Article
Michael Braam · Feb 20, 2017 14m read

# Making encrypted datafields SQL-searchable

## Overview

Encryption of sensitive data becomes more and more important for applications. For example patient names, SSN, address-data or credit card-numbers etc..

Cache supports different flavors of encryption. Block-level database encryption and data-element encryption. The block-level database encryption protects an entire database. The decryption/encryption is done when a block is written/read to or from the database and has very little impact on the performance.

With data-element encryption only certain data-fields are encrypted. Fields that contain sensitive data like patient data or credit-card numbers. Data-element encryption is also useful if a re-encryption is required periodically. With data-element encryption it is the responsibility of the application to encrypt/decrypt the data.

Both encryption methods leverage the managed key encryption infrastructure of Caché.

The following article describes a sample use-case where data-element encryption is used to encrypt person data.

But what if you have hundreds of thousands of records with an encrypted datafield and you have the need to search that field? Decryption of the field-values prior to the search is not an option. What about indices?

This article describes a possible solution and develops step-by-step a small example how you can use SQL and indices to search encrypted fields.

## Introduction

Cache provides a variety of methods to perform data-element encryption, hashing or message authentication code generation. These are all defined in the **%SYSTEM.Encryption** class.

For example if you want to encrypt a datafield, you can either use the managed key infrastructure of Caché and use **%SYSTEM.Encryption.AESCBCManagedKeyEncrypt(Plaintext As** %String**, KeyID As** %String**)** or you can use

**%SYSTEM.Encryption.AESCBCEncrypt(plaintext As** %String**, key As** %String**, IV As** %String**)** . In the former case you only have to provide the plain text and the Key-Id of the encryption key you want to use. In the latter you have to provide the plain text, the key itself and a so-called initialization vector.

Both methods implement a ***Cipher Block Chained (CBC)*** encryption. That means that prior to the encryption of a plain text block this plain text is XORed with the cipher text of the previous block. If there is no previous block the initialization vector comes into play.

One advantage of those CBC encryption methods is that two subsequent encryption-method calls for the same plain-text result in different cipher texts. In this sense the encryption is not deterministic. That also means that the cipher texts are not suitable for indexing. How can we resolve this?

## Building a simple class

Start Atelier or Studio and create a class DC.Person. The class inherits from %Persistent.

To keep it simple the class only contains properties LastName, FirstName, DOB. The values of the property LastName will be encrypted.

It will also contain an implementation of the ***%OnNew()***-callback and a helper method ***GenerateData(pRowCount)***. GenerateData will generate test data. By default if not specified otherwise it will create 100 rows.

Below you find the class-code:

```
Class DC.Person extends %Persistent {

Property LastName as %String(MAXLEN = "");

Property FirstName as %String;

Property DOB as %Date;

Method %OnNew(pLastName as %String = "", pFirstName as %String = "", pDOB as %Date =
"") as %Status [ Private, ServerOnly = 1 ]
{
    try {
        if pLastName = "" {
            set ..LastName = ##class(%PopulateUtils).LastName()
        }
        else {
            set ..LastName = pLastName
        }
        if pFirstName = "" {
            set ..FirstName = ##class(%PopulateUtils).FirstName()
        }
        else {
            set ..FirstName = pFirstName
        }
        if pDOB = "" {
            set ..DOB = ##class(%PopulateUtils).Date()
        }
        else {
            set ..DOB = pDOB
        }
    }
    catch tEx {
        write !,tEx.DisplayString()
    }

    return $$$OK
}

ClassMethod GenerateData(pRowCount as %Integer = 100) as %Integer
{
#dim tCounter as %Integer = 0

    for i = 1:1:pRowCount {
        set tSc = ..%New().%Save()
        if tSc set tCounter = tCounter + 1
    }

    return tCounter
```

```
}

}
```

Please note that the property **LastName** sets the MAXLEN = "". The default MAXLEN for %String properties is 50. This will not be sufficient for an encrypted string value here.

