TigerData Architecture for Real-Time Analytics

Michael J. Freedman
TigerData

James Blackwood-Sewell
TigerData

Abstract

TigerData (formerly Timescale) is the creator of TimescaleDB, a powerful open-source application database designed for real-time analytics on time-series data. Built as a PostgreSQL extension, TimescaleDB integrates seamlessly with the Postgres ecosystem and enhances it with automatic time-based partitioning, hybrid row-columnar storage, and vectorized execution—enabling high-ingest performance, sub-second queries, and full SQL support at scale.

Through incrementally updated materialized views and advanced analytical functions, TimescaleDB reduces compute overhead and significantly improves query efficiency. Developers can continue using familiar SQL workflows and tools, while benefiting from a database purpose-built for fast, scalable analytics.

TigerData also offers Tiger Postgres, an enhanced PostgreSQL distribution that includes and extends TimescaleDB. Our fully managed, cloud-native database service, Tiger Cloud, runs optimized Tiger Postgres instances to deliver a seamless developer experience and powerful analytics at scale.

This document outlines the architectural choices and optimizations that power Timescale's performance and scalability while preserving PostgreSQL's reliability and transactional guarantees.

1 Introduction

1.1 What is real-time analytics?

Real-time analytics enables applications to process and query data as it is generated and as it accumulates, delivering immediate and ongoing insights for decision-making. Unlike traditional analytics, which relies on batch processing and delayed reporting, real-time analytics supports both instant queries on fresh data and fast exploration of historical trends—powering applications with sub-second query performance across vast, continuously growing datasets.

Many modern applications depend on real-time analytics to drive critical functionality. For example: (1) IoT monitoring systems track sensor data over time, identifying long-term performance patterns while still surfacing anomalies as they arise. This allows businesses to optimize maintenance schedules, reduce costs, and improve reliability. (2) Financial and business intelligence platforms analyze both current and historical data to detect trends, assess risk, and uncover opportunities—from tracking stock performance over a day, week, or year to identifying spending patterns across millions of transactions. (3) Interactive customer dashboards empower users to explore live and historical data in a seamless experience—whether it's a SaaS product providing real-time analytics on business operations, a media platform analyzing content engagement, or an e-commerce site surfacing personalized recommendations based on recent and past behavior.

Real-time analytics isn't just about reacting to the latest data, although that is critically important. It's also about delivering fast, interactive, and scalable insights across all your data, enabling better decision-making and richer user experiences. Unlike traditional ad-hoc analytics used by analysts, real-time analytics powers applications—driving dynamic dashboards, automated decisions, and user-facing insights at scale.

To achieve this, real-time analytics systems must meet several key requirements:

- ***Low-latency queries*** ensure sub-second response times even under high load, enabling fast insights for dashboards, monitoring, and alerting.
- ***Low-latency ingest*** minimizes the lag between when data is created and when it becomes available for analysis, ensuring fresh and accurate insights.
- ***Data mutability*** allows for efficient updates, corrections, and backfills, ensuring analytics reflect the most accurate state of the data.
- ***Concurrency and scalability*** enable systems to handle high query volumes and growing workloads without degradation in performance.
- ***Seamless access to both recent and historical data*** ensures fast queries across time, whether analyzing live, streaming data, or running deep historical queries on days or months of information.
- ***Query flexibility*** provides full SQL support, allowing for complex queries with joins, filters, aggregations, and analytical functions.

1.2 TigerData: Real-time analytics from PostgreSQL

Tiger Postgres is a high-performance database that runs on Tiger Cloud to bring real-time analytics to applications. It combines fast queries, high ingest performance, and full SQL support—all while ensuring scalability and reliability in the cloud. Tiger Postgres extends PostgreSQL with the TimescaleDB extension (which can also be self-hosted). It enables sub-second queries on vast amounts of incoming data

while providing optimizations designed for continuously updating datasets.

TigerData achieves this through the following optimizations:

- ***Efficient data partitioning***: Automatically and transparently partitioning data into chunks, ensuring fast queries, minimal indexing overhead, and seamless scalability
- ***Row-columnar storage***: Providing the flexibility of a row store for transactions and the performance of a column store for analytics
- ***Optimized query execution***: Using techniques like chunk and batch exclusion, columnar storage, and vectorized execution to minimize latency
- ***Continuous aggregates***: Precomputing analytical results for fast insights without expensive reprocessing
- ***Cloud-native operation***: Compute/compute separation, elastic usage-based storage, horizontal scale out, data tiering to object storage
- ***Operational simplicity*** Offering high availability, connection pooling, and automated backups for reliable and scalable real-time applications

With TigerData, developers can build low-latency, high-concurrency applications that seamlessly handle streaming data, historical queries, and real-time analytics while leveraging the familiarity and power of PostgreSQL.

