Article
Vitaliy Serdtsev · Jul 7, 2017 19m read

Indexing of non-atomic attributes

Quotes (1NF/2NF/3NF)[ru]:

> Every row-and-column intersection contains exactly one value from the applicable domain (and nothing else).
> The same value can be atomic or non-atomic depending on the purpose of this value. For example, "4286" can be
>
> - **atomic**, if its denotes "a credit card's PIN code" (if it's broken down or reshuffled, it is of no use any longer)
> - **non-atomic**, if it's just a "sequence of numbers" (the value still makes sense if broken down into several parts or reshuffled)

This article explores the standard methods of increasing the performance of SQL queries involving the following types of fields: string, date, simple list (in the $LB format), "list of " and "array of ".

Introduction

To begin with, let's take a look at a classic example – a list of phone numbers.
Let's create our test data:

```
create table cl_phones(tname varchar2(100), phone varchar2(30));
insert into cl_phones(tname,phone) values ('Andrew','867-843-25');
insert into cl_phones(tname,phone) values ('Andrew','830-044-35');
insert into cl_phones(tname,phone) values ('Andrew','530-055-35');
insert into cl_phones(tname,phone) values ('Max','530-055-35');
insert into cl_phones(tname,phone) values ('Max','555-011-35');
insert into cl_phones(tname,phone) values ('Josh','530-055-31');
insert into cl_phones(tname,phone) values ('Josh','531-051-32');
insert into cl_phones(tname,phone) values ('Josh','532-052-33');
insert into cl_phones(tname,phone) values ('Josh','533-053-35');
```

Now, let's display a comma-delimited list of phones for each name:

```
SELECT
   %exact(tname) tname,
   LIST(phone) phonestr
FROM cl_phones
GROUP BY tname
```

Or this way:

```
SELECT
   distinct %exact(tname) tname,
   LIST(phone %foreach(tname)) phonestr
FROM cl_phones
```

tname	phonestr
Andrew	867-843-25,830-044-35,530-055-35
Josh	530-055-31,531-051-32,532-052-33,533-053-35
Max	530-055-35,555-011-35

By building an index by *phone*, we can search really quickly by a specific phone. The only disadvantage of such a solution is name duplication: the more elements are in the list, the bigger our DB will grow.

Therefore, it is sometimes useful to store multiple values in the same field (this can be a list of phones or its parts, passwords, anything) in the form of a delimited string - and yet be able to quickly search by separate values. Of course, you can create a regular index for such a field and search for a substring in this long string, but, first of all, there can be many such elements, and the index length can become substantial, and secondly, such an index will not help us speed up the search in any way.

So what do we do?

Specifically for such cases involving fields with collections, a special type of index was implemented.
Collections can be either "real" (built-in *list of* and *array of*) or "virtual".

In case of built-in collections, the system is responsible for building such an index for them, and the developer cannot interfere with this process. However, in case of virtual collections, the responsibility for building the index is on the developer's shoulders.

A simple delimited string, a date, a simple list – these are examples of such "virtual" collections.

So, collection indexes have the following syntax:

```
INDEX idx1 ON (MyField(ELEMENTS));
```

or

```
INDEX idx1 ON (MyField(KEYS));
```

The index is built with the help of the ***propertyname****BuildValueArray* method that the developer must implement on his own.

The method has the following general signature:

```
ClassMethod propertynameBuildValueArray(value, ByRef valueArray) As %Status
```

Where:

- *value* – field value to be broken down into elements;
- *valueArray* – the resulting array containing separate elements.
  The array is a set of key/value of the following form:

array(key1)=value1
array(key2)=value2
and so on.

As mentioned above, this method is generated for built-in collections automatically by the system and has the **[Final]** attribute, which doesn't let the developer redefine it.

Let's build these indexes and see how we can use them in our SQL queries.

Note:
To make sure there are no artefacts remaining from the previous example, it is recommended to empty the globals and the storage scheme of the class prior to building a new index.

