CRATE.IO

CrateDB Benchmark Series

Comparing CrateDB and PostgreSQL Query Performance

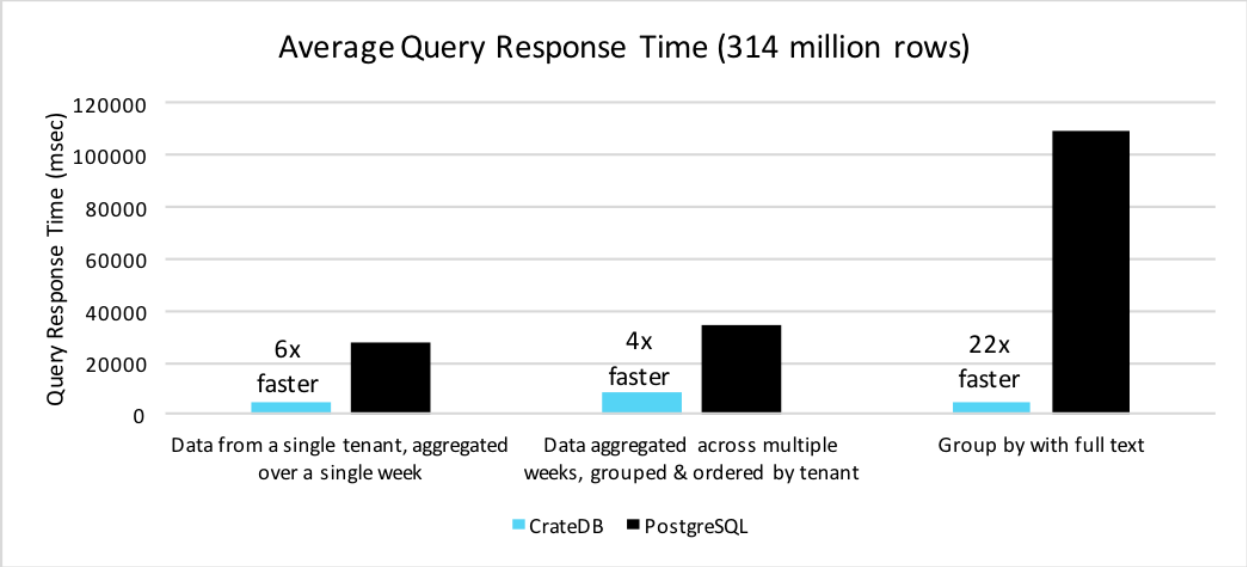
June 2017

Overview

CrateDB is an open core, distributed SQL database. It combines the ease of use of SQL with the horizontal scaling and data model flexibility people associate with NoSQL databases like MongoDB or Apache Cassandra.

CrateDB can perform very fast queries, over incredibly large datasets, in real-time as new data is being ingested. This makes CrateDB particularly well suited for storing and processing machine data (e.g. data collected by IoT devices).

To demonstrate this, we put together a benchmark that compares CrateDB vs PostgreSQL query performance. The results of this benchmark show that CrateDB performs significantly better than PostgreSQL with selects, aggregations, and grouping.



33x better price/performance ration for CrateDB *- CrateDB queried 314 million rows of data up to 22x faster than PostgreSQL, running on hardware that cost 30% less than the hardware on which PostgreSQL ran.*

To help you achieve similar results, we have put together this white paper which explains how the CrateDB and PostgreSQL databases were set up and benchmarked.

BENCHMARK USE CASE

This test simulates an application that collects time series data (sensor readings) from separate customer sites and stores them in a multi-tenant database. This is a common use case; a cloud *Software as a Service* (SaaS) application that enables customers to monitor sensor data via real-time dashboards.

The data set contains 314,496,000 records—data that simulates sensor readings gathered over the period of one year. Each record contains information about the sensor and the sensor readings:

- `uuid` - a universally unique identifier for the sensor reading
- `ts` - a timestamp of the sensor reading
- `tenant_id` - the ID of the customer tenant from which the sensor is reporting
- `sensor_id` - the ID of the sensor
- `sensor_type` - the type of the sensor, represented in a `root:type:subtype` tree-like topology
- `v1` - a sensor reading in the form of an integer
- `v2` - as above
- `v3` - a sensor reading in the form of a floating point number
- `v4` - as above
- `v5` - a sensor reading in the form of a boolean
- `week_generated` - a generated column that contains the week of the `ts`, for indexing/partitioning

With this data, we benchmarked a scenario that is common when dealing with large amounts of numerical sensor data: aggregations. Aggregations, like averaging and totaling, are everyday queries that are useful in real-time analysis of sensor data.

