VarnishQ - Sprachbeschreibung

Patrick Krusenotto

2. April 2015

Zusammenfassung

Das Dokument beschreibt die Abfragesprache VarnishQ für Varnish-Caches, die bei IT-Betrieb entstanden ist.

An den Varnish-Caches sind weitgehende Untersuchungen zum Zustand unserer Internet-Software möglich. Dazu müssen die Caches wegen der anfallenden Datenmasse (etwa 8000 Zeilen Loginformation pro Cache pro Sekunde) allerdings qualifiziert befragt werden können. Durch diese Abfragesprache wird dies möglich.

1 Grundlagen

1.1 Notatation

1.1.1 Polnische Notation

VarnishQ arbeitet mit *polnischer Notation*. Eine Funktionsaufruf wird nicht `f(x,y,z)` sondern `(f x y z)` notiert. Hinter der öffnenden Klammer steht immer der Name einer Funktion oder eines Operators. Dazu gibt es keine Ausnahmen.

1.1.2 Groß/Kleinschreibung

VarnishQ unterscheidet bei Namen und Symbolen nicht zwischen Groß– und Kleinschreibung. Bei Strings allerdings schon. In diesem Text sind zulässige Symbole im Fließtext `GROSS` und in Code-Besipielen `klein` geschrieben.

1.2 Requests

Requests sind eine *Folge von Datensätzen*, die einer Live-Beobachtung im Rahmen eins Tests oder einem Logfile entstammen. Die Datensätze selbst haben die Form:

Name	Typ
TAG	*Symbol*
CODE	*Symbol*
DATA	*String*

- **TAG** ist eines der Varnish-Tags wie `RXURL`, `RXHEADER`, `TXSTATUS` usw. *Eindeutige* TAGs kommen pro Request nur einmal vor (wie `RXURL`), *mehrfache* beliebig oft (wie `RXHEADER`)
- **CODE** hat den Wert `B` für *Backend-Request* (Varnish an Backend) oder `C` für *Client-Request* (Client an Varnish).
- **DATA** Der Datenbereich. Er enthält einen String. Bei Headern (mit Tag RXHEADER oder TXHEADER) zum Beispiel etwas wie `"Cache-Control: max-age=60"`

Beispiel

```
TXURL B /cda/fragment/teaser/342987329
```

Hier lautet der *TAG* `TXURL`, der *CODE* `B` und `/cda/fragment/teaser/..` das *DATA* -Element.

Ein Request besteht bei *www.dw.de* aus bis zu 600 solcher Datensätze (bei Strukturseiten etwa), mithin aus 600 Logfile-Zeilen.

Tag	Code	Data
RXREQUEST	C	GET
RXURL	C	\/deutsch\/2013\/twitter-com-33\/
RXPROTOCOL	C	HTTP/1.1
RXHEADER	C	Host: thebobs.com
RXHEADER	C	User-Agent: Mozilla/5.0 (+`http://yandex.com/bots`) From: support@s
RXHEADER	C	Accept-Encoding: gzip,deflate
RXHEADER	C	Accept-Language: tr, en;q=0.7, *;q=0.01
RXHEADER	C	Accept: */*
RXHEADER	C	X-Forwarded-For: 178.154.255.138
RXHEADER	C	Connection: close
VCL_{CALL}	C	recv lookup
VCL_{CALL}	C	hash
HASH	C	
HASH	C	\/deutsch\/2013\/twitter-com-33\/

2 Anwendung

2.1 Request-Streams

Request–Streams liefern eine *Folge von Requests*. Es gibt drei *primäre* Request–Streams:

1. `(live)` liefert die auf dem Server aktuell verarbeiteten Requests
2. `(file "dateiname")` liefert die Requests aus einem File, das mit `varnishlog > dateiname` auf der Konsole erstellt wurde. Dieses Format wird für die täglichen Mitschnitte unter `/web/logfiles/varnish/` verwendet. Pfade müssen voll angegeben werden.
3. `(vlog "dateiname")` dekodiert die Requests aus binären Mitschnitten mit `varnishlog -w` <*filename*>. Pfade müssen voll angegeben werden.

2.2 Starten und Stoppen

VarnishQ ist auf unseren Caches WEBCACHE01-LIVE, WEBCACHE02-LIVE, WEBCACHE11-LIVE, WEBCACHE12-LIVE, und WEBCACHE01-TEST installiert.

Es befindet sich dort unter `/web/apps/varnishq` und kann in diesem Verzeichnis aus der Bash mit `./varnishq` gestartet werden. Beenden erfolgt mit `(exit)`.

