Cockroach LABS

2018 Cloud Report

Written by
Masha Schneider & Andy Woods

Table of Contents

Introduction

Experiments

TPC-C Performance **5**
CockroachDB TPC-C Throughput on AWS vs GCP 5
3-Node TPC-C Throughput: AWS Machine Types with SSDs vs. GCP 5
3-Node TPC-C Throughput: AWS Machine Types with EBS vs. GCP 5
3-Node TPC-C Throughput: c5 Series with SSD, EBS-io1, and EBS-gp2 6
3-Node TPC-C Throughput: AWS Machine Type with Nitro System 6

CPU Experiment **7**
CPU Throughput: AWS vs GCP 7

Network Experiment **8**
Histogram of Network Throughput: AWS vs GCP 8
Histogram of Network Latency: AWS vs GCP 10

I/O Experiment **11**
I/O Write Performance With nobarrier: AWS vs GCP 12
I/O Write Performance Without nobarrier: AWS vs GCP 12
I/O Write Latency With nobarrier: AWS vs GCP 13
I/O Write Latency Without nobarrier: AWS vs GCP 13
I/O Read Performance Without nobarrier: AWS vs GCP 14
I/O Read Performance With nobarrier: AWS vs GCP 14
Avg and 95th Read Latency With nobarrier: AWS vs GCP 15
Avg and 95th Read Latency Without nobarrier: AWS vs GCP 15

Cost
TPC-C Price per Performance: AWS vs GCP 16

Conclusion

Reproduction Steps

Introduction

Our customers rely on us to help them navigate the complexities of the increasingly competitive cloud wars. Should they use Amazon Web Services (AWS)? Google Cloud Platform (GCP)? Microsoft Azure? How should they tune their workload for different offerings? Which is more reliable?

We are committed to building a cloud neutral product, and we run test clusters on all three leading US cloud providers. As we were testing features for our 2.1 release, we noticed something interesting: AWS offered 40% greater throughput than GCP.

We were curious as to why AWS offered such a stark difference in throughput, and set out to test the performance of GCP and AWS in more detail. Ultimately, we compared the two platforms on TPC-C performance (e.g., throughput and latency), CPU, network, I/O, and cost.

This inspired what has become the 2018 Cloud Report.

Our conclusion? AWS outperforms GCP on nearly every criteria we tested — including cost.

NOTE: We did not test Microsoft Azure due to limits on internal resources, but we plan to do so in the near future.

Machine Type

GCP has a variety of instance types (including standard and high CPU) but we focused on the `n1-standard-16` machine with Intel Xeon Scalable Processor (Skylake) in the us-east region (Skylake offers a marginal 4% improvement over standard hardware on `n1-standard-16`). We were familiar with this instance type as we used it to conduct our previous performance benchmarking.

A similar configuration is not quite as trivial as it sounds for AWS. AWS has more flavors of instances than GCP. It has the standard high CPU and general instances. We chose the latest compute-optimized AWS instance type, `c5d.4xlarge` instances, to `match n1-standard-16`, because they both have 16 CPUs and SSDs (although AWS only offers 32 GB of RAM as compared to 60 GB of RAM on GCP) within the us-east-2 region.

For those readers not familiar with AWS: the first letter `c` corresponds to machine type; the number `5` is the generation; `d` corresponds to SSD; and the `4xlarge` corresponds to CPU. Each of the most popular AWS machine types tested here varies the machine type, the generation, SSD or EBS, but not the CPU when evaluating AWS on TPC-C performance.

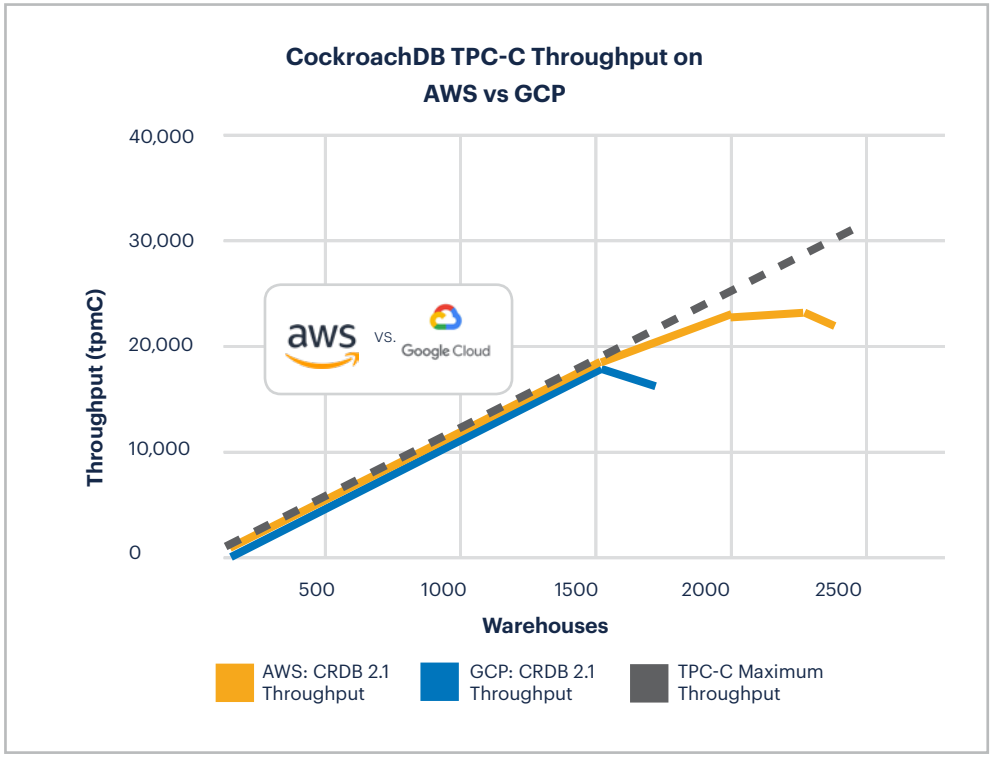
Experiments

We designed our first experiment to tease out whether or not AWS and GCP performance differed on a simulated customer workload. We started with a customer workload (and not micro-benchmarks) because it most directly simulates real-world customer behavior.

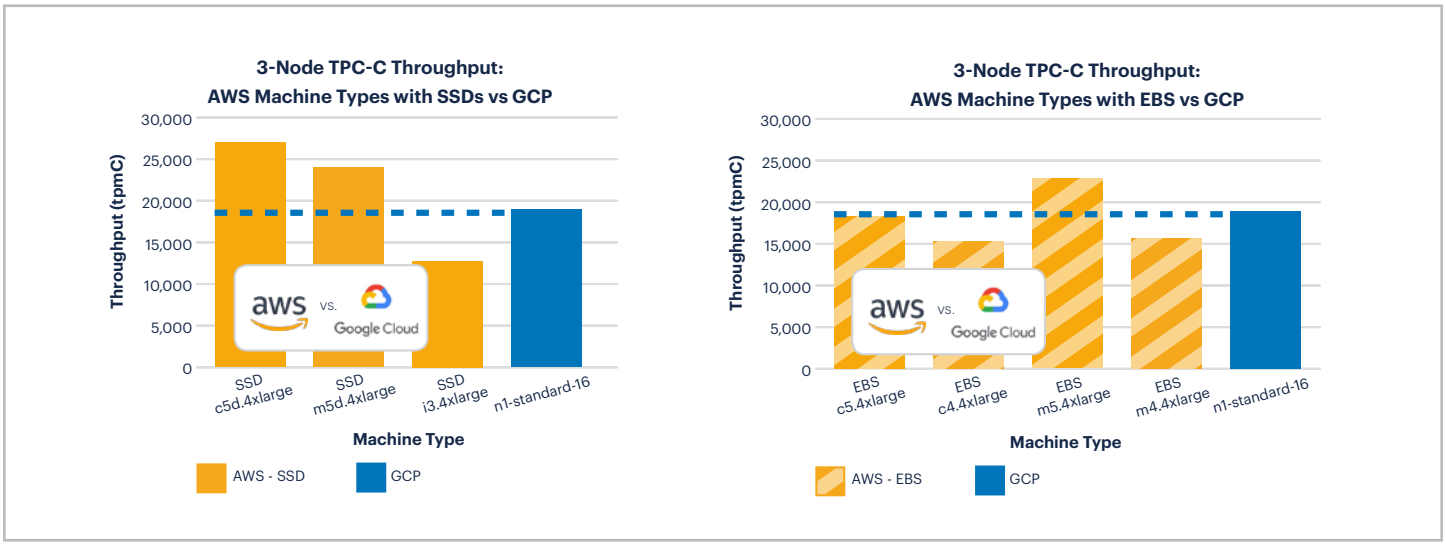
It was only after observing differences in applied workloads that we moved onto micro-benchmarks like CPU, network, and I/O performance. Differences in micro-benchmarks matter more when informed by the knowledge that the overall customer workload performance of the platforms differ. CPU, network, and I/O all represent separate hypothesis for why performance might vary between GCP and AWS.

TPC-C Performance

We chose to test workload performance by using TPC-C, a popular OLTP benchmark tool that simulates an e-commerce business, given our familiarity with this workload. These results were collected using `nobarrier`. See page 11 for a greater explanation.

