
Article

[Irène Mykhailova](#) · Mai 9, 2022 14m de lecture

Connaissez vos indexes

Cet article est le premier d'une série d'articles sur les indexes SQL.

Partie 1 - Découvrez vos indexes

[Qu'est-ce qu'un index, en fait ?](#)

Imaginez la dernière fois où vous êtes allé à la bibliothèque. En général, les livres y sont classés par sujet (puis par auteur et par titre), et chaque étagère comporte une étiquette avec un code décrivant le sujet de ses livres. Si vous voulez collectionner des livres d'un certain sujet, au lieu de traverser chaque allée et de lire la couverture intérieure de chaque livre, vous pouvez vous diriger directement vers l'étagère étiquetée avec le sujet désiré et choisir vos livres.

Un index SQL a la même fonction générale : améliorer les performances en donnant une référence rapide à la valeur des champs pour chaque ligne de la table.

La mise en place d'index est l'une des principales étapes de la préparation de vos classes pour une performance SQL optimale.

Dans cet article, nous allons examiner les questions suivantes :

1. Qu'est-ce qu'un index et pourquoi/quand dois-je l'utiliser ?
2. Quels types d'indexes existent et pour quels scénarios sont-ils parfaitement adaptés ?
3. Qu'est-ce qu'un index ?
4. Comment le créer ?

- Et si j'ai des index, qu'est-ce que j'en fais ?

Je vais me référer aux classes de notre schéma Sample. Celles-ci sont disponibles dans le stockage Github suivant, et elles sont également fournies dans l'espace de noms Samples dans les installations de Caché et Ensemble :

<https://github.com/intersystems/Samples-Data>

[Les principes de base](#)

Vous pouvez indexer chaque propriété persistante et chaque propriété qui peut être calculée de manière fiable à partir de données persistantes.

Disons que nous voulons indexer la propriété TaxID dans Sample.Company. Dans Studio ou Atelier, nous ajouterions ce qui suit à la définition de la classe :

```
Index TaxIDIdx On TaxID;
```

L'instruction SQL DDL équivalente ressemblerait à ceci :

```
CREATE INDEX TaxIDIdx ON Sample.Company (TaxID);
```

La structure globale de l'index par défaut est la suivante :

```
^Sample.CompanyI("TaxIDIdx ",<TaxIDValueAtRowID>,<RowID>) = ""
```

Notez qu'il y a moins d'index inférieurs à lire que de champs dans une globale de données typique.

Considérons la requête

```
SELECT Name,TaxID FROM Sample.Company WHERE TaxID = 'J7349'
```

C'est logiquement simple et le plan de requête pour l'exécution de cette requête le reflète :

Query Plan

Relative cost = 495.2

- Read index map Sample.Company.TaxIDIdx, using the given %SQLUPPER(TaxID), and looping on ID.
- For each row:
 - Read master map Sample.Company.IDKEY, using the given idkey value.
 - Output the row.

Ce plan indique essentiellement que nous vérifions l'index global pour les lignes avec la valeur TaxID donnée, puis nous nous référons à la globale de données ("carte principale") pour récupérer la ligne correspondante.

Considérons maintenant la même requête sans index sur TaxIDX. Le plan de requête résultant est, comme prévu, moins efficace :

Sans index, l'exécution de la requête sous-jacente d'IRIS repose sur la lecture en mémoire et l'application de la condition de la clause WHERE à chaque ligne de la table. Et comme nous ne nous attendons logiquement pas à ce qu'une société partage TaxID, nous faisons tout ce travail pour une seule ligne !

Bien sûr, avoir des indexes signifie avoir des données d'index et de ligne sur le disque. En fonction de ce sur quoi nous avons une condition et de la quantité de données que notre table contient, cela peut s'avérer avoir ses propres défis lorsque nous créons et alimentons un index.

Alors, quand ajoutons-nous un index à une propriété ?

