**بسمه تعالی**



برنامه نویسی بانک اطلاعاتی SQL Server

**نویسنده : محمد حسین فخرآوری**

تراکنش یک واحد کاری منطقی است که عملی را بر روی پایگاه داده انجام می‌دهد. عموما تراکنش‌ها دنباله ای از عملیات پایگاه داده هستند که رویه هم رفته انجام یک کار یا وظیفه را بر عهده دارند. نکته مهمی که در مورد تراکنش‌ها مطرح می‌شود اینست که آنها باید به گونه ای مدیریت شوند که پایگاه داده را از یک وضعیت سازگار و درست (consistent) به وضعیت سازگار دیگری ببرند. به بیان دیگر اگر تراکنش از چند عملیات تشکیل شده باشد، پس از پایان اجرای تمامی عملیات مربوط به تراکنش نباید در داده‌های پایگاه داده هیچ تناقضی با قوانین پایگاه داده (integrity rules) بوجود بیاید. مزیت استفاده از تراکنش نیز همین مسئله است که به توسعه دهنده نرم افزار این اطمینان را می‌دهد که صحت و درستی پایگاه داده در اثر اجرای دستورات او از بین نخواهد رفت. علاوه بر آن اگر در حین اجرای یکی از دستورات خللی ایجاد گردد، پایگاه داده دوباره به وضعیت سازگار قبلی خود باز گردانده خواهد شد. نسل‌های اولیه سیستم‌های مدیریت پایگاه داده فاقد پیاده سازی تراکنش بودند و بهمین دلیل توسعه دهندگان کار بسیار مشکلی در شبیه سازی این واحد‌های یکپارچه منطقی داشتند. خوشبختانه اکثر DBMS‌های امروزی این مفهوم مهم را پشتیبانی می‌کنند و نیازی به نگرانی در مورد پیاده سازی آن نیست. تنها کاری که لازم است انجام گیرد کسب مهارت در زمینه استفاده از آنهاست.

تعریف تراکنش‌ها و مشخص کردن عملیات موجود در آنها اغلب توسط خود توسعه دهنده برنامه صورت می‌گیرد. اوست که تعیین می‌کند تراکنشش باید چه عملیاتی را با چه ترتیبی انجام دهد. اما در کنار این قسم از تراکنش‌ها که توسط کاربران تعریف می‌شود، تراکنش‌های دیگری نیز وجود دارند که توسط خود سیستم مدیریت پایگاه داده تعریف می‌شوند. به این قبیل تراکنش‌ها که واحد‌های کاری بسیار کوچک و عموما تجزیه ناپذیری هستند تراکنش‌های خودکار یا auto transactions گفته می‌شود. بعنوان مثال اگر ما تراکنشی را تعریف کرده باشیم که شامل یک عمل خواندن و یک عمل درج باشد، در هنگام اجرا سیستم این تراکنش را به دو تراکنش کوچکتر می‌شکند که در یکی عمل خواندن و در دیگری عملی نوشتن و درج را انجام می‌دهد. البته توجه داشته باشید که اگر چه این دو عملیات جدا و مستقل از هم اجرا می‌شوند اما رابطه منطقی آنها با یکدیگر  حفظ می‌شود و در صورت خللی در یکی از آنها اثر دیگری نیز بازگردانده شده و پایگاه داده دوباره به حالت قبل از جرا برگردانده می‌شود. به این کار عمل undo شدن تراکنش گفته می‌شود.

گفتیم که تعریف تراکنش توسط کاربر صورت می‌پذیرد و مدیریت آن بر عهده پایگاه داده قرار می‌گیرد. در این میان نکته حائز اهمیتی وجود دارد که در اینجا باید به آن اشاره شود. اندازه تراکنش نقشی بسیار موثر در کارایی پایگاه داده ایفا می‌کند. توجه داشته باشید که اندازه تراکنش‌ها نباید خیلی بزرگ باشد. چراکه منجر به بزرگ شدن بیرویه فایل مربوط به ثبت وقایع پایگاه داده (log file) می‌گردد. تمامی علیات تاثیر گذار بر روی پایگاه داده در این فایل ثبت می‌شوند تا در موقع لزوم بتوان با استفاده از عمل بازیابی و ترمیم پایگاه داده (recovery) را انجام داد. بزرگ بودن این فایل در هنگام ترمیم می‌تواند بر روی کارایی تاثیر گذار باشد. علاوه بر این موضوع اندازه تراکنش‌ها اثر سوء دیگری نیز می‌تواند در پی داشته باشد و آن محدود نمودن درجه همروندی است. یعنی اگر اندازه تراکنش بیش از حد معمول باشد ممکن است بر روی تعداد تراکنش هایی که می‌توانند بطور موازی و همزمان اجرا شوند تاثیر منفی بگذارد. چرا که معمولا در آغاز تراکنش بر روی منابعی که مورد استفاده تراکنش قرار می‌گیرد قفل گذاری می‌شود تا بگونه ای مسئله نواحی بحرانی حل شود. این قفل‌ها زمانی آزاد می‌شوند که تمامی عملیات داخل تراکنش بطور کامل اجرا شده باشند یا اینکه مشکلی در حین اجرا بوجود آید. در این صورت هرچه تراکنش بزرگ‌تر باشد اجرای آن بیشتر طول خواهد کشید و در نتیجه قفل‌های آن نیز دیر‌تر آزاد می‌شوند. بدین ترتیب سایر تراکنش هایی که می‌خواهند از منابع مشترک استفاده کنند باید تا پایان اجرای تراکنش بزرگ ما منتظر بمانند. این مسئله یعنی کاهش درجه اجرای موازی با همروندی که اگر در سیستم‌های بزرگ به آن دقت نشود به گلوگاهی تبدیل خواهد شد و کارایی را به نحو قابل توجهی کاهش می‌دهد.

