Visualizing Mirrors

mirrors.ustc.edu.cn 服务器日志分析

李博杰 bojieli@gmail.com

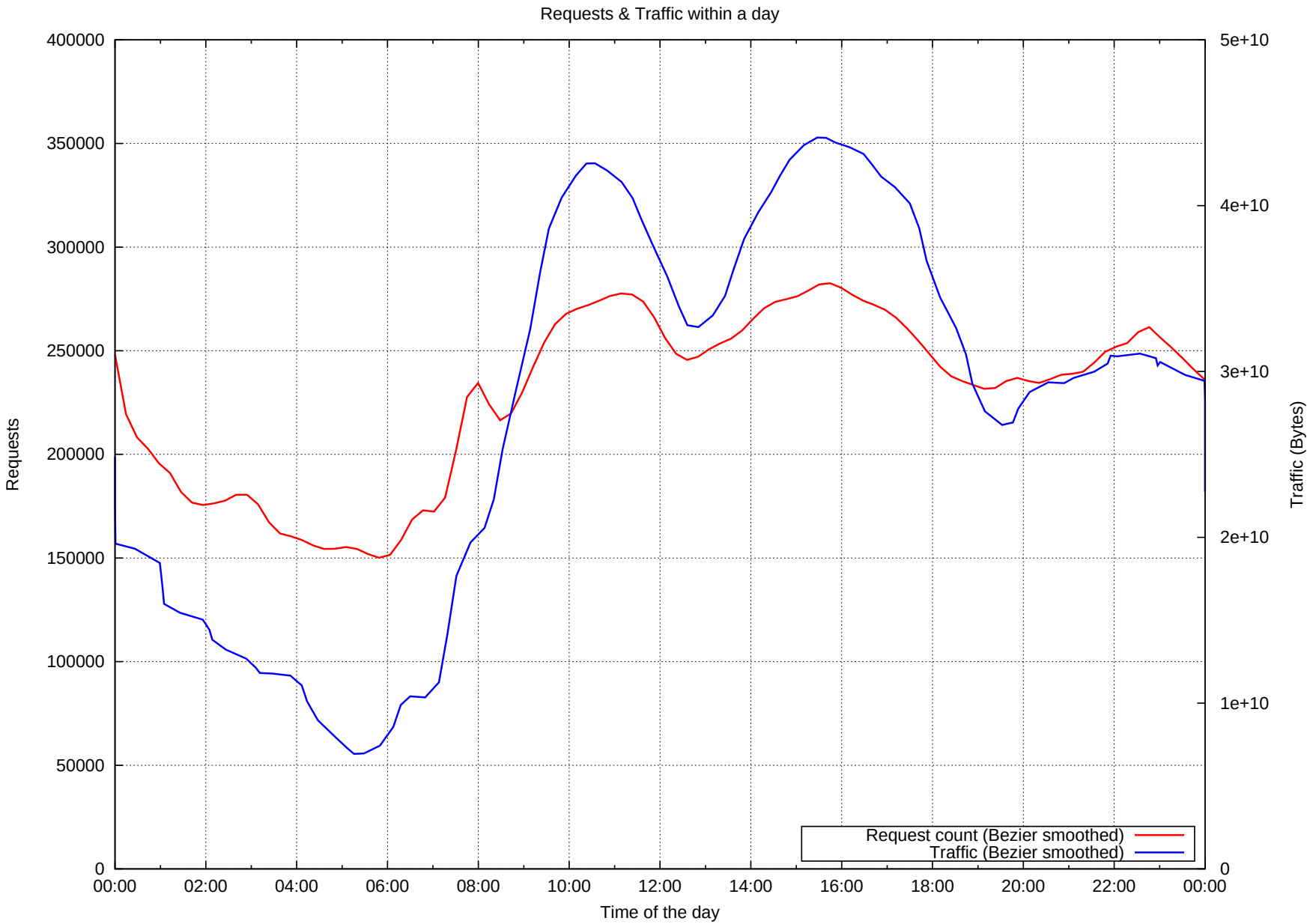
August 14, 2012

Outline

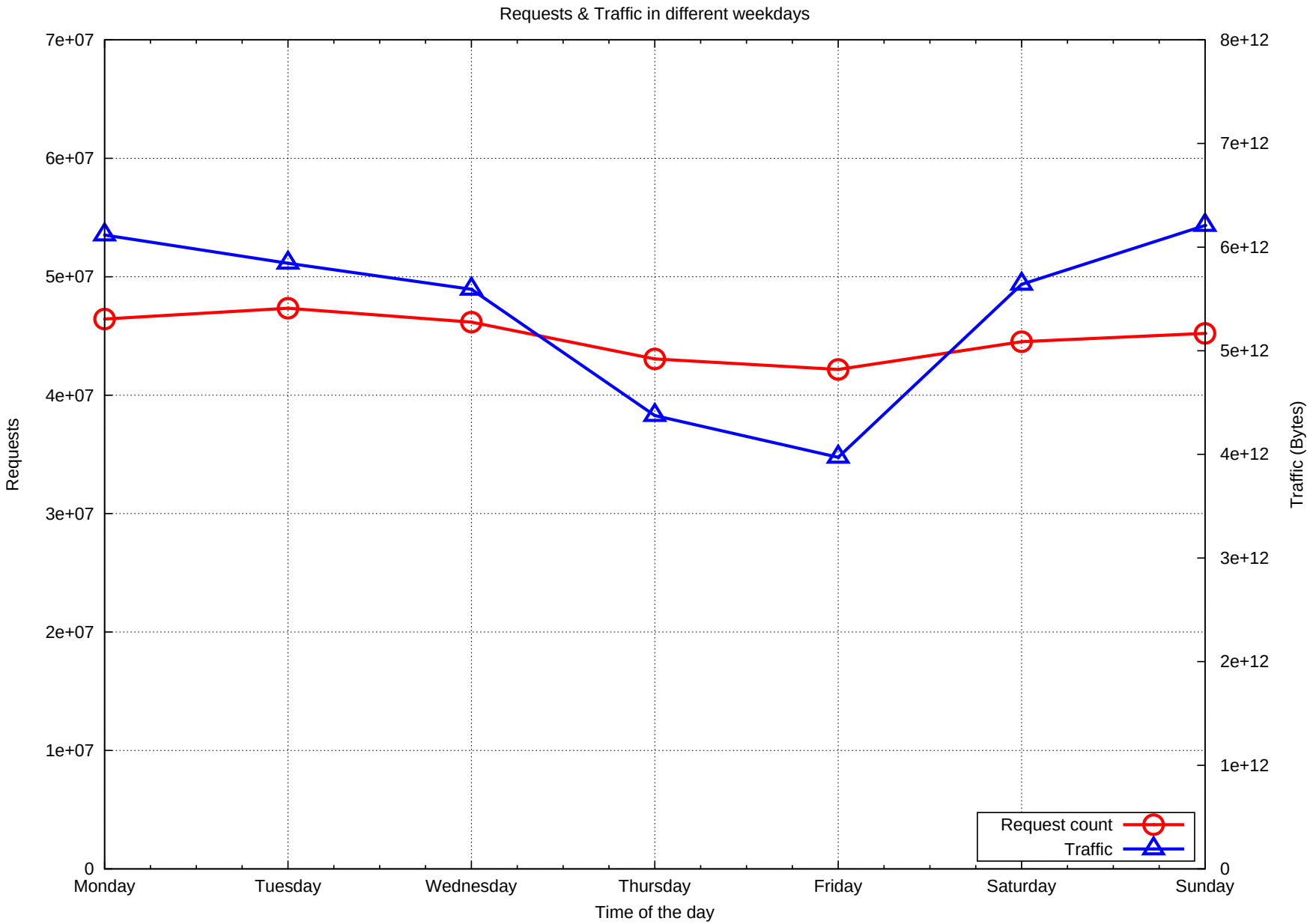
Notes

- The data is access log of mirrors.ustc.edu.cn in 51 days. See 'Technical Details' section for more info about dataset.
- Some graphs are in log-scale for clarity. Please note whether x axis, y axis or both are in log-scale. The graph title sometimes lies.
- Because there may be many points in a graph, sampling is made to reduce file size (they are vector graphics), hence there may be some 'straight lines'. I have checked the data to make sure the graphs illustrate real trends.
- Title length is limited, so the title itself may not explain well, please keep an eye on the axis and keys of the graph.
- Graphs are shown in the hope of conveying information without words. Any questions or suggestions, please email me.

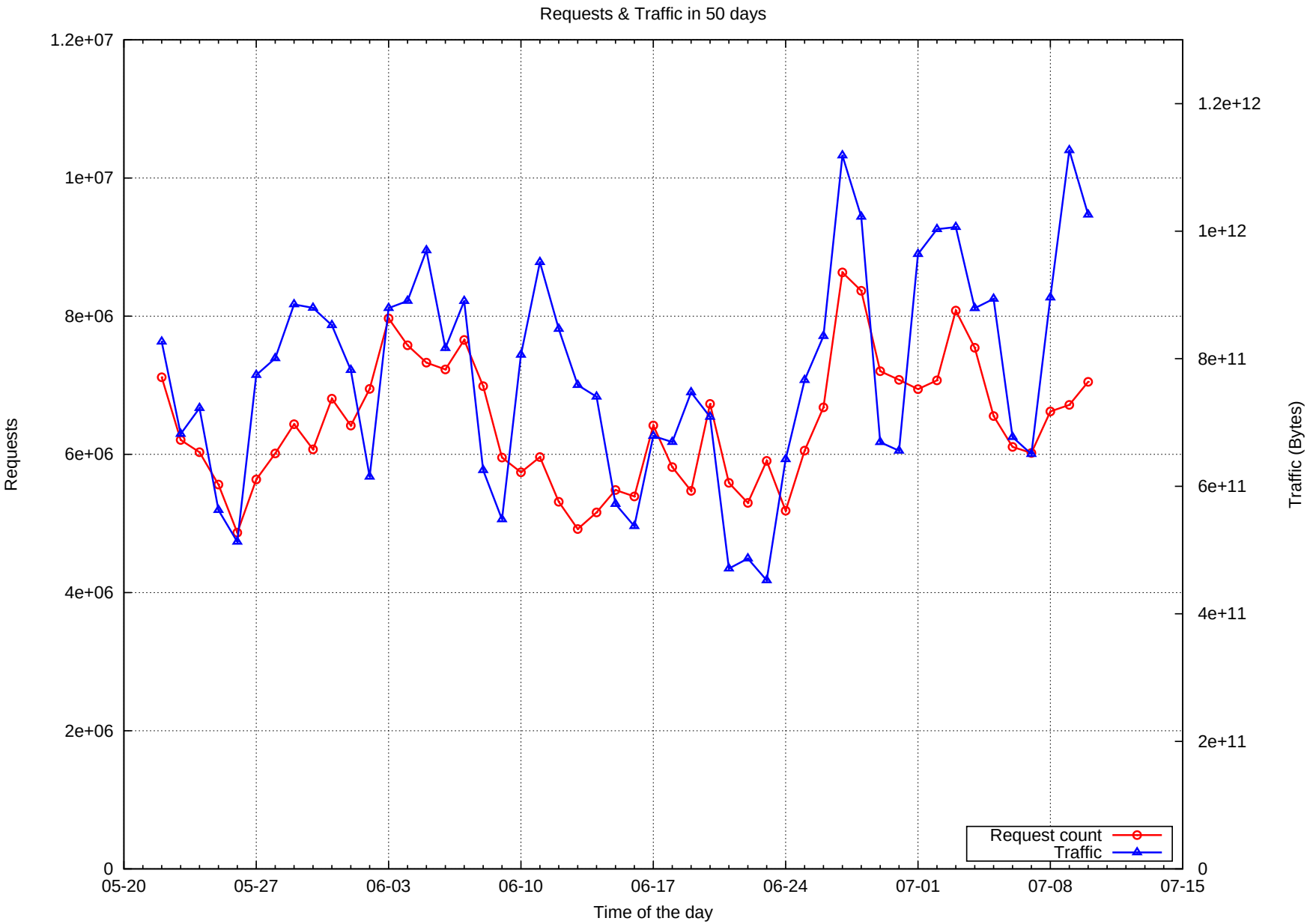
Requests & Traffic in a day



Requests & Traffic in a week



Requests & Traffic across 50 days

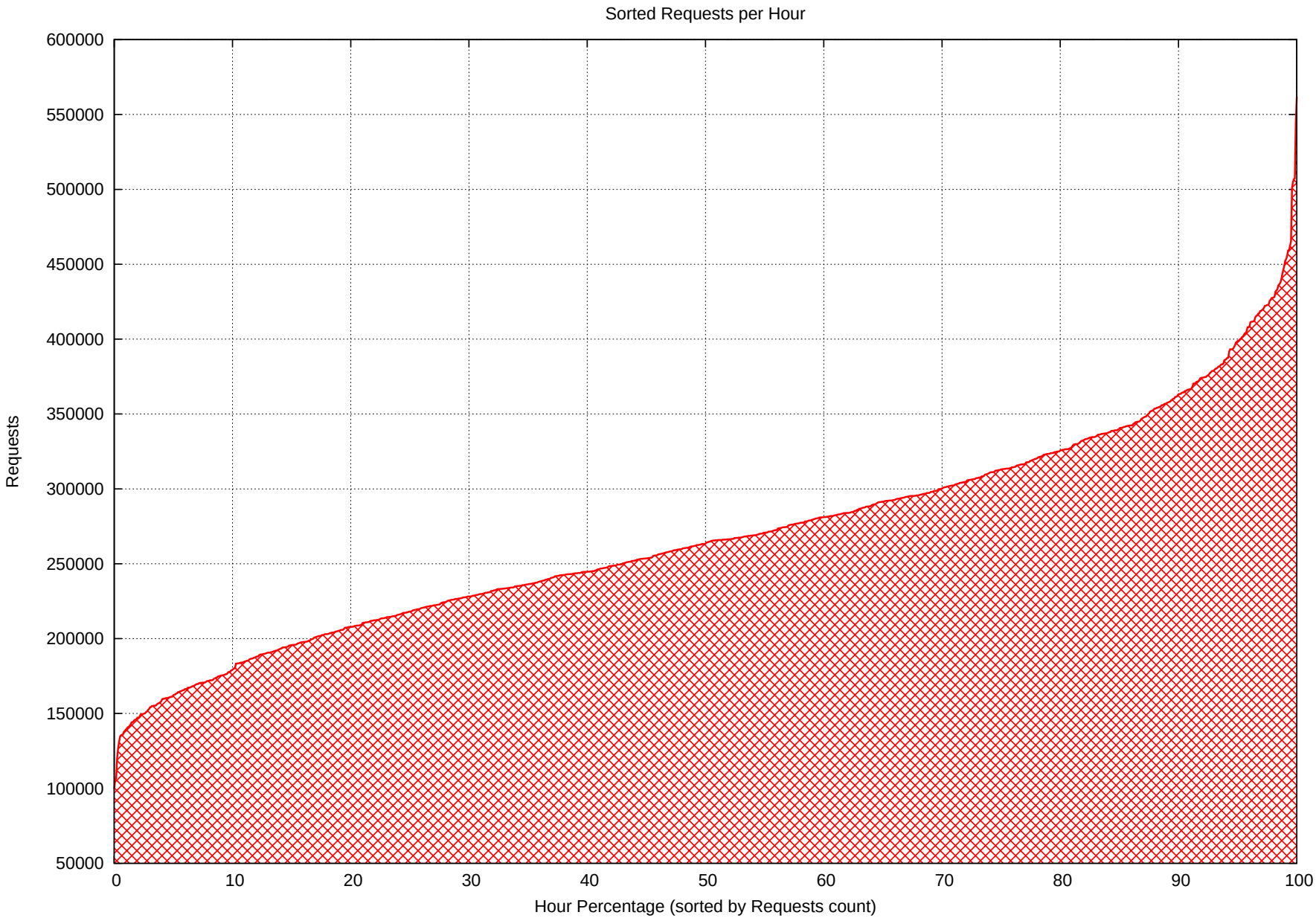


Statistics

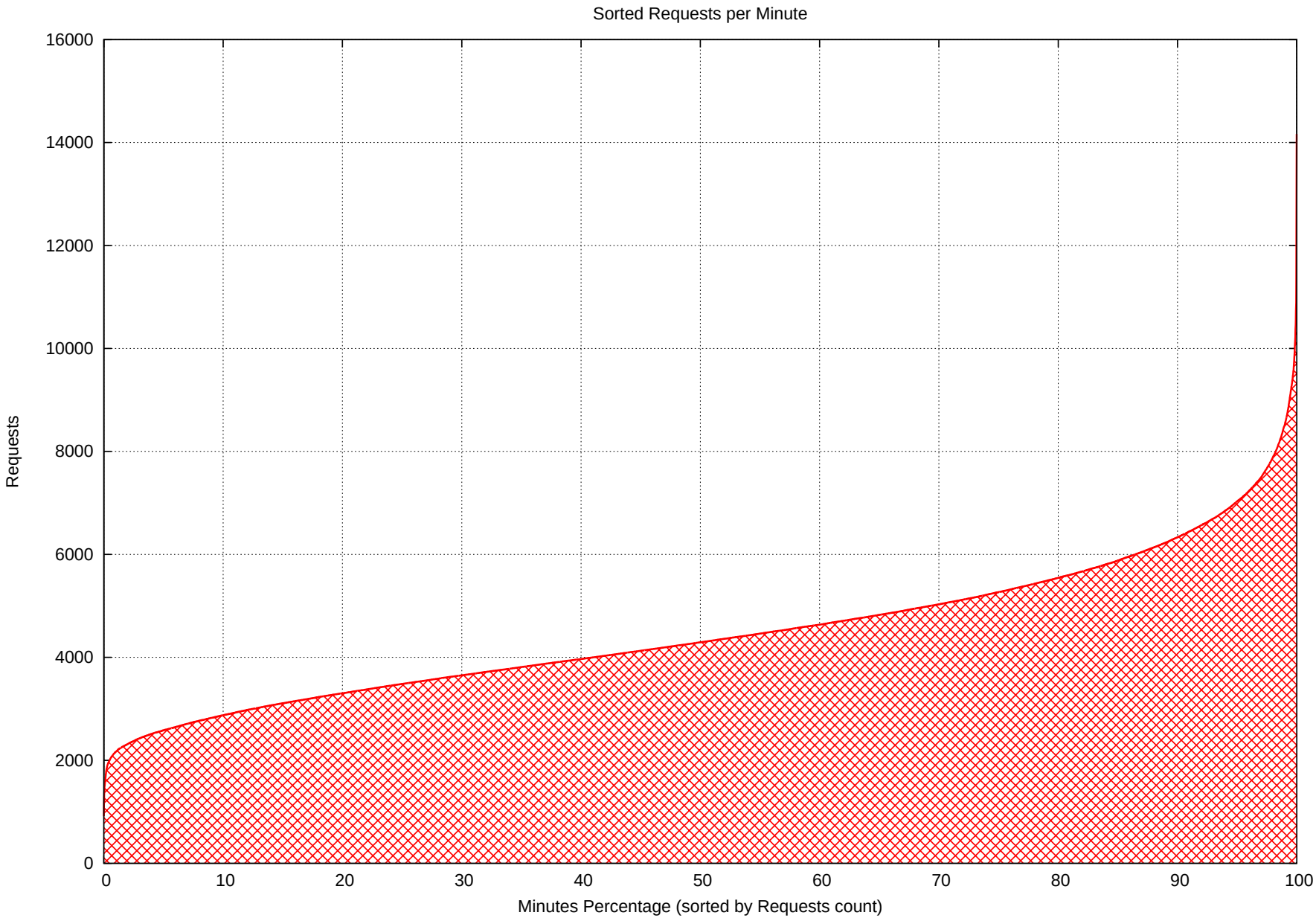
	Requests	Traffic
Total	328976877	36892 GB
Avg. per Day	6450527	723.4 GB
Max. per Day	8632963	1049.5 GB
Min. per Day	4868022	421.5 GB
Avg. per Hour	268771	30.14 GB
Max. per Hour	561925	79.75 GB
Min. per Hour	99506	2.97 GB
Avg. per Minute	4480	514.4 MB
Max. per Minute	14714	N/A
Min. per Minute	441	N/A
Avg. per Second	74.66	8779 KB
Max. per Second	2117	N/A
Min. per Second	1	N/A

- Because the time recorded is only completion time of the request, and large requests can span hours, so Max./Min. per minute/second is not applicable.

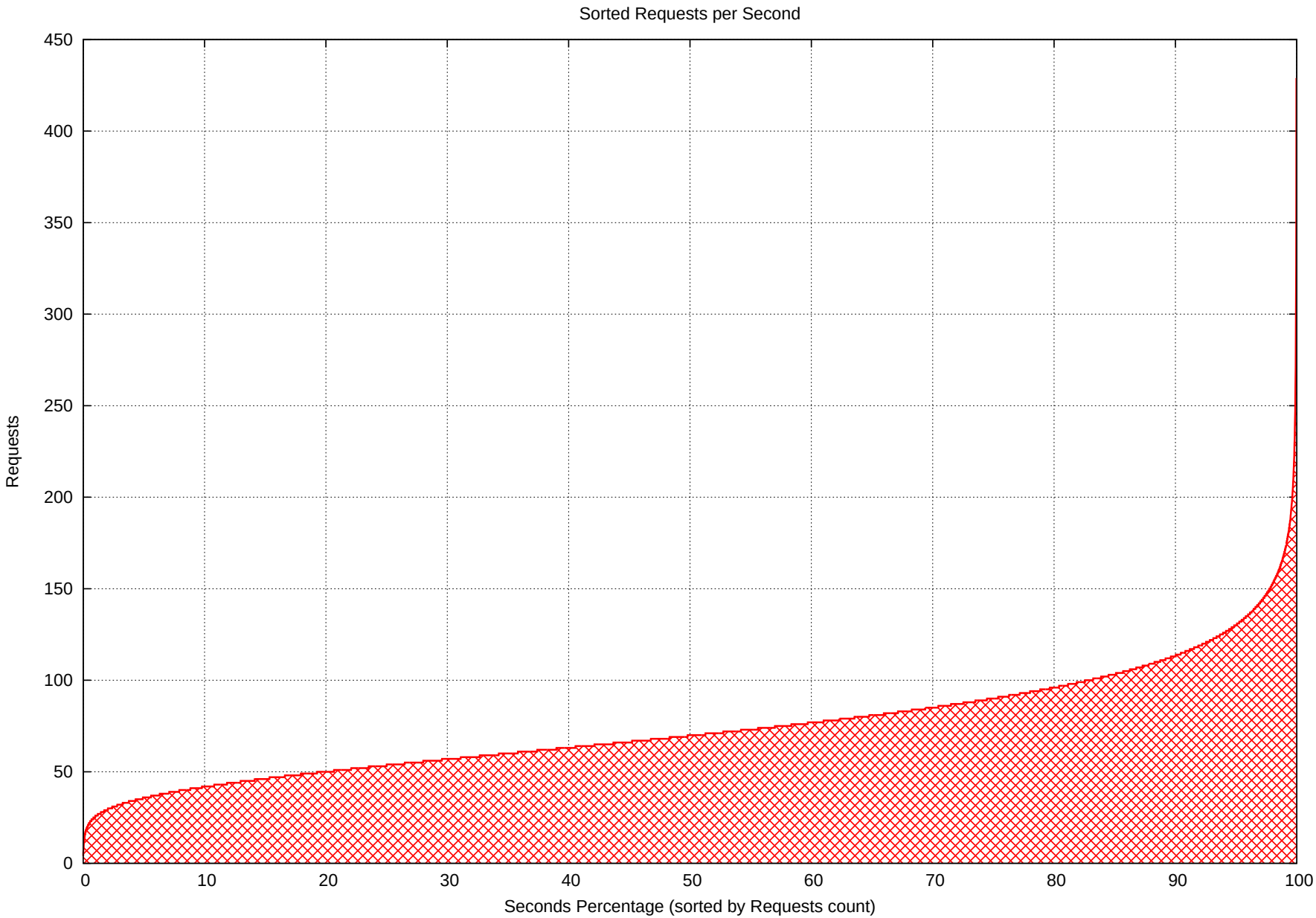
Cumulative Requests per Hour



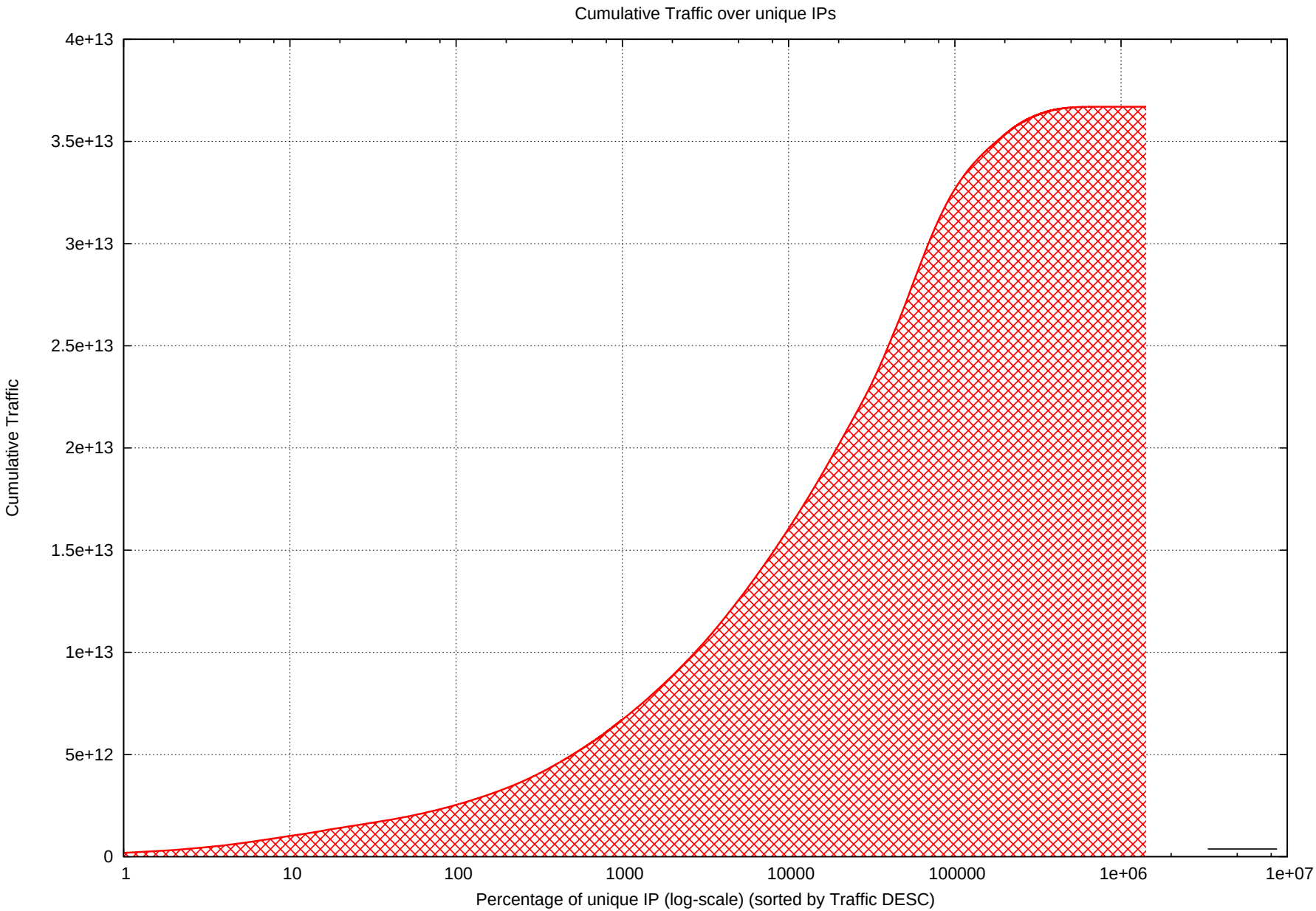
Cumulative Requests per Minute



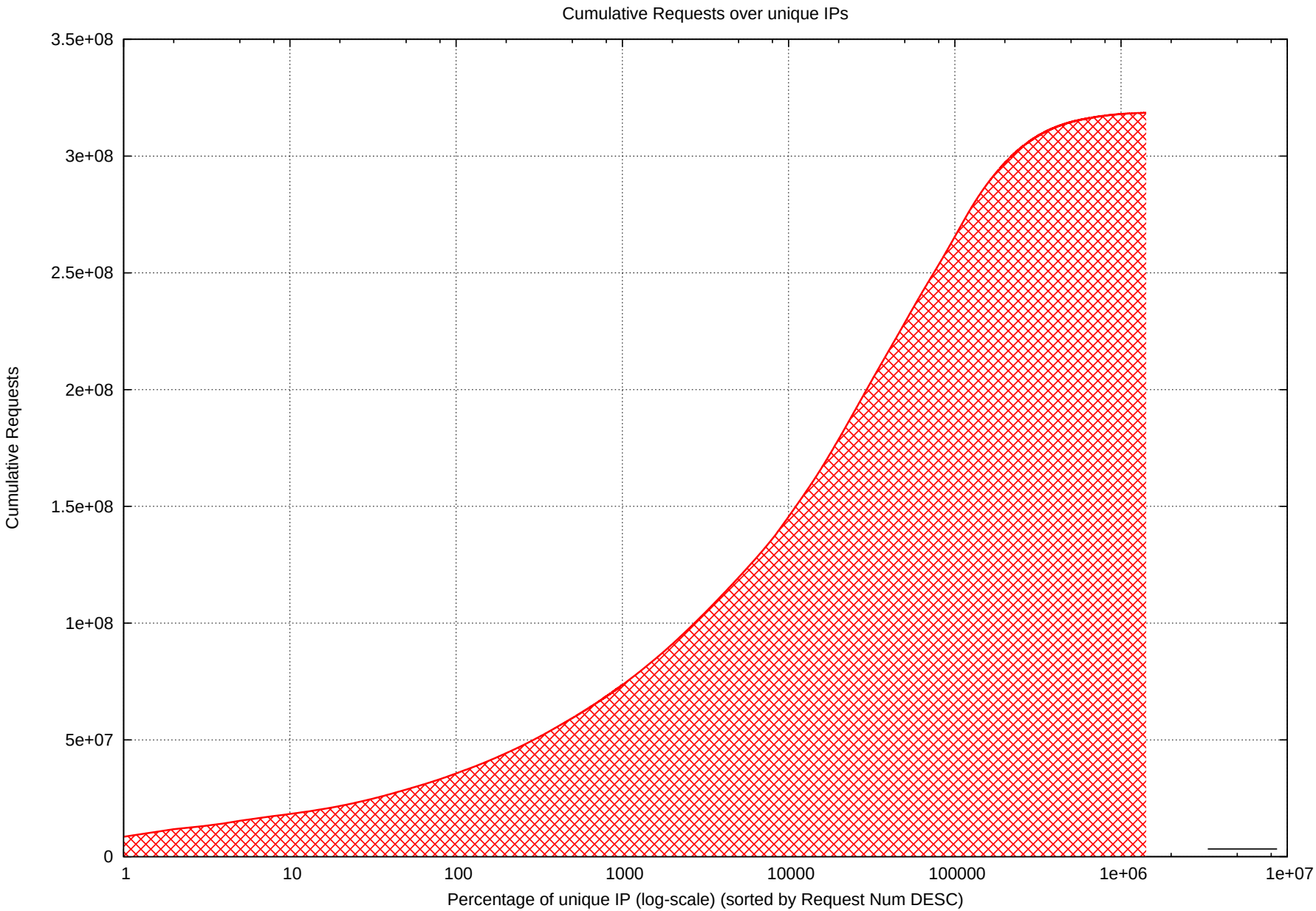
Cumulative Requests per Second



Cumulative Traffic over IPs: 20%-80% law



Cumulative Requests over IPs: 20%-80% law

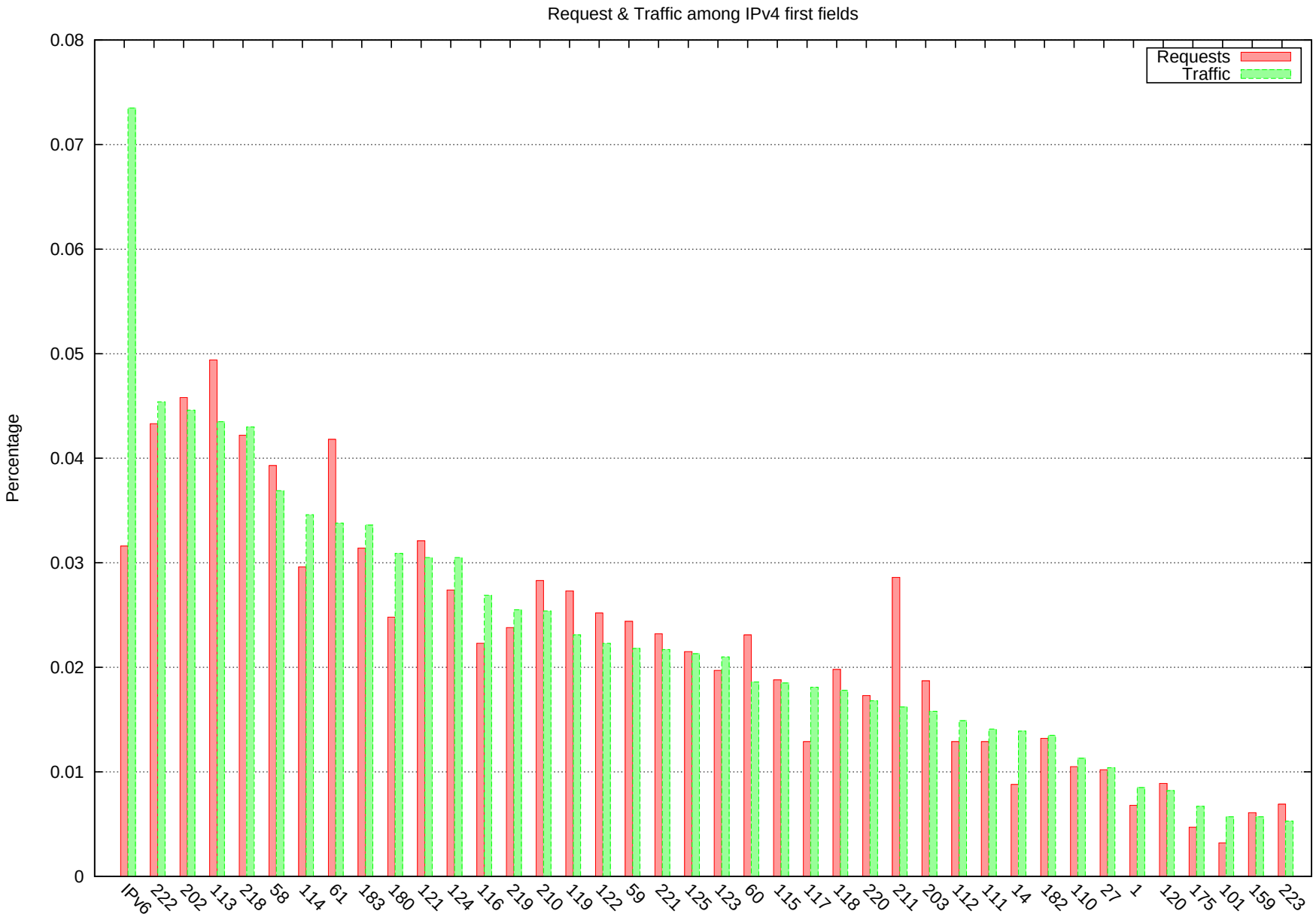


IPv4 vs. IPv6

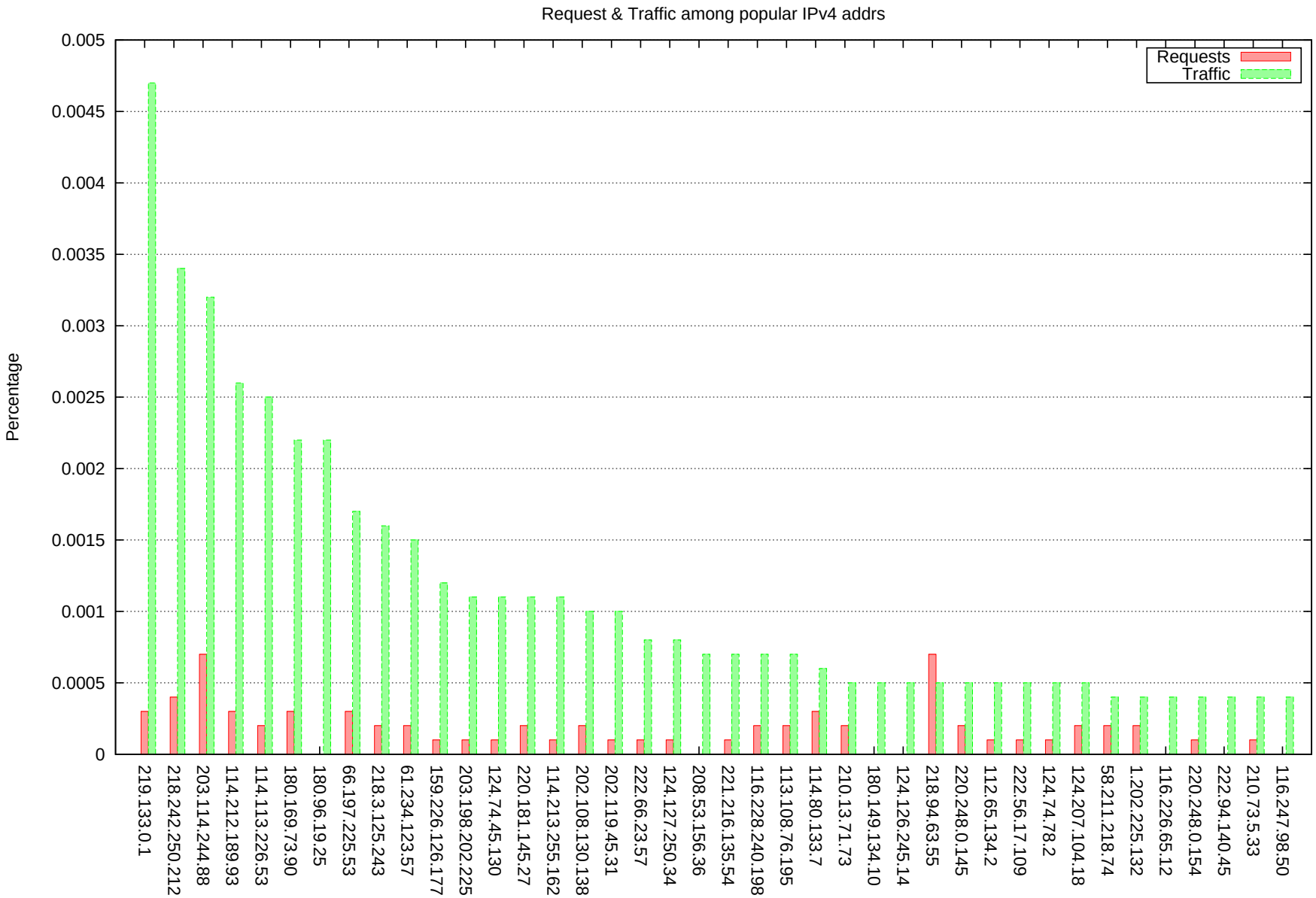
	Requests	Traffic
IPv4	318575688 (96.84%)	34180 GB (92.65%)
IPv6	10401189 (3.15%)	2712 GB (7.35%)

- It can be seen that IPv6 still have a long way to go...

