




Rencontre R19

Images

Bases de données et programmation Web



- ❖ Gestion des images 
- ❖ Monitorer la performance  
- ❖ Stratégies d'optimisation  



❖ Gestion des images

- ◆ L'objectif sera de stocker des images et de pouvoir les afficher dans les pages Web au besoin.
 - Il y aura peu de similarités avec la gestion des images du cours Prog Web services. Nous allons tenter d'isoler la gestion des fichiers dans SQL Server et d'utiliser des méthodes différentes pour apprendre de nouvelles choses.



❖ Dilemme : où stocker les images ?

- **Choix 1** : Dans la **base de données**. Le type utilisé serait **varbinary(max)** et le 'Handle' vers les images seraient stockées en tant que **BLOB**. (**B**inary **L**arge **O**bject, donc des gros amas de bytes)
- **Choix 2** : Dans le **système de fichiers** du serveur. Il n'y aurait donc aucun fichier image dans la BD, mais plutôt une **référence** qui permettrait de retrouver l'image dans le système de fichiers. (Cours 4W6 Prog. Web orientée Services)
- **Choix 3** : Dans un **cloud storage** qui gère les images pour nous. (Azure Blob Storage, Amazon S3, etc.) Permet de simplifier le stockage des fichiers lourds. L'application Web n'a pas besoin de rouler sur le même serveur que les images.
 - Nous écarterons cette option car elle est coûteuse et surtout pertinente seulement à grande échelle.



❖ Choix 1 : « Stocker » les images dans la BD

◆ Avantages possibles

- Pas besoin de nouvelles **stratégies de backup**, comme les images sont avec les autres données. (Les backups de la BD sont beaucoup plus **lourds** par contre)
- Plus facile d'assurer l'**intégrité des données** en lien avec les images.
- Facile de **limiter l'accès** aux images grâce à des **permissions SQL**.

◆ Désavantages possibles

- Les images peuvent prendre jusqu'à **deux fois plus d'espace** dans une BD que dans le **file system** à cause de leur conversion en **varbinary**.
- Réduction de la **performance**. Généralement, récupérer une image dans la BD est plus lent que récupérer une image dans le file system... et dans tous les cas, la base de données a plus de données à gérer que si les images étaient stockées ailleurs.
- Une BD met en cache les données fréquemment accédées. Si des images prennent beaucoup de place dans le cache, ça fait moins d'espace pour d'autres données plus petites et cela rend le **cache moins performant**.



❖ Choix 2 : Stocker les images dans le **File System**

◆ **Avantages** possibles

- Meilleure **performance** : Le File System est généralement mieux adapté pour charger des fichiers lourds et stocker une grande quantité de fichiers sans problèmes.
- **Simplicité** : Évite certains défis de conversion. Avec la plupart des applications côté serveur, récupérer des fichiers dans le File System est très simple et n'apporte pas de défi particulier.

◆ **Désavantages** possibles

- Nécessite une autre **stratégie de backup**.
- Nécessite une **stratégie d'accès** pour éviter que n'importe qui fouille dans les dossiers.
- La BD a moins de contrôle sur **l'intégrité des données** qui sont à l'extérieur de la BD, bien entendu. Par exemple, facile d'oublier de supprimer une image lorsqu'on DELETE la rangée associée dans la BD.



❖ Dans les diapos qui suivent, nous allons faire un mix de 1) et 2) pour que la gestion des images se fasse dans la BD à travers le Filestream de SQL Server.

- ◆ Configurer FILESTREAM avec SQL Server
- ◆ Préparer une table qui 'stockera' des images
- ◆ Action + Vue Razor + ViewModel pour upload d'une image dans l'appli
- ◆ Action + Vue Razor + ViewModel pour afficher des images dans l'appli



❖ Configurer **FILESTREAM** avec SQL Server

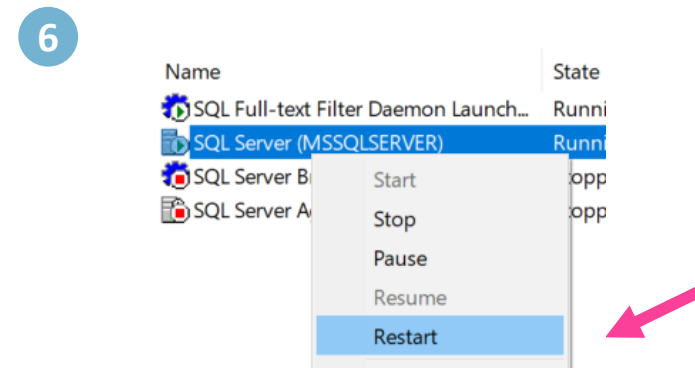
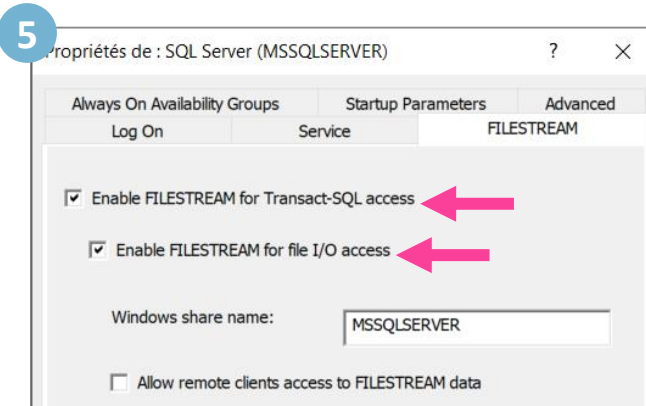
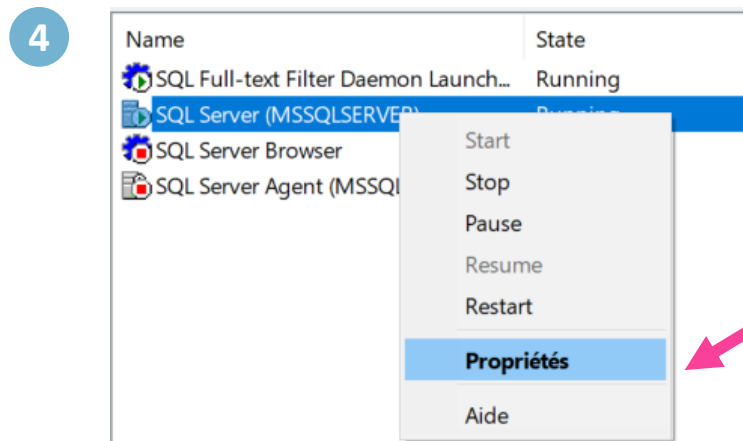
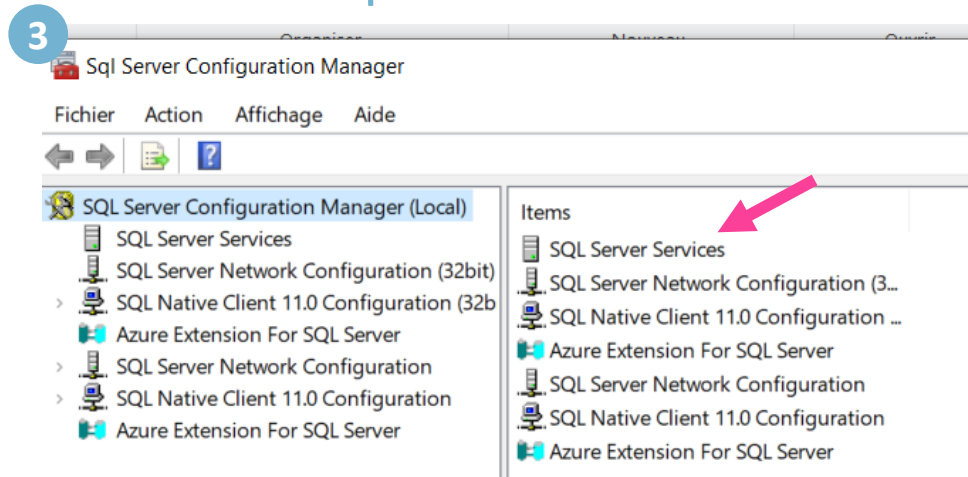
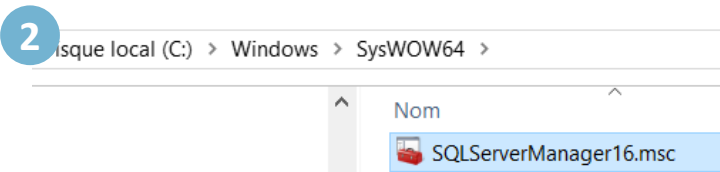
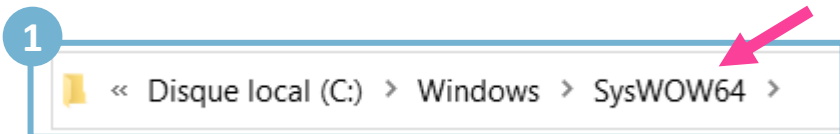
- ◆ SQL Server nous permet de stocker des fichiers en **varbinary(max)** ... mais il nous propose aussi une alternative pour nous accompagner dans le stockage des images dans le **File System** !
 - Cette alternative est le **FILESTREAM**.
- ◆ Rappel : Avec l'usage d'**Evolve**, on exécute les **migrations** à l'aide de **commandes** plutôt qu'exécuter les scripts dans **SSMS**.
 - **Exception** : On devait exécuter **CREATE DATABASE** à la main, car cette instruction ne peut pas être utilisée dans une **transaction** avec **Evolve**.
 - La configuration du **FILESTREAM** devra aussi être faite **à la main**, sans **Evolve**. Ça veut dire qu'à chaque fois que vous changez de poste de travail, il faudra configurer **FILESTREAM** à nouveau en créant la BD.
 - Il est donc suggéré de se garder à portée de main un **script SQL** qui crée la BD et qui configure **FILESTREAM**. C'est pour cela qu'on met un fichier de commande dans le dossier .SqlScript du projet.



