Chương 8:
Câu 1:

Nhân bản dữ liệu trong hệ thống phân tán là quá trình tạo và duy trì nhiều bản sao của cùng một dữ liệu trên các nút khác nhau trong hệ thống. Mục đích là để tăng độ tin cậy, khả năng chịu lỗi và cải thiện hiệu năng truy cập.

Có hai lý do chính để triển khai nhân bản dữ liệu:

- Tăng độ sẵn sàng: Khi một nút bị lỗi hoặc tạm thời không hoạt động, hệ thống vẫn có thể phục vụ yêu cầu người dùng thông qua các bản sao khác.
- Cải thiện hiệu năng truy cập: Người dùng được phục vụ từ bản sao gần nhất về mặt địa lý hoặc mạng, giúp giảm độ trễ và giảm tải cho máy chủ gốc.

Việc nhân bản dữ liệu trong hệ thống phân tán có thể dẫn đến vấn đề nhất quán (consistency) vì các bản sao dữ liệu không phải lúc nào cũng được cập nhật đồng thời. Khi dữ liệu thay đổi tại một nút (ví dụ như cơ sở dữ liệu gốc), việc truyền cập nhật đến tất cả các bản sao (cache, node dự phòng, v.v.) có thể bị chậm trễ, mất đồng bộ, hoặc gặp lỗi trong quá trình đồng bộ.

Trong khoảng thời gian giữa lúc dữ liệu được cập nhật tại nguồn và lúc các bản sao được đồng bộ hóa, một số người dùng có thể truy cập vào các bản sao chưa cập nhật và nhìn thấy dữ liệu cũ, trong khi người khác lại thấy dữ liệu mới. Đây là tình huống không nhất quán tạm thời – cũng gọi là eventual inconsistency.

Ví dụ: Kịch bản cache web

Giả sử một website tin tức sử dụng hệ thống cache phân tán (ví dụ như Redis hoặc CDN) để tăng tốc độ truy cập.

1. Ban biên tập cập nhật một bài viết trên cơ sở dữ liệu gốc (ví dụ sửa lỗi nội dung hoặc cập nhật thông tin mới).
2. Tuy nhiên, các node cache vẫn đang lưu phiên bản cũ của bài viết, vì cache chưa hết hạn hoặc chưa được làm mới.
3. Người dùng A truy cập bài viết và được phục vụ nội dung từ cache cũ → thấy nội dung chưa cập nhật.
4. Cùng thời điểm đó, người dùng B truy cập và được điều hướng đến máy chủ gốc hoặc cache đã làm mới → thấy nội dung mới.

Kết quả là cùng một URL, nhưng người dùng nhận được các nội dung khác nhau – đây là hiện tượng không nhất quán do nhân bản.

Để giảm thiểu vấn đề này, các hệ thống thường kết hợp các kỹ thuật như:

- Thiết lập thời gian sống (TTL) ngắn cho cache
- Chủ động xóa cache (cache invalidation) khi dữ liệu thay đổi
- Sử dụng các mô hình nhất quán như strong consistency hoặc quorum-based updates nếu dữ liệu rất quan trọng

. Phân bố bản sao dữ liệu (Geo-replication) để tối ưu độ trễ
Mục tiêu: Giảm độ trễ truy cập, tăng tính sẵn sàng, và chia tải cho các trung tâm dữ liệu.
Chiến lược đề xuất:

- Triển khai nhiều cụm dữ liệu (data clusters) tại các khu vực địa lý chiến lược: Bắc Mỹ, Châu Âu, Châu Á – TBD, Nam Mỹ...
- Mỗi cụm xử lý truy cập từ người dùng trong khu vực gần nhất (theo định tuyến DNS/Anycast hoặc Load balancer).
- Sử dụng mô hình đa lãnh thổ (multi-region), trong đó:
    - Dữ liệu không thay đổi thường xuyên (ví dụ: danh sách sản phẩm, thông tin mô tả) → replicate đầy đủ sang tất cả các khu vực.
    - Dữ liệu thay đổi thường xuyên (ví dụ: tồn kho, giá động, giỏ hàng, trạng thái đơn hàng) → replicate theo mô hình chủ–phụ (leader-follower) hoặc quorum-based để đảm bảo nhất quán hợp lý (eventual hoặc causal consistency tùy loại dữ liệu).
- Đồng bộ dữ liệu theo nhóm chức năng:
    - Giỏ hàng và đơn hàng: lưu vùng gần người dùng nhưng commit về vùng chủ để bảo toàn nhất quán.
    - Dữ liệu sản phẩm: cache toàn cầu, đồng bộ theo lô.

