فصل دوم

مدیریت تراکنش

مدیریت تراکنش به عنوان مفهوم مرکزی و یکی از مهمترین بخش‌ها در بانک اطلاعات مطرح است و همه‌ی آنچه که در بانک اطلاعات ۱ مطالعه کردیم، مثل حساب رابطه‌ای، جبر رابطه‌ای و SQL، به تراکنش مربوط می‌شوند.

مفاهیم

مقدمه

در جلد اول کتاب، مفهوم تراکنش به اجمال بیان شد و در ضمیمه اول این جلد نیز به اختصار آمده است. گفتیم که برنامه‌های کاربران به عنوان تراکنش به سیستم مدیریت بانک اطلاعات تحویل می‌شود، این سیستم چهار کنترل موسوم به ACID را روی آنها اعمال و در نهایت این برنامه‌ها یا به خوبی اجرا شده و پایان می‌یابند که به این حالت انجام یا تثبیت (commit) می‌گویند و یا اینکه ساقط (abort) می‌شوند.

مجموعه‌ای از عملگرهای بانک اطلاعات که از دید کاربر یک واحد منطقی کار را تشکیل می‌دهند، تراکنش (transaction) نام دارد.

مثال:

```
begin T1:
      read(A);
     A:=A-50;
     write(A);
     read(B);
     B:=B+50;
     write(B);
end T1
```

اگر خواص ACID را روی تمام تراکنش‌ها اعمال کنیم، جامعیت بانک اطلاعات (integrity) حفظ خواهد شد. بنابراین هدف اصلی در بانک اطلاعات، حفظ جامعیت است. این خواص که جامعیت بانک اطلاعات را حفظ می‌نمایند عبارتند از:

۱- Atomicity : یکپارچگی

۲- Consistency : همخوانی

۳- Isolation : انزوا

۴- Durability : پایایی

Atomicity : یک تراکنش یا یک برنامه که یک واحد کاری بوده و بین دستورهای begin و end قرار می‌گیرد که یا باید همه‌ی دستورات آن اجرا بشود یا هیچ‌کدام. بنابراین اگر تراکنشی شروع شد و بعد همه‌ی دستورات آن انجام شد، می‌گوییم تثبیت شد. ولی اگر در بین انجام کار نتوانست به هر دلیلی ادامه دهد، باید آن را بازگردانیم؛ یعنی تأثیراتی که روی بانک اطلاعات گذاشته است، باید خنثی شوند. بنابراین خاصیت Atomicity یعنی همه یا هیچ.

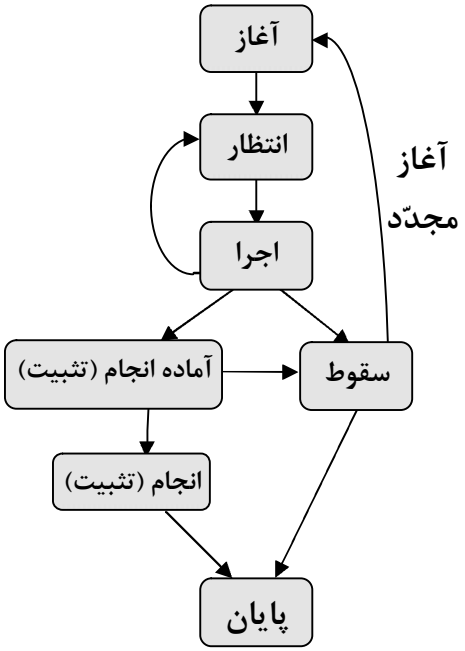
Consistency : اگر تراکنشی تثبیت شد، نباید بانک اطلاعات را خراب کند. مثلاً اگر تراکنشی نمره‌ای را تغییر داد، باید آن را به یک مقدار مجاز تبدیل کند.

Isolation : تراکنش‌های همروند یا تراکنش‌هایی که طول عمرشان همپوشانی دارد، نباید تأثیر مخرب روی هم داشته باشند. توجه کنید که تراکنش‌های مختلف از وجود همدیگر بی‌خبر هستند.

Durability : اگر تراکنشی تثبیت شد، تأثیر آن به طور اتفاقی از بین نخواهد رفت.

مهمترین بخش‌های اعمال این خواص، یکی واحدی به نام واحد کنترل همروندی (concurrency control component) است که تراکنش‌های همروند را کنترل می‌کند تا تأثیر مخرب نداشته باشند و دیگری واحدی به نام واحد مدیریت ترمیم (recovery management component) می‌باشد که وظیفه آن جلوگیری از تأثیر تراکنش‌های نیمه‌کاره بر روی بانک اطلاعات و از بین بردن آثار آنها است.

وضعیت (حالات) تراکنش



شکل مراحل اجرای تراکنش

توجه داشته باشید که اگر تراکنشی ساقط شود، می‌توان آن را مجدداً آغاز کرد و اگر تراکنشی آماده انجام شود، ممکن است باز هم سقوط کند.

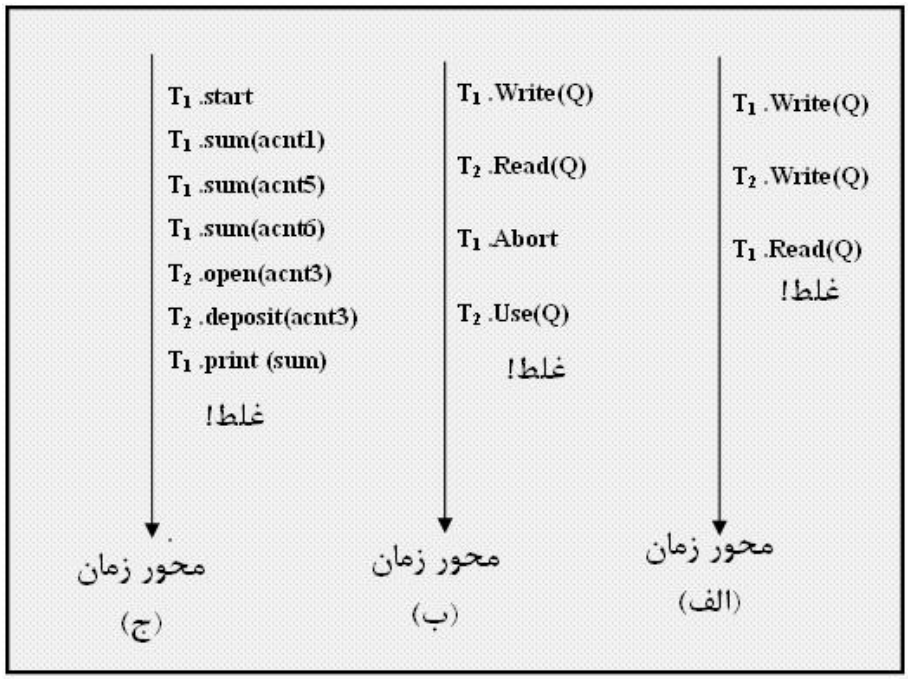
مرحله "آماده انجام" (ready-to-commit) به معنی زمانی است که تراکنش همه کارهای مربوطه را انجام داده اما باید بخش‌های مختلفی را با هم هماهنگ کند.

ما همواره با تراکنش‌های همروند یعنی تعدادی تراکنش که با یکدیگر کار می‌کنند سر و کار داریم. اگر بخواهیم تراکنش‌ها را یکی پس از دیگری اجرا کنیم، در این صورت هم گذردهی (throughput) سیستم پایین می‌آید و هم میانگین زمان پاسخ دهی افزایش پیدا می‌کند. در حین اجرای یک تراکنش طولانی مدت شاید بتوان صدها تراکنش کوتاه را اجرا کرد و نباید آنها را منتظر گذاشت. بنابراین همروندی تراکنش‌ها، کارایی سیستم را بالا می‌برد اما مشکلاتی هم به همراه دارد.

اجرای همروند تراکنش‌ها را در قالب ساختاری به نام زمانبندی (schedule) بررسی می‌کنیم. در یک زمانبندی می‌توان تراکنشی را منتظر تراکنش دیگری گذاشت یا می‌توان اجرای یک تراکنش را به تعویق انداخت. اما در درون یک تراکنش نمی‌توان دستورها را جابه‌جا کرد به دلیل اینکه منطق آن تراکنش به هم می‌ریزد.

هرچند همروندی تراکنش‌ها، کارایی سیستم را بالا می‌برد اما ممکن است مشکلات زیر را به دنبال داشته باشد. این مشکلات در شکل ۷/۲ نمایش داده شده‌اند.

در زمانبندی‌های همروند مشکلاتی به وجود می‌آیند که می‌توان این مشکلات را به سه دسته تقسیم کنیم.



شکل انواع مشکلات همروندی الف) تغییرات گمشده ب) دستیابی به داده نهایی نشده ج) بازیابی ناهمگام

الف) تغییرات گمشده (lost updates)

تراکنش T_1 در نقطه‌ای از زمان داده Q را می‌نویسد (مثلاً ۱۰۰). پس از آن تراکنش T_2 روی آن داده می‌نویسد (مثلاً ۱۰۰۰). توجه کنید که تراکنش‌ها از وجود یکدیگر بی‌خبرند. آنگاه تراکنش اولی مقدار Q را می‌خواند و انتظار همان مقداری را که نوشته دارد. غافل از اینکه دیگری آن را تغییر داده (مثلاً انتظار ۱۰۰ دارد در صورتی که داده ۱۰۰۰ را نشان می‌دهد)

ب) دستیابی به داده نهایی نشده (uncommitted-data access یا dirty-read)

تراکنش T_1 در نقطه‌ای از زمان داده را تغییر می‌دهد. سپس تراکنش T_2 همان داده را می‌خواند ولی بعداً T_1 به دلیلی ساقط می‌شود و داده به حال اولیه بر می‌گردد. T_2 بی‌خبر از همه جا از آن داده استفاده می‌کند و به راه خود ادامه می‌دهد و دچار خطا می‌شود.

دو حالت الف و ب یک خاصیت مشترک دارند و این خاصیت این است که هر دو تراکنش روی یک داده کار می‌کنند.

ج) بازیابی ناهمگام (inconsistent retrieval یا phantom problem)

ممکن است ما داده‌ی مشترکی نداشته باشیم، یعنی دو یا چند تراکنش داشته باشیم که روی داده‌های متفاوتی کار می‌کنند و داده‌ی مشترکی وجود ندارد ولی باز نتیجه‌ی کار غلط است. این غلط بودن به دلیل وابستگی بین این داده‌هاست.

مثال: تراکنش T_1 جمع موجودی تعدادی حساب بانکی را محاسبه می‌کند که مثلاً این حساب‌ها از حساب شماره ۱ تا شماره ۶ هستند. اما در این میان تراکنش دیگری، حساب جدیدی با نام شماره ۳ را باز می‌کند و مبلغی را به حساب آن می‌ریزد. طبیعی است که جمع موجودی حساب‌های ۱ تا ۶ که توسط تراکنش T_1 انجام شده، دیگر درست نیست.

معروف‌ترین متد کنترل همروندی با نام پی‌درپی پذیری (serializability) از آن یاد می‌شود.

همچنان که می‌دانیم یک تراکنش اگر خاصیت ACID داشته باشد نتیجه‌اش درست است و نیز اگر چند تراکنش داشته باشیم که پی در پی باشند، این زمانبندی که به آن زمانبندی پی‌درپی می‌گوییم نیز نتیجه‌اش درست است. پی‌درپی پذیر یعنی معادل پی‌درپی؛ یعنی اگر زمانبندی‌های همروندی داشته باشیم که پی‌درپی نیستند اما معادل پی‌درپی هستند، در این صورت نیز نتیجه کار درست خواهد بود.

دو روش اصلی پی‌درپی پذیری عبارتند از:

۱. پی‌درپی پذیری در برخورد یا CSR (Conflict Serializability)
۲. پی‌درپی پذیری در دید یا VSR (View Serializability)

برای اینکه بتوانیم این روش‌ها را بیان کنیم، باید دستورات تراکنش‌ها را دسته‌بندی کنیم. از آنجا که دستورات محاسباتی تأثیری در پی‌درپی پذیری ندارند، در زمانبندی‌ها فقط دستورات خواندن (read()) و نوشتن (write()) را در نظر خواهیم گرفت. علائمی که برای دستورات تراکنش به کار می‌بریم عبارتند از:

$r_i(Q)$: تراکنش i داده Q را می‌خواند (read).

$w_i(Q)$: تراکنش i روی داده Q می‌نویسد (write).

c_i : تراکنش i به مرحله تثبیت (commit) می‌رسد.

a_i : تراکنش i ساقط (abort) می‌شود.

برای شروع و پایان تراکنش علائمی به کار نمی‌بریم؛ بلکه اولین دستور تراکنش شروع آن است و آخرین دستور آن (commit یا abort) نیز پایان آن.

پی‌درپی پذیری در برخورد

اگر دستورات خواندن و نوشتن مربوط به تراکنش‌های مختلف به صورت همروند اجرا شوند، بعضاً با هم برخورد دارند و بعضاً ندارند. برخورد داشتن یعنی تأثیر مخرب روی هم گذاشتن. ما دستوراتی را که با هم برخورد ندارند می‌توانیم همروند اجرا کنیم و آنهایی را برخورد دارند، باید به صورت پی‌درپی اجرا کنیم.

تعریف: برخورد

چنانچه q_j, p_i به ترتیب عملگرهای (دستورات) تراکنشهای t_j, t_i باشند، گوییم q_j, p_i با هم برخورد دارند اگر و تنها اگر:

۱- این دو عملگر مربوط به تراکنش‌های متمایز باشند $(t_i \neq t_j)$

۲- هر دو عملگر به یک داده دسترسی داشته باشند.

۳- حداقل یکی از این دو عملگر، عملگر نوشتن (()write) باشد.

T_i / T_j	$r_i(Q)$	$w_i(Q)$
$r_j(Q)$	بی‌برخورد	برخورددار
$w_j(Q)$	برخورددار	برخورددار

جدول برخورد بین دستورات

دستور ()write هر جا که باشد، با دستورات دیگر برخورد دارد و فقط دو دستور ()read از دو تراکنش متمایز با هم برخورد نخواهند داشت. البته برخورد فقط روی داده‌ی مشترک است.

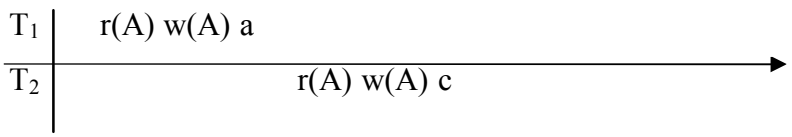
نکته: پی‌درپی پذیری در برخورد فقط مشکلات الف و ب را حل می‌کند چون هر دو باید به یک داده دسترسی داشته باشند. بنابراین مشکل ج توسط پی‌درپی پذیری در برخورد حل نخواهد شد.

تعریف: زمانبندی پی‌درپی

زمانبندی S را پی‌درپی گوییم اگر برای هر دو تراکنش، پایان یکی قبل از شروع دیگری باشد.

مثال: یک زمانبندی پی‌درپی

$$S_1 : r_1(A)w_1(A)a_1w_2(A)w_2(B)c_2$$



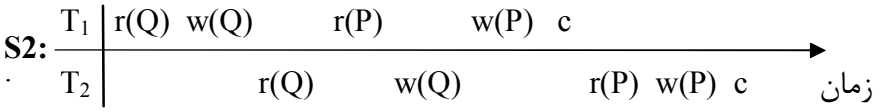
ترتیب اجرای پی‌درپی آنها T_1 و سپس T_2 می‌باشد و می‌نویسیم:

$$S_1 : T_1 < T_2$$

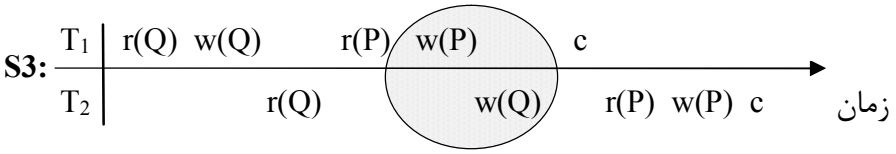
تعریف: زمانبندی معادل در برخورد (conflict equivalent)

زمانبندی‌های S و S' معادل در برخورد هستند، اگر هر دو روی یک مجموعه از دستورات و تراکنش‌ها کار کنند و با جابه‌جا کردن دستورات بدون برخورد در زمانبندی S، بتوانیم زمانبندی S' را تولید کنیم.

مثال: "معادل در برخورد" با زمانبندی S_2:



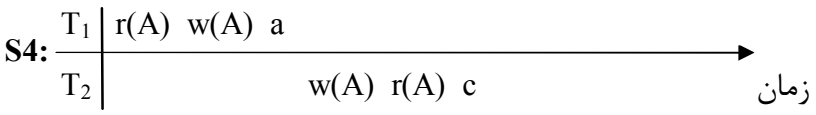
حل:



در این مثال فقط ترتیب زمانی $w_1(P)$ و $w_2(Q)$ عوض شده. هر چند هر دو دستور w() هستند ولی برخورد ندارند چون روی دو داده مختلف عمل می‌کنند.

نکته مهم: همان طور که قبلاً هم بیان شد به هیچ عنوان نمی‌توانیم ترتیب دستورات یک تراکنش را عوض کنیم چون منطق (semantic) آن تراکنش عوض می‌شود.

