syntegris information solutions

Are regular expressions slow?

Tested, Quantified, Commentary, Recommendations

Author: Sven Weller

www.syntegris.de

About me

SVEN-UWE WELLER

COMPANY syntegris information solutions

BLOG svenweller.wordpress.com

TWITTER SvenWOracle

MAIL Sven.Weller@syntegris.de

HOBBIES ā'pěks (#orclapex) Oracle Learning Explorer Oracle Cloud Infrastructure syntegris information solutions 20 Jahre

UNTERNEHMEN LEISTUNGEN LÖSUNGEN REFERENZEN KARRIERE KONTAKT

Innovativ durch Forschung

WAS WIR MACHEN

FRAGEN STELLEN

FON (06102) 29 86 68
FAX (06102) 55 88 06
INFO@SYNTEGRIS.DE
SYNTEGRIS.DE

Current pet projects

WANT TO KNOW MORE? – TALK TO ME

FLAPS
Flamegraphs
for **A**PEX and **PLS**QL

- Performance analysis tool
- Generate PLSQL Flamegraphs
- Input = dbms_hprof trace
- Output = interactive SVG
- Target group: developers

Human readable audit trail

- based upon Connor McDonalds [Audit Generator]
- add context and subcontext data (tenents)
- automatic FK resolve
- Single audit table only
- JSON to store dependend audit data
- custom hook text builder
- sophisticated generator setup

Motivation

mathguy Jun 13, 2019 4:56 PM (in response to **Solomon Yakobson**)
8. Re: Slow performance when using REGEXP_REPLACE in ORDER BY

LEVEL 16

With or without FBI, if speed is important I wouldn't use regular expressions. This is why I asked the OP all those detailed questions (and I may ask even more, depending on the answers - or if I realize I didn't think of all the relevant questions yet).

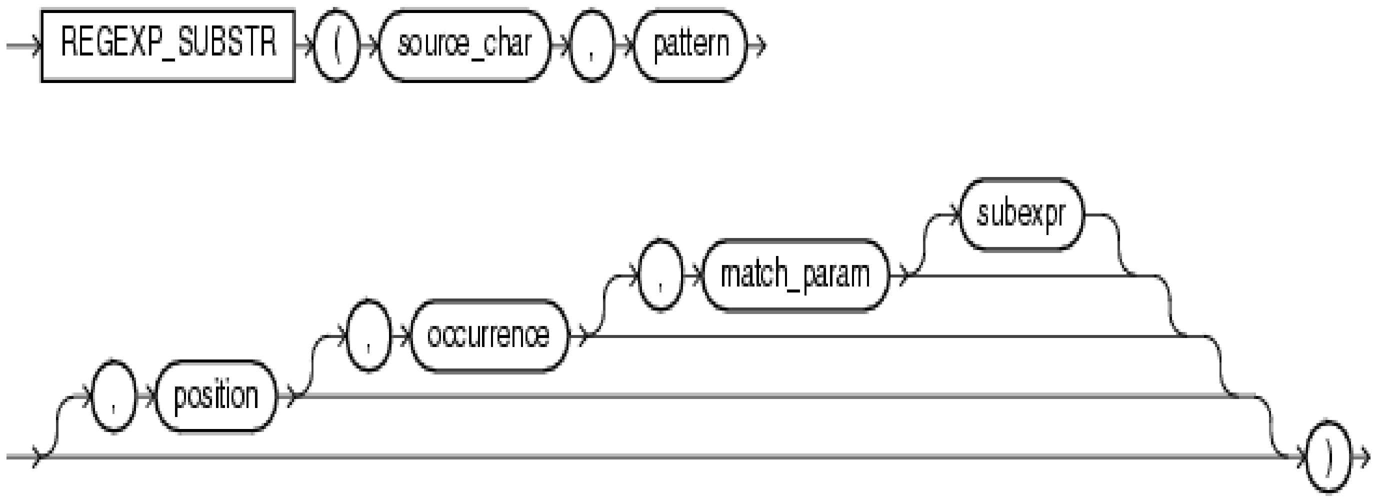
Everybody says that.
Is this is fact?
Is it a myth?
How slow?

