Sizing Fedora on Open Archive Architectures

A Sun ISV Engineering Test Report

Eric R. Reid, ISV Engineering
Sun Microsystems, Inc.
9 June 2009

Introduction

Fedora Commons' Fedora open source repository software has been identified as the central software component of Sun's new Open Archive architectures. Intended to provide open, enterprise-ready data preservation and archival systems, this combined hardware/software stack promises stellar performance and reliability for long-term data archival.

N.B.: *The open source archival software from Fedora Commons is not to be confused with Fedora the Linux variant.*

This document details initial sizing studies done to date on configurations based on the following components:

- Fedora 3.x atop the included Tomcat 5.5
- MySQL 5.x
- OpenSolaris 2008.11, including ZFS
- Sun x64-based servers
- Sun Storage JBOD disk arrays

The stated goals of this project:

1. Characterize Fedora data ingest performance across configurations
2. Characterize Fedora data access performance access configurations
3. Determine performance differences introduced by different technologies and topologies
4. Determine areas for future performance work

The author recognizes the significant contributions of Dan Davis of Fedora Commons, without whom this work could not have been possible.

Testing Approach

As this is the first round of testing on a brand-new archival architecture, we started simply, with a few basic ground rules:

- Start with a small number of configurations, per Sun Storage Marketing
- Test with configurations 'out of the box', i.e. little or no tuning, to establish baselines
- Publish results by end of May, 2009, even if issues or follow-up testing remain

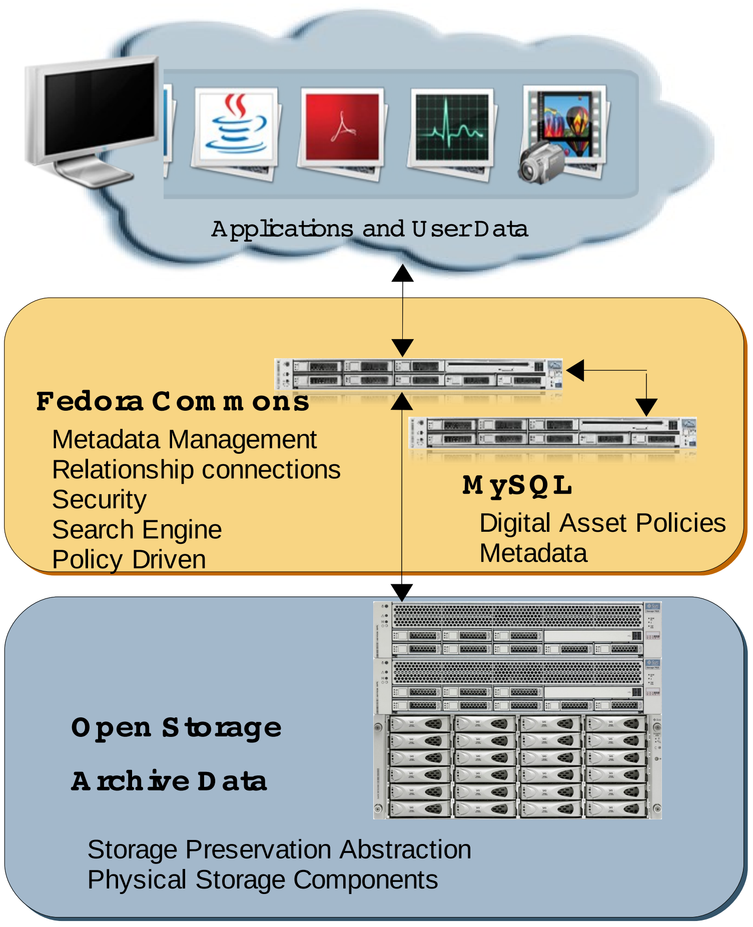
Fedora Commons' choice of test harness was The Grinder (http://grinder.sourceforge.net), an open source, Java-based load generation platform. Running from a separate load server, the Grinder allowed for customized test scripts to run against the configuration in a variety of ways.