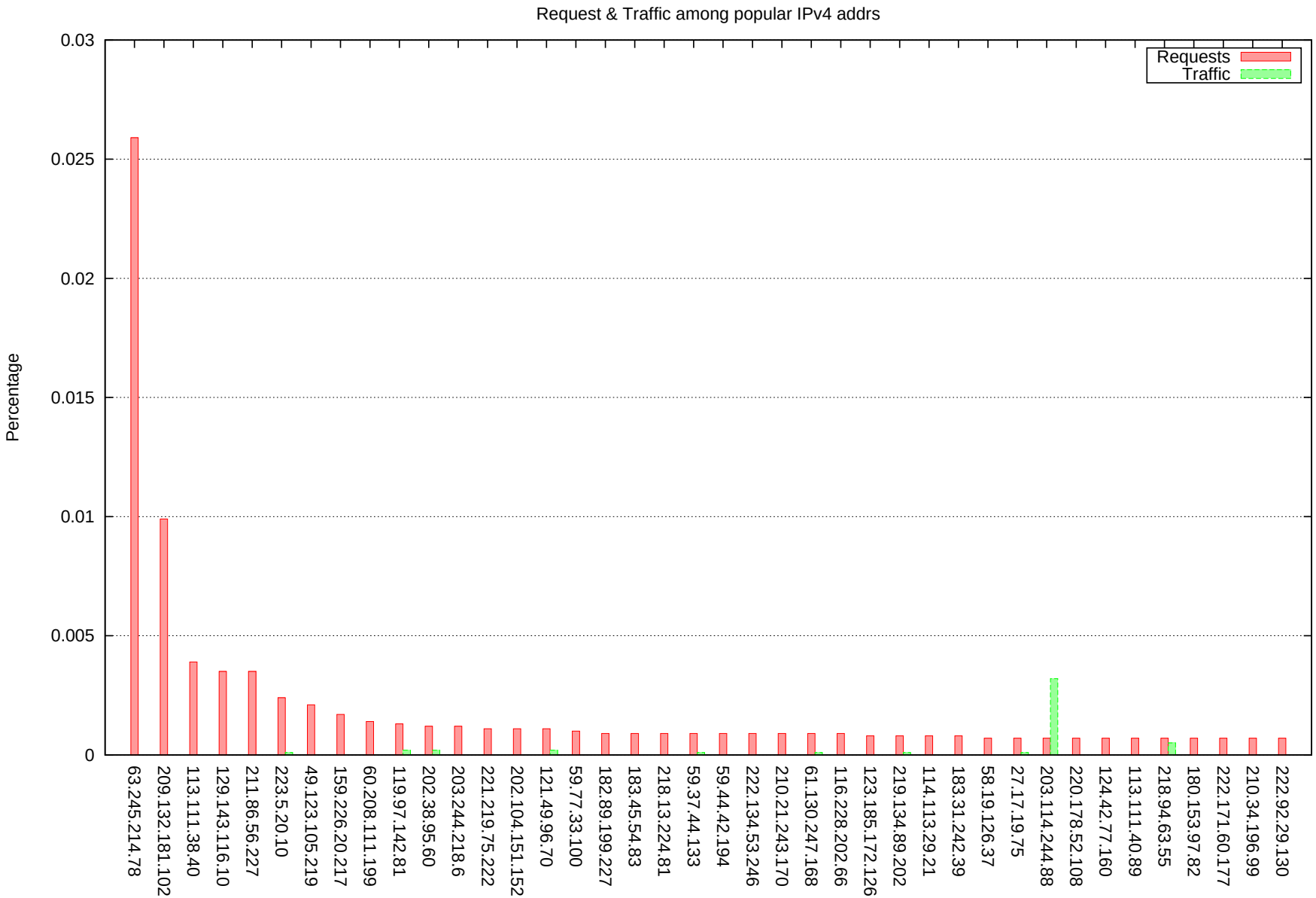
Requests & Traffic TOP 40: xxx.0.0.0/24



Traffic TOP 40: IPv4 addrs



Request count TOP 40: IPv4 addrs

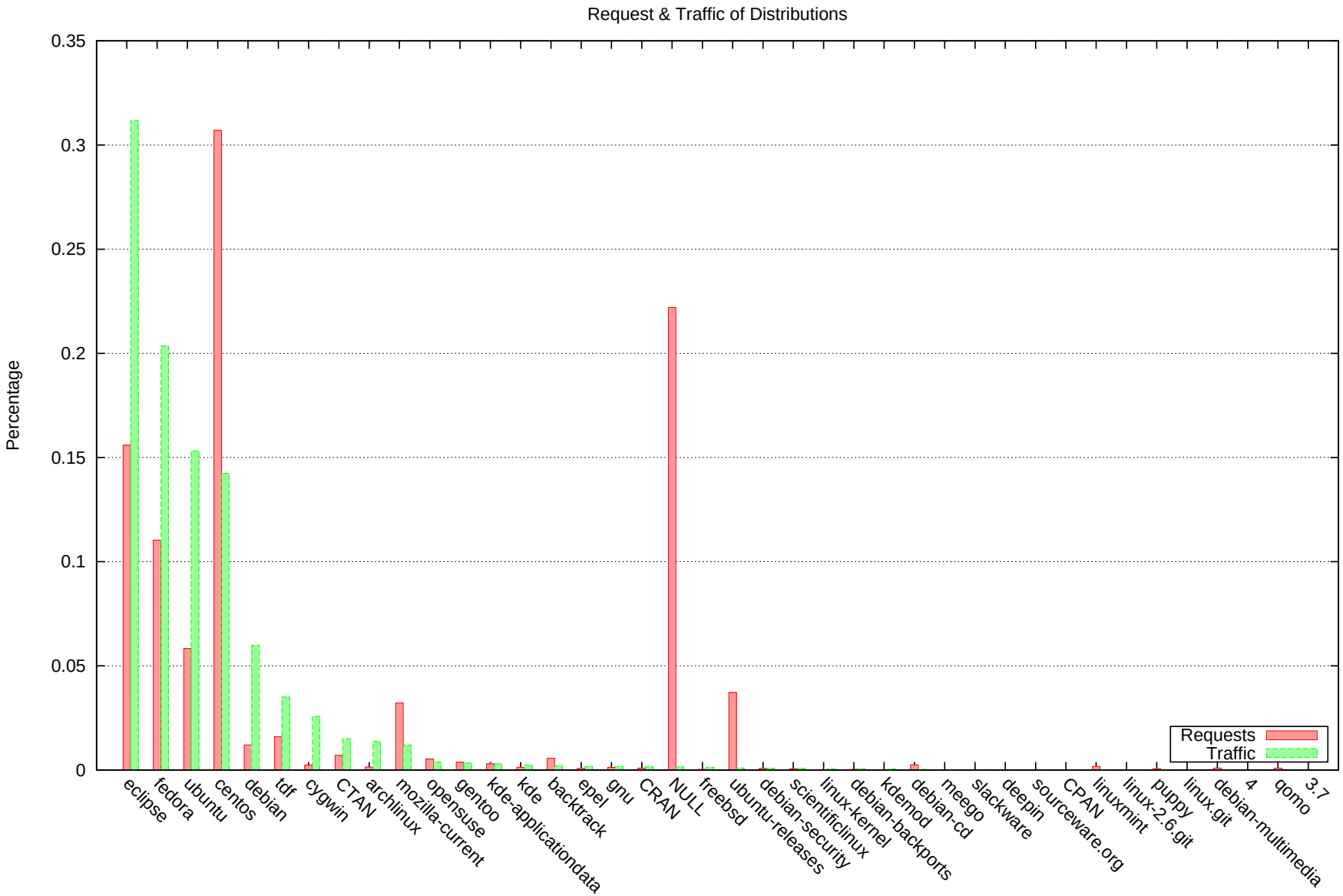


USTC Mirrors Usage (IPv4 only)

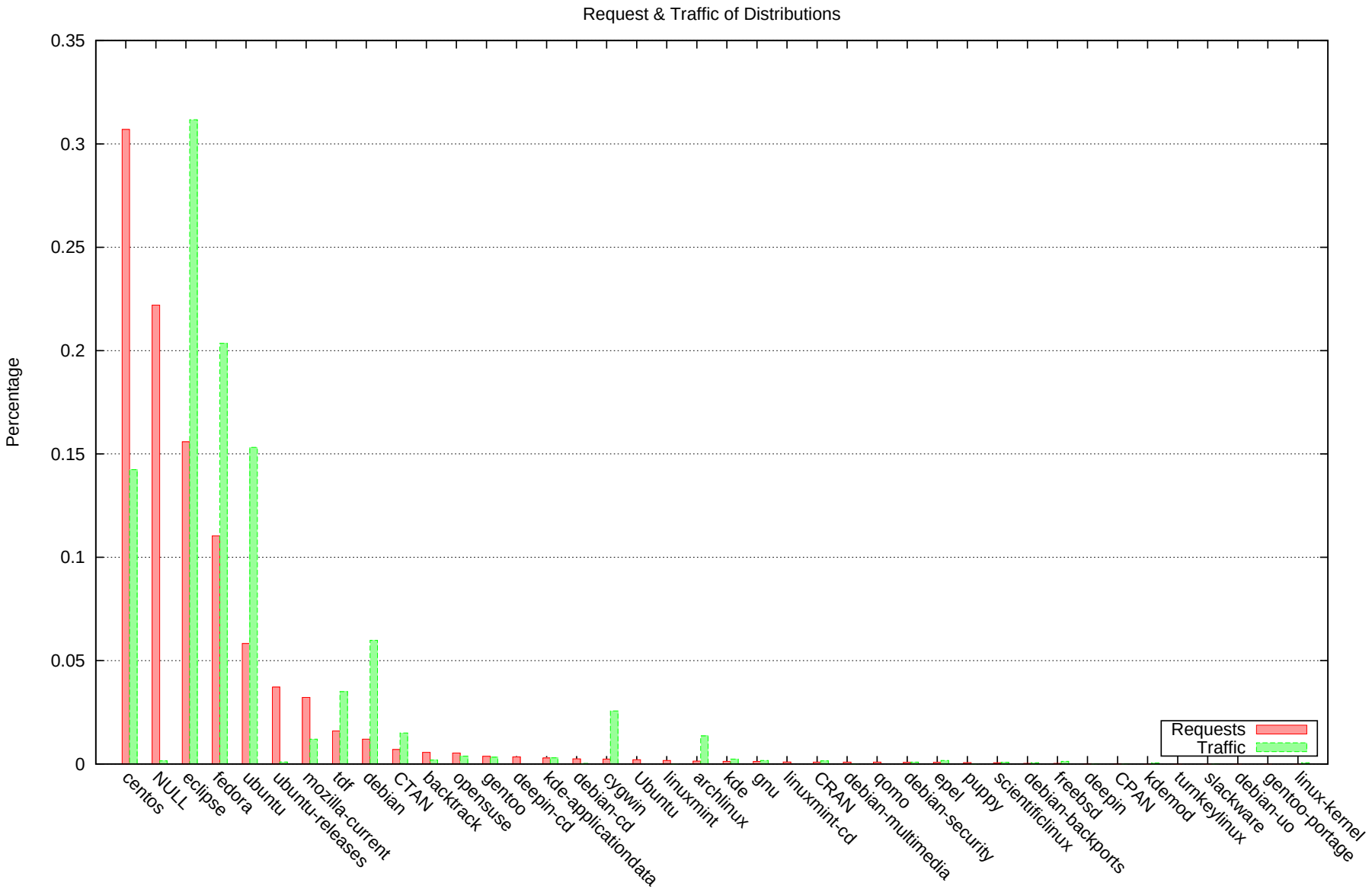
IP range	Requests	Traffic	Note
202.38.64.0-202.38.95.255	994661 (0.30%)	149.3 GB (0.40%)	CERNET
210.45.64.0-210.45.79.255	191141 (0.06%)	68.31 GB (0.19%)	CERNET
210.45.112.0-210.45.127.255	243976 (0.07%)	37.59 GB (0.10%)	CERNET
211.86.144.0-211.86.159.255	81035 (0.02%)	24.96 GB (0.07%)	CERNET
222.195.64.0-222.195.95.255	319435 (0.10%)	86.88 GB (0.24%)	CERNET
114.214.160.0-114.214.255.255	0	0	CERNET
210.72.22.0-210.72.22.255	3622 (0.00%)	11.86 MB (0.00%)	TechNet (?)
218.22.21.0-218.22.21.31	1 (0.00%)	0.01 MB (0.00%)	China Telecom
218.104.71.160-218.104.71.175	0	0	China Unicom
202.141.160.0-202.141.175.255	123455 (0.04%)	12.60 GB (0.03%)	China Telecom
202.141.176.0-202.141.191.255	187 (0.00%)	120.4 MB (0.00%)	China Mobile
Total	1957513 (0.60%)	379.77 GB (1.03%)	USTC IPv4

- Data source of USTC IP range: http://lib.ustc.edu.cn/ustcip.html

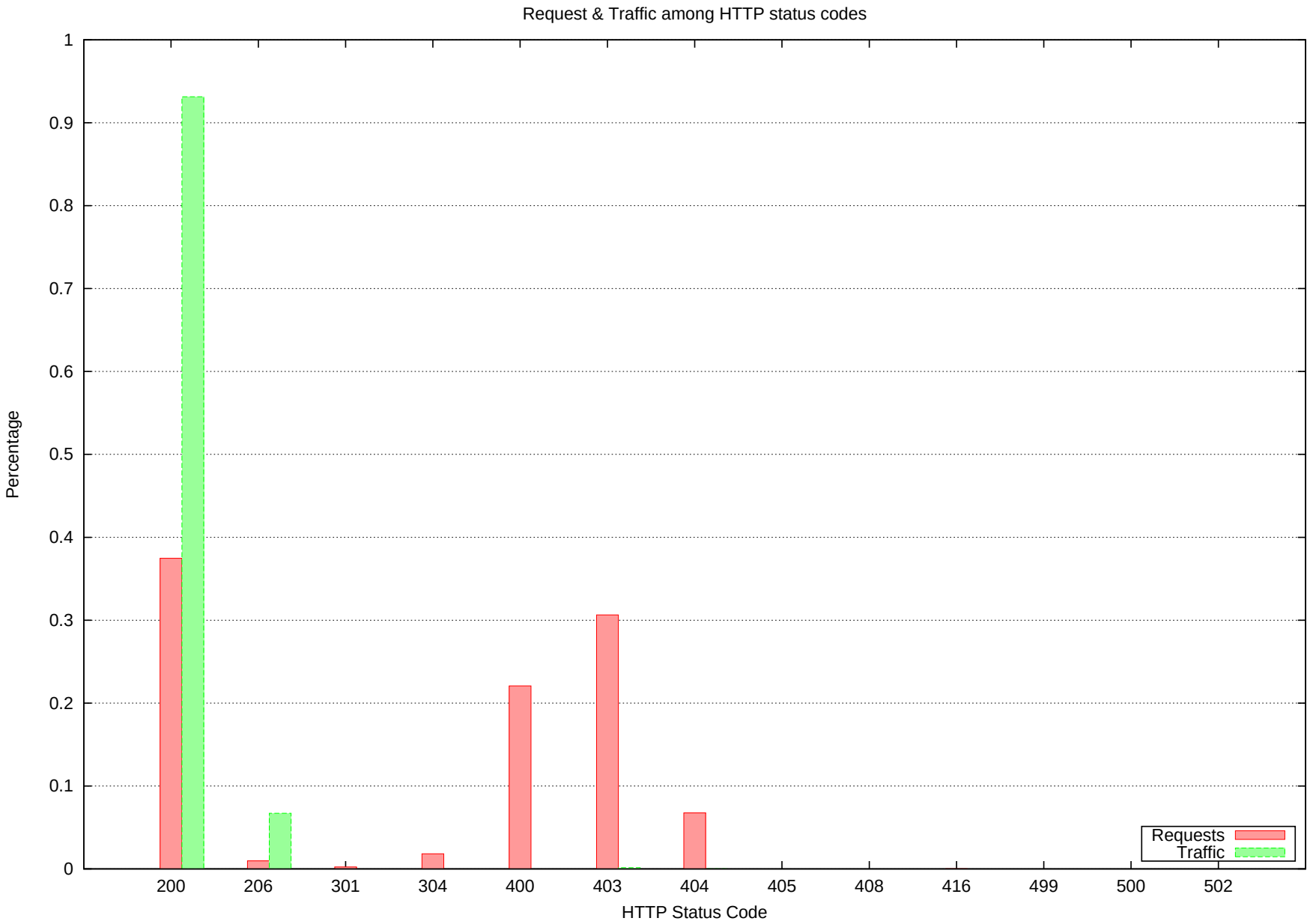
Requests & Traffic of distributions



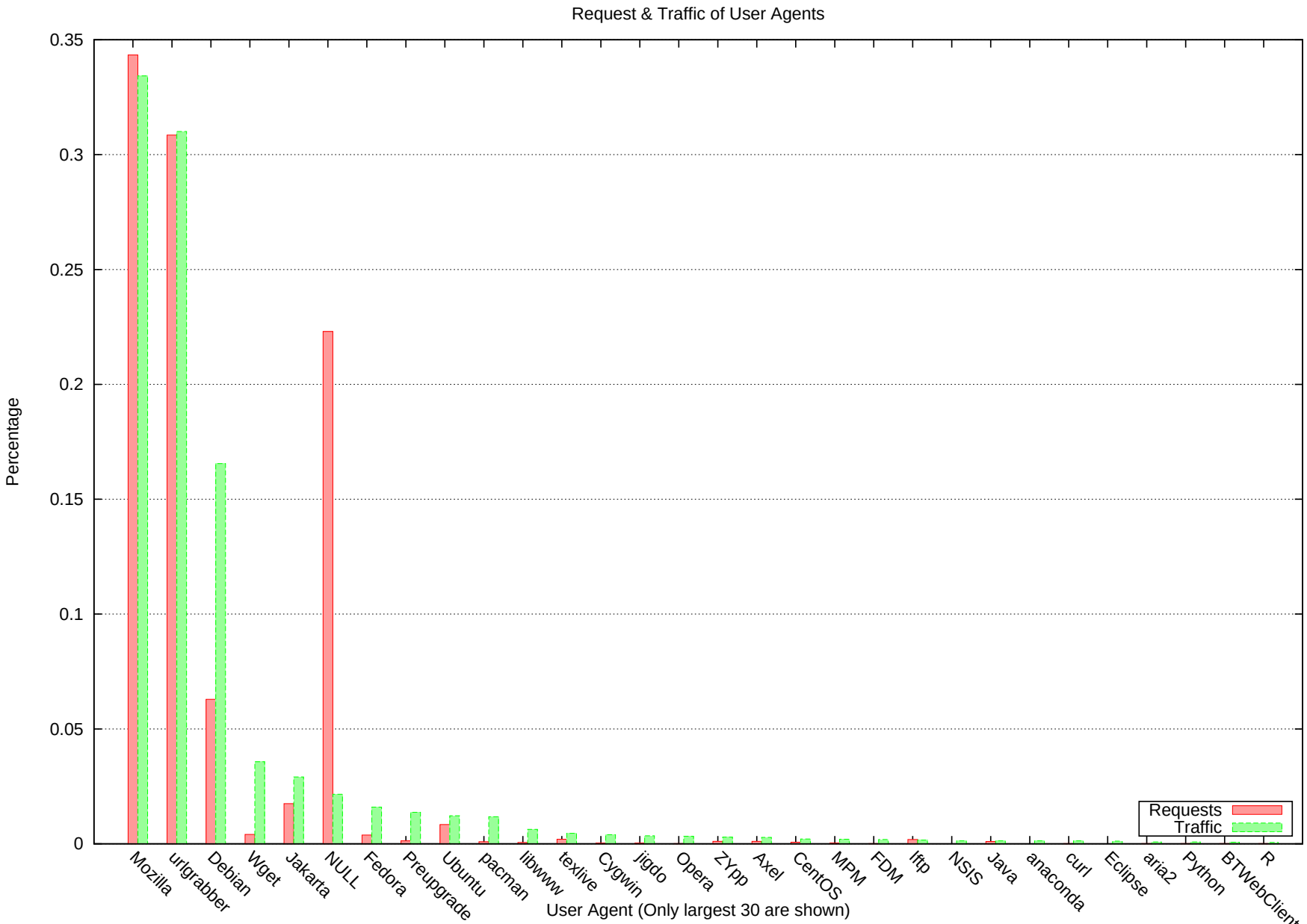
Requests & Traffic of distributions



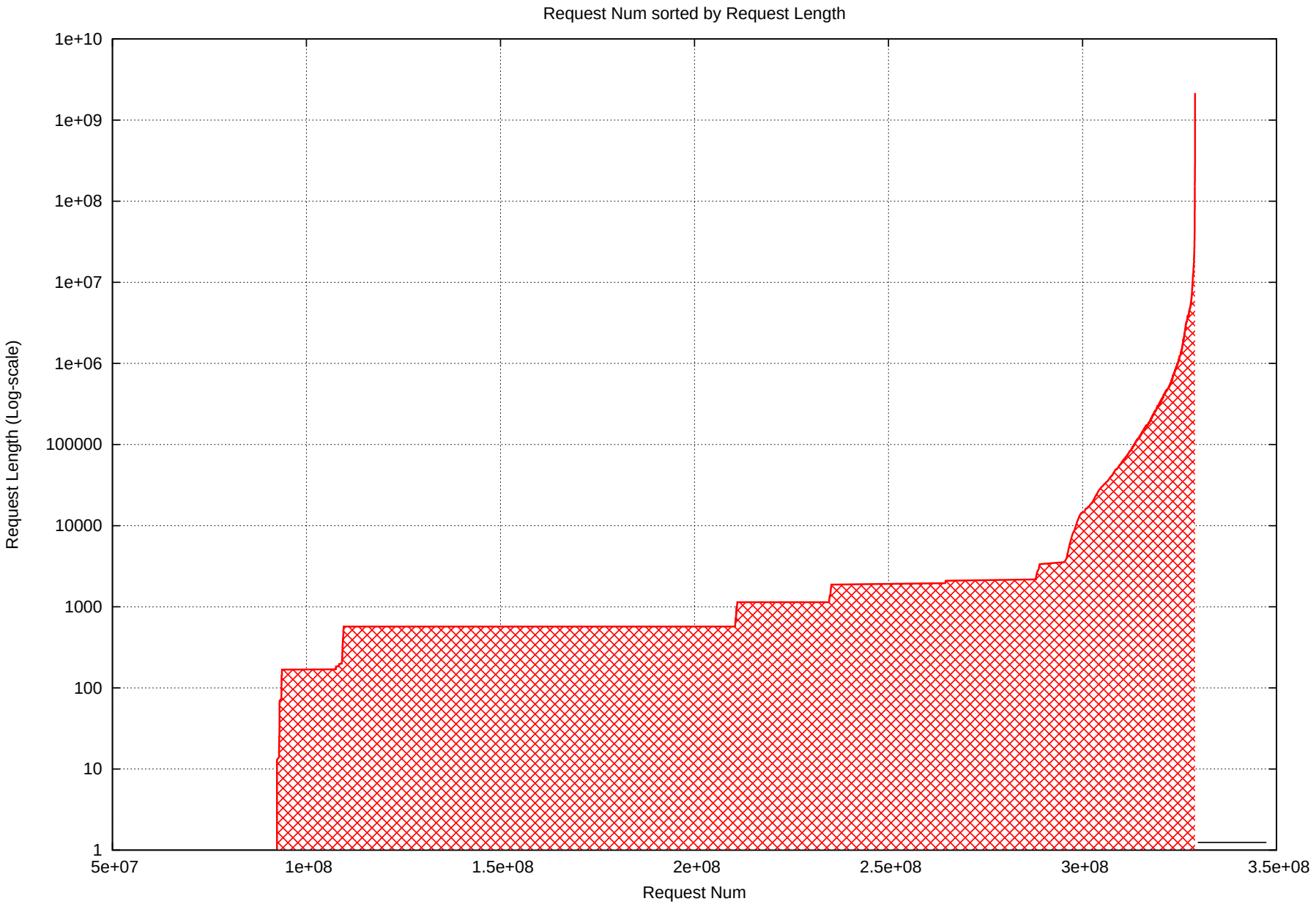
Requests & Traffic among HTTP Status Codes



