# Bài tập Chương 8

## Bài tập 8.1

* Thích ứng kỹ thuật phân mảnh/sao chép: Trong cluster shared-disk, việc phân mảnh (ví dụ: range hoặc hash partitioning) được sử dụng để phân chia một quan hệ lớn trên nhiều đơn vị đĩa vật lý nhằm cho phép quét song song (parallel I/O). Sao chép có thể được thực hiện ở cấp độ đĩa (ví dụ: RAID) hoặc bằng cách duy trì các bản sao logic trên các bộ đĩa khác nhau.
* Tác động đến hiệu năng: Hiệu năng cho các truy vấn quét toàn bộ (full scan) được cải thiện đáng kể do tăng băng thông I/O. Cân bằng tải dễ dàng hơn shared-nothing vì bất kỳ node nào cũng có thể xử lý bất kỳ dữ liệu nào. Tuy nhiên, tranh chấp trên mạng kết nối đến hệ thống đĩa dùng chung có thể trở thành một nút thắt cổ chai.
* Tác động đến khả năng chịu lỗi: Rất cao. Nếu một node xử lý bị lỗi, một node khác có thể tiếp quản công việc ngay lập tức vì dữ liệu vẫn có thể truy cập được trên đĩa dùng chung.

## Bài tập 8.2

* Thuật toán sắp xếp song song dùng băm bảo toàn thứ tự:
  1. Giai đoạn Phân mảnh: Áp dụng một hàm băm bảo toàn thứ tự (order-preserving hash function) trên thuộc tính cần sắp xếp của quan hệ. Gửi mỗi tuple đến node tương ứng với giá trị băm của nó. Kết quả là n phân mảnh, trong đó tất cả các tuple trong phân mảnh i có giá trị khóa sắp xếp nhỏ hơn tất cả các tuple trong phân mảnh j (với i < j).
  2. Giai đoạn Sắp xếp cục bộ: Mỗi node thực hiện sắp xếp phân mảnh cục bộ của mình một cách độc lập và song song (ví dụ: dùng quicksort).
  3. Giai đoạn Ghép nối: Kết quả cuối cùng được tạo ra bằng cách ghép nối (concatenate) các phân mảnh đã được sắp xếp từ các node theo đúng thứ tự.
* Ưu điểm so với b-way merge sort: Không cần giai đoạn trộn (merge) cuối cùng, vốn có thể là một nút thắt cổ chai tuần tự.
* Hạn chế: Rất nhạy cảm với dữ liệu bị lệch (data skew). Nếu dữ liệu không phân bố đều, một số node sẽ nhận được nhiều dữ liệu hơn các node khác, dẫn đến mất cân bằng tải nghiêm trọng. Hiệu quả của thuật toán phụ thuộc hoàn toàn vào chất lượng của hàm băm bảo toàn thứ tự.

## Bài tập 8.3

* Build Phase (Giai đoạn Xây dựng): Quan hệ bên trong (thường là quan hệ nhỏ hơn) được băm trên thuộc tính nối. Các tuple của nó được gửi đến một tập các node xử lý. Mỗi node trong số này xây dựng một bảng băm (hash table) trong bộ nhớ cho các tuple mà nó nhận được.
* Probe Phase (Giai đoạn Dò tìm): Quan hệ bên ngoài (thường là quan hệ lớn hơn) được băm trên thuộc tính nối bằng cùng một hàm băm. Các tuple của nó được gửi đến các node xử lý tương ứng. Mỗi node sau đó dò tìm trong bảng băm đã xây dựng bằng các tuple của quan hệ bên ngoài để tìm các cặp nối.
* Tính đối xứng: Thuật toán nối băm song song **không đối xứng**. Vai trò của quan hệ bên trong (build) và bên ngoài (probe) là khác nhau và không thể hoán đổi trong quá trình thực thi.

## Bài tập 8.4

Thuật toán Parallel Hash Join được sửa đổi như sau:

1. Build Phase: Giai đoạn này được bỏ qua cho quan hệ R. Thay vào đó, mỗi node chứa một phân mảnh của R sẽ xây dựng một bảng băm cục bộ trên phân mảnh đó.
2. Probe Phase: Quan hệ S được băm trên thuộc tính nối bằng cùng một hàm băm đã được sử dụng để phân mảnh R. Các tuple của S được gửi đến các node tương ứng nơi chứa các phân mảnh của R. Mỗi node sau đó dò tìm bảng băm cục bộ của mình bằng các tuple S đến để tạo kết quả nối.

Chi phí thực thi: Chi phí được giảm đáng kể vì không có chi phí truyền thông để di chuyển quan hệ R và giai đoạn xây dựng bảng băm cho R được thực hiện song song tại chỗ.

## Bài tập 8.5

Giả sử |R| và |S| là kích thước (số tuple) của các quan hệ, m và n là số node chứa R và S.

