前言

分散式資料庫系統對於大多數企業和眾多軟體應用程式來說是不可或缺的。這些應用程式提供邏輯和使用者介面，而資料庫系統則負責資料的完整性、一致性和冗餘性。

在 2000 年代初，如果要找資料庫，你只有幾個選擇，而其中大部分都在關聯式資料庫的範圍內，因此它們之間的差異相對較小。當然，這不意味所有資料庫都完全相同，但它們的功能和使用案例的確非常相似。

其中一些資料庫專注於水平擴展（horizontal scaling），或稱橫向擴展（scaling out），指透過運行多個資料庫實例作為單一邏輯單元，來提高效能和增加容量：例如 Gamma Database Machine Project、Teradata、Greenplum 和 Parallel DB2 等等。如今，水平擴展仍然是客戶對資料庫期望的最重要特性之一，這可以解釋為雲端服務的日益普及。通常情況下，比起將資料庫轉移到更龐大、更強大機器的垂直擴展，或稱縱向擴展（scaling up），水平擴展更容易啟動新實例，並將其添加到叢集中。而且，遷移可能很耗時，也會有導致停機的可能性。

在大約 2010 年左右，出現新的最終一致性（*eventually consistent*）資料庫，同時相關詞彙如 *NoSQL*，以及後來的大資料（*big data*）也開始越來越受歡迎。這 15 年來，開源社群、大型網際網路公司和資料庫供應商，創建了非常多的資料庫和工具，以至於在試圖理解其使用案例、詳細說明和特定細節時很容易迷失方向。

由 Amazon 團隊於 2007 年發表的 Dynamo 論文【DECANDIA07】，對資料庫界產生巨大影響，在短期內就啟發許多變體和實施。其中最著名的是由 Facebook 創建的 Apache Cassandra、由 LinkedIn 創建的 Project Voldemort、以及由前 Akamai 工程師創建的 Riak。

今天，這個領域再次發生變化：在經歷了鍵值儲存、NoSQL 和最終一致性時期之後，我們開始看到更具可擴展和高效能的資料庫，能夠執行複雜查詢，並提供更強的一致性保證。

本書的受眾

在技術會議的交流中，我經常聽到同樣的問題：「該如何更深入地了解資料庫的內部原理？我甚至不知道該從何處開始。」大多數有關資料庫系統的書籍，都未深入探討儲存引擎的實施細節，而且對於像 B 數存取方法也只是大致粗略的介紹。很少有書籍涵蓋新的概念，例如不同的 B 樹變體和日誌結構儲存，因此我通常推薦閱讀相關論文。

但每一位閱讀論文的人都知道這並不容易：因為常常缺乏背景資訊，措辭可能模糊不清，論文之間幾乎沒有關聯，而且很難找到。本書提供重要資料庫系統概念的簡明彙總以作為指南，幫助那些想深入了解的讀者；也可以作為備忘錄，提供給對這些概念已經熟悉的讀者參考。

不是每個人都想成為資料庫開發者，但本書將幫助那些構建使用資料庫系統軟體的人：包括軟體開發者、可靠性工程師、架構師和工程管理者。

如果你的公司依賴於任何基礎架構元件，無論是資料庫、訊息佇列、容器平台還是任務排程器，你必須閱讀專案變更日誌和郵件列表，以保持與社群的聯繫，並掌握專案中的最新發展。了解術語並搞清楚內部原理，將使你能夠從這些資源中獲得更多資訊，並更有效地使用工具來排除故障、識別和避免潛在風險與瓶頸；對資料庫系統運作的概覽和一般理解，也將有助於在出現問題時處理。憑藉這些知識，你將能夠做出假設、驗證假設、找到根本原因，並呈現給其他專案維護者。

本書也適合滿懷好奇心的讀者：對於那些喜歡學習事物而不需要立即應用的人，那些在空閒時間喜歡嘗試有趣事物的人，像是創建編譯器、撰寫自製作業系統、文本編輯器、電腦遊戲、學習程式設計語言和吸收新知識的人。

本書假設讀者具有一定的後端系統開發經驗，並且曾以使用者身分，使用過資料庫系統。有一些關於不同資料結構的先備知識，將有助於更快地消化書中的內容。

為什麼要閱讀本書？

很多人常常用資料庫系統實施的概念和演算法來描述資料庫系統：「這個資料庫使用八卦來傳播成員資格」（見第十二章）、「他們實施了 Dynamo」，或者，「這與 Spanner 論文中描述的很相似」（見第十三章）。又或是，當討論演算法和資料結構時，你可能會聽到類似這樣的描述：「ZAB 和 Raft 有很多共同之處」（見第十四章）、「Bw 樹就像在日誌結構儲存之上實施的 B 樹」（見第六章）、或者「他們使用了像 Blink 樹中的兄弟指標」（見第五章）。

確實，我們需要抽象概念來討論複雜的主題，但每次對話都花在討論術語上也很不切實際。擁有通用語言的快捷方式，將有助於我們將注意力轉移到其他更複雜的問題上。

學習基本概念、證明和演算法的一個優勢在於，這些永遠不會過時；當然，總會出現新的演算法，但通常是在傳統演算法中找到缺陷或改進的空間後，才創建新的演算法。了解歷史將有助於更清楚之中的差異和動機。

學習這些知識能激勵人心。你會看到各種不同的演算法，看到這個行業解決問題的方式，並敬佩起這些工作。同時，學習也將有所回報：你可以感受到腦海中多個拼圖快速共同移動的方式，最後形成一幅完整的圖景，讓你永遠都能夠與他人分享這些知識。

