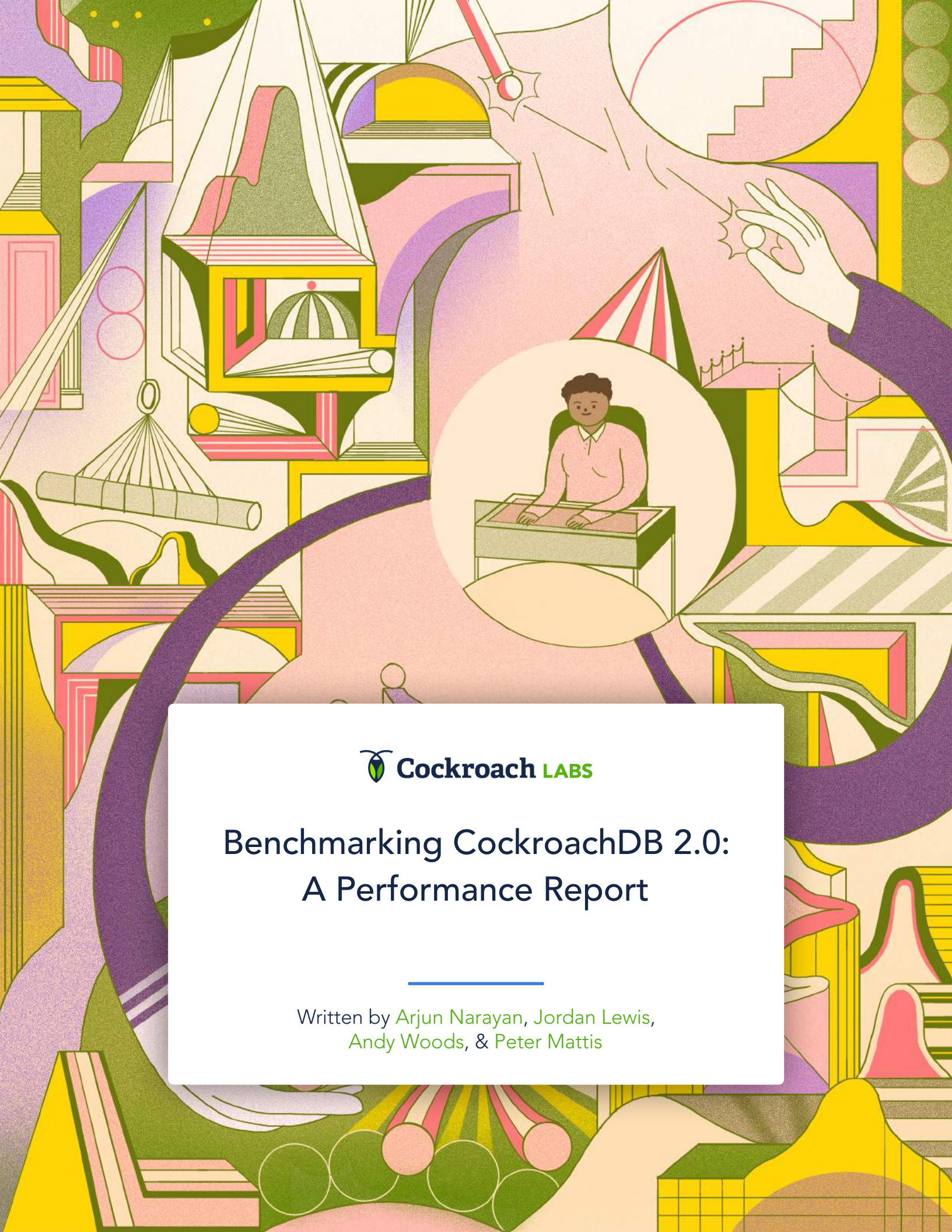

Cockroach LABS

Benchmarking CockroachDB 2.0: A Performance Report

Written by Arjun Narayan, Jordan Lewis, Andy Woods, & Peter Mattis

Introduction

Having achieved our correctness and stability goals for over a year with CockroachDB 1.0 and 1.1, with CockroachDB 2.0, we have been focusing on performance. This comprehensive whitepaper demonstrates how CockroachDB achieves high OLTP performance of over 128,000 tpmC on a TPC-C dataset over 2 terabytes in size. This OLTP performance is over *10x* more TPC-C throughput than Amazon Aurora, in a 3x replicated deployment with a zero recovery point objective (RPO), single-digit seconds recovery time objective (RTO), and zero-downtime migrations and upgrades. This far surpasses a typical active-passive database deployment with manual failure recovery and migration. Finally, CockroachDB achieves this in *serializable* isolation, unlike competing databases that sacrifice isolation for performance.

In keeping with our open source philosophy at Cockroach Labs, in addition to this whitepaper we are also open-sourcing the full experimental playbook for all our experiments, along with the load generators, cloud provisioning scripts, and configurations that we used. We are now ready to demonstrate our first performance milestone: ***CockroachDB 2.0 is the most scalable OLTP cloud-native database that adheres to the strongest correctness and stability guarantees.***

A History of OLTP Benchmarking

Benchmarking is a long-established field with existing practices and norms. Without careful adherence to vendor-neutral standardized benchmarks, it is impossible to make fair comparisons across databases. Furthermore, not every database vendor reports complete results with hardware specifications, tuning parameters, and any other tradeoffs. For instance, while databases often report high throughput numbers, benchmarks don't always come with standardized latency metrics, nor is it standard to report on long running loads. We at Cockroach Labs have used our own custom load generators and queries while developing CockroachDB, but we have done so completely in the open. We also believe that when discussing performance publicly and making comparisons to other databases, we should stick to vendor-neutral benchmarks.

