Run JMeter test plan from CMD:

For JMeter-ScrumTaskboard testcases-After Code Improvements, Login\_Add New Developer\_Delete Old Developer\_Logout testcases:

* **100 threads**:

D:\apache-jmeter-2.13\bin>jmeter -n -t Login-Add\_New\_Developer-Delete\_Old\_Developer-Logout-100\_threads.jmx -l Login-Add\_New\_Developer-Delete\_Old\_Developer-Logout-100\_threads.jtl

***Output from CMD*:**

Creating summariser <summary>

Created the tree successfully using Login-Add\_New\_Developer-Delete\_Old\_Developer-Logout-100\_threads.jmx

Starting the test @ Sat Mar 12 17:25:16 EET 2016 (1457796316464)

Waiting for possible shutdown message on port 4445

summary + 119 in 18s = 6.8/s Avg: 6997 Min: 50 Max: 8643 Err: 0 (0.00%) Active: 100 Started: 100 Finished: 0

summary + 333 in 26s = 13.0/s Avg: 7664 Min: 244 Max: 18441 Err: 54 (16.22%) Active: 100 Started: 100 Finished: 0

summary = 452 in 43.3s = 10.4/s Avg: 7488 Min: 50 Max: 18441 Err: 54 (11.95%)

summary + 194 in 30s = 6.5/s Avg: 15300 Min: 83 Max: 37799 Err: 148 (76.29%) Active: 100 Started: 100 Finished: 0

summary = 646 in 73.4s = 8.8/s Avg: 9834 Min: 50 Max: 37799 Err: 202 (31.27%)

summary + 287 in 30s = 9.6/s Avg: 8113 Min: 5 Max: 30047 Err: 121 (42.16%) Active: 89 Started: 100 Finished: 11

summary = 933 in 103s = 9.0/s Avg: 9305 Min: 5 Max: 37799 Err: 323 (34.62%)

summary + 167 in 10s = 17.0/s Avg: 7985 Min: 6 Max: 27722 Err: 77 (46.11%) Active: 0 Started: 100 Finished: 100

summary = 1100 in 113s = 9.7/s Avg: 9104 Min: 5 Max: 37799 Err: 400 (36.36%)

Tidying up ... @ Sat Mar 12 17:27:09 EET 2016 (1457796429948)

... end of run

***The report for this test case was generated in:***

D:\School\Master\dissertation-thesis\experiments\JMeter-ScrumTaskboard testcases-After Code Improvements\Login\_Add New Developer\_Delete Old Developer\_Logout testcases\100 threads\Login-Add\_New\_Developer-Delete\_Old\_Developer-Logout-100\_threads.jtl file

* Same error values as the ones obtained in Summary Report listener (when running from JMeter GUI) => no improvement
* **300 threads:**

D:\apache-jmeter-2.13\bin>jmeter -n -t Login-Add\_New\_Developer-Delete\_Old\_Developer-Logout-300\_threads.jmx -l Login-Add\_New\_Developer-Delete\_Old\_Developer-Logout-300\_threads.jtl

***Output from CMD*:**

Creating summariser <summary>

Created the tree successfully using Login-Add\_New\_Developer-Delete\_Old\_Developer-Logout-300\_threads.jmx

Starting the test @ Sat Mar 12 20:24:01 EET 2016 (1457807041938)

Waiting for possible shutdown message on port 4445

summary + 76 in 28s = 2.7/s Avg: 19843 Min: 2022 Max: 25883 Err: 42 (55.26%) Active: 300 Started: 300 Finished: 0

summary + 592 in 30.1s = 19.6/s Avg: 14754 Min: 4 Max: 57102 Err: 445 (75.17%) Active: 299 Started: 300 Finished: 1

summary = 668 in 58s = 11.5/s Avg: 15333 Min: 4 Max: 57102 Err: 487 (72.90%)

summary + 545 in 30s = 18.2/s Avg: 19637 Min: 4 Max: 77499 Err: 314 (57.61%) Active: 287 Started: 300 Finished: 13

summary = 1213 in 88s = 13.8/s Avg: 17267 Min: 4 Max: 77499 Err: 801 (66.03%)

summary + 302 in 30.1s = 10.0/s Avg: 20241 Min: 500 Max: 48827 Err: 144 (47.68%) Active: 283 Started: 300 Finished: 17

summary = 1515 in 118s = 12.8/s Avg: 17859 Min: 4 Max: 77499 Err: 945 (62.38%)

summary + 272 in 30s = 9.1/s Avg: 27097 Min: 7587 Max: 73874 Err: 135 (49.63%) Active: 270 Started: 300 Finished: 30

summary = 1787 in 148s = 12.1/s Avg: 19266 Min: 4 Max: 77499 Err: 1080 (60.44%)

summary + 290 in 30.1s = 9.6/s Avg: 37102 Min: 7484 Max: 98274 Err: 143 (49.31%) Active: 254 Started: 300 Finished: 46

summary = 2077 in 178s = 11.7/s Avg: 21756 Min: 4 Max: 98274 Err: 1223 (58.88%)

summary + 242 in 30s = 8.1/s Avg: 30657 Min: 9291 Max: 79455 Err: 119 (49.17%) Active: 232 Started: 300 Finished: 68

summary = 2319 in 208s = 11.2/s Avg: 22685 Min: 4 Max: 98274 Err: 1342 (57.87%)

summary + 234 in 30.2s = 7.7/s Avg: 28562 Min: 12082 Max: 70408 Err: 131 (55.98%) Active: 211 Started: 300 Finished: 89

summary = 2553 in 238s = 10.7/s Avg: 23224 Min: 4 Max: 98274 Err: 1473 (57.70%)

summary + 276 in 30s = 9.3/s Avg: 23380 Min: 7 Max: 64572 Err: 153 (55.43%) Active: 171 Started: 300 Finished: 129

summary = 2829 in 268s = 10.6/s Avg: 23239 Min: 4 Max: 98274 Err: 1626 (57.48%)

summary + 351 in 30.1s = 11.7/s Avg: 16112 Min: 14 Max: 51011 Err: 143 (40.74%) Active: 57 Started: 300 Finished: 243

summary = 3180 in 298s = 10.7/s Avg: 22452 Min: 4 Max: 98274 Err: 1769 (55.63%)

summary + 120 in 12s = 10.4/s Avg: 6591 Min: 6 Max: 26216 Err: 61 (50.83%) Active: 0 Started: 300 Finished: 300

summary = 3300 in 309s = 10.7/s Avg: 21875 Min: 4 Max: 98274 Err: 1830 (55.45%)

Tidying up ... @ Sat Mar 12 20:29:11 EET 2016 (1457807351628)

... end of run

***The report for this test case was generated in:***

D:\School\Master\dissertation-thesis\experiments\JMeter-ScrumTaskboard testcases-After Code Improvements\Login\_Add New Developer\_Delete Old Developer\_Logout testcases\300 threads\Login-Add\_New\_Developer-Delete\_Old\_Developer-Logout-300\_threads.jtl file

* Greater error values (55.45%) than the ones obtained in Summary Report listener (54.55%) (when running from JMeter GUI) => no improvement, actually worst results were obtained when running JMeter in Non-GUI mode
* **500 threads:**

D:\apache-jmeter-2.13\bin>jmeter -n -t Login-Add\_New\_Developer-Delete\_Old\_Developer-Logout-500\_threads.jmx -l Login-Add\_New\_Developer-Delete\_Old\_Developer-Logout-500\_threads.jtl

***Output from CMD*:**

Creating summariser <summary>

Created the tree successfully using Login-Add\_New\_Developer-Delete\_Old\_Developer-Logout-500\_threads.jmx

Starting the test @ Sat Mar 12 20:38:47 EET 2016 (1457807927459)

Waiting for possible shutdown message on port 4445

summary + 533 in 12.3s = 43.4/s Avg: 2803 Min: 1994 Max: 10493 Err: 532 (99.81%) Active: 500 Started: 500 Finished: 0

summary + 1432 in 30s = 47.8/s Avg: 3870 Min: 344 Max: 39716 Err: 1421 (99.23%) Active: 423 Started: 500 Finished: 77

summary = 1965 in 42.3s = 46.5/s Avg: 3580 Min: 344 Max: 39716 Err: 1953 (99.39%)

summary + 880 in 31.3s = 28.1/s Avg: 13418 Min: 7 Max: 71453 Err: 809 (91.93%) Active: 380 Started: 500 Finished: 120

summary = 2845 in 74s = 38.7/s Avg: 6623 Min: 7 Max: 71453 Err: 2762 (97.08%)

summary + 466 in 29s = 16.3/s Avg: 29088 Min: 6 Max: 97769 Err: 376 (80.69%) Active: 352 Started: 500 Finished: 148

summary = 3311 in 102s = 32.4/s Avg: 9785 Min: 6 Max: 97769 Err: 3138 (94.77%)

summary + 350 in 30.5s = 11.5/s Avg: 29614 Min: 62 Max: 118223 Err: 234 (66.86%) Active: 320 Started: 500 Finished: 180

summary = 3661 in 133s = 27.6/s Avg: 11681 Min: 6 Max: 118223 Err: 3372 (92.11%)

summary + 354 in 30s = 11.8/s Avg: 23026 Min: 612 Max: 113734 Err: 185 (52.26%) Active: 280 Started: 500 Finished: 220

summary = 4015 in 163s = 24.7/s Avg: 12681 Min: 6 Max: 118223 Err: 3557 (88.59%)

summary + 266 in 30s = 8.9/s Avg: 39949 Min: 9490 Max: 106930 Err: 140 (52.63%) Active: 257 Started: 500 Finished: 243

summary = 4281 in 193s = 22.2/s Avg: 14375 Min: 6 Max: 118223 Err: 3697 (86.36%)

summary + 249 in 30s = 8.3/s Avg: 32390 Min: 260 Max: 113515 Err: 126 (50.60%) Active: 220 Started: 500 Finished: 280

summary = 4530 in 223s = 20.3/s Avg: 15366 Min: 6 Max: 118223 Err: 3823 (84.39%)

summary + 225 in 30.1s = 7.5/s Avg: 29330 Min: 26 Max: 98744 Err: 137 (60.89%) Active: 193 Started: 500 Finished: 307

summary = 4755 in 253s = 18.8/s Avg: 16026 Min: 6 Max: 118223 Err: 3960 (83.28%)

summary + 244 in 32.1s = 7.6/s Avg: 29929 Min: 9386 Max: 66046 Err: 140 (57.38%) Active: 154 Started: 500 Finished: 346

summary = 4999 in 285s = 17.6/s Avg: 16705 Min: 6 Max: 118223 Err: 4100 (82.02%)

summary + 281 in 27.5s = 10.2/s Avg: 16046 Min: 7 Max: 50477 Err: 119 (42.35%) Active: 85 Started: 500 Finished: 415

summary = 5280 in 312s = 16.9/s Avg: 16670 Min: 6 Max: 118223 Err: 4219 (79.91%)

summary + 220 in 23s = 9.6/s Avg: 8202 Min: 3 Max: 33178 Err: 109 (49.55%) Active: 0 Started: 500 Finished: 500

summary = 5500 in 335s = 16.4/s Avg: 16331 Min: 3 Max: 118223 Err: 4328 (78.69%)

Tidying up ... @ Sat Mar 12 20:44:22 EET 2016 (1457808262979)

... end of run

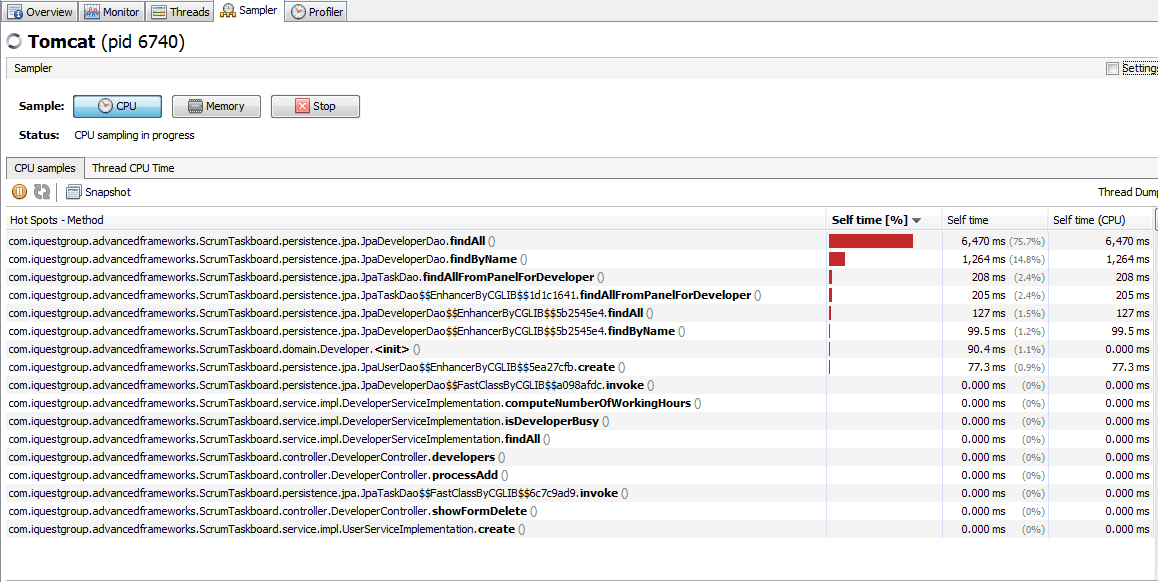
***The report for this test case was generated in:***

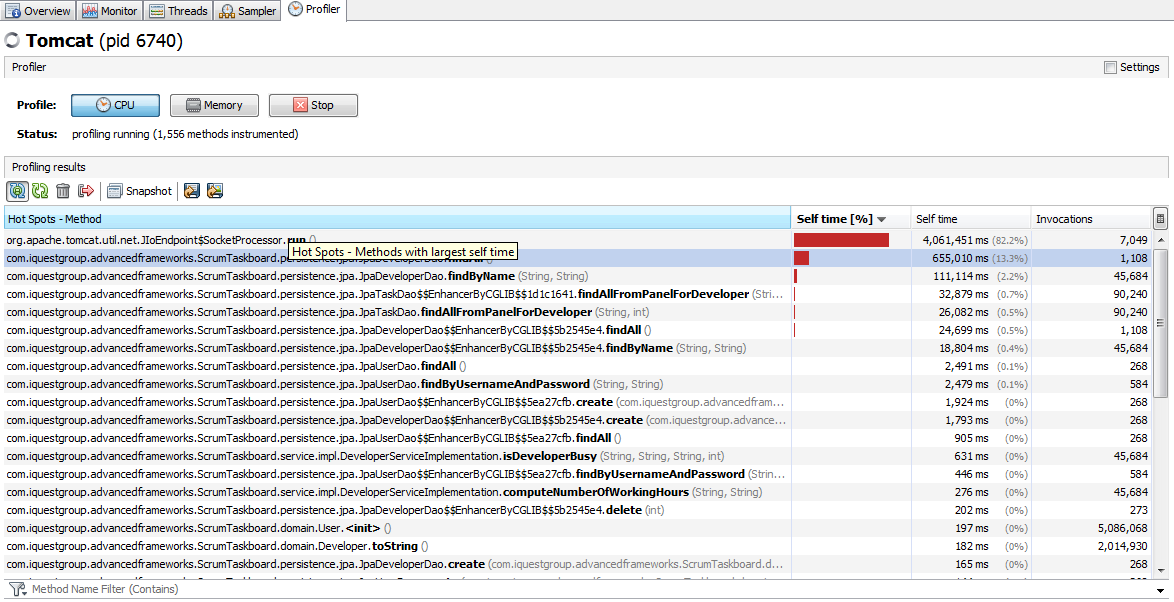
D:\School\Master\dissertation-thesis\experiments\JMeter-ScrumTaskboard testcases-After Code Improvements\Login\_Add New Developer\_Delete Old Developer\_Logout testcases\500 threads\Login-Add\_New\_Developer-Delete\_Old\_Developer-Logout-500\_threads.jtl file

* Greater error values (78.69%) than the ones obtained in Summary Report listener (77.58%) (when running from JMeter GUI) => no improvement, actually worst results were obtained when running JMeter in Non-GUI mode

Conclusions after analyzing VisualVM results:

* When profiling/sampling different application use-cases, the main CPU bottleneck is caused by methods from the persistence layer (when querying the database) => further improvements should be considered on the database level





DB improvements (how to explain/debug SQL queries)

<http://www.ducea.com/2006/11/06/identifying-mysql-slow-queries/>

<https://dev.mysql.com/doc/refman/5.5/en/execution-plan-information.html>

* in C:\Program Files\MySQL\MySQL Server 5.7\bin :

mysql –u root –p

apoi parola : root

* log-urile sunt in : C:\ProgramData\MySQL\MySQL Server 5.7\Data

EXPLAIN SELECT \* FROM Developer;

* Problema e apelul repetat de metode din showDevelopers.jsp

EXPLAIN EXTENDED SELECT \* from Task t where t.status like 'not-taken';

* Am adaugat index pt status pe field in clasa de Spring entity Task:

@Column(name="status")

@Index(name="statusIDX")

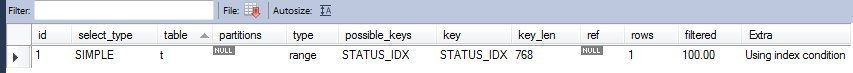
**private** String status;

dar MySql-ul nu ia in considerare acest Index la rularea query-ului

* Am adaugat indexul si din MySql pe tabela task :

**ALTER TABLE `scrum`.`task`**

**ADD INDEX `STATUS\_IDX` (`status` ASC);**



* Indexul pt status este folosit acum
* In fisierul din MySQl (\*-slow.log) apar mai putine aparitii ale query-ului:

SELECT \* from Task t where t.status like ‘…’; dupa adaugarea index-ului

EXPLAIN EXTENDED SELECT \* FROM Developer d WHERE d.firstName = 'miha' and d.lastName = 'man';

* Am adaugat composite index pt field-urile firstName and lastName in clasa de Spring entity Developer:

@Entity

@Table(name="developer", uniqueConstraints={@UniqueConstraint(columnNames={"firstName","lastName"})})

**public** **class** Developer {

@Index(name="firstNameIDX")

@Column(name="firstName")

**private** String firstName;

@Index(name="lastNameIDX")

@Column(name="lastName")

**private** String lastName;

**…**

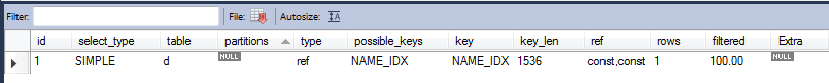
}

dar MySql-ul nu ia in considerare acest Index la rularea query-ului

* Am adaugat indexul si din MySql pe tabela developer :

**ALTER TABLE `scrum`.`developer`**

**ADD INDEX `NAME\_IDX` (`firstName` ASC, `lastName` ASC);**

****

* Indexul compus pt cautarea developerilor dupa firstName si lastName este folosit acum
* In fisierul din MySQl (\*-slow.log) nu mai apare query-ul:

SELECT \* FROM Developer d WHERE d.firstName = ‘…’ and d.lastName = ‘…’; dupa adaugarea index-ului si in JMeter apare error code: 500 – Internal Server Error

* Adaugarea unui index pt un coloana apartinand unei tabele din Java (pe clasa de model) se foloseste la rularea SQL query-urilor doar daca in persistence.xml avem:

**<property name="hibernate.hbm2ddl.auto" value="create"/>**

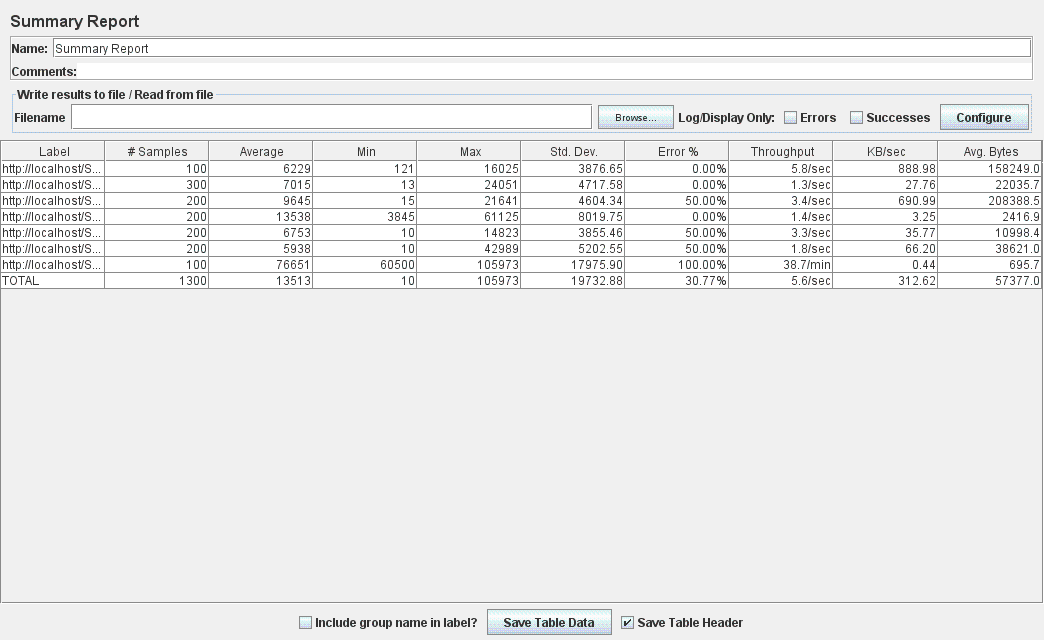
ceea ce presupune re-crearea DB-ului la fiecare restart de server.

Dupa o prima generare a DB-ului, cu indexul adaugat din Java, am schimbat valoarea acestei proprietati in update pt a nu pierde din nou datele din DB:

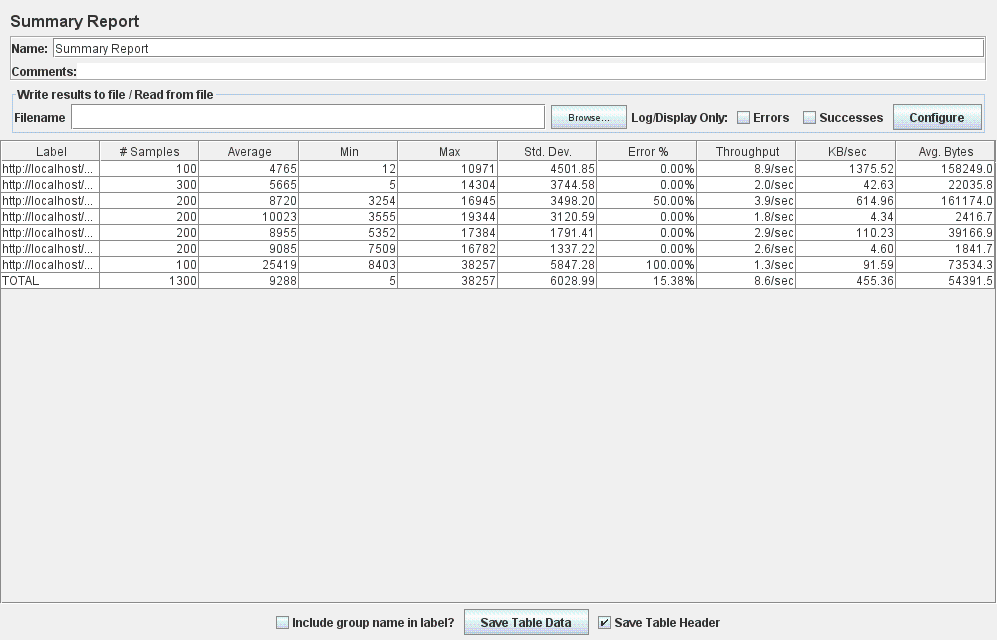
**<property name="hibernate.hbm2ddl.auto" value="update"/>**

Testul improvement-ului adus de adaugarea index-ului pe coloana status din tabela Task a fost realizat pt **Login\_Add New Developer\_Delete Old Developer\_Logout** testcase, cu un numar de 100 thread-uri concurente.

* Astfel inainte de adaugarea index-ului, eroarea era de 30.77 %:



* Dupa adaugarea index-ului, eroarea este de 15.38%:



Aceste rezultate au obtinute din JMeter. De asemenea, la rularea testcase-ului, folosind indexul adaugat, numarul query-urilor de tipul ***SELECT \* from Task t where t.status like ‘…’;*** din fiserul de MySql **\*-slow.log**a fost redus semnificativ.

* Indexul a adus performance improvement de 50% (de la 30… % la 15…%)

Next: de adaugat succesiv indecsi pt field-uri din diverse tabele (entitati) si de comparat performanta adusa (sau nu) de adaugarea acelui index. Aceste teste se vor face folosind JMeter. Daca, intr-adevar, se dovedeste o imbunatatire a performantei aduse de adaugarea indexului, acesta se va pastra, altfel nu.

Pentru test case-ul: **Login\_Skill Upgrades for developer\_Logout**, slow query-urile logate erau pentru a gasi un user dupa username si password. Astfel am adaugat index din cod java pe atributul username al entitatii User:

@NotEmpty

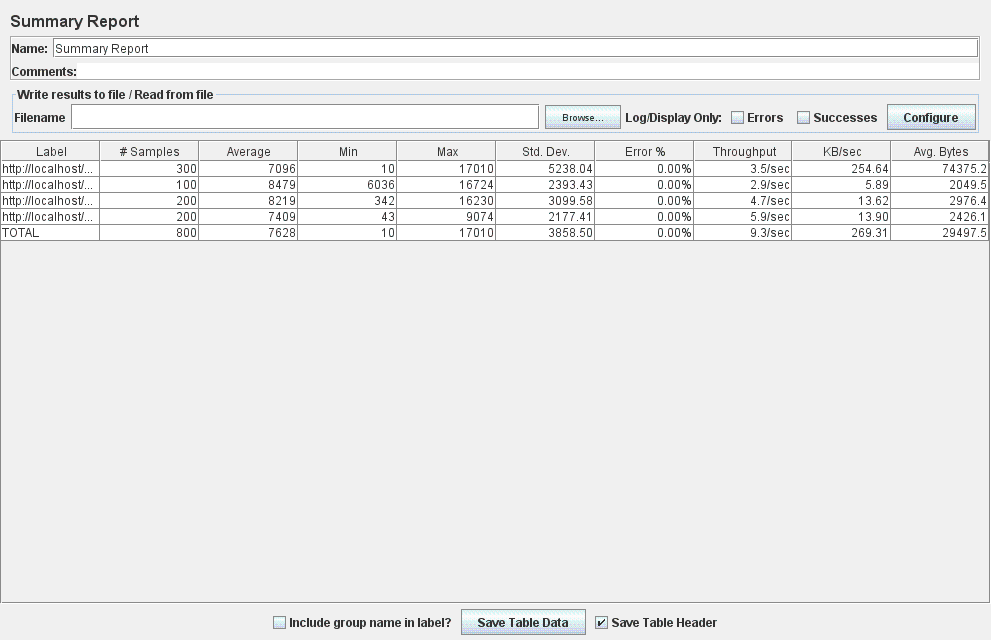
@Column(name="username")

@Index(name = "usernameIDX")

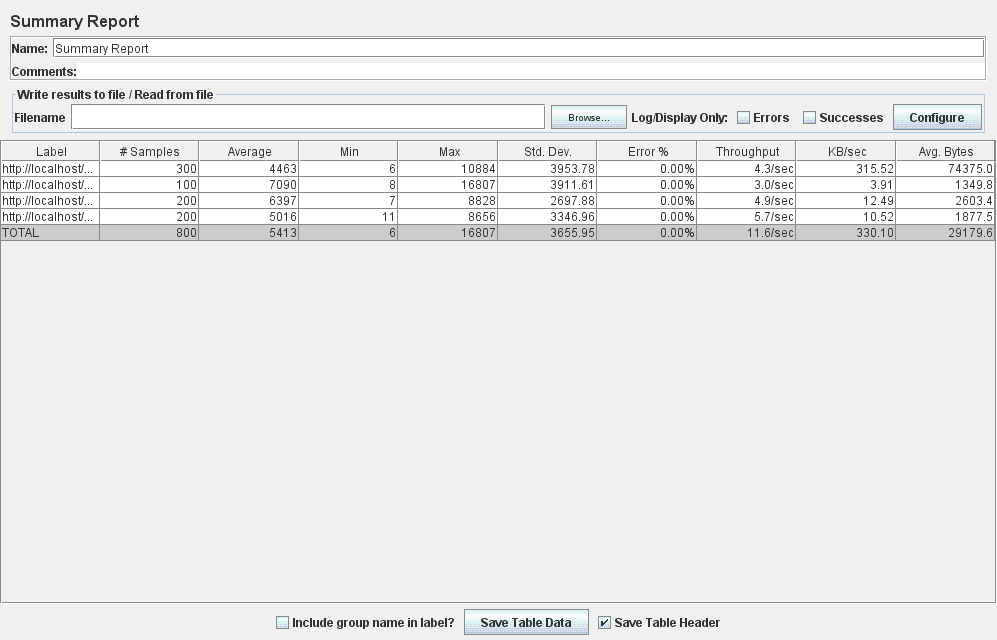
**private** String username;

Dupa adaugarea acestui index, query-ul logat ca fiind slow nu mai apare deloc in loguri. Acest lucru atesta o imbunatatire a performantei adusa de index-ul adaugat pe atributul username al entitatii User. Mai mult, comparand rezultatele oferite de JMeter, inainte si dupa adaugarea index-ului, putem concluziona urmatoarele:

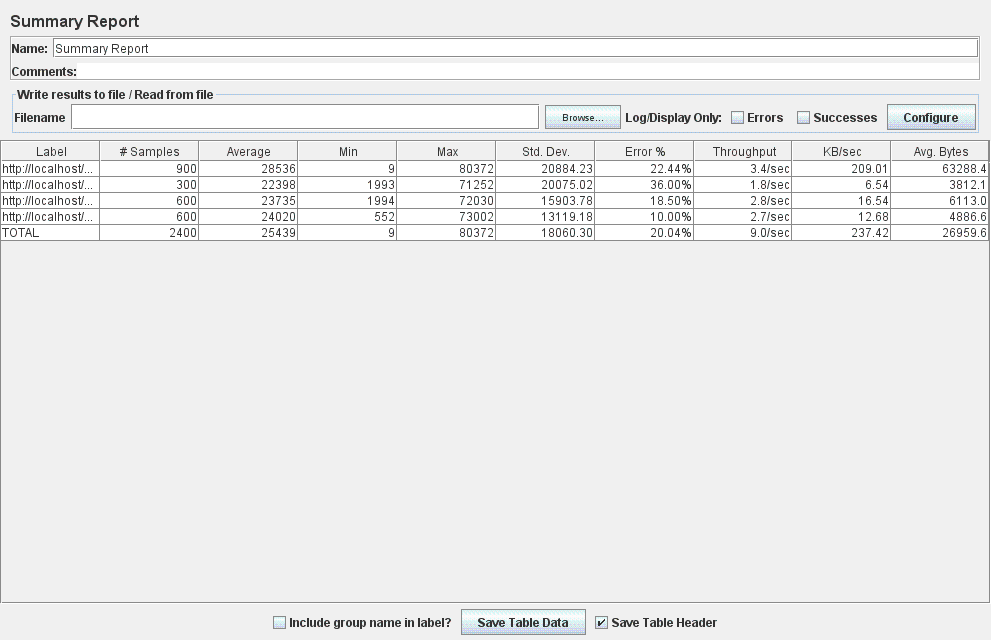
* Inainte de adaugarea index-ului, pentru un numar de 100 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, era de **0%** :



