植基於資料庫系統統計資訊之索引調校方法

廖宜恩　　高國峰　　張嘉俊
國立中興大學資訊科學系
ieliao@nchu.edu.tw、kfkao@cs.nchu.edu.tw、s9256048@cs.nchu.edu.tw

摘要

在資料庫系統中，我們經常藉由建立索引來增加讀取的速度。但是若建立不正確的索引，不但會浪費儲存的空間，而且還會增加索引更新時的成本。針對一個工作負載(workload)，如何建立一個有最佳效能的索引組態，我們則稱之為索引選擇問題。這個問題的傳統作法，需要針對每一個查詢語句，去詢問最佳化器(optimizer)每種索引組態的成本。這樣的作法相當浪費儲存空間及運算能力。為了避免這些缺點，我們利用儲存於資料庫系統中的統計資訊，及最佳化器在評選索引時，所使用到的成本數據，發展出一個新的索引選擇演算法。在這個演算法中，我們將會考量到功能相近索引之間的替代性，並且刪除不適合存在的索引。本論文所提出之方法已在 PostgreSQL 資料庫上實作，並以 TPC-H 的資料架構做實驗，以驗證本論文所提方法之有效性；實驗證明了我們提出的方法可以推薦好的索引組態，增加整個工作負載執行時的效能。

關鍵詞：資料庫系統、索引選擇、索引替代、索引調校、PostgreSQL。

1. 簡介

資料庫索引選擇問題(Index Selection Problem, ISP) [5]是一個在 NP-hard 的問題，學術界一直有人在這方面進行研究。所以也不斷的有人提出一些誘導式方法(heuristic)，以在有限的時間中，解出近似最佳解。索引的存在雖然可以增加資料庫的效能，但是建立不正確的索引不但無法加強資料庫的效能而且還會浪費儲存的空間，且增加索引更新時的成本;故如何推薦好的索引，使資料庫系統發揮最佳的效能，是索引選擇問題探討的重點。

在傳統資料庫中，索引選擇問題的做法是：記錄完整的工作負載(workload)以供事後評選時運算，以及針對每一個查詢去詢問最佳化器(optimizer)每一種索引組態的成本；這樣的做法相當耗費磁碟的空間及運算時間，為了解決傳統資料庫索引選擇問題的做法，本論文針對索引選擇問題提出了一個新的解決方法，我們記錄儲存於系統目錄中的有限資訊，將這些資訊彙總起來，並根據這些有限的資訊來評估整個工作負載的整體效能。

因為我們所擁有的資訊有限，但只要最佳化器決定在某一個查詢使用某個索引，我們就可以斷定使用這個索引的效能要比使用循序搜尋好。所以循序搜尋的成本可以當作這個索引的上限(upper bound)，當使用某一索引的成本大於循序搜尋的成本時，則我們可以說此索引不適合存在。此外，我們也發展了評估同一個資料表間索引替代性的技術。當此查詢語句的索引因為不適合存在而被刪除時，則此查詢語句會選擇其他的屬性做為索引，此時我們就必需將被刪除索引的成本轉移到其他的「替代性」屬性中。運用這些技術，我們就可以準確的預估索引值不值得建立，作為資料庫系統推薦索引的參考。

2. 相關研究

儘管索引選擇問題是 NP-hard 的問題，然而現今已發展出許多近似的解決方法來減少問題的複雜度。目前許多的商業資料庫系統軟體皆已將此功能實作於資料庫系統中。

2.1 索引選擇問題

索引選擇問題可以被描述為：在一個包含多個資料庫查詢(query)的系統中，決定要建立哪些索引，能使整個資料庫系統獲得最佳的效能，並且符合使用空間的限制，這些資料庫查詢的集合我們稱為工作負載(workload)。我們假設工作負載M包含m個資料庫查詢，此m個資料庫查詢以q_i表示，$1 \leq i \leq m$。