With this in mind, we've thought carefully about what benchmarks to use for our initial performance analysis of CockroachDB and settled on following the TPC-C. The TPC benchmarks are far more rigorous in their specification than other benchmarks like YCSB or sysbench, and thus far less prone to being gamed. Second, each benchmark is opinionated in what it intends to test. TPC-C tests transaction processing over large sizes of data at high throughput and under moderate contention. This is the scenario for which CockroachDB 2.0 is most performant today.

TPC-C overview

We've decided to start with a TPC-C-like benchmark. The TPC-C is a specification for a set of load that simulates the backend data processing requirements of a large retailer. Unlike NoSQL benchmarks like YCSB, which involve a single logical table of key-values, no transactions, and only two queries, TPC-C exercises a richer set of SQL semantics, with foreign keys, multiple indexes, transaction rollbacks, and joins. It also stresses the underlying storage capabilities of the database.

DISCLAIMER: this benchmark script was not validated and certified by the Transaction Processing Council. The results obtained are not official TPC-C results, and the results are not comparable with any official TPC-C results. Instead, we only compare our results to other unofficial TPC-C-like results by competing vendors such as Amazon Aurora. We have also made one modification to the TPC-C specification: we run every transaction in serializable mode, as opposed to degrading the isolation guarantees selectively. As we show in this benchmark, one can achieve high performance without compromising safety, maintainability, and correctness.

Data layout

There are nine tables in TPC-C, modeling the tables one might create to manage inventory for an online store. Example tables include ones to track SKUs, inventory, customers, and orders.

In order to maintain a fixed level of contention, but permit scaling the overall load, the tables and transaction volume scale based on a single parameter: the number of warehouses. A warehouse is a column in the stock table denoting that a given piece of inventory is located in a particular physical warehouse. But the TPC-C specification uses this variable as its unit of scaling, requiring that all other variables (such as the number of customers, the number of items in stock, etc.) also scale linearly with the number of warehouses. Finally, the spec imposes a maximum per-warehouse transaction rate. The philosophy behind this is to make TPC-C performance comparable along a single dimension, while keeping the amount of contention constant at all scales. This is one point along a spectrum of possibilities, but one that has been validated by decades of relevance to many consumers in real-world scenarios.

