PGCONF.US NYC 2025

Mastering Time-Series Data in PostgreSQL
Advanced Partitioning Strategies and BRIN Indexes to Speed Up Ingestion

Domenico di Salvia (he/him)

Sr. WW SSA PostgreSQL, EMEA
Amazon Web Services

Agenda

- The problem
- The nature of time-series data
- Benchmark scenario
- Improving with:
    - ...the right data types
    - ...the right indexes
    - ...partitioning
- Q&A

What’s the problem here?

Key Challenges:

- High write throughput
- Query performance
- Storage efficiency
- Maintenance overhead

Common Scenarios:

- IoT sensor data
- Financial market data
- Application performance metrics
- Log aggregation systems
- Monitoring systems

Single Table Problem

```
CREATE TABLE sensor_data (
      timestamp TIMESTAMPTZ,
      sensor_id INTEGER,
      value NUMERIC,
      metadata JSONB
);
```

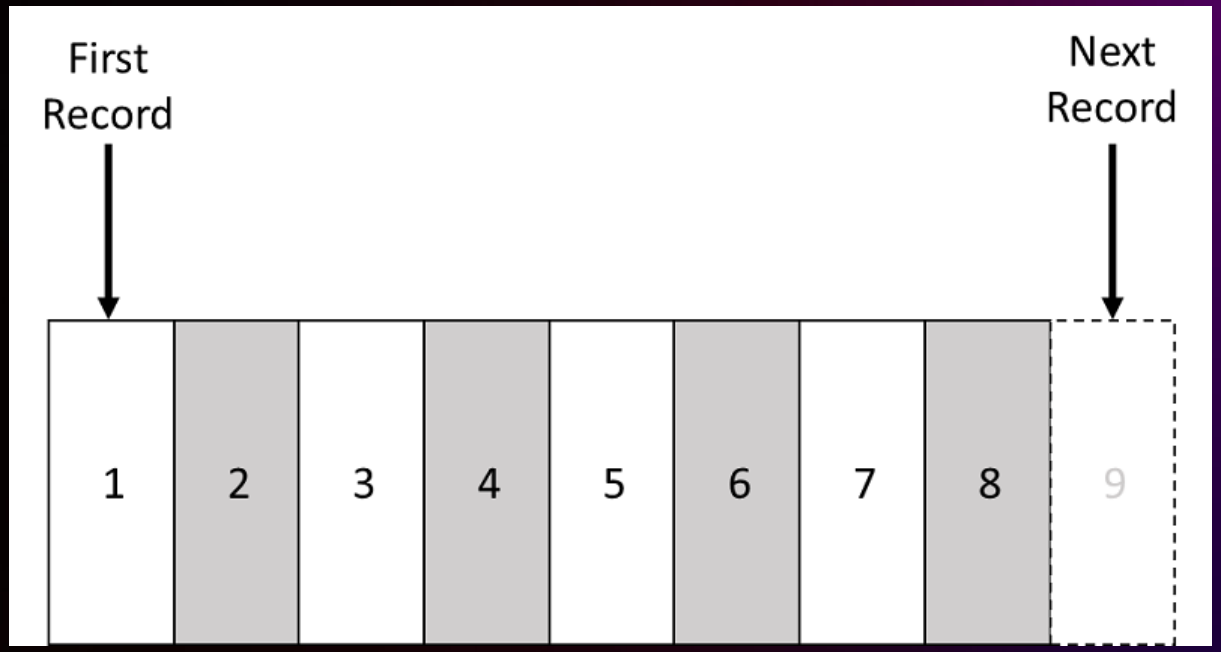
Performance Impact:

- Index Bloat
- Vacuum operations
- Query planning
- Backup/restore

The nature of time-series data

Few key characteristics:

1. Treated as an immutable append-only log
2. Data is inserted in time order



Benchmark scenario

The benchmark simulates a fictional company that tracks metrics on its fleet of trucks.

To simulate the ingestion of the metrics, the benchmark loads pre-generated data using the PostgreSQL COPY command in parallel threads.