資料庫伺服器去查找符合查詢條件的方式稱為存取路徑(access path)，最基本的存取路徑為循序搜尋(sequential scan)，另外我們也可以使用索引當作存取路徑，有可能當作存取路徑的屬性欄位，我們稱為相關欄位(relevant columns)。所有相關欄位的聯集稱為候選索引集(candidate index set)，表示為N。我們假設N包含n個候選的索引，以a_j表示，$1 \leq j \leq n$。每個候選索引所佔的空間以d_j表示，$1 \leq j \leq n$；系統允許建立索引的空間上限以D表示。假設系統只提供一種索引，則每一個候選索引便有建立與不建立兩種狀態，因此N的冪集合便代表系統所有可能建立索引的狀態。我們把這個狀態稱為索引組態(index configuration)。我們假設所有索引組態的集合P包含p個索引組態，P包含的索引組態以c_k表示，$1 \leq k \leq p$。每個索引組態k都可對應到一

個索引的集合N_k，代表索引組態c_k中所建立索引的集合。

我們定義$cost(q_i, c_k)$代表在組態c_k時，執行資料庫查詢q_i所需要的執行時間成本。q_i使用c_k的獲利則表示成$gain(q_i, c_k) = \cos t(q_i, \phi) - \cos t(q_i, c_k)$，其中$\phi$代表空集合。如果以窮舉法(exhausted search)來解索引選擇問題(ISP)，便是求出所有索引組態加總資料庫查詢執行成本後的值；這個值如果符合空間限制並且總時間成本最小，便是問題的最佳解了。如圖 1 所示。

$$solution(ISP) = \min\left[\sum_{i \in M} \cos t(q_i, c_k)\right] \text{ and } \sum_{j \in N_k} d_j < D, \text{ for all } k \in P$$

上列的目標函式跟限制式，跟已經被深入研究的 UFLP(Uncapacitated Facility Location Problem)問題[2, 6]，擁有類似的數學模式。所以很明確的，索引選擇問題也是一個 NP-hard 的問題。

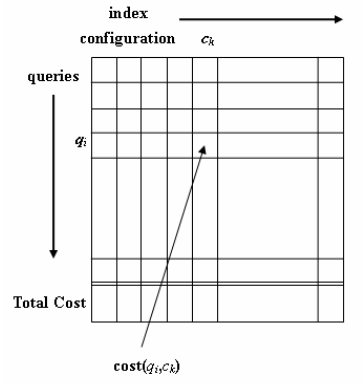

圖 1: 索引交互作用的成本估算表

2.2 Microsoft SQL Server

微軟(Microsoft)所發展的資料庫系統軟體 SQL Server，在 1997 年所提供的 SQL Server 7.0 中，提供索引微調精靈[4]，以方便資料庫管理者管理資料庫系統。該系統利用最佳化器評估工作負載成本的方式，實作索引微調工具，其主要步驟如下：

分析工作負載(workload)，將可能在尋找資料時影響成本的欄位一一挑選出來，並成為候選索引(candidate index)。接著仔細分析候選索引，並重新計算新的設定後的成本，判斷當該索引存在時的成本及造成其他查詢的成本影響等因素。經由不斷的計算，決定出最後的索引設定集合來推薦。最後則是更精確分析是否有必要使用多重欄位元索引的屬性測試，若有符合多重欄位屬性則加入候選索引中再次評估效益最後將推薦集合回應給使用者。

2.3 IBM DB2

IBM 所提供的資料庫系統 DB2，在 2000 年 DB2 6.0 版中亦提供索引微調工具[12]，DB2 微調方法與微軟的架構作法相似，以 SAEFIS 演算法，將查詢語句分成五個部份：

1. EQ：出現在等於條件的欄位。
2. O：出現在 order by、group by 及 join 欄位的集合。
3. RANGE：出現在範圍查詢的欄位集合。
4. SARG：出現在指定欄位查詢的欄位集合。
5. REF：其餘在查詢語句中被提及的欄位。

經過分析評估後，在不超過空間限制之下，挑出的欄位依重要性高低建立虛擬索引(virtual index)，並計算出可能需要的統計資訊，最後由最佳化器挑選，若虛擬索引出現在挑選集合中，則表示推薦該欄位建立索引。