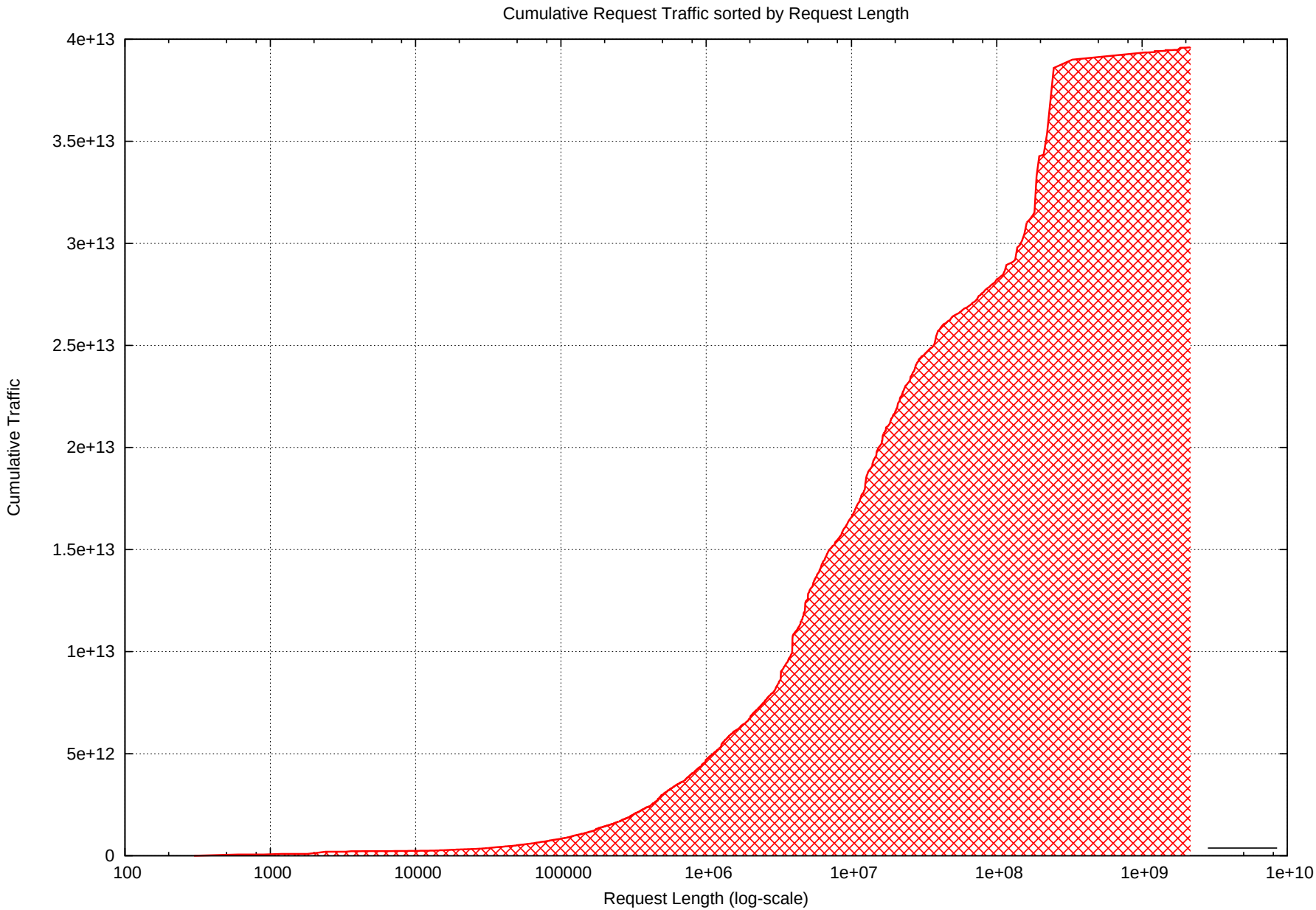
Requests & Traffic among User Agents



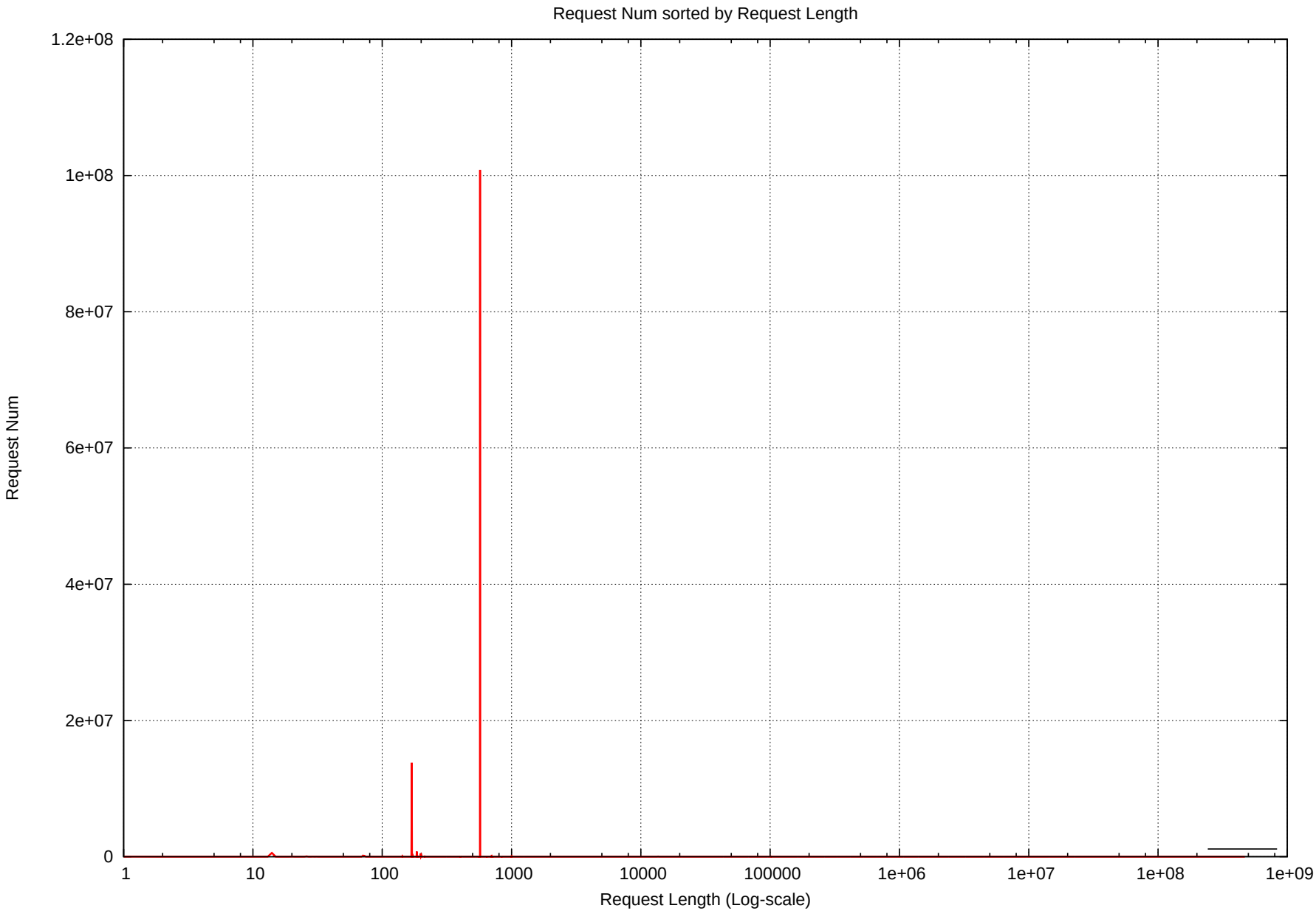
Traffic per Request order by Length



Cumulative Traffic sorted by Request Length

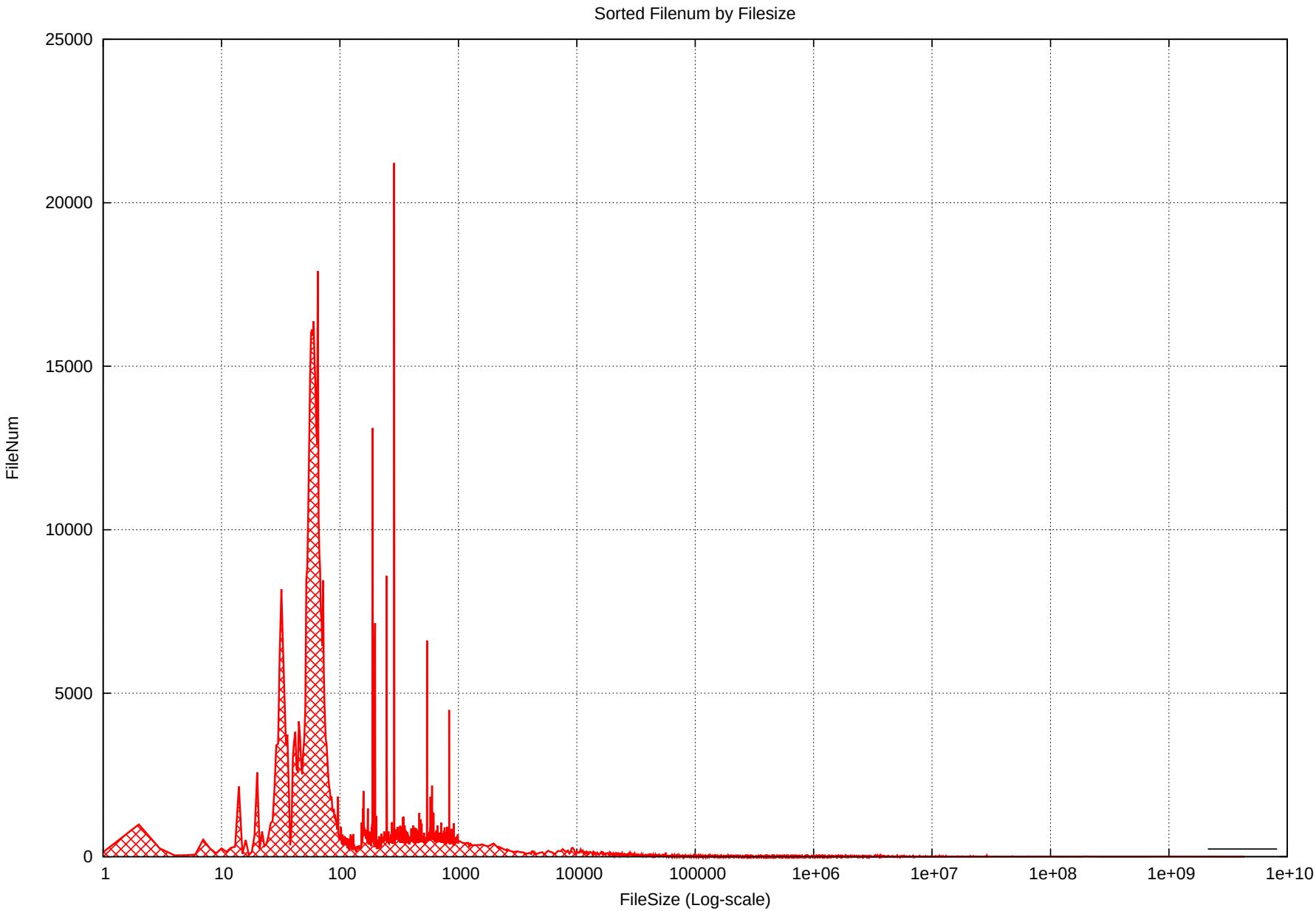


Request Num sorted by Request Length

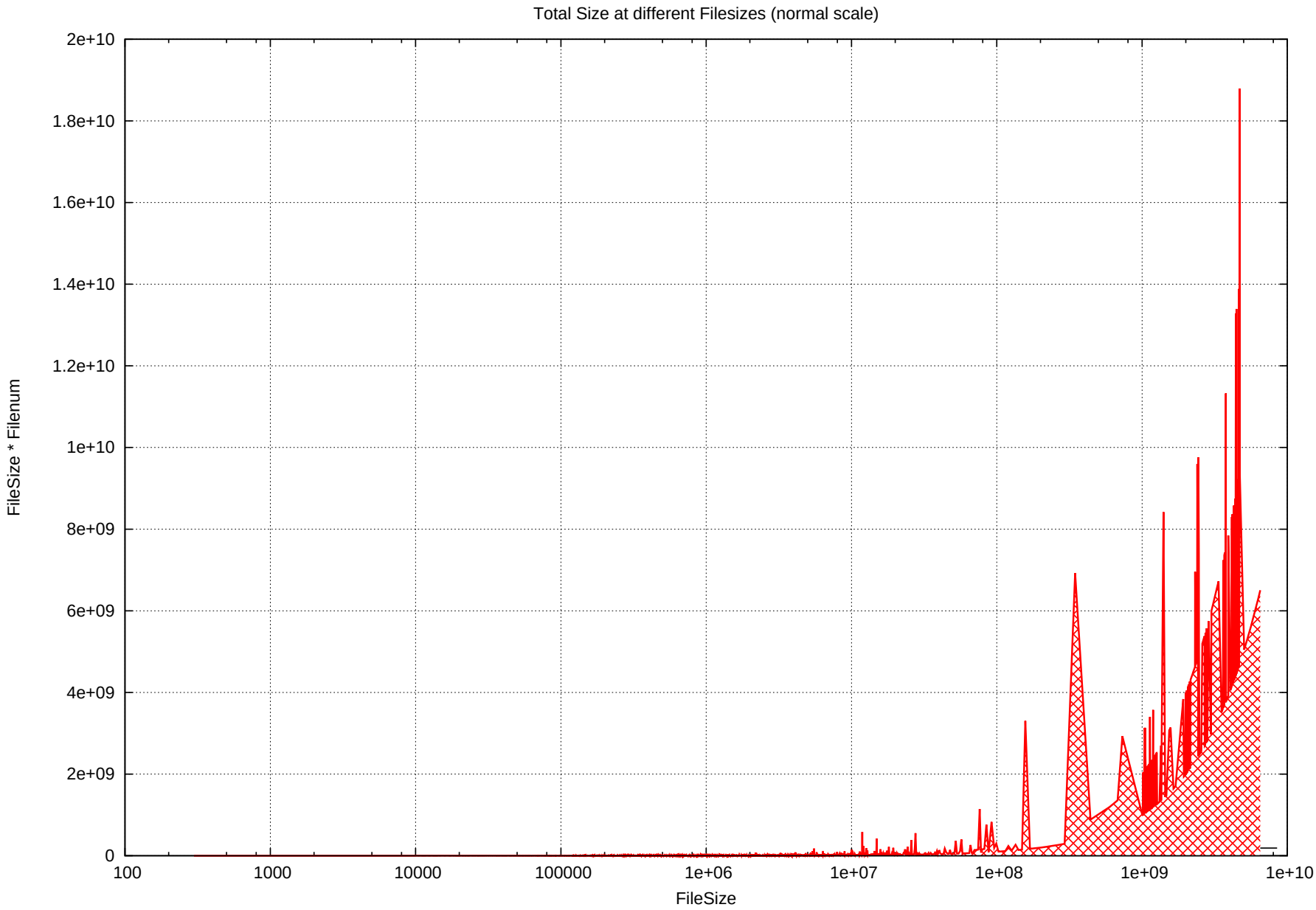


Outline

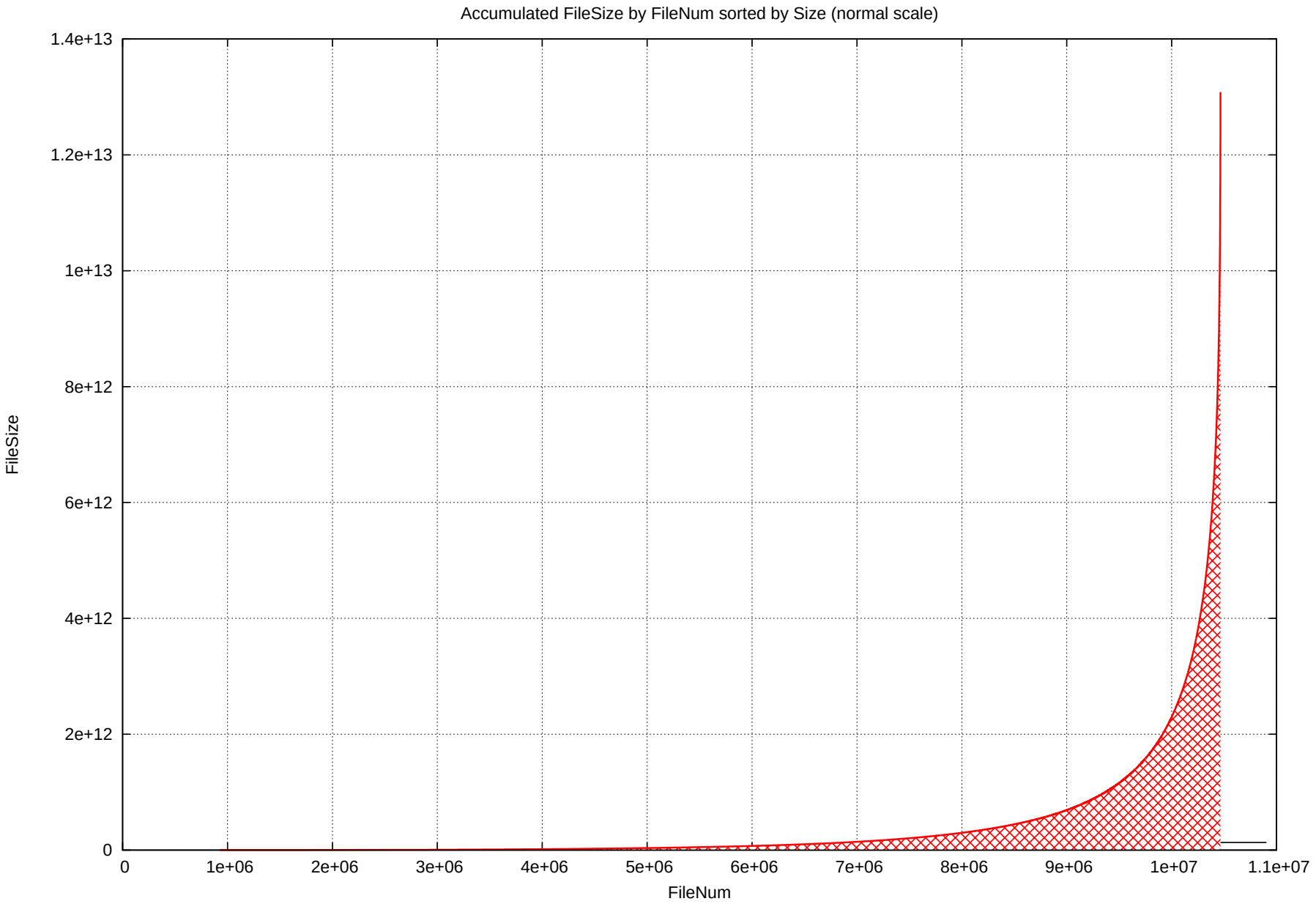
File Number at different FileSizes (log scale)



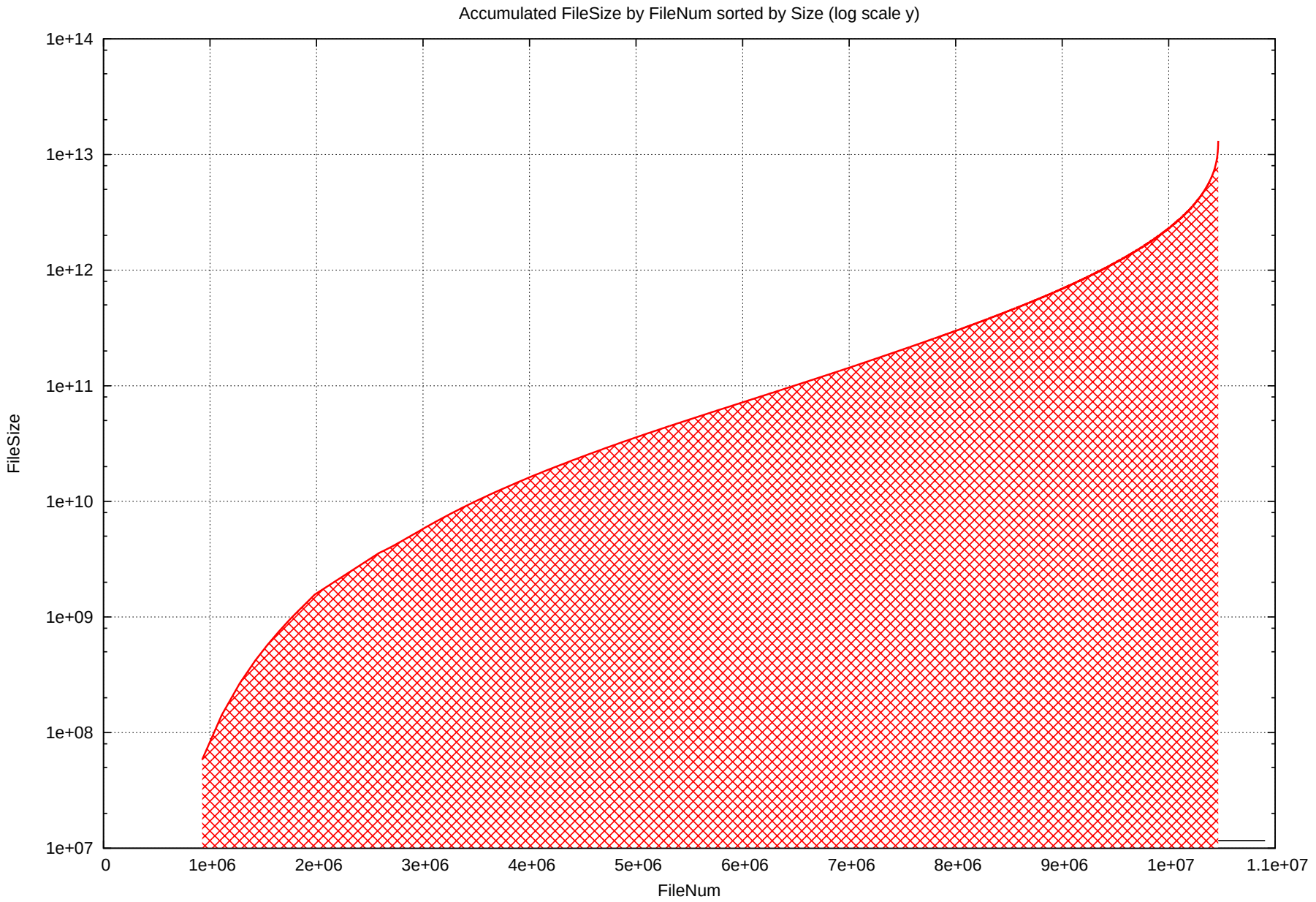
Total Size at different Filesizes (log scale)



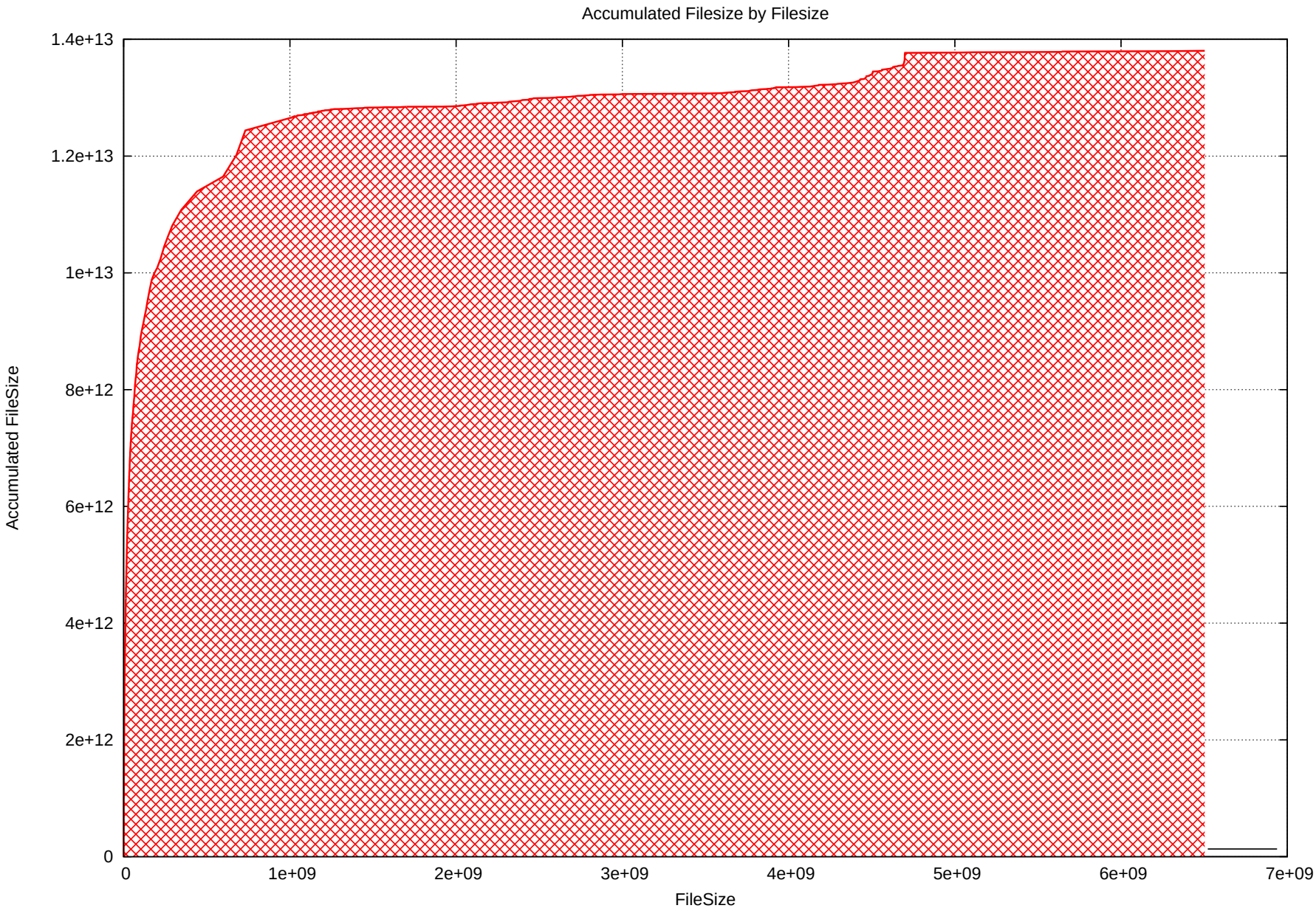
Cumulative Filesize per File Num (normal scale)



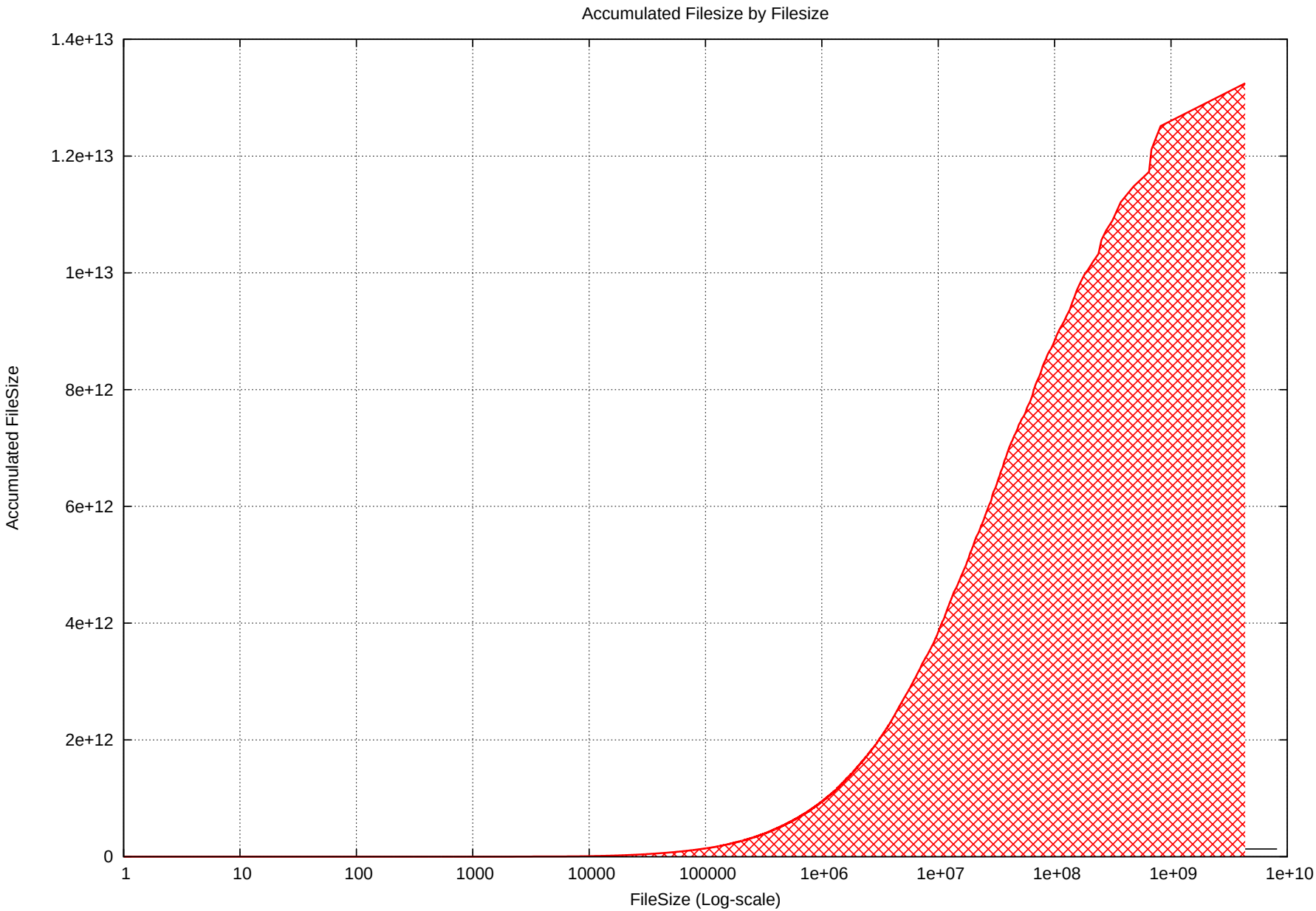
Cumulative Filesize per File Num (log scale)



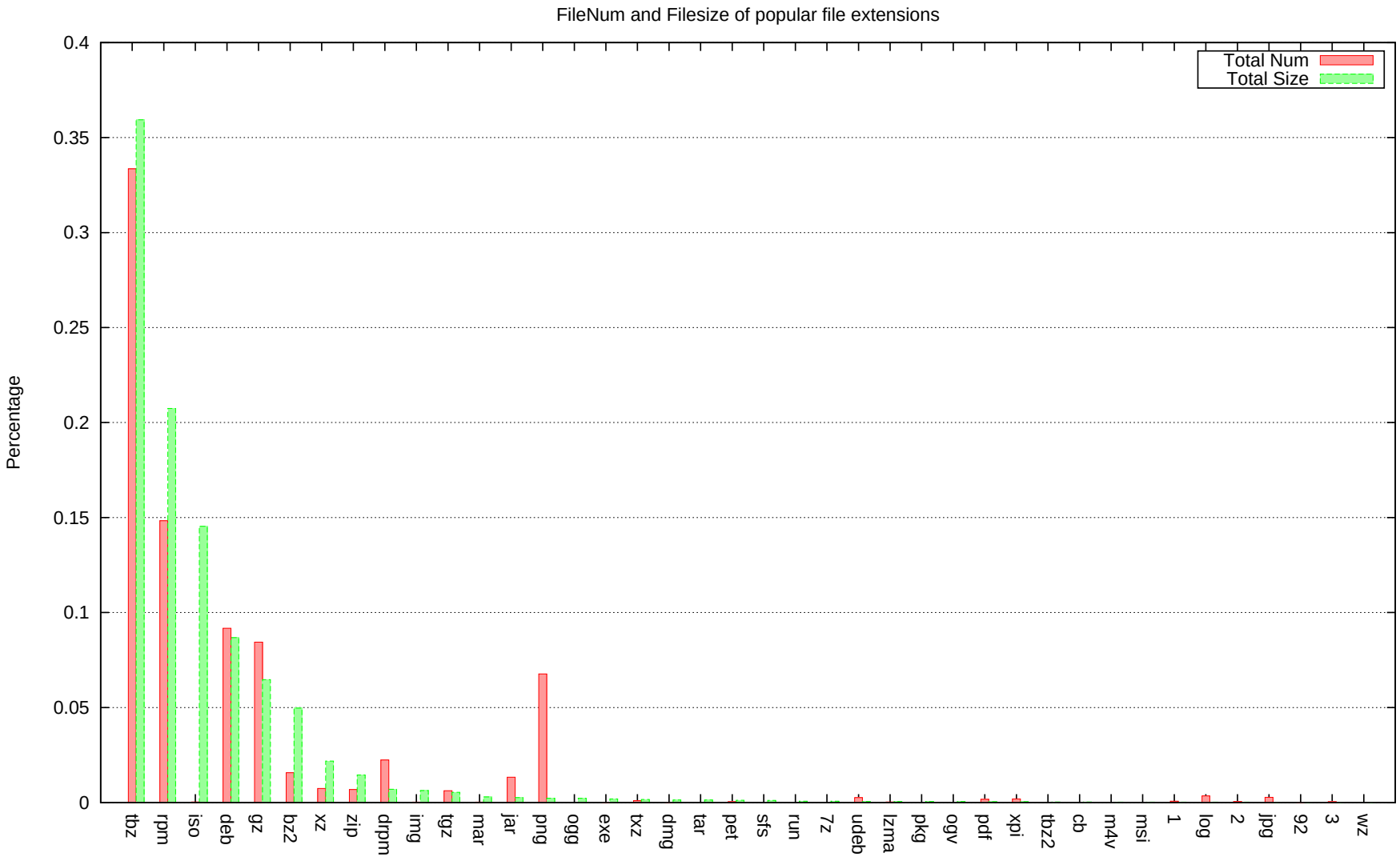
Cumulative Filesize per FileSize (normal scale)



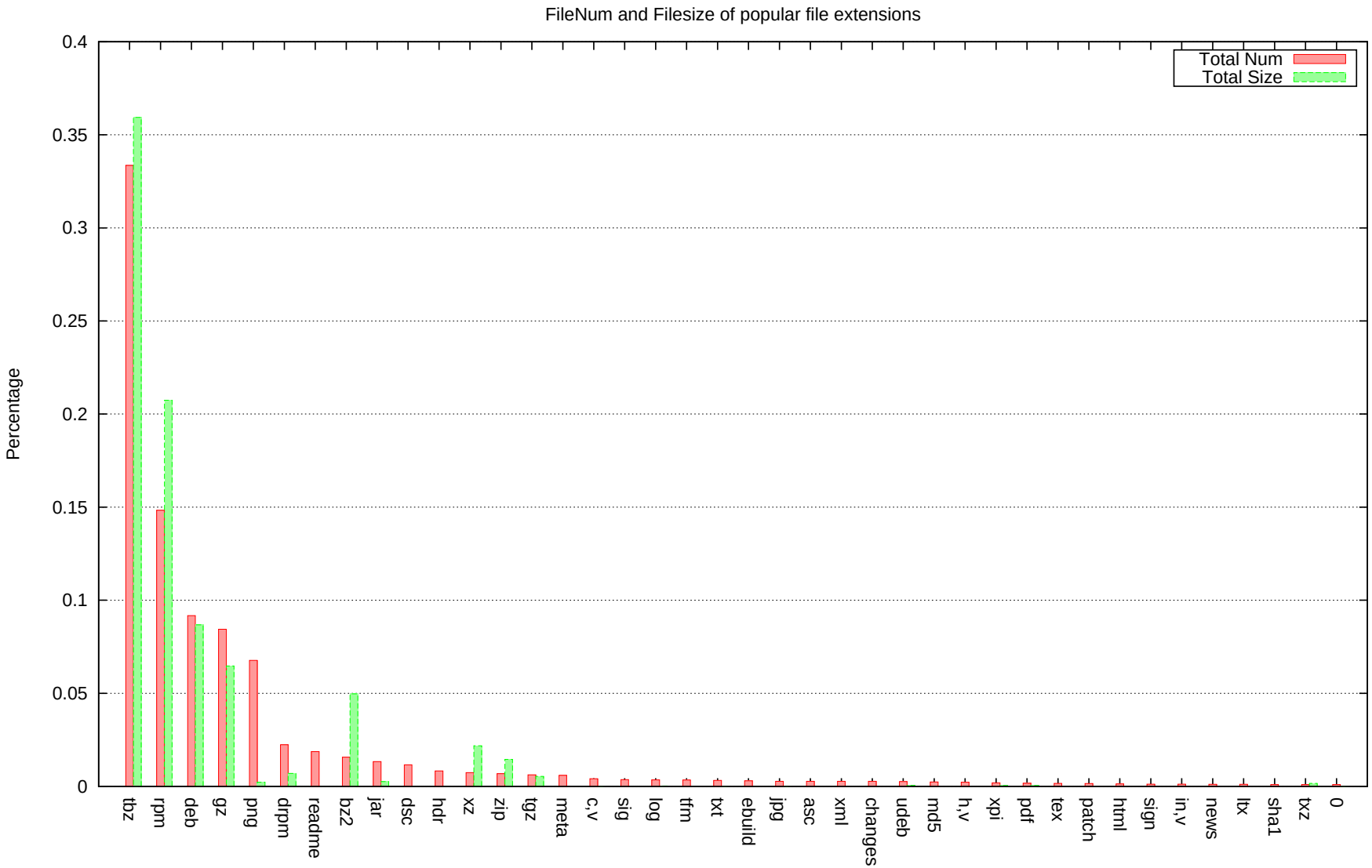
Cumulative Filesize per FileSize (log scale)



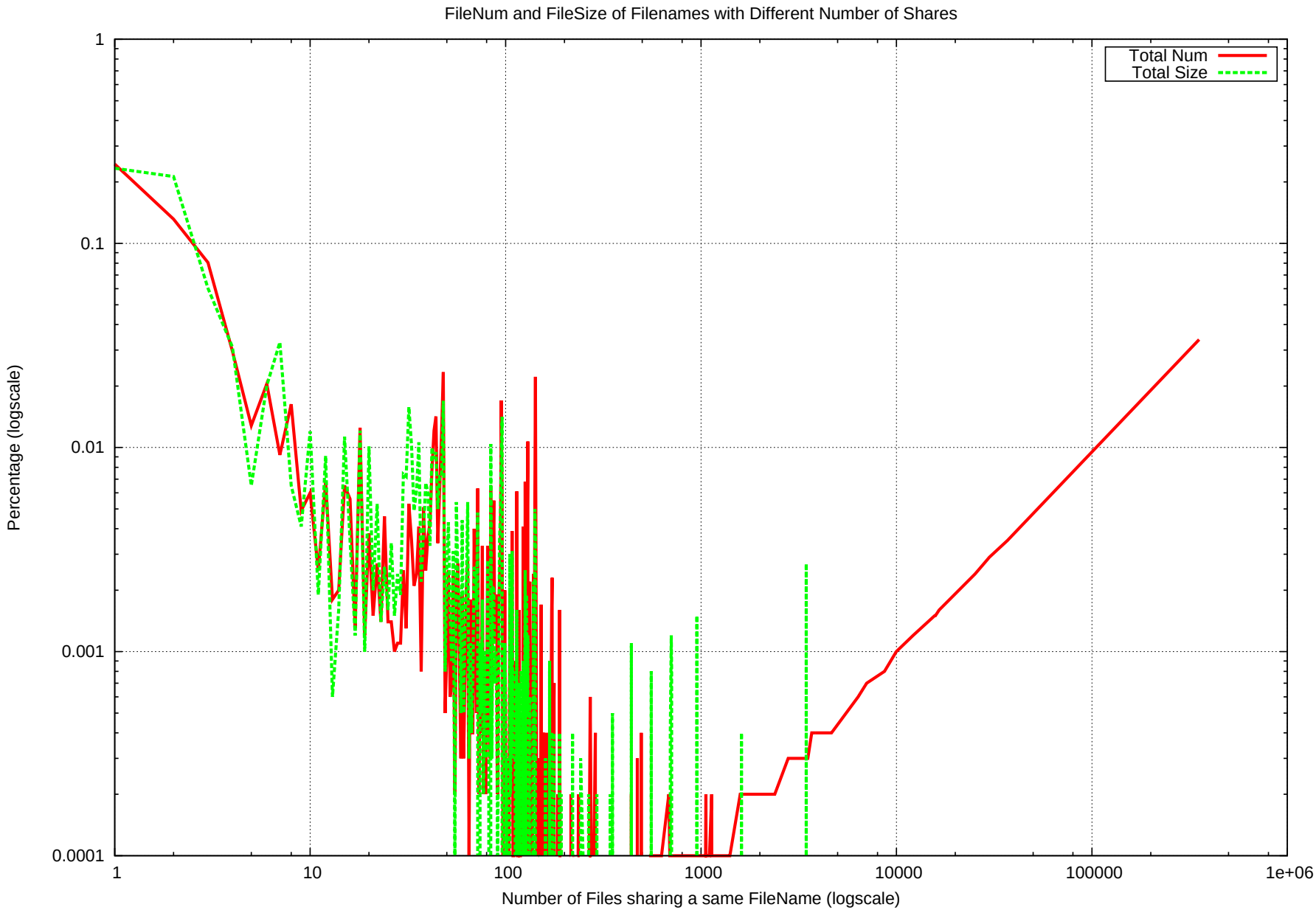
File Extensions TOP 40 (order by Total Size)



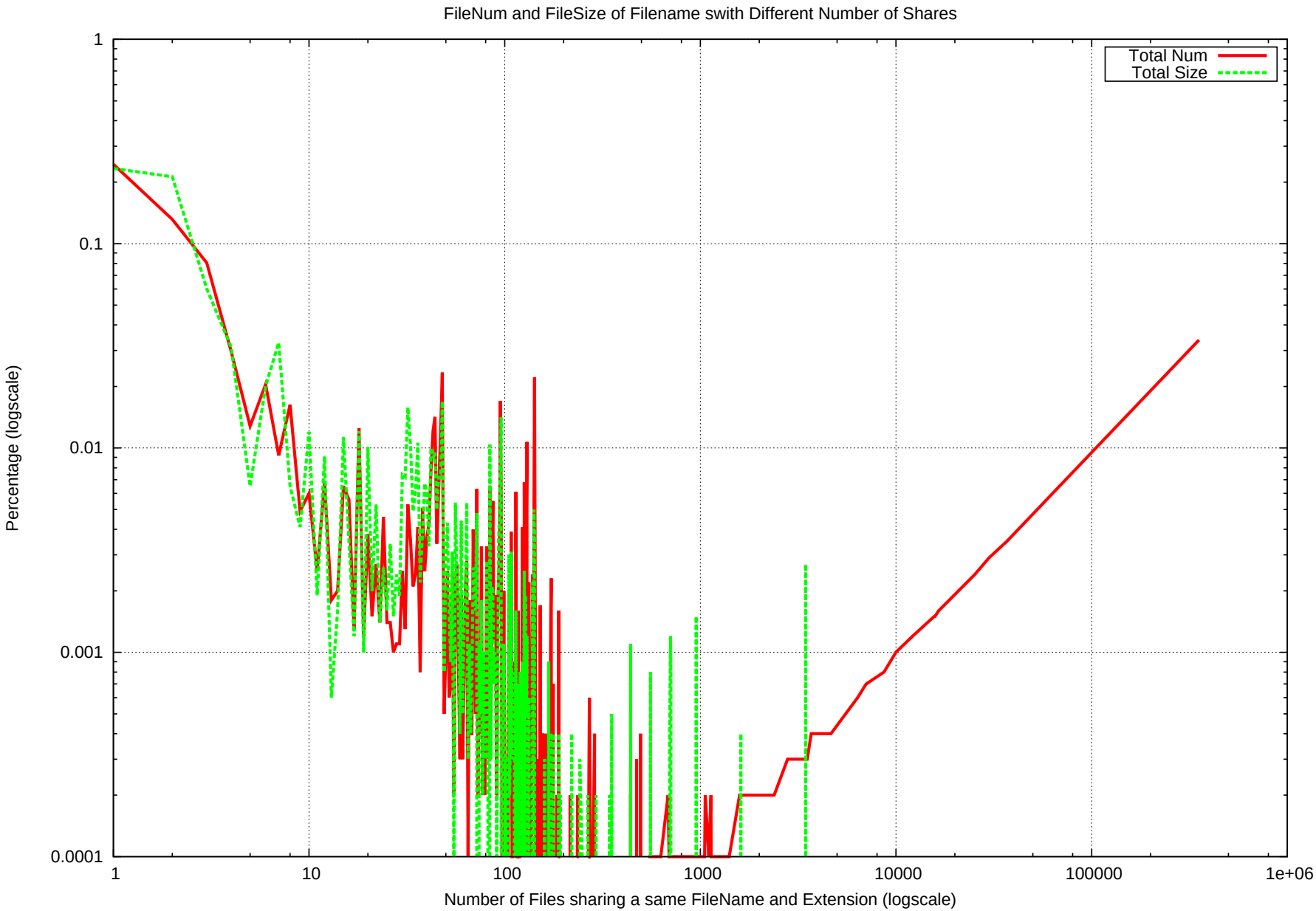
File Extensions TOP 40 (order by File Num)



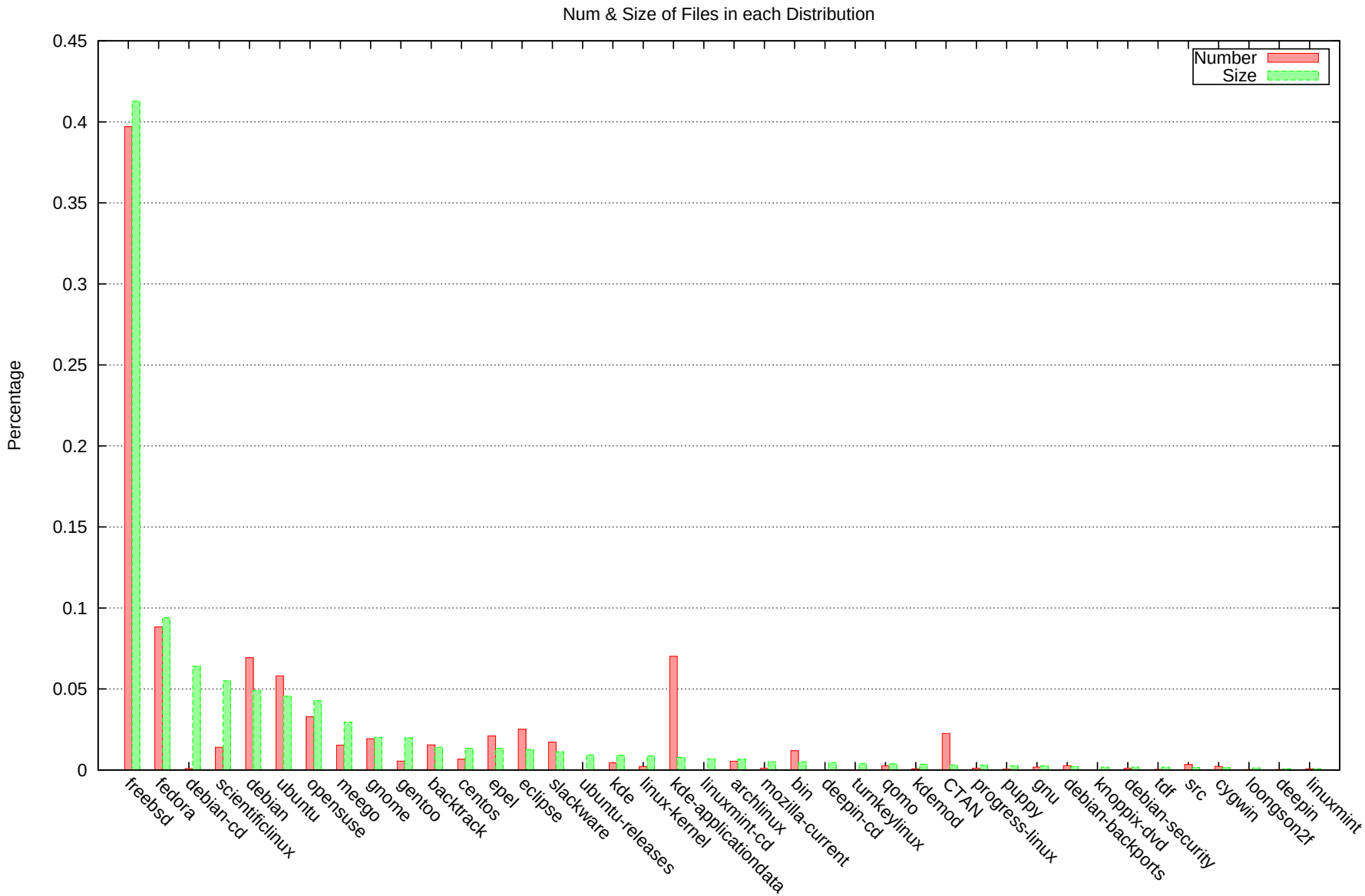
How many files share a same filename (extension excluded)



