極大部分的 Singular values 皆可以被忽視，且不影響執行的成效。

5.3.5.Discussion

Cluster Discussion

在本篇論文中共使用了五個不同的 Cluster Configurations 來進行不同 CF Recommendation System 的測試，分別是 Delta 的 12 個 node，和 Formosa 分別為 4、8、16、32 nodes 的四個 Cluster。從圖和圖 5-19 中可以看出不論是哪種演算法，Formosa 的 Clusters 表現皆比 Delta 還好，這是因為兩組 Cluster 的測試環境條件有所不同而產生的差異。

用 Formosa 固定的 32 cores 數分成不同 node 和 core 數的 Cluster 測試 MapReduce Algorithm 的 Scalability。在圖 5-18 Algorithm1 可看出大致當 Cluster node 數越多，能夠讓 MapReduce 程式達到更好的平行化；但當 Cluster size 越大，每個 node 在 core 個數和整體 memory 大小的條件就越差，使得加速效果隨 Cluster size 增加而下降，甚至於在 Cluster size = 32 時的執行時間反而拉長，這點在圖

5-19 的比較相當明顯。且 Algorithm3 因為在作 StochasticSVDJob 的部分在 Cluster size 增加時並沒有明顯的加速，導致測試結果顯示在 Cluster size 增加時 Algorithm3 的加速效果並不如 Algorithm1 好。

