عنوان: **ایندکس منحصر به فرد با استفاده از Data Annotation در EF Code First**
نویسنده: پدرام جباری
تاریخ: ۱۳۹۱/۰۹/۲۲ ۲۳:۲۸
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Data Annotation, Unique, Index

در حال حاضر امکان خاصی برای ایجاد ایندکس منحصر به فرد در EF First Code وجود ندارد, برای این کار راه‌های زیادی وجود دارد مانند پست قبلی آقای نصیری, در این آموزش از Data Annotation و یا همان Attribute هایی که بالای Propertyهای مدل‌ها قرار می‌دهیم, مانند کد زیر :

```
public class User
    {
        public int Id { get; set; }

        [Unique]
        public string Email { get; set; }

        [Unique("MyUniqueIndex",UniqueIndexOrder.ASC)]
        public string Username { get; set; }

        [Unique(UniqueIndexOrder.DESC)]
        public string PersonalCode{ get; set; }

        public string Password { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }
```

همانطور که در کد بالا می‌بینید با استفاده از Attribute Unique ایندکس منحصر به فرد آن در دیتابیس ساخته خواهد شد.
ابتدا یک کلاس برای Attribute Unique به صورت زیر ایجاد کنید :

```
using System;

namespace SampleUniqueIndex
{
    [AttributeUsage(AttributeTargets.Property, Inherited = false, AllowMultiple = false)]
    public class UniqueAttribute : Attribute
    {
        public UniqueAttribute(UniqueIndexOrder order = UniqueIndexOrder.ASC) {
            Order = order;
        }
        public UniqueAttribute(string indexName,UniqueIndexOrder order = UniqueIndexOrder.ASC)
        {
            IndexName = indexName;
            Order = order;
        }
        public string IndexName { get; private set; }
        public UniqueIndexOrder Order { get; set; }
    }

    public enum UniqueIndexOrder
    {
        ASC,
        DESC
    }
}
```

در کد بالا یک Enum برای مرتب سازی ایندکس به دو صورت صعودی و نزولی قرار دارد, همانند کد ابتدای آموزش که مشاهده می‌کنید امکان تعریف این Attribute به سه صورت امکان دارد که به صورت زیر می‌باشد:
1. ایجاد Attribute بدون هیچ پارامتری که در این صورت نام ایندکس با استفاده از نام جدول و آن فیلد ساخته خواهد شد : IX_Unique_TableName_FieldName و مرتب ساری آن به صورت صعودی می‌باشد.
2.نامی برای ایندکس انتخاب کنید تا با آن نام در دیتابیس ذخیره شود, در این حالت مرتب سازی آن هم به صورت صعودی می‌باشد.
3. در حالت سوم شما ضمن وارد کردن نام ایندکس مرتب سازی آن را نیز وارد می‌کنید.

بعد از کلاس Attribute حالا نوبت به کلاس اصلی میرسد که به صورت زیر می‌باشد:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations.Schema;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Data.Metadata.Edm;
using System.Linq;
using System.Reflection;

namespace SampleUniqueIndex
{
    public static class DbContextExtention
    {
        private static BindingFlags PublicInstance = BindingFlags.Public | BindingFlags.Instance |
BindingFlags.FlattenHierarchy;

        public static void ExecuteUniqueIndexes(this DbContext context)
        {
            var tables = GetTables(context);
            var query = "";
            foreach (var dbSet in GetDbSets(context))
            {
                var entityType = dbSet.PropertyType.GetGenericArguments().First();
                var table = tables[entityType.Name];
                var currentIndexes = GetCurrentUniqueIndexes(context, table.TableName);
                foreach (var uniqueProp in GetUniqueProperties(context, entityType, table))
                {
                    var indexName = string.IsNullOrWhiteSpace(uniqueProp.IndexName) ?
                        "IX_Unique_" + uniqueProp.TableName + "_" + uniqueProp.FieldName :
                        uniqueProp.IndexName;

                    if (!currentIndexes.Contains(indexName))
                    {
                        query += "ALTER TABLE [" + table.TableSchema + "].[" + table.TableName + "] ADD
CONSTRAINT [" + indexName + "] UNIQUE ([" + uniqueProp.FieldName + "] " + uniqueProp.Order + "); ";
                    }
                    else
                    {
                        currentIndexes.Remove(indexName);
                    }
                }
                foreach (var index in currentIndexes)
                {
                    query += "ALTER TABLE [" + table.TableSchema + "].[" + table.TableName + "] DROP
CONSTRAINT " + index + "; ";
                }
            }

            if (query.Length > 0)
                context.Database.ExecuteSqlCommand(query);
        }

        private static List<string> GetCurrentUniqueIndexes(DbContext context, string tableName)
        {
            var sql = "SELECT CONSTRAINT_NAME FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS where
table_name = '"
                      + tableName + "' and CONSTRAINT_TYPE = 'UNIQUE'";
            var result = context.Database.SqlQuery<string>(sql).ToList();
            return result;
        }
        private static IEnumerable<PropertyDescriptor> GetDbSets(DbContext context)
        {
            foreach (PropertyDescriptor prop in TypeDescriptor.GetProperties(context))
            {
                var notMapped = prop.GetType().GetCustomAttributes(typeof(NotMappedAttribute),true);
                if (prop.PropertyType.Name == typeof(DbSet<>).Name && notMapped.Length == 0)
                    yield return prop;
            }
        }
        private static List<UniqueProperty> GetUniqueProperties(DbContext context, Type entity,
TableInfo tableInfo)
        {
            var indexedProperties = new List<UniqueProperty>();
            var properties = entity.GetProperties(PublicInstance);
            var tableName = tableInfo.TableName;
            foreach (var prop in properties)
            {
```

```
                if (!prop.PropertyType.IsValueType && prop.PropertyType != typeof(string)) continue;

                UniqueAttribute[] uniqueAttributes =
(UniqueAttribute[])prop.GetCustomAttributes(typeof(UniqueAttribute), true);
                NotMappedAttribute[] notMappedAttributes =
(NotMappedAttribute[])prop.GetCustomAttributes(typeof(NotMappedAttribute), true);
                if (uniqueAttributes.Length > 0 && notMappedAttributes.Length == 0)
                {
                    var fieldName = GetFieldName(context, entity, prop.Name);
                    if (fieldName != null)
                    {
                        indexedProperties.Add(new UniqueProperty
                        {
                            TableName = tableName,
                            IndexName = uniqueAttributes[0].IndexName,
                            FieldName = fieldName,
                            Order = uniqueAttributes[0].Order.ToString()
                        });
                    }
                }
            }
            return indexedProperties;
        }
        private static Dictionary<string, TableInfo> GetTables(DbContext context)
        {
            var tablesInfo = new Dictionary<string, TableInfo>();
            var metadata = ((IObjectContextAdapter)context).ObjectContext.MetadataWorkspace;
            var tables = metadata.GetItemCollection(DataSpace.SSpace)
              .GetItems<EntityContainer>()
              .Single()
              .BaseEntitySets
              .OfType<EntitySet>()
              .Where(s => !s.MetadataProperties.Contains("Type")
                || s.MetadataProperties["Type"].ToString() == "Tables");
            foreach (var table in tables)
            {
                var tableName = table.MetadataProperties.Contains("Table")
                    && table.MetadataProperties["Table"].Value != null
                  ? table.MetadataProperties["Table"].Value.ToString()
                  : table.Name;
                var tableSchema = table.MetadataProperties["Schema"].Value.ToString();
                tablesInfo.Add(table.Name, new TableInfo
                {
                    EntityName = table.Name,
                    TableName = tableName,
                    TableSchema = tableSchema,
                });
            }