Kết quả: Người dùng được phục vụ từ bản sao gần nhất, giảm độ trễ mà vẫn đảm bảo tính đúng đắn của dữ liệu phù hợp với loại truy cập.

2. **Cơ chế làm tươi cache (Cache Invalidation/TTL)**
Mục tiêu: Cân bằng giữa độ "mới" của dữ liệu và chi phí băng thông/cache refresh.
Chiến lược đề xuất:

- Phân loại dữ liệu để kiểm soát độ tươi khác nhau:

Loại dữ liệu	TTL đề xuất	Làm mới
Sản phẩm mới, mô tả	10–30 phút	TTL đơn giản
Giá, tồn kho	10–60 giây	Cache + thông báo chủ động (invalidation)
Trạng thái đơn hàng	không cache hoặc cache vài giây	Cache cá nhân (per user/session)

Danh mục sản phẩm	5–10 phút	TTL đơn giản hoặc cập nhật theo sự kiện

Kỹ thuật cache cụ thể:

- Cache phân tầng:
    - CDN (biên) → Cache trong cluster → Cơ sở dữ liệu
- TTL có kiểm soát (adaptive TTL):
    - Tự động rút ngắn TTL nếu dữ liệu hay thay đổi (dựa trên thống kê truy cập/cập nhật).
- Cache invalidation theo sự kiện:
    - Khi sản phẩm được cập nhật (giá, tồn kho), push thông báo xóa cache tại các vùng liên quan (publish–subscribe, message queue).
    - Ví dụ: Kafka + Redis + cache key tagging để xóa cache đồng bộ.

. So sánh chi phí mạng và lợi ích: cập nhật đồng bộ vs không đồng bộ
Cập nhật đồng bộ (Synchronous replication)

- Cách hoạt động: Dữ liệu chỉ được xác nhận là "ghi thành công" sau khi tất cả các bản sao được cập nhật.
- Chi phí mạng: Rất cao – vì mỗi ghi yêu cầu gửi dữ liệu đến tất cả bản sao trước khi trả kết quả cho client.
- Lợi ích:
    - Dữ liệu giữa các bản sao luôn nhất quán.
    - Đọc từ bất kỳ bản sao nào đều chính xác.
- Không phù hợp với đặc tính $N \ll M$:
    - Vì ta đang ghi rất nhiều, đồng bộ sẽ gây tắc nghẽn mạng và tăng độ trễ ghi, trong khi đọc thì ít, không tận dụng lợi ích của sự nhất quán cao.

Cập nhật không đồng bộ (Asynchronous replication)

- Cách hoạt động: Ghi hoàn tất ngay tại nút chính, rồi cập nhật bản sao sau (qua background job hoặc theo lịch).
- Chi phí mạng: Thấp hơn nhiều – chỉ gửi bản sao theo batch hoặc định kỳ.
- Lợi ích:
    - Ghi rất nhanh, ít tiêu tốn băng thông.
    - Chấp nhận một độ trễ nhỏ trong việc đồng bộ hóa dữ liệu.
- Phù hợp với hệ thống $N \ll M$:
    - Vì dữ liệu ghi thường xuyên nhưng ít khi được đọc lại, nên không cần thiết phải duy trì bản sao cập nhật theo thời gian thực.

2**. Chiến lược đặt vị trí bản sao và tần suất làm tươi**
Chiến lược đặt bản sao:

- Tối thiểu hóa số lượng bản sao gần người dùng:
    - Vì số lần đọc thấp, không cần phân phối nhiều điểm đọc toàn cầu.

    - Có thể giữ bản sao ở một cụm trung tâm gần nơi xử lý ghi để giảm băng thông truyền bản cập nhật.
- Ưu tiên bản sao ghi nhanh, đọc gần trung tâm (read from write region):
    - Nếu có yêu cầu đọc → chuyển đến bản gốc (write master) hoặc bản sao đã được cập nhật gần đây.

