عنوان: **تراکنش‌ها در RavenDB**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۱۶ ۲۱:۳۵
آدرس: www.dotnettips.info
برچسب‌ها: NoSQL, RavenDB

پیش از شروع به بحث در مورد تراکنش‌ها و نحوه مدیریت آن‌ها در RavenDB، نیاز است با مفهوم ACID آشنا شویم.

ACID چیست؟

ACID از 4 قاعده تشکیل شده است (Atomic, Consistent, Isolated, and Durable) که با کنار هم قرار دادن آن‌ها یک تراکنش مفهوم پیدا می‌کند:

الف) Atomic: به معنای همه یا هیچ
اگر تراکنشی از چندین تغییر تشکیل می‌شود، همه‌ی آن‌ها باید با موفقیت انجام شوند، یا اینکه هیچکدام از تغییرات نباید فرصت اعمال نهایی را بیابند.
برای مثال انتقال مبلغ X را از یک حساب، به حسابی دیگر درنظر بگیرید. در این حالت X ریال از حساب شخص کسر و X ریال به حساب شخص دیگری واریز خواهد شد. اگر موجودی حساب شخص، دارای X ریال نباشد، نباید مبلغی از این حساب کسر شود. مرحله اول شکست خورده است؛ بنابراین کل عملیات لغو می‌شود. همچنین اگر حساب دریافت کننده بسته شده باشد نیز نباید مبلغی از حساب اول کسر گردد و در این حالت نیز کل تراکنش باید برگشت بخورد.

ب) Consistent یا یکپارچه
در اینجا consistency علاوه بر اعمال قیود، به معنای اطلاعاتی است که بلافاصله پس از پایان تراکنشی از سیستم قابل دریافت و خواندن است.

ج) Isolated: محصور شده
اگر چندین تراکنش در یک زمان با هم در حال اجرا باشند، نتیجه نهایی با حالتی که تراکنش‌ها یکی پس از دیگری اجرا می‌شوند باید یکی باشد.

د) Durable: ماندگار
اگر سیستم پایان تراکنشی را اعلام می‌کند، این مورد به معنای 100 درصد نوشته شدن اطلاعات در سخت دیسک باید باشد.

مراحل چهارگانه ACID در RavenDB به چه نحوی وجود دارند؟

RavebDB از هر دو نوع تراکنش‌های implicit و explicit پشتیبانی می‌کند. Implicit به این معنا است که در حین استفاده معمول از RavenDB (و بدون انجام تنظیمات خاصی)، به صورت خودکار مفهوم تراکنش‌ها وجود داشته و اعمال می‌شوند. برای نمونه به متد ذیل توجه نمائید:

```
public void TransferMoney(string fromAccountNumber, string toAccountNumber, decimal amount)
{
   using(var session = Store.OpenSession())
   {
         session.Advanced.UseOptimisticConcurrency = true;

         var fromAccount = session.Load<Account>("Accounts/" + fromAccountNumber);
         var toAccount = session.Load<Account>("Accounts/" + toAccountNumber);

         fromAccount.Balance -= amount;
         toAccount.Balance += amount;

         session.SaveChanges();
   }
}
```

