Introduction to LevelDB

Hung-Chang Hsiao
Professor
Dept. Computer Science and Information Engineering
National Cheng Kung University, Tainan, Taiwan

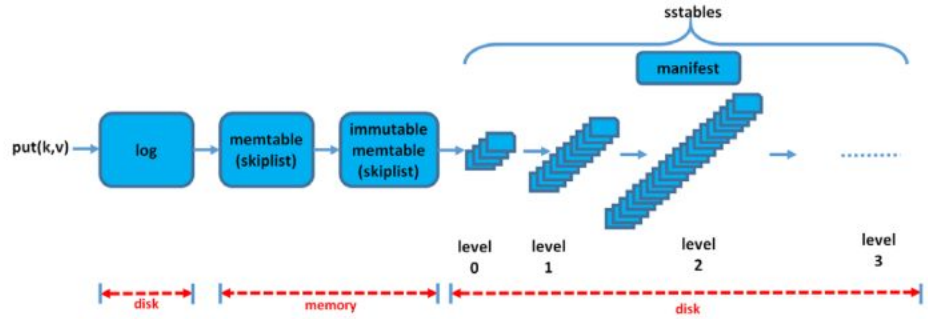
LevelDB

- A key-value store
- Presented as a "library"
- Developed by Google
- Default DB in Google Chrome
- A good entry point to learn DB internals (50k+ lines source code in C)
- Extended by Facebook, RocksDB

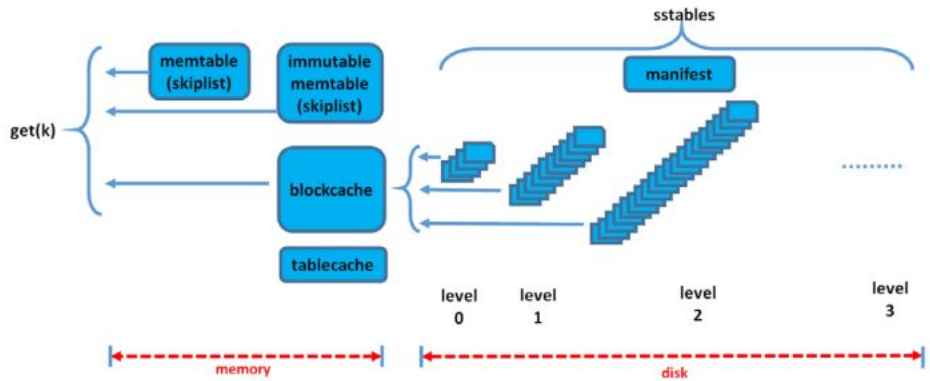
LevelDB 為一個 key-value storage engine，即每筆存在 LevelDB 之資料被表為 key 與 value pair，key (二元字串) 值必須唯一，而對應的 value 值則為真正資料的內容值，可以是任何的資料型態，如整數、字串、等等。一個資料庫表格可能有數個列與欄位 (比如 r 列且每一列有 c 個欄位)，當其被儲存在 LevelDB 中時，第 m 列、第 n 欄的 key 值為 m 與 n 的前後字串串連組合，而 value 則為該表格裡第 m 列、第 n 欄的 cell 值，即 (key,value) = (m concatenates n, (m,n)'s cell value in the table)，這裡 m concatenates n 我們記為 m⊕n。LevelDB 目前不支援多張資料庫表格操作，在 LevelDB 裡實體上僅有一張表格，因此若欲支援多張表格，則 key 值必須額外增加對表格名稱代號之編碼，比方上述的表格名稱代號若為 t，則 key 值可編碼成 t concatenates m and then concatenates n，即 t⊕m⊕n。上述並非將使用者表格儲存在 LevelDB 的唯一方式，LevelDB 得介接 SQLite compiler，SQLite compiler 提供與任何 storage engine 的接口實現介面，端視該些介面實現之方式，上述是一種常見的作法。我們強調： LevelDB 單純是一 storage engine，它並沒有內建 SQL compiler，SQL 語言的提供必須另外研製。

