Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica per il Management

Gestione di Serie Temporali con TimescaleDB: un'analisi sperimentale

Relatore:
Chiar.mo Prof.
MARCO DI FELICE

Presentata da:
ANDREA SERRANO

Sessione I
Anno Accademico 2024/2025

La qualità? Sì, vabbè... Ma mica stiamo a fa' Kubrick!

René Ferretti, Boris

Abstract

Nell'attuale contesto della raccolta e gestione di grandi volumi di dati temporali, le **serie temporali** assumono un ruolo cruciale in numerosi ambiti applicativi, dall'Internet of Things al monitoraggio ambientale, dai sistemi industriali ai servizi digitali. Tuttavia, le peculiarità di questo tipo di dati, ad alta frequenza di registrazione e rapida crescita nel volume, pongono sfide significative in termini di scalabilità, efficienza delle interrogazioni e persistenza nel lungo periodo.

Questa Tesi esplora le potenzialità di **TimescaleDB**, un'estensione per PostgreSQL progettata specificamente per la gestione efficiente delle serie temporali, confrontandola con soluzioni alternative tradizionali e specializzate. Viene approfondita l'architettura di TimescaleDB, evidenziando le sue soluzioni innovative come l'approccio ibrido tra database relazionali e soluzioni pensate ad-hoc per le time-series, come l'uso di hypertable e chunking automatico, nonché le ottimizzazioni per query aggregate e compressione dei dati.

Attraverso un caso di studio basato su dati raccolti da un sensore reale, viene simulata la strutturazione di un'applicazione concreta di gestione di serie temporali. Questo scenario funge da riferimento pratico per introdurre le problematiche tipiche del dominio. I test prestazionali veri e propri, invece, vengono condotti separatamente su dati generati artificialmente, in scenari controllati, al fine di analizzare in modo sistematico le performance del sistema in termini di **scalabilità**, **tempi di risposta**, **efficienza delle query** e **capacità di compressione**. I risultati evidenziano l'efficienza complessiva di TimescaleDB nell'elaborazione e compressione di grandi volumi di dati temporali, anche in scenari ad elevato carico.

L'obiettivo è comprendere vantaggi, limiti e implicazioni pratiche dell'adozione di una soluzione come TimescaleDB nella gestione di time series a lungo termine. Infine, vengono presentate alcune considerazioni conclusive e possibili sviluppi futuri, con particolare attenzione all'evoluzione delle tecnologie per la gestione di dati temporali su larga scala.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Contesto e motivazioni

Negli ultimi anni, l'enorme crescita dei dispositivi connessi e dei sistemi di monitoraggio, ha generato una quantità sempre maggiore di dati strutturati come serie temporali. Questi dati, caratterizzati da valori associati a istanti di tempo, trovano applicazione in numerosi ambiti, tra cui l'Internet of Things (IoT), il monitoraggio ambientale, i sistemi finanziari, il controllo industriale e la gestione di infrastrutture.

Gestire e analizzare in modo efficiente questo tipo di informazioni comporta diverse sfide, sia dal punto di vista dell'archiviazione che dell'elaborazione. La principale difficoltà deriva dall'elevato volume di dati, conseguenza diretta dell'alta frequenza di acquisizione, e dalla necessità di eseguire analisi su una tipologia di dato che, di fatto, si presenta come una "lista potenzialmente infinita" di **coppie chiave:valore**.

In questo contesto, le basi di dati relazionali, così come pensate tradizionalmente, non sempre si dimostrano ottimali, rendendo necessario l'utilizzo di soluzioni specializzate.

TimescaleDB nasce come estensione di PostgreSQL con l'obiettivo di ottimizzare la gestione delle serie temporali, unendo i vantaggi dei database relazionali alla scalabilità e alle performance richieste per questo tipo di dato.

1.2 Obiettivi della tesi

L'obiettivo principale di questa tesi è analizzare il tema della gestione delle serie temporali, con particolare attenzione all'utilizzo di TimescaleDB.

Nello specifico, il lavoro si propone di:

- Fornire una panoramica teorica sulle serie temporali e sulle problematiche legate alla loro gestione;

- Descrivere l'architettura e le funzionalità principali di TimescaleDB;
- Progettare e realizzare una sperimentazione pratica basata sulla raccolta di dati ambientali tramite un dispositivo ESP32 dotato di sensore DHT-11, utile al rilevamento di umidità e temperatura, con invio dei dati ad un database PostgreSQL esteso con TimescaleDB;
- Analizzare le performance di TimescaleDB rispetto a soluzioni alternative attraverso test mirati;
- Valutare i vantaggi offerti da TimescaleDB sulla base delle caratteristiche architetturali precedentemente analizzate.

1.3 Struttura del lavoro

Il percorso di questa tesi si è articolato attraverso diverse fasi operative, che combinano studio teorico, progettazione sperimentale e analisi dei risultati.

In una prima fase, è stato affrontato lo studio teorico delle serie temporali, con particolare riferimento al testo *Practical Time Series Analysis: Prediction with Statistics and Machine Learning*[16] proposto dal docente, come riferimento introduttivo. Questo studio ha permesso di comprendere le caratteristiche fondamentali dei dati temporali, le tecniche statistiche e le problematiche più comuni nella loro gestione e analisi.

Successivamente, è stata progettata e realizzata una demo sperimentale, che ha consentito di mettere in pratica i concetti appresi e di configurare un ambiente concreto di raccolta e gestione di serie temporali.

A seguire, è stato realizzato l'ambiente di test necessario per valutare le prestazioni di TimescaleDB, configurando diversi scenari di carico e definendo le metriche di valutazione, utilizzando il linguaggio di programmazione Python.

Parallelamente, sono state analizzate le caratteristiche architetturali di TimescaleDB, basandosi sul paper *Timescale Architecture for Real-Time Analytics*[15] presente nella documentazione ufficiale di Timescale, con l'obiettivo di comprenderne a fondo il funzionamento e le ottimizzazioni specifiche per la gestione delle serie temporali.

Questa analisi ha permesso di progettare test mirati e coerenti con le peculiarità della piattaforma, individuando gli aspetti più critici e significativi da valutare nella fase di test.

Infine, sono stati condotti i suddetti test di performance, confrontando TimescaleDB con PostgreSQL puro e analizzando i risultati ottenuti in termini di velocità di interrogazione. Questa fase ha permesso di valutare empiricamente i vantaggi offerti da TimescaleDB per la gestione delle serie temporali.

L'intero lavoro è stato quindi strutturato per integrare studio teorico e sperimentazione pratica, con l'obiettivo di fornire una valutazione completa ed equilibrata della tecnologia analizzata.

Capitolo 2

Serie temporali: stato dell'arte

2.1 Cosa sono le time series

Le *serie temporali*, o *time series*, sono collezioni di dati ordinati cronologicamente, in cui ogni osservazione è associata a un preciso istante di tempo. È importante sottolineare che ogni osservazione può consistere non solo in un singolo valore, ma anche in un insieme di variabili rilevate contemporaneamente [16].

Questo tipo di dato è caratterizzato proprio dal fatto che il tempo rappresenta una variabile fondamentale, in quanto il valore di ciascun punto è strettamente dipendente dal momento in cui è stato rilevato.

Esempi comuni di serie temporali includono rilevazioni meteorologiche, prezzi finanziari, consumi energetici, dati biometrici e sensori IoT [16] [20].

La rilevazione di dati in formato *time series* trova la sua utilità nelle funzionalità di data analytics. Esse infatti consentono di studiare l'andamento di un fenomeno nel tempo,

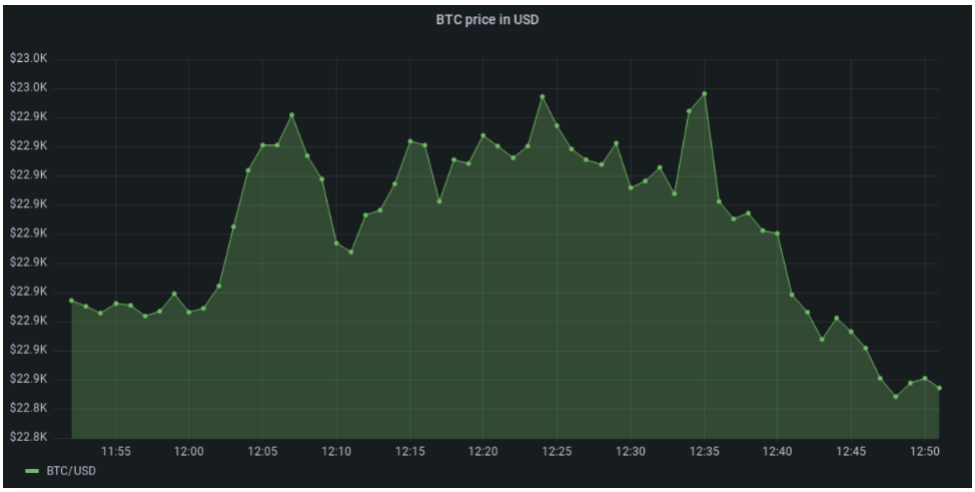

Figura 2.1: Serie temporale applicata al settore finanziario [20]

individuando tendenze, ciclicità, stagionalità e anomalie[16]. E' necessario sottolineare, però, che a differenza dei dati convenzionali, le serie temporali richiedono strumenti e tecniche specifiche, in grado di tener conto della dipendenza temporale tra le osservazioni.

Da qui nasce una branca specifica della statistica, denominata *Time Series Analysis*. Basandoci sulla definizione proposta dal testo di riferimento *Practical Time Series Analysis: Prediction with Statistics and Machine Learning*[16], essa è definita come:

> *"Time Series Analysis is the endeavor to extract meaningful summary and statistical information from points arranged in chronological order. It is done to diagnose past behavior as well as to predict future behavior."*

Per perseguire questi obiettivi è possibile ricorrere sia a tecniche statistiche classiche, consolidate nel tempo, sia a metodi più moderni basati sull'apprendimento automatico e sulle reti neurali, capaci di modellare fenomeni complessi e non lineari[16].

Sebbene l'approfondimento di queste metodologie non rientri tra gli obiettivi di questa trattazione, è importante sottolinearne il valore pratico e il potenziale applicativo.

2.2 Caratteristiche dei dati temporali

Una delle caratteristiche principali di una serie temporale è la seguente: i suoi dati non necessitano aggiornamenti[16]. Questo rende l'accesso casuale per operazioni di scrittura, un elemento di bassa priorità rispetto ad altri tipi di dato, in quanto queste ultime avvengono in modo sequenziale. Di conseguenza, l'ottimizzazione delle performance per operazioni di scrittura, da sempre obiettivo dei database, non è prioritario.

Le caratteristiche principali dei dati temporali sono:

- le operazioni di scrittura tendono a dominare quelle di lettura;
- i dati vengono letti e scritti seguendo l'ordine della sequenza temporale, rispettando la cronologia degli eventi;
- il tempo rappresenta la chiave primaria per l'accesso ai dati, con ogni record che è associato a un preciso istante temporale;
- l'eliminazione di un gran numero di record in un'unica operazione (*bulk deletes*) è più comune rispetto alla rimozione di singoli punti di dati[25].

2.3 Criticità per l'analisi

Vi sono inoltre caratteristiche e criticità tipiche delle serie temporali che, sebbene non rientrino nel nostro ambito di interesse, risultano fondamentali nell'analisi dei dati e saranno pertanto brevemente richiamate per una piena comprensione di queste ultime.

La più evidente, e già accennata, è la *dipendenza temporale* tra le osservazioni: il valore in un certo istante dipende, in modo diretto o indiretto, dai valori precedenti [16].

Un'altra caratteristica importante è la possibile presenza di *trend* e *stagionalità*: il trend rappresenta un andamento di lungo periodo, crescente o decrescente, mentre la stagionalità riguarda variazioni cicliche che si ripetono a intervalli regolari [16]. Individuare e separare queste componenti è fondamentale per ottenere analisi affidabili e previsioni corrette.

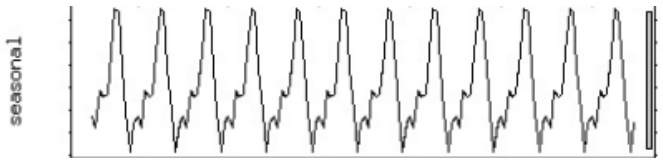

Figura 2.2: Stagionalità dei dati evidenziati con un line plot [16]

Un aspetto spesso complesso da gestire è la *non-stazionarietà* dei dati. Una serie si definisce stazionaria quando le sue proprietà statistiche, come media e varianza, non cambiano nel tempo [16]. La non-stazionarietà rende più difficile applicare modelli tradizionali in quanto la serie tende ad avere un comportamento poco prevedibile.

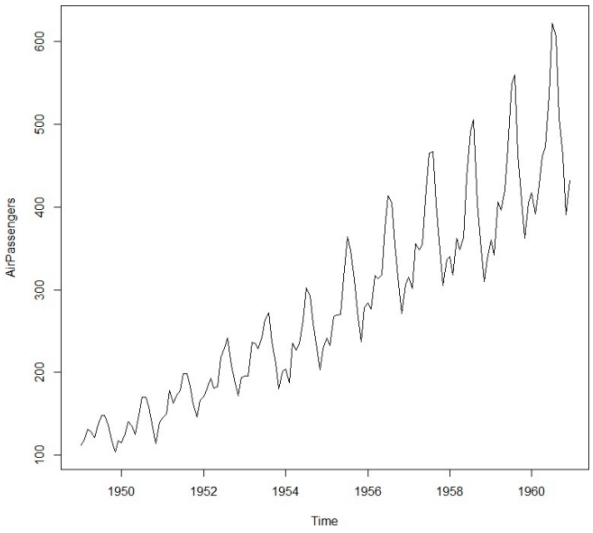

Figura 2.3: Esempio di non-stazionarietà, con il passare del tempo sia la media che la varianza stanno cambiando [16]

Un'ulteriore criticità è legata alla *gestione delle anomalie*. Le serie temporali possono infatti presentare valori anomali, come picchi improvvisi o dati mancanti, che rischiano di alterare i risultati delle analisi. È quindi importante individuare e trattare correttamente queste anomalie, ad esempio usando le giuste tecniche di imputazione per i dati mancanti[16].

Una caratteristica di nostro interesse, invece, è la *frequenza* con cui i dati vengono acquisiti. Le serie temporali possono avere frequenze molto diverse: dai dati finanziari registrati ogni millisecondo ai dati climatici raccolti quotidianamente, mensilmente o annualmente[16]. Le serie ad alta frequenza generano quantità di dati molto elevate, che pongono sfide sia per la *memorizzazione* che per l'*elaborazione*.

2.4 Modelli di archiviazione

Proprio la *dimensione dei dati* è una delle principali difficoltà nella gestione delle serie temporali. In contesti di monitoraggio continuo o acquisizione massiccia, la quantità di informazioni può mettere in difficoltà i sistemi tradizionali basati su database relazionali, che spesso non garantiscono prestazioni adeguate per gestire grandi volumi di dati ed in continua crescita[16].

Una soluzione di archiviazione ottimale deve garantire facilità di accesso e affidabilità dei dati, mantenendo contenuti i costi in termini di risorse computazionali[16].

Nel caso delle serie temporali, è fondamentale scegliere il modello di *storage* più adatto in base al contesto applicativo e alle modalità di utilizzo dei dati.

A seconda dello scenario, infatti, il sistema di archiviazione dovrà supportare modalità di accesso diverse, gestire volumi di dati variabili nel tempo e, in alcuni casi, automatizzare operazioni di pulizia o trasformazione dei dati[16].

Di seguito vengono descritti tre casi tipici di utilizzo, confrontati in base a criteri rilevanti per la gestione delle serie temporali.

Caratteristica	**Dati che crescono nel tempo**	**Pull da una repository**	**Dataset proprietario**
Crescita del dataset	Crescita continua, si gestiscono i dati più vecchi per mantenere dimensioni contenute	Crescita stabile, il dataset non cresce nel tempo	Crescita continua, collezione ampia e in espansione
Prestazioni in funzione della dimensione	Poco rilevanti, grazie alla rimozione periodica dei dati	Molto rilevanti, poiché il dataset è grande e stabile	Molto rilevanti, il dataset cresce costantemente
Accesso ai dati	Più frequente sui dati recenti (accesso sequenziale)	Frequenza uniforme su tutti i dati (accesso casuale)	Più frequente sui dati recenti, ma con accesso misto
Automazione degli script di manutenzione	Fondamentale per la gestione automatica dei dati vecchi	Non necessaria, i dati sono statici e immutabili	Utile per automatizzare il pre-processing e la gestione delle diverse scale temporali

Tabella 2.1: Confronto tra diversi modelli di archiviazione per serie temporali [16]

Come si può osservare, i requisiti di archiviazione cambiano sensibilmente in base al contesto:

- Nei sistemi dove i dati crescono nel tempo, è prioritario implementare meccanismi di gestione automatica dei dati obsoleti, limitando così la dimensione del dataset e ottimizzando l'accesso alle informazioni più recenti.
- Nel caso del pull da repository, invece, il dataset ha una dimensione definita e stabile: l'attenzione si sposta quindi sulle prestazioni, che devono essere garantite anche per collezioni ampie, senza necessità di script di manutenzione o aggiornamento.
- Infine, nei dataset proprietari, caratterizzati da una collezione eterogenea e in continua crescita, è essenziale adottare soluzioni scalabili e personalizzabili. In questo scenario, l'automazione gioca un ruolo importante, non solo nella gestione dei dati, ma anche nel pre-processing, che può variare a seconda della scala temporale o delle esigenze specifiche del progetto.

2.5 Tecnologie di Archiviazione

In scenari pratici, la selezione della tecnologia di archiviazione è determinata dalle esigenze specifiche del sistema, con una varietà di opzioni tra cui scegliere, come:

- database SQL;
- database NoSQL;
- formati di file di tipo flat [16].

Ognuna di queste tecnologie presenta vantaggi e svantaggi, che esamineremo di seguito.

2.5.1 Database

In generale, i database rappresentano una soluzione robusta e pronta all'uso. I vantaggi principali comprendono [3]:

- Un sistema di archiviazione che può scalare su più server, supportando carichi elevati di dati e richieste.
- Operazioni di lettura e scrittura a bassa latenza, che garantiscono un accesso rapido ai dati.
- Funzioni predefinite per il calcolo di metriche comuni, come aggregazioni e calcoli statistici, riducendo la necessità di implementare logiche personalizzate.
- Strumenti di monitoraggio integrati, utili per analizzare le performance e identificare colli di bottiglia nel sistema.

Un database, in particolare un sistema NoSQL, può essere vantaggioso per mantenere una certa flessibilità nella gestione dei dati [3].

Inoltre, è utile tenere presente che, anche nel caso in cui si scelga un sistema di archiviazione basato su file, in una fase di progettazione iniziale, l'utilizzo di un database può essere utile per determinare la struttura e l'organizzazione dei file stessi, affinché questi ultimi possano essere adattati man mano che i processi di gestione dei dati si evolvono [16].

SQL vs NoSQL Sebbene sia vero che qualsiasi formato di dati possa essere descritto attraverso un opportuno schema di tabelle relazionali, nella pratica si riscontrano difficoltà quando si ha la necessità di scalare per gestire grandi volumi di dati. In questo contesto, una soluzione NoSQL si rivela utile poiché è per sua natura un sistema pensato per essere flessibile e scalabile, capace di adattarsi a esigenze future che potrebbero essere

difficili da prevedere. Questo lo rende particolarmente adatto per scenari in cui si avvia un monitoraggio senza una chiara visione dell'evoluzione che il sistema avrà[16].

Al contrario, i dati tipicamente trattati nei database SQL differiscono dai dati temporali per due aspetti principali: innanzitutto, i punti dati esistenti vengono frequentemente aggiornati, inoltre non è richiesto un ordine nell'accesso ai dati che è tipicamente casuale[16].

Dal punto di vista concettuale, i database NoSQL, dunque, si adattano meglio alla struttura dei dati temporali poiché un dato di tipo time-series può comprendere più dimensioni (ovvero più valori) che potrebbero non essere sempre presenti in tutte le rilevazioni[16]. Questo ci porterebbe a pensare che la via maestra sia l'utilizzo di database NoSQL.

Tuttavia, è possibile utilizzare i database SQL, continuandone a sfruttare i tipici vantaggi (integrità e coerenza grazie ai vincoli, transazioni ACID, query complesse), apportando modifiche architetturali nella gestione dello stesso.
Timescale rappresenta un esempio concreto di questi cambiamenti pensati ad hoc.

2.5.2 Flat File

L'uso di flat file è una soluzione percorribile quando[16]:

- Il formato dei dati è consolidato e non subirà modifiche significative per un lungo periodo di tempo.
- Il processamento dei dati presenta limitazioni legate all'I/O, in quanto la CPU è in grado di elaborare un volume maggiore di dati, mentre il sistema è vincolato dalla velocità delle operazioni su dischi, reti o altre periferiche. Avendo accesso direttamente ai file, è possibile ottimizzare le prestazioni attraverso un fine tuning del sistema.
- Non è necessario un accesso casuale ai dati (come visto, tipico dei dati temporali).