مثال: "معادل در برخورد" با زمانبندی S_1



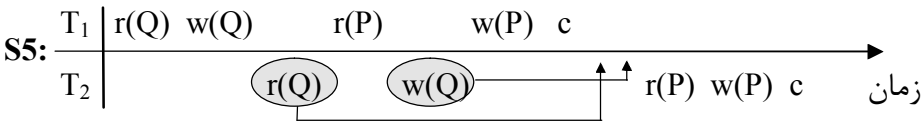
این مثال غلط است زیرا ترتیب زمانی دستورهای دو یا چند تراکنش عوض نشده بلکه در درون یک تراکنش، جای دستورها عوض شده است.

تعریف: پی‌درپی پذیر در برخورد

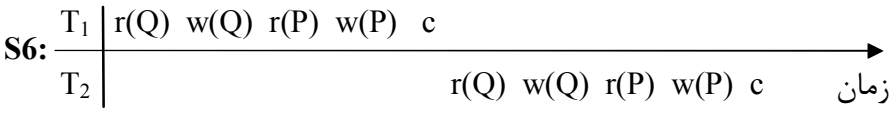
زمانبندی S پی‌درپی پذیر در برخورد است اگر معادل در برخورد با یک زمانبندی پی‌درپی باشد.

زمانبندی‌های همروند به شرطی که معادل در برخورد با یکی از ترتیب‌های (زمانبندی‌های) پی‌درپی باشند، مشکل همروندی ندارند.

مثال:

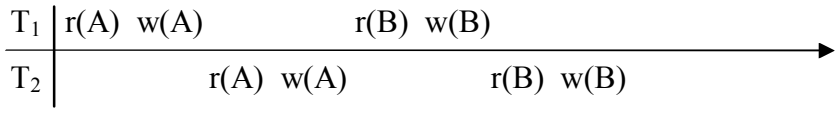


دستور $w_2(Q)$ را می‌توان از $w_1(P)$ و c_1 گذراند زیرا با آنها برخورد ندارد. همچنین می‌توان دستور $r_2(Q)$ را از $r_1(P)$، $w_1(P)$ و c_1 گذراند. حاصل این عمل‌ها، زمانبندی S_6 را به دست می‌دهدکه پی‌درپی و معادل S_5 است.



S_6 معادل اجرای پی‌درپی $T_1 < T_2$ می‌باشد پس می‌توان گفت S_3 و S_5 زمانبندهای پی‌درپی پذیر در برخورد هستند.

تمرین: آیا زمانبندی زیر پی‌درپی پذیر در برخورد می‌باشد یا خیر؟ اگر جواب مثبت است معادل پی‌درپی آن چه خواهد بود؟



حل: بله، پی‌درپی پذیر است و معادل پی‌درپی آن T_1 قبل از T_2 خواهد بود.

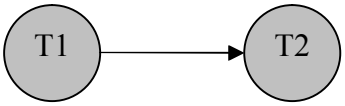
تشخیص پی‌درپی پذیری در برخورد (Conflict Serializability)

در مفهوم پی‌درپی پذیری در برخورد اگر زمانبندی با n تراکنش داشته باشیم، دستورهای بدون برخورد را آن‌قدر جابه‌جا می‌کنیم تا ببینیم به یک ترکیب پی‌درپی می‌رسیم یا خیر. این کار بسیار مشکل و زمان‌گیر است، زیرا زمانبندی‌ها نوعاً از تعداد زیادی تراکنش تشکیل شده‌اند که می‌آیند و می‌روند و دستورات گوناگون خود را فراخوانی می‌کنند.

خوشبختانه راه حلی بسیار ساده برای این مسئله پیدا شده و در واقع دلیل اصلی موفقیت و کارایی روش پی‌درپی پذیری در برخورد نیز همین راه حل ساده است.

گراف پی‌درپی پذیری (serializability graph)

رأس‌های این گراف تراکنش‌هایی هستند که در سیستم فعال می‌باشند و یال‌های آن جهت‌دار هستند.



از یک تراکنش به تراکنش دیگر یالی رسم می‌شود در صورتی که دستور برخورداری قبل از آن داشته باشد.

مفهوم قبل و بعد، از زمان گرفته شده است.

الگوریتم‌های بسیار ساده‌ای برای نگهداری گراف‌ها و تغییر آنها وجود دارد.

گراف‌ها به طور مرتب تغییر می‌کنند؛ یعنی به محض اینکه یک دستور خواندن و یا نوشتن وارد سیستم می‌شود، ممکن است یک یال را به گراف اضافه کند و زمانی که تراکنشی ساقط و یا تثبیت شده و از سیستم خارج می‌شود، یال‌هایی و نیز گره‌ای از گراف حذف خواهد شد. نگهداری این گراف بسیار ساده است.

هرگاه در گراف پی در پی پذیری، حلقه (cycle) وجود داشته باشد؛ آنگاه زمانبندی آن پی‌درپی پذیر نیست و بالعکس.

۱- هرگاه تراکنشی درخواستی برای خواندن یا نوشتن داد، اگر به آن اجازه بدهیم آیا گراف دچار حلقه می‌شود؟ اگر دچار حلقه می‌شود اجازه ندهیم.

۲- اجازه دهیم تراکنش‌ها به طور طبیعی کار خودشان را ادامه بدهند .فرض کنیم حلقه‌ای ایجاد نمی‌شود یا حداقل فرض کنیم تعداد حلقه‌هایی که ایجاد می‌شود بسیار ا ندک است.

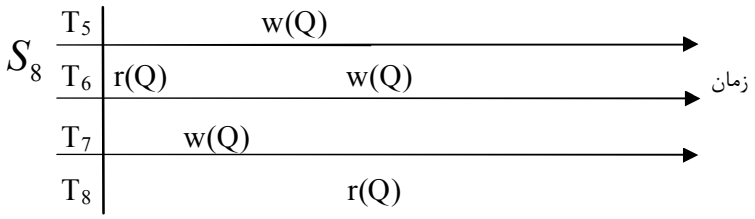
اگر حلقه ایجاد شده است آن را ساقط می‌کنیم و اگر حلقه ایجاد نکرده به آن اجازه می‌دهیم تثبیت شود.

مثال S_1:

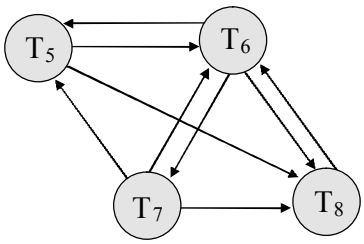
گراف پی‌درپی پذیری S_1:

مثال:

گراف پی‌درپی‌پذیری مربوط به زمانبندی S_8 زیر را رسم کنید و تشخیص دهید پی‌درپی‌پذیر است یا نه؟ (توجه شود که تقدّم زمانی $w_6(Q)$ و $r_8(Q)$ مشخص نیست.



حل:

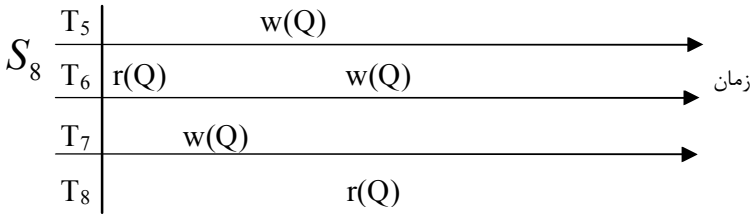


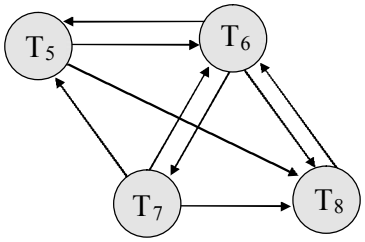
در این گراف تعدادی حلقه وجود دارد مانند $(T_5, T_7, T_6, T_8, T_5)$ و (T_6, T_7, T_6) می‌توان تشخیص داد که اگر تراکنش T_6 ساقط شود همه حلقه‌ها از بین می‌روند و زمانبندی پی‌درپی پذیر $T_7 < T_5 < T_8$ به دست می‌آید که به این مسئله پیدا کردن قربانی می‌گویند.

پیدا کردن قربانی:

دارای الگوریتم‌های مختلف می‌باشد. مثلاً می‌توانیم تراکنشی را به عنوان قربانی انتخاب کنیم که بیشترین حلقه‌ها را به وجود آورده (مانند T_6 در مثال بالا) یا تراکنشی که کمترین کار را انجام داده انتخاب کنیم؛ زیرا زمانی که آن را ساقط می‌کنیم، میزان از دست رفتن کار کمتر خواهد بود.

در مواردی که ما بیش از یک cpu داریم، ممکن است دستورهای ما همزمان باشند. به عبارت دیگر ممکن است ما ترتیب زمانی دو دستور را ندانیم.


S_8
T_5
T_6
T_7
T_8
w(Q)
r(Q)
w(Q)
w(Q)
r(Q)
زمان


T_5
T_6
T_7
T_8

پی‌درپی پذیری در دید

۱- آیا روش‌های دیگری هم برای پی‌درپی پذیری وجود دارد؟ بلی

- commit-serializability یا پی در پی پذیری در تثبیت
- FSR (Final-state SeRializability) یا پی‌درپی پذیری در حالت نهایی

۲- آیا روش بهتر از پی در پی پذیری در برخورد وجود دارد؟ خیر. روش پی‌درپی پذیری در برخورد بهترین روش موجود می‌باشد.

مزایا:

- بازتر از روش پی‌درپی پذیری دربرخورد عمل می‌کند، یعنی زمانبندی یا schedule را می‌پذیرد که آن رد می‌کند.

معایب:

- روش تشخیص آن زمانبر و طولانی می‌باشد.

تعریف:

تراکنش T_j از تراکنش T_i می‌خواند اگر T_j داده‌ای را که آخرین بار T_i در آن نوشته است، بخواند و T_i ساقط نشده باشد.

معادل در دید:

تعریف:

زمانبندی‌های s و s' **معادل در دید** هستند اگر تراکنشها و مجموعه عملگرهای s و s' یکسان باشد و سه شرط زیر برقرار باشد:

۱) برای هر داده Q، اگر تراکنش T_i مقدار اولیه Q را در s می‌خواند، آنگاه T_i مقدار اولیه Q را در s' نیز بخواند.

۲) برای هر داده Q اگر T_i در s داده Q را از T_j می‌خواند، در s' نیز داده Q را از T_j بخواند.

۳) برای هر داده Q، آخرین تراکنشی از زمانبندی s که روی Q می‌نویسد، همان تراکنشی باشد که در زمانبندی s' آخرین بار روی Q می‌نویسد.

ما از دیدگاه یکسانی به این دو زمانبندی نگاه می‌کنیم و اگر یکسان بودند می‌گوییم معادل در دید هستند.

- مقدار اولیه‌ای که می‌خواند.
- read from یا داده‌هایی که تراکنشی از تراکنش دیگر می‌خواند.
- final write یا آخرین نوشتن‌های تراکنش‌ها در دو زمانبندی را نگاه می‌کنیم.

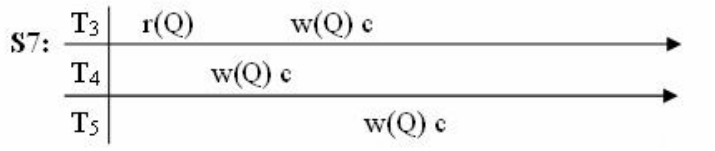
با اعمال این شرط‌ها اولاً تضمین می‌کنیم که هر تراکنش در هر دو زمانبندی مقادیر یکسانی را بخواند. ثانیاً با تضمین یکسان بودن مقادیر نوشته شده روی هر داده، اطمینان حاصل می‌شود که وضعیت نهایی بانک اطلاعات در هر دو زمانبندی یکسان است.

تعریف پی‌درپی پذیری در دید

زمانبندی S پی‌درپی پذیر در دید است اگر معادل در دید با یک زمانبندی پی‌درپی باشد.

مثال:

آیا زمانبندی زیر پی‌درپی پذیر در برخورد می‌باشد؟ پی‌درپی پذیر در دید چه طور؟



جواب: S_7 پی‌درپی پذیر در برخورد نیست ولی پی‌درپی پذیر در دید می‌باشد. ترتیب پی‌درپی آن $T_3 < T_4 < T_5$ خواهد بود. برای تشخیص پی‌درپی پذیر بودن در دید، فقط یک داده به نام Q داریم که:

۱. در هر دو زمانبندی، T_3 مقدار اولیه Q را می‌خواند.
۲. هیچ تراکنشی مقداری برای Q از تراکنشی دیگر نمی‌خواند.
۳. در هر دو زمانبندی، تراکنش T_5 آخرین عمل نوشتن روی Q را انجام می‌دهد.

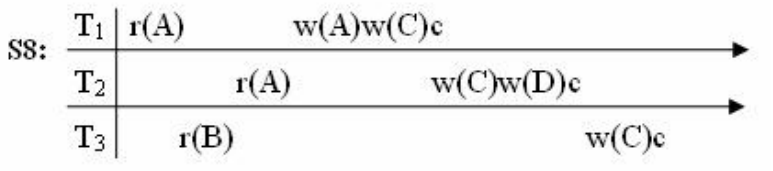
«پس این زمانبندی CSR نیست اما VSR هست»

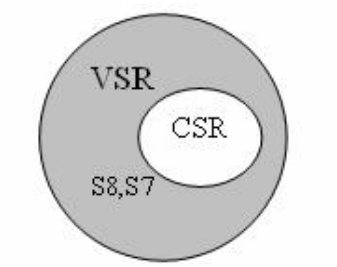
هر زمانبندی که CSR نباشد ولی VSR باشد، نوشتن کورکورانه blind write انجام داده است، یعنی بدون اینکه قبلاً داده مربوطه را خوانده باشد در آن می‌نویسد (در اینجا T_4 و T_5).

تمرین:

نشان دهید که زمانبندی زیر CSR نیست اما VSR است.

$$S_8 : r_1(A)r_3(B)r_2(A)w_1(A)w_1(C)c_1w_2(C)w_2(D)c_2w_3(C)c_3$$





شکل: محدوده عملکرد دو روش اصلی پی‌درپی پذیری

یادآوری: درکتاب روش متفاوتی برای تشخیص پی‌درپی پذیزی دردید آمده است.

- VSR بازتر عمل می‌کند.
- CSR زمانبندی‌هایی را بیهوده نمی‌پذیرد.
- VSR کاربردی نیست زیرا مهمترین مشکل آن پیچیدگی بسیار بالای الگوریتم تشخیص پی‌درپی پذیری در دید است.

تشخیص پی‌درپی پذیری در دید مسئله‌ای است که با پیچیدگی زمانی چند جمله‌ای قابل حل نیست (NP-complete) و تشخیص پی‌درپی پذیری در برخورد با پیچیدگی زمانی n^2 (n تعداد تراکنش‌ها) که قابل قبول است.

سؤال:

آیا پی‌درپی پذیر بودن به معنی درستی یک زمانبندی است؟ یعنی جامعیت بانک اطلاعات را تضمین می‌کند؟

پاسخ: خیر.

جامعیت بانک اطلاعات دو شرط دارد:

۱- از لحاظ کنترل همروندی مشکل نداشته باشد (پی‌درپی پذیری یکی از روش‌های کنترل همروندی است).

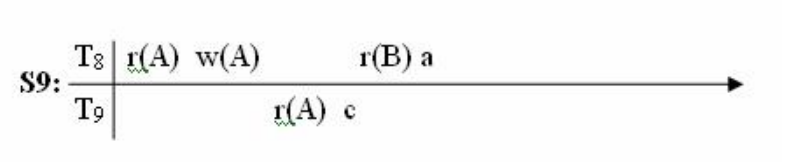
۲- ترمیم پذیر باشد.

ترمیم‌پذیری

تا اینجا امکان اینکه تراکنشی دچار خرابی شود را نادیده گرفته بودیم.

مثال:

فرض کنید در زمانبندی S_9، تراکنش T_9 به محض انجام دستور خواندن، تثبیت شود. از طرفی اگر T_8 پس از تثبیت شدن T_9 مجبور به سقوط شود، T_9 به داده‌های تثبیت نشده دسترسی داشته است که صحت بانک اطلاعاتی را خدشه‌دار می‌نماید. بنابراین زمانبندی برای درست بودن علاوه بر پی‌درپی پذیری باید ترمیم‌پذیر نیز باشد.



تعریف:

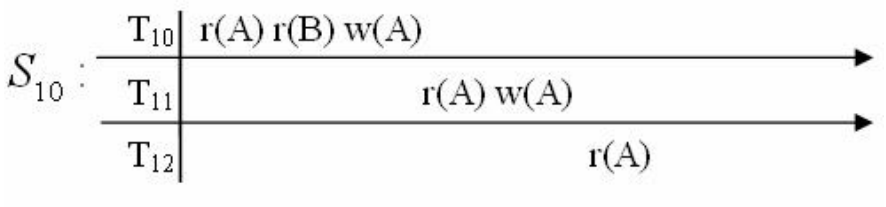
زمانبندی را **ترمیم‌پذیر** گوییم اگر برای تمام T_j ها که از T_i می‌خوانند، تثبیت تراکنش T_i قبل از تثبیت تراکنش T_j صورت گیرد.

اشکالات:

زمانبندی ترمیم‌پذیر ممکن است سبب ساقط شدن تراکنش‌های دیگر به صورت آبشاری (cascading abort) گردد. این مشکل می‌تواند سبب از دست رفتن حجم قابل توجهی از کارهایی که تا به حال انجام شده‌اند گردد.

مثال:

در زمانبندی S_{10} چنانچه T_{10} دچار خرابی شود، T_{11} و T_{12} نیز مجبور به سقوط خواهند شد.



