HCP request analytics

Release 1.5.5

Thorsten Simons

Sep 23, 2020

Contents

Tip: Make sure to use **version 1.4.0** or better for HCP 8.x logs!

hcprequestanalytics reads HTTP access logs from log packages created by *Hitachi Content Platform* (HCP), loads the content into a SQL database and runs SQL-queries against it to provide information about topics like:

- types of requests
- types of requests to specific HCP nodes
- types of reuests from specific clients
- HTTP return codes
- size distribution of requested objects
- HCP internal latency distribution
- clients

It can be easily extended with individual queries (see *Queries*).

Results are generated as a multi-sheet XLSX workbook per default; optionally, CSV files can be requested.

CHAPTER 1

Installation

1.1 Pre-built Executable

For most modern *Linux* derivates you should be able to simply run the executable provided here[1].

There is also a binary for macOS, built and tested with macOS Sierra (10.12.6) here[2].

Grab it there, move it to a folder in your $PATH (`/usr/local/bin`, for example) and follow the instructions in the *Usage* chapter.

1.2 Self-built Executable

In case the provided executable fails to run on your system, you can easily build it on your own. Here's how to do that:

- Clone the repository from GitLab[3]:

```
$ git clone https://gitlab.com/simont3/hcprequestanalytics.git
```

- Change into the project folder and create a Python 3 virtual environment and activate it:

```
$ cd hcprequestanalytics/src
$ python3 -m venv .venv
$ source .venv/bin/activate
```

- Update pip and setuptools, then load all required dev-packages:

```
(.venv) $ pip install -U setuptools pip
(.venv) $ pip install -r pip-requirements-dev.txt
```

- Run the build tool:

```
(.venv) $ pyinstaller hcprequestanalytics.spec
```

[1] https://gitlab.com/simont3/hcprequestanalytics/blob/master/src/dist/hcprequestanalytics.linux
[2] https://gitlab.com/simont3/hcprequestanalytics/blob/master/src/dist/hcprequestanalytics.macos
[3] https://gitlab.com/simont3/hcprequestanalytics.git

You should find the executable in the `dist` subfolder.

Tip: Most likely, in `hcprequestanalytics.spec`, you will have to adopt the path set in the `_pathext` variable to your setup!

- Now move it to a folder in your $PATH (`/usr/local/bin`, for example) and follow the instructions in the *Usage* chapter.

1.3 Python virtual environment

- Create a Python 3 virtual environment and activate it:

```
$ python3 -m venv .venv
$ source .venv/bin/activate
```

- Install the **hcprequestanalytics** package:

```
(.venv) $ pip install -U setuptools pip
(.venv) $ pip install hcprequestanalytics
```

- Now you can run **hcprequestanalytics** this way:

```
(.venv) $ hcprequestanalytics
```

Note: Remember, every time you want to run **hcprequestanalytics**, you need to activate the virtual environment before!

CHAPTER 2

Command Syntax

hcprequestanalytics consists of several subcommands, each used for a specific piece of work. Use `--help` (or `-h`) for details:

```
$ hcprequestanalytics -h
usage: hcprequestanalytics [-h] [--version]
                           {load,analyze,showqueries,dumpqueries} ...

positional arguments:
  {load,analyze,showqueries,dumpqueries}
    load                load the database
    analyze             analyze the database
    showqueries         show the available queries
    dumpqueries         dump the built-in queries to stdout

optional arguments:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
```

2.1 load

The `load` subcommand loads the http gateway logs into a *sqlite3* database file for later analytics:

```
$ hcprequestanalytics load -h
usage: hcprequestanalytics load [-h] -d DB logpkg

positional arguments:
  logpkg      the HCP log package to process

optional arguments:
  -h, --help  show this help message and exit
  -d DB       the database file
```

2.2 showqueries

The `showqueries` subcommand shows the avaible queries - the ones built-in as well as the ones added through the `-a` parameter:

```
$ hcprequestanalytics showqueries -h
usage: hcprequestanalytics showqueries [-h] [-a ADDITIONALQUERIES] [-1]

optional arguments:
  -h, --help            show this help message and exit
  -a ADDITIONALQUERIES  a file containg addition queries (see documentation)
  -1                    print a concatenated list of queries, for easy cut and
                        paste
```

2.3 analyze

The `analyze` subcommand runs queries against the database created with the `load` subcommand to create an xlsx file as result. Alternatively, a set of csv files can be requested as well.

```
$ hcprequestanalytics analyze -h
usage: hcprequestanalytics analyze [-h] [-a ADDITIONALQUERIES] -d DB
                                   [-p PREFIX] [-c] [--procs PROCESSES]
                                   [queries [queries ...]]

positional arguments:
  queries               a list of query names, or nothing for "all"; you can
                        select a group of queries by using the first few
                        characters followed by an asteriks ('req*' for
                        example)

optional arguments:
  -h, --help            show this help message and exit
  -a ADDITIONALQUERIES  a file containg addition queries (see documentation)
  -d DB                 the database file
  -p PREFIX             prefix for the output files
  -s                    analyze requests recorded by snodes
  -c                    create CSV files instead of a XLSX file
  --procs PROCESSES     no. of subprocesses to run, defaults to no. of CPUs
```

2.4 dumpqueries

The `dumpqueries` subcommand dumps the built-in queries to stdout. They can be used as templates to build own queries for use with the `-a` parameter:

```
$ hcprequestanalytics dumpqueries -h
usage: hcprequestanalytics dumpqueries [-h]

optional arguments:
  -h, --help  show this help message and exit
```

