

به شما خواننده گرامی پیشنهاد می‌کنم مطلب قبلی "[آشنایی با JSON؛ ساده - خوانا - کم حجم](#)" که پیش درآمدی بر این موضوع است را مطالعه کنید.

[NoSQL](#) یک مفهوم عام است و تعریف ساده آن "پایگاه داده بدون SQL است". به این معنی که در آن خبری از جدول ها، روابط بین آن‌ها و ... نیست!

اما چرا باید با وجود اینکه SQL به اغلب نیازهای ما پاسخ داده است، باید سراغ تکنولوژی‌های دیگر رفت؟

وقتی نگاهی به لیست شرکت‌های بزرگی می‌اندازیم که جز مشتریان پر و پا قرص NoSQL هستند ([+](#) و [+](#))، تعجب می‌کنیم! آیا آن‌ها از قدرت و قابلیت‌های SQL بی‌خبراند؟

پاسخ این گونه از سوال‌ها به تحلیل سیستم مربوط می‌شود. به عهده تحلیل گر است تا با توجه به اجزاء سیستم و ارتباط آن‌ها بهترین روش را برای ذخیره سازی اطلاعات انتخاب کند.

NoSQL بر اساس نحوه پیاده سازی اش دسته بندی شده است؛ که مهم‌ترین آن‌ها در زیر آمده است :
Wide Column Store

Document Store

Key Value / Tuple Store

Graph Databases

Multimodel Databases

Object Databases

برای آشنایی بهتر با هر کدام به nosql-database.org مراجعه کنید.

انتخاب روش؛ یک مثال ساده :

فرض کنید روال استخدام نیروی کار جدید در یک سازمان، از قرار زیر باشد:

ثبت مشخصات فردی

ارائه مدارک تحصیلی

شرکت در آزمون استخدامی

شرکت در مصاحبه (در صورت قبول شدن در آزمون)

شرکت در دوره آموزشی (در صورت قبول شدن در مصاحبه)

روش‌های ممکن برای نگهداری اطلاعات :

روش اول، تهیه پوشه‌هایی برای نگهداری اطلاعات مربوط به هر مرحله به صورت مجزا است.



روش دوم، تهیه یک پرونده برای هر شخص و نگهداری اسناد مربوط به شخص (در هر مرحله) است.



انتخاب روش اول امکان پذیر است، اما باعث پیچیده تر شدن سیستم و اتلاف زمان می شود که مطلوب نیست. برای پیاده سازی روش دوم، SQL پاسخ گوی نیاز پروژه نیست و با توجه به نیاز پروژه بهترین روش نگهداری اطلاعات، Document Store (نگهداری اطلاعات بر اساس ساختار اسناد) است. خوش بختانه تعداد پایگاه های داده ای که بر اساس تکنولوژی Document Store پیاده سازی شده اند، زیاد است و از قدرتمندترین آن ها می توان به [CouchDB](#)، [MongoDB](#) و [RavenDB](#) اشاره کرد. هرکدام از این انتخاب ها مزایا و معایبی دارند که باید با توجه به نیاز خود، [مقایسه ای](#) انجام داده و بهترین را انتخاب کنید. انتخاب من RavenDB بوده است و دلایل آن :

بر اساس زبان سی شارپ نوشته شده است و همچنین با LINQ خیلی خوب کار می کند.

Transaction را پشتیبانی می کند.

اساس ذخیره سازی آن JSON است.

محیط Management Studio کاربر پسندی دارد.

نقطه آغازین بحث بعد RavenDB خواهد بود که *Bryan Wheeler* (مدیر توسعه بسترهای نرم افزاری در [msn](#)) در باره آن گفته :

RavenDB just rocked my world. It's extremely approachable, even for non-database guys – it took me less than 30 minutes to get up and running

خوشحال می شوم، نظرات و تجربیات شما را در رابطه با NoSQL بدانم.

نظرات خوانندگان

نویسنده: RaminMjjz
تاریخ: ۱۸:۵۴ ۱۳۹۱/۰۴/۱۱

سلام.
ابتدا تشکر میکنم از مطلبی که دارید ارائه میدهید.
شیوه نگارشتون هم بسیار خوبه.
فقط یک پیشنهاد دارم. اونهم اینه که یک مطلب اختصاص بدهید به؛ در چه پروژه‌هایی باید از NoSQL استفاده کرد و چه پروژه‌هایی نباید.

نویسنده: mze666
تاریخ: ۱۹:۰۰ ۱۳۹۱/۰۴/۱۱

سلام - خیلی ممنون بابت مطلب خوبتون. فقط اگر براتون ممکنه یه آموزش گام به گام یا یه نمونه پروژه از RavenDB که به نظرم بهترین هستش رو بذارید. ممنون

نویسنده: مجتبی چنانی
تاریخ: ۱۹:۲۰ ۱۳۹۱/۰۴/۱۱

با سلام
من به عنوان کسی که در پروژه‌های خود از انواع ذخیره سازی‌ها بر اساس نیاز استفاده کردم(سرعت! راحتی! پلتفرم‌ها! ...) هم نظر میدم و هم پاسخ شما دوست عزیز را می‌دم.
قطعا انتخاب اینکه از چه روشی برای ذخیره سازی داده‌ها استفاده شود بسته به تیم پیاده سازکننده پروژه و نیز طراحان و ... دارد.
من با یک مثال توضیحی را خدمت شما می‌دهم.

در یک پروژه که اخیرا در حال اجرا هست(در دست من و هم تیمی‌های من) این پروژه یک پروژه بزرگ و با دیدها و اهداف وسیعی هست. ما در این برنامه هم از ادرس دهی بر اساس پوشه‌ها و دایرکتوری‌ها داده‌ها را ذخیره کردیم(اطلاعاتی مانند لینک فایل‌ها و یا تصاویر و...) و حتی در بعضی محل‌ها نیاز بود که اطلاعات یک فرد را در یک فایل xml قرار می‌دادیم و بعضی وقتها هم در پایگاه داده و هم فایل xml به این دلیل که در مورد اول تنها برنامه سمت کلاینت نیاز به این اطلاعات داشت و در آنجا پارسر قوی xml وجود داشت اما در مورد دوم ما به یک سری دیتا نیاز داشتیم که هم در سرور به آنها نیاز داریم و هم کلاینت! خب در بحث وب ما به مدیران اگر می‌خواستیم xml ارائه کنیم قطعا راه حل خوبی نبود و از سرعت و کارایی ما کم می‌کرد لذا از پایگاه داده استفاده کردم ولی برای زمانی که کاربر کلاینتی ما نیاز به اطلاعات داشت به این دلیل که بار سرور زیاد نشود از xml‌ها استفاده می‌شد که با یک لینک مستقیم می‌توانست به دست آورد(البته خود لینک همین فایل xml هم ساخته می‌شد! هیچ جا ذخیره نمی‌شد!)

عذر می‌خوام اگر بجای نویسنده پاسخ دادم البته این پاسخ من خیلی سربسته بود و انشا.. مفید بوده.

از نویسنده مطلب بابت مطلب خوبشون که کم دیدم در تارنماهای فارسی به اون بپردازن(متأسفانه بسیاری از اساتید دانشگاهی با این مفهوم حتی آشنایی ندارند با اینکه دانستن کلیت ان یک تعریف ساده است!) موفق باشید.

نویسنده: سروش ترک زاده
تاریخ: ۱۹:۲۷ ۱۳۹۱/۰۴/۱۱

از شما ممنونم به خاطر پیشنهاد خوبتون.

به نظر خودم موضوعی که شما مطرح کردید جای بحث بیشتری دارد و حتما این مورد رو برای نوشته‌های آینده مد نظر قرار خواهیم داد.

نویسنده: سروش ترک زاده
تاریخ: ۱۳۹۱/۰۴/۱۱ ۱۹:۳۰

ممنون از شما که این مثال رو مطرح کردید.
در نوشته‌های من جای یک مثال برای واضح شدن بیشتر موضوع خالی بود!

نویسنده: سروش ترک زاده
تاریخ: ۱۳۹۱/۰۴/۱۱ ۱۹:۳۶

موافقم، هیچ چیز مثل یک مثال کاربردی یادگرفتنی نیست...

نویسنده: امیرحسین جلوداری
تاریخ: ۱۳۹۱/۰۴/۱۱ ۱۹:۴۰

یه مشکلی داره RavenDb ! ... اونم اینکه مجوزش از هموناس که اگه پروژه تجاری باشه باس پولشو بدی! (اگه متن باز که باشه هیچ!)

نویسنده: mze666
تاریخ: ۱۳۹۱/۰۴/۱۱ ۱۹:۴۷

بله درسته ولی اگه بخوایم حساب کنیم ما از خیلی چیزا توی برنامه‌های تجاریمون استفاده میکنیم (Telerik, Stimulsoft, ...) ولی پولشو نمیدیم. اینم روش. (البته نمیگم کار خوبی میکنیم!)

نویسنده: RaminMjj
تاریخ: ۱۳۹۱/۰۴/۱۱ ۲۲:۱۱

با تشکر از جوابی که دادید.
ولی من میخوام مطلبی مشابه این مقاله ارائه بشه تا بیشتر با NoSQL آشنا بشیم
<http://www.dbta.com/Articles/Editorial/Trends-and-Applications/SQL-or-NoSQL-How-to-Choose-the-Right-Database-for-Your-Application-71240.aspx>

نویسنده: peyman
تاریخ: ۱۳۹۱/۰۴/۱۱ ۲۲:۲۶

فکر میکنم Neo4j هم عالی باشه ! گر چه مایکروسافت هم داره روی Trinity کار میکنه که هر دو از نوع گراف دیتابیس‌ها هستن و به نظر من کار با گراف دیتابیس‌ها خیلی زیاتر و لذتبخش‌تر هست

نویسنده: محمد
تاریخ: ۱۳۹۱/۰۴/۱۱ ۲۲:۵۷

اتلاف زمان صحیح است و نه «اتلاف زمان». اتلاف از تلف کردن میاد. ممنون به هر حال.

نویسنده: ghafoori
تاریخ: ۱۳۹۱/۰۴/۱۱ ۲۳:۱۵

اگر برنامه داخل ایران باشه اینکارو می‌کنیم اما اگر بخواهیم اون را داخل سرور امریکا یا هر دیتاسنتری که به مجوزها گیر میده برنامه را داشته باشیم باید چکار کنیم اینجا مانگو خودش را بهتر نشون میده

نویسنده: نیما
تاریخ: ۱۳۹۱/۰۴/۱۲ ۱:۴۰

سلام دوست عزیز

ممنون از مطلبتون. در نگاه اول مطلب شما اینجوری به من القا کرد که اطلاعات مثلا سریالایز بشن حالا چه بصورت json یا xml. من رو پروژه ای کار کردم که اطلاعات بصورت xml بود و فرض کنید حجم این فایل‌های xml به حدود 10 کیلو بطور میانگین میرسید. گزارشگیری از این xmlها بسیار وقت گیر بود مخصوصا که اگر قرار بود group by یا اعمال دیگری رو انجام بدیم و خیلی اوقات به timeout میخورد که با عوض کردن این شیوه و قرار دادن اطلاعات در جداول مختلف مشکلات بکلی حل شد. ممنون میشم بیشتر توضیح بدین. موفق باشید

نویسنده: رضا.ب
تاریخ: ۱۳۹۱/۰۴/۱۲ ۲:۱۴

دوست عزیز، لطفا بیشتر توضیح دهید. من چند بار کامنت‌تون رو خوندم متوجه نشدم چی میگین. ممنون.

نویسنده: رضا.ب
تاریخ: ۱۳۹۱/۰۴/۱۲ ۳:۱۱

دو سوال داشتم:
- امکان انتقال (Migrate) بین یه دیتابیس relational و nosql در عمل ممکن هست؟ (منظورم تبدیل رابطه‌ای‌ها به nosqlهاست. چون برعکسش محاله ظاهرا؟)
- نقش ORMها در برقراری ارتباط Objectی و منطق برنامه‌های شی‌گرا با این نوع دیتابیسی‌های براساس سند(بدون ساختار) کجاست؟ اصلا ORM معنی میده هنگام کار با NoSQL؟
با تشکر. ممنونم.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۴/۱۲ ۸:۴۰

- در ravendb امکان replication به sql server وجود دارد.
- یکی از اهداف مهم ORMها در دات نت، نوشتن کوئری‌های strongly typed است. در ravendb شما از روز اول با کوئری‌های strongly typed سروکار دارید. همچنین از همان ابتدای کار هم با کلاس‌های دات نت و نگاشت خودکار آنها کار می‌کنید. کلا ravendb بر مبنای معماری و همچنین توانمندی و پیشرفت‌های زبان‌های دات نت تهیه شده.

نویسنده: mze666
تاریخ: ۱۳۹۱/۰۴/۱۲ ۹:۴۴

سلام آقای نصیری - میخوامستم از شما یا آقای ترک زاده خواهش کنم که یه مورد از این‌ها (RavenDB, CouchDB, ...) که به نظر خودتون خوبه رو آموزش بدید.
یه آموزش اساسی مثل MVC یا Code First که تو چند جلسه تمام مباحثش رو گفتید.
ممنون.

نویسنده: سروش ترک زاده
تاریخ: ۱۳۹۱/۰۴/۱۲ ۱۰:۳۹

ما توی مکتب این جوری گفته بودن بهمون...
ممنون که تذکر دادین. اصلاح شد :-)

نویسنده: سروش ترک زاده
تاریخ: ۱۰:۵۶ ۱۳۹۱/۰۴/۱۲

ممنون از جناب آقای نصیری که پاسخشان در رابطه با ORM کامل و کافی بود.
اما در مورد سوال اول شما :
در بعضی موارد تبدیل پایگاه داده Table-Relational به بعضی موارد مثل Document Store کاملاً امکان پذیر است؛ اما تبدیل آن به نوع KeyValue اساساً معنی ندارد، زیرا کاربرد این دو روش کاملاً متفاوت است.
اما این نکته قابل توجه است که اگر تحلیل سیستم شما بر اساس Table-Relational انجام گرفته باشد؛ بعد از تبدیل به Document-Store، با کاهش سرعت مواجه می‌شوید.
و به نظر من زمانی باید سراغ روش‌های NoSQL رفت که ساختار Table-Relational پاسخ مناسبی برای نیاز ما نباشد.

نویسنده: سروش ترک زاده
تاریخ: ۱۱:۱۳ ۱۳۹۱/۰۴/۱۲

سلام
نظر شما تا حدودی صحیح است اما کلاس‌های دات نت مثل [XMLWriter](#) , [XDocument](#) و ... قابل مقایسه با Engine قدرتمندی که برای یک پایگاه داده نوشته می‌شود، نیستند.
همچنین یکی از نیازها که باعث می‌شود سراغ NoSQL برویم، حجم عظیم اطلاعات است.
پس هیچ نگرانی در مورد حجم اطلاعات نباید وجود داشته باشد...

نویسنده: رضا ب.
تاریخ: ۱۵:۳۷ ۱۳۹۱/۰۴/۱۲

خب یعنی برای رفتن سمت هر NoSQL باید دلیل مرجعی داشته باشیم. و بخاطر جدید بودن و استفاده سازمان‌های عظیم از آنها و یا حتی آسان‌تر بودن، دلیل نمیشود که پایگاه‌داده‌ای رابطه‌ای رو رها کنیم.
و این زمانی اتفاق می‌وفته که این 6 نوعی که ذکر کردید، رو کاملاً بشناسیم. مزایا معایب و موارد کاربرد اون رو بدونیم و با اثبات ردِ کارایی مطلوب دیتابیس‌های رابطه‌ای به انتخاب NoSQLی دست بگذاریم.
در مورد کامنتتون متوجه نشدم علت اینکه یه پایگاه داده‌ی رابطه‌ای چرا نمیتونه به جفت مقدار/کلید تبدیل بشه؟ شاید بلعکس‌اش محال باشه. مثلاً وراثت یا جدول‌های با ستون‌های پویا و ... اصلاً در پایگاه‌های رابطه‌ای بی‌معنی هستند.

نویسنده: سروش ترک زاده
تاریخ: ۱۵:۵ ۱۳۹۱/۰۴/۱۳

شاید من نتوانستم منظور خودم رو واضح بگم؛
Table-Relational و NoSQL نقطه مقابل هم نیستند و انتخاب شما بین یکی از روش‌های ذخیره کردن اطلاعات (Graph Databases , Table Relational , Object Databases ، و ...) مشابه مثال انتخاب یکی از Type هایی مثل bit ، TimeSpam ، long و ... برای ذخیره کردن یک مقدار کوچک است. درست است که همه این کارها را با string می‌توان انجام داد و لی می‌توان با انتخاب درست در سرعت و فضایی که قرار است مصرف شود، صرفه جویی کرد.

و در باره مورد بعد که مطرح کردید، شاید یک مثال ساده قضیه رو روشن‌تر کند؛ می‌شود یک عدد کوچک رو در متغیری از جنس TimeSpam ریخت، اما اگر این عدد به معنی زمان نباشد، روش ما بهینه و حتی درست نیست، اما کار انجام شده است...
در صورتی که می‌شود این مقدار را در یک متغیر از جنس int ذخیره کرد.

امیدوارم شبهه‌ای که برای شما ایجاد شده است، با ارائه یک مثال کاربردی از RavenDB که در پست بعدی خواهم گفت، برطرف شود...

نویسنده: محمد صاحب
تاریخ: ۱۲:۴۰ ۱۳۹۱/۰۴/۱۴

تا آماده شدن مثال کاربردی؛ دیدن این [پست](#) خالی از لطف نیست.

نویسنده: محسن
تاریخ: ۱۳۹۱/۱۰/۲۳ ۱۲:۱۰

سلام

از RavenDB راضی بودین؟ آیا واقعا از جستجوی Full-Text بهره مند است و تونسته Lucene رو خوب تعبیه کنه؟

نویسنده: سروش ترک زاده
تاریخ: ۱۳۹۱/۱۰/۲۵ ۱۳:۳۰

سلام

تا اندازه ای که کارکردم خوب بود، البته پیش نیومد که توی پروژه Enterprise از آن استفاده کنم و در مورد Full-Text , Lucene راستش تا حالا امتحان نکردم... شاید دوستان دیگر بتوانند راهنمایی کنند.

نویسنده: بازرگان
تاریخ: ۱۳۹۱/۱۱/۲۰ ۱۴:۴۴

گوگل پلاس و فیسبوک برای بانک اطلاعاتشون از چه شیوه هایی استفاده میکنند ؟

نویسنده: سعید
تاریخ: ۱۳۹۱/۱۱/۲۰ ۱۷:۲۱

گوگل از بانک اطلاعاتی ساخت خودش استفاده می‌کنه: [اطلاعات بیشتر](#) ، فیس بوک هم [در اینجا](#)

عنوان: RavenDB؛ تجربه متفاوت از پایگاه داده

نویسنده: سروش ترک زاده

تاریخ: ۱۹:۳۱ ۱۳۹۱/۰۴/۱۵


















آدرس: www.dotnettips.info

برچسب‌ها: C#, JSON, NoSQL, RavenDB

" به شما خواننده گرامی پیشنهاد می‌کنم [مطلب قبلی](#) را مطالعه کنید تا پیش زمینه مناسبی در باره این مطلب کسب کنید. "

ماهیت این پایگاه داده وب سرویسی مبتنی بر REST است و فرمت اطلاعاتی که از سرور دریافت می‌شود، [JSON](#) است.

گام اول: باید آخرین نسخه RavenDB را دریافت کنید. همان طور که مشاهده می‌کنید، ویرایش‌های مختلف کتابخانه‌هایی که برای نسخه Client و همچنین Server طراحی شده است، در این فایل قرار گرفته است.

Name	Date modified	Type	Size
 Backup	۲۰۱۲/۰۲/۰۶ ۰۴:۲۰ ...	File folder	
 Bundles	۲۰۱۲/۰۲/۰۶ ۰۴:۲۰ ...	File folder	
 Client	۲۰۱۲/۰۲/۰۶ ۰۴:۲۰ ...	File folder	
 Client-3.5	۲۰۱۲/۰۲/۰۶ ۰۴:۲۰ ...	File folder	
 EmbeddedClient	۲۰۱۲/۰۲/۰۶ ۰۴:۲۰ ...	File folder	
 Samples	۲۰۱۲/۰۲/۰۶ ۰۴:۲۰ ...	File folder	
 Server	۲۰۱۲/۰۲/۰۶ ۰۴:۲۰ ...	File folder	
 Silverlight	۲۰۱۲/۰۲/۰۶ ۰۴:۲۰ ...	File folder	
 Silverlight-4	۲۰۱۲/۰۲/۰۶ ۰۴:۲۰ ...	File folder	
 Smuggler	۲۰۱۲/۰۲/۰۶ ۰۴:۲۰ ...	File folder	
 Web	۲۰۱۲/۰۲/۰۶ ۰۴:۲۰ ...	File folder	
 acknowledgments	۲۰۱۲/۰۲/۰۶ ۰۴:۱۸ ...	Text Document	
 license	۲۰۱۲/۰۲/۰۶ ۰۴:۱۸ ...	Text Document	
 Raven-GetBundles	۲۰۱۲/۰۲/۰۶ ۰۴:۱۸ ...	PS1 File	
 Raven-UpdateBundles	۲۰۱۲/۰۲/۰۶ ۰۴:۱۸ ...	PS1 File	
 readme	۲۰۱۲/۰۲/۰۶ ۰۴:۱۸ ...	Text Document	
 Start	۲۰۱۲/۰۲/۰۶ ۰۴:۱۸ ...	Windows Comma...	

برای راه اندازی Server باید فایل Start را اجرا کنید، چند ثانیه بعد محیط مدیریتی آن را در مرورگر خود مشاهده می‌کنید. در بالای صفحه روی لینک Databases کلیک کنید و در صفحه باز شده گزینه New Database را انتخاب کنید. با دادن یک نام دلخواه حالا شما یک پایگاه داده ایجاد کرده اید. تا همین جا دست نگه دارید و اجازه دهید با این محیط دوست داشتنی و قابلیت‌های آن بعداً آشنا شویم.

در گام دوم به Visual Studio می‌رویم و نحوه ارتباط با پایگاه داده و استفاده از دستورات آن را فرا می‌گیریم.

گام دوم:

با یک پروژه Test شروع می‌کنیم که در هر گام تکمیل می‌شود و می‌توانید پروژه کامل را در پایان این پست دانلود کنید.

