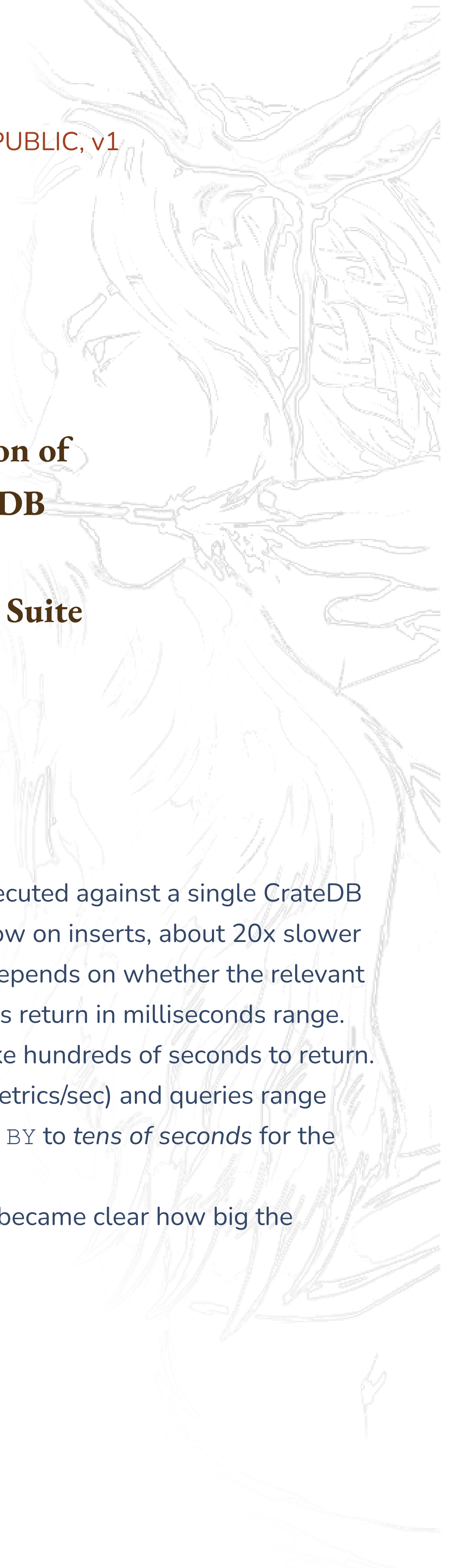
NYRKIÖ

PUBLIC, v1

Technical Report

2024-07-10



Independent comparison of CrateDB and MongoDB using Time Series Benchmark Suite

Executive Summary

TimeSeries Benchmark (devops workload) was executed against a single CrateDB node and a single MongoDB node. MongoDB is slow on inserts, about 20x slower than CrateDB. On reads, MongoDB performance depends on whether the relevant columns are indexed or not. When they are, queries return in milliseconds range. When only the **time** column is indexed, queries take hundreds of seconds to return.
CrateDB is able to insert 180k rows/second (2M metrics/sec) and queries range from *hundreds of milliseconds* for a simple `GROUP BY` to *tens of seconds* for the more complex double `GROUP BY`.
Plans for a scale-out test were abandoned once it became clear how big the difference is already in the single node case.

Author(s): Henrik Ingo, Nyrkiö Oy

Table of Contents

Introduction

The TimeSeries Benchmark Suite (TSBS) was created by TimescaleDB to serve as a tool that can be used to compare performance of different (competing) time series database solutions. In this report we share results from running TSBS against CrateDB and MongoDB.

The benchmarks were setup and executed by Nyrkiö Oy, a company that specializes in software performance tools and consulting. The benchmarks have been commissioned by CrateDB, but Nyrkiö has executed them independently, and the results and conclusions presented in this report are presented as objectively and fairly as possible by Nyrkiö Oy acting as an "independent expert" in the domain of database performance. CrateDB was not given any influence over the results or conclusions. However, having commissioned this work, CrateDB Gmbh was given the right to decide whether or not they choose to publish this report.

The author has 7 years of experience working as a performance expert at MongoDB Inc. Prior to this work the author was not acquainted with CrateDB, but as they had access to CrateDB engineers and expert support, they feel confident that they were able to capture the best results also for this product.