CHAPTER 3

Usage

3.1 Pre-requisites

- **hcprequestanalytics** has been installed as described in chapter *Installation*
- Either the binary has placed in the $PATH or the Python virtual environment has been activated and **hcprequestanalytics** can be started successfully:

```
$ hcprequestanalytics --version
  hcprequestanalytics: v.1.3.2 (2017-10-10/Sm)
```

- HCP internal logs have been downloaded into an empty folder:

```
$ ls -lh HCPLogs-*
-rw-r--r--@  1 tsimons  273924073   4.3M Sep 13 07:44 HCPLogs-hcp72.archivas.com-acc-
↪20170913-0742.zip
```

- Enough free space is available to uncompress the log package and all *http_gateway_request.log* files contained in it

3.2 Running hcprequestanalytics

Running **hcprequestanalytics** is a two step process:

1. Create and load the database from an HCP log package:

```
$ hcprequestanalytics load -d hcp72.db HCPLogs-hcp72.archivas.com-acc-20170913-0742.zip
un-packing HCPLogs-hcp72.archivas.com-acc-20170913-0742.zip
        un-packing access logs for node 192.168.0.176
        un-packing access logs for node 192.168.0.177
        un-packing access logs for node 192.168.0.178
        un-packing access logs for node 192.168.0.179
unpacking HCPLogs-hcp72.archivas.com-acc-20170913-0742.zip took 5.762 seconds
        reading node 176 - ./tmpdkllzu3y/.../20170812-0316/http_gateway_request.log.0 - 5,
↪295 records
        reading node 176 - ./tmpdkllzu3y/.../20170813-0341/http_gateway_request.log.0 -␣
↪944 records
```

(continues on next page)

(continued from previous page)

```
        [..]
            lots of more entries listed here
        [..]
        reading node 179 - ./tmpdkllzu3y/.../20170913-0328/http_gateway_request.log.0 - 1␣
↪records
        reading node 179 - ./tmpdkllzu3y/.../20170913-0743/http_gateway_request.log.0 - 0␣
↪records
loading database with 590,734 records took 30.288 seconds

$ ls -lh hcp72.db
-rw-r--r--  1 tsimons  273924073   109M Oct 10 16:58 hcp72.db
```

You can repeat the loading for more Log packages, in which case the existing database will be used.

Of course, you'll want to load Logs from a single HCP into the database, as results would be falsified, otherwise!

> **Warning: hcprequestanalytics** doesn't check for duplicate records. That means, if you load the database with the same log package twice, the query results will be falsified, as well.

2. Run queries against the database

> **Tip: hcprequestanalytics analyze** starts as much subprocesses as CPUs are available. Using that pool of subprocesses, it runs queries in parallel. On a 4-CPU system, the overal runtime should go down to roughtly a quarter; the limiting factors are descibed in the *Good to know* chapter.

```
$ hcprequestanalytics analyze -d hcp72.db -p hcp72
scheduling these queries for analytics using 8 parallel process(es):
        500_highest_throughput        : The 500 records with the highest throughput␣
↪(Bytes/sec)
        500_largest                   : The records with the 500 largest requests
        500_worst_latency             : The records with the 500 worst latencies
        clientip                      : No. of records per client IP address
        clientip_httpcode             : No. of records per http code per client IP address
        clientip_request_httpcode     : No. of records per http code per request per␣
↪client IP address
        count                         : No. of records, overall
        day                           : No. of records per day
        day_hour                      : No. of records per hour per day
        day_hour_req                  : No. of records per request per hour per day
        day_req                       : No. of records per request per day
        day_req_httpcode              : No. of records per http code per request per day
        node                          : No. of records per node
        node_req                      : No. of records per request per node
        node_req_httpcode             : No. of records per http code per request per node
        percentile_req                : No. of records per request analysis, including␣
↪percentiles for size and latency
        percentile_throughput_128kb   : No. of records per request, with percentiles on␣
↪throughput (Bytes/sec) for objects >= 128KB
        req                           : No. of records per request
        req_httpcode                  : No. of records per http code per request
        req_httpcode_node             : No. of records per node per http code per request
wait for queries finishing:
        count                         : 0.290 seconds
        500_worst_latency             : 0.761 seconds
        500_highest_throughput        : 1.298 seconds
        clientip                      : 1.436 seconds
```

(continues on next page)

(continued from previous page)

```
        500_largest                  : 1.951 seconds
        clientip_httpcode            : 2.017 seconds
        day                          : 2.244 seconds
        clientip_request_httpcode    : 2.553 seconds
        day_hour                     : 3.269 seconds
        node                         : 1.522 seconds
        percentile_throughput_128kb  : 0.665 seconds
        node_req                     : 2.444 seconds
        day_req                      : 3.385 seconds
        day_hour_req                 : 3.972 seconds
        day_req_httpcode             : 3.439 seconds
        node_req_httpcode            : 2.643 seconds
        req                          : 1.400 seconds
        req_httpcode                 : 1.483 seconds
        req_httpcode_node            : 1.284 seconds
        percentile_req               : 17.030 seconds
analytics finished after 20.094 seconds
```

Tip: You can run selected queries by adding them to the end of the command:

```
$ hcprequestanalytics -d hcp72.db analyze -p hcp72 req count
```

This will run just the *req* and the *count* query.

