درس و کنکور پایگاه داده پیشرفته

تألیف:

دکتر رمضان عباس نژادورزی
دکتر مرتضی بابازاده
دکتر میر سعید حسینی

فن‌آوری نوین

سرشناسه	:	عباس نژادورزی، رمضان، ۱۳۴۸ -
عنوان و نام پدیدآور		درس و کنکور پایگاه داده پیشرفته/تألیف رمضان عباس‌نژادورزی، مرتضی بابازاده، میرسعید حسینی.
مشخصات نشر	:	مازندران:فن‌آوری نوین, ۱۳۹۶
مشخصات ظاهری	:	۲۵۶ص
شابک	:	۲۷۵۰۰۰ریال :۳-۰۵-۷۲۷۲-۶۰۰-۹۷۸
وضعیت فهرست نویسی	:	فیپا
موضوع		دانشگاه‌ها و مدارس عالی -- ایران -- آزمون‌ها
موضوع		Universities and colleges --Iran -- Examinations
موضوع		پایگاه‌های اطلاعاتی -- راهنمای آموزشی(عالی)
موضوع		Databases-- Study and teaching (Higher)
موضوع		پایگاه‌های اطلاعاتی-- آزمون‌ها و تمرین‌ها (عالی)
موضوع		Databases -- Examinations, questions, etc. (Higher)
موضوع		پایگاه‌های اطلاعاتی -- مسائل، تمرین‌ها و غیره (عالی)
موضوع		Problems, exercises, etc. (Higher -- Databases)
موضوع		آزمون دوره‌های تحصیلات تکمیلی -- ایران
موضوع		Iran -- Graduate Record Examination
شناسه افزوده		بابازاده، مرتضی، ۱۳۶۰ -
شناسه افزوده		حسینی شیروانی، میرسعید، ۱۳۵۷ -
رده بندی کنگره		LB۲۳۵۳
رده بندی دیویی		۳۷۸/۱۶۶۴
شماره کتابشناسی ملی		۴۸۷۳۱۴۴

www.fanavarienovin.net

فن‌آوری نوین **بابل، کدپستی ۴۷۱۶۷-۷۳۴۴۸** **تلفن: ۳۲۲۵۶۶۸۷-۰۱۱**

درس و کنکور پایگاه داده پیشرفته

تألیف: رمضان عباس نژاد ورزی- مرتضی بابا زاده- میر سعید حسینی

نوبت چاپ: چاپ اول

سال چاپ: پاییز ۱۳۹۶

شمارگان: ۲۰۰ جلد

قیمت: ۲۷۵۰۰ تومان

نام چاپخانه و صحافی:

شابک: ۳-۰۵-۷۲۷۲-۶۰۰-۹۷۸

نشانی ناشر: بابل، چهارراه نواب، کاظم‌بیگی، جنب مسجد منصور کاظم‌بیگی، طبقه همکف

طراح جلد: کانون آگهی و تبلیغات آبان (احمد فرجی)

تهران، خ اردیبهشت، نبش وحید نظری، پلاک ۱۴۲ تلفکس: ۶۶۴۰۰۱۴۴-۶۶۴۰۰۲۲۰

فهرست مطالب

فصل سوم: قفل گذاری دو مرحله‌ای

فصل چهارم: زمان بندی غیر قفلی

فصل پنجم: خرابی ها

مقدمه

امروزه با رشد اینترنت، استفاده از داده‌ها به خصوص داده‌های عظیم روز به روز در حال افزایش می‌باشد. به همین دلیل، پایگاه داده پیشرفته به عنوان یکی از دروس در دوره کارشناسی ارشـد در وزارت علـوم و دانشـگاه آزاد اسلامی تدریس می‌شود.

به طوری که در آزمون دکتری نرم‌افزار نیز از آن سوال می‌آید. بر همین اساس، کتاب حاضر تهیه گردید و در اختیار علاقه‌مندان قرار گرفته است. این کتاب شامل مباحث زیر می‌باشد:

۱. مفهوم تراکنش و پیاده‌سازی آن، معماری پایگاه داده، انواع زمان‌بندی با قفل گذاری و بدون قفل گذاری، ترمیم و امنیت در پایگاه داده.

۲. بیان حدود ۲۵۰ تست فراگیر و سوالات ترمی پیام نور با پاسخ تشریحی آن‌ها

۳. بیان حدود ۱۰۰ مسئله پایگاه داده پیشرفته و پاسخ تشریحی آن‌ها.

امیدوارم این اثر مورد توجه جامعه انفورماتیک کشور، اساتید و دانشجویان عزیز قرار گیرد.

در پایان از تمامی اساتید و دانشجویان عزیز تقاضا داریم، هر گونه اشکال، ابهام در متن کتاب، پیشنهاد و انتقادات را به آدرس پست‌الکترونیک **fanavarienovin@gmail.com** ارسال نمایند.

مولفین

fanavarienovin@gmail.com

فصل ۱ مقدمه

کنترل هم‌روندی فعالیتی است که رفتار پردازش‌هایی را که به‌طور موازی عمل می‌کنند، هماهنگ می‌سازد، دسترسی داده را به اشتراک می‌گذارد و بنابراین به‌طور بالقوه تراکنش‌ها را با یکدیگر قاطی می‌کند. **ترمیم**[1]، فعالیتی است که تضمین می‌کند خرابی‌ها سخت‌افزار و نرم‌افزار باعث تخریب ماندگاری داده نمی‌شود. کنترل هم‌روندی و مشکلات ترمیم در طراحی سخت‌افزار، سیستم‌های عامل، سیستم‌های بلادرنگ، سیستم‌های ارتباطی، سیستم‌های پایگاه و غیره به وجود می‌آیند. در این کتاب درباره کنترل هم‌روندی و مشکلات ترمیم در سیستم‌های پایگاه داده بحث می‌کنیم.

با استفاده از یک مدل سیستم‌های پایگاه داده را بررسی می‌کنیم. این مدل انتزاعی از انواع سیستم‌های اداره‌کننده داده از قبیل سیستم‌های مدیریت پایگاه داده برای برنامه‌های کاربردی پردازش داده، سیستم‌های پردازش تراکنش برای رزرو خطوط هوایی یا بانکداری و سیستم‌های فایل برای محیط محاسبه عمومی است. کنترل هم‌روندی و ترمیم در هر سیستم که با مدل مطابقت دارد، اعمال می‌شود. قطعه اصلی این مدل **تراکنش**[2] است. به‌طور غیررسمی یک تراکنش اجرایی از یک برنامه است که به پایگاه داده به اشتراک گذاشته‌شده دسترسی دارد. هدف از کنترل هم‌روندی و ترمیم این است که تضمین کند، تراکنش‌ها به‌طور **اتمیک**[3] اجرا می‌شوند. یعنی:

۱. هر تراکنش به داده به اشتراک گذاشته‌شده، بدون تداخل با بقیه تراکنش‌ها دسترسی دارد.

۲. اگر یک تراکنش به‌طورمعمول پایان پذیرد، پس از اتمام آن، اثرات آن به‌صورت **پایدار (دائمی)**[4] می‌گردد.

هدف این فصل این است که به این مدل به‌طور جامع و دقیق بپردازیم. در این بخش، **مدل مبتنی برکاربر**[5] از سیستم را نشان می‌دهیم. این مدل، پایگاه داده‌ی را شامل می‌شود که کاربر بتواند به‌وسیله اجرای تراکنش‌ها به آن دسترسی داشته باشد و باوجود خرابی‌ها، تراکنش در آن به‌صورت اتمیک اجرا می‌شود. در ادامه این فصل، یک مدل از سیستم‌های کنترل هم‌روندی پایگاه داده و ترمیم ارائه‌شده است که هدف آن درک یکپارچگی تراکنش است.

۱-۱. سیستم‌های پایگاه داده

یک پایگاه داده شامل مجموعه‌ی از اقلام داده نام‌گذاری شده است. هر قلم داده دارای یک مقدار است. مقادیر اقلام داده‌ای در هر زمان شامل وضعیت پایگاه داده است. در عمل یک‌قلم داده باید یک بخش حافظه اصلی مانند یک صفحه دیسک، یک رکورد از یک فایل، یا یک فیلد از رکورد باشد. اندازه داده موجود در قلم داده، **دانه‌بندی**[6] قلم داده نامیده می‌شود.

---

[1]. Recovery [2]. Transaction [3]. Atomically [4]. Persistent [5]. user-oriented model
[6]. granularity

دانه‌بندی ممکن است برای بررسی‌هایمان مهم نباشد و بنابراین آن را به‌طور نامشخص باقی می‌گذاریم. وقتی‌که دانه‌بندی را به‌طور نامشخص باقی می‌گذاریم، اقلام داده‌ی را با حروف کوچک و معمولاً x ، y و z نشان می‌دهیم.

یک سیستم پایگاه داده (DBS[1])، کلکسیونی از سخت‌افزارها و ماژول‌های نرم‌افزاری است که مجموعه‌ای از دستورات دسترسی به پایگاه داده که **اعمال پایگاه داده** [2] (به‌طور ساده اعمال) نامیده می‌شود را پشتیبانی می‌کند. مهم‌ترین عملیاتی که پایگاه داده انجام می‌دهد Read (خواندن) و Write (نوشتن) هستند. این اعمال به‌صورت زیر به کار می‌روند:

۱. **Read(x)**، مقدار ذخیره‌شده در قلم داده‌ی x را برمی‌گرداند.

۲. **Write(x, Val)**، مقدار x را به Val تغییر می‌دهد.

همچنین پایگاه داده عملیات دیگری را انجام می‌دهد که در ادامه می‌بینیم.

سیستم پایگاه داده (DBS) هر عمل را به‌طور **اتمیک**[3] (یکپارچه) اجرا می‌کند. یعنی، سیستم پایگاه داده اعمال را به‌طور **پی‌درپی**[4] (**به ترتیب**) اجرا می‌کند. پس، هر عمل را در یک‌زمان انجام می‌دهد. برای به دست آوردن این رفتار سیستم پایگاه داده ممکن است واقعاً اعمال را به‌صورت پی‌درپی اجرا کند. هرچند، حتی اگر عملیات در سیستم پایگاه داده به‌صورت **هم‌روند**[5] اجرا شوند، اثر نهایی آن‌ها باید معادل اجرای پی‌درپی آن اعمال باشد. به‌عنوان‌مثال، فرض کنید اقلام داده‌ی x و y در دو دیسک مختلف ذخیره‌شده‌اند. سیستم پایگاه داده ممکن است اعمال روی x و y را به ترتیب زیر اجرا کند:

۱. اجرای Read(x) را انجام دهد.