**تعریف تراکنش‌ها**

بدنه اصلی هر تراکنش را چهار کلمه کلیدی تشکیل می‌دهند

**BEGIN TRANSACTION - END TRANSACTION - ROLLBACK - COMMIT**

**بررسی مفهوم ACID**

کلمه **ACID** خلاصه شده چهار کلمه Atomicity Consistency Isolation Durability می باشد.

که به ترتیب از سمت چپ به راست دارای معانی اتمی بودن، سازگاری، جداسازی و پایداری می باشند.

**Atomicity** تمام تغییرات داده در داخل تراکنش می بایست با موفقیت انجام شود یا اینکه هیچ یک از تغییرات انجام نشود (همه یا هیچ کدام). به عبارتی عملیات دستکاری داده ها در تراکنش یک واحد به حساب می آیند و غیر قابل تجزیه هستند. یا همه با هم با موفقیت اجرا میشن یا اینکه هیچ کدام اجرا نمی شوند.

**Consistency** تراکنش ها، سازگاری و جامعیت داده های پایگاه داده (database) را حفظ می کنند به بیان دیگر، تراکنش، پایگاه داده را از یک حالت سازگار (consistent) به حالت سازگار دیگری تبدیل می کند. به این معنا که در صورت roll back شدن تراکنش (با شکست مواجه شدن یکی از دستورات) پایگاه داده به حالت (state) سازگاری قبل از اجرا شدن تراکنش بر میگرد. یا بعد از پذیرفته شدن ترکانش (commit) یک وضعیت سازگاری جدید شکل میگیرد (مطابق نمودار هایی که در مقاله قبل ارائه شد). در نتیجه هیچ گاه پایگاه داده را در حالت ناسازگار رها نخواهد کرد (یکسری دستورات با موفقیت اجرا شده و یکسری شکست خورده اند).

**Isolation** تغییرات هر تراکنش مستقل از سایر ترکانش ها می باشد.

تراکنش ها جدا از یکدیگر هستند به بیان دیگر، هر چند به طور کلی ممکن است چند تراکنش به طور همزمان اجرا شوند اما به هنگام سازی هر یک از این تراکنش ها از بقیه ی تراکنش ها پنهان نگه داشته می شود تا وقتی که تراکنش پذیرفته شود. به عبارت دیگر برای دو تراکنش متمایز T1 و T2، تراکنش T1 می تواند به هنگام سازی های T2 را ببیند (پس از پذیرفته شدن T2) یا تراکنش T2 می تواند به هنگام سازی های T1 را ببیند (پس از پذیرفته شدن T1) اما هر دو به طور همزمان نمی توانند به هنگام سازی های یکدیگر را ببینند.

**توجه**: این بحث مربوط به همزمانی (concurrency) و قفل گذاری (Locking) می شود که قصد بر این است که در سلسله مقالات بعدی آنها را معرفی کنیم.

**Durability** پس از آن که تراکنشی پذیرفته شد (committed) به هنگام سازی های آن، در پایگاه داده باقی می ماند و state جدید پایگاه داده قابل دسترسی بوده، حتی اگر سیستم اندکی بعد به دلیل مشکلات سخت افزاری و یا نرم افزاری از کار بیفتد.

در SQL Server عملا این ترمیم و بازسازی توسط checkpoint ها صورت گرفته و رویه ترمیم پایگاه داده (recovery) در هنگام startup انجام میشود.

**نمایش Tran های فعال**

|  |
| --- |
| SELECT DB\_NAME(DES.database\_id) AS DatabaseName,\*  FROM sys.sysprocesses AS S  JOIN sys.dm\_exec\_sessions AS DES WITH(NOLOCK) ON S.spid = DES.session\_id  CROSS APPLY sys.dm\_exec\_sql\_text(sql\_handle) T  WHERE open\_tran = 1 |
| **غیر فعال کردن دستورات**  KILL 52 |

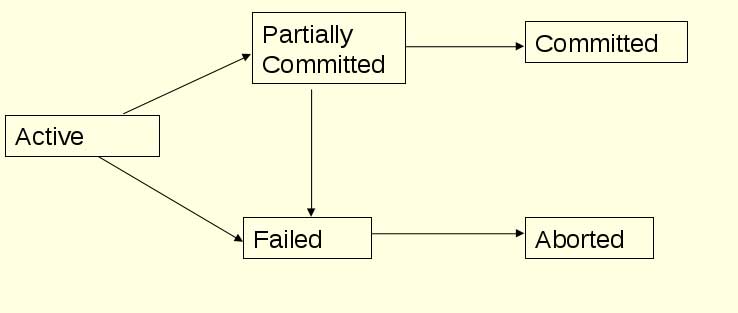
**وضعیت‌های یک تراکنش**

تراکنش‌ها در سیستم همانند یک موجودیت (entity) فعال است هستند. همانطور که می‌دانید ساده‌ترین موجودیت فعال در سیستم فرآیند‌ها (process) می‌باشند که cpu را بعنوان یک ابزار در اختیار گرفته و وظایفی را انجام می‌دهند. تراکنش نیز یک موجودیت فعال می‌باشد و همانند سایر موجودیت‌های فعال دارای وضعیت هایی (state) می‌باشند که در ادامه هریک شرح داده شده اند :

* فعال (Active) : تراکنشی که در حالت اجرا است در وضعیت فعال می‌باشد.
* کامیت جزئی (Partially Committed): پس از اجرای آخرین دستور تراکنش به وضعیت کامیت جزئی می‌رود.
* شکست (Failed): در این وضعیت، در روند اجرا خطایی رخ داده و اجرای ادامه تراکنش امکان پذیر نمی‌باشد.
* خاتمه (Aborted): پس از تشخیص خطا تراکنش می‌تواند به وضعیت Aborted که در انجا اجرا متوفق شده و تغییرات ROLLBACK می‌شوند.

