TidalScale

Scaling your computer to your problem

Scale | Simplify | Optimize | Evolve

TidalScale System Performance Analysis for R

Contents

Chuck Piercey <chuck.piercey@tidalscale.com>

1 Executive Summary

This white paper explores the performance of R analytics with a large data set on a TidalScale system.

The TidalScale HyperKernel provides the ability to scale-up the execution of applications beyond the limits of a single physical server. TidalScale's approach delivers better performance for the application while avoiding the need for re-architecting or re-engineering for scale-out to achieve good performance on a large data set.

R is an analytic and statistical programming language whose use is rapidly spreading as organizations sharpen their ability to understand and learn from data they amass. Many operations in R are memory intensive, and analysts and data scientists often struggle to keep their working data sets within the limitations of a single computer to avoided the dreaded "Memory Cliff".

> **Memory Cliff Defined**: As the RAM usage of an application increases, for any number of reasons (including algorithms that take more space, more data being processed, or memory management bugs), there will tend to be a point at which memory usage exceeds the bounds of the physical RAM that can be allocated to the application. At that point, the Virtual Memory system (usually part of the Operating System) will take over and start moving the application's data between RAM and disk to compensate. Disk access is very slow, so the speed at which the application runs usually drops sharply at that point. This sudden speed drop is called a Memory Cliff, because it looks like a cliff on a plot of memory usage to application speed; performance is high before the cliff and low after it. With Virtual Memory systems that move individual pages, the Memory Cliff manifests itself as a large number of page faults requiring context switches and disk I/O. With Virtual Memory systems that move the whole address space (suspending the application in the meantime), the Memory Cliff manifests itself as a large amount of time spent suspended on disk.
>
> — Source http://c2.com/cgi/wiki?MemoryCliff

Customers report that once R hits the memory cliff it becomes effectively unusable and they are forced either to down-sample their data or completely rewrite their application in Python specifically for a scale out architecture (*if* the problem is able be split across multiple machines).

The TidalScale HyperKernel allows multiple physical computers to run as a single system which allows the unmodified application to use the aggregate RAM, CPUs and I/O of the underlying hardware. The R applications therefore "see" a single computer system while the TidalScale HyperKernels enable growth and performance by scaling-out on commodity hardware. This matters because the cost per bit of commodity hardware is several orders of magnitude less expensive than high end hardware.

In this report, we measure the performance and capacity of R applications with very large datasets, running on "bare metal" servers versus a server instance on TidalScale. The performance results described below illustrate important aspects of TidalScale performance as of version 1.0. We demonstrate scalable performance of seven different R operations which, more generally, are representative of the kinds of analytic workloads that benefit from running on large, coherent systems.

We will demonstrate that the TidalScale HyperKernel conquers the memory cliff that applications typically encounter when they exceed the size of reasonably priced currently available systems. By removing the memory barrier and constraints of a single piece of hardware, R applications running on TidalScale can continue to scale with good performance to very large workloads.

2 Test Hardware

All of the R tests were run on standard server systems built with the following components:

- **Motherboard**: Colfax CX1260i-X6
- **Processor**: Intel® Xeon® CPU E5-2660 v2 @ 2.20GHz
- **Chipset**: Intel C612
- **Memory**: 8 slots filled (out of 24 slots) with 16GB DDR4 1600MHz DIMMs for a total of 128GB RAM

- **Storage**: 2 Samsung 840EVO 512GB SSDs and 2 Western Digital 2TB hard drives (CMS data files were read from a single SSD)

The tests are run across two system configurations:

- **Linux directly on Hardware**: Our base of comparison is a single 128GB Colfax CX1260i as described above.
- **Linux on a Multi-node TidalScale "TidalPod"**: We use multiple standard 128GB servers in a 5 node TidalScale "TidalPod" configured with 8 vCPUS and 640GB of RAM for the guest OS [1]. The TidalPod uses a 10G network as its "system resource bus" that is invisible to the guest CentOS 6.6 OS running on the TidalPod.

3 Test Software

The testing environment included the following software:

- CentOS 6.6
- Revolution R Enterprise 3.1
  - R modules installed: base, dplyr, stats, nlme, mgcv, randomForest, Matrix

The complete software environment for this test is available for download at the github site noted below.

4 R Test Data

To performance test R with a large data set we use the Centers for Medicare and Medicaid Service's (CMS) 2008-2010 Data Entrepreneurs' Synthetic Public Use Data Set (DE-SynPUF). Through Medicare, Medicaid, the Children's Health Insurance Program, and the Health Insurance Marketplace, CMS provides medical coverage to 100 million people and has a plethora of data to offer for analysis.