Query access pattern

TPC-C has five transactions in total, which perform business functions that you would expect from a retailer. These include a customer placing a new order, a backend logistics supplier querying stock levels for items that are running low, getting payment for an existing order, delivering an order, and an impatient customer querying the status of an existing order.

The five transactions are chosen from a fixed probability distribution by "workers" and one transaction - the new order transaction - is used as the benchmark for measuring throughput. Each transaction has some chance of requiring a rollback, to ensure that databases support proper atomicity.

Measuring Performance

The goal of the specification is to provide a level playing field, where all databases have a single variable that they can tune (the warehouse count) and then report a single value. This is their aggregate New Order transaction throughput rate, a measure known as “tpmC”, or “new order transactions fulfilled per minute”. The tpmC is the sole output of the benchmarking process.

Preventing Overscaling

All variables are pegged to linear multiples of the warehouse count, including a maximum rate limit on the tpmC. The philosophy behind the rate limit is to prevent “overscaling”. Turning off the rate limiter would allow a database vendor to process millions of transactions over a few megabytes of stored data, which is not realistic or meaningful to customers. Similarly, scaling one of the tables without scaling the others would allow the processing of transactions over a small subset of the dataset while the vast majority of the stored data was cold. This might allow a vendor to say their database can process transactions over “64 terabytes of stored data” while only really touching a small fraction. For instance, AWS Aurora RDS claims to scale to 64 terabytes of stored data, but cannot sustain maximum tpmC on 10,000 warehouses. 10,000 warehouses translates to approximately 800 gigabytes of stored data, two orders of magnitude below the claimed maximum storage.

Ensuring Opaque Sharding

TPC-C requires that 10% of the orders require fulfillment from a randomly picked warehouse. This remote fulfillment ensures that any internal sharding scheme that a database might use to scale out supports cross-shard transactions, as perfect horizontal sharding is an unrealistic expectation. A clustered database should be able to process transactional throughput without requiring users to take on the burden of figuring out which data

lives on which server and also occasionally process atomic, consistent transactions that span multiple servers.

CockroachDB TPC-C Benchmarks

There are two benchmarks that we run, the first with 1,000 warehouses (for a total dataset size of 200GB) on 3 nodes, and the second with 10,000 warehouses (for a total dataset size of 2TB) on 30 nodes. These two points on the spectrum show how CockroachDB scales from modest sized production workloads to larger scale deployments.

Benchmarking CockroachDB on TPC-C

Small clusters

We first set up a 3 node CockroachDB cluster on Google Compute Engine. Three nodes is the minimum for proper fault tolerance in CockroachDB, and a good initial setup. We used 3 `n1-highcpu-16` VMs, and loaded it with 1,000 warehouses of data. This translates to 80GB of data replicated 3 ways, which, after compression, results in 200GB of data stored across the cluster.

Large Clusters

We then set up a 30 node CockroachDB cluster on Google Compute Engine, using the same `n1-highcpu-16` VMs, loaded 10,000 warehouses of data (a 2TB replicated dataset).

We compared our unofficial TPC-C results to Amazon Aurora RDS unofficial TPC-C results from AWS re:Invent 2017. We also used Aurora's SIGMOD 2017 paper for additional information as to their test setup and load generator.

AWS Comparison Summary of Large Clusters

Database	CockroachDB 2.0	Amazon Aurora RDS MySQL
Throughput (1,000 warehouses)	12,475 tpmC	12,582 tpm
Median latency (1,000)	88.1ms	Not reported
90th percentile latency (1,000)	151.0ms	Not reported
Hardware (1,000)	3x Google Compute Engine n1-highcpu-16 with attached Local SSDs	2x r3.8XL (One read replica with automatic failover)

Database	CockroachDB 2.0	Amazon Aurora RDS MySQL
Throughput (10,000 warehouses)	128,587 tpmC	9,406 tpmC
Median latency (10,000)	88ms	Not reported
90th percentile latency (10,000)	176ms	Not reported

Database	CockroachDB 2.0	Amazon Aurora RDS MySQL
Availability	Can tolerate failure of any node.	Can tolerate failure of a master and automatically promote read replicas to master.
RPO	Zero (data is replicated 3x across the cluster)	Zero (underlying elastic block storage is 6x replicated)
Default Transaction Isolation Level	Serializability	Repeatable read

This is merely a summary of their reported numbers. Benchmarks should be run by vendors or independent consultants, not competitors. The numbers above are the numbers that AWS reports.

