# Bài tập Chương 10

## Bài tập 10.1

Bảng so sánh các phương pháp thiết kế hệ thống lưu trữ:

| Tiêu chí | Block Storage (DBMS) | Object Storage | Distributed File System (HDFS) |
| --- | --- | --- | --- |
| **Khả năng mở rộng** | Hạn chế (scale-up) | Rất cao (scale-out) | Rất cao (scale-out) |
| **Dễ sử dụng** | Khó (yêu cầu schema) | Dễ (API đơn giản put/get) | Trung bình (giao diện file) |
| **Kiến trúc** | Tích hợp chặt chẽ | Flat namespace | Phân cấp (hierarchical) |
| **Tính nhất quán** | Mạnh (ACID) | Thường là nhất quán sau cùng (eventual) | Nhất quán cho file đã đóng |
| **Khả năng chịu lỗi** | Phụ thuộc vào DBMS | Rất cao (sao chép nội tại) | Rất cao (sao chép chunk) |
| **Quản lý Metadata** | Tích hợp trong DBMS (catalog) | Metadata linh hoạt, đi kèm object | Tập trung tại NameNode |

## Bài tập 10.2

* Ngăn xếp DBMS quan hệ truyền thống (Hình 1.9): Là một hệ thống tích hợp chặt chẽ. Bộ máy DBMS có toàn quyền kiểm soát việc lưu trữ ở cấp độ block vật lý. Việc xử lý và lưu trữ được tích hợp làm một.
* Ngăn xếp Big Data (Hình 10.1): Là một hệ thống tích hợp lỏng lẻo. Lớp xử lý (ví dụ: Spark) nằm trên một lớp lưu trữ phân tán riêng biệt (ví dụ: HDFS). Việc tách biệt này cho phép lưu trữ và tính toán có thể mở rộng một cách độc lập. Lớp lưu trữ dựa trên file hoặc object, không phải block.

## Bài tập 10.3

* Sử dụng Object Storage khi:
  + Cần lưu trữ một số lượng cực lớn các đối tượng tương đối nhỏ, không có cấu trúc (ví dụ: ảnh, video, file đính kèm email).
  + Ứng dụng yêu cầu API truy cập đơn giản dựa trên REST (put/get).
  + Cần siêu dữ liệu (metadata) linh hoạt cho từng đối tượng.
  + Môi trường là đám mây (cloud) và cần khả năng mở rộng không gian tên phẳng (flat namespace) lớn.
* Sử dụng File Storage (Phân tán) khi:
  + Cần lưu trữ các file rất lớn và thường được ghi nối tiếp (append), ví dụ như file log.
  + Ứng dụng được hưởng lợi từ cấu trúc thư mục phân cấp.
  + Đây là lớp lưu trữ cho các framework xử lý dữ liệu lớn như MapReduce, vốn hoạt động hiệu quả trên các chunk file lớn.

## Bài tập 10.4

* Phân mảnh ngang (Horizontal Partition - Chương 2): Là một khái niệm logic. Nó là một tập hợp con các tuple của một quan hệ thỏa mãn một vị từ ngữ nghĩa (ví dụ: CITY = 'Paris'). Việc phân mảnh dựa trên nội dung của dữ liệu.
* Chunk (trong GFS/HDFS): Là một khái niệm vật lý. Nó là một khối có kích thước cố định của một file (ví dụ: 128MB). Việc phân mảnh dựa trên kích thước của dữ liệu, không phải nội dung.

## Bài tập 10.5

* Tính tổng quát: Repartition join tổng quát hơn, hoạt động với các bảng có kích thước bất kỳ. Broadcast join chỉ hiệu quả khi một trong hai bảng (quan hệ bên trong) đủ nhỏ để vừa với bộ nhớ của mỗi mapper.
* Chi phí Shuffling: Repartition join có chi phí shuffling cao vì cả hai bảng đều phải được phân vùng lại và gửi qua mạng đến các reducer. Broadcast join không có chi phí shuffling; chi phí của nó nằm ở việc quảng bá (broadcast) bảng nhỏ đến tất cả các mapper, nhưng nó tránh được giai đoạn sắp xếp-trộn tốn kém ở phía reduce.