The **CMS DE-SynPUF** data set was created with the goal of providing a realistic set of claims data in the public domain while providing the very highest degree of protection to the Medicare beneficiaries' protected health information. The purposes of the DE-SynPUF are to:

1. Allow data entrepreneurs to develop and create software and applications that may eventually be applied to actual CMS claims data;
2. Train researchers on the use and complexity of conducting analyses with CMS claims data prior to initiating the process to obtain access to actual CMS data; and,
3. Support safe data mining innovations that may reveal unanticipated knowledge gains while preserving beneficiary privacy.

The DE-SynPUF contains five types of data for 2008, 2009 and 2010 – Beneficiary Summary, Inpatient Claims, Outpatient Claims, Carrier Claims, and Prescription Drug Events.

Table 1: CMS DE-SynPUF Data Set Overview

DE-SynPUF	Unit of record	Number of Records 2008	Number of Records 2009	Number of Records 2010
Beneficiary Summary	Beneficiary	2,326,856	2,291,320	2,255,098
Inpatient Claims	Claim	547,800	504,941	280,081
Outpatient Claims	Claim	5,673,808	6,519,340	3,633,839
Carrier Claims	Claim	34,276,324	37,304,993	23,282,135
Prescription Drug Events (PDE)	Event	39,927,827	43,379,293	27,778,849

[1] … do not count the memory footprint of the TidalScale hypervisor itself, which is about 5% overhead, or the TidalScale …d by the user).

Although the CMS DE-SynPUF has very limited inferential research value to draw conclusions about Medicare beneficiaries due to the synthetic processes used to create the data, it provides a realistic insurance claims data set for **R** testing purposes. For more details about the data structures in this data set and coding guidelines see the SynPUFs web site.

5 R Benchmark Program

This benchmark records the execution time of seven R operations run sequentially:

1. Read .csv files of a specified sample range and load the matching data structures in an R data frame,
2. Add some required metadata columns for the regressions and join the data structures by patient ID,
3. Perform a **Generalized Additive Model** linear regression,
4. Perform a **Generalized Linear Model** linear regression,
5. Perform a **Decision Tree** analysis,
6. Perform a **Random Forest** analysis, and
7. Perform a **K Nearest Neighbor** analysis.

For details on the code in the test see the appendix below. For more details on GLM and GAM see this CMU Lecture.

The testing methodology involves running these operations at various in-memory sizes as shown in the table below:

Table 2: Test Data Points

Number of CMS Samples	In-Memory Size (CentOS guest top)
1	34GB
3	102GB
5	170GB
8	272GB
10	340GB
13	442GB
17	578GB
20	680GB

As shown in this table, the workload in-memory size is driven by the number of samples loaded. The test time results are scatter-plot charted against In-Memory size to visualize system scalability across the range of workloads.

The complete test program and the dataset used are available on git for external use at http://www.github.com/tidalscale/R_benchmark_test. The complete system boot image for CentOS & Revolution R can be downloaded and assembled from the following sources:

- CentOS 6.6
  - https://www.centos.org/download/
- Revolution R Open - a free, enhanced open source R distribution from Revolution Analytics. Note that these tests were run with Revolution R Enterprise, a version including additional extensions and support from Revolution Analytics
  - http://www.revolutionanalytics.com/get=revolution-r

The identical system image was used to boot and execute the R tests on both the bare metal and TidalScale systems.

6 Bare Metal Performance Results

R will take advantage of as much system memory as it needs to perform its work. During an R operation such as GLM or GAM, R may demand memory several multiples of the original data set size to complete its work. If system memory is insufficient to complete the operation, R will begin paging from storage (i.e. swap from disk), thus falling over the memory cliff.

We can see the memory cliff by running the R benchmark up to a workload size that is bigger than a single 128GB server with SWAP ON.