2 Data model

Today's applications demand a database that can handle real-time analytics and transactional queries without sacrificing speed, flexibility, or SQL compatibility (including joins between tables). TimescaleDB achieves this with *hypertables*, which provide an automatic partitioning engine, and *hypercore*, a hybrid row-columnar storage engine designed to deliver high-performance queries and efficient compression (up to 95%) within PostgreSQL.

2.1 Efficient data partitioning

TimescaleDB provides hypertables (Figure 1), a table abstraction that automatically partitions data into chunks in real time (using time stamps or incrementing IDs) to ensure fast queries and predictable performance as datasets grow. Unlike traditional relational databases that require manual partitioning, hypertables automate all aspects of partition management, keeping locking minimal even under high ingest load.

At ingest time, hypertables ensure that PostgreSQL can deal with a constant stream of data without suffering from table bloat and index degradation by automatically partitioning data across time. Because each chunk is ordered by time and has its own indexes and storage, writes are usually isolated to small, recent chunks—keeping index sizes small, improving cache locality, and reducing the overhead

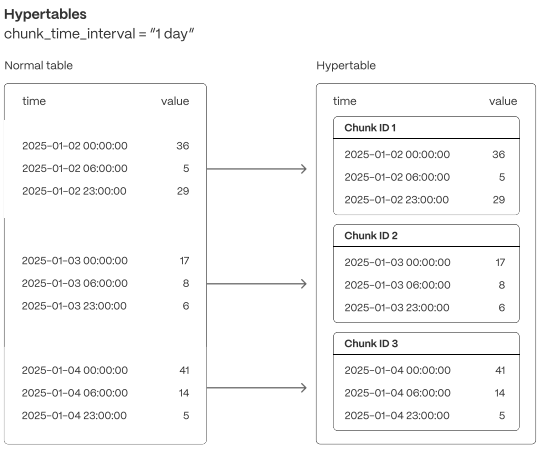


Figure 1. Hypertables automatically partition data into disjoint chunks along a primary column (typically a timestamp or monotonic ID), allowing for efficient data management.

of vacuum and background maintenance operations. This localized write pattern minimizes write amplification and ensures consistently high ingest performance, even as total data volume grows.

At query time, hypertables efficiently exclude irrelevant chunks from the execution plan when the partitioning column is used in a `WHERE` clause. This architecture ensures fast query execution, avoiding the gradual slowdowns that affect non-partitioned tables as they accumulate millions of rows. Chunk-local indexes keep indexing overhead minimal, ensuring index operations scans remain efficient regardless of dataset size.

Hypertables are the foundation for all of TimescaleDB's real-time analytics capabilities. They enable seamless data ingestion, high-throughput writes, optimized query execution, and chunk-based lifecycle management—including automated data retention (drop a chunk) and data tiering (move a chunk to object storage).

2.2 Row-columnar storage

Traditional databases force a trade-off between fast inserts (row-based storage) and efficient analytics (columnar storage). TimescaleDB's storage engine, hypercore, eliminates this trade-off, allowing real-time analytics without sacrificing transactional capabilities.

Hypercore dynamically stores data in the most efficient format for its lifecycle (Figure 2):

- ***Row-based storage for recent data***: The most recent chunk (and possibly more) is always stored in the row-store, ensuring fast inserts, updates, and low-latency single record queries. Additionally, row-based storage is used as a write through for inserts and updates to columnar storage.
- ***Columnar storage for analytical performance***: Chunks are automatically compressed into the columnstore,

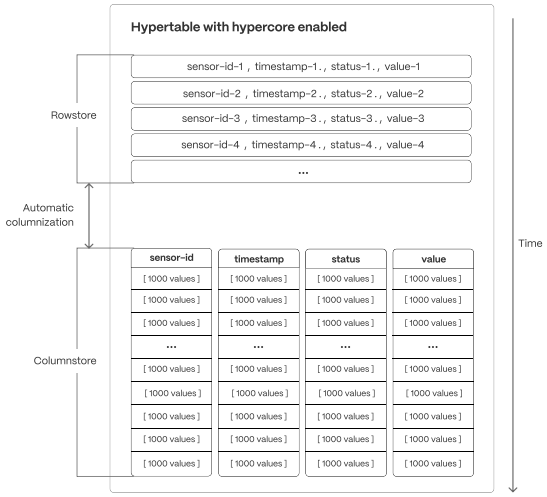


Figure 2. With hypercore, recent chunks can remain in rowstore format, while chunks covering older ranges are automatically converted into columnar format.

optimizing storage efficiency and accelerating analytical queries.

Unlike traditional columnar databases, hypercore allows data to be inserted or modified at any stage, making it a flexible solution for both high-ingest transactional workloads and real-time analytics—within a single database.