## Bài tập 10.6

1. Mã giả cho hàm Combiner:

// Input: (CITY, list\_of\_ones) ví dụ: ('New York', [1, 1, 1, ...])  
// Output: (CITY, partial\_sum) ví dụ: ('New York', 1500)  
function Combine(city, list\_of\_ones):  
 sum = 0  
 for each one in list\_of\_ones:  
 sum = sum + 1  
 emit(city, sum)

1. Cách giảm chi phí shuffling: Thay vì gửi hàng nghìn cặp ('New York', 1) qua mạng từ một mapper đến reducer, combiner sẽ tổng hợp chúng cục bộ ngay tại node của mapper thành một cặp duy nhất, ví dụ ('New York', 1500). Chỉ có cặp đã được tổng hợp này được gửi qua mạng, do đó làm giảm đáng kể lưu lượng dữ liệu cần shuffle.

## Bài tập 10.7

Việc sử dụng luồng dữ liệu của theta-join tổng quát (Hình 10.10) cho một phép nối bằng (equijoin) là rất không hiệu quả. Luồng dữ liệu này sao chép và gửi các tuple đến nhiều reducer không cần thiết. Ngược lại, thuật toán repartition join chuyên dụng (Hình 10.11) hiệu quả hơn nhiều vì nó sử dụng hàm băm trên khóa nối để đảm bảo tất cả các tuple có cùng khóa chỉ được gửi đến duy nhất một reducer, loại bỏ việc sao chép dữ liệu thừa.

## Bài tập 10.8

Trong thuật toán k-means trên Spark, bước tính toán lại tâm của các cụm (centroids) yêu cầu shuffling. Cụ thể, bước Tạo một RDD M\_new chứa giá trị trung bình của các xi được gán cho mỗi μj tương đương với một thao tác groupByKey (nhóm theo ID của centroid) theo sau là một phép tổng hợp (tính trung bình). Tất cả các điểm dữ liệu xi được gán cho cùng một centroid μj phải được chuyển đến cùng một worker để có thể tính toán tâm mới.

## Bài tập 10.9

Giả sử trang nguồn là P1.

* Khởi tạo: PR(P1) = 1, PR(P\_i) = 0 cho i ≠ 1.
* Vòng lặp 1:
  + PR(P2) = (1-0.85) + 0.85 \* (PR(P1)/2) = 0.15 + 0.85 \* (1/2) = 0.575
  + PR(P3) = (1-0.85) + 0.85 \* (PR(P1)/2) = 0.575
  + Các trang khác vẫn có PR = 0.15 (vì chúng nhận được (1-d) và đóng góp từ các trang khác là 0).
* Vòng lặp 2:
  + PR(P4) = (1-0.85) + 0.85 \* (PR(P3)/3) = 0.15 + 0.85 \* (0.575/3) ≈ 0.312
  + … (Quá trình tiếp tục cho đến khi các giá trị hội tụ).

## Bài tập 10.10

Mã giả cho một vòng lặp của Personalized PageRank trên MapReduce:

* Map Function:
  + Input: (PageID, (CurrentRank, AdjacencyList))
  + contribution = CurrentRank / len(AdjacencyList)
  + For each neighbor in AdjacencyList:
    - emit(neighbor, contribution)
  + emit(PageID, AdjacencyList) // Để giữ lại cấu trúc đồ thị
* Reduce Function:
  + Input: (PageID, list\_of\_values)
  + sum\_contributions = 0, adj\_list = []
  + For each value in list\_of\_values:
    - If value is a contribution (a number): sum\_contributions += value
    - If value is an adjacency list: adj\_list = value
  + new\_rank = (1 - d)
  + if PageID == source\_page: new\_rank += d
  + new\_rank += d \* sum\_contributions
  + emit(PageID, (new\_rank, adj\_list))

Một chương trình điều khiển (driver) sẽ lặp lại job MapReduce này cho đến khi các giá trị rank hội tụ.

## Bài tập 10.11

Logic cho một vòng lặp của Personalized PageRank trên Spark:

1. Tải đồ thị vào một RDD graph(PageID, AdjacencyList).
2. Khởi tạo RDD ranks(PageID, RankValue).
3. Trong một vòng lặp:
   * contributions = graph.join(ranks).flatMap { (id, (adj, rank)) => adj.map(dest => (dest, rank / adj.size)) }
   * new\_ranks = contributions.reduceByKey(\_ + \_).mapValues { sum => (1 - d) + d \* sum }
   * (Thêm phần +d cho trang nguồn)
   * Cập nhật ranks với new\_ranks.
   * Kiểm tra sự hội tụ.