Compile the class and test the GenerateData method by starting a terminal and entering the following code at the command-prompt:

```
write ##class(DC.Person).GenerateData()
```

This should generate 100 test records. You can verify this in the System Management Portal. Make sure you are connected to the correct namespace. Then start:
"**System Explorer**" -> "**SQL**" and type in the following SELECT-statements:

```
select count(*) from dc.person
```

and

```
select * from dc.person
```

You should see your 100 test records. A sample result should look like this:

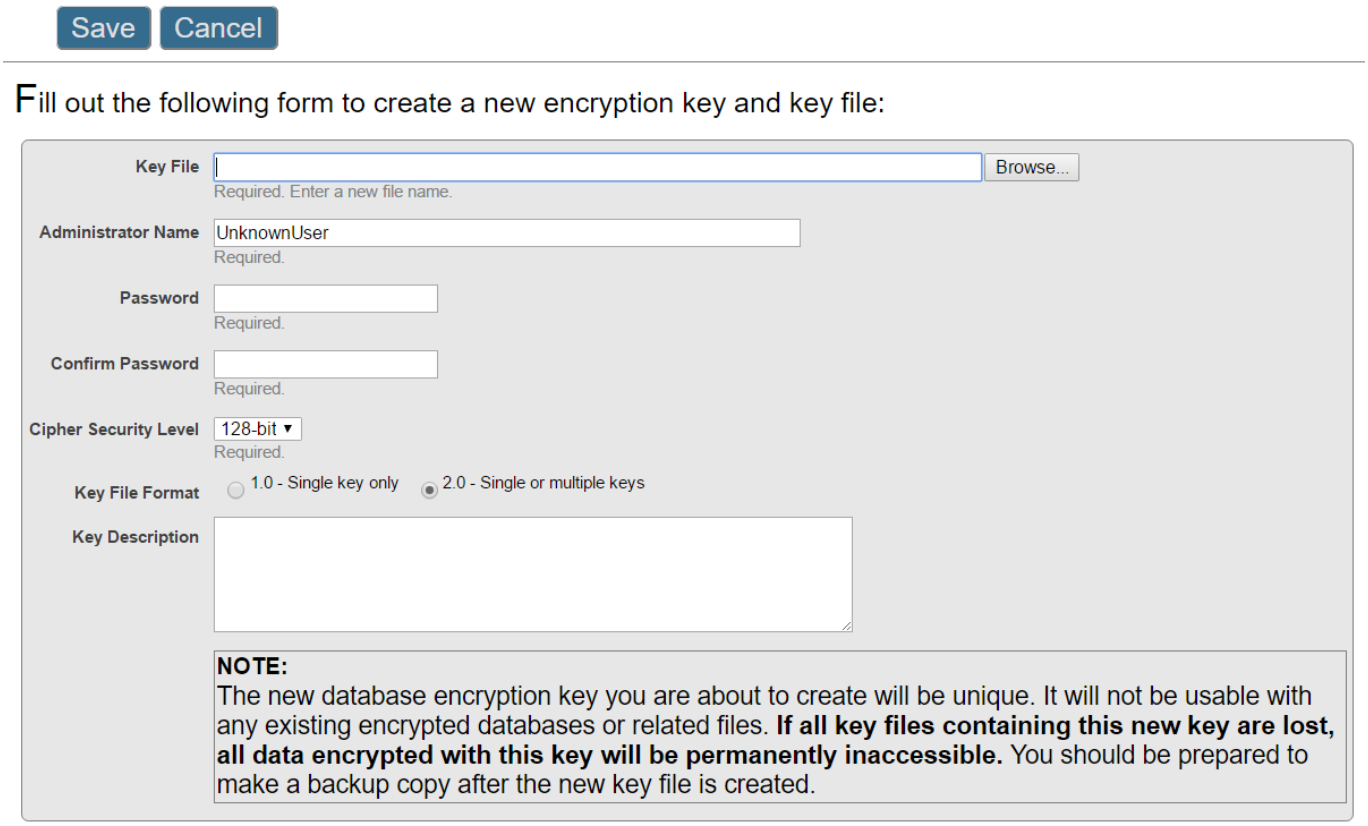
| ID | DOB | FirstName | LastName |
|---|---|---|---|
| 1 | 11/08/1937 | Usha | Wilson |
| 2 | 04/02/1997 | Mo | Taylor |
| 3 | 01/27/1949 | Zelda | Drabek |
| 4 | 03/29/1948 | Susan | Lee |
| 5 | 06/26/1962 | Michelle | Ximines |
| 6 | 06/19/1928 | James | Lepon |
| 7 | 07/13/1958 | Howard | Noodleman |
| 8 | 10/26/1953 | Peter | Eno |
| 9 | 08/07/1983 | Patrick | Hills |
| 10 | 03/20/1960 | Roberta | Rotterman |
| 11 | 03/29/2003 | Pat | Chesire |
| 12 | 05/01/1996 | Xavier | Zweifelhofer |
| 13 | 08/26/1974 | Charlotte | Burroughs |
| 14 | 06/22/1930 | Linda | Allen |
| 15 | 03/22/1928 | Jules | Townsend |
| 16 | 10/30/2001 | Pat | Simpson |
| 17 | 11/21/2011 | Susan | Lopez |
| 18 | 08/13/2004 | Al | Bukowski |
| 19 | 08/17/1933 | William | McCormick |
| 20 | 04/27/1974 | Hannah | Pantaleo |
| 21 | 11/07/1958 | Kevin | Smith |
| 22 | 06/15/1994 | Chris | Adams |
| 23 | 06/22/2000 | Belinda | Ximines |
| 24 | 02/08/1994 | Frances | Huff |
| 25 | 08/22/1968 | Buzz | Munt |
| 26 | 02/22/1989 | Kristen | Sato |
| 27 | 05/06/1951 | Jocelyn | Xerxes |
| 28 | 02/12/2005 | Emily | Eno |
| 29 | 10/25/1939 | Dan | Noodleman |

