TimescaleDB: SQL made scalable for time-series data

hello@timescale.com

1 Background

Time-series data is cropping up in more and more places: monitoring and DevOps, sensor data and IoT, financial data, logistics data, app usage data, and more. Often this data is high in volume and complex in nature (e.g., multiple measurements and labels associated with a single time). This means that storing time-series data demands both scale and efficient complex queries. Yet achieving both of these properties has remained elusive. Users have typically been faced with the trade-off between the horizontal scalability of NoSQL vs. the query power of relational databases.

TimescaleDB is a new open-source database designed to make SQL scalable for time-series data. In a world typically split between RDBMS and NoSQL databases, TimescaleDB provides a third option that combines the best of both: a clustered scale-out architecture and rich support for complex queries. It scales out horizontally to support **high ingest rates** by transparently performing automatic space-time partitioning and query optimizations, yet allows users to interact with their data as if it were in a single, continuous table. It supports **fast queries** by performing efficient indexing and optimizations for selecting and aggregating non-primary keys. Data is available for querying in **real-time**, avoiding the need to delay writes in order to bulk load data. In addition, because TimescaleDB is engineered up from Postgres as an extension, it provides a **full SQL interface** (including support for secondary indexes and joins), while inheriting Postgres's reliability, mature ecosystem, and operational ease of use.

As time becomes a more critical dimension along which data is measured, TimescaleDB enables developers and organizations to harness more of its power: analyzing the past, understanding the present, and predicting the future. Unifying time-series data and relational data at the query level removes data silos, and makes demos and prototypes easier to get off the ground. The combination of scalability and a full SQL interface empowers a broad variety of people across an organization (e.g., developers, product managers, business analysts, etc.) to directly ask questions of the data. In other words, by supporting a query language already in wide use, TimescaleDB ensures that your questions are limited by your imagination, not the database.

Key Challenges. A central goal for a time-series database is to support the high write rates typical of many of these applications, which span across industries. For example, in Internet of Things (IoT) settings—whether industrial, agricultural, urban, or consumer—high write rates result from large numbers of devices coupled with modest to high write rates per device. In logistics settings, both planning data and actuals comprise time series that can be associated with each tracked object. Monitoring applications, such as in DevOps, may track many metrics per system component. Many forms of financial applications, such as those based on tick data, also rely on time-series data. All require a database that can scale to a high ingest rate.

Further, these applications often want to query their data in complex and arbitrary (yet performant) ways, beyond simply fetching or aggregating a single metric across a particular time period. Such query patterns may involve rich predicates (e.g., complex conjunctions in a WHERE clause), aggregations, statistical functions, windowed operations, JOINs against relational data, and so forth. Rather than create a new query language—which would require both new training by developers and analysts, as well as new customer interfaces or connectors to integrate with other systems—a time-series database should ideally support standard SQL and expose the abstraction of a single global table, even though the underlying storage may be partitioned or sharded between servers and/or disks. Such a design allows one to interact with data as if it were within a standard table, hiding the complexity of any data partitioning and query optimization from the user.

How do we scale SQL? At first glance, this claim of a scale-out, efficient SQL database may seem incredulous; after all, didn't the entire NoSQL movement emerge due to the limitations of SQL databases for precisely these properties? Yet this movement came in response to the use of SQL databases for traditional transactional (OLTP) workloads. In OLTP, writes are typically transactional updates of multiple rows of existing data (e.g., debiting money from one bank account and crediting another in an atomic fashion). Time-series workloads are different in two key ways.

(1) Time-series data is largely immutable. New data continually arrives, typically corresponding to the latest time periods. In other words, writes primarily occur as *new inserts*, not as updates to existing rows. Further, while the database needs to be able to support backfill for delayed data, writes are made primarily to *recent time intervals*.

(2) Workloads have a natural partitioning across both time and space. Writes typically are made to the latest time interval(s) and across the "partitioning key" in the space dimension (e.g., data sources, devices, users, etc.). Queries typically ask questions about a specific time series or data source, or across many data sources constrained to some time interval. Yet the queries might not be limited to a particular metric, but may regularly select multiple metrics at once (or use predicates that rely on multiple metrics).

