Chương 6:

Câu 1:

Khái niệm "tương tranh" là gì?
“Tương tranh” (tiếng Anh: race condition hoặc concurrency) là hiện tượng xảy ra khi nhiều tiến trình hoặc luồng truy cập và thao tác đồng thời lên cùng một tài nguyên dùng chung, như vùng nhớ, biến dữ liệu, hay tập tin, mà trình tự thực hiện các thao tác không được kiểm soát chặt chẽ, dẫn đến kết quả sai lệch, không xác định hoặc không nhất quán.

Định nghĩa khái niệm tương tranh theo Edsger Dijkstra:
Edsger Dijkstra – nhà khoa học máy tính tiên phong trong lĩnh vực đồng bộ hóa tiến trình – định nghĩa tương tranh như sau:

“Concurrency is not interleaving of actions; it is the absence of order.”
(Tạm dịch: “Tương tranh không chỉ là sự xen kẽ của các hành động; mà là sự vắng mặt của trật tự.”)

Theo Dijkstra, trong một hệ thống tương tranh, không có thứ tự cố định nào đảm bảo cho các hành động xảy ra, nên việc xử lý tài nguyên chung phải được đồng bộ hóa để đảm bảo tính nhất quán.

Hai hiện tượng phổ biến do tương tranh:

Mất cập nhật (Lost Update):

- Nguyên nhân: Hai tiến trình đọc cùng một dữ liệu, sau đó cập nhật và ghi đè lên nhau, khiến một trong hai bản cập nhật bị mất.
- Mô tả:
  Giả sử hai tiến trình T1 và T2 cùng đọc giá trị của biến x = 5.
  - T1 tăng x lên 6 rồi ghi lại.
  - T2 cũng tăng x lên 6 rồi ghi lại.
    Kết quả cuối cùng là x = 6, nhưng lẽ ra phải là x = 7. → Một cập nhật bị mất.

Đọc không nhất quán (Inconsistent Read):

- Nguyên nhân: Một tiến trình đang đọc dữ liệu tại thời điểm dữ liệu đó đang bị tiến trình khác cập nhật dở dang, dẫn đến kết quả đọc không đầy đủ hoặc sai lệch.
- Mô tả:
  Giả sử tiến trình T1 đang chuyển tiền từ tài khoản A sang B:
  - Trừ 100 từ A (xong)

- Chưa kịp cộng 100 vào B
  Trong khi đó, tiến trình T2 đọc số dư A và B → thấy tổng số tiền ít hơn thực tế.
  → Đọc dữ liệu chưa nhất quán.

Mất cập nhật (Lost Update)

Hiện tượng:
Xảy ra khi hai tiến trình đồng thời truy cập và cập nhật cùng một tài khoản, nhưng do thực hiện không đồng bộ, một trong hai cập nhật bị ghi đè và mất đi.

Ví dụ:
Giả sử có hai tiến trình thực hiện thao tác:

- Tiến trình T1: Chuyển 100 từ tài khoản A sang B
- Tiến trình T2: Chuyển 50 từ tài khoản A sang C

Giả sử ban đầu số dư của A là 1000. Trình tự thực thi có thể như sau:

- T1 thực hiện GetBalance(A) → 1000
- T2 thực hiện GetBalance(A) → 1000
- T1 thực hiện SetBalance(A, 900)
- T2 thực hiện SetBalance(A, 950)

Kết quả cuối cùng tài khoản A có số dư là 950 thay vì đúng ra là 850. Lý do là vì T2 đã ghi đè kết quả cập nhật của T1 do cả hai đều lấy số dư ban đầu là 1000 và cập nhật độc lập. Cập nhật của T1 đã bị mất.

Nguyên nhân:
Thiếu cơ chế đồng bộ khi thực hiện đọc và ghi đồng thời trên cùng một dữ liệu.

Đọc không nhất quán (Inconsistent Read)

Hiện tượng:
Xảy ra khi một tiến trình đọc dữ liệu tại thời điểm mà một tiến trình khác đang cập nhật dở dang. Kết quả đọc sẽ phản ánh một trạng thái không đúng hoặc không đầy đủ.

Ví dụ:
Giả sử tiến trình T1 chuyển 100 từ tài khoản A sang B, và tiến trình T2 thực hiện tính tổng số dư của A và B.

Trình tự thực hiện:

- T1 thực hiện GetBalance(A) → 1000
- T1 thực hiện SetBalance(A, 900)
- T2 thực hiện GetBalance(A) → 900
- T2 thực hiện GetBalance(B) → 1000
- T1 thực hiện SetBalance(B, 1100)

Tổng số dư mà T2 tính được là 900 + 1000 = 1900, trong khi tổng đúng phải là 900 + 1100 = 2000. Như vậy, T2 đã đọc dữ liệu ở một trạng thái chưa hoàn tất do T1 đang thực hiện chuyển tiền nhưng chưa cập nhật xong.

