Building a scalable time-series database using Postgres

Mike Freedman

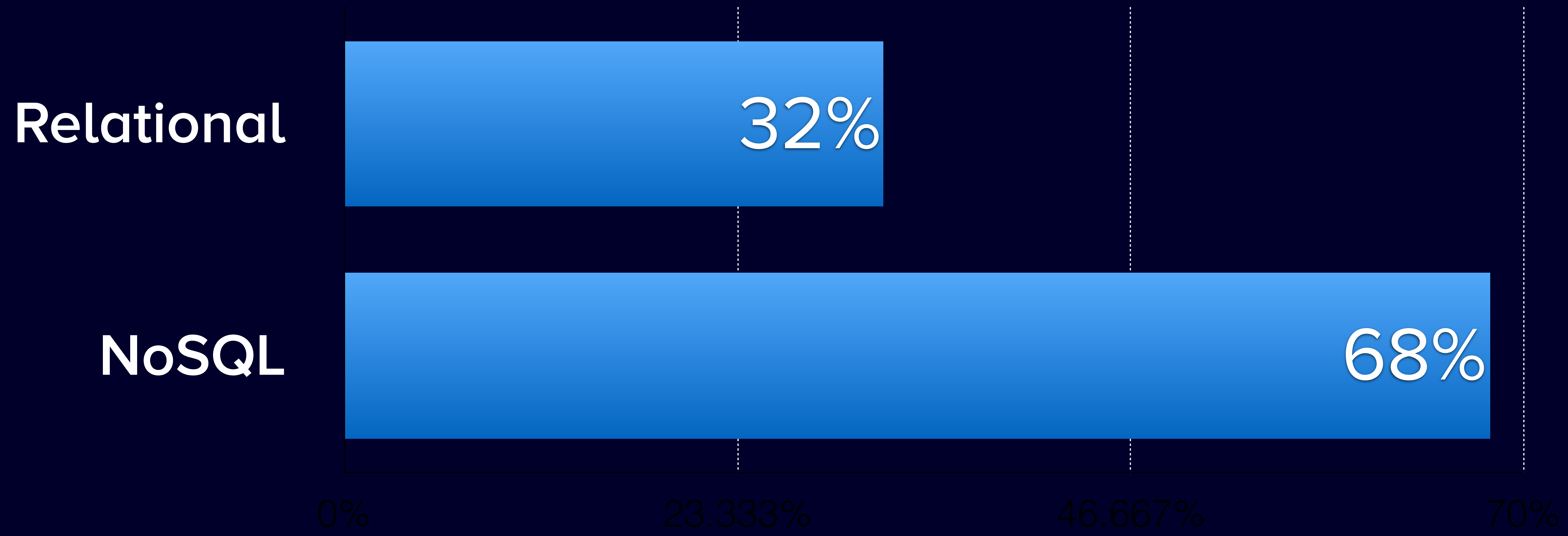
Co-founder / CTO, Timescale

mike@timescale.com

https://github.com/timescale/timescaledb


Time-series data is everywhere,
greater volumes than ever before

What DB for time-series data?



https://www.percona.com/blog/2017/02/10/percona-blog-poll-database-engine-using-store-time-series-data/

Why so much NoSQL?

1. Schemas are a pain
2. Scalability!

Postgres, MySQL:

- JSON/JSONB data types
- Constraint validation!

1. Schemas are a pain
2. Scalability!

Why don't relational DBs scale?

Two Challenges

1. Scaling **up**: Swapping from disk is expensive

2. Scaling **out**: Transactions across machines expensive

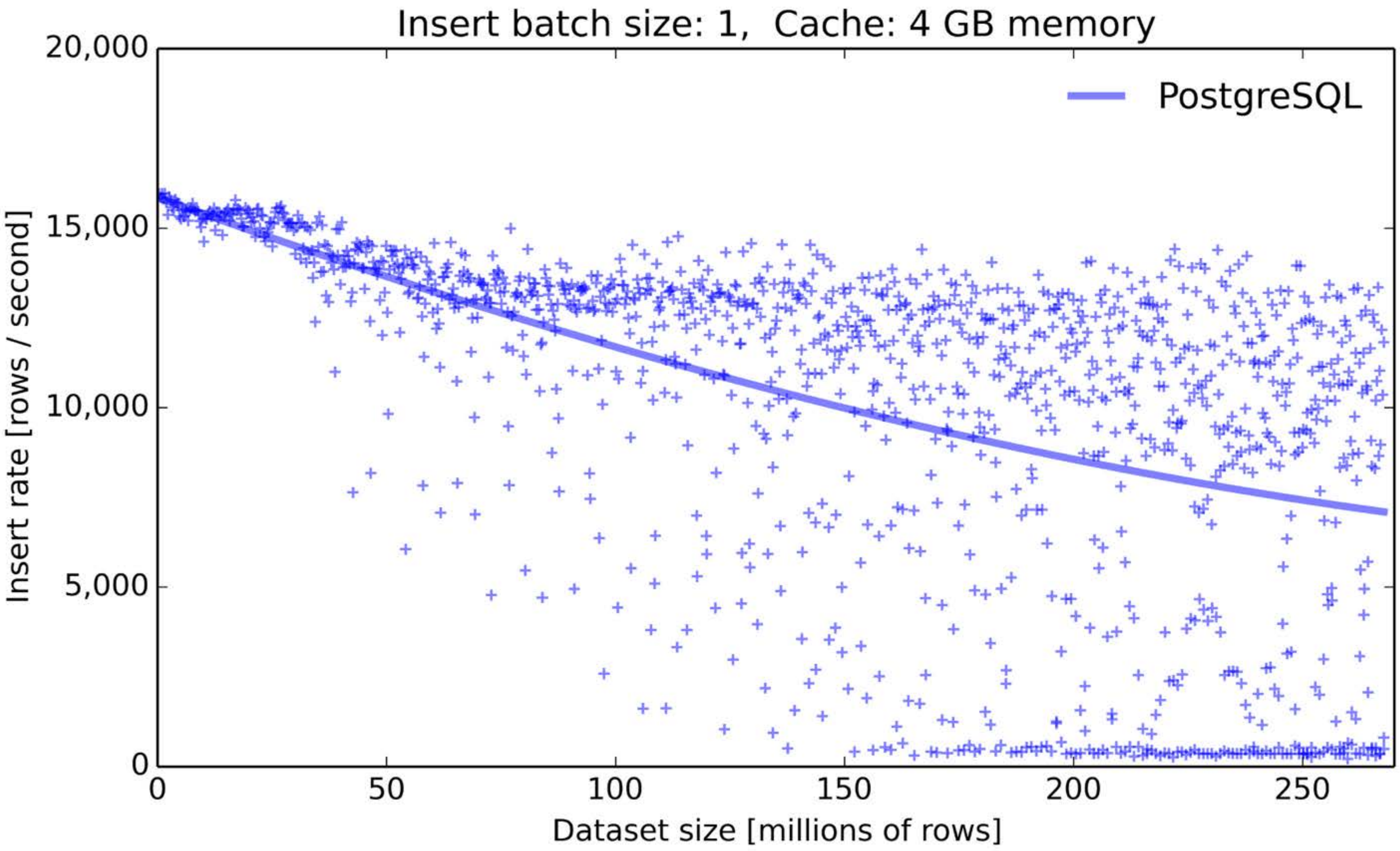
Two Challenges

1. Scaling **up**: Swapping from disk is expensive

2. Scaling **out**: Transactions across machines expensive

Not applicable:

1. Don't need for time-series
2. NoSQL doesn't solve anyway



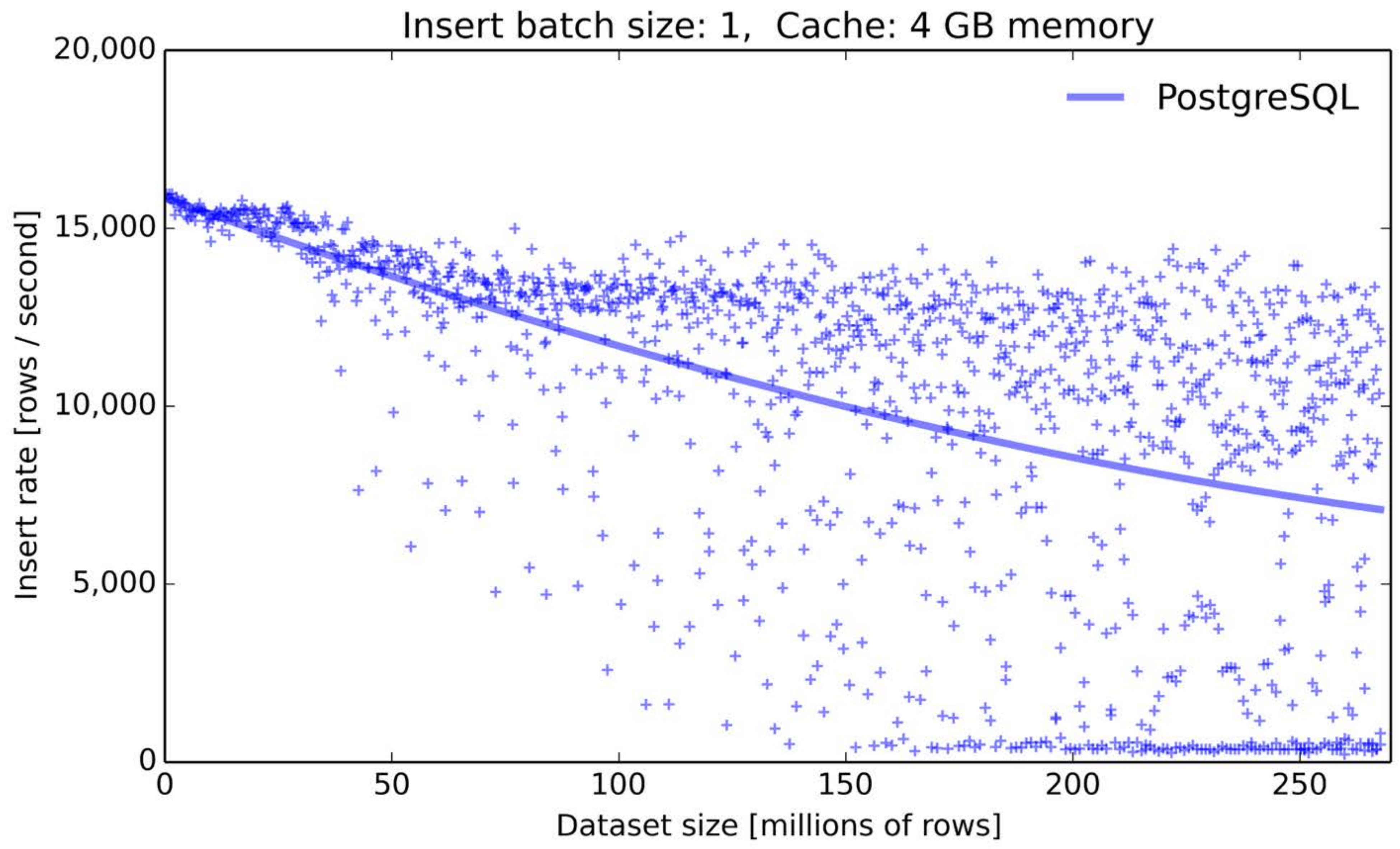
Postgres 9.6.2 on Azure standard DS4 v2 (8 cores), SSD (premium LRS storage)

Each row has 12 columns (1 timestamp, indexed 1 host ID, 10 metrics)

TIMESCALE

Challenge in Scaling Up

- **As table grows large:**
  - Data and indexes no longer fit in memory
  - Reads/writes to random locations in B-tree
  - Separate B-tree for each secondary index



- **I/O amplification makes it worse**
  - Reads/writes at full-page granularity (8KB), not individual cells
  - Doesn't help to shrink DB page: HDD still seeks, SSD has min Flash page size

Enter NoSQL and Log-Structured Merge Trees

(and new problems)

Google BIGTABLE LEVELDB cassandra mongoDB + WT InfluxDB

- **LSM trees avoid small, in-place updates to disk**
  - Keep latest inserts/updates in memory table
  - Write immutable sorted batch to disk
  - In-memory indexes typically maps to batches
- **But comes at cost**
  - Large memory use: multiple indexes, no global ordering
  - Poor secondary index support

Is there a better way?

