Article
Lorenzo Scalese · Mai 16, 2022 11m de lecture

Création d'index personalisé dans Caché

Les modèles de données objet et relationnel de la base de données Caché supportent trois types d'index, à savoir standard, bitmap et bitslice. En plus de ces trois types natifs, les développeurs peuvent déclarer leurs propres types d'index personnalisés et les utiliser dans toutes les classes depuis la version 2013.1. Par exemple, les index de texte iFind utilisent ce mécanisme.

Un Custom Index Type est une classe qui implémente les méthodes de l'interface *%Library.FunctionalIndex* pour effectuer des insertions, des mises à jour et des suppressions. Vous pouvez spécifier une telle classe comme type d'index lorsque vous déclarez un nouvel index.

Exemple:

```
Property A As %String;
Property B As %String;
Index someind On (A,B) As CustomPackage.CustomIndex;
```

La classe *CustomPackage.CustomIndex* est la classe même qui implémente les index personnalisés.

Par exemple, analysons le petit prototype d'un index à base de quadtrees pour les données spatiales qui a été développé pendant le Hackathon par notre équipe : Andrey Rechitsky, Aleksander Pogrebnikov et moi-même. (Le Hackathon a été organisé dans le cadre de la formation annuelle de l'école d'innovation d'InterSystems Russie, et nous remercions tout particulièrement le principal inspirateur du Hackathon, Timur Safin.)

Dans cet article, je ne vais pas parler des [quadtrees] (https://en.wikipedia.org/wiki/Quadtree) et de la façon de les utiliser. Nous allons plutôt examiner comment créer une nouvelle classe qui implémente l'interface *%Library.FunctionalIndex* pour l'implémentation de l'algorithme quadtree existant. Dans notre équipe, cette tâche a été confiée à Andrey. Andrey a créé la classe *SpatialIndex.Indexer* avec deux méthodes :

- Insert(x, y, id)
- Delete(x, y, id)

Lors de la création d'une nouvelle instance de la classe *SpatialIndex.Indexer*, il était nécessaire de définir un nom de nœud global dans lequel nous stockons les données d'index. Tout ce que j'avais à faire était de créer la classe *SpatialIndex.Index* avec les méthodes *InsertIndex*, *UpdateIndex*, *DeleteIndex* et *PurgeIndex*. Les trois premières méthodes acceptent l'*Id* de la chaîne à modifier et les valeurs indexées exactement dans le même ordre que celui dans lequel elles ont été définies dans la déclaration de l'index au sein de la classe correspondante. Dans notre exemple, les arguments d'entrée sont *pArg(1) — A* and *pArg(2) — B*.

```
Class SpatialIndex.Index Extends %Library.FunctionalIndex [ System = 3 ]
{

ClassMethod InsertIndex(pID As %CacheString, pArg... As %Binary) [ CodeMode = generat
or, ServerOnly = 1 ]
{
    if %mode'="method" {
        set IndexGlobal = ..IndexLocation(%class,%property)
```

```
        $$$GENERATE($C(9)_"set indexer = ##class(SpatialIndex.Indexer).%New($Name("_I
ndexGlobal_"))")
        $$$GENERATE($C(9)_"do indexer.Insert(pArg(1),pArg(2),pID)")
    }
}

ClassMethod UpdateIndex(pID As %CacheString, pArg... As %Binary) [ CodeMode = generat
or, ServerOnly = 1 ]
{
    if %mode'="method" {
        set IndexGlobal = ..IndexLocation(%class,%property)
        $$$GENERATE($C(9)_"set indexer = ##class(SpatialIndex.Indexer).%New($Name("_I
ndexGlobal_"))")
        $$$GENERATE($C(9)_"do indexer.Delete(pArg(3),pArg(4),pID)")
        $$$GENERATE($C(9)_"do indexer.Insert(pArg(1),pArg(2),pID)")
    }
}
ClassMethod DeleteIndex(pID As %CacheString, pArg... As %Binary) [ CodeMode = generat
or, ServerOnly = 1 ]
{
    if %mode'="method" {
        set IndexGlobal = ..IndexLocation(%class,%property)
        $$$GENERATE($C(9)_"set indexer = ##class(SpatialIndex.Indexer).%New($Name("_I
ndexGlobal_"))")
        $$$GENERATE($C(9)_"do indexer.Delete(pArg(1),pArg(2),pID)")
    }
}

ClassMethod PurgeIndex() [ CodeMode = generator, ServerOnly = 1 ]
{
    if %mode'="method" {
        set IndexGlobal = ..IndexLocation(%class,%property)
        $$$GENERATE($C(9)_"kill " _ IndexGlobal)
    }
}

ClassMethod IndexLocation(className As %String, indexName As %String) As %String
{
    set storage = ##class(%Dictionary.ClassDefinition).%OpenId(className).Storages.Ge
tAt(1).IndexLocation
    quit $Name(@storage@(indexName))
}

}
```

