Timescale

Blog

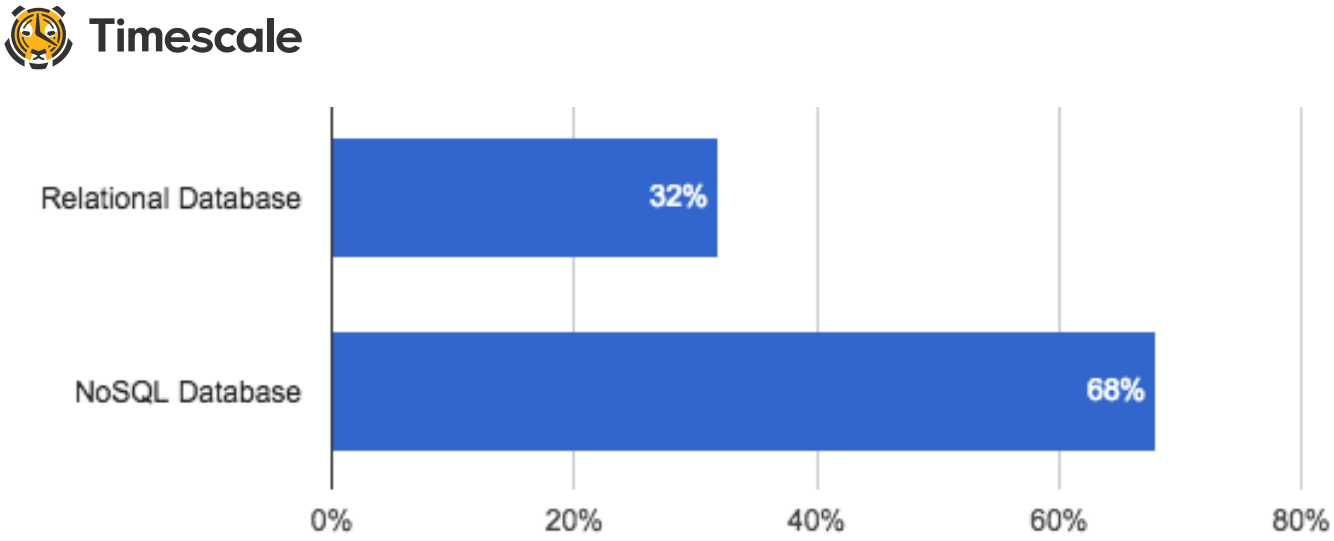
Home
Company
Engineering
Product

20 April 2017 / Engineering

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.



Relational databases include: MySQL, MariaDB Server, PostgreSQL. **NoSQL databases include:** Elastic, InfluxDB, MongoDB, Cassandra, Couchbase, Graphite, Prometheus, ClickHouse, OpenTSDB, DalmatinerDB, KairosDB, RiakTS. **Source: Percona.**

Typically, the reason for adopting NoSQL time-series databases comes down to scale. While relational databases have many useful features that most NoSQL databases do not (robust secondary index support; complex predicates; a rich query language; JOINs, etc), they are difficult to scale.

And because time-series data piles up very quickly, many developers believe relational databases are ill-suited for it.

We take a different, somewhat heretical stance: relational databases can be quite powerful for time-series data. One just needs to solve the scaling problem. That is what we do in TimescaleDB.

When we announced TimescaleDB two weeks ago, we received a lot of positive feedback from the community. But we also heard from skeptics, who found it hard to believe that one should (or could) build a scalable time-series database on a relational database (in our case, PostgreSQL).

There are two separate ways to think about scaling: **scaling up** so that a single machine can store more data, and **scaling out** so that data can be stored across multiple machines.

Why are both important? The most common approach to scaling out across a cluster of *N* servers is to partition, or shard, a dataset into *N* partitions. If each server is limited in its throughput or performance (i.e., unable to scale up), then the overall cluster throughput is greatly reduced.

This post discusses **scaling up**. (A **scaling-out** post will be published on a later date.)

In particular, this post explains:

- Why relational databases do not normally scale up well

Timescale

How time-series data is unique, how one can leverage those differences to overcome the scaling problem, and some performance results

Our motivations are twofold: **for anyone facing similar problems**, to share what we've learned; and **for those considering using TimescaleDB for time-series data** (including the skeptics!), to explain some of our design decisions.

Why databases do not normally scale up well: Swapping in/out of memory is expensive

A common problem with scaling database performance on a single machine is the significant cost/performance trade-off between memory and disk. While memory is faster than disk, it is much more expensive: about 20x costlier than solid-state storage like Flash, 100x more expensive than hard drives. Eventually, our entire dataset will not fit in memory, which is why we'll need to write our data and indexes to disk.