I vantaggi dei flat file

- Indipendenza dal sistema, il che rende il file portabile e utilizzabile su qualsiasi piattaforma.

- L'overhead associato alle operazioni di I/O è inferiore rispetto a quello di un database, poiché si accede direttamente al file, evitando la lettura di strutture dati più complesse.
- Un file flat codifica naturalmente l'ordine in cui i dati devono essere letti, favorendo la lettura sequenziale. Questo non è sempre garantito dai database, che potrebbero non seguire un ordine fisso.
- I dati occupano generalmente meno spazio, in quanto le opportunità di compressione sono massimizzate rispetto a quelle offerte da strutture dati più complesse.

Queste caratteristiche rendono i file flat una scelta vantaggiosa in contesti dove la gestione dei dati è relativamente semplice e non sono richiesti accessi complessi o frequenti aggiornamenti dei dati[16].

Tale soluzione, di contro, presenta un'elevata barriera all'ingresso, dettata dalla minore facilità con cui si possono impostare query complesse, gestire l'integrità dei dati o implementare meccanismi avanzati di indicizzazione e accesso concorrente. Inoltre, l'assenza di un'interfaccia di alto livello per l'interrogazione e la gestione dei dati può aumentare il rischio di errori, richiedendo uno sforzo maggiore in termini di sviluppo e manutenzione del codice[16].

2.5.3 Conclusioni

La scelta della tecnologia più adeguata non può prescindere da un'attenta valutazione del contesto d'uso. Non esiste, infatti, una soluzione che risulti la migliore in termini assoluti, ma solo quella più adatta a soddisfare i requisiti specifici del caso.

Sicuramente, possiamo dire che soluzioni come Timescale rappresentano un compromesso ideale per la gestione di dati temporali.

Basato su un motore SQL, Timescale consente di mantenere i vantaggi dei database relazionali come integrità, coerenza e possibilità di eseguire query complesse ottimizzando al contempo la gestione dei *time-series data* grazie a tecniche architetturali dedicate[25].

Capitolo 3

TimescaleDB: panoramica, architettura e funzionalità

3.1 Panoramica su TimescaleDB

TimescaleDB è un'estensione open source di PostgreSQL, progettata specificamente per l'elaborazione e l'analisi di dati temporali su larga scala[25].

Grazie a una serie di ottimizzazioni architetturali, come l'adozione di uno storage ibrido riga/colonna (*hypercore*), la partizione automatica delle tabelle (*hypertables*) e il supporto per aggregazioni continue (*continuous aggregates*), TimescaleDB si propone come soluzione ideale per applicazioni moderne che richiedono elevate prestazioni in fase di scrittura, capacità di interrogazione rapida su grandi volumi di dati e piena compatibilità con lo standard SQL.

3.1.1 PostgreSQL: la base di Timescale

PostgreSQL è un sistema di gestione di basi di dati relazionali (*RDBMS*) open source tra i più avanzati e completi attualmente disponibili (vedi documentazione ufficiale[21]). E' noto per la sua aderenza agli standard SQL, l'estensibilità e conformità ACID, che lo rende adatto a contesti in cui affidabilità, integrità dei dati e transazioni complesse sono elementi essenziali.

Rispetto ad altri sistemi di gestione come MySQL, anch'esso ampiamente diffuso e open source, PostgreSQL si distingue per il supporto nativo a tipi di dato complessi (come JSONB, array e UUID), l'estensibilità attraverso funzioni definite dall'utente, linguaggi procedurali (es. PL/pgSQL) e il supporto a funzionalità come le viste materializzate, i trigger complessi e l'indicizzazione avanzata (inclusi indici GIN e GiST)[21].

MySQL, per contro, tende ad offrire una configurazione più semplice e prestazioni migliori in scenari di lettura intensiva con modelli dati meno complessi, rendendolo spesso la scelta preferita in ambienti web-oriented e applicazioni in cui la semplicità di gestione sono più rilevanti rispetto ad un sistema necessariamente più sofisticato[26].

3.2 Database row-store e column-store

Timescale utilizza una combinazione di row-store e column-store. Introduciamo ora i due concetti, che verranno utilizzati successivamente[2].

Le modalità di memorizzazione dei dati nei database relazionali influenzano in modo significativo le prestazioni, in particolare in relazione al tipo di carico di lavoro (scrittura intensiva, lettura analitica, ecc.)[15]. In questo contesto, si distinguono due principali approcci: i database **row-store** e quelli **column-store**.

3.2.1 Row-store

Nei database row-store, come PostgreSQL o MySQL, i dati sono memorizzati riga per riga. Di conseguenza, quando viene eseguita una query il sistema accede a tutte le colonne di una determinata riga, anche se l'interrogazione richiede solo un sottoinsieme dei valori (una o più colonne, ma non tutte).

Questa modalità di memorizzazione è particolarmente efficiente nelle operazioni di scrittura e nelle transazioni che coinvolgono molteplici campi di una stessa entità, poiché consente l'inserimento di una riga completa in un'unica operazione, riducendo il numero di accessi alla memoria[2].

3.2.2 Column-store

Nei database column-store, invece, i dati vengono memorizzati colonna per colonna. Questo approccio si rivela estremamente vantaggioso nelle operazioni analitiche, in cui si interrogano solo specifiche colonne su un elevato numero di righe. Il database, in tal caso, può accedere unicamente alle colonne richieste, evitando la lettura dell'intera riga per poi prendere solo il dato di interesse.

Ulteriore vantaggio dei sistemi column-store è rappresentato dalla possibilità di applicare tecniche di compressione molto efficaci. Pensiamo, ad esempio, a quando le colonne contengono valori ripetitivi[2].

3.3 Hypertable e Chunk

Secondo la documentazione ufficiale[24], in *TimescaleDB* una tabella viene organizzata in diverse tabelle "figlie" chiamate *chunks*, tale tabella viene dunque chiamata *hypertable*.

3.3.1 Cos'è un chunk

Per essere tale, in una *hypertable*, i dati vengono suddivisi automaticamente in *chunk*, ovvero blocchi separati che contengono i dati relativi a intervalli temporali specifici (ad esempio, un giorno o una settimana). Questa partizione consente al database di eseguire query in modo più efficiente, analizzando solo i chunk rilevanti piuttosto che l'intero dataset. Ogni chunk può inoltre essere gestito, indicizzato e compresso individualmente, migliorando così l'uso delle risorse e le prestazioni su grandi volumi di dati[15].

Esempio. Supponiamo di avere una serie temporale di dati meteorologici registrati ogni minuto. TimescaleDB può suddividerla in chunk settimanali: una query che analizza solo il mese di marzo richiamerà solo quattro chunk, ignorando completamente il resto dell'anno, con evidenti vantaggi in termini di velocità e consumo di risorse.

TimescaleDB permette di configurare **intervalli di tempo differenti per ciascun chunk**, ottimizzando così la gestione dei dati in base alla loro temporalità[15]. Ad esempio, i dati recenti possono essere suddivisi in chunk con intervalli di tempo più piccoli, mentre i dati più vecchi sono raggruppati in chunk più grandi. Questo approccio consente di adattare la gestione delle risorse alle necessità specifiche delle operazioni analitiche, migliorando l'efficienza sia nelle operazioni di scrittura, che vengono eseguite principalmente sui chunk recenti, sia nelle operazioni di lettura, che possono essere più rapide grazie alla presenza di chunk più grandi e compatti per i dati storici[15].

Time-Partitioned vs Space-Partitioned Hypertable In TimescaleDB, le *hypertable* possono essere suddivise in chunk secondo uno schema di partizionamento definito dall'utente[15]. Le due modalità principali sono[25]:

- **Time-partitioned hypertable**: suddivisione basata esclusivamente sulla colonna temporale. È la modalità standard, dove ogni chunk rappresenta un intervallo temporale specifico (es. ogni ora, giorno o settimana).
- **Space-partitioned hypertable**: suddivisione basata sulla colonna temporale e su una o più colonne aggiuntive, dette *dimensioni spaziali*, come `device_id`, `location`, ecc.

Il partizionamento spaziale consente di migliorare la scalabilità e la distribuzione dei dati, soprattutto in scenari con:

- un elevato numero di sorgenti dati (es. sensori, dispositivi, utenti);
- necessità di parallelizzare letture e scritture su più partizioni;
- ottimizzazione di query specifiche per sottogruppi (es. per singolo dispositivo)[25].

La combinazione di partizionamento temporale e spaziale permette a TimescaleDB di distribuire i dati su più chunk in maniera più efficiente, riducendo il contenuto di ciascun chunk e aumentando le opportunità di *chunk exclusion*, migliorando così le prestazioni complessive del sistema.

3.3.2 Hypertables e gestione automatica delle partizioni

Come detto, Timescale introduce il concetto di *hypertable*, una struttura che astrae la partizione dei dati in *chunk*. Questi ultimi, basati tipicamente su timestamp o identificativi crescenti, vengono creati attraverso un processo automatizzato, a differenza dei database relazionali tradizionali, dove la gestione delle partizioni richiede configurazioni manuali e complesse[15].

Durante la fase di *ingestione*, TimescaleDB suddivide automaticamente i dati in *chunk* temporali (Figura 3.1), ciascuno con propri indici e spazio di memorizzazione. Poiché le scritture avvengono principalmente nei chunk più recenti (in una serie temporale aggiungiamo il dato sempre alla fine), che corrispondono a quelli di dimensioni più contenute, si ottengono diversi vantaggi: gli indici rimangono piccoli e veloci da aggiornare; la località della cache migliora, in quanto gli accessi successivi si concentrano su dati "caldi" già presenti in memoria; infine, si riduce la necessità di manutenzione in background (come il *vacuum*), poiché solo pochi chunk attivi necessitano di essere costantemente gestiti, mentre quelli più vecchi rimangono invariati[15].

Inoltre, tale modello mitiga il rischio di *table bloat* e *degrado degli indici*, assicurando performance elevate e costanti anche in scenari ad alto tasso di ingestione[15].

Durante l'esecuzione delle query, l'architettura delle hypertables consente di escludere efficientemente i chunk, a condizione che la colonna di partizionamento sia presente nella clausola `WHERE`. Grazie a questa esclusione intelligente e all'uso di indici locali ai chunk, le query mantengono prestazioni elevate anche su dataset di grandi dimensioni, evitando il degrado progressivo che caratterizza le tabelle non partizionate[15].

Esempio. Se vogliamo accedere a dati di un chunk useremo:

```
WHERE time BETWEEN '2024-01-01' AND '2024-01-31'
```

dove `time`, la colonna con il timestamp, è quella utilizzata automaticamente per il partizionamento.

Notiamo che non specifichiamo l'indice del chunk, questo è qualcosa di astratto per noi ed utilizzato internamente da timescale.

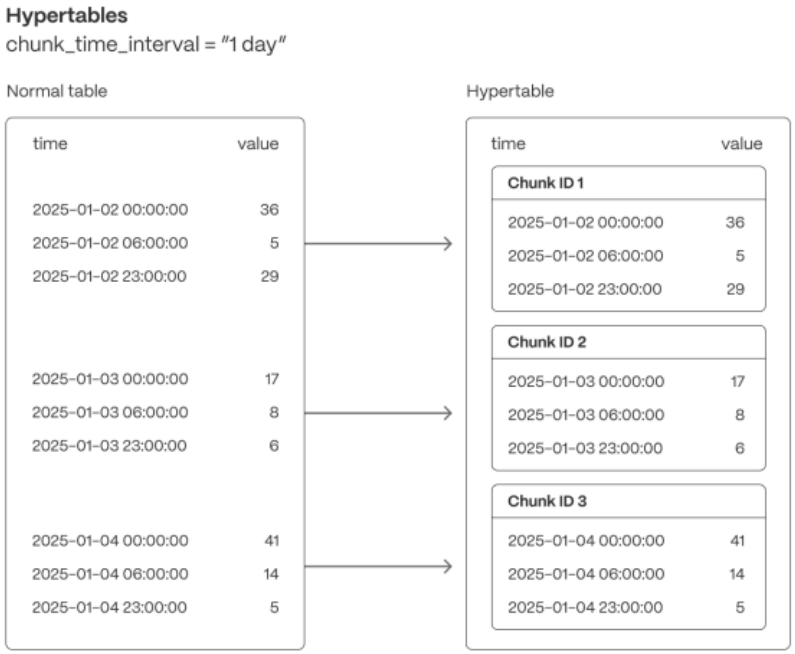

Figura 3.1: Creazione di una hypertable[15]

Di seguito spieghiamo alcuni termini tecnici precedentemente utilizzati:

Vacuum Il `vacuum`[17] è una tecnica di manutenzione utilizzata per pulire e ottimizzare lo spazio di memorizzazione, si procede rimuovendo i dati "morti" (cioè i dati che non sono più necessari) e riempiendo gli spazi vuoti lasciati da operazioni di aggiornamento o eliminazione.

Esistono due principali varianti `vacuum`:

- `Vacuum semplice (Plain Vacuum)`: Rimuove le righe morte e riutilizza lo spazio all'interno delle tabelle, ma non recupera fisicamente lo spazio sui dischi. È utile per evitare che il database cresca eccessivamente a causa delle versioni obsolete dei dati.
- `Vacuum completo (Full Vacuum)`: Recupera anche lo spazio fisico sul disco, riducendo la dimensione del file di archiviazione del database. Questa operazione è più pesante e richiede un blocco esclusivo della tabella durante l'esecuzione.

La scelta tra un `vacuum` semplice e uno completo dipende dalle esigenze specifiche del database e dal carico di lavoro. Ad esempio, in ambienti ad alta concorrenza è preferibile eseguire solo il `vacuum` semplice per evitare blocchi prolungati.

Table bloat Crescita eccessiva di una tabella dettata dalla presenza di dati obsoleti non ancora gestiti[22].

Degrado degli indici In PostgreSQL, le righe aggiornate non sovrascrivono direttamente le precedenti, ma vengono create nuove versioni, lasciando le vecchie versioni come "morte" fino a quando non vengono rimosse. Analogamente, le righe cancellate non vengono immediatamente rimosse, ma segnate come morte. L'indice che punta a tali righe per permettere, in teoria, query più veloci, in realtà si ritrova dati obsoleti ed è costretto ad effettuare una nuova ricerca, diventando così più grande e contenendo voci inutili.

Grazie al sistema di chunk di timescale, questa problematica si limita ai più recenti e diventa più facilmente gestibile[23].

3.4 Hypercore: la soluzione di Timescale

I database tradizionali impongono un compromesso tra la velocità di inserimento dei dati, tipica dei sistemi row-store, e l'efficienza nelle analisi, propria dei sistemi column-store. Hypercore, il motore alla base di TimescaleDB, supera questo limite adottando una strategia di storage ibrido che adatta dinamicamente il formato di memorizzazione in base alla fase del ciclo di vita del dato[15].

Row-storage per i dati più recenti: i chunk (blocchi) di dati più recenti sono memorizzati nel row-store, il che consente inserimenti rapidi, aggiornamenti efficienti e query a bassa latenza su singoli record, cioè la capacità di accedere rapidamente a una singola riga (o piccolo insieme di righe) senza dover scansionare intere colonne, come accadrebbe in un column-store puro[15].

Inoltre, lo storage basato su righe funge anche da *writethrough* per lo storage colonnare: ciò significa che quando un dato viene scritto o aggiornato, esso viene inizialmente scritto nel row-store, garantendo prestazioni ottimali, e successivamente viene trasferito nel column-store per essere compresso e ottimizzato per le analisi[15].

Columnar-storage per prestazioni analitiche: i chunk vengono compressi automaticamente nello storage colonnare, ottimizzando dunque lo spazio occupato e creando una struttura più facile da interrogare per le query analitiche[15].

In questo contesto, "ottimizzare le query analitiche" significa ridurre il numero di dati da leggere accedendo solo alle colonne necessarie e sfruttando l'elevata compressione per migliorare la velocità di scansione e calcolo su grandi dataset.

3.4.1 Layout del columnar storage

A differenza dei column-store tradizionali, il sistema di Timescale supporta insert, upsert, update e delete, anche a livello di singolo record, il tutto continuando a garantire le transazioni[15]. Vediamo perché:

- tipicamente, nei **sistemi column-store**, i dati sono ottimizzati per la lettura in blocco e non sono pensati per essere facilmente modificati, soprattutto se tale modifica riguarda un singolo record. In questi casi per un aggiornamento andremo incontro ad operazioni complesse, come la riscrittura dell'intero blocco colonnare;
- in **timescale**, invece, i dati sono immediatamente visibili alle query dal momento stesso in cui vengono inseriti, a differenza dei sistemi colonnari classici in cui si segue il seguente processo: i dati vengono inizialmente scritti in batch, spesso in una memoria intermedia (buffer o *write-ahead log*); in un secondo momento, essi vengono compattati o caricati in blocchi compressi attraverso un processo noto come *batch ingestion*, *compaction* o *segment merge*;

dunque, fin quando non è finito il processo (o si sta aspettando di riempire il buffer), i dati non sono disponibili per l'analisi[15].

Timescale evita questo problema grazie alla coesistenza di row-store e column-store (Figura 3.2).

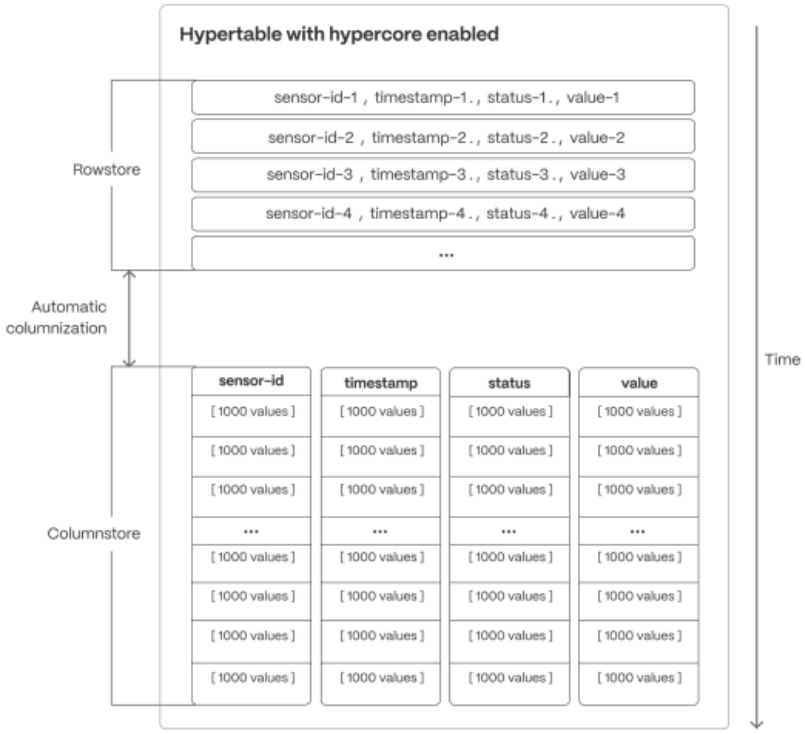

Figura 3.2: Hypertable con hypercore abilitato: ibrido rowstore e columnstore[1] [15]

Batch colonnari: conversione da row a column Nel contesto di TimescaleDB, abbiamo capito che l'uso della compressione colonnare è una strategia chiave per migliorare le performance nelle query analitiche.

La conversione dei chunk da riga a colonna (Figura 3.3) consente una compressione efficace dei dati, riducendo lo spazio di archiviazione e migliorando l'efficienza delle scansioni che ora avvengono per colonne[15].

Le tecniche avanzate di compressione[14], tra cui la *Run-Length Encoding* e la *Gorilla Compression*, permettono una riduzione del footprint fino al 95%, migliorando dunque le prestazioni I/O.

- **Run-Length Encoding (RLE)**: invece di salvare ogni singolo valore, salva *il valore* e *quante volte si ripete consecutivamente* (la "run").
- **Gorilla compression**: sviluppata da Facebook appositamente per le serie temporali, salva solo la *differenza tra timestamp consecutivi*; *per i valori numerici* confronta

[1] I dati vengono inizialmente scritti nel row-store, in formato riga, per garantire inserimenti rapidi e query a bassa latenza. Nel tempo, questi dati vengono automaticamente convertiti in formato column-store, in cui ogni colonna contiene blocchi compressi di valori. Questo approccio ibrido consente di combinare le performance in scrittura del row-store con l'efficienza analitica e la compressione del column-store, tutto all'interno della stessa hypertable, supportando operazioni insert, upsert, update e delete.

il corrente con il precedente in formato binario e *salva solo le parti che cambiano*. È particolarmente efficiente con dati che cambiano poco nel tempo, caratteristica tipica dei punti dati delle serie temporali.