**Committed**در این وضعیت اجرای تراکنش با موفقیت انجام شده و تراکنش پایان می‌پذیرد.

در ادامه نمودار حالت تراکنش‌ها نشاد داده شده است :

[](https://www.dotnettips.info/file/image?name=564564.jpg)

نکته ای که در اینجا لازم به ذکر است اینست که در حالت پس از حالت شکست به دو شکل امکان ادامه کار وجود دارد. در صورتی که خطای منطقی در تراکنش دیده شود که عموما توسط کاربر تشخیص داده می‌شود تراکش پس از شکست به حالت خاتمه برده می‌شود و کار تمام است. اما در برخی از شرایط خطایی سیستم توسط خود سیستم رخ می‌دهد. که در چنین حالاتی پس از شکست تراکنش مجددا تراکنش ممکن است به حالت فعال برگردانده شود و اجرای ان دوباره از ابتدای تراکنش شروع شود. به این وضعیت اصطلاحا REDO شدن تراکنش گفته می‌شود که در بخش RECOVERY و ترمیم پایگاه داده باید به آن پرداخته شود.

**اعمال زمان COMMIT:**

در زمان COMMIT (بصورت صریح و یا ضمنی)  باید اعمالی انجام شود که در اینجا به آن می‌پردازیم. اولین کاری که صورت می‌گیرد اینست که سیگنالی به DBMS ارسال می‌شود مبنی بر اینکه تراکنش با موفقیت به پایان رسیده است. پس از اینکار سیستم مدیریت پایگاه داده شروع به آزاد کردن قفل هایی می‌کند که در طول اجرای تراکنش بر روی منابع مختلف پایگاه داده زده شده است تا از تاثیر سوء تراکنش‌ها بر روی یکدیگر جلوگیری به عمل آید. علاوه بر کار ذکر شده تغییراتی که توسط تراکنش داده شده است باید پایدار و قابل رویت توسط سایر تراکنش‌ها گردد.

همانطور که در بخش ابتدایی این مطلب آموزشی اشاره کردیم COMMIT به معنی نوشته شدن تغییرات بر روی دیسک سخت نیست. سیستم مدیریت پایگاه داده تنها درخواست نوشتن داده‌ها را به سیستم مدیریت حافظه می‌دهد و نوشتن ان بر عهده مدیریت حافظه می‌باشد. سیستم مدیریت پایگاه داده باید اطلاع داشته باشد که چه تغییراتی نوشته شده است و چه تغییراتی هنوز در حافظه نوشته نشده است. بنابراین یکی دیگر از پیچیدگی‌های طراحی سیستم‌های مدیریت پایگاه داده اینست که تغییراتی را برای سایرین قابل رویت کند که هنوز در حافظه سخت نوشته نشده است.

**اعمال زمان ROLLBACK:**

در زمان ROLLBACK ناموفق بودن تراکنش باید به DBMS اطلاع داده شود. پس از انکه سیستم مدیریت پایگاه داده مطلع شد تمامی تغییرات اعمال شده تا آن لحظه را UNDO می‌کند. البته توجه داشته باشید که در این زمان همانند زمان COMMIT قفل‌ها نیز آزاد می‌شوند تا سایر تراکنش‌ها بتوانند از منابع در اختیار این تراکنش استفاده کنند و درجه همروندی پایین نیاید.

این مهم است که بدانید SQL Server هرگز COMMIT را در نظر نمی گیرد. اگر شما از SQL Server قطع شوید بدون اینکه صراحتا COMMIT کرده باشید، SQL Server یک ROLLBACKبرای تراکنش در نظر می گیرد.

اگر یک error شدید و جدی در جریان اجرای یک تراکنش اتفاق افتد، SQL Server تراکنش را بصورت خودکار roll back خواهد نمود.

QL Serverتعداد تراکنش ها را نگهداری می کند، که تعداد تراکنش های فعال برای connection جاری را بر می گرداند. شما می توانید تعداد تراکنش های جاری را بوسیله تابع @@TRANCOUNT بدست آورید.

هر بار که شما BEGIN TRANSACTIONرا صدا می زنید به مقدار @@TRANCOUNT یکی اضافه می شود. و به طور مشابه هر بار که شما COMMIT TRANSACTION را صدا می زنید SQL Server مقدار @@TRANCOUNT را یکی کاهش می دهد. تا مادامی که مقدار @@trancount به صفر برنگشته است تراکنش فعال خواهد ماند. زمانی که شما BEGIN TRANSACTION را داخل یک بلاک تراکنش صدا می زنید عملا شما یک تراکنش تو در تو (nested transaction) ایجاد کرده اید.

* متغییر سیستمی تعداد Transaction های باز مربوط به Session جاری (هر کاربر) را در نظر می گیرد. ضمنا با Rollback شدن Transaction مقدار موجود دراین متغییر صفر می شود حتی اگر چند Transaction باز داشته باشید همچنین به ازای Commit شدن هر Trnasaction یک واحد از مقدار موجود در این متغییر کسر می شود.

