در درس پایگاه داده، قبلاً گفتیم که یک پایگاه داده با معماری ANSI توسط DBMS مدیریت می‌شود.

همه تعاریف و موضوعات در درس پایگاه داده از نگاه یک کاربر خارجی بوده که کاربر چه سرویس‌هایی را دریافت می‌کند و اینکه چگونه باید برای یک Application با یک پایگاه داده طراحی کرد.

براساس این نگاه از بیرون همه سرویس‌ها و تعاریف انجام شد. استانداردسازی‌ها انجام شد ...

در این درس نگاه را می‌خواهیم عوض کنیم و از داخل به یک سیستم پایگاه داده نگاه کنیم. حالا بررسی می‌کنیم که سرویس‌ها چگونه ارائه (ایجاد) می‌شوند. واقعاً ماشینی چگونه توسط سیستم پایگاه داده پشتیبانی می‌شوند، چگونه همروندی فراهم می‌شود؟ و ...

یک سیستم پایگاه داده را به ۴ (۳ یا) لایه تقسیم می‌کنیم:

Core (هسته) شامل: Disk Manager، Time Manager، Mem. Manager و ...

کنترل محیط عملیاتی، شامل کنترل امنیت، همروندی، ترمیم و ... و

همه این بخش‌ها جزء برنامه‌ها و اجزای کارکردی اصلی سیستم پایگاه داده می‌شناسیم

در لایه آخر هم App‌ها و ابزارهای تولید قرار می‌گیرند مثل Power Designer، ODBC و ...

* وقتی ما می‌خواهیم از دیدگاه داخلی صحبت کنیم روی کدام لایه‌ها صحبت کنیم؟

Core به کار تعامل با OS را انجام می‌دهد و ارتباط مستقیم با OS دارد. مثلاً اوراق یک

Subject: Date:

دارد که اگر مستقیم روی یک پاشینم مروک OS نصب شود باز هم کار می کند embeded OS

پس ما برای صحبت از دید داخلی روی DBMS یعنی کنترل محیط عملیاتی می رویم - روی جنبا از Functionality های

محیط عملیاتی کنترل می کنیم:

- کنترل همروندی Concurrency Control	" کنترل محیط عملیاتی
- ترمیم Recovery	- چرا؟ ⇐
- امنیت Security	- چگونه؟
- جامعیت (یکپارچگی) Integrity	- خودش پیاده سازی؟
	- تعریف و تقاضا

یکسری موضوعات دیگر هم به عنوان Advanced Topic مطرح خواهد شد (کتاب Date)

* منابع : هر کتابی که شامل این موضوعات باشد (و هر Paper)
Advanced Topic را از کتاب Date می خوانیم

* ارزیابی : امتحان ۱۶ نمره
هر کدام از آن ۴-۵ Functionality های از کنترل محیط عملیاتی
مقاله ۴ نمره : درباره هر موضوع دلخواه در حوزه پایگاه داده های متمرکز Central Data Base

زمان تحویل مقاله روز امتحان است

تحویل دادنی : بر روی CD ←
۱) منابع (۱۰-۱۵ منبع تحقیق) بر فولدر به نام Reference
۲) کد فایل Word (مقاله ۸-۱۲ صفحه)
۳) Power Point از گزارش

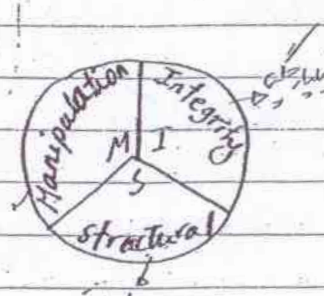
پایگاه داده: مجموعه ای است از داده های ذخیره شده مبتنی بر یک ساختار مشخص (مبتنی بر یک Data Structure مشخص) به صورت صوری (اتلاف با لاترین سطح) دارای حداقل افزونگی تحت یک نام مشخص و مدیریت یک سیستم متمرکز (مدیریت متمرکز) مورد استفاده یک یا چند کاربر به صورت مشترک و همزمان

ممکن است یک سیستم توزیع شده باشد اما مدیریت متمرکز است

اگر به صورت مشترک استفاده شود به صورت جزئی هم می توان استفاده شود اما عکس آن لزوماً صحیح نیست

ساختار مشخص: (ساختار پایگاه داده)
- ✓ سلسله مراتبی (بر حسب ارتباط Parent و Child)
- ✓ شبکه ای
- ✓ رابطه ای (در مباحث کاربردی گفتیم Table)

در مدل داده ۳ چیز خیلی مهم است



Manipulation: دستکاری داده ها به صورت (بهره برداری)

به صورت صوری:

✓ مدل ANSI:
- سطح خارجی
- سطح ادراکی
- سطح داخلی
- سطح فیزیکی

Subject: ______ Date: ______

افزونگی: به دو نوع است ← ذاتی و تلفیقی ← براساس روش طراحی انتخاب می‌شود

(روش مدل رابطه‌ای انتخاب شد و یکی از نرمال‌سازی و فرم‌ها را داشتیم)

مورد استفاده چند کاربره به صورت مشترک و همزمان ← موضوع درس ماست!

مسائل با زبان‌های اوراکل باید تکالیف جدید بنویسیم

کنترل همروندی و ترمیم ← این دو بخش را با هم جلو می‌بریم

کنترل همروندی: مجموعه فعالیت‌هایی است جهت هماهنگ کردن اعمال تراکنش‌هایی که به صورت همزمان اجرا می‌شوند و با داده‌های مشترک کار می‌کنند و احتمالاً با یکدیگر تداخل دارند (به صورت بالقوه)

تراکنش: مجموعه‌ای از دستورات که با پایگاه داده‌ها ← مجموعه‌ای از یک یا چند برنامه

داده مشترک: داده‌ای که همزمان چند تراکنش از آن استفاده می‌کنند

کارها: عملاً فقط دو عمل Read و Write وجود دارد:

تداخل (Interference): رخدادهای نامطلوب (نادرستی)

جایی که تراکنش‌ها در یک واحد زمانی می‌خواهند از داده مشترک استفاده کنند. شرط لازم (و نه کافی) برای کنترل همروندی است. شرط کافی این است که باعث تداخل شود

آیا چیزی مشخص می‌کند و چگونه همه چیز را انجام می‌دهد که تداخل رخ ندهد:

اگر کنترل همروندی نباشد ممکن است تداخل رخ دهد.

ترمیم: تعیین حد داده‌های سازگار در پایگاه داده در هر گونه (حفاظت از داده‌های سازگار در برابر انواع خطاها)

✓ مثلاً اگر برق برود پایگاه داده‌ها را به آخرین حالت سازگار برگرداند

خصوصیات تراکنش: ACID

تجزیه‌ناپذیری - خاصیت اتمی

A) Atomicity) : تراکنش یا انجام نمی‌شود یا کامل انجام می‌شود.

یکپارچگی - سازگاری

C (Consistancy) : تراکنش‌ها پایگاه داده را از یک حالت سازگار به حالت سازگار دیگر ببرد.

I (Isolation) : یک تراکنش باید تغییر و نقض بقیه نشود. تراکنش‌ها از هم ایزوله هستند

D (Durability) : اثرات یک تراکنش ماندگار شود.

دو خود داخل با خود خارج با یک زبان مشترک به نام Transaction با هم حرف می‌زنند.

هدف اصلی کنترل همروندی و ترمیم این است که تراکنش‌ها به صورت اتمیک انجام شود یعنی:

(۱) بدون اینکه اثرات نامطلوب با بقیه داشته باشند انجام شود

(۲) یک تراکنش به صورت نرمال خاتمه یابد و تمام اثراتش ماندگار شود یا هیچ اثری نداشته باشد.

✓ اگر یک تراکنش اتمیک انجام شود چه خصوصیت ACID را دارد.

پایگاه داده از دیدگاه داخلی: مجموعه‌ای است از اقلام داده‌ای (Data Item) دارای نام مشخص

اقلام داده‌ای دارای سطوح دانه‌بندی مختلف هستند: فیلد - رکورد - فایل - بلاک - کل دیتابیس

سیستم پایگاه داده : مجموعه ای است از یک افزارها و نرم افزارهایی که امکان دسترسی به (داده ها) را انجام عملیات روی آن ها را فراهم می کند یعنی امکان انجام و اجرای تراکنش ها را می دهد .

دستورات موجود در یک تراکنش : -
- Read
- Write
- Commit
- Abort

(Commit و Abort:) هر تراکنش حتماً یکی از این دو را دارد

- Start : هر تراکنش با این دستور شروع می شود .

[Abort ← Commit]

زمانی که n تراکنش داریم و Commit ها و Abort ها را انجام می دهیم باید دقیقاً مشخص شود هر A/C به کدام تراکنش تعلق دارد .

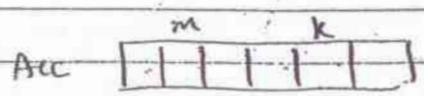
Start که Id منحصر به فردی را برای تراکنش می گیرد و آن Id را (انتهای) همه دستورات می چسبانیم تا مشخص شود هر دستور برای کدام تراکنش است .

تا اینجا در مورد صرف تراکنش گفتیم ! حال به سراغ نحو تراکنش می رویم

نحو تراکنش

مثال) از حساب شماره m به حساب شماره k ، V ریال انتقال دهید .

فرض می کنیم موجودی حساب ها در آرایه هستند



Subject: Date:

```
Procedure Transfer
Start
    output ("شماره حساب‌های مبدأ و مقصد و مقدار پول را وارد کنید")
    Input (m, k, v)
    Temp = Read (Acc(m))
    if Temp < v then
        output ("موجودی برای انتقال کافی نیست")
        Abort
    End if
    write (Acc(m), Temp - v)
    Temp2 = Read (Acc(k))
    write (Acc(k), Temp2 + v)
    Commit
End Procedure Transfer
```

ماشین داخلی از تراکنش بالا برای خودش یک تراکنش معادل می‌سازد همان طور که ما نیز با مقادیر کاری نداریم در اینجا هم اینکه چه عددی نوشته یا خوانده می‌شود مهم نیست.

Input و Output هم برایش مهم نیست چون اصل عملیاتش مهم است.

ACC(M) در ماشین به نام x است یعنی یک Data Item است که یک Unique Id می‌گیرد

مثلاً 11.

Subject: ______ Date: ______

$R_{11}(X)\ a_{11}$

$R_{11}(X)\ W_{11}(X)\ R_{11}(Y)$

$W_{11}(Y)\ C_{11}$

ادم درون با جدول بالا کار دارد . در زمان اجرا ساخته می‌شود

با این مثال ارتباط درون و برون را نشان دادیم . به جدول از تراکنش ساختیم و Map کردیم :

ال) یک تراکنش به نام Deposit بنویسید (انتقال وجه)

موجودی حساب را می‌خوانیم ، Update می‌کنیم و Commit می‌کنیم

R(X) W(X) C

به طور انتزاعی بر می‌گردیم ، به آن دست نمی‌زنیم ، چنین عمل ریاضی روی آن انجام دهیم و ... همه از جدول بالا پیروی می‌کند

با برداشت از حساب به جای + ، - می‌شود باز هم جدول بالا است

Subject: 2nd Session Date: 91 , 7 ,15

می‌خواهیم درباره معنا و مفهوم این دستورات صحبت کنیم و اینکه پایگاه داده چه برخوردی با آنها دارد.

Start : id یعنی برای تراکنش می‌گیرد و این id یعنی (انتهای) همه دستورات تراکنش اضافه می‌شود

تا کنترل همروندی بتواند که هر دستور مربوط به کدام تراکنش است و توالی و تقدم و تأخر را مشخص کند

Read , Write : توضیح دادیم که عملیات روی پایگاه داده است. write یک اثری روی پایگاه داده

دارد حالا از هر نوع پس اثر آن باید ثبت شود وقتی ثبت کرد باید Acknowledge یا تصدیق

آن را بفرستد. تصدیق عمل Read هم باید فرستاده شود که همان مقداری است که برمی‌گرداند.

Commit : عملی است که وقتی سیستم پایگاه داده انجام داد یعنی تراکنش که انجام شده و همه Read

و Write ها درست بوده و باید اثرش ماندگار شود، یعنی کارهایی که می‌خواستیم انجام شده

✓ زمانی Commit می‌دهیم که مطمئن بشود همه چیز درست بوده است.

Abort : هیچ اثری نباشد

منابع صادر کردن Abort : خود تراکنش (ابطال کند و کاربر) ، سیستم پایگاه داده

اگر تراکنش گفت Abort حتماً سیستم پایگاه داده آن را تصدیق می‌کند ولی ممکن است تراکنش

بگوید Commit ولی سیستم پایگاه داده آن را قبول نکند. سیستم پایگاه داده باید Commit ها و

Abort ها را تصدیق کند و در صورت تراکنش همه چیز را می‌بیند و چک می‌کند

✓ اگر بین Abort و Commit قرار گیرد، Abort می‌کند برای حفظ سازگاری چون هیچ چیز مهم‌تر از آن نیست

Subject: Date:

* سیستم با نگاه داده در تصدیق Commit به وقت حسابداره رفتار می‌کند!
البته تا حد امکان که Abort هم جلوگیری می‌کند

* اگر چه Read و Write های تراکنش های مختلف (هم‌زمان) بوده و در آخر Commit یا Abort شوند وظیفه کنترل همروندی است

تداخل نتیجه ناظم شدن دستور R است

دو راه حل برای عدم تداخل:

1) تراکنش‌ها به صورت پی در پی اجرا شوند — اجرای پی در پی
2) تراکنش‌ها به صورت همروندی (ناظم) اجرا شوند و نتیجه درست باشد — اجرای پی در پی پذیر

Serializable

هدف کنترل همروندی گزینه ۲ است. به گونه‌ای ناظم شدن‌ها اتفاق بیفتد به اجرا‌ها پی در پی پذیر باشند

* می‌دانیم که Commit و Abort انتهای کار هستند (انتهای تراکنش) موجب ناسازگاری شود

اگر کنترل همروندی نباشد چه می‌شود؟ وقتی تداخل رخ می‌دهد چه اتفاقی می‌افتد؟ اثرات تراکنش‌ها

چیست؟ تأثیرگذاری چگونه است؟

۱) حافظه داده مشترک: تمرکز ما فقط روی این گزینه است (حوزه کار ما)

۲) Calling، یکدیگر را صدا بزنند Call By Value، Call By Reference و ...

۳) Messaging:

5

Subject: Date:

تداخل منجر به ناسازگاری می‌شود (اگر کنترل همروندی نباشد)

تداخل انواع و اقسام متفاوت دارد:

(۱) بهنگام سازی گم شده Lost Update

(۲) بازیابی ناسازگار Incasistant Retrival

(۳) شبح Phantom

:

بهنگام سازی گم شده

$T_1 = R_1(x)\ W_1(x)\ C_1$ — $T_1 : x \rightarrow x^2 + 10 = 12$

$T_2 = R_2(x)\ W_2(x)\ C_2$ — $T_2 : x \rightarrow x^2 + 5 = 7 \quad x = 2$ مقدار اولیه

نتیجه اجرای بی‌درپی درست $x = 17$ خواهد بود

T_1	T_2
	$R_2(x)$
$R_1(x)$	
	$W_2(x)$
$W_1(x)$	
	C_2
C_1	

$\Rightarrow R_2(x)\ R_1(x)\ W_2(x)\ W_1(x)\ C_2\ C_1$

x
$1 \rightarrow 7 \rightarrow 12$

اگر تراکنش T_2 وجود ندارد نتیجه نامطلوب

بهنگام سازی اتفاق افتاده است و گم شده

اگر به صورت Sequential (ترتیبی) اجرا کنیم نتیجه مطلوب بدست می‌آید ولی وقتی قاطی اجرا شوند نتیجه نادرست دارد. ترانزکشن ۱ و ۲ اثرات یکدیگر را اصلاً نمی‌بینند

کنترل همروندی جلوی این نحوه اجرا را می‌گیرد و نحوه‌ی صحیح اجرا را می‌دهد:

$R_2(x)\ W_2(x)\ R_1(x)\ W_1(x)\ C_2\ C_1$

تقدم و تأخیری را مشخص می‌کند که ترانزکشن‌ها با هم به گونه‌ای قاطی شوند که نتیجه‌ی درست بدهد

این قاطی کردن بالا را نیز سیستم ممکن است نحوه‌ی اجرای دیگری نیز باشد که باز هم نتیجه‌ی درست بدهد

بازیابی ناسازگار

فرض کنیم ۲ ترانزکشن داریم: x و y

$\begin{cases} x = 105 \\ y = 2 \end{cases}$ موجودی اولیه

T_3: انتقال پول $x \xrightarrow{100} y$

T_5: نمایش موجودی حساب $x =$ $y =$ Totally $=$

$T_3: R_3(x)\ W_3(x)\ R_3(y)\ W_3(y)\ C_3$

$T_5: R_5(x)\ R_5(y)\ C_5$

T_3	T_5
$R_3(x)$	
$W_3(x)$	
	$R_5(x)$ → (5)
	$R_5(y)$ → (2)
$R_3(y)$	
$W_3(y)$	
	C_5
C_3	

کل موجودی = 7 !?!! ← غلط

$\rightarrow R_3(x)\ W_3(x)\ R_5(x)\ R_5(y)\ R_3(y)\ W_3(y)\ C_5\ C_3$

x	y
$105 \rightarrow 5$	$2 \rightarrow 102$

Subject: Date:

کل موجودی 107 است ما فقط 7 را می‌بینیم. اشکال انجام شد اما غلط خواندیم

$X=5$

$Y=2$

خواندن ناسازگار !

کل حساب $=7$

$X=105$ $Y=2$ $Tot=107$ نتیجه درست 1

$X=5$ $Y=102$ $Tot=107$ نتیجه درست ۲

T_3	T_5
$R_3(x)$	
	$R_5(x)$
$W_3(x)$	

به تغییر توجه می‌دهیم تا نتیجه‌ی درست حاصل شود

چون غلط شود تا نتیجه‌ی درست بدهد ؟

در این مثال در ابتدا تراکنش ۵ تنها بخشی از اثر تراکنش ۳ را دید

تراکنش ۵ باید اثرات تغییر را کامل ببیند *

اجرای نادرست باعث از بین رفتن سازگاری می‌شود

چگونه ناسازگاری که به این ختم می‌شود!

۲ تا تراکنش داریم: $x=10$

T_1	T_2
$R_1(x)$	
$W_1(x)$	
	$R_2(x)$
	$W_2(x)$
	C_2
A_1	

$T_1 = x \rightarrow x^2$

$T_2 = x \rightarrow 2x$

$10 \Rightarrow 100 \Rightarrow 200 \rightarrow 10$

Abort : هیچ اثری نداشته باشد به حالت قبل برگردد (10)

Commit : اثرش ماندگار شود که نشد یا اثر ماندگار شد 200 به 10 تبدیل شد و این با اثرات Commit در تضاد است

اگر هم 200 بماند با اثرات Abort در تضاد است

Subject: Date:

سوال: چرا خواص تراکنش، اثرات تراکنش‌های دیگر را کامل دور نگه نمی‌دارد؟ تراکنش هر چند که درست است اما ناسازگاری رخ داد

باید راه حل پیدا کنیم.

* بدون اینکه اطمینان پیدا کند Commit دارد ناسازگاری ایجاد شد. سیستم پایگاه داده باید در اول

Commit به شدت مراقبت کند

اثرات تراکنش

۱. مقدار داده را بازنویسی می‌کند (اثر write) : اثر گذاشتن روی سایر تراکنش‌ها

۲. مقدارهایی را از سایر تراکنش‌ها می‌خواند (اثر Read) : اثرپذیرفتن از سایر تراکنش‌ها

وقتی تراکنشی Abort می‌شود هیچ اثری نباید داشته باشد یعنی:

۱. هر مقدار داده‌ای که بازنویسی شده مقدارش به مقدار قبلی بازگردد

۲. هر تراکنشی که اثر پذیرفته باید برگردانده شود

با ساقط کردن یک تراکنش ممکن است یک یا چندین تراکنش دیگر ساقط شود

↓

پدیده انتشار سقط Cascading Abort Phenomenal

راه حل مشکل اول: Undo کردن، یعنی از مقدار اولیه X نگه داریم تصویر قبلی Before Image

منشأ مشکل: write

راه حل مشکل دوم: منشأ مشکل Read . برای ... خوانده (خواندن از)

گوییم تراکنش T_i قلم داده x را از T_j خوانده است اگر:

۱) قبل از خواندن T_i، T_j، x را نوشته باشد (شرط لازم)

۲) قبل از خواندن T_i، T_j سقط (Abort) نکرده باشد (شرط کافی)

۳) به ازای هر تراکنش مثل T_k، اگر T_k پس از T_j روی x نوشت قبل از خواندن T_i، $T_k < T_i$ سقط شود

* Commit را به تعویق انداختیم وقتی نتیجه مشخص شد (Abort یا نه)، Commit یا Abort کردیم

اجرای ترمیم پذیر: اگر تراکنشی از تراکنشی را خواند، Commit آن منوط می‌شود به Commit کردن تراکنشی که از آن اثر گرفته است (هر تراکنشی زمانی Commit می‌کند که همه تراکنش‌هایی که از آنها خوانده است Commit کرده باشند.)

