Plan d'Exécution Graphique pour des Requêtes SQL Simples

Cet article est paru initialement le 16 décembre 2008 sur le magazine en ligne Simple-Talk http://www.simple-talk.com/sql/performance/graphical-execution-plans-for-simple-sql-queries/

L'auteur de cet article est Grant Fritchey : il est DBA dans une grosse compagnie d'assurance américaine et il travaille sur SQL Server depuis la version 6.0 en 1995. Il a également travaillé sur Sybase, et a développé en VB, VB.Net, C#, et Java.

Son livre « SQL Server Execution Plans » est disponible chez Amazon http://www.amazon.com/Server-Execution-Plans-Grant-Fritchey/dp/1906434026/ref=sr_1_3?ie=UTF8&s=books&qid=1280842445&sr=8-3

Grant Fritchey est MVP SQL Server.

Cet article a pour objectif de vous permettre d'interpréter les plans d'exécution graphiques de base, c'est-à-dire ceux de requêtes simples de type **SELECT**, **UPDATE**, **INSERT** ou **DELETE,** avec peu de jointures et sans fonctions avancées. Pour cela, nous aborderons les points suivants :

- **Opérateurs** – présentés dans un précédent article, nous approfondirons
- **Jointures** – peut-on parler de système relationnel sans jointures entre les tables?
- **Clause WHERE** – en quoi vos filtres de données affectent les plans d'exécution ?
- **Les aggrégats** – comment le regroupement des données modifie les plan d'exécution
- **Insert, Update, Delete** – leurs plans d'exécution

Le Langage des Plans d'Exécution Graphique

D'une certaine façon, apprendre à lire un plan d'exécution graphique revient à apprendre un nouveau langage, à ceci près que ce langage est construit sur des icônes, et qu'il n'emploie qu'un nombre restreint de mots (les icônes). Dans le plan d'exécution, chaque icône représente un opérateur spécifique. Nous utiliserons donc indifféremment les termes « icônes » et « opérateurs » tout au long de cet article.

Nous avons déjà abordé les 2 opérateurs **Select** et **Table Scan** dans un article précédent. Mais nous disposons de 79 opérateurs. Heureusement pour nous, il ne sera pas nécessaire de les connaître tous pour pouvoir lire un plan d'exécution. La plupart des requêtes n'utilisent que quelques icones, et ce sont celles-là que nous allons étudier dans cet article. Nous vous conseillons de consulter l'aide en ligne (BOL) s'il vous arrive de trouver une icône que nous n'aurions pas étudiée dans cet article.

http://msdn2.microsoft.com/en-us/library/ms175913.aspx

Un plan d'exécution graphique affiche quatre types d'opérateurs différents :

- **Les opérateurs logiques et physiques**, également appelés itérateurs, représentent l'exécution des requêtes ou les déclarations du Langage de Manipulation des Données (DML). Ils sont affichés en **bleu**.

- **Les opérateurs de parallélisme physique** représentent les opérations parallèles. Bien qu'il s'agisse d'un sous-ensemble des opérateurs physiques et logiques, on les représente séparément parce qu'ils impliquent un nouveau niveau d'analyse dans le plan d'exécution. Ils sont également affichés en **bleu**.
- **Les opérateurs de curseurs** représentent les opérations sur des curseurs de Transact-SQL. Ils sont affichés en **jaune**.
- **Les éléments du langage** représentent tous les éléments du langage Transct-SQL, tels que Assign, Declare, If, Select (en tant que résultat), While, etc. ils sont affichés en **vert**.

Dans cet article, nous étudierons surtout les opérateurs physiques et logiques, y compris les opérateurs de parallélisme. Dans l'aide en ligne (BOL), ils sont présentés par ordre alphabétique, ce qui ne nous semble pas être la meilleure façon pour un apprentissage. Laissons de côté l'alphabétiquement correct, et voyons ceux qui sont les plus utilisés. Bien évidemment, chaque DBA aura un point de vue différent sur cette question (lesquels sont les plus utilisés), mais il me semble que le tableau suivant présente les plus courants ; leur ordre de fréquence est à lire de gauche à droite et de haut en bas.

Select (Résultat)	**Trier**	**Clustered Index Seek**	**Clustered Index Scan**	**Non-clustered Index Scan**
Non-clustered Index Seek	**Table Scan**	**RID Lookup**	**Recherche de Clés**	**Hash Match**
Boucles Imbriquées	**Merge Join**	**Haut**	**Compute Scalar**	**Constant Scan**
Filtre	Lazy Spool	Spool	**Eager Spool**	Stream Aggregate
Distribute Streams	Repartition Streams	Gather Streams	Bitmap	Split

Dans cet article, nous aborderons ceux qui sont en gras. Les autres sont étudiés dans mon livre.

Les opérateurs ont un comportement qu'il est bon de connaître. Certains d'entre eux – essentiellement **trier**, **hash match** (aggrégation), et **hash join** – nécessitent une certaine quantité de mémoire. De ce fait, il se peut qu'une requête dans laquelle ils figurent doive attendre que de la mémoire soit disponible pour pouvoir s'exécuter, ce qui fera baisser les performances. Quasiment tous les opérateurs peuvent être bloquants ou non-bloquants. Un opérateur non-bloquant crée les données de sortie en même temps qu'il en reçoit en entrée. Un opérateur bloquant doit d'abord obtenir toutes ses données d'entrée avant de pouvoir produire ses données de sortie. Un opérateur bloquant peut contribuer aux problèmes d'accès concurrentiels, et dégrader les performances.

Quelques requêtes sur des tables seules

Commençons par quelques plans d'exécution très simples, basés sur des requêtes sur une seule table.

Clustered Index Scan

Prenons la requête suivante, simple (mais non-performante), sur la table **Person.Contact** de la base de données AdventureWorks :

```
SELECT *
FROM    Person.Contact
```

Elle suit le plan d'exécution suivant :

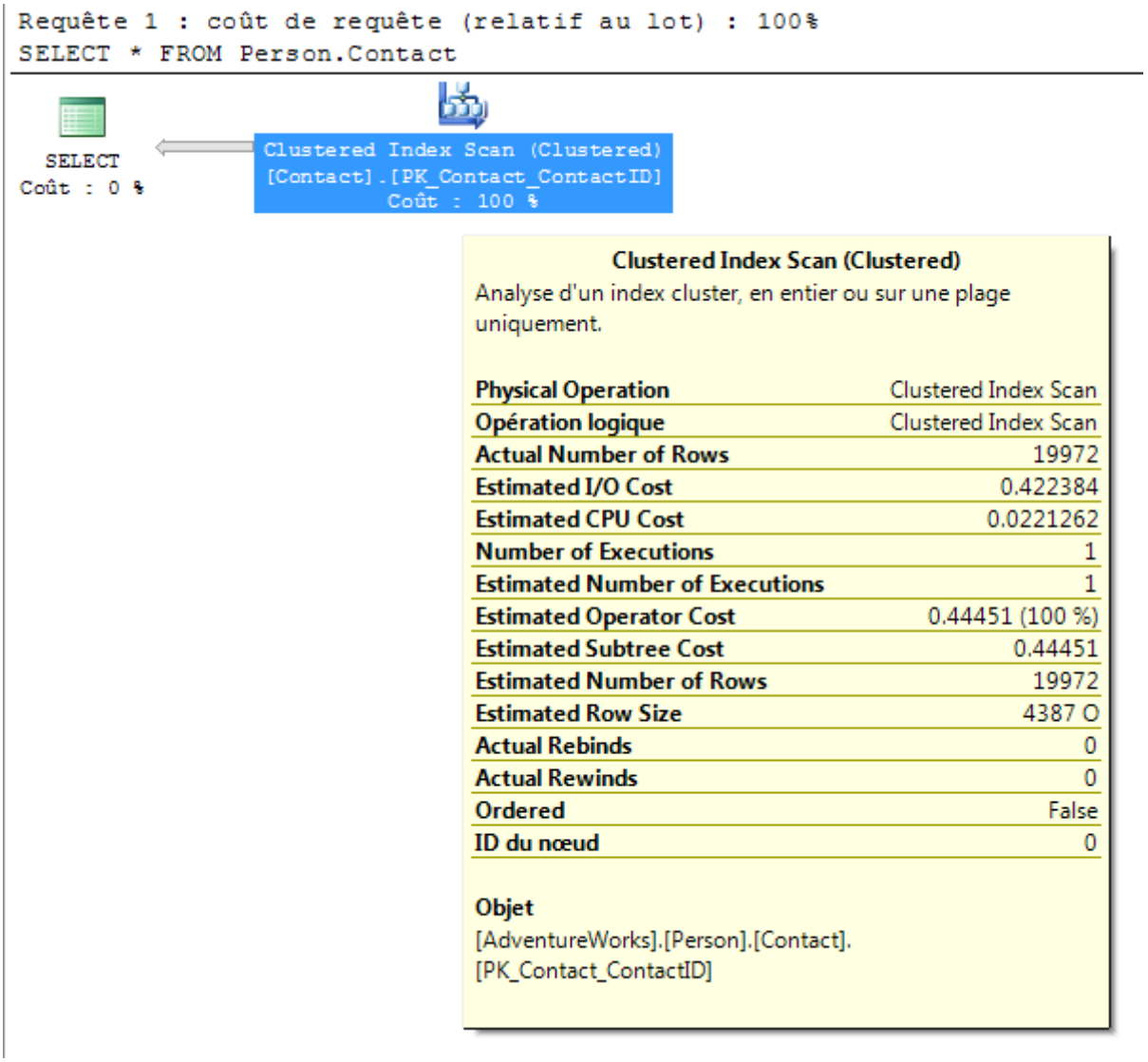


Figure 1

Nous observons que pour ramener les données demandées, une opération « Clustered Index Scan » est effectuée. En positionnant le pointeur de souris au dessus de l'icône, vous affichez une info-bulle dans laquelle vous pouvez voir que c'est l'index clustered **PK_Contact_ContactID** qui a été utilisé, et que le nombre de lignes de cette opération est estimé à 19972.

Dans SQL Server, les index sont stockés sous la forme de B-tree (une série de nœuds pointant sur un parent). À la différence d'un index régulier, un index clustered ne stocke pas seulement la structure de la clé, mais il ordonne et stocke les données ; c'est d'ailleurs la principale raison pour laquelle il ne peut y avoir qu'un seul index clustered par table.

De la sorte, un **scan** sur un index clustered revient conceptuellement à un scan sur la table. C'est l'index entier qui est parcouru (ou un pourcentage important), ligne par ligne, pour identifier les données requises.

Un index scan se produit le plus souvent quand, comme dans notre exemple, il existe un index mais que l'optimiseur calcule que le nombre de lignes à retourner est tellement important qu'il sera plus économique de parcourir toutes les valeurs dans l'index plutôt que d'utiliser les clés que contient cet index.

La première question à se poser quand on voit un index scan dans un plan d'exécution, c'est de vérifier si on ne ramène pas plis de lignes que nécessaire. Si le nombre de lignes est plus grand que ce que vous attendez, alors il est très probable qu'il faut affiner votre requête avec une clause **WHERE** pour ne ramener que les lignes nécessaires. Le retour de lignes inutiles consomme de la ressource et nuit à de bonnes performances du Serveur SQL.

Clustered Index Seek

Nous pouvons améliorer la qualité de la requête précédente, en lui ajoutant une clause **WHERE**.

```
SELECT *
FROM    Person.Contact
WHERE   ContactID = 1
```

Le plan d'exécution est maintenant celui que montre la figure 2 :

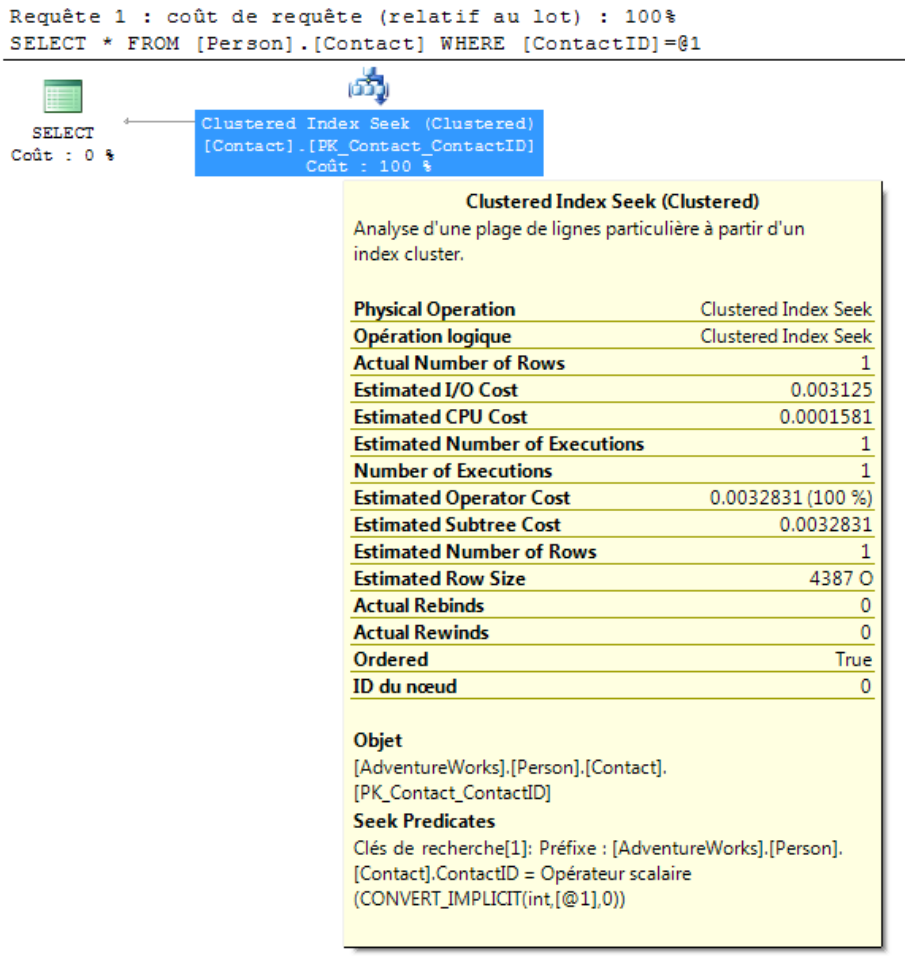


Figure 2

Un Index Seek est totalement différent d'un Index Scan, dans lequel le moteur parcourait les lignes pour trouver ce qu'il recherchait. Que l'index soit clustered ou pas, un Index Seek est déclenché si l'optimiseur peut disposer d'un index utilisable pour trouver les lignes voulues. Dans ce cas, il demande au moteur de stockage de chercher les valeurs dans les clés de l'index en question.