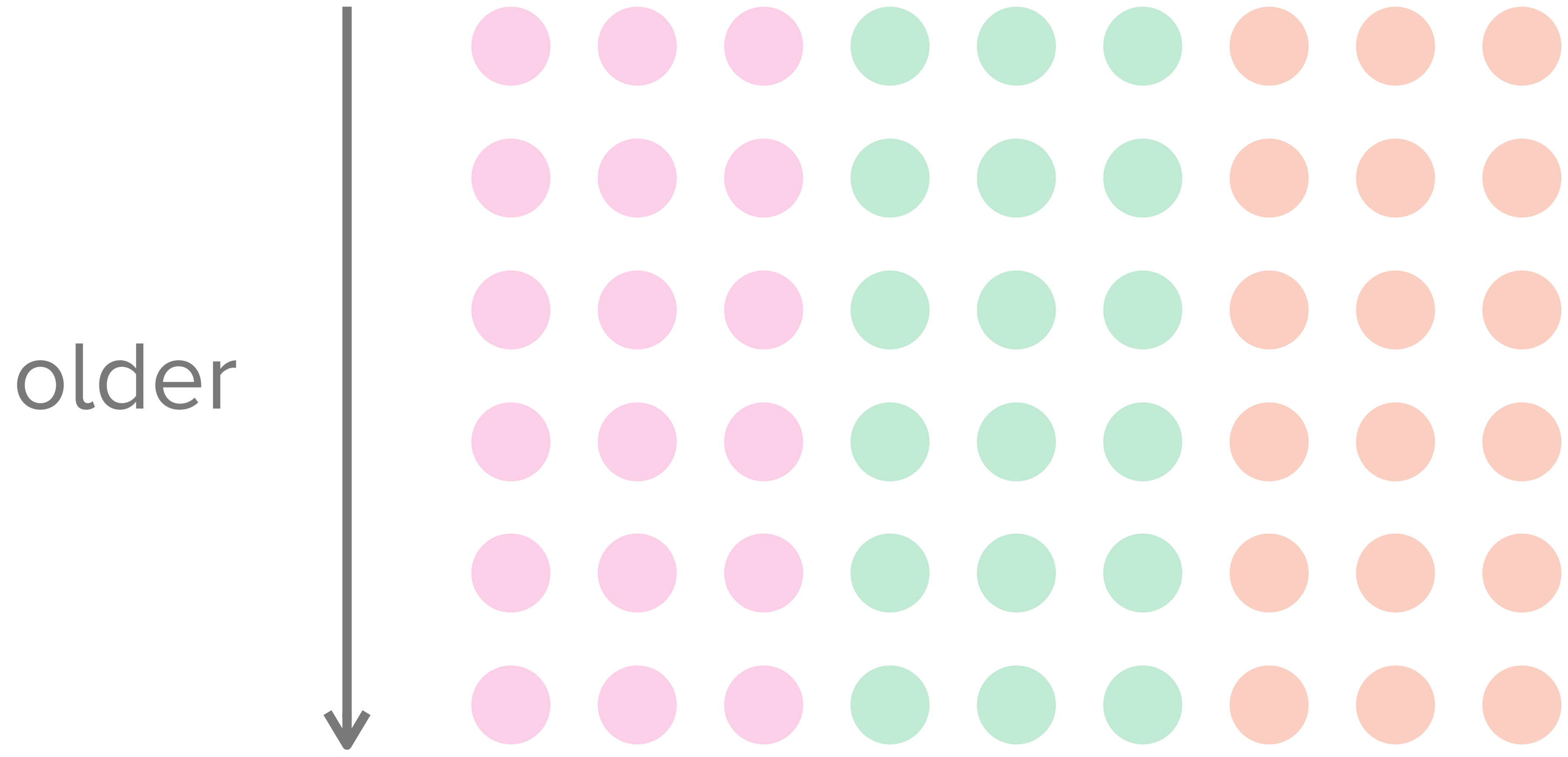
Yes.
Time-series workloads are different

OLTP

- ✗ Primarily UPDATEs
- ✗ Writes randomly distributed
- ✗ Transactions to multiple primary keys

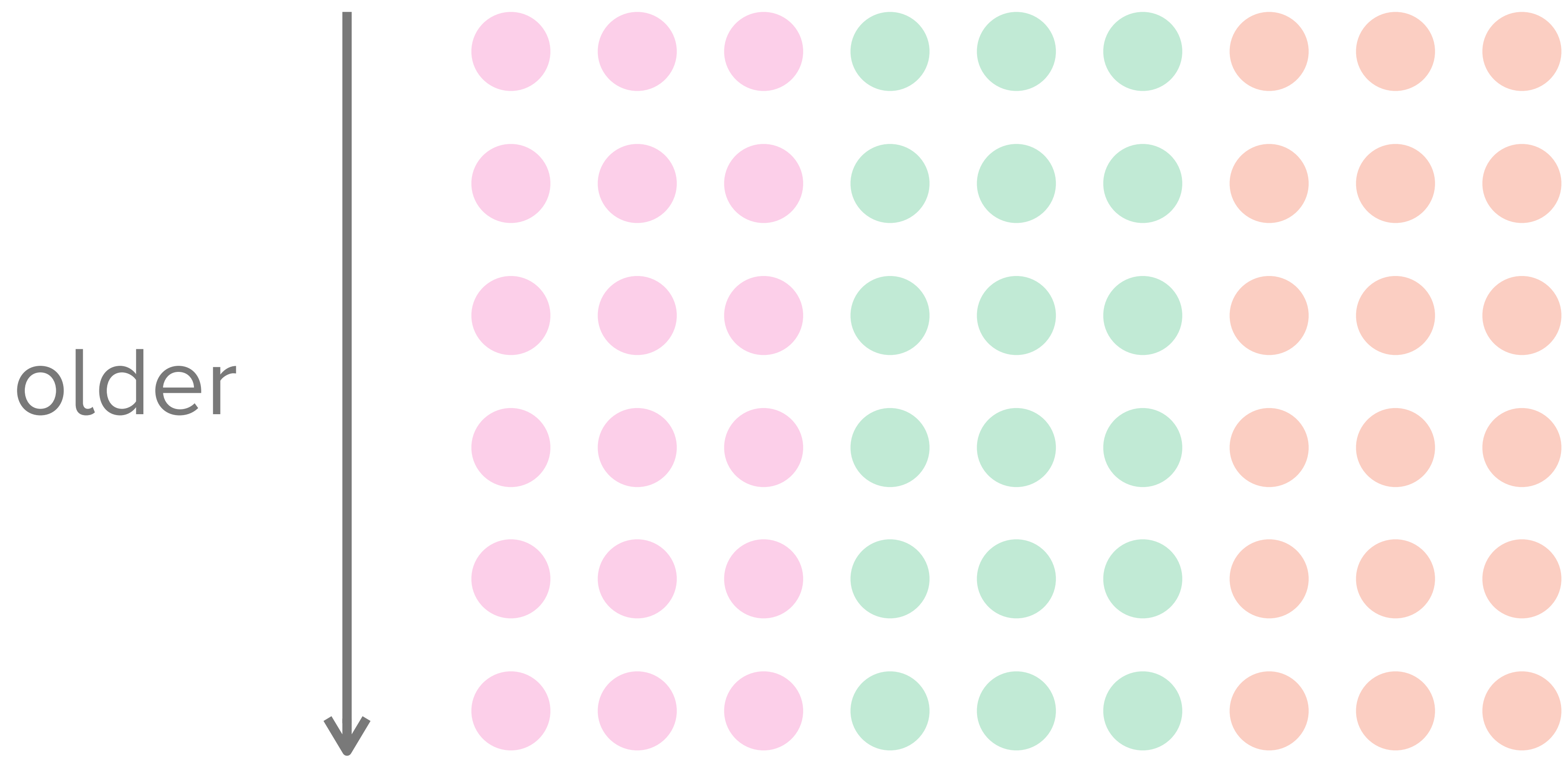
Time Series

- ✓ Primarily INSERTs
- ✓ Writes to recent time interval
- ✓ Writes associated with a timestamp and primary key



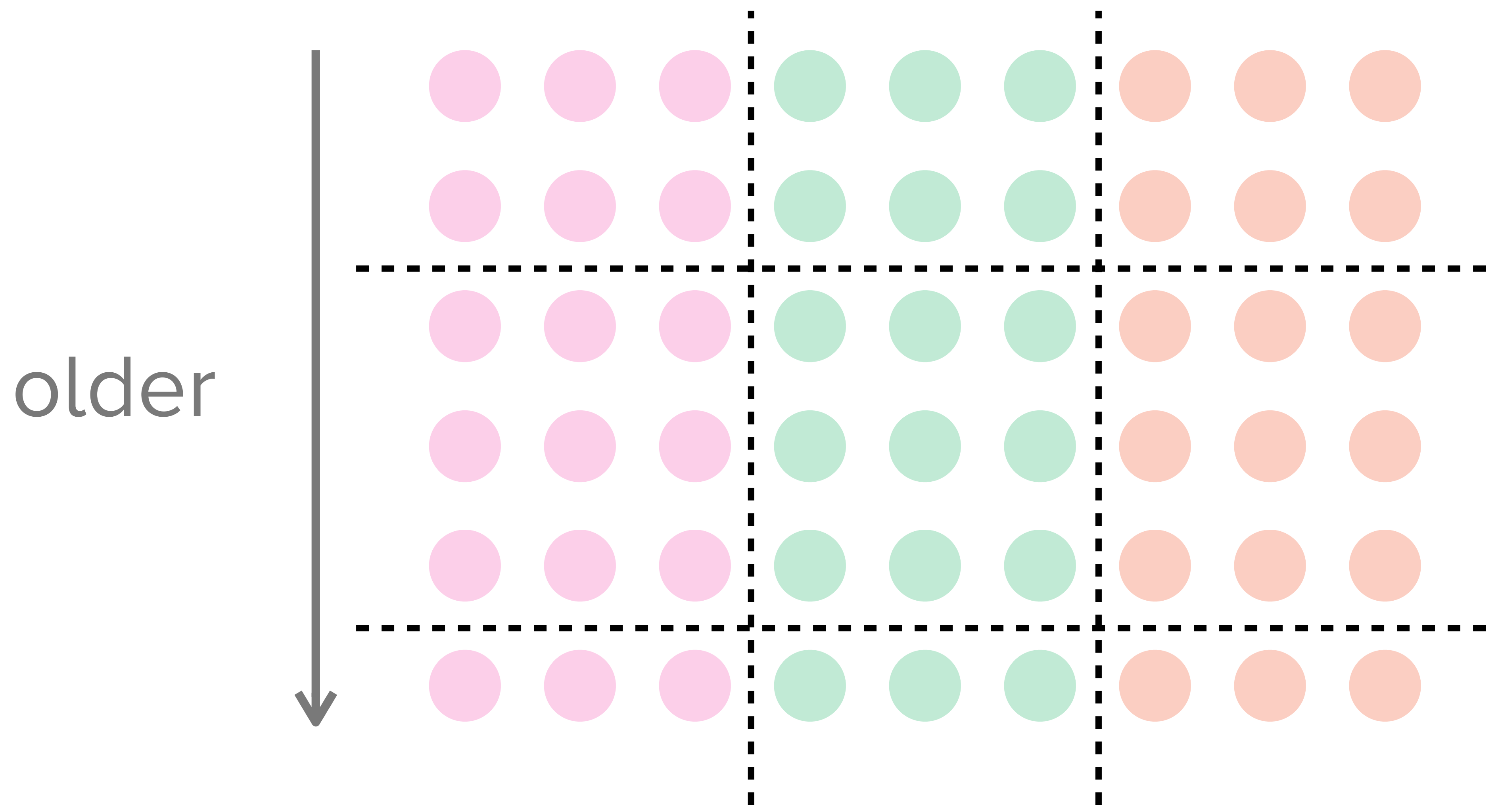
TIMESCALE

- **Strawman: Just use time as primary index?**
  - Yes? Writes are to recent time, can keep in memory
  - Nope! Secondary indexes still over entire table



