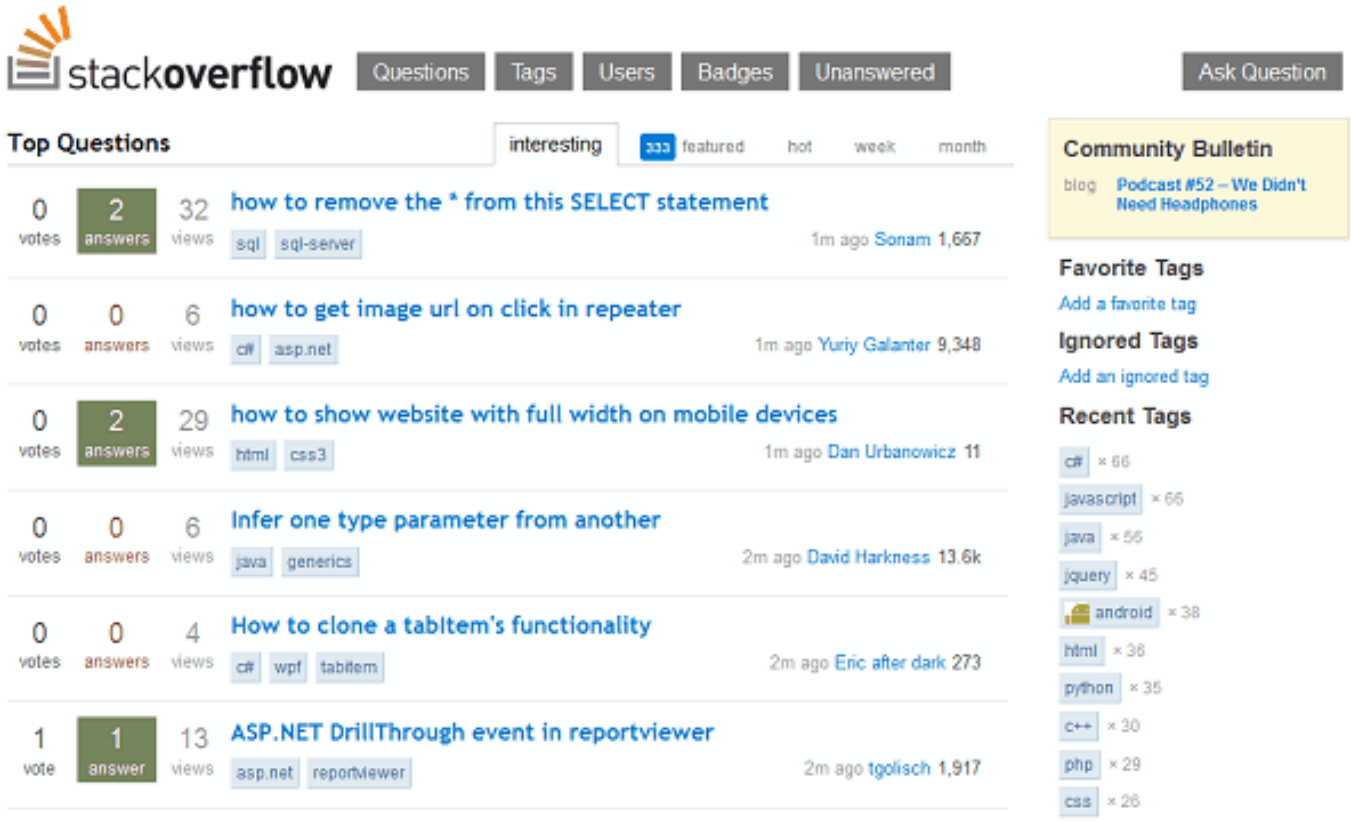
عنوان: مدل سازی داده‌ها در RavenDB
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۱۴ ۱۴:۴۱
آدرس: www.dotnettips.info
برچسب‌ها: NoSQL, RavenDB

در مطلب جاری، به صورت اختصاصی، مبحث مدل سازی اطلاعات و رسیدن به مدل ذهنی مرسوم در طراحی‌های NoSQL سندگرا را در مقایسه با دنیای Relational، بررسی خواهیم کرد.

