Optimization

Presented by: **Thibaud Arguillere**

INTRODUCTION

We, developers, always want our apps to run fast. As fast as possible. Light speed would be cool eventually.

This session is about optimization, about things that can make your application run faster.

- Things that can make your application run faster, from "a bit fast" to "my god, are you sure the code did run?"
  - Some of them require you to change the code
  - Others are, somehow, free regarding the need of code modification

```
Things that can slow down an application and thus should be avoided.
```

We are going to mix tips and tricks – I do love tips and tricks – and fundamentals, especially when we'll be talking about data access.

What this session is not about

- Design, choose the correct database schema and such things (we'll talk about that, but just for a minute)
- How to use 4D shortcuts and functionalities in the method editor of the form editor (we'll remind some points here or there, but it's not the main subject)
- Optimize code writing and maintainability (this actually will be a side effect, sometimes a good one, sometimes a, well, less-good one)

To illustrate how much faster optimized code run compared to the "old" one, we have made benchmarks. But those benchmarks are just like any other benchmarks in the world: Done on a specific machine, with specific CPU, speed, RAM, drive, … For client-server testing, we did not launch 10, 100, 300 clients but max 3. Sometimes on Mac, sometimes on Windows. So, at the end, you must be aware of one important thing: The improvement you'll notice in your development environment may be different at the customer site, in a real life situation. We can state that if it runs faster with you it *should* run faster elsewhere. But there are four main possibilities once the application is installed on site and you compare to your own private results:

- No change
- Run faster but not really evident
- Run much faster
- Run slower

Obviously, the last item is a problem. It can happen if, for example, the changes you made require more memory, more disk space, …

So, when we say "this code is nnn% faster than that code", it's something you want to check yourself in your appropriate environment.

DO YOU REALLY WANT TO GO FASTER?

This is probably the main, major, most important question you should – actually, you *must* – ask to yourself:

Is There Really a Need For Speed?

Which can be translated to:

Who Is Complaining About The Current Version Of Your Application?

This point is definitely a major, relevant point: Do the users complain? Do the customers complain?

If nobody seems to notice that this or that part of your application is slow, why should *you* care? Let's take a basic example. It's kind of evident; but it's good to rely on common sense. Say your application displays a list of invoices. A click on a button calculates miscellaneous amount on that list. It takes 1-3 seconds, depending on the number of rows in the list. A user clicks this button maximum once a day, and never reported to anybody this operation was slow. On the other side of the computer, *you* run that code hundred of times, for testing and debug purpose, and *you*, alone at your desk complain about those 2 seconds.

Now the question is: Do you really want to spend hours optimizing this part? You will be so happy to reduce the time by 500%, displaying the results about immediately. But the user will not even notice the difference. And will not thank you for this, which is sad for your ego. At the end, it would have been much better to spend this time in something else: New feature, debug somewhere, go to the gym, etc.

We are not saying this part should never be optimized. We are saying there's no hurry. If you know this button will eventually be used hundreds of times by hundreds of simultaneous connected users, while it is currently hit by one user over a maximum 10 connected users, then maybe be it could be a good idea to try to optimize its code.

For now, we'll consider that you need to optimize some part of your application.

THE LAZY WAY

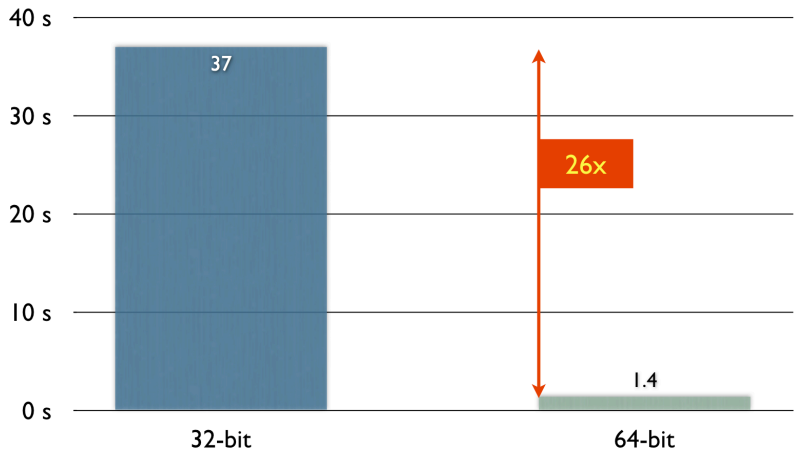
You can be lazy if the conditions for laziness are here.

Basically, and we'll say more about this point in a few moments, optimizing parts of an application mean writing code. Which can take time and may have side effects. There is one easy way to make an app run faster:

Upgrade the computer

No need to spend a lot of time on this point. Here is a quick summary. The three main parts of a computer that will speed up your applications are:

- Memory. Look at this sample benchmark extracted from 4D website regarding 64-bit ("Queries under 4D Server v12 64-bit", in http://www.4d.com/support/resources.html). By just moving to 4D Server to 64-bit (and allocating enough memory), we can speed up a lot the cache access:



- Hard drive. Move to SSD! SSD is much, much, much faster than the classic hard drive. There are some points to be aware of about SSDs (such as the implementation of the TRIM command in the

OSs). But in a test we made to compare the time it takes to flush the cache of a huge fragmented database (the database was created for the test), *the SSD was 35 times faster*. The flush window did not even show.

- CPU. If your application runs in standalone mode, move to a faster computer (more processing power). If it runs on the same machine as other applications, think about increasing the number of cores. If it runs 4D server and more and more clients are connected, upgrade the number of cores.

THE CODING WAY: PARETO?

