



## **Geospatial Database Administration**

**Dylan Rollicheck, California State University, Monterey Bay**

*Mentor: Mike McCann*

*Summer 2013*

**Keywords: Postgresql, pgbench, pgbench-tools, STOQS, geospatial database**

### **ABSTRACT**

Quick and easy access to growing collections of data is important for understanding the ocean. STOQS (Spatial Temporal Oceanographic Query System) is a geospatial database tool designed to retrieve different types of data and display them in an easy to use web application. In some instances, the application was taking a long time to retrieve and display the data, so a new dedicated database server was purchased. Along with buying a new more powerful dedicated database server, the configuration of postgresql was looked at, benchmarking was done using pgbench and pgbench-tools and logs were examined using the log analyzer pgfouine.



Figure 1: The inside of Kraken, 128GB RAM, 4x 146GB 15K SAS hard drives, 365GB SSD FusionIO card.

## INTRODUCTION

Postgresql 9.1 is an open source object-relational database system, which can also use geospatial data with postgis installed. Postgresql has around 200 parameters that can be changed in the postgresql.conf file (figure 2) to fit different needs and make it run more efficiently.

Pgbench is a simple benchmarking tool that runs a sequence of the same structured query language (SQL) commands multiple times to produce the average transactions per second (TPS). The test run is a simulation of a bank transaction, using branches, tellers and accounts. There are six different scripts that come with the pgbench install that can be used to run different types of tests; insert.sql and nobranch.sql are examples of the scripts. Pgbench also allows the use of custom scripts, so any test that can be written can be used with it.

Pgbench-tools makes automated running of pgbench with different database sizes (scale) and various numbers of clients possible. It also can be used to produce multiple graphs based on the results of the test; these graphs are made in html format so the results can easily be accessed if a webserver is configured to serve them.

Pgfouine is a log-analyzing tool that reads the log files of postgresql and produces a detailed report about the queries executed. The reports show which queries are run most often, take the longest, and overall statistics about the queries. Pgfouine also outputs these results into html format, so they can easily be displayed in a web browser.

```

#
# - Memory -
shared_buffers = 1GB           # min 128kB
                                # (change requires restart)
#temp_buffers = 8MB           # min 800kB
#max_prepared_transactions = 0 # zero disables the feature
                                # (change requires restart)
# Note: Increasing max_prepared_transactions costs ~600 bytes of shared memory
# per transaction slot, plus lock space (see max_locks_per_transaction).
# It is not advisable to set max_prepared_transactions nonzero unless you
# actively intend to use prepared transactions.

# Changed from 1 MB to 400MB to support large STOPS sorts - mpm 22 May 2013
work_mem = 400MB               # min 64kB
#maintenance_work_mem = 16MB  # min 1MB
#max_stack_depth = 2MB         # min 100kB

# - Kernel Resource Usage -
#max_files_per_process = 1000  # min 25
                                # (change requires restart)
#shared_preload_libraries = '' # (change requires restart)

# - Cost-Based Vacuum Delay -
#vacuum_cost_delay = 0ms       # 0-100 milliseconds
#vacuum_cost_page_hit = 1      # 0-10000 credits
#vacuum_cost_page_miss = 10    # 0-10000 credits
#vacuum_cost_page_dirty = 20   # 0-10000 credits
#vacuum_cost_limit = 200       # 1-10000 credits

```

Figure 2: Example of what the postgresql.conf file look likes using vi postgresql.conf command, specifically the memory section.

## MATERIALS AND METHODS

### OVERVIEW

To figure out which combination of settings would be best for Kraken, the new dedicated server that has hardware specifically configured to run PostgreSQL, I started out by reading up on postgresql 9.1. [1] After learning a lot about how certain settings make a difference in different areas, I started tuning some of the postgres.conf settings on a virtual machine. The virtual machine was run inside a MacBook Pro using VMware Fusion with Centos 6 as the operating system. Once I made changes to the postgres.conf file I ran pgbench test using pgbench-tools and looked at the how those changes affected the TPS. After seeing which settings worked best I made those same changes to Pgtest, another virtual machine in MBARI's VMware ESX cluster, used as a testing platform before changes are made on production machines. Then pgbench-tools was used on Pgtest to produce more TPS graphs and see if the changes made the same impact on Pgtest as they did on the other virtual machine. Once the Pgtest configuration was working correctly the same configuration changes were made to Kraken. Since Kraken is a

