

Rencontre 11

Performance

Bases de données et programmation Web



- ❖ Monitorer la performance  
- ❖ Stratégies d'optimisation  



❖ 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.
 - Ex : la dénormalisation sert à éviter des jointures, au coût de dupliquer des données. Cela dit, l'espace disque coûte moins cher que la puissance de calcul.



❖ Monitorer la performance

◆ Bémol

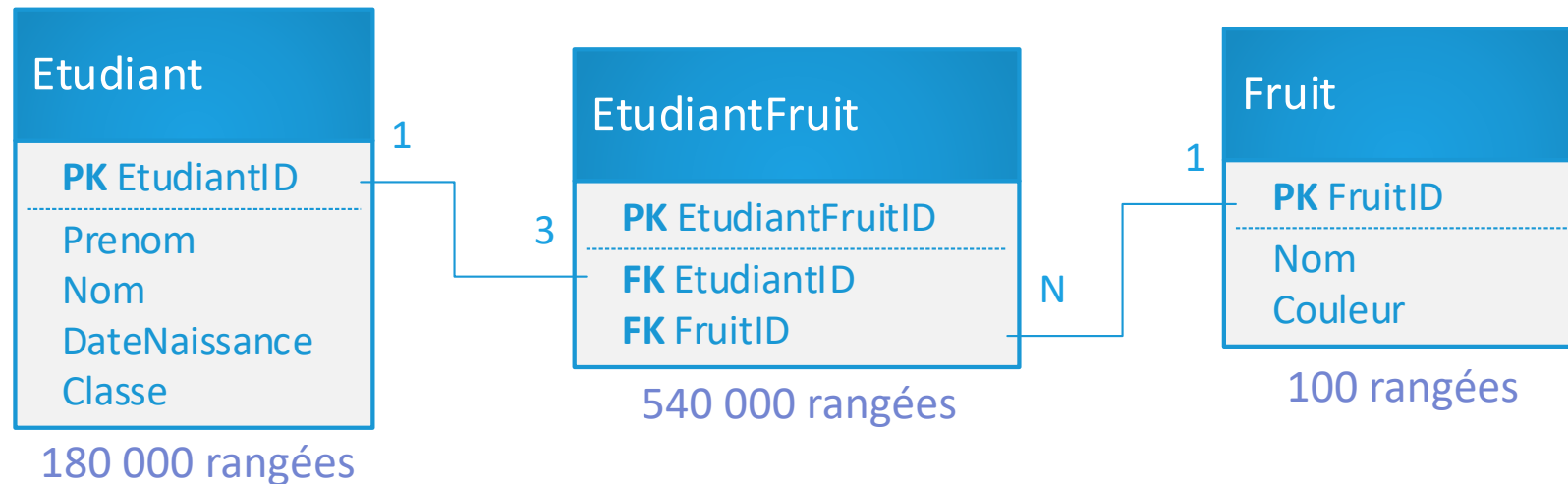
- Pour vraiment pouvoir conduire des tests intéressants sur la performance, avoir des Datasets de **plusieurs Go** est préférable. La variété et la richesse des tests que nous pourrons faire dans ce cours seront donc limitées.
 - Cela reste intéressant d'aborder des stratégies d'optimisation, ne serait-ce que théoriquement.



❖ Stratégies d'optimisation

◆ Dataset utilisé

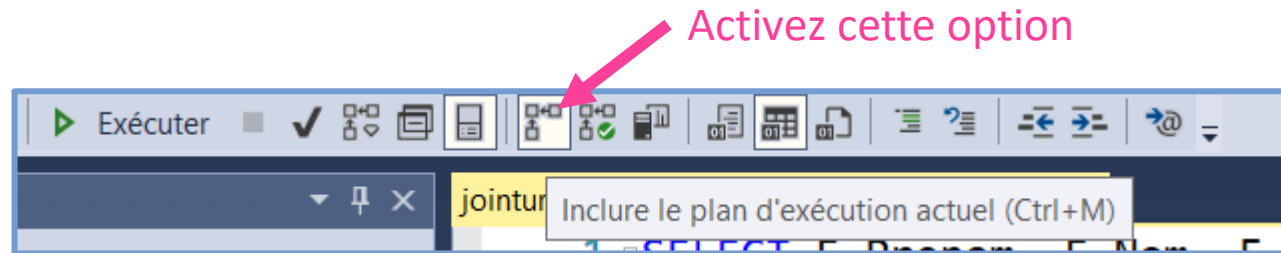
- Dans une école, on a demandé à 180 000 jeunes quels étaient leurs trois fruits préférés parmi une liste de 100 fruits.