Three areas were to be tested for this effort:

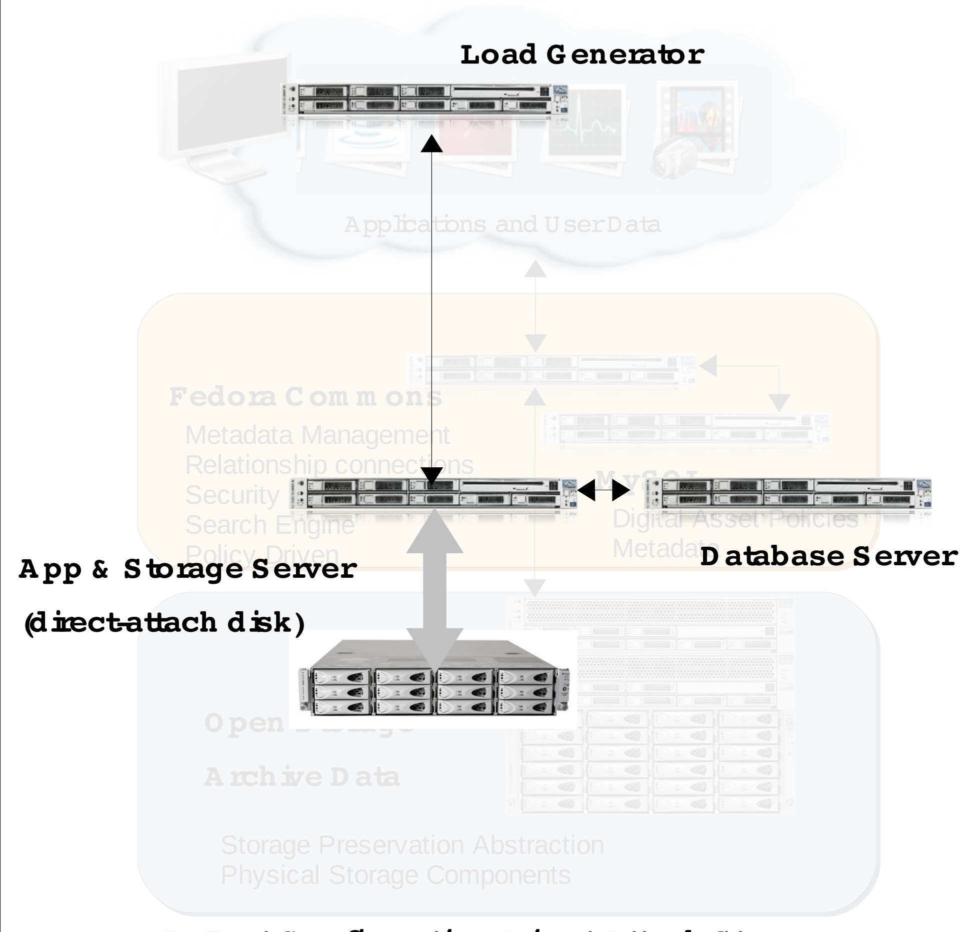
1. Fedora data and object ingest performance
2. Fedora data access performance
3. Fedora object-only burn-in testing (“how many objects can Fedora ingest at one time?”)

Test Architectures

Two configurations (variations of the generic architecture) were tested.