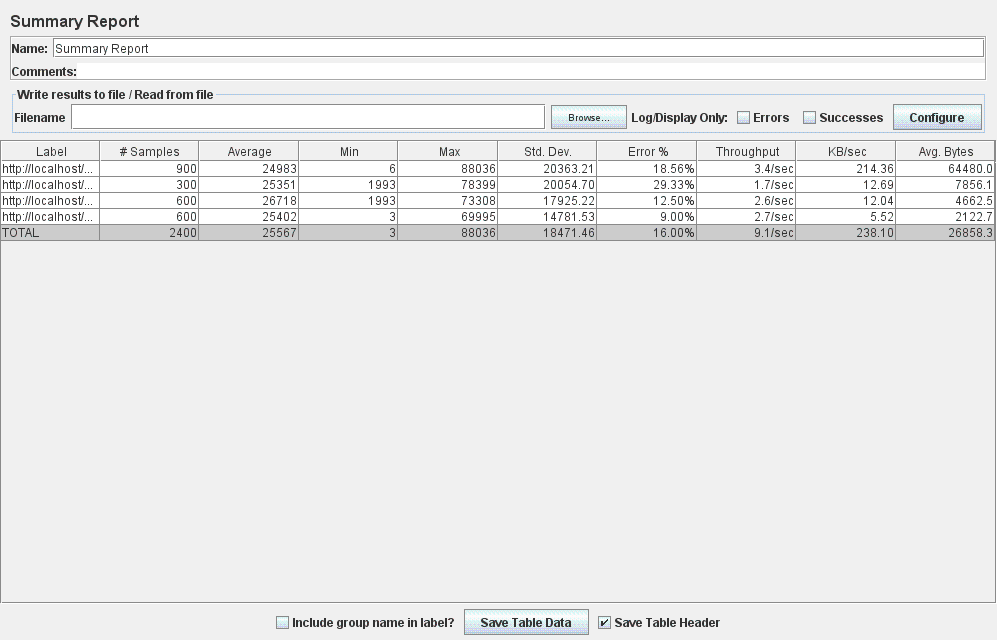
* Dupa adaugarea index-ului, pentru un numar de 100 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, este tot de **0%** :



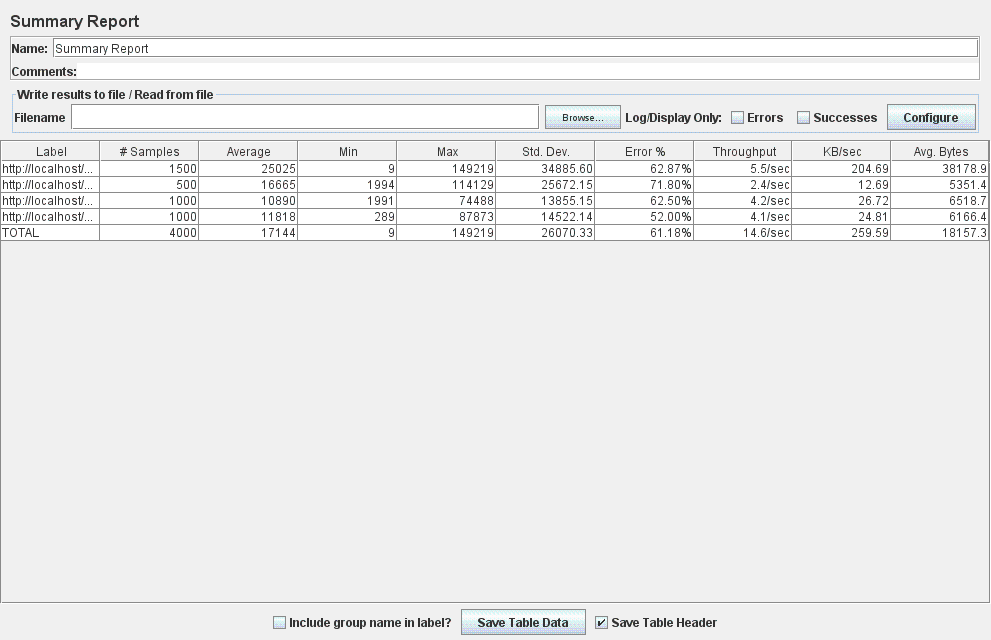
* Inainte de adaugarea index-ului, pentru un numar de 300 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, era de **20.04**% :



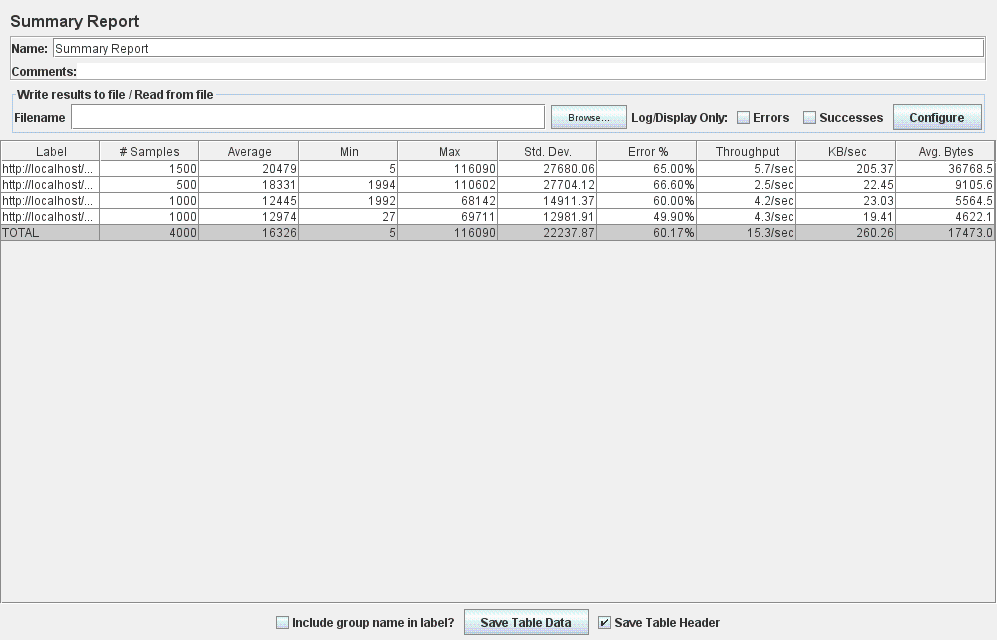
* Dupa adaugarea index-ului, pentru un numar de 300 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, este de **16.00%** :



* Inainte de adaugarea index-ului, pentru un numar de 500 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, era de **61.18**% :



* Dupa adaugarea index-ului, pentru un numar de 500 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, este de **60.17%** :



Concluziile care se pot trage din aceste rezultate:

* Performanta de 0% eroare pentru 100 utilizatori executand simultan use case-ul este pastrata si dupa adaugarea index-ului pentru username.
* In cazul a 300 utilizatori excutand simultan use case-ul, eroarea este redusa de la 20.04% la 16.00% dupa adaugara index-ului pe atributul username al entitatii User.
* Considerand 500 utilizatori excutand simultan use case-ul, eroarea este redusa de la 61.18% la 60.17% dupa adaugara index-ului pe atributul username al entitatii User.

In concluzie, in toate cele 3 cazuri, performanta a fost pastrata sau imbunatatita dupa adaugarea acestui index.

Cum query-ul initial care aparea in \*-slow.log , cautarea user-ului se facea filtrandu-se dupa atributele username si password, vom considera stergerea index-ului pt username si vom efectua aceleasi masurari dupa adaugarea unui index pe atributul password.

Astfel am sters index-ul adaugat pentru atributul username si am creat unul (tot din cod java) pe atributul password al entitatii User:

@NotEmpty

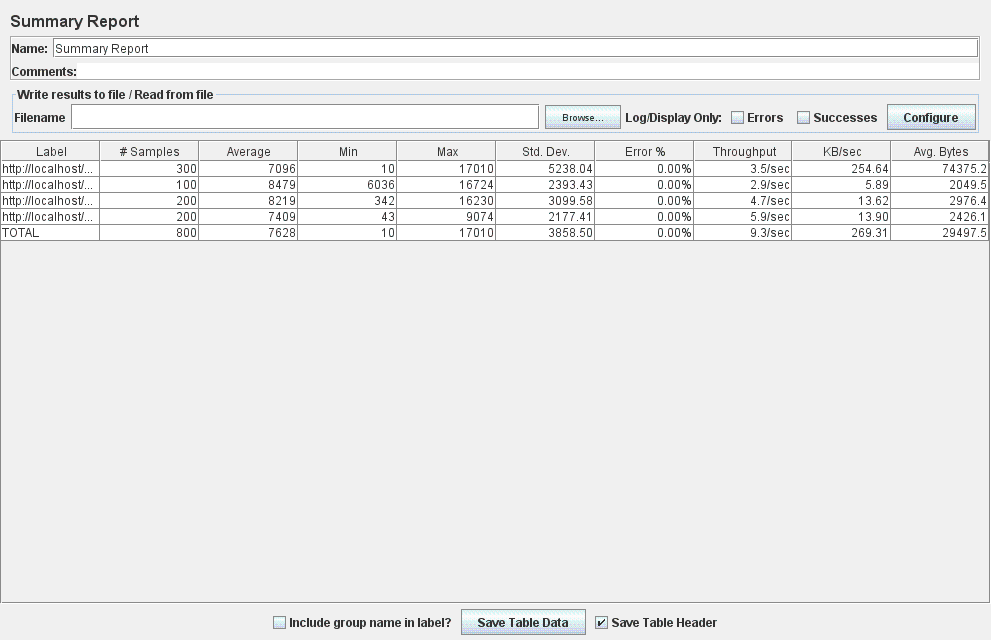
@Column(name="password")

@Index(name = "passwordIDX")

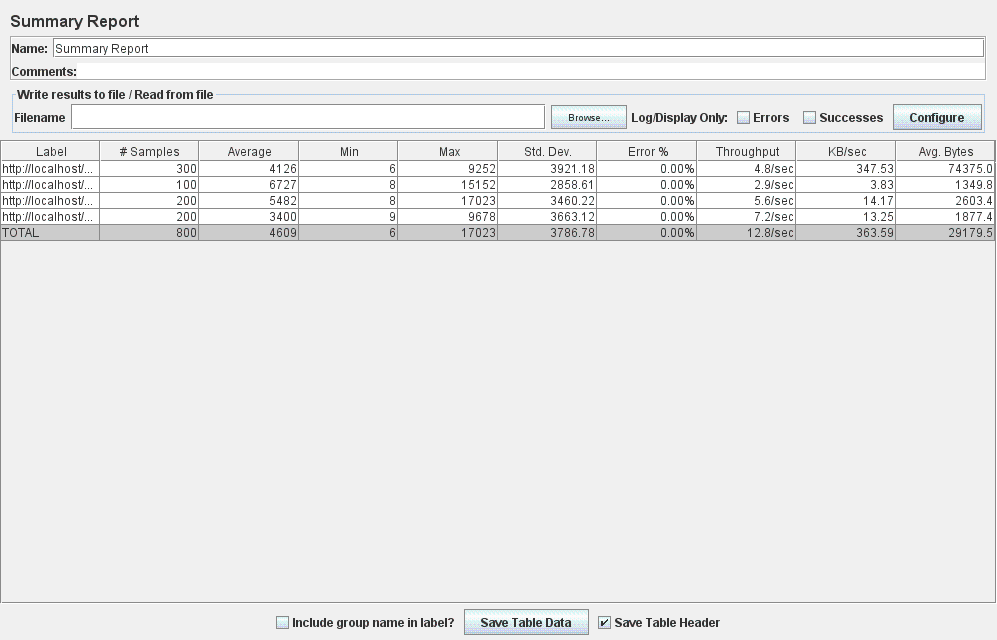
**private** String password;

Dupa adaugarea acestui index, query-ul logat ca fiind slow nu mai apare deloc in loguri. Acest lucru atesta o imbunatatire a performantei adusa de index-ul adaugat pe atributul password al entitatii User. Acest behavior s-a intalnit si la adaugarea index-ului pe atributul username. Mai mult, comparand rezultatele oferite de JMeter, inainte si dupa adaugarea index-ului, putem concluziona urmatoarele:

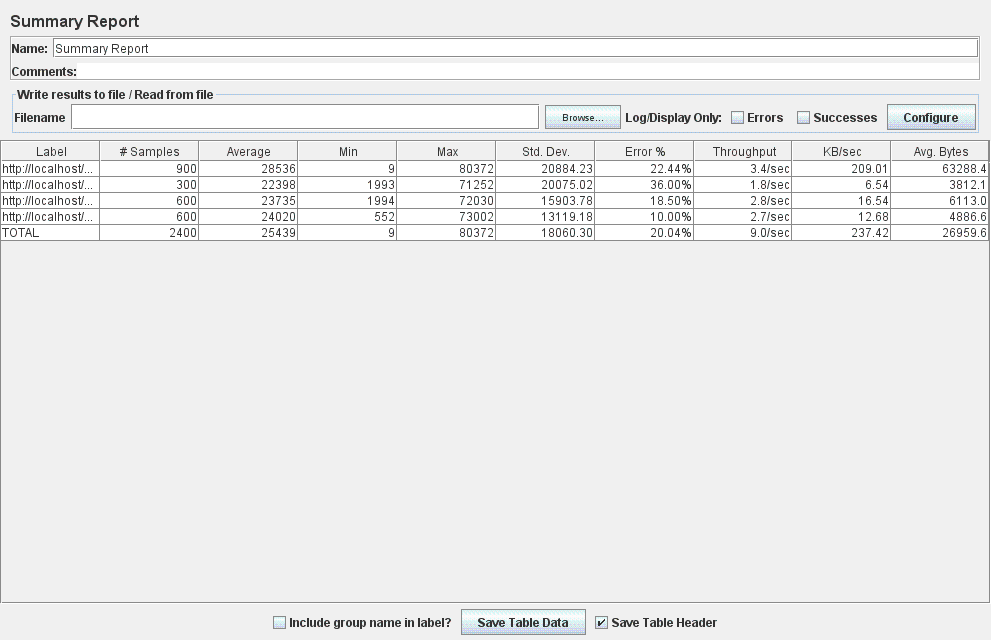
* Inainte de adaugarea index-ului, pentru un numar de 100 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, era de **0%** :



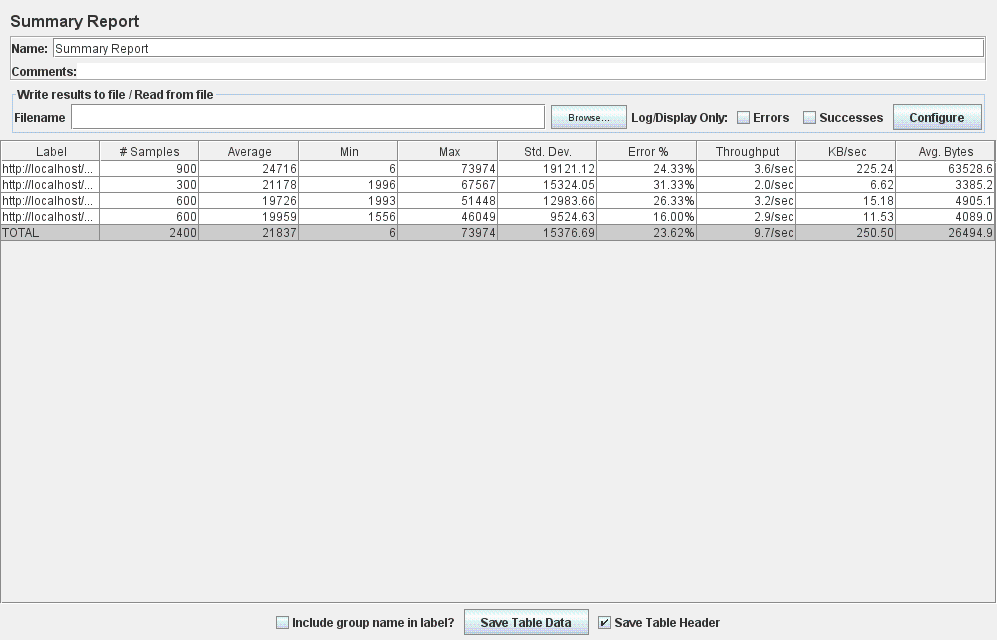
* Dupa adaugarea index-ului, pentru un numar de 100 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, este tot de **0%** :



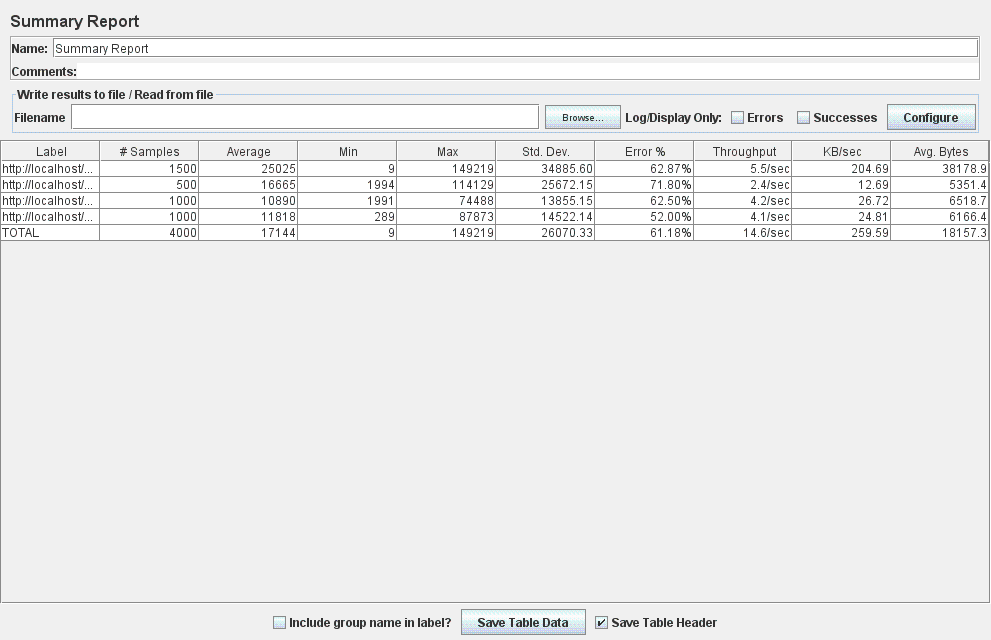
* Inainte de adaugarea index-ului, pentru un numar de 300 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, era de **20.04**% :



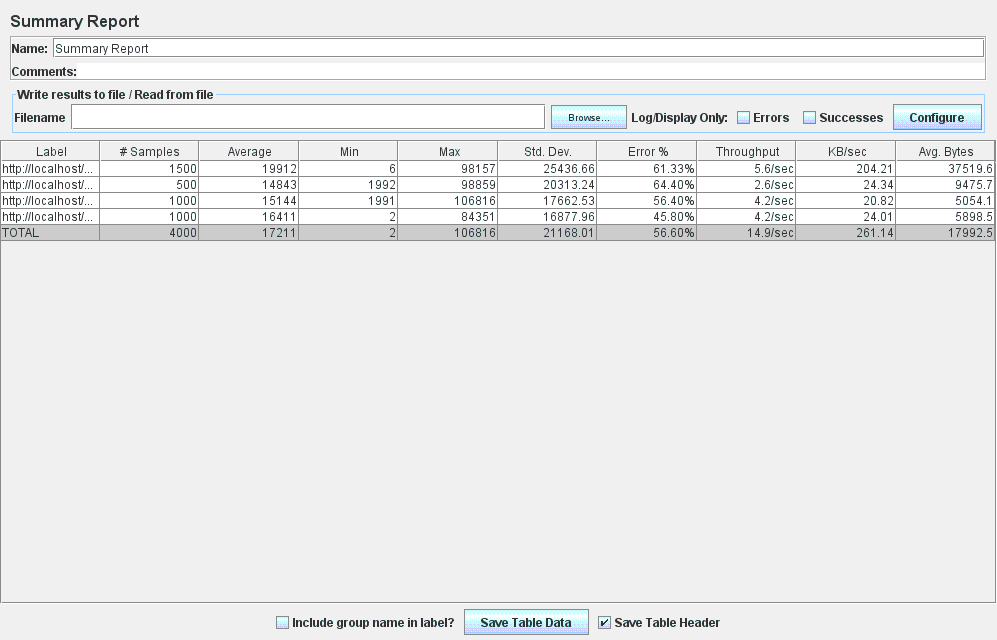
* Dupa adaugarea index-ului, pentru un numar de 300 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, este de **23.62%** :



* Inainte de adaugarea index-ului, pentru un numar de 500 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, era de **61.18**% :



* Dupa adaugarea index-ului, pentru un numar de 500 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, este de **56.60%** :



Concluziile care se pot trage din aceste rezultate:

* Performanta de 0% eroare pentru 100 utilizatori executand simultan use case-ul este pastrata si dupa adaugarea index-ului pentru password.
* In cazul a 300 utilizatori excutand simultan use case-ul, eroarea este marita de la 20.04% la 23.62% dupa adaugara index-ului pe atributul password al entitatii User.
* Considerand 500 utilizatori excutand simultan use case-ul, eroarea este redusa de la 61.18% la 56.60% dupa adaugara index-ului pe atributul password al entitatii User.

In concluzie, index-ul adaugat pe atributul password manifesta comportamente diferite pentru un numar diferit de ultizatori simultani al aceluiasi use case. Mai precis, putem deduce o imbunatatire a performantei pentru un numar de 100 si 500 thread-uri concurente, insa o degradare a acesteia la executia simultana a use case-ului de catre 300 utilizatori.

Cum query-ul initial care aparea in \*-slow.log , cautarea user-ului se facea filtrandu-se dupa atributele username si password, vom considera pastrarea index-ului pentru atributul password si vom adauga cel creat initial pentru atributul username, efectuand apoi aceleasi masurari in contextul utilizarii a doi indecsi separati pentru atributele folosite in clauza WHERE al query-ului SQL de cautare a user-ului in baza de date.

Astfel am pastrat index-ul pentru atributul password si l-am adaugat pe cel corespunzator atributului username password al entitatii User:

@NotEmpty

@Column(name="username")

@Index(name = "usernameIDX")

**private** String username;

@NotEmpty

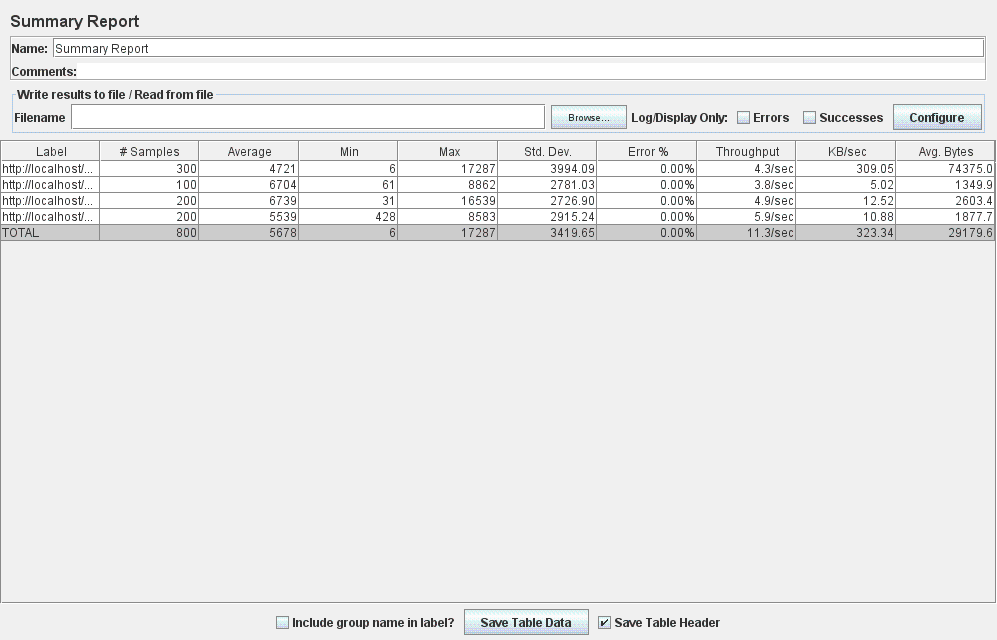
@Column(name="password")

@Index(name = "passwordIDX")

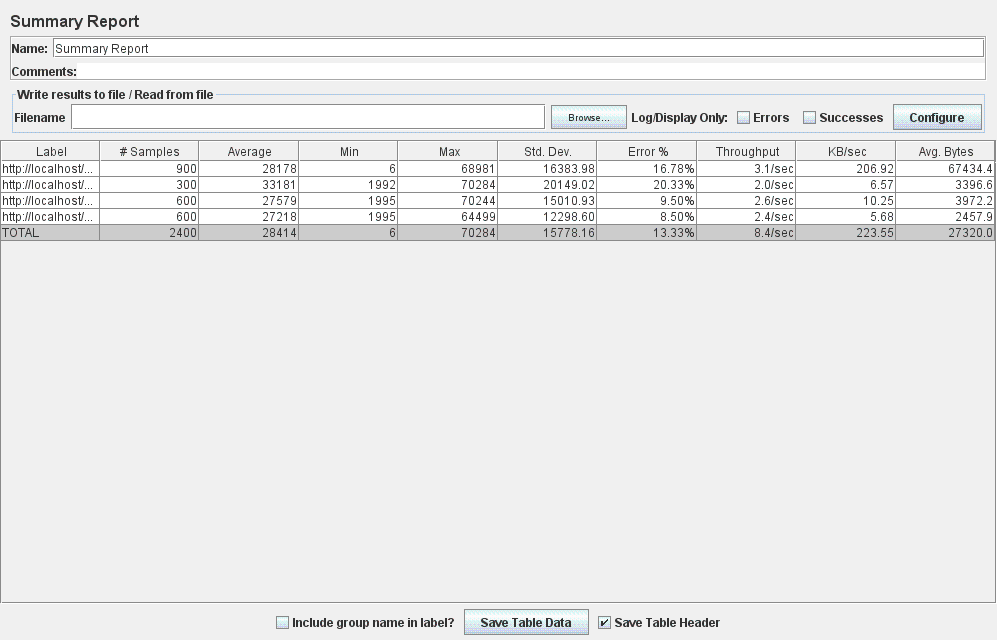
**private** String password;

La executia use case-ului folosind acesti doi indecsi, query-ul logat ca fiind slow nu mai apare deloc in loguri. Acest lucru atesta o imbunatatire a performantei adusa de indecsi adaugati pe atributele username si password al entitatii User. Acest behavior s-a intalnit si la utilizarea separata a indecsilor (initial pentru atributul username si apoi pentru password). In continuare, vom prezenta rezultatele obtinute din JMeter la folosirea a doi indecsi pentru use case-ul **Login\_Skill Upgrades for developer\_Logout**, facand apoi o comparatie a metricilor obtinute in folosirea a cate unuia dintre indecsi si respectiv al ambilor:

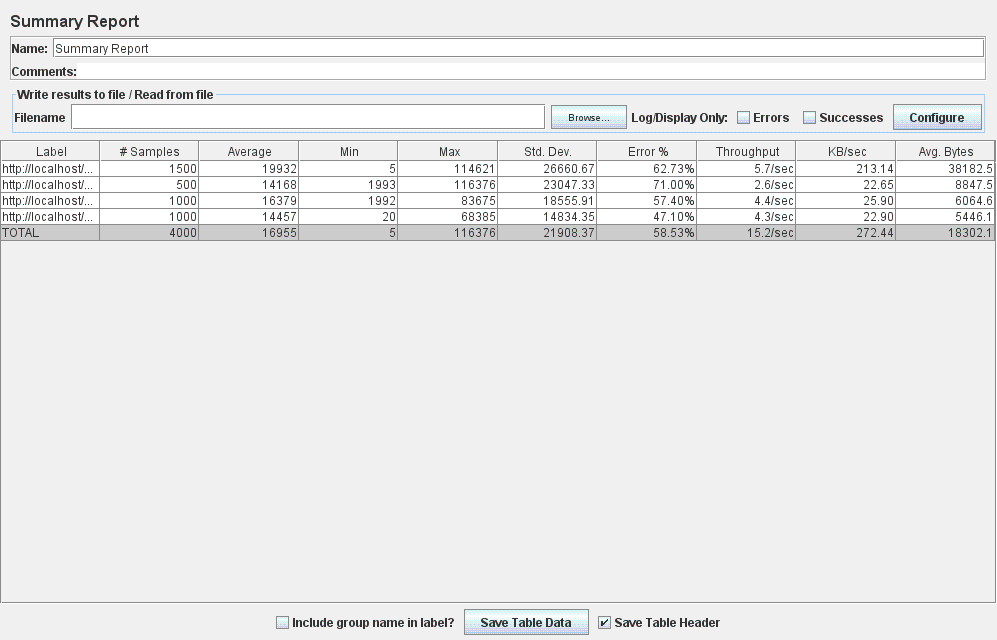
* Dupa adaugarea celor doi indecsi, pentru un numar de 100 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, este tot de **0%,** asemenea ca si in celelalte cazuri (fara index sau cu folosirea a cate unuia dintre indecsi) :



* Dupa adaugarea celor doi indecsi, pentru un numar de 300 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, este de **13.33%** :



* Dupa adaugarea celor doi indecsi, pentru un numar de 500 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, este de **58.53%** :



Concluziile care se pot trage din aceste rezultate:

* Performanta de 0% eroare pentru 100 utilizatori executand simultan use case-ul este pastrata si dupa adaugarea celor doi indecsi (pentru username si password).
* In cazul a 300 utilizatori excutand simultan use case-ul, eroare este de 13.33% dupa adaugarea celor doi indecsi (pentru username si password).
* Considerand 500 utilizatori excutand simultan use case-ul, eroarea este de 58.53% dupa adaugarea celor doi indecsi (pentru username si password).

In concluzie, index-ul adaugat pe cele doua atribute aduce o imbunatatire a performantei in toate cele trei cazuri (atat fata de neutilizarea indecsilor, cat si fata de utilizarea separata a cate unuia dintre acestia).

Next: index compus din username si password

In continuare vom sterge cei doi indecsi separati (pt username si password) si vom crea unul compus din cele doua atribute ale entitaii User (username si password), efectuand apoi aceleasi masurari in contextul utilizarii a unui index compus din atributele folosite in clauza WHERE al query-ului SQL de cautare a user-ului in baza de date.