Delimited string

Let's create the following class:

```
Class demo.test Extends %Persistent
{

Index iPhones On Phones(ELEMENTS);

Property Phones As %String;

ClassMethod PhonesBuildValueArray(
  value,
  ByRef array) As %Status
{
  i value="" {
    s array(0)=value
  }else{
    s list=$lfs(value,","),ptr=0
    while $listnext(list,ptr,item){
      s array(ptr)=item
    }
  }
  q $$$OK
}

ClassMethod Fill()
{
  k ^demo.testD,^demo.testI
  &sql(insert into demo.test(phones)
    select null union all
    select 'a' union all
    select 'b,a' union all
    select 'b,b' union all
    select 'a,c,b' union all
    select ',,'
  )
  zw ^demo.testD,^demo.testI
}
}
```

Let's call the **Fill**() method in the terminal:

```
USER>d ##class(demo.test).Fill()
^demo.testD=6
^demo.testD(1)=$lb("","")
^demo.testD(2)=$lb("","a")
^demo.testD(3)=$lb("","b,a")
^demo.testD(4)=$lb("","b,b")
^demo.testD(5)=$lb("","a,c,b")
^demo.testD(6)=$lb("",",,")
^demo.testI("iPhones"," ",1)=""
^demo.testI("iPhones"," ",6)=""
^demo.testI("iPhones"," A",2)=""
^demo.testI("iPhones"," A",3)=""
^demo.testI("iPhones"," A",5)=""
^demo.testI("iPhones"," B",3)=""
^demo.testI("iPhones"," B",4)=""
^demo.testI("iPhones"," B",5)=""
^demo.testI("iPhones"," C",5)=""
```

As you can see, the index doesn't have the entire string, just its parts. Therefore, it's up to you how you break one long string into substrings. Apart from delimited strings, these can be xml, json files or something else.

ID	Phones
1	(null)
2	a
3	b,a
4	b,b
5	a,c,b
6	,,

Let's now try to find all the substrings containing "a". As a rule, such predicates as **like '%xxx%'** or **['xxx'** are used for this purpose:

```
select * from demo.test where Phones [ 'a'
select * from demo.test where Phones like '%a%'
```

However, in this case, our **iPhones** index won't be used. To use it, you need to use a special predicate

```
FOR SOME %ELEMENT(ourfield) (%VALUE = elementvalue)
```

Considering the above, our query will look like this:

```
select * from demo.test where for some %element(Phones) (%value = 'a')
```

As the result, thanks to the use of a specialized index, the speed of this query will be much higher compared with previous variants.

Of course, more complex conditions are also possible, for example:

```
(%Value %STARTSWITH 'a')
(%Value [ 'a' and %Value [ 'b')
(%Value in ('c','d'))
(%Value is null)
```

Time for some magic...

Hiding sensitive data

In the **BuildValueArray** method, we usually populate the *array* array with *value*.

But what will happen if we don't follow this rule?

Let's try the following example:

```
Class demo.test Extends %Persistent
{

Index iLogin On Login(ELEMENTS);

Property Login As %String;

ClassMethod LoginBuildValueArray(
  value,
  ByRef array) As %Status
{
  i value="Jack" {
    s array(0)="test1"
    s array(1)="test2"
    s array(2)="test3"
  }elseif value="Pete" {
    s array("-")="111"
    s array("5.4")="222"
    s array("fg")="333"
  }else{
    s array("key")="value"
  }
  q $$$OK
}

ClassMethod Fill()
{
  k ^demo.testD,^demo.testI
  &sql(insert into demo.test(login)
    select 'Jack' union all
    select 'Jack' union all
    select 'Pete' union all
    select 'Pete' union all
    select 'John' union all
    select 'John'
  )
  zw ^demo.testD,^demo.testI
}
}
```

ID	Login
1	Jack
2	Jack
3	Pete
4	Pete
5	John

ID	Login
6	John

And now – attention! – let's try executing the following query:

```
select * from demo.test where for some %element(Login) (%value = '111')
```

ID	Login
3	Pete
4	Pete

As the result, we see that some of the data is visible in the table, and some is hidden in the index, but is still searchable. How can we use that?