            return tablesInfo;
        }
        public static string GetFieldName(DbContext context, Type entityModel, string propertyName)
        {
            var metadata = ((IObjectContextAdapter)context).ObjectContext.MetadataWorkspace;
            var osMembers = metadata.GetItem<EntityType>(entityModel.FullName,
DataSpace.OSpace).Properties;
            var ssMebers = metadata.GetItem<EntityType>("CodeFirstDatabaseSchema." + entityModel.Name,
DataSpace.SSpace).Properties;

            if (!osMembers.Contains(propertyName)) return null;

            var index = osMembers.IndexOf(osMembers[propertyName]);
            return ssMebers[index].Name;
        }

        internal class UniqueProperty
        {
            public string TableName { get; set; }
            public string FieldName { get; set; }
            public string IndexName { get; set; }
            public string Order { get; set; }
        }
        internal class TableInfo
        {
            public string EntityName { get; set; }
            public string TableName { get; set; }
            public string TableSchema { get; set; }
        }
    }
}
```

در کد بالا با استفاده از Extension Method برای کلاس DbContext یک متد با نام ExecuteUniqueIndexes ایجاد می‌کنیم تا برای ایجاد ایندکس‌ها در دیتابیس از آن استفاده کنیم.
روند اجرای کلاس بالا به صورت زیر می‌باشد:
در ابتدای متد ExecuteUniqueIndexes() :

```
public static void ExecuteUniqueIndexes(this DbContext context)
        {
            var tables = GetTables(context);
            ...
        }
```

با استفاده از متد GetTables() ما تمام جداول ساخته توسط دیتایس توسط DbContext را گرفته:

```
private static Dictionary<string, TableInfo> GetTables(DbContext context)
        {
            var tablesInfo = new Dictionary<string, TableInfo>();
            var metadata = ((IObjectContextAdapter)context).ObjectContext.MetadataWorkspace;
            var tables = metadata.GetItemCollection(DataSpace.SSpace)
              .GetItems<EntityContainer>()
              .Single()
              .BaseEntitySets
              .OfType<EntitySet>()
              .Where(s => !s.MetadataProperties.Contains("Type")
                || s.MetadataProperties["Type"].ToString() == "Tables");
            foreach (var table in tables)
            {
                var tableName = table.MetadataProperties.Contains("Table")
                    && table.MetadataProperties["Table"].Value != null
                  ? table.MetadataProperties["Table"].Value.ToString()
                  : table.Name;
                var tableSchema = table.MetadataProperties["Schema"].Value.ToString();
                tablesInfo.Add(table.Name, new TableInfo
                {
                    EntityName = table.Name,
                    TableName = tableName,
                    TableSchema = tableSchema,
                });
            }

            return tablesInfo;
        }
```

با استفاده از این طریق چنانچه کاربر نام دیگری برای هر جدول در نظر بگیرد مشکلی ایجاد نمی‌شود و همینطور Schema جدول نیز گرفته می‌شود, سه مشخصه نام مدل و نام جدول و Schema جدول در کلاس TableInfo قرار داده می‌شود و در انتها تمام جداول در یک Collection قرار داده میشوند و به عنوان خروجی متد استفاده می‌شوند.
بعد از آنکه نام جداول متناظر با نام مدل آنها را در اختیار داریم نوبت به گرفتن تمام DbSetها در DbContext می‌باشد که با استفاده از متد GetDbSets() :

```
public static void ExecuteUniqueIndexes(this DbContext context)
        {
            var tables = GetTables(context);
            var query = "";
            foreach (var dbSet in GetDbSets(context))
            {
            ....
        }
```

در این متد چنانچه Property دارای Attribute NotMapped باشد در لیست خروجی متد قرار داده **نمی‌شود**.
سپس داخل چرخه DbSetها نوبت به گرفتن ایندکس‌های موجود با استفاده از متد GetCurrentUniqueIndexes() برای این مدل می‌باشد تا از ایجاد دوباره آن جلوگیری شود و البته اگر ایندکس هایی را در مدل تعریف نکرده باشید از دیتابیس حذف شوند.

```
public static void ExecuteUniqueIndexes(this DbContext context)
        {
```

```
            ...
            foreach (var dbSet in GetDbSets(context))
            {
                var entityType = dbSet.PropertyType.GetGenericArguments().First();
                var table = tables[entityType.Name];
                var currentIndexes = GetCurrentUniqueIndexes(context, table.TableName);
            }
        }
```

بعد از آن نوبت به گرفتن Propertyهای دارای Attribute Unique می‌باشد که این کار نیز با استفاده از متد GetUniqueProperties() انجام خواهد شد.

در متد GetUniqueProperties() چند شرط بررسی خواهد شد از جمله اینکه Property از نوع Value Type باشد و نه یک کلاس سپس Attribute NotMapped را نداشته باشد و بعد از آن می‌بایست نام متناظر با آن Property را در دیتابیس به دست بیاریم برای این کار از متد GetFieldName() استفاده می‌کنیم:

```
public static string GetFieldName(DbContext context, Type entityModel, string propertyName)
        {
            var metadata = ((IObjectContextAdapter)context).ObjectContext.MetadataWorkspace;
            var osMembers = metadata.GetItem<EntityType>(entityModel.FullName,
DataSpace.OSpace).Properties;
            var ssMebers = metadata.GetItem<EntityType>("CodeFirstDatabaseSchema." + entityModel.Name,
DataSpace.SSpace).Properties;

            if (!osMembers.Contains(propertyName)) return null;

            var index = osMembers.IndexOf(osMembers[propertyName]);
            return ssMebers[index].Name;
        }
```

برای این کار با استفاده از MetadataWorkspace در DbContext دو لیست SSpace و OSpace استفاده می‌کنیم که در ادامه در مورد این گونه لیست ها بیشتر توضیح می‌دهیم , سپس با استفاده از Memberهای این دو لیست و ایندکس‌های متناظر در این دو لیست نام متناظر با Property را در دیتابیس پیدا خواهیم کرد, **البته یک نکته مهم هست چنانچه برای فیلدهای دیتابیس OrderColumn قرار داده باشید دو لیست Memberها از نظر ایندکس متناظر متفاوت خواهند شد پس در نتیجه ایندکس به اشتباه برروی یک فیلد دیگر اعمال خواهد شد.**

لیست‌ها در MetadataWorkspace:

1. CSpace : این لیست شامل آبجکت‌های Conceptual از مدل‌های شما می‌باشد تا برای Mapping دیتابیس با مدل‌های شما مانند مبدلی این بین عمل کند.
2. OSpace : این لیست شامل آبجکت‌های مدل‌های شما می‌باشد.
3. SSpace : این لیست نیز شامل آبجکت‌های مربوط به دیتابیس از مدل‌های شما می‌باشد
4. CSSpace : این لیست شامل تنظیمات Mapping بین دو لیست SSpace و CSpace می‌باشد.
5. OCSpace : این لیست شامل تنظیمات Mapping بین دو لیست OSpace و CSpace می‌باشد.

روند Mapping مدل‌های شما از OSpace شروع شده و به SSpace ختم میشود که سه لیست دیگز شامل تنظیماتی برای این کار می‌باشند.

و حالا در متد اصلی ExecuteUniqueIndexes() ما کوئری مورد نیاز برای ساخت ایندکس‌ها را ساخته ایم.

حال برای استفاده از متدExecuteUniqueIndexes() می‌بایست در متد Seed آن را صدا بزنیم تا کار ساخت ایندکس‌ها شروع شود، مانند کد زیر:

```
protected override void Seed(myDbContext context)
        {
            //  This method will be called after migrating to the latest version.