Such workloads open up a new space of database architecture possibilities, of which TimescaleDB heavily takes advantage. Notably, not only are these characteristics different from traditional OLTP workloads, but also from analytical (OLAP) workloads that focus on read-heavy rollups and aggregations of single metrics.

2 Limitations of existing solutions

Existing solutions require users to choose between either scalability or rich query support.

Vanilla RDBMS (e.g., PostgreSQL, MySQL). Traditional SQL databases have two key problems in handling high ingest rates: They have poor write performance for large tables, and this problem only becomes worse over time as data volume grows linearly in time. These problems emerge when table indexes can no longer fit in memory, as each insert will translate to many disk fetches to swap in portions of the indexes' B-Trees. Further, any data deletion (to save space or to implement data retention policies) will require expensive "vacuuming" operations to defragment the disk storage associated with such tables. Also, out-of-the-box open-source solutions for scaling-out RDBMS across many servers are still lacking.

NoSQL and time-series DBs. In response, developers sometimes adopt NoSQL databases (e.g., Cassandra, Mongo) or modern time-series databases (e.g., OpenTSDB, InfluxDB) for their needs. These are typically column-oriented databases for fast ingest and fast analytical queries over one column, often exposing a simpler key-value interface. Depending on data model choices, this works well for visualizing either a *single metric* (e.g., the CPU utilization of a device) or some aggregate metric (e.g., average CPU over all devices). But, they often lack a rich query language or secondary index support, and suffer high latency on complex queries.

For example, databases like InfluxDB will fall back to full table scans if the WHERE clause of their custom query language includes a numerical column (e.g., cpu > 0.7). String columns pose other challenges for most NoSQL time-series databases. For example, one approach for indexing string columns (e.g., ip_address, uuid, event_name) is to use a concatenated string of all indexed values as a key. This creates a tradeoff between the number of indexed fields and performance: searching for one field requires a scan across all other indexed fields. These problems make multi-field queries (e.g., "all metrics from devices of a certain type with low battery") less efficient. Such queries are quite common in dashboards, reports, and data exploration. Further, they lack the reliability, tooling, and ecosystem of more widely-used traditional RDBMS systems.

Data lakes (e.g., SQL / Hadoop / Spark on HDFS). Distributed block/file systems like HDFS avoid the need to predefine data models or schemas, and easily scale by adding more servers. They also can handle high write rates by large, immutable batches of data. However, they pay the cost at query time, lacking the highly structured indexes needed for fast and resource-efficient queries. Further, data backfill and updates are difficult and very expensive, given their underlying storage in large immutable blocks.

So, while query engines on top of HDFS can expose a SQL or rich programming interface (e.g., Presto, Pig, SparkSQL, etc.) many types of queries that can be handled efficiently by RDBMS with appropriate index support turn into full table scans in their underlying storage interface. While HDFS allows such scans to be easily parallelized, query latency and throughput still suffer significantly. This relegates such architectures more for single-tenant data exploration or report generation, not dashboarding or real-time operational monitoring.

3 TimescaleDB architecture

TimescaleDB is an open-source database for time-series data. Its goal is to enable both the scale-out nature of NoSQL databases, and the reliability and query support of a traditional relational DB. We've designed a new clustered DB in this manner, built around a Postgres core running on each server.

TimescaleDB supports the key features of a modern time-series database, as summarized in Figure 1.

Hypertables and chunks. At a high-level, the database exposes the abstraction of a single continuous table—a ***hypertable***—across all space and time intervals, such that one can query it via vanilla SQL. A hypertable is defined with a standard schema with column names and types, with at least one column specifying a time value, and—in clustered deployments—one column specifying a "partitioning key" over which the dataset can be additionally partitioned.