Msg 5591, Niveau 16, État 3, Ligne 5
FILESTREAM feature is disabled.
Msg 1921, Niveau 16, État 4, Ligne 8
Invalid filegroup 'FG_Images' specified.

❖ Configurer **FILESTREAM** avec SQL Server (au cégep c'est fait)

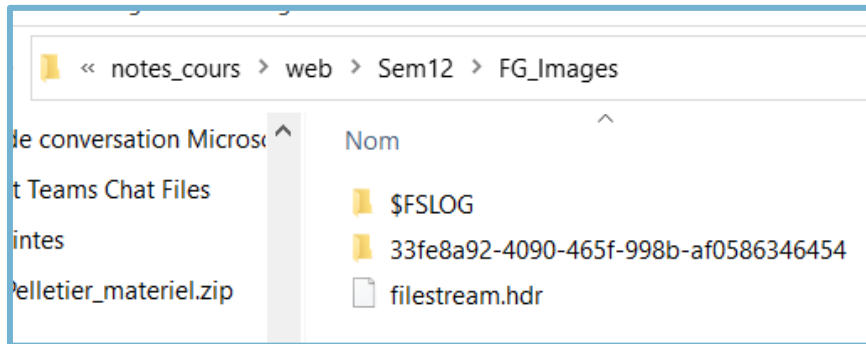
- ◆ Si vous avez une erreur qui mentionne que **FILESTREAM** n'est pas activé sur le serveur SQL, voici la marche à suivre pour l'activer :





❖ Configurer **FILESTREAM** avec SQL Server

- ◆ Voici un exemple de script SQL qui crée une BD et configure un **FILESTREAM**.



- Assurez-vous que le dossier parent existe (Ici, « **C:\EspaceLabo** »), mais pas le dossier à créer. (Ici, « **FG_Images** »)



```
CREATE DATABASE Sem12
GO
USE Sem12
GO

EXEC sp_configure filestream_access_level, 2 RECONFIGURE

ALTER DATABASE Sem12
ADD FILEGROUP FG_Images CONTAINS FILESTREAM;
GO

ALTER DATABASE Sem12
ADD FILE (
    NAME = FG_Images,
    FILENAME = 'C:\EspaceLabo\FG_Images'
)
TO FILEGROUP FG_Images
GO
```



❖ Préparer une table qui stockera des images

◆ Les deux colonnes importantes sont **Identifiant** et **Photo**.

- Sans colonne de type **uniqueidentifier NOT NULL ROWGUIDCOL**, il sera impossible d'ajouter la colonne de type **varbinary(max) FILESTREAM** car chaque fichier a besoin d'un identifiant unique (généré aléatoirement dans ce cas) pour être stocké.

```
CREATE TABLE Animaux.Animal (  
    AnimalID int IDENTITY NOT NULL,  
    Nom nvarchar(100) NOT NULL,  
    Identifiant uniqueidentifier NOT NULL ROWGUIDCOL ,  
    CONSTRAINT PK_Animal.AnimalID PRIMARY KEY (AnimalID)  
)
```

- De plus, ces deux contraintes doivent avoir été créées avant de pouvoir ajouter la colonne pour le fichier.

```
ALTER TABLE Animaux.Animal ADD CONSTRAINT UC_Animal_Identifiant  
UNIQUE (Identifiant);  
GO
```

- Enfin, on peut ajouter la colonne qui s'occupera de ranger le fichier et de garder une **référence** vers le fichier.

```
ALTER TABLE Animaux.Animal ADD CONSTRAINT DF_Animal_Identifiant  
DEFAULT newid() FOR Identifiant;  
GO
```

- On a mis **NULL** ici car l'image est **optionnelle**, mais c'est libre à vous.

- Gardez bien à l'esprit que l'image **n'est pas dans la BD** : elle est dans le **File System**, mais le **FILESTREAM** s'occupe de l'y ranger.

```
ALTER TABLE Animaux.Animal ADD  
Photo varbinary(max) FILESTREAM NULL;  
GO
```



❖ Avec la gestion de images à travers le Filestream

◆ Nous avons les images liées à la BD.

- Mais elles sont réellement dans le File System
- Si on supprime une photo dans la table Animaux.Animal
 - Le Filestream va s'occuper de supprimer la photo dans le File System.

◆ Avantages

- Pas besoin de nouvelles **stratégies de backup**, comme les images gérées à travers la BD
- Plus facile d'assurer l'**intégrité des données** en lien avec les images.
- Facile de **limiter l'accès** aux images grâce à des **permissions SQL**.
- Meilleure **performance** : Les images sont en fait dans le File System qui est généralement mieux adapté pour charger des fichiers lourds et stocker une grande quantité de fichiers sans problèmes.



❖ Préparer une table qui stockera des images

- ◆ Au moment opportun, on peut ensuite **scaffold** les **Models** dans notre application Web si la BD est dans l'état souhaité.

▲ Data
▸ C# Sem12Context.cs

▲ Models
▸ C# Image.cs

```
dotnet ef dbcontext scaffold Name=Sem12 Microsoft.EntityFrameworkCore.SqlServer -o Models --context-dir Data --data-annotations --force
```

- N'oubliez pas l'option **--force** si jamais des **Models** existants doivent être écrasés. (Remplacés)

- Remarquons que le type pour le fichier image est un **array de bytes** et que le type de l'identifiant unique du fichier est **Guid**.

```
[Table("Animal", Schema = "Animaux")]  
[Index("Identifiant", Name = "UC_Animal_Identifiant", IsUnique = true)]  
public partial class Animal  
{  
    [Key]  
    [Column("AnimalID")]  
    public int AnimalId { get; set; }  
  
    [StringLength(100)]  
    [Unicode(false)]  
    public string Nom { get; set; } = null!;  
  
    public Guid Identifiant { get; set; } ←  
  
    public byte[]? Photo { get; set; } ←  
}
```




❖ Upload une image dans l'application Web

◆ ViewModel et Action

- Pour que l'utilisateur puisse envoyer le fichier à l'aide d'un formulaire dans la **vue Razor**, nous utiliserons un **IFormFile** qui accompagne l'**objet** à ajouter dans la BD.