Adaptive time/space partitioning

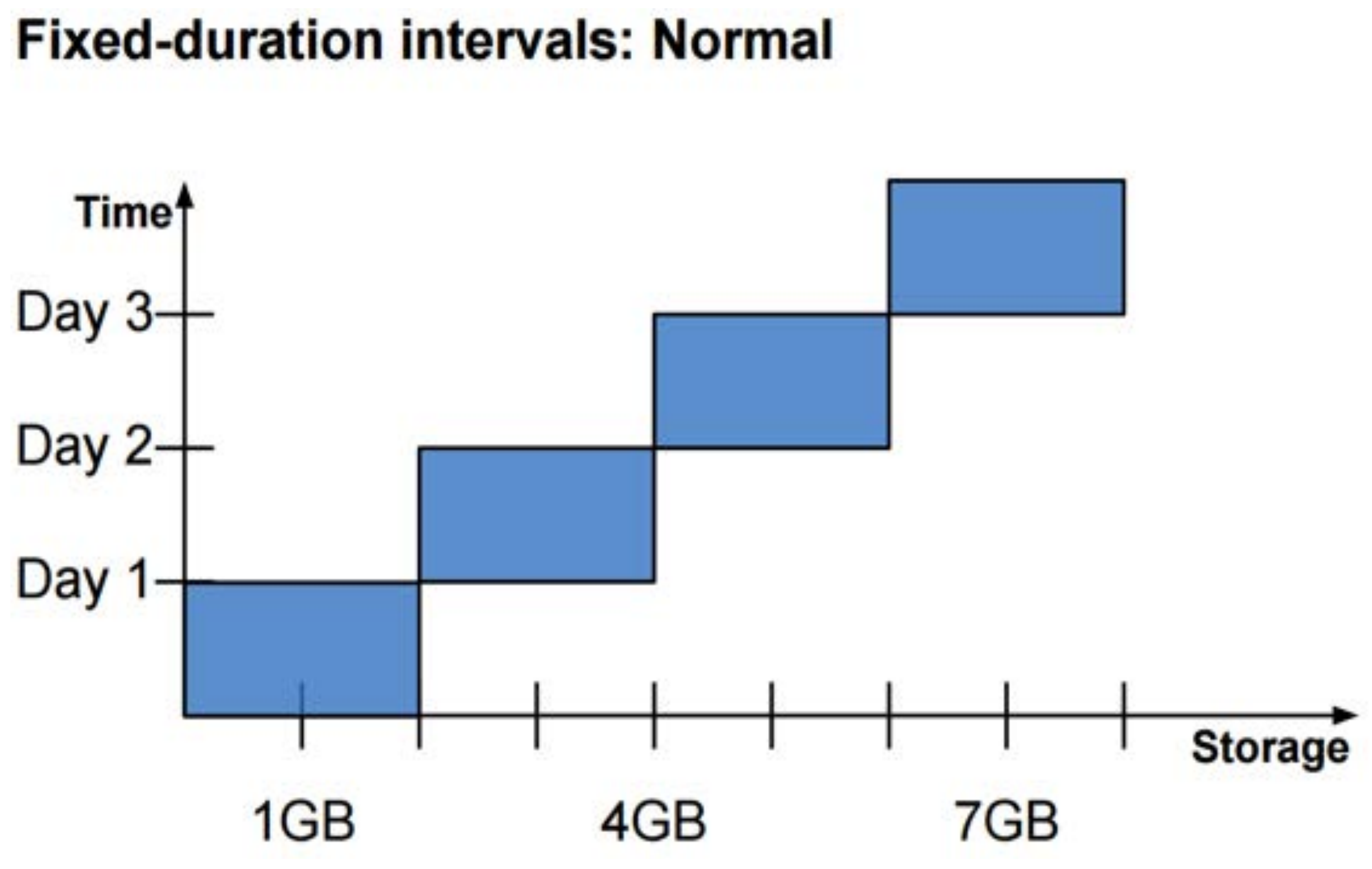
(for both scaling up & out)

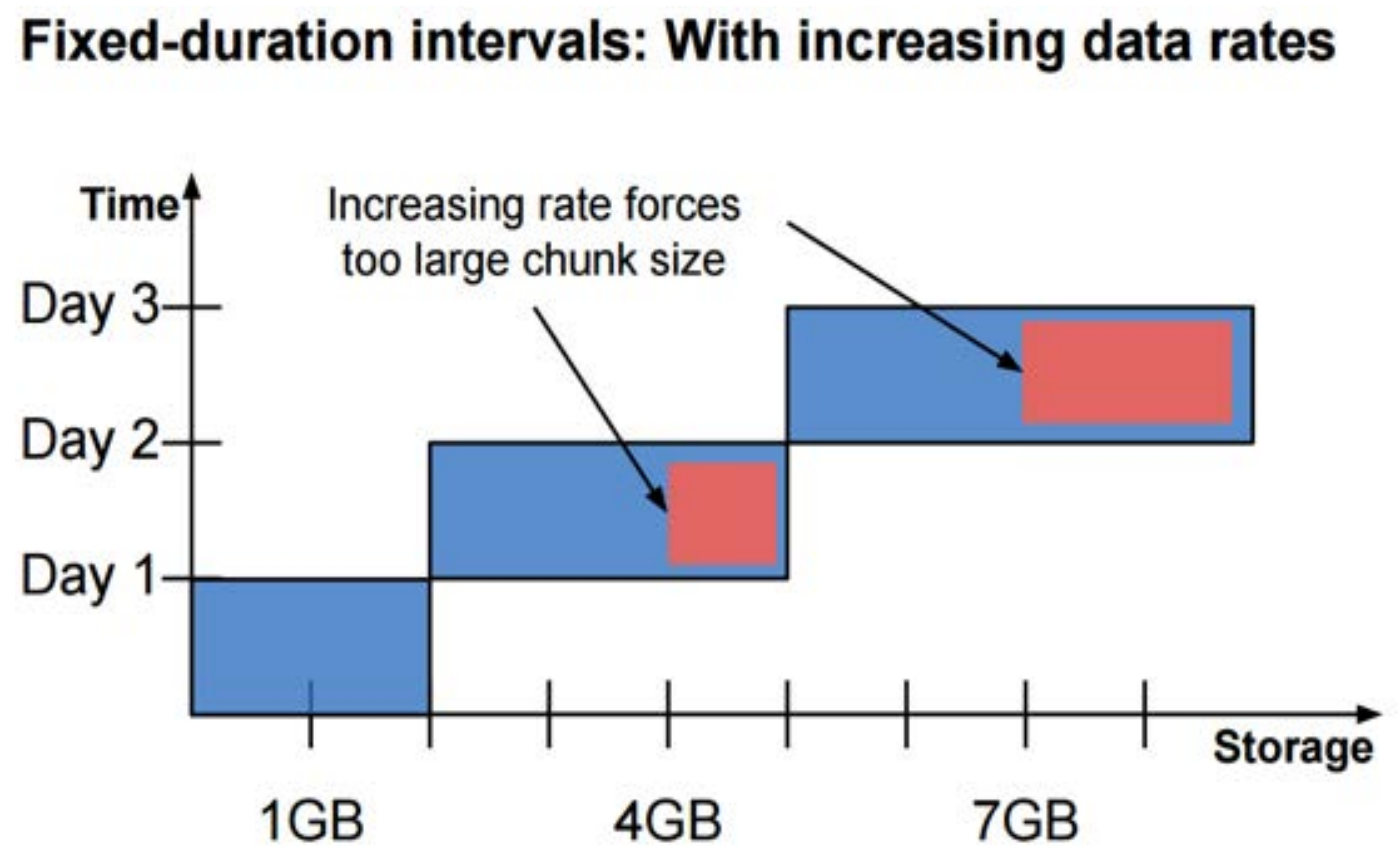


How EXACTLY do we partition by time?

Static, fixed duration?