2.4 Oracle

Oracle 公司在 Oracle 8i 的產品中也提供系統微調的工具：Tuning Pack 來因應管理者方便的管理資料庫系統，整個 Tuning Pack[8]所提供的微調工具包含下列數種相關的微調功能：

- Oracle Expert：利用一些簡易的步驟來完檢視並提升系統效能的工具。
- SQL Analyze：主要是分析使用者查詢語句的功能，可以詳細判斷查詢語句是否有改寫成更有效率的方式，避免造成資料庫發生嚴重的系統資源佔用而降低效能。
- Index Tuning Wizard：運作方式是以工作負載為基礎，經系統內部的分析後產生推薦結果回應給使用者。

3. 系統架構

本文提出的方式是希望索引選擇工具只透過系統目錄(system catalog)內的資訊就可以決定要推薦的索引組態；既不要儲存工作負載，也不要再透過資料庫最佳化器去估算各種不同的索引組態下的成本。圖 2 為系統目錄統計方法(CSA)索引選擇工具的工作流程架構圖。雖然我們在推薦索引時不再去問最佳化器，但不代表我們的決策沒有依照最佳化器的估算成本決定。相反的，我們的方式是完全依照最佳化器的資訊來決定，而這些資訊在執行查詢的時後便被分析並記錄在目錄中，這些記錄便是我們稱呼的聚集統計(aggregated statistics)，故此方式仍是最佳化器成本基礎(optimizer-cost based)的方式。我們將這樣的方式，歸納出下列幾種特性：

1. 不用再去分析或使用工作負載，沒有必要儲存工作負載。
2. 統計資訊所佔的空間只和資料表(table)中的屬性個數成正比。
3. 索引選擇工具使用目錄統計作為索引推薦的基礎。
4. 系統目錄會依每次的查詢計劃(query plan)累計資訊，其記錄的資訊以一般最佳化器挑選查詢計劃所運用的資訊及系統監控 I/O 的效能為主。

上述四點特性中的第四點，其所運用的資訊都

要求是即時取得的，不希望像傳統解 ISP 的演算法，運用虛擬索引(virtual index)去排列各種索引組態，以取得額外的數據。我們之所以採取這樣做法的主要目的是希望能善加利用最佳化器建立查詢計劃時所用到的資訊。如果我們將最佳化器評選計劃後的資訊記錄或者運算，所花費的額外成本幾乎為零，因為產生這些資訊的成本原本就是最佳化器建立計劃時就要付出的。我們適當的運算並記錄這些有用的資訊，正當我們在做索引選擇時，就不需要再去詢問最佳化器了。

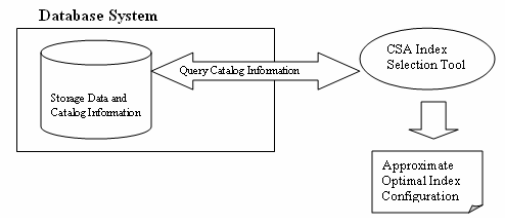

圖 2:CSA 索引選擇工具的工作流程架構圖

3.1 最佳化器的資訊與假設

最佳化器在執行查詢時，其所運用的資訊如果無限制膨脹的話，我們前述所討論的問題將無意義，故在此我們對最佳化器所該知道的訊息作一個明確的限制。我們提出以下兩個假設：

假設1. 最佳化器在決定查詢計劃(query plan)時，只知道(至少知道)其所使用查詢計劃的成本與使用循序搜尋的成本。

假設2. 最佳化器會從目前當下的查詢計劃中，挑選出成本最低的查詢計劃。換句話說，若最佳化器所挑選的查詢計劃有使用索引，則效能應該比使用循序搜尋好。

基於以上假設，我們定義了典型的最佳化器針對單一查詢評選計劃時，所用到的資訊：

資訊1. 查詢語句中，where 子句裡每個符合條件屬性的選取率(selectivity)。

資訊2. 每一個資料表使用循序搜尋的成本。

資訊3. 查詢語句的總成本及查詢樹(query tree)。

資訊4. 查詢樹中每一個節點(node)的成本及預估的資料筆數(tuple)是多少。

另外一類有用的資訊就是資料庫系統記錄了有關 I/O 次數的相關資訊，例如某個資料表檔案累計區塊存取(block access)的次數或者緩衝區命中(buffer hit)的次數。我們如果利用這些資訊，來做為評估索引效能的參考，其所花的「額外」成本幾乎為零，因為建立這些資訊的成本原本就是資料庫系統所要付出的。