تفاوت‌های دوره ما با زمانیکه بانک‌های اطلاعاتی رابطه‌ای پدیدار شدند

- دنیای بانک‌های اطلاعاتی رابطه‌ای برای Write بهینه سازی شده‌اند؛ از این جهت که تاریخچه پیدایش آن‌ها به دهه 70 میلادی بر می‌گردد، زمانیکه برای تهیه سخت دیسک‌ها باید هزینه‌های گزافی پرداخت می‌شد. به همین جهت الگوریتم‌ها و روش‌های بسیاری در آن دوره ابداع شدند تا ذخیره سازی اطلاعات، حجم کمتری را به خود اختصاص دهند. اینجا است که مباحثی مانند Normalization بوجود آمدند تا تضمین شود که داده‌ها تنها یکبار ذخیره شده و دوبار در جاهای مختلفی ذخیره نگردند. جهت اطلاع در سال 1980 میلادی، یک سخت دیسک 10 مگابایتی حدود 4000 دلار قیمت داشته است.
- تفاوت مهم دیگر دوره ما با دهه‌های 70 و 80 میلادی، پدیدار شدن UI و روابط کاربری بسیار پیچیده، در مقایسه با برنامه‌های خط فرمان یا حداکثر فرم‌های بسیار ساده ورود اطلاعات در آن زمان است. برای مثال در دهه 70 میلادی تصور UI ایی مانند صفحه ابتدایی سایت Stack overflow احتمالا به ذهن هم خطور نمی‌کرده است.



تهیه چنین UI ایی نه تنها از لحاظ طراحی، بلکه از لحاظ تامین داده‌ها از جداول مختلف نیز بسیار پیچیده است. برای مثال برای رندر صفحه اول سایت استک اورفلو ابتدا باید تعدادی سؤال از جدول سؤالات واکشی شوند. در اینجا در ذیل هر سؤال نام شخص مرتبط را هم مشاهده می‌کنید. بنابراین اطلاعات نام او، از جدول کاربران نیز باید دریافت گردد. یا در اینجا تعداد رای‌های هر سؤال را نیز مشاهده می‌کنید که به طور قطع اطلاعات آن در جدول دیگری نگه داری می‌شود. در گوشه‌ای از صفحه،

برچسب‌های مورد علاقه و در ذیل هر سؤال، برچسب‌های اختصاصی هر مطلب نمایش داده شده‌اند. تگ‌ها نیز در جدولی جداگانه قرار دارند. تمام این قسمت‌های مختلف، نیاز به واکشی و رندر حجم بالایی از اطلاعات را دارند.
- تعداد کاربران برنامه‌ها در دهه‌های 70 و 80 میلادی نیز با دوره ما متفاوت بوده‌اند. اغلب برنامه‌های آن دوران تک کاربره طراحی می‌شدند؛ با بانک‌های اطلاعاتی که صرفا جهت کار بر روی یک سیستم طراحی شده بودند. اما برای نمونه سایت استک اور فلویی که مثال زده شده، توسط هزاران و یا شاید میلیون‌ها نفر مورد استفاده قرار می‌گیرد؛ با توزیع و تقسیم اطلاعات آن بر روی سرورها مختلف.

معرفی مفهوم Unit of change

همین پیچیدگی‌ها سبب شدند تا جهت ساده‌سازی حل اینگونه مسایل، حرکتی به سمت دنیای NoSQL شروع شود. ایده اصلی مدل سازی داده‌ها در اینجا کم کردن تعداد اعمالی است که باید جهت رسیدن به یک نتیجه واحد انجام داد. اگر قرار است یک سؤال به همراه تگ‌ها، اطلاعات کاربر، رای‌ها و غیره واکشی شوند، چرا باید تعداد اعمال قابل توجهی جهت مراجعه به جداول مختلف مرتبط صورت گیرد؟ چرا تمام این اطلاعات را یکجا نداشته باشیم تا بتوان همگی را در طی یک واکشی به دست آورد و به این ترتیب دیگر نیازی نباشد انواع و اقسام JOINها را به چند ده جدول موجود نوشت؟
اینجا است که مفهومی به نام Unit of change مطرح می‌شود. در هر واحد تغییر، کلیه اطلاعات مورد نیاز برای رندر یک شیء قرار می‌گیرند. برای مثال اگر قرار است با شیء محصول کار کنیم، تمام اطلاعات مورد نیاز آن‌را اعم از گروه‌ها، نوع‌ها، رنگ‌ها و غیره را در طی یک سند بانک اطلاعاتی NoSQL سندگرا، ذخیره می‌کنیم.