Results are presented for a single database node. A scale out test was originally planned, but given the large differences seen already for the single node case, a scale out test seemed unnecessary.

About MongoDB

MongoDB is the most popular NoSQL database. It is a general purpose database most commonly used for OLTP / transactional workloads. That is, the typical usage is as the database for an application used in real time by users: e-commerce, social media, websites, mobile games and apps... Famously, MongoDB popularized the use of JSON documents as the database record. Hence it falls into the category of document databases.

While MongoDB therefore wasn't purpose built to serve as a time series database, that is of course yet another use case you might consider using a good, general purpose database for. MongoDB's strengths in such a choice would be in its excellent scaling properties: It's relatively easy to add more shards to a MongoDB cluster, if the setup is working well on a smaller scale. The document model also has proven powerful in IoT use cases, where characteristically the incoming metrics may vary over time, maybe even at the same time, and therefore a classical relational schema might be too rigid.

In fact, this use case has been so popular with its users that since version 5.0 MongoDB has provided a special "timeseries collection" type. For this study we tried to use both a regular collection and the purpose built timeseries collections. Results are reported for the latter as they unsurprisingly were faster for this use case, but also more intuitive to use than the old school way of "bucketing" the time series events into large documents.

Even for a time series collection MongoDB still offers the user the choice to index, or not, additional columns in addition to just "**`time`**". This is a two-edged sword: Adding the optimal set of indexes can be powerful in situations where the queries are known in advance. However, an ad hoc query not covered by such additional indexing will likely perform miserably. And in any case, each additional index has a significant cost when inserting data - as we will see.

In this report we chose to present 2 variations of the MongoDB results: the first column indexed only the "time" column alone. The second column additionally defined the index **[hostname, time]**.

About CrateDB

Though less known, CrateDB is also over 10 years old as an open source software project. Development is led by the Austrian namesake Crate.io Gmbh.

CrateDB is written in Java, and is part of the wider Apache Software Foundation Big Data ecosystem. It includes components from ASF projects such as Lucene, Netty, Trino and ElasticSearch. [1] Java 21 is supported, which is worth pointing out in the context of performance, since it enables us to choose one of the new low latency garbage collectors: ZGC or Shenandoah. *[This version of this report does not however include such results.]*

Like MongoDB, CrateDB isn't a purpose built timeseries database, but more general purpose in nature. It provides a SQL-like language, and supports most of what you would expect in a modern database, except for transactions. JSON documents are

[1] CrateDBs strategy wrt architecture is in other words similar to the better known ElasticSearch and Cassandra projects, that similarly aggregated and productized components from the ASF ecosystem. In fact, a popular way to describe CrateDB is "it's like if ElasticSearch was an OLTP database".

supported as the OBJECT column type, a solution not unlike PostgreSQL's JSONB. In fact, CrateDB is also wire-compatible with any PostgreSQL client.

While not built specifically for time series, previous experiments had already shown that CrateDB performs particularly well for inserts and reads, making it an interesting contender for workloads like the TimeSeries Benchmark.

Being general purpose and offering decent OLTP performance, a potential motivation to select either of these databases for a timeseries use case, is that the same database could then be used for other purposes as well. (Either a different installation of the same product, which would simplify the need to train and maintain competence to administer many different specialized databases; or, ultimately, an opportunity to use the very same database cluster both for your OLTP application and analytical needs, thus obviating the need for ELT or Streaming pipelines from one database to the next.)

CrateDB presents a rather novel, and potentially revolutionary approach to indexing. Leveraging efficient indexing algorithms and structures of the Lucene project, it is possible to index every column by default. This means that a timeseries or analytical database in CrateDB allows the user to execute any arbitrary ad hoc queries, using any column for WHERE, GROUP, or ORDER attributes, and expect decent / similar response times every time, regardless of which column is used.

Finally, it is worth highlighting a certain polish and attention to usability that permeates CrateDB. For example it ships with its own JVM bundled, thus freeing the user from having to install and configure a JVM as a pre-requisite. As the results will show, it performs well in this test with default configuration. [2]

About Time Series Benchmark Suite

The TimeSeries Benchmark Suite (TSBS) was originally created by TimescaleDB for the purpose of supporting multiple competing timeseries databases and allowing users to compare their performance with a standard benchmark.