Dans une opération de Seek sur un index, les valeurs des clés sont utilisées pour identifier rapidement la ligne ou les lignes de données voulues. C'est identique à la recherche d'un mot dans l'index d'un livre, qui renverrait le numéro de la page où il se trouve. Un index clustered apporte en plus à cette opération d'Index Seek le fait qu'aucune étape supplémentaire n'est nécessaire : non seulement l'Index Seek est une opération non couteuse par rapport à un Scan, mais les données sont disponibles parce que stockées dans l'index clustered.

Dans l'exemple ci-dessus, une opération de **Clustered Index Seek** est exécutée sur la table Person.Contact, très exactement sur le **PK_Contact_ContactId** qui se trouve être à la fois la clé primaire et l'index clustered de cette table.

Remarquez bien dans la fenêtre d'Info-Bulle la mention Ordered à True, qui indique que les données ont été ordonnées par l'optimiseur.

Non-clustered Index Seek

Lançons maintenant une requête légèrement différente sur la table **Person.Contact** ; cette requête utilise maintenant un index non-clustered.

```
SELECT  ContactID
FROM    Person.Contac
WHERE   EmailAddress LIKE 'sab%'
```

Nous avons maintenant un non-clustered index seek. Remarquez dans l'info-bulle de la figure 3 que c'est l'index non-clutered **IX_Contact_EmailAdress** qui est utilisé.

Attention : l'icône est mal nommée dans le plan d'exécution, elle est présentée comme Index Seek. C'est une erreur de Microsoft, qui devrait être corrigée par la suite. C'est sans grande importance, mais faites-y attention.

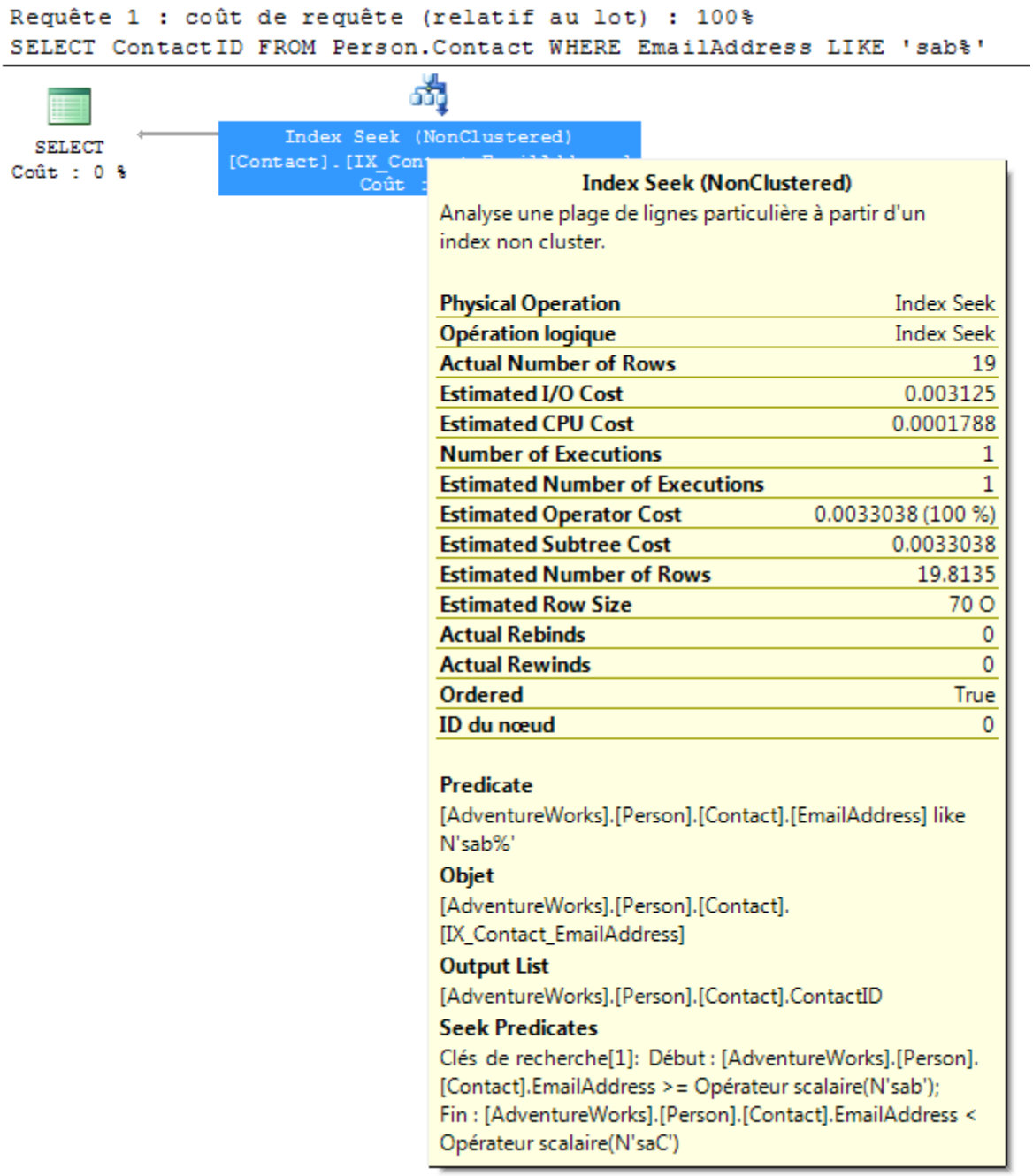


Figure 3

De la même façon qu'un Clustered Index Seek, un non-clustered Index Seek utilise un index pour trouver les lignes à renvoyer. Mais ici, c'est un index non-clustered qui est utilisé pour accomplir cette tâche. En fonction de la requête et de l'index, l'optimiseur pourra trouver toutes les données dans l'index non-clustered, ou bien il devra rechercher des données dans l'index clustered, ce qui affectera légèrement les performances, à cause des I/O supplémentaires nécessaires pour cette recherche (on en reparlera plus loin).

Recherche de Clés

Reprenons notre requête précédente, et modifions-la de façon à ce qu'elle nous retourne quelques colonnes de plus.

```
SELECT  ContactID,
        LastName
        Phone
FROM    Person.Contact
WHERE   EmailAddress LIKE 'sab%'
```

Vous devriez voir un plan d'exécution comme celui de la figure 4.

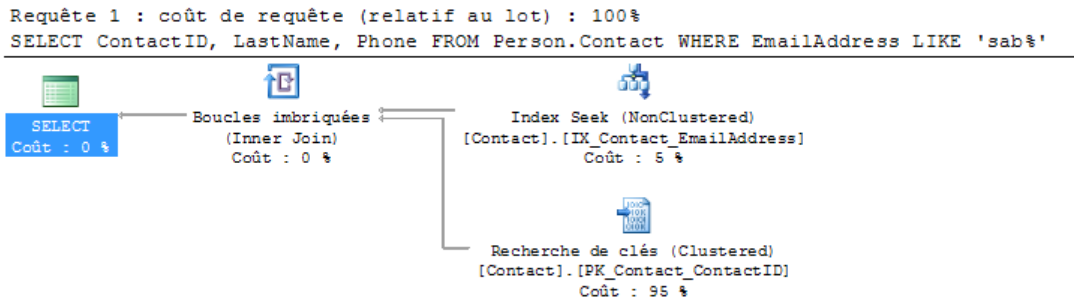


Figure 4

Enfin nous voyons un plan qui affiche plus qu'une seule et unique opération ! en lisant ce plan de droite à gauche et de haut en bas, la première opération que nous trouvons est un Index Seek sur l'index **IX_Contact_EmailAdress**. C'est un index non-clustered, non unique, et pour cette requête précise, il est *non couvrant*. Un index non couvrant est un index qui ne contient pas toutes les colonnes demandées en retour de la requête, et qui oblige donc l'optimiseur non seulement à lire cet index, mais aussi à lire l'index clustered pour y collecter les données qui sont demandées.

C'est ce que nous indique l'info-bulle de la figure 5 dans la section *Output List*, qui mentionne les colonnes **EmailAddress** et **ContactId**.

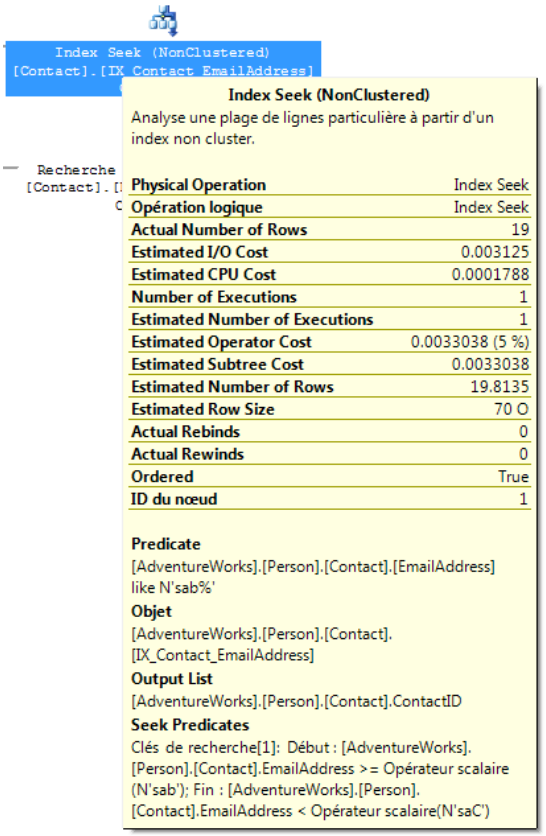


Figure 5

Les valeurs des clés sont maintenant utilisées dans une **Recherche de Clés** sur l'index clustered **PK_Contact_ContactId**, pour trouver les lignes correspondantes, en mettant en liste de sortie les colonnes **LastName** et **Phone**, comme on le voit sur la figure 6.

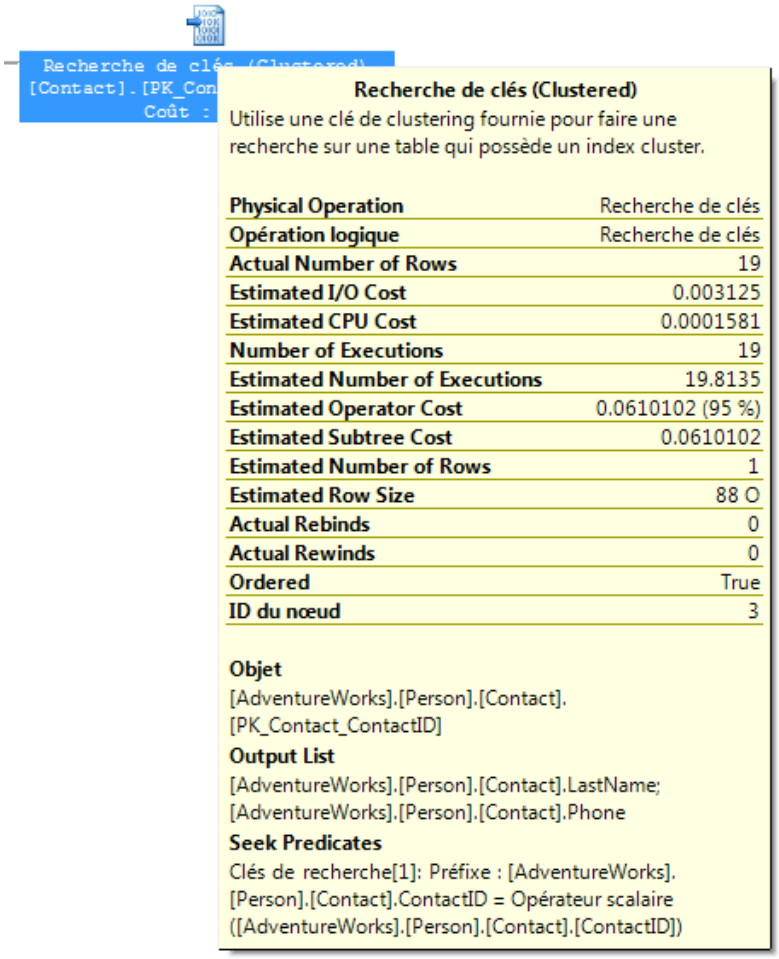


Figure 6

Une Recherche de Clés est une recherche de repère dans une table dotée d'un index clustered. Avant le SP2, cette opération était représentée par un Clustered Index Scan avec une valeur True sur le LookUp.

La première signification d'une Recherche de clés, c'est que l'optimiseur ne peut pas trouver les lignes en une seule opération, et qu'il doit utiliser une clé clustered (ou un identifiant unique de ligne) pour récupérer les lignes correspondantes dans un index clustered (ou dans la table elle-même).

La présence d'une Recherche de Clés indique que les performances de la requête pourraient être améliorées par un index couvrant ou inclusif. Dans un index couvrant ou dans un index inclusif, on trouve toutes les colonnes qui doivent figurer en retour de requête ; de ce fait, pour chaque ligne, toutes les colonnes sont dans l'index, et la Recherche de Clés n'est plus nécessaire.

Une Recherche de Clés est toujours accompagné d'une opération de jointure Boucles Imbriquées, qui fusionne les résultats des deux opérations.