For example, you can hide not one, as is usually the case, but an entire set of passwords accessible to a particular user, from the index. Alternatively, you can hide any sensitive information that you don't want to be opened with SQL. Of course, there are other ways of doing this, such as GRANT column-privilege. But in this case, you will need to use stored procedures for accessing the protected fields.

Hiding sensitive data (continued)

If you take a look at the globals containing data and indexes for our table, we will not see the values of our keys: "5.4", "fg", etc.:

```
^demo.testD=6
^demo.testD(1)=$lb("","Jack")
^demo.testD(2)=$lb("","Jack")
^demo.testD(3)=$lb("","Pete")
^demo.testD(4)=$lb("","Pete")
^demo.testD(5)=$lb("","John")
^demo.testD(6)=$lb("","John")
^demo.testI("iLogin"," 111",3)=""
^demo.testI("iLogin"," 111",4)=""
^demo.testI("iLogin"," 222",3)=""
^demo.testI("iLogin"," 222",4)=""
^demo.testI("iLogin"," 333",3)=""
^demo.testI("iLogin"," 333",4)=""
^demo.testI("iLogin"," TEST1",1)=""
^demo.testI("iLogin"," TEST1",2)=""
^demo.testI("iLogin"," TEST2",1)=""
^demo.testI("iLogin"," TEST2",2)=""
^demo.testI("iLogin"," TEST3",1)=""
^demo.testI("iLogin"," TEST3",2)=""
^demo.testI("iLogin"," VALUE",5)=""
^demo.testI("iLogin"," VALUE",6)=""
```

So why did we define them in the first place?

To answer this question, let's modify our index a bit and re-populate the table.

```
Index iLogin On (Login(KEYS), Login(ELEMENTS));
```

Globals will now look differently (I am only showing a global with indexes):

```
^demo.testI("iLogin"," -"," 111",3)=""
^demo.testI("iLogin"," -"," 111",4)=""
^demo.testI("iLogin"," 0"," TEST1",1)=""
^demo.testI("iLogin"," 0"," TEST1",2)=""
^demo.testI("iLogin"," 1"," TEST2",1)=""
^demo.testI("iLogin"," 1"," TEST2",2)=""
^demo.testI("iLogin"," 2"," TEST3",1)=""
^demo.testI("iLogin"," 2"," TEST3",2)=""
^demo.testI("iLogin"," 5.4"," 222",3)=""
^demo.testI("iLogin"," 5.4"," 222",4)=""
^demo.testI("iLogin"," FG"," 333",3)=""
^demo.testI("iLogin"," FG"," 333",4)=""
^demo.testI("iLogin"," KEY"," VALUE",5)=""
^demo.testI("iLogin"," KEY"," VALUE",6)=""
```

Great, now we are storing both key values and element values. How can we use this in the future?

For instance, in the example with passwords (see above), we can store passwords along with their expiry dates or something else. In our query, this can be used in the following way:

```
select * from demo.test where for some %element(Login) (%key='-' and %value = '111')
```

It's up to you where you want to store things, but keep in mind that keys are unique and values are not.

Besides, a "collection" index, just like a regular one, can be used for storing additional data:

```
Index iLogin On (Login(KEYS), Login(ELEMENTS)) [ Data = (Login, Login(ELEMENTS)) ];
```

In this case, the query above will not even access data and will take everything from the index, thus saving you time.

Date (time, etc.)

One could ask: how are dates related to collections? The answer is "directly", since we often need to search by day, month or year only. The usual search will be useless here, while "collection-based" search will be just what we need.

Let's consider the following example:

```
Class demo.test Extends %Persistent
{

Index iBirthDay On (BirthDay(KEYS), BirthDay(ELEMENTS));

Property BirthDay As %Date;

ClassMethod BirthDayBuildValueArray(
  value,
  ByRef array) As %Status
{
```

```
  i value="" {
    s array(0)=value
  }else{
    s d=$zd(value,3)
    s array("yy")=+$p(d,"-",1)
    s array("mm")=+$p(d,"-",2)
    s array("dd")=+$p(d,"-",3)
  }
  q $$$OK
}

ClassMethod Fill()
{
  k ^demo.testD,^demo.testI
  &sql(insert into demo.test(birthday)
    select {d '2000-01-01'} union all
    select {d '2000-01-02'} union all
    select {d '2000-02-01'} union all
    select {d '2001-01-01'} union all
    select {d '2001-01-02'} union all
    select {d '2001-02-01'}
  )
  zw ^demo.testD,^demo.testI
}
}
```

