UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

DIPARTIMENTO DI INGEGNERIA ENZO FERRARI

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

PROVA FINALE

Tecnologie per l'archiviazione dei dati generati dall' Internet of Things

RELATRICE **CANDIDATO**

Prof.ssa Sonia Bergamaschi ***Rosario Lissandrello***

ANNO ACCADEMICO 2018 - 2019

Sommario

Introduzione

Se guardiamo alle rivoluzioni tecnologiche avvenute negli ultimi anni non possiamo che notare come l'Internet of Things (IoT) sia sempre più presente nella nostra vita privata e lavorativa.
Siamo circondati da decine di oggetti smart in ogni ambito: guardiamo l' ora tramite gli smart watch, ascoltiamo musica tramite auricolari smart, a lavoro interagiamo con macchine industriali sempre più autonome e connesse tra loro, le nostre case sono domotizzate grazie all' utilizzo di sensori e dispositivi come Google Home o Amazon Alexa e le nostre automobili sono dotate di applicazioni che ci sanno sempre indicare in tempo reale la strada più veloce grazie alla connessione tra i veicoli.
Come tutte le rivoluzioni tecnlogiche, anche l'Internet of Things ci ha messo davanti nuove sfide da risolvere, una di queste è l' enorme quantità di informazioni (i così detti big data generati dall' IoT) che è necessario gestire e archiviare.
L' obbiettivo di questa tesi è quello di offrire una visione generale di alcuni tra i più famosi database che vengono utilizzati per la gestione dei Big Data generati dall' IoT.
La tesi sarà quindi divisa in tre macro-parti:
la prima sarà una parte di introduzione dove verrà descritto cos'è l'Internet of Things, cosa sono i Big Data che vengono generati da esso e in particolare si ci soffermerà sui dati time-series e sui database che vengono utilizzati per gestirli.
La seconda parte si occuperà di descrivere alcune delle tecnologie più utilizzate per l'archiviazione dei dati time-series soffermandosi su due database che risultano particolarmente utili per l' archiviazione dei dati generati dalla sensoristica: InfluxDB e TimescaleDB.
Per ogni tecnlogia presa in considerazione, in un primo momento si desciveranno le caratatteristiche approfondendo l'architettura e il modello dei dati, dopodichè si esporranno i pregi e i difetti di data tecnologia e verrano tratte delle conclusioni personali.
Nella terza e ultima parte verranno effettuati dei test, delle sperimentazioni e varranno approfonditi gli strumenti che TimescaleDB e InfluxDB possono offrire al mondo dell'IoT.

Big Data

In informatica il termine Big Data viene utilizzato genericamente per identificare una raccolta di dati che presenta determinate caratteristiche in termini di volume, velocità, varietà, veridicità e variabilità:

- **Volume**: si riferisce alla grande quantità di dati (strutturati o non strutturati) generati, esempi classici di enormi volumi di dati sono i dati contenuti nei social network, i movimenti sui mercati finanziari o i dati generati dall' Internet of Things.
- **Varietà**: riguarda la diversità dei formati e le differenti tipologie di dati che non hanno una struttura rappresentabile attraverso una tabella di un database relazionale, tra essi sono inclusi anche documenti di vario genere (txt, PDF, Word, Excel, ecc.) ma anche commenti sui social network e tanti altri casi specifici.
- **Velocità**: è la rapidità con cui i nuovi dati nascono e vengono acquisiti.
  Nell'IoT ad esempio vengono generati migliaia di dati al secondo e sono necessarie, oltre a tecnologie adeguate alla raccolta, anche tecnologie adeguate all' analisi real-time.
- **Veridicità**: cioè la qualità dell'informazione (in termini di accuratezza) che è possibile estrarre dai dati analizzati.
- **Variabilità**:il mondo in cui vengono generati i big data è mutevole e può portare a una inconsistenza dei dati analizzati, i dati cambiano velocemente e bisogna avere gli strumenti adatti ad interpretare l'informazione nella maniera corretta.

I big data possono essere generati dagli utenti o in maniera automatica dalle macchine.

IoT: Internet of Things

L’Internet of Things (IoT o Internet delle cose) è un neologismo che indica l’estensione di internet al mondo delle “cose”, aumentando la capacità di raccolta e di utilizzo dei dati da una moltitudine di sorgenti (prodotti industriali, sistemi di fabbrica, veicoli di trasporto, etc…) a vantaggio di una maggiore digitalizzazione e automazione dei processi.

Qualsiasi sistema di dispositivi fisici che ricevono e trasferiscono dati e che non richiedono interventi manuali può essere definito IoT.

Per “cosa” o “oggetto” si intendono tutte le apparecchiature o dispositivi che si contraddistinguono per determinate caratteristiche quali: identificazione, connessione, localizzazione, capacità di elaborare dati e capacità di interagire con l’ambiente esterno.

Un oggetto diventa “smart”, e quindi IoT, quando ai sensori dell’oggetto si integrano dei dispositivi di elaborazione.

Ad esempio, un termostato diventa smart quando integra delle componenti capaci di elaborare la temperatura rilevata e prendere decisioni di conseguenza come regolare la temperatura di casa o inviare un’e-mail al proprietario di casa.

Alcuni dei principali campi applicativi interessati dallo sviluppo della IoT sono:

- Agricoltura
- Domotica
- Reti wireless di sensori
- Smart city
- Industria

Time-series data

I time-series data sono sequenze di dati che rappresentano come un sistema, un processo o un evento si evolve con il passare del tempo.

Consistono solitamente in misurazioni successive applicate alla stessa sorgente su un intervallo di tempo.

Gli esempi più comuni dove vengono utilizzati sono:

- **Monitoraggio dei sistemi informatici:** VM, server, CPU, memoria libera, latenza, ecc.
- **Sistemi di trading finanziario:** criptovalute, pagamenti, eventi di tansazione, ecc.
- **Internet of Things:** dati provenienti dai sensori o da macchine industriali, dai dispositivi indossabili, automobili, e in generale da tutti i dispositivi “smart”.
- **Eventing applications:** dati sulle interazioni dell’utente come click, pagine visitate, logins, signup, ecc..
- **Business intelligence:** Monitoraggio delle metriche chiave e dell'integrità generale dell'azienda.
- **Monitoraggio ambientale:** temperatura, umidità, pressione, PH, flusso d’aria, CO2, NO2, ecc.

Time-series DBMS

I TimeSeries database sono database ottimizzati per collezionare, immagazzinare, recuperare e processare i time-series data.

Fondamentalmente raccolgono dati in funzione del tempo, quindi ad ogni valore raccolto associano un “Time Stamp”.

Per loro natura si prestano in maniera ottima ad essere utilizzati in applicazioni di IoT o Real-Time Analytics.

RANK	DBMS	SCORE		
NOV 2019		NOV 2019	24 MOS ▲	12 MOS ▲
1	**InfluxDB**	**19.93**	**+10.59**	**+6.29**
2	Kdb+	5.29	+3.44	+0.45
3	Prometheus	3.64	+2.83	+1.69
4	Graphite	3.32	+0.46	+0.47
5	RRDtool	2.90	-0.29	+0.17
6	OpenTSDB	2.13	+0.42	+0.11
7	Druid	1.79	+0.81	+0.43
8	TimescaleDB	1.73	+1.73	+1.19
9	FaunaDB	0.61	+0.44	+0.40
10	GridDB	0.57	+0.47	+0.40

Source: DB-Engines 23 Systems in Ranking, November 2019

Figura 1: Top 10 Time-series DataBases

Soluzioni per l’archiviazione dei dati generati dall’ IoT

La rivoluzione IoT ci ha messo davanti nuove sfide tecnologiche che implicano la gestione di problematiche che prima non esistevano. Una di queste è l’archiviazione dell’ enorme quantità di dati al secondo che vengono generati dagli oggetti smart. In questa tesi si descriveranno e si esporranno i pro e i contro di alcuni database utilizzati maggiormente per la gestione di questi tipi di dati.

Il primo argomento trattato sarà quello dei database server time-series e in particolare si parlerà di InfluxDB, che è attualmente il DBMS time-series più usato, successivamente si parlerà di TimescaleDB, un nuovo DBMS time-series nato come estensione di PostgreSQL che riesce ad offrire soluzioni che altri database noSQL non sono in grado di dare.

Si farà accenno a Prometheus, anch’ esso un database server time-series che si occupa del monitoriaggio dei sistemi informatici, integrarbile con InfluxDB e TimescaleDB.

Si descriverà poi Apache Accumulo, un sistema di archiviazione dati distribuito e ordinato basato su un sistema chiave-valore famoso per la sua capacità di gestione degli accessi a livello dei singoli elementi.

Infine si parlerà di Redis, uno store di strutture dati chiave-valore conosciuto per essere in-memory e quindi molto veloce.

InfluxDB

InfluxDB è un DBMS time-series NoSQL open-source scritto in Go.

Figura 2: logo influxDB

La sintassi del DB, chiamata influxQL, è SQL-like, inoltre da poco implementa un nuovo linguaggio per le query chiamato Flux che rende alcune query più semplici (anche se a discapito di dover imparare un nuovo linguaggio diverso da SQL).

Modello dei dati

In influxDB, ogni misurazione ha un timestamp, un set di campi (field value) e un set di tag (tag value).

Timestamp è una colonna obbligatoria per ogni dato di influxDB e contiene la data e l'ora (rappresentate in RFC3339 UTC).

Il **campo dei dati** contiene i valori della misurazione, ogni valore è composto da una colonna che identifica il dato (field key) e l'altra che contiene il valore effettivo.

I campi dei dati in influxDB non sono indexati, questo significa che le query sui dati devono scorrere tutta la colonna fino a quando non trovano il valore cercato e questo li rende poco performanti soprattutto quando è immagazzinata una grande mole di dati.

Infine, il **campo dei tag** contiene i valori dei tag, ogni tag è composto da una colonna key che identifica il tag e da una colonna che contiene il tag effettivo.

I tag sono opzionali ma è altamente consigliato utilizzarli, infatti sono indexati in-memory, ciò significa che le query sui tag sono molto più veloci.