典型的資料庫系統執行查詢時，所監控的訊息很多，以下我們定義兩項評估索引效能時會用到的資訊：

資訊5. 查詢執行時，每一個資料表檔案(table file)或索引檔案(index file)所花費的區塊存取次數。

上述的資訊中，**資訊 3** 代表的是最佳化器的預估值，它會包含部分中央處理器(CPU)運算的成本；而**資訊 5** 代表的是實測值，它單純是 I/O 的成本；**資訊 3** 和**資訊 5** 都可以代表一個查詢的成本。

3.2 查詢語句的成本模型

一個查詢語句可以分成查詢(query)、更新(update)、刪除(deletion)、插入(insertion)四種狀況。因為刪除和插入與更新類似，所以在此我們主要討論查詢及更新兩種狀況。在[4]中已建立一般性的成本模型，從這個成本模型，我們可以把 2.2 節定義的$cost(q_i, c_k)$再細分為讀取成本(read_cost)、維護索引成本(maintain_index_cost)及與索引無關的固定成本(constant_cost，包含更新資料表、輸出結果…等動作的成本)。表示成下列式子：

$$cost(q_i, c_k) = r\_cost(q_i, c_k) + mi\_cost(q_i, c_k) + c\_cost(q_i) \quad \cdots(1)$$

資料庫的維護索引成本(maintain_index_cost)對某一確定的工作負載而言是固定的，不會隨著索引組態不同而變動。所以對某一索引組態c_k的維護索引成本，我們表示為：

$$MI\_cost(c_k) = \sum_{i \in M} mi\_cost(q_i, c_k), \text{ for all } k \in P \quad \cdots(2)$$

故對某一索引組態c_k中的索引{a}的維護索引成本可以表示為：

$$MI\_cost(\{a\}) = \sum_{i \in M} mi\_cost(q_i, c_k) - \sum_{i \in M} mi\_cost(q_i, c_{k-\{a\}}), \quad a \in c_k \quad \cdots(3)$$

3.3 索引效能評估

我們將利用系統目錄中的有限資訊，比較索引存在的成本與不存在時使用循序搜尋的成本，來評估索引是否值得存在。索引的存在，對資料庫更新的動作必定要付出額外的維護成本，對查詢的動作則有可能提供效能較好的存取路徑。換句話說，一個索引對整個工作負載而言值得存在，是因為使用索引在讀取操作所獲得的利益大於其維護的成本。

是否使用某個索引作存取路徑，則會隨著索引組態的不同而改變。我們令現在的索引組態為$c_{current}$；在$c_{current}$時，最佳化器判斷使用索引a作存取路徑的查詢為$M_{(current,a)}$，$M_{(current,a)} \subset M$。$M_{(current,a)}$簡稱$M_a$。

令$M_{a'} = M - M_a$，對$M_{a'}$而言，最佳化器在$c_{current}$與$c_{current}$-{a}所挑選存取路徑的讀取成本將會一樣。因為我們假設最佳化器會挑選當時成本最低的索引組態，所以既然在$c_{current}$時不會使用索引a，則表示有比索引a成本更低的存取路徑，因此在$c_{current}$-{a}也會使用該成本較低的存取路徑，否則就違反最佳化器是理智的假設(**假設 2**)。

我們使用刪除索引的誘導式(heuristic)方法來

判斷索引是否適合存在。對索引a而言，當 $\sum_{i\in M}\cos t(q_i, c_{current}) > \sum_{i\in M}\cos t(q_i, c_{current-\{a\}})$，代表索引a不適合存在，因為對$M_a$′而言，其讀取成本是一樣的，所以只要考慮$M_a$的讀取成本及索引的維護成本。因此可將上式轉化成以下形式：

$$\sum_{i\in M_a} r\_\cos t(q_i, c_{current}) + MI\_\cos t(c_{current}) > \sum_{i\in M_a} r\_\cos t(q_i, c_{current-\{a\}}) + MI\_\cos t(c_{current-\{a\}}) \quad \cdots(4)$$