- **Time-series data optimized**
- **Full SQL interface**
- **Scaling up and out**
  - Transparent time/space partitioning
  - Parallelized ops across chunks and servers
  - Right-sized chunks for single nodes
- **High data write rates**
  - Batched commits
  - In-memory indexes
  - Transactional support
  - Support for data backfill
- **Optimizations for complex queries**
  - Intelligent chunk selection for queries
  - Minimize scanning for distinct items
  - Limit pushdowns
  - Parallelized aggregation
- **Leverage existing query planner**
  - Join against relational data
  - Geo-spatial queries
- **Flexible management**
  - Leverage existing DB ecosystem and tooling
  - Highly reliable (streaming replication, backups)
  - Automated data retention policies

Figure 1: Key features of TimescaleDB.

Internally, TimescaleDB automatically splits the hypertable into ***chunks***, where a chunk corresponds to a "two-dimensional" split according to a specific time interval and a region of the partition key's space (e.g., using hashing). Each chunk is implemented using a standard database table that is automatically placed on one of the database nodes (or replicated between multiple nodes), although this detail is largely hidden from users. A single TimescaleDB deployment can store multiple hypertables, each with different schemas.

Engineered up from PostgreSQL. By choosing to engineer up from PostgreSQL, rather than building from scratch, TimescaleDB gains four immediate benefits.

1. **Rock-solid reliability**. At its core, TimescaleDB's reliability manifests from Postgres' 20-year open-source record and strong developer community.
2. **Mature ecosystem**. TimescaleDB users can connect via standard ODBC, JDBC, or Postgres for third-party visualization tools, BI tools, management interfaces, web platforms and ORMs.
3. **Standard interface**. TimescaleDB users do not need to learn a new query language and management framework, and can leverage their existing comfort with SQL and Postgres.
4. **Operational Ease of Use**. Users can reuse known and trusted methods for backups, snapshots, active replication, and other operational tasks.

In fact, all of TimescaleDB is implemented as a Postgres extension, rather than a fork, so can be installed on a standard distribution of Postgres.

When you might want to consider alternatives. Database design typically requires making deliberate trade-offs, and we would be remiss not to mention the scenarios where there may be better alternatives:

- **Simple read requirements**: When most of your query patterns are simple in nature (e.g., key-based lookups, or one dimensional rollups over time).
- **Low available storage**: When resource constraints place storage at a premium, and heavy compression is required. (Although this is an area of active development, and we expect TimescaleDB to improve.)
- **Sparse and/or unstructured data**: When your time-series data is especially sparse and/or generally unstructured. (But even if your data is partially structured, TimescaleDB includes a JSONB field type for the unstructured part(s). This allows you to maintain indexes on the structured parts of your data combined with the flexibility of unstructured storage.)

4 Technical details

Ideally database users should be able to interact with time-series data as if it were in a simple continuous database table. However, for reasons discussed above, using a single table does not scale. Yet requiring users to manually partition their data exposes a host of complexities, e.g., forcing users to constantly specify which partitions to query, how to compute JOINs between them, or how to properly size these tables as workloads change.

To avoid this management complexity while still scaling and supporting efficient queries, TimescaleDB hides its automated data partitioning and query optimizations behind its hypertable abstraction. Creating a hypertable and its corresponding schema is a simple, standard SQL command (see Figure 2), and this hypertable can then be accessed as if it were a single table using standard SQL commands (see Figure 3). Further, just like a normal database table, this schema can be altered via standard SQL commands; transparently to the user, TimescaleDB

```
# Create a schema for a new hypertable
CREATE TABLE sensor_data (
  "time" timestamp with time zone NOT NULL,
  device_id TEXT NOT NULL,
  location TEXT NULL,
  temperature NUMERIC NULL,
  humidity NUMERIC NULL,
  pm25 NUMERIC
);

# Create a hypertable from this data
SELECT create_hypertable
  ('sensor_data', 'time', 'device_id', 16);

# Migrate data from existing Postgres table into
# a TimescaleDB hypertable
INSERT INTO sensor_data (SELECT * FROM old_data);

# Query hypertable like any SQL table
SELECT device_id, AVG(temperature) from sensor_data
  WHERE temperature IS NOT NULL AND humidity > 0.5
    AND time > now() - interval '7 day'
  GROUP BY device_id;
```