Detailed Reproduction

We don't want you to simply take our word on our performance. We want you to try it out yourself! To help you benchmark CockroachDB, here are the complete steps to reproduce *our* results.

Prerequisites

You will need access to bare virtual machines on a public cloud. Our instructions use `roachprod`, our orchestration tool for quickly spinning up virtual machines on Google Compute Engine and running CockroachDB. However, you should be able to adapt these instructions to any cloud or on-prem cluster.

Picking a hardware configuration, and spinning up VMs

For our TPC-C benchmarking, we use `n1-highcpu-16` machines. Currently, we believe this (or higher vCPU count machines) is the best configuration for CockroachDB under high traffic scenarios. `n1-standard-4` or `n1-highcpu-8` are fine for smaller loads, but our intent here is to stress the *performance* and that requires the higher 16 vCPU count VMs. We also attach a single local SSD to each virtual machine. Local SSDs are low latency disks attached to each VM, as opposed to networked block storage, which maximizes performance.

We chose this configuration because it best resembles what a bare metal deployment would look like, with machines directly connected to one physical disk each (as opposed to network-attached replicated block storage). Do note that if you are following this deployment for production in the cloud, you should ensure that you spread your nodes across at least three availability zones and set the CockroachDB zone configuration to spread replicas across zones. **Local SSDs on Cloud VMs can lose data if the VM is lost**, so it is important that CockroachDB can automatically recover from the loss of any single zone's disks. `roachprod` does not spread VMs across availability zones, but your production deployment should.

To spin up machines, download roachprod from the GitHub repository. You will also need to set up the Google Cloud command line tool, gcloud. Set up a Google Cloud project, and set your `GCE_PROJECT` environment variable to your chosen project.

Create a 4 node cluster (3 for the database, 1 for the load generator):

```
$ export CLUSTERNAME="${USER}-tpcc"
$ roachprod create $CLUSTERNAME --gce-machine-type "n1-highcpu-16" --local-ssd --nodes 4
```

Download the latest version of CockroachDB on all nodes:

```
$ roachprod run $CLUSTERNAME 'wget https://binaries.cockroachdb.com/cockroach-v2.0.0.linux-amd64.tgz'
$ roachprod run $CLUSTERNAME 'tar xf cockroach-v2.0.0.linux-amd64.tgz'
$ roachprod run $CLUSTERNAME 'mv cockroach-v2.0.0.linux-amd64/cockroach cockroach'
```

Set up the cluster for production use

At this point, your VMs need to be modified for optimal production performance. We would like to re-emphasize that these settings have been chosen to mimic an enterprise bare-metal deployment as much as possible and are *not safe* unless you are running on battery-backed enterprise SSDs that guarantee write durability even under the event of power loss. If you are using this deployment on a public cloud, you should configure your machines to span three availability zones, and set the CockroachDB zone configuration to spread replicas across all zones.

Start your CockroachDB cluster

At this point you should be able to log into your cluster. **roachprod** makes this easy, with the command:

```
$ roachprod run $CLUSTERNAME -- 'sudo umount /mnt/data1; sudo
mount -o discard,defaults,nobarrier /dev/disk/by-id/google-
local-ssd-0 /mnt/data1/; mount | grep /mnt/data1'
```

If your cluster is running smoothly, you should see a CockroachDB SQL prompt.