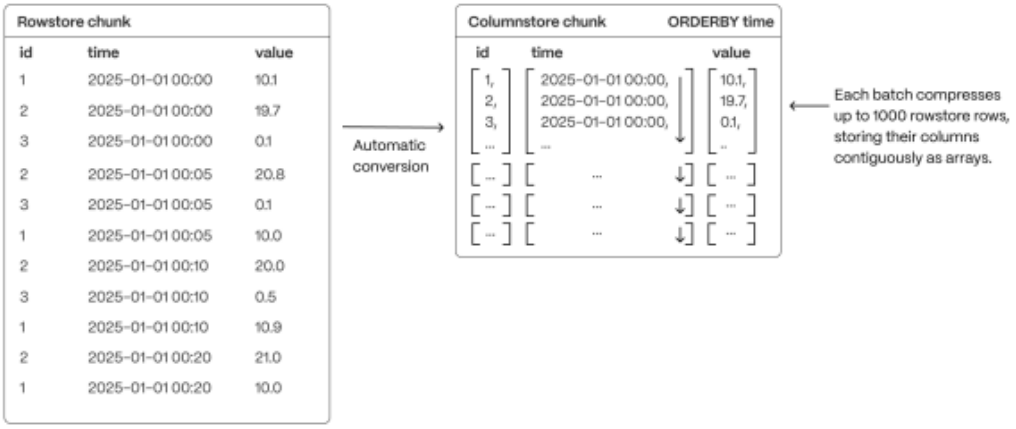

Figura 3.3: Conversione di una rowstore in una columnstore[15]

Vectorized execution e controllo sul layout fisico Per ottimizzare le performance della query, timescale permette di controllare esplicitamente come i dati sono fisicamente organizzati nel column-storage. Grazie a questo, le query possono ridurre significativamente le letture da disco, accedendo soltanto ai dati realmente necessari. Inoltre, l'esecuzione delle query avviene in modalità vettoriale (*vectorized execution*), che permette di elaborare blocchi di dati in parallelo attraverso il *batch processing*, migliorando ulteriormente le prestazioni analitiche[15].

Esempio. Supponiamo di star analizzando milioni di letture di sensori per calcolare la media oraria della temperatura, Timescale: legge solo le colonne `timestamp` e `temperature`; carica blocchi interi in memoria; calcola la media su batch, in parallelo; di conseguenza risparmia tempo e risorse rispetto a un database tradizionale.

Riguardo, invece, il controllo sul layout fisico dei dati, quando questi vengono compressi e spostati nel *column-store*, TimescaleDB consente di specificare in maniera esplicita le colonne da utilizzare come chiavi di ordinamento (*sort keys*). Questa scelta ha un impatto significativo sia sull'efficienza delle query che sulla compressione dei dati.

In primo luogo, l'ordinamento consente al motore di esecuzione di ottimizzare le interrogazioni tramite una tecnica nota come *data skipping*, che permette di ignorare interi blocchi di dati non rilevanti, riducendo il numero di letture da disco. In secondo luogo, un ordinamento adeguato può migliorare il tasso di compressione: dati con valori simili

o ricorrenti vengono memorizzati consecutivamente, favorendo algoritmi di compressione più efficaci[15].

Segmentare e ordinare dati TimescaleDB, dunque, consente un controllo fine sulla disposizione fisica dei dati nel *column-store*, permettendo di ottimizzare l'efficienza delle query tramite due strategie principali: `SEGMENTBY` e `ORDERBY`.

- **Raggruppamento dei dati correlati (`SEGMENTBY`)[15]:** suddividere i dati in segmenti logici consente di migliorare l'efficienza delle scansioni. Quando le righe sono organizzate in base a valori comuni (ad esempio per ID o categoria), le query che filtrano per uno di questi valori possono limitarsi a leggere solo i segmenti rilevanti, riducendo significativamente le letture da disco.
- **Ordinamento interno ai segmenti (`ORDERBY`)[15]:** ordinare i dati all'interno di ciascun segmento secondo una o più colonne (tipicamente timestamp o altri valori numerici ordinabili) accelera le interrogazioni basate su intervalli. Questo approccio migliora l'efficienza delle *range queries* (query che filtra in base ad un intervallo) e riduce la necessità di ordinamenti post-query, favorendo performance elevate anche su grandi volumi di dati.

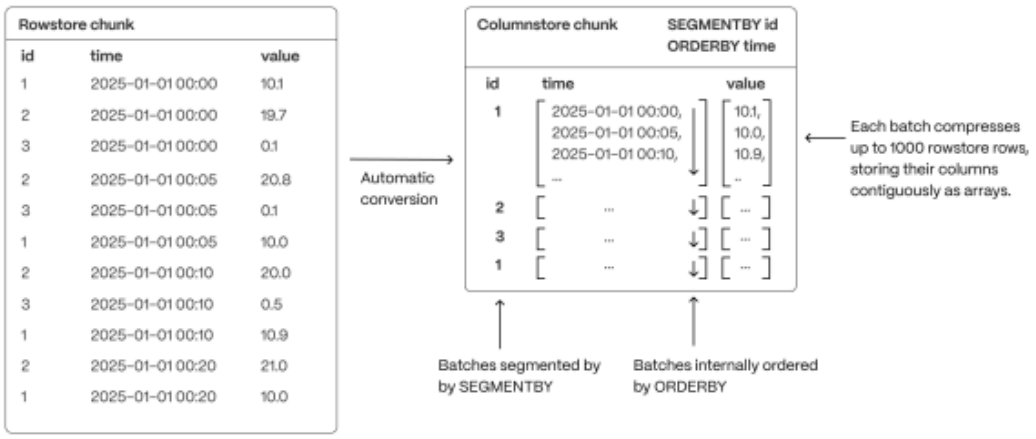

Figura 3.4: Conversione da row-store a column-store con segmentazione[15]

Dunque, combinando segmentazione e ordinamento, TimescaleDB mette a disposizione un layout di storage altamente efficiente, che ottimizza sia le letture che l'esecuzione delle query. In particolare, questa struttura consente l'uso di tecniche avanzate come la *vectorized execution* (basata su SIMD *Single Instruction, Multiple Data*), favorendo l'elaborazione parallela in batch e massimizzando le prestazioni delle analisi in tempo reale.

3.4.2 Data mutability

Nel mondo reale, dove i dati vengono spesso modificati, uno storage totalmente immutabile risulta poco pratico. Anche soluzioni che applicano le modifiche in modo asincrono, rendendole visibili solo dopo un'elaborazione batch, possono causare ritardi incompatibili con i flussi operativi in tempo reale[15].

Hypercore rifiuta una mutabilità in place, in quanto significherebbe: decomprimere, modificare, ricomprimere e reintegrare il dato nel giusto segmento e ordine[15]. Un processo lento e inefficiente, specialmente se gli aggiornamenti sono frequenti.

Le modifiche vengono gestite in modo efficiente attraverso un *rowstore chunk* intermedio, dove gli aggiornamenti e le cancellazioni sono applicati immediatamente, senza dover modificare direttamente i dati compressi nel columnstore[15]. Questo modello consente di mantenere aggiornamenti rapidi e consistenti, preservando al contempo le prestazioni analitiche garantite dallo storage colonnare.

Scritture real-time senza ritardi Le query accedono in modo trasparente sia ai chunk rowstore sia a quelli columnstore, assicurando che le applicazioni possano sempre visualizzare i dati più recenti, indipendentemente dal formato di memorizzazione[15].

Operazioni efficienti senza penalità nelle performance Invece di modificare direttamente lo storage compresso, *hypercore* gestisce gli aggiornamenti decomprimendo i batch interessati e posizionandoli in un *rowstore chunk* intermedio, dove le modifiche vengono applicate immediatamente (Figura 3.5). I batch modificati rimangono temporaneamente in questo storage orientato alle righe fino a quando non vengono ricompattati e reintegrati nel *columnstore*, tramite un processo automatico in background[15]. Questo approccio permette di rendere immediatamente visibili le modifiche, evitando al tempo stesso il costoso overhead legato alla decompressione e riscrittura completa dei chunk.

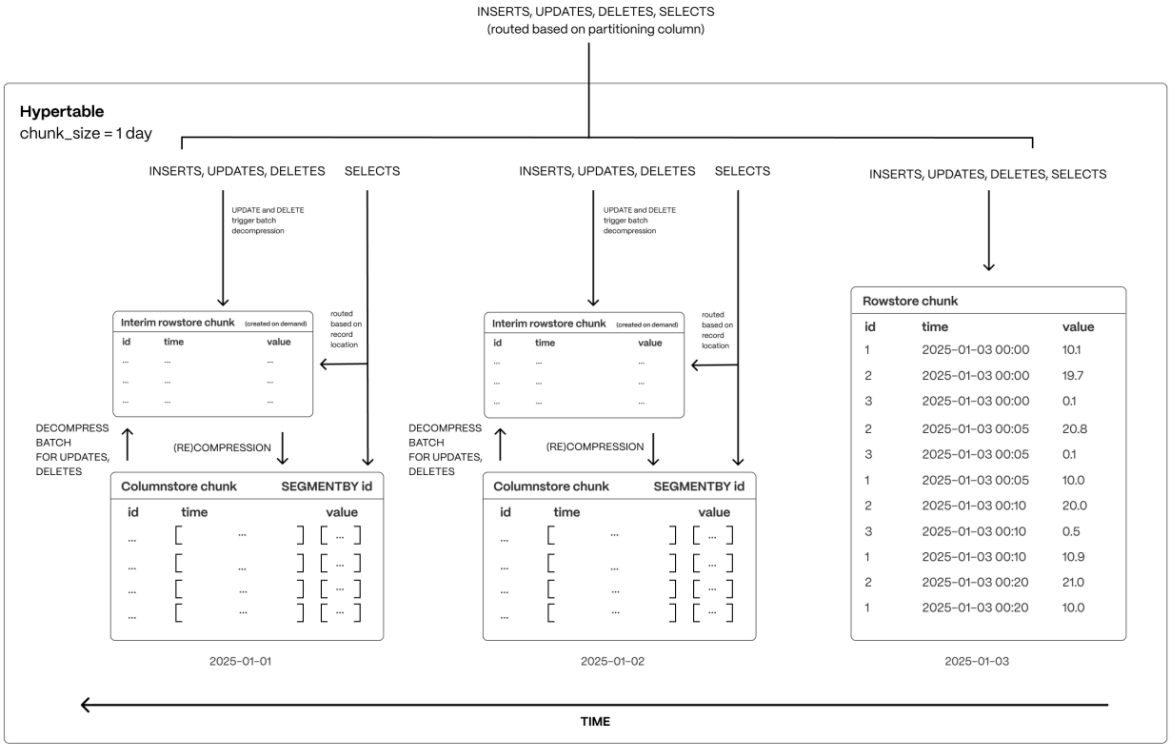

Figura 3.5: Mutabilità dei dati attraverso rowstore e columnstore [15]

3.5 Query optimizations

Timescale ottimizza ogni fase del ciclo di vita di una query per garantire che vengano letti solo i dati realmente necessari, sfruttando la località dei dati ed eseguendo le operazioni in parallelo, così da ottenere risposte in tempi inferiori al secondo anche su dataset di grandi dimensioni [15].

3.5.1 Saltare dati non utili

Timescale riduce al minimo la quantità di dati coinvolti in una query, diminuendo così l'I/O e migliorando le prestazioni [15]. Rivedremo ora alcuni concetti già spiegati, mettendone però in evidenza l'utilità nel contesto pratico.

L'obiettivo si può riassumere così: lavorare solo sui dati davvero rilevanti, e farlo nel modo più efficiente possibile. Questo è reso possibile grazie a:

1. Primary partition exclusion (sia row-store che column-store) Le query saltano automaticamente le partizioni (chunk) irrilevanti basandosi sulla chiave di partizione primaria (tipicamente il timestamp), assicurando di analizzare solo i dati rilevanti [15] (Figura 3.6).

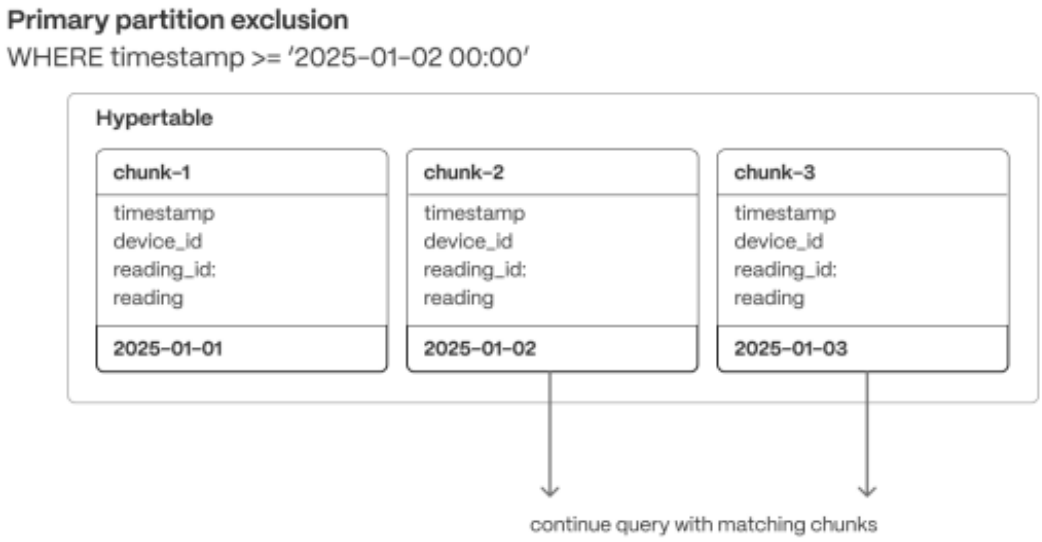

Figura 3.6: Primary partition exclusion [15]

2. Secondary partition exclusion (solo column-store) I metadati min/max permettono alle query che filtrano su dimensioni correlate (ad esempio ID ordine o timestamp secondari) di escludere direttamente i chunk che non contengono dati rilevanti [15].

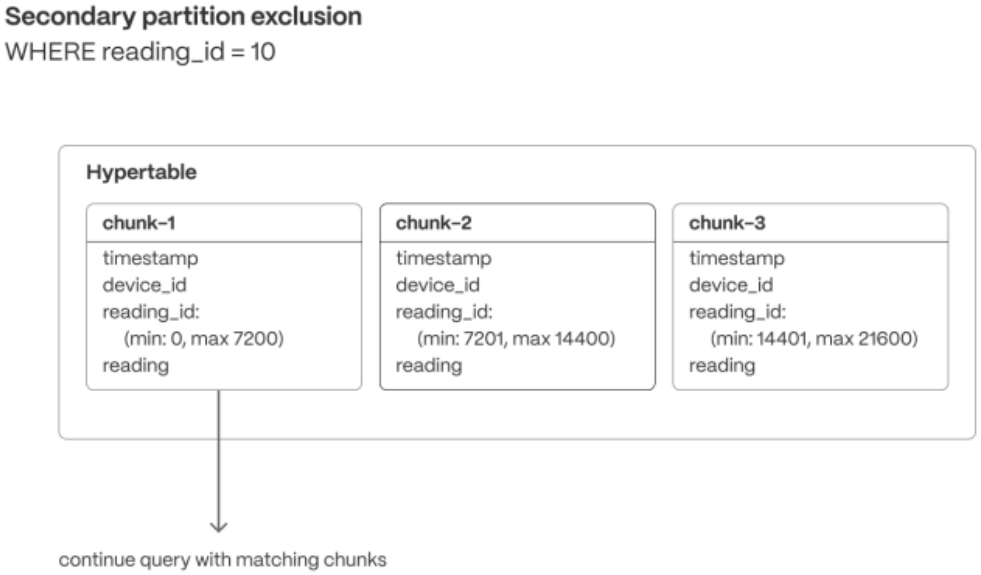

Figura 3.7: Secondary partition exclusion [15]

3. Indici PostgreSQL (sia row-store che column-store) A differenza di molti altri database, Timescale supporta gli indici standard di PostgreSQL anche sui dati in formato column-store (attualmente B-tree e hash, quando si utilizza il metodo di accesso hypercore) [15]. Questo permette alle query di individuare in modo efficiente valori specifici

sia nei dati a riga che in quelli compressi a colonne. Gli indici abilitano ricerche rapide, query su intervalli e operazioni di filtro, riducendo ulteriormente le scansioni inutili dei dati.

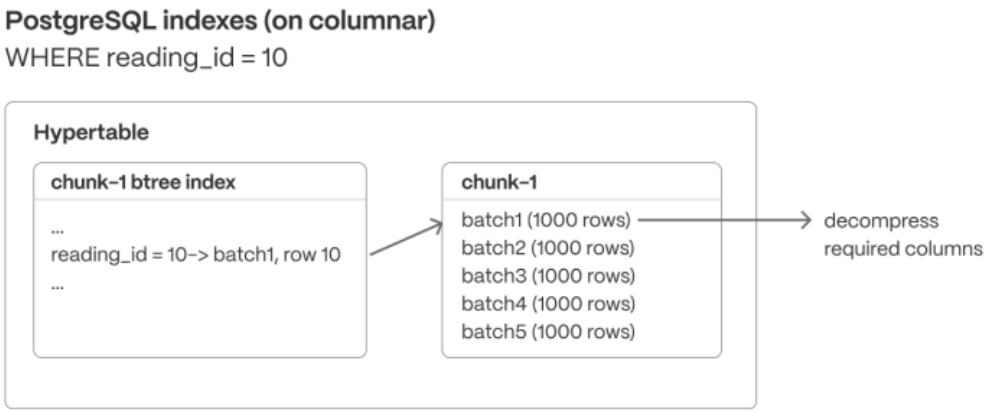

Figura 3.8: Indici PostgreSQL nel column-store[15]

4. Batch-level filtering (solo column-store) In ogni chunk i batch colonnari compressi sono organizzati utilizzando chiavi basate su `SEGMENTBY` e colonne basate su `ORDERBY`. Gli indici e i metadata min/max possono essere usati per escludere rapidamente i batch che non rispondono correttamente ai criteri della query[15] (Figura 3.9).

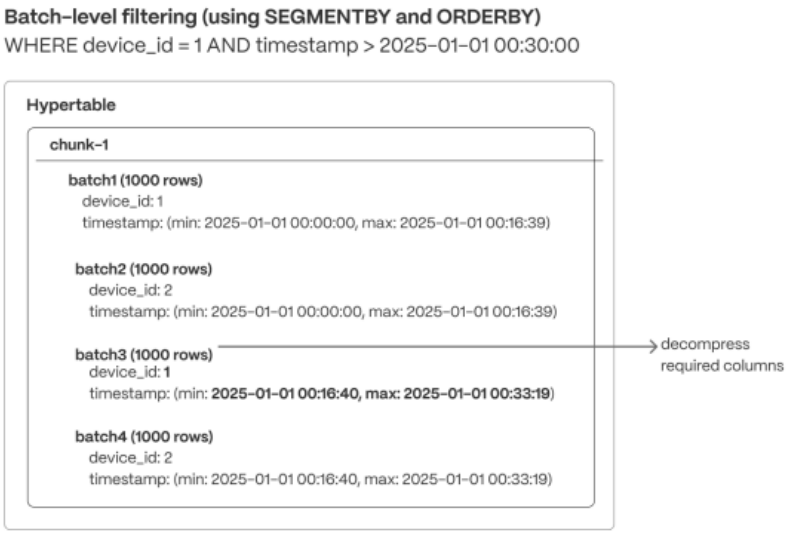

Figura 3.9: Batch skipping indexes[15]

3.5.2 Massimizzare località

Organizzare i dati per un accesso efficiente garantisce che le query vengano eseguite nell'ordine più ottimale, riducendo le letture casuali non necessarie e limitando le scansioni

di dati superflui[15].

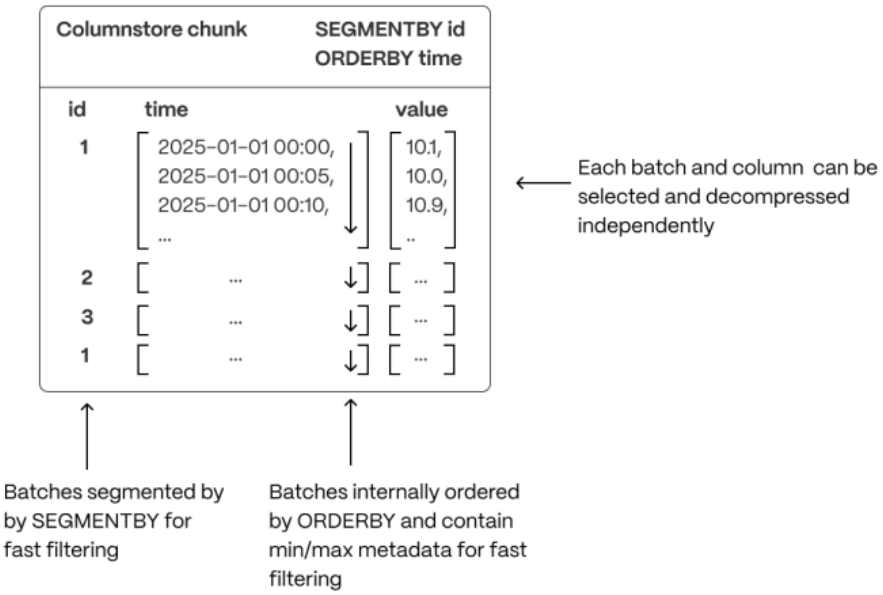

Figura 3.10: Segment by e Order by per località dei dati

- **Segmentazione**[15]: i batch colonnari vengono raggruppati tramite la clausola `SEGMENTBY`, mantenendo vicini i dati correlati e migliorando l'efficienza delle scansioni.
- **Ordinamento**[15]: all'interno di ogni batch, i dati sono fisicamente ordinati usando `ORDERBY`, aumentando l'efficienza delle scansioni (e riducendo le operazioni di I/O), abilitando query per intervalli più rapide e minimizzando la necessità di ordinamenti post-query.
- **Selezione delle colonne**[15]: le query leggono solo le colonne necessarie, riducendo l'I/O su disco, il carico di decompressione e l'utilizzo di memoria.