**توجه**

در حالی که ROLLBACK داده ها را به حالت قبل از شروع تراکنش بر میگرداند برخی از ویژگی ها مثل seed یک ستون identity به قبل بر نمی گردد reset نمی شود. بطور نمونه اگر ده سطر بعد از درج لغو شوند seed این ستون 10 واحد افزوده خواهد شد.

**مشکلات همزمانی (Concurrency Effects)**

### **Dirty Read**

زمانی روی می‌دهد که تراکنشی رکوردی را می‌خواند، که بخشی از تراکنشی است که هنوز تکمیل نشده است، اگر آن تراکنش Rollback شود اطلاعاتی از بانک اطلاعاتی دارید که هرگز روی نداده است.

 اگر سطح جداسازی تراکنش (پیش فرض) Read Committed باشد، این مشکل بوجود نمی‌آید.

### **Non-Repeatable Read**

زمانی ایجاد می‌شود که رکوردی را دو بار در یک تراکنش می‌خوانید و در این اثنا یک تراکنش مجزای دیگر داده‌ها را تغییر می‌دهد. برای پیشگیری از این مسئله باید سطح جداسازی تراکنش برابر با Repeatable Read یا Serializable باشد.

### **Phantoms**

با رکوردهای مرموزی سروکار داریم که گویی تحت تاثیر عبارات Update و Delete صادر شده قرار نگرفته اند. به طور خلاصه شخصی عبارت Insert را درست در زمانی که Update مان در حال اجرا بوده انجام داده است، و با توجه به اینکه ردیف جدیدی بوده و قفلی وجود نداشته، به خوبی انجام شده است. تنها چاره این مشکل تنظیم سطح Serializable است و در این صورت بهنگام رسانی‌های جداول نباید درون بخش Where قرار گیرد، در غیر این صورت Lock خواهند شد.

### **Lost Update**

زمانی روی می‌دهد که یک Update به طور موفقیت آمیزی در بانک اطلاعاتی نوشته می‌شود، اما به طور اتفاقی توسط تراکنش دیگری بازنویسی می‌شود. راه حل این مشکل بستگی به کد شما دارد و بایست به نحوی تشخیص دهید، بین زمانی که داده‌ها را می‌خوانید و زمانی که می‌خواهید آنرا بهنگام کنید، اتصال دیگری رکورد شما را بهنگام کرده است.

**منابع قابل قفل شدن**

منبع قابل قفل شدن برای SQL Server وجود دارد و آن‌ها سلسله مراتبی را تشکیل می‌دهند. هر چه سطح قفل بالاتر باشد، Granularity کمتری دارد. در ترتیب آبشاری Granularity عبارتند از:  
  **Database**کل بانک اطلاعاتی قفل شده است، معمولاً طی تغییرات Schema بانک اطلاعاتی روی می‌دهد.  
  **Table**کل جدول قفل شده است، شامل همه اشیای مرتبط با جدول.  
  **Extent**کل Extent متشکل از هشت Page قفل شده است.  
  **Page**همه داده‌ها یا کلیدهای Index در آن Page قفل شده اند.  
  **Key**قفلی در کلید مشخصی یا مجموعه کلید هایی Index وجود دارد. ممکن است سایر کلید‌ها در همان Index Page تحت تاثیر قرار نگیرند **Row or Row Identifier RID**هر چند قفل از لحاظ فنی در Row Identifier قرار می‌گیرد ولی اساساً کل ردیف را قفل می‌کند.

**سطوح جداسازی (Isolation Level)**

**وضعیت پیش فرض Read Committed**

با Read Committed همه Shared Lock‌های ایجاد شده، به محض اینکه عبارت ایجاد کننده آنها تکمیل شود، به طور خودکار آزاد می‌شوند. به طور خلاصه قفل‌های مرتبط با عبارت Select به محض تکمیل عبارت Select آزاد می‌شوند و SQL Server منتظر پایان تراکنش نمی‌ماند. اگر تراکنش پرس و جویی را انجام می‌دهد که داده‌ها را اصلاح می‌کند Insert، Delete و Update قفل‌ها برای مدت تراکنش نگه داشته می‌شوند.

با این سطح پیش فرض، می‌توانید مطمئن شوید جامعیت کافی برای پیشگیری از Dirty Read دارید، اما همچنان Phantoms و Non-Repeatable Read می‌تواند روی دهد.

یک مثال :یک صفحه New Query باز کنید و دستور زیر را در آن کپی نمایید:

|  |
| --- |
| BEGIN TRAN  UPDATE dbo.Product SET Name = 'ss' WHERE ProductID=1  WAITFOR DELAY '00:00:20'  ROLLBACK |

یک صفحه New Query دیگر باز کنید و دستور زیر را در آن کپی نمایید:

|  |
| --- |
| SELECT \* FROM dbo.Product |

* همان طور که می بینید نتیجه خروجی دستور دوم با تاخیر نمایش داده خواهد شد. و این به خاطر تاخیر در اتمام تراکنش در دستورات اول می باشد.

**Read Uncommitted**

خطرناک‌ترین گزینه از میان تمامی گزینه‌ها است، اما بالاترین عملکرد را به لحاظ سرعت دارد. در واقع با این تنظیم سطح تجربه همه مسائل متعدد هم زمانی مانند Dirty Read امکان پذیر است. در واقع با تنظیم این سطح به SQL Server اعلام می‌کنیم هیچ قفلی را تنظیم نکرده و به هیچ قفلی اعتنا نکند، بنابراین هیچ تراکنش دیگری را مسدود نمی‌کنیم. با این اوصاف می توان گفت اطلاعاتی که در این Selectها به دست می آید به احتمال زیاد نامعتبر خواهد بود.