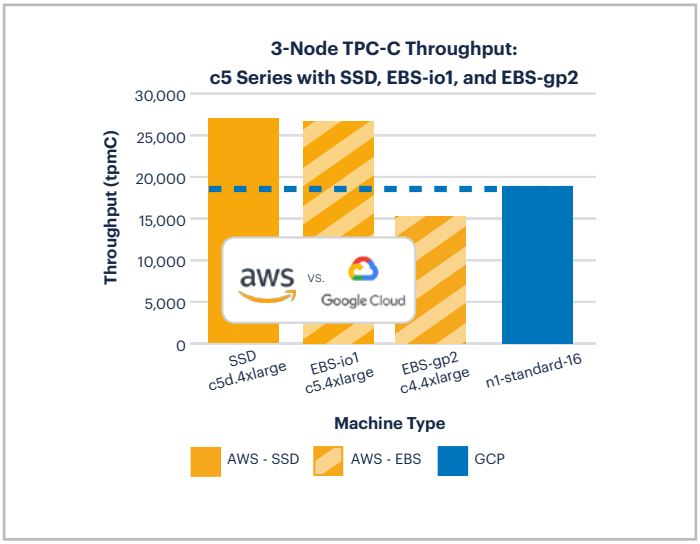


CockroachDB 2.1 achieves 40% more throughput (tpmC) on TPC-C when tested on AWS using `c5d.4xlarge` than on GCP via `n1-standard-16`. We were shocked that AWS offered such superior performance. Previously, our internal testing suggested more equitable outcomes between AWS and GCP. We decided to expand beyond the `c5` series to test TPC-C against some of the most popular AWS instance types.

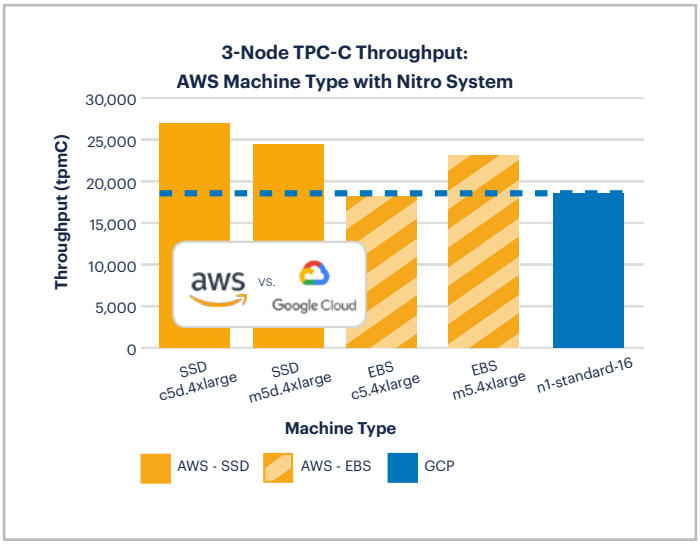


At first blush, it appears that SSDs offered by `c5d` and `m5d` outperform EBS. Unfortunately, it's a bit more complicated than that as AWS offers EBS out of the box with gp2 volume types rather than the higher performing io1 volume type.

To isolate this change, we focused on the higher performing `c5` series with SSDs, EBS-gp2, and EBS-io1 volume types:



Clearly, EBS volumes offer effective performance if tuned to the io1 volume type and provided with sufficient iOPS. So if the difference in TPC-C performance observed among various AWS instance types is not explained by SSD vs. EBS, what else might explain it? AWS recently introduced their new Nitro System present in `c5` and `m5` series. The AWS Nitro System offers approximately the same or superior performance when compared to a similar GCP instance.



The results were clear: AWS wins on TPC-C benchmark performance. But what causes such large performance differentials? We set out to learn more by testing a series of micro-benchmarks on CPU, network, and I/O.

CPU Experiment

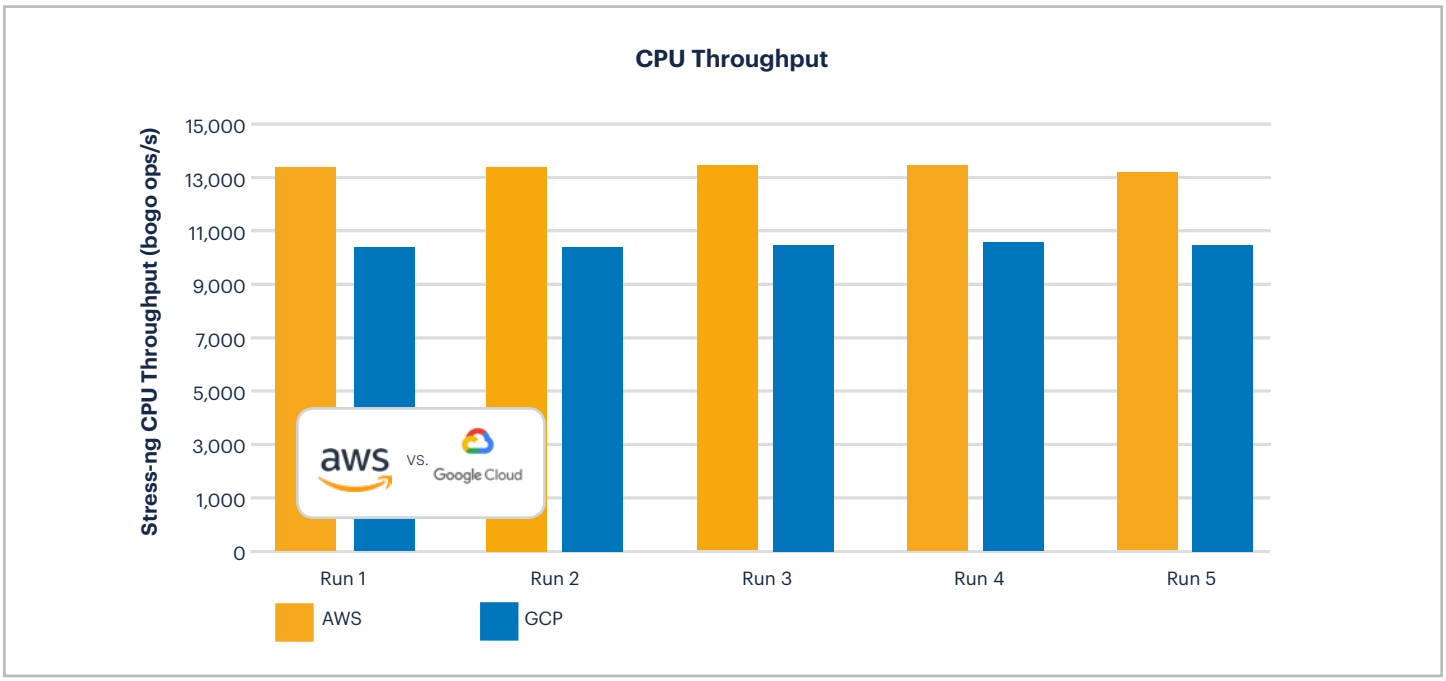
We began our micro-benchmark testing like any aspirational scientists by seeking to disprove our main hypothesis: that differences in AWS and GCP provisions might affect CPU performance.

We focused on a CPU performance micro-benchmark first as CPU can have a large impact on performance.

To test CPU performance, we evaluated third party benchmarks based on popularity and ease of use. The two most frequently used benchmark test suites in the market today are sysbench and stress-ng. We chose stress-ng because it offered more benchmarks and provided more flexible configurations than sysbench.

We ran the following Stress-ng command five times on both AWS and GCP:

```
stress-ng --metrics-brief --cpu 16 -t 1m
```