Indexul compus din atributele username si password se specifica pe entitatea de care apartin (User) in cod java:

@Entity

@Table(name="userList", uniqueConstraints=@UniqueConstraint(columnNames={"username", "password"}))

**public** **class** User {

…

@NotEmpty

@Column(name="username")

**private** String username;

@NotEmpty

@Column(name="password")

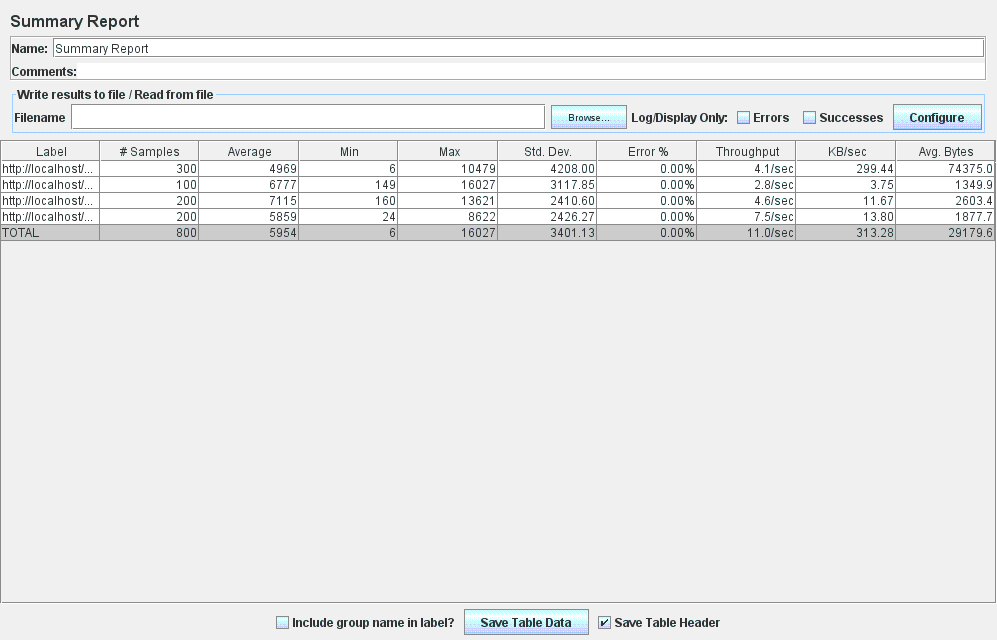
**private** String password;

…

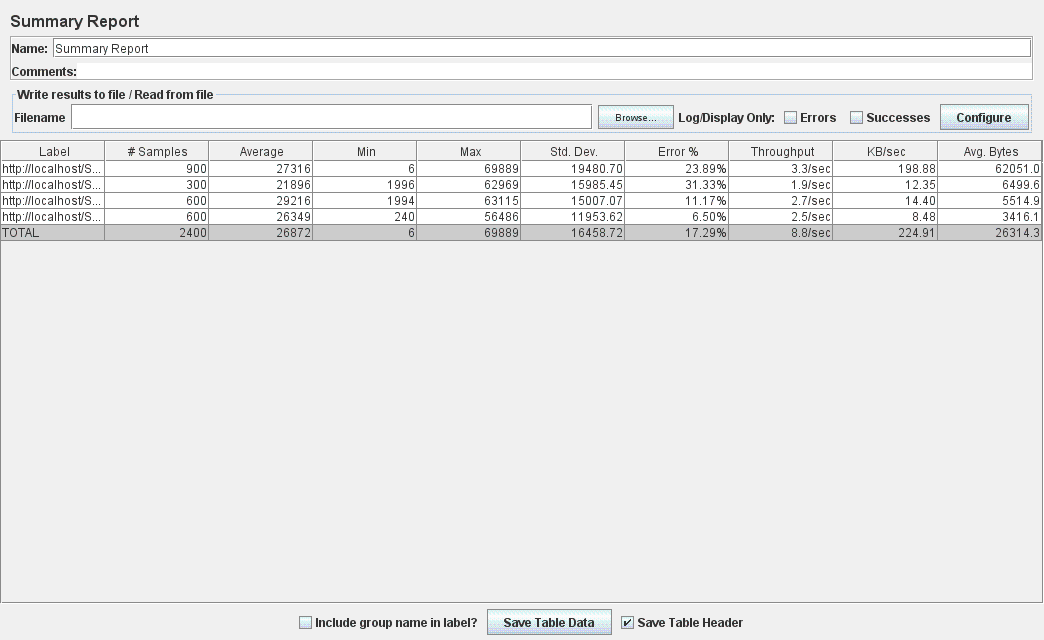
}

La executia use case-ului folosind acest index compus, query-ul logat ca fiind slow nu mai apare deloc in loguri. Acest lucru atesta o imbunatatire a performantei adusa de indexul compus din atributele username si password al entitatii User. Acest behavior s-a intalnit si in cazurile anterior preszentate: la utilizarea separata a indecsilor (initial pentru atributul username si apoi pentru password) si la utilizarea cate unui index separat pentru fiecare atribut (username si password). In continuare, vom prezenta rezultatele obtinute din JMeter la folosirea indexului compus pentru use case-ul **Login\_Skill Upgrades for developer\_Logout**, facand apoi o comparatie intre rezultatele obtinute in cazurile precedente si cel current:

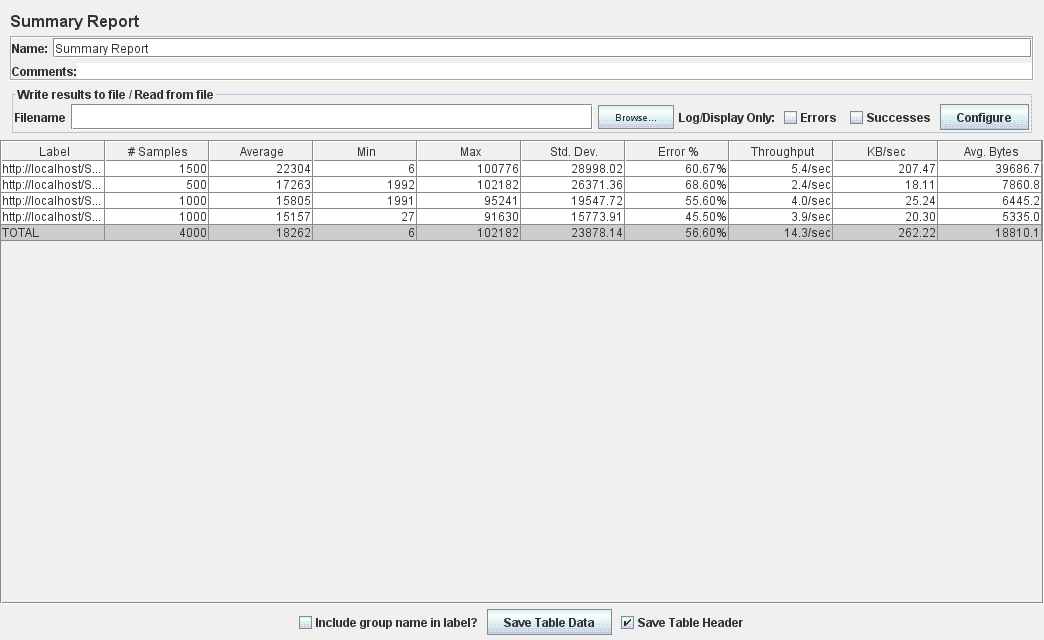
* Dupa adaugarea indexului compus, pentru un numar de 100 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, este tot de **0%,** asemenea ca si in celelalte cazuri (fara index, utilizand un singur index si respectiv, a cate doi indecsi separati pentru atributele username si password) :



* Dupa adaugarea indexului compus, pentru un numar de 300 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, este de **17.29%** :



* Dupa adaugarea indexului compus, pentru un numar de 500 thread-uri (utilizatori) concurente, eroarea rezultata din rularea test case-ului corespunzator use case-ului **Login\_Skill Upgrades for developer\_Logout**, este de **56.60%** :



Concluziile care se pot trage din aceste rezultate:

* Performanta de 0% eroare pentru 100 utilizatori executand simultan use case-ul este pastrata si dupa adaugarea indexului compus (din username si password).
* In cazul a 300 utilizatori excutand simultan use case-ul, eroarea este de 17.29% dupa adaugarea indexului compus (din username si password).
* Considerand 500 utilizatori excutand simultan use case-ul, eroarea este de 56.60% dupa adaugarea indexului compus (din username si password).

Analizand rezultatele obtinute in studiul experimental efectuat, putem concluziona faptul ca index-ul compus nu ofera o imbunatatire a performantei in toate cele trei cazuri (100, 300 si respectiv 500 de utilizatori simultani). Mai precis, pentru un numar de 300 utilizatori, se observa metrici mai bune obtinute la utilizarea unui singur index corespunzator atributului username, precum si a cate unui index separat pentru username si, respectiv, password. Pentru celelalte cazuri analizate, performanta aplicatiei este pastrata (prin mentinerea unei erori de 0% la un numar de 100 utilizatori) sau imbunatatita (prin reducerea erorii la un numar de 500 utilizatori).

Tabelul de mai jos ofera o sumarizare a erorii raportate in raportul generat de JMeter pentru toate cazurile analizate:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 100 utilizatori | 300 utilizatori | 500 utilizatori |
| Fara Index | 0% | 20.04% | 61.18% |
| Index pt username | 0% | 16.00% | 60.17% |
| Index pt password | 0% | 23.62% | 56.60% |
| Index pt username + Index pt password | 0% | 13.33% | 58.53% |
| Index compus din username si password | 0% | 17.29% | 56.60% |

Din studiul experimental legat de metodologia de alegere potrivita a indexarii pentru entitatile stocate in baza de date, putem concluziona faptul ca in majoritatea cazurilor, indexarea ofera o imbunatatire a performantei aplicatiei. Mai mult, pornind de la query-urile SQL logate ca fiind “slow”, se recomanda adaugarea indexarii pentru atributul/atributele care apar in clauza WHERE. In cazul a mai multor atribute folosite in filtrarea rezultatelor (in clauza WHERE a SELECT query-ului), performanta este maximizata in crearea unui index separat pentru fiecare atribut, sau a unuia singur (alegerea acestuia determinande-se prin rezultate experimentale). Surprinzator, crearea unui index compus din aceste atribute poate aduce “overhead”, oferind o performanta putin imbunatatita decat in cazul utilizarii a cate unui index separat pentru fiecare atribut, sau a unuia singur. Desigur, testele experimentale vor oferi rezultate particulare fiecarui use case in parte, ajutand la alegerea adecvata a indexarii entititatilor stocate in baza de date.

Pentru use case-ul studiat (**Login\_Skill Upgrades for developer\_Logout)**, in urma rezultatelor dobandite, se for folosi doi indecsi (unul pentru username si unul pentru password), ducand astfel la maximizarea performantei.

Next: index pentru alte slow query-uri logate (pt alte use case-uri)

Pentru use case-ul **Login\_Add New Developer\_Delete Old Developer\_Logout**, SQL query-ul logat ca fiind slow este :

***SELECT \* FROM Developer d WHERE d.firstName = ‘… ‘ and d.lastName =’…’;***

Astfel, tinand cont de rezultatele obtinute la use case-ul anterior(**Login\_Skill Upgrades for developer\_Logout)**, unde filtrarea rezultatelor in urma SELECT-ului se facea tot dupa doua atribute (username si password), vom alege adaugarea a doi indecsi separati pentru atributele utilizate in clauza WHERE (firstName si lastName) corespunzatoare entitatii Developer.

@Column(name="firstName")

@Index(name = "firstNameIDX")

**private** String firstName;

@Column(name="lastName")

@Index(name = "lastNameIDX")

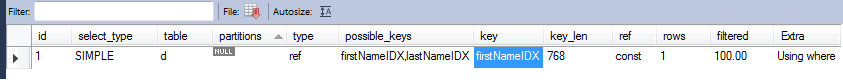
**private** String lastName;

Performanta adusa de acesti doi indecsi adaugati s-a observat din logurile de MySQL pentru slow query-uri. Astfel, inainte de adaugarea indecsilor, clauza SELECT (SELECT \* FROM Developer d WHERE d.firstName = ‘… ‘ and d.lastName =’…’;) avea un numar de 240 de aparitii in fisierul de loguri slow. Dupa crearea acestor doi indecsi, executand use case-ul in aceleasi conditii, query-ul nu mai aparea deloc in logul de slow queries.

Mai mult, pentru validarea aplicarii corecte a celor doi indecsi in momentul executiei query-ului SQL, am executat comanda:

***EXPLAIN EXTENDED SELECT \* FROM Developer d WHERE d.firstName = ‘…’ and d.lastName = ‘…’;***

Rezultatul obtinut sub forma tabelara atesta recunoasterea acestor doi indecsi si utilizarea lor in clause SELECT:



---------------------------------------------------------------------------------------------------------------------

**Caching**

Next step in the dissertation thesis research is adding **Caching** for the SCRUM web application and then evaluate the performance improvement obtained.

First: **study** different caching methodologies and approaches, **compare** them and establish the one(s) appropriate for the current application.

Then **implement** the caching strategy established and **assess** the performance enhancements gathered.

Consider another suitable caching methodology, implement it and test the results obtained.

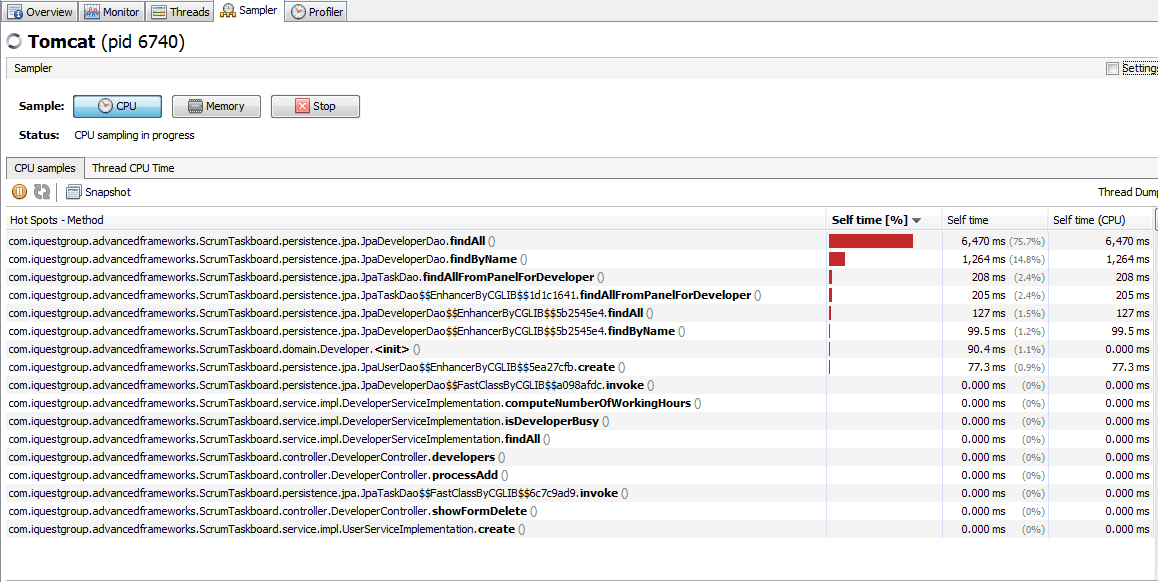
Finally, consider also developing a combination of cache strategies (if appropriate) and evaluate the performance metrics (using JMeter and VisualVM application performance monitoring tools).

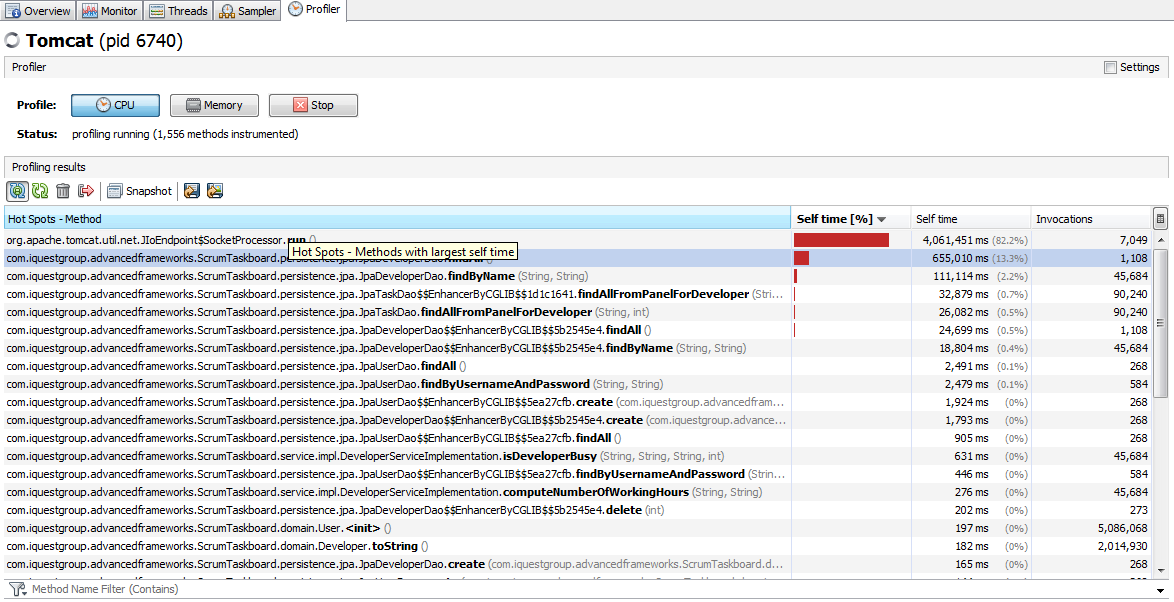
Caching:

* **Spring eh-cache** + **Hibernate eh-cache**
* study and document comparison between Spring & Hibernate cache
* implement caching using **Spring eh-cache for a service method** (from Service layer) and test the performance obtained
* implement caching using **Hibernate eh-cache for a Java entity class** (from Model layer) and test the performance obtained
* implement caching using **both Spring eh-cache and Hibernate eh-cache** and test the performance obtained

**Caching with Spring eh-cache**

For the **Login\_Add New Developer\_Delete Old Developer\_Logout** testcase, the conclusions gathered after analyzing VisualVM results:





* When profiling/sampling different application use-cases, the main CPU bottleneck is caused by methods from the persistence layer (when querying the database) => further improvements should be considered on the database level
* After implementing the appropriate improvements for the persistence layer, **caching of the most CPU consuming functionality was also considered**
* Furthermore, from the two images (obtained after experiencing sampling and profiler activities), it can be determined that most of the CPU time is spent on *JpaDeveloperDao.findAll ()* method that consume the most CPU when the considered use case scenario is performed
* Therefore, **Spring eh-cache** was configured and implemented for the service method which calls this persistence-layer method: *DeveloperServiceImplementation.findAll()*
* The main steps in the implementation of Spring eh-cache are:
  + Add the required dependencies in ***pom.xml***:

<properties>

<!-- Eh-cache -->

<ehcache.version>2.10.0</ehcache.version>

</properties>

<dependencies>

<dependency>

<groupId>net.sf.ehcache</groupId>

<artifactId>ehcache</artifactId>

<version>${ehcache.version}</version>

</dependency>

<!-- Spring caching framework inside this -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring-framework.version}</version>

</dependency>

<!-- Support for Ehcache and others -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>${spring-framework.version}</version>

</dependency>

</dependencies>

* + Create ***ehcache.xml*** file under *resources* directory in order to tell Eh-cache how and where to cache the data:

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"

updateCheck="true"

monitoring="autodetect"

dynamicConfig="true">

<cache name="findAllDevelopersCache"

maxEntriesLocalHeap="1000"

maxEntriesLocalDisk="1000"

eternal="false"

timeToIdleSeconds="300" timeToLiveSeconds="600"

memoryStoreEvictionPolicy="LFU"

transactionalMode="off">

<persistence strategy="localTempSwap" />

</cache>

</ehcache>

* + Add ***@Cacheable*** annotation on the method you want to cache (***findAll()*** from ***DeveloperServiceImplementation.java*** class) :

@Cacheable("findAllDevelopersCache")

public List<Developer> findAll() {

…

}

* + Configure and enable caching inside the Spring configuration file (***application-config.xml***) :

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

**xmlns:p="http://www.springframework.org/schema/p"**

...

**xmlns:cache="http://www.springframework.org/schema/cache"**

xsi:schemaLocation="...

**http://www.springframework.org/schema/cache**

**http://www.springframework.org/schema/cache/spring-cache.xsd** ">

**<cache:annotation-driven />**

<bean id="**cacheManager**" class="org.springframework.cache.ehcache.EhCacheCacheManager"

p:cacheManager-ref="ehcache" />

<bean id="**ehcache**"

class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"

p:configLocation="classpath:ehcache.xml" p:shared="true" />

</beans>

* An aspect that has to be taken into account when introducing caching to a web application concerns the possible **data inconsistencies** that might occur. More precisely, performing updates on the data that is being cached, offers misleading information to the end user. For the particular cache implemented for the *findAll()* method from ***DeveloperServiceImplementation.java*** class, in case the Scrum Master performs some modification on the list of developers (by adding or removing developers) will bring some inconsistencies and even errors when using the application:
  + If new developers were added to the application, they won’t be displayed in the web page until the cache has expired (after 10 mins, as configured for this cache region in ehcache.xml file).
  + If developers were removed from the application while they are still being cached, an error is thrown because the database search is performed with a NULL entity.

These issues were solved by manually invalidating the cache after performing updates on the developer list. This is performed explicitly through Java code by retrieving the specific cache (determined by its name) from a new CacheManager instance and invoking the clearAll() method:

private void invalidateCache(String cacheToClear) {

CacheManager cacheManager = CacheManager.create();

Ehcache ehcache = cacheManager.getEhcache(cacheToClear);

ehcache.removeAll();

}

This method is called after operating (on successful addition and removal) on developers data, given the cache region’s name to be invalidated as parameter:

public void delete(int developerId) {

try {

... // perform developer removal

invalidateCache(FIND\_ALL\_DEVELOPERS\_CACHE\_NAME);

} catch (...) {

...

}

}

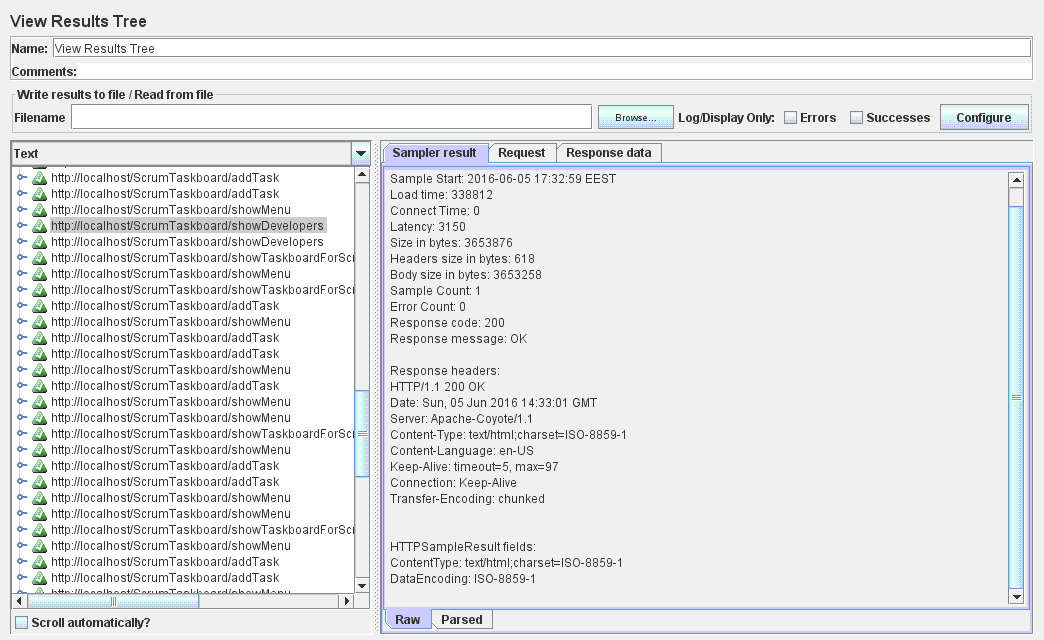
The next step after adding caching using Spring eh-cache is to test the performance improvements obtained. The **Login\_Add New Developer\_Delete Old Developer\_Logout** testcase is not a suitable case scenario to assess the performance enhancements offered by Spring eh-cache because it consists of actions that perform updates on the cached data (list of developers): additions and removal of developers from the database. As we have implemented caching for the list of developers, executing this use case (or any other one that performs manipulations of the data being caches), would cause data inconsistencies and even errors (as described in a previous paragraph). Therefore, we may conclude this scenario is not appropriate to test the performance improvements obtained after implementing Spring eh-caching. These performance considerations has to be assessed on behalf of a use case scenario through which the developers are retrieved several times from the database. In this context, the usage of the cached list of developers will benefit from the Spring eh-cache implemented and therefore performance metrics will show numeric results.

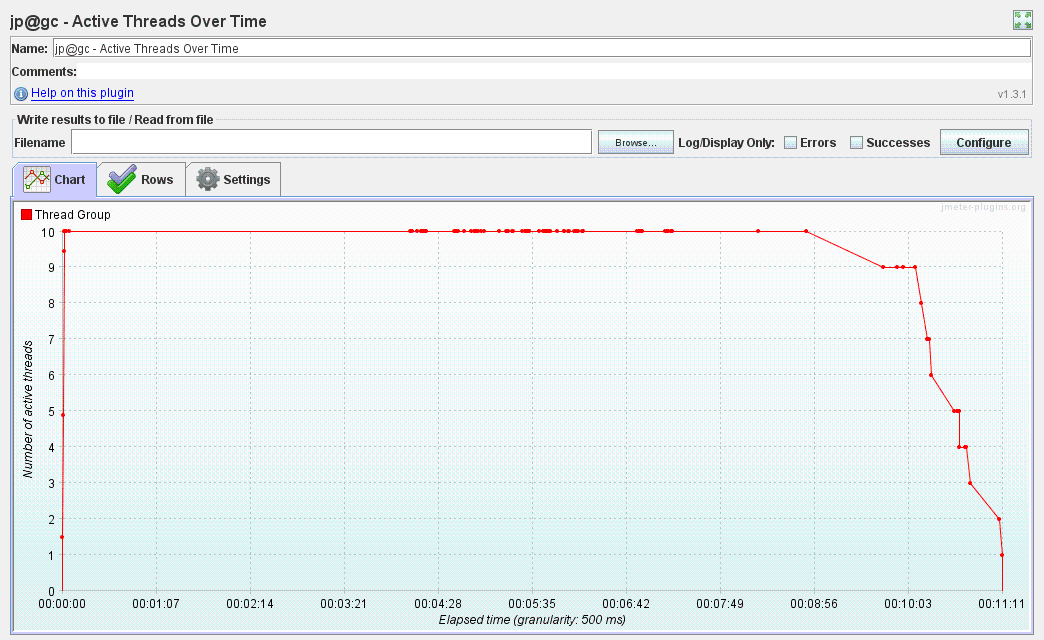
The use case created for evaluating this concern is **Login\_Visualize developers\_Visualize taskboard\_Add task\_Visualize developers\_Logout**, which is performed by an admin user (say Scrum Master) who logins into the application, visualizes all developers, then visualizes the taskboard, adds a new task (together with the required information for it), visits again the list of developers and the logs out from the application.

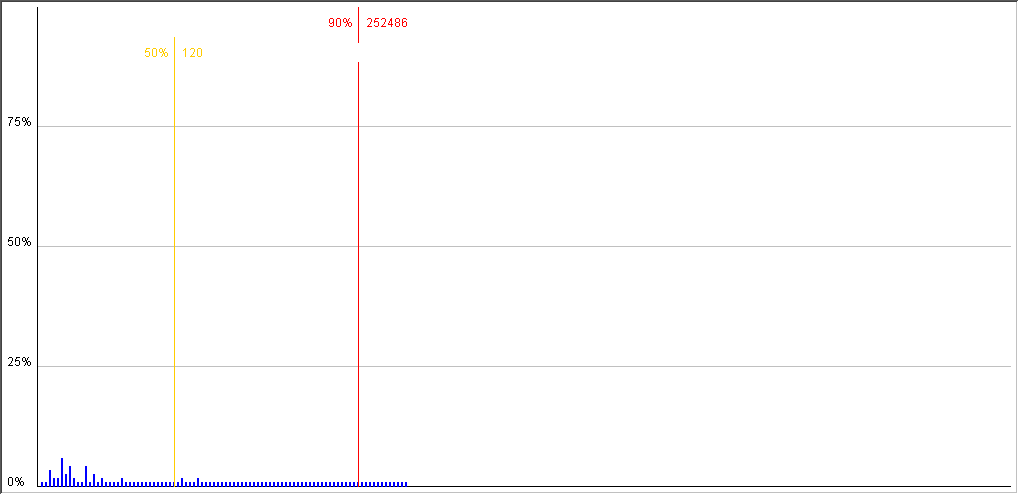
In order to evaluate the performance enhancements obtained through a series of metrics, JMeter tool will be used to simulate the execution of this use case for many concurrent users retrieving the developers. The desired behavior of the implemented cache is to access the database only once (on first call) and then retrieve the developers’ information from the cache for the successive calls of the find developers functionality.

