

InfoScale Storage & Media Server Workloads

VERITAS™

Maximise Performance when Storing and Retrieving Large Amounts of Unstructured Data

Carlos Carrero

Colin Eldridge

Shrinivas Chandukar

Table of Contents

- 01** Introduction
- 02** Hardware Setup
- 03** Maximize Volume Throughput
- 04** File System Throughput

Introduction

+ *Document purpose*

+ *Internet of Things*

01

DOCUMENT PURPOSE

This document uses a given hardware configuration as a base framework and uses it as a guide to find what is the best possible software configuration.

Several tests will be performed in order to understand how the hardware responds to different I/O demands so bottlenecks can be identified at different parts of the architecture.

As a typical I/O pattern for media solutions, sequential I/O read will be used. There are some software tunables like the number of bytes that are pre-fetched using read ahead whose values can affect overall performance. This guide will evaluate how to make the best choices.

While this is a high level overview, a very detailed guide with more information and examples can be found in this [technical article](#).

Ways to exercise I/O

Measure performance

Achieve balanced I/O

Identify bottlenecks

File System tuning

Find a perfect balanced configuration that will get the most out of your hardware

INTERNET OF THINGS

Data is at the core of the business and it is a key asset for the Internet of Things Market where some analyst predict 30% CAGR. Even without having to read any analyst paper, it is quite clear that the amount of data in any form of video, images, social information that we are storing is dramatically being increased year after year.

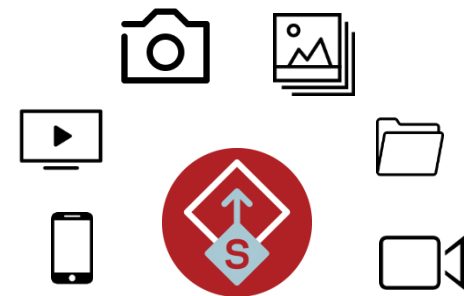
Companies are looking for ways to take advantages of that huge amount of information and workloads such as Rich Media Analytics are expected to grow by 3 times in the coming year.

The platform needed to store and analyse all that information needs to be continuously available and has to be prepared to satisfy all the I/O requirements for both stream analytics for immediate results or post analysis to find common patterns.

InfoScale Storage is a perfect Software Defined Solution to create a storage backend that can adapt to changing I/O demands.

Because the flexibility of using a Software Defined Solution any hardware can be used to create a commodity storage backend able to store and manage the data needed for Internet of Things solutions.

This document highlights a step by step procedure to understand the hardware capabilities used under InfoScale Storage so you can make sure that maximum throughput and capacity can be obtained from your current and future hardware investments.



InfoScale Storage

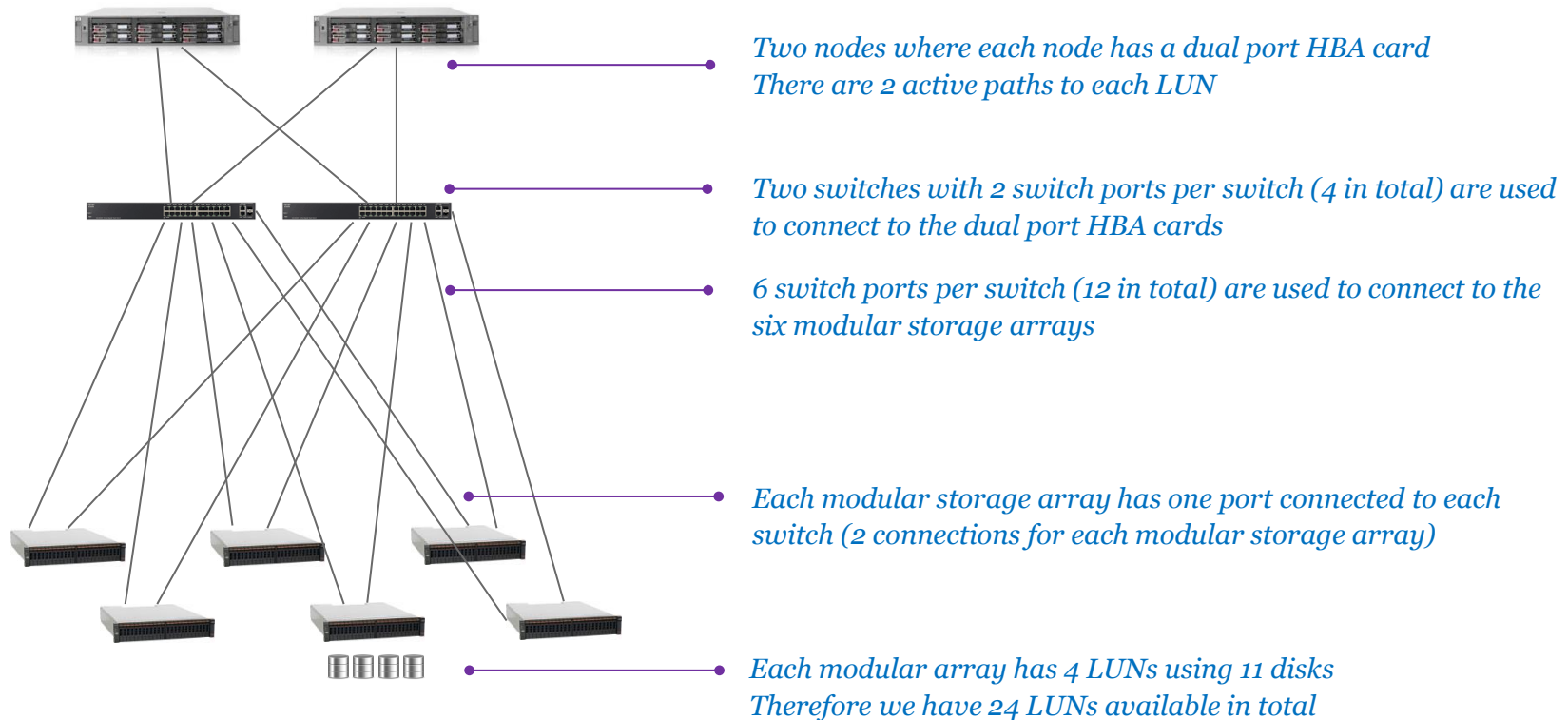
Hardware Setup

- + *Hardware configuration*
- + *Host side*
- + *Switch side*
- + *Modular array side*
- + *LUN side*

