CHƯƠNG III

LƯU TRỮ VÀ CẤU TRÚC TẬP TIN
(Storage and File Structure)

MỤC ĐÍCH

Chương này trình bày các vấn đề liên quan đến vấn đề lưu trữ dữ liệu (trên lưu trữ ngoài, chủ yếu trên đĩa cứng). Việc lưu trữ dữ liệu phải được tổ chức sao cho có thể cất giữ một lượng lớn, có thể rất lớn dữ liệu nhưng quan trọng hơn cả là sự lưu trữ phải cho phép lấy lại dữ liệu cần thiết mau chóng. Các cấu trúc trợ giúp cho truy xuất nhanh dữ liệu được trình bày là: chỉ mục (indice), B^+ cây (B^+-tree), băm (hashing) ... Các thiết bị lưu trữ (đĩa) có thể bị hỏng hóc không lường trước, các kỹ thuật RAID cho ra một giải pháp hiệu quả cho vấn đề này.

YÊU CẦU

Hiểu rõ các đặc điểm của các thiết bị lưu trữ, cách tổ chức lưu trữ, truy xuất đĩa.

Hiểu rõ nguyên lý và kỹ thuật của tổ chức hệ thống đĩa RAID

Hiểu rõ các kỹ thuật tổ chức các mẩu tin trong file

Hiểu rõ các kỹ thuật tổ chức file

Hiểu và vận dụng các kỹ thuật hỗ trợ tìm lại nhanh thông tin: chỉ mục (được sắp, B^+-cây, băm)

KHÁI QUÁT VỀ PHƯƠNG TIỆN LƯU TRỮ VẬT LÝ

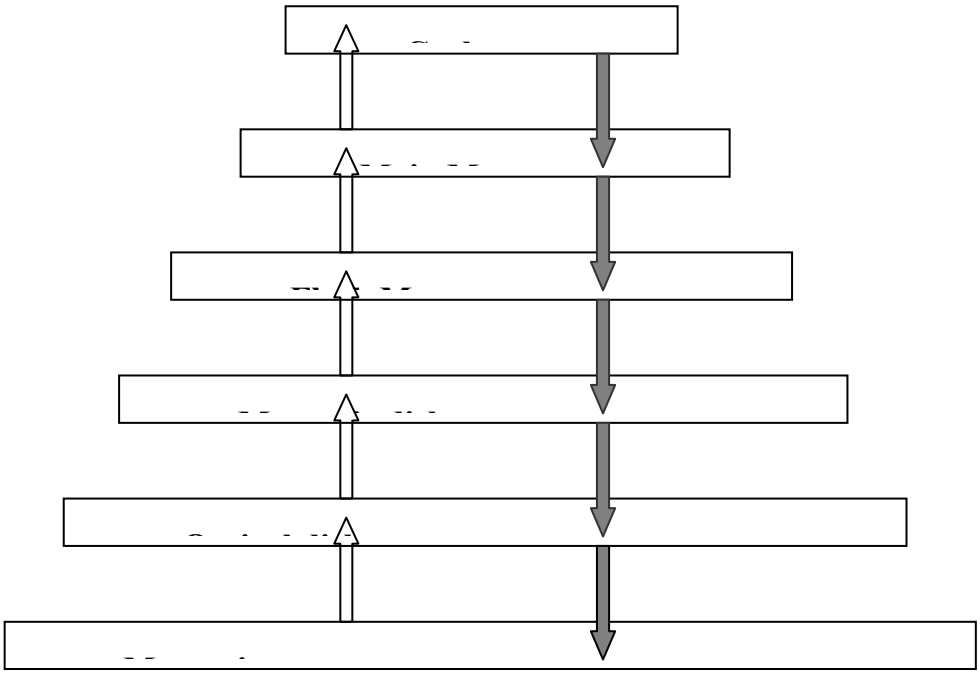
Có một số kiểu lưu trữ dữ liệu trong các hệ thống máy tính. Các phương tiện lưu trữ được phân lớp theo *tốc độ truy xuất, theo giá cả và theo độ tin cậy của phương tiện*. Các phương tiện hiện có là:

- **Cache**: là dạng lưu trữ nhanh nhất và cũng đắt nhất trong các phương tiện lưu trữ. Bộ nhớ cache nhỏ; sự sử dụng nó được quản trị bởi hệ điều hành
- **Bộ nhớ chính (main memory)**: Phương tiện lưu trữ dùng để lưu trữ dữ liệu sẵn sàng được thực hiện. Các chỉ thị máy mục đích chung (general-purpose) hoạt động trên bộ nhớ chính. Mặc dầu bộ nhớ chính có thể chứa nhiều megabytes dữ liệu, nó vẫn là quá nhỏ (và quá đắt giá) để lưu trữ toàn bộ một cơ sở dữ liệu. Nội dung trong bộ nhớ chính thường bị mất khi mất cấp nguồn
- **Bộ nhớ Flash:** Được biết như bộ nhớ chỉ đọc có thể lập trình, có thể xoá (EEPROM: Electrically Erasable Programmable Read-Only Memory), Bộ nhớ Flash khác bộ nhớ chính ở chỗ dữ liệu còn tồn tại trong bộ nhớ flash khi mất cấp nguồn. Đọc dữ liệu từ bộ nhớ flash mất ít hơn 100 ns , nhanh như đọc dữ liệu từ bộ nhớ chính. Tuy nhiên, viết dữ liệu vào bộ nhớ flash phức tạp hơn nhiều. Dữ liệu được viết (một lần mất khoảng 4 đến 10 μs) nhưng không thể viết đè trực tiếp. Để viết đè bộ nhớ đã được viết, ta phải xoá trắng toàn bộ bộ nhớ sau đó mới có thể viết lên nó.
- **Lưu trữ đĩa từ (magnetic-disk): (**ở đây, được hiểu là đĩa cứng) Phương tiện căn bản để lưu trữ dữ liệu trực tuyến, lâu dài. Thường toàn bộ cơ sở dữ liệu được lưu trữ trên đĩa từ. Dữ liệu phải được chuyển từ đĩa vào bộ nhớ chính trước khi được truy nhập. Khi dữ liệu trong bộ nhớ chính này bị sửa đổi, nó phải được viết lên đĩa. Lưu trữ đĩa được xem là truy xuất trực tiếp vì có thể đọc dữ liệu trên đĩa theo một thứ tự bất kỳ. Lưu trữ đĩa vẫn tồn tại khi mất cấp nguồn. Lưu trữ đĩa có thể bị hỏng hóc, tuy không thường xuyên.
- **Lưu trữ quang (Optical storage):** Dạng quen thuộc nhất của đĩa quang học là loại đĩa **CD-ROM** : Compact-Disk Read-Only Memory. Dữ liệu được lưu trữ trên các đĩa quang học được đọc bởi laser. Các đĩa quang học CD-ROM chỉ có thể đọc. Các phiên bản khác của chúng là loại đĩa quang học: viết một lần, đọc nhiều lần (write-once, read-many: **WORM**) cho phép viết dữ liệu lên đĩa một lần, không cho phép xoá và viết lại, và các đĩa có thể viết lại (rewritable) v..v
- **Lưu trữ băng từ (tape storage):** Lưu trữ băng từ thường dùng để backup dữ liệu. Băng từ rẻ hơn đĩa, truy xuất dữ liệu chậm hơn (vì phải truy xuất tuần tự). Băng từ thường có dung lượng rất lớn.

Các phương tiện lưu trữ có thể được tổ chức phân cấp theo tốc độ truy xuất và giá cả. Mức cao nhất là nhanh nhất nhưng cũng là đắt nhất, giảm dần xuống các mức thấp hơn.

Các phương tiện lưu trữ nhanh (*cache, bộ nhớ chính*) được xem như là lưu trữ sơ cấp (primary storage), các thiết bị lưu trữ ở mức thấp hơn như đĩa từ được xem như lưu trữ thứ cấp hay lưu trữ trực tuyến (on-line storage), còn các thiết bị lưu trữ ở mức thấp nhất và gần thấp nhất như đĩa quang học, băng từ kể cả các đĩa mềm được xếp vào lưu trữ tam cấp hay lưu trữ không trực tuyến (off-line).

Bên cạnh vấn đề tốc độ và giá cả, ta còn phải xét đến tính lâu bền của các phương tiện lưu trữ.



Phân cấp thiết bị lưu trữ

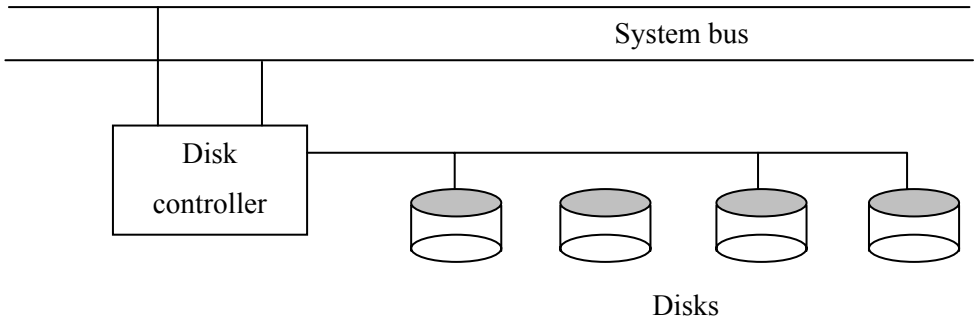
ĐĨA TỪ

ĐẶC TRƯNG VẬT LÝ CỦA ĐĨA

Mỗi tấm đĩa có dạng hình tròn, hai mặt của nó được phủ bởi vật liệu từ tính, thông tin được ghi trên bề mặt đĩa. Đĩa gồm nhiều tấm đĩa. Ta sẽ sử dụng thuật ngữ đĩa để chỉ các đĩa cứng.

Khi đĩa được sử dụng, một động cơ ổ đĩa làm quay nó ở một tốc độ không đổi. Một đầu đọc-viết được định vị trên bề mặt của tấm đĩa. Bề mặt tấm đĩa được chia logic thành các rãnh, mỗi rãnh lại được chia thành các sector, một sector là một đơn vị thông tin nhỏ có thể được đọc, viết lên đĩa. Tuỳ thuộc vào kiểu đĩa, sector thay đổi từ 32 bytes đến 4095 bytes, thông thường là 512 bytes. Có từ 4 đến 32 sectors trên một rãnh, từ 20 đến 1500 rãnh trên một bề mặt. Mỗi bề mặt của một tấm đĩa có một đầu đọc viết, nó có thể chạy dọc theo bán kính đĩa để truy cập đến các rãnh khác nhau. Một đĩa gồm nhiều tấm đĩa, các đầu đọc-viết của tất cả các rãnh được gắn vào một bộ được gọi là cánh tay đĩa, di chuyển cùng nhau. Các tấm đĩa được gắn vào một trục quay. Vì các đầu đọc-viết trên các tấm đĩa di chuyển cùng nhau, nên khi đầu đọc-viết trên một tấm đĩa đang ở rãnh thứ i thì các đầu đọc-viết của các tấm đĩa khác cũng ở rãnh thứ i , do vậy các rãnh thứ i của tất cả các tấm đĩa được gọi là trụ (cylinder) thứ i . Một bộ điều khiển đĩa -- giao diện giữa hệ thống máy tính và phần cứng hiện thời của ổ đĩa. Nó chấp nhận các lệnh mức cao để đọc và viết một sector, và khởi động các hành động như di chuyển cánh tay đĩa đến các rãnh đúng và đọc viết dữ liệu. bộ điều khiển đĩa cũng tham gia vào checksum mỗi sector được viết. Checksum được tính từ dữ liệu được viết lên sector. Khi sector được đọc lại, checksum được tính lại từ dữ liệu được lấy ra và so sánh với checksum đã lưu trữ. Nếu dữ liệu bị sai lạc, checksum được tính sẽ không khớp với checksum đã lưu trữ. Nếu lỗi như vậy xảy ra, bộ điều khiển sẽ lặp lại việc đọc vài lần, nếu lỗi vẫn xảy ra, bộ điều khiển sẽ thông báo việc đọc thất bại. Bộ điều khiển đĩa còn có

chức năng tái ánh xạ các sector xấu: ánh xạ các sector xấu đến một vị trí vật lý khác. Hình dưới bày tỏ các đĩa được nối với một hệ thống máy tính:



Các đĩa được nối với một hệ thống máy tính hoặc một bộ điều khiển đĩa qua một sự hợp nhất tốc độ cao. Hợp nhất hệ thống máy tính nhỏ (Small Computer-System Interconnect: SCSI) thường được sử dụng để nối kết các đĩa với các máy tính cá nhân và workstation. Mainframe và các hệ thống server thường có các bus nhanh hơn và đắt hơn để nối với các đĩa.

Các đầu đọc-viết được giữ sát với bề mặt đĩa như có thể để tăng độ dày đặc (density).

Đĩa đầu cố định (Fixed-head) có một đầu riêng biệt cho mỗi rãnh, sự sắp xếp này cho phép máy tính chuyển từ rãnh này sang rãnh khác mau chóng, không phải di chuyển đầu đọc-viết. Tuy nhiên, cần một số rất lớn đầu đọc-viết, điều này làm nâng giá của thiết bị.

ĐO LƯỜNG HIỆU NĂNG CỦA ĐĨA

Các tiêu chuẩn đo lường chất lượng chính của đĩa là ***dung lượng, thời gian truy xuất, tốc độ truyền dữ liệu và độ tin cậy***.

- ***Thời gian truy xuất (access time)***: là khoảng thời gian từ khi yêu cầu đọc/viết được phát đi đến khi bắt đầu truyền dữ liệu. Để truy xuất dữ liệu trên một sector đã cho của một đĩa, đầu tiên cánh tay đĩa phải di chuyển đến rãnh đúng, sau đó phải chờ sector xuất hiện dưới nó, thời gian để định vị cánh tay được gọi là thời gian tìm kiếm (seek time), nó tỷ lệ với khoảng cách mà cánh tay phải di chuyển, thời gian tìm kiếm nằm trong khoảng 2..30 ms tuỳ thuộc vào rãnh xa hay gần vị trí cánh tay hiện tại.
- ***Thời gian tìm kiếm trung bình (average seek time)***: Thời gian tìm kiếm trung bình là trung bình của thời gian tìm kiếm, được đo luờng trên một dãy các yêu cầu ngẫu nhiên (phân phối đều), và bằng khoảng 1/3 thời gian tìm kiếm trong trường hợp xấu nhất.
- ***Thời gian tiềm ẩn luân chuyển (rotational latency time)***: Thời gian chờ sector được truy xuất xuất hiện dưới đầu đọc/viết. Tốc độ quay của đĩa nằm trong khoảng 60..120 vòng quay trên giây, trung bình cần nửa vòng quay để sector cần thiết nằm dưới đầu đọc/viết. Như vậy, thời gian tiềm ẩn trung bình (average latency time) bằng nửa thời gian quay một vòng đĩa.

Thời gian truy xuất bằng tổng của thời gian tìm kiếm và thời gian tiềm ẩn và nằm trong khoảng 10..40 ms.

- ***Tốc độ truyền dữ liệu:*** là tốc độ dữ liệu có thể được lấy ra từ đĩa hoặc được lưu trữ vào đĩa. Hiện nay tốc này vào khoảng1..5 Mbps

- ***Thời gian trung bình không sự cố (mean time to failure):*** lượng thời gian trung bình hệ thống chạy liên tục không có bất kỳ sự cố nào. Các đĩa hiện nay có thời gian không sự cố trung bình khoảng 30000 .. 800000 giờ nghĩa là khoảng từ 3,4 đến 91 năm.

TỐI ƯU HÓA TRUY XUẤT KHỐI ĐĨA (disk-block)

Yêu cầu I/O đĩa được sinh ra cả bởi hệ thống file lẫn bộ quản trị bộ nhớ ảo trong hầu hết các hệ điều hành. Mỗi yêu cầu xác định địa chỉ trên đĩa được tham khảo, địa chỉ này ở dạng số khối. Một khối là một dãy các sector kề nhau trên một rãnh. Kích cỡ khối trong khoảng 512 bytes đến một vài Kbytes. Dữ liệu được truyền giữa đĩa và bộ nhớ chính theo đơn vị khối. Mức thấp hơn của bộ quản trị hệ thống file sẽ chuyển đổi địa chỉ khối sang số của trụ, của mặt và của sector ở mức phần cứng.

Truy xuất dữ liệu trên đĩa chậm hơn nhiều so với truy xuất dữ liệu trong bộ nhớ chính, do vậy cần thiết một chiến lược nhằm nâng cao tốc độ truy xuất khối đĩa. Dưới đây ta sẽ thảo luận một vài kỹ thuật nhằm vào mục đích đó.

- ***Scheduling:*** Nếu một vài khối của một trụ cần được truyền từ đĩa vào bộ nhớ chính, ta có thể tiết kiệm thời gian truy xuất bởi yêu cầu các khối theo thứ tự mà nó chạy qua dưới đầu đọc/viết. Nếu các khối mong muốn ở trên các trụ khác nhau, ta yêu cầu các khối theo thứ tự sao cho làm tối thiểu sự di chuyển cánh tay đĩa. Các thuật toán scheduling cánh tay đĩa (Disk-arm-scheduling) nhằm lập thứ tự truy xuất các rãnh theo cách làm tăng số truy xuất có thể được xử lý. Một thuật toán thường dùng là thuật toán thang máy (elevator algorithm): Giả sử ban đầu cánh tay di chuyển từ rãnh trong nhất hướng ra phía ngoài đĩa, đối với mỗi rãnh có yêu cầu truy xuất, nó dừng lại, phục vụ yêu cầu đối với rãnh này, sau đó tiếp tục di chuyển ra phía ngoài đến tận khi không có yêu cầu nào chờ các rãnh xa hơn phía ngoài. Tại điểm này, cánh tay đổi hướng, di chuyển vào phía trong, lại dừng lại trên các rãnh được yêu cầu, và cứ như vậy đến tận khi không còn rãnh nào ở trong hơn được yêu cầu, rồi lại đổi hướng .. v .. v .. Bộ điều khiển đĩa thường làm nhiệm vụ sắp xếp lại các yêu cầu đọc để cải tiến hiệu năng.
- ***Tổ chức file:*** Để suy giảm thời gian truy xuất khối, ta có thể tổ chức các khối trên đĩa theo cách tương ứng gần nhất với cách mà dữ liệu được truy xuất. Ví dụ, Nếu ta muốn một file được truy xuất tuần tự, khi đó ta bố trí các khối của file một cách tuần tự trên các trụ kề nhau. Tuy nhiên việc phân bố các khối lưu trữ kề nhau này sẽ bị phá vỡ trong quá trình phát triển của file $\Rightarrow$ file không thể được phân bố trên các khối kề nhau được nữa, hiện tượng này dược gọi là sự phân mảnh (fragmentation). Nhiều hệ điều hành cung cấp tiện ích giúp suy giảm sự phân mảnh này (Defragmentation) nhằm làm tăng hiệu năng truy xuất file.
- ***Các buffers viết không hay thay đổi:*** Vì nội dung của bộ nhớ chính bị mất khi mất nguồn, các thông tin về cơ sở dữ liệu cập nhật phải được ghi lên đĩa nhằm đề phòng sự cố. Hiệu năng của các ứng dụng cập nhật cường độ cao phụ thuộc mạnh vào tốc độ viết đĩa. Ta có thể sử dụng bộ nhớ truy xuất ngẫu nhiên không hay thay đổi (nonvolatile RAM) để nâng tốc độ viết đĩa. Nội dung của nonvolatile RAM không bị mất khi mất nguồn. Một phương pháp chung để thực hiện nonvolatile RAM là sử dụng RAM pin dự phòng (battery-back-up RAM). Khi cơ sở dữ liệu yêu cầu viết một khối lên đĩa, bộ điều khiển dĩa viết khối này lên buffer nonvolatile RAM, và thông báo ngay cho hệ điều hành là việc viết đã thành công. Bộ điều khiển sẽ viết dữ liệu đến đích của nó trên đĩa, mỗi khi đĩa rảnh hoặc buffer nonvolatile RAM đầy. Khi hệ cơ sở dữ liệu yêu cầu một viết khối, nó chỉ chịu một khoảng lặng chờ đợi khi buffer nonvolatile RAM đầy.