The 80/20 rule (or 20/80) has a name: The Pareto Principle. Here is its definition, extracted from Wikipedia (http://en.wikipedia.org/wiki/Pareto_principle):

"The Pareto principle (also known as the 80-20 rule […]) states that, for many events, roughly 80% of the effects come from 20% of the causes."

If you read the article a lot of things will look very familiar, such as:

> "80% of your revenues come from 20% of your customers."
>
> "80% of your complaints come from 20% of your customers."

Applied to code optimization, there is bad news…

> *80% of your time will be spent on 20% of the program functionalities*

…and good news:

> *20% of the code is responsible of 80% of the result.*

Listen to Microsoft itself (http://www.crn.com/news/security/18821726/microsofts-ceo-80-20-rule-applies-to-bugs-not-just-features.htm;jsessionid=WeGqcpuOkU+OWhGyf4iakw**.ecappj02):

> *[…] by fixing the top 20% of the most reported bugs, 80% of the errors and crashes would be eliminated.*

So, what can be learned with this principle?

Well.

It is likely that the time it takes to optimize your application is much higher than the time it takes to just make your application run.

THE STRANGE ADVICE (GIVEN THE "OPTIMIZATION" TITLE)

Sometime, optimizing is an easy process, with about no side effects. A quick example is when for some business logic your customer needs to do queries on none indexed fields. The fields were not indexed because at the time the application was first created, those fields were just informative. Now, the user needs them and doesn't want to wait seconds each time he queries on the table. In this situation:

- "is an easy process": Just index the field(s)
- "about with no side effecst": Why "about"? Because you must take care of disk space (which is probably not a serious problem as of today) and memory in the cache. If you add indexes on 3 string fields of a table that contains millions of records, then you must increase the cache size, and probably the RAM. Eventually, move to 4D Server 64-bit.

Another example is when you replace your 4D code by a new 4D command, which runs faster. Here, you won't have side effects and you made your code easier to read. We'll see some examples of this later.

If the optimization requires that you change your code, or even worse that you change it *a lot*, then a good way to optimize is to, well, not optimize:

Don't Optimize Your Code

Because it may happen that optimizing your code leads to denormalization or breaks generic behavior and reusability. Which means that your code may become more complex, less readable, more difficult to maintain, etc. by the end

Optimizing Your Code May Optimize Your Bugs

By "optimizing your bug", we mean that bugs themselves become more complex, more difficult to find, hidden behind complex, poorly documented code.

Another case why you don't really want to optimize your code is when – and we talked about that a few moments ago – it is already fast, but you want it to run faster and faster. If your loop takes 4 ms to complete and you boost it by 50%, it will run in 2 ms. You saved 2 ms each time the routine is called. Great job. To save one second, the routine must be called 1,000 times. Not saying this isn't worth it. Maybe the routine is called millions of time in a client-server environment. Just saying that common sense should lead the decision.

From now, we'll explain or just mention how an application can be optimized for speed.

RUN COMPILED

Obvious, isn't it?

There is no need to make benchmarks comparing speed execution of compiled code *vs* interpreted code. Actually, we did a lot of them, but real life applications speak for compiled code by itself.

In our knowledge, developers want to run their 4D application interpreted for the following reasons:

- This is a private development. For example, a 4D developer develops the CRM for the company he works for. Here, the development somehow never ends and running interpreted makes bug-fixing and implementing easier: no need to quit the clients, to quit/restart 4D, server, etc.
- For the same reasons, applications are deployed interpreted, and the developer connects to it via VPN or directly with 4D Client. Again, bug fixing and implementing may be a good reason
- Sometime, the database just doesn't compile. The classical good old code written since 15 years by 15 different developers, each implementing a feature in a hurry.

Compiling an application sure leads to speed improvement, but may also be time consuming: It sometime is very hard to debug.

MY FIRST OPTIMIZATION: STRICT-COMPARE STRINGS

Say you need to "strict-compare" (diacritical and case sensitive) 2 strings. You may already have such method. Most of the time, spontaneously, the comparison is made char by char, comparing the character code. Here is the typical code, which starts by checking the length of the strings, then run a loop for each char:

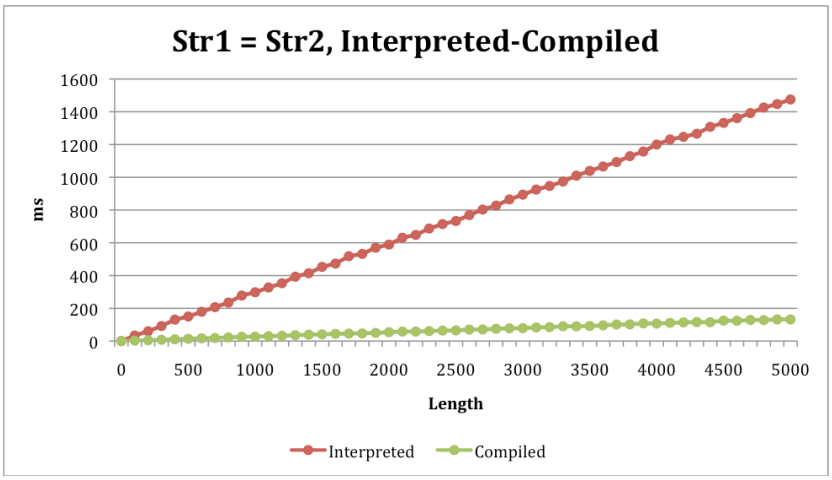
```
C_TEXT($1;$2)
C_BOOLEAN($0)

C_LONGINT($i;$L_max)

$L_max:=Length($1)
```

```
If (Length($2))
	$0:=True
	For ($i;1;$L_max)
		If (Character code($1[[$i]])#Character code($2[[$i]]))
			$0:=False
			$i:=$L_max
		End if
	End for
End if
```

Compile

If you just compile this code, you'll see a major improvement. Here, we run the test for a string which length is increased from 0 to 5,000 (by 500). For each test, we have run the comparison 100 times, and the amount of milliseconds is the total amount of time for the whole 100 iterations (for each string length).

