



RAPPORTI TECNICI INGV

A Time Series Database for Continuous
Gravimetric Data Powered by
PostgreSQL, TimescaleDB, and Grafana



ISTITUTO NAZIONALE DI GEOFISICA E VULCANOLOGIA

500

Direttore Responsabile

Daniela VERSACE

Editor in Chief

Milena MORETTI

Editorial Board

Laura ALFONSI

Christian BIGNAMI

Simona CARANNANTE

Viviana CASTELLI

Luca COCCHI

Luigi CUCCI

Lorenzo CUGLIARI

Alessia DI CAPRIO

Roberto DI MARTINO

Domenico DI MAURO

Domenico DORONZO

Filippo GRECO

Alessandro IAROCCI

Mario MATTIA

Daniele MELINI

Anna NARDI

Lucia NARDONE

Marco OLIVIERI

Pierangelo ROMANO

Maurizio SOLDANI

Sara STOPPONI

Umberto TAMMARO

Andrea TERTULLIANI

Stefano URBINI

REGISTRAZIONE AL TRIBUNALE DI ROMA N.173 | 2014, 23 LUGLIO

© INGV Istituto Nazionale di Geofisica e Vulcanologia

Rappresentante legale: Presidente INGV

Sede: Via di Vigna Murata, 605 | Roma

Tutti i diritti riservati

www.ingv.it



ISTITUTO NAZIONALE DI GEOFISICA E VULCANOLOGIA

RAPPORTI TECNICI INGV

A Time Series Database for Continuous Gravimetric Data Powered by PostgreSQL, TimescaleDB, and Grafana

Luca Timoteo Mirabella*, Alfio Messina, Danilo Contrafatto

INGV | Istituto Nazionale di Geofisica e Vulcanologia, Sezione di Catania - Osservatorio Etneo

**Corresponding author*

Accepted 26 May 2025 | *Accettato 26 maggio 2025*

Come citare | *How to cite* Mirabella L.T., Messina A., Contrafatto D., (2025). A Time Series Database for Continuous Gravimetric Data Powered by PostgreSQL, TimescaleDB, and Grafana. Rapp. Tec. INGV, 500: 1-36, <https://doi.org/10.13127/rpt/500>

Cover Adobe Stock standard license, file n. 548580099 | *In copertina Adobe Stock licenza standard, file n. 548580099*

500

INDEX

Abstract	7
Introduction	7
1. Software tools for time series data handling	8
1.1 TimescaleDB	9
1.1.1 Hypertables and chunks	10
1.1.2 Column native compression	11
1.2 Grafana	12
1.2.1 Data sources and queries	13
1.2.2 Dashboard and Panels	13
2. Datasets	14
3. Database	14
3.1 Design	14
3.1.1 Conceptual model	15
3.1.2 Logical model	15
3.1.3 Physical model	17
3.2 Implementation	18
3.2.1 Installing self-hosted TimescaleDB and PostgreSQL	19
3.2.2 PostgreSQL - Creating database and tables	20
3.2.3 PostgreSQL - Managing users, passwords and access restrictions	20
3.2.4 Grafana - Installation and configuration	21
3.3 Optimization	22
3.3.1 PostgreSQL - Change the default data directory	22
3.3.2 PostgreSQL - Tuning configuration parameters	23
3.3.3 PostgreSQL - Set up TimescaleDB hypertables	23
3.3.4 Grafana - Tuning configuration parameters	24
4. Developed scripts for data injection	24
4.1 Historical data	25
4.2 Corrections scripts	25
4.2.1 Synthetic Earth Tide	25
4.2.2 Gaps	25
4.2.3 Jumps	26
4.3 Real-time data	26
5. System test and performance	27
Conclusions and remarks	30
References	30
Sitography	31

Abstract

Mt. Etna, the most active volcano in Europe, is extensively monitored by the Osservatorio Etneo (OE), of the Istituto Nazionale di Geofisica e Vulcanologia (INGV), using various geophysical sensors, including gravimeters. This paper focuses on the design and implementation of a system for storing continuous gravimetric data recorded by three permanent stations on Mt. Etna, powered by GWR iGrav Superconducting Gravimeters. These devices provide high-precision gravity measurements along with numerous internal and external parameters required for data correction and interpretation.

The continuous data from these stations generate large datasets stored as daily ASCII-encoded TSF files, synchronized hourly on a central server. To optimize data storage and retrieval, a time series database system was developed using TimescaleDB, built on PostgreSQL. This approach overcomes the limitations of traditional ASCII file storage and addresses the challenges of managing time series data. TimescaleDB's capabilities, such as hypertables, chunks, and native column compression, significantly enhance the efficiency and scalability of data handling.

Additionally, the implementation of Grafana provides an efficient web-based solution for interactive data visualization and monitoring. The OE Gravimetric Database leverages these technologies to enable quick and efficient access to time series data, facilitating comprehensive analysis and real-time monitoring.

The paper details the design choices, implementation steps, and optimization techniques employed in developing this advanced database system, demonstrating its effectiveness in managing a huge amount of continuous real-time data.

Keywords Time series Database; PostgreSQL; Timescale DB; Grafana

Introduction

Mt. Etna is the most active volcano in Europe and also one of the best monitored. Permanent networks of geophysical sensors, such as seismometers, GPSes, tiltmeters, gas samplers, cameras and ash radars, are deployed all around the volcano to monitor, in real-time, the state of its activity. This leads to a huge quantity of continuous data to be stored at the Osservatorio Etneo (OE), the nearest department of Istituto Nazionale di Geofisica e Vulcanologia (INGV).

The focus of this paper is the description of a system specifically designed and implemented to store continuous gravimetric data recorded by a network of three permanent stations, installed at different elevations on Mt Etna (Figure 1). These stations are powered by GWR iGrav devices [1], based on “superconducting persistent currents” technology, which makes them among the most stable (drift less than 0.5 μ Gal/month) and precise (1 nanoGal) gravimeters. They record a continuous relative gravity value (at 1 Hz), but also a long series of internal and external parameters, which are necessary to correct and interpret the gravimetric variations. This network is well described in Carbone et al. [2019].

Each gravimetric station produces large datasets, stored as daily TSF files, which are ASCII encoded. The structure of this file format is well described in the Datasets chapter. Daily data files are regularly synchronized (every hour) on a central server at OE. The considerable amount of continuous data produced by these stations required a deep study of state-of-the-art techniques for efficient continuous data storage. The optimization of data retrieving was also taken into account, in order to provide users with the possibility to request and retrieve time

sequences over the entire dataset, in a quick yet efficient way.

The strategy pursued is based on TimescaleDB [2], a time series database built on the popular relational database (Relational DataBase Management System, RDBMS) PostgreSQL [3]. Exploiting a RDBMS allowed to overcome the limits of ASCII files storage, which are definitely not efficient, in terms of storage occupation and handling of long temporal sequences. Nevertheless, “usual” RDBMS, such as PostgreSQL, MySQL, Oracle Database, etc., are poorly suited to time series data which develops over long periods. TimescaleDB was designed to get through this limit, due to several techniques and optimizations which make it ideal for time series. The usage of a proper database management system allowed also the application of an efficient web solution for interactive data retrieval, visualization and monitoring. The name of this open-source tool is Grafana [4].

This paper describes the database system for continuous gravimetric data realized at OE, reporting the main design choices which guided its development.



Figure 1 Network of superconducting gravimeters installed on Mt Etna. Each blue point (except the one that indicates the craters position) represents a permanent gravimetric station for continuous data acquisition.

1. Software tools for time series data handling

Time series data refer to data that capture the temporal evolution of a system, where each parameter is continuously monitored and recorded over time. Examples include temperature measurements from a sensor and workload metrics of a computational host. Time series data have several distinctive features. They are characterized by a natural temporal ordering, so each record has a timestamp (Figure 2). Time series records are mostly inserted in batches, not requiring further updates. Lastly, queries on them are very often aimed at selecting records over specific time periods.

Time series data pile up very quickly and relational databases (RD hereafter) are commonly considered unsuited for them. Oracle, MySQL, Microsoft SQL Server and PostgreSQL are among the most popular RDs [5]. They are based on SQL (Structured Query Language) model which implies a convenient secondary key management, the possibility to join data on different tables and an easy, but powerful, query language. Conversely, RDs are difficult to scale - scalability being the system's ability to handle increasing amounts of data or users while maintaining acceptable performance - leading developers to adopt NoSQL (Not only SQL) databases for time series data management. The latter are non-relational DBs because they are not based on the

relational data model or in other words, data are not stored in table format [Mukherjee, 2019]. MongoDB, Apache Cassandra and Apache HBase are the most popular NoSQL databases [6]. NoSQL databases have a distributed structure, so they easily scale horizontally (on more computational nodes). Basically, scaling can be thought of at two levels: *scaling up* on a single machine, so that it can store more data, and *scaling out* across multiple hosts, in a distributed data storage configuration.