- ***Đĩa log (log disk):*** Một cách tiếp cận khác để làm suy giảm tiềm năng viết là sử dụng log-disk: Một đĩa được tận hiến cho việc viết một log tuần tự. Tất cả các truy xuất đến log-disk là tuần tự, nhằm loại bỏ thời gian tìm kiếm, và một vài khối kề có thể được viết một lần, tạo cho viết vào log-disk nhanh hơn viết ngẫu nhiên vài lần. Cũng như trong trường hợp sử dụng nonvolatile RAM, dữ liệu phải được viết vào vị trí hiện thời của chúng trên đĩa, nhưng việc viết này có thể được tiến hành mà hệ cơ sở dữ liệu không cần thiết phải chờ nó hoàn tất. Log-disk có thể được sử dụng để khôi phục dữ liệu. Hệ thống file dựa trên log là một phiên bản của cách tiếp cận log-disk: Dữ liệu không được viết lại lên đích gốc của nó trên đĩa; thay vào đó, hệ thống file lưu vết nơi các khối được viết mới đây nhất trên log-disk, và hoàn lại chúng từ vị trí này. Log-disk được "cô đặc" lại (compacting) theo một định kỳ. Cách tiếp cận này cải tiến hiệu năng viết, song sinh ra sự phân mảnh đối với các file được cập nhật thường xuyên.

RAID

Trong một hệ thống có nhiều đĩa, ta có thể cải tiến tốc độ đọc viết dữ liệu nếu cho chúng hoạt động song song. Mặt khác, hệ thống nhiều đĩa còn giúp tăng độ tin cậy lưu trữ bằng cách lưu trữ dư thừa thông tin trên các đĩa khác nhau, nếu một đĩa có sự cố dữ liệu cũng không bị mất. Một sự đa dạng các kỹ thuật tổ chức đĩa, được gọi là **RAID (Redundant Arrays of Inexpensive Disks)**, được đề nghị nhằm vào vấn đề tăng cường hiệu năng và độ tin cậy.

CẢI TIẾN ĐỘ TIN CẬY THÔNG QUA SỰ DƯ THỪA

Giải pháp cho vấn đề độ tin cậy là đưa vào sự dư thừa: lưu trữ thông tin phụ, bình thường không cần thiết, nhưng nó có thể được sử dụng để tái tạo thông tin bị mất khi gặp sự cố hỏng hóc đĩa, như vậy thời gian trung bình không sự cố tăng lên (xét tổng thể trên hệ thống đĩa).

Đơn giản nhất, là làm bản sao cho mỗi đĩa. Kỹ thuật này được gọi là mirroring hay shadowing. Một đĩa logic khi đó bao gồm hai đĩa vật lý, và mỗi việc viết được thực hiện trên cả hai đĩa. Nếu một đĩa bị hư, dữ liệu có thể được đọc từ đĩa kia. Thời gian trung bình không sự cố của đĩa mirror phụ thuộc vào thời gian trung bình không sự cố của mỗi đĩa và phụ thuộc vào thời gian trung bình được sửa chữa (mean time to repair): thời gian trung bình để một đĩa bị hư được thay thế và phục hồi dữ liệu trên nó.

CẢI TIẾN HIỆU NĂNG THÔNG QUA SONG SONG

Với đĩa mirror, tốc độ đọc có thể tăng lên gấp đôi vì yêu cầu đọc có thể được gửi đến cả hai đĩa. Với nhiều đĩa, ta có thể cải tiến tốc độ truyền bởi phân nhỏ (striping data) dữ liệu qua nhiều đĩa. Dạng đơn giản nhất là tách các bít của một byte qua nhiều đĩa, sự phân nhỏ này được gọi là sự phân nhỏ mức bit (bit-level striping). Ví dụ, ta có một dàn 8 đĩa, ta viết bít thứ i của một byte lên đĩa thứ i . dàn 8 đĩa này có thể được xử lý như một đĩa với các sector 8 lần lớn hơn kích cỡ thông thường, quan trọng hơn là tốc dộ truy xuất tăng lên tám lần. Trong một tổ chức như vậy, mỗi đĩa tham gia vào mỗi truy xuất (đọc/viết), như vậy, số các truy xuất có thể được xử lý trong một giây là tương tự như trên một đĩa, nhưng mỗi truy xuất có thể đọc/viết nhiều dữ liệu hơn tám lần.

Phân nhỏ mức bit có thể được tổng quát cho số đĩa là bội hoặc ước của 8, Ví dụ, ta có một dàn 4 đĩa, ta sẽ phân phối bít thứ i và bít thứ 4+i vào đĩa thứ i. Hơn nữa, sự phân nhỏ không nhất thiết phải ở mức bit của một byte. Ví dụ, trong sự phân nhỏ mức khối, các khối của một file được phân nhỏ qua nhiều đĩa, với n đĩa, khối thứ i có thể được phân phối qua đĩa (i mod n) + 1. Ta cũng

có thể phân nhỏ ở mức byte, sector hoặc các sector của một khối. Hai đích song song trong một hệ thống đĩa là:

1. Nạp nhiều truy xuất nhỏ cân bằng (truy xuất trang) sao cho lượng dữ liệu được nạp trong một đơn vị thời gian của truy xuất như vậy tăng lên.
2. Song song hoá các truy xuất lớn sao cho thời gian trả lời các truy xuất lớn giảm.

CÁC MỨC RAID

Mirroring cung cấp độ tin cậy cao, nhưng đắt giá. Phân nhỏ cung cấp tốc độ truyền dữ liệu cao, nhưng không cải tiến được độ tin cậy. Nhiều sơ đồ cung cấp sự dư thừa với giá thấp bằng cách phối hợp ý tưởng của phân nhỏ với "parity" bit. Các sơ đồ này có sự thoả hiệp *giá-hiệu năng* khác nhau và được phân lớp thành các mức được gọi là các mức RAID.

• **Mức RAID 0 :** Liên quan đến các dàn đĩa với sự phân nhỏ mức khối, nhưng không có một sự dư thừa nào.

• **Mức RAID 1 :** Liên quan đến mirror đĩa

• **Mức RAID 2 :** Cũng được biết dưới cái tên mã sửa lỗi kiểu bộ nhớ (memory-style error-correcting-code : ECC). Hệ thống bộ nhớ thực hiện phát hiện lỗi bằng bit parity. Mỗi byte trong hệ thống bộ nhớ có thể có một bit parity kết hợp với nó. Sơ đồ sửa lỗi lưu hai hoặc nhiều hơn các bit phụ, và có thể dựng lại dữ liệu nếu một bit bị lỗi. ý tưởng của mã sửa lỗi có thể được sử dụng trực tiếp trong dàn đĩa thông qua phân nhỏ byte qua các đĩa. Ví dụ, bít đầu tiên của mỗi byte có thể được lưu trên đĩa 1, bit thứ hai trên đĩa 2, và cứ như vậy, bit thứ 8 trên đĩa 8, các bit sửa lỗi được lưu trên các đĩa thêm vào. Nếu một trong các đĩa bị hư, các bít còn lại của byte và các bit sửa lỗi kết hợp được đọc từ các đĩa khác có thể giúp tái tạo bít bị mất trên đĩa hư, như vậy ta có thể dựng lại dữ liệu. Với một dàn 4 đĩa dữ liệu, RAID mức 2 chỉ cần thêm 3 đĩa để lưu các bit sửa lỗi (các đĩa thêm vào này được gọi là các đĩa overhead), so sánh với RAID mức 1, cần 4 đĩa overhead.

• **Mức RAID 3 :** Còn được gọi là tổ chức parity chen bit (bit-interleaved parity). Bộ điều khiển đĩa có thể phát hiện một sector được đọc đúng hay sai, như vậy có thể sử dụng chỉ một bit parity để sửa lỗi: Nếu một trong các sector bị hư, ta biết chính xác đó là sector nào, Với mỗi bit trong sector này ta có thể hình dung nó là bít 1 hay bit 0 bằng cách tính parity của các bit tương ứng từ các sector trên các đĩa khác. Nếu parity của các bit còn lại bằng với parity được lưu, bit mất sẽ là 0, ngoài ra bit mất là 1. RAID mức 3 tốt như mức 2 nhưng it tốn kém hơn (chỉ cần một đĩa overhead).

• **Mức RAID 4 :** Còn được gọi là tổ chức parity chen khối (Block-interleaved parity), lưu trữ các khối đúng như trong các đĩa chính quy, không phân nhỏ chúng qua các đĩa nhưng lấy một khối parity trên một đĩa riêng biệt đối với các khối tương ứng từ N đĩa khác. Nếu một trong các đĩa bị hư, khối parity có thể được dùng với các khối tương ứng từ các đĩa khác để khôi phục khối của đĩa bị hư.

Một đọc khối chỉ truy xuất một đĩa, cho phép các yêu cầu khác được xử lý bởi các đĩa khác. Như vậy, tốc độ truyền dữ liệu đối với mỗi truy xuất chậm, nhưng nhiều truy xuất đọc có thể được xử lý song song, dẫn đến một tốc độ I/O tổng thể cao hơn. Tốc độ truyền đối với các đọc dữ liệu lớn (nhiều khối) cao do tất cả các đĩa có thể được đọc song song; các viết dữ liệu lớn (nhiều khối) cũng có tốc độ truyền cao vì dữ liệu và parity có thể được viết song song. Tuy nhiên, viết một khối đơn phải truy xuất đĩa trên đó khối được lưu trữ, và đĩa parity (do khối parity cũng phải được cập nhật). Như vậy, viết một khối đơn yêu cầu 4 truy xuất: hai để đọc hai khối cũ, và hai để viết lại hai khối.

- **Mức RAID 5 :** Còn gọi là parity phân bố chen khối (Block-interleaved Distributed Parity), cải tiến của mức 4 bởi phân hoạch dữ liệu và parity giữa toàn bộ N+1 đĩa, thay vì lưu trữ dữ liệu trên N đĩa và parity trên một đĩa riêng biệt như trong RAID 4. Trong RAID 5, tất cả các đĩa có thể tham gia làm thoả mãn các yêu cầu đọc, như vậy sẽ làm tăng tổng số yêu cầu có thể được đặt ra trong một đơn vị thời gian. Đối với mỗi khối, một đĩa lưu trữ parity, các đĩa khác lưu trữ dữ liệu. Ví dụ, với một dàn năm đĩa, parity đối với khối thứ n được lưu trên đĩa (n mod 5)+1. Các khối thứ n của 4 đĩa khác lưu trữ dữ liệu hiện hành của khối đó.

- **Mức RAID 6 :** Còn được gọi là sơ đồ dư thừa P+Q (P+Q redundancy scheme), nó rất giống RAID 5 nhưng lưu trữ thông tin dư thừa phụ để canh chừng nhiều đĩa bị hư. Thay vì sử dụng parity, người ta sử dụng các mã sửa lỗi.

CHỌN MỨC RAID ĐÚNG

Nếu đĩa bị hư, Thời gian tái tạo dữ liệu của nó là đáng kể và thay đổi theo mức RAID được dùng. Sự tái tạo dễ dàng nhất đối với mức RAID 1. Đối với các mức khác, ta phải truy xuất tất cả các đĩa khác trong dàn đĩa để tái tạo dữ liệu trên đĩa bị hư. Hiệu năng tái tạo của một một hệ thống RAID có thể là một nhân tố quan trọng nếu việc cung cấp dữ liệu liên tục được yêu cầu (thường xảy ra trong các hệ CSDL hiệu năng cao hoặc trao đổi). Hơn nữa, hiệu năng tái tạo ảnh hưởng đến thời gian trung bình không sự cố.

Vì RAID mức 2 và 4 được gộp lại bởi RAID mức 3 và 5, Việc lựa chọn mức RAID thu hẹp lại trên các mức RAID còn lại. Mức RAID 0 được dùng trong các ứng dụng hiệu năng cao ở đó việc mất dữ liệu không có gì là trầm trọng cả. RAID mức 1 là thông dụng cho các ứng dụng lưu trữ các log-file trong hệ CSDL. Do mức 1 có overhead cao, mức 3 và 5 thường được ưa thích hơn đối với việc lưu trữ khối lượng dữ liệu lớn. Sự khác nhau giữa mức 3 và mức 5 là tốc độ truyền dữ liệu đối lại với tốc độ I/O tổng thể. Mức 3 được ưa thích hơn nếu truyền dữ liệu cao được yêu cầu, mức 5 được ưa thích hơn nếu việc đọc ngẫu nhiên là quan trọng. Mức 6, tuy hiện nay ít được áp dụng, nhưng nó có độ tin cậy cao hơn mức 5.

MỞ RỘNG

Các quan niệm của RAID được khái quát hoá cho các thiết bị lưu trữ khác, bao hàm các dàn băng, thậm chí đối với quảng bá dữ liệu trên các hệ thống không dây. Khi áp dụng RAID cho dàn băng, cấu trúc RAID cho khả năng khôi phục dữ liệu cả khi một trong các băng bị hư hại. Khi áp dụng đối với quảng bá dữ liệu, một khối dữ liệu được phân thành các đơn vị nhỏ và được quảng bá cùng với một đơn vị parity; nếu một trong các đơn vị này không nhận được, nó có thể được dựng lại từ các đơn vị còn lại.

LƯU TRỮ TAM CẤP (tertiary storage)

ĐĨA QUANG HỌC

CR-ROM có ưu điểm là có khả năng lưu trữ lớn, dễ di chuyển (có thể đưa vào và lấy ra khỏi ổ đĩa như đĩa mềm), hơn nữa giá lại rẻ. Tuy nhiên, so với ổ đĩa cứng, thời gian tìm kiếm của ổ CD-ROM chậm hơn nhiều (khoảng 250ms), tốc độ quay chậm hơn (khoảng 400rpm), từ đó dẫn đến độ trễ cao hơn; tốc độ truyền dữ liệu cũng chậm hơn (khoảng 150Kbytes/s). Gần đây, một định dạng mới của đĩa quang học - Digital video disk (DVD) - được chuẩn hoá, các đĩa này có dung lượng trong khoảng 4,7GBytes đến 17 GBytes. Các đĩa WORM, REWRITABLE cũng trở thành phổ biến. Các WORM jukeboxes là các thiết bị có thể lưu trữ một số lớn các đĩa WORM và có thể nạp tự động các đĩa theo yêu cầu đến một hoặc một vài ổ WORM.

BĂNG TỪ

Băng từ có thể lưu một lượng lớn dữ liệu, tuy nhiên, chậm hơn so với đĩa từ và đĩa quang học. Truy xuất băng buộc phải là truy xuất tuần tự, như vậy nó không thích hợp cho hầu hết các đòi hỏi của lưu trữ thứ cấp. Băng từ được sử dụng chính cho việc backup, cho lưu trữ các thông tin không được sử dụng thường xuyên và như một phương tiện ngoại vi (off-line medium) để truyền thông tin từ một hệ thống đến một hệ thống khác. Thời gian để định vị đoạn băng lưu dữ liệu cần thiết có thể kéo dài đến hàng phút. Jukeboxes băng chứa một lượng lớn băng, với một vài ổ băng và có thể lưu trữ được nhiều TeraBytes (10^{12} Bytes)

TRUY XUẤT LƯU TRỮ

Một cơ sở dữ liệu được ánh xạ vào một số các file khác nhau được duy trì bởi hệ điều hành nền. Các file này lưu trú thường trực trên các đĩa với backup trên băng. Mỗi file được phân hoạch thành các đơn vị lưu trữ độ dài cố định được gọi là khối - đơn vị cho cả cấp phát lưu trữ và truyền dữ liệu.

Một khối có thể chứa một vài hạng mục dữ liệu (data item). Ta giả thiết không một hạng mục dữ liệu nào trải ra trên hai khối. Mục tiêu nổi trội của hệ CSDL là tối thiểu hoá số khối truyền giữa đĩa và bộ nhớ. Một cách để giảm số truy xuất đĩa là giữ nhiều khối như có thể trong bộ nhớ chính. Mục đích là để khi một khối được truy xuất, nó đã nằm sẵn trong bộ nhớ chính và như vậy không cần một truy xuất đĩa nào cả.

Do không thể lưu tất cả các khối trong bộ nhớ chính, ta cần quản trị cấp phát không gian sẵn có trong bộ nhớ chính để lưu trữ các khối. Bộ đệm (Buffer) là một phần của bộ nhớ chính sãn có để lưu trữ bản sao khối đĩa. Luôn có một bản sao trên đĩa cho mỗi khối, song các bản sao trên đĩa của các khối là các phiên bản cũ hơn so với phiên bản trong buffer. Hệ thống con đảm trách cấp phát không gian buffer được gọi là bộ quản trị buffer.

BỘ QUẢN TRỊ BUFFER

Các chương trình trong một hệ CSDL đưa ra các yêu cầu cho bộ quản trị buffer khi chúng cần một khối đĩa. Nếu khối này đã sẵn sàng trong buffer, địa chỉ khối trong bộ nhớ chính được chuyển cho người yêu cầu. Nếu khối chưa có trong buffer, bộ quản trị buffer đầu tiên cấp phát không gian trong buffer cho khối, rút ra một số khối khác, nếu cần thiết, để lấy không gian cho khối mới. Khối được rút ra chỉ được viết lại trên đĩa khi nó có bị sửa đổi kể từ lần được viết lên đĩa gần nhất. Sau đó bộ quản trị buffer đọc khối từ đĩa vào buffer, và chuyển địa chỉ của khối trong bộ nhớ chính cho người yêu cầu. Bộ quản trị buffer không khác gì nhiều so với bộ quản trị bộ nhớ ảo, một điểm khác biệt là kích cỡ của một CSDL có thể rất lớn không đủ chứa toàn bộ trong bộ nhớ chính do vậy bộ quản trị buffer phải sử dụng các kỹ thuật tinh vi hơn các sơ đồ quản trị bộ nhớ ảo kiểu mẫu.

- **Chiến lược thay thế.** Khi không có chỗ trong buffer, một khối phải được xoá khỏi buffer trước khi một khối mới được đọc vào. Thông thường, hệ điều hành sử dụng sơ đồ LRU (Least Recently Used) để viết lên đĩa khối ít được dùng gần đây nhất, xoá bỏ nó khỏi buffer. Cách tiếp cận này có thể được cải tiến đối với ứng dụng CSDL.

- **Khối chốt (pinned blocks).** Để hệ CSDL có thể khôi phục sau sự cố, cần thiết phải hạn chế thời gian khi viết lại lên đĩa một khối. Một khối không cho phép viết lại lên đĩa được gọi là khối chốt.

- **Xuất ra bắt buộc các khối (Forced output of blocks).** Có những tình huống trong đó cần phải viết lại một khối lên đĩa, cho dù không gian buffer mà nó chiếm là không cần đến. Việc

viết này được gọi là *sự xuất ra bắt buộc của một khối*. Lý do ngắn gọn của yêu cầu xuất ra bắt buộc khối là nội dung của bộ nhớ chính bị mất khi có sự cố, ngược lại dữ liệu trên dĩa còn tồn tại sau sự cố.