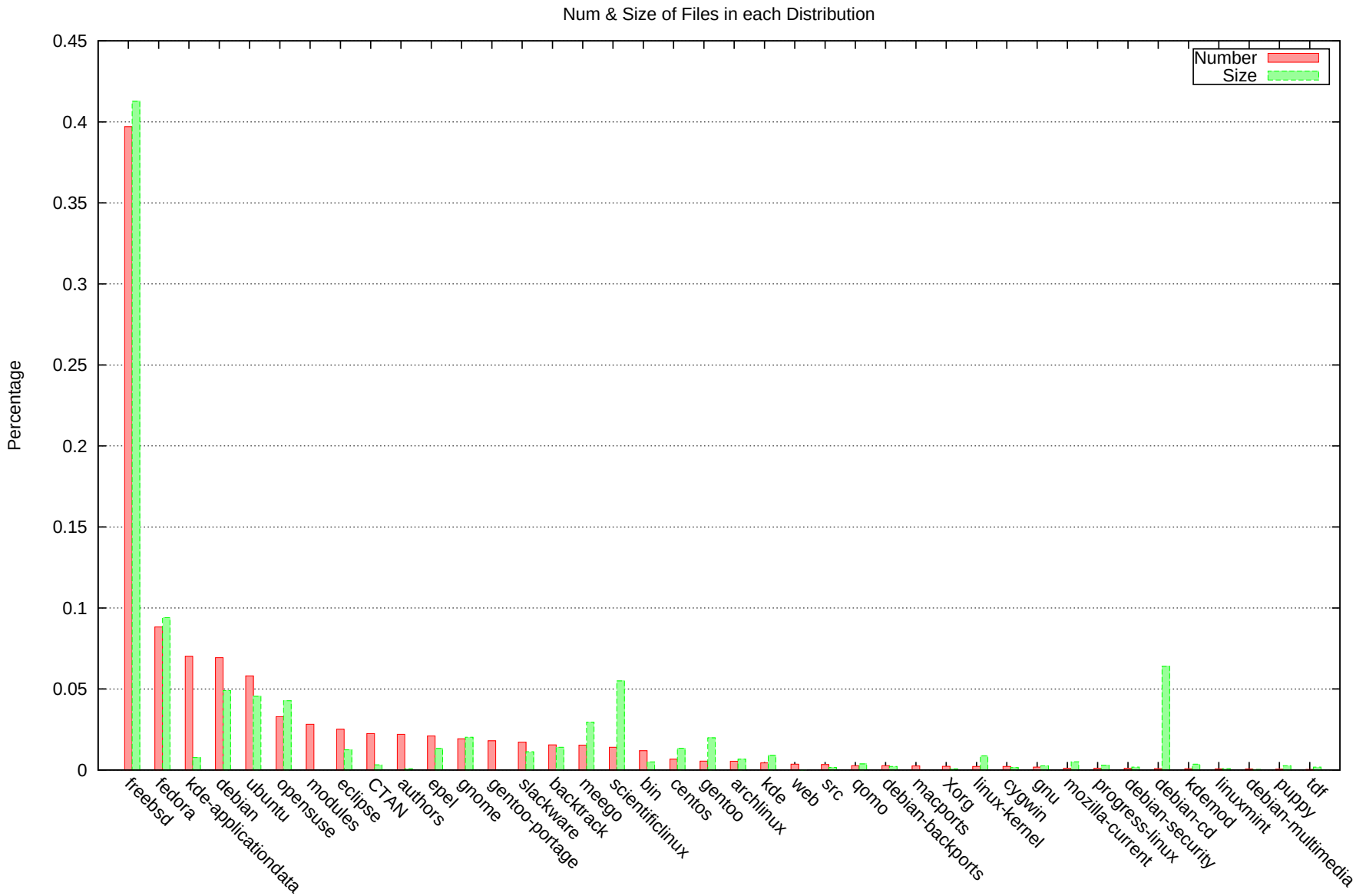
How many files share a same filename and extension



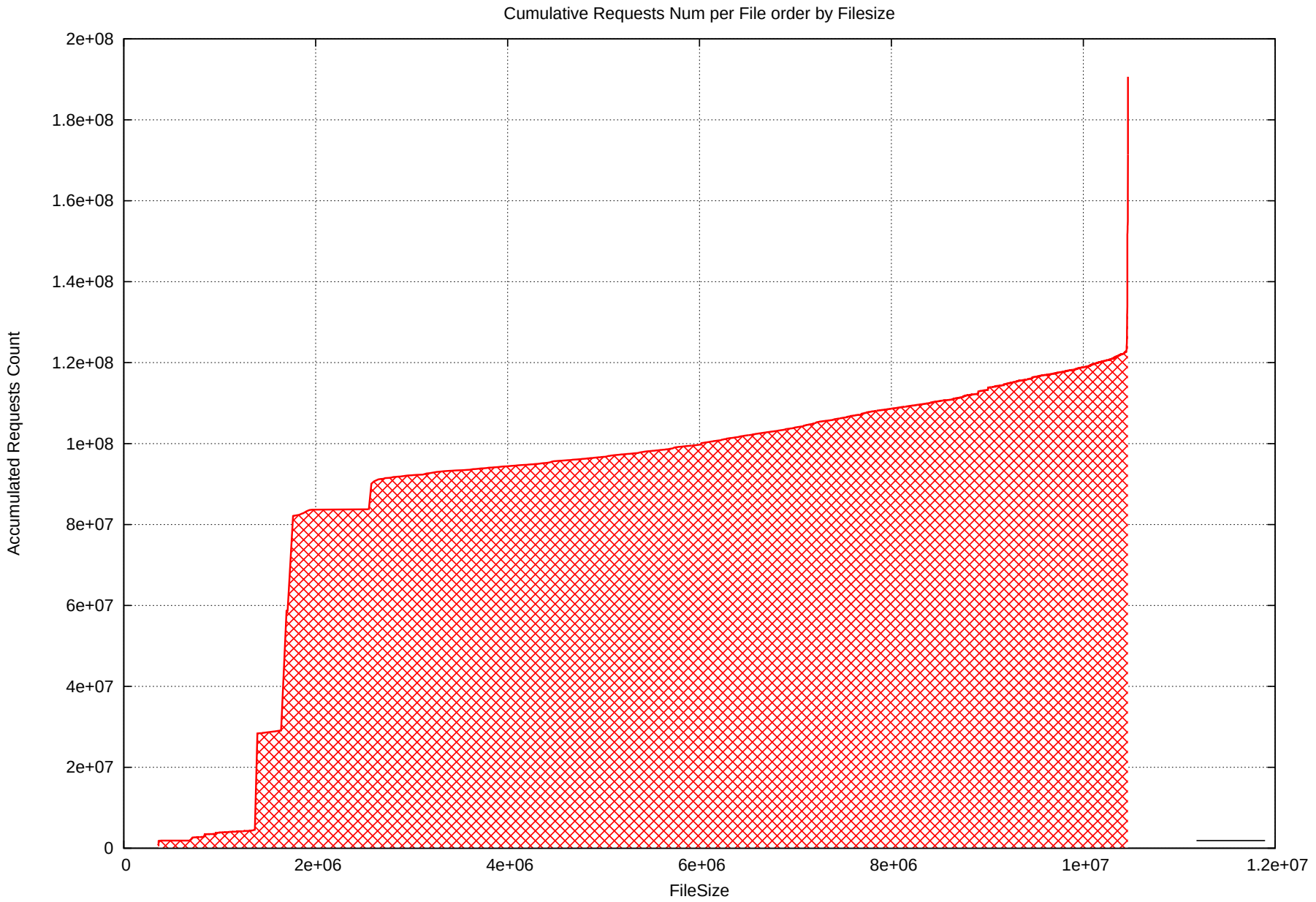
Num & Size of Files in each Distribution (sorted by Size)



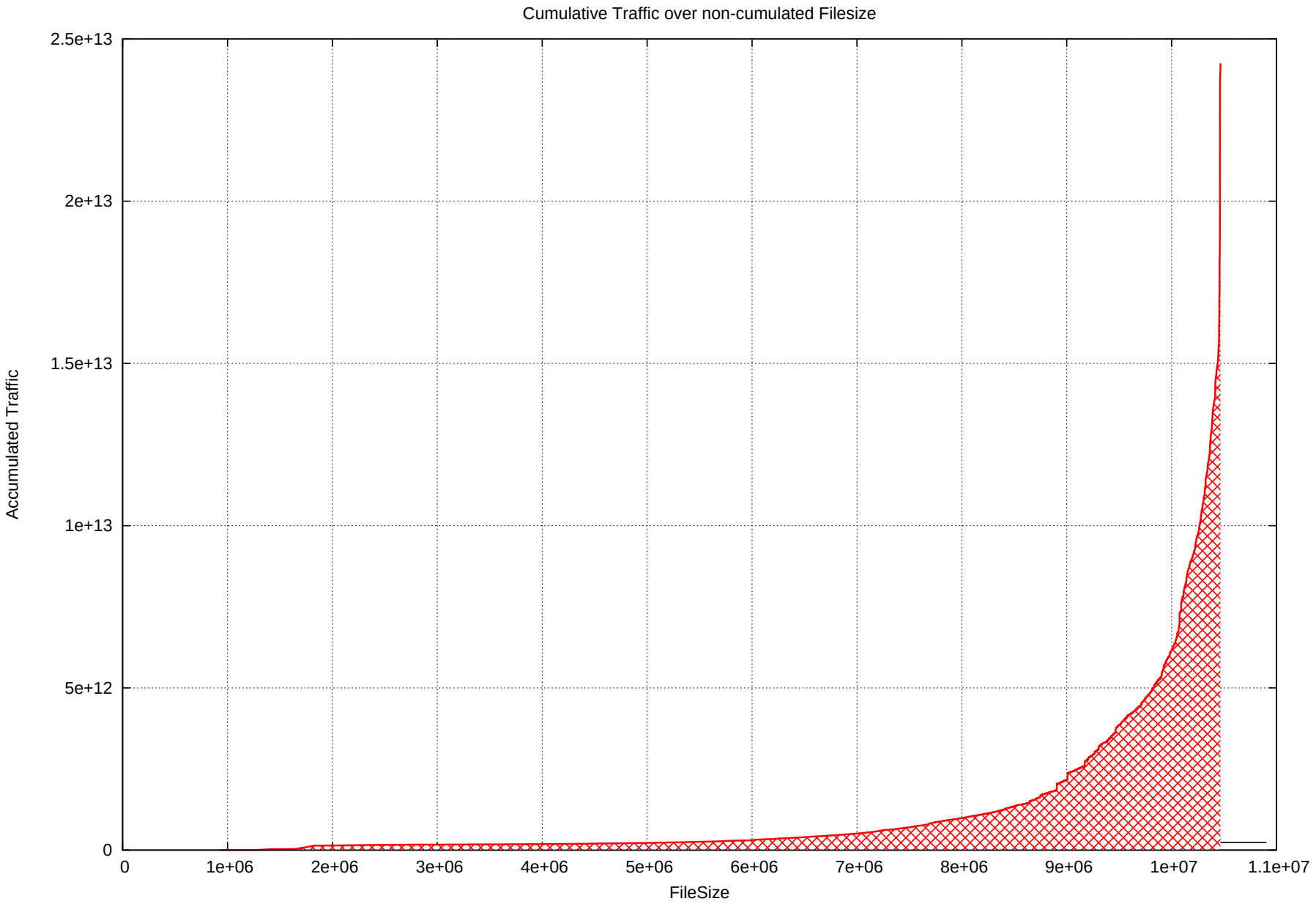
Num & Size of Files in each Distribution (sorted by Num)



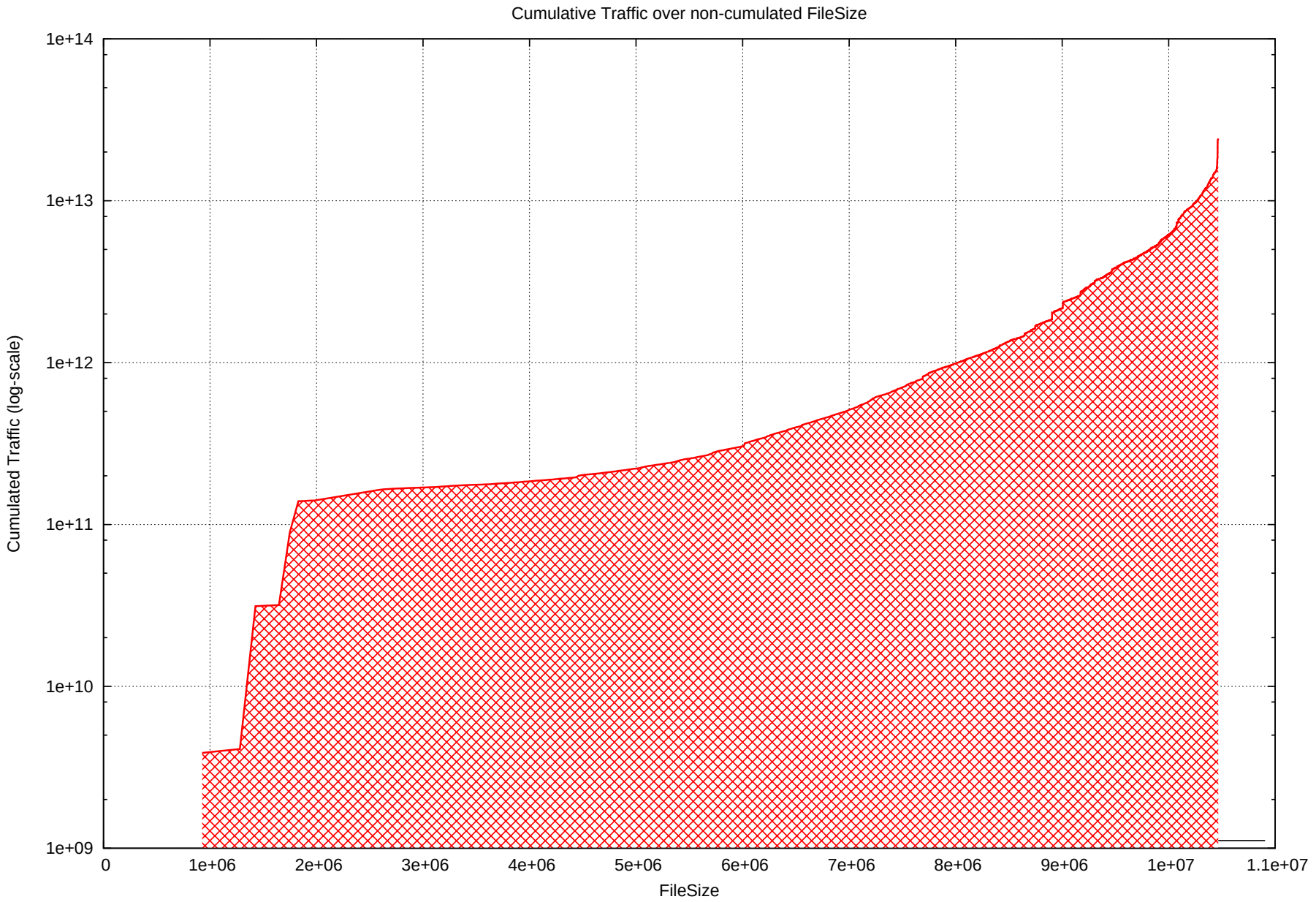
Cumulative Requests over FileSize order by Size



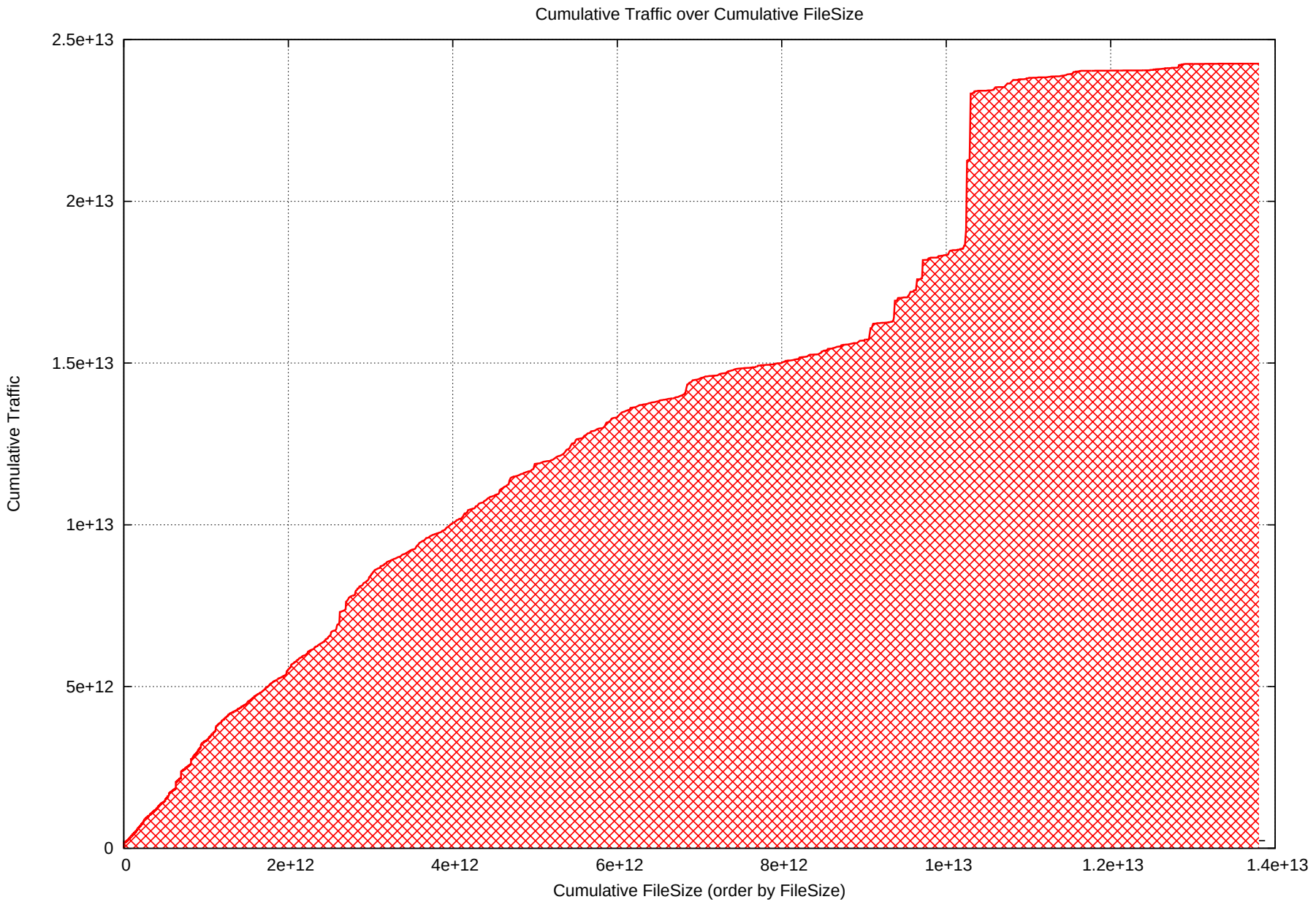
Cumulative Traffic over non-cumu. FileSize



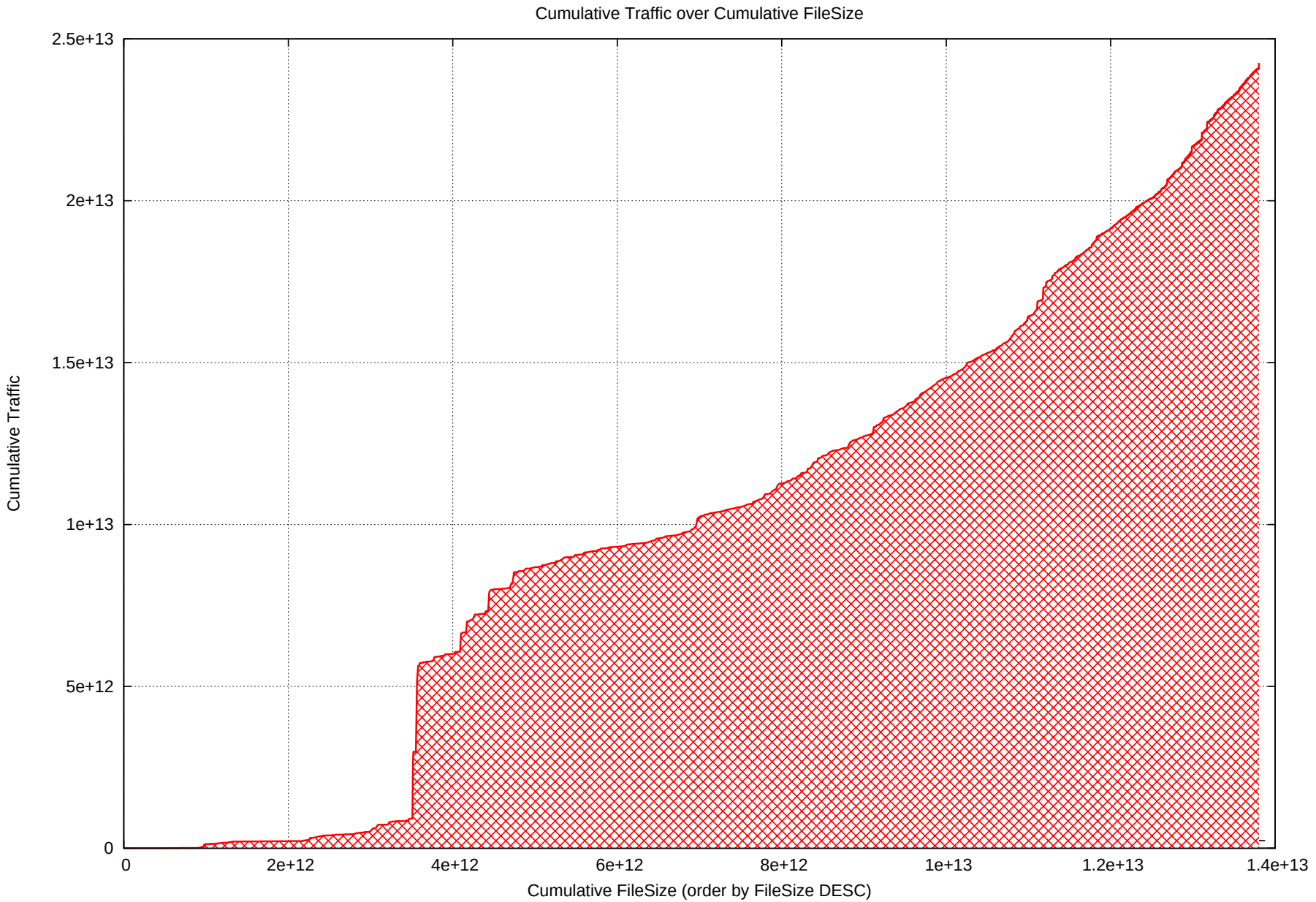
Cumulative Traffic over non-cumu. Filesize (log-scale)



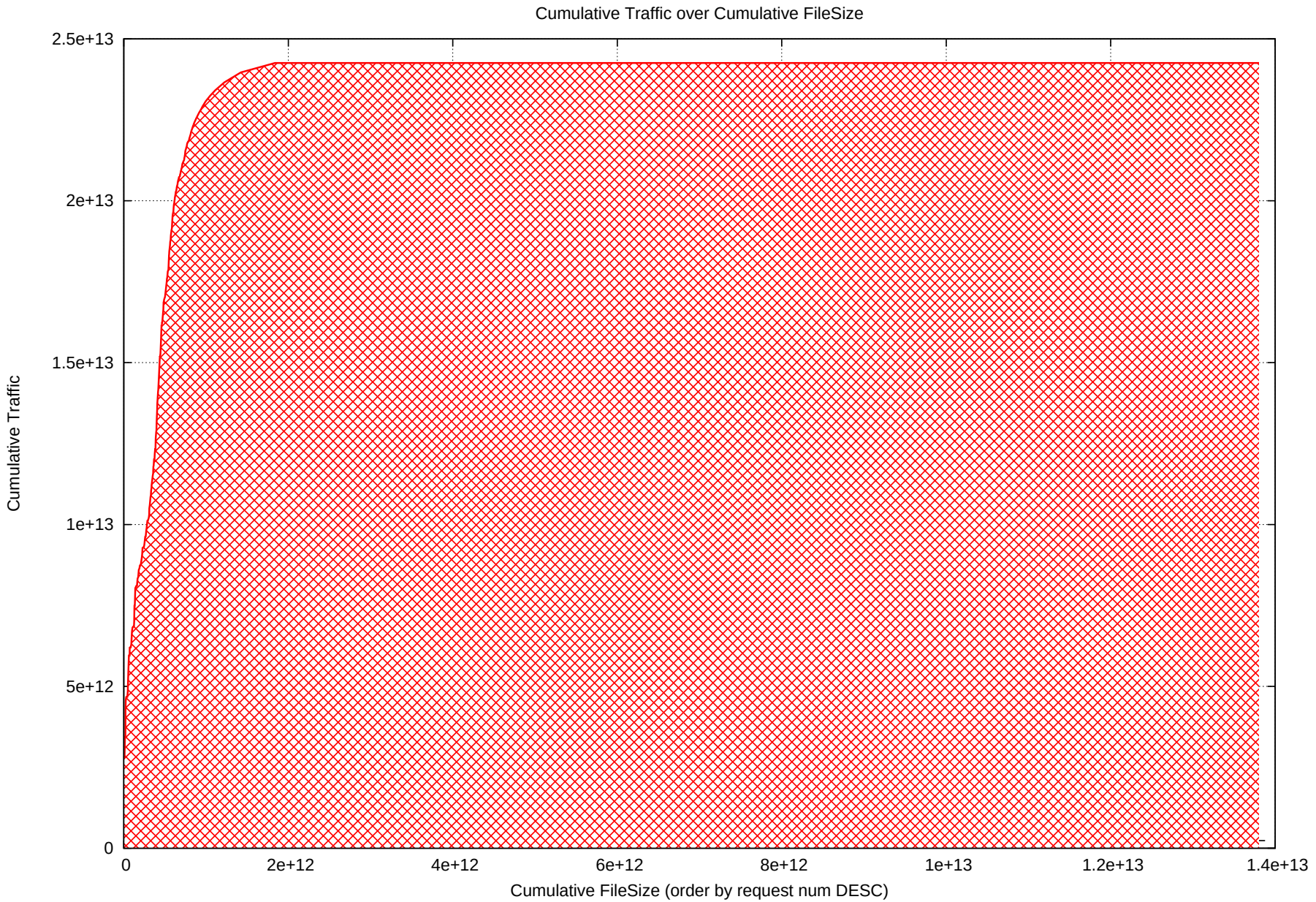
Cache 1: Cumu. Traffic over FileSize order by Size



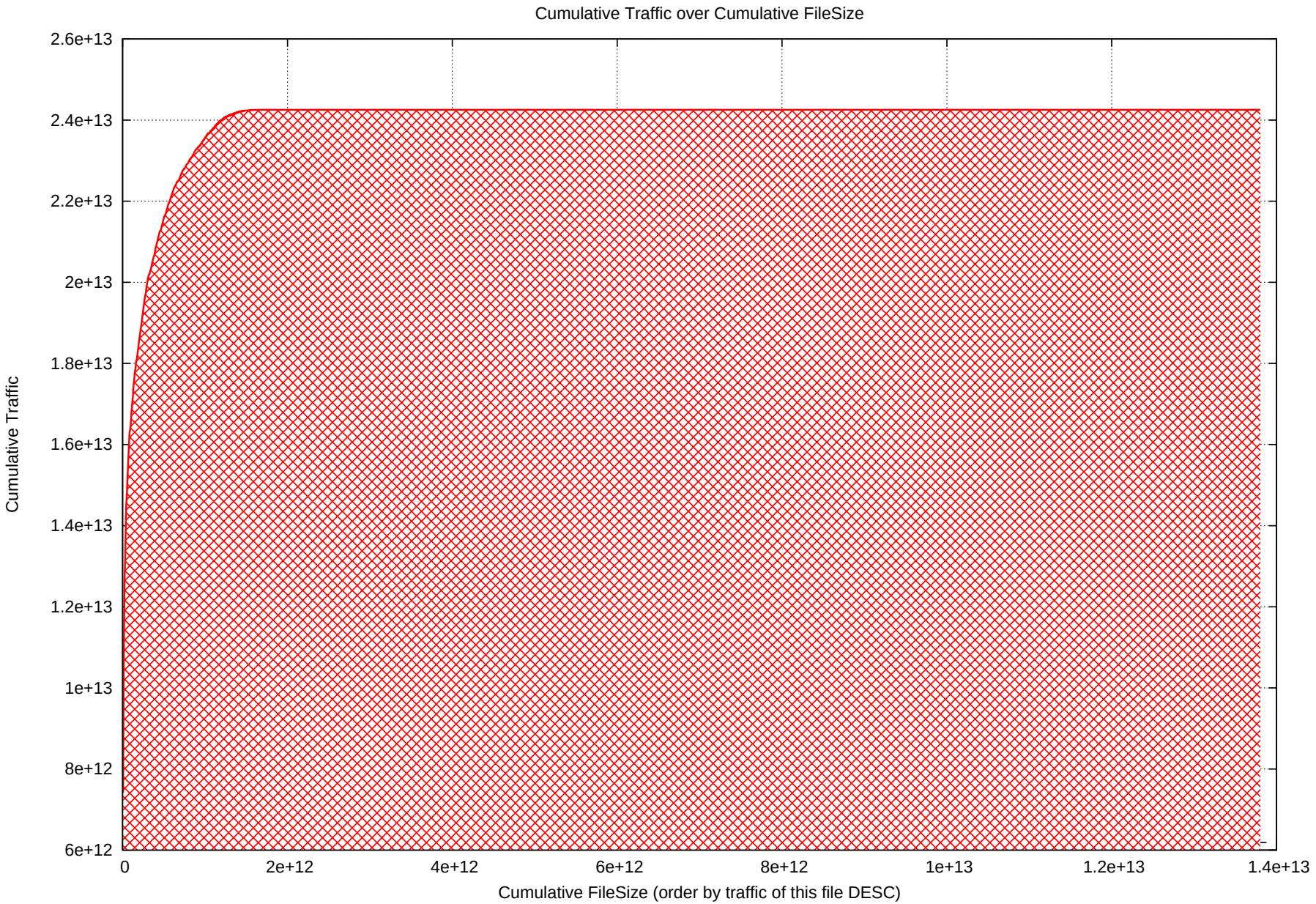
Cache 2: Cumu. Traffic over FileSize order by Size DESC



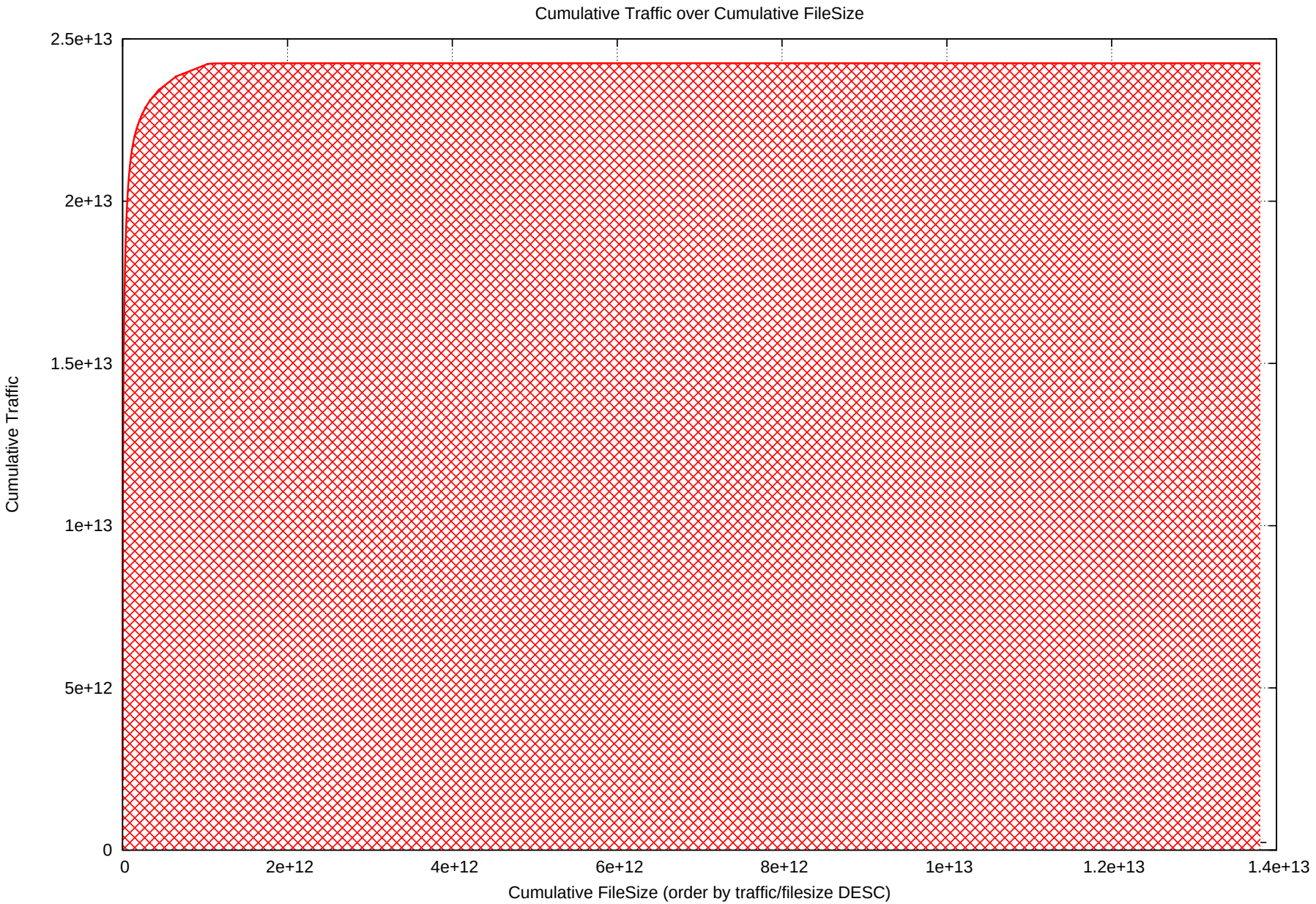
Cache 3: Cumu. Traffic over FileSize order by Req Num



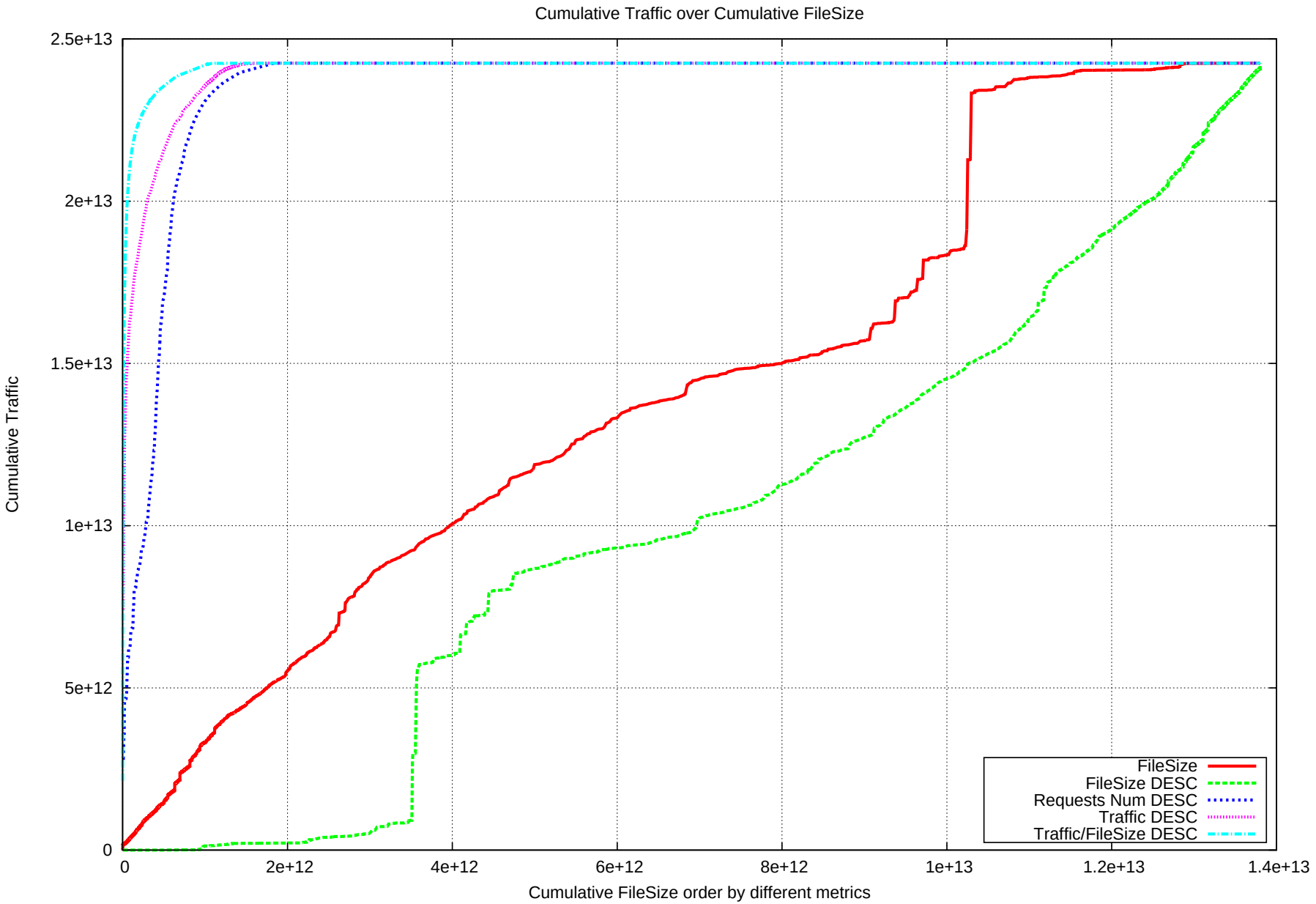
Cache 4: Cumu. Traffic over FileSize order by Traffic