Le regole generali che si possono seguire per l'archiviazione dei dati sono quindi:

- Archiviare i dati nei tag se sono meta data su cui si effettuano spesso query
- Archiviare i dati nei tag se si prevede di utilizzarli con i GROUP BY ()
- Archiviare i dati nei campi se si pevede di utilizzarli con le funzioni di InfluxQL[1]
- Archiviare i dati nei campi se è necessario che siano di un tipo diverso da string (i valori dei tag sono sempre interpretati come string)

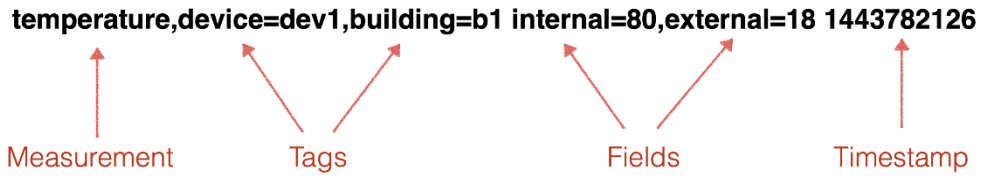


Figura 3: esempio di una misurazione su InfluxDB

I tipi di dati contenuti nei campi possono essere float, int, string e boolean e non possono essere cambiati senza essere riscritti. I valori dei tag contengono stringhe e non possono essere aggiornati.

InfluxDB non permette la modifica e la distruzione dei singoli campi dei dati e dei tag per migliorare le prestazioni di creazione e lettura, il motivo di questa scelta da parte degli sviluppatori di influx sta nell' idea che in un time-seriesDB dedicato alla raccolta e all' analisi dei dati ci si aspettino elevate prestazioni per quanto riguarda il numero di insert al secondo ed elevate prestazioni nell' analisi dei dati aggregati, cioè è più utilile fare un analisi su un gruppo di elementi e non sul singolo elemento, per esempio se si stanno monitorando le prestazioni di un server, è più utile sapere cosa sia successo tra le 9:00 e le 11:00 piuttosto che sapere cosa sia successo all' istante 9:45, quindi per aggevolare queste politiche di aggregazione, si è deciso di evitare l' eliminazione e la modifica dei singoli elementi.

[1] Funzioni InfluxQL: https://docs.influxdata.com/influxdb/v1.7/query_language/functions/

Politiche di conservazione (Retention policy RP)

Ogni singola misurazione può appartenere a diverse politiche di conservazione (retention policies). La politica di conservazione descrive per quanto tempo InfluxDB deve conservare il dato (durata) e quante copie del dato deve conservare nel cluster (replicazione), inoltre permette di mantenere i dati ad alta precisione solo per un periodo di tempo limitato ed immagazzinare i dati a bassa precisione per molto più tempo o per sempre in modo da consentire una migliore gestione dello spazio di memoria.

Se le politiche di conservazione non vengono settate manualmente, influxDB le crea in automatico impostando durata infinita e replicazione uguale a uno.

Archiviazione

InfluxDB archivia i dati tramite gli shard group.

Ogni gruppo ha la sua politica di conservazione (RP) e un suo intervallo temporale ed è composto da tutti i dati time-series che hanno il proprio time-stamp appartenente all'intervallo temporale del gruppo.

I dati appartenenti allo stesso gruppo aderiscono alla politica di conservazione del gruppo.

La durata del gruppo, cioè la durata dopo la quale il gruppo viene eliminato, può essere specificata durante la configurazione delle politiche di conservazione del gruppo e se non viene settata manualmente è impostata di default secondo i valori riportati nella tabella:

Durata politica di conservazione	Durata gruppo shard
< 2 giorni	1 ora
>= 2 giorni & <= 6 mesi	1 giorno
> 6 mesi	7 giorni

Scegliere l'intervallo di tempo corretto per gli shard group è molto importante, infatti una durata lunga permette a influxDB di archiviare più dati nella stessa posizione logica e questo riduce la

duplicazione dei dati, aumenta l' efficienza della compressione e aumenta la velocità delle query in alcuni casi.

D'altra parte, scegliere una durata corta permette al sistema di eliminare più efficentemente i dati ed effettuare backup incrementali.

Bisogna però tenere a mente che quando scade la durata di un shard group, influx elimina l'intero gruppo e non i singoli dati, anche se i dati hanno una politica di conservazione maggiore della durata del gruppo.

Compressione e Motore di archiviazione TSM

I dati timeseries vengono raggruppati in shard in base all' intervallo di tempo a cui appartengono, ogni shard viene mappato su un database del motore di archiviazione, ogni database ha il proprio WAL e il proprio file TSM.

Il motore di archiviazione di InfluxDB è chiamato **TSM** (Time-Structured Merge Tree) e molto simile ad un albero LSM[2].

È composto da più componenti con ruoli diversi:

- **In-Memory Index**: è un indice condiviso tra gli shard e fornisce l'accesso alle misurazioni, ai tag e ai dati timeseries.
- **WAL**: write ahead log, è un formato di archiviazione ottimizzato per la scrittura a lungo termine non facilmente interrogabile.
- **File TSM**: contengono i dati timeseries ordinati e compressi organizzati in shard.
- **Cache**: la cache viene interrogata in fase di esecuzione quando bisogna accedere ai dati memorizzati nel WAL.
- **FileStore**: serve per accedere ai file TSM presenti nel disco, inoltre si occupa di rimuovere i file TSM che non vengono più utilizzati.
- **Compactor**: si occupa di ottimizzare le cache e i file TSM, comprimere i timeseries, rimuovere i dati eliminati, ottimizzare gli indici, ecc.

[2] LSM-tree: https://en.wikipedia.org/wiki/Log-structured_merge-tree

- **Compaction Planner**: determina quali file TSM sono pronti per essere compattati
- **Compression**: consiste in vari codificatori e decodificatori che si occupano di comprimere i dati.
- **Writers/Readers**: ogni tipo di file (WAL, file TSM, ecc…) ha i suoi writers e readers specifici.

Time Series Index (TSI)

TSI (Time Series Index) è il nuovo motore di archiviazione log-structured merge tree-based database per la gestione dei dati time-series di InfluxDB.

È composto da più componenti con ruoli diversi:

- **Index:** contiene l'intero dataset indicizzato di ogni.
- **Partition:** contiene la partizione dei dati di ogni shard.
- **LogFile:** Contiene serie appena scritte come indice in memoria ed è persistente come WAL.
- **IndexFile:** Contiene un indice immutabile, mappato in memoria, creato da un file di registro o unito da due file di indice contigui.

Vantaggi

- Sintassi della query familiare (SQL-Like).
- Non ha dipendenze esterne.
- Scalabilità orizzontale.
- In-memory index.
- Compatta automaticamente i dati per minimizzare lo spazio di archiviazione.
- Politice di conservazione.

- Gli strumenti per il backup di influxDB permettono un'esecuzione del backup totale e/o parziale, permettendo di recuperare i dati da un punto specifico del passato.
- Offre una compressione on-disk significativamente migliore che in PostgreSQL e TimeScaleDB.

Svantaggi

- La sintassi non è SQL al 100%
- Non è possibile effettuare JOIN come in SQL
- Se si lavora con una grande quantità di dati per un periodo di tempo molto lungo si possono riscontrare problemi di archiviazione e di lentezza delle queries.
- InfluxDB implementa da zero alcuni strumenti per la fault-tolerance, come la replicazione, la disponibilità ai guasti e il backup/restore, chi usa influxDB è limitato a usare questi strumenti e molti di questi toold, come HA, sono disponibili solo nella versione a pagamento enterprise.

The Tick Stack

Lo stack TICK (Telegraf, InfluxDB, Chronograf e Kapacitor) è un set vagamente accoppiato ma strettamente integrato di applicazioni open source progettate la gestione e l' analisi di grandi quantità di dati time-series.

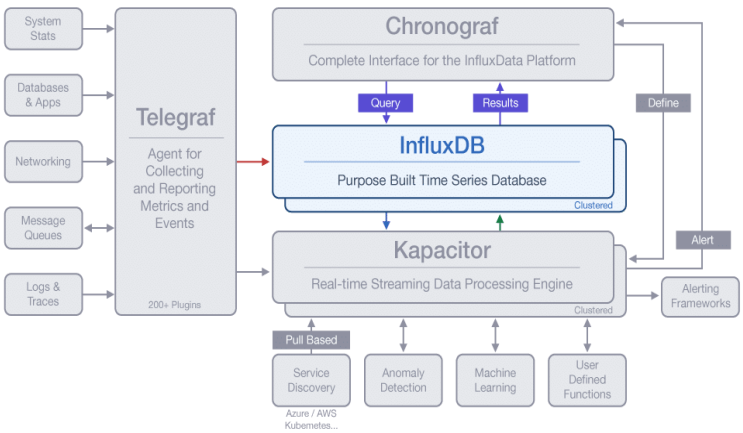


Figura 4: The Tick Stack

- Telegraf: è un server basato su plug-in per la raccolta e il monitoraggio delle metriche e degli eventi.
- InfluxDB: è il data-base time-series che si occupa di immagazzinare i dati.
- Chronograf: è l'interfaccia utente e il motore di visualizzazione dello stack.
- Kapacitor: è il motore di elaborazione dati real-time.

TimescaleDB

Figura 5: logo TimescaleDB

TimescaleDB è un DBMS timeseries opensource.

È un estensione di PostgreSQL ed è quindi un database relazionale che supporta il linguaggio SQL.

TimescaleDB sfrutta molti degli attributi di PostgreSQL come affidabilità, duttilità, sicurezza, connettività e una vasta gamma di programmi ed estensioni di terze parti, allo stesso tempo però l' alto grado di personalizzazione permette di avere un modello dei dati e un motore di archiviazione ottimizzato per la gestione dei dati time-series.

Modello dei dati

In TimescaleDB ogni misurazione time-series viene salvata nella propria riga con un campo temporale seguito da un numero qualsiasi di altri campi che possono essere float, int, string, boolean, array, Json blobs, dim. Geospaziale, date/time/timestamp, valuta, dati binari e molti altri. La possibilità di poter implementare molti campi (wide-table model) per ogni time-stamp e poter scegliere tra diversi tipi di dati è un vantaggio che possono offrire solo i database relazionali, molti database timeseries NoSQL infatti associano un solo campo di tipo stringa per ogni timestamp. TimescaleDB inoltre supporta i JOIN.

Architettura

Dal punto di vista dell'utente, TimescaleDB fornisce come interfaccia una singola tabella contenente tutti i dati delle serie temporali.

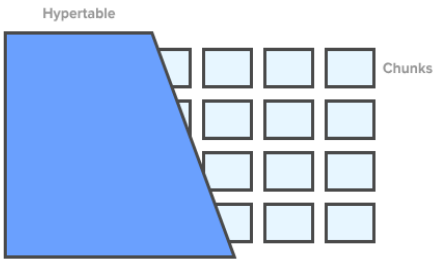


Figura 6: Hypertable

La tabella è suddivisa in blocchi e la dimensione di ogni blocco è calcolata per garantire le migliori prestazioni.
Questa tabella, chiamata **Hypertable**, è utilizzabile come una tabella standard PostgreSQL ed è quindi possibile utilizzare queries, inserts, upserts, triggers e schema changes su di essa.