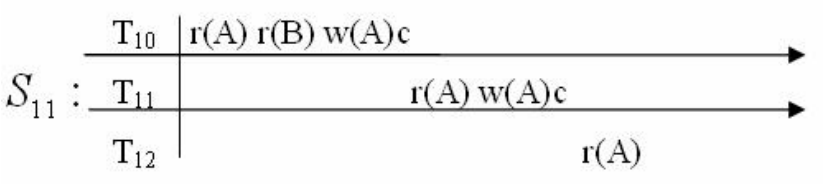
تعریف:

زمانبندی را **فاقد سقوط‌های آبشاری** Avoiding Cascading Aborts (ACA) گوییم چنانچه برای هر دو تراکنش T_i و T_j، اگر T_j از T_i بخواند، آنگاه T_i قبل از خواندن T_j تثبیت شده باشد.

«زمانبندی فاقد سقوط‌های آبشاری بهتر از زمانبندی‌های ترمیم پذیر هستند.»

مثال:

مشکل سقوط‌های آبشاری زمانبندی S_{10}، در زمانبندی S_{11} رفع شده است:



مشکل زمانبندی‌های آبشاری:

- فقط به خواندن می‌پردازند در عین حال این زمانبندی‌ها جامعیت بانک اطلاعات را تضمین می‌کنند.

زمانبندی محض (سخت‌گیر):

چنانچه برای هر دو تراکنش T_i و T_j، اگر T_j داده‌ای را پس از نوشتن T_i، بخواند یا بنویسد، این عمل T_j بعد از خاتمه (تثبیت یا سقوط) T_i اجرا شود.

از لحاظ خواندن معادل فاقد سقوط آبشاری است ولی مفهوم نوشتن را هم در نظر می‌گیرد. به عبارت دیگر زمانبندی سخت‌گیر، اجازه خواندن یا نوشتن هیچ داده‌ای را نمی‌دهد مگر آنکه تراکنشی که روی آن داده نوشته است، خاتمه یافته باشد (انجام یا ساقط شده باشد).

«زمانبندی محض (سخت‌گیر) به طور همزمان خاصیت پی‌درپی پذیری و ترمیم پذیری را دارد و بنابراین جامعیت بانک اطلاعات را یکجا تضمین می‌کند.»

نکته:

در کنترل‌هایی که انجام می‌دهیم، زمانبندی‌ها اگر غلط بودند abort یا ساقط می‌کردیم و این کار درست نیست. ساقط کردن تراکنش‌ها باعث می‌شود مقدار زیادی از کارهایی که انجام دادند از بین برود.

سؤال:

آیا می‌توان از ساقط کردن جلوگیری کرد؟

بلی؛ می‌توان پروتکل‌هایی را برای کنترل همروندی و ترمیم پذیری تراکنش‌ها تعریف کنیم که اگر زمانبندی‌ها با اجرای این پروتکل‌ها به پیش بروند، همواره درست باشند و صحت و جامعیت بانک اطلاعات را به مخاطره نیاندازند و جز در موارد استثنایی که خواهیم دید نیازی به ساقط کردن تراکنش‌ها نیست. بنابراین پروتکل‌هایی را در آینده بیان می‌کنیم.

ب- پروتکل‌های کنترل همروندی
(Concurrency Control Protocols)

آشنایی

در خصوص زمانبندی تراکنش‌ها، دو رویکرد کلی وجود دارد:

خوش‌بینانه optimistic

بدبینانه pessimistic

در برخی از کاربردها می‌توان فرض کرد که اجرای همروند تراکنش‌ها شرایط صحت همروندی را معمولاً حفظ می‌کند. پس اجازه می‌دهیم که تراکنش‌ها کارشان را انجام دهند و در آخر، قبل از نوشتن نتایج آنها روی رسانه و انعکاس تغییرات در بانک‌های اطلاعات، صحت عملکرد آنها را بررسی می‌کنیم. یا اینکه برعکس، می‌توانیم با شک و تردید و عدم اطمینان به آنها نگاه کنیم و فقط در صورتی که صحت و جامعیت بانک اطلاعات را حفظ کنند، به آنها اجازه اجرا بدهیم.

زمانبندی‌ها در مواجهه با هر عملگر دریافت شده، یکی از سه انتخاب زیر را پیش رو دارد:

الف) اجرای عملگر (*execute*)

ب) به تأخیر انداختن اجرای عملگر (قرار دادن آن در صف) (*delay*)

ج) نپذیرفتن (رد کردن) عملگر (که منجر به ساقط شدن تراکنش می‌گردد) (*reject*)

زمانبندی‌ها را می‌توان به طور کلی به دو دسته تقسیم کرد:

- زمانبند محافظه‌کار conservative
- زمانبند مهاجم aggressive

زمانبندی محافظه‌کار اگر لازم باشد، اجرای دستورات تراکنش را به تعویق می‌اندازد و محتاطانه و محافظه‌کارانه اجرای دستورات را پی می‌گیرد تا حتی‌الامکان بتواند دستورات را اجرا و از سقوط آنها جلوگیری کند.

زمانبندی محافظه‌کار تأخیر (*delay*) دارد اما حتی الامکان ساقط کردن و (*reject*) ندارد. بالعکس، در زمانبندی مهاجم، هدف پرهیز از تأخیر در اجرای دستورات است، لذا دستورات را فوراً اجرا می‌کند که البته ممکن است مشکلاتی پیش بیاید و مجبور به ساقط کردن برخی از تراکنش‌ها گردد.

به طور عمومی، زمانبندی‌های محافظه‌کار که از سقوط تراکنش جلوگیری می‌کنند، عملکرد بهتری دارند.

به هرحال، الگوریتم ها و پروتکل‌های کنترل همروندی را می‌توان در قالب یکی از این دو رویکرد جامه عمل پوشانید.

مهمترین پروتکل‌های کنترل همروندی را می‌توان به صورت زیر تقسیم بندی نمود.

پروتکل‌های مبتنی بر قفل (lock – based)

پروتکل‌های مبتنی بر قفل کاربردی‌ترین روش کنترل همروندی می‌باشند. در این روش‌ها که براساس تخصیص داده‌ها به تراکنش‌ها است، هرگاه تراکنشی بخواهد برای خواندن یا نوشتن به داده‌ای دسترسی داشته باشد، ابتدا درخواست قفل مناسب با آن دستور را به واحدی به نام **مدیر قفل lock manager** می‌دهد.

مدیر قفل در صورتی داده‌ای را برای تراکنشی قفل می‌کندکه:

- داده قبلاً توسط تراکنش دیگری قفل نشده باشد.
- اگر قفل شده قفل جدید(درخواستی) با قفل قدیم سازگار باشد.(در آن واحد دو قفل را روی داده داشته باشیم).

سازگاری قفل‌ها

به طور کلی دو نوع قفل زیر مرسوم است:

الف: قفل دو حالتی (باینری-binary)

سازگاری بین قفل‌ها وجود ندارد؛ یعنی اگر داده‌ای توسط تراکنشی قفل شده باشد، به هیچ وجه تراکنش دیگری نمی‌تواند آن را قفل کند و اگر قفل نشده باشد می‌تواند آن را قفل کند.

ب: قفل اشتراکی – انحصاری (Shared – exclusive)

قفل‌ها به دو نوع اشتراکی (S) و انحصاری (X) تقسیم می‌شوند. قفل اشتراکی مخصوص خواندن read و قفل انحصاری مخصوص نوشتن write می‌باشد. قفل اشتراکی قابل اشتراک بین چندین تراکنش می‌باشد. مثلاً چندین تراکنش می‌توانند همزمان یک متن را بخوانند.

تعریف همزمان: در یک لحظه زمانی دو عمل انجام می‌شود و این در صورتی امکان پذیر است که بیش از یک cpu داشته باشیم (چه به صورت متمرکز و چه نامتمرکز).

تعریف همروند: کارهای همزمان ممکن است اتفاق بیافتد و ممکن است اتفاق نیفتد. اما طول عمر تراکنش‌ها همپوشانی دارد (طول عمرشان over lapping می‌باشد).

با توجه به تعریف قفل‌های S و X، جدول سازگاری (compatibility) این قفل‌ها را (که توسط مدیر قفل مورد استفاده قرار می‌گیرد) نشان می‌دهد:

قفل	s_i	x_i
s_j	سازگار	ناسازگار
x_j	ناسازگار	ناسازگار

جدول سازگاری قفل‌های S و X

چند نکته :

- هر تراکنش قبل از اجرای **هر** دستور r یا w باید درخواست قفل مربوطه را به مدیر قفل بدهد.
- چنانچه درخواست قفلی اجابت نشود، تراکنش به حالت انتظار (wait) می‌رود و تا زمانی که قفل داده باز شود و یا حالت سازگار پیش آید، در انتظار باقی بماند.
- $s_i(Q)$: تراکنش i روی داده Q درخواست قفل اشتراکی داده است.
  $x_i(Q)$: تراکنش i روی داده Q درخواست قفل انحصاری داده است.
  $u_i(Q)$: تراکنش i روی داده Q درخواست باز کردن قفل داده است.

- به محض آنکه تراکنشی کارش با داده‌ای پایان یافت، اجازه ندارد قفل آن داده را باز کند.
- قفل گذاری داده‌ها می‌تواند با دانه‌بندی‌های granularity مختلف (از لحاظ اندازه داده) انجام گیرد.

به عنوان مثال در بانک‌های اطلاعاتی می‌توان یک جدول، یک سطر یا بخشی از جدول یا ستونی را قفل کنیم.

ار آنجا که بانک اطلاعات از سرویس‌های سیستم عامل برای قفل‌گذاری استفاده می‌کند، معمولاً واحدهای قفل عبارتند از page ها یا segment ها.

« هرقدر سطح قفل گذاری پایین باشد (دانه بندی کوچکتر باشد) تعداد قفل‌ها و سربار قفل‌گذاری بالا می‌رود ولی سطح همروندی افزایش پیدا می‌کند.»

- جدول قفل (lock table) : مشخص می‌کند هر داده‌ای توسط چه تراکنش‌هایی قفل زده شده و یا باز می‌شود.

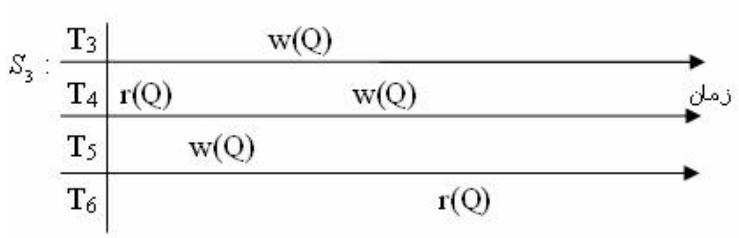
مثال: معادل زمانبندی S_1 را با استفاده از قفل‌های اشتراکی و انحصاری بنویسید

$$.S_1 : r_1(A)w_1(A)a_1w_2(A)w_2(B)c_2$$

$$S_2 : s_1(A)r_1(A)x_1(A)w_1(A)a_1u_1(A)x_2(A)w_2(A)x_2(B)w_2(B)u_2(A)u_2(B)c_2$$

$$S_2 : \begin{array}{c|l} t_1 & s(A), r(A), x(A), w(A), u(A), a \\ \hline t_2 & x(A), w(A), x(B), w(B), u(A), u(B), c \end{array}$$

مثال: معادل زمانبندی S_3 را با استفاده از قفل‌های اشتراکی– انحصاری بنویسید.



حل:

$$S_4 : s_4(Q)r_4(Q)x_5(Q)x_3(Q)x_4(Q)w_4(Q)u_4(Q)a_4 \text{ اجابت } w_5(Q)u_5(Q)$$

$$c_5 \text{ اجابت } w_3(Q)c_3u_3(Q)s_6(Q)r_6(Q)u_6(Q)c_6$$

یک تراکنش **باید** در صورت لزوم قفل خود را تبدیل کند. اگر تراکنشی قبلاً قفل s(Q) دارد و بعداً می‌خواهد عمل w(Q) را انجام دهد، باید درخواست x(Q) بدهد. توجه کنید که این درخواست ممکن است اجابت نشود! (در صورتی که تراکنش‌های دیگر هم Q را قفل اشتراکی کرده باشند). همچنین قفل x(Q) هم برای خواندن بعدی باید به s(Q) تبدیل شود تا سطح همروندی بالا برود. این دو مفهوم را به ترتیب افزایش درجه قفل upgrade و کاهش درجه قفل downgrade می‌نامیم.

از مزایای انجام کاهش درجه قفل، افزایش درجه همروندی تراکنشهاست. دقت کنید که اگر قفل x را آزاد کنیم و بعد دوباره تقاضای قفل s بدهیم، ممکن است تراکنش دیگری آن داده را از نوع x قفل نماید و نتوانیم آن قفل را بگیریم. بنابراین تبدیل قفل (کاهش یا افزایش درجه قفل) یک عمل مناسب است.

تمرین: معادل مثال‌های فوق را با قفل باینری بنویسید.

بن‌بست (deadlock) و قحطی (starvation)

زمانبندی زیر را در نظر بگیرید.

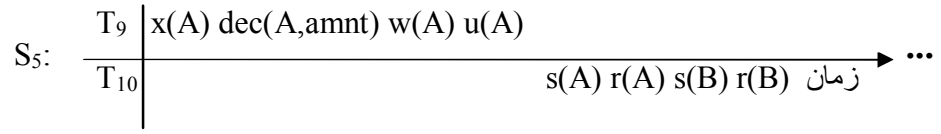
	T_7	$x(B)w(B)$	$x(A) \rightarrow$ انتظار
	T_8	$s(A)r(A)s(B) \rightarrow$ انتظار	

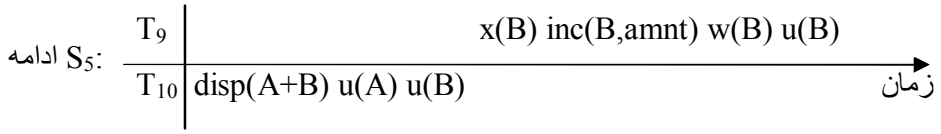
یک راه حل بن‌بست این است که تراکنش‌هایی را که دچار بن‌بست شده‌اند بعضاً ساقط کنیم تا دیگران بتوانند به داده‌های آنها دسترسی پیدا کنند و بن‌بست شکسته شود. اما خود این راه حل ممکن است مشکل دیگری به نام **قحطی** را پدید آورد، بدین صورت که مثلاً یک تراکنش که قصد زدن قفل x روی داده‌ای را دارد، منتظر دنباله‌ای از تراکنش‌ها بماند که همگی قفل s روی همان داده می‌زنند و این انتظار به پایان نرسد. در این صورت تراکنش درخواست دهنده قفل x دچار قحطی شده است.

پس از تکمیل مباحث مربوط به قفل، به مشکل بن‌بست خواهیم پرداخت.

پروتکل‌های قفل دو مرحله‌ای (Two-Phase Locking یا 2PL)

زمانبندی S_5 را در نظر بگیرید. تراکنش T_9 مبلغی را از حساب بانکی A به حساب بانکی B منتقل می‌کند. به این منظور کارهای قفل‌گذاری معمولی را انجام می‌دهد و سپس (با استفاده از دستورهای dec و inc برای کاهش و افزایش موجودی) مبلغی را از حسابی برداشته و به حساب دیگری منتقل می‌کند. بنابراین تراکنش T_9 ابتدا مبلغی را از حساب برداشته و بعد از آن، تراکنش همروند T_{10} جمع موجودی دو حساب را محاسبه می‌کند:





با وجود اینکه همه قوانین قفل‌گذاری رعایت شده، باز هم نتیجه T_{10} غلط است! این مشکل از آنجا ناشی می‌شود که دو داده‌ی A و B در اینجا به یکدیگر وابسته هستند که ما این وابستگی را در نظر نگرفتیم. برای حل این مشکل، پروتکل‌های قفل‌گذاری معرفی گردیده‌اند. پروتکل‌های قفل‌گذاری دو مرحله‌ای فرض می‌کنند که همه داده‌ها به هم وابسته هستند. بنابراین باید همه آنها را با هم در اختیار داشته باشیم. به عبارت دیگر، زمانی که کارمان با یکی از آنها تمام شد، نباید قفل آنها را باز کنیم. در پروتکل‌های قفل‌گذاری دو مرحله‌ای، مرحله اول قفل کردن و مرحله دوم باز کردن قفل است؛ قانون طلایی آن این است که زمانی که اولین قفل داده‌ای را باز کردیم، دیگر اجازه‌ای نداریم که هیچ داه‌ی دیگری را قفل کنیم.

بنابراین پروتکل‌های قفل‌گذاری دو مرحله‌ای شامل دو مرحله است:

* مرحله اول (مرحله رشد - growing)

در این مرحله تراکنش فقط می‌تواند قفل بگیرد (و احتمالاً با داده‌ها کار کند) اما نمی‌تواند قفل را آزاد کند.

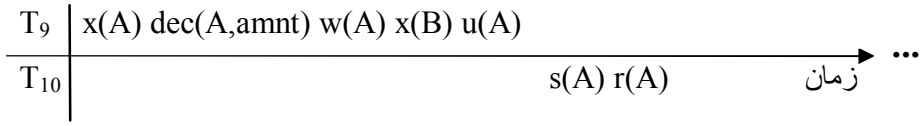
* مرحله دوم (مرحله نقصان یا عقب نشینی - shrinking)

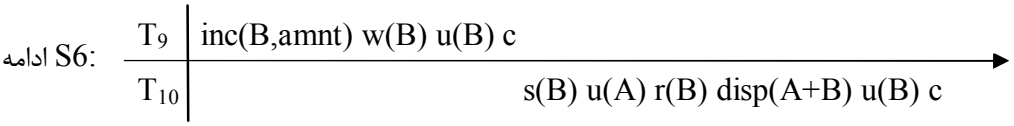
در این مرحله تراکنش فقط می‌تواند قفل‌ها را آزاد کند (و احتمالاً با داده‌ها کار کند) ولی نمی‌تواند قفل جدیدی روی هیچ داده‌ای بگیرد.