本書涵蓋範圍

本書既不與關聯式資料庫管理系統有關，也不與 NoSQL 資料庫有關，而是與所有種類的資料庫系統使用的演算法和概念有關，尤其專注於儲存引擎（storage engine）和負責分散式（distribution）運作的元件。

有些概念，例如查詢規劃、查詢最佳化、調度、關聯模型等，已經涵蓋在許多優秀的資料庫系統教科書中。這些概念通常是從使用者的角度來描述，但本書專注於內部原理。在第二部分的結論和各章節彙總中，你會找到一些有用的參考文獻，很可能就可以找到你對資料庫相關問題的答案。

本書並未討論查詢語言，因為這些資料庫系統中並沒有通用的單一語言。

為了收集本書資料，我閱讀超過 15 本書、300 多篇論文、無數篇部落客文章、原始程式碼，以及幾個開源資料庫的文檔。在決定是否將特定概念納入本書時，我的原則是自問：「資料庫行業和研究界的人是否在談論這個概念？」 如果答案為肯定的，就將這個概念添加到長長的討論列表中。

本書的結構

有一些支持可擴展性的資料庫例子具有插件化元件（例如【SCHWARZ86】），但相對較少見；同時，很多資料庫使用可插件式的儲存方式。同樣地，也很少聽到資料庫供應商談論查詢執行，但他們非常熱衷於討論讓他們的資料庫保持一致性的方法。

資料庫系統之間最重要的區別主要集中在兩個方面：儲存（*store*）資料以及分散（*distribute*）資料的方法，其他子系統有時也很重要，但本書不涉及這些內容。本書分為兩部分，第一部分討論負責儲存（*store*）的子系統和元件，以及第二部分負責分散（*distribute*）的子系統和元件。

第一部分 討論本地節點的處理過程，並著重在儲存引擎，這是資料庫系統的核心元件之一，也是最重要的區別因素之一。首先，我們從資料庫管理系統的架構出發，介紹根據主要儲存介質和布局的幾種分類資料庫系統方式。

接下來繼續討論儲存結構，試圖了解磁碟儲存結構與記憶體內結構的不同之處，介紹 B 樹，並涵蓋在磁碟上高效維護 B 樹結構的演算法，包括序列化、頁面布局和磁碟內存表示。隨後討論多種變體，以展示這個概念的威力，以及受 B 樹影響和啟發的各種資料結構多樣性。

最後，討論幾種日誌結構儲存的變體，通常用於實施檔案和儲存系統，介紹其動機和使用原因。

第二部分 介紹將多個節點組織成一個資料庫叢集的方式。首先討論構建容錯分散式系統理論概念的重要性，以及分散式系統與單節點應用程式的區別，以及在分散式環境中面臨的問題、約束和複雜性。

接著，深入探討分散式演算法。此處先介紹用於故障檢測的演算法，透過檢測報告故障並避免故障節點，有助於提高效能和穩定性。由於本書後面討論的許多演算法都依賴於理解領導概念，因此會介紹幾個用於領導選舉的演算法，並討論其適用性。

分散式系統中最困難的事情，其中之一就是我們討論的資料一致性概念，接著探討複製、一致性模型、副本之間可能出現的差異，以及最終一致性。由於最終一致的系統有時依賴於反熵以實現匯聚，和使用八卦以資料傳播，所以會討論幾種反熵和八卦的方法。最後，我們在資料庫交易背景下討論邏輯一致性，並介紹共識演算法作為結束。

無所不在的 B 樹

我們比蜜蜂更勇敢，而且喔……比樹延伸更長遠……

—小熊維尼

可以將 B 樹視為圖書館中的一間龐大目錄室：首先必須找到正確的櫃子，然後在那個櫃子上找到正確的書架，再從書架上找到正確的抽屜，然後在抽屜裡瀏覽卡片，找到你要尋找的那張卡。同樣地，B 樹構建了一個階層，有助於快速尋找和定位搜尋的項目。

正如第 25 頁的「二元搜尋樹」中的討論，B 樹建立在平衡搜尋樹的基礎上，但與其不同的是，它擁有較高的扇出，即更多子節點，和較小的高度。

在大多數的文獻中，通常將二元樹節點繪製成圓圈。由於每個節點只負責一個鍵，並將範圍分為兩部分，因此這種詳細程度是足夠且直觀的。同時，B 樹的節點通常會繪製成矩形，且明確顯示指標塊以突顯子節點和分隔鍵之間的關係。圖 2-7 將二元樹、2-3 樹和 B 樹的節點並排在一起，好方便理解它們之間的相似處和不同處。

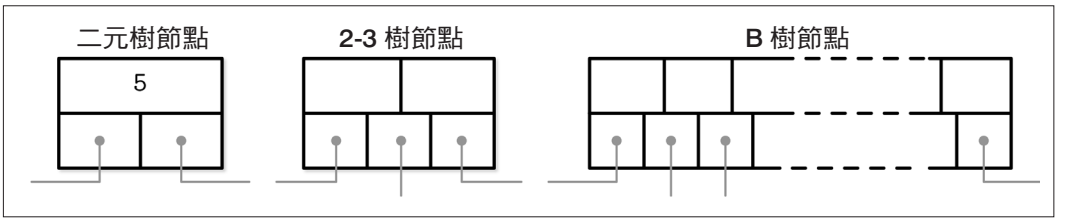

圖 2-7　二元樹、2-3 樹和 B 樹的節點並排在一起

可以用相同方式來描述二元樹。這兩種結構具有相似的指標跟隨語義，差異只在於如何保持平衡。圖 2-8 能看出這一點，並提示 BST 和 B 樹之間的相似性：在這兩種情況下，鍵將樹分裂為子樹，用於導航樹和查找搜尋的鍵。你可以將其與圖 2-1 比較。