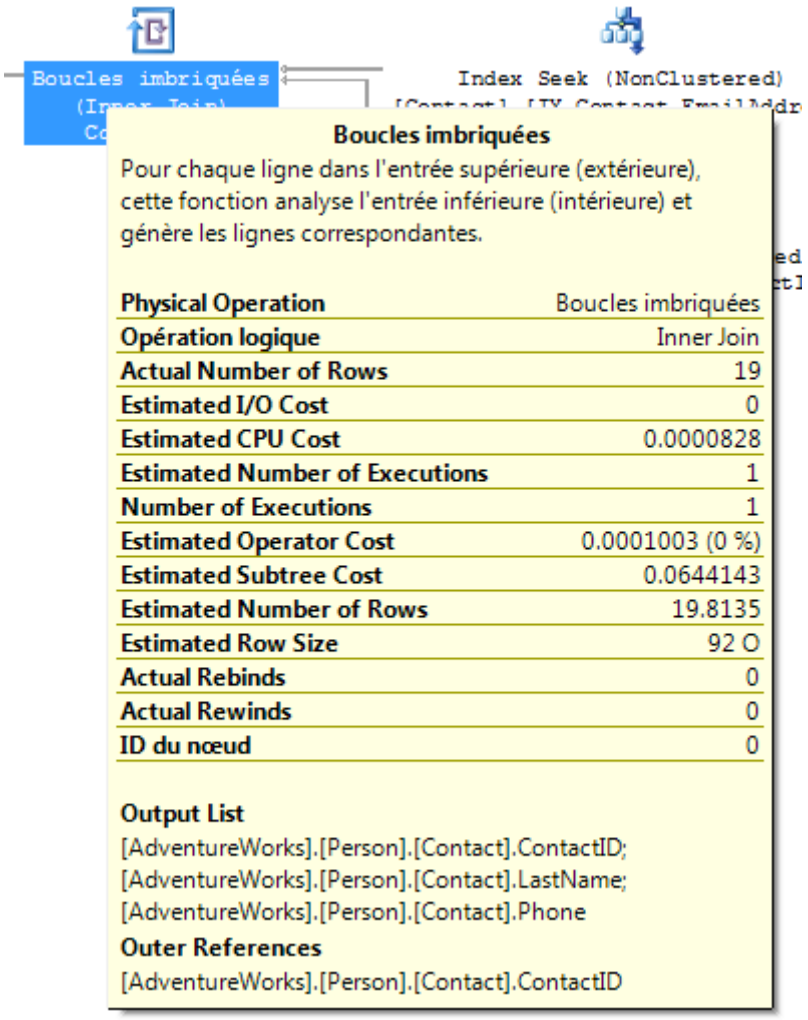


Figure 7

En soi, les Boucles Imbriquées sont une jointure classique, de type standard, et qui n'indique aucun problème de performance. Dans notre exemple, c'est parce qu'une Recherche de Clés est nécessaire que nous avons besoin de Boucles Imbriquées pour recombiner les lignes de l'Index Seek et de la Recherche de Clés. Sans le besoin de cette Recherche de Clés, nous n'aurions pas non plus besoin de ces Boucles Imbriquées, qui n'apparaitrait pas dans notre plan d'exécution graphique.

Table Scan

Le nom de cet opérateur est explicite, c'est un de ceux que nous avons étudié dans notre premier article. Il signale que les lignes demandées ont été retournées en parcourant toute la table, ligne par ligne. Vous pouvez voir une opération de Table Scan en exécutant la requête suivante :

```
SELECT  *
FROM    [dbo].[DatabaseLog]
```

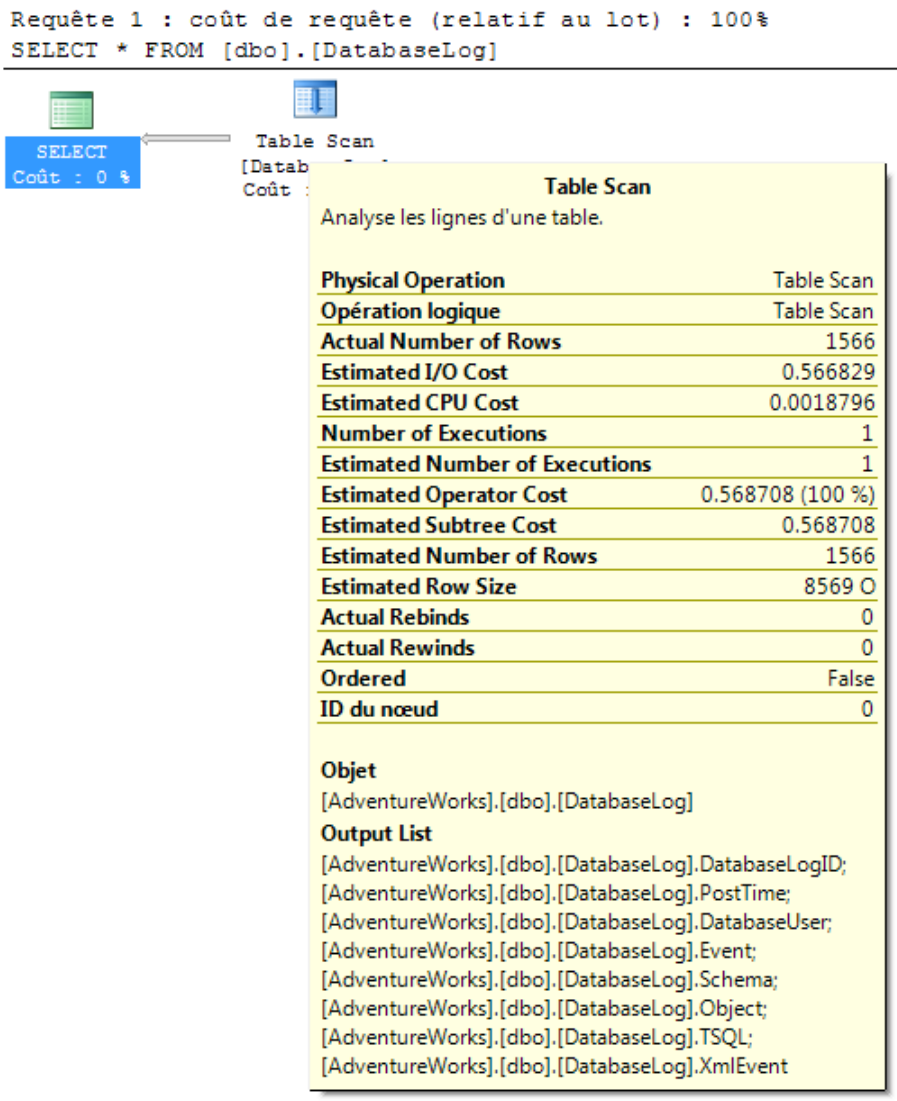


Figure 8

Un scan de table peut survenir pour de nombreuses raisons, mais le plus souvent, c'est parce qu'il n'y a aucun index utilisable dans cette table, et que l'optimiseur doit rechercher ligne par ligne pour identifier celles qui doivent être renvoyées. Une autre raison fréquente est la demande de retour de toutes les lignes, comme dans notre exemple. Quand il faut retourner toutes les lignes d'une table (ou même la majorité d'entre elles), qu'un index existe ou non, il est souvent plus rapide pour l'optimiseur de parcourir la table ligne par ligne plutôt que de chercher chaque ligne dans un index. Une dernière raison est que parfois, il est plus rapide pour l'optimiseur de parcourir toutes les lignes plutôt que d'utiliser un index ; c'est ce qui se passe fréquemment pour les tables comportant peu de lignes.

Pour une table qui contient peu de lignes, un Table Scan ne pose en général pas de problème. Par contre, si la table est grande et que vous demandez beaucoup de lignes, alors vérifiez si vous ne pouvez pas requêter moins de lignes, ou bien ajouter un index pour améliorer les performances.

RID LookUp

Si nous filtrons la précédente requête **DatabaseLog** sur la colonne de clé primaire, alors nous voyons un nouveau plan d'exécution, qui va combiner un Index Seek et un **RID LookUp**.

```
SELECT  *
FROM    [dbo].[DatabaseLog]
WHERE   DatabaseLogID = 1
```

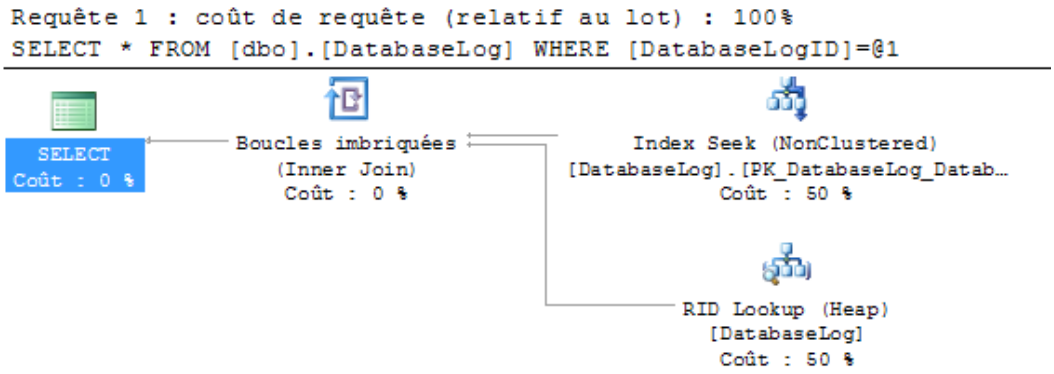


Figure 9

Pour retourner les résultats requis, l'optimiseur commence par un Index Seek sur la clé primaire. Bien que cet index permette d'identifier les lignes correspondant à la clause **WHERE**, toutes les colonnes demandées ne sont pas présentes dans l'index. Comment le savons-nous ?

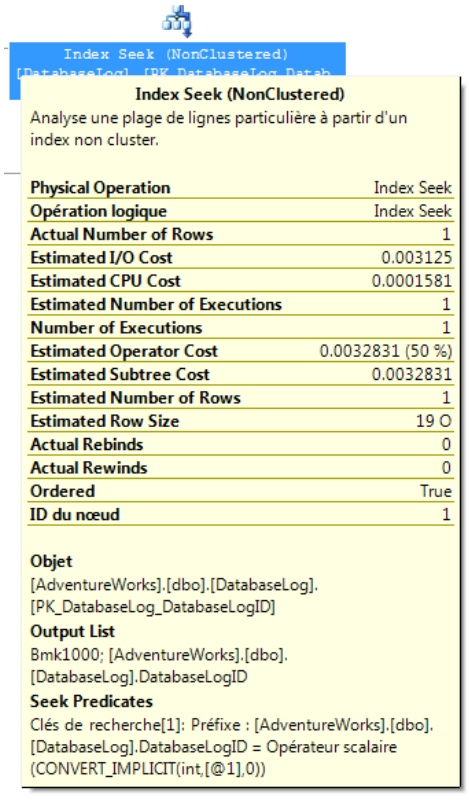


Figure 10

En regardant l'info-bulle sur l'Index Seek, nous voyons « Bmk1000 » dans l'Output List. Ce « Bmk1000 » nous informe que cet Index Seek fait partie d'un plan d'exécution qui contient une recherche de repère.

Ensuite, l'optimiseur exécute un RID LookUp, qui est un type de recherche de repère qui se produit pour les tables « heap » – en tas – les tables qui n'ont pas d'index clustered et qui ont donc besoin d'un identifiant de ligne pour trouver les lignes à renvoyer. En d'autres termes, on pourrait dire qu'en l'absence d'index clustered qui inclut toutes les lignes, il doit utiliser un identifiant de ligne pour faire le lien entre l'index et le tas. Ceci occasionne des opérations supplémentaires en I/O de disque, parce qu'il faut effectuer deux opérations différentes au lieu d'une seule, et qu'elles doivent ensuite être combinées dans des Boucles Imbriquées.

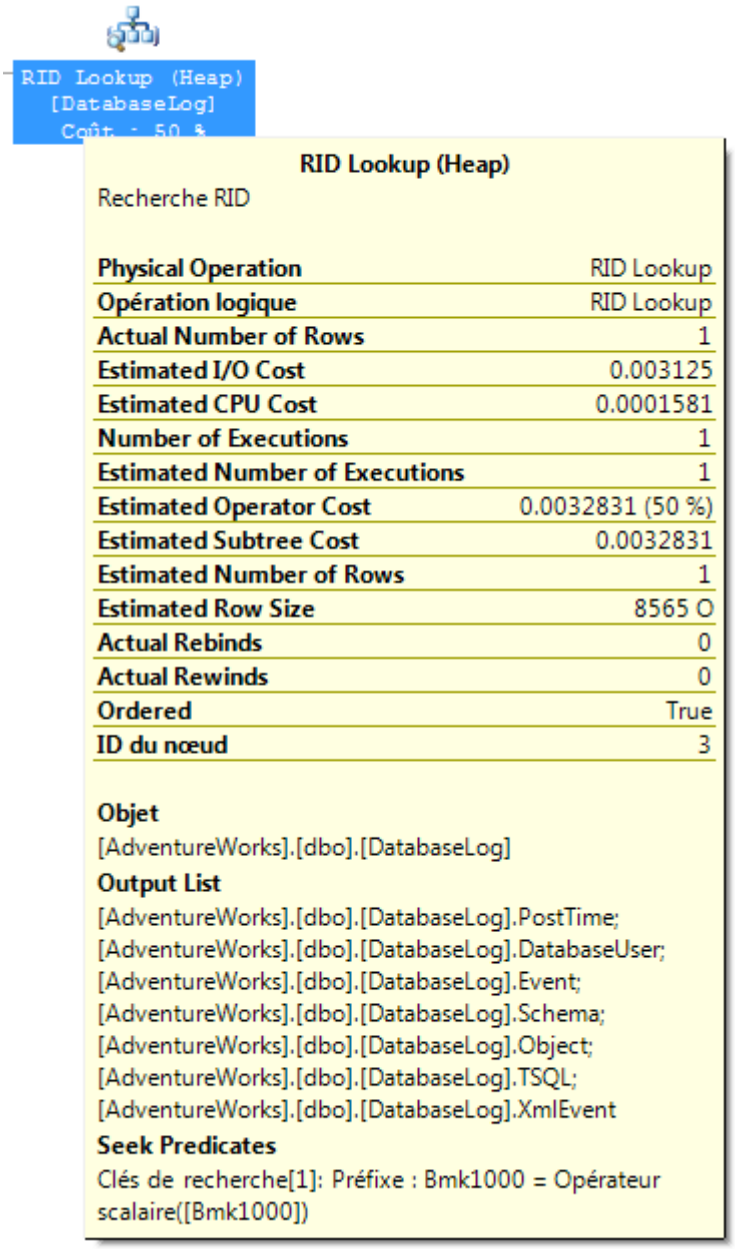


Figure 11

Dans l'info-bulle du RID LookUp ci-dessus, remarquez bien que « Bmk1000 » est à nouveau utilisé, mais cette fois dans la section Seek Predicates (prédicats de recherche). Ceci nous informe de l'utilisation d'une recherche de repère (dans notre exemple, c'est une recherche de RID) en tant que partie de notre plan d'exécution. Dans ce cas particulier, il n'y a qu'une seule ligne à rechercher, ce qui est sans grande conséquence du point de vue performances. Mais si cette recherche de RID doit retourner beaucoup de lignes,

alors il serait bon d'étudier attentivement comment améliorer les performances en diminuant les I/O disque, soit en réécrivant la requête, soit en ajoutant un index clustered, soit en utilisant un index couvrant ou inclusif.