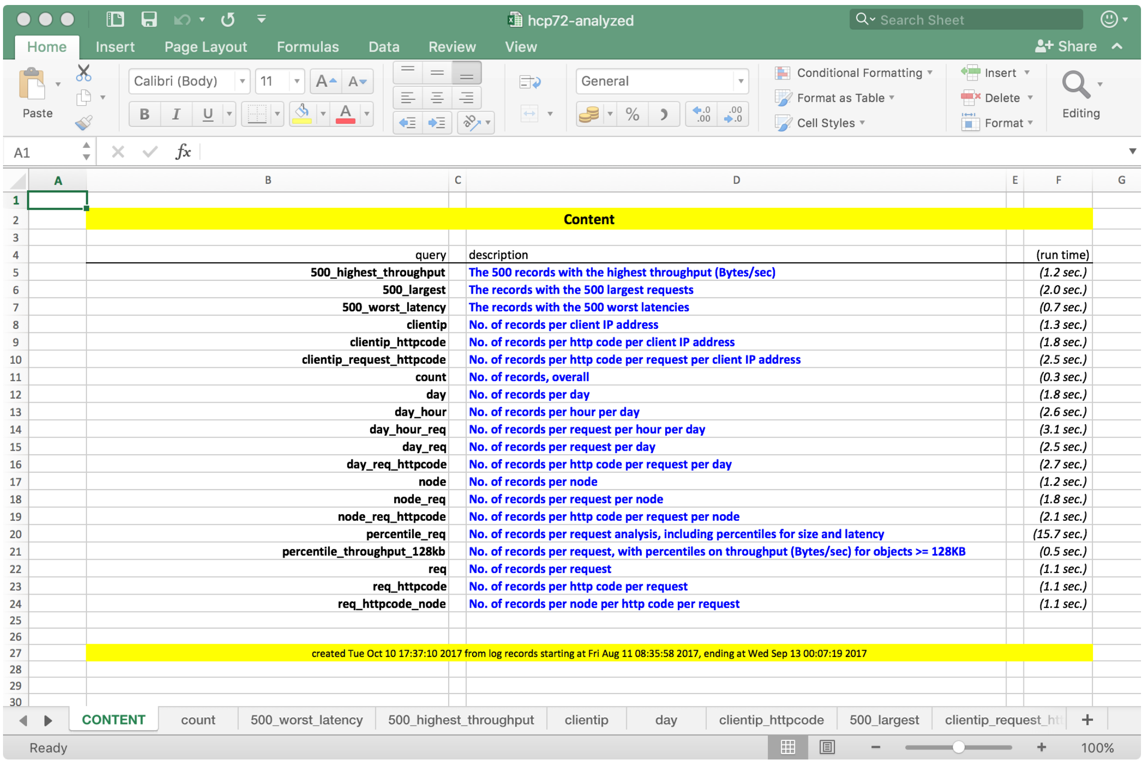
It's also possible to select a group of queries by adding an asteriks:

```
$ hcprequestanalytics -d hcp72.db analyze -p hcp72 'req*'
```

This will run all queries beginning with *req*.

Anyhow, you now have an **xlsx** (Excel) file with the results per query:

```
$ ls -lh *.xlsx
-rw-r--r--  1 tsimons  273924073   178K Oct 10 17:02 hcp72-analyzed.xlsx
```



query	description	(run time)
	Content	
500_highest_throughput	The 500 records with the highest throughput (Bytes/sec)	(1.2 sec.)
500_largest	The records with the 500 largest requests	(2.0 sec.)
500_worst_latency	The records with the 500 worst latencies	(0.7 sec.)
clientip	No. of records per client IP address	(1.3 sec.)
clientip_httpcode	No. of records per http code per client IP address	(1.8 sec.)
clientip_request_httpcode	No. of records per http code per request per client IP address	(2.5 sec.)
count	No. of records, overall	(0.3 sec.)
day	No. of records per day	(1.8 sec.)
day_hour	No. of records per hour per day	(2.6 sec.)
day_hour_req	No. of records per request per hour per day	(3.1 sec.)
day_req	No. of records per request per day	(2.5 sec.)
day_req_httpcode	No. of records per http code per request per day	(2.7 sec.)
node	No. of records per node	(1.2 sec.)
node_req	No. of records per request per node	(1.8 sec.)
node_req_httpcode	No. of records per http code per request per node	(2.1 sec.)
percentile_req	No. of records per request analysis, including percentiles for size and latency	(15.7 sec.)
percentile_throughput_128kb	No. of records per request, with percentiles on throughput (Bytes/sec) for objects >= 128KB	(0.5 sec.)
req	No. of records per request	(1.1 sec.)
req_httpcode	No. of records per http code per request	(1.1 sec.)
req_httpcode_node	No. of records per node per http code per request	(1.1 sec.)

created Tue Oct 10 17:37:10 2017 from log records starting at Fri Aug 11 08:35:58 2017, ending at Wed Sep 13 00:07:19 2017