Conventional (relational) databases are commonly considered difficult to scale essentially due to the cost/performance trade-off between memory and disk. While memory is faster than disk, it is much more expensive. This is the reason why on a workstation you could have “only” 64 GB of RAM and, on the other hand, 4 TB of storage. Generally, the entire dataset doesn’t fit into memory, so data tables and indexes are written to disk. Only the working set of data can be kept in memory. When a query involves a new dataset, a significant disk I/O is required for data transfer between disk and memory, thus affecting the overall performance. These concepts are clearly explained in the TimescaleDB blog [7].

		Columns - Parameters		
Rows - Records	Timestamp	Sensor ID	Value 1	Value 2
	2023-08-10 06:10:40	iG016	13.6	141.2
	2023-08-10 06:10:40	iG020	12.7	137.5
	2023-08-10 06:10:40	iG025	15.1	147.5
	2023-08-10 06:10:50	iG016	13.7	145.0
	2023-08-10 06:10:50	iG020	13.0	150.2
	2023-08-10 06:10:50	iG025	15.9	161.1

Figure 2 Example of time series data represented as a table modeled by data source, reporting different observations, i.e. different recording times, in rows and recorded values in columns, one for each parameter.

1.1 TimescaleDB

TimescaleDB is an open-source software, released in a first beta version in April 2017. It demonstrates that RDs can be a valid option for time series data, through the adoption of several optimizations, overcoming the scaling issue. It is engineered from PostgreSQL and packaged as an extension of it. This design allows TimescaleDB to take advantage of PostgreSQL’s reliability, robustness and security.

TimescaleDB introduces advanced capabilities specifically designed for time series data. The latter have distinctive features, such as they are inserted (written) and rarely updated, all the records have a timestamp and the latest records inserted are typically the most recent ones. The adoption of hypertables and chunks as well as a particular compression technique are the main special features introduced by TimescaleDB.

A valuable benchmark of TimescaleDB was carried out by Kammakomati [2021] in a project involving the development of new solutions for Supervisory Control and Data Acquisition (SCADA) systems at CERN (almost 700 systems). SCADA systems monitor the states of sensors and actuators, archiving each state value, called signal, and a series of alarm states that can be defined against the value of a signal. One of these systems can generate tens of millions events per day, to be stored on databases, to allow operators to see the history of signals and alarms. This important study revealed that an ingestion rate of 80K rows/second was obtained for TimescaleDB on a single node architecture and 150K rows/second for TimescaleDB on a two-node configuration, making them 2X and 3X higher than for a “standard” PostgreSQL node (around 40K rows/second). Another valuable result of this research was the observation of a

considerable decline in the ingestion rate for PostgreSQL, during the benchmark run, due to the increasing number of records stored in the DB. On the other hand, the ingestion rate was stable for TimescaleDB, highlighting its optimizations with respect to the original PostgreSQL on which it is based.

Mostafa et al. [2022] developed SciTS, a benchmark for time series databases, and exploited it to evaluate the performance of four database systems: ClickHouse, InfluxDB, TimescaleDB, and PostgreSQL. In light of their benchmarks, TimescaleDB confirms to be a huge improvement over PostgreSQL. It can hold a high ingestion rate of time series data, around 490K records/second on their setup, optimizing the usage of system resources with better disk I/O and efficient CPU and virtual memory usage. Also, TimescaleDB offers much better query latencies than PostgreSQL.

The good performance reported in these studies persuaded authors to exploit TimescaleDB for the implementation of Continuous Gravimetric Database, at INGV - OE.

1.1.1 Hypertables and chunks

The table abstraction provided by TimescaleDB is named hypertable [8]. It represents the primary point of interaction between a user and its dataset stored in TimescaleDB. Hypertables adoption is completely transparent for the user who can query them via standard SQL functions (insert, update, delete, alter table and so forth). It is only a virtual view and behind it there are several “real” tables, where data is recorded, called chunks (Figure 3).

Data into hypertables are partitioned by the values of a time column, which is usually a timestamp. Each of these partitions is a chunk, which can be considered a child table of a hypertable, which is the parent. On the basis of a time interval, which can be configured, all the rows that have a timestamp within a specific time period are written within the same chunk. These partitions are non-overlapping, so that all chunks in a hypertable are disjoint. When an INSERT INTO (insertion of new data on the DB) process involves new records whose timestamps are outside time periods covered by existing chunks, TimescaleDB creates a new chunk, automatically. Otherwise, it determines the proper chunk for the insertion of new rows. Behind the scenes, a chunk is implemented using a standard PostgreSQL table.

TimescaleDB supports multi-dimensional partitioning, allowing additional columns, such as device ID, to be considered for chunk definition. This approach, referred to by TimescaleDB developers as time-and-space partitioning, is primarily designed for distributed hypertables, where time intervals are further partitioned across multiple hosts.

Performance of database operations are strictly related to knowledge of chunks time intervals. Standard SQL statements, such as INSERT INTO and SELECT on a hypertable, are indeed internally redirected to the proper chunks, transparently for the user. This chunk-based architecture is the key for TimescaleDB high efficiency in time series data management, making possible a high scalability of PostgreSQL databases. Essentially, chunks should be configured to fit in main memory. This means that recent chunks, containing recent data, can be accessed in the fastest way, rather than from disk. TimescaleDB uses local indexes, built on each chunk, rather than global on the entire hypertable. This ensures that indexes are kept in memory together with chunks they are related to.

TimescaleDB can also manage the data retention, according to a policy defined by the user. For example, the oldest chunks could be automatically deleted based on their time ranges.

Moreover, data reordering and compression functions can be set up. The former consists in chunk asynchronous rewriting when new rows are added to an existing chunk. Reordering can be based on other table fields, such as the device ID, other than on the timestamp. Consequently, queries on a specific device in a specific time interval become much faster.

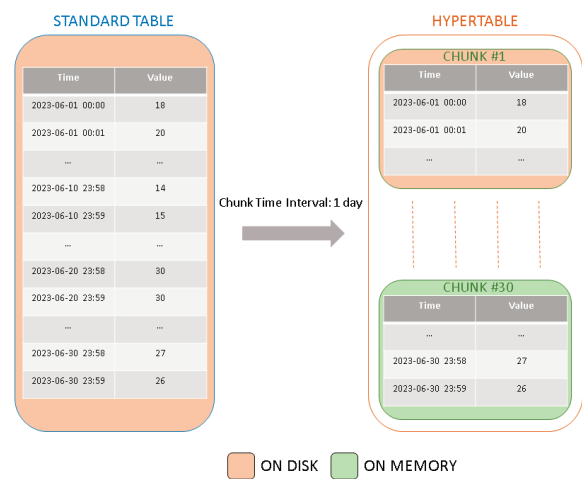


Figure 3 A hypertable is a virtual view which can be transparently accessed by the user through standard SQL queries. Data into hypertables is partitioned according to timestamps; each partition is a “real” table called chunk.

1.1.2 Column native compression

Chunked data allows to improve performance and it is the basis for several data-management capabilities. The latest chunks naturally remain in memory, ensuring fast inserts and performant queries of recent data. Nevertheless, also less recent chunks, on disk, have to be stored according to high performance requirements. TimescaleDB includes a high-efficiency compression feature exploiting specific algorithms which are applied to table columns [9]. Chunks are converted from their row-major scheme (when they are in memory) into a columnar form, making possible the application of a type-specific compression on each column. Table 1 reports an example table in the typical row-major form.

Time	First_param	Second_param	Device_code
2022-09-10 09:00:10	110.5	25.3	1
2022-09-10 09:00:10	120.1	27.9	2
2022-09-10 09:00:00	107.3	25.8	1
2022-09-10 09:00:00	115.6	28.1	2

Table 1 Example of a typical database table form.

In order to enable compression on a hypertable, the user has to declare which column to segment by. Applying the TimescaleDB compression at the example in Table 1, selecting Device_code as “segment-by” column, results in a new table with a single row for each Device_code and arrays of values for each original column. The result is represented in Table 2. This technique allows a storage occupation reduction between 94% and 97% while also making faster some queries on massive amounts of historical time series data.

Time	First_param	Second_param	Device_code
[2022-09-10 09:00:10, 2022-09-10 09:00:00]	[110.5, 107.3]	[25.3, 25.8]	1
[2022-09-10 09:00:10, 2022-09-10 09:00:00]	[120.1, 115.6]	[27.9, 28.1]	2

Table 2 Example of TimescaleDB compressed hypertable.

To enable this feature, a compression policy interval, i.e. a time interval after which the oldest data are compressed, has to be configured. This choice is crucial and may vary depending on the context. Generally, queries on recent data (shallow in time) are wider in columns than queries on older data which becomes more analytical, involving less columns. For wide and shallow queries, the uncompressed data, in row-based format, is the best option, while deep and narrow queries are faster on compressed arrays of data.

Nevertheless, data compression comes with some limitations: SQL DELETE actions cannot be performed on compressed data as well as the reordering of out-of-order data is not possible. Definitely, the choice of a good compression policy interval can make queries more efficient, allowing also a certain interval of uncompressed data for wider queries and data reordering.