از آنجا که ۴ عمل انجام می‌شود یعنی: گرفتن قفل، کار کردن با داده‌ها، باز کردن قفل و پایان دادن به تراکنش، ۴ نوع پروتکل قفل‌گذاری دو مرحله‌ای به وجود می‌آورد.

❖ پروتکل B2PL

در این پروتکل که در واقع **قفل دومرحله‌ای پایه (Basic Two - Phase Locking)** است، فقط همان قواعد گفته شده رعایت می‌شود و هیچ شرط اضافه‌ای روی باز کردن قفل و پایان دادن به تراکنش‌ها وجود ندارد. بنابراین اگر ما قواعد پروتکل B2PL را روی زمانبندی قبلی پیاده کنیم، به این زمانبندی S_6 می‌رسیم:





در پروتکل B2PL امکان بروز بن‌بست وجود دارد. یک راه برای رفع این مشکل نسخه‌ای دیگر از 2PL به نام پروتکل **قفل دو مرحله‌ای محافظه کارانه** یا Conservative Two-Phase Locking یا C2PL است.

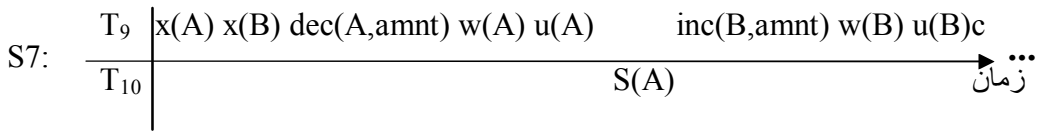
❖ پروتکل C2PL

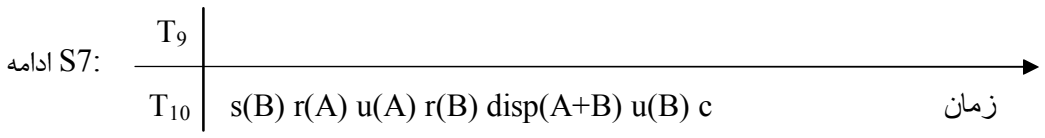
در این پروتکل، زمانبندی‌ها باید برای هر تراکنش سه مرحله قائل بشوند؛ یعنی ابتدا باید تمام قفل‌های مورد نیازش را بگیرند و اگر موفق نشد، دوباره قفل‌های گرفته را باز کند و آن‌قدر تکرار کند تا تراکنش تمام قفل‌هایی را که لازم دارد بگیرد. سپس شروع به اجرا کند، بقیه کار مانند پروتکل B2PL ادامه می‌یابد تا کار تمام شود.

مهمترین مزّیت این پروتکل این است که بن‌بست رخ نخواهد داد، زیرا ما تمام قفل‌ها را گرفتیم و تا زمانی که درخواست قفلی وجود نداشته باشد، طبیعی است که این درخواست‌ها در حلقه نمی‌افتد و بنابراین بن‌بستی رخ نخواهد داد.

مشکلات این روش پایین آمدن سطح همروندی (سرعت) است.

مثال: معادل C2PL زمانبندی S_5 به صورت زیر است:





❖ پروتکل S2PL

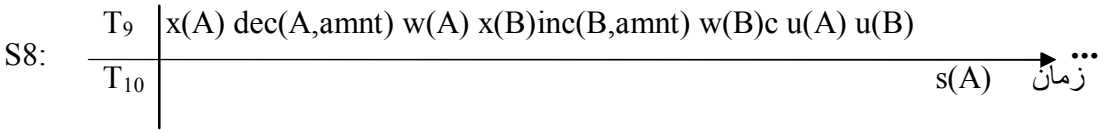
در پروتکل B2PL علاوه بر بن‌بست، امکان سقوط‌های آبشاری (cascading abort) نیز وجود دارد. سقوط آبشاری به این صورت است که اگر تراکنشی به هر دلیلی ساقط شود، ممکن است سقوط آن باعث ساقط شدن پشت سر هم و بدون جهت تراکنش‌های دیگر شود. برای پیشگیری از وقوع سقوط‌های آبشاری، پروتکل قفل دو مرحله‌ای محض (S2PL یا Strict Two-Phase Locking) را معرفی می‌کنیم.

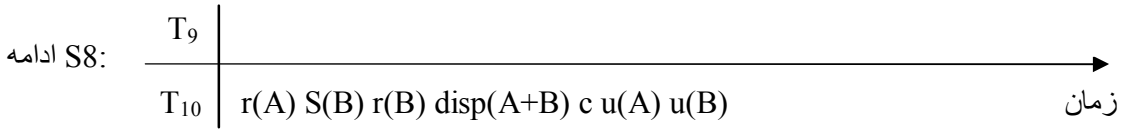
مزّیت اصلی این پروتکل که آن را به پرکاربردترین و بهترین گزینه تبدیل کرده است، چنانکه خواهیم دید، تضمین توأم پی‌درپی‌پذیری و ترمیم‌پذیری می‌باشد.

این پروتکل یک بند به B2PL اضافه می‌کند که این بند این است که وقتی تراکنشی رو به پایان می‌رود، اجازه ندارید قفل آن را باز کنید تا اینکه به پایان برسد. به عبارت دیگر، باز کردن قفل تا بعد از سقوط و تثبیت به تعویق می‌افتد.

مزّیت دیگر این پروتکل، کم کردن پیام‌ها در بانک‌های اطلاعات نامتمرکز است، زیرا نیازی به پیام‌های بازکردن قفل ندارد!

مثال: معادل S2PL زمانبندی S_5 به صورت زیر است:





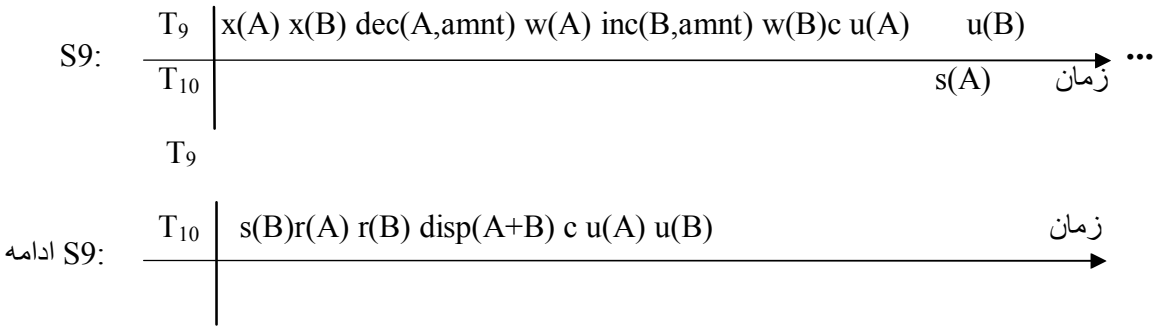
تذکر: می‌توان قفل‌های S(X) را قبل از اتمام تراکنش باز کرد.

❖ پروتکل SC2PL

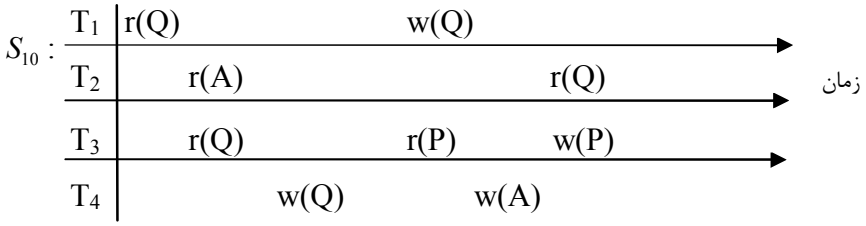
پروتکل SC2PL ترکیبی از دو پروتکل C2PL و S2PL می‌باشد یعنی Strict Conservative Two-Phase Locking که خواص هر دو پروتکل ترکیبی رادارد؛ یعنی نه دچار بن‌بست می‌شود و نیز خواص Strict را دارا می‌باشد.

مسئله بن‌بست اولاً مسئله‌ی مهمی نیست به دلیل اینکه گاهی رخ می‌دهد و ثانیاً راه حل‌های دیگر و بهتری هم دارد. بنابراین نمی‌توان گفت که پروتکل SC2PL پروتکل خیلی خوبی است و در مجموع همان پروتکلS2PL بهترین و کاربردی‌تر است.

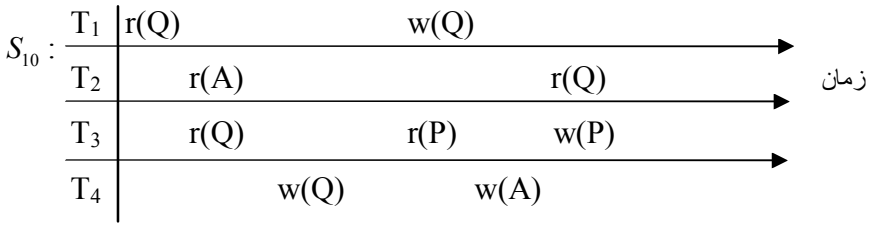
مثال: معادل SC2PL زمانبندی S_5 به صورت زیر است:



مثال: معادل زمانبندی زیر را یکبار با قفل باینری و یکبار با قفل s/x و رعایت پروتکل B2PL بنویسید:

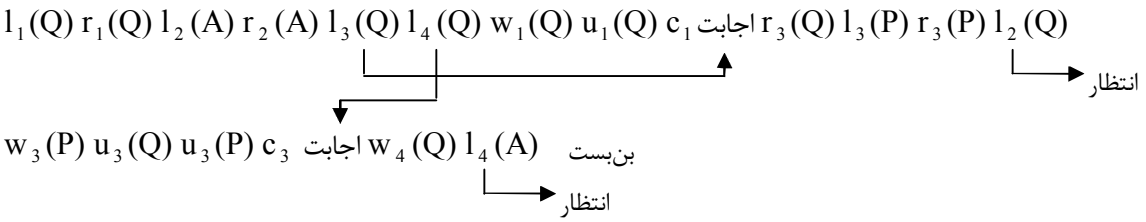


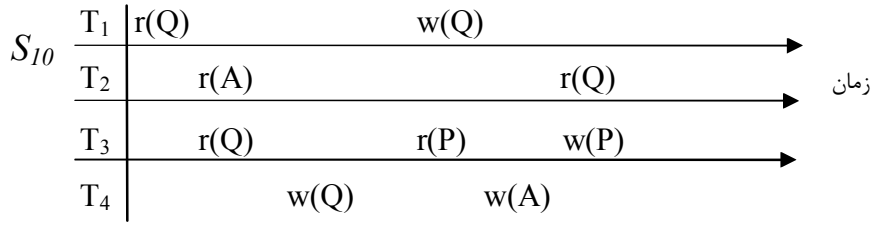
مثال: معادل زمانبندی زیر را یکبار با قفل باینری و یکبار با قفل s/x و رعایت پروتکل B2PL بنویسید:



حل: یک پاسخ ممکن

الف) با استفاده از قفل باینری





ب) با استفاده از قفل‌های s/x

s_1 (Q) r_1 (Q) s_2 (A) r_2 (A) s_3 (Q) r_3 (Q) x_4 (Q) x_1 (Q) s_3 (P) r_3 (P) s_2 (Q) r_2 (Q) u_2 (A)

u_2 (Q) c_2 x_3 (P) w_3 (P) u_3 (Q) u_3 (P) c_3 اجابت w_1 (Q) u_1 (Q) c_1 اجابت w_4 (Q) x_4 (A)w_4 (A)

u_4 (A) u_4 (Q) c_4

تمرین: مثال بالا را با C2PL، S2PL و CS2PL حل کنید.

پروتکل‌های مبتنی بر گراف (graph – based)

معروف‌ترین و پرکاربردترین پروتکل‌ها، پروتکل‌های S2PL یاStrict Two-Phase Locking است. در سیستم‌ها و DBMS های موجود همواره استفاده می‌شود و کمتر اتفاق افتاده که از پروتکل دیگری استفاده کنند.

پروتکل S2PL تمام شرایط صحت و جامعیت بانک اطلاعات را یکجا دارد.

- خاصیت پی‌درپی پذیری
- خاصیت ترمیم پذیری
- از ساقط شدن تراکنش‌ها جلوگیری می‌کند.
- از بازیابی نابهنگام جلوگیری به عمل می‌آورد.

پروتکل‌های مبتنی بر گراف از آنجایی که خیلی مرسوم نیستند، به اختصار به آنها می‌پردازیم. در این پروتکل‌ها مجموعه تمام داده‌های مورد نظر در یک schedule به صورت یک گراف جهت‌دار بدون حلقه (DAG) یا Directed Acyclic Graph نشان داده می‌شوند. به این گراف، گراف بانک اطلاعات (database graph) گفته می‌شود.

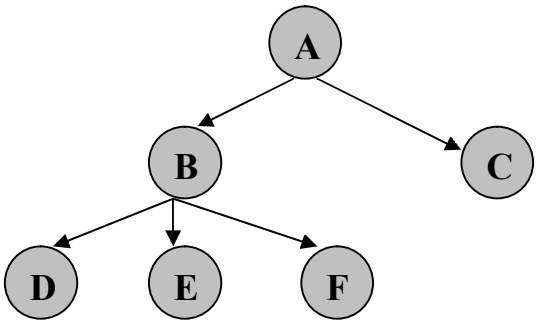
اگر در این گراف بین هر دو گره d_i و d_j (که مربوط به داده‌های متمایز باشند)، لبه‌ای به شکل $d_i \rightarrow d_j$ وجود داشته باشد، آن‌وقت هر تراکنش که می‌خواهد از داده‌های d_i و d_j استفاده نماید، ابتدا باید d_i را قفل کند (یعنی کل مسیر مورد نیاز را قفل نماید).

نوع خاصی از این پروتکل‌ها که پرکاربردتر است، پروتکل درختی tree protocol است که در آن فقط از قفل باینری استفاده می‌شود. در این پروتکل هر تراکنش وقتی شروع می‌کند، اولین قفلش را روی هر داده‌ای می‌تواند بزند. اما از آن پس داده‌ای مثل Q فقط به شرطی می‌تواند قفل شود که پدر داده Q هم توسط همان تراکنش قفل شده باشد.

تذکر: برای هر تراکنش، در شروع گرفتن قفل، داشتن قفل برای پدر ضروری نیست. اما آزاد کردن قفل‌ها در هر زمانی مجاز می‌باشد.

مثال: در درخت زیر، زمانبندی S_{11} قفل‌ها را به صورت زمانبندی S_{12} اخذ می‌کند.

$$S_{11} : r_1(E),\ w_1(C),\ w_2(D),\ r_2(E),\ w_2(C)$$



S_{12} : l_1 (E), r_1 (E), u_1 (E), l_1 (A), l_1 (c), w_1 (c), u_1 (c), u_1 (A), l_2 (D), w_2 (D), u_2 (D), l_2 (A), l_2 (B), l_2 (E), r_2 (E), u_2 (E), u_2 (B), u_2 (A), l_2 (A), l_2 (c), w_2 (c), u_2 (c), u_2 (A),

پروتکل‌های درختی، پی‌در پی پذیری در برخورد و فاقد بن‌بست بودن را تضمین می‌کنند. یکی از مشکلات این دسته پروتکل‌ها این است که بدون دلیل داده‌هایی را قفل می‌کنند. البته زمانی کاربرد دارند که داده‌های ما ارتباط درختی با یکدیگر داشته باشند که معمولاً در بانک اطلاعات معمولی، داده‌ها ارتباط درختی با یکدیگر ندارند.

مجموعه زمانبندی‌های قابل قبول پروتکل درختی با مجموعه زمانبندی‌های قابل قبول پروتکل قفل دومرحله‌ای دقیقاً یکسان نیست، یعنی زمانبندی‌هایی وجود دارند که در پروتکل‌های درختی معتبرند ولی در قفل دو مرحله‌ای قابل قبول نمی‌باشند و بالعکس.

پروتکل‌های مبتنی بر برچسب زمانی (timestamp – based)

این پروتکل‌ها به ویژه در بانک‌های اطلاعات نامتمرکز به کار می‌روند. به هر تراکنشی به محض ورود، یک برچسب زمانی تصاعدی تخصیص داده می‌شود. در بانک‌های اطلاعات متمرکز این کار خیلی ساده است؛ یعنی به محض آنکه تراکنش شروع می‌شود، ساعت سیستم را به عنوان برچسب زمانی به آن اختصاص می‌دهیم و به دلیل آنکه ساعت سیستم در بانک‌های اطلاعاتی متمرکز یک ساعت می‌باشد و درحال افزایش است، timestamp یا برچسب‌های زمانی تصاعدی خواهند بود.

در سیستم‌های نامتمرکز این مسئله کمی مشکل می‌باشد، زیرا ما مطمئن نیستیم ساعت‌هایی که در سیستم در اختیار می‌باشد دقیقاً synchronize یا همزمان هستند یا خیر.

راه حل‌ها:

- با پروتکل‌های موجود در سیستم عامل می‌توان ساعت‌ها را synchronize کرد.
- از یک logical time استفاده کنیم.

«با استفاده از logical time یا physical time ما می‌توانیم ترتیب زمانی تصاعدی داشته باشیم.»