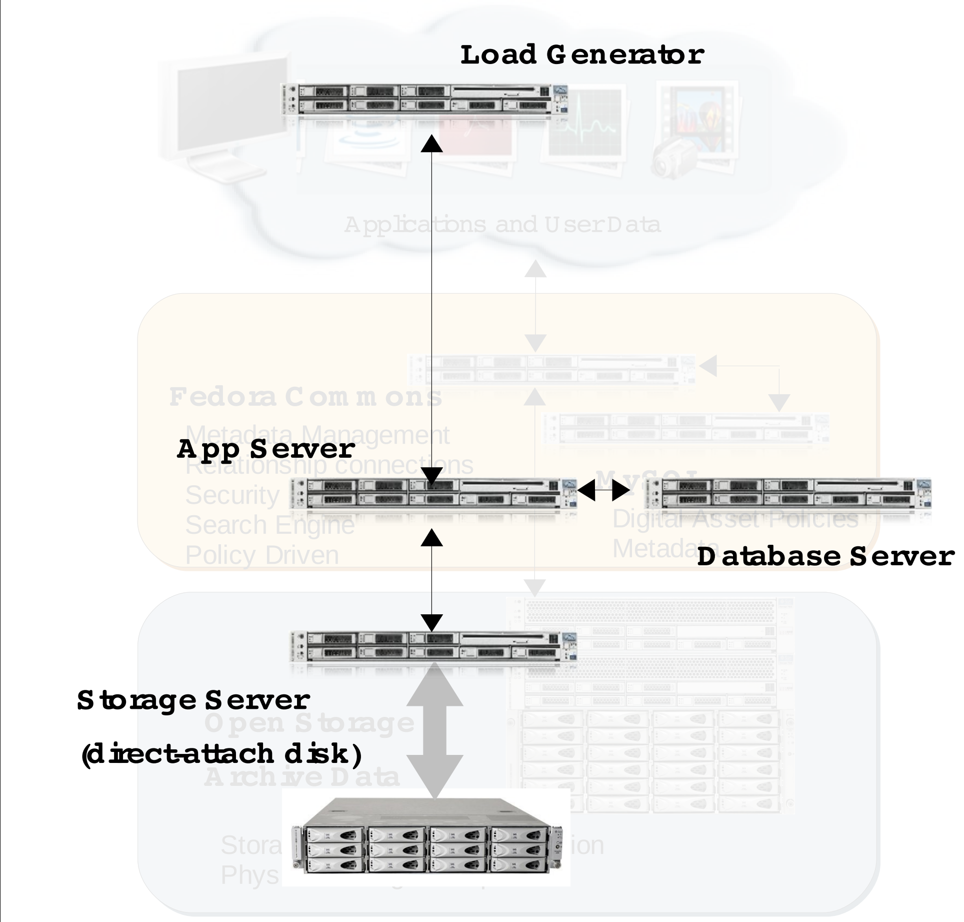


A. Generic Open Archive Architecture



B. Test Configuration 'Direct-Attach Storage'

	Hardware	Software
Load Generator	Sun Fire x4450 4 x 2.93GHz dual-core Xeon 16GB Memory	The Grinder Solaris 10 update 5
App/Storage Server	Sun Fire x4150 2 x 3.16GHz quad-core Xeon 32GB Memory 2 x 32GB SSD devices	Fedora 3.1 Tomcat 5.5 app server JDK/JRE 1.6 OpenSolaris 2008.11
Storage	Sun Storage J4200 JBOD Array 12 x 250GB 7200RPM SATA II drives SAS interface	1 striped ZFS pool over 12 disks
Database Server	Sun Fire x4150 2 x 3.16GHz quad-core Xeon 32GB Memory	MySQL 5.0 Opensolaris 2008.11
Networking	Trunked 3 x Gigabit Ethernet	



C. Test Configuration 'Network Server'

	Hardware	**Software**
Load Generator	Sun Fire x4450 4 x 2.93GHz dual-core Xeon 16GB Memory	The Grinder Solaris 10 u5
App Server	Sun Fire x4150 2 x 3.16GHz quad-core Xeon 32GB Memory	Fedora 3.1 Tomcat 5.5 app server JDK/JRE 1.6 OpenSolaris 2008.11
Storage Server	Sun Fire x4150 2 x 2.66GHz quad-core Xeon 32GB Memory 2 x 32GB SSD devices	OpenSolaris 2008.11
Storage	Sun Storage J4200 JBOD Array 12 x 250GB 7200RPM SATA II drives SAS interface	1 striped ZFS pool over 12 disks NFS v4 mounts of ZFS pools
Database Server	Sun Fire x4150 2 x 2.66GHz quad-core Xeon 32GB Memory	MySQL 5.0 Opensolaris 2008.11
Networking	Trunked 3 x Gigabit Ethernet	

The Fedora Workloads

While Fedora Commons' offering provides myriad functions, the two most important operations (in the eyes of archivists) are Ingest (creation of archived objects from external sources) and Access (retrieval of archived objects in read-only mode). Ingest performance is important for both initial and ongoing data object ingest into an archive; in addition, most archives are not just write-only, and the ability to access archived data is quite important.

This testing effort employed ingest and access tests against the two configurations detailed above:

- Ingest objects (API-A)
- Access objects via SOAP (API-A)
- Access objects via HTTP (API-M-LITE)

In addition, several 'standard' object sizes were used for each test:

- 'Object-only': Zero-sized or 'placeholder' objects
- 'Small': 20KB objects (exemplifying a PDF file)
- 'Medium': 15MB object (exemplifying a YouTube video)
- 'Large': 700MB object (exemplifying a CD image)

Test Runs

A single test run consists of (for now) 8 Grinder processes with 2 threads each on the Load Generator, making Ingest or Access requests against the App Server. The runs are at least 300 seconds each. CPU, memory and disk usage was logged on each server for each run.