SCHEMA SETUP

Tables were defined in CrateDB and PostgreSQL to hold the data.

CrateDB Schema

```
CREATE ANALYZER "tree" (
  TOKENIZER tree WITH (
    type = 'path_hierarchy',
    delimiter = ':'
  )
);

CREATE TABLE IF NOT EXISTS t1 (
  "uuid" STRING,
  "ts" TIMESTAMP,
  "tenant_id" INTEGER,
  "sensor_id" STRING,
  "sensor_type" STRING,
  "v1" INTEGER,
  "v2" INTEGER,
  "v3" FLOAT,
  "v4" FLOAT,
  "v5" BOOLEAN,
  "week_generated" TIMESTAMP GENERATED ALWAYS AS date_trunc('week', ts),
  INDEX "taxonomy" USING FULLTEXT (sensor_type) WITH (analyzer='tree')
) PARTITIONED BY ("week_generated")
CLUSTERED BY ("tenant_id") INTO 3 SHARDS;
```

PostgreSQL Schema

```
CREATE TABLE IF NOT EXISTS t1 (
  "uuid" VARCHAR,
  "ts" TIMESTAMP,
  "tenant_id" INT4,
  "sensor_id" VARCHAR,
  "sensor_type" VARCHAR,
  "v1" INT4,
  "v2" INT4,
  "v3" FLOAT4,
  "v4" FLOAT4,
  "v5" BOOL,
  "week_generated" TIMESTAMP,
  "taxonomy" ltree
);

CREATE OR REPLACE FUNCTION insert_sensor_reading() RETURNS trigger AS '
    BEGIN
        NEW.week_generated := date_trunc(''week'', NEW.ts);
        RETURN NEW;
    END;
' LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION generate_taxonomy() RETURNS trigger AS '
    BEGIN
        NEW.taxonomy := regexp_replace(NEW.sensor_type, '':(\w):(\w):(\w)'',
''\1.\2.\3'', ''g'')::ltree;
        RETURN NEW;
    END;
' LANGUAGE plpgsql;

CREATE TRIGGER sensor_insert BEFORE INSERT OR UPDATE ON t1 FOR EACH ROW EXECUTE
PROCEDURE insert_sensor_reading();
CREATE TRIGGER generate_taxonomy BEFORE INSERT OR UPDATE ON t1 FOR EACH ROW
EXECUTE PROCEDURE generate_taxonomy();

CREATE INDEX v1_idx ON t1 (v1);
CREATE INDEX v2_idx ON t1 (v2);
CREATE INDEX v3_idx ON t1 (v3);
CREATE INDEX v4_idx ON t1 (v4);
CREATE INDEX ts_idx ON t1 (ts);`
CREATE INDEX tenant_id_idx ON t1 (tenant_id);
CREATE INDEX sensor_type_idx ON t1 (sensor_type);
CREATE INDEX week_generated_idx ON t1 (week_generated);
CREATE INDEX taxonomy_ltree_idx ON t1 USING gist("taxonomy");

CLUSTER t1 USING tenant_id_idx;
```

Differences

There are two differences to point out, having to do with table partitioning and with enabling text searches.

Table Partitioning and Sharding

The sensor data was partitioned by week in CrateDB. The partitions act as virtual tables by grouping sensor readings from the same calendar week together. Once a new week begins, CrateDB automatically creates a new partition. Queries that access data contained within a single partition will perform faster than queries that access data from multiple partitions.

We also divided the CrateDB data into three physical shards using the `CLUSTERED BY` clause in the `CREATE TABLE` statement. Rows having the same value in the routing column are stored in the same shard. Queries that have the routing column in the `WHERE` clause (`tenant_id` in this case) are routed directly to the shard that contains the relevant data.

We considered partitioning the data in PostgreSQL, but the DDL required to create 52 individual weekly partitions was impractical. Refer to the appendix of this document for more detail on partitioning a PostgreSQL table.

Text Searches

The CrateDB DDL defines a `FULLTEXT` index on the `sensor_type` field. This allows CrateDB to run a full-text search query on a hierarchical taxonomy of sensors. We used the *ltree* module to perform the same text search queries in PostgreSQL.

QUERIES

The benchmark simulated multiple users querying the 314 million row database concurrently. Three different SQL queries were used, and are shown below. Parameters in the queries were randomly generated to ensure the queries produced different responses, as they would in real life.

With the exception of the full-text query, the SQL syntax was the same for both CrateDB and PostgreSQL.