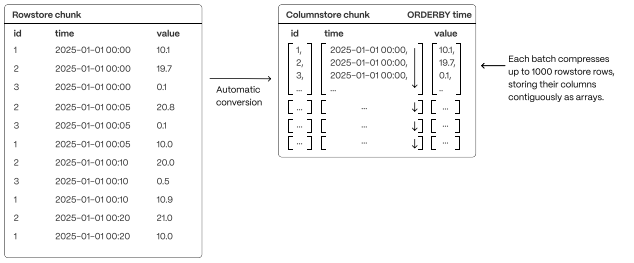
2.3 Columnar storage layout

TimescaleDB's columnar storage layout optimizes analytical query performance by structuring data efficiently on disk, reducing scan times, and maximizing compression rates. Unlike traditional row-based storage, where data is stored sequentially by row, columnar storage organizes and compresses data by column, allowing queries to retrieve only the necessary fields in batches rather than scanning entire rows. But unlike many column store implementations, TimescaleDB's columnstore supports full mutability—inserts, upserts, updates, and deletes, even at the individual record level—with transactional guarantees. Data is also immediately visible to queries as soon as it is written.

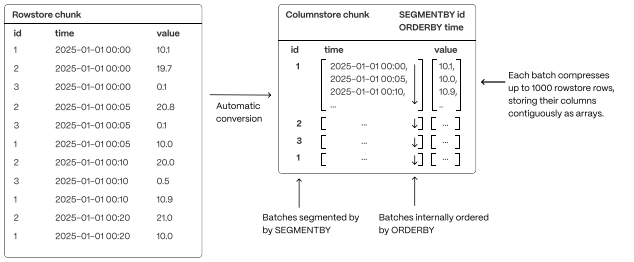
Columnar batches. TimescaleDB uses columnar collocation and columnar compression within row-based storage to optimize analytical query performance while maintaining full PostgreSQL compatibility. This approach ensures efficient storage, high compression ratios, and rapid query execution.

A rowstore chunk is converted to a columnstore chunk by successfully grouping together sets of rows (typically up to 1000) into a single *batch*, then converting the batch into columnar form (as shown in Figure 3a).

Each compressed batch does the following:



(a) When converting a hypertable chunk to a columnstore, batches of rows (typically up to 1000) are rewritten as columnar array and compressed with type-specific compression algorithms.



(b) To optimize query performance, batches of rows with a common identifier (`SEGMENTBY`) are grouped and collocated together, and ordered within batches as appropriate (`ORDERBY`).

Figure 3. Columnstore layout

- Encapsulates columnar data in compressed arrays of up to 1,000 values per column, stored as a single entry in the underlying compressed table
- Uses a column-major format within the batch, enabling efficient scans by co-locating values of the same column and allowing the selection of individual columns without reading the entire batch
- Applies advanced compression techniques at the column level, including run-length encoding, delta encoding, and Gorilla compression, to significantly reduce storage footprint (by up to 95%) and improve I/O performance.

While the chunk interval of rowstore and columnstore batches usually remains the same, TimescaleDB can also combine columnstore batches so they use a different chunk interval.

This architecture provides the benefits of columnar storage – optimized scans, reduced disk I/O, and improved analytical performance — while seamlessly integrating with PostgreSQL's row-based execution model.

Segmenting and ordering data. To optimize query performance, TimescaleDB allows explicit control over how data is physically organized within columnar storage (as shown in

Figure 3b). By structuring data effectively, queries can minimize disk reads and execute more efficiently, using vectorized execution for parallel batch processing where possible.

Timescale's data model employs several optimizations:

- ***Group related data together to improve scan efficiency.*** Organizing rows into logical segments ensures that queries filtering by a specific value only scan relevant data sections. For example, in the above, querying for a specific ID is particularly fast. (Implemented with `SEGMENTBY`.)
- ***Sort data within segments to accelerate range queries.*** Defining a consistent order reduces the need for post-query sorting, making time-based queries and range scans more efficient. (Implemented with `ORDERBY`.)
- ***Reduce disk reads and maximize vectorized execution.*** A well-structured storage layout enables efficient batch processing (Single Instruction, Multiple Data, or SIMD vectorization) and parallel execution, optimizing query performance.

By combining segmentation and ordering, TimescaleDB ensures that columnar queries are not only fast but also resource-efficient, enabling high-performance real-time analytics.

2.4 Data mutability

Traditional databases force a trade-off between fast updates and efficient analytics. Fully immutable storage is impractical in real-world applications, where data needs to change. Asynchronous mutability—where updates only become visible after batch processing—introduces delays that break real-time workflows. In-place mutability, while theoretically ideal, is prohibitively slow in columnar storage, requiring costly decompression, segmentation, ordering, and recompression cycles.