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

Benchmarking is greatly sped up by using the enterprise RESTORE feature, rather than creating the fixtures manually. You can obtain an enterprise evaluation 30-day license and do an enterprise restore of the TPC-C dataset. Configure the cluster with your credentials with the following SQL:

```
$ SET CLUSTER SETTING cluster.organization = '<organization>';
$ SET CLUSTER SETTING enterprise.license = '<secret>';
```

You can now CTRL+D out of the SQL prompt.

Restoring the TPC-C dataset

Download the most recent version of the CockroachDB **workload** tool on the load generator node:

```
$ roachprod run $CLUSTERNAME:4 "wget https://edge-
binaries.cockroachdb.com/cockroach/workload.LATEST && chmod a+x
workload.LATEST"
```

Tell workload to load the dataset into your roachprod cluster:

```
$ roachprod run $CLUSTERNAME:4 "./workload.LATEST fixtures load
tpcc {pgurl:1} --warehouses=1000"
```

This will take approximately an hour. You can follow along the progress on the Admin UI's JOBS table. Get the URL with:

```
$ roachprod adminurl $CLUSTERNAME:1
```

If your terminal supports it, you can append the `--open` flag to automatically open the link in your browser.

Click on the "Jobs" link on the sidebar on the left. TPC-C has 9 tables, and you can see estimates for each table's completion time to fully `RESTORE`.

Run the Benchmark

Then run the load generator for 5 minutes:

```
$ roachprod run $CLUSTERNAME:4 "./workload.LATEST run tpcc --
ramp=30s --warehouses=1000 --duration=300s --split --scatter
{pgurl:1-3}"
```

The above command runs the benchmark using all three CockroachDB nodes as gateways. In production, you can hide this behind a load balancer. Do note that if you only use a subset (or one) of the machines, your data will still be replicated across the cluster and remain durable in the event of the loss of a machine You will not, however, get optimal performance, as all queries will go through the specified subset of the machines.

Interpreting the results

At the end of your run, you should see a final output line that looks like this:

```
$ _elapsed_______tpmC____efc__avg(ms)__p50(ms)__p90(ms)__p95(ms)
__p99(ms)_pMax(ms)
$ 600.1s    12475.2  97.0%     95.8     88.1    151.0    176.2
251.7    704.6
```

You will also see some audit checks and latency statistics for each individual query. For this run, some of those checks might indicate that they were `SKIPPED` due to insufficient data. For a more comprehensive test, run it with a longer duration(e.g. two hours). The `tpmC` number is the headline number and `efc`, or "efficiency", tells you how close we get to the theoretical maximum `tpmC`.

The TPC-C specification has p90 latency requirements on the order of seconds, but as you see here, CockroachDB far surpasses that requirement with p90 latencies in the hundreds of milliseconds.

Reproducing TPC-C-10k on 30 nodes.

The methodology for reproducing our 30-node, 10,000 warehouse TPCC result is very similar to that for the 3-node, 1,000 warehouse test case. The only difference (besides the larger node count and dataset) is that we will use CockroachDB's partitioning feature to make sure that replicas for any given slice of the data are usually located on the same nodes that will be queried by the load generator for that slice of data. Partitioning is an enterprise feature, so to perform this test you will need to have an enterprise license. If you don't currently have one, you can also request a free, 30 day trial on our website.

Test setup

Create a 31 node cluster (30 for the database, 1 for the load generator), download CockroachDB onto each node, and configure the SSDs. These are roughly the same steps as for the 3-node test, but are condensed for readability.