۲. پس از اجرای Read(x) (خاتمه گام ۱)، به‌صورت هم‌روند یا موازی اجرای Write(x, 1) و Read(y) را انجام دهد.

۳. پس از اجرای گام ۲، اجرای Write(y, 0) را انجام دهد.

هرچند Write(x, 1) و Read(y) به‌طور هم‌روند اجرا می‌شوند، می‌توان فرض کرد که به‌طور اتمیک اجراشده‌اند. چون این اجرا همان اثری را دارد که به‌طور پی‌درپی به‌صورت دستورات زیر اجرا شوند:

Read(x), Write(x, 1), Read(y), Write(y, 0)

همچنین سیستم پایگاه داده **اعمال دیگر تراکنش**[6] از قبیل **Start**، **Commit** و **Abort** را پشتیبانی می‌کند. **عمل Start**، شروع اجرای تراکنش جدیدی را به سیستم پایگاه داده اعلان می‌کند، **عمل Commit** یا **Abort** یک تراکنش را خاتمه می‌دهد. عمل **Commit**، خاتمه نرمال (تثبیت) تراکنش را به سیستم پایگاه داده اعلان می‌نماید و تمام اثرات تراکنش را دائمی می‌کند. **عمل Abort**، بیان می‌کند که تراکنش با خطا مواجه شده است و تمام اثرات آن باید نادیده گرفته شود.

به ازای هر تراکنش خاص، باید مشخص گردد که کدام برنامه دستورات پایگاه داده مربوطه را صادر نموده است. برای این منظور، در این مدل، فرض می‌شود که سیستم پایگاه داده به ازای هر Start یک **شناسه‌ی**

---

[1].Database System(DBS) [2]. Database Operations [3].Atomicily [4]. Sequentially
[5]. Concurrently [6]. Transaction Operations

منحصربه‌فرد (یکتا) [1] به تراکنش تخصیص می‌دهد. برنامه این شناسه را به هر یک از اعمال پایگاه داده خودش ضمیمه می‌کند تا زمانی که تراکنش Commit یا Abort شود. در نتیجه، از نظر سیستم پایگاه داده یک تراکنش با عمل Start شروع می‌شود، با تعدادی از اعمال دیگر پایگاه داده (احتمالاً هم‌روند) ادامه می‌یابد و در نهایت با یک دستور Commit یا Abort خاتمه می‌یابد. یک تراکنش ممکن است اجرای هم‌روند دو یا چند برنامه باشد. یعنی، در تراکنش ممکن است که قبل از این‌که نتیجه عمل اول مشخص باشد، عمل دوم (بعدی) ارائه گردد. هرچند آخرین عمل تراکنش‌ها باید Commit یا Abort باشد. بنابراین، یک سیستم پایگاه داده باید از پردازش مجدد داده‌های یک تراکنش که پس از اعمال Commit یا Abort رسیده‌اند، اجتناب نماید.

۲ - ۱. تعریف و نحو تراکنش

کاربران معمولاً از طریق اجرای برنامه‌ها با سیستم پایگاه داده ارتباط برقرار می‌کنند. از دیدگاه کاربر، یک تراکنش اجرای یک یا چند برنامه است که شامل اعمال پایگاه داده‌ی است. در پایگاه داده، تراکنش حاوی مجموعه‌ای از یک یا چند پرس‌وجو است که به‌عنوان یک واحد عمل می‌کنند. نتیجه اجرای هر پرس‌وجو در این واحد به **تثبیت**[2] یا **سقط**[3] دستورات دیگر بستگی دارد. یعنی، این واحد به‌عنوان یک کل، قابل‌تفکیک نیست. اگر حداقل یکی از پرس‌وجوها در این واحد با موفقیت اجرا نشود، کل آن واحد **خنثی**[4] خواهد شد و داده‌هایی که در اجرای آن واحد تغییر کرده‌اند، به مقدار قبل از اجرای تراکنش (آخرین مقدار تثبیت‌شده)[5] برمی‌گردند. هنگامی تراکنشی با موفقیت تثبیت خواهد شد که همه دستورات آن تثبیت گردند.

دنیای واقعی پر از تراکنش‌هایی مانند تراکنش‌ها در بورس، خریدهای اینترنتی از طریق کارت، کنترل موجودی حساب‌های بانکی و غیره است. در همه این موارد، موفقیت اجرای تراکنش به تعدادی دستورات بستگی دارد که به هم وابسته هستند. شکست هر یک از آن دستورات، تراکنش را سقط کرده و سیستم را به حالت قبل از اجرای تراکنش برمی‌گرداند.

برای درک بهتر مفهوم تراکنش، مثال انتقال پول از حساب A به‌حساب B را در نظر بگیرید (شکل ۱-۱). در این شکل می‌خواهیم ۱۰۰۰۰۰۰ ریال از حساب A به‌حساب B منتقل شود. در این سیستم زمانی انتقال کامل می‌گردد که حساب A، ۱۰۰۰۰۰۰ ریال بدهکار و حساب B، ۱۰۰۰۰۰۰ ریال بستانکار شود. اگر هر یک از این مراحل با شکست مواجه شود (انجام نگردد)، انتقال پول انجام نمی‌شود و هیچ مبلغی نباید از حساب A برداشت گردد. بنابراین، انتقال پول از حساب A به‌حساب B را می‌توان یک تراکنش فرض کرد. یعنی، یک واحد کار مستقل که شامل دو دستور است. این دو دستور عبارت‌اند از:

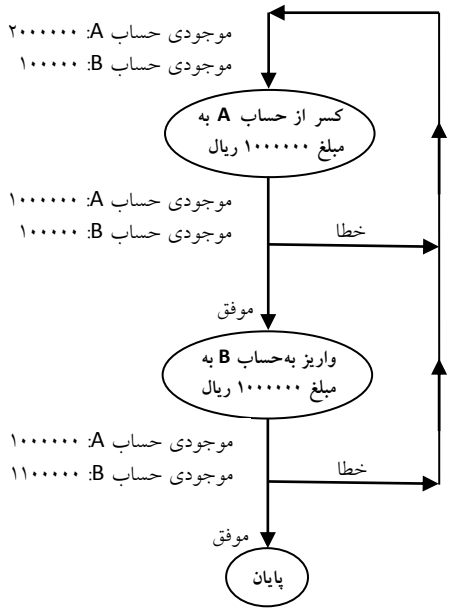
۱. برداشت ۱۰۰۰۰۰۰ ریال از حساب A

۲. واریز ۱۰۰۰۰۰۰ ریال به‌حساب B

در صورتی این تراکنش با موفقیت اجرا خواهد شد که هر دو کار(برداشت از حساب A و واریز به‌حساب B) با موفقیت تثبیت گردند. اگر یکی از این کارها با شکست مواجه شود، تراکنش سقط می‌گردد و پایگاه داده به وضعیت پایدار قبل از شروع این تراکنش برمی‌گردد.

---

[1]. Unique Transaction identifier [2]. Commit [3]. Abort [4]. Undo [5]. Last Committed Value



شکل ۱ - ۱ تراکنش انتقال ۱۰۰۰۰۰۰ ریال از حساب A به‌حساب B.

یک پایگاه داده بانکداری را در نظر بگیرید که فایل Accounts شماره‌حساب مشتریان را نگه‌داری می‌کند. در این فایل برای هر مشتری مانده‌حسابش نگه‌داری می‌شود و شماره‌حساب کلید آن در فایل Accounts است. یک تراکنش نمونه برای انتقال پول از یک حساب به‌حساب دیگر رویه‌ای به‌صورت زیر است (پیاده‌سازی تراکنش شکل ۱-۱):

```
Procedure Transfer
begin
  Start;
  input(fromAccount, toAccount, amount);
  (* This procedure transfers "amount" from "fromAccount" into "toAccount." "*)
  temp : = Read(Accounts[fromAccount]);
  if temp < amount then
   begin
    output( "موجودی کافی نیست");
    Abort
 end
 else
 begin
   Write(Accounts[fromAccount], temp - amount);
    temp : = Read(Accounts[toAccount]);
   Write(Accounts[toAccount], temp + amount);
   Commit;
   output( "عمل انتقال کامل شد");
end;
return
End
```

نام این رویه Transfer است. این رویه شامل اعلان زیر روال (کلمه کلیدی procedure به‌علاوه نام زیر روال، Begin و End)، دستور انتساب (عبارت =: متغیر)، یک دستور شرطی (دستور **else** دستور **then** عبارت

منطقی **if**، input (لیستی از مقادیر را از پایانه یا دیگر دستگاه‌های ورودی خوانده در متغیرها قرار می‌دهد)، output (لیستی از مقادیر ثابت یا مقدار متغیرها روی یک پایانه یا دستگاه‌های خروجی دیگر نمایش می‌دهد)، begin-end براکت‌هایی برای شروع و پایان لیست دستورات مرتبط به هم (**end** لیست دستورات **begin**)، یک دستور return برای برگشت از رویه، و براکتی برای نشان دادن یک توضیح (*comment*) است. در این رویه نماد ; جداکننده دستورات است.

همان‌طور که مشاهده کردید، جهت نمایش تراکنش Transfer از زبان خاصی استفاده نشده است، بلکه ارائه شبه کد صرفاً جهت نمایش منطق تراکنش است. انتخاب زبان برای بیان تراکنش‌ها جهت مطالعه کنترل هم روندی و ترمیم برای ما مهم نیست. در عمل، زبان می‌تواند یک زبان پرس‌وجو پایگاه داده، یک‌زبان نوشتن گزارش یا یک زبان برنامه‌نویسی سطح بالا با عملیات افزوده شده پایگاه داده باشد.

مهم نیست چگونه تراکنش‌ها بیان می‌شوند. ولی، مهم است به برنامه‌هایی ترجمه گردند که اعمال پایگاه داده را اجرا می‌کنند، چون این اعمال تنها راه دسترسی به پایگاه داده هستند. بنابراین، فرض می‌کنیم که برنامه‌هایی که شامل تراکنش‌ها هستند به زبان سطح بالایی که اعمال پایگاه داده‌ی در آن تعبیه شده، نوشته می‌شوند.

رویه Transfer یک برنامه واقعی نیست. چون در آن هیچ‌گونه کنترل خطا انجام نمی‌شود. کنترل خطا در برنامه‌های کاربردی **قابل‌اعتماد**، ضروری است. لذا، در فهم و درک مسائل ترمیم و کنترل هم‌روندی مهم نیست. بنابراین، از کنترل خطا در چنین برنامه‌هایی چشم‌پوشی می‌کنیم.