CÁC ĐỐI SÁCH THAY THẾ BUFFER (Buffer-Replacement Policies).

Mục đích của chiến lược thay thế khối trong buffer là tối thiểu hoá các truy xuất đĩa. Các hệ điều hành thường sử dụng chiến lược LRU để thay thế khối. Tuy nhiên, một hệ CSDL có thể dự đoán mẫu tham khảo tương lai. Yêu cầu của một người sử dụng đối với hệ CSDL bao gồm một số bước. Hệ CSDL có thể xác định trước những khối nào sẽ là cần thiết bằng cách xem xét mỗi một trong các bước được yêu cầu để thực hiện hoạt động được yêu cầu bởi người sử dụng. Như vậy, khác với hệ điều hành, hệ CSDL có thể có thông tin liên quan đến tương lai, chí ít là tương lai gần. Trong nhiều trường hợp, chiến lược thay thế khối tối ưu cho hệ CSDL lại là MRU (Most Recently Used): Khối bị thay thế sẽ là khối mới được dùng gần đây nhất!

Bộ quản trị buffer có thể sử dụng thông tin thống kê liên quan đến xác suất mà một yêu cầu sẽ tham khảo một quan hệ riêng biệt nào đó. Tự điển dữ liệu là một trong những phần được truy xuất thường xuyên nhất của CSDL. Như vậy, bộ quản trị buffer sẽ không nên xoá các khối tự điển dữ liệu khỏi bộ nhớ chính trừ phi các nhân tố khác bức chế làm điều đó. Một chỉ mục (Index) đối với một file được truy xuất thường xuyên hơn chính bản thân file, vậy thì bộ quản trị buffer cũng không nên xoá khối chỉ mục khỏi bộ nhớ chính nếu có sự lựa chọn.

Chiến luợc thay thế khối CSDL lý tưởng cần hiểu biết về các hoạt động CSDL đang được thực hiện. Không một chiến lược đơn lẻ nào được biết nắm bắt được toàn bộ các viễn cảnh có thể. Tuy vậy, một điều đáng ngạc nhiên là phần lớn các hệ CSDL sử dụng LRU bất chấp các khuyết điểm của chiến lược đó.

Chiến lược được sử dụng bởi bộ quản trị buffer để thay thế khối bị ảnh hưởng bởi các nhân tố khác hơn là nhân tố thời gian tại đó khối được tham khảo trở lại. Nếu hệ thống đang xử lý các yêu cầu của một vài người sử dụng cạnh tranh, hệ thống (con) điều khiển cạnh tranh (concurrency-control subsystem) có thể phải làm trễ một số yêu cầu để đảm bảo tính nhất quán của CSDL. Nếu bộ quản trị buffer được cho các thông tin từ hệ thống điều khiển cạnh tranh mà nó nêu rõ những yêu cầu nào đang bị làm trễ, nó có thể sử dụng các thông tin này để thay đổi chiến lược thay thế khối của nó. Đặc biệt, các khối cần thiết bởi các yêu cầu tích cực (active requests) có thể được giữ lại trong buffer, toàn bộ các bất lợi đổ dồn lên các khối cần thiết bởi các yêu cầu bị làm trễ.

Hệ thống (con) khôi phục (crash-recovery subsystem) áp đặt các ràng buộc nghiêm nhặt lên việc thay thế khối. Nếu một khối bị sửa đổi, bộ quản trị buffer không được phép viết lại phiên bản mới của khối trong buffer lên đĩa, vì điều này phá huỷ phiên bản cũ. Thay vào đó, bộ quản trị khối phải tìm kiếm quyền từ hệ thống khôi phục trước khi viết khối. Hệ thống khôi phục có thể đòi hỏi một số khối nhất định khác là xuất bắt buộc (forced output) trước khi cấp quyền cho bộ quản trị buffer để xuất ra khối được yêu cầu.

TỔ CHỨC FILE

Một file được tổ chức logic như một dãy các mẩu tin (record). Các mẩu tin này được ánh xạ lên các khối đĩa. File được cung cấp như một xây dựng cơ sở trong hệ điều hành, như vậy ta sẽ giả thiết sự tồn tại của hệ thống file nền. Ta cần phải xét những phương pháp biểu diễn các mô hình dữ liệu logic trong thuật ngữ file.

Các khối có kích cỡ cố định được xác định bởi tính chất vật lý của đĩa và bởi hệ điều hành, song kích cỡ của mẩu tin lại thay đổi. Trong CSDL quan hệ, các bộ của các quan hệ khác nhau nói chung có kích cỡ khác nhau.

Một tiếp cận để ánh xạ một CSDL đến các file là sử dụng một số file, và lưu trữ các mẩu tin thuộc chỉ một độ dài cố định vào một file đã cho nào đó. Một cách khác là cấu trúc các file sao cho ta có thể điều tiết nhiều độ dài cho các mẩu tin. Các file của các mẩu tin độ dài cố định dễ dàng thực thi hơn file của các mẩu tin độ dài thay đổi.

MẪU TIN ĐỘ DÀI CỐ ĐỊNH (Fixed-Length Records)

Xét một file các mẩu tin account đối với CSDL ngân hàng, mỗi mẩu tin của file này được xác định như sau:

```
type
    depositor = record
        branch_name: char(20);
        account_number: char(10);
        balance:real;
    end
```

Giả sử mỗi một ký tự chiếm 1 byte và mỗi số thực chiếm 8 byte, như vậy mẩu tin account có độ dài 40 bytes. Một cách tiếp cận đơn giản là sử dụng 40 byte đầu tiên cho mẩu tin thứ nhất, 40 byte kế tiếp cho mẩu tin thứ hai, ... Cách tiếp cận đơn giản này nảy sinh những vấn đề sau;

0	Perryridge	A-102	400
1	Round Hill	A-305	350
2	Mianus	A-215	700
3	Downtown	A-101	500
4	Redwood	A-222	700
5	Perryridge	A-201	900
6	Brighton	A-217	750
7	Downtown	A-110	600
8	Perryridge	A-218	700

1. File F chứa các mẩu tin account

0	Perryridge	A-102	400
1	Round Hill	A-305	350
3	Downtown	A-101	500
4	Redwood	A-222	700
5	Perryridge	A-201	900
6	Brighton	A-217	750
7	Downtown	A-110	600
8	Perryridge	A-218	700

2. File F sau khi xóa mẩu tin 2 và di chuyển các mẩu tin sau nó

0	Perryridge	A-102	400
1	Round Hill	A-305	350
8	Perryridge	A-218	700
3	Downtown	A-101	500
4	Redwood	A-222	700
5	Perryridge	A-201	900
6	Brighton	A-217	750
7	Downtown	A-110	600

3. File F sau khi xóa mẩu tin 2 và di chuyển mẩu tin cuối vào vị chí của

header				
0	Perryridge	A-102	400	
1				
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4				
5	Perryridge	A-201	900	
6				
7	Downtown	A-110	600	
8	Perryridge	A-218	700	

1. Khó khăn khi xoá một mẩu tin từ cấu trúc này. Không gian bị chiếm bởi mẩu tin bị xoá phải được lấp đầy với mẩu tin khác của file hoặc ta phải đánh dấu mẩu tin bị xoá.
2. Trừ khi kích cỡ khối là bội của 40, nếu không một số mẩu tin sẽ bắt chéo qua biên khối, có nghĩa là một phần mẩu tin được lưu trong một khối, một phần khác được lưu trong một khối khác. như vậy đòi hỏi phải truy xuất hai khối để đọc/viết một mẩu tin "bắc cầu" đó.

Khi một mẩu tin bị xoá, ta có thể di chuyển mẩu tin kề sau nó vào không gian bị chiếm một cách hình thức bởi mẩu tin bị xoá, rồi mẩu tin kế tiếp vào không gian bị chiếm của mẩu tin vừa được di chuyển, cứ như vậy cho đến khi mỗi mẩu tin đi sau mẩu tin bị xoá được dịch chuyển hướng về đầu. Cách tiếp cận này đòi hỏi phải di chuyển một số lớn các mẩu tin. Một cách tiếp cận khác đơn giản hơn là di chuyển mẩu tin cuối cùng vào không gian bị chiếm bởi mẩu tin bị xoá. Song cách tiếp cận này đòi hỏi phải truy xuất khối bổ xung. Vì hoạt động xen xảy ra thường xuyên hơn hoạt động xoá, ta có thể chấp nhận việc để "ngỏ" không gian bị chiếm bởi mẩu tin bị xoá, và chờ một hoạt động xen đến sau để tái sử dụng không gian đó. Một dấu trên mẩu tin bị xoá là không đủ vì sẽ gây khó khăn cho việc tìm kiếm không gian "tự do" đó khi xen. Như vậy ta cần đưa vào cấu trúc bổ xung. ở đầu của file, ta cấp phát một số byte nhất định làm header của file. Header này sẽ chứa đựng thông tin về file. Header chứa địa chỉ của mẩu tin bị xoá thứ nhất, trong nội dung của mẩu tin này có chứa địa chỉ của mẩu tin bị xoá thứ hai và cứ như vậy. Như vậy, các mẩu tin bị xoá sẽ tạo ra một danh sách liên kết dược gọi là danh sách tự do (free list). Khi xen mẩu tin mới, ta sử dụng con trỏ đầu danh sách được chứa trong header để xác định danh sách, nếu danh sách không rỗng ta xen mẩu tin mới vào vùng được trỏ bởi con trỏ đầu danh sách nếu không ta xen mẩu tin mới vào cuối file.

Xen và xoá đối với file mẩu tin độ dài cố định thực hiện đơn giản vì không gian được giải phóng bởi mẩu tin bị xoá đúng bằng không gian cần thiết để xen một mẩu tin. Đối với file của các mẩu tin độ dài thay đổi vấn đề trở nên phức tạp hơn nhiều.

MẨU TIN ĐỘ DÀI THAY ĐỔI (Variable-Length Records)

Mẩu tin độ dài thay đổi trong CSDL do bởi:

- Việc lưu trữ nhiều kiểu mẩu tin trong một file
- Kiểu mẩu tin cho phép độ dài trường thay đổi
- Kiểu mẩu tin cho phép lặp lại các trường

Có nhiều kỹ thuật để thực hiện mẩu tin độ dài thay đổi. Để minh hoạ ta sẽ xét các biểu diễn khác nhau trên các mẩu tin độ dài thay đổi có định dạng sau:

```
Type   account_list = record
                branch_name: char(20)          ;
                account_info: array[ 1.. ∞ ] of record
                        account_number: char(10);
                        balance: real;
                end;
       end
```

Biểu diễn chuỗi byte (Byte-String Representation)

Một cách đơn giản để thực hiện các mẩu tin độ dài thay đổi là ***gắn một ký hiệu đặc biệt End-of-record (⊥) vào cuối mỗi record***. Khi đó, ta có thể lưu mỗi mẩu tin như một chuỗi byte liên tiếp. Thay vì sử dụng một ký hiệu đặc biệt ở cuối của mỗi mẩu tin, một phiên bản của biểu diễn chuỗi byte ***lưu trữ độ dài mẩu tin ở bắt đầu của mỗi mẩu tin***.

0	Perryridge	A-102	400	A-201	900	A210	700	⊥
1	Round Hill	A-301	350	⊥				
2	Mianus	A-101	800	⊥				
3	Downtown	A-211	500	A-222	600	⊥		
4	Redwood	A-300	650	A-200	1200	A-255	950	⊥
5	Brighton	A-111	750	⊥				

Biểu diễn chuỗi byte của các mẩu tin độ dài thay đổi

Biểu diễn chuỗi byte có các bất lợi sau:

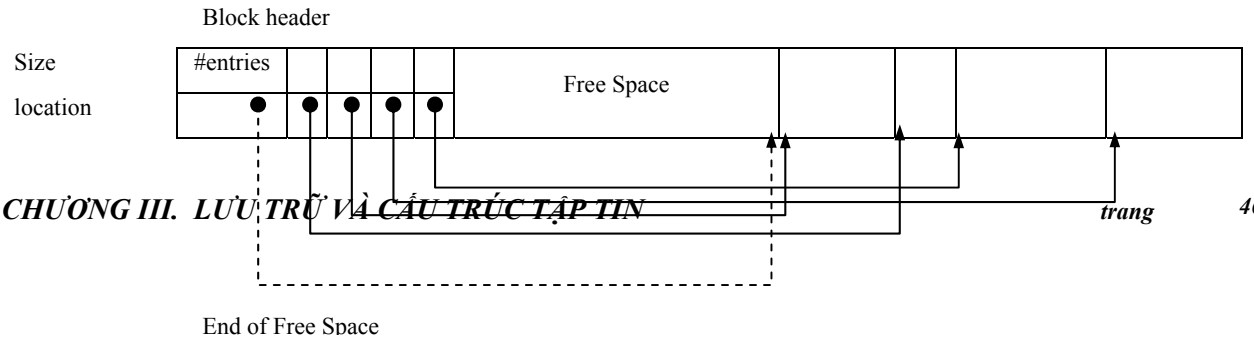
- Khó sử dụng không gian bị chiếm hình thức bởi một mẩu tin bị xoá, điều này dẫn đến một số lớn các mảnh nhỏ của lưu trữ đĩa bị lãng phí.
- Không có không gian cho sự phát triển các mẩu tin. Nếu một mẩu tin độ dài thay đổi dài ra, nó phải được di chuyển và sự di chuyển này là đắt giá nếu mẩu tin bị chốt.

Biểu diễn chuỗi byte không thường được sử dụng để thực hiện mẩu tin độ dài thay đổi, song một dạng sửa đổi của nó được gọi là ***cấu trúc khe-trang*** (slotted-page structure) thường được dùng để tổ chức mẩu tin trong một khối đơn.

Trong cấu trúc slotted-page, có một header ở bắt đầu của mỗi khối, chứa các thông tin sau:

- Số các đầu vào mẩu tin (record entries) trong header
- Điểm cuối không gian tự do (End of Free Space) trong khối
- Một mảng các đầu vào chứa vị trí và kích cỡ của mỗi mẩu tin

Các mẩu tin hiện hành được cấp phát kề nhau trong khối, bắt đàu từ cuối khối, Không gian tự do trong khối là một vùng kề nhau, nằm giữa đầu vào cuối cùng trong mảng header và mẩu tin đàu tiên. Khi một mẩu tin được xen vào, không gian cấp phát cho nó ở cuối của không gian tự do, và đầu vào tương ứng với nó được thêm vào header.



Nếu một mẩu tin bị xoá, không gian bị chiếm bởi nó được giải phóng, đầu vào ứng với nó được đặt là bị xoá (kích cỡ của nó được đặt chẳng hạn là -1). Sau đó, các mẩu tin trong khối trước mẩu tin bị xoá được di chuyển sao cho không gian tự do của khối lại là phần nằm giữa đầu vào cuối cùng của mảng header và mẩu tin đầu tiên. Con trỏ điểm cuối không gian tự do và các con trỏ ứng với mẩu tin bị di chuyển được cập nhật. Sự lớn lên hay nhỏ đi của mẩu tin cũng sử dụng kỹ thuật tương tự (trong trường hợp khối còn không gian cho sự lớn lên của mẩu tin). Cái giá phải trả cho sự di chuyển không quá cao vì các khối có kích cỡ không lớn (thường 4Kbytes).

Biểu diễn độ dài cố định

Một cách khác để thực hiện mẩu tin độ dài thay đổi một cách hiệu quả trong một hệ thống file là sử dụng một hoặc một vài mẩu tin độ dài cố định để biểu diễn một mẩu tin độ dài thay đổi. Hai kỹ thuật thực hiện file của các mẩu tin độ dài thay đổi sử dụng mẩu tin độ dài cố định là:

1. **Không gian dự trữ (reserved space)**. Giả thiết rằng các mẩu tin có độ dài không vượt quá một ngưỡng (độ dài tối đa). Ta có thể sử dụng mảu tin độ dài cố định (có độ dài tối đa), Phần không gian chưa dùng đến được lấp đầy bởi một ký tự đặc biệt: null hoặc End-of-record.
2. **Contrỏ (Pointers)**. Mẩu tin độ dài thay đổi được biểu diễn bởi một danh sách các mẩu tin độ dài cố định, được "móc xích" với nhau bởi các con trỏ.

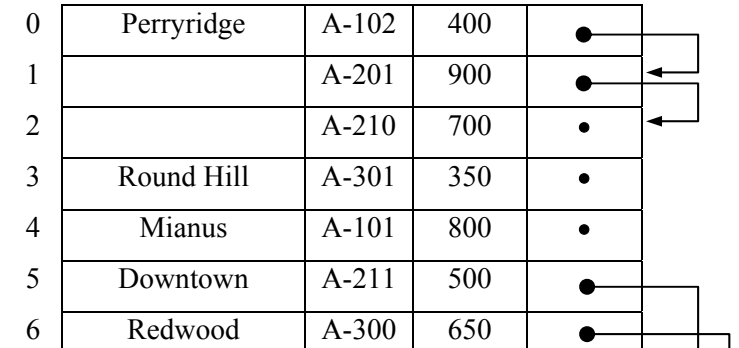
Sự bất lợi của cấu trúc con trỏ là lãng phí không gian trong tất cả các mẩu tin ngoại trừ mẩu tin đầu tiên trong danh sách (mẩu tin đầu tiên cần trường branch_name, các mẩu tin sau trong danh sách không cần thiết có trường này!). Để giải quyết vấn đề này người ta đề nghị phân các khối trong file thành hai loại:

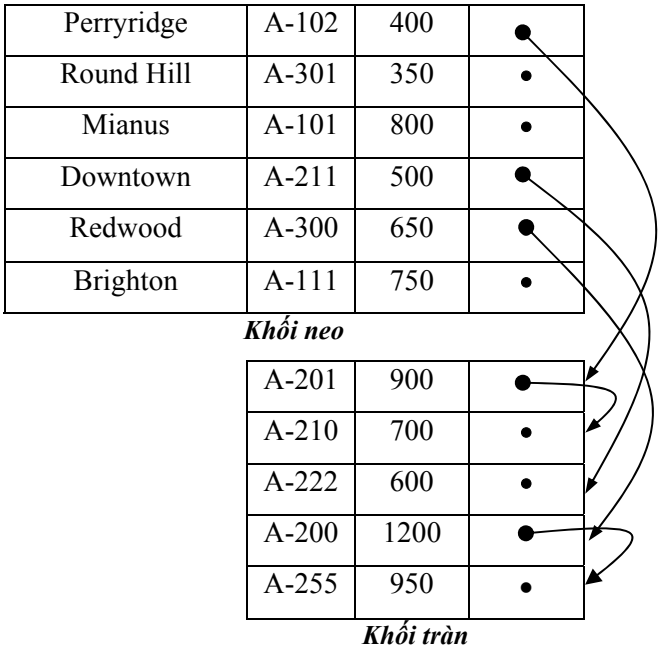
- **Khối neo (Anchor block).** chứa chỉ các mẩu tin đầu tiên trong danh sách
- **Khối tràn (Overflow block).** chứa các mẩu tin còn lại của danh sách

Như vậy, tất cả các mẩu tin trong một khối có cùng độ dài, cho dù file có thể chứa các mẩu tin không cùng độ dài.