02

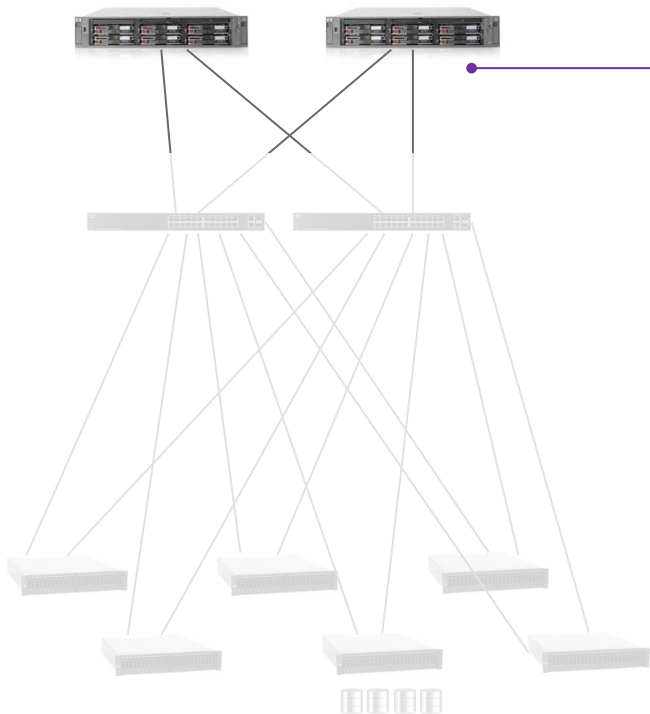
HARDWARE CONFIGURATION

Understanding your hardware capabilities is the first step towards a tuning exercise. The hardware we used during our testing is represented below.



HOST SIDE

Understand what are the theoretical limits at the host side



*Two nodes where each node has a dual port HBA card
There are 2 active paths to each LUN*

*Max theoretical throughput
per port is **8Gbits/sec***

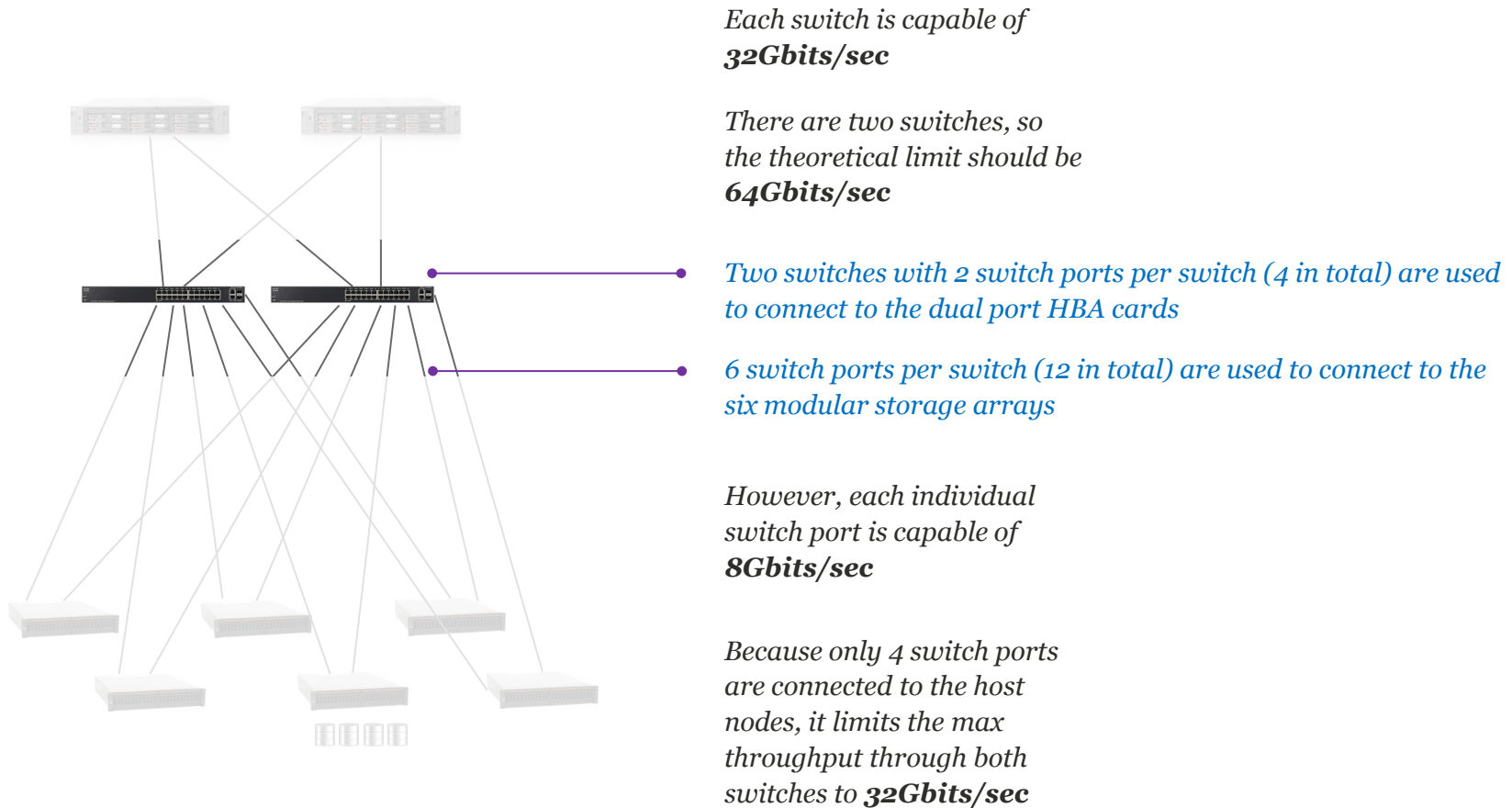
*Because two paths are used,
max theoretical throughput
per node would be
16Gbits/sec*