Nguyên nhân:
Tiến trình đọc đã truy cập vào các dữ liệu trung gian khi một giao dịch chưa hoàn tất, dẫn đến kết quả không phản ánh đúng trạng thái thực tế của hệ thống.

MoneyTransfer(Accounts X, Accounts Y, float Z)

```
{
   float Balance = Y.GetBalance();       // Lấy số dư tài khoản Y
   float Trans = Balance * Z;            // Tính số tiền cần chuyển
   Y.SetBalance(Balance + Trans);        // Cập nhật số dư tài khoản Y
   X.Withdraw(Trans);                    // Trừ tiền từ tài khoản X
}
```

Vấn đề của đoạn mã (do không có giao tác)

Không có đồng bộ giữa các thao tác:

Nếu nhiều luồng/thao tác truy cập và cập nhật cùng lúc tài khoản X hoặc Y, có thể dẫn đến mất cập nhật hoặc trạng thái không nhất quán.

Không đảm bảo nguyên tử:

Nếu chương trình bị gián đoạn sau khi Y.SetBalance(...) nhưng trước khi X.Withdraw(...), thì hệ thống đã cộng tiền vào Y nhưng chưa trừ ở X → sai lệch dữ liệu.

Hai điểm cần sửa đổi

(1) Gộp các thao tác lại thành một giao tác (transaction):

Đảm bảo rằng toàn bộ thao tác phải thực hiện hết hoặc không thực hiện gì cả (tính nguyên tử - atomicity).

Dùng begin_transaction(), commit(), rollback().

(2) Đồng bộ hóa truy cập tài khoản bằng khóa (lock/mutex):

Ngăn chặn cạnh tranh truy cập (race condition) giữa các tiến trình cùng cập nhật X hoặc Y.

Minh họa bằng pseudocode có giao tác

Cách 1: Sử dụng giao tác đơn giản

cpp

Sao chép

Chỉnh sửa

```
MoneyTransfer(Accounts X, Accounts Y, float Z)
{
```

```
begin_transaction();
try {
    float balanceY = Y.GetBalance();
    float transAmount = balanceY * Z;

    Y.SetBalance(balanceY + transAmount);
    X.Withdraw(transAmount);

    commit_transaction();
} catch (Exception e) {
    rollback_transaction();
}
```

. Bối cảnh

- Số dư ban đầu của B: 200$
- Hai tiến trình đồng thời:
    - T1: A chuyển 20$ vào B
    - T2: C chuyển 20$ vào B
- Công thức tính tiền chuyển: Trans = Balance * 0.1
- Kỳ vọng: sau cả hai chuyển tiền, B phải có ≥240$
- Thực tế: kết quả B = 220$ → mất 20$

Trình tự thực thi 8 câu lệnh (theo thời gian)

Giả sử hai tiến trình T1 (từ A) và T2 (từ C) cùng thực thi, xen kẽ như sau:

Bước	Tiến trình	Lệnh thực thi	Ghi chú

1	T1	Balance = B.GetBalance(); → 200$	Đọc số dư B: 200
2	T1	Trans = Balance * 0.1; → 20$	Tính tiền chuyển
3	T2	Balance = B.GetBalance(); → 200$	Cũng đọc số dư 200 (do B chưa cập nhật từ T1)
4	T2	Trans = Balance * 0.1; → 20$	Tính tiền chuyển của C
5	T1	B.SetBalance(Balance + Trans); → 220$	B = 200 + 20 = 220
6	T2	B.SetBalance(Balance + Trans); → 220$	Ghi đè lại: B = 200 + 20 = 220 (ghi mất cập nhật của T1)
7	T1	A.Withdraw(Trans); → A = 100 - 20 = 80	Trừ tiền từ A
8	T2	C.Withdraw(Trans); → C = 300 - 20 = 280	Trừ tiền từ C

Kết quả sau cùng

- B chỉ có 220$, trong khi đáng lẽ phải là 200 + 20 + 20 = 240$
- Sai lệch do ghi đè (lost update):
    - T1 và T2 đều đọc cùng số dư (200), cộng thêm 20, và cùng gọi SetBalance(...) → lần ghi thứ hai đè mất kết quả của lần đầu.