TSBS includes a handful of different workloads, each of which has to be implemented separately for each target database. The most commonly used "devops" database simulates a monitoring system of a data-center. Incoming

[2] This is somewhat unheard of in database benchmarking!

(generated) data looks like metrics from a Linux server, and measurements are about things like CPU or disk utilization. (See table below for specific example.)

The original GitHub repository for TSBS seems to have become unmaintained a few years ago. There's a queue of pull requests that nobody is responding to. For this work we used a more up-to-date repository maintained by BenchANT. This incorporates among other things two patches that are needed to make TSBS work against CrateDB.

For the MongoDB tests we used MongoDB's own fork of TSBS, as this is also used by MongoDB engineers themselves to test nightly builds.

Queries:

load	`INSERT INTO "benchmark"."redis"`
	`(tags,ts,uptime_in_seconds,total_connections_received,expired_keys,evicted_keys,keyspace_hits,keyspace_misses,instantaneous_ops_per_sec,instantaneous_input_kbps,instantaneous_output_kbps,connected_clients,used_memory,used_memory_rss,used_memory_peak,used_memory_lua,rdb_changes_since_last_save,sync_full,sync_partial_ok,sync_partial_err,pubsub_channels,pubsub_patterns,latest_fork_usec,connected_slaves,master_repl_offset,repl_backlog_active,repl_backlog_size,repl_backlog_histlen,mem_fragmentation_ratio,used_cpu_sys,used_cpu_user,used_cpu_sys_children,used_cpu_user_children)`
	`VALUES`
	`(?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)`
	
	`INSERT INTO "benchmark"."disk"`
	`(tags,ts,total,free,used,used_percent,inodes_total,inodes_free,inodes_used)`
	`VALUES`
	`(?,?,?,?,?,?,?,?,?)`
	
	`INSERT INTO "benchmark"."postgresl"`
	`(tags,ts,numbackends,xact_commit,xact_rollback,blks_read,blks_hit,tup_returned,tup_fetched,tup_inserted,tup_updated,tup_deleted,conflicts,temp_files,temp_bytes,deadlocks,blk_read_time,blk_write_time)`
	`VALUES`
	`(?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)`
	
	`INSERT INTO "benchmark"."diskio"`
	`(tags,ts,reads,writes,read_bytes,write_bytes,read_time,write_time,io_time)`
	`VALUES`
	`(?,?,?,?,?,?,?,?,?)`
	
	`INSERT INTO "benchmark"."kernel"`

```
(tags,ts,boot_time,interrupts,context_switches,processes
_forked,disk_pages_in,disk_pages_out)
VALUES
(?,?,?,?,?,?,?,?)

INSERT INTO "benchmark"."mem"
(tags,ts,total,available,used,free,cached,buffered,used_
percent,available_percent,buffered_percent)
VALUES
(?,?,?,?,?,?,?,?,?,?,?)

INSERT INTO "benchmark"."nginx"
(tags,ts,accepts,active,handled,reading,requests,waiting
,writing)
VALUES (?,?,?,?,?,?,?,?,?)

INSERT INTO "benchmark"."net"
(tags,ts,bytes_sent,bytes_recv,packets_sent,packets_recv
,err_in,err_out,drop_in,drop_out)
VALUES
(?,?,?,?,?,?,?,?,?,?)

INSERT INTO "benchmark"."cpu"
(tags,ts,usage_user,usage_system,usage_idle,usage_nice,u
sage_iowait,usage_irq,usage_softirq,usage_steal,usage_gu
est,usage_guest_nice)
VALUES
(?,?,?,?,?,?,?,?,?,?,?,?)
```

single-groupby-1-1-1

```
SELECT
date_trunc('minute', ts) as minute,
max(usage_user) AS max_usage_user
FROM cpu
WHERE tags['hostname'] IN ('host_992')
AND ts >= 1451819979159
AND ts < 1451823579159
GROUP BY minute
ORDER BY minute ASC
```

single-groupby-1-8-1

```
SELECT
date_trunc('minute', ts) as minute,
max(usage_user) AS max_usage_user
FROM cpu
WHERE tags['hostname'] IN ('host_383', 'host_1188',
'host_1642', 'host_3536', 'host_3973', 'host_3572',
'host_3785', 'host_1076')
AND ts >= 1451726849998
AND ts < 1451730449998
GROUP BY minute
ORDER BY minute ASC
```