Tutte le interazioni tra l' utente e timescaleDB come ad esempio l'interrogazione e l'inserimento dei dati avvengono tramite l' hypertable, è poi compito di timescaleDB assicurarsi che ogni nuova riga venga inviata in maniera trasparente al blocco appropriato e che ogni query interessi solo un numero minimo di blocchi.

Un database può avere più hypertable, ciascuno con il proprio schema e partizionamento.

Un Hypertable è definito da uno schema standard consistente in nomi colonne e tipi, con almeno una colonna che specifica il time-stamp e una colonna (opzionale) che specifica una key.

per creare un hypertable in TimescaleDB basta utilizzare due semplici comandi:

CREATE TABLE (come la sintassi SQL standard)

seguita da SELECT create_hypertable()

TimescaleDB divide lo spazio di archiviazione dell' hypertable in maniera automatica e trasparente.

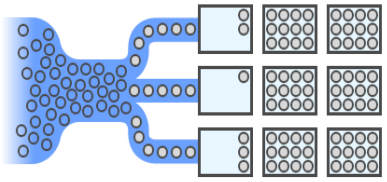


Figura 7: Chunks

Ciascuna partizione è chiamata **chunk** e corrisponde a uno specifico intervallo di tempo e a una regione dello spazio di partizionamento.

Queste partizioni sono disjointed, in modo da aiutare il query planner a minimizzare il set di chunks che bisogna utilizzare per risolvere una query.

Ogni chunk è implementato usando una tabella standard.

Compressione

La compressione è gestita dallo scheduler framework integrato di TimescaleDB e consiste nella conversione asincrona dei chunk da una forma decompressa “row-based” a una forma colonnare “colomn-based” attraverso l’ hypertable quando il chunk è considerato abbastanza vecchio dallo scheduler.
La compressione è trasparente, ciò significa che quando gli utenti applicano delle query vedono sempre lo stesso schema standard row-based, i blocchi non compressi vengono elaborati normalmente, mentre i blocchi compressi al momento della query vengono prima decompressi e convertiti nel formato “row-based” e poi uniti insieme agli altri.
Questo approccio è compatibile con tutto ciò che ci si aspetta da TimescaleDB, come i JOIN relazionali e le query analitiche.

Clustening

TimescaleDB non supporta ancora il clustening ma è in sviluppo e verrà rilasciato a breve.
Nel frattempo però già supporta tutte le funzionalità di PostgreSQL che riguardano la replicazione, la continua disponibilità delle risorse, la ridondanza e le query sharding read.

TimescaleDB vs PostgreSQL

I motivi per cui è preferibile usare TimescaleDB invece di PostgreSQL quando si tratta di gestire dati time-series sono principalmente tre:

- Un ingest-rate molto più elevato, soprattutto nel caso di database molto grandi.
- Prestazioni delle query simili o superiori in alcuni casi.
- Caratteristiche “time-oriented”.

Ingest-rate molto più elevato

Il motivo per cui nella gestione dei dati time-series timescaleDB ha un ingest-rate molto più elevato rispetto a PostgreSQL è da cercare nella gestione degli insert da parte di PostgreSQL.

Ogni volta che una riga viene inserita, postgres ha bisogno di aggiornare gli indici per ciascuna delle colonne indicizzate della tabella,se una tabella ha migliaia di righe (come nel caso delle tabelle time-series) questo si traduce in lavoro computazionale su ogni riga della tabella.

TimescaleDB risolve il problema utilizzando il partizionamento spazio-temporale, quindi tutte le scritture relative a intervalli di tempo recenti riguardano solo le tabelle che rimangono in memoria e, di conseguenza, anche l' aggiornamento di eventuali indici secondari è rapido.

Prestazioni delle query superiori o simili

Le query che riguardano in modo specifico l' ordinamento temporale possono essere molto più performanti in TimescaleDB.

Ad esempio, TimescaleDB sfrutta il fatto che i campi temporali siano già ordinati per effettuare un "merge append" ottimizzato, basato sul tempo, che riduce al minimo il numero di gruppi che devono essere elaborati.

Caratteristiche "time-oriented"

TimescaleDB include una serie di caratteristiche "time-oriented" che non sono presenti nei database relazionali tradizionali, questo include alcune ottimizzazioni speciali per le query orientate al tempo (come il merge-append di cui abiamo discusso in precedenza).

Inoltre TimescaleDB implementa e migliora alcune politiche di conservazione come ad esempio l'eliminazione dei timeseries in un hypertable dopo un intervallo di tempo definito.

Vantaggi

- Utilizzo del linguaggio SQL.

  TimescaleDB è uno dei pochi timeseriesDB che non richiede l' apprendimento di nuovi linguaggi per le query o l' utilizzo di linguaggi simili all' SQL.

- Utilizzo di un solo Database per tutti i tipi di dati, cioè la possibilità di non dover usare nello stesso sistema due DB, uno per i dati relazionali e uno per i dati time-series.

- JOINS: possibilità di effettuare join tra dati relazionali e dati time-series.

- Query performance: un Database relazionale garantisce un' alta velocità di risposta a query complesse.

- Possibilità di utilizzare vari tipi di dati e indici (B-tree, hash, range, BRIN, GiST, GIN), ereditando il supporto da PostgreSQL.

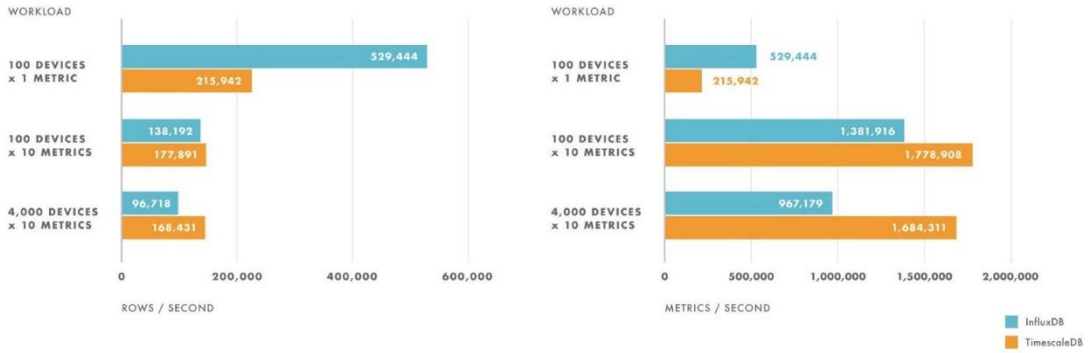
- Supporto nativo per i dati geospaziali: i dati memorizzati in TimescaleDB possono sfruttare i tipi di dati geometrici, gli indici e le query di PostGIS.

InfluxDB vs TimescaleDB

Nome	InfluxDB	TimescaleDB
Ranking DB-Engines	#32 Overall #1 TimeSeriesDBMS	#117 Overall #8 TimeSeriesDBMS
Primo rilascio	2013	2017
licenza	Open Source	Open Source
Linguaggio di implementazione	Go	C
Sistemi operativi supportati	Linux OS X	Linus OS X Windows
Tipi di dati supportati	Dati numerici e Stringe	Numerici, stringhe,boolean,array, JSON blobs, dim. Geospaziali, valute, binario, altri tipi di dati complessi
Supporto XML	No	Si
SQL	SQL-like	Si
API e altri metodi di accesso	HTTP API JSON over UDP	ADO.NET JDBC Librerie native C ODBC Streaming API for large objects

Linguaggi di programmazione supportati	.Net Clojure Erlang Go Haskell Java JavaScript JavaScript (Node.js) Lisp Perl PHP Python R Ruby Rust Scala	.Net C C++ Delphi Java JavaScript Perl PHP Python R Ruby Scheme Tcl
Triggers	No	Si

INSERT RATE



QUERY PERFORMANCE

measured in milliseconds

	100 DEVICES x 1 METRIC			100 DEVICES x 10 METRICS			4,000 DEVICES x 10 METRICS		
SIMPLE ROLLUPS[1]	**InfluxDB**	**TimescaleDB**	**InfluxDB / TimescaleDB**	**InfluxDB**	**TimescaleDB**	**InfluxDB / TimescaleDB**	**InfluxDB**	**TimescaleDB**	**InfluxDB / TimescaleDB**
single-groupby-1-1-1	3.28	2.32	141.1%	3.10	2.22	139.6%	3.19	4.54	70.3%
single-groupby-1-1-12	13.59	9.19	147.9%	13.16	9.68	136.0%	12.88	43.85	29.4%
single-groupby-1-8-1	12.30	15.89	77.4%	6.93	16.32	42.5%	6.83	27.93	24.5%
single-groupby-5-1-1	N/A	N/A	N/A	6.78	2.78	243.9%	6.64	4.79	138.6%
single-groupby-5-1-12	N/A	N/A	N/A	45.60	14.22	320.7%	43.55	47.99	90.7%
single-groupby-5-8-1	N/A	N/A	N/A	24.26	19.29	125.8%	23.42	28.57	82.0%
DOUBLE ROLLUPS[2]									
double-groupby-1	133.76	241.02	55.5%	82.35	194.57	42.3%	3438.00	5938.40	57.9%
double-groupby-5	N/A	N/A	N/A	386.38	252.64	152.9%	16580.34	8061.28	205.7%
double-groupby-all	N/A	N/A	N/A	754.40	284.66	265.0%	33354.96	10568.49	315.6%
THRESHOLDS[3]									
high-cpu-1	13.14	6.71	195.8%	29.32	12.00	244.3%	31.25	46.44	67.3%
high-cpu-all	1040.82	223.55	465.6%	2525.88	693.47	364.2%	121561.86	26617.25	465.7%
COMPLEX QUERIES									
lastpoint	466.04	4.93	9453.1%	307.52	5.78	5320.4%	2113.05	275.51	767.0%
groupby-orderby-limit	8542.55	3.38	252738.2%	8151.25	3.44	236954.9%	46593.98	58.27	79962.2%

[1] Queries are in this form: single-groupby - [number of metrics] - [number of devices] - [number of hours]
[2] Queries are in this form: double-groupby - [number of metrics] (for all devices, for 12 hours)
[3] Queries are in this form: high-cpu - [number of devices] (for random window of time)

- InfluxDB better
- InfluxDB slightly better
- TimescaleDB slightly better
- TimescaleDB better
- TimescaleDB much better

Insert rate

- Per carichi di lavoro con una cardinalità veramente bassa (es. 100 dispositivi), influxDB ha prestazioni di insert molto superiori a TimescaleDB, quando la cardinalità cresce, le prestazioni di insert sono superiori in TimescaleDB
- per carichi di lavoro con cardinalità da moderata ad elevata (es. 100 dispositivi che inviano 10 metriche, TimescaleDB è migliore di InfluxDB

Queries performance

- Per query semplici, i risultati variano un poco: ci sono alcuni casi in cui un database è chiaramente migliore dell' altro, mentre in altri casi dipende dalla cardinalità del dataset.
- Per query complesse, TimescaleDB è nettamente migliore di InfluxDB, e supporta una gamma più ampia di tipi di query. La differenza qui è spesso nell'intervallo da secondi a decine di secondi.