可以再化簡為：

$$\sum_{i\in M_a} r\_\cos t(q_i, c_{current}) + MI\_\cos t(\{a\}) > \sum_{i\in M_a} r\_\cos t(q_i, c_{current-\{a\}}) \quad \cdots(5)$$

其中$\sum_{i\in M_a} r_\cos t(q_i, c_{current-\{a\}})$代表索引a不存在的情況下，最佳化器可能挑的存取路徑，可能的做法為使用其他的索引或使用循序搜尋。由於我們無法知道在$c_{current}$-{a}的組態下，M_a是否使用其他索引，因此我們只好計算$\sum_{i\in M_a} r_\cos t(q_i, c_{current-\{a\}})$的上限值(upper bound)，亦即使用循序搜尋的成本。即為：

$$upper\_bound = (index\_scan次數)\times(sequential\_scan的成本) \quad \cdots(6)$$

所以會如公式(7)所示：

$$\sum_{i\in M_a} r\_\cos t(q_i, c_{current}) + MI\_\cos t(\{a\}) > (index\_scan次數)\times(sequential\_scan的成本) \quad \cdots(7)$$

如果公式(7)成立，則我們可以推斷索引 a 不適合存在，故將索引 a 刪除，對整個工作負載而言，會有更好的總成本(total cost)。

3.4 索引替代

在 3.3 節使用的方法，雖然在功能上已經足以篩選出一些明顯不好的索引，但準確度不夠高，原因在於：不同的索引組態，最佳化器可能會挑選不同的存取路徑。舉例來說，當我們用$c_{current}$的數據做判斷的基準，在刪除一個索引之後，原本使用這個索引的查詢語句，可能會轉而使用其他的索引，所以其他索引的搜尋次數就可能增加。因此，我們在改變索引組態後，繼續使用舊的成本資訊來決定要刪除哪一個索引，其正確性是可能產生偏差的。所以，我們發展一個技巧，從查詢語句上，去找出具有「替代性」的屬性集。具「替代性」的屬性集中，如果其中某一個索引被刪除，則因為其他屬性的索引有高度可能替代這個屬性的索引，執行類似的查詢計劃，則我們可以把被刪除索引的屬性，其使用索引相關的成本及索引搜尋次數轉換到替代屬性集裡尚未被刪除的索引。

刪除某個索引後，最佳化器會決定使用其他哪一個索引？且「利益」的轉移該如何「換算」？我們在此採取了一個誘導式(heuristic)的方法，我們定義了所謂具有「替代性」的屬性集，替代性的屬性集僅需觀察 SQL 查詢語句即可，替代屬性集內的屬性需要具有以下兩個特點：

1. 都是在 where 子句中，限定查詢條件的屬性。
2. 選取率(selectivity)都低於最佳化器使用索引的門檻(threshold)。

「利益」的轉移跟索引的特性有關。我們假設一個關係表R(relation R)有N筆資料(tuples)，使用B^+樹(B^+-tree)的索引，則根據[1]，存取索引的成本如下：

$$c_j = \lceil \log_b N \rceil + \left\lceil N \bullet s_j \bullet \frac{1}{b} \right\rceil - 1 \quad \cdots(8)$$

$b = block\_factor = number\ of\ pair(key, TID)\ \ for\ each\ page$

$s_j = selectivity\ of\ index\ a_j$

$c_j = the\ access\ cost\ of\ the\ a_j\ index\ in\ order\ to\ \ find\ the\ TIDlist$

