Benchmarking YugaByte DB vs. CockroachDB for Internet-Scale Transactional Workloads

A YUGABYTE TECHNICAL PAPER

Karthik Ranganathan
Co-Founder & CTO, YugaByte

January 2019

Executive Summary

Driven by their digital transformation initiatives, enterprises are increasingly building modern applications that generate internet-scale, transactional workloads. Application architects at these enterprises spend a significant amount of time evaluating databases that can power these workloads with ease. Invariably, a key criteria for evaluation is high performance, measured in terms of high throughput (number of operations per second) and low latency (time taken to process each operation).

The benchmark documented in this paper compares the performance of two popular databases, YugaByte DB and CockroachDB, for an internet-scale, transactional workload. The workload consists of various data access patterns including pure inserts, pure queries, mix of updates & queries, consistent secondary indexes as well as distributed transactions. As highlighted in the "Results" section, YugaByte DB outshined CockroachDB for every such access pattern. Averaged across all the patterns in the workload, YugaByte DB had 3.5x higher throughput and 3x lower latency than CockroachDB.

What is an Internet-scale transactional workload?

Today's, internet-scale applications and microservices that power SaaS, eCommerce or financial services verticals are demanding backend data services that can deliver both massive write scalability and transactional guarantees without making any sacrifices on performance. Previously, organizations had to stitch together multiple databases using complex application architectures or yield on either ACID transactions, continuous availability, low latency or high throughput if they went with a single system. Recently, with the rise of transactional NoSQL and distributed SQL systems like YugaByte DB, organizations now have the ability to standardize on a single database that can deliver all the necessary features highly scalable microservices require along with multiple data models, consistent secondary indexes, low latency reads, global data distribution and the ability to transparently handle failures without downtime or bottlenecks.

Our goal with this benchmark is to create a consistent, up-to-date comparison that reflects the latest developments in both YugaByte DB and CockroachDB. Periodically, we will re-run these benchmarks and update this document with our findings. The sample app code used in the benchmarks is available on GitHub . Feel free to open up issues or pull requests on that repository or if you have any questions, comments, or suggestions.

Note that an internet-scale transactional workload is different in a few aspects than another increasingly popular access pattern known as the scale-out RDBMS workload.

The scale-out RDBMS workload values query flexibility higher than scale and performance because the query pattern changes more frequently. These typically require full relational data modeling support, including the use of joins, views, stored procedures and so on. We intend to compare YugaByte DB and CockroachDB for the scale-out RDBMS workload in another round of benchmarks.

About YugaByte DB

YugaByte DB Version Tested: v1.1.0

YugaByte, the company behind YugaByte DB was founded in 2016. Prior to founding YugaByte, the founders developed and scaled the business-critical NoSQL platforms at Facebook. YugaByte DB is an open source, high-performance, multi-model database that brings together transactional NoSQL and globally distributed SQL on a single strongly-consistent document store, built ground-up with inspiration from Google Spanner. Application architects have unparalleled data modeling freedom when it comes to internet-scale transactional workloads as they can choose from transactional key-value, transactional flexible-schema and distributed SQL APIs.

About CockroachDB

CockroachDB Version Tested: v2.1.3

Cockroach Labs, the company behind CockroachDB was founded in 2015. Prior to founding Cockroach Labs, the founders were engineers at Google. CockroachDB is an open-source database that is also inspired by Google's Spanner database. CockroachDB is a distributed SQL database that stores copies of data in multiple locations. CockroachDB can be described as a scalable, consistently-replicated and transactional datastore.

Comparison At-a-Glance

	YugaByte DB	CockroachDB
Description	Distributed SQL & NoSQL database	Distributed SQL database
Website	https://www.yugabyte.com/	https://www.cockroachlabs.com/

GitHub	https://github.com/YugaByte/yugabyte-db	https://github.com/cockroachdb/cockroach
Documentation	https://docs.yugabyte.com/	https://www.cockroachlabs.com/docs/stable/
Initial Release	v1.0, May 2018	v1.0, May 2017
Version Tested	v1.1.0, Sept 2018	v2.1.3 , Dec 2018
License	Open Source, Apache 2	Open Source, Apache 2
Language	C and C++	Go and C++
Operating Systems	Linux, OS X, Kubernetes	Linux, OS X, Windows, Kubernetes
Data Access APIs	PostgreSQL, Cassandra and Redis-compatible	PostgreSQL-compatible
Data Models	Relational, document, key-value, and wide-column	Relational
CAP Theorem	Consistent and Partition-tolerant (CP)	Consistent and Partition-tolerant (CP)
Storage Engine	DocDB (Optimized RocksDB)	RocksDB
Consensus Protocol	Raft	Raft

Architectural Similarities and Differences

Before jumping into the benchmarks, it will help to highlight a few of the key architectural similarities and differences between the two systems. The differences between the two systems will help us understand why some of the performance benchmarks are skewed heavily in favor of YugaByte DB.