In the next step we will modify our class to store the values of LastName as encrypted strin*gs.*

## *Adding encryption*

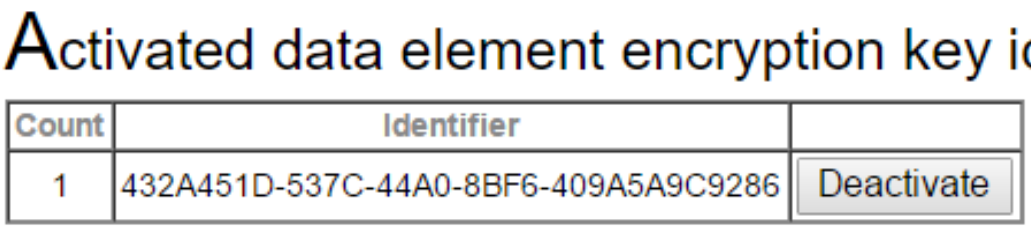
In our example we will use the manged key infrastructure of Caché. So please use the System Management Portal to create and activate an ***Encryption Key***. In the System management Portal go to "**System Administration**" -> "**Encryption**" -> "**Create New Encryption Key File**".

Fill out the form in the upcoming page and save the key file.




Once you have created the key file, please activate it via the System Management Portal. Please go to "**System Administration**" -> "**Encryption**" -> "**Data Element Encryption**"

Click "**Activate Key**" and then "**Browse**" in the upcoming form. Select your key file enter the Key Adminstrator name and the password you had entered during the key creation. In case of a successful activation the activated key is shown as below:




Next we will use the encryption key and the available encryption API to encrypt the lastnames in our person class.

In our example we will use the **%SYSTEM.Encryption.AESCBCManagedKeyEncrypt(Plaintext As** %String**, KeyID As** %String**)** method.

This method expects two parameters. The plaintext and the ID of the encryption key we want to use. The **%SYSTEM.Encryption** class also provides a method which returns the IDs of the activated encryption key. This is done by the method **ListEncryptionKeys**. This method returns a comma-separated list of the activated Key-IDs.

So we have everything we need available to develop a method **Encypt** which receives a plain-text string as a parameter and returns a cipher-text. Our method code looks like this:

```
Method Encrypt(pVal As %String) As %String [ Private ]
{
```

```
#dim tCipher as %String = ""

    try {
        set tKeyId = $system.Encryption.ListEncryptionKeys()
        set tCipher = $system.Encryption.AESCBCManagedKeyEncrypt(pVal,tKeyId)
    }
    catch tEx {
        write !,tEx.DisplayString()
    }

    return tCipher
}
```

In the code above we assume that only one encryption key is active.
We add this method code to our class and modify the **%OnNew** method so that it uses the **Encrypt** method (see the changes in blue color) :

```
Method %OnNew(pLastName as %String = "", pFirstName as %String = "", pDOB as %Date =
"") as %Status [ Private, ServerOnly = 1 ]
{
#dim tName as %String

    try {
        if pLastName = "" {
            set tName = ##class(%PopulateUtils).LastName()
        }
        else {
            set tName = pLastName
        }
        if pFirstName = "" {
            set ..FirstName = ##class(%PopulateUtils).FirstName()
        }
        else {
            set ..FirstName = pFirstName
        }
        if pDOB = "" {
            set ..DOB = ##class(%PopulateUtils).Date()
        }
        else {
            set ..DOB = pDOB
        }
        set ..LastName = ..Encrypt(tName)
    }
    catch tEx {
        write !,tEx.DisplayString()
    }

    return $$$OK
}
```