La soluzione migliore per scegliere quale db è migliore è quindi testare i due database utilizzando le query e l'hardware che si ha intenzione di utilizzare.

Osservazioni finali

Se dopo il benchmark specifico sui dati e sulle query risulta che InfluxDB sia più veloce e non è prevista un evoluzione dell' applicazione in futuro, allora può essere presa in considerazione l' idea di utilizzare InfluxDB, anche perchè come per la maggior parte dei database che utilizzano un approccio orientato alle colonne, offre una compressione su disco migliore dei database relazionali e quindi di TimescaleDB, in oltre influxDB utilizza un indice in-memory che gli permette di ottimizzare i tempi di risposta.

Tuttavia, il modello relazionale è più versatile e offre più funzionalità, flessibilità e controllo rispetto a influxDB, questo è molto importante se è prevista un evoluzione dell' applicazione e si necessita quindi un alta scalabilità e dell' utilizzo di query complesse o la necessità di lavorare con dati relazionali oltre che time-series.

In TimescaleDB è possibile, dato un DB relazionale composto da tabelle che contengono dati generici e tabelle che contengono campi di tipo timestamp, convertire in hypertable solo le tabelle che contengono il timestamp.

Al contrario in influxDB non solo non è possibile convertire tabelle relazionali in tabelle influx, ma non esistono strumenti semplici che permettano la migrazione di dati relazionali nelle tabelle di influxDB.

Se si vuole quindi convertire un DB relazionale in un DB timeseries o si vogliono migrare dai dati relazionali in timeseries data è assolutamente consigliato l' uso di timescaleDB.

Prometheus

Prometheus è un toolkit open-source basato sulla raccolta di dati time-series per il monitoraggio delle infrastrutture informatiche.

Figura 8: logo Prometheus

È supportato ufficialmente sia da influxDB che da timescaleDB.

Il suo scopo è quello di essere il sistema a cui si guarda durante un'interruzione per diagnosticare rapidamente i problemi, ogni server Prometheus rappresenta un nodo autonomo e non ha nessuna dipendenza dall' archiviazione distribuita e questo fa sì che si possa fare affidamento su di lui quando altre parti dell' infrastruttura sono danneggiate.

Utilizza PromQL come linguaggio per le query e la raccolta dei dati avviene tramite una pull HTTP.

Architettura

L' ecosistema consiste in più componenti, molti dei quali sono opzionali:

- **Server Prometheus:** archivia i dati time-series
- **Librerie client**
- **Push gateway**: supporta piccoli lavori real-time sul sistema in funzione
- Exporter con scopi specifici per servizi come HAProxy, StatsD, Graphite, ecc.
- **Alertmanager:** si occuppa di allertare l' amministratore di sistema se qualcosa ha smesso di funzionare
- **Grafana:** l'interfaccia utente, si occupa di visualizzare i dati raccolti, oltre a grafana si possono utilizzare anche altri API client.
- Altri tools di supporto

La maggior parte delle componenti di Prometheus sono scritte in Go.

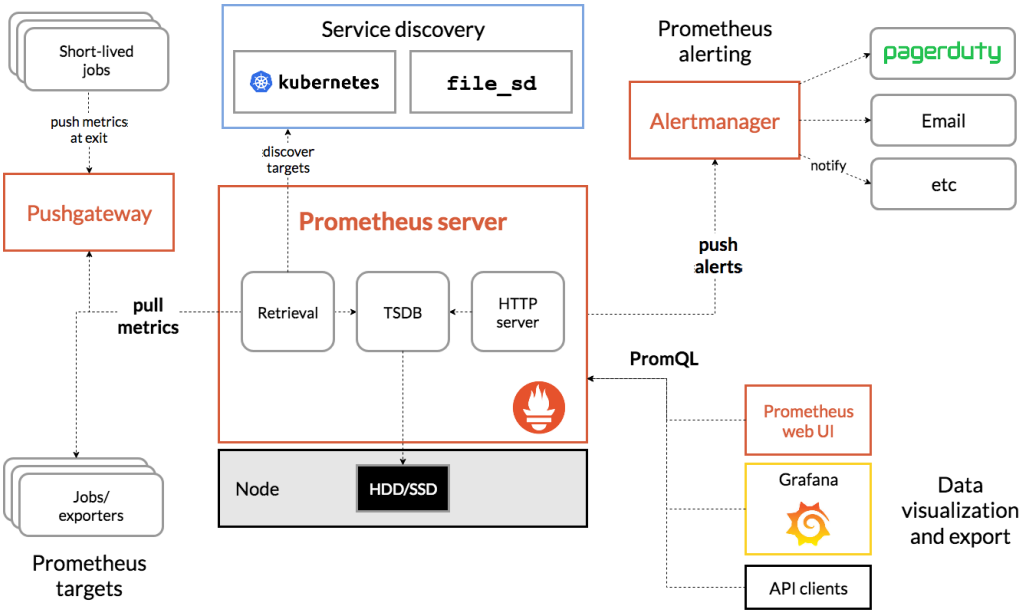


Figura 9: Architettura Prometheus

Apache Accumulo

Apache Accumulo è un sistema di archiviazione dati distribuito e ordinato basato su un sistema chiave-valore.

Figura 10: logo Accumulo

Accumulo utilizza il modulo HDFS di Apache Hadoop per immagazzinare i dati e Apache ZooKeeper per gestire i consensi, è scritto in Java ed è conosciuto per la sua capacità di gestione degli accessi a livello di cella, ogni coppia chiave/valore infatti ha una propria etichetta di sicurezza che limita i risultati della query sulla base delle autorizzazioni che possiede l'utente.

Modello dei dati

Accumulo archivia i dati secondo un modello key-value ordinato e distribuito.

I dati contenuti nel campo value vengono sempre considerati come array di bytes, questo significa che in Accumulo non ci sono tipi di dati, sarà responsabilità dell' applicazione che sta sopra al database tradurre gli array nel tipo di dato effettivo, permettendo così al DB di poter immagazzinare qualsiasi tipo di informazione.

Ogni key è rappresentata da un array di byte (ad eccezione del campo timestamp che è un long) ed divisa logicamente in vari campi che permettono agli sviluppatori di avere un maggior controllo dei dati a livello di sicurezza negli accessi:

- **Row ID** – identificatore univoco della riga; se più righe hanno lo stesso ID, vuol dire che fanno parte della stessa riga logica che contiene più valori.

- **Column Family** - Raggruppamento logico delle chiavi, questo campo può essere utilizzato per partizionare i dati all'interno di un nodo aumentando così la velocità delle query.

- **Column Qualifier** – attributo che fornisce un ulteriore univocità alla chiave.

- **Column Visibility** – Etichetta di sicurezza che controlla l'accesso alla coppia chiave/valore, indica quali autorizzazioni sono necessare per leggere il valore.

- **Timestamp** – Generato automaticamente, viene interpretato come un Java Long Integer ed è usato per identificare le varie versioni del valore, accumulo da la possibilità di scegliere se tornare tutte le versioni come risposta ad una query (nel caso in cui ne esistano più di uno) o tornare solo la più recente.

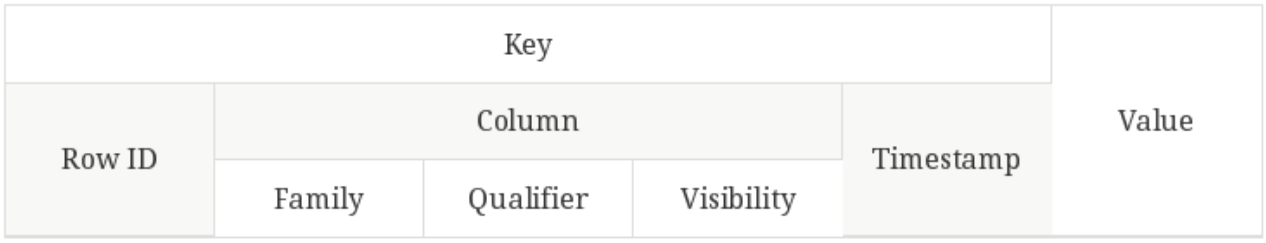


Figura 11: Modello elemento chiave-valore di Accumulo

Accumulo ordina le righe tramite le chiavi, attuando prima un ordinamento alfanumerico ascendente sulle rowID, poi sulle colonne e infine un ordinamento numerico discendente sui timestamp, quindi la versione più vecchia della stessa chiave appare per prima in uno scan sequenziale. La tabella consiste quindi in un set ordinato di coppie chiave-valore.

Autorizzazioni

Le autorizzazioni sono un insieme di stringhe che consentono a un utente di leggere i dati protetti. Gli utenti ottengono le autorizzazioni e scelgono quali utilizzare per la scansione di una tabella. Le autorizzazioni scelte vengono valutate in base alla ColumnVisibility di ciascuna chiave durante scansione, se l'espressione booleana di ColumnVisibility viene valutata come true, i dati saranno visibili all'utente.

Architettura

Accumulo è un data storage distribuito che consiste in un insieme di componenti architetturali che comunicano tra loro, alcuni dei quali sono installati su server differenti.