ترمیم پذیری و کنترل همروندی دو مولفه جدا هستند و هر کدام یک جنبه سازگاری را حفظ می‌کنند.

سازگاری علت وجود پدیده‌ی انتشار سقط رخ داد یعنی اتلاف منابع

کاری کنیم که تراکنش‌ها با هم سازگار انجام شوند ولی انتشار سقط رخ ندهد.

سه نوع اجرایی که تا الان یاد گرفتیم: اجرای پی در پی: ساده‌ترین نوع اجرا، قطعاً درست است

- اجرای پی در پی پذیر

- اجرای ترمیم پذیر

Subject: 3rd Session Date: 91/7/22

اجرای پی‌درپی: تراکنش‌ها پشت سر هم می‌بایست اجرا شوند. از لحاظ استفاده از منابع مناسب نیست.

اجرای پی‌درپی‌پذیر: هدف کنترل همروندی است. یعنی بدون تداخل هم به وجود بیاید چند طرز از آن را گفتیم.

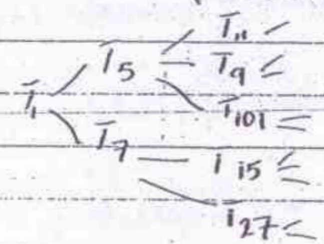
اجرای ترمیم‌پذیر: تضمین می‌کند به‌طوری از نابسامانی رخ ندهد (شرایط آن را گفتیم)

در یک سیستم واقعی، به محض شروع اجرای تراکنش‌ها و نوشتن یک داده که خواندن آن به وجود داده ایجاد می‌شود که

بیانگر ارتباط میان تراکنش‌ها است.

یک تراکنش هر چقدر داده را تغییر بدهد می‌تواند و می‌نویسد پس خطا رخ می‌دهد (این شرط را هم گذاردیم)

اگر یک تراکنش Abort شد بقیه‌ای که به آن وابسته هستند نیز A می‌کنند (انتشار لغو)



انتشار لغو پدیده‌ی مطلوبی نیست (به علت اتلاف منابع) و برای جلوگیری از آن باید تمام ارتباطات وجود داده را

داشته باشیم که منابع و هزینه می‌خواهد

راهکار: چرا به خاطر A شدن یک تراکنش بقیه تراکنش‌ها هم A شوند؟ اگر نخواهیم پدیده انتشار لغو رخ دهد

باید جلوی خواندن از را بگیریم (که در عمل ممکن نیست) ولی می‌توانیم بنویسیم که آن. اما از نظر زمان آن هم ... است

«یعنی خواندن فقط از تراکنش‌های نهایی شده (Commited) انجام شود»

به تعویق انداختن Read: همچنین صبر کن C شد بعد بخوان

Subject: Date:

اجراهای اجتناب کننده از انتشار سقوط Avoid Cascading Abort (ACA)

مثال) $R_1(x)\ W_1(x)\ R_2(x)\ -\ C_1\ R_2(x)$

چون هنوز C_1 رخ نداده و $R_2(x)$ از $W_1(x)$ تأثیر می‌پذیرد باید بعد از C_1 بیاید. پیروی انتشار سقوط رخ

نداده و لذا همروندی کاهش پیدا کرد.

مثال) $W_1(x)\ W_2(x)\ C_2\ A_1$ $\quad T_1: x \to 100$

$T_2: x \to 200$

$x: 5$

x

$5 \to 100 \to 200 \to 5$

اثر تراکنش T_2 از بین رفت و ماندگار نشد (با وجودی که C کرده بود)

اجرا ترمیم پذیر است چون خواندن از ندارد. اجرا ACA نیز هست چون خواندن از ندارد:

با مثال زیر: $W_1(x)\ W_2(x)\ a_1\ a_2$

چون هر کدام قبل از W یک کپی از x نگه می‌دارند $\quad x: 5 \to 100 \to 200 \to 5 \to 100$

(Before Image)

ناسازگاری پیش آمده. نقص از کجاست؟ بدلیل قلم داده مشترک است و مشکل از اعمال write است

داخل چیزی ننوشتیم که تکلیفش معلوم نیست. پس اینجا است وقتی بنویسیم که تکلیفش معلوم باشد.

علت ناسازگاری: نوشتن‌های کورکورانه (Blind Write) یعنی روی قلم داده‌ای نوشتیم که هنوز تکلیفش

روشن نشده

راهکار: write ها رو به تعویق بیندازیم (به تعویق انداختن W و R)

این اجرا اجرای محض (اکید) strict نام دارد.

اجرای محض strict

اجرایی هستند که در آن هر تراکنش تنها هنگامی داده‌ای را می‌خواند که اثر تراکنش نهایی شده باشد و تنها زمانی قلم داده‌ای را می‌نویسد که اثر تراکنش نهایی شده باشد.

وقتی اجرای محض رخ می‌دهد باز هم هم‌روندی کاهش می‌یابد چون شرایط سخت‌تر می‌شود.

تا الان فقط در مورد کنترل هم‌روندی صحبت کردیم حالا به سراغ ترمیم می‌رویم

هر وقت اجرا سریال دیگر به کنترل هم‌روندی نیاز نیست

ترمیم: سیستم پایگاه داده باید مکانیزم‌هایی را داشته باشد برای ترمیم. تضمین ماندگاری داده‌های نهایی شده

نتیجه اجرایی که از کنترل هم‌روندی برگشته، اگر باید در پایگاه داده ذخیره و ماندگار شود، خطا شود، اگر اتفاقی دیگری بیافتد نتیجه را عوض کند، در این صورت باز نتیجه صحیح حفظ شود

پس هر اتفاقی افتاد باید آخرین حالت سازگار را حفظ کند

انواع مشکلات در پایگاه داده ممکن است رخ دهد

Media Failure (خطاهای رسانه): خطاهایی که در اثر آن‌ها رسانه‌های ذخیره‌سازی با اطلاعات از بین می‌رود. مثلاً Hard می‌سوزد و ... . در ترمیم باید مکانیزم‌هایی داشته باشیم Backup و نسخه‌های

Subject: Date:

داشته باشیم از Cool Backup یا Hot Backup مثل Raid گرفتن برای حفظ ترمیم و

با استفاده از SAN همه این روش‌ها دنبال ترمیم هستند.

System Failure (خطاهای سیستمی) خطاهایی که باعث می‌شوند تراکنش state داخلی خود را از

دست بدهد. هر برنامه داخل حافظه (Working Area) انجام می‌شود. ممکن است برق برود،

ویروسی شود، اشتباه محاسباتی، تقسیم بر صفر و ... رخ دهد و محتوای حافظه اصلی از دست برود و با مشکل مواجه

شود. به این خطاها خطاهای سیستمی می‌گوییم. مکانیزم‌هایی برای مقابله با آن‌ها وجود دارد. سیستم پایگاه داده

باید بپذیرد چیزهایی را برای خودش حفظ کند تا بتواند مقابله کند

چگونه مرجع اطلاعاتی را با پایگاه داده ذخیره خواهیم کرد؟

بحث پایگاه داده ترمیم است که در هر لحظه ممکن است اتفاقی بیفتد و باید ترمیم را انجام دهد. پس همیشه

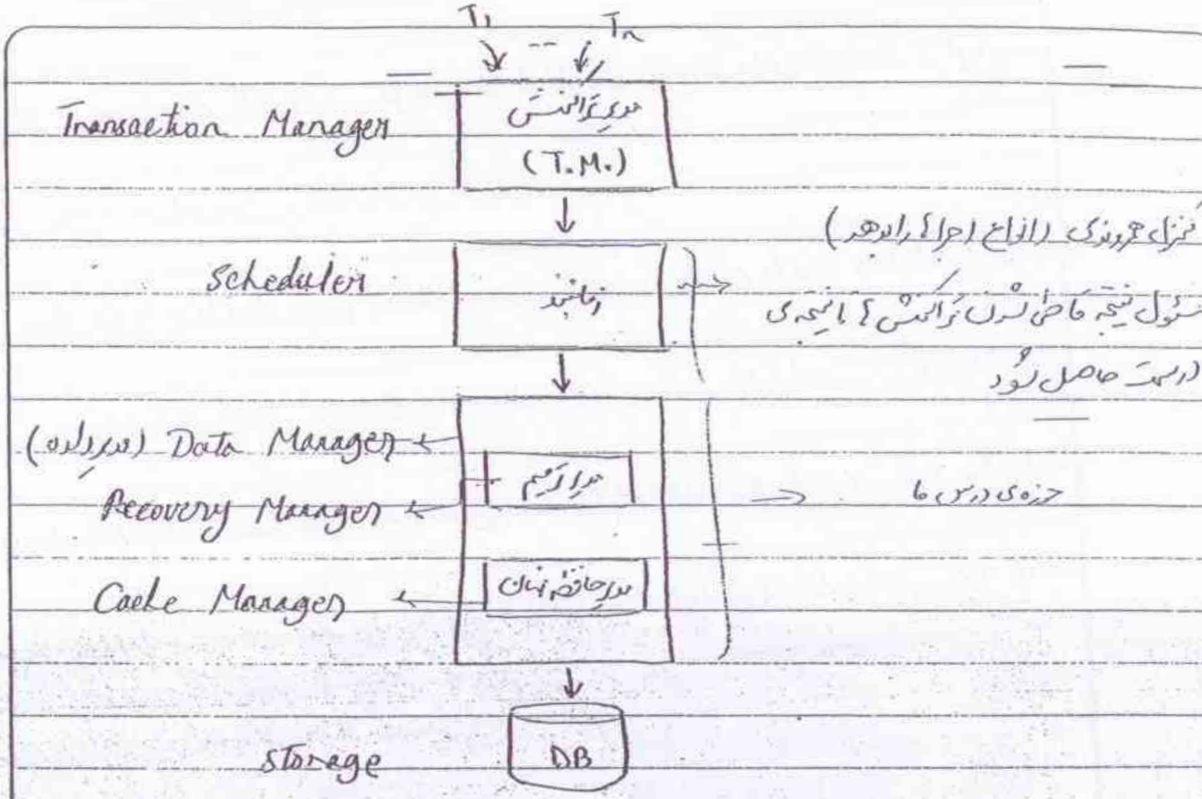
باید آماده رفتن به آخرین حالت سازگار باشد.

معماری سیستم پایگاه داده

معماری لایه‌ای برای مولفه‌ها که وظایف‌شان مشخص است

شرح وظایف می‌کنیم که هر وظیفه بر عهده کیست

Subject: Date:



پایگاه داده‌ای که از این معماری تبعیت کند می‌تواند از این مکانیزم‌ها استفاده کند

مدیر حافظه‌ی نهان: Cache را مدیریت می‌کند. مدیریت خواندن و نوشتن داده‌ها به حافظه‌ی ماندگار.

چون با محدودیت حافظه روبرو هستیم همه حافظه را نمی‌توان آورد پس با بهترین کارایی خواندن و نوشتن را انجام می‌دهد. الگوریتم‌های LIFO، FIFO و ... . دو دستور بیشتر ندارد:

Fetch(x): داده مورد نظر را در حافظه نهان برو از DB بیار

Flush(x): داده را داخل DB ببر

مدیر ترمیم: مقابله با انواع Failure ها (رسانه، سیستم)

علاوه بر دستورات R و W و C و A یک دستور دیگر به نام Restart داریم که این دستور اول

از زمانبند می خواهد که آنچه از خارج می آید (منابع خارجی)

زمانبند: انواع اجرا را تولید می کند به گونه ای دستورات را اجرا می کند تا سازگاری حفظ شود. هر دستوری که برای

زمانبند بیاید سه کار را می تواند انجام دهد: - بفرستد برای اجرا

- دستور را سقط کند که خود تراکنش لغو می شود

- انتظار wait بدهد برای اینکه برای برگشت دوران توالی دلخواه

مدیر تراکنش کل دستورات تراکنش را با هم به زمانبند نمی دهد.

مکانیزم الگوریتم وجود دارد که به زمانبند اجرای درست را می دهد.

زمانبند اجرای بی درنگ را برای مدیر ترمیم فراهم کند (یعنی است ACA و آگاه شود) را اتخاذ کند

تا سازگاری حفظ شود

مدیر تراکنش: برای تراکنش ها id تولید می کند و دستورات برای زمانبند ارسال می کند.

خواهیم دید که گاهی مدیر تراکنش ممکن است در پی اطلاعاتی را برای تعیین اجرای خاص برای

زمانبند فراهم کند. در پایگاه داده های توزیع شده مدیر تراکنش باید مشخص کند هر دستور کجا برود.

مرجع یک: مانیتور

Subject: Date:

ماژول‌ها [illegible] هر یک چنین نسخه وجود داشته باشد که کارایی بالا برود مثلاً دو تا مدیر داده داشته باشیم

زیاد باشند اگر تراکنش بخواهد تغییری بدهد ابتدا به الگوریتم [illegible] برای مدیر داده و هنگامی که Ack را از مدیر داده

گرفت دستورالعمل را اجرا کند اصطلاحاً Hand shaking می‌گوییم.

چرا ۲ تا مدیر داده؟ چون از تراکنش‌ها که هستند با هم ارتباطی ندارند و تراکنش‌ها با هم نیست و می‌توانند

در Data Manager های مختلف انجام شوند

بحث Threading از اینجا پیش می‌آید. n تا Thread داریم که به هم ربطی ندارند و ۲ تا مدیر داده

اینجا به درد می‌خورد.

هر وقت لازم است هماهنگ سازی رخ دهد Hand Shaking می‌کنیم.
(Synchronize)

در تراکنش‌ها که هستند هیچ وقت تمام نمی‌شوند مثل [illegible] ختم نمی‌شود و [illegible] منابع محدود

داریم که در اینجا دیگر Hand shaking معنا ندارد

برای پروسس‌هایی که مدام در حال اجرا هستند یک ناحیه بحرانی تعریف می‌کنیم

مانعت دو جانبه (Mutual Exclution) راه حل‌های مشخصی برای این اتفاق هستند.

شرایط:

- پایگاه داده متمرکز است.

- برای سادگی فرض کردیم هر تراکنش فقط یکبار می خواند و می نویسد.

- تنها ارتباط بین تراکنش ها قلم داده مشترک است.

- مدیر داده تضمین می کند که دستورات را اجرا و ترمیم کند.

- زمانبند اجرای پی در پی پذیر و حداقل ترمیم پذیر می دهد.

چگونه پی در پی پذیر باشد (با استفاده از گراف همروندی)؟

یک تئوری اساسی به نام تئوری پی در پی پذیری وجود دارد. از ابزار استفاده می کنیم به نام سابقه یا سناریو که یک مدل از اجراست.

اگر سابقه ای پی در پی پذیر باشد آنگاه اجرای حاصل از این سابقه پی در پی پذیر است.

سابقه متشکل از تعدادی تراکنش است به علاوه ترتیب اجرا. پس ارتباطی برای تراکنش ها توصیف می کند.

تراکنش ها و ارتباط آن ها سابقه را تشکیل می دهند.

تراکنش: دنباله ای از دستورات تراکنش (R و W و A و C) که هدف آن تراکنش پایگاه داده اصلاً هم منتهی به C

می شود یا A. تراکنش منتهی باید از جمله A و C در انتها هستند و روی یک قلم داده مشخص ترتیب خواندن و

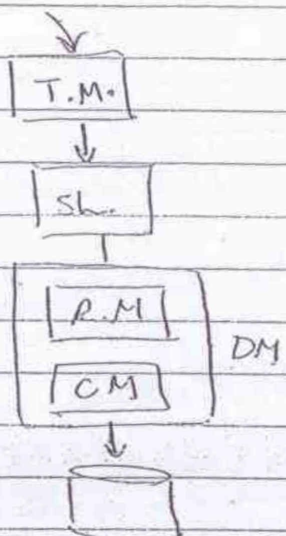
نوشتن مشخص شود.

Subject: _____ Date: _____

Subject: 4th Session Date: 91/7/29

یک معماری را به عنوان معماری پایه انتخاب کردیم و نام مجب... آنچه که گفتیم در چارچوب این معماری است.

این معماری یک معماری استاندارد است. در این معماری چهار مولفه اصلی داشتیم:



T.M.: به هر تراکنش یک Id یکتا اختصاص می‌داد

Sh: خاطر بروز دستورات به گونه‌ای به اجرا که پیچیدگی پیش نیاید

ترسیم نمودار را فراهم کند. Sh. برای این کار ابزاری

به نام wait داشت

موقعی که روی پیچیدگی‌ها بحث می‌کنیم از ابزاری به نام سابقه (سناریو / history) استفاده می‌کنیم که مجموعه‌ای از

تراکنش‌ها است.

تراکنش (Transaction):

۱- دنباله‌ای از دستورات {R, W, C, A}

۲- آخرین دستور یا C است یا A و ۳- تراکنش فقط یا C می‌کند یا A.

۴- روی یک قلم داده ترتیب خواندن و نوشتن مهم است (چون روی نتیجه تأثیرگذار است)

۵- توالی مشخص

(داشتن چند قلم داده‌ای که می‌توان با آن‌ها کار کرد برای سیستم پایگاه داده مهم نیست)

Subject: Date:

رابطه با ترتیب جزئی: $H(\Sigma, <)$

برای حداقل تعریف مرحل از تراکنش ارائه شده باید از عناصر فرمال و ریاضیاتی استفاده نمود

Σ: مجموعه دستورات

$<$: توالی (یا) دستورات روی تراکنش

چه رابطه ترتیب جزئی؟ نتیجه اجرای سه توالی زیر یکسان است

$R_1(x)\ R_1(y)\ W_1(z)\ C_1$

$R_1(y)\ R_1(x)\ W_1(z)\ C_1$

$R_1(x) \rightarrow W(z)\ C_1$
$R_1(y) \rightarrow$

یعنی لازم نیست رابطه بین هر دو عمل جزء با شد (شبکلی)

$R(\Sigma, <) \Longrightarrow T_i = (T_i, <_i)$

به صورت می نویسم

تعریف رسمی تراکنش: یک رابطه با ترتیب جزئی $\{T_i, <_i\}$ است که:

1) $T_i = \{R(x), W(x) \mid x \text{ is a data item}\} \cup \{C_i, a_i\}$

2) $C_i \in T_i$ iff $a_i \notin T_i$

3) if $a_i = t$ or $C_i = t$ then
for each instruction such as $p \in T : p <_i t$

4) if $W_i(x) \in T_i$ and $R_i(x) \in T_i$ then
$W_i(x) <_i R_i(x)$ OR $R_i(x) <_i W_i(x)$

۱۳۱

این تعریف با این فرض است:

۱) بدون آنکه خللی در کلیت مباحث وارد شود فرض می‌کنیم که هر تراکنش هر قلم داده را حداکثر یکبار می‌خواند یا می‌نویسد. (به منظور ساده سازی مدلینگ)

۲) فرض: هر مقداری که نوشته می‌شود تابعی است از مقادیر خوانده شده (با این فرض تقدم قرار داده‌ایم برای R و W؛ تابع از نوع داده) (حالا تابع را برو از BiZ در بیار چه)

* R(x), R(y) → W(Z) : Z تابعی از x, y است

R(x)
R(y) → W(Z) : Z تابعی است از y

ولی ما برای ساده سازی فرض ۲ را قرار داده‌ایم

مثلاً مثال اول را به صورت زیر بیان می‌کنیم:

* $T_i = \{R(x), R(y), W(Z), C\}$

$<_i = \{(R(x), W(Z)), (R(y), W(Z)), (W(Z), C), (R(x), C), (R(y), C)\}$

R(x) → C
R(y) → W(Z) → C

$<_j = \{(R(y), W(Z)), (R(x), C), (W(Z), C), (R(y), C)\}$

از این به بعد برای نمایش روابط از گراف استفاده می‌کنیم

سابقه history / سناریو Scenario

- مدل اجرای توام دستورات برای تراکنش‌ها است

- ارتباط بین تراکنش‌ها را نشان می‌دهد (ترتیب اجرای دستورات بین تراکنش‌ها)

- ابزاری ریاضی برای نشان دادن همروندی

- سابقه تشکیل شده است از تعدادی تراکنش

در یک اجرای همروند، ترتیب تقدم و تأخر انجام دستورات و توالی دستورات مهم است.