This graph speaks by itself:



Can't run compiled?

In interpreted mode, there is still room for improvement. The idea is remove the string from the algorithm. This may sound strange, but we are in a particular case: We want every character of both strings to be exactly equals. So, instead of calling hundred, thousands of time the `Character` code function, let's compare bytes in a blob. Here the algorithm becomes the following:

1/ Put the strings in 2 BLOBs

2/ Compare each byte of the BLOBs

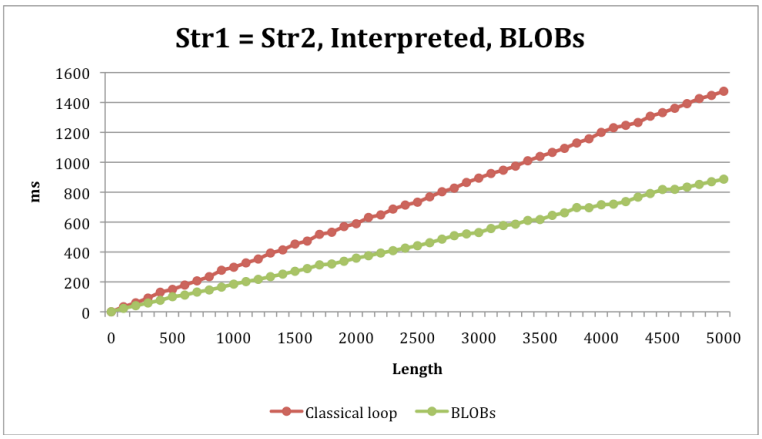
As the strings are UTF-8 encoded, they may have the same Length as strings, but not the same final count of bytes. A quick example: "senior" and "señior" both have a length of 6, but once put in a BLOB as UTF-8 without length, first blob has a size of 6 bytes, but the second one has a size of 7 bytes because the "ñ" Unicode character has been encoded in 2 bytes (basically, on for "n", one for "~". A bit tells the string both bytes are bound).

That is why, after saving the strings in BLOBs, we check the size:

```
C_TEXT($1;$2)
C_BOOLEAN($0)

C_LONGINT($i;$L_max)
C_BLOB($blob1;$blob2)

$L_max:=Length($1)
If (Length($2)=$L_max)
	$0:=True
```

```
        TEXT TO BLOB($1;$blob1;UTF8 text without length)
        TEXT TO BLOB($2;$blob2;UTF8 text without length)
        $L_max:=BLOB size($blob1)
        If ($L_max=BLOB size($blob2))
                $0:=True
                For ($i;0;$L_max-1)
                        If ($blob1{$i}#$blob2{$i})
                                $0:=False
                                $i:=$L_max
                        End if
                End for
        End if
End if
```

Here are the results (again, 100 iterations each 500 new characters):



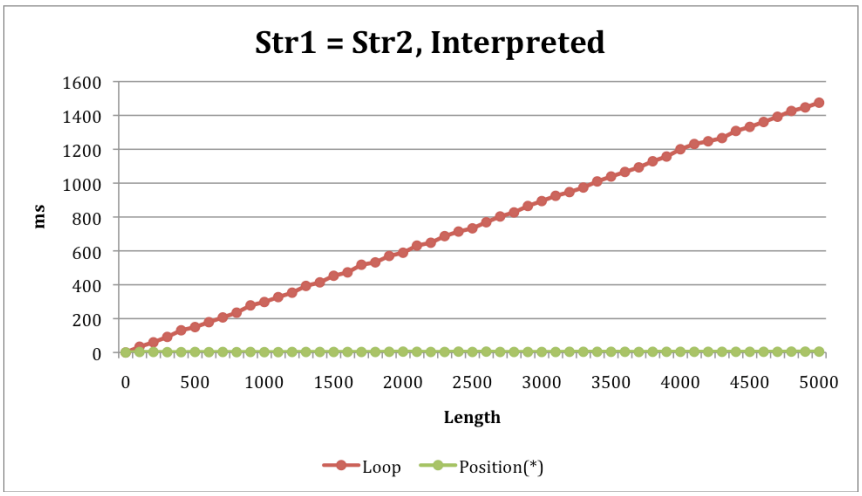
Optimize Always (Interpreted and compiled)

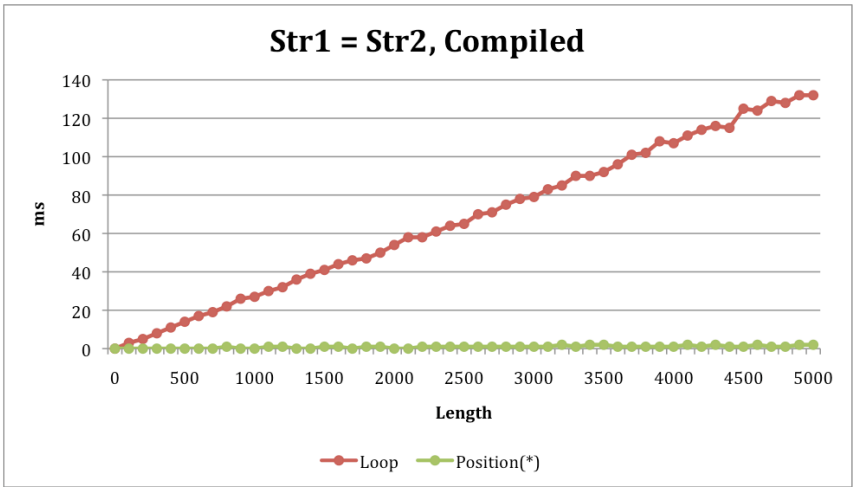
Since v11, you can – among thousands other functionalities - add a star (*) to the **Position** command. When you do this, 4D automatically compares the strings in a strict way, which is exactly what we want. So, now, we write a code that contains less lines of code and is easy to read:

```
C_TEXT($1;$2)
C_BOOLEAN($0)

Case of
  : (Length($1)#Length($2))
  : (Position($1;$2;*)#1)
Else
  $0:=True
End case
```