Dans le cas général, nous avons fréquemment à remettre une propriété en état. Des exemples sont des informations d'identification telles que le SSN d'une personne ou un numéro de compte bancaire. Vous pouvez également considérer les dates de naissance ou les fonds d'un compte. Pour en revenir à Sample.Company, la classe bénéficierait peut-être de l'indexation de la propriété Revenu si nous voulions collecter des données sur les organisations à hauts revenus. À l'inverse, les propriétés sur lesquelles il est peu probable que nous remettions des conditions sont moins appropriées pour être indexées : disons un slogan ou une description d'entreprise.

Facile - sauf qu'il faut aussi considérer quel type d'index est le meilleur !

Types d'index

Il existe six principaux types d'index que je vais aborder ici : standard, bitmap, compound, collection, bitslice et data. Je vais également aborder brièvement les index iFind, qui sont basés sur les flux. Il y a des chevauchements possibles ici et nous avons déjà abordé les indexes standards avec l'exemple ci-dessus.

Je vais présenter des exemples sur la façon de créer des indexes dans votre définition de classe, mais l'ajout de nouveaux index à une classe est plus complexe que le simple ajout d'une ligne dans votre définition de classe. Nous aborderons des considérations supplémentaires dans la partie suivante.

Prenons l'exemple de Sample.Person. Notez que Person a une sous-classe Employee, ce qui sera utile pour comprendre certains exemples. Employee partage son stockage global de données avec Person, et tous les indexes de Person sont hérités par Employee - ce qui signifie qu'Employee utilise l'index global de Person pour ces indexes hérités.

Si vous n'êtes pas familier avec ces classes, voici un aperçu général de celles-ci : Person a les propriétés SSN, DOB, Name, Home (un objet d'adresse intégré contenant l'état et la ville), Office (également une adresse), et la collection de listes FavoriteColors. Employee a une propriété supplémentaire Salary (que j'ai moi-même définie).

Standard

[Index](#) DateIDX On DOB;

J'utilise ici le terme "standard" pour désigner les indexes qui stockent la valeur brute d'une propriété (par opposition à une représentation binaire). Si la valeur est une chaîne de caractères, elle sera stockée sous une certaine collation - celle de SQLUPPER par défaut.

Par rapport aux index bitmap ou bitslice, les indexes standard sont plus compréhensibles pour les humains et relativement faciles à maintenir. Nous avons un nœud global pour chaque ligne de la table.

Voici comment DateIDX est stocké au niveau global.

```
^Sample.PersonI("DateIDX",51274,100115)="Sample.Employee~; Date is 05/20/81
```

Notez que le premier index inférieur après le nom de l'index est la valeur de la date, le dernier index inférieur est l'ID de la personne ayant cette date de naissance, et la valeur stockée sur ce noeud global indique que cette personne est également membre de la sous-classe Sample.Employee. Si cette personne n'était membre d'aucune sous-classe, la valeur du noeud serait une chaîne vide.

Cette structure de base sera cohérente avec la plupart des indexes non binaires, où les indexes sur plus d'une propriété créent plus d'indexes inférieurs dans la globale, et où le fait d'avoir plus d'une valeur stockée au nœud produit un objet \$listbuild, par exemple :

```
^Package.Class(IndexName,IndexValue1,IndexValue2,IndexValue3,RowID) = $lb(SubClass,DataValue1,DataValue2)
```

[Bitmap - Une représentation binaire de l'ensemble des ID-codes correspondant à une valeur de propriété.](#)

[Index](#) HomeStateIDX On Home.State [Type = bitmap];

Les indexes bitmap sont stockés par valeur unique, contrairement aux indexes standard, qui sont stockés par ligne.

Pour aller plus loin dans l'exemple ci-dessus, disons que la personne avec l'ID 1 vit dans le Massachusetts, avec l'ID 2 à New York, avec l'ID 3 dans le Massachusetts et avec l'ID 4 à Rhode Island. HomeStateIDX est essentiellement stocké comme suit :

(...)	0	0	0	0	-
MA	1	0	1	0	-
NY	0	1	0	0	-
RI	0	0	0	1	-

(...)	0	0	0	0	-
-------	---	---	---	---	---

Si nous voulions qu'une requête renvoie les données des personnes vivant en Nouvelle-Angleterre, le système effectue un bitwise OR sur les lignes pertinentes de l'index bitmap. On voit rapidement que nous devons charger en mémoire des objets Personne avec les ID 1, 3 et 4 au minimum.

