HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG
KHOA CÔNG NGHỆ THÔNG TIN 1

_____ o0o _____

BÁO CÁO THỰC TẬP TỐT NGHIỆP

Đề tài: Những vấn đề trong hệ thống phân tán

Vũ Công Tuấn Dương MSSV: B22DCKH024

HÀ NỘI, 08/2026

MỤC LỤC

DANH MỤC HÌNH VẼ

DANH MỤC BẢNG BIỂU

CHƯƠNG 1. GIỚI THIỆU

1.1 Bối cảnh và bài toán

Trong quá trình phát triển phần mềm hiện đại, dữ liệu (data) trở thành tài nguyên trung tâm của hầu hết các ứng dụng. Với web, mobile app, phần mềm dạng dịch vụ (software as a service – SaaS) và các dịch vụ đám mây (cloud services), dữ liệu từ nhiều người dùng khác nhau được lưu trữ trong một hạ tầng dữ liệu dùng chung trên máy chủ. Dữ liệu từ hoạt động của người dùng, giao dịch kinh doanh, thiết bị và cảm biến cần được lưu trữ và sẵn sàng cho việc phân tích [1].

Khái niệm *data-intensive application* (ứng dụng có cường độ dữ liệu cao) chỉ các ứng dụng mà việc quản lý dữ liệu là một trong những thách thức chính của quá trình phát triển. Khác với *compute-intensive system* (hệ thống có cường độ tính toán cao) – nơi vấn đề chủ yếu là song song hóa một phép tính lớn – các ứng dụng data-intensive thường phải lo nhiều hơn về việc lưu trữ và xử lý khối lượng dữ liệu lớn, quản lý các thay đổi dữ liệu, đảm bảo nhất quán (consistency) trước sự cố và truy cập đồng thời (concurrency), cũng như giữ cho dịch vụ có tính sẵn sàng cao (high availability) [1].

Dữ liệu với khối lượng nhỏ, lưu được trên một máy duy nhất, thường dễ xử lý. Tuy nhiên khi khối lượng dữ liệu hoặc tốc độ truy vấn tăng lên, dữ liệu cần được phân phối (distribute) trên nhiều máy, kéo theo một loạt thách thức mới. Việc xây dựng và vận hành các hệ thống như vậy đối mặt với nhiều vấn đề nan giải. Báo cáo này tập trung vào việc khảo sát, phân loại và phân tích các vấn đề cơ bản trong hệ thống phân tán, đồng thời giới thiệu các giải pháp đã được nghiên cứu và áp dụng trong thực tế.

1.2 Mục tiêu của báo cáo

Mục tiêu của báo cáo này bao gồm:

1. Trình bày khái quát về hệ thống phân tán (distributed system): định nghĩa, lý do sử dụng, kiến trúc tham chiếu.
2. Phân loại và phân tích chi tiết các vấn đề điển hình mà hệ thống phân tán gặp phải, bao gồm vấn đề về mạng, đồng hồ, nhất quán dữ liệu, giao dịch, sao chép dữ liệu, phân mảnh dữ liệu và vận hành.
3. Giới thiệu các giải pháp tương ứng đã được công bố trong các công trình nghiên cứu và áp dụng trong các hệ thống công nghiệp thực tế như Google Spanner, Amazon DynamoDB, Apache Cassandra, MongoDB, CockroachDB, v.v.

4. Đúc kết bài học thiết kế cho các kỹ sư khi xây dựng hệ thống phân tán.

1.3 Phạm vi và phương pháp nghiên cứu

1.3.1 Phạm vi

Báo cáo giới hạn ở các khía cạnh *phi chức năng* (nonfunctional) của hệ thống dữ liệu: độ tin cậy (reliability), khả năng mở rộng (scalability), tính bảo trì (maintainability), và đặc biệt là các vấn đề phát sinh khi dữ liệu được đặt trên nhiều máy kết nối qua mạng. Các khía cạnh về mô hình dữ liệu, ngôn ngữ truy vấn, bảo mật, luật pháp nằm ngoài phạm vi chính, chỉ được nhắc đến khi cần thiết.

1.3.2 Phương pháp nghiên cứu

Phương pháp được sử dụng là nghiên cứu tài liệu (literature review) kết hợp nghiên cứu tình huống (case study):

- Nguồn tài liệu chính: cuốn sách *Designing Data-Intensive Applications* (DDIA), ấn bản thứ hai (2026) của Martin Kleppmann và Chris Riccomini [1].
- Nguồn bổ sung: các bài viết kỹ thuật của cộng đồng Vietnam Java User Group về định lý CAP, mô hình PACELC và các mô hình nhất quán [2], [3].
- Các công trình nghiên cứu gốc: định lý CAP (Gilbert & Lynch, 2002) [4], CAP Twelve Years Later (Brewer, 2012) [5], PACELC (Abadi, 2012) [6], Eventually Consistent (Vogels, 2008) [7].
- Các nghiên cứu tình huống từ sự cố thực tế của các hệ thống lớn như GitHub (sự cố ngày 21/10/2018) và các hệ thống dữ liệu đám mây như Amazon DynamoDB, Amazon S3, Microsoft Azure Cosmos DB.

1.4 Cấu trúc báo cáo

Báo cáo gồm 4 chương:

- Chương 1 – Giới thiệu: bối cảnh, mục tiêu, phạm vi, phương pháp.
- Chương 2 – Những vấn đề trong hệ thống phân tán: trình bày hệ thống các vấn đề, phân theo các nhóm: nền tảng lý thuyết (CAP, PACELC), vấn đề môi trường (mạng, đồng hồ, dừng tiến trình), vấn đề nhất quán dữ liệu, vấn đề giao dịch, vấn đề thiết kế (sao chép, phân mảnh, đồng bộ), và vấn đề vận hành.
- Chương 3 – Các giải pháp trong hệ thống phân tán: giới thiệu các giải pháp tương ứng với từng nhóm vấn đề, minh họa bằng các hệ thống công nghiệp.
- Chương 4 – Kết luận và bài học: tổng kết các vấn đề, giải pháp, và bài học thiết kế.

CHƯƠNG 2. NHỮNG VẤN ĐỀ TRONG HỆ THỐNG PHÂN TÁN

2.1 Nền tảng lý thuyết: định lý CAP và mô hình PACELC

2.1.1 Khái niệm hệ thống phân tán

Theo trình bày của Kleppmann và Riccomini [1], một hệ thống phân tán (distributed system) được hiểu là một hệ thống bao gồm nhiều máy giao tiếp với nhau thông qua mạng; mỗi tiến trình tham gia được gọi là một nút (node). Việc áp dụng kiến trúc phân tán xuất phát từ nhiều động lực khác nhau [1]:

- Phân tán tự nhiên (inherent distribution): đối với các ứng dụng có hai hoặc nhiều người dùng tương tác với nhau thông qua thiết bị riêng của mỗi người, sự trao đổi thông tin giữa các thiết bị buộc phải thực hiện qua mạng, do đó hệ thống trở nên phân tán một cách tất yếu.
- Mối quan hệ giữa các dịch vụ trong kiến trúc đám mây và microservices: khi dữ liệu được lưu trữ tại một dịch vụ nhưng được xử lý tại một dịch vụ khác, dữ liệu phải được truyền qua mạng giữa hai dịch vụ đó.
- Khả năng chịu lỗi và tính sẵn sàng cao (fault tolerance / high availability): việc sử dụng nhiều máy tạo ra sự dự phòng, cho phép hệ thống tiếp tục hoạt động khi một máy, nhiều máy, toàn bộ mạng hoặc một trung tâm dữ liệu gặp sự cố.
- Khả năng mở rộng (scalability): khi khối lượng dữ liệu hoặc yêu cầu tính toán vượt quá khả năng của một máy duy nhất, tải có thể được phân tán trên nhiều máy.
- Giảm độ trễ (latency): việc đặt máy chủ tại nhiều khu vực địa lý giúp người dùng được phục vụ từ máy chủ gần nhất, tránh thời gian truyền gói tin qua khoảng cách xa.
- Tính co giãn (elasticity): triển khai trên nền tảng đám mây cho phép điều chỉnh tài nguyên theo nhu cầu, tránh việc phải đầu tư cấu hình tối đa ngay cả khi hệ thống gần như nhàn rỗi.
- Phần cứng chuyên dụng, tuân thủ quy định pháp luật về lưu trữ dữ liệu (data residency), và tính bền vững (sustainability).

Tuy nhiên, kiến trúc phân tán cũng kéo theo những hệ quả bất lợi. Mọi yêu cầu (request) và lời gọi API đi qua mạng đều phải đối mặt với khả năng thất bại: mạng có thể bị gián đoạn, dịch vụ có thể bị quá tải hoặc ngừng hoạt động, do đó bất kỳ request nào cũng có thể bị quá hạn (timeout) mà không nhận được phản hồi. Trong trường hợp đó, không thể xác định được dịch vụ đã nhận request hay chưa, và việc

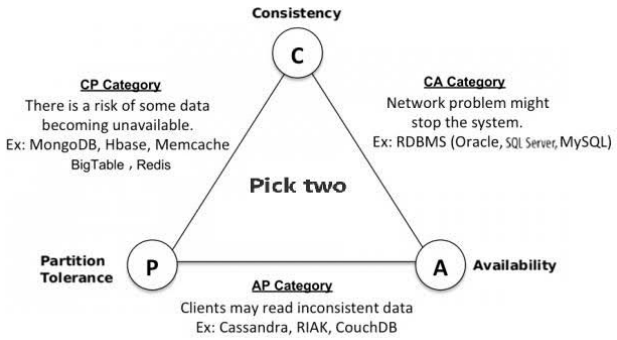
thực hiện lại (retry) request có thể không an toàn. Bên cạnh đó, việc gọi một dịch vụ khác qua mạng chậm hơn đáng kể so với gọi một hàm trong cùng một tiến trình; việc sử dụng nhiều nút không phải lúc nào cũng đem lại tốc độ cao hơn so với một máy duy nhất. Việc chẩn đoán và gỡ lỗi (debug) một hệ thống phân tán cũng phức tạp hơn nhiều so với hệ thống đơn nút [1].

2.1.2 Định lý CAP

Định lý CAP được Eric Brewer nêu ra vào năm 2000 và được Seth Gilbert cùng Nancy Lynch chứng minh chặt chẽ về mặt toán học vào năm 2002 [4]. Định lý phát biểu rằng một kho dữ liệu phân tán (distributed data store) không thể đồng thời bảo đảm trọn vẹn cả ba thuộc tính sau [2]:

- C – Consistency (nhất quán): thuộc tính này được hiểu cụ thể là linearizability: mọi phép đọc tại bất kỳ nút nào đều quan sát được phép ghi mới nhất đã thành công; toàn bộ hệ thống hành xử như thể chỉ tồn tại duy nhất một bản sao dữ liệu.
- A – Availability (tính sẵn sàng): mọi request được gửi tới một nút còn hoạt động đều nhận được phản hồi, không xảy ra lỗi và không bị treo. Đây là một định nghĩa toán học nghiêm ngặt, khác biệt về bản chất so với khái niệm "availability 99,99%" thường được ghi trong các thỏa thuận mức dịch vụ (SLA).
- P – Partition tolerance (khả năng chịu đựng phân chia): hệ thống vẫn vận hành được khi đường truyền giữa các nút bị gián đoạn, tức là khi các thông điệp giữa chúng bị mất hoặc bị trì hoãn vô hạn.

Hình 2.1 tóm tắt ba thuộc tính và mối quan hệ đánh đổi của định lý CAP.



Hình 2.1: Ba thuộc tính của định lý CAP và mối quan hệ đánh đổi

Cách tóm tắt phổ biến "chọn hai trong ba" (pick two) là cách hiểu sai phổ biến nhất đối với định lý CAP. Nguyên nhân là trong một hệ thống phân tán thực tế, P không phải một thuộc tính được lựa chọn: sự phân chia mạng (network partition) là sự kiện tất yếu xảy ra do nhiều nguyên nhân ngoài tầm kiểm soát, chẳng hạn cáp

biển bị đứt, thiết bị chuyển mạch khởi động lại, hoặc một đợt thu gom rác toàn bộ (Full GC) kéo dài khiến một nút bị quan sát như đã ngừng hoạt động. Việc tuyên bố "hệ thống của tôi chọn CA" thực chất tương đương với việc giả định mạng không bao giờ bị phân chia, một giả định thường bị môi trường production bác bỏ. Cách diễn đạt chính xác hơn như sau: trong điều kiện bình thường, hệ thống có thể đồng thời đạt được C và A; nhưng tại thời điểm partition xảy ra, hệ thống buộc phải lựa chọn một trong hai thuộc tính C hoặc A [2].

Chính Brewer, 12 năm sau đó, đã công bố bài viết *CAP Twelve Years Later* (2012) nhằm đính chính sự hiểu lầm này, đồng thời chỉ ra rằng trọng tâm thực sự của bài toán không nằm ở việc lựa chọn thuộc tính, mà nằm ở ba vấn đề: phát hiện partition như thế nào, xử lý hệ thống ra sao trong thời gian partition diễn ra, và khôi phục hệ thống bằng cách nào sau khi partition kết thúc [5].

Nghiên cứu tình huống 1 – Sự cố GitHub ngày 21 tháng 10 năm 2018 [2].

Tối ngày 21 tháng 10 năm 2018, đội ngũ kỹ thuật của GitHub thực hiện thay thế một thiết bị quang 100G tại khu vực bờ Đông nước Mỹ. Trong quá trình thay thế, đường kết nối giữa trung tâm dữ liệu bờ Đông và phần còn lại của mạng bị gián đoạn trong đúng 43 giây. Khoảng thời gian ngắn ngủi này đủ để kích hoạt chuỗi sự kiện sau: mạng bị gián đoạn $\rightarrow$ công cụ failover tự động (orchestrator) kết luận nhầm rằng cụm bờ Đông đã ngừng hoạt động $\rightarrow$ hệ thống thăng cấp (promote) cụm MySQL ở bờ Tây lên làm primary $\rightarrow$ khi mạng được khôi phục, mỗi bờ đang lưu trữ một tập các phép ghi mà bên kia chưa hề biết, dẫn đến hai nguồn sự thật (two sources of truth). GitHub buộc phải lựa chọn giữa hai phương án: tiếp tục vận hành với dữ liệu có thể sai lệch (ưu tiên A), hoặc dừng hệ thống để đối chiếu và khôi phục nhất quán (ưu tiên C). GitHub lựa chọn phương án thứ hai, và hậu quả là 24 giờ 11 phút suy giảm dịch vụ (service degradation): người dùng không thể đăng nhập, dữ liệu cũ được trả về cho người dùng, và website gần như chỉ hoạt động ở chế độ đọc (read-only). Từ sự cố trên có thể rút ra hai bài học quan trọng:

- Một timeout không thể phân biệt ba trạng thái khác nhau: nút ở xa đã chết hẳn, nút ở xa vẫn hoạt động nhưng đang chậm, hoặc đường truyền giữa hai bên bị đứt. Việc đặt ngưỡng timeout ngắn để phát hiện sự cố sớm dễ dẫn đến failover nhầm, đúng như tình huống GitHub: mạng chỉ gián đoạn 43 giây nhưng hệ thống kết luận nhầm là nút chết hẳn và thực hiện thăng cấp primary xuyên giữa hai bờ.
- Giai đoạn khôi phục (reconciliation) sau sự cố mới là giai đoạn tốn thời gian nhất: tỉ lệ giữa 43 giây partition và 24 giờ 11 phút khôi phục cho thấy rõ điều này. Cả CAP lẫn PACELC chỉ mô tả hành vi của hệ thống trong lúc đang có

sự cố, không đề cập tới quá trình khôi phục nhất quán sau sự cố.

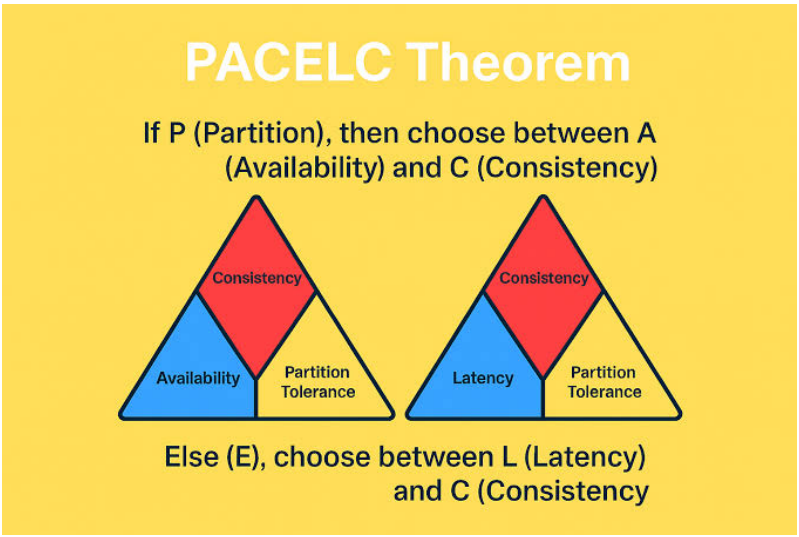
2.1.3 Mô hình PACELC

Định lý CAP chỉ mô tả hành vi của hệ thống tại thời điểm mạng bị phân chia, trong khi partition trên thực tế là sự kiện hiếm gặp. Năm 2010, Daniel Abadi đề xuất mô hình PACELC nhằm trả lời câu hỏi: trong khoảng thời gian mạng hoạt động bình thường, hệ thống đang đánh đổi điều gì? [6]

Mô hình PACELC được đọc như sau:

- P: nếu có Partition, hệ thống phải đánh đổi giữa A (Availability) và C (Consistency), đúng như nội dung định lý CAP;
- ELC: Else (tức khi mạng vẫn bình thường), hệ thống tiếp tục phải đánh đổi giữa L (Latency) và C (Consistency).

Cấu trúc hai vế P–ELC của mô hình được minh họa trong Hình 2.2.



Hình 2.2: Cấu trúc hai vế của mô hình PACELC

Vế thứ hai của mô hình (ELC) chính là phần trả lời cho vấn đề đã đặt ra: muốn đạt nhất quán thì phải phối hợp, muốn phối hợp thì phải trao đổi thông tin, và việc trao đổi thông tin đòi hỏi thời gian khứ hồi (round-trip). Cụ thể, để bảo đảm các bản sao (replica) đều đã nhận được một phép ghi, hệ thống phải chờ ít nhất một vòng khứ hồi tới replica trước khi xác nhận hoàn tất cho người dùng [2].

Chi phí độ trễ của vòng khứ hồi là một đại lượng đo được: với các replica nằm trong cùng một vùng (region) nhưng khác khu vực sẵn sàng (availability zone), vòng khứ hồi chỉ vào khoảng dưới một đến vài mili-giây; nhưng khi các replica đặt tại hai region khác nhau, vòng khứ hồi Singapore–Tokyo vào khoảng 70 ms, còn từ Việt Nam tới us-east-1 thường vượt quá 200 ms. Từng ấy mili-giây được cộng

trực tiếp vào mỗi phép ghi [2]. Độ trễ này có ảnh hưởng kinh tế trực tiếp: một thử nghiệm nội bộ tại Amazon năm 2006 cho thấy cứ tăng thêm 100 ms độ trễ thì doanh số giảm khoảng 1% [2].

Khi soi chiếu qua mô hình PACELC, các hệ thống cơ sở dữ liệu quen thuộc được phân loại khá rõ ràng [2]:

- Apache Cassandra và Riak: thuộc nhóm PA/EL: khi có partition thì ưu tiên availability, khi bình thường thì ưu tiên latency.
- Google Cloud Spanner: thuộc nhóm PC/EC: ưu tiên consistency trong cả hai tình huống; hệ thống sử dụng giao thức Paxos và TrueTime để duy trì strong consistency, chấp nhận trả giá bằng latency và đôi khi từ chối phục vụ trong các tình huống hiếm.
- Amazon DynamoDB: khó gán nhãn cố định: bên dưới là leader-based replication chạy Multi-Paxos; mọi phép ghi phải đi qua leader và chỉ được xác nhận sau khi một quorum đã ghi write-ahead log; strongly consistent read chỉ do leader phục vụ, follower chỉ trả về eventually consistent read. Do đó, mức nhất quán của DynamoDB được quyết định ở từng request, không phải do hệ quản trị lựa chọn sẵn.

Trade-off theo PACELC không nằm ở cấp độ hệ quản trị cơ sở dữ liệu, mà ở cấp độ từng thao tác nghiệp vụ. Phép đọc số dư tài khoản trước khi thực hiện lệnh rút tiền và phép đọc số lượt thích của một bài đăng không có lý do gì phải dùng chung một mức nhất quán [2].

2.1.4 Các khái niệm nền tảng

Phân biệt fault và failure. Trong bối cảnh đánh giá độ tin cậy, cần phân biệt hai khái niệm [1]:

- Fault (sự cố cục bộ / lỗi bộ phận): một bộ phận cụ thể của hệ thống ngừng hoạt động đúng chức năng, ví dụ một ổ đĩa cứng hỏng, một máy chủ sập, hoặc một dịch vụ ngoài mà hệ thống phụ thuộc gặp sự cố.
- Failure (sự cố ở mức hệ thống / hỏng hóc tổng thể): toàn bộ hệ thống ngừng cung cấp dịch vụ cần thiết cho người dùng, tức là hệ thống không đạt mục tiêu mức dịch vụ (SLO) đã đặt ra.

Hai khái niệm này thường bị nhầm lẫn vì chúng mô tả cùng một hiện tượng nhưng ở các cấp độ khác nhau. Một hệ thống có khả năng chịu lỗi tốt (fault-tolerant) về mặt lý thuyết có thể hấp thụ các fault cục bộ mà không chuyển hóa thành failure ở cấp hệ thống.

Độ trễ và percentile. Thời gian phản hồi của một dịch vụ không phải là một con số duy nhất mà là một phân phối (distribution) các giá trị, do sự biến thiên của độ trễ mạng (gọi là jitter). Người ta thường sử dụng percentile thay vì giá trị trung bình để mô tả phân phối này [1]:

- Median (p50): thời gian phản hồi nằm ở vị trí giữa của danh sách các thời gian phản hồi được sắp xếp; một nửa số request trả về nhanh hơn giá trị này, một nửa chậm hơn.
- Các percentile cao: p95, p99, p99.9 mô tả các request chậm nhất (outliers). Giá trị trung bình (mean) hữu ích để ước lượng thông lượng, nhưng không phản ánh được trải nghiệm của đa số người dùng.