۱ – ۲ – ۱. خواص تراکنش

هر تراکنش چهار خاصیت دارد که مفهوم این خواص در زیر آمده‌اند:

۱. یکپارچگی

خاصیت **یکپارچگی یا اتمیک**[1] همه‌یاهیچ نام دارد. یعنی، اگر بخواهید تراکنشی اجرا شود یا باید همه دستورات آن اجرا شوند یا در صورت اجرا نشدن یکی، هیچ‌کدام از دستورات اجرا نشوند. فرض کنید، می‌خواهید پولی را از یک حساب کارت اعتباری به‌حساب کارت اعتباری دیگر انتقال دهید. این تراکنش از دو بخش تشکیل‌شده است. **در بخش اول**، مبلغی باید از حساب کارت اعتباری اول برداشت شود و **در بخش دوم**، همان مبلغ باید به‌حساب کارت اعتباری دوم واریز گردد. فرض کنید، بخش اول تراکنش انجام شود (مبلغ موردنظر از حساب کارت اعتباری اول برداشت شود) و ارتباط عابر بانک قطع شود. در این صورت، بخش دوم تراکنش قابل انجام نیست. اکنون این سؤال مطرح می‌شود که مبلغ برداشت‌شده از حساب کارت اعتباری اول چه می‌شود؟ بدیهی است که این مبلغ باید به‌حساب کارت اعتباری اول برگردانده شود تا صحت و جامعیت داده‌های پایگاه داده حفظ گردد. این عمل معادل این است که بگوییم، هیچ دستوری از تراکنش انجام نگردید.

---

[1]. Atomicity

یا فرض کنید می‌خواهید از یک کشور به کشور دیگری مسافرت کنید. برای این منظور، باید دو عمل زیر را انجام دهید:

۱. بلیط هواپیما را رزرو نمایید. ۲. هتل را رزرو کنید.

فرض کنید، بلیط هواپیما را رزرو کردید، ولی هتلتان رزرو نشد. در این صورت، باید بلیط هواپیما را کنسل کنید که می‌گوییم هیچ کاری از تراکنش انجام‌نشده است.

۲. خاصیت هم‌خوانی یا سازگاری

خاصیت **هم‌خوانی** یا **صحت** [1] بیان می‌کند که هر تراکنش باید تمامی قوانین جامعیت پایگاه داده را رعایت نماید. در مثال برداشت مبلغ از یک کارت اعتباری و واریز آن به کارت اعتباری دیگر، مبلغی که از کارت اعتباری اول برداشت می‌شود، باید دقیقاً برابر مبلغی باشد که به کارت اعتباری دوم واریز می‌گردد. خاصیت هم‌خوانی، جامعیت پایگاه داده را تضمین می‌کند. گاهی اتفاق می‌افتد کاربران داده‌های غلط را وارد می‌نمایند یا برنامه‌های غلط را اجرا می‌کنند و نتیجه نادرست می‌شود. جلوی چنین اشتباهاتی را نمی‌شود گرفت. پس، نتیجه می‌گیریم که خاصیت هم‌خوانی به‌طور کامل قابل‌کنترل نیست. خاصیت هم‌خوانی به این صورت بیان می‌گردد: «**هر تراکنشی اگر به‌تنهایی اجرا شود، پایگاه داده را از یک حالت صحیح به حالت صحیح دیگر منتقل می‌نماید**».

۳. خاصیت انزوا

خاصیت **انزوا** [2]، بیان می‌کند که تراکنش‌های هم‌روند (هم‌زمان) طوری اجرا خواهند شد که گویا هرکدام به‌طور مجزا انجام می‌گردند. همان‌طور که می‌دانید، در پایگاه داده تراکنش‌ها می‌توانند به‌صورت هم‌روند اجرا شوند، ولی هم‌روندی آن‌ها کنترل می‌گردد تا اثر مخرب بر روی یکدیگر نداشته باشند. به‌عنوان‌مثال، فرض کنید قصد دارید مبلغ ۹۰۰۰۰۰ ریال به‌حساب A که موجودی آن یک‌صد هزار ریال است واریز نمایید و در همین زمان مبلغ ۱۰۰۰۰۰۰ ریال از باجه دیگر به‌صورت چک از حساب A برداشت می‌شود. اکنون حالتی را در نظر بگیرید که تراکنش اول (واریز پول به حساب) سقط گردد. آیا چک ۱۰۰۰۰۰۰ ریالی باید از باجه دیگر پاس شود یا خیر؟ برای چنین مواردی خاصیت انزوا از اثر مخرب تراکنش‌های هم‌روند بر روی یکدیگر جلوگیری می‌کند. یعنی، تا زمانی که وضعیت تراکنش باجه اول مشخص نشود، اجازه نمی‌دهد از باجه دوم چک پاس گردد.

۴. خاصیت پایایی یا ماندگاری

خاصیت **پایایی** [3] یا **ماندگاری**، بیان می‌کند که تراکنش‌هایی که تثبیت‌شده باشند، اثرشان ماندنی است و به‌هیچ‌وجه نباید اثر آن‌ها از بین برود. مثلاً، اگر مبلغی به‌حساب واریز شد یا از حسابی برداشت گردید، اثرش باید بماند. حتی، اگر زلزله بیاید، آتش‌سوزی اتفاق افتد، سیل یا سایر حوادث غیرمترقبه رخ دهد، اثر این تراکنش‌ها نباید از دست برود. یعنی، مانده‌حساب مشتری باید همواره صحیح باشد. بنابراین، عمل واریز یا

---

[1].Consistency [2].Isolation [3]. Durability

برداشت از حساب (یا هر تراکنش دیگر) باید قبل از اعلام تثبیت موفق به شکل ثبت رویداد و بعد از ثبت موفق به شکل نسخه پشتیبان نگه‌داری شود.

۲-۲-۱. حالات تراکنش

تراکنش از شروع تا خاتمه مراحل مختلفی را طی می‌کند. این مراحل در شکل ۲ - ۱ آمده‌اند.

همان‌طور که در این شکل می‌بینید، اجرای هر تراکنش با مرحله **شروع**، آغاز می‌شود. سپس، تراکنش به مرحله **انتظار** می‌رود. انتظار زمانی اتفاق می‌افتد که تراکنش به منبعی نیاز داشته باشد که این منبع در اختیار تراکنش دیگر باشد، باید منتظر بماند تا آن منبع آزاد شود و با آزاد شدن منبع، آن را در اختیار بگیرد و اجرای تراکنش را ادامه دهد. مراحل بین «**اجرا**» و «**انتظار**» ممکن است چندین بار تکرار شود تا تراکنش با موفقیت به پایان برسد یا با مشکلی برخورد کرده، **سقط** گردد.

- مرحله «**سقط**» هنگامی رخ می‌دهد که تراکنش با مشکلی مواجه شود. بنابراین، تراکنش نباید ادامه یابد. وقتی تراکنشی را سقط می‌کنید، بعضی از دستورات این تراکنش که بر روی پایگاه داده اجراشده‌اند و تغییراتی در آن داده‌اند، باید **لغو** (**خنثی UNDO**) گردند. یعنی، باید پایگاه داده به قبل از شروع تراکنش برگردد. این کار با دستور ROLLBACK یا ABORT انجام می‌پذیرد. مرحله «**آماده تثبیت**» زمانی اتفاق می‌افتد که تراکنش همه‌کارهای مربوطه را انجام داده و کافی است بخش‌های مختلفی که با آن‌ها سروکار داشته‌اند، برای خاتمه کار باهم هماهنگ گردند. در این مرحله نیز ممکن است تراکنش سقط گردد. زیرا، ممکن است بخش‌های مختلف باهم هماهنگ نشوند.
- در مرحله «**تثبیت**»[1] همه دستورات تراکنش برای نهایی کردن کار با یکدیگر توافق نموده‌اند. مرحله تثبیت با اجرای دستور COMMIT اتفاق می‌افتد.

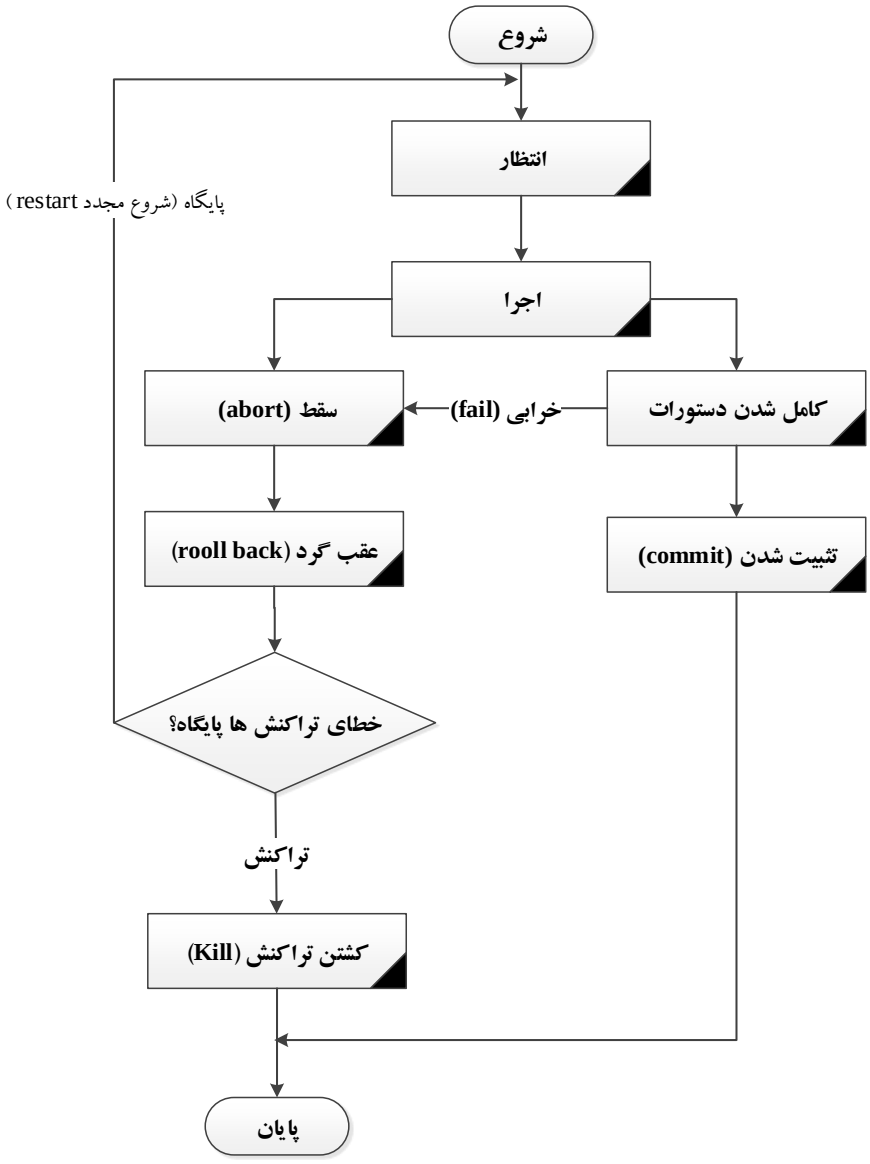
پس از اجرای عمل Commit (یا Abort) تراکنش تثبیت‌شده (یا سقط شده) تلقی می‌شود. تراکنشی که عمل Start مربوطه‌اش صادرشده باشد، ولی هنوز تثبیت یا سقط نشده باشد، **فعال**[2] نامیده می‌شود، تراکنشی که فعال باشد یا **سقط نشده** باشد، تراکنش **تثبیت نشده**[3] گویند.