*The max theoretical
throughput for two nodes
should be **32Gbits/sec***

*In our One-node testing, the dual port HBA card
bottlenecked at approximately **12Gbits/s***

AT THE SWITCH

Understand the theoretical limits at the switch



MODULAR STORAGE ARRAY

Understand the theoretical limits of the storage modular arrays

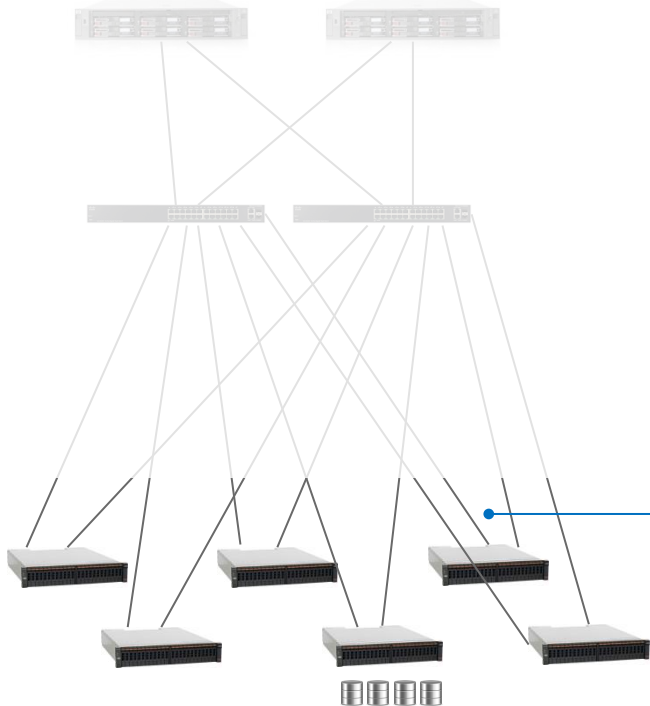
There are 6 modular storage arrays, each has 2 ports

*Each port has a theoretical max throughput of
4Gbits/sec*

There are a total of 12 storage array connections to the two FC switches

*The total theoretical max throughput is therefore
48Gbits/sec for all six modular storage arrays*

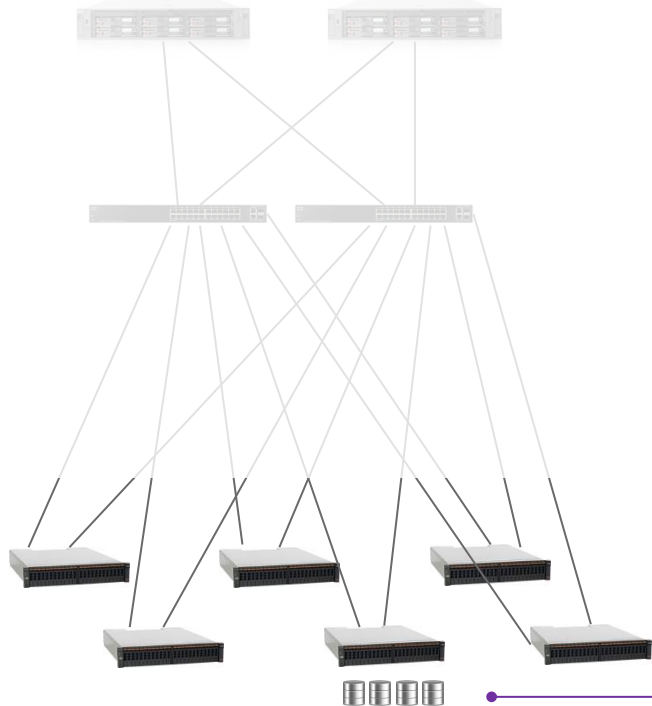
Each modular storage array has one port connected to each switch (2 connections for each modular storage array)



*In our Two-node testing, the combination of 6 storage arrays bottlenecked at **20Gbits/sec***

LUN CONFIGURATION

Understand the LUN configuration, ensure all the LUNs are configured equally



Each LUN is composed of 11 disks in a RAID-0 configuration

There are 4 LUNS available per modular array

There are a total of 24 LUNs available in the environment

Each LUN is 3TB approx

Because there are 2 paths per LUN, there is a total of 48 active paths on each node

*Each modular array has 4 LUNs using 11 disks
Therefore we have 24 LUNs available in total*

LUN THROUGHPUT

Details on the LUN throughput



Reading from a single LUN
we achieve **170MBytes/sec**



Reading from two LUNs we
achieve **219MBytes/sec**
because the contention at the
modular array level

