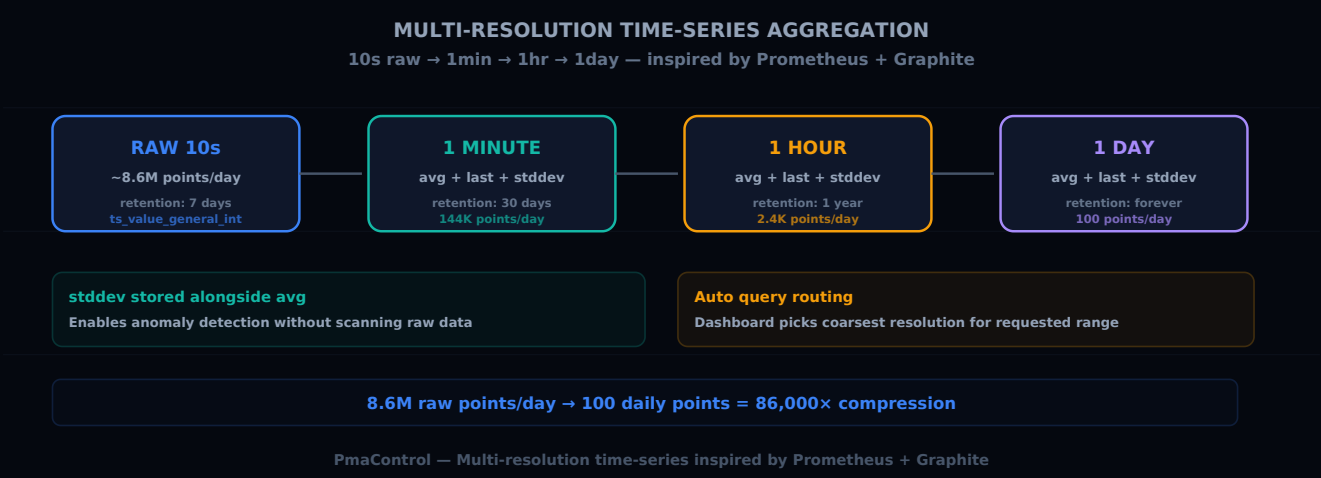
Time-Series Aggregation: From Millions of Raw Points to Fast Queries

Aurélien LEQUOY · March 21, 2026

PMACONTROL TIME-SERIES AGGREGATION MONITORING ARCHITECTURE



The Raw Volume

PmaControl collects metrics from each monitored MariaDB / MySQL instance every **10 seconds**. For each server, this represents:

- 6 points per minute
- 360 points per hour
- 8,640 points per day
- **60,480 points per week**

With 100 servers and 50 metrics per server:

```
100 servers × 50 metrics × 8,640 points/day = 43,200,000 points/day
```

43 million points per day. 302 million per week. Over one billion per month.

Storing all of this at raw resolution (10 seconds) indefinitely is technically possible but practically useless. Nobody looks at a graph with 10-second resolution on data from 6 months ago. And queries scanning millions of rows to display a year-long graph are slow and expensive.

The Inspiration: Prometheus and Graphite

The problem is not new. Two systems have solved it elegantly:

- **Prometheus** with its **recording rules**: pre-computed PromQL queries that aggregate raw data into derived metrics at regular intervals
- **Graphite** with its **Whisper** format: a multi-resolution retention system where data is automatically aggregated as it ages

PmaControl draws from both approaches to design its own aggregation system.

The Multi-Resolution Schema

Four resolution levels:

Level	Interval	Retention	Estimated Volume (100 srv)
Raw	10 seconds	7 days	302M points/week
1 minute	1 minute	30 days	216M points/month
1 hour	1 hour	1 year	43.8M points/year
1 day	1 day	Indefinite	1.8M points/year

The total volume stored at any given time (with 100 servers):

```
Raw (7 days):      302M points
1min (30 days):    216M points
1hr (1 year):       43.8M points
1day (all):          ~2M points
Total:             ~564M points
```

Without aggregation, keeping one year of raw data would represent **15.8 billion points**. Aggregation reduces storage by a factor of 28.

What Gets Stored at Each Aggregated Level

For each aggregated point, three values are stored:

```
CREATE TABLE ts_aggregated_1min (
    server_id     INT,
    metric_id     INT,
    timestamp     DATETIME,
    last_value    DOUBLE,    -- last value in the interval
    avg_value     DOUBLE,    -- average over the interval
    stddev_value  DOUBLE,    -- standard deviation over the interval
    PRIMARY KEY (server_id, metric_id, timestamp)
);
```

Why `last_value`?

For counter-type metrics (query count, bytes sent), the last value in the interval is often more relevant than the average. It represents the most recent state.

Why `avg_value`?

For gauge-type metrics (CPU usage, memory, active threads), the average is the most faithful representation of behavior over the interval.