یک تراکنش اگر نمی‌تواند به‌درستی کامل شود، عمل Abort را صادر می‌نماید. تراکنش ممکن است به دلیل خطا که نتوانسته آن را ترمیم کند (نظیر کافی نبودن موجودی در هنگام انتقال پول) خودش عمل Abort را به‌صورت ضمنی صادر کند یا Abort ممکن است روی یک تراکنش خارج از کنترل خودش توسط کاربر یا تراکنش دیگر تحمیل شود.

منبع صادرکننده COMMIT یا ABORT می‌تواند خود تراکنش یا سیستم پایگاه داده باشد.
عمل ABORT همیشه بر عمل COMMIT تقدم دارد.

---

[1]. Commit [2]. Active [3]. Uncommitted



شکل ۲ – ۱ دیاگرام وضعیت تراکنش.

۳–۱. انواع تراکنش‌ها

تراکنش‌ها را می‌توان با توجه مدت‌زمان اجرا به تراکنش‌های کوتاه و تراکنش‌های طولانی دسته‌بندی کرد. **تراکنش‌های کوتاه**، تراکنش‌هایی هستند که زمان اجرا و پاسخ‌دهی بسیار کوتاه (در حدود ثانیه یا دقیقه) را دارند و معمولاً به بخش کوچکی از پایگاه داده دسترسی دارند. در **تراکنش‌های طولانی**، مدت‌زمان اجرا و پاسخ‌گویی طولانی است(ممکن است چند دقیقه یا چند ساعت باشد) و به بخش بزرگی از پایگاه داده دسترسی دارند.

اما، با توجه به قوانین ترتیب **خواندن/ نوشتن** تراکنش‌ها به چهار نوع دسته‌بندی می‌شوند:

تراکنش‌های عمومی، در این نوع تراکنش، دستورات خواندن و نوشتن از ترتیب خاصی پیروی نمی‌کنند.

مثال ۱-۱. تراکنش زیر یک نمونه تراکنش عمومی است:

$T_1 = \{ R(D_1), R(D_2), W(D_2), R(D_3), W(D_1), W(D3), W(D_4), C\}$

تراکنش‌های دومرحله‌ای، در این نوع تراکنش، همه دستورات خواندن پیش از دستورات نوشتن اجرا می‌شوند.

مثال ۲-۱. تراکنش زیر یک نمونه تراکنش دو مرحله‌ای است:

$T_2 = \{ R(D_1), R(D_2), R(D_3), W(D_1), W(D3), W(D_4), C\}$

تراکنش‌های محدودشده یا نوشتن مقید[1]، در این نوع تراکنش، هر قلم داده باید قبل از به‌روزرسانی خوانده شود.

مثال ۳-۱. تراکنش زیر یک نمونه تراکنش محدود شده است:

$T_3 = \{ R(D_1), R(D_2), W(D_2), R(D_3), W(D_1), W(D_3), R(D_4), W(D_4), C\}$

تراکنش‌های دومرحله‌ای محدودشده ، در این نوع تراکنش، هر قلم داده باید قبل از به‌روزرسانی خوانده شود و تمام خواندن‌ها باید قبل از تمام نوشتن‌ها بی‌آید.

مثال ۴-۱. تراکنش زیر یک نمونه تراکنش دومرحله‌ای محدودشده است:

$T_4 = \{ R(D_1), R(D_2), R(D_3), R(D_4), W(D_1), W(D_3), W(D_2), W(D_4), C\}$

تراکنش ممکن است بر اساس **مدل عمل**[2] کار کند. یعنی، یا هر جفت عمل خواندن و نوشتن بر روی یک فقره داده پشت سر هم(بلافاصله بعد از یکدیگر) بی‌آیند.

مثال ۵-۱. در تراکنش زیر هر جفت عمل خواندن و نوشتن پشت سر هم می‌آیند:

$T_5 = \{ (R(D_1), W(D_1)), (R(D_2), W(D_2)), (R(D_3), W(D_3)), (R(D_4), W(D_4)), C\}$

یا **نوشتن کور**[3]، در این نوع تراکنش بدون این‌که داده‌ی را بخواند، بر روی آن می‌نویسد.

مثال ۶-۱. در تراکنش زیر نوشتن D_4 یک نوشتن کور است:

$T_6 = \{ R(D_1), R(D_2), W(D_2), R(D_3), W(D_1), W(D_3), W(D_4), C\}$

ولی تراکنش‌ها ازلحاظ تعداد سطوح به تراکنش‌های مسطح (تخت) و تراکنش‌های تودرتو (Nested transaction) طبقه‌بندی می‌شوند. **تراکنش‌های تخت،** یک نقطه شروع (BEGIN TRANSACTION) و یک نقطه پایان (END TRANSACTION) دارند و می‌توانند شامل هر تعداد دستورات ساده باشند.

مثال ۷-۱. تراکنش زیر یک تراکنش تخت یا مسطح است:

```
BEGIN TRANSACTION
        دستور ۱
        دستور ۲
          ...
        دستور N
END TRANSACTION
```

اما، **تراکنش‌های تودرتو،** از تعدادی تراکنش‌های فرعی مرتبط به هم تشکیل می‌شوند.

---

[1]. Constrainted Write [2]. Action Model [3]. Blind Write

مثال ۸-۱. تراکنش‌های زیر تودرتو هستند:

```
BEGIN TRANSACTION    T1
   . . .
      BEGIN TRANSACTION    T2
         . . .
            BEGIN TRANSACTION    T3
               . . .
            END TRANSACTION      T3
            . . .
      END TRANSACTION    T2
   . . .
END TRANSACTION    T1
```

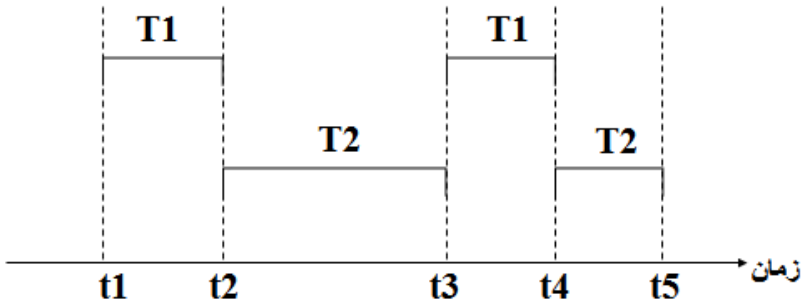
۴-۱. انواع اجرا

دو یا بیش از دو تراکنش به صورت‌های زیر اجرا می‌شوند:

۱. اجرای هم‌روند ۲. اجرای پی‌درپی

اجرای هم‌روند

وقتی دو یا چند تراکنش به‌صورت هم‌روند اجرا می‌شوند، عملیات پایگاه داده‌ی آن‌ها به‌صورت **در هم** (**متداخل** یا **قاطی**) اجرا می‌گردند(شکل ۳-۱). یعنی، عملیات یک برنامه ممکن است در بین دو عمل برنامه دیگر اجرا شود.



شکل ۳-۱ اجرای متداخل دو تراکنش T_1 و T_2.

۵-۱. مشکلات کنترل هم‌روندی

اجرای قاطی تراکنش‌ها می‌تواند منجر به تداخل یا رفتار نادرست برنامه‌ها و درنتیجه ناسازگاری پایگاه داده گردد. این تداخل به‌طور کامل نتیجه قاطی بودن است. یعنی، حتی اگر کد برنامه کاملاً درست باشد و هیچ قطعه‌ای از سیستم هم خطا نداشته باشد، می‌تواند تداخل اتفاق بی‌افتد. اجرای متداخل تراکنش‌ها ممکن است سبب مشکلاتی شود که در ادامه این مشکلات را می‌بینیم. هدف کنترل هم‌روندی اجتناب از تداخل و درنتیجه اجتناب از خطاها است. به‌عنوان‌مثال، فرض کنید، تراکنش‌های T_1 و T_2 به‌طور هم‌روند اجرا شوند. اگر این تراکنش‌ها بخواهند یک‌قلم مشترک داده از پایگاه داده را بخوانند یا بر روی آن بنویسند، برخی مشکلات می‌تواند اتفاق بی‌افتد. در این حالت چهار امکان برای **خواندن** (R) و **نوشتن** (W) وجود دارد. این چهار امکان

خواندن – خواندن (RR)، **نوشتن – خواندن** (RW)، **خواندن – نوشتن** (WR) و **نوشتن – نوشتن** (WW) نام دارند.

خواندن – خواندن (RR)، یعنی تراکنش‌های T_1 و T_2 یک داده مشترک از پایگاه داده را می‌خوانند. از آنجائی که خواندن نمی‌تواند در اجرای تراکنش‌ها برخورد[1] ایجاد نماید. پس دو خواندن هیچ مشکلی را به وجود نمی‌آورند. در فصل دوم با مفهوم برخورد بیش‌تر آشنا می‌شویم. اما بقیه حالات مشکلاتی را ایجاد می‌کنند که عبارت‌اند از.

۱. تغییرات گم‌شده[2] ۲. دستیابی به داده تثبیت نشده ۳. بازیابی ناسازگار

۱-۵-۱. تغییرات گم‌شده

به مثال بانکداری برگردید و فرض کنید برنامه‌ی را فراخوانی می‌کنید که واریز یک مبلغ به حسابی را انجام می‌دهد.

مثال ۹-۱. برنامه‌ای که مبلغی را به حسابی واریز می‌کند:

```
Procedure Deposit
begin
  Start;
  input( account#, amount);
  temp : = Read(Accounts[account#]);
  temp : = temp + amount;
  Write(Accounts[account#], temp);
  Commit
End
```

فرض کنید که موجودی حساب شماره ۱۳، 1000000 ریال است و مشتری‌های شماره‌های یک و دو به‌طور هم‌زمان مبالغ ۱۰۰۰۰۰ و ۱۰۰۰۰۰۰ ریال را به‌حساب شماره ۱۳ واریز می‌کنند. هر یک از مشتریان برنامه Deposit را به‌طور مستقل فراخوانی می‌کنند. بنابراین، برای انجام به‌روزرسانی هرکدام از مشتریان یک تراکنش مجزا ایجاد می‌شود. اجرای هم‌روند این دو رویه Deposit (دو تراکنش) می‌تواند دنباله‌های (ترتیب‌های) مختلفی از اعمال خواندن و نوشتن را ایجاد کند.