This is an old, common problem for relational databases. Under most relational databases, a table is stored as a collection of fixed-size pages of data (e.g., 8KB pages in PostgreSQL), on top of which the system builds data structures (such as B-trees) to index the data. With an index, a query can quickly find a row with a specified ID (e.g., bank account number) without scanning the entire table or "walking" the table in some sorted order.

Now, if the working set of data and indexes is small, we can keep it in memory.

But if the data is sufficiently large that we can't fit all (similarly fixed-size) pages of our B-tree in memory, then updating a random part of the tree can involve significant disk I/O as we read pages from disk into memory, modify in memory, and then write back out to disk (when evicted to make room for other B-tree pages). And a relational database like PostgreSQL keeps a B-tree (or other data structure) for *each* table index, in order for values in that index to be found efficiently. So, the problem compounds as you index more columns.

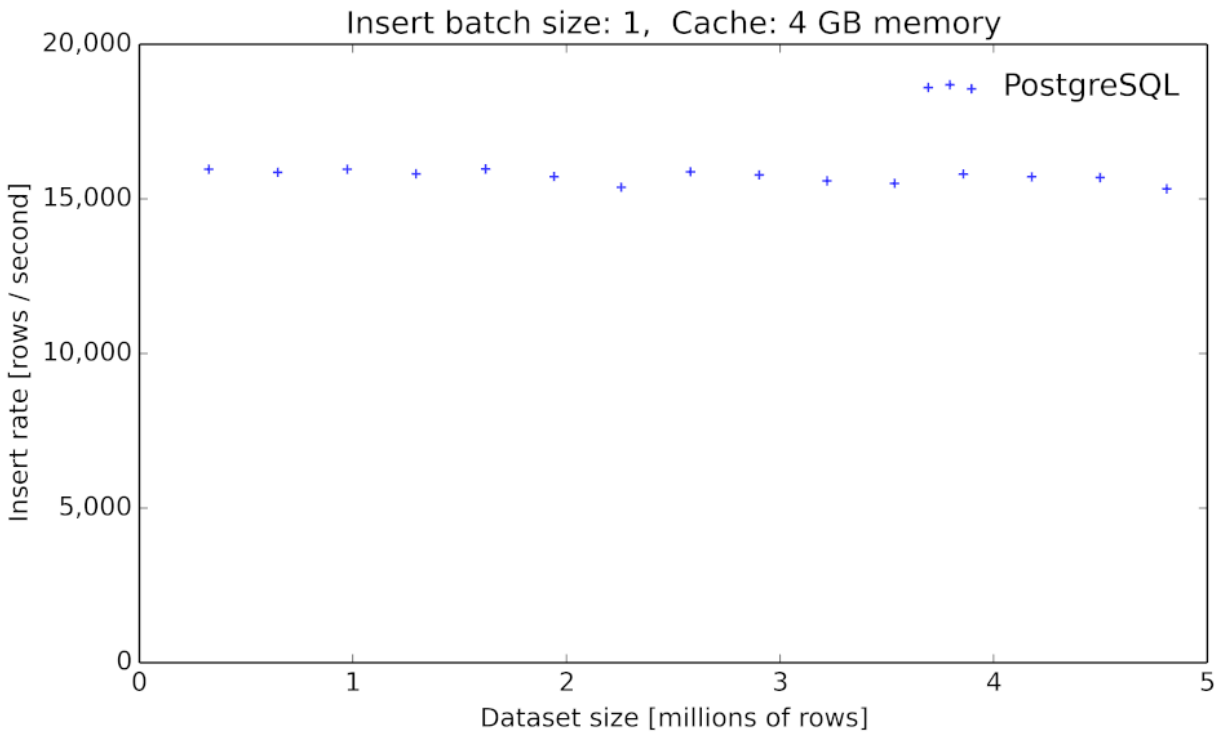
In fact, because the database only accesses the disk in page-sized boundaries, even seemingly small updates can cause these swaps to occur: To change one cell, the database may need to swap out an existing 8KB page and write it back to disk, then read in the new page before modifying it.

But why not use smaller- or variable-sized pages? There are two good reasons: minimizing disk fragmentation, and (in case of a spinning hard disk) minimizing the overhead of the "seek time" (usually 5–10ms) required in physically moving the disk head to a new location.

Timescale

the page, then write the updated 8KB page back to a new disk block.