In addition, a separate, one-time Ingest test against the 'Local' configuration was made with Object-only, and allowed to run until the system failed. We denote this as the 'burn-in' test.

It should be noted that the numbers obtained via this and future Open Archive testing represent the first testing of its kind against Fedora, and as such cannot really be compared to anything competitive at present (anecdotally, we're told by Fedora Commons that the Ingest rates obtained far exceed anything they've seen to date).

Test 1: Object-only Ingest	**Average TPS**	**Average response time (ms)**	**Max App Server CPU Usage**	**Max I/O Burst (MB/s)**	**Average Data Throughput (MB/s)**
Local Storage Config	117	137	18%	3.5	N/A
Storage Server Config	54	295	12%	1	N/A

Test 1:

- Object-only means no data, therefore no data throughput
- 2:1 Ratio between untuned direct-attach storage and network storage (expected)

Test 2: Small Ingest	**Average TPS**	**Average response time (ms)**	**Max App Server CPU Usage**	**Max I/O Burst (MB/s)**	**Average Data Throughput (MB/s)**
Local Storage Config	117	137	19%	11	2.4
Storage Server Config	44	363	10%	4.5	0.9

Test 3: Medium Ingest	**Average TPS**	**Average response time (ms)**	**Max App Server CPU Usage**	**Max I/O Burst (MB/s)**	**Average Data Throughput (MB/s)**
Local Storage Config	6.71	2380	19%	145	100.7
Storage Server Config	5.75	2770	30%	264	86.3

Tests 2-3:

- Data throughput for small objects indicates significant overhead (see data throughput)
- 3:1 ratio between direct-attach storage and network storage for small objects was expected
- Better ratio between direct-attach storage and network storage for medium objects
- A single GigE interface was saturated using network storage for medium objects

Test 4: Large Ingest	**Average TPS**	**Average response time (ms)**	**Max App Server CPU Usage**	**Max I/O Burst (MB/s)**	**Average Data Throughput (MB/s)**
Local Storage Config	0.16	97300	19%	149	109.9
Storage Server Config	0.14	105000	29%	291	100.8

Test 4:

- Close to parity in terms of direct-attach vs network attach storage performance
- Storage server case uses more App Server CPUs to support NFS client
- A single GigE interface was saturated using network storage for large objects

Test 5: Small API-M-LITE Access	**Average TPS**	**Average response time (ms)**	**Max App Server CPU Usage**	**Max I/O Burst (MB/s)**	**Average Data Throughput (MB/s)**
Local Storage Config	406	7	12%	Negligible	8.3
Storage Server Config	400	8	13%	Negligible	8.2

Test 6: Small API-A Access	**Average TPS**	**Average response time (ms)**	**Max App Server CPU Usage**	**Max I/O Burst (MB/s)**	**Average Data Throughput (MB/s)**
Local Storage Config	370	34	16%	Negligible	7.6
Storage Server Config	370	34	17%	Negligible	7.6

Test 7: Medium API-M-LITE Access	**Average TPS**	**Average response time (ms)**	**Max App Server CPU Usage**	**Max I/O Burst (MB/s)**	**Average Data Throughput (MB/s)**
Local Storage Config	5.4	2930	10%	27	81
Storage Server Config	5.5	2880	9%	7	82.5

Test 8: Medium API-A Access	**Average TPS**	**Average response time (ms)**	**Max App Server CPU Usage**	**Max I/O Burst (MB/s)**	**Average Data Throughput (MB/s)**
Local Storage Config	0.51	25100	18%	8	7.7
Storage Server Config	0.53	24700	7%	8	8

Tests 5-8:

- Very close to parity between performance for direct storage and network storage
- ZFS and NFS/ZFS seem to cache reads very well, disks are barely accessed for Small cases

Test 9: Large API-M-LITE Access	**Average TPS**	**Average response time (ms)**	**Max App Server CPU Usage**	**Max I/O Burst (MB/s)**	**Average Data Throughput (MB/s)**
Local Storage Config	0.11	134000	10%	70	77
Storage Server Config	0.11	132000	20%	54	77

Test 9:

- Direct-attach performance identical to network attach
- Storage server case uses more App Server CPUs to support NFS client

Test 10: Object-only ingest burn-in	**Average TPS**	**Average response time (ms)**	**Consecutive Transactions w/o Errors**	**Elapsed Wall Time**
Local Storage Config	41	24	25M	150h 32m

Test 10:

- Something in the Fedora and/or Tomcat servers 'falls over'
- Further testing and tuning of load will definitely improve this number

Observations

This first round of Fedora/Open Archive testing yielded some important untuned baseline numbers, and an indication of potential areas for improvement.