- Approximately 1 Year of Data
- 100 Trucks
- 586 million rows of data
- Data size of about 133GB

Credits: https://github.com/timescale/tsbs

Benchmark scenario continued

```
CREATE TABLE readings (
    time                TIMESTAMPTZ,
    tags_id             INTEGER,
    name                TEXT,
    latitude            DOUBLE PRECISION,
    longitude           DOUBLE PRECISION,
    elevation           DOUBLE PRECISION,
    velocity            DOUBLE PRECISION,
    heading             DOUBLE PRECISION,
    grade               DOUBLE PRECISION,
    fuel_consumption    DOUBLE PRECISION,
    additional_tags     JSONB
);
```

```
CREATE INDEX readings_latitude_time_idx
ON readings
USING btree (latitude, time DESC);

CREATE INDEX readings_tags_id_time_idx
ON readings
USING btree (tags_id, time DESC);

CREATE INDEX readings_time_idx
ON readings
USING btree (time DESC);
```

Benchmark scenario continued

relname	bytes	pg_size_pretty
diagnostics	22226821120	21 GB
diagnostics_fuel_state_time_idx	15525781504	14 GB
diagnostics_tags_id_time_idx	16172163072	15 GB
diagnostics_time_idx	12988997632	12 GB
readings	34228625408	32 GB
readings_latitude_time_idx	12058607616	11 GB
readings_tags_id_time_idx	16003366912	15 GB
readings_time_idx	12963602432	12 GB
tags	60825600	58 MB
tags_id_seq	8192	8192 bytes
tags_name_idx	22192128	21 MB
tags_pkey	14811136	14 MB

First Ingestion Trial Results

- Amazon RDS for PostgreSQL db.r6g.2xlarge (Graviton2, 8 vCPU, 64 GiB memory)
- Storage type io1 with 20k provisioned IOPS
- 4 parallel workers loading data (COPY)

Results:

- Load Time → 5,609 seconds (93.5 minutes)
- At a rate of 522,832 metrics per second

...by using the right Data Types

- ID column → SMALLINT? *...maybe better INT (up to 2 billion of values)*
- Metrics columns → from DOUBLE PRECISION to REAL

```
ALTER TABLE readings
      ALTER COLUMN elevation TYPE REAL,
      ALTER COLUMN velocity TYPE REAL,
      ALTER COLUMN heading TYPE REAL,
      ALTER COLUMN grade TYPE REAL,
      ALTER COLUMN fuel_consumption TYPE REAL;
```

Second Ingestion Trial

- Size of the database goes from 133GB to 126GB (-5.2%)
- Less storage and likely less CPU cycles needed
- More rows can be held in memory

Results:

- Load Time → 5,487 seconds (about 91 minutes)
- At a rate of 534,517 metrics per second

And even the order of the columns does matter...

- Group columns with similar data types together (integer, text, blob, ...)
- Put the fixed size columns first
- Put the most frequently used columns first
- Don’t use VARCHAR if you can make better choices (e.g., CHAR, INT, DATE, TIMESTAMP, ...)

Benefits:

- Less disk space
- Better performances
- CPU and RAM used efficiently

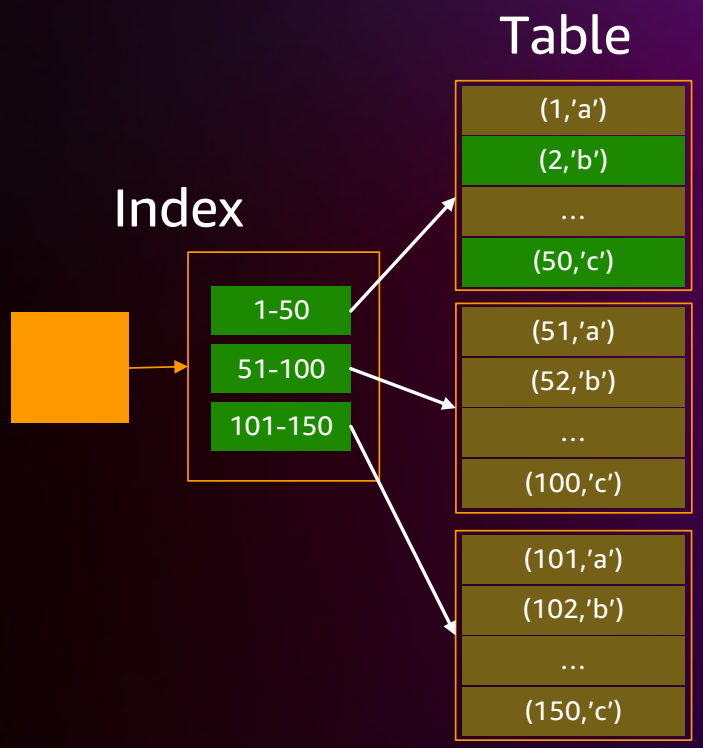
Credits: Hans-Jürgen Schönig (PGConf.EU Berlin 2022)
https://www.cybertec-postgresql.com/en/column-order-in-postgresql-does-matter/

