عنوان: **Full Text Search و Rank فیلدهای بازیابی شده**
نویسنده: دل محسن
تاریخ: ۱۳۹۲/۰۳/۱۰ ۱۰:۵۵
آدرس: www.dotnettips.info
گروه‌ها: SQL Server, Full Text Search

حالتی را در نظر بگیرید که بخواهید تعداد زیادی رکورد را که حجم هر رکورد هم قابل ملاحظه هست، نگهداری کنید(مثلا چندین هزار مقاله) و همچنین قابلیت جستجو را در این رکوردها لحاظ کنید به صورتی که بر اساس رکوردهایی که بیشترین تعداد تکرار کلمات مدنظر را دارند مرتب شوند.
شاید اولین راه حل، مطلب آقای سلیم آبادی در اینجا باشه، که تعداد تکرار یک کلمه را در فیلدی در جدول بیان کردند و درست هم هست اما با 2 شرط:
1) رکوردهای ما حجم کمی داشته باشند چرا که دستور LIKE پاسخ سریعی را با حجم بالای اطلاعات به ما نمی‌دهد.
2) رکوردهای ما از خانواده‌ی char باشند. برای مثال اگر مقالات ما به صورت pdf باشند این کد جواب نمی‌دهد.

اما راه دوم استفاده از Ful Text Search و دستور CONTAINSTABLE هست که 2 شرط لازم را برای راه حل اول احتیاج نداره. در اینجا فقط نحوه‌ی استفاده از CONTAINSTABLE رو مطرح میکنیم.

CONTAINSTABLE جدولی از موارد یافت شده را بر اساس معیارهایی که ما به اون معرفی می‌کنیم، ایجاد می‌کند. این جدول حاوی دو فیلد KEY(کلید فیلد مورد نظر) و RANK(مقداری بین 0 تا 1000) است که میزان همسانی رکورد با معیار ما را مشخص می‌کند و ما با استفاده از این فیلد می‌توانیم رکوردهایمان را مرتب کنیم.

به این کد توجه کنید:

```
SELECT t.Title, p.[RANK]
    FROM Articles AS t
        INNER JOIN CONTAINSTABLE(Articles, Data, 'management' ) AS p
            ON t.Id = p.[KEY]
ORDER BY p.RANK
```

در اینجا کار جستجو انجام شده و بر حسب میزان نزدیکی محتویات رکورد با معیار ما مرتب شده است.
نکته: هیچ فرقی نمی‌کند که محتویات فیلد مورد نظر شما یک متن ساده، یک فایل word یا حتی pdf باشد. فقط باید تنظیمات Full Text Search درست انجام شود.

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۳/۱۰ ۲۲:۴۹

برای PDF که عنوان کردید، آیا جستجوی فارسی کار می‌کند؟ از چه IFilter ایی برای اینکار استفاده کردید؟

نویسنده: دل محسن
تاریخ: ۱۳۹۲/۰۳/۱۱ ۱۴:۵۰

بله کار میکنه. من با Adob IFilter(رایگان) و Foxit IFilter(تجاری هست) کار کردم.

نویسنده: علی
تاریخ: ۱۳۹۲/۰۶/۰۶ ۱۲:۲۲

سلام.
من با rank کار می‌کنم توی پروژه هام.
منتها خیلی وقت‌ها رکوردهایی رنک بالایی می‌گیرند در حالی که رکوردهای مشابه دیگر رنک پایینی می‌گیرند. البته کمتر از 5 درصد اتفاق می‌افته
در مورد خود رنک مطلب خوبی پیدا نکردم که اصلا خود مفهوم رنک رو برام توضیح بده

نویسنده: علی
تاریخ: ۱۳۹۲/۱۰/۲۴ ۱۹:۲۷

منم توی این رنک بدجور گیر کردم. به جابجایی کامات حساسه . مثلا «خواجه علی» و «علی خواجه ».

عنوان: **نگاهی به Latent Semantic Indexing**
نویسنده: حامد خسروجردی
تاریخ: ۱۳۹۳/۰۳/۰۹ ۱۲:۵۵
آدرس: www.dotnettips.info
گروه‌ها: Lucene.NET, Index, Full Text Search, search