IndexLocation est une méthode supplémentaire qui renvoie le nom du nœud dans le global où la valeur de l'index est enregistrée.

Analysons maintenant la classe de test dans laquelle l'index du type *SpatialIndex.Index* est utilisé :

```
Class SpatialIndex.Test Extends %Persistent
{
  Property Name As %String(MAXLEN = 300);
  Property Latitude As %String;
  Property Longitude As %String;
  Index coord On (Latitude, Longitude) As SpatialIndex.Index;
}
```

Lorsque la classe *SpatialIndex.Test* est compilée, le système génère les méthodes suivantes dans le code INT pour chaque index du type *SpatialIndex.Index* :

```
zcoordInsertIndex(pID,pArg...) public {
    set indexer = ##class(SpatialIndex.Indexer).%New($Name(^SpatialIndex.TestI("coord
")))
    do indexer.Insert(pArg(1),pArg(2),pID) }
zcoordPurgeIndex() public {
    kill ^SpatialIndex.TestI("coord") }
zcoordSegmentInsert(pIndexBuffer,pID,pArg...) public {
    do ..coordInsertIndex(pID, pArg...) }
zcoordUpdateIndex(pID,pArg...) public {
    set indexer = ##class(SpatialIndex.Indexer).%New($Name(^SpatialIndex.TestI("coord
")))
    do indexer.Delete(pArg(3),pArg(4),pID)
    do indexer.Insert(pArg(1),pArg(2),pID)
}
```

Les méthodes *%SaveData*, *%DeleteData*, *%SQLInsert*, *%SQLUpdate* et *%SQLDelete* appellent les méthodes de notre index. Par exemple, le code suivant fait partie de la méthode *%SaveData* :

```
if insert {
     ...
     do ..coordInsertIndex(id,i%Latitude,i%Longitude,"")
      ...
 } else {
      ...
     do ..coordUpdateIndex(id,i%Latitude,i%Longitude,zzc27v3,zzc27v2,"")
      ...
 }
```

Un exemple pratique est toujours mieux que la théorie, vous pouvez donc télécharger les fichiers depuis notre entrepôt : https://github.com/intersystems-ru/spatialindex/tree/no-web-interface. Ceci est un lien vers une branche sans l'interface web. Pour utiliser ce code :

1. Importez les classes
2. Décompresser RuCut.zip
3. Importez les données en utilisant les appels suivants :

```
do $system.OBJ.LoadDir("c:\temp\spatialindex","ck")
do ##class(SpatialIndex.Test).load("c:\temp\rucut.txt")
```

Le fichier rucut.txt contient des données sur 100 000 villes et villages de Russie, avec leur nom et leurs coordonnées. La méthode Load lit chaque chaîne de caractères du fichier, puis l'enregistre comme une instance distincte de la classe SpatialIndex.Test. Une fois la méthode Load exécutée, le fichier global ^SpatialIndex.TestI("coord") contient un quadtree avec les coordonnées de latitude et de longitude.

Et maintenant, exécutons des requêtes !