...by using the right Index Types

- B-Tree (default index type)
- Hash
- GiST (Generalized Search Tree)
- SP-GiST (Space Partitioned GiST index)
- GIN (Generalized Inverted Index)
- BRIN (Block Range Index) *→ perfect for timeseries data...*
- bloom (extension)

BRIN (Block Range Index)

- It stores Min/Max values for a range of data pages
- Uses less storage (order of magnitude)
- Tiny index designed to index large tables
- Supports equality and range queries, supported operators: < <= = >= >
- Ideal for natural ordered table, examples:
  - timestamps
  - IoT sensor data




```
CREATE INDEX readings_time_brin_idx
    ON readings
 USING BRIN (time)
  WITH (pages_per_range = 32);
```

Third Ingestion Trial

- Size of the database goes from 126GB to 101GB (-19.8%)
- The size of both BRIN indexes are only 24 KB in this scenario…
- … meaning better performance for metrics ingestion

Results:

- Load Time → 4,761 seconds (about 79 minutes)
- At a rate of 616,002 metrics per second

But, what about read performances with BRIN?

```
EXPLAIN ANALYZE
    SELECT count(*)
      FROM readings
     WHERE time BETWEEN
     '2025-06-25'
       AND
     '2025-06-26';
```

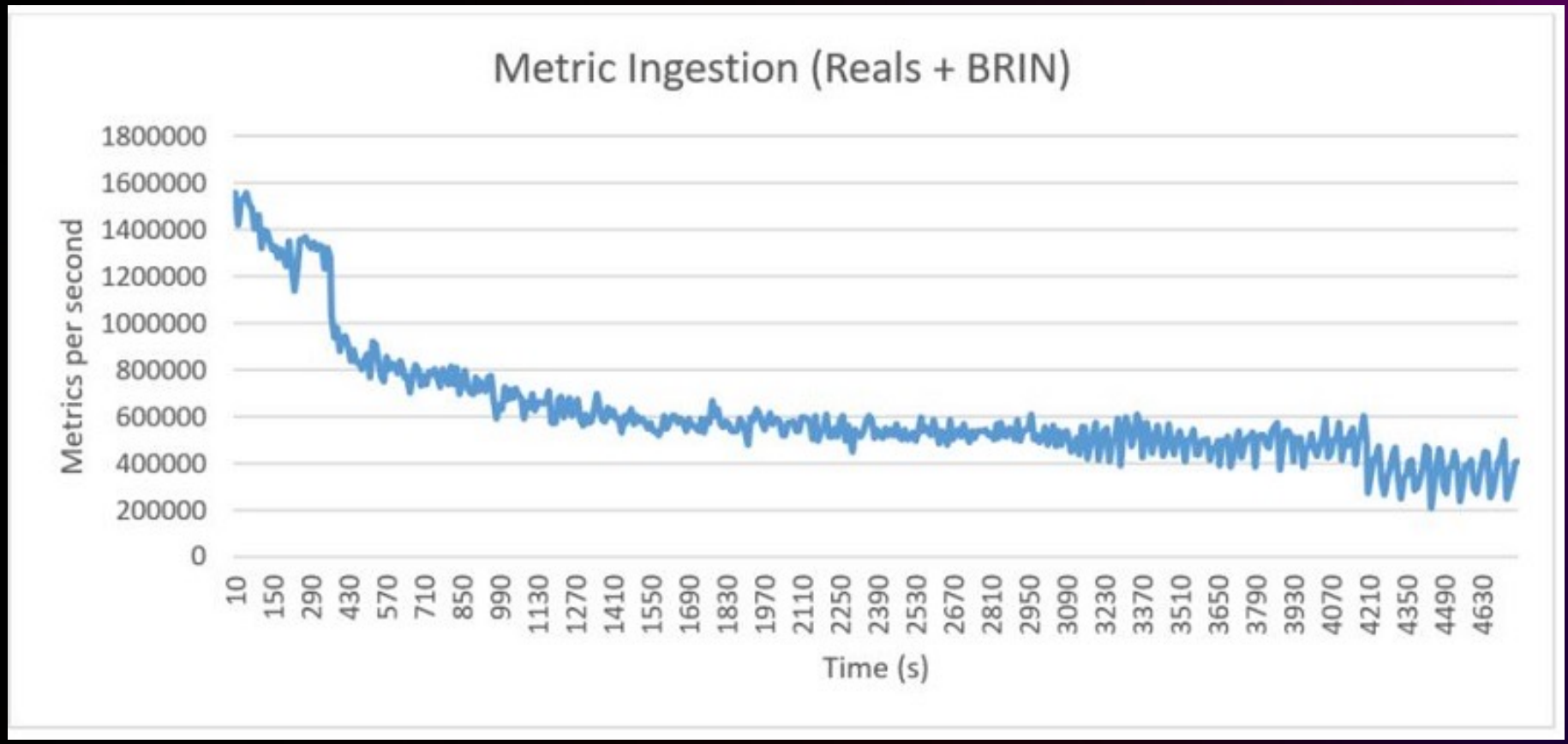
```
Aggregate  (cost=43256.51..43256.52 rows=1 width=8)
      (actual time=222.670..222.671 rows=1 loops=1)
  ->  Index Only Scan using readings_time_idx on readings
                    (cost=0.57..41335.85 rows=768264 width=0)
                    (actual time=0.024..174.810 rows=777456 loops=1)
        Index Cond: (("time" >= '2020-12-25 00:00:00+00'::timestamp with
                    time zone) AND ("time" <= '2020-12-26
                    00:00:00+00'::timestamp with time zone))
        Heap Fetches: 777456
Planning Time: 0.079 ms
Execution Time: 222.701 ms
(6 rows)
```