مقدمه ای بر Latent Semantic Indexing

هنگامیکه برای اولین بار، جستجو بر مبنای کلمات کلیدی (keyword search) بر روی مجموعه‌ای از متون، به دنیای بازیابی اطلاعات معرفی شد شاید فقط یک ذهنیت مطرح می‌شد و آن یافتن لغت در متن بود. به بیان دیگر در آن زمان تنها بدنبال متونی می‌گشتیم که دقیقا شامل کلمه کلیدی مورد جستجوی کاربر باشند. روال کار نیز بدین صورت بود که از دل پرس و جوی کاربر، کلماتی بعنوان کلمات کلیدی استخراج می‌شد. سپس الگوریتم جستجو در میان متون موجود بدنبال متونی می‌گشت که دقیقا یک یا تمامی کلمات کلیدی در آن آمده باشند. اگر متنی شامل این کلمات بود به مجموعه جواب‌ها اضافه می‌گردید و در غیر این صورت حذف می‌گشت. در پایان جستجو با استفاده از الگوریتمی، نتایج حاصل رتبه بندی می‌گشت و به ترتیب رتبه با کاربر نمایش داده می‌شد.
نکته مهمی که در این روش دیده می‌شود اینست که متون به تنهایی و بدون در نظر گرفتن کل مجموعه پردازش می‌شدند و اگر تصمیمی مبنی بر جواب بودن یک متن گرفته می‌شد، آن تصمیم کاملا متکی به همان متن و مستقل از متون دیگر گرفته می‌شد. در آن سال‌ها هیچ توجهی به وابستگی موجود بین متون مختلف و ارتباط بین آنها نمی‌شد که این مسئله یکی از عوامل پایین بودن دقت جستجوها بشمار می‌رفت.
در ابتدا بر اساس همین دیدگاه الگوریتم‌ها و روش‌های اندیس گذاری (indexing) پیاده سازی می‌شدند که تنها مشخص می‌کردند یک لغت در یک سند (document) وجود دارد یا خیر. اما با گذشت زمان محققان متوجه ناکارآمدی این دیدگاه در استخراج اطلاعات شدند. به همین دلیل روشی بنام Latent Semantic Indexing که بر پایه Latent Semantic Analysis بنا شده بود به دنیای بازیابی و استخراج اطلاعات معرف شد. کاری که این روش انجام می‌داد این بود که گامی را به مجموعه مراحل موجود در پروسه اندیس گذاری اضافه می‌کرد. این روش بجای آنکه در اندیس گذاری تنها یک متن را در نظر بگیرد و ببیند چه لغاتی در آن آورده شده است، کل مجموعه اسناد را با هم و در کنار یکدیگر در نظر می‌گرفت تا ببیند که چه اسنادی لغات مشابه با لغات موجود در سند مورد بررسی را دارند. به بیان دیگر اسناد مشابه با سند فعلی را به نوعی مشخص می‌نمود.
بر اساس دیدگاه LSI اسناد مشابه با هم، اسنادی هستند که لغات مشابه یا مشترک بیشتری داشته باشند. توجه داشته باشید تنها نمی‌گوییم لغات مشترک بیشتری بلکه از واژه لغات مشابه نیز استفاده می‌کنیم. چرا که بر اساس LSI دو سند ممکن است هیچ لغت مشترکی نداشته باشند (یعنی لغات یکسان نداشته باشند) اما لغاتی در آنها وجود داشته باشد که به لحاظی معنایی و مفهومی هم معنا و یا مرتبط به هم باشند. بعنوان مثال لغات شش و ریه دو لغت متفاوت اما مرتبط با یکدیگر هستند و اگر دو لغات در دوسند آورده شوند می‌توان حدس زد که ارتباط و شباهتی معنایی بین آنها وجود دارد. به روش هایی که بر اساس این دیدگاه ارائه می‌شوند روش‌های جستجوی معنایی نیز گفته می‌شود. این دیدگاه مشابه دیدگاه انسانی در مواجهه با متون نیز است. انسان هنگامی که دو متن را با یکدیگر مقایسه می‌کند تنها بدنبال لغات یکسان در آن‌ها نمی‌گردد بلکه شباهت‌های معنایی بین لغات را نیز در نظر می‌گیرد این اصل و نگرش پایه و اساس الگوریتم LSI و همچنین حوزه ای از علم بازیابی اطلاعات بنام مدل سازی موضوعی (Topic Modeling) می‌باشد.
هنگامیکه شما پرس و جویی را بر روی مجموعه ای از اسناد (که بر اساس LSI اندیس گذاری شده‌اند) اجرا می‌کنید، موتور جستجو ابتدا بدنبال لغاتی می‌گردد که بیشترین شباهت را به کلمات موجود در پرس و جوی شما دارند. بعبارتی پرس و جوی شما را بسط می‌دهد (query expansion)، یعنی علاوه بر لغات موجود در پرس و جو، لغات مشابه آنها را نیز به پرس و جوی شما می‌افزاید. پس از بسط دادن پرس و جو، موتور جستجو مطابق روال معمول در سایر روش‌های جستجو، اسنادی که این لغات (پرس و جوی بسط داده شده) در آنها وجود دارند را بعنوان نتیجه به شما باز می‌گرداند. به این ترتیب ممکن است اسنادی به شما بازگردانده شوند که لغات پرس و جوی شما در آنها وجود نداشته باشد اما LSI بدلیل وجود ارتباطات معنایی، آنها را مشابه و مرتبط با جستجو تشخیص داده باشد. توجه داشته باشید که الگوریتم‌های جستجوی معمولی و ساده، بخشی از اسناد را که مرتبط با پرس و جو هستند، اما شامل لغات مورد نظر شما نمی‌شوند، از دست می‌دهد (یعنی کاهش recall).
برای آنکه با دیدگاه LSI بیشتر آشنا شوید در اینجا مثالی از نحوه عملکرد آن می‌زنیم. فرض کنید می‌خواهیم بر روی مجموعه ای از اسناد در حوزه زیست شناسی اندیس گذاری کنیم. بر مبنای روش LSI چنانچه لغاتی مانند کروموزم، ژن و DNA در اسناد زیادی در کنار یکدیگر آورده شوند (یا بعبارتی اسناد مشترک باهم زیادی داشته باشند)، الگوریتم جستجو چنین برداشت می‌کند که به احتمال زیاد نوعی رابطه معنایی بین آنها وجود دارد. به همین دلیل اگر شما پرس و جویی را با کلمه کلیدی "کروموزوم" اجرا نمایید، الگوریتم علاوه بر مقالاتی که مستقیما واژه کروموزوم در آنها وجود دارد، اسنادی که شامل لغات "DNA" و "ژن" نیز باشند را بعنوان نتیجه به شما باز خواهد گرداند. در واقع می‌توان گفت الگوریتم جستجو به پرس و جوی شما این دو واژه را نیز اضافه می‌کند که