3.5.3 Esecizione in parallelo

Una volta che la query accede solo ai dati colonnari, nell'ordine più efficiente, Timescale è in grado di massimizzare le prestazioni grazie all'esecuzione parallela. Proprio come avviene con l'utilizzo di più worker, Timescale accelera l'elaborazione delle query nel column-store sfruttando la vettorizzazione SIMD (Single Instruction, Multiple Data), che consente alle moderne CPU di processare più punti dati in parallelo[15].

La vettorizzazione SIMD in Timescale viene applicata in tre fasi principali[15]:

- **Vectorized decompression**: ripristina in modo efficiente i dati compressi in una forma analizzabile.

- **Vectorized filtering**: applica rapidamente le condizioni di filtro su interi insiemi di dati.
- **Vectorized aggregation**: esegue calcoli aggregati, come somme o medie, su più punti dati contemporaneamente.

3.6 Continuous Aggregates

L'aggregazione di grandi dataset in tempo reale può risultare molto onerosa, poiché richiede scansioni ripetute e calcoli intensivi che gravano su CPU e I/O. Alcuni database tentano di gestire queste operazioni forzando l'esecuzione delle query al momento del bisogno, ma le risorse computazionali e di input/output restano limitate, causando così alta latenza, prestazioni instabili e costi infrastrutturali sempre maggiori con l'aumentare dei dati[15].

Le **continuous aggregates**, ovvero l'implementazione di Timescale delle viste materializzate aggiornate in modo incrementale, risolvono questo problema spostando il carico di calcolo dalla fase di esecuzione della query a un processo asincrono che avviene dopo l'ingestione dei dati. In pratica, solo gli intervalli temporali (time buckets) che hanno ricevuto nuovi dati o modifiche vengono ricalcolati. Le query successive non devono più analizzare i dati grezzi, ma leggono direttamente risultati già precomputati, con un notevole miglioramento in termini di prestazioni ed efficienza[15].

Le continuous aggregates in Timescale sono definite con metriche (o aggregazioni) preimpostate. Quando si crea una continuous aggregate, si specifica quali metriche calcolare (es. AVG, SUM, COUNT, MAX, ecc.) e su quali colonne e intervalli di tempo[15].

Code Listing 3.1: Esempio di continuous aggregate dalla documentazione Timescale

```
CREATE MATERIALIZED VIEW conditions_summary_daily
WITH (timescaledb.continuous) AS
SELECT device,
   time_bucket(INTERVAL '1 day', time) AS bucket,
   AVG(temperature),
   MAX(temperature),
   MIN(temperature)
FROM conditions
GROUP BY device, bucket;
```

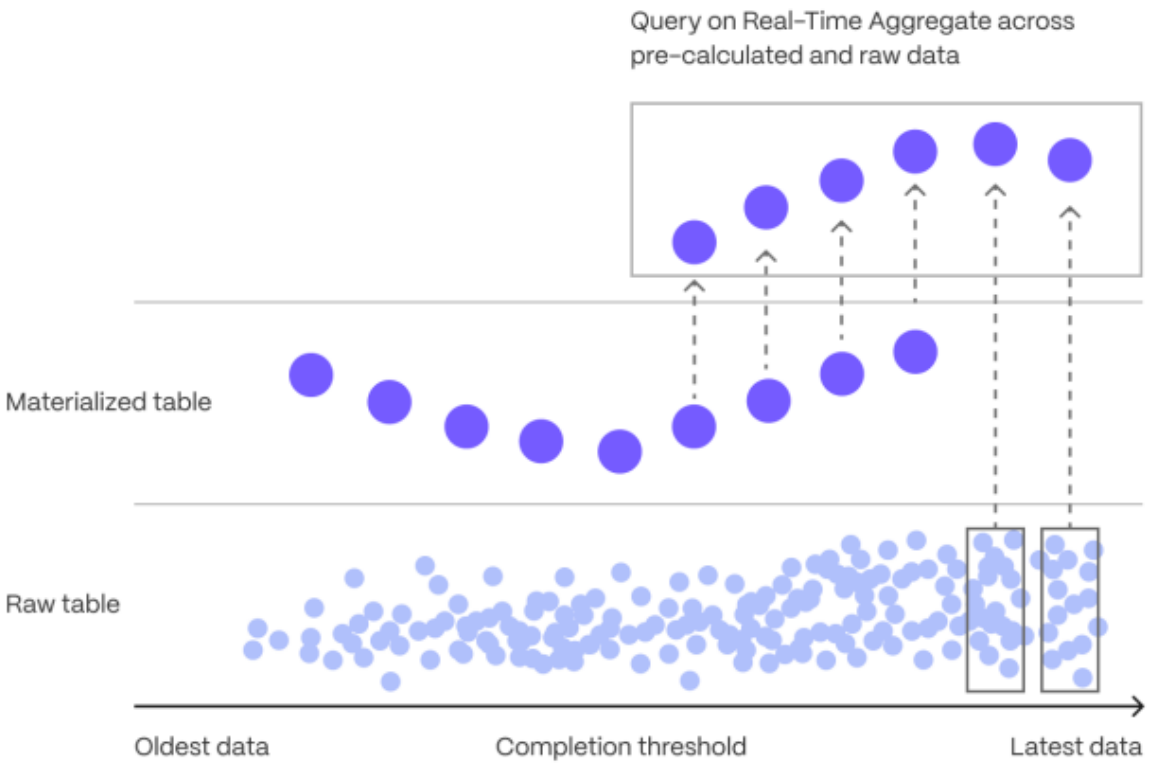

Figura 3.11: Selezione dei dati per continuous aggregate [15]

Questa tecnica consente di ottenere risultati istantanei senza la necessità di eseguire calcoli sul momento durante l'esecuzione delle query, migliorando notevolmente le performance, in particolare quando si conoscono in anticipo i tipi di query che si intende eseguire [15]. I rollup di un database classico fanno esattamente questo, ma timescale aggiunge l'automatismo.

Dunque, un ulteriore vantaggio delle continuous aggregates è che evitano il lungo e complesso processo di mantenimento manuale dei rollup, riducendo così il rischio di errori e migliorando l'affidabilità del sistema. Nonostante l'automazione del processo, viene comunque garantita la mutabilità dei dati, che consente di effettuare aggiornamenti, correzioni e completamenti retroattivi [15].

Ogni volta che vengono inseriti o modificati nuovi dati in chunks già materializzati, Timescale registra delle invalidazioni, queste segnalano che i risultati precedenti sono obsoleti e necessitano di essere ricalcolati. Successivamente, un processo asincrono si occupa di rielaborare le aree contenenti i dati invalidati e aggiornare i risultati materializzati [15].

Timescale traccia in modo preciso la *lineage* (tracciamento della storia del dato), le dipendenze tra le *continuous aggregates*, e i dati sottostanti, assicurando che le aggregazioni

vengano mantenute costantemente aggiornate.

Questo processo avviene in maniera efficiente, in quanto le invalidazioni multiple possono essere aggregate in un unico aggiornamento, evitando l'overhead associato all'aggiornamento delle dipendenze al momento della scrittura, come avverrebbe con un approccio basato su trigger[15]. In questo modo, si ottimizzano le risorse e si riducono i costi operativi, mantenendo nel contempo una gestione efficiente e affidabile dei dati.

Le continuous aggregates sono memorizzate all'interno delle hypertables e possono essere convertite in storage colonnare per la compressione. Inoltre, i dati grezzi possono essere eliminati, riducendo così l'impatto sulla memoria e i costi di elaborazione. Le continuous aggregates supportano anche i rollup gerarchici (ad esempio, da orari a giornalieri, a mensili) e la modalità in tempo reale, che unisce i risultati pre-calcolati con i dati appena inseriti per garantire analisi precise e aggiornate[15].

Questa architettura consente analisi scalabili e a bassa latenza, mantenendo l'utilizzo delle risorse prevedibile, rendendola ideale per dashboard, sistemi di monitoraggio e qualsiasi carico di lavoro con schemi di query noti.

3.6.1 Hyperfunctions per l'analisi in tempo reale

L'analisi in tempo reale delle serie temporali richiede un livello di computazione più avanzato rispetto alle funzioni SQL di base. Man mano che la dimensione e la complessità dei dataset aumentano, diventa fondamentale disporre di metodi di calcolo efficienti per ottenere risposte rapide e accurate[15].

In questo contesto, Timescale offre le **hyperfunctions**, funzioni avanzate e ad alte prestazioni integrate nativamente in SQL, pensate appositamente per l'analisi delle serie temporali. Le hyperfunctions includono strumenti per il *gap-filling*, la *stima dei percentili*, la *media ponderata nel tempo*, la *correzione dei contatori* e il *monitoraggio dello stato*, che permettono di affrontare in modo efficiente problemi complessi legati ai dati temporali[15].

Una delle innovazioni più importanti delle hyperfunctions è il supporto per l'**aggregazione parziale**, che consente a Timescale di memorizzare stati computazionali intermedi piuttosto che solo i risultati finali. Questo approccio permette di evitare costose rielaborazioni dei dati grezzi e riduce significativamente il carico computazionale. Gli stati parziali calcolati possono essere successivamente combinati per ottenere i rollup, migliorando l'efficienza nell'elaborazione delle query senza dover rifare tutto il processo da zero[15].

L'integrazione delle **hyperfunctions** con le **continuous aggregates** offre un potente strumento per l'analisi in tempo reale, permettendo l'elaborazione rapida di grandi volumi di dati senza compromettere la precisione o l'affidabilità dei risultati. Questo approccio non solo garantisce analisi scalabili e a bassa latenza, ma consente anche un utilizzo delle risorse più prevedibile ed efficiente, ideale per applicazioni come dashboard, sistemi di monitoraggio e qualsiasi carico di lavoro con schemi di query ben definiti.

3.7 Ottimizzazioni nelle performance di TimescaleDB

Come spiegato in "*8 performance improvements in recent timescaledb releases for faster query analytics*"[10], negli ultimi aggiornamenti, TimescaleDB ha introdotto una serie di ottimizzazioni per migliorare le prestazioni delle query, specialmente in presenza di hypertables partizionate e dati compressi. Di seguito, si evidenziano le migliorie più interessanti:

3.7.1 Ottimizzazione tramite aggregazioni parziali

Per migliorare le prestazioni delle query di aggregazione, è stata introdotta una strategia basata sull'uso di **aggregazioni parziali a livello di chunk**. Tradizionalmente, le query di aggregazione (es. `SUM`, `AVG`, `COUNT`) venivano eseguite esclusivamente nel nodo "root" del piano di esecuzione, ossia nella fase finale del processo. Questo significava che tutti i dati grezzi provenienti da ciascun chunk dovevano essere letti e aggregati globalmente, comportando un notevole carico computazionale e possibili colli di bottiglia.

Con la nuova ottimizzazione, l'esecuzione delle aggregazioni è stata distribuita in due fasi:

1. Ogni **chunk** esegue localmente una **aggregazione parziale** sui propri dati.
2. Gli aggregati parziali (ad esempio, la somma e il numero di elementi per calcolare la media) vengono poi combinati nel nodo root per produrre l'aggregato finale.

Questo approccio consente di lavorare su input significativamente più piccoli in ciascun chunk, riducendo il volume di dati che devono essere trasferiti e processati globalmente.

Esempio: Supponiamo di voler calcolare la media dei valori di temperatura. Invece di sommare tutti i valori a livello globale, ogni chunk calcola la propria somma e il proprio conteggio:

- Chunk 1: somma = 100, count = 10

- Chunk 2: somma = 200, count = 20

Il nodo root combina poi questi risultati:

$$\text{media} = \frac{100 + 200}{10 + 20} = \frac{300}{30} = 10$$

Grazie a questo schema, è stato osservato un miglioramento delle performance di circa **15%** nelle query di aggregazione.

3.7.2 Ottimizzazione del sorting dei mini-batch columnar

Durante la compressione dei dati, TimescaleDB divide i dati in piccoli gruppi chiamati *mini-batch* in formato *columnar*, ovvero memorizzando i valori colonna per colonna anziché riga per riga. Prima della compressione, ogni mini-batch viene ordinato secondo una o più colonne di riferimento, tipicamente la colonna temporale.

Precedentemente, quando più mini-batch compressi dovevano essere uniti per rispondere a una query, il sistema spesso eseguiva nuovamente un ordinamento globale, poiché l'ordine tra i batch compressi non era garantito. Questo passaggio comportava un costo computazionale non trascurabile.

L'ottimizzazione introdotta consiste nel **riutilizzare l'ordinamento eseguito a livello di mini-batch** durante la fase di unione dei dati compressi in chunk, evitando così l'ordinamento ridondante.

Questa strategia è particolarmente vantaggiosa per query che limitano il numero di tuple restituite, come quelle che utilizzano `LIMIT`, oppure funzioni come `first()` e `last()`, poiché:

- il sistema può sfruttare l'ordinamento già presente per estrarre rapidamente le righe richieste;
- si evita il costo di un ordinamento globale sui dati decompressi;
- è possibile effettuare un'uscita anticipata (early exit) non appena si ottengono i risultati desiderati.

Di conseguenza, questa ottimizzazione contribuisce a migliorare significativamente le prestazioni delle query che richiedono un sottoinsieme limitato di dati ordinati.

3.7.3 Ottimizzazione della Chunk Exclusion

Una delle migliorie più significative introdotte da TimescaleDB riguarda l'efficienza nell'elaborazione delle query `LIMIT ORDER BY`, in particolare nei casi in cui i chunk siano **parzialmente compressi** o si stiano utilizzando **hypertable space-partitioned**.

Nel contesto delle serie temporali, TimescaleDB suddivide i dati in blocchi fisici chiamati *chunk*. Quando si esegue una query, il sistema cerca di escludere i chunk che non contengono dati rilevanti per ridurre il carico computazionale: questo processo è noto come **chunk exclusion** (o *constraint exclusion*).

In passato, le query che utilizzavano `ORDER BY` in combinazione con `LIMIT` erano meno efficienti, in quanto richiedevano il caricamento e l'ordinamento di un numero maggiore di chunk prima di applicare il limite. Con l'ottimizzazione introdotta, TimescaleDB è ora in grado di:

- identificare rapidamente i chunk che contengono i dati più rilevanti (es. i più recenti);
- **escludere chunk compressi o non necessari** in modo più preciso;
- **interrompere la scansione non appena viene raggiunto il limite** della query.

Questa nuova logica ha portato a un **incremento delle performance fino a 20 volte** in query complesse su dati compressi e partizionati, rendendo TimescaleDB ancora più adatto per scenari real-time ad alta frequenza.

3.8 Ulteriori ottimizzazioni rispetto a PostgreSQL

Ulteriori peculiarità, rispetto al solo PostgreSQL, sono evidenziate nell'articolo "*PostgreSQL + TimescaleDB: 1,000x Faster Queries, 90 % Data Compression, and Much More*" [6]. Di seguito ne vediamo gli aspetti di maggior rilievo.

3.8.1 Funzionalità per accelerare i tempi di sviluppo

TimescaleDB, come accennato, include una libreria di oltre 100 hyperfunction che semplificano l'analisi complessa dei dati time-series mediante SQL, quali medie ponderate nel tempo e funzioni specifiche come `time_bucket()`, `time_bucket_gapfill()` e altre. Inoltre, è presente un motore integrato per lo scheduling dei job, che permette di pianificare l'esecuzione di procedure personalizzate, fornendo funzionalità simili a scheduler esterni come `pg_cron`, ma senza richiedere l'installazione e la manutenzione di estensioni multiple.

Questo risulta essere senza ombra di dubbio un enorme vantaggio per gli sviluppatori.

3.8.2 Prestazioni di ingestione

Le prestazioni di ingestione dei dati tra TimescaleDB e PostgreSQL risultano molto simili, anche se TimescaleDB mostra un tasso medio di ingestione superiore di circa 3.000-4.000 righe al secondo rispetto a una singola tabella PostgreSQL. La dimensione del batch di inserimento incide in modo analogo su entrambi i sistemi.

3.8.3 Confronto con il partizionamento dichiarativo di PostgreSQL

Il confronto con il partizionamento dichiarativo nativo di PostgreSQL dimostra come, nonostante i miglioramenti di quest'ultimo, TimescaleDB sia ancora fino a 1.000 volte più veloce nell'esecuzione di alcune query, con guadagni significativi in termini di prestazioni complessive. Un aspetto fondamentale riguarda la semplicità d'uso: mentre il partizionamento dichiarativo richiede di specificare manualmente la logica delle partizioni e la loro gestione, TimescaleDB automatizza completamente tali operazioni tramite un unico comando: `create_hypertable`.

Partizionamento dichiarativo in PostgreSQL

Il *partizionamento dichiarativo* (in inglese *declarative partitioning*) è una funzionalità introdotta in PostgreSQL a partire dalla versione 10 [1] [1], che consente di suddividere una tabella molto grande in più parti più piccole, dette *partizioni*, in modo semplice e nativo, senza la necessità di gestire manualmente la creazione e la manutenzione di tabelle secondarie o l'uso di trigger complessi.

Il funzionamento si basa su:

- **Dichiarazione esplicita:** durante la creazione della tabella si definisce una tabella *genitore* che funge da contenitore e si specifica una strategia di partizionamento (ad esempio *RANGE*, *LIST* o *HASH*).
- **Gestione automatica da parte di PostgreSQL:** il sistema crea automaticamente le tabelle partizionate figlie (partizioni), gestisce l'instradamento delle inserzioni, le query e la rimozione dei dati.

Questi risultati sottolineano come TimescaleDB rappresenti una soluzione avanzata per la gestione e l'analisi di dati temporali, facilitando sia lo sviluppo sia le performance applicative.

Capitolo 4

TimescaleDB: analisi delle performance

4.1 Setup dell'ambiente di test

Per il confronto delle performance tra **TimescaleDB** e **PostgreSQL** è stato predisposto un ambiente controllato e quanto più pulito possibile, privo di processi terzi attivi e con pulizia della cache prima dell'esecuzione.

Sono state dunque eseguite misurazioni ripetute su un insieme di query rappresentative. Nello specifico, l'ambiente di test è stato eseguito su macchina locale, con la seguente configurazione:

- `Versioni`: PostgreSQL 17.5 e TimescaleDB 2.19.3
- `OS`: Fedora 42
- `CPU`: Intel Core i7 10750H 2.60 GHz (6 cores, 12 threads)
- `RAM`: 16GB DDR4
- `Disco`: SSD NVMe WD Blue SN570 1TB
- `Parametri DB`: configurazioni di default per entrambe le installazioni
- `Data del test`: Giugno 2025

Nel test presentato, l'esecuzione è stata automatizzata tramite uno script Python (si veda appendice A.3).

Tale script legge due file di configurazione, `input.txt` e `config.ini`, che specificano rispettivamente i parametri delle query da eseguire e la configurazione dell'ambiente di test (si veda appendice A.4 e A.5).

Il processo prosegue con la creazione di due tabelle (come descritto nei paragrafi successivi), il loro popolamento, l'esecuzione delle query e la produzione dei risultati in formato testuale e grafico.

Il file `input.txt` contiene:

- un intervallo numerico `[0, 30]` utilizzato per la generazione dei dati;
- una lista di dimensioni (numero di record) da testare;
- una serie di query SQL da eseguire.

Il file `config.ini`, strutturato secondo il formato INI, definisce:

- i parametri di connessione al database;
- i nomi delle tabelle da creare e interrogare;
- i parametri per la generazione del dataset;
- le impostazioni per la ripetizione dei test e il calcolo degli intervalli di confidenza.

L'analisi si svolge utilizzando due tabelle identiche nella struttura ma differenti nella gestione:

- `sensor_data_normal`: tabella PostgreSQL tradizionale.
- `sensor_data_hypertable`: tabella gestita come hypertable tramite TimescaleDB.

Entrambe le tabelle contengono una colonna temporale `time TIMESTAMPTZ` e una colonna numerica `value DOUBLE PRECISION`.

I dati sono stati generati tramite lo script Python, variando il numero totale di record e mantenendo un intervallo temporale costante (ogni 5 minuti), a partire da una data iniziale.

4.2 Criteri e metodologia di test

Le misurazioni sono state eseguite secondo i seguenti criteri:

- Ogni query è stata eseguita **100 volte**, calcolando il tempo medio di esecuzione.
- È stato calcolato l'**intervallo di confidenza al 95%**.
- I dataset usati sono di dimensione crescente: **1.000**, **10.000**, **100.000**, **1.000.000** e **10.000.000** record.
- I dati sono stati generati con valori casuali in virgola mobile all'interno di un intervallo prestabilito.