            //  You can use the DbSet<T>.AddOrUpdate() helper extension method
            //  to avoid creating duplicate seed data. E.g.
            //
            //    context.People.AddOrUpdate(
            //      p => p.FullName,
            //      new Person { FullName = "Andrew Peters" },
            //      new Person { FullName = "Brice Lambson" },
```

```
            //      new Person { FullName = "Rowan Miller" }
            //    );
            //
            context.ExecuteUniqueIndexes();
        }
```

چند نکته برای ایجاد ایندکس منحصر به فرد وجود دارد که در زیر به آنها اشاره می‌کنیم:
1. فیلدهای متنی باید حداکثر تا 350 کاراکتر باشند تا ایندکس اعمال شود.
2. همانطور که بالاتر اشاره شد برای فیلدهای دیتابیس OrderColumn اعمال نکنید که علت آن در بالا توضیح داده شد

دانلود فایل پروژه:

Sample_UniqueIndex.zip

نظرات خوانندگان

نویسنده: rahimi
تاریخ: ۱۳۹۱/۰۹/۲۳ ۱۶:۳۲

سلام ممنون از آموزش‌های خوبتون
می‌خواستم خواهش کنم ازتون مثال هایی که توضیح دادید را فایلشو هم قرار بدید تا بتونیم استفاده کنیم
ممنون

نویسنده: پدرام جباری
تاریخ: ۱۳۹۱/۰۹/۲۴ ۹:۲۷

سلام
خواهش می‌کنم
فایل پروژه به انتهای آموزش اضافه شد.

نویسنده: آرش مصیر
تاریخ: ۱۳۹۲/۰۲/۰۴ ۱۶:۶

با تشکر از سایت خوبتون من چند ماه پیش به این مشکل بر خورده بودم و در متد Seed مربوط به Context مستقیما اسکریپت ساخت ایندکس رو گذاشته بودم حالا می‌خوام از روشی که گفتید استفاده کنم

نویسنده: اکبر
تاریخ: ۱۳۹۲/۰۷/۱۲ ۲۱:۱۰

با سلام.
وقتی از این اتریبیوت بر روی پراپرتی email استفاده میکنم، و چون مقدار این فیلد الزامی نیست، وقتی کاربر این فیلد را خالی بگذارد خطای زیر را دریافت میکنم.

```
Violation of UNIQUE KEY constraint 'IX_Unique_Members_Email'. Cannot insert duplicate key in object
'dbo.Members'
```

با تشکر.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۷/۱۲ ۲۱:۲۸

از چه دیتابیسی استفاده می‌کنید؟ اگر SQL Server است که تا قبل از نگارش 2008 آن چنین اجازه‌ای رو به شما نمی‌ده تا یک فیلد منحصربفرد نال پذیر داشته باشید. اگر 2008 به بعد است، باید ایندکس فیلتر شده برای اینکار تعریف کنید. مثلا:

```
create unique nonclustered index idx on dbo.DimCustomer(emailAddress)
where EmailAddress is not null;
```

اطلاعات بیشتر اینجا و اینجا
بر همین مبنا باید قسمت ADD CONSTRAINT متد ExecuteUniqueIndexes را در صورت نیاز بازنویسی کنید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۲/۲۷ ۲۳:۱۳

یک نکته‌ی تکمیلی

از EF 6.1 به بعد ، دیگر نیازی به این مطلب نیست. تعریف ایندکس به صورت توکار میسر شده‌است .

عنوان: **اصول پایگاه داده - اندیس ها (indices)**
نویسنده: حامد خسروجردی
تاریخ: ۱۳۹۲/۰۸/۰۳ ۰:۵
آدرس: www.dotnettips.info
برچسب‌ها: Index, Database

با افزایش حجم بانک‌های اطلاعاتی دسترسی سریع به داده‌های مطلوب به یک معضل تبدیل می‌شود. بهمین دلیل نیاز به مکانیزم هایی برای بازیابی سریع داده‌ها احساس می‌شود. یکی از این مکانیزم‌ها اندیس گذاری (indexing) است. اندیس گذاری مکانیزمی است که به ما امکان دسترسی مستقیم (direct access) را به داده‌های بانک اطلاعاتی می‌دهد.

عمل اندیس گذاری وظیفه طراح بانک اطلاعاتی است که با توجه به دسترسی هایی که در آینده به بانک اطلاعاتی وجود دارد مشخص می‌کند که بر روی چه ستون هایی می‌خواهد اندیس داشته باشد. بعنوان مثال با تعیین کلید اصلی اعلام می‌کند که بیشتر دسترسی‌های آینده من بر اساس این کلید اصلی است و بنابراین بانک اطلاعاتی بر روی کلید اصلی اندیس گذاری را انجام می‌دهد. علاوه بر کلید اصلی می‌توان بر روی هر ستون دیگری از جدول نیز اندیس گذاشت که همانطور که گفته شد این مسئله بستگی به تعداد دسترسی آینده ما از طریق آن ستون‌ها دارد.

پس از اندیس گذاری بر روی یک ستون بسته به نوع اندیس فایلی در پایگاه اطلاعاتی ما ایجاد می‌شود که به آن فایل اندیس (index file) گفته می‌شود. این فایل یک فایل مبتنی بر رکورد (record-based) است که هر رکورد آن محتوی زوج *کلید جستجو* - *اشاره گر* می باشد. کلید جستجو را مقدار ستون مورد نظر و اشاره گر را اشاره گری به رکورد مربوط به ان می‌تواند در نظر گرفت.

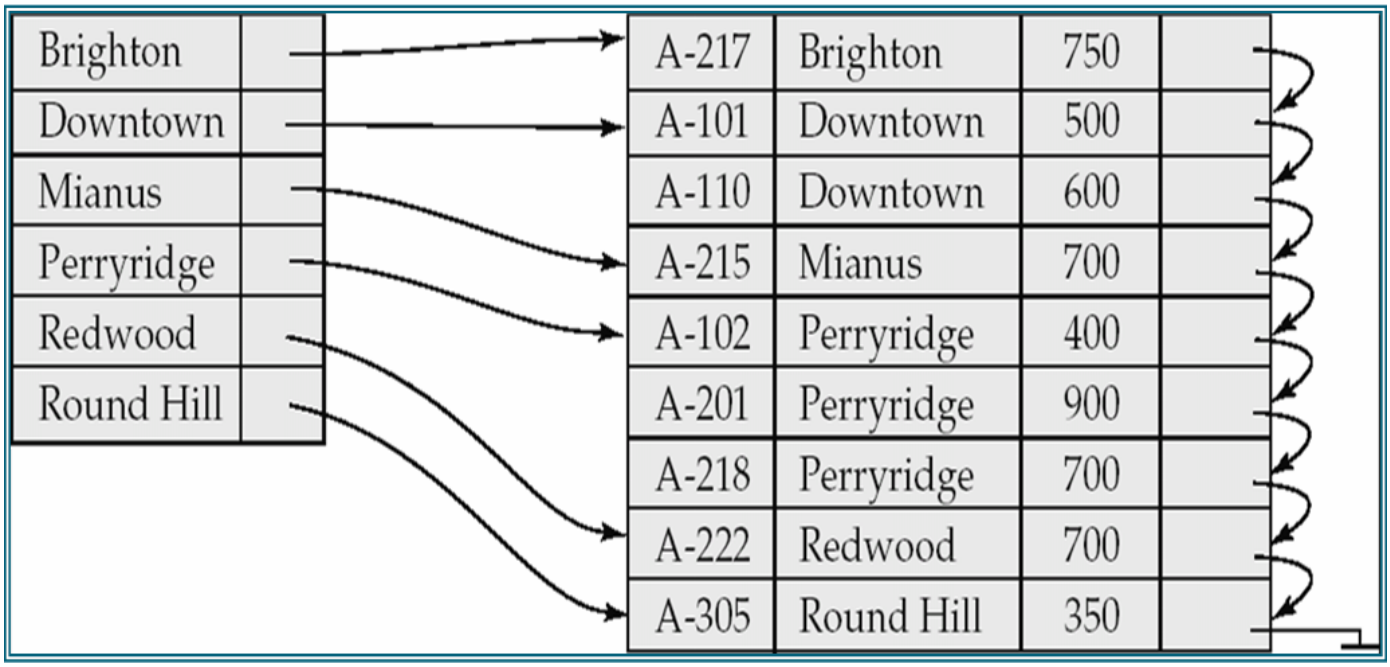
توجه داشته باشید که اندیس گذاری و مدیریت اندیس ها، همانطور که در این مقاله آموزشی گفته خواهد شد سر بار هایی (از نظر حافظه و پردازش) را بر سیستم تحمیل می‌نمایند. بعنوان مثال با اندیس گذاری بر روی هر ستونی یک فایل اندیس نیز ایجاد می‌شود بنابراین اگر اندیس‌های ما بسیار زیاد باشد حجم زیادی از بانک اطلاعاتی ما را خواهند گرفت. مدیریت و بروز نگهداری فایل‌های اندیس نیز خود مسئله ایست که سربار پردازشی را بدنبال دارد. بنابراین توصیه می‌شود در هنگام اندیس گذاری حتما بررسی‌ها و تحلیل‌های لازم را انجام دهید و تنها بر روی ستون هایی اندیس بگذرید که در آینده بیشتر دسترسی‌های شما از طریق ان ستون‌ها خواهد بود.

عموما در بانک‌های اطلاعاتی دو نوع اندیس می‌تواند بکار گیری شود که عبارتند از :
اندیس‌های مرتب (ordered indices) : در این نوع کلیدهای جستجو (search-key) بصورت مرتب نگداری می‌شوند.
اندیس‌های هش (Hash indices) : در این نوع از اندیس‌ها کلیدهای جستجو در فایل اندیس مرتب نیستند. بلکه توسط یک تابع هش (hash function) توزیع می‌شوند.

در این مقاله قصد داریم به اندیس‌های مرتب بپردازیم و بخشی از مفاهیم مطرح در این باره را پوشش دهیم.

اندیس‌های متراکم (dense index):

اولین و ساده‌ترین نوع از اندیس‌های مرتب **اندیس‌های متراکم (dense)** هستند. در این نوع از اندیس‌ها وقتی بر روی ستونی می‌خواهیم عمل اندیس گذاری را انجام دهیم می‌بایست به ازای هر کلید - جست و جو (search-key) غیر تکراری در ستون مورد نظر، یک رکورد در فایل اندیس مربوط به ان ستون اضافه کنیم. برای روشن شدن بیشتر موضوع به شکل زیر توجه کنید.