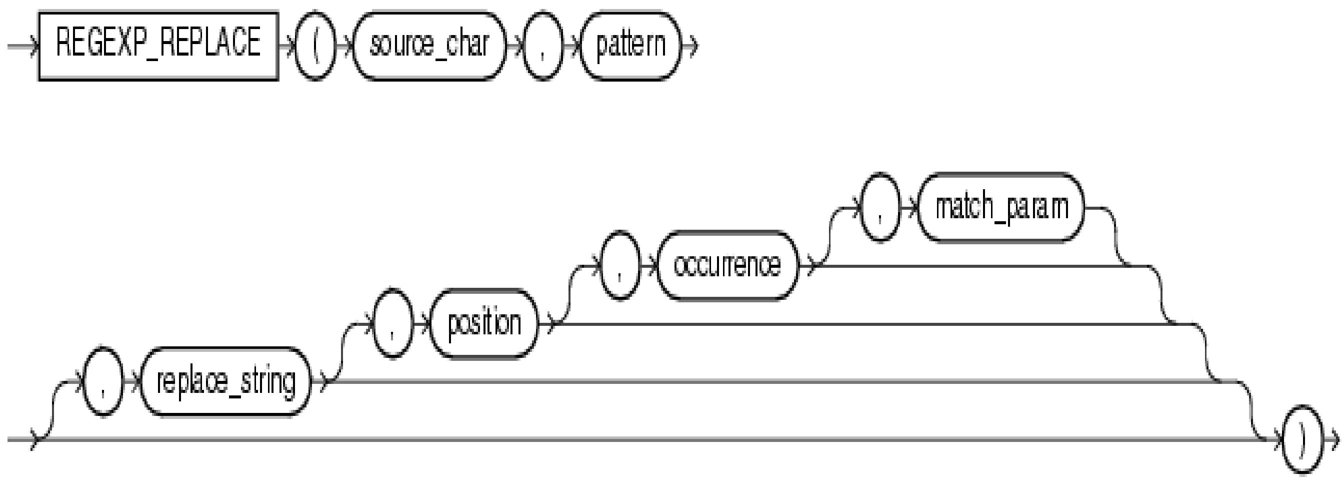
Are Regular Expressions Slow?

Tested, Quantified, Commentary, Recommendations

Oracle Regular Expression

- regexp_count
- regexp_instr
- regexp_like
- regexp_replace
- regexp_substr





Tests Overview

EXAMPLE TEST DATA

100000 rows of random text

```
EIg:629795:WxpRKI
cTSX:304863:xQewB
vZeb:447055:hhbDQv
kPmI:236160:KuM
MEbTTz:390157:AiKI
QeXt:274766:JqD
KBDFb:498907:AZAB
uWOviJ:277196:QeOm
zNRBy:883826:NHhu
IUnAjI:52019:eaFCca
...
```

```
create table testregexp (txt varchar2(1000));

-- add some data
insert into testregexp (txt)
select teststr
from
  (select dbms_random.string('a',dbms_random.value(3,6))
          ||':'
          ||trunc(dbms_random.value * 1000000)
          ||':'
          ||dbms_random.string('a',dbms_random.value(3,6))
         as teststr
  from dual
  connect by level <= 100000
  )
;
```

Tests Overview

GENERIC TEST SETUP

```
-- TEST 0 - base timings without substr | regexp
set serveroutput on
alter session set plsql_optimize_level = 0;
declare
  vTime number;
  vResult varchar2(100);
  vIterations number := 20;
  vtest varchar2(100) := 'Test 0: base time';
begin
  dbms_application_info.set_module('test performance of regular expressions');
  dbms_application_info.set_action(vTest);
  dbms_output.put_line (vTest);
  vTime := dbms_utility.get_time;
  for i in 1..vIterations loop
      for r in (select   SQL   from testregexp) loop
         vResult :=    PLSQL    ;
      end loop;
  end loop;
  dbms_output.put_Line ('Elapsed Time NOTHING = '
                      ||(dbms_utility.get_time - vTime)/100);

end;
/
```

either SQL Expression or PLSQL Expression

Test Size

MASSIVE # OF FUNCTIONS CALLS

100000 strings
x 40 tests
x 20 iterations
x 3 measurements
x 5 environments

6 main test scenarios
Test 0 - baseline
Test 1 - fetch section 1
Test 2 - fetch section 2
Test 3 - fetch section 3
Test 4 - greediness comparison
Test 5 - effect of string size

x 3 different expressions per scenario (substr, regxp_subst, regexp_replace,…)

x 2 SQL vs. PLSQL

> 1 billion function calls

1 billion = 1.000.000.000

Environments

I assume that for regular expressions CPU power might be an important factor, therefore I list the processors for the systems where I know it.