Các percentile cao còn được gọi là tail latency (độ trễ đuôi), có ảnh hưởng trực tiếp tới trải nghiệm người dùng. Amazon mô tả yêu cầu thời gian phản hồi cho các dịch vụ nội bộ theo p99.9, dù thuộc tính này chỉ ảnh hưởng tới 1 trong 1.000 request, vì những khách hàng nhận được request chậm nhất thường là những khách hàng có nhiều dữ liệu nhất, tức những khách hàng có giá trị nhất [1].

Vấn đề khuếch đại độ trễ đuôi (tail latency amplification): khi một request của người dùng cuối phải gọi nhiều dịch vụ nền (backend) song song, request vẫn phải chờ lời gọi chậm nhất hoàn tất. Chỉ cần một lời gọi chậm là toàn bộ request của người dùng trở nên chậm. Dù tỉ lệ lời gọi nền bị chậm là nhỏ, xác suất có ít nhất một lời gọi chậm trong một request sẽ tăng lên khi số lượng lời gọi tăng, khiến tỉ lệ request chậm của người dùng tăng vượt mức tỉ lệ lời gọi chậm đơn lẻ [1]. Vấn đề này được phân tích sâu hơn ở Mục 2.6

2.2 Vấn đề môi trường: mạng, đồng hồ và sự dừng tiến trình

Các vấn đề thuộc nhóm này được trình bày chi tiết trong Chương 9 của DDIA [1]. Đây là các vấn đề mang tính nền tảng hạ tầng, áp đặt lên mọi hệ thống phân tán bất kể kiến trúc hay công nghệ được lựa chọn.

2.2.1 Mạng không tin cậy

Trong hệ thống phân tán, mọi sự giao tiếp đều diễn ra qua mạng, và mạng là một thành phần tiềm ẩn nhiều dạng lỗi khác nhau [1]:

- Request có thể bị mất (ví dụ do mạng quá tải).
- Request có thể bị xếp hàng đợi và chỉ được gửi đi sau một khoảng thời gian.
- Nút ở xa có thể đã xử lý request nhưng phản hồi bị mất trên đường truyền về.
- Nút ở xa có thể đã xử lý và gửi phản hồi, nhưng phản hồi bị trì hoãn quá lâu.
- Nút ở xa có thể đã ngừng hoạt động, nhưng sự cố này chỉ được phát hiện sau

một khoảng thời gian.

- Nút ở xa có thể đã xử lý request, nhưng quá trình xử lý bị gián đoạn trước khi phản hồi được gửi đi.

Hệ quả về mặt lý luận quan trọng nhất là: không thể suy diễn rằng một timeout nghĩa là nút ở xa đã ngừng hoạt động. Một timeout chỉ cho biết "không nhận được phản hồi trong khoảng thời gian quy định", không cung cấp bất kỳ thông tin nào về việc request đã được xử lý hay chưa. Điều này làm cho việc thực hiện retry trở nên không an toàn: nếu không biết request đã được xử lý hay chưa, việc gửi lại request có thể gây ra các tác dụng phụ trùng lặp [1].

Một số đặc điểm cụ thể của mạng máy tính cần được lưu ý:

- Biến thiên độ trễ (jitter): cùng một loại request, thời gian phản hồi có thể thay đổi đáng kể giữa các lần gọi.
- Khó khăn trong việc chọn ngưỡng timeout: ngưỡng quá ngắn dễ dẫn tới kết luận nhầm rằng nút còn hoạt động đã chết (đúng như sự cố GitHub đã phân tích ở Mục 2.1.2); ngưỡng quá dài lại làm chậm việc phát hiện sự cố thực sự. Không tồn tại một giá trị phù hợp cho mọi hoàn cảnh.
- Phân biệt mạng đồng bộ và phi đồng bộ: mạng Internet nói chung, bao gồm mạng trung tâm dữ liệu dựa trên Ethernet và giao thức TCP/IP, là mạng không đồng bộ (asynchronous): gói tin có thể bị trễ với thời gian không xác định và phải bị hủy bỏ khi trễ vượt quá giới hạn. Việc giả định rằng các nút phản hồi trong một khoảng thời gian xác định (như mạng điện thoại) là không đúng đối với mạng máy tính [1].
- Phát hiện lỗi (fault detection): việc phát hiện một nút đã ngừng hoạt động thường dựa trên cơ chế timeout hoặc heartbeat; tuy nhiên như đã phân tích, timeout không phân biệt được ba trạng thái "chết hẳn", "đang chậm" và "đường bị đứt". Một số hệ thống sử dụng các kỹ thuật nâng cao hơn, chẳng hạn Phi Accrual Failure Detector, trong đó mỗi nút được gán một "mức độ nghi ngờ" (suspicion level) tăng dần theo thời gian không nhận được heartbeat, cho phép các thành phần khác nhau sử dụng các ngưỡng phản ứng khác nhau.

2.2.2 Đồng hồ không tin cậy

Đồng hồ là một nguồn vấn đề lớn trong hệ thống phân tán vì hai nhóm nguyên nhân chính [1]:

- Đồng hồ thời gian thực (time-of-day clock): đo thời gian tuyệt đối theo giờ trong ngày, thường được đồng bộ qua giao thức NTP (Network Time Protocol). Tuy nhiên việc đồng bộ NTP không hoàn hảo: đồng hồ time-of-day có thể nhảy

về trước hoặc nhảy về sau (ví dụ khi NTP điều chỉnh sai lệch), do đó không phù hợp để đo khoảng thời gian trôi qua.

- Đồng hồ đơn điệu (monotonic clock): đo thời gian trôi qua và không thể nhảy về sau, phù hợp để đo elapsed time, nhưng không cung cấp giá trị thời điểm tuyệt đối và ý nghĩa của nó không nhất quán giữa các nút.

Hệ quả trực tiếp: việc sử dụng time-of-day clock để xác định thứ tự giữa các sự kiện là không đáng tin cậy, vì đồng hồ của hai máy có thể lệch nhau hàng trăm mili-giây hoặc hơn. Một phép ghi có timestamp nhỏ hơn thực tế có thể xảy ra muộn hơn một phép ghi khác có timestamp lớn hơn [1]. Vấn đề này liên quan trực tiếp tới thuật toán giải quyết xung đột "last write wins" (LWW) sẽ được bàn chi tiết ở Mục 3.4: khi đồng hồ hai máy lệch nhau, khái niệm "ai ghi sau" không còn là khái niệm đáng tin cậy [2].

Nghiên cứu tình huống 2 – Giây nhuận (leap second) năm 2012 [1]. Ngày 30 tháng 6 năm 2012, một giây nhuận được chèn thêm vào hệ thống thời gian; do một lỗi trong Linux kernel, nhiều ứng dụng Java bị treo đồng loạt, kéo theo nhiều dịch vụ Internet ngừng hoạt động. Sự kiện này là một ví dụ điển hình về lỗi phần mềm mang tính hệ thống (systematic software fault), trong đó mọi nút gặp lỗi cùng một lúc trong cùng một hoàn cảnh.

Đối với các hệ thống đòi hỏi giới hạn độ trễ chặt chẽ (chẳng hạn giao dịch tài chính), NTP trở thành vấn đề nghiêm trọng: nếu đồng hồ bị lệch và bị điều chỉnh đột ngột, các quyết định dựa trên thời gian có thể trở nên sai. Một số hệ thống hiện đại đã áp dụng giải pháp phần cứng, như Google Spanner sử dụng đồng hồ nguyên tử và GPS (phân tích ở Mục 3.3), để đạt độ chính xác đồng hồ cao hơn nhiều so với NTP thông thường.

2.2.3 Sự dừng tiến trình (process pauses)

Một nút còn hoạt động không có nghĩa là nút đó đang xử lý các request của hệ thống. Tiến trình có thể bị dừng đột ngột trong một khoảng thời gian dài vì các nguyên nhân sau [1]:

- Thu gom rác (garbage collection – GC): máy ảo Java và các ngôn ngữ có cơ chế GC có thể dừng toàn bộ tiến trình trong vài chục giây để thực hiện thu gom rác, đặc biệt khi vùng nhớ heap có kích thước lớn. Một pha Full GC kéo dài vài chục giây đủ khiến nút đó được quan sát như đã ngừng hoạt động trong mắt toàn bộ hệ thống [2].
- Đình chỉ máy ảo: máy ảo bị đình chỉ (suspend) và khởi động lại (resume) sau một khoảng thời gian.

- Chuyển đổi ngữ cảnh của hệ điều hành: tiến trình không được hệ điều hành lên lịch thực thi trong một thời gian dài.
- Chờ I/O hoặc chờ lock.
- Steal time: trên máy chủ ảo, tài nguyên CPU bị trích ra để phục vụ các máy chủ ảo khác chạy trên cùng một máy chủ vật lý.

Hệ quả: một tiến trình vừa thức dậy sau một khoảng dừng dài có thể nhận thấy đồng hồ thời gian thực đã trôi qua rất nhiều, các lease đã hết hạn, v.v. Điều này dẫn tới vấn đề về "kiến thức, sự thật và lời nói dối" được phân tích ở mục dưới đây.

2.2.4 Kiến thức, sự thật và quy tắc đa số

Trong một hệ thống phân tán, một nút chỉ có thể *biết* những gì nó tự quan sát được: giá trị các biến cục bộ của nó, đồng hồ của nó, và các thông điệp nó nhận được. Mọi thông tin khác, chẳng hạn trạng thái của nút khác hay ý định của nút khác, chỉ là suy đoán [1]. Do không thể phân biệt được một nút chết với một nút đang tạm dừng, một nút không thể tự quyết định "mình là leader" chỉ dựa trên việc không nhận được sự phản đối nào trong khoảng thời gian timeout: timeout chỉ cho biết "không nghe thấy", không chứng minh được "không ai nói" [1].

Giải pháp được sử dụng là quy tắc đa số (the majority rules). Khi một quyết định đòi hỏi sự đồng ý của đa số các nút (quorum), một nút không thể "chen ngang" khi một quorum khác đã ra quyết định, bởi vì hai quorum khác nhau trong cùng một tập nút buộc phải có phần giao nhau, tức là có ít nhất một nút chung. Đây chính là nền tảng của các thuật toán consensus (đồng thuận), sẽ được phân tích ở Mục 3.5 [1].

2.2.5 Khóa phân tán (distributed lock) và lease

Một kỹ thuật phổ biến để bảo vệ tài nguyên dùng chung là khóa (lock) kết hợp với lease. Một nút giữ lock trong một khoảng thời gian nhất định (lease); nếu nút đó không gia hạn (renew) kịp thời, lock sẽ hết hạn và một nút khác có thể giành được [1].

Vấn đề nảy sinh khi kết hợp lease với process pause: giả sử nút A giành được lock, sau đó bị GC dừng lại 30 giây; lease của A hết hạn; nút B giành được lock và bắt đầu thao tác trên tài nguyên; lúc này nút A tỉnh dậy và tưởng rằng mình vẫn đang giữ lock, tiếp tục ghi dữ liệu, gây hỏng dữ liệu. Giải pháp chuẩn cho vấn đề này là fencing token:

- Mỗi lần lock được cấp, hệ thống phát hành một số nguyên tăng dần (token).
- Nút giữ lock phải đính kèm token vào mọi yêu cầu ghi gửi tới máy chủ tài

nguyên.

- Máy chủ tài nguyên lưu giữ token lớn nhất đã nhận được và từ chối mọi yêu cầu có token nhỏ hơn giá trị đã thấy.

Như vậy, dù nút A có tỉnh dậy và tiếp tục gửi yêu cầu ghi, token của nó đã cũ và bị từ chối. Sự kết hợp giữa lease (giới hạn thời gian giữ lock) và fencing token (giới hạn quyền ghi theo token) là phòng thủ chuẩn chống lại hiện tượng process pause [1]. Chi tiết về cơ chế này được trình bày ở Mục 3.5

2.2.6 Lỗi Byzantine

Các vấn đề được phân tích ở trên đều nằm trong giả định rằng các nút "trung thực": chúng có thể chậm, có thể sập, có thể mất thông điệp, nhưng không cố tình gây hại. Lỗi Byzantine (Byzantine fault) là trường hợp một nút cố tình gửi thông tin sai lệch hoặc gây hại cho hệ thống, có thể do bị xâm nhập, do lỗi phần mềm gây hành vi sai, hoặc do tác nhân cố ý. Phần lớn các hệ thống cơ sở dữ liệu trong một tổ chức giả định không tồn tại lỗi Byzantine, vì các thuật toán chịu lỗi Byzantine (Byzantine fault tolerant – BFT) rất tốn kém và phức tạp. BFT chủ yếu cần thiết trong các hệ thống công khai có nhiều bên không tin tưởng lẫn nhau, điển hình là blockchain [1].

2.2.7 Mô hình hệ thống (system model)

Để lý luận một cách chặt chẽ về các vấn đề nêu trên, các nhà nghiên cứu định nghĩa system model, tức là tập hợp các giả định về hành vi của mạng và các nút. Các mô hình phổ biến bao gồm [1]:

- Synchronous model (mô hình đồng bộ): giả định độ trễ mạng, thời gian dừng tiến trình và độ lệch đồng hồ đều có giới hạn trên xác định.
- Partially synchronous model (mô hình bán đồng bộ): phần lớn thời gian hoạt động như mô hình đồng bộ, nhưng thỉnh thoảng vượt quá các giới hạn đã định. Đây là mô hình thực tế nhất cho hầu hết hệ thống.
- Asynchronous model (mô hình phi đồng bộ): không tồn tại bất kỳ giới hạn nào; hệ thống phải hoạt động đúng ngay cả khi mạng có độ trễ vô hạn.

Tương tự, đối với hành vi lỗi của các nút có các mô hình: crash-stop (nút sập thì ngừng hẳn), crash-recovery (nút sập rồi khởi động lại, có thể mất trạng thái trong bộ nhớ nhưng vẫn còn dữ liệu trên đĩa), và Byzantine.

Hệ quả thiết kế quan trọng: một thuật toán có thể *không tồn tại* trong mô hình này nhưng lại *tồn tại* trong mô hình khác. Ví dụ điển hình là định lý FLP (Fischer, Lynch, Paterson): thuật toán consensus không thể tồn tại trong mô hình asyn-

chronous thuần túy nếu chỉ có một nút có thể bị lỗi; tuy nhiên trong mô hình partially synchronous với crash-recovery, các thuật toán như Paxos và Raft hoạt động được [1].

2.2.8 Phương pháp hình thức và kiểm thử ngẫu nhiên

Do lỗi trong hệ thống phân tán thường tiềm ẩn (latent) và chỉ bộc lộ trong những điều kiện hiếm gặp, việc chỉ dựa vào kiểm thử thủ công thông thường là không đủ. DDIA nhấn mạnh hai nhóm kỹ thuật kiểm chứng [1]:

- Phương pháp hình thức (formal methods): mô hình hóa thuật toán bằng ngôn ngữ hình thức và sử dụng các công cụ kiểm tra mô hình (model checker), điển hình là TLA+ do Leslie Lamport phát triển, để quét toàn bộ không gian trạng thái của thuật toán nhằm phát hiện lỗi.
- Kiểm thử ngẫu nhiên (randomized testing): chạy thuật toán với một lượng lớn đầu vào ngẫu nhiên và đối chiếu kết quả với một mô hình tham chiếu (property-based testing).
- Kiểm thử mô phỏng xác định (deterministic simulation testing – DST): chạy toàn bộ hệ thống trong một trình mô phỏng kiểm soát được thời gian, mạng và đồng hồ, sao cho mỗi lần chạy là xác định (deterministic), nhờ đó mọi lỗi phát hiện được đều có thể tái tạo (reproduce). FoundationDB và TigerBeetle là hai hệ thống tiêu biểu xây dựng và kiểm chứng hệ thống của mình theo hướng này (chi tiết ở Mục 3.7).

2.3 Vấn đề nhất quán dữ liệu và replication lag

2.3.1 Khái niệm replication và eventual consistency

Replication (sao chép dữ liệu) là kỹ thuật giữ một bản sao của cùng một tập dữ liệu trên nhiều máy kết nối qua mạng, nhằm ba mục đích chính: giữ dữ liệu gần người dùng về mặt địa lý (giảm độ trễ truy cập), cho phép hệ thống tiếp tục hoạt động khi một bộ phận gặp sự cố (tăng tính sẵn sàng và độ bền), và tăng thông lượng phục vụ các truy vấn đọc [1].

Khi dữ liệu được sao chép bất đồng bộ (asynchronous replication), nghĩa là leader ghi dữ liệu xong rồi gửi log thay đổi cho các follower mà không chờ xác nhận, một ứng dụng đọc từ follower có thể quan sát dữ liệu cũ nếu follower chưa đuổi kịp. Kết quả là: chạy cùng một truy vấn trên leader và follower tại cùng một thời điểm có thể thu được hai kết quả khác nhau, bởi vì không phải mọi phép ghi đều đã được phản ánh trên follower. Trạng thái không nhất quán này mang tính tạm thời: nếu ngừng ghi dữ liệu và chờ một khoảng thời gian, các follower cuối cùng sẽ đuổi kịp và nhất quán với leader. Hiệu ứng này được gọi là eventual consistency

(nhất quán cuối cùng) [1].

Thuật ngữ "eventual consistency" do Douglas Terry và cộng sự đặt ra, sau đó được Werner Vogels phổ biến rộng rãi qua bài viết *Eventually Consistent* (2008) [7], trở thành khẩu hiệu của nhiều dự án NoSQL. Chữ "eventually" (cuối cùng) được chọn một cách chủ ý mơ hồ: không tồn tại giới hạn cho mức độ tụt lại của một replica. Trong điều kiện hoạt động bình thường, replication lag (độ trễ sao chép) chỉ là một phần của giây; nhưng khi hệ thống hoạt động gần hết công suất hoặc mạng gặp sự cố, lag có thể tăng lên nhiều giây, thậm chí nhiều phút [1].

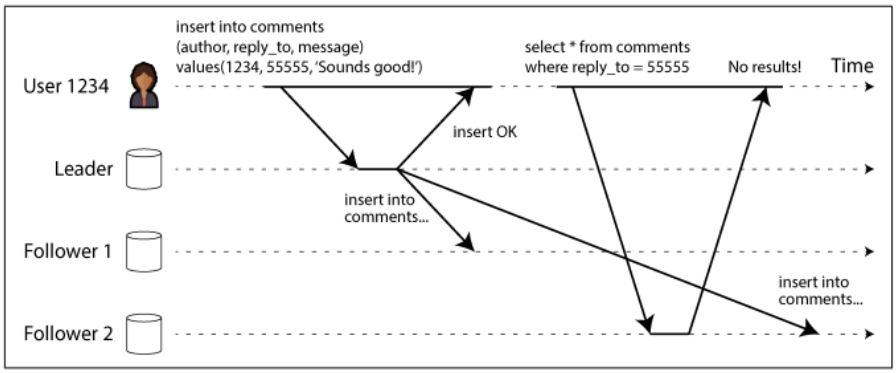
Eventual consistency không phải đặc trưng riêng của các hệ NoSQL. Các follower trong một cơ sở dữ liệu quan hệ sao chép bất đồng bộ cũng có cùng đặc tính này [1].

2.3.2 Ba dị thường điển hình do replication lag gây ra

DDIA liệt kê ba dị thường (anomaly) điển hình khi đọc dữ liệu từ follower bất đồng bộ [1]: read-your-writes, monotonic reads và consistent prefix reads. Ba dị thường này lần lượt được minh họa trong Hình 2.3, Hình 2.4 và Hình 2.5.

a) Reading your own writes (đọc được phép ghi của chính mình).

Nhiều ứng dụng cho phép người dùng gửi dữ liệu rồi xem lại dữ liệu vừa gửi. Nếu phép ghi đi qua leader nhưng phép đọc lại được định tuyến tới một follower chưa nhận được phép ghi đó, người dùng có cảm giác dữ liệu mình vừa gửi đã bị mất, một trải nghiệm rất tiêu cực. Do đó cần một bảo đảm gọi là read-after-write consistency (hay read-your-writes consistency): nếu người dùng tải lại trang, họ luôn quan sát được các cập nhật do chính họ thực hiện. Bảo đảm này không cam kết bất cứ điều gì về các phép ghi của người dùng khác [1].



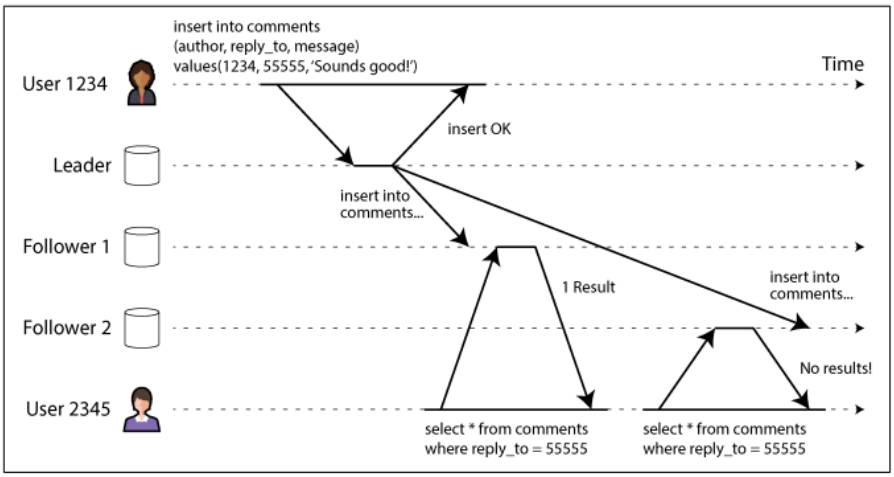
Hình 2.3: Dị thường read-your-writes: phép đọc định tuyến tới follower chưa nhận được phép ghi của người dùng

Một số kỹ thuật triển khai read-after-write consistency: đọc từ leader hoặc follower đồng bộ đối với những dữ liệu người dùng có thể đã sửa (ví dụ hồ sơ cá

nhân chỉ chủ sở hữu mới sửa được, nên luôn đọc hồ sơ của chính người dùng từ leader); theo dõi thời điểm cập nhật cuối và trong một khoảng thời gian sau đó đọc từ leader; hoặc client ghi nhớ timestamp (logical timestamp hoặc hệ thống clock thực) của phép ghi gần nhất và hệ thống bảo đảm replica phục vụ phép đọc phản ánh các cập nhật tới mốc đó. Nếu replica chưa đủ mới, hoặc phép đọc được chuyển sang replica khác, hoặc truy vấn chờ replica bắt kịp [1]. Vấn đề trở nên phức tạp hơn khi cùng một người dùng truy cập dịch vụ từ nhiều thiết bị (máy tính và điện thoại): metadata về lần cập nhật cuối phải được tập trung hóa, và các request của cùng một người dùng có thể cần được định tuyến về cùng một region [1].

b) Monotonic reads (đọc đơn điệu).