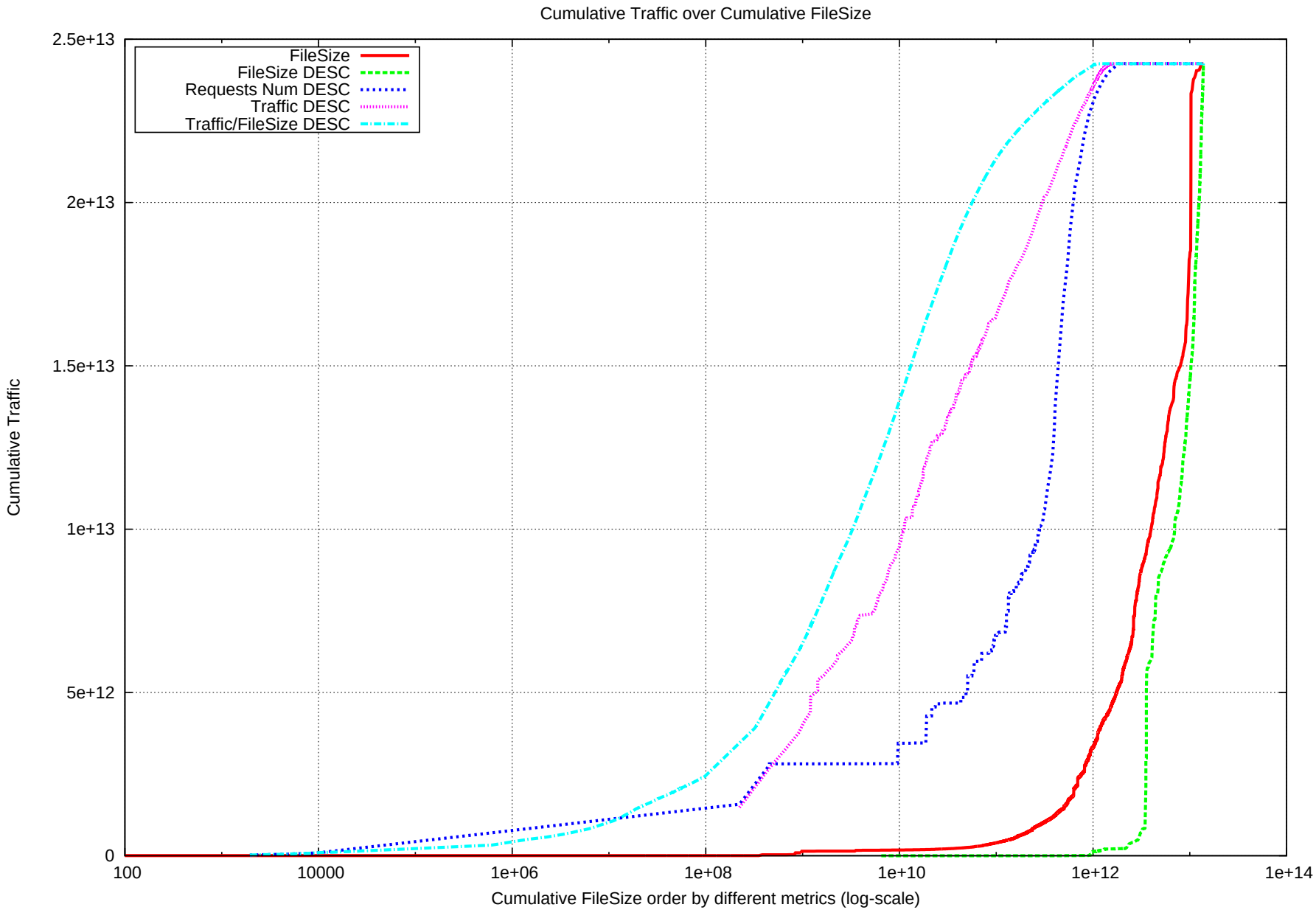
Cache 5: Cumu. Traffic order by Traffic/FileSize



Comparison of the Previous five 'Caching Policies'



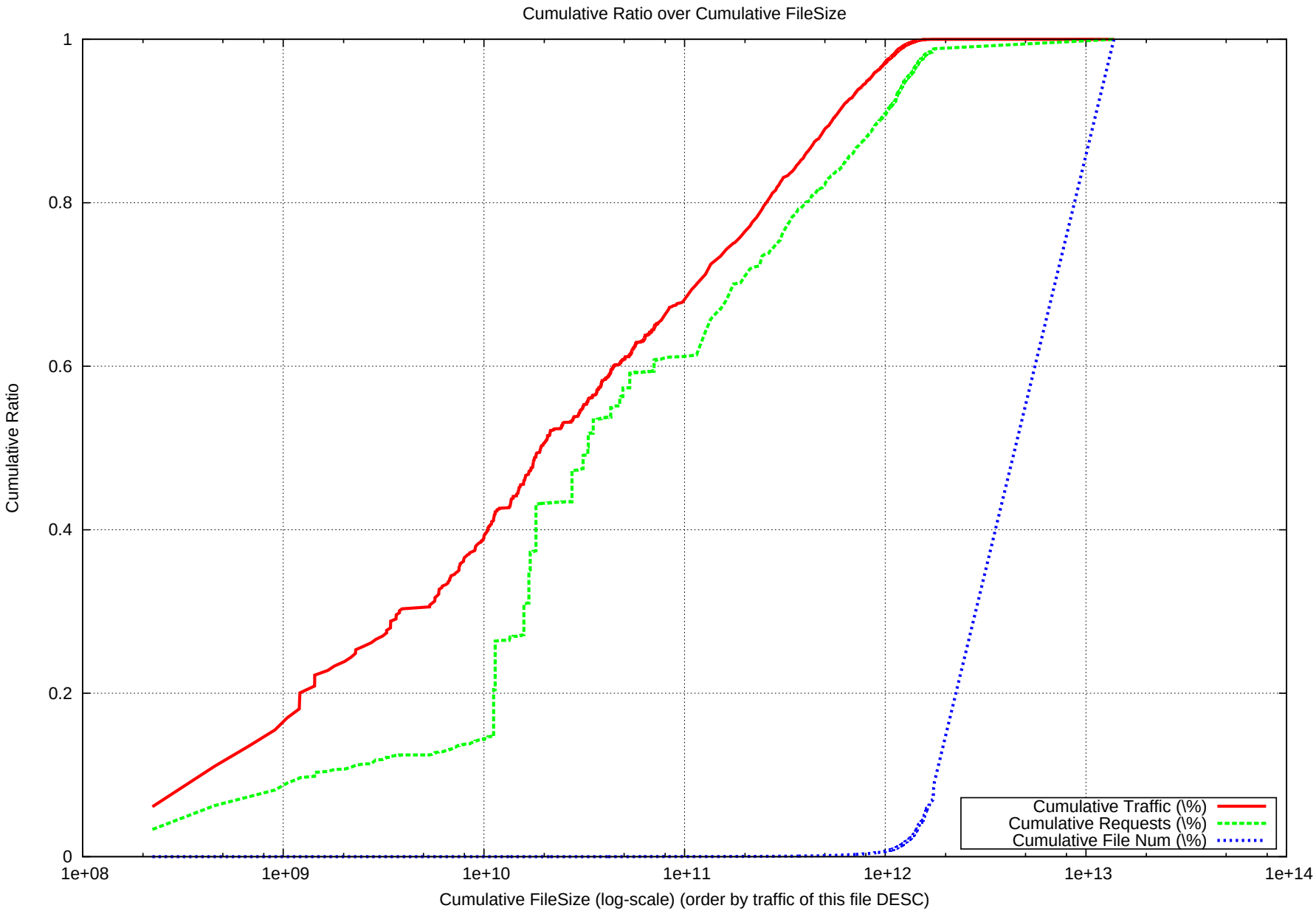
Comparison of 'Caching Policies' (log-scale)



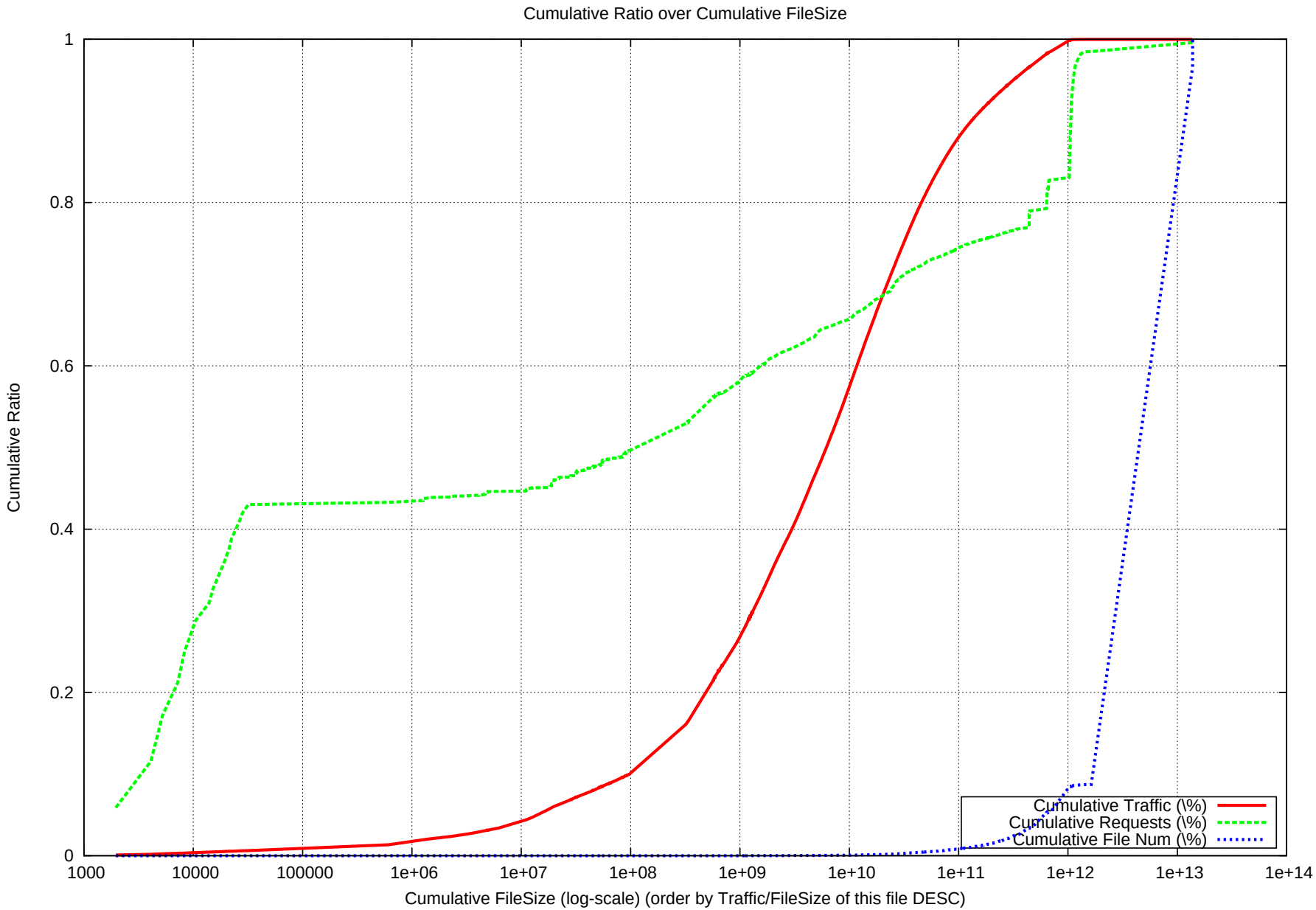
Comparison of 'Caching Policies'

- Among these static caching policies, caching files with most traffic or requests is acceptable.
- Caching files that carried most traffic in history has a good performance. A 10GB cache of 85 files can cover 40% of the total traffic. If cache size continue to increase, the cache efficiency will deteriorate, since x axis of the graph is in log-scale.
- Caching files with largest traffic/filesize ratio shows best performance (by definition). A 10GB cache of 7162 files can cover 58% of the total traffic.

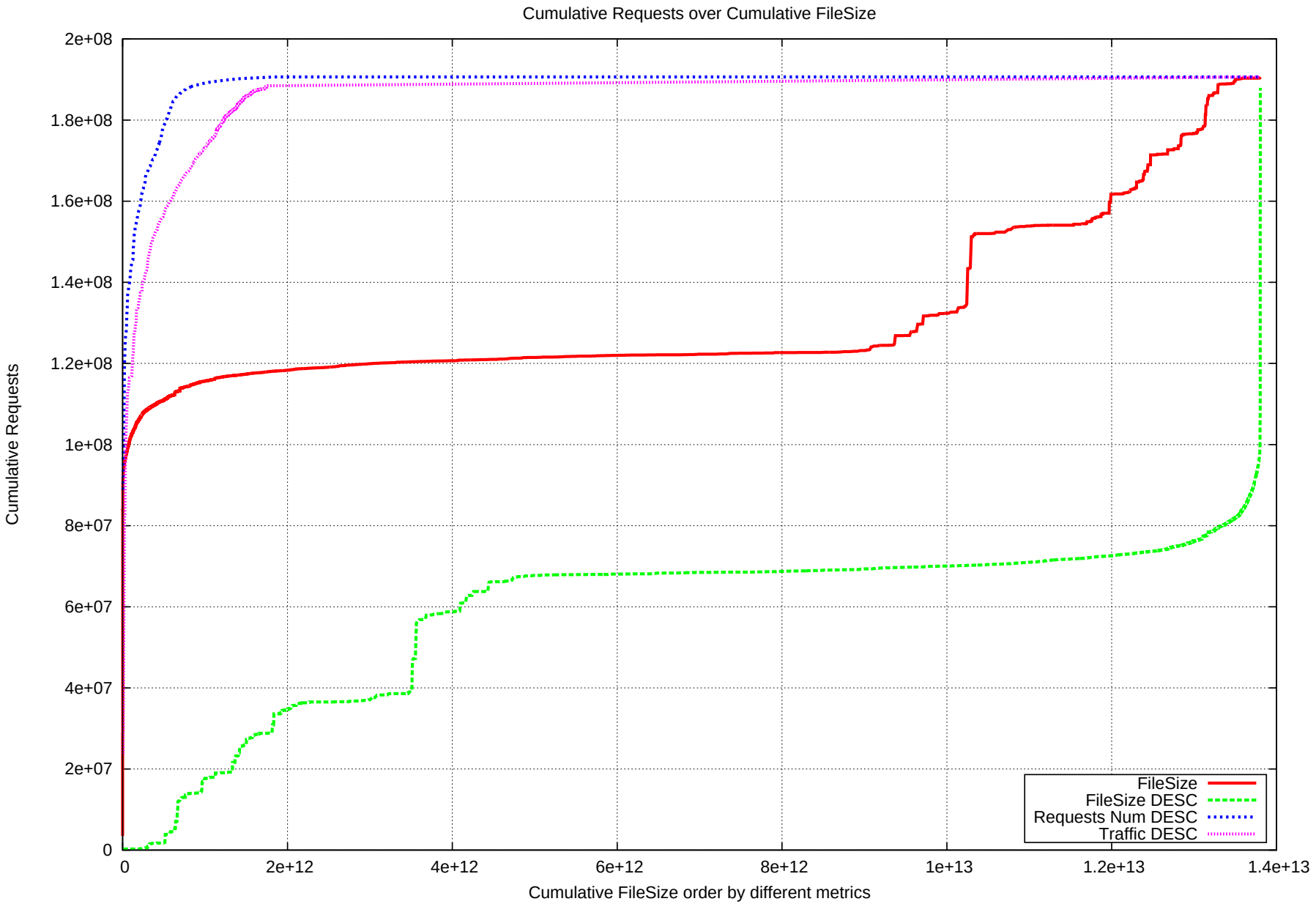
Details of Most Traffic Caching (log-scale)



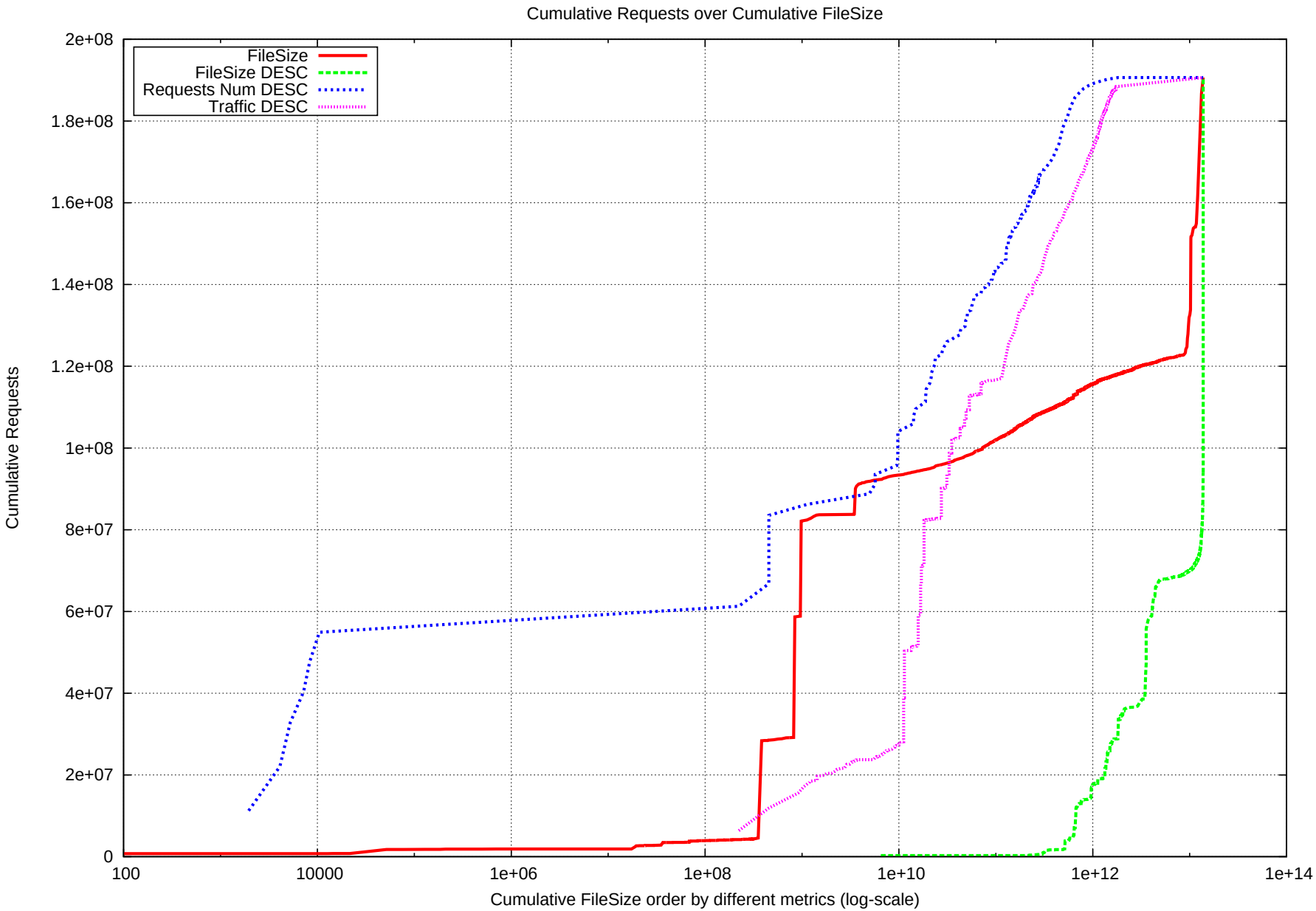
Details of Traffic/FileSize Caching (log-scale)



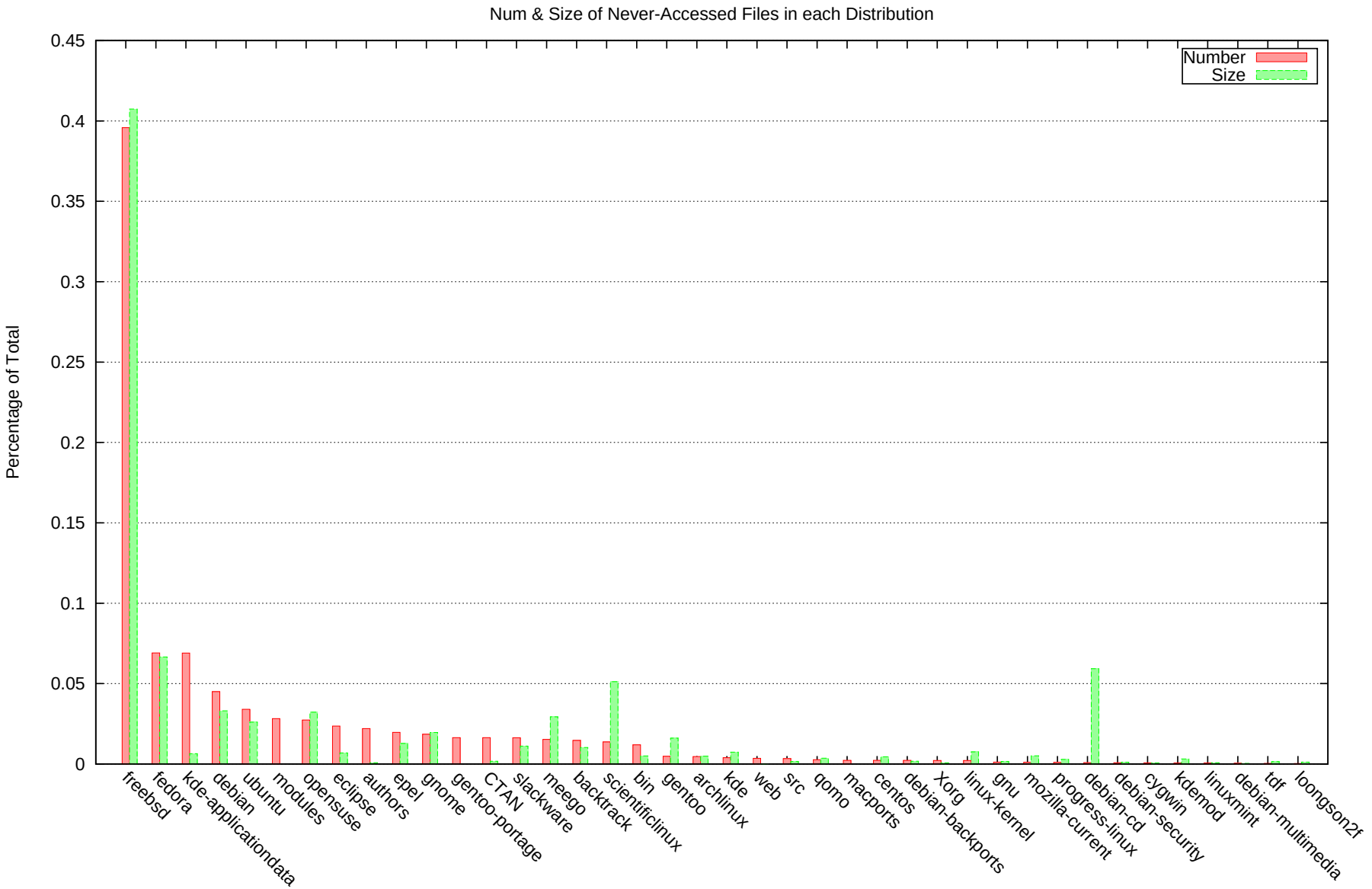
An Alternate Metric: Request Hits



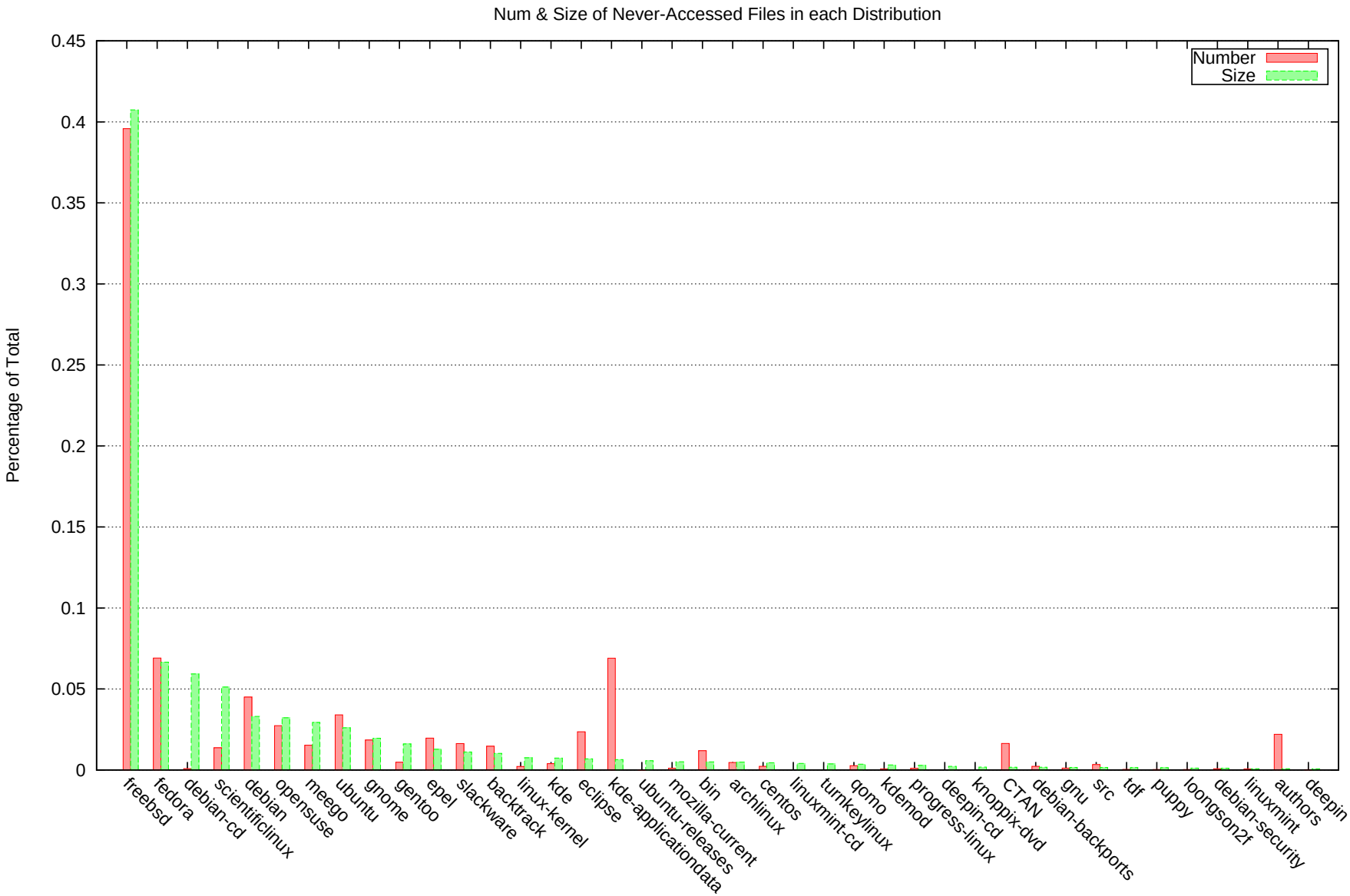
An Alternate Metric: Request Hits



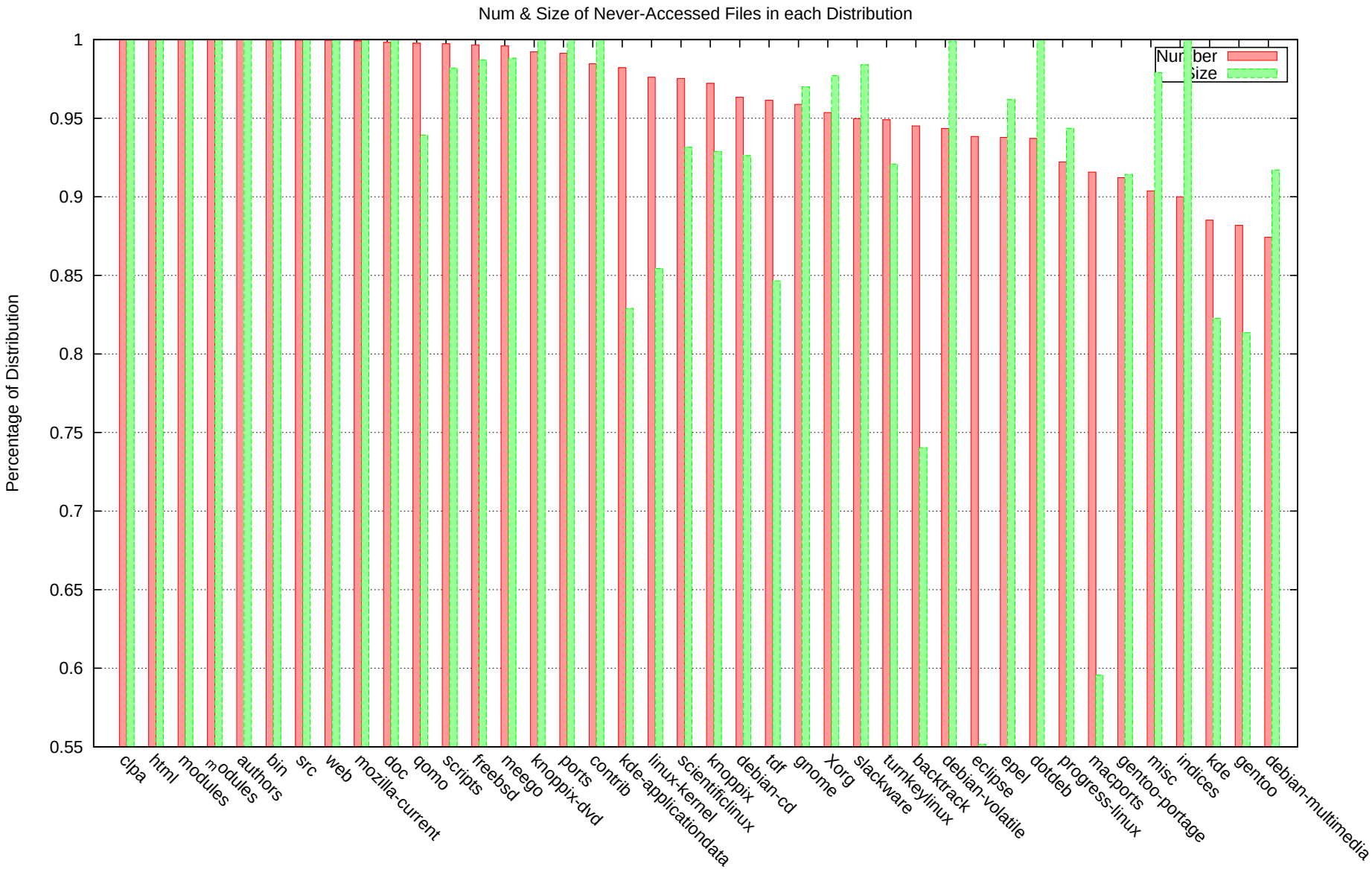
Num & Size of Never-accessed Files (% of total)



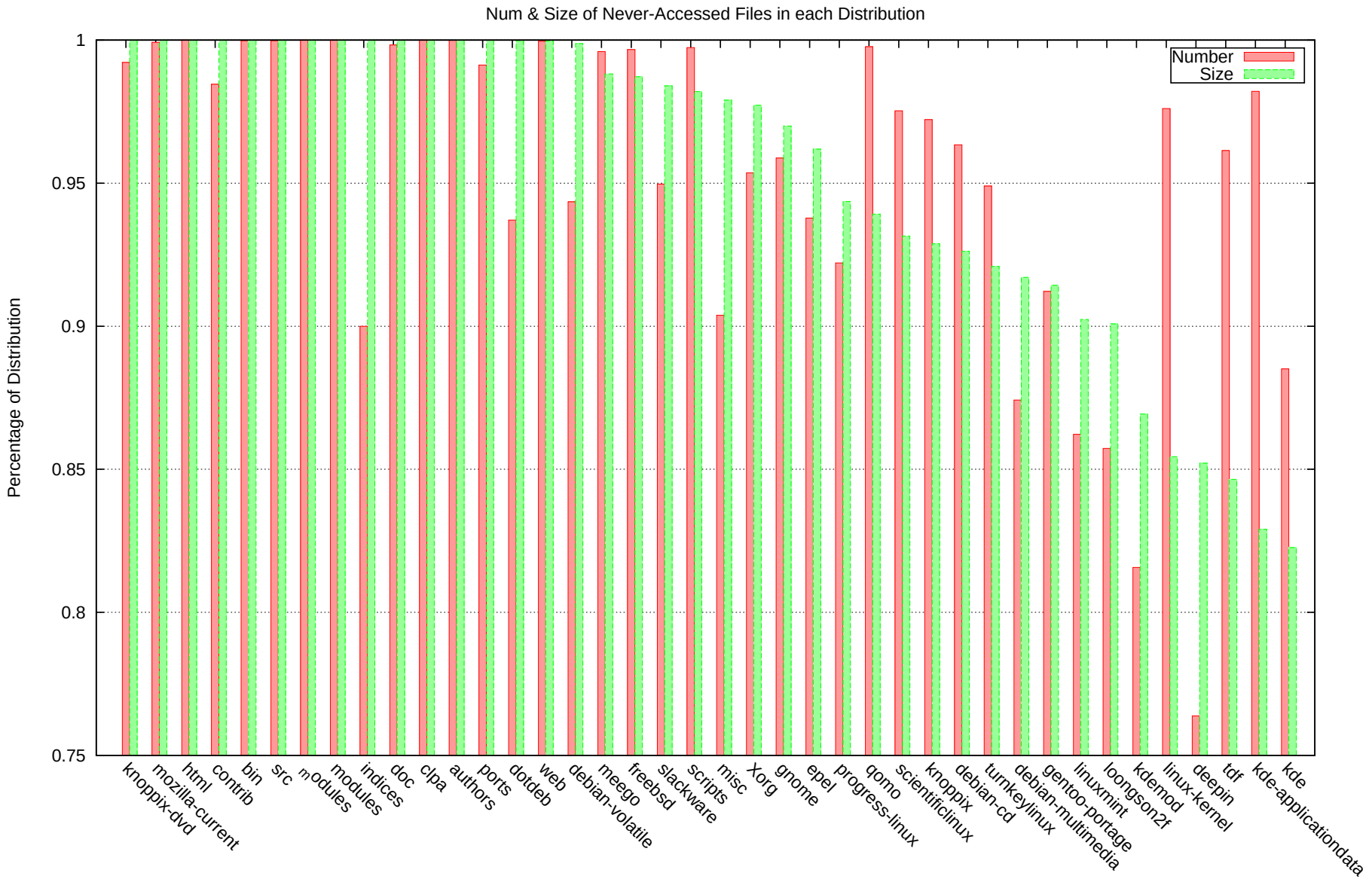
Num & Size of Never-accessed Files (% of total)



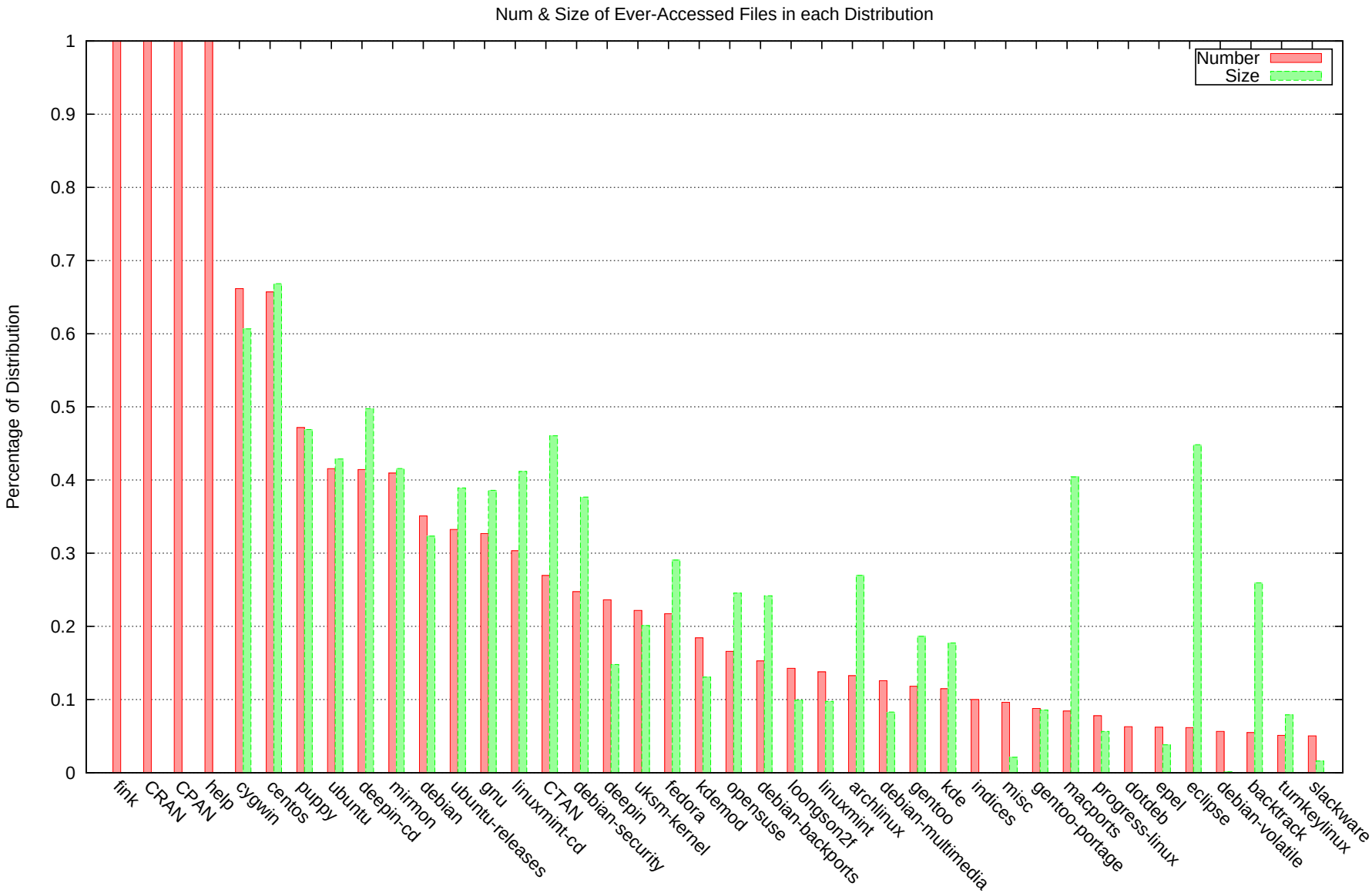
Num & Size of Never-accessed Files (% of distribution)