پس { درون تراکنش ← در یک سابقه توالی دستورات هر تراکنش عیناً حفظ می‌شود

بین تراکنش‌ها ← در یک سابقه باید تقدم و تأخر اجرای دستورات تراکنش‌های مختلف مشخص شود. بین هر دو دستور از دو تراکنش

لزوماً تعیین توالی مهم نیست (حتماً باید عمل داده مشترک

داشته باشند) ⇐ اعمال برخورددار

* (حتماً توالی دستورات به ترتیب اجرایشان در نتیجه تأثیرگذار است

باید مشخص شود)

در سابقه باید تقدم و تأخر (توالی) اعمال برخورددار مشخص شود

$R_i(x)$ $R_j(x)$ ✗ برخورد نیست

$R_i(x)$ $W_j(x)$ ✓ برخورد

$W_j(x)$ $R_j(x)$ ✓

$W_i(x)$ $W_j(x)$ ✓

دقت کنید که فقط دو دستور R و W داریم می‌توانیم بگوییم دستورات برخوردار دستوراتی هستند که هم قلم داده مشترک داشته باشند و یکی از آنها W باشد

نتیجه: هر مجموعه‌ای از ترتیب‌های جزئی یک ترتیب جزئی است پس: سابقه یک رابطه با ترتیب جزئی است که می‌توان برای آن یک تعریف رسمی ارائه داد.

تعریف رسمی سابقه: یک سابقه (History) روی مجموعه تراکنش‌های $\{T_1 \ldots T_n\}$ یک رابطه با ترتیب جزئی است که

$$H = \langle \underbrace{\{T_1 \ldots T_n\}}_{H}, <_H \rangle$$

$$1)\ H = \bigcup_{i=1}^{n} T_i$$

$$2)\ \bigcup_{i=1}^{n} <_i \ \subseteq\ <_H$$

۳) برای هر دو تراکنش که $T_i \in H$ و $T_j \in H$ و برای هر دو عمل برخوردار $p_i \in T_i$ و $q_j \in T_j$ و به شرط آنکه $c_i, c_j \in H$ داریم

$$\begin{cases} p_i <_H q_j \\ \text{یا} \\ q_j <_H p_i \end{cases}$$

اگر تراکنشی Commit نشود در نتیجه تأثیرگذار نیست.

- روی مجموعه تراکنش‌ها فرض بر این است که همه Commit می‌شوند

Subject: ______ Date: ______

یادآوری: اگر تراکنش Commit کرده باشد، آن تراکنش نهایی شده یا Commited گویند

" " " Abort " " " ساقط شده یا Aborted "

" " " start خورده و هنوز نه C شده نه A، تراکنش فعال یا Active گویند.

aborted } => Uncommited نهایی نشده
Active }

یک سابقه کامل، همه تراکنش هایش Commited هستند. این سابقه کامل به درد scheduler نمی‌خورد.

عموم سابقه را تعریف می‌کنیم

سابقه: هر prifix (پیشوند) از یک سابقه کامل است (خود سابقه کامل هم یک سابقه است)

تراکنش به محض اینکه Abort شد، از سابقه حذف می‌شود. پس در سابقه یا Commit داریم یا Active

جزء نهایی شده یک سابقه: $H \Rightarrow C(H)$

یک سابقه کامل روی زیرمجموعه‌ای از تراکنش‌هاست و تنها شامل دستورات تراکنش‌های نهایی شده می‌باشد

از یک سابقه، اثر تصویر نهایی شده آن است.

مثال) یک سابقه کامل از سه تراکنش زیر:

$T_1 = R_1(x)\ W_1(x)\ C_1$

$T_2 = R_2(x)\ W_2(y)\ W_2(x)\ C_2$

$T_3 = R_3(y)\ W_3(x)\ W_3(y)\ W_3(z)\ C_3$ — برای هر دو عمل برخورددار، تقدم و تأخر مشخص شده است.

$R_2(x) \to W_2(y) \to W_2(x) \to C_2$

$H_1 = R_3(y) \to W_3(x) \to W_3(y) \to W_3(z) \to C_3$

$R_1(x) \to W_1(x) \to C_1$

$H_2 =$ { خواص

بین تراکنش‌های مختلف چند تا سابقه با ترتیب متفاوت می‌توان ساخت

H_1 : اجرای ۱ : $R_3(y)\ R_1(x)\ W_1(x)\ W_3(x)\ W_3(y)\ C_1\ R_2(x)\ W_3(z)$
$W_2(y)\ W_2(x)\ C_2\ C_3$

اجرای ۲ : $R_1(x)\ R_3(y)\ W_1(x)$ ---

اجرای ۳

⋮

n تا اجرای متفاوت برای هر سابقه و n تا سابقه متفاوت می‌توان داشت. این که همین اجراها ترتیب‌پذیر هستند پس باید دنبال سابقه‌ای ترتیب‌پذیر باشیم که تمام اجراهایش ترتیب‌پذیر خواهد بود

حال می‌خواهیم ببینیم در ادامه بحث H_1 سابقه‌ای ترتیب‌پذیر است یا H_2 نیست
ترتیب‌پذیر

$C(H_1) = H_1$

H ↑ سابقه = $R_2(x)$, $R_3(y)$, $R_1(x)$ } → $C(H)$ پیشوند سابقه است ولی ... نیست

تراکنش نهایی شده : در سابقه H تراکنش T_i نهایی شده است اگر $C_i \in H$

تراکنش سقط شده : " " " " سقط شده است اگر $a_i \in H$: حذف می‌شود
↓ هر نوع اطلاعاتی خواهیم

تراکنش فعال : " " " " فعال است اگر $a_i \notin H$ و $C_i \notin H$

سوالات پیروی‌پذیر : سابقه‌ای که به ما اجازه‌ی پیروی‌پذیری می‌دهد

چرا باید به یک سابقه‌ی پیروی‌پذیر پایبند داشته باشد

سابقه‌ی پیروی را تعریف می‌کنیم. هم ارزی سوالات را اثبات. آنگاه می‌گوییم سابقه‌ای پیروی‌پذیر است

که هم ارز سابقه‌ی پیروی باشد

Subject: 5th Session Date: 91/8/6

Sh. هر شء دنبال این است که یک سابقه پی‌درپی‌پذیر بسازد هر اجرای از این سابقه پی‌درپی‌پذیر را پی‌درپی‌پذیر خواهد بود.

- سابقه پی‌درپی
- هم‌ارزی سوابق

⟸ سابقه پی‌درپی‌پذیر

سابقه پی‌درپی: سابقه‌ای که در آن تمامی اعمال هر تراکنش مثل T_i قبل از اعمال دیگر تراکنش مثل T_j باشد سابقه پی‌درپی است. در سابقه پی‌درپی تراکنش‌ها شروع و ختم خواهند شد (نسبت به هم). سابقه پی‌درپی فقط یک اجرا خواهد ساخت که آن هم اجرای پی‌درپی است.

$H: T_1 \mid T_2 \mid T_3$

هم‌ارزی دو سابقه: دو سابقه با ترتیب جزئی A و B به شرطی با هم هم‌ارزند:

$A = (A', <_{A'})$ ① $A' = B'$

و

$B = (B', <_{B'})$ ② $<_{A'} = <_{B'}$

دو سابقه هم‌ارز تأثیر ...

۱) مجموعه دستورات یکسان (تراکنش‌های یکسان)

۲) اعمال برخوردار را به صورت یکسانی مرتب کرده باشند (سایر دستورات داخل تراکنش‌ها به هم ترتیب تناسبی دارند)

Subject: Date:

* مشخص کردن ترتیب اعمال برخورددار به دلیل تاثیرگذار بودن آنها در نتیجه است

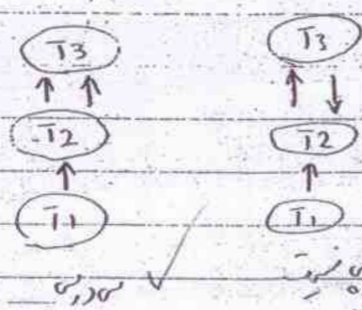
دو سابقه زمانی هم ارز هستند هرگاه عملیات تراکنش آنها یکسان باشد و آنهایی که از اعمال که در نتیجه تاثیرگذار هستند، در هر دو به یک شکل مرتب شده باشند

سابقه پی درپی پذیری:

یک سابقه H پی درپی پذیر است اگر و فقط اگر هم ارز یک سابقه پی درپی باشد (سابقه ای پی درپی پذیر است که اعمال برخورددار آن مثل یک سابقه پی درپی مرتب کرده باشد)

در یک سابقه پی درپی، تمامی اعمال برخورددار یک تراکنش قبل از اعمال برخورددار تراکنش دیگر انجام می‌پذیرد

یعنی سابقه ای پی درپی پذیر است که تمامی اعمال برخورددار یک تراکنش قبل از اعمال برخورددار تراکنش دیگر باشد



محک پی درپی پذیری

این محک از ابزاری استفاده می‌کند به نام گراف پی درپی پذیری Serialization Graph (SG)

گراف SG را برای یک سابقه ترسیم می‌کنیم. $H \rightarrow SG(H)$

هر گراف دو عنصر پایه دارد: گره و لبه. SG یک گراف جهت دار است.

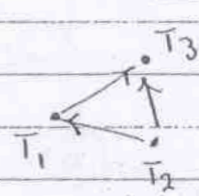
تعریف گراف SG: هر تراکنش نماینده یک گره در گراف دارد

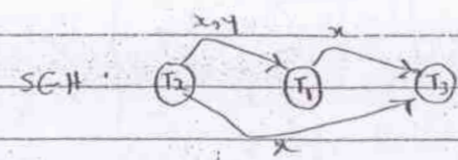
برای هر دو گره T_i و T_j در $SG(H)$ ، $T_i , T_j \in SG(H)$ ، $T_i \to T_j \in SG(H)$

اگر $p_i^{(x)} <_H q_j^{(x)}$ ، $p_i \in T_i$ ، $q_j \in T_j$

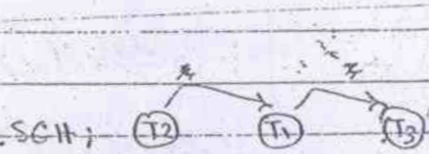
مثال

$H:$
$R_1(x) \to W_1(x) \to W_1(y) \to C_1$
$R_2(x) \to W_2(y) \to C_2$
$R_3(x) \to W_3(x) \to C_3$





$H':$
$R_3(x) \to W_3(z) \to C_3$
$R_1(x) \to W_1(x) \to W_1(y) \to C_1$
$R_2(x) \to W_2(y) \to C_2$



این بار بین T2 و T3 یالی وجود ندارد

چون T2 و T3 اعمال برخورددار ندارند

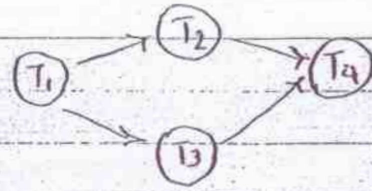
تئوری سریال پذیری (قضیه): یک سابقه H سریال پذیر است اگر و فقط اگر SG(H) فاقد حلقه باشد

اثبات: (اگر) (SG(H) فاقد حلقه است آنگاه H سریال پذیر است) فرض کنیم H سابقه‌ای

است روی تراکنش های T_1 تا T_n تعریف شده است $T=\{T_1 \dots T_n\}$ H

فرض می‌کنیم تراکنش‌های $T_1 \dots T_n$ به $m \le n$ ناجدا شده‌اند بنابراین گراف SG شامل m گره است. فرض کردیم SG(H) فاقد حلقه است پس با استفاده از توپولوژیکال sort (Topological Sort) می‌توانیم گراف را مرتب کنیم. بنابراین می‌توان ترتیب $i_1 \dots i_m$ برای این m تراکنش پیدا کرد. $T_{i_1}, T_{i_2}, \dots, T_{i_m}$

برای هر یک از این ترتیب‌ها تراکنش‌ها را می‌چینیم. این یک سابقه پی‌درپی است.



مثال با هر ترتیب یک سابقه پی‌درپی داریم:
T_1, T_2, T_3, T_4
T_1, T_3, T_2, T_4

~~[illegible]~~ در سابقه H $\begin{cases} p_i \in T_i \\ q_j \in T_j \end{cases}$ که هر دو کانفلیکت T_i و T_j نامیده شده‌اند

فرض می‌کنیم عمل p_i قبل از عمل q_j وجود دارد (اعمال :) بنابراین در گراف SG یک یال از T_i به T_j وجود دارد. بنابراین در هر Topological Sort حتماً T_i قبل از T_j است پس در هر سابقه پی‌درپی مثل H_s که از Top. sort حاصل شود، T_i قبل از T_j است و در نتیجه در هر سابقه $p_i <_{H_s} q_j$ پس نتیجه می‌گیریم که سابقه H اعمال موجود در تراکنش‌هایش به سابقه پی‌درپی مرتب کرده است $\Rightarrow$ H پی‌درپی‌پذیر است.

اثبات قسمت دوم (شرط لازم قضیه): فرض می‌کنیم H پی‌درپی‌پذیر است آن گاه ثابت می‌کنیم SG(H) فاقد حلقه است:

H پی‌درپی‌پذیر است پس هم ارز یک تاریخچه پی‌درپی مثل H_s است. فرض کنید $T_i \to T_j \in SG(H)$

یعنی وجود دارد $p_i \in T_i$ و $q_j \in T_j$ که $p_i <_H q_j$ و چون $C(H) \equiv H_s \Rightarrow p_i <_{H_s} q_j$

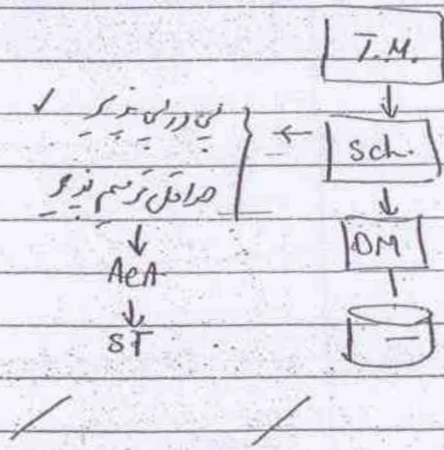
چون H_s پی‌درپی است پس در H_s کل T_i قبل از T_j است.

برهان خلف: به فرض خلف: در $SG(H)$ حلقه $T_i \to T_j \to \dots \to T_k \to T_i$ وجود دارد و این یعنی

در H_s، T_j باید قبل از T_i و T_i قبل از T_j باشد که فاقد معنی است پس فرض

خلف باطل است و حکم ثابت شد.

Sch هم از گراف پی‌درپی‌پذیری برای تشخیص کارها در سمت تراکنش‌ها استفاده می‌کند.



حال سعی می‌کنیم راه حلی برای پیدا کردن اجراهای ترسیم نوع پیدا کنیم.

سابقه ترمیم پذیر

اجرای ترمیم پذیر: اگر تراکنشی قلم داده ای را که از تراکنش دیگری خواند Commit اثر تا هنگام Commit

آن تراکنش به تعویق بیفتد.

خواندن از: تراکنش T_i قلم داده x را از T_j خوانده است اگر:

1) T_j ، x را نوشته باشد سپس T_i ، x را بخواند

2) قبل از خواندن T_i ، T_j سقط نکرده باشد.

3) برای هر تراکنش دیگری مثل T_k اگر پس از T_j در x نوشته باشد، قبل از خواندن

T_i abort کرده باشد.

در سابقه H گوییم تراکنش T_i قلم داده x را از T_j خوانده اگر:

1) $W_j(x) <_H R_i(x)$

2) $a_j \nless_H R_i(x)$

3) اگر $W_k(x)$ وجود دارد، آنگاه

اگر $W_j(x) < W_k(x) < R_i(x)$ آنگاه $a_k < R_i(x)$

سابقه H ترمیم پذیر است اگر برای هر دو تراکنش T_i و T_j اگر T_i قلم داده ای را از T_j خوانده

باشد آنگاه $C_j <_H C_i$

Subject: Date:

اجراهایی از یک سابقه ترمیم پذیر الف که $C_j <_H C_i$ یعنی زمانبند یک شرط ساده می گذارد: اگر

«خواندن از» داشتیم، wait می کند تا Commit ها بیاید.

همین کار را می تواند برای اجرای ACA انجام دهد. یعنی یک شرط می گذارد که از آن هایی بخوان

که Commit شده اند.

سابقه H از انتشار سقط جلوگیری می کند (ACA) اگر هرگاه T_i مقدار داده ای را از T_j خواند، آنگاه

$C_j < R_i(x)$

این شرط را زمانبند می گذارد و به این ترتیب سابقه همیشه اجرای ACA می دهد.

* سابقه H را محض (strict) (st) گوییم هرگاه

$$w_j(x) < O_i(x) \Rightarrow C_j < O_i(x)$$

$$\left(O_i(x) < \begin{matrix} R_i(x) \\ W_i(x) \end{matrix} \right) \qquad a_j < O_i(x)$$

مثال)

$T_1 : W_1(x)\ W_1(Y)\ W_1(Z)\ C_1$

$T_2 : R_2(x)\ W_2(x)\ R_2(Y)\ W_2(Y)\ C_2$

$H_1 : W_1(x)\ W_1(Y)\ R_2(x)\ W_2(x)\ R_2(Y)\ W_2(Y)\ C_2\ W_1(Z)\ C_1$

نیست ACA

نیست st

نیست RC

ترمیم پذیر نیست

$T_1 \rightarrow T_2$

☑ SR ☒ RC ☒ ACA ☒ ST

سریال پذیر ترمیم

Subject: Date

H_2 : $w_1(x)$ $w_1(y)$ $R_2(u)$ $w_2(x)$ $R_2(y)$ $w_2(y)$ $w_1(z)$ C_1 C_2

☑ SR ☑ RC ☒ ACA ☒ ST

$T_1 \rightarrow T_2$

H_3 : $w_1(x)$ $w_1(y)$ $R_2(x)$ $R_2(u)$ $w_2(x)$ $w_1(z)$ C_1 $R_2(y)$ $w_2(y)$ C_2

SR ✓ RC ✓ ACA ✓ ST ✗

H_4 : $w_1(x)$ $w_1(y)$ $R_2(u)$ $w_1(z)$ C_1 $w_2(x)$ $R_2(y)$ $w_2(y)$ C_2

SR ✓ RC ✓ ACA ✓ ST ✓

Subject: 6th session Date: 91/8/20

قضیه: $ST \subset ACA \subset RC$

اول ثابت می‌کنیم $ST \subset ACA$ و سپس ثابت می‌کنیم $ACA \subset RC$ و در نتیجه کل قضیه ثابت می‌شود.

روش اثبات $A \subset B$:

$$\begin{cases} \forall x \in A \Rightarrow x \in B \Rightarrow A \subseteq B \\ \exists y \in B \Rightarrow y \notin A \Rightarrow A \subset B \end{cases}$$

اثبات: فرض کنیم H یک سابقه اکید است $H \in ST$، $T_i, T_j \in H$ و T_j x را از T_i خوانده است و تراکنش T_i و $C_i, C_j \in H$

x را از T_i خوانده است $\Leftarrow w_i(x) < R_j(x)$

$$ST \subseteq ACA \Leftarrow H \in ACA \Leftarrow \begin{cases} w_i(x) < R_j(x) \\ a_i \not< R_j(x) \rightarrow \text{بنا بر ...} \\ C_i < R_j(x) \rightarrow \text{بنا بر ...} \end{cases}$$

در مثال جلسه پیش H_3 سابقه‌ای بود که ACA بود ولی اکید نبود بنابراین $ST \subset ACA$

اثبات قسمت دوم

فرض می‌کنیم $H \in ACA$ و $T_i, T_j \in H$ و تراکنش T_j، x را از T_i خوانده است، $C_i, C_j \in H$

$$\begin{cases} w_i(x) < R_j(x) \\ a_i \not< R_j(x) \\ C_i < R_j(x) \text{ بنابر ACA} \\ R_j(x) < C_j \end{cases} \Rightarrow w_i(x) < C_i < R_j(x) < C_j \Rightarrow C_i < C_j$$

مسیر کردن C_i قبل از C_j شده.