Tần suất làm tươi bản sao (replica refresh):

- Batch-based hoặc interval-based update:
    - Làm tươi mỗi X phút hoặc sau Y bản ghi → giúp gom cập nhật lại, giảm tần suất truyền dữ liệu.
    - Có thể dùng đồng bộ nền (background replication).
- Làm tươi theo sự kiện nếu có đọc phát sinh:
    - Khi có yêu cầu đọc đến bản sao chưa cập nhật → trigger làm mới bản sao ngay trước khi phục vụ.
- TTL động hoặc log-based sync:
    - Ghi lại log cập nhật, gửi định kỳ cho bản sao để đồng bộ lại trạng thái.

Đánh giá mô hình nhân bản đồng bộ (strong consistency) trong hệ thống phân tán toàn cầu
Ưu điểm:

- Dữ liệu nhất quán tuyệt đối: người dùng ở bất kỳ đâu luôn thấy cùng một kết quả tại cùng một thời điểm.
- Dễ suy luận về hành vi hệ thống: đặc biệt quan trọng với các ứng dụng như ngân hàng, thanh toán, chứng khoán.
- Tránh được lỗi logic do đọc dữ liệu lỗi thời: ví dụ không bị "mua cùng 1 sản phẩm hết hàng" từ hai nơi khác nhau.

Nhược điểm:

- Độ trễ cao: mọi cập nhật phải đồng bộ giữa các vùng địa lý trước khi xác nhận → mất hàng trăm mili giây đến vài giây, nhất là giữa các châu lục.
- Giảm độ sẵn sàng: nếu một bản sao (node) không phản hồi, toàn bộ hệ thống có thể bị treo (theo CAP theorem).
- Chi phí cao: về băng thông, đồng bộ mạng, và xử lý lỗi tạm thời.

Trong hệ thống thanh toán toàn cầu, strong consistency là cần thiết cho các giao dịch cốt lõi như trừ tiền, chuyển khoản. Tuy nhiên, áp dụng toàn cục cho toàn bộ hệ thống sẽ gây giảm hiệu năng nghiêm trọng.

Thiết kế giải pháp nhân bản nhất quán lỏng (loose consistency)
1. Lựa chọn mức độ "lỏng": causal hoặc eventual consistency

- Causal consistency:
  Cập nhật được đảm bảo theo thứ tự nhân quả. Nếu hành động A xảy ra trước B, mọi node phải thấy A trước B.
    - Phù hợp với: hệ thống giao dịch có liên quan giữa các bước (ví dụ: cập nhật số dư rồi gửi thông báo).

    - Yêu cầu tracking dependency (vector clock, Lamport clock).
- Eventual consistency:
  Miễn là không có cập nhật mới, các bản sao sẽ dần hội tụ về cùng trạng thái theo thời gian.
    - Phù hợp với: các phần không quan trọng như lịch sử truy cập, trạng thái đơn hàng, lượt thích…
    - Không đảm bảo thứ tự giữa các cập nhật, nhưng đơn giản, hiệu quả và nhanh.

Đề xuất phối hợp:

- Dùng causal consistency cho các dữ liệu liên quan đến logic giao dịch (thứ tự thực hiện).
- Dùng eventual consistency cho phần hiển thị, log không quan trọng.

2. Cơ chế đảm bảo cập nhật quan trọng không bị mất
Đối với dữ liệu nhạy cảm như giao dịch tài chính, áp dụng:

- Ghi log định danh giao dịch toàn cục (Global Transaction ID hoặc UUID), lưu tại ít nhất quorum bản sao (ví dụ: 3/5 node).
- Áp dụng Write-ahead logging (WAL) hoặc commit log bất biến (append-only), ghi lại mọi thay đổi trước khi gửi về client.
- Dùng cơ chế write quorum / read quorum:
    - W + R > N đảm bảo không mất dữ liệu và luôn có ít nhất một bản sao giao nhau giữa ghi và đọc.
- Nếu cần đảm bảo toàn vẹn, có thể sử dụng 2PC (two-phase commit) hoặc atomic broadcast ở mức dịch vụ tài chính.