The **Login\_Visualize developers\_Visualize taskboard\_Add task\_Visualize developers\_Logout** testcase was executed from JMeter ***before*** implementing the Spring eh-caching in order to collect the performance measurements results. In order to see considerable performance improvements offered by the Spring eh-caching implemented, the data being cached (list of developers) has to be of *large amount*. Therefore, all performance tests were executed in the context of *1000 developers* existent in the database. However, as this list is retrieved twice for each thread in the selected use case scenario (**Login\_Visualize developers\_Visualize taskboard\_Add task\_Visualize developers\_Logout**), the simulated number of users (threads) from JMeter *cannot* be very large (like hundreds or thousands) because this *consumes all of the CPU’s activity and takes a large amount of time* to complete. As a result, the tested use case was executed in the context of 10 and 50 concurrent users and the results are presented as follows:

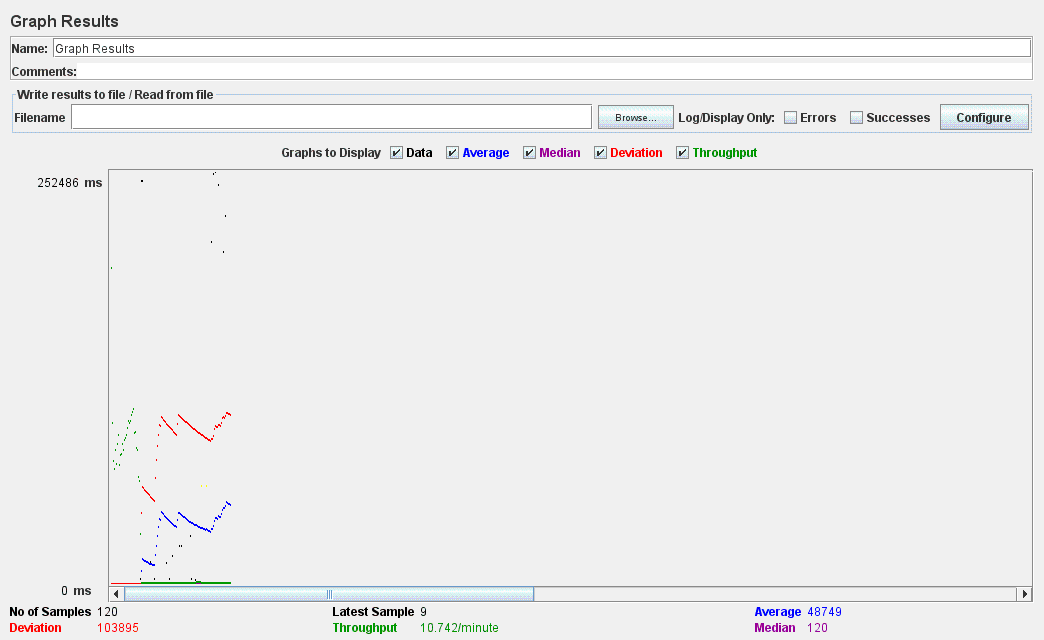
* For **10 concurrent users**:

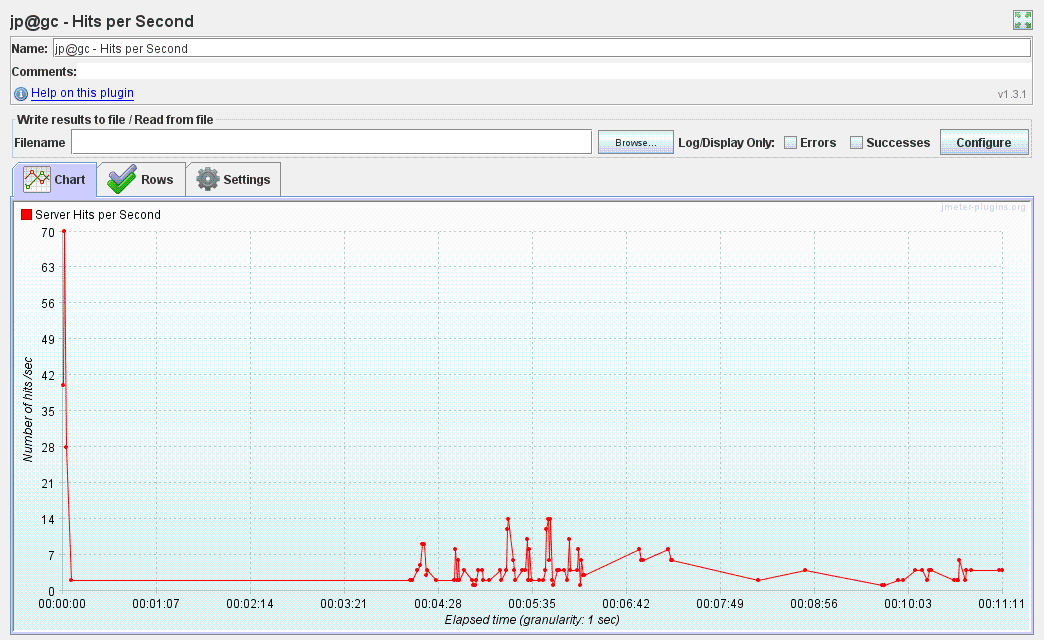


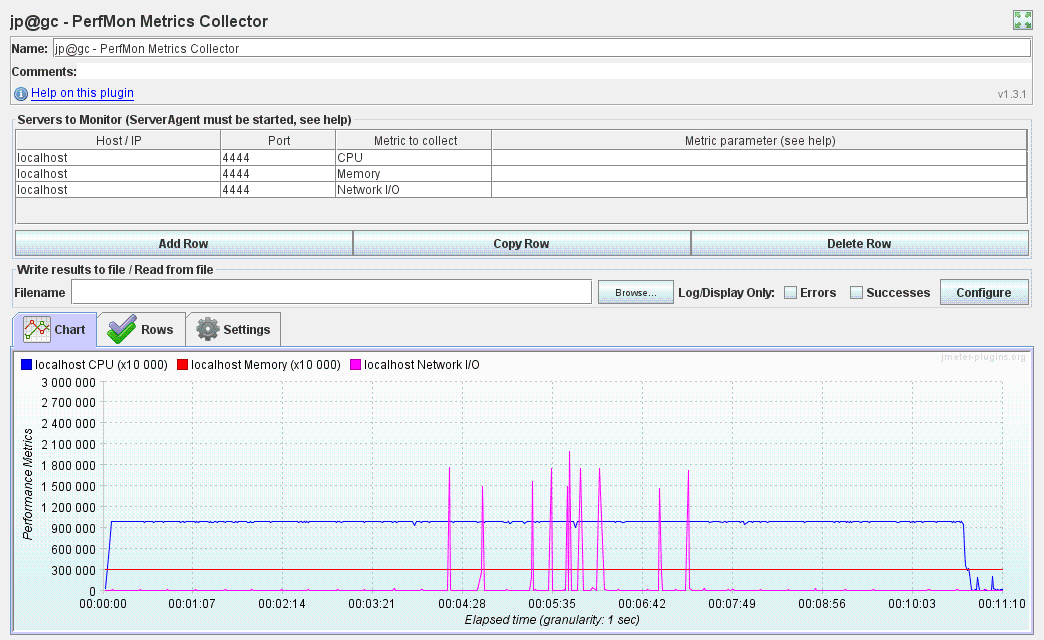


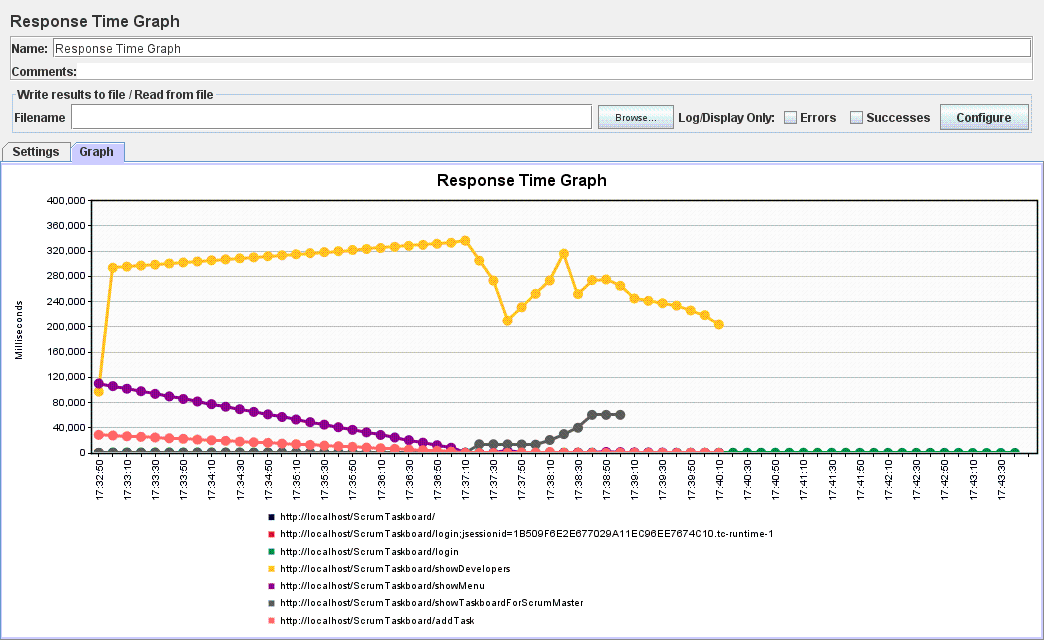


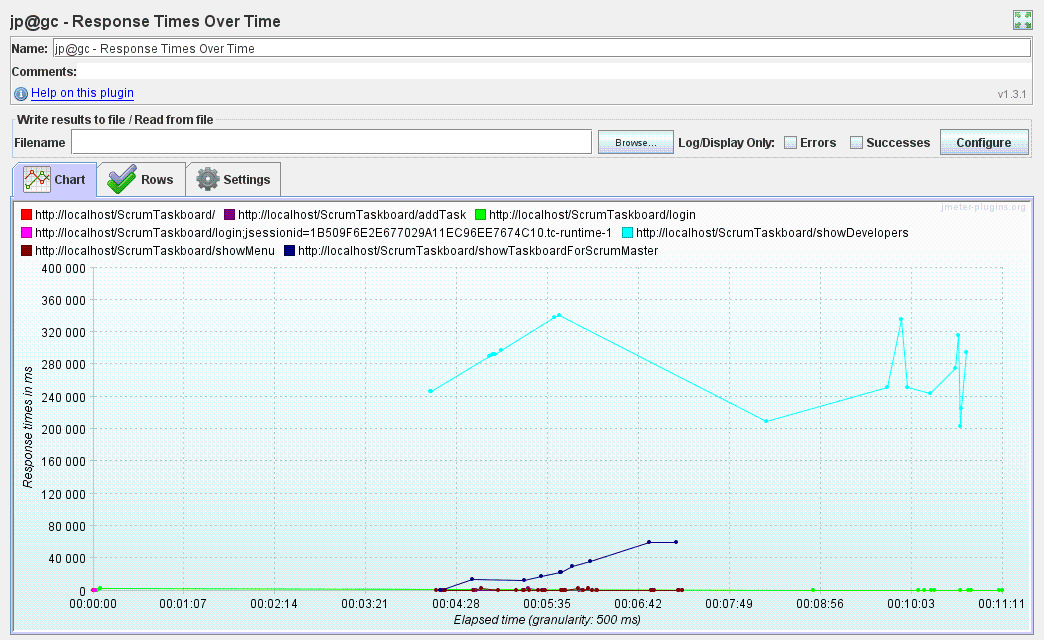
Distribution Graph (alpha)

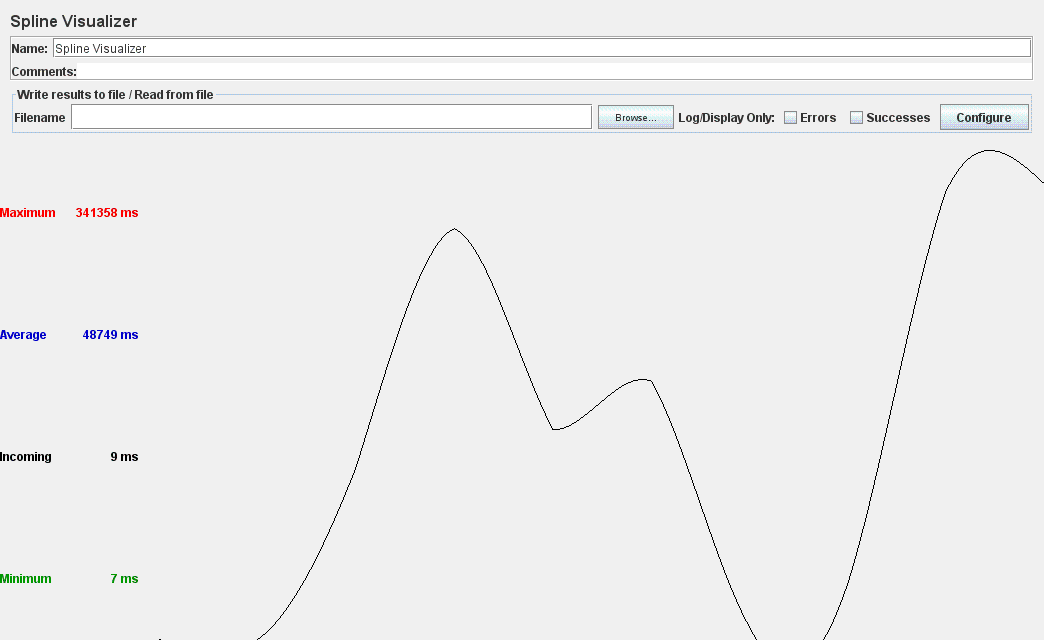


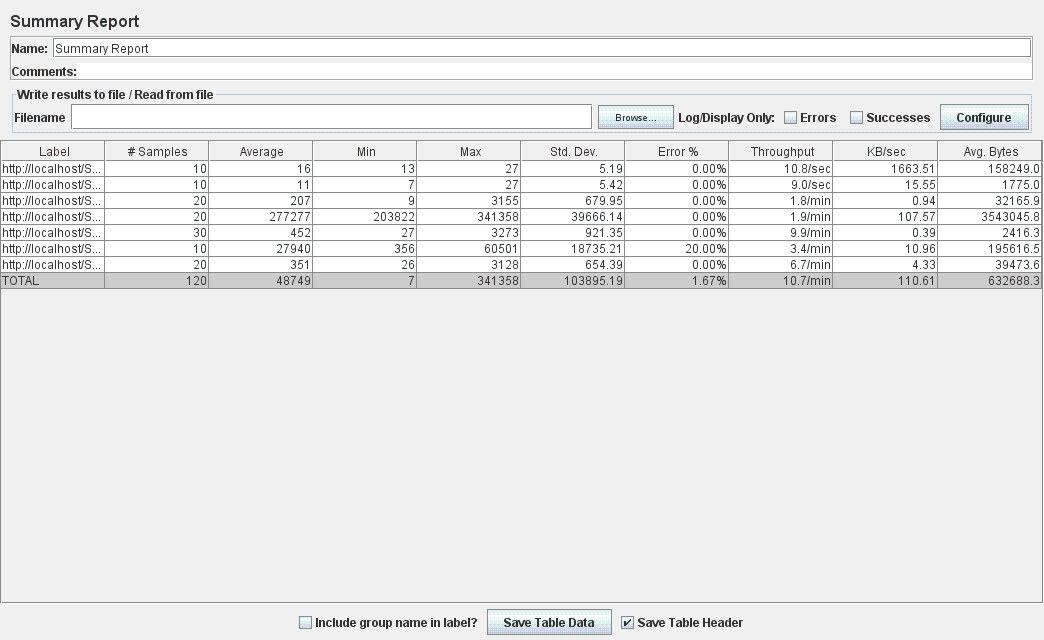


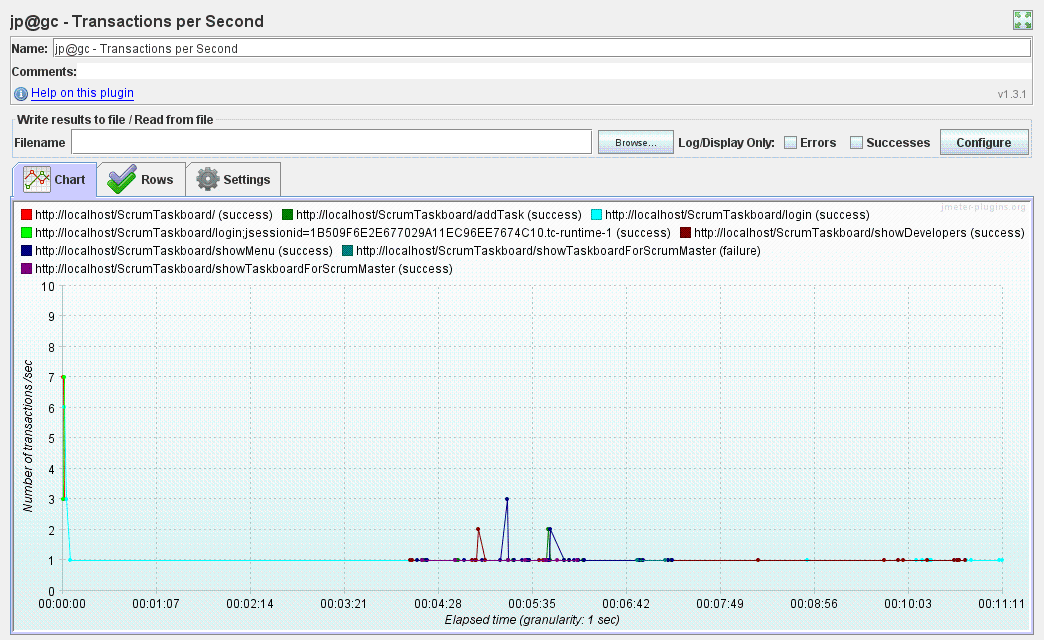


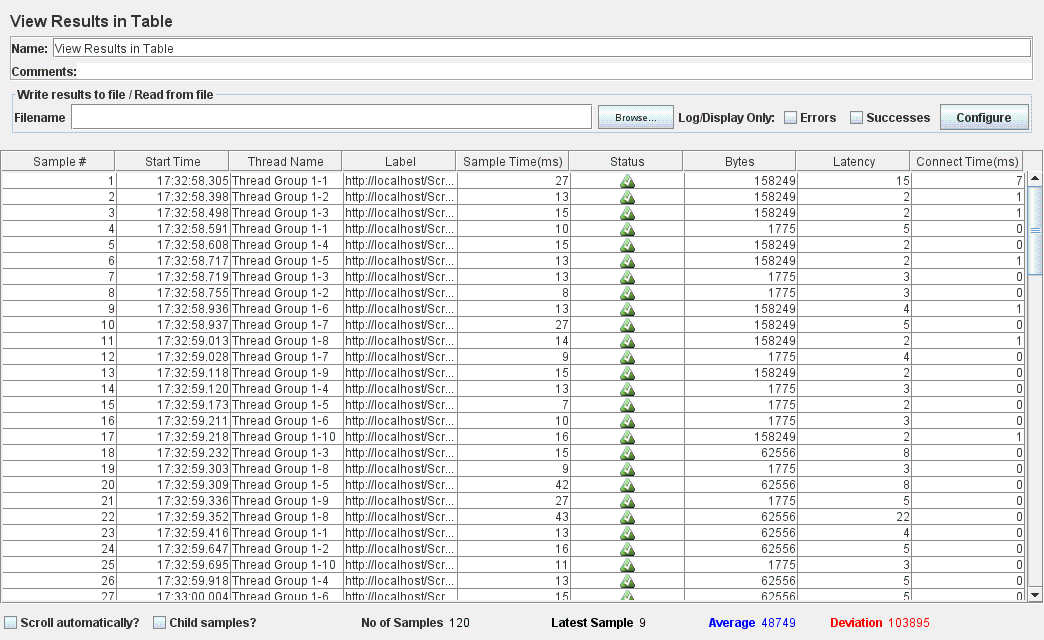




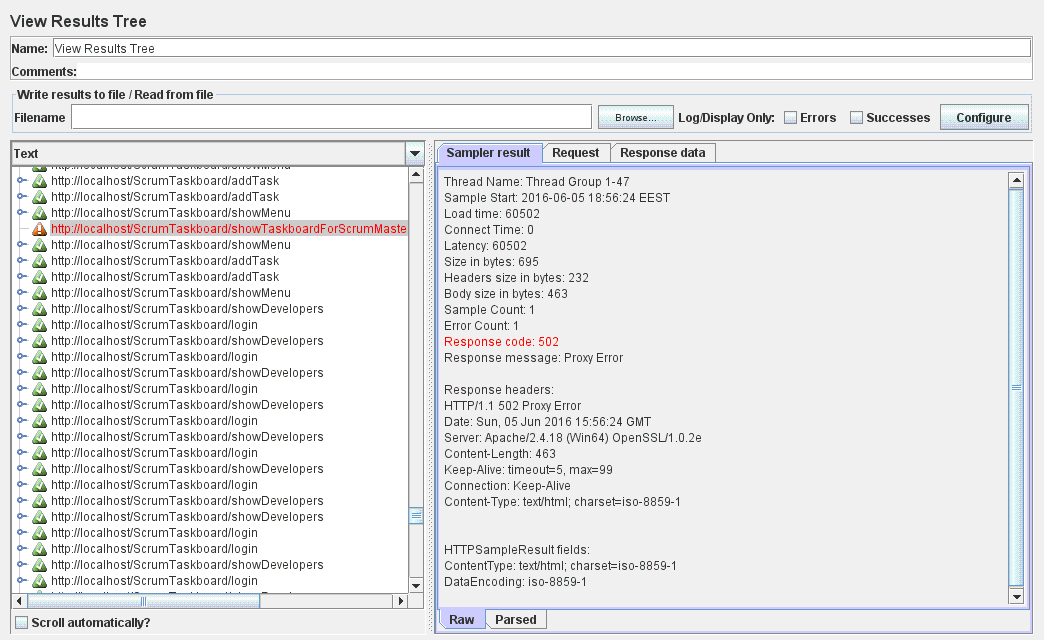


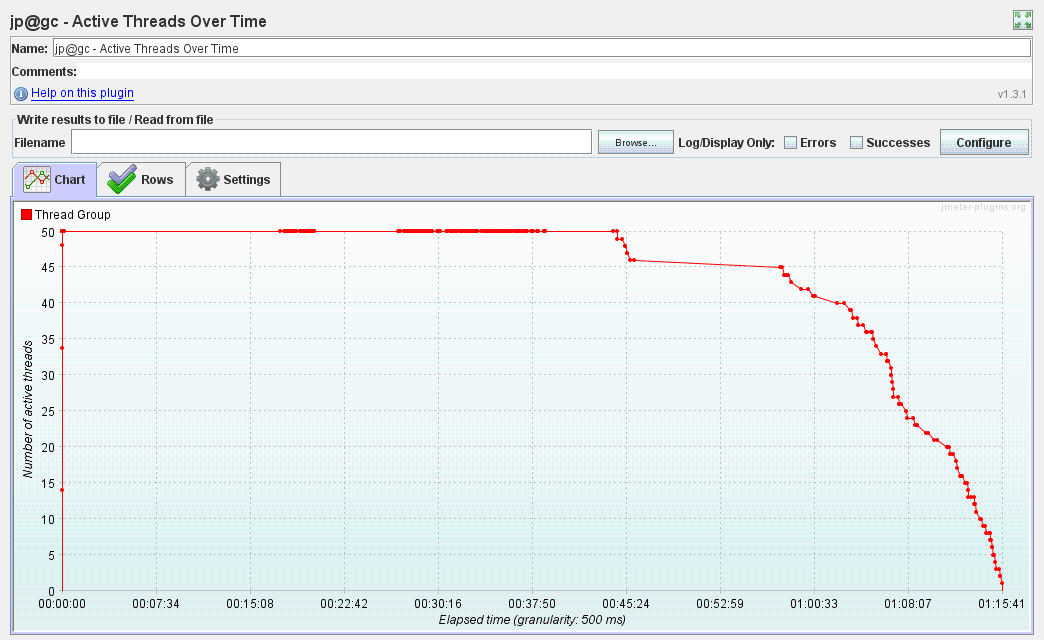


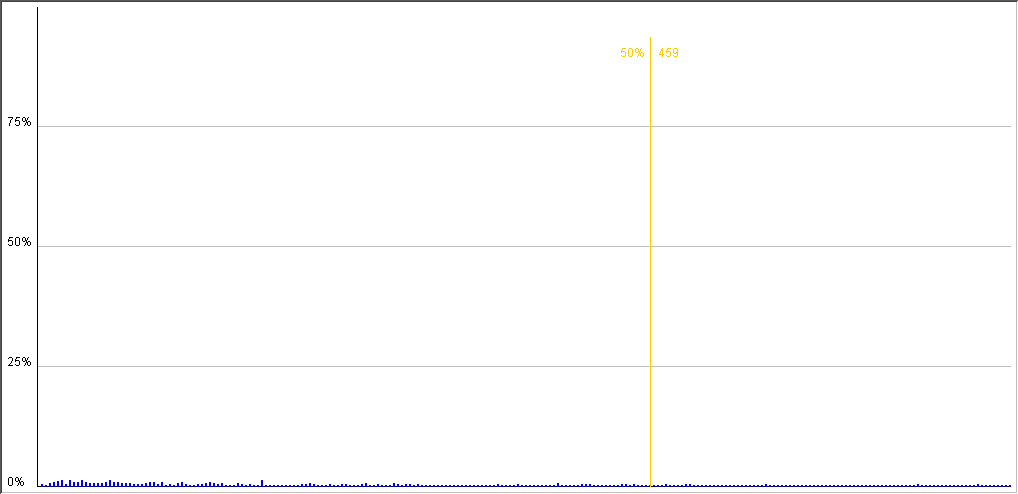




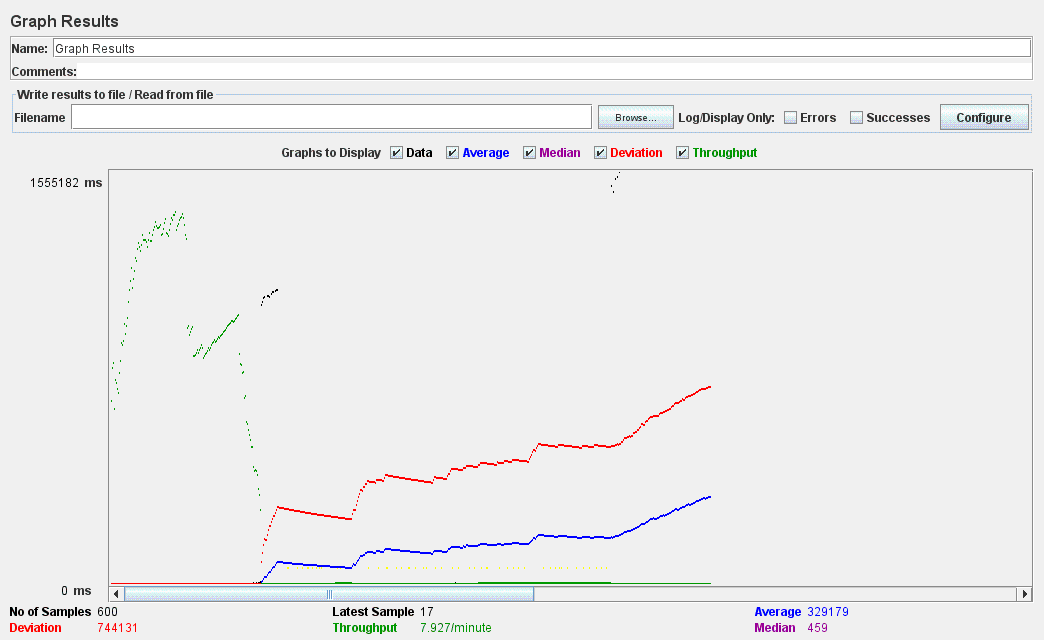
* For **50 concurrent users**:

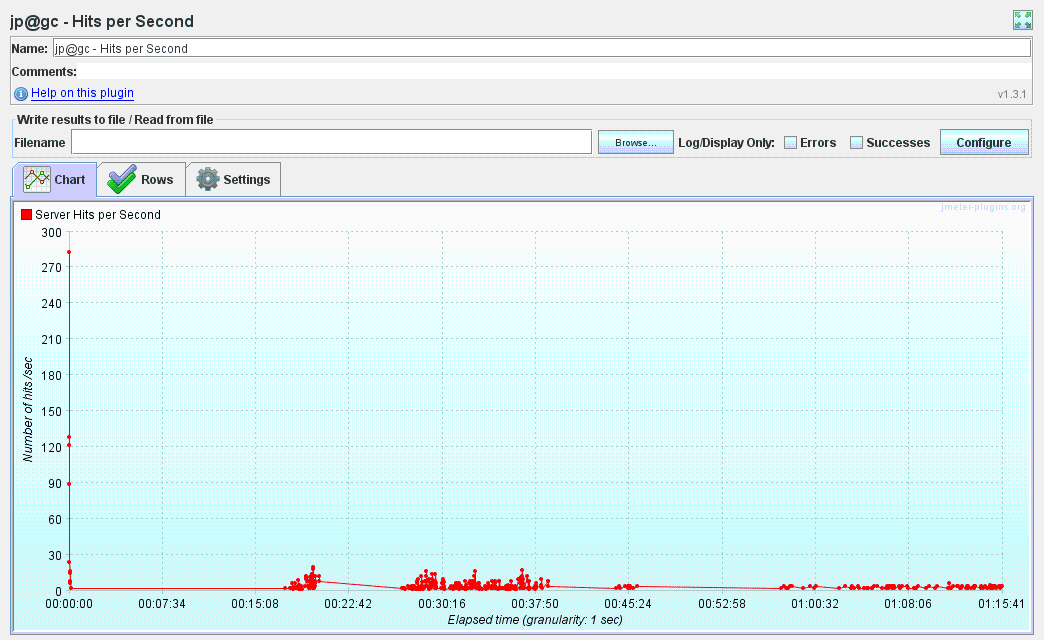


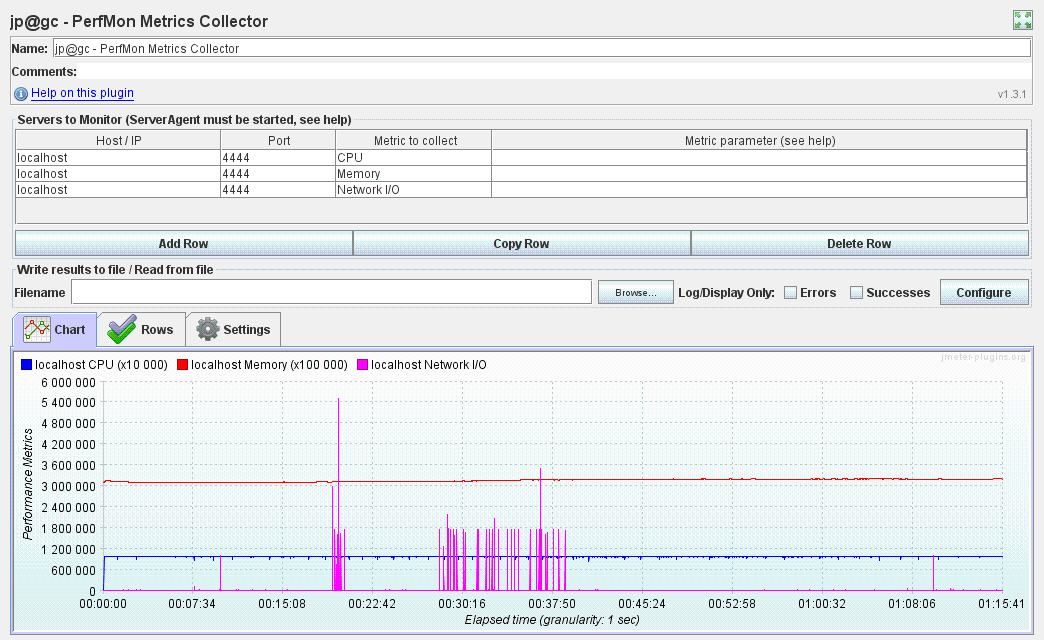


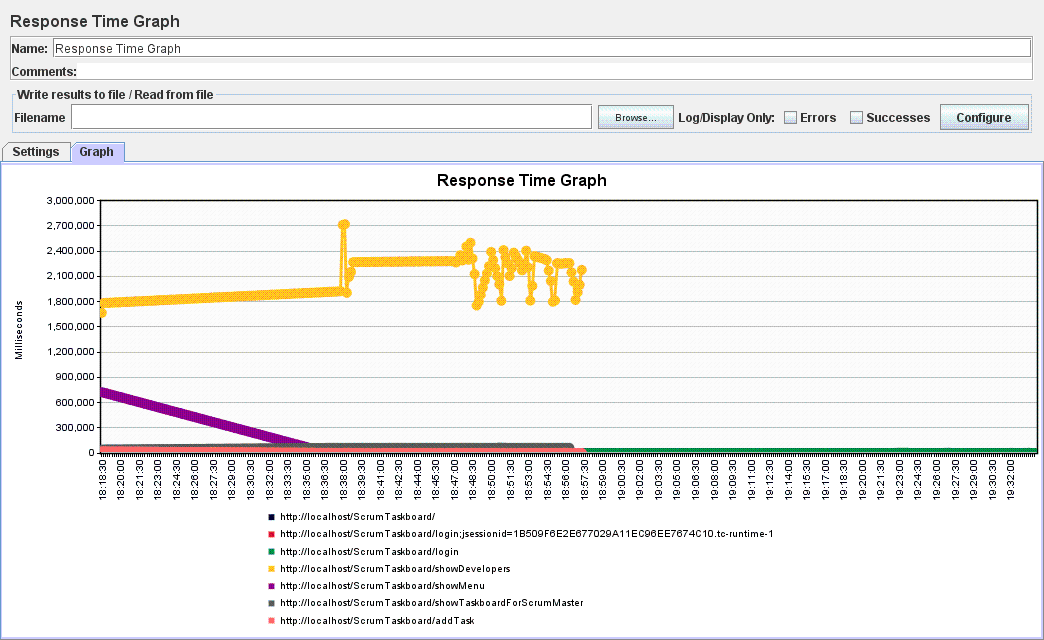


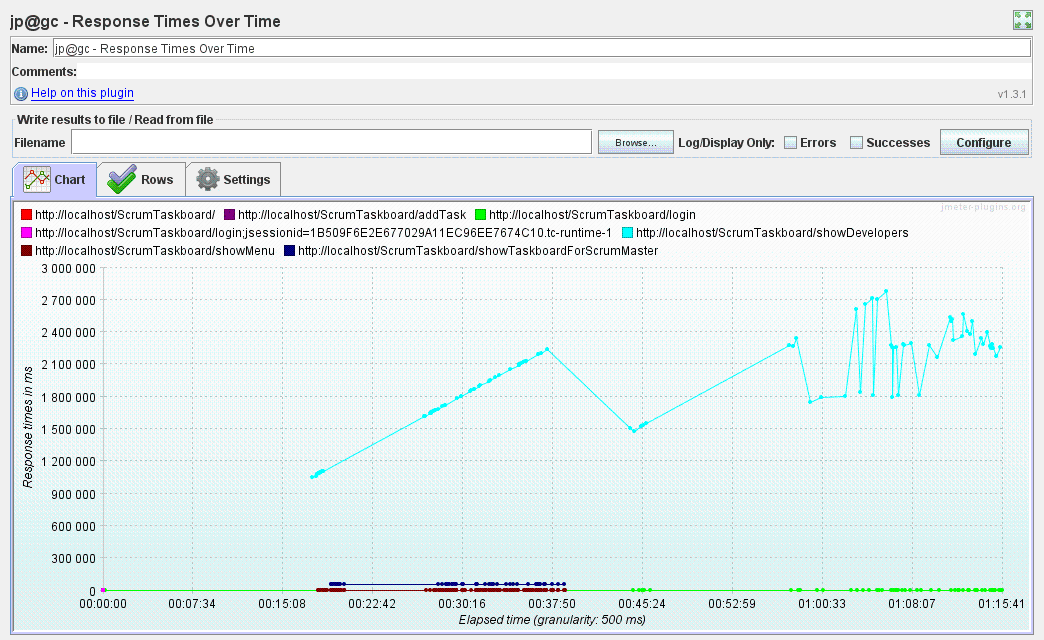
Distribution Graph (alpha)