Hypercore navigates these trade-offs with a hybrid approach that enables immediate updates without modifying compressed columnstore data in place, as shown in Figure 4. By staging changes in an interim rowstore chunk, hypercore allows updates and deletes to happen efficiently while preserving the analytical performance of columnar storage.

Real-time writes without delays. All new data which is destined for a columnstore chunk is first written to an interim rowstore chunk, ensuring high-speed ingestion and immediate queryability. Unlike fully columnar systems that require ingestion to go through compression pipelines, hypercore allows fresh data to remain in a fast row-based structure before being later compressed into columnar format in ordered batches as normal.

Queries transparently access both the rowstore and columnstore chunks, meaning applications always see the latest data instantly, regardless of its storage format.

Efficient updates and deletes without performance penalties. When modifying or deleting existing data, hypercore avoids the inefficiencies of both asynchronous updates and in-place modifications. Instead of modifying compressed storage directly, affected batches are decompressed and staged in the interim rowstore chunk, where changes are applied immediately.

These modified batches remain in row storage until they are recompressed and reintegrated into the columnstore (which happens automatically via a background process). This approach ensures updates are immediately visible, but without the expensive overhead of decompressing and rewriting entire chunks. This approach avoids: (1) the rigidity of immutable storage, which requires workarounds like versioning or copy-on-write strategies; (2) the delays of asynchronous updates, where modified data is only visible after batch processing; (3) the performance hit of in-place mutability, which makes compressed storage prohibitively slow for frequent updates; and (4) the restrictions some databases have on not altering the segmentation or ordering keys.

3 Query optimizations

Real-time analytics isn't just about raw speed—it's about executing queries efficiently, reducing unnecessary work, and maximizing performance. TimescaleDB optimizes every step of the query lifecycle to ensure that queries scan only what's necessary, make use of data locality, and execute in parallel for sub-second response times over large datasets.

3.1 Skip unnecessary data

TimescaleDB minimizes the amount of data a query touches, reducing I/O and improving execution speed:

Primary partition exclusion (Figure 5a). Queries automatically skip irrelevant partitions (chunks) based on the primary partitioning key (usually a timestamp), ensuring they only scan relevant data. Applies to both rowstore and columnstore chunks.

Secondary partition exclusion (Figure 5b). Min/max metadata allows queries filtering on correlated dimensions (e.g., `order_id` or secondary timestamps) to exclude chunks that don't contain relevant data. Applies to columnstore chunks.

Batch-level filtering (Figure 5d). Within each chunk, compressed columnar batches are organized using `SEGMENTBY` keys and ordered by `ORDERBY` columns. Indexes and min/max metadata can be used to quickly exclude batches that don't match the query criteria. Applies to columnstore chunks.

3.2 Maximize locality

Organizing data for efficient access ensures queries are read in the most optimal order (Figure 6), reducing unnecessary random reads and reducing scans of unneeded data.