Nguyên nhân sai lệch

- Cả hai tiến trình không đồng bộ, cùng đọc số dư ban đầu B = 200
- Cùng tính toán và cập nhật B độc lập → không có sự cách ly giữa các tiến trình
- Không đảm bảo tính nguyên tử (atomicity) → lỗi do cạnh tranh truy cập

Vấn đề nếu chỉ gom 3 câu lệnh đầu vào giao tác

Giả sử chỉ gom phần đọc và tính toán vào một giao tác (như sau):

cpp
Sao chépChỉnh sửa
```
begin_transaction()
   Balance = B.GetBalance();      // Câu 1
   Trans = Balance * 0.1;         // Câu 2
end_transaction()
```

B.SetBalance(Balance + Trans); // Câu 3 (không nằm trong giao tác)
A.Withdraw(Trans); // Câu 4

Sơ hở:

- Dù phần đọc và tính toán nằm trong giao tác, phần ghi kết quả (SetBalance) vẫn diễn ra bên ngoài giao tác
- Khi đó, các tiến trình vẫn có thể ghi đè kết quả của nhau giống như kịch bản trên
  → Giao tác không đảm bảo tính toàn vẹn vì chỉ bảo vệ phần đọc, không bảo vệ phần cập nhật

Dưới đây là thiết kế kiến trúc điều khiển tương tranh cho hệ thống phân tán lưu trữ giao dịch ngân hàng, đáp ứng ba yêu cầu quan trọng:

- Không xảy ra mất cập nhật và đọc không nhất quán
- Giảm thiểu khóa chết (deadlock) và đói tài nguyên (starvation)
- Vận hành hiệu quả trên môi trường nhiều nút, nhiều bản sao dữ liệu

Mô hình dữ liệu hệ thống

- Tài khoản ngân hàng được lưu dưới dạng bản ghi (record), phân mảnh và sao chép trên nhiều nút (node) trong hệ thống phân tán.
- Mỗi tài khoản sẽ có một ID duy nhất, một số dư (balance), và một trường version (để hỗ trợ kiểm tra đồng bộ khi dùng kỹ thuật không khóa).
- Dữ liệu phân tán sử dụng mô hình replication theo nhóm (quorum replication):
  - Mỗi bản ghi tồn tại trên n nút, cập nhật yêu cầu ghi tối thiểu W nút, đọc từ ít nhất R nút, đảm bảo W + R > n.

Kiến trúc và cơ chế điều khiển tương tranh

1. Mô hình: Giao tác phân tán phối hợp với kiểm soát phiên bản và kỹ thuật không khóa (lock-free)

Thành phần chính:

- Transaction Manager (TM): quản lý giao tác xuyên nút
- Data Nodes (DN): lưu bản sao dữ liệu, xử lý đọc/ghi tại chỗ
- Version Control + Optimistic Concurrency Control (OCC): kiểm soát đồng bộ và tránh mất cập nhật mà không cần khóa

2. Cơ chế hoạt động (giải thuật giao tác)

Giao tác chuyển tiền từ A sang B thực hiện như sau:

1. Bắt đầu giao tác: TM khởi tạo giao tác T với UUID
2. Đọc dữ liệu:
    - TM gửi lệnh đọc R đến R nút chứa A và B
    - Mỗi nút trả về balance và version
3. Tính toán và tạo bản ghi mới:
    - Trên TM: A' = A - x, B' = B + x
4. Kiểm tra phiên bản (OCC validation):
    - TM gửi yêu cầu ghi kèm phiên bản gốc đến W nút
    - Nếu phiên bản dữ liệu tại nút khớp với phiên bản đọc được ban đầu, ghi sẽ thành công
    - Nếu có phiên bản bị thay đổi → giao tác bị huỷ và thực hiện lại
5. Cam kết giao tác (commit/quorum):
    - Nếu ghi thành công trên W nút → TM gửi thông báo hoàn tất
    - TM cập nhật trạng thái giao tác → thành công

Kỹ thuật giảm thiểu deadlock và starvation

- Không dùng khóa cứng (pessimistic locking) → tránh deadlock
- Không cho phép chờ vòng tròn (các nút chỉ xử lý một giao tác tại một thời điểm trên cùng dữ liệu)
- Retry giao tác bị xung đột sau thời gian ngẫu nhiên (backoff) để tránh starvation
- Có thể áp dụng conflict resolution theo thời gian hoặc ưu tiên để quyết định giao tác nào thắng

Ưu – Nhược điểm của giải pháp

Tiêu chí	Ưu điểm	Nhược điểm
Tính toàn vẹn	Đảm bảo không mất cập nhật, không đọc sai nhờ OCC và quorum	Cần đảm bảo đồng nhất version giữa nhiều nút
Hiệu quả	Không khóa → không deadlock, tốt cho giao dịch ngắn	Nếu xung đột cao, giao tác có thể bị hủy nhiều lần
Khả năng mở rộng	Dễ mở rộng theo chiều ngang (scale-out)	Chi phí truyền thông giữa các nút lớn khi số lượng nút tăng
Tương thích dữ liệu	Quorum đảm bảo nhất quán mạnh hơn so với eventual consistency	Ghi nhiều nút → độ trễ tăng so với hệ thống đơn nút

Câu 2:

Định nghĩa "Giao tác" trong hệ quản trị cơ sở dữ liệu phân tán

Giao tác (Transaction) là một đơn vị công việc logic gồm một hoặc nhiều thao tác (như đọc, ghi, cập nhật) trên cơ sở dữ liệu, được thực hiện như một khối nguyên tử nhằm đảm bảo tính nhất quán của dữ liệu.