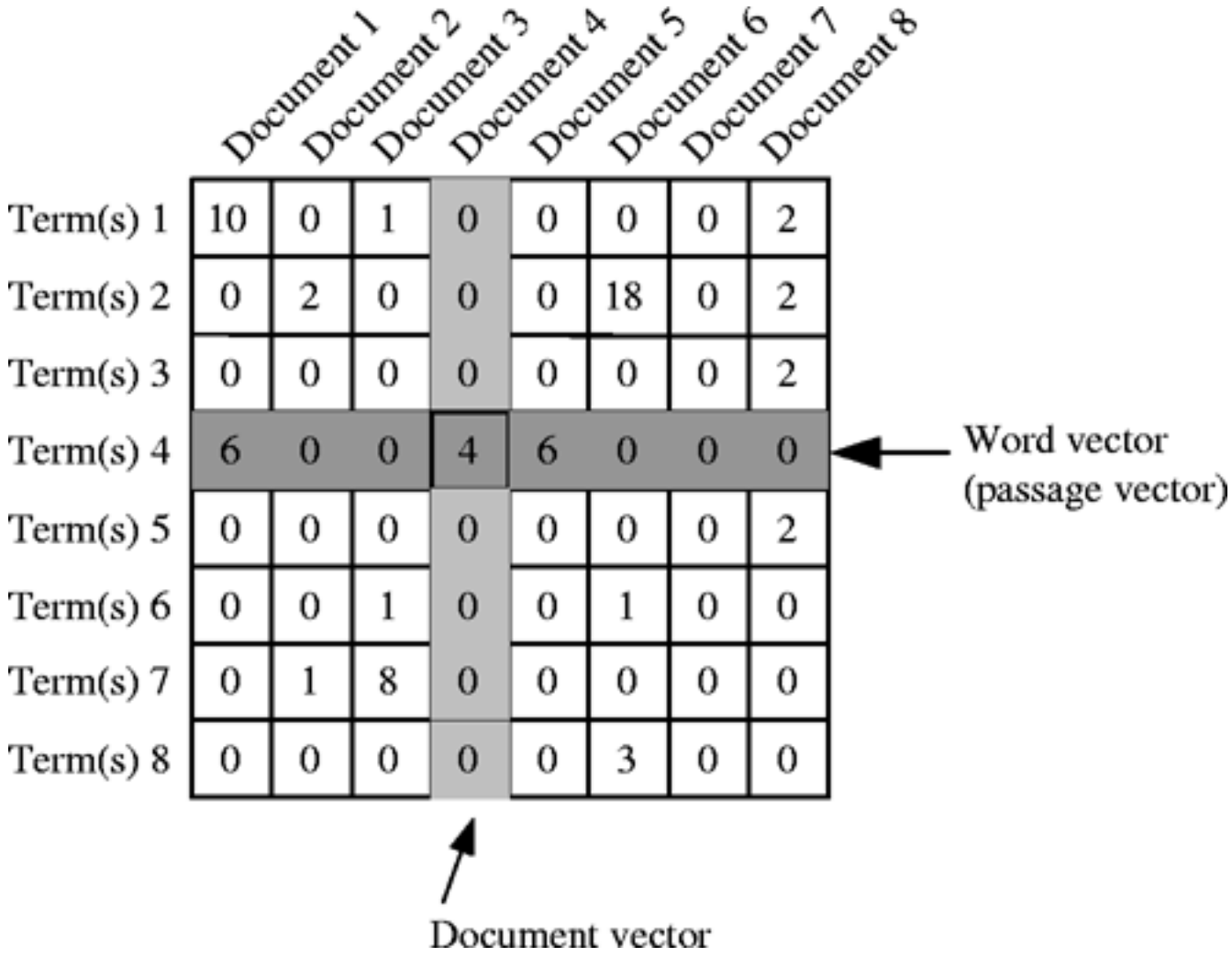
همان بسط دادن پرس و جوی شما است. دقت داشته باشید که الگوریتم جستجو هیچ اطلاع و دانشی از معنای لغات مذکور ندارد و تنها بر اساس تحلیل‌های ریاضی به این نتیجه می‌رسد که در بخش‌های بعدی چگونگی آن را برای شما بازگو خواهیم نمود. یکی از برتری‌های مهم LSI نسبت به روش‌های مبتنی بر کلمات کلیدی (keyword based) این است که در LSI، ما به recall بالاتری دست پیدا می‌کنیم، بدین معنی که از کل جواب‌های موجود برای پرس و جوی شما، جواب‌های بیشتری به کاربر نمایش داده خواهند شد.
یکی از مهمترین نقاط قوت LSI اینست که این روش تنها متکی بر ریاضیات است و هیچ نیازی به دانستن معنای لغات یا پردازش کلمات در متون ندارد. این مسئله باعث می‌شود بتوان این روش را بر روی هر مجموعه متنی و با هر زبانی بکار گرفت. علاوه بر آن می‌توان LSI را بصورت ترکیبی با الگوریتم‌های جستجوی دیگر استفاده نمود و یا تنها متکی بر آن موتور جستجویی را پیاده سازی کرد.

نحوه عملکرد Latent Semantic Indexing

در روش LSI مبنا وقوع همزمان لغات در اسناد می‌باشد. در اصطلاح علمی به این مسئله word co-occurrence گفته می‌شود. به بیان دیگر LSI بدنبال لغاتی می‌گردد که در اسناد بیشتری در با هم آورده می‌شوند. پیش از آنکه وارد مباحث ریاضی و محاسباتی LSI شویم بهتر است کمی بیشتر در مورد این مسوله به لحاظ نظری بحث کنیم. **لغات زائد**
به نحوه صحبت کردن روز مره انسان‌ها دقت کنید. بسیاری از واژگانی که در طول روز و در محاوره‌ها از انها استفاده می‌کنیم، تاثیری در معنای سخن ما ندارند. این مسئله در نحوه نگارش ما نیز صادق است. خیلی از لغات از جمله حروف اضافه، حروف ربط، برخی از افعال پر استفاده و غیره در جملات دیده می‌شوند اما معنای سخن ما در آنها نهفته نمی‌باشد. بعنوان مثال به جمله "جهش در ژن‌ها می‌تواند منجر به بیماری سرطان شود" درقت کنید. در این جمله لغاتی که از اهمیت بالایی بر خوردار هستند و به نوعی بار معنایی جمله بر دوش آنهاست عبارتند از "جهش"، "ژن"، بیماری" و "سرطان". بنابراین می‌توان سایر لغات مانند "در"، "می تواند" و "به" را حذف نمود. به این لغات در اصطلاح علم بازیابی اطلاعات (Information Retrieval) لغات زائد (redundant) گفته می‌شود که در اکثر الگوریتم‌های جستجو یا پردازش زبان طبیعی (natural language processing) برای رسیدن به نتایج قابل قبول باید حذف می‌شوند.روش LSI نیز از این قاعده مستثنی نیست. پیش از اجرای آن بهتر است این لغات زائد حذف گردند. این مسئله علاوه بر آنکه بر روی کیفیت نتایج خروجی تاثیر مثبت دارد، تا حد قابل ملاحظه ای کار پردازش و محاسبات را نیز تسهیل می‌نماید.

مدل کردن لغات و اسناد

پس از آنکه لغات اضافی از مجموعه متون حذف شد باید بدنبال روشی برای مدل کردن داده‌های موجود در مجموعه اسناد بگردیم تا بتوان کاربر پردازش را با توجه به آن مدل انجام داد. روشی که در LSI برای مدلسازی بکار گرفته می‌شود استفاده از ماتریس لغت – سند (term-document matrix) است. این ماتریس یک گرید بسیار بزرگ است که هر سطر از آن نماینده یک سند و هر ستون از ان نماینده یک لغت در مجموعه متنی ما می‌باشد(البته این امکان وجود دارد که جای سطر و ستون‌ها عوض شود). هر سلول از این ماتریس بزرگ نیز به نوعی نشان دهنده ارتباط بین سند و لغت متناظر با آن سلول خواهد بود. بعنوان مثال در ساده‌ترین حات می‌توان گفت که اگر لغتی در سند یافت نشد خانه متناظر با انها در ماتریس لغت – سند خالی خواهد ماند و در غیر این صورت مقدار یک را خواهد گرفت. در برخی از روش‌ها سلول‌ها را با تعداد دفعات تکرار لغات در اسناد متناظر پر می‌کنند و در برخی دیگر از معیارهای پیچیده‌تری مانند tf*idf استفاده می‌نمایند. شکل زیر نمونه از این ماتریس‌ها را نشان می‌دهد :



برای ایجاد چنین ماتریسی باید تک تک اسناد و لغات موجود در مجموعه متنی را پردازش نمود و خانه‌های متناظر را در ماتریس لغت – سند مقدار دهی نمود.خروجی این کار ماتریسی مانند ماتریس شکل بالا خواهد شد (البته در مقیاسی بسیار بزرگتر) که بسیاری از خانه‌های ان صفر خواهند بود (مانند آنچه در شکل نیز مشاهده می‌کنید). به این مسئله تنک بودن (sparseness) ماتریس گفته می‌شود که یکی از مشکلات استفاده از مدل ماتریس لغت – سند محسوب می‌شود.