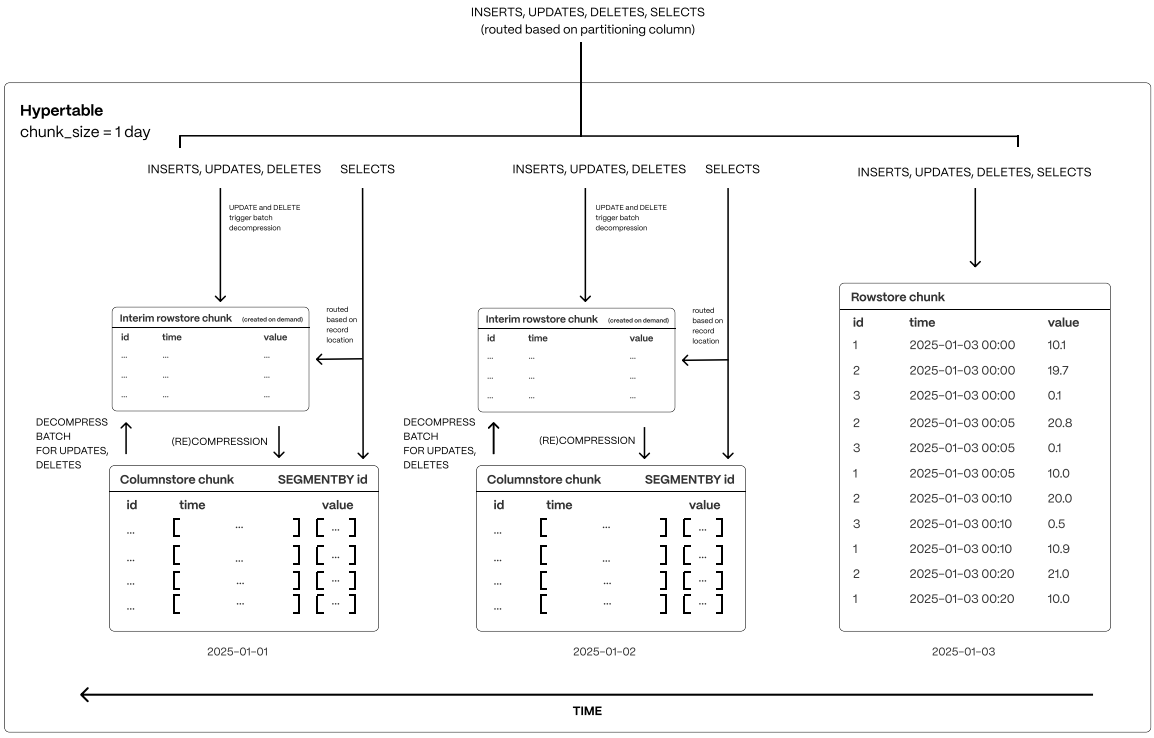


Figure 4. Hypercore supports real-time data mutability by routing inserts, updates, and deletes to transparently queryable intermim rowstore chunks. This design allows efficient modifications without compromising analytical performance, with compressed columnstore data only being updated during asynchronous background recompression.

- *Segmentation*: Columnar batches are grouped using `SEGMENTBY` to keep related data together, improving scan efficiency.
- *Ordering*: Data within each batch is physically sorted using `ORDERBY`, increasing scan efficiency (and reducing I/O operations), enabling efficient range queries, and minimizing post-query sorting.
- *Column selection*: Queries read only the necessary columns, reducing disk I/O, decompression overhead, and memory usage.

3.3 Parallelize execution

Once a query is scanning only the required columnar data in the optimal order, TimescaleDB is able to maximize performance through parallel execution. As well as using multiple workers, TimescaleDB accelerates columnstore query execution by using Single Instruction, Multiple Data (SIMD) vectorization, allowing modern CPUs to process multiple data points in parallel.

The TimescaleDB implementation of SIMD vectorization (Figure 7) currently supports several forms:

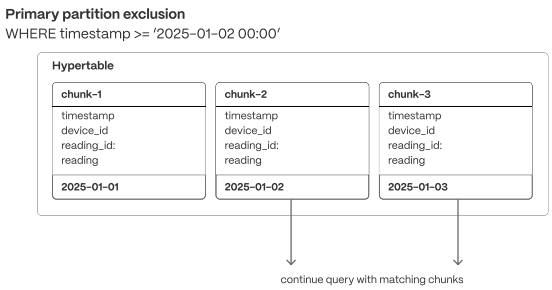
- *Vectorized decompression*, which efficiently restores compressed data into a usable form for analysis.
- *Vectorized filtering*, which rapidly applies filter conditions across data sets.
- *Vectorized aggregation*, which performs aggregate calculations, such as sum or average, across multiple data points concurrently.

4 Accelerating queries with continuous aggregates

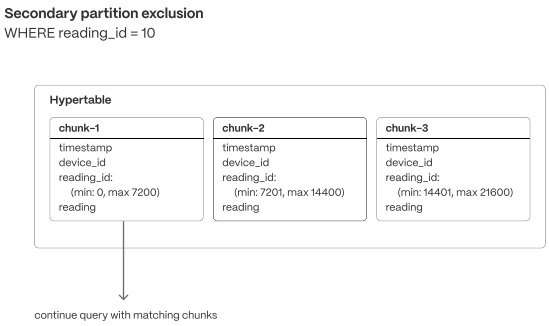
Aggregating large datasets in real time can be expensive, requiring repeated scans and calculations that strain CPU and I/O. While some databases attempt to brute-force these queries at runtime, compute and I/O are always finite resources—leading to high latency, unpredictable performance, and growing infrastructure costs as data volume increases.

Continuous aggregates, Timescale's implementation of incrementally updated materialized views, solve this by shifting computation from every query run to a single, asynchronous step after data is ingested (Figure 8). Only the time buckets that receive new or modified data are updated, and queries read precomputed results instead of scanning raw data—dramatically improving performance and efficiency.