應用端操作 LevelDB 不是透過 SQL compiler 就是直接經由 LevelDB 所提供之 storage engine 底層 APIs 來操作。LevelDB 的 storage engine APIs 大體可分為兩大類：GET(key) 及 PUT(key,value)。GET 為讀取操作，指定 key 值並回傳對應該 key 值儲存在 LevelDB 的 value；而 PUT 則將 key 與 value pair 寫入 LevelDB。應用端若對效能有某種程度要求，則我們建議使用底層 storage engine APIs 來操作、駕馭 LevelDB (前題是對 LevelDB 內部運作有相當程度的熟悉)。事實上 SQL compiler，如 SQLite，也是 LevelDB 的一種應用端，SQL compiler 負責編譯出一系列將 SQL 語言翻譯成 LevelDB 底層操作之 storage engine APIs。

LevelDB 的讀、寫流程如 Fig. 3 所示：



(a)



(b)

Fig. 3: LevelDB 的讀寫流程，(a)寫及(b)讀

- **資料寫入流程**：如 Fig. 3(a) 所示，使用者透過 PUT(key) 將資料每次一筆寫入 (或指定一批資料整批寫入)，寫入的動作先記錄在 log 裡，log 為檔案系統裡的檔案，得視應用程式需求啟用或關閉，啟用時若 LevelDB 毀損且當時暫存於記憶體之資料未能寫入磁碟，則透過 log 可回復該些未能寫入檔案系統之對應寫入動作下的資料。資料寫入 log 後，接著會將使用者指定的 key-value 內容暫存於記憶體，即 memtable。LevelDB 會預先配置一記憶體空間並自我管理該空間，memtable 是 LevelDB 所使用記憶體空間下的基本組成單位，memtable 使用完畢後 LevelDB 回收至自己管理的記憶體空間，使得以降低 LevelDB 與作業系統在空間使用與歸還上耗費的 system call overhead。Memtable 的核心為 skiplist (稍後說明)。memtable 在 LevelDB 中當累積一定資料量大小時，會被標示為 immutable memtable，LevelDB 透過另一背景程序將 immutable memtable 刷寫至檔案系統。LevelDB 將寫入檔案系統的檔案稱為 sorted string tables (SStables)。一 SStable 紀錄連續 keys 值範圍內之資料，該些 key-value pairs 以 keys 值遞增的順序被記錄在 SStable 檔案裡。SStables 被分類成數群，分別稱為 Level-0, Level-1, Level-2,…,Level-k，其中除了在 Level-0 的 SStables，其餘任一給定在 Level-i (i=1,2,3,…) 裡倆倆 SStables 間所儲存的資料其 keys 值範圍不相重疊。Level-0 中的相異 SStables 對應管理的 keys 值則允許重疊，此乃 LevelDB 的設計上強調快

速寫入資料所致，即當記憶體中有 immutable memtable 產生，則一 immutable memtable 能快速被寫入檔案系統並形成一個獨立的 SStable 檔案儲存在 Level-0 分類裡。SStables 的內部結構攸關我們本計畫第二年的執行規劃，稍後說明。Level-0 的 SStables 檔案在 LevelDB 預設數量為四 (可調整)，而 Level-1, Level-2, Level-3,…則分別預設為 10, 100, 1000,…個檔案數。Level-0, Level-1, Level-2,…內檔案的組織實現了 log-structured merge-tree (LSM) [23]。當 Level-0 的 sstables 檔案數超過預定最大個數 (比如四個)，則當中的一個 SStable (令為 m)會被移置 Level-i，若 Level-1, Level-2, Level-3,…, Level i-1 裡所有的 SStables 不存在與 m 的 keys 值範圍發生重疊的話；也就是第一個發現與 m 所管理 keys 值重疊的 SStables 是出現在 Level-i。那些 Level-i 裡若干個與 m 的 keys 值範圍重疊的 SStables 則會被 LevelDB 讀進記憶體並與 m 合併並對 keys 值排序 (LevelDB 稱此為 compaction)。在合併的同時，若發現有些 key-value pairs 需要被移除，則此刻可能會被刪除 (是否移除與使用者對 LevelDB snapshot 操作有關，稍後討論)。刪除資料的動作在 LevelDB 裡以特別的 PUT 命令來標示，因此這些 SSTables 合併後須能反映刪除資料後的結果。合併完儲存在記憶體裡的資料則寫入 Level-i，寫入後需移除在 Level-i 那些當時參與合併的 SSTables，當然 Level-0 的 m 因與 Level-i 合併了，m 也從 Level-0 移除，使 Level-0 能持續接受來自記憶體的 immutable memtable 所生成的檔案。在記憶體那些合併完待寫入 Level-i 的資料可能會被分割成數個 SStables，切割的依據是需要參考 Level-(i+1) 層的 SStables，假設記憶體之合併資料被切割成若干個 SStables，則這些 SStables 的每一個不得與 Level-(i+1) 的 SStables 發生超過某個預先定義的重疊數量 (即，Level-(i+1) SStables 檔案與任一給定 Level-i 的 SStable 它們之間 keys 值範圍發生重疊的檔案個數)，換言之它日若 Level-i 的 SStables 因個數太多需下移合併至 Level-(i+1)，則屆時合併所花費磁碟 IO 頻寬的最大上限便能被控制，此能有效管理磁碟頻寬資源，因該些資源必須能保留一部分給 immutable memtables 寫入 Level-0 並供存使用者讀取 (透過 GET 操作) 檔案系統裡 SStables 使用。我們強調 LevelDB 實務並不會一次讀取所有在 Level-i 的檔案至記憶體中，compaction 動作在讀取 Level-i 的 SStables 與 m 合併是 pipeline 發生的，即 m 裡的一部分先與幾個對應 keys 值範圍重疊的 Level-i 的 SStables 合併，m 其餘的部分則稍後再與那些未被合併的 SStables 進行合併，反覆直到 m 的所有資料被寫入 Level-i。不難發現，LevelDB 的 LSM 中的每筆資料可能存在數個版本 (給定某個 key 其 values 反覆被使用者更新) 在相異的 Levels 裡 [24]，compaction 執行的越頻繁，則相同 key 的 values 版本數目則越少，有助於將來讀取動作的加速，但也耗費更多磁碟頻寬整理 SStables；這也會降地了新增資料能寫入磁碟的速度。讀與寫同時發生時如何優化須視應用程式存取 LevelDB 的行為而定。LevelDB 觸發 compaction 的時機亦會考慮每個 SStables 檔案被存取的次數，惟此不在我們計畫的討論範圍內，故不在此贅述。