* Parallel Nested Loop (PNL):
  + CCOM = m \* msg(|R|/m) \* n (Gửi toàn bộ R đến n node của S)
  + CPRO = n \* CLOC(|R|, |S|/n)
  + Nên dùng khi một trong hai quan hệ rất nhỏ.
* Parallel Sort-Merge Join:
  + CCOM = m \* msg(|R|/m) + n \* msg(|S|/n) (Giai đoạn sắp xếp lại phân mảnh)
  + CPRO = m\*CLOC\_sort(|R|/m) + n\*CLOC\_sort(|S|/n) + p\*CLOC\_merge(...) (p là số node nối)
  + Nên dùng khi kết quả cần được sắp xếp.
* Parallel Hash Join (PHJ):
  + CCOM = m \* msg(|R|/m) + n \* msg(|S|/n) (Giai đoạn phân mảnh lại)
  + CPRO = m\*CLOC\_build(|R|/m) + n\*CLOC\_probe(|S|/n)
  + Thường là hiệu quả nhất cho các phép nối bằng (equijoin) nếu không có vấn đề về dữ liệu lệch.

## Bài tập 8.6

Bốn cây toán tử khả dĩ cho truy vấn:

1. Right-deep: (((σ(PROJ) ⋈ σ(ASG)) ⋈ EMP)) - Cho phép pipeline tối đa.
2. Left-deep: ((EMP ⋈ σ(ASG)) ⋈ σ(PROJ)) - Yêu cầu lưu trữ các kết quả trung gian.
3. Zigzag: ((EMP ⋈ σ(PROJ)) ⋈ σ(ASG)) - Một dạng lai giữa left-deep và right-deep.
4. Bushy: (EMP ⋈ σ(ASG)) ⋈ σ(PROJ) - Cho phép thực thi song song độc lập các phép nối EMP ⋈ ASG và (kết quả) ⋈ PROJ. Đây là loại cây linh hoạt nhất cho tính toán song song.

## Bài tập 8.7

Với 10 quan hệ (N=10):

* Số cây right-deep: N! = 10! = 3,628,800 (có N! hoán vị của các quan hệ)
* Số cây left-deep: N! = 10! = 3,628,800
* Số cây bushy: (2(N-1))! / (N-1)! \* C(N) với C(N) là số Catalan thứ N. Con số này cực kỳ lớn, vào khoảng 1.76 \* 10^11.

Kết luận: Không gian tìm kiếm cho các cây bushy lớn hơn rất nhiều so với các cây tuyến tính, làm cho việc tối ưu hóa song song trở nên rất phức tạp và tốn kém.

## Bài tập 8.8

Chiến lược đặt dữ liệu cho một cluster NUMA:

Dữ liệu nên được phân mảnh (ví dụ: hash partitioning) và mỗi phân mảnh nên được đặt trong bộ nhớ cục bộ (local memory) của node có khả năng cao nhất sẽ xử lý nó. Mục tiêu là tối đa hóa truy cập bộ nhớ cục bộ và giảm thiểu truy cập bộ nhớ từ xa (remote memory access) qua interconnect, vì truy cập từ xa tốn kém hơn nhiều. Cần có sự phối hợp giữa chiến lược đặt dữ liệu và bộ lập lịch truy vấn để đảm bảo các tác vụ được thực thi trên node chứa dữ liệu của chúng.

## Bài tập 8.9

Để mô hình thực thi DP (Dynamic Processing) hỗ trợ song song liên truy vấn (inter-query parallelism), cần thay đổi sau:

Thay vì mỗi truy vấn có một tập hàng đợi kích hoạt (activation queues) riêng, cần có một tập hàng đợi kích hoạt toàn cục hoặc dùng chung cho tất cả các truy vấn đang chạy trên một node SM. Các thread trên node đó có thể lấy các activation từ bất kỳ truy vấn nào đang có trong hàng đợi. Điều này cho phép các tài nguyên (CPU) được chia sẻ linh hoạt giữa các truy vấn khác nhau, giúp cân bằng tải trên toàn hệ thống thay vì chỉ trong một truy vấn.

## Bài tập 8.10

Để cho phép song song liên truy vấn trong một hệ CSDL tập trung:

* Từ góc độ nhà phát triển CSDL: Cần thiết kế lại bộ máy thực thi (execution engine) để trở thành đa luồng (multi-threaded), cho phép nhiều truy vấn chạy đồng thời trên các lõi CPU khác nhau. Cần có các cơ chế điều khiển tương tranh và quản lý tài nguyên hiệu quả.
* Từ góc độ người quản trị (DBA): Cần cấu hình hệ thống để cho phép một mức độ song song nhất định (ví dụ: max\_parallel\_workers), và theo dõi hiệu suất hệ thống để điều chỉnh tham số này.
* Từ góc độ người dùng cuối: Thông lượng (throughput) của hệ thống tăng lên, nghĩa là nhiều người dùng có thể thực hiện truy vấn cùng lúc. Tuy nhiên, thời gian phản hồi cho một truy vấn cá nhân có thể tăng lên một chút khi hệ thống chịu tải nặng do tranh chấp tài nguyên.