Similarities

YugaByte DB and CockroachDB share many similar design traits. For example:

- Both are inspired by Google's Spanner database
- Both make use of the RocksDB storage engine
- Both make use of the Raft consensus protocol
- Both offer PostgreSQL-compatible APIs

- Both are Apache 2.0 open source projects
- Both support multiple clouds and Kubernetes environments
- Both are designed to favor consistency and partition tolerance over availability.

Languages

CockroachDB relies on a combination of Go and C++. Go for its client, parser and server, while employing C++ for its RocksDB storage engine. Like Java, Go has Garbage Collection (GC) overhead and pauses. As a result, the performance of the system will ultimately become impacted during these GC operations. By employing multiple languages in a single system, CockroachDB must traverse an interface switch in languages and memory models inside the critical path for read/write transactions. YugaByte avoids these problems by being written entirely in C and C++.

An Optimized vs "Black Box" RocksDB Storage Engine

Both YugaByte and CockroachDB utilize RocksDB as their underlying storage engine. Unlike YugaByte DB, CockroachDB utilizes a "black box" implementation of RocksDB. In this type of an implementation there are two logs used by the system. One for all transactions and one for the Raft protocol (which provides data consistency and the leader-election of nodes.) This setup forces CockroachDB to perform multiple writes to both logs for the same transaction. YugaByte has forked and customized RocksDB to create the enhanced DocDB document store which addresses this issue along with a lot of others. DocDB unifies the transaction and Raft logs, and in the process eliminates the need for double-writes.

According to Design and Architecture of CockroachDB: "Nodes maintain a separate instance of RocksDB for each disk. Each RocksDB instance hosts any number of ranges. RPCs arriving at a RoachNode are multiplexed based on the disk name to the appropriate RocksDB instance. A single instance per disk is used to avoid contention. If every range maintained its own RocksDB, global management of available cache memory would be impossible and writers for each range would compete for non-contiguous writes to multiple RocksDB logs." In a nutshell, CockroachDB uses the KV (key-value) store of RocksDB, which is a loose integration that causes columns in a row to be mapped as individual key-value pairs, instead of as an entire "document" entity like they are in YugaByte DB.

The negative side effects of CockroachDB's key-value implementation are as follows:

- It cannot handle high data density in the face of node-failure or when additional nodes are added to the cluster.
- Its buffer cache efficiency is compromised by the multiplexing of data from different tables.

Support for Read-Only Replicas

CockroachDB does not support read-only replicas. This means that all of the replicas in a cluster must participate in distributed consensus writes. Conversely, YugaByte has enhanced the Raft consensus protocol to support read-replicas in geographically dispersed regions where the latency related to distributed consensus writes would not be tolerable.

Fast and Efficient Backup Restore Operations

Backup and restore operations in CockroachDB require a logical scan of all the data in a cluster in order to make an image of the current dataset. This type of a design has the following negative consequences for CockroachDB users:

- Performing a full logical scan on a running a cluster puts a lot of stress on it by forcing it to compete with the foreground application for database resources like CPU, disk IOPS and network bandwidth. This can result in increased latencies.
- Backing up a large dataset with a logical scan can take a very long time. This can force users to limit the frequency of backups so as not to impact users or other operations.

YugaByte DB takes a different approach and instead maintains references to compressed read-only file-sets from the tablet leaders. For more details on the speed and efficiency of YugaByte DB's distributed backups, check out: Getting Started with Distributed Backups in YugaByte DB.

APIs Used in the Benchmarks

For the purposes of this paper we are comparing CockroachDB's PostgreSQL-compatible API vs YugaByte DB's Cassandra-compatible YCQL API. Why? Because we believe that our YCQL API is actually a better choice today for internet-scale transactional workloads. As highlighted before, we will be re-running the benchmark with YSQL, YugaByte DB's

PostgreSQL-compatible API, in the future for the scale-out RDBMS workloads with additional access patterns such as JOINs and foreign keys.