The cost of swapping in and out of memory can be seen in this performance graph from PostgreSQL, where insert throughput plunges with table size and increases in variance (depending on whether requests hit in memory or require (potentially multiple) fetches from disk).



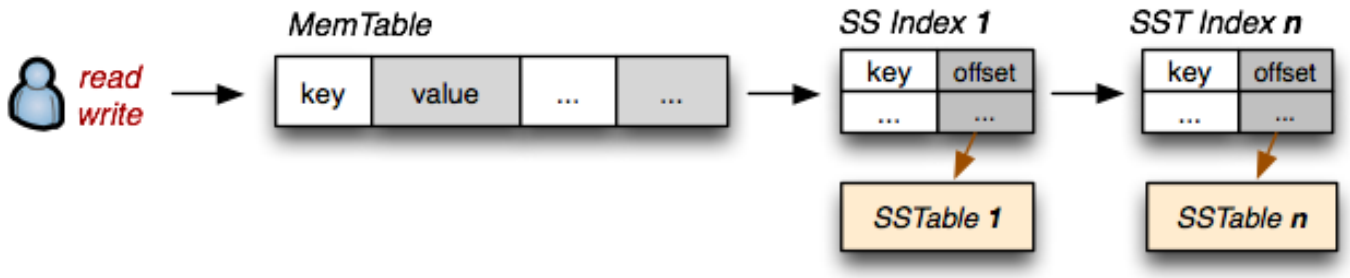
Insert throughput as a function of table size for PostgreSQL 9.6.2, running with 10 workers on an Azure standard DS4 v2 (8 core) machine with SSD-based (premium LRS) storage. Clients insert individual rows into the database (each of which has 12 columns: a timestamp, an indexed randomly-chosen primary id, and 10 additional numerical metrics). The PostgreSQL rate starts over 15K inserts/second, but then begins to drop significantly after 50M rows and begins to experience very high variance (including periods of only 100s of inserts/sec).

Enter NoSQL databases with Log-Structured Merge Trees (and new problems)

About a decade ago, we started seeing a number of "NoSQL" storage systems address this problem via Log-structured merge (LSM) trees, which reduce the cost of making small writes by only performing larger

Timescale

into pages and write them as a single batch to disk. In particular, all writes in an LSM tree are performed to a sorted table maintained *in memory*, which is then flushed to disk as an immutable batch when of sufficient size (as a "sorted string table", or SSTable). This reduces the cost of making small writes.



In an LSM tree, all updates are first written a sorted table in memory, and then flushed to disk as an immutable batch, stored as an SSTable, which is often indexed in memory. Source: igvita.com

This architecture—which has been adopted by many "NoSQL" databases like LevelDB, Google BigTable, Cassandra, MongoDB (WiredTiger), and InfluxDB—may seem great at first. Yet it introduces other tradeoffs: higher memory requirements and poor secondary index support.

Higher-memory requirements: Unlike in a B-tree, in an LSM tree there is no single ordering: no global index to give us a sorted order over all keys. Consequently, looking up a value for a key gets more complex: first, check the memory table for the latest version of the key; otherwise, look to (potentially many) on-disk tables to find the latest value associated with that key. To avoid excessive disk I/O (and if the values themselves are large, such as the webpage content stored in Google's BigTable), indexes for all SSTables may be kept entirely in memory, which in turn increases memory requirements.

Poor secondary index support: Given that they lack any global sorted order, LSM trees do not naturally support secondary indexes. Various systems have added some additional support, such as by duplicating the data in a different order. Or, they emulate support for richer predicates by building their primary key as the concatenation of multiple values. Yet this approach comes with the cost of requiring a larger scan among these keys at query time, thus supporting only items with a limited cardinality (e.g., discrete values, not numeric ones).

There is a better approach to this problem. Let's start by better understanding time-series data.

Time-series data is different

Let's take a step back, and look at the original problem that relational databases were designed to solve. Starting from IBM's seminal System R in the mid-1970s, relational databases were employed for what

Timescale

a bank transfer: a user debits money from one account and credits another. This corresponds to updates to two rows (or even just two cells) of a database table. Because bank transfers can occur between any two accounts, the two rows that are modified are somewhat randomly distributed over the table.

Time-series data arises from many different settings: industrial machines; transportation and logistics; DevOps, datacenter, and server monitoring; and financial applications.

Now let's consider a few examples of time-series workloads:

DevOps/server/container monitoring. The system typically collects metrics about different servers or containers: CPU usage, free/used memory, network tx/rx, disk IOPS, etc. Each set of metrics is associated with a timestamp, unique server name/ID, and a set of tags that describe an attribute of what is being collected.