- **讀取資料流程**：當接收自使用者的 GET(key) 命令，因最新進的資料儲存在 memtable，LevelDB 會先查找該處是否存在符合指定 key 的資料 (見 Fig. 3(b))。若無，則下一個被查找的對象為 immutable memtable。若仍無，則 LevelDB 查找 blockcache。Blockcache 為 SStables 的檔案區塊 blocks 快取。一 SStable 當中儲存 key-value 的區域會被分割成數個區塊 (data blocks)，比方一個區塊大小為 4 Kbytes，若一區塊最近過去曾被讀取，則該區塊會被暫存在記憶體預先規劃好的 blockcache 區域等待為後續可能的讀取提供資料。Blockcache 得視應用程式存取行為選擇關閉以讓出更多的記憶體空間供 memtable 與 immutable memtable 使用。一般 blockcache 使用 LRU 策略來儲存或替換檔案 blocks。若 blockcache 不能提供使用者的資料，則 LevelDB 只好尋求儲存在檔案系統裡的資料。如上述，一筆資料其相異的本版可能坐落在不同的 level 中的 SStables。比方一筆資料的兩個版本出現在 Level-i 與 Level-k，則 LevelDB 將以 Level-i 中的資料來回覆使用者的查找，若 i<k。這裡 LevelDB 在檔案系統紀錄了一個稱為 manifest 的 metadata 檔案，其中紀錄了所有 levels 之中所有的 SStables 個別管轄的 keys 值範圍。透過 manifest，LevelDB 得以快速挑選出哪些 levels 中的 SStables 涵蓋了此次使用者查找資料的 keys 值範圍。當設定欲查找 SStables 的對