Firstly, the DB Server never had its resources significantly taxed. As such, an 'all-in-one' solution (perhaps based on the Sun Fire x4540 storage server) should be considered in the future.

At no time were any of the servers' CPU or I/O resources strained. A single GigE link would become the bottleneck for certain of these tests, but we have shown that the inherent trunking capabilities within OpenSolaris can tie together unused ports (the x4100 series has 4 standard) to avoid this; alternatively, there exist 10GigE options for these servers.

The Fedora/App Server did see some significant memory usage in even this 32GB configuration. The nature of this J2EE-based application needs to be observed in future tests, with an eye towards garbage collection strategies optimal to Fedora and Tomcat.

ZFS proved to be performant, although when combined with NFS showed room for tuning and improvement.

Lastly, Fedora itself needs to be examined for the way in which it accesses files/filesystems from Java. When the results obtained were compared with simplistic Java test programs which wrote 16 files of various sizes simultaneously (on the 'Local' configuration), significant performance discrepancies were uncovered:

Comparision: Fedora Ingest vs Java test program	**Small Objects (files/sec)**	**Medium Objects (files/sec)**	**Large Objects (files/sec)**
Fedora	117	6.71	0.16
Java	11264	27.6	0.8
Ratio	***9627.0%***	***411.3%***	***500.0%***

Whither Hybrid Storage Pools?

This initial round of testing was conducted with storage configured as single RAID 0 ZFS Pools, without ZILs or Caches. Any other RAID levels used will likely impact the performance numbers.

The astute reader will note that Solid-State Storage Devices (SSDs) were listed in the hardware manifest, and indeed some testing using SSDs in ZFS Hybrid Storage Pools was conducted. In theory, Hybrid Storage Pools can use SSDs as very fast cache devices to potentially improve I/O performance.

In the case of these tests (see Appendix B for details), however, we did not see performance improvements, and in fact Hybrid Storage Pools over NFS showed a significant performance degradation. With the self-imposed timeframe of this testing, it was decided to get these results out, and HSPs will be more fully explored and tuned-for in the subsequent rounds of Fedora sizing testing to follow this effort.

Open Issues and Next Steps

- ***TODO: Test on single Sun Fire x45xx storage server***
- ***TODO: Create Fedora 'update' tests***
- ***TODO: Test 'Extra Large' (4.7GB, DVD size) objects***
- ***TODO: Debug and re-run 'Burn-in' test***
- ***TODO: Test with Sun 7000-series Unified Storage Server***
- ***TODO: Upgrade configuration to OpenSolaris 2009.06 and Fedora 3.2***
- ***TODO: Collapse MySQL DB server onto the App Server, and rerun the tests***
- ***TODO: Test a SAM filesystem including copies to tape. Modify Fedora to take into account access of data from 'disk' when it is on tape and needs to be staged to disk***
- ***ISSUE: Understand why ZFS Hybrid Storage Pools do not improve (or, in the case of NFS, degrade) overall performance***
- ***ISSUE: Review how Fedora is writing data objects to filesystem, understand discrepancy with 'Simple Java Test Program'***

Appendix A – The 'Simple Java File Writing Program'