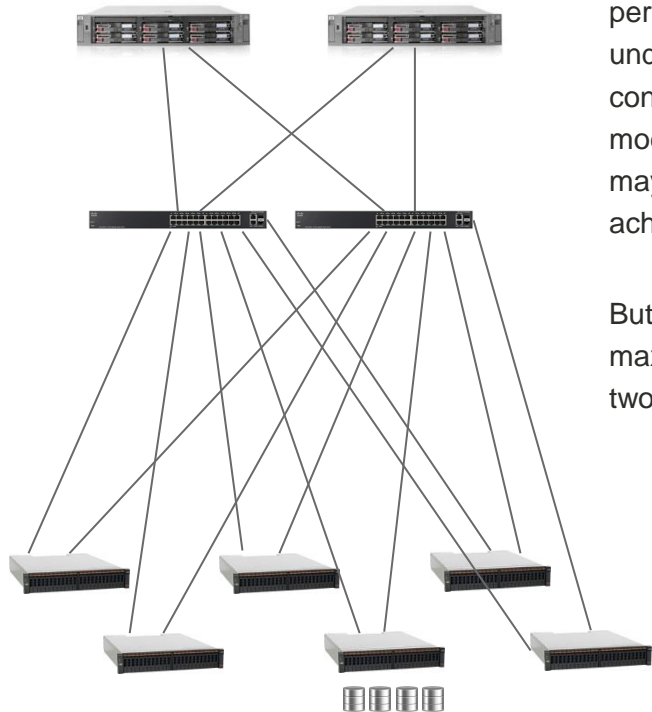


Reading from two LUNs,
where each LUN is located in
a different modular storage
array, the performance is
342MBytes/sec

*Each modular array controller cache is shared when I/O is
generated to multiple LUNs within the same array*

HARDWARE CONFIGURATION

It is important to understand the difference between the theoretical performance of each of the components and the real performance exhibited in your particular configuration



Understanding all the different performance we may get will help us to understand the limits we have in our configuration. If we only take a look at the modular storage array performance, we may have the perception of being able to achieve **48Gbits/s**.

But as we have seen, we have a maximum theoretical throughput for the two nodes of **32Gbits/s**.

In fact, and as we will see later, the maximum performance achieved is **20Gbits/sec** when reading from both nodes and **12Gbits/sec** when reading from one node.

The following sections will describe the volume and file system configuration and the different tests performed to achieve a balanced IO configuration across all six storage modular arrays.

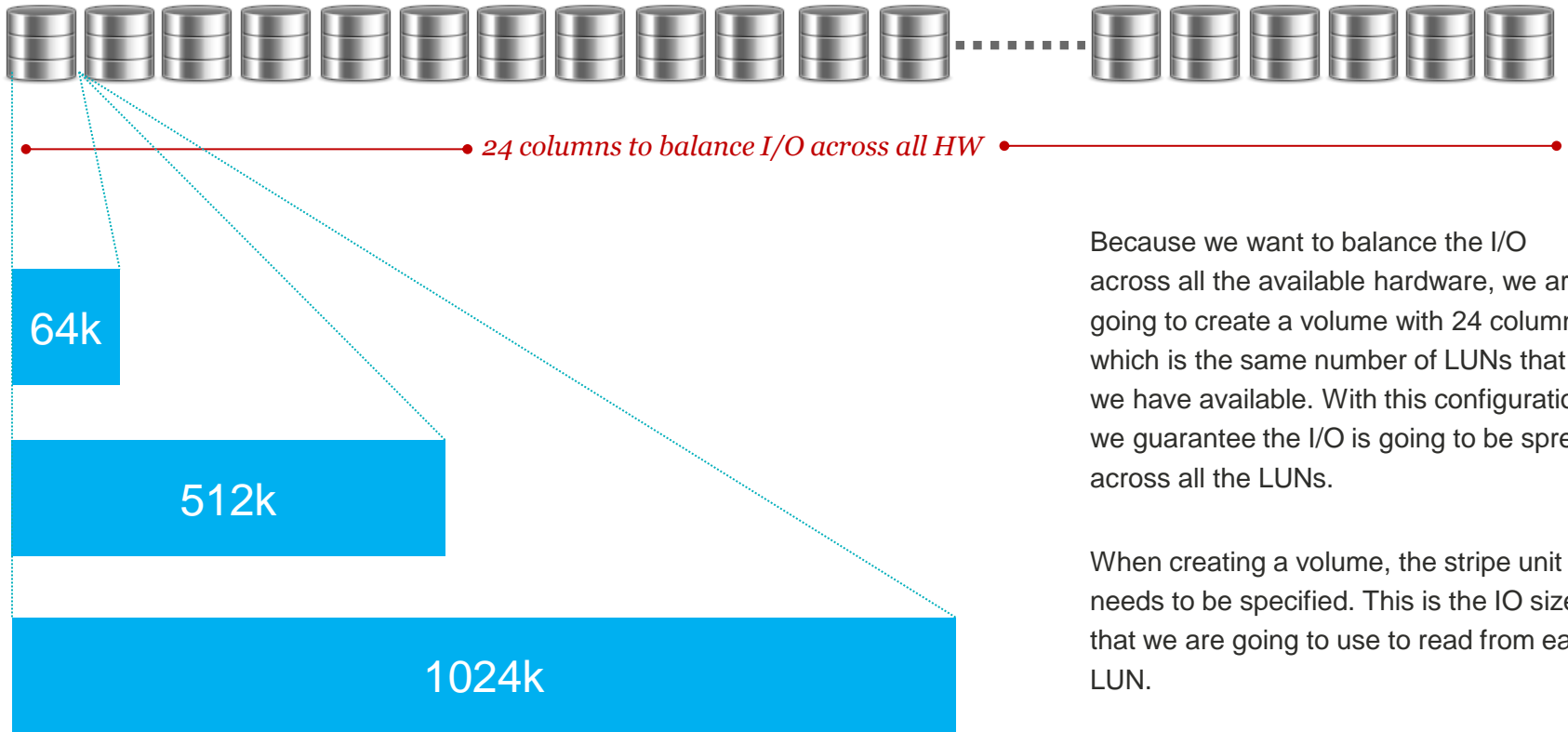
We will also see how we can take performance metrics at different levels to understand how each piece in our architecture is performing.