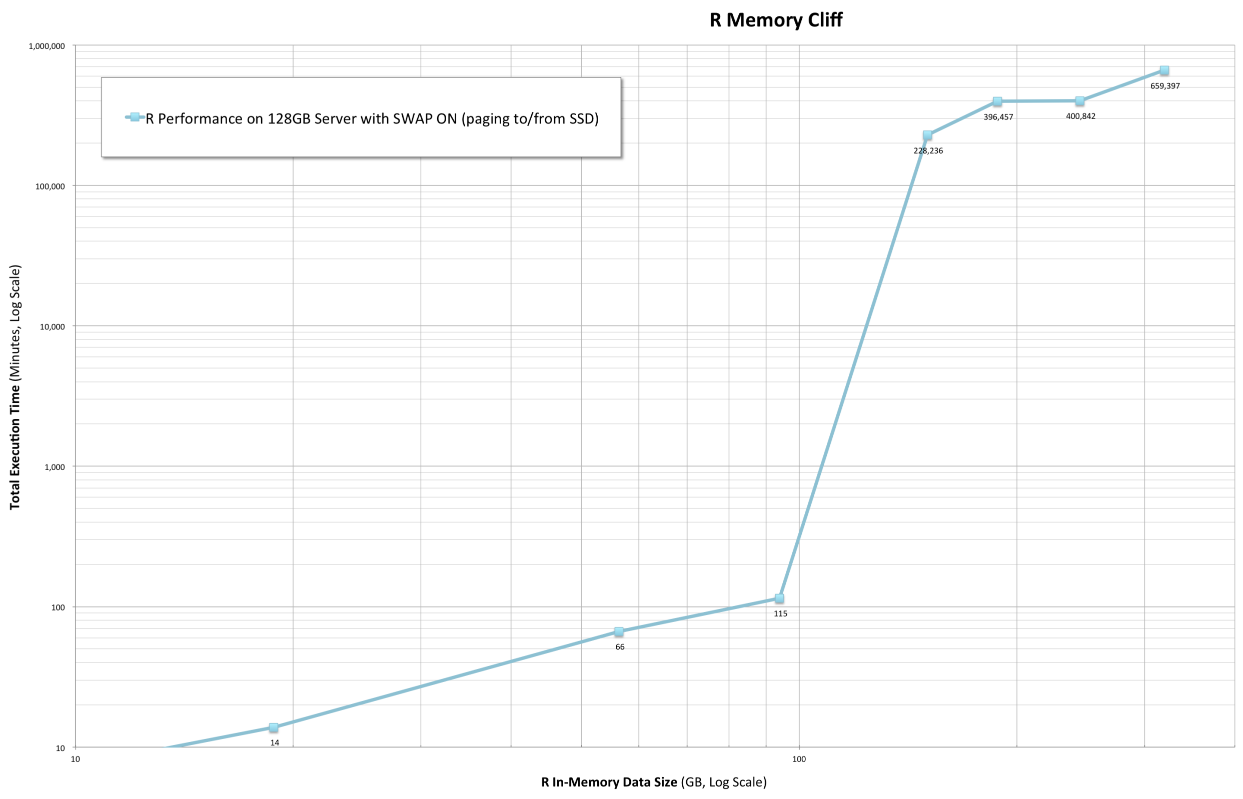

Figure 1: The Memory Cliff - R Execution Job Times by In-Memory Job Size on a Single Server with memory SWAP ON

In other words, as long as the workload data fits within RAM the performance is excellent. As soon as the memory requirement exceeds physical RAM, the OS starts paging to/from SSD storage and the workload executes several orders of magnitude more slowly.

To focus this test on in-memory performance, in this benchmark we eliminate the effects of paging to storage by configuring the OS with SWAP OFF. The chart below shows what happens in the SWAP OFF configuration: on 128GB bare metal nodes our test R workload functions up to size 3 (92GB) of available memory after which point the subsequent jobs fail.