Người dùng có thể quan sát hiện tượng thời gian dường như quay ngược lại: lần đọc đầu tiên trả về một comment mới được thêm, lần đọc thứ hai (được định tuyến tới một follower có lag lớn hơn) không trả về comment đó, khiến comment "biến mất". Hiện tượng này rất dễ xảy ra khi mỗi request được định tuyến ngẫu nhiên tới một follower khác nhau. Monotonic reads là bảo đảm: nếu một người dùng thực hiện nhiều lần đọc liên tiếp, họ sẽ không quan sát thấy dữ liệu cũ hơn sau khi đã quan sát dữ liệu mới, tức là thời gian không quay ngược lại. Một cách triển khai đơn giản: bảo đảm mọi phép đọc của cùng một người dùng được phục vụ bởi cùng một replica, chẳng hạn chọn replica theo giá trị băm của user ID thay vì chọn ngẫu nhiên [1].

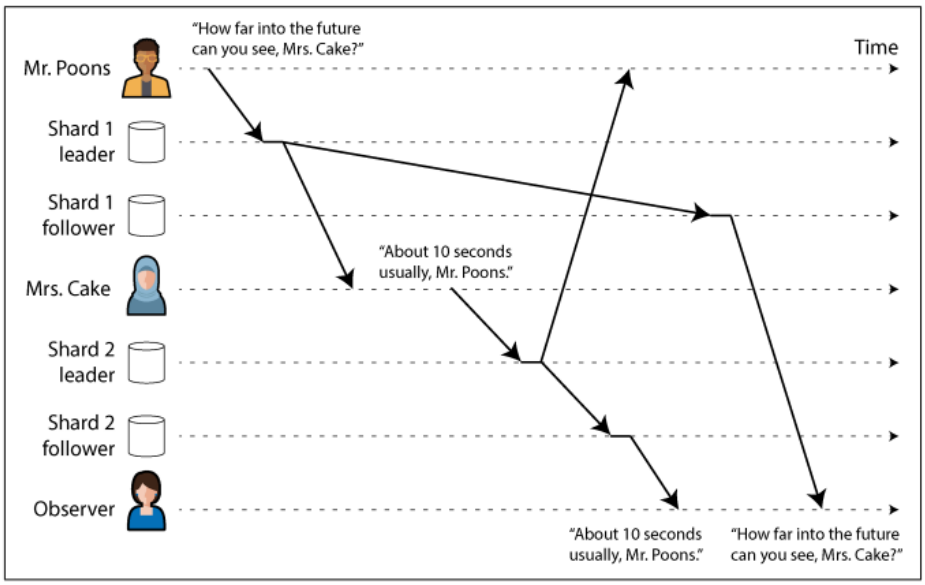


Hình 2.4: Dị thường monotonic reads: lần đọc sau từ follower lag hơn khiến comment "biến mất"

c) Consistent prefix reads (đọc theo tiền tố nhất quán).

Xét một cuộc hội thoại giữa hai người: người thứ nhất đặt câu hỏi, người thứ hai trả lời. Câu trả lời của người thứ hai phụ thuộc nhân quả vào câu hỏi của người

thứ nhất. Nếu lời đáp được chuyển qua một follower có lag nhỏ, còn câu hỏi được chuyển qua một follower có lag lớn, một người quan sát thứ ba có thể nghe được câu trả lời trước khi câu hỏi được đặt ra, tức là một nghịch lý nhân quả gây bối rối cho người quan sát. Consistent prefix reads là bảo đảm: nếu một chuỗi các phép ghi xảy ra theo một thứ tự nhất định, bất kỳ ai đọc các phép ghi đó sẽ quan sát thấy chúng xuất hiện theo đúng thứ tự đó [1].



Hình 2.5: Dị thường consistent prefix reads: câu trả lời được quan sát trước câu hỏi

Vấn đề này đặc biệt nghiêm trọng trong các cơ sở dữ liệu phân mảnh (sharded): các shard hoạt động độc lập nên không tồn tại một thứ tự toàn cục cho các phép ghi; người đọc có thể quan sát thấy một phần cơ sở dữ liệu ở trạng thái cũ trong khi phần khác đã ở trạng thái mới. Một giải pháp là bảo đảm các phép ghi có quan hệ nhân quả được ghi vào cùng một shard; tuy nhiên với một số ứng dụng, điều này không thể thực hiện hiệu quả. Giải pháp khác là theo dõi tường minh các phụ thuộc nhân quả, dựa trên quan hệ happens-before (sẽ được đề cập ở Mục 3.1) [1].

2.3.3 Các mô hình nhất quán: bức tranh tổng quan

Từ eventual consistency yếu nhất tới linearizability mạnh nhất tồn tại một dải (spectrum) gồm nhiều mức độ, không phải một công tắc hai trạng thái "strong vs eventual" [3]. Dải này được trình bày chi tiết trong Mục 3.1 Điểm cần lưu ý ở đây: mỗi mức chặt hơn đều đắt hơn, và sự vi phạm các bảo đảm nhất quán không gây exception, không làm tăng tỉ lệ lỗi, không làm dashboard hiển thị cảnh báo; chúng chỉ biểu hiện thành các phản hồi rời rạc kiểu "tôi bấm lưu mà không thấy gì". Đại lượng duy nhất đo được khi đó là replication lag, vốn chỉ phản ánh gián tiếp vấn đề [3].

2.4 Vấn đề giao dịch và cô lập

2.4.1 Khái niệm giao dịch (transaction) và ACID

Giao dịch (transaction) là một nhóm các phép đọc và ghi tạo thành một đơn vị logic. Giao dịch cung cấp một lớp trừu tượng cho phép ứng dụng giả định rằng các vấn đề về truy cập đồng thời và một số loại sự cố phần cứng/phần mềm không tồn tại: một lớp lớn các lỗi được thu gọn thành một lần hủy bỏ giao dịch (transaction abort), và ứng dụng chỉ cần thử lại [1]. Giao dịch không cần thiết cho mọi ứng dụng; ứng dụng chỉ đọc và ghi một record đơn lẻ có thể không cần đến nó. Nhưng với các mô hình truy cập phức tạp, giao dịch giảm đáng kể số trường hợp lỗi mà nhà phát triển phải xử lý [1].

Khái niệm ACID (Atomicity, Consistency, Isolation, Durability) cần được hiểu một cách chính xác:

- Atomicity (tính nguyên tử): khả năng hủy bỏ. Nếu một giao dịch bị hủy giữa chừng, không có phần nào của giao dịch được ghi lại; không tồn tại trạng thái "ghi nửa chừng".
- Consistency (tính nhất quán): chữ C trong ACID khác về bản chất với chữ C trong CAP. Nó nói về tính đúng đắn của dữ liệu so với các ràng buộc (constraint) mà ứng dụng đặt ra, không nói về việc các replica có quan sát thấy cùng một sự thật hay không [2].
- Isolation (tính cô lập): các giao dịch thực thi đồng thời không gây nhiễu lẫn nhau.
- Durability (tính bền vững): một khi giao dịch commit thành công, dữ liệu không bị mất dù máy sập, mất điện hay gặp sự cố khác.

2.4.2 Các mức cô lập yếu và các dị thường

Khi nhiều giao dịch thực thi đồng thời mà không có sự cô lập thích hợp, có thể xảy ra các dị thường sau. DDIA tổng hợp chúng trong Bảng 8-1 [1]:

Bảng 2.1: Các mức cô lập và các dị thường

Mức cô lập	Dirty reads	Read skew	Phantom reads	Lost updates	Write skew
Read uncommitted	có thể	có thể	có thể	có thể	có thể
Read committed	ngăn chặn	có thể	có thể	có thể	có thể
Snapshot isolation	ngăn chặn	ngăn chặn	ngăn chặn	tùy triển khai	có thể
Serializable	ngăn chặn	ngăn chặn	ngăn chặn	ngăn chặn	ngăn chặn

Giải thích các dị thường [1]:

- Dirty read: một client đọc phép ghi của client khác trước khi phép ghi đó được commit. Mức read committed trở lên ngăn chặn dị thường này.
- Dirty write: một client ghi đè dữ liệu mà client khác đã ghi nhưng chưa commit. Hầu hết mọi triển khai đều ngăn chặn dị thường này (do đó nó không xuất hiện trong bảng).
- Read skew (hay nonrepeatable read): một client quan sát các phần khác nhau của cơ sở dữ liệu tại các thời điểm khác nhau. Dị thường này thường được ngăn chặn bằng snapshot isolation.
- Phantom read: một giao dịch đọc các đối tượng khớp với một điều kiện tìm kiếm; một client khác thực hiện phép ghi làm thay đổi kết quả của phép tìm kiếm đó. Snapshot isolation ngăn chặn phantom read thông thường, nhưng phantom trong bối cảnh write skew đòi hỏi xử lý đặc biệt (khóa vùng chỉ mục – index-range lock).
- Lost update (mất cập nhật): hai client đồng thời thực hiện chu trình read-modify-write; một client ghi đè phép ghi của client kia mà không gộp các thay đổi, dẫn tới mất dữ liệu. Một số triển khai snapshot isolation tự động ngăn chặn dị thường này, số khác đòi hỏi khóa thủ công (SELECT FOR UPDATE).
- Write skew: một giao dịch đọc dữ liệu, đưa ra quyết định dựa trên giá trị đã đọc, rồi ghi quyết định vào cơ sở dữ liệu; tuy nhiên tại thời điểm ghi, tiền đề của quyết định không còn đúng nữa. Chỉ có mức serializable ngăn chặn được dị thường này.

Nghiên cứu tình huống 3 – Write skew giữa hai bác sĩ trực [1]. Xét một hệ thống quản lý lịch trực của bệnh viện với ràng buộc: tại mọi thời điểm phải có ít nhất một bác sĩ trực. Hai bác sĩ đang trực đồng thời vào hệ thống xin nghỉ. Mỗi giao dịch kiểm tra điều kiện "số bác sĩ trực >= 2" (sau khi mình nghỉ còn lại 1, vẫn hợp lệ) rồi đăng ký nghỉ. Với snapshot isolation, cả hai giao dịch đọc cùng một snapshot (2 bác sĩ đang trực), đều kết luận điều kiện được thỏa mãn và đều commit, khiến không còn bác sĩ nào trực, vi phạm ràng buộc. Đây là write skew: hai giao dịch đọc cùng một tập đối tượng, mỗi giao dịch ghi vào một đối tượng khác nhau, và không giao dịch nào quan sát được phép ghi của giao dịch kia do đọc từ snapshot cũ. Dị thường này rất khó phát hiện vì nó không gây ra lỗi hiển thị hay exception.

Nghiên cứu tình huống 4 – Mất cập nhật dẫn tới phá sản Flexcoin [1]. Năm 2014, sàn giao dịch tiền mã hóa Flexcoin bị mất khoảng 600 bitcoin do lỗi đua (race condition): nhiều request rút tiền chạy đồng thời, mỗi request đọc số dư,

kiểm tra điều kiện đủ tiền, rồi ghi số dư mới. Do thiếu cơ chế cô lập đúng đắn, các request "đọc trước khi ghi" khiến số dư bị ghi đè chồng chéo, dẫn tới mất tiền thật và cuối cùng sàn phải đóng cửa. Đây là ví dụ kinh điển về lost update gây tổn thất tài chính thực tế.

2.4.3 Serializability và ba cách triển khai

Các mức cô lập yếu bảo vệ được một số dị thường nhưng để ứng dụng tự xử lý các dị thường còn lại (ví dụ bằng khóa tường minh). Riêng write skew, dị thường không gây lỗi hiển thị và rất khó phát hiện, chỉ có mức serializable mới ngăn chặn hoàn toàn. DDIA giới thiệu ba cách triển khai serializable [1]:

- Thực thi tuần tự thực sự (actual serial execution): các giao dịch được thực thi lần lượt trên một lõi CPU duy nhất (hoặc mỗi shard một lõi). Điều kiện tiên quyết: mỗi giao dịch phải thực thi rất nhanh (thường thông qua stored procedures), thông lượng đủ thấp để xử lý trên một lõi hoặc được phân mảnh, và toàn bộ dữ liệu phải nằm trong bộ nhớ. Đây là giải pháp đơn giản và hiệu quả khi các điều kiện trên được đáp ứng, nhưng khả năng mở rộng bị giới hạn bởi một lõi (hoặc một shard).
- Two-phase locking (2PL): chuẩn triển khai serializability trong nhiều thập kỷ. Một giao dịch phải có shared lock để đọc và exclusive lock để ghi; các lock được giữ tới cuối giao dịch. Do hai giao dịch không thể đồng thời giữ các lock xung đột, thứ tự thực thi tương đương với một thứ tự tuần tự nào đó. Nhược điểm: hiệu năng kém (đặc biệt dưới tải cao), dễ xảy ra deadlock phải phát hiện và hủy bỏ.
- Serializable snapshot isolation (SSI): thuật toán tương đối mới, tiếp cận theo hướng tối ưu (optimistic): giao dịch thực thi không bị chặn như snapshot isolation; tuy nhiên khi chuẩn bị commit, hệ thống kiểm tra xem thực thi có thực sự serializable hay không, và nếu không thì hủy bỏ (abort) một trong các giao dịch liên quan. SSI phát hiện write skew bằng cách theo dõi các "tiền đề" (premises) mà giao dịch đã dựa vào và các truy cập có khả năng tạo ra vòng phụ thuộc (dependency cycle). Chi tiết trong Mục 3.2

2.4.4 Giao dịch phân tán: bài toán atomic commit

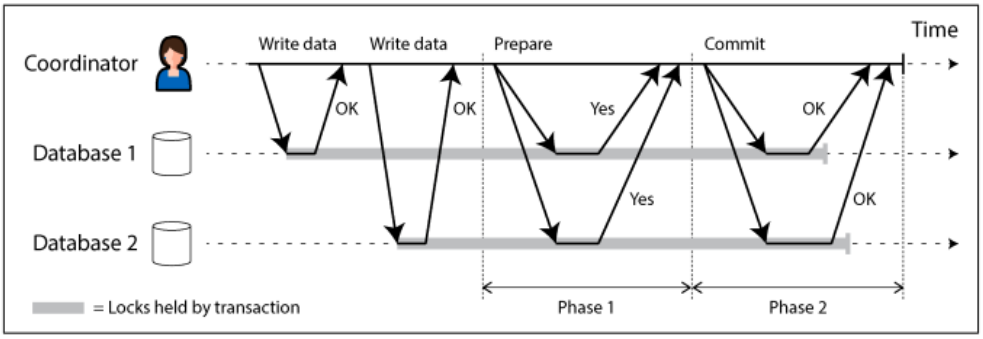
Khi dữ liệu nằm trên nhiều nút, một giao dịch có thể cần đọc và ghi dữ liệu trên nhiều nút khác nhau. Để bảo đảm atomic commit, nghĩa là hoặc tất cả các nút cùng commit hoặc tất cả cùng hủy bỏ, cần một thuật toán phối hợp. Thuật toán kinh điển là two-phase commit (2PC) [1]. Quy trình 2PC gồm các bước:

1. Ứng dụng xin một transaction ID duy nhất toàn cục từ một thành phần gọi là

coordinator (hay transaction manager).

2. Ứng dụng thực hiện các giao dịch đơn nút trên từng participant, mỗi giao dịch đơn nút được gắn transaction ID toàn cục.
3. Khi ứng dụng sẵn sàng commit, coordinator gửi request prepare tới mọi participant.
4. Mỗi participant khi nhận request prepare phải bảo đảm có thể commit được trong mọi hoàn cảnh: ghi toàn bộ dữ liệu của giao dịch ra đĩa, kiểm tra các xung đột và vi phạm ràng buộc. Việc trả lời "yes" nghĩa là từ bỏ quyền hủy bỏ (nhưng chưa commit).
5. Coordinator ghi quyết định commit/abort của mình ra đĩa (điểm commit – commit point); chỉ commit khi mọi participant đều trả lời "yes".
6. Coordinator gửi request commit/abort tới mọi participant và retry không giới hạn cho tới khi thành công.

Quy trình trao đổi giữa coordinator và các participant được minh họa trong Hình 2.6.



Hình 2.6: Quy trình two-phase commit giữa coordinator và các participant

2PC có hai "điểm không thể quay lại": khi participant nói "yes", nó không được từ chối commit sau đó; và khi coordinator đã ghi quyết định, quyết định đó là bất khả hồi [1].

Vấn đề chính của 2PC – coordinator failure [1]: Nếu coordinator sập trước khi gửi request prepare, mọi participant có thể hủy bỏ một cách an toàn. Nhưng nếu coordinator sập sau khi một participant đã trả lời "yes", participant đó không thể tự hủy bỏ (vì đã hứa sẽ commit) và cũng không thể tự commit (vì có thể một participant khác đã hủy bỏ). Nó buộc phải chờ coordinator. Giao dịch ở trạng thái này được gọi là in doubt (không chắc chắn). Timeout không giúp ích trong trường hợp này: nếu participant tự hủy bỏ sau timeout, nó sẽ không nhất quán với participant khác đã commit; ngược lại, tự commit sau timeout cũng không an toàn vì một par-

ticipant khác có thể đã hủy bỏ. Cách duy nhất để 2PC hoàn tất là chờ coordinator phục hồi, và điều này có nghĩa giao dịch có thể bị treo trong một khoảng thời gian không xác định. Chính vì lý do này, 2PC được coi là vấn đề khi triển khai qua nhiều công nghệ lưu trữ khác nhau (XA transactions): rất nhạy cảm với sự cố của coordinator và mã ứng dụng điều khiển giao dịch, và tương tác kém với các cơ chế kiểm soát đồng thời [1].

Khi các nút tham gia đều chạy cùng một phần mềm cơ sở dữ liệu, giao dịch phân tán nội bộ có thể hoạt động tốt (ví dụ giao dịch qua nhiều shard của cùng một database). Vấn đề chỉ bộc lộ khi giao dịch vượt qua ranh giới giữa các công nghệ lưu trữ khác nhau; khi đó, giải pháp được khuyến nghị là sử dụng idempotence để đạt exactly-once semantics mà không cần atomic commit xuyên công nghệ (phân tích ở Mục 3.3) [1].

2.4.5 Consensus và tổng thứ tự

Consensus (đồng thuận) là bài toán làm sao để nhiều nút thống nhất về một giá trị một cách an toàn trong môi trường phân tán, ví dụ "nút nào là leader". Về mặt lý thuyết, các thuật toán consensus như Paxos và Raft tương đương với total order broadcast (phát sóng tổng thứ tự): mọi nút nhận được cùng một chuỗi thông điệp theo cùng một thứ tự. Từ tổng thứ tự này có thể xây dựng nên [1]:

- Replicated state machine (máy trạng thái nhân bản): mọi replica áp dụng cùng một log theo cùng một thứ tự và đạt cùng một trạng thái.
- Bầu leader an toàn: chỉ một nút được bầu, không xảy ra split brain.
- Lưu trữ tuyến tính hóa: các thao tác so sánh-và-gán (compare-and-set) với tính chất linearizability.
- Các dịch vụ điều phối: như ZooKeeper và etcd.

Về mặt lý thuyết, một hệ thống single-leader không có cơ chế failover tự động (leader chết thì hệ thống đứng yên) chỉ bảo đảm được tính an toàn (safety) chứ không bảo đảm tính sống (liveness), tức không cam kết mọi request cuối cùng sẽ được phục vụ. Trong thực tế người ta thường muốn cả hai, nhưng đạt được đồng thời cả hai chính là bài toán consensus, một bài toán tốn kém và nhiều sắc thái được phân tích trong Mục 3.5 [1].

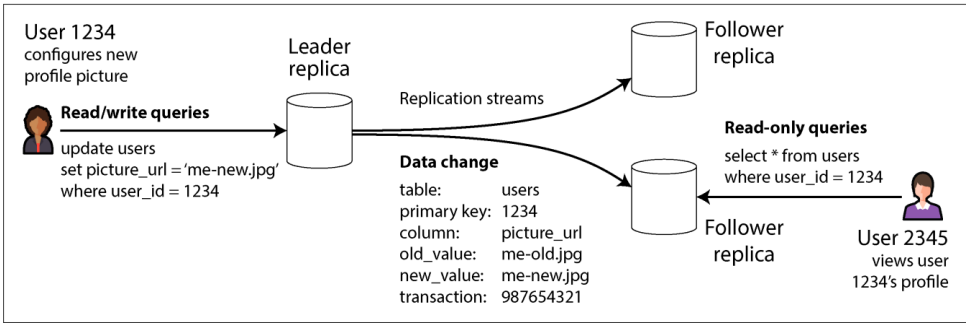
2.5 Vấn đề thiết kế: sao chép, phân mảnh và đồng bộ dữ liệu

2.5.1 Ba họ thuật toán sao chép

Có ba họ thuật toán sao chép cơ bản [1], được minh họa lần lượt trong Hình 2.7, Hình 2.8 và Hình 2.9:

a) Single-leader replication (sao chép một leader).