由公式(8)，我們可以說，使用索引存取同一個資料表的成本大約與選取率(selectivity)成正比。我們就使用這個性質來換算具「替代性」屬性集裡索引的成本。

第二部分要轉移的是當使用索引時，對資料表存取的成本。我們使用[5]所提出的成本模型；從 p 個頁面中提取 nt 筆資料的頁面存取數(NPA)計算方式：

$$NPA(nt, p) = \begin{cases} nt & \text{if } nt \le \frac{p}{2} \\ \left\lceil \frac{nt+p}{3} \right\rceil & \text{if } \frac{p}{2} < nt \le 2p \\ p & \text{if } 2p < nt \end{cases} \quad \cdots(9)$$

3.5 系統目錄統計資訊

為了讓 3.3 節提出的評估索引效能的技巧能夠實現，我們設計了一個新的資料庫目錄的綱要，包含了 relation_cost、index_statistics、index_substitution 三個資料表。這三個資料表內所包含的屬性如表 1 所示。

relation_cost 記錄關係表(relation)相關的基本資訊。index_statistics 及 index_substitution 資料表則記錄資料庫執行時相關累計的統計資訊，除了 index_substitution 這個資料表中屬性的數值需要經由公式(8)及公式(9)按照選取率的比例去計算外，其餘兩個資料表中屬性的數值，都必須從 PostgreSQL 資料庫系統中的系統目錄(system catalogs)查得。

資料庫的每一種操作，依表 1 的定義做更新的動作。例如採取循序搜尋時，則對應的 n_table_scan 就會加 1；而在做索引查詢時，我們會從 SQL 的語句中找出所使用的索引是否有「替代性」的屬性集。如果有的話，會依照其選取率，挑選出一個當所使用的索引不存在時，最可能替換的索引，並將之記錄到 index_substitution 這個資料表中。當更新 index_substitution 的資料時，要依照 3.4 節敘述的成本模型，按照選取率的比例來計算轉換的成本。

表 1:資料表的名稱及其所包含的屬性

資料表名稱	屬性欄位名稱	屬性欄位描述
relation_cost	relid	關係表的 ID
	relname	關係表的名字
	table_scan_cost	循序搜尋的成本
index_statistics	relid	關係表的 ID
	attname	屬性的名字
	indexid	索引的 ID
	n_idx_scan	使用索引查詢的次數
	n_table_scan	使用循序搜尋的次數
	ifile_blks_read	索引檔案的區塊讀取數
	ifile_blks_write	索引檔案的區塊更新數
	rfile_blks_read	資料表檔案的區塊讀取數
index_substitution	indexid	索引的 ID
	tran_attname	替代的屬性名字
	n_tran_idx_scan	替代的索引查詢次數
	tran_ifile_blks_read	轉換到替代索引的索引檔案讀取數
	tran_rfile_blks_read	轉換到替代索引的資料表檔案讀取數

公式(7)中，$\sum_{i \in M_a} r_\cos t(q_i, c_{current}) + MI_\cos t(\{a\})$代表當使用索引 a 時，讀取操作的成本及維護的成本，其值會剛好等於 index_statistic 中索引 a 的 ifile_blks_read、ifile_blks_write 與 rfile_blks_read 相加。所以公式(7)可以改寫成：

$$ifile\_blks\_read + ifile\_blks\_write + rfile\_blks\_read > n\_idx\_scan \times table\_scan\_cost \quad \cdots\textbf{(10)}$$

當某一個索引被刪除後，我們便可以依照 indexid 檢查 index_substitution 資料表，將可以轉換的成本計算出來並累加到 index_statistics 資料表中。再依照 index_statistics 資料表中的資訊重新計算後，挑選出下一個要刪除的索引。

4. 系統實驗

我們將第 3 章的理論基礎實際實作在 PostgreSQL 7.3.3 中[9]，並使用 phpPgAdmin 做為使用者介面的開發工具，藉著透過網頁的操作來存取資料庫系統，而且以 TPC-H 的綱要(schema)及資料[11]做為我們實驗用的資料庫數據，並建立了實驗用的工作負載(包含查詢及更新)。