بنابراین می‌توانیم به تراکنش T_i، ترتیب زمانی $TS(T_i)$ بدهیم. اگر تراکنش T_j بعد از تراکنش T_i شروع شود، خواهیم داشت $TS(T_i) < TS(T_j)$. بر این اساس، پروتکل‌های مبتنی بر برچسب زمانی، تراکنش‌ها را به ترتیب برچسب زمانی آنها، به صورت پی‌درپی پذیر اجرا می‌کنند.

برای هر داده Q، برچسب زمانی خواندن و نوشتن آن به صورت زیر تعریف می‌شود:

- W-TS(Q): برچسب زمانی نوشتن داده Q، که برابر است با بزرگترین برچسب زمانی تراکنشی که (به طور موفقیت آمیز) روی Q نوشته است.
- R-TS(Q): برچسب زمانی خواندن Q، که برابر است با بزرگترین برچسب زمانی تراکنشی که (به طور موفقیت آمیز) Q را خوانده.

«به طور موفقیت‌آمیز به این معنی است که تراکنش‌ها commit شده‌اند.»

با اعمال **قواعد** زیر، پروتکل‌های مبتنی بر برچسب زمانی تضمین می‌کنند که دستورات r و w که با هم برخورد دارند، به ترتیب برچسب زمانی اجرا شوند و زمانبندی‌های مربوطه پی‌درپی‌پذیر باشند.

قواعد خواندن:

فرض کنید تراکنش T_i شامل یک دستور read(Q) است:

۱) اگر $TS(T_i) \le W - TS(Q)$ آنگاه تراکنش T_i داده‌ای را می‌خواهد که قبلاً روی آن نوشته شده است. پس در این صورت با دستور خواندن تراکنش موافقت نمی‌شود و تراکنش رد (reject) می‌شود.

۲) اگر $TS(T_i) \ge W - TS(Q)$ آنگاه دستور خواندن تراکنش T_i اجرا می‌شود و برچسب زمانی خواندن Q، با ماکزیمم بین برچسب زمانی تراکنش T_i و برچسب زمانی خواندن Q مقدار دهی می‌شود.

تذکر:

رد (reject) شدن تراکنش به معنی در انتظار ماندن یا سقوط و شروع مجدد می‌باشد.

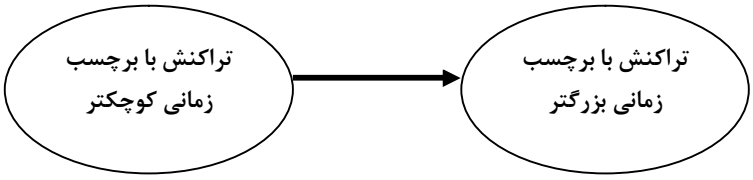
قواعد نوشتن:

فرض کنید تراکنش T_i شامل یک دستور write(Q) باشد:

۱) اگر $TS(T_i) < R - TS(Q)$ یا $TS(T_i) < W - TS(Q)$ باشد، آنگاه با دستور نوشتن تراکنش موافقت نمی‌شود و تراکنش T_i رد (reject) می‌شود.

۲) در غیر این صورت دستور نوشتن تراکنش T_i اجرا می‌شود و نیز برچسب زمانی نوشتن Q با برچسب زمانی تراکنش موجود مقدار دهی می‌شود.

گراف پی‌درپی پذیری زمانبندی‌ها در پروتکل‌های مبتنی بر برچسب زمانی، همیشه مشابه شکل زیر می‌باشد، این گراف مطمئناً فاقد حلقه (cycle) است و زمانبندی مربوطه پی‌درپی پذیر است.



لبه‌ها در گراف پی‌درپی پذیری پروتکل‌های مبتنی بر برچسب زمانی

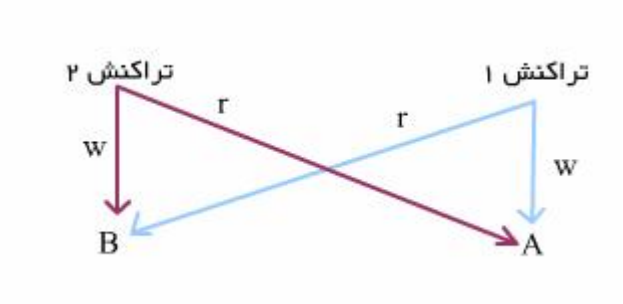
مرجع: [Silberschatz 2005].

اگر در رد کردن تراکنش‌ها (reject)، تراکنش‌ها را restart کنیم کسی منتظر دیگری نمی‌ماند و حالت انتظار پیش نمی‌آید و بن‌بست رخ نمی‌دهد.

بن‌بست

دو یا چند تراکنش در یک حلقه منتظر یکدیگر هستند.

مثال:



روش‌های مدیریت بن‌بست

- **پیشگیری از وقوع بن‌بست (prevention) –**
- **تشخیص و رفع بن‌بست (deadlock detection & resolution) –**

پیشگیری از وقوع بن‌بست (prevention)

این روش تضمین می‌کند که سیستم هرگز دچار بن‌بست نشود. (بدین منظور می‌توان):

- از پروتکل‌هایی استفاده نمود که در آن هر تراکنش قبل از شروع کردن به اجرا، تمام قفلهای مورد نیازش را بگیرد (مثل C2PL و SC2PL).

- بین داده‌ها ترتیبی در نظر گرفته و تراکنش‌ها نیز فقط براساس این ترتیب مجاز به قفل کردن داده‌ها باشند (مثل پروتکل‌های مبتنی بر گراف).
- از مفهوم برچسب‌زمانی مخصوص بن‌بست استفاده نمود و یکی از روش‌های زیر را اعمال کرد:

*روش wait – die: تراکنش پیرتر (با برچسب زمانی کمتر) منتظر تراکنش جوان‌تر می‌ماند تا قفلهایش را آزاد کند. تراکنش جوان‌تر هرگز منتظر تراکنش پیرتر نمی‌ماند بلکه ساقط می‌شود (می‌میرد) و باعث می‌شود بن‌بست پیش نیاید.

- روش wound – wait: تراکنش پیرتر به جای انتظار، تراکنش جوانتر را می‌کشد (او را مجبور به سقوط می‌کند). تراکنش جوان‌تر منتظر تراکنش پیرتر می‌ماند. تراکنش پیرتر الویت بالاتری داشته و فرایند کار را در اختیار ترکنش‌های پیرتر قرار می‌دهیم.

«روش wound – wait ممکن است کمتر منجر به ساقط شدن تراکنش‌ها شود.»

در هر دو روش فوق، تراکنشی که ساقط می‌شود با همان بر چسب زمانی مخصوص بن‌بست شروع به کار مجدد می‌کنند که همین سبب می‌شود تراکنش جوان پس از مدتی تبدیل به یک تراکنش پیر می‌شود.

روش‌های مبتنی بر فرصت (timeout):

در این روش قبل از شروع تراکنش‌ها به آنها یک فرصت یا مهلت زمانی اختصاص می‌دهیم.

مزایا:

- باعث می‌شود اگر تراکنش‌ها در بن‌بست قرار گیرند ساقط خواهند شد و بن‌بست‌ها به طور خودکار شکسته می‌شود.
- افراد و کاربران مختلف نمی‌توانند وقت سیستم را به طور نامحدود بگیرند.

معایب:

به سادگی نمی‌توان میزان فرصت یا timeout را تعیین کرد.

تشخیص و رفع بن‌بست (deadlock detection & resolution)

برای تشخیص بن‌بست گرافی به نام **گراف انتظار** Wait-For Graph وجود دارد که گره‌های آن تراکنش‌ها هستند و لبه $T_i \rightarrow T_j$ در صورتی وجود دارد که تراکنش T_i منتظر T_j باشد. چنانچه در این گراف حلقه‌ای وجود داشته باشد، در این صورت بن‌بست به وجود آمده است.

گراف انتظار را رسم می‌کنیم و هر عملی که انجام شد با توجه به لبه‌های به وجود آمده گراف را up to date می‌کنیم و به صورت پریودیک (چند ثانیه یا دقیقه‌ای یا هر چند دقیقه یکبار) چک می‌کنیم که آیا در گراف دور یا cycle وجود دارد یا خیر. اگر وجود داشت سعی می‌کنیم با abort یا ساقط کردن یک یا چند تراکنش، حلقه را از بین ببریم.

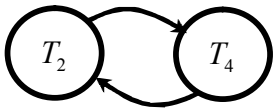
«تراکنشی که باید ساقط شوند قربانی (victim) گفته می‌شود.»

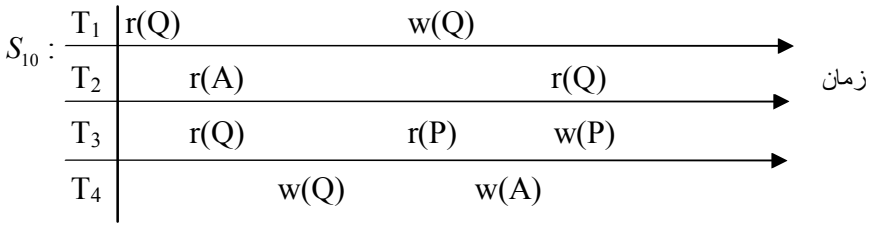
مهمترین معیارهای انتخاب قربانی عبارتند از:

- تعداد حلقه‌هایی از گراف انتظار که این تراکنش در آنها شرکت دارد حداکثر باشد.
- حجم کاری که تا کنون در آن تراکنش صورت گرفته است کمتر باشد.
- تعداد به‌روزرسانی‌هایی که تراکنش انجام داده کمتر باشد.
- حجم کارباقی مانده تراکنش کمتر باشد. (خیلی ساده نیست)

مثال:

گراف انتظار زمانبندی S_{10} با قفل باینری به شکل زیر می‌باشد:





همواره هرگاه تراکنش‌ها را قربانی یا abort می‌کنیم ممکن است قحطی ایجاد شود. یک راهکار برای رفع مشکل قحطی این است که تعداد دفعات ساقط شدن تراکنش را یادداشت کنیم و اگر تعداد دفعات ساقط کردن تراکنشی از یک حدی گذشت، دیگر آن را ساقط نکنیم و تراکنش‌های دیگر را قربانی کنیم.

ج- مدیریت ترمیم
(Recovery Management)

مقدمه

در مدیریت تراکنشها، این امکان وجود دارد که در هر لحظه از اجرای تراکنش، یک خرابی (failure) اتفاق بیفتد و تراکنش نتواند ادامه یابد. به عبارتی برخی از دستورات آن اجرا شده و بقیه به دلیل وقوع خرابی متوقف شوند. همچنین زمانی که تراکنشی انجام (commit) می‌شود، بعداً اتفاقی بیفتد که تأثیر آن را از بین ببرد. این دو نوع اتفاق به عنوان failureها مطرح هستند و ما باید آنها را ترمیم کنیم؛ یعنی باید کاری کنیم که اگر چنین اتفاقاتی افتاد، تأثیر مخربی روی بانک اطلاعات نگذارند.

به عبارت دیگر طبق خواص پایایی (durability) و یکپارچگی (atomicity) که برای تضمین جامعیت بانک اطلاعات توسط تراکنش‌ها الزامی بوده و این دو از چهار خواص ACID هستند، هر تراکنشی که انجام یا تثبیت می‌شود، باید پس از آن اثرات آن در بانک اطلاعات حتی در صورت وقوع خرابی، دائمی و همیشگی باشد و نیز برای تراکنشی که فقط برخی از دستورات آن اجرا شده‌اند، باید اثر دستورات اجرا شده آن خنثی (بلااثر- ملغی – کأن‌لم‌یکن) گردد.

تأمین این دو ویژگی از جمله وظایف بخشی از مدیریت تراکنش‌ها به نام **واحد مدیریت ترمیم (recovery management component)** می‌باشد که در این بخش با آن آشنا می‌شویم.

انواع خرابی (failure)

خرابی‌هایی که در یک سیستم بانک اطلاعات (متمرکز) ممکن است اتفاق بیفتد را می‌توان به سه دسته زیر تقسیم نمود:

۱. خرابی تراکنش (transaction failure)

ممکن است جایی در اجرای تراکنش، منطق آن دچار خطا شود. لذا خرابی تراکنش شامل دو مورد زیر است:

- **خطای منطقی**: تراکنش به دلیل شرایط داخلی خود نتواند کامل اجرا شود.
- **خطای سیستمی**: تراکنش به خودی خود درست کار می‌کند اما در سیستم شرایطی پیش می‌آید که این تراکنش را از کار می‌اندازد. بارزترین مصداق این دسته، وقوع بن‌بست در سیستم است.

۲. خرابی سیستم (system crash)

خرابی‌های سخت‌افزاری یا نرم‌افزاری که سبب از کار افتادن سیستم (down شدن) می‌شود مثل قطع برق و یا باگها و خطاهای موجود در نرم‌افزارها و ... رایج‌ترین نوع خرابی هستند. در این نوع خرابی‌ها اطلاعات حافظه اصلی سیستم از بین می‌رود ولی آسیبی به اطلاعات روی دیسک (حافظه جانبی-رسانه) وارد نمی‌شود.

۳. خرابی رسانه

خرابی که سبب شود اطلاعات روی رسانه از بین برود یا قابل بازیابی نباشد، مانند خرابی دیسک یا خرابی هد و یا کنترلر دیسک.

«به طور پیش‌فرض، بانک اطلاعات روی رسانه‌ها یا حافظه‌های جنبی نگهداری می‌شود.»

انواع رسانه‌های ذخیره‌سازی

از جهت قابلیت نگهداری اطلاعات در صورت وقوع خرابی، رسانه‌های ذخیره سازی به سه دسته تقسیم می‌شوند:

- **رسانه فرار (volatile storage)**

رسانه‌ای که در صورت وقوع خرابی سیستم، اطلاعات آن از بین می‌رود، مثل حافظه اصلی، حافظه نهان (cache) و ثبات (register).

- **رسانه غیرفرار (non-volatile storage)**

خرابی سیستم را تحمل می‌کنند و اطلاعات آنها حتی با وقوع خرابی سیستم، قابل بازیابی است. مانند دیسک، نوار مغناطیسی، حافظه flash یا حتی RAM ای که دارای باتری پشتیبان است.

- **رسانه پایدار (stable storage)**

رسانه‌ای که ایده آل ماست و در برابر تمام خرابی‌ها مصون می‌باشد. این نوع رسانه در واقع یک مفهوم منطقی است نه فیزیکی و هنوز چنین رسانه‌ای وجود خارجی ندارد، بلکه سعی می‌کنیم با راهکارهایی که از رسانه‌های فیزیکی موجود استفاده می‌کنند به این هدف ایده‌آل نزدیک شویم.

متداول‌ترین راهکار، استفاده از چندین کپی از داده‌ها روی رسانه‌های مختلف است (backup). با این کار سعی می‌کنیم احتمال از دست رفتن اطلاعات در صورت وقوع خرابی را به صفر هر چه نزدیکتر کنیم. هر چه تعداد کپی‌ها، ناهمگونی رسانه‌ها، درجه اطمینان آنها و ... بیشتر باشد، احتمال از دست رفتن کامل اطلاعات کمتر است. اما کپی کردن‌ها، به روز نگه داشتن داده‌ها و مدیریت آنها پیچیدگی‌های زیادی را به دنبال دارد. یکی از سیستم‌های رایج در این راستا RAID یا Redundant Array of Independent Disks است که در ضمیمه اول معرفی شده است.

روال دسترسی به داده‌ها برای انجام تراکنش

داده‌های بانک اطلاعات روی رسانه‌ها قرار دارند که معمول‌ترین آنها دیسک می‌باشد. عمده‌ترین عامل سرعت در پردازش اطلاعات، **تعداد دفعات دستیابی به رسانه** است. در بانک‌های اطلاعات نامتمرکز علاوه بر این عامل، **پهنای باند** نیز اهمیت دارد.

برای کاهش تعداد دفعات مراجعه به دیسک، رکوردها با هم بلوک‌بندی می‌شوند و در هر بار مراجعه به دیسک برای خواندن یا نوشتن، به جای یک رکورد، یک بلوک خوانده می‌شود. پس از خواندن یک بلوک، اطلاعات آن بلوک به بخشی از حافظه اصلی منتقل می‌شوند که بافر (buffer) یا حافظه میان‌گیر نامیده می‌شود. سپس تراکنش‌ها از این حافظه‌های میان‌گیر که به صورت RAM هستند و سرعت خیلی بالایی دارند، رکوردها را برداشته و عملیات‌ها را انجام می‌دهند. خروجی برعکس این فرایند است؛ یعنی تراکنش‌ها از ناحیه کاری (work area) خود، رکوردها را روی بافرها نوشته و با پرشدن بافر، به یکباره به دیسک منتقل می‌شوند.

مرحله اول این کار یعنی انتقال اطلاعات از دیسک به بافر و بالعکس، input() و output() نام دارد و مرحله دوم یعنی انتقال اطلاعات از بافرها به ناحیه‌های کاری تراکنش‌ها و بالعکس به نام read() و write() مرسوم است.

سیستم‌های مختلف تعدادی بافر دارند که دارای سایز مشخصی هستند. بنابراین سایز بلوک‌هایی که تعیین می‌کنیم، یعنی تعداد رکوردهایی که در بلوک‌ها قرار می‌دهیم با توجه به سایز بافر می‌باشد. اگر سایز بلوک‌ها از سایز بافرها کمتر باشد در این صورت فضای بافر را از دست داده‌ایم و اگر سایز بلوک‌ها از سایز بافرها بیشتر باشد، در این صورت باید بافرها را به هم چسباند و آنها را خواند که کار بسیار سختی است. بنابراین سایز بلوک‌ها باید هر چه نزدیک‌تر به سایز بافرها باشد و نه بیشتر.