0	Perryridge	A-102	400	A-201	900	A210	700	⊥
1	Round Hill	A-301	350	⊥	⊥	⊥	⊥	⊥
2	Mianus	A-101	800	⊥	⊥	⊥	⊥	⊥
3	Downtown	A-211	500	A-222	600	⊥	⊥	⊥
4	Redwood	A-300	650	A-200	1200	A-255	950	⊥
5	Brighton	A-111	750	⊥	⊥	⊥	⊥	⊥

Sử dụng phương pháp không gian dự trữ





Cấu trúc khối neo và khối tràn

TỔ CHỨC CÁC MẪU TIN TRONG FILE

Ta đã xét làm thế nào để biểu diễn các mẩu tin trong một cấu trúc file. Một thể hiện của một quan hệ là một tập hợp các mẩu tin. Đã cho một tập hợp các mẩu tin, vấn đề đặt ra là làm thế nào để tổ chức chúng trong một file. Có một số cách tổ chức sau:

- **Tổ chức file đống (Heap File Organization).** Trong tổ chức này, một mẩu tin bất kỳ có thể được lưu trữ ở bất kỳ nơi nào trong file, ở đó có không gian cho nó. Không có thứ tự nào giữa các mẩu tin. Một file cho một quan hệ.

- **Tổ chức file tuần tự (Sequential File Organization).** Trong tổ chức này, các mẩu tin được lưu trữ *thứ tự tuần tự*, dựa trên ***giá trị của khoá tìm kiếm*** của mỗi mẩu tin.

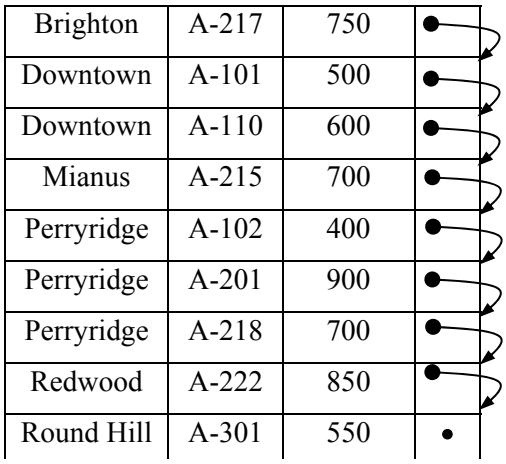
- **Tổ chức file băm (Hashed File Organization).** Trong tổ chức này, có một hàm băm được tính toán trên thuộc tính nào đó của mẩu tin. Kết quả của hàm băm xác định mẩu tin được bố trí trong khối nào trong file. Tổ chức này liên hệ chặt chẽ với cấu trúc chỉ mục.

• **Tổ chức file cụm (Clustering File Organization).** Trong tổ chức này, các mẩu tin của một vài quan hệ khác nhau có thể được lưu trữ trong cùng một file. Các mẩu tin có liên hệ của các quan hệ khác nhau được lưu trữ trên cùng một khối sao cho một hoạt động I/O đem lại các mẩu tin có liên hệ từ tất cả các quan hệ.

TỔ CHỨC FILE TUẦN TỰ

Tổ chức file tuần tự được thiết kế để xử lý hiệu quả các mẩu tin trong thứ tự được sắp dựa trên một khoá tìm kiếm (*search key*) nào đó. Để cho phép tìm lại nhanh chóng các mẩu tin theo thứ tự khoá tìm kiếm, ta "xích" các mẩu tin lại bởi các con trỏ. Con trỏ trong mỗi mẩu tin trỏ tới mẩu tin kế theo thứ tự khoá tìm kiếm. Hơn nữa, để tối ưu hoá số khối truy xuất trong xử lý file tuần tự, ta lưu trữ vật lý các mẩu tin theo thứ tự khoá tìm kiếm hoặc gần với khoá tìm kiếm như có thể.

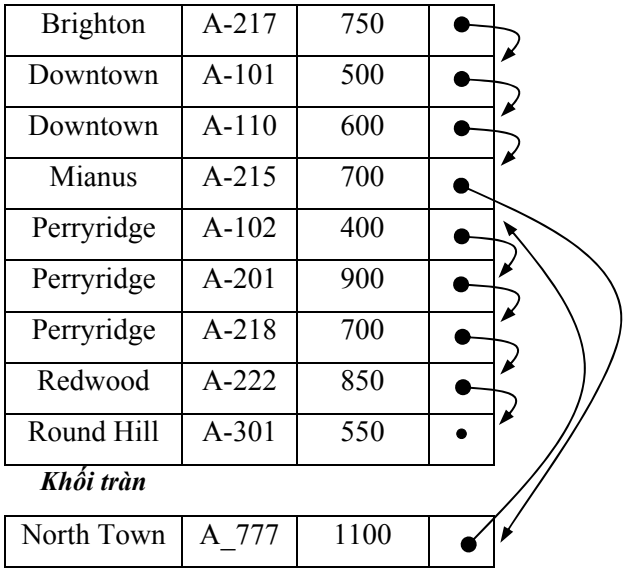
Tổ chức file tuần tự cho phép đọc các mẩu tin theo thứ tự được sắp mà nó có thể hưu dụng cho mục đích trình bày cũng như cho các thuật toán xử lý vấn tin (*query-processing algorithms*).



Brighton	A-217	750	
Downtown	A-101	500	
Downtown	A-110	600	
Mianus	A-215	700	
Perryridge	A-102	400	
Perryridge	A-201	900	
Perryridge	A-218	700	
Redwood	A-222	850	
Round Hill	A-301	550	

Khó khăn gặp phải của tổ chức này là việc duy trì thứ tự tuần tự vật lý của các mẩu tin khi xẩy ra các hoạt động xen, xoá, do cái giá phải trả cho việc di chuyển các mẩu tin khi xen, xoá. Ta có thể quản trị vấn đề xoá bởi dùng dây chuyền các con trỏ như đã trình bày trước đây. Đối với xen, ta có thể áp dụng các quy tắc sau:

1. Định vị mẩu tin trong file mà nó đi trước mẩu tin được xen theo thứ tự khoá tìm kiếm.
2. Nếu có mẩu tin tự do (không gian của mẩu tin bị xoá) trong cùng khối, xen mẩu tin vào khối này. Nếu không, xen mẩu tin mới vào một khối tràn. Trong cả hai trường hợp, điều chỉnh các con trỏ sao cho nó móc xích các mẩu tin theo thứ tự của khoá tìm kiếm.



Brighton	A-217	750	
Downtown	A-101	500	
Downtown	A-110	600	
Mianus	A-215	700	
Perryridge	A-102	400	
Perryridge	A-201	900	
Perryridge	A-218	700	
Redwood	A-222	850	
Round Hill	A-301	550	

Khối tràn

North Town	A_777	1100	

TỔ CHỨC FILE CỤM

Nhiều hệ CSDL quan hệ, mỗi quan hệ được lưu trữ trong một file sao cho có thể lợi dụng được toàn bộ những cái mà hệ thống file của điều hành cung cấp. Thông thường, các bộ của một quan hệ được biểu diễn như các mẩu tin độ dài cố định. Như vậy các quan hệ có thể ánh xạ vào một cấu trúc file. Sự thực hiện đơn giản đó của một hệ CSDL quan hệ rất phù hợp với các hệ CSDL được thiết kế cho các máy tính cá nhân. Trong các hệ thống đó, kích cỡ của CSDL nhỏ. Hơn nữa, trong một số máy tính cá nhân, chủ yếu kích cỡ tổng thể mã đối tượng đối với hệ CSDL là nhỏ. Một cấu trúc file đơn giản làm suy giảm lượng mã cần thiết để thực thi hệ thống.

Cách tiếp cận đơn giản này, để thực hiện CSDL quan hệ, không còn phù hợp khi kích cỡ của CSDL tăng lên. Ta sẽ thấy những điểm lợi về mặt hiệu năng từ việc gán một cách thận trọng các mẩu tin với các khối, và từ việc tổ chức kỹ lưỡng chính bản thân các khối. Như vậy, có vẻ như là một cấu trúc file phức tạp hơn lại có lợi hơn, ngay cả trong trường hợp ta giữ nguyên chiến lược lưu trữ mỗi quan hệ trong một file riêng biệt.

Tuy nhiên, nhiều hệ CSDL quy mô lớn không nhờ cậy trực tiếp vào hệ điều hành nền để quản trị file. Thay vào đó, một file hệ điều hành được cấp phát cho hệ CSDL. Tất cả các quan hệ được lưu trữ trong một file này, và sự quản trị file này thuộc về hệ CSDL. Để thấy những điểm lợi của việc lưu trữ nhiều quan hệ trong cùng một file, ta xét vấn tin SQL sau:

SELECT *account_number, customer_number, customer_treet, customer_city*

FROM *depositor, customer*

WHERE *depositor.customer_name = customer.customername*;

Câu vấn tin này tính một *phép nối* của các quan hệ *depositor* và *customer*. Như vậy, đối với mỗi bộ của *depositor*, hệ thống phải tìm bộ của *customer* có cùng giá trị *customer_name*. Một cách lý tưởng là việc tìm kiếm các mẩu tin này nhờ sự trợ giúp của chỉ mục. Bỏ qua việc tìm kiếm các mẩu tin như thế nào, ta chú ý vào việc truyền từ đĩa vào bộ nhớ. Trong trường hợp xấu nhất, mỗi mẩu tin ở trong một khối khác nhau, điều này buộc ta phải đọc một khối cho một mẩu tin được yêu cầu bởi câu vấn tin. Ta sẽ trình bày một cấu trúc file được thiết kế để thực hiện hiệu quả các câu vấn tin liên quan đến *depositor* ⋈ *customer*. Các bộ *depositor* đối với mỗi *customer_name* được lưu trữ gần bộ *customer* có cùng *customer_name*. Cấu trúc này trộn các bộ của hai quan hệ với nhau, nhưng cho phép xử lý hiệu quả phép nối. Khi một bộ của của quan hệ *customer* được đọc, toàn bộ khối chứa bộ này được đọc từ đĩa vào trong bộ nhớ chính. Do các bộ tương ứng của *depositor* được lưu trữ trên đĩa gần bộ *customer*, khối chứa bộ *customer* chứa các bộ của quan hệ *depositor* cần cho xử lý câu vấn tin. Nếu một *customer* có nhiều *account* đến nỗi các mẩu tin *depositor* không lấp đầy trong một khối, các mẩu tin còn lại xuất hiện trong khối kế cận. Cấu trúc file này, được gọi là ***gom cụm (clustering)***, cho phép ta đọc nhiều mẩu tin được yêu cầu chỉ sử dụng một đọc khối, như vậy ta có thể xử lý câu vấn tin đặc biệt này hiệu quả hơn.

customer_name	customer_street	customer_city
Hays	Main	Brooklyn
Turner	Putnam	Stamford

Quan hệ customer

Hays	Main	Brooklyn
Hays	A-102	
Hays	A-220	
Hays	A-503	
Turner	Putnam	Stamford
Turner	A-305	

Cấu trúc file cụm

Tuy nhiên, cấu trúc gom cụm trên lại tỏ ra không có lợi bằng tổ chức lưu mỗi quan hệ trong một file riêng, đối với một số câu vấn tin, chẳng hạn:

SELECT *

FROM *customer*

Việc xác định khi nào thì gom cụm thường phụ thuộc vào kiểu câu vấn tin mà người thiết kế CSDL nghĩ rằng nó xẩy ra thường xuyên nhất. Sử dụng thận trọng gom cụm có thể cải thiện hiệu năng đáng kể trong việc xử lý câu vấn tin.

LƯU TRỮ TỰ ĐIỂN DỮ LIỆU

Một hệ CSDL cần thiết duy trì *dữ liệu về các quan hệ,* như sơ đồ của các quan hệ. Thông tin này được gọi là ***tự điển dữ liệu (data dictionary)*** hay ***mục lục hệ thống (system catalog).*** Trong các kiểu thông tin mà hệ thống phải lưu trữ là:

- Các tên của các quan hệ
- Các tên của các thuộc tính của mỗi quan hệ
- Các miền (giá trị) và các độ dài của các thuộc tính
- Các tên của các View được định nghĩa trên CSDL và định nghĩa của các view này
- Các ràng buộc toàn vẹn

Nhiều hệ thống còn lưu trữ các thông tin liên quan đến người sử dụng hệ thống:

- Tên của người sử dụng được phép
- Giải trình thông tin về người sử dụng

Các dữ liệu thống kê và mô tả về các quan hệ có thể cũng được lưu trữ:

- Số bộ trong mỗi quan hệ
- Phương pháp lưu trữ được sử dụng cho mỗi quan hệ (cụm hay không)

Các thông tin về mỗi chỉ mục trên mỗi quan hệ cũng cần được lưu trữ :

- Tên của chỉ mục
- Tên của quan hệ được chỉ mục
- Các thuộc tính trên nó chỉ mục được định nghĩa

- Kiểu của chỉ mục được tạo

Toàn bộ các thông tin này trong thực tế bao hàm một CSDL nhỏ. Một số hệ CSDL sử dụng những cấu trúc dữ liệu và mã *mục đích đặc biệt* để lưu trữ các thông tin này. Nói chung, lưu trữ dữ liệu về CSDL trong chính CSDL vẫn được ưa chuộng hơn. Bằng cách sử dụng CSDL để lưu trữ dữ liệu hệ thống, ta đơn giản hoá cấu trúc tổng thể của hệ thống và cho phép sử dụng đầy đủ sức mạnh của CSDL trong việc truy xuất nhanh đến dữ liệu hệ thống.

Sự chọn lựa chính xác biểu diễn dữ liệu hệ thống sử dụng các quan hệ như thế nào là do người thiết kế hệ thống quyết định. Như một ví dụ, ta đề nghị sự biểu diễn sau:

System_catalog_schema = (relation_name, number_of_attributes)

Attribute_schema = (attribute_name, relation_name, domain_type, position, length)

User_schema = (user_name, encrypted_password, group)

Index_schema = (index_name, relation_name, index_type, index_attributes)

View_schema = (view_name, definition)

CHỈ MỤC

Ta xét hoạt động tìm sách trong một thư viện. Ví dụ ta muốn tìm một cuốn sách của một tác giả nào đó. Đầu tiên ta tra trong mục lục tác giả, một tấm thẻ trong mục lục này sẽ chỉ cho ta biết có thể tìm thấy cuốn sách đó ở đâu. Các thẻ trong một mục lục được thư viện sắp xếp thứ tự theo vần chữ cái , như vậy giúp ta có thể tìm đến thẻ cần tìm nhanh chóng không cần phải duyệt qua tất cả các thẻ. Chỉ mục của một file trong các công việc hệ thống rất giống với một mục lục trong một thư viện. Tuy nhiên, chỉ mục được làm như mục lục được mô tả như trên, trong thực tế, sẽ quá lớn để được quản lý một cách hiệu quả. Thay vào đó, người ta sử dụng các kỹ thuật chỉ mục tinh tế hơn. Có hai kiểu chỉ mục:

- **Chỉ mục được sắp (Ordered indices)**. được dựa trên một thứ tự sắp xếp theo các giá trị
- **Chỉ mục băm (Hash indices).** được dựa trên các giá trị được phân phối đều qua các bucket. Bucket mà một giá trị được gán với nó được xác định bởi một hàm, được gọi là hàm băm (hash function)

Đối với cả hai kiểu này, ta sẽ nêu ra một vài kỹ thuật, đáng lưu ý là không kỹ thuật nào là tốt nhất. Mỗi kỹ thuật phù hợp với các ứng dụng CSDL riêng biệt. Mỗi kỹ thuật phải được đánh giá trên cơ sở của các nhân tố sau:

- **Kiểu truy xuất**: Các kiểu truy xuất được hỗ trợ hiệu quả. Các kiểu này bao hàm cả tìm kiếm mẩu tin với một giá trị thuộc tính cụ thể hoặc tìm các mẩu tin với giá trị thuộc tính nằm trong một khoảng xác định.
- **Thời gian truy xuất**: Thời gian để tìm kiếm một hạng mục dữ liệu hay một tập các hạng mục.
- **Thời gian xen**: Thời gian để xen một hạng mục dữ liệu mới. giá trị này bao hàm thời gian để tìm vị trí xen thích hợp và thời gian cập nhật cấu trúc chỉ mục.
- **Thời gian xoá**: Thời gian để xoá một hạng mục dữ liệu. giá trị này bao hàm thời gian tìm kiếm hạng mục cần xoá, thời gian cập nhật cấu trúc chỉ mục.
- **Tổng phí tổn không gian**: Không gian phụ bị chiếm bởi một cấu trúc chỉ mục.

Một file thường đi kèm với một vài chỉ mục. Thuộc tính hoặc tập hợp các thuộc tính được dùng để tìm kiếm mẩu tin trong một file được gọi là ***khoá tìm kiếm***. Chú ý rằng định nghĩa này

khác với định nghĩa khoá sơ cấp (primary key), khoá dự tuyển (candidate key), và siêu khoá (superkey). Như vậy, nếu có một vài chỉ mục trên một file, có một vài khoá tìm kiếm tương ứng.

CHỈ MỤC ĐƯỢC SẮP.

Một chỉ mục lưu trữ các giá trị khoá tìm kiếm trong thứ tự được sắp, và kết hợp với mỗi khoá tìm kiếm, các mẩu tin chứa khoá tìm kiếm này. Các mẩu tin trong file được chỉ mục có thể chính nó cũng được sắp. Một file có thể có một vài chỉ mục trên những khoá tìm kiếm khác nhau. Nếu file chứa các mẩu tin được sắp tuần tự, chỉ mục trên khoá tìm kiếm xác định thứ tự này của file được gọi chỉ mục sơ cấp (primary index). Các chỉ mục sơ cấp cũng được gọi là chỉ mục cụm (clustering index). Khoá tìm kiếm của chỉ mục sơ cấp thường là khoá sơ cấp (khoá chính). Các chỉ mục, khoá tìm kiếm của nó xác định một thứ tự khác với thứ tự của file, được gọi là các chỉ mục thứ cấp (secondary indices) hay các chỉ mục không cụm (nonclustering indices).

Brighton	A-217	750	●
Downtown	A-101	500	●
Downtown	A-110	600	●
Mianus	A-215	700	●
Perryridge	A-102	400	●
Perryridge	A-201	900	●
Perryridge	A-218	700	●
Redwood	A-222	850	●
Round Hill	A-301	550	•

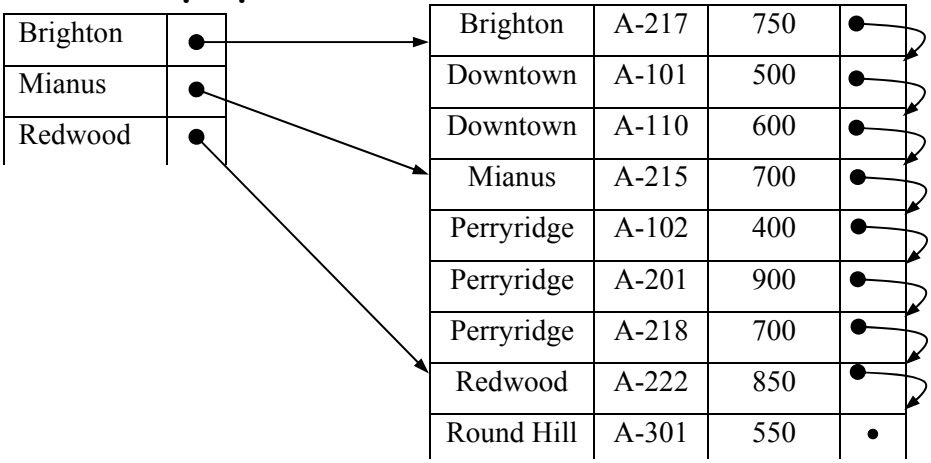
file tuần tự các mẩu tin account

Chỉ mục sơ cấp.