IoT sensor data. Each IoT device may report multiple sensor readings for each time period. As an example, for environmental and air quality monitoring this could include: temperature, humidity, barometric pressure, sound levels, measurements of nitrogen dioxide, carbon monoxide, particulate matter, etc. Each set of readings is associated with a timestamp and unique device ID, and may contain other metadata.

Timescale

transactions, which would include a unique account ID, timestamp, transaction amount, as well as any other metadata. (Note that this data is different than the OLTP example above: here we are recording every transaction, while the OLTP system was just reflecting the current state of the system.)

Fleet/asset management. Data may include a vehicle/asset ID, timestamp, GPS coordinates at that timestamp, and any metadata.

In all of these examples, the datasets are a stream of measurements that involve inserting "new data" into the database, typically to the latest time interval. While it's possible for data to arrive much later than when it was generated/timestamped, either due to network/system delays or because of corrections to update existing data, this is typically the exception, not the norm.

In other words, these two workloads have very different characteristics:

OLTP Writes

Primarily UPDATES

Randomly distributed (over the set of primary keys)

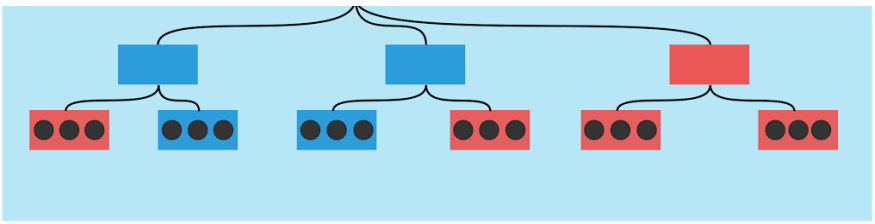
Often transactions across multiple primary keys

Time-series Writes

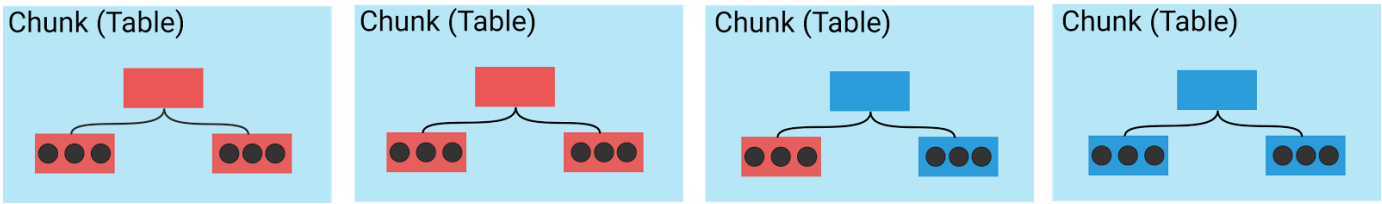
Primarily INSERTs

Primarily to a recent time interval

Primarily associated with both a timestamp and a separate primary key (e.g., server ID, device ID, security/account ID, vehicle/asset ID, etc.)



TimescaleDB hypertable



TimescaleDB stores each chunk in an internal database table, so indexes only grow with the size of each chunk, not the entire hypertable. As inserts are largely to the more recent interval, that one remains in memory, avoiding expensive swaps to disk.

Why does this matter? As we've seen, one can take advantage of these characteristics to solve the scaling-up problem on a relational database.

A new way: Adaptive time/space chunking

When previous approaches tried to avoid small writes to disk, they were trying to address the broader OLTP problem of UPDATEs to random locations. But as we just established, time-series workloads are different: writes are primarily INSERTS (not UPDATES), to a recent time interval (not a random location). In other words, time-series workloads are *append only.*

This is interesting: it means that, if data is sorted by time, we would always be writing towards the "end" of our dataset. Organizing data by time would also allow us to keep the actual working set of database pages rather small, and maintain them in memory. And reads, which we have spent less time discussing, could also benefit: if many read queries are to recent intervals (e.g., for real-time dashboarding), then this data would be already cached in memory.

At first glance, it may seem like indexing on time would give us efficient writes and reads for free. But once we want any other indexes (e.g., another primary key like server/device ID, or any secondary indexes), then this naive approach would revert us back to making random inserts into our B-tree for that index.

There is another way, which we call, "adaptive time/space chunking". This is what we use in TimescaleDB.