شکل 1 – اندیس متراکم (sparse index)

همانطور که در تصوری مشاهده می‌کنید بر روی ستون دوم از این جدول (جدول سمت راست)، اندیس متراکم (dense) گذاشته شده است. بر همین اساس به ازای هر کدام از اسامی خیابان‌ها یک رکورد در فایل اندیس (جدول سمت چپ) آورده شده است. در فایل اندیس می‌بینید که در کنار کلید جستجو یک اشاره گر نیز به جدول اصلی وجود دارد که در هنگام دسترسی مستقیم (direct access) از این اشاره گر استفاده خواهد شد. دقت کنید که کلیدهای جستجو در فایل اندیس بصورت مرتب نگهداری شده اند که نکته ای کلیدی در اندیس‌های مرتب می‌باشد.

مرتب بودن فایل اندیس موجب می‌شود که ما در هنگام جستجوی کلید مورد نظرمان در جدول اندیس بتوانیم از روش‌های جستجویی نظری جست و جوی دو دویی استفاده کنیم و در نتیجه سریع‌تر کلید مورد نظر را پیدا کنیم. این مسئله باعث ببهبود کارایی می‌شود. بعنوان مثال فرض کنید در فایل اندیس یک ملیون رکورد داریم. در این صورت برای یافتن کلید مورد نظرمان در جدول اندیس بروش جست و جوی دو دویی تنها کافی است 20 عمل مقایسه انجام دهیم. بنابراین می‌بینید که مرتب نگهداشتن جدول اندیس چقدر در سرعت بازیابی، تاثیر دارد.

نکته مهمی که در اندیس‌های متراکم باید به آن دقت شود اینست که ما به ازای کلیدهای جستجوی غیر تکراری یک رکورد در جدول اندیس نگهداری می‌کنیم. برای مثال در شکل بالا در ستون مورد نظر ما دو رکورد برای Downtown و سه رکورد برای Perryridge وجود دارد. این در حالی است که در فایل اندیس فقط یک Downtown و Perryridge داریم.

در اندیس‌های متراکم ما امکان دو نوع دسترسی را داریم :
دسترسی مستقیم (direct access)
دسترسی ترتیبی (sequential access)

دسترسی مستقیم :

توجه داشته باشید که در هنگام کار با یک جدول، فایل‌های اندیس آن به حافظه اصلی آورده می‌شوند (البته ممکن است که بخشی از فایل‌های اندیس به حافظه اصلی نیایند). این در حالی است که فایل اصلی جدول در حافظه جانبی قرار دارد. بنابراین در هنگام

بازیابی یک رکورد از برای یافتن محل ان رکورد نیازی به مراجعه زیاد به حافظه جانبی نیست. بلکه در حافظه اصلی بسرعت با یک عمل جستجو اشاره گر مربوط به رکورد مورد نظر در حافظه جانبی پیدا شده و مستقیما به آدرس همان رکورد می‌رویم و آن را می‌خوانیم. به این دسترسی، دسترسی مستقیم (direct access) می گوییم.

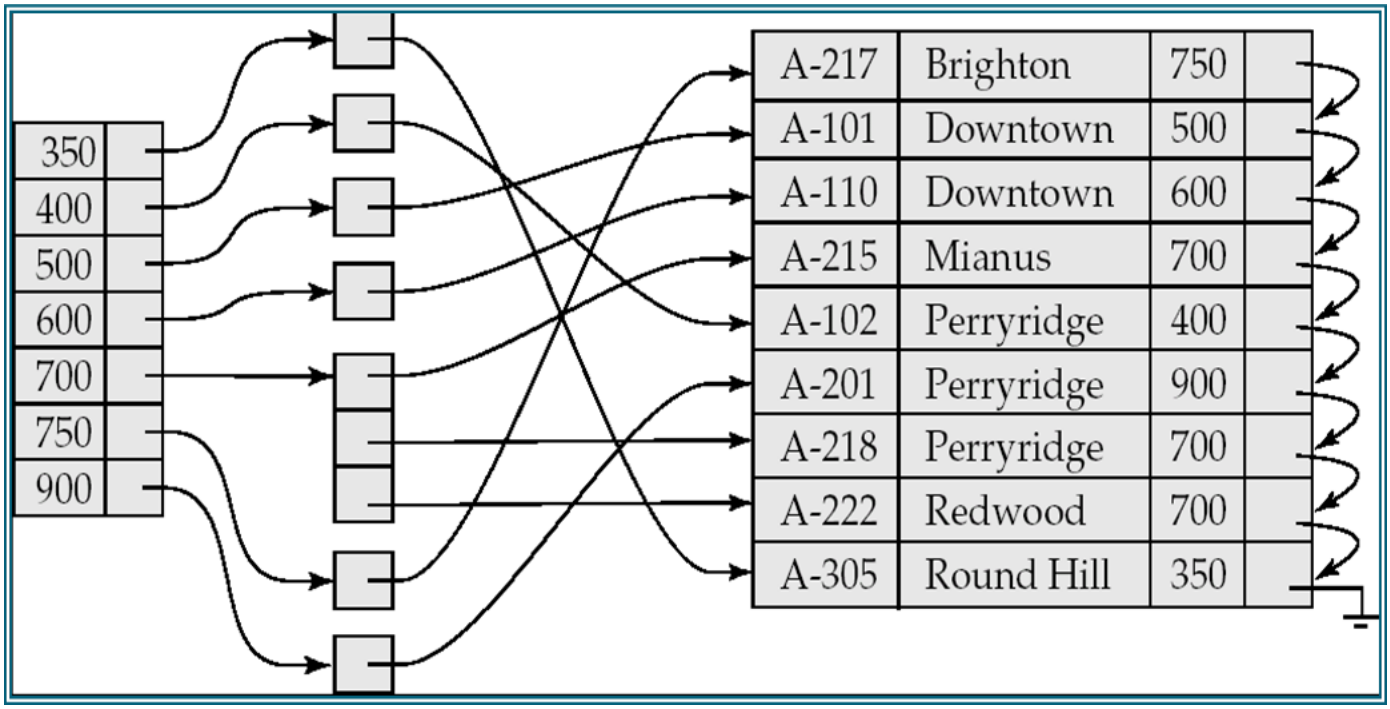
دسترسی ترتیبی :

در برخی از روش‌های اندیس گذاری علاوه بر دسترسی مستقیم امکان دسترسی بصورت ترتیبی نیز وجود دارد. در دسترسی ترتیبی این امکان وجود دارد که از یک رکورد خاص در جدول اصلی بتوانیم رکوردهای بعد از آن را به ترتیبی منطقی پیمایش کنیم. برای روشن‌تر شدن موضوع به شکل شماره 1 توجه کنید. در انتهای هر رکورد اشاره گری به رکورد منطقی بعدی مشاهده می‌کنید. این اشاره گرها امکان پیمایش و دسترسی ترتیبی را به ما می‌دهند. بعنوان مثال فرض کنید قصد داریم تمامی رکوردهای حاوی کلید Perryridge را بازیابی نماییم. از آنجایی که در جدول اندیس تنها برای یکی از رکوردهای حاوی این کلید اندیس داریم، برای بازیابی باقی رکوردها چه باید کرد؟ در چنین شرایطی ابتدا با دسترسی مستقیم اولین رکورد حاوی Perryridge را پیدا کرده و آن را بازیابی می‌کنیم. سپس از طریق اشاره گر انتهای آن رکورد، می‌توان به رکورد بعدی آن دست یافت و به همین ترتیب می‌توان یک به یک به رکوردهای دیگر دسترسی ترتیبی پیدا نمود.

