Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Almacenamiento y optimización de series temporales de medidas ambientales mediante TimescaleDB y caché Redis

Autor: Amando Antoñano Puerta
Tutor: Antonio Jesús Sierra Collado

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2025

Trabajo Fin de Grado
en Ingeniería de las Tecnologías de Telecomunicación

Almacenamiento y optimización de series temporales de medidas ambientales mediante TimescaleDB y caché Redis.

Autor:
Amando Antoñano Puerta

Tutor:
Antonio Jesús Sierra Collado

Dpto. Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2025

Trabajo Fin de Grado: Almacenamiento y optimización de series temporales de medidas ambientales mediante TimescaleDB y caché Redis.

Autor: Amando Antoñano Puerta

Tutor: Antonio Jesús Sierra Collado

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2025

El secretario del Tribunal

A mi familia y a mis amigos por su apoyo y paciencia sin límites.

Agradecimientos

Quiero expresar mi más sincero agradecimiento a todas las personas que me han acompañado y apoyado a lo largo de esta etapa universitaria.

En primer lugar, gracias a mi familia por su apoyo incondicional, por confiar en mí y apoyarme incluso en los momentos donde ya no tenía ánimos. Sin vuestro ánimo, este camino hubiera sido más empinado y con más obstáculos.

A mi tutor, Antonio Jesús Sierra Collado, por su orientación, paciencia y compromiso durante todo el proceso. Su experiencia y su guía han sido fundamentales para poder realizar este trabajo.

También quiero agradecer a mis amigos fuera y dentro de la carrera, no solo por darme animo cuando veía todo negro o compartir apuntes, sino también por todos los buenos momentos.

Finalmente, a todas aquellas personas que, de una manera u otra, han contribuido a mi formación y crecimiento durante estos años. Gracias por formar parte de este proceso.

Amando Antoñano Puerta

Sevilla, 2025

Resumen

En la actualidad, el tratamiento eficiente de grandes volúmenes de datos en tiempo real se ha convertido en un desafío clave dentro del ámbito tecnológico, especialmente en el contexto del Internet de las Cosas (IoT). La monitorización ambiental, y en particular el control de variables como la temperatura y la humedad es una de las áreas que más se ha beneficiado de este avance, permitiendo el desarrollo de soluciones inteligentes para distintos entornos.

En este contexto, el presente Trabajo de Fin de Grado se centra en el diseño e implementación de un sistema de obtención y almacenamiento de medidas de temperatura y humedad, haciendo uso de tecnologías especializadas en el tratamiento de series temporales y almacenamiento en caché. Para la toma de datos se emplea una placa Arduino conectada a un sensor DHT11, encargado de capturar periódicamente las lecturas del entorno. Estos datos se envían a una base de datos TimescaleDB, una extensión de PostgreSQL optimizada para el almacenamiento y consulta eficiente de series temporales.

Con el fin de mejorar el rendimiento del sistema en las consultas más recurrentes, se ha incorporado Redis como sistema de caché, permitiendo reducir el tiempo de respuesta y la carga sobre la base de datos. Gracias a esta arquitectura, el sistema logra ofrecer una solución eficaz para la recopilación, almacenamiento, y visualización de datos ambientales en tiempo real.

En este trabajo se ha desarrollado un sistema funcional capaz de recopilar, almacenar y consultar datos ambientales en tiempo real mediante el uso combinado de Arduino, TimescaleDB y Redis. Los resultados obtenidos muestran una mejora notable en la eficiencia y en los tiempos de respuesta, lo que valida la idoneidad de la arquitectura propuesta para escenarios IoT.

Abstract

Nowadays, the efficient processing of large volumes of real-time data has become a key challenge in the field of technology, particularly within the context of the Internet of Things (IoT). Environmental monitoring, and specifically the control of variables such as temperature and humidity, is one of the areas that has greatly benefited from these advancements, enabling the development of intelligent solutions for various environments.

In this context, the present Final Degree Project focuses on the design and implementation of a system for acquiring and managing temperature and humidity measurements, using technologies specialized in time-series data handling and caching. An Arduino board connected to a DHT sensor is used to periodically capture environmental readings. These measurements are then sent to a TimescaleDB database, a PostgreSQL extension optimized for the efficient storage and querying of time-series data.

To enhance system performance in frequent queries, Redis has been incorporated as a caching layer, reducing response times and easing the load on the main database. With this architecture, the system provides an effective solution for collecting, storing, and visualizing environmental data in real time.

The main objective of this project is to demonstrate how the integration of technologies such as Arduino, TimescaleDB, and Redis can lead to scalable and efficient systems for data management in IoT scenarios, laying the groundwork for future applications in fields such as smart homes, industrial monitoring, or precision agriculture.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

RAM	Memoria de Acceso Aleatorio
SSD	Unidad de Estado Sólido
CPU	Unidad Central de Procesamiento
TB	Terabyte
IDE	Entorno de Desarrollo Integrado
IOT	Internet de las Cosas
RFID	Identificación por Radiofrecuencia
SQL	Lenguaje de Consulta Estructurado
RLE	Codificación por Longitud de Ejecución
TTL	Tiempo de Vida
REDIS	Servidor de Diccionario Remoto

1 INTRODUCCIÓN

La ciencia puede divertirnos, pero es la ingeniería la que cambia el mundo.

- Isaac Asimov -

1.1 Motivación

En los últimos años, el crecimiento del internet de las cosas (IoT) ha impulsado el desarrollo de sistemas capaces de recopilar, procesar y analizar grandes volúmenes de datos en tiempo real. Esta tendencia ha despertado un creciente interés personal y académico por explorar soluciones que permitan una gestión eficiente de datos provenientes del entorno, especialmente aquellos relacionados con condiciones ambientales como la temperatura y humedad. Por esta razón, surgió la motivación por diseñar un sistema que integrara hardware de bajo coste con tecnologías de almacenamiento y consultas avanzadas, capaces de responder de forma ágil a las demandas de datos temporales.

El uso de una placa Arduino junto a un sensor de temperatura y humedad permite simular un escenario realista y accesible de adquisición de datos, ideal para proyectos de monitorización ambiental en hogares, invernaderos, laboratorios o entornos industriales. Por otro lado, el almacenamiento eficiente de series temporales representa un reto técnico que va más allá del simple registro de datos. La elección de TimescaleDB como base de datos no fue casual ya que su arquitectura está especialmente diseñada para trabajar con grandes volúmenes de datos distribuidos en el tiempo, ofreciendo consultas potentes y herramientas de agregación avanzadas.

Finalmente, la incorporación de Redis como sistema caché responde a la necesidad de optimizar el rendimiento del sistema en escenarios donde la latencia y la rapidez de acceso a la información son factores críticos. Poder combinar estas tecnologías en un mismo proyecto no solo ha supuesto un desafío técnico interesante, sino también una oportunidad para aprender a construir sistemas modernos y escalables que pueden ser aplicados a casos reales dentro del ámbito del IoT.

1.2 Objetivos

En esta sección, se enumeran los objetivos que se han marcado para el desarrollo del proyecto:

- Diseñar e implementar un sistema de adquisición de datos ambientales mediante una placa Arduino y un sensor DHT11 para medir temperatura y humedad de forma continua.
- Realizar un estudio sobre las tecnologías de PostgreSQL con la extensión de TimescaleDB, Docker y Redis para comprender sus características y cuáles podrían ser sus requisitos de integración.
- Utilizar Docker para contener la base de datos, el script JavaScript de inserción de datos y Redis para facilitar su despliegue, gestión, comunicación e integración.

1.3 Antecedentes

Este trabajo es una extensión del trabajo realizado por Enrique Sánchez Cardoso "Base de datos para Series Temporales y caché" [1], bajo la tutela de Antonio Jesús Sierra Collado. Ambos proyectos se centran en demostrar la eficacia de las tecnologías de gestión de datos para entornos de alto rendimiento, haciendo uso de TimescaleDB y Redis para optimizar el almacenamiento y la consulta de grandes volúmenes de información.

El proyecto parte de esa base teórica y técnica, y añade una capa práctica de adquisición de datos reales mediante hardware físico, concretamente una placa Arduino conectada a un sensor DHT11. Esta diferencia fundamental introduce una dimensión IoT al sistema, conectando el mundo físico con el entorno digital a través de sensores, haciendo que la generación de datos no sea artificial, sino proveniente de condiciones ambientales reales y en tiempo real. Esta innovación permite no solo la monitorización en entornos controlados, sino también el análisis de datos ambientales reales, ampliando así el abanico de aplicaciones potenciales del sistema.

Considerando los avances alcanzados en el campo de la monitorización y gestión de datos, este proyecto se propone complementar y ampliar las investigaciones previas mediante el uso de tecnologías actuales, optimizando la eficiencia en la captura, almacenamiento y consulta de datos en tiempo real.

1.4 Descripción de la solución

En este apartado se presenta una descripción detallada de la solución desarrollada en este proyecto. La solución tiene como objetivo principal proporcionar una arquitectura funcional para la adquisición, almacenamiento, procesamiento y consulta eficiente de datos ambientales (temperatura y humedad) en tiempo real, haciendo uso de tecnologías como Arduino, TimescaleDB y Redis. Esta solución integra componentes de hardware y software mediante una arquitectura modular, escalable y orientada a escenarios del Internet de las Cosas.

A lo largo de este apartado, se describen los componentes principales de la solución, así como su interconexión, flujos de datos y funcionalidades clave.

1.4.1 Objetivos Específicos

Los objetivos específicos de este Proyecto son los siguientes:

- **Captura de datos físicos en tiempo real:**

  Se utiliza una placa Arduino conectada a un sensor DHT11 para la adquisición periódica de datos de temperatura y humedad del entorno. Estas mediciones se obtienen a intervalos definidos mediante un *sketch*[1] en Arduino.

- **Procesamiento y envío de datos al backend:**

  Un script en JavaScript, ejecutado en un equipo local, se encarga de leer los datos transmitidos por el Arduino (vía puerto serie) y formatearlos adecuadamente para su inserción en la base de datos.

- **Persistencia temporal optimizada con TimescaleDB:**

  Almacenar los datos en una base de datos TimescaleDB, una extensión de PostgresSQL específicamente diseñada para el manejo de series temporales, que permite consultas complejas y eficientes sobre datos distribuidos en el tiempo incluyendo operaciones de agregación, filtrado por intervalo y ordenamiento cronológico.

- **Mejora del rendimiento mediante caché:**

  Implementar Redis como sistema de caché para reducir la carga sobre la base de datos y acelerar las respuestas a las consultas más frecuentes lo que reduce la latencia de respuesta.

[1] Forma de referirse a los programas que se cargan y ejecutan en la placa Arduino.

- **Contenerización del sistema:**

  Todos los componentes del sistema se ejecutan en contenedores Docker independientes, conectados a través de una red virtual Docker común.

1.4.2 Funcionalidades Principales

Las principales funcionalidades de este proyecto son las siguientes:

- **Adquisición de datos en tiempo real:**

  El sensor DHT11 envía datos cada cierto intervalo de tiempo mediante la placa Arduino, los cuales son procesados por un script que los inserta en la base de datos.

- **Almacenamiento de series temporales:**

  Cada dato de temperatura y humedad se registra en TimescaleDB con su correspondiente *timestamp*[2], permitiendo el seguimiento histórico y análisis temporal.

- **Consultas optimizadas mediante Redis:**

  Al realizarse una consulta, el sistema primero verifica si la respuesta ya está cacheada en Redis. Si es así, devuelve el resultado inmediatamente. Si no, realiza la consulta a la base de datos TimescaleDB, almacena el resultado en Redis y luego lo devuelve.

- **Contenedores para cada componente:**

  Toda la infraestructura está desplegada en contenedores Docker, lo que facilita la escalabilidad, el despliegue y la replicación en entornos diferentes.

1.4.3 Esquema de la Arquitectura

En la figura 1-1, se presenta un esquema con la estructura general del proyecto con el fin de facilitar la comprensión de este al lector. Podemos distinguir 5 elementos principales:

- **Placa Arduino + Sensor DHT11:**

  Captura los datos ambientales del entorno físico. El sensor toma medidas periódicamente, y las transmite vía puerto serie.

- **Script de procesamiento (JavaScript):**

  Lee los datos del sensor, los procesa y los envía para su inserción en la base de datos TimescaleDB.

- **Base de datos TimescaleDB:**

  Base de datos especializada en series temporales. Almacena los registros juntos con sus marcas de tiempo.

- **Sistema Caché Redis:**

  Sistema de almacenamiento en memoria para la caché de resultados de consultas recurrentes.

- **Contenedores Docker:**

  Cada uno de los servicios anteriores (TimescaleDB, Redis, script) se ejecuta en su contenedor respectivo

[2] Marca de tiempo que identifica cuándo ocurrió un evento, generalmente con precisión de milisegundos.

gestionado por Docker y que los conecta entre sí mediante una red virtual docker común.

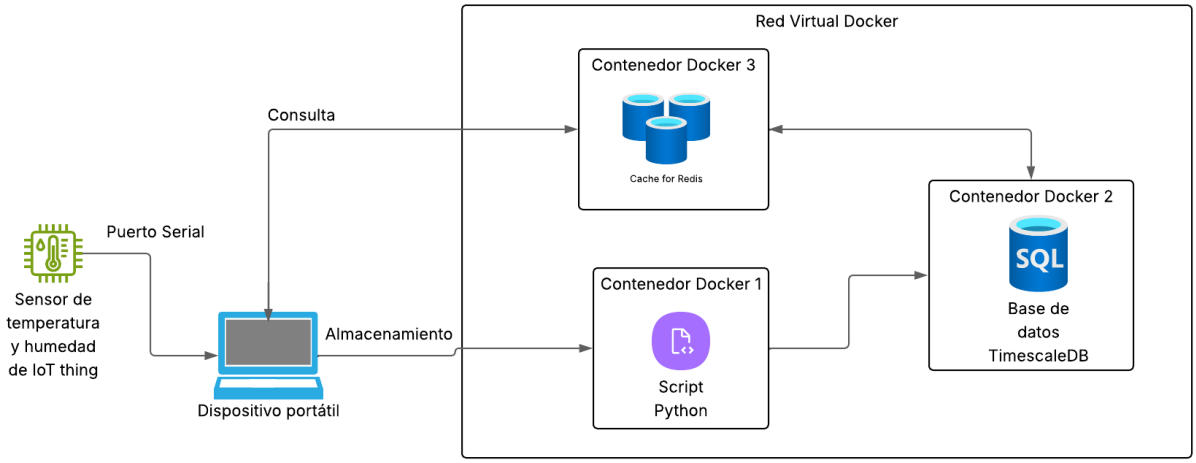

Figura 1-1. Arquitectura

1.5 Estructura de la memoria

En esta sección, se da un breve resumen de lo que trata cada apartado clave de la memoria:

A. Introducción:

Este capítulo inicial establece el contexto general del proyecto, exponiendo la motivación que ha impulsado su desarrollo, así como los objetivos planteados, los antecedentes relevantes y las funcionalidades. Se justifica la elección de las tecnologías empleadas y se presenta una visión preliminar de la arquitectura implementada para facilitar su comprensión al lector.

B. Recursos Utilizados:

En este apartado se detallan los recursos tanto hardware como softwares empleados para el desarrollo del sistema

C. Estado del arte:

Este capítulo recoge un análisis de las tecnologías fundamentales utilizadas en el proyecto, con especial atención a TimescaleDB y Redis. Se revisan sus características, arquitectura interna ventajas y casos de uso típicos, así como su adecuación para sistemas de series temporales y almacenamiento en caché.

D. Desarrollo del proyecto:

El núcleo técnico del trabajo se presenta en este apartado. Se describe de forma detallada el proceso de diseño, implementación y puesta en marcha del sistema completo. Se aborda la obtención de datos mediante Arduino, la transmisión y tratamiento de estos, su almacenamiento en TimescaleDB y la integración de Redis como capa de caché. También se incluye la estructura de los contenedores Docker y la interconexión entre los diferentes módulos.

E. Conclusiones y líneas futuras:

Este apartado recoge una reflexión sobre los resultados obtenidos y el cumplimiento de los objetivos propuestos. Se presentan tanto conclusiones técnicas como personales, valorando la experiencia adquirida a lo largo del proyecto. Además, se identifican posibles líneas de mejora y extensión del sistema, proponiendo futuras actuaciones.

En este capítulo se han expuesto los objetivos, la motivación y el contexto en el que se desarrolla el proyecto. Estos elementos permiten comprender la importancia de gestionar datos en tiempo real y la necesidad de diseñar un solución eficiente. En los capítulos siguientes se profundizará en las tecnologías disponibles y en cómo se aplican al caso de estudio.

2 RECURSOS UTILIZADOS

Durante este proyecto, se han utilizado múltiples recursos hardware y software para lograr su correcta implementación.

2.1 Recursos Hardware

2.1.1 Portátil

Para la implementación del proyecto, las pruebas y la redacción de este documento, se ha usado el portátil Acer TravelMate TMP215-53 que se puede ver en la figura 2-1.

Figura 2-1. Portátil Acer TravelMate TMP215-53.

Sus principales características son las siguientes [2]:

- Procesador Intel Core i5 de undécima generación.
- Pantalla de 15.6 pulgadas con resolución Full HD (1920x1080 pixeles).
- Memoria RAM de 8 GB.
- Memoria SSD de almacenamiento de 1TB.