Trong hệ quản trị cơ sở dữ liệu phân tán (Distributed Database Management System - DDBMS), một giao tác có thể bao gồm các thao tác thực hiện trên nhiều nút, nhiều CSDL độc lập, nhưng phải được xử lý như một thể thống nhất, đảm bảo dữ liệu không rơi vào trạng thái bất nhất.

Bốn tính chất ACID của giao tác

Tính chất	Diễn giải
A – Atomicity (Tính nguyên tử)	Giao tác hoặc thực hiện toàn bộ, hoặc không thực hiện gì cả. Nếu có lỗi giữa chừng, toàn bộ tác động sẽ được rollback.
C – Consistency (Tính nhất quán)	Giao tác đưa CSDL từ trạng thái nhất quán này sang trạng thái nhất quán khác, không làm sai lệch quy tắc dữ liệu.
I – Isolation (Tính độc lập)	Các giao tác thực thi độc lập, không bị ảnh hưởng lẫn nhau dù xảy ra đồng thời (tức là kết quả như thể được thực hiện tuần tự).
D – Durability (Tính bền vững)	Khi giao tác đã cam kết (commit) thì các thay đổi là vĩnh viễn, không bị mất đi kể cả khi hệ thống gặp sự cố.

So sánh 3 loại giao tác

Loại giao tác	Đặc điểm	Ví dụ minh họa đơn giản
Giao tác phẳng (Flat Transaction)	- Giao tác đơn, không chứa giao tác con- Không thể chia nhỏ hoặc rollback từng phần	Chuyển tiền từ A sang B trong một bước duy nhất
Giao tác lồng ghép (Nested Transaction)	- Giao tác chứa nhiều giao tác con bên trong- Có thể rollback một phần nếu một nhánh con thất bại	Giao dịch thương mại gồm 3 bước: kiểm tra hàng, thanh toán, cập nhật kho – mỗi bước là một giao tác con
Giao tác phân tán (Distributed Transaction)	- Giao tác thực hiện trên nhiều site, nhiều CSDL phân tán- Cần giao thức như 2PC (Two-Phase Commit) để đảm bảo tính nhất quán	Một giao tác đặt vé máy bay: cập nhật CSDL ở Hà Nội (chỗ ngồi), TP.HCM (hành khách), và ngân hàng (thanh toán)

Pseudocode ban đầu (không có giao tác):

plaintext
Sao chépChỉnh sửa

```
UpdateAccount(A, delta) {
   balance = Read(A);
   Write(A, balance + delta);
}
```

Pseudocode mới: Giao tác phẳng với COMMIT/ROLLBACK

```
UpdateAccount(A, delta) {
   BEGIN TRANSACTION;
   TRY {
      balance = Read(A);
      Write(A, balance + delta);
      COMMIT;
   } CATCH (Exception e) {
      ROLLBACK;
   }
}
```

Giải thích:

- BEGIN TRANSACTION: Khởi đầu giao tác, đảm bảo mọi thao tác sau nằm trong phạm vi giao tác.
- TRY ... CATCH: Kiểm tra nếu xảy ra lỗi trong quá trình cập nhật.
- COMMIT: Xác nhận và ghi vĩnh viễn các thay đổi nếu không có lỗi.
- ROLLBACK: Quay về trạng thái trước giao tác nếu xảy ra lỗi (giữ nguyên trạng thái nhất quán).

So sánh giao tác lồng ghép và giao tác phẳng

Bối cảnh

Giao dịch lớn T bao gồm hai phần độc lập T1 và T2, cùng truy cập một số tài nguyên chung (ví dụ: hai phần của quá trình chuyển tiền hoặc xử lý nhiều tài khoản liên quan).

---

1. Giao tác phẳng

Đặc điểm:

- T được coi là một khối giao tác không chia nhỏ được.
- Nếu có lỗi ở T1 hoặc T2, toàn bộ T sẽ bị rollback.

Ưu điểm:

- Đơn giản, dễ triển khai.
- Quản lý đồng nhất và ít phức tạp trong phục hồi.