Jointure de Tables

Jusqu'à maintenant, nous n'avons travaillé qu'avec des tables seules. Rendons les choses un peu plus croustillantes, en introduisant des jointures dans notre requête. La requête suivante les information sur les employés, en la présentant de façon plus conviviale par la concaténation des colonnes **FirstName** et **LastName**.

```
SELECT  e.[Title],
        a.[City],
        c.[LastName] + ', ' + c.[FirstName] AS EmployeeName
FROM    [HumanResources].[Employee] e
JOIN [HumanResources].[EmployeeAddress] ed ON e.[EmployeeID]    = ed.[EmployeeID]
JOIN [Person].[Address] a ON [ed].[AddressID] = [a].[AddressID]
        JOIN [Person].[Contact] c ON e.[ContactID] = c.[ContactID];
```

La figure 12 montre le plan d'exécution de cette requête.

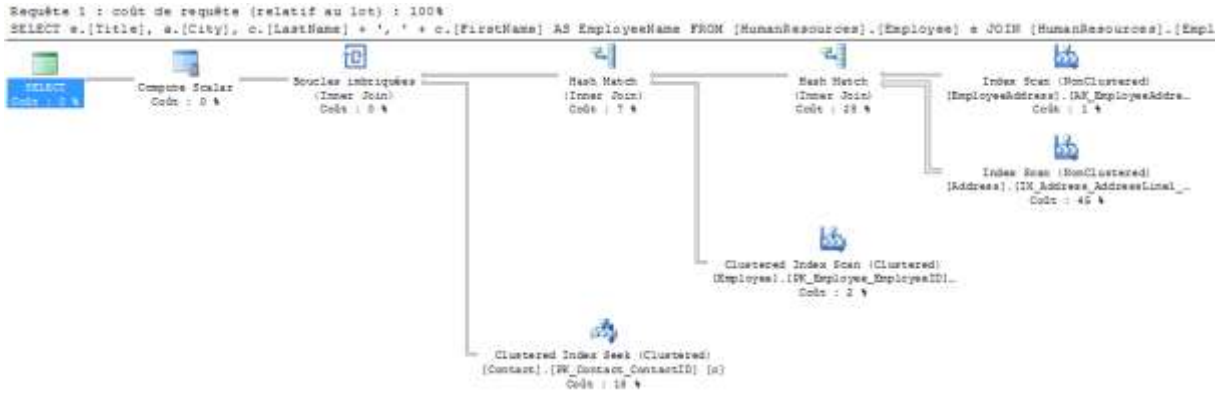


Figure 12

Plusieurs étapes se déroulent dans cette requête, chacune avec des coûts en processeur différents. Les coûts s'additionnent les un aux autres, au fur et à mesure qu'on avance dans l'arbre d'exécution de droite à gauche.

À partir des coûts relatifs affichés sous chaque icône d'opérateur, nous identifions les trois opérations les plus couteuses du plan, par ordre décroissant :

1. L'Index Scan sur la table **Person.Address** (45%)
2. L'opération de jointure Hash Match entre **HumanResources.EmployeeAdress** et **Person.Address** (28%)
3. Le Clustered Index Seek sur la table **Person.Contact (**18%)

Examinons tous les opérateurs de ce plan.

En commençant par la droite de la figure 12 ci-dessus, la première chose que nous voyons est un Index Scan sur la table **HumanResources.EmployeeAdress** et immédiatement en dessous nous avons un autre Index Scan, sur la table **Person.Address**. Étant donné que cette dernière est l'opération la plus couteuse du plan, regardons-la plus attentivement. L'affichage de l'info-bulle visible à la figure 13 nous révèle l'exécution d'un scan sur l'index **IX_Address_AddressLine_AddressLine2_City_StateProvinceId_PostalCode** et que le moteur doit parcourir 19 614 lignes pour atteindre les données demandées.

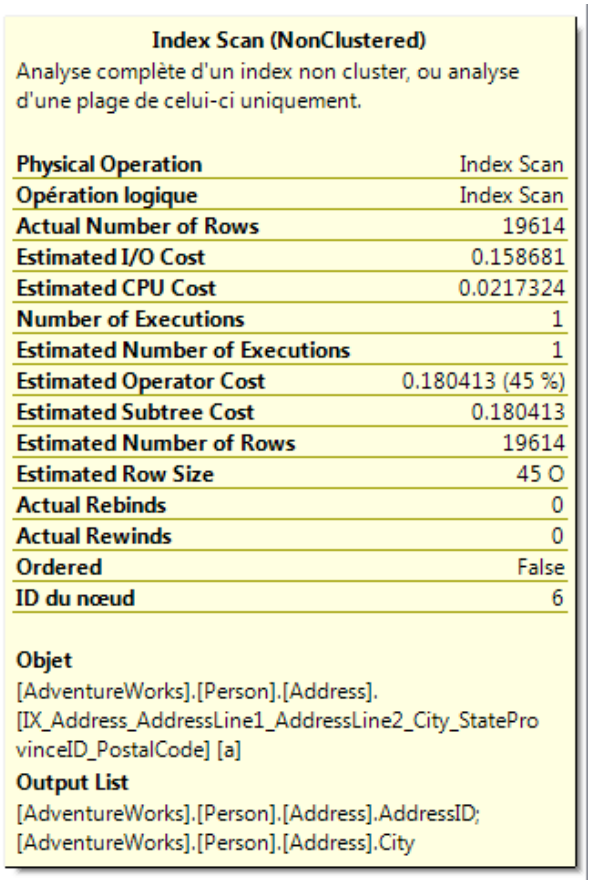


Figure 13

L'optimiseur a eu besoin des colonnes **AdressId** et **City**, comme on le voit dans la Output List. En se basant sur la sélectivité des index et sur les colonnes de la tables, l'optimiseur a déterminé que la meilleure façon d'atteindre ces données était de traverser l'index. C'est ce parcours au travers de ces 19 614 lignes qui représente 45% du coût total, pour un coût estimé de 0.180413. l'opérateur de coût estimé représente ce qu'il en a couté à l'optimiseur pour effectuer cette opération spécifique. Il s'agit d'un calcul interne fait par l'optimiseur, et qu'il utilise pour déterminer les coûts relatifs de toutes les opérations. Plus bas sera ce coût, plus efficace sera l'opération.

Hash Match (Join)

En continuant dans notre exemple ci-dessus, nous voyons que la sortie des deux index scan est combinée dans une **jointure Hash Match**, notre deuxième opération par ordre de coût dans notre plan d'exécution. L'info-bulle de cet opérateur est montrée figure 14 :

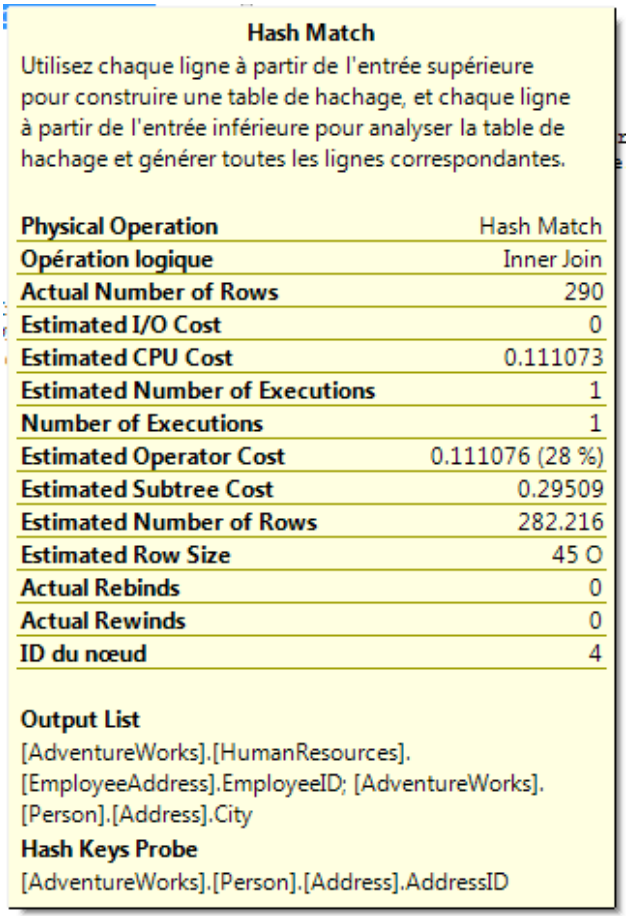


Figure 14

Pour comprendre ce qu'est une jointure Hash Match, nous devons d'abord comprendre deux nouveaux concepts : le **hashage** et la **hash table**. Le hashage est une technique de programmation dans laquelle des données sont converties dans une forme symbolique qui permet de les rechercher plus rapidement. Par exemple, une ligne de données dans une table va être convertie en une valeur unique qui représentera le contenu de la ligne. En quelque sorte, c'est un peu comme si on prenait une ligne et qu'on l'encryptait. Comme le cryptage, une valeur hashée peut être reconvertie dans sa valeur d'origine. SQL Server utilise souvent le hashage pour convertir les données dans une forme plus efficace pour les travailler, ici pour accélérer les recherches.

Une Hash Table est une structure de données qui divise et répartit tous les éléments dans des blocs de dimension identique, pour accéder plus rapidement à ces éléments. La fonction de hashage détermine le bloc de destination d'un élément. Par exemple, on peut prendre une ligne d'une table, lui appliquer un hashage pour obtenir une valeur hashée, et stocker cette valeur hashée dans une has table.

Nous pouvons maintenant comprendre ce qu'est une jointure **Hash Match** : SQL Server joint deux tables en hashant les lignes de la plus petite, puis insère toutes ces lignes hashées dans une hash table, et traite ensuite la table la plus grande ligne par ligne, en recherchant dans la table hashée (la plus petite) les occurrences des lignes qui doivent être jointes. Cette hash table est petite, puisqu'elle est issue de la plus petite des tables d'origine, et comme elle stocke des valeurs hashées à la place des vraies valeurs, les recherches y sont très rapides. Tant que la table destinée au hashage est assez petite, ce process est très rapide ; mais a contrario, si les deux tables sont grosses, une jointure Hash Match peut s'avérer beaucoup moins performante que d'autres types de jointure.

Dans notre exemple, les données de **HumanResources.EmployeeAddress.AddressId** sont mises en correspondance avec celles de la table **Person.Address**.

Les jointures Hash Match sont souvent très performantes sur de gros ensembles de données, en particulier quand une des tables est significativement plus petite que l'autre. Elles fonctionnent aussi correctement pour les tables qui ne sont pas ordonnées sur les colonnes de jointure, et elles peuvent se montrer performantes quand il n'y a pas d'index utilisable. D'un autre côté, la présence d'une jointure Hash Match signale qu'on aurait pu utiliser une autre méthode de jointure plus efficace (Boucles Imbriquées ou Merge). Par exemple, une jointure Hash Match dans un plan d'exécution indique parfois :

- Un index manquant ou incorrect
- Une clause **WHERE** manquante
- Une clause **WHERE** comportant un calcul ou une conversion qui la rend « non sargeable ». ce néologisme très fréquent signifie que l'argument de la recherche (sarg en anglais) n'est pas utilisable. Concrètement, on ne peut pas utiliser d'index existant.

Même si la jointure Hash Match se trouve être la meilleure solution trouvée par l'optimiseur pour joindre deux tables, ça ne veut pas dire qu'il n'y a pas d'approche plus performante – par exemple l'ajout d'index appropriés à la jointure, la diminution du nombre de lignes retournées en rendant la clause WHERE plus restrictive, ou la modification de cette clause pour la rendre « sargeable ». En bref, quand vous voyez une jointure Hash Match, vous devez vérifier si vous pouvez améliorer cette jointure. Si c'est possible, tant mieux ! et si ça ne l'est pas, il n'y a rien d'autre à faire, la jointure Hash Match sera la meilleure solution pour effectuer la jointure.

Dans notre exemple, le petit écart entre le nombre de lignes estimé (282.16) et le nombre réel (290) est sans signification ; il montre seulement qu'il s'agit d'une estimation calculée, puisqu'il est impossible de renvoyer 0.216 lignes. Il n'y a pas lieu de se préoccuper d'un écart si petit, mais un écart plus important serait l'indicateur que vos statistiques ne sont plus à jour et doivent être actualisées. Un écart important peut amener à ce que le plan d'exécution réel ne soit pas le même que le plan estimé.