در این متد مراحل ذیل رخ می‌دهند:
- از document store ایی که پیشتر تدارک دیده شده، جهت بازکردن یک سشن استفاده شده است.
- به سشن صراحتا عنوان شده است که از Optimistic Concurrency استفاده کند. در این حالت RavenDB اطمینان حاصل می‌کند که اکانت‌های بارگذاری شده توسط متدهای Load، تا زمان فراخوانی SaveChanges تغییر پیدا نکرده‌اند (و در غیراینصورت یک استثناء را صادر می‌کند).
- دو اکانت بر اساس Id آن‌ها از بانک اطلاعاتی واکشی می‌شوند.
- موجودی یکی تقلیل یافته و موجودی دیگر، افزایش می‌یابد.
- متد SaveChanges بر روی شیء سشن فراخوانی شده است. تا زمانیکه این متد فراخوانی نشده است، کلیه تغییرات در حافظه نگهداری می‌شوند و به سرور ارسال نخواهند شد. فراخوانی آن سبب کامل شدن تراکنش و ارسال اطلاعات به سرور می‌گردد.
بنابراین شیء سشن بیانگر یک atomic transaction ماندگار و محصور شده است (سه جزء ACID تاکنون محقق شده‌اند). محصور شده بودن آن به این معنا است که:
الف) هر تغییری که در سشن اعمال می‌شود، تا پیش از فراخوانی متد SaveChanges از دید سایر تراکنش‌ها مخفی است.
ب) اگر دو تراکنش همزمان رخ دهند، تغییرات هیچکدام بر روی دیگری اثری ندارد.

اما Consistency یا یکپارچگی در RavenDB بستگی دارد به نحوه‌ی خواندن اطلاعات و این مورد با دنیای رابطه‌ای اندکی متفاوت است که در ادامه جزئیات آن‌را بیشتر بررسی خواهیم کرد.

عاقبت یک دست شدن یا eventual consistency

درک Consistency مفهوم ACID در RavenDB بسیار مهم است و عدم آشنایی با نحوه عملکرد آن می‌تواند مشکل‌ساز شود. در دنیای بانک‌های اطلاعاتی رابطه‌ای، برنامه نویس‌ها به «immediate consistency» عادت دارند (یکپارچگی آنی). به این معنا که هرگونه تغییری در بانک اطلاعاتی، پس از پایان تراکنش، بلافاصله در اختیار کلیه خوانندگان سیستم قرار می‌گیرد. در RavenDB و خصوصا دنیای NoSQL، این یکپارچگی آنی دنیای رابطه‌ای، به «eventual consistency» تبدیل می‌شود (عاقبت یک‌دست شدن).
عاقبت یک دست شدن در RavenDB به این معنا است که اگر تغییری به یک سند اعمال گردیده و ذخیره شود؛ کوئری انجام شده بر روی این اطلاعات تغییر یافته ممکن است «stale data» باز گرداند. واژه stale در RavenDB به این معنا است که هنوز اطلاعاتی در دیتابیس موجود هستند که جهت تکمیل ایندکس‌ها پردازش نشده‌اند. به این مورد در قسمت بررسی ایندکس‌ها در RavenDB اشاره شد.
در RavenDB یک سری تردهای پشت صحنه، مدام مشغول به کار هستند و بدون کند کردن عملیات سیستم، کار ایندکس کردن اطلاعات را انجام می‌دهند. هر زمانیکه اطلاعاتی را ذخیره می‌کنیم، بلافاصله این تردها تغییرات را تشخیص داده و ایندکس‌ها را به روز رسانی می‌کنند. همچنین باید درنظر داشت که RavenDB جزو معدود بانک‌های اطلاعاتی است که خودش را بر اساس نحوه استفاده شما ایندکس می‌کند! (نمونه‌ای از آن‌را در قسمت ایندکس‌های پویای حاصل از کوئری‌های LINQ پیشتر مشاهده کرده‌اید)

نکته مهم
در RavenDB اگر از کوئری‌های LINQ استفاده کنیم، ممکن است به علت اینکه هنوز تردهای پشت صحنه‌ی ایندکس سازی اطلاعات، کارشان تمام نشده است، تمام اطلاعات یا آخرین اطلاعات را دریافت نکنیم (که به آن stale data گفته می‌شود). هر آنچه که ایندکس شده است دریافت می‌گردد (مفهوم عاقبت یک دست شدن ایندکس‌ها). اما اگر نیاز به یکپارچگی آنی داشتیم، متد Load یک سشن، مستقیما به بانک اطلاعاتی مراجعه می‌کند و اطلاعات بازگشت داده شده توسط آن هیچگاه احتمال stale بودن را ندارند.
بنابراین برای نمایش اطلاعات یا گزارشگیری، از کوئری‌های LINQ استفاده کنید. RavenDB خودش را بر اساس کوئری شما ایندکس خواهد کرد و نهایتا به کوئری‌هایی فوق العاده سریعی در طول کارکرد سیستم خواهیم رسید. اما در صفحه ویرایش اطلاعات بهتر است از متد Load استفاده گردد تا نیاز به مفهوم immediate consistency یا یکپارچگی آنی برآورده شود.