Nhược điểm:

- Thiếu linh hoạt: nếu T1 thực hiện thành công nhưng T2 thất bại, kết quả của T1 cũng bị mất.
- Tài nguyên bị giữ lâu hơn → nguy cơ deadlock cao hơn.

---

2. Giao tác lồng ghép (Nested Transaction)

Đặc điểm:

- T bao gồm các giao tác con T1, T2 thực thi độc lập nhưng phụ thuộc vào giao tác cha T.
- Có thể rollback một giao tác con mà không hủy toàn bộ giao tác cha.

Ưu điểm:

- Tính linh hoạt cao hơn: rollback cục bộ mà không cần hủy cả giao tác cha.
- Tăng tính song song: T1 và T2 có thể chạy độc lập, tận dụng đa luồng.
- Phù hợp cho hệ thống lớn, phức tạp.

Nhược điểm:

- Phức tạp hơn trong quản lý rollback, commit và cách ly.
- Phải xử lý thứ tự commit giữa cha – con để duy trì nhất quán.

3. Cách đảm bảo tính cô lập (Isolation) và nguyên tử (Atomicity)

Cấp độ	Cách đảm bảo Isolation	Cách đảm bảo Atomicity
Giao tác cha T	Không cho T hoàn tất (commit) nếu có bất kỳ giao tác con nào chưa hoàn tất.	Chỉ khi tất cả giao tác con T1, T2 commit thành công → T mới commit
Giao tác con T1, T2	Thực hiện trên snapshot riêng hoặc thông qua lock cục bộ	Nếu một giao tác con thất bại → rollback cục bộ hoặc propagate lỗi lên giao tác cha

Thiết kế cơ chế giao tác phân tán cho hệ thống ngân hàng nhiều nút

1. Quy trình cam kết hai pha (Two-Phase Commit - 2PC)

Bối cảnh:

- Một giao tác chuyển tiền giữa hai chi nhánh A và B, được thực hiện trên hai nút phân tán.
- Giao dịch phải đảm bảo ACID, bất chấp lỗi mạng hoặc lỗi máy chủ.

Quy trình 2PC:

Pha 1: Chuẩn bị (Prepare Phase)

- Trình điều phối (Coordinator) gửi PREPARE đến tất cả các nút tham gia (Participants).
- Mỗi nút kiểm tra điều kiện (khóa tài nguyên, ghi log tạm) và trả lời:
    - VOTE-COMMIT nếu sẵn sàng.
    - VOTE-ABORT nếu xảy ra lỗi hoặc không thể cam kết.

Pha 2: Cam kết (Commit/Abort Phase)

- Nếu tất cả trả lời VOTE-COMMIT, Coordinator gửi GLOBAL-COMMIT.
- Nếu có ít nhất một VOTE-ABORT, gửi GLOBAL-ABORT.
- Các nút cập nhật trạng thái và ghi log vĩnh viễn.

. Đánh giá nguy cơ "khóa chết phân tán" (Distributed Deadlock)

Nguyên nhân:

- Các nút giữ tài nguyên chờ lẫn nhau trong khi chưa nhận được lệnh commit → deadlock giữa các nút.
- Trong 2PC, tài nguyên có thể bị giữ lâu do chờ phiếu từ các nút khác.

Biện pháp giảm thiểu:

- Giới hạn thời gian giữ khóa (timeout): nếu quá thời gian, nút chủ động rollback.
- Giao thức timeout + phát hiện vòng chờ (wait-for graph).
- Dùng giao thức 3PC (Three-Phase Commit) để loại bỏ trạng thái chờ không xác định (blocking).

Cách xử lý khi một nút tham gia cam kết bị lỗi giữa chừng

Tình huống:

- Một nút bị mất kết nối hoặc crash sau khi gửi VOTE-COMMIT nhưng chưa kịp nhận GLOBAL-COMMIT.

Giải pháp:

- Ghi log trạng thái trước khi phản hồi (đảm bảo durability).
- Khi khởi động lại:
  - Dò log để biết đã gửi VOTE-COMMIT, nhưng chưa commit.
  - Hỏi Coordinator để biết giao dịch cuối cùng đã được commit hay abort.
- Nếu Coordinator cũng lỗi → chờ đến khi có phản hồi hoặc rollback tạm thời sau timeout.

Câu 3:

Ba cách tiếp cận chính trong điều khiển tương tranh

Trong hệ quản trị cơ sở dữ liệu, điều khiển tương tranh (Concurrency Control) nhằm đảm bảo rằng các giao tác đồng thời không làm mất tính nhất quán của dữ liệu. Có ba phương pháp chính:

(1) Điều khiển bằng khóa – Pessimistic Concurrency Control

Đặc điểm cơ bản:

- Sử dụng các khóa (locks) để ngăn các giao tác khác truy cập cùng lúc vào dữ liệu đang được sử dụng.
- Hai loại khóa phổ biến:
    - Khóa chia sẻ (Shared Lock – S): cho phép đọc nhưng không ghi.
    - Khóa độc quyền (Exclusive Lock – X): cho phép ghi, chặn các thao tác khác.