Now let's delete the previously generated test data and run the **GenerateData** method again. To delete the test-data simply delete either the global ^DC.PersonD or run a SQL delete statement

```
delete from dc.person
```

Then run the GenerateData method again and check the results as above. You should see something like this:

| ID | DOB | FirstName | LastName |
|---|---|---|---|
| 1 | 11/18/1943 | Keith | $432A451D-537C-44A0-8BF6-409A5A9C9286£jW91J÷õ^êì§í3ò=eÊ¿b¬MÀÜ] |
| 2 | 01/05/1931 | Christen | $432A451D-537C-44A0-8BF6-409A5A9C92862tî±!ûòÍ¸úêW>r:Kzý2ÃCV/ÞØ¥ |
| 3 | 07/04/1979 | Usha | $432A451D-537C-44A0-8BF6-409A5A9C9286Ttv)íý½UÄ.BóaH9ý.%ìQýYÎ |
| 4 | 03/02/2004 | Danielle | $432A451D-537C-44A0-8BF6-409A5A9C9286y w+=¶|êÃÏ¹#½~ó>UÉfrô_ |
| 5 | 05/27/2009 | Patricia | $432A451D-537C-44A0-8BF6-409A5A9C9286@: ÿ°ó¼¦ÜF¬Cïòé¾²K=X²ÀñT:8¡ |
| 6 | 08/03/1980 | Jules | $432A451D-537C-44A0-8BF6-409A5A9C9286.æ¦Ãnc"ô1ðÝòZ G@¾îånAÀ+ÝH0é¿j |
| 7 | 01/14/1961 | Heloisa | $432A451D-537C-44A0-8BF6-409A5A9C9286Ùewy°ïAZñÁ>´àCÏC¢ 0¬( |
| 8 | 06/19/2015 | Filomena | $432A451D-537C-44A0-8BF6-409A5A9C9286+k®w3!¢ZW0øXtð¹¾EMzmÍÎàHt³É!ø |
| 9 | 01/06/2002 | Imelda | $432A451D-537C-44A0-8BF6-409A5A9C92862¿QËlúÝj¾ÈÑc^ÝKS*ÿ£.§Î |
| 10 | 06/12/2011 | Greta | $432A451D-537C-44A0-8BF6-409A5A9C9286¤1óÏpõÃ¼D 2>Jõ4{Ê¼)ýÞFÙÿ¾#Ú |
| 11 | 03/19/2010 | Brenda | $432A451D-537C-44A0-8BF6-409A5A9C9286yñiZ¹üGb~Ô/³Pdrx(mL§1ÜÆÁ{4« |
| 12 | 03/23/1992 | Molly | $432A451D-537C-44A0-8BF6-409A5A9C9286uÈ³tµm¼é· Ä|d¶SÚ¸»ï£¦{² |
| 13 | 11/08/1983 | Zelda | $432A451D-537C-44A0-8BF6-409A5A9C9286BÖ²Ãë ¨`CÁg;tæÆ¿\Ø' |
| 14 | 12/16/1952 | Elmo | $432A451D-537C-44A0-8BF6-409A5A9C9286÷#êÒiûòj+Ï£±Ã{õã¹G_z«« |
| 15 | 12/01/2008 | Phyllis | $432A451D-537C-44A0-8BF6-409A5A9C9286ýOµýho{¦û© Ím6b«XÛKzG§´í |
| 16 | 08/29/1937 | Jules | $432A451D-537C-44A0-8BF6-409A5A9C9286±FÇ¡ 8Íæø £±djí%eHU-´úµ |
| 17 | 03/31/1974 | Ted | $432A451D-537C-44A0-8BF6-409A5A9C9286ÿHéÈ&!3/½³!ÛÒ¶Y2|$ \¿ÉVÿh |
| 18 | 10/10/1966 | Sam | $432A451D-537C-44A0-8BF6-409A5A9C9286ñó¡0õö¦æ´ájÁóÙìò|7Ý)?-ëô |
| 19 | 08/19/1931 | Diane | $432A451D-537C-44A0-8BF6-409A5A9C9286kr¢,g³æäV#êuÎXØ³`ëZ`_qëH¦ |
| 20 | 11/19/1951 | Alfred | $432A451D-537C-44A0-8BF6-409A5A9C9286Ähaú»áAÞèÐÏ´o§H98sî%cÛbÐ |
| 21 | 07/22/1944 | Brenda | $432A451D-537C-44A0-8BF6-409A5A9C9286å_{ÚîÁÖË¤v²ó*¿TZ°Oe:¦øÊ |
| 22 | 03/07/2004 | Wilma | $432A451D-537C-44A0-8BF6-409A5A9C9286C+àCëêëqòÔÃ³Z¤¶Y2¯UOý |
| 23 | 08/17/1926 | Olga | $432A451D-537C-44A0-8BF6-409A5A9C9286ÀY³éÿÈD ±"7{ð÷"µÒJÇ¶owÝÛÑöÃT |
| 24 | 03/11/1989 | Alexandra | $432A451D-537C-44A0-8BF6-409A5A9C9286l1ôo é-//J£E©URn¥'RNÑ°¤¹¹ |
| 25 | 04/06/1935 | Jocelyn | $432A451D-537C-44A0-8BF6-409A5A9C9286î?»ß%½Hù¯>uátÃÿÎ{¢xPÔ«wH ^ô |
| 26 | 07/16/1987 | Dick | $432A451D-537C-44A0-8BF6-409A5A9C9286¸¤xÒ1÷N W®-{Rå£±¡$ð_`%1Ó±hS |
| 27 | 04/26/2003 | Roberta | $432A451D-537C-44A0-8BF6-409A5A9C9286J+èä.<ñ¤ -û4- ¢@.£ßOB× |
| 28 | 07/02/1938 | Debra | $432A451D-537C-44A0-8BF6-409A5A9C9286¤'P -BRàù-õNÁý.Çæ'ñáÝ« |
| 29 | 07/27/1957 | Sam | $432A451D-537C-44A0-8BF6-409A5A9C9286¿µCÛz `«NL¼ñJ_ö7¦53½Óõ^k |