La requête continue ensuite avec un autre Index Scan sur la table **HumanResources.Employee**, puis un autre Hash Match entre les résultats du premier et ceux de l'index scan.

Clustered Index Seek

Après la jointure Hash Match, nous trouvons un **Clustered Index Seek** ; cette opération est faite sur la table **Person.Contact**, très exactement sur la **PK_Contact_ContactId**, qui est à la fois clé primaire et index clustered de cette table. Par ordre de coût décroissant, cette opération vient au troisième rang du plan d'exécution. Son info-bulle est reproduite figure 15 :

Clustered Index Seek (Clustered)

Analyse d'une plage de lignes particulière à partir d'un index cluster.

Physical Operation	Clustered Index Seek
Opération logique	Clustered Index Seek
Actual Number of Rows	290
Estimated I/O Cost	0.003125
Estimated CPU Cost	0.0001581
Estimated Number of Executions	282.216
Number of Executions	290
Estimated Operator Cost	0.0702501 (18 %)
Estimated Subtree Cost	0.0702501
Estimated Number of Rows	1
Estimated Row Size	113 O
Actual Rebinds	0
Actual Rewinds	0
Ordered	True
ID du nœud	10

Objet
[AdventureWorks].[Person].[Contact].
[PK_Contact_ContactID] [c]

Output List
[AdventureWorks].[Person].[Contact].FirstName;
[AdventureWorks].[Person].[Contact].LastName

Seek Predicates
Clés de recherche[1]: Préfixe : [AdventureWorks].[Person].
[Contact].ContactID = Opérateur scalaire([AdventureWorks].
[HumanResources].[Employee].[ContactID] as [e].
[ContactID])

Figure 15

On remarquera dans la section **Seek Predicates** de la figure 15 ci-dessus, que cette opération a fait directement la jointure entre la colonne **ContactId** de la table **HumanResources.Employee** et la table **Person.Contact**.

Jointure de Boucles Imbriquées

Après le Clustered Index Seek, les données provenant des autres opérations sont jointes aux données issues de ce Seek, par une **Jointure de Boucles Imbriquées**, comme on le voit sur la figure 16.

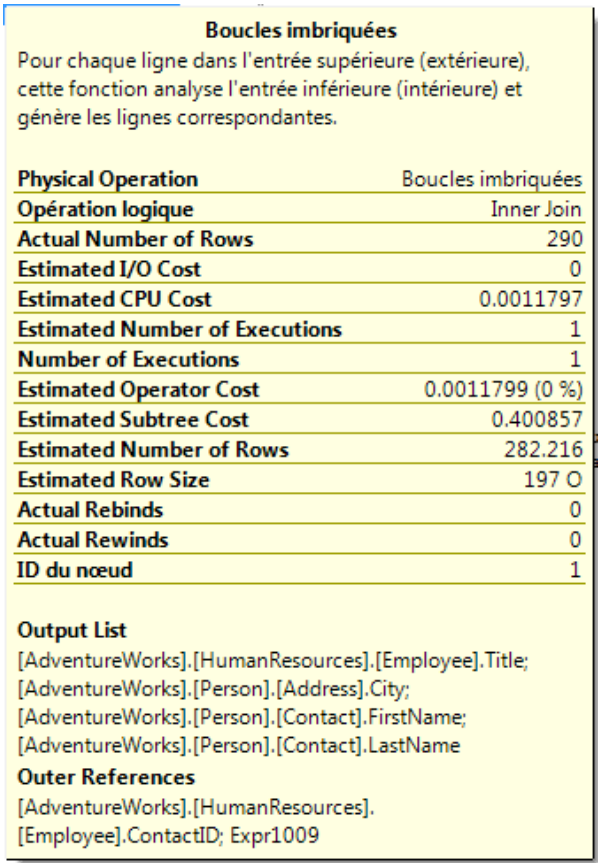


Figure 16

On appelle aussi cette **Jointure de Boucles Imbriquées** « itération imbriquée ». En entrée, cette opération utilise deux ensemble de données. Elle les joint en parcourant l'ensemble de données externe (l'opérateur du bas, sur le plan d'exécution graphique) pour chaque ligne de l'ensemble interne. Étant donné que ces deux ensembles comportaient peu de lignes, c'est une opération très performante. Ce mécanisme de jointure reste d'une extraordinaire efficacité tant que le nombre de lignes du jeu de données interne est petit, et que le jeu de données externe est indexé (quelle que soit sa taille). C'est le genre de jointure qu'il faut obtenir, sauf dans le cas de très grands jeux de données.

Compute Scalar

Pour finir, juste avant l'opération de Select, notre plan d'exécution de la figure 12 nous montre une opération **Compute Scalar**. L'info-bulle de cet opérateur est affichée figure 17.

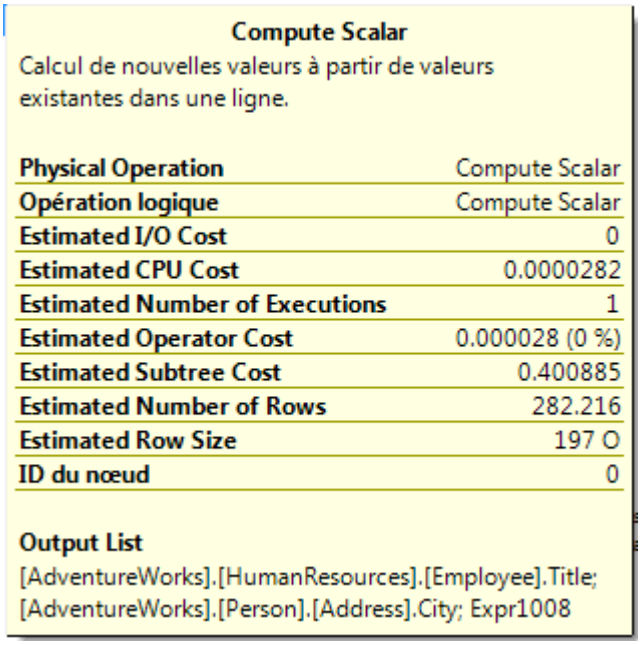


Figure 17

Il s'agit de la représentation d'une opération qui produit un scalaire (une valeur unique), en général à partir d'un calcul. Ici, c'est l'alias **EmployeeName**, qui combine les colonnes **Contact.LastName** et **Contact.FirstName** avec une virgule de séparation. Bien que le coût de cette opération ne soit pas égal à zéro (0.0000282), il est si faible par rapport aux autres qu'on le négligera.

Merge Join

En plus des jointures de Hash et des boucles imbriquées, l'optimiseur de requêtes peut aussi réaliser des **Merge Join** (jointures de fusion). Lançons la requête suivante sur la base de données AdventureWorks pour observer une Merge Join :

```
SELECT  c.CustomerID
FROM    Sales.SalesOrderDetail od
        JOIN Sales.SalesOrderHeader oh
            ON od.SalesOrderID = oh.SalesOrderID
        JOIN Sales.Customer c
            ON oh.CustomerID = c.CustomerID
```

Cette requête produit le plan d'exécution visible figure 18 :

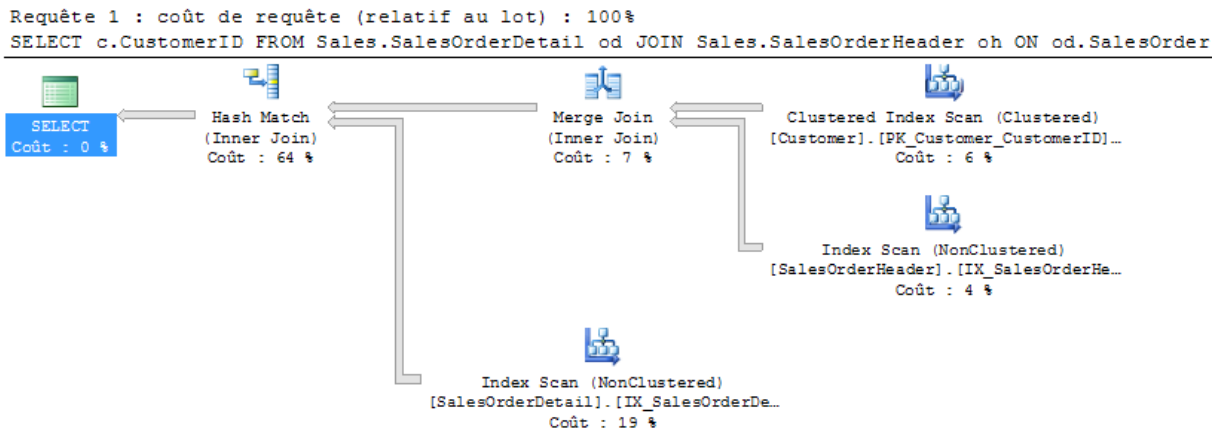


Figure 18

En suivant le plan d'exécution, on voit que l'optimiseur de requête effectue un Clustered Index Scan sur la table **Customer** et un non-clustered index scan sur la table **SalesOrderHeader**. Comme nous n'avons pas spécifié de clause **WHERE**, un scan a été fait sur chaque table pour en extraire toutes les lignes.

Ensuite, toutes les lignes provenant des tables **Customers** et **SalesOrderHeader** sont jointes par l'opérateur **Merge Join**. Si les colonnes de jointures sont déjà ordonnées (triées), alors on aura un Merge Join sur les tables. L'info-bulle du Merge Join présentée figure 19 en montre un exemple, dans lequel nous voyons que les colonnes de jointure sont les colonnes **CustomerId** des tables **SalesOrderHeader** et **Customer**. Dans notre cas, les données des colonnes de jointure étaient déjà triées dans l'ordre requis. Un Merge Join est une approche performante, quand il s'agit de joindre deux tables dont les colonnes de jointure sont déjà ordonnées. Dans le cas où ces colonnes ne sont pas ordonnées, l'optimiseur de requête a le choix entre deux options : a) commencer par trier les colonnes de jointure pour faire ensuite un Merge Join, ou bien b) utiliser un Hash Join, moins performant. L'optimiseur de requêtes va évaluer ces options, et en règle générale, va choisir le plan d'exécution qui consommera le moins de ressources.

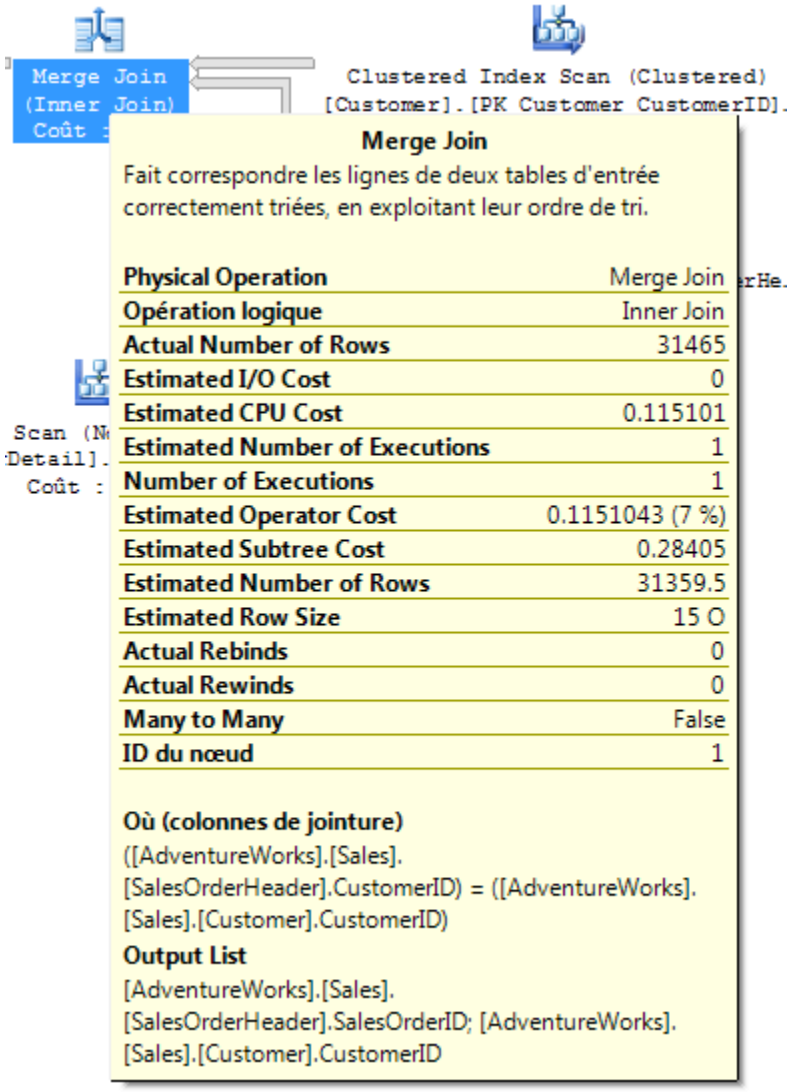


Figure 19

Après que le Merge Join ait joint deux des tables, c'est un Hash Match (dont nous avons déjà parlé) qui va joindre la troisième aux deux premières. Pour finir, on retourne les lignes jointes.

Le point clé de la performance d'un Merge Join, c'est que les colonnes de jointure soient déjà ordonnées. Si elles ne le sont pas, et que l'optimiseur de requêtes choisit de trier les données avant de faire un Merge Join, alors cela montre que le Merge Join n'est pas le moyen idéal pour joindre ces tables, et devrait vous inciter à envisager d'autres index.

Ajoutons une Clause WHERE

On voit rarement des requêtes qui ne comportent pas de restriction sur l'ensemble résultant, en d'autre termes, qui n'ont pas de clause **WHERE**. Nous allons donc étudier deux requêtes conditionnelles multi-tables à travers leurs plans d'exécution.