2.3 Requestauswahl mit `WHERE`

Requests beschreiben den Ablauf einer erfolgten Requestverarbeitung. Mit `(where` <*where-bedingung*> <*request-stream*> `)` kann aus einem Request-

Stream ein neuer Request-Stream gebaut werden, der nur die Requests enthält die einer Bedingung genügen.

Jedes eindeutige Tag kann in der Bedingung direkt verwendet werden. Der Ausdruck

```
(where (matches "/a-" rxurl) (live))
```

liefert einen neuen Request-Stream, der diejenigen Requests enthält, bei denen der mit RXURL getaggte Datensatz im DATA-Bereich den String `/a-` enthält (Zugriffe auf DW-Artikel).

Bedingungen können mit `(and..)`, `(or..)` und `(not ..)` verbunden werden. Für mehrfache TAGs gibt es den Quantor `(any..)`. Um allein das *Vorkommen* eines bestimmten Tags zu prüfen, kann es einfach genannt werden:

```
(where (and hit (any (matches "akamai" rxheader))) (live))
```

Dieser Stream fragt nach Requests, die das Tag HIT enthalten und bei denen irgendeine der Headerzeilen den String `akamai` enthält. Für die (mehrfachen) Header-Tags `RXHEADER`, `TXHEADER` und `OBJHEADER` gibt es eigene Funktionen `(rxheader` <*string*> `)`, `(txheader` <*string*> `)` und `(objheader` <*string*> `)`:

```
(where (contains (rxheader "Host") "thebobs") (live))
```

Dier Stream liefert alle Requests, die an die Bobs gegeangen sind.

Reguläre Ausdrücke (Perl-kompatibel) werden mit `(matches` <*regex*> <*datum*> `)` angewendet. Der `\` muss gedoppelt werden, also `\\d+` statt `\d+` für eine Folge von Ziffern.

2.4 Requests ausführen

Ein Request–Stream kann mit `(dmp ..)`, `(dump ..)` oder `(take` *n* <*quelle*> `)` befragt werden werden. Ist in dem Stream eine Query kodiert, so wird sie „ausgeführt”:

```
(dmp (where (contains rxurl "/s-") (live)))
```

Diese Eingabe füllt den Bildschirm mit den Requests an dw.de-Strukturseiten

2.5 Zeilenauswahl mit `SELECT`, `SELECT-TAG` und `SELECT-TAGS`

Man möchte nicht immer alles von einem Request zu sehen bekommmen sondern ist nur an dem URL, der Frage ob HIT oder MISS, dem verwendeten Backend, dem Hostnamen, dem HTTP-Return-Code, dem Referrer oder an Headern mit bestimmten Werten interessiert. Dazu dienen die select-Anweiusngen.

`SELECT-TAG` liefert nur Datensätze mit einem bestimmten Tag. Die Query

```
(dmp (select-tag 'rxurl (live)))
```

gibt nur den RXURL aus. Das Häkchen ist obligatorisch. *Insbesondere ist es auch kein Tippfehler: Es steht nur vor dem Symbol ein Häkchen.* Bei Backend-Requests erfolgen in diesem Besipiel nur Leerzeilen, da diese kein `RXURL` haben. Der Output wird also schöner, wenn man von vorne herein nur Client-Requests zulässst:

```
(dmp (select-tag 'rxurl (where (type-is 'c) (live))))
```

`select-tags` (plural) liefert mehrere Datensätze:

```
(dmp (select-tags '(rxurl reqend) (live)))
```

`select` wählt aufgrund einer Bedingung aus. Diese darf die sich auf die Datenelemente TAG CODE und DATA beziehen.

```
(dmp (select (contains data "english") (live)))
```

Auch hier darf mit `AND`, `OR` und `NOT` gearbeitet werden.

Alle Datensätze, die „english" enthalten oder deren Tag `RXURL` lautet:

```
(dmp (select (or (eq tag 'rxurl) (contains data "english")) (live)))
```

Dabei dient `(eq ...)` dem Tag-vergleich („equal"). `select`, `select-tag` und `select-tags` liefern ihrerseits *Request–Streams*. Also können sie gemeinsam mit `WHERE` in beliebiger Folge verschachtelt werden:

```
(dmp (select-tag ... (where ..(select .. (where.. )))))
```

2.6 Datenisolierung mit `NTH-NUM` und `NUMB`

`(nth-numb` <*n*> <*string*> `)` isoliert die *n*-te Zahl aus einem String. Die Verabeitungsdauer eines Requests kann dem TAG `REQEND` entnommen werden:

```
(dmp (where (> (nth-num 4 reqend) 2) (live)))
```

liefert die Requests, die länger als 2 Sekunden Render-Zeit hatten.(Die vierte Zahl von `reqend` ist die Laufzeit).