## Bài tập 10.12

Thuật toán để thực hiện ngữ nghĩa at-least-once: Sử dụng cơ chế upstream backup.

1. Toán tử upstream (nguồn) gửi một tuple t đến toán tử downstream (đích) và lưu t vào một bộ đệm (buffer) cục bộ.
2. Toán tử downstream sau khi xử lý xong t sẽ gửi lại một thông điệp xác nhận (acknowledgement - ACK) cho toán tử upstream.
3. Khi toán tử upstream nhận được ACK cho t, nó sẽ xóa t khỏi bộ đệm của mình.
4. Nếu toán tử downstream bị lỗi và khởi động lại, nó sẽ yêu cầu toán tử upstream gửi lại tất cả các tuple chưa được ACK từ bộ đệm. Điều này đảm bảo mỗi tuple được xử lý ít nhất một lần.

## Bài tập 10.13

1. Filter Operator (stateless): Dùng shuffle partitioning (hoặc round-robin). Dòng dữ liệu đến được phân phối đều cho các worker. Mỗi worker áp dụng điều kiện lọc một cách độc lập. Không cần bước tổng hợp.
2. Aggregate Operator (stateful): Phải dùng hash partitioning dựa trên thuộc tính GROUP BY. Điều này đảm bảo tất cả các tuple có cùng khóa sẽ được gửi đến cùng một worker. Mỗi worker duy trì trạng thái tổng hợp (ví dụ: tổng và số đếm hiện tại) cho các khóa mà nó chịu trách nhiệm. Không cần bước tổng hợp kết quả giữa các worker.

## Bài tập 10.14

* Tính xác định (deterministic): Phép nối cửa sổ trượt không xác định. Kết quả của nó phụ thuộc vào thứ tự và thời gian đến của các tuple từ các dòng dữ liệu khác nhau.
* Thiết kế xác định: Sử dụng punctuations hoặc heartbeats. Toán tử nối chỉ xử lý cửa sổ của nó đến thời điểm t khi và chỉ khi nó nhận được một punctuation từ cả hai dòng dữ liệu đầu vào, đảm bảo rằng sẽ không có tuple nào có timestamp ≤ t đến nữa. Điều này buộc thứ tự xử lý phải dựa trên thời gian logic, nhưng làm tăng độ trễ.

## Bài tập 10.15

| Tiêu chí | BSP (Bulk Synchronous Parallel) | GAS (Gather-Apply-Scatter) |
| --- | --- | --- |
| **Tính tổng quát và biểu cảm** | Cả hai đều có thể biểu diễn hầu hết các thuật toán đồ thị lặp. | Cả hai đều có thể biểu diễn hầu hết các thuật toán đồ thị lặp. |
| **Tối ưu hóa hiệu năng** | Mô hình đơn giản, dễ lý giải nhưng bị ảnh hưởng bởi các worker chậm (stragglers) do rào cản đồng bộ toàn cục. | Linh hoạt hơn. Bằng cách tách biệt việc cập nhật trạng thái và kích hoạt tính toán, nó cho phép lập lịch thông minh hơn, có thể hội tụ nhanh hơn và ít bị ảnh hưởng bởi stragglers. |

## Bài tập 10.16

Mã giả cho hàm Compute(v) trong mô hình vertex-centric BSP cho Personalized PageRank:

function Compute(vertex v, messages):  
 if superstep == 0:  
 if v.ID == source\_page:  
 v.value = 1.0  
 else:  
 v.value = 0.0  
 else:  
 sum\_contributions = 0  
 for each msg in messages:  
 sum\_contributions += msg.value  
 v.value = (1 - d) + d \* sum\_contributions  
 if v.ID == source\_page:  
 v.value += d  
  
 if v.out\_degree > 0:  
 contribution = v.value / v.out\_degree  
 for each neighbor u in v.neighbors:  
 sendMessage(to: u, value: contribution)  
  
 // Kiểm tra hội tụ và voteToHalt()

## Bài tập 10.17

Sử dụng cấu trúc dữ liệu Union-Find (Disjoint Set Union).