Lancez la requête suivante sur AdventureWorks, et regardez son plan d'exécution. Cette requête est identique à celle que nous avons vu au début du chapitre Jointure de Tables, si ce n'est qu'elle comporte maintenant une clause **WHERE**.

```
SELECT  e.[Title],
        a.[City],
        c.[LastName] + ',' + c.[FirstName] AS EmployeeName
FROM    [HumanResources].[Employee] e
        JOIN [HumanResources].[EmployeeAddress] ed ON e.[EmployeeID] = ed.[EmployeeID]
   JOIN [Person].[Address] a ON [ed].[AddressID] = [a].[AddressID]
        JOIN [Person].[Contact] c ON e.[ContactID] = c.[ContactID]
WHERE   e.[Title] = 'Production Technician - WC20' ;
```

La figure 20 montre le plan d'exécution actuel de cette requête :

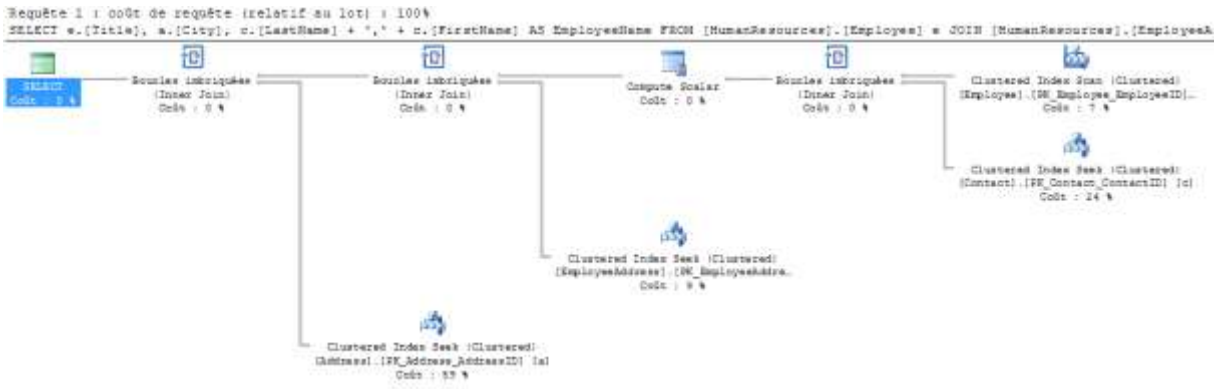


Figure 20

Toujours en commençant par la droite, nous voyons que l'optimiseur a utilisé le critère de la clause **WHERE** pour faire un Clustered Index Scan en utilisant la clé primaire. La clause **WHERE** a limité le nombre de ligne à 22 ; vous pouvez le voir en positionnant le pointeur de la souris sur la flèche sortant de l'opérateur **Clustered Index Scan** (figure 21).

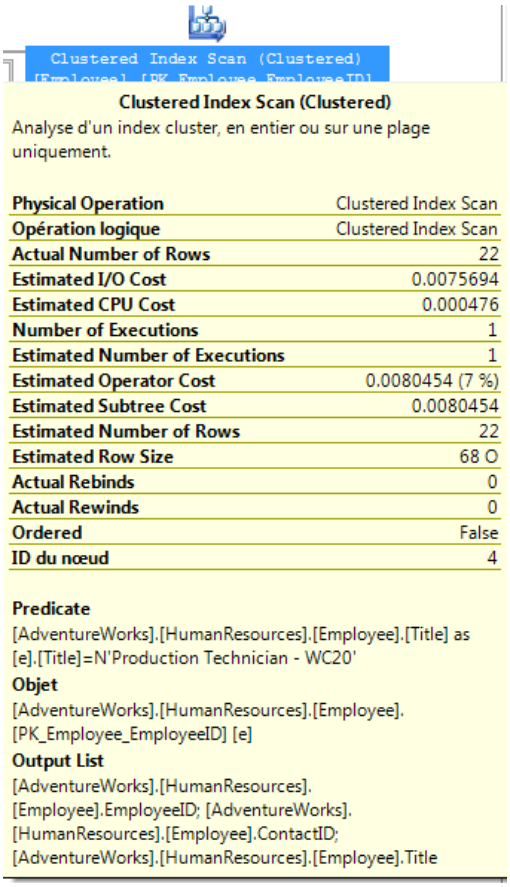


Figure 21

En se basant sur les statistiques disponibles, l'optimiseur a pu le déterminer avant le reste, comme on le constate en comparant dans l'info-bulle le nombre de lignes estimé avec le nombre de lignes réel.

Par rapport à la requête précédente, nous travaillons sur un ensemble de données plus petit, et avec un bon index sur la table **Person.Contact** ; de ce fait, l'optimiseur a pu utiliser la jointure **Nested Loop**, plus performante. Et comme l'optimiseur a déplacé l'emplacement où faire la jointure, il a aussi déplacé le calcul scalaire (Compute Scalar) juste à droite de la jointure. Et comme il ne reste que 22 lignes à la sortie de l'opération de scalaire, c'est un Clustered Index Seek et un Nested Loop qui ont été utilisés pour joindre les données à celles de la table **HumanResources.EmployeeAdress**. Ceci nous amène au dernier Clusteed Index Seek et au dernier Nested Loop. Si on compare à la requête initiale (qui ne comportait pas de clause **WHERE**), c'est la réduction de l'ensemble de données initial par la clause **WHERE** qui a rendu possible l'utilisation de ces jointures plus performantes.

Souvent, les développeurs qui ne sont pas à l'aise avec le T-SQL proposent comme solution simple de retourner toutes les lignes à l'application, sans faire de jointure entre les tables, voire même sans ajouter de clause **WHERE**. La requête que nous venons d'étudier était très simple, et comportait très peu de lignes, mais vous pouvez la prendre comme exemple si vous êtes amenés à argumenter dans ce genre de débat. Avec une clause **WHERE**, le coût du sous-arbre final dans l'optimiseur était de 0.112425 ; comparez-le au 0.400885 de la requête précédente. C'est quatre fois plus rapide, même sur cette petite requête, toute simple. Alors imaginez ce que ça peut donner sur des ensembles de données plus gros et des requêtes plus complexes.

Plans d'Exécution avec GROUP BY et ORDER BY

Les plans d'exécution affichent d'autres opérateurs quand on ajoute d'autres clauses de base.

Trier

Pour notre exemple, prenons un simple select avec une clause **ORDER BY** :

```
SELECT  *
FROM    [Production].[ProductInventory]
ORDER BY [Shelf]
```

Le plan d'exécution est affiché figure 22.

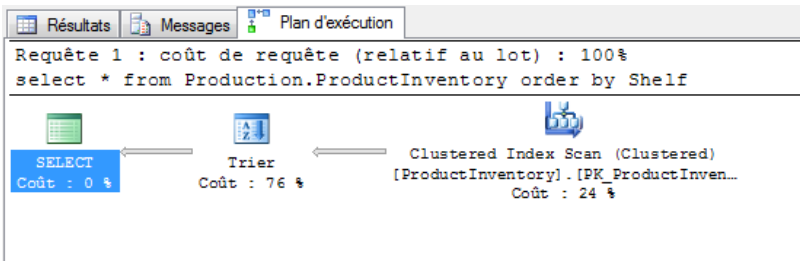


Figure 22

L'opérateur **Clustered Index Scan** dirige sa sortie sur l'opérateur **Trier**. Comparé à de nombreuses autres icones de plan d'exécution, l'opérateur Trier est sans surprise. Il montre vraiment quand l'optimiseur de requête trie les données dans le plan d'exécution. Sans précision dans la clause **ORDER BY**, l'ordre par défaut est ascendant, comme vous pouvez le voir dans l'info-bulle de l'icône **Trier** (figure 23 ci-dessous).

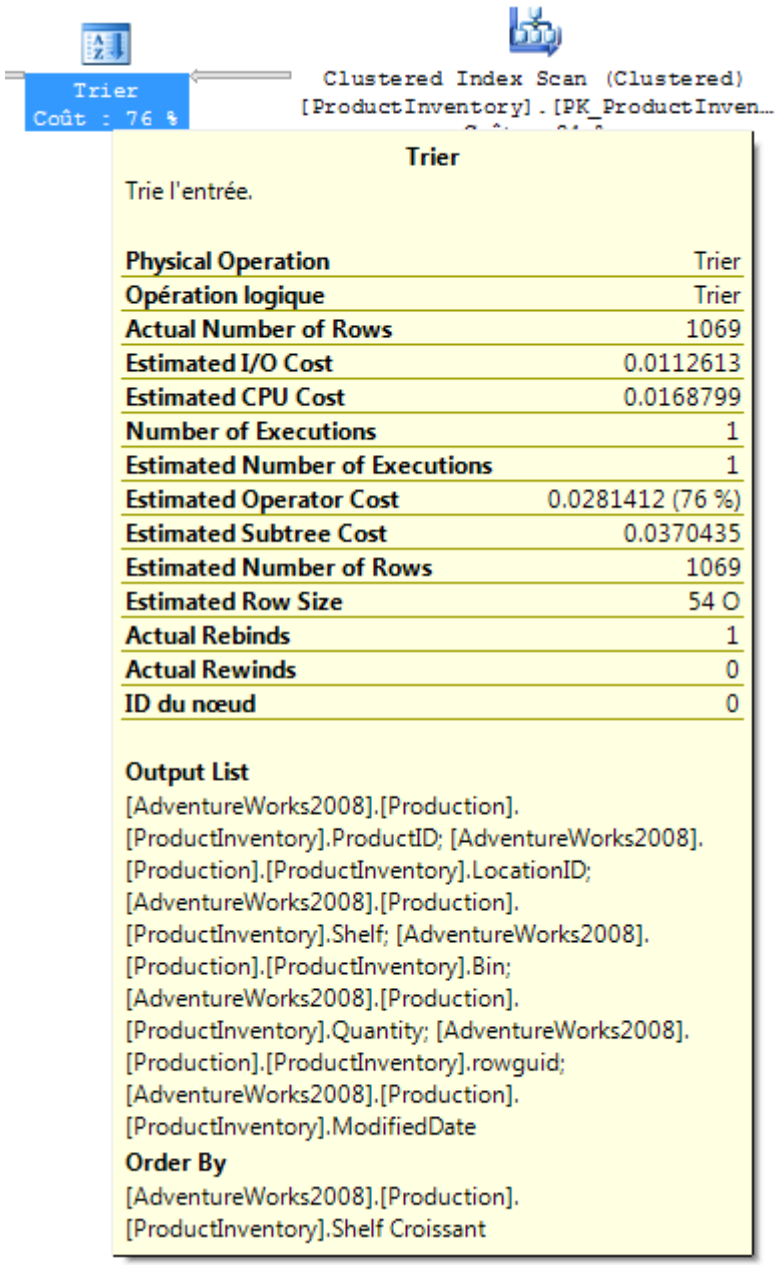


Figure 23

Regardons l'info-bulle de la flèche qui précède l'icône **Trier** (figure 24) : nous constatons que 1069 lignes sont envoyées à l'opérateur Sort. L'opérateur Sort reçoit ces 1069 lignes du Clustered Index Scan, les trie, et restitue ensuite ces 1069 lignes triées.

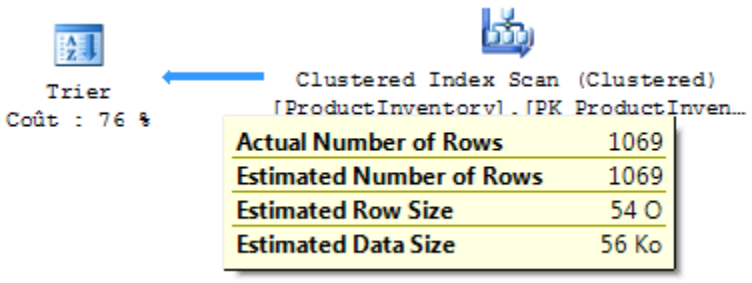


Figure 24

Le point le plus important à noter est que l'opération de Tri représente 76% du coût total de la requête. Comme il n'y a pas d'index sur cette colonne, c'est dans la requête que ce tri est réalisé.

De façon empirique, je dirais qu'il convient de réexaminer attentivement une requête pour l'optimiser, si le tri prend plus de 50% du temps d'exécution total. La raison pour laquelle je ne mets pas ce précepte en application ici est très simple : il nous manque une clause **WHERE**. Il est probable que cette requête retourne plus de lignes à trier que ce qui serait nécessaire. Toutefois, même en présence d'une clause **WHERE**, vérifiez bien que cette clause restreint bien le nombre de lignes à ce qui vous est strictement nécessaire, sans lignes qui ne seront jamais utilisées ensuite.

Il vous faut aussi les points suivants :

- Le tri est-il réellement nécessaire ? si non, enlevez-le pour réduire la charge de travail
- Est-il possible d'obtenir les données déjà triées de façon à ne pas avoir à le faire ? par exemple, pourrait-on utiliser un index clustered qui trierait déjà les données dans l'ordre requis ? ça n'est pas toujours possible, mais si ça l'est, vous éviterez la charge liée au tri en créant l'index clustered adéquat.
- Si un plan d'exécution présente plusieurs opérateurs de tri, vérifiez s'ils sont tous nécessaires, ou s'il n'est pas envisageable de réécrire le code en ne gardant que les tris dont vous avez vraiment besoin pour les besoins de cette requête.