If you prefer comma-separated-value (CSV) files, just add -c to the analyze command:

```
$ hcprequestanalytics analyze -d hcp72.db -p hcp72 -c
scheduling these queries for analytics using 8 parallel process(es):
        500_highest_throughput        : The 500 records with the highest throughput␣
↪(Bytes/sec)
        500_largest                   : The records with the 500 largest requests
        500_worst_latency             : The records with the 500 worst latencies
        clientip                      : No. of records per client IP address
        clientip_httpcode             : No. of records per http code per client IP address
        clientip_request_httpcode     : No. of records per http code per request per␣
↪client IP address
        count                         : No. of records, overall
        day                           : No. of records per day
        day_hour                      : No. of records per hour per day
        day_hour_req                  : No. of records per request per hour per day
        day_req                       : No. of records per request per day
        day_req_httpcode              : No. of records per http code per request per day
        node                          : No. of records per node
        node_req                      : No. of records per request per node
        node_req_httpcode             : No. of records per http code per request per node
        percentile_req                : No. of records per request analysis, including␣
↪percentiles for size and latency
        percentile_throughput_128kb   : No. of records per request, with percentiles on␣
↪throughput (Bytes/sec) for objects >= 128KB
        req                           : No. of records per request
        req_httpcode                  : No. of records per http code per request
        req_httpcode_node             : No. of records per node per http code per request
wait for queries finishing:
        count                         : 0.323 seconds
        500_worst_latency             : 0.805 seconds
        clientip                      : 1.309 seconds
        500_highest_throughput        : 1.315 seconds
        day                           : 1.797 seconds
        clientip_httpcode             : 1.807 seconds
        500_largest                   : 2.188 seconds
        clientip_request_httpcode     : 2.616 seconds
        node                          : 1.440 seconds
        day_hour                      : 2.970 seconds
        percentile_throughput_128kb   : 0.627 seconds
        node_req                      : 2.144 seconds
        day_req                       : 2.890 seconds
        day_hour_req                  : 3.454 seconds
        day_req_httpcode              : 3.087 seconds
        req                           : 1.222 seconds
        node_req_httpcode             : 2.385 seconds
        req_httpcode                  : 1.237 seconds
        req_httpcode_node             : 1.410 seconds
        percentile_req                : 17.067 seconds
analytics finished after 19.720 seconds
```

You now have one **csv** file per query:

```
$ ls -lh *.csv
-rw-r--r--  1 tsimons  273924073   87K Oct 10 17:05 hcp72-500_highest_throughput.csv
-rw-r--r--  1 tsimons  273924073   86K Oct 10 17:05 hcp72-500_largest.csv
-rw-r--r--  1 tsimons  273924073   77K Oct 10 17:05 hcp72-500_worst_latency.csv
-rw-r--r--  1 tsimons  273924073  462B Oct 10 17:05 hcp72-clientip.csv
-rw-r--r--  1 tsimons  273924073  1.9K Oct 10 17:05 hcp72-clientip_httpcode.csv
-rw-r--r--  1 tsimons  273924073  3.0K Oct 10 17:05 hcp72-clientip_request_httpcode.csv
-rw-r--r--  1 tsimons  273924073   18B Oct 10 17:05 hcp72-count.csv
-rw-r--r--  1 tsimons  273924073  2.0K Oct 10 17:05 hcp72-day.csv
```

(continues on next page)

(continued from previous page)

```
-rw-r--r--  1 tsimons  273924073   7.8K Oct 10 17:05 hcp72-day_hour.csv
-rw-r--r--  1 tsimons  273924073    18K Oct 10 17:05 hcp72-day_hour_req.csv
-rw-r--r--  1 tsimons  273924073   6.1K Oct 10 17:05 hcp72-day_req.csv
-rw-r--r--  1 tsimons  273924073   8.7K Oct 10 17:05 hcp72-day_req_httpcode.csv
-rw-r--r--  1 tsimons  273924073   359B Oct 10 17:05 hcp72-node.csv
-rw-r--r--  1 tsimons  273924073   1.2K Oct 10 17:05 hcp72-node_req.csv
-rw-r--r--  1 tsimons  273924073   3.5K Oct 10 17:05 hcp72-node_req_httpcode.csv
-rw-r--r--  1 tsimons  273924073   1.1K Oct 10 17:05 hcp72-percentile_req.csv
-rw-r--r--  1 tsimons  273924073   506B Oct 10 17:05 hcp72-percentile_throughput_128kb.csv
-rw-r--r--  1 tsimons  273924073   371B Oct 10 17:05 hcp72-req.csv
-rw-r--r--  1 tsimons  273924073   1.0K Oct 10 17:05 hcp72-req_httpcode.csv
-rw-r--r--  1 tsimons  273924073   3.5K Oct 10 17:05 hcp72-req_httpcode_node.csv
```

CHAPTER 4

Queries

4.1 Built-in queries

hcprequestanalytics comes with a set of pre-defined queries:

```
$  hcprequestanalytics showqueries
available queries:
        500_highest_throughput          The 500 records with the highest throughput (Bytes/sec)
        500_httpcode_409                The 500 newest records with http code 409
        500_httpcode_413                The 500 newest records with http code 413
        500_httpcode_503                The 500 newest records with http code 503
        500_largest_req_httpcode_node   The records with the 500 largest requests by req,␣
↪httpcode, node
        500_largest_size                The records with the 500 largest requests sorted by␣
↪size
        500_worst_latency               The records with the 500 worst latencies
        clientip                        No. of records per client IP address
        clientip_httpcode               No. of records per http code per client IP address
        clientip_node                   No. of records per clientip per node
        clientip_request_httpcode       No. of records per http code per request per client IP␣
↪address
        count                           No. of records, overall
        day                             No. of records per day
        day_hour                        No. of records per hour per day
        day_hour_req                    No. of records per request per hour per day
        day_req                         No. of records per request per day
        day_req_httpcode                No. of records per http code per request per day
        mapi_endp_req_http              MAPI request: endpoints, request, http code
        mapi_user_req_http              MAPI requests by user
        node                            No. of records per node
        node_req                        No. of records per request per node
        node_req_httpcode               No. of records per http code per request per node
        percentile_req                  No. of records per request analysis, including␣
↪percentiles for size and latency
        percentile_throughput_128kb     No. of records per request, with percentiles on␣
↪throughput (Bytes/sec) for objects >= 128KB
        req                             No. of records per request
        req_httpcode                    No. of records per http code per request
        req_httpcode_node               No. of records per node per http code per request
```