$\Rightarrow H \in RC$

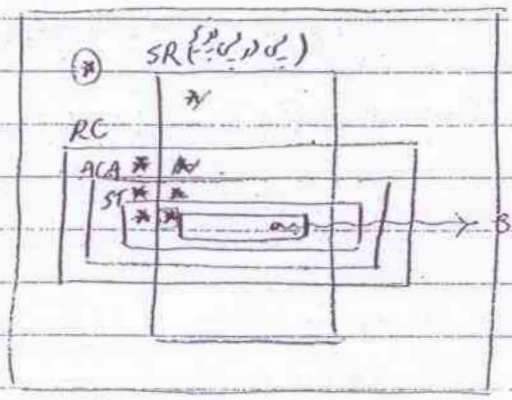
$\Rightarrow ACA \subseteq RC$

H_{11} سابقه‌ای بود که RC بود ولی ACA نبود پس $ACA \subset RC$

Yeah

Subject: Date:

رابطه بین انواع اجرا



تمرین: سه تراکنش مثال بزنید با همین سه تراکنش انواع اجراهای زیر را مثال بزنید

⊛ نه SR و نه RC

* SR باشد ولی RC نباشد

برای تمام نقاط علامت گذاری شده توسط این ظاهر انجام دهید

⊛ هم SR و هم RC: $w_1(x)\ R_2(x)\ w_2(y)\ R_1(x)\ C_1\ C_2$ ✓

* فقط SR: $w_1(x)\ R_2(x)\ C_2\ C_1$ ✓

*! هم SR و هم RC: $w_1(x)\ R_2(x)\ C_1\ C_2$ ✓

* هم SR و هم ACA (یعنی RC هم هست): $w_1(x)\ w_1(y)\ w_2(y)\ C_1\ R_2(x)\ C_2$ ✓

⊛ هم SR و هم ST: (یعنی RC و ACA هم هست) $w_1(x)\ w_1(y)\ C_1\ w_2(y)\ R_2(x)\ C_2$ ✓

* SR باشد RC نباشد: $R_2(x)\ w_1(x)\ w_1(y)\ w_2(z)\ R_2(y)\ C_1\ C_2$ ✓

* SR باشد ACA نباشد: $R_2(x)\ w_1(x)\ w_2(x)\ w_1(y)\ C_1\ R_2(y)\ C_2$ ✓

* SR باشد اکا نباشد: $R_2(x)\ w_1(x)\ w_1(y)\ C_1\ w_2(x)\ R_2(y)\ C_2$ ✓

۲۱

In the name of GOD

Subject: | Date:

سؤال؟

۲۲) تمام صحبت‌هایی که کردیم تئوری‌هایی که مطرح کردیم برای روالی کامل بوده ولی در دنیای واقعی این جوری نیست. زمانبند چگونه این تئوری‌ها را برای روالی که هنوز کامل نشده اعمال کند؟

خاصیت PCC (Prefix Commit Closed)

ترمیم پذیری: فرض کنیم ترمیم پذیری خاصیت PCC باشد در اینصورت اگر سابقه‌ای ترمیم پذیر است تمام زیر تاریخچه‌ها (Prefix داشتن) نیز ترمیم پذیر خواهند بود.

تعریف خاصیت PCC: یک خاصیت زمانی PCC است، اگر سابقه H آن را داشته باشد هر زیر سابقه‌اش مثل H'، C(H') آن خاصیت را داشته باشد.

ثابت می‌شود که تسلسل پذیری [1]، ترمیم پذیری [2]، ACA [3] و ST [4] خاصیت PCC هستند!

Sch سعی می‌کند تا جایی که دلبخواهش است از خواص مورد نظرش را حفظ کند تا در نهایت اجرای مورد نظرش را داشته باشد.

به عنوان تمرین ثابت کنید تسلسل پذیری، ترمیم پذیری، ACA و آکید بودن PCC هستند.

$H \in SR \Rightarrow SG(H)$ بدون حلقه است

فرض کنیم $H' \subseteq H$ بنابراین اگر $p_i(x) < q_j(x) \in H' \Rightarrow p_i(x) < q_j(x) \in H$

اگر $T_i \rightarrow T_j \in SG(H') \in SG(H) \Rightarrow SG(H') \subseteq SG(H)$

برهان خلف: اگر $SG(H')$ دارای حلقه باشد، $SG(H)$ نیز دارای حلقه خواهد بود که این تناقض است!

بنابراین H' نیز SR است

Subject: Date:

تا به حال تئوری مبتنی بر اعمال پایگاه داده‌ای بود حتی هم‌ارزی که مطرح شد به همین علت به آن هم‌ارزی پایگاه داده‌ای گفتیم.

تاکنون اعمال پایگاه داده‌ای را اعمالی می‌دانستیم که یکی از آنها W باشد و عملاً فقط در مورد R و W صحبت کردیم. اما اعمال دیگری نیز در مجموعه اعمال پایگاه داده‌ای وجود دارند.

اعمال جدید

T_i \ T_j	R	W	Inc	Dec
R	y	n	n	n
W	n	n	n	n
Inc	n	n	y	y
Dec	n	n	y	y

مثال: نتیجه R(x) Inc(x) با نتیجه Inc(x) R(x) متفاوت است پس با هم سازگار نیستند.

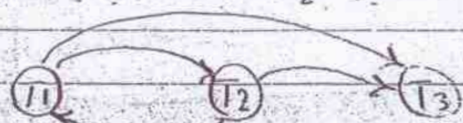
این جدول را می‌توانیم توسعه دهیم. توجه داشته باشید اعمال اتمیک هستند مثلاً Inc را نمی‌توان به R و W شکست.

این جدول، جدول سازگاری است که گاهی جدول ناسازگاری رسم می‌شود که در این صورت جای y و n عوض می‌شود.

مثال)

$w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1\ w_3(x)\ w_3(y)\ c_3$

چنین گراف:



این اجرا پی‌درپی‌پذیر نیست

ثابت می‌شود که این اجرا با اجراهای زیر هم‌ارز است:

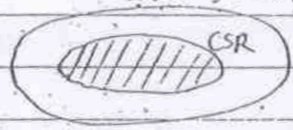
$H \equiv T_2\ T_1\ T_3$

این اجراها پی‌درپی هستند پس نتیجه می‌گیریم گراف پی‌درپی‌پذیری

$H \equiv T_1\ T_2\ T_3$

در صورتی که فاقد حلقه باشد اجرا حتماً پی‌درپی‌پذیر است ولی وجود حلقه عدم پی‌درپی‌پذیری را ثابت نمی‌کند

CSR (Conflict - SR):



اجرای سریالی‌پذیر برخوردی: همان‌طور که با گراف ثابت می‌شود

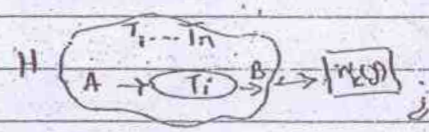
هم‌ارزی برخوردی

سریالی‌پذیری برخوردی

هم‌ارزی دیدی View Equivallance

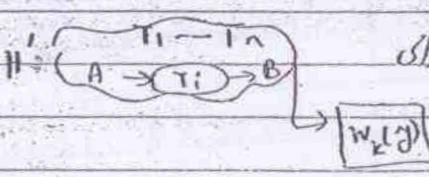
سریالی‌پذیری دیدی View Serializity

هم‌ارزی دیدی



← از تراکنش‌های دیگر

۱. اگر خواندن از ها در هر دو سابقه یکسان باشد آن‌گاه نتایج یکسانی



تولید می‌کند (یک تراکنش که از دو سابقه مختلف است به ازای

ورودی‌های یکسان باید خروجی‌های یکسان تولید کند)

اگر T_i در H از T_j می‌خواند در H' هم باید از T_j بخواند و خروجی را در هر دو به T_k بدهد

۲. از لحاظ نهایی یکسان باشند یعنی باید آخرین نوشتن‌ها یکسان باشد

اگر برای هر قلم داده x این دو شرط اتفاق بیفتد ← دو سابقه نتایج یکسان تولید می‌کنند ← هم‌ارز هستند

با این استدلال مثال قبل نیز اجرای سریالی‌پذیر است هرچند گراف آن دارای حلقه است.

تعریف رسمی هم ارزی دیدی

آخرین نوشتن: گوییم در سابقه H ، $W_i(x) \in H$ آخرین نوشتن روی x است اگر

۱) $C_i \in H$

۲) برای هر $W_j(x) \in H$ یا $W_j(x) < W_i(x)$ یا $a_j \in H$

هم ارزی دیدی: دو سابقه H و H' هم ارز دیدی هستند اگر:

۱) مجموعه تراکنش ها یکسان باشد

۲) (خواندن از یکسان) برای هر قلم داده x اگر در سابقه H ، T_i ، x را از T_j می خواند در سابقه H' هم T_i ، x را از T_j بخواند.

۳) (آخرین نوشتن ها یکسان) برای هر قلم داده x اگر در سابقه H ، T_i آخرین تراکنشی است که روی x نوشته است ($W_i(x)$) آنگاه در سابقه H' هم T_i آخرین نوشتن را روی x انجام دهد.

پی درپی پذیری دیدی (VSR)

سابقه H پی درپی پذیر دیدی است اگر هم ارزی دیدی با یک سابقه پی درپی داشته باشد (این تعریف تعریف کاملی نیست و یک مشکل اساسی دارد و آن هم از بین رفتن خاصیت PCC روی پی درپی پذیری است)

برای حفظ PCC روی پی درپی پذیری مبتنی بر هم ارزی دیدی اینطور گوییم:

سابقه H پی درپی پذیر دیدی است اگر برای هر زیرسابقه از آن مثل H' ، C(H') هم ارزی دیدی با یک سابقه پی درپی داشته باشد

قضیه: CSR ⊂ VSR

اثبات این قضیه در کتاب آورده است. حتماً مطالعه شود.

کاربرد بی‌درنگ پذیری دیدی در چند نسخه‌ای بودن و توزیعی است و در حالت متمرکز و تک نسخه‌ای بیشتر

بی‌درنگ پذیری برخوردی کاربرد دارد و ما در این زمینه تمرکز می‌کنیم بنابراین از این به بعد بیشتر فرضی

برخوردی است مگر آنکه صریحاً ذکر شود.

روش‌های زمانبندی

روش‌های زمانبندی بر اساس ابزاری که استفاده می‌کنند به دو دسته تقسیم می‌شوند:

- مبتنی بر قفل: معروف‌ترین این روش‌ها 2PL (2 Phase Locking) است

- غیر قفلی: روش‌های ترتیب مهر زمانی Time Stamp Ordering (TSO)
  روش آزمون گراف‌ها Serialization Graph Testing (SGT)

هر این روش‌ها نیز [illegible] نوع‌های پایه‌ای دارند که این انواع پایه ترکیب یافته‌اند و ورژن‌های مختلف با اسامی

مختلف ارائه داده‌اند.

بر اساس ماهیت روش‌های زمانبندی به دو دسته تقسیم می‌شوند:

- مهاجم: Agressive: بالای از Abort کردن تراکنش‌ها ندارند

- محافظه‌کار: Conservative: حتی‌الامکان سعی می‌کنند تا تراکنش‌ها حتی‌الامکان Abort نشود

Subject: Date:

[Sch] در برای هر دستور سه عمل می‌توانند انجام دهند : Run ، Wait ، Abort

روش‌های مهاجم اولویتشان با Run است اما روش‌های محافظه‌کار سعی می‌کنند Wait کنند تا

توالی دلخواه را درست کنند . اما روش‌های مهاجم اگر در حین اجرا متوجه شوند توالی درست نیست

Abort می‌کنند .

- نوعی زمانبندهم داریم بنام گواهی دهنده‌ها (Certifiers) . این زمانبندها هیچ مکانیزمی

برای زمانبندی ندارند هر دستوری را برای اجرا می‌فرستند تا زمانی که تراکنش می‌خواهد Commit کند و هنگام

Commit کردن توالی دستورات اجرا شده را چک می‌کند و اگر توالی درست بود Commit را تصدیق می‌کند

در غیر این صورت Abort می‌کند . این زمانبندها نوعی از مهاجم‌ها و غیرقفلی‌ها قرار می‌گیرد

این زمانبند در محیط‌هایی که اشتراک داده بسیار کم است کاربرد دارد و در این قبیل محیط‌ها لازم نیست

مکانیزم‌های پیچیده برای زمانبندی در نظر گرفته شود .

روش 2PL :

مفهوم قفل :

قفل

ترتیب اجرای اعمال برخورددار در تمام اجراهای ذکر شده موثر است. قفل ابزاری است برای زمانبند تا بتواند ترتیب دلخواهش را بدست آورد.

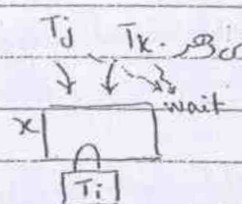
محل دعوا: داده های مشترک. قفل روی قلم داده مشترک زده می شود

براساس ترتیب دلخواه امکان استفاده از قلم داده به تراکنش ها داده می شود.

چیزی که در اصل پیاده سازی می شود: قلم داده مشترک قابل دسترسی توسط اجرای تراکنش است. مفهوم قفل این است که

تراکنشی که می خواهد از قلم داده استفاده کند ابتدا باید قلم داده را مال خود کند (قفل قلم داده را اخذ کند)

زمانبند به هر تراکنشی که بخواهد دسترسی بدهد قفل مربوط را به آن تراکنش تخصیص می دهد.



در مثال روبرو، تراکنش T_i مثلاً می خواهد x را بخواند، T_j می خواهد x را بنویسد.

T_k می خواهد آن را بخواند چون x مال T_i است هیچ کدام از T_j و T_k نمی توانند x دسترسی داشته باشند و این سخت گیرانه تر از تئوری است حتی. یعنی T_k طبق تئوری باید بتواند x دسترسی داشته باشد (دو R همزمان) اما با این روش نمی تواند

حال این بهتر است بین قفل ها تفاوت بگذاریم و متناسب با هر عمل قفل متناظر با آن را داشته باشیم

R ← قفل R

W ← قفل W

⋮

Subject: _____ Date: _____

بنابراین همانگونه که اعمال برخوردار داشتیم حالا قفل‌های برخوردار داریم.

* دو قفل با یکدیگر برخوردار هستند اگر اعمال متناظرشان با یکدیگر برخوردار باشد.

زمانبند برای تشخیص قفل‌های برخوردار، از همان جدول سازگاری (ناسازگاری) استفاده می‌کند.

برای انجام هر عمل زمانبند ابتدا باید قفل را اخذ، سپس عمل را انجام و پس از انجام عمل قفل را آزاد کند.

$$R_i L(x) \prec R_i(x) \prec R_i U(x)$$

Lock ↓ Unlock ↓

$$W_i L(x) \prec W_i(x) \prec W_i U(x)$$

در حالت کلی برای هر عمل موجود مثل $O_i(x)$:

$$O_i L(x) \prec O_i(x) \prec O_i U(x)$$

هر روش قفل‌گذاری باید به چند مسئله اساسی پاسخ دهد. تفاوت بین روش‌های مختلف قفل‌گذاری نیز ناشی از نحوه پاسخ به این مسائل است: ۱) چه نوع قفلی ← متناسب با اعمال

۲) روی چه شیئی ← قلم داده ... ۳) چه کسی چه وقت قفل را درخواست

۴) چه وقتی داده می‌شود؟

۵) چه وقتی آزاد می‌شود؟

* تفاوت روش‌ها در این بخش خلاصه می‌شود (۳ و ۴ و ۵)

۲۵

Subject: Date:

روش 2pl (قفل گذاری دو مرحله ای) (2phase Locking

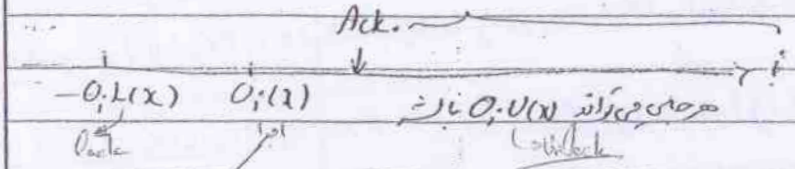
یکی از پرکاربردترین روش های قفل گذاری در پایگاه داده است و نسخه های زیادی هم برای پایگاه داده های توزیع و هم برای غیر توزیع تا کنون ارائه شده است.

روش B2pl (Basic 2pl)

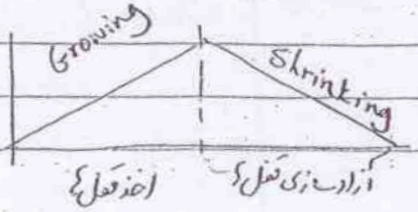
قواعد 2pl (این قواعد برای تمام نسخه ها یکسان است) (فرض: دو عمل P_i و q_j)

۱) عمل $P_i(x)$ به زمانی وارد می شود که تراکنش کنترل می کند آیا روی قلم داده x، قفل برجود دارد $q_j(x)$ وجود دارد یا خیر. اگر وجود نداشت $P_i(x)$ زمانبندی شده و برای اجرا Data Manager فرستاده می شود در غیر این صورت $P_i(x)$ Waits می شود. (یعنی قفل برجود دارد، نداشته، قفل لازم اختصاص داده شده و به Data Man. فرستاده می شود)

۲) حداقل تا پس از اخذ تأیید انجام عمل قفل اخذ شده، آزاد نمی شود (D.M. تأیید انجام عمل را می دهد) "Acknowledgment"



۳) هرگاه تراکنش قفلی را برای تراکنش آزاد نمود، هیچ قفلی روی هیچ قلم داده ای برای آن تراکنش اخذ نخواهد شد (تمام قفل هایی را که می خواهد تکمیل جمع بندی و بعد از آنکه همه قفل های مورد نیاز را گرفت می تواند (اگر خواست) قفلی را آزاد کند) ← قاعده دو مرحله ای



دلایل قواعد

تضمین‌ها :

۱. از اجرای آبشاری جلوگیری می‌کند

۲. Sch علاوه بر زمانبندی، تضمین اجرای ترتیب را نیز بر عهده دارد و این تضمین از طریق Hand Shacking

است. اگر قبل از اتمام اجرای دستور اول، دومی بر D.M فرستاده شود ممکن است ترتیب اجرا

بهم بخورد. پس دلیل قاعده دوم پیاده‌سازی Hardshacking است.

$P_i(x) < q_j(x)$ این باید تضمین شود

به این طریق تضمین می‌شود: $P_iL(x) < P_i(x) < P_iU(x) < q_jL(x) < q_j(x) < \dots$

۳. تضمین کننده پی‌درپی پذیری، عدم ایجاد حلقه (تمام لبه‌ها یکسان هستند)

مثال) $T_1 : R_1(x)\ W_1(y)\ C_1$

$T_2 : W_2(x)\ W_2(y)\ C_2$

فرض می‌کنیم قاعده سوم وجود ندارد

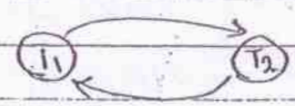
$R_1L(x), R_1(x), R_1U(x), W_2L(x), W_2L(y), W_2(x), W_2(y), C_2, W_2U(y),$

$W_1L(y), W_2U(x), W_1(y), W_1U(y), C_1$

SG: $R_1(x)\ W_2(x)\ W_2(y)\ C_2\ W_1(y)\ C_1$

گراف اجرای فوق به صورت زیر است

Cycle ✗ SG



گراف حلقه دارد. این اجرا پی‌درپی پذیر نیست

Subject: Date:

با فرض وجود قاعده دو فاز:

$R_1L(x)$, $R_1(x)$, $W_1L(y)$, $W_1(y)$, $R_1U(x)$, $W_1U(y)$, C_1,

$W_2L(x)$, $W_2(x)$, $W_2L(y)$, $W_2(y)$, $W_2U(x)$, $W_2U(y)$, C_2

$T_1 \rightarrow T_2$

با مثال برسی زیر:

$R_1L(x)$, $R_1(x)$, $W_1L(y)$, $R_1U(x)$, $W_2L(x)$, $W_2(x)$, $W_1(y)$, C_1

$W_1U(y)$, $W_2L(y)$ $W_2(y)$ C_2 $W_2U(x)$, $W_2U(y)$, C_1

$T_1 \rightarrow T_2$

قاعده دو فاز تضمین می کند تمام برنامه ها به جهت باشد و حلقه وجود نداشته باشد.

هر روش قفل گذاری که ارتباط زائی دارد و آن دیده می نیست است.

روش 2pl هم از این قاعده مستثنی نیست.

T_1: $R_1(x)$ $W_1(y)$ C_1

T_3: $W_3(y)$ $W_3(x)$ C_3

$R_1L(x)$ $R_1(x)$ $W_3L(y)$ $W_3(y)$??? Dead Lock

Dead Lock: شکل زائی تمام روش های قفل گذاری

Subject: ______ Date: ______

راه حل مشکل بن بست: بن بست‌گیری

- مقابله: Abort کردن و آزاد کردن منابع

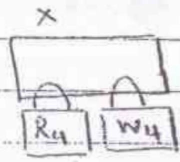
- انتخاب قربانی (و جلوگیری از Starvation)

⋮

(رجوع شود به مباحث OS)

مشکل دیگر:

$T_4 : R_4(x)\ W_4(x)\ C_4$



روی یک قلم داده، یک تراکنش هم قفل R زده و هم W. فرض کنیم

تراکنش T_5 یک دستور $R_5(x)$ را دارد. در برخورد با قفل؟

تکلیفش روشن نیست. اگر قفل R باشد می‌تواند به x دسترسی داشته باشد، و اگر قفل W باشد

نمی‌تواند.

راه حل می‌تواند این باشد که T_4 ابتدا قفل R، قفل R را آزاد کند و بعد قفل W را بزند؟ این با قانون

دوم در تضاد است.

راه حل مفهوم جدیدی است به نام تبدیل قفل

ارتقای

که قفل روی یک قلم داده نمی‌توان زد. یک قفل را هم نمی‌توان باز کرد و دوباره زد. ولی قفل را می‌توان تبدیل کرد.