Maximize Volume Throughput

- + *Volume configuration*
- + *Performing sequential reads from the volume*
- + *Stripe unit*
- + *Balanced IO across the storage system*

03

VOLUME CONFIGURATION



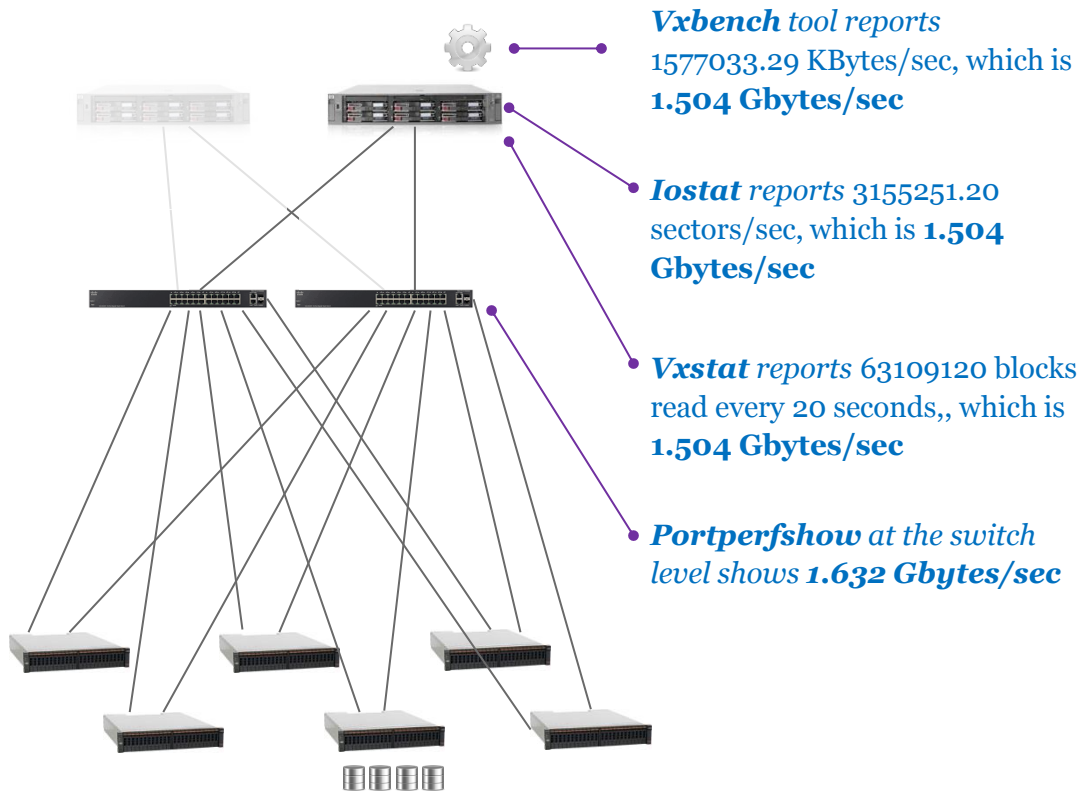
Because we want to balance the I/O across all the available hardware, we are going to create a volume with 24 columns which is the same number of LUNs that we have available. With this configuration we guarantee the I/O is going to be spread across all the LUNs.

When creating a volume, the stripe unit needs to be specified. This is the IO size that we are going to use to read from each LUN.

We performed experiments with three different stripe units sizes to understand the best choice for performance in our configuration.

PERFORMING SEQUENTIAL READS FROM THE VOLUME – SINGLE NODE

vxbench is a tool that can be used to perform sequential reads. We are reading using a block size of 1MB and 64 parallel processes. We are going to measure the performance at different levels.



Because the 8b/10b encoding overhead, the switch metrics reports a higher throughput. Using the metrics at the host level is a better approach.

At Fibre channel roadmap v1.8 the 8GFC throughput is 1600MB/s for a full duplex.