more powerful machine than the other virtual machines are and is a production machine, some adjustments had to be made to how the programs interact with the server in order to make the same configuration changes. Most of the adjustments had to do with the added security a production machine has and others had to do with Kraken being able to use its 128 GB of RAM. The same pgbench tests were run on Kraken to see how the changes affected the TPS and pgfouine is being used to find queries that STOQS might be executing inefficiently.

```
-bash-4.1$ diff postgresql.conf.130716 postgresql.conf 59c59,60
< #listen_addresses = 'localhost'          # what IP
address(es) to listen on;
---
> # Allow connections from mbari hosts inside our firewall - mpm 7
May 2013
> listen_addresses = '*'                    # what IP
address(es) to listen on; 64c65,66
< max_connections = 100                    # (change requires
restart)
---
> # Changed max_connections from 100 to 10 corresponding to
increase in work_mem - mpm 22 May 2013
> max_connections = 10                    # (change requires
restart) 109c111
< shared_buffers = 32MB                   # min 128kB
---
> shared_buffers = 3GB                     # min 128kB 118c120,122
< #work_mem = 1MB                         # min 64kB
---
> # Changed from 1 MB to 400MB to support large STOQS sorts - mpm
22 May 2013
> work_mem = 400MB                        # min 64kB 174c178,179
```

Figure 3: A diff of the postgresql.conf file on Kraken, showing some of the changes made.

## USING PGBENCH & PGBENCH-TOOLS

Pgbench and pgbench-tools are not included in the default postgresql package and have to be installed on top of the default package. In order to use pgbench just run the command “*sudo yum install postgres-contrib*”, which will install the postgresql extension package that has pgbench inside. In order to install pgbench-

tools use the command, *git clone git://git.postgresql.org/git/pgbench-tools.git*. [3] Also in order for pgbench-tools to be able to produce graphs from the pgbench results, gnuplot had to be installed using the command, “*sudo yum install gnuplot*”.

To find where TPS dropped off as a function of database scale, multiple pgbench test had to be run on each machine. Pgbench-tools helped with this because it uses a script to run multiple tests in sequence. I would adjust the pgbench-tools config file so it would run pgbench test with a range of scales, clients, and run the same test a set number of times, figure 4, when the command *./runset* was executed. Starting with a wide range of scales gave an overview of where the TPS drop off would happen and then using a smaller scale range, within the general drop off area, the exact scale where the drop off occurs was found.

```
# These values are for pgbench 8.4 and later
TABLEPREFIX="pgbench_"
TESTDIR="tests"

# Use these values for pgbench 8.3
#TABLEPREFIX=""
#TESTDIR="tests-8.3"

# SKIPINIT should be set to 1 either when simulating a cold cache, or
# if you are not using the pgbench tables for your test
SKIPINIT=0

# Test/result database connection
TESTHOST=localhost
TESTUSER=postgres
TESTPORT=5432
TESTDB=pgbench

RESULTHOST=$TESTHOST
RESULTUSER=$TESTUSER
RESULTPORT=$TESTPORT
RESULTDB=results

#
# Main test run customization
#

# Set this to a number only when using pgbench 9.0 or later. This will set
# the number of worker threads up to this maximum for each client count
MAX_WORKERS="1"

SCRIPT="select.sql"
SCALES="1 100"
SETCLIENTS="1"
SETTIMES=3

# RUNTIME is in seconds. If it is specified, then TOTTRANS is ignored.
# Setting a time instead of a transaction account is only available
# in 8.4 and later.
RUNTIME=60
TOTTRANS=""

# Use a transaction count instead for 8.3 or earlier
#RUNTIME=""
#TOTTRANS=100000
```

Figure 4: Pgbench-tools config file that contains the settings for what scales, number of clients and number of times a test is run when *./runset* command is used.

## USING PGFOUINE

Like pgbench and pgbench-tool, pgfouine didn't come with postgresql and had to be installed separately using the command "*sudo yum install pgfouine*". In order to get pgfouine to work properly I had to make some changes to what information was being logged, two of the changes are visible in figure 5. Also the value of date.timezone in the php.ini file needed to be changed from the default value to America/Los\_Angeles, to avoid errors in the program. [4] Pgfouine is mainly aimed to help find which queries are taking the longest to run and which queries are being run the most when using the STOQS application. These queries are interesting because they can help find other places that might need change, like how some of the queries are written.

```
-bash-4.1$ diff postgresql.conf.130723 postgresql.conf 296c296
< log_filename = 'postgresql-%a.log'  # log file name pattern,
---
> log_filename = 'postgresql-%Y%m%d.log'      # log file name
pattern,

-bash-4.1$ diff php.ini php.ini.130717 946c946
< date.timezone = America/Los_Angeles
---
> ;date.timezone =
```

Figure 5: The results of two diff commands one on the postgresql.conf file and one on the php.ini file, showing a few of the changes made so pgfouine could work properly.

## RESULTS

### PGBENCH-TOOLS

We hypothesized that by changing specific postgresql.conf settings database performance for larger databases, measured by TPS as a function of database size, could be altered. Pgbench was first used on the virtual machine with basic postgresql.conf file, as a base test for later comparisons. The results (figure 6) show that the TPS of my virtual machine dropped off at a TPS of ~7,000 and a scale of 35. Shared memory was the first parameter

that was tested; thinking an increase in shared memory would avoid redundant copies of data and allow faster data access between processes.