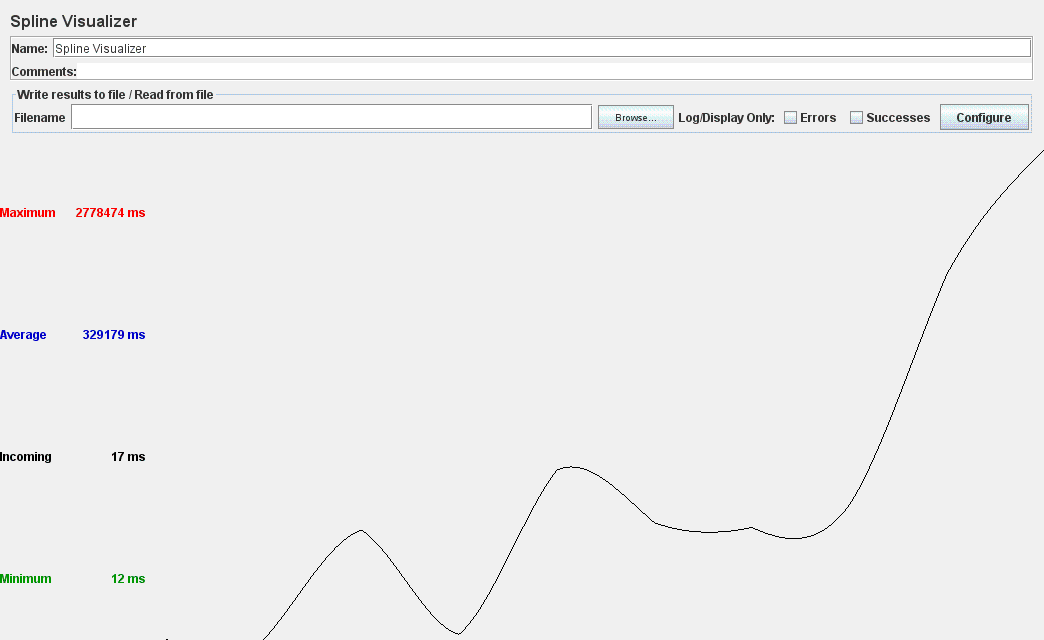


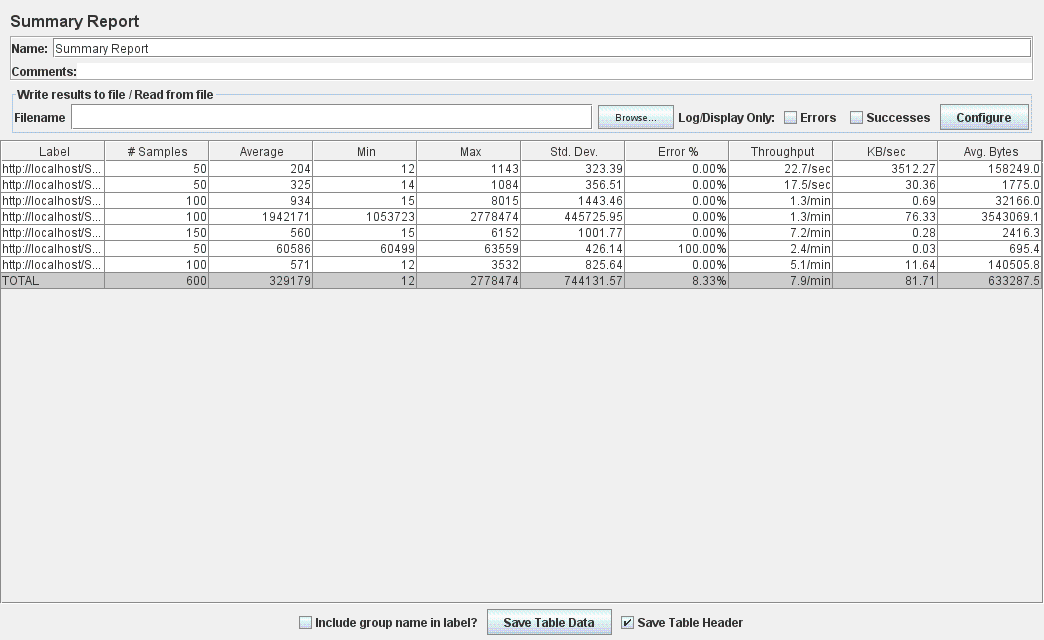


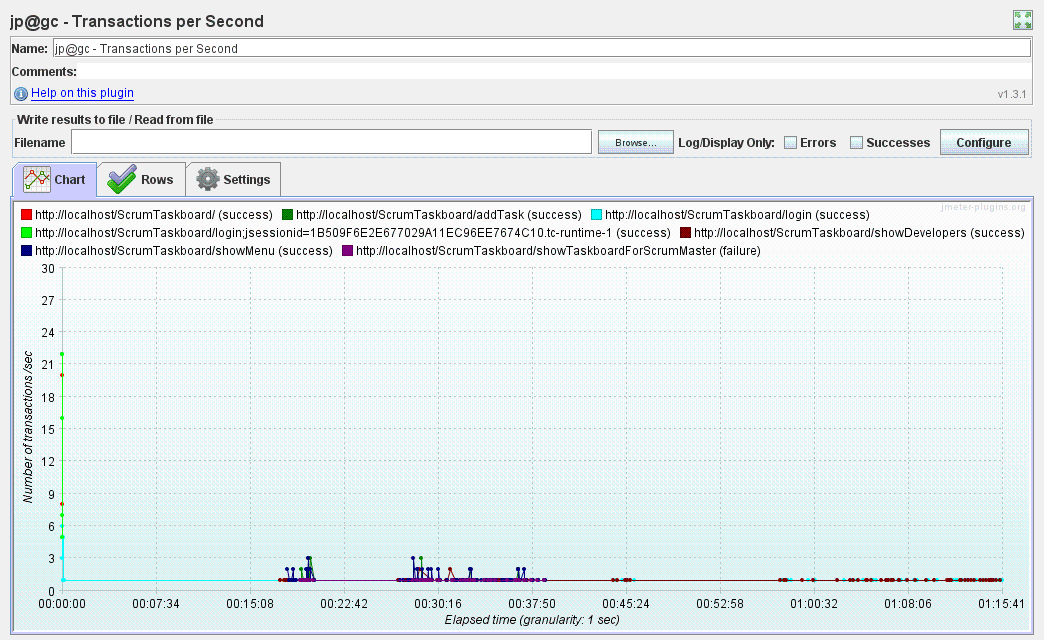


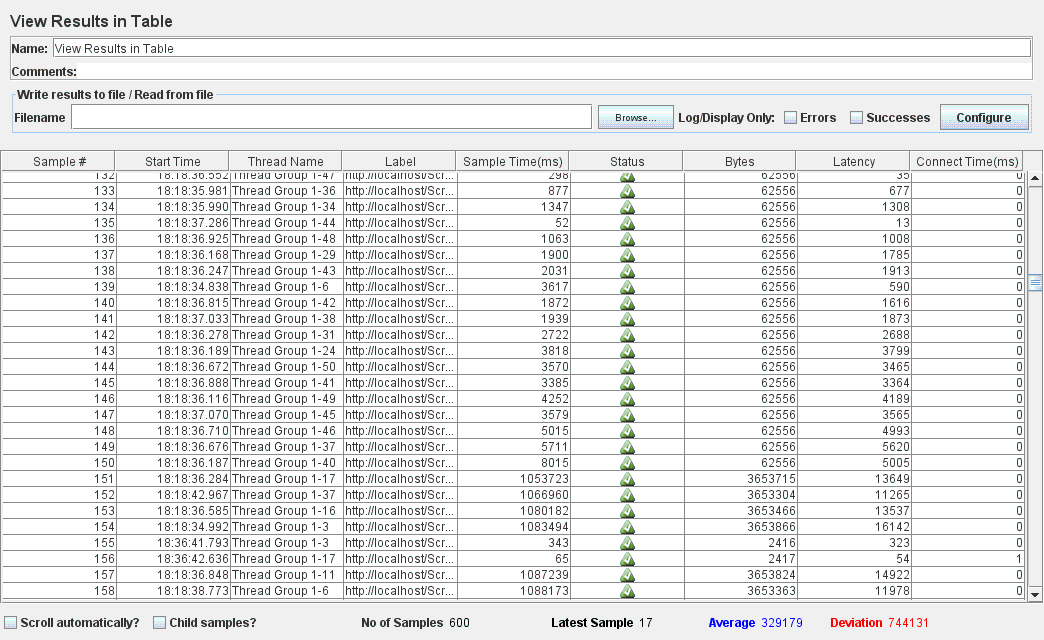






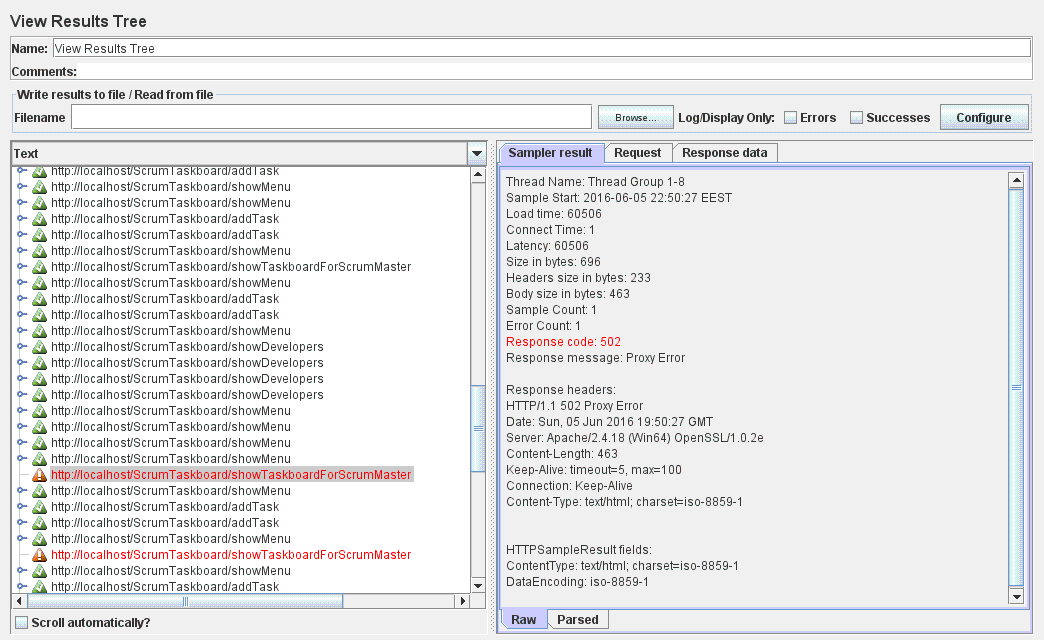


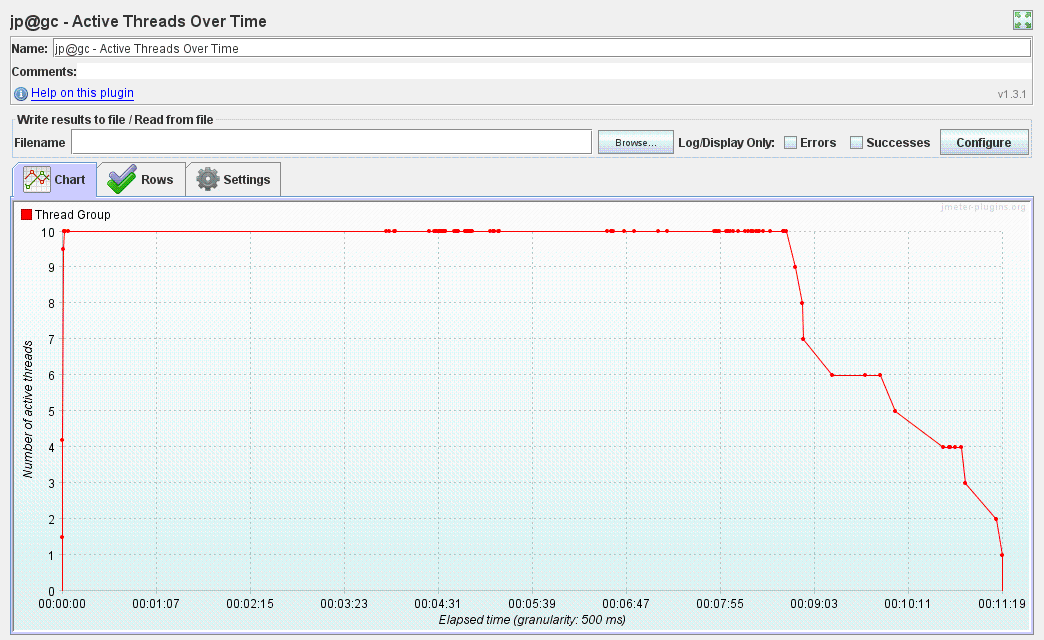


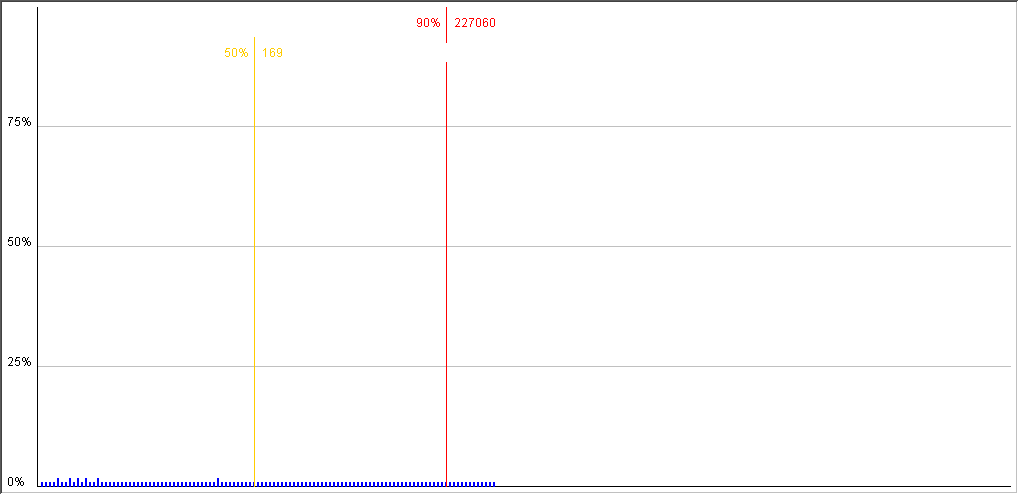


***After*** implementing the Spring eh-caching (as described in a previous paragraph), the **Login\_Visualize developers\_Visualize taskboard\_Add task\_Visualize developers\_Logout** testcase was executed from JMeter under the same conditions: *1000 developers* existent in the database and by simulating a different number of simultaneous users performing the use case (10 and 50). The results obtained from JMeter performance tests are presented as follows:

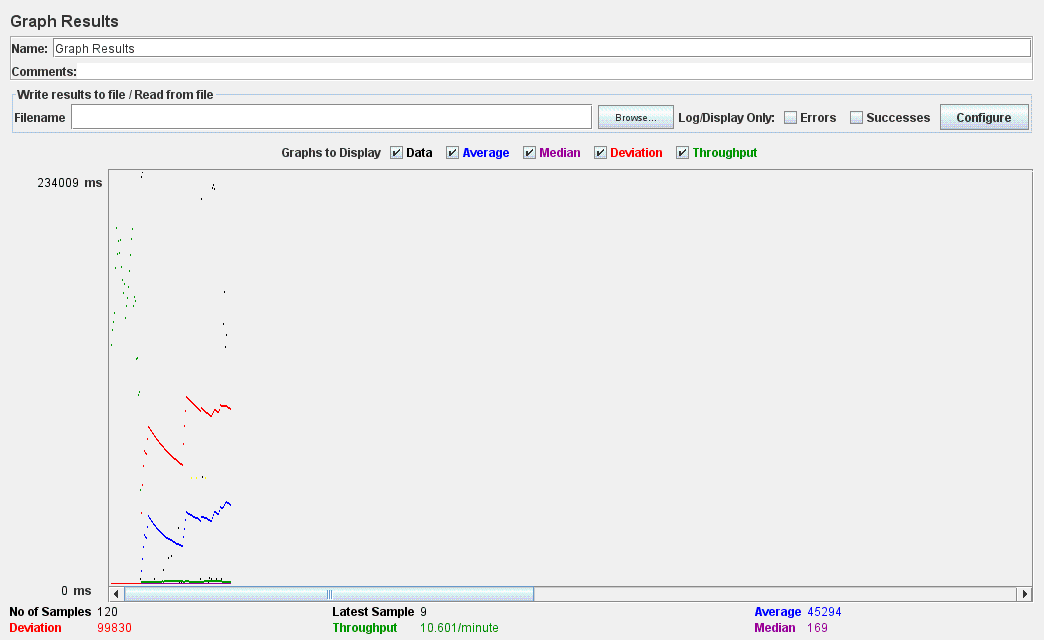
* For **10 concurrent users**:

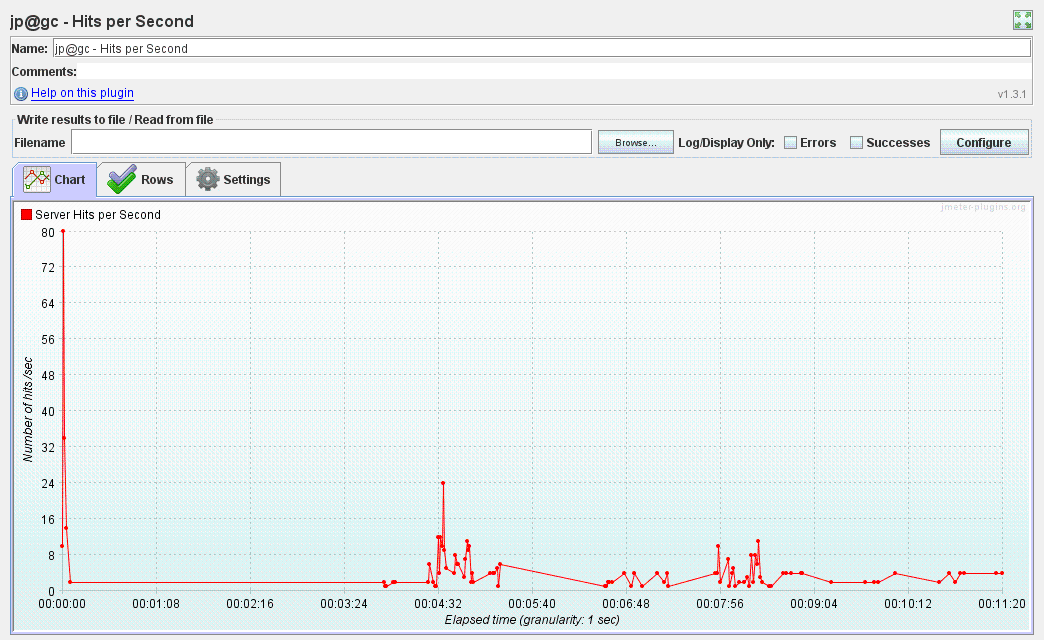


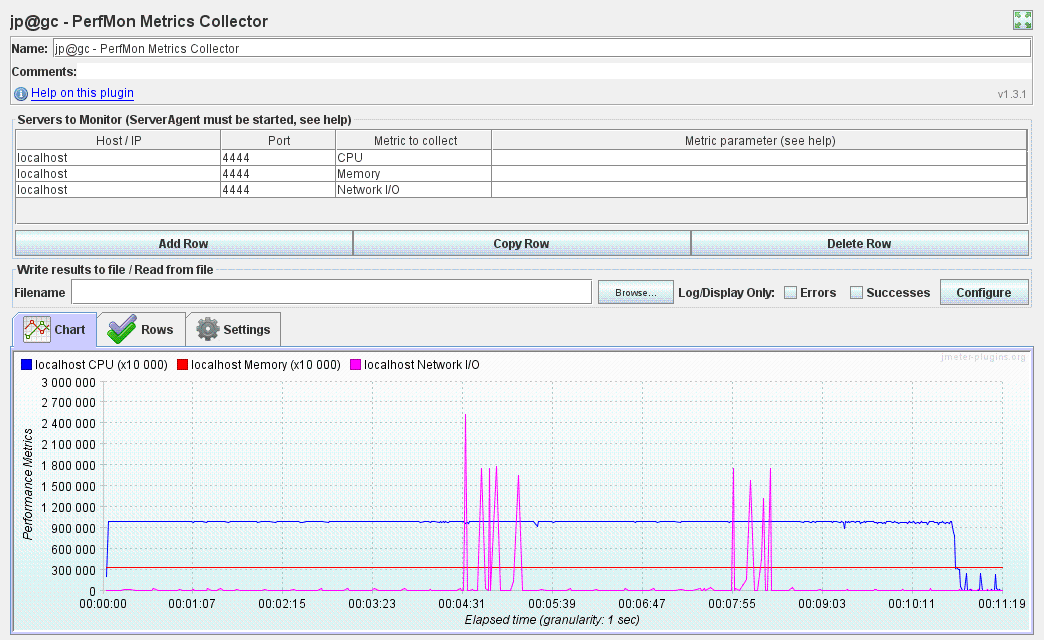


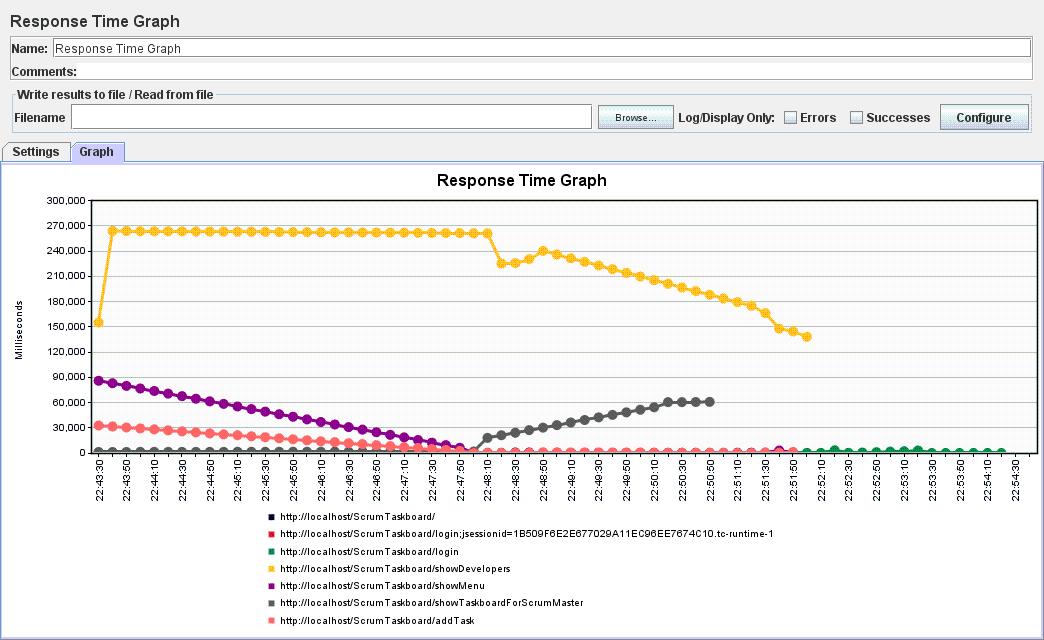


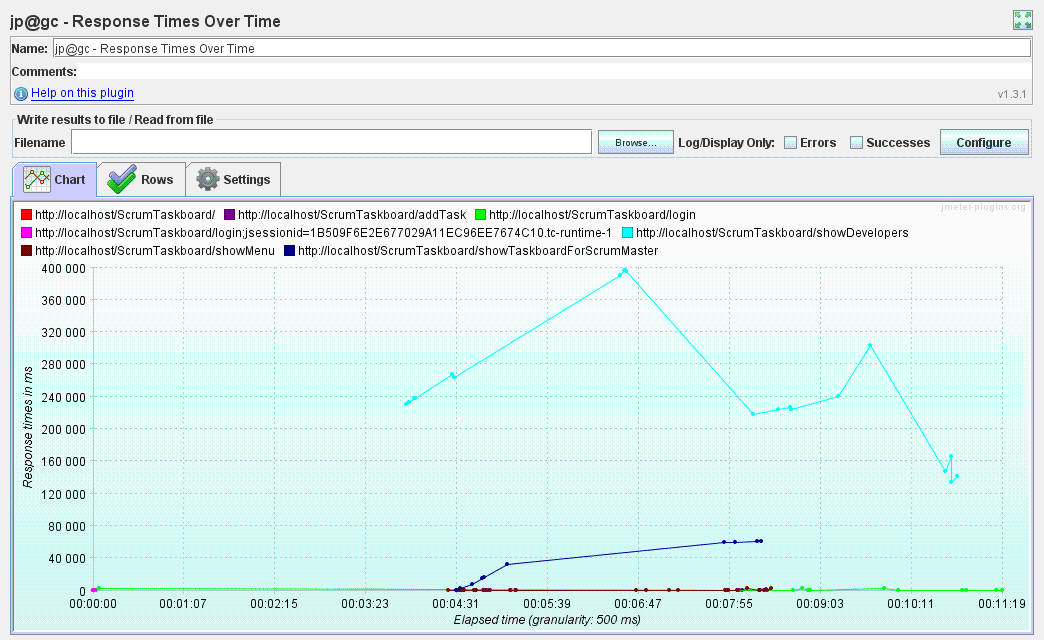
Distribution Graph (alpha)