ثابت شده که برای هر نوع IO بهتر است دو بافر اختصاص دهیم. یک بافر توسط IO Proccessor پر می‌شود؛ یعنی بلوک در آنجا قرار می‌گیرد که عملی طولانی است. بافر دیگر در صورتی که پر شده باشد به طور همزمان توسط cpu خالی می‌شود؛ یعنی به ترتیب رکوردهای آن در اختیار ناحیه کاری تراکنش قرار می‌گیرند و زمانی بافر پر و دیگری خالی شد، نقش آنها عوض می‌شود. شکل، روال را نمایش می‌دهد.

«به ازای هر IO، داشتن دو بافر بهترین حالت ممکن است.»

الگوریتم‌های ترمیم

برای تضمین جامعیت بانک اطلاعات و اعمال خواص یکپارچگی و پایایی تراکنشها در صورت وقوع خرابی، از الگوریتم‌هایی استفاده می‌کنیم که عموماً شامل دو مرحله زیر می‌باشند:

۱. **مرحله اول:** در حین عملکرد عادی سیستم، اطلاعاتی که برای انجام ترمیم (پس از وقوع خرابی) به آنها نیاز داریم در جایی ثبت شوند. به این جا معمولاً Log می‌گوییم به معنی کارنامه.

۲. **مرحله دوم:** پس از وقوع خرابی (وقتی سیستم دوباره بالا می‌آید)، با استفاده از اطلاعات ثبت شده در مرحله قبل و الگوریتم‌هایی که از پیش تهیه شده‌اند، سیستم را به حالت اول برمی‌گردانیم.

طبق خاصیت پایایی، اثرات تراکنشی که انجام شده باید دائمی و همیشگی گردد. طبق خاصیت یکپارچگی، تراکنش‌هایی که نیمه کاره متوقف شده‌اند، نباید هیچ اثری روی بانک اطلاعات گذاشته باشند.

پس در مرحله دوم از الگوریتم ترمیم، هر تراکنشی که انجام (commit) شده است را **تکرار (redo)** کرده، یعنی دوباره اجرا می‌کنیم تا اثراتش روی بانک اطلاعات دائمی شود و هر تراکنشی که ساقط شده را **خنثی (undo)** می‌کنیم. فرض می‌کنیم تراکنشها پی‌درپی هستند.

الگوریتم‌های ترمیم به دو رویکرد کلی تقسیم می‌شوند: رویکرد **کارنامه (log-based)** و رویکرد **رونوشت (shadow paging)**.

- رویکرد کارنامه

در این رویکرد اطلاعات مورد نیاز برای انجام ترمیم را در حافظه پایدار ثبت می‌کنیم. برای هر یک از دستورات یک تراکنش، رکوردی به نام رکورد کارنامه (log) با ساختار زیر نوشته می‌شود:

- برای شروع تراکنش T_i، رکورد $< T_i, start >$
- برای دستور write(x) از تراکنش T_i، در حالت کلی رکورد $< T_i, x, V_1, V_2 >$ که V_1 و V_2 به ترتیب مقادیر قبل و بعد از انجام نوشتن x می‌باشند.
- با اجرای آخرین دستور تراکنش T_i، رکورد $< T_i, Commit >$ در کارنامه ثبت می‌شود.

در رویکرد مبتنی بر کارنامه که رایج‌ترین روش انجام ترمیم است، انعکاس تغییرات روی بانک اطلاعات (روی رسانه) به یکی از دو روش زیر می‌تواند صورت پذیرد:

- ❖ انعکاس معوق تغییرات در بانک‌اطلاعات (deferred database modification)
- ❖ انعکاس فوری تغییرات در بانک‌اطلاعات (immediate database modification)

انعکاس معوق تغییرات در بانک‌اطلاعات

در این روش تمام رکوردهای کارنامه مربوط به انجام تغییرات، در کارنامه ثبت می‌شود اما انعکاس این تغییرات روی رسانه (یعنی اجرای واقعی write ها) تا زمان پس از اجرای آخرین دستور تراکنش یا انجام جزئی (partial commit) به تعویق می‌افتد.

با اجرای دستور نوشتن تراکنش، عمل نوشتن روی رسانه انجام نمی‌شود بلکه تمام دستورات نوشتن پس از مرحله انجام جزئی با استفاده از کارنامه انجام خواهد شد.

چنانچه خرابی در سیستم اتفاق بیفتد، در واقع بانک اطلاعات ما هیچ تغییری نکرده و می‌توانیم بعداً با استفاده از این log ها، تغییرات را روی بانک اطلاعات اعمال کنیم.

نکته: چنانچه از روش انعکاس معوق تغییرات استفاده کنیم، در انجام ترمیم نیازی به خنثی کردن نداریم، زیرا تراکنش‌هایی که تثبیت نشده‌اند، به مرحله انجام جزئی نرسیده‌اند و بنابراین هیچ‌گونه تغییری روی بانک اطلاعات ما صورت نگرفته است.

انعکاس فوری تغییرات در بانک اطلاعات

در این روش به محض اجرای دستور نوشتن تراکنش، آن عمل فوراً روی رسانه نیز منعکس می‌شود. در نتیجه تراکنش‌ها در حین اجرا بانک‌های اطلاعات را تغییر می‌دهند. بنابراین ممکن است خرابی اتفاق بیفتد و نیاز به خنثی کردن تراکنشهای نیمه کاره داشته باشیم.

سؤال مهمی که در این روش مطرح می‌شود این است که آیا ترتیب " نوشتن روی کارنامه (log)" و "نوشتن روی رسانه (بانک اطلاعات)" اهمیتی دارد؟

به عبارت دیگر: آیا برای یک دستور نوشتن، ترتیب "نوشتن روی رسانه" و "ثبت روی کارنامه" تأثیر و اهمیتی دارد؟

برای پاسخ به این سؤال دو سناریوی زیر را در نظر بگیرید:

الف) ابتدا دستور نوشتن را روی رسانه اجرا کنیم و سپس رکورد کارنامه را ثبت کنیم.

ممکن است بین اجرای این دو دستور خرابی اتفاق بیفتد و چون رکورد کارنامه ثبت نشده است در مرحله دوم الگوریتم ترمیم، نمی‌توانیم ترمیم را انجام دهیم.

ب) ابتدا رکورد مربوط به دستور نوشتن روی کارنامه ثبت و سپس دستور نوشتن روی رسانه اجرا شود. در این صورت حتی اگر بین این دو دستور هم خرابی رخ دهد، چون قبلاً رکورد نوشتن را ثبت کرده‌ایم، می‌توانیم عمل ترمیم را انجام دهیم.

این قاعده مهم که ثبت رکورد مربوط به دستور نوشتن در کارنامه باید قبل از اجرای دستور نوشتن روی رسانه صورت گیرد، با نام پروتکل WAL یا Write-Ahead Log نام دارد.

در مرحله دوم از الگوریتم‌های ترمیم که تغییرات فوری را روی دیسک منعکس می‌کنند، تمام تراکنش‌هایی مثل T_i که انجام شده باشند، هم رکورد $< T_i, Start >$ و هم رکورد $< T_i, Commit >$ برای آنها روی کارنامه وجود دارد و تمام تراکنش‌هایی مانند T_j که آغاز شده اما انجام نشده‌اند، رکورد $< T_j, Start >$ آن روی کارنامه موجود است ولی رکورد $< T_j, Commit >$ وجود ندارد.

تکرار کردن یعنی قرار دادن مقدار جدید و خنثی کردن شامل قرار دادن مقدار قدیمی در متغیر مربوطه می‌باشد.

چند نکته مهم:

۱- ابتدا خنثی کردن‌ها و سپس تکرارها را انجام می‌دهیم.

۲- برای تکرار، از آغاز کارنامه شروع کرده و به ترتیب تراکنش‌های مربوطه را تکرار می‌کنیم تا به پایان کارنامه برسیم. اما برای خنثی کردن از انتهای کارنامه به سمت شروع کارنامه پیش می‌رویم و تراکنشهای مورد نظر را خنثی می‌نماییم؛ یعنی اگر مثلاً به ترتیب تراکنشهای T_1، T_2 و T_3 وارد سیستم شده باشند و هر سه نیاز به خنثی کردن داشته باشند، آنها را به ترتیبِ $T_3 < T_2 < T_1$ خنثی می‌نماییم:

۳- از آنجا که خرابی هم ممکن است در حین عملکرد عادی سیستم اتفاق بیفتد و هم در زمان انجام ترمیم، لذا خنثی و تکرار کردن باید به گونه‌ای باشند که اثر چندین بار اجرا شدن آنها معادل با اثر فقط یکبار اجرا شدن آنها باشد. این خاصیت را **همانی بودن (idempotent)** گوییم.

رویکرد کارنامه معایبی دارد که مهمترین آنها عبارتند از:

۱. بزرگ شدن اندازه فایل کارنامه.
۲. زمان‌گیر بودن جستجو در کارنامه.
۳. احتمال تکرار مجدد تراکنشهایی که قبلاً آنها را تکرارکرده‌ایم.
۴. افزایش هزینه به‌روزرسانی بانک اطلاعات به دلیل کار با رسانه‌ها.

برای رفع یا بهبود این معایب روشهایی ارائه شده که از جمله بهترین‌ها، استفاده از **نقطه بازرسی** (checkpoint) می‌باشد.

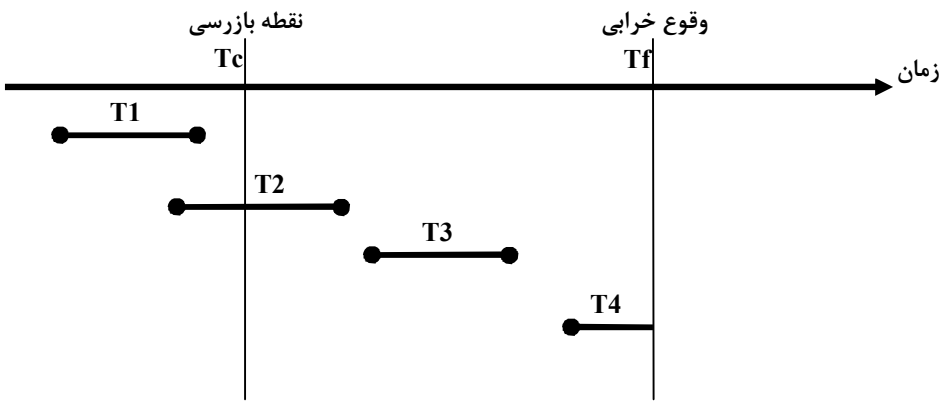
نقطه بازرسی

در این روش، به طور متناوب (periodic) در برهه‌های زمانی معینی از عملکرد عادی سیستم، کارهای انجام شده روی بانک اطلاعات (تا آن زمان) را قطعی و نهایی می‌کنیم که این لحظات نقاط بازرسی سیستم نام دارند. بنابراین اگر اتفاقی در سیستم رخ بدهد، نباید تا ابتدای log را بررسی کنیم، بلکه کافیست تا اولین نقطه بازرسی این کار را انجام دهیم. به عبارتی دیگر کافیست log را از آخر شروع کنیم، اطلاعات آن را مورد استفاده قرار دهیم، سیستم را ترمیم کنیم و زمانی که به اولین نقطه بازرسی رسیدیم متوقف شویم.

در هر نقطه بازرسی مجموعه عملیات زیر به انجام می‌رسند:

- رکوردهای کارنامه (که در حین عملکرد عادی سیستم در بافر نوشته می‌شدند) به حافظه پایدار منتقل می‌شوند.
- داده‌های تغییر یافته در بافر به دیسک منتقل می‌گردند.
- رکوردی به نام رکورد بازرسی با ساختار <checkpoint> ثبت می‌شود.

مثال: در این شکل پس از وقوع خرابی، به صورت زیر عمل می‌شود:



- تراکنش T_1 در زمان T_c نهایی شده است و نیاز به ترمیم ندارد.

- تراکنشهای T_2 و T_3 را بنا به خاصیت پایایی باید تکرار نمود تا اثر آنها در سیستم نهایی و دائمی شود (برای T_2 فقط بخشی از دستورات که پس از T_c اجرا شده‌اند را تکرار می‌کنیم).

- تراکنش T_4 که نیمه کاره مانده است را بنا به خاصیت یکپارچگی، خنثی می‌کنیم.

رویکرد رونوشت

رویکرد رونوشت خیلی معمول نیست که مهمترین دلیل این است که برای پیاده‌سازی رویکرد رونوشت، ما به کارنامه نیاز داریم. حال اگر قرار است که کارنامه و الگوریتم‌های پردازش کارنامه را داشته باشیم تا رویکرد رونوشت را پیاده‌سازی کنیم، بهتر همان است که رویکرد مبتنی بر کارنامه را انجام دهیم.

قبل از انجام تغییرات (قبل از شروع به اجرای تراکنش) یک کپی از صفحات مورد نیاز آن تراکنش از بانک‌اطلاعات تهیه می‌کنیم. به این نسخه، نسخه **جاری (current) می‌گوییم**. تغییرات مورد نظر تراکنش روی نسخه جاری صورت می‌گیرد بنابراین نسخه اصلی دست نخورده باقی می‌ماند. پس ما دو نسخه داریم: نسخه اصلی که نسخه **سایه (shadow)** نیز نامیده می‌شود به دلیل اینکه وقتی تراکنش‌ها موفق می‌شوند، آن نسخه دور ریخته می‌شود و نسخه دیگر، نسخه جاری که تغییرات را روی آن انجام می‌دهیم. چنانچه تراکنش به انجام رسید، این نسخه جاری را به عنوان بانک اطلاعات جدید تلقی می‌کنیم و نسخه سایه را از بین می‌بریم. اما اگر تراکنش نتوانست انجام شود، نسخه جاری را از بین برده و بانک اطلاعات معادل همان نسخه سایه خواهد بود.

مزایای رویکرد رو نوشت:

۱. سربار مربوط به نوشتن رکوردهای کارنامه را ندارد.
۲. انجام عملیات ترمیم بسیار ساده و ناچیز است.

معایب رویکرد رونوشت:

۱. برای پیاده‌سازی آن نیاز به کارنامه داریم.
۲. کپی کردن اطلاعات مربوطه از بانک بسیار پر هزینه است (حتی با وجود بهینه‌سازی‌های انجام شده در این زمینه).
۳. توسعه این روش برای حالتی که تراکنشها بتوانند همروند اجرا شوند بسیار مشکل است (بر خلاف روش کارنامه).
۴. سربار انجام عملیات انجام زیاد است.

نتیجه: با توجه به مقایسه اجمالی این دو رویکرد، در سیستم‌های بانک اطلاعات رویکرد کارنامه از رویکرد رونوشت بهتر است و عموماً از روش کارنامه برای انجام ترمیم استفاده می‌شود.

د- مبانی نظری مدیریت تراکنش

در این بخش ما تقریباً مفهوم تازه‌ای را بیان نمی‌کنیم بلکه همان مفاهیم سابق را که شامل پی‌درپی پذیری، ترمیم پذیری و پروتکل‌های مربوطه بودند به صورت ریاضی بیان می‌کنیم و سعی می‌کنیم معانی آنها را به صورت کاملاً دقیق عرضه کرده و سپس خواص آنها را اثبات کنیم. بنابراین دو هدف را دنبال می‌کنیم:

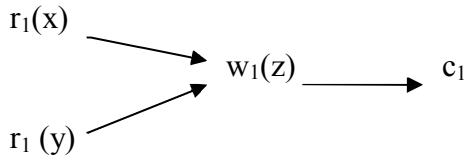
۱- بیان دقیق و ریاضی مفاهیم

۲- اثبات خواص آنها

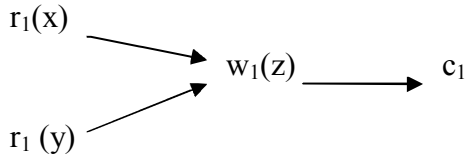
پی‌درپی‌پذیری

در رابطه با پی‌درپی‌پذیری، ابتدا تراکنش و سپس زمانبندی را و آنگاه زمانبندی‌های معادل را بیان کردیم و در اینجا نیز به آنها اشاره می‌کنیم. علائمی که در اینجا استفاده می‌شود، یکی گراف جهت‌دار فاقد حلقه‌ای (DAG یا Directed Acyclic Graph) است که با آن می‌توانیم هم تراکنش‌ها وزمانبندی‌ها را نشان دهیم که نمونه‌ای از این گراف در این مثال آمده است. دیگری علائم ریاضی است که باز هم توسط آنها تراکنش‌ها، زمانبندی‌ها و مفاهیم دیگر را بیان می‌کنیم.

مثال: شکل زیر نشان می‌دهد که تراکنش T_1 به طور همزمان داده‌های x و y را خوانده و پس از این دو رویداد، روی داده z نوشته و بعداً به انجام رسیده است.



پس در اینجا دیده می‌شود که تراکنش یک partial order است. partial order یا ترتیب جزئی یعنی یک مجموعه‌ای که بعضی یا تعدادی از اعضای آن ترتیب خاصی دارند و اگر در یک مجموعه‌ای همه‌ی اعضاء ترتیب مشخصی داشته باشند به آن total order یا ترتیب کلی می‌گوییم. یک تراکنش به تنهایی در صورتی یک partial order است که ما بیش از یک cpu و یا IO processor در اختیار داشته باشیم.