(continues on next page)

(continued from previous page)

```
        ten_ns_proto_clientip_httpcode  No. of records per Tenant / Namespace / protocol /␣
↪client IP address / http code
        ten_ns_proto_httpcode           No. of records per Tenant / Namespace / protocol /␣
↪http code
        ten_ns_proto_percentile_req     No. of records per Tenant / Namespace / protocol,␣
↪including percentiles for size and latency
        ten_ns_proto_user_httpcode      No. of records per Tenant / Namespace / protocol /␣
↪user / http code
        ten_proto_httpcode              No. of records per Tenant / protocol / http code
        ten_user_ns_req_http            Tenants with all users accessing Namespaces, incl.␣
↪request and httpcode
```

Tip: More queries might have been added with newer versions - always check with the command above!

4.2 Adding individual queries

If additional queries are wanted, **hcprequestanalytics** can be easily extended by creating a query file and adding it to the call:

```
$ cat addqueries
[add_count]
comment = count all records
query = SELECT count(*) FROM logrecs

[add_req_count]
comment = count records per request
query = SELECT request, count(*) FROM logrecs GROUP BY request
freeze pane : C3

[add_node_req_http]
comment = node-per-request-per-httpcode analysis
query = SELECT node, request, httpcode, count(*),
        min(size), avg(size), max(size),
        percentile(size, 10), percentile(size, 20),
        percentile(size, 30), percentile(size, 40),
        percentile(size, 50), percentile(size, 60),
        percentile(size, 70), percentile(size, 80),
        percentile(size, 90), percentile(size, 95),
        percentile(size, 99), percentile(size, 99.9),
        min(latency), avg(latency),
        max(latency),
        percentile(latency, 10), percentile(latency, 20),
        percentile(latency, 30), percentile(latency, 40),
        percentile(latency, 50), percentile(latency, 60),
        percentile(latency, 70), percentile(latency, 80),
        percentile(latency, 90), percentile(latency, 95),
        percentile(latency, 99), percentile(latency, 99.9)
        FROM logrecs GROUP BY node, request, httpcode
freeze pane : E3
```

You can check the available queries, including the additional ones:

```
$ hcprequestanalytics -d dbfile.db -a addqueries showqueries
available queries:
    500_highest_throughput          The 500 records with the highest throughput (Bytes/sec)
    500_largest_req_httpcode_node   The records with the 500 largest requests by req, httpcode,
↪ node
```

(continues on next page)

(continued from previous page)

```
    500_largest_size                The records with the 500 largest requests sorted by size
    500_worst_latency               The records with the 500 worst latencies
    add_count                       count all records
    add_node_req_http               node-per-request-per-httpcode analysis
    add_req_count                   count records per request
    clientip                        No. of records per client IP address
    clientip_httpcode               No. of records per http code per client IP address
    clientip_request_httpcode       No. of records per http code per request per client IP␣
↪address
    count                           No. of records, overall
    day                             No. of records per day
    day_hour                        No. of records per hour per day
    day_hour_req                    No. of records per request per hour per day
    day_req                         No. of records per request per day
    day_req_httpcode                No. of records per http code per request per day
    node                            No. of records per node
    node_req                        No. of records per request per node
    node_req_httpcode               No. of records per http code per request per node
    percentile_req                  No. of records per request analysis, including percentiles␣
↪for size and latency
    percentile_throughput_128kb     No. of records per request, with percentiles on throughput␣
↪(Bytes/sec) for objects >= 128KB
    req                             No. of records per request
    req_httpcode                    No. of records per http code per request
    req_httpcode_node               No. of records per node per http code per request
    ten_ns_proto_httpcode           No. of records per Tenant / Namespace / protocol / http␣
↪code
    ten_ns_proto_percentile_req     No. of records per Tenant / Namespace / protocol,␣
↪including percentiles for size and latency
    ten_ns_proto_user_httpcode      No. of records per Tenant / Namespace / protocol / user /␣
↪http code
    ten_proto_httpcode              No. of records per Tenant / protocol / http code
```

Rules:

- You need to stick to the format as shown above - not doing so will most likely result in a crash
- the [**term**] is the name of the query, which you can use in the `analyze` call
- the **comment** entry is what is shown in when calling `showqueries`
- the **query** entry is where to put the query in
- The QUERY has to follow the SQLite3 SELECT rules[4]
- You can use all the column names listed below, the aggregate functions offered by SQLite[5] as well as the private functions listed below

[4] https://www.sqlite.org/lang_select.html
[5] https://www.sqlite.org/lang_aggfunc.html

4.3 Columns in the `logrecs` table

column	type	description
`node`	TEXT	the HCP nodes backend IP address
`clientip`	TEXT	the requesting clients IP address
`user`	TEXT	the user who did the request
`timestamp`	FLOAT	the point in time of the request (seconds since Epoch)
`timestampstr`	TEXT	the point in time of the request (string)
`request`	TEXT	the HTTP request
`path`	TEXT	the requested object
`httpcode`	INT	the HTTP return code
`size`	INT	the size of the transfers body
`namespace`	TEXT	the HCP Namespace accessed (usually, in the form of `namespace.tenant[@protocol]`)
`latency`	INT	the internal latency needed to fullfil the request

4.4 Private SQL functions that can be used in queries

- `getNamespace(path, namespace)`

  Extract the name of the Namespace (bucket, container) from the `path` and `namespace` database columns.