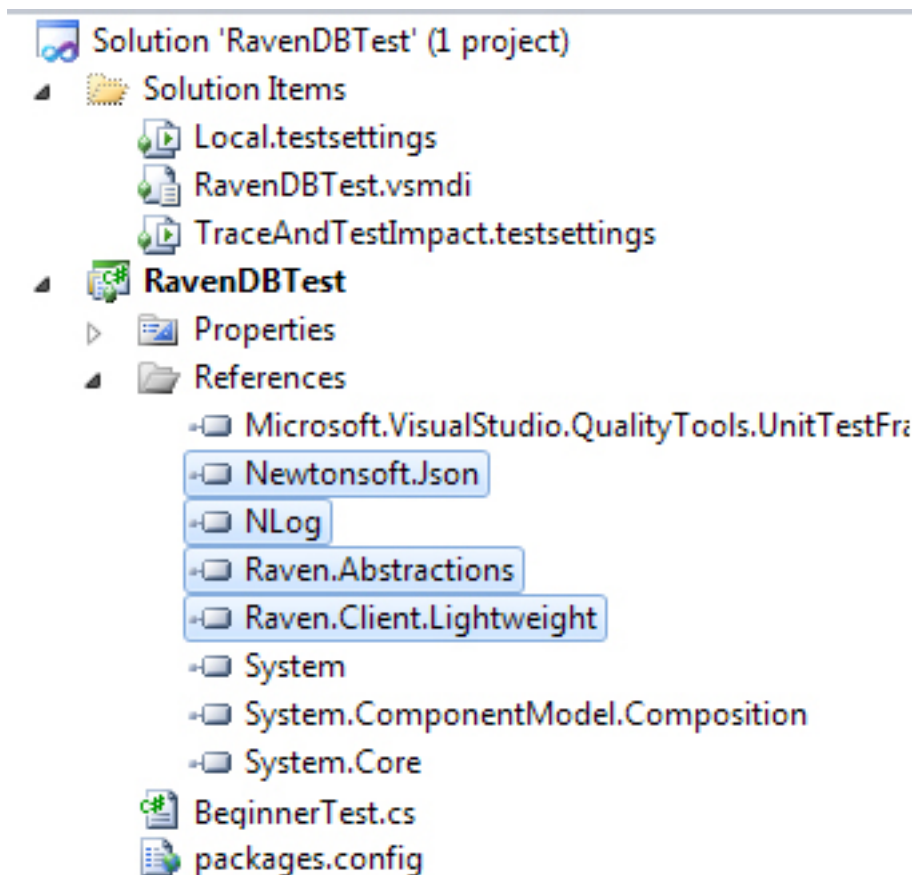
برای استفاده از کتابخانه‌های مورد نیاز دو راه وجود دارد:

استفاده از NuGet : با استفاده از دستور زیر Package مورد نیاز به پروژه شما افزوده می‌شود.

```
PM> Install-Package RavenDB -Version 1.0.919
```

اضافه کردن کتابخانه‌ها به صورت دستی : کتابخانه‌های مورد نیاز شما در همان فایل‌ها که دانلود شده بود و در پوشه Client قرار دارند.

کتابخانه‌هایی را که NuGet به پروژه من اضافه کرد، در تصویر زیر مشاهده می‌کنید :



با Newtonsoft.Json در اولین بخش بحث آشنا شدید. NLog هم یک کتابخانه قوی و مستقل برای مدیریت Log است که این پایگاه داده از آن بهره برده است.

" دلیل اینکه از پروژه تست استفاده کردم ؛ تمرکز روی کدها و مشاهده تاثیر آن‌ها ، مستقل از UI و لایه‌های دیگر نرم افزار است. بدیهی است که استفاده از آن‌ها در هر پروژه امکان پذیر است. "

برای شروع نیاز به آدرس Server و نام پایگاه داده داریم که می‌توانید در App.config به عنوان تنظیمات نرم افزار شما ذخیره شود و هنگام اجرای نرم افزار مقدار آن‌ها را خوانده و در متغیرهای readonly ذخیره شوند.

```
<appSettings>
  <add key="ServerName" value="http://SorousH-HP:8080/" />
  <add key="DatabaseName" value="TestDatabase" />
</appSettings>
```

هنگامی که صفحه Management Studio در مرورگر باز است، می‌توانید از نوار آدرس مرورگر خود آدرس سرور را به دست آورید.

```
[TestClass]
public class BeginnerTest
{
    private readonly string serverName;
    private readonly string databaseName;

    public BeginnerTest()
    {
        serverName = ConfigurationManager.AppSettings["ServerName"];
        databaseName = ConfigurationManager.AppSettings["DatabaseName"];
    }
}
```

برای برقراری ارتباط با پایگاه داده نیاز به یک شی از جنس DocumentStore و جهت انجام عملیات مختلف (ذخیره، حذف و ...) نیاز به یک شی از جنس IDocumentSession است. کد زیر، نحوه کار با آن‌ها را به شما نشان می‌دهد:

```
[TestClass]
public class BeginnerTest
{
    private readonly string serverName;
    private readonly string databaseName;

    private DocumentStore documentStore;
    private IDocumentSession session;

    public BeginnerTest()
    {
        serverName = ConfigurationManager.AppSettings["ServerName"];
        databaseName = ConfigurationManager.AppSettings["DatabaseName"];
    }

    [TestInitialize]
    public void TestStart()
    {
        documentStore = new DocumentStore { Url = serverName };
        documentStore.Initialize();
        session = documentStore.OpenSession(databaseName);
    }

    [TestCleanup]
    public void TestEnd()
    {
        session.SaveChanges();
        documentStore.Dispose();
        session.Dispose();
    }
}
```

در طراحی این پایگاه داده از الگوی Unit Of Work استفاده شده است. به این معنی که تمام تغییرات در حافظه ذخیره می‌شوند و به محض اجرای دستور session.SaveChanges() ارتباط برقرار شده و تمام تغییرات ذخیره خواهند شد.

هنگام شروع (تابع TestStart) متغیر session مقدار دهی می‌شود و در پایان کار (تابع TestEnd) تغییرات ذخیره شده و منابعی که توسط این دو شی در حافظه استفاده شده است، رها می‌شود.

البته بر مبنای طراحی شما، دستور `session.SaveChanges();` می‌تواند پس از انجام هر عملیات اجرا شود.

برای آشنا شدن با نحوه ذخیره کردن اطلاعات، به کد زیر دقت کنید:

```
class User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public int Zip { get; set; }
}

[TestMethod]
public void Insert()
{
    var user = new User
    {
        Id = 1,
        Name = "John Doe",
        Address = "no-address",
        Zip = 65826
    };
    session.Store(user);
}
```

اگر همه چیز درست پیش رفته باشد، وقتی به محیط RavenDB Studio که هنوز در مرورگر شما باز است، نگاهی می‌اندازید، یک سند جدید ایجاد شده است که با کلیک روی آن، اطلاعات آن قابل مشاهده است. لحظه‌ی لذت بخشی است... یکی از روش‌های خواندن اطلاعات هم به صورت زیر است:

```
[TestMethod]
public void Select()
{
    var user = session.Load<User>(1);
}
```

نتیجه خروجی این دستور هم یک شیء از جنس کلاس `User` است.

تا این جا، ساده‌ترین مثال‌های ممکن را مشاهده کردید و حتما در بحث بعد مثال‌های جالب‌تر و دقیق‌تری را بررسی می‌کنیم و همچنین نگاهی به جزئیات طراحی و قراردادهای از پیش تعیین شده می‌اندازیم.

" به شما پیشنهاد می‌کنم که منتظر بحث بعدی نباشید! همین حالا دست به کار شوید... "

نسخه بدون کتابخانه‌های موردنیاز (2 مگابایت) : [RavenDBTest_Small.zip](#)

نسخه کامل (15 مگابایت) : [RavenDBTest.zip](#)

نظرات خوانندگان

نویسنده: ناصر طاهری
تاریخ: ۱۷:۱۲ ۱۳۹۱/۰۴/۱۶

سلام.
مقوله‌ی جالبیه برای من.
منتظر ادامه هستم. موفق باشید.

نویسنده: حسین
تاریخ: ۱۹:۱۳ ۱۳۹۱/۰۴/۱۶

بسیار ممنون که کاربردی پیش میرین. من پروژمو با همین روش پیاده سازی می‌کنم و از اطلاعات خوبتون استفاده کردم.

نویسنده: ramín_rp
تاریخ: ۱۰:۴۳ ۱۳۹۱/۰۴/۱۷

سلام
وقتی raven db رو استارت میکنم و محیط مدیریت اون تو browser اجرا میشه برای انجام عملیاتی مثل ایجاد دیتابیس user,pass میخواد که هرچی میدم قبول نمیکنه
حتی raven رو به عنوان service هم نصب کردم ولی مشکل حل نشد
جستجو تو نت هم نتیجه نداد
مشکل از چیه؟

(مستندات رسمی raven db خوب نیست، مستندات mongodb واقعا کامل و جامع هست)

نویسنده: وحید نصیری
تاریخ: ۱۱:۳۷ ۱۳۹۱/۰۴/۱۷

توضیحات بیشتر [در اینجا](#)

By default RavenDB allow anonymous access only for read requests (HTTP GET), and since we creating data, we need to specify a username and password. You can control this by changing the AnonymousAccess setting in the server configuration file. Enter your username and password of your Windows account and a sample data will be generated for you.

نویسنده: سروش ترک زاده
تاریخ: ۲۱:۱۸ ۱۳۹۱/۰۴/۱۷

سلام
ببخشید که دیر به سوال شما پاسخ دادم...
یه راه دیگه، علاوه بر راهی که توسط جناب آقای نصیری ارائه شده است، وجود دارد.
در پوشه Server فایل Raven.Server.exe را با Notepad باز کنید، سپس مقدار تنظیمات با کلید "Raven/AnonymousAccess" را به "All" تغییر دهید. توجه کنید که به بزرگ و کوچک بودن حروف حساس است.

در ضمن RavenDB از نظر سابقه و تعداد کاربران، قابل مقایسه با پایگاه داده هایی مثل SQL نیست و حق با شماست...

نویسنده: صابر

تاریخ: ۱۱:۱۹ ۱۳۹۱/۰۴/۲۱

سلام

نمی‌دانم مشکل از چیه ؟ ولی وقتی من سعی می‌کنم که بسته‌ی RavenDB رو از طریق Nuget دریافت کنم Error زیر رو می‌ده .

```
Install-Package : The element 'metadata' in namespace
'http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd' has invalid child element
'frameworkAssemblies' in namespace
'http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd'. List of possible elements expected:
'summary' in namespace
'http://schemas.microsoft.com/packaging/2010/07/nuspec.xsd'.
At line:1 char:1
+ Install-Package RavenDB -Version 1.0.919
+ ~~~~~
+ CategoryInfo          : NotSpecified: (:) [Install-Package], InvalidOperationException
+ FullyQualifiedErrorId : NuGet.VisualStudio.Cmdlets.InstallPackageCmdlet
```

نویسنده: وحید نصیری

تاریخ: ۱۱:۵۸ ۱۳۹۱/۰۴/۲۱

به احتمال زیاد VS.NET شما دسترسی به اینترنت ندارد ([^](#) و [^](#)).

نویسنده: peyman

تاریخ: ۹:۱۸ ۱۳۹۱/۰۵/۰۸

آقا سروش کی شروع میکنی سری جدید رو منتظریم قربان !

نویسنده: سروش ترک زاده

تاریخ: ۱۸:۵۹ ۱۳۹۱/۰۵/۰۸

سلام

ببخشید که دیر شد، به احتمال زیاد پنجشنبه ادامه آموزش را روی سایت قرار خواهیم داد...

نویسنده: پژمان

تاریخ: ۲۳:۲۵ ۱۳۹۱/۰۶/۰۱

ravendb هم مثل اینکه برای جستجو از لوسین استفاده می‌کنه.

نویسنده: فرزاد

تاریخ: ۲۳:۵۰ ۱۳۹۱/۰۷/۱۰

سلام

می‌خواهم به نرم افزار تحت ویندوز بنویسم که نیاز دارم از بانک اطلاعاتی استفاده کنم از طرفی هم نمی‌خواهم با Sql server و یا access کار کنم چون نیاز به نرم افزار هایی با حجم زیاد هست که برای کاربر دردسر میشه.

می‌خواستم اگه میشه بفرمایید که به نظرتون از چی استفاده کنم بهتره؟

ممنون

نویسنده: حسین مرادی نیا

تاریخ: ۵:۴۸ ۱۳۹۱/۰۷/۱۱

نکته اینکه وقتی بانک اطلاعات Access رو استفاده کنین ، حتما نیازی نیست که Access روی کامپیوتر کاربر نصب باشه تا بتونه از برنامه شما استفاده کنه.

به هر حال میتونید از Sql Server CE استفاده کنید: <http://www.dotnettips.info/search/label/SQL%20Server%20CE>

در این پست نگاهی کلی به ویژگی‌های پایگاه‌های داده NOSql خواهیم داشت و با بررسی تاریخچه و دلیل پیدایش این سیستم‌ها آشنا خواهیم شد.

با فراگیر شدن اینترنت در سال‌های اخیر و افزایش کاربران، سیستم‌های RDBMS جوابگوی نیازهای برنامه‌نویسان در حوزه وب نبودند زیرا نیاز به نگهداری داده‌ها با حجم بالا و سرعت خواندن و نوشتن بالا از جمله نقطه ضعف سیستم‌های RDBMS می‌باشد، چرا که با افزایش شدید کاربران داده‌ها اصولاً به صورت منطقی ساختار یکدست خود را جهت نگهداری از دست می‌دهند و به این ترتیب عملیات نرمال سازی منجر به ساخت جداول زیادی می‌شود که نتیجه آن برای هر کوئری عملیات Joinهای متعدد می‌باشد که سرعت خواندن و نوشتن را به خصوص برای برنامه‌های با گستره‌ی وب پایین می‌آورد و مشکلات دیگری در سیستم‌های RDBMS که ویژگی‌های سیستم‌های NoSql مشخص کننده آن مشکلات است که در ادامه به آن می‌پردازیم.

طبق **تعریف کلی** پایگاه داده NOSql عبارت است از:

نسل بعدی پایگاه داده (نسل از بعد RDBMS) که اصولاً دارای چند ویژگی زیر باشد:

۱- داده‌ها در این سیستم به صورت رابطه‌ای (جدولی) نمی‌باشند

۲- داده‌ها به صورت توزیع شده نگهداری می‌شوند.

۳- سیستم نرم‌افزاری متن باز می‌باشد.

۴- پایگاه داده مقیاس پذیر به صورت افقی می‌باشد (در مطالب بعدی توضیح داده خواهد شد).

همان‌گونه که گفته شد این نوع پایگاه داده به منظور رفع نیازهای برنامه‌های با حجم ورود و خروج داده بسیار بالا (برنامه‌های مدرن وب فعلی) ایجاد شدند.

شروع کار پیاده‌سازی این سیستم‌ها در اوایل سال ۲۰۰۹ شکل گرفت و با سرعت زیادی رشد کرد و همچنین ویژگی‌های کلی دیگری نیز به این نوع سیستم اضافه شد.

که این ویژگی‌ها عبارتند از:

Schema-free : بدون شما!، با توجه به برنامه‌های وبی فعلی ممکن است شمای نگهداری داده‌ها (ساختار کلی) مرتباً و یا گهگاهی تغییر کند. لذا در این سیستم‌ها اصولاً داده‌ها بدون شمای اولیه طراحی و ذخیره می‌شوند. (به عنوان مثال می‌توان در یک سیستم که مشخصات کاربران وارد سیستم می‌شود برای یک کاربر یک سری اطلاعات اضافی و برای کاربری دیگر از ورود اطلاعات اضافی صرف نظر کرد، و در مقایسه با RDBMS به این ترتیب از ورود مقادیر Null و یا پیوندهای بیمورد جلوگیری کرد.

کنترل اطلاعات الزامی توسط لایه سرویس برنامه انجام می‌شود. (در زبان جاوا توسط jsr-303 و یا Bean Validation ها)

easy replication support : در این سیستم، نحوه‌ی گرفتن نسخه‌های پشتیبان و sync بودن نسخه‌های مختلف بسیار ساده و سر راست می‌باشد و سرور پایگاه داده به محض عدم توانایی خواندن و یا نوشتن از روی دیسک سراغ نسخه‌ی پشتیبان می‌رود و آن نسخه را به عنوان نسخه‌ی اصلی در نظر می‌گیرد.

Simple API : به دلیل متن‌باز بودن و فعال بودن Community این سیستم‌ها APIهای ساده و بهینه‌ای برای اکثر زبان‌های برنامه‌نویس محبوب ایجاد شده است که در پست‌های بعدی با ارائه مثال آنها را بررسی خواهیم کرد.

eventually consistent : در سیستم‌های RDBMS که داده‌ها خاصیت ACID را (در قالب Transaction) پیاده می‌کنند، در این سیستم‌های داده‌ها در وضعیت BASE قرار دارند که سرنام کلمات Basicly Available، Soft State، Eventual Consistency می‌باشد.

huge amount of data : این سیستم‌ها به منظور کار با داده‌های با حجم بالا ایجاد شده‌اند، یک تعریف کلی می‌گوید اگر مقدار داده‌های نگهداری شده در پایگاه‌های داده برنامه شما ظرفیتی کمتر از یک ترابایت داده دارد از پایگاه داده RDBMS استفاده کنید و اگر ظرفیت آن از واحد ترابایت فراتر می‌رود از سیستم‌های NOSql استفاده کنید.

به طور کلی پایگاه داده‌ای که در چارچوب موارد ذکر شده قرار گیرد را می‌توان از نوع NoSql که سرنام کلمه (Not Only SQL) می‌باشد قرار داد. تاکنون پیاده‌سازی‌های زیادی از این سیستم‌ها ایجاد شده است که رفتار و نحوه‌ی نگهداری داده‌ها (پرس و جو ها) در این سیستم‌ها با یکدیگر متفاوت می‌باشد.

جهت پیاده سازی پایگاه داده با این سیستم‌ها تا حدودی نگرش کلی به داده‌ها و نحوه‌ی چیدمان آنها تغییر می‌کند، به صورت کلی

مباحث مربوط به normalization و de-normalization و تصور داده‌ها به صورت جدولی کنار می‌رود. سیستم NoSql به جهت دسته‌بندی نحوه‌ی ذخیره‌سازی داده‌ها و ارتباط بین آنها به ۴ دسته کلی تقسیم می‌شود که معرفی کلی آن دسته‌بندی‌ها موضوع [مطلب بعدی](#) می‌باشد.

اس کیوال سرور، از سال 2005 به بعد، به صورت توکار امکان تعریف و ذخیره سازی اطلاعات [schema less](#) و یا schema free را به کمک فیلدهایی از نوع XML ارائه داده است؛ به همراه یکپارچگی آن با زبان XQuery برای تهیه کوئری‌های سریع سمت سرور. در فیلدهای XML می‌توان اطلاعات انواع و اقسام اشیاء را بدون اینکه نیازی به تعریف تک تک فیلدهای مورد نیاز، در بانک اطلاعاتی وجود داشته باشد، ذخیره کرد. یک نمونه از کاربرد چنین امکانی، نوشتن برنامه‌های «فرم ساز» است. برنامه‌هایی که کاربران آن می‌توانند فیلد اضافه و کم کرده و نهایتاً اطلاعات را ذخیره و از آن‌ها کوئری بگیرند.

خوب، این فیلد کمتر بحث شده XML، فقط در اس کیوال سرور و نگارش‌های اخیر آن وجود دارد. اگر نیاز به کار با بانک‌های اطلاعاتی سبک‌تری وجود داشت چطور؟ یک راه حل عمومی برای این مساله مراجعه به روش‌های NoSQL است. یعنی بطور کلی بانک‌های اطلاعاتی رابطه‌ای کنار گذاشته شده و به یک سکوی کاری دیگر سوئیچ کرد. در این بین، [SisoDb](#) راه حل میانه‌ای را ارائه داده است. با کمک SisoDb می‌توان اطلاعات را به صورت schema less و بدون نیاز به تعریف فیلدهای متناظر آن‌ها، در انواع و اقسام بانک‌های اطلاعاتی SQL Server با فرمت JSON ذخیره و بازیابی کرد. این انواع و اقسام، شامل SQL Server CE نیز می‌شود.

دریافت و نصب SisoDb

دریافت و نصب [SisoDb](#) بسیار ساده است. به کمک package manager و امکانات NuGet، کلمه Sisodb را جستجو کنید. در بین مداخل ظاهر شده، پروایدر مورد علاقه خود را انتخاب و نصب نمایید. برای مثال اگر قصد دارید با SQL Server CE کار کنید، SisoDb.SqlCe4 را انتخاب و یا اگر SQL Server 2008 مدنظر شما است، SisoDb.Sql2008 را انتخاب و نصب نمایید.

ثبت و بازیابی اطلاعات به کمک SisoDb

کار با SisoDb بسیار روان است. نیازی به تعاریف نگاشت‌ها و ORM خاصی نیست. یک مثال مقدماتی آن‌را در ادامه ملاحظه می‌کنید:

```
using SisoDb.Sql2008;

namespace SisoDbTests
{
    public class Customer
    {
        public int Id { get; set; }
        public int CustomerNo { get; set; }
        public string Name { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            /*var cnInfo = new SqlCe4ConnectionInfo(@"Data source=sisodb2013.sdf;");
            var db = new SqlCe4DbFactory().CreateDatabase(cnInfo);
            db.EnsureNewDatabase();*/

            var cnInfo = new Sql2008ConnectionInfo(@"Data Source=(local);Initial
            Catalog=sisodb2013;Integrated Security = true");
            var db = new Sql2008DbFactory().CreateDatabase(cnInfo);
            db.EnsureNewDatabase();

            var customer = new Customer
            {
                CustomerNo = 20,
                Name = "Vahid"
            };
            db.UseOnceTo().Insert(customer);

            using (var session = db.BeginSession())
            {
                var info = session.Query<Customer>().Where(c => c.CustomerNo == 20).FirstOrDefault();
                var info2 = session.Query<Customer>().Where(c => c.CustomerNo == 20 &&
                c.Name=="Vahid").FirstOrDefault();
            }
        }
    }
}
```

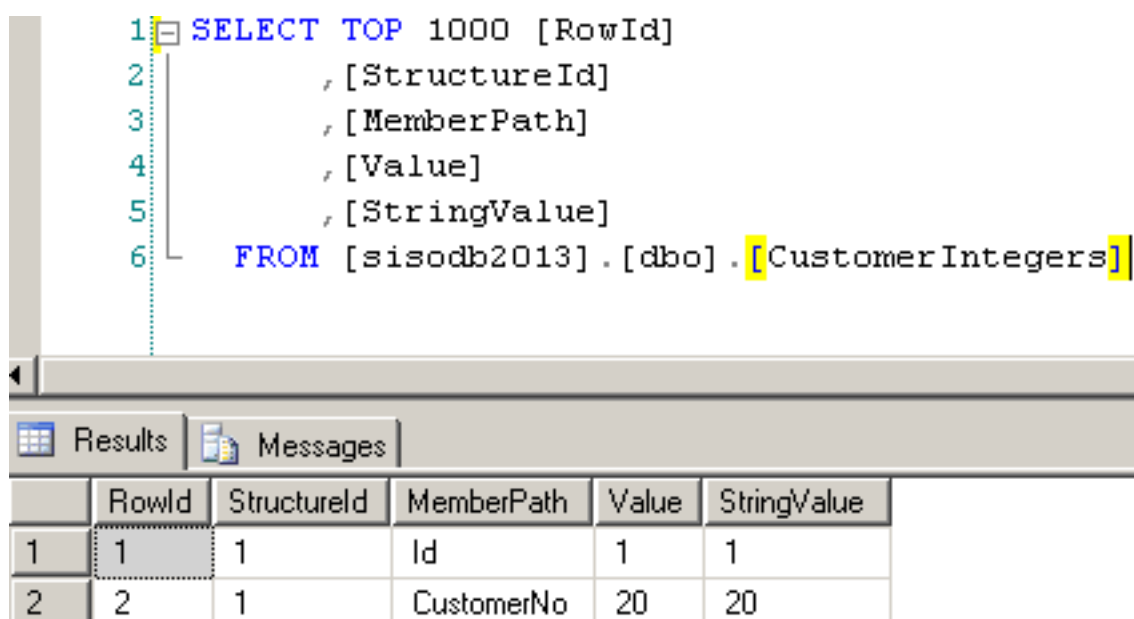
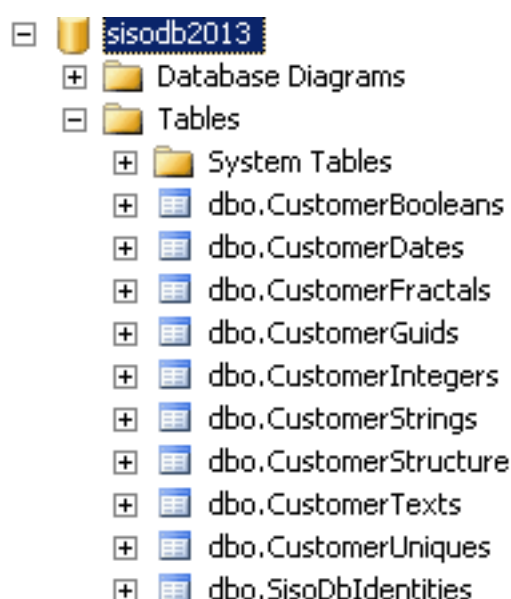


```
}
  }
}
```

در این مثال، ابتدا اتصال به بانک اطلاعاتی برقرار شده و سپس بانک اطلاعاتی جدید تهیه می‌شود. سپس یک وهله از شیء مشتری ایجاد و ذخیره می‌گردد. در ادامه دو کوئری بر روی بانک اطلاعاتی انجام شده است.

ساختار داخلی SisoDb

SisoDb به ازای هر کلاس، حداقل 9 جدول را ایجاد می‌کند. در ادامه نحوه ذخیره سازی شیء مشتری ایجاد شده و مقادیر خواص آنرا نیز مشاهده می‌نمائید:



ذخیره سازی جداگانه خواص عددی

```

1 SELECT TOP 1000 [RowId]
2     , [StructureId]
3     , [MemberPath]
4     , [Value]
5 FROM [sisodb2013].[dbo].[CustomerStrings]
    
```

	RowId	StructureId	MemberPath	Value
1	1	1	Name	Vahid

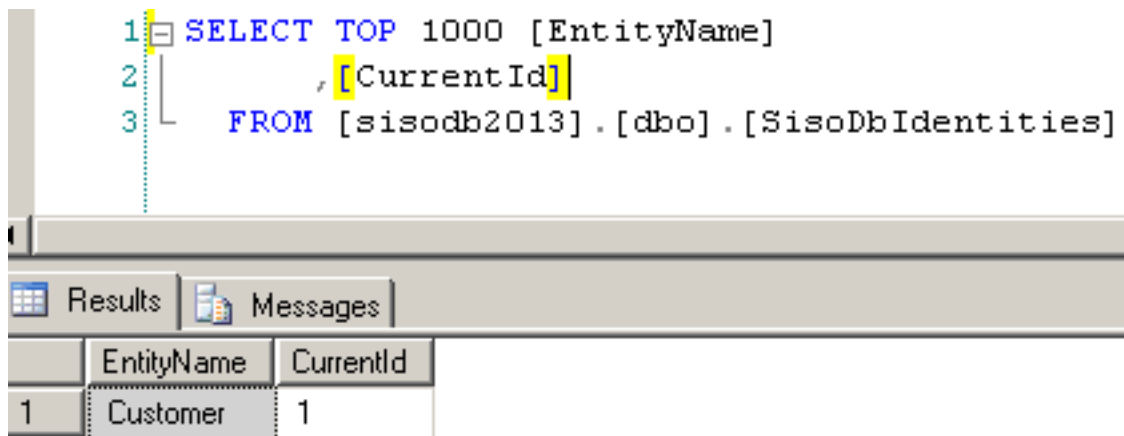
ذخیره سازی جداگانه خواص رشته‌ای

```

1 SELECT TOP 1000 [RowId]
2     , [StructureId]
3     , [Json]
4 FROM [sisodb2013].[dbo].[CustomerStructure]
    
```

	RowId	StructureId	Json
1	1	1	{"Id":1,"CustomerNo":20,"Name":"Vahid"}

ذخیره سازی کلی شیء مشتری با فرمت JSON به صورت یک رشته



همانطور که ملاحظه می‌کنید، یک جدول کلی SisoDbIdentities ایجاد شده است که اطلاعات نام اشیاء را در خود نگهداری می‌کند. سپس اطلاعات خواص اشیاء یکبار به صورت JSON ذخیره می‌شوند؛ با تمام اطلاعات تو در توی ذخیره شده در آن‌ها و همچنین یکبار هم هر خاصیت را به صورت یک رکورد جداگانه، بر اساس نوع کلی آن‌ها، در جداول رشته‌ای، عددی و امثال آن ذخیره می‌کند.

شاید بپرسید که چرا به همان فیلد رشته‌ای JSON اکتفاء نشده است؟ از این جهت که پردازشگر سمت بانک اطلاعاتی آن همانند فیلدهای XML در SQL Server و نگارش‌های مختلف آن وجود ندارد (برای مثال به کمک زبان T-SQL می‌توان از زبان XQuery در خود بانک اطلاعاتی، بدون نیاز به واکنشی کل اطلاعات در سمت کلاینت، به صورت یکپارچه استفاده کرد). به همین جهت برای کوئری گرفتن و یا تهیه ایندکس، نیاز است این موارد جداگانه ذخیره شوند. به این ترتیب زمانیکه کوئری تهیه می‌شود، برای مثال:

```
var info = session.Query<Customer>().Where(c => c.CustomerNo == 20).FirstOrDefault();
```

به کوئری زیر ترجمه می‌گردد:

```
SELECT DISTINCT TOP(1) (s.[StructureId]),
                        s.[Json]
FROM [CustomerStructure] s
LEFT JOIN [CustomerIntegers] mem0
ON mem0.[StructureId] = s.[StructureId]
AND mem0.[MemberPath] = 'CustomerNo'
WHERE (mem0.[Value] = 20);
```

و یا کوئری ذیل:

```
var info2 = session.Query<Customer>().Where(c => c.CustomerNo == 20 &&
c.Name=="Vahid").FirstOrDefault();
```

معادل زیر را خواهد داشت:

```
SELECT DISTINCT TOP(1) (s.[StructureId]),
                        s.[Json]
FROM [CustomerStructure] s
LEFT JOIN [CustomerIntegers] mem0
ON mem0.[StructureId] = s.[StructureId]
AND mem0.[MemberPath] = 'CustomerNo'
LEFT JOIN [CustomerStrings] mem1
ON mem1.[StructureId] = s.[StructureId]
AND mem1.[MemberPath] = 'Name'
WHERE ((mem0.[Value] = 20) AND (mem1.[Value] = 'Vahid'));
```

در هر دو حالت از جداول کمکی تعریف شده برای تهیه کوئری استفاده کرده و نهایتاً فیلد JSON اصلی را برای نگاشت نهایی به اشیاء تعریف شده در برنامه بازگشت می‌دهد.

در کل این هم یک روش تفکر و طراحی Schema less است که با بسیاری از بانک‌های اطلاعاتی موجود سازگاری دارد. برای مشاهده اطلاعات بیشتری در مورد جزئیات این روش می‌توان به [Wiki](#) آن مراجعه کرد.

عنوان:	NOSQL قسمت دوم
نویسنده:	حمید سامانی
تاریخ:	۹:۲۵ ۱۳۹۱/۱۱/۲۶
آدرس:	www.dotnettips.info
گروه‌ها:	NoSQL, Database, پایگاه داده, نوسی کوال, نواس کیوال, key-value, کلید-مقدار

در مطلب قبلی با تعاریف سیستم‌های NoSQL آشنا شدیم و به طور کلی ویژگی‌های یک سیستم NoSQL را بررسی کردیم.

در این مطلب دسته‌بندی کلی و نوع ساختار داده‌ای این سیستم‌ها و بررسی ساده‌ترین آنها را مرور می‌کنیم.

در حالت کلی پایگاه‌های داده NoSQL به ۴ دسته تقسیم می‌شوند که به ترتیب پیچیدگی ذخیره‌سازی داده‌ها عبارتند از:

Key/Value Store Databases

Document Databases

Graph Databases

Column Family Databases

در حالت کلی در پایگاه‌های داده NoSQL داده‌ها در قالب KEY/VALUE (کلید/مقدار) نگهداری می‌شوند، به این صورت که مقادیر توسط کلید یکتایی نگاشت شده و ذخیره می‌شوند، هر مقدار صرفاً توسط همان کلید نگاشت شده قابل بازگردانی می‌باشد و راهی جهت دریافت مقدار بدون دانستن کلید وجود ندارد. در این ساختار داده منظور از مقادیر، داده‌های اصلی برنامه هستند که نیاز به نگهداری دارند و کلیدها نیز رشته‌هایی هستند که توسط برنامه‌نویس ایجاد می‌شوند. به دلیل موجود بودن این نوع ساختار داده‌ای در اکثر کتابخانه‌های زبان‌های برنامه‌نویسی (به عنوان مثال پیاده‌سازی‌های مختلف اینترفیس Map شامل HashMap، Hashtable و موارد دیگر در کتابخانه‌های JDK) این نوع ساختار برای اکثر برنامه‌نویسان آشنا بوده و فراگیری آن نیز ساده می‌باشد.

بدیهی است که اعمال فرهنگ داده‌ای (درج، حذف، جستجو) در این سیستم به دلیل اینکه داده‌ها به صورت کلید/مقدار ذخیره می‌شوند دارای پیچیدگی زمانی $O(1)$ می‌باشد که بهینه‌ترین حالت ممکن به لحاظ طراحی می‌باشد. همان‌گونه که مستحضرید در الگوریتم‌هایی که دارای پیچیدگی زمانی با مقدار ثابت دارند کم یا زیاد بودن داده‌ها تأثیری در کارایی الگوریتم نداشته و همواره با هر حجم داده‌ای زمان ثابتی جهت پردازش نیاز می‌باشد.

:Key/Value Store Databases

این سیستم ساده‌ترین حالت از دسته‌بندی‌های NoSQL می‌باشد، به طور کلی جهت استفاده در سیستم‌هایی است که داده‌ها متمایز از یکدیگر هستند و اصولاً Availability و یا در دسترس بودن داده‌ها نسبت به سایر موارد نظیر پایداری اهمیت بالاتری دارد.

از موارد استفاده این گونه سیستم‌ها به موارد زیر می‌توان اشاره کرد:

در پلتفرم‌های اشتراک گذاری داده‌ها، هدف کلی صرفاً هندل کردن آپلود محتوای (باینری) و به صورت همزمان بروز کردن در سمت دیگر می‌باشد. (اپلیکیشنی مانند اینستاگرام را تصور کنید) در اینگونه نرم‌افزارها با تعداد بسیار زیاد کاربر و تقاضا، استفاده از این نوع پایگاه داده به مراتب کارایی و سرعت را بالاتر می‌برد. و با توجه به عدم پیش‌بینی حجم داده‌ها یکی از ویژگی‌های این نوع پایگاه داده تحت عنوان Horizontal Scaling مطرح می‌شود که در صورت Overflow شدن سرور، داده‌ها را به سمت سرور دیگری می‌توان هدایت کرد و بدون مشکل پردازش را ادامه داد، این ویژگی یک وجه تمایز کارایی این سیستم با سیستم‌های RDBMS می‌باشد که جهت مقابله با چنین وضعیتی تنها راه پیش‌رو بالا بردن امکانات سرور می‌باشد و به طور کلی داده‌ها را در یک سرور می‌توان نگهداری کرد (البته راه‌حل‌هایی نظیر پارتیشن کردن و غیره وجود دارد که به مراتب پیچیدگی و کارایی کمتری نسبت به Horizontal Scaling در پایگاه‌های داده NoSQL دارد).

برای Cache کردن صفحات بسیار کارا می‌باشد، به عنوان مثال می‌توان آدرس درخواست را به عنوان Key در نظر گرفت و مقدار آن را نیز معادل JSON نتیجه که توسط کلاینت پردازش خواهد شد قرار داد.

یک نسخه کپی شده از توئیتر که کاملاً توسط این نوع پایگاه داده پیاده شده است نیز از [این آدرس](#) قابل مشاهده است. این برنامه به زبان‌های php , ruby و java نوشته شده است و سورس نیز در مخزن github می‌جود می‌باشد. (یک نمونه پیاده سازی ایده‌آل جهت آشنایی با نحوه مدیریت داده‌ها در این نوع پایگاه داده)

از پیاده‌سازی‌های این نوع پایگاه داده به موارد زیر می‌توان اشاره کرد:

[Amazon SimpleDB](#)

[Memcached](#)

[Oracle Key/value Pair](#)

[Redis](#)

هر یک از پیاده‌سازی‌ها دارای ویژگی‌های مربوط به خود هستند به عنوان مثال Memcached داده‌ها را صرفاً در DRAM ذخیره می‌کند که نتیجه‌ی آن Volatile بودن داده‌ها می‌باشد و به هیچ وجه از این سیستم جهت نگهداری دائمی داده‌ها نباید استفاده شود. از طرف دیگر Redis داده‌ها را علاوه بر حافظه اصلی در حافظه جانبی نیز ذخیره می‌کند که نتیجه‌ی آن سرعت بالا در کنار پایداری می‌باشد.

همان‌گونه که در تعریف کلی عنوان شد یکی از ویژگی‌های این سیستم‌ها متن‌باز بودن آنها می‌باشد که نتیجه‌ی آن وجود پیاده‌سازی‌های متنوع از هر کدام می‌باشد ، لازم است قبل از انتخاب هر سیستم به خوبی با ویژگی‌های اکثر سیستم‌های محبوب و پر استفاده آشنا شویم و با توجه به نیاز سیستم را انتخاب کنیم.

در مطلب [بعدی](#) با نوع دوم یعنی Document Databases آشنا خواهیم شد.

نظرات خوانندگان

نویسنده: مجید هزاری
تاریخ: ۱۵:۴۷ ۱۳۹۱/۱۱/۲۸

عالی است.
متشکرم.

نویسنده: احمد ولی پور
تاریخ: ۱۷:۵۵ ۱۳۹۱/۱۱/۲۸

یه سوال برام پیش اومده:
با رایج شدن nosql پایگاه داده هایی مثل Oracle یا Sql Server چی میشن؟

نویسنده: مجید هزاری
تاریخ: ۱۹:۵۰ ۱۳۹۱/۱۱/۲۸

اینها تداخلی با یکدیگر ندارند.
NoSQL تنها برای رفع نیاز هایی ظهور کرده است که RelDB در آنها ضعیف بوده. همانطور که NoSQL در زمینه هایی که RelDB قوی است ضعیف عمل خواهد کرد.
(البته من کاملا مختصر گفتم)

نویسنده: حمید سامانی
تاریخ: ۲۰:۱۷ ۱۳۹۱/۱۱/۲۸

در حالت کلی هرکدام از پایگاه داده ها بسته به نیاز استفاده می شن ، توی برنامه های اینترپرایز وبی مفهوم Polyglot Persistence مطرحه (که می شه اونو نگهداری یا ذخیره سازی چند زبانی ترجمه کرد) که می گه توی یک سیستم از چندین نوع پایگاه داده می شه (باید) استفاده کرد. به عنوان مثال برای نگهداری داده هایی جهت گزارش گیری و یا ایجاد Transaction ها بهترین گزینه همان سیستم های RDBMS هستند ، در مطالب آتی به این موضوع اشاره بیشتری خواهم کرد ، مارتین فویلر در [این مطلب](#) مفهوم Polyglot Persistence را به خوبی توضیح داده اند.

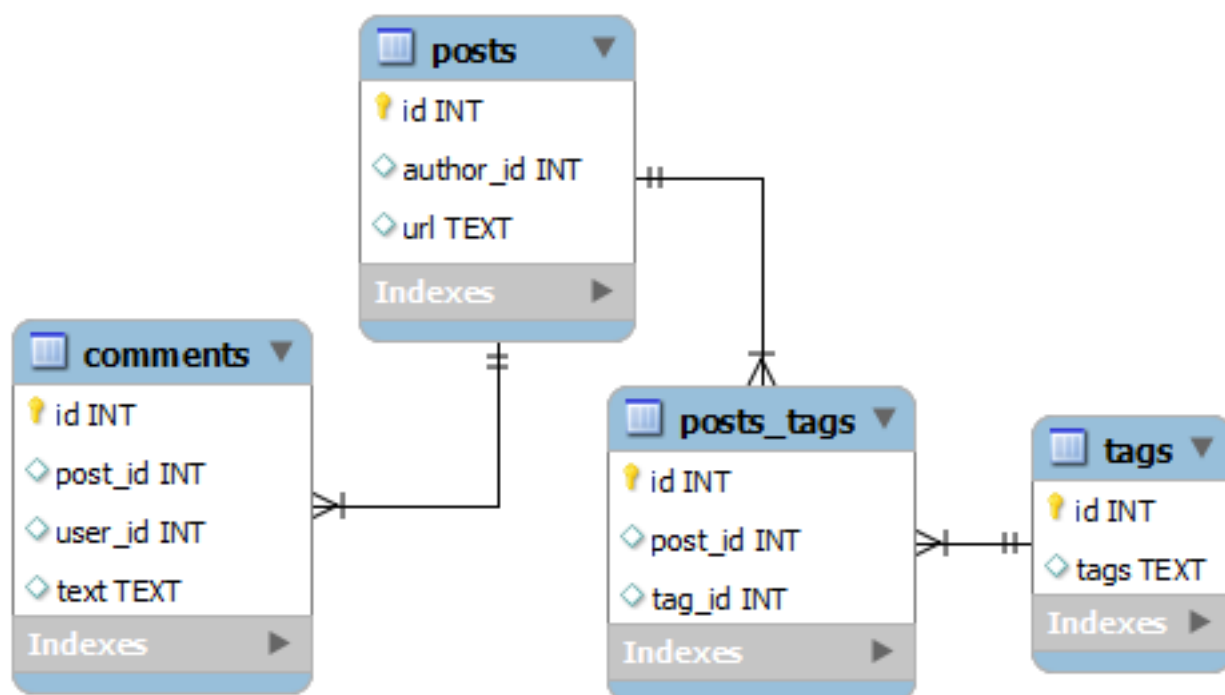
نویسنده: saremi
تاریخ: ۱۷:۲۲ ۱۳۹۲/۱۱/۲۹

سلام
می خواستم بپرسم bucket در key value store دقیقا چیه؟
بعد خیلی از جاها راجع به hash table و hash code هم مطالبی گفته اند. آیا منظور فقط hash کردن کلید است یا فرآیند پیچیده تر از این حرفاست؟

در مطلب قبلی با نوع اول پایگاه‌های داده NoSQL یعنی Key/Value Store آشنا شدیم و در این مطلب به معرفی دسته دوم یعنی Document Database خواهیم پرداخت.

در این نوع پایگاه داده، داده‌ها مانند نوع اول در قالب کلید/مقدار ذخیره می‌شوند و بازگردانی مقادیر نیز دقیقاً مشابه نوع اول یعنی Key/Value Store بر اساس کلید می‌باشد. اما تفاوت این سیستم با نوع اول در دسته‌بندی داده‌های مرتبط با یکدیگر در قالب یک Document می‌باشد. سعی کردم در این مطلب با ذکر مثال مطالب را شفاف‌تر بیان کنم:

به عنوان مثال اگر بخواهیم جداول مربوط به پست‌های یک سیستم CMS را بصورت رابطه‌ای پیاده کنیم، یکی از ساده‌ترین حالات پایه برای پست‌های این سیستم در حالت نرمال به صورت زیر می‌باشد.



جداول واضح بوده و نیازی به توضیح ندارد، حال نحوه‌ی ذخیره‌سازی داده‌ها در سیستم Document Database برای چنین مثالی را بررسی می‌کنیم:

```

{
  _id: ObjectId('4bf9e8e17cef4644108761bb'),
  Title: 'NoSQL Part3',
  url: 'http://dotnettips.info/yyy/xxxx',
  author: 'hamid samani',
  tags: ['databases', 'mongoDB'],
  comments: [
    {user: 'unknown user',
      text: 'unknown test'
    },
    {user: 'unknown user2',
      text: 'unknown text2'
    }
  ]
}
  
```