Environment 1: 11.2.0.4 database on an old HP-UX server (CPU unknown)

Environment 2: 18.3 database on an ODA X7-2S
ODA=Oracle Database Appliance.
"S" is the smallest Oracle engineered system you can buy. Excellent performance for your money. The X7-2S version features a 10-core Intel® Xeon® Silver 4114 2.2 GHz processor.

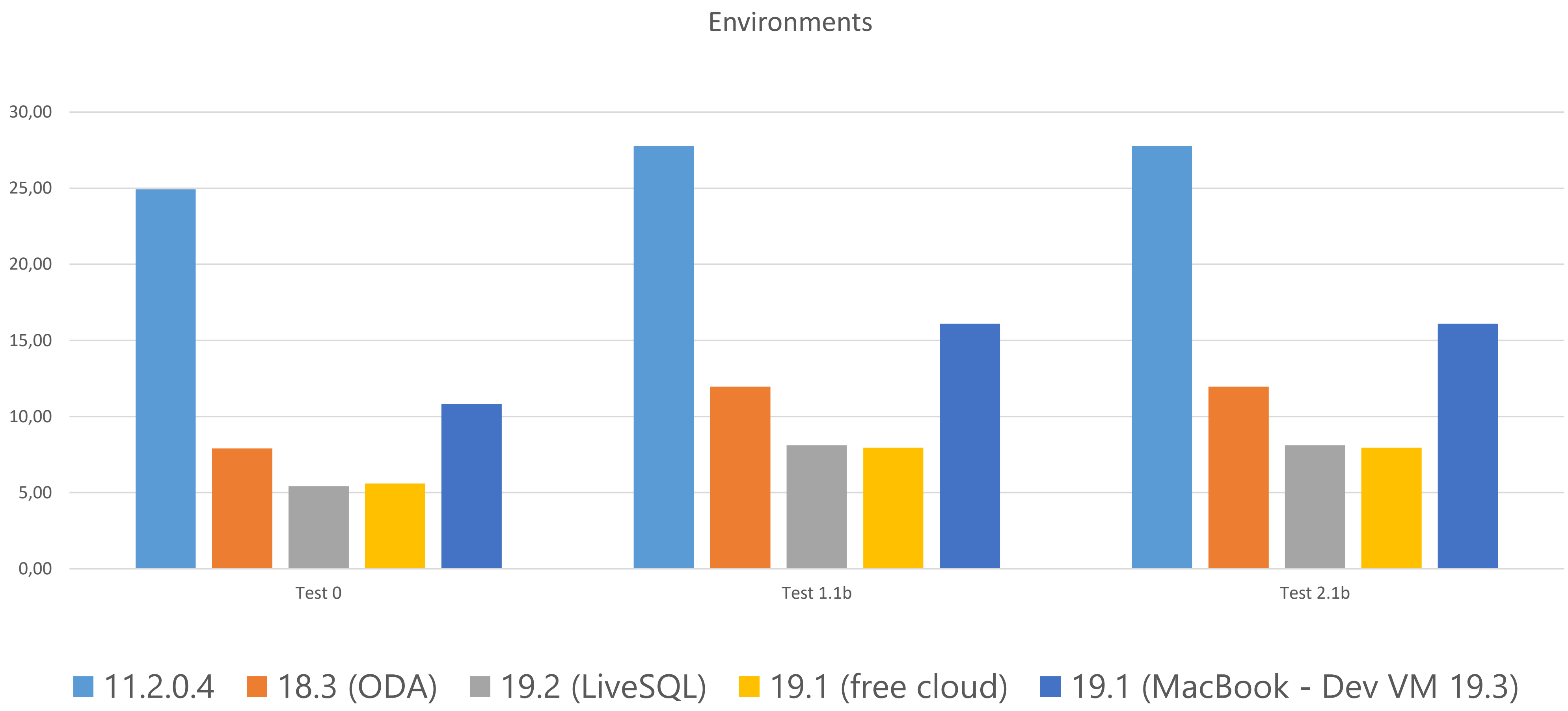
Environment 3: 19.1/19.2 database on LiveSQL (CPU unknown but likely the same as on all OCI)
There were some limitations on LiveSQL that prevented me for running all the tests there. See the environment comparison later on.