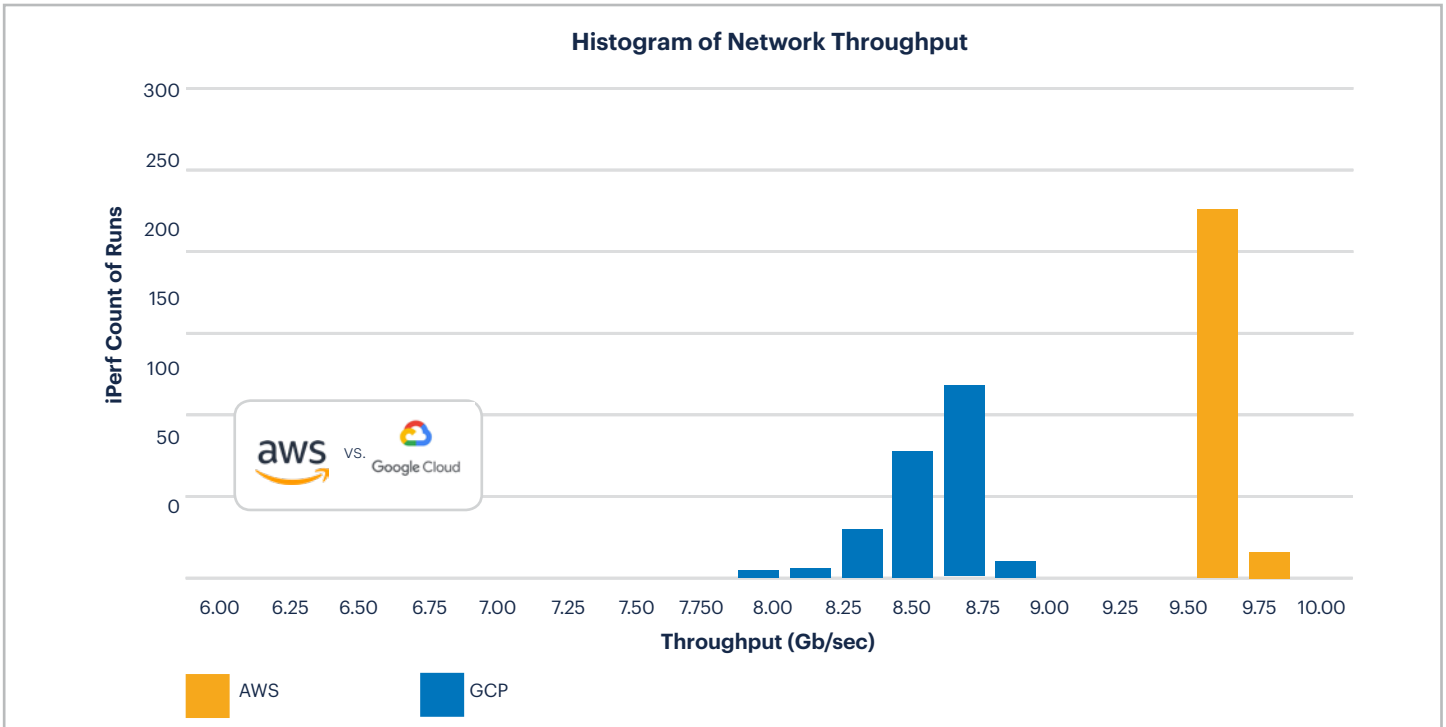
> AWS offered 28% more throughput (~2,900 bogo ops/s) on stress-ng than GCP. Both AWS and GCP offered generally consistent CPU performance across runs. This is a credit to the investments made by both platforms as unpredictability can have a material cost for business paid in the over-provisioning of virtual machines.

Now that we observed an initial difference in both CPU performance on GCP and AWS, we couldn't help ourselves from continuing to investigate other potential differences. Was the entirety of the TPC-C difference generated from the advantage in CPU performance?

Network Experiment

Next, we tested the network throughput and latency. To test network, we measured throughput using a popular tool called iPerf and latency via another popular tool ping.

iPerf's configurations include a buffer data size (128KB), a protocol, a server and and a client. iPerf attempts to connect the client and the server with the data from buffer size via the protocol. We set up iPerf similarly to this blog post. This test provides a throughout for the network which allows for us to compare the performance of the network on AWS and GCP. We ran the test four times each for AWS and GCP and aggregated the results of all four tests in histograms (each 1 sec run is stacked to form this chart).

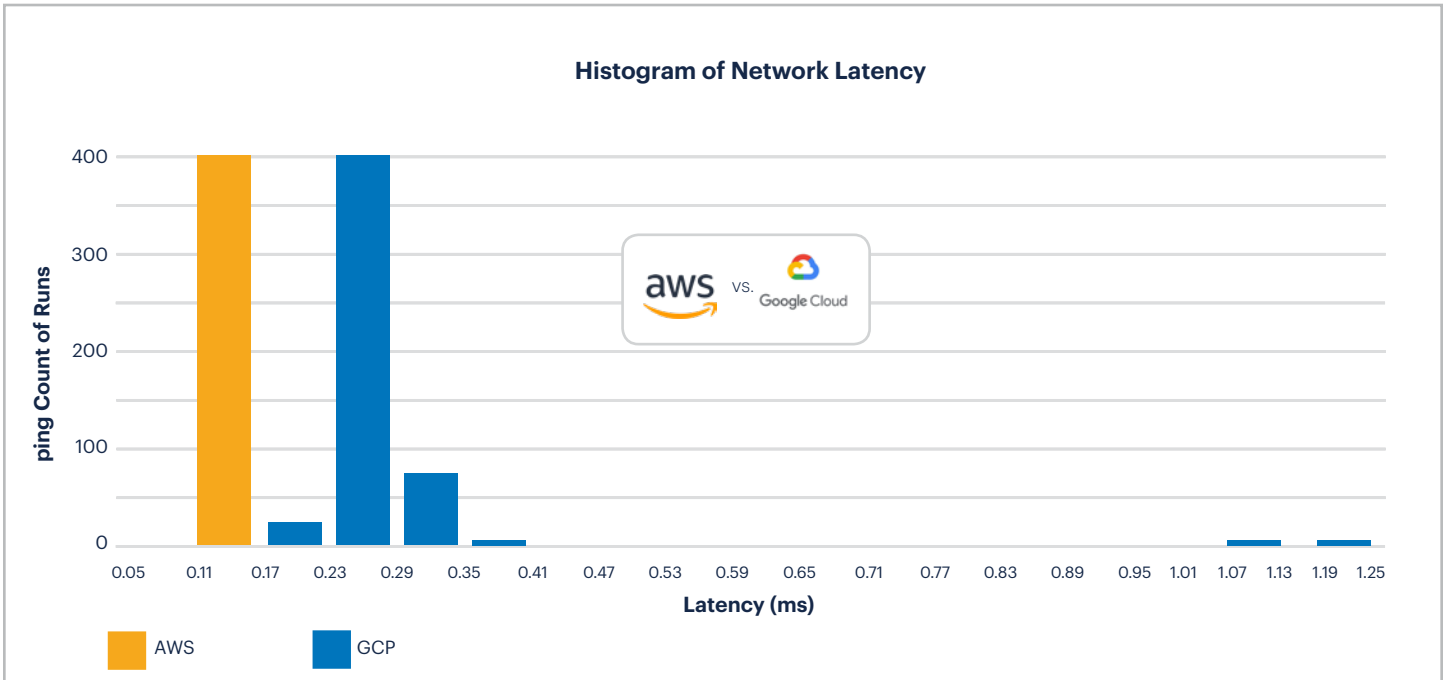


	SUM Throughput (Gb)		AVERAGE Throughput (Gb/sec)		STDEV Throughput (Gb/sec)		MIN Throughput (Gb/sec)		MAX Throughput (Gb/sec)	
RUN	GCP	AWS	GCP	AWS	GCP	AWS	GCP	AWS	GCP	AWS
1	508.66	576.43	8.62	9.61	0.18	0.007	7.97	9.60	8.87	9.63
2	517.78	576.48	8.63	9.61	0.20	0.007	7.62	9.60	8.98	9.63
3	520.50	576.25	8.68	9.60	0.11	0.005	8.34	9.60	8.86	9.62
4	516.15	566.67	8.60	9.60	0.18	0.005	8.01	9.60	8.91	9.62
Grand Total	**2063.09**	**2295.83**	**8.63**	**9.61**	**0.17**	**0.006**	**7.62**	**9.60**	**8.98**	**9.63**

GCP shows a skewed left normal distribution of network throughput centered at ~8.6 Gb/sec. In addition to the raw network throughput, we also care about the variance of the network throughput so that we can have consistent expectations for the network performance. Throughput ranges from 7.62 Gb/sec to 8.91 Gb/sec — a somewhat unpredictable spread of network performance, reinforced by the observed average variance for GCP of 0.17 Gb/sec.

AWS, on the other hand, offers higher throughput, centered on 9.6 Gb/sec, while providing a tighter spread between 9.60 Gb/sec and 9.63 Gb/sec when compared to GCP. On AWS, iPerf transferred a total network throughput of 2,296 Gb in 60 seconds across all four runs. This is an increase of 11% over GCP. On average this is more than 1 Gb/sec increase in throughput.

What about network throughput standard deviation? On AWS, the standard deviation is only 0.006 Gb/sec. This means that the GCP network standard deviation of 0.17 Gb/sec is ~27x more than on AWS.



	AVERAGE of Latency		STDEV of Latency		MIN of Latency		MAX of Latency	
Run	AWS	GCP	AWS	GCP	AWS	GCP	AWS	GCP
1	0.056	0.227	0.003	0.106	0.052	0.164	0.072	1.250
2	0.057	0.217	0.003	0.090	0.054	0.153	0.077	1.080
3	0.057	0.221	0.003	0.094	0.051	0.167	0.075	1.120
4	0.057	0.227	0.003	0.106	0.053	0.162	0.074	1.240
Grand Total	**0.057**	**0.223**	**0.003**	**0.099**	**0.051**	**0.153**	**0.077**	**1.250**

Like network throughput, AWS has a tighter network latency than GCP. Looking at the data closely, we can see that there are severeal outliers that the max latency, 1.25 ms, is more than 5 times the average.

Similarly to network throughput, AWS offers a stark difference to GCP.

AWS's values are centered on an average latency, 0.057 ms. **In fact the spread is so tight it can't be visualized on the same scale as GCP.** The max latency is only 0.077 ms — a difference of only .02 ms (or 35%) from the average!

AWS offers significantly better network throughput and latency with none of the variability present in GCP. Further, it looks like Amazon may be racing further ahead in network performance with the introduction of the `c5n` machine type that offers significantly higher network performance across all instance sizes as compared to the rest of the c series.