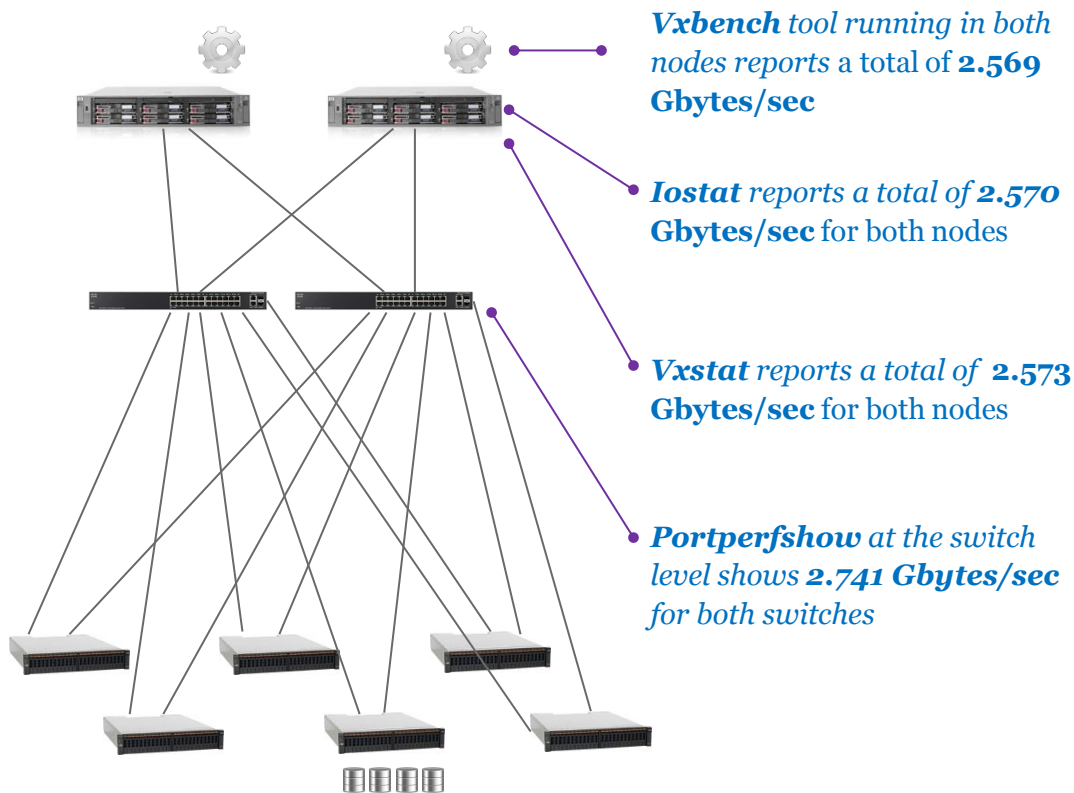
The HBA is a dual port card.
Documentation shows that it is actually **797MB/sec** for each direction.

Therefore, using our dual port card in the host, the maximum theoretical throughput will be **1.5566 GBytes/sec**, which matches with our results

*This shows the HBA bottlenecks at approx. **1.5 Gbytes/sec** in our environment.*

PERFORMING SEQUENTIAL READS FROM THE VOLUME - TWO NODES

Now vxbench is going to be run in each of the nodes. Again we measure the performance at different levels when running IO from both nodes at the same time



Again, because the 8b/10b encoding overhead, the switch metrics reports a higher throughput.

In this two node test, Vxbench only reports approx. **1.285 Gbytes/sec** per node, therefore we do not reach the per node HBA bottleneck when performing IO from both nodes.

Iostat reports for each node shows how the performance per LUN decrements when performing IO from both nodes

*This shows the storage bottlenecks at approx. **2.5 Gbytes/sec** in our environment.*

STRIPE UNIT



	1 Server	2 Servers
64k	11.5Gbits/sec	19.5Gbits/sec
512k	12.0Gbits/sec	20.5Gbits/sec
1024k	12.0Gbits/sec	20.3Gbits/sec

Running the vxbench program using different stripe units we see that the 512k width is the most optimal. We have used sequential reads with 1024k IO size.

Using one single node we reached the 12Gbits/sec (**1.5Gbytes/sec**) throughput which is limited by the HBA as we saw in the previous page.

Using two servers, we reached 20Gbits/sec (**2.5Gbytes/sec**).

From this point onwards we now know the maximum throughput achievable using our hardware configuration.

BALANCED IO ACROSS THE STORAGE SYSTEM

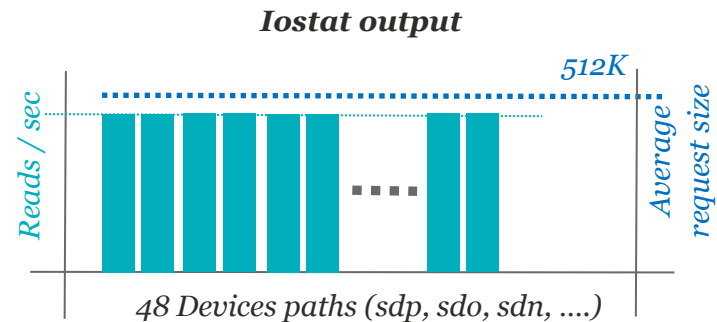
Vxbench captures the throughput per process.

Vxstat captures the throughput per disk (LUN).

lostat captures the throughput per path.

The graph shows how the IO is evenly distributed across all the paths and how the average request size for each path is 512K, which matches with the stripe unit size we used.

Balanced IO is observable across all the different metrics.



Each path to each LUN is doing the same amount of work

File System Throughput

- + *File System performance using Direct-IO reads*
- + *Performance with read ahead disabled*
- + *Enabling read ahead and tuning read_nstream*
- + *Summary*

04

FILE SYSTEM PERFORMANCE USING DIRECT-IO READS

File System direct IO mimics the Volume Manager raw disk test. To make it more clear we are creating a file with a single extent so all the blocks are contiguous. Then we can use vxbench with `–direct` option to read the file using direct IO.

As Direct IO is used, each read will fetch data directly from disk, so no buffering is being performed, therefore data is not being pre-fetched from disk, i.e. no read ahead is being performed.