Le componenti che compongono un istanza di Accumulo sono:

- **TabletServer**: ne esiste almeno uno per ogni server e ognuno di essi gestisce più sottogruppi di tutte le tabelle (le partizioni delle tabelle vengono chiamate tablet).
  Si occupa di restiture i dati al cliente in caso di richiesta di lettura e di scrivere dati sulla partizione in maniera persistente in caso di richiesta di scrittura.
  Inolte si occupa di ordinare le coppie chiave-valore.
- **Logger**: quando un Tablet-server ha bisogno di scrivere in maniera persistente sul disco incarica il logger. Ogni Tablet-server può inviare la richiesta a più logger in modo da preservare i dati in caso di errori (replicazione dei dati).
- **Garbage Collector**: poichè Accumulo processa i dati archiviati nell' HDFS, ciclicamente il Garbage Collector si occupa di trovare i file che non sono stati utilizzati per molto tempo da nessuno dei processi ed eliminarli.
- **Master**: il master è responsabile del rilevamento e della correzione dei fallimenti dei Tablet Server. Si occupa di bilanciare il carico di lavoro distribuito tra tutti i Tablet Server, anche migrando i dati quando necessario.
- **Client**: ad ogni applicazione viene collegata una libreria client. Quando l' applicazione ha bisogno di leggere o scrivere un determinato set di dati, la libreria client si occupa di trovare il server dove sono immagazzinati o dove devono essere immagazzinati i dati e comunicare con il Tablet-server corrispondente per scrivere e/o recuperare le coppie chiave-valore.

Gestione dei dati

Accumulo archivia i dati nelle tabelle, ogni tabella è partizionata in tablet, i tablet a loro volta sono partizionati e distribuiti dal master ai vari tablet-server. In base al carico di lavoro, il master può decidere di spostare i tablet in modo da assicurarsi che non si creino rallentamenti nel cluster e che i dati risultino sempre disponibili.

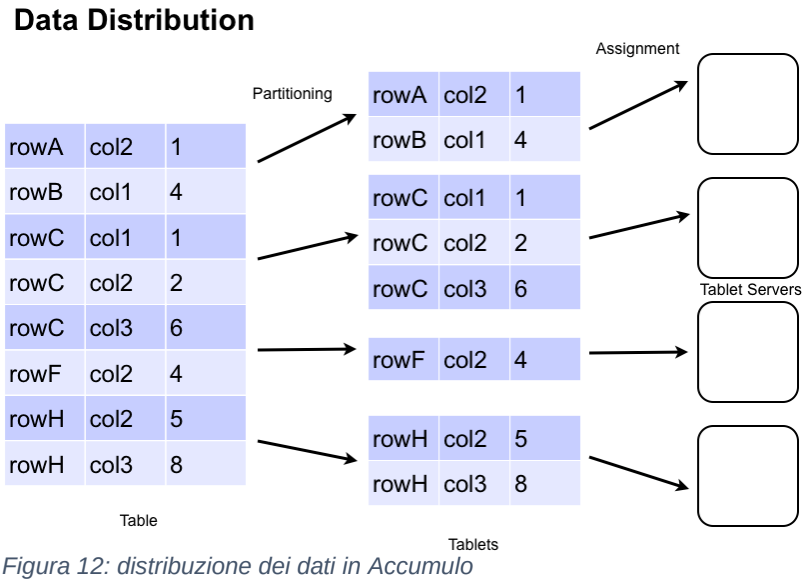


Figura 12: distribuzione dei dati in Accumulo

Tolleranza ai guasti

Se un Tablet-server si guasta, il master lo rileva e riassegna automaticamente i tablet assegnati al server guasto ad un altro server.

Tutte le coppie chiave-valore che erano in memoria al momento del guasto vengono replicate automaticamente dal registro dei log evitando così la perdita dei dati.

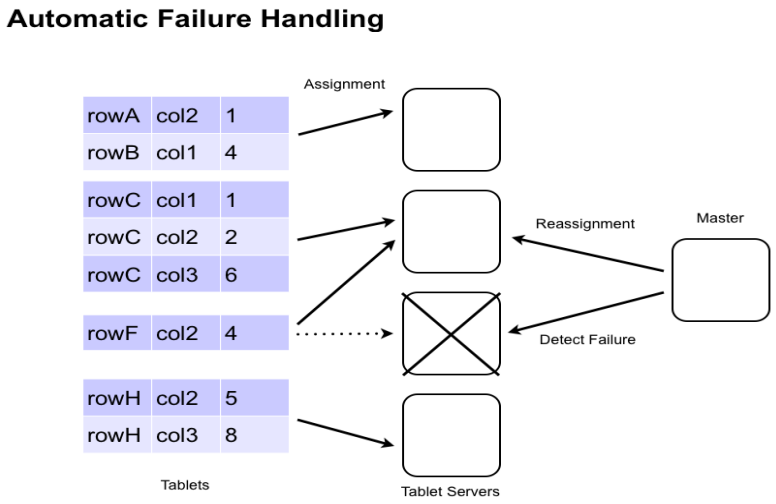


Figura 13: gestione delle perdite di dati in Accumulo

Considerazioni finali sull'utilizzo di Accumulo in applicazioni IoT

Essendo uno store distribuito, l' implementazione di Accumulo non è certo facile.

Se bisogna implementare soluzioni per piccole applicazioni abbiamo già visto che esistono altre soluzioni di più facile implementazione che garantiscono delle ottime prestazioni. A mio parere, è consigliabile scegliere Accumulo quando si sta lavorando in applicativi veramente grandi che richiedono una distribuzione dei dati in un cloud (partizione e replicazione) e soprattutto è richiesto un alto livello di sicurezza negli accessi ai dati.

Redis

Redis è uno store di strutture dati in-memory e open source che può essere utilizzato come database.

Si basa sul concetto di struttura chiave-valore ed è scritto in c.

Data la sua velocità (dovuta alla struttura in-memory) e data inoltre la capacità di supportare la raccolta dei dati tramite un flusso continuo (streaming), Redis è un scelta molto comune per l' archiviazione dei dati in applicazioni IoT.

Figura 14: logo redis

Modello dei dati

Redis è considerato un evoluzione della semplice struttura chiave-valore, infatti, mentre nella struttura classica sia la chiave che il valore sono stringhe, in redis è possibile utilizzare dei valori più complessi consistenti in delle strutture dati, ogni struttura determina quali operazioni (comandi sul terminale) è possibile fare su quel determinato valore.

Oltre alle stringhe, le strutture dati che possono essere utilizzate come valori sono:

- **List:** collezione di stringhe ordinate in base all' ordine di inserimento, viene implementato tramite una linked list e ciò porta ad un tempo di inserimento sempre costante anche nel caso di liste molto grandi, questo risulta molto utile nell' archiviazione di dati time-series che normalmente richiedono un elevato insert-rate.
- **Set:** collezione di stringe univoche e non ordinate.
- **Sorted set:** è simile a un set ma ad ogni elemento è associato un float chiamato "score" e gli elementi sono sempre ordinati in base al loro score.
- **Hash:** è un map formato da elementi chiave-valore, sia le chiavi che i valori sono stringhe.
- **Bitmap:** considera le stringhe come se fossero array di bit e viene utilizzato quando c' è bisogno di lavorare sui singoli bit.
- **HyperLogLogs:** spesso abbreviata in HLL, è una struttura dati utilizzata per il conteggio degli elementi archiviati nei set, infatti, mentre normalmente per contare gli elementi

contenuti in un set è necessaria un utilizzazione di memoria proporzionale al numero di elementi che corrispondono ai parametri di ricerca, HLL si serve di un algoritmo di conteggio che utilizza sempre una quantità costante di memoria.

- **Stream:** è una struttura dati log append-only.

Per quanto riguarda il campo chiave, redis tratta il valore delle chiavi come se fossero degli array di bit, ciò significa che anche se normalmente si preferisce usare delle stringhe, in realtà può essere utilizzata letteralmente qualsiasi cosa sia una sequenza binaria, come ad esempio un immagine JPEG o anche una stringa vuota.

La lunghezza massima consentita di una chiave è di 512MB.

Nonostante Redis lasci molto potere decisionale nella scelta della chiave, è buona norma seguire alcune accortezze:

- Non conviene utilizzare key molto lunghe, sia per una questione di occupazione inutile di memoria (che ricordiamo essere in-memory) sia per una questione di costo di comparazione tra le chiavi.
- Non conviene utilizzare key molto corte, soprattutto per una questione di leggibilità.
- Sempre per una questione di leggibiltà è consigliato assegnare delle chiavi che siano comprensibili

Architettura

A differenza di un DBMS tradizionale, che archivia i propri dati direttamente in memoria secondaria, Redis è una struttura dati in-memory, cioè una struttura dove i dati vengono scritti in un primo momento in memoria primaria (RAM) e solo successivamente archiviati in memoria secondaria (disco fisso o ssd), questo garantisce delle prestazioni elevate dovute al fatto che non è necessario accedere alla memoria secondaria ogni volta che bisogna leggere/scrivere dei dati.

L' architettura di redis è dunque composta da due parti principali: un client e un server.

Il server è responsabile di archiviare i dati in maniera persistente sul disco, il client invece archivia i dati in maniera temporanea sulla memoria primaria.

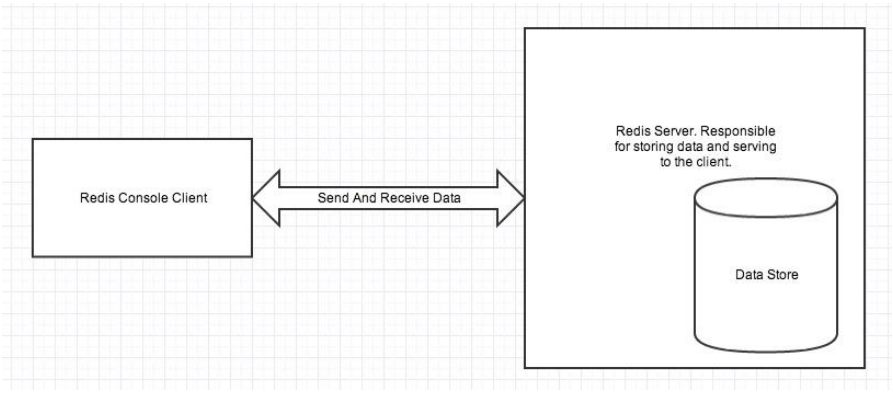


Figure 15: architettura client-server di Redis

Poiché la memoria primaria è temporanea e relativamente piccola, è necessario che entrambe le componenti si parlino così che possano inviarsi e scambiarsi i dati.

Redis fornisce tre soluzioni per garantire la persistenza dei dati: RDB, AOF e SAVE.