تنظیمات خاص کار با ایندکس سازها برای انتظار جهت اتمام کار آن‌ها

عنوان شد که اگر ایندکس سازهای پشت صحنه هنوز کارشان تمام نشده است، در حین کوئری گرفتن، هر آنچه که ایندکس شده بازگشت داده می‌شود.

در اینجا می‌توان به RavenDB گفت که تا چه زمانی می‌تواند یک کوئری را جهت دریافت اطلاعات نهایی به تاخیر بیندازد. برای اینکار باید اندکی کوئری‌های LINQ آن‌را سفارشی سازی کنیم:

```
RavenQueryStatistics stats;
var results = session.Query<Product>()
    .Statistics(out stats)
    .Where(x => x.Price > 10)
    .ToArray();

if (stats.IsStale)
{
    // Results are known to be stale
}
```

توسط امکانات آماری کوئری‌های LINQ در RavenDB مطابق کدهای فوق، می‌توان دریافت که آیا اطلاعات دریافت شده stale است یا خیر.
همچنین زمان انتظار تا پایان کار ایندکس ساز را نیز توسط متد Customize به نحو ذیل می‌توان تنظیم کرد:

```
RavenQueryStatistics stats;
var results = session.Query<Product>()
    .Statistics(out stats)
    .Where(x => x.Price > 10)
    .Customize(x => x.WaitForNonStaleResults(TimeSpan.FromSeconds(5)))
    .ToArray();
```

به علاوه می‌توان کلیه کوئری‌های یک documentStore را وارد به صبر کردن تا پایان کار ایندکس سازی کرد (متد Customize پیش فرضی را با WaitForNonStaleResultsAsOfLastWrite مقدار دهی و اعمال می‌کند):

```
documentStore.Conventions.DefaultQueryingConsistency = ConsistencyOptions.QueryYourWrites;
```