می‌توانید همین اثر Read Uncommitted را با اضافه کردن نکته بهینه ساز  **NOLOCK** در پرس و جو‌ها بدست آورید.

یک مثال :یک صفحه New Query باز کنید و دستور زیر را در آن کپی نمایید:

|  |
| --- |
| BEGIN TRAN  UPDATE dbo.Product SET Name = 'ss' WHERE ProductID=1  WAITFOR DELAY '00:00:20'  ROLLBACK |

یک صفحه New Query دیگر باز کنید و دستور زیر را در آن کپی نمایید:

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED  SELECT \* FROM dbo.Product |

* همان طور که می بینید در اولین اجرای دستورات دوم، اطلاعاتی را به دست آوردیم که تغییر یافته بود ولی هنوز نهایی نشده بود. در واقع Dirty Read اتفاق افتاد.

**Repeatable Read**

سطح جداسازی را تا حدودی افزایش می‌دهد و سطح اضافی محافظت همزمانی را با پیشگیری از Dirty Read و همچنین Non-Repeatable Read فراهم می‌کند.

پیشگیری از Non-Repeatable Read بسیار مفید است اما حتی نگه داشتن Shared Lock تا زمان پایان تراکنش می‌تواند دسترسی کاربران به اشیا را مسدود کند، بنابراین به بهره وری لطمه وارد می‌کند.

نکته بهینه ساز برای این سطح **REPEATEABLEREAD**  است.

یک مثال :یک صفحه New Query باز کنید و دستور زیر را در آن کپی نمایید:

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL REPEATABLE READ  BEGIN TRAN  SELECT \* FROM dbo.Product  WAITFOR DELAY '00:00:10'  SELECT \* FROM dbo.Product  ROLLBACK |

یک صفحه New Query دیگر باز کنید و دستور زیر را در آن کپی نمایید:

|  |
| --- |
| UPDATE dbo.Product SET Name = 'xx' WHERE ProductID=1 |

* همان طور که مشاهده می کنید دستور Update که اجرا شده است تاثیری در خروجی Selectهای ما نداشته و خروجی هر دو Select شبیه هم می باشد.
* اگر در دستورات اول Isolation Level را به Read Committed تغییر دهید خروجی Selectها شبیه هم نخواهد بود. به عبارتی دستور Update منتظر اتمام دستورات اول نخواهد ماند.
* **نکته :** اگر در حین تراکنش با سطح ارزیابی Reapeatable Read دستور Insert در جدول مربوطه اجرا شود و داده های جدید اضافه شود، Phantom Read اتفاق خواهد افتاد. یعنی در خروجی آخرین Select سطرهای اضافه شده نیز نمایش داده خواهد شد.

**Serializable**

این سطح ارزیابی شبیه Reapeatable Read می باشد منتها Serializable یک تضمین اضافی را نیز به ما می دهد که اجازه درج رکورد جدید در اطلاعات مربوطه را تا اتمام تراکنش نخواهد داد. بنابر این جلوی Phantom Read نیز گرفته خواهد شد. با یک مثال بهتر متوجه موضوع خواهید شد.

یک مثال :یک صفحه New Query باز کنید و دستور زیر را در آن کپی نمایید:

|  |
| --- |
| SET TRANSACTION ISOLATION LEVEL SERIALIZABLE  BEGIN TRAN  SELECT \* FROM dbo.Product  WAITFOR DELAY '00:00:10'  SELECT \* FROM dbo.Product  ROLLBACK |

یک صفحه New Query دیگر باز کنید و دستور زیر را در آن کپی نمایید:

|  |
| --- |
| INSERT INTO dbo.Product(ProductID, Name, SellStartDate, IdCategory)  VALUES(50, N'123' ,GETDATE(), 1 ) |

* همان طور که مشاهده می کنید خروجی هر دو Select شبیه هم می باشد. بنابر این دستور Insert که در دستورات دوم نوشته شده است منتظر اتمام تراکنش در دستورات اول مانده است.

**Snapshot**

جدترین سطح جداسازی است که در نسخه 2005 اضافه شد، که شبیه ترکیبی از Read Committed و Read Uncommitted است. به طور پیش فرض در دسترس نیست، در صورتی در دسترس است که گزینه ALLOW\_SNAPSHOT\_ISOLATION برای بانک اطلاعاتی فعال شده  باشد.(برای هر بانک اطلاعاتی موجود در تراکنش)

Snapshot مشابه Read Uncommitted هیچ قفلی ایجاد نمی‌کند. تفاوت اصلی آن‌ها در این است که تغییرات صورت گرفته در بانک اطلاعاتی را در زمان‌های متفاوت تشخیص می‌دهند. هر تغییر در بانک اطلاعاتی بدون توجه به زمان یا Commit شدن آن، توسط پرس و جو هایی که سطح جداسازی Read Uncommitted را اجرا می‌کنند، دیده می‌شود. با Snapshot فقط تغییراتی که قبل از شروع تراکنش، Commit شده اند، مشاهده می‌شود.

از شروع تراکنش Snapshot، تمامی داده‌ها دقیقاً مشاهده می‌شوند، زیرا در شروع تراکنش Commit شده اند.

**نکته:** در حالی که Snapshot توجهی به قفل‌ها و تنظیمات آنها ندارد، یک حالت خاص وجود دارد. چنانچه هنگام انجام Snapshot یک عمل Rollback (بازیافت) بانک اطلاعاتی در جریان باشد، تراکنش Snapshot قفل‌های خاصی را برای عمل کردن به عنوان یک مکان نگهدار  و سپس انتظار برای تکمیل Rollback تنظیم می‌کند. به محض تکمیل Rollback، قفل حذف شده و Snapshot به طور طبیعی به جلو حرکت خواهد کرد.