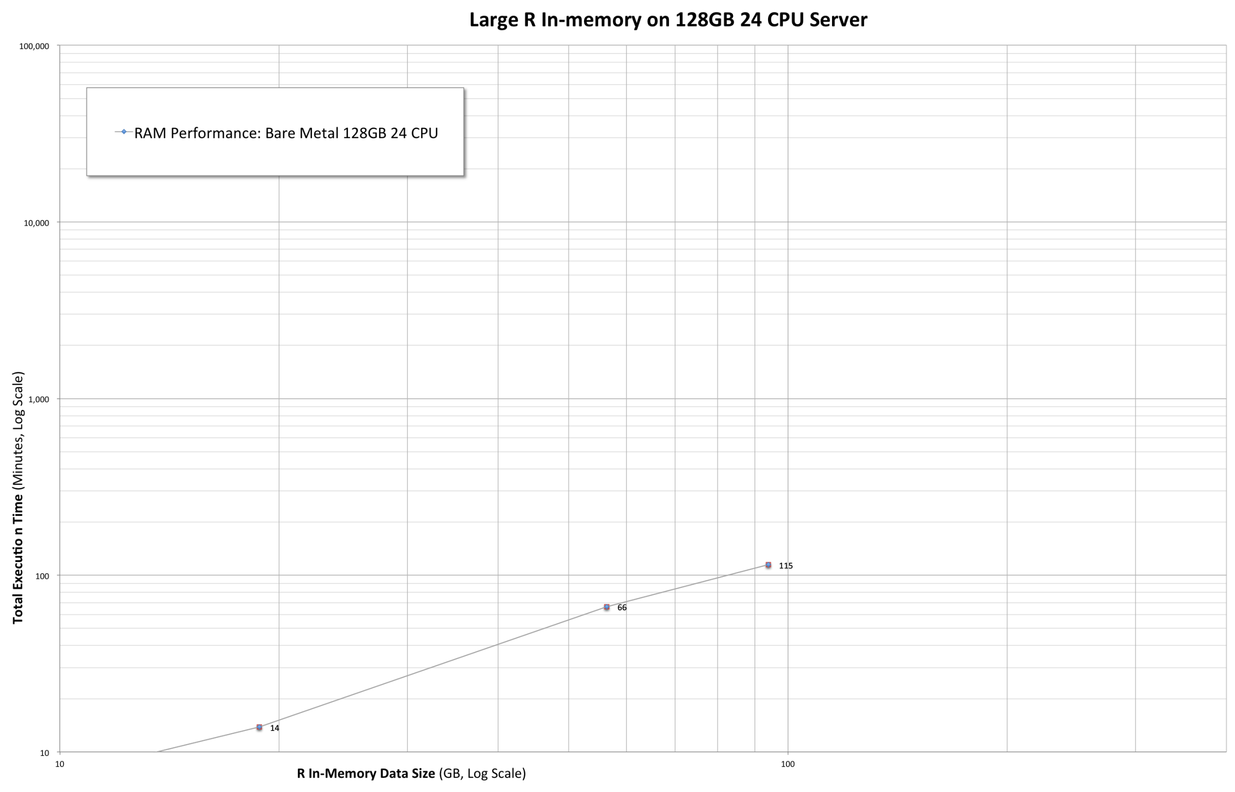

Figure 2: Bare Metal - R Execution Job Times by In-Memory Job Size

With R and SWAP OFF, one of two failures occur when memory is exhausted:

1. R fails to obtain a block of memory, or
2. CentOS kills the R process with a `kill -9` error (this means that the process is using too much memory for the system).

Beyond the size 3 test job, using 128GB bare metal systems would require rewriting the R application and incorporating new libraries to divide the work across multiple 128GB servers. This introduces risk of result variations and inconsistencies in the analytic results.

7 TidalScale Performance Results

We booted a TidalScale system with 640GB of RAM (a TidalPod comprised of five 128GB worker nodes) and an unmodified CentOS 6.5 guest OS and ran the same test sequence as in the bare metal case. TidalScale allows the guest OS and applications to continue running in-memory beyond the limits of a single physical system. We found that R is able to run and perform well when it has access to very large amounts of RAM. The results show that TidalScale enables R and CentOS to handle growing data analysis requirements seamlessly by binding multiple smaller hardware systems into a single, coherent virtual system.

TidalScale's architecture virtualizes and mobilizes all system resources. The guest operating system's view of what hardware it manages and how it is connected does not correspond to the real physical system's structure. Instead, the guest system runs on a virtual machine. The TidalScale HyperKernel presents the GuestOS with virtual processors, virtual physical memory blocks and virtualized I/O devices. TidalScale assigns virtual processors, memory, and I/O to the physical processors, memory blocks and I/O devices so that the guest system will execute as efficiently as possible. The TidalScale HyperKernel reacts to Guest system activity, learns and reconfigures physical resources so that processors that access the same memory blocks will be on the same node and share the same caches as much as possible for as long as they continue to interact. The physical location of memory, processors, and I/O devices is rearranged on the fly based on the observed working set of the current workload.