Syntactically, YCQL is very similar to SQL. As you'd expect the language has the notion of tables, rows and columns, but it also has some advantages over a generic SQL API. For example:

- **Cluster-aware drivers:** Cassandra Query Language (CQL) offers mature client-drivers in multiple languages that are cluster aware. These drivers were designed to work with an elastic and multi-node database. As an example, the drivers are aware of node additions and removals transparently.
- **Ideal for event data:** The YCQL data model supports clustering columns to allow for the efficient storing and retrieving of event or time series data (*which is always growing!*) This makes it ideal for applications that need to deal with large datasets.
- **Partition aware smart drivers:** The YCQL drivers are partition aware, and will route the data directly to the correct node when using the PREPARE/BIND/EXECUTE execution model which delivers excellent performance vs partition *unaware* drivers.
- **Topology aware routing in a geo-distributed cluster:** The YCQL drivers can perform reads from the local or the nearest DC, as well as transparently route writes to the correct geographic location for globally consistent writes.
- **Richer data types:** Most common data types are supported. Additionally, collection types such as maps, sets and lists are supported as well.
- **Richer built-in database features.** Most of the common features are supported such as role based access control (RBAC), online schema changes, etc. There are other capabilities supported such as the ability to automatically expire data (by setting a TTL at table level or row level).

While the above benefits are gained by being compatible with CQL, YCQL goes well beyond CQL by adding critical transactional and flexible-schema data modeling features that are missing in CQL.

- **Global, strongly consistent secondary indexes:** Increases development agility by ensuring flexible, correct and low-latency queries based on non-primary key columns.
- **Unique constraints on columns:** Allows business logic to be represented more directly in the schema by ensuring that no two rows can have identical values for set of columns.
- **Native JSONB column type:** Enables modeling of unstructured data seamlessly, similar to the JSONB column type in PostgreSQL.

- **Distributed transactions:** Simplifies app development by ensuring that multi-key, multi-table updates across shards can be committed with ACID compliance.

Benchmarking Setup and Specifications

Benchmarks

The benchmarking tests we used were used to simulate the following behaviors:

- **Pure Inserts:** Insert 50 Million unique key-values using the maximum number of threads the database could support running in parallel.
- **Pure Queries:** Query the previously written 50 Million unique key-values using the maximum number of threads the database could support running in parallel.
- **Updates and Queries:** A mixed read and write workload where the previously written values are updated. In this benchmark there were 64 writers and 128 readers.
- **Secondary Indexes - Pure Inserts:** Insert 5 Million unique key-values into a database table with a secondary index enabled on the value column using the maximum possible concurrency.
- **Secondary Indexes - Pure Reads:** Query the previously written 5 Million unique key-values from the database. Each query uses a random value to look up the key, exercising the secondary index feature.
- **Distributed Transactions:** In this benchmark, each transaction updated two random key values using the maximum number of threads the database could support running in parallel.

Hardware

- Three AWS i3.4xlarge type nodes
- Each node had 16 vCPUs, 122 GiB RAM, 3.8TB (2 SSD disks) for storage
- Disks mounted using XFS filesystem

Software

- YugaByte DB v1.1.0
- CockroachDB v2.1.3

- Both clusters had their Replication Factor set to 3
- Both clusters used a multi-availability zone configuration that spanned 3 zones
- The databases were brought up on the exact same machines
- The benchmark driver program was run from a separate machine

YugaByte DB Setup

Install YugaByte DB by following the instructions here.

Example YugaByte DB Benchmarking Script

Source code and build instructions for the yb-sample-apps.jar used below are available on GitHub.

```
java -jar yb-sample-apps.jar --workload CassandraKeyValue --nodes
$YB_NODES --nouuid --num_unique_keys 50000000 --num_writes 50000000
--num_threads_write 356 --num_threads_read 0 2>&1 | tee
~/benchmarks/pure-inserts-yb.log
```

CockroachDB Setup

Install CockroachDB by following the instructions here.