Instead of just indexing by time, TimescaleDB builds distinct *tables* by splitting data according to two dimensions: the time interval *and* a primary key (e.g., server/device/asset ID). We refer to these as *chunks* to differentiate them from *partitions*, which are typically defined by splitting the primary key space. Because each of these chunks are stored as a database table itself, and the query planner is aware of the chunk's ranges (in time and keyspace), the query planner can immediately tell to which chunk(s) an

Timescale

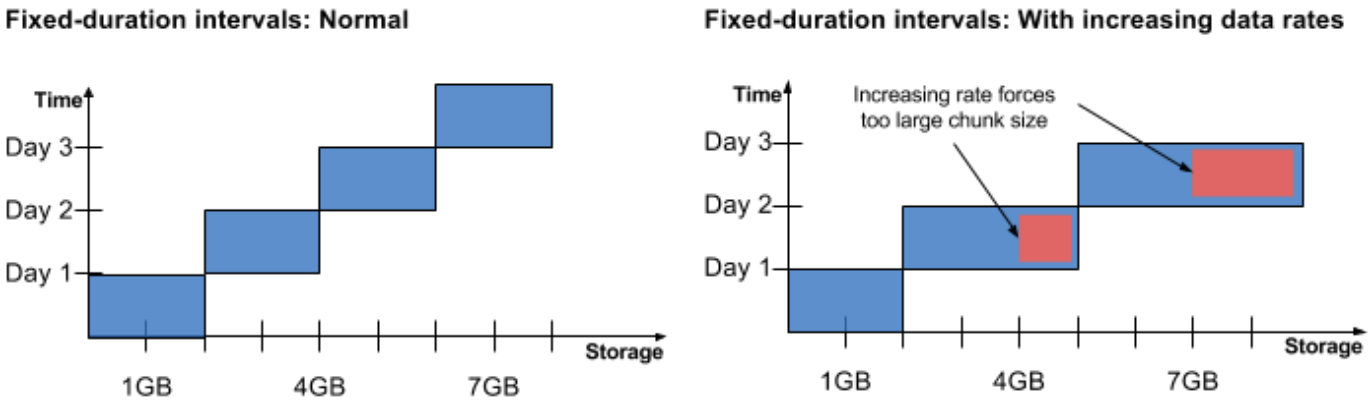
The key benefit of this approach is that now all of our indexes are built only across these much smaller chunks (tables), rather than a single table representing the entire dataset. So if we size these chunks properly, we *can* fit the latest tables (and their B-trees) completely in memory, and avoid this swap-to-disk problem, while maintaining support for multiple indexes.

Approaches to implementing chunking

The two intuitive approaches to design this time/space chunking each have significant limitations:

Approach #1: Fixed-duration intervals

Under this approach, all chunks can have fixed, identical time intervals, e.g., 1 day. This works well if the volume of data collected per interval does not change. However, as services become popular, their infrastructure correspondingly expands, leading to more servers and more monitoring data. Similarly, successful IoT products will deploy ever more numbers of devices. And once we start writing too much data to each chunk, we're regularly swapping to disk (and will find ourselves back at square one). On the flip side, choosing too-small intervals to start with leads to other performance downsides, e.g., having to touch many tables at query time.



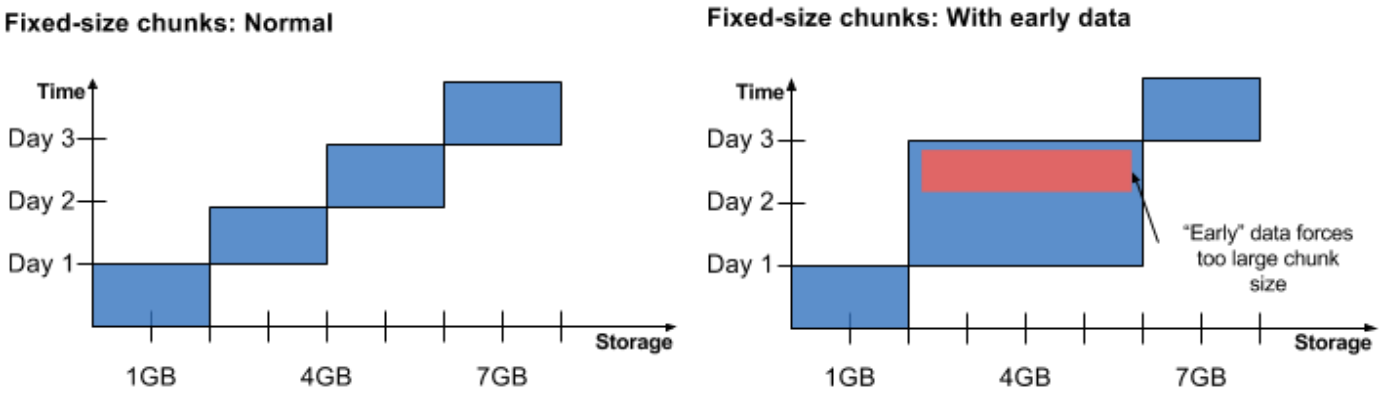
Each chunk has a fixed duration in time. Yet if the data volume per time increases, then eventually chunk size becomes too large to fit in memory.