I/O Experiment

In this section, we investigate the maximum I/O performance attainable when the application is able to tolerate unreliable writes to its disk (e.g., when VMs are ephemeral). We tested I/O with the same machine types as above (using c5d.4xlarge and n1-standard-16) using a configuration of sysbench that simulates small writes with frequent syncs for both write and read performance. This test measures throughput and latency based on a fixed set of threads, or the number of items concurrently writing to disk.

We ran this experiment with `nobarrier` and without `nobarrier` (don't you just love double negatives?).

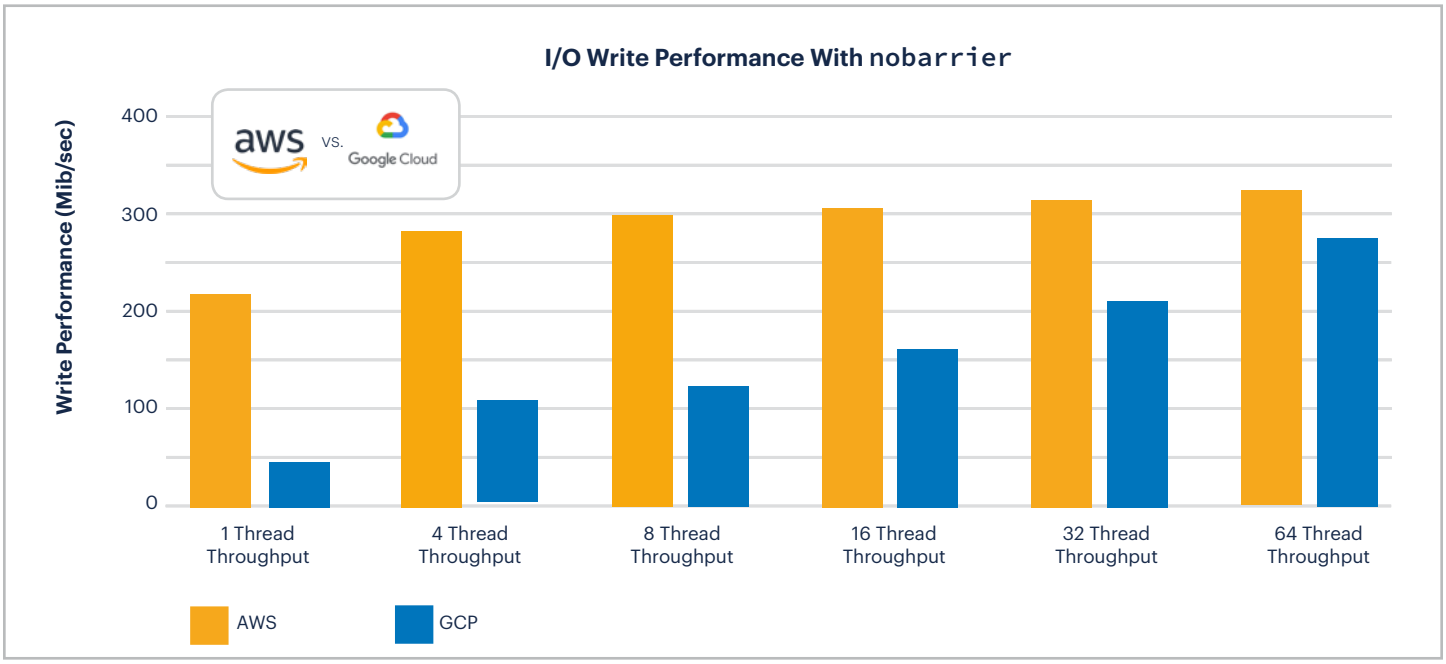
Why `nobarrier` Matters

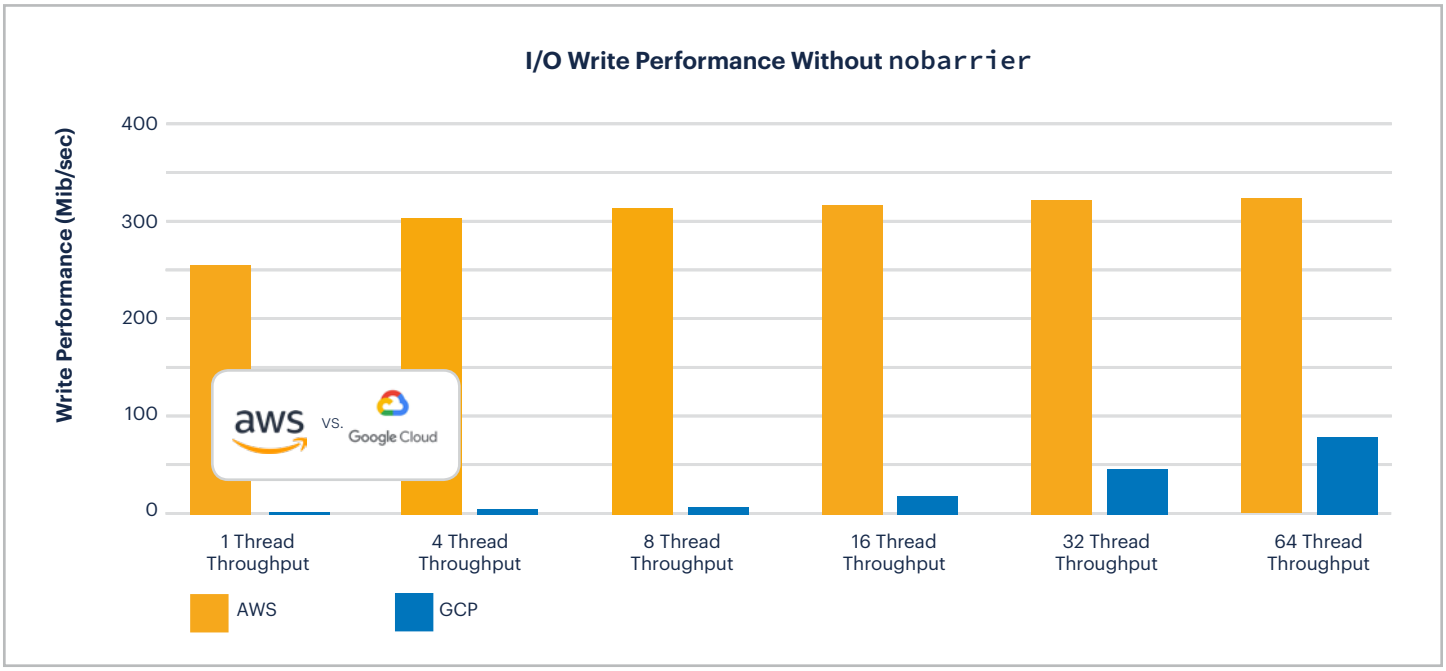
As a refresher, `nobarrier` is a method of writing directly to disk without guaranteeing that writes will be persisted in the same order in which they were performed. In many cases, `nobarrier` can offer superior performance, but at the risk of data loss. In the event of a power failure, data on disk can be corrupted causing you to lose meaningful data. In traditional deployments, `nobarrier` is used with battery-backed write caches. In cloud environments it is difficult to tell exactly what hardware is used, but it's also unnecessary: local SSDs do not survive a reboot of the host machine, so it is safe to set nobarrier on a cloud local SSD.

We conduct most of our performance tests using nobarrier to demonstrate the best possible performance but understand that not all use cases can support this option. As a result, we tested I/O with and without nobarrier on GCP and AWS. Note, `nobarrier` has no impact on CPU or network testing and was not used to conduct those experiments.

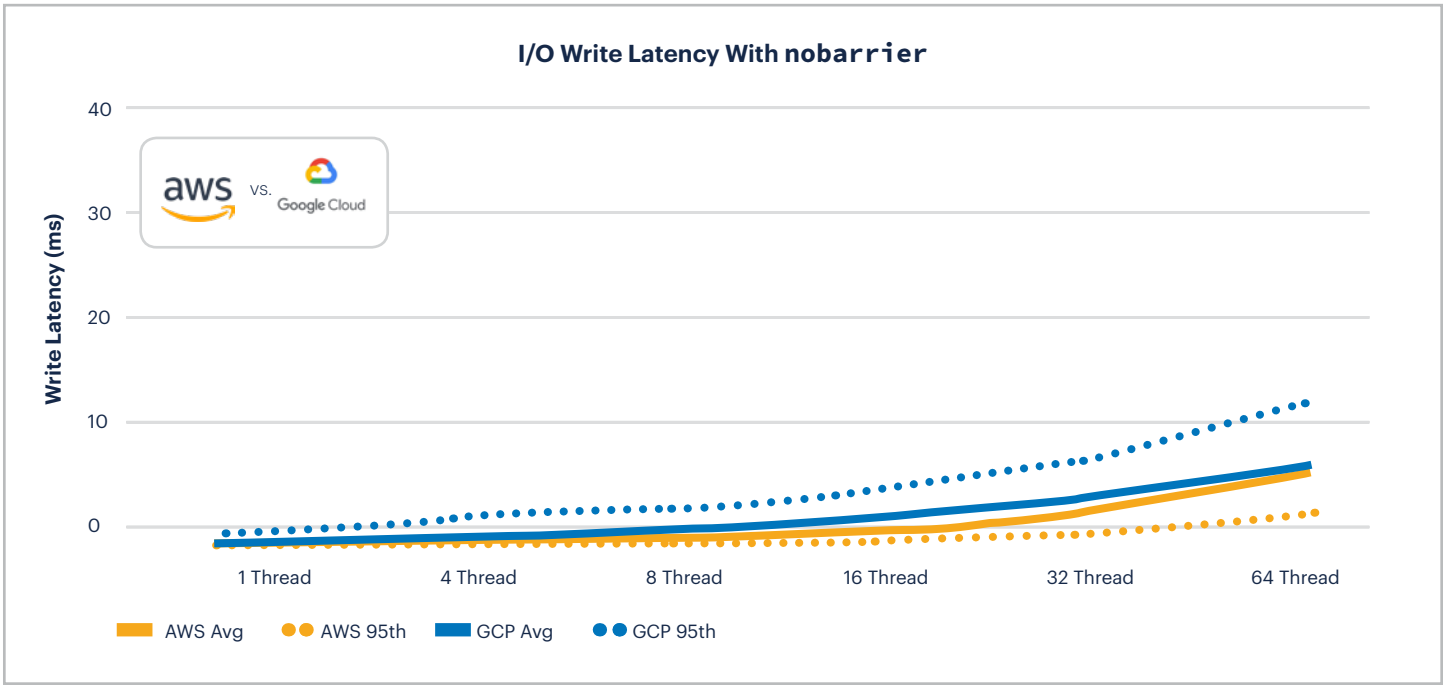
I/O Write Performance: With `nobarrier` & Without `nobarrier`