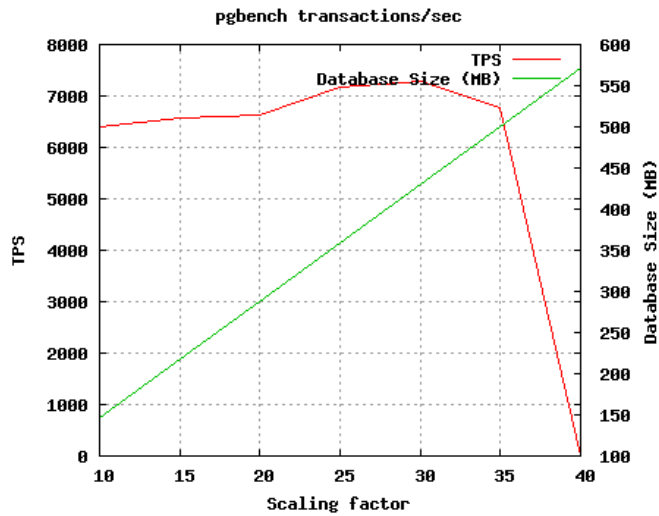


Figure 6: Pgbench-tools graph for virtual machine with base settings. Graph shows TPS vs Scaling factor, red line, and database scale size, green line.

I started by increasing shared memory from 128kb to 500kb, and ran some test to see if this would produce the TPS increase that was expected. However, the change in shared memory had little effect on the TPS. As you can see by figure 7, TPS still dropped off at ~7,000 and a scaling factor of ~35.

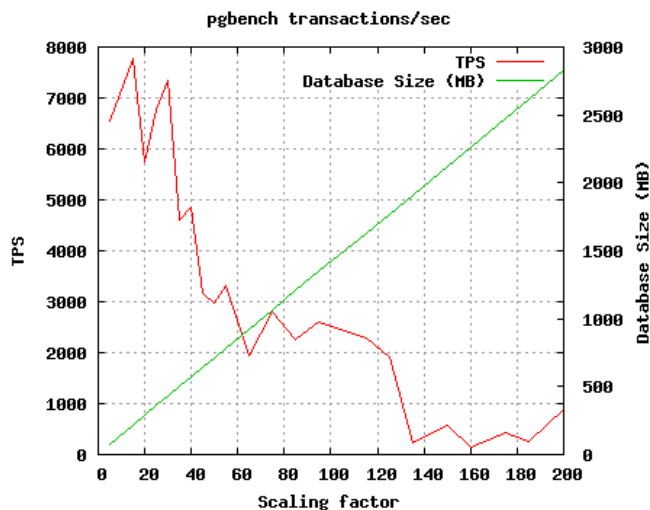


Figure 7: Pgbench-tools graph for virtual machine with shared memory at 500kB TPS vs Scaling factor, red line, and database size, green line.

For the second test with shared memory I changed it from 500kb to 26 MB thinking the first value was just too small of an increase to produce any noticeable affect to TPS and that this larger increase would produce the effect I thought the lower value would. However, this made little difference to the TPS, as you can see by figure 8, TPS still dropped off at ~7,000 and a scaling factor of ~35.

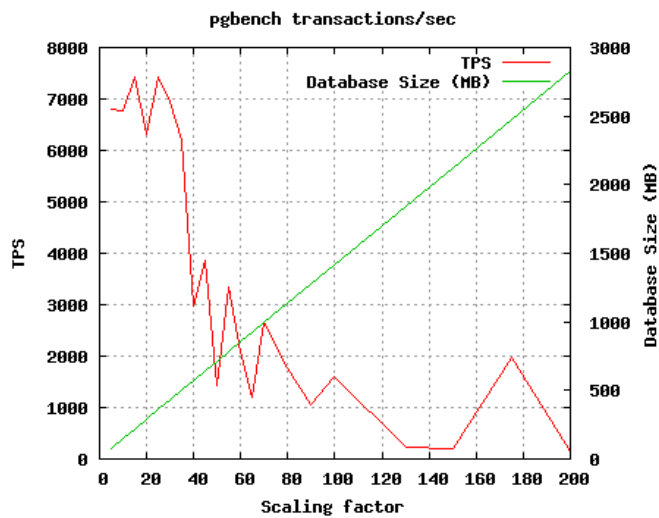


Figure 8: Pgbench-tools graph for virtual machine with shared memory at 26 MB TPS vs Scaling factor, red line, and database size, green line.

The next parameter that I thought would cause a shift in the TPS shoulder was `work_mem`. Thinking this value was too low and it was causing a disk based sort instead of a sort in memory. I started by changing the `work_mem` setting from 1MB to 100MB. As with the previous test I didn't notice any significant shift in the curve. (Figure 9) If anything this had the opposite effect, lowered TPS, of what I thought it would since the scaling factor was  $\sim 30$  compared to  $\sim 35$ . I wasn't sure if this decrease in TPS was actually being caused by the change to `work_mem` or if it was caused by some other factor. To see if I could get a more significant shift and a rise in TPS, I decided to lower the `workmem` for the next test.