What this code does is: (1) still checks length are the same and (2) if yes, then call Position-with-star. If the strings are equals, `Position` will return 1.

When you run this code, you're dramatically faster. Basically, the longer the string is, the faster `Position(*)` is compared to the loop.

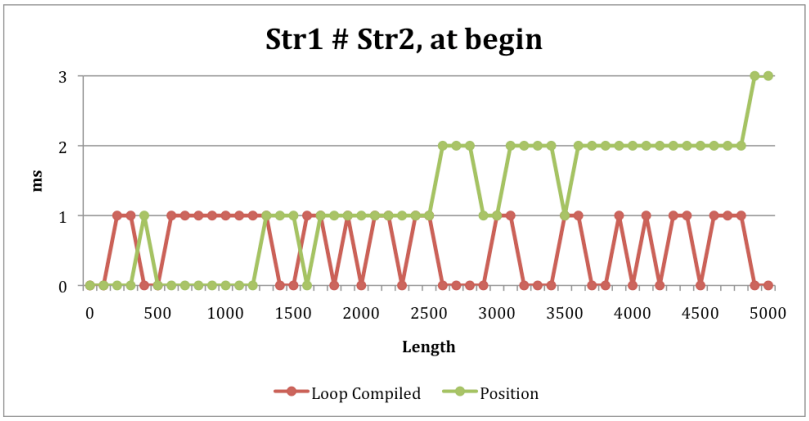




The compiled test is still faster than the interpreted one, but this is mainly because of the test condition: The loop iteration is faster in compiled mode.

What If Strings Are Different?

If the two strings are different but have the same length, the loop may be faster if the difference is found at the very beginning of the string. For example, on a 5,000 characters string, if first chars in the strings are different, then the loop stops immediately. In this situation, the loop may be faster, but this occurs only in compiled mode, not in interpreted:



In this situation, the Grail of optimization would be to have valuable stats about the strings being compared so the algorithm tries to use both ways: Sometimes the loop, sometimes the command.

READ THE DOCUMENTATION

Previous example highlighted the fact that new or enhanced commands in new versions of 4D can help a lot in optimizing your code.

But do *you* read the documentation provided with every new version of 4D? Do you *really* read it from first to last page? And here, we are not only talking about the upgrade manual for major versions, when you move from bv2004 to v11.0 to v12.0 to v13.0. More precisely, we are not talking about those major upgrades, but also about minor updates, when an upgrade is provided (if not provided, it means the version contains only bug-fixes).

- No? You don't read intermediate upgrades, and maybe worse, not even the major upgrade? Then, in 2011 (end of 2011 actually), you still don't know that you can have pointers to local variables since version 2004.2. Too bad.

- Yes? That's good, great job! Just hope you detect where the new things can be used in your code.

In each new version, 4D provides some new commands that, being natively implemented, are faster than your own code. So, if you're looking for performance, then you should look at them.

Let's take a simple example using v13. How do you calculate the sum of a numerical array? Basically, you loop in the array, cumulating each element. In v13, the `Sum` command now works also on arrays (and the other math functions). Let's compare the following two code snippets, applied to a real array of 10,000 elements:

```
  // ============= Code #1, classical loop
C_LONGINT($i)
C_REAL($theSum)

$theSum:=0
For ($i;1;Size of array($theArray))
      $theSum:=$theSum+$theArray{$i}
End for

  // ============= Code #2, v13
theSum:=Sum($theArray)
```

First, notice how simple the code is now.

Now, quick benchmarks. When we run it 100 times, the full time it takes for the loop is 2,373ms vs. 3 ms for the call to Sum. In interpreted mode. When run in compiled mode, the times are respectively 11 and 1 ms.

This is a typical example of a good, easy, interesting, safe, and smart optimization.

ARE TRIGGERS EVIL?

This is a good question.

4D language runs cooperatively. Which means that whatever the count of CPUs, only one at a time is used to run the code.

> **Important (a reminder?)**
>
> In client server-mode, when a client calls a database engine command (such as `QUERY`, `ORDER BY`, or `SAVE RECORD`-with-no-triggers, …) the later is executive preemptively on the server (there are exceptions to this rule. We'll see that later). This means that when, for example, on a four core machine, four clients call QUERY at the same time, basically the four queries will execute simultaneously on the server. While, if the four clients call `GET PROCESS`

`VARIABLE(-1;…)` at the same time, the execution is stacked and will be handled by the server one at a time.

Triggers run the language. On the server, they are executed in the pool of cooperative processes, one at a time. In the previous "Important" note, we wrote "`SAVE RECORD`-with-no-trigger". This is because if the table has a trigger, then when the 4 clients call `SAVE RECORD` at the same time, execution will not be run in parallel. It fells in the pool of cooperative threads, and each trigger will be executed one after the other. With a bit more in-depth knowledge: With today's versions (October, 2011), a process executes for one tick (1/60s), and then 4D gives the hand to the "next" process in the loop.