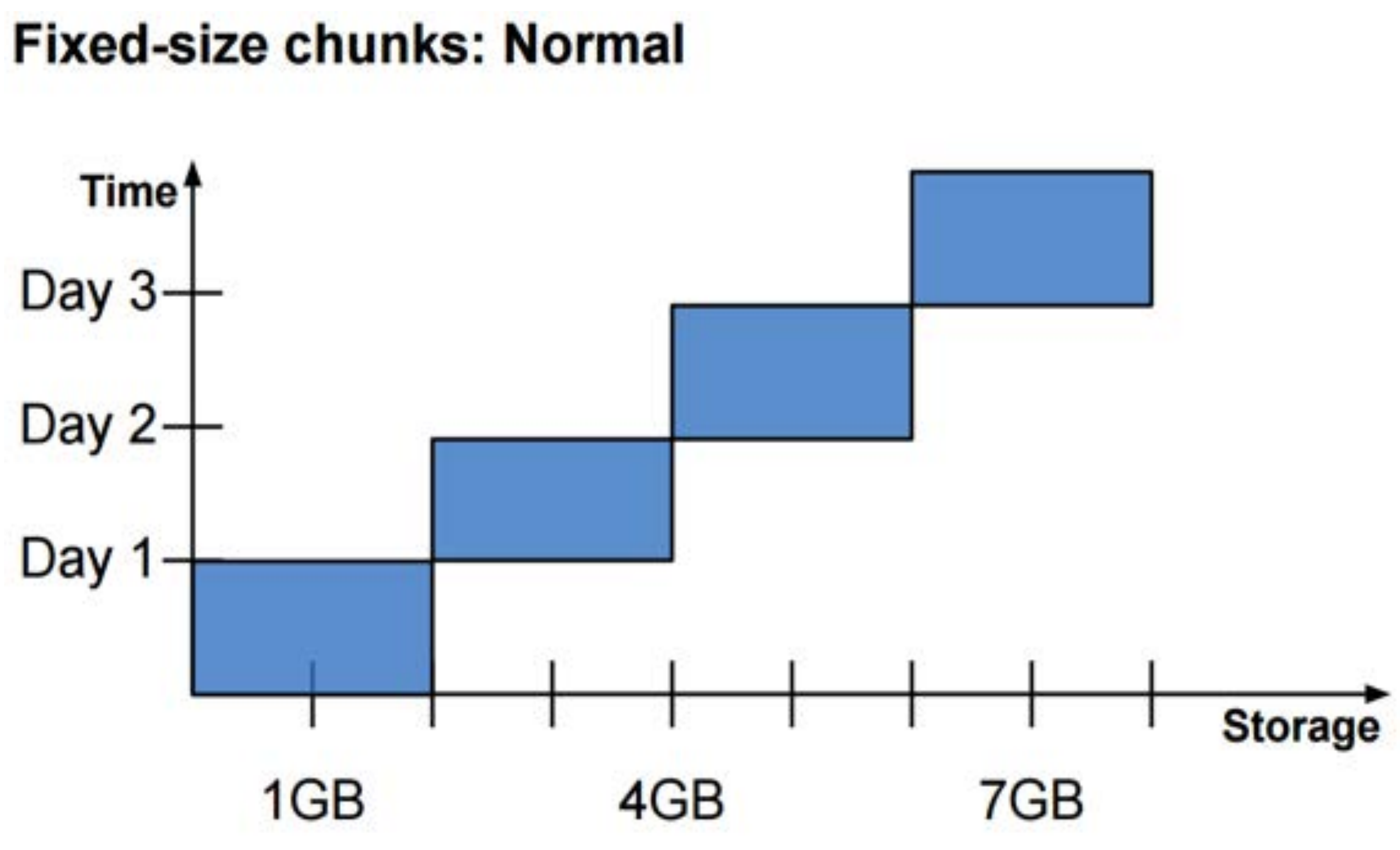
- Insufficient: Data volumes can change

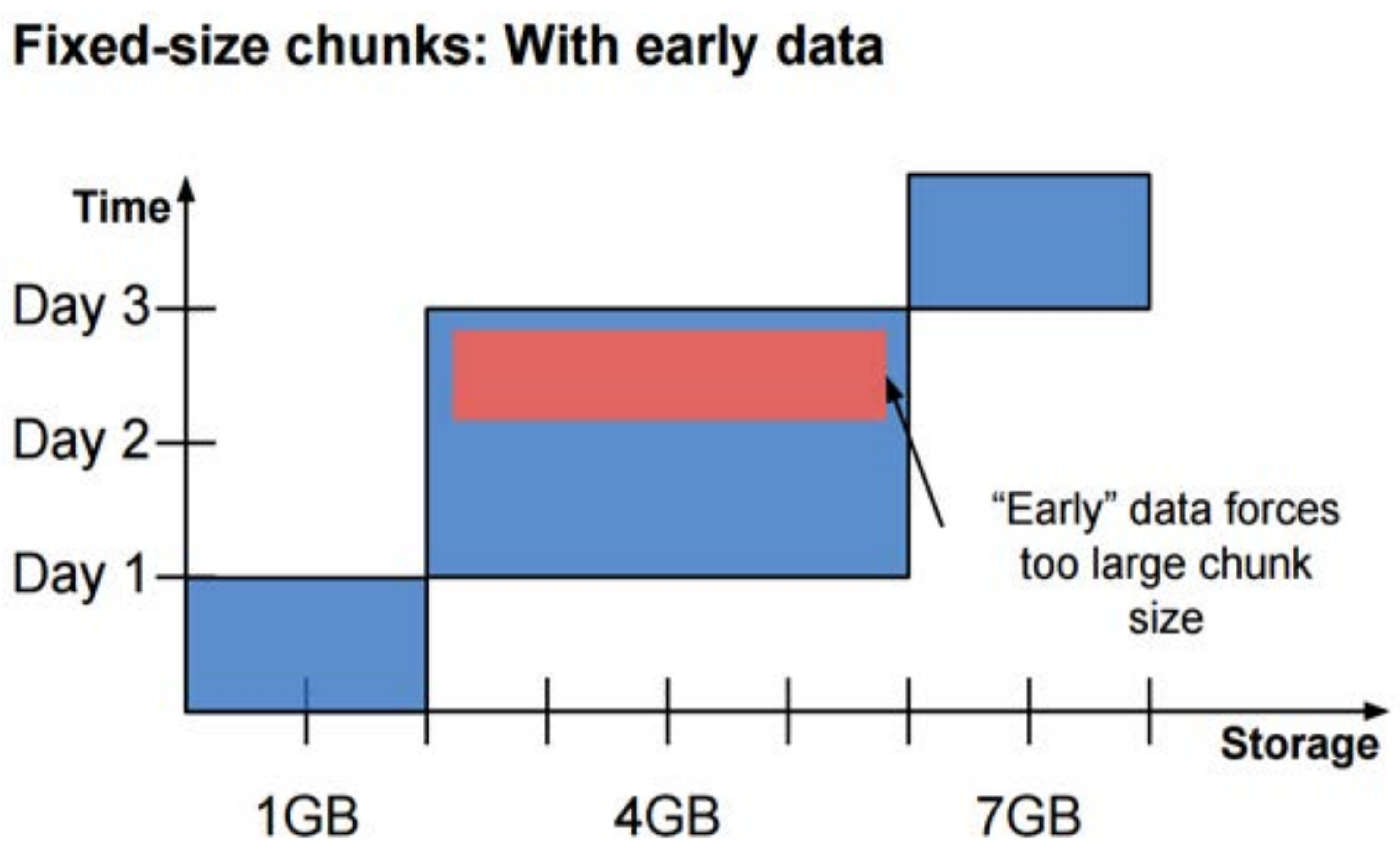




Fixed target size?

- Early data can create too long intervals
- Bulk inserts expensive



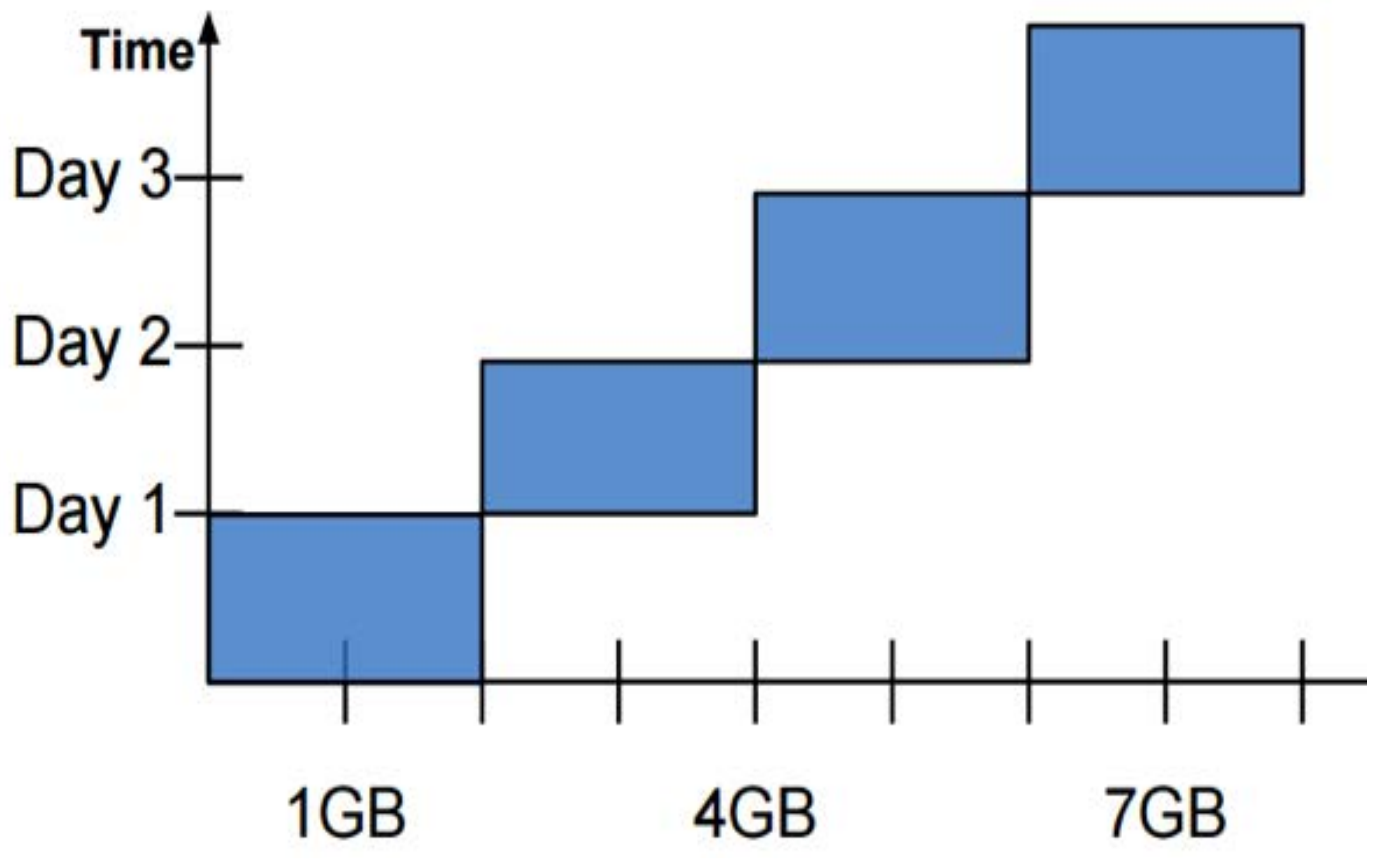


Adaptive time/space partitioning benefits

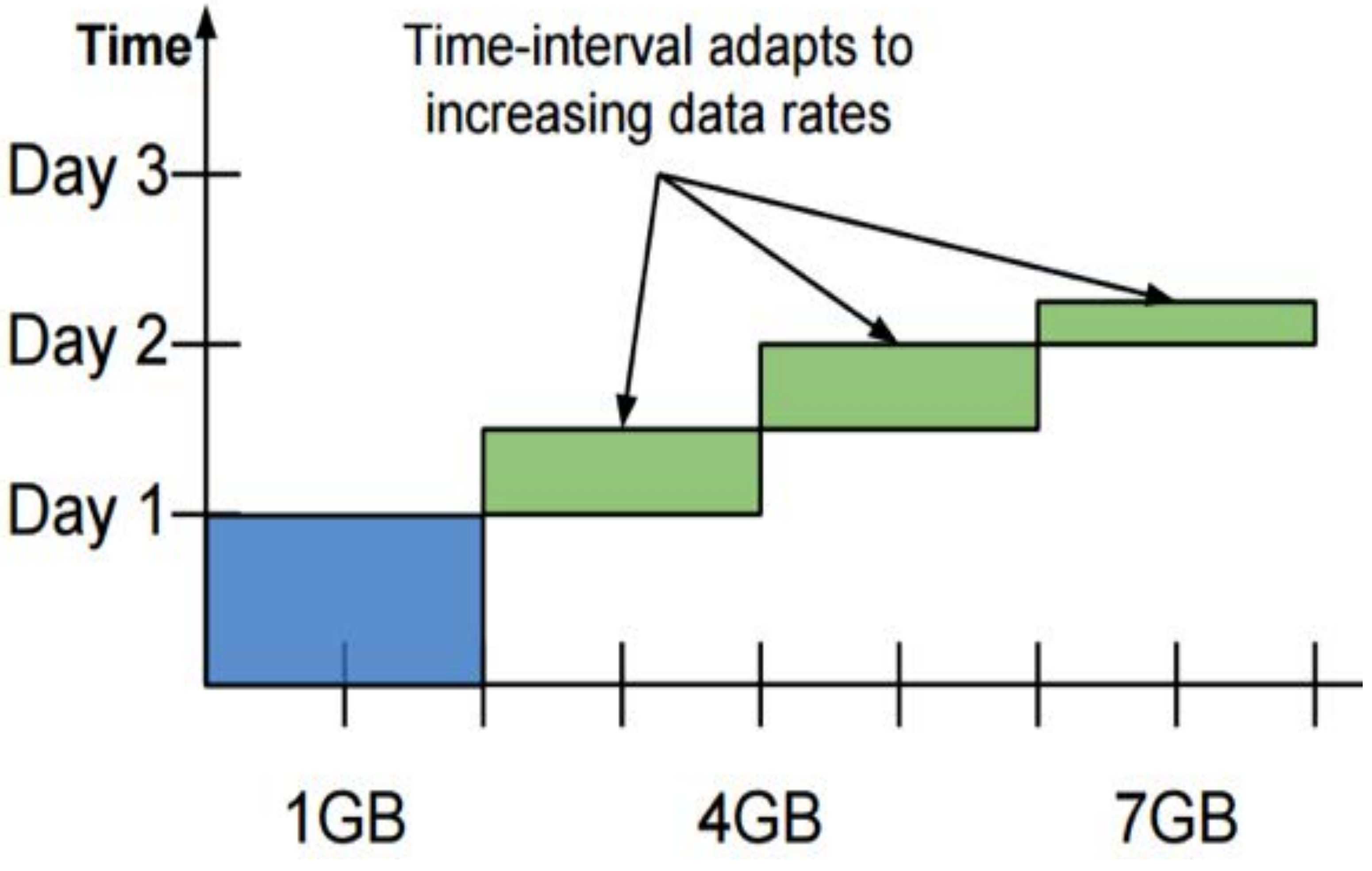
New approach: Adaptive intervals

- Partitions created with fixed time interval, but interval adapts to changes in data volumes