Approach #2: Fixed-sized chunks

With this approach, all chunks have fixed target sizes, e.g., 1GB. A chunk is written to until it reaches its maximum size, at which point it becomes "closed" and its time interval constraints become fixed. Later data falling within the chunk's "closed" interval will still be written to the chunk, however, in order to preserve the correctness of the chunk's time constraints.

A key challenge is that the time interval of the chunk depends on the order of data. Consider if data (even a single datapoint) arrives "early" by hours or even days, potentially due to a non-synchronized clock, or

batch writes (such as large COPY operations), as the database needs to make sure it inserts data in temporal order to determine when a new chunk should be created (even in the *middle* of an operation). Other problems exist for fixed- or max-size chunks as well, including time intervals that may not align well with data retention policies (“delete data after 30 days”).



Each chunk’s time interval is fixed only once its maximum size has been reached. Yet if data arrives early, this creates a large interval for the chunk, and the chunk eventually becomes too large to fit in memory.

TimescaleDB takes a third approach that couples the strengths of both approaches.

Approach #3: Adaptive intervals (our current design)

Chunks are created with a fixed interval, but the interval adapts from chunk-to-chunk based on changes in data volumes in order to hit maximum target sizes.

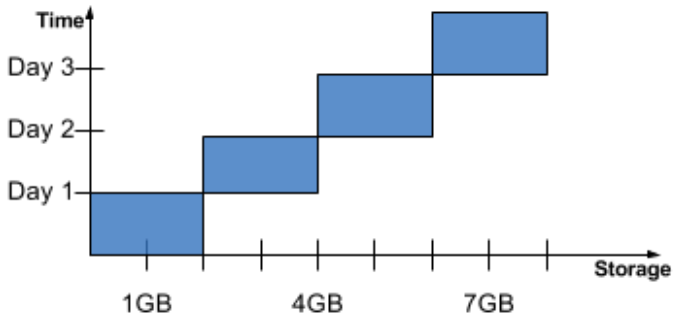
By avoiding open-ended intervals, this approach ensures that data arriving early doesn’t create too-long time intervals that will subsequently lead to over-large chunks. Further, like static intervals, it more naturally supports retention policies specified on time, e.g., “delete data after 30 days”. Given TimescaleDB’s time-based chunking, such policies are implemented by simply dropping chunks (tables) in the database. This means that individual *files* in the underlying file system can simply be deleted, rather than needing to delete individual *rows,* which requires erasing/invalidating portions of the underlying file. Such an approach therefore avoids fragmentation in the underlying database files, which in turn avoids the need for vacuuming. And this vacuuming can be prohibitively expensive in very large tables.

Still, this approach ensures that chunks are sized appropriately so that the latest ones can be maintained in

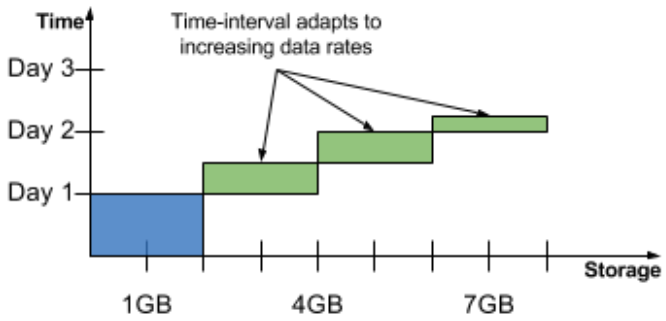
Timescale

enables better parallelization both on servers with multiple disks—for both inserts and queries — as well as multiple servers. More on these issues in a later post.