- L'insertion est assez standard, à l'exception qu'on récupère le **fichier** dans le **IFormFile** et qu'on l'intègre à notre objet sous forme d'**array de bytes**.



```
public class ImageUploadVM
{
    [Required (ErrorMessage = "Il faut joindre un fichier image.")]
    public IFormFile FormFile { get; set; } = null!;
    [Required (ErrorMessage = "Il faut spécifier le nom de l'image.")]
    [DisplayName("Nom de l'image en minuscules")]
    public string NomImage { get; set; } = null!;
}
```

```
[HttpPost]
public async Task<IActionResult> AjouterImage(ImageUploadVM iuvm)
{
    if (ModelState.IsValid)
    {
        // Trouver l'animal choisi par l'utilisateur
        Animal? animal = await _context.Animals.FirstOrDefaultAsync(x => x.Nom == iuvm.NomAnimal);

        // Récupérer l'image dans iuvm.FormFile
        // Remplir la propriété Animal.Photo
        if(iuvm.FormFile != null && iuvm.FormFile.Length >= 0)
        {
            MemoryStream stream = new MemoryStream();
            await iuvm.FormFile.CopyToAsync(stream);
            byte[] fichierImage = stream.ToArray();
            animal.Photo = fichierImage;
        }

        await _context.SaveChangesAsync();
        return View("Index");
    }
    return View();
}
```



❖ Upload une image dans l'application Web

◆ Vue Razor

```
public class ImageUploadVM
{
    4 references
    public IFormFile? FormFile { get; set; }

    6 references
    public Image Image { get; set; } = null!;
}
```

```
@model Sem12.ViewModels.ImageUploadVM
```

- On peut commencer par **auto-générer** une vue Razor avec le template **Create**.

Les modifications les plus importantes sont :

- Utiliser le **ViewModel** qu'on a créé pour pouvoir utiliser un **IFormFile**.
- Ajouter **enctype=** dans le **<form>**.
- Modifier un peu l'input du fichier pour l'envoyer via le **IFormFile**. (Car par défaut l'input auto-généré ne permet pas exactement d'envoyer un fichier)


```
<form asp-action="Create" enctype="multipart/form-data">
  <div asp-validation-summary="ModelOnly" class="text-danger"></div>
  <div class="form-group">
    <label asp-for="Image.Nom" class="control-label"></label>
    <input asp-for="Image.Nom" class="form-control" />
    <span asp-validation-for="Image.Nom" class="text-danger"></span>
  </div>
  <div class="form-group">
    <label for="FormFile">Fichier image : </label>
    <input id="FormFile" name="FormFile" type="file" accept="image/*" class="form-control-file" />
    <span asp-validation-for="FormFile" class="text-danger"></span>
  </div>
  <div class="form-group">
    <input type="submit" value="Create" class="btn btn-primary" />
  </div>
</form>
```



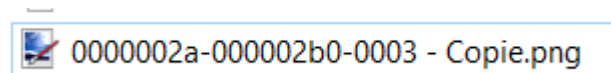
❖ Upload une image dans l'application Web

◆ Coup d'œil dans le **File Group** :

- Si vous fouillez un peu dans le dossier qui est utilisé par **FILESTREAM**, vous pourrez retrouver le fichier de l'image qui été téléversé dans l'application Web.

Nom	Modifié le	Type	Taille
 0000002a-000002b0-0003	2023-04-16 15:05	Fichier	463 Ko

- Si vous êtes vraiment curieux, vous pouvez en faire une **copie**, la renommer avec son **extension**, et ouvrir l'image :





❖ Afficher des images dans l'application Web

◆ ViewModel et Action

- Attention, ce n'est pas le même **ViewModel** que pour **upload** les images. Cette fois le champ supplémentaire servira à stocker un **énorme string** qui représente l'image.
- Cette action envoie la liste photos à la **vue Razor**, mais elle envoie également tous les fichiers images sous forme d'un **groooooooooo string**. (Ça fait une réponse HTTP potentiellement très lourde ! Cela dit, le client n'aura pas à faire des requêtes HTTP supplémentaires pour afficher les images individuellement)

```
public class AnimalPrefereVM
{
    public List<string>? NomsAnimals { get; set; }

    // On a ici une liste qui contient des strings et des NULL
    // (NULL signifie qu'aucune image n'a été trouvée pour un animal)
    // (Un string signifie qu'une image a été trouvée pour un animal)
    public List<string?>? PhotosAnimals { get; set; }
}
```

```
// On remplit la liste apvm.PhotosAnimals qui contiendra les PHOTOS des animaux préférés
apvm.PhotosAnimals = AnimalsPreferes
    .Select(x => x.Photo == null ? null : $"data:image/png;base64, {Convert.ToString(x.Photo)}")
    .ToList();
```

Ce bidule « convertit » l'image en string pour l'envoyer

Attention aux fautes de frappe ! Ce string est facile à casser.

Assurez-vous que votre image soit au format **.png**. On aurait pu noter le **MimeType** lors de l'upload pour gérer plus de formats.