4.1 實驗步驟

首先在 PostgreSQL 資料庫系統中建立 TPC-H 的資料，再根據我們所建立的工作負載，從資料庫中擷取資訊來判斷索引適不適合存在：

1. 從 index_statistics 中計算每一個索引的 ifile_blks_read + ifile_blks_write + rfile_blks_read 的總合。
2. 將 index_statistics 中的 n_idx_scan 乘以 relation_cost 中的 table_scan_cost。
3. 比較 1 和 2 的結果，如果 1 的結果小於 2 的結果，則不予理會，反之，1 的結果大於 2 的結果，則此索引不適合存在，根據我們的演算法，應該將之刪除。
4. 根據被刪除索引的 indexid，從 index_substitution 中找出替代性的屬性，並將 tran_ifile_blks_read 及 tran_rfile_blks_read 轉換到替代性的屬性上。
5. 回到第一步驟，繼續查找不適合存在的索引。
6. 當所有的索引都查找完後，沒有要被刪除的索引時，此時的索引組態就是我們要推薦的索引組態。

4.2 實驗結果

我們使用兩種方法：一種是有使用索引替代的觀念，將刪除索引的利益轉換到替代性的屬性中；另一種為沒有使用索引替代，單純將不符合成本的索引刪除。

【方法一】無使用索引替代的方法

依照我們所建立的 index_statistics 資料表，我們可以計算每一個索引的成本及使用循序搜尋的成本，如表 2 所示；索引 c_address、c_phone、c_acctbal 的索引搜尋成本大於循序搜尋成本，故此三個索引都不適合存在，如果沒有使用索引的替代方法，則索引組態為{c_name、c_nationkey}，我們將之命名為 C1，故 C1={c_name、c_nationkey}。

【方法二】有使用索引替代的方法

如果使用索引替代的方法，則根據我們提出的演算法，會先將最不適合存在的索引刪除；由表 2 知，c_phone 最不適合存在，故將之刪除，並根據 c_phone 的 indexid 去查找 index_substitution 資料表，查找的結果發現刪除 c_phone 之後會將成本轉換到 c_address 及 c_acctbal，所以我們要將舊的統計資訊更新，將轉換後的成本分別加到 c_address 及 c_acctbal 中，如表 3 所示；更新完 index_statistics 後，要重新計算是否還有不適合存在的索引，結果發現 c_acctbal 符合刪除的條件，所以要將 c_acctbal 刪除並將成本轉換到替代性的屬性中，依照 index_substitution 資料表，c_acctbal 沒有替代性的屬性，故只需將 c_acctbal 從統計資料中刪除即可，如表 4；在刪除 c_phone 及 c_acctbal 後，索引組態變為{ c_name、c_address、c_nationkey }，我們將之命名為 C2，故 C2={c_name、c_address、c_nationkey}。

表 2:使用索引的成本與循序搜尋的成本

attname	indexid	n_idx_scan	ifile_blks_read	ifile_blks_write	rfile_blks_read	table_scan_cost	index_scan_cost
c_name	45832267	5	123	21573	3218	28360	24914
c_address	45832268	2	12	10864	685	11344	11561
c_nationkey	45832269	5	128	5618	21926	28360	27672
c_phone	45832270	3	93	14093	22920	17016	37106
c_acctbal	45832271	1	4	19513	346	5672	19863

表 3: 刪除 c_phone 後，各索引的相關成本

attname	indexid	n_idx_scan	ifile_blks_read	ifile_blks_write	rfile_blks_read	table_scan_cost	index_scan_cost
c_name	45832267	5	123	21573	3218	28360	24914
c_address	45832268	3	45	10864	4899	17016	15808
c_nationkey	45832269	5	128	5618	21926	28360	27672
c_acctbal	45832271	2	61	19513	9804	11344	29378

表 4: 刪除 c_acctbal 後，其他索引的相關成本

attname	indexid	n_idx_scan	ifile_blks_read	ifile_blks_write	rfile_blks_read	table_scan_cost	index_scan_cost
c_name	45832267	5	123	21573	3218	28360	24914
c_address	45832268	3	45	10864	4899	17016	15808
c_nationkey	45832269	5	128	5618	21926	28360	27672