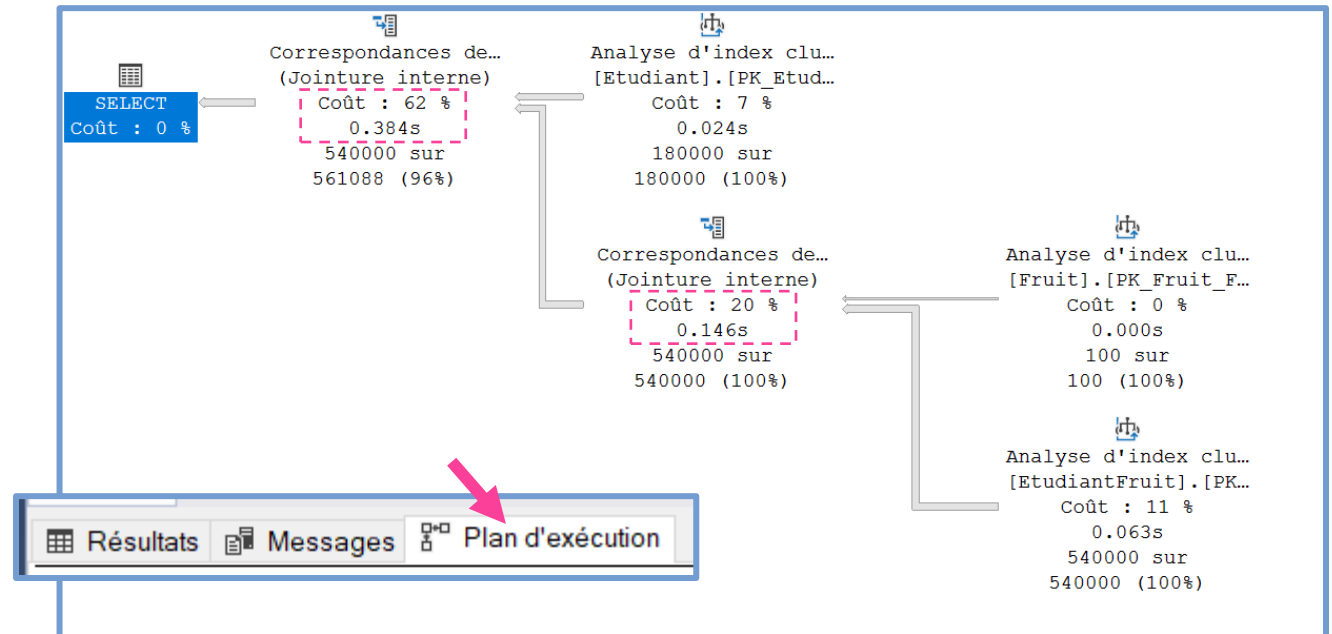
❖ Monitorer la performance

- ◆ Il existe plusieurs façons de monitorer la performance d'opérations SQL.
- ◆ Ici on voit le plan d'exécution actuel par SSMS, sous forme de graphique.



```
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

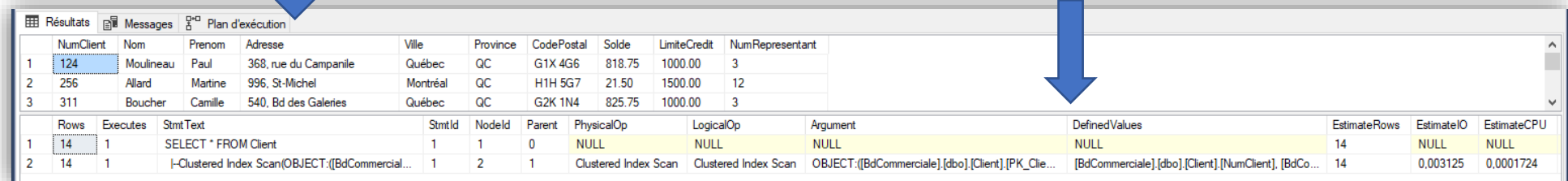
- Le plan d'exécution graphique est facile à lire, mais il est difficile de partager tous ces détails.
- La commande : **SET STATISTICS PROFILE ON**
 - ☐ permet de réaliser un plan d'exécution complet à chaque requête (en tableau).
 - ☐ Pour arrêter son effet, il faut utiliser la commande :

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***



The screenshot shows the SQL Server Enterprise Manager interface. The top pane displays the results of the query 'SELECT * FROM Client', showing a table with columns: NumClient, Nom, Prenom, Adresse, Ville, Province, CodePostal, Solde, LimiteCredit, and NumRepresentant. The bottom pane displays the execution plan for the same query, showing a 'Clustered Index Scan' operation. A blue arrow points from the text 'SELECT * FROM CLIENT' to the top pane, and another blue arrow points from the text 'Plan d'exécution en tableau' to the bottom pane.

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...	14	0.003125	0.0001724



❖ Stratégies et concepts liés à l'optimisation

◆ Il existe plusieurs stratégies simples:

- Index
- Partitions
- Cache
- Autres éléments de configuration

◆ Nous n'allons parler que des **index** ici.

- C'est la stratégie d'optimisation la plus simple qui donne le plus grand résultat rapidement.



❖ Stratégies d'optimisation

◆ Index


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



❖ Stratégies d'optimisation

- ◆ 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

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

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

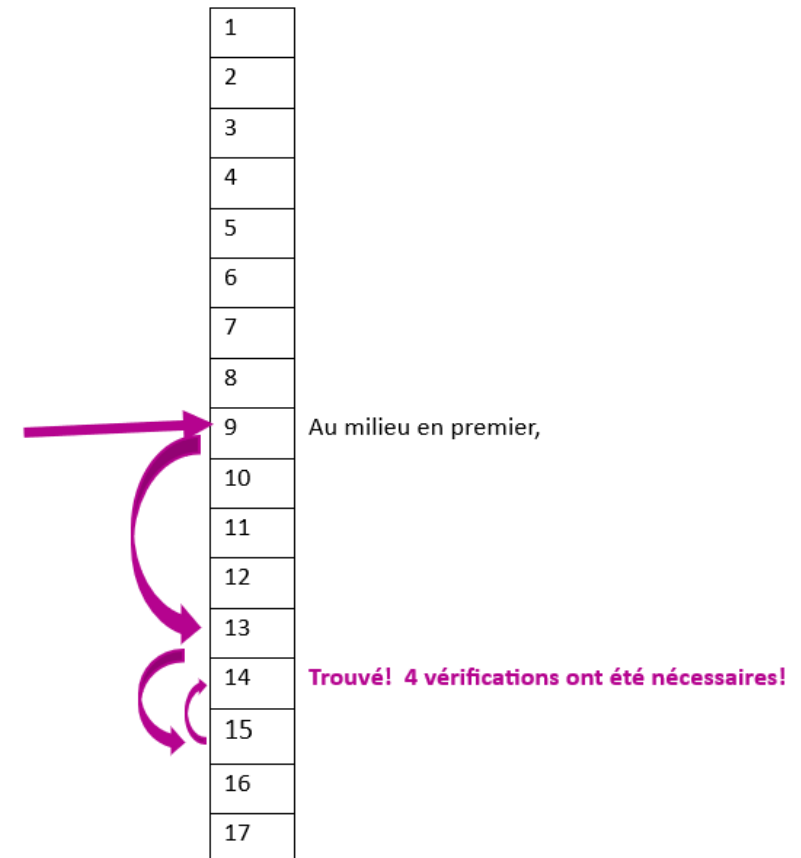