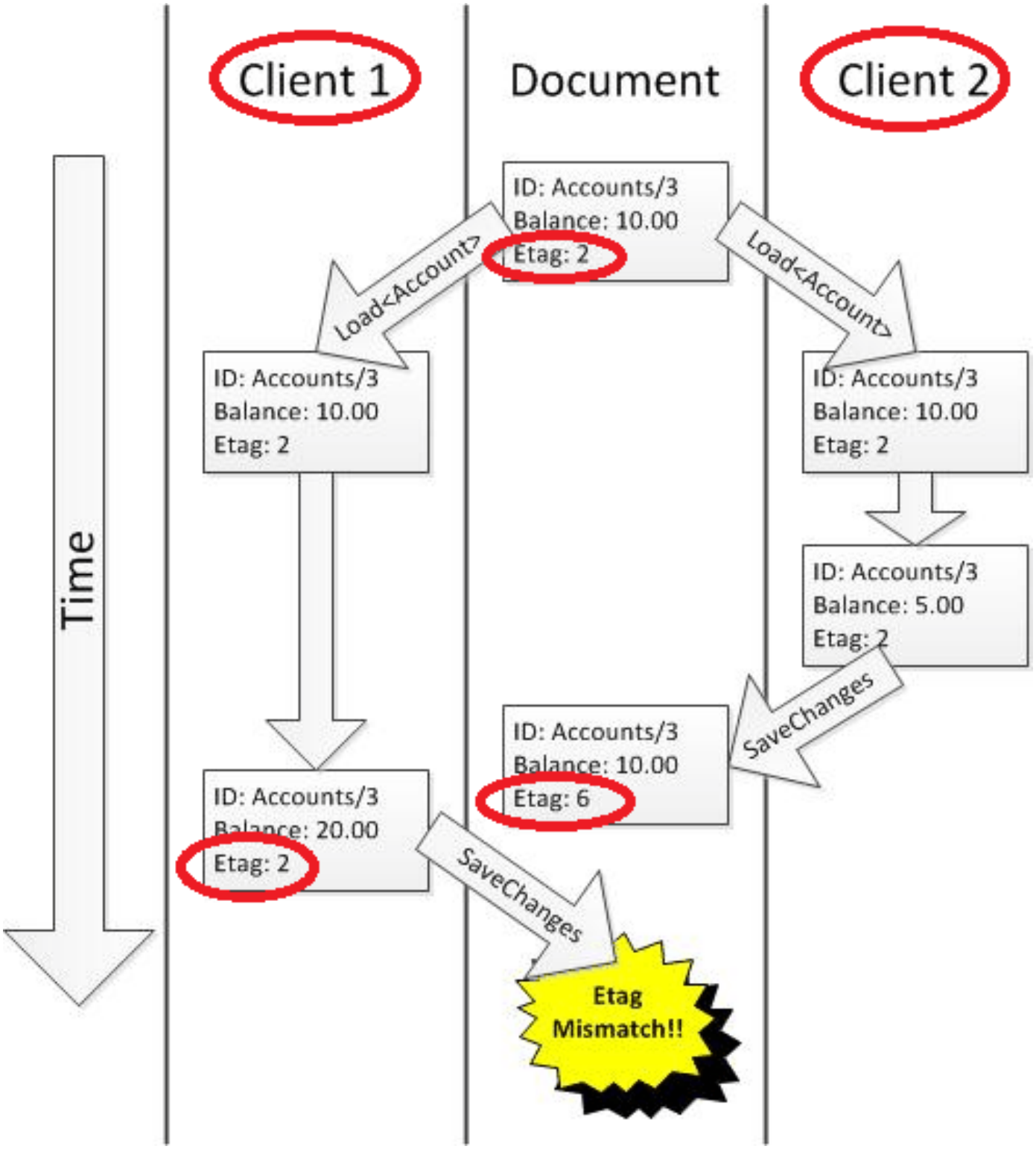
این مورد در برنامه‌های وب توصیه نمی‌شود چون کل سیستم در حین آغاز کار با آن بر اساس یک documentStore سینگلتون باید کار کند و همین مساله صبر کردن‌ها، با بالا رفتن حجم اطلاعات و تعداد کاربران، پاسخ دهی سیستم را تحت تاثیر قرار خواهد داد.
به علاوه این تنظیم خاص بر روی کوئری‌های پیشرفته Map/Reduce کار نمی‌کند. در این نوع کوئری‌های ویژه، برای صبر کردن تا پایان کار ایندکس شدن، می‌توان از روش زیر استفاده کرد:

```
while (documentStore.DatabaseCommands.GetStatistics().StaleIndexes.Length != 0)
{
    Thread.Sleep(10);
}
```