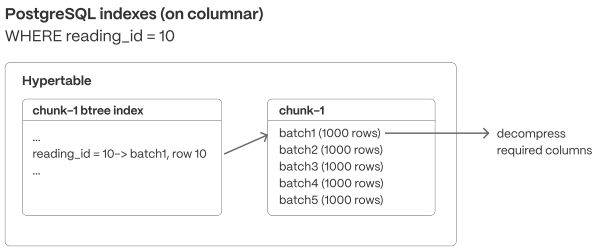
When you know the types of queries you'll need ahead of time, continuous aggregates allow you to pre-aggregate data along meaningful time intervals—such as per-minute, hourly,



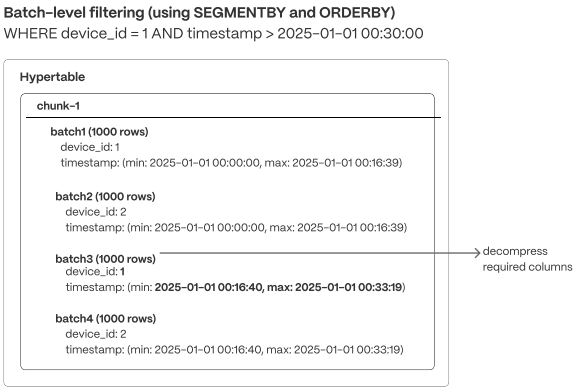
(a) TimescaleDB uses primary partition exclusion to skip entire chunks during query execution based on time filters. Only chunks whose time range intersects the query's `WHERE` clause are included, significantly improving query efficiency.



(b) Min/max metadata enables secondary partition exclusion on selected non-primary dimensions like `reading_id`. Queries skip chunks that fall outside the filtered range, further narrowing the data scanned and reducing query latency.



(c) TimescaleDB supports PostgreSQL-style indexes on columnstore data. Queries use these indexes to locate specific values and decompress only relevant batches, enabling fast lookups and selective scans.



(d) Segmented and ordered batches enable fine-grained filtering within chunks. Combined with min/max metadata for `ORDERBY` columns, queries can skip entire batches and only decompress those containing relevant values.

Figure 5. Query optimizations to skip unnecessary data.

or daily summaries—delivering instant results without on-the-fly computation.

Continuous aggregates also avoid the time-consuming and error-prone process of maintaining manual rollups, while continuing to offer data mutability to support efficient updates, corrections, and backfills. Whenever new data is inserted or modified in chunks which have been materialized, TimescaleDB stores invalidation records reflecting that these results are stale and need to be recomputed. Then, an asynchronous process re-computes regions that include invalidated data, and updates the materialized results. TimescaleDB tracks the lineage and dependencies between continuous aggregates and their underlying data, to ensure the continuous aggregates are regularly kept up-to-date. This happens in a resource-efficient manner, and where multiple invalidations can be coalesced into a single refresh (as opposed to refreshing any dependencies at write time, such as via a trigger-based approach).

Continuous aggregates themselves are stored in hypertables, and they can be converted to columnar storage for compression, and raw data can be dropped, reducing storage footprint and processing cost. Continuous aggregates also support hierarchical rollups (e.g., hourly to daily to monthly) and real-time mode (also shown in Figure 8), which merges precomputed results with the latest ingested data to ensure accurate, up-to-date analytics.

This architecture enables scalable, low-latency analytics while keeping resource usage predictable—ideal for dashboards, monitoring systems, and any workload with known query patterns.

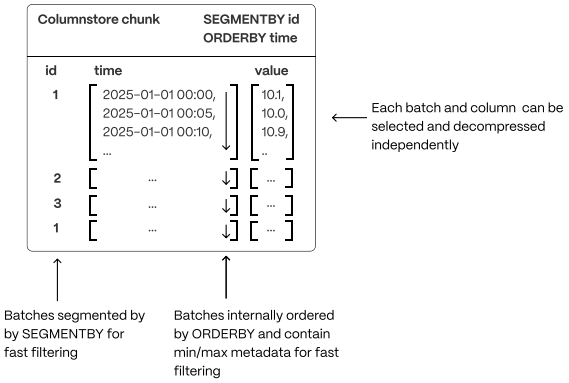


Figure 6. Columnstore batches are segmented and ordered using `SEGMENTBY` and `ORDERBY` keys. Each batch carries metadata to enable selective decompression, allowing queries to efficiently scan and filter only the necessary data.

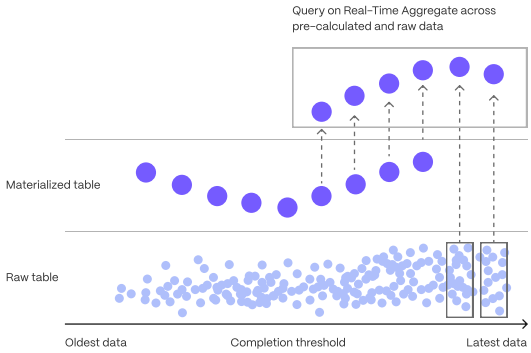


Figure 8. Continuous aggregates extend PostgreSQL materialized views with incremental updates and real-time merging of raw and precomputed data, enabling fast, always-fresh analytics.