2.1.2 Placa Arduino

Para programar y poner en funcionamiento el sensor de temperatura y humedad se utiliza una placa Arduino AZ-Delivery UNO que podemos observar en la figura 2-2.

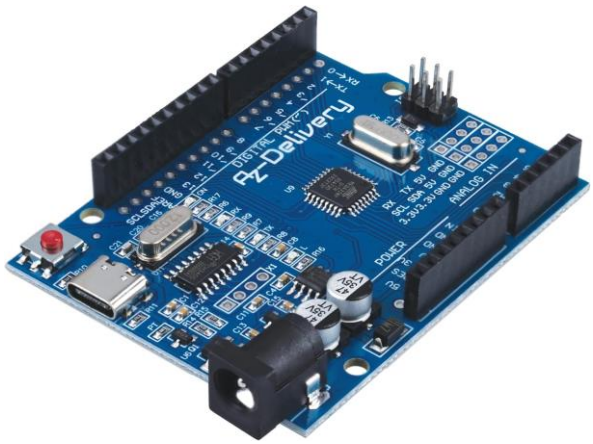

Figura 2-2. Placa Arduino AZ-Delivery UNO.

Sus principales características son [3]:

- Microcontrolador AZ-ATMega328.
- 14 pines de E/S digitales.
- 6 pines analógicos.
- 6 pines E/S digitales PWM.

2.1.3 Sensor DHT11

El dispositivo que toma las medidas de temperatura y humedad es el sensor DHT11 que se puede observar en la figura 2-3.

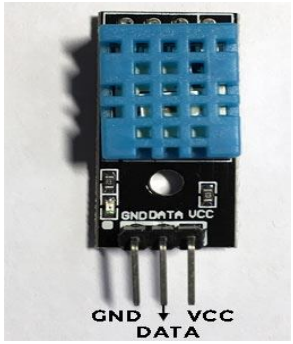

Figura 2-3. Sensor DHT11

Sus principales características son [4]:

- Tensión de alimentación de 5 voltios.
- Consumo típico en ejecución de 0.2 miliamperios.

- Rango de temperaturas de 0 a +50 grados Celsius.
- Rango de humedad de 20 a 90% de Humedad Relativa.

2.2 Recursos Software

2.2.1 Arduino IDE

El software Arduino, también llamado Arduino IDE, es una aplicación que permite escribir, compilar y cargar código en placas Arduino. Su editor de código está basado en C/C++ con funciones específicas para controlar entradas/salidas. Permite la lectura de sensores, controlar actuadores, enviar/Recibir datos vía puerto serie, comunicarte con otros dispositivos y automatizar procesos físicos simples. [5]

Figura 2-4. Arduino

2.2.2 JavaScript

JavaScript ha sido el lenguaje escogido para desarrollar el script de adquisición de datos. Es un lenguaje de programación de scripts, utilizado principalmente para añadir interactividad y contenido dinámico a las páginas web. lenguaje de programación interpretado, dinámico, basado en prototipos y asíncrono [6].

Figura 2-5. JavaScript

2.2.3 Node.js

Node.js ha sido elegido para ser el entorno de ejecución de la API. Es un entorno de ejecución JavaScript que se caracteriza por ser asíncrono y basado en eventos. [7]

Figura 2-6. Node.js

2.2.4 Redis

Es una base de datos en memoria RAM que funciona como un almacén de estructuras de datos clave-valor, lo que permite acceder a los datos con una rapidez excepcional. Esta característica lo convierte en una herramienta ideal para implementar sistemas de caché donde la velocidad de lectura y escritura es crítica. [8]

Figura 2-7. Redis

2.2.5 Ubuntu

El sistema operativo elegido para el desarrollo del script, la base de datos TimescaleDB, la ejecución en local de la base de datos y del script, el desarrollo de los contenedores Docker y su posterior despliegue, ha sido Ubuntu.

Figura 2-8. Ubuntu

2.2.6 Windows 10 PRO

El sistema operativo elegido para desarrollar el sketch de Arduino y su puesta en marcha es Windows 10 PRO.

Figura 2-9. Windows 10 PRO

2.2.7 TimescaleDB

TimescaleDB es una extensión de la base de datos relacional PostgreSQL optimizada para el almacenamiento y consulta eficiente de series temporales. En el proyecto, actúa como el repositorio principal de los datos ambientales obtenidos.

Figura 2-10. TimescaleDB

2.2.8 Docker

Docker es una herramienta de contenerización para desplegar de forma modular y aislada distintos componentes de un sistema. Permite una configuración homogénea y una gestión sencilla de dependencias, redes virtuales y volúmenes de persistencia. [9]

Figura 2-11. Docker

2.2.9 Mozilla Firefox

Es el navegador elegido para poder visualizar los resultados de las consultas a la base de datos.

Figura 2-12. Mozilla Firefox

2.2.10 Visual Studio Code

Es un editor de código gratuito y ligero de Microsoft. Se caracteriza por el soporte de múltiples lenguajes y entornos, autocompletado de código inteligente, depuración y una amplia variedad de extensiones para nuevas funcionalidades. [10]

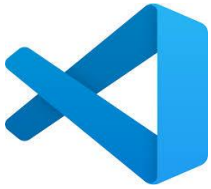

Figura 2-13. Visual Studio Code

En este capítulo se han descrito los recursos de hardware y software necesarios para el desarrollo del proyecto. La selección realizada responde a criterios de disponibilidad, eficiencia y compatibilidad. Estos recursos constituyen la base práctica sobre la que se construirá la solución.

3 ESTADO DEL ARTE

En este capítulo se revisan y analizan principales tecnologías, herramientas y enfoques relacionados con el diseño e implementación de sistemas de monitorización ambiental en entornos IoT, con especial énfasis en el uso de bases de datos de series temporales, sistemas de caché y contenedores para despliegue.

3.1 Contextualización: IoT y Monitorización ambiental

En este apartado hablaremos sobre IoT y su uso en la monitorización ambiental.

3.1.1 Definición y evolución del Internet de las Cosas (IoT)

El término Internet de las Cosas hace referencia al proceso de conectar los elementos físicos cotidianos al internet como sensores, actuadores, electrodomésticos etc. [11] Con el objetivo de recopilar, intercambiar y procesar datos de manera autónoma a través de redes de comunicación. El concepto ha ido evolucionando desde el reconocimiento de la importancia de la identificación y localización de dispositivos mediante RFID en la década de 2000, fue entonces cuando la madurez de las tecnologías inalámbricas y la electrificación masiva permitió la proliferación de soluciones IoT escalables [12].

En la actualidad, el IoT abarca desde aplicaciones de domótica básica (luces, climatización, alarmas) hasta sistemas industriales de gran envergadura (Industria 4.0), pasando por sectores como agricultura de precisión, salud y ciudades inteligentes [12]. En todos estos ámbitos, la característica común radica en la necesidad de monitorizar parámetros físicos como temperatura y humedad, y prestar servicios asociados como notificaciones, análisis predictivo o control remoto.

3.1.2 Importancia de la monitorización ambiental en IoT

La monitorización ambiental consiste en medir y recopilar de manera continua y sistemática distintas variables mediante sensores y dispositivos conectados [13]. Hay varias razones por las que es importante [13]:

- **Control de entornos peligrosos:** Lugares como laboratorios, almacenes de productos o invernaderos requieren de un control total de los factores ambientales para garantizar la seguridad e integridad de lo que se encuentra en su interior.
- **Aplicaciones en salud y seguridad:** Monitorear la calidad del aire en hospitales y espacios públicos ayuda a detectar a tiempo condiciones peligrosas.
- **Agricultura de precisión:** El control de la temperatura y humedad del suelo en cultivos puede mejorar la producción y reducir el consumo de agua.
- **Eficiencia energética:** La gestión de climatización y sistemas de ventilación se basa en datos ambientales que permiten optimizar su consumo energético.

3.1.3 Arquitecturas de sistemas IoT para monitorización ambiental

A continuación, se lista las capas que suelen formar parte de la mayoría de las arquitecturas IoT [14]:

- **Capa de percepción:** Es la base de la arquitectura IoT. Incluye los sensores y actuadores instalados en el entorno físico. En nuestro caso, es el sensor DHT11 junto con la placa Arduino.
- **Capa de red:** Es la responsable de transportar los datos recopilados por los dispositivos hacia sistemas centrales para su análisis y procesamiento. En este proyecto sería el enlace serial entre la placa Arduino y el equipo local.
- **Capa de procesamiento:** Engloba los servicios de almacenamiento, procesamiento, análisis y visualización. En nuestro caso, sería la base de datos TimescaleDB.
- **Capa de aplicación:** Representa las aplicaciones, interfaces o paneles de control para el usuario final. En este trabajo sería el navegador donde se muestran los resultados de las consultas a la base de datos.
- **Capa de seguridad:** Implementa autenticación, encriptación y control de accesos para proteger la integridad de datos. En nuestro caso, no implementamos esta capa.

3.2 Datos Temporales

A la hora de diseñar un sistema de monitorización ambiental, es preciso comprender las particularidades de los datos generados. Gestionar eficientemente las series temporales de datos de sensores (periódicos o por eventos) es un desafío característico de las aplicaciones IoT.

3.2.1 Definición de series temporales

Una serie temporal es una variable estadística cuyas observaciones están ordenadas cronológicamente [15]. En el contexto de la monitorización ambiental, cada observación comprende un valor de temperatura o humedad junto con una marca de tiempo que indica cuándo se hizo la lectura.

3.2.2 Componentes fundamentales de los datos de series temporales

A continuación, se enumeran los componentes fundamentales [16]:

- **Tendencia:** Dirección general de los datos a lo largo del tiempo, como aumento, disminución o constante.
- **Estacionalidad:** Patrones de datos que se repiten a lo largo de un conjunto de períodos de tiempo, como diario, mensual o anual.
- **Variaciones Cíclicas:** Patrones de datos que se repiten, pero no son estacionales y se producen a lo largo de varios años.
- **Variaciones irregulares:** Altibajos impredecibles que no se pueden explicar con otros componentes.

3.2.3 Retos en la gestión de series temporales

Durante este apartado se hablará de los siguientes retos en la gestión de retos temporales:

- **Escalabilidad horizontal o vertical:**

  La base de datos debe escalar sin perder rendimiento a medida que aumenta el número de sensores o la frecuencia de muestreo. En fases iniciales puede que la escalabilidad vertical sea suficiente, pero a largo plazo probablemente se necesite escalabilidad horizontal para distribuir carga

- **Control de la latencia de escritura:**

  Un aumento de la latencia de escrituras puede provocar la pérdida de datos o inconsistencias temporales por ello se debe garantizar que las inserciones de nuevos datos no se retrasen demasiado.

- **Gestión de grandes volúmenes históricos:**

  El almacenamiento indefinido de lecturas conduce en pocos años a volúmenes de información de terabytes o más. Es necesario implementar estrategias de compresión, particionado temporal y políticas de retención.

- **Optimización de consultas de agregados:**

  Cálculos de estadísticas en rangos temporales muy amplios pueden provocar sobrecarga en tiempo de cómputo.

- **Consistencia temporal:**

  Asegurar que las marcas de tiempo sean comparables y constantes entre nodos es un reto.

- **Integración con sistemas de caché:**

  Para reducir la latencia de acceso a los datos se recurre a sistemas de caché en memoria, lo cual añade complejidad arquitectónica.

3.3 Plataformas para Series Temporales: Enfoque en TimescaleDB

Existen muchas alternativas en el mercado para afrontar el desafío de gestionar series temporales. A continuación, se realiza una revisión comparativa de las más relevantes, con especial profundización en TimescaleDB, la tecnología seleccionada en este proyecto.

3.3.1 Bases de datos especializadas en series temporales

En este apartado se realizará una comparativa entre los distintos sistemas especializados de bases de datos de series temporales:

- **InfluxDB:**
  - **Descripción:** Base de datos de código abierto diseñada exclusivamente para datos de series temporales. Emplea un motor de almacenamiento propio optimizado para escrituras frecuentes y compresión de datos [17].
  - **Ventajas:** Alta velocidad de escritura, compresión de datos eficiente y lenguaje de consulta especifico enfocado en operaciones temporales [17].
  - **Desventajas:** Menos ecosistema SQL genérico y algunos componentes requieren licencias en el caso de la versión de empresa.
- **OpenTSDB:**
  - **Descripción:** Es una base de datos de series temporales distribuida y escalable montada sobre *Hbase*[3] [18].
  - **Ventajas:** Escalabilidad horizontal ilimitada [19].
  - **Desventajas:** Complejidad de despliegue y mantenimiento, latencias de consulta más elevadas para agregaciones complejas y una curva de aprendizaje elevada [19].
- **Graphite:**
  - **Descripción:** Herramienta orientada a métricas de infraestructuras como servidores o redes [20].
  - **Ventajas:** Fácil instalación y buena integración con herramientas de visualización [20].
  - **Desventajas**: Baja escalabilidad al almacenar los datos en archivos locales y carece de lenguaje de consulta SQL [20].

[3] Sistema de gestión de base de datos NoSQL distribuida de código abierto.

- **QuestDB:**
    - **Descripción:** Base de datos de series temporales de código abierto que ofrece una ingesta ultrarrápida de datos y consultas SQL dinámicas de baja latencia [21].
    - **Ventajas:** Ingesta de alta velocidad de datos, rendimiento solido en hardware limitado y formato de almacenamiento en columnas [21].
    - **Desventajas:** Ecosistema y comunidad más reducidos que PostgreSQL.
- **Prometheus:**
    - **Descripción:** Es un motor de código abierto para monitorización de sistemas que recolecta métricas a través de un modelo basado en HTTP [22].
    - **Ventajas:** Lenguaje de consulta flexible, no depende del almacenamiento distribuido y compatibilidad con múltiples modos de gráfico y paneles [22].
    - **Desventajas:** Su modelo de retención de datos y escalabilidad nativa está pensado para la monitorización de infraestructuras, no para datos IoT heterogéneos a gran escala.

3.3.2 Bases de datos relacionales con extensiones para series temporales

En este apartado se hablará sobre PostgreSQL y las distintas extensiones disponibles para el manejo de series temporales y se analizará en profundidad TimescaleDB, que es la que se ha elegido para implementar en este proyecto.

3.3.2.1 PostgreSQL con extensión para series temporales

PostgreSQL es una base de datos relacional que ofrece fiabilidad, robustez, integridad transaccional y un amplio abanico de extensiones [23]. Algunas de las extensiones ofrecen capacidades específicas de series temporales:

- **TimescaleDB:** Es la extensión más relevante ya que implementa *hypertables*[4] y *chunks*[5] que facilitan la inserción masiva, la compresión de datos y la creación de agregaciones continuas (vistas materializadas que se actualizan automáticamente de manera incremental).
- **PipelineDB:** Orientada a procesamiento continuo de flujo de datos (ya no está mantenida oficialmente).
- **Cstore_fdw:** Es una extensión que proporciona almacenamiento columnar en PostgreSQL, útil para lectura analítica en lotes, pero no está optimizada para inserciones frecuentes.

Por un lado, las ventajas de usar PostgreSQL con extensiones son: La integridad transaccional y soporte completo SQL estándar, un ecosistema maduro con multitud de herramientas, facilidad de migración y portabilidad de datos [24]. Por otra parte, las desventajas son un *overhead*[6] de las transacciones ACID para inserciones de alta frecuencia si no se configura correctamente [25] y tener que ejecutar la base de datos en versiones recientes para obtener las mejoras de optimización [26].

[4] Tablas de PostgreSQL que se particionan automáticamente por tiempo, lo que las convierte en una forma eficiente de almacenar y consultar datos de series temporales.
[5] Unidades fundamentales de almacenamiento dentro de las hipertablas.
[6] Se refiere al costo, en términos de rendimiento y recursos, que implica garantizar que las transacciones cumplan con las propiedades ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad)

3.3.2.2 TimescaleDB

A continuación, se analiza en profundidad TimescaleDB, ya que es la tecnología que se ha utilizado en el proyecto:

1. **Arquitectura y fundamentos de TimescaleDB:**
   - **Hypertables:** Una hypertable se define como una tabla virtual que se particiona en múltiples tablas físicas denominadas chunks. Cada chunk corresponde a un intervalo de tiempo y puede distribuirse en múltiples nodos [27]. La definición de una hypertable se realizaría de la siguiente manera:

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

Figura 3-1. Definición de una Hypertable.

Se crea una hypertable a partir de una tabla normal creada anteriormente de nombre "conditions" y se le añade la columna temporal obligatoria que es la que contiene los timestamp.

   - **Indexación:** Para agilizar consultas por rango temporal, TimescaleDB usa índices compuestos por la columna de timestamp más otra columna clave. Un ejemplo sería este:

```
CREATE INDEX ON readings (time DESC, sensor_id);
```

Figura 3-2.Crear Índice.

Se crea un índice en "readings" en orden descendente porque los filtros temporales son prioritarios y luego la segunda columna del índice es de "sensor_id".

   - **Compresión:** TimescaleDB utiliza un subsistema de compresión que aplica algoritmos de tipo RLE para reducir el tamaño de los históricos. Un ejemplo sería este:

```
ALTER TABLE readings SET (
  timescaledb.compress,
  timescaledb.compress_orderby = 'time DESC',
  timescaledb.compress_segmentby = 'sensor_id'
);
SELECT add_compression_policy('readings', compress_after => INTERVAL '7 days');
```

Figura 3-3.Compresión datos.