象（候選查找對象），為了進一步降低緊接著存取該些候選 SStables IO 所需磁碟頻寬，每一個 SStable 裡會額外儲存若干個 bloom filters [25]。Bloom filters 為一種機率型資料結構，是一個二元字串，得用來記錄一個集合裡的元素。該些 bloom filters 彙整所屬 SStable 檔案中的 blocks 裡的 keys 值。由於 bloom filters 存在偽陽 (false positive) 的特性，因此當 bloom filter 指出 blocks 裡不存在要查找的 keys 值時，則該些 blocks 一定不存在對應 keys 值的 values。反之，若 bloom filter 指出當中某些 blocks "可能" 存在使用者指定的 keys，則 LevelDB 僅能進一步掃描該檔案的內容使確定是否存在欲查找的資料。LevelDB 透過夾帶在 SStable 中的 bloom filters 進一步快速從候選 SStables 排除不可能含有使用者欲查找的資料，使進一步縮小了候選 SStables 的集合。我們強調一 SStable 並非是 immutable memtable 記憶體內容複製。一 SStable 儲存的是 immutable memtable 對 keys 值排序後的 key-value pairs。然在儲存該些排序過的 key-value pairs 時，LevelDB 還額外對 keys 值進行編碼，使壓縮 keys 值所佔的磁碟空間。概念上每連續幾對 key-value pairs (預設三對) 進行一次編碼，除了第一個 key-value pair 的 key 被完整表示之外，其餘兩對的 key-value pairs 的 keys 值則以與第一個 key 值的差異來表示。一 SStable 除了 bloom filters 這組 metadata，還額外儲存的是對 SStable 內的 key 值索引 (in-disk indexing)，基本上該索引是用來快速定位被尋找的 key-value pairs 可能存在檔案裡的位置 (i.e., offset)。有了 in-disk indexing，LevelDB 在一 SStable 裡尋找指定搜尋的 key-value pair 時得不用大範圍對檔案進行掃描，然整體上來說讀取動作仍可能帶來磁碟讀寫頭不斷地在非連續磁碟範圍內移動，即需存取多個 SStables，且每個 SStable 會被參照儲存於其中的 bloom filters 及 in-disk indexing 等 metadata。對此，LevelDB 會儘可能將 manifest 以及每個 SStables 的 in-disk indexing 及 bloom filter 暫存於 Fig. 3(b)的 tablecache 中。最後，SStables 檔案又會透過 Google Snappy 壓縮儲存 [26]。

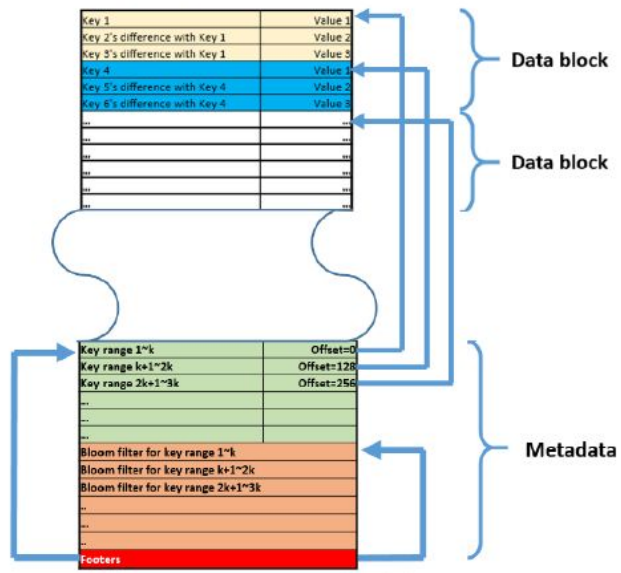


Fig. 4: SStable 檔案結構與內容

Fig. 4 為一 SStable 檔案之內部結構。SStable 在 LevelDB 的形成過程是透過檔案的 append 操作，即 pipeline 從 data blocks 一一輸出至 SStable 檔案裡，使降低記憶體使用量，接著再一次性輸出 in-disk indexing，最後才 bloom filters 與一些 footers、統計量等資訊於檔尾。File append 操作能有效開發磁碟