single-groupby-5-1-12	`SELECT date_trunc('minute', ts) as minute, max(usage_user) AS max_usage_user, max(usage_system) AS max_usage_system, max(usage_idle) AS max_usage_idle, max(usage_nice) AS max_usage_nice, max(usage_iowait) AS max_usage_iowait FROM cpu WHERE tags['hostname'] IN ('host_2675') AND ts >= 1451610525883 AND ts < 1451653725883 GROUP BY minute ORDER BY minute ASC`
single-groupby-5-8-1	`SELECT date_trunc('minute', ts) as minute, max(usage_user) AS max_usage_user, max(usage_system) AS max_usage_system, max(usage_idle) AS max_usage_idle, max(usage_nice) AS max_usage_nice, max(usage_iowait) AS max_usage_iowait FROM cpu WHERE tags['hostname'] IN ('host_330', 'host_1736', 'host_3555', 'host_8', 'host_1488', 'host_1630', 'host_1670', 'host_2122') AND ts >= 1451666719477 AND ts < 1451670319477 GROUP BY minute ORDER BY minute ASC`
double-groupby-1	`SELECT date_trunc('hour', ts) AS hour, mean(usage_user) AS mean_usage_user FROM cpu WHERE ts >= 1451610525883 AND ts < 1451653725883 GROUP BY hour, tags['hostname'] ORDER BY hour`
double-groupby-5	`SELECT date_trunc('hour', ts) AS hour, mean(usage_user) AS mean_usage_user, mean(usage_system) AS mean_usage_system, mean(usage_idle) AS mean_usage_idle, mean(usage_nice) AS mean_usage_nice, mean(usage_iowait) AS mean_usage_iowait FROM cpu WHERE ts >= 1451627541687 AND ts < 1451670741687 GROUP BY hour, tags['hostname'] ORDER BY hour`
double-groupby-all	`SELECT date_trunc('hour', ts) AS hour,`

	mean(usage_user) AS mean_usage_user, mean(usage_system)
	AS mean_usage_system, mean(usage_idle) AS
	mean_usage_idle, mean(usage_nice) AS mean_usage_nice,
	mean(usage_iowait) AS mean_usage_iowait, mean(usage_irq)
	AS mean_usage_irq, mean(usage_softirq) AS
	mean_usage_softirq, mean(usage_steal) AS
	mean_usage_steal, mean(usage_guest) AS mean_usage_guest,
	mean(usage_guest_nice) AS mean_usage_guest_nice
	 FROM cpu
	 WHERE ts >= 1451613697096
	   AND ts < 1451656897096
	 GROUP BY hour, tags['hostname']
	 ORDER BY hour

groupby-orderby-limit	SELECT date_trunc('hour', ts) AS hour, mean(usage_user) AS mean_usage_user, mean(usage_system) AS mean_usage_system, mean(usage_idle) AS mean_usage_idle, mean(usage_nice) AS mean_usage_nice, mean(usage_iowait) AS mean_usage_iowait, mean(usage_irq) AS mean_usage_irq, mean(usage_softirq) AS mean_usage_softirq, mean(usage_steal) AS mean_usage_steal, mean(usage_guest) AS mean_usage_guest, mean(usage_guest_nice) AS mean_usage_guest_nice FROM cpu WHERE ts >= 1451612459572 AND ts < 1451655659572 GROUP BY hour, tags['hostname'] ORDER BY hour[3]

About DSI framework

The Distributed Systems Infrastructure (DSI) is a python based framework developed at MongoDB. While its primary use case is to automate end to end performance testing for unattended nightly builds running in Continuous Integration system, it can also be used for manually executed performance benchmarking, such as in this case. In the latter use case the value of using DSI is that the framework guarantees that the exact same configurations were used for each run, and automatically collects all results and log files before shutting down cloud instances.

[3] The CrateDB implementation of this query seems to be a bug: there is no LIMIT in the query. For this version of this report we did not touch the benchmark (TSBS) code at all, so we have omitted this query from the test.

DSI also incorporates a number of kernel and EC2 configurations that have been found to be best practices over time. Many of the configurations are counter-intuitive as they actually decrease performance, but improve repeatability of a given configuration. An example of this is to disable hyper-threading, which almost cuts in half the peak performance, but significantly improves the determinism of the benchmarking results.