Why `stddev_value`? The Key Insight

This is the main innovation in this design. **Storing standard deviation alongside the average enables anomaly detection without raw data.**

Consider two hours with the same 45% average CPU:

- **Hour A**: CPU stable between 42% and 48%. `avg=45%, stddev=2%`
- **Hour B**: CPU oscillating between 5% and 85%. `avg=45%, stddev=28%`

Without the stddev, these two hours are indistinguishable in aggregated data. With the stddev, Hour B is immediately identifiable as abnormal.

This enables building alerts based on historical stddev:

```
IF current_stddev > 3 × average_stddev_last_30_days
THEN alert: abnormal behavior detected
```

The Aggregation Process

Aggregation works in a cascade, driven by a cron job:

Step 1: Raw to 1 Minute

Every minute, a worker reads the last 6 raw points for each (server, metric) pair and computes:

```
INSERT INTO ts_aggregated_1min (server_id, metric_id, timestamp, last_value, avg_value,
stddev_value)
SELECT
    server_id,
    metric_id,
    DATE_FORMAT(timestamp, '%Y-%m-%d %H:%i:00') AS minute,
    -- last_value: subquery for the last point
    (SELECT value FROM ts_raw r2
     WHERE r2.server_id = ts_raw.server_id
       AND r2.metric_id = ts_raw.metric_id
       AND r2.timestamp >= DATE_FORMAT(ts_raw.timestamp, '%Y-%m-%d %H:%i:00')
       AND r2.timestamp < DATE_FORMAT(ts_raw.timestamp, '%Y-%m-%d %H:%i:00') + INTERVAL 1
MINUTE
     ORDER BY r2.timestamp DESC LIMIT 1),
    AVG(value),
    STDDEV(value)
FROM ts_raw
WHERE timestamp >= NOW() - INTERVAL 1 MINUTE
GROUP BY server_id, metric_id, minute;
```

Step 2: 1 Minute to 1 Hour

Every hour, a worker aggregates the 60 one-minute points into a single one-hour point. The combined stddev calculation uses the pooled variance formula:

```
σ_combined = sqrt( mean(σ²_i) + var(μ_i) )
```

Where `σ_i` are the stddevs of the sub-intervals and `μ_i` their means. This formula is mathematically exact and does not need raw data.

Step 3: 1 Hour to 1 Day

Same principle, once a day, 24 one-hour points become a single one-day point.

Step 4: Purge of Old Data

After each aggregation, data beyond the retention window is deleted:

```
DELETE FROM ts_raw WHERE timestamp < NOW() - INTERVAL 7 DAY;
DELETE FROM ts_aggregated_1min WHERE timestamp < NOW() - INTERVAL 30 DAY;
DELETE FROM ts_aggregated_1hr WHERE timestamp < NOW() - INTERVAL 1 YEAR;
-- ts_aggregated_1day: never purged
```

Query Routing

When the PmaControl dashboard displays a graph, it must choose the right resolution. The principle is simple: **use the coarsest resolution that covers the requested range**.

```
function selectResolution(int $timeRangeSeconds): string {
    if ($timeRangeSeconds <= 3600) {       // <= 1 hour
        return 'ts_raw';                   // 10s resolution
    } elseif ($timeRangeSeconds <= 86400 * 2) {  // <= 2 days
        return 'ts_aggregated_1min';       // 1min resolution
    } elseif ($timeRangeSeconds <= 86400 * 90) { // <= 90 days
        return 'ts_aggregated_1hr';        // 1hr resolution
    } else {
        return 'ts_aggregated_1day';       // 1day resolution
    }
}
```

Result: a one-year graph loads only **365 points** (1-day resolution) instead of 3.1 million (10-second resolution). The query goes from several seconds to a few milliseconds.

Impact on Queries

Requested Range	Resolution	Points Loaded	Query Time
1 hour	10s (raw)	360	< 10 ms
24 hours	1 min	1,440	< 20 ms
30 days	1 hour	720	< 15 ms
1 year	1 day	365	< 10 ms

Query times become **independent of the time range**. A one-year graph is as fast as a one-hour graph.

Anomaly Detection with Stored stddev

Thanks to the pre-computed stddev, PmaControl can detect anomalies on aggregated data without going back to raw data:

1. **Baseline calculation**: average and stddev of the stddev over the last 30 days for each metric
2. **Comparison**: the current hour's stddev is compared to the baseline
3. **Alert**: if the stddev exceeds 3 times the baseline, abnormal behavior

Concrete example:

- Baseline for threads_running: `avg_stddev = 2.1, stddev_stddev = 0.8`
- Current hour: `stddev = 14.3`
- Score: `(14.3 - 2.1) / 0.8 = 15.25` sigmas — **certain anomaly**

This mechanism detects anomalies that simple average monitoring would miss: a server whose CPU oscillates wildly but always returns to a correct average.

Conclusion

Multi-resolution aggregation is the key to managing time-series data at scale. Storing stddev alongside the average is an uncommon but powerful design choice: it preserves information about variability, enabling anomaly detection even on aggregated data.

With this system, PmaControl can monitor 100+ MariaDB / MySQL servers over a full year while guaranteeing dashboard queries under 20 milliseconds.