Environment 4: 19.1 database on Oracle Cloud - Trial version
I used the Frankfurt Data Center, we get to use 1 or 0.5 OCPUs (Oracle Compute Units) for the trial cloud.
I think at that time an Intel(R) Xeon(R) E5-2699 CPU with 2.20GHz was used.

Environment 5: 19.1 database on my old MacBook Pro inside the Oracle Developer VM 19.3. The MacBook uses a Intel Core i7 2.5GHz. There is some overhead to be expected because of the Virtual Box environment, but this could be similar for the cloud DBs.

Test Results

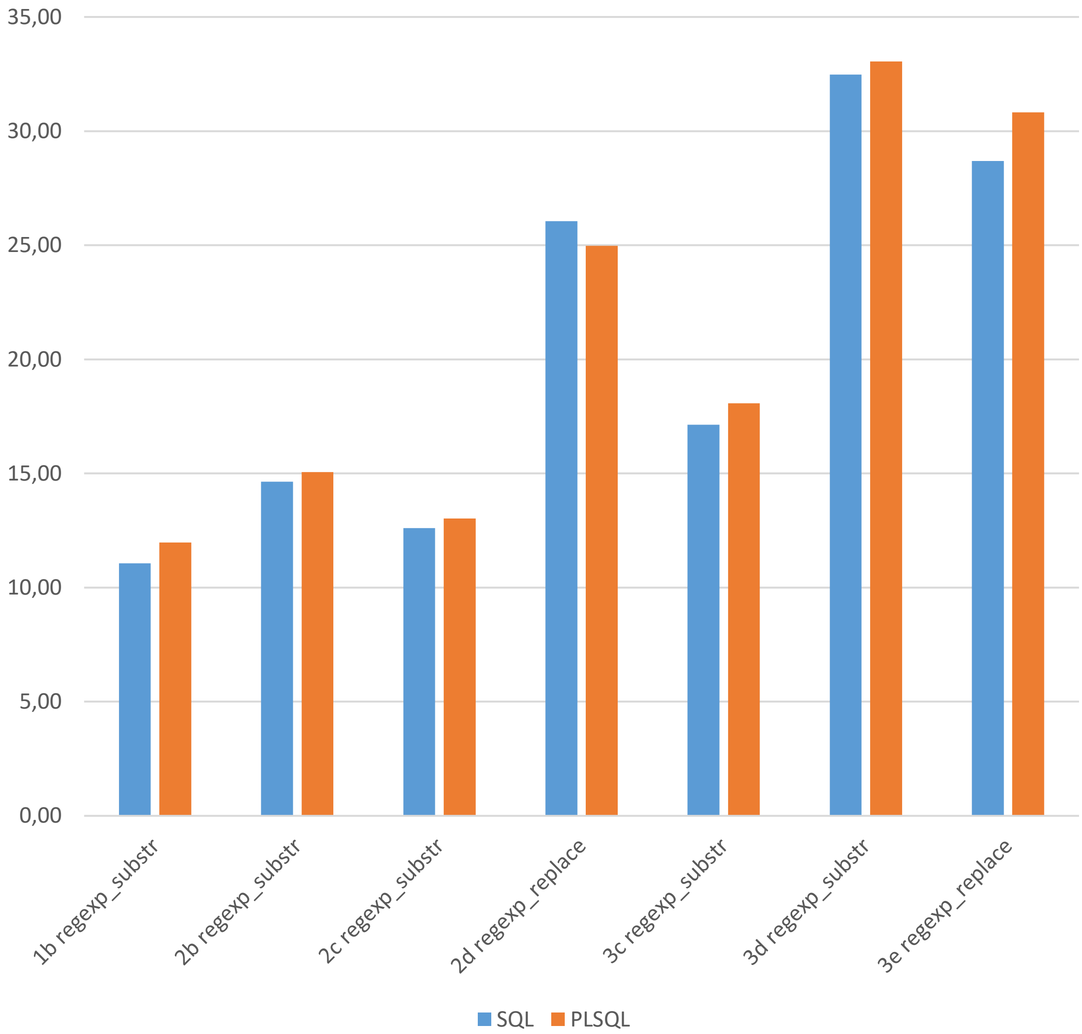
ENVIRONMENT COMPARISON (2019)