Con esto conseguiremos que las lecturas de hace más de 7 días se compriman, manteniendo los datos actualizados y optimizando el espacio en disco.

   - **Agregados Continuos:** Nos permite preprocesar y almacenar resultados de consultas de agregación en ventanas temporales, reduciendo drásticamente el tiempo de ejecución para consultas históricas frecuentes. Un ejemplo sería:

```
CREATE MATERIALIZED VIEW readings_hourly
  WITH (timescaledb.continuous) AS
SELECT time_bucket('1 hour', time) AS hour,
       sensor_id,
       AVG(temperature) AS avg_temp,
       MAX(temperature) AS max_temp,
       MIN(temperature) AS min_temp
FROM readings
GROUP BY hour, sensor_id;
```

Figura 3-4.Creación de Vista Materializada.

La vista materializada "readings_hourly" se actualizará de forma incremental a medida que lleguen datos nuevos.

- **Políticas de retención y particionado:** Se pueden aplicar reglas automáticas de eliminación de chunks antiguos: Un ejemplo sería:

```
SELECT add_retention_policy('readings', drop_after => INTERVAL '30 days');
```

Figura 3-5.Creación de política de eliminación.

Ahora, cualquier chunk cuya ventana temporal supere los 30 días se eliminará, manteniendo un tamaño de base de datos controlado.

- **Paralelismo en consultas:** TimescaleDB explota las capacidades de PostgreSQL para consultas paralelas, de forma que las agregaciones sobre chunks múltiples se ejecutan en paralelo, reduciendo tiempos de respuesta en análisis de grandes volúmenes.

2. Comparativa de rendimiento

Diversos estudios han comparado el rendimiento de TimescaleDB con otras bases de datos de series temporales como InfluxDB y QuestDB. A continuación, se presentan los hallazgos más relevantes:

- **Velocidad de inserción:**

  InfluxDB logra una tasa de inserción de aproximadamente 330.000 puntos de datos por segundo con 4 millones de series temporales únicas, mientras que TimescaleDB alcanza unos 480.000 puntos por segundo. Es cierto que InfluxDB tiene un rendimiento algo mejor en inserciones puras debido a su motor TSM optimizado, TimescaleDB puede mejorar su rendimiento con la configuración adecuada de algunos parámetros, el uso de hypertables multimodo y mayor paralelismo [28].

- **Costo de almacenamiento:**

  TimescaleDB puede alcanzar ratios de compresión de 3:1 a 5:1 en datos históricos. Aunque InfluxDB también incorpora compresión, en escenarios de datos heterogéneos, se ve superada por la eficiencia de TimescaleDB [29].

- **Flexibilidad del lenguaje SQL:**

  A diferencia de *InfluxQL*[7] (InfluxDB) o *PromQL*[8] (Prometheus), TimescaleDB soporta SQL estándar con extensiones propias. Esto facilita la integración con sistemas existentes, herramientas de BI y arquitecturas que ya emplean PostgreSQL [30].

[7] Lenguaje de consulta de tipo SQL diseñado para interactuar con InfluxDB, una base de datos de series temporales.
[8] Lenguaje de consulta funcional utilizado para interrogar y agregar datos de series temporales recopilados por el sistema de monitorización Prometheus

- **Consultas de agregados:**

  TimescaleDB destaca en consultas complejas sobre rangos largos de tiempo gracias a sus agregaciones continuas. Estas vistas materializadas incrementales permiten tiempos de respuesta muy reducidos, en el orden de milisegundos, para estadísticas diarias o semanales. Además, las políticas de agregación continua han sido optimizadas en versiones recientes, reduciendo el uso de recursos del sistema y mejorando la eficiencia [31].

3. Caso de uso en monitorización de temperatura y humedad

En este trabajo, la arquitectura que se propone requiere gestionar las lecturas de temperatura y humedad que envía el sensor DHT11 cada cierto intervalo de tiempo. Las operaciones vitales que se beneficiarán de TimescaleDB son:

- Inserciones continuas: El script de JavaScript ejecuta una inserción por cada lectura recibida por el sensor DHT11. Gracias a la partición por chunk, estas medidas se dirigen siempre al chunk del día en curso, reduciendo la contención de índices.
- Consulta de valores recientes: Si se desea obtener los valores de las mediciones de la última hora es bastante sencillo ya que basta con filtrar en la hypertable aprovechando los índices en la columna "time". Un ejemplo de lo anterior sería:

```
SELECT * FROM sensor_data
WHERE time > NOW() - INTERVAL '1 hour'
ORDER BY time DESC;
```

Figura 3-6. Ejemplo Filtrado por time.

- Agregados diarios/semanales: Se hace uso de las "Vistas Materializadas" para obtener promedios, máximos y mínimos, con actualizaciones incrementales. Un ejemplo de lo anterior sería:

```
CREATE MATERIALIZED VIEW metrics_hourly
WITH (timescaledb.continuous) AS
SELECT
    sensor_id,
    time_bucket('1 hour', time) AS bucket,
    AVG(value) AS avg_value,
    MAX(value) AS max_value,
    MIN(value) AS min_value
FROM sensor_data
GROUP BY sensor_id, bucket
WITH DATA;
```

Figura 3-7. Ejemplo de Agregación continua.

3.4 Sistemas de Caché en entornos IoT: Enfoque en Redis.

En soluciones IoT donde la base de datos de series temporales recibe miles de lecturas diarias y tiene que atender consultas de múltiples usuarios, la incorporación de un sistema caché en memoria es esencial para reducir la latencia de consultas frecuentes o complejas. En esta sección del trabajo, se analiza el papel de la memoria caché, se comparan alternativas y se profundiza en Redis como tecnología elegida.

3.4.1 Rol de la Memoria Caché

Un Sistema de caché interpuesto entre la capa de almacenamiento permanente (Base de datos TimescaleDB) y las capas de consultas tiene las siguientes funciones:

- **Reducción de latencia:**

  Al almacenar en la memoria caché los resultados de consultas ya ejecutadas como por ejemplo la temperatura media diaria, cuando se realiza una consulta idéntica, se devuelve el resultado directamente desde el caché en lugar de volver a tener que calcularlo en la base de datos permanente.

- **Escalabilidad Horizontal:**

  Al desplegar instancias replicadas de un sistema de caché, es posible distribuir la carga de lectura del sistema. El caché puede repetirse en múltiples réplicas para servir muchas peticiones simultáneas sin saturar la capa persistente.

- **Menos carga en la base de datos:**

  La memoria caché hace de primer frente de respuesta, lo que disminuye el numero de consultas a TimescaleDB y la reserva para operaciones críticas o consultas complejas no cacheadas.

- **Control de expiración y consistencia:**

  El sistema caché debe tener políticas que indiquen cuándo un valor almacenado es obsoleto. En este caso, un TTL corto de unos 60 segundos para datos recientes puede asegurar que las lecturas reflejan cambios casi en tiempo real.

3.4.2 Comparativa general de soluciones de Sistemas Caché

A continuación, se listan las distintas soluciones de sistemas caché:

- **Memcached**
  - **Descripción:** Es un almacén de datos en la memoria de alto rendimiento y fácil de usar [32].
  - **Ventajas:** Tiempos de respuesta por debajo del milisegundo, simplicidad y facilidad de uso, escalabilidad y buena comunidad [32].
  - **Desventajas:** No tiene persistencia incorporada, no soporta estructuras complejas y no admite replicación nativa ni *clustering*[9] automático [33].
- **Redis**
  - **Descripción:** Es un almacén de estructura de datos de valores de clave en memoria rápido y de código abierto [34].
  - **Ventajas:** Desempeño muy rápido, estructuras complejas de datos en memoria, versatilidad, facilidad de uso, replicación asíncrona, persistencia con *snapshots*[10] y gran compatibilidad con muchos lenguajes de programación [34].
  - **Desventajas:** Consumo de memoria relativamente elevado.
- **Apache Ignite**
  - **Descripción:** Es una plataforma distribuida en memoria que funciona como caché, base de datos y sistema de procesamiento [35].
  - **Ventajas:** Integración de caché y procesamiento, tolerancia a fallos y persistencia opcional [35].
  - **Desventajas:** Complejidad de despliegue, curva de aprendizaje elevada y sobredimensionado para IoT simple.
- **Couchbase (en modo caché)**
  - **Descripción:** Base de datos NoSQL que mediante su módulo de “couchbase bucket” en

[9] Técnica de análisis de datos que agrupa elementos similares en conjuntos denominados "clústeres".
[10] Copia lógica de un sistema de archivos, base de datos o incluso una máquina virtual en un momento específico.

memoria, ofrece funcionalidades de caché [36].

- **Ventajas:** Integración nativa con capa de persistencia en disco, alto rendimiento, desarrollo simple, versátil [36].
- **Desventajas:** Complejidad de configuración.

En entornos IoT donde prima la alta velocidad y se requiere de una caché de bajo nivel, Redis se considera la mejor opción, debido al amplio conjunto de estructuras de datos que maneja y a su modelo de replicación y persistencia configurable.

3.4.3 Arquitectura de Redis y características relevantes

Redis es una base de datos en memoria orientada a pares clave-valor con persistencia opcional.

3.4.3.1 Estructura de datos

En esta sección se listarán los distintos tipos de datos que maneja Redis [37]:

- **Strings:** Estructura de datos binaria segura. Puede almacenar cualquier tipo de datos: una cadena, un entero, un valor de punto flotante, una imagen JPEG, un objeto Ruby serializado o cualquier otra cosa que desee que tenga.
- **Hashes:** Almacena un conjunto de pares campo-valor.
- **Lists:** Contienen colecciones de elementos de cadena ordenados según su orden de inserción.
- **Sets:** Almacena un conjunto único de miembros.
- **Sorted Sets:** Contienen un conjunto único de miembros ordenados por puntuaciones de punto flotante
- **Bitmaps:** Es una estructura de datos compacta para almacenar lógica y estados binarios.
- **Bitfields:** Ofrecen una forma eficiente y compacta de implementar múltiples contadores en una sola matriz.
- **HyperLogLog:** Es una estructura de datos probabilística que se usa para contar valores únicos con un tamaño de memoria constante.
- **Geospatial indexes:** Proporcionan una forma extremadamente eficiente y sencilla de gestionar y utilizar los datos geoespaciales en Redis.
- **Streams:** Es una estructura de datos increíblemente potente para gestionar flujos de datos de alta velocidad.

3.4.3.2 Persistencia

Redis ofrece dos modos principales [38]:

- **AOF (Append Only File):**

  Registra cada operación de escritura en un archivo de disco en tiempo real. Este mecanismo garantiza que, al reiniciar el servidor, todas las operaciones pueden volverse a ejecutar, recuperando así los datos incluso después de una caída abrupta.

- **RDB (Redis Database Backup):**

  Genera snapshots periódicos de la base de datos en un archivo con extensión ".rdb", configurables por números de escrituras o intervalos de tiempo. Su ventaja es el bajo impacto en el rendimiento, pero no garantiza la persistencia de cada operación entre snapshots.

3.4.3.3 Replicación y alta disponibilidad

Redis ofrece múltiples mecanismos para garantizar alta disponibilidad y tolerancia a fallos. Entre ellos destacan tres enfoques [39]:

- **Redis Máster-Slave:**

  Un nodo maestro (master) gestiona las operaciones de escritura y replica los cambios en uno o más nodosos esclavos (slaves), permitiendo lecturas escalables.

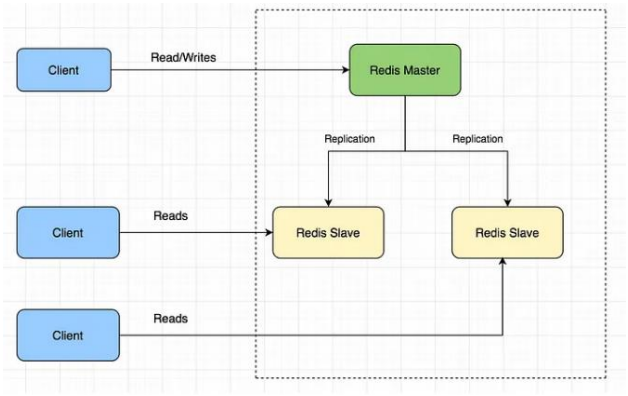

Figura 3-8. Redis Master-Slave.

- **Redis Sentinel:**

  Monitorea nodos maestros/esclavos, detecta fallos y promueve un esclavo a maestro si el maestro original cae.

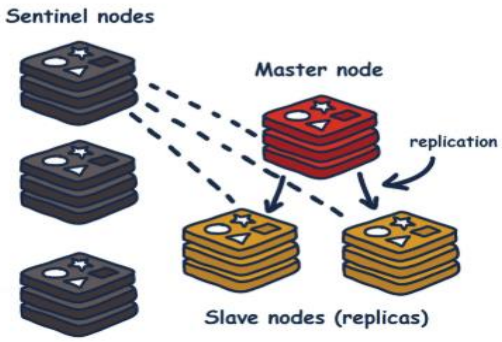

Figura 3-9. Redis Sentinel.

- **Redis Cluster:**

  Fragmenta el espacio de claves en *slots*[11] y distribuye nodos maestros y esclavos para escalabilidad y tolerancia a fallos.

[11] Representan la manera en que Redis divide el espacio de claves para distribuir las operaciones entre los diferentes nodos del clúster.

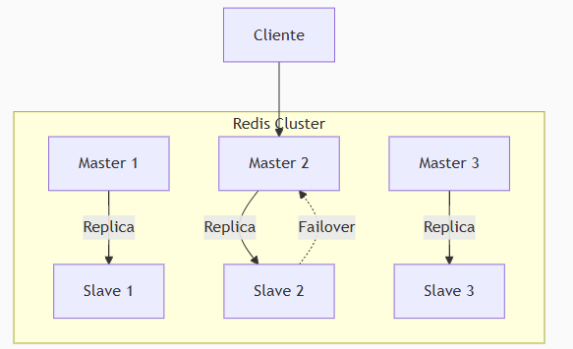

Figura 3-10. Redis Cluster.

3.4.4 Uso de Redis como capa caché en monitorización ambiental.

En el escenario de recopilación de datos de temperatura y humedad, las consultas más frecuentes podrían clasificarse en:

- **Lecturas en vivo:** Solicitudes de las lecturas más recientes para graficar en tiempo real.
- **Agregados de corto plazo:** Promedios, mínimos y máximos de la última hora.
- **Agregados prolongados:** Estadísticas diarias o semanales.
- **Datos históricos puntuales:** Lectura de eventos específicos o comparaciones.

Redis puede cachear las lecturas en vivo almacenando en una lista o Sorted los últimos N valores ordenados por timestamp y cuando un cliente solicite los "últimos 15 valores", la aplicación lee directamente del caché. Para los agregados de corto plazo, se calcula la consulta en TimescaleDB y se almacena el resultado en Redis con su clave y con TTL de 60 segundos para que la clave expire al pasar ese tiempo y la siguiente consulta recalcule. En el caso de los agregados prolongados se utilizarían vistas materializados y un TTL superior, de 3600 segundos por ejemplo ya que las estadísticas horarias para días anteriores cambian solo cuando TimescaleDB recalcule la vista materializada. Por último, para los datos históricos puntuales, como no se consultan con frecuencia, no hace falta cachearlos y se dirige la solicitud directamente a TimescaleDB.

3.5 Contenerización y Despliegue: Enfoque en Docker

El uso de contenedores ha revolucionado la forma de desarrollar, probar y desplegar aplicaciones, ofreciendo entornos reproducibles, aislados y ligeros. En proyectos IoT que integran múltiples componentes, Docker facilita la orquestación y gestión de dependencias.

3.5.1 Concepto de contenedor y diferencias con máquinas virtuales

Los contenedores docker proporcionan un entorno aislado para ejecutar aplicaciones, incluyendo todos los componentes necesarios para su funcionamiento, como librerías y configuraciones. Esta solución, más ligera que las máquinas virtuales tradicionales, se beneficia del uso compartido del *kernel*[12] del sistema operativo, lo que reduce el consumo de recursos y acelera el inicio del servicio.

[12] Parte fundamental del sistema operativo que se ejecuta en modo privilegiado para gestionar los recursos del sistema y permitir la comunicación entre el hardware y el software.

Las principales ventajas de emplear Docker son [40]:

- **Aislamiento:** Cada contenedor corre en su espacio aislado, evitando conflictos de versiones o puertos TCP.
- **Portabilidad:** Los contenedores construidos en un sistema operativo (Windows, macOS, Linux) se despliegan igual en otro host con Docker instalado.
- **Reproducibilidad:** Garantiza que las versiones de dependencias y configuraciones sean idénticas en todos los entornos.
- **Ligereza:** Un contenedor ocupa *megabytes*[13], mientras que una máquina virtual ocupa *gigabytes*[14].
- **Escalabilidad:** Con herramientas de orquestación, se puede conseguir balancear la carga y mantener una alta disponibilidad mediante la replicación de contenedores.

3.5.2 Docker en arquitecturas IoT

En IoT, Docker se utiliza como herramienta habitual para desplegar:

- **Servidores de bases de datos:** InfluxDB, PostgreSQL etc.
- **Sistemas de caché:** Memcached, Redis.
- **Dashboards y aplicaciones web:** Grafana, Node-RED, Kibana.
- **APIs y microservicios:** Servicios REST que reciben datos de sensores y los encaminan a la base de datos
- **Gateways o microservicios de edge computing:** Raspberry Pi o micro-PCs que ejecutan contenedores para preprocesar datos localmente.