Le query utilizzate riflettono scenari comuni su serie temporali:

- **Q1 - Full Scan:**
  ```
  SELECT *
  FROM sensor_data;
  ```
  Restituisce tutti i record presenti nella tabella, senza filtri o condizioni. Utile per misurare il costo di una scansione completa.
- **Q2 - Range Filter:**
  ```
  SELECT *
  FROM sensor_data
  WHERE time BETWEEN '2025-01-01 00:00:00' AND '2025-01-07 00:00:00';
  ```
  Estrae solo i dati relativi a un intervallo temporale definito. Rappresenta una query temporale di selezione tipica.
- **Q3 - Future Average:**
  ```
  SELECT AVG(value) AS avg_value
  FROM sensor_data
  WHERE time >= '2028-12-01 00:00:00';
  ```
  Calcola la media dei valori a partire da una certa data futura. Utile per testare la gestione di dati scarsamente popolati.
- **Q4 - Daily Average:**
  ```
  SELECT time::date AS day, AVG(value) AS avg_value
  FROM sensor_data WHERE time BETWEEN '2025-01-01' AND '2025-01-07'
  GROUP BY day
  ORDER BY day;
  ```
  Raggruppa i dati per giorno e calcola la media giornaliera, utile per analisi statistiche aggregate nel tempo.

- **Q5 - Rolling Window Avg:**
  ```
  SELECT time, AVG(value)
  OVER (ORDER BY time ROWS BETWEEN 11 PRECEDING AND CURRENT ROW)
  AS rolling_avg
  FROM sensor_data
  WHERE time BETWEEN '2025-01-05' AND '2025-01-08';
  ```
  Calcola una media mobile (rolling average) su una finestra di 12 righe temporali. È una query più avanzata tipica di analisi su serie temporali.

4.3 Risultati dei test con TimescaleDB

Di seguito sono riportati i risultati dei test prestazionali, inizialmente per ciascun volume di dati (da 1.000 a 10.000.000 di record), e successivamente attraverso rappresentazioni grafiche su scala logaritmica per facilitare il confronto visivo tra scenari di diversa scala.

Le **tabelle** illustrano i tempi medi di esecuzione (in secondi) delle diverse query, con relativo intervallo di confidenza, confrontando le prestazioni di una *hypertable* in TimescaleDB con quelle di una tabella tradizionale in PostgreSQL. La colonna "Vincitore" indica quale delle due soluzioni ha mostrato le migliori performance per ciascuna query.

I **grafici a barre** "*Query Times for sensor_data_hypertable (# records)*" e "*Query Times for sensor_data_normal (# records)*" visualizzano questi stessi risultati separatamente per le due tipologie di tabella, rispettivamente:

- la prima, mostra le performance della hypertable. L'asse delle ascisse (x) rappresenta le diverse query eseguite (Q1 - Q5), mentre l'asse delle ordinate (y) indica il tempo medio di esecuzione, comprensivo di barre d'errore.
- la seconda, riporta le stesse informazioni per la tabella tradizionale, consentendo un confronto diretto.

Questa rappresentazione consente di visualizzare chiaramente l'incertezza associata alle misurazioni, evidenziando il ruolo dell'intervallo di confidenza nell'interpretazione dei risultati.

Nei **grafici su scala logaritmica**, l'asse delle ascisse (x) rappresenta il numero di record analizzati (su scala logaritmica), mentre l'asse delle ordinate (y) indica il tempo medio di esecuzione delle query (anch'esso su scala logaritmica), permettendo di osservare l'andamento delle performance al variare del volume dei dati.

4.3.1 Risultati: performance su 1.000 record

Tabella 4.1: Performance su 1.000 record

Query	Hypertable (TSDB)	Normal Table (PGSQL)	Vincitore
Q1 - Full Scan	0.0005 ± 0.0001	0.0004 ± 0.0000	Pareggio
Q2 - Range Filter	0.0008 ± 0.0001	0.0004 ± 0.0000	Pareggio
Q3 - Future Average	0.0001 ± 0.0000	0.0001 ± 0.0000	Pareggio
Q4 - Daily Average	0.0006 ± 0.0001	0.0003 ± 0.0000	Pareggio
Q5 - Rolling Window Avg	0.0002 ± 0.0000	0.0001 ± 0.0000	Pareggio

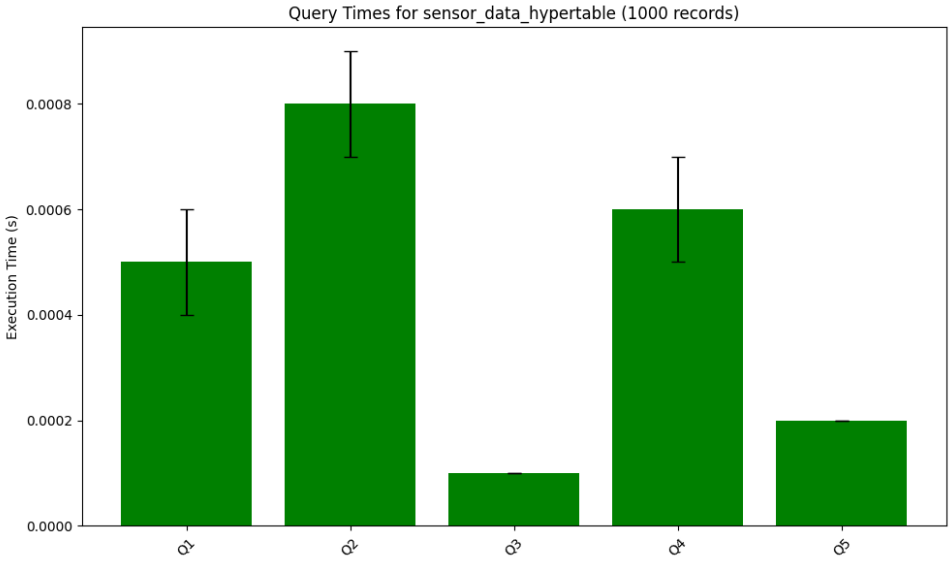

Figura 4.1: Performance hypertable su 1.000 record

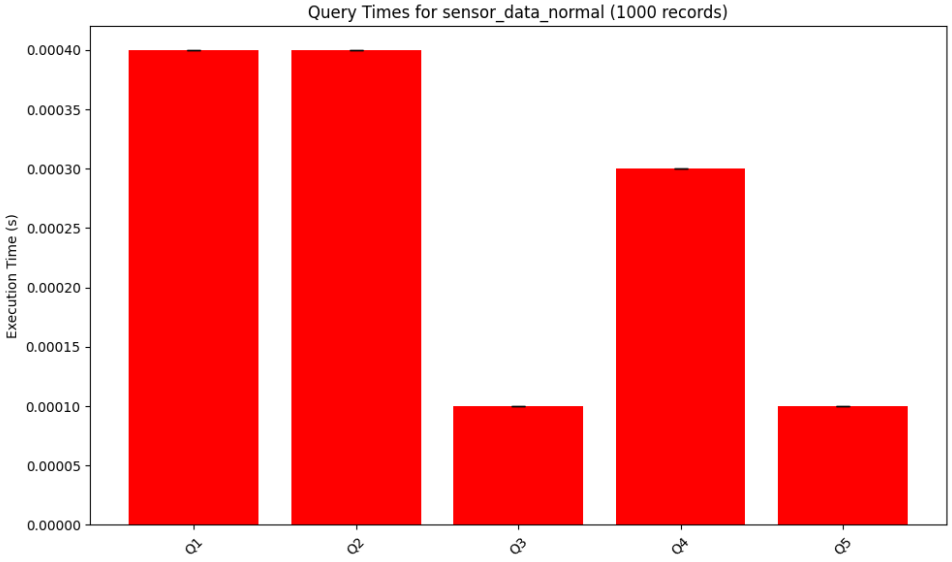

Figura 4.2: Performance hypertable su 1.000 record

4.3.2 Risultati: performance su 10.000 record

Tabella 4.2: Performance su 10.000 record

Query	**Hypertable (TSDB)**	**Normal Table (PGSQL)**	**Vincitore**
Q1 - Full Scan	0.0031 ± 0.0001	0.0029 ± 0.0001	PostgreSQL
Q2 - Range Filter	0.0007 ± 0.0000	0.0009 ± 0.0000	TimescaleDB
Q3 - Future Average	0.0001 ± 0.0000	0.0005 ± 0.0000	TimescaleDB
Q4 - Daily Average	0.0006 ± 0.0000	0.0008 ± 0.0000	TimescaleDB
Q5 - Rolling Window Avg	0.0013 ± 0.0000	0.0016 ± 0.0000	TimescaleDB

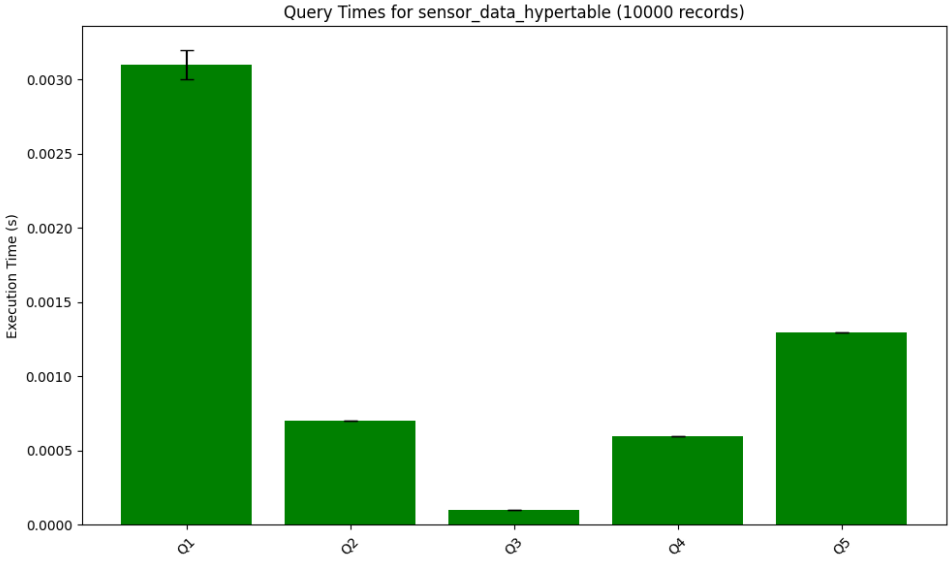

Figura 4.3: Performance hypertable su 10.000 record

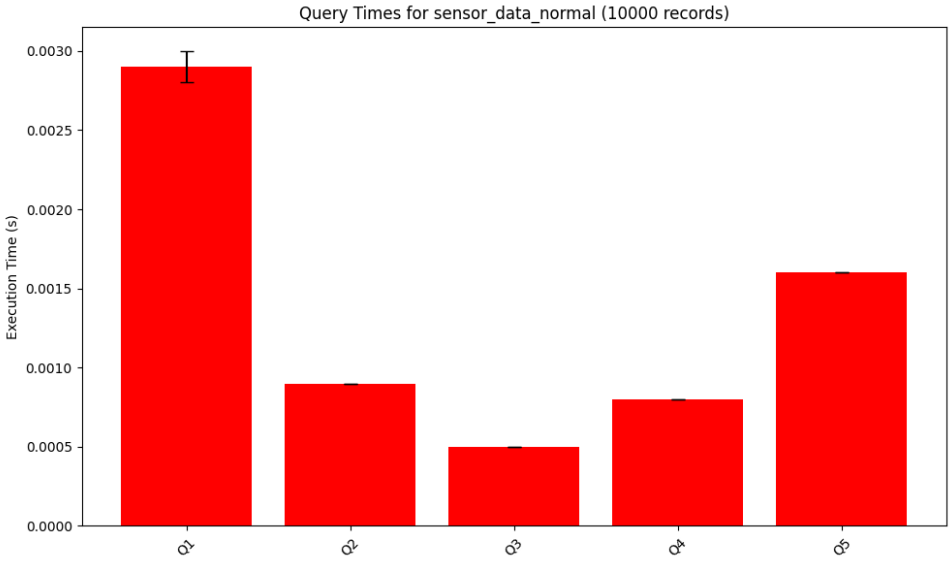

Figura 4.4: Performance hypertable su 10.000 record

4.3.3 Risultati: performance su 100.000 record

Tabella 4.3: Performance su 100.000 record

Query	Hypertable (TSDB)	Normal Table (PGSQL)	Vincitore
Q1 - Full Scan	0.0291 ± 0.0003	0.0275 ± 0.0003	PostgreSQL
Q2 - Range Filter	0.0008 ± 0.0000	0.0046 ± 0.0001	TimescaleDB
Q3 - Future Average	0.0001 ± 0.0000	0.0040 ± 0.0001	TimescaleDB
Q4 - Daily Average	0.0006 ± 0.0000	0.0045 ± 0.0001	TimescaleDB
Q5 - Rolling Window Avg	0.0013 ± 0.0000	0.0053 ± 0.0001	TimescaleDB

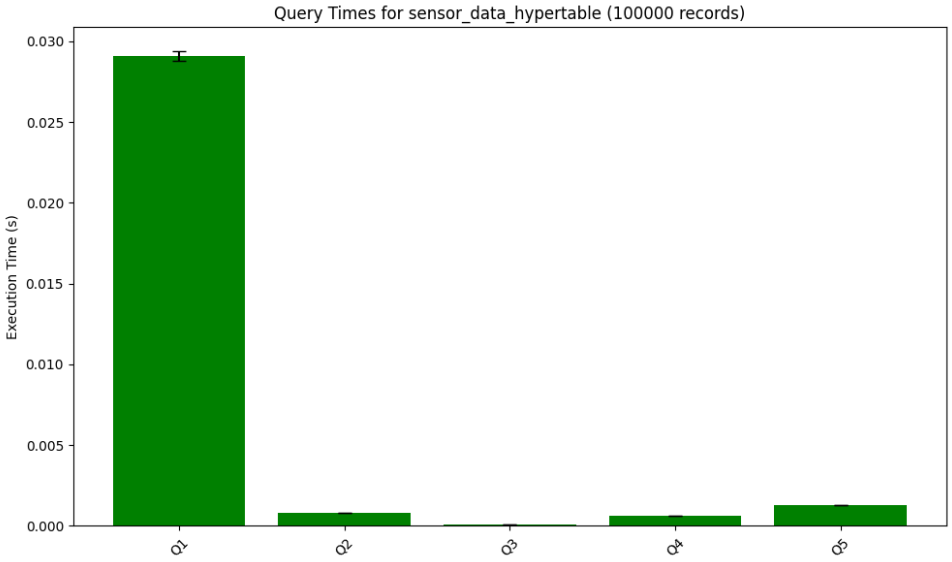

Figura 4.5: Performance hypertable su 100.000 record

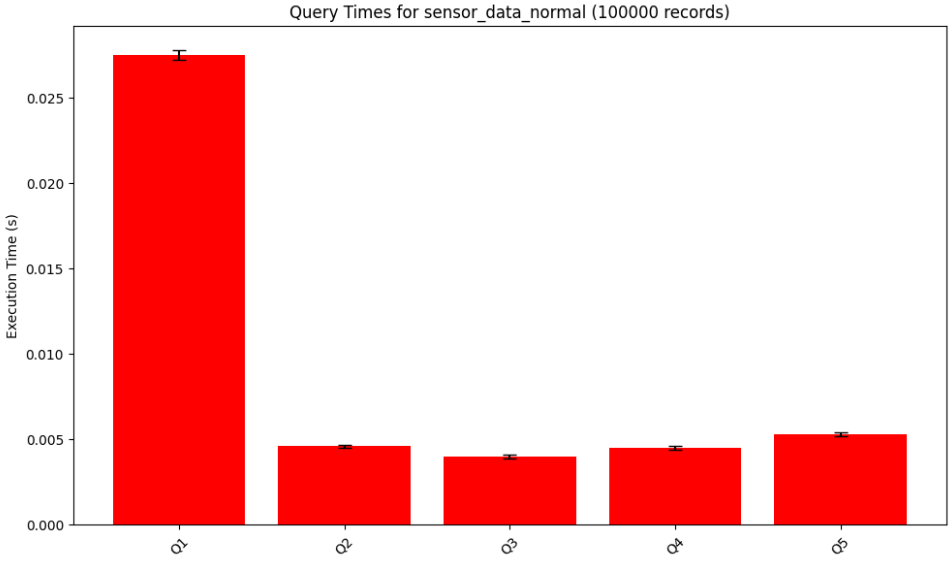

Figura 4.6: Performance hypertable su 100.000 record

4.3.4 Risultati: performance su 1.000.000 record

Tabella 4.4: Performance su 1.000.000 record

Query	Hypertable (TSDB)	Normal Table (PGSQL)	Vincitore
Q1 - Full Scan	0.289 ± 0.005	0.277 ± 0.005	PostgreSQL
Q2 - Range Filter	0.0009 ± 0.0004	0.0191 ± 0.0008	TimescaleDB
Q3 - Future Average	0.0369 ± 0.0005	0.0235 ± 0.0001	PostgreSQL
Q4 - Daily Average	0.0006 ± 0.0000	0.0187 ± 0.0001	TimescaleDB
Q5 - Rolling Window Avg	0.0013 ± 0.0000	0.0187 ± 0.0001	TimescaleDB

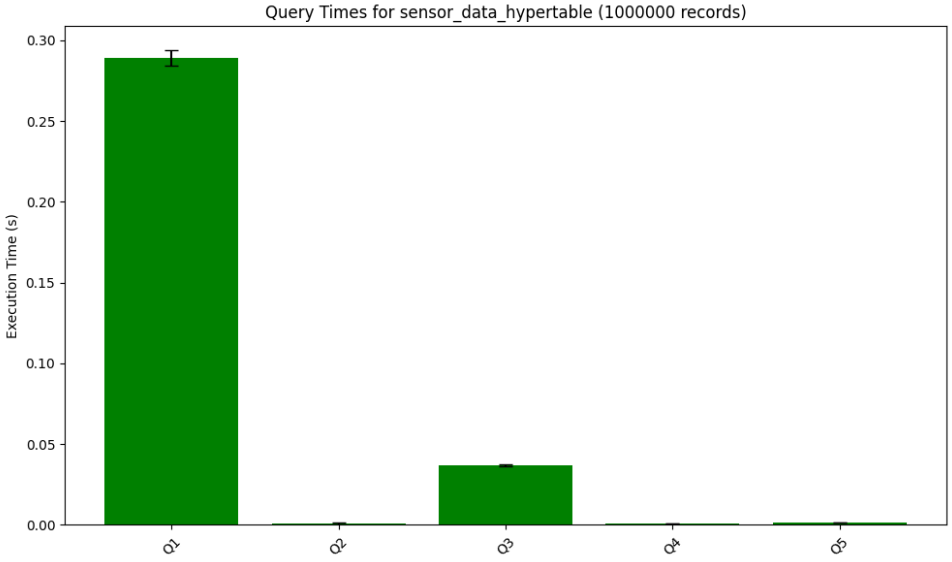

Figura 4.7: Performance hypertable su 1.000.000 record

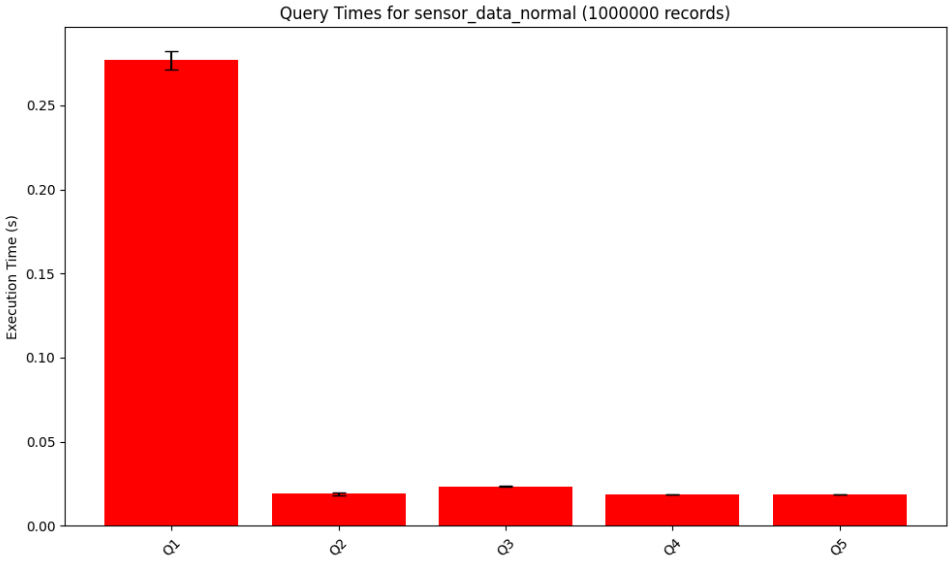

Figura 4.8: Performance hypertable su 1.000.000 record

4.3.5 Risultati: performance su 10.000.000 record

Tabella 4.5: Performance su 10.000.000 record

Query	Hypertable (TSDB)	Normal Table (PGSQL)	Vincitore
Q1 - Full Scan	2.9288 ± 0.0503	2.7194 ± 0.0424	PostgreSQL
Q2 - Range Filter	0.0023 ± 0.0030	0.1007 ± 0.0031	TimescaleDB
Q3 - Future Average	0.7036 ± 0.0231	0.1637 ± 0.0018	PostgreSQL
Q4 - Daily Average	0.0006 ± 0.0000	0.1028 ± 0.0004	TimescaleDB
Q5 - Rolling Window Avg	0.0013 ± 0.0000	0.1038 ± 0.0003	TimescaleDB