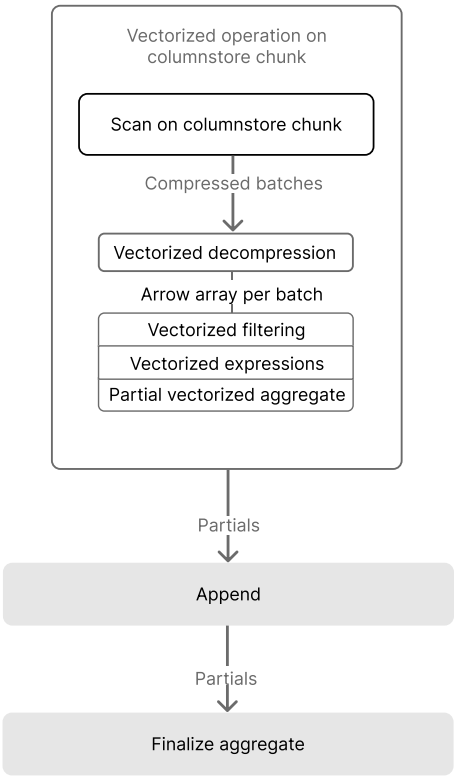


Figure 7. TimescaleDB employs SIMD vectorization when processing columnar batches.

4.1 Hyperfunctions for real-time analytics

Real-time analytics requires more than basic SQL functions; efficient computation is essential as datasets grow in size and complexity. TimescaleDB provides ***hyperfunctions***, available through the `timescaledb_toolkit` extension, as high-performance, SQL-native functions tailored for time-series analysis. These include advanced tools for gap-filling, percentile estimation, time-weighted averages, counter correction, and state tracking, among others.

A key innovation of hyperfunctions is their support for partial aggregation, which allows TimescaleDB to store intermediate computational states rather than just final results. These partials can later be merged to compute rollups efficiently, avoiding expensive reprocessing of raw data and reducing compute overhead. This is especially effective when combined with continuous aggregates.

Consider a real-world example: monitoring request latencies across thousands of application instances. You might want to compute p95 latency per minute, then roll that up into hourly and daily percentiles for dashboards or alerts. With traditional SQL, calculating percentiles requires a full scan and sort of all underlying data—making multi-level rollups computationally expensive.

With Timescale, you can use the `percentile_agg` hyperfunction in a continuous aggregate to compute and store a partial aggregation state for each minute. This state efficiently summarizes the distribution of latencies for that time bucket, without storing or sorting all individual values. Later, to produce an hourly or daily percentile, you simply combine the stored partials—no need to reprocess the raw latency values.

This approach provides a scalable, efficient solution for percentile-based analytics. By combining hyperfunctions with continuous aggregates, TimescaleDB enables real-time systems to deliver fast, resource-efficient insights across high-ingest, high-resolution datasets, without sacrificing accuracy or flexibility.

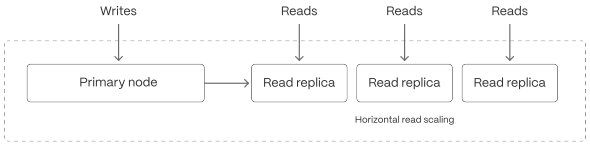


Figure 9. Tiger Cloud supports horizontal read scaling by replicating data across read replicas. This allows analytical workloads to be distributed across multiple nodes, isolating them from ingest traffic and improving performance.

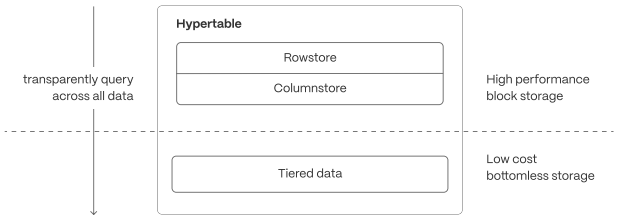


Figure 10. Tiger Postgres transparently queries across tiered data, spanning rowstore, columnstore, and object storage. Recent, high-velocity data stays in fast-access storage, while older data is compressed and tiered for cost efficiency.

5 Cloud-native architecture

Real-time analytics requires a scalable, high-performance, and cost-efficient database that can handle high-ingest rates and low-latency queries without overprovisioning. Tiger Cloud is designed for elasticity, enabling independent scaling of storage and compute, workload isolation, and intelligent data tiering.

5.1 Independent storage and compute scaling

Real-time applications generate continuous data streams while requiring instant querying of both fresh and historical data. Traditional databases force users to pre-provision fixed storage, leading to unnecessary costs or unexpected limits. Tiger Cloud eliminates this constraint by dynamically scaling storage based on actual usage:

- Storage expands and contracts automatically as data is added or deleted, avoiding manual intervention.
- Usage-based billing ensures costs align with actual storage consumption, eliminating large upfront allocations.
- Compute can be scaled independently to optimize query execution, ensuring fast analytics across both recent and historical data.