Les bitmaps peuvent être efficaces pour les opérateurs AND, RANGE et OR dans vos clauses WHERE.

Bien qu'il n'y ait pas de limite officielle au nombre de valeurs uniques que vous pouvez avoir pour une propriété avant qu'un index bitmap soit moins efficace qu'un index standard, la règle générale est d'environ 10 000 valeurs distinctes. Ainsi, si un index bitmap peut être efficace pour un état des États-Unis, un index bitmap pour une ville ou un comté des États-Unis ne serait pas aussi utile.

Un autre concept à prendre en compte est l'efficacité du stockage. Si vous prévoyez d'ajouter et de supprimer fréquemment des lignes de votre table, le stockage de votre index bitmap peut devenir moins efficace. Prenons l'exemple ci-dessus : supposons que nous ayons supprimé de nombreuses lignes pour une raison quelconque et que notre table ne contienne plus de personnes vivant dans des états moins peuplés tels que le Wyoming ou le Dakota du Nord. Le bitmap comporte donc plusieurs lignes contenant uniquement des zéros. D'un autre côté, la création de nouvelles lignes dans les grandes tables peut finir par devenir plus lente, car le stockage bitmap doit accueillir un plus grand nombre de valeurs uniques.

Dans ces exemples, j'ai environ 150 000 lignes dans Sample.Person. Chaque nœud global stocke jusqu'à 64 000 ID, de sorte que l'index bitmap global à la valeur MA est divisé en trois parties :

```
^Sample.Person("HomeStateIDX", " MA", 1)=$zwc(135,7992)$c(0,(...))
```

```
    ^Sample.Person("HomeStateIDX", " MA", 2)=$zwc(404,7990,(...))
```

```
    ^Sample.Person("HomeStateIDX", " MA", 3)=$zwc(132,2744)$c(0,(...))
```

Cas particulier : Bitmap étendu

Un bitmap étendue, souvent appelé `$(ClassName)`, est un index bitmap sur les ID d'une classe - cela donne à IRIS un moyen rapide de savoir si une ligne existe et peut être utile pour les requêtes COUNT ou les requêtes sur les sous-classes. Ces indexes sont générés automatiquement lorsqu'un index bitmap est ajouté à la classe ; vous pouvez également créer manuellement un index bitmap d'étendue dans une définition de classe comme suit :

```
Index Company [ Extent, SqlName = "$Company", Type = bitmap ];
```

Ou via le mot-clé DDL appelé BITMAPEXTENT :

```
CREATE BITMAPEXTENT INDEX "$Company" ON TABLE Sample.Company
```

[Composés - Les indexes basés sur deux ou plusieurs propriétés](#)

```
Index OfficeAddrIDX On (Office.City, Office.State);
```

Le cas général d'utilisation des index composés est le conditionnement de requêtes fréquentes sur deux propriétés ou plus.

L'ordre des propriétés dans un index composé est important en raison de la manière dont l'index est stocké au niveau global. Le fait d'avoir la propriété la plus sélective en premier est plus efficace en termes de performances car cela permet d'économiser les lectures initiales du disque de l'index global ; dans cet exemple, Office.City est en premier car il y a plus de villes uniques que d'états aux États-Unis.

Le fait d'avoir une propriété moins sélective en premier est plus efficace en termes d'espace. En termes de

structure globale, l'arbre d'indexation serait plus équilibré si State était placé en premier. Pensez-y : chaque état contient de nombreuses villes, mais certains noms de ville n'appartiennent qu'à un seul état.

Vous pouvez également vous demander si vous vous attendez à exécuter des requêtes fréquentes ne conditionnant qu'une seule de ces propriétés - cela peut vous éviter de définir un autre index.

Voici un exemple de la structure globale des indexes composés :

```
^Sample.Person("OfficeAddrIDX"," BOSTON"," MA",100115)="Sample.Employee"
```

Commentaires : Index composé ou index bitmap ?

Pour les requêtes comportant des conditions sur plusieurs propriétés, vous pouvez également vous demander si des indexes bitmap séparés seraient plus efficaces qu'un seul index composé.