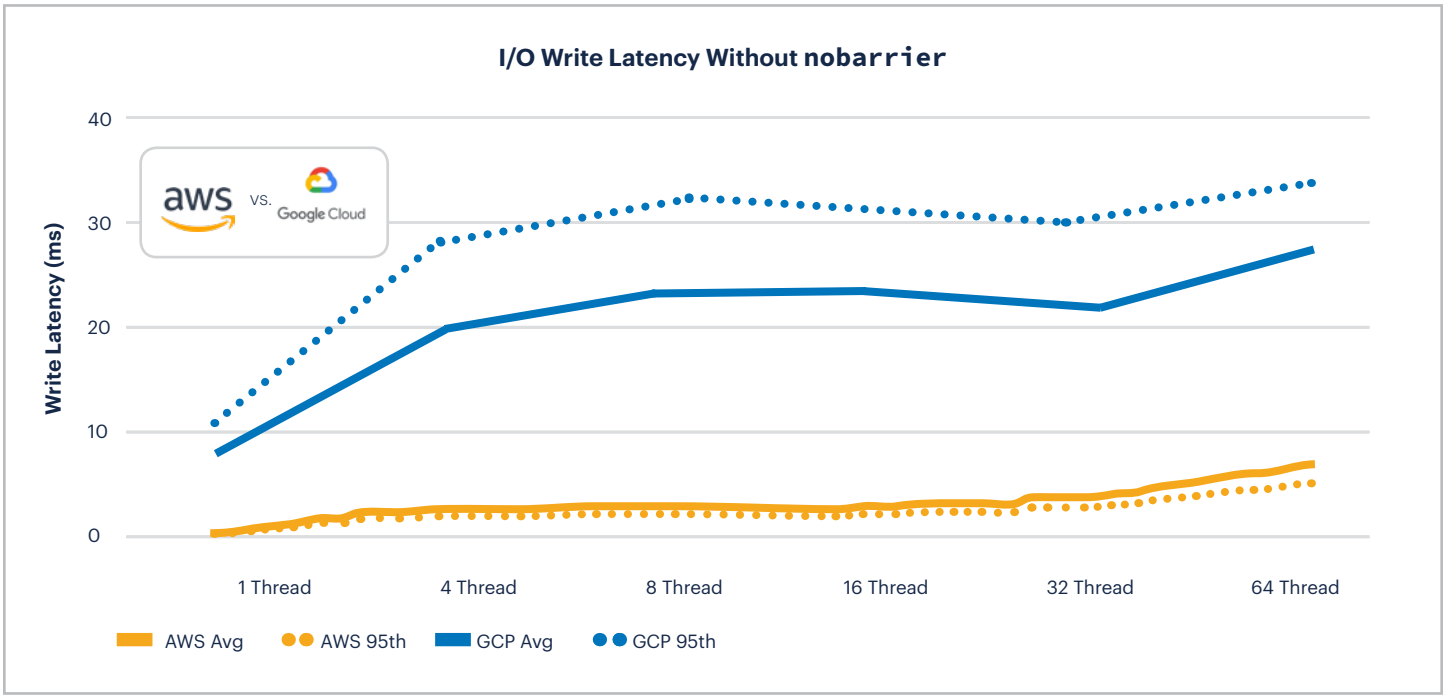
First, we tested write performance:





AWS consistently offers more write throughput across all thread variance from 1 thread up to 64 both with nobarrier and without. In fact, it can be as high as 67x difference in throughput on 1 thread throughput without `nobarrier`. It's also easy to observe that GCP benefits from `nobarrier` to a much greater degree than AWS. It's unclear why this difference between cloud providers exists on `nobarrier` and we will avoid speculating but do hope to learn more.