data:image/png;base64,iVBORw0KGgoAAAANSUheUGAAAXUAAAJ2CAIAAACmaDgdAAAAAXNSR0IARs4c6QAAAArnQU1BAACxjwv8YQUAAAAJcEhZcwAADsMAAA7DAdcvqGQAAAP+LSURBVHhe7Plbk+xKt6HAHXAg7n1aa32nfer61JKT1Widj3Q2DPdKqKkC4U13dqgR/jm+tcw/mrD2+B85QhQWYh3a30L77jWmr07qwoF+H3eKUABKAAPVPecca32bmc2KpE5coyR1wHxkoMoVP5//X/+H7PXQJwnZa3188HGKpxSUVvly1a/iJHW6j8bz2Ho6ir1Widj3M3ANwBAJ1jctSthRE9bWg/CqhftfebvRqd3ayXhJvz4yerekyMwpmJBGYmj_fX0BJ+2c4LNN50DL/OqMdlTErvtkhr5ShnjPl0syYixqnyY1BLVveie1L+qmeV1n0/kqGtmVMT3ag7Wtwlucx5uXhO/HC/Wq1TE6Gbu9KHbFFqBuqMNVfnNBkbLzyEzj7Iyya+KYmDVwe8P+6k1c6PXRvQzxpNj2wwKSSGgq+G6k5n204MG1XqXYeqtJVZNbcPjCX/+AggA4PvR1Hkv7isHsDo/ADIA36xkR/ETUUCiYcYihw205TKtYGSZgZacnDf5gc+em8ktVssTnBgT/q1T4Gm0U1V8ZrEi+K1r2qyj7fv9QJHw4f+5W0S06PaQzWuBUDapw0bJw6XW6nGoxxa7ddlclPT32T20nEwMzdbMFBFLD45cvoXGh9mZ0TUJXHW6ASy7idM4w1/HWP9S5G56mcMeF8GcW9VLy8t1Drx2M9802/7pPCAEZFORYFIKdxJvRbK1AR54G8g2tXKdQJ1InxczDool6PwJEeury01MoYUimTmsFlwQW169vjAoMq1OGWbglPML5nq5KXr+SydN1bR35QLGrOSMVUroQsJv31RrunggN221jrLyHRkm7ZUUMYtUsVnsrrlmZ0gs+150chy2t185En8CAx5F38sxYVD8ycr/R4wjxt3i9NJQoayHS52pootU1wBenbLXMeck/dKIaDbpFrRbD12HjXPums121rAXH7DYrcFlUO/1kg5+h+pP9EQBvQbnd3B8m30mo82fHrcFcsvAdrvYbcWd7dbNS2wVWSh/oaPeLSZp/L4ZMk4Xzhq7h2j1+HL9v29HNOuayXsXqRtH6JGSGXGUGmjFd00r15h2Zv5mpvvvFKCQwvJ00Jw6r1kdkKR1lUSofDFvcg01B8rC+PszczTQppg9RvWring3G82eal7T98QX4z4i5KtGD6+Ryw12Bq2Tkbj+172KERO6bhn1LAsqiTSFkbznHC9q0X9RG5X+cnHuRZpMkY5Ld42V2S3r0RugLme203jnoc3PYtAdbH6d8fHHyEA8tx2b6B2Ygvt7JT7TteD6/bsenYP0U9h0fowxd7S+NP7QA/7K/qORCwFSgQ7RgTtHnjZGoem1XDbPyxM9PPLQSG8/n/fEIiBuZS82vz18P9BSRexXvUCGXGxvzH+6uQeis74/1qn5K7ZtlHdb5a5Pt+1PDFJXvUTzeRbDybn62mgHvJVUtxfXp1QwFdhA+chWibYmF6kEHfnXw1/ytEUF812S2vuxK3DAB8wTqjFS4Z9E9F8P0UmpdLkX+KXRGULHr2WYczRdxuktPmf4JuYtXct10q6k6/5K5Z7c1a7066rAdwauN4XpM+Xjkt12uhRbQgEailwUwA6AK2TqWpZv8f9E2u1d6Gdv0PWF+0FDb83TRmjHcF0LAX01XyX+KkORGrSPIdU1QaBxP31D6Nx4v9rWglFa+203XJYQ02aiVaxTetf22CP2mST7kvg+RCD02nAnyZi9X0EE+zpG/b0WRYX0Kc6c4MI5WH1chfAJkb+VMPYyXtgK78uJHc7Sa8bqfboeAqtpdpbMcM8To+vsrUkct0Fand8PE+oiSKO/MKies/Soe0orD+JDZ8TtVNRxooetSA6nDvRwu6QDrNd12OhD3qy1JQhW5wqngv9RE9kZ1dg6d13KBCbef10B0VoGvdUtdInJmZ5HkZ5gWmQLCP1vY2bCJUN0gqFxaS6Vm3sXyJPsoTQ1BX8NUCv47d3dgBd+OpZHJUL02FWMN77YUN08hD9JdTJp+N464M6b2ke1j21PmGduQeUjTGPoraKwZpfGxew4H1cuGv8achBMSbMuHmRGxfngZHBK3NPCL5tha/pw9zux5aSz/hzke646BRTxbH6KoeOCyQAN2ULRrgU59G1XhTgAPGkARHEC6NQXvtb69jJdtIrxK0zeD/rUyJpDxKVhU1I4hmowhBS7t3YhpEx0gVx+jKclUoGCap+uNEPtjQOCiPg4GR6Bx42pIXwhsLY058k0tWN9WD78MiEPKgt7uqiwnMsDOD3NL0B79H6GQcWymN7MJ4c0c1y7b3045cf0c/cAnAvCmL1K80vc+Ar1CRwi+nzwWqFS5nABSe1WzabHq/enhNXH0A0+867/1XsHopf8dtk8PmDDUcX0u1WL123u4w4hnnNpGmPyC/1uTPVY015e3oj1keJVrUp+BEJ27y23pSGWb15+cvsr4f+WMJjkdPGWXBBo0xM1KGQ1sAyx1J7IXwkcQ6gal5SL6hQZ5DX94jB5+KmyX93YvGwngNCIAL5Lp9v6GvttDDXns6Gf4xbcb2T2XdoJCHUJ88r+czEnUmaL5/twCp6YvorBxK0CSF8am77s8804j1/q8OWANhdF47VGEAyUvFqGAsJTVJ3Kszzd26KsZdP4cmCEJtXlRCWmWrdT450tgFyJnW7gS4Z2H8HNRkoFeStupNgkEcZyTQuek2k7qNZ/blw4iSkYqETIobSuVud009Fu3xKs9HMh2agRHajTmtCmrYIZKRC4kL1WUCJRLEENaw/WQPNomwpeGxj/Nwu9McrfXLMo2nVtVh0ND6BhcoNL96R1P6Xg106Vxin17kwp805cnxxk1xbiHcZvTtoH1sAv3Qec+GLcw3YhGwV6Hn9GYPLTBewje2tJ5CEnH6vftCFH01qWmLQoKXkKXatPlrJnsSTH11U12RqQYv5H1S5qqiK04u4YrxxvSDsszsq3U0jYve7P7LmASDp16NUHyofofotdnK92tEhOCRCHSAvFBEGeQZBrbj5F34L2mU5i4zC467P9DLVKaqaohFb1IXgouhXh+Lbpx3aHXK4uheuRvR3s3VDYpARH5b47DyGPSS+5t173sSDPdxXXW1ELow2zHOCWE0ke3Axf4fJ7cepqISCMQIu13UwLwBZb3X9HYjkuW1BAX6RCM3ncYcmStIfiA7+hkpSc+C4G5FSAM0MqrobZ1YGL9XQrjYpC7dSSkY0rZmTspE2mQPhQAjmsWKVE9D001MVkyx+JnBotUrtUJK5tQb72ys2Ynvt2p+aaqUD1zm4vBSpgFgZ4LamJATU51C2J/v4Y96PK17HmnktXU2vWS5MxAdVyoKp0DIXRGcQjQX0X4qap6D/Sz18dQKGuB18DxNvswjJfHnAK7pQHduXUc2ZCcCdenbvvQ3XrreBdV5u54YNUZxf7ubKkWhpuUk0sy4gIRUD6GNRO0Kcm9PSQ/1/+H+H10W0zogsLwB+Bu/FK1912suc26mFa7Yj1CHL2laE1A828pewk6QoiqVpNNi80KwMnNebauL6KjcpzbJhe5JOMMRHFSUteUukJY61y9z8GpR/X+RzeziFULj0W73Eqs7zUf56d25ARkg414Wmxm05YF72qzq6v5eQ9ZrRrilt3KURWPMq+nQL4uo2hlZQZTrs6MAHGt32G1wbOR+21zrQ7oz+Xbw+GjkkdA5I28MU0mvk9Dco0dJ209K5pWoz9cCB10nvzIb2ud1+nqT06S2A40f3BacPaW/+x70pYvtxNL8SwoHw24tHvBSK9J3P0eKNh73bte/AQNgaaj0jSUG6v5Loo16LCCXDpUJlBNKXYKXsKpud30ibkL7H4OKY7rSKc4z2m3cWomw+Q1h12FEET1dBBEN49/uROciP/dRmJYfRexowI4+dEtMc4d2qkyIZYH1Y++gvNb131VC016ovD9K60Xchr73ybtX0tCaeRcPyQwEPvTsz7zUzSmdNJAuYKX5xerV4u8yoeCE/VFwVZhnP1WuGyX4+ogn1fcbTGNB6ZJUHmVfGp7ZEAs0mU7C7sujsjly1w/ypK8DOKIwY5pfwplF/AnV1p7PDOA9tWxAYmaJhEPREJZTM6I8krvgNg223TBm8Cd8SG3P3T/jRU5aH3+vyp8EjmQ6x2F9u3uvBZ+JY0OWDIPdkX71t1NRY00e+cPaXhtyYPbY4s61fHN5PKP+UcT0f5y9TqmXvW7J4V7aVaAs332q715ZxpXo1BbRskhr3a1i42/1QPr0nCdC/otK+BnnFNs7jWcQJ0Q6CEU/OongpPTJ0kbtFS7LdX4747b/g1GZBG5c0etU2Yz6FEATrTwrubhKfHme4Df3Ygox0K/adod1fBNq0Vutcs7rATFwC1W5SEhJuR+327jvtvtp1te542t16QFinatFuvt+Jps16KVJ503Kjpf1qnm0UmESOXgKXSSfR03uicr1S9pfm8kKwMSWcalCRXYFO1C600SkObgSfpxkGpT0KDrZd3pY3N362HrexadU9F9KJngUJ06XN18VH8Twa7C1ORZgNWUfslkZ9CN0S631IF2z2wuHX8dJz3UXelwQFio+ZMrAn7RyJjTr2VIHb9Ra/LHSi5CuATdJfmfj7r6dSgQYpu6Htb0I1+amHoijf4UMDEWHLc8va61kSc8GrTZYGKvkvT181842dp2QPTN9u612ozNFG3r1h127/eq64bkl1rur95JqEHDHYZguagELUATJ0k4CSfrgz+mwMk5a6z2G56CChy1/Cr1L/NmY159mRKHU/VYUHEnc3WwY3bPAPGqFFBgmX1/vRrnsY5w95fr/zVn323q3W44dc/Em+jepk3+1CmX5C0j81SKYkzgr/y051d4qJYyww0tUouk//URFkjuu8mOyM5JbcWf1C1IPk84+0s6J88tXDCyJELx28YeuJka1TVJ9129RhsMakfwUalT0ievmuhsd+EvYViDLr4Dh1yqBkYj9CXmOjGHJ4hgXPbC3+Vuarh071Xn1B67XI3jntJnpl1XRLPqLVbc8ae59E1eA6+g0b98SZ9R2FbKAX9R20A1DtTjdvaoAXAFOBjTLzUs0/DM/koJD3RYDX+g+grk6sLwzwl/dWq7NMytS2vRhp27Fwn15H6k0Uf5efhYfH686BT+mqY+gV0OX7Y23Bpn2M4BCJ0B8//9/91VE2Mcu