3. Phục hồi khi một số bản sao quá cũ
Khi một bản sao quá “lạc hậu” (ví dụ do bị ngắt mạng):

- Phát hiện bằng vector clock hoặc lamport timestamp:
    - Khi một bản sao quay lại, so sánh thời gian logic với bản sao chủ hoặc cluster.
- Cơ chế phục hồi (resync):
    - Gửi lại tất cả các delta updates hoặc thực hiện state snapshot từ bản sao mới nhất.
    - Nếu cập nhật conflict (ví dụ hai bản khác nhau ghi cùng khóa), dùng:
        - Last-write-wins (LWW) nếu cho phép ghi đè.
        - Giao thức hợp nhất (CRDTs, OT) nếu cho phép chỉnh hợp.
- Cơ chế làm mới có kiểm soát:
    - Định kỳ xác minh độ lệch giữa bản sao và dữ liệu chủ.
    - Nếu chênh quá giới hạn → đánh dấu bản sao là "stale", loại khỏi hệ thống cho đến khi cập nhật xong.

Câu 2:

Bốn mô hình nhất quán theo thứ tự thao tác (8.2.2)

a. Nhất quán nghiêm ngặt (Strict Consistency)

Mọi thao tác đọc luôn trả về giá trị mới nhất đã được ghi, bất kể vị trí hay thời điểm thực hiện. Đây là mô hình mạnh nhất.

b. Nhất quán tuần tự (Sequential Consistency)

Kết quả thực thi các thao tác giống như thể các thao tác được sắp xếp theo một thứ tự tuần tự toàn cục, và mỗi tiến trình tuân theo thứ tự thao tác của riêng nó.

c. Nhất quán nhân quả (Causal Consistency)

Nếu một thao tác ghi có quan hệ nhân quả với một thao tác ghi khác, thì tất cả các tiến trình đều phải thấy chúng theo cùng thứ tự. Các thao tác độc lập có thể được sắp xếp khác nhau.

d. Nhất quán hàng đợi (FIFO Consistency)

Các bản ghi từ một tiến trình cụ thể sẽ được thấy theo đúng thứ tự chúng được gửi ra, nhưng không yêu cầu thứ tự giữa các tiến trình khác nhau.

So sánh strict consistency và sequential consistency

Tiêu chí	Strict consistency	Sequential consistency
Mô tả	Luôn đọc được giá trị mới nhất đã ghi	Các thao tác xuất hiện theo một thứ tự hợp lý toàn cục
Yêu cầu đồng bộ	Cực kỳ nghiêm ngặt, phải đồng bộ hóa thời gian thực	Không yêu cầu đồng bộ tức thì
Triển khai	Không khả thi trong môi trường phân tán thực tế	Có thể triển khai bằng cơ chế phân tán hợp lý
Độ mạnh	Mạnh nhất	Yếu hơn nhưng khả thi hơn

Sequential consistency yếu hơn vì cho phép chênh lệch nhỏ về thời gian giữa các bản sao, miễn là thứ tự vẫn hợp lệ toàn cục. Đây là mô hình thực tế hơn trong hệ thống phân tán vì tránh phải đồng bộ lập tức giữa các vùng địa lý xa nhau.

Lựa chọn mô hình nhất quán cho ứng dụng live scoring

Yêu cầu:

- Người dùng ở bất kỳ bản sao nào phải thấy điểm cập nhật trong vòng 2 giây.
- Không cần đảm bảo thứ tự bàn thắng giữa các trận khác nhau.

Chọn mô hình: Causal Consistency là phù hợp hơn.

Lý do:

- Không cần thứ tự tuyệt đối, nhưng trong mỗi trận đấu, điểm số phải cập nhật đúng theo thời gian thực.
- Cho phép xử lý song song giữa các trận, miễn là mỗi chuỗi sự kiện trong một trận vẫn theo đúng thứ tự nhân quả (ví dụ bàn thắng trước, cập nhật tỷ số sau).

Cấu hình CONIT (Continuous Consistency Model):

- Độ trễ tối đa: 2 giây (cài đặt $\Delta \leq 2s$)
- Độ lệch giá trị cho phép: 0 (điểm số phải chính xác)

- Vector clock: Mỗi trận đấu có một vector clock riêng để theo dõi thứ tự bàn thắng và cập nhật. Các bản sao đồng bộ theo vector này.