```
}
}
}
```

همانگونه که مشاهده می‌کنید نحوه‌ی ذخیره‌سازی داده‌ها بسیار با سیستم رابطه‌ای متفاوت می‌باشد ، با جمع‌بندی تفاوت نحوه‌ی نگهداری داده‌ها در این سیستم و RDBMS و بررسی این سیستم نکات اصلی به شرح زیر می‌باشند:

۱- فرمت ذخیره سازی داده‌ها مشابه فرمت JSON می‌باشد.

۲- به مجموعه داده‌های مرتبط به یکدیگر Document گفته می‌شود.

۳- در این سیستم JOIN ها وجود ندارند و داده‌های مرتبط کنار یکدیگر قرار می‌گیرند ، و یا به تعریف دقیق‌تر داده‌ها در یک داکيومنت اصلی Embed می‌شوند .

به عنوان مثال در اینجا مقدار comment ها برابر با آرایه‌ای از Document ها می‌باشد.

۴- مقادیر می‌توانند بصورت آرایه نیز در نظر گرفته شوند.

۵- در سیستم‌های RDBMS در صورتی که بخواهیم از وجود JOIN ها صرف‌نظر کنیم. به عدم توانایی در نرمال‌سازی بخواهیم خورد که یکی از معایب عدم نرمال‌سازی وجود مقادیر Null در جداول می‌باشد؛ اما در این سیستم به دلیل Schema free بودن می‌توان ساختارهای متفاوت برای Document ها در نظر گرفت.

به عنوان مثال برای یک پست می‌توان مقدار n کامنت تعریف کرد و برای پست دیگر هیچ کامنتی تعریف نکرد.

۶- در این سیستم اصولاً نیازی به تعریف ساختار از قبل موجود نمی‌باشد و به محض اعلان دستور قرار دادن داده‌ها در پایگاه داده ساختار متناسب ایجاد می‌شود.

با مقایسه دستورات CRUD در هر دو نوع پایگاه داده با نحوه‌ی کوئری گرفتن از Document Database آشنا می‌شویم:

در SQL برای ایجاد جدول خواهیم داشت:

```
CREATE TABLE posts (
  id INT NOT NULL
    AUTO_INCREMENT,
  author_id INT NOT NULL,
  url VARCHAR(50),
  PRIMARY KEY (id)
)
```

دستور فوق در Document Database معادل است با:

با قرار دادن مقدار نوع // db.posts.insert({id: "256" , author_id:"546",url:"http://example.com/xxx"}) ساختار مشخص می‌شود

در SQL جهت خواندن خواهیم داشت:

```
SELECT * from posts  
WHERE author_id > 100
```

و معادل آن برابر است با:

```
db.posts.find({author_id:{$gt:"1000"}})
```

در SQL جهت بروزرسانی داریم:

```
UPDATE posts  
SET author_id= "123"
```

که معادل است با:

```
db.posts.update({ $set: { author_id: "123" } })
```

در SQL جهت حذف خواهیم داشت:

```
DELETE FROM posts  
WHERE author_id= "654"
```

که معادل است با:

```
db.posts.remove( { author_id: "654" } )
```

همانگونه که مشاهده می‌فرمایید نوشتن کوئری برای این پایگاه داده ساده بوده و زبان آن نیز بر پایه جاوا اسکریپت می‌باشد که برای اکثر برنامه‌نویسان قابل درک است.

تاکنون توسط شرکت‌های مختلف پیاده‌سازی‌های مختلفی از این سیستم انجام شده است که از مهم‌ترین و پر استفاده‌ترین آنها می‌توان به موارد زیر اشاره کرد:

[MongoDB](#)

[CouchDB](#)

[RavenDB](#)

نظرات خوانندگان

نویسنده: سعید یزدانی
تاریخ: ۱۹:۵۷ ۱۳۹۱/۱۱/۲۹

با تشکر از مطلب زیباتون
یک سوال داشتم ایا این روش اونقدر به بلوغ رسیده که بشه در پروژه‌ها روش حساب کرد . یا اینکه فعلا از همون روش قبلی استفاده کنیم
سوال دیگر من هم این هست که به نظر شما در nosql آینده ایی دیده میشه ؟
با تشکر

نویسنده: سعید یزدانی
تاریخ: ۱۹:۵۹ ۱۳۹۱/۱۱/۲۹

اگر هم امکان داره refrence ی در این زمینه هست link بدید

نویسنده: حمید سامانی
تاریخ: ۲۱:۳ ۱۳۹۱/۱۱/۲۹

در رابطه با سوال اولتون عارضم که در حال حاضر همه‌ی شرکت‌های بزرگ و فعال در این صنعت مثل گوگل ، فیس ب و ک توئیترو از این شیوه استفاده می‌کنند ، در حالت کلی این مبحث یک تکنولوژی خاص نیست که مصرفی باشه و بعد از مدتی تاریخش بگذره ، یک Movement و یا یک نگرش کلی در تعریف عامه از مجموعه‌ای از راه حل‌ها به منظور رفع مشکلات RDBMS در پردازش داده‌های بزرگ (BigData) ، داده‌ها در حوزه‌ی وب هم که رشدی نمایی دارند.
در رابطه با سوال دوم هم بستگی به خود فرد و یا شرکت مربوطه داره ، در حوزه‌ی نرم‌افزارهای داخلی به دلیل پایین‌تر بودن حجم داده‌ها الزامی در استفاده از این روش‌ها نیست. (استفاده و یا عدم استفاده مستقیما به نوع نرم‌افزار و ساختار آن بستگی دارد)

نویسنده: حمید سامانی
تاریخ: ۲۱:۵ ۱۳۹۱/۱۱/۲۹

از [اینجا](#) که شما شروع کنید به همه جا لینک می‌شوید .)

نویسنده: سعید یزدانی
تاریخ: ۲۱:۲۴ ۱۳۹۱/۱۱/۲۹

ممنون بابت جواب کاملتون

نویسنده: توحید عزیزی
تاریخ: ۳:۵۲ ۱۳۹۱/۱۱/۳۰

سلام

سپاسگزارم از موضوع جالبی که انتخاب کرده اید و مطالب خوبی که می‌نویسید.
آیا امکان دارد که در مورد هر کدام از انواع دیتابیس نوسیکوئل، مثالهای بیشتری بزنید.
از یک سرویس رایگان برای نوشتن مثال‌ها می‌تواند استفاده کرد که برای همه در دسترس باشد، مثل: cloudant.com
با تشکر

نویسنده: Meysam Navaei
تاریخ: ۹:۱۶ ۱۳۹۱/۱۱/۳۰

سلام

توحید این سایت که معرفی کردی خیلی جالب بود. می خاستم بینم محدودیت حجمی در استفاده ازش وجود داره یا نه نامحدود. ریسک محسوب نمیشه ی پروژه بزرگ داشته باشی و بخای دیتابیس رو از سرویس این سایت استفاده کنی؟ منظورم اینکه از این سایتها نباشه که یهو محدودیت ایجاد بکنه و یا پولی بشه و...

نویسنده: حمید سامانی
تاریخ: ۱۶:۱۵ ۱۳۹۱/۱۱/۳۰

سلام

سعی می کنم مثال های بیشتری را در مطالب آتی بگنجانم.
(با سپاس)

نویسنده: توحید عزیزی
تاریخ: ۹:۳ ۱۳۹۱/۱۲/۰۳

سلام.

نسخه رایگانش محدودیت داره: فکر کنم 2000 کوئری در روز.
اگر می خواهید پروژه ی بزرگ روش ببرید، باید از نسخه های تجاریش استفاده کنید. البته من خودم تستش نکرده ام هنوز.

<https://cloudant.com/#home-pricing>

نویسنده: masi
تاریخ: ۱:۵۶ ۱۳۹۲/۰۲/۱۹

سلام ، واقعا مطالب خوبی بود هر جا رو گشتم کاملتر و جامع تر از همه بودید ، موضوع پروژه ی من روی این موضوعه ، ای کاش میشد در مورد موضوع زیر صحبت کنید. key value store

نویسنده: جواد زبیدی
تاریخ: ۳:۳۳ ۱۳۹۲/۰۵/۱۶

سلام تشکر از مطلب بسیار مفیدتون .

می خواستم بدونم که کدام یک از روش ها بیشتر امتحان خودش رو توی داده های زیاد پس داده . و بشه راحت تر باهاش کار کرد .
روش

Document store

Key value

روش هایی دیگری رو هم توی سایت دیدم اگر امکان داره مزایا و معایب هر کدوم رو توضیح دهید ممنون.

نویسنده: دادخواه
تاریخ: ۱۲:۱۰ ۱۳۹۲/۰۶/۰۷

سلام

تشکر از مطالب خوبتون

اما چند تا سوال دارم.

1- از این سه تا پایگاه داده که در اخر نوشتید فکر کنم فقط MongoDB مجانی باشه. درسته؟

2- آیا دستورات در همه این پایگاه داده ها به همین صورت است؟

3- آیا همه سرورها و هاست ها از این پایگاه داده ها مانند MS SQL پشتیبانی می کنند و یا سرورهای خاص را باید پیدا کرد؟
تشکر

نویسنده: محسن خان

تاریخ: ۱۳۹۲/۰۶/۰۷ ۱۲:۲۵

اگر مطالب [مقدماتی تر رو](#) مطالعه می کردید، می دید که اصلا هدف از بانک اطلاعاتی NoSQL این نیست که باهش سایت معمولی درست کنند اون هم روی سرور اجاره ای با 100 مگ فضا. هدفش توزیع شده بودن در سرورهایی متعدد و یا با پراکندگی جغرافیایی بالا است.

نتیجه گیری؟ ابزار زده نباشید. اول مفاهیم رو مطالعه کنید. اول تئوری کار مهمه.

نویسنده: saremi

تاریخ: ۱۳۹۲/۱۱/۲۴ ۱۶:۴۴

با سلام؛ میخواستم در مورد UNQL، CQL، HQL، map reduce و... بپرسم. توی همون سایتی که لینکش رو دادین اینها جزو انواع query method هستند. من دقیق نمیفهمم الان ما دستورات مشابه sql رو معادلش رو با java script نوشتیم. در مورد تفاوت اینها و استفاده شون اگر میشه کمی توضیح بدین لطفا. دقیقا توی انواع مختلف پایگاه داده با چه زبانی کوئری نویسی می شه؟ با تشکر

نویسنده: محسن خان

تاریخ: ۱۳۹۲/۱۱/۲۴ ۱۶:۵۲

در مورد تفاوت اینها در مطلب [مروری بر مفاهیم مقدماتی NoSQL](#) بیشتر توضیح داده شده. برچسب [NoSQL](#) را بهتر است دنبال کنید.

نویسنده: وحید نصیری

تاریخ: ۱۳۹۲/۱۱/۲۴ ۱۷:۴۹

در مورد MongoDB یک کتابچه ی فارسی 90 صفحه ای [موجود است](#) .

نویسنده: salam

تاریخ: ۱۳۹۳/۰۵/۰۹ ۱۱:۱۶

سلام

در قسمت دوم این مطلب اومده که "در حالت کلی پایگاه های داده NoSQL به ۴ دسته تقسیم می شوند " دسته 3 و 4 را توضیح نمی دین؟

نویسنده: وحید نصیری

تاریخ: ۱۳۹۳/۰۵/۰۹ ۱۱:۲۶

برای دنبال کردن مطالب هم خانواده در این سایت، در ذیل هر مطلب یک سری گروه یا برچسب تعریف شده اند. برای مثال اگر برچسب [NoSQL](#) را دنبال کنید، در مطالب دیگری پاسخ خود را خواهید یافت.

هدف از این مبحث، آشنایی با مفاهیم پایه‌ای اغلب بانک‌های اطلاعاتی NoSQL است که به صورت مشترکی در تمام آن‌ها بکار رفته است. برای مثال بانک‌های اطلاعاتی NoSQL چگونه مباحث یکپارچگی اطلاعات را مدیریت می‌کنند؟ نحوه ایندکس نمودن اطلاعات در آن‌ها چگونه است؟ چگونه از اطلاعات کوئری می‌گیرند؟ الگوریتم‌های محاسباتی مانند MapReduce چیستند و چگونه در اینگونه بانک‌های اطلاعاتی بکار رفته‌اند؟ همچنین الگوهای Sharding و Partitioning که در اغلب بانک‌های اطلاعاتی NoSQL مشترکند، به چه نحوی پیاده سازی شده‌اند.

لیست مشترکات بانک‌های اطلاعاتی NoSQL

قبل از اینکه بخواهیم وارد ریز جزئیات بانک‌های اطلاعاتی NoSQL شویم، نیاز است لیست و سرفصلی از مفاهیم اصلی و مشترک بین اینگونه بانک‌های اطلاعاتی را تدارک ببینیم که شامل موارد ذیل می‌شود:

الف) Non-Relational یا غیر رابطه‌ای

از کلمه NoSQL عموماً اینطور برداشت می‌شود که در اینجا دیگر خبری از SQL نویسی نیست که در عمل برداشت نادرستی است. شاید جالب باشد که بدانید، تعدادی از بانک‌های اطلاعاتی NoSQL از زبان SQL نیز به عنوان اینترفیسی برای نوشتن کوئری‌های مرتبط، پشتیبانی می‌کنند. کلمه NoSQL بیشتر به Non-Relational یا غیر رابطه‌ای بودن اینگونه بانک‌های اطلاعاتی بر می‌گردد. مباحثی مانند مدل‌های داده‌ای نرمال شده، اتصالات و Join جداول، در دنیای NoSQL وجود خارجی ندارند.

ب) Non-schematized/schema free یا بدون اسکیم

مفهوم مهم و مشترک دیگری که در بین بانک‌های اطلاعاتی NoSQL وجود دارد، بدون اسکیم بودن اطلاعات آن‌ها است. به این معنا که با حرکت از رکورد یک به رکورد دو، ممکن است با دو ساختار داده‌ای متفاوت مواجه شوید.

ج) Eventual consistency یا عاقبت یک دست شدن

عاقبت یک دست شدن، به معنای دریافت دستوری از شما و نحوه پاسخ دادن به آن (یا حتی پاسخ ندادن به آن) از طرف بانک اطلاعاتی NoSQL است. برای مثال، زمانی که یک رکورد جدید را اضافه می‌کنید، یا اطلاعات موجودی را به روز رسانی خواهید کرد، اغلب بانک‌های اطلاعاتی NoSQL این دستور را بسیار سریع دریافت و پردازش خواهند کرد. اما تفاوت است بین دریافت پیام و پردازش واقعی آن در اینجا. اکثر بانک‌های اطلاعاتی NoSQL، پردازش و اعمال واقعی دستورات دریافتی را با یک تاخیر انجام می‌دهند. به این ترتیب می‌توان خیلی سریع به بانک اطلاعاتی اعلام کرد که چه می‌خواهیم و بانک اطلاعاتی بلافاصله مجدداً کنترل را به شما بازخواهد گرداند. اما اعمال و انتشار واقعی این دستور، مدتی زمان خواهد برد.

د) Open source یا منبع باز بودن

اغلب بانک‌های اطلاعاتی NoSQL موجود، منبع باز هستند که علاوه بر بهره بردن از مزایای اینگونه پروژه‌ها، استفاده کنندگان سورس باز دیگری را نیز ترغیب به استفاده از آن‌ها کرده‌اند.

ه) Distributed یا توزیع شده

هرچند امکان پیاده سازی توزیع شده بانک‌های اطلاعاتی رابطه‌ای نیز وجود دارد، اما نیاز به تنظیمات قابل توجهی برای حصول این امر می‌باشد. در دنیای NoSQL، توزیع شده بودن جزئی از استاندارد تهیه اینگونه بانک‌های اطلاعاتی است و بر اساس این مدل ذهنی شکل گرفته‌اند. به این معنا که اطلاعات را می‌توان بین چندین سیستم تقسیم کرد، که حتی این سیستم‌ها ممکن است فواصل جغرافیایی قابل توجهی نیز با یکدیگر داشته باشند.

و) Web scale یا مناسب برای برنامه‌های تحت وب پر کاربر

امروزه بسیاری از کمپانی‌های بزرگ اینترنتی، برای مدیریت تعداد بالایی از کاربران همزمان خود، مانند فیس‌بوک، یاهو، گوگل، LinkedIn، مایکروسافت و غیره، نیاز به بانک‌های اطلاعاتی پیدا کرده‌اند که باید در مقابل این حجم عظیم درخواست‌ها و همچنین اطلاعاتی که دارند، بسیار بسیار سریع پاسخ دهند. به همین جهت بانک‌های اطلاعاتی NoSQL ابداع شده‌اند تا بتوان برای این نوع سناریوها پاسخی را ارائه داد.

و نکته مهم دیگر اینجا است که خود این کمپانی‌های بزرگ اینترنتی، بزرگترین توسعه دهنده‌های بانک‌های اطلاعاتی NoSQL نیز هستند.

نحوه مدیریت یکپارچگی اطلاعات در بانک‌های اطلاعاتی NoSQL

مدیریت یکپارچگی اطلاعات بانک‌های اطلاعاتی NoSQL به علت ذات و طراحی توزیع شده آن‌ها، با نحوه مدیریت یکپارچگی اطلاعات بانک‌های اطلاعاتی رابطه‌ای متفاوت است. اینجا است که **تئوری خاصی به نام CAP** مطرح می‌شود که شامل یکپارچگی یا Consistency به همراه Availability یا دسترسی پذیری (همیشه برقرار بودن) و partition tolerance یا توزیع پذیری است. در تئوری CAP مطرح می‌شود که هر بانک اطلاعاتی خاص، تنها دو مورد از سه مورد مطرح شده را می‌تواند با هم پوشش دهد. به این ترتیب بانک‌های اطلاعاتی رابطه‌ای عموماً دو مورد C و P یا یکپارچگی (Consistency) و partition tolerance یا میزان تحمل تقسیم شدن اطلاعات را ارائه می‌دهند. اما بانک‌های اطلاعاتی NoSQL از این تئوری، تنها دو مورد A و P را پوشش می‌دهند (دسترسی پذیری و توزیع پذیری مطلوب).

بنابراین مفهومی به نام ACID که در بانک‌های اطلاعاتی رابطه‌ای ضامن یکپارچگی اطلاعات آن‌ها است، در دنیای NoSQL وجود خارجی ندارد. کلمه ACID مخفف موارد ذیل است:

Durability, Atomicity, Consistency, Isolation

ACID در بانک‌های اطلاعاتی رابطه‌ای تضمین شده است. در این نوع سیستم‌ها، با ایجاد تراکنش‌ها، مباحث ایزوله سازی و یکپارچگی اطلاعات به نحو مطلوبی مدیریت می‌گردد؛ اما دنیای NoSQL، دسترسی پذیری را به یکپارچگی ترجیح داده است و به همین جهت پیشتر مطرح شد که مفهوم «Eventual consistency یا عاقبت یک دست شدن» در این نوع بانک‌های اطلاعاتی در پشت صحنه بکار گرفته می‌شود. یک مثال دنیای واقعی از عاقبت یک دست شدن اطلاعات را حتماً در مباحث DNS مطالعه کرده‌اید. زمانیکه یک رکورد DNS اضافه می‌شود یا به روز خواهد شد، اعمال این دستورات در سراسر دنیا به یکباره و همزمان نیست. هرچند اعمال این اطلاعات جدید در یک نود شبکه ممکن است آنی باشد، اما پخش و توزیع آن در سراسر سرورهای DNS دنیا، مدتی زمان خواهد برد (گاهی تا یک روز یا بیشتر).

به همین جهت است که بانک‌های اطلاعاتی رابطه‌ای در حجم‌های عظیم اطلاعات و تعداد کاربران همزمان بالا، کند عمل می‌کنند. حجم اطلاعات بالا است، مدتی زمان خواهد برد تا تغییرات اعمال شوند، و چون مفهوم ACID در این نوع بانک‌های اطلاعاتی تضمین شده است، کاربران باید مدتی منتظر بمانند و نمونه‌ای از آن‌ها را با dead lockهای شایع، احتمالاً پیشتر بررسی یا تجربه کرده‌اید. در مقابل، بانک‌های اطلاعاتی NoSQL بجای یکپارچگی، دسترسی پذیری را اولویت اول خود می‌دانند و نه یکپارچگی اطلاعات را. در یک بانک اطلاعاتی NoSQL، دستور ثبت اطلاعات دریافت می‌شود (این مرحله آنی است)، اما اعمال نهایی آن آنی نیست و مدتی زمان خواهد برد تا تمام اطلاعات در کلیه سرورها یک دست شوند.

نحوه مدیریت Indexing اطلاعات در بانک‌های اطلاعاتی NoSQL

اغلب بانک‌های اطلاعاتی NoSQL تنها بر اساس اطلاعات کلیدهای اصلی جداول آن‌ها index می‌شوند (البته نام خاصی به نام «جدول»، بسته به نوع بانک اطلاعاتی NoSQL ممکن است متفاوت باشد، اما منظور ظرف دربرگیرنده تعدادی رکورد است در اینجا). این ایندکس نیز از نوع clustered است. به این معنا که اطلاعات به صورت فیزیکی، بر همین مبنا ذخیره و مرتب خواهند شد. یک مثال: بانک اطلاعاتی NoSQL خاصی به نام Hbase که بر فراز Hadoop distributed file system طراحی شده است، دقیقاً به همین روش عمل می‌کند. این فایل سیستم، تنها از روش Append only برای ذخیره سازی اطلاعات استفاده می‌کند و در آن مفهوم دسترسی اتفاقی یا random access پیاده سازی نشده است. در این حالت، تمام نوشتن‌ها در بافر، لاگ می‌شوند و در بازه‌های زمانی متناوب و مشخصی سبب باز تولید فایل‌های موجود و مرتب سازی مجدد آن‌ها از ابتدا خواهند شد. دسترسی به این اطلاعات پس از تکمیل نوشتن، به علت مرتب سازی فیزیکی که صورت گرفته، بسیار سریع است. همچنین مصرف کننده سیستم نیز چون بلافاصله پس از ثبت اطلاعات در بافر سیستم، کنترل را به دست می‌گیرد، احساس کار با سیستمی را خواهد داشت که بسیار سریع است.

به علاوه Index های دیگری نیز وجود دارند که بر اساس کلیدهای اصلی جداول تولید نمی‌شوند و به آن‌ها ایندکس‌های ثانویه یا secondary indexes نیز گفته می‌شود و تنها تعداد محدودی از بانک‌های اطلاعاتی NoSQL از آن‌ها پشتیبانی می‌کنند. این مساله هم از اینجا ناشی می‌شود که با توجه به بدون اسکیم بودن جداول بانک‌های اطلاعاتی NoSQL، چگونه می‌توان اطلاعاتی را ایندکس کرد که ممکن است در رکورد دیگری، ساختار متناظر با آن اصلا وجود خارجی نداشته باشد.

نحوه پردازش Queries در بانک‌های اطلاعاتی NoSQL

بانک‌های اطلاعاتی NoSQL عموماً از زبان کوئری خاصی پشتیبانی نمی‌کنند. در اینجا باید به اطلاعات به شکل فایل‌هایی که حاوی رکوردها هستند نگاه کرد. به این ترتیب برای پردازش و یافتن اطلاعات درون این فایل‌ها، نیاز به ایجاد برنامه‌هایی است که این فایل‌ها را گشوده و بر اساس منطق خاصی، اطلاعات مورد نظر را استخراج کنند. گاهی از اوقات زبان SQL نیز پشتیبانی می‌شود ولی آنچنان عمومیت ندارد. الگوریتمی که در این برنامه‌ها بکار گرفته می‌شود، Map Reduce نام دارد. Map Reduce به معنای نوشتن کدی است، با دو تابع. اولین تابع اصطلاحاً Map step یا مرحله نگاشت نام دارد. در این مرحله کوئری به قسمت‌های کوچکتری خرد شده و بر روی سیستم‌های توزیع شده به صورت موازی اجرا می‌شود. مرحله بعد Reduce step نام دارد که در آن، نتیجه دریافتی حاصل از کوئری‌های اجرا شده بر روی سیستم‌های مختلف، با هم یکی خواهند شد. این روش برای نمونه در سیستم Hadoop بسیار مرسوم است. Hadoop دارای یک فایل سیستم توزیع شده است (که پیشتر در مورد آن بحث شد) به همراه یک موتور Map Reduce توکار. همچنین رده دیگری از بانک‌های اطلاعاتی NoSQL، اصطلاحاً Wide column store نام دارند (مانند Hbase) که عموماً به همراه Hadoop بکار گرفته می‌شوند. موتور Map Reduce متعلق به Hadoop بر روی جداول Hbase اجرا می‌شوند. به علاوه Amazon web services دارای سرویسی است به نام Elastic map reduce یا EMR که در حقیقت مجموعه‌ی پردازش ابری است که بر مبنای Hadoop کار می‌کند. این سرویس قادر است با بانک‌های اطلاعاتی NoSQL دیگر و یا حتی بانک‌های اطلاعاتی رابطه‌ای نیز کار کند.

بنابراین MapReduce، یک بانک اطلاعاتی نیست؛ بلکه یک روش پردازش اطلاعات است که فایل‌ها را به عنوان ورودی دریافت کرده و یک فایل را به عنوان خروجی تولید می‌کند. از آنجائیکه بسیاری از بانک‌های اطلاعاتی NoSQL کار عمده‌اشان، ایجاد و تغییر فایل‌ها است، اغلب جداول اطلاعات آن‌ها ورودی و خروجی‌های معتبری برای یک موتور Map reduce به حساب می‌آیند. در این بین، افزونه‌ای برای Hadoop به نام [Hive](#) طراحی شده است که با ارائه HivesQL، امکان نوشتن کوئری‌هایی SQL مانند را بر فراز موتورهای Map reduce ممکن می‌سازد. این افزونه با Hive tables خاص خودش و یا با Hbase سازگار است.

آشنایی مقدماتی با مفاهیمی مانند الگوهای Sharding و Partitioning در بانک‌های اطلاعاتی NoSQL

Sharding (شاردینگ تلفظ می‌شود) یک الگوی تقسیم اطلاعات بر روی چندین سرور است که اساس توزیع شده بودن بانک‌های اطلاعاتی NoSQL را تشکیل می‌دهد. این نوع تقسیم اطلاعات، از کوئری‌هایی به نام Fan-out پشتیبانی می‌کند. به این معنا که شما کوئری خود را به نود اصلی ارسال می‌کنید و سپس به کمک موتورهای Map reduce، این کوئری بر روی سرورهای مختلف اجرا شده و نتیجه نهایی جمع‌آوری خواهد شد. به این ترتیب تقسیم اطلاعات، صرفاً به معنای قرار دادن یک سری فایل بر روی سرورهای مختلف نیست، بلکه هر کدام از این سرورها به صورت مستقل نیز قابلیت پردازش اطلاعات را دارند. امکان تکثیر و همچنین replication هر کدام از سرورها نیز وجود دارد که قابلیت بازیابی سریع و مقاومت در برابر خرابی‌ها و مشکلات را افزایش می‌دهند.

از آنجائیکه Shard ها را می‌توان در سرورهای بسیار متفاوت و گسترده‌ای از لحاظ جغرافیایی قرار داد، هر Shard می‌تواند همانند مفاهیم CDN نیز عمل کند؛ به این معنا که می‌توان Shard مورد نیاز سروری خاص را در محلی نزدیک‌تر به او قرار داد. به این ترتیب سرعت عملیات افزایش یافته و همچنین بار شبکه نیز کاهش می‌یابد.

4 رده و گروه عمده بانک‌های اطلاعاتی NoSQL وجود دارند؛ شامل:

- الف) Key-Value stores که پایه بانک‌های اطلاعاتی NoSQL را تشکیل داده و اهدافی عمومی را دنبال می‌کنند.
- ب) Wide column stores که در شرکت‌های بزرگ اینترنتی بیشتر مورد استفاده قرار گرفته‌اند.
- ج) Document stores یا بانک‌های اطلاعاتی NoSQL سندگرا.
- د) Graph databases که بیشتر برای ردیابی ارتباطات بین موجودیت‌ها بکار می‌روند.

و در تمام این گروه‌ها، مکانیزم‌های Key-Value به شدت مورد استفاده‌اند.

الف) Key-Value stores

[Key-Value stores](#) یکی از عمومی‌ترین و پایه‌ای‌ترین گروه‌های بانک‌های اطلاعاتی NoSQL را تشکیل می‌دهند. البته این مورد بدین معنا نیست که این رده، جزو محبوب‌ترین‌ها نیز به‌شمار می‌روند.

dynomite



riak



Windows Azure

این نوع بانک‌های اطلاعاتی شامل جداولی از اطلاعات هستند. هر جدول نیز شامل تعدادی ردیف است؛ چیزی همانند بانک‌های اطلاعاتی رابطه‌ای. اما در هر ردیف، یک Dictionary یا آرایه‌ای از اطلاعات key-value شکل را شاهد خواهید بود. در اینجا ساختار و اسکیمای ردیف‌ها می‌توانند نسبت به یکدیگر کاملاً متفاوت باشند (دید لیبرال نسبت به اسکیمای، که [در قسمت قبل](#) به آن پرداخته شد). در این بین، تنها تضمین خواهد شد که هر ردیف، Id منحصر بفردی دارد.

از این نوع بانک‌های اطلاعاتی، در سکوهاى کارى ابرى زیاد استفاده می‌شود. دو مثال مهم در اینباره شامل [Amazon SimpleDB](#) و [Azure Table Storage](#) هستند.

سایر نمونه‌های مهم دیگری از بانک‌های اطلاعاتی NoSQL که بر مبنای مفهوم Key-Value stores کار می‌کنند، عبارتند از [MemcacheDB](#) و [Voldemort](#). به علاوه در Amazon web services بانک اطلاعاتی دیگری به نام [DynamoDB](#) به عنوان یک سرویس عمومی در دسترس است. همچنین [Dynomite](#) نیز به عنوان نمونه سورس باز Dynamo مطرح است. [Redis](#) و [Riak](#) نیز جزو بانک‌های اطلاعاتی Key-Value store بسیار معروف به‌شمار می‌روند.

Key-Value Stores

Database	
Table: Customers	Table: Orders
Row ID: 101 First_Name: Vahid Last_Name: Nasiri Address: Iran Last_Order: 1501	Row ID: 1501 Price: 400 Item1: 42134 Item2: 23455
Row ID: 201 First_Name: Ali Last_Name: Moshfegh Address: Iran Last_Order: 1502	Row ID: 1502 Price: 300 Item1: 52134 Item2: 26455

همانطور که در تصویر فوق ملاحظه می‌کنید، Key-Value stores دارای بانک‌های اطلاعاتی شامل جداول مختلف هستند. در اینجا همچنین ساختار ردیف‌هایی از اطلاعات این جداول نیز مشخص شده‌اند. هر ردیف، یک کلید دارد به همراه تعدادی جفت کلید-مقدار. در این جداول، اسکیما ثابت نگه داشته شده است و از ردیفی به ردیف دیگر متفاوت نیست؛ اما این مساله اختیاری است. برای مثال می‌توان در ردیف اطلاعات یک مشتری خاص، کلید-مقدارهایی خاص او را نیز درج کرد که لزوماً در سایر ردیف‌ها، نیازی به وجود آن‌ها نیست. به علاوه باید به خاطر داشت که هرچند به ظاهر last_orderها به شماره Id سفارشات مرتبط هستند، اما مفاهیمی مانند کلیدهای خارجی بانک‌های اطلاعاتی رابطه‌ای، در اینجا وجود خارجی ندارند. بیشتر در اینجا هدف سهولت جستجوی اطلاعات است.

Wide column stores (ب)

Wide column stores دارای جداولی است که درون آن‌ها ستون‌هایی قابل تعریف است. درون این ستون‌ها که یادآور بانک‌های اطلاعاتی رابطه‌ای هستند، اطلاعات به شکل key-value با ساختاری متفاوت، قابل ذخیره سازی هستند. در اینجا هر ستون، می‌تواند شامل گروهی از ستون‌ها که بر اساس مفاهیم جفت‌های key-value کار می‌کنند، باشد. این نوع بانک‌های اطلاعاتی عموماً در سایت‌های اینترنتی بسیار بزرگ و برنامه‌های «Big data» استفاده می‌شوند. برای مثال:



- [BigTable](#) گوگل که یک محصول اختصاصی و غیرعمومی است؛ اما جزئیات آن را به عنوان مقالات علمی منتشر کرده است.
 - دنیای سورس باز به رهبری Yahoo، نمونه سورس باز BigTable را به نام [Hbase](#) ارائه داده است.
 - در فیس بوک، از بانک اطلاعاتی دیگری به نام [Cassandra](#) استفاده می‌کنند. در اینجا به گروهی از ستون‌ها super columns و جداول super column families گفته می‌شود.

Wide column stores

Table: Customers	Table: Orders
Row ID: 101 Super column: Name Column: First_Name: Vahid Column: Last_Name: Nasiri Super column: Address Column: Number: 10 Column: Street: Somewhere Super column: Orders Column: Last_Order: 1501	Row ID: 1501 Super column: Pricing Column: Price: 400 Super column: Items Column: Item1: 12345 Column: Item2: 14345
Row ID: 201 Super column: Name Column: First_Name: Ali Column: Last_Name: Moshfegh Super column: Address Column: Number: 101 Column: Street: Somewhere Super column: Orders Column: Last_Order: 1502	Row ID: 1502 Super column: Pricing Column: Price: 500 Super column: Items Column: Item1: 17345 Column: Item2: 14945

در اینجا نیز جداول و ردیف‌ها وجود دارند و هر ستون باید عضوی از خانواده یک super column باشد. ساختار ردیف‌ها در این تصویر یکسان در نظر گرفته شده‌اند، اما اگر نیاز بود، برای مثال می‌توان در ردیفی خاص، ساختار را تغییر داد و مثلا middle name را نیز بر اساس نیاز، به ردیفی اضافه کرد.

Document stores (ج)

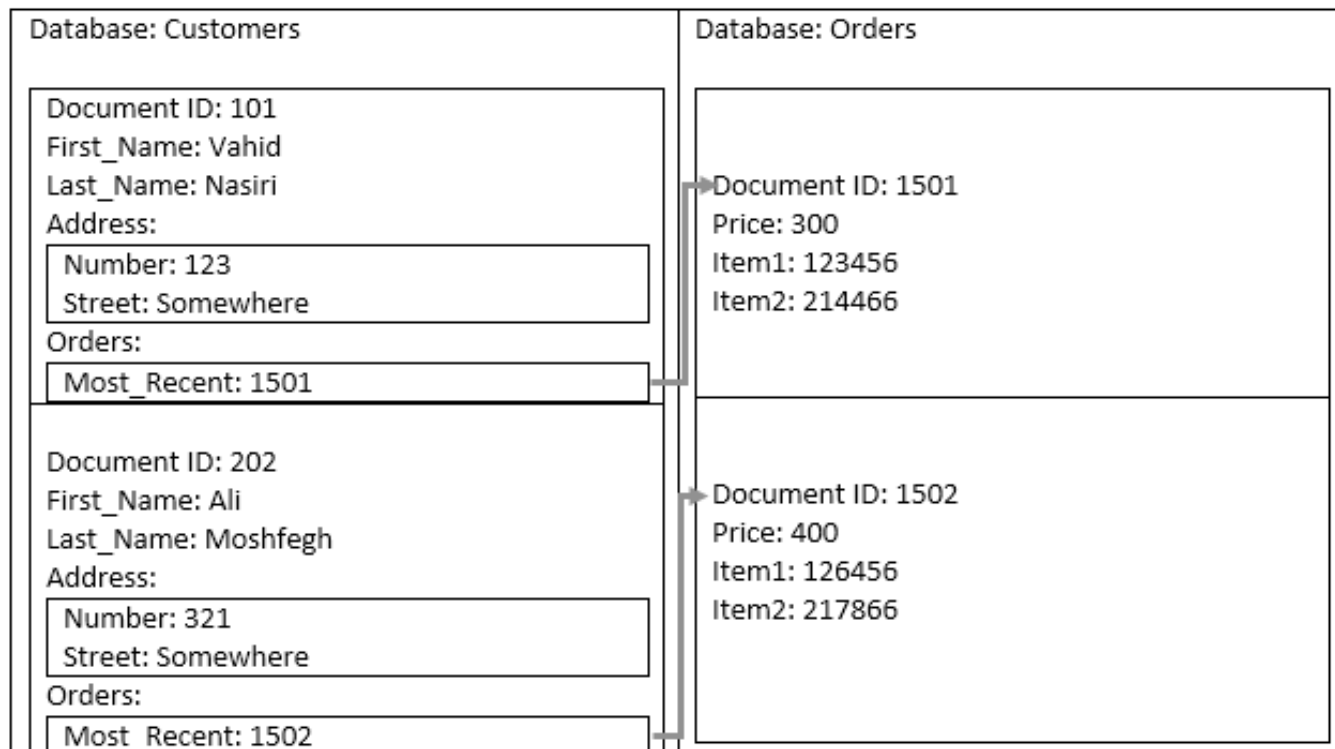
Document stores بجای جداول، دارای بانک‌های اطلاعاتی مختلفی هستند و در اینجا بجای ردیف‌ها، سند یا document دارند. ساختار سندها نیز عموماً بر مبنای اشیاء [JSON](#) تعریف می‌گردد (که البته این مورد الزامی نبوده و از هر محصول، به محصول دیگری ممکن است متفاوت باشد؛ اما عمومیت دارد). بنابراین هر سند دارای تعدادی خاصیت است (چون اشیاء JSON به این نحو تعریف می‌گردند) که دارای مقدار هستند. در نگاه اول، شاید این نوع اسناد، بسیار شبیه به key-value stores به نظر برسند. اما در حین تعریف اشیاء JSON، یک مقدار می‌تواند خود یک شیء کامل دیگر باشد و نه صرفاً یک مقدار ساده. به همین جهت عده‌ای به این نوع بانک‌های اطلاعاتی، بانک‌های اطلاعاتی Key-value store سفارشی و خاص نیز می‌گویند. این نوع ساختار منعطف، برای ذخیره سازی اطلاعات اشیاء تو در تو و درختی بسیار مناسب است. همچنین این اسناد می‌توانند حاوی پیوست‌هایی نیز باشد؛ مانند پیوست یک فایل به یک سند. در Document stores، نگارش‌های قدیمی اسناد نیز نگهداری می‌گردند. به همین جهت این نوع بانک‌های اطلاعاتی برای ایجاد برنامه‌های مدیریت محتوا نیز بسیار مطلوب می‌باشند. با توجه به مزایایی که برای این رده از بانک‌های اطلاعاتی NoSQL ذکر گردید، Document stores در بین برنامه نویسی‌ها بسیار محبوب و پرکاربرد هستند.

از این دست بانک‌های اطلاعاتی NoSQL، می‌توان به [MongoDB](#)، [CouchDB](#) و [RavenDB](#) اشاره کرد. سایر مزایای Document stores که به پرکاربرد شدن آن‌ها کمک کرده‌اند به شرح زیر هستند:

- هر سند را می‌توان با یک URI آدرس دهی کرد.
- برای نمونه CouchDB از یک full REST interface برای دسترسی و کار با اسناد پشتیبانی می‌کند (چیزی شبیه به ASP.NET WEB API در دات نت). در اینجا با استفاده از یک وب سرور توکار و بکارگیری HTTP Verbs مانند Put, Delete, Get و غیره، امکان کار با اسناد وجود دارد.

- اغلب بانک‌های اطلاعاتی Document stores از JavaScript به عنوان native language خود بهره می‌برند (جهت سهولت کار با اشیاء JSON).

Document stores



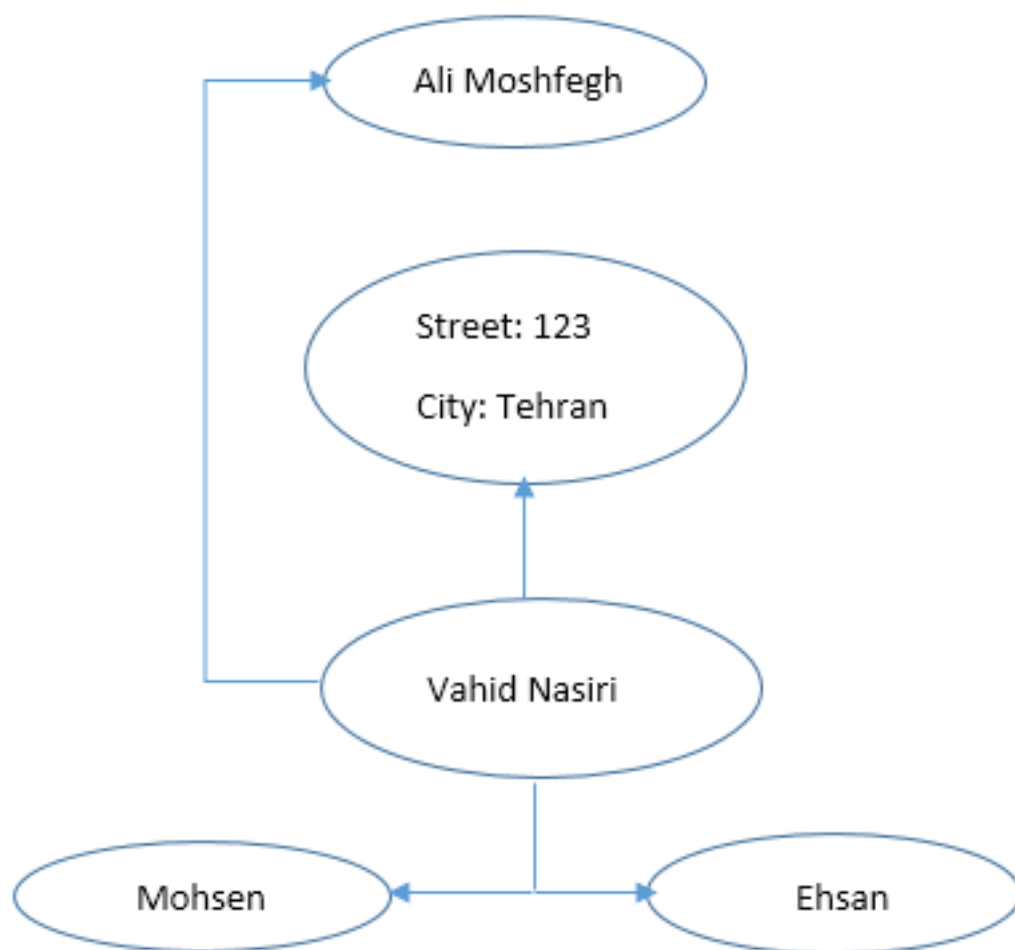
در اینجا دو دیتابیس، بجای دو جدول وجود دارند. همچنین در مقایسه با بانک‌های اطلاعاتی key-value، برای نمونه، مقدار خاصیت آدرس، خود یک شیء است که از دو خاصیت تشکیل شده است. به علاوه هر خاصیت Most_Recent یک Order، به سند دیگری در بانک اطلاعاتی Orders لینک شده است.

Graph databases (د)

Graph databases نوع خاصی از بانک‌های اطلاعاتی NoSQL هستند که جهت ردیابی ارتباطات بین اطلاعات طراحی شده‌اند و برای برنامه‌های شبکه‌های اجتماعی بسیار مفید هستند.

در واژه نامه این بانک‌های اطلاعاتی Nodes و Edges (اتصال دهنده‌های نودها) تعریف شده‌اند. در اینجا نودها می‌توانند دارای خاصیت‌ها و مقادیر متناظر با آن‌ها باشند.

یکی از معروفترین Graph databases مورد استفاده، [Neo4j](#) نام دارد.



در اینجا یک شخص را که دارای رابطه آدرس با شیء آدرس ذکر شده است را مشاهده می‌کنید. همچنین این شخص دارای رابطه دوستی با سه شخص دیگر است.

در سناریوهای خاصی، بانک‌های اطلاعاتی NoSQL خوش می‌درخشند و در بسیاری از موارد دیگر، بانک‌های اطلاعاتی رابطه‌ای بهترین گزینه انتخابی می‌باشند و نه بانک‌های اطلاعاتی NoSQL. در ادامه به بررسی این موارد خواهیم پرداخت.

در چه برنامه‌هایی استفاده از بانک‌های اطلاعاتی NoSQL مناسب‌تر است؟

- 1) برنامه‌های مدیریت محتوا
- 2) کاتالوگ‌های محصولات (هر برنامه‌ای با تعدادی شیء و خصوصاً متادیتای متغیر)
- 3) شبکه‌های اجتماعی
- 4) Big Data
- 5) سایر

1) برنامه‌های مدیریت محتوا

بانک‌های اطلاعاتی NoSQL سندگرا، جهت تهیه برنامه‌های مدیریت محتوا، بسیار مناسب هستند. در این نوع برنامه‌ها، یک سری محتوا که دارای متادیتایی هستند، ذخیره خواهند شد. این متادیتاها مانند نوع، گروه و هر نوع خاصیت دیگری، می‌تواند باشند. برای ذخیره سازی این نوع اطلاعات، جفت‌های key-value بسیار خوب عمل می‌کنند. همچنین در بانک‌های اطلاعاتی سندگرای NoSQL، با استفاده از مفهوم برچسب‌ها، امکان الصاق فایل‌های متناظری به اسناد پیش بینی شده است. همانطور که [در قسمت قبل](#) نیز ذکر شد، در Document stores، نگارش‌های قدیمی اسناد نیز حفظ می‌شوند. به این ترتیب، این خاصیت و توانمندی توکار، امکان دسترسی به نگارش‌های مختلف یک محتوای خاص را به سادگی میسر می‌سازد. به علاوه اکثر Document stores امکان دسترسی به این مستندات را به کمک URL ها و REST API، به صورت خودکار فراهم می‌سازند.

برای نمونه به CouchDB، عنوان Web database نیز داده شده است؛ از این جهت که یک برنامه وب را می‌توان داخل بانک اطلاعاتی آن قرار داد. در اینجا منظور از برنامه وب، یک وب سایت قابل دسترسی از طریق URL ها است و نه برنامه‌های سازمانی وب. برای نمونه ساختاری شبیه به برنامه معروف EverNote را می‌توان داخل این نوع بانک‌های اطلاعاتی به سادگی ایجاد کرد (خود بانک اطلاعاتی تشکیل شده است از یک وب سرور که REST API را پشتیبانی کرده و امکان دسترسی به اسناد را بدون نیاز به کدنویسی اضافه‌تری، از طریق URL ها و HTTP Verbs استاندارد مهیا می‌کند).

2) کاتالوگ‌های محصولات

محصولات در یک کاتالوگ، ویژگی‌های مشابه یکسان فراوانی دارند؛ اما تعدادی از این محصولات، دارای ویژگی‌هایی خاص و منحصر بفردی نیز می‌باشند.

مثلاً یک شیء محصول را در نظر بگیرید که دارای خواص مشترک و یکسان شماره، نام، توضیحات و قیمت است. اما بعضی از محصولات، بسته به رده‌ی خاصی که دارند، دارای ویژگی‌های خاصی مانند قدرت تفکیک، رنگ، سرعت و غیره نیز هستند که از هر گروه، به گروه دیگری متغیر است.

برای مدیریت یک چنین نیازی، هر دو گروه key-value stores و wide column stores بانک‌های اطلاعاتی NoSQL مناسب هستند؛ از این جهت که در یک key-value store نیازی به تعریف هیچ نوع ساختار خاصی، در ابتدای کار نیست و این ساختار می‌تواند از هر رکورد، به رکورد دیگری متفاوت باشد.

یا برای نمونه، یک برنامه فرم ساز را در نظر بگیرید که هر فرم آن، هر چند دارای یک سری خواص ثابت مانند نام، گروه و امثال آن است، اما هر کدام دارای فیلدهای تشکیل دهنده متفاوتی نیز می‌باشد. به این ترتیب با استفاده از key-value stores، دیگری نیازی به نگران بودن در مورد نحوه مدیریت اسکیمای متغیر مورد نیاز، نخواهد بود.

3) شبکه‌های اجتماعی

همانطور که [در قسمت قبل](#) نیز بحث شد، نوع خاص Graph databases برای کاربردهای برنامه‌های شبکه‌های اجتماعی و ردیابی تغییرات آن‌ها بسیار مفید و کارآ هستند. برای مثال در یک شبکه افراد دارای تعدادی دنبال کننده هستند؛ عضو گروه‌های مختلف می‌باشند، در قسمت‌های مختلفی نظر و مطلب ارسال می‌کنند. در اینجا، اشیاء نسبت به یکدیگر روابط مختلفی دارند. با استفاده از