- RDB copia ciclicamente i dati in-memory e li archivia nella memoria secondaria. Questo avviene a intervalli di tempo predefiniti e ciò significa che potrebbero perdersi dei dati qualora il client smetta di funzionare prima che l'RDB effetui la sua copia periodica. Inoltre, se il dataset è grande, effettuare la copia dei dati potrebbe rallentare le performance del database.
- AOF copia i dati in-memory nella memoria secondaria ogni volta che vengono modificati, il problema di utilizzare AOF sta nel fatto che la scrittura/lettura del disco per ogni operazione impiega molte risorse ed inoltre a parità di dati il file del AOF è più grande del file RDB e quindi occupa più memoria.
- SAVE è un comando su console utilizzabile ogni volta che si vogliono archiviare in maniera manuale i dati sul disco.

Per quanto riguarda il backup dei dati, redis non ha nessuna componente che si occupi di backup e recovery. Ciò significa che se si utilizza Redis è necessario affiancarlo a software esterni che si occupino del backup.

Redis supporta il partizionamento e la replicazione dei dati:

Il partizionamento consente la creazione di database molto grandi e aumenta la potenza computazionale poiché la memoria primaria totale è data dalla somma delle memorie di più computer, senza il partizionamento invece si è limitati dalla grandezza della memoria di un solo computer.

La replicazione aumenta la disponibilità dei dati in caso di guasti.

Considerazioni finali sull'utilizzo di redis in applicazioni IoT

La struttura in-memory di Redis garantisce un elevata velocità di lettura e scrittura dei dati e questo è molto utile quando bisogna gestire sensori IOT che generano una grande quantità di dati al secondo o in applicazioni IOT che hanno bisogno dei valori archiviati nel minor tempo possibile.

Inoltre la stuttura chiave-valore si presta molto bene alla raccolta dei dati time-series.

Bisogna tener conto però che una struttura in-memory presenta un grande limite dovuto alla grandezza della memoria e che quindi non è adatto ad essere utilizzato in applicazioni IoT che generano una quantità di dati molto elevata.

Sperimentazione sul dataset

Tra tutte le tecnologie descritte in questa tesi posso concludere che TimescaleDB sia quella che mi ha colpito di più, soprattutto per la sua capacità di fondere la duttilità di un database relazionale con la velocità di un database timeseries.

In questa terza ed ultima parte verrà descritto come scaricare e installare TimescaleDB,

si effettuerà una comparazione tra Timescale e Postgres e verranno approfondite alcune caratteristiche di Timescale come la possibilità di supportare i dati geospaziali.

Infine verrà descritto come scaricare e installare InfluxDB, si utilizzerà un software chiamato Outflux che si occupa della migrazione dei dati da Influx a Timescale e si effettuerà una comparazione tra InfluxDB e TimescaleDB.

Nota Bene:

Durante la sperimentazione mi riferirò a PostgreSQL, TimescaleDB e InfluxDB chiamandoli DB Server o DBMS.

Quando invece parlerò di database allora mi riferirò allo schema e ai dati veri e propri.

Installazione di TimescaleDB

Poiché TimescaleDB è un estensione di PostgreSQL, per prima cosa è necessario scaricare e installare PostgreSQL.

È possibile effettuare il download attraverso questo link: https://www.postgresql.org/download/

Se si desidera scaricare e installare PostgreSQL per windows, bisogna scaricare l' esecutivo e seguire i passaggi guidati, una volta installato, basterà scaricare l' esecutivo di TimescaleDB e avviare il setup.exe .

Per quanto riguarda Linux Ubuntu i passaggi per installare postgreSQL sono:

- Creare il file /etc/apt/sources.list.d/pgdg.list e aggiungre una riga per il repository

```
deb http://apt.postgresql.org/pub/repos/apt/ YOUR_UBUNTU_VERSION_HERE-pgdg main
```

- Importare la chiave di firma del repository e aggiornare gli elenchi dei pacchetti

```
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add
sudo apt-get update
```

- Ubuntu include PostgreSQL di default, per installarlo bisogna usare il comando apt-get:

```
apt-get install postgresql-11
```

La repository contiene molti pacchetti incusi addons di terze parti, I pacchetti più famosi ed importanti sono:

postgresql-client-11	client libraries and client binaries
postgresql-11	core database server
postgresql-contrib-9.x	additional supplied modules
libpq-dev	libraries and headers for C language frontend development
postgresql-server-dev-11	libraries and headers for C language backend development
pgadmin4	pgAdmin 4 graphical administration utility

Per il dowload e l' istallazione di timescaleDB basta seguire i seguenti passaggi:

```
# Add our PPA
sudo add-apt-repository ppa:timescale/timescaledb-ppa
sudo apt-get update

# Now install appropriate package for PG version
sudo apt install timescaledb-postgresql-11
sudo timescaledb-tune
```

infine bisogna riavviare postgres:

```
# Restart PostgreSQL instance
sudo service postgresql restart
```

Creazione e settaggio di un database Timescale

Una volta installati PostgreSQL e TimescaleDB possiamo procedere alla creazione del database, il database si può creare in maniera semplice tramite interfaccia grafica con pgAdmin4, altrimenti da terminale.

Per creare un db da terminale bisogna entrare su postgreSQL tramite il comando:

```
sudo -U postgres psql postgres
```

dopodichè bisogna digitare il comando per creare un nuovo db:

```
CREATE DATABASE nomedb;
```

per vedere la lista dei database creati basta digitare \l

```
postgres=# create database DBtesi;
CREATE DATABASE
postgres=# \l
                                  Lista dei database
   Nome    | Proprietario | Codifica | Ordinamento |    Ctype    | Privilegi di accesso
-----------+--------------+----------+-------------+-------------+-----------------------
 dbtesi    | postgres     | UTF8     | it_IT.UTF-8 | it_IT.UTF-8 |
 nyc_data  | postgres     | UTF8     | it_IT.UTF-8 | it_IT.UTF-8 |
 postgres  | postgres     | UTF8     | it_IT.UTF-8 | it_IT.UTF-8 |
 prova2    | postgres     | UTF8     | it_IT.UTF-8 | it_IT.UTF-8 |
 provats   | postgres     | UTF8     | it_IT.UTF-8 | it_IT.UTF-8 |
 template0 | postgres     | UTF8     | it_IT.UTF-8 | it_IT.UTF-8 | =c/postgres          +
           |              |          |             |             | postgres=CTc/postgres
 template1 | postgres     | UTF8     | it_IT.UTF-8 | it_IT.UTF-8 | =c/postgres          +
           |              |          |             |             | postgres=CTc/postgres
 test_db   | postgres     | UTF8     | it_IT.UTF-8 | it_IT.UTF-8 |
(8 righe)
```

Nel mio caso, per fare i test utilizzerò uno schema e dei dati già esistenti, quindi, una volta creato il database, importiamo il file dump che contiene lo schema del database e i dati contenuti al suo interno.

Per fare ciò apriamo un nuovo terminale nella stessa directory del file dump e digitiamo il seguente comando:

```
psql -U postgres -X -d nomedb -h localhost < dbimportato.dump
```

l' ultimo step dopo aver creato il db e importato lo schema e i dati è quello di estendere il database Postgres a TimescaleDB.

Dal terminale psql ci connettiamo al db tramite il comando \c nomedb e applichiamo l' estensione:

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

```
dbtesi=# CREATE EXTENSION IF NOT EXISTS timescaledb CASCADE;
ATTENZIONE:
WELCOME TO
 _____ _                               _     ____________
|_   _(_)                             | |    |  _  \ ___ \
  | |  _ _ __ ___   ___  ___  ___ __ _| | ___| | | | |_/ /
  | | | |  _ ` _ \ / _ \/ __|/ __/ _` | |/ _ \ | | | ___ \
  | | | | | | | | |  __/\__ \ (_| (_| | |  __/ |/ /| |_/ /
  |_| |_|_| |_| |_|\___||___/\___\__,_|_|\___|___/ \____/
               Running version 1.5.1
For more information on TimescaleDB, please visit the following links:

 1. Getting started: https://docs.timescale.com/getting-started
 2. API reference documentation: https://docs.timescale.com/api
 3. How TimescaleDB is designed: https://docs.timescale.com/introduction/architecture

Note: TimescaleDB collects anonymous reports to better understand and assist our users.
For more information and how to disable, please see our docs https://docs.timescaledb.co

CREATE EXTENSION
dbtesi=#
```

Adesso che abbiamo il nostro database Timescale possiamo cominciare a fare i primi test.

Creazione di un Hypertable

Il database in questione è composto da varie tabelle, alcune sono delle classiche tabelle relazionali che descrivono l' ambiente come ad esempio i sensori, gli utenti e altri soggetti generici.
Altre invece contengono i dati time-series e sono composte da un campo id, un campo timestamp e vari campi che contengono i valori letti.
Il nostro primo passo sarà quindi quello di convertire la tabella contenente i timestamp (nel nostro caso la tabella si chiama sensor_channel_value_prova2) in un Hypertable di timescaleDB che chiameremo prova_ts2.
Poiché non si può direttamente convertire una tabella contenente dati in hypertable, sarà necessario creare una nuova tabella vuota con la stessa struttura e con le stesse costanti della vecchia tabella. Un metodo semplice per fare ciò è utilizzare il comando LIKE:

```
CREATE TABLE new_table (LIKE old_table INCLUDING DEFAULTS INCLUDING CONSTRAINTS INCLUDING INDEXES);
```

trasformiamo la nuova tabella in hypertable:

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

ps: il campo time indica il nome del campo timestamp della tabella
dopodichè copiamo i dati nella nuova hypertable:

```
INSERT INTO new_table SELECT * FROM old_table;
```

alcuni errori che ho commesso e risolto sono stati:
- convertire una tabella postgres in un hypertable di timescale senza che il timestamp fosse chiave primaria.
- durante la copia dei dati, poiché abbiamo posto il campo time chiave primaria nell' hypertable, è necessario non passare valori duplicati, io ho risolto usando il comando DISTINCT:

```
INSERT INTO new_table SELECT distinct on (time) * FROM old_table;
```

come risultato finale, erano presenti due tabelle contenente gli stessi dati nello stesso database:
-una tabella postgreSQL denominata **sensor_channel_value_prova2**
- un hypertable timescaleDB denominato **prova_ts2**