So sánh hiệu năng và tính triển khai: Sequential vs FIFO consistency

Tiêu chí	Sequential Consistency	FIFO Consistency
Yêu cầu thứ tự toàn cục	Có	Không
Yêu cầu giữa tiến trình	Tuân thủ cả thứ tự giữa tiến trình	Chỉ tuân thủ thứ tự trong từng tiến trình
Hiệu năng	Thấp hơn, vì cần đồng bộ thứ tự toàn cục	Cao hơn, vì giảm yêu cầu đồng bộ
Dễ triển khai	Khó hơn (phải có coordinator hoặc total ordering)	Dễ hơn (chỉ cần gửi đúng thứ tự từ từng node)

Khi nên chọn FIFO:

- Khi hệ thống có lưu lượng ghi cao, từ nhiều tiến trình không liên quan.
- Mức độ tương tranh giữa tiến trình thấp.
- Thứ tự giữa tiến trình không quan trọng, ví dụ log hành động từ nhiều cảm biến.

Chiến lược nhất quán dữ liệu cho nền tảng chat nhóm đa khu vực

Yêu cầu:

- Trong mỗi phòng chat, tin nhắn phải đến theo thứ tự gửi.
- Phòng chat khác nhau không cần đồng bộ hóa.
- Độ trễ trung bình ≤ 100 ms khi hiển thị tin nhắn mới.

Mô hình nhất quán được chọn:

Sequential consistency trên từng nhóm (phòng chat), kết hợp với causal consistency toàn hệ thống.

Cơ chế tổ chức đơn vị nhất quán:

- Mỗi phòng chat là một đơn vị nhất quán (CONIT unit) riêng.
- Mỗi phòng dùng vector clock hoặc Lamport timestamp để đánh dấu thứ tự tin nhắn.

Quy trình lan tỏa và đồng bộ:

- Tin nhắn được gửi đến server khu vực → đẩy đến các replica trong nhóm → gửi đến client.
- Đồng bộ bằng giao thức causal broadcast hoặc gRPC with timestamp.
- Tin nhắn từ người gửi được gắn timestamp và được sắp xếp lại theo thứ tự trước khi hiển thị tại client.

Xử lý bản sao bị trễ:

- Theo dõi độ trễ nhận tin (latency monitor).
- Nếu bản sao chậm > 100 ms so với timestamp chung:
    - Tạm ngừng phục vụ trực tiếp từ replica đó.

    - Đẩy client sang replica khác hoặc dùng bản tạm qua CDN/server gần nhất.
    - Khi đồng bộ xong, cho phép phục vụ lại.

Câu 3:

Định nghĩa eventual consistency và so sánh với continuous consistency

Nhất quán sau cùng (eventual consistency) là mô hình trong đó, nếu không có cập nhật mới, thì tất cả các bản sao dữ liệu cuối cùng sẽ đồng bộ và đạt cùng một trạng thái.

- Đặc điểm: không đảm bảo thời điểm đồng bộ cụ thể; chỉ hứa rằng trạng thái sẽ hội tụ "sau cùng".

Nhất quán liên tục (continuous consistency) là mô hình cho phép định lượng mức độ sai lệch cho phép giữa các bản sao về mặt:

- Giá trị dữ liệu (W),
- Số lượng cập nhật chênh lệch (N),
- Độ trễ thời gian (T).

Điểm khác biệt:

- *Eventual consistency* không ràng buộc thời gian, chỉ đảm bảo hội tụ.
- *Continuous consistency* có thể cấu hình mức lệch cho phép, nên kiểm soát mức độ nhất quán tốt hơn trong thời gian thực.

Mô hình nhất quán đọc đều (read-your-writes consistency)

Nguyên tắc hoạt động:

Một tiến trình (client) luôn thấy phiên bản mới nhất của dữ liệu mà chính nó đã ghi. Sau khi một client cập nhật dữ liệu, các lần đọc sau của chính client đó sẽ không thấy phiên bản cũ.

Tại sao đảm bảo không thấy dữ liệu cũ hơn lần đọc trước?

Vì client duy trì quá trình theo dõi các bản ghi đã ghi, hoặc hệ thống đảm bảo rằng mọi đọc của client đều được phục vụ từ replica đã thấy bản ghi trước đó. Điều này tránh hiện tượng "rollback" khi người dùng đọc thấy dữ liệu lỗi thời mà chính họ đã cập nhật.