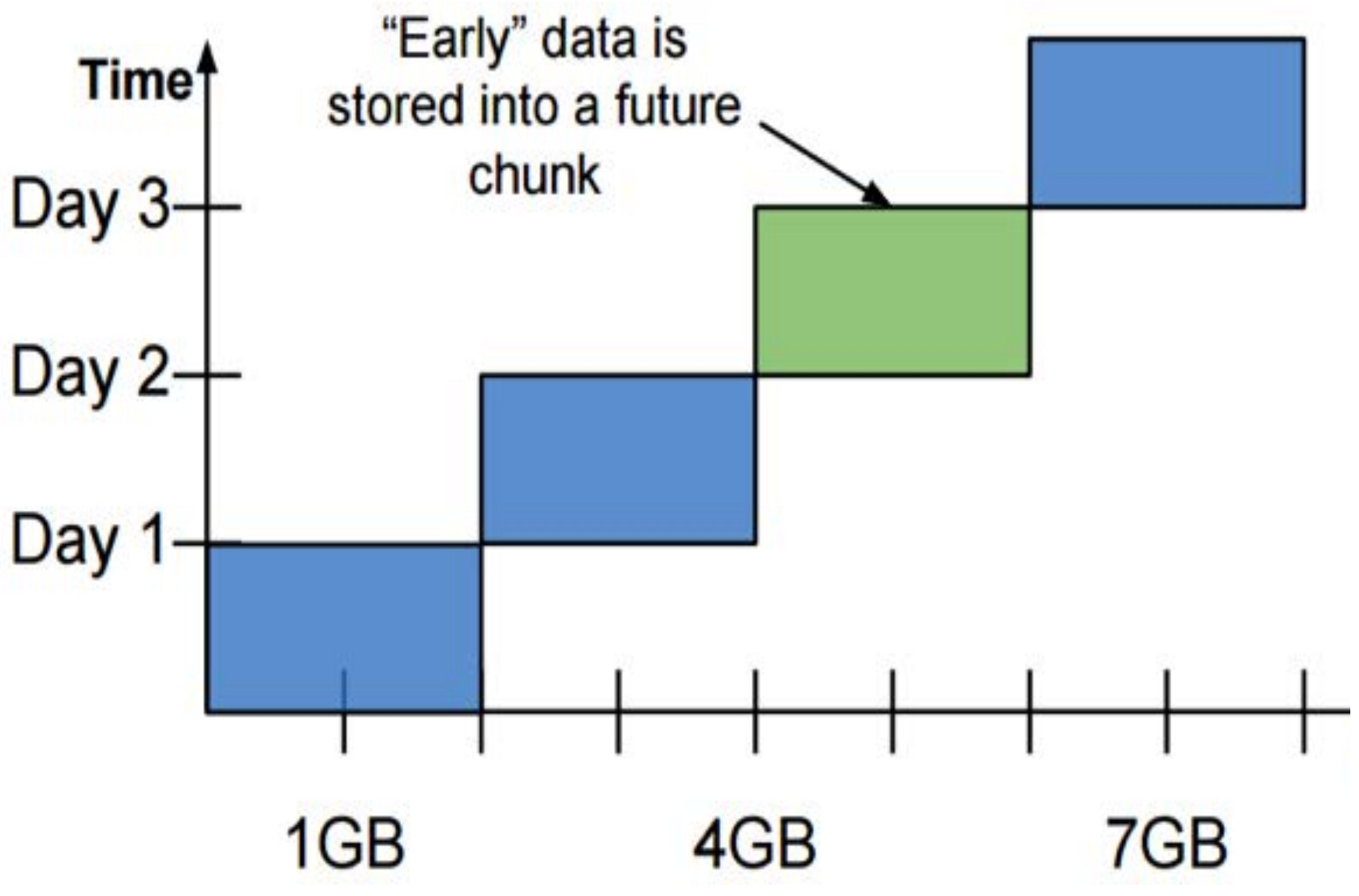
Adaptive chunks: Normal



Adaptive chunks: With increasing data rates



Adaptive chunks: With early data



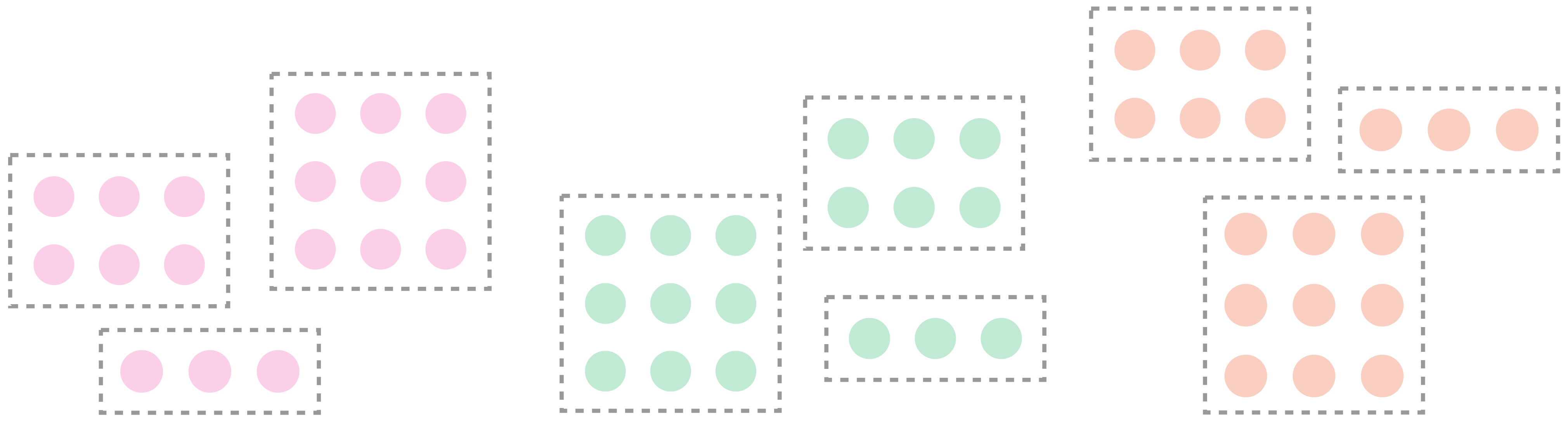
Adaptive time/space partitioning benefits

New approach: Adaptive intervals

- Partitions created with fixed time interval, but interval adapts to changes in data volumes

1. **Partitions are "right sized":**
   Recent (hot) partitions fit in memory

2. **Efficient retention policies:**
   Drop chunks, don't delete rows ⇒ avoids vacuuming



Adaptive time/space partitioning benefits

Common mechanism for scaling up & out

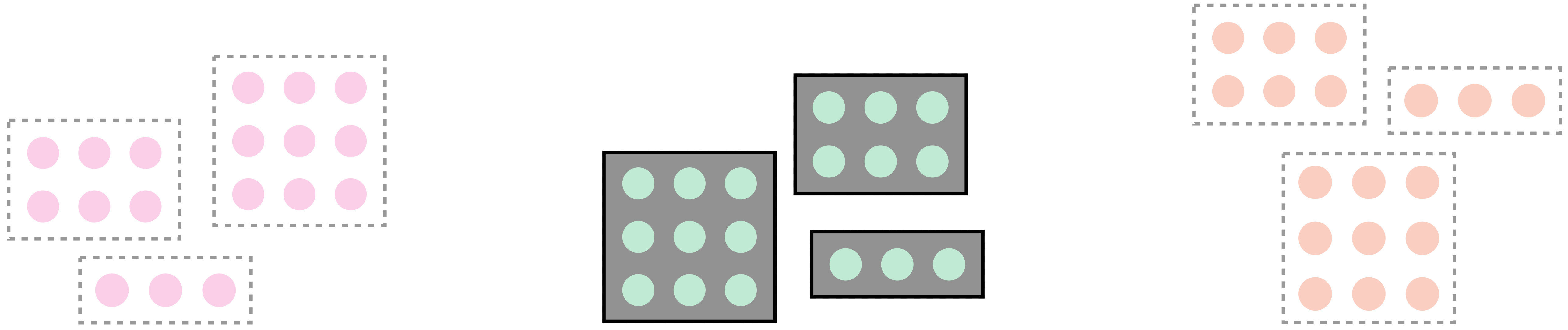
- **Partitions spread across servers**
- **No centralized txn manager or special front-end**
  - Any node can handle any INSERT or QUERY
  - Inserts are routed/sub-batched to appropriate servers
  - Partition-aware query optimizations

Partition-aware Query Optimization

Common mechanism for scaling up & out

- Avoid querying chunks via **constraint exclusion analysis**

SELECT time, temp FROM data
    WHERE time > now() - interval ‘7 days’
        AND device_id = ‘12345’

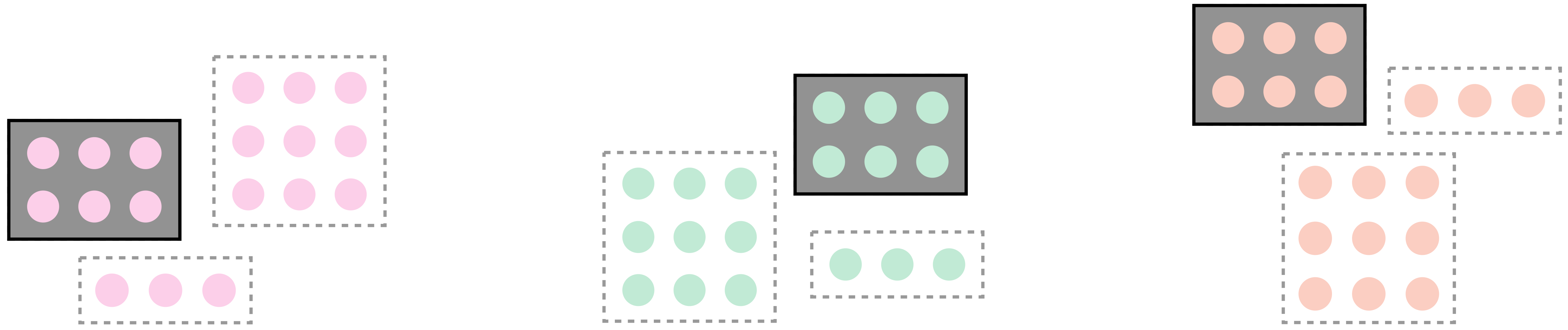


Partition-aware Query Optimization

Common mechanism for scaling up & out

- Avoid querying chunks via **constraint exclusion analysis**

SELECT time, device_id, temp FROM data
WHERE **time > now() - interval ‘24 hours’**

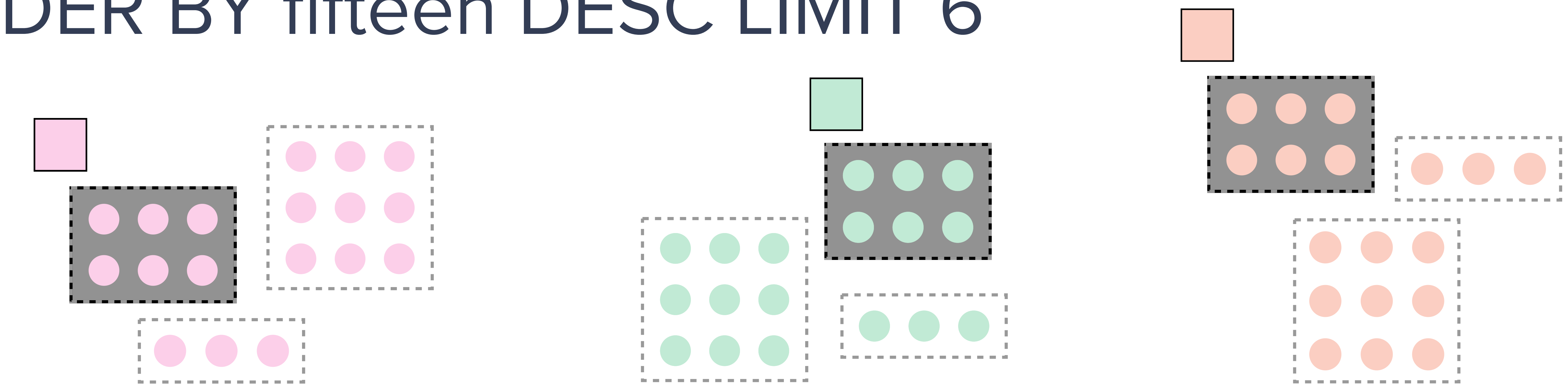


Partition-aware Query Optimization

Common mechanism for scaling up & out

- Efficient **merge appends** of time aggregates across partitions

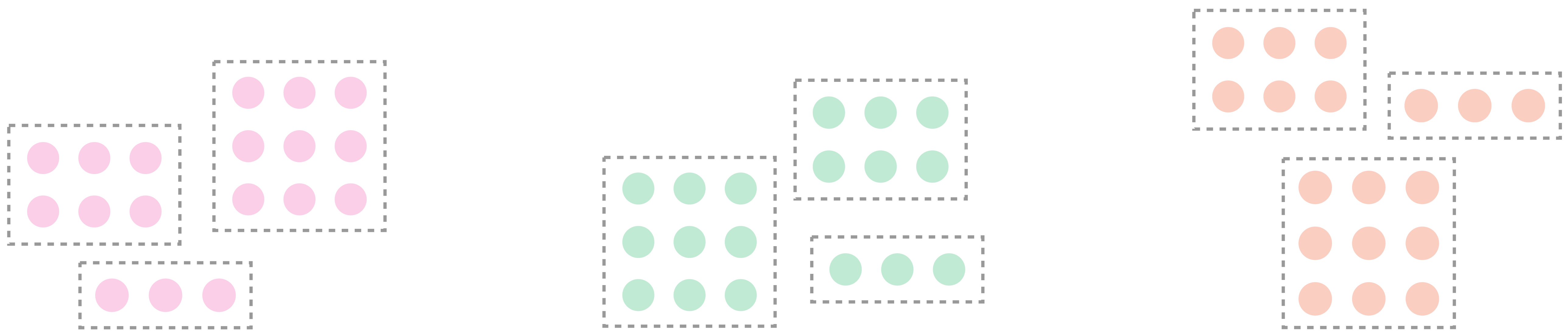
SELECT time_bucket(‘15 minute’, time) fifteen, AVG(temp) FROM data
WHERE firmware = “2.3.1” AND wifi_quality < 25
GROUP BY fifteen
ORDER BY fifteen DESC LIMIT 6



Partition-aware Query Optimization

Common mechanism for scaling up & out

- Efficient **merge appends** of time aggregates across partitions
- Perform **partial aggregations** on distributed data
- Avoid full scans for **last K records of distinct items**