`(numb` <*regex*> <*string*> `)` liefert eine Zahl aufgrund eines Patterns. Zum Beispiel ist

```
(numb "max-age=(\\d+)" (txheader "Cache-Control"))
```

die *max-age-Angabe* des Servers für den betreffenden Request.

2.7 Auswertungen mit `CHART` und `AVG`

Zur Erstellung eines Charts gibt es `(chart` *<anzahl-sätze>* *<länge>* *<ausdruck>* *<quelle>* `)`.

```
(chart 1000 10 (rxheader "Host") (live))
```

erstellt ein Chart der zehn häufigsten Vorkommen der Hostnamen aus den 1000 nächsten Zugriffen auf das Live-System. Durchschnitte werden mit `(avg..)` berechnet. Die durchschnittliche Verarbeitungszeit von 100 Requests, die kein Hit sind, findet sich mit

```
(avg (nth-numb 4 reqend) (where (not hit) (live)))
```

2.8 Hilfe

Viele Operatoren sind dokumentiert. Dokumentation zu `where` lässt sich zum Beispiel abrufen mit

```
(describe 'where)
```

Eine kürzere Ausgabe liefert

```
(documentation 'where 'function)
```

2.9 Scripting und Client-Server-Betrieb

Unter `/web/apps/varnishq` befindet sich `vq-script`. Damit kann mit von der Shell aus eine Query aus einem Textfile abgesetzt werden (`./vq-script` <filename>).

2.9.1 Remote Sitzung

Im gleichen Verzeichnis ist `vq-server`, das es gestattet, eine remote Sitzung von einem lokalen Emacs aus zu starten und damit mehrere Caches gleichzeitig zu überwachen. Dazu ist ein lokaler Emacs auf dem Client-PC und ein ssh-Tunnel von Nöten:

Die Einrichtung erfolgt so:

1. Putty SSH Tunnel errichten
   - Server (WebcacheXX-TEST/LIVE) eintragen
   - Unter Menüpunkt *SSH Tunnel*
     - Source-Port: 4005
     - Destination: localhost:4005
     - ÄDD"Drücken
     - Connectäusführen

- über die Console einloggen

2. Im nun offenen Putty-Terminal Server starten:

```
./vq-server

;; Swank started at port: 4005.
```

3. Im Client Slime-Sitzung öffnen und über den Tunnel mit dem Server verbinden M-x slime-connect, Server: 127.0.0.1, Port: 4005

   Jetzt kann vom PC aus der Webcache überwacht werden, ohne die Prozessoren des Webcaches zu belasten.

3 Praxisbeispiele

Folgende Queries sind so oder ähnlich in der Praxis schon relevant gewesen

3.1 Charts

3.1.1 Returncodes der Backends

```
VARNISHQ> (chart 1000 10 rxstatus (where (type-is 'b) (live)))
554 55.4       200
352 35.2       502
39  3.9        301
25  2.5        500
14  1.4000001  404
7   0.70000005 302
5   0.5        403
4   0.4        400
```

3.1.2 Anteil der wiederverwendeten TCP-Verbindungen

```
VARNISHQ> (chart 1000 10 (txheader "connection") (where (type-is 'c) (live)))
576 57.6      close
418 41.800003 keep-alive
6   0.6       NIL
```

3.1.3 Strukturseiten, die allein aus dem Cache bedient werden konnten:

```
VARNISHQ> (chart 100 2 (null backend) (where (matches "/s-" rxurl) (live)))
69 69.0 NIL
31 31.0 T
```

3.1.4 Welche Teaser liefern die häufigsten Fehler

```
(chart 1000 30  (nth-num 0 txurl)
       (where (and (type-is 'b)
                   (not (member rxstatus '(200 301 302 400 403 404 500))))
              (live)))
```

```
52 23308996
52 23281961
27 23260699
27 23260698
27 23260696
24 23331501
22 23290265
17 23149812
17 23331740
16 23178486
15 23326280
15 23324471
14 23280573
14 23280570
14 23280568
14 23280567
14 22983799
14 22983800
14 23326775
14 23331953
13 23268300
12 23304107
12 23268299
12 23148688
12 23322383
12 20893911
12 20893916
11 23319408
10 23329334
9  23324846
```

3.1.5 Welcher Anteil der Client-Requests is von "Bad Gateway-Meldungen betoffen?

```
VARNISHQ> (chart 1000 2 (any (contains "Bad Gateway" objresponse)) (where (type-is 'c
'varnishlog' closure abandoned
878 87.8 NIL
```

```
122 12.2 0
```

Antwort: 12%

4 Performance

Varnishq schafft rund 150000 Requests/s. Für jedes Gigabyte Logfile sind etwa 3 Minuten Verabeitungszeit anzusetzen. Hängt aber von der Query ab. Bei verknüpften Bedingungungen sollten die einfachsten so weit wie möglich vorne stehen. Also lieber `(where (and hit (contains (rxheader polish"))) (live))` als `(where (and (contains (rxheader polish")) hit) (live))`.