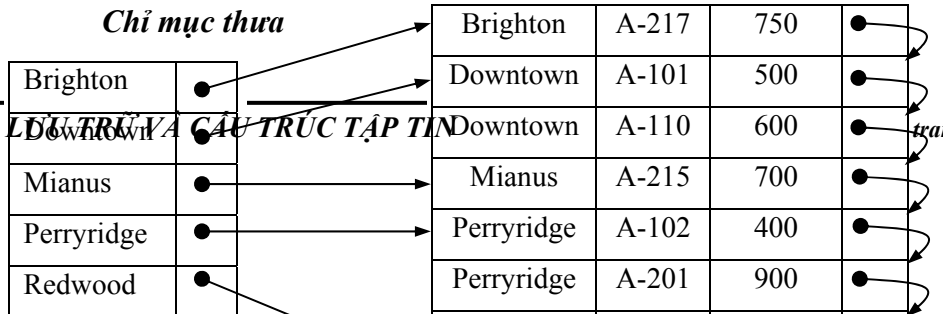
Trong phần này, ta giả thiết rằng tất cả các file được sắp thứ tự tuần tự trên một khoá tìm kiếm nào đó. Các file như vậy, với một *chỉ mục sơ cấp* trên khoá tìm kiếm này, được gọi là file tuần tự chỉ mục (index-sequential files). Chúng biểu diễn một trong các sơ đồ xưa nhất được dùng trong hệ CSDL. Chúng được thiết kế cho các ứng dụng đòi hỏi cả xử lý tuần tự toàn bộ file lẫn truy xuất ngẫu nhiên đến một mẩu tin.

Chỉ mục đặc và chỉ mục thưa (Dense and Sparse Indices)

Chỉ mục đặc



Chỉ mục thưa



Có hai loại chỉ mục được sắp:

- **Chỉ mục đặc**. Mỗi mẩu tin chỉ mục (đầu vào chỉ mục/ index entry) xuất hiện đối với mỗi giá trị khoá tìm kiếm trong file. mẩu tin chỉ mục chứa giá trị khoá tìm kiếm và một con trỏ tới mẩu tin dữ liệu đầu tiên với giá trị khoá tìm kiếm đó.
- **Chỉ mục thưa**. Một mẩu tin chỉ mục được tạo ra chỉ với một số giá trị. Cũng như với chỉ mục đặc, mỗi mẩu tin chỉ mục chứa một giá trị khoá tìm kiếm và một con trỏ tới mẩu tin dữ liệu đầu tiên với giá trị khoá tìm kiếm này. Để định vị một mẩu tin, ta tìm *đầu vào chỉ mục* với giá trị khoá tìm kiếm lớn nhất trong các giá trị khoá tìm kiếm nhỏ hơn hoặc bằng giá trị khoá tìm kiếm đang tìm. Ta bắt đầu từ mẩu tin được trỏ tới bởi đầu vào chỉ mục, và lần theo các con trỏ trong file (dữ liệu) đến tận khi tìm thấy mẩu tin mong muốn.

Ví dụ: Giả sử ta tìm các kiếm mẩu tin đối với chi nhánh Perryridge, sử dụng chỉ mục đặc. Đầu tiên, tìm Perryridge trong chỉ mục (tìm nhị phân!), đi theo con trỏ tương ứng đến mẩu tin dữ liệu (với Branch_name = Perryridge) đầu tiên, xử lý mẩu tin này, sau đó đi theo con trỏ trong mẩu tin này để định vị mẩu tin kế trong thứ tự khoá tìm kiếm, xử lý mẩu tin này, tiếp tục như vậy đến tận khi đạt tới mẩu tin có Branch_name khác với Perryridge.

Đối với chỉ mục thưa, đầu tiên tìm trong chỉ mục, đầu vào có Branch_name lớn nhất trong các đầu vào có Branch_name nhỏ hơn hoặc bằng Perryridge, ta tìm được đầu vào với Mianus, lần theo con trỏ tương ứng đến mẩu tin dữ liệu, đi theo con trỏ trong mẩu tin Mianus để định vị mẩu tin kế trong thứ tự khoá tìm kiếm và cứ như vậy đến tận khi đạt tới mẩu tin dữ liệu Perryridge đầu tiên, sau đó xử lý bắt đầu từ điểm này.

Chỉ mục đặc cho phép tìm kiếm mẩu tin nhanh hơn chỉ mục thưa, song chỉ mục thưa lại đòi hỏi ít không gian hơn chỉ mục đặc. Hơn nữa, chỉ mục thưa yêu cầu một tổn phí duy trì nhỏ hơn đối với các hoạt động xen, xoá.

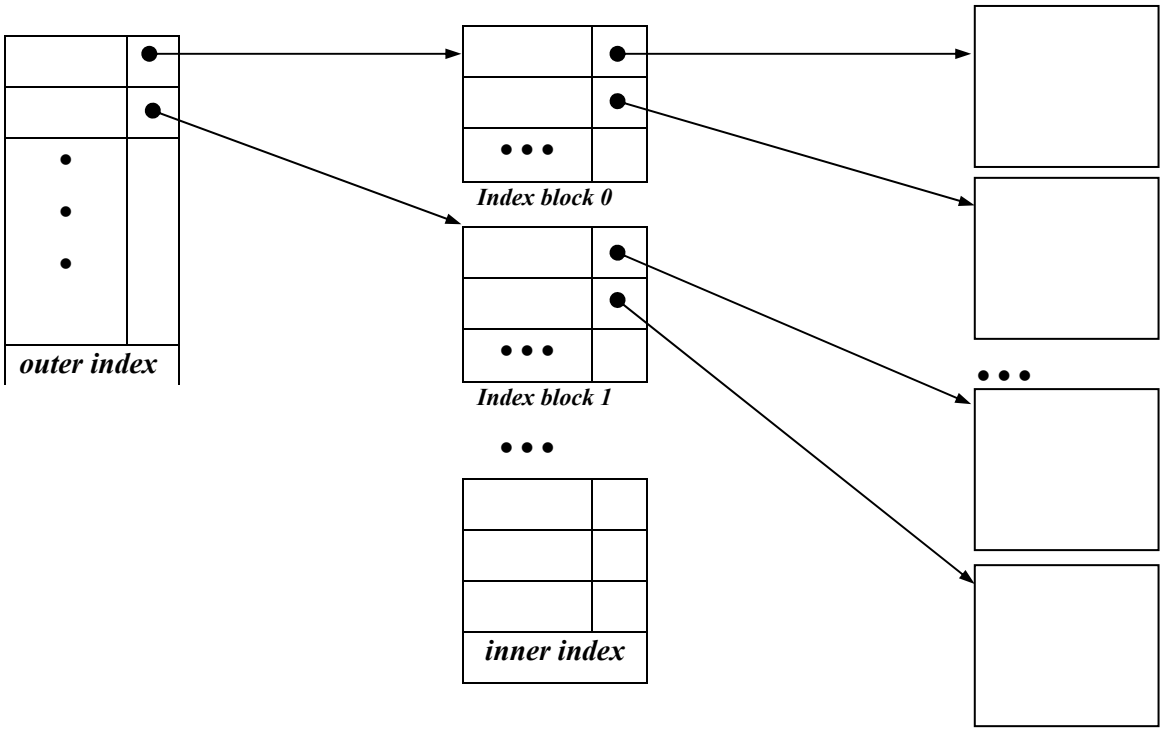
Người thiết kế hệ thống phải cân nhắc sự cân đối giữa thời truy xuất và tổn phí không gian. Một thoả hiệp tốt là có một chỉ mục thưa với một đầu vào chỉ mục cho mỗi khối, vì như vậy cái giá nổi trội trong xử lý một yêu cầu CSDL là thời gian mang một khối từ đĩa vào bộ nhớ chính. Mỗi khi một khối được mang vào, thời gian quét toàn bộ khối là không đáng kể. Sử dụng chỉ mục thưa, ta tìm khối chứa mẩu tin cần tìm. Như vậy, trừ phi mẩu tin nằm trên khối tràn, ta tối thiểu hoá được truy xuất khối, trong khi giữ được kích cỡ của chỉ mục nhỏ như có thể.

Chỉ mục nhiều mức

Chỉ mục có thể rất lớn, ngay cả khi sử dụng chỉ mục thưa, và không thể chứa đủ trong bộ nhớ một lần. Tìm kiếm đầu vào chỉ mục đối với các chỉ mục như vậy đòi hỏi phải đọc vài khối đĩa. Tìm kiếm nhị phân có thể được sử dụng để tìm một đầu vào trên file chỉ mục, song vẫn phải truy xuất khoảng $\lceil \log B \rceil$ khối, với B là số khối đĩa chứa chỉ mục. Nếu B lớn, thời gian truy xuất

này là đáng kể! Hơn nữa nếu sử dụng các khối tràn, tìm kiếm nhị phân không sử dụng được và như vậy việc tìm kiếm phải làm tuần tự. Nó đòi hỏi truy xuất lên đến B khối!!

Để giải quyết vấn đề này, Ta xem file chỉ mục như một file tuần tự và xây dựng chỉ mục thưa cho nó. Để tìm đầu vào chỉ mục, ta tìm kiếm nhị phân trên chỉ mục "ngoài" để được mẩu tin có khoá tìm kiếm lớn nhất trong các mẩu tin có khoá tìm kiếm nhỏ hơn hoặc bằng khoá muốn tìm. Con trỏ tương ứng trỏ tới khối của chỉ mục "trong". Trong khối này, tìm kiếm mẩu tin có khoá tìm kiếm lớn nhất trong các mẩu tin có khoá tìm kiếm nhỏ hơn hoặc bằng khoá muốn tìm, trường con trỏ của mẩu tin này trỏ đến khối chứa mẩu tin cần tìm. Vì chỉ mục ngoài nhỏ, có thể nằm sẵn trong bộ nhớ chính, nên một lần tìm kiếm chỉ cần một truy xuất khối chỉ mục. Ta có thể lặp lại quá trình xây dựng trên nhiều lần khi cần thiết. Chỉ mục với không ít hơn hai mức được gọi là chỉ mục nhiều mức. Với chỉ mục nhiều mức, việc tìm kiếm mẩu tin đòi hỏi truy xuất khối ít hơn đáng kể so với tìm kiếm nhị phân.



Cập nhật chỉ mục

Mỗi khi xen hoặc xoá một mẩu tin, bắt buộc phải cập nhật các chỉ mục kèm với file chứa mẩu tin này. Dưới đây, ta mô tả các thuật toán cập nhật cho các chỉ mục một mức

- **Xoá.** Để xoá một mẩu tin, đầu tiên phải tìm mẩu tin muốn xoá. Nếu mẩu tin bị xoá là mẩu tin đầu tiên trong dây chuyền các mẩu tin được xác định bởi con trỏ của đầu vào chỉ mục trong quá trình tìm kiếm, có hai trường hợp phải xét: nếu mẩu tin bị xoá là mẩu tin duy nhất trong dây chuyền, ta xoá đầu vào trong chỉ mục tương ứng, nếu không, ta thay thế khoá tìm kiếm trong đầu vào chỉ mục bởi khoá tìm kiếm của mẩu tin kề sau mẩu tin bị xoá trong dây chuyền, con trỏ bởi địa chỉ mẩu tin kề sau đó. Trong trường hợp khác, việc xoá mẩu tin không dẫn đến việc điều chỉnh chỉ mục.
- **Xen.** Trước tiên, tìm kiếm dựa trên khoá tìm kiếm của mẩu tin được xen. Nếu là chỉ mục đặc và giá trị khoá tìm kiếm không xuất hiện trong chỉ mục, xen giá trị khoá này và con trỏ tới mẩu tin vào chỉ mục. Nếu là chỉ mục thưa và lưu đầu vào cho mỗi khối, không cần

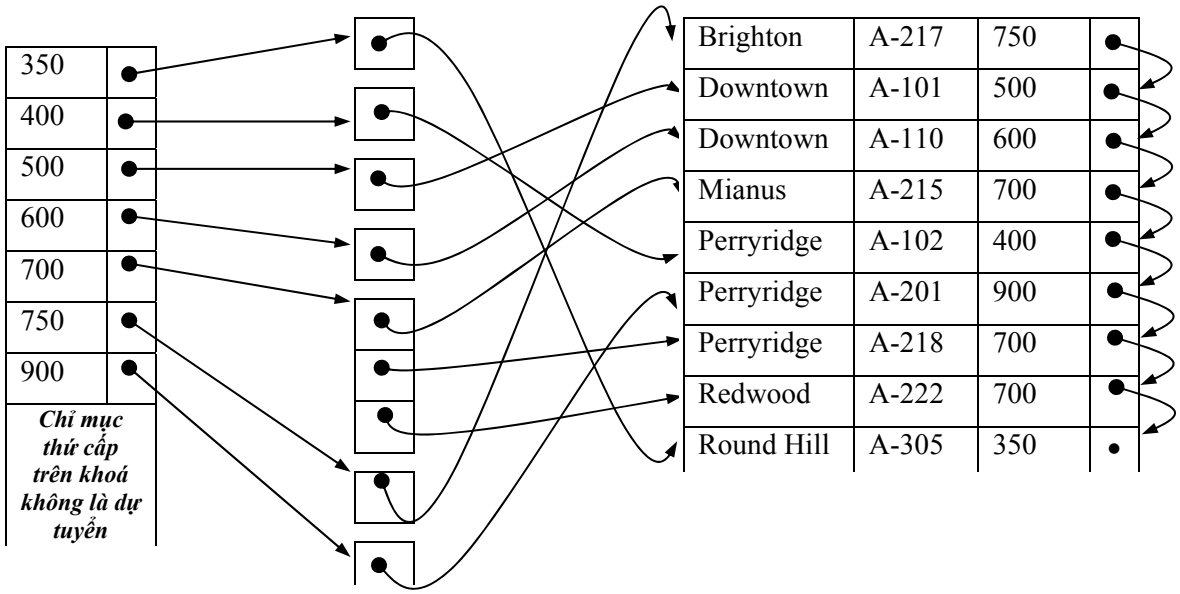
thiết phải thay đổi trừ phi khối mới được tạo ra. Trong trường hợp đó, giá trị khoá tìm kiếm đầu tiên trong khối mới được xen vào chỉ mục.

Giả thuật xen và xoá đối với chỉ mục nhiều mức là một mở rộng đơn giản của các giả thuật vừa được mô tả.

Chỉ mục thứ cấp.

Chỉ mục thứ cấp trên một khoá dự tuyển giống như chỉ mục sơ cấp đặc ngoại trừ các mẩu tin được trỏ đến bởi các giá trị liên tiếp trong chỉ mục không được lưu trữ tuần tự. Nói chung, chỉ mục thứ cấp có thể được cấu trúc khác với chỉ mục sơ cấp. Nếu khoá tìm kiếm của chỉ mục sơ cấp không là khoá dự tuyển, chỉ mục chỉ cần trỏ đến mẩu tin đầu tiên với một giá trị khoá tìm kiếm riêng là đủ (các mẩu tin khác cùng giá trị khoá này có thể tìm lại được nhờ quét tuần tự file).

Nếu khoá tìm kiếm của một chỉ mục thứ cấp không là khoá dự tuyển, việc trỏ tới mẩu tin đầu tiên với giá trị khoá tìm kiếm riêng không đủ, do các mẩu tin trong file không còn được sắp tuần tự theo khoá tìm kiếm của chỉ mục thứ cấp, chúng có thể nằm ở bất kỳ vị trí nào trong file. Bởi vậy, chỉ mục thứ cấp phải chứa tất cả các co trỏ tới mỗi mẩu tin. Ta có thể sử dụng mức phụ gián tiếp để thực hiện chỉ mục thứ cấp trên các khoá tìm kiếm không là khoá dự tuyển. Các con trỏ trong chỉ mục thứ cấp như vậy không trực tiếp trỏ tới mẩu tin mà trỏ tới một bucket chứa các con trỏ tới file.



Chỉ mục thứ cấp phải là đặc, với một đầu vào chỉ mục cho mỗi mẩu tin. Chỉ mục thứ cấp cải thiện hiệu năng các vấn tin sử dụng khoá tìm kiếm không là khóa của chỉ mục sơ cấp, tuy nhiên nó lại đem lại một tổn phí sửa đổi CSDL đáng kể.Việc quyết định các chỉ mục thứ cấp nào là cần thiết dựa trên đánh giá của nhà thiết kế CSDL về tần xuất vấn tin và sửa đổi.

FILE CHỈ MỤC B^+-CÂY (B^+-Tree Index file)

Tổ chức file chỉ mục tuần tự có một nhược điểm chính là làm giảm hiệu năng khi file lớn lên. Để khắc phục nhược điểm đó đòi hỏi phải tổ chức lại file. Cấu trúc chỉ mục B^+-cây là cấu trúc được sử dụng rộng rãi nhất trong các cấu trúc đảm bảo được tính hiệu quả của chúng bất chấp các hoạt động xen, xoá. Chỉ mục B^+-cây là một dạng cây cân bằng (mọi đường dẫn từ gốc đến lá có cùng độ dài). Mỗi nút không là lá có số con nằm trong khoảng giữa $\lceil m/2 \rceil$ và m, trong đó m là một số cố định được gọi là bậc của B^+-cây. Ta thấy rằng cấu trúc B^+-cây cũng đòi hỏi một tổn phí

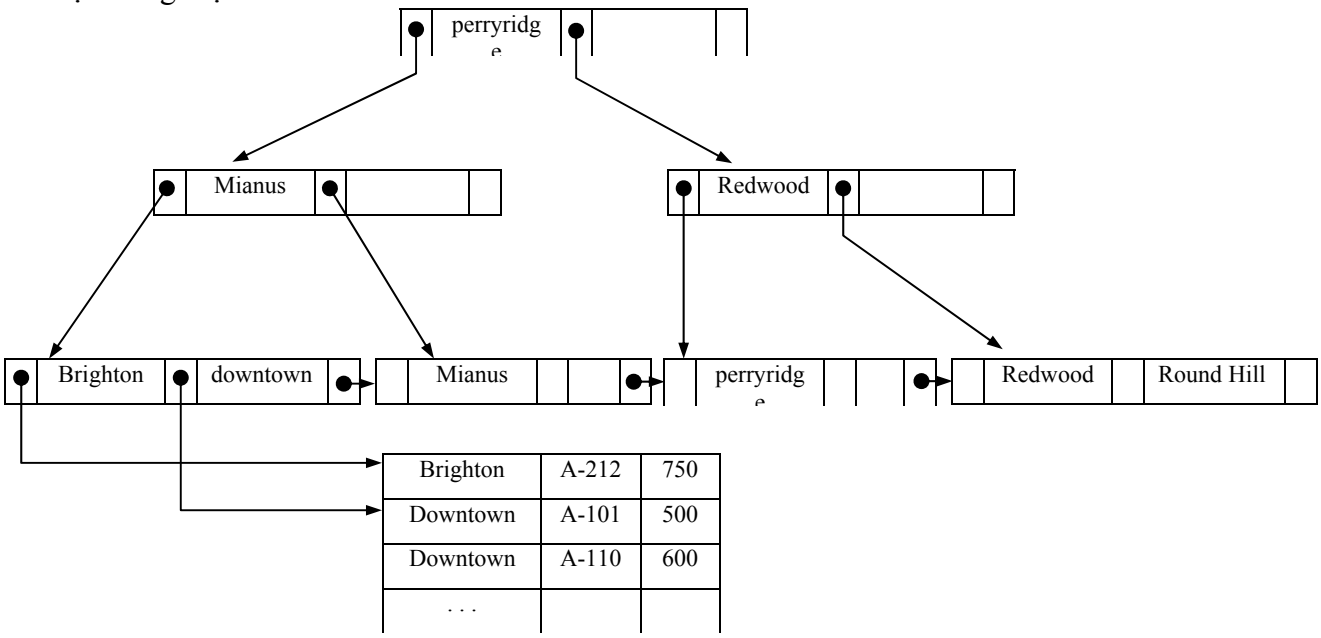
hiệu năng trên xen và xoá cũng như trên không gian. Tuy nhiên, tổn phí này là chấp nhận được ngay cả đối với các file có tần suất sửa đổi cao.