Query #1

Data from a single tenant, aggregated over a single week:

```
SELECT min(v1) as v1_min, max(v1) as v1_max, avg(v1) as v1_avg, sum(v1) as v1_sum
FROM t1
WHERE tenant_id = ? AND ts BETWEEN ? AND ?;
```

Query #2

Data aggregated across multiple weeks, grouped and ordered by tenant:

```
SELECT count(*) AS num_docs, tenant_id,
  min(v1) AS v1_min, max(v1) AS v1_max, avg(v1) AS v1_avg, sum(v1) AS v1_sum,
  min(v2) AS v2_min, max(v2) AS v2_max, avg(v2) AS v2_avg, sum(v2) AS v2_sum,
  min(v3) AS v3_min, max(v3) AS v3_max, avg(v3) AS v3_avg, sum(v3) AS v3_sum,
  min(v4) AS v4_min, max(v4) AS v4_max, avg(v4) AS v4_avg, sum(v4) AS v4_sum
FROM t1
WHERE ts BETWEEN ? AND ?
GROUP BY tenant_id
ORDER BY tenant_id;
```

Query #3

Group by with full text.

CrateDB:

```
SELECT sensor_type, COUNT(*) as sensor_count
FROM t1
WHERE taxonomy = ? AND tenant_id = ?
GROUP BY sensor_type;
```

PostgreSQL:

```
SELECT sensor_type, COUNT(*) as sensor_count
FROM t1
WHERE taxonomy <@ ?::ltree AND tenant_id = ?
GROUP BY sensor_type;
```

CLUSTER SETUP

We attempted to keep the hardware assigned to both CrateDB and PostgreSQL as similar as possible. This was a little tricky because CrateDB is a distributed system, and PostgreSQL is not.

To this end, we set up CrateDB v. 1.1.4 with:

- A cluster of 3 equally configured nodes, each with:
  - 8 CPU cores (Intel Xeon CPU D-1541 @ 2.10GHz)
  - 16GB RAM
  - 32GB SSD

We set up PostgreSQL v. 9.6 for RHEL with:

- 1 master and 1 replica node, each with:
  - 16 CPU cores (Intel Xeon CPU E5-2650 v4 @ 2.20GHz)
  - 24GB RAM
  - 256GB SSD

We ran both on the same internal hardware. There were cloud services involved. Both databases have a JDBC driver, and so nothing needed to be changed with our JMeter setup.

The total cost of the PostgreSQL hardware, $9,516 USD, was 30% higher than than the CrateDB hardware cost of $6,102. (Pricing source: Thinkmate.com).

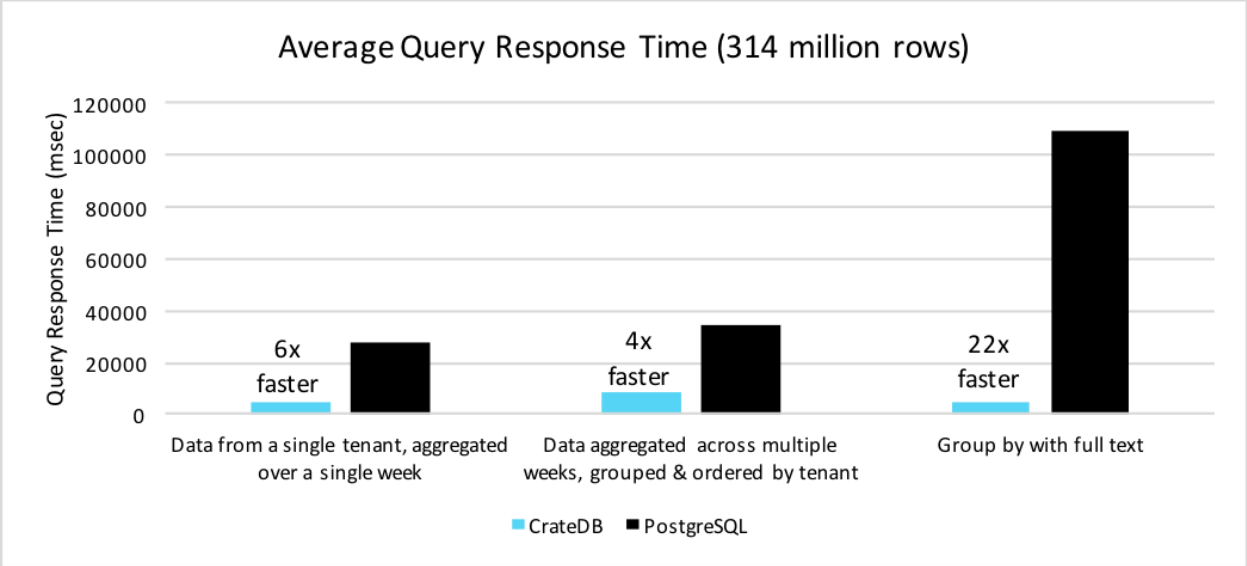
BENCHMARK RESULTS

A JMeter test harness simulated 30 concurrent users, each connecting to the database and executing the three test queries in random sequence and with random query parameters. Each database executed over 4,000 queries during their respective benchmark runs.