Si nous modifions la requête comme suit :

```
SELECT  *
FROM    [Production].[ProductInventory]
ORDER BY [ProductID]
```

Nous obtenons le plan d'exécution visible figure 25 :

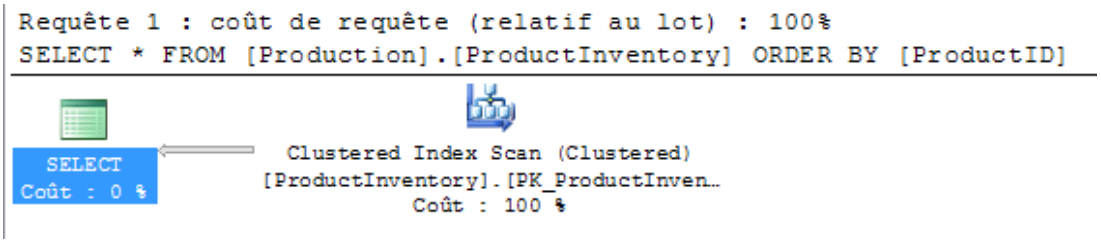


Figure 25

Bien que cette requête soit quasiment identique à la précédente et comporte une clause **ORDER BY**, nous constatons qu'il n'y a pas d'opérateur de tri dans le plan d'exécution. C'est parce que nous avons changé la colonne sur laquelle porte le tri, et qu'il existe un index clustered sur cette colonne. Il n'est donc plus nécessaire de trier à nouveau les données retournées, puisqu'elles l'ont déjà été nativement en étant l'index clustered. L'optimiseur de requête est suffisamment astucieux pour voir que les données sont déjà triées et qu'il n'est pas utile de le refaire.

Si vous n'avez vraiment pas le choix et que vous devez trier beaucoup de données, il sera bon de vérifier dans le Profiler SQL si une Alerte de Tri apparaît. Pour améliorer les performances, SQL Server essaie de trier en mémoire plutôt que sur disque. Un tri dans la RAM est beaucoup plus rapide qu'un tri sur le disque. Mais quand cette opération est importante en volume, SQL Server est parfois dans l'impossibilité de trier en mémoire, et doit écrire des données dans la base de données TempDb. Quand cela se produit, SQL Server génère un événement d'Avertissement de Tri, que le Profiler SQL peut intercepter dans une trace. Et si vous constatez que votre serveur fait beaucoup de tris, et que de nombreux Avertissements de Tri sont générés, alors il vous faut ou bien ajouter de la RAM, ou bien accélérer l'accès à TempDb.

Hash Match (Aggregate)

Nous avons déjà parlé de l'opérateur de jointures Hash Match précédemment. C'est ce même opérateur qui sera utilisé quand une requête comporte une agrégations. Prenons cette requête d'agrégat simple, qui utilise l'opérateur **COUNT** sur une table unique :

```
SELECT  [City]
        COUNT([City]) AS CityCount
FROM    [Person].[Address]
GROUP BY [City]
```

Voici son plan d'exécution :

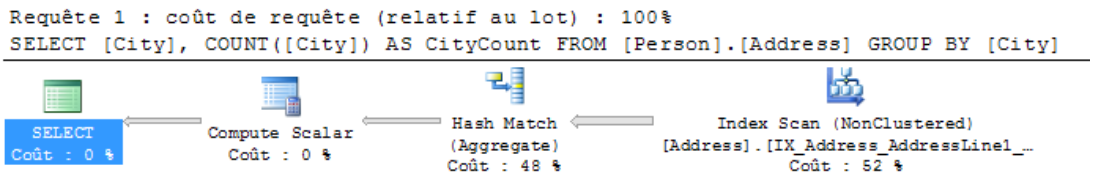


Figure 26

La requête débute par un Index Scan, puisqu'elle doit renvoyer toutes les lignes. Il n'y a pas de clause **WHERE** pour filtrer les données. Pour exécuter l'opération d'agrégat demandée **COUNT**, ces lignes doivent être agrégées. Pour que l'optimiseur de requête puisse compter les lignes séparément pour chaque ville (city), il doit faire une opération de Hash Match. Remarquez que sur le pan d'exécution, le Hash Match en question est agrémenté du mot « Aggregate » entre parenthèses. C'est ce qui permet de le différencier d'une opération de Hash Match de Jointure. Comme dans un Hash Match de jointure, un Hash Match d'agrégat fait que SQL Server crée une table de Hashage temporaire en mémoire, pour compter le nombre de lignes qui correspondent à la colonne de **GROUP BY** (dans notre cas, c'est la colonne « city »). Après que les résultats sont agrégés, ils nous sont retournés.

La plupart du temps, les agrégations dans des requêtes sont des opérations couteuses. La seule approche pour accélérer les performances depuis le code, est de s'assurer qu'on a une clause **WHERE** suffisamment restrictive pour limiter le nombre de lignes à ce qui doit être agrégé, et donc faire porter l'agrégation sur le strict nécessaire.

Filtre

En ajoutant juste une clause HAVING à la requête précédente, notre plan d'exécution devient plus complexe.

```
SELECT  [City],
        COUNT([City]) AS CityCount
FROM[Person].[Address]
GROUP BY [City]
HAVING  COUNT([City]) > 1
```

Voici notre plan d'exécution (figure 27)

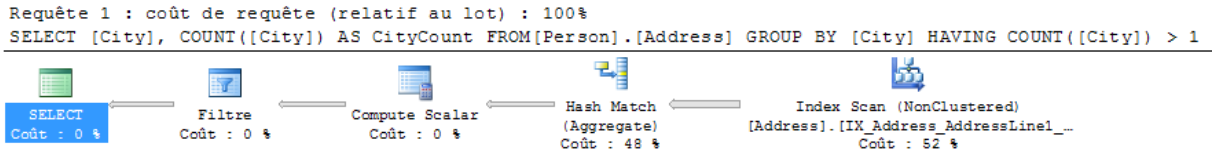


Figure 27

En ajoutant la clause HAVING, l'opérateur de Filtre a été ajouté au plan d'exécution. Nous voyons que cet opérateur de filtre sert à limiter la sortie aux valeurs dont la colonne **City** est supérieure à 1. On peut aussi déduire de ce plan que la clause **HAVING** n'est appliquée que quand toute l'agrégation des données a été effectuée. On ne peut le voir qu'en comparant le nombre de lignes sortant du **Hash Match** (575) avec ce qui sort du **Filtre** (348).

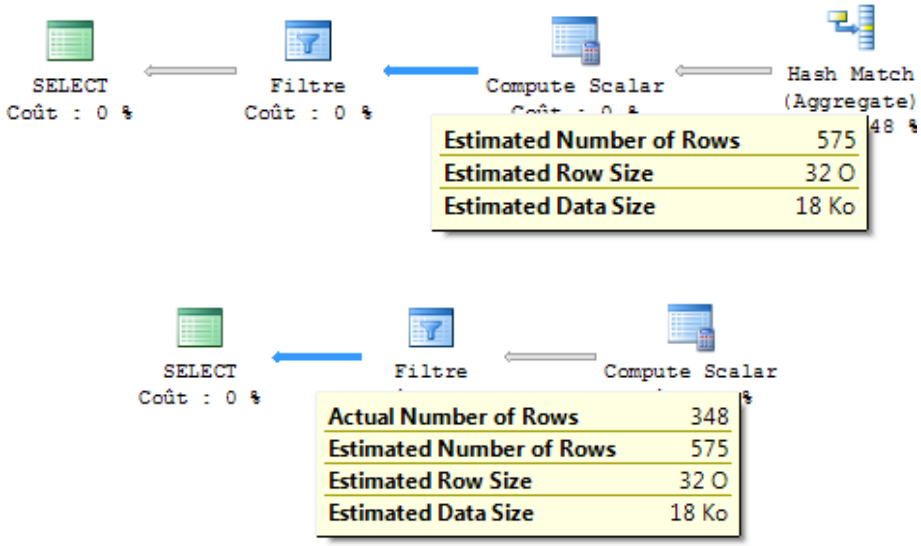


Figure 28

Même si l'ajout d'une clause HAVING réduit la quantité de données retournées, il faut plus de ressources pour produire le résultat de la requête : en effet, la clause **HAVING** n'entre en jeu qu'après l'agrégation. Comme dans l'exemple précédent, pour améliorer les performances d'une requête avec agrégation, il n'y a qu'un moyen : ajouter une clause **WHERE** à la requête pour limiter le nombre de lignes à extraire et à agréger.

Expliquons les Rebinds et les Rewinds

Tout au long de cet article, nous avons pu voir plusieurs fois dans les info-bulles d'opérateurs physiques ces expressions :

- « Actual Rebinds » ou « Estimated Rebind »
- « Actual Rewind » ou « Estimated Rewinds »

La plupart du temps dans cet article, la valeur de ces rebinds et rewinds [1] était zéro, mais pour l'opérateur de Tri (vu précédemment), nous pouvions voir **un** « actual rebind » et **zéro** « actual rewind ».

[1] Note du Traducteur : les mots Rebind et Rewind peuvent être traduits par « Reliaison » et « Rembobinage ». toutefois, j'ai préféré conserver les mots en anglais, la traduction n'apparaissant nulle part à l'affichage des plans d'exécution.

Nous avons besoin de quelques connaissances supplémentaire pour comprendre ce que ces valeurs signifient. À chaque fois que dans un plan d'exécution survient un opérateur physique tel qu'un TRI, les trois choses suivantes se produisent :

- En premier, l'opérateur physique est initialisé, et toutes les structures de données nécessaires sont définies. Ceci porte le nom de méthode **Init()**. Dans tous les cas, ceci advient une fois par opérateur (il est toutefois possible que cela adviennent plusieurs fois).
- Deuxièmement, l'opérateur physique prend (ou reçoit) les lignes de données sur lesquelles il doit agir. Cette méthode s'appelle **GetNext()**. En fonction de son type, un opérateur peut recevoir zéro ou plusieurs appels **GetNext()**.
- Troisièmement, après que l'opérateur ait fait ce qu'il avait à faire, il lui faut s'auto-nettoyer et s'auto-fermer. C'est la méthode **Close()**. Un opérateur physique reçoit toujours un seul appel **Close()**.

Les Rebind et Rewind comptent le nombre de fois où la méthode **Init()** est appelée par un opérateur. Ils comptent tous les deux ce nombre d'appels, mais ils le font dans des circonstances différentes.

Un compte de Rebind se produit quand un ou plusieurs des paramètres corrélés à une jointure sont modifiés, et qu'il faut réévaluer le côté interne de la jointure. Un compte de Rewind se produit quand aucun des paramètres corrélés n'est changé, et qu'il est possible de réutiliser l'ensemble de résultat interne précédent. À chaque fois qu'une de ces conditions est vérifiée, un Rebind ou un Rewind est déclenché, et son décompte est augmenté.

Sur la base de ces explications, il semblerait normal de voir une valeur supérieure ou égale à un pour les Rebind et les Rewind dans toutes les info-bulles et fenêtres de propriété pour les opérateurs physiques. Eh bien non ! bizarrement, le nombre de Rebind et de Rewind n'est incrémenté que pour certains opérateurs physiques précis, et n'est pas modifié pour les autres. Par exemple, le nombre de Rebind et de Rewind est incrémenté pour chacun des six opérateurs suivants :

- NonClustered Index Spool
- Remote Query
- Row Count Spool
- Tri
- Table Spool
- Fonction de Table Valuée

Si un des opérateurs suivants survient, le nombre de Rebind et de Rewind sera incrémenté seulement si la **StartupExpression** est définie à TRUE, ce qui dépend de la façon dont l'optimiseur de requête évalue la requête. Il s'agit de code Microsoft, et nous n'avons aucune possibilité d'intervenir dessus.

- Assert
- Filtrer

Pour ce qui concerne les autres opérateurs, les décomptes ne sont pas touchés. Dans ces cas-là, une valeur zéro pour un Rebind peut signifier aussi bien qu'il n'y a eu aucun Rebind ou qu'il y en a eu. On sait uniquement que le décompte n'est pas tenu. Vous imaginez bien que c'est parfois un peu génant. Ceci explique également pourquoi c'est la valeur zéro qu'on voit le plus souvent pour les Rebind et les Rewind.

Cela dit, quelle est donc la signification d'une valeur de Rebind ou de Rewind pour un de ces huit opérateurs qui peuvent en incrémenter le décompte ?

Quand vous voyez un opérateur dont le Rebind vaut un et le Rewind vaut zéro, alors cela signifie qu'il y a eu un appel de la méthode Init() pour un opérateur physique qui n'est PAS du côté interne d'une boucle de jointure. Si l'opérateur physique est du côté interne de la boucle de jointure utilisée par un opérateur, alors la

somme des Rebinds et des Rewinds sera égale au nombre de lignes traitées du côté externe de la jointure utilisée par cet opérateur.

En quoi est-ce utile au DBA ? en règle générale, il vaut mieux que le nombre de Rebinds et de Rewinds soient les plus petits possibles, un nombre plus élevé indiquant plus de I/O disques. Des valeurs élevées peuvent signaler qu'un opérateur spécifique est sollicité plus qu'il ne conviendrait, ce qui nuit aux performance du serveur. dans un tel cas, il conviendrait de réécrire la requête, ou bien de modifier l'indexation actuelle, pour utiliser un plan différent, qui consommerait moins de Rebinds et de Rewinds, ce qui diminuerait les I/O disque et améliorerait les performances.

Les Plans d'Exécution pour Insert, Update, et Delete

Toute requête sur une base de données va créer un plan d'exécution qui permettra au moteur de requête de traiter au mieux la requête soumise. Dans les exemples précédents, nous avons examiné des requêtes pour **SELECT**, mais dans cette section, nous allons voir les plans d'exécution pour des requêtes **INSERT**, **UPDATE**, et **DELETE**.

Instruction Insert

Voici un INSERT tout simple :

```
INSERT  INTO [AdventureWorks].[Person].[Address]
        (
          [AddressLine1],
          [AddressLine2],
          [City],
          [StateProvinceID],
          [PostalCode],
          [rowguid],
          [ModifiedDate]
        )
VALUES  (
          '1313 Mockingbird Lane',
          'Basement',
          'Springfield',
          '79',
          '02134',
          NEWID(),
          GETDATE()
        ) ;
```

Cette instruction produit ce plan d'exécution estimé (pour ne pas modifier les données) assez intéressant, qu'on peut voir figure 29.