- `getTenant(namespace)`

  Extract the name of the Tenant from the `namespace` database column.

- `getProtocol(namespace)`

  Extract the access protocol used from the `namespace` database column. Returns either `S3`, `Swift` or `native REST`.

- `percentile(column, float)`

  Aggregate function that calculates the percentage (given by *float*) of *column* from all selected records.

  Warning: Due to it's nature, *percentile()* collects a list of the columns' value from each selected row. As this list is held in memory, it can consume a lot of it. A rough calculation would be:

  ```
    no. of percentile() occurrences in the query
  * no of rows selected
  * 24 bytes
  ```

- `tp(size, latency)`

  Calculates the throughput (in bytes/second) from an objects size and the internal latency.

CHAPTER 5

Result Interpretation

Proper interpretation of **hcprequestanalytics** results requires some good knowledge about how HCP works, as well as about http, networking and client behaviour. The information in this chapter hopefully helps understanding the results a bit.[6]

5.1 Load distribution

You can use the **node_*** queries to find out how load is distributed across the nodes.

No. of records per node

node	count(*)	min(size)	avg(size)	max(size)
176	147,471	0	14,015	581,632,000
177	134,957	0	12,359	66,216,807
178	151,553	0	11,410	94,083,599
179	156,753	0	11,245	94,083,599

As the example shows, the load distribution is OK so far. A slight deviation is normal due to DNS (and/or loadbalancer) behaviour.

Due to the nature of HCP, you'll want all load to be distributed evently across all available HCP Nodes.

5.2 Who's generating load

Often, it is of interest to find out who exactly is generating load towards HCP. The **clientip_*** queries are your friend in this case:

No. of records per client IP address

clientip	count(*)	min(size)	avg(size)	max(size)
192.168.0.100	1,137	0	42	599
192.168.0.219	195,215	0	6,119	94,083,599
192.168.0.220	168,077	0	16,463	581,632,000
192.168.0.228	224,839	0	14,493	94,083,599
192.168.0.31	1,466	0	4,416	573,270

[6] All queries referenced in this chapter are based on the built-in queries.

You will still need to map the IP-addresses to your clients, as usual.

5.3 Request size

All versions of HCP prior to version 8 are logging request sizes for GET requests (and *some* POST requests), only. That's why often enough a request size of *zero* is reported for everything else.

That of course has its implications regarding throughput (*Bytes/sec*), which can only be calculated for requests with sizes > *zero*.

5.4 Latency

The latency column, seen in the result of many queries, state what is called the *HCP internal latency*. That means, it talks about the time passed between the clients' request being received by HCP *until* the last byte of HCPs answer was sent back to the client. During this time, things like fetching the object from the backend storage, de-compression and/or de-cryption will take place, adding to the overall time needed for sending or receiving the objects data itself.

The records with the 500 worst latencies

	request	httpcode	latency	size	Bytes_sec	clientip
189	GET	200	308	14,863	48,256	192.168.0.228
190	GET	200	308	2,674,539	8,683,568	192.168.0.220
191	GET	200	307	3,860,502	12,574,925	192.168.0.220
192	GET	200	307	3,907,264	12,727,244	192.168.0.220
193	GET	200	306	2,550,516	8,335,020	192.168.0.220
194	GET	200	305	2,548,966	8,357,266	192.168.0.220
195	GET	200	302	519	1,719	192.168.0.228
196	GET	200	301	519	1,724	192.168.0.228
197	GET	200	301	519	1,724	192.168.0.228

The latency value itself doesn't tell too much, as long it's not put into relation with the size of the request. In addition, latency created by the network and even the client will go into this value, as long as these latencies take place while the request is between the two states mentioned in the beginning.

That means that a huge latency most likely isn't an issue with huge objects, but *might* be with small ones.

5.5 Throughput

Throughput, mentioned as *Bytes/sec* in some of the queries' results, is a simple calculation of *size* devided by *latency*. It does not necessarily tell you the network throughput for a single object, as the latency also takes in account the time needed to de-crypt or un-compress the object before delivery to the client, for example.

5.6 Interpretation of percentiles

A percentile (or a centile) is a measure used in statistics indicating the value below which a given percentage of observations in a group of observations fall. For example, the 20th percentile is the value (or score) below which 20% of the observations may be found.[7]

The **percentile_*** queries try to make use of this by presenting a wide range of percentiles for *size*, *latency* and *Bytes/sec* (see the Throughput section!). Basically, it will tell you how your values are distributed within the entire range of 100% of the data.