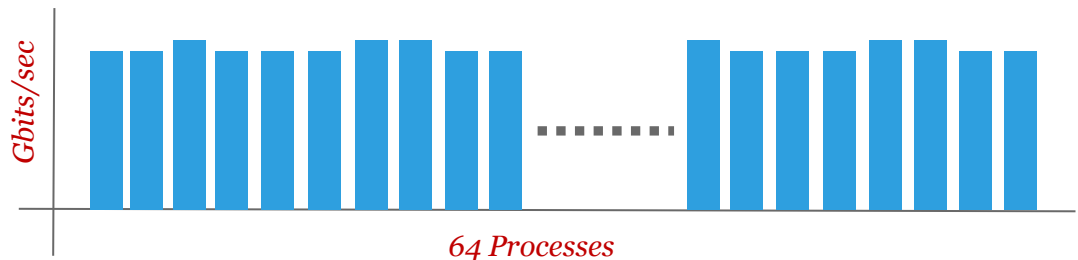
The sequential read throughput was the same using the file system with direct IO as reading from the Volume Manager raw disks devices.

In this direct IO test all 64 processes read from the same file and all see similar throughput. In the previous raw disk test all 64 processes read from the same device and also see similar throughput. In both test maximum possible throughput is achieved.

```
$ touch /data1/file1
$ /opt/VRTS/bin/setext -r 4194304 -f contig /data1/file1
$ dd if=/dev/zero of=/data1/file1 bs=128k count=262144

$ /opt/VRTS/bin/fsmap -A /data1/file1
Volume  Extent Type      File Offset      Dev Offset      Extent Size  Inode#
  vol1           Data              0      34359738368      34359738368    4
$ ls -lh /data1/file1
-rw-r--r-- 1 root root 32G Mar  3 14:12 /data1/file1
```

Each of the 64 processes is achieving similar throughput



FILE SYSTEM PERFORMANCE WITH BUFFERED IO SEQUENTIAL READS

To perform the test using Buffered IO, independent files for each processes are going to be created. 16GB of data will be pre-allocated to each file.

As we want to make sure that data is not in memory, the file system can be remounted (-o remount) between each test to make sure that all reads will be coming from disk.

Each process will now be reading from a different file and the vxbench IO block size will be 32K, as this is closer to real world workloads.

The greatest impact to the performance of sequential reads when using buffered IO is **read ahead**. It asynchronously pre-fetches data into memory, providing clear benefits for sequential read performance.

There are two parameters that control *read ahead* behaviour.

read_pref_io

This parameter means the “preferred read IO size”. It will be the **maximum IO request** submitted by the file system to the volume manager.

The default value is set by the volume manager stripe unit, so in our case it will be 512K (524228 bytes).

If an application is requesting only 8K reads, *read ahead* will pre-fetch the file data using IO requests of size 512K to the volume manager. This converts the 8K application reads into 512K disk reads.

The larger IO size reading from disk improves read performance, 512k is our optimal size.

It is not recommended to tune read ahead by changing the value of *read_pref_io*, instead we can tune the value of *read_nstream* if it is needed.

read_nstream

This parameter defaults to the number of columns (LUNs) in the volume. In our configuration the default value will be 24.

*read_pref_io * read_nstream determines the max amount of data that is pre-fetched from disk using read_ahead.*

To reduce the amount of read ahead this parameter can be reduced.

In our example, as we are using 24 columns with 512K stripe unit, the max amount of read ahead will be 12MB. As we will see this will be too much data to read ahead and will cause an imbalance in read IO performance between the processes.

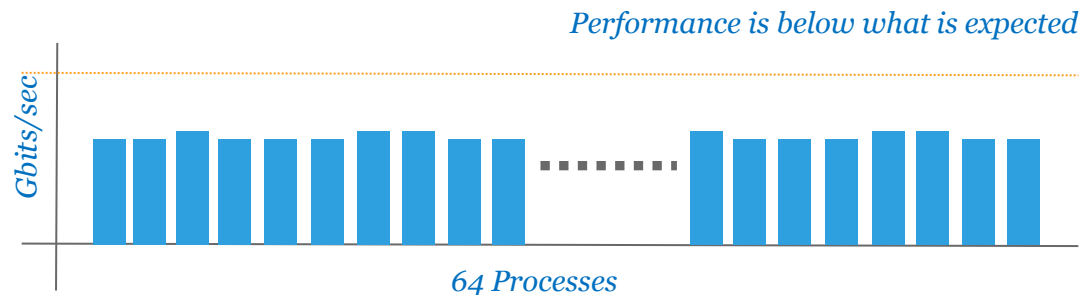
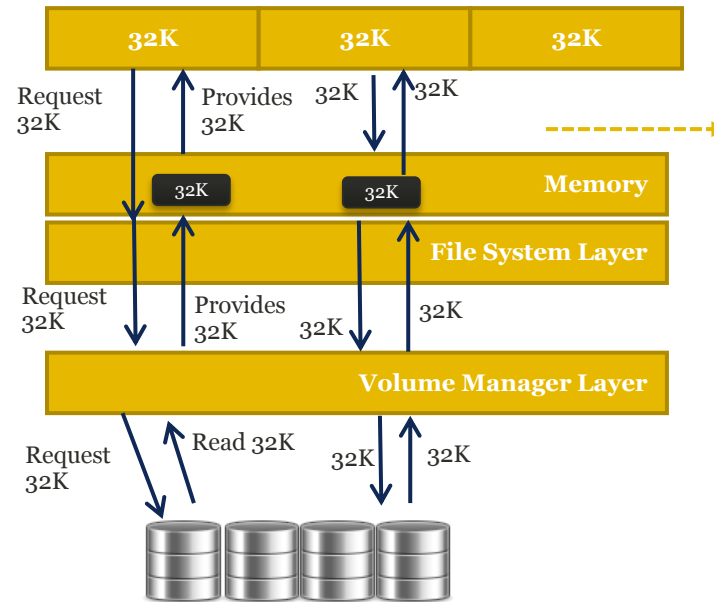
IO SIZES WITH READ AHEAD DISABLED

When read ahead is disabled, the maximum throughput threshold cannot be achieved.