1.2 Grafana

Grafana is an open-source visualization and analytics platform released in 2014. It targeted time series databases like InfluxDB and OpenTSDB but evolved to support even relational databases like MySQL and PostgreSQL. Grafana's goal is to allow users to build, analyze, and monitor their data from multiple sources such as internal systems, cloud services, and databases. The platform is highly configurable and provides powerful tools to easily query, visualize, and alert on data. Additionally, Grafana offers a wide range of integrations and plugins that allow users to further tailor their dashboards, reports, and analytics to meet their unique needs. Grafana is a web application, so it's easily accessible from many users of a local network as well as from the Internet. In [10] the author investigates the performance and the scalability of Grafana, in order to evaluate its effectiveness in real-time monitoring and visualization in large-scale and distributed systems. He also provides a list of different visualization tools, such as Tableau, Power BI and Kibana, reporting for each one its main features. In conclusion, the author lists pros and cons of using Grafana; among the former, the wide range of input data sources, the user-friendly interface and the variety of graphs, charts and maps. Among cons, the alerting feature which is not as advanced as some commercial products and the improvable scalability.

In their book, Reddy Yeruva and Basavegowda Ramu [2023] highlight the need for in-depth understanding of modern complex systems. Observability is a critical aspect and a simple monitoring of individual components is no longer sufficient to gain a comprehensive view of a complex system. They introduce Grafana as a key to the realm of observability, which empowers engineers to have real-time visibility into their systems. Therefore, it's a valuable support for the decisional process and to address any issue that may arise.

Sierra-Fernández et al. [2021] propose a system to monitor the instantaneous frequency in power systems, which is a key parameter to assess the quality of the electricity supply. They developed an instrument based on a data acquisition module, equipped with a GPS receiver for accurate timing. The computation is carried out by a Python [11] module specifically developed, while Grafana was exploited for the visualization of the stored data (not only instantaneous frequency). This work is part of an emerging trend of developing new low-cost systems based on open-source tools, such as Python and Grafana. They highlight the advantages of the visualization of the data within a web environment which results to be immediate and interactive, and therefore, that can be carried out by an operator with a very basic training.

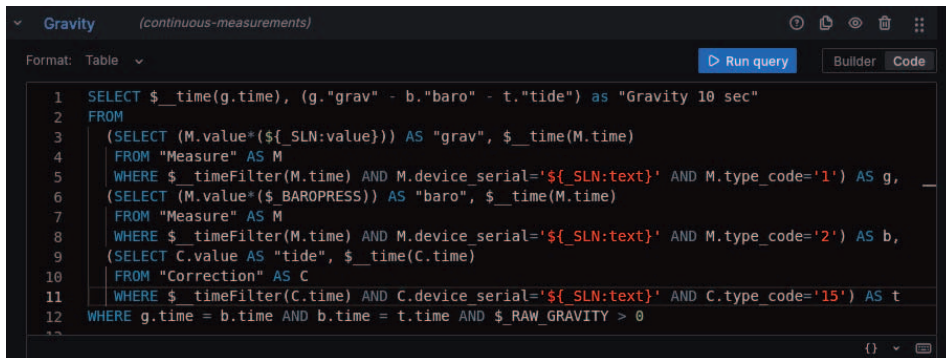
Grafana is a web-service which can be hosted in a company machine (Grafana open source and Enterprise) or in the cloud (Grafana Cloud). The former solution implies that installation, maintenance and scaling are up to the administrators, while the latter relieves them from these tasks, providing a fully-managed, fast and highly available platform.

The OE Gravimetric Database exploits the open source version of Grafana, installed on a high-performance virtual machine powered by Ubuntu Linux 22.04.

1.2.1 Data sources and queries

Grafana can manage a wide range of data sources. It comes with built-in support for many types of databases. All modern DB systems are supported, such as Apache Cassandra, AstraDB, MySQL, PostgreSQL (exploited for INGV Gravimetric DB) as well as a huge number of other cloud services providing data. Each data source has a proper query language, this is why Grafana provides different query editors on the basis of the selected data source.

The query editor enables the user to generate custom queries on the input data source (Figure 4).



```

1 SELECT $__time(g.time), (g."grav" - b."baro" - t."tide") as "Gravity 10 sec"
2 FROM
3   (SELECT (M.value*($ { SLN:value})) AS "grav", $__time(M.time)
4    FROM "Measure" AS M
5    WHERE $ _timeFilter(M.time) AND M.device_serial='{ $ { SLN:text }' AND M.type_code='1') AS g,
6   (SELECT (M.value*($ BAROPRESS)) AS "baro", $__time(M.time)
7    FROM "Measure" AS M
8    WHERE $ _timeFilter(M.time) AND M.device_serial='{ $ { SLN:text }' AND M.type_code='2') AS b,
9   (SELECT C.value AS "tide", $__time(C.time)
10    FROM "Correction" AS C
11    WHERE $ _timeFilter(C.time) AND C.device_serial='{ $ { SLN:text }' AND C.type_code='15') AS t
12 WHERE g.time = b.time AND b.time = t.time AND $ _RAW_GRAVITY > 0

```

Figure 4 Example of query for retrieving gravimetric data to be plotted on a Grafana time series panel.

1.2.2 Dashboard and Panels

Once a proper data source was set and a series of queries were designed to extract a chunk of data, the issue of an efficient visualization comes out. The block which plots the data selected through a proper query is called panel. Grafana offers a number of visualizations to support different needs. Time series, bar charts, histograms, classic representations are provided, but also more advanced plots, such as radial gauges, pie charts and heatmaps are implemented. Each panel is highly configurable in terms of colors, legends and axes properties. Several panels (graphs) can be combined with each other to compose a complex and synoptic representation, called dashboard (Figure 5). Grafana developers allow interested people to explore and “play” with different dashboard configurations through a public demo [12].



Figure 5 Grafana dashboard with three panels, each containing the plotted signal of an instrument.

2. Datasets

The dataset generated by an iGrav gravimeter is made up of TSF daily files. Essentially, TSF files, where TSF stands for TSoft Format, are ASCII encoded files structured using tagged blocks. Each block consists of two parts: (i) the tag, which is a short string delimited with square brackets always positioned at the very beginning of a new line and (ii) the information, which starts just after the tag, and continues until a new tag or the end of the file is reached. Some tags should always be present in a TSF file, while others can be omitted. More information about TSF format can be found at the page of the Royal Observatory of Belgium dedicated to TSoft package [13]. iGrav files start with format details, mainly TSF file version ([TSF-file] tag) and sampling rate in seconds ([INCREMENT] tag). Then, [CHANNELS] and [UNITS] tags provide necessary information (name of the channel and its measure unit) about the columns in the data section. The last and largest block in the file is the [DATA] one. After the tag a TAB separated data structure containing a large matrix of values begins. Each recorded row is made up of 20 channels (excluding the timestamp). The iGrav sampling rate is 1 Hz, i.e. one line per second, from 00:00:00 to 23:59:59, for a total of 86.400 daily data lines. Each data file takes up about 21.4 MB on the disk. iGrav network, in continuous 24/7 recording since 2014, has collected a huge amount of data. In the light of this, two considerations were made before the injection of this big dataset into the new RDBMS: (i) some information (columns) are not useful to be stored, (ii) the 1 Hz sampling rate is too high with respect to the frequency of significant gravimetric variations. A consistent dataset shrinking was obtained by reducing the number of columns as well as the number of rows. In particular, the final dataset to be written on the database was selected on a subset of original data columns and by applying an average over 10 consecutive samples. In conclusion, a set of 14 channels (excluding the time-stamp) recorded every 10 seconds were selected as database source.

3. Database

3.1 Design

The concept behind the design of the database structure was the definition of a generic and flexible structure, well suited for the integration of different types of instruments, not only iGrav gravimeters. In order to obtain this goal, *Measure* and *Correction* tables were structured according to the metrics-based model [14] (Figure 6), widely used for time series data. In this model, each row of a table represents a measurement of a certain metric, from a data source, at a certain time.

Metric	Timestamp	Cluster	Hostname	Metric Value
cpu	2015-04-28T17:50:00Z	Cluster-A	host-a	10
cpu	2015-04-28T17:50:10Z	Cluster-A	host-b	20
cpu	2015-04-28T17:50:20Z	Cluster-A	host-a	5
iops	2015-04-28T17:50:00Z	Cluster-A	host-a	10
iops	2015-04-28T17:50:10Z	Cluster-A	host-b	30
iops	2015-04-28T17:50:20Z	Cluster-A	host-a	8

Metrics Timestamp Subject dimension Measurements

Figure 6 Example of a table designed according to a metric-based model.

Another goal was to limit the relationships between entities (tables in the DB) as much as possible, in order to limit the number of tables involved in database queries. The latter becomes slower in relation to the amount of data and the number of joined tables.

3.1.1 Conceptual model

The first step in a database building is the conceptual design. This phase involves the definition of entities and relationships between them, i.e. the main elements to be modeled and the interaction with each other. In order to understand the problem domain, it could be useful to write a description of the problem. At this stage the abstraction level is high, without getting into technical details. The result is the conceptual schema, which relates the entities to each other through the identified relationships. The latter can have three cardinalities: one-to-one, one-to-many, and many-to-many. For example, a Device belongs to a (only one) Model, while for a Model there could be many (n) Devices. Differently, a Device can be installed in several Places (in different time periods, obviously) and a Place can be exploited for several Devices (simultaneously also). The resulting diagram is known as Entity/Relationship diagram (E/R diagram, in short form). The E/R diagram of Gravimetric DB is shown in Figure 7. The cardinality of each relationship is indicated according to the Look-Here method [15].