Note that when you start the cluster, you'll need to specify an argument, `--racks 10`, that informs `roachprod` to start each node with a locality that includes an artificial "rack number". We choose 10 racks for 30 nodes, so that every 10th node is part of the same rack. We'll use this later to partition the database.

```
$ export CLUSTERNAME="${USER}-tpcc"
$ roachprod create $CLUSTERNAME --gce-machine-type "n1-highcpu-16" --local-ssd --nodes 31
$ roachprod run $CLUSTERNAME 'wget https://binaries.cockroachdb.com/cockroach-v2.0.0.linux-amd64.tgz'
$ roachprod run $CLUSTERNAME 'tar xf cockroach-v2.0.0.linux-amd64.tgz'
```

```
$ roachprod run $CLUSTERNAME 'mv cockroach-v2.0.0.linux-amd64/cockroach cockroach'
$ roachprod run $CLUSTERNAME -- 'sudo umount /mnt/data1; sudo mount -o discard,defaults,nobarrier /dev/disk/by-id/google-local-ssd-0 /mnt/data1/; mount | grep /mnt/data1'
$ roachprod start $CLUSTERNAME:1-30 --racks 10
$ roachprod sql $CLUSTERNAME:1 -- -e "SET CLUSTER SETTING cluster.organization='<organization>'; SET CLUSTER SETTING enterprise.license='<secret>';"
$ roachprod ssh $CLUSTERNAME:31 "wget https://edge-binaries.cockroachdb.com/cockroach/workload.LATEST && chmod a+x workload.LATEST"
$ roachprod run $CLUSTERNAME:31 "wget https://edge-binaries.cockroachdb.com/cockroach/workload.LATEST && chmod a+x workload.LATEST"
$ roachprod run $CLUSTERNAME:31 "./workload.LATEST fixtures load tpcc {pgurl:1} --warehouses=10000"
```

Next, we'll need to partition the database. This uses CockroachDB's partitioning feature to split all of the TPCC tables and indexes into 10 partitions, one per rack, and then uses zone configs to pin those partitions to a particular rack. We will set a cluster setting before we start the partitioning to increase the snapshot rate, which helps speed up this large-scale data movement.

```
$ roachprod sql $CLUSTERNAME:1 -- -e "set cluster setting kv.snapshot_rebalance.max_rate='64MiB';"
$ roachprod ssh $CLUSTERNAME:31 "./workload.LATEST run tpcc --partitions=10 --split --scatter --warehouses=10000 -duration=1s {pgurl:1-30}"
```

Partitioning all of this data takes quite a while - at least 12 hours. It's slow because all of the data (over 2 TB replicated for TPCC-10k) needs to be shuffled around to the right locations. To watch the progress, open the Admin UI (`roachprod adminurl --open $CLUSTERNAME:1`) and look at the Replication Queue in the Queues section of the Metrics page. Once that queue goes to 0 and stays there, the cluster should be finished rebalancing and ready for actual testing.

If you're curious about the actual commands that the `--partitions` flag of `workload` runs, you can take a look at the table and index schemas after running that command. You'll see that they now contain several `PARTITION` clauses that refer to the named racks that we created earlier.

Test run

Now, we're ready to run the load generator.

```
$ roachprod ssh $CLUSTERNAME:31 "ulimit -n 10000 && ./
workload.LATEST run tpcc --warehouses=10000 --duration=300s
{pgurl:1-30}"
```

This command should look familiar to the one we ran in the 3-node case: it runs `tpcc` against all 30 nodes, with 10000 warehouses, for 5 minutes.

At the end of the run, you'll see output similar to the output in the 3-node case, but the `tpmC` should be about 10x higher, reflecting the increase in the number of warehouses.

A Note on Database Isolation Guarantees

The TPC-C spec allows some queries to run in degraded isolation modes. The query-set itself also does not stress serializability: it produces serializable histories even if only run in snapshot isolation mode.

We believe this is too lenient and our benchmarking runs every transaction in serializable mode. While TPC-C has some isolation tests defined, they are rudimentary tests that consist of starting a small set of predefined concurrent transactions and aborting them to seeing if the other transaction observed any anomalous reads.

We hold this viewpoint because it is a difficult task to manually analyze your queries and decide what isolation level each query

must run at: this requires understanding the pairwise interactions between all queries, and combinatorially explodes. Prototyping a new set of queries requires buy-in from all existing stakeholders that already run queries on your database, greatly slowing down developer productivity.