❖ Afficher des images dans l'application Web

◆ Vue Razor

```
public class ImageViewModel
{
    public Image Image { get; set; } = null!;
    public string? ImageUrl { get; set; }
}
```

```
@model IEnumerable<Sem12.ViewModels.ImageViewModel>
```

On peut auto-générer une vue avec le template **List** pour notre Model. Les changements les plus importants à faire sont :

- Utiliser le **ViewModel** qu'on a créé pour recevoir les strings qui représentent les images.
- Afficher l'image dans un élément **** tel que dans l'exemple. Ce n'est pas très complexe : on intègre le string dans l'attribut **src**.

```
<table class="table">
  <thead>
    <tr>
      <th>
        @Html.DisplayNameFor(model => model.Image.Nom)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.Image.Identifiant)
      </th>
      <th>
        Image
      </th>
    </tr>
  </thead>
  <tbody>
    @foreach (var item in Model) {
      <tr>
        <td>
          @Html.DisplayFor(modelItem => item.Image.Nom)
        </td>
        <td>
          @Html.DisplayFor(modelItem => item.Image.Identifiant)
        </td>
        <td>
          
        </td>
        <td>
          <a asp-action="Edit" asp-route-id="@item.Image.Identifiant">Edit</a> |
          <a asp-action="Details" asp-route-id="@item.Image.Identifiant">Details</a> |
          <a asp-action="Delete" asp-route-id="@item.Image.Identifiant">Delete</a>
        </td>
      </tr>
    }
  </tbody>
</table>
```




- N'hésitez pas à redimensionner les images avec du CSS.

- Dans le cours Prog Web Services (4W6), redimensionner des images sur le serveur pour alléger les fichiers à envoyer au client est déjà abordé, alors nous ne le ferons pas à nouveau dans ce cours.

Sem12 Home Privacy

Index

[Create New](#)

Nom	Identifiant	Image	
Birb	7be572bf-5253-4b37-855b-2f0392c7d730		Edit Details Delete



❖ Supprimer les images

- ◆ Vous remarquerez peut-être qu'après avoir supprimé une rangée qui contient un fichier avec **FILESTREAM** dans la BD, le fichier ne disparaît pas du File Group... Est-ce un fichier zombie qui traînera pour toujours ?! 🧟🤖
- Pas de panique 😓 : **FILESTREAM** vient avec un système de *Garbage Collection* discret qui supprime les fichiers jugés obsolètes après un certain nombre de transactions.
 - Gardez à l'esprit qu'il ne faut pas compter sur le fait que les fichiers restent, mais il ne faut pas non plus espérer qu'ils soient supprimés instantanément.
- Si vous souhaitez expédier le *Garbage Collector* pour voir qu'il supprime bien les images obsolètes, vous pouvez utiliser ce code SQL sur votre BD :

```
CHECKPOINT  
GO
```

```
EXEC sp_filestream_force_garbage_collection 'Sem12'  
GO
```

- Remplacez **Sem12** par le nom de votre BD, entre apostrophes.



❖ Monitorer la performance

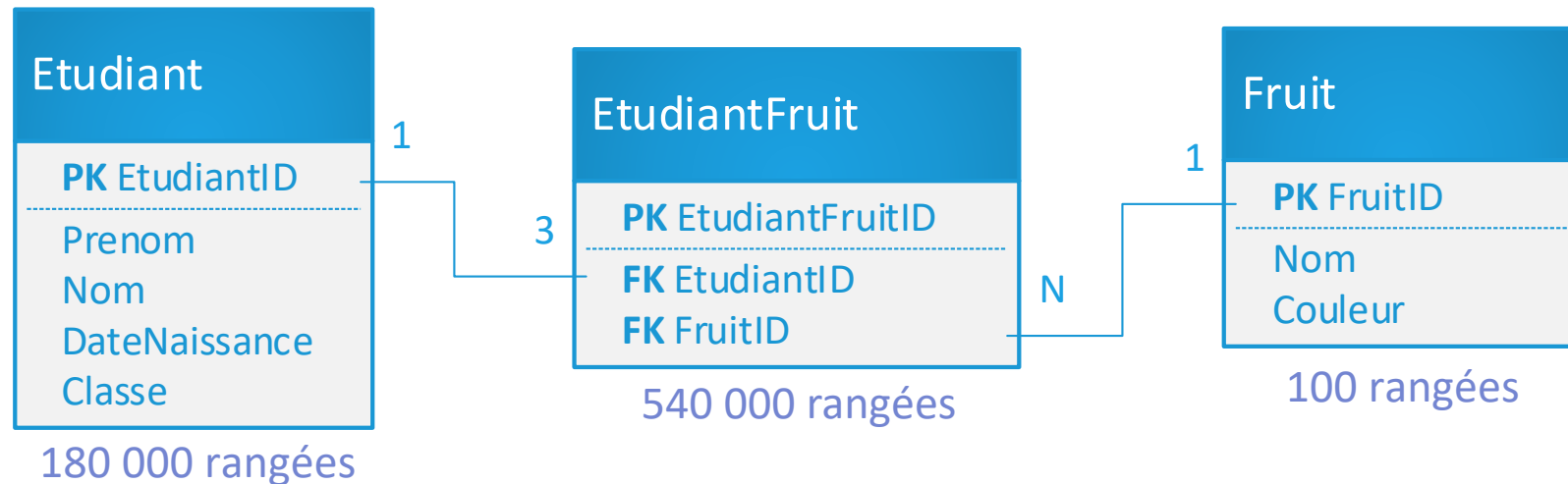
- ◆ Avant d'aborder des stratégies pour améliorer la performance d'une BD, il faut bien entendu pouvoir la monitorer.
- ◆ Toute manipulation visant principalement à améliorer la performance ...
 - Doit être testée.
 - Et si possible avec une grande quantité de données ! L'impact d'un changement peut réserver des surprises à grande échelle.
 - Doit présenter des avantages plus significatifs que ses désavantages.
 - Pour vraiment pouvoir conduire des tests intéressants sur la performance, avoir des gros Datasets est préférable. La variété et la richesse des tests que nous pourrons faire dans ce cours seront donc limitées.
 - Cela reste intéressant d'aborder des stratégies d'optimisation, ne serait-ce que théoriquement.