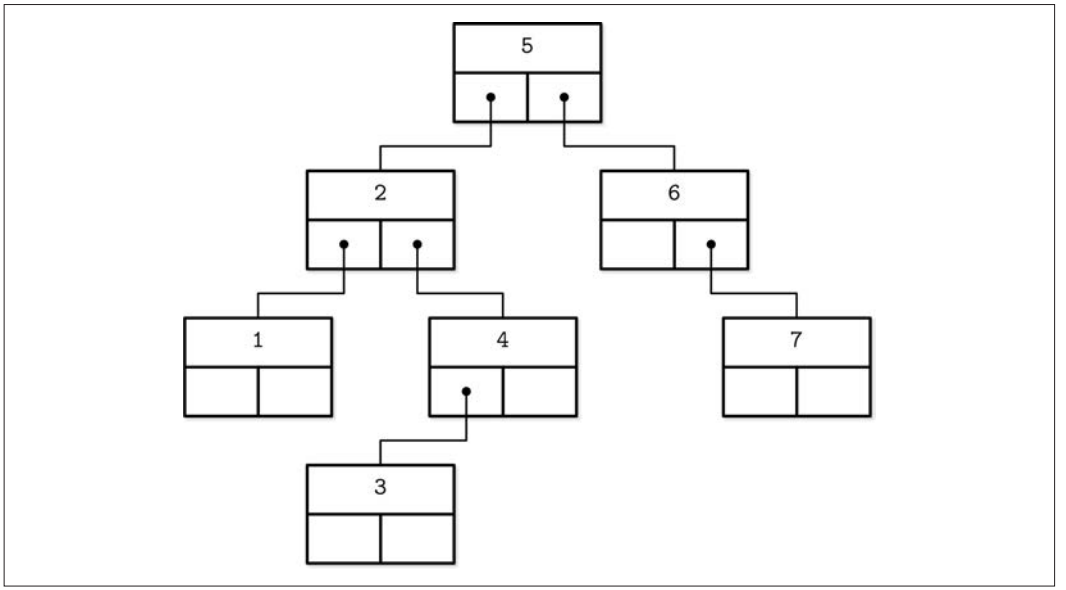

圖 2-8　二元樹的另一種表示方式

B 樹是有序的：B 樹節點內的鍵按照順序儲存，由於這個特性，所以可以使用類似於二元搜尋的演算法來找到搜索到的鍵，這也意味著 B 樹中的查找具有對數複雜性。舉例來說，在 40 億（4×109）個項目中尋找搜尋的鍵大約需要 32 次比較（有關此主題的更多資訊，請參見第 37 頁的「B 樹查找複雜性」）。如果每一次比較都必須進行一次磁碟查找，會明顯減慢速度，但由於 B 樹節點儲存了幾十個甚至幾百個項目，所以只需要在每級跳躍時進行一次磁碟查找，本章稍後會更詳細地討論查找演算法。

使用 B 樹，可以高效地執行點查詢和範圍查詢。點查詢通常由等於（=）的條件表示，在大多數查詢語言中，它用於定位單個項目；另一方面，範圍查詢由比較（<、>、≤ 和 ≥）的條件表示，用於依序查詢多個資料項目。

B 樹階層

B 樹由多個節點組成。每個節點最多包含 N 個鍵和 N + 1 個指向子節點的指標。這些節點在邏輯上分為三組：

根節點

沒有父節點，是樹的頂部。

葉節點

是樹的底層節點，沒有子節點。

內部節點

連接根節點和葉節點的其他所有節點，通常會有多層的內部節點。

階層結構如圖 2-9 所示。

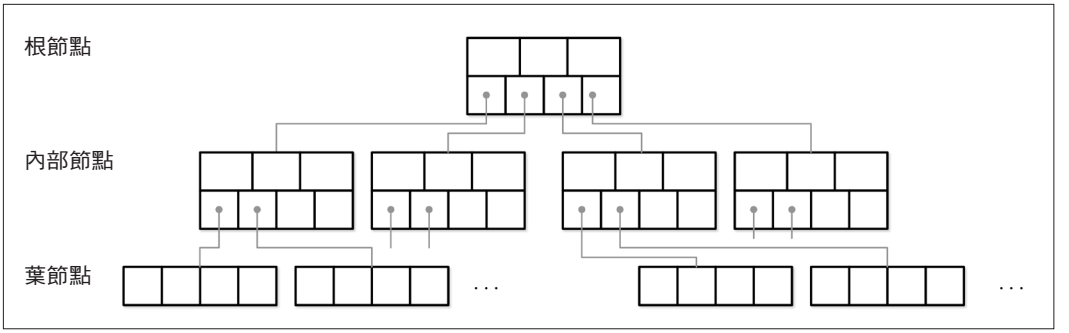

圖 2-9　B 樹節點階層

由於 B 樹是一種頁面（*page*）組織技術，即用於組織和導航固定大小的頁面，因此通常節點（*node*）和頁面（*page*）這兩個詞會互換使用。

節點容量和實際包含鍵數之間的關係，稱為占用率（*occupancy*）。

B 樹的特點是其扇出（*fanout*）：每個節點中儲存的鍵數。較高的扇出有助於分攤保持樹平衡所需的結構更改成本，並透過將指向子節點的鍵，和指標儲存在單個區塊或多個連續區塊中，來減少查找次數。當節點已滿或接近空時，會觸發平衡操作，即分裂（*splits*）和合併（*merges*）。