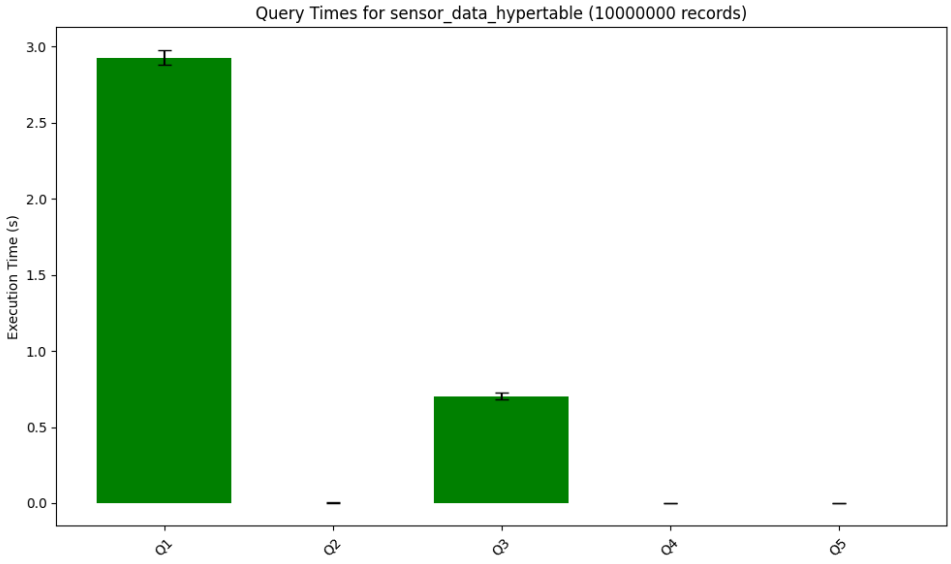

Figura 4.9: Performance hypertable su 10.000.000 record

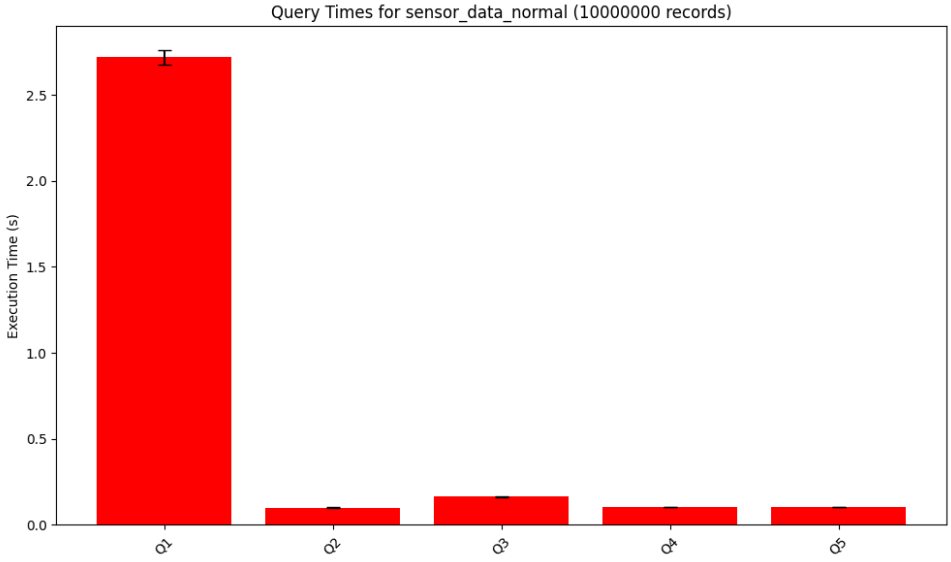

Figura 4.10: Performance hypertable su 10.000.000 record

4.3.6 Risultati: grafici su scala logaritmica

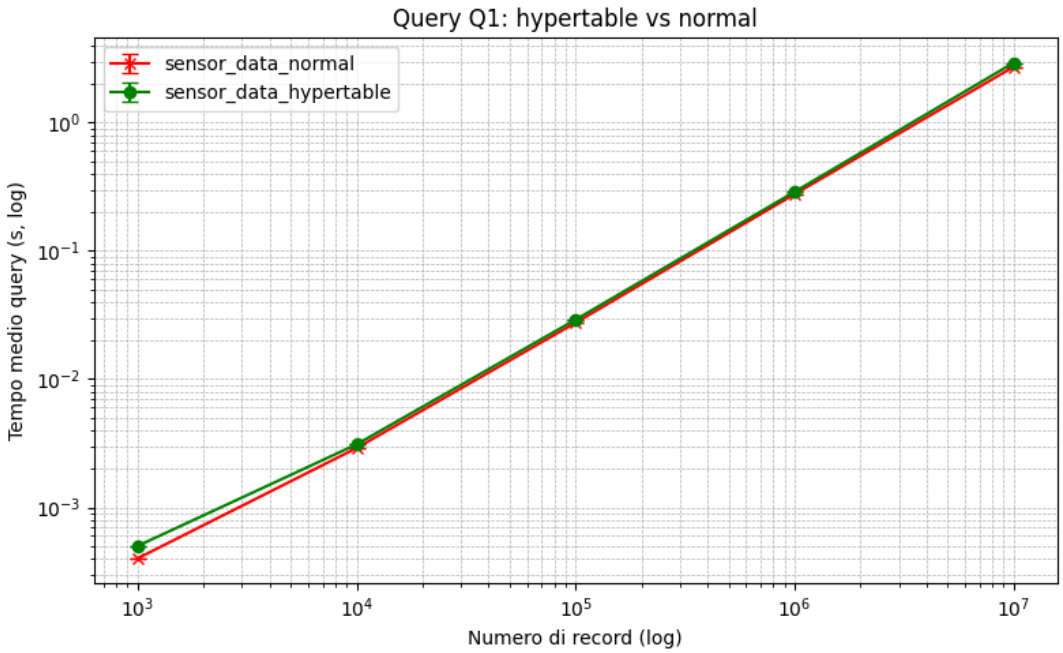

Figura 4.11: Q1 - Full Scan

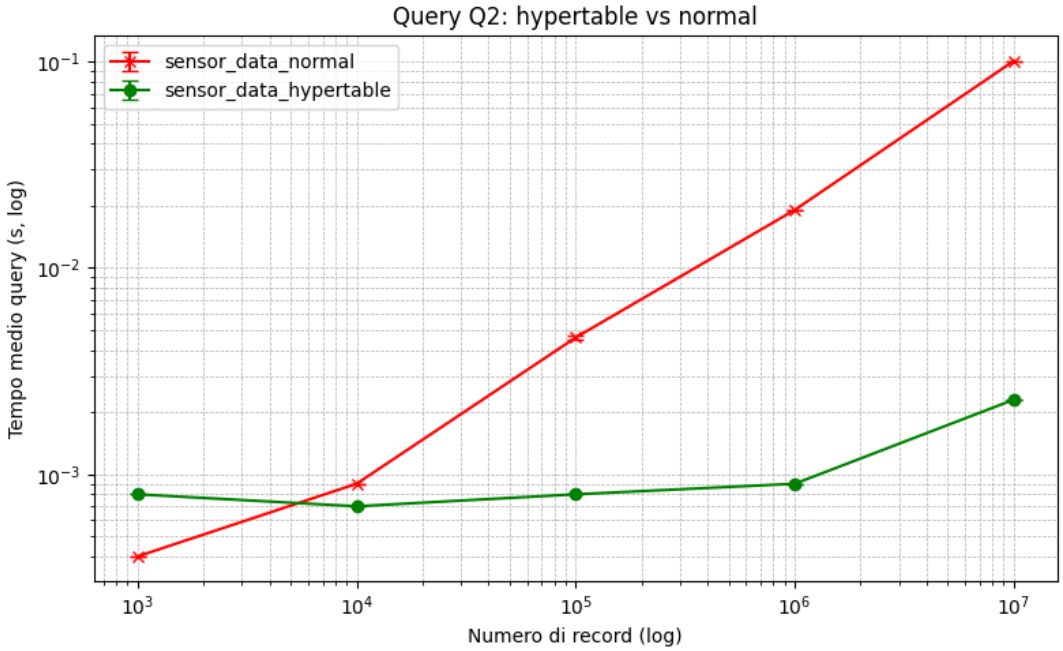

Figura 4.12: Q2 - Range Filter

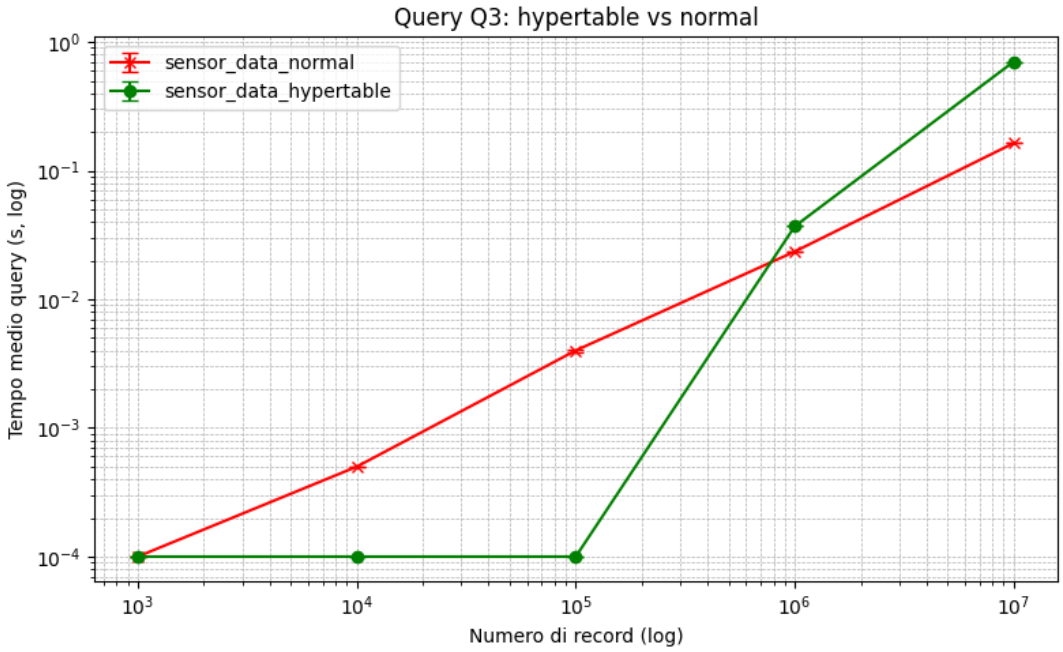

Figura 4.13: Q3 - Future Average

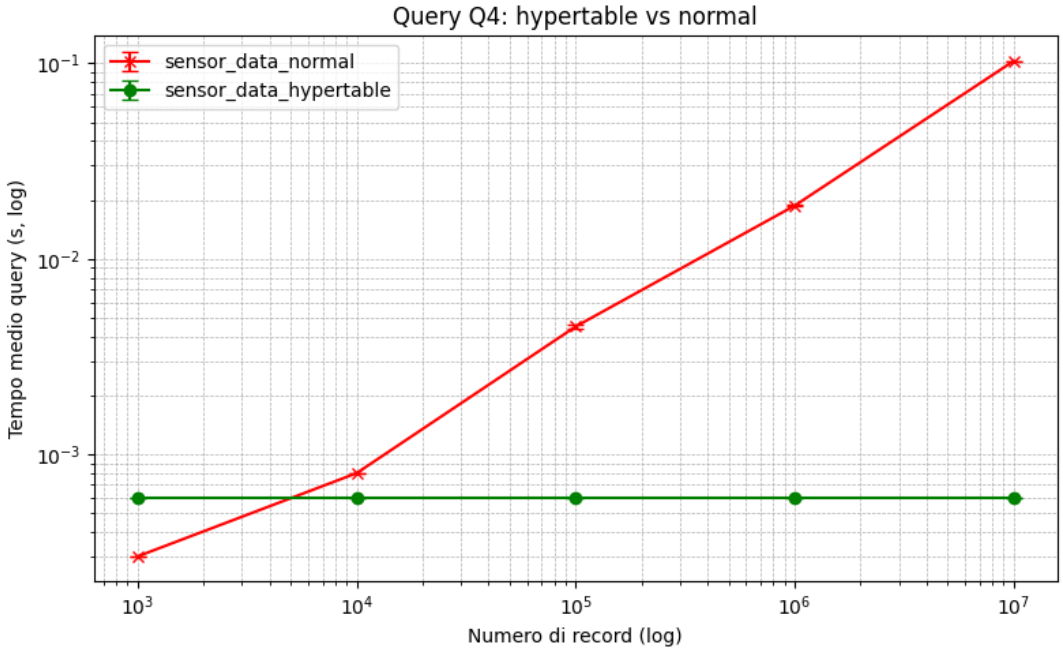

Figura 4.14: Q4 - Daily Average

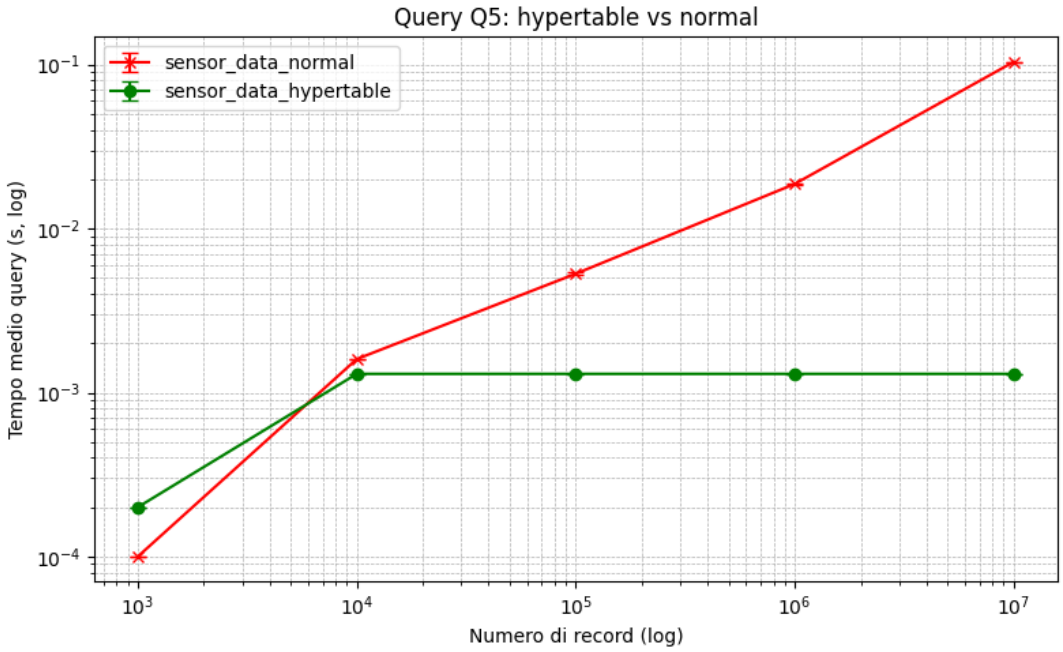

Figura 4.15: Q5 - Rolling Window Avg

4.4 Discussione dei risultati

I risultati ottenuti dimostrano l'elevata efficienza di **TimescaleDB** nelle query che coinvolgono **filtri temporali**, **aggregazioni temporali** o **funzioni di finestra**.

Nello specifico:

- In **Q2 (Range Filter)**, TimescaleDB mostra un vantaggio netto: con 10 milioni di righe, la query viene eseguita in circa `0.0023s`, contro `0.1007s` di PostgreSQL, questo perché si sfrutta appieno la struttura dell'*hypertable* divisa in chunk.
- In **Q4 (Daily Average)** e **Q5 (Rolling Window Avg)**, l'ottimizzazione nativa per analisi temporali consente di mantenere i tempi di esecuzione sotto i `2 ms`, anche con dataset molto ampi.
- In **Q3 (Future Average)**, notiamo un comportamento particolare: a partire da 1 milione di record, l'efficacia di TimescaleDB diminuisce rispetto a PostgreSQL. Questo comportamento è legato sia all'architettura del database, sia alla natura della query. Vediamo nel dettaglio.

  La clausola `WHERE time >= '...'` porta il motore a scartare una parte iniziale dei dati. Tuttavia, con l'aumentare del volume complessivo, la frazione di record da ignorare diventa sempre meno rilevante: il numero di dati "da leggere" tende ad avvicinarsi alla totalità del dataset. Di conseguenza, l'eventuale vantaggio di TimescaleDB nel saltare rapidamente i chunk iniziali perde "peso" rispetto all'operazione complessiva. Il vantaggio di scartare chunk perde dunque rilevanza e, anzi, porta ad un overhead dettato dalla gestione di questi ultimi (apertura, pianificazione, unione dei risultati).

  Tale overhead, in TimescaleDB, aumenta proporzionalmente al numero di chunk coinvolti, penalizzando le prestazioni in scenari ad alto volume, che in questo caso non vengono "coperte" dall'esclusione di molti chunk come accade nel caso d'uso ottimale.

Per quanto riguarda la **Q1 (Full Scan)**, invece, il vantaggio di TimescaleDB non è evidente. Su dataset grandi, PostgreSQL risulta persino più veloce, poiché il chunking introdotto da TimescaleDB non porta benefici quando è necessario leggere l'intero contenuto della tabella.

Da notare, inoltre, come le prestazioni di Timescale su dataset piccoli (1.000 record) siano simili a quelle di PostgreSQL, a causa dell'impossibilità, per il primo, di sfruttare appieno tutti i vantaggi visti (es. pochi chunk ignorati).

L'analisi mostra che **TimescaleDB** è particolarmente adatto in scenari in cui:

- Le query hanno una forte componente temporale.
- Si lavora con volumi di dati significativi (da milioni a decine di milioni di record).
- Sono necessarie analisi in tempo reale, rolling average o aggregazioni su intervalli regolari.

Le performance superiori di TimescaleDB, come spiegato nel capitolo precedente, sono giustificate dalla sua architettura:

- **Chunking automatico** dei dati in intervalli temporali.
- **Indicizzazione** ottimizzata per il tempo.
- **Compressione** e gestione efficiente dello spazio per storage a lungo termine.

4.5 Confronto con PostgreSQL

Il confronto evidenzia quanto segue:

- PostgreSQL è competitivo su query semplici o piccoli volumi.
- Al crescere dei dati, PostgreSQL mostra un degrado significativo delle performance su query temporali complesse.
- TimescaleDB, invece, scala in modo efficiente, grazie al **partizionamento automatico**, all'**indicizzazione sui chunk**, e a ottimizzazioni specifiche per l'elaborazione temporale.

In sintesi, PostgreSQL rimane valido per workload generalisti, ma mostra limiti quando utilizzato per analisi temporali intensive. TimescaleDB, invece, rappresenta una soluzione solida e scalabile per applicazioni come l'IoT, il monitoraggio, le metriche e tutte quelle applicazioni dove il tempo è una dimensione primaria dei dati.

Capitolo 5

Sperimentazione pratica

In questo capitolo verrà descritta la progettazione e l'implementazione della soluzione di raccolta e monitoraggio dei dati tramite un dispositivo ESP32 con sensore DHT-11, l'architettura del database TimescaleDB utilizzato per archiviare i dati, e l'integrazione con il sistema di monitoraggio real-time tramite Grafana.

5.1 Architettura del sistema

La Figura 5.1 mostra l'architettura generale del sistema sviluppato per la raccolta, l'archiviazione e la visualizzazione dei dati ambientali. L'intero flusso si basa su una pipeline che parte dall'acquisizione dei dati tramite un dispositivo embedded e si conclude con la visualizzazione in tempo reale tramite una dashboard. L'architettura è composta da quattro elementi principali:

- un microcontrollore ESP32 connesso a un sensore DHT11 per la misurazione di temperatura e umidità;
- un server backend leggero, sviluppato con Flask in Python, che riceve i dati via HTTP POST in formato JSON;
- un database TimescaleDB, estensione di PostgreSQL ottimizzata per la gestione di serie temporali, utilizzato per la memorizzazione dei dati ricevuti;
- una dashboard Grafana per l'esplorazione e la visualizzazione interattiva delle misurazioni che effettua delle query sul database, definite in fase di configurazione.