Figure 2: Creating a new hypertable is a simple SQL command. Subsequently, users can interact with a hypertable as if it's a standard, single database table, including issuing full SQL queries on it.

```
# Metrics about resource-constrained devices
SELECT time, cpu, freemem, battery FROM devops
  WHERE device_id='foo'
    AND cpu > 0.7 AND freemem < 0.2
  ORDER BY time DESC
  LIMIT 100;

# Calculate total errors by latest firmware versions
# per hour over the last 7 days
SELECT date_trunc('hour', time) as hour, firmware,
    COUNT(error_msg) as errno FROM data
  WHERE firmware > 50
    AND time > now() - interval '7 day'
  GROUP BY hour, firmware
  ORDER BY hour DESC, errno DESC;

# Find average bus speed in last hour
# for each NYC borough
SELECT loc.region, AVG(bus.speed) FROM bus
    INNER JOIN loc ON (bus.bus_id = loc.bus_id)
  WHERE loc.city = 'nyc'
    AND bus.time > now() - interval '1 hour'
  GROUP BY loc.region;
```

Figure 3: Query examples using standard SQL and the abstraction of a single global table. Note that the complexities of data partitioning for scalability is hidden from the user.

is atomically modifying the schemas of all the underlying chunks that comprise a hypertable.

TimescaleDB provides this functionality by hooking into the query planner of Postgres, so that it receives the native SQL parse tree. It then can use this tree to determine which servers and hypertable chunks (native database tables) to access, how to perform distributed and parallel optimizations, etc.

Many of these same optimizations even apply to single-node deployments, where automatically splitting hypertables into chunks and related query optimizations still provides a number of performance benefits.

We highlight and explain TimescaleDB's architecture and design choices in the rest of this section.

4.1 Scaling up and out

Transparent time/space partitioning. As discussed, the database scales by partitioning hypertables in two dimensions: by time interval, and by a "partitioning key" over some primary index for the data (e.g., device identifiers for sensor data, locations, customers, users, etc.). Each time/space partition is called a chunk, which is created and placed on a server and disk automatically by the system. An illustration of TimescaleDB's architecture is shown in Figure 4.

Every database node knows about the time and space ranges comprising data in each chunk, and each node builds local indexes on each individual chunk. As we discuss later, information about chunks' ranges allows the query planner to determine which chunks to query when resolving a query, particularly when WHERE or GROUP BY clauses include the time or space (partitioning key) dimension (this is commonly referred to as constraint exclusion analysis).

Local indexes can be built on any database column, not only on the main partitioning key, and are defined as the conjunction of both time and the column being indexed. For example, in the hypertable shown in Figure 2, if the partitioning key is device id, secondary indexes also can be defined on location and any of the numerical sensor readings (temperature, humidity, particulate matter). By defining the local index on both these columns and time, TimescaleDB's planner again knows how to optimize queries given any use of time in the query predicate.

Parallelizing across chunks and servers. Chunks are dynamically created by the runtime and sized to optimize performance in both cluster and single-node environments. When run as a cluster, chunks are placed on different servers; on a single machine, chunks can also be automatically spread across disks. With either approach, partitioning by space parallelizes inserts to recent time intervals. Similarly, query patterns often slice across time or space, so also enjoy performance improvements through smart chunk placement.

By default, chunks belonging to the same region of partition keyspace, yet varying by time intervals, are collocated on the same servers. This avoids queries touching

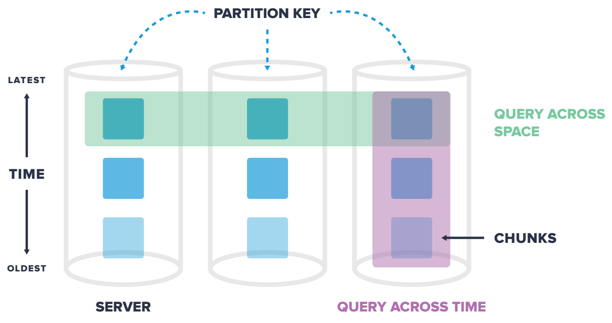


Figure 4: TimescaleDB architecture. Data is partitioned across both time and space, with the resulting chunks organized for common query patterns. Writes are largely sent to the latest chunks, while queries slice across both time and space.

all servers when performing queries for a single object in space (e.g., a particular device), which helps reduce tail latency under higher query loads.