Example CockroachDB Benchmarking Script

Source code and build instructions for the yb-sample-apps.jar used below are available on GitHub.

```
java -jar yb-sample-apps.jar --workload SqlInserts --nodes $CR_NODES
--nouuid --num_unique_keys 50000000 --num_writes 50000000
--num_threads_write 256 --num_threads_read 0 2>&1 | tee
~/benchmarks/pure-inserts-cr.log
```

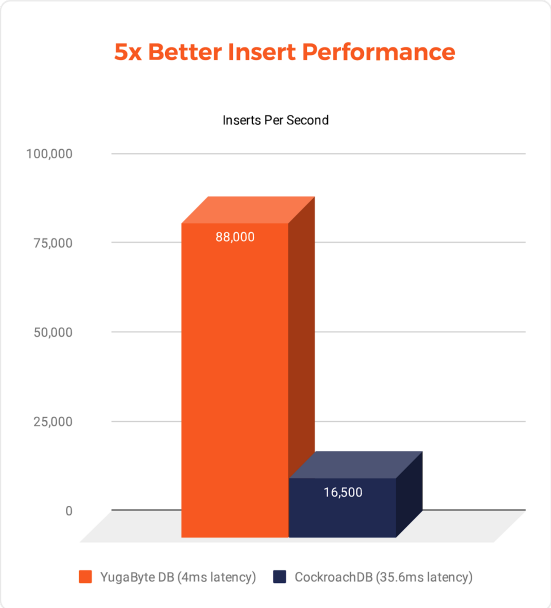
Performance Benchmarking Results

Pure Inserts

Insert 50M unique key-values into the database using insert statements using maximum possible concurrency (256 writer threads running in parallel). There were no reads against the database during this period.

	YugaByte DB	CockroachDB
Create Database	CREATE KEYSPACE IF NOT EXISTS name;	CREATE DATABASE IF NOT EXISTS name;
Create Table	CREATE TABLE table_name (k varchar PRIMARY KEY, v varchar);	CREATE TABLE table_name (k varchar PRIMARY KEY, v varchar);
Insert Row	INSERT INTO table (k, v) VALUES (?, ?);	INSERT INTO table (k, v) VALUES (?, ?);
Keys Written	50 Million	50 Million
Number of Writers	356	256 (Unable to sustain more threads)
Initial Performance	90k writes/sec at 3.94ms	7.3k writes/sec at 35.4ms
Sustained Performance	88k writes/sec at 4ms	7.1K writes/sec at 35.6ms * Sometimes up to 16.5K writes/sec

* Different benchmark runs yielded different max write IOPS, hence the range.



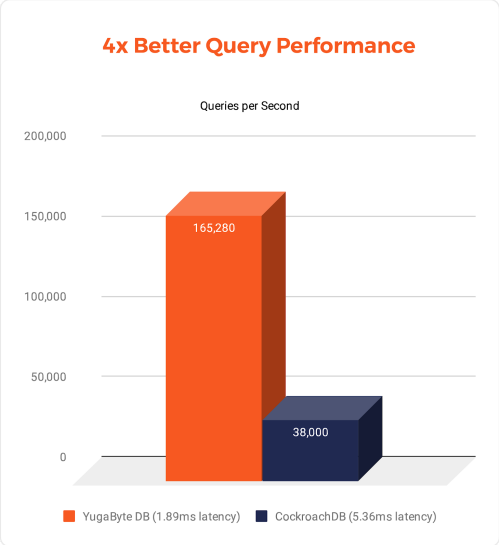
CONCLUSION:

YugaByte DB delivered 5x better insert throughput at 9x lower latency than CockroachDB.

Pure Queries

Query the previously written 50 Million unique key-values from the database. Each query uses a random key to look up the value. There are no writes against the database during this period.

	YugaByte DB	CockroachDB
Create Database	CREATE KEYSPACE IF NOT EXISTS name;	CREATE DATABASE IF NOT EXISTS name;
Create Table	CREATE TABLE table_name (k varchar PRIMARY KEY, v varchar);	CREATE TABLE table_name (k varchar PRIMARY KEY, v varchar);
Select Row	SELECT * FROM table WHERE k=?;	SELECT * FROM table WHERE k=?;
Keys Written	50 Million	50 Million
Readers	312	192 (Latency increases and IOPS decreases with more)
Read ops/sec	165,280	38,000
Latency	1.89ms	5.36ms