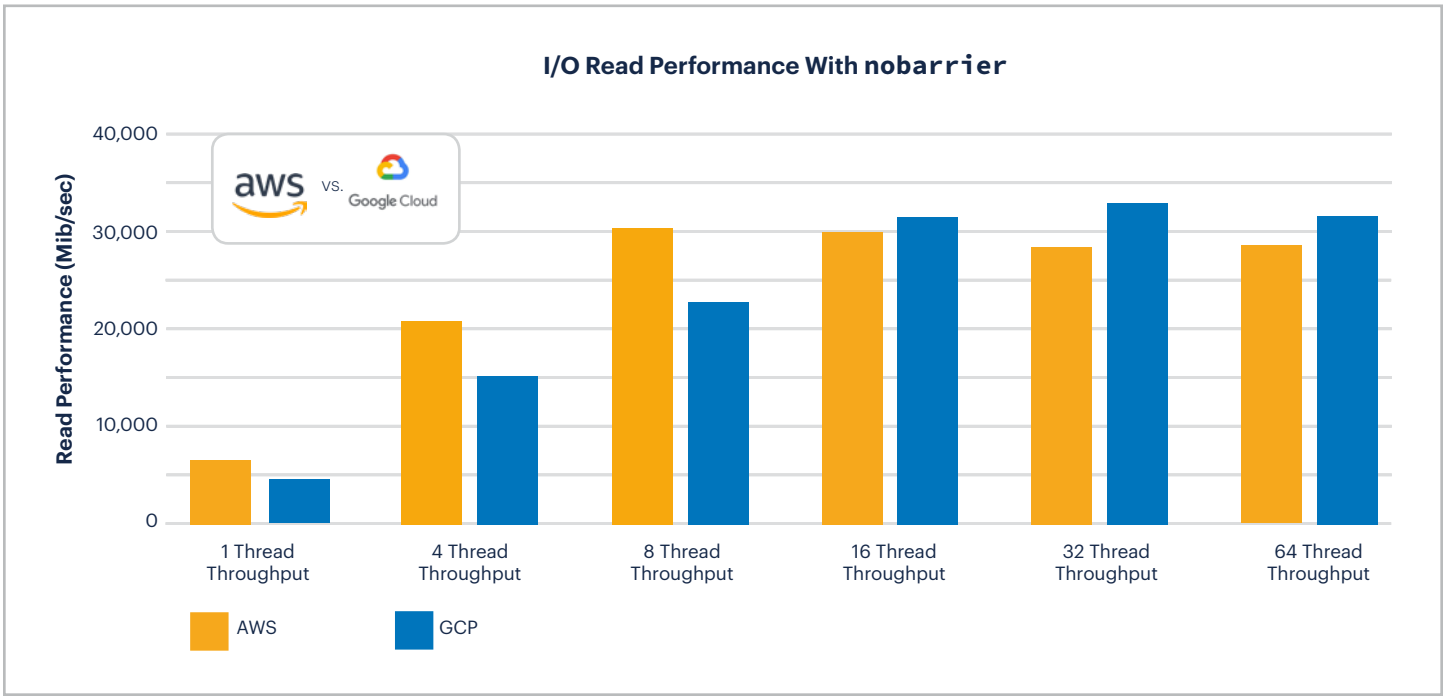


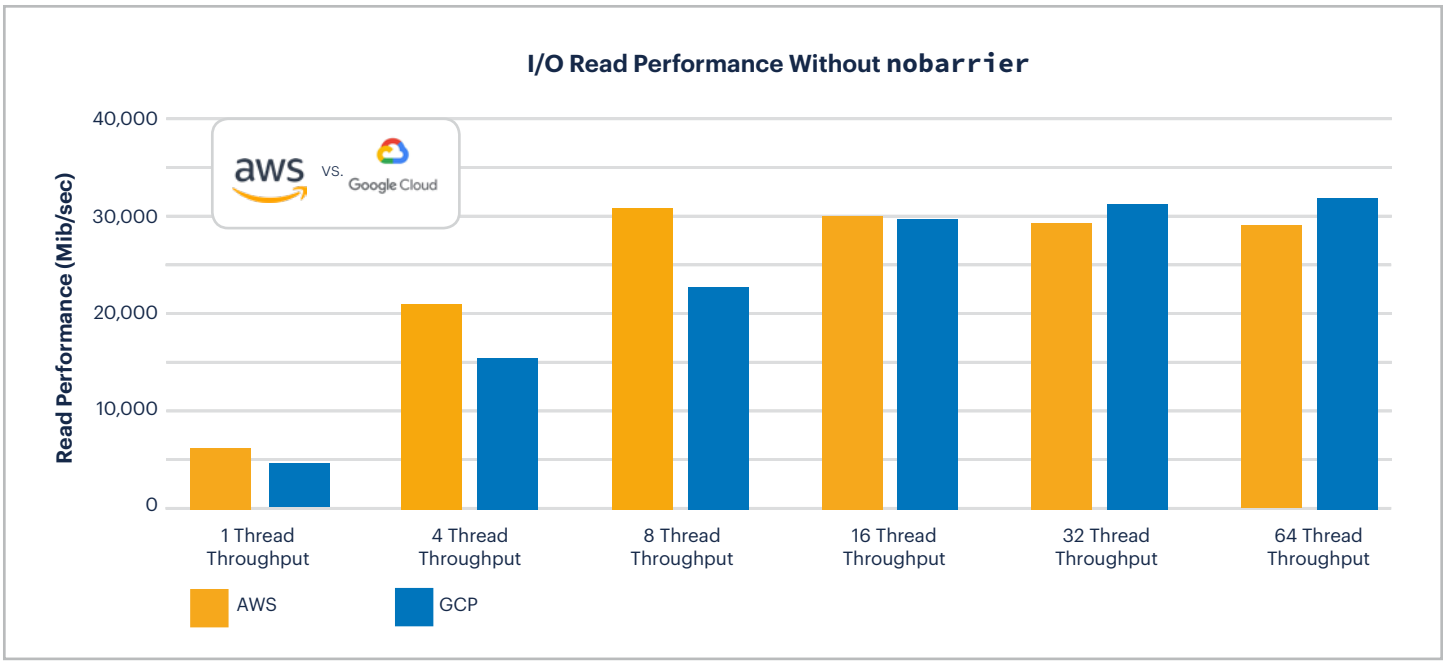


AWS also offers better average and 95th percentile write latency across all thread tests with and without `nobarrier`.

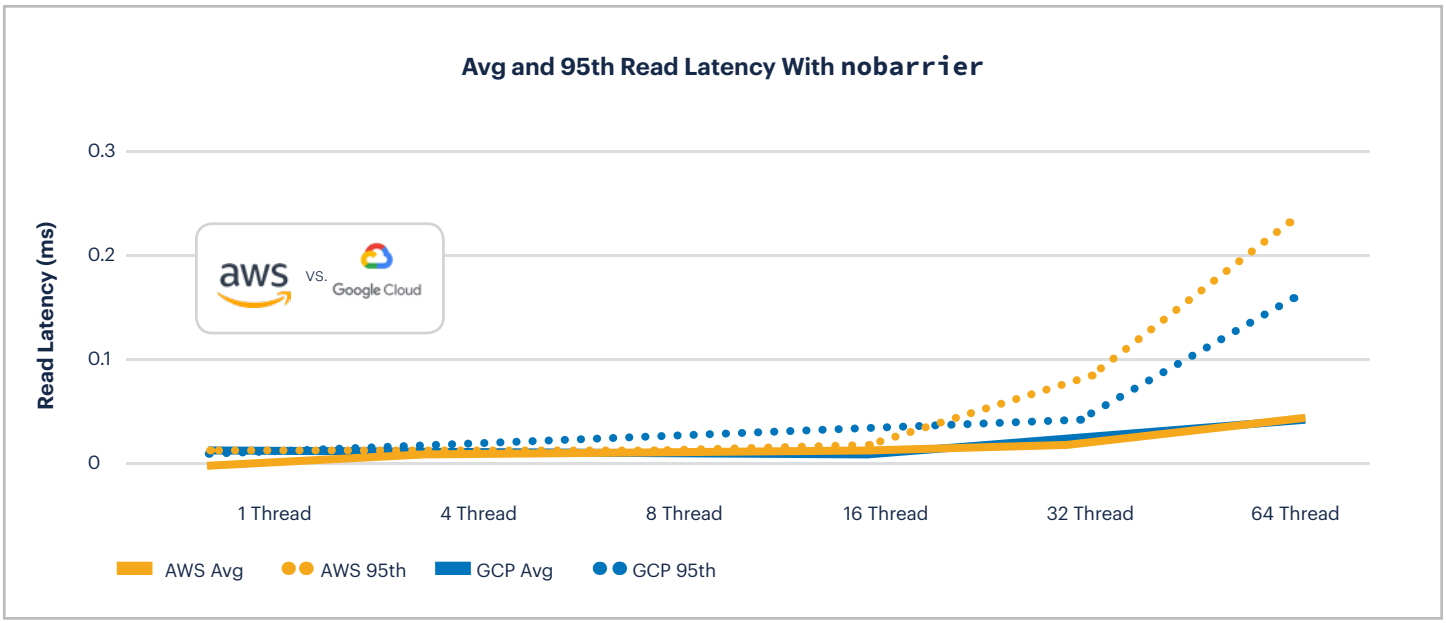
AWS held clear advantages in write throughput and latency. What about read throughput and latency?

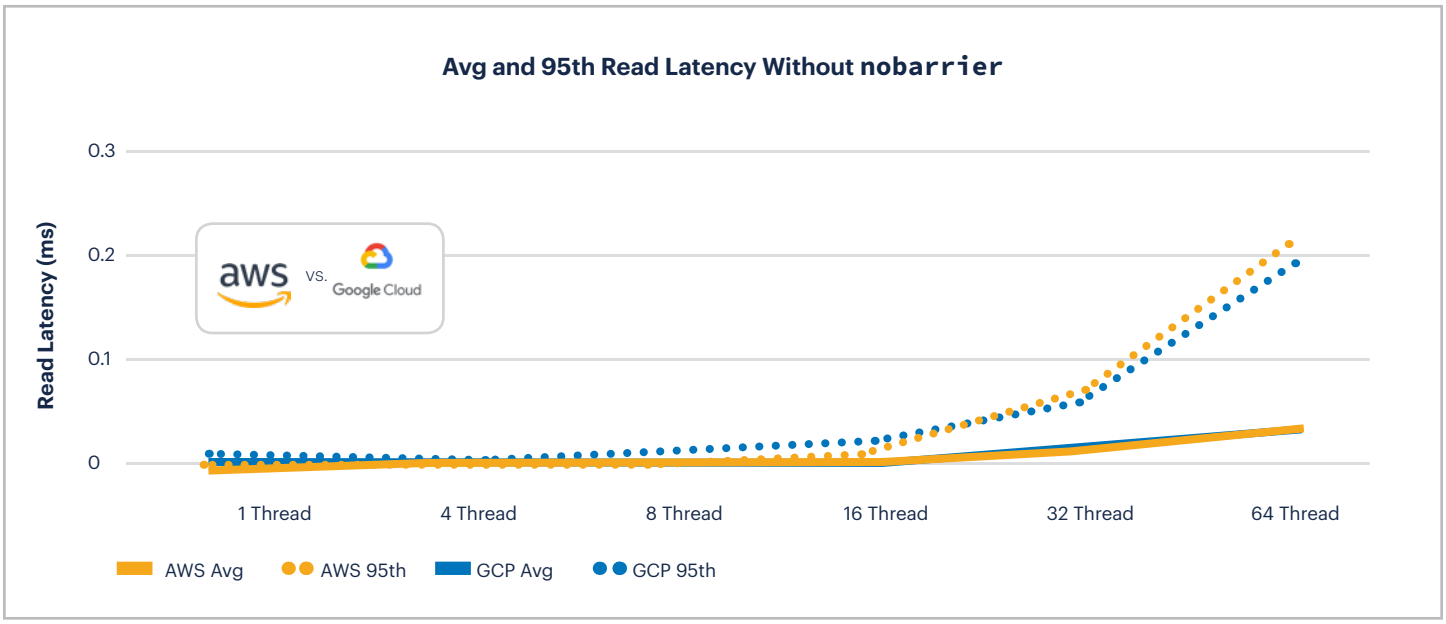
I/O Read Performance: With `nobarrier` & Without `nobarrier`





AWS provides more read throughput from 1 to 8 threads with and without `nobarrier`. `nobarrier` begins to make a difference at 16 threads tipping the advantage to GCP and marginally increases the advantage GCP has at higher thread count.





Similarly to read throughput, AWS wins the read latency battle up to 16 threads. At 32 and 64 threads GCP and AWS split the results with and without `nobarrier`.

Overall, AWS wins for write performance at all threads and read performance up to 16 threads. GCP offers a marginally better performance with similar latency to AWS for read performance at 32 threads and up which can be improved upon by using the `nobarrier` option

Note: We evaluated the I/O performance of storage using locally attached SSD devices. Results for durable network storage devices are likely to be different and should be considered independent of these findings.