مثال ۱۰-۱. ترتیب اجرای زیر یک نمونه از ترتیب اجرای تراکنش‌ها را نشان می‌دهد. در مورد این ترتیب اجرا بحث کنید.

```
Read۱(Accounts[13])          // returns the value 1000۰۰۰
Read۲(Accounts[l3])           // returns the value 100۰۰۰0
Write۲(Accounts[l3], ۲۰۰۰۰۰۰)
Commit۲
Write۱(Accounts[l3], ۱۱۰۰۰۰۰)
Commit۱
```

با اجرای این دستورات، موجودی شماره‌حساب ۱۳، ۱۱۰۰۰۰۰ ریال خواهد شد که نادرست است (یعنی، Accounts[13] =۱۱۰۰۰۰۰). درصورتی‌که موجودی واقعی حساب شماره ۱۳ بعد از اجرای پی‌درپی این دستورات می‌بایست ۲۱۰۰۰۰ ریال باشد. تداخل اجرای دستورات مشتری اول و مشتری دوم موجب از دست رفتن نتیجه واریزی مشتری اول شده است. لذا، موجودی حساب شماره ۱۳ نادرست است. این **پدیده تغییر**

---

[1]. Conflict [2]. lost updated [3]. Lost Update Phenomenon

گم‌شده[3] نامیده می‌شود. وقتی این پدیده اتفاق می‌افتد که دو تراکنش در تلاش برای تغییر یک‌قلم داده هستند و هر دو تراکنش مقدار قبلی آن قلم داده را خوانده باشند و سپس به ترتیب مقدار جدید را در آن قلم داده بنویسند(مانند شکل زیر).

T1: r[x = 100] w[x:=600]
T2: r[x=100] [x:=500]

این مثال نوع برخورد نوشتن – نوشتن (WW) را نشان می‌دهد که تراکنش T_1 بر روی یک داده از پایگاه داده می‌نویسد و تراکنش T_2 نیز می‌خواهد بر روی همان داده بنویسد. اگر تراکنش T_2 اجازه یابد بر روی همان داده در پایگاه داده بنویسد، ممکن است باعث ایجاد مشکل تغییرات گم‌شده (نتیجه ازدست‌رفته) یا **نوشتن کثیف**[1] (ناجور) گردد.

۲-۵-۱. دست‌یابی به داده‌های تثبیت نشده

در تراکنش‌های هم‌زمان گاهی اتفاق می‌افتد که تراکنشی به داده‌های تثبیت نشده تراکنش‌های دیگر دسترسی می‌یابد. حال فرض کنید که موجودی حساب شماره ۱۳، 1000000 ریال است و مشتری‌های شماره یک و دو به‌طور هم‌زمان مبالغ ۱۰۰۰۰۰ و ۱۰۰۰۰۰۰ ریال را به‌حساب شماره ۱۳ واریز می‌کنند. به مثال زیر توجه کنید.

مثال ۱۱-۱. اگر یک اجرای هم‌روند دورویه Desposit به‌صورت زیر باشد. در مورد این ترتیب اجرا بحث کنید.

```
Read۱(Accounts[13])          // returns the value ۱۰00۰۰۰
Write۱(Accounts[۱3], ۱۱۰۰۰۰۰)
Read۲(Accounts[13])          // returns the value 1100۰۰۰
Abort۱
Write۲(Accounts[۱3], ۲1۰۰۰۰۰۰)
Commit۲
```

در پایان این اجرا، موجودی شماره‌حساب ۱۳، ۲۱۰۰۰۰۰ ریال خواهد بود (یعنی، Accounts[13] =۲۱۰۰۰۰۰). درواقع، موجودی نهایی حساب شماره ۱۳ می‌بایست ۲۰۰۰۰۰ ریال باشد. چون تراکنش T_1(مشتری اول) واریزی‌اش را کنسل نموده است. بااین‌وجود، موجودی نهایی حساب شماره ۱۳ غلط است. زیرا، تداخل اجرای دستورات مشتری اول و مشتری دوم موجب شده تا مشتری دوم واریزی تثبیت نشده(نهایی نشده) مشتری اول را بخواند و از آن استفاده کند. چون مشتری اول دستور Abort را صادر کرده است **دست‌یابی به داده‌های تثبیت نشده**[3] اتفاق افتاده است.

این مثال برخورد خواندن – نوشتن (WR) را نشان می‌دهد که تراکنش T_1 بر روی یک داده پایگاه داده می‌نویسد و سپس تراکنش T_2 می‌خواهد مقدار همان داده را بخواند. اگر به تراکنش T_2 اجازه داده شود تا این داده را بخواند، می‌تواند مشکل **وابستگی تثبیت نشده**[4] یا **خواندن کثیف** (خواندن ناجور Dirty Read) اتفاق افتد(مانند شکل زیر).

T1: r[x =200] w[x:=100] abort
T2: r[x=100]r [y=500]

---

[1]. Dirty Write [2].Uncommitted [3]. Uncommitted Dependency

۳-۵-۱. بازیابی ناسازگار

مشکل دیگر کنترل هم‌روندی توسط برنامه زیر(یعنی PrintSum که جمع مانده دو حساب مشتری را چاپ می کند) نشان داده‌شده است:

```
Procedure PrintSum
begin
  Start;
  input(account1, account2);
  temp1 : = Read( Accounts[ account1]);
  output(temp1);
  temp2 : = Read(Accounts[account2]);
  output(temp2);
  temp1 := temp1 + temp2;
  output(temp1);
  Commit
End
```

فرض کنید موجودی هر یک از حساب‌های شماره ۷ و ۸۶ دویست هزار ریال است و همان زمانی که مشتری شماره ۳ مجموع موجودی حساب‌های ۷ و ۸۶ را چاپ می کند (با استفاده از PrintSum)، مشتری شماره ۴ مقدار ۱۰۰ هزار ریال از حساب شماره ۷ به‌حساب شماره ۸۶ انتقال می‌دهد(با استفاده از رویه Transfer که قبلاً بحث گردید). نمونه‌ی از اجرای هم‌روند این دو تراکنش در زیر آمده است:

```
Read4(Accounts[7])              //    returns the value 200000
Write4(Accounts[7], 100000)
Read3(Accounts[7])              //returns the value 100000
Read3(Accounts[86])             //  returns the value 200000
Read4(Accounts[86])             //  returns the value 200000
Write4(Accounts[86], 300000)
Commit4
Commit3
```

تداخل رویه Transfer با PrintSum در این اجرا، منجر به چاپ مقدار ۳۰۰ هزار ریال می‌شود که مجموع صحیح موجودی‌های حساب‌های ۷ و ۸۶ نیست. چون رویه PrintSum، ۱۰۰ هزار ریال که در حال انتقال از حساب ۷ به‌حساب 86 هست را منظور نکرده است. دقت کنید که تداخل اجرای رویه Transfer و رویه PrintSum موجب این مشکل شده است. یعنی، رویه PrintSum واریزی رویه Transfer را درمجموع در نظر نگرفته است. چون هنوز واریزی رویه Transfer در پایگاه تثبیت نشده است. این نوع تداخل، یک **بازیابی ناسازگار** نامیده می‌شود. بازیابی ناسازگار وقتی رخ می‌دهد که تراکنش یک‌قلم داده‌ای را قبل از این‌که تراکنش دیگری آن را تغییر دهد، می‌خواند و قلم داده دیگر را پس از تغییر همان تراکنش می‌خواند. یعنی، یک تراکنش فقط بخشی از نتایج تغییر تراکنش دیگر را می‌بیند.

در این مثال نوع برخورد نوشتن – خواندن (RW) نشان داده است که تراکنش T_1 داده‌ی از پایگاه داده را می‌خواند. سپس، تراکنش T_2 بر روی همان داده می‌نویسد. در این حالت ممکن است دو مشکل زیر ایجاد شود:

- **تحلیل ناسازگاری**، اگر تراکنش T_1 داده‌ی از پایگاه داده را بخواند و سپس تراکنش T_2 اجازه یابد، بر روی همان داده بنویسد، مشکل تحلیل ناسازگاری اتفاق می‌افتد.

خواندن غیرقابل تکرار (تکرار نشدنی)، اگر تراکنش T_1، یک داده از پایگاه داده بخواند و سپس تراکنش T_2 بر روی همان داده بنویسد و در ادامه تراکنش T_1 دوباره همان داده را بخواند. در این صورت، مقدار خوانده‌شده قبلی و فعلی تراکنش T_1، فرق می‌کند. این مورد **خواندن تکرار نشدنی** نام دارد(مانند شکل زیر).

T1: r[x = 100] r[x=500]
T2: w[x:=500]

پس RW ممکن است موجب مشکلات تحلیل ناسازگاری و خواندن تکرار نشدنی گردد.

مثال ۱۲-۱. اجراهای زیر کدام‌یک از مشکلات را دارند. چرا؟(اندیس روی هر Read و Write معرف شماره تراکنش هست که آن عملیات را صادر کرده است. X و Y مقادیر داده هستند).

1) $Read_1(X), Write_1(X), Read_2(Y), Write_2(Y), Commit_1, Commit_2$
2) $Read_1(X), Write_1(X), Read_2(X), Write_2(X), Abort1_1, Commit_2$
3) $Read_1(X), Write_1(X), Read_2(X), Write_2(X), Commit_1, Commit_2$

گزینه ۱)، هیچ مشکلی ندارد. چون، دو تراکنش به دو داده مختلف دسترسی دارند. گزینه ۲)، مشکل دستیابی به داده تثبیت نشده را دارد. چون تراکنش T_2 به داده تثبیت نشده تراکنش T_1 دسترسی دارد. گزینه ۳)، مشکل تغییر گم‌شده را دارد. چون، تراکنش T_2 بر روی داده نوشته شده تراکنش T_1 نوشته است.