Graph databases، تشکیل روابط self-joins و تو در تو و بسیاری از روش‌های خاص، مانند روابط many-to-many که در بانک‌های اطلاعاتی رابطه‌ای با تمهیدات ویژه‌ای قابل تشکیل هستند، با سهولت بهتری مدیریت خواهند شد.

Big Data (4)

الگوریتم MapReduce، برای کار با حجم داده‌های عظیم، طراحی شده است و در این بین، بانک‌های اطلاعاتی Wide column store (که در قسمت قبل بررسی شدند) و یا حتی Key-value store (مانند [Amazon DynamoDB](#)) بیشتر کاربرد دارند. در سناریوهای داده‌های عظیم، واژه‌های Hadoop و [Hbase](#) دنیای NoSQL را زیاد خواهید شنید. Hadoop نسخه سورس باز MapReduce گوگل است و Hbase نیز نسخه سورس باز BigTable گوگل می‌باشد. [مفاهیم پایه‌ای Sharding](#) و فایل سیستم‌های Append-only (با سرعت بالای نوشتن) نیز به مدیریت BigData کمک می‌کنند.

در اینجا بحث مهم، خواندن اطلاعات و آنالیز آن‌ها است و نه تهیه برنامه‌های معروف [CRUD](#). بسیاری از اعمال آماری و ریاضی مورد نیاز بر روی داده‌های عظیم، نیازی به اسکیمای از پیش مشخص شده بانک‌های اطلاعاتی رابطه‌ای را ندارند و یا در اینجا قابلیت‌های نوشتن کوئری‌های پیچیده نیز آنچنان مهم نیستند.

(5) سایر کاربردها

- هر سیستمی که اطلاعات Log مانند را تولید می‌کند، منظور از Log، اطلاعاتی است که در حین رخداد خاصی تولید می‌شوند.
- عموماً مرسوم است که این نوع اطلاعات را در فایل‌ها، بجای بانک اطلاعاتی ذخیره کرد. بنابراین مدیریت این نوع فایل‌ها توسط بانک‌های اطلاعاتی NoSQL، قابلیت انجام امور آماری را بر روی آن‌ها ساده‌تر خواهد ساخت.
- مدیریت اطلاعات برنامه‌هایی مانند سیستم‌های EMail.

و در چه برنامه‌هایی استفاده از بانک‌های اطلاعاتی رابطه‌ای مناسب‌تر است؟

اگر تا اینجا به مزایای استفاده از بانک‌های اطلاعاتی NoSQL اشاره شد، بدین معنا نیست که بانک‌های اطلاعاتی رابطه‌ای، منسوخ شده‌اند یا دیگر قدر و قیمتی ندارند. واقعیت این است که هنوز بازه وسیعی از کاربردها را می‌توان به کمک بانک‌های اطلاعاتی رابطه‌ای بهتر از بانک‌های اطلاعاتی NoSQL مدیریت کرد. این کاربردها و مزیت‌ها در 5 گروه عمده خلاصه می‌شوند:

- 1) نیاز به تراکنش‌ها
- 2) اسکیمای پیش فرض
- 3) برنامه‌های LOB یا Line of business applications
- 4) زبان‌های کوئری نویسی پیشرفته
- 5) نیاز به امکانات گزارشگیری پیشرفته

1) نیاز به تراکنش‌ها

در سیستم‌های تجاری عمومی، نیاز به پیاده سازی مفهوم ACID که در قسمت‌های قبل به آن پرداخته شد، مانند Atomic transactions وجود دارد. Atomic transaction به زبان ساده به این معنا است که سیستم قادر است چندین دستور را در قالب یک گروه و در طی یک مرحله، به بانک اطلاعاتی اعمال کند و اگر یکی از این دستورات گروه در حال اعمال، با شکست مواجه شد، باید کل تراکنش برگشت خورده و امنیت کار تضمین گردد. در غیراینصورت با یک سیستم غیر هماهنگ مواجه خواهیم شد. و همانطور که [بیشتر نیز عنوان شد](#)، سیستم‌های NoSQL، مبنای کار را بر اساس «عاقبت یک دست شدن» اطلاعات قرار داده‌اند؛ تا دسترسی پذیری به آن‌ها افزایش یافته و سرعت عملیات به این نحو بهبود یابد. در این نوع سیستم‌ها تضمینی در مورد ACID وجود ندارد.

2) اسکیمای پیش فرض

پروژه‌های متداول، دارای ساختاری مشخص و معمولی هستند. زیرا طراحی اولیه یک پروژه، بر مبنای مجموعه‌ای از اطلاعات است که همیشه باید وجود داشته باشند و اگر همانند بحث کاتالوگ‌های محصولات، نیاز به متادیتای متغیر نباشد، ساختار و اسکیمای یک پروژه، از ابتدای طراحی آن مشخص می‌باشد. و ... تمام این‌ها را به خوبی می‌توان توسط بانک‌های اطلاعاتی رابطه‌ای، با تعریف یک اسکیمای مشخص، مدیریت کرد.

3) برنامه‌های LOB یا Line of business applications

در برنامه‌های تجاری متداول، طراحی طرح‌بندی فرم‌های برنامه یا انقیاد داده‌ها، بر اساس یک اسکیمای مشخص صورت می‌گیرد. بدون داشتن یک اسکیمای مشخص، امکان تعاریف انقیاد داده‌ها به صورت strongly typed وجود نخواهد داشت. همچنین کل مفهوم Object relational mapping و ORM‌های مختلف نیز بر اساس وجود یک اسکیمای مشخص و از پیش تعیین شده کار می‌کند. بنابراین بانک‌های اطلاعاتی رابطه‌ای، انتخاب بسیار مناسبی برای تهیه برنامه‌های تجاری روزمره هستند.

4) زبان‌های کوئری نویسی پیشرفته

همانطور که [عنوان شد](#) برای تهیه کوئری بر روی اغلب بانک‌های اطلاعاتی NoSQL، باید توسط یک برنامه ثانویه، کار پیاده سازی الگوریتم Map Reduce را انجام داد. هر چند تعدادی از این نوع بانک‌های اطلاعاتی به صورت توکار دارای موتور MapReduce هستند، اما بسیاری از آن‌ها خیر. به همین جهت برای تهیه کوئری‌های متداول، کار پیاده سازی این برنامه‌های ثانویه مشکل خواهد بود. به این ترتیب نوشتن Ad Hoc queries و گزارشگیری بسیار مشکل می‌شوند. علاوه بر امکانات خوب کوئری گرفتن در بانک‌های اطلاعاتی رابطه‌ای، این کوئری‌ها در زمان اجرا نیز بر اساس اسکیمای موجود، بسیار بهینه و با سرعت بالا اجرا می‌شوند. قابلیتی که رسیدن به آن در بانک‌های اطلاعاتی با اسکیمای متغیر، کار ساده‌ای نیست و باید آن‌را با کدنویسی شخصی بهینه کرد. البته اگر تعداد این نوع برنامه‌های ثانویه که به آن‌ها imperative query در مقابل declarative query بانک‌های رابطه‌ای می‌گویند، کم باشد، شاید یکبار نوشتن و بارها استفاده کردن از آن‌ها اهمیتی نداشته باشد؛ در غیراینصورت تبدیل به یک عذاب خواهد شد.

5) نیاز به امکانات گزارشگیری پیشرفته

گزارشگیرهای برنامه‌های تجاری نیز بر اساس یک ساختار و اسکیمای مشخص به کمک قابلیت‌های پیشرفته کوئری نویسی بانک‌های اطلاعاتی رابطه‌ای به سادگی قابل تهیه هستند. برای تهیه گزارشاتی که قابلیت چاپ مناسبی را داشته باشند، محل قرارگیری فیلدهای مختلف در صفحه مهم هستند و با متغیر بودن آن‌ها، قابلیت طراحی از پیش آن‌ها را از دست خواهیم داد. در این حالت با اسکیمای متغیر، حداکثر بتوان یک dump از اطلاعات را به صورت ستونی نمایش داد.

بنابراین به صورت خلاصه، بانک‌های اطلاعاتی رابطه‌ای، جهت مدیریت کارهای روزمره تجاری اغلب شرکت‌ها، بسیار ضروری و جزو مسایل پایه‌ای به‌شمار می‌روند و به این زودی‌ها هم قرار نیست با نمونه‌ی دیگری جایگزین شوند.

نظرات خوانندگان

نویسنده: مرادی
تاریخ: ۸:۱۲ ۱۳۹۲/۰۶/۱۷

با سلام، با توجه به این که Raven Db یکی از دیتابیس‌های No SQL، در پشت صحنه خود از دیتابیس ویندوز با نام ESENT استفاده می‌کند، که مطابق با مستندات سایت‌های معتبر چون MSDN و ویکی پدیا هم Transactional است، و هم دارای امکانات دیگر، آیا می‌توان Transactional بودن را مزیتی صرف برای دیتابیس‌های رابطه ای به شمار آورد، و برای دیتابیس‌های No SQL این مزیت را قائل نشد ؟

<http://blogs.msdn.com/b/windowssdk/archive/2008/10/23/esent-extensible-storage-engine-api-in-the-windows-sdk.aspx> Windows comes with an embeddable, transactional database engine
http://en.wikipedia.org/wiki/Extensible_Storage_Engine

ESE provides transacted data update and retrieval

با سپاس

نویسنده: وحید نصیری
تاریخ: ۹:۲۲ ۱۳۹۲/۰۶/۱۷

- بحث فوق، بحثی است عمومی و حاصل برآیند بررسی اکثر بانک‌های اطلاعاتی NoSQL.
- بدیهی است این بین ممکن است استثنائهایی هم وجود داشته باشند. برای مثال RavenDB داخل document store خود transactional عمل می‌کند؛ اما کوئری‌های آن (که بر روی ایندکس‌های لوسین اجرا می‌شوند) از اصل عاقبت یک‌دست شدن پیروی می‌کنند. همچنین این کوئری‌ها را هم می‌شود طوری تنظیم کرد که stale data باز نگردانند.

روشی را که مایکروسافت برای پرداختن به مقوله NoSQL تاکنون انتخاب کرده است، قرار دادن ویژگی‌هایی خاصی از دنیای NoSQL مانند امکان تعریف اسکیمای متغیر، داخل مهم‌ترین بانک اطلاعاتی رابطه‌ای آن، یعنی SQL Server است، که در ادامه به آن خواهیم پرداخت. همچنین در سمت محصولات پردازش ابری آن نیز امکان دسترسی به محصولات NoSQL کاملی وجود دارد.

Azure table storage (1)

Azure table storage در حقیقت یک Key-value store ابری است و برای کار با آن از اینترفیس پروتکل استاندارد OData استفاده می‌شود. علت استفاده و طراحی یک سیستم Key-value store در اینجا، مناسب بودن اینگونه سیستم‌ها جهت مقاصد عمومی است و به این ترتیب می‌توان به بازه بیشتری از مصرف کنندگان، خدمات ارائه داد. پیش از ارائه Azure table storage، مایکروسافت سرویس خاصی را به نام SQL Server Data Services که به آن SQL Azure نیز گفته می‌شود، معرفی کرد. این سرویس نیز یک Key-Value store است؛ هرچند از SQL Server به عنوان مخزن نگهداری اطلاعات آن استفاده می‌کند.

SQL Azure XML Columns (2)

فیلدهای XML از سال 2005 به امکانات توکار SQL Server اضافه شدند و این نوع فیلدها، بسیاری از مزایای دنیای NoSQL را درون SQL Server رابطه‌ای مهیا می‌سازند. برای مثال با تعریف یک فیلد به صورت XML، می‌توان از هر ردیف به ردیفی دیگر، اطلاعات متفاوتی را ذخیره کرد؛ به این ترتیب امکان کار با یک فیلد که می‌تواند اطلاعات یک شیء را قبول کند و در حقیقت امکان تعریف اسکیمای پویا و متغیر را در کنار امکانات یک بانک اطلاعاتی رابطه‌ای که از اسکیمای ثابت پشتیبانی می‌کند، میسر می‌شود. در این حالت در هر ردیف می‌توان تعدادی ستون ثابت را با یک ستون XML با اسکیمای کاملاً پویا ترکیب کرد. همچنین SQL Server در این حالت قابلیت را ارائه می‌دهد که در بسیاری از بانک‌های اطلاعاتی NoSQL میسر نیست. در اینجا در صورت نیاز و لزوم می‌توان اسکیمای کاملاً مشخصی را به یک فیلد XML نیز انتساب داد؛ هر چند این مورد اختیاری است و می‌توان یک un typed XML را نیز بکار برد. به علاوه امکانات کوئری گرفتن توکار از این اطلاعات را به کمک XPath ترکیب شده با T-SQL، نیز فراموش نکنید.

بنابراین اگر یکی از اهداف اصلی گرایش شما به سمت دنیای NoSQL، استفاده از امکان تعریف اطلاعاتی با اسکیمای متغیر و پویا است، فیلدهای نوع XML اس کیوال سرور را مدنظر داشته باشید. یک مثال عملی: فناوری Azure Dev Fabric's Table Storage (نسخه Developer ویندوز Azure که روی ویندوزهای معمولی اجرا می‌شود؛ یک شبیه ساز خانگی) به کمک SQL Server و فیلدهای XML آن طراحی شده است.

SQL Azure Federations (3)

در اینجا منظور از Federations در حقیقت همان پیاده سازی قابلیت Sharding بانک‌های اطلاعاتی NoSQL توسط SQL Azure است که برای توزیع اطلاعات بر روی سرورهای مختلف طراحی شده است. به این ترتیب دو قابلیت Partitioning و همچنین Replication به صورت خودکار در دسترس خواهند بود. هر Partition در اینجا، یک SQL Azure کامل است. بنابراین چندین بانک اطلاعاتی فیزیکی، یک بانک اطلاعاتی کلی را تشکیل خواهند داد.

هرچند در اینجا Sharding (که به آن Federation member گفته می‌شود) و در پی آن مفهوم «عاقبت یک دست شدن اطلاعات» وجود دارد، اما درون یک Shard یا یک Federation member، مفهوم ACID پیاده سازی شده است. از این جهت که هر Shard واقعا یک بانک اطلاعاتی رابطه‌ای است. اینجا است که مفهوم برنامه‌های Multi-tenancy را برای درک آن باید در نظر داشت. برای نمونه یک برنامه وب را در نظر بگیرید که قسمت اصلی اطلاعات کاربران آن بر روی یک Shard قرار دارد و سایر اطلاعات بر روی سایر Shards پراکنده شده‌اند. در این حالت است که یک برنامه وب با وجود مفهوم ACID در یک Shard می‌تواند سریع پاسخ دهد که آیا کاربری پیشتر در سایت ثبت نام کرده است یا خیر و از ثبت نام‌های غیرمجاز جلوگیری به عمل آورد. در اینجا تنها موردی که پشتیبانی نشده است، کوئری‌های Fan-out می‌باشد که [پیشتر](#) در مورد آن بحث شد. از این جهت که با نحوه خاصی که Sharding آن طراحی شده است، نیازی به تهیه کوئری‌هایی که به صورت موازی بر روی کلیه Shards برای جمع آوری اطلاعات اجرا می‌شوند، نیست. هر چند از هر shard با استفاده از برنامه‌های دات نت، می‌توان به صورت جداگانه نیز کوئری

گرفت.

4 OData

اگر به CouchDB و امکان دسترسی به امکانات آن از طریق وب دقت کنید، در محصولات مایکروسافت نیز این دسترسی REST API پیاده سازی شده‌اند.

OData یک RESTful API است برای دسترسی به اطلاعاتی که به شکل XML یا JSON بازگشت داده می‌شوند. انواع و اقسام کلاینت‌هایی برای کار با آن از جاوا اسکریپت گرفته تا سیستم‌های موبایل، دات نت و جاوا، وجود دارند. از این API نه فقط برای خواندن اطلاعات، بلکه برای ثبت و به روز رسانی داده‌ها نیز استفاده می‌شود. در سیستم‌های جاری مایکروسافت، بسیاری از فناوری‌ها، اطلاعات خود را به صورت OData در اختیار مصرف کنندگان قرار می‌دهند مانند Azure table storage، کار با SQL Azure از طریق WCF Data Services (جایی که OData از آن نشأت گرفته شده)، Azure Data Market (برای ارائه فیدهای از اطلاعات خصوصاً رایگان)، ابزارهای گزارش‌گیری مانند SQL Server reporting services، لیست‌های شیرپوینت و غیره. به این ترتیب به بسیاری از قابلیت‌های دنیای NoSQL مانند کار با اطلاعات JSON بدون ترک دنیای رابطه‌ای می‌توان دسترسی داشت.

5 امکان اجرای MongoDB و امثال آن روی سکوی کاری Azure

امکان توزیع MongoDB بر روی یک Worker role سکوی کاری Azure وجود دارد. در این حالت بانک‌های اطلاعاتی این سیستم‌ها بر روی Azure Blob Storage قرار می‌گیرند که به آن‌ها Azure drive نیز گفته می‌شود. همین روش برای سایر بانک‌های اطلاعاتی NoSQL نیز قابل اجرا است. به علاوه امکان اجرای Hadoop نیز بر روی Azure وجود دارد. مایکروسافت به کمک شرکتی به نام HortonWorks نسخه ویندوزی Hadoop را توسعه داده‌اند. HortonWorks را افرادی تشکیل داده‌اند که پیشتر در شرکت یاهو بر روی پروژه Hadoop کار می‌کرده‌اند.

6 قابلیت‌های فرا رابطه‌ای SQL Server

الف) فیلدهای XML (که در ابتدای این مطلب به آن پرداخته شد). به این ترتیب می‌توان به یک اسکیمای انعطاف پذیر، بدون از دست دادن ضمانت ACID رسید.

ب) فیلد HierarchyId برای ذخیره سازی اطلاعات چند سطحی. برای مثال در بانک‌های اطلاعاتی NoSQL سندگرا، یک سند می‌تواند سند دیگری را در خود ذخیره کند و الی آخر.

ج) Sparse columns؛ ستون‌های اسپارس تقریباً شبیه به Key-value stores عمل می‌کنند و یا حتی Wide column stores نیز با آن قابل مقایسه است. در اینجا هنوز اسکیمای وجود دارد، اما برای نمونه علت استفاده از Wide column stores این نیست که واقعا نمی‌دانید ساختار داده‌های مورد استفاده چیست، بلکه در این حالت می‌دانیم که در هر ردیف تنها از تعداد معدودی از فیلدها استفاده خواهیم کرد. به همین جهت در هر ردیف تمام فیلدها قرار نمی‌گیرند، چون در اینصورت تعدادی از آن‌ها همواره خالی باقی می‌مانند. مایکروسافت این مشکل را با ستون‌های اسپارس حل کرده است؛ در اینجا هر چند ساختار کلی مشخص است، اما مواردی که هر بار استفاده می‌شوند، تعداد محدودی می‌باشند. به این صورت SQL Server تنها برای ستون‌های دارای مقدار، فضایی را اختصاص می‌دهد. به این ترتیب از لحاظ فیزیکی و ذخیره سازی نهایی، به همان مزیت Wide column stores خواهیم رسید.

د) FileStreams در اسی کیوال سرور بسیار شبیه به پیوست‌های بانک‌های اطلاعاتی NoSQL سندگرا هستند. در اینجا نیز اطلاعات در فایل سیستم ذخیره می‌شوند اما ارجاعی به آن‌ها در جداول مرتبط وجود خواهند داشت.

7 SQL Server Parallel Data Warehouse Edition

SQL PDW، نگارش خاصی از SQL Server است که در آن یک شبکه از SQL Serverها به صورت یک وهله منطقی SQL Server در اختیار برنامه نویس‌ها قرار می‌گیرد.

این نگارش، از فناوری خاصی به نام MPP یا massively parallel processing برای پردازش کوئری‌ها استفاده می‌کند. در اینجا همانند بانک‌های اطلاعاتی NoSQL، یک کوئری به نود اصلی ارسال شده و به صورت موازی بر روی تمام نودها پردازش گردیده (همان مفهوم Map Reduce که پیشتر در مورد آن بحث شد) و نتیجه در اختیار مصرف کننده قرار خواهد گرفت. نکته مهم آن نیز در عدم نیاز به نوشتن کدی جهت رخ دادن این عملیات از طرف برنامه نویس‌ها است و موتور پردازشی آن جزئی از سیستم اصلی است. تنها کافی است یک کوئری SQL صادر گردد تا نتیجه نهایی از تمام سرورها جمع آوری و بازگردانده شود.

این نگارش ویژه تنها به صورت یک Appliance به فروش می‌رسد (به صورت سخت افزار و نرم افزار باهم) که در آن CPUها،

فضاهای ذخیره سازی اطلاعات و جزئیات شبکه به دقت از پیش تنظیم شده‌اند.

در یک برنامه فروشگاه، جداول مشتری و خریدهای او را در نظر بگیرید. خرید 3 سال قبل مشتری خاصی به آدرس قبلی او ارسال شده‌است. خرید امروز او به آدرس جدید او ارسال خواهد شد. سؤال: آیا با وارد کردن و به روز رسانی آدرس جدید مشتری، باید سابقه اطلاعاتی قبلی او حذف شود؟ اجناس ارسالی پیشین او، واقعا به آدرس دیگری ارسال شده‌اند و نه به آدرس جدید او. چگونه باید اینگونه اطلاعاتی را که در طول زمان تغییر می‌کنند، در بانک‌های اطلاعاتی رابطه‌ای نرمال شده مدیریت کرد؟ از این نمونه‌ها در دنیای کاری واقعی بسیارند. برای مثال قیمت اجناس نیز چنین وضعی را دارند. یک بستنی مگنوم، سال قبل 300 تومان بود؛ امسال شده است 1500 تومان. یک سطل ماست 2500 تومان بود؛ امروز همان سطل ماست 6500 تومان است. چطور باید سابقه فروش این اجناس را نگهداری کرد؟

منابع مطالعاتی مرتبط

[این موضوع](#) اینقدر مهم است که تابحال چندین کتاب در مورد آن تالیف شده است:

[Temporal Data & the Relational Model](#)

[Trees and Hierarchies in SQL](#)

[Developing Time-Oriented Database Applications in SQL](#)

[Temporal Data: Time and Relational Databases](#)

[Temporal Database Entries for the Springer Encyclopedia of Database Systems](#)

[Temporal Database Management](#)

نکته مهمی که در این مآخذ وجود دارند، واژه کلیدی « [Temporal data](#) » است که می‌تواند در جستجوهای اینترنتی بسیار مفید واقع شود.

بررسی ابعاد زمان

فرض کنید کارمندی را استخدام کرده‌اید که ساعتی 2000 تومان از ابتدای فروردین ماه حقوق دریافت می‌کند. حقوق این شخص از ابتدای مهرماه قرار است به ساعتی 2400 تومان افزایش یابد. اگر مامور مالیات در بهمن ماه در مورد حقوق این شخص سؤال پرسید، ما چه پاسخی را باید ارائه دهیم؟ قطعاً در بهمن ماه عنوان می‌کنیم که حقوقش ساعتی 2400 تومان است؛ اما واقعیت این است که این عدد از ابتدای استخدام او ثابت نبوده است و باید تاریخچه تغییرات آن، در نحوه محاسبه مالیات سال جاری لحاظ شود.

بنابراین در مدل سازی این سیستم به دو زمان نیاز داریم:

الف) actual time یا زمان رخ دادن واقعه‌ای. برای مثال حقوق شخصی در تاریخ ابتدای مهر ماه تغییر کرده است. به این تاریخ در منابع مختلف Valid time نیز گفته می‌شود.

ب) record time یا زمان ثبت یک واقعه؛ مثلاً زمان پرداخت حقوق. به آن Transaction time هم گفته شده است. یک مثال:

record date	actual date	حقوق دریافتی
1392/01/01	1392/01/01	روز/2000
1392/02/01	1392/01/01	روز/2000
1392/07/01	1392/07/01	روز/2400
1392/17/01	1392/07/01	روز/2400

در این لیست، ریز حقوق پرداختی به یک شخص را ملاحظه می‌کنید. actual date ها، زمان‌هایی هستند که حقوق پایه شخص در

آن‌ها تغییر کرده و record date زمان‌هایی هستند که به شخص حقوق داده شده‌است. به ترکیب Valid Time و Transaction Time، اصطلاحاً Bitemporal data می‌گویند.

مشکلات طراحی‌های متداول اطلاعات وابسته به زمان

در طراحی‌های متداول، عموماً یک جدول کارمندان وجود دارد و یک جدول لیست حقوق‌های پرداختی. رکوردهای لیست حقوق‌های پرداختی نیز توسط یک کلید خارجی به اطلاعات هر کارمند متصل است؛ از این جهت که نمی‌خواهیم اطلاعاتی تکراری را در جدول لیست حقوقی ثبت کنیم و طراحی نرمال سازی شده‌ای مدنظر می‌باشد. خوب؛ اول مهرماه حقوق شخصی تغییر کرده است. بنابراین کارمند بخش مالی اطلاعات شخص را به روز می‌کند. با این کار، کل سابقه حقوق‌های پرداختی شخص نیز از بین خواهد رفت. چون وجود این کلید خارجی به معنای استفاده از آخرین اطلاعات به روز شده یک کارمند در جدول لیست حقوقی است. الان اگر از جدول لیست حقوقی گزارش بگیریم، کارمندان همواره از آخرین حقوق به روز شده خودشان استفاده خواهند کرد.

راه حل‌های متفاوت مدل سازی اطلاعات وابسته به زمان

برای رفع این مشکل مهم، راه حل‌های متفاوتی وجود دارند که در ادامه آن‌ها را بررسی خواهیم کرد.

الف) نگهداری اطلاعات وابسته به زمان در جدول نهایی مرتبط

اگر حقوق پایه شخص در زمان‌های مختلف تغییر می‌کند، بهتر است عدد نهایی این حقوق پرداختی نیز در یک فیلد مشخص، در همان جدول لیست حقوقی ثبت شود. این مورد به معنای داشتن «داده‌ای تکراری» نیست. از این جهت که داده‌ای تکراری است که اطلاعات آن در تمام زمان‌ها، دارای یک مقدار و مفهوم باشد و اطلاعات حقوق یک شخص اینچنین نیست.

ب) نگهداری اطلاعات تغییرات حقوقی در یک جدول جداگانه

یک جدول ثانویه حقوق جاری کارمندان مرتبط با جدول اصلی کارمندان باید ایجاد شود. در این جدول هر رکورد آن باید دارای بازه زمانی (valid_start_time و valid_end_time) مشخصی باشد. مثلاً از تاریخ X تا تاریخ Y، حقوق کارمند شماره 11، 2000 تومان در ساعت بوده است. از تاریخ H تا تاریخ Z اطلاعات دیگری ثبت خواهند شد. به این ترتیب با گزارشگیری از جدول لیست حقوق‌های پرداخت شده، سابقه گذشته اشخاص محو نشده و هر رکورد بر اساس قرارگیری در یک بازه زمانی ثبت شده در جدول ثانویه حقوق جاری کارمندان تفسیر می‌شود. در این حالت باید دقت داشت که بازه‌های زمانی تعریف شده، با هم تداخل نکنند و برنامه ثبت کننده اطلاعات باید این مساله را به ازای هر کارمند کنترل کند و یا با ثبت record_date، اجازه ثبت بازه‌های تکراری را نیز بدهد (توضیحات در قسمت بعد). به این جدول، یک Temporal table نیز گفته می‌شود. نمونه دیگر آن، نگهداری قیمت یک کالا است از یک تاریخ تا تاریخی مشخص. به این ترتیب می‌توان کوئری گرفت که بستنی مگنوم فروخته شده در ماه آبان سال قبل، بر مبنای قیمت آن زمان، دقیقاً چقدر فروش کرده است و نه اینکه صرفاً بر اساس آخرین قیمت روز این کالا گزارشگیری کنیم که در این حالت اطلاعات نهایی استخراج شده صحیح نیستند. حال اگر به این طراحی در جدولی دیگر Transaction time یا زمان ثبت یک رکورد یا زمان ثبت یک فروش را هم اضافه کنیم، به جدول حاصل Bitemporal Tables می‌گویند.

مدیریت به روز رسانی‌ها در جدول Temporal

در جدول Temporal، حذف فیزیکی اطلاعات مطلقاً ممنوع است؛ چون سابقه سیستم را تخریب می‌کند. اگر اطلاعاتی در این جدول دیگر معتبر نیست باید تنها تاریخ پایان دوره آن به روز شوند یا یک رکورد جدید بر اساس بازه‌ای جدید ثبت گردد. همچنین به روز رسانی‌ها در این جدول نیز معادل هستند با یک Insert جدید به همراه فیلد record_date و نه به روز رسانی واقعی یک رکورد قبلی (شبیه به سیستم‌های حسابداری باید عمل کرد). یک مثال:

فرض کنید حقوق کارمندی که مثال زده شد، در مهرماه به ساعتی 2400 تومان افزایش یافته است و حقوق نهایی نیز پرداخته شده است. بعد از یک ماه مشخص می‌شود که مدیر عامل سیستم گفته بوده است که ساعتی 2500 تومان و نه ساعتی 2400 تومان! (از این نوع مسایل در دنیای واقعی زیاد رخ می‌دهند!) خوب؛ اکنون چه باید کرد؟ آیا باید رفت و رکورد ساعتی 2400 تومان را به روز کرد؟ خیر. چون سابقه پرداخت واقعی صورت گرفته را تخریب می‌کند. به روز رسانی شما ابداً به این معنا نخواهد بود که دریافتی

واقعی شخص در آن تاریخ خاص، ساعتی 2500 بوده است.

بنابراین در جداول Temporal، تنها «تغییرات افزودنی» مجاز هستند و این تغییرات همواره به عنوان آخرین رکورد جدول ثبت می‌شوند. به این ترتیب می‌توان اصطلاحاً «مابه‌التفاوت» حقوق پرداخت نشده را به شخص خاصی، محاسبه و پرداخت کرد (می‌دانیم در یک بازه زمانی خاص به او چقد حقوق داده‌ایم. همچنین می‌دانیم که این بازه در یک record_date دیگر لغو و با عددی دیگر، جایگزین شده‌است).

برای مطالعه بیشتر

[Bitemporal Database Table Design - The Basics](#)

[Temporal Data Techniques in SQL](#)

[Database Design: A Point in Time Architecture](#)

[Temporal database](#)

[Temporal Patterns](#)

راه حلی دیگر؛ استفاده از بانک‌های اطلاعاتی NoSQL

بانک‌های اطلاعاتی NoSQL برخلاف بانک‌های اطلاعاتی رابطه‌ای برای اعمال Read بهینه‌سازی می‌شوند و نه برای Write. در چند دهه قبل که بانک‌های اطلاعاتی رابطه‌ای پدیدار شدند، یک سخت دیسک 10 مگابایتی حدود 4000 دلار قیمت داشته است. به همین جهت مباحث نرمال‌سازی اطلاعات و ذخیره نکردن اطلاعات تکراری تا این حد در این نوع بانک‌های اطلاعاتی مهم بوده است. اما در بانک‌های اطلاعاتی NoSQL امروزی، اگر قرار است فیش حقوقی شخصی ثبت شود، می‌توان کل اطلاعات جاری او را یکجا داخل یک سند ثبت کرد (از اطلاعات شخص در آن تاریخ تا اطلاعات تمام اجزای فیش حقوقی در قالب یک شیء تو در توی JSON). به همین جهت بسیار سریع هستند برای اعمال Read و گزارش‌گیری. همچنین این نوع سیستم‌ها برای نگهداری نگارش‌های مختلف یک سند بهینه‌سازی شده‌اند و جزو ساختار توکار آن‌ها است. بنابراین در این نوع سیستم‌ها اگر نیاز است از یک سند خاصی گزارش بگیریم، دقیقاً اطلاعات همان تاریخ خاص را دارا است و اگر اطلاعات پایه سیستم را به روز کنیم، از امروز به بعد در سندهای جدید ثبت خواهد شد. این نوع سیستم‌ها رابطه‌ای نیستند و بسیاری از مباحث نرمال‌سازی اطلاعات در آن‌ها ضرورتی ندارد. قرار است یک فیش حقوقی شخص را نمایش دهیم؟ خوب، چرا تمام اطلاعات مورد نیاز او را در قالب یک شیء JSON تو در توی حاضر و آماده نداشته باشیم؟

نظرات خوانندگان

نویسنده: ناصر
تاریخ: ۱۳۹۲/۰۷/۱۷ ۰:۵۴

من هم قبلا از این روش برای قیمت‌های جدید و قدیم یک کالا در یک سیستم فروشگاهی استفاده کردم. با نوشتن یک تریگر که به محض تغییر روی قیمت کالا ، بلافاصله داخل یک جدول دیگه این تغییرات درج میشد. و موقع نمایش ، قیمت جدید و قدیم هر دو با هم به مشتری نمایش داده میشد.

نویسنده: سیروس
تاریخ: ۱۳۹۲/۰۷/۱۸ ۱۱:۱۱

به نظر من در نگهداری به این روش نیازی به تاریخ پایان نیست، مثلا هنگام تغییر قیمت کالا، رکوردی با تاریخ روز در جدول temporal ثبت می‌کنیم و در تغییر دوباره رکورد جدید دیگری ثبت می‌شود. کارکردن به این روش آسانتر به نظر می‌رسد و یک فیلد کمتر داریم و نیازی هم به چک کردن درست بودن بازه‌ی تاریخی نیست.

Date	Price	ProdcutId
1392/01/01	1000	1
1392/03/05	1500	1
1392/06/27	1780	1

نویسنده: ایلیا اکبری فرد
تاریخ: ۱۳۹۳/۰۴/۰۲ ۱۱:۴۵

البته روش شما برای حالتی مناسب است که بازه‌های تاریخی به هم متصل باشند.



در مطلب اول هدف فقط آشنایی و نحوه نصب PouchDB قرار خواهد داشت و در مطالب بعدی نحوه آشنایی با نحوه کدنویسی و استفاده به صورت آفلاین یا آنلاین بررسی خواهد شد .

فهرست مطالب :

بخش اول : معرفی PouchDB

شروع به کار با PouchDB

نحوه استفاده از API ها

سوالات متداول در مورد PouchDB

خطاهای احتمالی

پروژه ها و پلاگین های PouchDB

PouchDB یک دیتابیس NoSQL می باشد که به وسیله Javascript نوشته شده و هدف آن این است که برنامه نویسی ها بتوانند برنامه هایی را توسعه و ارائه کنند که بتواند هم به صورت آفلاین و هم آنلاین سرویس دهی داشته باشند. برنامه اطلاعات خودش را به صورت آفلاین ذخیره می کند و کاربر می تواند زمانیکه به اینترنت متصل نیست، از آنها استفاده کند. اما به محض اتصال به اینترنت، دیتابیس خودش را با دیتابیس آنلاین همگام (Sync) می کند. اینجاست که قدرت اصلی PouchDB مشخص می شود. بزرگترین برتری PouchDB همین است. دیتابیسی است که به صورت توکار قابلیت های همگام سازی را دارا می باشد و به صورت اتوماتیک این کار را انجام می دهد. PouchDB یک پروژه ی اوپن سورس است که توسط [این افراد](#) به روز می شود. البته باید گفت که PouchDB از CouchDB الهام گرفته شده است. اگر شما هم قصد همکاری در این پروژه را دارید بهتر است که [راهنمای همکاری](#) را مطالعه کنید .

پشتیبانی مرورگرها

PouchDB پیش زمینه های مختلفی دارد که به آن این امکان را می دهد تا روی همه مرورگرها و صد البته روی NodeJs کار کند. از IndexedDB بر روی Firefox/Chrome/Opera/IE و WebSql بر روی Safari و همچنین LevelDB بر روی NodeJs استفاده می کند. در حال حاضر PouchDB بر روی مرورگرهای زیر تست شده است:

فایرفاکس 12 و بالاتر

گوگل کروم 19 و بالاتر

اپرا 12 و بالاتر

سافاری 5 و بالاتر

اینترنت اکسپلورر 10 و بالاتر

NodeJs 0.10 و بالاتر

و به صورت شگفت انگیزی در Apache Cordova

برای اطلاعات بیشتر در مورد مرورگرهایی که IndexedDB و WebSql را پشتیبانی می کنند به لینک های زیر مراجعه کنید:

[Can I use IndexedDB](#)

[Can I use Web SQL Database](#)

نکته : در صورتی که برنامه شما نیاز دارد تا از اینترنت اکسپلورر نسخه پایینتر از 10 استفاده کند می توانید از دیتابیسی های آنلاین استفاده کنید، که البته دیگر قابلیت استفاده آفلاین را نخواهد داشت.

وضعیت فعلی PouchDB

PouchDB برای مرورگر، فعلا در وضعیت بتا به سر می برد و به صورت فعالی در حال گذراندن تست هایی می باشد تا باگ های آن برطرف شود و به صورت پایدار (Stable) ارائه گردد. البته فقط ممکن است که شما باگی را در قسمت Api ها پیدا کنید که البته Api ها هم در حال حاضر پایدار هستند و گزارشی مبنی بر باگ در آن ها موجود نیست. اگر هم باگی پیدا بشود شما می توانید PouchDB را بدون ریسک از دست رفتن اطلاعات آپگرید کنید.

PouchDB برای NodeJs فعلا در وضعیت آلفا است و آپگرید کردن ممکن است به اطلاعات شما آسیب بزند. البته با آپدیت به صورت دستی خطری شما را تهدید نخواهد کرد .

نحوه‌ی نصب PouchDB

PouchDB به صورت یک کتابخانه‌ی کوچک و جمع و جور طراحی شده است تا بتواند همه نیازها را برطرف و روی همه نوع Device اعم از موبایل، تبلت، مرورگر و کلا هر چیزی که جاوا اسکریپت را ساپورت می‌کند کار خود را به خوبی انجام بدهد.

برای استفاده از PouchDB میبایست [این فایل را با حجم فوق العاده 97 کیلوبایت دانلود کنید](#) و آن را به یک صفحه وب اضافه کنید :