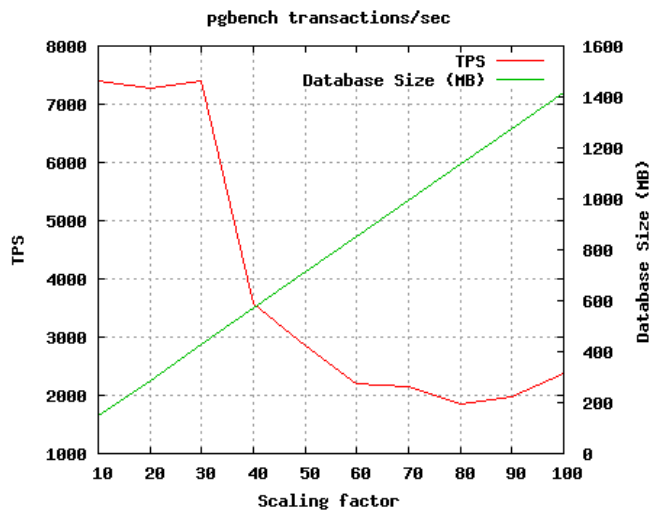


Figure 9: Pgbench-tools graph for virtual machine with `work_mem` at 100MB TPS vs Scaling factor, red line, and database size, green line.

I lowered the `work_mem` setting to its minimum setting of 128kb, thinking this would either confirm that `work_mem` affects TPS or confirm that some other factor was causing the small drop in TPS. The results were the same as before, the scaling factor was still  $\sim 30$  and the TPS was still  $\sim 7,000$  (Figure 10). `Work-mem` was returned to the default 1 MB value and after reading more into pgbench I found it doesn't do any sorts in the test being run.

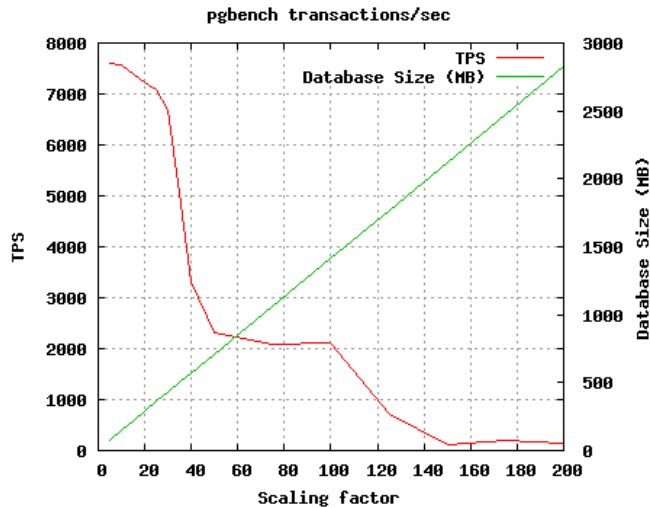


Figure 10: Pgbench-tools graph for virtual machine with work\_mem at the minimum setting TPS vs Scaling factor, red line, and database size, green line.

After not being able to shift the curve with these first test we started to do some more research into what settings would actually produce this effect. We found that the only thing that will significantly shift the curve is the amount of memory a machine has. As found inside PostgreSQL 9.0 on page 198:

“ This curve has a similar shape no matter what hardware you have. The only thing that changes is where the two big break points are at, depending on how much RAM is in the server. You’re limited by the speed of well cached system RAM on the left side, by disk seek rate on the right, and some mix of the two in between.” [1]

Based of this information I changed my virtual machines memory from 640MB to 2,048MB. I expected this to significantly sift the TPS curve to the right. This change worked, (Figure 11) the curve now dropped off at a scaling factor of ~120 compared to ~35. The TPS before the drop was lower however ~5,500 compared to ~7,000, I’m not sure if this was caused by the change in memory or other outside reasons, further investigation would be needed to know for sure.

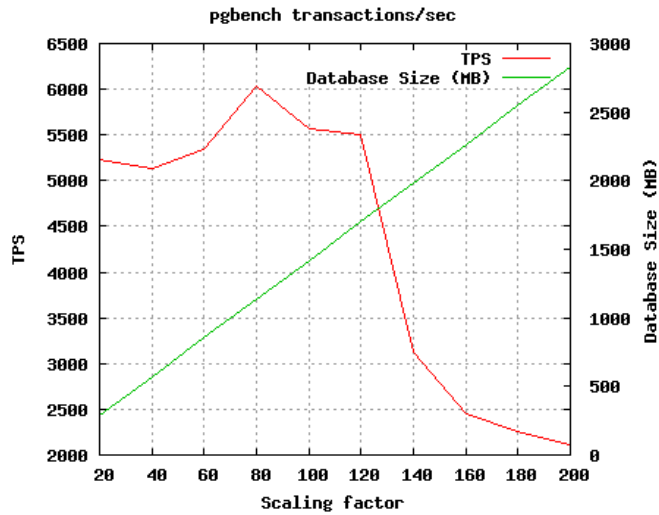


Figure 11: Pgbench-tools graph for virtual machine with system memory at 2048MB TPS vs Scaling factor, red line, and database size, green line.