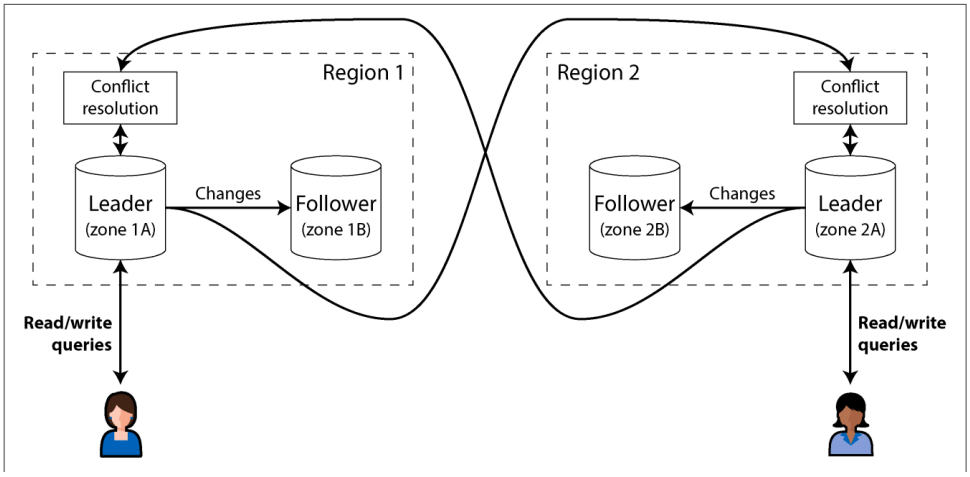
Mọi phép ghi phải đi qua leader; leader gửi log thay đổi cho các follower. Đây là mô hình được sử dụng rộng rãi trong PostgreSQL, MySQL, Oracle Data Guard, SQL Server Always On, MongoDB, DynamoDB, Kafka; các thuật toán consensus như Raft (CockroachDB, TiDB, etcd, RabbitMQ quorum queues) cũng dựa trên một leader và tự bầu leader mới khi leader cũ gặp sự cố [1].



Hình 2.7: Single-leader replication: mọi phép ghi đi qua leader rồi được sao chép tới các follower

b) Multi-leader replication (sao chép nhiều leader).

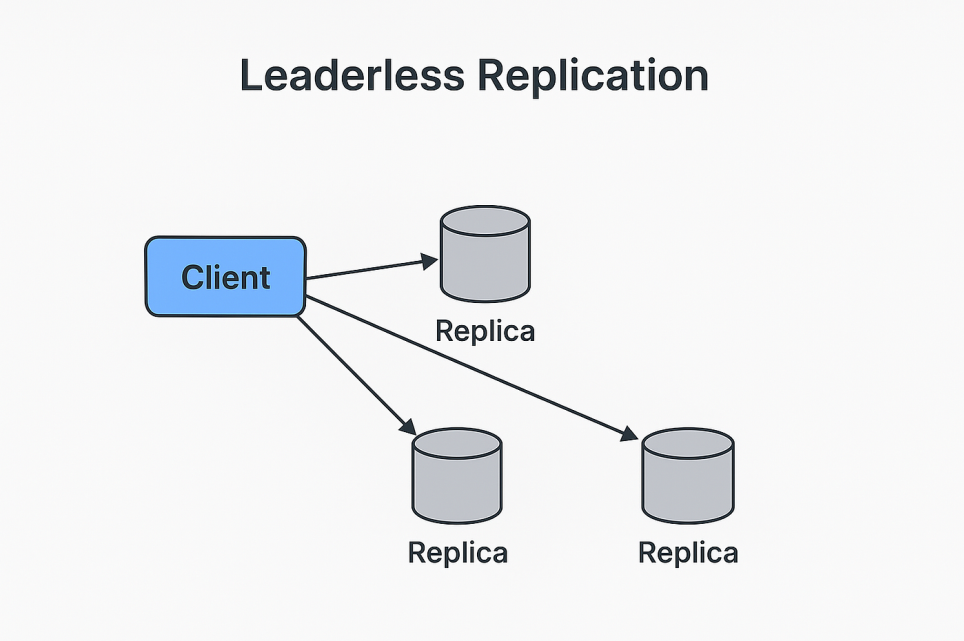
Nhiều leader cùng xử lý phép ghi và đồng bộ với nhau, thường qua mạng liên vùng địa lý. Được hỗ trợ bởi MySQL, Oracle, SQL Server, YugabyteDB; ở một số hệ thống là tính năng retrofit (pglogical, EDB Postgres Distributed, Redis Enterprise) [1].



Hình 2.8: Multi-leader replication: nhiều leader ở các trung tâm dữ liệu đồng bộ chéo phép ghi

c) Leaderless replication (sao chép không leader).

Client ghi trực tiếp tới nhiều nút và đọc từ nhiều nút, sử dụng quorum. Được sử dụng bởi DynamoDB, Cassandra, Riak, Voldemort [1].



Hình 2.9: Leaderless replication: client ghi và đọc trực tiếp tới nhiều nút theo cơ chế quorum

2.5.2 Vấn đề đồng bộ so với bất đồng bộ trong sao chép

Một câu hỏi thiết kế quan trọng là sao chép được thực hiện đồng bộ hay bất đồng bộ [1]:

- Sao chép đồng bộ (synchronous): follower được bảo đảm có bản sao dữ liệu cập nhật; nếu leader sập, dữ liệu chắc chắn còn trên follower. Tuy nhiên nếu follower đồng bộ không phản hồi (sập, mạng lỗi...), leader phải chặn mọi phép ghi và chờ, vì một nút hỏng là toàn hệ thống đứng yên.
- Sao chép bán đồng bộ (semisynchronous): một follower được cấu hình đồng bộ, các follower còn lại bất đồng bộ; nếu follower đồng bộ gặp sự cố, một follower bất đồng bộ được nâng cấp lên vai trò đồng bộ. Cấu hình này bảo đảm dữ liệu được sao chép trên ít nhất hai nút (leader và một follower đồng bộ).
- Quorum đa số: một đa số các replica (ví dụ 3 trên 5) được cập nhật đồng bộ, số còn lại bất đồng bộ; ví dụ của quorum, thường dùng trong các hệ thống eventual consistency hoặc các hệ thống sử dụng consensus.
- Sao chép hoàn toàn bất đồng bộ: leader tiếp tục nhận phép ghi dù mọi follower đều tụt lại phía sau. Hệ quả nghiêm trọng: khi leader sập và không phục hồi được, các phép ghi chưa kịp sao chép sẽ bị mất, dù chúng đã được xác nhận thành công cho client. Đây là một hệ quả về độ bền cần được cân nhắc, nhưng cấu hình này vẫn được sử dụng rộng rãi, đặc biệt khi có nhiều follower hoặc

chúng nằm phân tán về mặt địa lý [1].

2.5.3 Failover và split brain

Khi leader gặp sự cố, một follower được thăng cấp (promote) làm leader mới. Quá trình failover này tiềm ẩn các vấn đề nghiêm trọng [1]:

- Mất phép ghi: nếu follower được thăng cấp chưa nhận được các phép ghi cuối của leader cũ (do sao chép bất đồng bộ), các phép ghi đó bị mất, người dùng có thể quan sát thấy dữ liệu mình vừa ghi "biến mất".
- Split brain (chẻ não): hai nút đồng thời nghĩ rằng mình là leader và cùng nhận phép ghi, dẫn tới hai nguồn sự thật, đúng như tình huống sự cố GitHub được phân tích ở Mục 2.1.2
- Rủi ro do process pause và timeout: một leader cũ còn sống nhưng đang bị GC dừng có thể bị nhầm là đã chết và bị thay thế, dẫn tới hai leader tồn tại đồng thời.

Để failover an toàn cần giải quyết hai vấn đề: phát hiện leader chết và bầu leader mới một cách an toàn (bài toán consensus), và ngăn chặn leader cũ tiếp tục ghi dữ liệu sau khi đã bị thay thế (fencing token, Mục 3.5) [1].

2.5.4 Xung đột ghi trong multi-leader replication

Trong multi-leader replication, hai leader khác nhau có thể xử lý các phép ghi xung đột, ví dụ cùng một username được đăng ký trên hai leader, hoặc hai lần đặt phòng trùng thời gian cho cùng một phòng họp. Mỗi phép ghi riêng lẻ đều hợp lệ, nhưng khi được xem xét cùng nhau, chúng vi phạm ràng buộc. Đây là một giới hạn cơ bản của hệ thống phân tán [1]: multi-leader replication không thể bảo đảm các ràng buộc như "tài khoản không được âm" hay "username không được trùng". Nếu ứng dụng cần các ràng buộc dạng này, nên sử dụng single-leader replication [1].

Khi xung đột xảy ra, hệ thống phải áp dụng một luật giải quyết xung đột (conflict resolution):

- LWW – last write wins (ai ghi sau thì thắng): thuật toán đơn giản nhất, nhưng cần được xem xét thận trọng. LWW loại bỏ một trong các giá trị xung đột mà không phát sinh thông báo nào, và khi đồng hồ của hai máy lệch nhau, khái niệm "ai ghi sau" không còn đáng tin cậy [2]. Để phân định thứ tự một cách xác định, nên gắn kèm một unique ID vào timestamp [1].
- Lưu trữ các giá trị đồng thời (siblings) và giải quyết thủ công: hệ thống lưu tất cả các giá trị được ghi đồng thời (gọi là siblings) và trả về tất cả cho ứng dụng quyết định; đây là cách tiếp cận của CouchDB. Nhược điểm: API của cơ sở dữ

liệu thay đổi (một trường kiểu chuỗi trở thành một tập các chuỗi), và việc yêu cầu người dùng tự hợp nhất các siblings tốn công sức [1].

- Giải quyết tự động – strong eventual consistency (SEC): hợp nhất các phép ghi đồng thời về một trạng thái hội tụ, bất kể thứ tự các phép ghi đến. Hai họ thuật toán chính là CRDT (conflict-free replicated datatypes) và OT (operational transformation) [1], được phân tích chi tiết ở Mục 3.4

Nghiên cứu tình huống 5 – Giỏ hàng Amazon [1]. Giỏ hàng của Amazon từng cho phép cập nhật đồng thời rồi hợp nhất các siblings bằng phép hợp của tập hợp (set union). Hệ quả: nếu khách hàng xóa một món hàng khỏi giỏ ở một thiết bị trong khi một sibling ở thiết bị khác vẫn còn món hàng cũ, món hàng đã bị xóa tự động xuất hiện lại trong giỏ hàng. Đây là một lỗi kinh điển của thuật toán hợp nhất đơn giản: thuật toán phải theo dõi cả các thao tác xóa (tombstone), không chỉ các thao tác thêm, nếu không các phần tử đã bị xóa sẽ tái xuất hiện.

2.5.5 Vấn đề về topology trong multi-leader replication

Topology mô tả các đường truyền thông qua đó phép ghi được lan truyền từ nút này sang nút khác. Với hơn hai leader, có thể dùng các topology khác nhau: all-to-all (mỗi leader gửi tới mọi leader khác), circular (mỗi nút nhận từ một nút và chuyển tiếp tới một nút), star (một nút gốc chuyển tiếp tới mọi nút khác). Các vấn đề liên quan [1]:

- Với topology circular và star, nếu một nút hỏng, dòng sao chép giữa các nút khác bị gián đoạn, khiến chúng không thể liên lạc cho tới khi nút hỏng được khắc phục.
- Với topology all-to-all, do một số đường mạng nhanh hơn đường khác, các thông điệp sao chép có thể vượt qua nhau (overtake): một nút có thể nhận được phép cập nhật trước khi nhận được phép chèn tương ứng (phép cập nhật tác động lên một dòng dữ liệu mà từ góc nhìn của nút đó chưa tồn tại). Đây là vấn đề về nhân quả, tương tự như consistent prefix reads (Mục 2.3.2) [1].

2.5.6 Phân mảnh dữ liệu (sharding) và các vấn đề liên quan

Sharding (partitioning, phân mảnh) là kỹ thuật chia tập dữ liệu thành các shard độc lập, mỗi shard được đặt trên một nút riêng, nhằm mở rộng dung lượng và thông lượng vượt quá khả năng của một máy đơn [1]. Có hai cách cơ bản để phân mảnh dữ liệu kiểu key-value [1]:

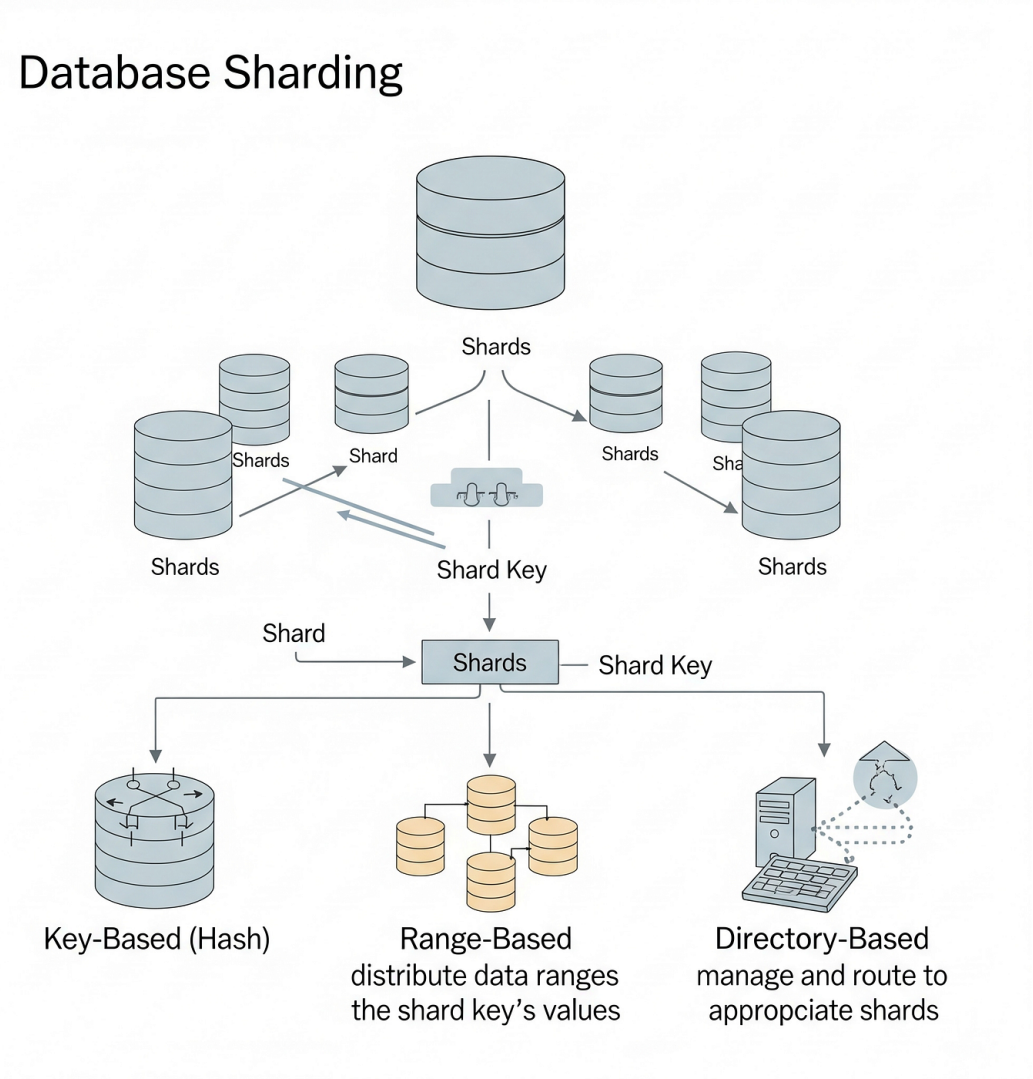
- Sharding theo dải khóa (key range): gán một dải khóa liên tục cho mỗi shard, giống như các tập của một bộ bách khoa toàn thư. Ưu điểm: truy vấn theo dải (range scan) hiệu quả. Nhược điểm nghiêm trọng: dễ tạo ra hot shard, vì nếu

khóa là timestamp, mọi phép ghi trong tháng đều đổ về shard của tháng đó, các shard khác nhàn rỗi. Khắc phục: thêm tiền tố (ví dụ sensor ID) vào trước timestamp để phân tán phép ghi, nhưng đánh đổi bằng việc mất khả năng truy vấn dải trên nhiều sensor cùng lúc [1].

- Sharding theo băm khóa (hash of key): băm khóa rồi gán vào shard, giúp dữ liệu phân phối đều hơn ngay cả khi các khóa ban đầu lệch. Các biến thể [1]:
  - Băm modulo số nút (hash mod N): đơn giản, nhưng khi số nút N thay đổi, hầu hết các khóa phải di chuyển từ nút này sang nút khác, khiến việc rebalancing kém hiệu quả.
  - Số lượng shard cố định (fixed number of shards): tạo nhiều shard hơn số nút ngay từ đầu (ví dụ 1.000 shard trên 10 nút), mỗi nút giữ nhiều shard; khi thêm/loại nút chỉ di chuyển nguyên shard. Hạn chế: không thể có nhiều nút hơn số shard; thay đổi số shard đòi hỏi resharding tốn kém.
  - Băm theo dải giá trị hash (hash range): mỗi shard giữ một dải giá trị hash; shard có thể tách/ghép linh hoạt theo khối lượng dữ liệu. Cassandra và ScyllaDB chia không gian giá trị hash thành nhiều dải với ranh giới ngẫu nhiên (Cassandra mặc định 16 dải mỗi nút, ScyllaDB 256 dải mỗi nút); mỗi nút giữ nhiều dải để bù trừ sự chênh lệch kích thước giữa các dải [1].

Các vấn đề điển hình của sharding [1]:

Hai cách phân mảnh cơ bản được so sánh trong Hình 2.10.



Hình 2.10: Phân mảnh theo dải khóa và theo băm khóa

- Skew và hot spots: dù băm phân phối đều các khóa, tải thực tế có thể vẫn lệch; một bài đăng của người nổi tiếng với hàng triệu người theo dõi có thể gây ra một "cơn bão" đọc/ghi vào một khóa duy nhất (hot key). Giải pháp ở tầng ứng dụng: thêm một vài chữ số ngẫu nhiên vào đầu hoặc cuối khóa để phân tán phép ghi ra nhiều shard, nhưng đánh đổi bằng việc các phép đọc phải truy vấn và gộp kết quả từ tất cả các khóa đã chia; và kỹ thuật này chỉ có ý nghĩa với một số ít khóa nóng, đòi hỏi cơ chế theo dõi khóa nào đang được tách. Một số dịch vụ đám mây xử lý tự động vấn đề này (Amazon gọi là heat management / adaptive capacity) [1].
- Rebalancing rủi ro: việc di chuyển shard tốn kém tài nguyên mạng và CPU. Nếu kết hợp rebalancing tự động với phát hiện lỗi tự động, có thể gây cascading failure: một nút quá tải tạm thời chậm phản hồi bị các nút khác nghi ngờ là chết, kích hoạt quá trình rebalance tự động, làm tăng thêm tải cho các nút khác và mạng, khiến tình hình tồi tệ hơn và nhiều nút khác cũng bị nghi ngờ sai. Vì

lý do này, nhiều hệ thống chủ trương giữ con người trong vòng lặp (human in the loop) cho quá trình rebalancing [1].

- Request routing: làm sao xác định khóa nằm trên nút nào? Ba cách tiếp cận: (1) client gửi request tới nút bất kỳ, nút đó chuyển tiếp request tới nút đúng; (2) một tầng định tuyến (routing tier) đóng vai trò load balancer nhận biết shard; (3) client tự biết ánh xạ shard→nút (smart client). Các hệ thống thường dùng một coordination service (ZooKeeper, etcd) để lưu trữ ánh xạ shard→nút và thông báo thay đổi cho tầng định tuyến, đảm bảo không xảy ra split brain nhờ consensus. Vấn đề đặt ra: trong thời gian một shard được chuyển từ nút này sang nút khác, phải xử lý các request đang in flight tới nút cũ [1].
- Secondary index với sharding: chỉ mục phụ không ánh xạ gọn vào các shard. Hai hướng tiếp cận: local index (mỗi shard tự xây dựng chỉ mục trên phần dữ liệu của mình, khiến phép đọc theo chỉ mục phụ phải truy vấn mọi shard rồi gộp kết quả, tốn kém và dễ gây khuếch đại độ trễ đuôi) và global index (chỉ mục được phân mảnh theo giá trị của từ khóa chỉ mục, phép đọc gọn hơn nhưng phép ghi phải cập nhật chỉ mục trên nhiều shard, thường là bất đồng bộ) [1].

2.5.7 Dual writes và bài toán đồng bộ nhiều hệ lưu trữ

Trong thực tế, một ứng dụng thường phải kết hợp nhiều hệ thống lưu trữ: cơ sở dữ liệu OLTP phục vụ request người dùng, cache tăng tốc truy cập, full-text index phục vụ tìm kiếm, data warehouse phục vụ phân tích. Cùng một dữ liệu xuất hiện ở nhiều nơi nên cần được giữ đồng bộ. Cách tiếp cận "ngây thơ" là dual writes: mã ứng dụng ghi tường minh vào từng hệ thống khi dữ liệu thay đổi (ghi vào database, rồi cập nhật search index, rồi xóa cache). Dual writes có hai vấn đề nghiêm trọng [1]:

1. Race condition: hai client đồng thời cập nhật cùng một bản ghi. Client 1 đặt X = A, client 2 đặt X = B. Do thứ tự đan xen không may, database nhận phép ghi A rồi B (kết quả cuối cùng là B), trong khi search index nhận phép ghi B rồi A (kết quả cuối cùng là A), khiến hai hệ thống vĩnh viễn không nhất quán với nhau dù không có lỗi nào xảy ra. Nếu không có cơ chế phát hiện ghi đồng thời (như version vector), ta thậm chí không nhận biết được đã có các phép ghi đồng thời.
2. Một phép ghi thành công, phép ghi kia thất bại: ghi database thành công nhưng ghi search index thất bại, khiến hai hệ trở nên không nhất quán. Việc bảo đảm cả hai cùng thành công hoặc cùng thất bại chính là bài toán atomic commit, mà lời giải (2PC) rất tốn kém và nhiều vấn đề (phân tích ở Mục 2.4.4).