This is why we believe that **all queries should run in serializable mode**, always. Simple optimizations that appear to speed up an application can dramatically change the isolation properties of queries, and this greatly slows down developer productivity, if every change must reason about the entire set of queries run against the database.

We believe that this is a Faustian bargain. Your database should simply never produce incorrect results, under any conditions. As we've shown today, this is not a tradeoff you have to make in order to achieve high performance at scale --- **CockroachDB gives you both correctness and performance, without sacrificing one or the other**.

What about Official Audits?

This whitepaper and implementation is not an official TPC-C benchmark, and cannot be compared to official TPC-C results. Instead, we compare only to Amazon Aurora's unofficial benchmark. In keeping with our open source philosophy at Cockroach Labs, we are also open sourcing everything. We'd like you to replicate our results and examine our code, instead of simply trusting our performance claims.

Other Benchmarks

What about TPC-E?

TPC-E is a newer OLTP benchmark from the Transaction Processing Council. While TPC-E is newer, it does not obsolete TPC-C. The two are more complementary, stressing different facets of databases. TPC-E is different in that it stresses an order of magnitude more reads than writes, as opposed to TPC-C's 2:1 ratio. TPC-E also stresses more random IO than TPC-C. It also has a wider set of SQL features, adding check constraints and referential integrity checks. Finally, it uses a different style of schema, with 33 tables that have an average of 5 columns each, compared to TPC-C's 9 tables with an average of 10 columns each. Unfortunately, TPC-E has not seen much benchmarking.

Only Microsoft SQL Server has ever published official TPC-E results.

TPC-H and TPC-DS?
TPC-H and TPC-DS are analytics benchmarks, and we are focused on OLTP workloads first. While we can process the occasional analytics query with our distributed SQL processing architecture, we are not yet performant on those workloads when compared to dedicated data warehouses like Vertica. However, improved analytics performance is the focus of our current work, as we do seek to make CockroachDB a Hybrid Transactional and Analytics (HTAP) database in the long term.

Performance Tuning Disclosure

In order to achieve high performance, our load generator demonstrates some CockroachDB performance best practices. The source code for the TPCC load generator is the best reference for seeing the details, but we want to bring up some highlights:

Spreading load across the cluster:
Our load generator shards database connection pools by warehouse. CPU and disk latencies showed that the gateway node was overloaded while the other nodes in the cluster had spare capacity. Sending requests to all of the nodes dramatically reduced the disparity.

Manually splitting tables and scattering ranges. While the TPC-C-1000 data size is large enough to require thousands of ranges, 3 tables are small enough to fit in a single range: district, warehouse and items. These 3 tables were acting as bottlenecks. The `-split` flag forces the district and warehouse tables to be split every 100 warehouses and the items table to be split every 1000 items. We also noticed that at startup clusters had a leaseholder imbalance that took a significant time to alleviate. The `-scatter` flag forces leaseholders to be scattered across the cluster and removed odd behavior where a newly restarted cluster would show bad behavior. We intend to change leaseholder rebalancing to be more aggressive in future

releases, and add load-based splitting, so that this manual splitting and scattering is not necessary.

Batching statements in each transaction
Our SQL statements in the New Order transaction batch writes into a single statement, which greatly improves performance versus unbatched serial writes. Our original naive implementation performed this work by executing a series of SQL statements (1 select, 1 update and 1 insert) for each item.

Manually specifying a lookup join in Stock Level
We use a planning hint to force the use of lookup joins for the one use of JOIN in TPC-C. While the "stock level" transaction is rare (4% of transactions), the `join` of the "order_line" and "stock" tables is very expensive for a hash or merge join as there isn't sufficient filtering on the "stock" table. The lookup join reduces reading all of the stock rows for a particular warehouse, to reading ~20 rows in the "stock" table. In a future release, our planning should automatically detect when this is the fastest plan and not require explicit hinting.

Frequently Asked Questions

What about cost?
Our machine specifications allow for straightforward hardware cost calculations. At current prices on Google Cloud Platform Compute Engine, the 3-year discounted price for one `n1-highcpu-16` machine is 0.25506/hr. One Local SSD costs $30/month.