Cơ chế hoạt động:

- Giao tác phải giành được khóa trước khi thực hiện đọc/ghi.
- Giải phóng khóa sau khi commit hoặc rollback.

Ưu điểm:

- Tránh xung đột dữ liệu hiệu quả.

Nhược điểm:

- Dễ xảy ra deadlock và đói tài nguyên.
- Làm giảm mức độ song song trong hệ thống.

---

(2) Điều khiển lạc quan – Optimistic Concurrency Control (OCC)

Đặc điểm cơ bản:

- Không dùng khóa trong quá trình thực hiện giao tác.
- Dữ liệu được xử lý lạc quan, kiểm tra xung đột chỉ khi giao tác kết thúc (commit).

Gồm 3 pha chính:

1. Giai đoạn đọc: giao tác đọc và xử lý dữ liệu cục bộ.
2. Giai đoạn xác nhận (validation): kiểm tra xem dữ liệu có bị thay đổi bởi giao tác khác không.

3. Giai đoạn ghi (write): nếu không có xung đột, cập nhật chính thức.

Ưu điểm:

- Tối ưu cho các hệ thống ít xung đột, độ song song cao.
- Tránh được deadlock.

Nhược điểm:

- Có thể phải rollback nhiều lần nếu xung đột thường xuyên.
- Không phù hợp với hệ thống có tần suất cập nhật cao.

---

(3) Điều khiển dựa trên nhãn thời gian – Timestamp-based Concurrency Control

Đặc điểm cơ bản:

- Mỗi giao tác khi bắt đầu sẽ được gán một dấu thời gian (timestamp) duy nhất.
- Quy tắc thực thi dựa trên thứ tự dấu thời gian: giao tác cũ hơn phải được thực hiện trước giao tác mới hơn.

Cơ chế:

- Với mỗi dữ liệu, hệ thống lưu:
  - Thời gian đọc gần nhất (read timestamp)
  - Thời gian ghi gần nhất (write timestamp)
- Khi giao tác đọc/ghi, so sánh timestamp để quyết định cho phép hay rollback.

Ưu điểm:

- Tránh deadlock vì không dùng khóa.
- Tự động đảm bảo thứ tự tuần tự hóa.

Nhược điểm:

- Rollback thường xuyên nếu lịch truy cập không đúng thứ tự.
- Phức tạp khi xử lý xung đột đọc/ghi ngược thứ tự thời gian.

2. Khái niệm "Tương đương tuần tự" – Serializability

Định nghĩa:

Một lịch giao tác (schedule) được gọi là tương đương tuần tự (serializable) nếu kết quả của nó tương đương với một lịch thực hiện tuần tự (từng giao tác một, không xen kẽ), và vẫn đảm bảo nhất quán dữ liệu.

Tại sao chỉ áp dụng luật xung đột đọc/ghi là chưa đủ:

Luật xung đột (conflict rules) chỉ kiểm tra các cặp thao tác đọc – ghi, ghi – đọc, ghi – ghi trên cùng một dữ liệu và khác giao tác. Tuy nhiên:

- Có những lịch không gây xung đột rõ ràng nhưng vẫn cho ra kết quả sai do đọc dữ liệu trung gian hoặc chưa commit.
- Ví dụ: phi tuần tự nhưng không gây xung đột → vẫn vi phạm tính nhất quán.

→ Do đó, cần kiểm tra tính tương đương tuần tự logic toàn cục, chứ không chỉ dựa trên xung đột.

Làm thế nào để đảm bảo lịch là tương đương tuần tự:

Có hai cách tiếp cận phổ biến:

(1) Kiểm tra đồ thị tuần tự hóa (Serialization Graph – SG):

- Tạo đồ thị các giao tác.
- Mỗi cạnh từ Ti → Tj nếu Ti thực hiện thao tác xung đột trước Tj.
- Nếu đồ thị không có chu trình, lịch là conflict-serializable.

(2) Dùng giao thức kiểm soát tương tranh đảm bảo sẵn tính tuần tự:

- Strict 2PL (Two-Phase Locking): đảm bảo mọi lịch tạo ra là tuần tự hóa được.
- Timestamp Ordering: sắp xếp giao tác theo timestamp để tránh xung đột sai thứ tự.