To highlight the whole thing, lets imagine a table whose trigger takes one tick to complete. If you have 60 clients calling `SAVE RECORD` on this table, then one of them will wait for one full second (60 clients X 1/60 = 60 ticks = 1 second).

The answer to the question "Are triggers evil" is then a bit complex. Triggers are very efficient to centralize the business logic of an application. They also help a lot in maintaining an application (changes in the code are applied to the data, whatever the way this data is modified). On the other hand, being run cooperatively, they can slow down an application. Which means that it depends a lot of the environment: How many simultaneous users, with or without a lot of create/save, etc.

If you really need speed in a trigger, you have two main choices:

- Remove the code in the trigger. Put it in a method, and call this method everywhere at the appropriate place (when you create a new record, before saving a record, …).
- Optimize the code running in the trigger

First solution (remove the trigger) *may* look as being the most efficient in client-server environment. But it is not always the case. And by moving the triggers, you may then have a serious maintenance problem: You must think about calling your method every time you modify (create/save/delete) a record in this table. Will you or your colleague think about that in four months? Not talking about the nightmare you'll have to update your calls to `ARRAY TO SELECTION`. Our personal advice is the following: Triggers are such an effective way to centralize code that you should think twice before removing them.

The second solution looks a bit better. A trigger is the only place where a developer can code without using generic methods (this is not a Golden Rule, just our opinion). This may affect code readability, but if you need those microseconds, then you may do it.

To optimize the speed:

- Remove useless code
  For example, if you used to call `Sequence number` in a trigger, then you should remember that since v11, you could just check the "Auto increment" property of the field. Unfortunately, this is only ok as long as you just use `Sequence number` and you're not using it among other information.

- Don't call generic methods
  Let's take a real-life example, issued from a real-life-in-production application. For some reasons, in a dedicated table, we want to store an MD5 value describing the record. We use a generic method that merges all the fields of the record in a blob (then a MD5 is calculated on this BLOB). Here is the code of the `[Contacts]` trigger:

```
C_BLOB($blobedRecord)
Case of
  : (Database event=On Saving Existing Record Event)
    t_RecordToBlob(->[Contacts];->$blobedRecord)
    // . . . the code continue
```

The code for t_*RecordToBlob* is simple: We loop on each field and use the NNN TO BLOB command accordingly to the type of the field. The following code has been cleanup of all its defensive programming part, and it does not handle subrecords:

```
  // t_RecordTBlob
C_POINTER($1;$toTable)
C_POINTER($2;$toBlob)

C_POINTER($toCurrentField)
C_LONGINT($tableNum;fieldType;$srceSize;$srceOffset;$destOffset)
c_blob($blobedRecord)

$toTable:=$1
$toBlob:=$2
$tableNum:=Table($toTable)

$L_countFields:=Get last field number($toTable)
For ($i;1;$L_countFields)
      If(Is field number valid($tableNum;$i))
            $toCurrentField:=Field($tableNum;$i)
            fieldType:=Type($toCurrentField->)
            Case of
            : ((fieldType=Is Text) | (fieldType=Is Alpha Field))
            TEXT TO BLOB($toCurrentField->;$blobedRecord;UTF8 text without length;*)

            : (fieldType=Is Integer)
            INTEGER TO BLOB($toCurrentField->;$blobedRecord;Native byte ordering;*)

            : (fieldType=Is LongInt)
            LONGINT TO BLOB($toCurrentField->;$blobedRecord;Native byte ordering;*)

            : (fieldType=Is Real)
            REAL TO BLOB($toCurrentField->;$blobedRecord;Native real format;*)

            : (fieldType=Is Boolean)
            INTEGER TO BLOB(Num($toCurrentField->);$blobedRecord;Native byte
            ordering;*)

            : (fieldType=Is Date)
            TEXT TO BLOB(String($toCurrentField->;ISO Date);$blobedRecord;UTF8 text
            without length;*)

            : (fieldType=Is Time)
            LONGINT TO BLOB($toCurrentField->+0;$blobedRecord;Native byte ordering;*)

            : (fieldType=Is BLOB)
            $srceSize:=BLOB size($toCurrentField->)
            if($srceSize>0)
            $destOffset:=BLOB size($blobedRecord)
            $srceOffset:=0
            COPy BLOB($toCurrentField-
            >;$blobedRecord;$srceOffset;$destOffset;$srceSize)
            end if

            : (fieldType=Is Picture)
            C_PICTURE($thePict)
            $thePict:=$toCurrentField->
            VARIABLE TO BLOB($thePict;$blobedRecord;*)

            end case
      End if
End for
$toBlob->:=$blobedRecord
```

Running this 100 times on a record (table: 124 fields, final size of the BLOB: 5,763 bytes) takes 912 ms in interpreted and 219 in compiled mode.

Now, we want to unroll the loop. We know the type of each field, so we don't need to call `Type`. We also use only the valid fields, so we also avoid calling `Is field number valid`. The code becomes (we don't put all the fields here):

```
C_BLOB($blobedRecord)
Case of
   : (Database event=On Saving Existing Record Event)
    LONGINT TO BLOB[Contacts]ID;$blobedRecord;Native byte ordering)// First
    LONGINT TO BLOB[Contacts]CreateStamp;$blobedRecord;Native byte ordering;*)
    TEXT TO BLOB([Contacts]FullName;$blobedRecord;UTF8 text without length;*)
    // …etc…
```

Now, run 100 times on the same record (which generates a 5Kb BLOB), it takes 236 ms (interpreted) or 135 ms (compiled): By unrolling the loop, reducing the calls to 4D commands, the code runs about 4 time faster in interpreted and 2 times faster in compiled mode.