The maximum throughput per node cannot be achieved when read ahead is disabled in our test. Although the throughput is evenly balanced across the processes, the total throughput is reduced by half.

`read_pref_io` is only used if read ahead is enabled. `vxbench` is reading 32K at a time and because read ahead is disabled we will only read a maximum of 32K at a time from disk (the size of the `vxbench` reads).

In this example we are using 64 process reading 64 files.

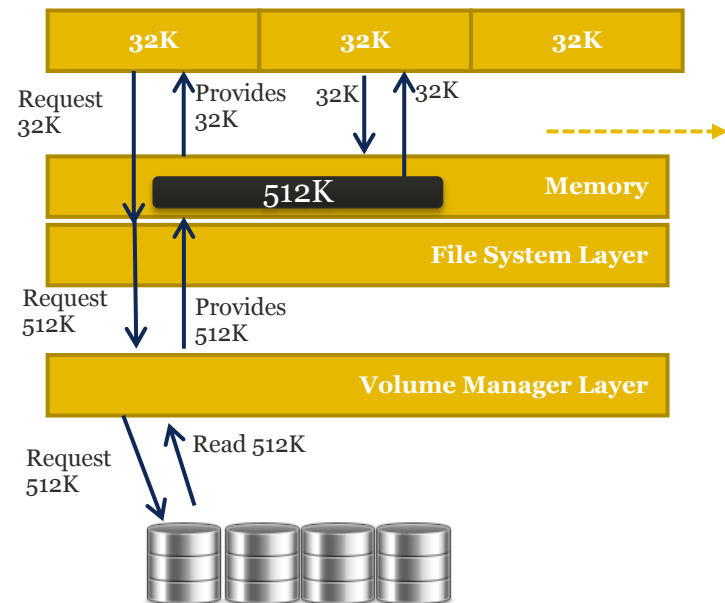


IO SIZES WITH READ AHEAD ENABLED

Read ahead improves sequential read performance by increasing the IO size and prefetching data from disk.

Now read ahead is enabled and vxbench is still reading 32K at a time. However now we are reading a maximum of 512K at a time from disk because `read_pref_io` is set to 512K. This larger IO size improves performance.

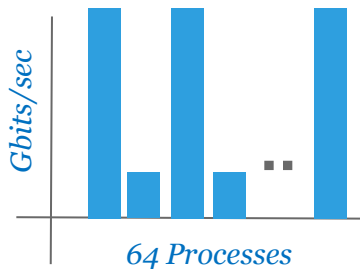
Also data is being pre-fetched from disk before vxbench has requested it, so the next vxbench IO request will fetch the data directly from memory without having to go to the disk.



ENABLING READ AHEAD AND TUNING READ_NSTREAM

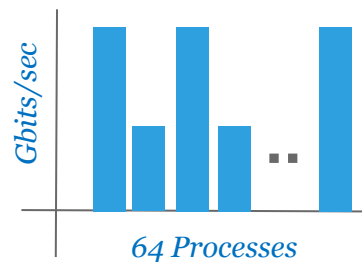
*Reading too much data in advance can create an imbalance between processes. Remember that $\text{read_pref_io} * \text{read_nstream}$ determines the max amount of data pre-fetched from disk.*

$\text{read_nstream} = 24$



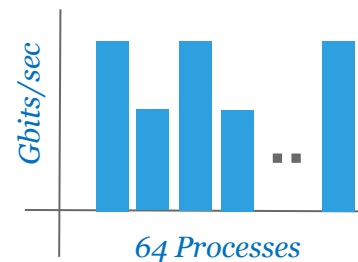
24 x 512K = 12MB

$\text{read_nstream} = 12$



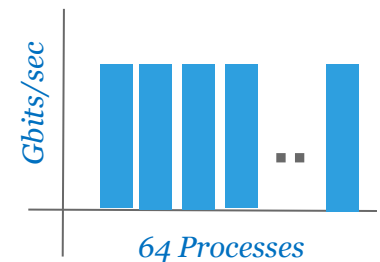
12 x 512K = 6MB

$\text{read_nstream} = 6$



6 x 512K = 3MB

$\text{read_nstream} = 1$



1 x 512K = 512K

With read ahead enabled all the tests achieved the max amount of throughput, however the IO is not evenly balanced between the processes until read_nstream is set to one

The amount of data that is pre-fetched is reduced as read_nstream is reduced. By default, read_nstream was 24 in our configuration, which was pre-fetching 12MB of data at a time, which is too aggressive.

READ AHEAD TUNING SUMMARY

We have seen how read ahead parameters obtain their default values from the stripe unit size and the number of columns in the volume. In our test *read_pref_io* default was 512K and *read_nstream* default was 24 based in the number of columns used. This meant that each process would pre-fetch 12MB of data from disk at a time.

Pre-fetching this amount of data causes an imbalance in throughput between the processes reading from different files.

Our goal was to maintain the total performance of **1.5Gbytes/sec** but also to maintain a balanced throughput across all the processes.

When tuning read ahead parameter values, our recommendation is to reduce read_nstream rather than read_pref_io.

When adjusting the parameters it is important to consider the number of running processes that will be reading from disk at the same time, as the available throughput will be distributed between these processes.

Below graphic shows how disabling read ahead resulted in poor performance and how in our case using read ahead with *read_nstream=1* provided the maximum throughput and perfectly balanced performance across all 64 processes.