```
<script src="pouchdb-2.1.0.min.js"></script>
```

آخرین نسخه و بهترین نسخه : [pouchdb-2.1.0.min.js](#)

برای اطلاع از آخرین آپدیتها و نسخه‌ها به [این صفحه در گیت هاب](#) مراجعه کنید .
برای کسانی هم که از NodeJS استفاده می‌کنند نحوه نصب به این صورت است :

```
$ npm install pouchdb
```

نظرات خوانندگان

نویسنده: سعیدزمان
تاریخ: ۱۱:۲۰ ۱۳۹۳/۰۱/۲۶

سلام مهندس مطلب بسیار جالبی بود فقط من یک سوال برام پیش اومده که این اطلاعات که میخواد در حالت افلاین استفاده بشه کجا قرار میگیره ؟ ایا امنیت داده های حساس رو پایین نمی یاره ؟
با تشکر

نویسنده: محمد رضا صفری
تاریخ: ۱۵:۱۵ ۱۳۹۳/۰۱/۲۶

پیشنهاد می کنم این اسلاید هارو ببینید : <http://www.slideshare.net/wurbanski/nosql-no-security>
<http://www.slideshare.net/gavinholt/nosql-no-security-16514872>
واقعا مفیده و خیلی از سوالات شمارو پاسخ میده .
موفق باشید ./

در طی این [پست ها](#) با مفاهیم NoSql آشنا شدید. همچنین در این [دوره](#) مفاهیم و مبانی RavenDb (یکی از بی نقص‌ترین دیتابیس‌های NoSql) بررسی شد. اما قرار است در طی چند پست با یکی دیگر از انواع دیتابیس‌های NoSql طراحی شده برای دات نت به نام BrightStarDb یا به اختصار B*Db آشنا شویم.

*در دنیای NoSql پیاده سازی‌های متفاوتی از دیتابیس‌ها انجام شده است و هر دیتابیس، ویژگی‌ها و مزایا و معایب خاص خودش را دارد. باید قبول کرد که همیشه و همه جا نمی‌توان از یک پایگاه داده NoSql مشخص استفاده نماییم (دلایلی نظیر محدودیت‌های License، هزینه پیاده سازی و...). در نتیجه بررسی یک دیتابیس دیگر با شرایط و توانمندی‌های خاص آن خالی از سود نیست. از ویژگی مهم این دیتابیس می‌توان به عناوین زیر اشاره کرد.

« این دیتابیس در گروه **Graph databases** قرار دارد و از زبان [SPARQL](#) (بخوانید Sparkle) برای کوئری گرفتن و کار با داده‌ها بهره می‌برد؛

« متن باز و رایگان است

« پشتیبانی از دات نت چهار به بعد؛

« قابل استفاده در 7 , 8 Windows phone - Mobile Application؛

« بدون شما (Schema Less) و با قابلیت ذخیره سازی triple و به فرمت RDF

« پشتیبانی از Linq و OData؛

« پشتیبانی از تراکنش‌ها ؛

« پیاده سازی مدل برنامه به صورت Code First؛

« سرعت بالا جهت ذخیره سازی و لود اطلاعات؛

« نیاز به پیکربندی‌های خاص جهت پیاده سازی ندارد؛

« ارائه افزونه رایگان Polaris جهت کوئری گفتن و نمایش Visual داده ها.

و غیره که در ادامه با آن‌ها آشنا خواهید شد.

در B*Db دو روش برای ذخیره سازی اطلاعات وجود دارد:

« **Append Only** : در این روش تمامی تغییرات (عملیات نوشتن) در انتهای فایل index اضافه خواهد شد. این روش مزایای زیر را به دنبال خواهد داشت:

عملیات نوشتن هیچگاه عملیات خواندن اطلاعات را block نمی‌کند. در نتیجه هر تعداد عملیات خواندن فایل (منظور اجرای کوئری‌های SPQRL است) می‌تواند به صورت موازی بر روی Store‌ها اجرا شود.

به دلیل اینکه عمل ویرایش واقعی هیچ گاه انجام نمی‌شود (داده‌ها فقط اضافه خواهند شد) همیشه می‌توانید وضعیت Store را به حالت‌های قبلی بازگردانید.

عملیات نوشتن اطلاعات بسیار سریع خواهد بود.

از معایب این روش این است که حجم Store‌ها فقط با افزایش داده‌ها زیاد نمی‌شود، بلکه با هر عملیات ویرایش نیز حجم فایل‌های Store افزایش پیاده خواهد کرد. در نتیجه از این روش فقط زمانی که از نظر فضای هارد دیسک محدودیت ندارید استفاده کنید (روش پیش فرض در B*Db نیز همین حالت است)

« **Rewritable** : در این روش در هنگام اجرای عملیات نوشتن، ابتدا یک کپی از اطلاعات گرفته می‌شود. سپس داده‌های مورد نظر به کپی گرفته شده اعمال می‌شوند. تا زمانیکه عملیات نوشتن اطلاعات به پایان نرسد، هر گونه دسترسی به اطلاعات جهت عملیات Read بر روی داده اصلی اجرا می‌شود. با استفاده از این روش عملیات Read و Write هیچ گونه تداخلی با هم نخواهند داشت.

(چیزی شبیه به [^](#))

نکته ای که باید به آن دقت داشت این است که فقط در هنگام ساخت Storeها می‌توانید نوع ذخیره سازی آن را تعیین نمایید، بعد از ساخت Store امکان سوئیچ بین حالات امکان پذیر نیست.

همان طور که پیشتر گفته شد B*Db برای ذخیره سازی اطلاعات از سند RDF بهره می‌برد. البته با RDF Syntaxهای متفاوت :

RDF Syntax	File Extension (uncompressed)	File Extension (GZip compressed)
NTriples	.nt	.nt.gz
NQuads	.nq	.nq.gz
RDF/XML	.rdf	.rdf.gz
Turtle	.ttl	.ttl.gz
RDF/JSON	.rj or .json	.rj.gz or .json.gz

هم چنین در B*Db چهار روش برای دست یابی با داده‌ها (پیاده سازی عملیات CRUD) وجود دارد از قبیل:

« B*Db EntityFramework

« Data Object Layer

« RDF Client Api

« Dynamic API

که هر کدام در طی پست‌های متفاوت بررسی خواهد شد.

بررسی یک مثال با روش B*Db EntityFramework

برای شروع ابتدا یک پروژه جدید از نوع Console Application ایجاد کنید. سپس از طریق Nuget اقدام به نصب Package زیر نمایید:

```
pm> install-Package BrightStarDb
```

پکیج بالا تمام کتابخانه‌های لازم جهت کار با B*Db را شامل می‌شود. اگر قصد ندارید از افزونه‌های مربوط به EntityFramework و Code First استفاده نمایید می‌توانید Package زیر را نصب نمایید:

```
PM> Install-Package BrightStarDbLibs
```

این پکیج فقط شامل کتابخانه‌های لازم جهت کار با استفاده از SPRQL است.

بعد از نصب پکیج‌های بالا یک فایل Text Template با نام MyEntityContext.tt نیز به پروژه افزوده خواهد شد. این فایل جهت تولید خودکار مدل‌های برنامه استفاده می‌شود. اما برای این کار لازم است به ازای هر مدل ابتدا یک اینترفیس ایجاد نمایید. برای مثال:

```
[Entity]
public interface IBook
{
    public int Code { get; set; }
    public string Title { get; set; }
```

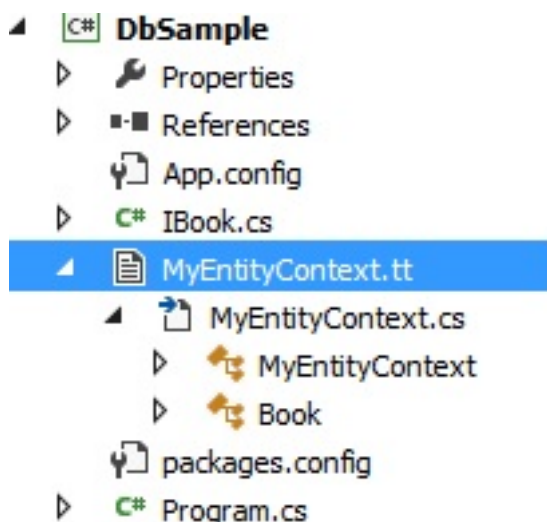
}

نکته:

« نیاز به ایجاد یک خاصیت به عنوان Id وجود ندارد. به صورت پیش فرض خاصیت Id با نوع string برای هر مدل پیاده سازی می‌شود. اما اگر قصد دارید این نام خاصیت را تغییر دهید می‌توانید به صورت زیر عمل کنید:

```
[Entity]
public interface IBook
{
    [Identifier]
    public string MyId { get; }
    public int Code { get; set; }
    public string Title { get; set; }
}
```

در مثال بالا خاصیت MyId به جای خاصیت Id در نظر گرفته می‌شود. مزین شدن با Identifier و همچنین نداشتن متد set را فراموش نکنید. بعد از ایجاد اینترفیس مورد نظر و اجرای Run Custom Tool بر روی فایل Text Template.tt کلاسی به نام Book به صورت زیر ساخته می‌شود:



استفاده از اینترفیس برای ساخت مدل انعطاف پذیری بالایی را در اختیار ما قرار می‌دهد که بعداً مفصل بحث خواهد شد. برای عملیات درج داده می‌توان به صورت زیر عمل کنید:

```
MyEntityContext context = new
MyEntityContext("type=embedded;storedirectory=c:\brightstar;storename=test");
var book = context.Books.Create();
book.Code = 1;
book.Title = "Test";

context.Books.Add(book);

context.SaveChanges();
```

با یک نگاه می‌توان به شباهت مدل پیاده سازی شده بالا به EntityFramework پی برد. اما نکته مهم در مثال بالا Connection String پاس داده شده به Context پروژه است. در B*Db چهار روش برای دستیابی به اطلاعات ذخیره شده وجود دارد:

« embedded : در این حالت از طریق آدرس فیزیکی فایل مورد نظر می‌توان یک Connection ایجاد کرد.

«rest : یا استفاده از HTTP یا HTTPS می‌توان به سرویس B*Db دسترسی داشت.

«dotNetRdf : برای ارتباط با یک Store دیگر با استفاده از اتصال دهنده‌های DotNetRdf.

«sparql : اتصال به منبع داده ای دیگر با استفاده از پروتکل SPARQL

در هنگام ایجاد اتصال باید نوع مورد نظر را از حتما تعیین نمایید. با استفاده از storedirectory مکان فیزیکی فایل تعیین خواهد شد.

در این آموزش هدف ما ایجاد برنامه‌ای بر اساس [TodoMVC](#) است که می‌تواند خودش را با یک دیتابیس آنلاین همگام سازی کند.

مطمئن باشید بیشتر از 10 دقیقه وقت شمارا نخواهد گرفت !

نصب PouchDB

فایل index.html را باز کنید و فایل‌های PouchDB را به آن اضافه کنید :

```
<script src="//cdn.jsdelivr.net/pouchdb/2.2.0/pouchdb.min.js"></script>
<script src="js/base.js"></script>
<script src="js/app.js"></script>
```

حالا PouchDB نصب شده و آماده به کار است . (در نسخه نهایی بهتر است از فایل‌های local استفاده کنید)

ایجاد بانک اطلاعاتی

بقیه کارها باید در فایل app.js انجام شود . برای شروع باید بانک اطلاعاتی جدیدی را برای درج اطلاعات خودمان ایجاد کنیم . برای ایجاد بانک اطلاعاتی خیلی ساده یک instance جدید از آبجکت PouchDB می‌سازیم .

```
var db = new PouchDB('todos');
var remoteCouch = false;
```

نیازی نیست که برای بانک خود، نما (Schema) ایجاد کنید! بعد از اینکه اسم را مشخص کنید، بانک آماده به کار است.

ثبت اطلاعات در بانک اطلاعاتی

اولین کاری که باید انجام دهیم این است که یک متد را ایجاد کنیم و توسط آن اطلاعات خودمان را در بانک اطلاعاتی ذخیره کنیم. نام متد را addTodo انتخاب می‌کنیم و کارش این است که وقتی کاربر کلید Enter را فشار داد، اطلاعات را داخل بانک اطلاعاتی ذخیره کند. متد مورد نظر به این صورت هست:

```
function addTodo(text) {
  var todo = {
    _id: new Date().toISOString(),
    title: text,
    completed: false
  };
  db.put(todo, function callback(err, result) {
    if (!err) {
      console.log('Successfully posted a todo!');
    }
  });
}
```

در PouchDB هر پرونده نیاز دارد تا یک فیلد unique با نام _id داشته باشد. اگر داده‌ای بخواهد در بانک اطلاعاتی ثبت شود و دارای _id مشابه باشد، با آن به صورت یک update رفتار خواهد شد. در اینجا ما از تاریخ با عنوان id استفاده کردیم که در این مورد خاص می‌باشد. شما می‌تواند از db.post() نیز استفاده کنید؛ اگر یک id را به صورت random می‌خواهید. تنها چیزی که اجباری است در هنگام ساختن یک پرونده، همین _id است و بقیه موارد کاملاً اختیاری هستند.

نمایش اطلاعات

در اینجا ما از یک function کمکی به نام redrawTodosUI استفاده خواهیم کرد که وظیفه‌اش این است تا یک array را دریافت

کرده و آن را هر طور که مشخص کردید نمایش دهد. البته آن را به سلیقه خودتان می‌توانید آماده کنید. تنها کاری که باید انجام دهیم این است که اطلاعات را از بانک اطلاعاتی استخراج کرده و به function مورد نظر پاس دهیم. در اینجا می‌توانیم به سادگی از db.allDocs برای خواندن اطلاعات از بانک اطلاعاتی استفاده کنیم. خاصیت include_docs به PouchDB این دستور را می‌دهد که ما درخواست دریافت همه اطلاعات داخل پرونده‌ها را داریم و descending هم نحوه مرتب سازی را که بر اساس id_ هست، مشخص می‌کند تا بتوانیم جدیدترین اطلاعات را اول دریافت کنیم.

```
function showTodos() {
  db.allDocs({include_docs: true, descending: true}, function(err, doc) {
    redrawTodosUI(doc.rows);
  });
}
```

به روزرسانی خودکار

هر بار که ما اطلاعات جدیدی را در بانک اطلاعاتی وارد می‌کنیم، نیازمند این هستیم تا اطلاعات جدید را به صورت خودکار دریافت و نمایش دهیم. برای این منظور می‌توانیم به روش زیر عمل کنیم :

```
var remoteCouch = false;

db.info(function(err, info) {
  db.changes({
    since: info.update_seq,
    live: true
  }).on('change', showTodos);
});
```

حالا هر بار که اطلاعات جدیدی در بانک اطلاعات ثبت شود، به صورت خودکار نمایش داده خواهد شد. خاصیت live مشخص می‌کند که این کار می‌تواند بی نهایت بار انجام شود.

ویرایش اطلاعات

وقتی که کاربر یک آیتم Todo را کامل می‌کند، یک چک باکس را علامت می‌زند و اعلام می‌کند که این کار انجام شده است.

```
function checkboxChanged(todo, event) {
  todo.completed = event.target.checked;
  db.put(todo);
}
```

این بخش شبیه به قسمت ثبت اطلاعات است، با این تفاوت که باید شامل یک فیلد rev_ (اضافه بر id_) نیز باشد. در غیر اینصورت تغییرات ثبت نخواهد شد. این کار برای این است که اشتباهی، اطلاعاتی در بانک ثبت نشود.

حذف اطلاعات

برای حذف اطلاعات باید از متد db.remove به همراه آبجکت مورد نظر استفاده کنید .

```
function deleteButtonPressed(todo) {
  db.remove(todo);
}
```

مانند بخش ویرایش نیز باید در اینجا هم id_ و هم rev_ مشخص شود. باید توجه داشته باشید در اینجا همان آبجکتی را که از بانک فراخوانی کرده‌ایم، به این متد پاس می‌دهیم. البته شما می‌توانید آبجکت خودتان را نیز ایجاد کنید {id: todo._id, _rev: todo._rev_} اما خوب همان روش قبلی عاقلانه‌تر است و احتمال خطای کمتری دارد .

نصب CouchDB

حالا می‌خواهیم همگام سازی را اجرا کنیم و برای این کار نیاز هست یا [CouchDB را به صورت Local نصب کنیم](#) یا از سرویس‌های آنلاین مثل [IrisCouch](#) استفاده کنید .

فعال سازی CORS

برای اینکه به صورت مستقیم با CouchDB در ارتباط باشید باید مطمئن شوید که CORS فعال هست. در اینجا فقط نام کاربری و رمز ورود را مشخص کنید. به صورت پیش فرض CouchDB به صورت Admin Party نصب می شود و نیازی به User و password ندارد. مگر اینکه برایش مشخص کنید. همچنین شما باید myname.iriscouch.com را با سرور خودتان جایگزین کنید که اگر به صورت local کار می کنید 127.0.0.1:5984 است.

```
$ export HOST=http://username:password@myname.iriscouch.com
$ curl -X PUT $HOST/_config/httpd/enable_cors -d '"true"'
$ curl -X PUT $HOST/_config/cors/origins -d '"*"'
$ curl -X PUT $HOST/_config/cors/credentials -d '"true"'
$ curl -X PUT $HOST/_config/cors/methods -d '"GET, PUT, POST, HEAD, DELETE"'
$ curl -X PUT $HOST/_config/cors/headers -d \
  '"accept, authorization, content-type, origin"'
```

راه اندازی ارتباط ساده دو طرفه

به فایل app.js برگردید . در اینجا باید آدرس بانک اطلاعاتی آنلاین را مشخص کنیم.

```
var db = new PouchDB('todos');
var remoteCouch = 'http://user:pass@mname.iriscouch.com/todos';
```

فراموش نکنید که نام کاربری و رمز ورود را بسته به نیاز خود تغییر دهید. حالا می توانیم function که وظیفه همگام سازی اطلاعات را به عهده دارد بنویسیم :

```
function sync() {
  syncDom.setAttribute('data-sync-state', 'syncing');
  var opts = {live: true};
  db.replicate.to(remoteCouch, opts, syncError);
  db.replicate.from(remoteCouch, opts, syncError);
}
```

db.replicate() به PouchDB می گوید که همه اطلاعات را "به" یا "از" remoteCouch انتقال دهد. ما دوبار این درخواست را دادیم. در بار اول اطلاعات به داخل سرور ریموت منتقل می شود و در بار دوم اطلاعات local جایگزین می شوند.

یک callback هم وقتی که این کار به پایان برسد اجرا خواهد شد . می توانید برنامه خود را در دو مرورگر مختلف اجرا کنید تا نتیجه کار را ببینید.

تبریک می گوئیم !

شما توانستید اولین برنامه خود را توسط PouchDB ایجاد کنید. البته این یک برنامه ساده بود و در دنیای واقعی نیاز هست تا خیلی کارهای بیشتری را انجام دهید. اما باز هم اصول اولیه کار را یاد گرفتید و ادامه راه را می توانید تنهایی ادامه دهید .

در این پست با BrightStarDb و مفاهیم اولیه آن آشنا شدید. همان طور که پیش‌تر ذکر شد BrightStarDb از تراکنش‌ها جهت ذخیره اطلاعات پشتیبانی می‌کند. قصد داریم روش شرح داده شده در [اینجا](#) را بر روی BrightStarDb فعال کنیم. ابتدا بهتر است با روش ساخت مدل در B*Db آشنا شویم.

*یکی از پیش‌نیازهای این پست مطالعه این دو مطلب ([_](#)) و ([_](#)) می‌باشد.

فرض می‌کنیم در دیتابیس مورد نظر یک Store به همراه یک جدول به صورت زیر داریم:

```
[Entity]
public interface IBook
{
    [Identifier]
    string Id { get; }

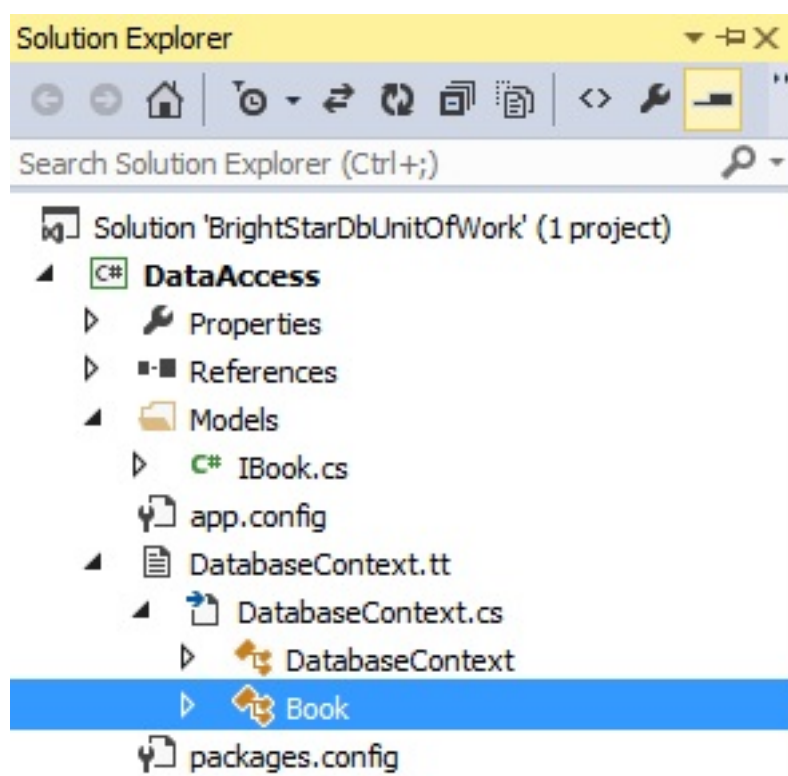
    string Title { get; set; }

    string Isbn { get; set; }
}
```

بر روی پروژه مورد نظر کلیک راست کرده و گزینه Add new Item را انتخاب نمایید. از برگه Data گزینه BrightStar Entity را انتخاب کنید



بعد از انتخاب گزینه بالا یک فایل با پسوند tt به پروژه اضافه خواهد شد که وظیفه آن جستجو در اسمبلی مورد نظر و پیدا کردن تمام اینترفیس‌هایی که دارای EntityAttribute هستند و همچنین ایجاد کلاس‌های متناظر جهت پیاده‌سازی اینترفیس‌های بالا است. در نتیجه ساختار پروژه تا این جا به صورت زیر خواهد شد.



واضح است که فایلی به نام Book به عنوان پیاده سازی مدل IBook به عنوان زیر مجموعه فایل DbContext.tt به پروژه اضافه شده است.

تا اینجا برای استفاده از Context مورد نظر باید به صورت زیر عمل نمود:

```
DatabaseContext context = new DatabaseContext();
context.Books.Add(new Book());
```

Context پیش فرض ساخته شده توسط B*Db از DbSet های Generic معادل EF پشتیبانی نمی کند و از طرفی IUnitOfWork مورد نظر به صورت زیر است

```
public interface IUnitOfWork
{
    BrightstarEntitySet<T> Set<T>() where TEntity : class;
    void DeleteObject(object obj);
    void SaveChanges();
}
```

در اینجا فقط به جای DbSet از BrightStarDbSet استفاده شده است. همان طور که در این [مقاله](#) توضیح داده شده است، برای پیاده سازی مفهوم UnitOfWork نیاز است تا کلاس DatabaseContext که نماینده BrightStarDbContext پروژه است، از اینترفیس IUnitOfWork طراحی شده ارث بری کند. جهت انجام این مهم و همچنین جهت اضافه کردن قابلیت ایجاد Generic DbSet ها نیز باید کمی در فایل Template Generator تغییر ایجاد نماییم. این تغییرات را قبلاً در طی یک پروژه ایجاد کرده ام و شما می توانید آن را از [اینجا](#) دریافت کنید. بعد از دانلود کافیتست فایل DbContext.tt مورد نظر را در پروژه خود کپی کرده و گزینه Run Custom Tools را فراخوانی نمایید.

نکته: برای حذف یک آبجکت از Store، باید از متد DeleteObject تعبیه شده در Context استفاده نماییم. در نتیجه متد مورد نظر نیز در اینترفیس بالا در نظر گرفته شده است.

استفاده از IOC Container جهت رجیستر کردن IUnitOfWork

در این قدم باید IUnitOfWork را در یک IOC container رجیستر کرده تا در جای مناسب عملیات وهله سازی از آن میسر باشد. من در اینجا از [Castle Windsor Container](#) استفاده کردم. کلاس زیر این کار را برای ما انجام خواهد داد:

```
public class DependencyResolver
{
    public static void Resolve(IWindsorContainer container)
    {
        var context = new
DatabaseContext("type=embedded;storedirectory=c:\brightstar;storename=test ");
        container.Register(Component.For<IUnitOfWork>().Instance(context).LifestyleTransient());
    }
}
```

حال کافیت در کلاس‌های سرویس برنامه UnitOfWork رجیستر شده را به سازنده آن‌ها تزریق نماییم.

```
public class BookService
{
    public BookService(IUnitOfWork unitOfWork)
    {
        UnitOfWork = unitOfWork;
    }

    public IUnitOfWork UnitOfWork
    {
        get;
        private set;
    }

    public IList<IBook> GetAll()
    {
        return UnitOfWork.Set<IBook>().ToList();
    }

    public void Add()
    {
        UnitOfWork.Set<IBook>().Add(new Book());
    }

    public void Remove(IBook entity)
    {
        UnitOfWork.DeleteObject(entity);
    }
}
```

سایر موارد دقیقاً معادل مدل EF آن است.

نکته: در حال حاضر امکان جداسازی مدل‌های برنامه (تعاریف اینترفیس) در قالب یک پروژه دیگر (نظیر مدل CodeFirst در EF) در B*Db امکان پذیر نیست.

نکته : برای اضافه کردن آیتم جدید به Store نیاز به وهله سازی از اینترفیس IBook داریم. کلاس Book ساخته شده توسط DatabaseContext.tt در عملیات Insert و update کاربرد خواهد داشت.