Obviously, you still have to be very careful and to document somewhere that if the table is modified, then its trigger must also be modified.

OPTIMIZE RUNNING CODE

About Generic Code and Cost of Defensive Programming

Generic code is great for coding. Take the example we talked about a few minutes ago about comparing strings with a loop (old fashion) *vs.* comparing strings with `Position(*)`. Having this code in a generic routine makes it very easy to optimize: We just optimize one single routine, and any code that uses it will benefit of the optimization. That is precisely what generic development is made for: Reusability and maintainability.

Now, when you're coding generically, then you should (must, actually) write super-rock-solid code. Which means a generic code should not fail by itself, but it can fail because the caller made a mistake when calling the generic routine. Let's illustrate this with a typical example. You have a routine that returns the sum of a numeric (integer/longint, real) array. It receives a pointer to that array as a parameter and then just loop:

```
C_POINTER($1)
C_REAL($0)

C_LONGINT($i)
For ($i;1;Size of array($1->))
    $0:=$0+$1->{$i}
End for
```

Now, if you call this routine with a Nil pointer, or with a pointer that is not a numeric array, it will fails but it's not a bug in the generic code itself.

That's why we use defensive-programming. We check the parameters are valid. This is not a 4D development specific pattern, it's widely used in every language. The idea is to detect a problem before something really serious happen. This is not always possible obviously…

Here is how one could protect the code execution:

```
C_POINTER($1)
C_REAL($0)

C_LONGINT($i;$L_type)
Case of
  : (Count parameters=0)
      // Alert, assert, ...

  : (Nil($1))
       // Alert, assert, ...

Else
      $L_type:=Type($1->)
      If (($L_type#Integer array) & ($L_type#LongInt array) & ($L_type#Real array))
```

```
            // Alert, assert, ...
      else
            For ($i;1;Size of array($1->))
                  $0:=$0+$1->{$i}
            End for
      end if
end case
```

Here, we tested the code for an array that grows from 10,000 to 100,000 items, by 10,000. For each step, we run each method 100 times. We found that in our environment*, the defensive programming may be up to 30% slower*.

But…

We strongly encourage you to continue using this defensive programming. If you really, really need those milliseconds, then you should try to identify the methods that are the most often used and work only on them. You can have an idea of which method is used very often if you activate the debug log (but then, while the debug log is running, everything will run slower…). Once those methods are identified, you can look at them to see which one can really be optimized.

The %R family

This optimization works only in compiled mode. In your code, you can change the behavior of range-checking at runtime by inserting specific comments:

//%R-	De-ativate range checking
//%R+	Activate range checking
//%R*	Restore range-checking as set in the Compiler preferences

Obviously, one should not deactivate range checking. Range-ckecking tries to avoid memory corruption when you write out of the range of a string or an array. For, example when you call…

```
myArray{100}:=5
```

…while `myArray` has only 50 elements.

This range-checking has a cost, noticeable only in huge loops, heavy array/string manipulation, etc. Here is a sample test code that runs a loop 1 *billion* times:

```
C_LONGINT($i;$ignore;$j;$HOP)
C_REAL($start;$end;$kLOOP_COUNT;$kSIZE)
$kSIZE:=100000
$kLOOP_COUNT:=10000
ARRAY LONGINT($array;$kSIZE)

$start:=Milliseconds
For ($i;1;$kLOOP_COUNT)
      For ($j;1;$kSIZE)
            $ignore:=$array{$i}
      End for
End for
$end:=Milliseconds
```

This code runs in 6,720 ms.

Then we deactivate the range checking:

```
C_LONGINT($i;$ignore;$j;$HOP)
C_REAL($start;$end;$kLOOP_COUNT;$kSIZE)
$kSIZE:=100000
```

```
$kLOOP_COUNT:=10000
ARRAY LONGINT($array;$kSIZE)

//%R-
$start:=Milliseconds
For ($i;1;$kLOOP_COUNT)
	For ($j;1;$kSIZE)
		$ignore:=$array{$i}
	End for
End for
$end:=Milliseconds
//%R+
```

Now, this one billion iterations run in 2,803 ms. 40% faster.

But we had one billion operations. Which is not that huge number actually during a full working day of a user.

We think you should turn range-checking off only when you read the array/the string and/or when you are 10000% sure you'll never try to write outside the array/string range. But if you need those micro seconds, then you have a trick here.

The cost of conditional expressions

Since its first version, the way 4D checks conditional expressions may be a surprise for developers used to other languages such as C/C++/JavaScript/PHP/… In such other languages:

- In an AND (&) test, as soon as an expression returns false, the evaluation stops
- In an OR (|) test, the evaluation stops as soon as an expression returns true

4D, on the other hand, *always* evaluates *all* the expressions. Always. Whatever their results.

So, for example, when you write…

```
If(condition1 & condition2 & condition3 & condition4)
```

…while `condition2` is False, 4D will still evaluates `condition3` and `condition3`.

And in some situations, this can slow down your application. Let's use an example that uses a method as an expression, and this method does a query and returns `True`/`False` depending on existence of records. For example:

```
// hasInvoices(customerID)
C_LONGINT($1;$id)
C_BOOLEAN($0)
$id:=$1
$0:=(Find in field([Invoices]CustomerID;$id)>-1)
```

Now, we uses this method elsewhere:

```
If(<>doThis & <>doThat & hasInvoices)
    . . . code for "all is true"
Else
    . . . code for "at least one is false""
End if
```

Let's say that `<>doThis` is false. Unfortunately, `hasInvoices` will still be run, and in C/S mode, it will send a request to the server, wait for the result, etc. This has a huge cost.