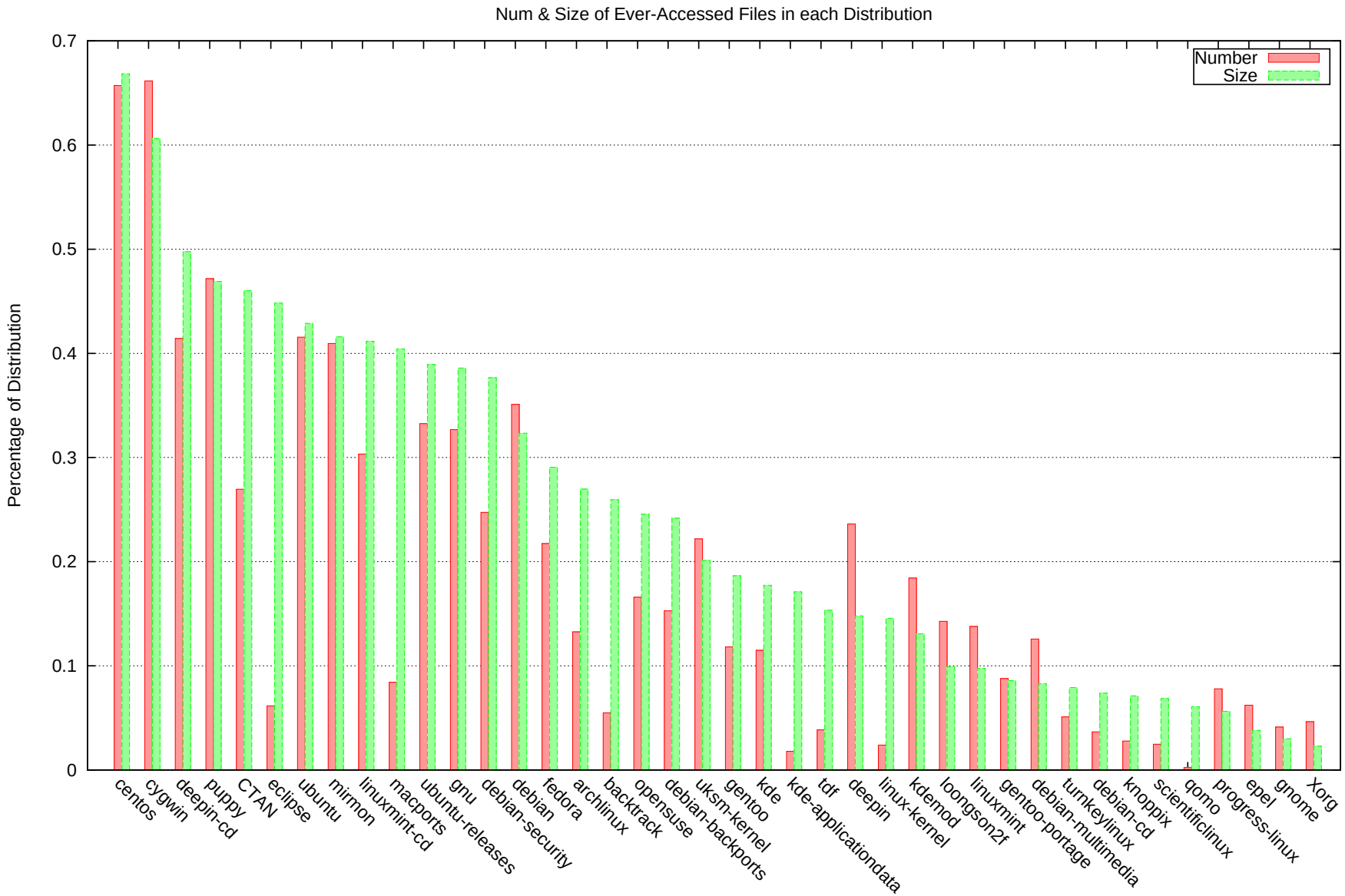
Num & Size of Never-accessed Files (% of distribution)



Num & Size of Ever-accessed Files (% of distribution)



Num & Size of Ever-accessed Files (% of distribution)



Outline

Discovering Sessions

Definition

Two requests are within a same *Interval Session* iff:

- Have same IP address
- The time difference does not exceed some limit

Definition

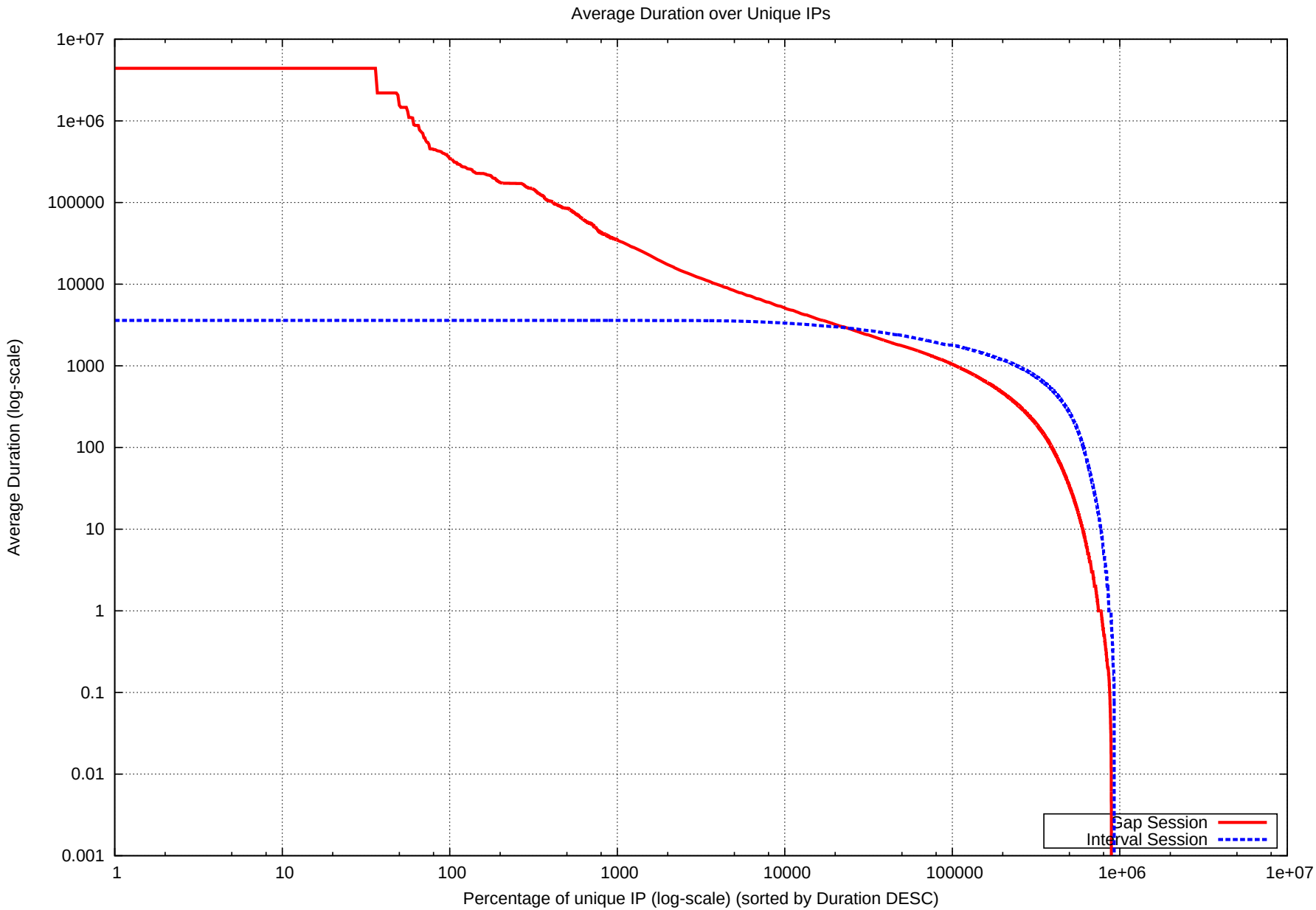
A *Gap Session* is a longest sequence of requests where:

- All requests are from the same IP address
- Time difference of every two adjacent requests do not exceed some limit

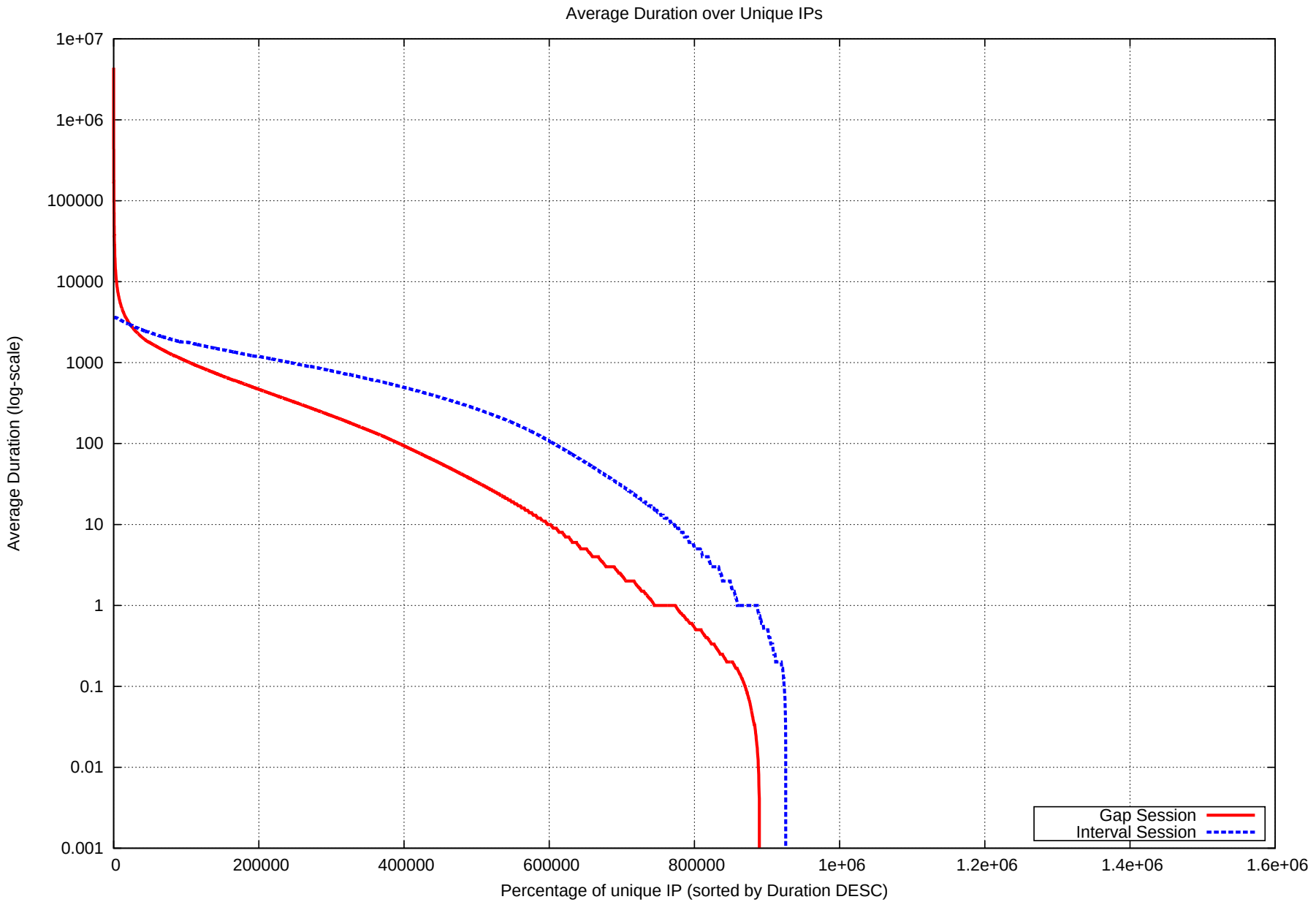
Discovering Sessions (continued)

- Since Mirrors is a resource-downloading site, many sessions download many files for hours. The time limit of Interval Session is 60 minutes.
- The time limit of Gap Session is 30 minutes. For long downloads that extend over 30 minutes, the session will be broken into two.
- Both algorithms suffer from false positives and true negatives.
- Some IPs access so frequently that the gap session never ends (see the next slide), making the Gap Session data highly biased.

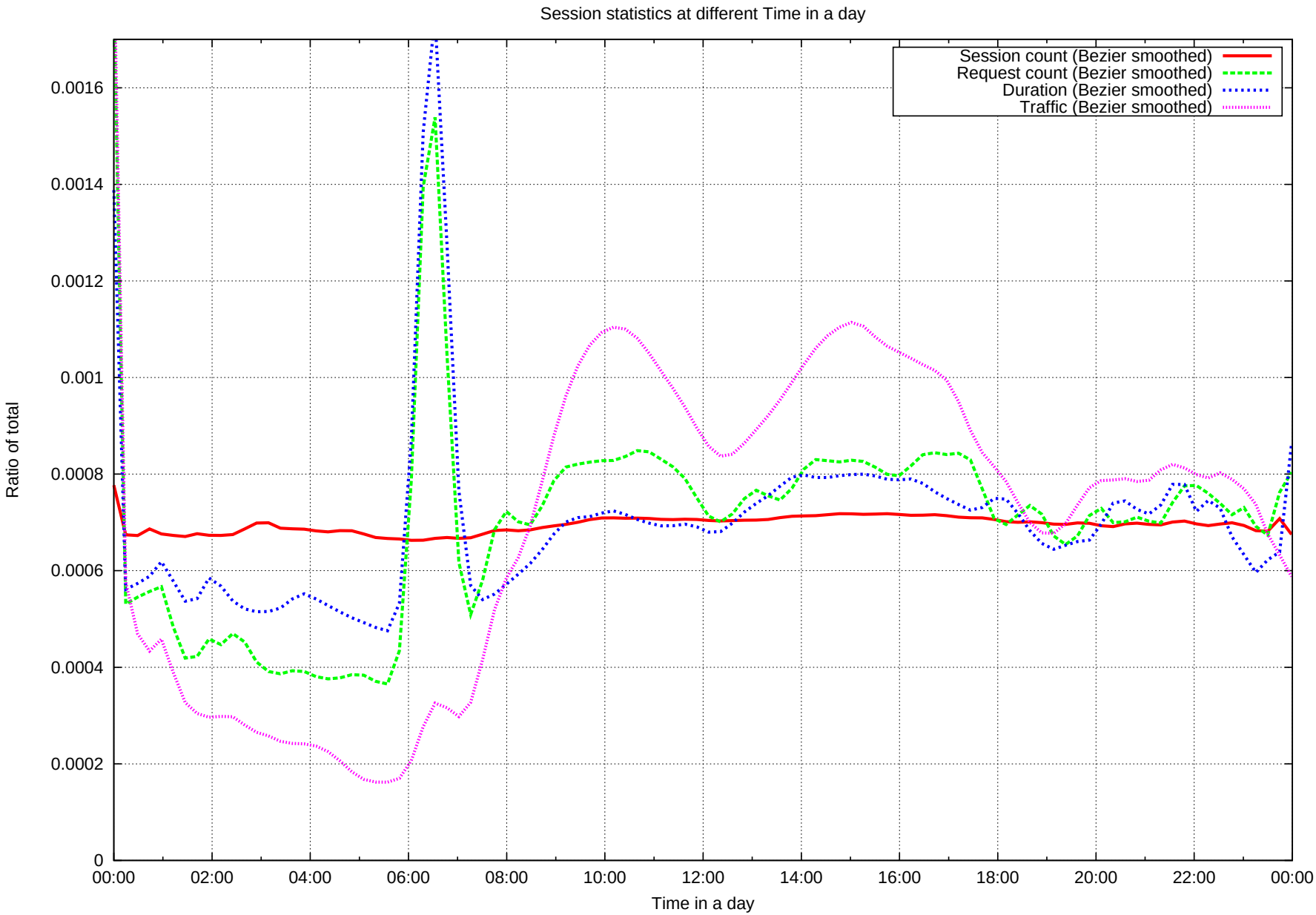
Average Session Duration over IP (log-scale)



Average Session Duration over IP (normal-scale)



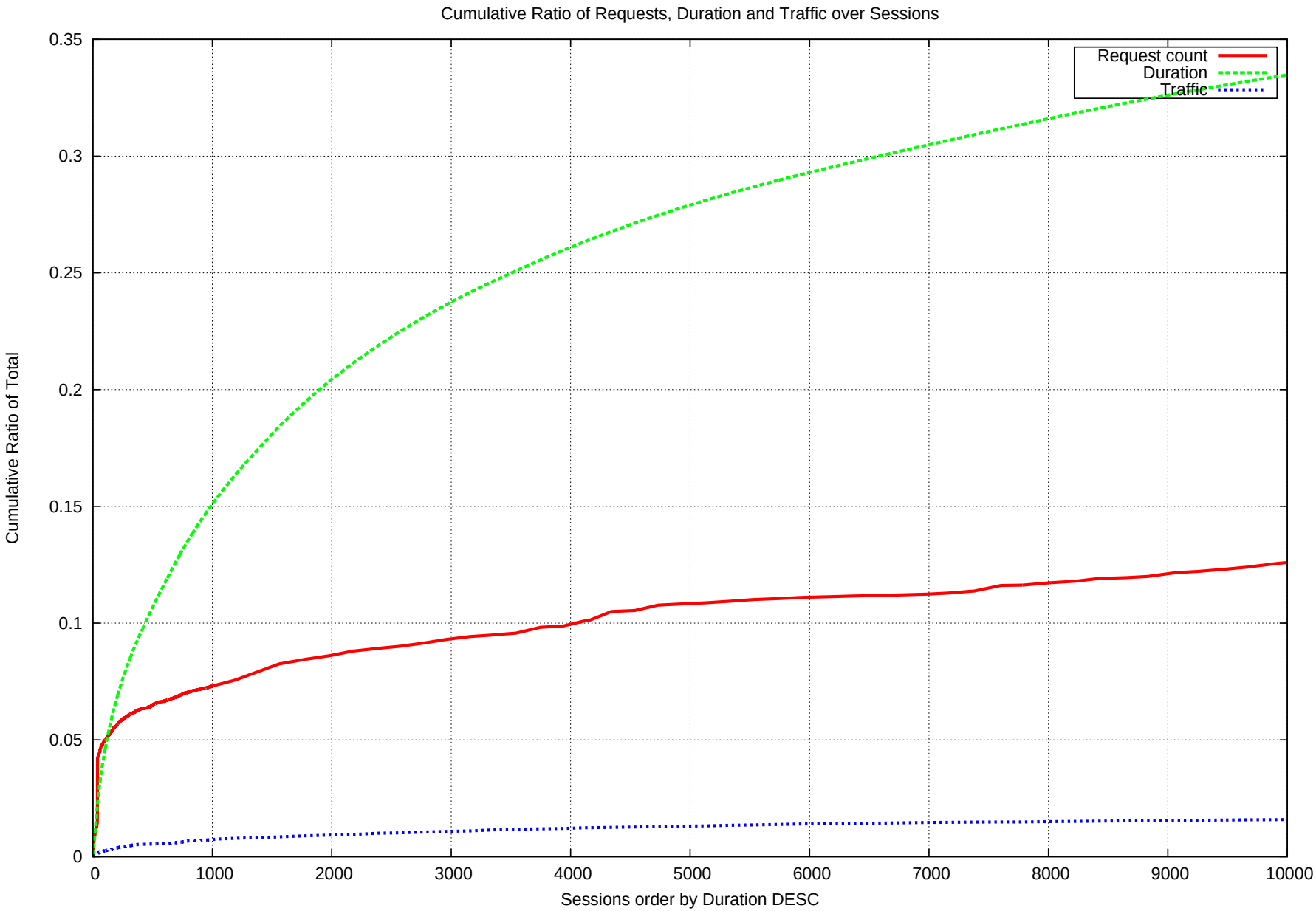
Gap Session Statistics in a day



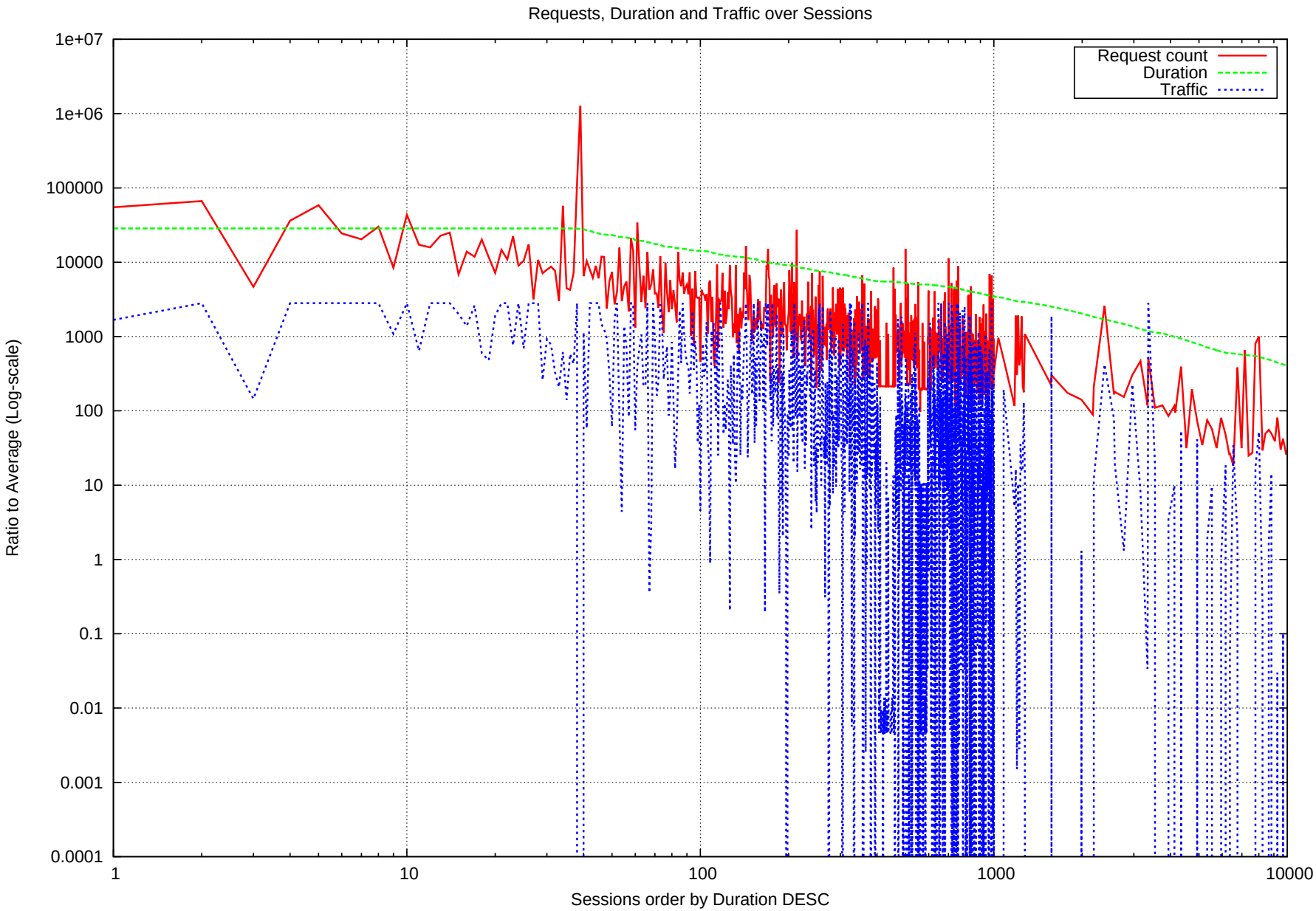
Explanation

- How strange the three graphs look! Durations are always integers (seconds), so there are straight lines in duration graphs.
- Many sessions are actually 'single' requests, so their duration is zero, and the amount of them can be seen in normal-scale graph.
- About 40 sessions extend throughout the whole 51 days, and more sessions extend less days. The amount of them is the length of red horizontal line in log-scale graph.
- Because the log starts at May 22 06:25:37, requests in these long-live Gap Sessions accumulate to a horrible peak at 06:27 in Gap Session statistics (much higher than shown, since the original data is noizy, and the graph is Bezier-smoothed).

Cumulative Over Long-live Gap Sessions



Ratio to Average for Long-live Gap Sessions (log-scale)



Statistics of Interval Sessions

	Total	Average	Max	Min	Std. Dev
Requests	328976877	7.5020	186043	1	173.1824
Traffic	36277 GB	867.46 KB	Overflow	0	15.227 MB
Duration	$4.026 * 10^{10}$	918.13 s	3599 s	-6 s	1502.9 s

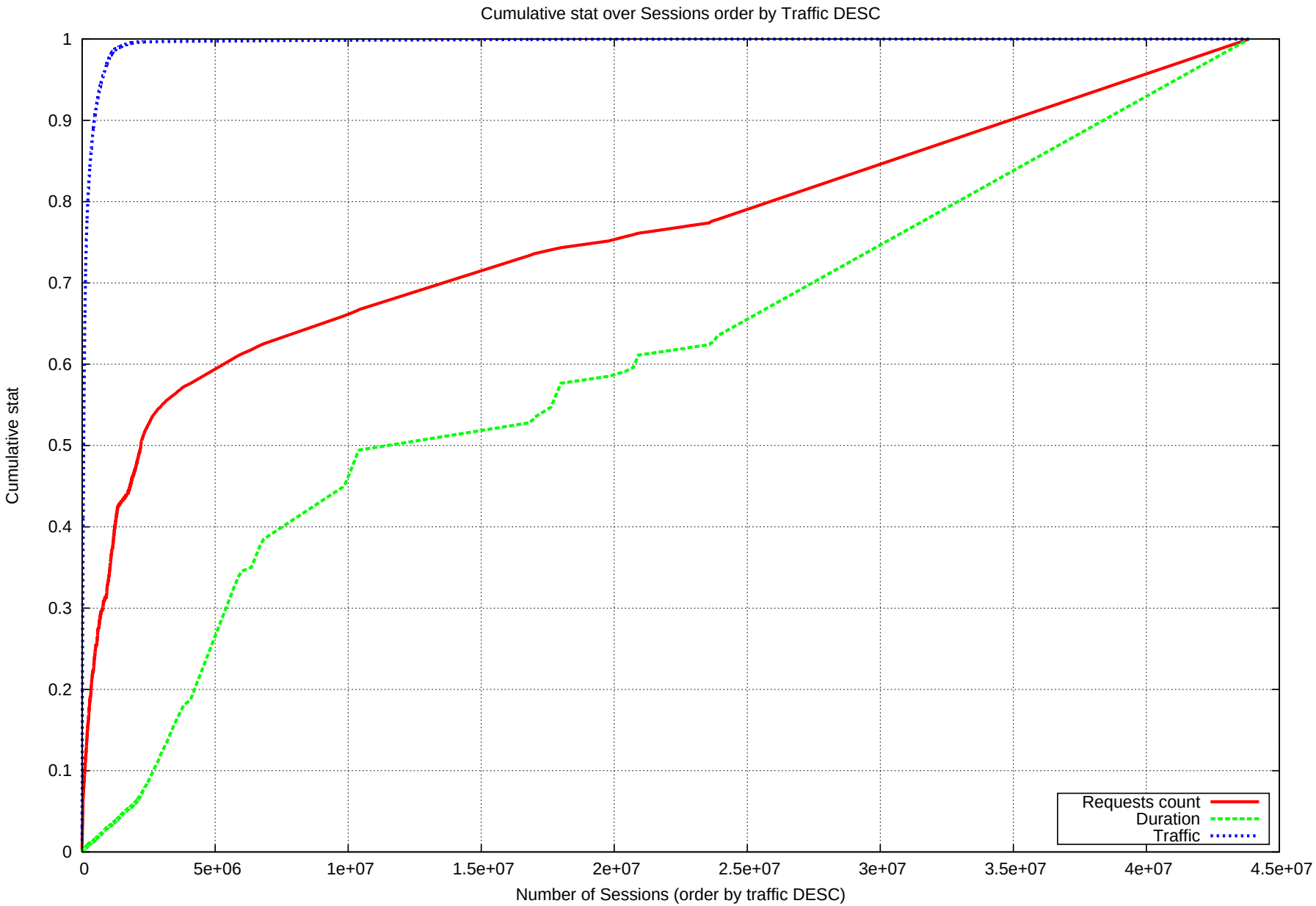
- 43852053 sessions in total.
- Sessions with negative duration is because log items can accidentally be not in time non-decreasing order. I'm not going to fix it, since these 230 wrong sessions have little influence.

Statistics of Gap Sessions

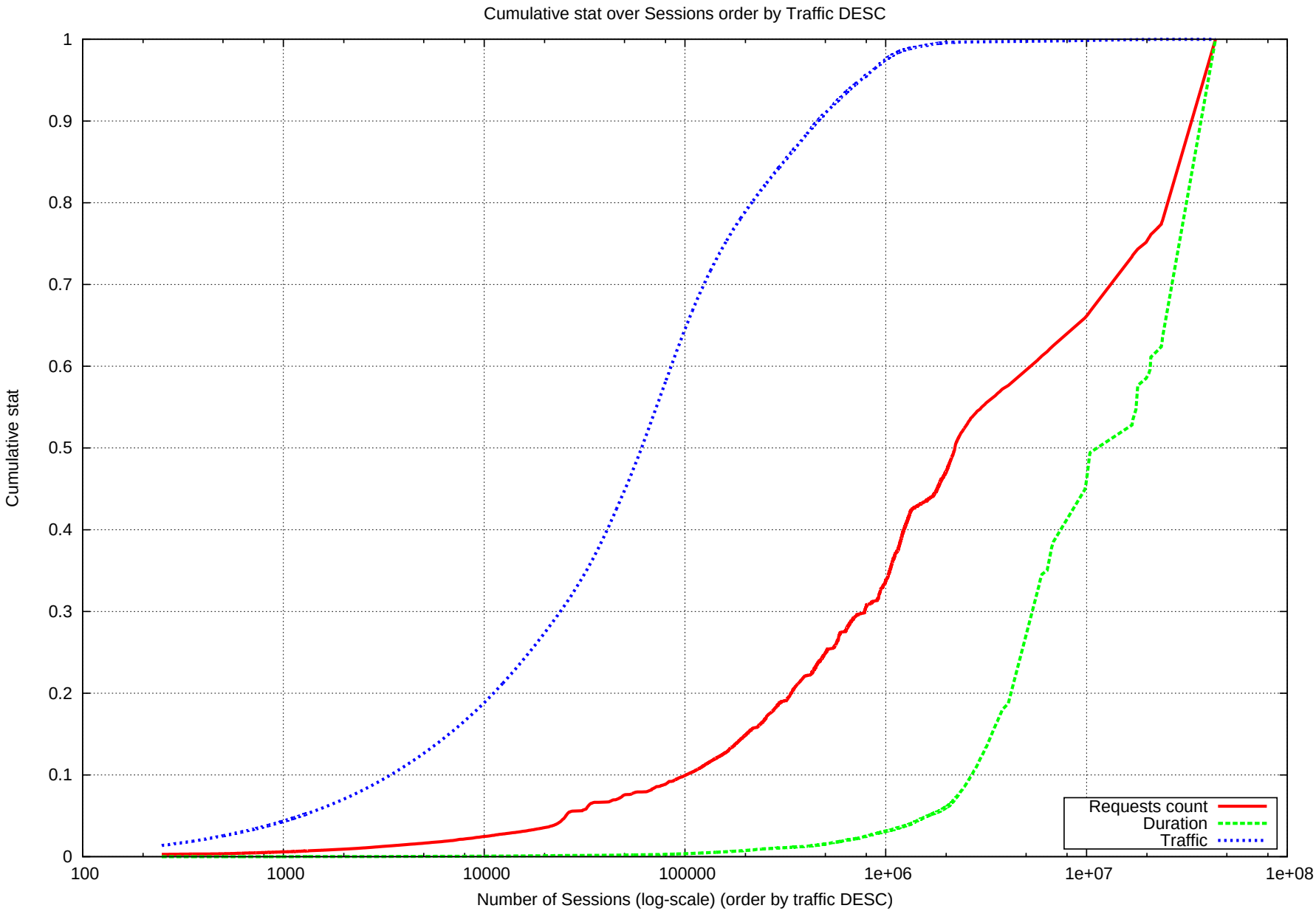
	Total	Average	Max	Min	Std. Dev
Requests	328976877	6.6200	8432330	1	1255.9488
Traffic	35291 GB	744.67 KB	Overflow	0	15.184 MB
Duration	$7.666 * 10^9$	154.27 s	4406403	-27 s	7060.6 s

- 49694582 sessions in total.
- For comparison with Interval Sessions.
- The following statistics are based on Interval Session if not noted specially.

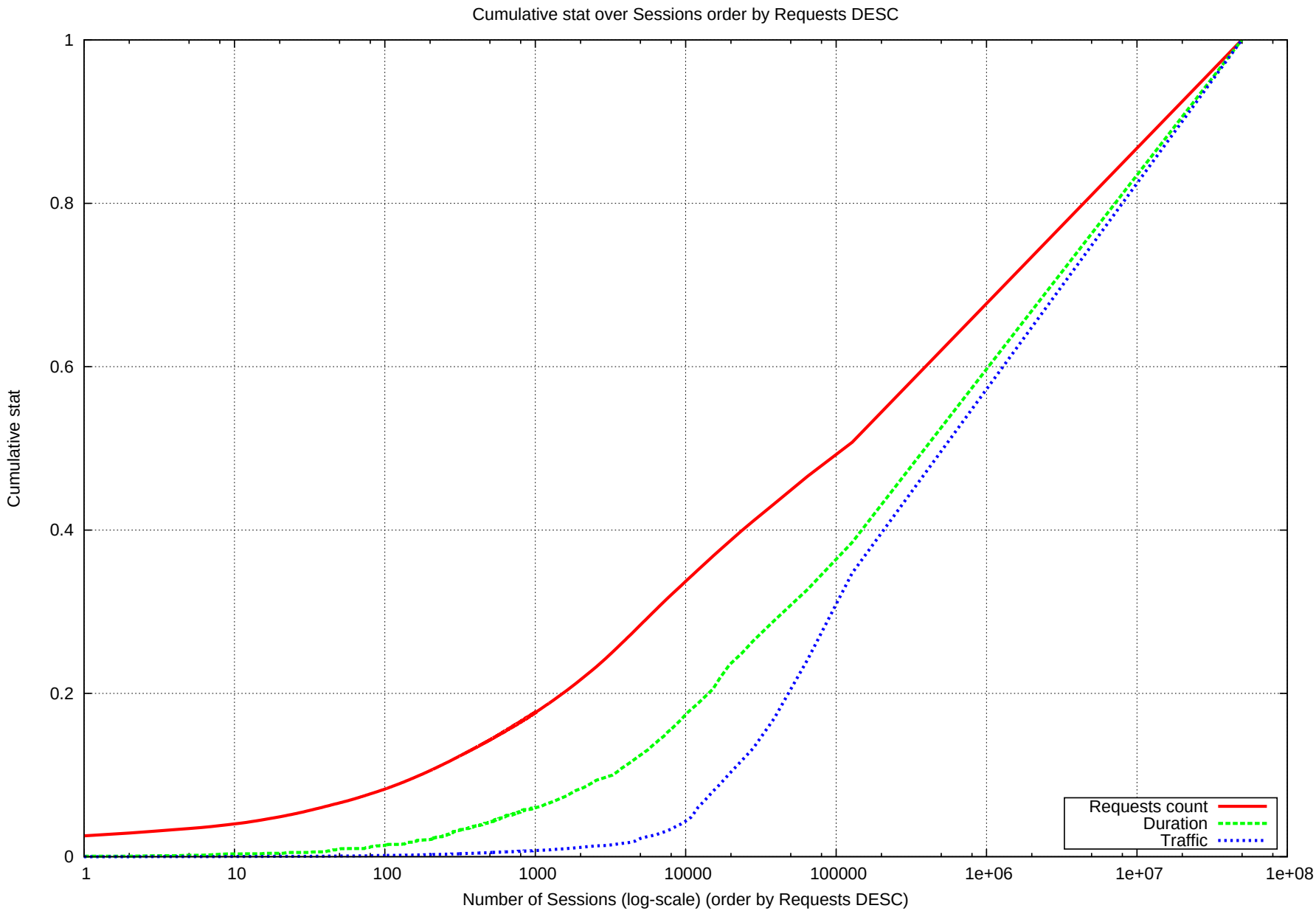
Cumulatives over Sessions order by Traffic DESC



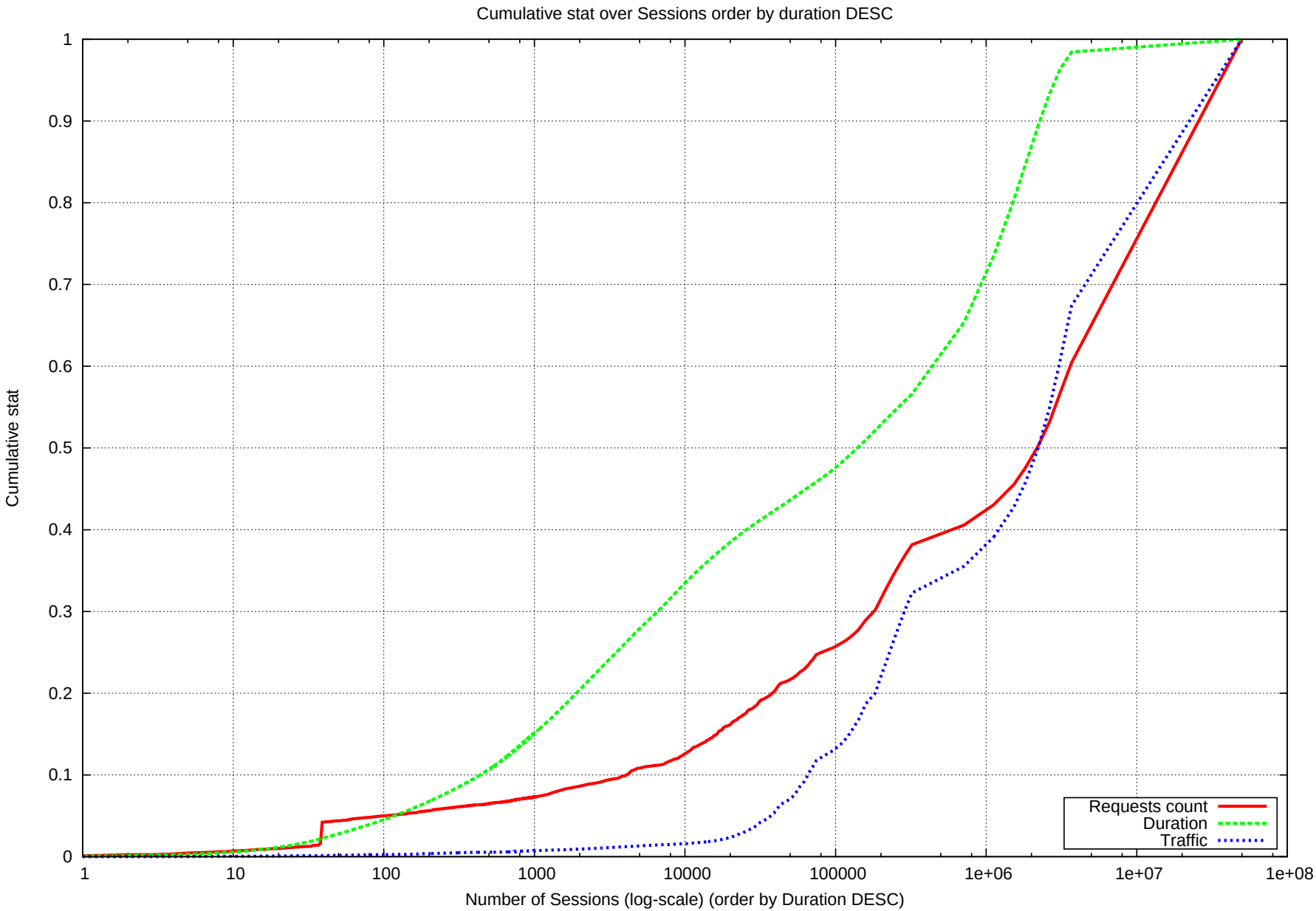
Cumulatives over Sessions order by Traffic DESC