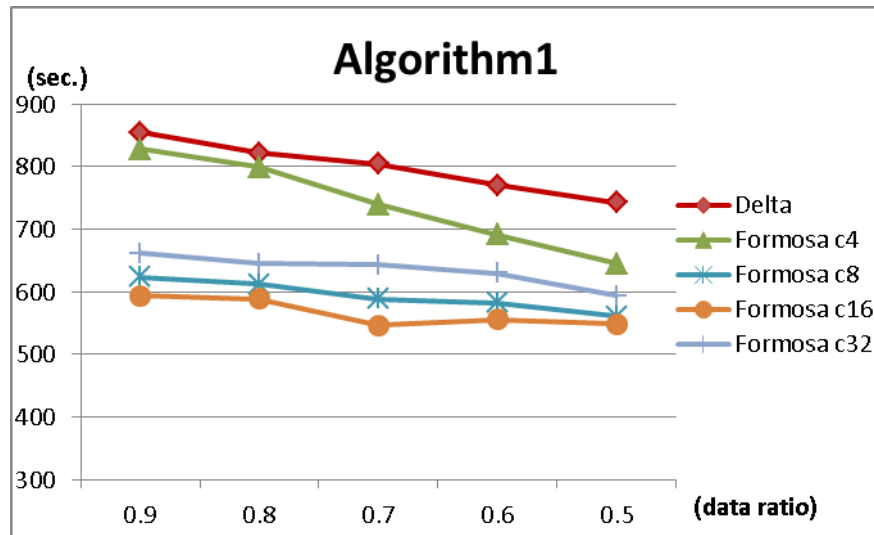


圖 5-18：Algorithm 1 在不同 Cluster 上的執行時間。

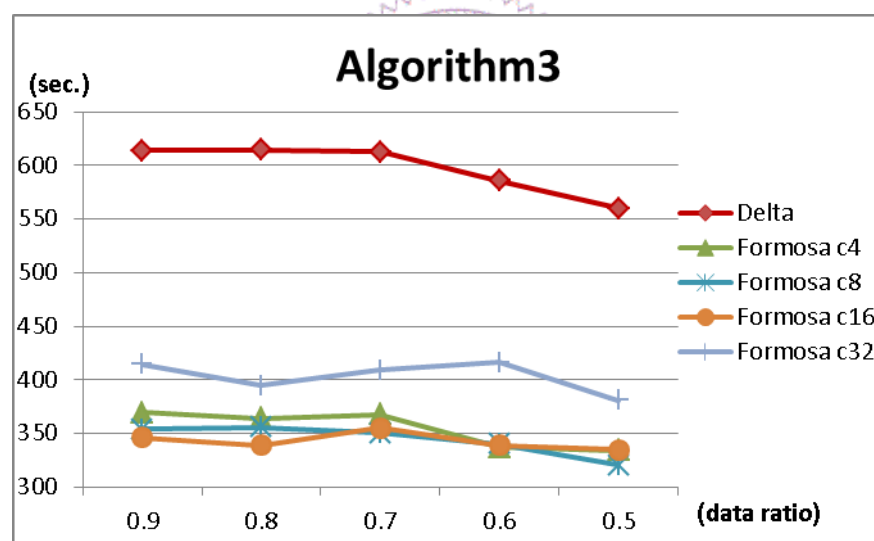


圖 5-19：Algorithm 3 在不同 Cluster 上的執行時間。

Algorithm Discussion

根據實驗的結果，本篇論文所實作的 Algorithm 3：Distributed SSVD Recommendation 不論在 Performance 還是 Accuracy 的表現，都優於 Algorithm 1：Mahout Item-Based Distributed Recommendation。從圖 5-20 和圖 5-21 可分別看出這兩個演算法中各個步驟執行所花時間的分佈，和在整個 RecommenderJob 執行

過程中所佔的時間比例，圖中直接以 Job 編號來表示各個步驟。

在圖 5-20 中 Mahout Item-Based Distributed Recommendation 大部分的執行時間是花在 Job2 $O(mn^2)$ 的 Similarity matrix 計算，雖然該部分比例測試環境的改善或是 Cluster size 的增加而有減少、進而加速執行時間，但因為 MapReduce 實作過程中 RowSimilarityJob 有多次的 HDFS 讀寫，而讓平行化的結果並不甚佳。Job1 則是受限於 Input ratings 的個數且和 Job3 都有繁複的 key-value pairs 轉換，導致難以透過平行化而加快執行速度。

在圖 5-21 的 Distributed SSVD Recommendation 中，顯而易見的是我們減少了一個 Job，並且不用 RowSimilarityJob 改用 StochasticSVDJob 和搭配的演算法。首先 Algorithm 3 藉由 Cluster 環境的改善而有顯著的加速效果，從圖最左邊的 Delta bar 可看出，Job3 的 **MultiplyAndRecommend** 經過 Cluster 環境的改善讓其在進行 MapReduce 處理過程中的執行時間有明顯的減少。Job2 的 SSVD 因為本來就有良好的平行化，所以並沒有因為 Cluster 的改變而有明顯的改善，但若和 Algorithm 1 的 RowSimilarityJob 相比，所需要的執行時間只需要不到一半。

雖然 Algorithm 3 並沒有因為 Cluster size 的增加而有明顯的加速，但整體來看我們替換了 Algorithm 1 原本在執行時間上的 Bottleneck—Job2，減少一個 Job，達到執行時間的改善。同時透過 SSVD 找出 Top-k Singular values，只留下重要訊息後再進行 Similarity 的計算和推薦。更能過濾掉大量資料中的 Noise，讓最後的推薦結果更加準確。