Cost

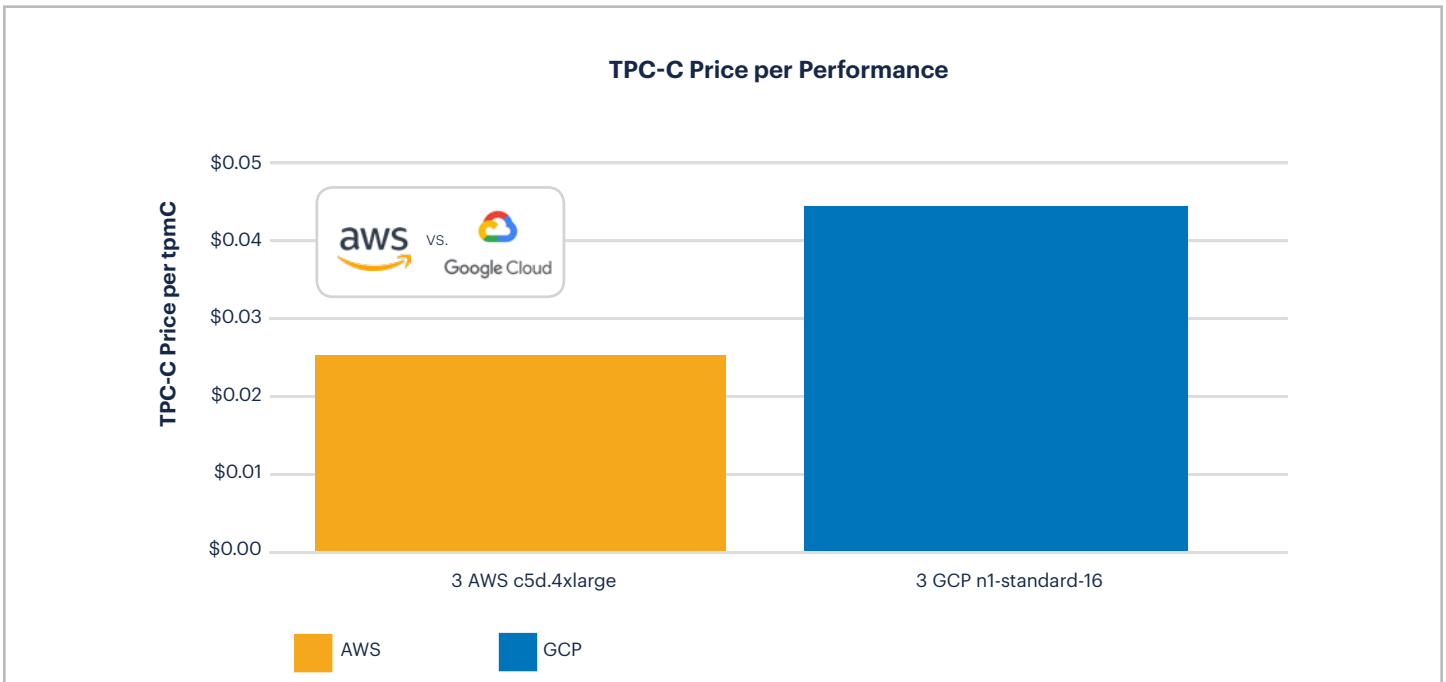
On applied benchmarks (e.g., TPC-C) and the more descriptive micro-benchmarks — CPU, network, and I/O — AWS outperformed GCP. But at what cost? Do you pay for this increased performance on AWS?

Let's circle back to the TPC-C setup discussed at the beginning. For TPC-C, we used `n1-standard-16` on GCP with local SSD and `c5d.4xlarge` on AWS. For both clouds we assumed the most generous discounts available:

On GCP we assumed a three-year committed use price discount with local SSD in the central region.

On AWS we assumed a three-year standard contract paid up front.

Not only is GCP more expensive than AWS, but it also achieves worse performance. This is doubly reflected in the price per performance (below), which shows **GCP costing nearly twice as much as AWS per tpmC** (the primary metric of throughput in TPC-C)!



Machine Name	CPUs	Ram (GB)	TPC-C Throughput (tpmC)	Price per Month	Price per tpmC
3 GCP n1-standard-16	16	60	18,568.90	$839.02	$0.05
3 AWS c5d.4xlarge	16	32	26,079.00	$606.63	$0.02

Conclusion

AWS outperformed GCP on applied performance (e.g., TPC-C) and a variety of micro-benchmarks (e.g, CPU, network, and I/O) as well as cost.

CockroachDB remains committed to our stance as a cloud-agnostic database. We will continue to use GCP, AWS, Microsoft Azure, and others for internal stability and performance testing. We also expect that these results will change over time as all three companies continue to invest in the modern infrastructure ecosystem.

Note, the 2018 Cloud Report focused on evaluating AWS and GCP because they are the most popular cloud platforms among our customers. In future editions, we plan to expand upon our testing with Microsoft Azure, Digital Ocean, and other cloud platforms.

Reproduction Steps

GCP Instance Details

Region: us-east1-b

Image: projects/ubuntu-os-cloud/global/licenses/ubuntu-1604-xenial

AWS Instance Details

Region: us-east-2b

Image: ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-20180126 (ami-965e6bf3)

Set up Roachprod for non-CRL users

1. Build & install Roachprod.

```
# Install Golang: https://golang.org/doc/install


# Install CockroachDB: https://github.com/cockroachdb/cockroach/blob/
master/CONTRIBUTING.md#getting-and-building
mkdir -p $(go env GOPATH)/src/github.com/cockroachdb
cd $(go env GOPATH)/src/github.com/cockroachdb
git clone https://github.com/cockroachdb/cockroach
cd cockroach


# Build Roachprod
make bin/roachprod


# Copy ./bin/roachprod to somewhere in your PATH or extend PATH to
include ./bin
```

2. Configure GCE project and use the Gcloud tool to log in locally.

```
# Install gcloud: https://cloud.google.com/sdk/install
gcloud auth login
export GCE_PROJECT=<project name>
```

3. Configure AWS and use the AWS tool to log in locally.

```
# Install aws cli: https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html
aws configure
```

TPC-C

GCP

1. Set up a cluster name.

```
export CLUSTER=$USER-gcp
```

2. Create roachprod cluster. This creates a 4 node cluster (the default) but the -n flag allows you to change the number of nodes. You can change the machine type with the --gce-machine-type.

```
roachprod create $CLUSTER -n 4 --gce-machine-type=n1-standard-16
```

3. Set no barrier.

```
roachprod run $CLUSTER:1-3 -- 'sudo umount /mnt/data1; sudo mount -o discard,defaults,nobarrier /dev/sdb /mnt/data1/; mount | grep /mnt/data1'
```

4. Then stage CockroachDB.

```
roachprod stage $CLUSTER:1-3 release v2.2.0-alpha.20181217
```

5. Stage workload.

```
roachprod stage $CLUSTER:4 workload
```

6. Start the cluster.

```
roachprod start $CLUSTER:1-3
```

7. Open admin ui.

```
roachprod adminurl --open $CLUSTER:1
```

8. Load fixtures (note what you can load is based on the disk size so make sure it aligns with the machine type chosen. You can use the TPC-C calculator to help).

```
roachprod run $CLUSTER:1 -- "./cockroach workload fixtures import tpcc
--warehouses=2500"
```

9. After the import job is completed (you can verify this in the webui), run tpc-c. Note, you can change the number of warehouses based on the performance (if you fail, move down, if you pass, move up).

```
roachprod run $CLUSTER:4 "./workload run tpcc --ramp=5m
--warehouses=1500 --duration=15m --split --scatter {pgurl:1-3}"
```

10. After you complete your testing, tear down your nodes.

```
roachprod destroy $CLUSTER
```

AWS

SSD

1. Set up a cluster name.

```
export CLUSTER=$USER-aws
```

2. Create roachprod cluster. To change to a specific cluster type, pick the right size and run:

```
roachprod create $CLUSTER -n 4 --clouds=aws  --aws-machine-type-
ssd=c5d.4xlarge
```

3. Set no barrier.

```
roachprod run $CLUSTER:1-3 -- 'sudo umount /mnt/data1; sudo mount -o
discard,defaults,nobarrier /dev/nvme1n1 /mnt/data1/; mount | grep /mnt/
data1'
```

4. Then stage CockroachDB.

```
roachprod stage $CLUSTER:1-3 release v2.2.0-alpha.20181217
```

5. Stage workload.

```
roachprod stage $CLUSTER:4 workload
```

6. Start the cluster.

```
roachprod start $CLUSTER:1-3
```

7. Open admin ui.

```
roachprod adminurl --open $CLUSTER:1
```

8. Load fixtures (note what you can load is based on the disk size so make sure it aligns with the machine type chosen. You can use the TPC-C calculator to help).

```
roachprod run $CLUSTER:1 -- "./cockroach workload fixtures import tpcc
--warehouses=2500"
```

9. After the import job is completed (you can verify this in the webui), run tpc-c. Note, you can change the number of warehouses based on the performance (if you fail, move down, if you pass, move up).

```
roachprod run $CLUSTER:4 "./workload run tpcc --ramp=5m
--warehouses=2100 --duration=15m --split --scatter {pgurl:1-3}"
```