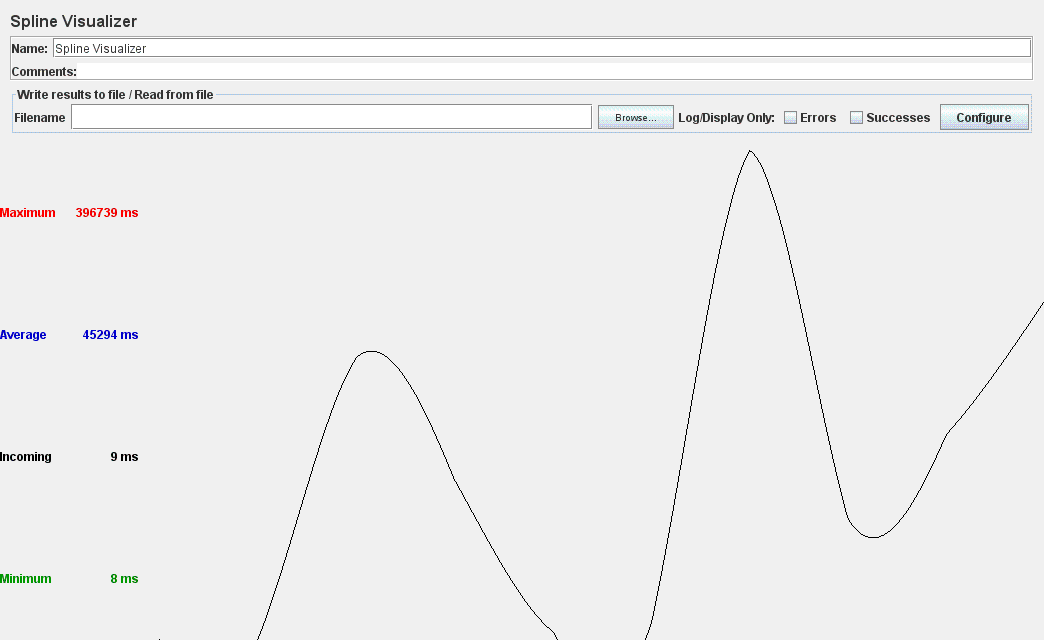


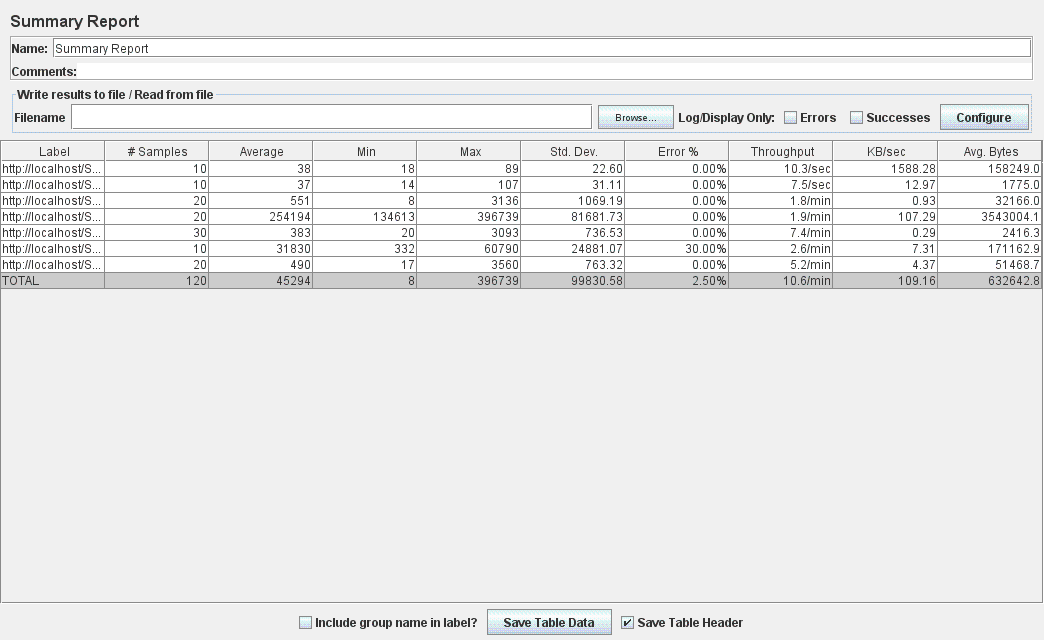


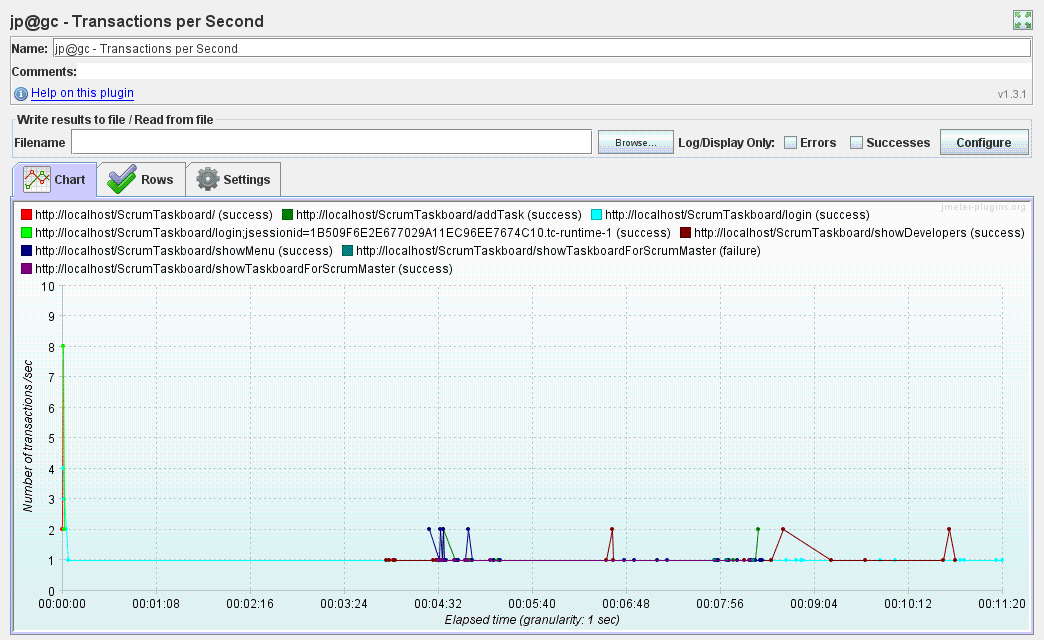


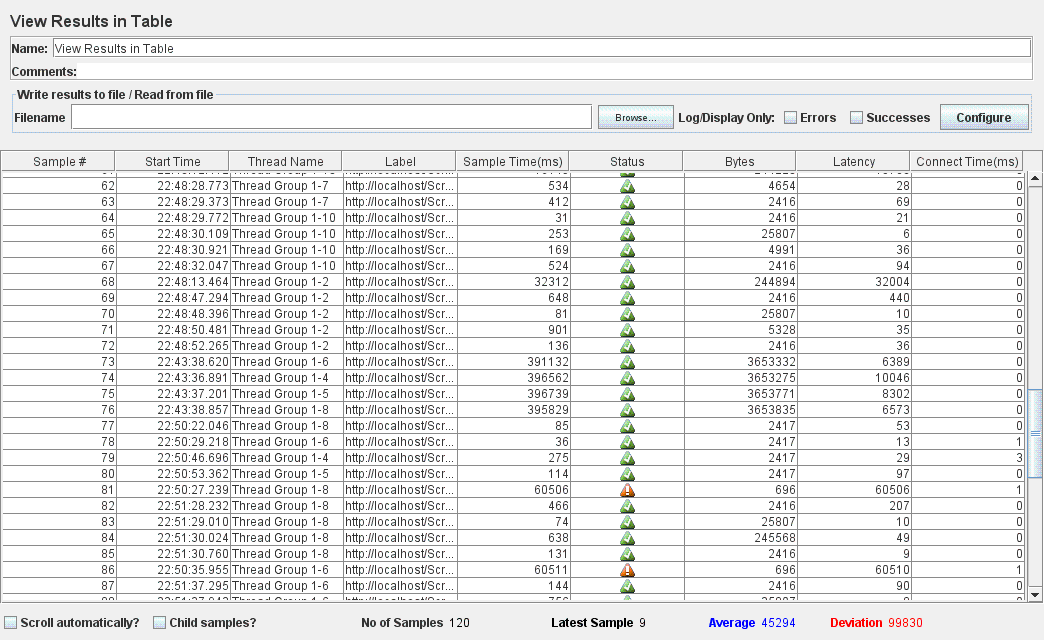






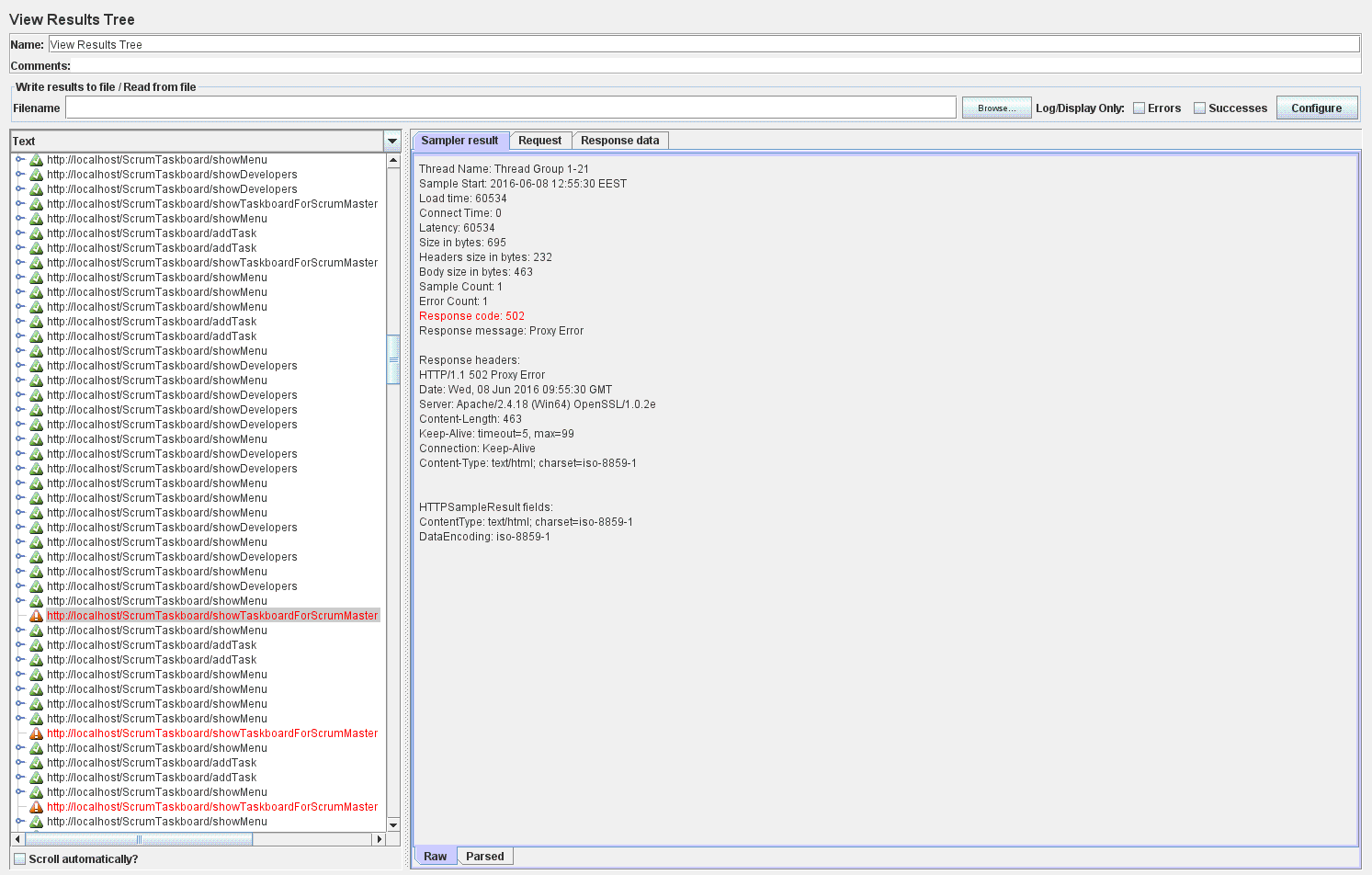


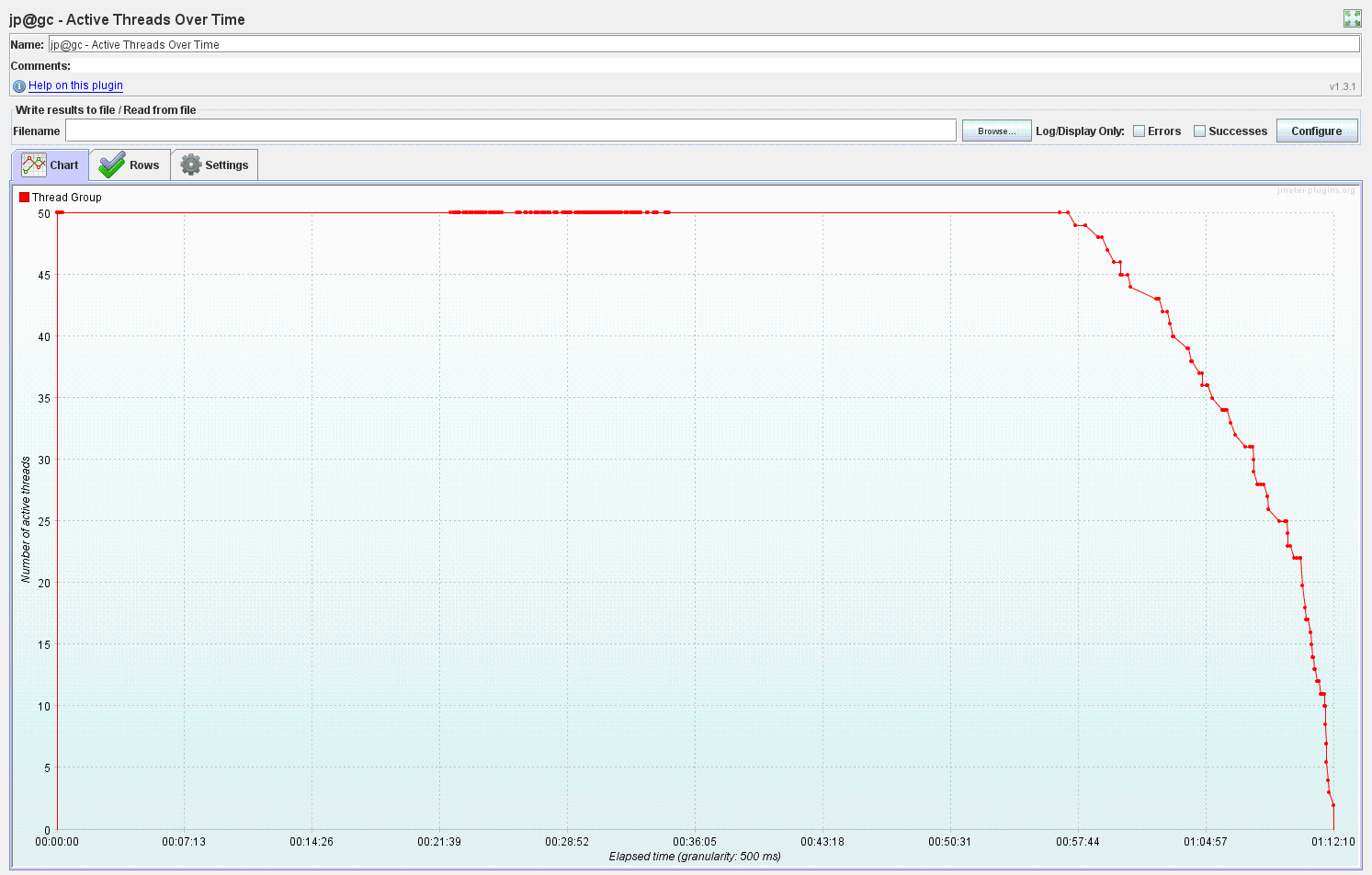


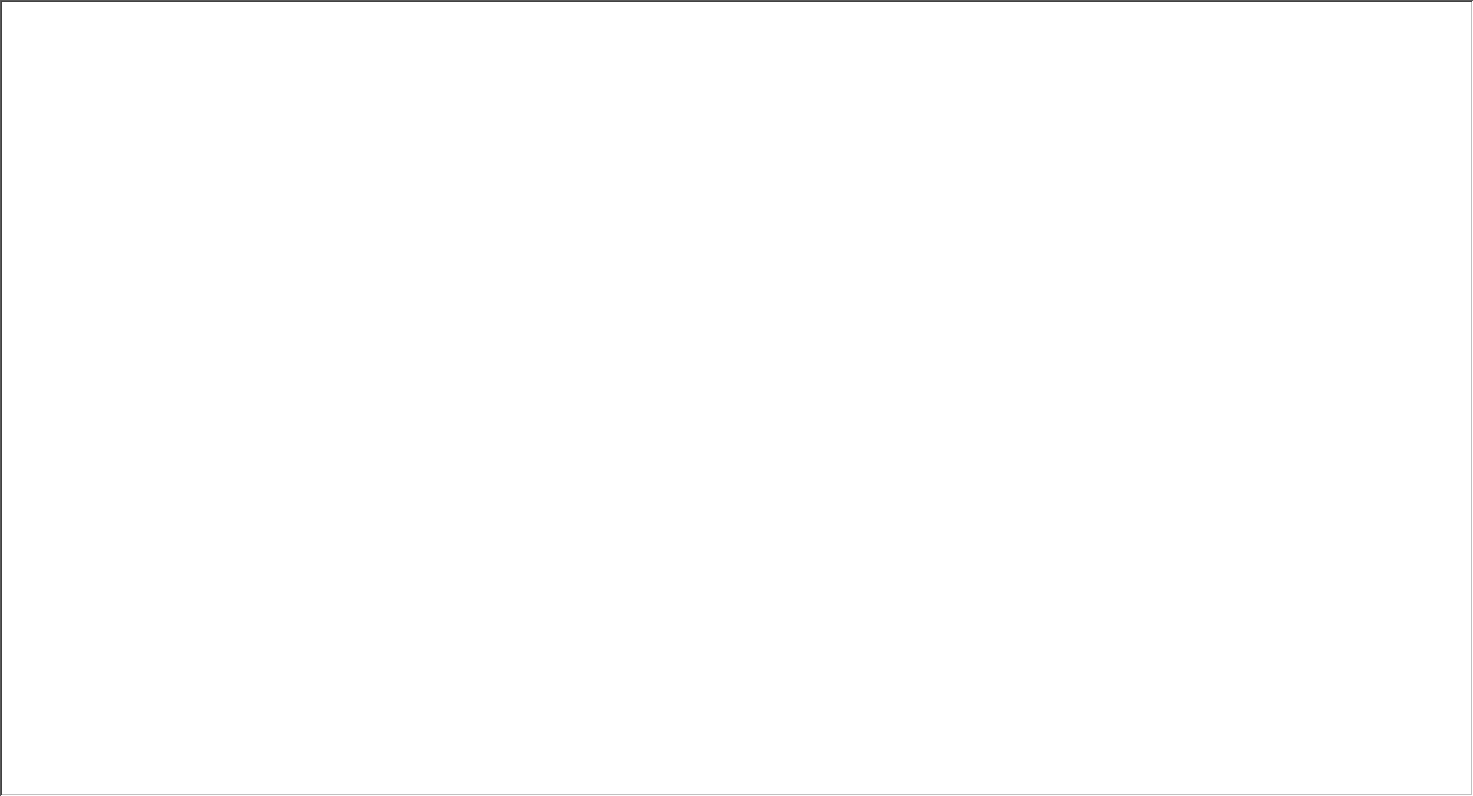


**Eroarea cu caching (2.50%) e mai mare decat fara caching (1.67%) ??? – de executat din nou testcase-ul inainte si dupa adaugarea cache-ului**

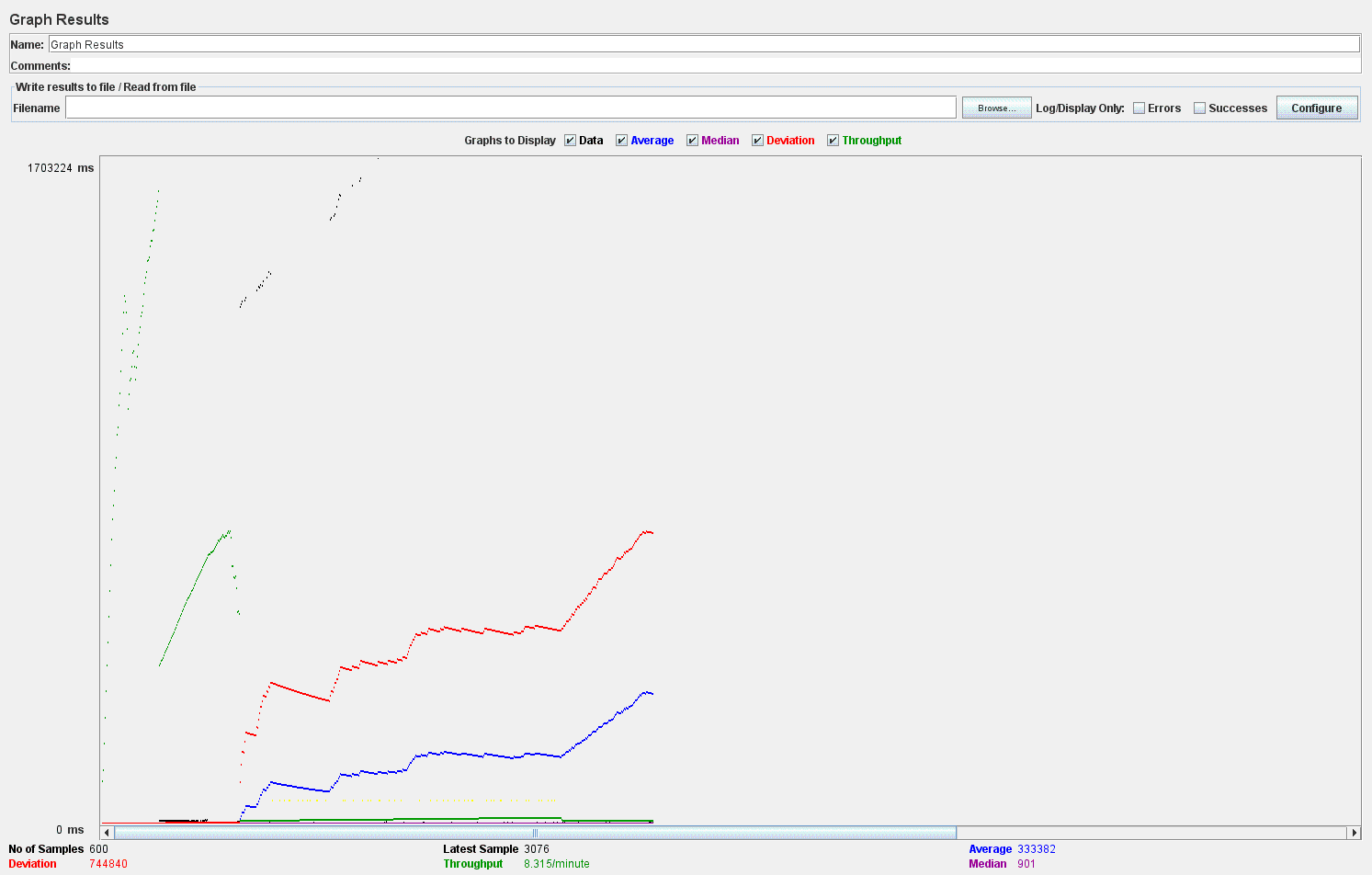
* For **50 concurrent users**:

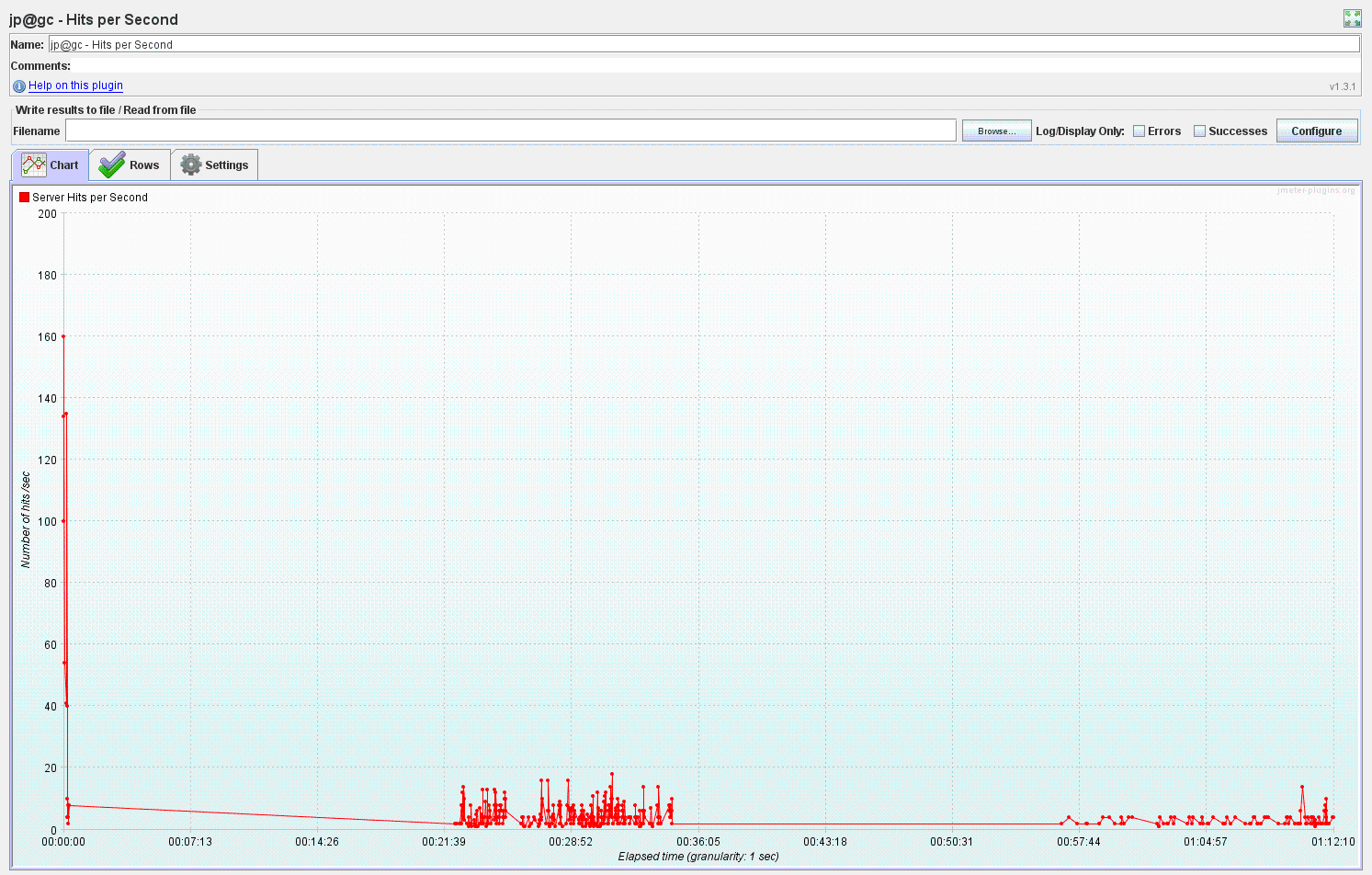


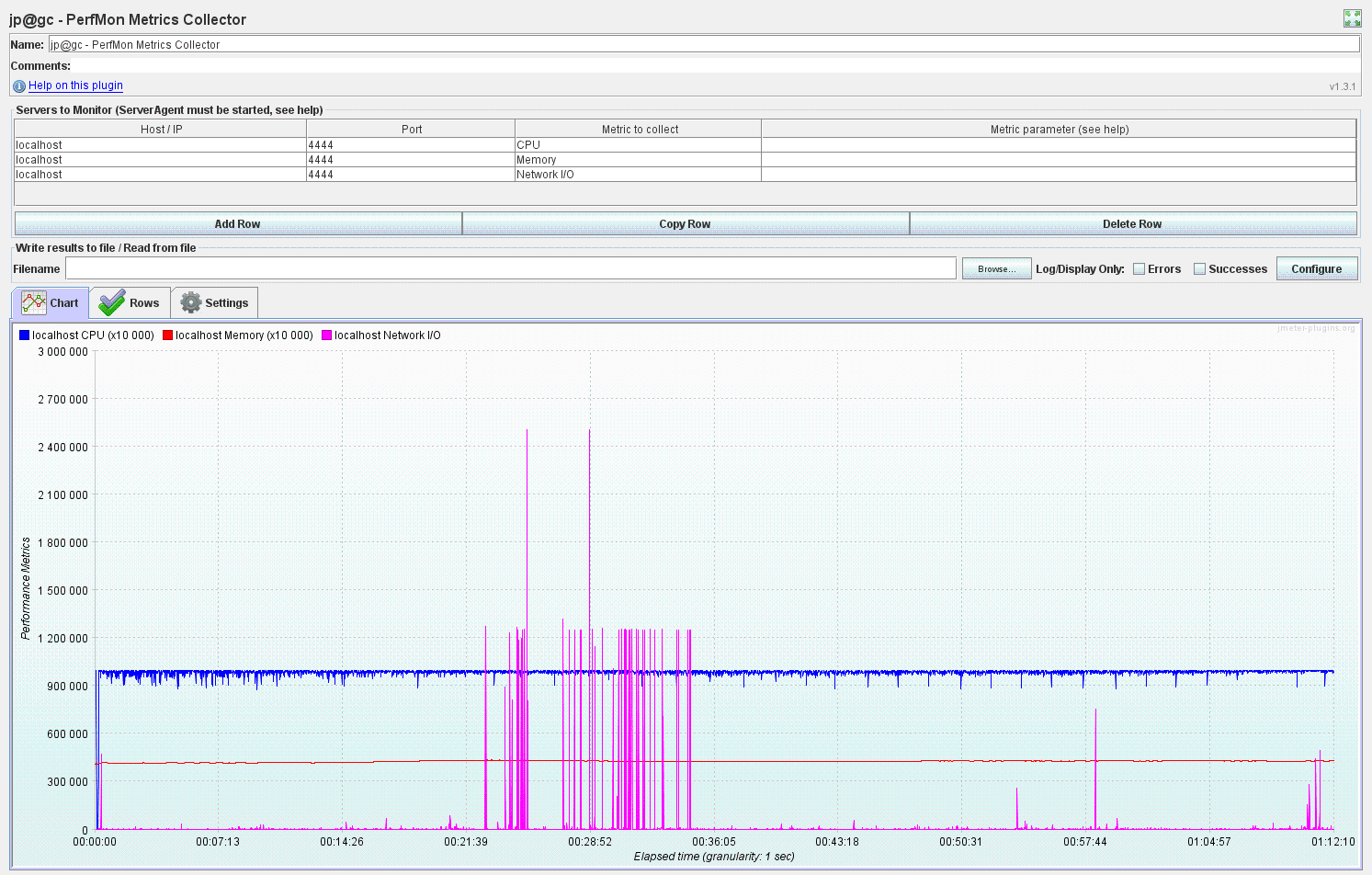


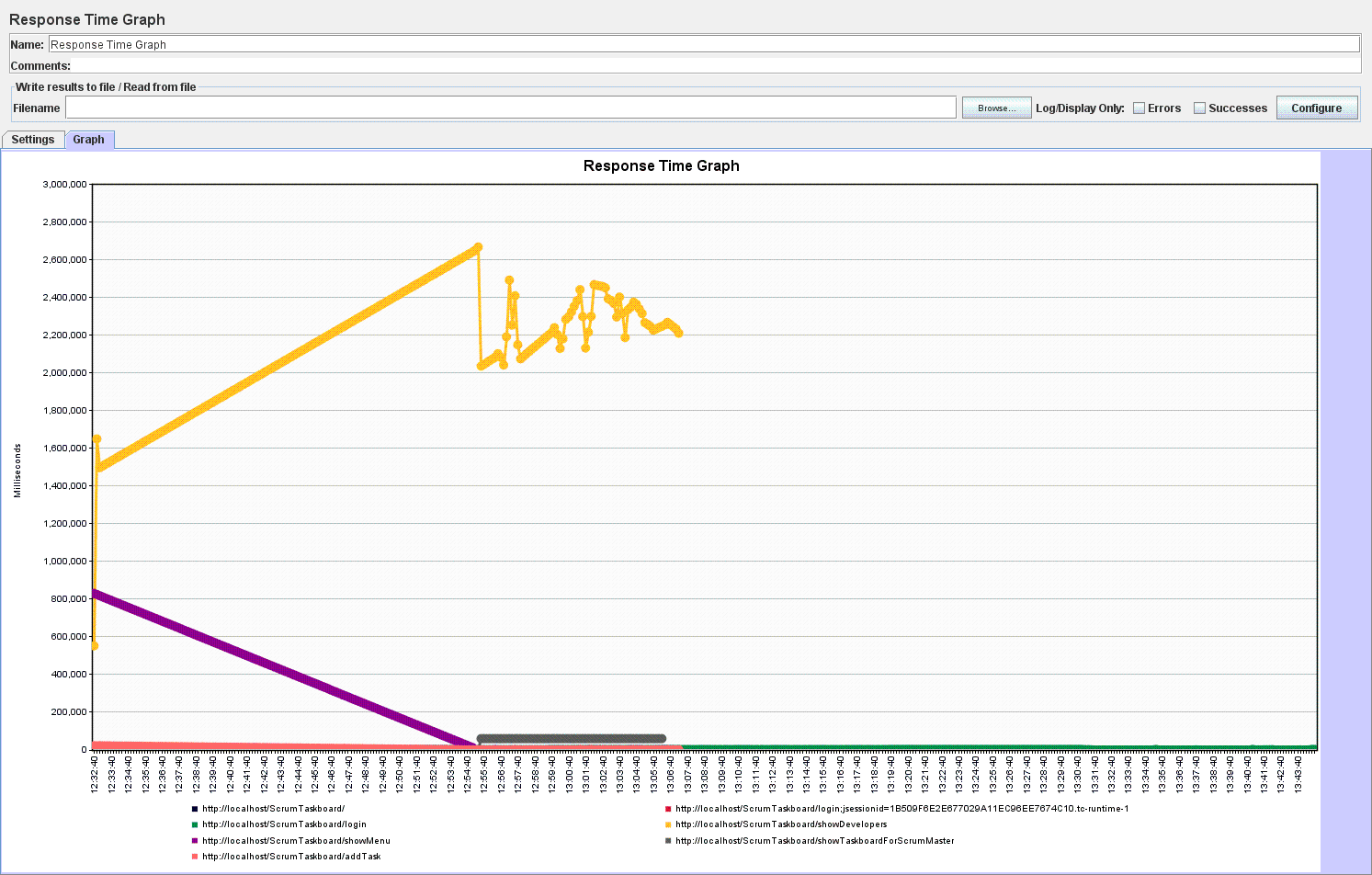


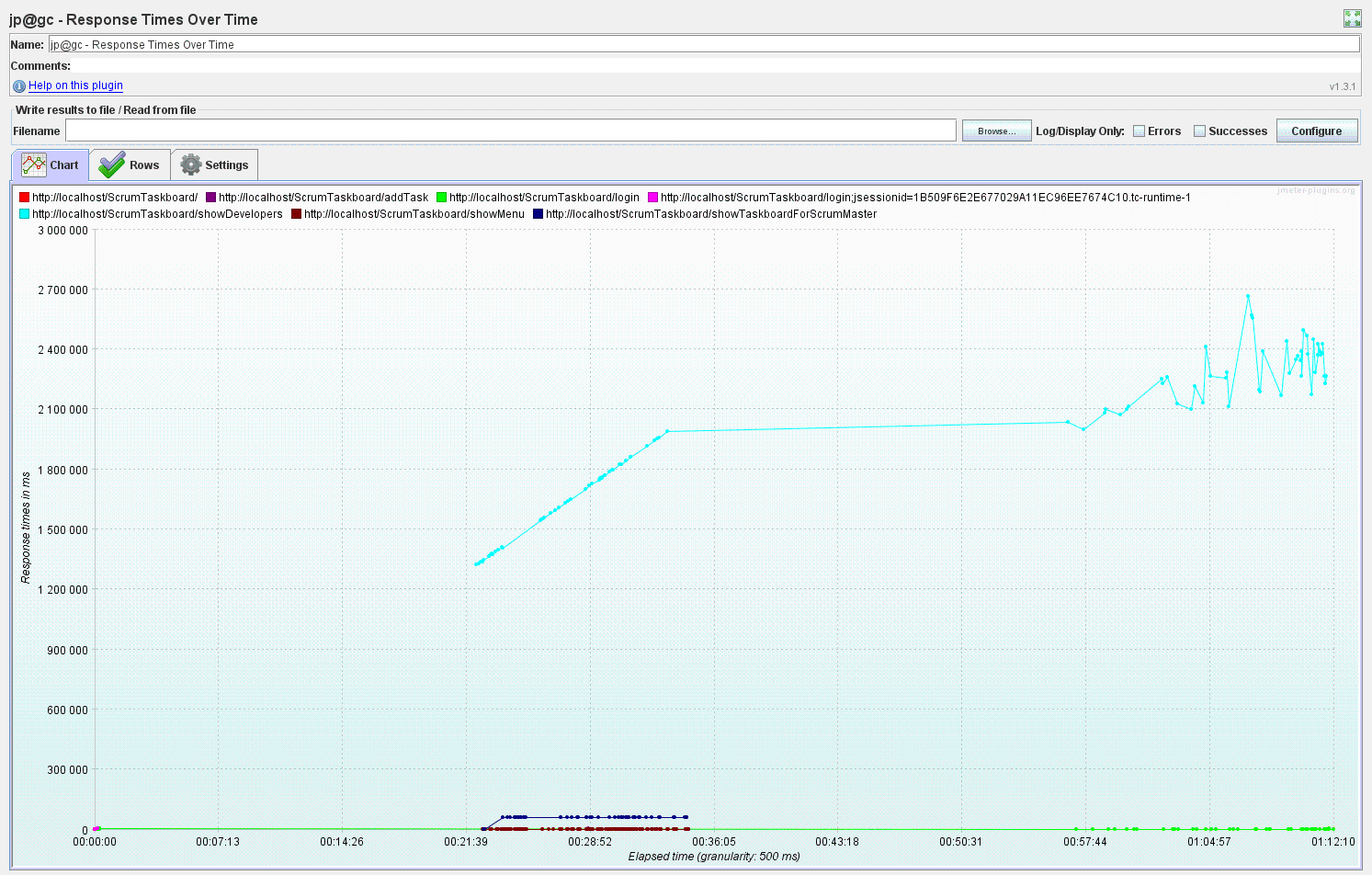
Distribution Graph (alpha)