TIMESCALE

SQL made **scalable** for **time-series** data

Packaged as a PostgreSQL extension

Full SQL, Fast ingest, Complex queries, Reliable

Easy to Use

- Supports full SQL
- Connects with any client or tool that speaks PostgreSQL

Scalable

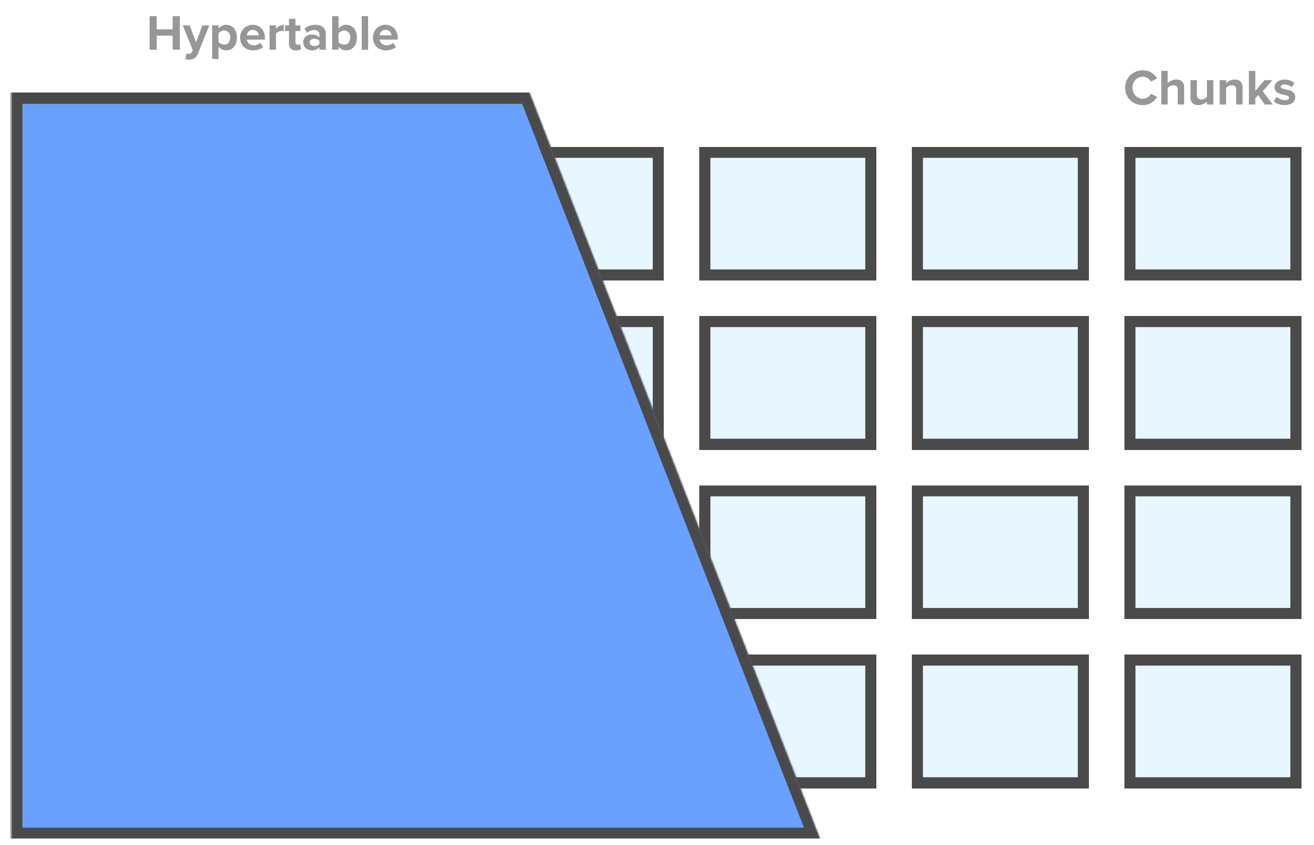
- High write rates
- Time-oriented features and optimizations
- Fast complex queries

Reliable

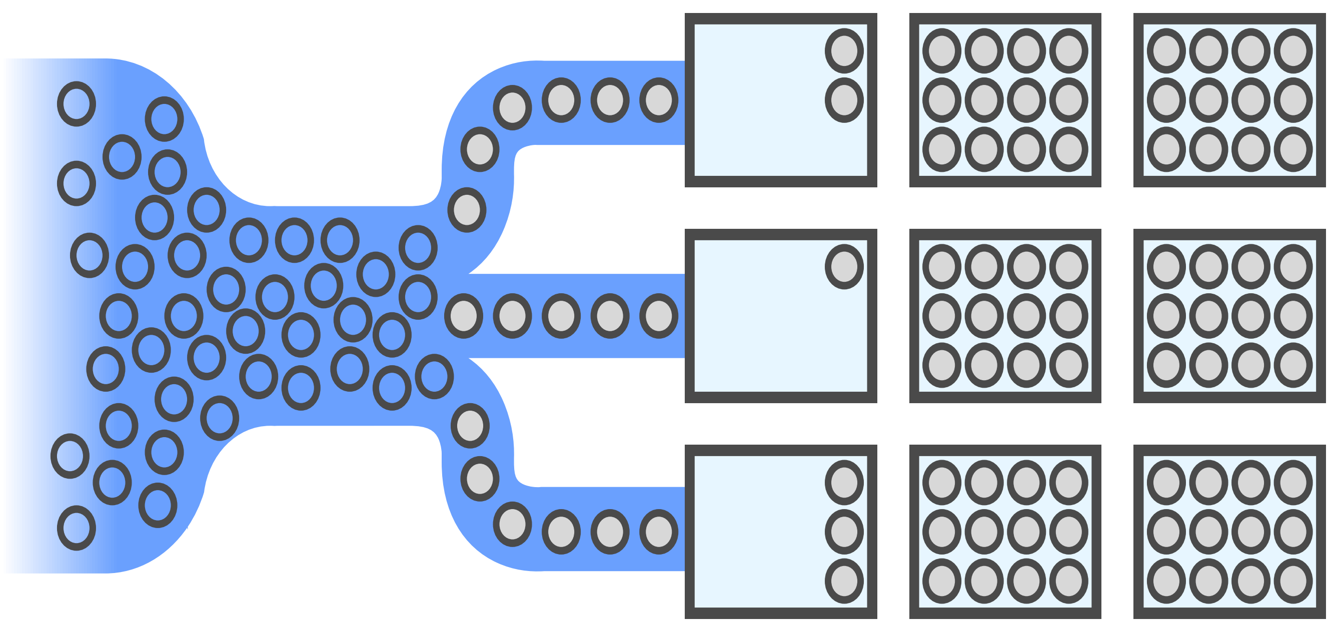
- Engineered up from PostgreSQL
- Inherits 20+ years of reliability and tooling

Familiar SQL interface

The hyper table abstraction



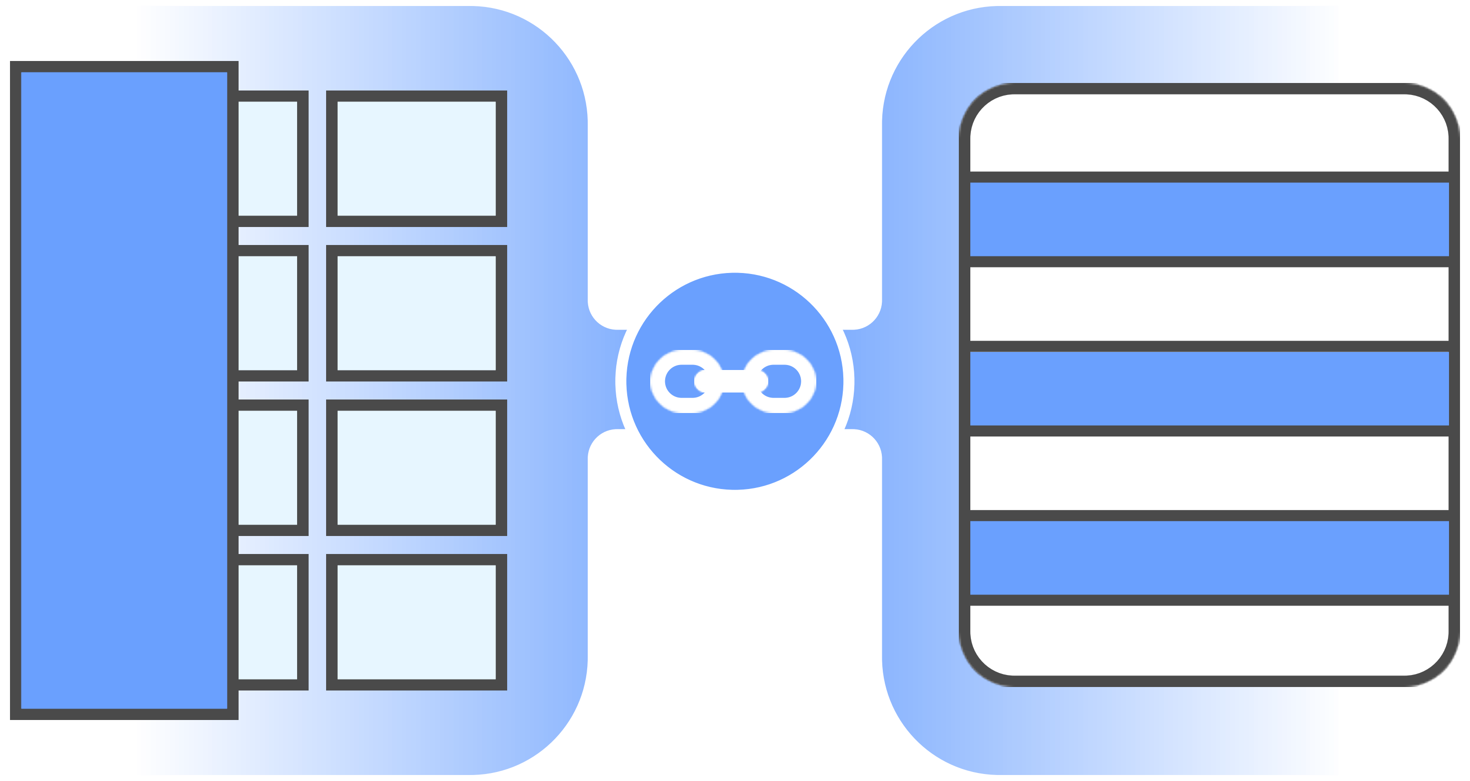
- **Illusion of a single table**
- **SELECT against a single table**
  - Distributed query optimizations across partitions



- **INSERT row / batch into single table**
  - Rows / sub-batches inserted into proper partitions
- **Engine automatically closes/creates partitions**
  - Based on both time intervals and table size

Familiar SQL interface

Avoid data silos via SQL JOINs



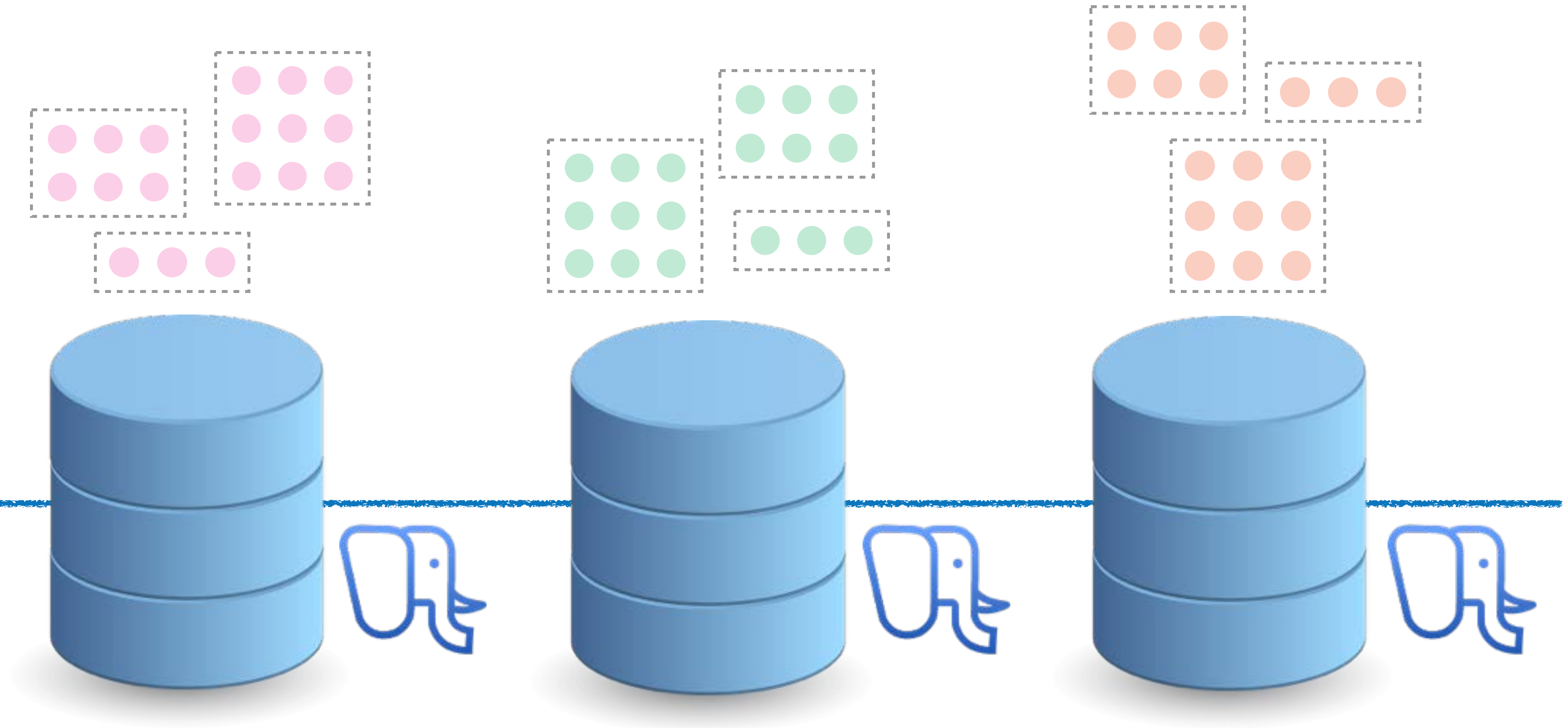
- **Typical time-series DB approaches today:**
  - Denormalize data: Inefficient, expensive to update, operationally difficult
  - Maintain separate relational DB: Application pain
- **TimescaleDB enables easy JOINs**
  - Against relational tables stored either within DB or externally (via foreign data wrapper)
  - Within DB, data fetched from one node or materialized across cluster

Familiar management

Engineered up from **PostgreSQL**

Connect to and query it
like Postgres

Manage it
like Postgres



Familiar management

Looks/feels/speaks **PostgreSQL**

Administration

- Replication (hot standby)
- Checkpointing and backup
- Fine-grain access control

Connectors!