دقت کنید که رکوردهای جدول ما بصورت فیزیکی مرتب نیستند. اما اشاره گرهای انتهای رکوردها طوری مقدار دهی شده اند که بتوان آنها را بصورت مرتب شده پیمایش نمود.

اندیس اولیه (primary index) و اندیس ثانویه (secondary index) :

بر روی ستون‌های یک جدول می‌توان چندین اندیس را تعریف نمود. اولین اندیسی که بر روی یک ستون از یک جدول گذاشته می‌شود اندیس اولیه (primary index) نامیده می‌شود. عموما این اندیس به کلید اصلی نسبت داده می‌شود، چراکه اولین اندیسی است که بر روی جدول زده می‌شود. توجه داشته باشید که رکوردهای جدول اصلی بر اساس کلیدهای جستجوی اندیس اولیه بصورت منطقی (با استفاده اشاره گرهای انتهای رکورد که توضیح داده شد) مرتب هستند. بنابراین امکان دسترسی بصورت ترتیبی وجود دارد. وقتی پس از اندیس اولیه اقدام به اندیس گذاری‌های دیگری می‌کنیم، اندیس‌های ثانویه را ایجاد می‌کنیم که اندکی با اندیس‌های اولیه متفاوت می‌باشند. در اندیس‌های ثانویه دیگر امکان پیمایش و دسترسی ترتیبی وجود ندارد چراکه اشاره گرهای انتهای رکوردها بر اساس اندیس اصلی (اولیه) مرتب شده اند. بنابراین ما در اندیس‌های ثانویه تنها دسترسی مستقیم خواهیم داشت. شکر زیر نمونه ای از یک اندیس ثانویه را نشان می‌دهد.

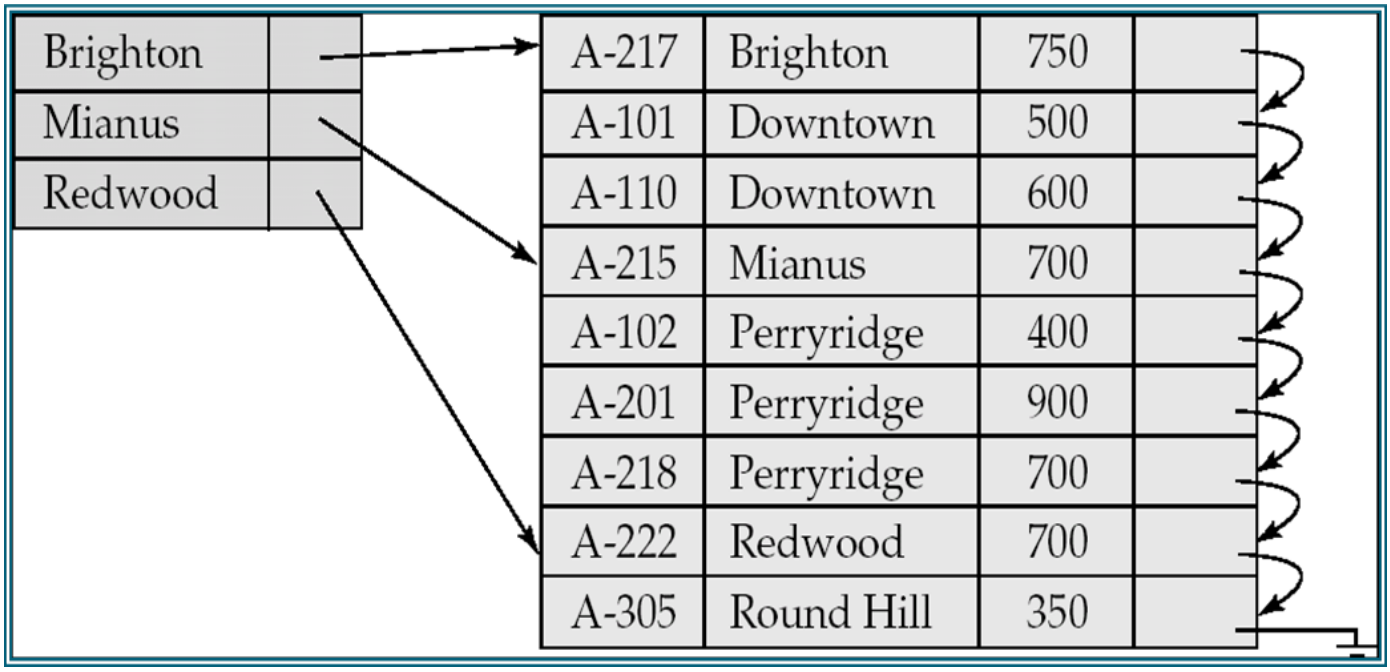


شکل 2 – اندیس ثانویه

همانطور که مشاهده می‌کنید علاوه بر اندیس اصلی (بر روی ستون 2) بر روی سومین ستون این جدول اندیس ثانویه متراکم زده شده است. دقت کنید که هر اشاره گر از جدول اندیس به یک باکت (bucket) اشاره دارد. در هر باکت اشاره گر هایی وجود دارد که به رکورد هایی از جدول اصلی اشاره می‌کنند. فلسفه وجود باکت‌ها اینست که در اندیس‌های ثانویه امکان دسترسی ترتیبی وجود ندارد. بنابراین برای مقادیری تکراری در جدول (مثلا عدد 700) نمی‌توان از اشاره گرهای انتهای رکوردها استفاده نمود. در چنین شرایطی در باکت‌ها اشاره گر مربوط به تمامی رکوردهای حاوی مقادیر تکراری یک کلید را نگهداری می‌کنیم تا بتوان به انها دسترسی مستقیم داشت. همانطور که مشاهده می‌کنید برای بازیابی رکوردهای حاوی مقدار 700 ابتدا از جدول اندیس (که مرتب است) باکت مربوطه را پیدا کرده و سپس از طریق اشاره گرهای موجود در این باکت به رکوردهای حاوی مقدار 700 دستیابی پیدا می‌کنیم.

اندیس‌های تنک (sparse index) :

در این نوع از اندیس‌ها بر خلاف اندیس‌های متراکم، تنها به ازای برخی از کلیدهای جستجو در جدول اندیس اشاره گر نگهداری می‌کنیم. بهمین دلیل فایل اندیس ما کوچکتر خواهد بود (نسبت به اندیس متراکم). در مورد اندیس‌های تنک نیز امکان دسترسی ترتیبی وجود دارد. در شکل زیر نمونه از اندیس تنک (sparse) را مشاهده می‌کنید.



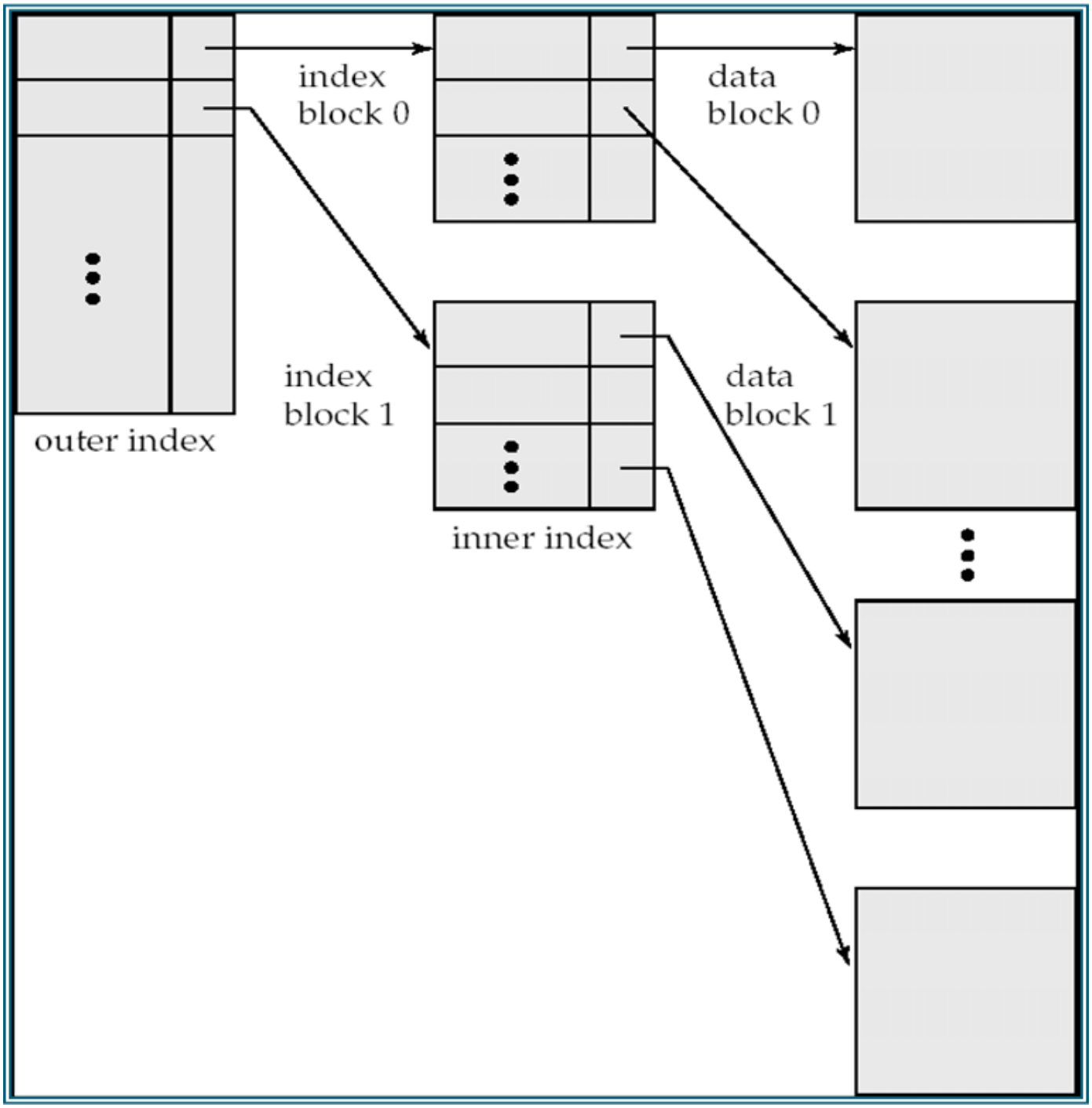
شکل 3 – اندیس تنک (sparse index)

همانند شکل 1، در این شکل نیز اندیس اولیه بر روی ستون دوم زده شده است. اما این بار از اندیس تنک استفاده گردیده است. مشاهده می‌کنید که از میان مقادیر مختلف این ستون تنها برای سه کلید Perryridge ، Brighton و Redwood در جدول اندیس رکورد درج شده است. بنابراین برای دست یابی به کلیدهای دیگر باید ابتدا محل تقریبی آن را با جستجو بر روی جدول اندیس پیدا نمود و سپس از طریق پیمایش ترتیبی به رکورد مورد نظر دست یافت. بعنوان مثال برای بازیابی رکورد حاوی مقدار Mianus ابتدا در جدول اندیس کلیدی که از Mianus کوچکتر باشد (یعنی Brighton) را پیدا می‌کنیم. سپس به رکورد حاولی Brighton می رویم و از آنجا با استفاده از اشاره گرهای انتهایی رکوردها به سمت رکورد حاوی Mianus حرکت می‌کنیم تا به آن برسیم.