Right sizing chunks for single nodes. Even in single-node settings, chunking still improves performance over the vanilla use of a single database table for both inserts and deletes. Right-sized chunks ensure that all of the B-trees for a table's indexes can reside in memory during inserts to avoid thrashing while modifying arbitrary locations in those trees. Further, by avoiding overly large chunks, we can avoid expensive "vacuuming" operations when removing deleted data according to automated retention policies, as the runtime can perform such operations by simply dropping chunks (internal tables), rather than deleting individual rows. At the same time, avoiding too-small chunks improves query performance by not needing to read additional tables and indexes from disk.

TimescaleDB performs such time/space partitioning automatically based on table *sizes*, rather than an approach based on static time intervals more commonly practiced (e.g., by creating a separate table per day). When systems lack TimescaleDB's transparent hypertable abstraction, interval-based partitioning might make the manual table selection and joins at least tractable (although not easy). But, such intervals work poorly as data volumes change, e.g., a time interval appropriate for a service pulling data from 100 devices is not appropriate when that system scales to 100K devices. TimescaleDB avoids the need to make this choice by managing data partitioning automatically.

4.2 High data write rates

Batched commits. Writes are typically made to recent time intervals, rather than old tables. This allows TimescaleDB to efficiently write batch inserts to a small number of tables as opposed to performing many small writes. Further, our scale-out design also takes advantage of time-series workloads to recent time intervals, in order to parallelize writes across many servers and/or disks to further support high data ingest rates. These approaches improve performance when employed on either HDDs or SSDs.

In-memory indexes. Because chunks are right-sized to servers, and thus the database never builds massive single tables, TimescaleDB avoids swapping indexes to disks for recent time intervals (where most writes occur). Yet it can efficiently support any type of Postgres index on columns, from more traditional text or numerical columns, to more specialized indexes on array data types or GIS (spatial) columns.

Transactional support. TimescaleDB supports full transactions over entries with the same partition key. In a monitoring application, for example, this ensures transactional semantics on a per-device basis and guarantees that multiple device measurements, which may each involve many individual sensor metrics, are atomically inserted.

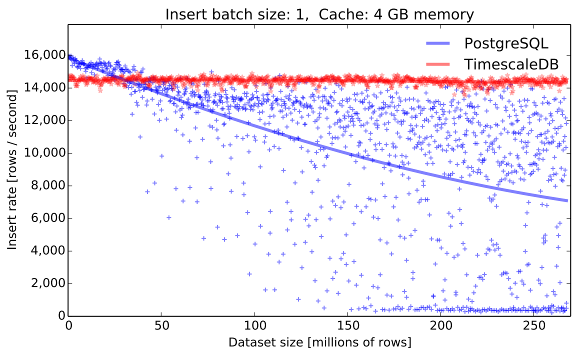
Support for data backfill. Even though TimescaleDB's architecture is optimized for the scenario when most writes are to the latest time intervals, it fully supports the "backfill" of delayed data. Additionally, the database's automated chunk management is also aware of the possibility of backfill, so a modest amount of delayed data can be configured to not "overflow" a chunk's size limit to unnecessarily create additional undersized chunks for that interval.

Performance benefits for single nodes. While data partitioning is traditionally seen as a mechanism for scaling out to many servers, TimescaleDB's approach also provides meaningful performance improvements even when employed on a single machine.

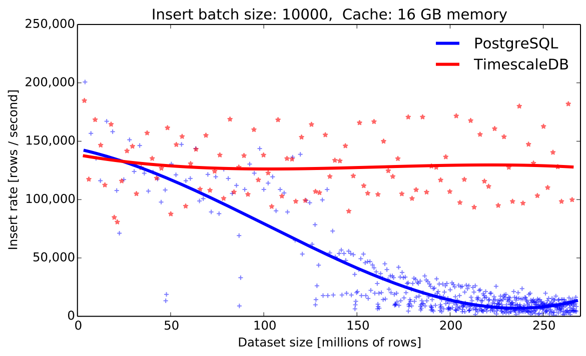
To evaluate this behavior, we performed experiments where clients both insert individual rows to the databases, as well as large batches of rows in single operations. Each row includes 12 values in separate columns: a timestamp, an indexed randomly-chosen primary id, and 10 additional numerical metrics. 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. Figure 5 illustrates the results.