۶-۱. ترمیم‌پذیری

مکانیسم ترمیم باید در سیستم پایگاه داده حالتی را ایجاد کند که پایگاه داده شامل همه اثرات تراکنش‌های تثبیت‌شده و فاقد اثرات تراکنش‌های سقط شده باشد. اگر تراکنش‌ها هرگز سقط نشوند، ترمیم نسبتاً ساده است. چون درنهایت تمام تراکنش‌ها تثبیت می‌شوند، سیستم پایگاه داده عملیات پایگاه داده‌ی را همان‌گونه که می‌رسند به‌سادگی اجرا می‌کند. بنابراین، برای درک مفهوم ترمیم باید به پردازش سقط‌ها دقت کنیم. وقتی تراکنش سقط می‌شود، سیستم پایگاه داده باید اثرات آن را از بین ببرد. هر تراکنش(مانند T) دو نوع اثر دارد:

۱. **اثر روی‌داده**، یعنی مقادیری که تراکنش T در پایگاه داده نوشته است.

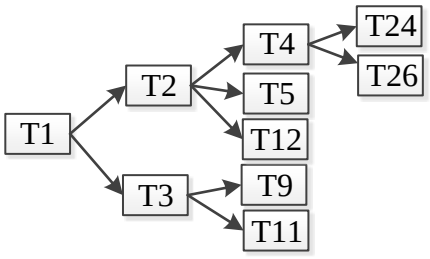
۲. **اثر روی تراکنش‌های دیگر**، یعنی، تراکنش‌های که مقادیر نوشته‌شده تراکنش T را خوانده‌اند.

سیستم پایگاه داده برای حذف اثرات تراکنش T باید برای هر قلم داده x که توسط تراکنش T به‌روز شده است، مقداری را بازیابی کند که انگار هرگز تراکنش T وجود نداشته است. می‌گوییم که سیستم پایگاه داده اعمال نوشتن تراکنش T را UNDO می‌کند. سیستم پایگاه داده باید اثرات تراکنش T متأثر از تراکنش‌های سقط شده را نیز حذف کند. سقط تراکنش‌های متأثر از تراکنش T خود ممکن است باعث سقط تراکنش‌های دیگر گردد که پدیده **انتشار سقط(سقط آبشاری)** نامیده می‌شود. یعنی، اگر یک تراکنش سقط گردد، بقیه تراکنش‌های وابسته به تراکنش سقط شده نیز باید سقط شوند. این عمل موجب سقط آبشاری می‌شود. سقط آبشاری در شکل ۴-۱ نشان داده‌شده است. در این شکل، اگر تراکنش T_1 سقط گردد، تراکنش T_{24} نیز باید سقط شود.

مثال ۱۳-۱. فرض کنید که مقادیر اولیه x و y یک هستند و اکنون حالتی را در نظر بگیرید که تراکنش‌های T_1 و T_2 عملیات زیر را صادر می‌کنند(سیستم پایگاه داده به ترتیب آن‌ها را اجرا می‌کند). اگر T_1 سقط شود چه رخ خواهد داد؟

$Write_1(x, 2); Read_2(x); Write_2(y, x);$

اگر T_1 سقط شده باشد، سیستم پایگاه داده عمل $Write_1(x, 2)$ را UNDO (خنثی یا لغو) می‌کند، مقدار یک را برای x بازیابی می‌نماید. چون T_2 مقدار نوشته‌شده x توسط تراکنش T_1 را خوانده بود، T_2 نیز باید سقط شود. بنابراین، سیستم پایگاه داده $Write_1(y, x)$ را UNDO می‌کند، یعنی، مقدار y را نیز به یک برمی‌گرداند.



شکل ۴-۱ نمایش ساختار درختی اجرای تراکنش و سقط آبشاری.

به خاطر بی‌آورید که پس از تثبیت یک تراکنش، سیستم پایگاه داده تضمین می‌کند که بعداً آن تراکنش را سقط نخواهد کرد. با توجه به‌احتمال سقط آبشاری، سیستم پایگاه داده درباره تضمین بیان‌شده باید خیلی دقیق باشد. حتی اگر تراکنش T عمل Commit را صادر کرده باشد، سیستم پایگاه هنوز ممکن است تصمیم به سقط کردن T را بگیرد. زیرا، تراکنش T هنوز در یک سقط آبشاری درگیر است. این هنگامی رخ می‌دهد که تراکنش T قلم داده‌ی از تعدادی تراکنش که بعداً سقط شده‌اند را بخواند. درنتیجه T نمی‌تواند تثبیت شود، مگر این‌که همه تراکنش‌هایی که داده‌های خوانده‌شده توسط T را تغییر داده‌اند، تضمین کنند که سقط نمی‌شوند. یعنی، تمام تراکنش‌ها تضمین کنند که تثبیت می‌شوند. اجرای تراکنش‌هایی که این شرط را برآورد می‌کنند، اجرای **ترمیم‌پذیر (قابل ترمیم)** نامیده می‌شوند.

این مسئله بسیار مهم است، پس اجازه دهید آن را دقیق‌تر بررسی کنیم. یعنی، مفهوم **خواندن از** (Read From) را بیان کنیم که می‌گوید تراکنش T_j در یک اجرا قلم داده x از تراکنش T_i را می‌خواند، اگر:

۱. T_j داده x را پس‌ازآن که T_i روی آن نوشته است، بخواند.

۲. T_i قبل از این‌که T_j داده x را بخواند سقط نشده باشد.

۳. هر تراکنش دیگر (در صورت وجود) که بین زمان نوشتن تراکنش‌های T_i و زمان خواندن تراکنش T_j روی x نوشته باشد، قبل از خواندن تراکنش T_j سقط شده باشد.

اگر تراکنش T_j قلم داده‌ای را از تراکنش T_i خوانده باشد، می‌گوییم که تراکنش T_j از تراکنش T_i خوانده است. **یک اجرا ترمیم‌پذیر است، اگر برای هر تراکنش T که تثبیت‌شده است، عمل Commit آن باید قبل از عمل Commit هر تراکنش دیگر که از T خوانده است، بی‌آید.**

برای اطمینان از این‌که سقط یک تراکنش، نحوه اعمال تراکنش‌های تثبیت‌شده دیگر را تغییر نمی‌دهد، ترمیم‌پذیری ضروری است. برای فهم بهتر این موضوع، اجازه دهید زیر را ببینید.

مثال ۱۴-۱. آیا اجرای زیر ترمیم‌پذیر است؟

$Write_1(x, 2); Read_2(x); Write_2(y, x); Commit_2;$

این اجرا ترمیم‌پذیری نیست، زیرا تراکنش T_2 مقدار x را از تراکنش T_1 خوانده است، ولی دستور Commit تراکنش T_2 قبل از دستور Commit تراکنش T_1 آمده است (هنوز تراکنش T_1 فعال است که تراکنش T_2 تثبیت کرده است). برای ترمیم‌پذیر بودن این اجرا، تراکنش T_1 باید قبل از تراکنش T_2 تثبیت کند.

مسئله این است که اکنون اگر تراکنش T_1 سقط شود، چه خواهد شد، اکنون T_2 باید به‌تنهایی اجرا شود که در این حالت با عمل $Read_2(x)$ تراکنش T_2 برخورد دارد. یعنی، $Read_2(x)$ به‌طور واقعی مقدار ۲ را برمی‌گرداند، اما اگر T_1 سقط شده باشد $Read_2(x)$ باید همان مقداری را برمی‌گرداند که x قبل از اجرای $Write_1(x, 2)$ داشته است. به‌طور مشابه می‌توانیم تراکنش T_2 را سقط کنیم، در این صورت با اجرای Commit تراکنش T_2 برخورد دارد. درهرصورت، هر دو روش غیرممکن است. ولی، اگر سیستم پایگاه داده $Commit_2$ را به تأخیر بی‌اندازه، اجرا ترمیم‌پذیر می‌شود و هیچ مشکلی با سقط T_1 ایجاد نمی‌شود. **به‌طورکلی، به تأخیر انداختن Commitهای قطعی یک تراکنش روشی است که سیستم پایگاه داده می‌تواند مطمئن شود که اجراها ترمیم‌پذیر هستند.**

۷-۱. پایانه ورودی / خروجی

همان‌طور که ملاحظه شده، اجرایی ترمیم‌پذیر است که اگر سیستم پایگاه داده همیشه قادر باشد اثرات یک تراکنش سقط شده روی تراکنش‌های دیگر را خنثی نماید. تعریف ترمیم‌پذیری بر اساس این فرضیه هست که تمام این اثرات از طریق خواندن‌ها و نوشتن‌ها هستند. بدون در نظر گرفتن این فرض، تعریف ترمیم‌پذیری صحیح نیست. نوع دیگری در تعامل بین تراکنش‌ها وجود دارد که تعریف ترمیم‌پذیری را زیر سؤال می‌برد، یعنی، تعامل از طریق کاربران است. یک تراکنش می‌تواند از طریق دستورات Input و Output با پایانه یا دستگاه‌های ورودی و خروجی محاوره نماید. چون یک کاربر با خواندن خروجی یک تراکنش تصمیم می‌گیرد که ورودی تراکنش بعدی چه باشد، دستورات Input و Output روش‌های دیگر محاوره غیرمستقیم تراکنش‌ها هستند. به‌عنوان‌مثال، فرض کنید تراکنش T_1 قبل از تثبیت خودش خروجی را روی پایانه می‌نویسد. فرض کنید کاربری اطلاعات روی صفحه پایانه را می‌خواند، و بر اساس آن تصمیم می‌گیرد تا ورودی‌هایی برای تراکنش دیگر مثل T_2 وارد کند. اکنون حالتی را در نظر بگیرید که تراکنش T_1 سقط شود، ازآنجایی‌که عملیات اجرای T_2 به‌طور غیرمستقیم بر اساس خروجی تراکنش T_1 بوده است، چون T_1 سقط کرده است، T_2 نیز باید سقط شود، این پدیده نیز نوعی سقط آبشاری است. متأسفانه، سیستم پایگاه داده درباره این وابستگی (ارتباط) بین T_1 و T_2 نمی‌داند، و بنابراین نمی‌تواند به‌طور خودکار درباره این نوع سقط آبشاری تصمیم بگیرد.

در حقیقت، در این حالت منشأ خطا، کاربر است. تا زمانی که سیستم پایگاه داده پیام "**تراکنش T_1 تثبیت‌شده است**" را روی پایانه کاربر ننوشته باشد، کاربر نباید به خروجی تولیدشده T_1 اعتماد کند. از طرف دیگر، تا زمانی که پیام ظاهر نشود، کاربر نمی‌داند که آیا T_1 تثبیت خواهد شد یا نه، یعنی، ممکن است سقط شود و درنتیجه خروجی پایانه نامعتبر می‌گردد. کاربر به نادرست فرض کرد که خروجی پایانه T_1 باید تثبیت‌شده باشد، و بنابراین، قبل از موعد مقرر اثرات تراکنش T_1 به تراکنش دیگری تسری داده‌شده است.