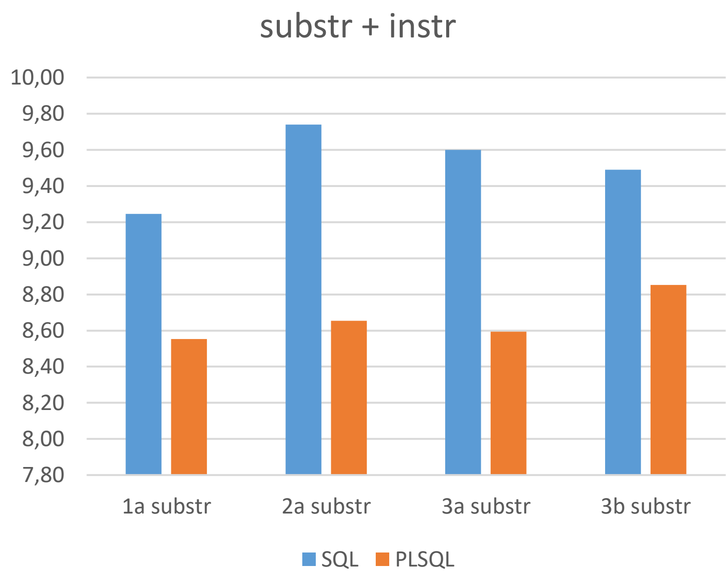
Quick result: In general Oracle Trial Cloud was fastest by a tiny tiny margin, followed by LiveSQL. One could argument that the basetime test had some random fluctuations that made LiveSql look slower than it really was compared to Free Trial Cloud. ODA X7-2S came in strong third. Then after a considerable gap the VM MacBook. By far the slowest was the 11g version on old HP-UX.

Tests Results

WHICH IS FASTER: REGEXP IN SQL OR PLSQL?



Quick result:
Equally fast (for regexp calls)



each bar = 2 million function calls

Tests Results

SUBSTR OR REGEXP?

substr | regexp_substr | regexp_replace

Sample txt

cTSX:304863 xQewB

1a	substr(txt, 1, instr(txt, ':') -1)	0,67 µs
1b	regexp_substr(txt,'^[^:]+')	1,58 µs
2a	substr(txt, vPos1, vPos2 - vPos1)	0,92 µs
2b	regexp_substr(txt,'[^:]+',1,2)	3,37 µs
2c	regexp_substr(txt,'\d+')	2,35 µs
2d	regexp_replace(txt,'^[^:]+:([^:]+):.*$','\1')	9,08 µs
3a	substr(txt, instr(txt, ':',1,2)+1)	0,85 µs
3b	substr(txt, instr(txt, ':',-1)+1)	0,80 µs
3c	regexp_substr(txt,'[^:]+',1,3)	4,62 µs
3d	regexp_substr(txt,'[^:]+$')	12,29 µs
3e	regexp_replace(txt,'^([^:]+:){2}')	10,40 µs

Knowledge gained

REMEMBER THIS

- substr and simple regexp
  are extremly fast
  (substr is faster)

- left anchored (^) is magnitudes faster than right anchored ($)

- regexp_replace usually is slower than regexp_substr

Pictures from wikicommons: https://commons.wikimedia.org/w/index.php?curid=71414633

Knowledge missing

NEW QUESTIONS

Why is the difference between left and right anchored so big?

What are the dangerous expressions?

Regexp concepts

GREEDY QUANTIFIERS

Greedy means "try to grab as much as possible".
Non-greedy means, stop as soon as you encounter something that the next token of the expression matches.