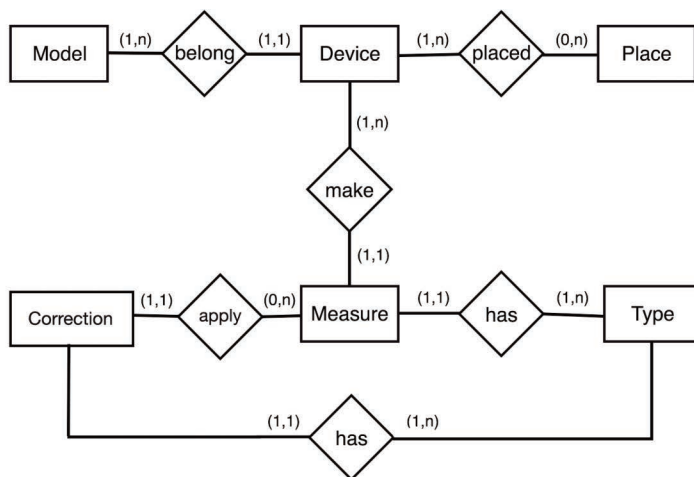


Figure 7 Final E/R diagram of Gravimetric database. Entities are represented using rectangles, while relationships are modeled by diamonds. Cardinalities are indicated next to relationship lines, such as (1,1) for mandatory one-to-one, (1,n) for mandatory one-to-many, and (0,n) for optional one-to-many relationships.

The E/R diagram in Figure 7 shows six entities:

- **Model** for the device model name.
- **Device** contains the device information, e.g. the device serial identifier.
- **Place** models the installation sites.
- **Measure** represents a single measurement recorded by a device.
- **Correction** models signals and measurements used as correction for gravimetric data (e.g. the synthetic Earth tides).
- **Type** for measurement types and their unit of measure.

3.1.2 Logical model

Once completed the first (conceptual) phase, it is time to build up the logical scheme. It is a data model that describes how data can be stored in the form of records inside proper data structures.

Starting from entities and relationships of the conceptual data model, which describes the semantics of the problem, it is now necessary to determine the data structures in terms of tables and their relative columns [16].

Essentially, each E/R diagram entity results in a table in the logical scheme. Table columns are derived from entity attributes; among them, one is identified as the primary key, that is a column of values which are different from each other, so uniquely representing each individual record in the table.

Also, E/R diagram relationships have a direct translation in the logical scheme. The one-to-one relationship is modeled placing the primary key of a table as an additional column in the other table; this column provides a “link” to a record in another table and is referred to as foreign key. The one-to-many relationship is modeled similarly, taking care to add the foreign key to the “many” side table. The many-to-many relationship translation is more complex. A new table must be defined, containing primary keys from both the “linked” tables, as foreign keys [17]. The associative table can also define new columns for attributes related to the association. For example, DevicePlaced is an associative table that contains the information about the geographical place where a device is or was installed; the installation date and, eventually, even the dismissal date are recorded as attributes of this association.

In the following tables, the logical schema of OE Gravimetric Database is shown. Primary keys are highlighted using the underlined format, whereas “foreign” keys are written in italic format.

Attribute name	Description
<u>id</u>	Incremental instrument model id
name	Model name

Table 3 Database Model table.

Attribute name	Description
<u>serial_code</u>	Device serial code
info	More information text field
is_absolute	Boolean that represents whether the gravimeter provides absolute or relative gravity values
<i>model_id</i>	Foreign key of the Model table

Table 4 Database Device table.

Attribute name	Description
<u>place_code</u>	Unique given code to place
name	Complete place name
latitude	GPS Latitude
longitude	GPS Longitude
altitude	GPS Altitude (above sea level)

Table 5 Database Place table.

Attribute name	Description
<u>device_serial</u>	Foreign key of the Device table
<u>place_code</u>	Foreign key of the Place table
<u>installation_date</u>	The date on which the instrument was installed at the designated place.
<u>dismission_date</u>	The date on which the instrument was removed from its assigned installation place.

Table 6 Database DevicePlaced table.

Attribute name	Description
value	The recorded data value
<u>time</u>	The recorded data datetime
<u>type_code</u>	Foreign key of the Type table
<u>device_serial</u>	Foreign key of the Device table

Table 7 Database Measure table.

Attribute name	Description
value	The correction value
<u>time</u>	The datetime to which the correction should be applied
<u>type_code</u>	Foreign key of the Type table (Type of correction)
<u>device_serial</u>	Foreign key of the Device table
<u>measure_type</u>	Foreign key of the Type table (Type of the measure to correct)

Table 8 Database Correction table.

Attribute name	Description
<u>id</u>	Incremental type id
label	Text label of the type
unit	The measurement unit of this type

Table 9 Database Type table.

3.1.3 Physical model

The physical one is the lowest model in terms of abstraction, so all data structure details have to be shown at this stage [18]. The table structures derive from the logical model. The columns, previously defined, with their constraints, such as primary keys, are maintained. At this level, data types are specified for each table column, depending on the chosen RDBMS, and relationships between tables are well defined and shown (Figure 8).

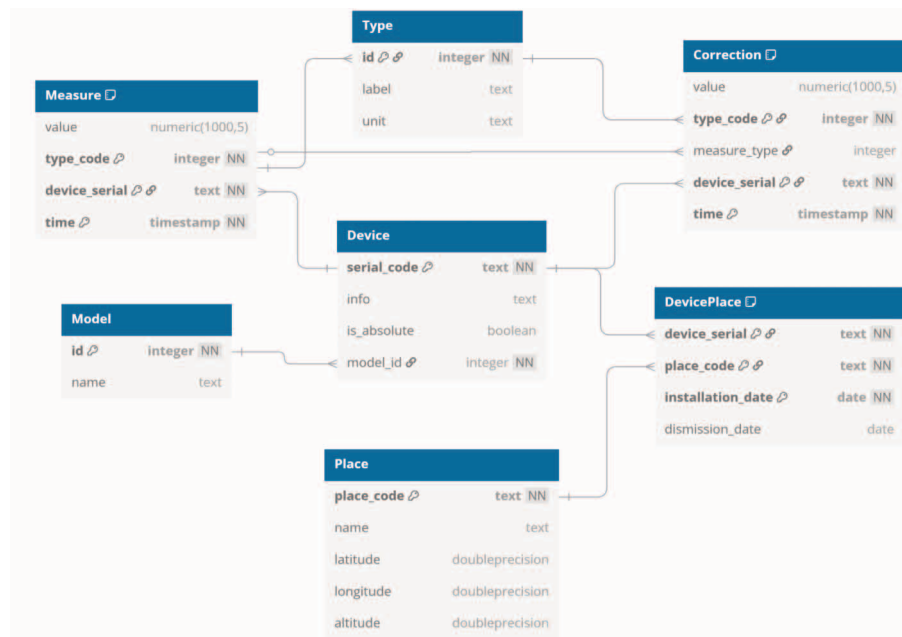


Figure 8 The physical data model of Gravimetric DB, obtained through dbdiagram.io online cloud service. Data types are specified for each table column. The label “NN” indicates the “Not Nullable” constraint (that field cannot be left blank).

Different approaches related to the choice of the data type for the primary keys were exploited. Except for the small size tables, where the primary key is an incremental ID (*Type* and *Model* tables), the UUIDv4 had initially been chosen. UUID is the acronym of “Universal Unique Identifier” and is an identifier of 128 bit (16 byte) widely used in the development of software solutions [19]. The version 4 of the UUID is a variant of the standard UUID and the identifier generation is a random type where the probability to find a collision is one in a billion on 103 trillions generated UUIDv4. This option was discarded because on larger tables, such as the *Measure* one, considering the period between August 2014 and February 2022, there was around 676 million of rows; exploiting a 128 bit UUIDv4 for each row, it would have taken up a considerable disk space, approximately 10.8 Gigabyte.

For that reason, the UUIDv4 option was discarded in favor of an alternative approach involving the creation of a composite key, formed by the combination of one or more columns. Looking at the *Measure* table, the primary key is the combination of *time*, *type_code*, and *device_serial*, considering that a specific device (with a unique *device_serial*) makes a measure (with a specific *type_code*) in a certain instant (*time*), so the combined primary key is not prone to collisions.

As described in the “Logical Model” section, relationships between tables are modeled by means of foreign keys. Nevertheless, TimescaleDB and its hypertables (see “Hypertables and chunks” section) have a limitation, namely, foreign keys are not supported [20]. In order to get around this problem, foreign keys were only kept in the conceptual data model, but not implemented in the physical model. Indeed, considering that data entry management is done in an automated and consistent manner, it is still possible to query the database joining tables, by means of “link” fields as if they were actually foreign keys (even if they are not in fact).