Cấu trúc của B$^+$-cây

Một chỉ mục B$^+$-cây là một chỉ mục nhiều mức, nhưng có cấu trúc khác với file tuần tự chỉ mục nhiều mức (multilevel index-sequential). Một nút tiêu biểu của B$^+$-cây chứa đến n-1 giá trị khoá tìm kiếm. K_1, K_2, ..., K_{n-1}, và n con trỏ P_1, P_2, ..., P_n, các giá trị khoá trong nút được sắp thứ tự: $i < j \Rightarrow K_i < K_j$.

P_1	K_1	P_2	K_2	. . .	P_{n-1}	K_{n-1}	P_n

Trước tiên, ta xét cấu trúc của nút lá. Đối với i = 1, 2, ..., n-1, con trỏ P_i trỏ tới hoặc mẩu tin với giá trị khoá K_i hoặc tới một bucket các con trỏ mà mỗi một trong chúng trỏ tới một mẩu tin với
giá trị khoá K_i. Cấu trúc bucket chỉ được sử dụng trong các trường hợp: hoặc khoá tìm kiếm không là khoá sơ cấp hoặc file không được sắp theo khoá tìm kiếm. Con trỏ P_n được dùng vào mục đích đặc biệt: P_n được dùng để móc xích các nút lá lại theo thứ tự khoá tìm kiếm, điều này cho phép xử lý tuần tự file hiệu quả. Bây giờ ta xem các giá trị khoá tìm kiếm được gắn với một nút lá như thế nào. Mỗi nút nút lá có thể chứa đến n-1 giá trị. Khoảng giá trị mà mỗi nút lá chứa là không chồng chéo. Như vậy, nếu L_i và L_j là hai nút lá với i < j thì mỗi giá trị khoá trong nút L_i nhỏ hơn mọi giá trị khoá trong L_j . Nếu chỉ mục B$^+$-cây là đặc, mỗi giá trị khoá tìm kiếm phải xuất hiện trong một nút lá nào đó.



Các nút không là lá của một B$^+$-cây tạo ra một chỉ mục nhiều mức trên các nút lá. Cấu trúc của các nút không là lá tương tự như cấu trúc nút lá ngoại trừ tất cả các con trỏ đều trỏ đến các nút của cây. Các nút không là lá có thể chứa đến m con trỏ và phải chứa không ít hơn $\lceil m/2 \rceil$ con trỏ ngoại trừ nút gốc. Nút gốc được phép chứa ít nhất 2 con trỏ. Số con trỏ trong một nút được gọi là số nan (fanout) của nút.

Con trỏ P_i của một nút không là lá (chứa p con trỏ, 1 < i < p) trỏ đến một cây con chứa các giá trị khoá tìm kiếm nhỏ hơn K_i và lớn hơn hoặc bằng K_{i-1}. Con trỏ P_1 trỏ đến cây con chứa các giá trị khoá tìm kiếm nhỏ hơn K_1. Con trỏ P_p trỏ tới cây con chứa các khoá tìm kiếm lớn hơn K_{p-1}.

Các vấn tin trên B^+-cây

Ta xét xử lý vấn tin sử dụng B^+-cây như thế nào ? Giả sử ta muốn tìn tất cả các mẩu tin với giá trị khoá tìm kiếm k. Đầu tiên, ta kiểm tra nút gốc, tìm giá trị khoá tìm kiếm nhỏ nhất lớn hơn k, giả sử giá trị khoá đó là K_i. Đi theo con trỏ Pi để di tới một nút khác. Nếu nút có p con trỏ và $k > K_{p-1}$, đi theo con trỏ P_p. Đến một nút tới, lặp lại quá trình tìm kiếm giá trị khoá tìm kiếm nhỏ nhất lớn hơn k và theo con trỏ tương ứng để đi tới một nút khác và tiếp tục như vậy đến khi đạt tới một nút lá. Con trỏ tương ứng trong nút lá hướng ta tới mẩu tin/bucket mong muốn. Số khối truy xuất không vượt quá $\left\lceil \log_{\lceil m/2 \rceil} K \right\rceil$, trong đó K là số giá trị khoá tìm kiếm trong B^+-cây, m là bậc của cây.

Cập nhật trên B^+-cây

- **Xen**. Sử dụng cùng kỹ thuật như tìm kiếm, ta tìm nút lá trong đó giá trị khoá tìm kiếm cần xen sẽ xuất hiện. Nếu khoá tìm kiếm đã xuất hiện rồi trong nút lá, xen mẩu tin vào trong file, thêm con trỏ tới mẩu tin vào trong bucket tương ứng. Nếu khoá tìm kiếm chưa hiện diện trong nút lá, ta xen mẩu tin vào trong file rồi xen giá trị khoá tìm kiếm vào trong nút lá ở vị trí đúng (bảo tồn tính thứ tự), tạo một bucket mới với con trỏ tương ứng. Nếu nút lá không còn chỗ cho giá trị khoá mới, Một khối mới được yêu cầu từ hệ điều hành, các giá trị khoá trong nút lá được tách một nửa cho nút mới, giá trị khoá mới được xen vào vị trí đúng của nó vào một trong hai khối này. Điều này kéo theo việc xen giá trị khoá đầu khối mới và con trỏ tới khối mới vào nút cha. Việc xen cặp giá trị khoá và con trỏ vào nút cha này lại có thể dẫn đến việc tách nút ra làm hai. Quá trình này có thể dẫn đến tận nút gốc. Trong trường hợp nút gốc bị tách làm hai, một nút gốc mới được tạo ra và hai con của nó là hai nút được tách ra từ nút gốc cũ, chiều cao cây tăng lên một.

Procedure Insert(*value* V, *pointer* P)

Tìm nút lá L sẽ chứa giá trị V

Insert_entry(L, V, P)

end procedure

Procedure Insert_entry(*node* L, *value* V, *pointer* P)

If (L có không gian cho (V, P) **then**

Xen (V, P) vào L

else begin /* tách L */

Tạo nút L'

If (L là nút lá) **then begin**

V' là giá trị sao cho $\lceil m/2 \rceil$ giá trị trong các giá trị $L.K_1$, $L.K_2$, ..., $L.K_{m-1}$, V nhỏ hơn V'

n là chỉ số nhỏ nhất sao cho $L.K_n \geq V'$

Di chuyển $L.P_n$, $L.K_n$, ..., $L.P_{m-1}$, $L.K_{n-1}$ sang L'

If (V < V') then xen (V, P) vào trong L else xen (P, V) vào trong L'

end else begin

V' là giá trị sao cho $\lceil m/2 \rceil$ giá trị trong các giá trị $L.K_1$, $L.K_2$, ..., $L.K_{m-1}$, V lớn hơn hoặc bằng V'

n là chỉ số nhỏ nhất sao cho $L.K_n \geq V'$

Thêm Nil, $L.K_n$, $L.P_{n+1}$, $L.K_{n+1}$, ..., $L.P_{m-1}$, $L.K_{m-1}$, $L.P_m$ vào L'

Xoá $L.K_n$, $L.P_{n+1}$, $L.K_{n+1}$, ..., $L.P_{m-1}$, $L.K_{m-1}$, $L.P_m$ khỏi L

```
                    If (V < V') then xen (P, V) vào trong L  else  xen (P, V) vào trong L'
                    xoá (Nil, V') khỏi L'
                end
                If (L không là nút gốc) then Insert_entry(parent(L), V', L')
                else begin
                    Tạo ra nút mới R với các nút con là L và L' với giá trị duy nhất trong nó là V'
                    Tạo R là gốc của cây
                end
                If (L) là một nút lá then begin
                    đặt L'.Pm = L.Pm
                    đặt L.Pm = L'
                end
        end
end procedure
```

- **Xoá**. Sử dụng kỹ thuật tìm kiếm tìm mẩu tin cần xoá, xoá nó khỏi file, xoá giá trị khoá tìm kiếm khỏi nút lá trong B^+-cây nếu không có bucket kết hợp với giá trị khoá tìm kiếm hoặc bucket trở nên rỗng sau khi xoá con trỏ tương ứng trong nó. Việc xoá một giá trị khoá khỏi một nút của B^+-cây có thể dẫn đến nút lá trở nên rỗng, phải trả lại, từ đó nút cha của nó có thể có số con nhỏ hơn ngưỡng cho phép, trong trường hợp đó hoặc phải chuyển một con từ nút anh em của nút cha đó sang nút cha nếu điều đó có thể (nút anh em của nút cha này còn số con ≥ $\lceil m/2 \rceil$ sau khi chuyển đi một con). Nếu không, phải gom nút cha này với một nút anh em của nó, điều này dẫn tới xoá một nút trong khỏi cây, rồi xoá khỏi nút cha của nó một hạng, ... quá trình này có thể dẫn đến tận gốc. Trong trường hợp nút gốc chỉ còn một con sau xoá, cây phải thay nút gốc cũ bởi nút con của nó, nút gốc cũ phải trả lại cho hệ thống, chiều cao cây giảm đi một.

```
Procedure delete(value V, pointer P)
        Tìm nút lá chứa (V, P)
        delete_entry(L, V, P)
end procedure
Procedure delete_entry(node L, value V, pointer P)
        xoá (V, P) khỏi L
        If (L là nút gốc and L chỉ còn lại một con) then
                Lấy con của L làm nút gốc mới của cây, xoá L
        else If (L có quá ít giá trị/ con trỏ) then begin
                L' là anh em kề trái hoặc phải của L
                V' là giá trị ở giữa hai con trỏ L, L' (trong nút parent(L))
                If (các đầu vào của L và L' có thể lấp đầy trong một khối) then begin
                    If (L là nút trước của L') then wsap_variables(L, L')
                    If (L không là lá) then nối V' và tất cả con trỏ, giá trị trong L với L'
                    else  begin nối tất cả các cặp (K, P) trong L với L'; L'.Pp = L.Pp  end
```

```
            delete_entry(parent(L), V', L); xoá nút L
        end
        else begin
            If (L' là nút trước của L) then begin
                If (L không là nút lá) then begin
                    p là chỉ số sao cho L'.P_p là con trỏ cuối trong L'
                    xoá (L'.K_p-1, L'.P_p) khỏi L'
                    xen (L'.P_p, V') như phần tử đầu tiên trong L (right_shift tất cả các phần tử của L)
                    thay thế V' trong parent(L) bởi L'.K_p-1
                end else begin
                    p là chỉ số sao cho L'.P_p là con trỏ cuối trong L'
                    xoá (L'.P_p, L'.K_p) khỏi L'
                    xen (L'.P_p, L'.K_p) như phần tử đầu tiên trong L (right_shift tất cả các phần tử của L)
                    thay thế V' trong parent(L) bởi L'.K_p
                end
        end < đối xứng với trường hợp then >
    end
end procedure
```

Tổ chức file B^+-cây

Trong tổ chức file B^+-cây, các nút lá của cây lưu trữ các mẩu tin, thay cho các con trỏ tới file. Vì mẩu tin thường lớn hơn con trỏ, số tối đa các mẩu tin được lưu trữ trong một khối lá ít hơn số con trỏ trong một nút không lá. Các nút lá vẫn được yêu cầu được lấp đầy ít nhất là một nửa.

Xen và xoá trong tổ chức file B^+-cây tương tự như trong chỉ mục B^+-cây.

Khi B^+-cây được sử dụng để tổ chức file, việc sử dụng không gian là đặc biệt quan trọng, vì không gian bị chiếm bởi mẩu tin là lớn hơn nhiều so với không gian bị chiếm bởi (khoá,con trỏ). Ta có thể cải tiến sự sử sụng không gian trong B^+-cây bằng cách bao hàm nhiều nút anh em hơn khi tái phân phối trong khi tách và trộn. Khi xen, nếu một nút là đầy, ta thử phân phối lại một số đầu vào đến một trong các nút kề để tạo không gian cho đầu vào mới. Nếu việc thử này thất bại, ta mới thực hiện tách nút và phân chia các đầu vào giữa một trong các nút kề và hai nút nhận được do tách nút. Khi xoá, nếu nút chứa ít hơn $\lfloor 2m/3 \rfloor$ đầu vào, ta thử mượn một đầu vào từ một trong hai nút anh em kề. Nếu cả hai đều có đúng $\lfloor 2m/3 \rfloor$ mẩu tin, ta phân phối lại các đầu vào của nút cho hai nút anh em kề và xoá nút thứ 3. Nếu k nút được sử dụng trong tái phân phối (k-1 nút anh em), mỗi nút đảm bảo chứa ít nhất $\lfloor (k-1)m/k \rfloor$ đầu vào. Tuy nhiên, cái giá phải trả cho cập nhật của cách tiếp cận này sẽ cao hơn.

FILE CHỈ MỤC B-CÂY (B-Tree Index Files)

Chỉ mục B-cây tương tự như chỉ mục B^+-cây. Sự khác biệt là ở chỗ B-cây loại bỏ lưu trữ dư thừa các giá trị khoá tìm kiếm. Trong B-cây, các giá trị khoá chỉ xuất hiện một lần. Do các khoá tìm kiếm xuất hiện trong các nút không là lá không xuất hiện ở bất kỳ nơi nào khác nữa trong B-cây, ta phải thêm một trường con trỏ cho mỗi khoá tìm kiếm trong các nút không là lá. Con trỏ thêm vào này trỏ tới hoặc mẩu tin trong file hoặc bucket tương ứng.

Một nút lá B-cây tổng quát có dạng:

P_1	K_1	P_2	K_2	...	P_{m-1}	K_{m-1}	P_m

Một nút không là lá có dạng:

P_1	B_1	K_1	P_2	B_2	K_2		$P_{...}$	$B_{m-...}$	$K_{...}$	P

Các con trỏ P_i là các con trỏ cây và được dùng như trong B^+-cây. Các con trỏ B_i trong các nút không là lá là các con trỏ mẩu tin hoặc con trỏ bucket. Rõ ràng là số giá trị khoá trong nút không lá nhỏ hơn số giá trị trong nút lá. Số nút được truy xuất trong quá trình tìm kiếm trong một B-cây phụ thuộc nơi khoá tìm kiếm được định vị.

Xoá trong một B-cây phức tạp hơn trong một B^+-cây. Xoá một đầu vào xuất hiện ở một nút không là lá kéo theo việc tuyển chọn một giá trị thích hợp trong cây con của nút chứa đầu vào bị xoá. Nếu khoá K_i bị xoá, khoá nhỏ nhất trong cây con được trỏ bởi P_{i+1} phải được di chuyển vào vị trí của K_i. Nếu nút lá còn lại quá ít đầu vào, cần thiết các hoạt động bổ xung.

III.9.4 Định nghĩa chỉ mục trong SQL

Một chỉ mục được tạo ra bởi lệnh **CREATE INDEX** với cú pháp

CREATE INDEX < index-name > **ON** < relation_name > (< attribute-list >)

attribute-list là danh sách các thuộc tính của quan hệ được dùng làm khoá tìm kiếm cho chỉ mục. Nếu muốn khai báo là khoá tìm kiếm là khoá dự tuyển, thêm vào từ khoá UNIQUE:

CREATE UNIQUE INDEX < index-name > **ON** < relation_name > (< attribute-list >)

attribute-list phải tạo thành một khoá dự tuyển, nếu không sẽ có một thông báo lỗi.

Bỏ đi một chỉ mục sử dụng lệnh DROP:

DROP INDEX < index-name >

BĂM (HASHING)

BĂM TĨNH (Static Hashing)

Bất lợi của tổ chức file tuần tự là ta phải truy xuất một cấu trúc chỉ mục để định vị dữ liệu, hoặc phải sử dụng tìm kiếm nhị phân, và kết quả là có nhiều hoạt động I/O. Tổ chức file dựa trên kỹ thuật băm cho phép ta tránh được truy xuất một cấu trúc chỉ mục. Băm cung cung cấp một phương pháp để xây dựng các chỉ mục.

Tổ chức file băm

Trong tổ chức file băm, ta nhận được địa chỉ của khối đĩa chứa một mẩu tin mong muốn bởi tính toán một hàm trên giá trị khoá tìm kiếm của mẩu tin. thuật ngữ bucket được dùng để chỉ một đơn vị lưu trữ. Một bucket kiểu mẫu là một khối đĩa, nhưng có thể được chọn nhỏ hơn hoặc lớn hơn một khối đĩa.

K ký hiệu tập tất cả các giá trị khoá tìm kiếm, B ký hiệu tập tất cả các địa chỉ bucket. Một hàm băm h là một hàm từ K vào B : h: $K \rightarrow \mathrm{B}$

Xen một mẩu tin với giá trị khoá K vào trong file: ta tính h(K). Giá trị của h(K) là địa chỉ của bucket sẽ chứa mẩu tin. Nếu có không gian trong bucket cho mẩu tin, mẩu tin được lưu trữ trong bucket.

Tìm kiếm một mẩu tin theo giá trị khoá K: đầu tiên tính h(K), ta tìm được bucket tương ứng. sau đó tìm trong bucket này mẩu tin với giá trị khoá K mong muốn.

Xoá mẩu tin với giá trị khoá K: tính h(K), tìm trong bucket tương ứng mẩu tin mong muốn, xoá nó khỏi bucket.

Hàm băm

Hàm băm xấu nhất là hàm ánh xạ tất cả các giá trị khoá vào cùng một bucket. Hàm băm lý tưởng là hàm phân phối ***đều*** các giá trị khoá vào các bucket, như vậy mỗi bucket chứa một số lượng mẩu tin như nhau. Ta muốn chọn một hàm băm thoả mãn các tiêu chuẩn sau:

- ***Phân phối đều***: Mỗi bucket được gán cùng một số giá trị khoá tìm kiếm trong tập hợp tất cả các giá trị khoá có thể
- ***Phân phối ngẫu nhiên***: Trong trường hợp trung bình, các bucket được gán một số lượng giá trị khoá tìm kiếm *gần bằng nhau*.

Các hàm băm phải được thiết kế thận trọng. Một hàm băm xấu có thể dẫn đến việc tìm kiếm chiếm một thời gian tỷ lệ với số khoá tìm kiếm trong file.

Điều khiển tràn bucket

Khi xen một mẩu tin, nếu bucket tương ứng còn chỗ, mẩu tin được xen vào bucket, nếu không sẽ xảy ra tràn bucket. Tràn bucket do các nguyên do sau:

- **Các bucket không đủ.** Số các bucket n_B phải thoả mãn $n_B > n_r / f_r$ trong đó n_r là tổng số mẩu tin sẽ lưu trữ, f_r là số mẩu tin có thể lấp đầy trong một bucket.
- **Sự lệch.** Một vài bucket được gán cho một số lượng mẩu tin nhiều hơn các bucket khác, như vậy một bucket có thể tràn trong khi các bucket khác vẫn còn không gian. Tình huống này được gọi là sự lệch bucket. Sự lệch xảy ra do hai nguyên nhân:
  1. Nhiều mẩu tin có cùng khoá tìm kiếm
  2. Hàm băm được chọn phân phối các giá trị khoá không đều

Ta quản lý tràn bucket bằng cách dùng các bucket tràn. Nếu một mẩu tin phải được xen vào bucket B nhưng bucket B đã đầy, khi đó một bucket tràn sẽ được cấp cho B và mẩu tin được xen vào bucket tràn này. Nếu bucket tràn cũng đầy một bucket tràn mới lại được cấp và cứ như vậy. Tất cả các bucket tràn của một bucket được "móc xích" với nhau thành một danh sách liên kết. Việc điều khiển tràn dùng danh sách liên kết như vậy được gọi là dây chuyền tràn. Đối với dây chuyền tràn, thuật toán tìm kiếm thay đổi chú ít: trước tiên ta cũng tính giá trị hàm băm trên khoá tìm kiếm, ta được bucket B, kiểm tra các mẩu tin, trong bucket B và tất cả các bucket tràn tương ứng, có giá trị khoá khớp với giá trị tìm không.