[7] Taken from the Percentile article at Wikipedia[8]
[8] https://en.wikipedia.org/wiki/Percentile

No. of records per request analysis, including percentiles for size and latency

request	count(*)	min(size)	avg(size)	max(size)	pctl-10 (size)	pctl-20 (size)	pctl-30 (size)	pctl-90 (size)	pctl-95 (size)	pctl-99 (size)	pctl-99.9 (size)
DELETE	107,166	0	0	0	0	0	0	0	0	0	0
GET	128,794	0	56,109	581,632,000	520	1,254	1,334	53,201	87,709	385,414	2,550,516
HEAD	242,790	0	0	0	0	0	0	0	0	0	0
POST	15	0	71	265	0	0	0	265	265	265	265
PUT	111,969	0	0	290	0	0	0	0	0	0	0

Let's take `row 6` as an example - it tells that the GET request with the hugest size was 581,632,000 bytes. But it also tells that 99.9% of the GET requests are 2,550,516 Bytes or smaller (`cell Q6`). This lets us know that the `max(size)` value is just a peak, appearing in the highest 0.1% of the requests. Looking at the *500_largest_size* query result will proof that:

The records with the 500 largest requests sorted by size

request	httpcode	node	latency	size	Bytes_sec	clientip	use
GET	503	176	33,158	581,632,000	17,541,227	192.168.0.220	n
GET	200	176	3,318	94,083,599	28,355,515	192.168.0.219	n
GET	200	176	15,858	94,083,599	5,932,879	192.168.0.228	n
GET	200	178	3,810	94,083,599	24,693,858	192.168.0.219	n
GET	200	179	3,860	94,083,599	24,373,989	192.168.0.228	n

This gives a good overview, but still needs to be taken in relation with other parameters - for example, if you have overall high latency, you might also have overall huge request sizes…

CHAPTER 6

Good to know

6.1 Database size

A single database record will use 200+ bytes if the paths in the requests are short in average (~25 characters), and will grow on longer paths.

6.2 Compute

As of today, *loading* the database is single-threaded. Depending on the disk throughput, it will use a single CPU at 100%.

Running *queries*, on the other hand, is done in parallel using subprocesses. Each of them will load a single CPU to up to 100%, again depending on disk throughput.

In the default setting (i.e. w/o specifying `--procs`). it will spawn as much subprocesses as there are CPUs in the system. This can easily load your system to its limits.

6.3 Disk

Depending on the size, the database itself can get quite big. A busy 12-node HCP generated a 7.3GB log package (compressed) for a single week. That translated into a 74GB database, holding 384.1 million log records.

Due to the fact that there are no indexes configured for the database (many different ones would be needed to facilitate all queries), these indexes are created (and loaded) on the fly when running queries. They will end up in your systems usual tmp folder - if that one doesn't have enough free capacity, the queries will fail. Some of the more complex queries will require as much disk space as the database itself.

Now think of running some of these queries in parallel, each creating its own temp indexes. While analyzing huge databases, this will likely overload your system, unless you have a lot of disk space.

If **hcprequestanalytics** prints error messages about *filesystem or database full*, you can make sure that an appropriately sized folder is used for the temporary database indexes by setting this environment variable before running **hcprequestanalytics**:

```
$ export SQLITE_TMPDIR=/wherever/you/have/enough/space
```

Make sure to replace `/wherever/you/have/enough/space` with a path that matches your systems reality, of course!

6.4 Memory

Especially the *percentile()* aggregate function needs a lot of memory when used in queries against huge databases, because it has to hold a list of all values to be able to calculate the percentile, at the end.

The mentioned *req_httpcode* query has been observed to use more than 35GB of real memory on the database mentioned above.

Trying to use more memory than available will usually kill a query. Running multiple queries in parallel, each of them allocation a huge amount of memory will quickly bring you to that point, and all queries will fail.

6.5 Conclusion

A simple task -*analyzing http log files*- can be much more challenging than expected.

Compute, Disk, Memory and parallelism are all relevant as soon as the amount of data exceeds a pretty low barrier. Depending on the amount of log data to analyze, these needs have to be balanced.

The only strategies here are:

- use the *percentile()* aggregate function sparingly, to save memory
- run less queries in parallel than the no. of CPUs would allow (`--procs 2`, for example)
- or even run queries one at a time (turn off multi-processing by `--procs 1`)

or:

- **throw in more hardware: CPUs, Memory, Disk capacity**

CHAPTER 7

Get info

You might want to see what's going on, especially if you are running **hcprequestanalytics** against a huge database.

7.1 Queries running

As the queries are running in parallel, you will receive info about its success (or fail) once each query has ended. To find out which queries are running at the moment, you can run this command in a second session:

```
$ lsof 2>/dev/null | grep '__\*'
hcpreques 602  sm  16u  REG  1,5  0  5461618  /private/var/folders/y3/74nllcpj5f511sgw18t55_
↪qh0000gn/T/I_am__*clientip_httpcode*__pbxvbswl
hcpreques 603  sm  17u  REG  1,5  0  5461730  /private/var/folders/y3/74nllcpj5f511sgw18t55_
↪qh0000gn/T/I_am__*clientip_request_httpcode*__198td_f7
```

In this example, the string **I_am__*clientip_httpcode*__pbxvbswl** in the last field of the output indicates that process **602** (the second field) runs the **clientip_httpcode** query.

7.2 Disk space used for tmp indexes

To find out how much disk space is used for temporary database indexes, you can run:

```
$ lsof 2>/dev/null | grep /wherever/you/have/enough/space
hcpreques  602  sm  txt  REG  1,2  5301620171   26454 /wherever/you/have/enough/space/etilqs_
↪rrlN0dgfFfwQg9E
hcpreques  602  sm  18u  REG  1,2  5302781691   26454 /wherever/you/have/enough/space/etilqs_
↪rrlN0dgfFfwQg9E
hcpreques  603  sm  txt  REG  1,2  1256108032   26456 /wherever/you/have/enough/space/etilqs_
↪7QxuTtMv8AtPYnw
hcpreques  603  sm  19u  REG  1,2  1256108032   26456 /wherever/you/have/enough/space/etilqs_
↪7QxuTtMv8AtPYnw
```

You will have to replace `/wherever/you/have/enough/space` by the folder you are using for the temporary database indexes (see *Good to know* for details).