To optimize this code is quite easy: Use a `Case of` statement instead of `If`. You must invert the condition for this to work:

```
$isTrue:=False
Case of
  : (Not(<>doThis))
  : (Not(<>doThat))
  : (Not(hasInvoices))
Else
  $isTrue:=True
End case

If($isTrue)
    . . . code for "all is true"
Else
    . . . code for "at least one is false""
End if
```

Caching Results

This time, we imagine you have a method called in a loop. This method does some calculations, but the values it uses for this calculation don't change during the loop. For example, here we use our *`StringsAreStrictEqual`* method:

```
For($i;1;Size of array($anArray))
      If(StringsAreStrictEqual ([Contacts]Details;[Customers]Details))
            . . . do something with the array element
      Else
            . . . do something else with the array element
      End if
End for
```

Here, we have a `[Contacts]` and a `[Customers]` records loaded. They are not used inside the loop code. The selection of each table is not modified either. In this situation, you have room for improvement: Evaluate the comparison before entering the loop:

```
$stringsAreOK:=StringsAreStrictEqual ([Contacts]Details;[Customers]Details)
For($i;1;Size of array($anArray))
      If($stringsAreOK)
            . . . do something with the array element
      Else
            . . . do something else with the array element
      End if
End for
```

Unroll loops

In previous example we test the condition *inside* the loop. If the condition takes a while to complete (which was not the case in previous example, after optimization), then you should consider to write two loops. One if the condition is true, one if it is false:

```
If(StringsAreStrictEqual ([Contacts]Details;[Customers]Details))
      For($i;1;Size of array($anArray))
            . . . do something with the array element
      End for
Else
      For($i;1;Size of array($anArray))
            . . . do something else with the array element
      End for
End if
```

You can also move a step further and unroll the loops. You already saw this in action when we talked about triggers: What we basically did was to unroll the loop that walks thru every field.

Unrolling a loop will not make sense in every situation and you should focus on methods that are called very, very frequently. You can only unroll loops for which you know the count of iterations. Or you can partially unroll it.

Inlining

Inlining a method means that you put the code of the called method inside the caller. This is a bit like unrolling a loop. It saves time in two main arts:

- The cost of method call
- Eventually, the cost of parameters passing

To evaluate the cost of method call, we just use a test method:

```
// Test_AddOne
theValue:=theValue+1
```

We check the cost of function call:

```
C_LONGINT($i)
C_LONGINT(theValue)
C_REAL($start;$end)

$start:=Milliseconds
For($i;1;1000000)
    Test_AddOne
End for
$end:=Milliseconds
Alert(String($end-$start))
```

Then we remove the function call:

```
C_LONGINT($i)
C_LONGINT(theValue)
C_REAL($start;$end)

$start:=Milliseconds
For($i;1;1000000)
    theValue:=theValue+1
End for
$end:=Milliseconds
Alert(String($end-$start))
```

Here are the results:

	Interpreted	Compiled
Call function	12,420	25
Inline	1,265	2

Now, inlining this sample test method was quite easy. But inlining a full method, with all its code, using objects instead of parameters, etc. has two main side-effects:

The most important one is that you loose its generic part. You may say “again?” because it’s about the same story since the beginning of this doc. But it’s true: What if the method is used in different places? You must then think about updating it everywhere whenever you need to fix a bug or to change something.

Clean-up Your Very Old Code If It Is Still Used

We did an expertise on a customer's site, in 2010. An old application, started with French version 4.0 at the beginning of the 90s, and then moved to 3, 3.5 (intl.), 6 etc. until v11. A lot of different developers worked on this app during those 20 years. It contains lot of different coding styles. And the current developers start to cry and to convulse when you suggest changing old code.

One of their complaints was about "4D is very slow in one of our background process that runs every hour on a dedicated client ". The application had, among other things, a `[Customers]` and `[Messages]` tables. Every hour, a background process was looping on about 1,000 messages to create a selection of the corresponding prospects. The code was written long, long ago:

```
. . . previous code creates a selection of [Messages]
CREATE SET([Prospects];"finalSet")
FIRST RECORD([Messages])
While (Not(End selection([Messages])))
       QUERY([Prospects];[Prospects]ID=[Messages]ProspectID)
       CREATE SET([Prospects];"newProspects")
       UNION("finalSet";"newProspects";"finalSet")
End while
use set("finalSet")
. . . code that handle the found messages
```

Here, some developers forgot to RT()M and/or totally missed the `JOIN` command. Because in this code, for 1,000 messages, the 4D client sends 2 requests to the server. One for `QUERY` and one for `CREATE SET` (because it is a process set). Basically, for those 5,000 messages, the client sends 10,000 requests. The new code:…

```
. . . previous code creates a selection of [Prospects]
JOIN([Messages];[Prospects]
. . . code that handle the found messages
```

…sends one single request.

So the message here is: Don't hesitate to cleanup the old code. Dig into it and face the problems now, before something breaks at the customer's site.

Use Unicode

Since v11, strings are handled with Unicode. And this allows something very interesting: In Unicode mode, the strings are ref-counted. Which means that when you copy a string, you actually don't duplicate it in memory. 4D increments the reference counter ("RefCount) of the original string. This speeds up a lot the copy of strings (and pictures. RefCounting also works with pictures. Not with BLOB). This also speeds up parameters passing: parameters are passed by copy in 4D. Since v11 in Unicode mode, if you pass a picture that uses, say, 1 Mb in memory , along 10 methods then at the 10th one, the memory footprint for the string is still 1 Mb. While in v2004, it would be 10 Mb (one copy in each call).