DSI is a versatile orchestration framework designed to support multiple different benchmark clients. In the version of DSI available as open source, TSBS was not supported, but support was added as part of this work.

Due to its heritage, DSI originally focused on MongoDB as the System Under Test. To enable this apples-to-apples comparative study, CrateDB sponsored the development work to generalize DSI to support any database (or other product) as the target.

All results presented in this report can be reproduced independently by starting from the configuration stored in https://github.com/ nyrkio/dsi/tree/cratedb-mongodb-tsbs/ configurations/bootstrap/ projects/crate .

More details about DSI and its usage have been published in:

- Ingo & Daly: Automated System Performance Testing at MongoDB
- Ingo & Daly: Reducing Variability in Performance Tests on EC2: Setup and Key Results
- Daly: Creating a Virtuous Cycle in Performance Testing at MongoDB

Test specifications

Infrastructure & scaling

Both database products will be tested with the same underlying infrastructure. As a starting point for these tests, we used the sizing of a CR3 cluster in CrateDB's managed cloud offering, which corresponds to 3 nodes of type m5.2xlarge. (8 vCPU, 32 GB RAM, EBS only)

Using the m5.2xlarge sizing as a model, the actual infrastructure components used for the benchmarks were selected to optimize for repeatable test results.[4]

[4] A comprehensive report on how to configure a database cluster for minimal variation was published in https://www.mongodb.com/blog/post/reducing-variability-performance-tests-ec2-setup-key-results. The findings of that study are directly implemented in DSI, the orchestration framework used in these tests.

Infrastructure specification[5]

EC2 instance type	c7i.4xlarge[6]
CPU	**8 x Intel Xeon Sapphire Rapid 3.2 GHz, Single socket No Hyper Threading, No "Flex"**
Memory	**32 GB, DDR5**
Network	**50 GBit/s**
Disk	**1 TB EBS (io2), 15,000 Provisioned IOPS[7] 40 GBit/s networking bandwidth**

Databases, versions & configurations

CrateDB

Version(s)	**5.7.1**
Java	**21.0.3 Default GC = G1**
Replication factor	**1 (baseline)**
Cratedb Shards[8]	**5 (default)**
Nodes	**1**

[5] https://aws.amazon.com/ec2/instance-types/c7i/

[6] Please note that the Operating System and storage configuration was heavily altered from what a c7i-family instance would usually

[7] https://aws.amazon.com/ebs/provisioned-iops/

[8] In CrateDB each node is host to several shards. Which is a different meaning than MongoDB (and many other databases) has for the same word.

Horizontal scaling	-
Indexing	**Default** (Both row and column storage, all columns indexed)
	Indexing on/default

MongoDB

Versions	**7.0.11 current**
Replication factor	**1 (baseline)**
Partitions[9]	**Default**
Nodes	**1**
Horizontal scaling / Replica sets	-
Schema	**Timeseries** (Internally implemented with a BSON document acting as a bucket that collects all rows/metrics within a configurable time interval, and internally stores each column as its own list)

Workload(s)

The "scaling factor" of TSBS is a function of 2-3 parameters affecting how much data is generated, and the traditional nr of client threads (workers) to scale the amount of concurrent load.

All configurations use the TSBS "Devops" data set, since the "IOT" dataset is not implemented for CrateDB. The Devops data set is basically monitoring data of a typical backend software stack. Data that a PostgreSQL or Redis might send to a Grafana or Datadog service.

[9] In MongoDB each shard is divided into smaller partitions of data called "chunks". A chunk is the unit of data that resides on the same shard and moves between shards when balancing occurs. Other than that the chunk has usually very little direct impact on performance. Hence the default is used and the topic of intra-shard partitioning is ignored for MongoDB.