در این مثال قفل R به قفل W تبدیل می‌شود.

۲۷

تبدیل قفل Lock Conversion

قفل ها هیچ وقت Down Grade نمی شوند و یا Upgrade می شوند یعنی قفل R به قفل W تبدیل می شود و یا بالعکس نه. یعنی اگر روی یک داده ای قفل W بود Read هم می تواند انجام دهد بدون

تبدیل قفل: Lock Upgrading

به غزال ترین شکل ممکن Downgrade کردن قفل می تواند کارایی سیستم را بهم بزند

$T_1: w_1(x) R_1(x) w_1(y) c_1$

$T_2: R_2(y) R_2(x) c_2$

$H_1: w_1(x) R_1(x) R_2(y) R_2(x) c_2 w_1(y) c_1$

همین مسئله تبدیل قفل می تواند مشکل دیگری را ایجاد کند

$T_4: R_4(x) W_4(x) C_4$

$T_7: R_7(x) W_7(x) C_7$

$R_4L(x) R_4(x) R_7L(x) R_7(x)$???

در اینجا، اگر بخواهد هر کدام از قفل ها را به قفل W ارتقاء دهد نمی تواند چون یکدیگر را مجبور خواهد کرد

یعنی ارتقاء قفل منجر می شود به بن بست

بن بست Dead Lock

برای تشخیص بن بست می توان از WFG (Wait For Graph) استفاده کرد. اگر در این گراف حلقه باشد بن بست رخ داده.

Subject: ______ Date: ______

انتخاب قربانی : - یک تراکنش چه وقتی اجرایش شروع شده؟

- یک تراکنش چه وقتی تا پایانش باقی مانده؟

- یک تراکنش چه تعداد قفل را گرفته، چقدر می‌خواهد ... .

⋮

(مطالعه شود →)

این پارامترها در انتخاب قربانی تاثیر دارند. چون قربانی کردن تراکنشی که پایان راهش نزدیک است سخت است.

اثبات درستی روش 2PL

باید ثابت کنیم اجراهای تولید شده توسط یک زمانبند 2PL سریال‌پذیر است.

یعنی SG فاقد حلقه است

قاعده ۱ : $O_iL(x) < O_i(x)$

قاعده ۲ : $O_i(x) < O_iU(x)$

← نتیجه ۱ : $O_iL(x) < O_i(x) < O_iU(x)$

(همان اصل ... در مورد قفل)

قاعده ۱ : اگر $P_i(x)$ و $q_j(x)$ را دو عمل برخوردار در نظر بگیریم یعنی قفل را اخذ کرده، دیگری wait می‌کند تا عمل اول قفل را آزاد کند. ← نتیجه ۲ : در هر تاریخچه H که توسط زمانبند 2PL تولید شده است، برای دو عمل برخوردار $P_i \in T_i$ و $q_j \in T_j$ ← $P_i, P_j \in C(H)$ آن گاه:

$$P_iU(x) < q_jL(x) \Rightarrow P_i(x) <_H q_j(x) \Rightarrow T_i \rightarrow T_j$$

یا

$$q_jU(x) < P_iL(x) \Rightarrow q_j(x) <_H P_i(x) \Rightarrow T_j \rightarrow T_i$$

Subject: Date:

قاعده ۳: اگر قفلی را تراکنشی آزاد کرد، قفل دیگری را اخذ نمی‌کند یعنی اخذ همه قفل‌ها قبل از آزادسازی اولین آن‌هاست. به عبارت دیگر آزادسازی اولین قفل بعد از اخذ آخرین قفل

نتیجه ۳: اگر H یک سابقه تولید شده توسط زمانبند 2pl باشد آنگاه برای هر تراکنش $T_i \in G(H)$

و هر دو عمل $R_i(x)$ و $S_i(y)$ داریم: $s_i L(y) < r_i U(x)$ و $r_i L(x) < s_i U(y)$

[illegible]

فرض کنیم در $SG(H)$، $T_i \to T_j$ یعنی $p_i < q_j$ یعنی طبق نتیجه دوم، $p_i U(x) < q_j L(x)$

یعنی T_i قفلی را آزاد کرده است که T_j توانسته قفلی را بگیرد

فرض کنیم داریم: $T_i \to T_j \to T_k$ (* : $T_i \to T_j$ ، ** : $T_j \to T_k$)

* → $p_i U(x) < q_j L(x)$ ، p,q بر سر x دارند

** → $r_j U(y) < t_k L(y)$ ، t,r بر سر y دارند

نتیجه ۳ ⇒ $q_j L(x) < r_j U(y)$

} $p_i U(x) < t_k L(y)$

⇒ یعنی تراکنش T_i قفلی را آزاد کرده که T_k قفلی را اخذ کرده است

برهان خلف:

فرض خلف:

فرض کنید در سابقه H گراف $SG(H)$ حلقه‌ای به صورت زیر وجود دارد $T_1 \to T_2 \to \dots \to T_n \to T_1$

این یعنی T_1 قفلی را آزاد کرده که T_1 توانسته قفلی را اخذ کند که این با قانون لاک دو مرحله‌ای در تضاد است پس

(قاعده ۳ روش 2PL)

فرض خلف باطل و حلقه رخ نمی‌دهد پس حکم ثابت است. ✓

که همان قاعده [illegible] پذیر است.

Subject: _____ Date: _____

لم ۱: اگر H کدینگ با فاصله 2pl باشد و فرض کنید $T_i \to T_j$ متناظر با SG(H) باشد

آن‌گاه $P_i V(x) < q_j L(x)$ (به عنوان برش ارتباط نمود)

لم ۲: اگر H کدینگ با فاصله 2pl باشد و فرض کنیم $T_n \to \dots \to T_2 \to T_1$ یک مسیر در SG(H) باشد آن‌گاه

ثابت می‌شود برای هر کلمه داده x و y $P_i V(x) < q_n L(y)$

ص 56 از فصل ۳

قضیه: هر فاصله 2pl می‌دیدنی پذیر است

Subject : 8th Session Date 91/9/11

گفتیم در قفل گذاری شکل اصلی بن بست است روش های بیشتری در مقابله با بن بست را قبلاً خواندیم

- طرح های ارزش Time Out برای مقابله با بن بست روش مناسبی نیست چون اگر Time Out

کم باشد ممکن است قبل از روزن بن بست، Time Out اتفاق بیفتد. اگر Time Out زیاد باشد

ممکن است اتلاف منابع داشته باشیم.

گفتیم زمانبندها یا بهاجم هستند یا محافظه کار.

زمانبند B2PL مهاجم هستند. چون به قاعده ای که گفتیم برای تضمین توالی پذیری درایی بزرگ بود

باکه از سقوط کردن تراکنش اجتناب نمی کند و بن بست رخ می دهد و روش حل بن بست هم

Abort کردن است. زمانبندهای محافظه کار، مکانیزم که برای جلوگیری از Abort شدن می دهند

محافظه کار 2PL (Conservative 2PL
Static 2PL)

زمانبندی است که مبتنی بر قواعد 2PL راهکاری ارائه می کند که بن بست رخ ندهد و از Abort شدن

تراکنش ها جلوگیری شود.

یکی از راهکارهای جلوگیری از بن بست اخذ تمام منابع در ابتدای تراکنش. در این روش تمام قفل ها

در ابتدا اخذ می شود و پس از اتمام کارش باید تمام داده قفل آن را آزاد می کند. گفتیم T.M

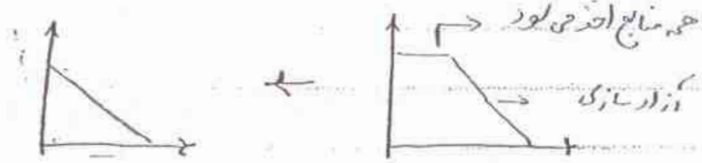
دستورات را برای sch می فرستد. sch از کجا بداند چه قفل هایی را لازم دارد؟

یک راه این است. sch تمام دستورات را تا Commit بگیرد و وقتی Commit آمد قفل ها را

بگیرد و اجرا کند. در این روش wait خیلی زیاد می شود و اتلاف منابع داریم

. T.M مدیریت داده تطبیقی برای برخی اجراهای خاص به sch می دهد T.M همراه با start
R-Set و W-Set را به sch می دهد. مجموعه تمام قلم داده هایی است که R-Set
می خواهد بخواند

تراکنش وقتی همه قفل ها را اخذ کرد شروع به اجرا می کند در نتیجه بن بست نداریم و سقط رخ نمی دهد بین
این روش محافظ کار خواهد بود.



محافظه کار بودن و تهاجم بودن نسبی هستند. انتهای افراطی محافظه کار بودن یک روش، اجرای سری در سقف است
و انتهای افراطی تهاجم بودن، certifies هستند

زمانبندها می توانند خوش بینانه و بدبینانه باشند

خوش بینانه می پندارد که تعداد کمی از تراکنش ها قلم داده مشترک دارند... از اینجا انتظار داریم که اگر قلم داده
مشترک وجود دارد، ندارد مگر اینکه خلافش ثابت شود. در بدبینانه عکس این است.

همه زمانبندها خوش بینانه و بدبینانه دارند. مثلاً 2PL خوش بینانه و بدبینانه داریم.

تمرین: تمام روش های سام 2PL مطالعه شود.

Subject : Date

آیا اجرای زیر توسط C2PL قابل تولید است ؟

$W_2(x)\ R_1(x)\ R_2(y)\ C_1\ C_2$

بله هم توسط C2PL و هم توسط B2PL قابل تولید است . ولی اجرای فوق ترمیم پذیر نیست

در حالی که گفته بودیم زمانبند باید اجراهای پی در پی RC و حداقل ترمیم پذیر را تولید کند . پس باید روش

را اصلاح کنیم . گفتیم ST ⊂ ACA ⊂ RC پس اگر زمانبند اجرای ST تولید کند اجرا

ترمیم پذیر هم خواهد بود

: S2PL

$$\begin{cases} w_j(x) < \underset{a_j}{C_j} < O_i(x) & \text{در اجرای ST} \\ w_j(x) < w_jU(x) < O_iL(x) & \text{می دانیم :} \end{cases}$$

$$\Downarrow$$

$$w_j(x) < \underset{a_j}{C_j} < w_jU(x) < O_iL(x) < O_i(x)$$

این یعنی تراکنش بعد از c یا A ، قفل را آزاد می کند . این یک محدودیت جدید است و به هیچ وجه

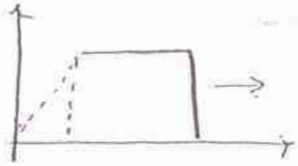
قاعده ۲ و ۳ را نقض نمی کند فقط آزادسازی قفل ها را باز هم به تعویق می اندازد .

در قاعده ۳ ، زمانبند از کجا می داند که تمام قفل های لازم را گرفته یا نه ؟

در این روش این مشکل زمانبند هم حل می شود یعنی وقتی a_i یا C_i می آید می داند که دیگر قفلی لازم نیست

(گفتیم W-set و R-set فقط برای اجرای حافظ کار به sch. فرستاده می شود

ولی در دنیای واقعی اینگونه نیست و برای ما لازم است این اطلاعات به sch. داده می شود)



همه قفل‌ها یکباره و بعد از C یا a آزاد می‌شود

قضیه: اگر آزادسازی قفل‌ها در روش 2PL بعد از C یا A باشد آنگاه اجراها نقص خواهد بود
(به عنوان تمرین اثبات را با جزئیات بنویسید)

برای داشتن اجراهای ترمیم‌پذیر، آکد و ... در خود اجراها به ترتیب آمده باید انجام شوند ؟؟!!

پیاده‌سازی 2PL

اگر یک جدول Lock (Lock Table) و پرچم‌های Lock و UnLock داشته باشیم
می‌توانیم پیاده‌سازی را انجام دهیم:

Trans. Id	Operation	Data Item
T_1	W	x
T_2	R	y
T_1	R	Z
~~T_5~~	~~R~~	~~x~~

انجام نمی‌شود چون T_1 روی x قفل W زده است

دستور UnLock منجر به حذف رکوردی از این جدول خواهد شد. Unlock(T_{id}, DI)

دستور Lock منجر به درج " " " " " " Lock(T_{id}, Op., DI)

ارتقاء قفل هم در این جدول انجام می‌شود.

Subject :   Date

مشکلی که در روش 2pl وجود دارد تعداد زیاد قفل است. در پایگاه داده برای هر دیتا آیتم، یک قفل باید بگذاریم (جدول قفل Lock Table) بسیار بزرگ خواهد شد و مدیریت آن نیز مشکل و دشوار است روش 2pl به گونه‌ای این مشکل را حل می‌کند.

MGL (Multi Granularity Locking)

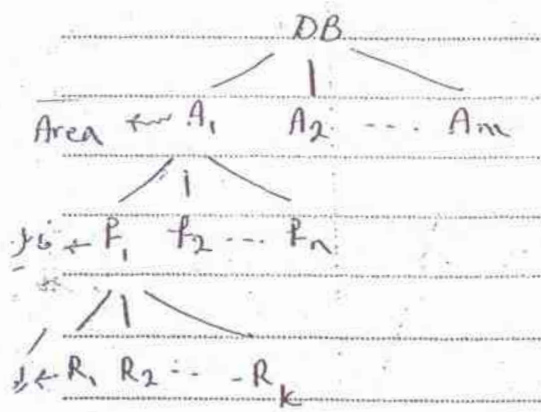
قفل گذاری چند گانه‌ای

سعی می‌کند از طریق کاهش تعداد قفل‌ها کارایی را افزایش دهد البته یک ایراد دارد به این شرح خواهیم داد و آن این است که عمل است در اثر این کار کاهش همزمانی می‌شود.

فرض کنیم فایلی داریم که در آن 10000 رکورد است می‌خواهیم این فایل را بخوانیم مثلاً می‌خواهیم اسامی کل دانشجویان را بخوانیم. در روش 2pl باید 10000 تا قفل بزنیم ولی در روش MGL فقط یک قفل بزرگتر روی کل فایل می‌زنیم. متناسب با اندازه تکه داده قفل می‌زنیم. گفتیم تکه داده؟ می‌تواند اندازه‌ای از فیلد تا کل دیتابیس داشته باشد.

چند نوع قفل داریم؟ به اندازه تعداد اعمال دیتابیس R و W و Inc و Dec و ...

چند اندازه قفل داریم؟ به تعداد انواع دانه‌بندی؟



بر اساس Lock Type Graph مشخص می‌شود در چند سطح قفل لازم داریم مثلاً در سطح دیتابیس، Area، فایل و رکورد و ...

«Lock Type Graph»

MICRO

زمانیکه وقتی می‌خواهد براین اساس قفل بزند از جدول قفل، L.T. Graph را بر Instance می‌سازد

برای Instance ← Lock Instance Graph می‌گویند LIG. نشست یافته LIG

است. مثلاً به جای f_1، اسم فایل با قفل‌هایی که به آن خورده قرار می‌گیرد. کاملاً!! 

* گفتیم شکل این روش کاهش همروندی است. مثلاً تراکنش T_1 می‌خواهد از R_1 تا R_{k-100}

را بخواند و k، برابر 10,000 است. به جای اینکه روی R_1 تا R_{k-100} قفل بزند روی کل

فایل قفل می‌زند. حالا تراکنش T_2 با R_k کار دارد ولی R_k قفل است پس همروندی

کاهش می‌یابد *

تمرین: روش MGL چه مزایا و معایب دیگری دارد؟

از اینجا به بعد همه صحبت‌ها روی LIG است!!

* روی هر سطحی (دانه‌بندی) قفل زده شد به آن قفل صریح می‌گوییم. مثلاً T_1 روی f_1 برای

R قفل زده پس روی f_1 قفل صریح داریم. ولی رکوردهای موجود در f_1 دارای قفل ضمنی

هستند یعنی مستقیماً روی رکوردها قفل نخورده ولی چون پدرشان (f_1) قفل است این‌ها نیز

قفل هستند. بنابراین T_1 زمانی می‌تواند روی قلم داده‌ای عملیات انجام دهد که قفل مربوط به

آن عملیات را به طور صریح یا ضمنی روی قلم داده زده باشد.

فرض کنیم تراکنش T_i روی f_1 می‌خواهد قفل R بزند و تراکنش T_j می‌خواهد قفل W

بزند روی رکورد R_k در f_1. T_i روی f_1 قفل را زد. حالا T_j می‌خواهد قفل W

Subject :   Date

راه‌حل: در Lock Table فقط هم داده‌ای به طور صریح قفل شده وجود دارد. T_j از کجا بفهمد که طوقه‌ی R_k قفل شده است؟

سناریوی ۱:

$T_i \rightarrow R \rightarrow F_1$

$T_j \rightarrow W \rightarrow R_k$

سناریوی ۲: T_i روی F_1 قفل R زده حالا T_j می‌خواهد روی A_1 قفل W بزند.

$T_i \rightarrow R \rightarrow F_1$

$T_j \rightarrow W \rightarrow A_1$

اگر بخواهیم هنگام قفل زدن تمام اجداد و فرزندان را چک کنیم Over head بسیار بالایی خواهد داشت

برای حل این مشکل قفل جدیدی مطرح شد به نام قفل خیالی!

قفل خیالی Intention Lock:

عوض کردن در درخت به‌نتیجه ایست پس مجبوریم به کمک اطلاع رسانی کنیم که در جایی از این درخت (رو به بالا یا به پایین) قفلی وجود دارد. قفل خیالی R و W و Inc و Dec و ... داریم.

iR

iW

وقتی هم داده‌ای قفل می‌شود، قفل خیالی برای پدران زده می‌شود.

iR DB

با این روش نیاز به جستجو ندارد و سناریوی دوم به راحتی حل می‌شود

iR A_1

در سناریوی اول وقتی T_j می‌خواهد روی R_1 قفل بزند همه قفل خیالی را

R T_i F_1

R_1

عکس آن یعنی F_1 می‌گیرد و در آنجا می‌بیند که روی F_1 قفل وجود دارد، پس

برمی‌گرده و قفلی را که روی R_1 زده باز می‌کند. برای این به مجبور نشویم برگردیم و قفل‌ها را باز کنیم

به ترتیب قفل از بالا به پایین خواهد بود (یعنی اول روی ایدال قفل خیالی می‌زند و بعد روی خود قلم داده قفل می‌زند.

⊕ پس قاعده این است که به ایدال قفل خیالی بزنیم و روی قفل کردن هم از بالا به پایین باشه ⊛

برخورد قفل‌های خیالی بلامانع است. ☺ (خیالی نسبت خیالیه! ☺)

ولی برخورد قفل صریح و قفل خیالی مشکل‌زا است و خطرناک می‌شود.

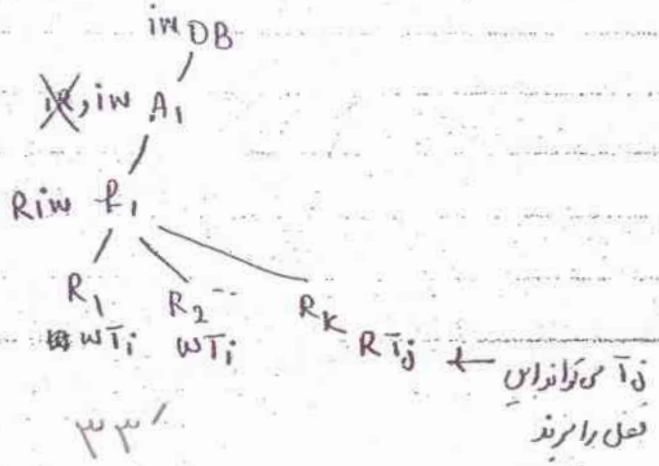
سناریوی سوم: تراکنش T_i می‌خواهد کل رکوردهای فایل F_1 را بخواند و از R_1 تا R_{10} را می‌خواهد بنویسد

روش اول: قفل R روی F_1 به W تبدیل شود. در این حالت ممکن است بعضی از قفل‌های R روی رکوردهایی گیر افتاده باشند که به تداخل می‌خورند.

روش دوم: روی F_1 قفل R بخورد و روی ۱۰ تا رکورد قفل W بخورد. در این صورت به F_1 چه قفلی بزنیم؟ میدانیم داریم R می‌کنیم یا W؟

قفل RiW را معرفی می‌کنیم، یعنی خودش قفل R دارد و با خیال هم دارد.

در اینجا هم قفل‌ها از بالا به پایین زده می‌شوند.