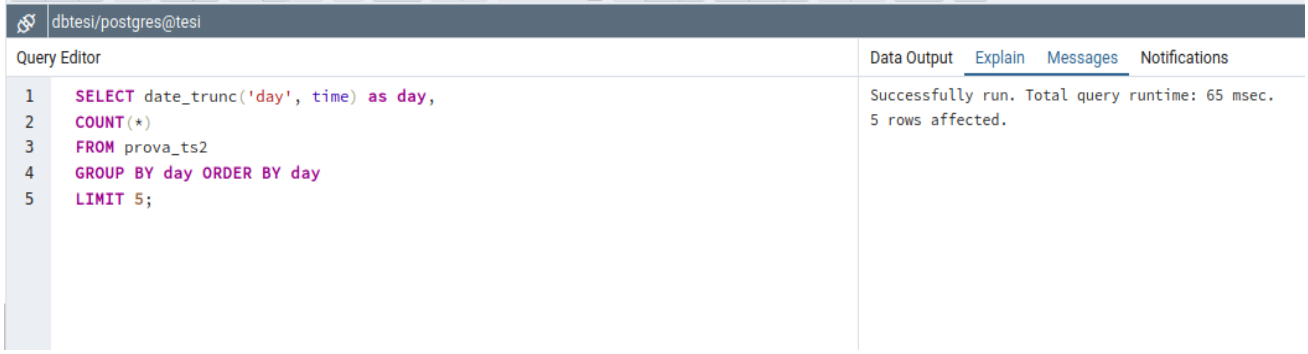
PostgreSQL vs TimescaleDB

Query n.1

```
SELECT date_trunc('day', time) as day
FROM prova_ts2
GROUP BY day ORDER BY day;
```

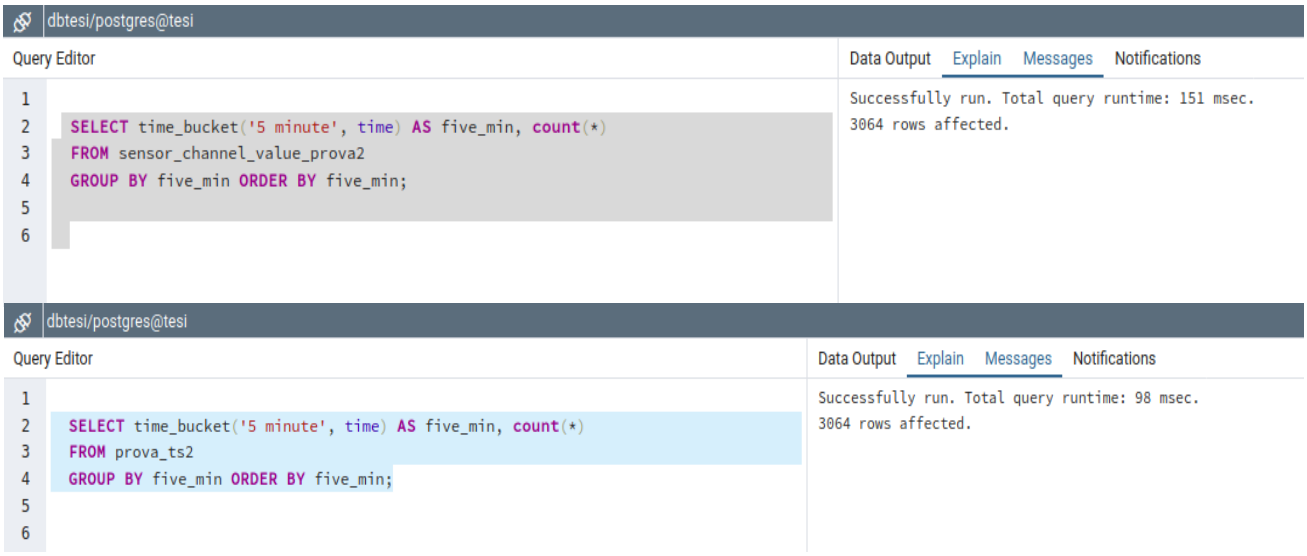
Query n.2

```
SELECT date_trunc('day', time) as day,
COUNT(*)
FROM prova_ts2
GROUP BY day ORDER BY day
LIMIT 5;
```



Query n.3

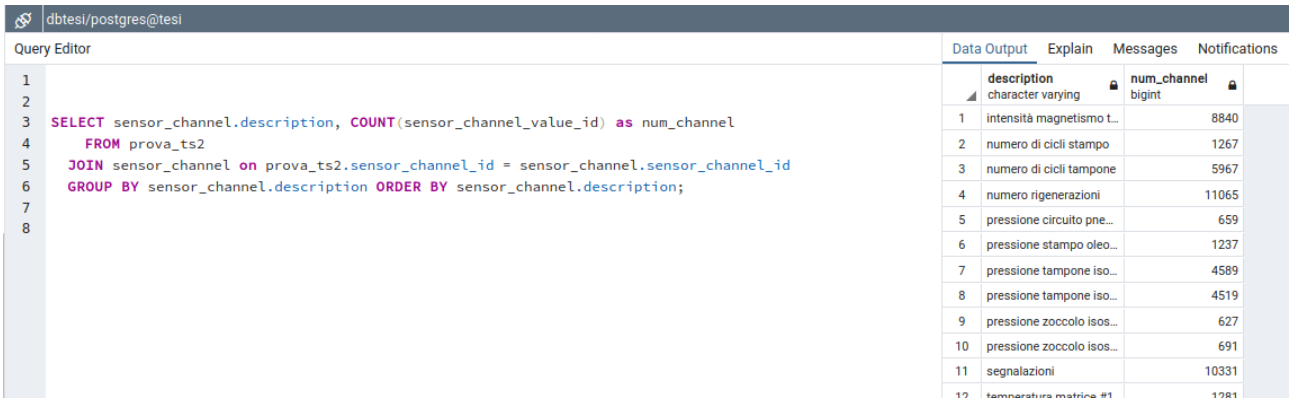
```
SELECT time_bucket('5 minute', time) AS five_min, count(*)
FROM prova_ts2
GROUP BY five_min ORDER BY five_min;
```



Query n.4

nell' hypertable prova_ts2 è presente una forreign key a una tabella relazionale di postgres, con questa query vediamo come timescale riesca a supportare anche i JOIN con tabelle relazionali.

```
SELECT sensor_channel.description, COUNT(sensor_channel_value_id) as num_channel
FROM prova_ts2
JOIN sensor_channel on prova_ts2.sensor_channel_id = sensor_channel.sensor_channel_id
GROUP BY sensor_channel.description ORDER BY sensor_channel.description;
```



Utilizzo di dati geospaziali tramite postGIS

TimescaleDB è un estensione di PostgreSQL, ciò significa che possiamo scaricare qualsiasi estensione di PostgresSQL e utilizzarla in maniera congiunta con TimescaleDB.

L' obbiettivo di questo test è dimostrare la possibilità di utilizzare dati geospaziali in un hypertable di Timescale, consentendoci così di suddividere i dati non solo a livello temporale ma anche a livello geospaziale.

Per prima cosa bisogna scaricare PostGis.

Il sito ufficiale dove effettuare il download è il seguente: http://postgis.net/install/

Se si utilizza Ubuntu, dando per scontato che PostgreSQL sia già installato, i comandi per scaricare e installare PostGIS sono I seguenti:

```
sudo apt-get update
sudo apt install postgresql-10-postgis2.4
sudo apt install postgresql-10-postgis-scripts
```

una volta installato torniamo al database e aggiungiamo l' estensione:

```
CREATE EXTENSION postgis;
```

poiché nel dataset con cui ho lavorato non sono presenti coordinate, le ho aggiunte per fini descrittivi.

Aggiungiamo quindi le colonne di latitudine e longitudine all' Hypertable:

```
ALTER TABLE prova_ts2 ADD longitude numeric;
ALTER TABLE prova_ts2 ADD latitude numeric;
```

riempiamo le colonne:

```
UPDATE prova_ts2
SET longitude = 36.949889
where mold_id like '20' ;

UPDATE prova_ts2
SET latitude = 14.537111
where mold_id like '20' ;

UPDATE prova_ts2
SET longitude = 44.645998
where mold_id like '21' ;


UPDATE prova_ts2
SET latitude = 10.925391
where mold_id like '21' ;
```

ho diviso in maniera casuale le misurazioni assegnandole a due posizioni differenti.

Infine ho creato la colonna contenente la posizione geospaziale formata dalla longitudine e latitudine:

```
ALTER TABLE prova_ts2 ADD COLUMN geom geometry(POINT,2163);

UPDATE rides SET geom = ST_Transform(ST_SetSRID(ST_MakePoint(longitude,
latitude),4326),2163);
```

Notifications

mold_id character varying (10)	ip character varying (30)	motherboard_id character varying	longitude numeric	latitude numeric	geom geometry
20	93.68.255.35	8892	36.949889	14.537111	0101000020730800...
20	93.68.255.35	8892	36.949889	14.537111	0101000020730800...
20	93.68.255.35	8892	36.949889	14.537111	0101000020730800...
mold_id character varying (10)	**ip** character varying (30)	**motherboard_id** character varying	**longitude** numeric	**latitude** numeric	**geom** geometry
21	93.68.255.35	8892	44.645998	10.925391	0101000020730800...
21	93.68.255.35	8892	44.645998	10.925391	0101000020730800...
21	93.68.255.35	8892	44.645998	10.925391	0101000020730800...
21	93.68.255.35	8892	44.645998	10.925391	0101000020730800...
21	93.68.255.35	8892	44.645998	10.925391	0101000020730800...
21					

Infine ho effettuato una semplice Query sull' Hypertable che torna i Timestamp delle misurazioni che sono avvenute entro 40 chilometri dalla posizione 44.493804, 11.342798

```
SELECT sensor_channel_id,time
FROM prova_ts2
WHERE ST_Distance(geom,  ST_Transform  (ST_SetSRID
(ST_MakePoint(44.493804,11.342798),4326),2163)) < 40000
```

Installazione di InfluxDB

Per installare InfluxDB su Ubuntu bisogna aggiungere la repository di InfluxData:

```
wget -qO- https://repos.influxdata.com/influxdb.key | sudo apt-key add -
source /etc/lsb-release
echo "deb https://repos.influxdata.com/${DISTRIB_ID,,} ${DISTRIB_CODENAME} stable" | sudo
tee /etc/apt/sources.list.d/influxdb.list
```

dopodichè, si può installare e avviare InfluxDB tramite i comandi:

```
sudo apt-get update && sudo apt-get install influxdb
sudo service influxdb start
```

Importazione di un database di prova su InfluxDB

Per i test comparativi tra InfluxDB e TimescaleDB ci avvaliamo di un database di prova chiamato

NOAA_data fornito dagli stessi programmatori di influx.