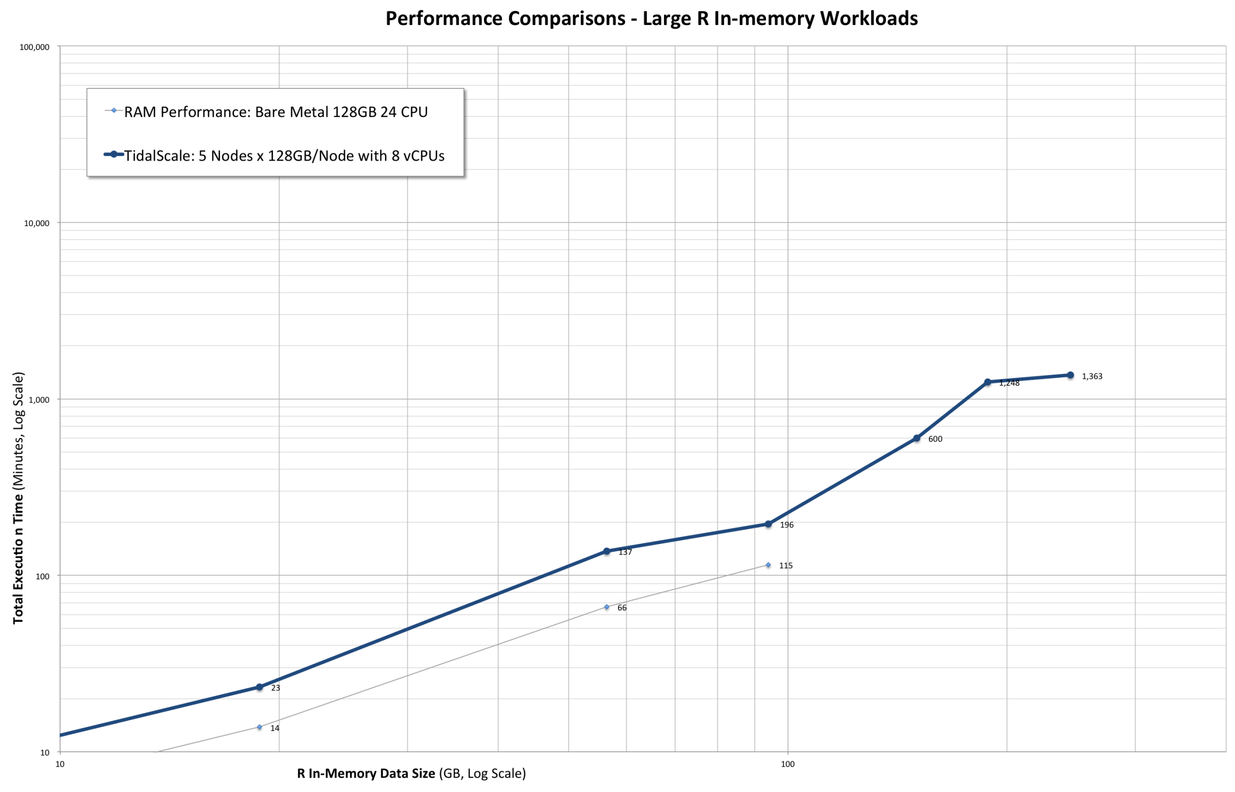

Figure 3: TidalScale (5x128GB nodes) - R Execution Job Times by In-Memory Job Size

This result shows that TidalScale is effectively performing with in-memory R data loads that are greater than any single 128GB node in the 5 node TidalPod.

This result compares favorably to paging from either SSD or HD. For example, for the size 8 R workload (272GB), TidalScale was able to complete the workload in 10 hours versus paging from SSD where the job would have taken 158 *days* to complete. At large scale in-memory workloads, R running on the TidalPod is 380 times faster than R paging from SSD and 380,000 times faster than R paging from hard drive. In other words, the TidalScale system performance is orders of magnitude faster than performance on a system that needs to swap memory to disk or SSD, even a very fast SSD. Where the swapping solution is so slow as to be unusable, TidalScale makes it possible to perform large data analysis that was previously out of reach of standard server hardware.