a) collect.sh

```
#! /bin/bash

trap cleanup 1 2 15

cleanup()
{
  echo  ***CLEANUP****
  for i in $PIDS; do kill -9 $i; done

  endtime=$(date +'%s')
  numfiles=`ls $dir/* | wc -l`
  numbytes=`wc -c $dir/* | awk '{tot=tot+$1} END {print tot}'`
  elapsedtime=`expr $endtime - $starttime`
  fps=`echo "scale=2; $numfiles / $elapsedtime" | bc`
  bps=`expr $numbytes \/ $elapsedtime`

  echo ">>>>>"$numfiles files written in $elapsedtime seconds
  echo ">>>>>""("$fps" files/sec; "$bps" bytes/sec)"
  echo

  rm -rf $dir
}

runit ()
{
#  echo  ***RUNIT****
  runit_dir=$1
  runit_thisrun=$2
  runit_threads=$3
  runit_filesize=$4
  runit_runfor=$5

  mkdir $runit_thisrun  > /dev/null 2>&1

  starttime=$(date +'%s')

  java exercise $runit_dir $runit_threads $runit_filesize 1  &
  PIDS=$!

  for command in "zpool iostat 10" "iostat -xnc 10" "prstat 10"; do
      shortname=`echo $command|awk '{print $1}'`
      eval "$command" > $runit_thisrun/$shortname 2>&1 & PIDS="$PIDS
$!"
  done

  sleep $runit_runfor
}
```

```
threads=16

for storage in ./teststorage; do
  dir=$storage/javatestfiles
  mkdir $dir
  for size in `expr 20 \* 1024` `expr 15 \* 1024 \* 1024` `expr 700 \*
1024 \* 1 024`; do
    name=$threads"_"$size"_"`echo $storage | sed -e "s*./**g"`
    echo ">>>>"$name
    runit $dir $name $threads $size 10
    cleanup
  done
done

pkill -9 prstat
pkill -9 zpool
pkill -9 iostat
```

b) exercise.java

```
import java.io.*;

public class exercise {

    //Display a message, preceded by the name of the current thread
    static void threadMessage(String message) {
        String threadName = Thread.currentThread().getName();
        System.out.format("%s: %s%n", threadName, message);
    }

    static int fileSize = 15 * 1024;
    static int patience = 1000 * 60 * 60;
    static int numThreads = 16;
    static Thread[] threads = new Thread[65536];
    static String dir;
    static byte buffer [];

    private static StringBuffer randomFilename(int n) {
      StringBuffer sb = new StringBuffer(  );
         int c = 'A';
         int r1 = 0;

         for (int i = 0; i < n; i++) {
         r1 = (int) (Math.random() * 3);
         switch (r1) {
           case 0:
             c = '0' + (int) (Math.random() * 10);
             break;
           case 1:
             c = 'a' + (int) (Math.random() * 26);
             break;
           case 2:
             c = 'A' + (int) (Math.random() * 26);
           break;
         }
         sb.append( (char) c );
      }
      return sb;
    }

    private static class FileBlast implements Runnable {
        public void run() {
            String filename = "";
            try {
                for (;;) {
                    filename = dir + "/" + "javatest" +
randomFilename(15);
                    FileOutputStream stream = new
FileOutputStream(filename);

                    threadMessage("Open File " + filename);
                    stream.write(buffer);
                    stream.flush();
                    stream.close();
                    Thread.sleep(patience);
                }
            } catch (FileNotFoundException e) {
                threadMessage("File Not Found! " + filename);
            } catch (IOException e) {
                threadMessage("IO Error!");
            } catch (InterruptedException e) {
                threadMessage("I wasn't done!");
            }
        }
    }

    public static void main(String args[]) throws InterruptedException
{

        if (args.length > 0) {
            dir = args[0];

            try {
                numThreads = Integer.parseInt(args[1]);
            } catch (NumberFormatException e) {
                System.err.println("Argument must be an integer.");
                System.exit(1);
            }

            try {
                fileSize = Integer.parseInt(args[2]);
                buffer = new byte[fileSize];
            } catch (NumberFormatException e) {
                System.err.println("Argument must be an integer.");
                System.exit(1);
            }

            try {
                patience = Integer.parseInt(args[3]) * 1000;
            } catch (NumberFormatException e) {
                System.err.println("Argument must be an integer.");
                System.exit(1);
            }
        }
        else {
                System.err.println("Arguments: <dir> <# threads>
<filesize> <wait>");
                System.exit(1);
        }

        for (int i = 0; i < numThreads; i++) {
           long startTime = System.currentTimeMillis();
           threads[i] = new Thread(new FileBlast());
           threads[i].start();
        }

        threadMessage("Sleeping");
    }
}
```

Appendix B – ZFS Pool Setup

a) Simple Striped ZPool

```
# zpool create storagepool c6t64d0 c6t65d0 c6t66d0 c6t67d0 c6t68d0
c6t69d0 c6t70d0 c6t71d0 c6t72d0 c6t73d0 c6t74d0 c6t75d0
```

b) Hybrid Storage Pool

```
# zpool create hybridpool c7t59d0 c7t60d0 c7t61d0 c7t62d0 c7t63d0
c7t64d0 c7t65d0 c7t66d0 c7t67d0 c7t68d0 c7t69d0 c7t70d0 logs c3t4d0
c3t5d0
```