10. After you complete your testing, tear down your nodes.

```
roachprod destroy $CLUSTER
```

EBS gp2

1. Set up a cluster name.

```
export CLUSTER=$USER-aws-gp2
```

2. Create roachprod cluster. The default set up will provide gp2 type EBS:

```
roachprod create $CLUSTER -n 4 --clouds=aws  --aws-machine-
type=c5.4xlarge --local-ssd=false
```

3. Set no barrier.

```
roachprod run $CLUSTER:1-3 -- 'sudo umount /mnt/data1; sudo mount -o
```

```
discard,defaults,nobarrier /dev/nvme1n1; mount | grep /mnt/data1'
```

4. Then stage CockroachDB.

```
roachprod stage $CLUSTER:1-3 release v2.2.0-alpha.20181217
```

5. Stage workload.

```
roachprod stage $CLUSTER:4 workload
```

6. Start the cluster.

```
roachprod start $CLUSTER:1-3
```

7. Open admin ui.

```
roachprod adminurl --open $CLUSTER:1
```

8. Load fixtures (note what you can load is based on the disk size so make sure it aligns with the machine type chosen. You can use the TPC-C calculator to help).

```
roachprod run $CLUSTER:1 -- "./cockroach workload fixtures import tpcc
--warehouses=2500 --db=tpcc"
```

9. After the import job is completed (you can verify this in the webui), run tpc-c. Note, you can change the number of warehouses based on the performance (if you fail, move down, if you pass, move up).

```
roachprod run $CLUSTER:4 "./workload run tpcc --ramp=5m
--warehouses=2100 --duration=15m --split --scatter {pgurl:1-3}"
```

10. After you complete your testing, tear down your nodes.

```
roachprod destroy $CLUSTER
```

EBS io1

1. Set up a cluster name.

```
export CLUSTER=$USER-aws-io1
```

2. Create roachprod cluster. To get an io1 set up we need to pass a few different flags:

```
roachprod create $CLUSTER -n 4 --clouds=aws  --aws-machine-
type=c5.4xlarge --local-ssd=false --aws-ebs-volume-type=io1 --aws-ebs-
iops=20000
```

3. Set no barrier.

```
roachprod run $CLUSTER:1-3 -- 'sudo umount /mnt/data1; sudo mount -o
discard,defaults,nobarrier /dev/nvme1n1; mount | grep /mnt/data1'
```

4. Then stage CockroachDB.

```
roachprod stage $CLUSTER:1-3 release v2.2.0-alpha.20181217
```

5. Stage workload.

```
roachprod stage $CLUSTER:4 workload
```

6. Start the cluster.

```
roachprod start $CLUSTER:1-3
```

7. Open admin ui.

```
roachprod adminurl --open $CLUSTER:1
```

8. Load fixtures (note what you can load is based on the disk size so make sure it aligns with the machine type chosen. You can use the TPC-C calculator to help).

```
roachprod run $CLUSTER:1 -- "./cockroach workload fixtures import tpcc
--warehouses=2500 --db=tpcc"
```

9. After the import job is completed (you can verify this in the webui), run tpc-c. Note, you can change the number of warehouses based on the performance (if you fail, move down, if you pass, move up).

```
roachprod run $CLUSTER:4 "./workload run tpcc --ramp=5m
--warehouses=2100 --duration=15m --split --scatter {pgurl:1-3}"
```

10. After you complete your testing, tear down your nodes.

```
roachprod destroy $CLUSTER
```

CPU

1. Set up a project.

```
export CLUSTER=$USER-perf
```

2. Set up nodes on your desired cloud.

```
# GCP:
roachprod create $CLUSTER --username=$USER --clouds=gce --gce-machine-type=n1-standard-16
# AWS:
roachprod create $CLUSTER --username=$USER --clouds=aws --aws-machine-type-ssd=c5d.4xlarge
```

3. Update apt-get so that you can install stress-ng.

```
roachprod run $CLUSTER -- sudo apt-get update
roachprod run $CLUSTER -- sudo apt-get install stress-ng -y
```

4. Run stress-ng.

```
roachprod run $CLUSTER -- stress-ng --metrics-brief --cpu=16 --timeout=1m
```

Network

iPerf

1. Set up a project (note you don't need new projects and nodes if you completed another of the benchmarks).

```
export CLUSTER=$USER-perf
```

2. Set up nodes on your desired cloud.

```
# GCP:
roachprod create $CLUSTER --username=$USER --nodes=2 --clouds=gce --gce-machine-type=n1-standard-16
```

```
# AWS:
roachprod create $CLUSTER --username=$USER --nodes=2 --clouds=aws
--aws-machine-type-ssd=c5d.4xlarge
```

3. Update apt-get so that you can install iperf.

```
roachprod run $CLUSTER -- sudo apt-get update
roachprod run $CLUSTER -- sudo apt-get install iperf -y
```

4. Determine the internal IP address of VM 1 before setting up a server on it.

```
roachprod ssh $CLUSTER:1 -- ip a
roachprod ssh $CLUSTER:1 -- iperf --server --len=128k
```

5. Set up the client on VM 2. Make sure to point it at the internal IP of VM 1. Terminate the run after the designated run with the last command.

```
roachprod ssh $CLUSTER:2 -- iperf --client=<server_ip> --len=128k
--interval=1 --time=60
```

6. Type `ctrl-c` to end the test.

Ping

1. Set up a project (note you don't need new projects and nodes if you completed another of the benchmarks).

```
export CLUSTER=$USER-perf
```

2. Set up nodes on your desired cloud.

```
# GCP:
roachprod create $CLUSTER --username=$USER --nodes=2 --clouds=gce
--gce-machine-type=n1-standard-16
# AWS:
roachprod create $CLUSTER --username=$USER --nodes=2 --clouds=aws
--aws-machine-type-ssd=c5d.4xlarge
```

3. Determine the internal IP of VM 1.

```
roachprod ssh $CLUSTER:1 -- ip a
```

4. Observe the [internal_ip] for use in the next step.

5. Ping VM 1 from VM 2, using its internal IP address.

```
roachprod ssh $CLUSTER:2 -- ping <internal_ip>
```

6. Type `ctrl-c` to end the test.

I/O

1. Set up a project (note you don't need new projects and nodes if you completed another of the benchmarks).

```
export CLUSTER=$USER-perf
```

2. Set up nodes on your desired cloud.

```
# GCP:
roachprod create $CLUSTER --username=$USER --clouds=gce --gce-machine-type=n1-standard-16
# AWS:
roachprod create $CLUSTER --username=$USER --clouds=aws --aws-machine-type-ssd=c5d.4xlarge
```

3. Install Sysbench repository following https://github.com/akopytov/sysbench#linux

```
roachprod run $CLUSTER -- 'curl -s https://packagecloud.io/install/repositories/akopytov/sysbench/script.deb.sh | sudo bash'
roachprod run $CLUSTER -- sudo apt -y install sysbench make automake libtool pkg-config libaio-dev libmysqlclient-dev libssl-dev libpq-dev
```

4. Mount no barrier.

```
# On GCP:
roachprod run $CLUSTER -- sudo umount /mnt/data1
roachprod run $CLUSTER -- sudo mount -o discard,defaults,nobarrier $(awk '/\/mnt\/data1/ {print $1}' /etc/mtab) /mnt/data1
```

```
# On AWS:
roachprod run $CLUSTER -- 'sudo umount /mnt/data1; sudo mount -o
discard,defaults,nobarrier /dev/nvme1n1 /mnt/data1/; mount | grep /mnt/
data1'
```

5. SSH into node one and cd into mnt/data1

```
roachprod ssh $CLUSTER:1
cd /mnt/data1
```

6. While still in /mnt/data1 set up Sysbench.

```
sysbench fileio --file-total-size=8G --file-num=64 prepare > IO_LOAD_
results
```

7. While still in /mnt/data1, run sysbench on multiple thread sizes for writes. This creates results in IO_WR results file on the node. To obtain those results you can run the second command.

```
for each in 1 4 8 16 32 64; do sysbench fileio --file-total-size=8G
--file-test-mode=rndwr --time=240 --max-requests=0 --file-block-
size=32K --file-num=64 --file-fsync-all --threads=$each run; sleep 10;
done > IO_WR_results
```

8. Exit the node and then run this command to review the results.

```
roachprod get $CLUSTER:1 /mnt/data1/IO_WR_results
```

9. Run sysbench on multiple thread sizes for reads. This creates results in IO_WR results file on the node. To obtain those results, cd into /mnt/data1 and then you can run this for each command.

```
cd /mnt/data1
for each in 1 4 8 16 32 64; do sysbench fileio --file-total-size=8G
--file-test-mode=rndrd --time=240 --max-requests=0 --file-block-
size=32K --file-num=64 --file-fsync-all --threads=$each run; sleep 10;
done > IO_RD_results
```

10. Exit the node and then run this command to review the results.

```
roachprod get $CLUSTER:1 /mnt/data1/IO_RD_results
```

CockroachLabs is the company behind CockroachDB, the ultra-resilient SQL database.

With a mission to Make Data Easy, Cockroach Labs is led by a team of former Google engineers who have had front row seats to nearly two decades of database evolution. The company is headquartered in New York City and is backed by an outstanding group of investors including Benchmark, G/V, Index Ventures, Redpoint, and Sequoia.

Cockroach LABS

Run your business on a database built right.
Learn more at:
www.cockroachlabs.com