Ứng dụng ghi chú offline – kết hợp read-your-writes và monotonic writes

Yêu cầu:

- Ghi chú tạo ra tại client luôn được ghi đúng thứ tự.
- Người dùng luôn thấy ghi chú vừa lưu, kể cả khi chuyển node.

Giải pháp:

Kết hợp hai mô hình:

- Monotonic writes: đảm bảo mọi ghi chú từ cùng một client được áp dụng theo đúng thứ tự.
- Read-your-writes: đảm bảo client luôn đọc thấy ghi chú mà mình đã ghi.

Cơ chế triển khai:

- Offline: client gán timestamp hoặc sequence ID cho mỗi ghi chú (ví dụ: local clock hoặc Lamport clock).
- Online: khi kết nối lại:
    - Đồng bộ ghi chú theo thứ tự ID.
    - Cập nhật replica gần nhất, đính kèm thông tin thứ tự.
- Khi đọc từ bất kỳ node nào: hệ thống chọn node đã nhận đầy đủ các bản ghi hoặc chờ cập nhật từ node chủ đã xử lý ghi trước đó.

So sánh Read-Your-Writes và Write-After-Read

Tiêu chí	Read-Your-Writes (RYW)	Write-After-Read (WAR)
Mô tả	Đảm bảo mọi đọc sau khi ghi luôn thấy kết quả đã ghi	Đảm bảo ghi luôn thực hiện trên phiên bản đã được đọc
Ứng dụng	Tránh rollback, phù hợp với trải nghiệm người dùng	Tránh ghi đè dữ liệu lỗi thời (conflict prevention)
Đồng bộ	RYW yêu cầu client đọc từ replica đã thấy bản ghi	WAR yêu cầu client ghi lên bản sao mới nhất đã đọc
Phức tạp client	Vừa phải (theo dõi replica đã ghi)	Cao hơn (phải lưu version đã đọc và xác thực trước khi ghi)

Khi nào nên chọn:

- Chọn RYW khi: hệ thống nhiều đọc, ít ghi; trải nghiệm người dùng cần phản hồi tức thì.
- Chọn WAR khi: hệ thống nhiều ghi song song, cần đảm bảo tránh xung đột do ghi lên phiên bản lỗi thời.

Giải pháp nhất quán lấy máy khách làm trung tâm cho chat nhóm offline-first

Mô hình kết hợp được chọn:

- Read-Your-Writes → đảm bảo người gửi thấy tin vừa gửi.
- Monotonic Writes → tin nhắn từ cùng người gửi theo đúng thứ tự.
- Eventual Consistency → giữa các client khác nhau (có thể chênh lệch tạm thời).
- Write-After-Read (khi cần): tránh gửi ghi đè tin nhắn cũ chưa thấy.

Cơ chế hoạt động:

- Khi client ghi tin nhắn offline:
    - Gán timestamp hoặc vector clock + ID người gửi.
    - Ghi vào local log.
- Khi online trở lại:
    - Tin nhắn được gửi theo thứ tự từ log, kèm metadata (user, timestamp).
    - Server áp dụng theo timestamp hoặc causal order.
    - Phát tán đến các client khác thông qua push hoặc polling.
- Khi client đọc:
    - Đọc local + replica gần nhất.
    - Nếu thấy thiếu tin, gửi request sync.

Xử lý xung đột ghi trùng:

- Tin nhắn từ 2 client khác nhau ghi vào cùng message ID (ví dụ: chỉnh sửa tin nhắn gốc):
    - Dùng last-write-wins nếu timestamp hợp lệ.
    - Hoặc giữ cả hai phiên bản và cảnh báo người dùng chọn giữ bản nào (manual conflict resolution).
    - Nếu tin nhắn khác ID nhưng trùng nội dung → vẫn ghi đầy đủ (không xung đột).

Đánh giá hiệu năng và trải nghiệm người dùng:

- Độ trễ tối đa: phụ thuộc vào thời gian đồng bộ. Có thể đảm bảo <200ms khi online.
- Không mất dữ liệu: nhờ log local, kể cả khi server không phản hồi.
- Mượt mà với người dùng: thấy ngay tin vừa gửi (RYW), có thể chat liên tục dù mất mạng.