You don't see the lastnames in plain text any longer. If you look into the global ^DC.PersonD you will also see only the encrypted values.


To retrieve the plain-text from the encrypted data you have to use the corresponding decryption-method

**%SYSTEM.Encryption.AESCBCManagedKeyDecrypt(Ciphertext As %String)**.

But how can I query the encrypted data without decrypting the data first? We will implement this in the next section.

## *Adding an index for SQL*

As stated above the cipher-texts are not suitable for indexing. This is due to the non deterministic results of the encryption. I.E. if you encrypt the same plain-text string twice you will get different cipher-texts.

So we need something that is deterministic. You can either use a hash or a MAC (Message Authentication Code) for this purpose. Caché supports both types of hashing. If you hash the same value twice you 'll get the same result. So you could ask, why don't we use the hash or MAC for the data field as well. Hashes or MACs are not reversible. That means you can't create the plain text from the hash or MAC.

So the idea is to use hash values or MACs to build the index. This is easy. Add an redundant field **hashname** to your class which is used to store the hash or MAC and build an index on that field.
This helps in situations in which the decrypted values of a field are either unique or you don't use wildcards in your queries.

Something like this will work:

```
Select id, DOB from DC.Person where hashname = '<hashvalue for Smith>'
```

This approach has some disadvantages. You can't run queries similiar to the one below:

```
Select id, DOB from DC.Person where name like 'Sm%'
```

And you have to add redundancy (the hashname field). So let's look for a different approach.

A possible solution here is to consider the name field as a collection with the collection values '**S**', '**Sm**', '**Smi**', '**Smit**', '**Smith**' and to define an index on that collection. But **LastName** really is just a normal %String property and not a collection. How can I build this index?
The answer is easy. If you define an index on a collection field you can have control on how this index is built. To do this you have to implement a method with the name **<propertyname>BuildValueArray(pName As %String, ByRef pNameArray As %String)** . In our case it is **LastNameBuildValueArray(pName As %String, ByRef pNameArray As %String)**.

```
ClassMethod LastNameBuildValueArray(pName As %String, ByRef pNameArray As %String) As
 %Status
{

}
```

The parameter pName contains the atcual value of LastName. The ref parameter pNameArray returns an array with the "collection" values of LastName.