B^+ 樹

一般使用 *B* 樹這個詞，作為共享所有或大部分上述屬性的一系列資料結構統稱，但所描述資料結構更精確的名稱是 B^+ 樹，【KNUTH98】將具有較高扇出的樹稱為多路（*multiway*）樹。

值可以儲存在 B 樹中的任何層級：根節點，內部節點和葉節點。而在 B^+ 樹中，只有葉節點儲存實際的值。內部節點僅儲存分隔鍵，用於引導搜尋演算法找到儲存在葉級別的關聯值。

由於 B^+ 樹的值僅儲存在葉節點，所有操作，包括插入、更新、刪除和擷取資料紀錄等，也只會影響到葉節點，分裂和合併的過程才會影響到較高層級的節點。

B^+ 樹變得普遍，可同其他文獻稱為 B 樹。舉例來說，在【GRAEFE11】中，B^+ 樹為預設設計，MySQL InnoDB 將其 B^+ 樹實施稱為 B 樹。

分隔鍵

在 B 樹節點中儲存的鍵稱為索引項目（*index entries*）、分隔鍵（*separator keys*）或分隔單元格（*divider cells*）。它們將樹分成子樹（*subtrees*），也稱為分支（*branches*）或子範圍（*subranges*），並保存相應的鍵範圍。這些鍵按照排序順序儲存，以允許二元搜尋。可透過定位鍵，並跟隨相應的指標，從較高層到較低層來找到子樹。

節點中的第一個指標指向保存小於第一個鍵項目的子樹，而節點中的最後一個指標則指向保存大於或等於最後一個鍵項目的子樹。其他指標是兩個鍵之間的引用子樹：$K_{i-1} \le K_s < K_i$，其中 K 是一組鍵，K_s 是屬於該子樹的鍵。圖 2-10 顯示了這些不變式。

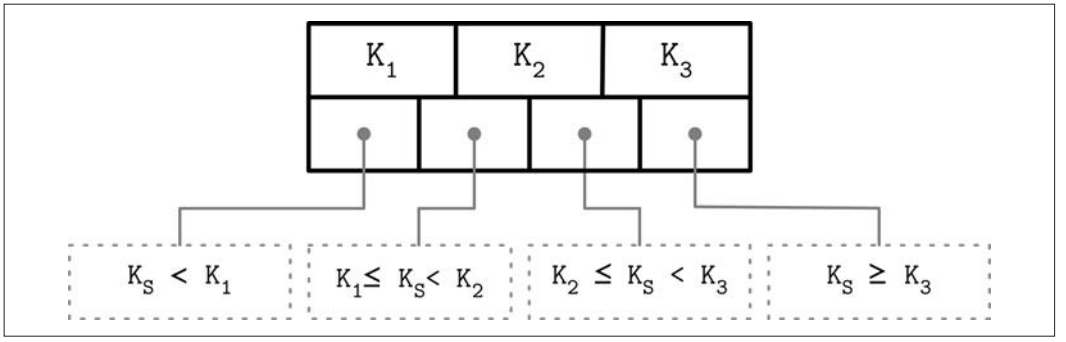

圖 2-10　分隔鍵如何將樹分成子樹

一些 B 樹變體在葉節點層上還有兄弟節點指標，以簡化範圍掃描操作，這些指標有助於避免返回父節點以查找下一個兄弟節點。有些實施在葉節點層上具有雙向指針，形成一個雙鏈表，這使得反向迭代成為可能。

B 樹的特點在於，如同二元搜尋樹，它們不從頂部向下構建，而是從底部向上構建。葉節點的數量增加，會導致內部節點數增加，也就增加樹的高度。

由於 B 樹在節點內保留額外空間以供未來插入和更新，所以樹的儲存利用率可能低至 50%，但通常要高得多，較高的占用率不會對 B 樹的效能產生負面影響。

B 樹查找複雜性

B 樹的查找複雜性可以從兩個角度來看：區塊傳送數和查找過程中進行的比較數。

就傳送數而言，對數的底數是 `N`（每個節點的鍵數）。在每個新層級上有 `K` 倍的節點數，而跟隨子指標會將搜尋空間減少 `N` 倍。在查找過程中，最多需要處理 $\log_K M$ 個頁面來找到所搜尋的鍵，其中 `M` 是 B 樹中的總項目數。換句話說，從根節點到葉節點的路徑上必須跟隨的子指標數，會等於樹的高度 `h`。

從比較數的角度來看，對數的底數是 `2`，因為在每個節點內使用二元搜尋來查找鍵；每一次比較都會將搜尋空間減半，所以複雜性是 $\log_2 M$。

了解搜索次數和比較次數之間的區別，有助於我們理解搜尋進行方式，並從這兩個角度理解查找複雜性。

在教科書和文獻中[2]，B 樹查找複雜性通常引用為 $\log M$。複雜性分析通常不使用對數的基數，因為改變基數只是增加了一個常數因數（*https://databass.dev/links/65*），而乘以常數因數不會改變複雜性。舉例來說，給定非零常數因數 `c`，$0(|c| \times n) == 0(n)$【KNUTH97】。