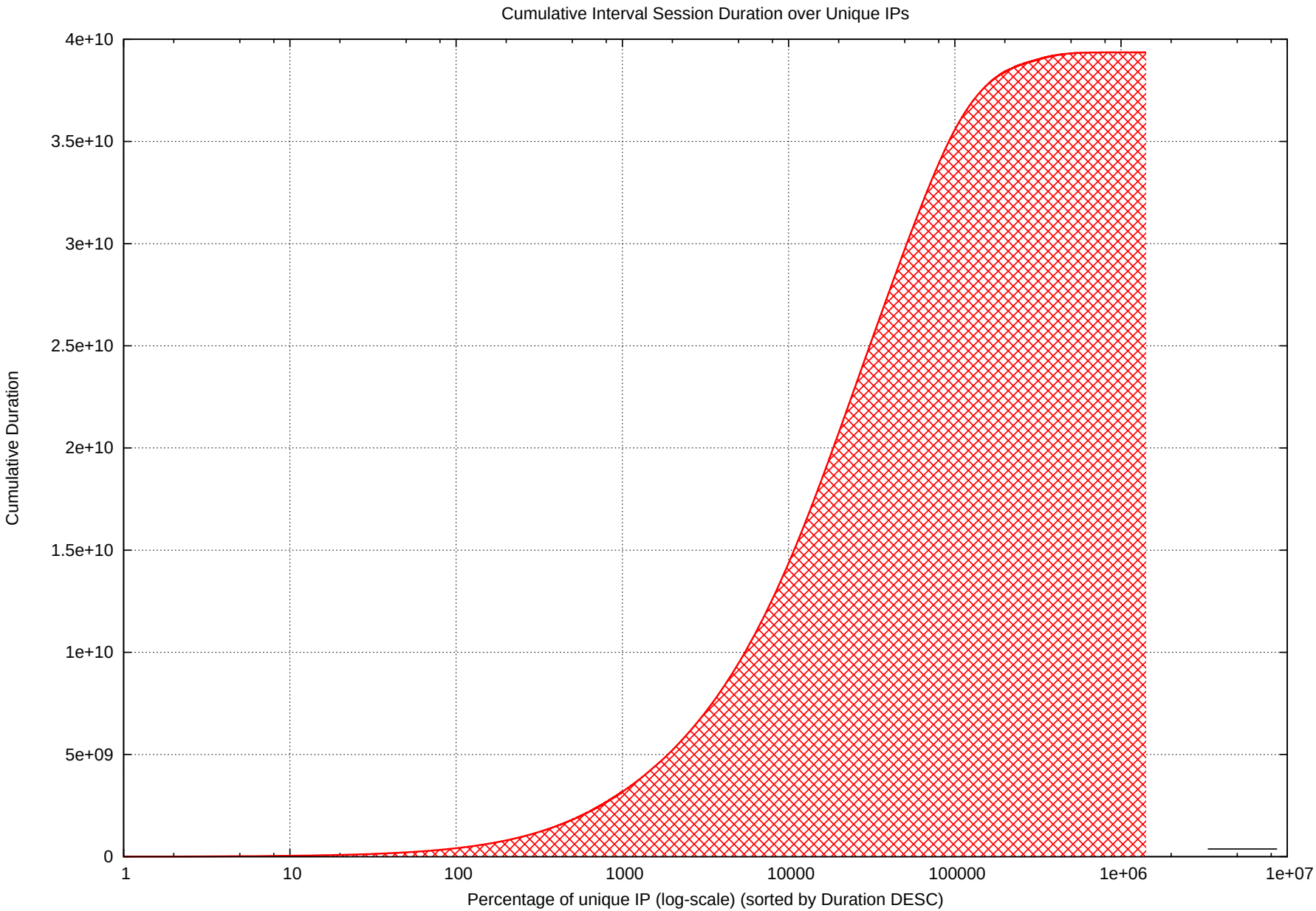
Cumulatives over Sessions order by Requests DESC



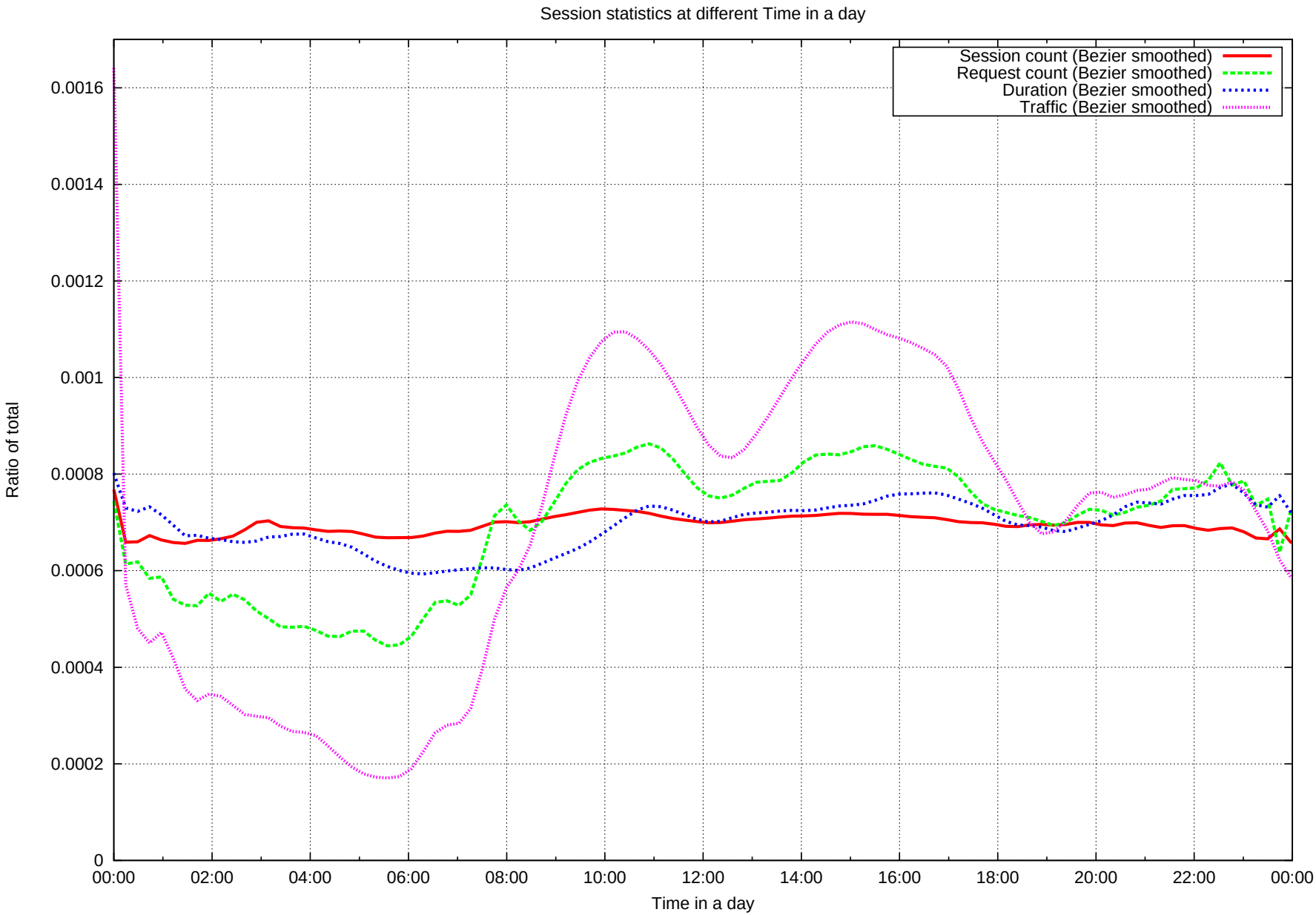
Cumulatives over Sessions order by Duration DESC



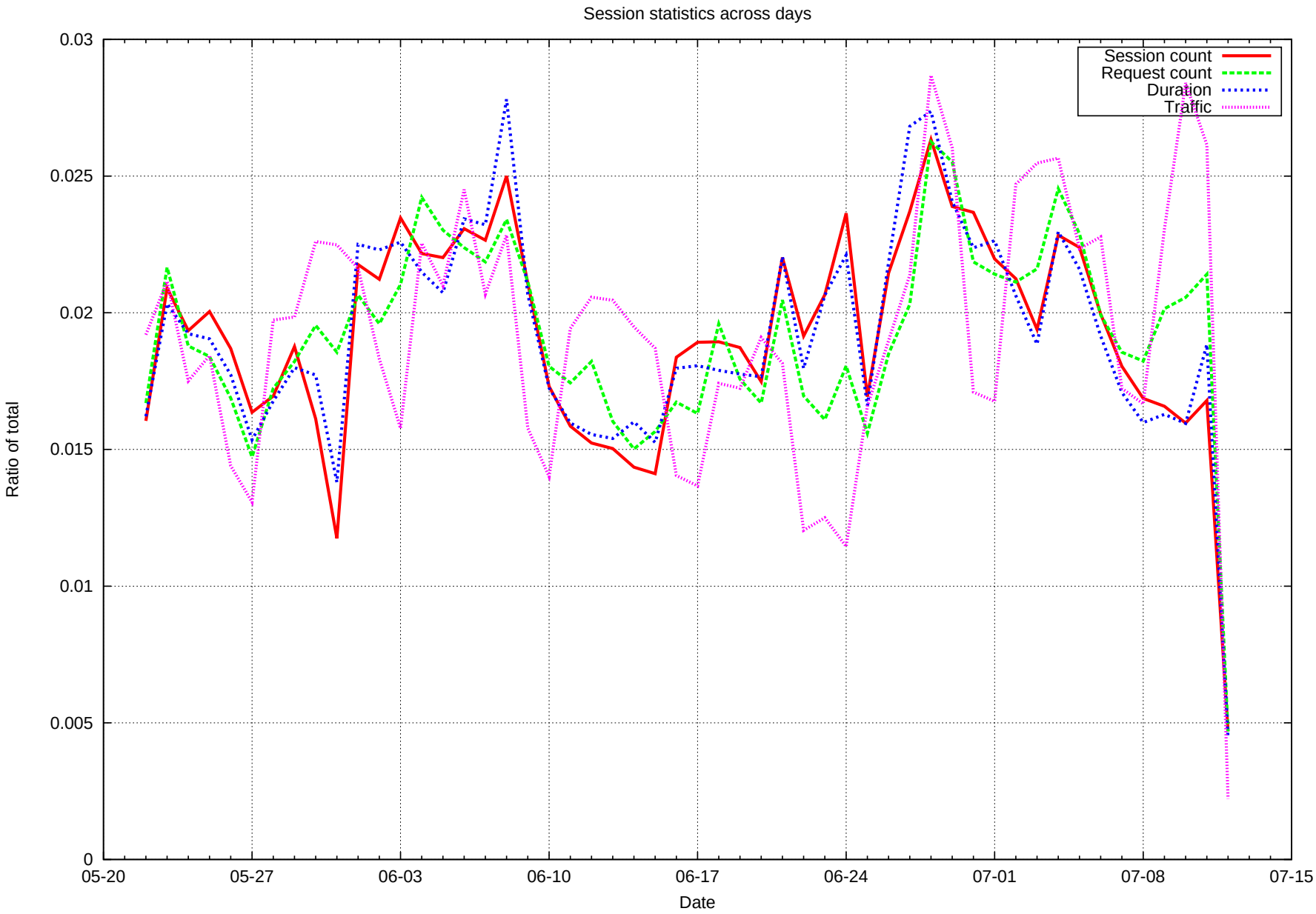
Cumulative Session Duration over Unique IPs



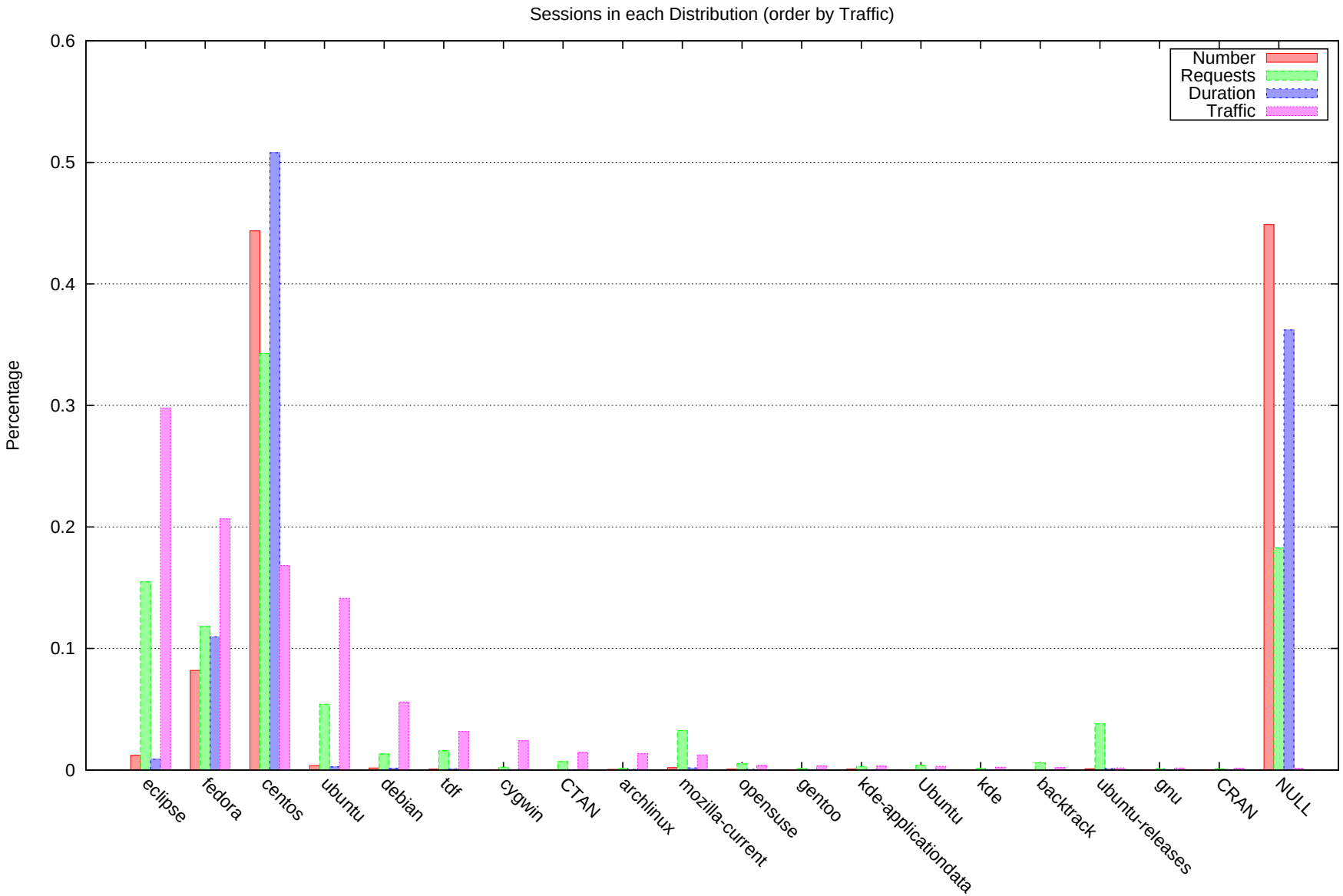
Sessions in a day



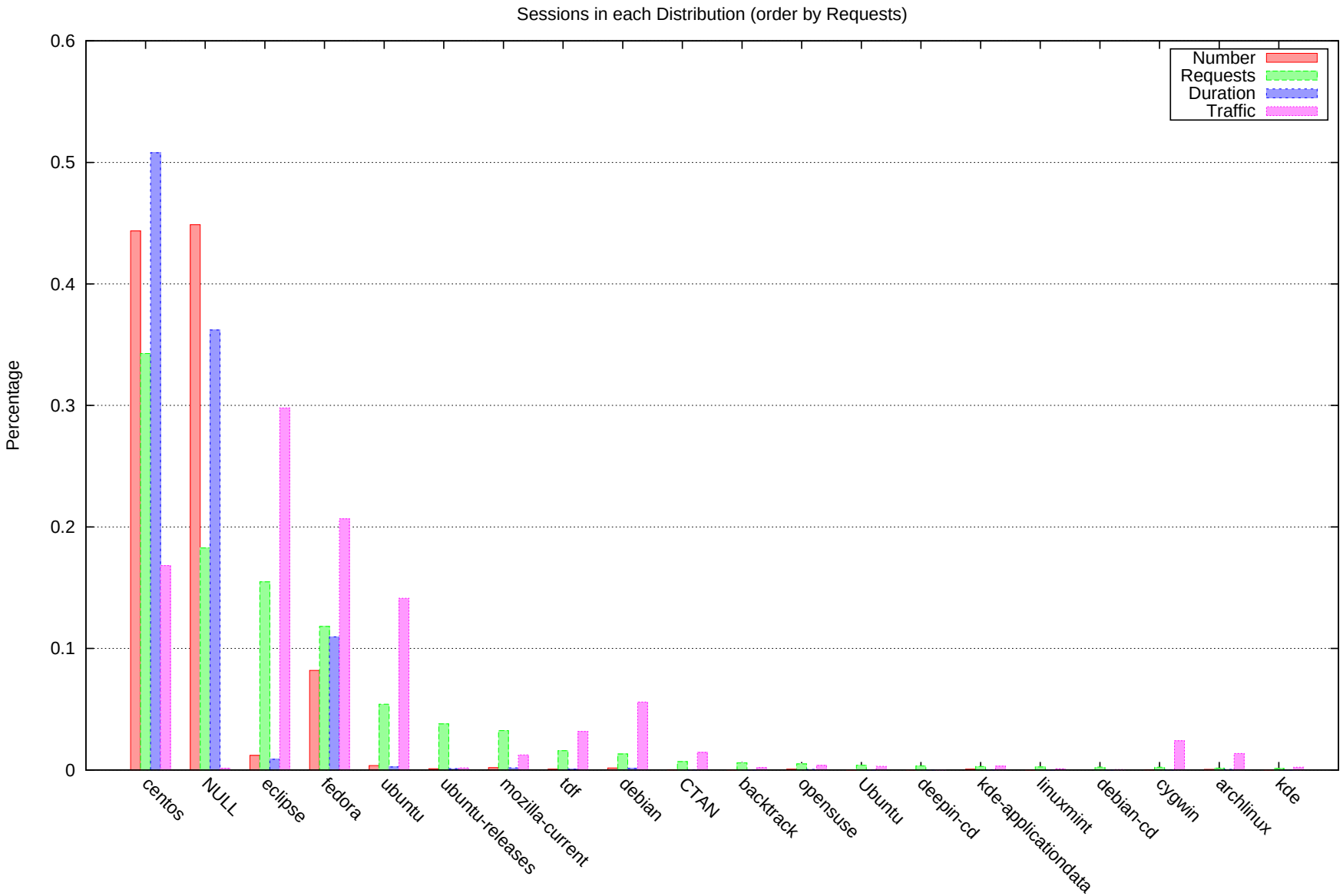
Sessions across days



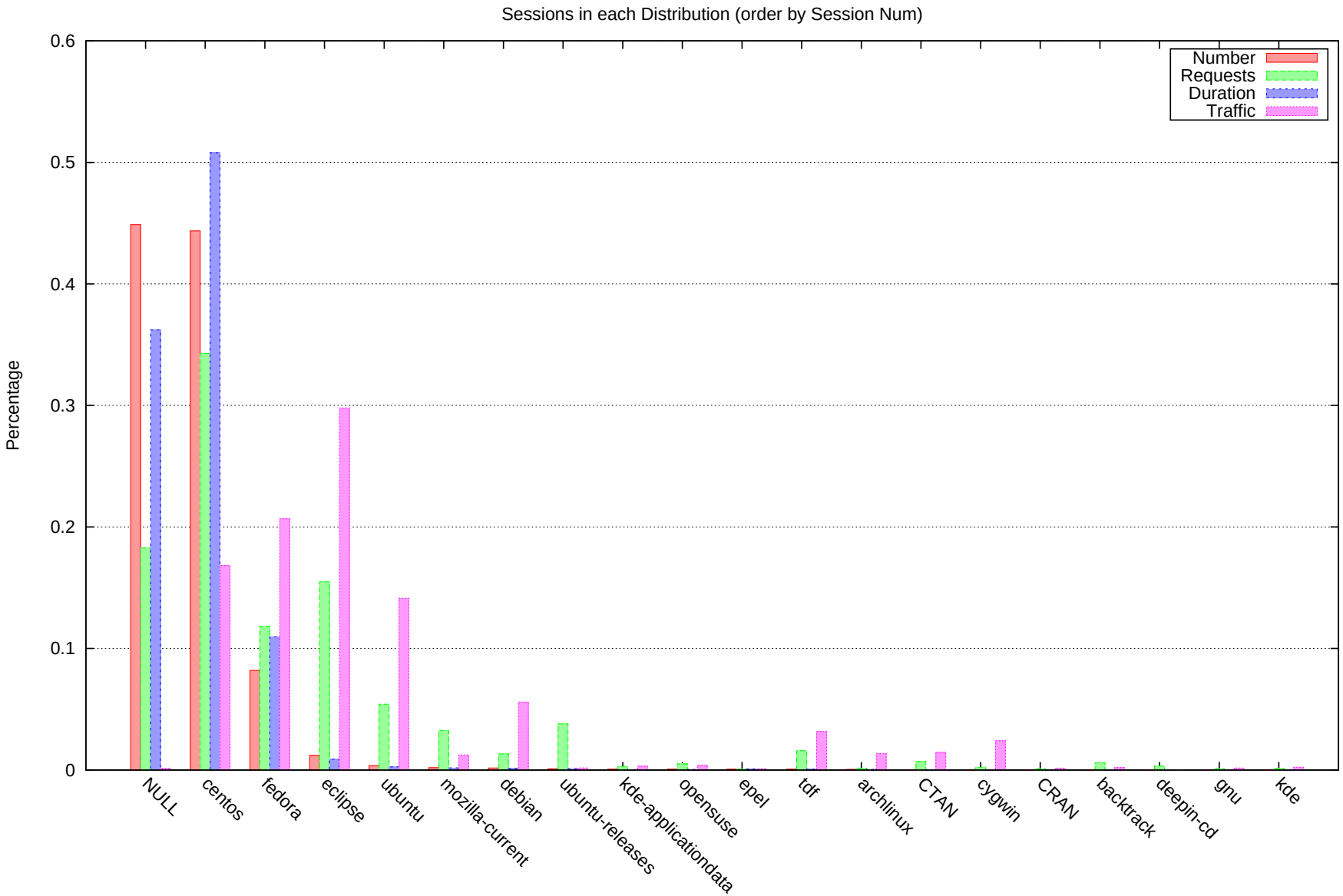
Sessions in Distributions order by Traffic



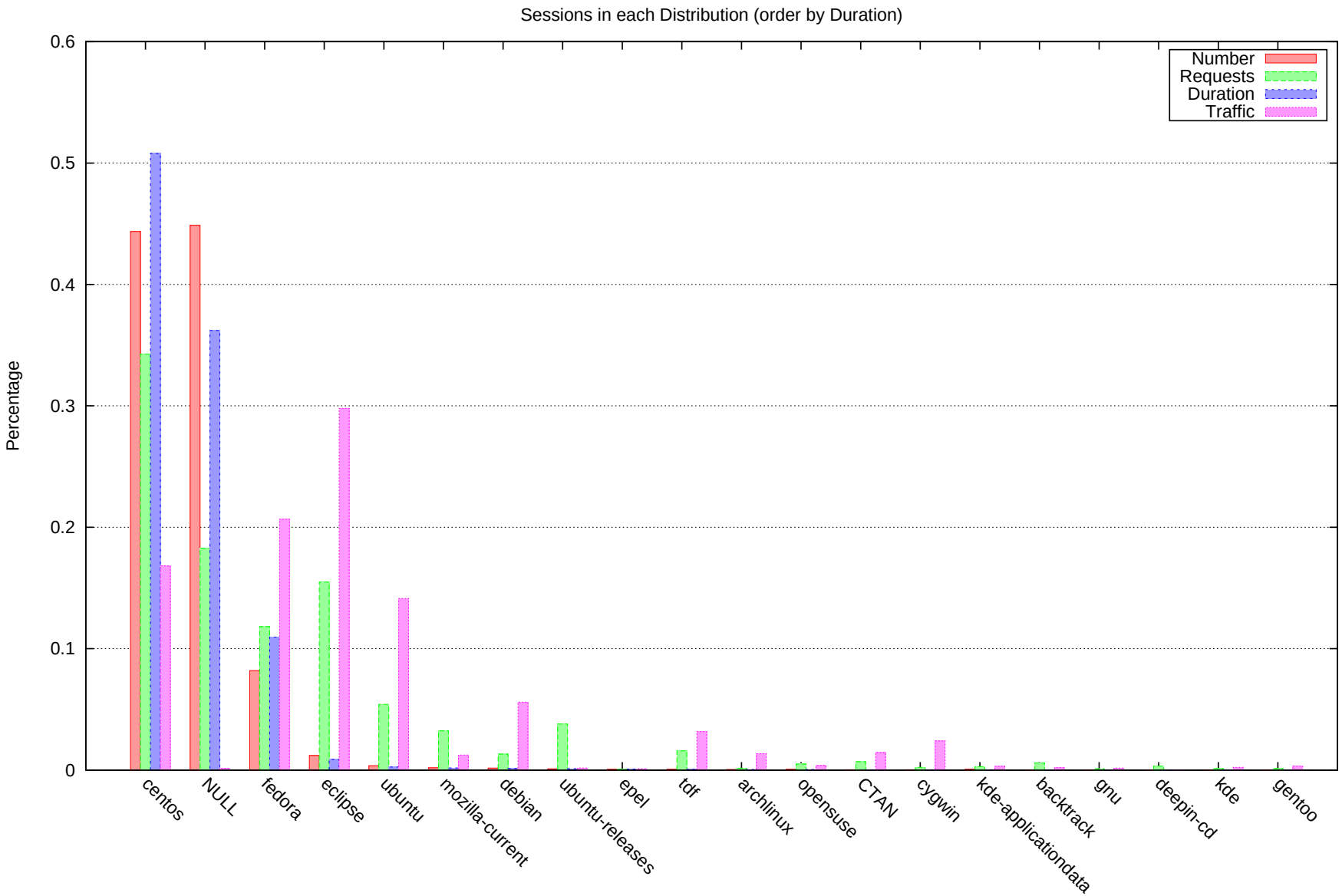
Sessions in Distributions order by Requests



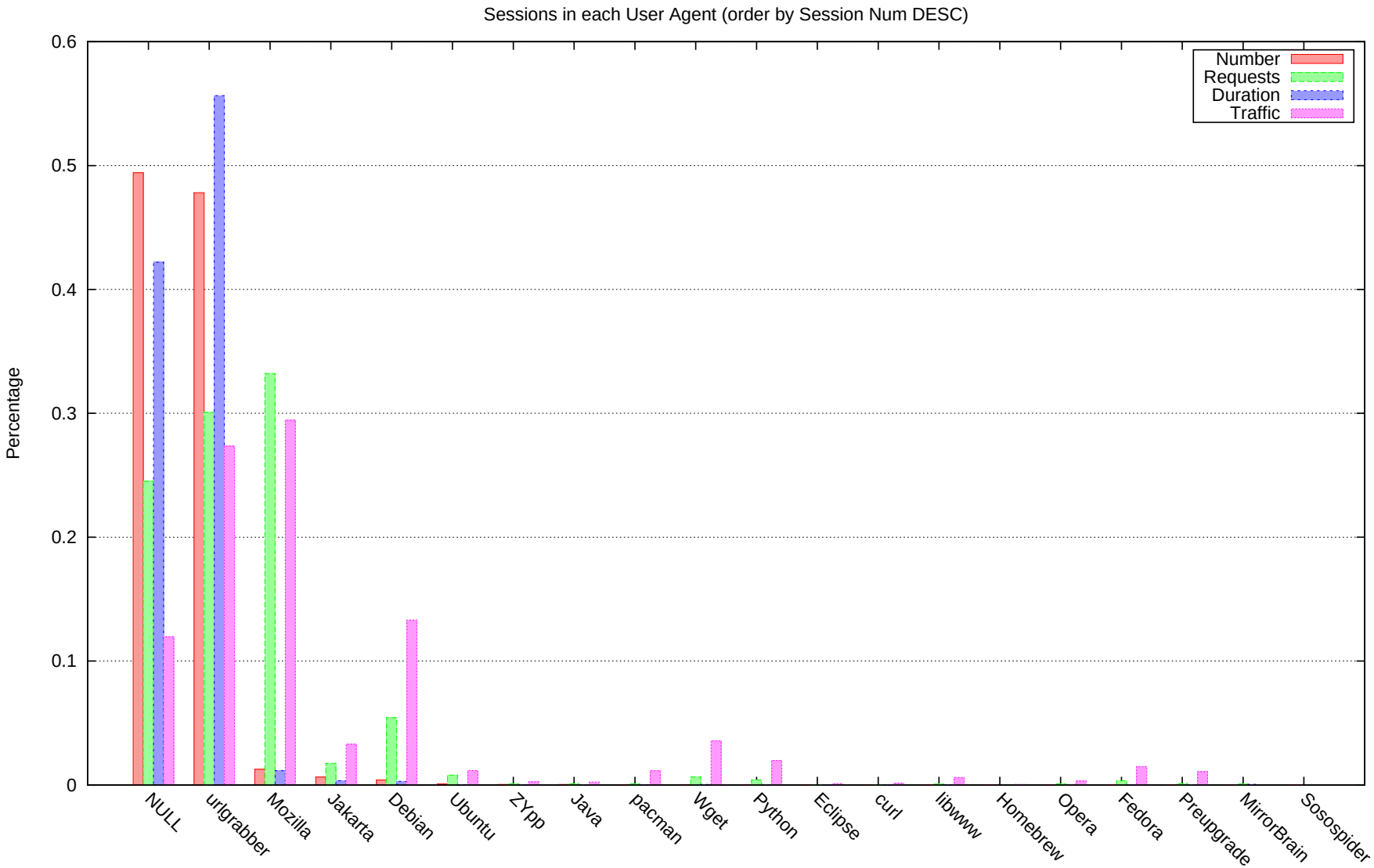
Sessions in Distributions order by Session Num



Sessions in Distributions order by Duration



Sessions and User Agents



Outline

CentOS Basic Stat

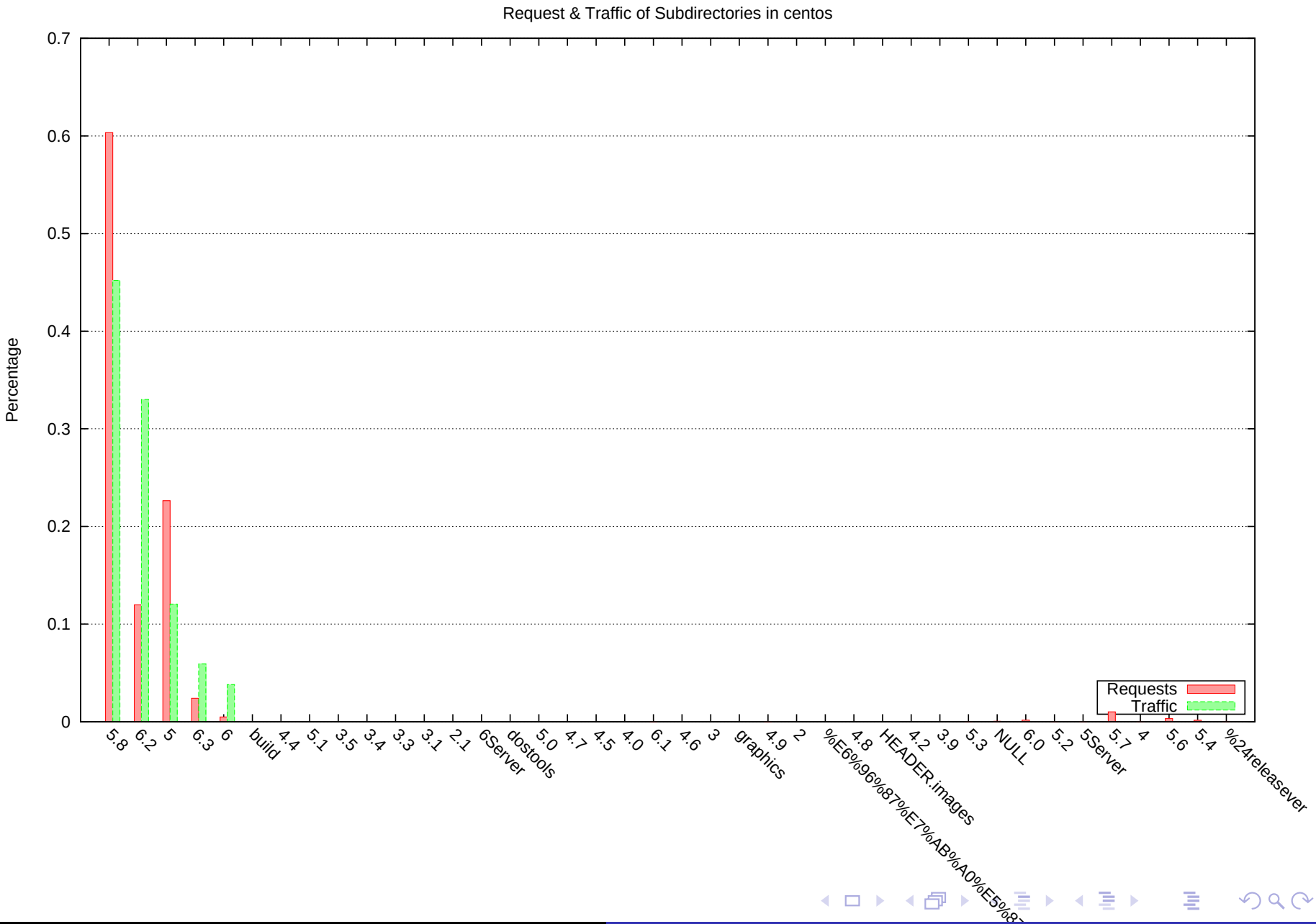
	Value	% of total	Rank
Requests	100986986	30.7%	1
Traffic	5252.5 GB	14.2%	4
Files	70043	0.7%	19
FileSize	172.8 GB	1.3%	12
Sessions	19452260	44.4%	1

CentOS Basic Stat - Average

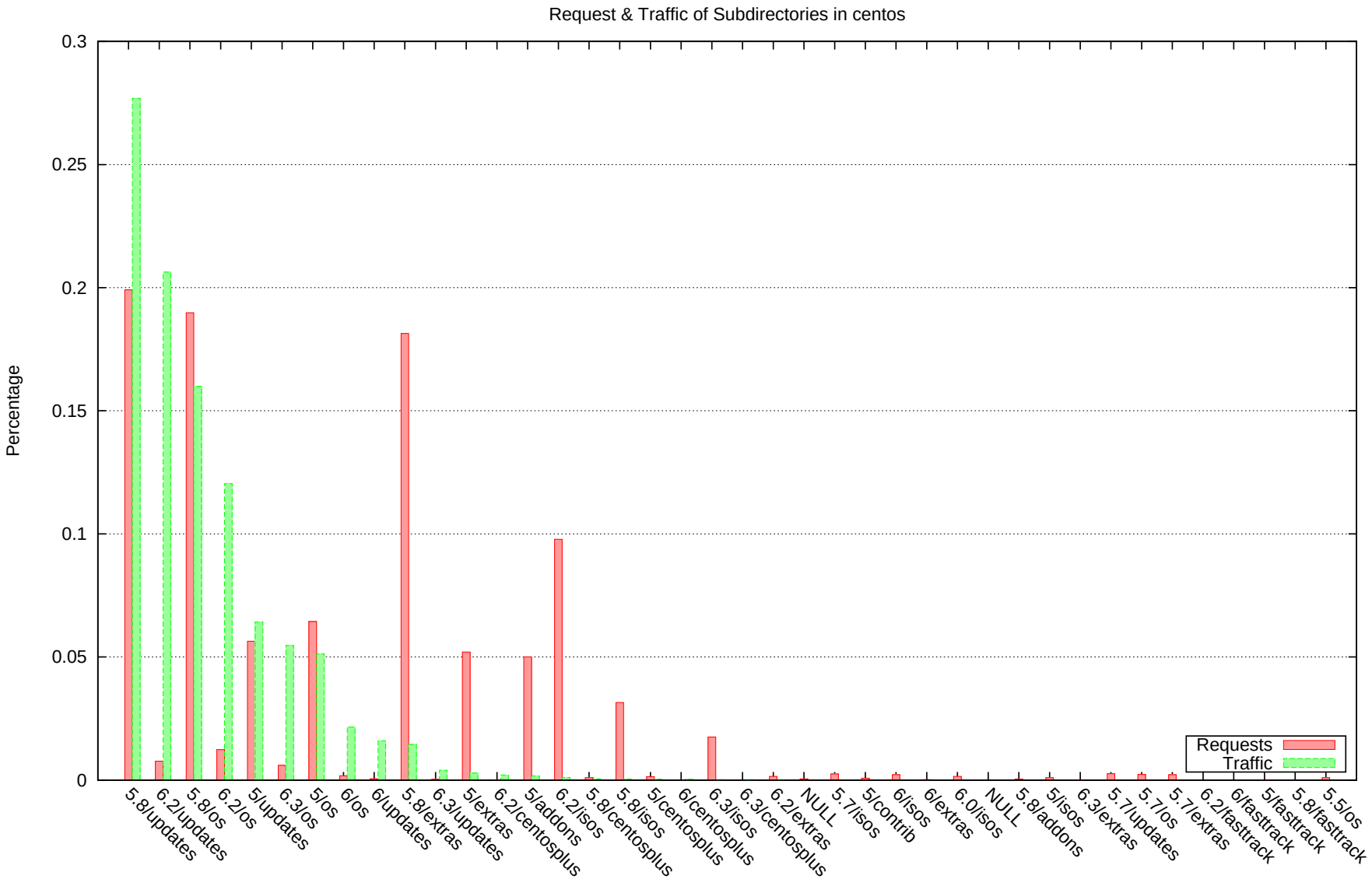
	Average	Ratio	Std.Dev	Ratio
Request Length	55846	0.464	882920	0.289
File Size	2665431	1.953	54320911	2.294
File Requests	1402.46	76.994	92327	11.029
File Traffic	74454404	32.12	1231957014	1.784
Session Duration	1051.19	1.145	1586.69	1.056
Session Requests	5.796	0.773	137.5	0.794
Session Traffic	336791	0.379	8931320	0.559

- Std.Dev stands for Standard Deviation.
- 'Ratio' in the third col stands for Ratio of Distribution Average to Global Average; 'Ratio' in the fifth col stands for Ratio of DIst Std.Dev to Global.
- Size & Traffic are in bytes, Duration is in seconds.