After seeing the increase caused by increasing memory I wanted to see what kind of increases would happen if I increased the number of cores being used from one to two. However since the computer I was running my virtual machine on only has two cores the results were actually worse then the original results. For 2 cores with 640MB the TPS dropped of at ~6,500 with a scaling factor of ~10 (Figure 12). For 2 cores with 2,048MB the TPS dropped of at ~6,500 with a scaling factor of ~100 (Figure 13). The results are all lower than the results with 1 core and this is due to the virtual machine being forced to share the cores with the main machine, which had similar browser windows and programs open when all the test were run. These tests showed that simply changing values in the postgresql.conf file will normally only have subtle effects on overall performance. However, some changes can have drastic effect in specific areas. For example, increased values of work\_mem will drastically improve queries that involve sorting but will have little affect on non-sorting queries.

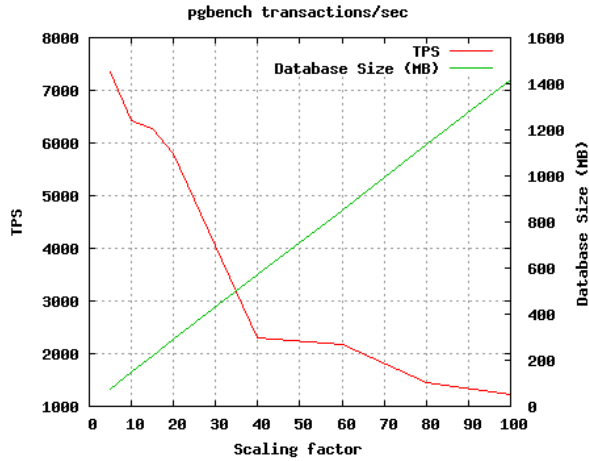


Figure 12: Pgbench-tools graph for virtual machine with system memory at 640MB and using 2 cores, TPS vs Scaling factor, red line, and database size, green line.

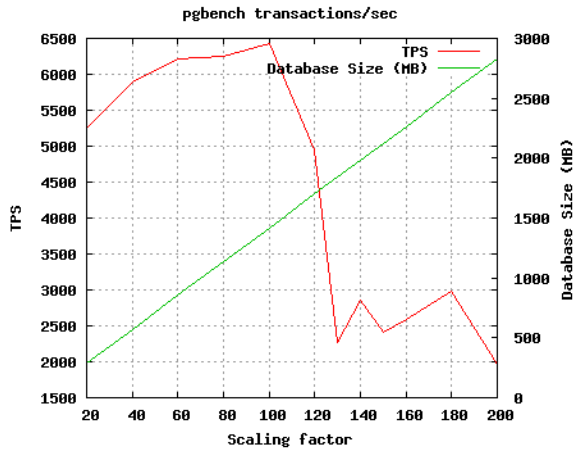


Figure 13: Pgbench-tools graph for virtual machine with system memory at 2048MB TPS vs Scaling factor, red line, and database size, green line.

To confirm that changes to the postgres.conf file don't cause significant TPS improvement pgbench was used on pgttest before and after configuration changes. The changes were to settings like shared memory, work\_mem and others that had been tested on the virtual machine first. As seen from figure 14 and figure 15, the results are almost identical both tests show a drop off at ~8,700 TPS, with a scaling factor of ~150.

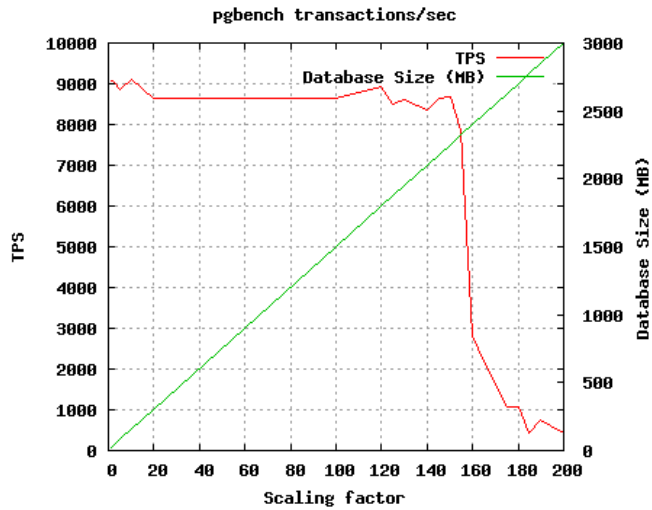


Figure 14: Pgbench-tools graph for pgtest before changing shared memory, work\_mem and other settings, showing TPS vs Scaling factor, red line, and database size, green line.