La construction des index n'est pas la partie la plus intéressante. Nous voulons utiliser notre index dans diverses

requêtes. Dans Caché, il existe une syntaxe standard pour les index non standard :

```
SELECT *
FROM SpatialIndex.Test
WHERE %ID %FIND search_index(coord, 'window', 'minx=56,miny=56,maxx=57,maxy=57')
```

%ID %FIND searchindex est la partie fixe de la syntaxe. Ensuite, il y a le nom de l'index, *coord* - et notez qu'aucun guillemet n'est nécessaire. Tous les autres paramètres *('window', 'minx=56,miny=56,maxx=57,maxy=57')* sont transmis à la méthode Find, qui doit également être définie dans la classe du type d'index (qui, dans notre exemple, est *SpatialIndex.Index*) :

```
ClassMethod Find(queryType As %Binary, queryParams As %String) As %Library.Binary [ C
odeMode = generator, ServerOnly = 1, SqlProc ]
{
    if %mode'="method" {
        set IndexGlobal = ..IndexLocation(%class,%property)
        set IndexGlobalQ = $$$QUOTE(IndexGlobal)
        $$$GENERATE($C(9)_"set result = ##class(SpatialIndex.SQLResult).%New()")
        $$$GENERATE($C(9)_"do result.PrepareFind($Name("_IndexGlobal_"), queryType, q
ueryParams)")
        $$$GENERATE($C(9)_"quit result")
    }
}
```

Dans cet exemple de code, nous avons seulement deux paramètres - *queryType* et *queryParams*, mais vous pouvez ajouter autant de paramètres que vous le souhaitez.

Lorsque vous compilez une classe dans laquelle la méthode *SpatialIndex.Index* est utilisée, la méthode *Find* génère une méthode supplémentaire appelée *z<IndexName>Find*, qui est ensuite utilisée pour exécuter des requêtes SQL :

```
zcoordFind(queryType,queryParams) public { Set:'$isobject($get(%sqlcontext)) %sqlcont
ext=##class(%Library.ProcedureContext).%New()
    set result = ##class(SpatialIndex.SQLResult).%New()
    do result.PrepareFind($Name(^SpatialIndex.TestI("coord")), queryType, queryParams
)
    quit result }
```

La méthode *Find* doit retourner une instance de la classe qui implémente l'interface *%SQL.AbstractFind*. Les méthodes de cette interface, *NextChunk* et *PreviousChunk*, renvoient des chaînes de bits par tranches de 64 000 bits chacune. Lorsqu'un enregistrement avec un certain *ID* répond aux critères de sélection, le bit correspondant (chunknumber * 64000 + positionnumberwithinchunk) est mis à 1.

```
Class SpatialIndex.SQLResult Extends %SQL.AbstractFind
{

Property ResultBits [ MultiDimensional, Private ];

Method %OnNew() As %Status [ Private, ServerOnly = 1 ]
{
    kill i%ResultBits
    kill qHandle
```

```
    quit $$$OK
}


Method PrepareFind(indexGlobal As %String, queryType As %String, queryParams As %Bina
ry) As %Status
{
    if queryType = "window" {
        for i = 1:1:4 {
            set item = $Piece(queryParams, ",", i)
            set IndexGlobal = ..IndexLocation(%class,%property)
            $$$GENERATE($C(9)_"kill " _ IndexGlobal)   set param = $Piece(item, "=",
1)
            set value = $Piece(item, "=" ,2)
            set arg(param) = value
        }
        set qHandle("indexGlobal") = indexGlobal
        do ##class(SpatialIndex.QueryExecutor).InternalFindWindow(.qHandle,arg("minx"
),arg("miny"),arg("maxx"),arg("maxy"))
        set id = ""
        for  {
            set id = $O(qHandle("data", id),1,idd)
            quit:id=""
            set tChunk = (idd\64000)+1, tPos=(idd#64000)+1
            set $BIT(i%ResultBits(tChunk),tPos) = 1
        }
    }
    quit $$$OK
}

Method ContainsItem(pItem As %String) As %Boolean
{
    set tChunk = (pItem\64000)+1, tPos=(pItem#64000)+1
    quit $bit($get(i%ResultBits(tChunk)),tPos)
}

Method GetChunk(pChunk As %Integer) As %Binary
{
    quit $get(i%ResultBits(pChunk))
}

Method NextChunk(ByRef pChunk As %Integer = "") As %Binary
{
    set pChunk = $order(i%ResultBits(pChunk),1,tBits)
    quit:pChunk="" ""
    quit tBits
}

Method PreviousChunk(ByRef pChunk As %Integer = "") As %Binary
{
    set pChunk = $order(i%ResultBits(pChunk),-1,tBits)
    quit:pChunk="" ""
    quit tBits
}
}
```