3.2. Implementation

The database, which is powered by PostgreSQL, TimescaleDB and Grafana, was implemented

in a virtual machine hosted at the INGV-OE data center. The underlying hypervisor is based on a Microsoft solution, specifically Hyper-V. The virtual machine runs GNU/Linux Ubuntu 20.04.6 LTS [21] and is equipped with an Intel Xeon Gold 5215 processor, featuring 10 cores and 20 threads, with a 2.50 GHz clock speed. It also includes 32 GB of RAM memory and a 128 GB main disk storage for the operating system and applications. An additional 1TB storage unit is mounted to accommodate the synced data files from remote stations.

Several essential software components were installed to ensure the proper operation of the database. Among the main installed packages, TimescaleDB version 2.5.2, Grafana version 9.1.0, Python versions 2.7.18 and 3.8.10, and PostgreSQL version 14.1 (as a dependency of TimescaleDB to support the database infrastructure).

3.2.1. Installing self-hosted TimescaleDB and PostgreSQL

PostgreSQL and TimescaleDB installation, on a Debian-based host, requires a series of steps, in particular, terminal commands, to configure the necessary repositories and install the required packages.

Necessary dependencies, including tools for handling third-party repositories and fetching packages, must be installed.

```
sudo apt install gnupg postgresql-common apt-transport-https lsb-release wget
```

Then, the PostgreSQL repository setup script must be run to configure the system for PostgreSQL installation.

```
sudo /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh
```

The TimescaleDB repository needs to be added to the system's package sources.

```
echo "deb https://packagecloud.io/timescale/timescaledb/ubuntu/ $(lsb_release -c -s)
main" | sudo tee /etc/apt/sources.list.d/timescaledb.list
```

Once the repository has been added, the GPG key for TimescaleDB must be installed to authenticate the packages.

```
wget --quiet -O - https://packagecloud.io/timescale/timescaledb/gpgkey | sudo apt-key
add -
```

The repository list should be updated to include the newly added TimescaleDB repository, ensuring that the system can retrieve the latest package information.

```
sudo apt update
```

Finally, the installation of TimescaleDB and PostgreSQL can be carried out.

```
sudo apt install timescaledb-2-postgresql-14
```

This will install the latest TimescaleDB version compatible with PostgreSQL version 14.x. For further details on the installation process and additional configuration options, the official TimescaleDB documentation [22] can be consulted.

3.2.2. PostgreSQL - Creating database and tables

Once the installation of TimescaleDB and PostgreSQL has been successfully completed, the next step is to create the database and its tables. This process involves executing a series of commands within a PostgreSQL session.

A new PostgreSQL session must be opened.

```
sudo psql -U postgres -h localhost
```

After executing this command, the user will be prompted to enter the password for the selected PostgreSQL user. The default user is “postgres” as well as the default password is typically set to “postgres” unless it has been changed during the installation process (see Section 3.2.3). Within the active session, the next step is the database creation.

```
CREATE DATABASE db_name;
```

Then, a connection to the new database can be established.

```
\c db_name
```

Once connected, the user can proceed to create the necessary tables. Here it's shown the command for the creation of a table called “Model” with two fields (id and name).

```
CREATE TABLE “Model” (  
  id integer GENERATED ALWAYS AS IDENTITY PRIMARY KEY,  
  name text  
);
```

This will define the table structure, with an automatically incrementing ID field and a text field for the name. After all the required tables have been created, the PostgreSQL session can be closed.

```
\q
```

The database setup process is then complete. The user can proceed with populating the tables and running queries as needed. It should be pointed out that the ones just created are “normal” PostgreSQL tables, not using the advantages provided by TimescaleDB plugin. In Section 3.3.3 it is well explained how to convert their schema into TimescaleDB hypertables.

3.2.3. PostgreSQL - Managing users, passwords and access restrictions

Creating new users is essential for assigning tailored access rights to a database. The process can be carried out using the commands below. A new PostgreSQL session must be initiated.

```
sudo psql -U postgres -h localhost
```

Once the session is active, a new user can be created.

```
CREATE USER user_name WITH PASSWORD ‘user_pass’;
```

Additional options can be added to this query to impose various restrictions on the user. A complete list of available options can be found in the PostgreSQL documentation [23]. The password for an existing user, such as the default user “postgres”, can be modified by exploiting a similar command.

```
ALTER USER postgres WITH PASSWORD 'new_password';
```

Very often a read-only access rule to the database is required to be set.

```
CREATE ROLE readaccess;  
GRANT CONNECT ON DATABASE name_of_your_db TO readaccess;  
GRANT USAGE ON SCHEMA public TO readaccess;
```

This rule can be applied to the entire database (all the tables).

```
GRANT SELECT ON ALL TABLES IN SCHEMA public TO readaccess;
```

Alternatively, the restriction could only be applied to specific tables (instead of all tables).

```
GRANT SELECT ON your_table_name IN SCHEMA public TO readaccess;
```

Once the “readaccess” role is created, it can be assigned to a user.

```
GRANT readaccess TO read_user;
```

Further details on specific limitations can be found in the PostgreSQL documentation [24].

3.2.4. Grafana Installation and configuration

Grafana installation on an Ubuntu based machine is quite straightforward. A short list of commands allows to install the required packages and retrieve the Grafana repository signing key.

```
sudo apt-get install -y apt-transport-https  
sudo apt-get install -y software-properties-common wget  
sudo wget -q -O /usr/share/keyrings/grafana.key https://apt.grafana.com/gpg.key
```

Add the repository for stable versions of Grafana.

```
echo "deb [signed-by=/usr/share/keyrings/grafana.key] https://apt.grafana.com stable  
main" | sudo tee -a /etc/apt/sources.list.d/grafana.list
```

Then, the list of all available packages must be updated.

```
sudo apt update
```

Once all the previous commands have been executed, the self-hosted Enterprise version of Grafana can be installed.

```
sudo apt install grafana-enterprise
```

Once installed, Grafana must be configured to connect to a data source (one or more) and a series of custom queries have to be developed to allow data extraction and graphs creation. Two dashboards have been designed for INGV gravimetric data. The main one, called “Gravity Stations” shows the g-value time series for each station of the network. Another dashboard, called “iGrav Details”, provides insight on each station, plotting the time series of instrument’s most relevant logged parameters (Figure 9). Users can enable/disable parameters by means of a filter list. This allows an efficient technical monitoring of the instrument SOH (State-of-Health).



Figure 9 The “iGrav Details” Grafana dashboard. It shows time series for several parameters related to an iGrav gravimetric station. Users can select/deselect parameter plots to be shown.

The Grafana dashboard is highly configurable, so the administrator can insert push buttons and filter lists to show or hide some graphs. The dashboard structure is saved on a configuration file on the server, so that the “normal” user can only navigate the dashboard, zooming on interested time periods, activating or hiding some graphs.

3.3. Optimization

Considering the great amount of data to be stored in the database (see Section 2), a necessary step was to change the default directory for database files from the default PostgreSQL directory (on the operating system partition, low capacity) to a 1TB storage unit. A second section describes another key step that can significantly improve system performance: the “tuning” of PostgreSQL configuration parameters, according to the characteristics of the system on which it is installed. After completing these initial configuration steps, the next phase involved working with database tables to turn them into TimescaleDB hypertables. The last section (3.3.4) is dedicated to the set-up of Grafana configuration to make it suitable for long time series data.

3.3.1 PostgreSQL - Change the default data directory

Changing the default data directory of a PostgreSQL instance [25] requires opening a new session.

```
sudo psql -U postgres -h localhost
```

The current data path can be retrieved.

```
SHOW data_directory;
```

Let's assume the default path is `"/var/lib/postgresql/14/main"`.

Before proceeding with the file movement, the PostgreSQL service process needs to be stopped, to prevent any access conflicts while working on the files.

```
sudo systemctl stop postgresql
```

Once the process is stopped, the current database files can be copied to the new location.

```
sudo rsync -av /var/lib/postgresql /new_destination_path
```

The original data directory can be renamed.

```
sudo mv /var/lib/postgresql/14/main /var/lib/postgresql/14/main.bak
```

Then, the PostgreSQL configuration file must be updated to reflect the new data directory.

```
sudo nano /etc/postgresql/14/main/postgresql.conf
```

The line `"data_directory = '/new_path/postgresql/14/main'"` should be modified accordingly. Finally, the PostgreSQL process can be restarted.

```
sudo systemctl start postgresql
```

Once verified the proper behaviour of PostgreSQL server, the old data directory can be removed.

```
sudo rm -Rf /var/lib/postgresql/14/main.bak
```

3.3.2 PostgreSQL - Tuning configuration parameters

The tuning of PostgreSQL server can be done either manually, by editing the `"postgresql.conf"` file, or by using the `"timescaledb-tune"` tool, which is installed alongside TimescaleDB. This tool suggests configuration adjustments through a wizard, based on the characteristics of the host machine. It can be easily launched via terminal.

```
sudo timescaledb-tune
```

Sometimes, it could be necessary to specify the path to the configuration file.

```
sudo timescaledb-tune --conf-path=/path_of_postgresql.conf
```

3.3.3 PostgreSQL - Set up TimescaleDB hypertables

TimescaleDB provides several SQL functions to interact with and manage tables in the PostgreSQL database. A full list of TimescaleDB API can be found at URL [26].