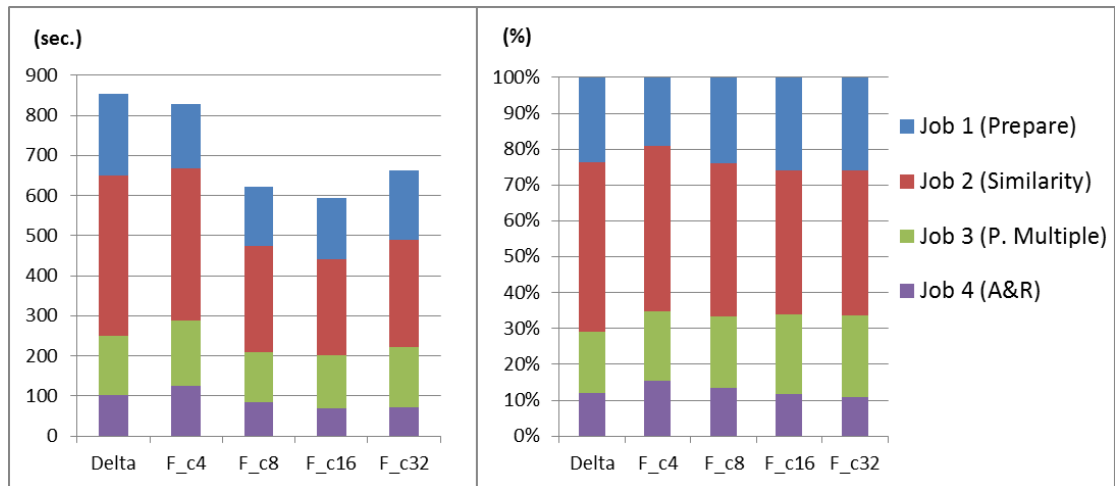


圖 5-20：Mahout Item-based Distributed Recommendation MapReduce 實作執行時間比例圖。

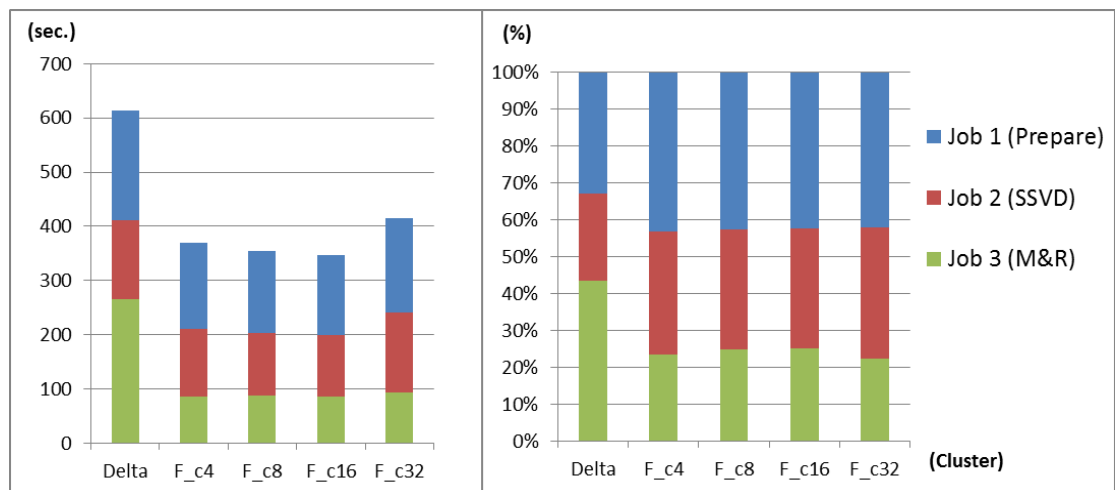


圖 5-21：Distributed SSVD Recommendation MapReduce 實作執行時間比例圖。

6. Conclusion

本篇論文首先對使用 MapReduce 實作的 Mahout Item-Based Distributed Recommendation System 進行詳細的分析和測試，而後根據需求使用 MapReduce Framework 實作出兩個 Distributed Recommendation System 演算法。

在這裡藉由結合 Mahout Item-Based Distributed Recommendation 和 Mahout Stochastic SVD，經過數學方法減少不必要的步驟和提高實驗結果的精確度，並透過 MapReduce 平行化程式以幫助加快整體執行速度。Distributed Item-Based SSVD Recommendation 提出用 SSVD 取代原先計算量較大步驟的 Recommendation System; Distributed SSVD Recommendation 則是配合對應的數學推導簡化原先步驟繁雜的演算法，使用 Mahout Stochastic SVD 從大量資料中找出重要資訊，更進一步改進上述演算法的執行效率。