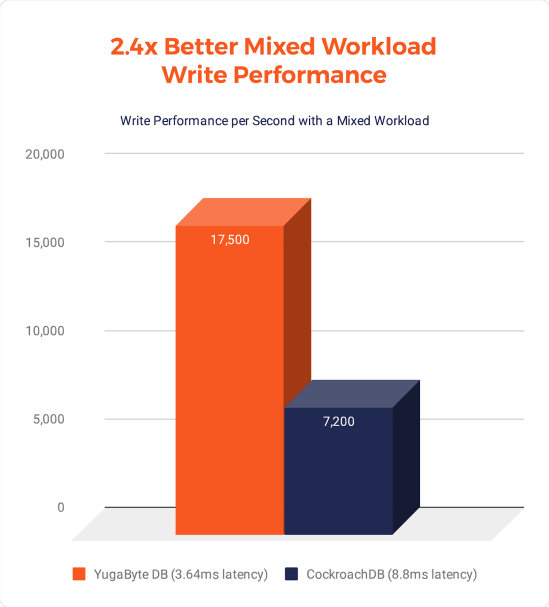
CONCLUSION:

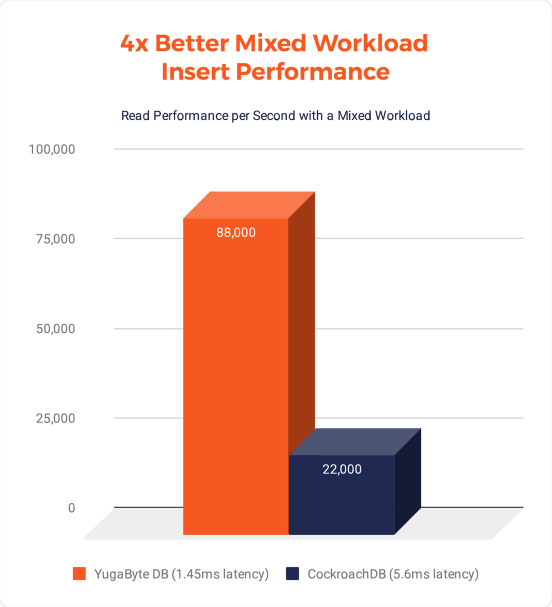
YugaByte DB delivered 4x better query throughput with 1/3 the latency compared to CockroachDB.

Updates and Queries

Mixed read-write workload where we update the values for previously written 50M unique key-values while simultaneously reading these. Each query uses a random key to look up the value.

	YugaByte DB	CockroachDB
Create Database	CREATE KEYSPACE IF NOT EXISTS name;	CREATE DATABASE IF NOT EXISTS name;
Create Table	CREATE TABLE table_name (k varchar PRIMARY KEY, v varchar);	CREATE TABLE table_name (k varchar PRIMARY KEY, v varchar);
Select Row	SELECT * FROM table WHERE k=?;	SELECT * FROM table WHERE k=?;
Update row	UPDATE table SET v = new-value WHERE k = ?;	UPDATE table SET v = new-value WHERE k = ?;
Keys Updated	1 million	1 million
Number of Writers	64	64
Number of Readers	128	128
Write Performance	17.5K ops/sec at 3.64ms	7.2K writes/sec at 8.8ms
Read Performance	88K ops/sec at 1.45ms	22K writes/sec at 5.6ms





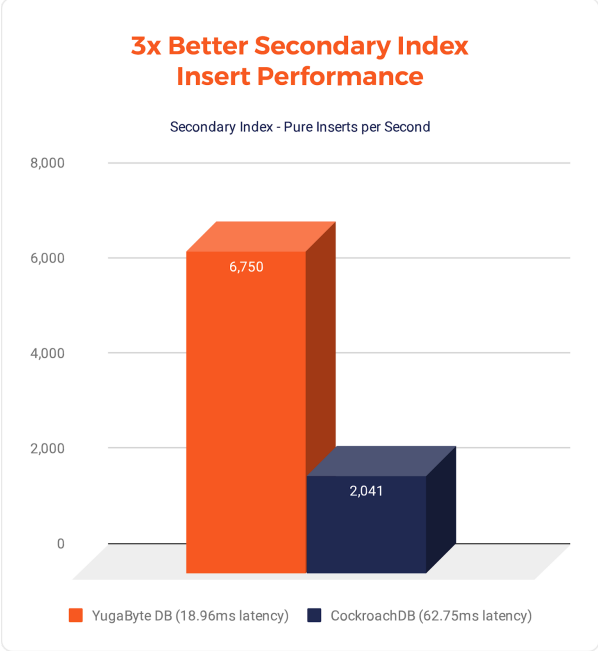
CONCLUSION:

> ***YugaByte DB delivered 2.4x the number of writes and 4x the reads with significantly lower latency than CockroachDB in a mixed workload.***

Secondary Indexes - Pure Inserts

In this benchmark we insert 50 Million unique key-values using the maximum number of threads the database could support running in parallel. Note that CockroachDB was not able to sustain this workload with 128 insert threads. Latency jumped up to 10+ seconds and writes ops/sec dropped to 0.

	YugaByte DB	CockroachDB
Create Database	CREATE KEYSPACE IF NOT EXISTS name;	CREATE DATABASE IF NOT EXISTS name;
Create Table	CREATE TABLE table_name (k varchar PRIMARY KEY, v varchar) WITH transactions = { 'enabled' : true };	CREATE TABLE table_name (k varchar PRIMARY KEY, v varchar);
Create Index	CREATE INDEX index_name ON table_name(v);	CREATE INDEX index_name ON table_name(v);
Insert Row	INSERT INTO table (k, v) VALUES (?, ?);	INSERT INTO table (k, v) VALUES (?, ?);
Keys Written	5 Million	5 Million
Num Writers	128	128
Write ops/sec	6,750	2,041
Latency	18.96ms	62.75ms



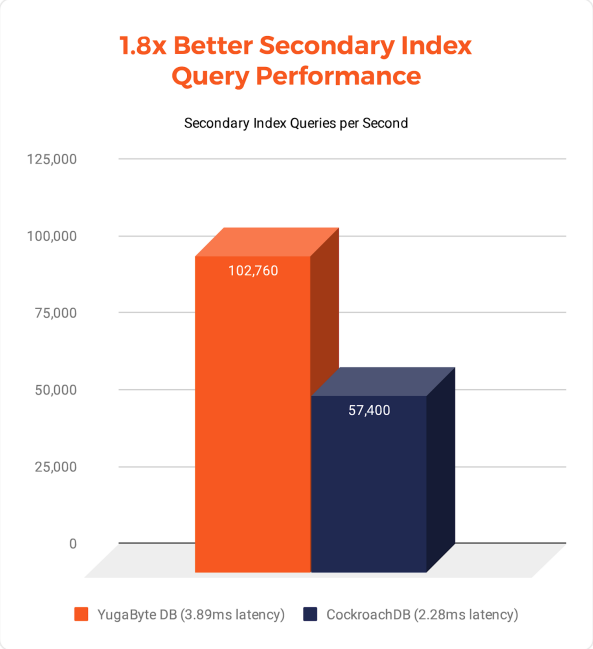
CONCLUSION:

YugaByte DB delivered 3x better write throughput with 3x lower latency than CockroachDB using secondary indexes.

Secondary Indexes - Pure Reads

In this benchmark we query the previously written 5 Million unique key-values from the database. Each query uses a random value to look up the key, exercising the secondary index feature.

	YugaByte DB	CockroachDB
Create Database	CREATE KEYSPACE IF NOT EXISTS name;	CREATE DATABASE IF NOT EXISTS name;
Create Table	CREATE TABLE table_name (k varchar PRIMARY KEY, v varchar) WITH transactions = { 'enabled' : true };	CREATE TABLE table_name (k varchar PRIMARY KEY, v varchar);
Create Index	CREATE INDEX index_name ON table_name(v);	CREATE INDEX index_name ON table_name(v);
Select	SELECT * FROM table WHERE v=?;	SELECT * FROM table WHERE v=?;
Keys Written	5 Million	5 Million
Number of Writers	128	128
Read ops/sec	102,760	57,400
Latency	3.89ms	2.28ms



CONCLUSION:

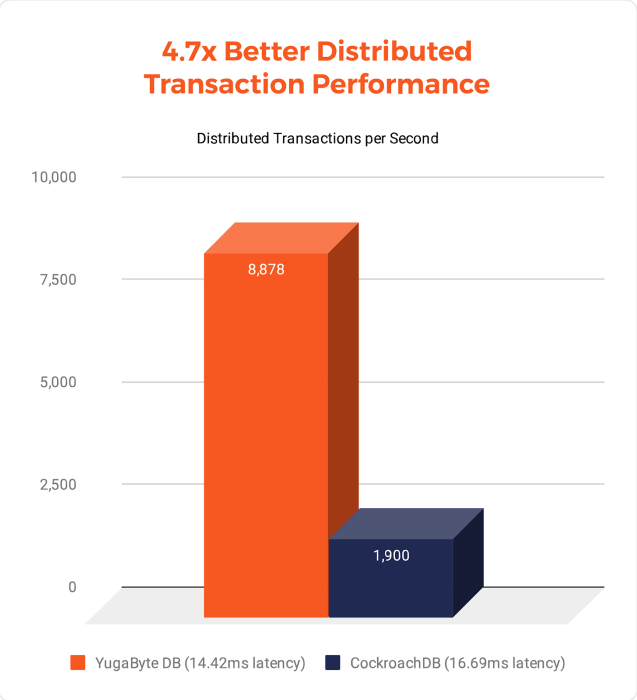
YugaByte DB delivered 1.8x better secondary index query performance than CockroachDB.

Distributed Transactions

In this benchmark, each transaction updated two random key values using the maximum number of threads the database could support running in parallel.

	YugaByte DB	CockroachDB
Create Database	CREATE KEYSPACE IF NOT EXISTS name;	CREATE DATABASE IF NOT EXISTS name;
Create Table	CREATE TABLE table_name (k varchar PRIMARY KEY, v varchar) WITH transactions = { 'enabled' : true };	CREATE TABLE table_name (k varchar PRIMARY KEY, v varchar);
Isolation Level	Snapshot Isolation	Snapshot Isolation
Transaction	BEGIN TRANSACTION INSERT INTO table_name (k, v) VALUES (?, ?); INSERT INTO table_name (k, v) VALUES (?, ?); END TRANSACTION;	BEGIN ISOLATION LEVEL SNAPSHOT; INSERT INTO table_name (k, v) VALUES (?, ?); INSERT INTO table_name (k, v) VALUES (?, ?); COMMIT;

Keys Written	5 Million	5 Million
Number of Writers	32	32
Write ops/sec	8,878	1,900
Write Latency	14.42ms	16.69ms



CONCLUSION:

YugaByte DB delivered 4.7x better throughput for distributed transactions than CockroachDB.

Conclusion

The benchmarks in this paper revealed the following results:

- In a pure insert workload, YugaByte DB delivered **5x better insert throughput at 4x lower latency** than CockroachDB.
- In a pure select workload, YugaByte DB delivered **4x better query throughput at half the latency** than CockroachDB.
- In a mixed workload of both selects and updates, YugaByte DB delivered **3.5x write throughput and almost 10x read throughput with significantly lower latency** than CockroachDB.

- In a pure insert workload with secondary indexes, YugaByte DB delivered **5x the write throughput with 2.5x lower latency** than CockroachDB.
- In a pure select workload by secondary index, YugaByte DB delivered **2.5x better secondary index query performance** than CockroachDB.
- In a distributed transaction workload, YugaByte DB delivered **3.5x better throughput for distributed transactions** than CockroachDB.

These results show that YugaByte DB outperforms CockroachDB for all access patterns important in an internet-scale, transactional workload. **Averaged across all the patterns in the workload, YugaByte DB had 3.5x higher throughput and 3x lower latency than CockroachDB.** Even though there are many architectural similarities between these databases, YugaByte DB delivers higher throughput and lower latency than CockroachDB by virtue of its superior storage engine and replication layer implementation. This implementation includes optimizations based on the learnings of the YugaByte team from building previous generation databases such as Oracle, Apache HBase and Apache Cassandra.

In conclusion, we highly encourage developers and architects to run these benchmarks themselves to independently verify the results on their hardware and data sets of choice. However, for those looking for a valid starting point on which transactional database will give massive scalability, superior throughput and lower latency, YugaByte DB is the clear winner across all these dimensions, especially when number of concurrent readers/writers to the database spike due to unexpected business growth and the system has to process a large volume of data continuously.

What's Next?

YugaByte DB Documentation, Downloads & Guides

- YugaByte DB Documentation
- YugaByte DB Downloads and Quickstart Guide

Have Questions or Need Help?

- YugaByte DB on Stackoverflow
- YugaByte DB on GitHub
- Contact Us