سیستم پایگاه داده می‌تواند از تکثیر اثرات تراکنش تثبیت نشده T توسط کاربران، به‌وسیله‌ی به تعویق انداختن دستور خروجی T تا بعد از تثبیت آن جلوگیری کند. پس، کاربر می‌تواند فقط خروجی تثبیت‌شده را ببیند. گاهی اوقات برای سیستم پایگاه داده قابل‌پذیرش است که به تعویق انداختن نتیجه خروجی را قبول کند. به‌طور خاص، اگر هر تراکنش تمامی ورودی را قبل از آن‌که خروجی تولید کند، از کاربر تقاضا نماید به‌خوبی کار می‌کند.

ولی اگر تراکنش T پیغامی را بر روی پایانه بنویسد و متعاقباً از کاربر ورودی درخواست کند، خروجی را به تعویق بی‌اندازد و کاربر را در وضعیت غیرقابل‌قبول قرار می‌دهد. پاسخ کاربر به درخواست ورودی T به خروجی تثبیت نشده بستگی دارد که هنوز دیده نشده‌اند. در این حالت، سیستم پایگاه داده باید قبل از آن‌که T تثبیت شود خروجی را به سمت پایانه ارائه کند.

فرض کنید سیستم پایگاه داده خروجی را ارائه می‌دهد و سپس کاربر به درخواست ورودی T پاسخ دهد و اکنون فرض کنید T سقط شود. بسته به دلیل این‌که چرا T سقط شده، کاربر ممکن است انتخاب کند که اجرای مجدد T را آزمایش کند. چون، بقیه تراکنش‌ها می‌توانند بین فاصله زمانی که T سقط و راه‌اندازی شده است، اجرا شوند. اجرای دوم T می‌تواند وضعیت مختلفی از پایگاه داده را نسبت به اجرای اولیه آن بخواند.

بنابراین، T می‌تواند خروجی متفاوت را تولید کند که به کاربر پیشنهاد دهد به ورودی متفاوتی نسبت به اجرای اولیه T نیاز دارد. بنابراین، در اجرای مجدد T، سیستم پایگاه داده نمی‌تواند مجدداً از اجرای اولیه T برای ورودی پایانه استفاده کند.

۸-۱. اجتناب از سقط‌های آبشاری

در اجرای ترمیم‌پذیر، امکان سقط‌های آبشاری وجود دارد. از طرف دیگر، سقط‌های آبشاری باید وجود داشته باشد تا تضمین شود یک اجرا ترمیم‌پذیر است. برای درک این موضوع مثال زیر را در نظر بگیرید:

مثال ۱۵-۱. آیا اجرای زیر ترمیم‌پذیر است؟ آیا از سقط آبشاری اجتناب می‌کند؟

$Write_1(x, 2); Read_2(x); Write_2(y, x); Abort_1;$

در نگاه اول، این اجرا ترمیم‌پذیر است. چون T_2 مقدار x را از تراکنش T_1 خوانده است و Abort تراکنش T_1 قبل از T_2 آمده است. اما چون تراکنش T_1 سقط شده است، T_2 نیز باید سقط شود، بنابراین، اجرای فوق ترمیم‌پذیر نخواهد بود. این مفهوم همان سقط آبشاری است. پس این اجرا از سقط آبشاری اجتناب نمی‌کند.

سقط آبشاری به دو دلیل خوشایند نیست، اول، تلاش زیادی برای نگه‌داری این‌که چه تراکنشی از تراکنش‌های دیگر خوانده است، لازم دارد. دوم، تعداد زیادی تراکنش به‌طور غیرقابل‌کنترل به خاطر اتفاقی که برای تراکنش دیگر رخ‌داده است، باید سقط شوند. این موارد بیان‌شده سقط آبشاری را پدیده نامطلوبی می‌سازد. زیرا،

۱. موجب اتلاف منابع می‌شود.

۲. برای مدیریت سقط آبشاری باید درخت مربوط به تراکنش‌ها را نگه‌داری کرد.

در عمل، سیستم‌های پایگاه داده طوری طراحی‌شده‌اند تا از سقط‌های آبشاری اجتناب کنند.

می‌گوییم که یک سیستم پایگاه داده از سقط‌های آبشاری جلوگیری می‌کند، اگر هر تراکنش فقط مقادیری را بخواند که توسط تراکنش تثبیت شده نوشته شده باشد. بنابراین، در چنین اجرای فقط تراکنش تثبیت شده روی تراکنش‌های دیگر اثر می‌گذارد.

برای تأمین اجرای فاقد سقط آبشاری، سیستم پایگاه داده باید هر Read(x) را تا زمانی که تمام تراکنش‌های قبلی‌ای که یک Write(x, Val) را اجرا کردند، تثبیت یا سقط کنند، به تأخیر بی‌اندازد. درواقع، با انجام این عمل ترمیم‌پذیری نیز حاصل می‌شود.

برای اجتناب از سقط شدن آبشاری، باید جلوی **خواندن از** (Read From) را بگیریم. یعنی، فقط از تراکنش‌های تثبیت شده بخوانیم. به عبارت دیگر، دستور READ را به تعویق بی‌اندازیم تا تراکنش قبلی که بر روی داده نوشته است را COMMIT کرده، بعدازآن بخواند.

مثال ۱۶-۱. آیا از سقط آبشاری اجتناب می‌کند؟

$Read_1(X)\ Write_1(X)\ ...\ Read_2(X)\ ...\ Commit_1$

در اینجا، چون $Read_2(x)$ (تراکنش ۲، x را خوانده) که پس از $Write_1(x)$ (نوشتن تراکنش، بر روی x) قرار دارد و تراکنش ۱، تثبیت نکرده است ($Commit_1$) قبل از $Read_2(X)$ قرار ندارد، پس سقط آبشاری اتفاق می‌افتد.

مثال ۱۷-۱. آیا از سقط آبشاری اجتناب می‌کند؟

$Read_1(X)\ Write_1(X)\ ...\ Commit_1\ Read_2(X)$

در اینجا، چون $Read_2(x)$ (تراکنش ۲، x را خوانده) که پس از $Write_1(x)$ (نوشتن تراکنش، بر روی x) قرار دارد و تراکنش ۱، تثبیت ($Commit_1$) قبل از $Read_2(X)$ قرار دارد، پس در این اجرا سقط آبشاری اتفاق نمی‌افتد.

اجراهای محض

متأسفانه، همیشه در عمل اجرای فاقد سقط‌های آبشاری و ترمیم‌پذیر کافی نیست. اغلب یک محدودیت دیگری روی اجراها وجود دارد. برای توضیح این محدودیت لغو نوشتن‌های یک تراکنش را در نظر بگیرید. یعنی، برای قلم داده x که تراکنشی بر روی آن نوشت، می‌خواهیم مقدار x را به مرحله قبل از اجرای تراکنش برگردانیم. یعنی، انگار این تراکنش هرگز انجام‌نشده است. اجازه دهید این موضوع را دقیق‌تر بررسی کنیم. فرض کنید که تراکنش T در یک اجرا روی قلم داده x می‌نویسد و سپس سقط می‌شود. اگر اجرا از سقط‌های آبشاری جلوگیری کرده باشد لازم نیست تراکنش دیگری سقط شود. اکنون، پس از اجرای موردنظر تمام عملیات متعلق به تراکنش T را حذف کنید. نتیجه یک اجرای جدید به دست می‌آید. " حاصل آن مقدار x را بدون در نظر گرفتن تراکنش T می‌دهد". دقیقاً مقدار x در این اجرا جدید است. به مثال زیر توجه کنید.

مثال ۱۸-۱. در مورد اجرای زیر بحث کنید:

$Write_1(x, 1);\ Write_1(y, 3);\ Write_2(y, 1);\ Commit_1;\ Read_2(x);\ Abort_2.$

چون T_2 سقط شده است، پس این اجرا با حذف عملیات T_2 به‌صورت زیر است:

$Write_1(x, ۱);\ Write_1(y, 3);\ Commit_1;$

بعد از این اجرا، بدیهی است که مقدار y، ۳ است. یعنی، پس از سقط تراکنش T_2 مقداری که باید y داشته باشد، ۳ است.

تصویر قبل[1]، از یک Write(x, Val) در یک اجرا، مقداری است که x قبل از این عمل داشته است. برای نمونه، در مثال فوق، مقدار تصویر قبل $Write_2(y, 1)$، ۳ است. این مقداری است که وقتی T_2 (زمانی که

[1].Before Image

$Write_2(y, 1)$ اجر شد)، سقط می‌شود باید برای y بازیابی شود. پیاده‌سازی سقط با بازیابی مقادیر تصویر قبل تمام نوشتن‌های یک تراکنش خیلی راحت است. خیلی از سیستم‌های پایگاه داده با این روش کار می‌کنند. متأسفانه این روش همیشه درست عمل نمی‌کند، مگر این‌که فرضیات بیش‌تری درباره اجراها داشته باشیم. مثال ذیل این مشکلات را شرح می‌دهد.

فرض کنید مقدار اولیه x یک باشد. اجرای زیر را در نظر بگیرید:

$Write_1(x, 2); Write_2(x, 3); Abort_1.$

تصویر قبل $Write_1(x, 2)$ مقدار اولیه (یعنی همان یک) است. اما مقداری که باید وقتی‌که T_1 سقط گردید، "بازیابی" شود، ۳ (یعنی، همان مقدار نوشته شده توسط تراکنش T_2) است. این موردی است که سقط T_1 نباید اثری روی x بگذارد، چون‌که x بعد از این‌که توسط تراکنش T_1 نوشته‌شده بود، مجدداً توسط تراکنش T_2 بازنویسی شده است. توجه کنید که این مورد سقط آبشاری نیست، چون T_2، بدون این‌که خواندن قبلی از x داشته باشد، در آن نوشته است (یعنی، T_2 از T_1 نخوانده است). برای فهم بیش‌تر این موضوع، فرض کنید که اکنون T_2 هم سقط شود. یعنی، داشته باشیم:

$Write_1(x, 2); Write_2(x, 3); Abort_1; Abort_2.$

تصویر قبل $Write_2(x, 3)$ همان مقدار نوشته‌شده توسط تراکنش T_1 (یعنی، ۲) است. هرچند، مقدار x بعد از $Write_2 (x, 3)$ لغو شده باید همان مقدار اولیه x (یعنی یک) باشد (چون هر دو به روزرسانی x سقط شداند). در این مورد مشکل این است که تصویر قبل توسط یک تراکنش سقط شده مجدداً نوشته‌شده است.