Una de las ventajas clave que ofrece Docker en el contexto IoT es la facilidad para crear y gestionar redes internas virtuales entre contenedores. Esta funcionalidad permite que los diferentes servicios que conforman la arquitectura de solución IoT se comuniquen entre si de forma directa, segura y eficiente. Docker permite definir redes personalizadas en las que cada contenedor puede ser identificado por un nombre de servicio, lo cual simplifica enormemente la configuración de los sistemas distribuidos [41].

En el caso de este proyecto, donde un contenedor ejecuta un script de adquisición de datos, otro gestiona la base de datos de series temporales (TimescaleDB) y un tercero actúa como caché (Redis), todos estos servicios pueden intercambiar información sin necesidad de exponer sus puertos al exterior. Esto mejora la seguridad, favorece el aislamiento del sistema y reduce la necesidad de configuraciones de red complejas.

3.5.3 Docker Compose para orquestación local

En entornos de desarrollo y prueba, Docker Compose es la herramienta que permite desplegar múltiples contenedores mediante un solo fichero llamado “docker-compose.yml”.

Las principales partes de este archivo son [42]:

- **Version:** Define la versión del esquema de Compose. Se pone al principio.

```yaml
yaml

version: '3.8'
```

Figura 3-11. Docker Compose-Version.

[13] Unidad estándar en la informática y la tecnología digital que indica el tamaño de un archivo o la capacidad de una memoria de datos. Equivale a 1 millón de bytes.
[14] Unidad de medida de almacenamiento de datos que equivale a 1.000 millones de bytes.

- **Services:** Es la sección principal donde se especifican los contendores que forman parte del sistema, junto con su configuración.

```yaml
services:
  web:
    build: .
    ports:
      - "5000:5000"
  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD: example
```

Figura 3-12.Docker Compose-Services.

- **Volumes:** Permite definir volúmenes persistentes que pueden ser montados por uno o más contenedores.

```yaml
volumes:
  - timescale_data:/var/lib/postgresql/data
```

Figura 3-13.Docker Compose-Volumes.

- **Networks:** Define redes personalizadas para conectar servicios, aunque Docker Compose crea una por defecto si no se especifica ninguna. En el ejemplo de abajo el servicio web puede comunicarse solo con el servicio app porque pertenecen a la misma red virtual.

```yaml
  web:
    image: nginx
    networks:
      - frontend

  app:
    build: ./app
    networks:
      - frontend
      - backend
```

Figura 3-14.Docker Compose-Networks.

- **Environment:** Las variables de entorno se puede definir directamente sobre el archivo o importarse desde un archivo de extensión ".env".

```yaml
environment:
  POSTGRES_USER: tfg_user
  POSTGRES_PASSWORD: tfg_pass
  POSTGRES_DB: tfg_db
```

Figura 3-15. Docker Compose-Environment.

- **Depends_on:** Establece el orden de inicio de los servicios, no garantiza que uno este ya levantado, pero sí que se ha iniciado. En el ejemplo de abajo el servicio backend debe esperar a que inicien los servicios db y redis.

```yaml
backend:
  build: ./backend
  depends_on:
    - db
    - redis

db:
  image: postgres

redis:
  image: redis:alpine
```

Figura 3-16. Docker Compose-Depends_on

3.5.4 Beneficios y limitaciones de Docker en IoT

A continuación, se analizan los beneficios y limitaciones de Docker en entornos IoT.

Los beneficios son los siguientes:

- **Aislamiento de entornos:** El contenedor del script en JavaScript no entrará en conflicto con las versiones de JavaScript o librerías instaladas en el host.
- **Reproducibilidad:** Compartir el repositorio con el docker-compose y los dockerfile permite a cualquier persona reproducir el entorno tal cual sin tareas manuales de configuración.
- **Despliegue rápido:** Con un solo comando "docker-compuse up -d", se inician todos los servicios interconectados.

Las limitaciones son las siguientes:

- **Sobrecarga en dispositivos:** En plataformas hardware limitadas (poca CPU o RAM).
- **Persistencia de datos:** Si no se configura correctamente los volúmenes, los datos pueden perderse al destruir los contenedores.
- **Complejidad inicial:** Curva de aprendizaje un poco elevada para usuarios nuevos.

En conclusión, se han analizado las tecnologías y enfoques existentes en la gestión de series temporales y almacenamiento en caché. Esta revisión ha permitido identificar las ventajas y limitaciones de diferentes propuestas. Como resultado, se ha justificado la selección de TimescaleDB, Redis y Docker como base de la solución.

4 DESARROLLO DEL PROYECTO

A continuación, en este capítulo clave del proyecto se describirán varios aspectos esenciales: La arquitectura general del sistema, Adquisición y tratamiento de datos, Persistencia de datos en TimescaleDB, Implementación de la capa caché con Redis, Visualización de datos mediante API REST, Contenerización y despliegue y Monitorización y logs.

4.1 Arquitectura General del Sistema

El sistema desarrollado integra la adquisición de los datos ambientales, pasando por su almacenamiento en una base de datos y terminando por una consulta optimizada mediante una capa de caché.

Para ello se han utilizado tecnologías de código abierto y de contenedores, garantizando así un despliegue modular y fácilmente reproducible.

Los componentes principales son los siguientes:

- **Hardware:** Placa Arduino con sensor DHT11 conectado por puerto USB al host.
- **Software de adquisición de datos:** Un servicio *Node.js* (*busConnector.js*) que lee los datos del puerto serie y los inserta en la base de datos.
- **Base de datos de series temporales:** TimescaleDB, desplegada en un contenedor Docker y cuya función es almacenar los datos de humedad y temperatura.
- **Capa de caché:** Redis, para almacenar temporalmente los resultados de las consultas más frecuentes, reduciendo la latencia.
- **API REST:** Servicio *Node.js* (*index.js*) que expone rutas GET para consultas con y sin caché, documentadas mediante swagger.
- **Orquestación:** Docker Compose, define y lanza los tres servicios principales: API Node.js, TimescaleDB y Redis.

En el siguiente diagrama se ilustra de manera sencilla la arquitectura del sistema:

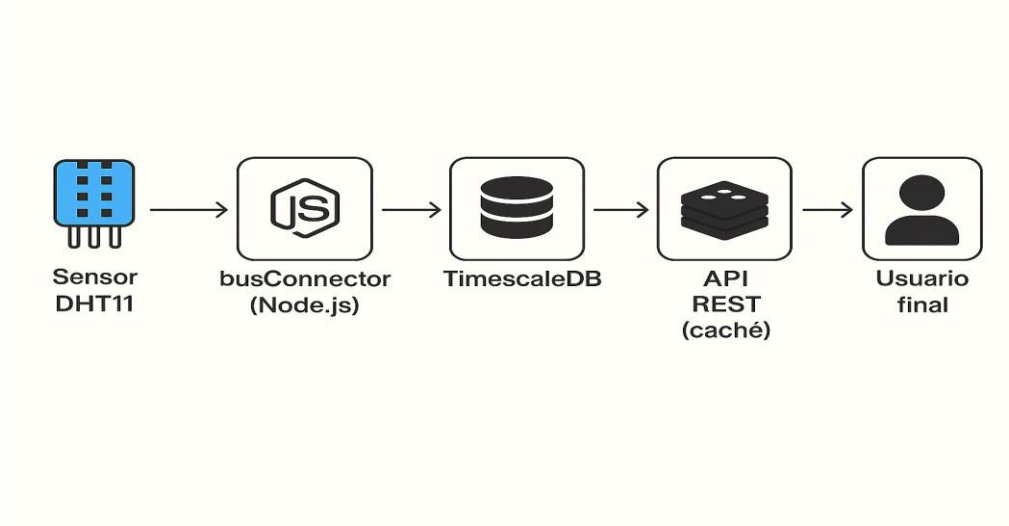

Figura 4-1. Arquitectura General del Sistema

4.2 Adquisición y tratamiento de datos

Por un lado, la captura de datos se realiza mediante la placa Arduino, que está conectada al host a través del puerto mapeado "dev/ttyUSB0". Por otra parte, el sensor tiene un sketch que lo programa para tomar las medidas de temperatura y humedad cada segundo y enviarla por el puerto serial.

El servicio *Node.js* a través del *busConnector.js* realiza las siguientes acciones:

- Abre el puerto serie y escucha tramas del sensor DHT11 donde cada par de lecturas recibidas corresponde a humedad y temperatura.
- Inserta los valores en la tabla *medidas_sensor* de la base de datos, acompañados de una marca de tiempo generada automáticamente.

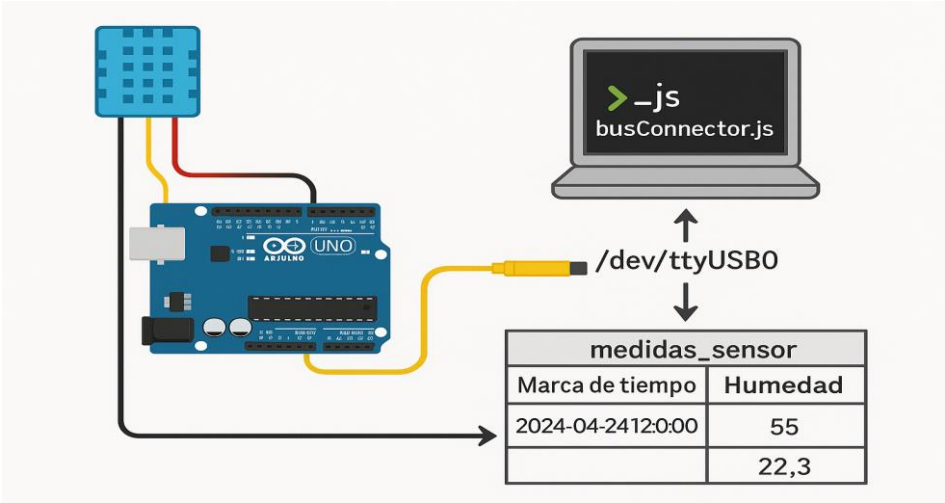

Figura 4-2. Adquisición y Tratamiento de datos

El módulo *busConnector.js* es responsable de recibir los datos del sensor desde el puerto serie e insertarlos en la base de datos. Algunos puntos importantes de su implementación son los que se muestran en la siguiente tabla:

Aspecto	Descripción
Lectura del puerto serie	Usa la librería *serialport* para abrir el dispositivo /Dev/ttyUSB0 a 9600 a una velocidad de 9600 baudios que coincide con la del sensor.
Procesamiento de datos	Almacena primero la humedad y después la temperatura antes de la inserción. Además, aplica un pequeño filtro para descartar datos no numéricos.
Inserción en base de datos	Emplea Knex para insertar los registros de humedad y temperatura en la tabla *medidas_sensor* con una marca de tiempo.
Gestión de errores	Maneja eventos de error del puerto serie y captura excepciones en la inserción.

Tabla 1. Aspectos busConnector.js

4.3 Base de datos de series temporales

La base de datos TimescaleDB se ejecuta como un servicio Docker, inicializado con un usuario, contraseña y nombre de base de datos definidos previamente en variable de entorno.

La creación de la tabla *medidas_sensores* se realiza mediante el script run.sh que automatiza la creación de la tabla y su conversión a *hypertable*, optimizándola para manejar series temporales.

El almacenamiento en una tabla optimizada para manejo de series temporales permite:

- Escalabilidad en la ingesta de datos
- Consultas rápidas mediante índices temporales.
- La creación de vistas materializadas (*vistaMaterialazada.sh*) que calculan estadísticas temporales que se actualizan de forma continua.

En la siguiente tabla se muestran las vistas materializadas que se han creado para este proyecto:

Nombre Vista Materializada	Descripción	Métricas calculadas
temp_y_humed_porMin	Calcula las métricas para el intervalo de tiempo de un minuto	Temperatura media, Humedad media, Máxima Temperatura y Máxima Humedad
temp_y_humed_porHora	Calcula las métricas para el intervalo de tiempo de una hora	Temperatura media, Humedad media, Máxima Temperatura y Máxima Humedad
temp_y_humed_porDia	Calcula las métricas para el intervalo de tiempo de un dia	Temperatura media, Humedad media, Máxima Temperatura y Máxima Humedad

Tabla 2. Vistas Materializadas

4.4 Implementación de la capa caché con Redis

Redis se ejecuta también como un servicio Docker independiente. Al realizarse una consulta, el sistema primero busca en Redis. Si el dato existe se responde inmediatamente, pero en caso contrario, se consulta a la base de datos, se almacena en caché y se devuelve al cliente.

La capa caché realiza las siguientes funciones:

- Almacenar resultados de *endpoints*[15] como */concache-50*, */temp-25-cache*, etc.
- Se marcan los datos con un TTL de 60 segundos para que se refresquen periódicamente.

4.5 Visualización de datos mediante API REST

El servicio index.js implementa una API REST con las siguientes características:

- Conexión a PostgreSQL a través de Knex.
- Documentación interactiva con Swagger.

[15] Es un punto final específico, a menudo una API, que devuelve datos que han sido previamente almacenados en caché para mejorar el rendimiento y reducir la carga del servidor

- Endpoints GET diferenciados entre consultas con caché y sin caché.
- Consultas avanzadas apoyadas en vistas materializadas para estadísticas por minuto, hora y día.

A continuación, se muestran imágenes de la interfaz Swagger UI del proyecto:

Figura 4-3. Interfaz Swagger UI 1

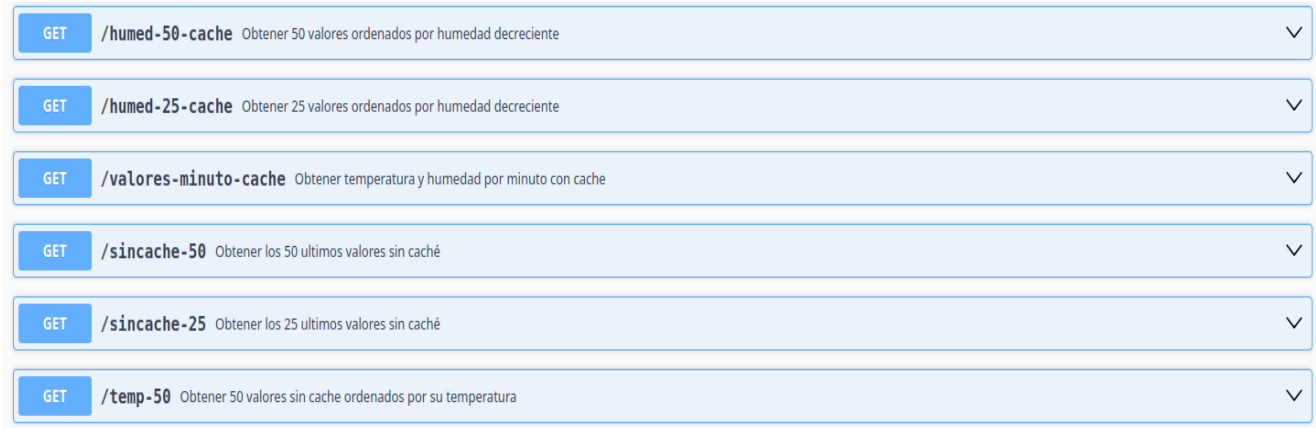

Figura 4-4. Interfaz Swagger UI 2

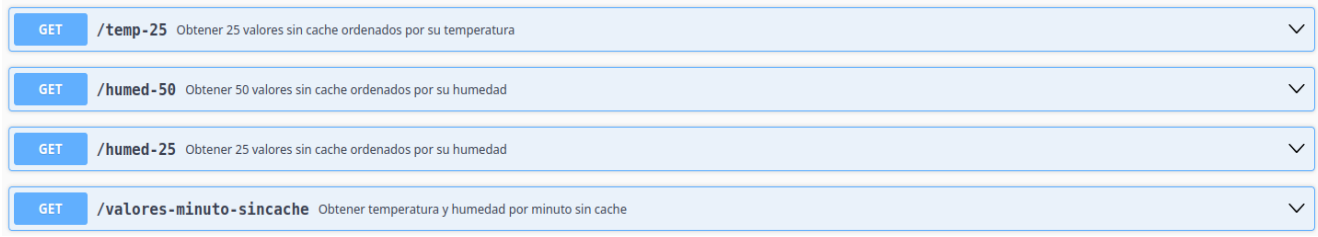

Figura 4-5. Interfaz Swagger UI 3

4.6 Contenerización y despliegue

El proyecto ha sido preparado y configurado para un despliegue reproducible mediante Docker:

- En el *Dockerfile* se define la imagen *Node.js*, instalando dependencias y configurando variables de entorno previamente definidas en *datosEntorno.env*
- En el *Compose.yaml* se definen los servicios principales que se muestran en la siguiente tabla:

Servicio	Imagen/base	Puertos	Rol

nodejs	Node18 + pm2	3000	API REST + busConnector
seriesTemporalesDB	timescale/timescaledb-ha:pg14-latest	5432	Base de datos de series temporales
redisCache	redis:latest	6379	Capa de caché

Tabla 3. Servicios en Docker-Compose

Se hace uso en el *Node.js* del administrador de procesos PM2 para:

- Ejecutar el servidor y el busConnector.js como procesos independientes definidos en el archivo *ecosystem.config.js.*
- Reiniciar los servicios automáticamente en caso de fallo.
- Limitar el consumo de memoria mediante parámetros.
- Gestionar logs de cada proceso de forma centralizada.