۳۳

Subject : Date

جدول تناسب قفل‌ها به صورت زیر در می‌آید :

	R	W	iR	iW	RiW
R	Y	N	Y	N	N
W	N	N	N	N	N
iR	Y	N	Y	Y	Y
iW	N	N	Y	Y	N
RiW	N	N	Y	N	N

* این پروتکل در روش MGL در قالب ۴ قاعده بیان می‌شود (فرض می‌کنیم اسم G، LIG است و می‌خواهیم قفل داده x را بخوانیم یا بنویسیم)

۱. اگر x ریشه درخت G نیست برای اخذ قفل R_x یا iR_x ابتدا باید روی والد x، قفل iw_x یا iR_x اخذ شود (لذا به این علت نیاز است قفل RiW را داشته باشیم) (اخذ قفل از بالا به پایین)

۲. برای اخذ قفل w_x یا iw_x ابتدا باید روی والد x قفل iw اخذ شود (اخذ قفل از بالا به پایین)

(اگر قفل بر روی خانه داده x باشد قفل صریح دارد و اگر روی والدی از x باشد روی x قفل ضمنی داریم)

۳. برای انجام عمل روی x باید قفل صریح یا ضمنی متناسب با آن عمل روی x وجود داشته باشد.

۴. آزادسازی قفل از پایین به بالا انجام می‌شود یعنی اول قفل صریح و بعد قفل‌های ضمنی سطوح بالاتر (یعنی اول خود قفل آزاد می‌شود و بعد اثر آن)

MICRO

روش های غیر قفلی

دو روش اصلی وجود دارد:
- TSO (Time Stamb Ordering)
- SGT (Searching Graph Testing)

این روش ها از قفل استفاده نمی کنند و فقط به صحت و محافظه کار دارند.

این روش ها اول مفهوم قفل را ندارند؛ توالی دستورات موجود را با به روش های دیگر مشخص کنند روش TSO از مهر زمانی (Time Stamb) استفاده می کند و روش SGT از گراف استفاده می کند.

روش های غیر قفلی

دو روش پایه ای: TSO یا Time Stamp Ordering | ترتیب مهر زمانی

SGT | Segmentation Graph Testing | آزمون گراف

هم روش های زمانبندی هم گونه تعاملات و هم گونه حافظه کار . فقط

نقطه مشترک این دسته روش ها . از مفهوم قفل استفاده نمی کنند . در روش های قفلی قفل ابزاری بود

برای مراقبت از ورود تراکنش ها . در این روش ها از شیوه متفاوتی برای مراقبت و ورود ترتیب تراکنش ها استفاده می شود

مزیت این روش ها : چون از مفهوم قفل استفاده نمی کنند مفهوم بن بست در این روش ها وجود ندارد .

عیب این روش ها : نسبت به روش های قفلی overhead بیشتری دارد .

* این روش ها هم نوع Optimistic و Pesimistic دارند

* روش TSO کاربردی تر از روش SGT است .

* دسته دیگری از روش های زمانبندی ارائه شده که ترکیب روش های قفلی و غیر قفلی است که این روش ها به روش های

ترکیبی یا Hybrid گفته می شود در این روش ها با ترکیب خواهیم توانست به دلیل اینکه انعطاف بیشتری دارند

هزینه بیشتری را فراهم می کند و سربار بیشتری دارد . این روش ها بیشتر در سیستم های توزیع شده اجرا می شوند .

Date

روش ترتیب مهرزمانی TSO

اشکال متفاوتی از TSO ارائه شده، ما در اینجا به Basic TSO می‌پردازیم به اسم BTSO

گفتیم روشی که زمانبندی باید پی‌درپی‌پذیری را فراهم کند. در تئوری پی‌درپی‌پذیری گفتیم اعمال برخوردار به گونه‌ای مرتب شوند که گراف SG اجرای سابقه فاقد حلقه باشد (شرط لازم و کافی)

در این روش برای رسیدن به این منظور، از قاعده مهرزمانی استفاده می‌کند. قاعده مهرزمانی می‌گوید

برای هر دو عمل برخوردار $p_i(x), q_j(x) \in H$ ، $p_i(x) \in T_i$ ، $q_j(x) \in T_j$ ، p_i قبل از q_j

خواهد بود ($p_i < q_j$) اگر و فقط اگر $Ts(T_i) < Ts(T_j)$

← شرط لازم و کافی

با این قاعده، اجرای پی‌درپی‌پذیر تضمین می‌شود.

در این روش به هر تراکنش یک مهرزمانی یکتا تخصیص می‌یابد (توسط مدیر تراکنش TM)، $T_s(T_i)$ مهر Time Stamp یکتا است که به تراکنش T_i تخصیص داده شده. این مهرزمانی انتهای تمام عملیات تراکنش قرار می‌گیرد، اگر تراکنش T_i زودتر از T_j به TM رسیده باشد مهرزمانی کوچکتری از T_j دارد. پس تمام اعمال برخوردار T_i و T_j، در صورتی T_i زودتر اجرا می‌شود، گراف فاقد حلقه خواهد بود و اجرا پی‌درپی‌پذیر می‌شود.

قضیه: اگر H سابقه‌ای باشد که به وسیله یک زمانبند ترتیب مهرزمانی تولید شده باشد آنگاه H پی‌درپی‌پذیر است

اثبات: فرض کنیم H سابقه تولید شده توسط زمانبند TSO است و در گراف SG(H)، لبه $T_i \rightarrow T_j$ وجود دارد یعنی $p_i(x) < q_j(x)$، طبق قاعده مهرزمانی می‌توانیم نتیجه بگیریم که $Ts(T_i) < Ts(T_j)$

اعمال برخوردار از T_i و T_j.

Subject :   Date

برهان خلف: فرض کنیم در SG(H) حلقه‌ای مثل $T_i \to T_j \to \dots \to T_k \to T_i$ وجود دارد. پس

نتیجه می‌گیریم $TS(T_i) < TS(T_j) < \dots < TS(T_k)$ یعنی $TS(T_i) < TS(T_i)$

فرض ما این است که TM برای هر تراکنش یک TS یکتا تولید می‌کند. بنابراین فرض خلف باطل و

حکم ثابت است. ■

* تمام روش‌های زمانبندی باید چند مورد را مورد توجه قرار دهند:

- ما باید توانایی برای هر روش قرار می‌دهیم، روش زمانبندی با نحوه‌ی نقض این قواعد را تشخیص دهد (شوند)

- Hand shaking

- زمانبند بعد از اینکه اجراهای پی در پی ترتیب را می‌دهد باید اجرای آن مراحل که هم ترتیب هم دهد.

فرض کنیم دو تراکنش T_i و T_j داریم که $TS(T_i) < TS(T_j)$ عمل p_i وجود دارد. q_j وارد می‌شود.

عمل q_j سه راهکار دارد: Run ، Wait ، Abort. زمانبند چه تصمیمی برای این دستور

می‌گیرد؟ فرض می‌کنیم $q_j(x)$ با عملی مثل $p_i(x)$ برخورد دارد که هنوز به زمانبند نرسیده. زمانبند نمی‌داند $p_i(x)$ ای

وجود دارد. چه رفتاری نشان دهد؟ Abort نمی‌تواند کند چون هنوز مسئله‌ای وجود ندارد. Wait هم نمی‌تواند بکند

چون اصلاً خبر ندارد که $p_i(x)$ قرار است بیاید. تنها کاری که می‌تواند بکند Run است. $q_j(x)$ را اجرا می‌کند.

برای اجرا Data Manager وارد می‌شود. چون $TS(p_i)$ کوچکتر از $TS(q_j)$ است p_i حالا $p_i(x)$

باید قبل از q_j اجرا می‌شد، حالا اگر p_i را اجرا کند قاعده هر زمانی نقض می‌شود و اجرای پی در پی

نخواهد بود. پس حالا یا $q_j(x)$ یا $p_i(x)$ باید Abort شود. $p_i(x)$ است که دیر رسیده و جا مانده

$P_i(x)$ عمل درخواست می‌گوییم. برای حفظ قاعده هر زمانی، الگوریتم TSO عمل درخواست را Abort می‌کند. درنتیجه تراکنش T_i سقط می‌شود.

Abort شدن هزینه دارد پس برای A کردن اعمال درخواست باید بتوانیم آن را در وقت تشخیص دهیم

تعریف عمل درخواست: می‌گوییم عمل $P_i(x)$ درخواست است اگر $TS(P_i) < TS(q_j)$ و $P_i(x)$ و $q_j(x)$ برخورد دارند و $q_j(x)$ زمانبندی شده باشد (یعنی DM ارسال شده باشد)، به $P_i(x)$ عمل درخواست می‌گوییم.

* برای حفظ قاعده TSO، عمل درخواست Abort می‌شود. تراکنش Abort شده، شروع مجدد انجام می‌دهد و این بار هر زمان برخورد می‌خورد.

* BTSO که زمانبند مهاجم است.

نحوه تشخیص عمل درخواست

برای هر قلم داده مثل x دو متغیر نگاه می‌داریم:

Max_r_scheduled[x]: ماکزیمم TS ای که R کرده است

Max_w_scheduled[x]: ماکزیمم TS ای که W کرده است

می‌خواهیم تشخیص دهیم که آیا TS ای که من با آن برخورد دارم از بین تراکنش‌های باقی، کمترین TS ای که اجرا شده را نگه داریم. چرا r و w؟ چون اعمال برخورد دارند

فرض کنیم عمل $R_i(x)$ باید، می‌خواهیم بفهمیم درخواست است یا نه؟ با Max_w_scheduled[x] مقایسه

Subject : Date

می‌کنیم. اگر TS_i بزرگتر از Max-w بود یعنی دیر رسیده ولی جدید است و می‌رود برای اجرا. در غیر این صورت دیر کرده است و A می‌شود.

فرض کنیم $W_i(x)$ باید هم با Max-w-sch(x) و هم با Max-R-sch(x) مقایسه می‌شود اگر از هر دو بزرگتر بود می‌رود برای اجرا در غیر این صورت دیر کرده است.

• اگر عمل $P_i(x)$ وارد شود با $Ts(P_i)$ با Max-q-scheduled(x) که p و q با هم وجود دارند مقایسه می‌شود. دو حالت ممکن:

$Ts(P_i) > Max\text{-}q\text{-}S(x) \Rightarrow$ مشکلی وجود ندارد و اجرا

$Ts(P_i) < Max\text{-}q\text{-}S(x) \Rightarrow$ P_i دیر رسیده است و Abort

پس در این روش برای هر قلم داده پایگاه داده دو متغیر تخصیص می‌دهیم و این یعنی Over head بالا.

در روش قفل گذاری مشکل دیر رسیدن چگونه حل شد؟ قفلی اخذ می‌شد قبل از این‌که دیگری قفلی را آزاد کند بنابراین عملی که باید دیرتر اجرا می‌شد خود به خود در حالت wait می‌ماند تا دیگری قفلی را آزاد کند.

نحوه پیاده‌سازی Hand Shaking

Hand Shaking در Data Manager برای تضمین این است که توالی‌ای را که زمانبند ایجاد کرده به هم نریزد. زمانبند هم باید روشی داشته باشد که از ارسال همزمان اعمال برخوردار به DM جلوگیری کند به عبارتی تا نتیجه اجرای عملی را نگرفته عمل دیگری را که با آن برخورد دارد به DM نفرستد.

برای این کار از دو متغیر R-in-Transit(x) و w-in-Transit(x) استفاده می‌کنیم. این دو متغیر برای هر قلم داده در نظر گرفته می‌شوند. R-in-Transit(x) تعداد Read‌هایی است که روی قلم داده x

برای اجرا به DM فرستاده شده w-in-Transit(x) هم‌مقدار w و روی x است

و هنوز تصدیق آن دریافت نشده است.

هر دستوری که زمانبندی می‌شود اگر R باشد R-in-T(x) و اگر W باشد W-in-T(x)

یکی واحد اضافه می‌شود و هرگاه Ack را دریافت کرد یکی واحد از متغیر مربوط کم می‌کند.

فرض کنیم $R_i(x)$ به زمانبند رسید اول چک می‌کند ببیند دیر رسیده است یا خیر؟ اگر نبود

باید چک کند که در DM عمل برخورد دارد با x وجود نداشته باشد برای این کار مقدار W-in-T(x)

را بررسی می‌کند اگر صفر بود یعنی عمل برخورد ندارد و وجود ندارد و گرنه ارسال می‌شود و در غیر این صورت

wait می‌رود

اگر $W_i(x)$ برسد برای اطمینان از اینکه در DM عمل برخوردی در حال اجرا نیست

هر دو متغیر W-in و R-in را چک می‌کند اگر هر دو صفر بود ارسال می‌شود در غیر این صورت wait

R-in-Transit(x) می‌تواند از صفر تا n مقدار بگیرد. چون R ها با هم برخورد ندارند و بطور همزمان

می‌توانیم n عمل Read داشته باشیم.

W-in-Transit(x) فقط می‌تواند مقدار صفر یا یک بگیرد.

R-in و W-in اگر یکی از آنها صفر نباشد یکی می‌تواند غیر صفر باشد ولی اگر یکی غیر صفر

بود دیگری حتما باید صفر باشد

Subject: Date

در روش همزمانی، تراکنش ها برای اینکه به DM برسند ممکن است wait شوند (در سایر روش های زمانبندی نیز). wait ابزاری بود برای [illegible] برای نگهداری این wait ها از صف استفاده می شود که تراکنش های منتظر در صف نگهداری می شوند.

بنابراین در روش TSO، صف $Q(x)$ برای هر قلم داده x استفاده می شود تا دستورات در انتظار اجرا روی x در این صف نگهداری شوند. در این صف اعمال برخوردار بر اساس مهر زمانی شان مرتب می شوند.

این صف خاصیت دیگری نیز دارد: ممکن است عمل $q_j(x)$ در صف قرار داشته باشد و $P_i(x)$ برسد. در این حالت $P_i(x)$ دیر رسیده محسوب نمی شود زیرا $q_j(x)$ هنوز به DM ارسال نشده است. در این حالت $P_i(x)$ در صف جلوتر از $q_j(x)$ قرار می گیرد.

مثال) فرض می کنیم $TS(T_i)=i$ باشد و

برای هر قلم داده x:

$$\begin{cases} Max\text{-}R\text{-}sch(x)=0 \\ Max\text{-}W\text{-}sch(x)=0 \\ R\text{-}in\text{-}Transit(x)=0 \\ W\text{-}in\text{-}Transit(x)=0 \end{cases}$$

دستورات زیر وارد شده اند:

$R_1(x)\ W_2(x)\ R_4(x)\ R_3(x)\ Ack(R_1(x))\ Ack(W_2(x))\ W_3(x)$

$R_1(x) \rightarrow TS(T_1)=1 > Max\text{-}W\text{-}sch(x)=0 \Rightarrow$ دیر رسیده نیست

اجرا $\Leftarrow$ $W\text{-}in\text{-}Transit(x)=0$ و صف خالی $\Leftarrow$

$\Rightarrow Max\text{-}R\text{-}S(x)=1$ و $R\text{-}in\text{-}T(x)=1$

- $W_2(x)$: $TS(T_2)=2 > Max\_w\_sch(x)=0$ } ⇐ دیرکرده نسبت و صف خالی نسبت
$> Max\_R\_sch(x)=1$

→ $R\_in\_T(x)=1$ ⇒ wait ⇒ | $W_2(x)$ |

- $R_4(x)$: $TS(T_4)=4 > Max\_w\_sch(x)=0$ ⇒ دیر کرده نیست

صف خالی نیست و عمل ناجور دارد در صف ۱ نیست ⇒ wait ⇒ | $w_2(x)$ | $R_4(x)$ |

مزیت دیگر نیست

- $R_3(x)$: $TS(T_3)=3 > Max\_w\_sch(x)=0$ ⇒ دیرکرده نیست ← صف خالی نیست

$R\_in\_T(x)=1$ ⇒ wait ⇒ | $w_2(x)$ | $R_4(x)$ | $R_3(x)$ |

[نیازی به تصحیح ترتیب R_4 و R_3 نیست چون ناجور دارند نیستند]

- Ack $(R_1(x))$: ⇒ $R\_in\_T(x)=0$ ← آیا می‌تواند سر صف را برای اجرا بفرستد ← $R\_in\_T(x)=0$, $w\_in\_T(x)=0$ ⇒

$w_2(x)$ می‌رود برای اجرا ⇒ $Max\_w\_sch[x]=2$
$w\_in\_T(x)=1$ | $R_4(x)$ | $R_3(x)$ |

- Ack $(w_2(x))$: ⇒ $w\_in\_T(x)=0$ ⇒ دستوری در صف است ⇐ بله می‌تواند بفرستد برای اجرا ؟

⇒ $w\_in\_T(x)=0$ ⇒ اجرا R_4 ⇒ $R\_in\_T(x)=1$
$Max\_R\_sch(x)=4$

آیا دستوری در صف است ← بله آیا می‌تواند برای اجرا فرستاده شود ؛ $w\_in\_T(x)=0$ ⇒ اجرا $R_3(x)$

چون TS آ بزرگتری است این را تغییر نمی‌دهد ← ⇒ $Max\_R\_sch(x)=4$

توجه: تمام دستورات موجود در صف، دیرکرده اند. اگر دیرکرده به صف نمی‌رود

نکته: اگر دستوری که قرار است اجرا شود مثلاً Rx، TS آن از Max-R-S بیشتر باشد، Max-R-S تغییر نمی‌کند.

Subject :   Date

$w_3(x)$: Max-R-sch(x) = 4 ⇒ عمل دیر رسیده است ⇐ Abort می‌شود

Max-w-sch(x) = 2

⇒ $R_3(x)$ هم Abort می‌شود

گفتیم زمانبند باید اجرای سریال‌پذیر و حداقل ترمیم‌پذیر ارائه کند. آیا TSO اجرای ترمیم‌پذیر می‌دهد؟

مثال زیر را در نظر بگیرید: آیا این اجرا توسط زمانبند TSO قابل تولید شدن است؟

$w_1(x)$ $R_2(x)$ $w_2(y)$ C_2

اجرای فوق سریال‌پذیر است و توسط BTSO قابل تولید است ولی ترمیم‌پذیر نیست. پس برای ترمیم‌پذیر بودن اجرا باید قوانین دیگری اضافه کنیم.

اگر زمانبند اجرای ST تولید کند اجرا ترمیم‌پذیر هم خواهد بود.

TSO محض (STSO)

برای ساختن اجرای محض باید طوری زمانبندی کنیم که: $w_i(x) < C_i$ (a_i) $< O_j(x)$

اگر تراکنش $w^{(x)}$ کرده تا زمانی که C یا A نکند هیچ $O_j(x)$ ای نباید اتفاق بیفتد.

برای این کار از همان w-in-T(x) استفاده می‌کنیم. مقدار w-in-T(x) تا زمانی که C تراکنش نیامده است صفر نمی‌شود (یا اینکه تراکنش Abort شود) به عبارتی یعنی با رسیدن A یا C تراکنش مقدار w-in-T(x) به صفر کاهش می‌یابد.

توجه کنید که مفهوم W-in-T(x) و R-in-T(x) با قفل تفاوت دارد. در این صورت خود روش های

مبتنی بر قفل می بودند. ولی این ها با قفل تفاوت دارد برای اثبات این مطلب مثالی مطرح می کنیم با روش های

قفلی قابل انجام نیست ولی با این روش قابل انجام است. (می خواهیم ثابت کنیم W-in-T و R-in-T

دقیقاً همین را قفل نمی کند):

STV

$R_2(x)\ W_3(x)\ C_3\ W_1(y)\ C_1\ R_2(y)\ W_2(z)\ C_2$

$TS(T_1) < TS(T_2) < TS(T_3)$

اجرای فوق توسط زمان سنج BTSO قابل تولید است. تمام اعمال برخوردار براساس مهر زمانی مرتب شده اند

و اجرای پی در پی پذیر است. اجرای فوق محض نیز هست.

ولی این اجرا قابل تولید توسط روش 2pL نیست. چون در روش 2pL، T_1 (T2) باید قفل x را آزاد کند

تا T3 بتواند آن را اخذ کند و از طرف دیگر وقتی قفلی را آزاد کند نمی تواند قفل دیگری را اخذ کند در اینجا اگر بعد از

$R_2(x)$ قفل x برای $W_3(x)$ آزاد شود $R_2(y)$ دیگر قابل اجرا نخواهد بود. تا $W_2(z)$. اگر هم

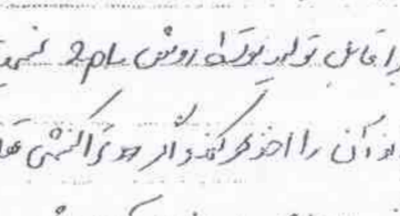
نخواهد قفل x را آزاد کند $W_3(x)$ اجرا نمی شود. پس اجرای فوق علیرغم پی در پی پذیر بودن قابل تولید

توسط روش 2pL نیست.

نتیجه: روش 2pL روش سخت گیرانه ای است و اجراهای پی در پی پذیری وجود دارند که نمی تواند تولید کند پس قاعده

مهر زمانی دسته بزرگتری از اجراهای پی در پی پذیر را پوشش می دهد.

قضیه: به عنوان تمرین این را ثابت کنید: برای اثبات ثابت کنید هر



$T_i \rightarrow T_j \in SG(H)$ که 2pL برای آن یا همان توسط

TSO قابل اجرا است. برای عکس قضیه مثال نقض کافی می باشد.