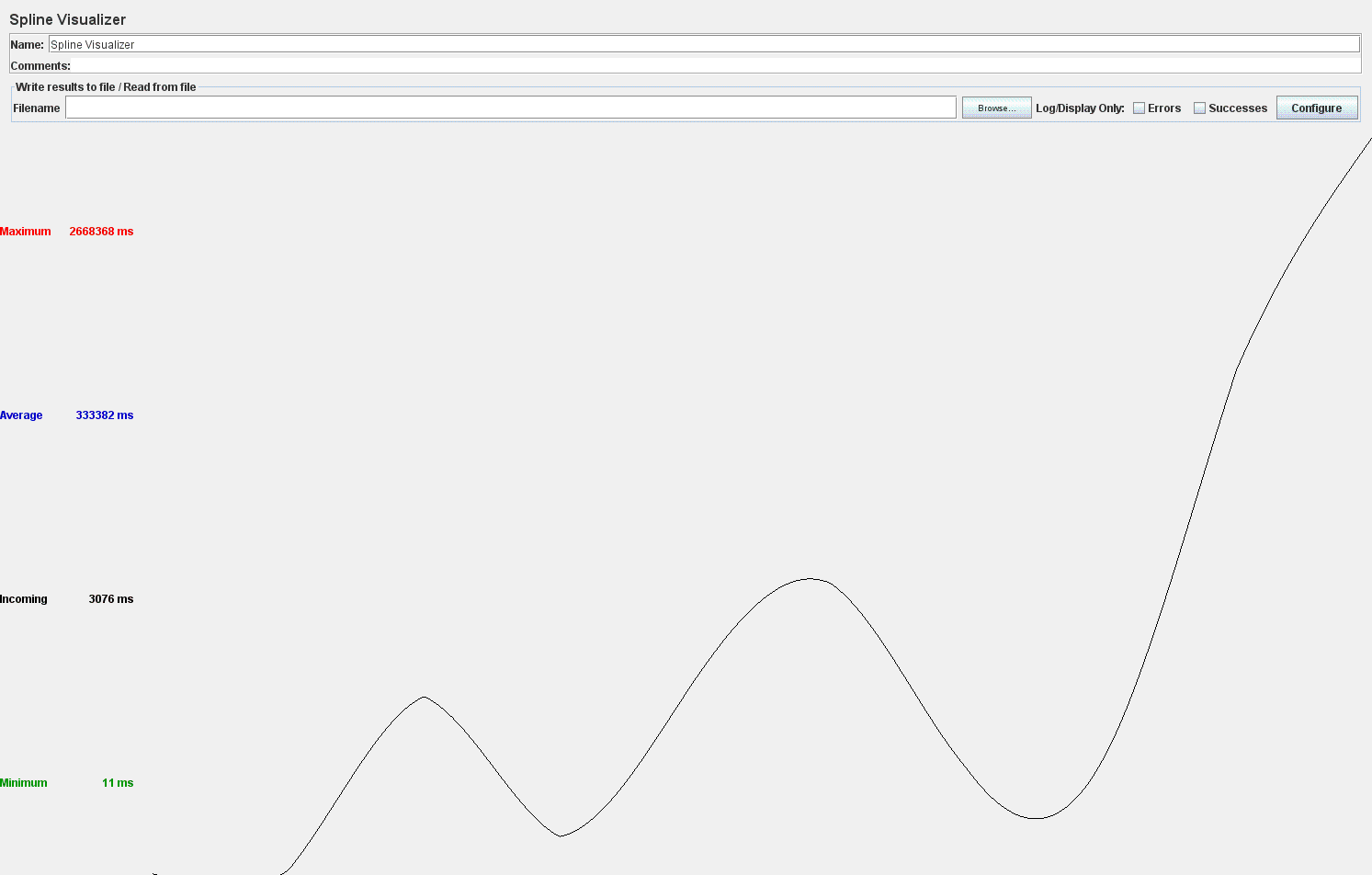


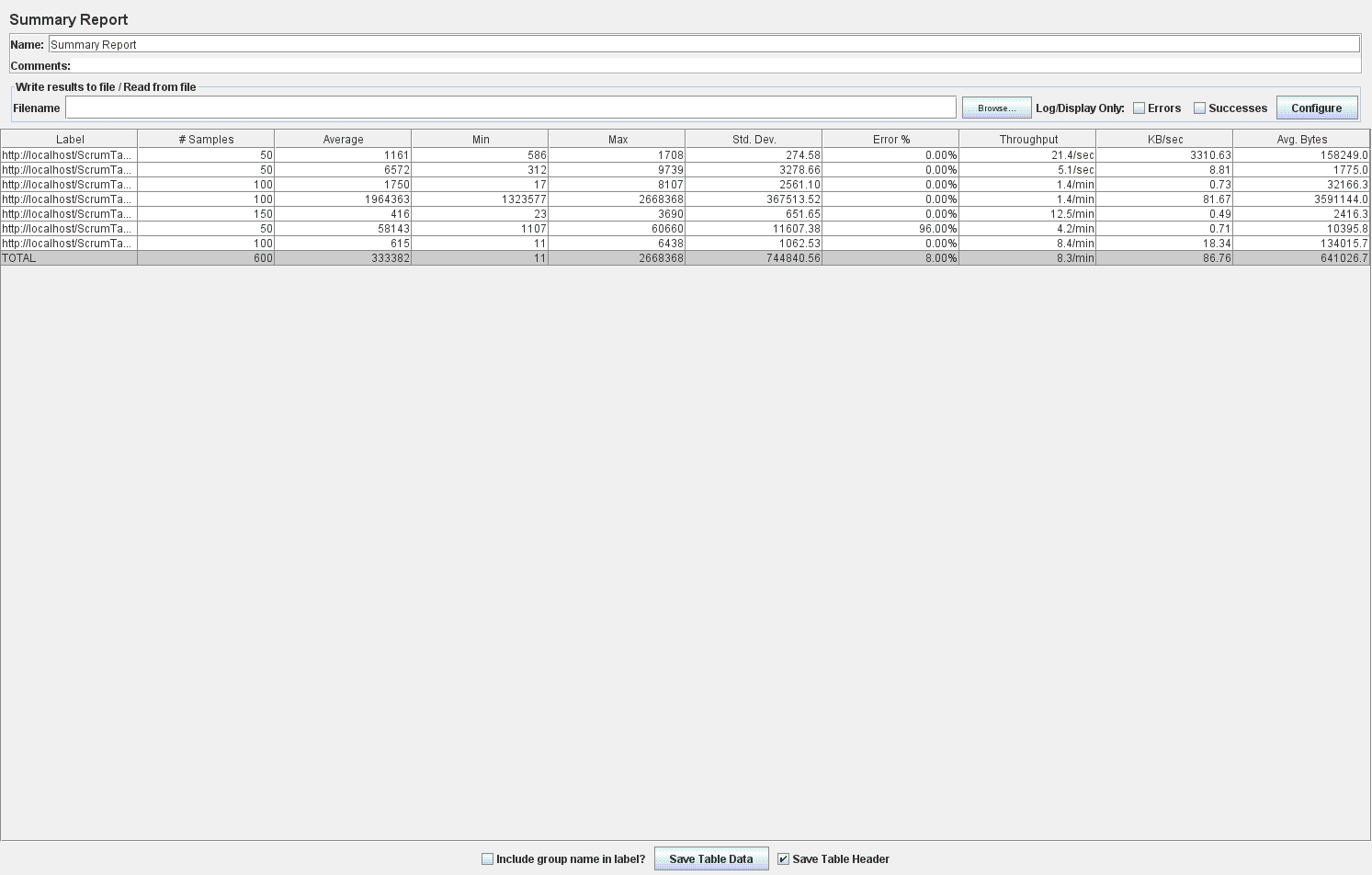


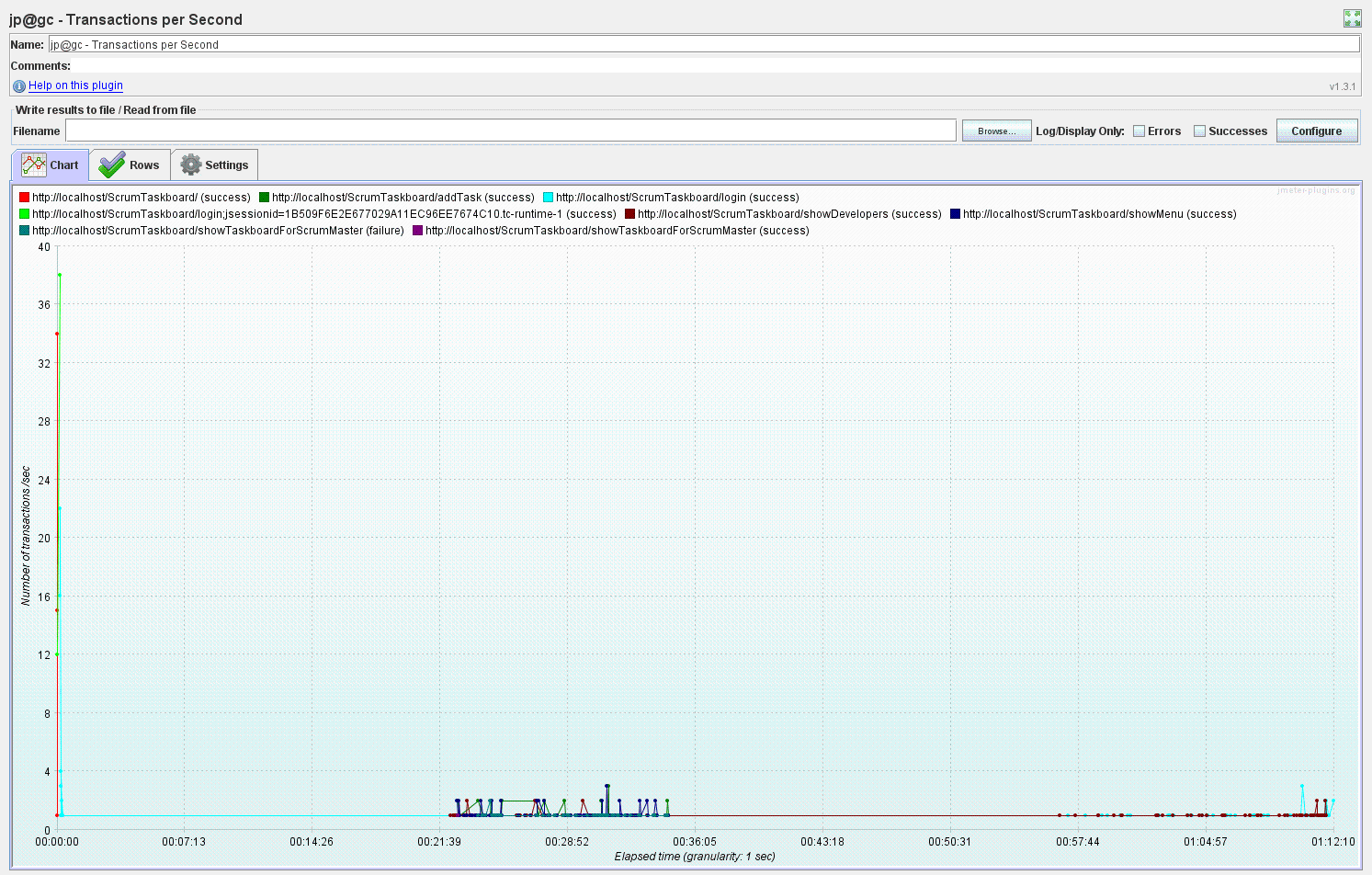


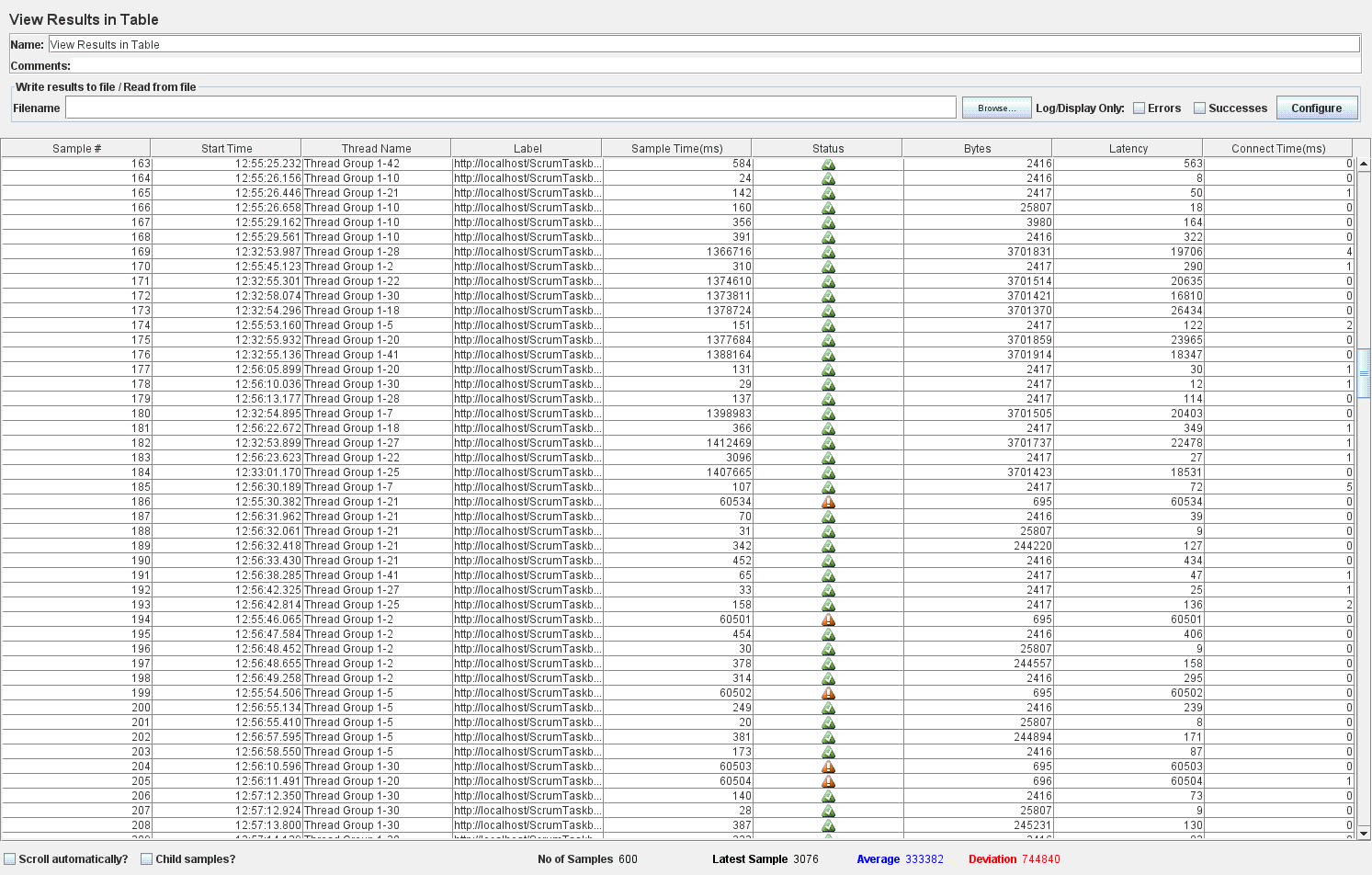












**Eroarea cu caching (8.00%) e cu putin mai mica decat fara caching (8.33%) ??? – de executat din nou testcase-ul inainte si dupa adaugarea cache-ului**

Am re-rulat testcase-ul din JMeter inainte de adaugarea cache-ului si eroarea raportata e aceeasi . Erorile. Erorile raportate sunt doar de tipul **502:ProxyError**.

**Next step: - solve ProxyErrors before adding caching**

**Methodology**:

1. Investigate the Apache and Tomcat servers logs:
   * Tomcat logs: nothing
   * Apache logs:

[proxy\_http:error] A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond. : [client 127.0.0.1:60718] AH01102: error reading status line from remote server 127.0.0.1:8080, referer: http://localhost/ScrumTaskboard/showMenu/

[proxy:error] [pid 1420:tid 892]: Error reading from remote server returned by /ScrumTaskboard/showTaskboardForScrumMaster, referer: <http://localhost/ScrumTaskboard/showMenu>

1. Modify timeout-specific settings for the Apache server (in *httpd-default.conf*):

***Timeout 600***

***ProxyTimeout 600***

Results:

* for 10 concurrent users: the 1.67% error reported before was reduced to 0%
* for 50 concurrent users: the 8.33% error reported before was reduced to 7.33%

1. Increase the timeout parameter values:

***Timeout 6000***

***ProxyTimeout 6000***

Results:

* for 10 concurrent users: the errorless (0%) behavior was preserved
* for 50 concurrent users: the 7.33% error reported for **2.** was further on reduced to 6.83%

1. Preserve the settings from **3.** and increase the *KeepAliveTimeout* one (from 5 to 600000):

***Timeout 6000***

***ProxyTimeout 6000***

***KeepAliveTimeout 600000***

Results:

* for 10 concurrent users: the error reported for **3.** (0%) was increased to 2.50%
* for 50 concurrent users: the 6.83% error reported for **3.** was further on reduced to 5.83%

1. Preserve the settings from **4.** (configured in *httpd-default.conf* file*)* and update the cluster’s properties in *httpd.conf* file:
   * specify the *timeout* property for the server node member of the cluster
   * set on the *nofailover* property for the ProxyPass

*httpd-default.conf*

***Timeout 6000***

***ProxyTimeout 6000***

***KeepAliveTimeout 600000***

*httpd.conf*

*<Proxy balancer://testcluster stickysession=JSESSIONID>*

*BalancerMember http://127.0.0.1:8080 min=10 max=100 route=node1 loadfactor=1* ***timeout=6000***

*</Proxy>*

*ProxyPass /ScrumTaskboard balancer://testcluster/ScrumTaskboard* ***nofailover=On***

*ProxyPassReverse /ScrumTaskboard http://127.0.0.1:8080/ScrumTaskboard*

Results:

* for 10 concurrent users: the 2.50% error reported for **4.** was reduced to **0%**
* for 50 concurrent users: the 5.83% error reported for **4.** was reduced to **0%**
* for 100 concurrent users: **0%** error was obtained (2h to handle all requests)
* the ProxyError reported when running the JMeter testcase (for a variable number of concurrent users) was solved by the Apache configuration specified above !
* after adding caching (using Spring eh-cache), the testcase was executed with 100 concurrent threads and the results obtained were:
  + 0% error was preserved
  + the total time to handle all requests was ~ 3h

**After call with Marian (22.06.2016):**

* Conclusions: performance improvements brought by Spring eh-cache ???
  + JMeter: increase memory in order to remove the “*OutOfMemory: Java Heap space*” error reported when running the testcase with 100 threads => configure JMeter parameters to allocate greater memory
  + Rerun testcase and evaluate results: compare if the time to handle all requests (100 threads) is reduced when using caching.
  + In case no time improvement is obtained when using caching: study eh-cache specific parameters (corresponding to the cache region defined in *ehcache.xml*) in order to understand how to properly configure them so that to take advantage of the Spring eh-caching functionality.
  + **The desired outcome would be a reduce in the testcase execution time when using the Spring eh-cache.**

**Steps performed** in order to **remove the *OutOfMemory****: Java Heap space* error obtained when running the JMeter testcase with 100 threads and **show the performance improvements brought by Spring eh-cache** (<https://www.blazemeter.com/blog/nine-easy-solutions-jmeter-load-test-%E2%80%9Cout-memory%E2%80%9D-failure>):

**Step 1**:

* Increase the JMeter memory: update the values for the JVM memory allocation parameters in *jmeter.bat* file:

Line:

**set HEAP=-Xms512m -Xmx512m**

was replaced with:

**set HEAP=-Xms512m -Xmx8192m**

* Remove several JMeter output listeners from the testcase (as they consume valuable resources that are needed by the test being run) and preserve only the ones needed in assessing the performance results: View Results Tree, Graph Results, Response Time Graph, Summary Report and View Results in Table.

Results:

* The ***OutOfMemory****: Java Heap space* error was removed when running the JMeter testcase with 100 threads.
* The testcase (run from JMeter GUI with 100 threads) took more than 2h (~3h) to complete, which indicates worse performance than for the non-cached version of the application (which took 2h to finish the testcase). Therefore, none of the solutions carried at Step1 to improve the JMeter testcase run and, therefore, illustrate the performance enhancements obtained for the cached version of the application, were of help.

**Step 2**:

* Run the JMeter testcase in Non-GUI mode as, according to the documentation, the GUI mode will not only freeze - but it will also consume lots of resources and produce unreliable load test results. Therefore, the VisualizeDevelopers-100threads.jmx testcase was executed from command line and the results obtained stored in VisualizeDevelopers-100threads-Results.jtl log file (which might be further on imported into JMeter GUI to be visually evaluated through the desired output listeners):

**jmeter -n -t VisualizeDevelopers-100threads.jmx -l VisualizeDevelopers-100threads-Results.jtl**

where:

**-n**: Specifies JMeter is to run in Non-GUI mode

**-t** [name of JMX file that contains the Test Plan]

**-l** [name of JTL file to log sample results to]

Results:

* Even if the JMeter testcase was run in Non-GUI mode (from the command line) with 100 threads, no execution enhancement was obtained. Therefore, running the test which output the results in the .jtl log file, took almost the same amount of time as the GUI-mode alternative.

**Step 3**:

* Remove **all** JMeter output listeners from the testcase because theory says that generating load and a report at the same time is not the best practice. According to Tsung's default report generation technique, the appropriate behavior is to only collect test results into a file and analyze them post test.

(<https://www.blazemeter.com/blog/how-reduce-memory-usage-jmeter?utm_source=Blog&utm_medium=BM_Blog&utm_campaign=OOMFailure>)

* Update JMeter’s *user.properties* file with the following configuration, intended to declare the type of results that will be saved and in which format:

jmeter.save.saveservice.data\_type=false

jmeter.save.saveservice.response\_code=true

jmeter.save.saveservice.response\_data.on\_error=false

jmeter.save.saveservice.response\_message=false

jmeter.save.saveservice.successful=true

jmeter.save.saveservice.thread\_name=true

jmeter.save.saveservice.time=true

jmeter.save.saveservice.subresults=false

jmeter.save.saveservice.assertions=false

jmeter.save.saveservice.latency=true

jmeter.save.saveservice.bytes=true

jmeter.save.saveservice.hostname=true

jmeter.save.saveservice.thread\_counts=true

jmeter.save.saveservice.sample\_count=true

jmeter.save.saveservice.response\_message=false

jmeter.save.saveservice.assertion\_results\_failure\_message=false

jmeter.save.saveservice.timestamp\_format=yyyy/MM/dd HH:mm:ss

jmeter.save.saveservice.default\_delimiter=;

* Run the JMeter testcase in **Non-GUI** mode (as in the previous step) and collect the results in a log file.
* Add the end of the test, open the JMeter GUI and add the "Simple Data Writer" listener (specifying also the path to collected results file). Moreover, any output listener might be added to the testplan, inputting the path to the results file and then visually evaluating performance features.

Results:

* Despite of the configuration performed when running the JMeter testcase in Non-GUI mode (from the command line) with 100 threads, no execution enhancement was obtained. Therefore, running the test which output the results in the .jtl log file, took 2 hours and 30 minutes.

**Step 4**:

* Modify the following JVM arguments JMeter’s startup file (*jmeter.bat):*
* As the JVM already uses 64-bit mode, no additional property has to be configured to customize this feature.
* Replace the NEW="-XX:NewSize=128m -XX:MaxNewSize=128m" by increasing the values to match the ones set for HEAP:

set NEW=-XX:NewSize=512m -XX:MaxNewSize=8192m

* Run the JMeter test with the JVM switched to *server* mode (with runtime parameters optimization). It is said that JMeter starts more slowly in *server* mode, but the overall throughput will be higher. To execute the test in server mode, the *jmeter-server* bat is executed from the cmd instead of the *jmeter* one.
* Set -XX:+UseConcMarkSweepGC to force the usage of the CMS garbage collector. It will lower the overall throughput but leads to much shorter CPU intensive garbage collections:

**set CMS\_GC=-XX:+UseConcMarkSweepGC**

* Set -XX:+DisableExplicitGC to prevent applications from forcing expensive garbage collections and helps avoid unexpected pauses

**set DISABLE\_APPS\_FORCING\_GC=-XX:+DisableExplicitGC**

* Finally append these last two properties to the ARGS variable in *jmeter.bat* file:

**set ARGS**=%DUMP% %HEAP% … %**CMS\_GC**% %**DISABLE\_APPS\_FORCING\_GC**%

* Run the JMeter testcase in **Non-GUI** mode (as in the previous steps) and collect the results in a log file:

**jmeter-server -n -t VisualizeDevelopers-100threads.jmx -l VisualizeDevelopers-100threads-Results.jtl**

* Add the end of the test, open the JMeter GUI and add the "Simple Data Writer" listener (specifying also the path to collected results file). Moreover, any output listener might be added to the testplan, inputting the path to the results file and then visually evaluating performance features.

Results:

* When running the JMeter test in Non-GUI mode, using server mode JVM, the following message is prompted, but the process does not execute and therefore manual shutdown has to be performed:

*Created remote object: UnicastServerRef [liveRef: [endpoint:[192.168.56.1:11243](local),objID:[74e03654:155ac40a054:-7fff, 2608491278354252972]]]*

**Step 5**:

* Modify the following JVM arguments JMeter’s startup file (*jmeter.bat):*
* As the JVM already uses 64-bit mode, no additional property has to be configured to customize this feature.
* Replace the NEW="-XX:NewSize=128m -XX:MaxNewSize=128m" by increasing the values to match the ones set for HEAP:

**set NEW=-XX:NewSize=512m -XX:MaxNewSize=8192m**

* Set -XX:+UseConcMarkSweepGC to force the usage of the CMS garbage collector. It will lower the overall throughput but leads to much shorter CPU intensive garbage collections:

**set CMS\_GC=-XX:+UseConcMarkSweepGC**

* Set -XX:+DisableExplicitGC to prevent applications from forcing expensive garbage collections and helps avoid unexpected pauses

**set DISABLE\_APPS\_FORCING\_GC=-XX:+DisableExplicitGC**

* Finally append these last two properties to the ARGS variable in *jmeter.bat* file:

**set ARGS**=%DUMP% %HEAP% … %**CMS\_GC**% %**DISABLE\_APPS\_FORCING\_GC**%

* Run the JMeter testcase in **Non-GUI** mode (as in the previous steps) and collect the results in a log file:

**jmeter -n -t VisualizeDevelopers-100threads.jmx -l VisualizeDevelopers-100threads-Results.jtl**

* Add the end of the test, open the JMeter GUI and add the "Simple Data Writer" listener (specifying also the path to collected results file). Moreover, any output listener might be added to the testplan, inputting the path to the results file and then visually evaluating performance features.

Results:

* Despite of the configuration performed when running the JMeter testcase in Non-GUI mode (from the command line) with 100 threads, no execution enhancement was obtained. Therefore, running the test which output the results in the .jtl log file, took about 3h.

**Step 6**:

* Run the JMeter test in GUI mode by using an alternative method for generating results: [**BlazeMeter's plugin to JMeter**](http://www.google.com/url?q=http%3A%2F%2Fcommunity.blazemeter.com%2Fknowledgebase%2Farticles%2F83191-blazemeter-plugin-to-jmeter&sa=D&sntz=1&usg=AFQjCNGT5UkJ2zTqdN8AyKMkxFqLmIxAWA) **(**<https://www.blazemeter.com/blog/how-reduce-memory-usage-jmeter?utm_source=Blog&utm_medium=BM_Blog&utm_campaign=OOMFailure>). The advantages offered by this method (as compared to the previous ones) are:
  + The report will be generated automatically on a cloud server (no need to save results into file).
  + Results might be visualized during test execution (no need to wait for the end of test).
  + Easy to use report charts are available.
* In order to use this plugin, the *blazemeter.jar* was downloaded and added to the *lib/ext* folder of JMeter. The plugin can be used in two modes: *Locally* and *Cloud* mode. While *Cloud* mode is for the launching of performance tests on the cloud, the *Locally* mode is totally free.
* To use the plugin in JMeter after installation, the **Blazemeter** listener was added at the end of the test and the following configurations performed:
  + Insert the User Key (from the BlazeMeter account settings).
  + Press the plus icon to create a new test in the Cloud and insert its name in the displayed pop-up.
  + The given test name and a generated id are automatically inserted in the **Test Info** field.
  + Select the **Locally (Reporting Only)** running mode.

Results:

* As the BlazeMeter plugin is freely available in its Trial version, the JMeterEngine size used is the SMALL one, with limited performance capabilities and features. Moreover, the Trial/Free-Tier BlazeMeter settings provide only one thread per engine to be used in test execution, which is **not appropriate to the context and target of our performance test** (simulate the use case execution with a large number of concurrent threads – say 100 – and perform the required configurations in order to reduce the total response time and maximize the performance).
* Also, the **test results provided in the BlazeMeter’s Cloud, never get generated** and, instead, the progress generation is continuously displayed to the user.

**Step 7**:

* Modify the configuration parameters for the ***findAllDevelopersCache*** region defined in *ehcache.xml* file:
  + Increase the value of the ***maxEntriesLocalHeap*** property from 1000 to ***2000***.
  + Increase the value of the ***maxEntriesLocalDisk*** property from 1000 to ***10000000***.
  + Add the ***maxElementsInMemory = “2000”*** property to the cache region.
* Run the JMeter testcase in **Non-GUI** mode (as in the previous steps) and collect the results in a log file:

**jmeter -n -t VisualizeDevelopers-100threads.jmx -l VisualizeDevelopers-100threads-Results.jtl**

* Add the end of the test, open the JMeter GUI and add the "Simple Data Writer" listener (specifying also the path to collected results file). Moreover, any output listener might be added to the testplan, inputting the path to the results file and then visually evaluating performance features.