ID	BirthDay
1	01.01.2000
2	02.01.2000
3	01.02.2000
4	01.01.2001
5	02.01.2001
6	01.02.2001

Now we can easily – and very quickly! – search for certain parts of the date. For example, here's how you can select everyone who was born in February:

```
select * from demo.test where for some %element(BirthDay) (%key='mm' and %value = 2)
```

ID	BirthDay
3	01.02.2000
6	01.02.2001

Simple list

The Caché DBMS has a special data type for a simple list (**%List**), which can be used instead of a string if the developer has a hard time choosing a delimiter.

Using this field is very similar to using a string.

Let’s consider a small example:

```
Class demo.test Extends %Persistent
{

Index iList On List(ELEMENTS);

Property List As %List;

ClassMethod ListBuildValueArray(
  value,
  ByRef array) As %Status
{
  i value="" {
    s array(0)=value
  }else{
    s ptr=0
    while $listnext(value,ptr,item){
      s array(ptr)=item
    }
  }
  q $$$OK
}

ClassMethod Fill()
{
  k ^demo.testD,^demo.testI
  &sql(insert into demo.test(list)
    select null union all
    select $LISTBUILD('a') union all
    select $LISTBUILD('b','a') union all
    select $LISTBUILD('b','b') union all
    select $LISTBUILD('a','c','b') union all
    select $LISTBUILD('a,,',null,null)
  )
  zw ^demo.testD,^demo.testI
}
}
```

ID	List
1	(null)
2	a
3	b,a
4	b,b
5	a,c,b
6	"a,,",,

Note:
Caché supports three data presentation modes: logical, ODBC and display mode (Data Display Options)

In this case, no element delimiter is used, so we can use any characters in our elements.

When displaying a field of the **%List** type in the ODBC mode, the ODBCDELIMITER parameter is used as a delimiter (equal to "," by default).

For example, in case of such a field

```
Property List As %List(ODBCDELIMITER = "^");
```

ID	List
1	(null)
2	a
3	b^a
4	b^b
5	a^c^b
6	a,,^^

Element search is identical to that for a delimited string:

```
select * from demo.test where for some %element(List) (%value = 'a,,')
```

ID	List
6	a,,^^

Please note that the option with %INLIST does not yet use "collection" indexes, and will therefore be slower than the one provided above:

```
select * from demo.test where 'a,,' %inlist List
```

Collection

Let's rewrite the example above, but use a collection list instead of a simple list:

```
Class demo.test Extends %Persistent
{

Index iListStr On ListStr(ELEMENTS);

Property ListStr As list Of %String;

ClassMethod Fill()
{
  k ^demo.testD,^demo.testI
  &sql(insert into demo.test(liststr)
    select null union all
    select $LISTBUILD('a') union all
    select $LISTBUILD('b','a') union all
    select $LISTBUILD('b','b') union all
    select $LISTBUILD('a','c','b') union all
    select $LISTBUILD('a,,',null,null)
  )
  zw ^demo.testD,^demo.testI
}
}
```

In this example, everything is almost the same, but there are some nuances. Please note the following:

- the COLLATION values of our fields, keys and index elements in *array* undergo corresponding transformations prior to being saved to a global.
  Compare the values in the global index in both examples, especially the representation of the NULL value;

- the **BuildValueArray** method is missing, so we won't be able to use keys, just element values;
- our field type is a special collection class (**%ListOfDataTypes**).

Array collection

As noted above, the list doesn't allow us to use keys. An array can fix this shortcoming.