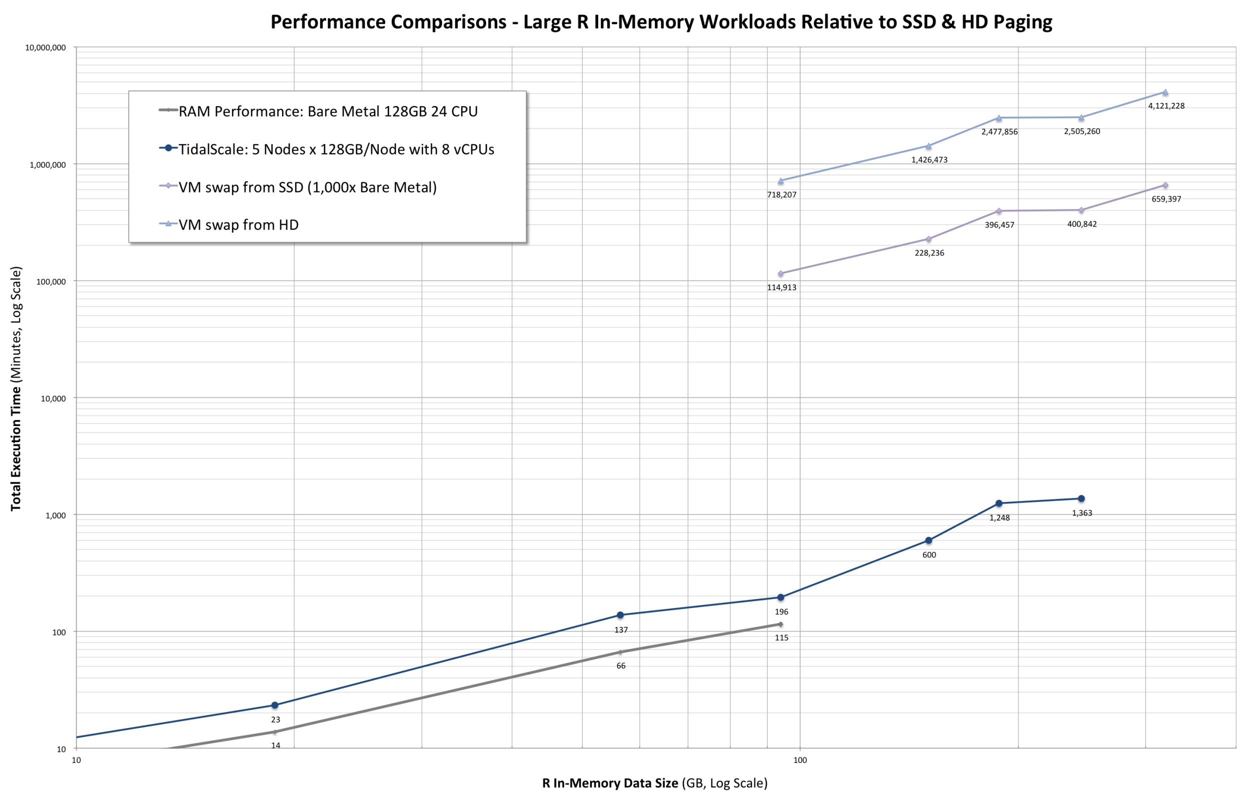

Figure 4: TidalScale Performance versus Bare Metal

In short, the TidalScale HyperKernel's software defined scalability effectively removes the Memory Cliff and delivers a scalable and high performance platform for running R analytic loads against datasets that exceed the capacity of traditional single servers upon which the TidalScale HyperKernel is running.

Note
This test configuration happened to use 5 nodes but the same results can be expected as you scale up or down by adding or subtracting nodes to or from the TidalScale system.

8 Conclusion

The results of these performance tests demonstrate that TidalScale conquers the memory cliff that applications typically encounter when they exceed the size of reasonably priced hardware systems. These tests show that as R scales in its memory requirements, TidalScale can provide a larger platform to run the application, without modification, at larger in-memory workload sizes. As shown in the chart below, this breaks through the dreaded memory cliff.