Một cách điều khiển tràn bucket khác là: Khi cần xen một mẩu tin vào một bucket nhưng nó đã đầy, thay vì cấp thêm một bucket tràn, ta sử dụng một hàm băm kế trong một dãy các hàm băm được chọn để tìm bucket khác cho mẩu tin, nếu bucket sau cũng đầy, ta lại sử dụng một hàm băm kế và cứ như vậy... Dãy các hàm băm thường được sử dụng là { $h_i(K) = (h_{i-1}(K) + 1) \bmod n_B$ với $1 \le i \le n_B - 1$ và h_0 là hàm băm cơ sở }.

Dạng cấu trúc băm sử dụng dây chuyền bucket được gọi là băm mở. Dạng sử dụng dãy các hàm băm được gọi là băm đóng. Trong các hệ CSDL, cấu trúc băm đóng thường được ưa dùng hơn.

Chỉ mục băm

Một chỉ mục băm tổ chức các khoá tìm kiếm cùng con trỏ kết hợp vào một cấu trúc file băm như sau: áp dụng một hàm băm trên khoá tìm kiếm để định danh bucket sau đó lưu giá trị khoá và con trỏ kết hợp vào bucket này (hoặc vào các bucket tràn). Chỉ mục băm thường là chỉ mục thứ cấp.

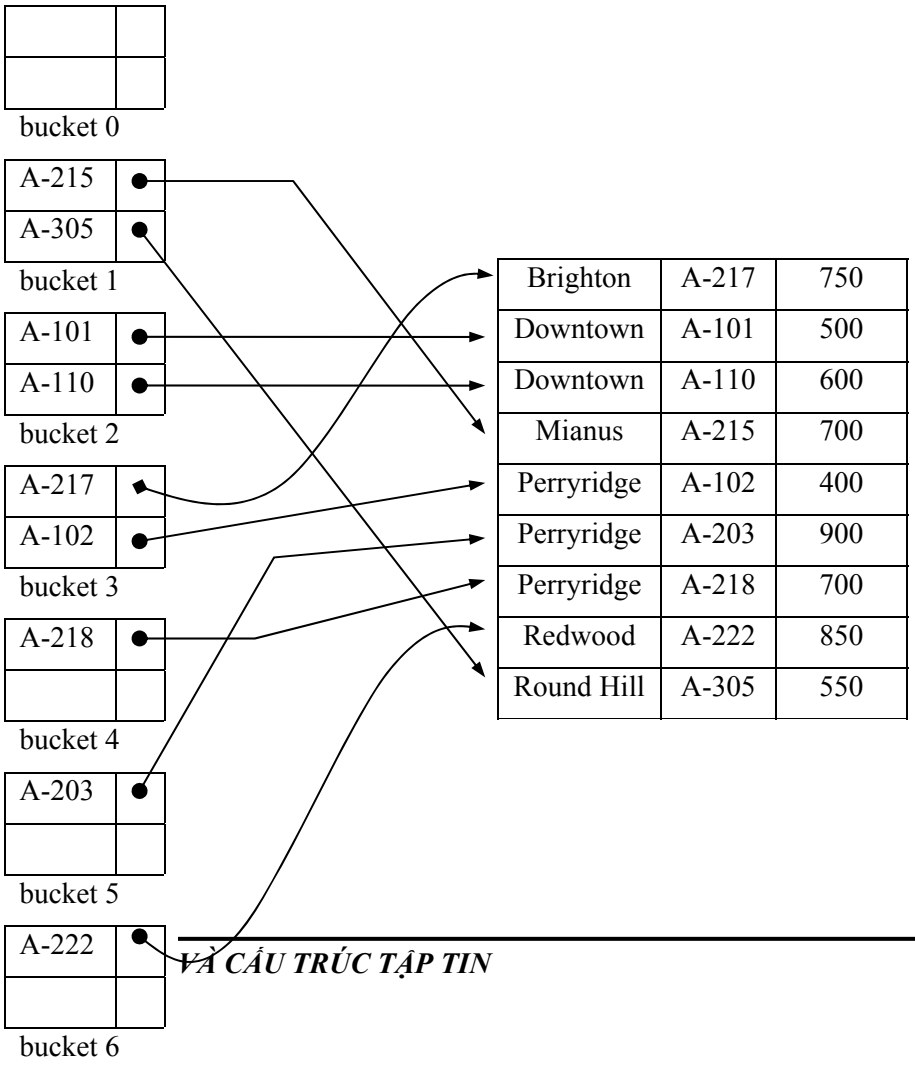
Hàm băm trên số tài khoản được tính theo công thức:

h(Account_number) = (tổng các chữ số trong số tài khoản) mod 7

BĂM ĐỘNG (Dynamic Hashing)

Trong kỹ thuật băm tĩnh (static hashing), tập B các địa chỉ bucket phải là cố định. Các CSDL phát triển lớn lên theo thời gian. Nếu ta sử dụng băm tĩnh cho CSDL, ta có ba lớp lựa chọn:

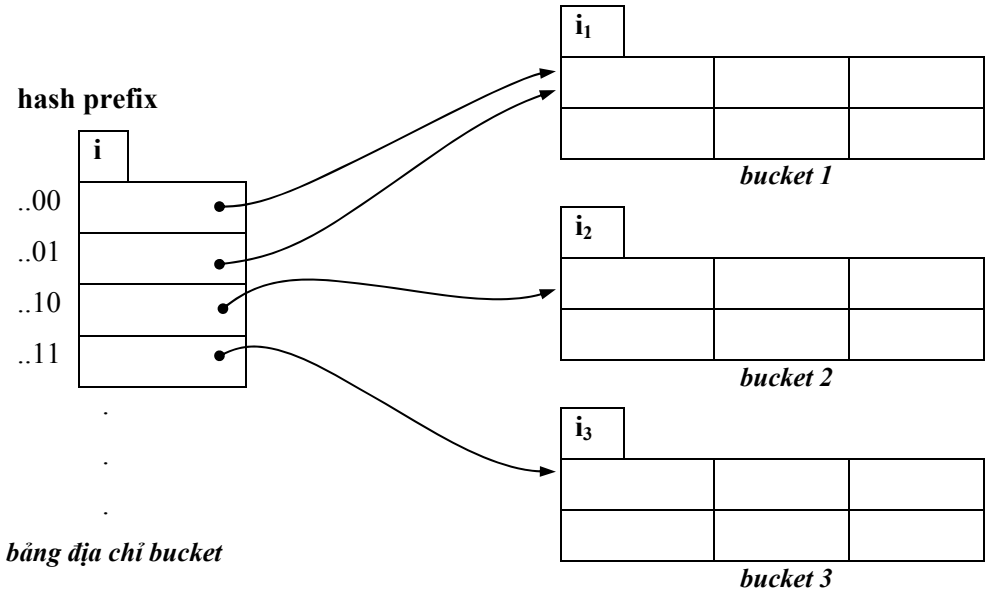
1. Chọn một hàm băm dựa trên kích cỡ file hiện hành. Sự lựa chọn này sẽ dẫn đến sự suy giảm hiệu năng khi CSDL lớn lên.
2. Chọn một hàm băm dựa trên kích cỡ file dự đoán trước cho một thời điểm nào đó trong tương lai. Mặc dù sự suy giảm hiệu năng được cải thiện, một lượng đáng kể không gian có thể bị lãng phí lúc khởi đầu.
3. Tổ chức lại theo chu kỳ cấu trúc băm đáp ứng sự phát triển kích cỡ file. Một sự tổ chức lại như vậy kéo theo việc lựa chọn một hàm băm mới, tính lại hàm băm trên mỗi mẩu tin trong file và sinh ra các gán bucket mới. Tổ chức lại là một hoạt động tốn thời gian. Hơn nữa, nó đòi hỏi cấm truy xuất file trong khi đang tổ chức lại file.



Chỉ mục băm trên khoá tìm kiếm account-number của file account

Kỹ thuật băm động cho phép sửa đổi hàm băm để phù hợp với sự tăng hoặc giảm của CSDL. Một dạng băm động được gọi là băm có thể mở rộng (extendable hashing) được thực hiện như sau: Chọn một hàm băm h với các tính chất đều, ngẫu nhiên và có miền giá trị tương đối rộng, chẳng hạn, là một số nguyên b bit (b thường là 32). Khi khởi đầu ta không sử dụng toàn bộ b bit giá trị băm. Tại một thời điểm, ta chỉ sử dụng i bit $0 \le i \le b$. i bit này được dùng như một độ dời (offset) trong một bảng địa chỉ bucket phụ. giá trị i tăng lên hay giảm xuống tuỳ theo kích cỡ CSDL.

Số i xuất hiện bên trên bảng địa chỉ bucket chỉ ra rằng i bit của giá trị băm h(K) được đòi hỏi để xác định bucket đúng cho K, số này sẽ thay đổi khi kích cỡ file thay đổi. Mặc dù i bit dược đòi hỏi để tìm đầu vào đúng trong bảng địa chỉ bucket, một số đầu vào bảng kề nhau có thể trỏ đến cùng một bucket. Tất cả các như vậy có chung hash prefix chung, nhưng chiều dài của prefix này có thể nhỏ hơn i. Ta kết hợp một số nguyên chỉ độ dài của hash prefix chung này, ta sẽ ký hiệu số nguyên kết hợp với bucket j là i_j. Số các đầu vào bảng địa chỉ bucket trỏ đến bucket j là $2^{(i-i_j)}$.



Cấu trúc băm có thể mở rộng tổng quát

Để định vị bucket chứa giá trị khoá tìm kiếm K , ta lấy i bit cao đầu tiên của h(K), tìm trong đầu vào bảng tương ứng với chuỗi bit này và lần theo con trỏ trong đầu vào bảng này. Để xen một mẩu tin với giá trị khoá tìm kiếm K, tiến hành thủ tục dịnh vị trên, ta được bucket, giả sử là bucket j. Nếu còn cho cho mẩu tin, xen mẩu tin vào trong bucket đó. Nếu không, ta phải tách bucket ra và phân phối lại các mẩu tin hiện có cùng mẩu tin mới. Để tách bucket, đầu tiên ta xác định từ giá trị băm có cần tăng số bit lên hay không.

- Nếu $i = i_j$, chỉ có một đầu vào trong bảng địa chỉ bucket trỏ đến bucket j. ta cần tăng kích cỡ của bảng địa chỉ bucket sao cho ta có thể bao hàm các con trỏ đến hai bucket kết quả

của việc tách bucket j bằng cách xét thêm một bit của giá trị băm. tăng giá trị i lên một, như vậy kích cỡ của bảng địa chỉ bucket tăng lên gấp đôi. Mỗi một đầu vào được thay bởi hai đầu vào, cả hai cùng chứa con trỏ của đầu vào gốc. Bây giờ hai đầu vào trong bảng địa chỉ bucket trỏ tới bucket j. Ta định vị một bucket mới (bucket z), và đặt đầu vào thứ hai trỏ tới bucket mới, đặt i_j và i_z về i, tiếp theo đó mỗi một mẩu tin trong bucket j được băm lại, tuỳ thuộc vào i bit đầu tiên, sẽ hoặc ở lại bucket j hoặc được cấp phát cho bucket mới được tạo.

- Nếu $i > i_j$ khi đó nhiều hơn một đầu vào trong bảng địa chỉ bucket trỏ tới bucket j. như vậy ta có thể tách bucket j mà không cần tăng kích cỡ bảng địa chỉ bucket. Ta cấp phát một bucket mới (bucket z) và đặt i_j và i_z đến giá trị là kết quả của việc thêm 1 vào giá trị i_j gốc. Kế đến, ta điều chỉnh các đầu vào trong bảng địa chỉ bucket trước đây trỏ tới bucket j. Ta để lại nửa đầu các đầu vào, và đặt tất cả các đầu vào còn lại trỏ tới bucket mới tạo (z). Tiếp theo, mỗi mẩu tin trong trong bucket j được băm lại và được cấp phát cho hoặc vào bucket j hoặc bucket z.

Để xoá một mẩu tin với giá trị khoá K, trước tiên ta thực hiện thủ tục định vị, ta tìm được bucket tương ứng, gọi là j, ta xoá cả khoá tìm kiếm trong bucket lẫn mẩu tin mẩu tin trong file. bucket cũng bị xoá, nếu nó trở nên rỗng. Chú ý rằng, tại điểm này, một số bucket có thể được kết hợp lại và kích cỡ của bảng địa chỉ bucket sẽ giảm đi một nửa.

Ưu điểm chính của băm có thể mở rộng là hiệu năng không bị suy giảm khi file tăng kích cỡ, hơn nữa, tổng phí không gian là tối tiểu mặc dù phải thêm vào không gian cho bảng địa chỉ bucket. Một khuyết điểm của băm có thể mở rộng là việc tìm kiếm phải bao hàm một mức gián tiếp: ta phải truy xuất bảng địa chỉ bucket trước khi truy xuất đến bucket. Vì vậy, băm có thể mở rộng là một kỹ thuật rất hấp dẫn.

CHỌN CHỈ MỤC HAY BĂM ?

Ta đã xét qua các sơ đồ: chỉ mục thứ tự, băm. Ta có thể tổ chức file các mẩu tin bởi hoặc sử dụng tổ chức file tuần tự chỉ mục, hoặc sử dụng B^+-cây, hoặc sử dụng băm ... Mỗi sơ đồ có những các ưu điểm trong các tình huống nhất định. Một nhà thực thi hệ CSDL có thể cung cấp nhiều nhiều sơ đồ, để lại việc quyết định sử dụng sơ đồ nào cho nhà thiết kế CSDL. Để có một sự lựa chọn khôn ngoan, nhà thực thi hoặc nhà thiết kế CSDL phải xét các yếu tố sau:

- Cái giá phải trả cho việc tổ chức lại theo định kỳ của chỉ mục hoặc băm có chấp nhận được hay không?
- Tần số tương đối của các hoạt động xen và xoá là bao nhiêu ?
- Có nên tối ưu hoá thời gian truy xuất trung bình trong khi thời gian truy xuất trường hợp xấu nhất tăng lên hay không ?
- Các kiểu vấn tin mà các người sử dụng thích đặt ra là gì ?

CẤU TRÚC LƯU TRỮ CHO CSDL HƯỚNG ĐỐI TƯỢNG

SẮP XẾP CÁC ĐỐI TƯỢNG VÀO FILE

Phần dữ liệu của đối tượng có thể được lưu trữ bởi sử dụng các cấu trúc file được mô tả trước đây với một số thay đổi do đối tượng có kích cỡ không đều, hơn nữa đối tượng có thể rất lớn. Ta có thể thực thi các trường tập hợp ít phần tử bằng cách sử dụng danh sách liên kết, các trường tập hợp nhiều phần tử bởi B-cây hoặc bởi các quan hệ riêng biệt trong cơ sở dữ liệu. Các trường tập hợp cũng có thể bị loại trừ ở mức lưu trữ bởi chuẩn hoá. Các đối tượng cực lớn khó có

thể phân tích thành các thành phần nhỏ hơn có thể được lưu trữ trong một file riêng cho mỗi đối tượng.

THỰC THI ĐỊNH DANH ĐỐI TƯỢNG

Vì đối tượng được nhận biết bởi định danh của đối tượng (OID = objject Identifier), Một hệ lưu trữ đối tượng cần phải có một cơ chế để tìm kiếm một đối tượng được cho bởi một OID. Nếu các OID là logic, có nghĩa là chúng không xác định vị trí của dối tượng, hệ thống lưu trữ phải duy trì một chỉ mục mà nó ánh xạ OID tới vị trí hiện hành của đối tượng. Nếu các OID là vật lý, có nghĩa là chúng mã hoá vị trí của đối tượng, đối tượng có thể dược tìm trực tiếp. Các OID điển hình có ba trường sau:

1. Một volume hoặc định danh file
2. Một định danh trang bên trong volume hoặc file
3. Một offset bên trong trang

Hơn nữa, OID vật lý có thẻ chứa một định danh duy nhất, nó là một số nguyên tách biệt OID với các định danh của các đối tượng khác đã được lưu trữ ở cùng vị trí trước đây và đã bị xoá hoặc dời đi. Định danh duy nhất này cũng được lưu với đối tượng, các định danh trong một OID và đối tượng tương ứng phù hợp. Nếu định danh duy nhất trong một OID vật lý không khớp với với định danh duy nhất trong đối tượng mà OID này trỏ tới, hệ thống phát hiện ra rằng con trỏ là bám và báo một lỗi. Lỗi con trỏ như vậy xảy ra khi OID vật lý tương ứng với đối tượng cũ đã bị xoá do tai nạn. Nếu không gian bị chiếm bởi đối tượng được cấp phát lại, có thể có một đối tượng mới ở vào vị trí này và có thể được định địa chỉ không đúng bởi định danh của đối tượng cũ. Nếu không phát hiện được, sử dụng con trỏ bám có thể gây nên sự sai lạc của một đối tượng mới được lưu ở cùng vị trí. Định danh duy nhất trợ giúp phát hiện lỗi như vậy. Giả sử một đối tượng phải di chuyển sang trang mới do sự lớn lên của đối tượng và trang cũ không có không gian phụ. Khi đó OID vật lý trỏ tới trang cũ bây giờ không còn chứa đối tượng. Thay vì thay đổi OID của đối tượng (điều này kéo theo sự thay đổi mỗi đối tượng trỏ tới đối tượng này) ta để địa chỉ forward ở vị trí cũ. Khi CSDL tìm đối tượng, nó tìm địa chỉ forward thay cho tìm đối tượng và sử dụng địa chỉ forward để tìm đối tượng.

QUẢN TRỊ CÁC CON TRỎ BỀN (persistent pointers)

Ta thực thi các con trỏ bền trong ngôn ngữ lập trình bền (persistent programming language) bằng cách sử dụng các OID. Các con trỏ bền có thể là các OID vật lý hoặc logic. Sự khác nhau quan trọng giữa con trỏ bền và con trỏ trong bộ nhớ là kích thước concủa con trỏ. Con trỏ trong bộ nhớ chỉ cần đủ lớn để định địa chỉ toàn bộ bộ nhớ ảo, hiện tại kích cỡ con trỏ trong bộ nhớ là 4 byte. Con trỏ bền để định địa chỉ toàn bộ dữ liệu trong một CSDL, nên kích cỡ của nó ít nhất là 8 byte.

Pointer Swizzling

Hành động tìm một đối tượng được cho bởi định danh được gọi là *dereferencing*. Đã cho một con trỏ trong bộ nhớ, tìm đối tượng đơn thuần là một sự tham khảo bộ nhớ. Đã cho một con trỏ bền, dereferencing một đối tượng có một bước phụ: phải tìm vị trí hiện hành của đối tượng trong bộ nhớ bởi tìm con trỏ bền trong một bảng. Nếu đối tượng chưa nằm trong bộ nhớ, nó phải được nạp từ đĩa. Ta có thể thực thi bảng tìm kiếm này hoàn toàn hiệu quả bởi sử dụng băm, song tìm kiếm vẫn chậm.