نکته بسیار مهمی که در مورد اندیس‌های تنک مطرح می‌شود اینست که سیستم چگونه باید تشخیص دهد که کدام کلیدها را در جدول اندیس نگهداری کند. این تصمیم به مفهوم بلاک‌های حافظه و اندازه انها باز می‌گردد. می‌دانیم که واحد خواندن اطلاعات از حافظه بر اساس بلاک‌ها می‌باشد. این بدان معنی است که در هنگام خواندن رکوردهای جداول بانک اطلاعاتی، عمل خواندن بصورت بلاکی انجام می‌شود. هنگامی که بر روی یک جدول می‌خواهیم اندیس تنک بزنیم ابتدا باید ببینیم این جدول چند بلاک از حافظه را اشغال کرده است. سپس رکوردهای اول هر بلاک را پیدا کرده و به ازای هر بلاک آدرس و کلید جستجوی رکورد اول آن را در جدول اندیس نگهداری کنیم. بدین ترتیب ما به ازای هر بلاک از جدول یک رکورد در فایل اندیس خواهیم داشت و با تخصیص بلاک‌های جدید به ان، طبیعی است که اندیس‌های جدید نیز در فایل اندیس ذخیره خواهند شد.

اندیس‌های چند سطحی (multi-level index)

در دنیایی واقعی معمولا تعداد رکوردهای جداول مورد استفاده بسیار بزرگ است و این اندازه دائما در حال زیاد شدن می‌باشد. افزایش اندازه جداول باعث می‌شود که اندازه فایل‌های اندیس نیز رفته رفته زیاد شود. گفتیم برای کارایی هرچه بیشتر باید جدول اندیس مورد استفاده به حافظه اصلی آورده شود تا تعداد دسترسی‌های ما به حافظه جانبی تا حد امکان کاهش یابد. اما اگر اندازه فایل اندیس ما بسیار بزرگ باشد ممکن است حجم زیادی از حافظه اصلی را بگیرد یا اینکه در حافظه اصلی فضای کافی برای ان وجود نداشته باشد. در چنین شرایطی از اندیس‌های چند سطحی استفاده می‌شود. به بیان دیگر بر روی جدول اندیس نیز اندیس زده می‌شود. تعداد سطوح اندیس ما بستگی به اندازه جدول اصلی دارد و هر چه این اندازه بزرگ‌تر شود، ممکن است باعث افزایش تعداد سطوح اندیس شود. در شکل زیر ساختار یک اندیس دو سطحی را مشاهده می‌کنید.



نکته مهم در مورد اندیس‌های چند سطحی اینست که اندیس‌های سطوح خارجی (outer index) از نوع تنک هستند. این مسئله به این دلیل است که اندازه اندیس‌ها کوچک‌تر شود. چراکه اگر اندیس خارجی از نوع متراکم باشد به این معناست که به ازای هر رکورد غیر تکراری باید یک رکورد در فایل اندیس نیز آورده شود و این مسئله باعث بزرگ شدن اندیس می‌شود. بهمین دلیل سطوح خارجی را در اندیس‌های چند سطحی از نوع تنک می‌گیرند. تنها آخرین سطحی که مستقیما به جدول اصلی اشاره می‌کند از نوع متراکم است. به این سطح از اندیس، اندیس داخلی (inner index) گفته می‌شود.

بروز نگهداشتن اندیس‌ها :

با انجام عملیات درج و حذف بروی جداول، جداول اندیس مربوطه نیز باید بروز رسانی شوند. در این بخش قصد داریم به نحوه بروز رسانی جداول اندیس در زمان حذف و درج رکورد بپردازیم.

بروز رسانی در زمان حذف :

اندیس متراکم :

هنگامی که رکوردی از جدول اصلی حذف می‌شود، در صورتی که بر روی ستون‌های آن اندیس‌های متراکم داشته باشیم، پس از حذف رکورد اصلی باید ابتدا کلید جستجوی ستون مربوط را در جدول اندیس پیدا کنیم. در صورتی که از این کلید تنها یک مقدار در جدول اصلی وجود داشته باشد، اندیس آن را از فایل اندیس حذف کرده و اشاره گرهای انتهای رکوردها را بروز رسانی می‌کنیم. اما اگر از کلید مورد نظر چندین مورد وجود داشته باشد نباید رکورد مورد نظر در جدول اندیس پاک شود. بلکه تنها ممکن است نیاز به ویرایش اشاره گر اندیس باشد. ویرایش در زمانی رخ می‌دهد که اشاره گر جدول اندیس مستقیما به رکوردی اشاره کند که حذف شده باشد، در این صورت باید اشاره گر اندیس را ویراش نمود تا به رکورد بعدی اشاره نماید.

اندیس تنک :