So let's define a collection index on LastName. This index definition looks like this:

```
Index idxLastName On LastName(elements);
```

This looks similar to "normal" index definitions. The key-word **elements** within the parenthesis specifies that we want to index the collection elements. Plese see the documentation for more details.

Next let's implement the **LastNameBuildValueArray()** method. It could look like this:

```
ClassMethod LastNameBuildValueArray(pName As %String, ByRef pNameArray As %String) As
 %Status
{
#dim tName as %String
#dim tRet as %Status = $$$OK

    try {
    set tName = $system.Encryption.AESCBCManagedKeyDecrypt(pName)
    set tKey = $system.Encryption.ListEncryptionKeys()
    for i=$length(tName):-1:1 {
        set pNameArray(i) = $system.Encryption.Base64Encode($system.Encryption.HMACSH
A1($extract(tName,1,i),tKey))
        }
    }
    catch tEx {
        set tRet = tEx.AsStatus()
    }

    return tRet
}
```

Let's take a quick look. The parameter pName contains the actual value of LastName. This means it contains the encrypted value. This is not suitable here. So we have to decrypt the value first:

```
set tName = $system.Encryption.AESCBCManagedKeyDecrypt(pName)
```

We then can use the plain-text value in **tName** to "construct" our collection values "S", "Sm", "Smi", "Smit", "Smith", create the MAC and add an entry to the output array **pNameArray**.

```
for i=$length(tName):-1:1 {
    set pNameArray(i) = $system.Encryption.Base64Encode($system.Encryption.HMACSHA1($
extract(tName,1,i),tKey))
}
```

Delete your test-data as described above and regenerate it. This will also create the index.

## Searching

The last step missing is how to run SQL queries against that table and use **LastName** as a search condition. Since we have indexed the MACs we have to convert our search value to a MAC. To do this we define a stored procedure which returns the MAC for a plain-text value:

```
ClassMethod MakeSearchKey(pVal) As %String [ SqlName = MakeSearchKey, SqlProc ]
{
    set tKey = $system.Encryption.ListEncryptionKeys()
    return $system.Encryption.Base64Encode($system.Encryption.HMACSHA1(pVal,tKey))
}
```

It's important that you use the same secrect-key in this method as you did for indexing.

**Important: To prevent unauthorized users from using this stored procedure you should use the SQL security/privilege system. This is also true for access to the LastName property. If a user is not allowed to decrypt it, its probably not neccessary that he can see it at all.**

Before we now run our searches against our data let's add a calculated property **decLastName** to our class-definition. This property returns the decrypted value of **LastName** and is only for debugging purposes to verify that our query returns the correct results.

Please find below the definition of **decLastName**:

```
Property decLastName As %String [ Calculated, SqlComputeCode = {set {*} = $system.Enc
ryption.AESCBCManagedKeyDecrypt({LastName})}, SqlComputed ];
```

So let's query our table and look for Persons with the LastName like 'S%':

```
SELECT *
FROM   dc.person
WHERE  for some %element(lastname)(%value=
       (
              SELECT dc.makesearchkey('S')))
```

If you run this query in the System-Management-Portal you should see an output similiar to the one below:

| ID | DOB | FirstName | LastName | decLastName |
|---|---|---|---|---|
| 32 | 03/29/1981 | Norbert | □$432A451D-537C-44A0-8BF6-409A5A9C9286□Vp¢[□2IYNøh¹¿k□èx/ùQ□□ØÝNáf□m¬□FK | Schulte |
| 38 | 02/09/1947 | Stavros | □$432A451D-537C-44A0-8BF6-409A5A9C9286□□Ubá□$□□º£°wÅñ¯□æ§Ã□v+°¨µÏ)□a) | Simpson |
| 70 | 06/09/1930 | Julie | □$432A451D-537C-44A0-8BF6-409A5A9C9286□V□¦□/ñ¾□Ø□□z□¤:r N□h,□ö□þ=□× □±ô | Semmens |
| 81 | 10/11/1953 | Emma | □$432A451D-537C-44A0-8BF6-409A5A9C9286□□□□/¢ R#□□©©¾□úBøTAç□R±(}mbVãâåA | Smyth |

Note that we use a Caché SQL extension (**for some %element()...**) in the query. This is a predicate condition especially for searches on collection properties/fields. You can find details on this in the documentation.