❖ Stratégies d'optimisation

◆ Dataset utilisé

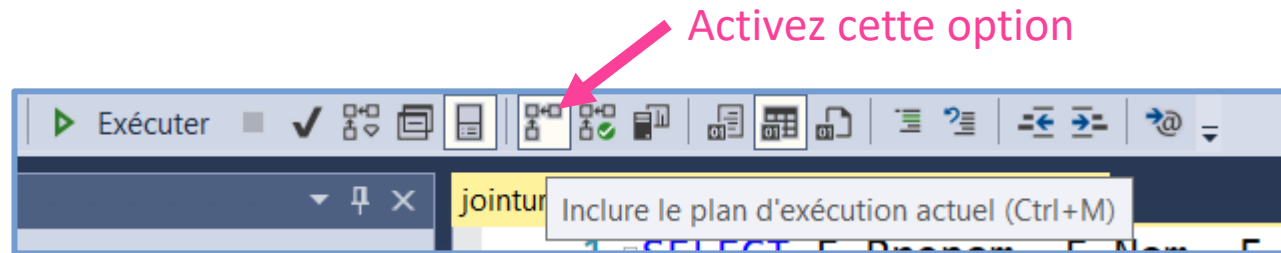
- Dans une école, on a demandé à 180 000 jeunes quels étaient leurs trois fruits préférés parmi une liste de 100 fruits.





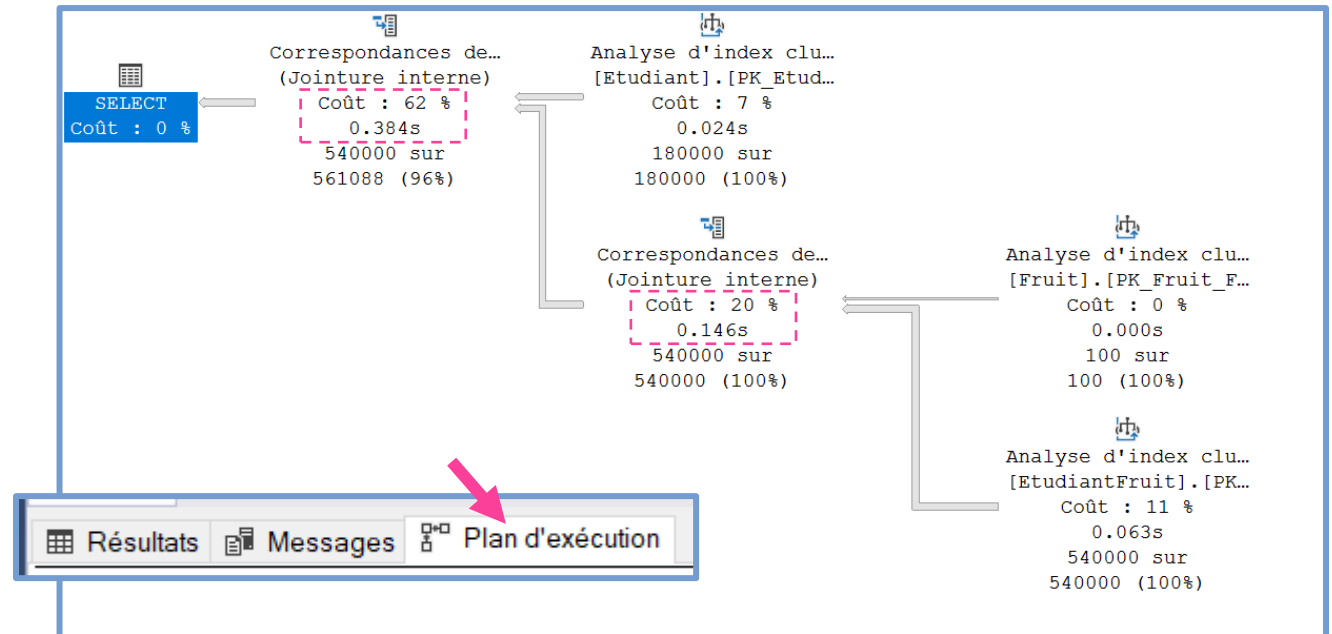
❖ Monitorer la performance

- ◆ Il existe plusieurs façons de monitorer la performance d'opérations SQL. Nous utiliserons le plan d'exécution proposé par SSMS.



```
1 SELECT E.Prenom, E.Nom, F.Nom, F.Couleur
2 FROM Fruits.EtudiantFruit EF
3 INNER JOIN Fruits.Fruit F
4 ON EF.FruitID = F.FruitID
5 INNER JOIN Etudiants.Etudiant E
6 ON E.EtudiantID = EF.EtudiantID;
```

▶ Exécuter



Plans d'exécution : En Graphique et en Tableau

- Le plan d'exécution graphique est facile à lire, mais il est difficile de partager tous ces détails.
- La commande : **SET STATISTICS PROFILE ON**
 - ☐ permet de réaliser un plan d'exécution complet à chaque requête (en tableau).
 - ☐ Pour arrêter son effet, il faut utiliser la commande :

SET STATISTICS PROFILE OFF

```
SQLQuery2.sql - BU...ISON\Bernard (53)) SQLQuer
SET STATISTICS PROFILE ON
SELECT * FROM Client
```

SELECT * FROM CLIENT



*Plan d'exécution en **tableau***



Résultats

Messages

Plan d'exécution

	NumClient	Nom	Prenom	Adresse	Ville	Province	CodePostal	Solde	LimiteCredit	NumRepresentant
1	124	Moulineau	Paul	368, rue du Campanile	Québec	QC	G1X 4G6	818.75	1000.00	3
2	256	Allard	Martine	996, St-Michel	Montréal	QC	H1H 5G7	21.50	1500.00	12
3	311	Boucher	Camille	540, Bd des Galeries	Québec	QC	G2K 1N4	825.75	1000.00	3

Rows	Executes	StmtText	StmtId	NodeId	Parent	PhysicalOp	LogicalOp	Argument	DefinedValues	EstimateRows	EstimateIO	EstimateCPU	
1	14	1	SELECT * FROM Client	1	1	0	NULL	NULL	NULL	14	NULL	NULL	
2	14	1	I-Clustered Index Scan(OBJECT:([BdCommercial...	1	2	1	Clustered Index Scan	Clustered Index Scan	OBJECT:([BdCommerciale].[dbo].[Client].[PK_Clie...	[BdCommerciale].[dbo].[Client].[NumClient], [BdCo...	14	0.003125	0.0001724



❖ Stratégies et concepts liés à l'optimisation

◆ Nous allons aborder quelques stratégies simples, mais pour la majorité nous n'en parlerons qu'en surface, pour que vous sachiez que ça existe.

- **Index**
- Partitions
- Cache
- Autres éléments de configuration



❖ Stratégies d'optimisation

◆ Index

- Les index représentent une stratégie pour ordonner les rangées dans une table et ainsi accélérer la recherche de rangées spécifiques.
- Pourquoi ordonner les rangées accélère la recherche ?
 - Car si on peut faire confiance à l'ordre, on peut utiliser un **algorithme de recherche dichotomique**.
- Les deux diapositives qui suivent vous montre la différence dans la recherche d'une information avec des données non ordonnées VS la recherche d'une information avec des données ordonnées.



❖ Stratégies d'optimisation

◆ Index

- Exemple **sans ordre** : On cherche une rangée avec ID = 14

Pas le choix, **14** pourrait être n'importe où, on vérifie toutes les rangées !