But, what about read performances with BRIN?

```
EXPLAIN ANALYZE
    SELECT count(*)
      FROM readings
     WHERE time BETWEEN
     '2025-06-25'
       AND
     '2025-06-26';
```

```
Aggregate  (cost=6054759.35..6054759.36 rows=1 width=8)
           (actual time=176.459..176.460 rows=1 loops=1)
  ->  Bitmap Heap Scan on readings
           (cost=1824.41..6052838.72 rows=768252 width=0)
           (actual time=18.612..129.882 rows=777456 loops=1)
        Recheck Cond: (("time" >= '2020-12-25 00:00:00+00'::timestamp with
              time zone) AND ("time" <= '2020-12-26 00:00:00+00'::timestamp
              with time zone))
        Rows Removed by Index Recheck: 19762
        Heap Blocks: lossy=9728
        ->  Bitmap Index Scan on readings_time_brin_idx
           (cost=0.00..1632.34 rows=768329 width=0)
           (actual time=18.540..18.540 rows=97280 loops=1)
              Index Cond: (("time" >= '2020-12-25 00:00:00+00'::timestamp
          with time zone) AND ("time" <= '2020-12-26 00:00:00+00'::timestamp
          with time zone))
Planning Time: 0.080 ms
Execution Time: 176.494 ms
```

Additional considerations



...by implementing Partitioning

Declarative Partitioning introduced in PostgreSQL version 10

Benefits:

- Divide & Conquer
- Partition Pruning
- Parallel Maintenance and Data Retrieval
- Efficient Data Lifecycle

Partitioning Strategies

Range Partitioning

Data is placed in partitions based on a range of values → *perfect for timeseries data...*

List Partitioning

Data is placed in partitions based on a list of discrete values

Hash Partitioning

Data is placed in partitions based on a hash algorithm applied to a key

Range Partitioning

- Not all partitions need to be defined
- Can not have overlapping ranges
- The special values MINVALUE and MAXVALUE can be used to indicate that there is no lower or upper bound
- The value can not be NULL

```
CREATE TABLE sensor_data (
      timestamp TIMESTAMPTZ NOT NULL,
      sensor_id INTEGER,
      value NUMERIC,
      metadata JSONB)
PARTITION BY RANGE (timestamp);

CREATE TABLE sensor_data_y2025m09
PARTITION OF sensor_data
    FOR VALUES FROM ('2025-09-01')
    TO ('2025-10-01');
```