+	one or more, greedy
*	zero or more, greedy
+?	one or more, non greedy
*?	zero or more, non greedy

*There are more greedy quantifiers than just * and + . For example {1,} or even {9} can suffer from the same issues (see backtracking).*

Regexp concepts

BACKREFERENCES - DEFINITION

Backreference means that we can refer back to a specific part (a subexpression) tagged by parenthesis (). It is possible to use a backreference in the expression itself or as part of the replacement parameter in REGEXP_REPLACE.

The second usage is encountered more often.

A backreference has a number which maps to the order of the opening parenthesis "(".

If this is the expression: `'^((\d)+)'` then the backreference `\1` matches all digits at the start of the string. And the backreference `\2` matches a single digit (multiple times).

Photo by Tobias Tullius on Unsplash

Regexp concepts

BACKREFERENCES - EXAMPLES

```
regexp_replace('12345ABCDE67890'
              ,'^((\d)+).*'
              ,'1)=\1; 2)=\2')
```

1)=12345; 2)=5

```
regexp_replace('aabbccccddd1234'
               ,'(.)\1+'
               ,'\1')
```

abcd1234

```
regexp_replace('The The cat and the dogdog played together. '
               ,'(\w+\s*?)\1+'
               ,'\1')
```

The cat and the dog played together.

Regexp concepts

THE PROBLEM: BACKTRACKING

Backreferences and greedy quantifiers can lead to excessive backtracking!

Expression	Text	Regex engine steps
.*.*=.*	x=xxxxxxxxxxxxxxxxxxxx	255

Backtracking story gone bad:

https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/

Regexp concepts

HOW TO AVOID EXCESSIVE BACKTRACKING

possessive quantifiers
Some regexp dialects (not Oracle) allow to use *possessive* quantifiers. A greedy quantifier like .* can be made possessive by adding a + . It means the regexp engine, will not backtrack, after the match. This is useful in some situations to avoid unnecessary backtracking, especially for *not so greedy* expressions like [^:]*+ that are nested inside other subexpressions.
The danger of possessive quantifiers is, that if the expression is copied from a regexp flavor that supports them to a regexp flavor that does not support them, the non supportive engine might fall into the excessive backtracking trap. So instead of keeping the performance in check, the opposite happens. Be aware of such constructs when you copy and adapt an example regexp expression to your system.
For more info about *possessive quantifiers* and *atomic groups* see: https://www.regular-expressions.info/atomic.html

Non backtracking subexpressions
Some regular expression dialects allow to declare non backtracking subexpressions.

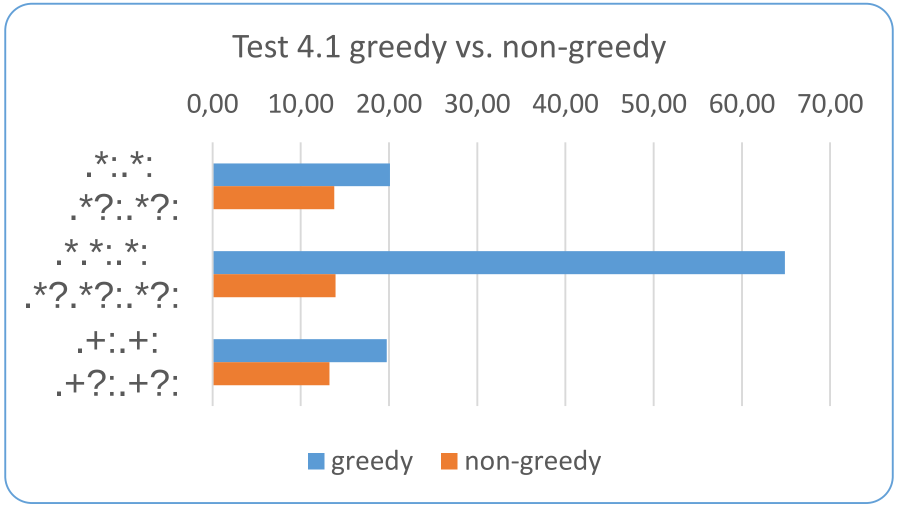
For example MicroSoft .NET
Nonbacktracking Subexpression
The (?> *subexpression*) language element suppresses backtracking in a subexpression. It is useful for preventing the performance problems associated with failed matches.
...
Lookbehind Assertions
.NET includes two language elements, (?<=*subexpression*) and (?<!*subexpression*), that match the previous character or characters in the input string. Both language elements are zero-width assertions; that is, they determine whether the character or characters that immediately precede the current character can be matched by *subexpression*, without advancing or backtracking.
(?<= *subexpression*) is a positive lookbehind assertion; that is, the character or characters before the current position must match *subexpression*. (?<!*subexpression*) is a negative lookbehind assertion; that is, the character or characters before the current position must not match *subexpression*. Both positive and negative lookbehind assertions are most useful when *subexpression* is a subset of the previous subexpression.
https://docs.microsoft.com/en-us/dotnet/standard/base-types/backtracking-in-regular-expressions#Nonbacktracking