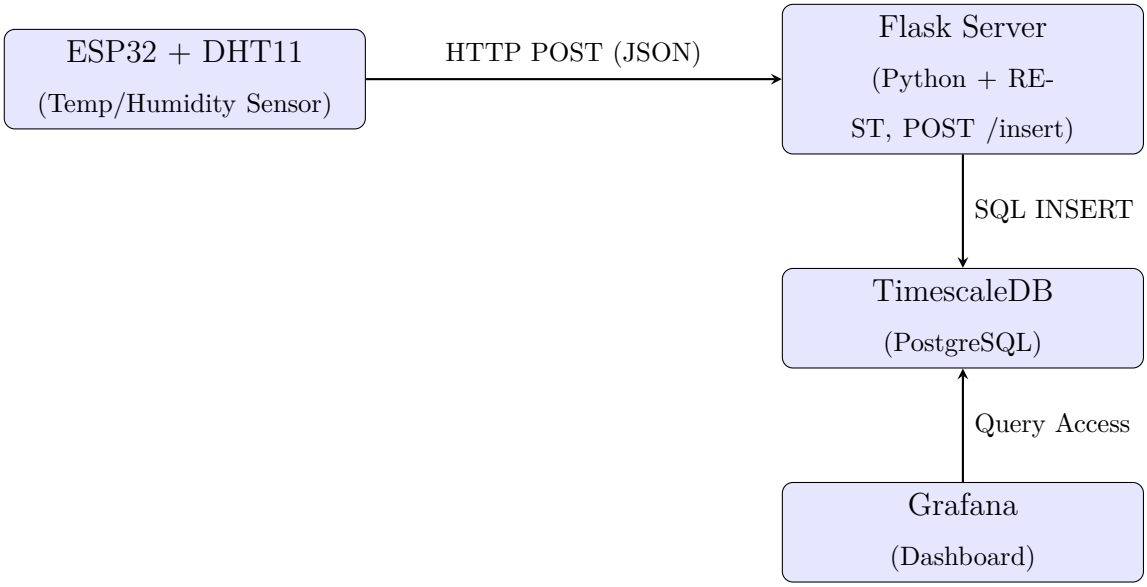

Figura 5.1: Architettura del sistema per la raccolta e visualizzazione dei dati

Nei paragrafi successivi, ciascun componente verrà descritto più nel dettaglio, evidenziandone il ruolo all'interno del sistema.

5.2 Raccolta dati tramite ESP32 e DHT-11

5.2.1 ESP32, breadboard e creazione del circuito

Un microcontrollore, come l'ESP32, è dotato di numerosi pin di ingresso/uscita (*GPIO - General Purpose Input/Output*) che consentono il passaggio di segnali elettrici, utilizzabili sia per il trasferimento di dati sia per l'alimentazione delle periferiche esterne[7].

La corretta identificazione e gestione di questi pin è fondamentale per garantire un'interfaccia affidabile e sicura con i dispositivi connessi.

Creare un circuito Un circuito è un percorso chiuso attraverso cui può fluire corrente elettrica. La sua realizzazione è necessaria perché i componenti elettronici - come microcontrollori, sensori, LED, resistenze, ecc. - funzionano solo se alimentati correttamente e collegati tra loro in modo che la corrente possa attraversarli e poi tornare al punto di partenza, chiudendo il ciclo chiamato, appunto, circuito[11].

Il concetto fondamentale è che l'elettricità ha bisogno di un percorso completo per fluire: dalla sorgente di tensione (ad esempio una batteria o un pin 3.3V dell'ESP32[7]), attraverso i componenti, fino al punto di massa (GND - ground). Se questo percorso non è chiuso, ad esempio se un componente è collegato solo al positivo ma non alla massa, la corrente non può fluire e il componente non funziona.

Collegare tutto con una breadboard Per facilitare l'organizzazione del circuito, viene comunemente utilizzata una breadboard, ovvero una piattaforma di prototipazione priva di saldature. Essa consente il collegamento rapido e modulare dei componenti elettronici mediante l'inserimento, a pressione, di cavi nei fori presenti sulla superficie (B).

Internamente, la breadboard è dotata di strisce conduttive (A) in metallo che permettono il passaggio della corrente elettrica, consentendo la connessione di più componenti tra loro senza la necessità di saldature.

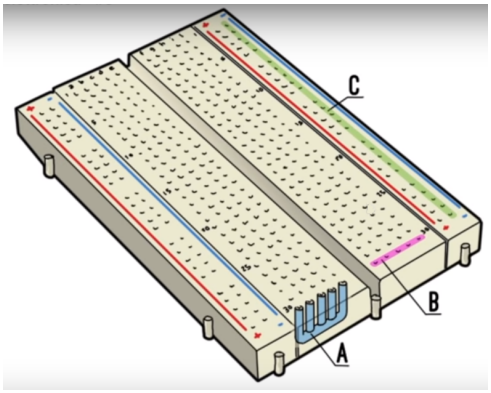

Figura 5.2: Breadboard [19]

La breadboard presenta due sezioni principali: le strisce di alimentazione (*power rails*) e le strisce terminali (*terminal strips*).

Le strisce di alimentazione, posizionate ai margini superiore e inferiore della breadboard, sono utilizzate per distribuire la tensione di alimentazione e la massa (GND) ai vari componenti del circuito. Queste strisce sono contrassegnate con simboli "+" e "-" e colori distintivi, come il rosso per il positivo (alimentazione) e il blu o nero per il negativo (massa), al fine di indicare chiaramente la polarità (la distinzione tra questi due poli).

Le strisce terminali, situate al centro della breadboard, sono utilizzate per collegare i componenti elettronici, a loro volta collegati al microcontrollore attraverso cavi chiamati *jumper* (quelli che inseriamo a pressione nei fori).

Chiudere il circuito: la massa Abbiamo detto che un circuito è considerato chiuso quando esiste un percorso continuo attraverso il quale la corrente può fluire dalla fonte di alimentazione, attraverso i vari componenti, e ritornare alla fonte stessa. In assenza di un percorso di ritorno, la corrente non può fluire, e il circuito risulta aperto, impedendo il funzionamento dei componenti.

Un aspetto cruciale nella progettazione dei circuiti è la messa a terra (grounding [1]). La massa (GND) funge da riferimento comune per tutte le tensioni nel circuito e fornisce un percorso di ritorno per la corrente. Collegare correttamente i componenti alla massa è fondamentale per evitare differenze di potenziale indesiderate che potrebbero compromettere il funzionamento del circuito o danneggiare i componenti sensibili [12].

5.2.2 Prototipo per rilevamento umidità/temperatura

Tra i vantaggi di un ESP32, vi è l'integrazione di un modulo Wi-Fi, qui utilizzato per inviare i dati letti dal DHT-11. Il DHT-11 è un sensore di temperatura e umidità che invia i dati in formato digitale, facilmente leggibile dal microcontrollore stesso.

Lo script è scritto in C++, utilizzando Arduino IDE (vedi Listing A.1 `sensor.ino`). l'ESP32 si collega a un server **NTP (Network Time Protocol)** per sincronizzare l'ora tramite il server pubblico `pool.ntp.org`. Il tempo ricevuto è in formato UTC (Coordinated Universal Time) e, di conseguenza, i dati vengono registrati nel database con un timestamp in formato `TIMESTAMPTZ`. Questo consente di garantire la coerenza e la correttezza temporale dei dati, indipendentemente dalla posizione geografica del dispositivo.

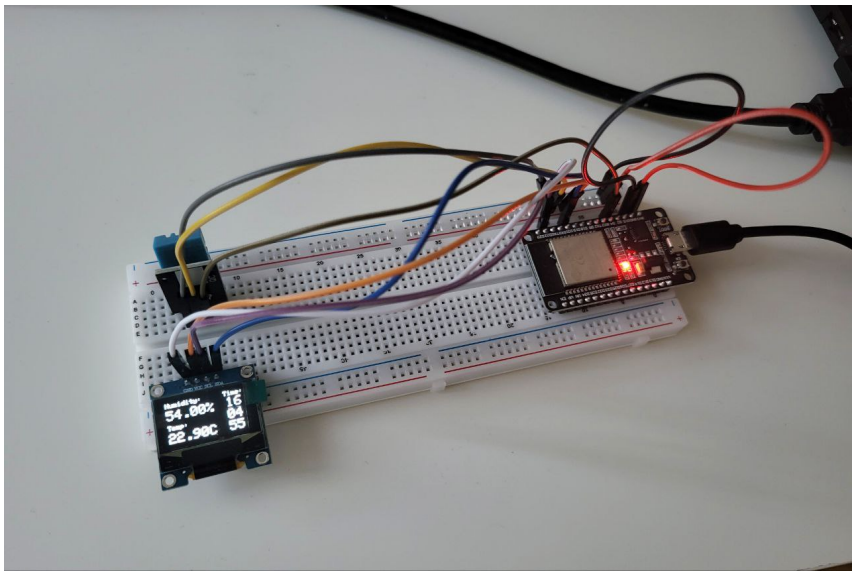

Figura 5.3: Prototipo con esp32, sensore e display

L'ESP32 genera i dati in formato **JSON**, che vengono inviati via HTTP POST a un server intermedio che gestisce la comunicazione con il database.

[1] Viene chiamata *Ground*, in italiano *messa a terra*, per via del fatto che, in passato (e ancora oggi), si utilizzava fisicamente il terreno per chiudere il circuito, in quanto la terra è un buon conduttore naturale, capace di assorbire e disperdere le cariche elettriche in modo sicuro [13].

Code Listing 5.1: Esempio di JSON inviato al server intermedio

```
{
  "temperature": 24.0,
  "humidity": 65.0,
  "timestamp": "1714582705"
}
```

5.3 Progettazione del database TimescaleDB

Il primo passo per configurare il database consiste nella creazione di un nuovo server tramite Railway, acceduto tramite **pgAdmin**. A questo punto, viene creato un database che ospiterà i dati raccolti dai sensori. All'interno di questo database, viene creata una *hypertable* per archiviare i dati provenienti dai sensori stessi.

I passaggi per inizializzare timescale Viene creata la tabella:

Code Listing 5.2: Creazione della tabella dei dati sensoriali

```
CREATE TABLE sensor_readings (
    time        TIMESTAMPTZ NOT NULL,
    humidity    DOUBLE PRECISION,
    temperature DOUBLE PRECISION
);
```

Successivamente, viene installata l'estensione **TimescaleDB** utilizzando il comando SQL:

```
CREATE EXTENSION IF NOT EXISTS timescaledb CASCADE;
```

Dopo aver installato l'estensione, la tabella viene trasformata in una **hypertable**, utilizzando il seguente comando:

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

È possibile verificare la creazione della hypertable con il comando:

```
SELECT * FROM timescaledb_information.hypertables;
```

A questo punto è tutto pronto per la ricezione dei dati e la loro gestione con TimescaleDB.

5.4 Inserimento dei dati tramite server Flask intermedio

Il sistema di inserimento dei dati è gestito da un **server backend** che funge da intermediario tra il dispositivo ESP32 e il database PostgreSQL. Il server, scritto in **Python** e basato sul framework **Flask**, riceve i dati JSON inviati dall'ESP32 tramite una richiesta HTTP POST e li inserisce nel database PostgreSQL.

Il backend è ospitato su **Railway.app**, una piattaforma cloud che permette di deployare applicazioni in modo semplice e veloce[18]. La repository su **GitHub**, da cui si prende il codice, contiene quattro file principali:

- `readings.py`: lo script che gestisce la logica di ricezione e inserimento dei dati nel database (vedi Listing A.2 - `readings.py`).
- `requirements.txt`: il file che contiene le librerie Python necessarie per l'applicazione (ad esempio, Flask e psycopg2 per la connessione al database).
- `Procfile`: un file di testo che indica a Railway come avviare l'applicazione.

Il codice Python in `readings.py` espone due endpoint:

- `POST /insert`: riceve i dati JSON inviati dal dispositivo ESP32 e li inserisce nel database.
- `GET /`: verifica che il server sia attivo.

Il backend è deployato su Railway con il link di accesso al server messo a disposizione da Railway stesso.

5.5 Utilizzo di Grafana per monitoraggio real-time

Dopo aver configurato correttamente il sistema di raccolta e inserimento dei dati, è possibile utilizzare **Grafana** per visualizzare i dati in tempo reale.

Grafana è uno strumento di visualizzazione open-source che si integra facilmente con TimescaleDB e consente di creare dashboard interattive per monitorare i dati in tempo reale.

L'aggiornamento dei dati avviene tramite l'esecuzione periodica di query SQL sul database sottostante: l'utente può definire l'intervallo temporale e le metriche da visualizzare[9].

Per configurare Grafana, si collega il database TimescaleDB come fonte di dati, utilizzando le credenziali di accesso e l'endpoint del database. Successivamente, vengono creati dei pannelli per visualizzare i dati di umidità e temperatura in tempo reale, con la possibilità di applicare filtri per visualizzare i dati in base a intervalli di tempo specifici (ad esempio, per visualizzare la temperatura per ora o per giorno).

Figura 5.4: Grafici di Grafana

Capitolo 6

Conclusioni e sviluppi futuri

L'analisi condotta ha approfondito le caratteristiche, le sfide e le opportunità offerte dalla gestione di dati time-series, con particolare attenzione alla soluzione proposta da *TimescaleDB*.

In un contesto tecnologico in cui i dati temporali assumono un ruolo sempre più centrale - basti pensare ai settori dell'IoT, del monitoraggio infrastrutturale, della finanza e del machine learning - la necessità di strumenti scalabili, affidabili e flessibili per la loro gestione risulta imprescindibile.

Dopo aver introdotto il concetto di time-series data, sono state analizzate le principali problematiche legate alla loro gestione: l'elevata cardinalità, la granularità temporale, l'ottimizzazione delle query per pattern temporali ricorrenti e la compressione efficiente dei dati storici. A fronte di queste sfide è emersa l'efficacia dell'approccio ibrido di TimescaleDB che unisce i vantaggi della modellazione relazionale di PostgreSQL alle ottimizzazioni specifiche per i dati temporali.

Dal punto di vista applicativo, TimescaleDB si è dimostrato particolarmente adatto a workload operativi che richiedono sia ingestion ad alta frequenza che analisi retrospettive complesse. Le sue funzionalità native, come le continuous aggregates e le politiche automatizzate di compressione e retention, riducono significativamente l'onere computazionale e operativo tipico dei sistemi time-series, restituendo valore aggiunto in termini di manutenibilità e scalabilità.

Sviluppi futuri

Gli sviluppi futuri possono seguire due direzioni principali. Da un lato, è possibile estendere ulteriormente l'analisi delle prestazioni di TimescaleDB su dataset reali o con dati creati artificialmente di maggiore complessità, includendo carichi di lavoro simulati su larga scala, scenari distribuiti. Dall'altro lato, sarebbe interessante esplorare l'integrazione

di TimescaleDB all'interno di pipeline di data engineering e machine learning, valutando l'efficienza della sua interazione con strumenti per l'analisi predittiva dei dati temporali.

Altro campo di studio potrebbe riguardare l'ottimizzazione delle query in ambienti multi-tenant (gestire dati di più utenti nello stesso database) o la valutazione dell'efficacia delle estensioni TimescaleDB nel cloud o l'opposto, cioè in un'architettura maggiormente orientata all'edge computing (es. filtrare e ridurre alla fonte la quantità di dati da inviare).

Sviluppi futuri: architettura peer-to-peer per la tutela dei dati in contesti critici

Un possibile sviluppo futuro per TimescaleDB potrebbe essere l'implementazione di un'architettura *peer-to-peer* (P2P) o distribuita avanzata. Tale evoluzione si rivelerebbe particolarmente utile in contesti ad alto rischio, dove il fallimento di un server centrale può avere conseguenze gravi, quali perdita di dati o interruzioni di servizio che potrebbero portare a rischi per la sicurezza e l'incolumità.

Un sistema TimescaleDB con architettura P2P permetterebbe di eliminare il singolo punto di fallimento tipico delle architetture centralizzate, garantendo una replica e ridondanza dei dati su più nodi distribuiti. Questo approccio aumenterebbe la resilienza del sistema, assicurando continuità operativa anche in presenza di guasti, attacchi informatici o condizioni di rete intermittenti.

Gli esempi di dati sensibili che beneficerebbero di questa tutela includono serie temporali di sorveglianza radar, telemetria da droni e satelliti militari, dati di monitoraggio di infrastrutture critiche come centrali energetiche, nonché flussi di dati provenienti da sensori per la sicurezza o il rilevamento di minacce chimiche, biologiche o radiologiche. O ancora, monitoraggio di infrastrutture di trasporto pubblico o impianti industriali. Questi dati sono caratterizzati da un'acquisizione continua e ad alta frequenza, e la loro integrità e disponibilità sono fondamentali per la sicurezza nazionale.

Nonostante le sfide tecnologiche legate alla coerenza dei dati in tempo reale, alla sincronizzazione sicura e alla risoluzione dei conflitti tra nodi, l'integrazione di meccanismi di consenso distribuito potrebbe fornire soluzioni efficaci per mantenere l'integrità e la sicurezza dei dati.

Un database distribuito tradizionale (come quelli basati su master-slave o cluster con replica sincronizzata) può garantire alta disponibilità e scalabilità, ma spesso continua a

dipendere da un coordinatore centrale o da nodi primari per la gestione della coerenza o del consenso.

In sintesi, l'adozione di un modello P2P per TimescaleDB rappresenta una promettente direzione di ricerca e sviluppo, in grado di ampliare notevolmente l'ambito di applicazione del database time-series in settori dove la tutela e la disponibilità continua dei dati sono fondamentali.

Conclusione In conclusione, l'adozione di TimescaleDB rappresenta oggi una soluzione matura e solida per affrontare le complessità associate ai dati temporali, ponendosi come ponte tra la tradizione dei database relazionali e le esigenze moderne della data-intensive economy.

Appendice A

Codice e script utilizzati

A.1 Codice ESP32

Code Listing A.1: Codice Arduino per ESP32

```
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include <DHT.h>
#include <WiFi.h>
#include <NTPClient.h>
#include <WiFiUdp.h>
#include <HTTPClient.h>
#include <WiFiClientSecure.h>
#include "wifi_credentials.h"

// Definizione dei dei pin e del tipo di sensore
#define DHTPIN 4          // Pin al quale il sensore è collegato
#define DHTTYPE DHT11

#define SCREEN_WIDTH 128 // OLED display width, in pixel
#define SCREEN_HEIGHT 64 // OLED display height, in pixel
// Declaration for an SSD1306 display connected to I2C (SDA, SCL pins)
// Dichiarazione per un display SSD1306 connesso con I2C (SDA, pin SCL)

Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire,-1);

// Inizializzazione del sensore dht
DHT dht(DHTPIN, DHTTYPE);

// Configura la connessione Wi-Fi
// Gia' definite in wifi_credentials.h

unsigned long lastPostTime = 0;  // Tempo dell'ultimo invio
const unsigned long postInterval = 5 * 60 * 1000; //5 minutes in ms

// Configurazione per il client NTP
WiFiUDP udp;
```

```
NTPClient timeClient(udp, "pool.ntp.org", 0, 3600000);  // Ottieni l'orario dal server
    NTP (fuso orario UTC)

void setup() {
  Serial.begin(9600);
  Serial.println("DHT Sensor Reading");

  // Inizializza il display OLED
  if(!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) { // Address 0x3D for 128x64
    Serial.println(F("SSD1306 allocation failed"));
    for(;;); // Loop infinito in caso di errore
  }


 // Codice per dare feedback sull'inizializzazione del display

  delay(2000);

  display.clearDisplay();
  display.setTextSize(2);
  display.setTextColor(WHITE);
  display.setCursor(0, 30);
  display.println("HI! :)");
  delay(100);
  display.display();

  dht.begin();          // Avvio del sensore DHT

  // Connessione al Wi-Fi
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    display.clearDisplay();
    display.setCursor(0, 0);
    delay(1000);
    Serial.println("Connecting to WiFi...");
    display.print("Connecting to WiFi...");
    display.display();
  }
  delay(1000);
  display.clearDisplay();  // Pulisci lo schermo
  display.setCursor(0, 0); // Posiziona il cursore
  Serial.println("Connected to WiFi");
  display.print("Connected to WiFi :)");
  display.display();
  delay(3000);

  // Avvia il client NTP
  timeClient.begin();
  timeClient.update();  // Sincronizza l'orario NTP
}


void loop() {
  delay(1000);  // Ritardo tra le letture (1000ms consigliato per DHT11)

  // Legge la temperatura in Celsius (default)
```

```
  float temperature = dht.readTemperature();
  // Legge umidità
  float humidity = dht.readHumidity();

  // Sincronizza l'orario NTP
  timeClient.update();
  // Ottieni l'orario dal client NTP
  String formattedTime = timeClient.getFormattedTime();
  // Ottieni timestamp UNIX
  unsigned long epochTime = timeClient.getEpochTime();

  // Check su errori di lettura
  if (isnan(temperature) || isnan(humidity)) {
    display.clearDisplay();  // Pulisci lo schermo
    display.setCursor(0, 0); // Posiziona il cursore
    display.print("Failed to read from DHT sensor!");
    display.display();       // Mostra il messaggio
    return;
  }

// Mostra le letture
  display.clearDisplay();
  display.setTextSize(1);
  display.setCursor(5, 0);
  display.print("Humidity: ");
  display.setTextSize(2);
  display.setCursor(5, 12);
  display.print(humidity);
  display.print("%");

  display.setCursor(5, 34);  // Sposta il cursore sulla riga successiva
  display.setTextSize(1);
  display.print("Temp: ");
  display.setTextSize(2);
  display.setCursor(5, 46);
  display.print(temperature);
  display.println("C");

  int hours = formattedTime.substring(0, 2).toInt();
  int minutes = formattedTime.substring(3, 5).toInt();
  int seconds = formattedTime.substring(6, 8).toInt();
  // Formatta con uno zero davanti per i numeri singoli (1-9)
  String hoursStr = (hours < 10) ? "0" + String(hours) : String(hours);
  String minutesStr = (minutes < 10) ? "0" + String(minutes) : String(minutes);
  String secondsStr = (seconds < 10) ? "0" + String(seconds) : String(seconds);
  display.setCursor(92, 0);  // Sposta il cursore sulla riga successiva
  display.setTextSize(1);
  display.print("Time:");
  display.setTextSize(2);
  display.setCursor(95, 10);
  display.print(hoursStr);
  display.setCursor(95, 28);
  display.print(minutesStr);
  display.setCursor(95, 46);
  display.print(secondsStr);
```

```
  display.display();  // Aggiorna il display

  // Mostra anche in console (debug)
  Serial.print("Humidity: ");
  Serial.print(humidity);
  Serial.print("%  Temperature: ");
  Serial.print(temperature);
  Serial.print("řC  Time");
  Serial.println(formattedTime);

  // Controlla se sono passati 10 secondi dall'ultimo invio
if (millis() - lastPostTime >= postInterval) {



  lastPostTime = millis();  // Aggiorna il tempo dell'ultimo invio

  if(WiFi.status() == WL_CONNECTED) {

    // Imposta l'URL per l'endpoint del server
    String serverUrl = String(dataRelayServiceURL) + "/insert";  // Aggiungi /insert al
        tuo URL base


    // Avvia la richiesta HTTP
  WiFiClientSecure secureClient;

  // Ignora la validazione SSL (questo è per la demo, non è sicuro)
  secureClient.setInsecure();

  // Crea un oggetto HTTPClient
  HTTPClient http;

  // Imposta l'URL di destinazione (assicurati che l'endpoint supporti HTTPS)
  http.begin(secureClient, serverUrl);
  http.addHeader("Content-Type", "application/json");

  unsigned long epochTime = timeClient.getEpochTime();

    // Formatta i dati in formato JSON
    String jsonData = "{\"temperature\": " + String(temperature) + ", \"humidity\": " +
        String(humidity) + ", \"timestamp\": \"" + epochTime + "\"}";

    // Invia la richiesta POST con il JSON
    int httpResponseCode = http.POST(jsonData);


if (httpResponseCode > 0) {
  Serial.print("HTTP Response code: ");
  Serial.println(httpResponseCode);
  String response = http.getString(); // Legge la risposta dal server
  Serial.println("Server Response: ");
  Serial.println(response);
} else {
  Serial.print("Error on sending POST: ");
```

```
  Serial.println(httpResponseCode);
  String errorResponse = http.getString(); // Legge la risposta d'errore
  Serial.println("Error Response: ");
  Serial.println(errorResponse);
}

    // Termina la richiesta HTTP
    http.end();
  } else {
    Serial.println("WiFi Disconnected");
  }
 }
}
```