ODBC, JDBC, Postgres

tableau

STATSD

pentaho

kafka

Grafana

Familiar management

Reuse & improve **PostgreSQL** mechanisms

- **Implementation details**
  - Partitions stored as "child" Postgres tables of parent hypertable
  - Secondary indexes are local to each partition (table)
- **Query improvements**
  - Better constrained exclusions avoid querying children
  - New time/partition-aware query optimizations
  - New time-oriented features
- **Insert improvements**
  - Adaptive auto-creation/closing of partitions
  - More efficient insert path (both single row and batch)

Familiar management

Creating/migrating is easy

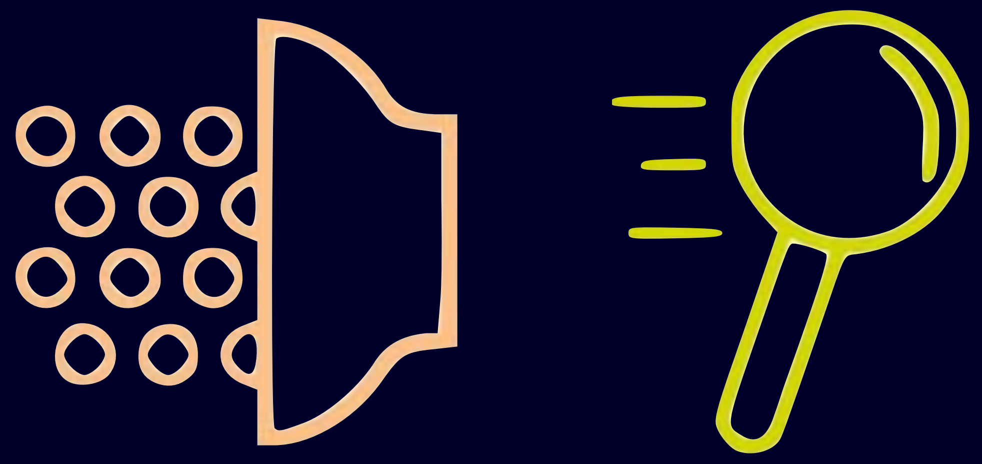
```
$ psql
psql (9.6.2)
Type "help" for help.

tsdb=#  CREATE TABLE data (
            time TIMESTAMP WITH TIME ZONE NOT NULL,
            device_id TEXT NOT NULL,
            temperature NUMERIC NULL,
            humidity NUMERIC NULL
        );

tsdb=#  SELECT create_hypertable ('data', 'time', 'device_id', 16);

tsdb=#  INSERT INTO data (SELECT * FROM old_data);
```

Performance benefits



Performance benefits

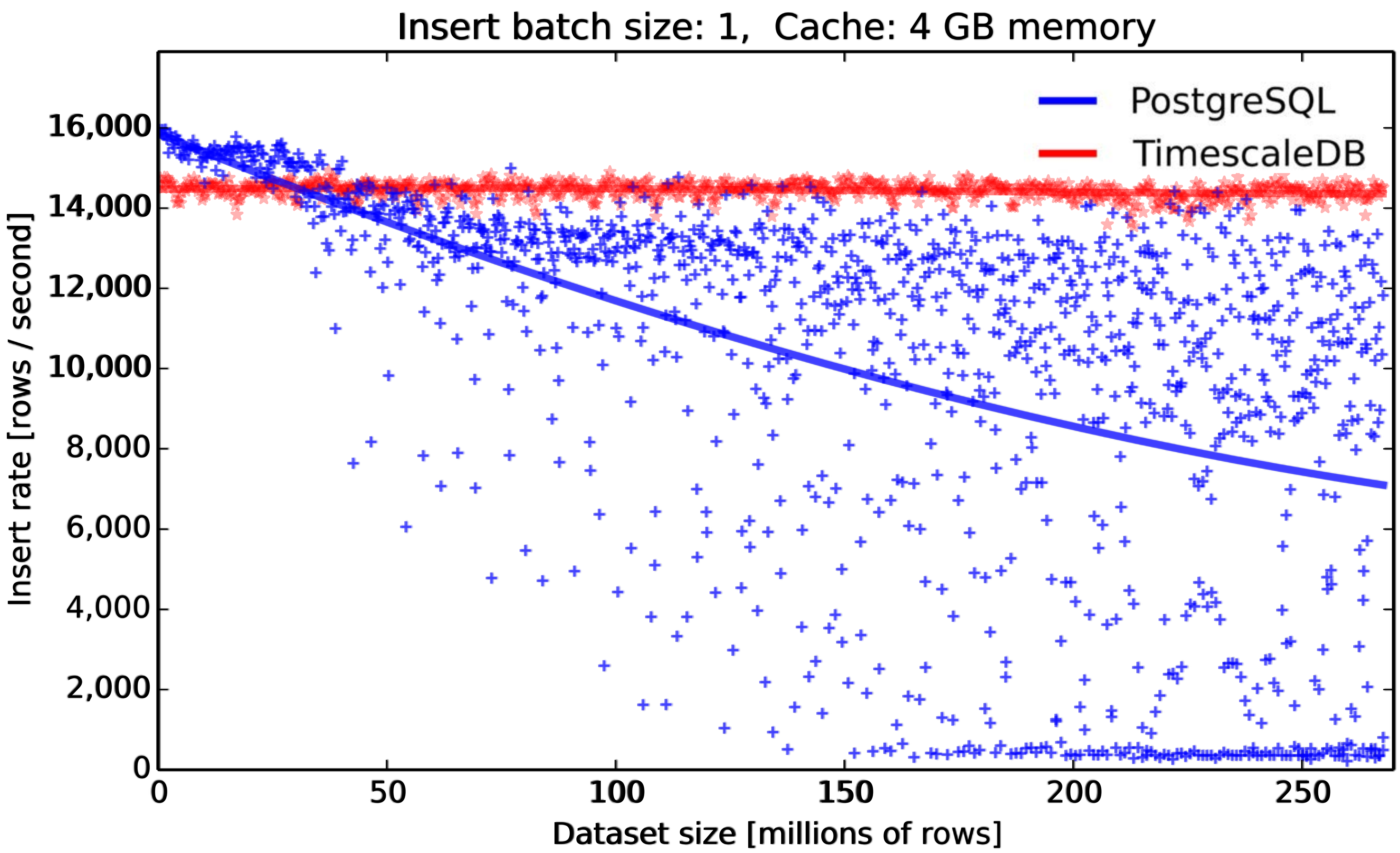
Single server

- Carefully sizing chunks
- Reduce amount of data read
  (e.g., merge appends, GROUP BYs)
- Parallelize across multiple chunks, disks

Clusters

- Reduce latency by parallelizing queries
- Reduce network traffic
  (e.g., aggregation pushdown, localizing GROUP BYs)

Single-node INSERT scalability



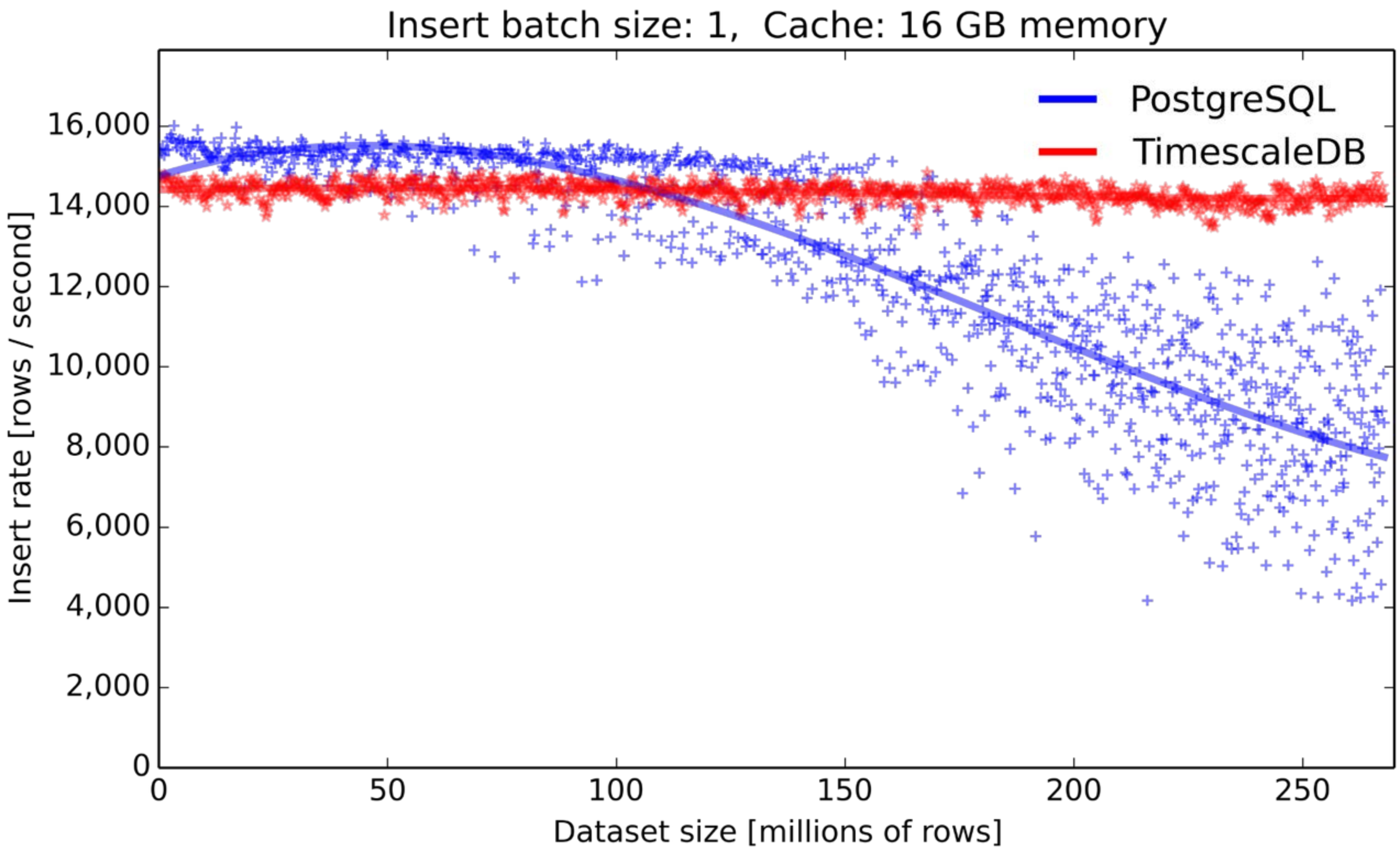
144K metrics/s

14.4K inserts/s

Postgres 9.6.2 on Azure standard DS4 v2 (8 cores), SSD (premium LRS storage)
Each row has 12 columns (1 timestamp, indexed 1 host ID, 10 metrics)