In both scenarios, standard Postgres tables hit a performance cliff after tens of millions of rows. Not only does throughput drop off, but the variance increases significantly. In single-row inserts, the database achieves only hundreds of inserts per second quite regularly. With batched inserts, the insert rate for Postgres converges to only thousands of inserts period, around a 30x decrease from its performance at the onset.

TimescaleDB, on the other hand, maintains constant insert performance and low variance regardless of the database size, as individual chunks remain appropriately sized. When inserting in batches, TimescaleDB starts out



(a) Single-row inserts, 4GB cache



(b) Batched inserts, 16GB cache

Figure 5: Insert throughput into TimescaleDB's chunked hypertable (red) compared to a standard table in PostgreSQL (blue) on a single database server. Experiments run using PostgreSQL 9.6.2 on a Azure standard DS4 v2 (8 core) machine with SSD-based (premium LRS) storage. Every inserted row has 12 columns. Trend lines constitute a polynomial fit of the underlying data; each datapoint shows the average insert throughput over a 20s period. Throughput drops in PostgreSQL as tables grow large, while hypertable insert performance remains constant regardless of total data volume.

at an average rate of about 140,000 rows per second, much like vanilla Postgres. But unlike Postgres, TimescaleDB maintains its stable performance regardless of data size.

4.3 Optimizations for complex queries

Intelligently selecting chunks needed to satisfy queries. Common queries to time-series data include (i) slicing across time for a given object (e.g., device), (ii) slicing across many objects for a given time interval, or (iii) querying the last reported data records across (a subset of) all objects or some other distinct object label. Such queries to time or space, which may require scanning over many chunks (disks, servers), are illustrated in Figure 4.

While users perform these queries as if interacting with a single hypertable, TimescaleDB leverages internally-managed metadata to only query those chunks that may *possibly* satisfy the query predicate. By aggressively pruning many chunks and servers to contact in its query plan, TimescaleDB improves both query latency and throughput.

Minimizing scanning back in time: "LIMIT BY" queries for distinct items. Similarly, for items like unique devices, users, or locations, one often wants to ask questions like "give me the last reading for every device." While this query can be natively expressed in SQL using windowing operators, such a query would turn into a full table scan for most relational databases. In fact, this full table scan could continue back to the beginning of time to capture "for every device" or at best sacrifice completeness with some arbitrarily-specified time range.

To efficiently support such queries, TimescaleDB automatically tracks metadata about "distinct" items in the database, as specified in the hypertable's schema. When coupled with its other optimizations, such queries touch only the necessary chunks and perform efficiently-indexed queries on each individual chunk.

Minimizing ordered data from distinct chunks. TimescaleDB provides a number of additional query optimizations that benefit both single-node and clustered deployments. Postgres already enables a "merge append" optimization when combining in-order data from multiple chunks, where a query efficiently combines data from these tables in sorted order, adding rows one-by-one to the query result in the proper sorted order, and then stopping when the global query has been satisfied (e.g., based on its LIMIT). This optimization ensures that a subquery is only incrementally processed on a table if its result would benefit the final global result set. This is particularly beneficial for queries with complex predicates, such that finding the "next" item that matches the predicate can involve scanning a significant amount of data.

TimescaleDB extends such merge-appends optimizations to bring these significant efficiencies to time-based aggregates. Such aggregates appear quite regularly for time-series analysis, such as "tell me the average of a metric per hour, for the last 12 hours that the device has reported data" which expressed as SQL involves "GROUP BY hour ORDER BY hour DESC LIMIT 12." This way, even without a strict time-range specified by the user (unlike in many time-series databases), the database will only process those minimal set of chunks and data needed to answer this query.