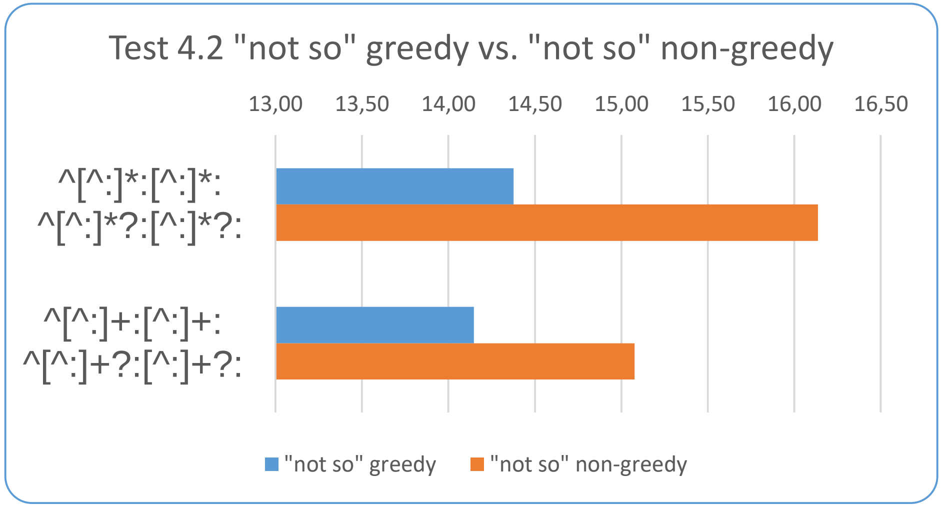
Tests Results

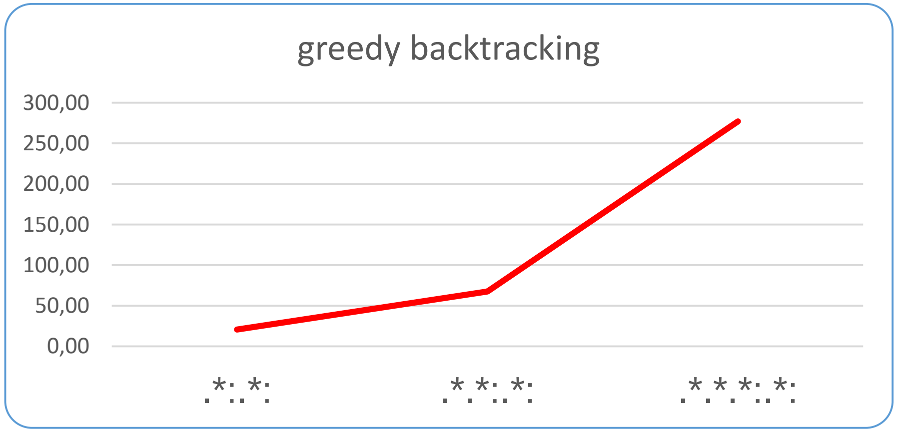
GREEDINESS

greedy .*: non-greedy .*?:

„not so“ greedy [^:]*







.*.*.*:.*: 276 seconds

Tests Results

SIZE MATTERS!

1. When cutting a string from the **left** (anchor "^") the performance degrades - even if the result is the same. This degradation seems to scale in a ***linear* way** with the input string size.