B 樹查找演算法

在論述 B 樹的結構和內部組織後，現在可以來定義查找、插入和刪除的演算法。要在 B 樹中找到一個項目，必須一次遍歷根節點到葉節點，這個搜尋目的是找到所查詢的鍵或其前身。找到完全匹配用於點查詢、更新和刪除，而找到其前身對於範圍掃描和插入非常有用。

演算法從根節點開始二元搜尋，比較查詢的鍵與儲存在根節點中的鍵，直到找到第一個大於查詢值的分隔鍵，這會定位一個被查詢的子樹。正如之前討論過的，索引鍵將樹分

2 例如【KNUTH98】。

成子樹，子樹的邊界位於兩個相鄰鍵之間。一旦找到子樹，跟隨相對應的指標，並繼續同樣的搜尋過程：定位分隔鍵，跟隨指標，直到抵達目標葉節點，在那裡不是會找到待搜尋的鍵，就是會找到它的前身，而可以得出它不存在的結論。

每一層都可以獲得更詳細的樹視圖：從最粗粒度層，即樹根開始，下降到下一層，其中鍵表示更精確、更詳細的範圍，直到最終到達資料紀錄所在的葉子。

在點查詢期間，搜尋會發生在找到或找不到搜尋鍵之後。在範圍掃描期間，迭代從最接近找到的鍵值對開始，然後繼續跟隨兄弟指標，直到到達範圍的結束，或範圍條件不滿足為止。

計算鍵數

在文獻中，你可以找到描述鍵和子偏移量計數的不同方式。【BAYER72】中提到了與設備相關的自然數 `k`，表示最佳頁面大小。在這種情況下，頁面可以容納 `k` 到 `2k` 個鍵，但可以部分填充並容納至少 `k + 1` 個，和最多 `2k + 1` 個指向子節點的指標。根頁面可以容納 `1` 到 `2k` 個鍵。稍後，引入了一個數字 `l`，任何非葉頁面可以有 `l + 1` 個鍵。

其他來源，例如【GRAEFE11】，描述節點可以容納最多 `N` 個分隔鍵（*separator keys*），和 `N+1` 個指標（*pointers*），其餘語義和不變性類似。

這兩種方法都會得到相同的結果，只有在強調每個來源的內容時會有不同。本書為了清楚起見，會一律使用 `N` 作為鍵數；若在葉節點的情況下則是鍵值對。

B 樹節點分裂

要將值插入 B 樹中，首先必須找到目標葉節點並找到插入點，為此可以使用前章節描述的演算法。在找到葉節點後，將鍵和值附加到該節點。透過使用查找演算法來定位目標葉節點，並將新的值與現有的鍵相關聯，好在 B 樹中更新。

如果目標節點沒有足夠的可用空間，可稱節點溢出（*overflowed*）【NICHOLS66】，必須將其分成兩個節點以適應新資料。更確切地說，如果以下條件成立，則節點會分裂：

- 葉節點：如果節點最多可以容納 `N` 個鍵值對，多插入一個鍵值對會使其超過最大容量 `N`。
- 非葉節點：如果節點最多可以容納 `N + 1` 個指標，多插入一個指標會使其超過最大容量 `N + 1`。

分裂是透過分配新節點來完成，將分裂節點的一半元件轉移到新節點，並將新節點的第一個鍵和指標添加到父節點中。在這種情況下，稱該鍵為*升級*（*promoted*）；進行分裂的索引位置稱為*分割點*（*split point*），也稱為中點。所有分割點之後的元件，包括葉節點分裂的分割點，都會轉移到新建的兄弟節點中，其餘元件則保留在分裂節點中。

如果父節點已滿，並且沒有空間容納升級的鍵和指向新建節點的指標，則父節點也必須分裂。這個操作可能會以遞迴方式一直傳播到根節點。

當樹容量已滿時，也就是分裂一直傳播到根節點，就必須分裂根節點，分裂時，會分配一個新的根節點，其中包含分割點的鍵。舊的根節點，也就是現在只保留一半的項目，將與其新建的兄弟節點一起降級到下一層，讓樹的高度增加一個層級。樹的高度在根節點分裂並分配新的根節點時，或者在兩個節點合併形成新的根節點時，會發生變化；而在葉節點和內部節點層級上，樹只會*水平*（*horizontally*）擴展。

圖 2-11 顯示在插入新元件 11 時的一個滿載葉節點。我們在滿節點中央畫一條線，將一半的元件保留在節點中，而將其餘的元件移動到新節點；分割點的值會放置在父節點中作為分隔鍵。