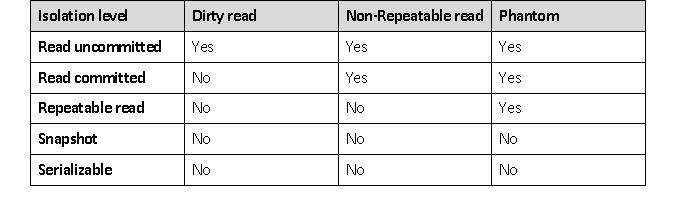
یک مثال :یک صفحه New Query باز کنید و دستور زیر را در آن کپی نمایید:

|  |
| --- |
| ALTER DATABASE [AdventureWorks2014]  SET ALLOW\_SNAPSHOT\_ISOLATION ON |
| SET TRANSACTION ISOLATION LEVEL SNAPSHOT  BEGIN TRAN  SELECT \* FROM dbo.Product WHERE Name=N'123321'  WAITFOR DELAY '00:00:30'  SELECT \* FROM dbo.Product WHERE Name=N'123321'  ROLLBACK |

یک صفحه New Query دیگر باز کنید و دستور زیر را در آن کپی نمایید:

|  |
| --- |
| INSERT INTO dbo.Product(ProductID, Name, SellStartDate, IdCategory)  VALUES(50, N'123321' ,GETDATE(), 1 ) |

* همان طور که می بینید اجرای دستورات دوم منتظر اتمام تراکنش دستورات اول نخواهد ماند ولی جالب اینجاست که خروجی هر دو Select در دستورات اول دقیقا مشابه هم می باشد.

[](https://www.dotnettips.info/file/image?name=Tbl-7e05eab803a54c6abe08a32e51fa3472.png)

* با استفاده مناسب از Isolation Level می تواند جلوی Deadlock را بگیرد. در ضمن یادتون باشه از Isolation Level نوع Uncommitted استفاده کنید تمامی مشکلات همزمانی را خواهید داشت.

**انواع تراکنش‌ها**

تراکنش‌ها انواع و اقسام مختلفی دارند که به سبب پیچیدگی بعضی از آنها به لحاظ پیاده سازی ممکن است آنها را در برخی از پایگاه داده‌ها نداشته باشیم.

1. Auto Commit Transaction
2. Explicit Transaction
3. Implicit Transaction
4. Distributed Transactions

**Auto Commit Transaction**

Auto Commit Transaction بمعنی **اعمال خودکار تراکنش** است

در حالت Auto Commit Transaction تغیرات بلافاصله بعد از اجرای دستورات، اعمال یا Commit می شوند. به همین خاطر است که این حالت را Auto Commit **Transaction** می نامند. Auto Commit Transaction حالت پیشفرض SQL Server است

در **SQL Server**  هر کوئری به تنهایی یک **Transaction** است. بعنوان مثال کوئری ای را در نظر بگیرد که قرار است 500 هزار رکورد را حذف کند. اگر در میانه ی اجرای کوئری برق برود، SQL Server تضمین می کند که بعد از بالا آمدن دوباره سرور، کل داده ها در حالت معتبر قرار بگیرند (Consistency)

**Explicit Transactions**

ساده‌ترین نوع تراکنش‌ها می‌باشند که در تمامی پایگاه‌های داده پشتیبانی می‌شوند و مثال هایی که تا کنون مثال زده شد از این دست می‌باشند.

**Implicit Transaction**

در حالت Implicit Transaction می توانیم روی عملیات های  **RollBack** و  **Commit** کنترل داشته باشیم. یک **Transaction** جدید زمانی می تواند شروع شود که یک Commit یا یک RollBack اتفاق بیافتد.

در حالت Implicit Transaction در پایان هر تراکنش باید به صراحت دستور Commit و یا RollBack ذکر شود.

**توجه**: با استفاده از دستورات زیر می توانیم این نوع **Transaction** را هر جا که نیاز شد **فعال** یا **غیرفعال** نماییم

|  |
| --- |
| SET IMPLICIT\_TRANSACTIONS ON;  SET IMPLICIT\_TRANSACTIONS OFF; |

* زمانی که Implicit Transaction غیرفعال شود، SQL Server به حالت Auto Commit Transaction برمی گردد

مثال:

|  |
| --- |
| SET IMPLICIT\_TRANSACTIONS ON;  --Transaction 1  UPDATE dbo.Product SET Name='Black 1' WHERE IdCategory=1;  COMMIT;  --Transaction 2  UPDATE dbo.Product SET Name='Black 2' WHERE IdCategory=2;  ROLLBACK;  --Transaction 3  UPDATE dbo.Product SET Name='Black 3' WHERE IdCategory=3;  COMMIT;  SELECT \* FROM dbo.Product WHERE IdCategory IN (1, 2, 3);  SET IMPLICIT\_TRANSACTIONS OFF; |

در مثال بالا، سه دستور Update همراه با Commit یا RollBack وجود دارد. توجه داشته باشید که در اینجا ما آزادیم که به ازای هر دستور، عملیات RollBack و یا Commit را انجام دهیم.

بعد از اولین Update دستور Commit آمده است یعنی ما می خواهیم تغییرات Commit شوند بنابراین در پایگاه داده، نام به مقدار "Black 1" تغییر پیدا می کند. بلافاصله بعد از آن Transaction جدید آغاز می شود.