Comme le montre l'exemple de code ci-dessus, la méthode *InternalFindWindow* de la classe *SpatialIndex.QueryExecutor* recherche les points situés dans le rectangle spécifié. Ensuite, les ID des lignes

correspondantes sont écrits dans les bitsets dans la boucle FOR.

Dans notre projet Hackathon, Andrey a également implémenté la fonctionnalité de recherche pour les ellipses :

```
SELECT *
FROM SpatialIndex.Test
WHERE %ID %FIND search_index(coord,'radius','x=55,y=55,radiusX=2,radiusY=2')
and name %StartsWith 'Z'
```

Un peu plus à propos de %FIND

Le prédicat *%FIND* possède un paramètre supplémentaire, *SIZE*, qui aide le moteur SQL à estimer le nombre de lignes correspondantes. En fonction de ce paramètre, le moteur SQL décide d'utiliser ou non l'index spécifié dans le prédicat *%FIND*.

Par exemple, ajoutons l'index suivant dans la classe SpatialIndex.Test :

```
Index ByName on Name;
```

Maintenant, recompilons la classe et construisons cet index :

```
write ##class(SpatialIndex.Test).%BuildIndices($LB("ByName"))
```

Et enfin, lancez TuneTable :

```
do $system.SQL.TuneTable("SpatialIndex.Test", 1)
```

Voici le plan de la requête :

```
SELECT *
FROM SpatialIndex.Test
WHERE name %startswith 'za'
and %ID %FIND search_index(coord,'radius','x=55,y=55,radiusX=2,radiusY=2') size ((10)
)
```

Query Text

```
SELECT * FROM SpatialIndex . Test WHERE name %STARTSWITH ? AND %ID %FIND search_index ( coord , ? , ? ) size ( ( 10 ) )
```

Query Plan

Relative cost = 2480

- Read given bitmap filter for SpatialIndex.Test.%ID, looping on ID.
- For each row:

  Read master map SpatialIndex.Test.IDKEY, using the given idkey value.
  Output the row.

Comme l'index coord est susceptible de retourner peu de lignes, le moteur SQL n'utilise pas l'index sur la propriété

Name.

Il y a un plan différent pour la requête suivante :

```
SELECT *
FROM SpatialIndex.Test
WHERE name %startswith 'za'
and %ID %FIND search_index(coord,'radius','x=55,y=55,radiusX=2,radiusY=2') size ((100
0))
```

Le moteur SQL utilise les deux index pour exécuter cette requête.

Et, comme dernier exemple, créons une requête qui utilise uniquement l'index sur le champ *Name*, puisque l'index *coord* renverra probablement environ 100 000 lignes et sera donc très peu utilisable :

```
SELECT *
FROM SpatialIndex.Test
WHERE name %startswith 'za'
and %ID %FIND search_index(coord,'radius','x=55,y=55,radiusX=2,radiusY=2') size ((100
000))
```

Merci à tous ceux qui ont lu ou au moins parcouru cet article.

Outre les liens de documentation ci-dessous, vous pouvez également trouver utile d'examiner les implémentations alternatives des interfaces *%Library.FunctionalIndex* et *%SQL.AbstractFind*. Pour visualiser ces implémentations, ouvrez l'une de ces classes dans Caché Studio et choisissez **Class > Inherited Classes** dans le menu.

Liens:

- %FIND
- searchindex
- %SQL.AbstractFind
- %Library.FunctionalIndex

#Bases de données #Indexation #Object Data Model #SQL #Caché

URL de la source:https://fr.community.intersystems.com/post/cr%C3%A9ation-dindex-personalis%C3%A9-dans-cach%C3%A9