Subject: Date

تمرین: چگونه می‌توان در روش TSO زمانبندی‌های ACA و RC داشت؟

CTSO، TSO محافظه‌کار (Conservative TSO)

اگر می‌خواهیم زمانبند TSO محافظه‌کار داشته باشیم باید مشکل دیررسیدن را حل کنیم (برای جلوگیری از Abort کردن تراکنش‌ها)، باید اگر $q_j(x)$ قرار است به DM برود پیش از آن مطمئن شویم که $p_i(x)$ ای قرار است برسد، و $q_j(x)$ باید منتظر بماند.

اگر $TS(T_i) < TS(T_j)$ و p_i و q_j وجود داشته باشند و $p_i(x)$ نیامده است، عمل ارسال به DM اتفاق نیفتد (از $q_j(x)$ به DM ارسال نشود) - برای این کار TM باید اطلاعات اضافی (R-Set و W-Set) را از هر تراکنش به Sch. ارسال کند.

تمرین: چگونه می‌توان این راه حل را پیاده‌سازی کرد؟

ساده‌ترین روش: ① wait بعد از رسیدن هر دستور ایجاد کنیم. این امید به هر دستوری که قرار است زودتر بیاید، و بعد شروع به اجرا کنیم. اما حتی با وجود wait های طولانی نمی‌توان تضمین کرد که عمل دیررسیده نداشته باشیم. بعلاوه این کار سرعت اجرا را به شدت کم می‌کند.

@MICRO

10th Session 91,9,25

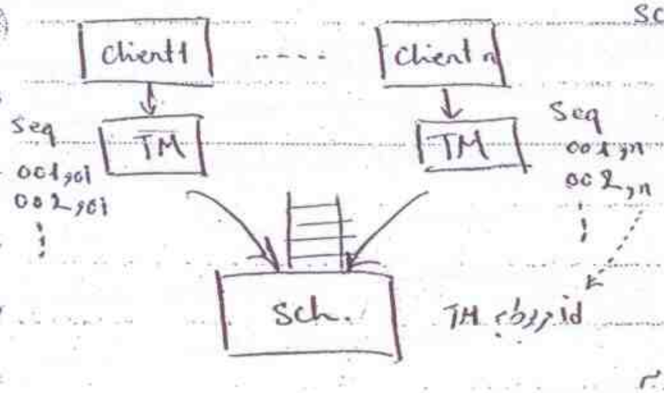
CTSO : محافظه‌کار

در روش BTSO اگرچه بالطبع A شدن تراکنشی که می‌شود دیر رسیدن آن کاملاً بهتر است برای جلوگیری از A شدن باید کاملاً سری پیاده‌سازی شود که تراکنش دیررسیده نداشته باشیم. متاسفانه تراکنشی که نمی‌توانیم تضمین کنیم دستورات آن را زودتر نفرستند. پس زمانبند باید کاری کند که تراکنش دیررسیده محسوب نشود

ساده‌ترین راه time wait است. اگر دستوری که دیر برسد می‌تواند در صف time wait جابجا شود. هرچه میزان این time wait بیشتر باشد احتمال اینکه دستور دیررسیده داشته باشیم کمتر است. با این روش Response Time خیلی بالا می‌رود و علاوه بر این بعضی دستورات کاملاً به دلیل time wait [illegible] خواهند شد.

راه دیگر این است که زمانبند به ازای هر تراکنش Ti اطلاعات تکمیلی R-set[i] و W-set[i] را از TM بگیرد.

روش سوم گونه‌ای کاملاً محافظه‌کار است با نگاهی دیگر. در این روش فرض کنید n تا کلاینت داریم که این Client که تراکنش‌ها را می‌فرستند و هر Client برای خودش TM دارد و هر TM، Sequence های خودش را دارد. TM که دستورات را به sch.



می‌فرستند و دور کاملاً ممکن است. Seq که برای شود برای جلوگیری از تکراری شدن، seq.

id هر TM به ابتدای seq های وارده از آن می‌چسبد. قبل از ورود به زمانبند صف un scheduled را داریم که پردازش‌های زمانبندی نشده در آن قرار می‌گیرند

(این روش بیشتر در سیستم‌های توزیع شده کاربرد دارد)

در چنین روش‌هایی، اگر بخواهیم به TM ها اولویت دهی کنیم، TM id را به قبل از seq می‌چسبانیم یعنی برای اولین TM داریم 01001 ، 01002 ، ... به این ترتیب seq های اولین TM همیشه TS کوچکتری نسبت به بقیه دارد پس اولویتش بالاتر خواهد بود. ولی اگر اولویت خاصی برایمان مطرح نیست، TM id را به انتهای seq می‌چسبانیم تا اولویت ایجاد نکند.

* در صف‌های UN sch ، دستورات تمام تراکنش‌ها به ترتیب تمرزمانی قرار می‌گیرند. پس همه دستورات (نه فقط دستورات برخورددار) براساس تمرزمانی مرتب شده‌اند

دستور سر صف بررسی می‌شود، اگر آماده اجرا بود زمانبندی می‌شود و برای اجرا به DM می‌رود. در غیر این صورت صف دست نمی‌خورد.

آماده بودن را چگونه بررسی می‌کند؟ توجه کنید در این زمانبند باید از A شدن جلوگیری شود

چند قاعده داریم (برای تشخیص آماده بودن):

1) از همه TM ها دستور رسیده باشد. (با این شیوه تضمین می‌شود دستوری که سر صف است از همه TS کوچکتری دارد)

2) تمامی دستوراتی که برخورددار احتمالی، مثلاً به DM ارسال شده‌اند باید اجراشان تمام شده باشد (برای اطمینان از اینکه ترتیبشان در DM به هم نمی‌ریزد) با همان متغیرهای R-in-T و W-in-T

در این روش دیگر به متغیرهای Max-R-sch و Max-W-sch نیازی نداریم. نسبت به BTSO سربار کمتری دارد.

* اگر یک TM ، دستوری نداشته باشد، روش قفل می‌شود، چون همه منتظر دریافت دستور از تمام TM ها می‌مانند. راه‌حل؟

راه‌حل: در چنین شرایطی TMهایی که دستور Null می‌فرستد و زمان‌بند متوجه می‌شود که بیکار هستند.

- فرض کنید نرخ رشد seqها در TMها متفاوت باشد و بعد از مدتی Gap بین این‌ها خیلی زیاد می‌شود و با زیاد شدن Gap، احتمال تراکنش‌هایی که نرخ رشدشان بالا است، خیلی زیاد می‌شود.

برای حل این مشکل هر در بازه‌ای زمانی مشخص شده TMها با هم sync می‌کنند مثلاً یکی تا 50 رفته یکی تا 10، بعد از sync هر دو از 100 شروع می‌کنند به .seq زدن.

- اگر ارتباط TM با زمان‌بند قطع شود، بعد از مدتی که از TM چیزی نرسید، زمان‌بند، TM را چک می‌کند و اگر پاسخی دریافت نکرد TM را از لیست TMها حذف می‌کند.

- هرگاه TM جدیدی وارد شد خودش را sync می‌کند و شروع به کار می‌کند.

* بهبودهایی که در این روش‌ها داده شده (روی سیستم‌های متمرکز): می‌تواند موضوع تحقیق باشد.

Serialization Graph Testing (SGT)

آزمون ترتیبی گراف

این روش نیز غیرقفلی است. در این روش یک گراف برای اجرا رسم می‌شود. در این گراف بررسی می‌کند که آیا حلقه وجود دارد یا خیر؟ و اگر حلقه داشته باشد می‌گوییم اجرا پذیر نیست.

گراف را می‌سازد با گراف SG و تفاوت عمده دارد با SG. گفتیم تمام گره‌ها، تراکنش‌های پایان یافته‌اند.

Subject : Date

ما گراف را برای اجرا می‌خواهیم و نه برای تراکنش‌های نهایی شده

نام این گراف را SSG می‌گذاریم. در این گراف تمام تراکنش‌های فعال گره‌ها دارند.

۴- تفاوت دوم این است که گره‌های نهایی شده به شرط اینکه اثری نداشته باشند از SSG حذف می‌شوند. چون الگوریتم‌های گراف اردر زمانی شان به تعداد گره‌ها وابستگی دارد و هر چه تعداد گره‌ها کمتر باشد کارایی بالاتر است.

اگر در این گراف حلقه ایجاد شود: مثل الگوریتم‌های پیشین و همچنین PCC نمی‌توانیم در ادامه، حلقه وجود خواهد داشت و اجرای دوباره پذیر نخواهد بود. بنابراین به محض ایجاد حلقه، تراکنش A حذف می‌شود و A شده‌ای بلافاصله گره و لبه‌های آن حذف می‌شود.

هر گاه عمل $p_i(x) \in T_i$ وارد می‌شود، زمانبند بررسی می‌کند که آیا تراکنش T_i گره متناظر دارد یا خیر و اگر خیر یعنی این اولین اکشن تراکنش T_i است پس یک گره T_i ایجاد می‌کند.

۲- برای تمام گره‌های T_j که متعلق به گراف هستند یال‌هایی از T_j به T_i $(T_j \to T_i)$ ترسیم می‌شود با این شرط که $q_j(x) \in T_j$ با $p_i(x)$ برخورد داشته باشد و $q_j(x)$ اجرا شده باشد.

۳- چک می‌کند که آیا در گراف SSG ایجاد شده حلقه وجود دارد. اگر حلقه وجود داشته باشد، نشان می‌دهد که اگر $p_i(x)$ اجرا شود، اجرای دوباره پذیر نخواهد بود پس $p_i(x)$، Reject و T_i ساقط می‌شود و بلافاصله گره T_i حذف و همه لبه‌های صادره و وارده به آن از گراف حذف می‌شود. اگر حلقه وجود نداشته باشد اجرای دوباره پذیر است و $p_i(x)$ می‌تواند زمانبندی شود. بررسی می‌شود که آیا اعمال برخورد دار قبلی ارسال شده به DM آیا Ack شان دریافت شده است (با استفاده از متغیرهای w-in-T و R-in-T)

MICRO

اگر Ack آمده باشد، دستور به DM ارسال می‌شود و هنگام سازی متغیرسازی مربوطه. در غیر

این صورت در صف قرار می‌گیرد تا نوبت اجرایش برسد.

↓ یکی از این متغیرها

P_scheduled است
P_in_Transit

در گام ۲، چگونه تشخیص می‌دهیم کدام تراکنش‌ها اعمال برخورد داری را اجرا کرده اند؟

باید اثری از اجراهای انجام شده را نگه داریم تا بتوانیم به این سوال پاسخ دهیم. برای هر تراکنش T_i، که R-sch[T_i] و W-sch[T_i] را نگه می‌داریم که داخل این لیست‌ها اقلام داده‌ای که خوانده و نوشته است نگه داری می‌شود. هرگاه دستوری R و W به DM ارسال شد این لیست‌ها Update می‌شود. هنگام ورود $P_i(x)$، اگر $P_i(x)$، R یا W-scheduled(x)، برای تمام تراکنش‌ها بررسی می‌شود تا ببینند آیا x در این لیست وجود دارد یا خیر؟ اگر وجود داشته باشد لبه ترسیم می‌گردد

کدام گره‌ها از گراف می‌توانند حذف شوند؟

۱. پایانی شده باشند

۲. در اجرا تأثیرگذار نباشند؛ یعنی در هیچ حلقه‌ای ظاهر نشود (ابداً)

هر گره در گراف چهار حالت می‌تواند داشته باشد: ○ ، →○ ، ○→ ، →○→

گره حالت اول را می‌توان حذف کرد چون ممکن است بعداً یک حلقه ایجاد شود. ✗

در گره حالت دوم: هرگز یالی وارد نخواهد شد پس امکان ایجاد حلقه وجود ندارد پس قابل حذف است. ✓

از گره حالت سوم، ممکن است با ورود تراکنش جدیدی، یالی خارج شود پس قابل حذف نیست. ✗

گره چهارم هم که قابل حذف است ✓

Subject : Date

پس برای حذف، گره‌ها باید یال ورودی داشته باشند.

اگر گره‌ای دو شرط فوق را داشته باشد قابل حذف است.

روش BSGT (Basic SGT) روشی بهم است چون ممکن است تراکنش‌ها را A کند.

گونه‌ی محافظه‌کار SGT : CSGT

باید از تشکیل حلقه جلوگیری شود. W-set و R-set به ازای هر تراکنش از TM به Sch. فرستاده

می‌شود پس Sch. در هر لحظه می‌داند چه چیز خواهد شد و چه بخشی اجرا شده است (با استفاده از R-sch و W-sch)

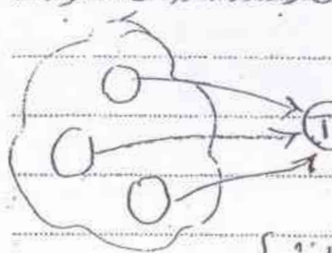
وقتی T_i وارد می‌شود $R\text{-}set(i)$ و $W\text{-}set(i)$ با $R\text{-}set(j)$ و $W\text{-}set(j)$ اشتراک داده می‌شوند و از همه می‌گیرد

که اشتراک دارد. T_i لبه می‌دهد (توجه کنید برخلاف BSGT که لبه‌ها هنگام اجرا می‌شد اینجا

قبل از اجرا لبه ترسیم شده و اجرا براساس لبه‌هاست). زمانیکه برای اجرای T_i، برای تمام تراکنش‌هایی

که به T_i لبه دارند مجموعه‌های R-sch و W-sch را بررسی می‌کند اگر اعمال برخورددار در این مجموعه

بودند یعنی اجرا شده‌اند و حالا T_i می‌تواند اجرا شود. مثلاً اگر



$P_i(x)$ از T_i می‌خواهد اجرا شود روی R-sch و W-sch

تمام گره‌های برخورددار دنبال x می‌گردد (توجه کنید ممکن است برخورد T_i و سایر تراکنش‌ها

روی یک قلم داده دیگری بوده باشد نه x) اگر قلم داده مشترک در R-sch و W-sch بود تراکنش

انتخاب می‌شود.

اجراهای تولید شده ممکن است ترمیم‌پذیر نباشند.

تمرین: چگونه می‌توان SGT ترمیم‌پذیر و ACA به دست آورد؟

@MICRO

Date

روش SGT کاملاً مبتنی بر تئوری بهینه‌پذیری اجرا را می‌سازد. روش قبلی سخت‌گیرانه‌تر بود و روش TSO محدودیت‌های بازهم کمتری داشت. روش SGT کاملاً ملایم و منعطف و منطبق با تئوری کامل است و تمام اجرای بهینه‌پذیری را می‌دهد و بسیار ضعیف زیادی دارد.

مثال: مثالی بزنید که TSO قابل اجرا باشد ولی SGO قابل اجرا نباشد

صفحه

Subject: 11th Session　　　　Date 91/10/2

یادآوری: متدهای ترمیم در برابر دو نوع Failure باید مقاوم باشند: 1. System Failure
2. Media Failure

ترمیم Recovery

بحث ترمیم به اطلاعات صحت و نظم اجرای ترمیم پذیرکاری ندارد (این کار زمانبندها است) مکانیزم‌های ترمیم فقط به این می‌پردازند که اگر تراکنش C کرد باید اثرش ماندگار شود و اگر A کرد اثرش نباید ماندگار شود

خطای رسانه: مثلاً بد سکتور خراب شدن‌ها ... تمرکز روی خراب شدن رسانه است

خطای سیستمی: محتوای حافظه داخلی از بین می‌رود و تراکنش‌ها state خود را از دست می‌دهند

تمرکز ما روی روش‌های ترمیم متمرکز کار می‌کنیم و در بین دو خطای مطرح شده تمرکز ما روی خطای سیستمی است. در مورد خطای رسانه انواع روش‌ها، RAID ها، DBMS ها، ابزارهای RMAN و ... برای مقابله با خطای رسانه است.

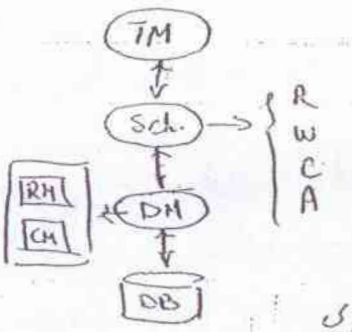
انواع خطاها
- خطاهای اپراتوری
- خطاهای تراکنشی: آن دسته از خطاهایی که نهایتاً تراکنش Abort می‌شود.
- خطاهای سیستمی: خطاهایی که محتویات حافظه اصلی یا بخشی از آن از بین می‌رود. مثلاً هنگام رفتن برق یا اتفاق افتادن یک حمله یا ویروس
- خطاهای رسانه: مربوط به Fail کردن مدیا است.

@MICRO

در مورد خطاهای سیستمی و رسانه خطاها باید نسخه دیگری از داده ها به طور مستقیم یا غیر مستقیم داشته باشیم تا بتوان با استفاده از آن خطا را رفع کرد. مستقیم یعنی خود داده ها را در جای دیگری داشته باشیم غیر مستقیم یعنی عملیات بر روی داده ها انجام شده را ثبت داریم

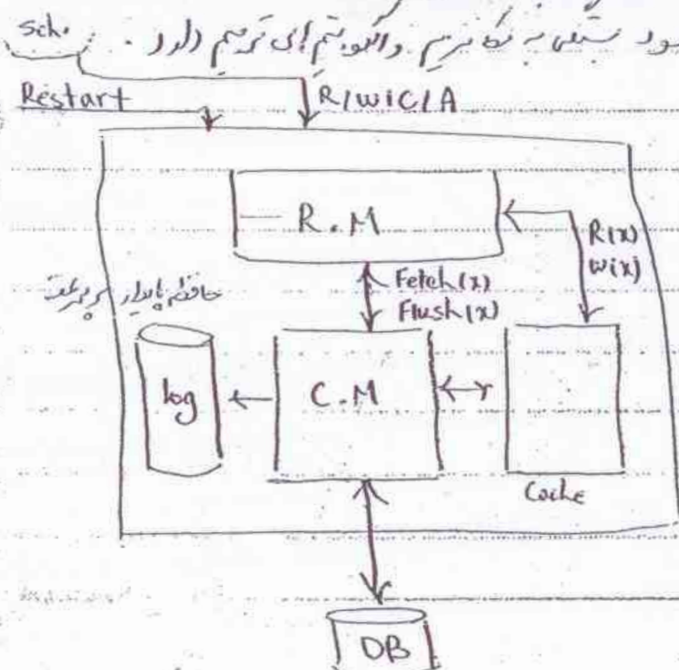
خطاهای رسانه در این بحث می رود که به طور مستقیم داده ها را داشته باشیم انواع مکانیزم های Backup گیری Cold ، Hot ، ... و RAID ، ... در این راستاست

خطاهای سیستمی از سایر خطاها پیچیدگی بیشتری دارند.



در مورد A هیچ عملی انجام نمی شود و به محض دریافت ، این دستور به DM فرستاده می شود ولی در مورد سایر دستورات یک تاخیر اتفاق می افتد.

Recovery Manag. مسئول ترمیم است و Cash Manag. مسئول آشکاری حافظه و آوردن و بردن داده ها. این دو با هم کاملاً همکاری دارند. این که داده ها باید چه زمانی در حافظه آشکاری شوند و به DB ارسال یا از آن Fetch شود بستگی به مکانیزم و الگوریتم های ترمیم دارد.



ورودی های DM از Sch. هستند (یعنی R) W، C، A یا Restart

(رخداد خطا)
Subject: Date

دستور Restart پایگاه داده را به آخرین وضعیت پایا شده (آخرین حالت سازگار) می‌برد. (آخرین حالت قبل از

حالت سازگار: حالتی است که در آن اثر تراکنش‌های پایا شده در پایگاه داده فیزیکی وجود دارد و تراکنش‌های

پایا نشده نیز هیچ اثری ندارند. یعنی اگر تراکنش پایا شده ای وجود داشته باشد که اثرات گم شده باشد

اثرات پیدا شود و اعمال شود. چه چگونه می‌شود اثر تراکنش پایا شده گم شود؟ — خطا $w_i(x)$ C $w_j(x)$

یا اثرات تراکنش پایا نشده هم نباید وجود داشته باشد، مثلاً تراکنش Abort کرده ولی پیش از پاک شدن

اثراتش اتفاقی (مثلاً Restart) رخ داده.

هنگام وقوع رویدادی که منجر به خطای سیستمی می‌شود باید به آخرین وضعیت سازگار برگردد؟ چگونه؟ هر قلم

داده حاوی اثر آخرین تراکنش پایا شده باشد

← Last Commited Value (LCV)

Restart این کار را می‌کند که هر قلم داده حاوی LCV خودش باشد. برای هر قلم داده x، x حاوی

مقدار نوشته شده (w) توسط آخرین تراکنش پایا شده باشد (آخرین نوشتن).