1
9
3
15
12
5
7
16
10
13
17
4
11
14
6
2
8

Trouvé! 14 vérifications ont été nécessaires



❖ Stratégies d'optimisation

◆ Index

- Exemple **avec ordre** : On cherche une rangée avec **ID = 14**

Les données étant ordonnées, on peut utiliser un algorithme de recherche dichotomique:

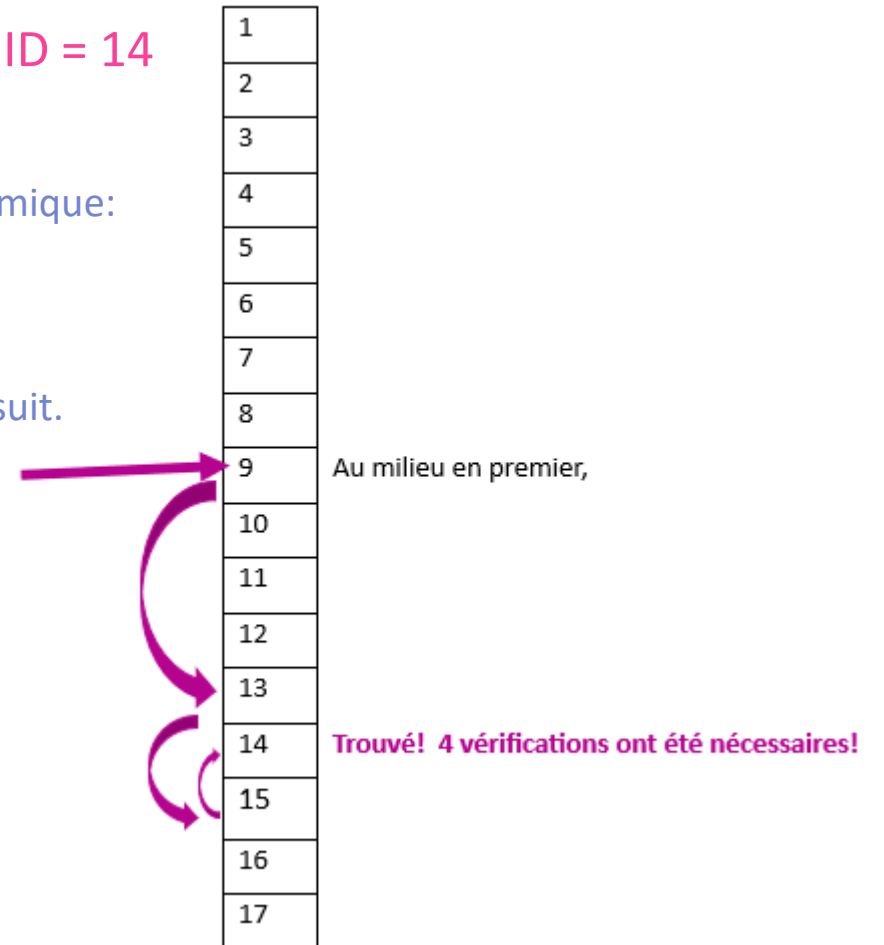
- On vérifie toujours la rangée au « **milieu** » de l'ensemble restant.
- On regarde si la valeur trouvée est plus petite que la valeur recherchée?
- Si oui, on élimine ce qui est avant et on continue la recherche dans le restant qui suit.

9 est **trop petit** ? on élimine la moitié des rangées.

13 est **trop petit** ? on élimine un quart des rangées.

15 est **trop grand** ? On élimine un huitième des rangées.

Finalement, on trouve **14**.





❖ Stratégies d'optimisation

◆ Index

- Un index est un objet du server SQL qui va lui permettre de voir les données d'une table selon les valeurs des champs qu'on va avoir spécifié lors de la création de l'index.
- SQL Server pourra alors utiliser l'algorithme de recherche dichotomique pour trouver rapidement les données recherchées.
- Nous n'avons pas besoin de spécifier à SQL Server quel index qu'il doit utiliser. Il se basera sur les champs qui sont dans les clauses WHERE, GROUP BY et ORDER BY des requêtes pour utiliser le meilleur des index existants sur une table.
- Nous devons par contre **créer les index** pour que le serveur puisse les utiliser.



❖ Il existe 2 types d'index:

◆ Clustered OU Non-Clustered

○ **Clustered** (max. 1 par table)

- Détermine l'ordre des rangées d'une table, **physiquement, sur le disque**,.
- Créé automatiquement par SQL Server lors de la création de la contrainte de clé primaire sur une table
 - Ex : En créant la clé primaire sur le champ FruitID de la table Fruit, SQL Server a créé automatiquement un index clustered sur le champ FruitID. Dans ce cas, dans la mémoire du système, les rangées de la table seront rangées dans l'ordre ci-dessous.

○ **Non-Clustered**

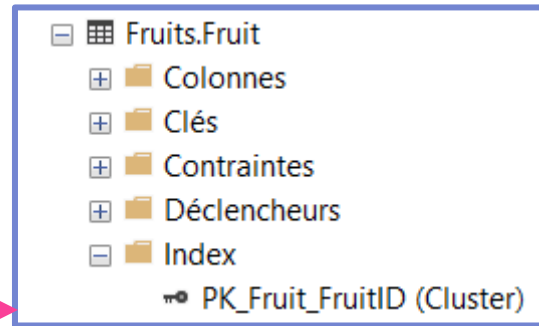
- Crée une **nouvelle structure** avec toutes les **valeurs** de la **colonne choisie**, chacune accompagnée d'un **pointeur** pour nous aider à retrouver la valeur plus rapidement dans la table.
- **Doit être créé par vous.**



❖ Index Clustered

- ◆ **Clustered** (max. 1 par table)

Quand on crée une contrainte PK, un index clustered est créé automatiquement, par défaut !



Impact sur la performance

- Si je cherche une rangée par son **FruitID**, la recherche sera **nettement accélérée**.
- Si je cherche une rangée par le **nom du fruit**, l'index n'aide pas du tout.

FruitID	Nom	Couleur
1	ananas	marron
2	avocat	vert
3	baie de goji	rouge
4	banane	jaune
5	bergamote	vert
6	carambole	jaune
7	cassis	noir
8	cerise	rouge
9	citron	jaune
10	canneberge	rouge



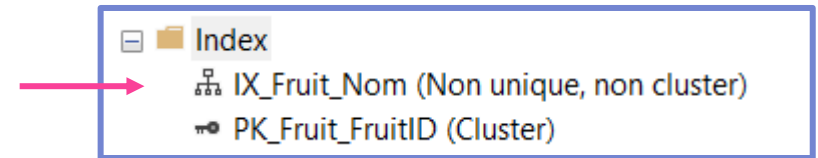
❖ Index

◆ Index Non-Clustered

- Exemple : **Index non clustered** pour la colonne **couleur**

```
CREATE NONCLUSTERED INDEX IX_Fruit_Nom ON Fruits.Fruit(Nom);
```

IX_Table_Colonne (nom) Schéma.Table(Colonne)



Impact sur la performance

- Si on cherche un fruit par sa **couleur**, c'est beaucoup plus rapide si on a un index. (On trouve la **couleur** dans la **structure de l'index**, qui est **ordonnée alphabétiquement**, PUIS on utilise le ou les **pointeurs** pour trouver la ou les données qui nous intéressent dans la table.

Structure de l'index

Couleur	Pointeur
jaune	
jaune	
jaune	
marron	
noir	
rouge	
rouge	
rouge	
vert	
vert	

Table

FruitID	Nom	Couleur
1	ananas	marron
2	avocat	vert
3	baie de goji	rouge
4	banane	jaune
5	bergamote	vert
6	carambole	jaune
7	cassis	noir
8	cerise	rouge
9	citron	jaune
10	canneberge	rouge



❖ Stratégies d'optimisation

◆ Index : quelle(s) colonne(s) choisir ?

○ Idéalement ...

- Colonne avec des **nombre**s entiers. (Données simples)
- Colonne avec des **valeurs** uniques ou très diversifiées. (Pire cas : toutes les rangées ont la même valeur pour la colonne -> l'index ne sert à rien)
- Colonne souvent utilisée pour **recupérer** les données (WHERE)
- Colonne souvent utilisée pour **regrouper** les données (GROUP BY)
- Colonne souvent utilisée pour **ordonner** les données (ORDER BY)



❖ Stratégies d'optimisation

◆ Index : quelle(s) colonne(s) choisir ?

○ Rappel

- Un index **clustered**, au maximum. (Impossible d'ordonner physiquement les rangées de plusieurs manières...). On ne le crée pas nous même.
- Autant d'index **non-clustered** que nécessaire, mais si possible, seulement pour les colonnes souvent utilisées dans les clauses WHERE, GROUP BY et ORDER BY.



❖ Index avec des champs de tri ASC ou DESC

- ◆ Par défaut, les tris qu'on fait avec ORDER BY sur des champs sont ASC, donc en ordre croissant.