محدوده‌های تراکنشی یا Transactional boundaries

محدوده‌های تراکنشی در Domain driven design به Aggregate root نیز معروف است. هر محدود تراکنشی حاوی یک Unit of change قرار گرفته داخل یک سند است. ابتدا بررسی می‌کنیم که در یک Read به چه نوع اطلاعاتی نیاز داریم و سپس کل اطلاعات مورد نیاز را بدون نوشتن JOIN ایی از جداول دیگر، داخل یک سند قرار می‌دهیم.
هر محدوده تراکنشی می‌تواند به محدوده تراکنشی دیگری نیز ارجاع داده باشد. برای مثال در RavenDB شماره‌های اسناد، یک سری رشته هستند؛ برخلاف بانک‌های اطلاعاتی رابطه‌ای که بیشتر از اعداد برای مشخص سازی Id استفاده می‌کنند. در این حالت برای ارجاع به یک کاربر فقط کافی است برای مثال مقدار خاصیت کاربر یک سند به "users/1" تنظیم شود. "users/1" نیز یک Id تعریف شده در RavenDB است.
مزیت این روش، سرعت واکشی بسیار بالای دریافت اطلاعات آن است؛ دیگر در اینجا نیازی به JOINهای سنگین به جداول دیگر برای تامین اطلاعات مورد نیاز نیست و همچنین در ساختارهای پیچیده‌تری مانند ساختارهای تو در تو، دیگر نیازی به تهیه کوئری‌های بازگشتی و استفاده از روش‌های پیچیده مرتبط با آن‌ها نیز وجود ندارد و کلیه اطلاعات مورد نظر، به شکل یک شیء JSON داخل یک سند حاضر و آماده برای واکشی در طی یک Read هستند.
به این ترتیب می‌توان به سیستم‌های مقیاس پذیری رسید. سیستم‌هایی که با بالا رفتن حجم اطلاعات در حین واکشی‌های داده‌های مورد نیاز، کند نبوده و بسیار سریع پاسخ می‌دهند.

Denormalization داده‌ها

اینجا است که احتمالا ذهن رابطه‌ای تربیت شده‌ی شما شروع به واکنش می‌کند! برای مثال اگر نام یک محصول تغییر کرد، چطور؟ اگر آدرس یک مشتری نیاز به ویرایش داشت، چطور؟ چگونه یکپارچگی اطلاعاتی که اکنون به ازای هر سند پراکنده شده‌است، مدیریت می‌شود؟
زمانیکه به این نوع سؤالات رسیده‌ایم، یعنی Denormalization رخ داده است. در اینجا سندهایی را داریم که کلیه اطلاعات مورد نیاز خود را یکجا دارند. به این مساله از منظر نگاه به داده‌ها در طی زمان نیز می‌توان پرداخت. به این معنا که صحیح است که آدرس مشتری خاصی امروز تغییر کرده است، اما زمانیکه سندی برای او در سال قبل صادر شده است، واقعا آدرس آن مشتری که سفارشی برایش ارسال شده، دقیقا همان چیزی بوده است که در سند مرتبط، ثبت شده و موجود می‌باشد. بنابراین سند قبلی با اطلاعات قبلی مشتری در سیستم موجود خواهد بود و اگر سند جدیدی صادر شد، این سند بدیهی است که از اطلاعات امروز مشتری استفاده می‌کند.

ملاحظات اندازه‌های داده‌ها

زمانیکه سندها بسیار بزرگ می‌شوند چه رخ خواهد داد؟ از لحاظ اندازه داده‌ها سه نوع سند را می‌توان متصور بود:
الف) سندهای محدود، مانند اغلب اطلاعاتی که تعداد فیلدهای مشخصی دارند با تعداد اشیاء مشخصی.
ب) سندهای نامحدود اما با محدودیت طبیعی. برای مثال اطلاعات فرزندان یک شخص را درنظر بگیرید. هرچند این اطلاعات نامحدود هستند، اما به صورت طبیعی می‌توان فرض کرد که سقف بالایی آن عموما به 20 نمی‌رسد!
ج) سندهای نامحدود، مانند سندهایی که آرایه‌ای از اطلاعات را ذخیره می‌کنند. برای مثال در یک سایت فروشگاه، اطلاعات فروش یک گروه از اجناس خاص را درنظر بگیرید که عموما نامحدود است. اینجا است که باید به اندازه اسناد نیز دقت داشت. برای مدیریت این مساله حداقل از دو روش استفاده می‌شود:
- محدود کردن تعداد اشیاء. برای مثال در هر سند حداکثر 100 اطلاعات فروش یک محصول بیشتر ثبت نشود. زمانیکه به این حد رسیدیم، یک سند جدید ایجاد شده و Id سند قبلی مثلا "products/1" در سند دوم ذکر خواهد شد.
- محدود کردن تعداد اطلاعات ذخیره شده بر اساس زمان
RavenDB برای مدیریت این مساله، مفهوم Includes را معرفی کرده است. در اینجا با استفاده از متد الحاقی Include، کار زنجیر کردن سندهای مرتبط صورت خواهد گرفت.