To give an analogy: Imagine eating a hot dog.
It is fast to take a bite when the hotdog is 10 cm long (~4 in).
It is way slower to take the *same* bite when the hot dog is 50 cm long (~20 in).

2. This performance degradation does **not** happen **for CLOBs**.

Analogy: If the hot dog is in an open bun then bite speed depends on hot dog length, but not if the hot dog is in a closed bun (french hot dog).

Both measurements combined give a surprising **break even point**. Left anchored clob expressions outperform varchar2 expressions somewhere before **1000 characters**.

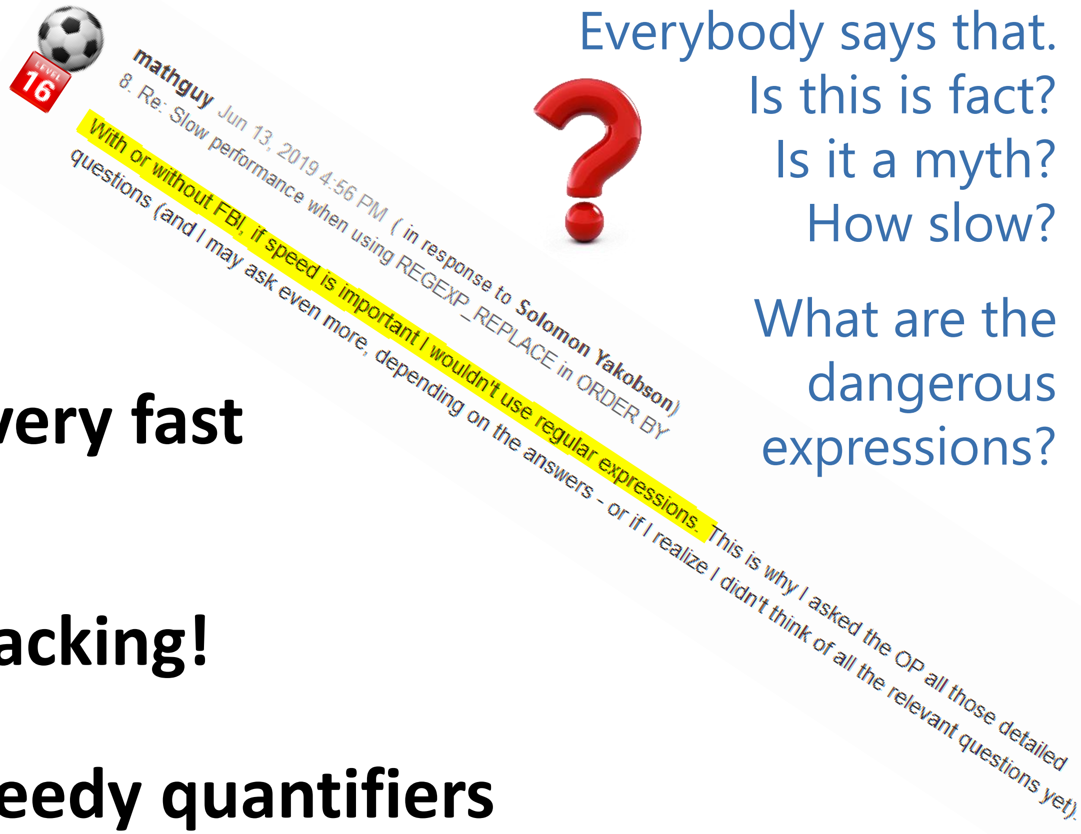
3. When cutting a string from the right (anchor "$") the performance degrades too but much faster. It seems to scale in a *polynomial* or *exponential* way with the string size.

4. This right anchored performance degradation also happens for CLOBs. And is even worse there!

1 bite fast  >>  1 bite slow

Conclusion

ARE REGULAR EXPRESSIONS SLOW?



Everybody says that.
Is this is fact?
Is it a myth?
How slow?

What are the dangerous expressions?

- **Simple expressions are very very fast (a few micro seconds)**
- **Be aware of excessive backtracking!**
- **avoid greedy, use "not so" greedy quantifiers**
- **Input string size is relevant**
- **left anchored expressions are good**