Xen kẽ thao tác giữa hai giao tác T và U

Cho:

- Giao tác T: T = read(x); write(y)
- Giao tác U: U = write(x); read(y)

---

a) Liệt kê tất cả các lịch khả dĩ (permutations)

Mỗi giao tác có 2 thao tác. Tổng số thao tác: 4
Cần liệt kê các lịch xen kẽ giữ nguyên thứ tự nội bộ của T và U:

- T: r(x) trước w(y)
- U: w(x) trước r(y)

Sử dụng phép hoán vị xen kẽ (interleaving) bảo toàn thứ tự nội bộ → có 6 lịch hợp lệ:

1. T1: r(x) T → w(y) T → w(x) U → r(y) U
2. T2: r(x) T → w(x) U → w(y) T → r(y) U
3. T3: r(x) T → w(x) U → r(y) U → w(y) T
4. T4: w(x) U → r(x) T → w(y) T → r(y) U
5. T5: w(x) U → r(x) T → r(y) U → w(y) T
6. T6: w(x) U → r(y) U → r(x) T → w(y) T

b) Lịch nào vi phạm tương đương tuần tự?

Dựa trên xung đột (conflict):

- r(x) (T) và w(x) (U) xung đột
- w(y) (T) và r(y) (U) xung đột

→ Nếu r(x) xảy ra trước w(x): thì T → U
→ Nếu r(y) xảy ra trước w(y): thì U → T

Nếu có chu trình giữa T và U trong đồ thị tuần tự hóa (SG), thì không serializable

Phân tích từng lịch:

Lịch	Thứ tự thao tác	Xung đột dẫn đến → đồ thị tuần tự hóa	Có chu trình?	Serializable?
T1	r(x) T → w(y) T → w(x) U → r(y) U	T → U qua r(x)/w(x), T → U qua w(y)/r(y)	Không	Có
T2	r(x) T → w(x) U → w(y) T → r(y) U	T → U (r/w x), T → U (w/r y)	Không	Có
T3	r(x) T → w(x) U → r(y) U → w(y) T	T → U (r/w x), U → T (r/w y)	Có chu trình	Không
T4	w(x) U → r(x) T → w(y) T → r(y) U	U → T (w/r x), T → U (w/r y)	Có chu trình	Không

T5	w(x) U → r(x) T → r(y) U → w(y) T	U → T (w/r x), U → T (r/w y)	Không	Có
T6	w(x) U → r(y) U → r(x) T → w(y) T	U → T (w/r x), U → T (r/w y)	Không	Có

Thứ tự cấp và nhả khóa trong phương pháp khóa hai pha nghiêm ngặt (Strict 2PL)

Nguyên tắc của Strict 2PL:

- Chỉ nhả tất cả các khóa khi giao tác kết thúc (commit/abort)
- Cấp khóa trước khi dùng tài nguyên

Thao tác và khóa cần thiết:

Giao tác	Thao tác	Loại khóa cần cấp
T	read(x)	S(x) (khóa chia sẻ)
T	write(y)	X(y) (khóa độc quyền)
U	write(x)	X(x)
U	read(y)	S(y)

Thứ tự cấp và nhả khóa (ví dụ cho lịch T1):

1. T: cấp S(x) → đọc x
2. T: cấp X(y) → ghi y
3. U: chờ X(x) (bị chặn do T đang giữ S(x))
4. T: commit → nhả S(x) và X(y)
5. U: cấp X(x) → ghi x
6. U: cấp S(y) → đọc y
7. U: commit → nhả khóa

→ Thứ tự khóa:

- T: lock_S(x), lock_X(y) → [commit] → unlock(x, y)
- U: lock_X(x), lock_S(y) → [commit] → unlock(x, y)

So sánh Pessimistic Locking và Optimistic Concurrency Control (OCC)

Trong kịch bản: hệ thống chấm điểm trực tuyến (online grading)

- Đặc điểm hệ thống:

- Rất nhiều người đọc: giáo viên, trợ giảng xem điểm.
- Cập nhật (ghi điểm) thỉnh thoảng xảy ra (ít ghi, nhiều đọc).

Pessimistic Locking

Ưu điểm:

- Tránh xung đột hiệu quả, đảm bảo an toàn dữ liệu khi ghi.

Nhược điểm:

- Hiệu năng thấp với nhiều người đọc: vì khóa có thể chặn đọc.
- Nguy cơ deadlock và starvation nếu nhiều người cố cập nhật cùng lúc.
- Không thân thiện với truy vấn chỉ đọc (tốn chi phí quản lý khóa không cần thiết).

Optimistic Concurrency Control (OCC)

Ưu điểm:

- Hiệu năng cao khi đọc nhiều vì không cần khóa.
- Không có deadlock (không giữ khóa).
- Thích hợp với mô hình "nhiều người đọc, ít người ghi".