Dal terminale, scarichiamo il file di testo contenete i dati del database:

```
curl https://s3.amazonaws.com/noaa.water-database/NOAA_data.txt -o NOAA_data.txt
```

dopodichè ci connettiamo a influx tramite il comando

$influx -precision rfc3339

creiamo il database vuoto:

```
> CREATE DATABASE NOAA_water_database
> exit
```

E importiamo i dati su InfluxDB tramite il CLI:

```
influx -import -path=NOAA_data.txt -precision=s -database=NOAA_water_database
```

per provare che il database sia stato importato correttamente possiamo utilizzare il comando SHOW

```
> SHOW measurements
name: measurements
------------------
name
average_temperature
h2o_feet
h2o_pH
h2o_quality
h2o_temperature
```

Migrazione dei dati: da InfluxDB a TimescaleDB

l' obbiettivo di questo paragrafo è esportare il database di prova NOAA_data da influxDB a TimescaleDB in modo da avere lo stesso database sui due DB Server e dopodichè attuare le stesse query e confrontare i tempi di esecuzione.

Per migrare il database da InfluxDB a timescaleDB utilizziamo un software che si chiama Outflux.

È possibile scaricare Outflux dal repository di Github: https://github.com/timescale/outflux/releases

Dopo aver scaricato la versione adatta al proprio sistema operativo (nel nostro caso Ubuntu) bisogna estrarre il file e aprire un terminale nella cartella d' estrazione.

Si può utilizzare il comando $ ./outflux - -help per avere una breve spiegazione di cosa può fare Outflux e quali sono i comandi da utilizzare.

Per migrare lo schema del database, si può utilizzare il comando schema-transfer, nel nostro caso:

```
$ ./outflux schema-transfer NOAA_water_database --input-server=http://localhost:8086 --output-conn ='dbname=test_db user=postgres password=***'
```

Per migrare i dati, si può utilizzare il comando migrate, nel nostro caso:

```
$ ./outflux migrate NOAA_water_database --input-server=http://localhost:8086 –output-conn='dbname=test_db user=postgres password=***' --schema-strategy=DropAndCreate
```

Visualizzazione del tempo di esecuzione delle Query in InfluxDB

Ora che abbiamo implementato il database anche in TimescaleDB dobbiamo trovare un modo per misurare il tempo di esecuzione delle query nei due DB Server.
Per quanto riguarda TimescaleDB, continueremo ad utilizzare l' interfaccia grafica di pgAdmin4.
Per quanto riguarda InfluxDB, avremo bisogno di accedere al file dei log di Influx tramite il comando sudo journalctl -u influxdb.service

dentro il file dei log, le informazioni delle query sono salvate in un formato HTTP, un log di una query d' esempio può essere:

172.13.8.13,172.39.5.169 - -

[21/Jul/2019:03:01:27 +0000]

"GET

user

/query?db=metrics&q=SELECT+MEAN%28value%29+as+average_cpu %2C+MAX%28value%29+as+peak_cpu+FROM+%22foo.load%22+WHERE+time+%3E%3D+now%28%29+-+1m+AND+org_id+%21%3D+%27%27+AND+group_id+%21%3D+%27%27+GROUP+BY+org_id%2Cgroup_id HTTP/1.0" 200 11450

"-"

"Baz Service"

d6ca5a13-at63-11o9-8942-000000000000

9337349

Component	Example
Host	`172.13.8.13,172.39.5.169`
Time of log event	`[21/Jul/2019:03:01:27 +0000]`
Request method	`GET`
Username	`user`
HTTP API call being made*	`/query?db=metrics%26q=SELECT%20used_percent%20FROM%20%22telegraf.autogen.mem%22%20WHERE%20time%20%3E=%20now()%20-%201m%20`
Request protocol	`HTTP/1.0`
HTTP response code	`200`
Size of response in bytes	`11450`
Referrer	`-`
User agent	`Baz Service`
Request ID	`d4ca9a10-ab63-11e9-8942-000000000000`
Response time in microseconds	`9357049`

tra tutti i dati forniti, quello che ci interessa è l' ultimo campo "response time in microseconds" che torna il tempo di esecuzione della query in microsecondi

InfluxDB vs TimescaleDB

Tutte le query sono state effettuate sul database NOAA_water_database.

Query n.1

Select semplice

Database:	InfluxDB	TimescaleDB
Query:	`SELECT *` `FROM "h2o_quality"`	`SELECT *` `FROM h2o_qualityTS`
Execution query runtime:	130 ms	80 ms

Query n.2 e n.3

select con clausola where

Database:	InfluxDB	TimescaleDB

Query:	`select *`
	`FROM "h2o_quality"`
	`WHERE location =`
	`'santa_monica'`
		`SELECT *`
		`FROM h2o_qualityTS`
		`WHERE location like`
		`'santa_monica'`

Execution query runtime:	65 ms	90 ms

Il motivo per cui in questa seconda query InfluxDB si è dimostrato più veloce di TimescaleDB è da attribuire al campo location, location infatti è stato definito come tag.

Ricordiamo che in Influx i campi tag sono indicizzati e il file degli indici è in-memory e ciò rende le query sui tag estremamente veloci.

Se infatti riproviamo a fare la stesa query questa volta però applicando la clausola where a un campo value di Influx, che ricordiamo non essere indicizzato e quindi molto più lento, notiamo che il tempo di esecuzione della query di influx peggiora notevolmente

Database:	InfluxDB	TimescaleDB

Query:	`select *`
	`FROM "h2o_quality"`
	`WHERE “index” > 40`
		`SELECT *`
		`FROM h2o_qualityts`
		`WHERE index > 40`

Execution query runtime:	105 ms	90 ms

Query n.4

Utilizzo del group by

Database:	InfluxDB	TimescaleDB
Query:	`SELECT MEAN("water_level") FROM "h2o_feet" GROUP BY "location"`	`SELECT AVG(water_level) FROM h2o_feetTS GROUP BY location`
Execution query runtime:	25 ms	60 ms

Query n.5

Utilizzo del group by ad intervalli di tempo

Database:	InfluxDB	TimescaleDB
Query:	`SELECT MEAN("water_level") FROM "h2o_feet" WHERE "location"='coyote_creek' AND time >= '2015-08-18T00:06:00Z' AND time <= '2015-08-18T00:54:00Z' GROUP BY time(18m)`	`SELECT TIMESTAMP WITH TIME ZONE 'epoch' + INTERVAL '1 second' * round(extract('epoch' from time) / 1080) * 1080,AVG(water_level) FROM h2o_feetTS WHERE location='coyote_creek' AND time >= '2019-08-18T00:06:00Z' AND time <= '2019-08-18T00:54:00Z' GROUP BY round(extract('epoch' from time) / 1080)`
Execution query runtime:	3 ms	60 ms

oltre ad avere un tempo di esecuzione della query quasi nullo rispetto a Timescale, in Influx scrivere una query che raggruppasse il tempo in intervalli si è rivelato nettamente più semplice. Il motivo di questa superiorità è da attribuire alle funzioni implementate da influx (in questo caso time()) create appositamente per la gestione dei dati time-series.

Query n.6

utilizzo dell' ORDER BY

Database:	InfluxDB	TimescaleDB
Query:	`SELECT "water_level" FROM "h2o_feet" WHERE "location" = 'santa_monica' ORDER BY time DESC`	`SELECT water_level FROM h2o_feetTS WHERE location = 'santa_monica' ORDER BY time DESC`
Execution query runtime:	80 ms	90 ms

Anche se il tempo di esecuzione è sostanzialmente molto simile, timescale ha un netto vantaggio rispetto a influxDB per quanto riguarda l' utilizzo dell' order by.

In influxDB infatti è possibile ordinare le tabelle solo in base al tempo di inserimento e non in base ad altri campi. Una query del tipo:

```
SELECT water_level
FROM h2o_feetTS
WHERE location = 'santa_monica'
ORDER BY water_level DESC
```

è possibile solo in timescale.

Considerazioni finali

In generale le query effettuate hanno confermato le osservazioni fatte nella seconda parte di tesi: sfruttando l' indicizzazione in-memory, Influx riesce ad essere superiore a Timescale quando si tratta di dataset non troppo grandi, inoltre la struttura noSQL e la creazione di funzioni apposite per la gestione dei dati time-series permettono ad Influx di eseguire alcune funzioni specifiche (una su tutte il GROUP BY) in pochissimo tempo.

Di contro però, Influx ha dimostrato dei limiti strutturali come ad esempio l' impossibilità di utilizzare a pieno l' ORDER BY o il peggioramento del tempo di esecuzione se si effettuano query sui campi e non sui tag o query che ritornano dataset molto grandi.

Glossario

continuous query (CQ)
An InfluxQL query that runs automatically and periodically within a database. Continuous queries require a function in the SELECT clause and must include a GROUP BY time() clause.

Data ingestion
Data ingestion is the process of obtaining and importing data for immediate use or storage in a database. To ingest something is to "take something in or absorb something."

In-memory database
An in-memory database is a type of nonrelational database that relies primarily on memory for data storage, in contrast to databases that store data on disk or SSDs. In-memory databases are designed to attain minimal response time by eliminating the need to access disks.

Sitografia

Classifica time-seriesDB:
https://db-engines.com/en/ranking/time+series+dbms
timescaleDB:
https://www.timescale.com/
InfluxDB:
https://www.Influxdata.com/
Prometheus:
https://prometheus.io/
Accumulo:
https://accumulo.apache.org/
Redis:
https://redis.io/
InfluxDB docs:
https://docs.influxdata.com
timescaleDB docs:
https://docs.timescale.com
shards and retention policies:
https://www.influxdata.com/blog/influxdb-shards-retention-policies/
timescaleDB wide-table model, narrow-table model:
https://docs.timescale.com/latest/introduction/data-model
Prometheus-timescaleDB
https://docs.timescale.com/latest/tutorials/prometheus-adapter
influxDV vs timescaleDB:
https://blog.timescale.com/time-series-data-why-and-how-to-use-a-relational-database-instead-of-nosql-d0cd6975e87c
https://severalnines.com/database-blog/which-time-series-database-better-timescaledb-vs-influxdb
Authentication and authorization in InfluxDB
https://docs.influxdata.com/influxdb/v1.7/administration/authentication_and_authorization/
InfluxQL Continuous Queries
https://docs.influxdata.com/influxdb/v1.7/query_language/continuous_queries/
Informazioni sul design e compromessi di InfluxDB
https://docs.influxdata.com/influxdb/v1.7/concepts/insights_tradeoffs/
Redis data type
https://redis.io/topics/data-types-intro
redis architecture
http://qnimate.com/overview-of-redis-architecture/
Accumulo user manual
https://accumulo.apache.org/1.9/accumulo_user_manual.html