A.2 Script server intermedio: endpoint /insert ed inserimento nel DB

Code Listing A.2: Script Python per invio dati e gestione DB

```
from flask import Flask, request, jsonify # libreria per web server
import psycopg2 # libreria per database PostgreSQL
from datetime import datetime # libreria per data e ora
import os # libreria per variabili d'ambiente (DB_HOST, DB_NAME, DB_USER, DB_PASS,
    DB_PORT)

# Creo un'istanza di Flask, che e' il server vero e propio
app = Flask(__name__)

# Connessione al database Railway:
# os.getenv(): prende i valori delle variabili d'ambiente.
# psycopg2.connect(): si connette al database usando quei parametri.
# cursor = conn.cursor(): rappresenta il cursore del database, permette di eseguire
    comandi SQL.
conn = psycopg2.connect(
    host=os.getenv('DB_HOST'),
    database=os.getenv('DB_NAME'),
    user=os.getenv('DB_USER'),
    password=os.getenv('DB_PASS'),
    port=os.getenv('DB_PORT')
)
cursor = conn.cursor()

# definisco l'endpoint per ricevere i dati dal client
@app.route('/insert', methods=['POST'])
def insert_data():
    data = request.get_json()
    temperature = data.get('temperature')
    humidity = data.get('humidity')
    timestamp = data.get('timestamp')

    # Se non c'è timestamp dal client, usa quello del server (non dovrebbe succedere mai)
    if not timestamp:
```

```
        timestamp = datetime.utcnow().isoformat()
    else:
        # Verifica se il timestamp è un intero (Unix timestamp)
        try:
            # Se è una stringa, prova a convertirla in intero
            if isinstance(timestamp, str):
                timestamp = int(timestamp)

            # Se è un timestamp Unix (numero di secondi), convertilo in formato ISO 8601
            if isinstance(timestamp, int) or isinstance(timestamp, float):
                timestamp = datetime.utcfromtimestamp(timestamp).isoformat()
        except (ValueError, TypeError):
            return jsonify({"error": "Timestamp non valido"}), 400

    try:
        # /% sono placeholder per i valori da inserire seguendo l'ordine indicato
            successivamente
        # consente di inserire i dati in modo sicuro, evitando SQL injection
        cursor.execute(
            "INSERT INTO sensor_readings (time, humidity, temperature) VALUES (%s, %s, %s
                )",
            (timestamp, humidity, temperature)
        )
        conn.commit()
        return jsonify({"message": "Dati inseriti con successo"}), 200
    except Exception as e:
        conn.rollback()
        return jsonify({"error": str(e)}), 500

@app.route('/')
def home():
    return 'Server attivo!'

# Eseguo il server Flask alla porta PORT definita in Railway, altrimenti 8000
# con host 0.0.0.0 accetto connessioni da qualsiasi indirizzo IP
if __name__ == '__main__':
    port = int(os.environ.get("PORT", 8000))
    app.run(host='0.0.0.0', port=port)
```

A.3 Codice ambiente di test Python

Code Listing A.3: Test Python per invio dati

```

import psycopg2
from psycopg2.extras import execute_values
import random
import datetime
import time
import os
import csv
import matplotlib.pyplot as plt
import math
import statistics
```

```
import configparser
import subprocess
from scipy.stats import norm  # z-score per intervallo di confidenza
from collections import defaultdict

def sync_and_drop_caches():
    print("[*] Sincronizzazione e svuotamento cache...")
    # Sincronizza i dati a disco
    subprocess.run(["sync"])
    # Svuota tutte le cache (pagecache, dentries, inodes)
    result = subprocess.run(["sudo", "sh", "-c", "echo 3 > /proc/sys/vm/drop_caches"])
    if result.returncode != 0:
        print("[!] Errore durante drop_caches. Hai i permessi di root?")
    else:
        print("[] Cache svuotata")

def pre_benchmark_clean():
    print("=== Preparazione ambiente pulito per benchmark ===")
    sync_and_drop_caches()
    print("=== Ambiente pronto ===")

# === Caricamento configurazione ===
config = configparser.ConfigParser()
config.read('config.ini')

DB_CONFIG = {
    "dbname": config["database"]["dbname"],
    "user": config["database"]["user"],
    "password": config["database"]["password"],
    "host": config["database"]["host"],
    "port": int(config["database"]["port"]),
}

TABLES = config["table"]["names"].split(",")
INPUT_FILE = config["input"]["file"]
START_DATE = datetime.datetime.strptime(config["input"]["start_date"], "%Y-%m-%d")
INTERVAL_MINUTES = int(config["input"]["interval_minutes"])
BATCH_SIZE = int(config["input"]["batch_size"])
REPETITIONS = int(config["query"]["repetitions"])
CONFIDENCE = float(config["query"]["confidence"])
Z_SCORE = norm.ppf(1 - (1 - CONFIDENCE) / 2)

# Salva i tempi medi per ogni query (Q1, Q2, ...) e tabella
log_data = defaultdict(lambda: defaultdict(list))  # {table: {query_index: [(num_records,
     avg_time)]}}

def create_tables(conn):
    cur = conn.cursor()
    cur.execute("""
        DROP TABLE IF EXISTS sensor_data_normal;
        CREATE TABLE sensor_data_normal (
            time TIMESTAMPTZ NOT NULL,
            value DOUBLE PRECISION
        );
    """)
    cur.execute("""
```

```
        DROP TABLE IF EXISTS sensor_data_hypertable;
        CREATE TABLE sensor_data_hypertable (
            time TIMESTAMPTZ NOT NULL,
            value DOUBLE PRECISION
        );
    """)
    cur.execute("SELECT create_hypertable('sensor_data_hypertable', 'time', if_not_exists
         => TRUE);")
    conn.commit()
    cur.close()
    print("Tabelle create correttamente.")

def define_output_filename(base_name, ext=".txt"):
    files = os.listdir()
    output_files = [f for f in files if f.startswith(base_name) and f.endswith(ext)]
    if not output_files:
        return f"{base_name}_1{ext}"
    max_num = 0
    for file in output_files:
        try:
            num = int(file[len(base_name) + 1 : -len(ext)])
            max_num = max(max_num, num)
        except ValueError:
            continue
    return f"{base_name}_{max_num + 1}{ext}"

def parse_input_file(filepath):
    with open(filepath, "r") as f:
        lines = [line.strip() for line in f.readlines()]
    val_range = eval(lines[0])
    num_records_list = list(map(int, lines[1].split(",")))
    queries = lines[2:]
    return val_range, num_records_list, queries

def clean_table(conn, table):
    cur = conn.cursor()
    cur.execute(f"DELETE FROM {table};")
    conn.commit()
    cur.close()

def populate_table(conn, table, val_range, num_records):
    cur = conn.cursor()
    data = []
    for i in range(num_records):
        timestamp = START_DATE + datetime.timedelta(minutes=i * INTERVAL_MINUTES)
        value = round(random.uniform(val_range[0], val_range[1]), 2)
        data.append((timestamp, value))
        if len(data) >= BATCH_SIZE:
            execute_values(cur, f"INSERT INTO {table} (time, value) VALUES %s", data)
            conn.commit()
            print(f"{table}: {i + 1} records inserted...")
            data = []
    if data:
        execute_values(cur, f"INSERT INTO {table} (time, value) VALUES %s", data)
        conn.commit()
    cur.close()
```

```
    print(f"All {num_records} records inserted into {table}.")

def compute_confidence_interval(times):
    n = len(times)
    mean = statistics.mean(times)
    stdev = statistics.stdev(times) if n > 1 else 0
    margin = Z_SCORE * (stdev / math.sqrt(n)) if n > 1 else 0
    return round(mean, 4), round(margin, 4)

def run_queries(conn, table, queries):
    results = []
    cur = conn.cursor()
    for query in queries:
        times = []
        result = None
        q = query.replace("sensor_data", table)
        for _ in range(REPETITIONS):
            start = time.perf_counter()
            cur.execute(q)
            result = cur.fetchone()
            end = time.perf_counter()
            times.append(round(end - start, 4))
        avg_time, conf_interval = compute_confidence_interval(times)
        results.append((q, times, avg_time, conf_interval, result))
    cur.close()
    return results

def save_results(table, results, val_range, num_records):
    base_name = f"output_{table}_{num_records}"
    txt_file = define_output_filename(base_name, ".txt")
    csv_file = txt_file.replace(".txt", ".csv")
    plot_file = txt_file.replace(".txt", ".png")

    with open(txt_file, "w") as f:
        f.write(f"Table: {table}\nRange: {val_range}\nRecords: {num_records}\n\n")
        for query, times, avg_time, conf, result in results:
            f.write(f"Query: {query}\nTimes (s): {times}\nAvg: {avg_time}s\nCI {
                CONFIDENCE * 100:.0f}%: ś{conf}s\nResult: {result[0] if result else 'None
                '}\n\n")

    with open(csv_file, "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(["Query", "Avg Time (s)", "Confidence Interval (s)", "Result"])
        for query, _, avg_time, conf, result in results:
            writer.writerow([query, avg_time, conf, result[0] if result else 'None'])

        # Salva i dati per grafico log
    for idx, (_, _, avg_time, _, _) in enumerate(results):
        log_data[table][idx + 1].append((num_records, avg_time, conf))

    generate_plot(results, plot_file, table, num_records)

def generate_plot(results, filename, table, num_records):
    queries = [r[0] for r in results]
    means = [r[2] for r in results]
    confs = [r[3] for r in results]
```

```
    colors = {
        "sensor_data_normal": "red",
        "sensor_data_hypertable": "green"
    }

    plt.figure(figsize=(10, 6))
    plt.bar(range(len(queries)), means, yerr=confs, capsize=5, color=colors.get(table, "
        gray"))
    plt.xticks(range(len(queries)), [f"Q{i+1}" for i in range(len(queries))], rotation
        =45)
    plt.ylabel("Execution Time (s)")
    plt.title(f"Query Times for {table} ({num_records} records)")
    plt.tight_layout()
    plt.savefig(filename)
    plt.close()


def generate_log_plots(log_data):
    plt.figure(figsize=(12, 6))
    markers = {"sensor_data_hypertable": "o", "sensor_data_normal": "x"}
    colors = {"sensor_data_hypertable": "green", "sensor_data_normal": "red"}

    for table, query_data in log_data.items():
        for query_idx, records_times in query_data.items():
            records, means, confs = zip(*sorted(records_times))
            plt.errorbar(records, means, yerr=confs,
                label=f"{table} Q{query_idx}",
                marker=markers[table],
                linestyle='-',
                color=colors[table],
                capsize=8,
                elinewidth=2,
                capthick=2,
                markeredgewidth=1.5)

            for x, y, ci in zip(records, means, confs):
                plt.text(x, y, f"š{ci:.1e}", fontsize=7, ha='center', va='bottom',
                    rotation=90)



    plt.xscale('log')
    plt.yscale('log')
    plt.xlabel("Numero di record (log)")
    plt.ylabel("Tempo medio query (s, log)")
    plt.title("Tutti i tempi di esecuzione query (log-log)")
    plt.grid(True, which="both", linestyle="--", linewidth=0.5)
    plt.legend()
    plt.tight_layout()
    plt.savefig("log_all_queries.png")
    plt.close()

    # Grafici per ogni query
    all_query_ids = set()
    for qidxs in log_data.values():
```

```
        all_query_ids.update(qidxs.keys())

    for query_idx in sorted(all_query_ids):
        plt.figure(figsize=(8, 5))
        for table in log_data:
            if query_idx in log_data[table]:
                records, means, confs = zip(*sorted(log_data[table][query_idx]))
                plt.errorbar(records, means, yerr=confs,
                             label=table,
                             marker=markers[table],
                             linestyle='-',
                             color=colors[table],
                             capsize=4)

        plt.xscale('log')
        plt.yscale('log')
        plt.xlabel("Numero di record (log)")
        plt.ylabel("Tempo medio query (s, log)")
        plt.title(f"Query Q{query_idx}: hypertable vs normal")
        plt.grid(True, which="both", linestyle="--", linewidth=0.5)
        plt.legend()
        plt.tight_layout()
        plt.savefig(f"log_query_Q{query_idx}.png")
        plt.close()


def main():

    pre_benchmark_clean()

    val_range, num_records_list, queries = parse_input_file(INPUT_FILE)
    conn = psycopg2.connect(**DB_CONFIG)

    print("Creazione tabelle...")
    create_tables(conn)

    for num_records in num_records_list:
        print(f"\n===== NUM_RECORDS: {num_records} =====")
        for table in TABLES:
            print(f"\n--- Testing {table} con {num_records} records ---")
            clean_table(conn, table)
            populate_table(conn, table, val_range, num_records)
            results = run_queries(conn, table, queries)
            save_results(table, results, val_range, num_records)


    print(f"\nGenerating log plots...")
    generate_log_plots(log_data)
    print(f"Log plots generated!")


    conn.close()
    print("\nBenchmark completato su tutti gli ordini di grandezza.")

if __name__ == "__main__":
    main()
```

Contenuto di `input.txt`

Code Listing A.4: File input.txt

```
[0, 30]
1000, 10000, 100000, 1000000, 10000000
SELECT * FROM sensor_data;
SELECT * FROM sensor_data WHERE time BETWEEN '2025-01-01 00:00:00' AND '2025-01-07
    00:00:00';
SELECT AVG(value) AS avg_value FROM sensor_data WHERE time >= '2028-12-01 00:00:00';
SELECT time::date AS day, AVG(value) AS avg_value FROM sensor_data WHERE time BETWEEN '
    2025-01-01' AND '2025-01-07' GROUP BY day ORDER BY day;
SELECT time, AVG(value) OVER (ORDER BY time ROWS BETWEEN 11 PRECEDING AND CURRENT ROW) AS
     rolling_avg FROM sensor_data WHERE time BETWEEN '2025-01-05' AND '2025-01-08';
```

Contenuto di `config.ini`

Code Listing A.5: File config.ini

```
[database]
dbname = test_timescaledb
user = postgres
password = 5151
host = localhost
port = 5432

[table]
names = sensor_data_normal,sensor_data_hypertable

[input]
file = input.txt
start_date = 2025-01-01
interval_minutes = 5
batch_size = 1000

[query]
repetitions = 30
confidence = 0.95
```

Riferimenti bibliografici

[1] *Comunicato stampa 5 ottobre 2017, postgresql global development group, posgtresql.org*, 2017. Retrieved June 28, 2025.

[2] D. J. Abadi, S. R. Madden, and N. Hachem, *Column-stores vs. row-stores: how different are they really?*, in Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, New York, NY, USA, 2008, Association for Computing Machinery, p. 967980.

[3] P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone, *Basi di dati*, McGraw-Hill Education, 5 ed., 2018.

[4] J. Blackwood-Sewell, *Timescaledb in 2024: Making postgres faster*, Jan. 2024. Accessed: 2025-05-10.

[5] ——, *What influxdb got wrong*, 2025.

[6] R. Booze, *Postgresql + timescaledb: 1000x faster queries, 90% data compression and much more*, June 2025.

[7] Espressif Systems, *Esp32 series datasheet*. `https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf`, 2024. Accessed: 2025-07-04.

[8] M. Freedman, *Timescaledb vs. influxdb: Purpose built differently for timeseries data*, Aug. 2018. Retrieved May 28, 2025.

[9] Grafana Labs, *Grafana documentation*. `https://grafana.com/docs/grafana/latest/`. Accessed: 2025-07-04.

[10] Z. Horvath, *8 performance improvements in recent timescaledb releases for faster query analytics*, Nov. 2023. Accessed: 2025-04-21.

[11] Istituto dell'Enciclopedia Italiana, *Circuito elettrico*. `https://www.treccani.it/enciclopedia/circuito-elettrico/`. Accessed: 2025-07-04.

[12] ——, *Messa a massa*. https://www.treccani.it/enciclopedia/messa-a-massa_(Dizionario-delle-Scienze-Fisiche)/. Accessed: 2025-07-04.

[13] ——, *Terra*. https://www.treccani.it/enciclopedia/terra_res-e435d4c9-8c69-11dc-8e9d-0016357eee51_(Dizionario-delle-Scienze-Fisiche)/. Accessed: 2025-07-04.

[14] Joshua Lockerman, Ajay Kulkarni, *Time series compression algorithms explained*. https://www.tigerdata.com/blog/time-series-compression-algorithms-explained, 2024. Consultato il 29 giugno 2025.

[15] J. B.-S. Michael Freedman, *Timescale architecture for real-time analytics*, tech. rep., Timescale Inc., March 2025. Documento ufficiale disponibile nella documentazione Timescale.

[16] A. Nielsen, *Practical Time Series Analysis: Prediction with Statistics and Machine Learning*, O'Reilly Media, 2019.

[17] PostgreSQL Global Development Group, *Vacuum postgresql documentation*. https://www.postgresql.org/docs/current/sql-vacuum.html. Accessed: 2025-07-04.

[18] Railway, *About railway*. https://docs.railway.com/overview/about-railway. Accessed: 2025-07-04.

[19] Separazione Galvanica, *Circuiti elettronici*. https://separazionegalvanica.wordpress.com/2021/02/28/circuiti-elettronici/, Feb. 2021. Accessed: 2025-07-04.

[20] Team Timescale, *What is a time series and how is it used?* https://www.tigerdata.com/blog/time-series-introduction, 2025. Accessed: 2025-07-04.

[21] The PostgreSQL Global Development Group, *PostgreSQL 17 Documentation*, PostgreSQL Global Development Group, 2024. Version 17, PDF manual.

[22] TigerData, *How to reduce bloat in large postgresql tables*. https://www.tigerdata.com/learn/how-to-reduce-bloat-in-large-postgresql-tables, Mar. 2024. Accessed: 2025-07-04.

[23] ——, *Postgresql performance tuning: Optimizing database indexes*. https://www.tigerdata.com/learn/

postgresql-performance-tuning-optimizing-database-indexes, Mar. 2024. Accessed: 2025-07-04.

[24] ——, *Hypertable API Documentation, Hypertables Chunks*, 2025. Consultato il 29 giugno 2025.

[25] Timescale, *Timescaledb documentation.* https://docs.tigerdata.com/, n.d. Accessed: 2025-06-27.

[26] K. Yank, *Sviluppare applicazioni con PHP e MySQL*, Tecniche Nuove, 2 ed., 2012.