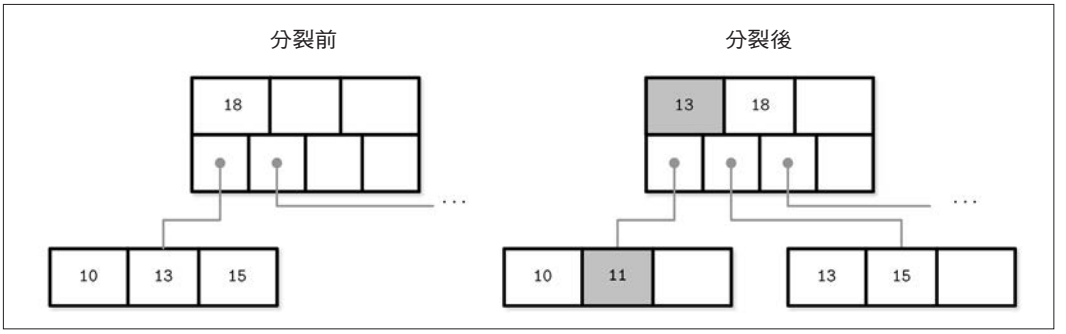

圖 2-11　插入 11 時的葉節點分裂，新元件和升級鍵以灰色顯示。

圖 2-12 顯示在插入新元件 11 時，一個已滿的*非葉*（*nonleaf*）節點，即根節點或內部節點的分裂過程。欲分裂，要先創建一個新節點，並將元件從索引 `N/2 + 1` 開始移動到新節點。分割點的鍵會升級到父節點。

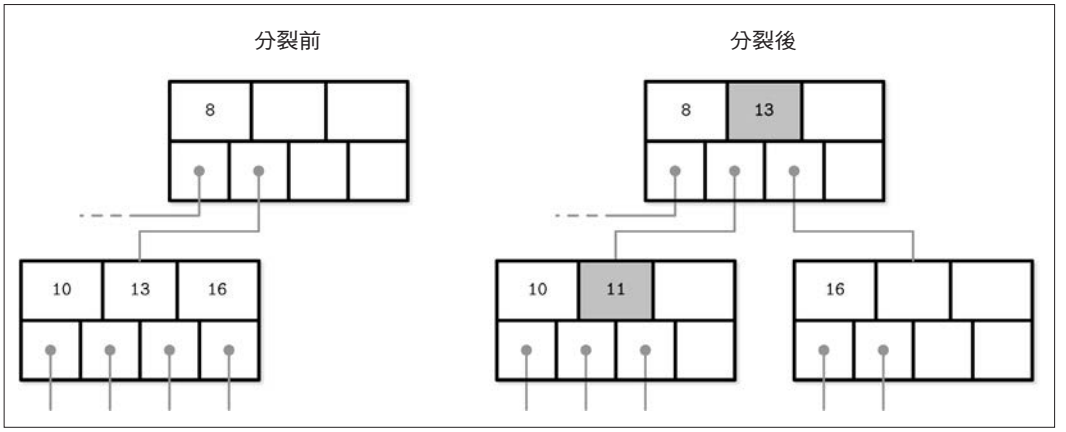

圖 2-12　插入 11 時的非葉節點分裂。新元件和升級鍵以灰色顯示。

由於非葉節點分裂是從下級別傳播的分裂表現形式，因此會有一個額外指標，指向下一層的新建節點。如果父節點沒有足夠空間，它也必須面臨分裂。

分裂葉節點還是非葉節點並不重要，即無論節點是否包含鍵和值，還是僅包含鍵；因為在葉節點分裂的情況下，鍵與其相關聯的值會一同移動。

分裂完成後會得到兩個節點，需要選擇正確的節點來完成插入操作，可以使用分隔鍵的不變式。如果插入的鍵小於升級的鍵，將將其插入到分裂節點中；否則，就將其插入到新創建的節點中。

總結來說，節點分裂包括以下 4 個步驟：

1. 分配一個新的節點。
2. 從分裂節點中將一半的元件複製到新節點中。
3. 將新元件放入對應的節點中。
4. 在分裂節點的父節點中添加一個分隔鍵，和一個指向新節點的指標。

第二部分

分散式系統

> 分散式系統指的是，你甚至不知道存在的計算機故障，可能導致你自己的計算機會無法使用的系統。
>
> —Leslie Lamport

如果沒有分散式系統，就沒有辦法打電話、轉帳或遠距離交換資訊。我們每天都在使用分散式系統，且有時候再怎麼不想承認，任何客戶端 / 伺服器應用程式也都是分散式系統。

對於許多現代軟體系統來說，垂直（*vertical*）擴展，也就是透過在具有更多 CPU、RAM，或更快磁碟的更龐大、更快機器上運行相同的軟體以擴展，是不可行的。更龐大的機器也就更昂貴，更難更換，並且可能需要特殊維護。另一種方法是水平（*horizontally*）擴展：在多台連接在網路上的機器上運行軟體，使其成為單一的邏輯實體運作。

分散式系統在規模上可能不同，從僅有數台機器到數百台機器，且參與者的特徵也可能有所不同，從小型手持裝置或感測器裝置到高效能電腦都有可能。

資料庫系統主要在單個節點上運行的時代早已一去不復返了，大多數現代資料庫系統都有多個節點連接成集群，以增加儲存容量、提高效能和增強可用性。

儘管分散式計算中的一些理論突破並不算新，但大部分都是最近才開始實際應用。今日，許多人對這個主題越來越感興趣，更多的研究和新的發展日新月益增加中。

圖 11-1 可以看到進程 P_1 和 P_2 執行不同的操作：

- a）進程 P_2 執行的操作，在進程 P_1 執行的操作完成後開始，這兩個操作是依次（*sequential*）的。
- b）這兩個操作之間存在重疊，因此，這些操作是並行（*concurrent*）的。
- c）進程 P_2 執行的操作，在進程 P_1 執行的操作之後開始，並在進程 P_1 執行的操作之前完成。這些操作也是並行（*concurrent*）的。