Also note that 4D will remove the ASCII-mode in a next major version of 4D.

More and more, always more

There is a lot of room; a lot of places where you can make your code run faster. For example, when a method receives a pointer to an array and it does intensive loops and access to the array (via something like `$1->{$i}`), then you'll probably find much faster to:

(1) `COPY ARRAY($1->;$localCopy)`

(2) Loop on this local array

(3) Eventually, if the array was modified, return it with **COPY ARRAY**($localCopy;$1->)

The cost of de-refencing pointer is huge (when done millions of time).

OPTIMIZE DATA/SERVER ACCESS

Until now, we spoke only about optimizing a running code, which means optimizing loops, method calls, etc. But you may also want to optimize data access, and in C/S environment, you want to optimize server requests (the less you access the server, the better it is).

Useless Requests

This part is probably in the top 3 of the reasons why an application runs slow. You (your users, actually) may not notice the problem until you increase the count of simultaneous connected users, or you try to move to a WAN network, etc.

There are a lot of useless requests, but the most frequently seen is FIRST RECORD (or its twin, GOTO SELECTED RECORD) called after a command that has already loaded the first record: QUERY and ORDER BY (and their brothers, QUERY SELECTION, etc…):

```
QUERY([Pizzas];[Pizzas]WithOnions=True)
FIRST RECORD([Pizzas])
. . .
```

So the code calls QUERY , 4D executes the query and returns the selection and the first record (in one request). Then the code immediately calls FIRST RECORD, which leads to a *new* request for loading the first, same record. Useless isn't it?

The same goes for LOAD RECORD. But it's sometime more difficult to track: LOAD RECORD may be hidden in sub methods. So, you have the main method that has already loaded a record. It then calls *Method2* that starts with LOAD RECORD and calls *Method3* that starts with LOAD RECORD, etc. This is where "cleanup your code" also goes.

Also notice that in both cases, you may have a bug: Say you are in read-write access for the table. The record is loaded the first time in write access: it is not locked. It may happen that while calling LOAD RECORD, another process takes the hand and loads the same record. The code returns with a record correctly loaded, but that can't be modified while it could be modified in previous methods. A nightmare to debug…

Scope of Set and Named Selections

This scope has no side-effect in standalone. But it can have a real impact in C/S mode. Basically:

- A *process* set (or named selection) has its name which does not start with "$" or "<>" or "◊", with one exception: the "userset" is a local set while its name does not start with "$" or "<>" or "◊". It is synchronized between the client and the server. So, if you ADD TO SET on the client and USE SET on the server within the same process (in a trigger or a method with "Execute on Server" property checked), you get the correct set.
- A *local* set starts with "$". It is not synchronized on the server and is visible only inside its process
- An *interprocess* set starts with "<>" or "◊", is not synchronized on the server and is visible by every process of the current 4D application.

Each time you call ADD TO SET, or REMOVE FROM SET on a process set, a request is sent to the server. That's why you should use a process set only if you *need* the set on the server. Don't hesitate to rename

your process set, adding a "$" at the beginning of their name if you don't need them on the server. But take care of a side-effect: set operations (union, difference and intersection) must be called on set which have the same scope. So if you rename some set, look in your code if you need to handle that (calling `COPY SET` to use a temporary set)

Also note that when you use a listbox whose datasource is a selection or a named selection, the default name for the set that handles the user selection in the grid are named "ListboxSet1", etc. Which means those sets are process sets. But they sure don't need to have this scope. So, when you create a listbox with this kind of datasource, think about immediately renaming the set (local, probably).

Be Preemptive on the Server

When a global process of a client needs to access records (query, create, delete), it communicates with a preemptive thread on 4D Server. Which means that when several clients do a query for example, this query is run in parallel on the server, using all the available cores. For creation and deletion, this is the same except if the table has a trigger. In this case, the code is run cooperatively on the server. So, to be the most efficient, you should use as much as preemptive command as possible.

- Commands that creates/modify/delete records are run preemptively unless the table has a trigger.
- Commands that just load, query records are run preemptively even if a trigger exists for the table. But there is unfortunately an exception to this rule. `QUERY WITH ARRAY` is still run cooperatively. Even in v13. So if you do a lot `of QUERY WITH ARRAY`, and you think that in C/S, it slows down your application, then you must "unroll the loop", and replace…

```
QUERY WITH ARRAY([TheTable]TheField;theArray)
```

…with a loop on the array:

```
QUERY([TheTable];[TheTable]TheField=theArray{1};*)
For($i;2;Size of array(theArray)-1)
      QUERY([TheTable]; | ;[TheTable]TheField=theArray{$i};*)
End for
QUERY([TheTable]; | ;[TheTable]TheField=theArray{Size of array(theArray)})
```

Do Not Index What Doesn't Need To Be Indexed

An index uses disk space and cache space. It takes tome to load an index in memory, to update it when a record is created/modified/deleted, etc. So, set an index only when you really need it. A typical example is the `ORDER BY` command. When you pass more than 1 parameter to this command (it's explained in the documentation), the sort is always done sequentially and never uses the index. So, if you have a `[Contacts]` table with the `FirstName` and `LastName` fields, and you think you can optimize your **`ORDER BY`**`[Contacts];[Contacts]FirstName;[Contacts] LastName`) command by indexing both fields, you're making a mistake: The sort will be done sequentially. You can safely remove the index on `FirstName`. If the sort is too slow for you, you still have a good solution: Create a composite index that uses both fields.

And more on speeding up data access

These are some tips and tricks about this topic, and any new idea is welcome!