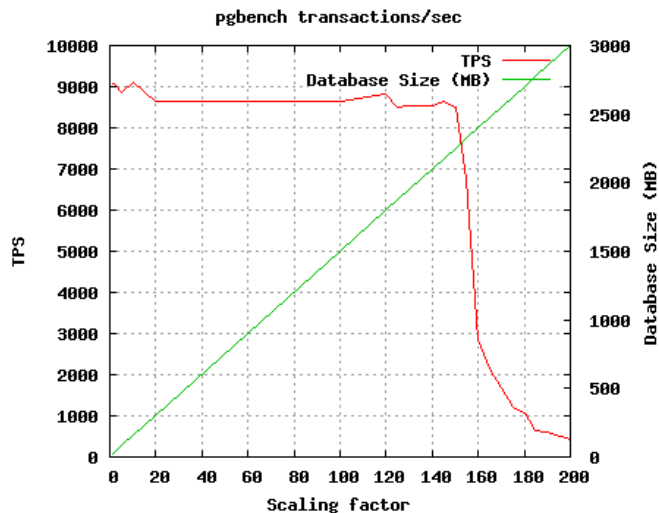


Figure 15: Pgbench-tools graph for pgtest after changing shared memory, work\_mem and other settings, showing TPS vs Scaling factor, red line, and database size, green line.

To further back up both our findings on how configuration changes affect the curve and how memory affects the curve we ran the same tests on Kraken. The results, figure 16 and figure 17, backed both hypotheses up. Kraken has a much higher TPS, scaling factor and database size before it drops off compared to pgtest and my virtual machine. Kraken has ~37,000 TPS and a scaling factor of ~7,000. One interesting thing that did happen

while these tests were executed Kraken ran out of disk space before it ran out of RAM so the results for data base sizes above 105 GB are meaningless. This is the reason you see the database size, green line in graph, not stay in a straight line like it should (Figure 16).

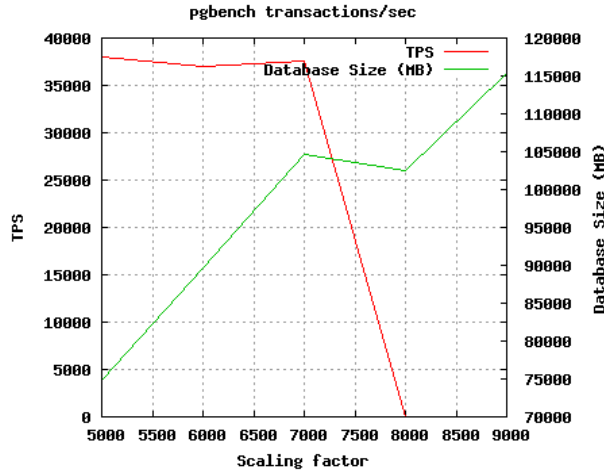


Figure 16: Pgbench-tools graph for Kraken out of space with base settings showing TPS vs Scaling factor, red line, and database size, green line.

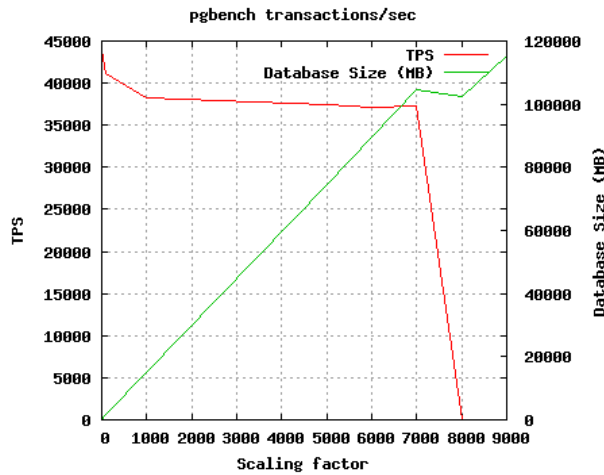


Figure 17: Pgbench-tools graph for Kraken after configuration changes with base settings showing TPS vs Scaling factor, red line, and database size, green line.

## STOQS

After running the tests mentioned in the previous section we installed the STOQS application on Kraken and started loading campaign data. When running loads we

had it set to tell us when it had completed loading 500 data points into the database. The old times for doing this were around 5-7 seconds, where the new times are around 3-5 seconds. We were even able to run multiple loads in parallel without seeing a drop in performance. To see how much of an improvement this was over pgtest, we run the same multiple loads in parallel on pgtest. We found that pgtest started taking ~23 seconds for every 500 data points loaded. Also the average utilization of pgtest was ~20%, where the average utilization of Kraken was ~1%. To get these utilization percents the command “*Sar -b -d*” was used. Since we saw this decrease in load time we thought there would be a similar decrease in the amount of time the STOQS web application would retrieve the data we asked for. One of the values STOQS gives you in the metadata section is how many seconds it took to produce the page, including the graph and map (Figure 18).