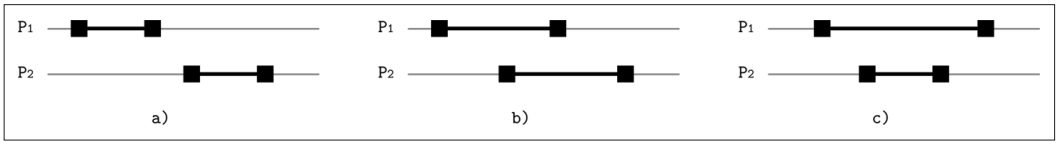

圖 11-1　順序和並行操作

多個讀取器或寫入器可以同時存取暫存器。對暫存器的讀取和寫入操作不是即時（*not immediate*）的，需要一些時間。不同進程執行的並行讀 / 寫操作不是串列（*serial*）的：根據操作重疊時暫存器的行為方式，它們的順序可能不同，並且可能產生不同的結果。根據暫存器在存在並行操作時的行為方式，可區分 3 種類型的暫存器：

安全（*Safe*）

在並行寫入操作期間，對安全暫存器的讀取可能會返回暫存器範圍內的任意（*arbitrary*）值，這聽起來不是很實用，但可能描述了不強制順序的異步系統語義。對於在寫入同時讀取的情況，具有二進制值的安全暫存器可能會顯示閃爍（*flickering*），即交替返回兩個值之間的結果。

常規（*Regular*）

對於常規暫存器有稍強保證：讀取操作只能返回最近完成（*completed*）寫入的值，或與目前讀取重疊的寫入操作的值。在這種情況下，系統具有一些順序概念，但寫入結果不會立即對所有讀取器可見；例如，這可能發生在複製的資料庫中，其中主節點接受寫入，並將其複製到提供讀取的工作節點。

原子（*Atomic*）

原子暫存器保證線性化：每個寫入操作都有一個時刻，在此之前每個讀取操作都返回一個舊值，之後每個讀取操作都返回一個新值。原子性是簡化系統狀態推理的基本屬性。

排序

當我們看到一系列事件時，對它們的執行順序會有一些直覺，然而，這在分散式系統中並不總是一件容易的事，因為很難*確切*知道何時發生了什麼事，也很難在整個叢集中立即提供這些資訊。每個參與者可能都有自己的狀態觀點，因此必須查看每個操作，並根據其*調用*（*invocation*）和*完成*（*completion*）事件對其定義，並描述操作界限。

讓我們定義一個系統，其中進程可以對共享暫存器執行 `read(register)` 和 `write(register, value)` 操作。每個進程按順序執行自己的一組操作，即每個調用的操作必須先完成，之後才能啟動下一個操作。順序進程執行的組合會形成一個總體歷史紀錄，其中操作可以同時執行。

考慮一致性模型的最簡單方法，是根據讀取和寫入操作以及它們可以重疊的方式：讀取操作沒有副作用，而寫入操作會更改暫存器的狀態。這有助於推斷寫入後資料可讀的時機。舉例來說，考慮兩個進程同時執行以下事件的歷史紀錄：

```
Process 1:      Process 2:
write(x, 1)     read(x)
                read(x)
```

在查看這些事件時，不清楚在這兩種情況下 `read(x)` 操作的結果為何，有幾個可能的歷史：

- 寫入在兩個讀取之前完成。
- 寫入和兩個讀取可以交錯進行，並且可以在兩個讀取之間執行。
- 兩個讀取在寫入之前完成。

如果只有一個資料副本，對於會發生的情況並沒有簡單的答案。在複製系統中有更多可能的狀態組合，當有多個進程讀取和寫入資料時，情況會變得更加複雜。

如果所有這些操作都由單個進程執行，可以強制執行嚴格的事件順序，但對於多個進程來說，這樣做可能會更加困難。可以將潛在的困難分為兩組：

- 操作可能重疊。
- 非重疊呼叫的效果可能不會立即顯現。

為了推理操作順序，並明確描述可能的結果，必須定義一致性模型。從共享記憶體和並行系統的角度討論分散式系統中的並行性，因為大多數定義和定義一致性的規則仍然適用。儘管並行系統和分散式系統之間的許多術語重疊，但由於通訊模式、效能和可靠性存在差異，所以無法直接應用大多數並行演算法。

一致性模型

由於共享記憶體暫存器上的操作允許重疊，我們應該定義清晰的語義：如果多個客戶端同時或在短時間內讀取或修改不同的資料副本，會發生什麼情況。這個問題沒有單一的正確答案，因為這些語義因應用程式而異，但有很多關於它在一致性模型的上下文研究。

一致性模型（*consistency models*）提供不同的語義和保證。你可以將一致性模型視為參與者之間的合約：每個副本必須遵守哪些規則，才能滿足所需的語義，以及使用者在發出讀取和寫入操作時的期望內容。

一致性模型描述了客戶端可能在存在多份資料副本和並行存取的情況下，對可能返回的值有哪些期望。本節將討論單操作（*single-operation*）一致性模型。

每個模型都描述了系統的行為，與我們可能期望或認為自然行為之間的差距，它幫助我們區分交錯操作的「所有可能的歷史」和「模型 X 下允許的歷史」，這能有效簡化關於狀態變化可見性的推理。

可以從狀態（*state*）角度思考一致性，描述哪些狀態不變式是可接受的，並建立放置在不同副本上的資料之間的允許關係。另一方面，也可以考慮操作（*operation*）的一致性，它提供資料儲存的外部視圖，描述操作並對操作發生的順序施加約束【TANENBAUM06】【AGUILERA16】。