- on vérifie toujours la rangée au « **milieu** » de l'ensemble restant.
- on regarde si la valeur trouvée est plus petite que la valeur recherchée
- si oui, l'ensemble restant est réduit à la valeur trouvée et le restant qui suit....

9 est **trop petit** ? on élimine la moitié des rangées, soit celles avant 9.

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

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

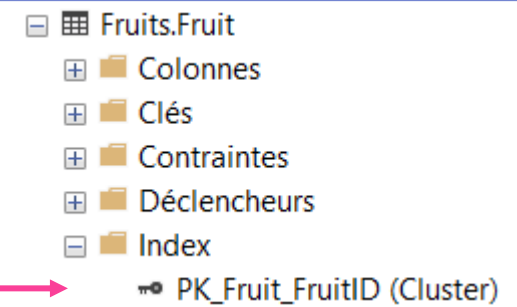
Finalement, on trouve 14.





❖ Les index

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



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

❖ Il existe deux types d'index:

◆ Index Clustered OU Non-Clustered

○ **Clustered** (max. 1 par table)

- Détermine l'ordre des rangées d'une table, **physiquement, sur le disque**.
- Ex : En créant la clé primaire sur le champ FruitID de la table Fruit, SQL serveur crée automatiquement un index clustered pour la table Fruit sur le champ FruitID. Dans ce cas, dans la mémoire du système, les enregistrements de la table seront rangées dans l'ordre ci-dessous.

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



❖ Stratégies d’optimisation

◆ Index : Clustered OU Non-Clustered

○ 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. Si plusieurs rangées dans la table ont la même valeur pour la colonne utilisée par l’index **non-clustered**, il y aura autant de pointeurs que de rangées pour cette valeur spécifique dans la structure de recherche. (Ex : **trois** fruits **jaune**, **trois** pointeurs pour **jaune**)
- Exemple : **Index non clustered** pour la colonne **couleur**

Index

IX_Fruit_Nom (Non unique, non cluster)

PK_Fruit_FruitID (Cluster)

Impact sur la performance

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

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

IX_Table_Colonne
(nom)

Schéma.Table(Colonne)