Each standard PostgreSQL relational table which is intended to contain a time series data should be converted into a hypertable, which is a PostgreSQL table that automatically partitions data by time.

```
SELECT create_hypertable('Correction', 'time');
```

If the table that needs to be transformed into a hypertable has been already populated, an additional parameter must be added to the *create_hypertable* function.

```
SELECT create_hypertable('Measure', 'time', migrate_data => true);
```

Once the two hypertables were created, the chunk interval had to be configured. A chunk is a partition of the hypertable, on the basis of the time column.

```
SELECT set_chunk_time_interval('Measure', INTERVAL '120 days');
```

The same query was also executed for the *Correction* table. The default partition interval is 7 days, but since the queries performed on the database generally cover periods of at least one month, it was decided to adjust the interval to 4 months. This configuration was a good compromise between the need to avoid too many chunks for each year, which could negatively impact performance and ensuring that the chunks size were not too large to nullify the performance benefits provided by the data segmentation.

The last database configuration was related to compression policies for the two hypertables. Specifically, the *Measure* table was set to compress data older than 2 months, while the *Correction* table was configured to compress data older than 1 year. To set a compression policy on a table, the *add_compression_policy* function should be used, specifying the table name and the desired time range.

```
SELECT add_compression_policy('Measure', INTERVAL '2 months');
```

Data chunks are compressed on the basis of a time period related to the current time.

3.3.4 Grafana - Tuning configuration parameters

Also the Grafana configuration needs a custom set up. By default, Grafana limits the number of rows processed from an SQL source. Each query is limited to 1 million rows, in other words, 1 million points shown on the graph. Considering INGV-OE gravimetric data, where a row represents 10 seconds of continuous signal, this limitation reflects in a maximum plot length of almost 4 months. To address this issue, the value was increased to 3.2 million points, allowing the user to plot a 1-year period on a single chart.

To modify the limit of points in a graph, the “grafana.ini” file needs to be edited.

```
sudo nano /etc/grafana/grafana.ini
```

Once the file is open, the variable “row_limit” can be modified (the default value is 1 million).

4. Developed scripts for data injection

The TSoft file format, adopted by iGrav gravimeters, is not natively supported by major ETL (Extract, Transform, Load) tools [27]. For this reason, it was necessary to develop custom scripts to facilitate the injection of historical data, real-time data, and synthetic corrections for earth tides, gaps, and jumps.

Python language was exploited for this job, due to its widespread adoption and efficiency in handling such tasks.

4.1 Historical data

To manage the massive import of historical data, a Python script was designed to preprocess TSoft files synchronized (copied) in a temporary directory. The script utilizes the glob library [28] to recursively retrieve file paths. During preprocessing, 14 meaningful parameters are extracted from the original 20 columns, while time holes, redundant time repetitions (due to GPS synchronization errors), and incomplete rows are removed. The preprocessed data is saved as a “stage” CSV file.

This stage file undergoes a further transformation, where the data is averaged over 10-seconds intervals using a recursive function. This downsampling reduces the original 1-second resolution to one row every 10 seconds, with the computed rows written into a “final” CSV file. The optimized PostgreSQL COPY command is then used to efficiently inject this data into the database.

4.2 Correction scripts

In order to remove the main “noise” sources for gravity signals, the raw gravimetric signal needs to be corrected for the effects of Earth tides and local atmospheric pressure changes. Moreover, further corrections should be applied to the signal for other occasional events such as jumps and gaps.

Therefore, in addition to the historical data injection code, three other Python scripts were developed to handle corrections for Earth tides, barometric pressure, jumps, and gaps.

The common initial step requires the user to define, for each file to be processed, which parameter the correction should be applied to and which device it targets. These selections can be done through a sequence of command-line menus.

4.2.1 Synthetic Earth Tide

This script imports pre-calculated 10-second averaged Earth tidal series generated by the specialized software ETERNA 3.3 [29], a widely recognized tool for tidal analysis and prediction. Designed for geophysical applications and extensively utilized in geodesy, seismology, and oceanography, ETERNA 3.3 enables precise computation of tidal parameters and corrections. These pre-averaged tidal values are subsequently provided with the necessary metadata to specify the type of correction (tidal) and the target device (the synthetic model is calculated taking into account the geographical position of the gravimeter).

The resulting time series is subsequently injected into the database.

4.2.2 Gaps

Gaps represent missing data due to instrument malfunctions or significantly altered signals unsuitable for analysis. This script reads a “stage” CSV file containing a series of gaps with start datetime entries and their respective durations. The time span is measured in 10-second intervals, so a duration of 6 will result in a 60-seconds gap. The script processes this data by

generating individual rows for each timestamp within the specified duration, starting from the provided datetime. For each row, metadata is added to define which signal and instrument the correction applies to. These entries, along with the metadata, are then prepared for insertion into the database to accurately reflect the interruptions.

4.2.3 Jumps

Jumps are abrupt changes in the gravimetric signal caused by external events, resulting in a positive/negative offset on the signal. This kind of variation is not related to a “real” gravity change but becomes a new reference level for the recorded signal, until the next jump occurs. The script processes the “stage” CSV file that contains the datetimes and the observed shift values, while it retrieves the data acquisition start date (the first measurement date) and, if it exists, the date of the most recent applied jump correction from the database.

The following function finds the nearest lower date in the “stage” file’s list of dates and compares it to the last jump date. This last jump date is either the most recent correction’s datetime or, if no correction has been made, the datetime when data acquisition began. The function returns either the index of the nearest date to the last jump date, or -1 if no such date is found.

```
def find_nearest_correction_idx(corrections, date):
    # corrections is an array of array
    # filter correction for get only the nearest bottom margin
    dates = [i[1] for i in corrections if i[1] <= date]
    if len(dates) > 0:
        # find the closest date
        closest_date = min(dates, key=lambda x: abs(x - date))
        index = 0
        # get the idx of the closest date finded above
        for idx, correction in enumerate(corrections):
            if closest_date in correction:
                index = idx
        return index
    else:
        return -1
```

If the function returns -1 and thus does not find a date within the lower margin, a row is created in the “final file” with a shift value of 0 and the current date. If a nearest date is found, a row is created with the corresponding shift value of that nearest date and the current date. The current date is then incremented, and the cycle continues until the end date is reached (i.e., the date when the script is executed).

As the script runs, for each new date corresponding to a jump, the shift value from that point onward is recalculated by adding the previous shift value to the new one. This process continues until the end of the list in the “stage” CSV file is reached.

4.3 Real-time data

After the initial phase of populating the database with historical data and its corresponding corrections, the new challenge was automating the loading of recorded data in near real-time.

Drawing on the expertise gained from the historical data script, a new script was developed to handle this task. This script is executed cyclically as a Linux crontab task [30], minimizing system resource consumption by avoiding persistent background execution.

The real-time loading script incorporates parts of the logic from the earlier script used for importing historical data but in a more complex manner. Indeed, the real-time file increments its size after each remote synchronization, launched by another crontab task. The script employs a configuration file to manage the processing queue (Figure 10); it contains all the relevant information about the last running session, namely the last processed date, the device serial number, the index of the last processed line in the file, the file path, and the file status. The latter can be:

- **Active:** Fully unprocessed file.
- **Daily:** Current day file, requiring updates (the file is not complete).
- **Stopped:** File unchanged since the last synchronization.
- **Remove:** Completed file ready for archival.

```
"files": [  
  {  
    "date": "2022-09-14 06:59:59",  
    "device": "020",  
    "idx": 25254,  
    "path": "/Real-Time-Import-Fix/data/iGrav020/iGrav020_2022/0914/Data_iGrav020_0914.tsf",  
    "status": "stopped"  
  }  
],
```

Figure 10 Queue array inside the configuration file; only one element inside.

This structure provides a clear understanding of the processing status for each device during each execution, ensuring efficient file handling. The configuration file is updated at the end of each script execution and loaded into a configuration object at the start of the script.

When the script begins, it retrieves all synchronized files for each device, adds them to the configuration, and starts processing. If files are already present in the configuration, the script verifies whether there is a next-day file and whether the current file has been fully processed. If the current file has been fully processed and a next-day file is available, the script adds the new file to the configuration, removes the processed file, and moves it to a directory designated for processed files.

If the current file has not been fully processed but a next-day file exists, the script adds the next file to the list and processes both files, resuming from where it left off in the incomplete file.