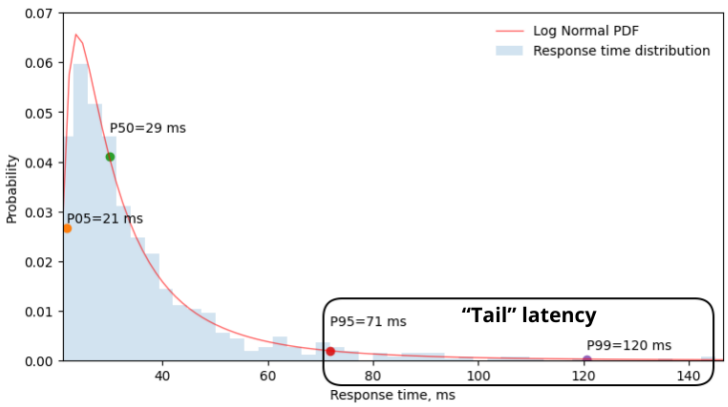
Vấn đề này được giải quyết bằng change data capture (CDC) và transactional outbox pattern, trình bày chi tiết ở Mục 3.8

2.6 Vấn đề vận hành

2.6.1 Tail latency và khuếch đại độ trễ đuôi

Như đã giới thiệu ở Mục 2.1.4, các percentile cao quyết định trải nghiệm thực tế của người dùng. Vấn đề trở nên nghiêm trọng hơn khi một request của người dùng cần gọi nhiều dịch vụ nền: chỉ cần một lời gọi chậm là toàn bộ request chậm, vì request phải chờ lời gọi chậm nhất hoàn tất. Về mặt xác suất, nếu một request cần N lời gọi nền song song và mỗi lời gọi có xác suất bị chậm là p, xác suất request bị chậm xấp xỉ $1 - (1 - p)^N$, tăng rất nhanh theo N. Hiện tượng này được gọi là tail latency amplification (khuếch đại độ trễ đuôi) [1].

Ví dụ minh họa: nếu một request của người dùng cần 10 lời gọi nền song song, và mỗi dịch vụ nền có 1% request chậm (tức p99 = ngưỡng chậm), thì xác suất request của người dùng bị chậm (có ít nhất một lời gọi chậm) vào khoảng $1-0{,}99^{10} \approx 9{,}6\%$, gần gấp mười lần xác suất chậm của một lời gọi đơn lẻ. Do đó, tối ưu hóa p99 của các dịch vụ nền quan trọng hơn nhiều so với tối ưu hóa giá trị trung bình. Hiện tượng khuếch đại độ trễ đuôi được minh họa trong Hình 2.11.



Hình 2.11: Khuếch đại độ trễ đuôi khi request phải chờ lời gọi chậm nhất

2.6.2 Cascading failure và retry storm

Một sự cố trong một thành phần có thể kéo theo cả một chuỗi sự cố lan truyền [1]:

- Cascading failure (sự cố lan truyền): một thành phần quá tải chậm lại → thành phần phụ thuộc vào nó phải chờ lâu hơn → hàng đợi đầy → các thành phần khác cũng bị chậm theo. Các nghiên cứu về loại sự cố này được tiến hành từ những năm 1990 trong bối cảnh hệ thống mạng lưới lớn [1].

- Retry storm (bão retry): khi một dịch vụ quá tải, các client nhận timeout bắt đầu thực hiện retry; retry làm tăng tải thêm, dịch vụ càng chậm, càng nhiều timeout, càng nhiều retry, hình thành một vòng phản hồi dương (feedback loop) đưa hệ thống xuống. DDIA cảnh báo cần tránh các vòng phản hồi như vậy và đề cập tới hiện tượng "hệ thống quá tải không thể tự phục hồi" (when an overloaded system won't recover): một số hệ thống cần giảm tải có chủ đích (load shedding) thay vì cố xử lý mọi request tới cùng.
- Cascading do rebalancing tự động: như phân tích ở Mục 2.5.6, sự kết hợp giữa rebalancing tự động và phát hiện lỗi tự động có thể biến một nút chậm thành chuỗi sự cố lan rộng.

2.6.3 Yếu tố con người trong vận hành

Một nghiên cứu về các dịch vụ Internet lớn cho thấy các thay đổi cấu hình do người vận hành thực hiện là nguyên nhân hàng đầu gây ra sự cố (outage), trong khi lỗi phần cứng chỉ đóng vai trò trong khoảng 10–25% trường hợp [1]. Điều này dẫn tới một quan điểm quan trọng: "lỗi con người" (human error) không phải là nguyên nhân gốc của sự cố, mà là triệu chứng của một vấn đề trong hệ thống kỹ thuật–xã hội (sociotechnical system) nơi con người đang cố gắng hoàn thành công việc của mình.

Các biện pháp kỹ thuật giúp giảm thiểu tác động của lỗi con người bao gồm: kiểm thử kỹ lưỡng (cả kiểm thử thủ công lẫn property-based testing), cơ chế roll-back nhanh cho các thay đổi cấu hình, triển khai dần dần (gradual rollout) mã mới, giám sát chi tiết và rõ ràng, công cụ observability để chẩn đoán sự cố production, và thiết kế giao diện "định hướng hành vi đúng và hạn chế hành vi sai". Ngoài ra, văn hóa blameless postmortem (không đổ lỗi cá nhân) đang ngày càng được áp dụng: sau mỗi sự cố, những người liên quan được khuyến khích chia sẻ chi tiết sự kiện mà không sợ bị trừng phạt, để cả tổ chức học hỏi và ngăn ngừa các sự cố tương tự trong tương lai [1].

2.6.4 Observability

Với hệ thống phân tán, việc chẩn đoán lỗi khó khăn hơn nhiều so với hệ thống đơn nút vì không biết lỗi nằm ở đâu trong chuỗi gọi. Kỹ thuật observability thu thập dữ liệu về quá trình thực thi của hệ thống và cho phép truy vấn dữ liệu đó theo cả hai hướng: các chỉ số tổng hợp (metrics) lẫn các sự kiện riêng lẻ (traces). Các công cụ distributed tracing như OpenTelemetry, Zipkin, Jaeger cho phép theo dõi client nào gọi server nào cho thao tác nào và mỗi lời gọi mất bao lâu. Đây là công cụ không thể thiếu để chẩn đoán các vấn đề về độ trễ và lỗi trong hệ thống phân tán [1].

2.6.5 Kiểm chứng độ tin cậy: fault injection

Các lỗi trong hệ thống phân tán thường tiềm ẩn và hiếm khi bộc lộ trong điều kiện hoạt động bình thường. Để chứng minh hệ thống thực sự có khả năng chịu lỗi, cần chủ động tiêm lỗi (fault injection) vào hệ thống [1]:

- Chaos engineering: chủ động gây ra sự cố cho các thành phần trong môi trường production (điển hình là Chaos Monkey của Netflix và các công cụ cùng họ) để kiểm tra khả năng tự phục hồi của hệ thống và tập huấn cho đội vận hành.
- Jepsen: công cụ của Kyle Kingsbury tiêm lỗi (partition, dừng nút, lệch đồng hồ, crash) vào hệ thống cơ sở dữ liệu rồi kiểm tra xem các bảo đảm nhất quán (như linearizability, snapshot isolation) có bị vi phạm hay không. Nhiều hệ thống nổi tiếng từng bị Jepsen phát hiện lỗi, ví dụ MongoDB, và ngay cả MySQL 8.0.34 cũng từng bị phát hiện vi phạm tính cô lập (năm 2023) [1].
- Deterministic simulation testing (DST): chạy toàn bộ hệ thống trong một môi trường mô phỏng kiểm soát được thời gian, mạng và đồng hồ, sao cho mọi lỗi đều có thể tái tạo; đây là cách FoundationDB và TigerBeetle xây dựng và kiểm chứng hệ thống của họ [1].

Các kỹ thuật này được phân tích chi tiết trong Mục 3.7

2.7 Tổng kết chương 2

Chương 2 đã trình bày một cách có hệ thống các vấn đề trong hệ thống phân tán, được phân loại thành năm nhóm:

1. Nền tảng lý thuyết: định lý CAP và mô hình PACELC cho thấy hệ thống phân tán luôn phải đánh đổi. Khi partition xảy ra, hệ thống đánh đổi giữa consistency và availability; khi mạng bình thường, đánh đổi giữa latency và consistency. Phần khó khăn nhất không nằm ở việc lựa chọn thuộc tính mà nằm ở ba vấn đề: phát hiện partition, xử lý trong thời gian partition và khôi phục nhất quán sau đó. Sự cố GitHub ngày 21/10/2018 (43 giây partition dẫn tới 24 giờ 11 phút khôi phục) là minh chứng rõ ràng.
2. Vấn đề môi trường: mạng không tin cậy, đồng hồ không đáng tin cậy và hiện tượng dừng tiến trình khiến một nút không bao giờ chắc chắn về trạng thái thực sự của các nút khác; đòi hỏi quy tắc đa số, lease, fencing token, và việc lựa chọn mô hình hệ thống (system model) phù hợp.
3. Vấn đề nhất quán: replication lag gây ra ba dị thường điển hình là không đọc được phép ghi của chính mình (read-your-writes), quan sát thời gian quay ngược (monotonic reads), và quan sát câu trả lời trước câu hỏi (consistent prefix). Các bảo đảm nhất quán tạo thành một dải liên tục, và sự vi phạm của

chúng rất khó phát hiện.

4. Vấn đề giao dịch và consensus: các mức cô lập yếu để lọt dirty read, read skew, lost update (ví dụ Flexcoin mất 600 bitcoin), write skew (ví dụ hai bác sĩ trực) và phantom; giao dịch phân tán đòi hỏi atomic commit, nhưng 2PC có điểm yếu chí mạng là giao dịch in doubt có thể treo vô hạn khi coordinator sập; consensus đắt đỏ và tinh tế, là nền tảng của bầu leader an toàn.
5. Vấn đề thiết kế và vận hành: ba họ sao chép có các rủi ro riêng (mất phép ghi, split brain, xung đột, LWW "nghiền" dữ liệu); phân mảnh gặp skew, hot spot và rebalancing có thể gây cascading failure; dual writes gây race condition và bài toán atomic commit; vận hành đối mặt với khuếch đại độ trễ đuôi, cascading failure, bão retry, lỗi con người (nguyên nhân hàng đầu gây sự cố), và nhu cầu kiểm chứng chủ động bằng fault injection.

Chương 3 sẽ giới thiệu các giải pháp tương ứng cho từng nhóm vấn đề trên.

CHƯƠNG 3. CÁC GIẢI PHÁP TRONG HỆ THỐNG PHÂN TÁN

Chương này trình bày các giải pháp tương ứng với từng nhóm vấn đề đã được phân tích ở Chương 2. Mỗi phần bao gồm nội dung lý thuyết, tiếp theo là nghiên cứu tình huống từ các hệ thống công nghiệp thực tế nhằm minh họa cách các lựa chọn thiết kế được áp dụng trong production.

3.1 Mô hình nhất quán: lựa chọn giữa các mức độ

3.1.1 Dải các mô hình nhất quán

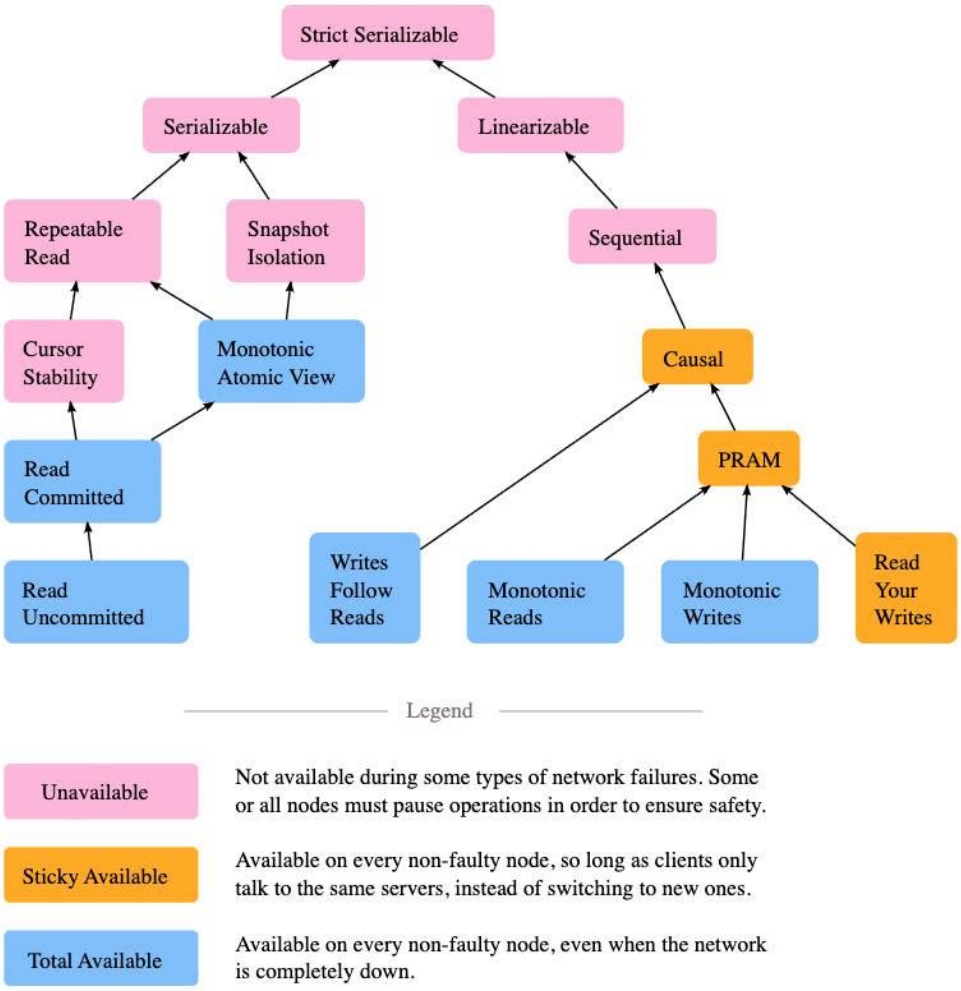
Như đã trình bày ở Mục 2.3.3, các mô hình nhất quán tạo thành một dải liên tục (spectrum) từ mạnh nhất tới yếu nhất, không phải một công tắc hai trạng thái "mạnh so với yếu". Dải này gồm các nấc sau, xếp theo thứ tự giảm dần về sức mạnh bảo đảm [3]:

- Linearizability (mạnh nhất): mỗi thao tác được coi như xảy ra tức thời tại một điểm nằm giữa lúc gửi request và lúc nhận response, và toàn bộ hệ thống thống nhất về thứ tự đó, kể cả thứ tự theo thời gian thực. Đây chính là thuộc tính C trong định lý CAP, và là bảo đảm đắt nhất trong danh sách [3].
- Sequential consistency: mọi nút vẫn thống nhất một thứ tự tổng thể duy nhất (total order) và thứ tự thao tác của từng tiến trình được giữ nguyên, nhưng thứ tự chung không bắt buộc khớp với thời gian thực. Cả hệ thống được phép chậm so với thực tế, miễn là mọi nút chậm theo cùng một cách và không nút nào quan sát được trạng thái "quay ngược thời gian". Khái niệm này bắt nguồn từ mô hình bộ nhớ chia sẻ của các hệ đa CPU. Trong thực tế, các hệ thống phân tán ít khi bán sequential consistency như một mức độ riêng biệt: khi đã chấp nhận chi phí chạy consensus để có một thứ tự chung, người ta thường hoàn thiện tới mức linearizability [3].
- Causal consistency: chỉ các thao tác có quan hệ nhân quả với nhau mới phải được quan sát theo đúng thứ tự đó ở mọi nơi; các thao tác độc lập (concurrent) có thể được quan sát theo bất kỳ thứ tự nào. Ví dụ kinh điển: một người dùng ẩn một tài khoản khỏi danh sách bạn bè rồi sau đó đăng ảnh liên quan tới tài khoản đó. Nếu thông tin về bức ảnh lan truyền tới một nút khác trước thông tin về lệnh ẩn, bức ảnh sẽ hiển thị một cách không hợp lý. Mahajan, Alvisi và Dahlin (2011) chứng minh một kết quả quan trọng: causal consistency là mô hình mạnh nhất mà một hệ thống vẫn có thể bảo toàn tính sẵn sàng khi mạng bị phân chia. Yêu cầu chặt hơn mức này một chút là chạm trực tiếp vào ranh giới của định lý CAP [3]; causal consistency vì thế là mức trần của các

hệ thống ưu tiên availability.

- Session / client-centric consistency: bảo đảm tính hợp lý trong phạm vi một phiên làm việc (session) của một người dùng, dựa trên bốn session guarantee xuất phát từ dự án Bayou của Xerox PARC (Terry và cộng sự, 1994) [3]:
  - Read-your-writes: phép đọc sau quan sát được phép ghi của chính người dùng đó (phân tích ở Mục 2.3.2).
  - Monotonic reads: phép đọc sau không được trả về dữ liệu cũ hơn phép đọc trước (phân tích ở Mục 2.3.2).
  - Monotonic writes: các phép ghi của cùng một người dùng phải được áp dụng theo đúng thứ tự người đó thực hiện.
  - Writes follow reads: nếu một phép đọc quan sát được một giá trị, các phép ghi sau đó của cùng người dùng phải được áp dụng dựa trên giá trị đã đọc, không ghi chồng lên quá khứ đã quan sát thấy.
- Bounded staleness: hệ thống khai báo giới hạn cho phép dữ liệu cũ, tối đa K phiên bản hoặc tối đa T đơn vị thời gian; vượt quá ngưỡng thì hệ thống buộc phải chờ cho tới khi dữ liệu được cập nhật. Đây là mức phù hợp cho các ứng dụng hiển thị số liệu không nhạy cảm về thời gian thực [3].
- Consistent prefix: các phép đọc được phép quan sát một thế giới cũ, nhưng luôn là một tiền tố hợp lệ của lịch sử các phép ghi, tức không bao giờ quan sát thấy một sự kiện xảy ra sau mà chưa thấy sự kiện xảy ra trước nó. Ví dụ tương đương với việc theo dõi một bộ phim trễ vài tập: chưa xem tới tập 8 thì chưa được xem, nhưng tập 8 không bao giờ được chiếu trước tập 5 [3].
- Eventual consistency (yếu nhất): nếu ngừng các phép ghi trong một khoảng thời gian đủ dài, các bản sao cuối cùng sẽ hội tụ về cùng một trạng thái; tuy nhiên mô hình này không xác định khoảng thời gian "đủ dài" và không cam kết bất kỳ hành vi nào trong giai đoạn chưa hội tụ. Quan sát bản mới rồi quay lại bản cũ, hay quan sát phép ghi thứ hai trước phép ghi thứ nhất, đều không vi phạm hợp đồng của mô hình [3].

Toàn bộ dải các mô hình nhất quán từ mạnh nhất tới yếu nhất được biểu diễn trong Hình 3.1.



Hình 3.1: Dải các mô hình nhất quán từ linearizability tới eventual consistency

3.1.2 Cosmos DB: năm mức nhất quán trên cùng một nền tảng

Nghiên cứu tình huống 6 – Microsoft Azure Cosmos DB [3]. Microsoft Azure Cosmos DB cung cấp một bộ điều khiển duy nhất gồm năm mức nhất quán, kết hợp cả hai góc nhìn: tính nhất quán nhìn từ phía dữ liệu và từ phía phiên làm việc:

```
strong → bounded staleness → session →
consistent prefix → eventual
```

Các mức này được hiểu như sau [3]:

- Strong: phép đọc luôn quan sát phép ghi mới nhất đã commit, tương đương linearizability đối với các phép đọc/ghi trong cùng một partition.
- Bounded staleness: cho phép dữ liệu cũ tối đa K phiên bản hoặc T khoảng thời gian (điều kiện nào tới trước được áp dụng trước).
- Session: gói bốn session guarantee trong phạm vi một phiên làm việc của client.

- Consistent prefix: các phép đọc luôn quan sát một tiền tố nhất quán của lịch sử phép ghi.
- Eventual: không có bất kỳ bảo đảm nào về thứ tự.

Mức mặc định cho mọi tài khoản mới là session, và theo công bố của Microsoft, phần lớn khách hàng giữ nguyên lựa chọn này. Điều này cho thấy mức "tầm thường" nằm giữa dải, không phải mức mạnh nhất cũng không phải mức rẻ nhất, lại là mức phục vụ phần lớn nhu cầu thực tế. Việc lựa chọn mức nhất quán vì thế không phải là bài toán "càng mạnh càng tốt", mà là lựa chọn mức vừa đủ cho từng loại thao tác nghiệp vụ [3].

3.1.3 Amazon S3 và bài học về bảo đảm nhất quán

Nghiên cứu tình huống 7 – Amazon S3 ngày 1 tháng 12 năm 2020 [3]. Amazon S3 ra đời năm 2006 với cơ chế nhất quán cuối cùng (eventual consistency), và hệ quả là cả một thế hệ hệ thống xử lý dữ liệu phải đối mặt với hiện tượng "ghi file xong nhưng liệt kê không thấy". Các hệ thống này khắc phục bằng cách dựng thêm một kho lưu trữ phụ trợ trên DynamoDB chỉ để ghi nhớ danh sách các file vừa được ghi, điển hình là S3Guard của Hadoop và consistent view của Amazon EMRFS. Nói cách khác, phải đưa thêm cả một cơ sở dữ liệu vào kiến trúc chỉ để bù đắp cho một bảo đảm còn thiếu.

Ngày 1 tháng 12 năm 2020, AWS thông báo S3 cung cấp strong read-after-write consistency cho toàn bộ các thao tác GET, PUT và LIST, áp dụng cho mọi object tại mọi region, không cần cấu hình, không tính thêm chi phí và không đánh đổi hiệu năng hay tính sẵn sàng [3]. Thuật ngữ "strong" trong bối cảnh này là read-after-write trong phạm vi từng object, hẹp hơn chữ C trong định lý CAP hay trong bảng lựa chọn của Cosmos DB; tuy nhiên đối với các hệ thống xử lý dữ liệu nói trên, đây chính là bảo đảm mà họ đã thiếu suốt nhiều năm. Toàn bộ các giải pháp thay thế (workaround) nói trên trở nên thừa thãi chỉ sau một đêm.

Chi phí của một bảo đảm nhất quán phụ thuộc rất nhiều vào cách thức triển khai. S3, với kiến trúc bucket ổn định và hệ thống metadata tập trung mạnh, đạt được read-after-write consistency với chi phí chấp nhận được, trong khi nhiều hệ thống NoSQL khác vẫn giữ eventual consistency làm mặc định vì mô hình truy cập (access pattern) của chúng khác biệt [3].