Nhược điểm:

- Cần rollback khi ghi bị xung đột (nhưng tần suất xung đột thấp trong hệ thống grading nên chấp nhận được).
- Phân tích hiệu năng & nguy cơ:

Tiêu chí	Pessimistic Locking	Optimistic Concurrency Control
Hiệu năng khi đọc nhiều	Thấp (vì cần cấp/giải khóa)	Cao (không cần khóa khi đọc)
Nguy cơ deadlock	Có	Không
Phù hợp khi ghi nhiều	Có	Không (nhiều rollback)
Phù hợp khi ghi ít	Không hiệu quả	Phù hợp

Đề xuất cơ chế phân vùng dữ liệu và phương pháp điều khiển tương tranh

1. Phân vùng dữ liệu

Chia dữ liệu ngân hàng theo tính chất truy cập và chức năng:

Phân vùng	Ví dụ dữ liệu	Tần suất truy cập	Loại giao dịch	Phương pháp đề xuất
Vùng 1: Tài khoản	Số dư, thông tin tài khoản	Ghi thường xuyên	Chuyển tiền, rút	Khóa nghiêm ngặt (pessimistic)
Vùng 2: Lịch sử GD	Nhật ký giao dịch, log chỉ ghi	Đọc nhiều, ghi ít	Truy vấn báo cáo	Lạc quan (OCC)
Vùng 3: Báo cáo tổng	Tổng kết ngày, báo cáo phân tích	Đọc thường xuyên	Đọc, thống kê	Nhãn thời gian (timestamp)

2. Giải thích lựa chọn phương pháp

- Vùng 1 – Khóa nghiêm ngặt:
  Vì liên quan trực tiếp đến số dư, cần đảm bảo an toàn tuyệt đối, tránh mất cập nhật khi nhiều giao dịch đồng thời thực hiện rút/chuyển tiền.
- Vùng 2 – OCC:
  Lịch sử giao dịch hiếm khi bị cập nhật, chủ yếu thêm mới. Việc rollback khi xảy ra xung đột là chấp nhận được. OCC tăng hiệu suất khi nhiều truy vấn đọc.
- Vùng 3 – Timestamp:
  Các báo cáo phân tích không cần nhất quán tuyệt đối tại thời điểm thực hiện. Nhãn thời gian cho phép truy cập song song hiệu quả, dễ tuần tự hóa.

II. Quy trình xử lý khi gặp xung đột

1. Vùng sử dụng khóa (Vùng 1 – tài khoản):

- Khi xung đột:
  - Nếu khóa đang bị giữ: giao dịch chờ theo FIFO hoặc có giới hạn timeout.
  - Nếu vượt timeout → rollback hoặc abort.
- Cơ chế hỗ trợ:
  - Strict Two-Phase Locking (Strict 2PL) để đảm bảo serializability.
  - Deadlock detection hoặc prevention (dựa trên wait-for graph hoặc timestamp).

2. Vùng sử dụng OCC (Vùng 2 – lịch sử):

- Khi xung đột (tại giai đoạn commit):
  - So sánh phiên bản dữ liệu đọc với phiên bản hiện tại.
  - Nếu có thay đổi → rollback giao dịch đó.
  - Nếu không → thực hiện ghi.
- Ưu tiên:
  - Có thể dùng chiến lược backoff hoặc ưu tiên giao dịch cũ hơn.

3. Vùng sử dụng nhãn thời gian (Vùng 3 – báo cáo):

- Khi xung đột thao tác (ghi sau đọc...):
  - Vi phạm thứ tự thời gian → giao dịch bị rollback.
- Không có chờ, vì không dùng khóa.
- Cần quản lý timestamp hợp lý để tránh rollback lan rộng.

Đánh giá ưu – nhược điểm của giải pháp kết hợp (hybrid)

Tiêu chí	Hybrid (kết hợp)	Dùng 1 phương pháp duy nhất
Hiệu năng	Tối ưu cho từng loại dữ liệu và giao dịch	Không phù hợp khi xử lý không đồng đều
Khả năng mở rộng	Dễ điều chỉnh từng vùng độc lập	Dễ bị nghẽn cổ chai nếu dùng khóa toàn hệ thống
Đảm bảo nhất quán	Dữ liệu nhạy cảm được bảo vệ kỹ (bằng khóa)	OCC và timestamp không đủ mạnh khi ghi nhiều
Xử lý phức tạp	Phức tạp hơn khi triển khai và bảo trì	Đơn giản nếu chỉ dùng 1 phương pháp
Độ linh hoạt	Cao – thích nghi từng loại truy cập	Kém linh hoạt khi nhu cầu thay đổi