我們使用這兩個索引組態(C1 及 C2)及工作負載實際驗證它們的效能；我們在此使用最佳化器的預估值(EXPLAIN)及實際執行工作負載所花費的時間，如表 5 是最佳化器預估的結果，表 6 是實際執行工作負載花費的時間；我們發現，不管是最佳化器的預估或是實際執行的結果，都顯示出使用索引替代方法來推薦索引組態的準確性及效能較佳，也證實了我們提出的方法可以有效的推薦合適的索引，來解決索引選擇的問題。

表 5:C1(無索引替代)與 C2(有索引替代)效能比較

索引組態	Total Cost (EXPLAIN)
C1={c_name、c_nationkey}	41424.12
C2={c_name、c_address、c_nationkey}	29876.99

表 6: C1(無索引替代)與 C2(有索引替代)實際執行的效能

索引組態	Execution Time (seconds)
C1={c_name、c_nationkey}	194.24409
C2={c_name、c_address、c_nationkey}	147.58797

5. 結論

本論文提出了一個使用統計資訊的方法，利用存在於資料庫系統目錄中的統計資訊，來評估索引適不適合存在；藉由刪除不適合存在的索引來增加資料庫的效能。我們將理論應用在開放原始碼的 PostgreSQL 資料庫系統上，根據記錄在系統目錄中的統計資料來計算每一個索引的索引檔案的區塊存取數及資料表檔案的區塊存取數並且加總起來，將其結果和循序搜尋的成本比較，如果大於循序搜尋的成本，則此索引不適合存在，所以必須將之刪除，並且將其成本轉換到替代性屬性集中；在實驗中，我們評估有考量替代性屬性集和沒有考量替代性屬性集，這兩種情形的索引組態，我們發現有考量替代性屬性集的結果，其效能較好，所以此方法可以有效的提供資料庫管理員哪些索引值得存在、哪些索引不值得存在，對整個資料庫的管理工作更有幫助。

參考文獻

[1] Barcucci, E., Pinzani, R., and Sprugnoli, R., “Optimal Selection of Secondary Indexes“, IEEE Transactions on Software Engineering, Vol. 16, No. 1, 1990. pp. 32-38.

[2] Caprara, A., Fischetti, M., and Maio, D., “Exact and Approximate Algorithms for the Index Selection Problem in Physical Database Design“, IEEE Transactions on Knowledge and Data Engineering, Vol. 7, No. 6, December 1995, pp. 955-967.

[3] Caprara, A., and Salazar, J.J., “A Branch-and-Cut Algorithm for the Index Selection Problem“, Discrete Applied Mathematics 92, 1999, pp. 111-134.

[4] Chaudhuri, S., Narasayya, V., "An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server," in Proceedings of the 23rd VLDB Conference Athens, Greece, 1997, pp146-155.

[5] Choenni, S., Blanken, H.M., and Chang, T., “Index Selection in Relational Databases“, In Proceedings International Conference on Computing and Information, April 1993, pp. 491-496.

[6] Cornuejols, G., Nemhauser, G.L., and Wolsey, L.A., “The Uncapacitated Facility Location Problem“, in P.B. Mirchandani and R.L. Francis (Ed.s), Discrete Location Theory, John Wiley, New York, 1991. pp. 119-171.

[7] Finkelstein, S., Schkolnick, M., and Tiberio, P., “Physical Database Design for Relational Databases“, ACM Transactions on Database Systems, Vol. 13, No. 1, March 1988, pp. 91-128.

[8] Oracle Manager Quick Tours http://otn.oracle.com/products/oem/htdocs/qtours/tuning/tune.htm

[9] PostgreSQL Documentation, [Administrator’s Guide], http://www.postgresql.org/docs/.

[10] Silberschatz, A., Korth, F. H., and Sudarshan, S., “Database System Concepts“, 4ed, Mc Graw Hill, Singapore, 2001, pp. 621-623.

[11] Transaction Processing Performance Council, TPC-H, http://www.tpc.org/tpch/.

[12] Valentin, G., Zuliani, M., Zilio, D. C., Lohman, G., and Skelley, A., “DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes“, Proceedings, 16th IEEE Conference on Data Engineering, San Diego, CA, 2000, pp.101-110.