These results show the average duration (in milliseconds) for each query to process 314 million rows:

Queries	CrateDB	PostgreSQL	Difference
Data from a single tenant, aggregated a single week	4,811	27,575	6x faster
Data aggregated across multiple weeks, grouped and ordered by tenant	8,871	34,660	4x faster
Group by with full text	4,863	108,966	22x faster

If we graph this, we get:



SUMMARY

CrateDB beats PostgreSQL quite substantially when dealing with selects, aggregations, and grouping. This is due to CrateDB's ability to distribute the workload across an elastic cluster of servers, and so has more CPU cores working in parallel. This is a unique feature of a distributed system and allows you to scale processing performance horizontally with the addition or removal of nodes. Most SQL databases are not distributed and so require you to scale them vertically if you want to increase processing performance.

If you would like to discuss your use-case or need help with CrateDB, please contact us. We can provide guidance on database setup, hardware recommendations, data modelling, query design advice, performance tuning, and so on.

APPENDIX

Here's a brief word on partitioning data in CrateDB vs. PostgreSQL. Partitioning in CrateDB accomplished via a simple "PARTITIONED BY" clause in the create table statement.

PostgreSQL does not have any built-in partitioning based on columns. Instead, you must partition your data manually in a series of steps, as shown here.

First, you must create a master table from which all the partitions will inherit:

```
CREATE TABLE IF NOT EXISTS t1 (
  "uuid" VARCHAR,
  "ts" TIMESTAMP,
  "tenant_id" INT4,
  "sensor_id" VARCHAR,
  "sensor_type" VARCHAR,
  "v1" INT4,
  "v2" INT4,
  "v3" FLOAT4,
  "v4" FLOAT4,
  "v5" BOOL,
  "week_generated" TIMESTAMP
);
```

Then, you must create all the partition tables you need.

Since our scenario involves a year's worth of data partitioned over the corresponding weeks, we need 52 tables. These tables must inherit the schema from the previously defined master table.

```
CREATE TABLE t1_1 (CHECK (EXTRACT(WEEK FROM week_generated) = 1)) INHERITS (t1);
CREATE TABLE t1_2 (CHECK (EXTRACT(WEEK FROM week_generated) = 2)) INHERITS (t1);
...
CREATE TABLE t1_51 (CHECK (EXTRACT(WEEK FROM week_generated) = 51)) INHERITS
(t1);
CREATE TABLE t1_52 (CHECK (EXTRACT(WEEK FROM week_generated) = 52)) INHERITS
(t1);
```

You can then create an index on each of these tables for the column you are using to partition.

```
CREATE INDEX t1_1_week_generated on t1_1 (week_generated);
CREATE INDEX t1_2_week_generated on t1_2 (week_generated);
...
CREATE INDEX t1_51_week_generated on t1_51 (week_generated);
CREATE INDEX t1_52_week_generated on t1_52 (week_generated);
```

You must then define the partitioning logic in a function and apply this to the master table.

```
CREATE OR REPLACE FUNCTION data_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
  IF (EXTRACT(WEEK FROM NEW.ts) = 1) THEN
    INSERT INTO t1_1 VALUES (NEW.*);
  ELSIF (EXTRACT(WEEK FROM NEW.ts) = 2) THEN
   INSERT INTO t1_2 VALUES (NEW.*);
...
  ELSIF (EXTRACT(WEEK FROM NEW.ts) = 51) THEN
   INSERT INTO t1_51 VALUES (NEW.*);
  ELSIF (EXTRACT(WEEK FROM NEW.ts) = 52) THEN
   INSERT INTO t1_52 VALUES (NEW.*);
  ELSE
      RAISE EXCEPTION 'Week out of range: (%)', EXTRACT(WEEK FROM NEW.ts);
  END IF;
  RETURN NULL;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER insert_sensor_trigger
    BEFORE INSERT ON t1
    FOR EACH ROW EXECUTE PROCEDURE data_insert_trigger();
```

Unfortunately, this leads to a very large schema (300+ lines in this case), which might be manageable, *if the partitions can be defined up-front*. Unfortunately, because dates are non-repeating, it is impossible to pre-define all possible partitions, unless you have a definite end-date.