With this architecture, databases grow alongside data streams, enabling seamless access to real-time and historical insights while efficiently managing storage costs.

5.2 Workload isolation for real-time performance

Balancing high-ingest rates and low-latency analytical queries on the same system can create contention, slowing down performance. Tiger Cloud mitigates this by allowing read and write workloads to scale independently (Figure 9):

- The primary database efficiently handles both ingestion and real-time rollups without disruption.
- Read replicas scale query performance separately, ensuring fast analytics even under heavy workloads.

This separation ensures that frequent queries on fresh data don't interfere with ingestion, making it easier to support live monitoring, anomaly detection, interactive dashboards, and alerting systems.

5.3 Intelligent data tiering for cost-efficient real-time analytics

Not all real-time data is equally valuable—recent data is queried constantly, while older data is accessed less frequently. Tiger Postgres extends TimescaleDB to allow automatic tiering of data to cheaper bottomless object storage (Figure 10), ensuring that hot data remains instantly accessible, while historical data is still available.

With such tiering, *recent, high-velocity data* stays in high-performance storage for ultra-fast queries, while *older, less frequently accessed data* is automatically moved to cost-efficient object storage but remains queryable and available for building continuous aggregates.

While many systems support this concept of data cooling, Tiger Postgres ensures that the data can still be queried from the same hypertable regardless of its current location. For real-time analytics, this means applications can analyze live data streams without worrying about storage constraints, while still maintaining access to long-term trends when needed.

5.4 Cloud-native database observability

Real-time analytics doesn't just require fast queries—it requires the ability to understand why queries are fast or slow, where resources are being used, and how performance changes over time. That's why Tiger Cloud is built with deep observability features, giving developers and operators full visibility into their database workloads.

At the core of this observability is Insights, Timescale's built-in query monitoring tool. Insights captures per-query statistics from our whole fleet in real time, showing you exactly how your database is behaving under load. It tracks key metrics like execution time, planning time, number of rows read and returned, I/O usage, and buffer cache hit rates—not just for the database as a whole, but for each individual query.

Insights lets you do the following:

- Identify slow or resource-intensive queries instantly;
- Spot long-term performance regressions or trends;

- Understand query patterns and how they evolve over time;
- See the impact of schema changes, indexes, or continuous aggregates on workload performance; and
- Monitor and compare different versions of the same query to optimize execution.

All this is surfaced through an intuitive interface, available directly in Tiger Cloud, with no instrumentation or external monitoring infrastructure required.

Beyond query-level visibility, Tiger Cloud also exposes metrics around service resource consumption, compression, continuous aggregates, and data tiering, allowing you to track how data moves through the system—and how those background processes impact storage and query performance.

Together, these observability features give you the insight and control needed to operate a real-time analytics database at scale, with confidence, clarity, and performance you can trust.

6 Ensuring reliability and scalability

Maintaining high availability, efficient resource utilization, and data durability is essential for real-time applications. Tiger Cloud provides robust operational features to ensure seamless performance under varying workloads.

- *High-availability (HA) replicas*: deploy multi-AZ HA replicas to provide fault tolerance and ensure minimal downtime. In the event of a primary node failure, replicas are automatically promoted to maintain service continuity.
- *Connection pooling*: optimize database connections by efficiently managing and reusing them, reducing overhead and improving performance for high-concurrency applications.
- *Backup and recovery*: leverage continuous backups, Point-in-Time Recovery (PITR), and automated snapshotting to protect against data loss. Restore data efficiently to minimize downtime in case of failures or accidental deletions.

These operational capabilities ensure Tiger Cloud remains reliable, scalable, and resilient, even under demanding real-time workloads.

7 Conclusion

Real-time analytics is critical for modern applications, but traditional databases struggle to balance high-ingest performance, low-latency queries, and flexible data mutability. Tiger Postgres extends PostgreSQL to solve this challenge, combining automatic partitioning, hybrid row-columnar storage, and intelligent compression to optimize both transactional and analytical workloads.

With continuous aggregates, hyperfunctions, and advanced query optimizations, TimescaleDB ensures sub-second queries even on massive datasets that combine current and historic data. Its cloud-native architecture further enhances scalability with independent compute and storage scaling, workload isolation, and cost-efficient data tiering—allowing applications to handle real-time and historical queries seamlessly.

For developers, this means building high-performance, real-time analytics applications without sacrificing SQL compatibility, transactional guarantees, or operational simplicity.

Tiger Data delivers the best of PostgreSQL, optimized for real-time analytics.

Last updated: **Jun 17, 2025**