系統循序寫入資料的頻寬，此種輸出方式影響了我們第二年計畫的執行方式，也是我們設計平行產生 SStables 時必需考量的因子。

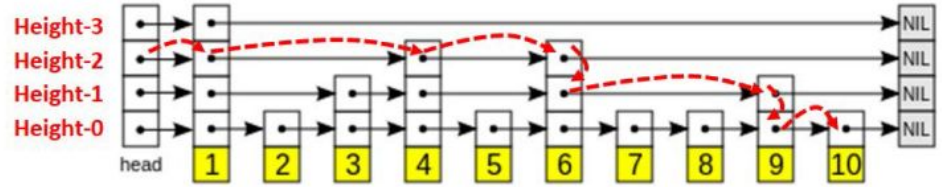


Fig. 5: Skiplist 資料結構，及查找資料 10 的走訪步驟

Fig. 5 為 LevelDB 在 memtable 及 immutable memtable 所採用之資料結構 skiplist [12]。圖中的資料數值有 1,2,3,…,10 共十筆。這個 skiplist 有四個 linked lists，分別管理高度 Height-0, Height-1, Height-2 及 Height-3 的資料。比方 Height-2 的資料集為 1,4 與 6 三個資料元素，該些元素以遞增的方式鏈結。所有的元素則連接在 Height-0 的 linked list。Height-0 的 linked list 有助於 immutable table 資料以是上述 pipeline 方式輸出至 SStables 檔案。每一筆在 skiplist 的資料能“獨立”決定自己在 skiplist 裡的高度。一筆資料高度的決定是透過機率計算，比方資料 4 的高度為 2 若其連續投擲二次公平的銅板皆出現正面的話，即 4 以 1/4 的機率出現在 Height-2 (一般我們會使用正反面出現機率各為 1/2 的銅板，但亦可使用非公平的銅板)。4 得以出現在 Height-2 則必也需存在於 Height-1。換言之，每一筆資料會分別以機率 1, 1/p, (1/p)^2,…的機率存在於 Height-0, Height-1, Height-2,…等 linked lists 裡 (p=2 若為公平的銅板)。若一筆資料出現在 Height-i，則 skiplist 必須同時保證如下 invariant：該筆資料也需同時出現在 Height-j，j=0,1,2,3,…,i-1。不難觀察，Height-i 的 linked list 內的資料元素個數的期望值占整體元素的 (1/p)^i。在 skiplist 裡查找資料一筆資料 k，比如 k=10 in Fig. 5，則查找的原則為盡可能快速遞增逼近 k=10，這裡的快速係指查找資料的過程使用最少的資料比較次數。所有的資料查找動作從 head 開始。以查找 k=10 為例，我們可透過 Height-2 來快速逼近 6 (經過走訪資料 1 及 4)，因相對 Height-1 及 Height-0 而言，Height-2 管理比較少的元素，該些資料元素能讓我們快速逼近要查找的目的地；接著再透過資料 6 連結 Height-1 (因 6 同時也是 Height-1 資料的成員) 來逼近至 9，透過 9 則又可連結 Height-0 直至到達目的地 10。透過簡單的機率分析，skiplist 裡查找一筆資料平均只需要 O(log n)的比較，n 為資料集的總資料筆數。插入一筆資料 k 至 skiplist，則 k 需先決定其在 skiplist 裡的高度(比方 i)，接著按上述走訪 skiplist 步驟 k 得以收集所有經過 Height-j (j=i-1,i-2,…,3,2,1)裡距離 k 最近的數值資料，k 一旦加入 skiplist 則必須與該些經過的資料連接外，k 也必須連結該些資料原本鏈結的資料。例如我們在 Fig. 5 加入一筆資料 5.5，若該筆資料的最大高度為 Height-1 (以上述銅板方式投擲來決定)，則資料 4 與 5 必須分別在 Height-1 及 Height-0 指向 5.5，而 5.5 也取代 4 及 5 分別在 Height-1 及 Height-0 裡去連結 6。Skiplist 是一個類 B-tree (B-tree like) 的資料結構 (嚴謹來說 skiplist 不是 tree)，雖然資料插入及搜尋上僅保證“期望值”O(log n) (註：B-tree 則保證其 worst case 下仍是 O(log n))，但增刪資料時不像 B-tree 可能造成樹的結構上為達到平衡而發生旋轉 (rotation) 變化，產生可觀的資料節點需修改記憶體指標的情況 (該些指標的修改是保證同步一致性，明顯影響效能)；換言之，一筆在 B-tree 裡的資料其在樹裡的深度 (depth) 受其他資料元素所影響，且可能隨著資料的增刪而有隨時改變樹狀結構的需要。B-tree 裡大量指標的單元性 (atomic) 修改其本質是交易 (transactions) 的議題，當同時有數筆資料需調整樹狀結構時，該些記憶體指標需一次性的修改犧牲了平行度，即 B-tree 較適合被運用在資料一筆接著一筆來的情境，這與我們設定大量資料整批進入系統的情境不同。與 B-tree 不同，skiplist 不涉及結構旋轉的議題，關鍵是每筆資料的高度被唯一且獨立決定，不受其他資料元素影響。這使得 skiplist 在操作上有相對高的平行度得被開發。最後 Fig. 5 中的資料數值即為 LevelDB 表示資料所用的

keys 值。

綜合來說，LevelDB 優化寫入資料的速度，特別適合於大量且長期資料寫入的應用；在 LevelDB 讀取資料若亦要講求速度，則連續 keys 值的存取情境較適合 LevelDB。