همانند روش قبل ابتدا رکورد اصلی را از جدول حذف می‌کنیم. سپس در فایل اندیس بدنبال کلید جستجوی مربوط به رکورد حذف شده می‌گردیم. در صورتی که کلید مورد نظر در جدول اندیس پیدا شد کلید جستجوی رکورد بعدی در جدول اصلی را جایگزین آن می‌کنیم. چنانچه کلید مربوط به رکورد بعدی در جدول اندیس وجود داشته باشد نیازی به جایگزینی نیست و باید فقط عمل حذف اندیس را انجام داد.

اگر کلید مورد جستجو در جدول اندیس وجود نداشته باشد نیاز به انجام هیچ عملی نیست. در پایان باید اشاره گرهای انتهای رکوردها را ویرایش نمود تا ترتیب منطقی برای پیمایش ترتیبی حفظ شود.

بروز رسانی در زمان درج:

اندیس متراکم:

در هنگام درج یک رکورد جدید، ابتدا باید کلید موجود در رکورد جدید را در جدول اندیس جستجو نمود. در صورتی که کلید مورد نظر در جدول اندیس یافت نشد، باید رکوردی جدیدی در فایل اندیس درج کرد و اشاره گر آن طوری مقدار دهی نمود تا به رکورد جدید اشاره نماید. اگر کلید مورد نظر در جدول اندیس وجود داشته باشد دیگر نیازی بروز رسانی اندیس‌ها نیست و تنها کافی است اشاره گرهای انتهای رکوردها بروز رسانی شوند.

اندیس تنک :

در مورد اندیس‌های تنک کمی پیچیدگی وجود دارد. در صورتی که رکورد جدید باعث تخصیص بلاک (block) جدیدی از حافظه به جدول شود، باید به ازای آن بلاک یک اندیس در جدول اندیس‌ها ایجاد شود و آدر آن بلاک را (که در واقع آدرس رکورد جدید نیز می‌شود) در اشاره گرد اندیس قرار داد. اما درغیز این صورت (در صورتی که رکورد در بلاک‌های موجود ذخیره شود) نیازی به بروز رسانی جدول اندیس‌ها وجود ندارد.

نوع دیگری از اندیس‌های مرتب نیز وجود دارد که اندیس های B-Tree هستند که در سیستم‌های اطلاعاتی دنیای واقعی بیشتر از آنها استفاده می‌شود. به امید خدا در مطالب بعدی این اندیس‌ها را نیز مورد بررسی قرار خواهیم داد.

موفق و پیروز باشید.

نظرات خوانندگان

نویسنده: مجید_فاضلی نسب
تاریخ: ۱۳۹۲/۰۸/۰۳ ۱۳:۲۹

سلام و درود.
فرق Indices و Index چیست ؟

نویسنده: حامد خسروجردی
تاریخ: ۱۳۹۲/۰۸/۰۴ ۹:۵۱

سلام. indices همون جمع index هستش.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۸/۰۴ ۱۳:۳۸

Indices مستقیما از زبان لاتین گرفته شده است. آمریکایی‌ها بیشتر indexes را بکار می‌برند بجای Indices. هر دو هم صحیح هستند.

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

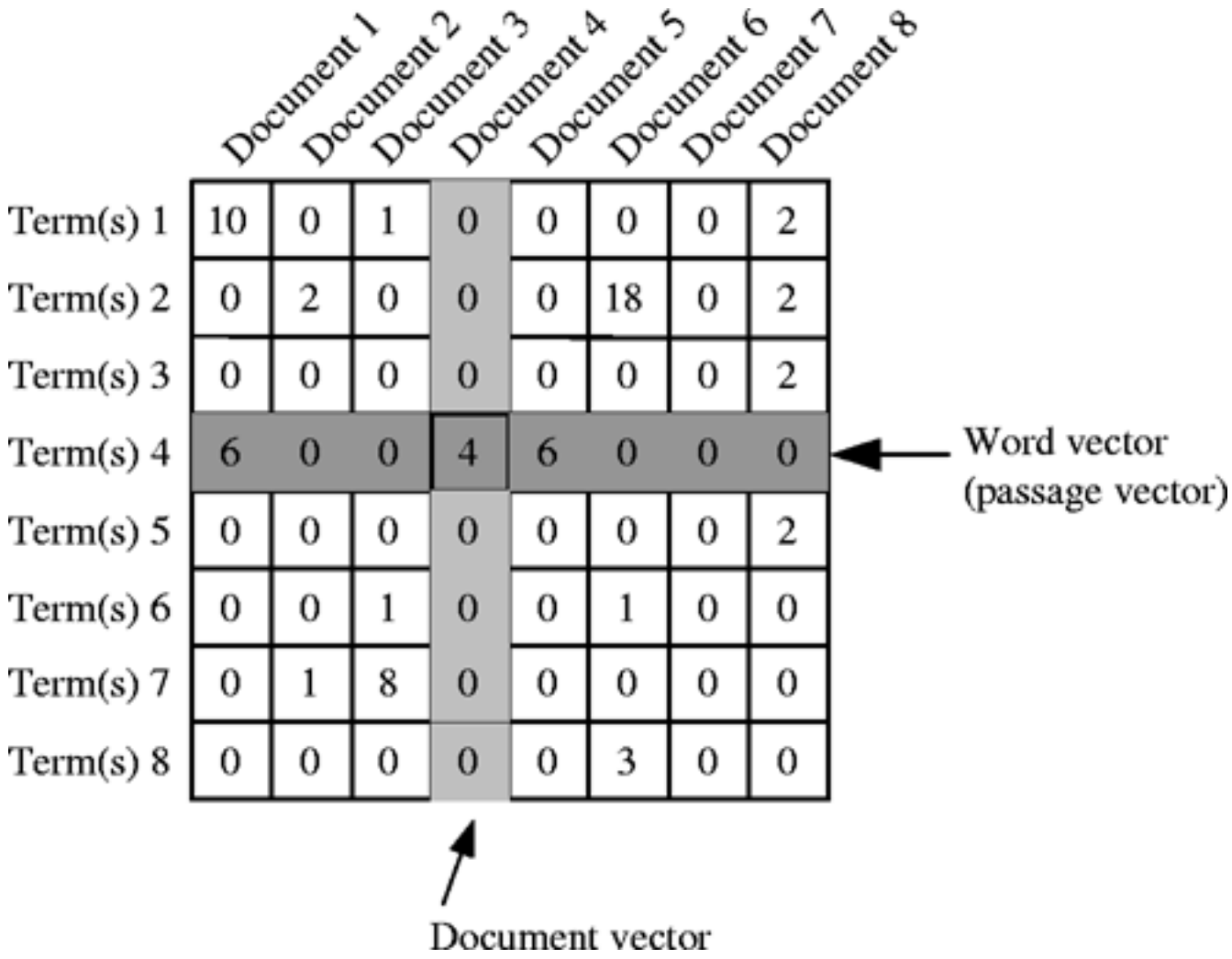
همان بسط دادن پرس و جوی شما است. دقت داشته باشید که الگوریتم جستجو هیچ اطلاع و دانشی از معنای لغات مذکور ندارد و تنها بر اساس تحلیل‌های ریاضی به این نتیجه می‌رسد که در بخش‌های بعدی چگونگی آن را برای شما بازگو خواهیم نمود. یکی از برتری‌های مهم LSI نسبت به روش‌های مبتنی بر کلمات کلیدی (keyword based) این 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.

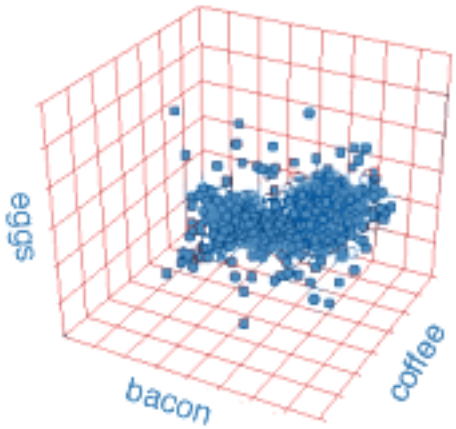
این ماتریس، بازتابی از کل مجموعه متنی را به ما می‌دهد. بعنوان مثال اگر بخواهیم ببینیم در سند i چه لغاتی وجود دارد، تنها کافی است به سراغ سطر iام از ماتریس برویم (البته در صورتی که ماتریس ما سند – لغت باشد) وآن را بیرون بکشیم. به این سطر در اصطلاح بردار سند (document vector) گفته می‌شود. همین کار را در 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

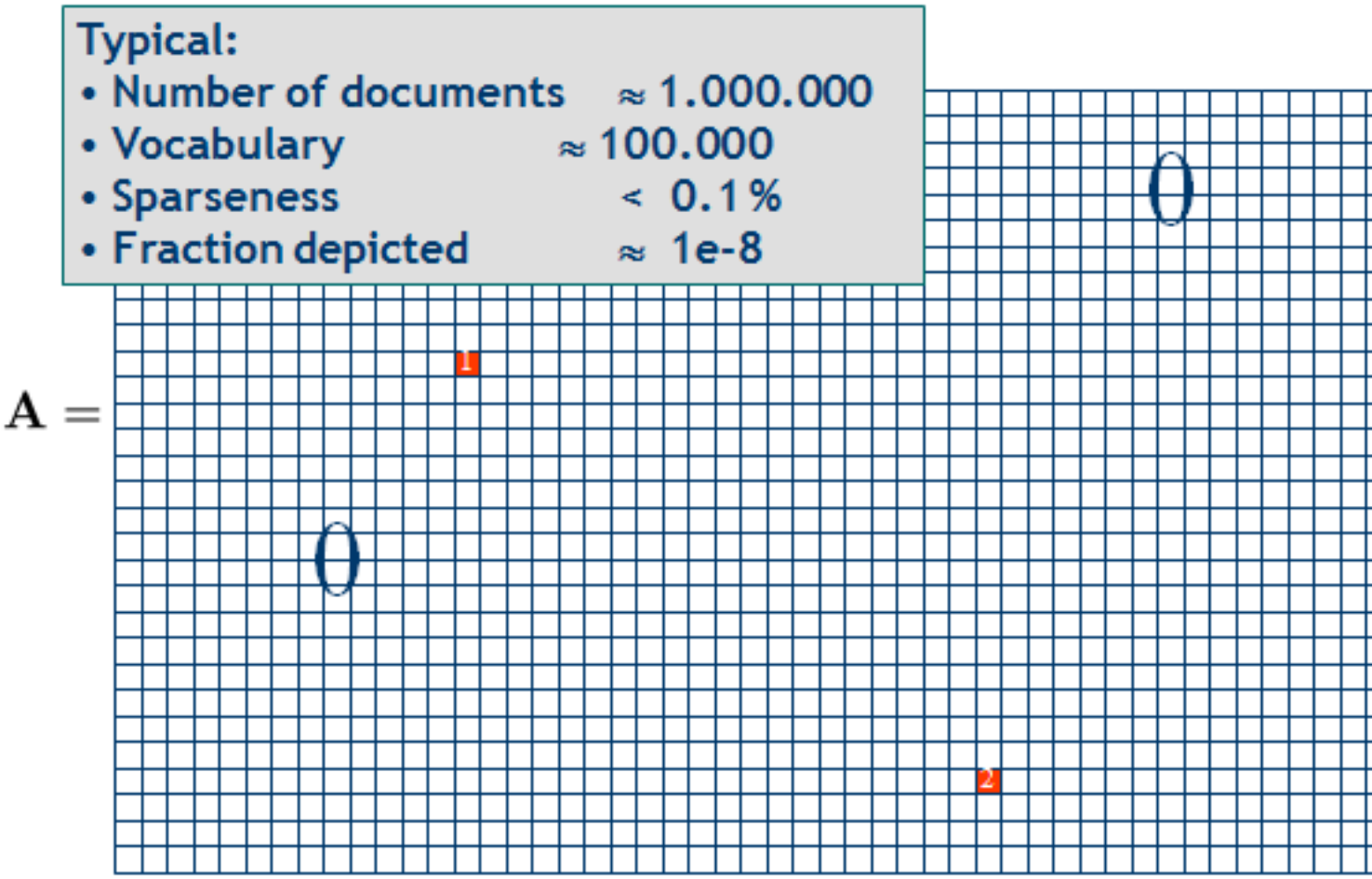
فضای برداری مثال ما قرار گیرند. علت نزدیک بودن این بردارها اینست که تعداد دفعات تکرار bacon، eggs و coffee در انها مشابه به هم بوده است. بنابراین می‌توان گفت که این نقاط (یا سفارشات مربوط به انها) به یکدیگر شبیه می‌باشند. در مورد فضای برداری مجموعه از اسناد نیز وضع به همین ترتیب است. اسنادی که لغات مشترک 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).