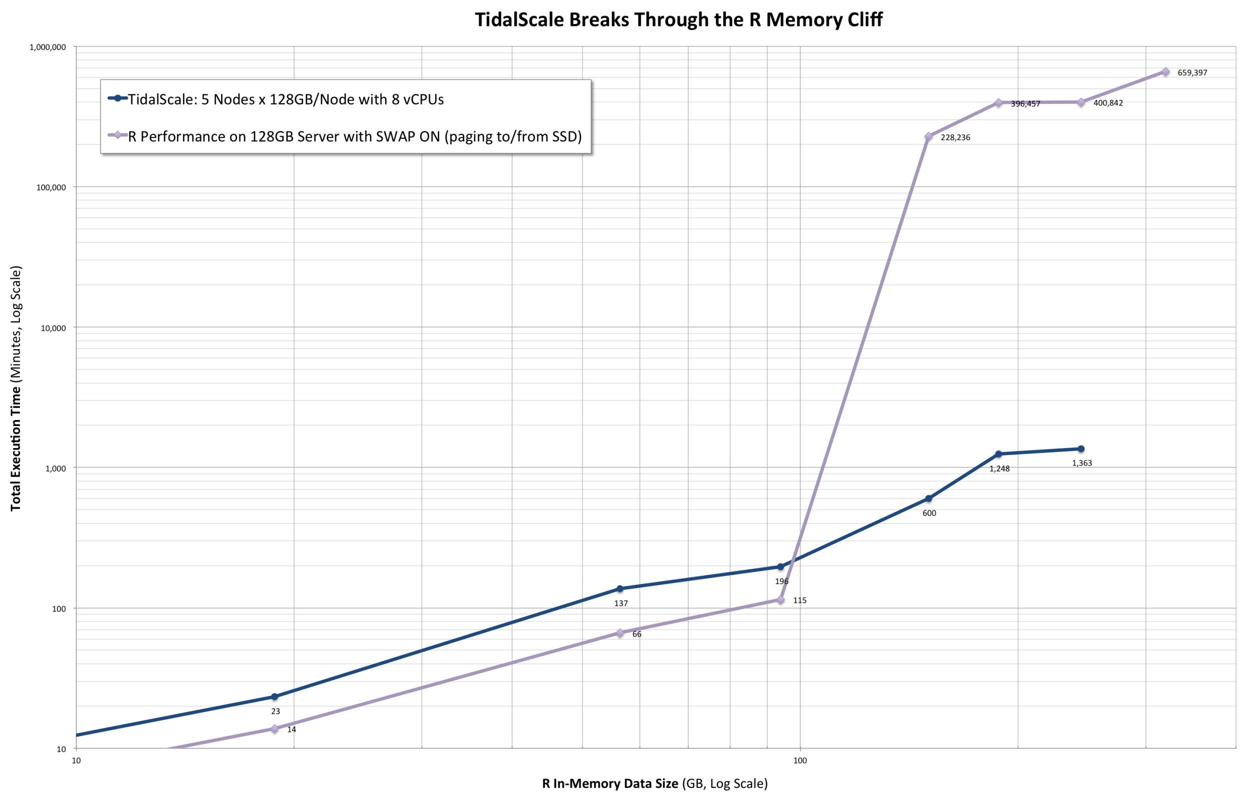

Figure 5: TidalScale Performance versus Bare Metal

Finally, because the HyperKernel software runs on industry standard hardware, TidalScale systems offer an economical alternative to data scientists hitting the limits of the memory cliff.

9 Appendix 1: Revolution R Test Environment

R.version

```
> R.version
platform       x86_64-unknown-linux-gnu
arch           x86_64
os             linux-gnu
system         x86_64, linux-gnu
status
major          3
minor          1.3
year           2015
month          03
day            09
svn rev        67962
language       R
version.string R version 3.1.3 (2015-03-09)
nickname       Smooth Sidewalk
```

10 Appendix 2: Data Size Details

The CMS data set is chunked into 20 "samples." Each sample consists of 8 raw .csv files:

1. DE1_0_2008_Beneficiary_Summary_File_Sample_1.csv"
2. DE1_0_2009_Beneficiary_Summary_File_Sample_1.csv"
3. DE1_0_2010_Beneficiary_Summary_File_Sample_1.csv"
4. DE1_0_2008_to_2010_Inpatient_Claims_Sample_1.csv"
5. DE1_0_2008_to_2010_Outpatient_Claims_Sample_1.csv"
6. DE1_0_2008_to_2010_Carrier_Claims_Sample_1A.csv"
7. DE1_0_2008_to_2010_Carrier_Claims_Sample_1B.csv"
8. DE1_0_2008_to_2010_Prescription_Drug_Events_Sample_1.csv"

Loaded as separate data structures in R, the 8 files of a single sample occupy 6.0GB on disk. Loading all 20 samples from all 160 csv files (8 files/sample * 20 samples) occupies ~120GB on disk.

Each test operation step requires additional memory. For instance, for a single sample the sequence of memory footprint progresses as follows:

Table 3: Physical Memory Used after each step of the Sample Size 1 Test

R Processing Step for 1 Sample	**RAM Used by R (according to mem_used())**
Load	6.1 GB
Join	15.6 GB
GAM	15.8 GB
GLM	18.2 GB
Decision Tree	18.2 GB
Random Forest	18.2 GB
K Nearest Neighbors	18.3 GB

Extended across all 20 samples, these four operations require 376GB of application memory in total (according to R mem_used()).