5 Sprachreferenz

5.1 BNF

5.1.1 Tags

```
<mehrfach-tag> ::= GZIP | OBJRESPONSE | OBJPROTOCOL | BACKEND
                   | HIT | VCL_ACL  | OBJHEADER | TTL | TXHEADER
                   | VCL_RETURN  | HASH | VCL_CALL  | RXHEADER

<einfach-tag ::= WORKTHREAD | TXURL | TXSTATUS | TXRESPONSE |
                  | TXREQUEST | TXPROTOCOL | STATSESS
                  | SESSIONOPEN | SESSIONCLOSE | RXURL
                  | RXSTATUS | RXRESPONSE | RXREQUEST
                  | RXPROTOCOL | REQSTART | REQEND | INTERRUPTED
                  | FETCH_BODY  | FETCHERROR | EXPKILL | EXPBAN
                  | DEBUG | CLI | BACKEND_HEALT  | BACKENDREUSE
                  | BACKENDOPEN | BACKENDCLOSE

<integer-tag> ::= RXSTATUS | TXSTATUS | LENGTH

<tag> ::= <einfach-tag> | <mehrfach-tag>
```

5.1.2 Werte

```
<integer> ::= <integer-tag>
              | (nth-num <integer> <string>)
              | (numb <regex> <string>)
              | (epoch <integer>^6 [<integer>])
              | integer-literal
```

```
<string> ::= "irgend ein text"
             | <einfach-tag>
             | (RXHEADER <string>)
             | (TXHEADER <string>)
             | (OBJHEADER <string>)

<regex>  ::= <string>
```

5.1.3 Bedingungen

```
<integer-vergleich> ::= =|<|>|>=|<=|/=

<string-vergleich>  ::= string=
                        |string<
                        |string>
                        |string<=
                        |string>=
                        |string/=

<condition>           ::= <tag>
                        |(<integer-vergleich> <integer>*)
                        |(<string-vergleich> <string>*)
                        |(MATCHES <regex> <string>)
                        |(AND <condition>*)
                        |(OR <condition>*)
                        |(NOT <condition>*)
                        |(ANY <condition>*)

<liste> ::= <mehrfach-tag>
            | '(<atom>*)
```

Die Mit `ANY` eingeleitete Bedingung muss *genau einen Mehrfachtag* enthalten, der innerhalb Bedingung einfach mit seinem Namen angesprochen wird. Also `(ANY (AND HIT (matches "Chattanooga Choo Choo" rxheader))` <*source*>).

Falls andere Quantoren (EVERY, SOME, NOTANY, NOTEVERY) benötigt werden oder mehrere Mehrfachtags gleichzeitig in eine Bedingung eingehen sollen, kann mit *Lambdas* gearbeitet werden:

```
(where (notevery (lambda (a b) (matches a b)) rxheader vcl_call) (live))
```

Das wäre aber ein weitergehendes Thema, für das hier kein Platz ist.

5.1.4 Streams

```
<stream> ::= (LIVE)
            |(FILE <string>)
            |(VLOG <sttring>)
            |(FILE-starting-at <string> <integer>
            |(WHERE <condition> <stream>)
            |(SELECT-TAG 'tag <stream>)
            |(SELECT-TAGS <list> <stream>)
            |(SELECT <tcd-condition> <stream>)
```

Eine <tcd-condition> darf die Datenelemente `TAG`, `CODE` und `DATA` verwenden und sich damit auf Datensätze allgemein beziehen

5.1.5 Queries

```
<query> ::=  (APP <expr> <stream>)
            |(DMP <stream>)
            |(DUMP <stream>)
            |(TAKE <integer> <stream>)
            |(CHART <integer> <integer> <expr> <stream>)
```