TSBS 1B rows

The sizing was copied directly from the TSBS README:

```
--scale=4000     --timestamp-start="2016-01-01T00:00:00Z"
--timestamp-end="2016-01-04T00:00:00Z"     --log-interval="10s"
```

This would simulate a datacenter with 4000 different processes (databases, disk subsystems, compute instances...), who each send a record of their monitoring data every 10 seconds. After 3 days, this yields 103 million rows. Finally, the devops dataset consists of 9 different tables:

```
cr> show tables in benchmark;
+------------+
| table_name |
+------------+
| cpu        |
| disk       |
| diskio     |
| kernel     |
| mem        |
| net        |
| nginx      |
| postgresl  |
| redis      |
+------------+
SHOW 9 rows in set (12.520 sec)
cr> select count(*)/1000/1000 AS million_rows from benchmark.cpu;
+--------------+
| million_rows |
+--------------+
|          103 |
+--------------+
SELECT 1 row in set (8.088 sec)
cr> select count(*)/1000/1000 AS million_rows from benchmark.diskio;
+--------------+
| million_rows |
+--------------+
|          103 |
+--------------+
...
```

This scaling factor of a total of 1 Billion rows can be stored in a 31GB gzip file, and takes 180GB on disk when inserted and indexed in CrateDB. As an uncompressed JSON-like text file the data was 278GB. (GOB encoded file)

Results

We executed TSBS against a single node of CrateDB and Mongodb. While this doesn't represent an acceptable production configuration, it is a meaningful first glance into the performance of each database product. The results for a cluster are essentially composed of the sum of single node performance, minus overhead (e.g. networking and locking) of the cluster.

TSBS 1B rows

	CrateDB	MongoDB (timeseries, index: time)	MongoDB (timeseries, index: hostname, time)
load	2M metrics/s 180k rows/s 22.5 r/s/cpu	110k metrics/s	<100k metrics/s
tsbs_single-groupby-1-1-1	0.15 - 3.9 s (median -> max)	11 - 106 s (median -> max)	.004 - 0.05 s (median -> max)
tsbs_single-groupby-1-1-12	0.18 - 1.4 s	33 - 129 s	0.08 - 0.3 s
tsbs_single-groupby-5-1-12	0.19 - 1.3 s	36- 52 s	0.08 - 0.2 s
tsbs_single-groupby-5-8-1	0.17 - 1.3 s	4 - 36 s	0.07 - 0.56
tsbs_double-groupby-1	20 - 28 s	154 - 292 s	167 - 466 s
tsbs_double-groupby-5	31 - 46 s	203 - 506 s	219 - 560 s
tsbs_double-groupby-all	44 - 60 s	266 - 549 s	-

CrateDB Conclusions

CrateDB with default configuration achieves an insert speed of 180k rows / second. For better comparability we also report the insert speed per CPU: 22.5k rows / second. The workload remains CPU bound through the test. (This is by design.)

We tried to tune CrateDB in order to see if we could get better results than the above. Improved results are almost always the case when tuning. In this case though we were unable to improve upon the defaults. This suggests that CrateDB engineers have invested in the user experience of first time users: The product just works without additional tuning needed.

Modifications tried:

- Add a Primary key to all tables (use the timestamp as the PK)
- Turn off or make asynchronous all flushing of the translog
- `INDEX OFF` and then `STORAGE WITH (columnstore = false)`

Each of these ends up having no effect, or a slight (less than 10%) degradation in performance of inserts.

For the simpler group by queries CrateDB median response times are all below 0.2 seconds and the maximums are all below 4 seconds. For the double group by queries this range is from 20 to 60 seconds.

Based on these tests and also comparing to other publicly posted TSBS results, it seems that CrateDB is a viable candidate for time series workloads and analytical queries.

The CrateDB result is even more impressive when we remember that CrateDB actually indexed all columns when we inserted the data. If we wanted to do a similar group by aggregation on some other column than hostname, we can expect similar performance.

CrateDB ad hoc query

To demonstrate the power of ad hoc queries in CrateDB, we modified the TSBS single-group-by query. To emphasize: The author chose these parameters randomly, the same result could be expected by selecting any other columns too, as they are all indexed. The only intentional choices for this query were:

- No use of the ts (timestamp) column anywhere. Timestamp is just one dimension among others and CrateDB can transition to a general OLAP database at any time (pun intended).
- No use of hostname. This tag is commonly used for filtering in TSBS and for example the MongoDB implementation takes advantage of that fact. CrateDB is agnostic on this point and performs equally well without hostname in the query.

ad hoc query

```
cr> SELECT tags['team'] as team,
        max(usage_user) AS max_usage_user

    FROM benchmark.cpu
    WHERE tags['os'] IN ('Ubuntu16.10') AND
tags['region'] IN ('sa-east-1')
    AND usage_idle >= 10 AND usage_idle < 20
        GROUP BY team
        ORDER BY team ASC
    ;
```

```
+------+----------------+
| team | max_usage_user |
+------+----------------+
| CHI  |          100.0 |
| LON  |          100.0 |
| NYC  |          100.0 |
| SF   |          100.0 |
+------+----------------+
SELECT 4 rows in set (0.237 sec)
cr>
```

As you can see, this query returns in the same range as the official TSBS queries did on CrateDB.