تعریف: دو عملگر با هم **برخورد conflict** دارند اگر، مربوط به تراکنش‌های متمایز باشند، روی یک داده کار کنند و حداقل یکی از آنها عملگر نوشتن باشد:

$$p_i(x) \not\prec q_j(y) \Leftrightarrow (i \neq j \wedge x = y \wedge (p = w \vee q = w))$$

برای دو عملگری که با هم برخورد دارند، هم در تراکنش و هم در زمانبندی‌ها حتماً باید ترتیب اجرای آنها را نسبت به هم تعیین کنیم، زیرا:

- در مورد $r_i(x)$ و $w_j(x)$، نتیجه خواندن داده، قبل و بعد از نوشتن روی آن داده ممکن است متفاوت باشد و مقداری که خوانده می‌شود بستگی به این دارد که خواندن، قبل از نوشتن اجرا می‌شود یا بعد از آن.
- در مورد $w_i(x)$ و $w_j(x)$، مقدار نهایی نوشته شده در داده (x) بستگی به این دارد که کدام عملگر w، آخر اجرا شود.

تعریف: تراکنش T_i، زوج مرتبی است که Σ_i و $<_i$ علائم عملگرها و ترتیب‌های آن هستند و دارای خواص زیر می‌باشند:

۱. T_i شامل مجموعه عملگرهای خواندن و نوشتن داده‌ها و تثبیت یا سقوط می‌باشد.

$$T_i \subseteq \{r_i(x), w_i(x) \mid x \text{ داده است}\} \cup \{a_i, c_i\}$$

۲. فقط یکی از دو عملگر تثبیت یا سقوط باید وجود داشته باشد و نه هردوی آنها.

$$a_i \in T_i \Leftrightarrow c_i \notin T_i$$

۳. عملگر تثبیت یا سقوط آخرین عملگر است.

$$(t = c_i \vee t = a_i) \Rightarrow \forall p \in T_i,\ p <_i t$$

۴. ترتیب اجرای خواندن و نوشتن تراکنش روی یک داده حتماً باید مشخص شده باشد.

$$\forall o_i(x), w_i(x) \in T_i \Rightarrow o_i(x) <_i w_i(x) \vee w_i(x) <_i o_i(x)$$

زمانبندی کامل (complete shedule)

تعریف: اگر $T = \{T_1, T_2, ..., T_n\}$ مجموعه تراکنش‌ها باشد، آنگاه **زمانبندی کامل** H روی T، ترتیب جزئی با رابطه $<_H$ است که:

۱. H شامل تمام تراکنش‌ها می‌باشد:

$$H = \cup_{i=1}^{n} T_i$$

۲. ترتیب اجرای دستورات تراکنش‌ها حفظ می‌گردد و ترتیب‌های دیگری نیز ممکن است اضافه شوند:

$$<_H \supseteq \cup_{i=1}^{n} <_i$$

۳. برای هر دو عملگر دارای برخورد، باید ترتیب آنها مشخص شود:

$$\forall\ p_i(x), q_j(y) \in \mathrm{T}: p_i(x) \asymp q_j(y) \Rightarrow p_i(x) <_H q_j(y) \vee q_j(y) <_H p_i(x)$$

زمانبندی (shedule)

تعریف: زمانبندی، پیشوندی (prefix) از یک زمانبندی کامل می‌باشد. پیشوند یعنی از ابتدا شروع کنید و تا یک جایی پیش بروید. آنچه که در واقعیت بانک‌های اطلاعاتی وجود دارد زمانبندی است. زمانبندی یک مفهوم پویاست در حالی که زمانبندی کامل یک مفهوم ایستا می‌باشد.

مثال: تراکنش‌های زیر را در نظر بگیرید:

$T_1 = r_1(x) \rightarrow r_1(y) \rightarrow w_1(x) \rightarrow c_1$
$T_2 = r_2(x) \rightarrow w_2(y) \rightarrow w_2(x) \rightarrow c_2$
$T_3 = r_3(y) \rightarrow w_3(x) \rightarrow w_3(y) \rightarrow c_3$

زمانبندی کامل H_1 روی مجموعه تراکنش‌های بالا به صورت زیر باشد:

$$H_1 = \begin{array}{l} r_1(x) \rightarrow r_1(y) \rightarrow w_1(x) \rightarrow c_1 \\ \quad\quad\quad\quad \downarrow \quad\quad\quad\quad \uparrow \\ r_2(x) \rightarrow w_2(y) \rightarrow w_2(x) \rightarrow c_2 \\ \quad\quad\quad\quad \downarrow \quad\quad\quad\quad \uparrow \\ \quad\quad\quad\quad r_3(y) \rightarrow w_3(x) \rightarrow w_3(y) \rightarrow c_3 \end{array}$$

زمانبندی H'_1 پیشوندی از H_1 است:

$$H'_1 = \begin{array}{l} r_1(x) \rightarrow r_1(y) \rightarrow w_1(x) \\ \quad\quad\quad\quad \downarrow \quad\quad\quad\quad \uparrow \\ r_2(x) \rightarrow w_2(y) \rightarrow w_2(x) \rightarrow c_2 \\ \quad\quad\quad\quad \downarrow \quad\quad\quad\quad \uparrow \\ \quad\quad\quad\quad r_3(y) \rightarrow w_3(x) \end{array}$$

تعریف: تراکنش T_i در H تثبیت شده است اگر $c_i \in H$ و ساقط شده اگر $a_i \in H$. در غیر این صورت T_i **فعال (active)** است.

تذکر: زمانبندی کامل، تراکنش فعال ندارد.

پرتو ثابت (committed projection)

تعریف: پرتو ثابت زمانبندی H که با C(H) نمایش داده می‌شود، با حذف تمام عملگرهای تراکنش‌های تثبیت نشده حاصل می‌گردد.

$$C(H) = \{ p_i(x) \mid 1 \le i \le n, \quad p_i(x) \in H \wedge c_i \in H \}$$

تذکر: پرتو ثابت، یک زمانبندی کامل روی مجموعه تراکنش‌های تثبیت شده می‌باشد که نه تراکنش فعال دارد و نه تراکنش ساقط شده.

باید توجه داشت که بر روی تراکنش‌های فعال نمی‌توان حساب کرد، زیرا تراکنش‌های فعال ممکن است به دلیل وقوع خرابی ساقط شوند و تراکنش‌های ساقط شده نیز تأثیری روی بانک‌اطلاعات ندارند.

«پرتوی ثابت، بخش ارزشمند یک زمانبندی است.»

پی‌درپی‌پذیری در برخورد

زمانبندی‌های معادل در برخورد

تعریف: دو زمانبندی H و H' **معادل در برخورد** هستند اگر و تنها اگر:

۱- مجموعه تراکنش‌ها و مجموعه عملگرهایشان یکسان باشد.

۲- ترتیب عملگرهای دارای برخورد تراکنش‌های ساقط نشده، در هر دو، یکسان باشد:

$$\forall p_i(x), q_j(y) : p_i(x) \asymp q_j(y) \wedge$$
$$a_i, a_j \notin H \wedge a_i, a_j \notin H' (p_i(x) <_H q_j(y) \Leftrightarrow p_i(x) <_{H'} q_j(y))$$

مثال: در زمانبندی‌های زیر، H_2 و H_3 معادل هستند اما H_4 و H_5 با هیچ‌کدام معادل نیستند.

H_2:
$$\begin{array}{lllll} r_1(x) & \rightarrow & w_1(y) & \rightarrow & w_1(z) \\ & & \downarrow & & \downarrow \\ & & w_2(y) & \rightarrow & r_2(z) \rightarrow c_2 \end{array}$$

H_3:
$$\begin{array}{llllll} r_1(x) & \rightarrow & w_1(y) & \rightarrow & w_1(z) & \\ & & & & \downarrow & \\ & & & & w_2(y) & \rightarrow r_2(z) \rightarrow c_2 \end{array}$$

H_4:
$$\begin{array}{llllll} r_1(x) & \rightarrow & w_1(y) & \rightarrow & w_1(z) & \rightarrow w_1(t) \rightarrow c_1 \\ & & & & \downarrow & \\ & & & & w_2(y) & \rightarrow r_2(z) \rightarrow c_2 \end{array}$$

H_5:
$$\begin{array}{lllll} r_1(x) & \rightarrow & w_1(y) & \rightarrow & w_1(z) \\ & & \downarrow & & \uparrow \\ & & w_2(y) & \rightarrow & r_2(z) \rightarrow c_2 \end{array}$$

توضیح:

- همه موارد فوق، خواص زمانبندی را دارند.
- H_4 شرط ۱ را ندارد و H_5 شرط ۲ را.

تعریف: تراکنش T_i در زمانبندی H قبل از T_j اجرا می‌شود اگر و تنها اگر تمام عملگرهای T_i قبل از اولین عملگر T_j اجرا شود.

$$T_i <_H T_j \Leftrightarrow \forall p_i(x) \in T_i, \forall q_j(y) \in T_j (p_i(x) <_H q_j(y))$$

زمانبندی پی‌درپی(serial shedule)

زمانبندی‌ها را نمی‌توانیم به صورت پی‌درپی تعریف کنیم به دلیل اینکه تراکنش فعال دارند و در زمانبندی‌های پی‌درپی فقط تراکنش آخر می‌تواند فعال باشد. بنابراین به صورت مقدماتی زمانبندی کامل را به صورت پی‌درپی تعریف می‌کنیم.

تعریف: زمانبندی کامل H را **پی‌درپی** گوییم اگر برای هر دو تراکنش T_i و T_j در H، T_i قبل از T_j اجرا شود یا بالعکس:

$$H \in SER \Leftrightarrow \forall T_i, T_j \in H (T_i <_H T_j \vee T_j <_H T_i)$$

پیش‌تعریف: زمانبندی H پی‌درپی‌پذیر در برخورد است اگر معادل در برخورد با یک زمانبندی پی‌درپی باشد.

با این تعریف، زمانبندی H در صورتی پی‌درپی‌پذیر است که کامل باشد، زیرا باید همه تراکنش‌های آن خاتمه یافته باشند. پس باید به راهکار جدیدی بیاندیشیم.

پی‌درپی‌پذیر در برخورد (confilict serializable schedule)

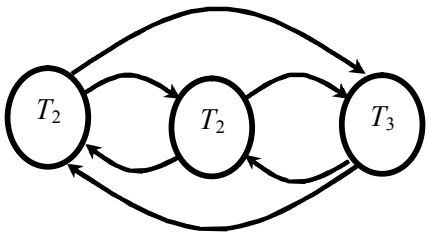
تعریف: زمانبندی H **پی‌درپی‌پذیر در برخورد (CSR)** است اگر **پرتو ثابت** آن معادل در برخورد با زمانبندی پی‌درپی H_5 باشد.

❖ تشخیص پی‌درپی پذیری در برخورد

تعریف: در **گراف پی‌درپی پذیری** H که آن را با SG(H) نمایش می‌دهیم، گره‌ها تراکنش‌ها هستند و لبه $T_i \rightarrow T_j$ ($i \neq j$) در صورتی وجود دارد که در H عملگری برخورددار از T_i قبل از عملگری برخورددار از T_j آمده باشد.

مثال: گراف پی‌درپی‌پذیری زمانبندی H_1 بالا به صورت زیر می‌باشد:

$H_{1:}$



قضیه ۱: زمانبندی H پی‌درپی‌پذیر در برخورد است اگر و فقط اگر SG(H) فاقد حلقه باشد.

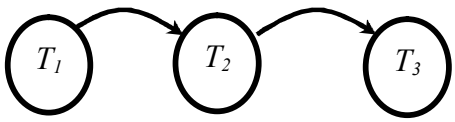
برای اثبات این قضیه به [Bernstein et al. 1987] مراجعه فرمایید.

□

قضیه ۲: در گراف پی‌درپی‌پذیری خاصیت انتقال وجود ندارد.

اثبات: مثال نقض

در گراف پی‌درپی‌پذیری زمانبندی $r_1(X), w_2(X), r_2(Y), w_3(Y)$ گره‌های T_1، T_2 و T_3 و لبه‌های $T_1 \rightarrow T_2$ و $T_2 \rightarrow T_3$ وجود دارند. نکته قابل توجه این است که $T_1 \rightarrow T_2$ با داده x به وجود آمده است ولی $T_2 \rightarrow T_3$ با داده y و از آنجا که این لبه‌ها روی دو داده مختلف ایجاد شده‌اند، لبه $T_1 \rightarrow T_3$ وجود ندارد.



□

پی‌درپی پذیری در دید

همان‌طور که قبلاً گفته شد ما دو هدف را دنبال می‌کنیم:

۱- بیان دقیق و ریاضی مفاهیم

۲- اثبات خواص آنها

تعریف: در زمانبندی H تراکنش T_i داده x را از تراکنش T_j می‌خواند (read-from)، اگر T_j آخرین عمل نوشتن روی x را انجام داده باشد و قبل از خواندن T_i ساقط نشده باشد:

$$T_j \xrightarrow[H]{x} T_i \Leftrightarrow w_j(x) <_H r_i(x) \wedge a_j \not<_H r_i(x) \wedge \left(\forall T_k : w_j(x) <_H w_k(x) <_H r_i(x) \Rightarrow a_k <_H r_i(x)\right)$$

تذکر: تراکنش می‌تواند داده‌ای را از خودش نیز بخواند.

تعریف: در زمانبندی H تراکنش T_i از تراکنش T_j می‌خواند اگر T_i داده‌ای را از T_j بخواند.

$$T_j \xrightarrow[H]{*} T_i \Leftrightarrow \exists x : T_j \xrightarrow[H]{x} T_i$$

تعریف: در زمانبندی H، تراکنش T_i **آخرین نوشتن (final write)** داده x را انجام می‌دهد اگر T_i ساقط نشده باشد و هر تراکنش T_j دیگری که روی آن داده می‌نویسد، یا قبل از T_i بنویسد یا ساقط شده باشد.

$$\widehat{w}_i^H(x) \Leftrightarrow a_i \notin H \wedge \forall w_j(x) \in H, i \neq j \Rightarrow (w_j(x) <_H \widehat{w}_i^H(x) \vee a_j \in H)$$

تعریف: دو زمانبندی H و H' که مجموعه تراکنش‌ها و مجموعه عملگرهای آنها یکسان باشد **معادل در دید** هستند اگر و تنها اگر:

۱- برای هر داده x، تراکنشی که ابتدا به ساکن داده x را می‌خواند، در هر دو زمانبندی یکسان باشد:

$$\forall x(\exists r_0(x) \in H : \nexists o_i(x) <_H r_0(x) \Rightarrow \exists r_0(x) \in H' \wedge \nexists o_j(x) <_{H'} r_0(x))$$

۲- برای هر دو تراکنش ساقط نشده T_i و T_j و برای هر داده x، اگر T_i در H داده x را از T_j می‌خواند، در H' نیز T_i داده x را از T_j بخواند و بالعکس:

$$\forall x, \forall T_i, T_j \in H : a_i, a_j \notin H, a_i, a_j \notin H'(T_j \xrightarrow[H]{x} T_i \Leftrightarrow T_j \xrightarrow[H']{x} T_i)$$

۳- برای هر داده x، تراکنش ساقط نشده‌ای که آخرین نوشتن در x را انجام می‌دهد، در هر دو زمانبندی یکسان باشد:

$$\forall x : \widehat{w}_i^H(x), \widehat{w}_j^{H'}(x), a_i \notin H, a_j \notin H' \Rightarrow i = j$$

تعریف: زمانبندی H **پی‌درپی پذیر در دید** یا **VSR (view serializable)** می‌باشد اگر معادل در دید با یک زمانبندی پی‌درپی باشد.

مثال: آیا زمانبندی زیر پی‌درپی‌پذیر در دید است؟

$$H_6 : r_1(A)r_2(A)w_1(C)w_1(B)r_3(B)r_2(C)c_1w_2(C)w_2(D)c_2w_3(C)c_3$$

حل:

با یک بررسی ساده دیده می‌شود که این زمانبندی، پی‌درپی‌پذیر در برخورد نیست (گراف آن را رسم کنید) اما پی‌درپی‌پذیر است و معادل پی‌درپی آن $T_1 < T_2 < T_3$ می‌باشد. یعنی:

$$r_1(A)w_1(C)w_1(B)c_1r_2(A)r_2(C)w_2(C)w_2(D)c_2r_3(B)w_3(C)c_3$$

زیرا :

۱- در هر دو، مجموعه تراکنش‌ها و مجموعه عملگرها یکسان می‌باشد و تراکنشی که اولین بار A را خوانده T_1 است.

۲- در هر دو، تراکنش T_2 داده C را از T_1 و تراکنش T_3 داده B را از T_1 خوانده است.

۳- در هر دو، آخرین نویسنده‌های B تراکنش T_1 است؛ C تراکنش T_3 و D تراکنش T_2 و داده‌ی A نوشته نشده است.

مثال: آیا زمانبندی زیر پی‌درپی‌پذیر در دید است؟

$H_7 = w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_3(y)\ w_1(y)\ c_1\ w_3(x)\ c_3$

حل: زمانبندی فوق پی‌درپی‌پذیر در دید نیست، زیرا برای داده x، تراکنشی که آخرین نوشتن را انجام می‌دهد T_3 است ولی برای Y، تراکنش T_1. بنابراین معادل در دید با هیچ یک از ترتیب‌های پی‌درپی T_1، T_2 و T_3 نمی‌باشد.

برای حل مثال به کتاب مراجعه فرمایید.

ترمیم

برای آنکه زمانبندی صحیح باشد، یعنی جامعیت بانک اطلاعات را حفظ کند، هم باید از نظر همروندی مشکلی نداشته باشد و هم از نظر ترمیم.