Please keep in mind that the decLastName column is just there to verify that the query returns the correct results. Let's search for LastName like 'Si%' and modify the statement accordingly:

```
SELECT *
FROM   dc.person
WHERE  for some %element(lastname)(%value=
       (
```

```
            SELECT dc.makesearchkey('Si')))
```

This is the output:

| ID | DOB | FirstName | LastName | decLastName |
|---|---|---|---|---|
| 38 | 02/09/1947 | Stavros | □$432A451D-537C-44A0-8BF6-409A5A9C9286□□Ubá□$□□º£ºwÅñ¯□æ§Ã□v+ °¨µÏ)□a) | Simpson |

Please take a look at the query-plan and you will see that our index **idxLastName** is used during the query execution.

Below you find the full class listing:

```
Class DC.Person extends %Persistent
{

Property LastName as %String(MAXLEN = "");

Property FirstName as %String;

Property DOB as %Date;

Property decLastName As %String [ Calculated, SqlComputeCode = {set {*} = $system.Enc
ryption.AESCBCManagedKeyDecrypt({LastName})}, SqlComputed ];

Index idxLastName on LastName(elements);

Method %OnNew(pLastName as %String = "", pFirstName as %String = "", pDOB as %Date =
"") as %Status [ Private, ServerOnly = 1 ]
{
#dim tName as %String

    try {
        if pLastName = "" {
            set tName = ##class(%PopulateUtils).LastName()
        }
        else {
            set tName = pLastName
        }
        if pFirstName = "" {
            set ..FirstName = ##class(%PopulateUtils).FirstName()
        }
        else {
            set ..FirstName = pFirstName
        }
        if pDOB = "" {
            set ..DOB = ##class(%PopulateUtils).Date()
        }
        else {
            set ..DOB = pDOB
        }
        set ..LastName = ..Encrypt(tName)
    }
    catch tEx {
        write !,tEx.DisplayString()
    }
```

```
    return $$$OK
}

Method Encrypt(pVal As %String) As %String [ Private ]
{
#dim tCipher as %String = ""

    try {
        set tKeyId = $system.Encryption.ListEncryptionKeys()
        set tCipher = $system.Encryption.AESCBCManagedKeyEncrypt(pVal,tKeyId)
    }
    catch tEx {
        write !,tEx.DisplayString()
    }

    return tCipher
}

ClassMethod GenerateData(pRowCount as %Integer = 100) as %Integer
{
#dim tCounter as %Integer = 0

    for i = 1:1:pRowCount {
        set tSc = ..%New().%Save()
        if tSc set tCounter = tCounter + 1
    }

    return tCounter
}

ClassMethod MakeSearchKey(pVal) As %String [ SqlName = MakeSearchKey, SqlProc ]
{
    set tKey = $system.Encryption.ListEncryptionKeys()
    return $system.Encryption.Base64Encode($system.Encryption.HMACSHA1(pVal,tKey))
}

ClassMethod LastNameBuildValueArray(pName As %String, ByRef pNameArray As %String) As
 %Status
{
#dim tName as %String
#dim tRet as %Status = $$$OK

    try {
    set tName = $system.Encryption.AESCBCManagedKeyDecrypt(pName)
    set tKey = $system.Encryption.ListEncryptionKeys()
    for i=$length(tName):-1:1 {
        set pNameArray(i) = $system.Encryption.Base64Encode($system.Encryption.HMACSH
A1($extract(tName,1,i),tKey))
        }
    }
    catch tEx {
        set tRet = tEx.AsStatus()
    }

    return tRet
}

}
```

## *Summary*

We have created a class **DC.Person** which contains a property **LastName**. The property values are encrypted and only the encrypted values are stored in the database. To make this property SQL-searchable without the need to decrypt it prior to the search we have defined an index which contains hashed values. Nowhere neither in the data-global nor in the index-global we store the plain-text values.
It is not necessary to decrypt the field values prior to the search even if you use the encrypted field in your where-clause.

Granting access to the column **LastName** and the stored procedure **MakeSearchKey** only to users who are allowed to decrypt the names might be an option to make the data more secure.

#Encryption #Indexing #Object Data Model #SQL #Caché

---

**Source URL:**https://community.intersystems.com/post/making-encrypted-datafields-sql-searchable