The 7th field will tell you how many bytes are actually used for this single temporary database index. Be aware that each temporary index shows up twice in this output, as it is opend twice by the process. The slight difference in size is caused by the process writing into the index during `lsof` was running. The 2nd field will tell you the pid of the process running the query using this temporary index.

BTW, you will **not** see the files containing the indexes in the filesystem, and they will not be accounted for when using the `df` or `du` commands.

CHAPTER 8

Release History

1.5.5 2020-09-23

- fixed a bug that caused loading a log package right in the beginning (missing 1 required positional argument: 'addqueries')

1.5.4 2019-06-05

- added the option to analyze S-node logs from a **hcphealth** database
- added *MQE* related queries
- matured the database functions to withstand incorrect values in numerical fields

1.5.3 2019-05-21

- added the *clientip_node* query

1.5.2 2019-05-15

- *analyze* now also allows to use a database created by **hcpheath**

1.5.1 2019-03-19

- replaced *shutil.unpack_archive* with *zipfile.Zipfile.extractall*, as *unpack_archive* seems to have issues with zip- file members > 2 GB.

1.5.0 2019-01-24

- added a table for MAPI-related logs to the database, as well as queries specially tailored for MAPI

1.4.5 2019-01-23

- added a query that list users accessing HCP

1.4.4 2019-01-14

- added some more queries

1.4.3 2019-01-13

- removed unnecessary debug output

1.4.2 2019-01-11

- added queries related to Tenant / Namespace / protocol

1.4.1 2019-01-04

- very minor optical changes to the result XLSX file (index sheet)

1.4.0 2018-12-27

- made compatibility changes for log packages created by HCP 8.x

1.3.8 2017-12-07

- fixed a bug that caused log packages to fail if they contained HCP-S logs
- Fixed a bug that caused a crash in analyze when a query didn't return any data
- made using *setproctitle* optional when installing through pip for environments that are not supported (CygWin, for example)

1.3.7 2017-12-07

- fixed setup.py to include pre-requisite *setproctitle* (thanks to Kevin, again)

1.3.6 2017-11-01

- now properly builds with Python 3.6.3 and PyInstaller 3.3; removed the note from docs

1.3.5 2017-10-30

- now using *setproctitle* to set more clear process titles (for ps, htop)

1.3.4 2017-10-13

- fixed a bug invented in 1.3.3 that caused long running queries to break xlsx creation (thanks to Kevin Varley for uncovering this)

1.3.3 2017-10-12

- removed gridlines from the content sheet
- fine-tuned the column width in the query sheets
- made the runtime column a bit more readable
- added *500_largest_size* query
- some documentation additions

1.3.2 2017-10-10

- added query runtime to content sheet in xlsx

1.3.1 2017-10-05

- added timestamp of first and last record to xlsx file
- added SQL function `tp(size, latency)` to calculate the throughput
- adopted queries to use `tp()`

1.3.0 2017-10-03

- some more xlsx luxury
- added more queries
- added the ability to dump the built-in queries to stdout
- re-worked the cmd-line parameters (-d is now where it belongs to…)

1.2.2 2017-09-26

- documentation fixes

1.2.1 2017-09-25

- removed percentile() from the most queries, due to too long runtime on huge datasets
- added the possibility to select a group of queries on *analyze*

1.2.0 2017-09-24

- now analyze runs up to cpu_count subprocesses, which will run the queries in parallel
- added cmdline parameter `--procs` to allow to set the no. of subprocesses to use, bypassing the cpu_count

1.1.1 2017-09-23

- added per-day queries
- all numerical fields in the XLSX file now formated as #.##0

1.1.0 2017-09-23

- re-built the mechanism to add individual queries
- *.spec file prepared to build with pyinstaller w/o change on macOS and Linux

1.0.4 2017-09-22

- a little more featured XLXS files

1.0.3 2017-09-21

- now creating a single XLSX file on *analyze*, added option -c to create CSV files instead

1.0.2 2017-09-16

- fixed the timestamp column (now hold the seconds since Epoch)

1.0.1 2017-09-15

- now we do understand log records of access to the Default Namespace properly
- speed-up of unpacking by just unpacking the required archives

1.0.0 2017-09-10

- initial release

CHAPTER 9

License / Trademarks

9.1 The MIT License (MIT)

9.2 Trademarks and Copyrights of used material

Hitachi is a registered trademark of Hitachi, Ltd., in the United States and other countries. Hitachi Data Systems is a registered trademark and service mark of Hitachi, Ltd., in the United States and other countries.

Archivas, Hitachi Content Platform, Hitachi Content Platform Anywhere and Hitachi Data Ingestor are registered trademarks of Hitachi Data Systems Corporation.

All other trademarks, service marks, and company names in this document or web site are properties of their respective owners.

The used icon was made by Freepik[9] from Flaticon[10] and is licensed by Creative Commons BY 3.0[11].

[9] http://www.freepik.com

[10] https://www.flaticon.com/

[11] http://creativecommons.org/licenses/by/3.0/