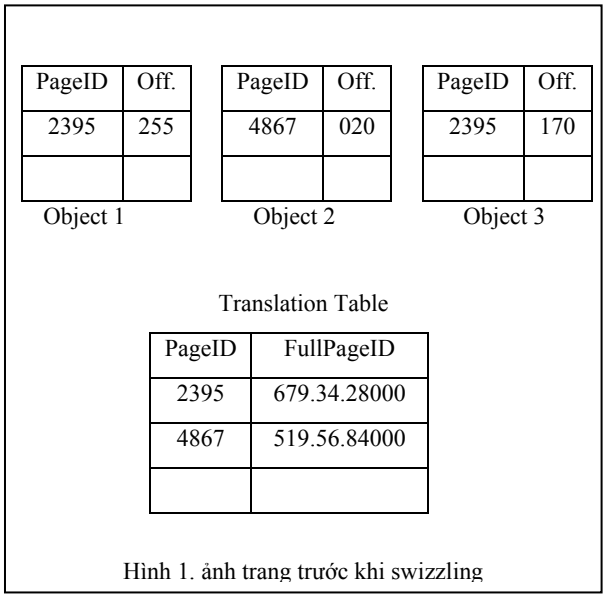
pointer swizling là một phương pháp để giảm cái giá tìm kiếm các đối tượng bền đã hiện diện trong bộ nhớ. ý tưởng là khi một con trỏ bền được dereference, đối tượng được định vị và mang vào trong bộ (nhớ nếu nó chưa có ở đó). Bây giờ một bước phụ được thực hiện: một con trỏ trong bộ nhớ tới đối tượng được lưu vào vị trí của con trỏ bền. Lần kế con trỏ bền tương tự được dereference, vị trí trong bộ nhớ có thể được đọc ra trực tiếp. Trong trường hợp các đối tượng bền phải di chuyển lên đĩa để lấy không gian cho đối tượng bền khác, cần một bước phụ để đảm bảo đối tượng vẫn trong bộ nhớ cũng phải được thực hiện. Khi một đối tượng được viết ra. bất kỳ con trỏ bền nào mà nó chứa và bị swizzling phải được unswizzling như vậy được chuyển đổi về biểu diễn bền của chúng. pointer swizzling

trên poiter dereferenc được mô tả này được gọi là software swizzling. Quan trị buffer sẽ phức tạp hơn nếu pointer swizzling được sử dụng.

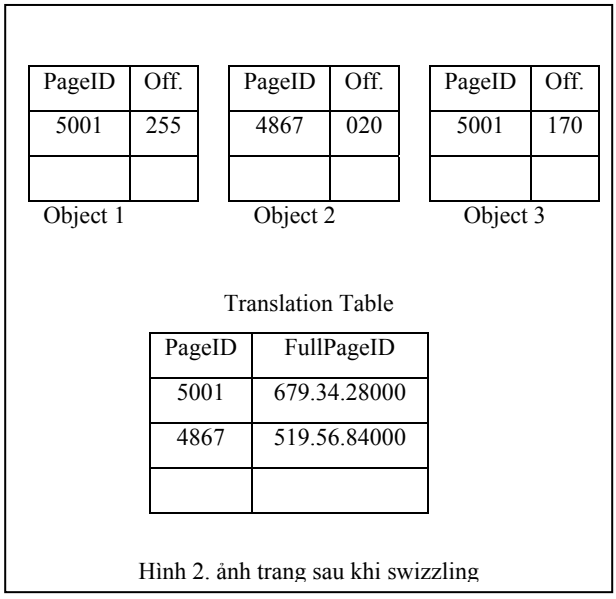
Hardware swizzling

Việc có hai kiểu con trỏ, con trỏ bền (persistent pointer) và con trỏ tạm (transient pointer / con trỏ trong bộ nhớ), là điều khá bất lợi. Người lập trình phải nhớ kiểu con trỏ và có thể phải viết mã chương trình hai lần- một cho các con trỏ bền và một cho con trỏ tạm. Sẽ thuận tiên hơn nếu cả hai kiểu con trỏ này cùng kiểu. Một cách đơn giản để trộn lãn hai con rỏ này là mở rộng chiều dài con trỏ bộ nhớ cho bằng kích cỡ con trỏ bền và sử dụng một bit của phần định danh để phân biệt chúng. Cách làm này sẽ làm tăng chi phí lưu trữ đối với các con trỏ tạm. Ta sẽ mô tả một kỹ thuật được gọi là hardware swizzling nó sử dụng phần cứng quản trị bộ nhớ để giải quyết vấn đề này. Hardware swizzling có hai điểm lợi hơn so với software swizzling: Thứ nhất, nó cho phép lưu trữ các con trỏ bền trong đối tượng trong lượng không gian bằng với lượng không gian con trỏ bộ nhớ đòi hỏi. Thứ hai, nó chuyển đổi trong suốt giữa các con trỏ bền và các con trỏ tạm một cách thông minh và hiệu quả. Phần mềm được viết để giải quyết các con trỏ trong bộ nhớ có thể giải quyết các con trỏ bền mà không cần thay đổi.

hardware swizzling sử dụng sự biểu diễn các con trỏ bền được chứa trong đối tượng trên đĩa như sau: Một con trỏ bền được tách ra thành hai phần, một là định danh trang và một là offset bên trong trang. Định danh trang thường là một con trỏ trực tiếp nhỏ: mỗi trưng có một bảng dịch (translation table) cung cấp một ánh xạ từ các định danh trang ngắn đến các định danh CSDL đầy đủ. Hệ thống phải tìm định danh trang nhỏ trong một con trỏ bền trong bảng dịch để tìm định danh trang đầy đủ. Bảng dịch, trong trường hợp xấu nhất, chỉ lớn bằng số tối đa các con trỏ có thể được chứa trong các đối tượng trong một trang. Với một trang kích thước 4096 byte, con trỏ kích thước 4 byte, số tối đa các con trỏ là 1024. Trong thực tế số tối đa nhỏ hơn con số này rất nhiều. Định danh trang nhỏ chỉ cần đủ số bit để định danh một dòng trong bảng, nếu số dòng tối đa là 1024, chỉ cần 10 bit để định danh trang nhỏ. Bảng dịch cho phép toàn bộ một con trỏ bền lấp đầy một không gian bằng không gian cho một con trỏ trong bộ nhớ.



Hình 1. ảnh trang trước khi swizzling



Hình 2. ảnh trang sau khi swizzling

Trong hình 1, trình bày sơ đồ biểu diễn con trỏ bền, có ba đối tượng trong trang, mỗi một chứa một con trỏ bền. Bảng dịch cho ra ánh xạ giữa định danh trang ngắn và định danh trang CSDL đầy đủ đối với mỗi định danh trang ngắn trong các con trỏ bền này. Định danh trang CSDL được trình bày dưới dạng ***volume.file.offset***. Thông tin phụ được duy trì với mỗi trang sao cho tất cảc các con trỏ bền trong trang có thể tìm thấy. Thông tin được cập nhất khi một đối tượng được tạo ra hay bị xoá khỏi trang. Khi một con trỏ trong bộ nhớ được dereferencing, nếu hệ điều hành phát hiện trang trong không gian địa chỉ ảo được trỏ tới không được cấp phát lưu trữ hoặc có truy xuất được bảo vệ, khi đó một sự vi phạm đoạn được ước đoán là xảy ra. Nhiều hệ điều hành cung cấp một cơ chế xác định một hàm sure được gọi khi vi phạm đoạn xảy ra, một cơ chế cấp phát lưu trữ cho các trang trong không gian địa chỉ ảo, và một tập các quyền truy xuất trang. Đầu tiên, ta xét một con trỏ trong bộ nhớ trỏ tới trang v được khử tham chiếu, khi lưu trữ chưa được cấp phát cho trang này. Một vi phạm đoạn sẽ xảy ra và kết quả là một lời gọi hàm trên hệ CSDL. Hệ CSDL dầu tiên xác định trang CSDL nào đã được cấp phát cho trang bộ nhớ ảo v, gọi định danh trang đầy đủ của trang CSDL là P, nếu không có trang CSDL cấp phát cho v, một lỗi được thông báo., nếu không, hệ CSDL cấp phát không gian lưu trữ cho trang v và nạp trang CSDL P vào trong v. Pointer swizzling bây giờ được làm đối với trang P như sau: Hệ thông tìm tất cả các con trỏ bèn được chứa trong các đối tương trong trang, bằng cách sử dụng thông tin phụ được lưu trữ trong trang. Ta xét một con trỏ như vậy và gọi nó là (p_i, o_i), trong đó p_i là định danh trang ngắn và

o_i là offset trong trang. Giả sử P_i là định danh trang đầy đủ của p_i được tìm thấy trong bảng dịch trong trang P. Nếu trang P_i chưa có một trang bộ nhớ ảo được cấp cho nó, một trang tự do trong không gian địa chỉ ảo sẽ được cấp cho nó. Trang P_i sẽ nằm ở vị trí địa chỉ ảo nàynếu và khi nó được mang vào. Tại điểm này, trang trong không gian địa chỉ ảo không có bất kỳ một lưu trữ nào được cấp cho nó, cả trong bộ nhớ lẫn trên đĩa, đơn tuần chỉ là một khoảng địa chỉ dự trữ cho trang CSDL. Bây giờ giả sử trang bộ nhớ ảo đã được cấp phát cho P_i là v_i . Ta cập nhật con trỏ (p_i, o_i) bởi thay thế p_i bởi v_i , cuối cùng sau khi swizzling tất cả các con trỏ bền trong P, sự khử tham chiếu gây ra vi phạm đoạn được cho phép tiếp tục và sẽ tìm thấy đối tượng đang được tìm kiếm trong bộ nhớ.

Trong hình 2, trình bày trạng thái trang trong hình 1 sau khi trang này được mang vào trong bộ nhớ và các con trỏ trong nó đã được swizzling. ở đây ta giả thiết trang định danh trang CSDL của nó là 679.34.28000 được ánh xạ đến trang 5001 trong bộ nhớ, trong khi trang định danh của nó là 519.56.84000 được ánh xạ đến trang 4867. Tất cả các con trỏ trong đối tượng đã được cập nhật để phản ánh tương ứng mới và bây giờ có thể được dùng như con trỏ trong bộ nhớ. ở cuối của giai đoạn dịch đối với một trang, các đối tượng trong trang thoả mãn một tính chất quan trọng: Tất cả các con trỏ bền được chứa trong đối tượng trong trang được chuyển đổi thành các con trỏ trong bộ nhớ.

CÂU HỎI VÀ BÀI TẬP CHƯƠNG III

III.1 Xét sự sắp xếp các khối dữ liệu và các khối parity trên bốn đĩa sau:

Đĩa 1	Đĩa 2	Đĩa 3	Đĩa 4
B_1	B_2	B_3	B_4
P_1	B_5	B_6	B_7
B_8	P_2	B_9	B_{10}
⋮	⋮	⋮	⋮

Trong đó các B_i biểu diễn các khối dữ liệu, các khối P_i biểu diễn các khối parity. Khối P_i là khối parity đối với các khối dữ liệu B_{4i-3} , B_{4i-2} , B_{4i-1} , B_{4i} . Hãy nêu các vấn đề gặp phải của cách sắp xếp này.

III.2 Một sự mất điện xảy ra trong khi một khối đang được viết sẽ dẫn tới kết quả là khối đó có thể chỉ được viết một phần. Giả sử rằng khối được viết một phần có thể phát hiện được. Một viết khối nguyên tử là hoặc toàn bộ khối được viết hoặc không có gì được viết (không có khối được viết một phần). Hãy đề nghị những sơ đồ để có được các viết khối nguyên tử hiệu quả trên các sơ đồ RAID:

1. Mức 1 (mirroring)
2. Mức 5 (block interleaved, distributed parity)

III.3 Các hệ thống RAID tiêu biểu cho phép thay thế các đĩa hư không cần ngưng truy xuất hệ thống. Như vậy dữ liệu trong đĩa bị hư sẽ phải được tái tạo và viết lên đĩa thay thế trong khi hệ thống vẫn tiếp tục hoạt động. Với mức RAID nào thời lượng giao thoa giữa việc tái tạo và các truy xuất đĩa còn đang chạy là ít nhất ? Giải thích.

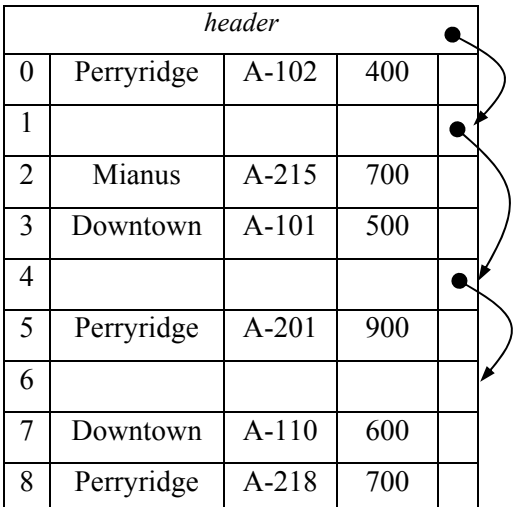
III.4 Xét việc xoá mẩu tin 5 trong file:

0	**Perryridge**	A-102	400
1	Round Hill	A-305	350
8	Perryridge	A-218	700
3	Downtown	A-101	500
4	Redwood	A-222	700
5	Perryridge	A-201	900
6	Brighton	A-217	750
7	Downtown	A-110	600

So sánh các điều hay/dở tương đối của các kỹ thuật xoá sau:

1. Di chuyển mẩu tin 6 đến không gian chị chiếm bởi mẩu tin 5, rồi di chuyển mẩu tin 7 đến chỗ bị chiếm bởi mẩu tin 6.
2. Di chuyển mẩu tin 7 đến chỗ bị chiếm bởi mẩu tin 5
3. Đánh dấu xoá mẩu tin 5.

III.5 Vẽ cấu trúc của file:



Sau mỗi bước sau:

1. Insert(Brighton, A-323, 1600)
2. Xoá mẩu tin 2
3. Insert(Brighton, A-636, 2500)

III.6 Vẽ lại cấu trúc file:

0	Perryridge	A-102	400	A-201	900	A210	700	⊥
1	Round Hill	A-301	350	⊥				
2	Mianus	A-101	800	⊥				
3	Downtown	A-211	500	A-222	600	⊥		
4	Redwood	A-300	650	A-200	1200	A-255	950	⊥
5	Brighton	A-111	750	⊥				

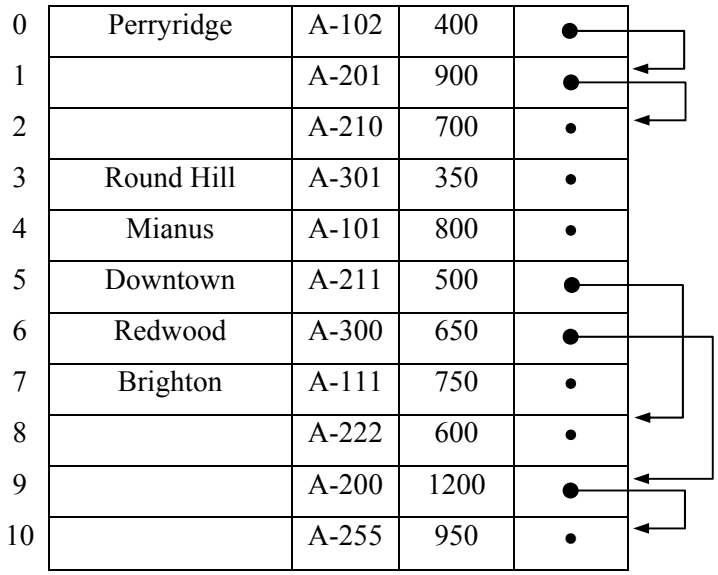
Sau mỗi bước sau:

1. Insert(Mianus, A-101, 2800)
2. Insert(Brighton, A-323, 1600)
3. Delete (Perryridge, A-102, 400)

III.7 Điều gì sẽ xảy ra nếu xen mẩu tin (Perryridge, A-999, 5000) vào file trong **III.6.**

III.8 Vẽ lại cấu trúc file dưới đây sau mỗi bước sau:

1. Insert(Mianus, A-101, 2800
2. Insert(Brighton, A-323, 1600)
3. Delete (Perryridge, A-102, 400)



(• = con trỏ nil)

III.9 Nêu lên một ví dụ, trong đó phương pháp không gian dự trữ để biểu diễn các mẩu tin độ dài thay đổi phù hợp hơn phương pháp con trỏ.

III.10 Nêu lên một ví dụ, trong đó phương pháp con trỏ để biểu diễn các mẩu tin độ dài thay đổi phù hợp hơn phương pháp không gian dự trữ.

III.11 Nếu một khối trở nên rỗng sau khi xoá. Khối này được tái sử dụng vào mục đích gì ?

III.12 Trong tổ chức file tuần tự, tại sao khối tràn được sử dụng thậm chí, tại thời điểm đang xét, chỉ có một mẩu tin tràn ?

III.13 Liệt kê các ưu điểm và nhược điểm của mỗi một trong các chiến lược lưu trữ CSDL quan hệ sau:

1. Lưu trữ mỗi quan hệ trong một file
2. Lưu trữ nhiều quan hệ trong một file

III.14 Nêu một ví dụ biểu thức đại số quan hệ và một chiến lược xử lý vấn tin trong đó:

1. MRU phù hợp hơn LRU
2. LRU phù hợp hơn MRU

III.15 Khi nào sử dụng chỉ mục đặc phù hợp hơn chỉ mục thưa ? Giải thích.

III.16 Nêu các điểm khác nhau giữa chỉ mục sơ cấp và chỉ mục thứ cấp .

III.17 Có thể có hai chỉ mục sơ cấp đối với hai khoá khác nhau trên cùng một quan hệ ? Giải thích.

III.18 Xây dựng một B^+-cây đối với tập các giá trị khoá: (2, 3, 5, 7, 11, 15, 19, 25, 29, 33, 37, 41, 47). Giả thiết ban đầu cây là rỗng và các giá trị được xen theo thứ tự tăng. Xét trong các trường hợp sau:

1. Mỗi nút chứa tối đa 4 con trỏ
2. Mỗi nút chứa tối đa 6 con trỏ
3. Mỗi nút chứa tối đa 8 con trỏ

III.19 Đối với mỗi B^+-cây trong bài tập **III.18** Bày tỏ các bước thực hiện trong các vấn tin sau:

1. Tìm mẩu tin với giá trị khoá tìm kiếm 11
2. Tìm các mẩu tin với giá trị khoá nằm trong khoảng [7..19]

III.20 Đối với mỗi B^+-cây trong bài tập **III.18.** Vẽ cây sau mỗi một trong dãy hoạt động sau:

1. Insert 9
2. Insert 11
3. Insert 11
4. Delete 25
5. Delete 19

III.21 Cùng câu hỏi như trong **III.18** nhưng đối với B-cây

III.22 Nêu và giải thích sự khác nhau giữa băm đóng và băm mở. Nêu các ưu, nhược điểm của mỗi kỹ thuật này.

III.23 Điều gì gây ra sự tràn bucket trong một tổ chức file băm ? Làm gì để giảm sự tràn này ?

III.24 Giả sử ta đang sử dụng băm có thể mở rộng trên một trên một file chứa các mẩu tin với các giá trị khoá tìm kiếm sau:

2, 3, 5, 7, 11, 17, 19, 23, 37, 31, 35, 41, 49, 55

Vẽ cấu trúc băm có thể mở rộng đối với file này nếu hàm băm là **h(x) = x mod 8** và mỗi bucket có thể chứa nhiều nhất được ba mẩu tin.

III.25 Vẽ lại cấu trúc băm có thể mở rộng trong bài tập **III.24** sau mỗi bước sau:

1. Xoá 11
2. Xoá 55
3. Xen 1

4. Xen 15