مقابله با تداخلات همزمانی

با تنظیم session.Advanced.UseOptimisticConcurrency = true، اگر سندی که در حال ویرایش است، در این حین توسط کاربر دیگری تغییر کرده باشد، استثنای ConcurrencyException صادر خواهد شد. همچنین این استثناء در صورتیکه شخصی قصد بازنویسی سند موجودی را داشته باشد نیز صادر خواهد شد (شخصی بخواهد سندی را با ID سند موجودی ذخیره کند). اگر از optimistic concurrency استفاده نشود، آخرین ترد نویسنده یا به روز کننده اطلاعات، برنده خواهد شد و اطلاعات نهایی موجود در بانک اطلاعاتی متعلق به او و حاصل بازنویسی آن ترد است.
optimistic concurrency به زبان ساده به معنای به خاطر سپردن شماره نگارش یک سند است، زمانیکه آن‌را بارگذاری می‌کنیم و سپس ارسال آن به سرور، زمانیکه قصد ذخیره آن‌را داریم. در SQL Server اینکار توسط RowVersion انجام می‌شود. در بانک‌های اطلاعاتی سندگرا چون تمایل به استفاده از HTTP در آن‌ها زیاد است (مانند RavenDB) از مکانیزمی به نام E-Tag برای این منظور کمک گرفته می‌شود. هر زمانیکه تغییری به یک سند اعمال می‌شود، E-Tag آن به صورت خودکار افزایش خواهد یافت.
برای مثال فرض کنید کاربری سندی را با E-Tag مساوی 2 بارگذاری کرده است. قبل از اینکه این کاربر در صفحه ویرایش اطلاعات کارش با این سند خاتمه یابد، کاربر دیگری در شبکه، این سند را ویرایش کرده است و اکنون E-Tag آن مثلا مساوی 6 است. در این زمان اگر کاربر یک سعی به ذخیره سازی اطلاعات نماید، چون E-Tag سند او با E-Tag سند موجود در سرور دیگر یکی نیست، با

استثنای ConcurrencyException متوقف خواهد شد.



مشکل! در برنامه‌های بدون حالت وب، چون پس از نمایش صفحه ویرایش اطلاعات، سشن RavenDB نیز بلافاصله Dispose خواهد شد، این E-Tag را از دست خواهیم داد. همچنین باید دقت داشت که سشن RavenDB به هیچ عنوان نباید در طول عمر یک برنامه باز نگهداشته شود و برای طول عمری کوتاه طراحی شده است. راه حلی که برای آن درنظر گرفته شده است، ذخیره سازی این E-Tag در بار اول دریافت آن از سشن می‌باشد. برای این منظور تنها کافی است خاصیتی را به نام Etag با ویژگی JsonIgnore (که

سبب عدم ذخیره سازی آن در بانک اطلاعاتی خواهد شد) تعریف کنیم:

```
public class Person
{
    public string Id { get; set; }

    [JsonIgnore]
    public Guid? Etag { get; set; }

    public string Name { get; set; }
}
```

اکنون زمانیکه سندی را از بانک اطلاعاتی دریافت می‌کنیم، با استفاده از متد session.Advanced.GetEtagFor، می‌توان این Etag واقعی را دریافت کرد و ذخیره نمود:

```
public Person Get(string id)
{
    var person = session.Load<Person>(id);
    person.Etag = session.Advanced.GetEtagFor(person);
    return person;
}
```

و برای استفاده از آن ابتدا باید UseOptimisticConcurrency به true تنظیم شده و سپس در متد Store این Etag دریافتی از سرور را مشخص نمائیم:

```
public void Update(Person person)
{
    session.Advanced.UseOptimisticConcurrency = true;
    session.Store(person, person.Etag, person.Id);
    session.SaveChanges();
    person.Etag = session.Advanced.GetEtagFor(person);
}
```

تراکنش‌های صریح

همانطور که عنوان شد، به صورت ضمنی کلیه سشن‌ها، یک واحد کار را تشکیل داده و با پایان آن‌ها، تراکنش خاتمه می‌یابد. اگر به هر علتی قصد تغییر این رفتار ضمنی پیش فرض را دارید، امکان تعریف صریح تراکنش‌های نیز وجود دارد:

```
using (var transaction = new TransactionScope())
{
   using (var session1 = store.OpenSession())
   {
     session1.Store(new Account());
     session1.SaveChanges();
   }

   using (var session2 = store.OpenSession())
   {
     session2.Store(new Account());
     session2.SaveChanges();
   }

   transaction.Complete();
}
```

باید دقت داشت که پایان یک تراکنش، یک non-blocking asynchronous call است و مباحث stale data که پیشتر در مورد آن بحث شد، برقرار هستند.