- ◆ C'est la même chose pour les index. Ces 2 commandes sont équivalentes:

```
CREATE NONCLUSTERED INDEX IX_Fruit_Couleur ON Fruits.Fruit(Couleur)
```

```
CREATE NONCLUSTERED INDEX IX_Fruit_Couleur ON Fruits.Fruit(Couleur ASC)
```

- ◆ Avec un tri descendant:

```
CREATE NONCLUSTERED INDEX IX_Fruit_DateAchat ON Fruits.Fruit(DateAchat DESC)
```

- Ce qui accélérera la recherche quand on voudra voir les fruits achetés le plus récemment en premier.

```
CREATE NONCLUSTERED INDEX IX_Fruit_Prix ON Fruits.Fruit(Prix DESC)
```

- Ce qui accélérera la recherche quand on voudra voir les fruits les plus chers en premier.



❖ Index sur plusieurs colonnes

- ◆ Si je veux voir les villes par province, faire UN index sur la province, suivi de la ville va accélérer les recherches:

```
CREATE NONCLUSTERED INDEX IX_Etudiant_Province_Ville ON Etudiants.Etudiant(Province, Ville)
```

- ◆ Si je veux voir les employés par leur nom de famille et par leur prénom, faire UN index sur le nom de famille, suivi du prénom va accélérer les recherches:

```
CREATE NONCLUSTERED INDEX IX_Employe_Nom_Prenom ON HR.Employe(Nom, Prenom)
```

ATTENTION: On verra très rarement un index sur plus de 2 champs, car plus l'index est court, plus il est performant.



❖ Stratégies d'optimisation

◆ Index : **considérations** et **coûts**

- **Constamment réordonner les rangées** : lors d'un INSERT , UPDATE ou DELETE, que ce soit dans la table (**clustered**) ou la structure de l'index (**non-clustered**), on doit maintenir **l'ordre de l'index**. C'est entre autres pour cela que les colonnes **auto-incrémentées** sont agréables pour les **index** : il n'y a **pas de réorganisation** à faire, on fait juste ajouter de nouvelles rangées **à la fin**.
- **Plus d'espace occupé** : Comme chaque index **non-clustered** ajoute une structure supplémentaire dans la BD, il peut être intéressant d'en limiter la quantité. Souvent on va créer un index, regarder la performance et décider ensuite si garde l'index ou non selon la grandeur de l'amélioration obtenue.



❖ Stratégies d'optimisation

◆ Index : **considérations** et **coûts**

- Table avec beaucoup de **WRITE**, peu de **READ** : index **pas très intéressant**. On est constamment en train d'entretenir les index, mais rarement en train d'en profiter.

Ex: Table de transactions boursières

- Table avec peu de **WRITE**, beaucoup de **READ** : index **très intéressant**. On profite constamment de la recherche accélérée et on a peu d'entretien à faire.
 - Ex : Table d'archives sur un groupe du passé, comme les Beatles. Il n'y a pas trop de modifications à faire...
 - Ex : Table d'utilisateurs (Chaque personne se crée un compte qu'une seule fois et modifie rarement les données de son profil, mais ces données sont souvent utilisées pour l'authentification, les paiements, l'affichage d'éléments du profil, etc.)



❖ Index : exemple d'accélération

EtudiantFruit	
PK EtudiantFruitID	
FK EtudiantID	
FK FruitID	

540 000 rangées


(On garde à l'esprit que 540 000 rangées c'est très peu)


```
SELECT * FROM Fruits.EtudiantFruit
WHERE EtudiantID = 150000;
```



EtudiantFruitID	EtudiantID	FruitID
83998	150000	56
83999	150000	58
84000	150000	41


Sans index sur EtudiantID: 19 millisecondes



SELECT
Coût : 0 %



Analyse d'index clu...
[EtudiantFruit].[PK...]
Coût : 100 %
0.019s
3 sur
4 (75%)


```
CREATE NONCLUSTERED INDEX IX_EtudiantFruit_EtudiantID
ON Fruits.EtudiantFruit(EtudiantID);
```

Avec index sur EtudiantID : instantané (négligeable)


SELECT
Coût : 0 %


Boucles imbriquées
(Jointure interne)
Coût : 0 %
0.000s
3 sur
3 (100%)


Recherche d'index (...
[EtudiantFruit].[IX...]
Coût : 32 %
0.000s
3 sur
3 (100%)


Recherche de clés (...
[EtudiantFruit].[PK...]
Coût : 68 %
0.000s
3 sur
3 (100%)



❖ Stratégies d'optimisation

◆ Partitions

- Méthode qui consiste à **séparer la table** en plusieurs **sous-tables** selon un critère au choix. (Ex : Chaque **catégorie de produits** est dans sa propre **partition**)
- Chaque partition peut avoir ses propres **index**.
- Chaque partition peut même être sauvegardée dans une BD ou un serveur différent.
- Les requêtes qui exploitent bien les valeurs partitionnées deviennent **plus performantes**.
 - Ex : Je fais une requête qui s'intéresse seulement aux produits de type « **Automobile** » : seule la partition pour les produits de type **automobile** sera fouillée !
- Les requêtes qui utilisent des colonnes qui n'ont pas été utilisées pour faire la **partition** seront généralement **moins performantes**.
 - Il faut fouiller dans plusieurs partitions, faire les opérations dans **chaque** partition, séparément, puis **fusionner** le résultat.



❖ Stratégies d'optimisation

◆ Cache

- Généralement, sert à stocker des données fréquemment utilisées pour limiter la quantité d'accès au disque pour améliorer la performance.
- SQL Server : Utilise deux caches
 - **Buffer cache** : contient des « pages » de données fréquemment utilisées. Lorsqu'il est plein, les données plus vieilles ou moins utilisées sont retirées pour faire de la place.
 - **Execution plan cache** : lorsqu'une requête est faite, un « plan d'exécution » est préparé et puis la requête peut ensuite être exécutée. Les plans d'exécution sont stockés dans ce cache pour ne pas avoir à les recréer si la même requête est utilisée.
- On peut **changer la taille** du cache !
 - Cache trop **souvent nettoyé** : Il est trop petit et n'aide pas vraiment à améliorer les performances.
 - Cache trop **rarement nettoyé** : Il est trop grand et un cache plus petit est généralement plus optimisé.
- Un indice intéressant pour juger la taille du cache est la « **Page Life Expantancy** », (**PLE**) c'est-à-dire la durée du passage d'une page dans le buffer cache. Il n'y a pas de mesure parfaite, mais sous 300 secondes, il est préférable d'augmenter la taille du cache.

Requête pour obtenir la PLE

```
SELECT*  
FROM sys.dm_os_performance_counters  
WHERE [counter_name] = 'Page life expectancy'
```



cntr_value → PLE en secondes

object_name	counter_name	instance_name	cntr_value	cntr_type
MSSQL\$SQLEXPRESS:Buffer Manager	Page life expectancy		52854	65792
MSSQL\$SQLEXPRESS:Buffer Node	Page life expectancy	000	52854	65792



❖ Stratégies d'optimisation

◆ Autres

- **Allocation de mémoire (*Scaling vertical*)** : Si le serveur est uniquement consacré à opérer la BD, s'assurer qu'autant de mémoire que possible est alloué à la BD SQL Server.
- **Séparation des fichiers (*Scaling horizontal*)** : Utiliser plusieurs machines et partitionner les données permet de diviser les tâches entre plusieurs serveurs.
- **Fréquence et moment de backup** : Les backups affectent beaucoup la performance. Selon l'importance des données, en faire moins souvent (si possible) ou de les faire à des moments de faible achalandage. N'oubliez pas qu'il existe **plusieurs types** de backups. (Complet, différentiel, incrémental, etc.)
- **Utiliser une BD NoSQL ?** : Si l'application utilise beaucoup de données non structurées, (fichiers variés, photos, vidéos, etc.) une base de données **NoSQL** est généralement mieux adaptée. (Ex : Pour un réseau social avec beaucoup de contenu généré par les utilisateurs) Ne choisissez pas un **SGBD relationnel** par défaut !