Por último, el despliegue se realiza ejecutando los siguientes comandos:

1. Construyo la imagen: *sudo docker-compose build.*
2. Levanto los servicios: *sudo docker-compose up -d.*

En este capítulo, se ha detallado el proceso de construcción del sistema, desde la captura de datos hasta su almacenamiento y consulta. La integración de Arduino, TimescaleDB, Redis y Docker asegura la coherencia de la propuesta. Una vez desplegado el sistema, se avanza hacia su validación experimental en los siguientes apartados.

5 PRUEBAS Y VALIDACIONES

El objetivo de este capítulo es demostrar el correcto funcionamiento del sistema y validar cada uno de los componentes.

Las pruebas se realizaron en un entorno controlado utilizando herramientas específicas para verificar la adquisición de datos, su almacenamiento, la correcta exposición mediante la API y la optimización de rendimiento.

5.1 Entorno de pruebas

Para las validaciones se utilizó el siguiente entorno:

Componente	Versión
Node.js	v18.19.1
PM2	Última estable
Redis	Última estable
PostgreSQL	Última estable
Docker	v24.0.7
Hardware	Placa Arduino + Sensor DHT11
Ubuntu	v24.04.1 LTS

Tabla 4. Entorno de pruebas

En la siguiente imagen se muestran los contenedores activos usados en el entorno de pruebas:

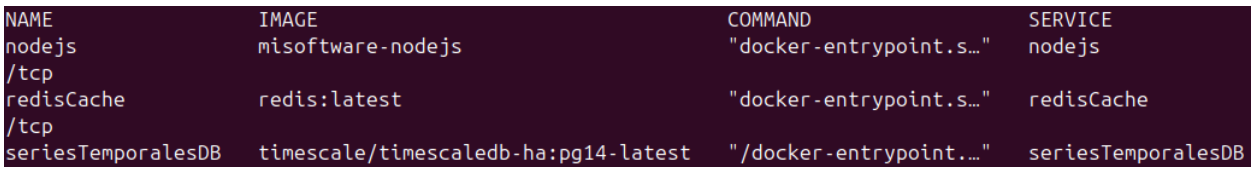

Figura 5-1. Contenedores Entorno Pruebas

5.2 Pruebas de adquisición de datos

Se verificó el funcionamiento del servicio *busConnector.js*:

- Con el sensor DHT11 conectado al puerto */dev/tty/USB0*, se recibieron tramas de datos de humedad y temperatura como se muestra en la Figura 5-2.

```
amando@amando-VirtualBox:~/Escritorio/TFG/MiSoftware$ sudo docker-compose logs -f nodejs
nodejs   | 2025-07-28T08:12:06: PM2 log: Launching in no daemon mode
nodejs   | 2025-07-28T08:12:07: PM2 log: App [api-server:0] starting in -cluster mode-
nodejs   | 2025-07-28T08:12:07: PM2 log: App [busConnector:1] starting in -cluster mode-
nodejs   | 2025-07-28T08:12:07: PM2 log: App [busConnector:1] online
nodejs   | 2025-07-28T08:12:07: PM2 log: App [api-server:0] online
nodejs   | @serialport/parser-readline: [class ReadlineParser extends DelimiterParser]
nodejs   | Puerto serie abierto correctamente (/dev/ttyUSB0)
nodejs   | Escuchando por el puerto interno de la red docker: 3000
nodejs   | Datos recibidos del puerto serie: 53
nodejs   | Humedad recibida: 53%
nodejs   | Datos recibidos del puerto serie: 30.4
nodejs   | Temperatura recibida: 30.4°C
nodejs   | Datos insertados correctamente en la base de datos
nodejs   | Datos recibidos del puerto serie: 53
nodejs   | Humedad recibida: 53%
nodejs   | Datos recibidos del puerto serie: 30.4
nodejs   | Temperatura recibida: 30.4°C
nodejs   | Datos insertados correctamente en la base de datos
nodejs   | Datos recibidos del puerto serie: 53
nodejs   | Humedad recibida: 53%
nodejs   | Datos recibidos del puerto serie: 30.3
nodejs   | Temperatura recibida: 30.3°C
nodejs   | Datos insertados correctamente en la base de datos
```

Figura 5-2. Comprobación de datos

- Se validó que los valores numéricos se procesan correctamente y se insertan en la tabla *medidas_sensor* como se muestra en la figura 5-3 donde le pido que me devuelva las ultimas 10 filas de la tabla.

```
         timestamp          | humedad | temperatura
----------------------------+---------+-------------
 2025-07-28 08:17:48.633+00 |      51 |        30.7
 2025-07-28 08:17:47.606+00 |      51 |        30.7
 2025-07-28 08:17:46.606+00 |      51 |        30.7
 2025-07-28 08:17:45.58+00  |      51 |        30.7
 2025-07-28 08:17:44.579+00 |      51 |        30.7
 2025-07-28 08:17:43.554+00 |      51 |        30.7
 2025-07-28 08:17:42.552+00 |      51 |        30.7
 2025-07-28 08:17:41.526+00 |      51 |        30.7
 2025-07-28 08:17:40.525+00 |      51 |        30.7
 2025-07-28 08:17:39.5+00   |      51 |        30.7
(10 rows)
```

Figura 5-3. Valores de la tabla medidas_sensor

5.3 Pruebas de almacenamiento y vistas materializadas

Una vez iniciada la base de datos TimescaleDB y ejecutados los scripts correspondientes, se comprobó:

- La creación de la tabla *medidas_sensor* como hypertable. En la figura 5-4 muestro que existe la tabla y su estructura y en la figura 5-5 compruebo que es una hypertable.

```
seriesTemporalesDB=# \d medidas_sensor
                    Table "public.medidas_sensor"
   Column    |           Type           | Collation | Nullable | Default
-------------+--------------------------+-----------+----------+---------
 timestamp   | timestamp with time zone |           | not null | now()
 humedad     | real                     |           |          |
 temperatura | real                     |           |          |
```

Figura 5-4. Estructura tabla medidas_sensor

```
seriesTemporalesDB=# SELECT * FROM timescaledb_information.hypertables;
 hypertable_schema | hypertable_name |  owner   | num_dimensions | num_chunks | compression_enabled | is_distributed |
-------------------+-----------------+----------+----------------+------------+---------------------+----------------+
 public            | medidas_sensor  | postgres |              1 |          1 | f                   | f              |
(1 row)
```

Figura 5-5. Hypertable medidas_sensor

- La correcta creación de las vistas materializadas: *temp_y_humed_porMin*, *temp_y_humed_porHora* y *temp_y_humed_porDia* explicadas en el apartado 4.3 de este documento. Se muestran en la figura 5-6.

```
seriesTemporalesDB=# \dv
                List of relations
 Schema |          Name          | Type |  Owner
--------+------------------------+------+----------
 public | temp_y_humed_pordia    | view | postgres
 public | temp_y_humed_porhora   | view | postgres
 public | temp_y_humed_pormin    | view | postgres
(3 rows)
```

Figura 5-6. Vistas materializadas

- Consultas SQL de prueba para validar los resultados agregados por minuto de una de las vistas materializadas, en este caso la de *temp_y_humed_porMin*. Se muestra en la figura 5-7.

```
seriesTemporalesDB=# SELECT * FROM temp_y_humed_porMin ORDER BY time DESC LIMIT 5;
          time          |     avg_temp      |     avg_humed      | max_temp | max_humed
------------------------+-------------------+--------------------+----------+-----------
 2025-07-28 08:35:00+00 | 30.969431716509476 | 50.957951401778786 |       31 |        51
 2025-07-28 08:34:00+00 | 30.907554730062902 | 50.048913855341524 |       31 |        51
 2025-07-28 08:33:00+00 | 30.907554730062902 | 50.048913855341524 |       31 |        51
 2025-07-28 08:32:00+00 | 30.907554730062902 | 50.048913855341524 |       31 |        51
 2025-07-28 08:31:00+00 | 30.907554730062902 | 50.048913855341524 |       31 |        51
(5 rows)
```

Figura 5-7. Consulta datos a vista materializada

- Consultas SQL de prueba para validar que la vista se actualiza, en este caso, inserto dos valores muy altos de temperatura y humedad (99) para luego consultar en la vista si se han actualizado como se ve en la figura 5-8.

```
seriesTemporalesDB=# INSERT INTO medidas_sensor (humedad, temperatura) VALUES (99, 99);
INSERT 0 1
seriesTemporalesDB=# SELECT * FROM temp_y_humed_porMin ORDER BY time DESC LIMIT 5;
         time          |      avg_temp      |     avg_humed      | max_temp | max_humed
-----------------------+--------------------+--------------------+----------+-----------
 2025-07-28 08:45:00+00 | 30.969431716509476 | 50.957951401778786 |     31.1 |        51
 2025-07-28 08:44:00+00 | 31.093557570958556 | 50.957951401778786 |       99 |        99
 2025-07-28 08:43:00+00 | 31.093557570958556 | 50.048913855341524 |     31.2 |        51
 2025-07-28 08:42:00+00 | 31.093557570958556 | 50.048913855341524 |     31.2 |        55
 2025-07-28 08:41:00+00 | 31.093557570958556 | 50.957951401778786 |     31.1 |        51
(5 rows)
```

Figura 5-8. Actualización de Vistas Materializadas.

5.4 Pruebas de la API REST

Para comprobar que la API REST funciona, las rutas expuestas en *index.js* fueron validadas utilizando Swagger UI. Se probaron tanto endpoints con caché (/concache-25), /temp-25-concache, etc.) como sin caché (/sincache-25, /temp-25, etc.).

A continuación, se muestran imágenes de los endpoints con caché:

- En */concache-25* obtenemos los últimos 25 valores de temperatura y humedad:

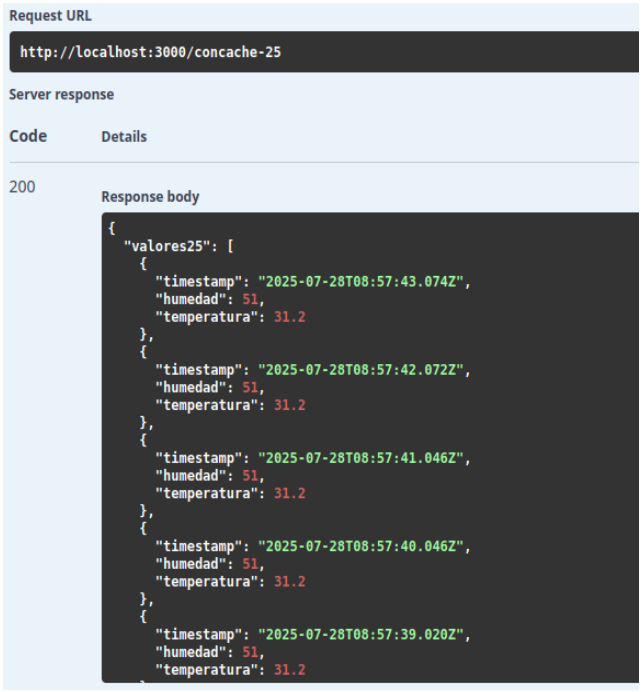

Figura 5-9. Comprobación de /concache-25

- En */temp-25-cache* obtenemos 25 valores ordenados por temperatura decreciente:

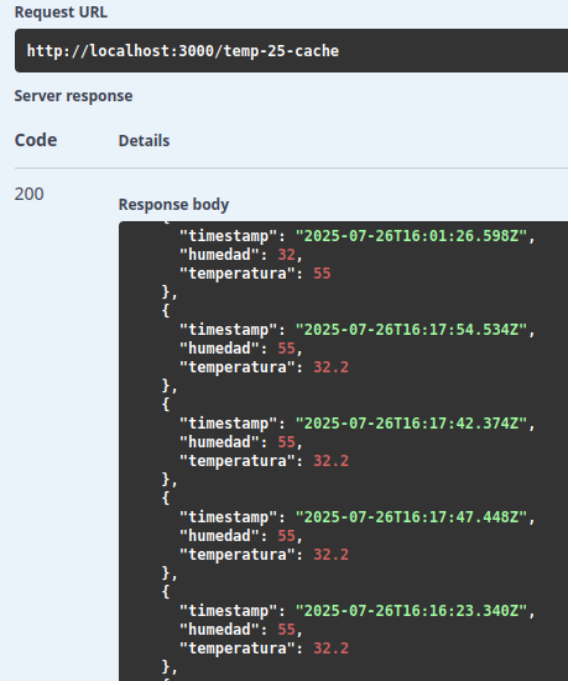

Figura 5-10. Comprobación de /temp-25-cache

- En */humed-25-cache* obtenemos 25 valores ordenados por humedad decreciente:

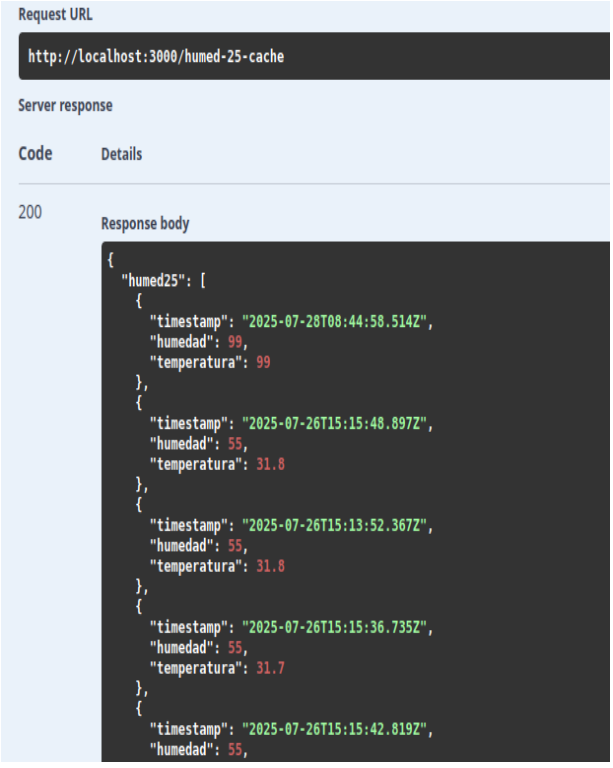

Figura 5-11. Comprobación de /humed-25-cache

- En */valores-minuto-cache* se obtiene la media y valor máximo por minuto de la temperatura y la humedad.

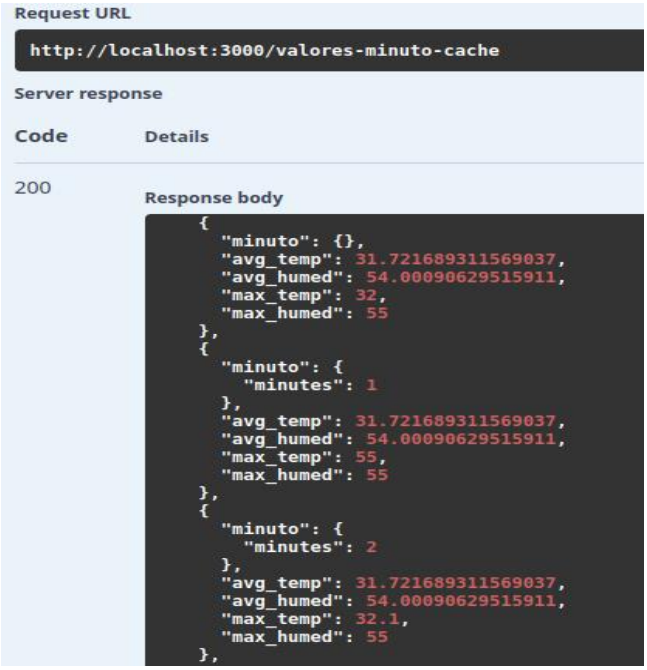

Figura 5-12. Comprobación de /valores-minuto-cache

Por otro lado, se muestran imágenes de los endpoints sin caché:

- En */sincache-25* obtenemos los últimos 25 valores de temperatura y humedad:

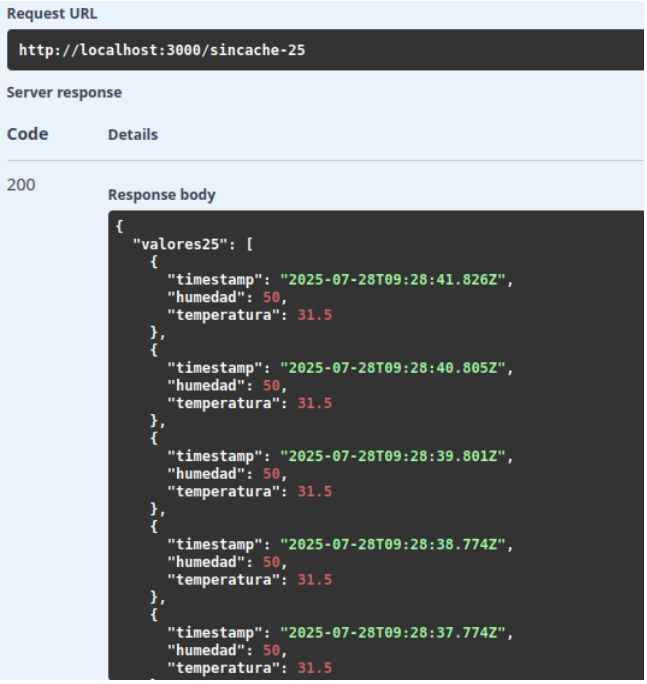

Figura 5-13. Comprobación de /sincache-25

- En */temp-25* obtenemos 25 valores ordenados por temperatura decreciente:

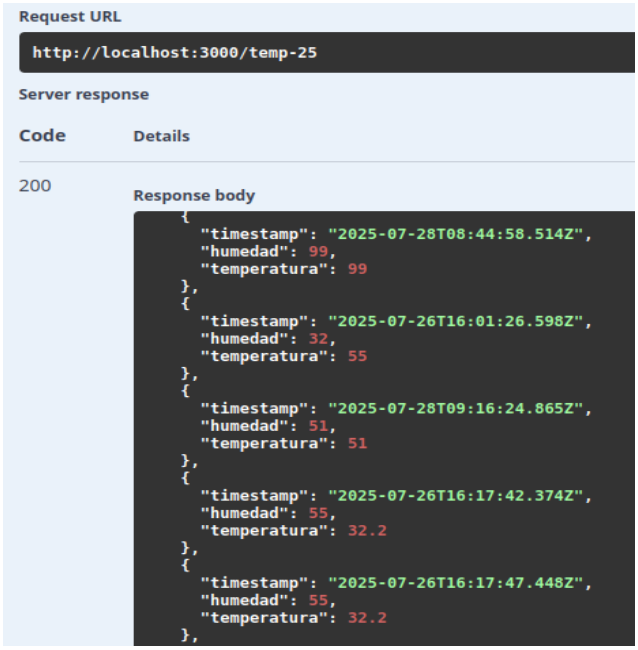

Figura 5-14. Comprobación de /temp-25

- En */humed-25* obtenemos 25 valores ordenados por humedad decreciente:

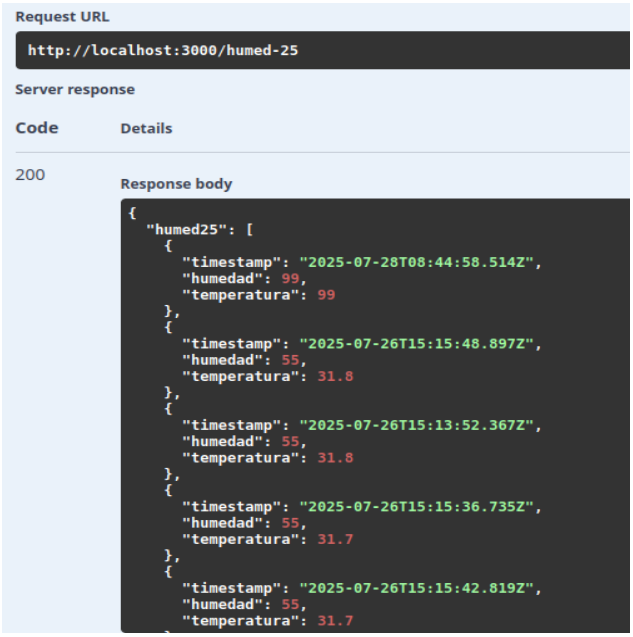

Figura 5-15. Comprobación de /humed-25

- En */valores-minuto-sincache* se obtiene la media y valor máximo por minuto de la temperatura y la humedad:

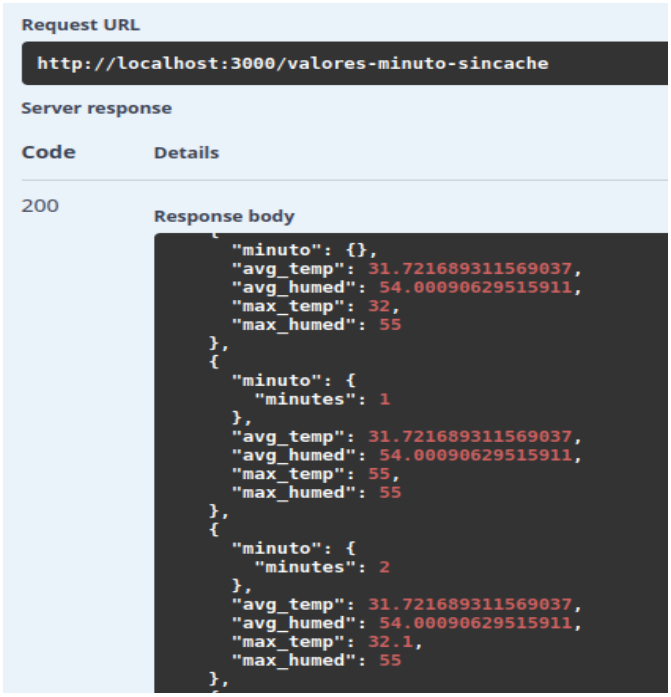

Figura 5-16. Comprobación de /valores-minuto-sincache

5.5 Pruebas de la capa de caché

Para validar la funcionalidad y rendimiento de la capa de caché basada en Redis, se realizaron varias pruebas prácticas que demuestran su correcta integración:

- **Verificación del almacenamiento en caché:** Desde una terminal en el sistema host, se ejecuta una petición http a un endpoint con caché. Posteriormente, entramos al cliente de Redis y se comprueba si la clave ha sido almacenada como se muestra en la figura 5-17.

```
amando@amando-VirtualBox:~/Escritorio/TFG/MiSoftware$ sudo docker exec -it redisCache redis-cli
127.0.0.1:6379> keys *
1) "25-valores"
```

Figura 5-17. Almacenamiento Key en Caché

- **Verificación de la expiración y regeneración de la caché:**

  1) Desde el terminal host accedemos al cliente Redis y verificamos que no hay ninguna clave porque no hemos hecho la consulta aun como se muestra en la figura 5-18.

```
amando@amando-VirtualBox:~/Escritorio/TFG/MiSoftware$ sudo docker exec -it redisCache redis-cli
127.0.0.1:6379> keys *
(empty array)
127.0.0.1:6379>
```

Figura 5-18. Prueba caché vacía

2) Realizamos la petición http y comprobamos que la clave se ha almacenado y que su TTL es de 60 segundos, aunque en la figura 5-19 será menor porque transcurre el tiempo.

```
127.0.0.1:6379> keys *
1) "25-valores"
127.0.0.1:6379> ttl 25-valores
(integer) 51
127.0.0.1:6379>
```

Figura 5-19. Comprobación TTL caché

3) Una vez pasados los 51 segundos de TTL que se ven en la imagen anterior, la caché debe haber eliminado la clave como se muestra en la figura 5-20 demostrando así, su capacidad de expiración y regeneración.

```
127.0.0.1:6379> ttl 25-valores
(integer) 51
127.0.0.1:6379> keys *
(empty array)
127.0.0.1:6379>
```

Figura 5-20. Comprobación expiración caché

- **Verificación del rendimiento con caché:** Para validar la mejora en tiempos de respuesta que ofrece el sistema con caché, se utilizó la herramienta autocannon. Se lanzaron 50 conexiones simultaneas durante 10 segundos contra los endpoints */sincache-50* y */concache-50*.

  El resultado del endpoint sin caché es el que se muestra en la siguiente figura 5-21:

c

```
amando@amando-VirtualBox:~/Escritorio/TFG/MiSoftware$ npx autocannon -c 50 -d 10 http://localhost:3000/sincache-50
Running 10s test @ http://localhost:3000/sincache-50
50 connections


| Stat | 2.5% | 50% | 97.5% | 99% | Avg | Stdev | Max |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Latency | 27 ms | 30 ms | 48 ms | 51 ms | 32.42 ms | 7.34 ms | 150 ms |

| Stat | 1% | 2.5% | 50% | 97.5% | Avg | Stdev | Min |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Req/Sec | 1090 | 1090 | 1563 | 1658 | 1518,3 | 158 | 1090 |
| Bytes/Sec | 4.26 MB | 4.26 MB | 6.1 MB | 6.47 MB | 5.93 MB | 617 kB | 4.25 MB |

Req/Bytes counts sampled once per second.
# of samples: 10

15k requests in 10.04s, 59.3 MB read
```

Figura 5-21. Estadísticas endpoint sin caché

El resultado del endpoint con caché es el que se muestra en la siguiente figura 5-22:

```
amando@amando-VirtualBox:~/Escritorio/TFG/MiSoftware$ npx autocannon -c 50 -d 10 http://localhost:3000/concache-50
Running 10s test @ http://localhost:3000/concache-50
50 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	10 ms	11 ms	19 ms	21 ms	12.26 ms	3.27 ms	81 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	3389	3389	3899	4311	3920,5	219,93	3389
Bytes/Sec	13.2 MB	13.2 MB	15.2 MB	16.8 MB	15.3 MB	858 kB	13.2 MB

```
Req/Bytes counts sampled once per second.
# of samples: 10

39k requests in 10.04s, 153 MB read
```

Figura 5-22. Estadísticas endpoint con caché

Por último, en la siguiente tabla se muestra una comparación de los resultados:

Métrica	Sin caché	Con caché
Latencia media (Latency)	32.42 ms	12.26 ms
Peticiones/Segundo (Req/Sec)	1.518 pet/seg	3.920 pet/seg
Throughput medio (Avg)	5.93 MB/s	15.3 MB/s
Máxima latencia (Max)	150 ms	81 ms
Peticiones totales	15.000 aprox	39.000 aprox

Tabla 5. Tabla comparativa rendimiento con caché

Los datos confirman que la capa de caché reduce la latencia en más de un 60% y que además casi triplica la capacidad de servir peticiones por segundo, lo que supone una optimización notable del sistema.

Las pruebas realizadas han confirmado el correcto funcionamiento del sistema y su capacidad para gestionar datos en tiempo real. Los resultados evidencian mejoras significativas en eficiencia gracias al uso de Redis y Vistas Materializadas. Estos hallazgos validan la solución planteada y dan pie a la reflexión global del trabajo en el siguiente apartado de conclusiones.

6 CONCLUSIONES Y LÍNEAS FUTURAS

6.1 Cumplimiento de objetivos

El presente Trabajo de Fin de Grado ha cumplido de forma satisfactoria los objetivos establecidos al inicio. Se ha desarrollado un sistema funcional capaz de adquirir datos de temperatura y humedad desde una placa física Arduino, almacenarlos en una base de datos especializada en series temporales de forma estructurada (TimescaleDB) y que puedan ser consultados por usuarios mediante una API REST documentada con Swagger. Esta arquitectura elegida ha permitido que haya una separación clara de funciones y responsabilidades entre la capa de adquisición, la de almacenamiento y la de visualización de datos.

Por otra parte, se integró un sistema de almacenamiento en caché con Redis para reducir la carga sobre la base de datos y mejorar el tiempo de respuesta del sistema. Además, se ha utilizado Docker para facilitar la gestión de servicios y permitir que la configuración sea reproducible, portátil y aislada del entorno host. La incorporación del gestor de procesos PM2 ha permitido lanzar y mantener la API en ejecución de manera estable dentro del contenedor. Este enfoque en contenedores ha hecho más sencillo el desarrollo y el despliegue del sistema, sentando una base sólida para su escalabilidad o futuras mejoras.

Este proyecto no solo ha alcanzado los objetivos funcionales propuestos, sino que también ha servido como una primera aproximación a la construcción de sistemas basados en microservicios y tecnologías actuales. Aunque se trata de una solución enfocada al ámbito académico, su diseño facilita que pueda evolucionar hacia aplicaciones más complejas.

6.2 Valoración del trabajo realizado

La realización de este proyecto ha supuesto una gran experiencia técnica y personal. Me ha dado la oportunidad para trabajar con herramientas y tecnologías que no había utilizado antes y que son muy utilizadas en el ámbito profesional como Node.js, Docker, Redis, TimescaleDB y PM2. Durante el proceso he tenido que estudiar, comprender y aplicar conceptos clave como la persistencia de datos en contenedores, la comunicación entre servicios mediante redes virtuales o la mejora del rendimiento en consultas a base de datos mediante técnicas de cacheo.

Durante el desarrollo del sistema también se ha prestado atención a aspectos como la documentación clara del sistema y la posibilidad de automatizar su despliegue. Herramientas como Swagger han permitido generar una interfaz simple y accesible para entender y probar la API, o como autocannon, que ha sido clave para analizar el comportamiento del sistema bajo carga, permitiendo detectar mejoras en el rendimiento al usar caché.

Por último, desde mi punto de vista este trabajo me ha servido para mejorar en metodologías de resolución de problemas, para enfrentarme a errores de integración y para coger confianza a la hora de empezar proyectos tecnológicos más complicados desde cero.

6.3 Líneas de mejora y trabajos futuros

Aunque el sistema cumple con los objetivos planteados, existen varias mejoras que permitirían ampliar el proyecto:

- **Visualización de datos:** Sería útil desarrollar un panel o visor web que permitiera consultar gráficamente los valore adquiridos, tendencias y alertas, facilitando su uso a usuarios no técnicos.
- **Almacenamiento persistente en Redis:** En este proyecto, Redis actúa como caché en memoria volátil. Se podría implementar su persistencia en disco para conservar los datos en caso de reinicio del

contenedor o del sistema.

- **Seguridad y autenticación:** Incorporar mecanismos de autenticación en los endpoints y validación de roles de usuario para aumentar la seguridad en caso de que este desplegado en un entorno accesible por terceros.
- **Escalabilidad en la nube:** El sistema puede adaptarse para ejecutarse en plataformas cloud, pudiendo hacer uso de balanceadores de carga y bases de datos distribuidas.
- **Alertas y monitorización avanzada:** Herramientas como Grafana podrían integrarse para visualizar métricas y generar alertas.

Estas mejoras harían evolucionar al sistema hacia una solución más completa, mantenible y orientada a entornos industriales reales.

ANEXO A: CÓDIGO FUENTE DEL SISTEMA

En el siguiente anexo se detalla el código fuente de cada módulo del sistema, dividido y ordenado de la siguiente manera:

1. Archivos para Docker
2. Archivo busConnector.js
3. Archivo index.js
4. Archivo swagger.js
5. Conjunto de scripts para iniciar o parar el sistema

1.1. Dockerfile

```
FROM node:18

# Usamos la misma carpeta que en docker-compose
WORKDIR /usr/src/app

# Copiamos package.json y package-lock.json primero
COPY package*.json ./

# Instalamos dependencias del package.json
RUN npm install

# Instalamos pm2 globalmente
RUN npm install -g pm2

# Ahora copiamos el resto del proyecto
COPY . .

# Variables de entorno pasadas como build args
ARG SQL_USER
ARG SQL_PASSWORD
ARG SQL_PORT
ARG SQL_IP
ARG SQL_DATABASE
ARG REDIS_IP
ARG REDIS_PORT

ENV SQL_USER=${SQL_USER} \
    SQL_PASSWORD=${SQL_PASSWORD} \
    SQL_PORT=${SQL_PORT} \
    SQL_IP=${SQL_IP} \
    SQL_DATABASE=${SQL_DATABASE} \
    REDIS_IP=${REDIS_IP} \
    REDIS_PORT=${REDIS_PORT}

# Arranca usando pm2-runtime
CMD ["pm2-runtime", "start", "ecosystem.config.js"]
```

1.2. compose.yaml

```yaml
version: '3.8' # Version del esquema de docker-compose

services:
  nodejs:
    container_name: nodejs
    restart: always
    depends_on:  ##El contenedor de nodejs depende de que estos 2 contenedores e
      - seriesTemporalesDB
      - redisCache

    build:
      context: .  ##El contexto de construccion es el directorio actual por eso
      args:
        SQL_USER: postgres
        SQL_PASSWORD: postgres
        SQL_IP: seriesTemporalesDB  ##Permite a nodejs conectarse a la bbdd y al
        SQL_PORT: 5432
        SQL_DATABASE: seriesTemporalesDB
        REDIS_IP: redisCache
        REDIS_PORT: 6379
    ports:
      - "3000:3000" #Via web accederemos como http://localhost:3333/ o por coman
    devices:
      - "/dev/ttyUSB0:/dev/ttyUSB0"  # Mapeo del puerto serie donde esta la plac
    privileged: true
    working_dir: /usr/src/app
    volumes:
      - .:/usr/src/app #Monta codigo local en el contenedor
      - /usr/src/app/node_modules #Uso volumen anonimo solo para node_volumes y
```

```yaml
  seriesTemporalesDB:
    container_name: seriesTemporalesDB
    image: timescale/timescaledb-ha:pg14-latest
    restart: always
    ports:
      - "5432:5432"
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: seriesTemporalesDB
    volumes:
      - pgdata:/home/postgres/pgdata/data # Persistencia de datos (monta carpeta local ./data)


  redisCache:
    container_name: redisCache
    image: redis:latest
    restart: always
    ports:
      - "6379:6379"
    volumes:
      - ./redis-data:/data # Persistencia de datos de Redis

volumes:
  pgdata:
```

1.3. ecosystem.config.js

```
module.exports = {
  apps: [
    {
      name: "api-server",
      script: "index.js",
      instances: 1,
      autorestart: true,
      watch: false,
      max_memory_restart: "300M"
    },
    {
      name: "busConnector",
      script: "busConnector.js",
      instances: 1,
      autorestart: true,
      watch: false,
      max_memory_restart: "300M"
    }
  ]
};
```

1.4. datosEntorno.env

```
# Datos de la base de datos PostgreSQL
SQL_USER=postgres
SQL_PASSWORD=postgres
SQL_PORT=5432
SQL_IP=127.0.0.1
SQL_DATABASE=seriesTemporalesDB