使用兩組不同的 Cluster，且進一步用其中一組進行 Scalability 的測試。在針對不同演算法和 Cluster 環境下，經過大量的實驗並對其結果進行分析和比較。雖然 Distributed SSVD Recommendation 的執行速度並沒有因為 Cluster 大小而有明顯變化，還是有 1.5~2 倍的 Speed up，且在準確度方面有明顯的改善，整體而言在 Performance 和 Accuracy 都有顯著的提升。

Future Work

1. 對 PreparePreferenceJob 做進一步的分析，透過更有效率的方式進行初始資料的處理，讓所有 Recommendation System 的效能都有所提升。
2. 對 Stochastic SVD 的各項參數做更多的調整，例如對 SSVD 執行有直接影響的 k 、 p 值，透過實驗找出最佳的 P/A ratio，讓 Distributed SSVD Recommendation 的執行更有效率。
3. 修改 Stochastic SVD 的演算法，像是使用其他更有效率方法實作 QR 分解。
4. 對於 Apache Hadoop 架設的 Cluster 進行更符合演算法中 MapReduce Job 設定，以加快一般 Map Task 或 Reduce Task 的執行，或者可搭配其他 Apache Hadoop 相關 libraries 像是 Apache HBase...等，讓 Cluster 有更好的平行化效

果。

5. 嘗試使用其他平行化方法實作本篇論文提及的 Distributed Recommendation System，例如使用平行程式如 MPI、OpenMP，或搭配硬體設備使用 CUDA 在 GPU 上加速等。



References

- [1] S. Owen, R. Anil, T. Dunning, E. Friedman. *Mahout in Action*. Manning Publication Co., 2012.
- [2] T White. *Hadoop: The Definitive Guide, Second Edition*. O'reilly.(2010).
- [3] GroupLens Research. *MovieLens Data Sets*. <http://www.grouplens.org/node/73>.
- [4] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. April 11, 2010.
- [5] J. L. Herlocker et al. *Evaluating Collaborative Filtering Recommender System*. ACM Transactions on Information Systems, 2004.
- [6] B.Sarwar, G. Karypis, J. Konstan and J. Riedl. *Analysis of Recommendation Algorithms for E-Commerce*. In Proceedings of the 2nd ACM Conference on Electronic Commerce (EC'00). ACM. New York. pp. 285-295.
- [7] B. Sarwar, G. Karypis, J. Konstan and J. Riedl. *Item-Based Collaborative Filtering Recommendation Algorithms*. ACM/Hong Kong. 2001.
- [8] D. Billsus, M.J. Pazzani. *Learning collaborative information filters*. In Proceedings of the 15th International Conference on Machine Learning, Madison, WI, 1998, pp. 46-53.
- [9] M.G. Vozalis, K.G. Margaritis. *Using SVD and demographic data for the enhancement of generalized Collaborative Filtering*. Information Sciences 177 (2007), pp. 3017-3037.
- [10] SongJie Gong, HongWu Ye, and YaEDai . *Combining Singular Value Decomposition and Item-based Recommender in Collaborative Filtering*. IEEE 2009.
- [11] Luiz AndréBarroso and UrsHölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2009.
- [12] N.Halko, P. G.Martinsson and J. A. Tropp. *Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions*. SIAM Review, 53(2) (2011), pp.217-288.
- [13] Dmitriy Lyubimov. *MapReduce SSVD Working Notes. MapReduce QR decomposition*. 2010.

- [14] Nathan P. Halko. *Randomized methods for computing low-rank approximations of matrices*. Ph. D., Department of Applied Mathematics, University of Colorado. 2012.
- [15] T. Elsayed, J. Lin., and D. W. Oard. *Pairwise Document Similarity in Large Collections with MapReduce*. Proceedings of ACL-08: HLT, Short Papers, pages 265-268.
- [16] <https://cwiki.apache.org/MAHOUT/stochastic-singular-value-decomposition.html>
- [17] <http://horicky.blogspot.tw/2011/09/recommendation-engine.html>
- [18] <https://cwiki.apache.org/confluence/display/MAHOUT/RowSimilarityJob>