Range Partitioning continued

Interval	Use Case	Pros	Cons
Daily	High-volume IoT	Fine-grained pruning	Many partitions
Weekly	Moderate volume	Balanced approach	Less granular
Monthly	Lower volume	Fewer partitions	Larger partitions

pg_partman - https://github.com/pgpartman/pg_partman

pg_cron - https://github.com/citusdata/pg_cron

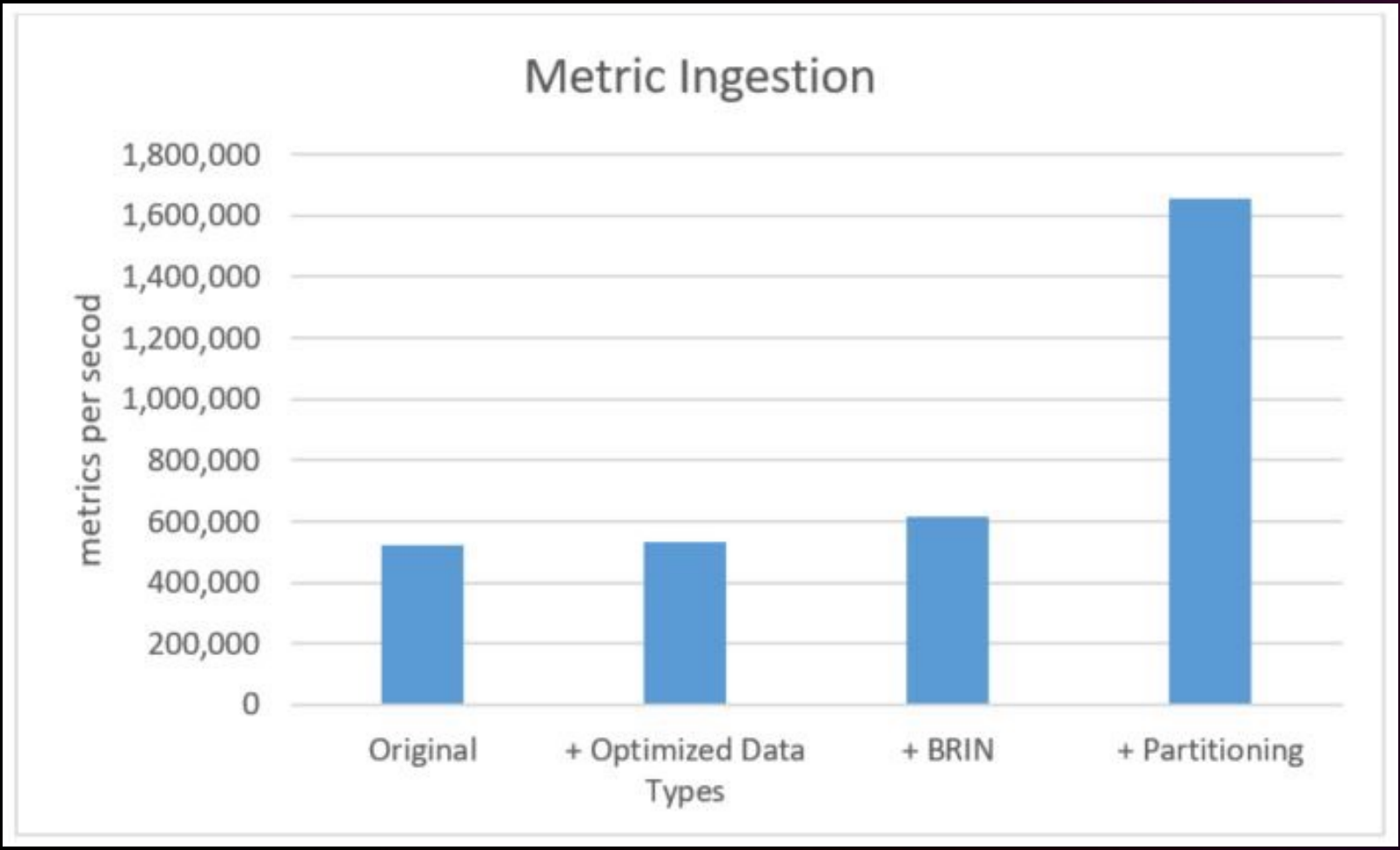
Fourth Ingestion Trial

- Size of the database is the same (101GB)