Figure 29

Toujours en lisant de droite à gauche, le plan d'exécution commence par un opérateur qui nous est nouveau : **Constant Scan**. Cet opérateur introduit un nombre constant de lignes dans une requête. Dans notre exemple,

il construit une ligne, pour permettre aux deux opérateurs suivants d'y ajouter leur sortie. Le premier de ces deux est un opérateur **Compute Scalar** qui invoque une fonction nommée GetIdentity. C'est à cette étape du plan d'exécution qu'est générée la valeur Identity pour les données qui suivront. Remarquez bien que c'est la première chose qui est faite dans le plan ; ceci nous aide à mieux comprendre pourquoi nous pouvons avoir des trous dans les valeurs d'identity d'une table, quand la suite d'une insertion échoue.

On trouve ensuite une autre opération scalaire, qui produit en sortie une série de paramètres qui seront utilisés par la suite, et qui crée une nouvelle valeur **d'UniqueIdentifier**, et qui récupère la date et l'heure par la fonction **GETDATE**. Tout ceci est envoyé à l'opérateur **Clustered Index Insert**, qui concentre la plus grosse partie du coût d'exécution de ce plan. Remarquez bien la valeur de sortie de cet INSERT, c'est **Person.Address.StateProvinceID**. cette valeur est passée à l'opérateur suivant, une jointure par **Boucles Imbriquées**, qui récupère aussi une entrée provenant d'un **Clustered Index Seek** sur la table **Person.StateProvince**. Autrement dit, pendant l'INSERT, nous avons une lecture qui vérifie l'intégrité référentielle sur la clé étrangère posée sur **StateProvinceID**. La jointure produit à ce moment-là une nouvelle expression, qui est vérifiée par l'opérateur suivant, **Assert**. Un **Assert** vérifie qu'une condition spécifiée existe. Celui-ci vérifie que la valeur de l'expression Expr1015 est égale à zéro. Énoncé de façon plus explicite, il vérifie que la donnée que nous essayons d'insérer dans le champ **Person.Adress.StateProvinceID** a une correspondance dans la table **Person.StateProvince** : c'est bien une vérification référentielle.

Instruction Update

Regardons l'instruction Update suivante :

```
UPDATE  [Person].[Address]
SET     [City] = 'Munro',
        [ModifiedDate] = GETDATE()
WHERE   [City] = 'Monroe' ;
```

Voici son plan d'exécution estimé :

Figure 30

Commençons la lecture de ce plan (toujours depuis la droite). Le premier opérateur est un Non-Clustered Index Scan, qui ramène toutes les lignes voulues depuis un index non-clustered, en le parcourant ligne à ligne. Ça n'est pas le plus efficace, et ceci devrait nous alerter sur un besoin probable d'un index qui accélérerait les performances [2]. Cet opérateur identifie toutes les lignes qui répondent à la clause **WHERE [City] = 'Monroe'**, et il les passe à l'opérateur suivant.

L'opérateur suivant est **HAUT**. Dans un plan d'exécution **UPDATE**, il sert à appliquer une limitation sur le nombre de lignes à impacter, si ce nombre est spécifié. Dans notre cas, il n'y a aucune restriction à appliquer puisque nous n'avons pas de clause **TOP** dans notre requête **UPDATE**.

Attention : si vous voyez un opérateur **HAUT** dans une instruction **SELECT** (pas un UPDATE), alors cela indique qu'un nombre ou un pourcentage spécifique de lignes a été retourné pour respecter la commande **TOP** de l'instruction **SELECT**.

[2] Note du Traducteur : cet article a été initialement rédigé à partir de SQL Server 2005. Les versions actuelles affichent directement le conseil de création de l'index manquant dans le pan d'exécution, comme on le voit sur la figure 30.

L'opérateur suivant est un **Eager Spool**, c'est une forme de Table Spool. Cet opérateur au nom étrange prend chaque ligne devant être mise à jour, et les stocke dans un objet temporaire masqué, dans la base de données TempDb. Si, dans la suite du plan d'exécution, un opérateur doit être rembobiné (Rewind) sans qu'une Reliaison (Rebind) soit nécessaire – par exemple lors de l'utilisation de Boucles Imbriquées – alors ces données pouront être réutilisées sans qu'il soit besoin de re-scanner les données à nouveau. Ceci permettra d'éviter de répéter un Clustered Index Scan, qui est une opération couteuse. Dans notre exemple, il n'y a pas besoin de rembobinage.

Les trois opérateurs suivants sont des opérateurs Compute Scalar, que nous avons déjà étudiés. Ici, ils servent à évaluer des expressions pour produire une valeur scalaire, comme nous l'avons vu dans l'utilisation de la fonction **GETDATE()**.

Nous arrivons maintenant au cœur de l'instruction **UPDATE**, l'opérateur **Clustered Index Update**. Dans notre exemple, les valeurs à mettre à jour font partie d'un index clustered. L'opérateur va donc identifier les lignes à mettre à jour, et procéder à cette mise à jour.

Et pour finir, nous trouvons l'opérateur passe-partout (un élément du langage T-SQL), qui nous informe que l'opération **UPDATE** est terminée.

Du point de vue des performances, un des points de vérification est la façon dont sont récupérées les lignes à mettre à jour. Dans notre exemple, on a fait un Non-Clustered Index Scan, très peu performant. Il aurait été bien meilleur de faire un Index Seek sur un index clustered ou non-clustered, qui auraient l'un comme l'autre utilisé moins d'I/O pour exécuter **l'UPDATE** demandé.

Instruction Delete

Quel peut bien être le plan d'exécution créé par une instruction **DELETE** ? pour le voir, lançons le code suivant, et examinons son plan d'exécution.

```
DELETE  FROM [Person].[Address]
WHERE   [AddressID] = 52;
```

Nous voyons le plan d'exécution estimé Figure 31 :

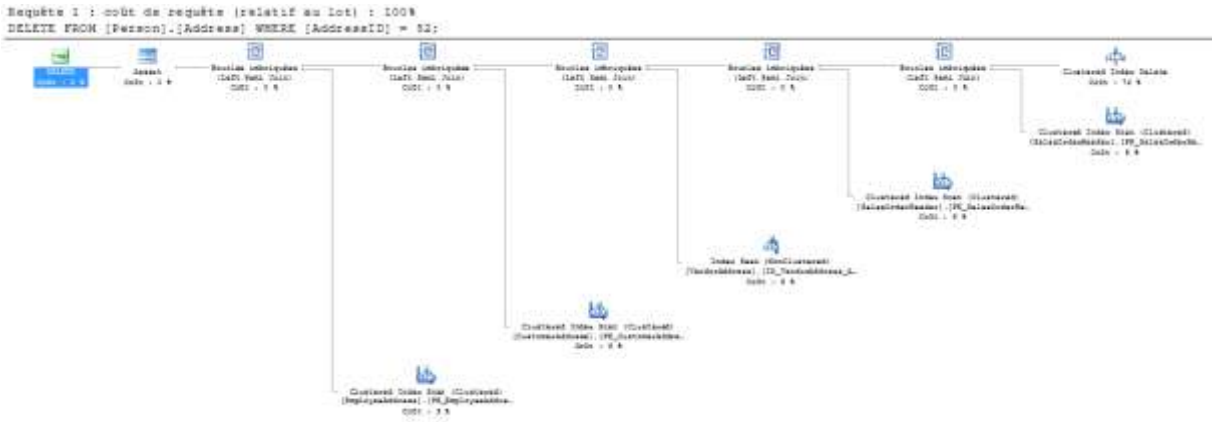


Figure 31

Je sais que c'est un peu difficile à lire, mais je voulais vraiment vous montrer la taille importante du plan d'exécution requis pour supprimer des données dans une base de données relationnelle. Souvenez-vous, la suppression d'une ou plusieurs lignes n'est pas un évènement isolé circonscrit à la table concernée. Il nous faut vérifier toutes les tables qui pointent sur la clé primaire de la table dans laquelle nous faisons la

suppression, afin de tester si la suppression de cette partie des données affecte leur intégrité. Pour sa plus grande part, ce plan ressemble plus à un **INSERT** qu'à un **DELETE**.

Commençons par la droite, et de haut en bas… et nous voyons tout de suite un opérateur Clustered Index Delete. Il y a deux points intéressants à souligner immédiatement : d'abord, il est bien de savoir que le traitement commence par la suppression. Et ensuite, en deuxième point, nous pouvons observer que le prédicat de recherche (Seek Predicate) de cette opération **Clustered Index Delete** était :

```
Prefix: [AdventureWorks].[Person].[Address].AddressID = Scalar Operator-
(CONVERT_IMPLICIT(int,[@1],0)).
```

Ceci indique que pour chercher **AdressId**, on a utilisé un paramètre **@1**. Or notre code ne comporte pas de paramètre, nous avons utilisé une constante de valeur 52. D'où vient donc ce paramètre ? voilà l'indication que le moteur de requête fabrique des plans de requête réutilisables, en observant les règles d'une paramétrisation simple.

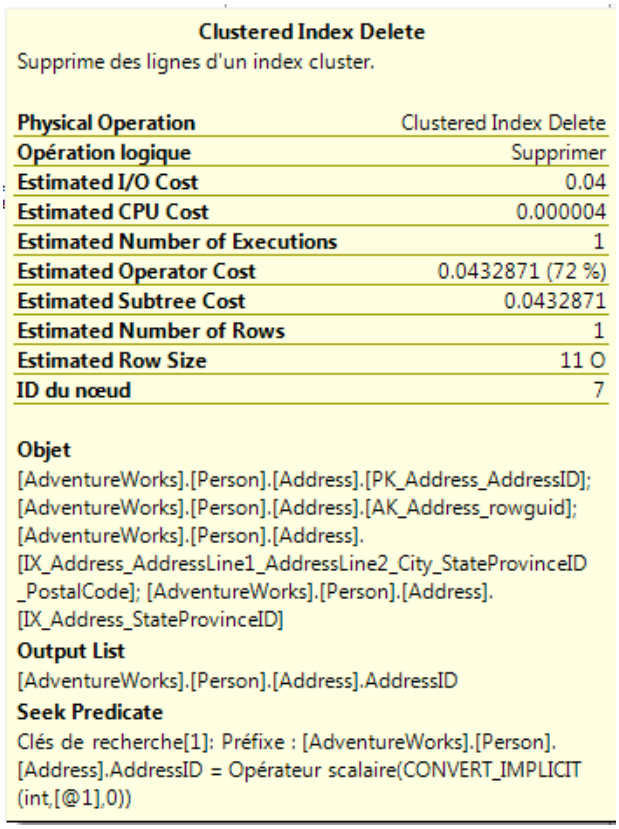
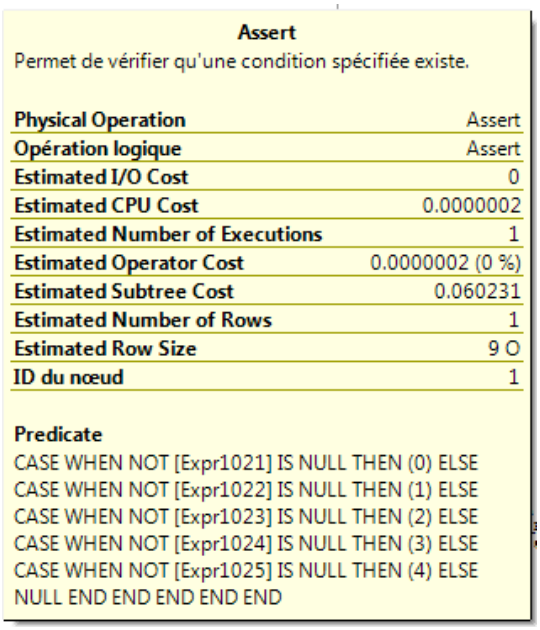


Figure 32

Après la suppression, on observe la combinaison d'une série d'Index Seek et de Clustered Index Scan par une série d'opérateurs de Boucles Imbriquées. Il s'agit très exactement de demi-jointures gauches. Ces opérateurs renvoient une valeur s'il y a une correspondance entre les deux tables sur le prédicat de jointure, ou bien s'il n'y a pas de prédicat de jointure. Chacun renvoie une valeur. Et à la fin, à la dernière étape, dans l'opérateur Assert, on vérifie toutes les valeur remontées par chaque jointure, toutes les tables liées à celle dans laquelle nous demandons une suppression, pour tester l'intégrité référentielle. S'il n'y a pas de retour, la suppression est effectuée. Si une valeur est renvoyée, alors une erreur sera générée et le **DELETE** sera annulé.



Assert

Permet de vérifier qu'une condition spécifiée existe.

Physical Operation	Assert
Opération logique	Assert
Estimated I/O Cost	0
Estimated CPU Cost	0.0000002
Estimated Number of Executions	1
Estimated Operator Cost	0.0000002 (0 %)
Estimated Subtree Cost	0.060231
Estimated Number of Rows	1
Estimated Row Size	9 O
ID du nœud	1

Predicate

CASE WHEN NOT [Expr1021] IS NULL THEN (0) ELSE
CASE WHEN NOT [Expr1022] IS NULL THEN (1) ELSE
CASE WHEN NOT [Expr1023] IS NULL THEN (2) ELSE
CASE WHEN NOT [Expr1024] IS NULL THEN (3) ELSE
CASE WHEN NOT [Expr1025] IS NULL THEN (4) ELSE
NULL END END END END END

Figure 33

En résumé…

Cet article vous permet d'apprendre à lire les plans d'exécution graphiques. Toutefois, comme nous l'annoncions au début de cet article, nous nous somme centrés sur les opérateurs les plus fréquents, et nous n'avons étudié que des requêtes simples. Par conséquent, n'imaginez pas que vous allez être capable d'analyser le plan d'exécution d'une requête de 200 lignes. L'apprentissage de la lecture et de l'analyse des plans d'exécution nécessite du temps et des efforts. Mais une fois que vous aurez acquis un peu d'expérience, vous verrez qu'il vous sera de plus en plus facile de lire et d'analyser les plans d'exécution aussi complexes soient-ils.

Traduction Michel Lévy, avec l'aimable autorisation de l'auteur et de l'éditeur