تعریف: زمانبندی H را **ترمیم‌پذیر (Recoverable)** می‌نامیم اگر هرگاه T_i از T_j ($i \neq j$) بخواند، تثبیت T_j قبل از تثبیت T_i باشد.

$$H \in RC \Leftrightarrow \forall T_i, T_j \in H, i \neq j (T_j \xrightarrow[H]{*} T_i \Rightarrow c_j <_H c_i)$$

اشکال عمده زمانبندی‌های ترمیم‌پذیر سقوط‌های آبشاری است.

تعریف: زمانبندی H را **فاقد سقوط‌های آبشاری** (Avoiding Cascading Aborts) می‌نامیم اگر هرگاه T_i از T_j ($i \neq j$) بخواند، آنگاه تثبیت T_j قبل از خواندن T_i باشد.

$$H \in ACA \Leftrightarrow \forall T_i, T_j \in H, i \neq j, \forall x (T_j \xrightarrow[H]{x} T_i \Rightarrow C_j <_H r_i(x))$$

در فصل گذشته دیدیم که زمانبندی‌های سقوط آبشاری و ترمیم‌پذیر فقط به مفهوم read from می‌پردازند و به عملگر write کاری ندارند و دیدیم که این عملگرهای write نیز ممکن است مشکل ایجاد کنند. بنابراین زمانبندی بهتری را که کامل نیز است معرفی می‌کنیم به نام زمانبندی **محض** یا **Strict** یا **سخت‌گیر**.

تعریف: زمانبندی H را **محض (سخت‌گیر)** می‌نامیم هرگاه هر عمل $o_i(x)$ که بعد از $w_j(x)$ انجام می‌شود، حتماً پس از خاتمه T_j باشد. یعنی به طور خلاصه هر کس از داده‌ای تأثیری می‌پذیرد، تکلیف تأثیرگذار باید قبلاً مشخص شده باشد.

$$H \in ST \Leftrightarrow \forall o_i(x) : (w_j(x) <_H o_i(x), i \neq j \Rightarrow c_j <_H o_i(x) \vee a_j <_H o_i(x))$$

مثال: تراکنش‌ها و زمانبندی‌های زیر را در نظر بگیرید:

T_1: $r_1(x)\ w_1(y)\ w_1(z)$
T_2: $r_2(u)\ w_2(y)\ r_2(z)$

H_8: $r_1(x)\ w_1(y)\ r_2(u)\ w_2(y)\ w_1(z)\ r_2(z)\ c_2\ c_1$
H_9: $r_1(x)\ w_1(y)\ r_2(u)\ w_2(y)\ w_1(z)\ r_2(z)\ c_1 c_2$
H_{10}: $r_1(x)\ w_1(y)\ r_2(u)\ w_2(y)\ w_1(z)\ c_1 r_2(z)\ c_2$
H_{11}: $r_1(x)\ w_1(y)\ r_2(u)\ w_1(z)\ c_1\ w_2(y)\ r_2(z)\ c_2$
H_{12}: $r_1(x)\ r_2(u)\ w_1(y)\ a_1\ w_2(y)\ r_2(z)\ c_2$

زمانبندی H_8 ترمیم پذیر نیست زیرا T_2 داده z را از T_1 می‌خواند اما $c_2 < c_1$. زمانبندی H_9 اگرچه ترمیم پذیر است اما فاقد سقوط‌های آبشاری نیست زیرا T_2، داده z را قبل از تثبیت T_1 از آن می‌خواند. زمانبندی H_{10} فاقد سقوط‌های آبشاری است اما سخت‌گیر نیست زیرا عمل نوشتن T_2 در y، بعد از نوشتن T_1 در y و قبل از اتمام T_1 انجام شده است. زمانبندهای H_{11} و H_{12} محض می‌باشند.

	RC	**ACA**	**ST**
H_8	×	×	×
H_9	√	×	×
H_{10}	√	√	×
H_{11}, H_{12}	√	√	√

قضیه: $\mathbf{ST \subset ACA \subset RC}$

در تعریف زیرمجموعه‌ی محض اگر $A \subset B$ باشد، اولاً A زیرمجموعه B است و ثانیاً این دو مساوی نیستند (در B چیزی است که در A نیست).

اثبات:

الف- $ST \subseteq ACA$

فرض کنید که در زمان‌بندی محض H، تراکنش T_i داده x را از تراکنش T_j ($i \neq j$) بخواند، پس داریم: $w_j(X) <_H r_i(X)$ و $c_j <_H r_i(X)$. از آنجا که $r_i(X) <_H c_i$ می‌باشد، در نتیجه $w_j(X) <_H c_i$، بنابراین H فاقد سقوط‌های آبشاری نیز هست.

اما با توجه به مثال فوق، زمان‌بندی مانند H_{10} وجود دارد که ACA هست ولی ST نیست، پس $ST \neq ACA$ و لذا $ST \subset ACA$.

ب- $ACA \subseteq RC$

فرض کنید H یک زمان‌بندی فاقد سقوط‌های آبشاری باشد و تراکنش T_i داده x را از تراکنش T_j ($i \neq j$) بخواند و T_i تثبیت شده باشد، پس داریم: $w_j(X) <_H c_j <_H r_i(X)$ و چون $c_i \in H$ و $r_i(X) <_H c_i$ لذا $c_j <_H c_i$. بنابراین H ترمیم‌پذیر نیز هست.

اما با توجه به مثال فوق، زمان‌بندی مانند H_9 وجود دارد که RC هست ولی ACA نیست، پس $ACA \neq RC$ و لذا $ACA \subset RC$.

تمرین: مستقیماً با استفاده از استدلال ریاضی ثابت کنید $ST \subset RC$.

قفل دومرحله‌ای یا 2PL (Two-Phase Locking)

نماد $ol_i(x)$ برای درخواست قفل، $ou_i(x)$ برای آزاد نمودن قفل عملگر o و $u_i(x)$ برای باز کردن قفل هر عملگری از تراکنش i روی دادهٔ x به کار می‌رود. ما به $l_i(x)$ یعنی قفل کردن نیازی نداریم بلکه درخواست قفل را نیاز داریم و درخواست قفل معمولی نداریم و فقط به قفل‌های S و X خواهیم پرداخت.

تعریف: دو قفل با هم **ناسازگار** هستند (برخورد دارند) اگر روی داده‌های یکسان، مربوط به تراکنش‌های متمایز باشند و حداقل یکی از آنها قفل نوشتن (write) باشد. در غیر این صورت با هم **سازگار** هستند.

$$pl_i(x) \in \mathrm{H} \not\approx ql_j(y) \in \mathrm{H} \Leftrightarrow i \neq j \wedge x = y \wedge (p = w \vee q = w))$$

$$pl_i(x) \in \mathrm{H} \approx ql_j(y) \in \mathrm{H} \Leftrightarrow \neg\ (pl_i(x) \in \mathrm{H} \not\approx ql_j(y) \in \mathrm{H})$$

تعریف: اصول کلی قفل‌گذاری

۱. هر عملگر بعد از قفل کردن و قبل از آزاد کردن قفل دادهٔ مربوطه اجرا می‌شود.

$$o_i(x) \in H \Rightarrow ol_i(x) <_H o_i(x) <_H ou_i(x)$$

۲. بازکردن قفل هر داده‌ی مربوط به عملگرهای هر تراکنش حداکثر یکبار انجام می‌شود (ارتقاء و کاهش قفل).

$$u_i(x) \in H \Rightarrow \exists u_j(x) \in H : i = j$$

۳. در صورتی داده‌ی قفل‌شده‌ی x توسط تراکنش همروند دیگری قفل می‌شود (یعنی عمل مربوطه انجام می‌شود) که این قفل‌ها با هم سازگار باشند.

$$p_i(x), q_j(x) \in H \Rightarrow pu_i(x) <_H ql_j(x) \ \vee \ pl_i(x) \approx ql_j(x)$$

پروتکل قفل دو مرحله‌ای پایه (Basic Two-Phase Locking Protocol)

تعریف: یک زمان‌بندی از قفل دو مرحله‌ای پایه پیروی می‌کند اگر و تنها اگر قفلی از هر تراکنشی آزاد شد، از آن پس آن تراکنش نتواند هیچ قفل دیگری (روی هیچ داده‌ای) بگیرد.

$$H \in B2PL \Leftrightarrow \forall T_i \in H, \forall x(u_i(x) \in H \Rightarrow \nexists y \ (u_i(x) <_H ol_i(y)))$$

لم ۱- اگر $T_i \to T_j$ لبه‌ای در گراف پی‌درپی‌پذیری زمان‌بندی قفل دومرحله‌ای پایه H یا SG(H) باشد، آنگاه لبه‌ای به صورت $T_j \to T_i$ در SG(H) وجود ندارد.

اثبات: اگر لبه‌ای مثل $T_i \to T_j$ برای داده‌ی P در SG(H) وجود داشته باشد، آنگاه داریم:

$$(I) \qquad r_i(P) <_H w_j(P) \quad \vee \quad w_i(P) <_H w_j(P)$$

در این صورت، مطمئناً پس از آن برای هیچ داده‌ای مانند Q شرط زیر وجود نخواهد **داشت**:

$$(II) \qquad r_j(Q) <_H w_i(Q) \quad \vee \quad w_j(Q) <_H w_i(Q)$$

زیرا در رابطه (I) تراکنش T_j در صورتی می‌تواند داده P را قفل نماید که T_i قفل روی P را **آزاد** کرده باشد که در این صورت پس از آن، T_i نمی‌تواند قفل دیگری را (از جمله روی Q) اخذ نماید.

نتیجه: پس اگر $T_i \to T_j$ لبه‌ای باشد، در این صورت $T_j \to T_i$ که بتواند تشکیل حلقه‌ی مستقیم دهد وجود نخواهد داشت.

□

لم۲- اگر $T_1 \to T_2 \to \ldots \to T_n$ مسیری با طول بزرگتر از یک در SG(H) باشد، آنگاه برای هیچ داده‌ای لبهٔ $T_n \to T_1$ در $SG(H)$ وجود ندارد.

اثبات: تعمیم‌یافته لم ۱ با روش استقراء:

۱- برای n=۲ با توجه به **لم ۱** درست است.

۲- فرض می‌کنیم برای $n = k$ درست باشد.

۳- باید ثابت کنیم برای $n = k + 1$ نیز درست است. اگر چنین نباشد، در این صورت T_{k+1} منتظر T_1 و چون T_1 منتظر T_2 می‌باشد پس خواهیم داشت $T_2 \to \ldots \to T_{k+1}$. با تغییر متغیر k+1 به k نتیجه می‌شود $T_1 \to T_2 \to \ldots \to T_k$ که خلاف فرض ۲ است.

□

لـم ۳- اگـر $T_i \to T_j$ لبـه‌ای در SG(H) باشـد، آنگـاه همـواره آزاد کـردن قفـل **هــر داده‌ی** Q در T_i، قبـل از گـرفتن هـر قفـل ناسـازگار روی Q توسـط T_j ($i \neq j$) انجـام مـی‌گیـرد، یعنی:

$$\forall (T_i, T_j) \in SG(H) : i \neq j \wedge H \in B2PL, \forall Q \ (pl_i(Q) \asymp ql_j(Q) \Rightarrow pu_i(Q) <_H ql_j(Q))$$

اثبات: مستقیماً از قاعده دومرحله‌ای استنتاج می‌شود.

□

لـــم۴- اگـــر مـــسیری ماننـــد $T_1 \to T_2 \to \ldots \to T_n$ بـــا طـــول بزرگتـــر از یـــک در گـــراف پی‌درپی‌پذیری زمـان‌بنـدی قفـل دومرحلـه‌ای پایـه H وجـود داشـته باشـد، آنگـاه همـواره آزاد کـردن قفـل هـر داده Q در T_1، قبـل از گـرفتن هـر قفـل دارای برخـورد (ناسـازگار) روی Q توسط T_n انجام می‌گیرد.

اثبات: تعمیم‌یافته لم ۳ با روش استقراء

قضیه ۴: زمان‌بندی قفل دومرحله‌ای پایه دارای خاصیت CSR می‌باشد؛ یعنی پی‌درپی‌پذیری در برخورد را تضمین می‌کند.

اثبات: براساس لم ۲ و لم ۴، در گراف پی‌درپی‌پذیری یک زمان‌بندی قفل دومرحله‌ای حلقه‌ای وجود ندارد. لذا طبق قضیه ۱ زمان‌بندی قفل دومرحله‌ای پایه، پی‌درپی‌پذیر در برخورد می‌باشد.

□

قضیه ۵: پروتکل قفل دومرحله‌ای پایه تضمین می‌کند که مشکل بازیابی ناهمگام پیش نیاید.

اثبات: مشکل بازیابی ناهمگام زمانی پیش می‌آید که داده‌های مورد دستیابی بعضاً به هم وابستگی داشته باشند. از آنجا که این وابستگی‌ها قابل شناسایی نیستند، مجبوریم فرض کنیم که همه داده‌های مورد دستیابی به هم وابسته‌اند. نگهداری قفل همه داده‌ها و جداکردن فاز گرفتن قفل از آزاد کردن قفل، این فرض را برآورده می‌سازد [Bernstein et al. 1987].

□

نتیجه: پروتکل قفل دومرحله‌ای پایه، هر سه مشکل همروندی را حل می‌کند.

پروتکل قفل دو مرحله‌ای محافظه‌کار (Conservative Two-Phase Locking)

در قفل دو مرحله‌ای پایه، تراکنش‌ها به دلیل وقوع بن بست ساقط می‌شوند. پروتکل قفل دو مرحله‌ای محافظه‌کار تضمین می‌کند که بن بست رخ نخواهد داد.

تعریف: یک زمانبندی از نوع قفل دو مرحله‌ای محافظه‌کار است اگر و تنها اگر از نوع قفل دو مرحله‌ای پایه باشد و هر تراکنش آن قبل از شروع به اجرا، تمام قفل‌های مورد نیازش را اخذ نموده باشد:

$$H \in C2PL \Leftrightarrow (H \in B2PL) \wedge \forall T_i \in H(\nexists p,q : p_i(x) <_H ql_i(y))$$

قضیه ۶: در گراف انتظار یک زمانبندی C2PL حلقه وجود ندارد و بن‌بست رخ نخواهد داد.

اثبات: در این زمانبندی، اگر تراکنشی مثل T_i منتظر قفلی باشد که در اختیار T_j قرار دارد، پس T_i قبل از گرفتن قفل، شروع به اجرا نموده است و منتظر مانده که خلاف فرض است، بنابراین هیچ تراکنشی منتظر آزاد کردن قفل نمی‌ماند.

مجموعه داده‌هایی که تراکنش برای خواندن (نوشتن) نیاز دارد را read-set (write-set) می‌نامیم. نقطه ضعف C2PL، علاوه بر کاهش سطح همروندی، نیاز به داشتن read-set و write-set می‌باشد.

پروتکل قفل دو مرحله‌ای محض (Strict Two-Phase Locking)

پروتکل قفل دو مرحله‌ای پایه مشکل سقوط‌های آبشاری دارد که در پروتکل قفل دومرحله‌ای محض حل می‌شود.

تعریف: در پروتکل قفل دو مرحله‌ای محض که از قفل دومرحله‌ای پایه پیروی می‌کند، قفل‌های w تا پایان تراکنش مربوطه (پس از اجرای c_i یا a_i) آزاد نمی‌شوند اما قفل‌های r پس از اجرای آخرین عملگر تراکنش و قبل از c_i یا a_i آزاد می‌گردند.

$$H \in S2PL \Leftrightarrow (H \in B2PL) \wedge \forall T_i \in H, \forall x$$
$$(((c_i <_H wu_i(x) \vee a_i <_H wu_i(x)) \wedge (\nexists o_i(x)(ru_i(x) <_H o_i(x) <_H c_i \vee ru_i(x) <_H o_i(x) <_H a_i)))$$

قضیه ۷: قفل دومرحله‌ای محض، اجرای محض زمانبندی را تضمین می‌کند.

اثبات:

اگر H یک زمانبندی قفل دو مرحله‌ای محض باشد و $w_i(X) <_H o_j(X)$ آنگاه داریم:

$$wl_i(X) <_H w_i(X) <_H wu_i(X) \text{ و } ol_j(X) <_H o_j(X) <_H ou_j(X)$$

با توجه به اینکه $wl_i(X)$ و $ol_j(X)$ با هم برخورد دارند، باید داشته باشیم: $wu_i(X) <_H ol_j(X)$ یا $ou_j(X) <_H wl_i(X)$ که با فرض $w_i(X) <_H o_j(X)$ در تناقض است و لذا $wu_i(X) <_H ol_j(X)$.

اما از آنجا که H از نوع S2PL است باید یکی از دو شرط زیر برقرار باشد:

$a_i <_H wu_i(X)$ یا $c_i <_H wu_i(X)$

از مجموعه قواعد حاصله برمی‌آید که یا $a_i <_H o_j(X)$ و یا $c_i <_H o_j(X)$، یعنی H یک اجرای محض است.

این خاصیت برای ما بسیار مهم است زیرا به طور ضمنی، ترمیم‌پذیر بودن و فاقد سقوط‌های آبشاری بودن را نیز تضمین می‌نماید (زیرا $ST \subset ACA \subset RC$).

ویژگی فوق را شاید بتوان مظهری از اوج نبوغ یک متخصص بانک اطلاعات دانست که ضمن حل مسئله پی‌درپی‌پذیری، مشکل ترمیم‌پذیری را نیز حل کرده و به قول معروف با یک تیر، دو نشان زده است.

همچنین در قفل دومرحله‌ای محض، به ویژه به دلیل آزاد کردن قفل‌ها در خاتمه تراکنش، نیاز به ارسال پیام آزاد کردن قفل از بین رفته و حجم مراودات کاهش چشمگیری می‌یابد.