Results:

* Despite of the configuration performed to the cache region’s definition (in *ehcache.xml* file), running the JMeter testcase in Non-GUI mode (from the command line) with 100 threads offered no execution enhancement. Therefore, running the test which output the results in the .jtl log file, took even more than 3h.

As a result of the **7 steps** performed in order to **remove the *OutOfMemory****: Java Heap space* error obtained when running the JMeter testcase with 100 threads and **show the performance improvements brought by Spring eh-cache** (by obtaining an execution time reduction as compared to the non-cached version), the following conclusions might be stated:

* The ***OutOfMemory****: Java Heap space* error was completely removed
* However, none of the configurations performed in the 7 steps were of help in reducing the testplan’s execution time. Therefore, for 100 concurrent threads, the testplan’s execution takes between 2 and 3 hours. As no valuable solutions to reduce this time amount were encountered, the effort to prove the performance improvements brought by the implemented **Spring eh-cache** was ***resumed***.

**Next**: implement caching using **Hibernate eh-cache** for a Java entity class (from Model layer) and test the performance obtained:

<http://www.journaldev.com/2980/hibernate-ehcache-second-level-caching-example-tutorial>

<http://howtodoinjava.com/hibernate/hibernate-ehcache-configuration-tutorial/>

<http://www.ehcache.org/documentation/2.8/integrations/hibernate.html>

<http://www.gridshore.nl/2008/04/29/using-ehcache-and-verifying-that-it-works-with-jpa-and-springframework/>

<http://www.tutorialspoint.com/hibernate/hibernate_caching.htm>

The **Hibernate Eh-Cache** was performed under the same use case as in the previous research conducted on behalf of the developed Spring Eh-Cache: **Login\_Visualize developers\_Visualize taskboard\_Add task\_Visualize developers\_Logout**. This scenario is performed by an admin user (say Scrum Master) who logins into the application, visualizes all developers, then visualizes the taskboard, adds a new task (together with the required information for it), visits again the list of developers and the logs out from the application.

In order to evaluate the performance enhancements obtained through a series of metrics, JMeter tool will be used to simulate the execution of this use case for many concurrent users retrieving the developers. The desired behavior of the implemented cache is to access the database only once (on first call) and then retrieve the developers’ information from the cache for the successive calls of the find developers functionality.

The main purpose of the developed Hibernate Eh-Cache is to improve the performance of the application when executing the **Login\_Visualize developers\_Visualize taskboard\_Add task\_Visualize developers\_Logout** testcase in the context of a larger number of concurrent users. More precisely, as no enhancement in the testcase’s execution time was obtained from the configurations performed on behalf of the Spring Eh-cache implemented for the functionality to **retrieve all the developers from the database**, the following step was to develop Hibernate Eh-Cache in hope of improving the long-running response time (between 2 and 3 hours for a number of 100 concurrent threads). Therefore, the **Hibernate Eh-Caching** was implemented and properly configured for the **Developer** entity in order to cache the database information corresponding to each developer and retrieve it from the cache on successive accesses of the same instance.

Moreover, in order to see considerable performance improvements offered by the Hibernate Eh-Caching implemented, the data being cached (all Developer instances), and further on accessed, has to be of *large amount*. Therefore, all performance tests were executed in the context of *1000 developers* existent in the database. However, as this list is retrieved twice for each thread in the selected use case scenario (**Login\_Visualize developers\_Visualize taskboard\_Add task\_Visualize developers\_Logout**), the simulated number of users (threads) from JMeter *cannot* be very large (like hundreds or thousands) because this *consumes all of the CPU’s activity and takes a large amount of time* to complete. As a result, the tested use case was executed in the context of **100 concurrent users**.

The **main steps in the implementation of Hibernate Eh-cache** are:

* Add the required **Maven** **dependencies** in *pom.xml*:

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>${hibernate.version}</version>

</dependency>

<dependency>

<groupId>net.sf.ehcache</groupId>

<artifactId>ehcache-core</artifactId>

<version>${ehcache-core.version}</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-ehcache</artifactId>

<version>${hibernate.version}</version>

</dependency>

* Add **Hibernate Eh-cache configurations** in *persistence.xml* file:

<property name="**hibernate.cache.region.factory\_class**" value="*org.hibernate.cache.ehcache.EhCacheRegionFactory*"/>

<property name="**hibernate.cache.use\_second\_level\_cache**" value="*true*"/>

<property name="**net.sf.ehcache.configurationResourceName**" value="*/hibernate-ehcache.xml*"/>

where:

**hibernate.cache.region.factory\_class** - is used to define the Factory class for Second level caching

**hibernate.cache.use\_second\_level\_cache** - is used to enable the second level cache

**net.sf.ehcache.configurationResourceName** - is used to define the Eh-Cache configuration file location. By default, Eh-Cache will create separate cache regions for each entity that you configure for caching. You can change the defaults for these regions in your customized *ehcache.xml*, specified as **net.sf.ehcache.configurationResourceName** property in hibernate configuration file.

* Define the **Eh-Cache configuration** in */hibernate-ehcache.xml* file in order to override the default one:

<ehcache xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:noNamespaceSchemaLocation=*"ehcache.xsd"* updateCheck=*"true"*

monitoring=*"autodetect"* dynamicConfig=*"true"*>

<cache name=*"****developer****"*

maxEntriesLocalHeap=*"10000"* maxElementsInMemory=*"10000"* eternal=*"false"*

timeToIdleSeconds=*"3000"* timeToLiveSeconds=*"6000"*>

<persistence strategy=*"localTempSwap"*/>

</cache>

</ehcache>

Through the **cache** element, regions are defined and configured. Therefore, a cache region with “**developer**” name was created and its attributes customized.

* Add ***@Cache* annotation on the model entity** class to be cached (*Developer.java*):

@Entity

@Table(name="developer")

**@Cache**(usage=**CacheConcurrencyStrategy.READ\_ONLY**, region="**developer**")

public class Developer {

...

}

The **@Cache** annotation is used to provide the caching configuration and allows us to define the **caching strategy** and **region** to use for the model entity. As the use case scenario under test only needs to read, but not modify, instances of the Developer class, the **read-only** cache strategy was used. Moreover, the mapping to the cache region defined in *hibernate-ehcache.xml* file was done through the **region** attribute of the **@Cache** annotation.

**Note:** As the **Hibernate Eh-Cache configuration** was defined in ***persistence.xml*** (and not in a ***hibernate.cfg.xml*** file, as usually done), there is **no need of developing configuration loading** (through Java code), because in **Spring** the context, and therefore ***persistence.xml*** configurations, are automatically loaded at server start-up.

The proper configuration of the Hibernate Eh-Cache for the Developer entity is also logged at server start-up:

18:07:21,979 INFO localhost-startStop-1 util.LogHelper:46 - HHH000204: Processing PersistenceUnitInfo [name: JpaPersistenceUnit...]

18:07:22,109 INFO localhost-startStop-1 hibernate.Version:54 - HHH000412: Hibernate Core {4.3.11.Final}

18:07:22,109 INFO localhost-startStop-1 cfg.Environment:239 - HHH000206: hibernate.properties not found

18:07:22,109 INFO localhost-startStop-1 cfg.Environment:346 - HHH000021: Bytecode provider name : javassist

18:07:22,459 INFO localhost-startStop-1 common.Version:66 - HCANN000001: Hibernate Commons Annotations {4.0.5.Final}

18:07:22,769 INFO localhost-startStop-1 dialect.Dialect:145 - HHH000400: Using dialect: org.hibernate.dialect.MySQLDialect

18:07:23,069 INFO localhost-startStop-1 ast.ASTQueryTranslatorFactory:47 - HHH000397: Using ASTQueryTranslatorFactory

18:07:23,320 INFO localhost-startStop-1 util.Version:27 - HV000001: Hibernate Validator 5.1.3.Final

18:07:23,780 WARN localhost-startStop-1 **strategy.EhcacheAccessStrategyFactoryImpl**:57 - HHH020007: **read-only cache configured for mutable entity [developer]**

18:07:23,980 INFO localhost-startStop-1 hbm2ddl.SchemaExport:344 - HHH000227: Running hbm2ddl schema export

18:07:28,376 INFO localhost-startStop-1 hbm2ddl.SchemaExport:406 - HHH000230: Schema export complete

Jul 15, 2016 6:07:30 PM org.apache.catalina.startup.Catalina start

INFO: Server startup in 12708 ms

**Evaluate results:**

The next step is to assess the performance improvements obtained in the testcase’ execution time. Therefore, the JMeter testcase was run in **GUI** mode with only 2 Listeners (*Summary Report* and *View Results Tree*) attached to the testplan and the execution time took about **2 hours and a half**. Afterwards, the testcase was run in **Non-GUI** mode with **no Listener** added to the testplan and the total execution time took **more than 3 hours**. Therefore, **no performance enhancements** in the testcase’s execution time were obtained by the implemented Hibernate Eh-Cache.

TODO: Hibernate Eh-Cache => performance improvements?

For the new cache config (in ehcache.xml):

<cache name=*"developer"*

maxEntriesLocalHeap=*"10000"* maxElementsInMemory=*"100000"* maxElementsOnDisk=*"1"*

eternal=*"false"*

timeToIdleSeconds=*"3600"* timeToLiveSeconds=*"3600"* statistics=*"true"*>

<persistence strategy=*"none"*/>

</cache>

* For 100 threads: Java Heap Space error + large execution time (~ 3h)
* For 50 threads: NO Java Heap Space error and large execution time (~2h)

Next step was to investigate if the implemented hibernate caching was actually used: by means of cache statistics (monitored from VisualVM). The display of cache statistics was enabled by the following configurations:

* Specifying the *<property name="hibernate.generate\_statistics" value="true" />* in persistence.xml file.
* Adding the *statistics="true"* attribute for the cache region defined in ehcache.xml file.
* Specifying the registering of CacheManager bean through Java code:

*CacheManager manager = CacheManager.newInstance();*

*MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();*

*ManagementService.registerMBeans(manager, mBeanServer, true, true, true, true);*

After enabling the display of cache statistics, the JMeter tests were re-executed and the cache statistics offered by VisualVM tool were analyzed. Therefore, the following conclusions might be stated:

* The developed hibernate eh-cache was identified (together with the configured attributes)
* NO entry is taken from the cache => cache **not** used

+ logging cache information in the tomcat’s console

Investigating this problem in order to determine the **reason for which the developed Hibernate Eh-Cache is not used** when accessing the cached functionality (retrieve the list of developers). The root cause of the not-applied Hibernate Eh-Cache was the fact that the SQL queries run against the database were not cached. More precisely, in the configuration for the developed Hibernate Eh-Cache, only the second level cache was enabled, but the property to enable the query cache was not. Therefore, the **solution to determine the application to take advantage of the Hibernate Eh-Cache developed for the retrieval developers functionality** consists of the following steps:

* Specify the property to activate query cache, without it HQL queries results will not be cached (in persistence.xml file):

*<property name="hibernate.cache.use\_query\_cache" value="true"/>*

* Set the enable caching property for the particular SQL query to be cached in the persistence-lever (Data Access Object) file (*JpaDeveloperDao.java*):

*Query query = entityManager.createQuery("select d from Developer d")*

*.setHint("org.hibernate.cacheable",* ***true****);*

* Moreover, a fine-grained configuration might be specified for the SQL query to be cached by naming the particular cache region to be applied (the one configured in *ehcache,xml* file):

*Query query = entityManager.createQuery("select d from Developer d")*

*.setHint("org.hibernate.cacheable",* ***true****)*

*.setHint("org.hibernate.cacheRegion", "developer");*

* Even if the particular cache region to be used for specified for the SQL query, an exception is thrown when the application context is started if a default cache region is not defined in the *ehcache.xml* configuration file. Therefore, in order to fix this error, a default cache region was added:

*<defaultCache*

*maxElementsInMemory="10000" maxElementsOnDisk="1" eternal="false"*

*timeToIdleSeconds="3600" timeToLiveSeconds="3600" statistics="true">*

*<persistence strategy="none" />*

*</defaultCache>*

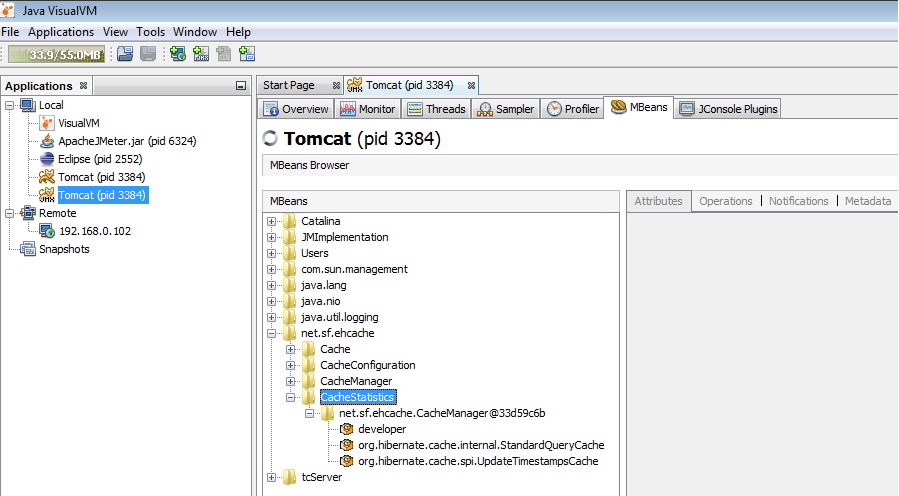
As a result of the configured query cache for the ***developer*** region, the following statements are logged at Tomcat server start-up:

*localhost-startStop-1 strategy.EhcacheAccessStrategyFactoryImpl:57 - HHH020007:* ***read-only cache configured for mutable entity [developer]***

*tomcat-http--14 internal.StandardQueryCache:90 - HHH000248:* ***Starting query cache at region: developer***

**Evaluate results:**

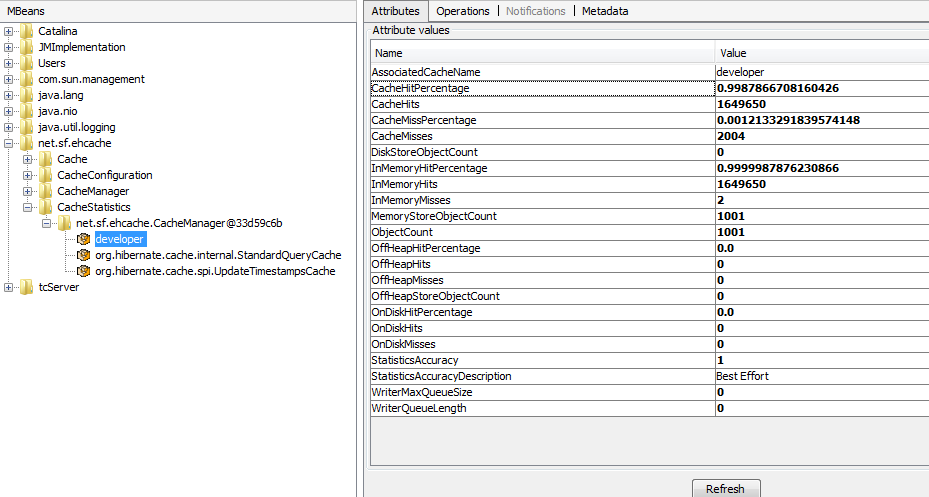
The properly configured and used Hibernate Eh-Cache for the developers retrieval functionality is assessed though the VisualVM tool by monitoring the **net.sf.ehcache** results listed under the **MBeans** tab. More precisely, the number of cache hits/misses and associated percentage values for different metrics related to the cache usage are displayed for each configured cache region. In the actual context, 3 caches are identified in the application: **developer**, org.hibernate.cache.internal.**StandardQueryCache** and org.hibernate.cache.spi.**UpdateTimestampsCache**:

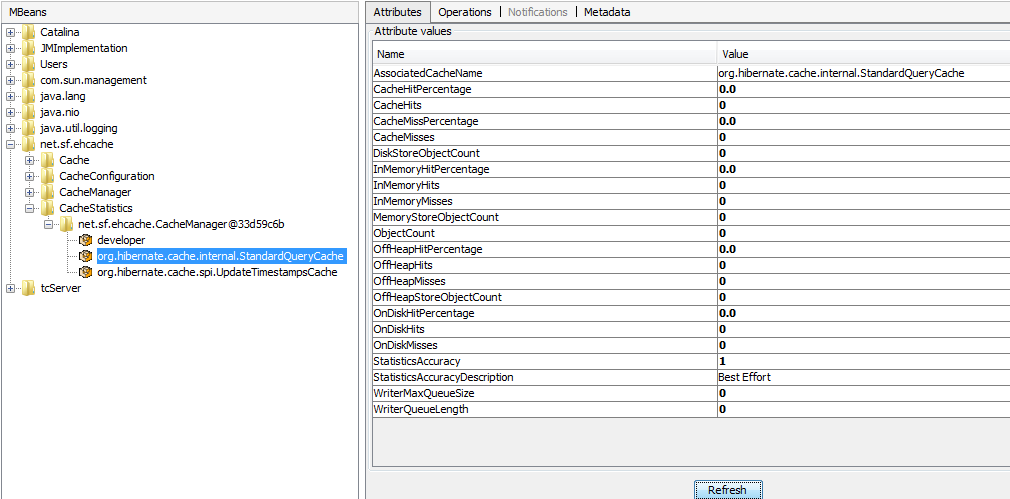


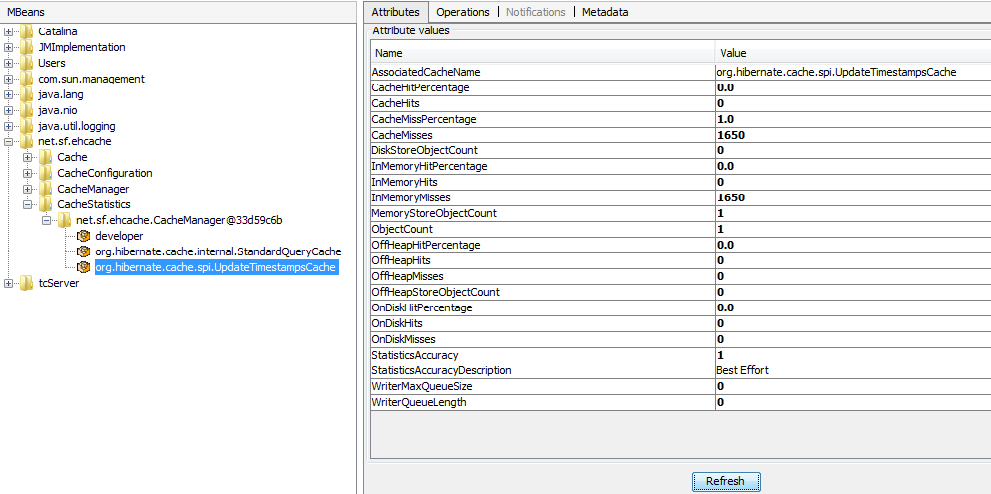
The JMeter test was executed again in **GUI mode** (with only *View Results Tree* listener attached to the testplan) and the following results are obtained for a number of **100 concurrent threads**:

* The execution time took **2 hours and 20 minutes**.
* Also, Java Heap Space errors were reported in the JMeter console.

Meanwhile running the JMeter testplan, the application was monitored using the VisualVM tool in order to collect statistical information related to the configured Second-Level Hibernate Eh-Cache. At the end of the JMeter testplan execution (run in GUI mode with 100 concurrent threads), the information associated to cache statistics were gathered and analyzed. Consequently, for the 3 caches listed in the MBeans tab, the following cache statistics were obtained:







By analyzing the statistical cache information, it can be easily figured out that **only** the ***developer*** cache is used by the application. This is due to the fact that both configured Hibernate Eh-Caches (the Second-Level one, implemented for the Developer entity and the Query-Level one, used when executing the SQL query to find all developers in the database) use the ***developer*** cache region defined in *ehcache.xml* configuration file. Therefore, all accesses of cached data from the ***developer*** cache region (for Second-Level or Query-Level caches) are saved in the statistics corresponding to the ***developer*** cache. The high scores for *CacheHits* and *InMemoryHits* and low ones for *CacheMisses* and *InMemoryMisses* (as well as their associated percentages) show a high usage of the cached data stored in the ***developer*** region. As a result, the actual goal of the developed Hibernate Eh-Cache was reached: when accessing data related to developers, the cached information is retrieved instead of invoking the database for each request.

However, **despite of the encouraging statistical results** related to the Hibernate Eh-Cache usage, **no performance enhancement** in the JMeter testplan’s execution time was obtained.

Several investigations were carried out in order to determine the root cause of the *OutOfMemory: Java Heap space* error repored when executing the JMeter testplan with 100 treads. This error was removed by adjusting memory-specific attributes in the *jmeter.bat* file:

* From set HEAP=-Xms2048m –Xmx**8192**m to set HEAP=-Xms2048m -Xmx**4096**m
* From set NEW=-XX:NewSize=**128**m -XX:MaxNewSize=**128**m to set NEW=-XX:NewSize=**256**m -XX:MaxNewSize=**512**m
* From set PERM=-XX:PermSize=**64**m -XX:MaxPermSize=**128**m to set PERM=-XX:PermSize=**256**m -XX:MaxPermSize=**512**m

The JMeter testplan was run again in GUI mode with 100 threads and the *OutOfMemory: Java Heap space* was no reported anymore but the total execution time took **3hours and 5 minutes**. Again, **no performance enhancement** in the JMeter testplan’s execution time was obtained.

**Cluster**

Cluster Configuration: Apache server with 2 tomcat servers.

The following configuration was specified in httpd-vhosts.conf file from Apache.

*<VirtualHost \_default\_:80>*

*DocumentRoot "${SRVROOT}/htdocs"*

*ServerName scrumtaskboard.localhost.com*

*Header add Set-Cookie "ROUTEID=.%{BALANCER\_WORKER\_ROUTE}e; path=/" env=BALANCER\_ROUTE\_CHANGED*

*<Proxy balancer://testcluster>*

*BalancerMember http://127.0.0.1:8080/ScrumTaskboard route=node1*

*BalancerMember http://127.0.0.1:8081/ScrumTaskboard route=node2*

*#BalancerMember http://192.168.0.102:8080 min=20 max=200 route=node3 loadfactor=1*

*ProxySet stickysession=ROUTEID*

*ProxySet lbmethod=byrequests*

*</Proxy>*

*ProxyPass /ScrumTaskboard/ balancer://testcluster/ timeout=6000*

*ProxyPassReverse /ScrumTaskboard/ balancer://testcluster/ScrumTaskboard*

*ProxyRequests off*

*<Location /balancer-manager>*

*SetHandler balancer-manager*

*AuthType Basic*

*AuthName "Balancer Manager"*

*AuthUserFile "C:/Apache24/conf/.htpasswd"*

*Require valid-user*

*</Location>*

*</VirtualHost>*

The JMeter testplan for the **Login\_Visualize developers\_Visualize taskboard\_Add task\_Visualize developers\_Logout** scenario was run with 100 threads and the execution time took 2h and 5 mins:

2016/08/07 03:33:10 INFO - jmeter.engine.StandardJMeterEngine: Running the test!

2016/08/07 03:33:10 INFO - jmeter.samplers.SampleEvent: List of sample\_variables: []

2016/08/07 03:33:10 INFO - jmeter.samplers.SampleEvent: List of sample\_variables: []

2016/08/07 03:33:10 INFO - jmeter.gui.util.JMeterMenuBar: setRunning(true,\*local\*)

2016/08/07 03:33:10 INFO - jmeter.engine.StandardJMeterEngine: Starting ThreadGroup: 1 : Thread Group

2016/08/07 03:33:10 INFO - jmeter.engine.StandardJMeterEngine: Starting 100 threads for group Thread Group.

2016/08/07 03:33:10 INFO - jmeter.engine.StandardJMeterEngine: Thread will continue on error

2016/08/07 03:33:10 INFO - jmeter.threads.ThreadGroup: Starting thread group number 1 threads 100 ramp-up 1 perThread 10.0 delayedStart=false

2016/08/07 03:33:10 INFO - jmeter.threads.JMeterThread: jmeterthread.startearlier=true (see jmeter.properties)

2016/08/07 03:33:10 INFO - jmeter.threads.JMeterThread: Running PostProcessors in forward order

2016/08/07 03:33:10 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-2

2016/08/07 03:33:10 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-1

2016/08/07 03:33:10 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-3

2016/08/07 03:33:10 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-6

......................................................................................

2016/08/07 05:39:41 INFO - jmeter.threads.JMeterThread: Thread finished: Thread Group 1-16

2016/08/07 05:39:41 INFO - jmeter.engine.StandardJMeterEngine: Notifying test listeners of end of test

2016/08/07 05:39:41 INFO - jmeter.gui.util.JMeterMenuBar: setRunning(false,\*local\*)

As no significant response time improvements were obtained by this Cluster configuration, a more suitable approach is to distribute the requests’ load on 2 tomcat servers located on different physical machines.

Unable to connect to the DB of the first laptop from the second one.

Solution:

* In my.ini MySql configuration file, add:

*bind-address = 0.0.0.0* (to allow connections to MySQL Server from remote hosts, not only from localhost)

* Grant priviledges to user *root*, identified by *password*, from all hosts (*@'%'*)

*GRANT ALL PRIVILEGES ON \*.\* TO 'root'@'%' IDENTIFIED BY 'password' WITH GRANT OPTION;*

*FLUSH PRIVILEGES;*

The following cluster configuration was implemented in Apache server’s *httpd-vhosts.conf* file:

<VirtualHost \_default\_:80>

DocumentRoot "${SRVROOT}/htdocs"

ServerName scrumtaskboard.dissertation.com

Header add Set-Cookie "ROUTEID=.%{BALANCER\_WORKER\_ROUTE}e; path=/" env=BALANCER\_ROUTE\_CHANGED

<Proxy balancer://testcluster>

**BalancerMember http://127.0.0.1:8080/ScrumTaskboard route=node1**

**BalancerMember http://192.168.0.102:8080/ScrumTaskboard route=node3**

ProxySet stickysession=ROUTEID

ProxySet lbmethod=byrequests

</Proxy>

ProxyPass /ScrumTaskboard/ balancer://testcluster/ timeout=6000

ProxyPassReverse /ScrumTaskboard/ balancer://testcluster/ScrumTaskboard

ProxyRequests off

<Location /balancer-manager>

SetHandler balancer-manager

AuthType Basic

AuthName "Balancer Manager"

AuthUserFile "C:/Apache24/conf/.htpasswd"

Require valid-user

</Location>

</VirtualHost>

The above configuration shows an Apache load balancer which distributes the requests among two tomcat server nodes: one on the localhost (**127.0.0.1 -** same computer where the JMeter testplan was run) and another one belonging to a remote host (**192.168.0.102**). Both tomcat servers run on behalf of the same code base and used the same database stored in the localhost machine.

The JMeter testplan for the **Login\_Visualize developers\_Visualize taskboard\_Add task\_Visualize developers\_Logout** scenario was run with 100 threads and the execution time took 3h and 8 mins:

2016/08/14 05:59:57 INFO - jmeter.gui.util.JMeterMenuBar: setRunning(true,\*local\*)

2016/08/14 05:59:57 INFO - jmeter.engine.StandardJMeterEngine: Starting ThreadGroup: 1 : Thread Group

2016/08/14 05:59:57 INFO - jmeter.engine.StandardJMeterEngine: Starting 100 threads for group Thread Group.

2016/08/14 05:59:57 INFO - jmeter.engine.StandardJMeterEngine: Thread will continue on error

2016/08/14 05:59:57 INFO - jmeter.threads.ThreadGroup: Starting thread group number 1 threads 100 ramp-up 1 perThread 10.0 delayedStart=false

2016/08/14 05:59:57 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-1

2016/08/14 05:59:57 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-2

2016/08/14 05:59:57 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-3

2016/08/14 05:59:57 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-4

2016/08/14 05:59:57 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-5

2016/08/14 05:59:57 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-6

2016/08/14 05:59:57 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-7

2016/08/14 05:59:57 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-8

2016/08/14 05:59:57 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-9

2016/08/14 05:59:57 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-10

2016/08/14 05:59:57 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-11

2016/08/14 05:59:57 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-13

2016/08/14 05:59:57 INFO - jmeter.threads.ThreadGroup: Started thread group number 1

2016/08/14 05:59:57 INFO - jmeter.engine.StandardJMeterEngine: All thread groups have been started

....

2016/08/14 09:07:44 INFO - jmeter.threads.JMeterThread: Thread is done: Thread Group 1-33

2016/08/14 09:07:44 INFO - jmeter.threads.JMeterThread: Thread finished: Thread Group 1-33

2016/08/14 09:07:44 INFO - jmeter.engine.StandardJMeterEngine: Notifying test listeners of end of test

2016/08/14 09:07:44 INFO - jmeter.gui.util.JMeterMenuBar: setRunning(false,\*local\*)

As no significant response time improvements were obtained by this Cluster configuration, the following step is to add a third tomcat server to the actual configuration:

<VirtualHost \_default\_:80>

DocumentRoot "${SRVROOT}/htdocs"

ServerName scrumtaskboard.dissertation.com

Header add Set-Cookie "ROUTEID=.%{BALANCER\_WORKER\_ROUTE}e; path=/" env=BALANCER\_ROUTE\_CHANGED

<Proxy balancer://testcluster>

**BalancerMember http://127.0.0.1:8080/ScrumTaskboard route=node1**

**BalancerMember http://127.0.0.1:8081/ScrumTaskboard route=node2**

**BalancerMember http://192.168.0.102:8080/ScrumTaskboard route=node3**

ProxySet stickysession=ROUTEID

ProxySet lbmethod=byrequests

</Proxy>

...

</VirtualHost>

Therefore, the new cluster setup will consist of an Apache load balancer which distributes the requests among **three** tomcat server nodes: two on the localhost (**127.0.0.1** on ports **8080 and 8081 -** same computer where the JMeter testplan was run) and another one belonging to a remote host (**192.168.0.102**). All tomcat servers run on behalf of the same code base and used the same database stored in the localhost machine.

The JMeter testplan for the **Login\_Visualize developers\_Visualize taskboard\_Add task\_Visualize developers\_Logout** scenario was run with 100 threads and the execution time took **1h** and **37mins**:

2016/08/15 06:52:56 INFO - jmeter.gui.util.JMeterMenuBar: setRunning(true,\*local\*)

2016/08/15 06:52:56 INFO - jmeter.engine.StandardJMeterEngine: Starting ThreadGroup: 1 : Thread Group

2016/08/15 06:52:56 INFO - jmeter.engine.StandardJMeterEngine: Starting 100 threads for group Thread Group.

2016/08/15 06:52:56 INFO - jmeter.engine.StandardJMeterEngine: Thread will continue on error

2016/08/15 06:52:56 INFO - jmeter.threads.ThreadGroup: Starting thread group number 1 threads 100 ramp-up 1 perThread 10.0 delayedStart=false

2016/08/15 06:52:56 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-1

2016/08/15 06:52:56 INFO - jmeter.threads.JMeterThread: Thread started: Thread Group 1-2

....

2016/08/15 08:29:47 INFO - jmeter.threads.JMeterThread: Thread finished: Thread Group 1-44

2016/08/15 08:30:01 INFO - jmeter.threads.JMeterThread: Thread is done: Thread Group 1-54

2016/08/15 08:30:01 INFO - jmeter.threads.JMeterThread: Thread finished: Thread Group 1-54

2016/08/15 08:30:01 INFO - jmeter.engine.StandardJMeterEngine: Notifying test listeners of end of test

2016/08/15 08:30:01 INFO - jmeter.gui.util.JMeterMenuBar: setRunning(false,\*local\*)

The obtained result represents a **significant improvement** as the total response time was reduced with about **30 mins** as compared to the best case scenario before setting up the load balancer (~ 2h and 5 mins).