Requests & Traffic among CentOS Versions



Requests & Traffic among CentOS 2-level Subdirs



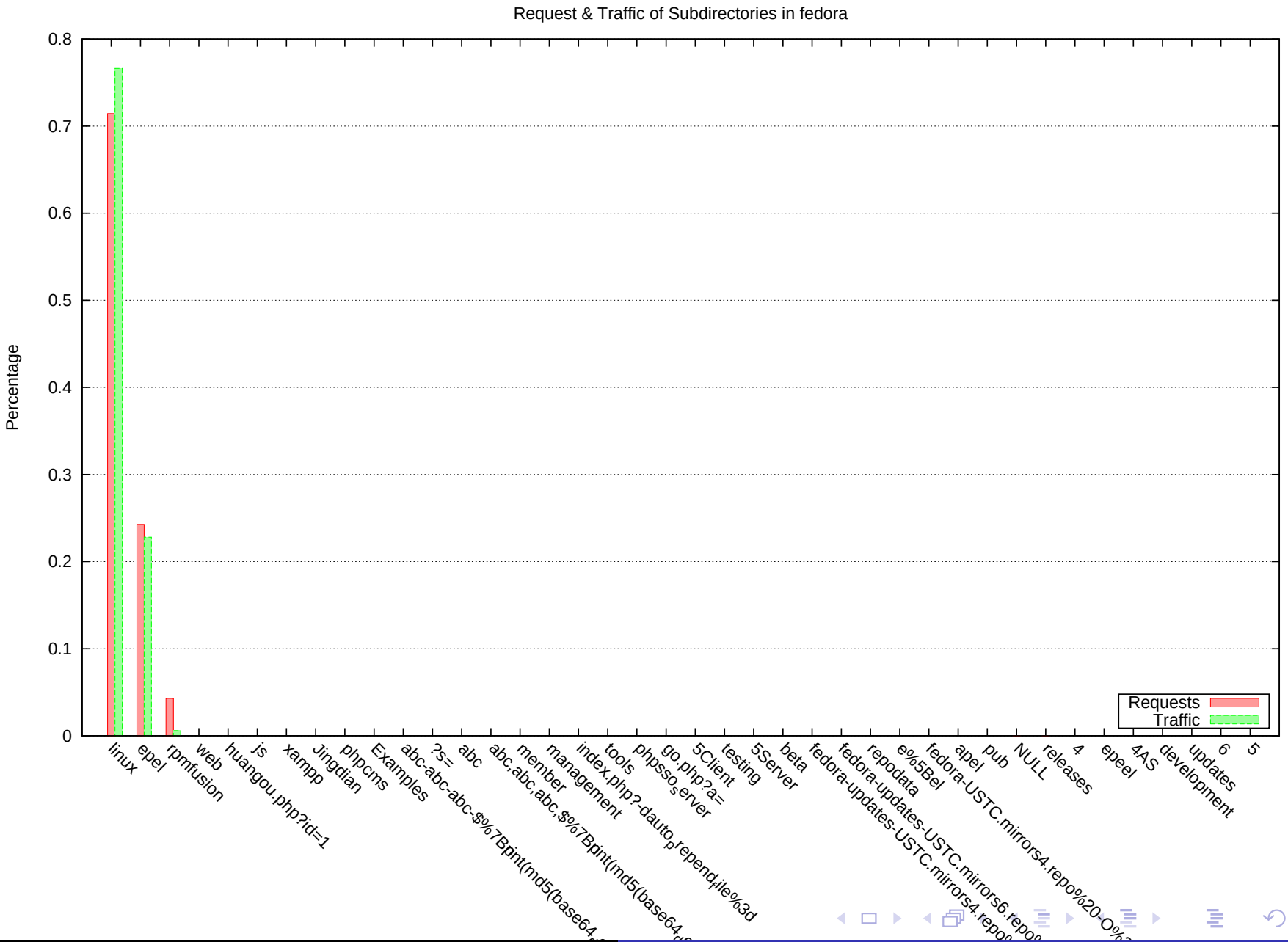
Fedora Basic Stat

	Value	% of total	Rank
Requests	36329528	11.0%	4
Traffic	7509.2 GB	20.3%	2
Files	924620	8.8%	2
FileSize	1207.8 GB	9.4%	2
Sessions	3596727	8.2%	2

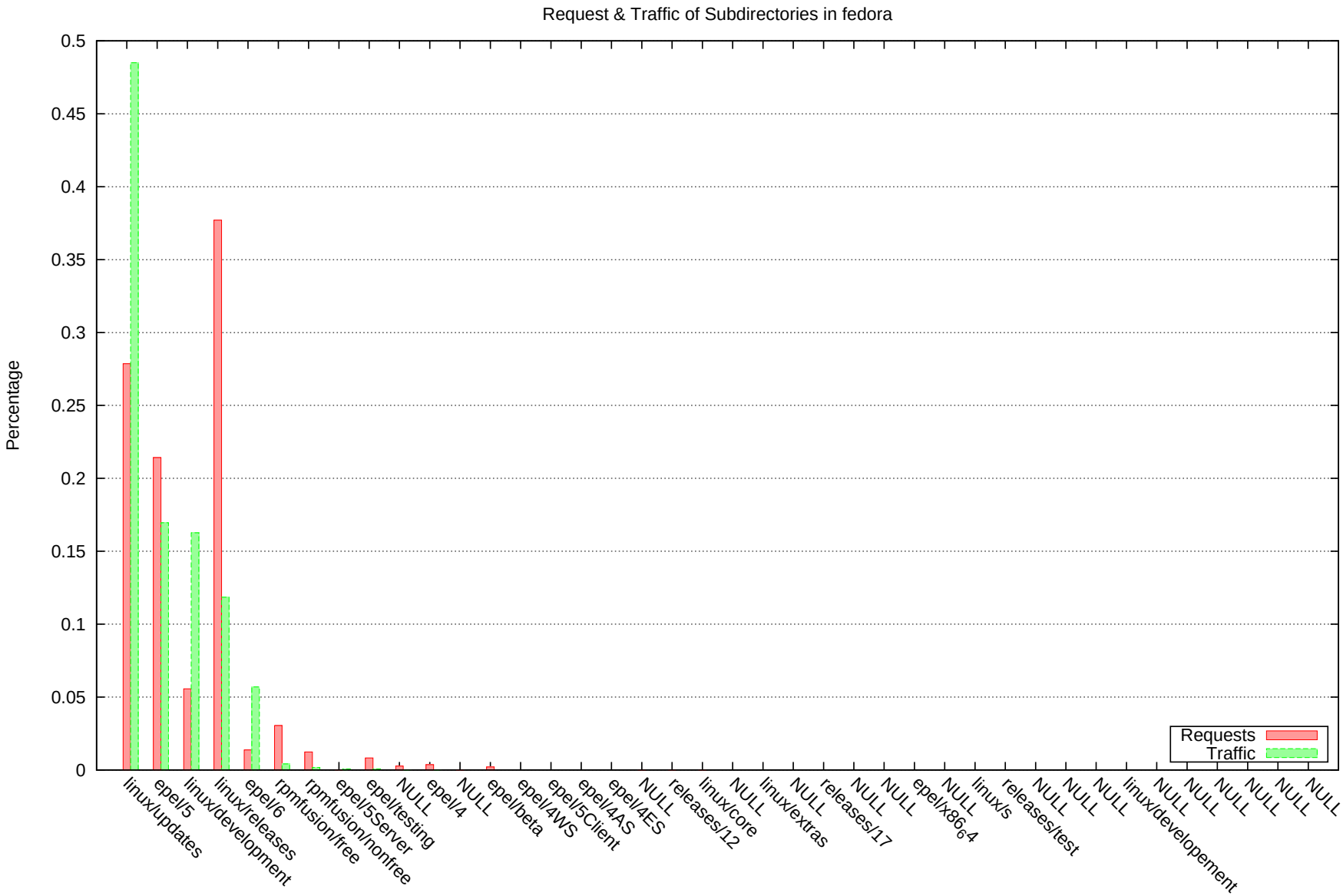
Fedora Basic Stat - Average

	Average	Ratio	Std.Dev	Ratio
Request Length	221945	1.843	2277229	0.744
File Size	1407150	1.031	21376814	0.903
File Requests	28.3134	1.554	6563	0.784
File Traffic	3601714	1.554	190851795	0.276
Session Duration	1226	1.336	1598	1.063
Session Requests	10.8047	1.440	251.43	1.452
Session Traffic	2238126	2.520	21424912	1.342

Requests & Traffic among Fedora Subdirs



Requests & Traffic among Fedora 2-level Subdirs



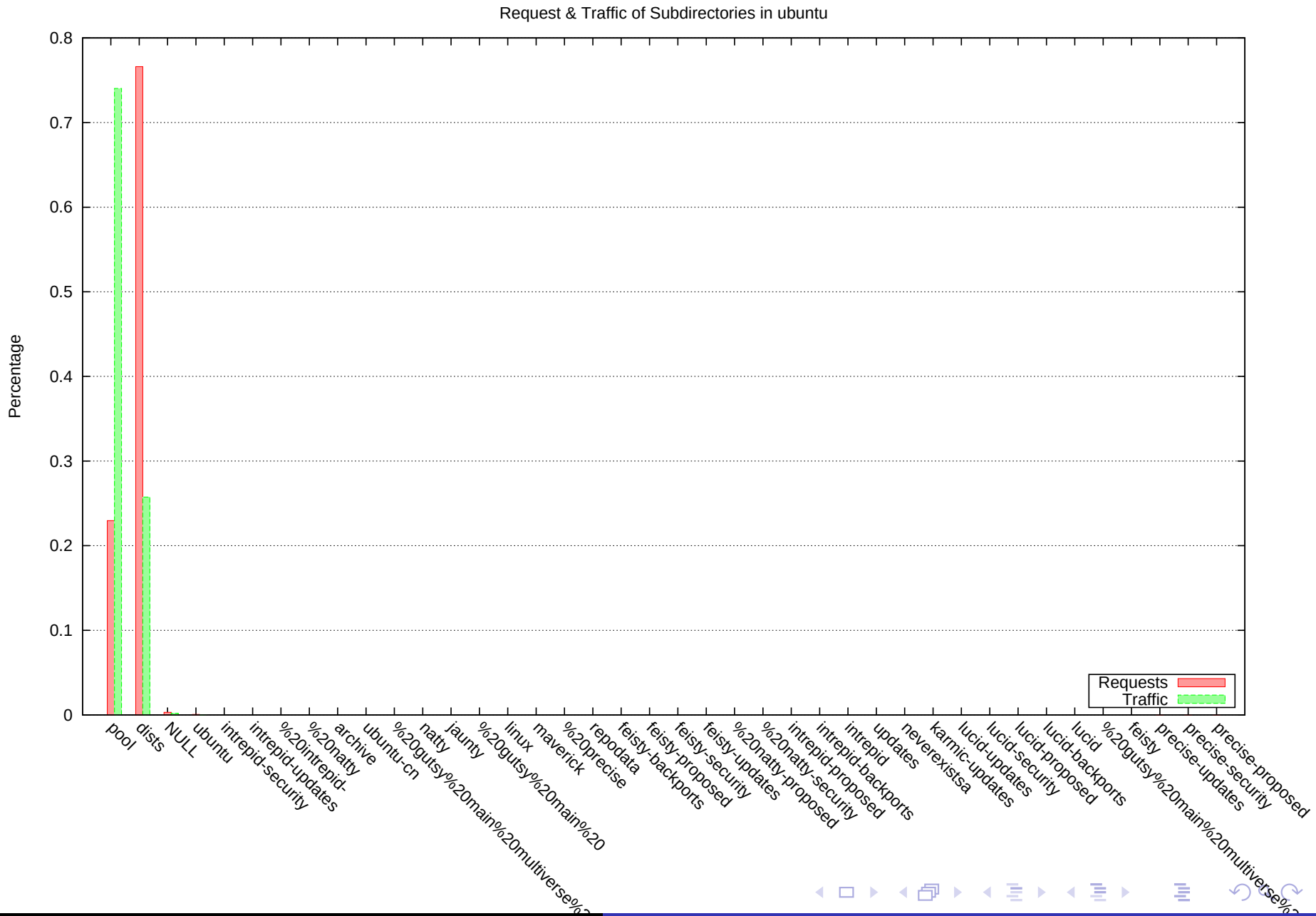
Ubuntu Basic Stat

	Value	% of total	Rank
Requests	19193451	5.8%	5
Traffic	5653.1 GB	15.3%	3
Files	608361	5.8%	5
FileSize	584.6 GB	4.6%	6
Sessions	160800	0.4%	4

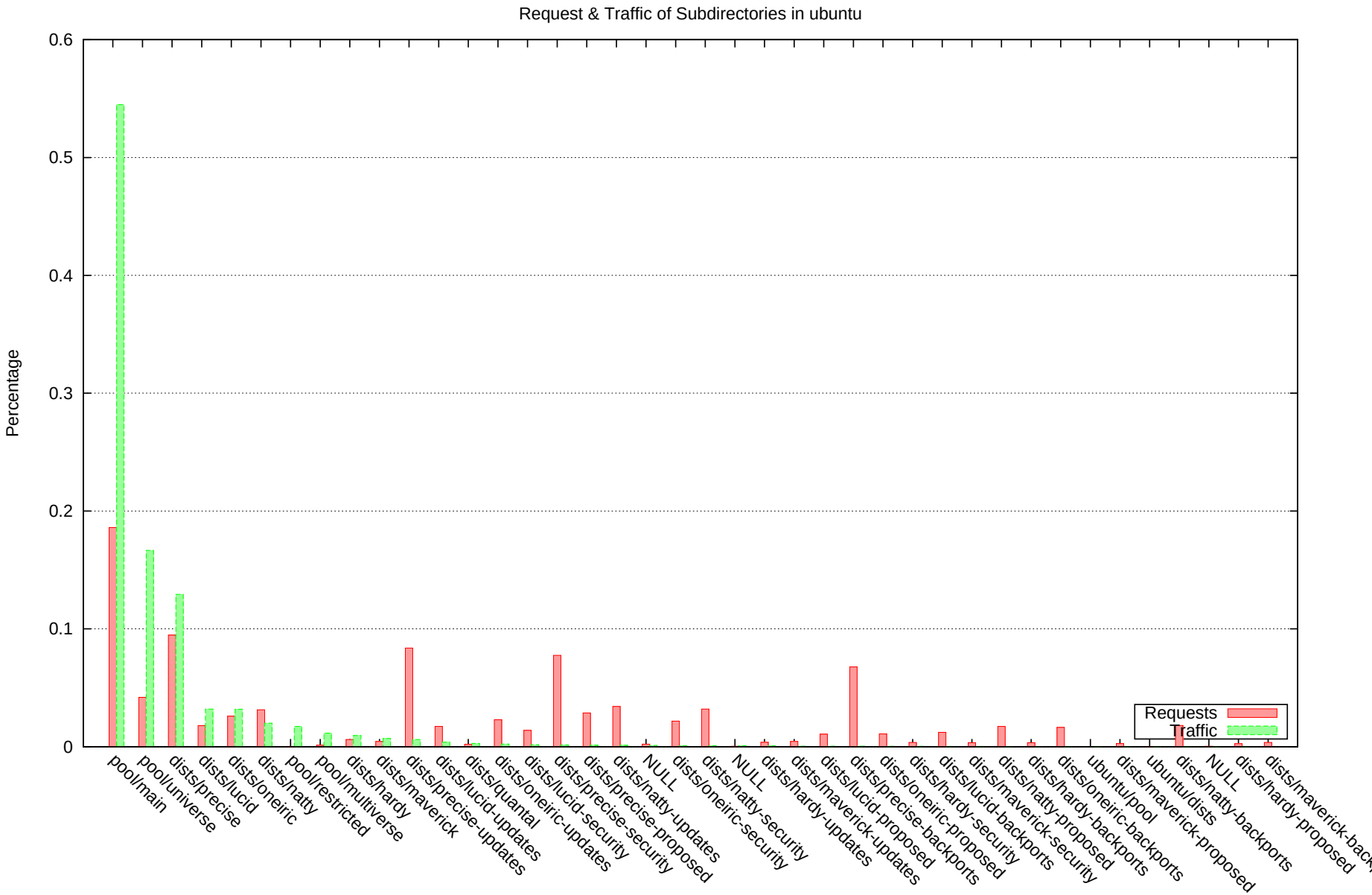
Ubuntu Basic Stat - Average

	Average	Ratio	Std.Dev	Ratio
Request Length	316251	2.626	2950601	0.964
File Size	1100038	0.806	8926254	0.377
File Requests	22.2714	1.222	860.6907	0.103
File Traffic	8152155	3.517	551019922	0.798
Session Duration	636.37	0.693	1053.94	0.701
Session Requests	110.93	14.788	294.68	1.702
Session Traffic	34260641	38.570	101247464	6.341

Requests & Traffic among Ubuntu Subdirs



Requests & Traffic among Ubuntu 2-level Subdirs



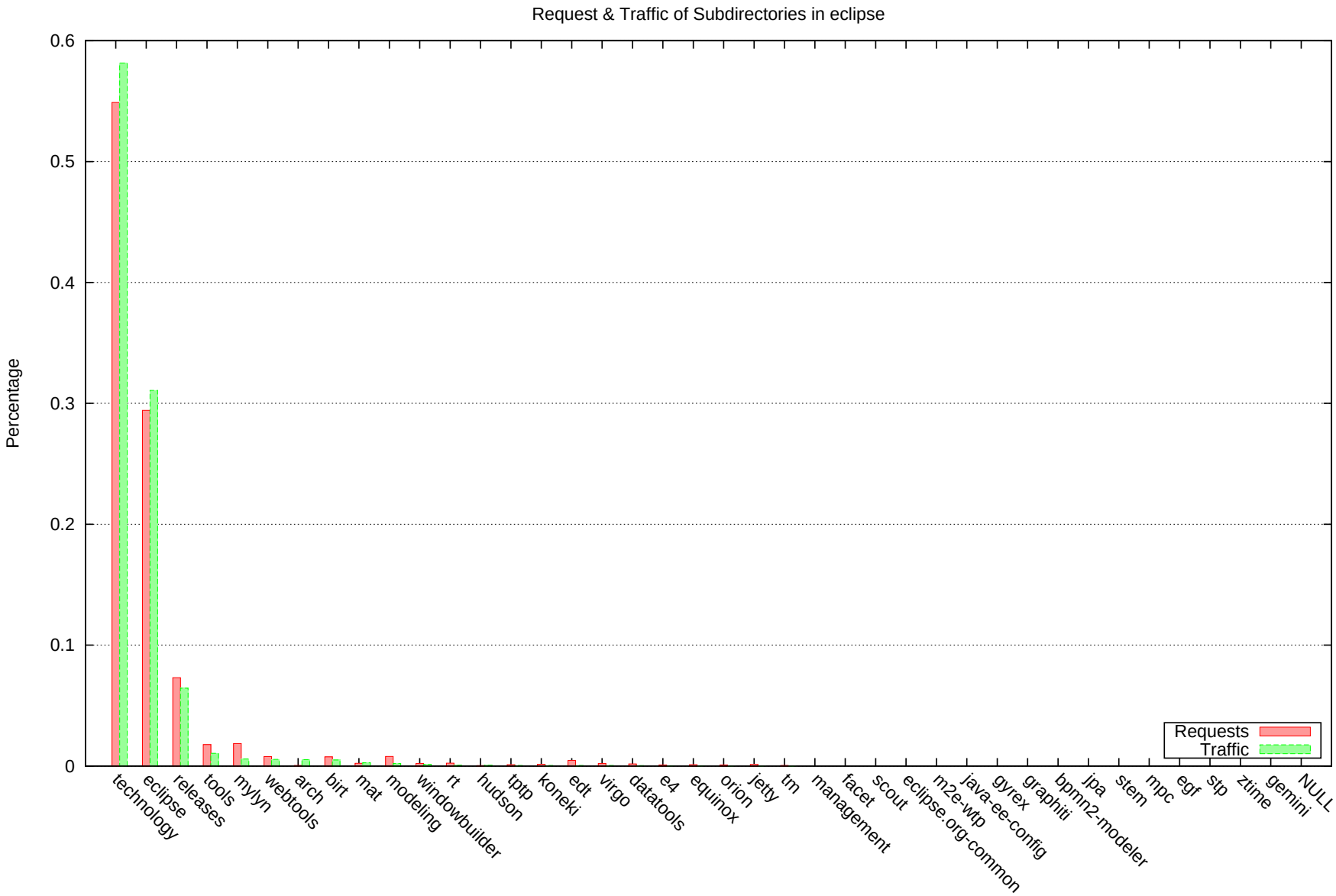
Eclipse Basic Stat

	Value	% of total	Rank
Requests	51279473	15.6%	3
Traffic	11498.6 GB	31.2%	1
Files	263355	2.5%	8
FileSize	161.1 GB	1.3%	14
Sessions	524586	1.2%	3

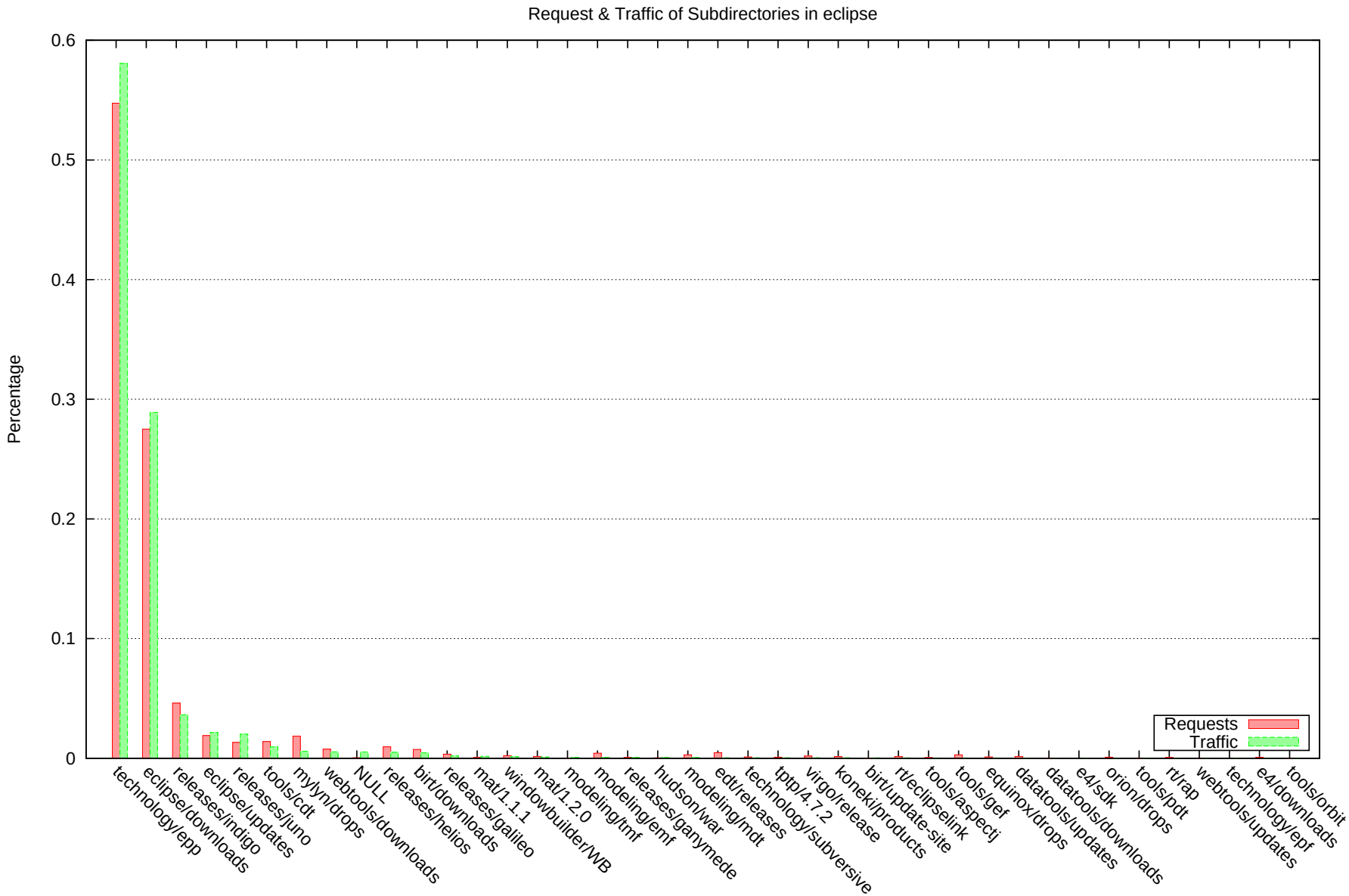
Eclipse Basic Stat - Average

	Average	Ratio	Std.Dev	Ratio
Request Length	240769	2.000	5813562	1.901
File Size	700002	0.513	8457323	0.357
File Requests	113.5340	6.233	17894.2	2.137
File Traffic	28156070	12.148	4197033259	6.079
Session Duration	673.17	0.733	1059.08	0.704
Session Requests	97.1089	12.944	782.80	4.520
Session Traffic	22130605	24.914	62746610	3.930

Requests & Traffic among Eclipse Subdirs



Requests & Traffic among Eclipse 2-level Subdirs



Outline

Data Source

- Nginx access log of mirrors.ustc.edu.cn
  - From 2012-05-22 to 2012-07-12, 51 days
  - 4041MB compressed, 62383MB decompressed
  - Thanks Guo JiaHua for providing data
- File list of mirrors.ustc.edu.cn crawled by spider
  - FTP for CPAN and CRAN
  - HTTP other directories
  - Need to detect symlinks to parent dir (e.g. /ubuntu/ubuntu/...)
- Scripts are written in bash and PHP.
  - All scripts are available at GitHub: https://github.com/bojieli/mirrors-log

Saving Data in BRIGHTHOUSE

- Scanning through such amount of data is time-consuming. And the data is not too large to fit in a relational database.
- I tried InfoBright and InfiniDB with $6.4 * 10^6$ rows of artificial data, InfiniDB is faster in queries, while InfoBright takes much less disk space.
  - InfoBright's compress rate is no less than gzip: 4316MB table size, compared to 4041MB gzip, not to mention that I have added some additional rows for faster statistics.
  - I do not have much disk space, so I choose InfoBright (a backend of MySQL).
  - Most of the queries I used (mostly GROUP BY, WHERE) take less than 2 minutes.
- Create two FIFOs foreach log: zcat logfile $>$ php (preprocessing) $>$ mysql-ib LOAD DATA INFILE

Preprocessing

- Make queries faster (I'm afraid of full-table scan)
- ip => integers: ipv4 ipv4_0 ipv4_1 ipv4_2 ipv4_3 ipv6_0 ipv6_1 ipv6_2 ipv6_3
- time => integers: time year yearday weekday daymin daytime hour
- status (200, 403, 404...)
- length (filesize)
- url => substrings: url_0 ...url_9 filename extension
- referer (I do not want to analyze it, for Mirrors is not a site of user-interaction)
- ua => ua_0 (Mozilla, Ubuntu...), ua (full)

Preprocessing

- PHP is *slow* ...only 3MB/s.
- At first preg_match (regular exps) take 85% time, then I optimized the regexp and it only takes 25% now.
- Xdebug show that stream_get_line (fgets) and fputs take about 50% of total time.
- InfoBright's data loading speed is 15MB/s (crawl_http.log). I don't think PHP's preprocessing work is harder than database's...
- Maybe PHP's interpretive nature makes it much slower than C. Anyone give a benchmark for Python etc?

Files Table

- Many files on mirrors are never accessed, so we have to make a full list of files on mirrors.
- Preprocessed url, filename, extension and filesize are recorded for each file.
- When processing logs and files, escape characters and VARCHAR max length should be taken care of, and filename should not be limited to a simple regular expression, since there are always exceptions: UTF-8 strings in malicious request, ",v" files in CVS ...

GNUPLOT

- GNUPLOT is pretty flexible in plotting, the documentation and online demos are consulted many times.
- However GNUPLOT is not flexible in processing data. It is strong type, where integers and strings need to be explicitly converted. And the integer type is limited to -2^{31} $2^{31}-1$.
- When the query result needs to be postprocessed, I write a simple sed s/regexp/replace/ for simple replacement, or awk NR%n==0 for sampling. When it comes to accumulation or some complex stuff, a PHP script goes between.

Discovering Sessions

- Sequentially scan the log. If a request can fill into an existing session, update it; otherwise create a new one and flush the timeout session if exists.
- Garbage collection of timeout sessions: if (count(array) > gc_limit) { unset timeout sessions; gc_limit = count(array) * 1.5; }
  - Inspired by JavaScript GC algorithm in IE7: If recycled memory is less than 15% of total, then limit is doubled; if recycled memory is more than 85%, then reset to initial value.
- Maintain a query buffer of 10000 rows to reduce query num.
- The PHP takes 4 hours, 15 times slower than C (see the following section).

Some Bugs Due to DIE Time Functions (PHP)

- PHP: Find some sessions with negative session length. The only explanation is that logs are not in time increasing order. Dive into the log table, only to find that a timestamp matches records in both April 31 and May 1.
- In fact, mktime() accepts 5 parameters just like mktime(struct time_t) in C, while its month is 1 to 12, different from 0 to 11. However strptime() is a simple encapsulation of its C respective.
- I dare not to use DATETIME in MySQL, because the arithmetic of such timestamps is tricky, and I would rather implement it in SQL or PHP.
- Thanks Godness, no timezone problem this time.

Some Bugs Due to DIE Time Functions (C)

- C: malloc() fall into deadlock, so strange, I GDBed an evening and no answer. Change some code and accidentally got Segmentation Error in time().
- In fact, time() need a parameter of type time_t and it is dynamically linked. I used it without any parameter, and time() treats the garbage on stack as its parameter. If it is NULL, nothing happens; if not, the position is considered a struct time_t and unpredictable stuff happen.

Outline

Query Optimization

- `select count(*), ipv4, count(*) as c, sum(length) as s, concat(ipv4_0,'.',ipv4_1,'.',ipv4_2,'.',ipv4_3) from log where ipv4 is not null group by ipv4 order by s limit 40;`
- Query_time: 798.179622
  - 2012-07-30 16:32:12 Cnd(0): VC:0(t0a0) IS NOT NULL (0)
  - 2012-07-30 16:33:02 Aggregating: 318575688 tuples left.
  - 2012-07-30 16:45:29 Aggregated (1415554 gr). Omitted packrows: 0 + 0 partially, out of 5020 total.
  - 2012-07-30 16:45:29 Heap Sort initialized for 1415554 rows, 8+61 bytes each.
  - 2012-07-30 16:45:30 Total data packs actually loaded (approx.): 35139
- Infobright is column-based, so cross-column queries are slow. Try to generate IP string from 'ipv4' field.

Query Optimization (continued)

- `SELECT ipv4, COUNT(*)/$COUNT, SUM(length)/$LENGTH AS c, CONCAT((ipv4 & 255<<24)>>24, '.', (ipv4 & 255<<16)>>16, '.', (ipv4 & 255<<8)>>8, '.', (ipv4 & 255)) FROM log WHERE ipv4 IS NOT NULL GROUP BY ipv4 ORDER BY c DESC LIMIT 40;`
- Query_time: 585.069205
- InfoBright packs column data into packages, and pre-computed MAX, MIN, AVG and GROUP data for each package. WHERE clause requires re-computation of these data:
  - 2012-07-30 14:33:06 Cnd(0): VC:0(t0a0) IS NOT NULL (0)
  - 2012-07-30 14:33:55 Aggregating: 318575688 tuples left.
  - 2012-07-30 14:42:47 Generating output.

Query Optimization (continued)

- `SELECT ipv4, COUNT(*)/$COUNT, SUM(length)/$LENGTH AS c, CONCAT((ipv4 & 255<<24)>>24, '.', (ipv4 & 255<<16)>>16, '.', (ipv4 & 255<<8)>>8, '.', (ipv4 & 255)) FROM log GROUP BY ipv4 ORDER BY c DESC LIMIT 40;`
- Query_time: 328.580979
- WHERE clause is removed, but the result is wrong (includes a large NULL which stands for IPv6).
  - 2012-07-30 14:45:41 Unoptimized expression near '/'
  - 2012-07-30 14:45:41 Unoptimized expression near '/'
  - 2012-07-30 14:45:41 Unoptimized expression near 'concat'
  - 2012-07-30 14:45:41 Aggregating: 328976877 tuples left.
- InfoBright calculates these columns for each tuple, no wonder it is slow. Keep the core subquery small. Moving these calculation outside may help.

Query Optimization (continued)

- `SELECT ipv4, c/$COUNT, s/$LENGTH, CONCAT((ipv4 & 255<<24)>>24, '.', (ipv4 & 255<<16)>>16, '.', (ipv4 & 255<<8)>>8, '.', (ipv4 & 255)) FROM (SELECT ipv4, COUNT(*) AS c, SUM(length) AS s FROM log GROUP BY ipv4 HAVING (ipv4 IS NOT NULL) ORDER BY s DESC LIMIT 40) AS t;`
- Query_time: 138.297943
  - Total data packs actually loaded (approx.): 10040
- Use HAVING clause to filter NULL. The internal exec order is: FROM TABLE, JOIN, OUTER JOIN, WHERE, SELECT clause, GROUP BY, HAVING, ORDER BY, LIMIT, projection & output.
- Turn on MySQL's query_cache: identical queries should not be re-executed if I change some GNUPLOT command and run the script again.
- Is there any way to make the query faster? I don't know.

Query Optimization (continued)

- Another example of moving expressions 'outside' inner query:
- `SELECT ipv4_0, c/$COUNT, s/$LENGTH FROM (SELECT IFNULL(ipv4_0, 'IPv6') AS ipv4_0, COUNT(*) AS c, SUM(length) AS s FROM log GROUP BY ipv4_0 ORDER BY s DESC LIMIT 40) AS t;`
- Query_time: 263.048662
- `SELECT IFNULL(ipv4_0, 'IPv6'), c/$COUNT, s/$LENGTH FROM (SELECT ipv4_0, COUNT(*) AS c, SUM(length) AS s FROM log GROUP BY ipv4_0 ORDER BY s DESC LIMIT 40) AS t;`
- Query_time: 84.589837
- I cannot believe my eyes at the first sight of these figures.

Query Optimization is Not Everything

- `SELECT url_0, COUNT(*)/$COUNT, SUM(files.filesize)/$LENGTH AS c FROM files WHERE NOT EXISTS (SELECT * FROM log WHERE log.filename = files.filename) GROUP BY url_0 ORDER BY c DESC LIMIT 40;`
- The query ran 12 hours and I have to kill it. Infobright treats NOT EXISTS clause as dependent subquery and might have to execute it for each row.
- I don't know how to use cursors and storage procedure in MySQL, and more importantly Infobright does not support dynamic update of tables, so SELECT INTO is impossible.
- I write a C program (using mysqlclient API) to count occurences of each url in log and write it into another table.
- Performance:
  - 10M rows in files table, 329M rows in log table
  - 17.4 minutes in total
  - 440000 rows per second at Probe phase
  - 460 MB of memory usage (360MB resident)

Simulating Hash JOIN

- I found my algorithm actually named Hash JOIN after programming finished. It is a fast JOIN algorithm for RDBMS. The task is to find, for each distinct value of the join attribute, the set of tuples in each relation which have that value. (Wikipedia)
- My work is to count the occurence in log table of each url in files table.
  - Build step: Traverse files table and add the url field of each row to a hash list.
  - Probe step: Traverse log table and add the counter of the corresponding hash slot.
  - Output step: Traverse files table again and output the corresponding counters. The output is FIFOed to LOAD DATA INFILE.
- Easy to shard horizontally. Steps are similar to Map, Shuffle and Reduce.

Simulating Hash JOIN (continued)

- Since there might be many files (currently only about 10M), storing the hash list entirely in memory is my top concern.
- Use one hash for position, two additional hashes for checking, and do not store the original string. 10^{-22} probability of undetected hash collision, but it is low enough for an analytic program.
- Only requires 12 bytes for each slot (2 * sizeof(int) hash check, sizeof(int) counter)
- Hash algorithm: DJBX33A (hash = ((hash«5) + hash) + *key++), used by PHP, Apache and more. Use three seeds for position and check hash.
- Hash collision: Linearly find the next empty slot. This is simple (I'm afraid of pointers) and memory-saving.

The End

- I intended to finish it in 3 days, but because I'm unfamiliar with these tools, it takes me a week (or more if including preparation time).
- Access logs are the origin of discoveries on access patterns. Data is precious, while disk is cheap. Please do not delete them after 52 days.
- I cannot draw conclusion on 'trends', for there is not data for an adequately long time.
- If you want other statistics, please email me and I will query it (if I have time :).

写在最后

- Thanks all maintainers and supporters of mirrors.ustc.edu.cn!
- All scripts and source of this slides are available at GitHub: https://github.com/bojieli/mirrors-log
- 终于搞定了中文字体问题，不过懒得翻译了……