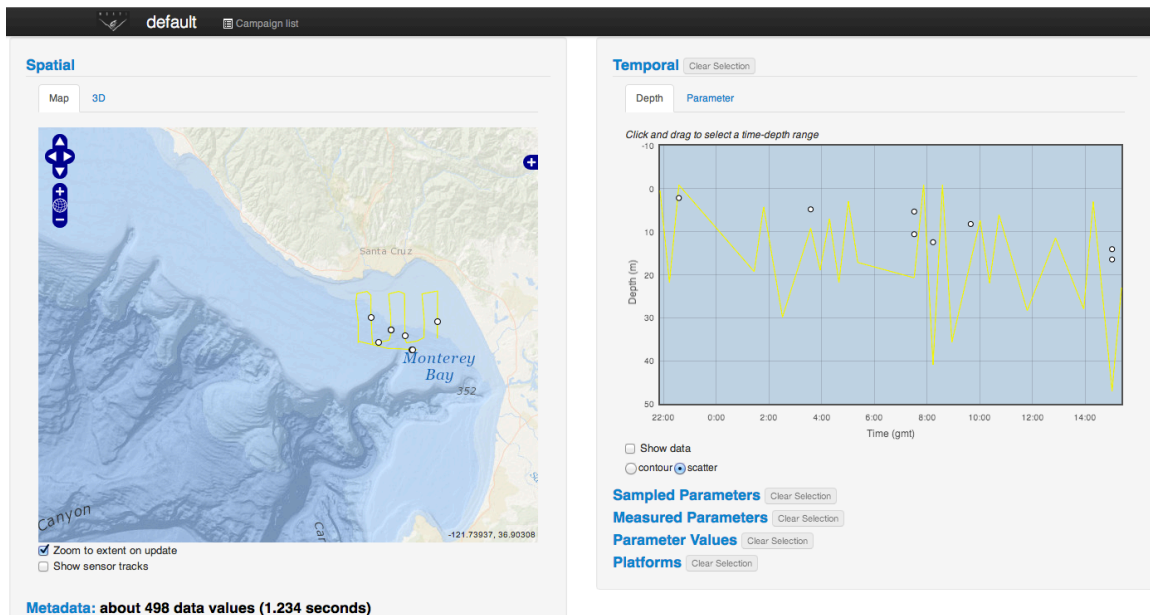


Figure 18: STOQS web application showing the time value, next to metadata on the bottom left, mentioned in the section above.

When the same campaign data was opened on Kraken and Pgtest the response time differed by .1 to .8 seconds, with Kraken being faster. Although this is an improvement it's not the 1 or more second difference we were hoping for. Note these values depend on the campaign being looked at, since some have more data

points than others. Once we had a campaign loaded into Kraken we started making queries using STOQS, so we could populate the logs and run pgfouine.

## PGFOUINE

With the results from pgfouine about what queries are taking the longest and being run the most we can find non-server factors that might be the cause for the long response times. This sample from a pgfouine report shows a query that was executed twenty-eight times for a total of four minutes and thirty-nine seconds and a query that takes an average of forty-six seconds to complete and is run five times (Figure 19). Now that queries similar to these can be found easily, we can analyze them and make changes as outlines in the book: *Indexes, Tricking the query planner*.