5.2 Variablen, Funktionen, Makros (noch unvollständig)

5.2.1 Mehrfach-Tags

`BACKEND VCL_ACL GZIP HASH OBJHEADER TTL OBJPROTOCOL OBJRESPONSE RXHEADER TXHEADER VCL_CALL VCL_RETURN`

5.2.2 Einfach-Tags

`BACKENDCLOSE BACKENDOPEN BACKENDREUSE BACKEND_HEALT CLI DEBUG EXPBAN EXPKILL FETCHERROR FETCH_BODY HIT INTERRUPTED REQEND REQSTART RXPROTOCOL RXREQUEST RXRESPONSE RXSTATUS RXURL SESSIONCLOSE SESSIONOPEN STATSESS TXPROTOCOL TXREQUEST TXRESPONSE TXSTATUS TXURL WORKTHREAD`

5.2.3 Makros

1. (any expr expr) Wird innerhalb von `WHERE` verwendet, um auszudrücken, dass bei mehrfach-tags irgeneine Zeile eine bestimmte Eigenschaft haben muss

2. (app expr source) Erstellt einen Request–Stream, die einen Bestimmten Ausdruck auf jeden Request eines anderen Request–Streams anwendet

3. (cnt n expr source) Zählt die n nächsten Elemente eines Request–Streams anhand des Merkmals *expr*
4. (objheader name) Liefert des Headerwert eines bestimmten Namens aus dem ObjHeader
5. (rxheader name) Liefert des Headerwert eines bestimmten Namens aus dem RxHeader
6. (select expr source) Liefert einen Request-Stream, die Zeilen nach einem Bestimmten Kriterium auswählt
7. (txheader name) Liefert des Headerwert eines bestimmten Namens aus dem TxHeader
8. (type-is <symbol>) Stellt fest, ob der aktuelle Request vom typ *a* ist. a kann 'c oder 'b sein.
9. (where expr source) Liefert einen Request–Stream , die die aus dem Request–Stream Source stammenden Objekte nach einem bestimmten Kriterium abfiltert.

5.2.4 Funktionen

1. (ban-list) Liefert die aktuelle Ban-List des Caches
2. (bulk source) Generiert Bulk-Ausgabe eines Request–Streams
3. (contains string pattern) Stellt fest, ob pattern in string enthalten ist
4. (dmp source) Liefert tabellarische Ausgabe aus eines Request–Streams
5. (dump source) Liefert Dump eines Request–Streams
6. (file name) Erzeugt Request–Stream aus Logdatei Liefert einn Request–Streams, die eine Datei einliest
7. (head n l) Liefert die ersten n Elemente einer liste
8. (is-prefix prefix string) Stellt fest, ob prefix ein Präfix zu string ist
9. (live) Erzeugt Request–Stream aus *varnishlog*.
10. (matches pattern string) Stellt fest, ob regex-pattern in einem string enthalten ist
11. (max-age r &optional (default nil)) Liefert das Max-Age eines Requests
12. (nth-num n string &optional (default 0)) Liefert die n-te Zahl innerhalb eines Strings

13. (numb pattern string) Liefert die Zahl innerhalb eines Strings
14. (select-tag tag source) Liefert einen Request–Stream, die ein bestimmtes Tag eines Requests passieren lässt
15. (select-tags tags source) Liefert einen Request–Stream, die nur bestimmte Tags eines Requests passieren lässt
16. (substr s from &optional to) Sichere Version von subseq
17. (tab table) Gibt eine Liste tabellarisch aus
18. (tabcar l) Gibt eine Liste tabellarisch aus
19. (take n source) Entnimmt einem Request–Stream eine bestimmte Anzahl Requests

5.2.5 interne Makros

1. (aif test then &optional else) „anaphorisches if”

```
(aif (rxheader "Cache-Control") it "Kein CC-Header")
```

2. (collect symbols body)

5.2.6 interne Funktionen

1. (code x)
2. (col-widths-of-table table)
3. (data x) "datenfeld eines Satzes"(nth 3 x))
4. (dissect pattern string)
5. (find-tag-data tag req)
6. (flatten-opnds l)
7. (inc-hash hk htab)
8. (iterator s); Zeilen-Iterator eines Streams erstellen.
9. (live-stream)
10. (logfile-stream name)
11. (parse line)
12. (print-log-line l)

13. (tag x) Tag eines Satzes"(nth 1 x))
14. (thread x) thread-# eines Satzes"(nth 0 x))
15. (time-string time)
16. (varnish-request-aggregator source)

5.2.7 Wichtige Common Lisp-Funktionen und Makros

1. `and`, `or` und `not` Boolsche Ausdrücke verknüpfen
2. `defparameter` und `setq` Bindungen erzeugen
3. `member` : Mengenzugehörigkeit feststellen

6 Glossar