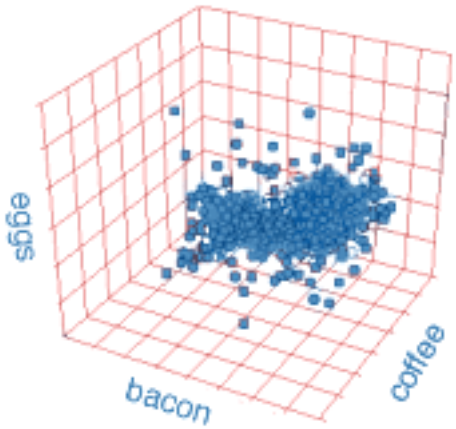
این ماتریس، بازتابی از کل مجموعه متنی را به ما می‌دهد. بعنوان مثال اگر بخواهیم ببینیم در سند i چه لغاتی وجود دارد، تنها کافی است به سراغ سطر iام از ماتریس برویم (البته در صورتی که ماتریس ما سند – لغت باشد) وآن را بیرون بکشیم. به این سطر در اصطلاح بردار سند (document vector) گفته می‌شود. همین کار را در مورد لغات نیز می‌توان انجام داد. بعنوان مثال با رفتن به سراغ ستون j ام می‌توان دریافت که لغت j ام در چه اسنادی آورده شده است. به ستون j ام نیز در ماتریس سند – لغت، بردار لغت (term vector) گفته می‌شود. توجه داشته باشید که این بردارها در مباحث و الگوریتم‌های مربوط به بازیابی اطلاعات و پردازش زبان طبیعی بسیار پر کاربرد می‌باشند.

با داشتن ماتریس لغت – سند می‌توان یک الگوریتم جستجو را پیاده سازی نمود. بسیاری از روش‌های جستجویی که تا کنون پیشنهاد شده اند نیز بر پایه چنین ماتریس هایی بنا شده اند. فرض کنید می‌خواهیم پرس و جویی با کلمات کلیدی "کروموزوم‌های انسان" اجرا کنیم. برای این منظور کافیست ابتدا کلمات کلیدی موجود در پرس و جو را استخراج کرده (در این مثال کروموزوم و انسان دو کلمه کلیدی ما هستند) و سپس به سراغ بردارهای هر یک برویم. همانطور که گفته شد با مراجعه به سطر یا ستون مربوط به لغات می‌توان بردار لغت مورد نظر را یافت. پس از یافتن بردار مربوط به کروموزوم و انسان می‌توان مشخص کرد که این لغات در چه اسناد و متونی اورده شده اند و آنها را استخراج و به کاربر نشان داد. این ساده‌ترین روش جستجو بر مبنای کلمات کلیدی می‌باشد. اما دقت داشته باشید که هدف نهایی در LSI چیزی فراتر از این است. بنابراین نیاز به انجام عملیاتی دیگر بر روی این ماتریس می‌باشد که بتوانیم بر اساس آن ارتباطات معنایی بین لغات و متون را تشخیص دهیم. برای این منظور LSI ماتری لغت – سند را تجزیه (decompose) می‌کند. برای این منظور نیز از تکنیک Singular Value Decomposition استفاده می‌نماید. پیش از

پرداختن به این تکنیک ابتدا بهتر است کمی با فضای برداری چند بعدی (multi-dimensional vector space) آشنا شویم. برای این منظور به مثال زیر توجه کنید. **مثالی از فضای چند بعدی**

فرض کنید قصد دارید تحقیقی در مورد اینکه مردم چه چیز هایی را معمولا برای صبحانه خود سفارش می‌دهند انجام دهید. برای این منظور در یک روز شلوغ به رستورانی در اطراف محل زندگی خود می‌روید و لیست سفارشات صبحانه را می‌گیرید. فرض کنید از بین اقلام متعدد، تمرکز شما تنها بر روی تخم مرغ (egg)، قهوه (coffee) و بیکن (bacon) است. در واقع قصد دارید ببینید چند نفر در سفارش خود این سه قلم را باهم درخواست کرده اند. برای این منظور سفارشات را تک تک بررسی می‌کنید و تعداد دفعات را ثبت می‌کنید.

پس از آنکه کار ثبت و جمع آوری داده‌ها به پایان رسید می‌توانید نتایج را در قالب نموداری نمایش دهید. یک روش برای اینکار رسم نموداری سه بعدی است که هر بعد آن مربوط به یکی از اقلام مذکور می‌باشد. بعنوان مثال در شکل زیر نموداری سه بعدی را که برای این منظور رسم شده است مشاهده می‌کنید. همانطور که در شکل نشان داده شده است محود x مربوط به "bacon"، محور y مربوط به "egg" و محور z نیز مربوط به "coffee" می‌باشد. از آنجایی که این نمودار سه بعدی است برای مشخص کردن نقاط بر روی آن به سه عدد (x ,y ,z) نیاز مندیم. حال اطلاعات جمع اوری شده از صورت سفارشات را یکی یکی بررسی می‌کنیم و بر اساس تعداد دفعات سفارش داده شدن این سه قلم نقطه ای را در این فضای سه بعدی رسم می‌کنیم. بعنوان مثال اگر در سفارشی 2 عدد تخم مرغ و یک قهوه سفارش داده شد بود، این سفارش با (0, 2, 1) در نمودار ما نمایش داده خواهد شد. به این ترتیب می‌توان محل قرار گرفتن این سفارش در فضای سه بعدی سفارشات صبحانه را یافت. این کار را برای تمامی سفارشات انجام می‌دهیم تا سر انجام نموداری مانند نمودار زیر بدست آید.



دقت داشته باشید که اگر از هریک از نقطه آغازین نمودار (0, 0, 1) خطی را به هر یک از نقاط رسم شده بکشید، بردار هایی در فضای "bacon-eggs-coffee"بدست خواهد آمد. هر کدام از این بردارها به ما نشان می‌دهند که در یک صبحانه خاص بیشتر از کدام یک از این سه قلم درخواست شده است. مجموع بردارها در کنار یکدیگر نیز می‌توانند اطلاعات خوبی راجع به گرایش و علاقه مردم به اقلام مذکور در صبحانه‌های خود به ما دهد. به این نمودار نمودار فضای بردار (vector – space) می‌گویند.