Parallelized aggregation. Much like its LIMIT push-down, TimescaleDB also pushes down aggregations for many common functions (e.g., SUM, AVG, MIN, MAX, COUNT) to the servers on which the chunks reside. Primarily a benefit for clustered deployments, this distributed query optimization greatly minimizes network transfers by performing large rollups or GROUP_BYs *in situ* on the chunks' servers, so that only the computed results need to be joined towards the end of the query, rather than raw data from each chunk.

4.4 Leveraging the RDBMS query planner

Because each node runs a full-fledged Postgres query planner and data model, deploying new optimizations for particular queries, indexes, and data types are easy.

From the start, TimescaleDB supports Postgres' full SQL interface. In the current implementation, some queries are optimized more than others. However, subsequent releases over time will include additional query optimizations, allowing the database to "grow" with users' needs, without requiring any changes by users.

Join against relational data. Today, you can compute joins between hypertables and standard relational tables, which are either stored directly in the database or accessed via foreign data wrappers to external databases. Future optimizations will minimize data movement during joins.

Most time-series databases today do not support such JOINs. That lack of support requires that users denormalize their data by storing additional metadata or labels with every time-series record. This approach greatly expands data sizes, and makes updating metadata very expensive. Alternatively, application writers "silo" their data between databases, and then require the application writer to perform this join between relational and time-series data outside of the database, which increases overall system complexity.

Geo-spatial queries. TimescaleDB supports arbitrary Postgres data types within the time series, including GPS-coordinate data by leveraging PostGIS, which provides best-in-class support for geo-spatial data. Because a chunk can use any type of indexing on its data, GIN/GiST indexes are supported on GIS data right out of the box.

4.5 Flexible management

Because chunks are native database tables internally, we can readily leverage the rich set of existing Postgres capabilities in TimescaleDB.

Tooling ecosystem. TimescaleDB leverages the database management tooling and features that have developed within the Postgres ecosystem over two decades. Users can connect to TimescaleDB via standard JDBC or ODBC connectors and `psql` command-line tools. Yet given the way that TimescaleDB's partitioning is implemented, users typically only see their hypertables (on which they can specify these management functions), rather than their concomitant chunks.

Highly reliable (replication and backups). TimescaleDB can reuse the battle-tested replication techniques employed by Postgres, namely, streaming replication and cold/hot standbys, as well as backups. It also uses Postgres' write-ahead log (WAL) for consistent checkpointing. In other words, even though replication or backup policies can be defined (or commands issued) on the hypertable, TimescaleDB performs these actions by replicating or checkpointing the hypertable's constituent chunks.

Automated data retention policies. TimescaleDB allows for easily defining data retention policies based on time. For example, users can configure the system to cleanup/erase data more than X weeks old. TimescaleDB's time-interval-based chunking also helps make such retention policies more efficient, as the database can then just DROP its internal data tables that are expired, as opposed to needing to delete individual rows and aggressively vacuum the returning tables. For efficiency, these policies can be implemented lazily, i.e., individual records that are older than the expiry period might not be immediately deleted. Rather, when all data in a chunk becomes expired, then the entire chunk can just be dropped.

5 Conclusion and Status

Many time-series applications today are asking more complex questions of their data than before: analyzing historical trends, monitoring current behavior, identifying possible problems, predicting future behavior, etc. The data, in turn, is being collected at higher volumes and velocities. In order to serve these applications, the modern time-series database needs to marry both scalability and support for highly performant complex queries.

TimescaleDB achieves that combination through its automatic time/space partitioning, optimized query planning, and deep integration with PostgreSQL. At the same time, TimescaleDB delivers this performance through an easy-to-use interface, thanks to its hypertable abstraction, as well as full SQL support.

TimescaleDB is in active development by a team of PhDs based in New York City and Stockholm, backed by top-tier investors. An open-source single-node version featuring scalability and full SQL support is currently available for download. A clustered version is in private beta with select customers. We welcome any feedback at hello@timescale.com.

Publication: v6 – April 7, 2017