在缺乏總體時鐘的情況下，很難替分散式操作提供精確且確定的順序。這就像資料的狹義相對論：每個參與者對狀態和時間都有自己的看法。

理論上，每次想要更改系統狀態時，都可以獲取系統範圍內的鎖定，但這是非常不切實際的；相反地，可以使用一組規則、定義和限制，來約束可能的歷史紀錄和結果的數量。

一致性模型為第 214 頁的「惡名昭彰的 CAP」討論內容，增加了另一個層面。現在，不僅要兼顧一致性和可用性，還要考慮同步成本方面的一致性【ATTIYA94】。同步成本可能包括延遲、執行額外操作所花費的額外 CPU 週期、用於保存恢復資訊的磁碟輸入輸出、等待時間、網路輸入輸出，以及透過避免同步可以防止的其他情況。

首先將關注操作結果的可見性和傳播。回到並行讀取和寫入的範例，可以透過將相依的寫入操作放在一起，或者定義一個新值傳播的時間點，來限制可能的歷史紀錄數量。

從針對資料庫狀態發出 `read` 和 `write` 操作的（客戶端）進程角度，討論一致性模型。由於在複製資料的上下文中討論一致性，因此假設資料庫可以有多個副本。

嚴格一致性

嚴格一致性（*strict consistency*）等同於完全的複製透明度：任何進程的任何寫入都可以立即供任何進程的後續讀取使用。它涉及總體時鐘的概念，如果在瞬時 t_1 有 `write(x, 1)` 操作，則任何在瞬時 $t_2 > t_1$ 的 `read(x)` 操作，都將返回新寫入的值 `1`。

不幸的是，這只是一個理論模型，不可能實施，因為物理定律和分散式系統的工作方式，會限制事情發生的速度【SINHA97】。

線性一致性

線性一致性（*linearizability*）是最強的單物件、單操作一致性模型。在這個模型下，寫入操作的效果在其開始和結束之間的某個時間點，對所有讀取器都可見正好一次，並且沒有客戶端可以觀察到部分，即未完成、仍在進行中；或不完整，即在完成之前中斷寫入操作的狀態轉換或副作用【LEE15】。

並行操作可表示為可能的序列歷史紀錄之一，其中可見性特性成立。線性一致性存在一些不確定性，因為可能存在多種事件排序的方式【HERLIHY90】。

如果兩個操作重疊，它們可以按任意順序生效，寫入操作完成後發生的所有讀取操作，都可以觀察到此操作的效果。一旦單個讀取操作返回特定值，所有在其之後的讀取操作都會返回至少與它返回的值，一樣新的值【BAILIS14a】。

在總體歷史紀錄中並行事件發生的順序方面存在一定的靈活性，但不能任意重新排序。操作結果不應在操作開始之前生效，因為這需要能夠預測未來操作的預言機器；同時，結果必須在完成之前生效，否則無法定義線性化點。

線性一致性既尊重順序的進程本地操作順序，也尊重相對於其他進程平行運行的操作順序，並定義事件的總順序（*total order*）。

此順序應該是一致（*consistent*）的，這意味著對共享值的每次讀取，都應返回在該讀取之前寫入此共享變數的最新值，或者返回與該讀取重疊的寫入值。對共享變數的線性化寫入存取也意味著互斥：在兩個並行寫入之間，只有一個可以先執行。

儘管操作是並行的，而且有一些重疊，但它們的效果以一種使它們看起來是連續的方式而可見。沒有操作是瞬間發生的，但它們仍然看起來像是原子的。

可考慮以下歷史：

```
Process 1:      Process 2:      Process 3:
write(x, 1)     write(x, 2)     read(x)
                                read(x)
                                read(x)
```

圖 11-2 有 3 個進程，其中兩個對暫存器 x 執行寫入操作，該暫存器的初始值為 ∅。讀取操作可以按以下方式之一，觀察這些寫入操作：

- a）第一個讀取操作可以返回 1、2 或 ∅，即初始值，在兩個寫入操作之前的狀態，因為兩個寫入操作仍在進行中。第一個讀取操作可以在兩個寫入操作之前、第一個和第二個寫入操作之間，以及兩個寫入操作之後排序。
- b）第二個讀取操作只能返回 1 和 2，因為第一個寫入操作已經完成，但第二個寫入操作還未返回。
- c）第三個讀取操作只能返回 2，因為第二個寫入操作是在第一個寫入操作之後排序的。

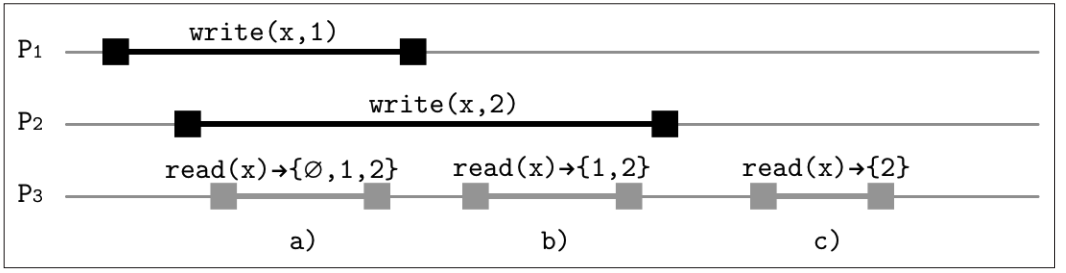

圖 11-2　線性一致性範例