این مثال‌ها، تناقض بین مقادیری که وقتی یک تراکنش سقط شده است باید بازیابی شوند و مقادیر که در تصویر قبل اقلام داده‌ی تثبیت‌شده است را نشان می‌دهد. این ناسازگاری وقتی اتفاق می‌افتد که دو تراکنش خاتمه نیافته (فعال) بر روی‌داده مشترکی نوشته باشند. توجه کنید که اگر T_1 قبل از این‌که T_2، روی x نوشته، سقط شده بود (یعنی، اگر $Abort_1$ و $Write_2 (x, 3)$ در مثال قبلی جایشان تعویض شود)، هیچ مشکلی ایجاد نمی‌شود. تصویر قبل $Write_2 (x, 3)$ باید یک باشد نه ۲، چون تراکنشی که ۲ را نوشت باید مجدداً سقط شود. بنابراین، وقتی T_2 سقط می‌شود، تصویر قبل $Write_2 (x, 3)$ باید مقدار یک باشد که می‌بایست برای x بازیابی می‌شد. به‌طور مشابه، اگر T_1 قبل از این‌که T_2 روی x نوشته باشد، تثبیت‌شده بود، پس‌تصویر قبل از $Write_2(x, 3)$ باید ۲ باشد، دوباره اگر T_2 سقط گردید، مقدار یک باید برای x بازیابی شود.

می‌توانیم تا زمانی که همه تراکنش‌هایی که قبلاً روی x نوشته‌اند، تثبیت یا سقط گردند، اجرای $Write(x, Val)$ را به تأخیر بیاندازیم.

> **اجراهای محض**، اجراهایی هستند که در آن تراکنش تنها اقلام داده‌ای را می‌خواند که اثر تراکنش تثبیت‌شده باشد و تنها بر روی اقلام داده‌ای می‌نویسد که اثر تراکنش آن‌ها تثبیت‌شده باشد.
> وقتی اجرا محض می‌گردد، کنترل هم‌روندی کاهش می‌یابد.

مثال ۱۹-۱. آیا اجرای زیر محض است؟

$Write_1(x, 2); Write_2(x, 3); Commi_1, Commit_2$

چون در این اجرا تراکنش ۲ بر روی داده نوشته شده تراکنش ۱ قبل از تثبیت آن نوشته است، پس اجرا محض نیست.

اجراهای پی‌درپی

در بخش‌های قبل با ارائه مثال‌هایی مشاهده کردیم که به دلیل تداخل اعمال تراکنش‌های مختلف، ممکن است خطا ایجاد شود. این مثال‌ها تمام حالات مختلف که اجرای تراکنش به‌طور هم‌روند می‌تواند تداخل داشته باشند را نمایش نمی‌دهند، اما، حداقل سه مسئله تداخل که اغلب اتفاق می‌افتد را نشان داده‌اند. برای جلوگیری از این سه مشکل و مشکلات دیگر تراکنش، باید انواع تداخل بین تراکنش‌ها کنترل شود.

روش اول اجتناب از مشکلات تداخل این است که اجازه ندهیم تراکنش‌ها عملیات قاطی باهم داشته باشند. اجرای که در آن دو تراکنش (تراکنش‌ها) قاطی نباشند، **سریال (پی‌درپی** Serial) نامیده می‌شود. به‌عبارت‌دیگر، یک اجرا سریال است، اگر، برای هر زوج تراکنش همه عملیات یک تراکنش قبل از هر یک از عملیات تراکنش دیگر اجرا شود. **در یک مجموعه $T=\{T_1, T_2, \ldots, T_n\}$ که شامل n تراکنش است، !n سناریوی پی‌درپی قابل‌تعریف است.** از دید کاربر، در یک اجرای سریال، اجرای تراکنش‌ها طوری به نظر می‌رسند که عملیات روی سیستم پایگاه داده به‌صورت اتمیک اجرا می‌شوند. اجراهای سریال درست هستند، زیرا اجرای هر تراکنش به‌طور مجزا صحیح است، و تراکنش‌هایی که به‌صورت سریال اجرا می‌شوند نمی‌توانند تداخلی با یکدیگر داشته باشند.

یک نیازمندی این است که سیستم پایگاه داده واقعاً تراکنش‌ها را به‌صورت سریال پردازش کند. گرچه، این بدین معنی است که سیستم پایگاه داده نتواند تراکنش‌ها را به‌صورت هم‌روند اجرا کند. هم‌روندی معنی اجراهای قاطی شده را می‌دهد. بدون هم‌روندی، استفاده مفیدی از منابع سیستم نمی‌شود، و بنابراین، ممکن است کارا نباشد (کارایی سیستم خیلی پایین آید). فقط در ساده‌ترین سیستم‌ها، اجرای سریال یک راه‌حل عملی برای اجتناب از تداخل است.

راه‌حل دوم این است که اجازه دهیم اجراهایی وجود داشته باشند که همان اثر اجرای سریال را داشته باشند. چنین اجراهای، اجرای **پی‌درپی پذیر**[1] نامیده می‌شوند. به‌عبارت‌دیگر، اجرای پی‌درپی پذیر است، که اگر خروجی و اثری مشابه اجرای پی‌درپی همان تراکنش‌ها (یکی از !n سناریوی پی‌درپی باشد) را روی پایگاه داده تولید کند. اجرای پی‌درپی درست است، چون هر اجرای پی‌درپی پذیر همان اثر مشابه یک اجرای پی‌درپی را دارد، پس اجراهای پی‌درپی پذیر هم درست هستند.

اجراهای که بازیابی ناسازگار و تغییرات گم‌شده را داشته باشند، پی‌درپی پذیر نیستند. به‌عنوان‌مثال، اجرای پی‌درپی دو تراکنش Deposit، به هر ترتیبی، نتایج متفاوتی باحالتی که دو تراکنش قاطی شده اجرا شوند یا تغییرات را گم کنند، می‌دهد. بنابراین، اجرای قاطی پی‌درپی پذیر نیست. به‌طور مشابه، اجرای قاطی تراکنش Transfer و PrintSum اثر متفاوتی باحالتی که دو تراکنش پی‌درپی اجرا شوند، دارد و بنابراین، پی‌درپی پذیر نیستند.

هرچند دو اجرای قاطی شده بالا پی‌درپی پذیر نیستند، موارد زیاد دیگری وجود دارند که پی‌درپی پذیر هستند. به‌عنوان‌مثال، اجرای قاطی Transfer و PrintSum زیر را در نظر بگیرید:

```
Read4(Accounts[7])              // returns the value 200000
Write4 (Accounts[7], 100000)
```

[1]. Serializable Execution

```
Read3(Accounts[7])              // returns the value 100000
Read4(Accounts[ 86])            // returns the value 200000
Write4(Accounts[86], 300000)
Commit4
Read3(Accounts[86])             // returns the value 5300000
Commit3
```

این اجرا همان اثری را دارد که Transfer و PrintSum به‌صورت پی‌درپی اجرا می‌شوند. یعنی، اجرای پی‌درپی $T_4 \rightarrow T_3$) T_4T_3 است. در چنین اجرای پی‌درپی، $Read_3$(Account[7]) فوراً مقدار $Write_4$(Account s[86], 300000) را برمی‌گرداند. هرچند ترتیب اجرای این عملیات با اجرای قاطی شده متفاوت است، اثر (نتیجه) هر یک از عملیات دقیقاً همان اجرای قاطی شده است. بنابراین، این اجرا قاطی شده پی‌درپی پذیر است.

پی‌درپی پذیری بیان صحت کنترل هم‌روندی در سیستم‌های پایگاه داده است. با توجه به اهمیت این مفهوم، اجازه دهید مزایا و معایب آن را شرح دهیم.

سیستم پایگاه داده‌ای که اجراهایش پی‌درپی پذیر هستند، درک آن ساده است. چنین سیستم پایگاه داده برای کاربران شبیه یک پردازش‌گر تراکنش ترتیبی به نظر می‌رسد. برنامه هر تراکنش را به شکل اجرای مجزا و مستقل از اعمال تراکنش‌های دیگر روی ماشین مستقل می‌بینند.

یک سیستم پایگاه داده که اجراهای پی‌درپی را تولید می‌کند از تداخل بیان‌شده در مثال‌های تغییرات گم‌شده و بازیابی ناسازگار اجتناب می‌کند. یک تغییر گم‌شده وقتی اتفاق می‌افتد که دو تراکنش به‌طور پشت سر هم بر روی یک‌قلم داده می‌نویسند. این عمل نمی‌تواند در اجرای پی‌درپی اتفاق بی‌افتد، چون یکی از تراکنش‌ها قلم داده‌ای که توسط تراکنش دیگر نوشته‌شده را می‌خواند. چون هر اجرا پی‌درپی پذیر همان اثر اجرای پی‌درپی را دارد، اجراهای پی‌درپی پذیر از تغییرات گم‌شده جلوگیری می‌کنند.

یک بازیابی ناسازگار وقتی اتفاق می‌افتد که یک تراکنش برخی از اقلام داده‌ی را قبل از این‌که تراکنش آن‌ها را به‌روز کند، می‌خواند و برخی اقلام داده‌ی دیگر بعدازاین‌که تراکنش به‌روزرسانی آن‌ها را انجام داده است، می‌خواند. این عمل نمی‌تواند در اجرای پی‌درپی اتفاق بی‌افتد، زیرا، تراکنش بازیابی تمام اقلام داده‌ی یا قبل از این‌که تراکنش به‌روزرسانی (در هر به‌روزرسانی) اجرا شود یا بعدازاین‌که تراکنش به‌روزرسانی تمام به‌روزرسانی خودش را اجرا کند، می‌خواند. چون، هر تراکنش پی‌درپی پذیر همان اثر اجرای پی‌درپی را دارد، اجراهای پی‌درپی پذیر همچنین از بازیابی‌های ناسازگار اجتناب می‌کنند.

۹–۱. حفظ سازگاری

مفهوم بازیابی سازگار می‌تواند به‌کل پایگاه داده تعمیم یابد. یعنی، فقط به اقلام داده‌ی که توسط یک تراکنش بازیابی می‌گردند، محدود نمی‌شود. این تعمیم توصیف دیگری از ارزش پی‌درپی پذیری را فراهم می‌کند. فرض کنید که بعضی از وضعیت‌های پایگاه داده که تعریف می‌شوند سازگار باشند. طراح پایگاه داده می‌تواند مستندات سازگاری را تعریف کند. این مستندات، برای وضعیت‌های سازگار درست و برای وضعیت‌های ناسازگار نادرست است. به‌عنوان‌مثال، فرض کنید که در پایگاه داده بانک‌داری فیلد Accounts شامل یک‌قلم داده‌ای، فیلد Total، که شامل مجموع موجودی‌های تمام شماره‌حساب‌ها است، باشد. یک