3.1.4 DynamoDB: chi phí của nhất quán được biểu thị trực tiếp bằng giá

Nghiên cứu tình huống 8 – Amazon DynamoDB [2], [3]. Ngược lại với S3, DynamoDB vẫn giữ eventually consistent read làm mặc định, và bảng giá của dịch

vụ biểu thị trực tiếp sự đánh đổi này thành chi phí tài chính: một strongly consistent read cho một item tối đa 4 KB tiêu tốn 1 read request unit, trong khi eventually consistent read chỉ tiêu tốn nửa đơn vị [2], [3]. Cùng thuộc một tập đoàn (Amazon), hai sản phẩm (S3 và DynamoDB) lại có hai lựa chọn mặc định đối lập nhau. Điều này không phải vì sản phẩm nào "tiến bộ hơn", mà vì mô hình truy cập của chúng khác nhau [3].

Về kiến trúc, DynamoDB sử dụng leader-based replication chạy giao thức Multi-Paxos: mọi phép ghi phải đi qua leader và chỉ được xác nhận sau khi một quorum đã ghi vào write-ahead log; strongly consistent read chỉ được phục vụ bởi leader, còn follower chỉ phục vụ eventually consistent read. Do đó, mức nhất quán của DynamoDB được quyết định ở cấp độ từng request bởi khách hàng, không phải do hệ quản trị lựa chọn sẵn [2].

3.1.5 Nguyên tắc lựa chọn mức nhất quán

Từ các phân tích trên, có thể tổng kết một nguyên tắc thực dụng: đánh giá từng thao tác nghiệp vụ riêng lẻ, không lựa chọn một mức nhất quán duy nhất cho toàn hệ thống [3]:

- Các thao tác hiển thị số lượt xem, lượt thích, dòng bài đăng (feed): consistent prefix là đủ, dữ liệu chậm vài giây không gây hệ quả nghiêm trọng, miễn là không để lời trả lời xuất hiện trước câu hỏi gốc.
- Các thao tác hiển thị thông tin cá nhân, giỏ hàng, đơn hàng vừa đặt: thuộc về session guarantee, vì người dùng chỉ cần quan sát được chính các thao tác của bản thân mình.
- Các thao tác liên quan tới tồn kho, hạn mức, số dư tài khoản ngay trước lệnh rút tiền: mới cần mức mạnh nhất, và cũng chỉ trong phạm vi các thao tác này.

Ép toàn bộ hệ thống vận hành theo một mức nhất quán duy nhất là cách chắc chắn nhất để vừa trả chi phí quá cao, vừa sai lệch đúng tại những chỗ không được phép sai lệch [3].

3.1.6 Triển khai session guarantee bằng token thay cho sticky routing

Phương pháp phổ biến để giữ bảo đảm read-your-writes là sticky routing, tức ghim một người dùng cố định vào một replica. Phương pháp này có nhiều nhược điểm [3]:

- Khi người dùng chuyển giữa các mạng truy cập khác nhau (ví dụ từ Wi-Fi sang 4G), khi thao tác trên laptop rồi tiếp tục trên điện thoại, hoặc khi replica được khởi động lại, phiên làm việc bị ngắt và bảo đảm theo đó bị mất, đúng vào những thời điểm khó tái tạo sự cố nhất.

- Sticky routing gây mất cân bằng tải: replica nào đang phục vụ một nhóm người dùng hoạt động tích cực sẽ trở nên quá tải hơn hẳn các replica khác.

Phương pháp phù hợp hơn là sử dụng token: sau mỗi phép ghi, hệ thống trả về một logical timestamp; client mang token này theo trong các request tiếp theo; replica nào chưa bắt kịp mốc thời gian đó phải chờ hoặc chuyển request sang replica khác. Bảo đảm lúc này gắn với dữ liệu thay vì gắn với đường mạng. Đây là phiên bản thu gọn của causal consistency: thay vì xây dựng toàn bộ đồ thị nhân quả toàn cục, hệ thống chỉ theo dõi chuỗi nhân quả của riêng một client, rẻ hơn nhiều bậc và vừa đủ để ngăn chặn các dị thường mà người dùng có thể quan sát được [3]. MongoDB áp dụng cách này với tính năng *causally consistent sessions* từ bản 3.6; Cosmos DB gọi tương tự là *session token*. Một điều kiện quan trọng: token phải được truyền xuyên suốt toàn bộ chuỗi xử lý, qua các gateway, các dịch vụ trung gian và cả ứng dụng di động; chỉ cần một thành phần làm rơi token là toàn bộ cơ chế lặng lẽ trở lại mức yếu nhất [3].

3.2 Giao dịch: các mức cô lập và ba cách triển khai serializable

3.2.1 Mức cô lập read committed

Mức cô lập yếu nhất còn được xem là có thể chấp nhận trong thực tế là read committed, với hai bảo đảm cơ bản [1]:

- Khi đọc, một giao dịch chỉ quan sát được dữ liệu đã commit, qua đó ngăn chặn dị thường dirty read.
- Khi ghi, một giao dịch chỉ ghi đè dữ liệu đã commit, qua đó ngăn chặn dị thường dirty write.

Tuy nhiên read committed vẫn cho phép read skew: trong cùng một giao dịch, các phép đọc có thể quan sát các trạng thái khác nhau của cơ sở dữ liệu tại các thời điểm khác nhau. Đối với các giao dịch kéo dài, ví dụ sao lưu toàn bộ dữ liệu hoặc tạo báo cáo tổng hợp, read skew có thể tạo ra kết quả vô nghĩa [1].

3.2.2 Snapshot isolation và MVCC

Snapshot isolation (SI) bảo đảm mọi phép đọc trong một giao dịch quan sát một snapshot nhất quán tại một thời điểm nhất định, tức cơ sở dữ liệu được nhìn như một khối tĩnh trong suốt thời gian thực hiện giao dịch. Cơ chế triển khai phổ biến là multiversion concurrency control (MVCC): hệ thống duy trì nhiều phiên bản của mỗi bản ghi, và mỗi giao dịch đọc phiên bản phù hợp với thời điểm snapshot của nó. SI ngăn chặn được dirty read, read skew và (tùy triển khai) lost update, nhưng không ngăn chặn được write skew, đúng như ví dụ hai bác sĩ trực đã phân tích ở Mục 2.4.2 [1].

Về mặt thực thi, một giao dịch trong MVCC quan sát các đối tượng phù hợp với thời điểm giao dịch bắt đầu (hoặc thời điểm snapshot được cấp). Cơ sở dữ liệu giữ các phiên bản cũ của bản ghi để phục vụ các giao dịch có snapshot sớm hơn. Khi hai giao dịch ghi đè cùng một bản ghi, cơ chế phát hiện xung đột ghi sẽ hủy bỏ (abort) một trong hai; việc ghi đè (lost update) thường được xử lý bằng cách yêu cầu ứng dụng đọc rồi ghi lại trong cùng một giao dịch, hoặc sử dụng khóa tường minh (SELECT ... FOR UPDATE) [1].

3.2.3 Ba cách triển khai mức serializable

Như đã trình bày ở Mục 2.4.3, chỉ có mức serializable mới ngăn chặn được đầy đủ các dị thường, trong đó có write skew. Ba cách triển khai được trình bày chi tiết dưới đây [1]:

a) Thực thi tuần tự thực sự (actual serial execution). Các giao dịch được thực thi lần lượt trên một lõi CPU duy nhất (hoặc mỗi shard một lõi). Điều kiện tiên quyết để triển khai hiệu quả: mỗi giao dịch phải thực thi rất nhanh (thường thông qua stored procedures để tránh vòng khứ hồi qua mạng trong lúc giữ tài nguyên), thông lượng phải đủ thấp để xử lý trên một lõi hoặc phải được phân mảnh, và toàn bộ dữ liệu phải nằm trong bộ nhớ. Các hệ thống tiêu biểu gồm VoltDB và H-Store. Ưu điểm lớn nhất là sự đơn giản: không còn bài toán xung đột khóa; nhược điểm là khả năng mở rộng bị chặn bởi giới hạn một lõi (hoặc một shard) [1].

b) Two-phase locking (2PL). Cách triển khai serializability kinh điển trong nhiều thập kỷ. Để ghi một đối tượng, giao dịch phải có shared lock khi đọc và exclusive lock khi ghi; các khóa được giữ tới khi giao dịch kết thúc. Vì hai giao dịch không thể đồng thời giữ các khóa xung đột, thứ tự thực thi của chúng tương đương với một thứ tự tuần tự nào đó. Nhược điểm: chi phí khóa cao dưới tải lớn, và hiện tượng deadlock đòi hỏi cơ chế phát hiện và hủy bỏ giao dịch [1].

c) Serializable snapshot isolation (SSI). Thuật toán tương đối mới (Cahill và Fekete, 2008), được triển khai trong PostgreSQL, CockroachDB và FoundationDB. SSI tiếp cận theo hướng tối ưu (optimistic): giao dịch thực thi không bị chặn như trong snapshot isolation; tuy nhiên khi chuẩn bị commit, hệ thống kiểm tra xem thực thi có thực sự serializable hay không, và nếu không thì hủy bỏ một trong các giao dịch liên quan. SSI phát hiện write skew bằng cách theo dõi các "tiền đề" (premises) mà giao dịch đã dựa vào và các truy cập có khả năng tạo ra vòng phụ thuộc (dependency cycle) [1].

Nghiên cứu tình huống 9 – CockroachDB sử dụng SSI [1]. CockroachDB, một cơ sở dữ liệu SQL phân tán về mặt địa lý, triển khai mức cô lập serializable bằng SSI kết hợp với Raft cho cơ chế sao chép. Hệ thống cung cấp mức cô lập mạnh

nhất trên một cơ sở dữ liệu vừa phân mảnh vừa nhân bản trên nhiều khu vực địa lý, đổi lại phải hủy bỏ một phần nhỏ các giao dịch khi phát hiện xung đột.

3.2.4 Tổng kết việc lựa chọn mức cô lập

Bảng 8-1 của DDIA (xem Mục 2.4.2) là bảng tham chiếu ngắn gọn nhất để lựa chọn mức cô lập [1]:

- Nếu ứng dụng không chấp nhận bất kỳ dị thường nào, sử dụng mức serializable.
- Nếu chỉ cần tránh dirty read và đọc snapshot nhất quán, snapshot isolation là điểm cân bằng được sử dụng phổ biến nhất.
- Read committed là mức tối thiểu mà mọi hệ thống nên cung cấp.

3.3 Giao dịch phân tán và NewSQL

3.3.1 Phạm vi áp dụng giao dịch phân tán

Như đã phân tích ở Mục 2.4.4, giao dịch 2PC vượt qua ranh giới của nhiều công nghệ lưu trữ khác nhau (XA transactions) là một lựa chọn có vấn đề: rất nhạy cảm với sự cố của coordinator, có thể bị treo không xác định khi coordinator sập, và tương tác kém với các cơ chế kiểm soát đồng thời. DDIA đưa ra hai nhận định quan trọng [1]:

- Khi các nút tham gia đều chạy cùng một phần mềm cơ sở dữ liệu, giao dịch phân tán nội bộ có thể hoạt động tốt, ví dụ giao dịch trải qua nhiều shard của cùng một cơ sở dữ liệu.
- Khi giao dịch vượt qua ranh giới giữa các công nghệ lưu trữ khác nhau (cơ sở dữ liệu và message broker, hoặc hai loại cơ sở dữ liệu khác nhau), 2PC không còn là lựa chọn phù hợp; giải pháp thay thế là sử dụng idempotence để đạt exactly-once semantics mà không cần atomic commit xuyên công nghệ [1].

Ý tưởng của idempotence được triển khai như sau [1]:

- Mỗi message được gán một message ID duy nhất.
- Khi xử lý message, ứng dụng ghi message ID cùng với dữ liệu kết quả trong một transaction nội bộ của cơ sở dữ liệu.
- Khi nhận message, ứng dụng kiểm tra ID đã tồn tại trong cơ sở dữ liệu hay chưa; nếu đã tồn tại thì bỏ qua.

Bởi vì việc ghi message ID và dữ liệu nằm trong cùng một transaction nội bộ, atomicity được bảo đảm mà không cần phối hợp với broker. Quá trình xử lý message trở thành idempotent: việc thực hiện lại (retry) là an toàn, không gây trùng lặp

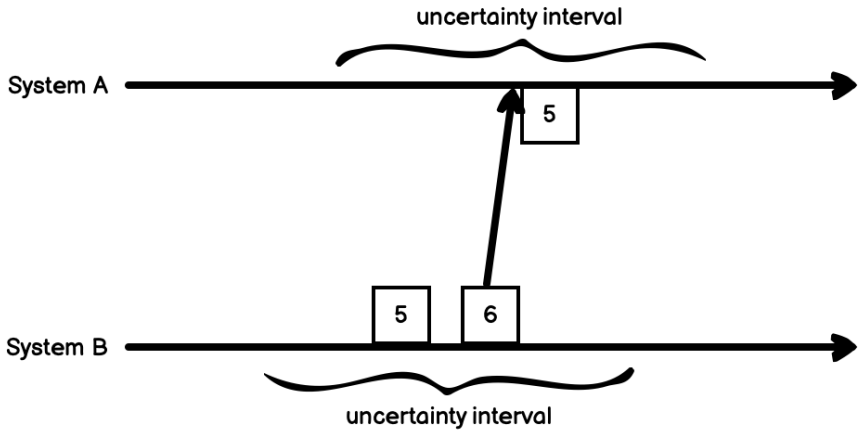
tác dụng phụ. Kafka Streams áp dụng cách tiếp cận tương tự để đạt exactly-once semantics [1]. Chi tiết triển khai qua outbox pattern được trình bày ở Mục 3.8

3.3.2 Spanner và TrueTime: nhất quán mạnh trên quy mô toàn cầu

Nghiên cứu tình huống 10 – Google Cloud Spanner [1], [2]. Google Cloud Spanner là cơ sở dữ liệu phân tán toàn cầu với tham vọng mang giao dịch serializable tới quy mô toàn cầu với độ trễ chấp nhận được. Chìa khóa kỹ thuật nằm ở hệ thống TrueTime [1], [2]:

- Spanner sử dụng đồng hồ nguyên tử (atomic clocks) và GPS trong mỗi trung tâm dữ liệu, kết hợp để ước lượng thời gian thực với sai số đã biết (trong khoảng vài chục mili-giây). TrueTime không trả về một mốc thời gian điểm, mà trả về một khoảng [earliest, latest] mà thời điểm thực chắc chắn nằm trong đó.
- Nhờ biết được sai số của đồng hồ, Spanner thực hiện commit wait: chờ cho khoảng không chắc chắn trôi qua trước khi commit, nhờ đó bảo đảm thứ tự commit khớp với thời gian thực, cho phép xây dựng external consistency (linearizability áp dụng cho giao dịch).
- Cơ chế sao chép dựa trên Paxos chạy giữa các replica trong cùng một zone; giao dịch xuyên shard và xuyên zone được xử lý bằng 2PC.

Cơ chế TrueTime trả về một khoảng thời gian chứa thời điểm thực được minh họa trong Hình 3.2.



Hình 3.2: TrueTime trả về khoảng [earliest, latest] chứa thời điểm thực

Soi chiếu qua mô hình PACELC, Spanner thuộc nhóm PC/EC, ưu tiên nhất quán

trong cả hai tình huống, trả giá bằng độ trễ và đôi khi từ chối phục vụ trong các tình huống hiếm [2]. Về mặt kỹ thuật Spanner là một hệ thống CP, nhưng vì mức sẵn sàng thực tế đạt tới năm số chín, đối với người dùng nó hoạt động như một hệ thống CA. Phần thiệt hại do partition gây ra nhỏ hơn nhiều so với tổng các nguyên nhân sự cố khác. Khi xác suất xảy ra một nhánh lỗi đủ nhỏ, sự đánh đổi về mặt lý thuyết vẫn tồn tại nguyên vẹn, nhưng không còn chi phối trải nghiệm thực tế của người dùng [2].

3.3.3 FoundationDB: kiến trúc tách rời và kiểm chứng bằng mô phỏng xác định

Nghiên cứu tình huống 11 – FoundationDB [1]. FoundationDB (được Apple mua lại, sau đó mở mã nguồn trở lại vào năm 2018) là một kho lưu trữ key-value phân tán nổi tiếng với giao dịch ACID trên quy mô toàn cầu, xây dựng theo kiến trúc tách rời (unbundled): tầng xử lý giao dịch, tầng lưu trữ (storage servers) và tầng phối hợp hoạt động như các thành phần độc lập. Hai điểm kỹ thuật đáng chú ý [1]:

- Deterministic simulation testing (DST): toàn bộ hệ thống được chạy trong một trình mô phỏng kiểm soát được thời gian, mạng và đồng hồ, sao cho mọi lỗi phát hiện được đều có thể tái tạo (reproduce). Đây là lý do FoundationDB được đánh giá là một trong những hệ thống có độ tin cậy thiết kế cao nhất (phân tích chi tiết ở Mục 3.7).
- FoundationDB sử dụng SSI cho serializability, Raft cho cơ chế phối hợp, và cho phép xây dựng các lớp (layer) dữ liệu khác nhau phía trên, tức "lưu trữ dữ liệu một cách giao dịch" để phục vụ nhiều mô hình dữ liệu.

3.4 Replication: ba họ thuật toán và cách xử lý xung đột

3.4.1 Single-leader: các biến thể đồng bộ và failover an toàn

Như đã trình bày ở Mục 2.5.1, single-leader replication là cấu hình phổ biến nhất. Các lựa chọn về mức độ đồng bộ của việc sao chép và các yêu cầu để failover an toàn được phân tích chi tiết dưới đây [1]:

Các biến thể đồng bộ [1]:

- Sao chép đồng bộ hoàn toàn (all synchronous): không thực tế trong thực hành, vì một nút gặp sự cố sẽ khiến toàn bộ hệ thống ngừng nhận phép ghi.
- Semisynchronous replication: một follower được cấu hình đồng bộ, các follower còn lại bất đồng bộ; khi follower đồng bộ gặp sự cố, một follower bất đồng bộ được nâng lên làm follower đồng bộ. Cấu hình này bảo đảm dữ liệu nằm trên ít nhất hai nút (leader và một follower đồng bộ).

- Quorum đa số: một đa số các replica (ví dụ ba trong năm) được cập nhật đồng bộ, số còn lại bất đồng bộ.
- Sao chép hoàn toàn bất đồng bộ: leader tiếp tục nhận phép ghi dù mọi follower đều tụt lại; khi leader gặp sự cố, các phép ghi chưa kịp sao chép bị mất dù đã được xác nhận thành công cho client.

Yêu cầu để failover an toàn [1]:

- Phát hiện leader chết (bằng timeout hoặc heartbeat) và bầu leader mới một cách an toàn; đây chính là bài toán consensus, được giải quyết bằng các thuật toán như Raft (sử dụng trong CockroachDB, TiDB, etcd, RabbitMQ quorum queues; Kafka sử dụng giao thức tương tự).
- Tránh mất phép ghi: khi follower được thăng cấp chưa nhận được các phép ghi cuối cùng của leader cũ, các phép ghi đó bị mất. Kafka giảm thiểu vấn đề bằng cách chỉ bầu leader mới trong số các follower đã bắt kịp (caught-up replicas) nhất, đồng thời chấp nhận một cửa sổ mất dữ liệu nhỏ khi hệ thống quá tải.
- Tránh split brain: sử dụng fencing token (xem Mục 3.5) để bảo đảm chỉ một leader thực sự có quyền ghi dữ liệu.

3.4.2 Multi-leader: phân tán địa lý và giải quyết xung đột

Trường hợp sử dụng multi-leader replication [1]:

- Phân tán về mặt địa lý (geo-distributed): mỗi region có một leader; phép ghi được xử lý ngay trong region rồi được sao chép bất đồng bộ sang các region khác. Kiến trúc này che giấu độ trễ liên vùng, chịu được sự cố vùng, và chịu đựng được chất lượng mạng liên vùng kém. Nhược điểm quan trọng: tính nhất quán yếu hơn nhiều, không thể bảo đảm các ràng buộc như "tài khoản không âm" hay "username không trùng" [1].
- Cộng tác trong phòng / soạn thảo cộng tác: nhiều thiết bị, nhiều văn phòng, có thời điểm ngoại tuyến (local-first software).

Các phương pháp giải quyết xung đột [1]:

- LWW – last write wins: phương pháp đơn giản nhất, nhưng cần thận trọng. Khái niệm "ai ghi sau" phụ thuộc vào timestamp, mà timestamp không đáng tin cậy khi đồng hồ lệch nhau (Mục 2.2.2); LWW "nghiền" dữ liệu xung đột mà không phát sinh bất kỳ thông báo nào [2]. Để phân định thứ tự một cách xác định, nên gắn kèm một unique ID vào timestamp. LWW chỉ nên dùng khi ứng dụng chấp nhận mất đi một trong các giá trị xung đột [1].
- Lưu trữ siblings và giải quyết thủ công: hệ thống lưu toàn bộ các giá trị được

ghi đồng thời (siblings) và trả về tất cả cho ứng dụng quyết định; đây là cách tiếp cận của CouchDB. Nhược điểm: API của cơ sở dữ liệu thay đổi, và việc yêu cầu người dùng tự hợp nhất các giá trị xung đột tốn công sức [1].

- Giải quyết tự động, strong eventual consistency (SEC): hợp nhất các phép ghi đồng thời về một trạng thái hội tụ, độc lập với thứ tự các phép ghi tới nơi. Hai họ thuật toán chính là CRDT và OT [1]:

CRDT (conflict-free replicated datatypes). Mỗi phần tử dữ liệu có một ID duy nhất, bất biến; thao tác chèn tham chiếu tới ID của phần tử liền trước; các phép chèn đồng thời tại cùng một vị trí được sắp xếp một cách xác định bằng ID. Các replica hội tụ về cùng một trạng thái mà không cần chuyển đổi thao tác (transform). CRDT hỗ trợ các kiểu dữ liệu: text (mỗi ký tự mang một ID), tập hợp (theo dõi cả các thao tác xóa dưới dạng tombstone để tránh lỗi giỏ hàng Amazon, xem Mục 2.5.4), counter (cộng dồn đúng các lần tăng/giảm), key-value map (kết hợp các thuật toán trên theo từng khóa) [1]. CRDT được sử dụng trong Redis Enterprise, Riak, Azure Cosmos DB; các thư viện đồng bộ JSON như Automerge và Yjs cũng dựa trên họ thuật toán này [1].

OT (operational transformation). Hệ thống ghi lại vị trí (index) của thao tác chèn/xóa; khi áp dụng một thao tác lên một trạng thái đã chứa các thao tác đồng thời khác, vị trí được biến đổi (transform) để bù trừ. OT chủ yếu được sử dụng cho soạn thảo cộng tác thời gian thực [1].