11 Appendix 3: R Code

The key lines of code for each R operation in the test are detailed below:

R Operation 1: Load Data

```
# Get patient data, also known as beneficiary summary
load_time <- load_time + system.time(patient <- get_data_read_csv_patients(patient_file_name,  ↩
    sample_id, years))
# Get Inpatient claims data
load_time <- load_time + system.time(inpatient <- get_data_read_csv(inpatient_file_name,  ↩
    sample_id, smoke_test))
# Get out patient claims data
load_time <- load_time + system.time(outpatient <- get_data_read_csv(outpatient_file_name,  ↩
    sample_id, smoke_test))
```

```
# Get carrier claims data
load_time <- load_time + system.time(carrier_claims <- get_data_read_csv(claims_file_name, ↩
    claims_sample, smoke_test))
# Get patient drugs data
load_time <- load_time + system.time(drug <- get_data_read_csv(drug_file_name, sample_id, ↩
    smoke_test))
```

R Operation 2: Join Data

```
# Create a new column for year, using existing date column
join_time <- join_time + system.time(cms_inpatient <- left_join(patient, inpatient, by = c(" ↩
    desynpuf_id", "year")))
# Create a new column for year, using existing date column
join_time <- join_time + system.time(cms_outpatient <- left_join(patient, outpatient, by = c(" ↩
    desynpuf_id", "year")))
join_time <- join_time + system.time(cms_claims_patient <- left_join(patient, carrier_claims, ↩
    by = c("desynpuf_id", "year")))
join_time <- join_time + system.time(cms_drug_patient <- left_join(patient, drug, by = c(" ↩
    desynpuf_id", "year")))
join_time <- join_time + system.time(tree_inpatient_drug <- get_inpatient_drugs_data( ↩
    cms_inpatient, drug))
```

R Operation 3: Generalized Additive Model 2 Linear Regression

```
# Predict total inpatient expenses using Generalized Additive Model 2.
# Includes patient demographic, chronic condition, and prescription drug covariates.
# Patient demographics varibales includes age, gender, and ethnicity/race.
formula_gam1 <- total_inpatient_expense ~ s(age) + bene_sex_ident_cd + sp_alzhdmta +
                sp_chf + sp_chrnkidn + sp_cncr + sp_copd + sp_depressn + sp_diabetes + ↩
                    sp_ischmcht +
                sp_osteoprs + sp_ra_oa + sp_strketia + s(qty_dspnsd_num) + s(days_suply_num) +
                s(ptnt_pay_amt) + s(tot_rx_cst_amt)
```

R Operation 4: Generalized Linear Model Linear Regression

```
# Predict total inpatient expenses using Generalized Linear Model.
# Includes patient demographic, chronic condition, and prescription drug covariates.
# Patient demographics varibales includes age, gender, and ethnicity/race.
formula_glm1 <- total_inpatient_expense ~ age + bene_sex_ident_cd +
                sp_alzhdmta + sp_chf + sp_chrnkidn + sp_cncr + sp_copd + sp_depressn + ↩
                    sp_diabetes + sp_ischmcht +
                sp_osteoprs + sp_ra_oa + sp_strketia + qty_dspnsd_num + days_suply_num +
                ptnt_pay_amt + tot_rx_cst_amt
```

R Operation 5: Decision Tree

```
# Predict response variable total inpatient expenses using beneficiary demographic, chronic ↩
    condition,
# and prescription drug as predictors.
formula_tree <- total_inpatient_expense ~ age + bene_sex_ident_cd + sp_alzhdmta +
                sp_chf + sp_chrnkidn + sp_cncr + sp_copd + sp_depressn + sp_diabetes + ↩
                    sp_ischmcht +
                sp_osteoprs + sp_ra_oa + sp_strketia + tot_quantity + tot_days_supply +
                tot_patient_pay_amount + tot_drug_cost
```

R Operation 6: Random Forest

```
# Predict response variable total inpatient expenses using beneficiary demographic, chronic ↩
    condition,
# and prescription drug as predictors by using an ensemble of decision tress.
```

```
formula_rf <- total_inpatient_expense ~ age + bene_sex_ident_cd + sp_alzhdmta +
              sp_chf + sp_chrnkidn + sp_cncr + sp_copd + sp_depressn + sp_diabetes +  ↩
                 sp_ischmcht +
              sp_osteoprs + sp_ra_oa + sp_strketia + tot_quantity + tot_days_supply +
              tot_patient_pay_amount + tot_drug_cost
```

R Operation 7: K - Nearest neighbours

```
# Predict response variable total inpatient expenses using beneficiary demographic, chronic  ↩
   condition,
# and prescription drug as predictors.
knn_cols <- c("total_inpatient_expense", "age", "bene_sex_ident_cd", "sp_alzhdmta",
              "sp_chf", "sp_chrnkidn", "sp_cncr", "sp_copd", "sp_depressn", "sp_diabetes", " ↩
                 sp_ischmcht",
              "sp_osteoprs", "sp_ra_oa", "sp_strketia", "tot_quantity", "tot_days_supply",
              "tot_patient_pay_amount", "tot_drug_cost")
```