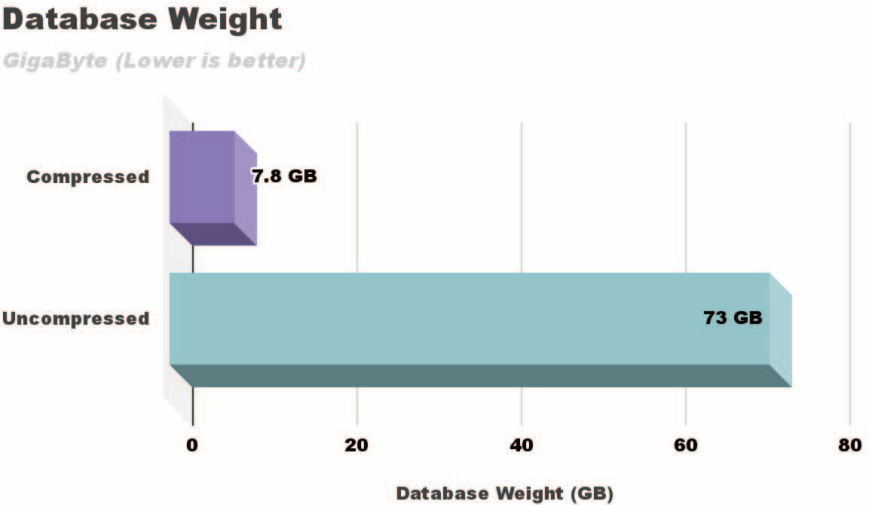
For daily files, the script checks that all 10-seconds intervals are complete, to avoid averaging incomplete periods. Surplus seconds are excluded until synchronization with the remote station is complete, ensuring accurate interval averaging.

5. System test and performance

Once the database was implemented and populated, several tests were conducted to evaluate the compression performance of TimescaleDB as well as the efficiency of queries carried out on uncompressed and compressed data. The goal was to verify whether the adoption of TimescaleDB improved the performance of the RDBMS compared to plain PostgreSQL or not. Compression results are illustrated in the graph in Figure 11. It shows how the compression

shrinks significantly the database size, dropping from 73 GB to just 7.8 GB, achieving approximately 89% data compression.

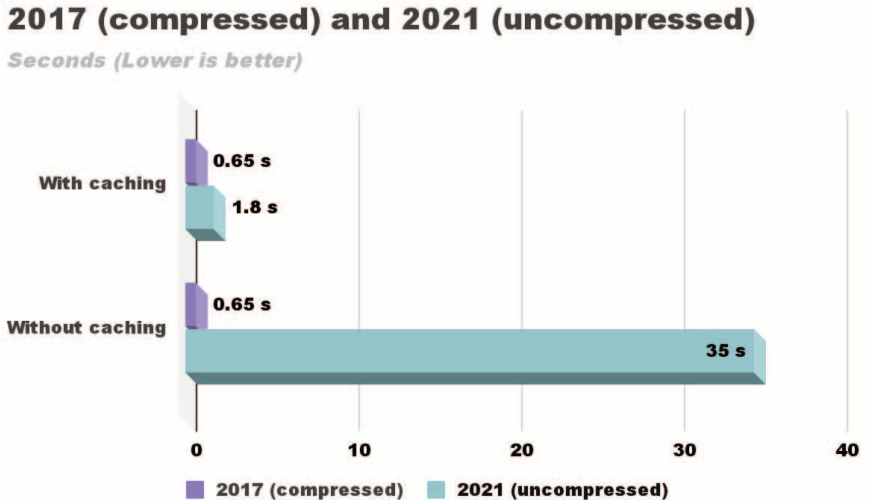
Figure 11 The gravimetric database size dramatically reduces when the heaviest (most populated) tables are compressed.



Other tests, on query performance, were also performed by examining two years of data, comparing the execution time on a compressed year to that of an uncompressed year. Tests were conducted both before and after the query was cached. Query caching occurs when the same query is executed multiple times consecutively, resulting in the first query run typically being slower than subsequent runs.

The graph in Figure 12 demonstrates that queries on uncompressed data are significantly slower compared to the same queries applied to compressed data. Additionally, there is no significant difference between cached and uncached queries if the data being retrieved is already compressed. These results confirm the efficiency of the column-native compression well described in 1.1.2 section.

Figure 12 SQL queries performance (timing) on two years of data. Compressed data allows much faster runs than uncompressed data. Caching technique heavily affects query performance on uncompressed data, while its contribution is negligible for queries on compressed data.



The graph in Figure 13 shows the comparison between the performance of simple vs complex queries across various scenarios, providing additional insights into query efficiency.

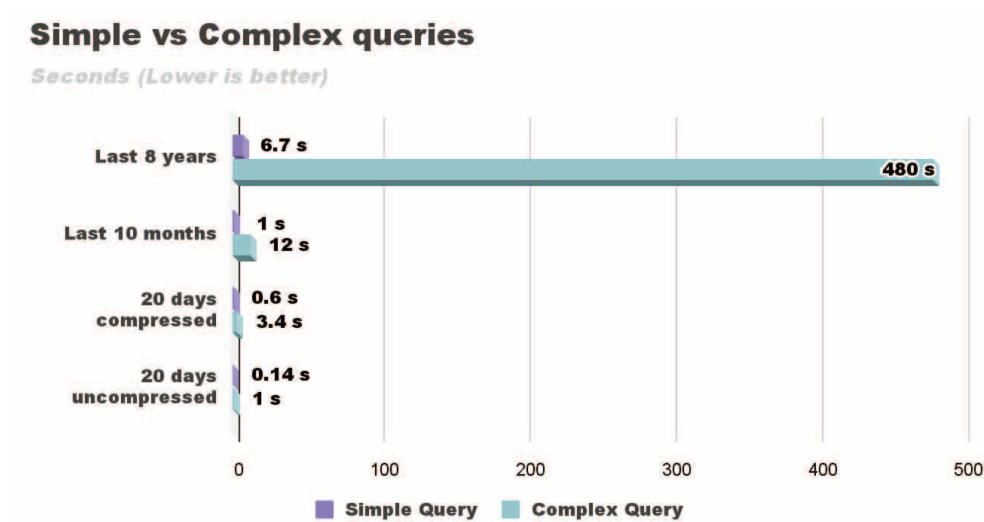


Figure 13 Comparison between performance of simple and complex queries.

A complex query such as the one in the following, which is used in Grafana to plot and monitor gravity data, is not optimized for time series databases. As widely described in this paper, queries for time series databases should always avoid joins whenever possible or at least, to minimize their usage.

```
SELECT g.time AS "time", (g."grav" - b."baro" - t."tide") as "Gravity"
FROM
(
  SELECT (M.value*(-101.55)) AS "grav", M.time AS "time"
  FROM "Measure" AS M
  WHERE BETWEEN '2015-01-01 00:00:00' AND '2023-11-20 23:59:50' AND
M.device_serial='016' AND M.type_code='1'
) AS g,
(
  SELECT (M.value*(-0.33)) AS "baro", M.time AS "time"
  FROM "Measure" AS M
  WHERE BETWEEN '2015-01-01 00:00:00' AND '2023-11-20 23:59:50' AND
M.device_serial='016' AND M.type_code='2'
) AS b,
(
  SELECT C.value AS "tide", c.time AS "time"
  FROM "Correction" AS C
  WHERE BETWEEN '2015-01-01 00:00:00' AND '2023-11-20 23:59:50' AND
C.device_serial='016' AND C.type_code='15'
) AS t
WHERE g.time = b.time AND b.time = t.time
```

On the other hand, a simple query that just selects two columns from a single table, such as the following one, results to be very efficient.

```
SELECT M.value, M.time
FROM "Measure" AS M
WHERE M.time BETWEEN '2015-01-01 00:00:00' AND '2023-11-20 23:59:50' AND
M.device_serial='016' AND M.type_code='1'
```

Joining large tables can significantly reduce query performance. This inefficiency is clear in the resulting graph, in Figure 13, where a complex 8-years query is approximately 70 times slower than a simple query on the same period.

Conclusions and remarks

This paper has presented the design, implementation, and optimization of a time series database for continuous gravimetric data at INGV-OE, leveraging PostgreSQL, TimescaleDB, and Grafana. The integration of these technologies has enabled efficient storage, retrieval, and visualization of large volumes of gravimetric data, addressing the inherent challenges of time series data management.

The adoption of TimescaleDB has proven to be a crucial factor in overcoming the limitations of traditional relational databases. By employing hypertables, chunking, and native column compression, TimescaleDB has significantly improved the performance and scalability of the database. The results of some performance tests demonstrate substantial improvements in data compression and query efficiency, making TimescaleDB an ideal solution for handling continuous data.

The implementation of Grafana, with its capability to create customized dashboards and panels, has provided a powerful and user-friendly platform for interactive data visualization and monitoring.

Overall, the development of this database system represents a significant step forward in the management of continuous gravimetric data. It not only addresses the specific needs of the Gravimetry Team of the INGV-OE, but can be exported, as a valuable model for other organizations dealing with similar time series data challenges. Future work will focus on further optimizing the system, exploring additional features of TimescaleDB and Grafana. Moreover, exploiting the multipurpose data structure of the database, other types of geophysical data could be stored.

In conclusion, the successful implementation of this system highlights the potential of combining cutting-edge database technologies with robust data visualization tools to achieve efficient and scalable solutions for complex data management tasks.