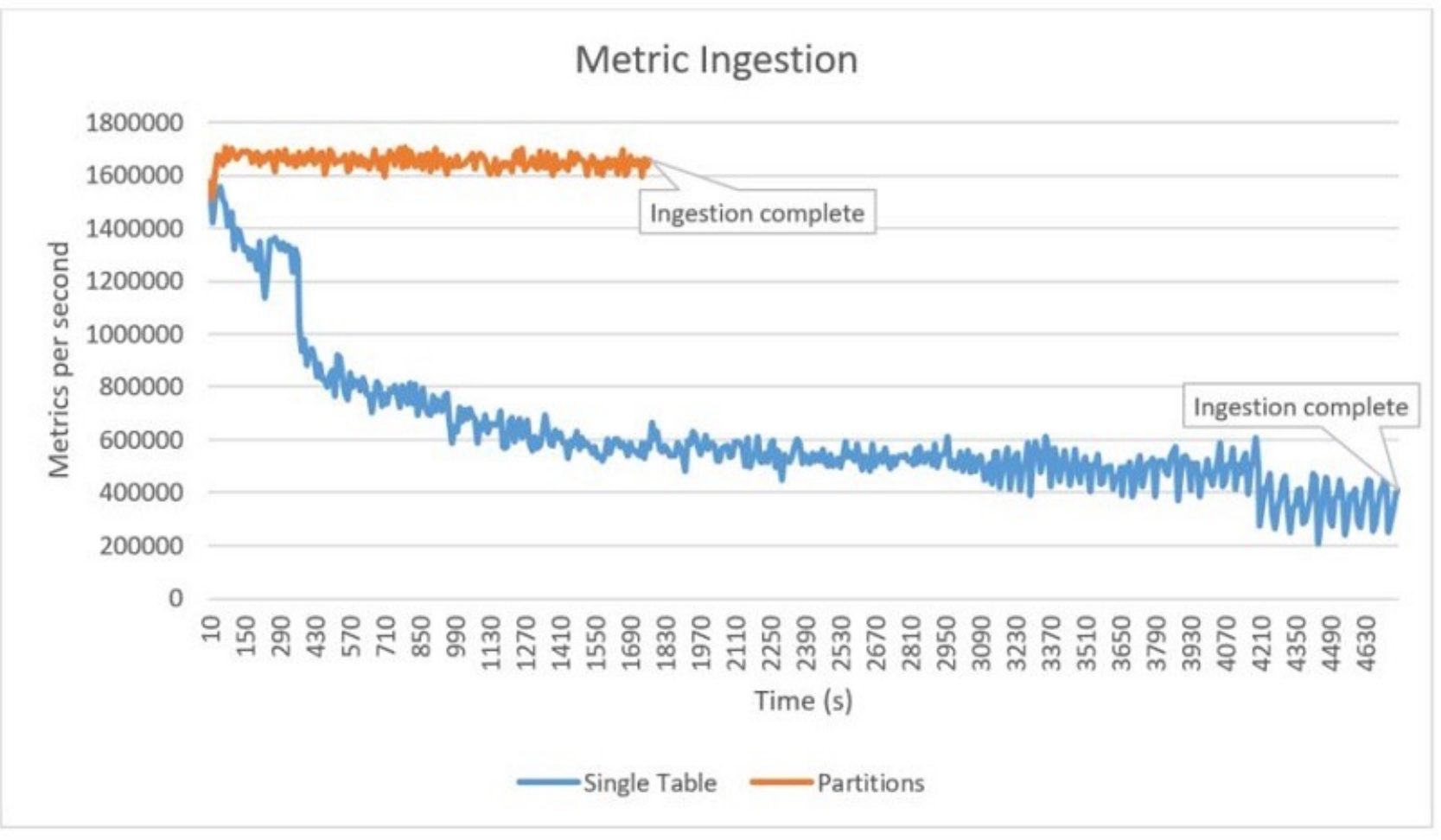
Results:

- Load Time → 1,774 seconds (about 29 minutes, -62.7%)
- At a rate of 1,653,024 metrics per second

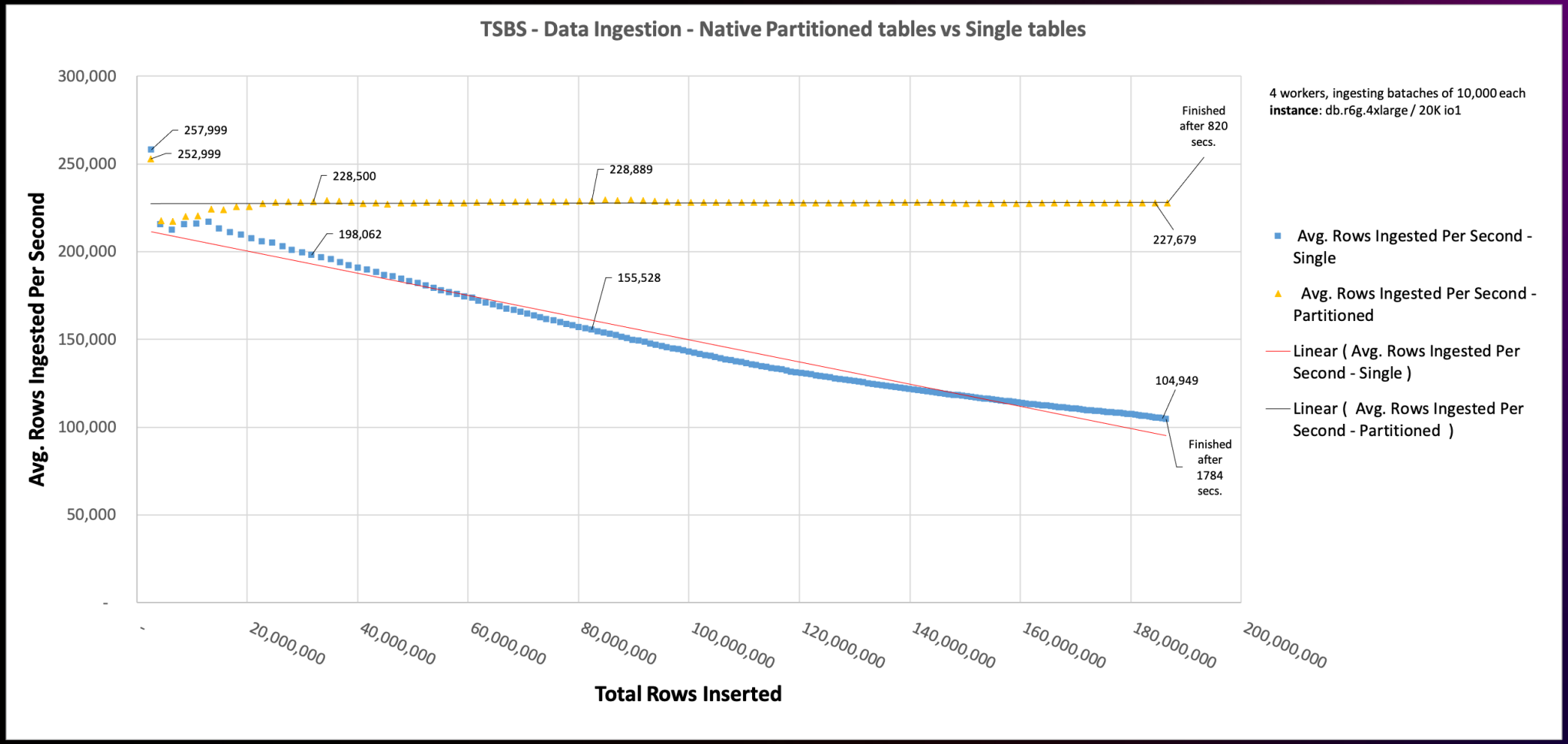
Results Summary



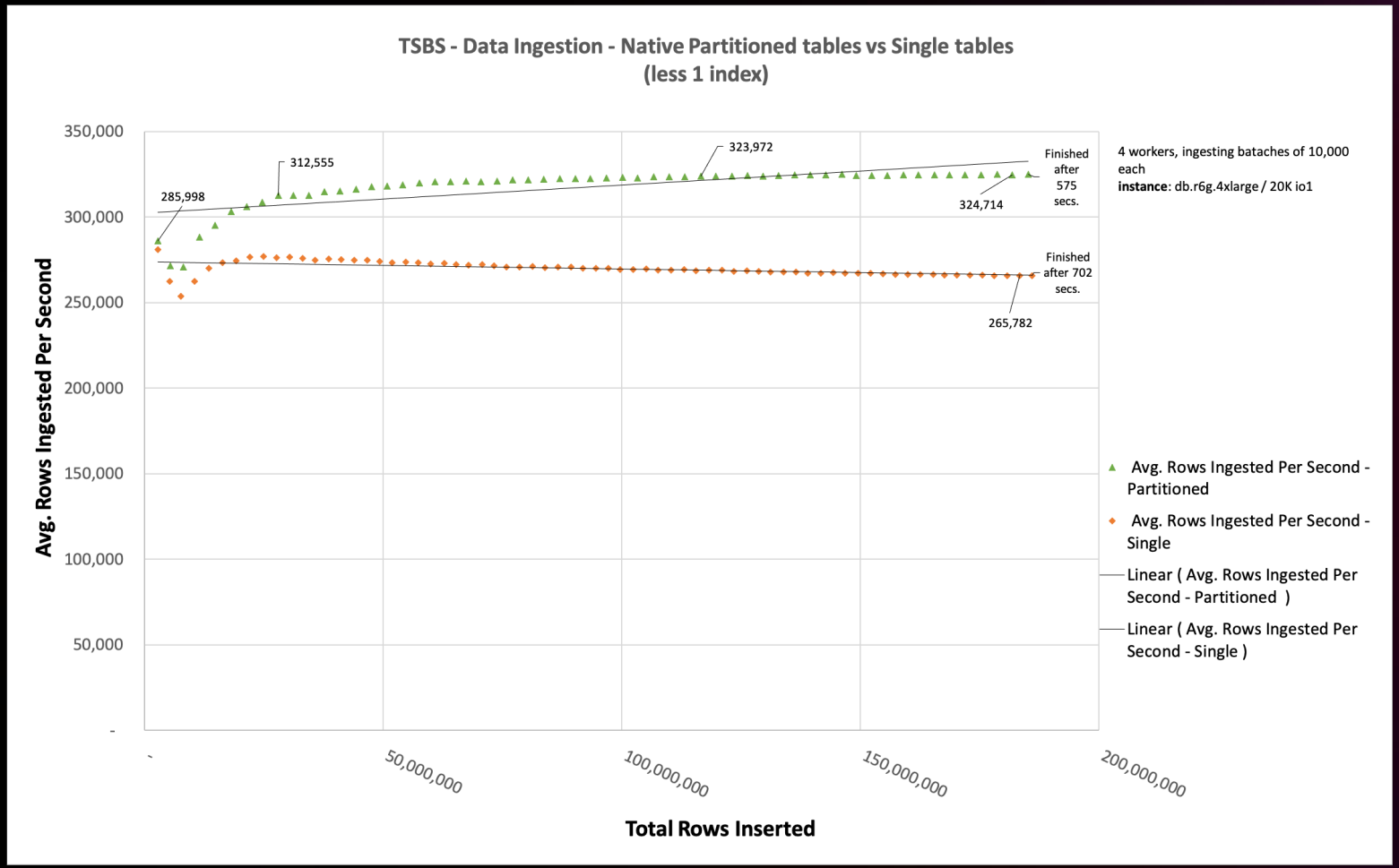
Results Summary (continued)



Results Summary (continued)



Results Summary (continued)



Demo

Any Question?

Thank you!

Domenico di Salvia

dssalvia@amazon.it

https://www.linkedin.com/in/domenicodisalvia/

Credits:

Jim Mlodgenski – Sr. Principal Engineer RDS

Andy Katz – Sr. Mgr Aurora Open-Source Services

Domenico di Salvia

Sr. WW PostgreSQL SSA presso

Amazon Web Services (AWS)