حالا وقت آن است که مجددا به بحث مربوط به بازیابی اطلاعات (information retrieval) باز گردیم. همانطور که گفتیم اسناد در یک مجموعه را می‌توان در قالب بردار هایی بنام Term – vector نمایش داد. این بردارها مشابه بردار مثال قبل ما هستند. با این تفاوت که به جای تعداد دفعات تکرار اقلام موجود در صبحانه افراد، تعداد دفعات تکرار لغات را در یک سند در خود دارند. از نظر اندازه نیز بسیار بزرگتر از مثال ما هستند. در یک مجموعه از اسناد ما هزاران هزار لغت داریم که باید بردارهای ما به اندازه تعداد کل لغات منحصر به فرد ما باشند. بعنوان مثال اگر در یک مجموعه ما هزار لغات غیر تکراری داریم بردارهای ما باید هزار بعد داشته باشند. نموداری که اطلاعات را در ان نمایش خواهیم داد نیز بجای سه بعد (در مثال قبل) می‌بایست هزار بعد (یا محور) داشته باشد که البته چنین فضایی قابل نمایش نمی‌باشد.

به مثال صبحانه توجه کنید. همانطور که می‌بینید برخی از نقاط بر روی نمودار نسبت به بقیه به یکدیگر نز دیکتر هستند و ابری از نقاط را در قسمتی از نمودار ایجاد کردند. این نقاط نزدیک به هم باعث می‌شوند که بردارهای آنها نیز با فاصله نزدیک به هم در

فضای برداری مثال ما قرار گیرند. علت نزدیک بودن این بردارها اینست که تعداد دفعات تکرار bacon، eggs و coffee در انها مشابه به هم بوده است. بنابراین می‌توان گفت که این نقاط (یا سفارشات مربوط به انها) به یکدیگر شبیه می‌باشند. در مورد فضای برداری مجموعه از اسناد نیز وضع به همین ترتیب است. اسنادی که لغات مشترک بیشتری با یک دیگر دارند بردارهای مربوط به انها در فضای برداری در کنار یکدیگر قرار خواهند گرفت. هر چه این مشترکات کمتر باشد منجر به فاصله گرفتن بردارها از یکدیگر می‌گردد. بنابراین می‌بینید که با داشتن فضای برداری و مقایسه بردارها با یکدیگر می‌توان نتیجه گرفت که دو سند چقدر به یکدیگر شباهت دارند.

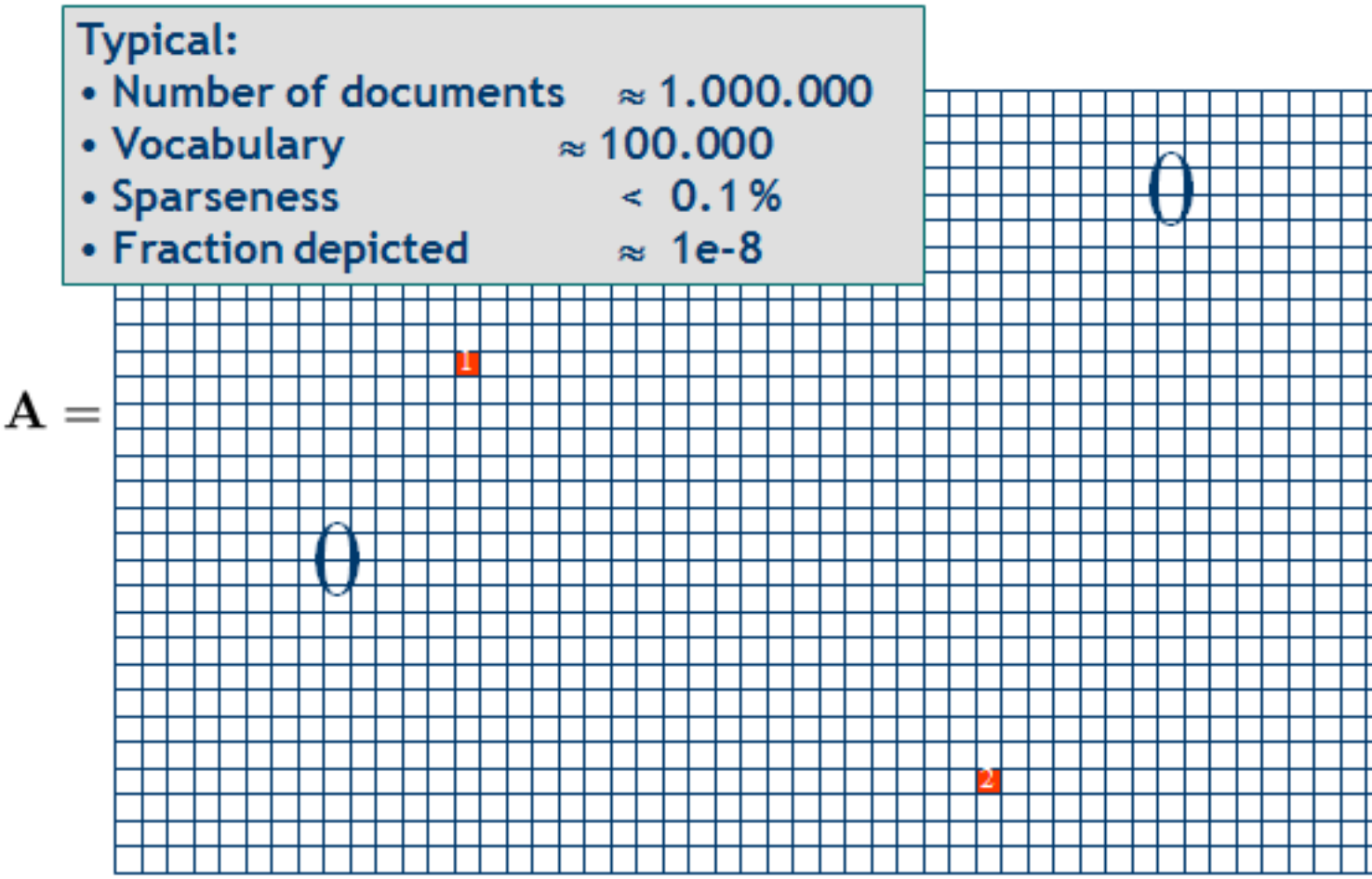
در بسیاری از روش‌های جستجو از چنین بردار هایی برای یافتن اسناد مرتبط به پرس و جوی کاربران استفاده می‌کنند. برای ان منظور تنها کافی اس پرس و جوی کاربر را بصورت برداری در فضای برداری مورد نظر نگاشت دهیم و سپس بردار حاصل را با بردارهای مربوط به اسناد مقایسه کنیم و در نهایت آنهایی که بیشترین شباهت را دارند باز به کاربر بازگردانیم. این روش یکی از ساده‌ترین روش‌های مطرح شده در بازیابی اطلاعات است.