# Datos de Redis
REDIS_IP=127.0.0.1
REDIS_PORT=6379
```

2. busConnector.js

```
//ARCHIVO busConnector.js

//Autor: Amando Antoñano Puerta

//Obtiene los datos de temperatura y humedad del sensor DHT11 por el serial port USB, los trata y luego los
//Los datos vienen entramas de 40 bytes. 8 humed num entero, 8 humed decimal, 8 temp entero, 8 temp decimal

// busConnector.js
const { SerialPort } = require('serialport');
const { ReadlineParser } = require('@serialport/parser-readline');
console.log('@serialport/parser-readline:', ReadlineParser);
const knex = require('knex')({
  client: 'pg',
  connection: {
    host: process.env.SQL_IP,
    port: process.env.SQL_PORT,
    user: process.env.SQL_USER,
    password: process.env.SQL_PASSWORD,
    database: process.env.SQL_DATABASE
  }
});

// Abrir puerto serie
const port = new SerialPort({ path: '/dev/ttyUSB0', baudRate: 9600 });
const parser = port.pipe(new ReadlineParser({ delimiter: '\n' }));
```

```
let humedad = null;

port.on('open', () => {
  console.log('Puerto serie abierto correctamente (/dev/ttyUSB0)');
});

port.on('error', (err) => {
  console.error('Error en puerto serie:', err.message);
});

// Escuchar datos del parser
parser.on('data', async (line) => {
  const value = parseFloat(line.trim());

  if (isNaN(value)) {
    console.warn(`Dato no numérico recibido: "${line}"`);
    return;
  }

  console.log(`Datos recibidos del puerto serie: ${value}`);
```

```
  if (humedad === null) {
    humedad = value;
    console.log(`Humedad recibida: ${humedad}%`);
  } else {
    const temp = value;
    console.log(`Temperatura recibida: ${temp}ºC`);

    // Guardar en base de datos
    try {
      await knex('medidas_sensor').insert({
        timestamp: new Date().toISOString(),
        temperatura: temp,
        humedad: humedad
      });
      console.log('Datos insertados correctamente en la base de datos');
    } catch (err) {
      console.error('Error insertando en base de datos:', err.message);
    }

    // Reset humedad para la próxima pareja
    humedad = null;
  }
});
```

3. index.js

```
//CONTIENE LAS CONEX A BBDD Y REDIS. LAS PETICIONES GET Y POST A LA BBD

const knex = require ('knex');  //Query builder que me permite interact
const express = require('express'); //Framework para crear server http
const {createClient} = require('redis'); //Creo cliente redis con la fu

const swaggerUi = require ('swagger-ui-express'); //Para visualiz y pro
const swaggerSpec = require('./swagger'); //IMporto config de swagger d



const app = express();

app.use(express.json());
app.use('/visor-swag', swaggerUi.serve, swaggerUi.setup(swaggerSpec));
```

```
//Creo instancia para la conex a bbdd
const bbdd = knex (
{
    client: 'pg',  //Indica que el cliente de la bbdd es para postgresSQL
    connection:
    {
        user: process.env.SQL_USER,
        password: process.env.SQL_PASSWORD,
        host: process.env.SQL_IP,
        port: process.env.SQL_PORT,
        database: process.env.SQL_DATABASE
    }
});


//Creo instancia redis que me permitira conectarme al server redis
const redis = createClient
({
    socket: {

        port: process.env.REDIS_PORT,
        host: process.env.REDIS_IP,
    }
});
```

```
/**
 * @swagger
 * /concache-50:
 *  get:
 *      summary: Obtener los 50 ultimos valores con caché
 *      description: Petición get para obtener mediante cache los 50 ultimos valores. Si no
 *      responses:
 *          200:
 *            description: Valores obtenidos correctamente
 */


app.get('/concache-50', async(request, response) => {

    const valorCache = await redis.get('50-valores');

    if(valorCache) //Si la anterior linea devolvio algo, responderé.
    {
        return response.send({valores50: JSON.parse(valorCache)});
    }

    //Consulta a bbdd. A la tabla medidas_sensor, que venga ordenada de manera descendiente
    const valores50 = await bbdd('medidas_sensor').orderBy('timestamp', 'desc').limit(50);

    await redis.setEx('50-valores', 60, JSON.stringify(valores50));

    response.send({valores50});
});
```

```
/**
 * @swagger
 * /concache-25:
 *  get:
 *      summary: Obtener los 25 ultimos valores con caché
 *      description: Petición get para obtener mediante cache los 25 ultimos valores. Si no
 *      responses:
 *          200:
 *            description: Valores obtenidos correctamente
 */


app.get('/concache-25', async(request, response) => {

    const valorCache = await redis.get('25-valores');

    if(valorCache) //Si la anterior linea devolvio algo, responderé.
    {
        return response.send({valores25: JSON.parse(valorCache)});
    }

    //Consulta a bbdd. A la tabla medidas_sensor, que venga ordenada de manera descendiente
    const valores25 = await bbdd('medidas_sensor').orderBy('timestamp', 'desc').limit(25);

    await redis.setEx('25-valores', 60, JSON.stringify(valores25));

    response.send({valores25});
});
```

```
/**
 * @swagger
 * /temp-50-cache:
 *  get:
 *      summary: Obtener 50 valores ordenados por temperatura decreciente
 *      description: Petición get para obtener mediante cache 50 valores de tempratura orde
 *      responses:
 *          200:
 *            description: Valores de temperatura obtenidos correctamente
 */


app.get('/temp-50-cache', async(request, response) => {

    const valorCache = await redis.get('50-temp');

    if(valorCache) //Si la anterior linea devolvio algo, responderé.
    {
        return response.send({temp50: JSON.parse(valorCache)});
    }

    //Consulta a bbdd. A la tabla medidas_sensor, que venga ordenada de manera descendiente
    const temp50 = await bbdd('medidas_sensor').orderBy('temperatura', 'desc').limit(50);

    await redis.setEx('50-temp', 60, JSON.stringify(temp50));

    response.send({temp50});
});
```

```
/**
 * @swagger
 * /temp-25-cache:
 *  get:
 *      summary: Obtener 25 valores ordenados por temperatura decreciente
 *      description: Petición get para obtener mediante cache 25 valores de tempratura orde
 *      responses:
 *          200:
 *            description: Valores de temperatura obtenidos correctamente
 */


app.get('/temp-25-cache', async(request, response) => {

    const valorCache = await redis.get('25-temp');

    if(valorCache) //Si la anterior linea devolvio algo, responderé.
    {
        return response.send({temp25: JSON.parse(valorCache)});
    }

    //Consulta a bbdd. A la tabla medidas_sensor, que venga ordenada de manera descendiente
    const temp25 = await bbdd('medidas_sensor').orderBy('temperatura', 'desc').limit(25);

    await redis.setEx('25-temp', 60, JSON.stringify(temp25));

    response.send({temp25});
});
```

```
/**
 * @swagger
 * /humed-50-cache:
 *  get:
 *      summary: Obtener 50 valores ordenados por humedad decreciente
 *      description: Petición get para obtener mediante cache 50 valores de humedad orde
 *      responses:
 *          200:
 *            description: Valores de humedad obtenidos correctamente
 */


app.get('/humed-50-cache', async(request, response) => {

    const valorCache = await redis.get('50-humed');

    if(valorCache) //Si la anterior linea devolvio algo, responderé.
    {
        return response.send({humed50: JSON.parse(valorCache)});
    }

    //Consulta a bbdd. A la tabla medidas_sensor, que venga ordenada de manera descendie
    const humed50 = await bbdd('medidas_sensor').orderBy('humedad', 'desc').limit(50);

    await redis.setEx('50-humed', 60, JSON.stringify(humed50));

    response.send({humed50});
});
```

```
/**
 * @swagger
 * /humed-25-cache:
 *  get:
 *      summary: Obtener 25 valores ordenados por humedad decreciente
 *      description: Petición get para obtener mediante cache 25 valores de humedad orde
 *      responses:
 *          200:
 *            description: Valores de humedad obtenidos correctamente
 */


app.get('/humed-25-cache', async(request, response) => {

    const valorCache = await redis.get('25-humed');

    if(valorCache) //Si la anterior linea devolvio algo, responderé.
    {
        return response.send({humed25: JSON.parse(valorCache)});
    }

    //Consulta a bbdd. A la tabla medidas_sensor, que venga ordenada de manera descendie
    const humed25 = await bbdd('medidas_sensor').orderBy('humedad', 'desc').limit(25);

    await redis.setEx('25-humed', 60, JSON.stringify(humed25));

    response.send({humed25});
});
```

```
/**
 * @swagger
 * /valores-minuto-cache:
 *  get:
 *      summary: Obtener temperatura y humedad por minuto con cache
 *      description: Calcula la media y el valor maximo por minuto de la temperatura y la humedad con cache.
 *      responses:
 *          200:
 *              description: Media y máxima obtenidos correctamente
 *          500:
 *              description: Error obteniendo o calculando los valores
 *
 */


app.get('/valores-minuto-cache', async (request, response) =>
{
    const valorCache = await redis.get('valor-min');

    if(valorCache) //Si la anterior linea devolvio algo, responderé.
    {
        return response.send({medymax: JSON.parse(valorCache)});
    }


    /** EN datos_minuto guardo el tiempo, avg_temp y avg_humed de hace un año y lo ordeno por tiempo.
     * Luego agrupo por minuto para calcular percentiles y máximos de esos valores ya agregados.
     */
```

```
  const medymax = await bbdd.raw(`
    WITH por_minuto AS (
      SELECT time, avg_temp AS temp, avg_humed AS humed, max_temp, max_humed
      FROM temp_y_humed_porMin
      WHERE time AT TIME ZONE 'Europe/Berlin' > date_trunc('month', now()) - interval '1 year'
      ORDER BY 1
    ),
    data_minuto AS (
      SELECT
        extract(MINUTE FROM time) * interval '1 minute' AS minuto,
        temp,
        humed,
        max_temp,
        max_humed
      FROM por_minuto
    )
    SELECT
      minuto,
      approx_percentile(0.50, percentile_agg(temp)) AS avg_temp,
      approx_percentile(0.50, percentile_agg(humed)) AS avg_humed,
      max(max_temp) AS max_temp,
      max(max_humed) AS max_humed
```

```
      FROM data_minuto
      GROUP BY 1
      ORDER BY 1;
    `);

    await redis.setEx('valor-min', 60, JSON.stringify(medymax));

    response.status(200).json(medymax);
});
```

```
/**
 * @swagger
 * /sincache-50:
 *  get:
 *      summary: Obtener los 50 ultimos valores sin caché
 *      description: Petición get para obtener mediante consuta a bbdd los 50 ultimos valore
 *      responses:
 *          200:
 *            description: Valores obtenidos correctamente.
 */


app.get('/sincache-50', async(request, response) => {

    //Consulta a bbdd. A la tabla medidas_sensor, que venga ordenada de manera descendiente
    const valores50 = await bbdd('medidas_sensor').orderBy('timestamp', 'desc').limit(50);

    response.send({valores50});

});
```

```
/**
 * @swagger
 * /sincache-25:
 *  get:
 *      summary: Obtener los 25 ultimos valores sin caché
 *      description: Petición get para obtener mediante consuta a bbdd los 25 ultimos valore
 *      responses:
 *          200:
 *            description: Valores obtenidos correctamente.
 */


app.get('/sincache-25', async(request, response) => {

    //Consulta a bbdd. A la tabla medidas_sensor, que venga ordenada de manera descendiente
    const valores25 = await bbdd('medidas_sensor').orderBy('timestamp', 'desc').limit(25);

    response.send({valores25});

});
```

```
/**
 * @swagger
 * /temp-50:
 *  get:
 *      summary: Obtener 50 valores sin cache ordenados por su temperatura
 *      description: Petición get para obtener mediante consulta a bbdd 50 valores ordenad
 *      responses:
 *          200:
 *            description: Valores de temperatura obtenidos correctamente.
 */


app.get('/temp-50', async(request, response) => {

    //Consulta a bbdd. A la tabla medidas_sensor, que venga ordenada de manera descendiente
    const temp50 = await bbdd('medidas_sensor').orderBy('temperatura', 'desc').limit(50);

    response.send({temp50});

});
```

```
/**
 * @swagger
 * /temp-25:
 *  get:
 *      summary: Obtener 25 valores sin cache ordenados por su temperatura
 *      description: Petición get para obtener mediante consulta a bbdd 25 valores ordenad
 *      responses:
 *          200:
 *            description: Valores de temperatura obtenidos correctamente.
 */


app.get('/temp-25', async(request, response) => {

    //Consulta a bbdd. A la tabla medidas_sensor, que venga ordenada de manera descendient
    const temp25 = await bbdd('medidas_sensor').orderBy('temperatura', 'desc').limit(25);

    response.send({temp25});

});
```

```
/**
 * @swagger
 * /humed-50:
 *  get:
 *      summary: Obtener 50 valores sin cache ordenados por su humedad
 *      description: Petición get para obtener mediante consulta a bbdd 50 valores order
 *      responses:
 *          200:
 *            description: Valores de humedad obtenidos correctamente.
 */


app.get('/humed-50', async(request, response) => {

    //Consulta a bbdd. A la tabla medidas_sensor, que venga ordenada de manera descendie
    const humed50 = await bbdd('medidas_sensor').orderBy('humedad', 'desc').limit(50);

    response.send({humed50});

});
```

```
/**
 * @swagger
 * /humed-25:
 *  get:
 *      summary: Obtener 25 valores sin cache ordenados por su humedad
 *      description: Petición get para obtener mediante consulta a bbdd 25 valores order
 *      responses:
 *          200:
 *            description: Valores de humedad obtenidos correctamente.
 */


app.get('/humed-25', async(request, response) => {

    //Consulta a bbdd. A la tabla medidas_sensor, que venga ordenada de manera descendie
    const humed25 = await bbdd('medidas_sensor').orderBy('humedad', 'desc').limit(25);

    response.send({humed25});

});
```

```
/**
 * @swagger
 * /valores-minuto-sincache:
 *  get:
 *      summary: Obtener temperatura y humedad por minuto sin cache
 *      description: Calcula la media y el valor maximo por minuto de la temperatura y la humedad sin cache.
 *      responses:
 *          200:
 *              description: Media y máxima obtenidos correctamente
 *          500:
 *              description: Error obteniendo o calculando los valores
 *
 */


app.get('/valores-minuto-sincache', async (request, response) =>
{

/** EN por_minuto guardo el tiempo, temp y humed de hace un mes y lo ordeno por tiempo. Luego en data_minuto s
* Por ultimo calculo para cada minuto los valores medios de temperatura/humedad y sus valores maximos, ademas
```

```
const medymax = await bbdd.raw(`
WITH por_minuto AS (
  SELECT time, avg_temp, avg_humed, max_temp, max_humed
  FROM temp_y_humed_porMin
  WHERE "time" AT TIME ZONE 'Europe/Berlin' > date_trunc('month', time) - interval '1 year'
  ORDER BY 1
),
data_minuto AS (
  SELECT
    extract(MINUTE FROM time) * interval '1 minute' AS minuto,
    avg_temp,
    avg_humed,
    max_temp,
    max_humed
  FROM por_minuto
)
SELECT
  minuto,
  approx_percentile(0.50, percentile_agg(avg_temp)) AS avg_temp,
  approx_percentile(0.50, percentile_agg(avg_humed)) AS avg_humed,
  max(max_temp) AS max_temp,
  max(max_humed) AS max_humed
FROM data_minuto
GROUP BY 1
ORDER BY 1;
`);


    response.status(200).json(medymax);
});




/**INICIO SERVER EXPRESS Y CONEX AL CLIENTE REDIS **/


/** Iniciamos la aplic en escucha por el puerto 3000 de dentro de la red docker
app.listen (3000, async() =>
{

    await redis.connect();
    console.log ('Escuchando por el puerto interno de la red docker: 3000')

})
```

4. Swagger.js

```
//Importo modulo

//Autor: Amando Antoñano Puerta
const swaggerJSDoc = require ('swagger-jsdoc');

const options = {
    definition: {
        openapi: '3.0.0',
        info:
        {
            title: 'Mi trabajo de Fin de Grado',
            version: '0.0.1',
            description: 'Aplicacion que recibe datos de temp/humed de un sensor DHT11 y los guarda en una base de datos de series temporales con caché'
        },
    },

    apis:['./index.js'],  //Le indico la ruta del archivo donde se encuentran las anotaciones swagger a interpretar
};

const swaggerSpec = swaggerJSDoc(options);
module.exports = swaggerSpec  //Exporto para que pueda ser utiliz en otras partes del contenedor nodejs
```

5. Scripts

5.1. vistasMaterializadas.sh

```
#!/bin/bash

##Autor: Amando Antoñano Puerta

#########################################################################################################
#######Shellscript que ejecutaremos con docker y que nos creara las MATEREALIZED VIEW que son una caract
#########################################################################################################

echo "======================"
echo "Creando la vista materializada temp_y_humed_porMin en la base de datos seriesTemporalesDB..."
echo "======================"


#Creamos la materialized view temp_y_humed_porMin

docker exec -i seriesTemporalesDB psql -U postgres -d seriesTemporalesDB -c "
CREATE MATERIALIZED VIEW IF NOT EXISTS temp_y_humed_porMin
WITH (timescaledb.continuous) AS
SELECT
  time_bucket ('1 minute', medidas_sensor.timestamp, 'Europe/Berlin') AS time,
  approx_percentile(0.50, percentile_agg(temperatura)) AS avg_temp,
  approx_percentile(0.50, percentile_agg(humedad)) AS avg_humed,
  max(temperatura) as max_temp,
  max(humedad) as max_humed
FROM medidas_sensor
GROUP BY 1;
"

echo "Vista materializada de temperatura/humedad por minuto creada correctamente."
echo "======================"
echo "Proceso completado."
echo "======================"
```