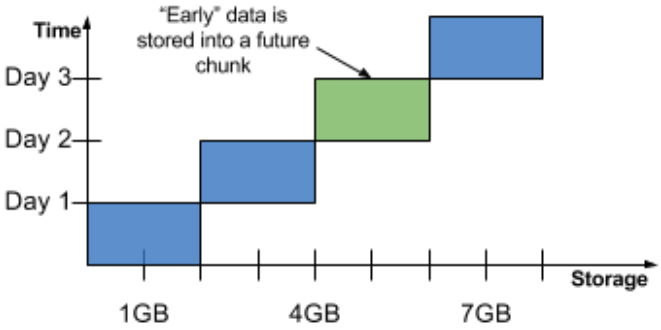
Adaptive chunks: Normal



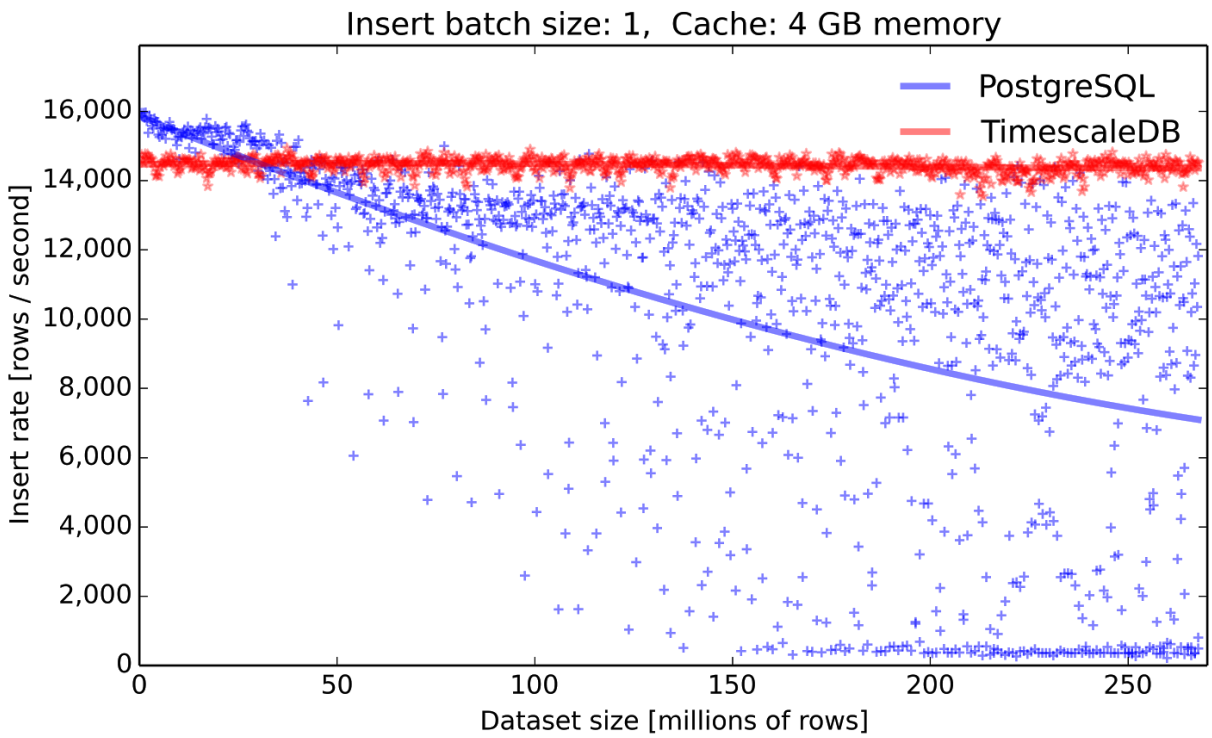
Adaptive chunks: With increasing data rates