چون ترتیب اجرا sch. به DM می‌دهد، بنابراین قطعاً آخرین حالت درست قبل از خرابی،

سازگار خواهد بود (سازگاری را sch. تضمین کرده است)

فرض کنیم دستور R می‌خواهد انجام شود، به RM، CM اگر Fetch(x) را صادر نکند، CM

در Cache نگاه می‌کند که آیا x در Cache وجود دارد یا خیر. اگر وجود داشت به RM می‌دهیم می‌تواند

از Cache بخواند و RM می‌خواند در غیر این صورت عیناً از DB به Cache می‌آورد و ...

MICRO

دیگر اسلات خالی می‌بریم به RM می‌گوییم حالا اگر لازم باشد در log می‌نویسد که x خوانده شده است. و مانند حافظه فیزیکی است (حافظه پایدار برمی‌گردد)

برای دستور بعدی، Flush(x)، CM نگاه می‌کند به x آیا x در Cache وجود دارد، اگر بود به RM می‌گوید بنویس.

پس CM وظیفه خواندن و نوشتن داده‌ها در DB را بر عهده دارد و CM می‌خواهد این کارها را با حداکثر کارایی انجام دهد. از طرفی حافظه Cache محدود است. اتفاقی که می‌افتد این است که ۱) اثرات برخی از تراکنش‌ها که نهایی شده در Cache باقی می‌ماند (CM برای کارایی بیشتر بلافاصله به دیتابیس فیزیکی منتقل نمی‌کند) ۲) اثرات تراکنش‌های نهایی نشده ممکن است به فیزیکال دیتابیس منتقل شده باشد مثلاً برای اینکه اسلاتی را خالی کند اسلاتی را که حاوی اثرات تراکنش نهایی نشده بوده به DB فیزیکی برده. حالا اگر اتفاق رخ دهد، در DB فیزیکی اثرات تراکنش نهایی نشده را داریم و برخی اثرات تراکنش نهایی شده را نداریم. مکانیزمی که برای این است به این DB درست و داخل رو ارسال کنیم.

DM برای مقابله با این مشکل دو استراتژی اساسی را می‌تواند پیاده‌سازی کند، ۱) درجا Implace ۲) برون از جا Outplace

در حالت Implace، مقدار داده روی خودش نوشته می‌شود

در حالت Outplace، مقدار داده update شده روی حافظه دیگری نوشته می‌شود و کپی‌ای بین داده اصلی و داده بروزرسانی شده وجود دارد

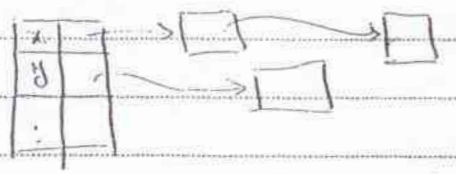
در Implace، نیاز به حافظه کمتری داریم و

در حالت Outplace، برگشت دوره‌ای سوال راحت‌تر است و با عملکرد لاگ تجزیه می‌توان سوال را اثبات کرد

Subject : Date

عیب روش Outplace ، مصرف بالای حافظه است . اصطلاحاً به حالت Outplace، Shadowing

می‌گویند . در Shadowing یک دایرکتوری از قلم داده‌ها وجود دارد که به آخرین مقادیر اشاره دارد .



اما این زنجیره طولانی شده و پیگیری آن سخت می‌شود . در

این حالت یک دایرکتوری جدید ساخته می‌شود که به آخرین مقادیر

اشاره می‌کند و دایرکتوری جدید Master می‌شود و قبلی Shadow .

بعد از مدتی که چند دایرکتوری ساخته شد Shadow های قبلی پاک شده و حافظه آزاد می‌شود

مزیت بازیابی Inplace این است که استفاده از log . پس باید log چیزی نوشت تا امکان

اصلاح DB فعلی وجود داشته باشد .

وقتی قرار است قلم داده‌ای Fetch شود : Fetch(x)

- اگر x در Cache نیست یک slot خالی برای x فراهم کن

- x را از DB بخوان و در اسلات مورد نظر قرار بده

- فیلد dirty را مساوی صفر قرار بده . این فیلد نشان دهنده این است که آیا فیلد موجود در Cache

با DB متناظر است یا خیر و این هنگام پیدا کردن اسلات کاندید برای خروج از Cache کاربرد دارد.

اگر مقدار فیلد Cache و DB متناظر باید برابر باشد dirty=0 است و این فیلد می‌تواند از حافظه

خارج شود (بهترین فیلد برای جایگزینی است) .

- فهرست Cache را به‌هنگام کن . (فهرست نشان می‌دهد چه قلم داده‌ای در Cache وجود دارد)

MICRO

Flush(x):

- Fetch(x)

- dirty = 1

- مقدار x بازنویسی شود

CM فقط می‌خواهد با کارایی اسلات‌ها را ببرد و بیاورد (از این پیام‌ها استفاده کنیم)

RM دو دستور اختصاصی خودش را دارد: pin(C)

Unpin(C)

اگر RM بنا به ملاحظات ریکاوری تصمیم بگیرد اسلاتی از Cache جابجا نشود آن را pin می‌کند

و تا زمانی که Unpin نکرده CM نمی‌تواند آن اسلات را از Cache خارج کند.

R.M. (Recovery Manager)

- RM-R: RM-R(Tid, data item)

- RM-W: RM-W(Tid, data item)

- RM-A: RM-A(Tid)

- RM-C: RM-C(Tid)

- Restart

خواهیم گفت در هر یک از این عملیات چه کارهایی باید انجام شود تا آنچه می‌خواهیم اتفاق بیفتد

Subject : Date

فرض: اجراهایی که به sch می‌دهیم ازهم‌پذیر است و [illegible] است. کمترین نوع [illegible]

ترتیب اجرای w ها همان ترتیبی است که باید نهایی شود مثلاً c_j ; w_j ; c_i ; w_i یعنی همیشه آخرین نوشته باید

باید کار شود. آخرین ترتیب همان ترتیب نوشتن است.

* هر چه اجراهای زمانبند محدودتر باشد مکانیزم ترمیم ساده‌تر است.

در اینجا ساده‌ترین حالت محض یعنی اجرای محض را در نظر می‌گیریم.

Befor Image: قبل از w یک کپی از آن قلم داده می‌گیریم تا در صورت A کردن تراکنش بتوانیم مقدار x را برگرداند (برای برگرداندن اثر تراکنش A شده)

log: حافظ پایدار پر برکت است که پشتیبانی دارد و از بین نمی‌رود. مجموعه‌ای است از رکوردها که جریان اجرا را نشان می‌دهد و ساختارش را می‌توان به صورت [T_i, x, v] نشان داد. فایل log یک فایل با ساختار ترتیبی Sequential است. نظم فایل ترتیبی می‌تواند زمانی باشد یا مکانی. در اینجا ترتیب فایل log ترتیب زمانی است. هر تراکنشی که پایین‌تر است آخرین عمل را انجام داده و اگر روی آن باید بماند.
نوشتن

بحث log نویسی مبحثی است که می‌تواند موضوع تحقیق باشد.

اساساً دو روش برای log نویسی وجود دارد:

log منطقی: منطق عملیات بدون جزئیات نوشته باشد. هنگام استفاده از این log یک مفسر لازم است که متناسب با شیء است، داده‌ای log را به گام‌های اجرایی تبدیل کند. حسن این روش این است که مصرف حافظه پایین می‌آید و log نویسی تسریع می‌شود. عیبش همان مفسر است که و روش [illegible]

MICRO

قابل تک شاخص و چند شاخص (ذخیره و بازیابی اطلاعات) مطالعه شود.

- Log فیزیکی: تمام قدم های طی شده را به ترتیب می نویسد، هر چه را که واقعاً انجام داده در Log می نویسد سازه به تغییر ندارد ولی حافظ زیادی مصرف می کند و نوشتن Log سخت می شود.

بیشتر از Log فیزیکی استفاده می شود.

در Log چه چیزی نوشته شود؟

1, write

2, C و a : در Log ها علاوه بر این لیست های دیگری نیز وجود دارد از جمله Aborted List ، Commited list و Active List مثلاً از Log می رسیم به W_i زمانی W_i ماندگار می شود که C_i متعلق به Commited list باشد.

* خواندن اثری ندارد و بنابراین نیازی به نوشتن در Log ندارد.

* Restart هر چیزی در Log می نویسد از Log استفاده می کند تا پایگاه را به آخرین حالت سازگار ببرد.

Redo / Undo

دو مفهوم بازچینی (Redo) و واچینی (Undo) دو مفهوم اساسی در ترمیم هستند. Redo: کاری اگر لازم باشد دوباره تکرار می شود. Undo: کار انجام شده را برمی گرداند.

از Redo برای اعمال اثرات تراکنش های پایان یافته استفاده می شود که ماندگار نشده اند.

بازسازی

از Undo برای از بین بردن اثرات تراکنش های پایان نیافته در پایگاه داده استفاده می شود و ماندگار شده اند.

Subject: Date

با توجه به جریان نویسش ها، R.M. ترکیبی از Redo ها و Undo ها را انجام می‌دهد متناسب با LCV برای ...

اساساً RM چهار حالت را می‌تواند در ارتباط با Redo و Undo اختیار کند.

۱. Redo/Undo

۲. Redo/non Undo

۳. non Redo / Undo

۴. non Redo / non Undo

وقتی non Undo داریم، یعنی RM مکانیزمی پیاده‌سازی کرده که تراکنش تا زمانی که C نکرده، اثرات آن به جزئیات دیتابیس منتقل نمی‌شود. این حالت را با Pin و Unpin انجام می‌دهد.

وقتی non Redo داریم، یعنی RM شرطی گذارده که به محض C کردن همه اثرات تراکنش به دیتابیس منتقل می‌شود.

در non Redo / non Undo، RM به CM خیلی سخت می‌گیرد و بیشترین وظیفه را بر CM دارد.

ما حالت اول را در نظر می‌گیریم چون کارایی بیشتر است و RM دیگر به CM وظیفه ندارد و راه مکانیزمی که برای Redo و Undo در نظر می‌گیرد.

مکانیزم Redo/Undo از Log استفاده می‌کند و بر اساس داده‌ای log تصمیم می‌گیرد کجا Redo کند و کجا Undo کند.

logها اهمیت زیادی می‌یابد. یک Garbage Collection در بحث logها مطرح می‌شود.

MICRO

12th Session — 91/10/9

یادآوری: با فرض این که زمانبند اجرای پی در پی پذیر و ترمیم پذیر را ایجاد کند، این برای تضمین اینکه اثرات این اجراها در پایگاه داده باقی بمانند کافی نیست. به همین علت بحث ترمیم مطرح می‌شود.

هدف اصلی ترمیم این است که پایگاه داده را به آخرین حالت سازگار ببرد.

در مورد Log صحبت کردیم: در برخی از Log ها اطلاعات به صورت زیر نوشته می‌شود

Tid	DI	V
T_{10}	x	5

Tid	Op. Type	DI	Old Val	New Val

→

Tid	DI	Old V	New V
T_{10}	x	10	5

چون فقط write ها نوشته می‌شود op. Type را می‌توان حذف کرد

این همان Before Image است. ولی با همان حالت اول هم ما Before Image را داریم. چون رکورد مربوط به مقدار قبلی هم در Log ها وجود دارد. بستگی به نحوه پیاده سازی دارد که از کدام حالت استفاده شود ولی حالت اول به واقعیت نزدیک‌تر است.

Log ها ابزاری است که RM از آن استفاده می‌کند. این Log ها به تدریج بزرگ و بزرگتر می‌شود. Log ها باید بتوانند به دو عمل Redo و Undo جواب بدهند. بعضی چیزهایی که در Log ها نوشته می‌شود بعد از مدتی Validity خود را از دست می‌دهند مثلاً اگر ۲ تراکنش روی یک DI نوشته باشند و کامیت کرده باشند اولی دیگر مهم نیست. بنابر دو دلیل به دنبال Garbage Collection در Log هستیم:

۱) حافظه مصرفی و پایین آوردن حجم Log.

۲) هرچه Log کوتاه‌تر باشد، کارایی بهتری در Restart و ترمیم خواهیم داشت.

Subject : Date

اصل اساسی: چه رکوردهایی از log ها می‌توانند حذف شوند؟ رکوردهایی که به درد Redo و Undo نمی‌خورند

تشخیص این رکوردها بر حسب قواعد حذف است:

قواعد حذف

می‌توان رکورد (x , Ti) را از log حذف کرد اگر:

۱/ Ti سقط کرده باشد.

۲/ اگر تراکنش دیگری مثل Tj پس از Ti روی x نوشته باشد و انجام شده باشد (LCT آن دیرتر بوده از Ti است) (رکوردهایی که به درد Redo نمی‌خورند)

۳/ رکوردهایی که برای Undo به کار می‌روند چه؟ اگر RM به Undo نیازی نداشته باشد، در این حالت می‌توان به محض اینکه تراکنش commit کرد اثراتش را از log ها حذف کرد

نکته مهم در log ها: خاصیت همانی (Idempatence)

در برخی منابع به جای اینکه خاصیت همانی را متعلق به log ها بدانند آن را متعلق به Restart می‌دانند و تاکید می‌کنند چون Restart از log ها استفاده می‌کند.

با وجود اینکه گفتیم R, W, A و C، اتمیک هستند ولی چون Restart در شرایط غیرعادی رخ می‌دهد ممکن است وسط هر یک از این اعمال و حتی وسط Restart اتفاق ناگواری رخ دهد.

خاصیت همانی بیانگر این است که عمل Restart باید برگشت‌پذیر باشد به این معنی که اجرای چند بار موفق آن، پایگاه داده را به آخرین حالت سازگار ببرد. اگر Restart در حین اجرا متوقف شود و لذا از اول شروع شود همان نتیجه‌ای را در پایگاه داده دارد که در اثر اجرای اول حاصل می‌شد. به عبارتی اجرای ناقص Restart هیچ اثری ندارد

MICRO

برای این کار، لازم است که هنگام عمل Restart در log هیچ تغییری انجام نشود و log به هیچ

عنوان عوض نشود. چون هر زمان که به مشکل بخوریم Restart کامل شده است.

log ابزار RM است. پس چرا آن را Cache Manager می‌نویسد؟ CM که هیچ ارتباطی از C و

A ندارد؟ CM مدیریت حافظه‌ها را برعهده دارد و از همه اتفاقات باخبر است. چون RM

دسترسی‌اش را به CM می‌دهد. **

CM همیشه ابتدا log را می‌نویسد و بعد عمل را انجام دهد. بدین ترتیب اگر بعد از نوشتن log

و قبل از انجام عمل اتفاقی رخ دهد، فرض بر این است که عمل انجام شده بوده است. چون log نوشته

شده و RM هم تصور می‌کند دستورش اجرا شده است.

** دسترسی مستقیم به حافظه دارد، بنابراین اگر log را هم CM بنویسد،

در وقت دارد آنها را جابجا کند، کارایی بسیار بالا انجام نمی‌شود (چون می‌داند در log چه خبر است)

۲. می‌تواند به صورت Inplace کار کند.

الگوریتم Redo/Undo

نسبت به سایر الگوریتم‌ها انعطاف‌پذیری و کارایی بیشتر است. چون RM و CM هر کدام مستقل کار می‌کنند

و در کار یکدیگر دخالت ندارند و مکانیزم‌های کارایی خواهند داشت. ولی چون اینجا هم Redo داریم

هم Undo، و این‌ها هر کدام مکانیزم‌هایی در مراحل مختلف دارند، پرهزینه است. ولی چون پیش‌فرض

این است که سیستم در بیشتر مواقع بالا است و به ندرت down می‌شود، کارایی بر هزینه اولویت دارد.

Subject :   Date

- توابع اجرای سیستمی (مدیر بازیابی)

پنج الگوریتم RM-R ، RM-W ، RM-C ، RM-A و Restart را شرح می‌دهیم

RM-Write (Ti, x, V)

- اگر Ti در لیست تراکنش‌های فعال نیست اضافه کن

- x را واکشی کن } می‌تواند موازی انجام شود

- رکورد [Ti, x, V] را در log بنویس

- V را در اسلات فراهم شده بنویس و dirty = 1

- تصدیق انجام عمل به زمانبند ارسال شود

این گام‌ها ایده است که باید انجام شود، برخی از آن‌ها را RM و برخی را CM انجام می‌دهد.

RM-Read (Ti, x)

- x را واکشی کن

- تصدیق عمل انجام عمل (مقدار خوانده شده) را به زمانبند ارسال کن

RM-Commit (Ti)

تصمیم در تصدیق C و A باید نهایت صداقت را به خرج بدهد. تا از همه چیز مطمئن نشده C نخواهد و اگر بین C و A گیر کرد A بدهد.

- T_i را به لیست تراکنش های نهایی شده (Committed List) اضافه کن
- تصدیق را به برنامه ارسال کن
- از لیست تراکنش های فعال، T_i حذف شود

اگر قبل از مرحله سوم ۳ سیستم به شکل بخورد و به ناچار Restart شود، T_i هم در Committed لیست هم در Active لیست وجود دارد. وقتی log جدید نوشته شده فرض می کنیم انجام شده

RM - Abort (T_i)

• در اینجا اول اثرات تراکنش را برمی گردانیم و سپس در log می نویسیم چون وقتی در log می نویسیم همه فرض این می شود که عمل انجام شده یعنی فرض می شود همه اثرات Undo شده

در اینجا ابتدا Undo می کنیم و سپس log را می نویسیم

• برای هر قلم داده نوشته شده توسط T_i مثل x
- x را واکشی کن
- متعلق به x را Before Image آن را در x بازنویسی شود. (Before Image در log هست)

• T_i را در لیست تراکنش های Abort شده قرار بده
• تصدیق نه را به برنامه ارسال کن
• T_i را از لیست Active حذف کن

Subject : Date

Restart

- همه slot های حافظه Cache را فلاش کن (این slot ها دیگر قابل اطمینان نیستند)

- عمل Restart از دو لیست Redone و Undone استفاده می‌کند. شامل تمام داده‌هایی است که Redo شده اند و Undone شامل تمام داده‌هایی است که Undo شده اند.

در این مرحله Redone = { } و Undone = { } می‌شود

Log را از پایین به بالا (به ترتیب) بخوان تا به پایان log برسد یا Redone U undone = DB

حلقه:

- رکورد (Ti, x, V) را از log بخوان و اگر x ∉ Redone U undone (یعنی x هنوز مقدار ندارد):

- x را واکشی کن

- اگر Ti ∈ Committed list ، V را در x کپی کن و Redone = Redone U {x}

- در غیر این صورت (یعنی Ti ∈ Aborted list ، (Ti ∈ Active List Before Image(x) را از log استخراج و در دادهٔ مربوطه x کپی و Undone = Undone U {x}

⊕ بعداً توضیح دارد

- همه تراکنش‌هایی که در Committed List و Active List هستند را از Active List خارج کن

- نهایت عمل Restart را ادامه بده

- Aborted list = Active list (می‌توان این دو را مساوی φ قرار داد؟؟ به دلیل کلم بیاورید)

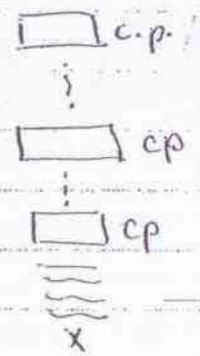
⊕ اگر شرط خوش‌رفتاری را از اجراها برداریم، این قسمت متفاوت خواهد بود. این قسمت برای بازنویسی نتیجه اجرا خوش رفتار است. در این بخش ما در ابتدا Aborted و Active را با هم یکی کردیم در حالی که

اگر در Aborted باشد در اصل x ، LCN خود را برگشت آورده است (مراجعه شود به الگوریتم Abort).

با این روند نقطه این تغییر را بازنویسی کنید.

نقطه وارسی / مقابله Check Point

Chk Point فقط به دنبال کاراتر کردن عمل Restart است. در بازه‌ای زمانی معین، برای چند عمل با هم، کار متوقف شده و کل Cache در دیتابیس کپی می‌شود. بعد و بعد در log درج می‌شود پیغام Check Point که نشان می‌دهد در این نقطه Cache و Physical DB با هم Sync شده‌اند. برای این ترتیب عملیات Redo بالقوه کاهش می‌یابد. در c.p. : Aborted list ، Committed list و Active list داریم و این نشان می‌دهد باید به سراغ Undo کردن کدام تراکنش‌ها برویم. برای Restart تا آخرین c.p. عمده کار انجام می‌شود و فقط ممکن است برای برگشت آوردن Before Img یک سری از تغییرات مجبور شود از آخرین c.p. به قبل‌تر برود.



تمرین: با فرض این که در فایل log ، Chk Point داریم الگوریتم Restart را بازنویسی کنید.

برای بهینه‌تر عمل Restart کاراتر شود علاوه بر مفهوم Check Point ، Fuzzy checkpoint مطرح شد. در این شیوه نقطه اصلاحات که بیت dirty آن‌ها 1 است از Cache روی DB نوشته می‌شوند.