TIMESCALE

Single-node INSERT scalability



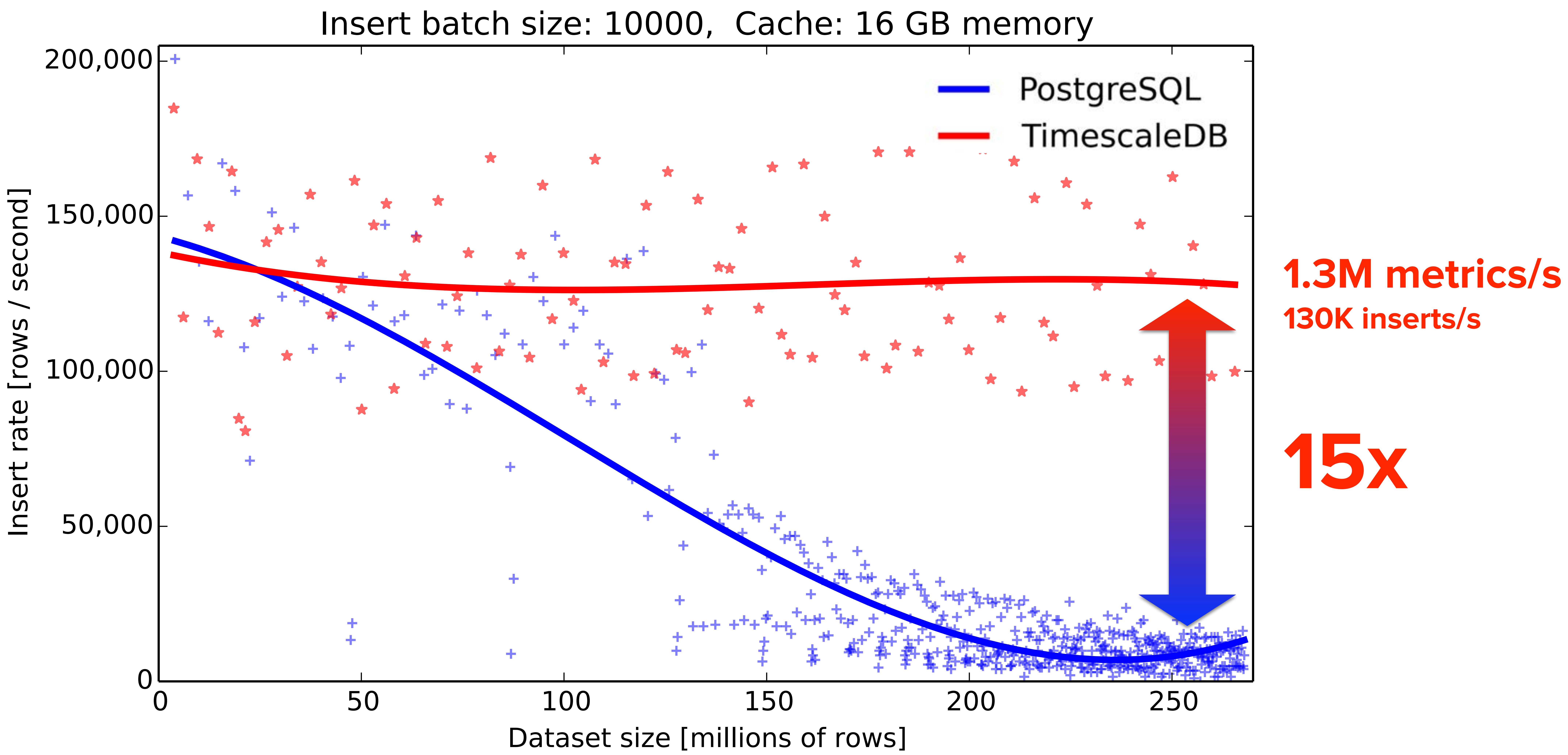
144K metrics/s

14.4K inserts/s

Postgres 9.6.2 on Azure standard DS4 v2 (8 cores), SSD (premium LRS storage)
Each row has 12 columns (1 timestamp, indexed 1 host ID, 10 metrics)

TIMESCALE

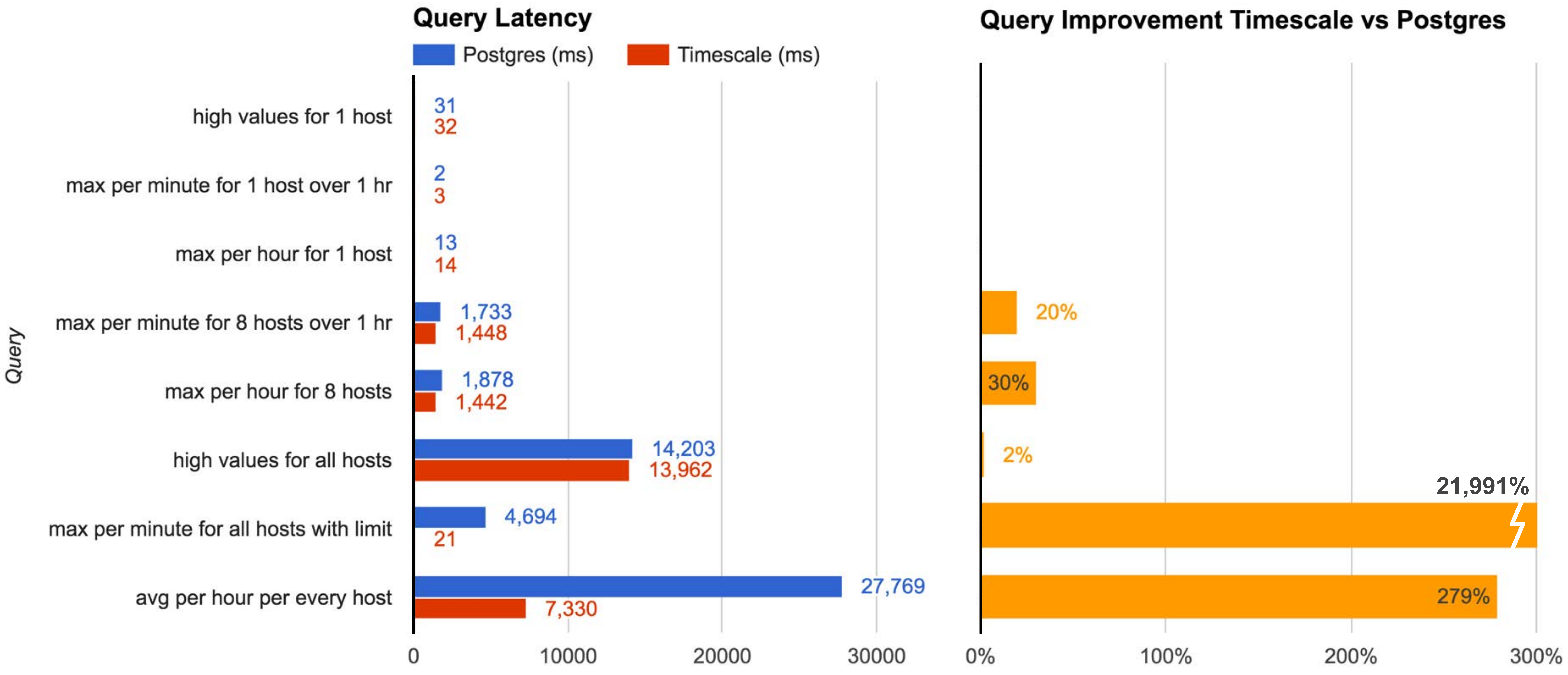
Single-node INSERT scalability



Postgres 9.6.2 on Azure standard DS4 v2 (8 cores), SSD (premium LRS storage)
Each row has 12 columns (1 timestamp, indexed 1 host ID, 10 metrics)

TIMESCALE

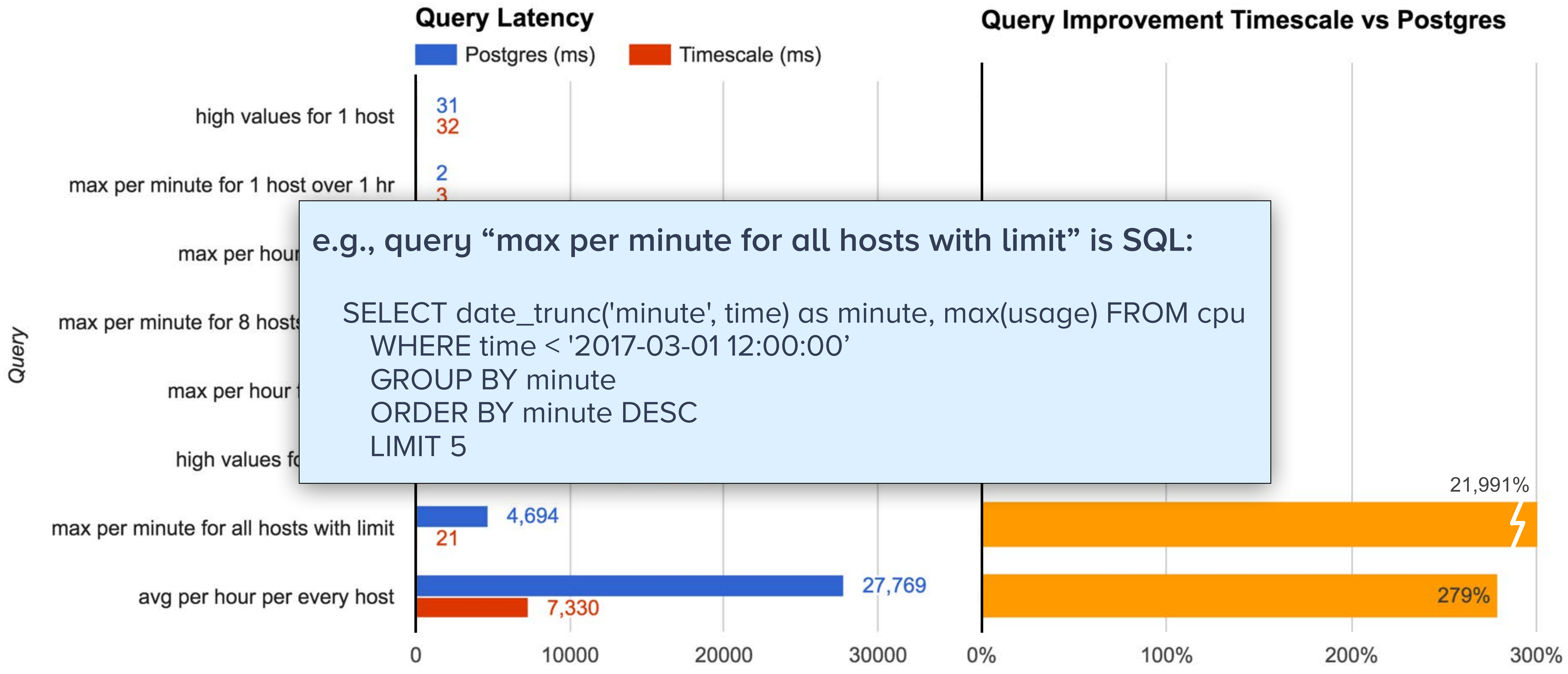
Single-node QUERY performance



Mean results for 2500 query, randomly chosen IDs and times for each query

TIMESCALE

Single-node QUERY performance



e.g., query "max per minute for all hosts with limit" is SQL:

```
SELECT date_trunc('minute', time) as minute, max(usage) FROM cpu
   WHERE time < '2017-03-01 12:00:00'
   GROUP BY minute
   ORDER BY minute DESC
   LIMIT 5
```

Mean results for 2500 query, randomly chosen IDs and times for each query

TIMESCALE

Should NOT use if:

- ✗ Simple read requirements: KV lookups, single-column rollup
- ✗ Heavy compression is priority
- ✗ Very sparse or unstructured data

Should use if:

- ✓ Full SQL: Complex predicates or aggregates, JOINs
- ✓ Rich indexing
- ✓ Mostly structured data
- ✓ Desire reliability, ecosystem, integrations of Postgres

Fork me on GitHub

Open-source release last month

https://github.com/timescale/timescaledb

Apache 2.0 license

Beta release for single-node

Visit us at booth #316

TIMESCALE

Mike Freedman
Co-founder/CTO of @timescaledb. Professor of Computer Science, Princeton University.
Apr 20 · 14 min read

Time-series data: Why (and how) to use a relational database instead of NoSQL

These days, time-series data applications (e.g., data center / server / microservice / container monitoring, sensor / IoT analytics, financial data analysis, etc.) are proliferating.

As a result, time-series databases are in fashion (here are 33 of them). Most of these renounce the trappings of a traditional relational database and adopt what is generally known as a NoSQL model. Usage patterns are similar: a recent survey showed that developers preferred NoSQL to relational databases for time-series data by over 2:1.

Fork me on GitHub

Open-source release last month

https://github.com/timescale/timescaledb

Apache 2.0 license

Beta release for single-node

Visit us at booth #316