در نهایت اولین و آخرین دستور Update اجرا می شوند و تغییرات روی پایگاه داده اعمال می شود و Update دوم انجام نشده و دور انداخته می شود.

**Distributed Transactions**

این قبیل تراکنش‌ها مربوط به پایگاه داده‌های توزیع شده می‌باشند که داده‌های آنها بر روی ماشین‌های مختلفی قرار دارند. بر روی هریک از این ماشین‌ها ممکن است DBMS‌های مختلفی نیز نصب شده باشد که هر یک سیستم مدیریتی مربطو به خود را دارند. از آنجایی که هر یک از این ماشین‌ها یک سیستم مدیریت پایگاه داده مستقل دارند بنابراین قوانین جامعیتی محلی ای را نیز باید لحاظ نمایند. البته باید توجه داشت که علاوع بر این قوانین محلی یک سری قوانین سراسری نیز وجود خواهد داشت که مربوط به کل پایگاه داده توزیع شده می‌باشد. بعنوان مثال سیستم در یکی سیستم دانشگاهی که در شهر‌های مختلفی توزیع شده است، ممکن است بخواهیم تعداد کل دانشجویان ثبت نام شده در سیستم از هزار نفر بیشتر نباشد. عموما درچنین سیستم هایی یک DBMS مدیریت کننده نیز وجود دارد که مسئول برقراری هماهنگی بین سایر DBMS‌ها و نیز اعمال اینگونه قوانین جامعیتی سراسری می‌باشد.

تراکنش‌های توزیع شده یک یا چند تراکنش جزئی تشکیل شده اند که ممکن است هریک از آنها مربوط به یکی از DBMS‌های سیستم باشد. چنین تراکنش هایی معمولا ابتدا توسط سیستم مدیریتی مرکزی دریافت می‌شوند و سپس هرکدام از پرس و جو‌های داخلی آن به DBMS مربوطه ارسال می‌گردد. اجرای هرکدام از پرس و جو‌های جزئی (که خود می‌توانند تراکنشی مستقل نیر باشند) بطور مستقل و محلی بر روی ماشین مربوطه اجرا شده و در انتها نیز نتیجه اجرا به سیستم مدیریتی باز گردانده می‌شود. سیستم مدیریتی مرکزی منتظر می‌ماند که تمامی تراکنش‌ها اعلام COMMIT کنند تا از انجام موفقیت آمیز همه انها اطمینان حاصل نماید. پس از کسب اطمینان کل تراکنش توسط این سیستم مرکزی COMMIT شده و در نتیجه تغییرات بر روی پایگاه داده توزیه شده اعمال می‌شوند. به این سیاست COMMIT کردن، کامیت دو مرحله ای یا Two-phase Commit گفته می‌شود. توجه داشته باشید که در صورتی که هریک از DBMS‌ها اعلام شکست نمایند تمامی تراکنش توزیع شده ROLLBACK می‌گردد.

tx\_begin();

execute T1 //at site D

execute T2 //at site C

Execute T3 //at site B

…

tX\_commit ();

همانطور که در مثال بالا مشاهده می‌کنید تراکنش اصلی از سه تراکنش T1، T2 و T3 تشکیل شده که مر بوط به سه سایت متفاوت می‌باشند. در زمانی تراکنش اصلی COMMIT خواهد شد که هر سه سایت اعلام موفقیت کنند.

**تراکنش‌های تو در تو (Nested Transaction)**

این نوع از تراکنش نسبت به دو نوع تراکنش قبلی پیچیدگی بیشتری به لحاظ پیاده سازی و مدیریت دارند. این گونه تراکنش‌ها عموما واحد‌های کاری بزرگی هستند که در داخل آنها درختی از تراکنش‌های تو در تو را داریم که مجموعه تمامی انها در نهایت یک کار واحد بلحاظ منطقی را انجام می‌دهند. هر یک از تراکنش‌های داخلی بعنوان یک گره در این ساختار درختی قرار دارند که می‌توانند پدر و یا فرزندانی داشته باشند.

• در تراکنش‌های تو در تو شرایطی حاکم است.

• هر گره در ساختار درختی تراکنش تنها قادر به دیدن برادر‌های خود می‌باشد. به بیان دیگر فرزندان برادران خود را نمی‌بیند و نسبت به انها هیچ اطلاعی ندارد.

• در تراکنش‌های تو در تو امکان اجرای موازی فرزندان یک گره وجود دارد.

• امکان اجرای موازی تراکنش‌ها منجر می‌شود به این که تراکنش‌های داخلی قادر به دیدن خروجی حاصل از اجرا همدیگر نباشند.

• هر تراکنشی به طور مستقل ویژگی atomicity را دارد اما پایداری (durability) و کامیت شدن آنها وابسته به پدرانشان می‌باشد.

• در صورتی که پدری تصمیم بگیرد می‌تواند تمامی زیر تراکنش هایش را خاتمه (abort) دهد.

در تراکنش‌های موازی COMMIT شدن یک گره پدر به دو صورت امکان پذیر است.