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** très diversifiées. (Pire cas : toutes les rangées ont la **même valeur** pour la colonne -> l'index ne sert à rien)
- Colonne **souvent utilisée** pour récupérer les données. (WHERE) Si la colonne n'est jamais utilisée pour chercher une rangée ... l'index est inutile.
- 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

- Colonne **auto-assignée** et **auto-incrémentée**. (IDENTITY) **For-mi-da-ble**. La BD n'a pas à constamment réordonner les rangées de la table ou de la structure dans la mémoire. Mais on n'a pas besoin de créer un index, SQL serveur va le faire pour nous! Un seul index **clustered**, au maximum. (Impossible d'ordonner physiquement les rangées de plusieurs manières...).
- Autant d'index **non-clustered** que nécessaire sur les champs d'une table, mais 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 intéressants**. On est constamment en train d'entretenir les index, mais rarement en train d'en profiter.
 - Ex : Table d'archives
- Table avec peu de **WRITE**, beaucoup de **READ** : index **très intéressants**. On profite constamment de la recherche accélérée et on a peu d'entretien à faire.
 - Ex : Table d'utilisateurs (Chaque personne ne se crée un compte qu'une seule fois et modifie rarement les données de son profil, mais ses données sont souvent utilisées pour l'authentification, les paiements, l'affichage d'éléments du profil, etc.)



❖ Stratégies d'optimisation

◆ Index : exemple d'accélération

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

EtudiantFruit

PK EtudiantFruitID

FK EtudiantID

FK FruitID

540 000 rangées


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




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


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


Sans index sur EtudiantID: 19 millisecondes



SELECT
Coût : 0 %



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

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%)