```
docker exec -i seriesTemporalesDB psql -U postgres -d seriesTemporalesDB -c "
SELECT add_continuous_aggregate_policy
  ('temp_y_humed_porMin',
  start_offset => NULL,
  end_offset => INTERVAL '1 minute',
  schedule_interval => INTERVAL '1 minute');
"

##############################################################################################
##############################################################################################
##############################################################################################

##Quiero crear una consulta que me diga las temperaturas/humedad media y maxima por dia de la semana

echo "======================"
echo "Creando la vista materializada temp_y_humed_porDia en la base de datos seriesTemporalesDB..."
echo "======================"

docker exec -i seriesTemporalesDB psql -U postgres -d seriesTemporalesDB -c "
CREATE MATERIALIZED VIEW temp_y_humed_porDia
WITH (timescaledb.continuous) AS
SELECT
  time_bucket ('1 day', medidas_sensor.timestamp, 'Europe/Berlin') AS time,
  approx_percentile(0.50, percentile_agg(temperatura)) AS avg_temp,
  approx_percentile(0.50, percentile_agg(humedad)) AS avg_humed,
  max(temperatura) AS max_temp,
  max(humedad) AS max_humed
FROM medidas_sensor
GROUP BY 1;
"

echo "Vista materializada de temperatura/humedad por dia creada correctamente."
echo "======================"
echo "Proceso completado."
echo "======================"
```

```
docker exec -i seriesTemporalesDB psql -U postgres -d seriesTemporalesDB -c "
SELECT add_continuous_aggregate_policy(
  'temp_y_humed_porDia',
  start_offset => NULL,
  end_offset => INTERVAL '1 hour',
  schedule_interval => INTERVAL '1 hour'
);
"


##############################################################################################
##############################################################################################
##############################################################################################


##Creamos la materialized view temp_y_humed_porHora

echo "======================"
echo "Creando la vista materializada temp_y_humed_porHora en la base de datos seriesTemporalesDB..."
echo "======================"

docker exec -i seriesTemporalesDB psql -U postgres -d seriesTemporalesDB -c "
CREATE MATERIALIZED VIEW IF NOT EXISTS temp_y_humed_porHora
WITH (timescaledb.continuous) AS
SELECT
  time_bucket('1 hour', timestamp, 'Europe/Berlin') AS time,
  approx_percentile(0.50, percentile_agg(temperatura)) AS avg_temp,
  approx_percentile(0.50, percentile_agg(humedad)) AS avg_humed,
  max(temperatura) AS max_temp,
  max(humedad) AS max_humed
FROM medidas_sensor
GROUP BY 1;
"
```

```
echo "Vista materializada de temperatura/humedad por Hora creada correctamente."
echo "======================"
echo "Proceso completado."
echo "======================"

##Establezco que se inicie desde el principio de los datos en la tabla. Que los dat
##se actualizara cada 1h.

docker exec -i seriesTemporalesDB psql -U postgres -d seriesTemporalesDB -c "
SELECT add_continuous_aggregate_policy(
  'temp_y_humed_porHora',
  start_offset => NULL,
  end_offset => INTERVAL '1 hour',
  schedule_interval => INTERVAL '1 hour'
);
"
```

```
docker exec -i seriesTemporalesDB psql -U postgres -d seriesTemporalesDB -c "
WITH datos_hoy AS (
  SELECT *
  FROM medidas_sensor
  WHERE timestamp::date = now()::date
),
agrupado AS (
  SELECT
    EXTRACT(HOUR FROM timestamp) AS hora,
    avg(temperatura) AS avg_temp,
    avg(humedad) AS avg_humed,
    max(temperatura) AS max_temp,
    max(humedad) AS max_humed
  FROM datos_hoy
  GROUP BY 1
),
max_temp_hora AS (
  SELECT DISTINCT ON (EXTRACT(HOUR FROM timestamp))
    EXTRACT(HOUR FROM timestamp) AS hora,
    timestamp AS time_max_temp
  FROM datos_hoy
  ORDER BY EXTRACT(HOUR FROM timestamp), temperatura DESC
),
max_humed_hora AS (
  SELECT DISTINCT ON (EXTRACT(HOUR FROM timestamp))
    EXTRACT(HOUR FROM timestamp) AS hora,
    timestamp AS time_max_humed
  FROM datos_hoy
  ORDER BY EXTRACT(HOUR FROM timestamp), humedad DESC
)
```

```
),
max_humed_hora AS (
  SELECT DISTINCT ON (EXTRACT(HOUR FROM timestamp))
    EXTRACT(HOUR FROM timestamp) AS hora,
    timestamp AS time_max_humed
  FROM datos_hoy
  ORDER BY EXTRACT(HOUR FROM timestamp), humedad DESC
)
SELECT
  a.hora,
  a.avg_temp,
  a.avg_humed,
  a.max_temp,
  mth.time_max_temp,
  a.max_humed,
  mhh.time_max_humed
FROM agrupado a
LEFT JOIN max_temp_hora mth ON a.hora = mth.hora
LEFT JOIN max_humed_hora mhh ON a.hora = mhh.hora
ORDER BY a.hora;
"
```

```
#Consulta que devolverá por días la temperatura/humedad media y el valor máximo
docker exec -i seriesTemporalesDB psql -U postgres -d seriesTemporalesDB -c "
WITH datos AS (
  SELECT
    time,
    avg_temp AS temp,
    avg_humed AS humed,
    max_temp,
    max_humed
  FROM temp_y_humed_porDia
  WHERE time AT TIME ZONE 'Europe/Berlin' > date_trunc('month', now()) - interval '1 year'
),
estadisticas AS (
  SELECT
    to_char(time, 'Dy') AS dia,
    approx_percentile(0.50, percentile_agg(temp)) AS avg_temp,
    approx_percentile(0.50, percentile_agg(humed)) AS avg_humed,
    max(max_temp) AS max_temp,
    max(max_humed) AS max_humed
  FROM datos
  GROUP BY 1
)
SELECT
  d.dia,
  d.ordinal,
  e.avg_temp,
  e.avg_humed,
  e.max_temp,
  e.max_humed
FROM unnest(array['Sun','Mon','Tue','Wed','Thu','Fri','Sat']) WITH ORDINALITY AS d(dia, ordinal)
LEFT JOIN estadisticas e ON lower(e.dia) = lower(d.dia);
"
```

5.2. run.sh

```
#!/bin/bash

##Autor: Amando Antoñano Puerta

##Este script contiene las lineas a ejecutar para poner en marcha el trabajo.
##También se puede coger linea por linea e ir metiendolas en la terminal.


#Inicio contenedores definidos en el compose. La opc -d es para hacerlo en 2º plano
docker-compose up -d
sleep 5 #Dejo tiempo para que se inicien todos los contenedores bien

echo "Creando tabla medidas_sensor en la base de datos seriesTemporalesDB..."
```

```
echo "Creando tabla medidas_sensor en la base de datos seriesTemporalesDB..."
##Ejecuto comando SQL dentro del contenedor seriesTemporalesDB usando el cliente psql de PostgreSQL.
##Creo tabla medidas_Sensor con 4 columnas: marcatiempo, identific, humedad, temperatura.
docker exec -i seriesTemporalesDB psql -U postgres -d seriesTemporalesDB -c 'create table medidas_sensor (timestamp timestamp with time zone default now(), humedad REAL, temperatura REAL);'


##Ejecuto una funcion de TimeScaleDB para convertir la tabla anterior en una hypertable que nos permitira manejar de forma eficiente series temporales y grandes volum de datos
docker exec -i seriesTemporalesDB psql -U postgres -d seriesTemporalesDB -c "SELECT create_hypertable('medidas_sensor', 'timestamp');"

echo "Tabla medidas_sensor creada y convertida en hypertable."
```

5.3. remove.sh

```
#!/bin/bash

docker compose down ##Detiene todos los co
## docker-compose down -v #Igual que el an

#Estas 2 no hacen falta pero por asegurar
docker compose rm #Elim conten detenidos
docker rmi redis-cache-nodejs #Elim la ima
```

ANEXO B: INSTRUCCIONES Y COMANDOS

En el siguiente anexo se adjunta una captura con instrucciones para arrancar el proyecto de 0 y posteriormente una serie de comandos de utilidad.

1. Instrucciones

```
Pasos para arrancar de 0 el proyecto en docker

1) npm init -y
2) npm install express body-parser cors swagger-ui-express swagger-jsdoc pg redis serialport knex

3) Para arrancar de cero, elimino todo lo que pudiera haber:
    3.1) sudo docker-compose down      --> Apagar y eliminar contenedores, red y volúmenes definidos en docker-compose
    3.2) sudo docker-compose down -v   --> (Opcional) Si quieres borrar también los volúmenes de datos persistentes, en este caso, las tablas de datos creadas
```

```
4) Construyo imagen (solo usar cuando modificas otra vez el Dockerfile, las dependencias, o quieres forzar reconstrucción):
    4.1) sudo docker-compose build            --> Para construir la imagen docker
    4.2) sudo docker-compose build --no-cache  --> Para construir la imagen cero sin usar el cache (recomendada)

5) Levanto los contenedores:
    5.1) sudo docker-compose up -d                    --> Levanta todos los contenedores definidos en el dockercompose.yaml
    5.2) sudo docker-compose up -d -nombreContenedor   --> Levanta solo el contenedor que le digas

6) Este comando hace lo que el paso 4 y 5 de una: sudo docker-compose up --build   --> Construye imagen y levanta todos los contenedores.

7) Para ver logs: sudo
    6.1) sudo docker-compose logs -f         --> Muestra log en tiempo real
    6.2) sudo docker-compose logs -f nombreContenedor  --> Muestra log del contenedor que le indiques

8) Para ver el estado de los contenedores: sudo docker ps
```

2. Comandos

```
Comandos Docker para gestionar la tabla y ver datos

# Entrar a la base de datos (modo interactivo)
sudo docker exec -it seriesTemporalesDB psql -U postgres -d seriesTemporalesDB

# Mostrar tablas (relations)
\dt

# Ver las primeras 10 filas
SELECT * FROM medidas_sensor LIMIT 10;

# Ver el esquema de la tabla
\d medidas_sensor;

# Borrar la tabla
DROP TABLE IF EXISTS medidas_sensor CASCADE;

# Salir del cliente psql
\q
```

```
Puedes ejecutar estos comandos directamente sin entrar al cliente psql:

# Mostrar tablas (relations)
sudo docker exec -i seriesTemporalesDB psql -U postgres -d seriesTemporalesDB -c "\dt"

# Ver las primeras 10 filas
sudo docker exec -i seriesTemporalesDB psql -U postgres -d seriesTemporalesDB -c "SELECT * FROM medidas_sensor LIMIT 10;"

# Ver las ultimas 10 filas insertadas
sudo docker exec -i seriesTemporalesDB psql -U postgres -d seriesTemporalesDB -c "SELECT * FROM medidas_sensor ORDER BY timestamp DESC LIMIT 10;"
```

```
Pasos para comprobar que el dispositivo arduino esta funcionando

1) ls /dev/ttyUSB*
2) Debe salir: /dev/ttyUSB0
3) sudo cat /dev/ttyUSB0
4) Debes ver como llegan las medidas de temp y humed
```

REFERENCIAS

[1] E. S. Cardoso, «Github,» [En línea]. Available: https://github.com/enriqueesanchz/redis-cache.

[2] Acer, «Acer Store,» [En línea]. Available: https://store.acer.com/es-es/acer-travelmate-p2-portatil-tmp215-53-negro-nx-vqbeb-00g?srsltid=AfmBOooJ6zADz10exlAauJ15GrU7L-xdMCJpHzs2ApBGu_Nza-7vRYSw.

[3] AZ-Delivery, «AZ-Delivery,» [En línea]. Available: https://www.az-delivery.de/es/products/mikrocontroller-board.

[4] Mkelectrónica, «Mkelectrónica,» [En línea]. Available: https://mkelectronica.com/producto/sensor-temperatura-humedad/.

[5] Arduino, «Arduino,» [En línea]. Available: https://docs.arduino.cc/.

[6] JavaScript, [En línea]. Available: https://developer.mozilla.org/es/docs/Learn_web_development/Core/Scripting/What_is_JavaScript.

[7] Node.js, [En línea]. Available: https://nodejs.org/es/about.

[8] Redis, «Redis,» [En línea]. Available: https://redis.io/docs/latest/.

[9] Docker, «Docker,» [En línea]. Available: https://docs.docker.com/get-started/.

[10] Visual Studio Code, «Visual Studio Code,» [En línea]. Available: https://code.visualstudio.com/docs.

[11] RedHat, «Redhat,» [En línea]. Available: https://www.redhat.com/es/topics/internet-of-things/what-i iot.

[12] Karpagan Institute of Technology, «Karpagamtech,» [En línea]. Available: https://karpagamtech-ac-in.translate.goog/iot-evolution-future-impact/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=rq#:~:text=By%20the%202000s%2C%20IoT%20evolution,manage%20objects%20in%20real%20time..

[13] Q. Jones, «DIGI,» [En línea]. Available: https://es.digi.com/blog/post/iot-based-environmenta monitoring.

[14] Alpha Telecom Solutions, «alphaenginyeria,» [En línea]. Available: https://alphaenginyeria.com/capas-arquitectura-iot.

[15] Universidad de Valladolid, «UVA,» [En línea]. Available: https://www5.uva.es/estadmed/datos/series/series.htm.

[16] MathWorks, «mathworks,» [En línea]. Available: https://la.mathworks.com/discovery/time-series-analysis.html.

[17] IONOS, «ionos,» [En línea]. Available: https://www.ionos.es/digitalguide/hosting/cuestiones-

tecnicas/que-es-influxdb/.

[18] OpenTSDB, «opentsdb,» [En línea]. Available: https://opentsdb.net/docs/build/html/index.html.

[19] influxdata, «influxdata,» [En línea]. Available: https://www.influxdata.com/comparison/mongodb-vs-tsdb/#:~:text=from%20various%20sources.-,IoT%20Data%20Storage,real%2Dtime%20insights%20and%20analytics..

[20] Graphite, «graphite,» [En línea]. Available: https://graphite.readthedocs.io/en/latest/.

[21] QuestdDB, «github,» [En línea]. Available: https://github.com/questdb/questdb.

[22] Prometheus, «prometheus,» [En línea]. Available: https://prometheus.io/docs/introduction/overview/.

[23] IBM, «ibm,» [En línea]. Available: https://www.ibm.com/es-es/topics/postgresql.

[24] Postgresql, «postgresql,» [En línea]. Available: https://www.postgresql.org/about/.

[25] PostgreSQL, «postgresql,» [En línea]. Available: https://www.postgresql.org/docs/current/populate.html.

[26] PostgreSQL, «postgresql,» [En línea]. Available: https://www.postgresql.org/docs/13/release-13.html.

[27] Timescale, «timescale,» [En línea]. Available: https://docs.timescale.com/use-timescale/latest/hypertables/.

[28] A. Valialkin, «valyala medium,» [En línea]. Available: https://valyala.medium.com/high-cardinality-tsdb-benchmarks-victoriametrics-vs-timescaledb-vs-influxdb-13e6ee64dd6b.

[29] Y. Hwang, «questdb,» [En línea]. Available: https://questdb.com/blog/comparing-influxdb-timescaledb-questdb-time-series-databases/.

[30] M. Freedman, «Medium,» [En línea]. Available: https://medium.com/timescale/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877.

[31] J. Blackwood Sewell, «DEV,» [En línea]. Available: https://dev.to/timescale/timescaledb-in-2024-making-postgres-faster-32f7.

[32] Amazon, «AWS,» [En línea]. Available: https://aws.amazon.com/es/memcached/.

[33] ManageEngine, «site24x7,» [En línea]. Available: https://www.site24x7.com/learn/memcached-vs-redis-comparison.html.

[34] Amazon, «AWS,» [En línea]. Available: https://aws.amazon.com/es/elasticache/what-is-redis/. [Último acceso: 05 Junio 2025].

[35] Ignite Apache, «ignite apache,» [En línea]. Available: https://ignite.apache.org/.

[36] Couchbase, «couchbase,» [En línea]. Available: https://www.couchbase.com/es/.

[37] Redis, «redis,» [En línea]. Available: https://redis.io/es/redis-enterprise/estructuras-de-datos/.

[38] ElWillie, «elwillie,» [En línea]. Available: https://elwillie.es/2022/10/17/redis-persistencia-y-durabilidad/.

[39] P. Khandelwal. [En línea]. Available: https://medium.com/@khandelwal.praful/understanding-redis-high-availability-cluster-vs-sentinel-420ecaac3236.

[40] R. Contreras, «computing,» [En línea]. Available: https://www.computing.es/informes/contenedores-software-que-son-ventajas-aplicacion/.

[41] Docker, [En línea]. Available: https://docs.docker.com/engine/network/.

[42] A. Fernández. [En línea]. Available: https://anderfernandez.com/blog/tutorial-docker-compose/.

[43] Autor, «Este es el ejemplo de una cita,» *Tesis Doctoral,* vol. 2, nº 13, 2012.

[44] O. Autor, «Otra cita distinta,» *revista,* p. 12, 2001.