Queries that took up the most time (N) ^

Rank	Total duration	Times executed	Av. duration (s)	Query
1	4m39s	28	9.97	<code>SELECT COUNT(*) FROM (SELECT "stoqs_parameter"."name" AS "name", "stoqs_parameter"."standard_name" AS "standard_name", "stoqs_measurement"."depth" AS "depth", "stoqs_measurement"."geom" AS "geom", "stoqs_instantpoint"."timevalue" AS "timevalue", "stoqs_activity"."name" AS "name" AS col5, "stoqs_platform"."name" AS col6, "stoqs_measuredparameter"."datavalue" AS "datavalue", "stoqs_parameter"."units" AS "units" FROM "stoqs_measuredparameter" INNER JOIN "stoqs_parameter" ON ("stoqs_measuredparameter"."parameter_id" = "stoqs_parameter"."id") INNER JOIN "stoqs_measurement" ON ("stoqs_measuredparameter"."measurement_id" = "stoqs_measurement"."id") INNER JOIN "stoqs_instantpoint" ON ("stoqs_measurement"."instantpoint_id" = "stoqs_instantpoint"."id") INNER JOIN "stoqs_activity" ON ("stoqs_instantpoint"."activity_id" = "stoqs_activity"."id") INNER JOIN "stoqs_platform" ON ("stoqs_activity"."platform_id" = "stoqs_platform"."id")) subquery;</code>
2	3m50s	5	46.13	<code>EXPLAIN (ANALYZE, buffers) SELECT COUNT(*) FROM (SELECT "stoqs_parameter"."name" AS "name", "stoqs_parameter"."standard_name" AS "standard_name", "stoqs_measurement"."depth" AS "depth", "stoqs_measurement"."geom" AS "geom", "stoqs_instantpoint"."timevalue" AS "timevalue", "stoqs_activity"."name" AS col5, "stoqs_platform"."name" AS col6, "stoqs_measuredparameter"."datavalue" AS "datavalue", "stoqs_parameter"."units" AS "units" FROM "stoqs_measuredparameter" INNER JOIN "stoqs_parameter" ON ("stoqs_measuredparameter"."parameter_id" = "stoqs_parameter"."id") INNER JOIN "stoqs_measurement" ON ("stoqs_measuredparameter"."measurement_id" = "stoqs_measurement"."id") INNER JOIN "stoqs_instantpoint" ON ("stoqs_measurement"."instantpoint_id" = "stoqs_instantpoint"."id") INNER JOIN "stoqs_activity" ON ("stoqs_instantpoint"."activity_id" = "stoqs_activity"."id") INNER JOIN "stoqs_platform" ON ("stoqs_activity"."platform_id" = "stoqs_platform"."id")) subquery;</code>
3	47.5s	1	47.53	<code>EXPLAIN ANALYZE SELECT COUNT(*) FROM (SELECT "stoqs_parameter"."name" AS "name", "stoqs_parameter"."standard_name" AS "standard_name", "stoqs_measurement"."depth" AS "depth", "stoqs_measurement"."geom" AS "geom", "stoqs_instantpoint"."timevalue" AS "timevalue", "stoqs_activity"."name" AS col5, "stoqs_platform"."name" AS col6, "stoqs_measuredparameter"."datavalue" AS "datavalue", "stoqs_parameter"."units" AS "units" FROM "stoqs_measuredparameter" INNER JOIN "stoqs_parameter" ON ("stoqs_measuredparameter"."parameter_id" = "stoqs_parameter"."id") INNER JOIN "stoqs_measurement" ON ("stoqs_measuredparameter"."measurement_id" = "stoqs_measurement"."id") INNER JOIN "stoqs_instantpoint" ON ("stoqs_measurement"."instantpoint_id" = "stoqs_instantpoint"."id") INNER JOIN "stoqs_activity" ON ("stoqs_instantpoint"."activity_id" = "stoqs_activity"."id") INNER JOIN "stoqs_platform" ON ("stoqs_activity"."platform_id" = "stoqs_platform"."id")) subquery;</code>

Figure 19: Example of the results produced by pgfouine, showing queries that took the longest.

## CONCLUSIONS/RECOMMENDATIONS

The results from pgbench showed us that the only way to have a significant impact on TPS is to increase system memory. Since Kraken does have more and faster system memory than the other servers its TPS is higher, however the overall response time of STOQS has not increased that significantly, only .1 to .8 seconds. This surprised us since we thought the new dedicated powerful server would have a huge impact on the response time and load times. The new server has decreased the load times when multiple data sets are loaded in parallel, which may be more useful then just a huge increase in individual load times.

For the next steps in increasing the performance of the STOQS user interface I believe getting more pgfouine results about what queries are taking the longest and what queries are being run the most will lead to figuring out which other areas of STOQS are impacting its performance. Based on our benchmark results, we're confident that postgresql is well configured and that by analyzing pgfouine reports and improving the way STOQS works we can achieve further performance improvement.

## **ACKNOWLEDGEMENTS**

First and foremost I would like to thank the David and Lucile Packard Foundation and Christopher Scholin for putting the MBARI intern program on. A huge thank you to my mentor Mike McCann for choosing me to work on this project, for helping me throughout my time at MBARI, and for teaching me a lot of things along the way. Thank you to George Matsumoto and Linda Kuhnz for organizing all the amazing intern activities, from BBQs to kayaking. For sponsoring and supporting me during this internship I want to thank the UROC program at CSUMB, and specifically Dr. Bill Head & Gerick Bergsma. For great company and memories I want to thank my fellow interns. Lastly, I would like to thank the entire MBARI staff for being very kind and helpful throughout my time here.

## **References:**

1. Smith, Gregory (2012). PostgreSQL 9.0 High Performance
2. STOQS application (2012) [Software]. Available from <https://code.google.com/p/stoqs>
3. Pgbench-tools (2013). [Software]. Available from <https://github.com/gregs1104/pgbench-tools>
4. Pgfouine (Version 1.2). [Software]. Available from <http://pgfouine.projects.pgfoundry.org>