References

- Carbone D., Cannavò F., Greco F., Reineman R., Warburton R.J., (2019). *The benefits of using a network of superconducting gravimeters to monitor and study active volcanoes*. Journal of Geophysical Research: Solid Earth, 124, 4035–4050. <https://doi.org/10.1029/2018JB017204>
- Kammakomati M., (2021). *Performance Evaluation of TimescaleDB for Storage of Historical Data from WinCC OA SCADA Systems*. Zenodo. <https://doi.org/10.5281/zenodo.5541641>
- Mostafa J., Wehbi S., Chilingaryan S., Kopmann A., (2022). *SciTS: A Benchmark for Time-Series Databases in Scientific Experiments and Industrial Internet of Things*. SSDBM 2022: 34th International Conference on Scientific and Statistical Database Management Copenhagen Denmark. Association for Computing Machinery (ACM). <https://doi.org/10.1145/3538712.3538723>
- Mukherjee Sourav, (2019). *The Battle between NoSQL Databases and RDBMS*. Available at SSRN: <https://ssrn.com/abstract=3393986> or <http://dx.doi.org/10.2139/ssrn.3393986>
- Reddy Yeruva A., Basavegowda Ramu V., (2023). *End-to-End Observability with Grafana: A comprehensive guide to observability and performance visualization with Grafana*. BPB Publications. ISBN: 9789355515483, eISBN: 9789355515445. Pages: 332.
- Sierra-Fernández J.M., Florencias-Oliveros O., Espinosa-Gavira M.J., González-de-la-Rosa J.J.,

Agüera-Pérez A., Palomares-Salas J.C., (2021). *Online System for Power Quality Operational Data Management in Frequency Monitoring Using Python and Grafana*. *Energies*, 14(24), 8304. <https://doi.org/10.3390/en14248304>

Sitography

- [1] iGrav Gravity Sensors.
<https://www.gwrinstruments.com/igrav-gravity-sensors.html>
- [2] TimescaleDB.
<https://www.timescale.com>
- [3] PostgreSQL.
<https://www.postgresql.org>
- [4] Grafana.
<https://grafana.com>
- [5] Ranking of the most popular RDBMS worldwide.
<https://www.statista.com/statistics/1131568/worldwide-popularity-ranking-relational-database-management-systems>
- [6] Top 10 NoSQL Databases in 2022.
<https://www.decipherzone.com/blog-detail/nosql-databases>
- [7] Why databases do not normally scale up well.
<https://www.timescale.com/blog/time-series-data-why-and-how-to-use-a-relational-database-instead-of-nosql-d0cd6975e87c>
- [8] About TimescaleDB hypertables.
<https://docs.timescale.com/use-timescale/latest/hypertables/about-hypertables/#main-content>
- [9] About TimescaleDB compression.
<https://docs.timescale.com/use-timescale/latest/compression/about-compression>
- [10] A Data Visualization Tool - Grafana.
https://www.researchgate.net/publication/367451523_A_Data_Visualization_Tool-_Grafana
- [11] Python official website.
<https://www.python.org>
- [12] Public Grafana demo.
<https://play.grafana.org>
- [13] TSoft software package.
<https://seismologie.oma.be/en/downloads/tsoft>
- [14] Key concepts and Features of Time series database.
https://www.alibabacloud.com/blog/key-concepts-and-features-of-time-series-databases_594734
- [15] E-R Diagram Cardinality and Participation.
<https://www.datasciencecentral.com/e-r-diagram-cardinality>
- [16] ER-To-Relational Mapping.
<https://medium.com/@kumarjai2466/er-to-relational-mapping-ac84b3c9f258>
- [17] Many-to-many relationships.
<https://www.datensen.com/blog/er-diagram/many-to-many-relationships>
- [18] What Is a Physical Data Model?
<https://www.gooddata.com/blog/physical-vs-logical-data-model>
- [19] Universally unique identifier.
https://en.wikipedia.org/wiki/Universally_unique_identifier
- [20] TimescaleDB hypertables limitations.

<https://docs.timescale.com/use-timescale/latest/limitations>
[21] Ubuntu OS.
<https://ubuntu.com/>
[22] Install and configure TimescaleDB on PostgreSQL.
<https://docs.timescale.com/self-hosted/latest/install/installation-linux/>
[23] PostgreSQL - Create a new user.
<https://www.postgresql.org/docs/8.0/sql-createuser.htm>
[24] PostgreSQL - Define access privileges.
<https://www.postgresql.org/docs/current/sql-grant.html>
[25] How to move the PostgreSQL data directory to a new location.
<https://www.digitalocean.com/community/tutorials/how-to-move-a-postgresql-data-directory-to-a-new-location-on-ubuntu-18-04>
[26] TimescaleDB API reference.
<https://docs.timescale.com/api/latest/>
[27] What Is ETL & Types of ETL Tools.
<https://www.dremio.com/resources/guides/intro-etl-tools>
[28] Python Glob library documentation.
<https://docs.python.org/3/library/glob.html>
[29] Eterna Earth Tide analysis and prediction program.
<https://eterna.bkg.bund.de>
[30] Command line “Cron” command.
<https://en.wikipedia.org/wiki/Cron>

QUADERNI di GEOFISICA

ISSN 1590-2595

<https://istituto.ingv.it/le-collane-editoriali-ingv/quaderni-di-geofisica.html/>

I QUADERNI DI GEOFISICA (QUAD. GEOFIS.) accolgono lavori, sia in italiano che in inglese, che diano particolare risalto alla pubblicazione di dati, misure, osservazioni e loro elaborazioni anche preliminari che necessitano di rapida diffusione nella comunità scientifica nazionale ed internazionale. Per questo scopo la pubblicazione on-line è particolarmente utile e fornisce accesso immediato a tutti i possibili utenti. Un Editorial Board multidisciplinare ed un accurato processo di peer-review garantiscono i requisiti di qualità per la pubblicazione dei contributi. I QUADERNI DI GEOFISICA sono presenti in "Emerging Sources Citation Index" di Clarivate Analytics, e in "Open Access Journals" di Scopus.

QUADERNI DI GEOFISICA (QUAD. GEOFIS.) welcome contributions, in Italian and/or in English, with special emphasis on preliminary elaborations of data, measures, and observations that need rapid and widespread diffusion in the scientific community. The on-line publication is particularly useful for this purpose, and a multidisciplinary Editorial Board with an accurate peer-review process provides the quality standard for the publication of the manuscripts. QUADERNI DI GEOFISICA are present in "Emerging Sources Citation Index" of Clarivate Analytics, and in "Open Access Journals" of Scopus.

RAPPORTI TECNICI INGV

ISSN 2039-7941

<https://istituto.ingv.it/le-collane-editoriali-ingv/rapporti-tecnici-ingv.html/>

I RAPPORTI TECNICI INGV (RAPP. TEC. INGV) pubblicano contributi, sia in italiano che in inglese, di tipo tecnologico come manuali, software, applicazioni ed innovazioni di strumentazioni, tecniche di raccolta dati di rilevante interesse tecnico-scientifico. I RAPPORTI TECNICI INGV sono pubblicati esclusivamente on-line per garantire agli autori rapidità di diffusione e agli utenti accesso immediato ai dati pubblicati. Un Editorial Board multidisciplinare ed un accurato processo di peer-review garantiscono i requisiti di qualità per la pubblicazione dei contributi.

RAPPORTI TECNICI INGV (RAPP. TEC. INGV) publish technological contributions (in Italian and/or in English) such as manuals, software, applications and implementations of instruments, and techniques of data collection. RAPPORTI TECNICI INGV are published online to guarantee celerity of diffusion and a prompt access to published data. A multidisciplinary Editorial Board and an accurate peer-review process provide the quality standard for the publication of the contributions.

MISCELLANEA INGV

ISSN 2039-6651

https://istituto.ingv.it/le-collane-editoriali-ingv/miscellanea-ingv.html

MISCELLANEA INGV (MISC. INGV) favorisce la pubblicazione di contributi scientifici riguardanti le attività svolte dall'INGV. In particolare, MISCELLANEA INGV raccoglie reports di progetti scientifici, proceedings di convegni, manuali, monografie di rilevante interesse, raccolte di articoli, ecc. La pubblicazione è esclusivamente on-line, completamente gratuita e garantisce tempi rapidi e grande diffusione sul web. L'Editorial Board INGV, grazie al suo carattere multidisciplinare, assicura i requisiti di qualità per la pubblicazione dei contributi sottomessi.

MISCELLANEA INGV (MISC. INGV) favours the publication of scientific contributions regarding the main activities carried out at INGV. In particular, MISCELLANEA INGV gathers reports of scientific projects, proceedings of meetings, manuals, relevant monographs, collections of articles etc. The journal is published online to guarantee celerity of diffusion on the internet. A multidisciplinary Editorial Board and an accurate peer-review process provide the quality standard for the publication of the contributions.

Progetto grafico
Barbara ANGIONI
Istituto Nazionale di Geofisica e Vulcanologia

Ufficio Editoriale
Francesca DI STEFANO | Coordinatore

Segreteria di redazione
segreteria.collane-editoriali@ingv.it

Editing | Produzione | Grafica redazionale
Barbara ANGIONI
Massimiliano CASONE
Francesca DI STEFANO
Patrizia PANTANI

© 2025 the Author(s). All rights reserved. Open Access



Creative Commons Attribution 4.0 International



ISTITUTO NAZIONALE DI GEOFISICA E VULCANOLOGIA