خوب حالا بیایید به Latent Semantic Indexing باز گردیم. روش LSI برمبنای همین فضای برداری عمل می‌کند با این تفاوت که فضای برداری را که دارای هزاران هزار بعد می‌باشد به فضای کوچکتری با ابعاد کمتر (مثلا 300 بعد) تبدیل می‌کند. به این کار در اصطلاح عملی کاهش ابعاد (dimensionality reduction) گفته می‌شود. دقت داشته باشید که هنگامیکه این عمل انجام می‌گیرد لغاتی که شباهت و یا ارتباط زیادی به لحاظ معنایی با یکدیگر دارند بجای اینکه هریک در قالب یک بعد نمایش داده شوند، همگی بصورت یک بعد در می‌آیند. بعنوان مثال لغات کروموزم و ژن از نظر معنایی با یکدیگر در ارتباط هستند. در فضای برداری اصلی این دو لغت در قالب دو بعد مجزا نمایش داده می‌شوند اما با اعمال کاهش ابعاد به ازای هر دوی آنها تنها یک بعد خواهیم داشت.

مزیت این کار اینست که اسنادی که لغات مشترکی ندارند اما به لحاظ معنایی با یکدیگر ارتباط دارند در فاضی برداری کاهش یافته نزدیکی بیشتری به یکدیگر خواهند داشت.

روش‌های مختلفی برای اعمال کاهش ابعاد وجود دارد. در LSI از روش Singular Value Decompistion استفاده می‌شود که در بحث بعدی در مورد آن صحبت خواهیم نمود. **Singular Value Decomposition**

پیشتر گفتیم که در LSI برای مدل کردن مجموعه اسناد موجود از ماتریس بزرگی بنام ماتریس لغت – سند استفاده می‌شود. این ماتریس در واقع نمایشی از مدل فضای برداری است که در بخش قبلی به آن اشاره شد. دقت داشته باشید که ما در دنیای واقعی در یک سیستم بزرگ تقریبا چیزی در حدود یک ملیون سند داریم که در مجموع این اسناد تقریبا صد هزار لغت غیر تکراری و منحصر به فرد یافت می‌شود. بنابراین می‌توان گفت میزان تنک بودن ماتریس ما تقریبا برابر با 0.1 درصد خواهد بود. یعنی از کل ماتریس تنها 0.1 درصد آن دارای اطلاعات است و اکثر سلول‌های ماتریس ما خالی می‌باشد. این مسئله را در شکل زیر می‌توانید مشاهده کنید.



در Latent Semantic Indexing با استفاده از روش Singular Value Decomposition این ماتریس را کوچک می‌کنند. به بیان بهتر تقریبی از ماتریس اصلی را ایجاد می‌کنند که ابعاد کوچکتری خواهد داشت. این کار مزایایی را بدنبال دارد. اول آنکه سطرها و ستون هایی (لغات و اسناد) که اهمیت کمی در مجموعه اسناد ما دارند را حذف می‌کند. علاوه بر آن این کار باعث می‌شود که ارتباطات معنایی بین لغات هم معنی یا مرتبط کشف شود. یافتن این ارتباطات معنایی بسیار در پاسخ به پرس و جوها مفید خواهد بود. چرا که مردم معمولا در پرس و جوهای خود از دایره لغات متفاوتی استفاده می‌کنند. بعنوان مثال برای جستجو در مورد مطالب مربوط به ژن‌های انسان برخی از واژه کروموزوم و برخی دیگر از واژه ژنوم و دیگران ممکن است از واژگان دیگری استفاده نمایند. این مسئله مشکلی را در جستجو بنام عدم تطبیق کلمات کلیدی (mismatch problem) بوجود می‌اورده که با اعمال SVD بر روی ماتریس سند – لغت این مشکل برطرف خواهد شد.

توجه داشته باشید که SVD ابعاد بردارهای لغات و سند را کاهش می‌دهد. بعنوان مثال بجای آنکه یک سند در قالب صد هزار بعد (که هر بعد مربوط به یک لغت می‌باشد) نمایش داده شود، بصورت یک بردار مثلا 150 بعدی نمایش داده خواهد شد. طبیعی است که این کاهش ابعاد منجر به از بین رفتن برخی از اطلاعات خواهد شد چرا که ما بسیاری از ابعاد را با یکدیگر ادغام کرده ایم. این مسئله شاید در ابتدا مسئله ای نا مطلوب به نظر آید اما در اینجا نکته ای در آن نهفته است. دقت داشته باشید که آنچه از دست می‌رود اطلاعات زائد (noise) می‌باشد. از بین رفتن این اطلاعات زائد منجر می‌شود تا ارتباطات پنهان موجود در مجموعه اسناد ما نمایان گردند. با اجرای SVD بر روی ماتریس، اسناد و لغات مشابه، مشابه باقی می‌مانند و انهایی که غیر مشابه هستند نیز غیر مشابه باقی خواهد ماند. پس ما از نظر ارتباطات بین اسناد و لغات چیزی را از دست نخواهیم داد.

در مباحث بعدی در مورد چگونگی اعمال SVD و همچنین نحوه پاسخگویی به پرس و جوها مطالب بیشتری را برای شما عزیزان خواهیم نوشت.

موفق و پیروز باشید.

نظرات خوانندگان

نویسنده: محمد رضا
تاریخ: ۱۳۹۳/۰۳/۱۰ ۱۰:۲۴

تشکر می‌کنم از مطلب مفیدتون
در این بازه منابعی دارید معرفی کنید ؟ بی صبرانه منظر بخش بعدی هستیم.
ممنون

نویسنده: حامد خسروجردی
تاریخ: ۱۳۹۳/۰۳/۱۴ ۲۱:۱۰

سلام دوست عزیز. از اونجایی که این روش سالهای زیادی است معرفی شده و مورد استفاده قرار گرفته (از اواخر دهه 90 میلادی) مقالات و منابع زیادی تو این حوزه منتشر شده تا بحال و بر روی اینترنت هم موجود است. ولی برای شروع می‌تونید سری به این لینک‌ها بزنید :

لینک زیر بطور آکادمیک توضیحاتی را در مورد Latent Semantic Analysis ارائه میده:
An introduction To Latent Semantic Analysis

این لینک مربوط به دانشگاه استندفورد هستش و واقعا یه مرجع عالی در مورد روش‌های مختلف بازیابی اطلاعات (Information Retrieval) هستش که اگر علاقه به سایر حوزه‌ها تو این زمینه دارید می‌تونید بعنوان یه مرجع خوب ازش استفاده کنید : Latent semantic indexing

اگر هم شرحی عامیانه‌تر از این مقوله می‌خواهید می‌تونید به این لینک سری بزنید : LATENT SEMANTIC INDEXING

نویسنده: محسن
تاریخ: ۱۳۹۳/۰۳/۲۱ ۱۲:۲۳

سلام
ممنون از مقاله جالبتون
آیا برنامه پیاده سازی شده ای هم وجود داره؟
نسخه ایرانی یا خارجی؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۳/۲۱ ۱۲:۵۶

Semantic Search جزو تازه‌های SQL Server 2012 است (البته این مورد خاص، زبان‌های محدودی را پشتیبانی می‌کند).