While CrateDB results are decent for the official TSBS queries, the ability to query on any column and still retain the same performance is CrateDB's rare, if not unique advantage over most competitors, and we wanted to highlight this ability by going outside the official TSBS queries and verify that this functionality really delivered what it promised.

MongoDB conclusions

The first hurdle with MongoDB is choice: There are many different ways one can store time series data in a MongoDB database. Even if we only focus on the timeseries collections, there's still the need to decide which of the text fields to index.

We therefore present 2 MongoDB results: In the middle column just the **time** column was indexed. In the right most column also a **[hostname, time]** compound index was added.

MongoDB inserts are surprisingly slow: 110k metrics per second is almost 20x slower than CrateDB. For the right most column adding the index on hostname has a visible additional cost, and the throughput quickly drops below 100k / sec.

Also MongoDB is cpu bound, which rules out several potential causes (and thus potential fixes) for the poor performance in the storage layer. To make doubly sure, we even tried the most obvious non-default setting: using j:false on the driver side.

This had no impact on the result - which is expected since disk I/O isn't the bottleneck.

It seems that simply receiving and sorting the incoming time series events into the classic BTree structure that MongoDB uses for indexing, is this much work. It makes sense: BTrees are read optimized and not ideal for write-heavy workloads.

For the read queries we can immediately see that if one can afford to pay the price of adding additional indexes, the result is lightning fast response times for the single group by queries that all either use hostname for filtering the query, or grouping. This is two orders of magnitude faster than CrateDB. However, when querying for any column that wasn't indexed, response times are 10 to 100 seconds, so two orders of magnitude slower.

An obvious question to ask is whether MongoDB could be somehow tuned for better performance. The answer is yes, and indeed one can see hints and traces in the repository itself, since we cloned it from github.com/mongodb-forks.

The optimizations used fall into two categories: MongoDB can benefit from the user pre-sorting the inserted events by time. Second, it makes a difference what order the fields are in the BSON document. Optimizations like these can of course be fun to try in a benchmarking contest, to show off all the tricks the expert in question knows to make MongoDB go faster. However, this author has declined to use such optimizations, as they aren't realistic to use in real life.

The second group of optimizations one could think of - and it seems MongoDB may have used this in the benchmark results they themselves publish - is to turn off compression and other things that would consume CPU. (...since CPU is the bottleneck in our tests.) These are kind of valid tunables to try, however, the difference between CrateDB and MongoDB was so huge, it wasn't meaningful to test different configurations when it is clear they cannot possibly catch up with a 20x difference.

The conclusion for MongoDB therefore is that it isn't a good choice for a pure time series workload, if performance or price performance is a significant consideration. However, we should still not forget that MongoDB actually provided the fastest response times for the queries, if and when the column used by the query - **hostname** in our case - is indexed.

One could therefore consider MongoDB a practical choice in circumstances where one or more of the following are true:

- The team or organization already uses MongoDB in general, or
- Time series data and analytics is only a small part of a bigger system or application
- ...and it's perceived as an advantage to be able to just get the job done with the incumbent database and not consider specialized alternatives.
- The time series data is not larger than tens of gigabytes in volume, and
- The insertion rate of time series data is small and does not dominate the overall writes and reads of the app/system in question. (Maybe it's a few thousand metrics / second.)

Notably, as CrateDB too can be classified as a general purpose database, it in theory shares also these benefits with MongoDB. However, since CrateDB isn't currently as widely used and well known as MongoDB, in practice it's an unlikely path to select CrateDB. However, CrateDB could see adoption happening with this same logic but from the opposite direction - a user might select CrateDB for a use case driven primarily by timeseries or analytical needs, and then expand the use case to also cover OLTP-like usage.