**حالت AND**: در این حالت یک تراکنش در صورتی کامیت خواهد شده که تمامی فرزندان آن با موفقیت اجرا و COMMIT شده باشند.

**حالت OR:**در این حالت اگر حتی یکی از تراکنش‌های فرزند نیز موفق به COMMIT شده باشد تراکنش پدر نیز COMMIT خواهد شد.

**تراکنش‌های چند سطحی (Multi-level Transactions) :**

این نوع نیز همانند تراکنش‌های تو در تو پیچیده است. از نظر ساختاری تراکنش‌های چند سطحی مشابه تراکنش‌های تو در تو می‌باشند ولی به لحاظ مفهومی با یکدیگر متفاوت هستند. اولین تفاوت موجود بین این دو نوع اینست که هر زیر تراکنشی قادر است خروجی زیر تراکنش‌های دیگر را ببیند. این مسئله باعث می‌شود که تنوانیم زیر تراکنش‌ها را بصورت همروند و موازی اجرا کنیم که این دومین تفاوت مفهومی بین این دو می‌باشد. هنگامی که زیر تراکنش کامل شد (COMMIT) تمامی قفل‌های مربوط به خود را آزاد می‌کند که این مورد نیز در مورد تراکنش‌های تو در تو صادق نمی‌باشد. یکی از مهمترین تفاوت‌های دیگر بین این دو نوع در اینست که در تراکنش‌های چند سطحی تمامی برگ‌ها در یک سطح از درخت قرار دارند و تنها تراکنش‌های برگ هستند که مستقیما به پایگاه داده مراجعه می‌کنند. در مورد کایت شدن نیز شروط مربوط به تراکنش‌های تو در تو در اینجا وجود ندارند و زیر تراکنش‌ها می‌توانند بدون هیچ شرطی کامیت شوند.

**تراکنش‌های زنجیره ای (Chained Transaction):**

همانطور که از نام این نوع از تراکنش‌ها پیداست، این تراکنش‌ها از زنجیره ای از زیر تراکنش‌های پی در پی تشکیل شده اند. تا زمانی که تمامی حلقه‌های این زنجیر با موفقیت اجرا نشوند سیستم به حالت سازگاری نخواهد نرفت. دراین نوع از تراکنش‌های COMMIT هر حلقه باعث پایداری شدن (durable) داده‌های در پایگاه داده خواهد شد. این مسئله ممکن است پایگاه داده را به وضعیت ناسازگاری ببرد. در هنگام کامیت شدن هر حلقه قفل‌های مربوط به آن نیز آزاد می‌شود.

حلقه‌های مختلف زنجیره تراکنشی می‌توانند با یکدیگر تبادل اطلاعات کنند. البته توجه داشته باشید که منابعی که هر کدام از آنها بر روی آن کار می‌کنند با دیگری متفاوت می‌باشد. بعنوان نمونه تراکنشی را نظر بگیرد که قصد دارد متوسط مبلغ مکالمه تلفن همراه مشترکان یک مخابرات را محاسبه کند. بدلیل تعداد بالای مشترکان ممکن است این تراکنش را در قالب یک تراکنش زنجیره ای پیاده سازی کنیم که هر حلقه از آن مسئول محاسبه این مبلغ برای ده هزار نفر از کاربران باشد. توجه داشته باشید که برای بدست آوردن مقدار متوسط نیاز داریم که هر زیر تراکنش‌ها قادر به تبادل اطلاعات باشند. از طرفی منابع مورد استفاده آنها (رکورد ها) با یکدیگر متفاوت خواهد بود و نمی‌توانند تغییرات یکدیگر را ببینند. سوالی که مطرح می‌شود اینست که مبادله اطلاعات بین حلقه‌های تراکنش به چه صورت باید انجام شود؟ در جواب این سوال باید گفت که مبادله اطلاعات بین تراکنش‌ها از طریق متغیر‌های رابطه ای که هما متغیر‌های پایگاه داده هستند انجام می‌گیرد.

**SavePoint**

در برخی شرایط ممکن است بخواهیم در هنگام ROLLBACK مجددا به ابتدای تراکنش باز نگردیم تا مجبور باشیم دوباره کار را از ابتدا از سر بگیریم. بعنوان مثال تا قسمتی از تراکنش پیش رفتیم، به خطایی بر خورد می‌کنیم و می‌خواهیم از نقطه ای خاص از تراکنش کا را از سر بگیریم. در چنین کاربرد هایی از ابزاری بنام SavePoint استفاده می‌کنم.

برای روشن‌تر شدن مفهوم SavePoint فرض کنید قصد داریم بلیطی از تهران به سیدنی رزرو کنیم. برای این منظور ابتدا عمل رزرواسیون را از تهران به دوبی انجام می‌دهیم و سپس از دوبی به سنگاپور و در نهایت از سنگاپور به سیدنی. حال در این بین می‌توانیم در نقطه تهران – دوبی SavePoint قرار دهیم تا در صورت بروز هرگونه خطا مجددا رزرواسیون را از ابتدا آغاز نکنیم. اگر در هنگام رزرو بلیط دوبی – سنگاپور خطایی بروز دهد می‌توانیم به نقطه تهران – دوبی ROLLBACK کنیم و از آنجا مسیر دیگری را انتخاب کنیم. توجه داشته باشید که ROLLBACK به SavePoint وضعیت پایگاه داده به همان نقطه بازگردانده می‌شود.

begin transaction();

s1;

sp1:= create savepoint(0);

s2;

sp2:= create savepoint(0);

if (condition)

rollback (spi);

…

…

commit

**Auto Transaction**

این قبیل تراکنش‌ها تراکنش‌های کوچکی هستند  که توسط سیستم تعریف می‌شوند. بعنوان مثال سیستم برای انجام دستورات زیر تراکنش تعریف می‌کند:

Alter table, Create, delete, insert, open, drop, fetch, grant, revoke, select, truncate table, update

یکی از علت‌های این امر اینست که در صورت بروز خطا در حین این تراکنش‌های خود کار امکان اجرای مجدد هر کدام فراهم گردد.