عنوان: **استفاده از Full text search توسط Entity Framework**
نویسنده: محسن موسوی
تاریخ: ۱۳۹۳/۰۴/۲۸ ۲۳:۱۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Full Text Search

پیشنیاز مطلب:
پشتیبانی از Full Text Search در SQL Server

Full Text Search یا به اختصار FTS یکی از قابلیت‌های SQL Server جهت جستجوی پیشرفته در متون میباشد. این قابلیت تا کنون در EF 6.1.1 ایجاد نشده است.
در ادامه پیاده سازی از FTS در EF را مشاهده مینمایید.
جهت ایجاد قابلیت FTS از متد Contains در Linq استفاده شده است.
ابتدا متدهای الحاقی جهت اعمال دستورات FREETEXT و CONTAINS اضافه میشود.سپس دستورات تولیدی EF را قبل از اجرا بر روی بانک اطلاعاتی توسط امکان Command Interception به دستورات FTS تغییر میدهیم.
همانطور که میدانید دستور Contains در Linq توسط EF به دستور LIKE تبدیل میشود. به جهت اینکه ممکن است بخواهیم از دستور LIKE نیز استفاده کنیم یک پیشوند به مقادیری که میخواهیم به دستورات FTS تبدیل شوند اضافه مینماییم.
جهت استفاده از Fts در EF کلاس‌های زیر را ایجاد نمایید.

کلاس FullTextPrefixes :

```
/// <summary>
    ///
    /// </summary>
    public static class FullTextPrefixes
    {
        /// <summary>
        ///
        /// </summary>
        public const string ContainsPrefix = "-CONTAINS-";

        /// <summary>
        ///
        /// </summary>
        public const string FreetextPrefix = "-FREETEXT-";

        /// <summary>
        ///
        /// </summary>
        /// <param name="searchTerm"></param>
        /// <returns></returns>
        public static string Contains(string searchTerm)
        {
            return string.Format("({0}{1})", ContainsPrefix, searchTerm);
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="searchTerm"></param>
        /// <returns></returns>
        public static string Freetext(string searchTerm)
        {
            return string.Format("({0}{1})", FreetextPrefix, searchTerm);
        }

    }
```

کلاس جاری جهت علامت گذاری دستورات FTS میباشد و توسط متدهای الحاقی و کلاس FtsInterceptor استفاده میگردد.

کلاس FullTextSearchExtensions :

```
public static class FullTextSearchExtensions
    {
```

```
        /// <summary>
        ///
        /// </summary>
        /// <typeparam name="TEntity"></typeparam>
        /// <param name="source"></param>
        /// <param name="expression"></param>
        /// <param name="searchTerm"></param>
        /// <returns></returns>
        public static IQueryable<TEntity> FreeTextSearch<TEntity>(this IQueryable<TEntity> source,
Expression<Func<TEntity, object>> expression, string searchTerm) where TEntity : class
        {
            return FreeTextSearchImp(source, expression, FullTextPrefixes.Freetext(searchTerm));
        }

        /// <summary>
        ///
        /// </summary>
        /// <typeparam name="TEntity"></typeparam>
        /// <param name="source"></param>
        /// <param name="expression"></param>
        /// <param name="searchTerm"></param>
        /// <returns></returns>
        public static IQueryable<TEntity> ContainsSearch<TEntity>(this IQueryable<TEntity> source,
Expression<Func<TEntity, object>> expression, string searchTerm) where TEntity : class
        {
            return FreeTextSearchImp(source, expression, FullTextPrefixes.Contains(searchTerm));
        }

        /// <summary>
        ///
        /// </summary>
        /// <typeparam name="TEntity"></typeparam>
        /// <param name="source"></param>
        /// <param name="expression"></param>
        /// <param name="searchTerm"></param>
        /// <returns></returns>
        private static IQueryable<TEntity> FreeTextSearchImp<TEntity>(this IQueryable<TEntity> source,
Expression<Func<TEntity, object>> expression, string searchTerm)
        {
            if (String.IsNullOrEmpty(searchTerm))
            {
                return source;
            }

            // The below represents the following lamda:
            // source.Where(x => x.[property].Contains(searchTerm))

            //Create expression to represent x.[property].Contains(searchTerm)
            //var searchTermExpression = Expression.Constant(searchTerm);
            var searchTermExpression = Expression.Property(Expression.Constant(new { Value = searchTerm
}), "Value");
            var checkContainsExpression = Expression.Call(expression.Body,
typeof(string).GetMethod("Contains"), searchTermExpression);

            //Join not null and contains expressions

            var methodCallExpression = Expression.Call(typeof(Queryable),
                                                       "Where",
                                                       new[] { source.ElementType },
                                                       source.Expression,
                                                       Expression.Lambda<Func<TEntity,
bool>>(checkContainsExpression, expression.Parameters));

            return source.Provider.CreateQuery<TEntity>(methodCallExpression);
        }
```

در این کلاس متدهای الحاقی جهت اعمال قابلیت Fts ایجاد شده است.

متد FreeTextSearch جهت استفاده از دستور FREETEXT استفاده میشود.در این متد پیشوند -FREETEXT- جهت علامت گذاری این دستور به ابتدای مقدار جستجو اضافه میشود. این متد دارای دو پارامتر میباشد ، اولی ستونی که میخواهیم بر روی آن جستجوی FTS انجام دهیم و دومی عبارت جستجو.

متد ContainsSearch جهت استفاده از دستور CONTAINS استفاده میشود.در این متد پیشوند -CONTAINS- جهت علامت گذاری این دستور به ابتدای مقدار جستجو اضافه میشود. این متد دارای دو پارامتر میباشد ، اولی ستونی که میخواهیم بر روی آن جستجوی FTS انجام دهیم و دومی عبارت جستجو.

متد FreeTextSearchImp جهت ایجاد دستور Contains در Linq میباشد.