Nghiên cứu tình huống 12 – Google Docs [1]. Google Docs sử dụng OT để cho phép nhiều người cùng soạn thảo một tài liệu trong thời gian thực: mỗi thao tác chèn/xóa ký tự mang theo vị trí (index); máy chủ và các client biến đổi vị trí khi gộp các thao tác đồng thời, nhờ đó mọi người tham gia quan sát được cùng một kết quả hội tụ.

Một giới hạn cần ghi nhận của giải quyết xung đột: không thể bảo đảm các ràng buộc toàn vẹn như "danh sách không quá 5 phần tử" khi nhiều người đồng thời thêm phần tử, hệ thống buộc phải chấp nhận vượt giới hạn. Tuy nhiên, đối với các ứng dụng cộng tác hoặc hoạt động ngoại tuyến, giải quyết xung đột là điều tất yếu, và việc tự động hóa nó thường là lựa chọn tốt nhất [1].

3.4.3 Leaderless: quorum reads/writes

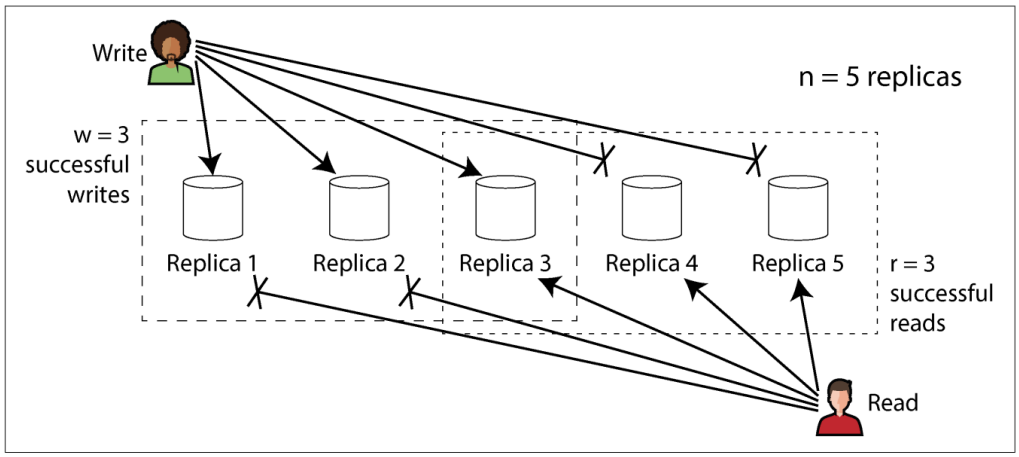
Leaderless replication (client ghi và đọc trực tiếp tới nhiều nút) mang tinh thần của bài báo Dynamo (2007) của Amazon, được hiện thực hóa trong DynamoDB, Cassandra, Riak, Voldemort [1], [2].

Cơ chế quorum [1]: với n replica, một phép ghi phải thành công trên w nút và

một phép đọc phải đọc từ r nút; nếu $w + r > n$, thì bất kỳ phép đọc quorum nào cũng gặp ít nhất một nút có dữ liệu mới nhất, nhờ đó bảo đảm phép đọc quan sát được phép ghi đã được xác nhận (nếu đọc sau ghi). Cấu hình phổ biến là n = 3, w = 2, r = 2 (tức w = r = đa số). Tuy nhiên cơ chế quorum có các giới hạn sau [1]:

- Nếu sử dụng sloppy quorum (quorum lỏng, phép ghi được gửi tới các nút khác khi quorum chính không đủ nút) kết hợp với hint handoff, bảo đảm nhất quán bị giảm xuống.
- Quorum không ngăn chặn được write skew và các ràng buộc liên quan tới nhiều đối tượng.
- Khi hai phép ghi đồng thời cùng một khóa, cần version vector để phát hiện và hợp nhất; LWW là lựa chọn đơn giản nhưng gây mất dữ liệu.

Điều kiện $w + r > n$ bảo đảm phép đọc quorum gặp ít nhất một nút có dữ liệu mới nhất, được minh họa trong Hình 3.3.



Hình 3.3: Đọc quorum gặp ít nhất một nút có dữ liệu mới nhất khi $w + r > n$

Nghiên cứu tình huống 13 – Cassandra [2]. Cassandra (kế thừa ý tưởng Dynamo) sử dụng kiến trúc leaderless với quorum cấu hình được. Soi chiếu qua PACELC, Cassandra thuộc nhóm PA/EL, ưu tiên tính sẵn sàng và độ trễ. Hệ thống nổi tiếng với việc vận hành trên quy mô lớn (được sử dụng trong các hệ thống của LinkedIn và Apple) và khả năng chịu đựng phân chia mạng rất tốt.

3.5 Consensus và các dịch vụ điều phối

3.5.1 Consensus tương đương với total order broadcast

Như đã trình bày ở Mục 2.4.5, các thuật toán consensus (Paxos, Raft) về bản chất tương đương với total order broadcast (phát sóng tổng thứ tự): mọi nút nhận được cùng một chuỗi thông điệp theo cùng một thứ tự. Từ tính chất này có thể xây dựng nên các thành phần sau [1]:

- Máy trạng thái nhân bản (replicated state machine): mọi replica áp dụng cùng một log theo cùng một thứ tự và đạt cùng một trạng thái.
- Bầu leader an toàn: chỉ một nút được bầu, không xảy ra split brain.
- Lưu trữ tuyến tính hóa (linearizable storage): các thao tác so sánh-và-gán (compare-and-set) với tính chất linearizability.
- Các dịch vụ điều phối (coordination service): như ZooKeeper và etcd.

Raft được sử dụng trong CockroachDB, TiDB, etcd, RabbitMQ quorum queues, YugabyteDB; Kafka sử dụng giao thức tương tự [1].

3.5.2 Các dịch vụ điều phối: ZooKeeper, etcd và Consul

Các dịch vụ điều phối được mô hình hóa theo Google Chubby lock service. Bề ngoài chúng giống một kho lưu trữ key-value, nhưng không được thiết kế cho khối lượng ghi cao hay lưu trữ dữ liệu mục đích chung; mục đích của chúng là điều phối giữa các nút của một hệ thống phân tán khác. Ví dụ: Kubernetes dựa trên etcd; Spark và Flink (chế độ high availability) dựa trên ZooKeeper [1]. Các tính năng chính [1]:

- Khóa và lease: triển khai bằng thao tác CAS (compare-and-set) nguyên tử chịu lỗi; khi nhiều nút cùng tranh một lease, chỉ một nút giành được.
- Fencing token: mỗi mục log được gán một ID tăng đơn điệu (zxid và cversion trong ZooKeeper, revision number trong etcd), được sử dụng làm fencing token khi tài nguyên được bảo vệ bởi lease.
- Phát hiện lỗi: client duy trì một session dài hạn và gửi heartbeat; nếu không nhận được heartbeat vượt quá lease timeout, dịch vụ điều phối coi client đã chết và giải phóng lease (ZooKeeper gọi là ephemeral nodes).
- Thông báo thay đổi: client đăng ký nhận thông báo khi một khóa thay đổi, nhờ đó tiết kiệm chi phí polling liên tục.

Các trường hợp sử dụng phổ biến [1]:

- Quản lý cấu hình: lưu cấu hình dạng key-value và đăng ký nhận thông báo thay đổi.
- Phân bổ công việc: bầu leader cho các cơ sở dữ liệu single-leader hoặc job scheduler; quyết định shard nào nằm trên nút nào; khi nút vào/ra khỏi hệ thống, thực hiện rebalance. Các thao tác này được thực hiện bằng atomic operations kết hợp ephemeral nodes và notifications.
- Service discovery: tìm địa chỉ IP để kết nối tới một dịch vụ. Trong trường

hợp này, service discovery thường không cần consensus/linearizability; yếu tố quan trọng hơn là tốc độ và tính sẵn sàng cao, do đó nên sử dụng bộ nhớ đệm (ví dụ DNS) thay vì gọi consensus cho mỗi lần tra cứu [1].

Một ưu điểm kiến trúc quan trọng: dịch vụ điều phối chạy trên một tập nút cố định nhỏ (thường ba hoặc năm nút), bất kể hệ thống phân tán phía trên có hàng nghìn nút hay không. Chạy consensus trên hàng nghìn nút là cực kỳ kém hiệu quả; việc giao việc thực hiện consensus cho một nhóm nhỏ nút chuyên biệt là một mẫu thiết kế chuẩn [1].

3.5.3 Lease và fencing token

Trở lại vấn đề được nêu ở Mục 2.2.5: một nút giữ lock nhưng bị GC dừng 30 giây, lease hết hạn, nút khác giành được lock; nút cũ tỉnh dậy và tưởng rằng mình vẫn đang giữ lock. Giải pháp chuẩn là fencing token: mỗi lần lock được trao, hệ thống cấp một số nguyên tăng dần. Nút giữ lock phải đính kèm token trong mọi yêu cầu ghi; máy chủ tài nguyên lưu giữ token lớn nhất đã nhận được và từ chối các yêu cầu có token nhỏ hơn. Nhờ đó, dù nút cũ có tỉnh dậy và tiếp tục ghi, token của nó đã cũ nên bị từ chối [1]. Sự kết hợp giữa lease (giới hạn thời gian giữ lock) và fencing token (giới hạn quyền ghi theo token) là phòng thủ chuẩn chống lại hiện tượng dừng tiến trình (process pause).

3.6 Phân mảnh và định tuyến

3.6.1 Lựa chọn khóa phân mảnh và tránh hot spot

Như đã trình bày ở Mục 2.5.6, có hai cách phân mảnh cơ bản. Các biến thể và vấn đề liên quan được phân tích chi tiết như sau [1]:

- Key range sharding (phân mảnh theo dải khóa): chia dải khóa thành các khoảng liên tục. Truy vấn theo dải (range scan) hiệu quả, nhưng dễ tạo hot shard khi khóa là timestamp, vì mọi phép ghi đổ dồn về một shard. Khắc phục: thêm tiền tố (ví dụ sensor ID) vào trước timestamp để phân tán phép ghi; đánh đổi bằng việc mất khả năng truy vấn dải gộp [1].
- Hash sharding (phân mảnh theo băm khóa): băm khóa để phân phối đều dữ liệu. Hai biến thể chính [1]:
  - Số lượng shard cố định (fixed number of shards): tạo nhiều shard hơn số nút ngay từ đầu (ví dụ 1.000 shard trên 10 nút), mỗi nút giữ nhiều shard; khi thêm nút chỉ cần di chuyển nguyên cả shard. Được sử dụng trong Citus, Riak, Elasticsearch, Couchbase. Hạn chế: không thể có nhiều nút hơn số shard, và việc thay đổi số shard đòi hỏi resharding tốn kém.
  - Băm theo dải giá trị hash (hash range): mỗi shard giữ một dải giá trị hash;

shard có thể tách/ghép linh hoạt theo khối lượng dữ liệu. Cassandra và ScyllaDB chia không gian giá trị hash thành nhiều dải với ranh giới ngẫu nhiên (Cassandra mặc định 16 dải mỗi nút, ScyllaDB 256 dải mỗi nút); mỗi nút giữ nhiều dải để bù trừ sự chênh lệch kích thước giữa các dải, và khi thêm nút, một phần dải được chuyển sang nút mới [1].

- Consistent hashing: băm ánh xạ khóa tới shard với hai tính chất mong muốn: (1) số khóa mỗi shard gần như đều nhau, (2) khi số shard thay đổi, càng ít khóa phải di chuyển càng tốt. Thuật ngữ cần được làm rõ: chữ "consistent" trong consistent hashing không liên quan tới replica consistency hay ACID consistency; nó chỉ mô tả xu hướng các khóa "ở yên" tại shard cũ khi số shard thay đổi [1]. Các thuật toán thay thế gồm rendezvous hashing (highest random weight) và jump consistent hashing [1].

Vấn đề skew và hot spot [1]: dù băm phân phối đều các khóa, tải thực tế có thể vẫn lệch. Một khóa duy nhất "nóng" (hot key), ví dụ bài đăng của một người nổi tiếng với hàng triệu người theo dõi, có thể gây một cơn bão đọc/ghi vào một shard. Giải pháp ở tầng ứng dụng: thêm một vài ký tự ngẫu nhiên vào khóa để phân tán phép ghi ra nhiều shard, nhưng phép đọc phải truy vấn và gộp kết quả từ tất cả các khóa đã tách; kỹ thuật này chỉ có ý nghĩa với một số ít khóa nóng và đòi hỏi cơ chế theo dõi khóa nào đang được tách. Một số dịch vụ đám mây xử lý tự động vấn đề này (Amazon gọi là heat management / adaptive capacity) [1].

3.6.2 Rebalancing: sự nguy hiểm của tự động hóa

Việc di chuyển shard giữa các nút (rebalancing) kết hợp với phát hiện lỗi tự động có thể gây cascading failure (phân tích ở Mục 2.5.6). Các cân nhắc thực tế [1]:

- Tự động hóa: tiện lợi và tự mở rộng (DynamoDB tự thêm/bớt shard theo tải trong vài phút), nhưng khó dự đoán; thao tác tách shard (shard-splitting) tiêu tốn tài nguyên và có thể không theo kịp tốc độ phép ghi khi hệ thống gần quá tải [1].
- Thủ công hoặc giữ con người trong vòng lặp (human in the loop): chậm hơn nhưng tránh được các "bất ngờ vận hành"; hữu ích cho việc rebalance chủ động trước các sự kiện đã biết (Cyber Monday, World Cup) [1].
- Phương án trung gian: Couchbase và Riak đề xuất tự động hóa nhưng yêu cầu quản trị viên phê duyệt trước khi thực hiện [1].

3.6.3 Định tuyến request và dịch vụ điều phối

Ba cách tiếp cận định tuyến request [1]:

1. Client gửi request tới nút bất kỳ; nút không chứa dữ liệu sẽ chuyển tiếp request

tới nút đúng.

2. Một tầng định tuyến (routing tier) đóng vai trò load balancer nhận biết shard, như mongos của MongoDB.
3. Client thông minh (smart client) tự biết ánh xạ shard→nút và kết nối trực tiếp.

Với các cách 2 và 3, phải trả lời hai câu hỏi: ai quyết định shard nào nằm trên nút nào, và thành phần định tuyến học các thay đổi như thế nào. Giải pháp chuẩn: một dịch vụ điều phối (ZooKeeper, etcd) sử dụng consensus để lưu ánh xạ shard→nút, bảo đảm không xảy ra split brain; tầng định tuyến đăng ký nhận thông báo thay đổi (HBase, SolrCloud dùng ZooKeeper; Kubernetes dùng etcd; MongoDB dùng config server và mongos; Kafka, YugabyteDB, TiDB, ScyllaDB dùng Raft nội bộ). Riak dùng giao thức gossip, yếu hơn và có thể gây split brain, nhưng được chấp nhận vì leaderless database vốn đã có các bảo đảm nhất quán yếu [1]. Một vấn đề cần xử lý thêm: trong thời gian một shard được chuyển từ nút này sang nút khác, các request đang thực thi (in flight) hướng tới nút cũ phải được xử lý một cách nhất quán.

3.6.4 Secondary index trên cơ sở dữ liệu phân mảnh

Chỉ mục phụ (secondary index) không ánh xạ gọn gàng vào các shard. Hai hướng tiếp cận [1]:

- Chỉ mục cục bộ (local / document-partitioned index): mỗi shard xây dựng chỉ mục trên phần dữ liệu của mình. Phép ghi nhanh vì chỉ tác động tới shard chứa bản ghi; nhưng phép đọc theo chỉ mục phụ phải truy vấn mọi shard rồi gộp kết quả (scatter/gather), tốn kém, dễ gây khuếch đại độ trễ đuôi, và việc thêm shard không làm tăng thông lượng truy vấn. Được sử dụng trong MongoDB, Riak, Cassandra, Elasticsearch, SolrCloud, VoltDB [1].
- Chỉ mục toàn cục (global / term-partitioned index): chỉ mục được phân mảnh theo giá trị của từ khóa, mỗi shard chỉ mục phủ một phần các từ khóa và chứa các bản ghi từ mọi shard. Phép đọc gọn gàng hơn vì một shard chứa từ khóa cần tìm; nhưng phép ghi chậm vì một phép ghi có thể phải cập nhật chỉ mục trên nhiều shard, và thường được thực hiện bất đồng bộ [1].

3.7 Kiểm chứng độ tin cậy: fault injection, Jepsen và deterministic simulation

3.7.1 Sự cần thiết của kiểm chứng chủ động

Các lỗi trong hệ thống phân tán thường nằm ẩn (latent): điều kiện gây lỗi hiếm gặp (mạng bị phân chia 43 giây như sự cố GitHub, giây nhuận năm 2012, GC pause kéo dài) khiến kiểm thử thông thường không chạm tới được. DDIA liệt kê

các nguyên nhân lỗi phần mềm mang tính hệ thống (systematic software faults): một bug khiến mọi nút sập cùng lúc trong một hoàn cảnh cụ thể (giây nhuận 2012 làm treo các ứng dụng Java; một firmware khiến toàn bộ ổ SSD của một dòng sản phẩm hỏng sau đúng 32.768 giờ hoạt động), tiến trình mất kiểm soát chiếm dụng tài nguyên chung, dịch vụ phụ thuộc chậm lại, hành vi nổi lên (emergent) từ sự tương tác của nhiều hệ thống, và các sự cố lan truyền (cascading failures) [1].

3.7.2 Fault injection và Jepsen

Jepsen (của Kyle Kingsbury) là công cụ tiêm lỗi vào một hệ cơ sở dữ liệu đang chạy, tạo phân chia mạng, dừng nút, làm lệch đồng hồ, làm sập nút, rồi kiểm tra xem các bảo đảm nhất quán của hệ thống có bị vi phạm hay không. Nhiều hệ thống nổi tiếng từng bị Jepsen phát hiện các lỗi nghiêm trọng; ngay cả MySQL 8.0.34 cũng từng bị phát hiện vi phạm tính cô lập vào năm 2023 [1]. Giá trị cốt lõi của Jepsen: biến một câu khẳng định bằng lời về bảo đảm nhất quán thành một thử nghiệm có thể lặp lại, thay vì chỉ tin vào tài liệu, nhà phát triển kiểm chứng bằng thí nghiệm có thể chạy lại được.

3.7.3 Deterministic simulation testing (DST)

DST hiện được coi là kỹ thuật kiểm chứng mạnh nhất: toàn bộ hệ thống được chạy trong một trình mô phỏng kiểm soát được, độ trễ và mất mát của mạng, sự lệch và nhảy của đồng hồ, lịch trình dừng tiến trình đều được điều khiển sao cho mỗi lần chạy là xác định (deterministic). Hệ quả: mọi lỗi bắt gặp đều tái tạo được, điều gần như không thể trong hệ thống phân tán thực. FoundationDB xây dựng và kiểm chứng toàn bộ cơ sở dữ liệu của mình theo phương pháp này; TigerBeetle (hệ thống ngân hàng dạng phần mềm) cũng áp dụng tương tự [1]. DDIA xếp DST cùng với phương pháp hình thức (formal methods, gồm TLA+ và model checking) vào nhóm các cách kiểm chứng tính đúng đắn của hệ thống phân tán hiện đại [1].

3.7.4 Chaos engineering trong vận hành

Trong vận hành production, chaos engineering chủ động tiêm lỗi vào hệ thống đang chạy (Netflix Chaos Monkey và các công cụ cùng họ) để rèn luyện khả năng tự phục hồi của hệ thống và của đội vận hành. Khác với DST, vốn tập trung kiểm chứng tính đúng đắn trước khi phát hành, chaos engineering hướng tới hành vi của hệ thống đang chạy dưới tác động của sự cố, giúp phát hiện các điểm yếu vận hành chứ không chỉ lỗi thuật toán.

3.8 CDC, outbox pattern và event sourcing

3.8.1 Bài toán giữ nhiều hệ lưu trữ đồng bộ

Như đã nêu ở Mục 2.5.7, dual writes (ghi tường minh vào từng hệ thống) gặp hai vấn đề: race condition đảo thứ tự và trường hợp một phép ghi thành công trong khi phép ghi kia thất bại. Lời giải nền tảng: biến một hệ thống thành "leader" và các hệ thống còn lại thành "follower", tức sử dụng change data capture (CDC) [1].

3.8.2 Change data capture (CDC)

CDC là quá trình quan sát mọi thay đổi dữ liệu được ghi vào một cơ sở dữ liệu và trích xuất chúng dưới dạng có thể sao chép sang các hệ thống khác, lý tưởng nhất là thành một stream ngay khi phép ghi xảy ra [1]. Ý tưởng cốt lõi: sử dụng chính replication log của cơ sở dữ liệu làm nguồn sự thật về thứ tự. Cơ sở dữ liệu quyết định thứ tự thực thi các phép ghi và ghi vào log theo đúng thứ tự đó; search index, data warehouse là các consumer của stream, áp dụng các thay đổi theo đúng thứ tự. Điều này giải quyết race condition của dual writes (Mục 2.5.7): không còn hai "leader" tranh giành thứ tự, chỉ còn một thứ tự duy nhất do cơ sở dữ liệu quyết định [1].

Triển khai CDC [1]:

- Sử dụng logical replication log (log mức dòng); cách này gặp các thách thức về schema changes và cách mô hình hóa các phép cập nhật.
- Nghiên cứu tình huống 14 – Debezium [1]: Debezium (mã nguồn mở) cung cấp các source connector cho MySQL, PostgreSQL, Oracle, SQL Server, Db2, Cassandra...: gắn vào replication log của cơ sở dữ liệu nguồn và phát các thay đổi theo một event schema chuẩn, sau đó chuyển xuống các hệ thống đích. Kafka Connect cũng cung cấp các CDC connector; Maxwell làm tương tự cho MySQL (phân tích binlog), GoldenGate cho Oracle, pgcapture cho PostgreSQL [1].
- Initial snapshot: khi cần dựng một chỉ mục mới (ví dụ full-text index), chỉ phát lại log từ đầu là không đủ vì log bị cắt ngắn (truncate); phải bắt đầu từ một consistent snapshot rồi áp dụng phần log còn thiếu (backlog), tương tự quy trình thiết lập follower mới [1].