Les opérations par bit sur deux indexes différents peuvent être plus efficaces à condition que les indexes bitmap conviennent à chaque propriété.

Il est également possible d'avoir des indexes bitmap composés, c'est-à-dire des indexes bitmap dont la valeur unique est l'intersection de plusieurs propriétés sur lesquelles vous effectuez l'indexation. Considérez la table donnée dans la section précédente, mais au lieu des états, nous avons toutes les paires possibles d'un état et d'une ville (par exemple, Boston, MA, Cambridge, MA, même Los Angeles, MA, etc.), et les cellules obtiennent des 1 pour les lignes qui adhèrent aux deux valeurs.

[Collection - Les index basés sur les propriétés de la collection](#)

Nous avons ici la propriété FavoriteColors définie comme suit :

Property FavoriteColors As list Of %String;

Avec chacun des indexes suivants définis à titre de démonstration :

```
Index fclDX1 On FavoriteColors(ELEMENTS);  
Index fclDX2 On FavoriteColors(KEYS);
```

J'utilise ici le terme "collection" pour désigner plus largement les propriétés à cellule unique contenant plus d'une valeur. Les propriétés List Of et Array Of sont pertinentes ici, et si vous le souhaitez, même les chaînes de caractères délimitées.

Les propriétés de la collection sont automatiquement analysées pour construire leurs indexes. Pour les propriétés délimitées, comme un numéro de téléphone, vous devez définir cette méthode, <PropertyName>BuildValueArray(value, .valueArray), explicitement.

Compte tenu de l'exemple ci-dessus pour FavoriteColors, fclDX1 ressemblerait à ceci pour une personne dont les couleurs préférées sont le bleu et le blanc :

```
^Sample.Person("fclDX1"," BLUE",100115)="Sample.Employee"
```

(...)

```
^Sample.Person("fclDX1"," WHITE",100115)="Sample.Employee"
```

fclDX2 ressemblerait à :

```
^Sample.Person("fclDX2",1,100115)="Sample.Employee"
```

```
^Sample.Person("fclDX2",2,100115)="Sample.Employee"
```

Dans ce cas, puisque FavoriteColors est une collection de listes, un index basé sur ses clés est moins utile qu'un index basé sur ses éléments.

Veuillez vous référer à notre [documentation](#) pour des considérations plus approfondies sur la création et la gestion des indexes sur les propriétés des collections.

[Bitslice - Représentation en bitmap de la représentation en chaîne de bits des données numériques](#)

Index SalaryIDX On Salary [Type = bitslice]; //In Sample.Employee

Contrairement aux indexes bitmap, qui contiennent des balises indiquant quelles lignes contiennent une valeur spécifique, les indexes bitslice convertissent d'abord les valeurs numériques de la décimale à la binaire, puis créent un bitmap sur chaque chiffre de la valeur binaire.

Reprenons l'exemple ci-dessus et, par souci de réalisme, simplifions le salaire en unités de 1 000 dollars. Ainsi, si le salaire d'un employé est enregistré sous la forme 65, il est compris comme représentant 65 000 dollars.

Disons que nous avons un employé avec l'ID 1 qui a un salaire de 15, l'ID 2 un salaire de 40, l'ID 3 un salaire de 64 et l'ID 4 un salaire de 130. Les valeurs binaires correspondantes sont :

	0	0	0	0	1	1	1	
	0	0	1	0	1	0	0	
	0	1	0	0	0	0	0	
	1	0	0	0	0	0	1	

Notre chaîne de bits s'étend sur 8 chiffres. La représentation bitmap correspondante - les valeurs d'indexes bitslice - est essentiellement stockée comme suit :

^Sample.Personl("SalaryIDX",1,1) = "1000" ; La ligne 1 a une valeur à la place 1

^Sample.Personl("SalaryIDX",2,1) = "1001" ; Les lignes 1 et 4 ont des valeurs à la place 2

^Sample.Personl("SalaryIDX",3,1) = "1000" ; La ligne 1 a une valeur à la place 4

^Sample.Personl("SalaryIDX",4,1) = "1100" ; Les lignes 1 et 2 ont des valeurs à la place 8

^Sample.Personl("SalaryIDX",5,1) = "0000" ; etc...