## Bài tập 8.11

* Các lớp phần mềm trong middleware của database cluster:
  1. Transaction Load Balancer: Tiếp nhận các giao dịch từ client, quyết định node nào sẽ thực thi dựa trên tải hiện tại và chuyển giao dịch đến node đó.
  2. Replication Manager: Quản lý việc truy cập dữ liệu được sao chép, đảm bảo tính nhất quán (ví dụ: 1SR) bằng cách điều phối thứ tự thực thi các giao dịch cập nhật trên các bản sao.
  3. Query Processor: Phân tích các truy vấn phức tạp, tạo kế hoạch thực thi song song (intra-query parallelism), gửi các truy vấn con đến các node và tổng hợp kết quả.
  4. Fault-Tolerance Manager: Phát hiện lỗi node, quản lý quá trình chuyển đổi dự phòng (failover) và phục hồi (recovery).
* Thông tin cần chia sẻ giữa các node: Trạng thái tải của mỗi node, thông tin danh mục (directory) về vị trí dữ liệu/bản sao, và các log sao chép để duy trì tính nhất quán. Việc chia sẻ có thể được thực hiện qua một dịch vụ danh mục được sao chép hoặc qua các thông điệp quảng bá (broadcast messages).

## Bài tập 8.12

Các vấn đề về khả năng chịu lỗi đối với giao thức sao chép phòng ngừa (preventive replication):

* Lỗi node: Nếu một node bị lỗi, các node khác sẽ timeout khi chờ thông điệp từ nó. Do giao thức yêu cầu đợi một khoảng delay để đảm bảo nhận được tất cả các giao dịch có timestamp nhỏ hơn, hệ thống phải có khả năng xử lý việc không nhận được thông điệp từ node lỗi trong khoảng thời gian này và vẫn duy trì được thứ tự tổng (total order) giữa các giao dịch từ các node còn lại.
* Phân hoạch mạng: Nếu mạng bị phân hoạch, các phân vùng khác nhau có thể không thấy các giao dịch của nhau. Giao thức này sẽ thất bại trong việc đảm bảo 1SR vì nó dựa trên giả định một hệ thống đồng bộ với độ trễ bị chặn, điều không còn đúng khi có phân hoạch mạng.

## Bài tập 8.13

So sánh Preventive Replication và Eager Replication trong Database Cluster:

| Tiêu chí | Preventive Replication | Eager Replication (dùng Group Comm.) |
| --- | --- | --- |
| Cấu hình sao chép | Hỗ trợ sao chép toàn phần. | Hỗ trợ sao chép toàn phần và cục bộ. |
| Yêu cầu mạng | Multicast FIFO tin cậy, hệ thống đồng bộ. | Multicast có thứ tự tổng (total order). |
| Tính nhất quán | 1SR, nhất quán mạnh (strong consistency). | 1SR, nhất quán mạnh (strong consistency). |
| Hiệu năng | Thông lượng cao do là lazy, nhưng có độ trễ do phải đợi. | Độ trễ thấp hơn lazy, nhưng thông lượng có thể thấp hơn do đồng bộ hóa. |
| Khả năng chịu lỗi | Nhạy cảm với lỗi node và phân hoạch mạng. | Chịu lỗi tốt hơn miễn là đa số node còn hoạt động. |

## Bài tập 8.14

1. Có thể sử dụng Virtual Partitioning (VP). \*Subquery: SELECT B, COUNT(C) AS CntC FROM R WHERE <partition\_predicate> GROUP BY B
   * Composition Query: SELECT B, SUM(CntC) FROM <subquery\_results> GROUP BY B
2. Có thể sử dụng VP. \*Subquery: SELECT C, SUM(D) AS SumD, COUNT(D) AS CntD, SUM(E) AS SumE, COUNT(E) AS CntE FROM R WHERE B=:v1 AND <partition\_predicate> GROUP BY C
   * Composition Query: SELECT C, SUM(SumD), SUM(SumE)/SUM(CntE) FROM <subquery\_results> GROUP BY C
3. Có thể sử dụng VP. Phân mảnh R. \*Subquery: SELECT B, SUM(E) AS SumE, COUNT(\*) AS Cnt FROM R, S WHERE R.A=S.A AND <partition\_predicate\_on\_R> GROUP BY B
   * Composition Query: SELECT B, SUM(SumE) FROM <subquery\_results> GROUP BY B HAVING SUM(Cnt) > 50
4. Không thể sử dụng VP một cách hiệu quả. Truy vấn con (SELECT SUM(G) FROM S WHERE S.A=R.A) là một truy vấn con tương quan (correlated subquery), việc song song hóa nó rất phức tạp.
5. Không thể sử dụng VP. Truy vấn con (SELECT MAX(H) FROM S WHERE G >= :v1) không tương quan và có thể được tính một lần, nhưng vị từ chính trên R (D > kết quả) không dễ để phân mảnh hiệu quả.