Các vấn đề của CDC [1]:

- CDC sử dụng schema của cơ sở dữ liệu nguồn, do đó biến schema đó thành public API. Việc xóa một cột có thể làm hỏng các consumer phía sau và gây sự cố ảnh hưởng tới khách hàng. Giải pháp: sử dụng data contracts và outbox pattern (dưới đây).

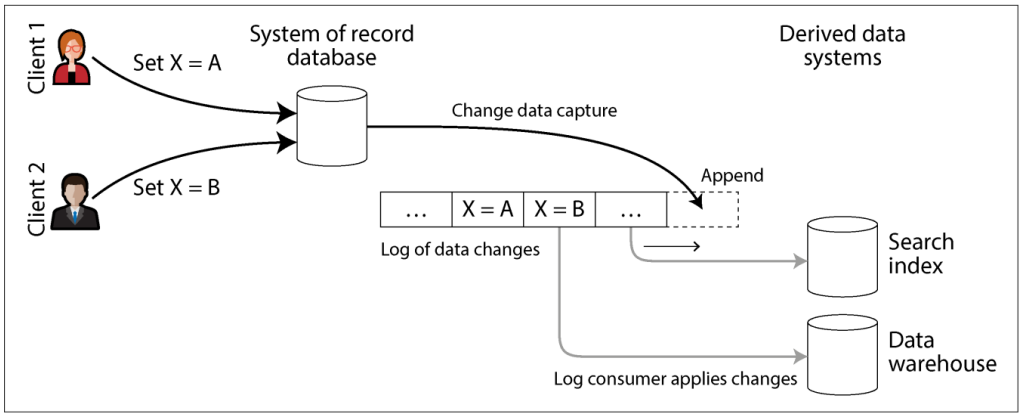
- CDC thường bất đồng bộ: cơ sở dữ liệu nguồn không chờ consumer, do đó mọi vấn đề của replication lag (Mục 2.3) đều áp dụng [1].

3.8.3 Transactional outbox pattern

Outbox pattern: thay vì ghi sự kiện trực tiếp ra hàng đợi hoặc stream (một dạng dual write), ứng dụng ghi vào một bảng outbox nằm trong chính cơ sở dữ liệu cùng với dữ liệu nghiệp vụ, trong cùng một transaction. Bảng outbox có schema riêng (được phơi ra cho CDC) tách khỏi schema nội bộ của ứng dụng. CDC đọc bảng outbox và phát sự kiện ra stream [1].

Ưu điểm: bề ngoài giống dual write, nhưng cả hai phép ghi nằm trong cùng một hệ thống (cơ sở dữ liệu) và cùng một transaction, tránh được cả race condition lẫn vấn đề atomic commit của dual writes xuyên hệ thống (Mục 2.5.7). Các nhà phát triển được tự do thay đổi schema nội bộ miễn là giữ bảng outbox ổn định. Đổi lại: phải duy trì quá trình chuyển đổi (transformation) giữa schema nội bộ và schema outbox, và khối lượng dữ liệu cơ sở dữ liệu phải ghi tăng lên [1].

Sự kết hợp giữa outbox pattern và CDC được biểu diễn trong Hình 3.4.



Hình 3.4: Outbox pattern phối hợp với CDC để phát sự kiện ra stream

3.8.4 Event sourcing và tính bất biến

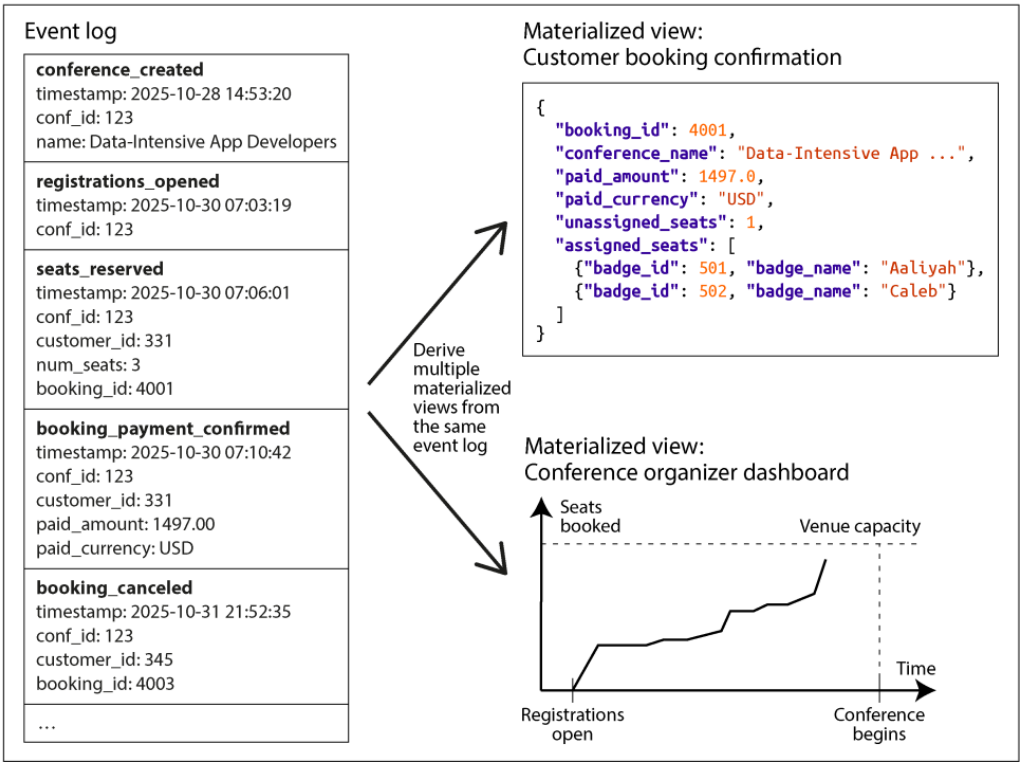
Event sourcing mô hình hóa sự thay đổi trạng thái như một log các sự kiện bất biến (immutable events), trong đó mỗi sự kiện diễn tả ý định của một hành động người dùng, không phải cơ chế cập nhật trạng thái. Trạng thái hiện tại được hiểu là kết quả của việc phát lại (replay) log theo thời gian; change stream là kết quả của việc "lấy đạo hàm" trạng thái theo thời gian [1].

Phân biệt với CDC [1]:

- CDC: một sự kiện cập nhật thường chứa toàn bộ phiên bản mới của bản ghi; log compaction có thể loại bỏ các sự kiện cũ cùng khóa, vì bản mới nhất đã đủ để xác định trạng thái.

- Event sourcing: các sự kiện ở mức cao hơn, không ghi đè lên nhau, nên cần toàn bộ lịch sử để dựng lại trạng thái; log compaction không áp dụng được. Các ứng dụng event sourcing thường lưu snapshot của trạng thái hiện tại để tăng tốc phép đọc và phục hồi, nhưng đó chỉ là tối ưu hiệu năng; thiết kế vẫn cho phép lưu mọi sự kiện vĩnh viễn và phát lại toàn bộ khi cần [1].

Mối quan hệ giữa sự kiện bất biến, log và trạng thái hiện tại được minh họa trong Hình 3.5.



Hình 3.5: Event sourcing: trạng thái hiện tại là kết quả phát lại log sự kiện bất biến

Về mặt tư duy, trạng thái biến đổi và log bất biến không mâu thuẫn với nhau; chúng là hai mặt của một đồng xu. "Không có nhu cầu cơ bản nào để phải giữ một cơ sở dữ liệu; log chứa mọi thông tin; lý do duy nhất để lưu trữ cơ sở dữ liệu (trạng thái cuối của log) là hiệu năng truy xuất" (Jim Gray & Andreas Reuter, 1992) [1]. Giới hạn của tính bất biến: một số loại dữ liệu bắt buộc phải xóa (quyền xóa dữ liệu theo GDPR), dữ liệu sai phải được sửa; event sourcing ứng phó bằng "sự kiện bù trừ" (compensating events) [1].

3.8.5 Unbundling databases: triết lý tổng thể

Nhìn ở tầm toàn cảnh, dòng dữ liệu xuyên suốt một tổ chức trông như một cơ sở dữ liệu khổng lồ (meta-database of everything): mỗi tiến trình batch, stream hay ETL chuyển dữ liệu từ nơi này sang nơi khác chính là một subsystem giữ cho các chỉ mục và materialized view luôn được cập nhật; các derived data systems (search

index, cache, data warehouse, đặc trưng máy học) giống như các loại chỉ mục khác nhau [1]. Hai hướng kiến trúc được DDIA trình bày [1]:

- Federated databases (hợp nhất phép đọc): giao diện truy vấn hợp nhất trên nhiều hệ thống lưu trữ (PostgreSQL foreign data wrappers, Trino) giải quyết vấn đề đọc dữ liệu từ nhiều nguồn, nhưng không giải quyết việc đồng bộ phép ghi.
- Unbundled databases (hợp nhất phép ghi): làm cho việc kết nối các hệ thống lưu trữ trở nên tin cậy thông qua CDC và event log, để mọi thay đổi dữ liệu tới mọi nơi cần đến, ngay cả khi có sự cố. Đây là việc tách rời ("unbundling") chức năng duy trì chỉ mục của cơ sở dữ liệu thành các công cụ độc lập. Cách đồng bộ phép ghi khả thi nhất: asynchronous event log với idempotent writes, thay cho distributed transactions xuyên hệ thống lưu trữ [1].

3.9 Vận hành và giám sát

3.9.1 Giảm tail latency

Các kỹ thuật giảm độ trễ đuôi được DDIA và thực hành công nghiệp đề xuất [1]:

- Tối ưu hóa các percentile cao (p99, p999) thay vì giá trị trung bình: vì tail latency quyết định trải nghiệm người dùng và gây hiện tượng khuếch đại độ trễ đuôi (Mục 2.6.1).
- Gửi các request dự phòng (hedged requests): gửi cùng một request tới nhiều replica và sử dụng kết quả của replica nhanh nhất, một kỹ thuật kinh điển của Google. Tuy nhiên việc nhân đôi request làm tăng tải hệ thống, cần được cân nhắc kỹ [1].
- Kiểm soát tải (load shedding): khi quá tải, giảm tải có chủ đích thay vì xử lý mọi request tới cùng, nhằm tránh bão retry và sự cố lan truyền.

3.9.2 Phòng chống cascading failure và retry storm

Các biện pháp cụ thể [1]:

- Tránh các vòng phản hồi: retry kèm jitter và exponential backoff; giới hạn số lần retry.
- Thiết lập timeout hợp lý cho từng lớp thành phần, không chỉ cho toàn bộ request.
- Giảm tải chủ động (load shedding, hàng đợi ưu tiên) để hệ thống "thở" khi quá tải thay vì bóp nghẹt chính mình.
- Độc lập hóa các thành phần: kiến trúc cell-based giới hạn ảnh hưởng của một

sự cố trong một "cell", các cell khác không bị ảnh hưởng [1].

3.9.3 Observability và blameless postmortem

- Distributed tracing: OpenTelemetry, Zipkin, Jaeger theo dõi chuỗi gọi client→server và thời gian từng chặng (Mục 2.6.4) [1].
- SLO/SLA và percentile: đo lường đúng cách; percentile phải được kết hợp bằng cách cộng các histogram, không được cộng hoặc lấy trung bình các percentile [1].
- Blameless postmortem: thay vì đổ lỗi cho cá nhân, điều tra hệ thống kỹ thuật–xã hội. Câu kết luận kiểu "Bob cần cẩn thận hơn" không hiệu quả, nhưng đề xuất kiểu "phải viết lại backend bằng một ngôn ngữ khác" cũng vô nghĩa. Bài học phải được rút ra từ góc nhìn của người vận hành hàng ngày [1].
- Giảm thiểu lỗi con người bằng thiết kế: kiểm thử (cả thủ công lẫn property-based), khả năng rollback nhanh, triển khai dần dần (gradual rollout), và giám sát rõ ràng [1].

3.10 Tổng kết chương 3

Chương 3 đã trình bày các giải pháp tương ứng với từng nhóm vấn đề được phân tích ở Chương 2:

1. Mô hình nhất quán: một dải liên tục từ eventual consistency tới linearizability; mức nhất quán được lựa chọn theo từng thao tác nghiệp vụ, không theo toàn hệ thống (minh họa bằng Cosmos DB, S3, DynamoDB); session guarantee được triển khai bằng token thay cho sticky routing.
2. Giao dịch: từ read committed qua snapshot isolation/MVCC tới serializable (thực thi tuần tự, 2PL, SSI); CockroachDB minh họa việc triển khai SSI trong một cơ sở dữ liệu phân tán.
3. Giao dịch phân tán & NewSQL: 2PC chỉ khả thi trong phạm vi một công nghệ lưu trữ; vượt qua ranh giới công nghệ, sử dụng idempotence để đạt exactly-once; Spanner (TrueTime) và FoundationDB (DST, SSI) là hai đại diện tiêu biểu.
4. Replication: single-leader (semisynchronous, quorum, failover an toàn nhờ consensus và fencing), multi-leader (phân tán địa lý, giải quyết xung đột bằng CRDT/OT), leaderless (quorum; Cassandra và DynamoDB).
5. Consensus & điều phối: Raft và total order broadcast; ZooKeeper/etcd/Consul với lock, lease, fencing token, phát hiện lỗi, thông báo thay đổi; "outsource" consensus cho một tập nhỏ nút chuyên biệt.

6. Phân mảnh & định tuyến: key range so với hash; consistent hashing; rebalancing cần thận trọng; định tuyến qua dịch vụ điều phối; secondary index cục bộ so với toàn cục.
7. Kiểm chứng: fault injection, Jepsen, deterministic simulation testing, chaos engineering.
8. Đồng bộ đa hệ lưu trữ: CDC (Debezium), transactional outbox pattern, event sourcing, và triết lý unbundled databases.
9. Vận hành: giảm tail latency, phòng chống cascading failure, observability, blameless postmortem.

Chương tiếp theo tổng kết toàn bộ báo cáo, nêu các bài học thiết kế và các hướng nghiên cứu phát triển.

CHƯƠNG 4. KẾT LUẬN VÀ BÀI HỌC

4.1 Tổng kết các vấn đề trong hệ thống phân tán

Báo cáo đã hệ thống hóa các vấn đề trong hệ thống phân tán thành năm nhóm:

1. Nền tảng lý thuyết (CAP, PACELC). Một hệ thống phân tán luôn phải đánh đổi: khi mạng bị phân chia (partition), giữa *consistency* (nhất quán) và *availability* (sẵn sàng); khi mạng bình thường, giữa *latency* (độ trễ) và *consistency*. Điều quan trọng nhất không phải "chọn 2 trong 3" mà là *detect* partition, *xử lý* trong lúc đó, và *dọn dẹp* sau đó; phần dọn dẹp thường tốn thời gian nhất (sự cố GitHub 21/10/2018: 43 giây partition dẫn tới 24 giờ 11 phút khôi phục).
2. Vấn đề môi trường. Mạng không đáng tin cậy (mất gói, trễ biến thiên, timeout không phân biệt được "chết hẳn/chậm/đứt"), đồng hồ không đáng tin (time-of-day clock có thể nhảy, monotonic clock không có ý nghĩa tuyệt đối), và tiến trình có thể bị dừng đột ngột trong thời gian dài (GC, VM suspend). Hệ quả: không bao giờ chắc chắn về trạng thái thật của nút khác; cần quy tắc đa số, lease, fencing token, và lựa chọn system model đúng (mô hình partially synchronous + crash-recovery là thực tế nhất).
3. Vấn đề nhất quán. Sao chép bất đồng bộ gây replication lag, dẫn tới ba dị thường: đọc không thấy write của mình (read-your-writes), thấy thời gian quay ngược (monotonic reads), và thấy câu trả lời trước câu hỏi (consistent prefix). Các bảo đảm nhất quán tạo thành một dải từ eventual tới linearizability; vi phạm nhất quán rất khó phát hiện vì không gây exception.
4. Vấn đề giao dịch và consensus. Các mức cô lập yếu để lọt dirty read, read skew, lost update (ví dụ Flexcoin mất 600 BTC), write skew (ví dụ hai bác sĩ trực), phantom. Giao dịch phân tán đòi hỏi atomic commit; 2PC có điểm yếu chí mạng là giao dịch *in doubt* treo vô hạn khi coordinator sập. Consensus (Paxos, Raft) đắt và tinh tế, là nền tảng của leader election an toàn.
5. Vấn đề thiết kế và vận hành. Sao chép có ba họ với các rủi ro riêng; phân mảnh gặp skew/hot spot và rebalancing nguy hiểm (có thể gây cascading failure); dual writes gây race condition và vấn đề atomic commit; vận hành đối mặt tail latency amplification, cascading failure, retry storm, và lỗi con người (nguyên nhân hàng đầu gây outage).

4.2 Tổng kết các giải pháp

- Mô hình nhất quán: lựa chọn theo từng thao tác nghiệp vụ. Nấc session (read-your-writes, monotonic reads...) là mặc định hợp lý cho nhiều ứng dụng (Cos-

mos DB mặc định session); causal consistency là trần của phe ưu tiên availability; bounded staleness và consistent prefix là các điểm cân bằng "cũ nhưng có giới hạn". Token thay cho sticky routing giúp giữ guarantee bám vào dữ liệu thay vì đường mạng.

- Giao dịch: snapshot isolation/MVCC là điểm cân bằng phổ biến; serializable đạt được bằng serial execution (đơn giản, giới hạn mở rộng), 2PL (cổ điển, chậm) hoặc SSI (optimistic, hiện đại; CockroachDB, FoundationDB).
- Giao dịch phân tán: 2PC chỉ khả thi trong cùng một công nghệ lưu trữ; qua biên giới công nghệ, dùng idempotence để có exactly-once semantics. Spanner dùng TrueTime (đồng hồ nguyên tử + GPS) để có serializable ở quy mô toàn cầu; FoundationDB dùng SSI + Raft + deterministic simulation testing.
- Replication: single-leader là mặc định cho các ràng buộc mạnh; multi-leader cho địa lý phân tán và collaborative apps (CRDT/OT); leaderless + quorum ($w + r > n$) cho khả năng chịu partition cao (Cassandra, DynamoDB).
- Consensus & điều phối: dùng coordination service (ZooKeeper, etcd) trên một tập nhỏ nút cố định, với lock, lease, fencing token, failure detection và notifications; tránh tự triển khai consensus từ đầu.
- Đồng bộ đa hệ lưu trữ: CDC (Debezium) + transactional outbox + event log bất biến + idempotent writes là phương án khả thi nhất, thay cho distributed transactions xuyên công nghệ.
- Kiểm chứng độ tin cậy: fault injection (Jepsen), deterministic simulation testing (FoundationDB, TigerBeetle), chaos engineering, và formal methods.

4.3 Bài học thiết kế cho kỹ sư

1. Không có "giải pháp đúng cho mọi nơi", chỉ có trade-off (Thomas Sowell, dẫn trong DDIA). Trước mỗi quyết định (mức nhất quán, mức cô lập, họ sao chép), hãy hỏi: thao tác nghiệp vụ này cần gì, và chấp nhận trả giá gì?
2. Chọn mức nhất quán/cô lập theo từng thao tác, không theo cả hệ thống. Phép đọc số dư trước lệnh rút tiền và phép đọc số lượt thích một bài đăng không nên dùng chung một mức [3].
3. Thiết kế cho sự thất bại (design for failure). Giả định mạng có thể đứt, đồng hồ có thể lệch, tiến trình có thể pause. Luôn có phương án: quorum, lease + fencing token, retry có jitter và exponential backoff, giảm tải có chủ đích.
4. Timeout là con dao hai lưỡi. Quá ngắn gây failover nhầm (GitHub); quá dài chậm phát hiện sự cố. Không có con số đúng cho mọi hoàn cảnh; cần đo và

điều chỉnh theo thực tế.

5. Đừng tự triển khai consensus. Dùng các thuật toán đã được kiểm chứng (Raft) hoặc coordination service chuyên dụng; "outsource" consensus cho một tập nhỏ nút.
6. Khi đồng bộ nhiều hệ lưu trữ, dùng event log + CDC + idempotence thay vì dual writes hay distributed transactions xuyên công nghệ.
7. Kiểm chứng tính đúng đắn là bắt buộc, không phải tùy chọn. Với các thuật toán phân tán: formal methods + DST + fault injection; với hệ thống đang chạy: chaos engineering + observability (tracing) + blameless postmortem.
8. Nghĩ về phần dọn dẹp (recovery) ngay từ đầu. Phần khó nhất của một sự cố phân tán không phải là phát hiện, mà là khôi phục nhất quán sau đó. Hãy chuẩn bị trước quy trình reconcile cho các tình huống "hai nguồn sự thật".

4.4 Hạn chế của báo cáo và hướng phát triển

Báo cáo tập trung vào các khía cạnh phi chức năng và lý thuyết nền tảng, minh họa bằng các hệ thống công nghiệp được mô tả trong tài liệu công khai. Các hạn chế:

- Chưa thực nghiệm trên hệ thống phân tán cụ thể; các nhận định dựa trên tài liệu nghiên cứu và báo cáo sự cố công khai.
- Một số chủ đề liên quan (chi tiết triển khai TCP/backpressure, thuật toán Paxos chi tiết, vector clock, các thuật toán CRDT cụ thể) chỉ được đề cập ở mức khái niệm.
- Hướng phát triển tiếp theo: xây dựng một nguyên mẫu hệ thống phân tán nhỏ (ví dụ key-value store với Raft/quorum) và kiểm chứng các bảo đảm nhất quán bằng Jepsen; hoặc phân tích sâu một sự cố thực tế (GitHub, DynamoDB) theo khung CAP/PACELC và các mô hình nhất quán.

TÀI LIỆU THAM KHẢO

[1] M. Kleppmann and C. Riccomini, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, 2nd. 2026.

[2] V. J. U. Group, *Luyên thuyên về cap theorem và pacelc (beginner++)*, Bài đăng công khai trên nhóm Vietnam JUG, 2026.

[3] V. J. U. Group, *Luyên thuyên về consistency models (intermediate++)*, Bài đăng công khai trên nhóm Vietnam JUG, 2026.

[4] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, 2002.

[5] E. A. Brewer, "Cap twelve years later: How the "rules" have changed," *Computer (IEEE)*, 2012.

[6] D. J. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *IEEE Computer*, 2012.

[7] W. Vogels, "Eventually consistent," *Communications of the ACM*, 2009.