^Sample.Personl("SalaryIDX",6,1) = "0100"

^Sample.Personl("SalaryIDX",7,1) = "0010"

^Sample.Personl("SalaryIDX",8,1) = "0001"

Notez que les opérations modifiant Sample.Employee ou les salaires dans ses lignes, c'est-à-dire les INSERTs, UPDATEs et DELETEs, nécessitent maintenant la mise à jour de chacun de ces nœuds globaux, ou bitslices. L'ajout d'un index bitslice à plusieurs propriétés d'une table ou à une propriété fréquemment modifiée peut présenter des risques pour les performances. En général, la maintenance d'un index bitslice est plus coûteuse que celle des indexes standard ou bitmap.

Les indexes Bitslice sont hautement spécialisés et ont donc des cas d'utilisation spécifiques : les requêtes qui doivent effectuer des calculs agrégés, par exemple SUM, COUNT ou AVG.

En outre, ils ne peuvent être utilisés efficacement que sur des valeurs numériques - les chaînes de caractères sont

converties en un 0 binaire.

Notez que si la table de données, et non les index, doit être lu pour vérifier la condition d'une requête, les indexes bitslice ne seront pas choisis pour exécuter la requête. Supposons que Sample.Person ne possède pas d'index sur Name. Si nous calculions le salaire moyen des employés portant le nom de famille Smith :

```
SELECT AVG(Salary) FROM Sample.Employee WHERE Name %STARTSWITH 'Smith,'
```

nous aurions besoin de lire des lignes de données pour appliquer la condition WHERE, et donc l'index bitslice ne serait pas utilisé en pratique.

Des problèmes de stockage similaires se posent pour les indexes bitslice et bitmap sur les tables où des lignes sont fréquemment créées ou supprimées.

[Data - Index dont les données sont stockées dans leurs nœuds globaux.](#)

```
Index QuickSearchIDX On Name [ Data = (SSN, DOB, Name) ];
```

Dans plusieurs des exemples précédents, vous avez peut-être observé la chaîne " Sample.Employee~" stockée comme valeur au niveau du nœud lui-même. Rappelez-vous que Sample.Employee hérite des indexes de Sample.Person. Lorsque nous effectuons une requête sur les employés en particulier, nous lisons la valeur aux nœuds d'index correspondant à notre condition de propriété pour vérifier que ladite personne est également un employé.

On peut aussi définir explicitement les valeurs à stocker. Le fait d'avoir des données définies au niveau des nœuds globaux de l'index permet d'éviter la lecture de l'ensemble des données globales ; cela peut être utile pour les requêtes sélectives ou les requêtes ordonnées fréquentes.

Considérons l'index ci-dessus comme un exemple. Si nous voulions extraire des informations d'identification sur une personne à partir de tout ou une partie de son nom (par exemple, pour rechercher des informations sur les clients dans une application de réception), nous pourrions avoir une requête telle que

```
SELECT SSN, Name, DOB FROM Sample.Person WHERE Name %STARTSWITH 'Smith,J' ORDER BY Name
```

Puisque les conditions de notre requête sur le nom et les valeurs que nous récupérons sont toutes contenues dans les nœuds globaux QuickSearchIDX, il nous suffit de lire notre I globale pour exécuter cette requête.

Notez que les valeurs de données ne peuvent pas être stockées avec des indexes de bitmap ou de bitslice.

```
^Sample.PersonI("QuickSearchIDX", " LARSON,KIRSTEN  
A.",100115)=$Ib("Sample.Employee~",555-55-5555",51274,"Larson,Kirsten A.")
```

[iFind Indexes](#)

Vous en avez déjà entendu parler ? Moi non plus. Les indexes iFind sont utilisés sur les propriétés des flux, mais pour les utiliser vous devez spécifier leurs noms avec des mots-clés dans la requête.

Je pourrais vous en dire plus, mais Kyle Baxter a déjà rédigé un [article utile à ce sujet](#).

[#Indexation](#) [#Performances](#) [#SQL](#) [#Caché](#) [#InterSystems](#) [IRIS](#)