1. Khởi tạo cấu trúc Union-Find, trong đó mỗi đỉnh (vertex) của đồ thị là một tập hợp riêng biệt.
2. Khi một cạnh mới (u, v) đến từ dòng dữ liệu:
   * Thực hiện thao tác union(u, v). Thao tác này sẽ hợp nhất hai tập hợp chứa u và v nếu chúng chưa cùng một tập hợp.
3. Sau khi xử lý tất cả các cạnh, các thành phần liên thông của đồ thị chính là các tập hợp riêng biệt còn lại trong cấu trúc Union-Find.

Thuật toán này có tính tăng dần (incremental) và hiệu quả hơn nhiều so với việc chạy lại thuật toán tìm thành phần liên thông từ đầu.

## Bài tập 10.18

1. Thứ tự bất lợi (Adversarial Order): Trình bày tất cả các cạnh nối với đỉnh có bậc cao nhất trước tiên. Ví dụ, trong Hình 10.23, nếu ta xử lý tất cả các cạnh của đỉnh v4 trước, sau đó là các cạnh của v3, thuật toán greedy có thể sẽ gán tất cả các cạnh này vào cùng một phân vùng để giảm thiểu số lần sao chép đỉnh, dẫn đến mất cân bằng tải nghiêm trọng.
2. Chiến lược giảm thiểu:

* Ngẫu nhiên hóa (Randomize): Xáo trộn ngẫu nhiên thứ tự của dòng cạnh đầu vào trước khi áp dụng thuật toán phân mảnh.
* Xử lý hai lượt (Two-pass): Lượt đầu tiên chỉ để thu thập thống kê (ví dụ: bậc của các đỉnh). Lượt thứ hai sử dụng thông tin thống kê này để đưa ra quyết định phân mảnh tốt hơn, ví dụ như ưu tiên xử lý các cạnh nối các đỉnh bậc thấp trước.

## Bài tập 10.19

Spark không đủ để xây dựng một data lake hoàn chỉnh, nhưng nó là một bộ máy xử lý chính trong đó. Dựa trên kiến trúc data lake (Hình 10.30), Spark cung cấp các lớp Data Access (Spark SQL) và Data Analysis. Tuy nhiên, một data lake hoàn chỉnh còn thiếu các thành phần quan trọng trong lớp Platform Management:

* Data Governance (Quản trị dữ liệu): Các công cụ chuyên dụng để quản lý dòng dữ liệu (data lineage), siêu dữ liệu, chất lượng dữ liệu và các chính sách của doanh nghiệp.
* Data Security (Bảo mật dữ liệu): Một hệ thống tập trung để xác thực, phân quyền và bảo vệ dữ liệu.
* Operations (Vận hành): Các công cụ để giám sát, lập lịch và cấp phát tài nguyên trên toàn bộ hệ sinh thái (ví dụ: Ambari, Zookeeper).

## Bài tập 10.20

So sánh tích hợp dữ liệu trong Data Lake và Data Warehouse hiện đại:

* Điểm chung:
  + Cả hai đều nhằm mục đích cung cấp một giao diện truy vấn thống nhất (thường là SQL) trên các nguồn dữ liệu không đồng nhất (quan hệ và HDFS/không cấu trúc).
  + Cả hai đều sử dụng khái niệm “connector” hoặc “wrapper” (ví dụ: Polybase HDFS Bridge, Spark connectors) để truy cập dữ liệu bên ngoài.
* Điểm khác biệt:

| * Tiêu chí | * Data Warehouse hiện đại (với External Table) | * Data Lake (với Spark) |
| --- | --- | --- |
| * **Trọng tâm dữ liệu** | * Dữ liệu chính là dữ liệu có cấu trúc, đã được quản lý trong warehouse. Dữ liệu bên ngoài là phụ. | * Dữ liệu chính là dữ liệu thô, đa định dạng trong lake. Dữ liệu quan hệ chỉ là một trong nhiều nguồn. |
| * **Mô hình dữ liệu** | * “Tâm điểm” là mô hình quan hệ. | * Không có mô hình dữ liệu trung tâm. Lược đồ được áp dụng khi đọc (schema-on-read). |
| * **Quy trình** | * Vẫn có xu hướng ETL (Extract-Transform-Load) cho dữ liệu chính. | * Hoàn toàn theo mô hình ELT (Extract-Load-Transform). |
| * **Độ khớp nối** | * Thường là tích hợp chặt chẽ hơn (ví dụ: Polybase tích hợp sâu vào bộ máy truy vấn của PDW). | * Thường là tích hợp lỏng lẻo hơn thông qua các connector. |