How did you reproduce performance numbers for Amazon Aurora?
All Aurora performance numbers are drawn from Aurora's re:Invent 2017 talk. All inferences as to their setup are from their published SIGMOD 2017 paper. We believe that vendors should not benchmark *each other* as it is easy to misrepresent a database's performance by misconfiguring it. We still have questions about their test setup. For instance, how many provisioned IOPS are required for their setup (since EBS pricing scales linearly with provisioned IOPS)?

What are the specific queries in TPC-C?

The TPC-C specification contains complete details on the schema and queries. Our load generator is also open-source, as is the Percona TPC-C load generator that Amazon Aurora used.

What about Join performance?

One of the TPC-C queries contain a single join. CockroachDB 2.0 supports three JOIN strategies (hash, merge, and lookup). OLTP Joins are currently performant.

What about durability within a single zone?

In the setup described here, data durability is reliant on losing at most one machine, which requires configuring the three machines in separate availability zones (which we do not do). Data written to Local SSD is not guaranteed to be persisted once the accompanying VM is lost as per Google's SLA. We chose this setup because it is most similar to what an on-premises deployment would look like. In an on-premises deployment, your SSDs would not be lost if your machine failed.

Where can I obtain an enterprise license?

You can obtain a self-service 30-day evaluation license here.

What about other benchmarks like Sysbench or YCSB?

We're starting off our performance benchmarking with a focus on OLTP workloads, and determined that TPC-C was the most insightful benchmark to begin with. We intend to publish more benchmarks in other scenarios going forward as well.

What about TPC-C performance without the rate limiter?

Removing the rate limiter allows vendors to "overscale" their database by performing lots of transactions on very small amounts of data. The rate limiter forces the database to also scale its underlying data storage.

You claim CockroachDB is 10x more throughput than Aurora. How do you get that 10x calculation?

Amazon Aurora's highest reported throughput is 12,582 tpmC. This occurs in their 1,000 warehouse configuration (their 10,000 warehouse configuration achieves *lower* throughput). CockroachDB's highest reported throughput is 128,587 tpmC, on the 10,000 configuration. This is 10x the throughput.

What about performance under Jepsen testing?

The gold standard for testing isolation guarantees is Jepsen. We have been Jepsen testing CockroachDB since before 1.0, and continue to run Jepsen regression tests. However, when Kyle Kingsbury published his Jepsen test of CockroachDB, there was some confusion about performance. For some tests, performance was slow enough to make Jepsen testing difficult.

A particular note from Kyle attracted some attention: one test had only 20 queries per second, which made anomaly detection difficult. This got mistranslated into "CockroachDB only achieves 20 QPS", which, as you can see, is not true! The specific tests that Kyle performed involved taking a table, manually splitting it into a hundred ranges, and then performing individual writes while performing full table scans across the table. In order to provide serializable guarantees, CockroachDB's transactional path performs many aborts if reads and writes overlap.

The 20 QPS in this situation isn't necessarily indicative of real-world performance. Under normal operation (i.e. without the intentional splitting) a scan over a hundred ranges would only happen if the scan was over a table between 3.2 and 6.4 gigabytes in size. Now add that the test involved performing that scan while the table was undergoing writes to individual rows in all the ranges: this is a complex situation, and not one in which we expect a lot of read queries per second!

Jepsen testing, as Kyle stressed, does intentionally pathological things to a database (such as splitting a 100 row table into 100 separate ranges, and spreading them across a cluster). This testing is necessary to verify that CockroachDB gives correct, serializable answers under *any* conditions. But this isn't a situation which should occur naturally, and not one where you should expect thousands of queries per second.

Illustration by Dalbert B. Vilarino

Cockroach LABS

CockroachLabs is the company behind CockroachDB, the SQL database for building global cloud services.

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.

Run your business on a database built right. Learn about pricing at:

https://www.cockroachlabs.com/pricing/