Adaptive chunks: With early data

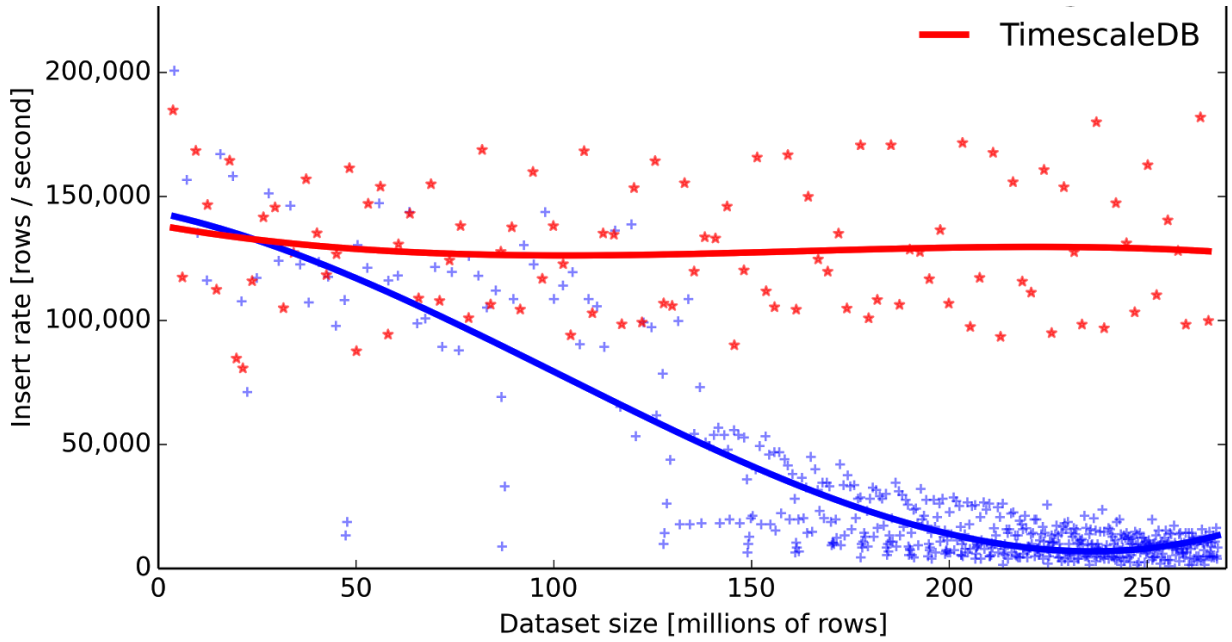


If the data volume per time increases, then chunk interval decreases to maintain right-sized chunks. If data arrives early, then data is stored into a "future" chunk to maintain right-sized chunks.



Insert throughput of TimescaleDB vs. PostgreSQL, using the same workload as described earlier. Unlike vanilla PostgreSQL, TimescaleDB maintains a constant insert rate (of about 14.4K inserts/second, or 144K metrics/second, with very low variance), independent of dataset size.

This consistent insert throughput also persists when writing large batches of rows in single operations to TimescaleDB (instead of row-by-row). Such batched inserts are common practice for databases employed in more high-scale production environments, e.g., when ingesting data from a distributed queue like Kafka. **In such scenarios, a single Timescale server can ingest 130K rows (or 1.3M metrics) per second, approximately 15x that of vanilla PostgreSQL once the table has reached a couple 100M rows.**



Insert throughput of TimescaleDB vs. PostgreSQL when performing INSERTs of 10,000-row batches.

Summary

A relational database can be quite powerful for time-series data. Yet, the costs of swapping in/out of memory significantly impacts their performance. But NoSQL approaches that implement Log Structured Merge Trees have only shifted the problem, introducing higher memory requirements and poor secondary index support.

By recognizing that time-series data is different, we are able to organize data in a new way: adaptive time/space chunking. This minimizes swapping to disk by keeping the working data set small enough to fit inside memory, while allowing us to maintain robust primary and secondary index support (and the full

Timescale

But what about performance comparisons to NoSQL databases? That post is coming soon.

In the meantime, you can download the latest version of TimescaleDB, released under the permissive Apache 2 license, on GitHub.

Like this post? Interested in learning more?

*Check out our **GitHub**, join our **Slack community**, and sign up for the community mailing list below. We're also hiring!*

Want the latest posts, time-series tutorials, and more?

Sign up for the Timescale Newsletter!

your@email.here

Sign up

Mike Freedman

Co-founder & CTO of Timescale, as well as Professor of Computer Science at Princeton University. Work broadly focuses on distributed systems, databases, networking, and security.

Read More from this author

— Timescale Blog —

Engineering

Timescale

Achieving optimal query performance with a distributed time-series database on PostgreSQL

Building columnar compression in a row-oriented database

Writing IT Metrics from Netdata to TimescaleDB

See all 27 posts →

Engineering

Analyzing Ethereum, Bitcoin, and 1200+ other Cryptocurrencies using PostgreSQL

Cryptocurrencies are fueling a modern day gold rush. Can data help us better understand this evolving market?

13 min read

Timescale

Company

When Boring is Awesome: Building a scalable time-series database on PostgreSQL

Today we are announcing the beta release of TimescaleDB, a new open-source time-series database optimized for fast ingest and complex queries, now available on GitHub under the Apache 2 license.

9 min read

Product

Timescale Cloud

TimescaleDB

Documentation

Cloud Pricing

Subscriptions

Resources

Learn

Blog

Tutorials

Webinars

Security

Community

GitHub

Stack Overflow

Slack

Email

Company

Newsroom

About Us

Careers

Contact

Privacy Policy Terms of Service