یک مثال عملی: مدل سازی داده‌های یک بلاگ در RavenDB

پس از این بحث مقدماتی که جهت معرفی ذهنیت مدل سازی داده‌ها در دنیای غیر رابطه‌ای NoSQL ضروری بود، در ادامه قصد داریم مدل‌های داده‌های یک بلاگ را سازگار با ساختار بانک اطلاعاتی NoSQL سندگرای RavenDB طراحی کنیم.
در یک بلاگ، تعدادی مطلب، نظر، برچسب (گروه‌های مطالب) و امثال آن وجود دارند. اگر بخواهیم این اطلاعات را به صورت رابطه‌ای مدل کنیم، به ازای هر کدام از این موجودیت‌ها یک جدول نیاز خواهد بود و برای رندر صفحه اصلی بلاگ، چندین و چند کوئری برای نمایش اطلاعات مطالب، نویسنده(ها)، برچسب‌ها و غیره باید به بانک اطلاعاتی ارسال گردد، که تعدادی از آن‌ها مستقیما بر روی یک جدول اجرا می‌شوند و تعدادی دیگر نیاز به JOIN دارند.
مشکلاتی که روش رابطه‌ای دارد:
- تعداد اعمالی که باید برای نمایش صفحه اول سایت صورت گیرد، بسیار زیاد است و این مساله با تعداد بالای کاربران از دید مقیاس پذیری سیستم مشکل ساز است.
- داده‌های مرتبط در جداول مختلفی پراکنده‌اند.
- این سیستم برای Write بهینه سازی شده است و نه برای Read. (همان بحث گران بودن سخت دیسک‌ها در دهه‌های قبل که در ابتدای بحث به آن اشاره شد)

مدل سازی سازگار با دنیای NoSQL یک بلاگ

در اینجا چند کلاس مقدماتی را مشاهده می‌کنید که تعریف آن‌ها به همین نحو صحیح است و نیاز به جزئیات و یا روابط بیشتری ندارند.

```
namespace RavenDBSample01.BlogModels
{
    public class BlogConfig
    {
        public string Id { set; get; }
        public string Title { set; get; }
        public string Description { set; get; }
        // ... more items here
    }

    public class User
    {
        public string Id { set; get; }
        public string FullName { set; get; }
        public string Email { set; get; }
        // ... more items here
    }
}
```

اما کلاس مطالب بلاگ را به چه صورتی طراحی کنیم؟ هر مطلب، دارای تعدادی نظر خواهد بود. اینجا است که بحث unit of change مطرح می‌شود و درج اطلاعاتی که در طی یک read نیاز است از بانک اطلاعاتی جهت رندر UI واکشی شوند. به این ترتیب به این نتیجه می‌رسیم که بهتر است کلیه کامنت‌های یک مطلب را داخل همان شیء مطلب مرتبط قرار دهیم. از این جهت که یک نظر، خارج از یک مطلب بلاگ دارای مفهوم نیست.
اما این طراحی نیز یک مشکل دارد. درست است که ساختار یک صفحه مطلب، از مطالب وبلاگ به همین نحوی است که توضیح داده شد؛ اما در صفحه اول سایت، هیچگاه کامنت‌های مطالب درج نمی‌شوند. بنابراین نیازی نیست تا تمام کامنت‌ها را داخل یک مطلب ذخیره کرد. به این ترتیب برای نمایش صفحه اول سایت، حجم کمتری از اطلاعات واکشی خواهند شد.

```
public class Post
    {
        public string Id { set; get; }
        public string Title { set; get; }
        public string Body { set; get; }

        public ICollection<string> Tags { set; get; }

        public string AuthorId { set; get; }

        public string PostCommentsId { set; get; }
        public int CommentsCount { set; get; }
    }

    public class Comment
    {
        public string Id { set; get; }
        public string Body { set; get; }
        public string AuthorName { set; get; }
        public DateTime CreatedAt { set; get; }
    }

    public class PostComments
    {
        public List<Comment> Comments { set; get; }
        public string LastCommentId { set; get; }
    }
```

در اینجا ساختار Post و Commentهای بلاگ را مشاهده می‌کنید. جایی که ذخیره سازی اصلی کامنت‌ها صورت می‌گیرد در شیء PostComments است. یعنی PostCommentsId شیء Post به یک وهله از شیء PostComments که حاوی کلیه کامنت‌های آن مطلب است، اشاره می‌کند.
به این ترتیب برای نمایش صفحه اول سایت، فقط یک کوئری صادر می‌شود. برای نمایش یک مطلب و کلیه کامنت‌های متناظر با آن دو کوئری صادر خواهند شد.

بنابراین همانطور که مشاهده می‌کنید، در دنیای NoSQL، طراحی مدل‌های داده‌ای بر اساس «سناریوهای Read» صورت می‌گیرد و نه صرفا طراحی یک مدل رابطه‌ای بهینه سازی شده برای حالت Write.

سورس کامل ASP.NET MVC این بلاگ‌را که « راکن بلاگ » نام دارد، از GitHub نویسندگان اصلی RavenDB می‌توانید دریافت کنید.