Let's create the following class:

```
Class demo.test Extends %Persistent
{

Index iArrayStr On (ArrayStr(KEYS), ArrayStr(ELEMENTS));

Property str As %String;

Property ArrayStr As array Of %String;

ClassMethod Fill()
{
  k ^demo.testD,^demo.testI
  &sql(insert into demo.test(str)
  select null union all
  select 'aaa' union all
  select 'bbb' union all
  select 'bbb' union all
  select 'ccc' union all
  select null
  )
  &sql(insert into demo.test_ArrayStr(test,element_key,arraystr)
    select 1,'0','test1' union all
    select 1,'1','test2' union all
    select 1,'2','test3' union all
    select 2,'0','test1' union all
    select 2,'1','test2' union all
    select 2,'2','test3' union all
    select 3,'-','111' union all
    select 3,'5.4','222' union all
    select 3,'fg','333' union all
    select 4,'-','111' union all
    select 4,'5.4','222' union all
    select 4,'fg','333' union all
    select 5,'key','value' union all
    select 6,'key','value'
  )
  zw ^demo.testD,^demo.testI
}
}
```

Some explanations may be needed here:

- our data is still stored in two globals: *^name of class**D*** (data) and *^name of class**I*** (indexes);
- having one class, we already have two tables: the usual demo.test and the additional demo.test_ArrayStr;
- the demo.test_ArrayStr table provides convenient SQL access to the array data and has the following fields, the names of which are partially predefined:
  element_key – value of the key (predefined field name);
  ArrayStr – element value;

test – link to the parent table demo.test;
ID – a service primary key that has the test||element_key format (predefined field name);
- the type of our field is a special collection class (**%ArrayOfDataTypes**).

This said, the content of our tables after the execution of the **Fill**() method will be as follows:

ID	str
1	(null)
2	aaa
3	bbb
4	bbb
5	ccc
6	(null)

ID	test	element_key	ArrayStr
1||0	1	0	test1
1||1	1	1	test2
1||2	1	2	test3
2||0	2	0	test1
2||1	2	1	test2
2||2	2	2	test3
3||5.4	3	5.4	222
3||-	3	-	111
3||fg	3	fg	333
4||5.4	4	5.4	222
4||-	4	-	111
4||fg	4	fg	333
5||key	5	key	value
6||key	6	key	value

It seems that now, having two tables instead of one, we are forced to use a JOIN between them, but it's not so.

Considering the object extensions for SQL provided by the Caché DBMS, our test query showing the **str** field from demo.test for strings with a "-" key and a "111" element value, will look like this:

```
select test ID,test->str from demo.test_ArrayStr where element_key='-' and
arraystr='111'
```

or like this:

```
select %ID, str from demo.test where test_ArrayStr->element_key='-' and
test_ArrayStr->arraystr='111'
```

ID	str
3	bbb
4	bbb

As we can see, there is nothing complex here and no **JOIN's**, since all our data is stored in a single global and Caché knows about the "kinship" of these tables.

This is why you can refer to these fields from both tables. In reality, there is no **test_ArrayStr** field in the demo.test teble, although we can use it to access a related table.

Conclusion

The indexing mechanism described here is widely used in some system classes, such as %Stream.GlobalCharacterSearchable, which indexes a text stream and makes it searchable via SQL. This article intentionally stayed away from the topic of indexing class collections due to their extreme variety: embedded, stored, streams, user-defined, with collections of collections and others. Besides, it's not always convenient to work with them via SQL, so the author assumed that such collections wouldn't be required in the overwhelming majority of situations, except for some very rare cases. The article also doesn't cover full-text search, since it's a separate topic with its own indexes and approach to working via SQL. Finally, the author omitted the examples of using such property parameters as SqlListType and SqlListDelimiter, but a curious reader will definitely find a way to try them in action.

Useful links:

- Collection Classes
- Indexing Collections
- Collection Indexing and Querying Collections through SQL
- FOR SOME %ELEMENT
- The Object/Relational Connection
- Extensions To SQL
- Special Features

This is a translation of the following article. Thanks [@Evgeny Shvarov] for the help in translation.

This post is also available on Habrahabr[ru].

#Indexing #Object Data Model #ObjectScript #Performance #SQL #Caché

Source URL: https://community.intersystems.com/post/indexing-non-atomic-attributes