کلاس FtsInterceptor :

```
public class FtsInterceptor : IDbCommandInterceptor
    {

        /// <summary>
        ///
        /// </summary>
        /// <param name="command"></param>
        /// <param name="interceptionContext"></param>
        public void NonQueryExecuting(DbCommand command, DbCommandInterceptionContext<int>
interceptionContext)
        {
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="command"></param>
        /// <param name="interceptionContext"></param>
        public void NonQueryExecuted(DbCommand command, DbCommandInterceptionContext<int>
interceptionContext)
        {
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="command"></param>
        /// <param name="interceptionContext"></param>
        public void ReaderExecuting(DbCommand command, DbCommandInterceptionContext<DbDataReader>
interceptionContext)
        {
            RewriteFullTextQuery(command);
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="command"></param>
        /// <param name="interceptionContext"></param>
        public void ReaderExecuted(DbCommand command, DbCommandInterceptionContext<DbDataReader>
interceptionContext)
        {
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="command"></param>
        /// <param name="interceptionContext"></param>
        public void ScalarExecuting(DbCommand command, DbCommandInterceptionContext<object>
interceptionContext)
        {
            RewriteFullTextQuery(command);
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="command"></param>
        /// <param name="interceptionContext"></param>
        public void ScalarExecuted(DbCommand command, DbCommandInterceptionContext<object>
interceptionContext)
        {
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="cmd"></param>
        public static void RewriteFullTextQuery(DbCommand cmd)
        {
            var text = cmd.CommandText;
            for (var i = 0; i < cmd.Parameters.Count; i++)
            {
                var parameter = cmd.Parameters[i];
```

```
                if (
                    !parameter.DbType.In(DbType.String, DbType.AnsiString, DbType.StringFixedLength,
                        DbType.AnsiStringFixedLength)) continue;
                if (parameter.Value == DBNull.Value)
                    continue;
                var value = (string)parameter.Value;
                if (value.IndexOf(FullTextPrefixes.ContainsPrefix, StringComparison.Ordinal) >= 0)
                {
                    parameter.Size = 4096;
                    parameter.DbType = DbType.AnsiStringFixedLength;
                    value = value.Replace(FullTextPrefixes.ContainsPrefix, ""); // remove prefix we
added n linq query
                    value = value.Substring(1, value.Length - 2); // remove %% escaping by linq
translator from string.Contains to sql LIKE
                    parameter.Value = value;
                    cmd.CommandText = Regex.Replace(text,
                        string.Format(
                            @"\[(\w*)\].\[(\w*)\]\s*LIKE\s*@{0}\s?(?:ESCAPE N?'~')",
parameter.ParameterName),
                        string.Format(@"CONTAINS([$1].[$2], @{0})", parameter.ParameterName));
                    if (text == cmd.CommandText)
                        throw new Exception("FTS was not replaced on: " + text);
                    text = cmd.CommandText;
                }
                else if (value.IndexOf(FullTextPrefixes.FreetextPrefix, StringComparison.Ordinal) >= 0)
                {
                    parameter.Size = 4096;
                    parameter.DbType = DbType.AnsiStringFixedLength;
                    value = value.Replace(FullTextPrefixes.FreetextPrefix, ""); // remove prefix we
added n linq query
                    value = value.Substring(1, value.Length - 2); // remove %% escaping by linq
translator from string.Contains to sql LIKE
                    parameter.Value = value;
                    cmd.CommandText = Regex.Replace(text,
                        string.Format(
                            @"\[(\w*)\].\[(\w*)\]\s*LIKE\s*@{0}\s?(?:ESCAPE N?'~')",
parameter.ParameterName),
                        string.Format(@"FREETEXT([$1].[$2], @{0})", parameter.ParameterName));
                    if (text == cmd.CommandText)
                        throw new Exception("FTS was not replaced on: " + text);
                    text = cmd.CommandText;
                }
            }
        }

    }
```

در این کلاس دستوراتی را که توسط متدهای الحاقی جهت امکان Fts علامت گذاری شده بودند را یافته و دستور LIKE را با دستورات CONTAINSو FREETEXT جایگزین میکنیم.

در ادامه برنامه ای جهت استفاده از این امکان به همراه دستورات تولیدی آنرا مشاهده مینمایید.

```
public class Note
    {
        /// <summary>
        ///
        /// </summary>
        public int Id { get; set; }

        /// <summary>
        ///
        /// </summary>
        public string NoteText { get; set; }
    }

  public class FtsSampleContext : DbContext
    {
        /// <summary>
        ///
        /// </summary>
        public DbSet<Note> Notes { get; set; }

        /// <summary>
        ///
        /// </summary>
        /// <param name="modelBuilder"></param>
```

```
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Configurations.Add(new NoteMap());
        }
    }

/// <summary>
    ///
    /// </summary>
    class Program
    {
        /// <summary>
        ///
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            DbInterception.Add(new FtsInterceptor());
            const string searchTerm = "john";
            using (var db = new FtsSampleContext())
            {
                var result1 = db.Notes.FreeTextSearch(a => a.NoteText, searchTerm).ToList();
                //SQL Server Profiler result ===>>>
                //exec sp_executesql N'SELECT
                //    [Extent1].[Id] AS [Id],
                //    [Extent1].[NoteText] AS [NoteText]
                //    FROM [dbo].[Notes] AS [Extent1]
                //    WHERE FREETEXT([Extent1].[NoteText], @p__linq__0)',N'@p__linq__0
                //char(4096)',@p__linq__0='(john)'
                var result2 = db.Notes.ContainsSearch(a => a.NoteText, searchTerm).ToList();
                //SQL Server Profiler result ===>>>
                //exec sp_executesql N'SELECT
                //    [Extent1].[Id] AS [Id],
                //    [Extent1].[NoteText] AS [NoteText]
                //    FROM [dbo].[Notes] AS [Extent1]
                //    WHERE CONTAINS([Extent1].[NoteText], @p__linq__0)',N'@p__linq__0
                //char(4096)',@p__linq__0='(john)'
            }
            Console.ReadKey();
        }
    }
```

ابتدا کلاس FtsInterceptor را به EF معرفی مینماییم. سپس از دو متد الحاقی مذکور استفاده مینماییم. خروجی هر دو متد توسط SQL Server Profiler در زیر هر متد مشاهده مینمایید.

تمامی کدها و مثال مربوطه در آدرس https://effts.codeplex.com قرار گرفته است.

منبع:

Full Text Search in Entity Framework 6

نظرات خوانندگان

نویسنده: محسن موسوی
تاریخ: ۱۳۹۳/۰۵/۰۹ ۱۶:۵۶

بسته نوگت پروژه جاری:

```
Install-Package EfFts
```