UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Installazione e sperimentazione di NoSQL datastore mediante Yahoo! Cloud Serving Benchmarking

Studenti:

Ascani Christian
Sopranzetti Lorenzo
Tiseni Lorenzo

Docente:

Prof.ssa Diamantini Claudia

A.A. 2022-2023

Sommario

Indice delle Figure

1. Introduzione

Nell'era digitale, l'ottimizzazione delle prestazioni e dei servizi web ha assunto un ruolo di primaria importanza per soddisfare le esigenze degli utenti e garantire la competitività delle organizzazioni. In questo contesto, la gestione efficiente delle risorse e la selezione delle tecnologie più adeguate sono diventate prioritarie per le aziende operanti nel settore dell'informatica. Infatti, le applicazioni web moderne si trovano ad affrontare sfide complesse che includono la gestione di enormi quantità di dati, la minimizzazione della latenza e la scalabilità, in modo da gestire picchi di traffico spesso imprevedibili.

Innanzitutto, il sistema di gestione di basi di dati (DBMS) è un sistema software che deve essere in grado di gestire collezioni di dati grandi, persistenti, condivise, e di garantire privatezza, affidabilità, efficacia. Esistono diverse tipologie di database, che si differenziano per la struttura dei dati, il supporto alle transazioni, la scalabilità e la coerenza. In tale contesto, la scelta tra i database relazionali (comunemente noti come SQL) e i database NoSQL assume un ruolo decisivo.

I database relazionali sono basati sul modello relazionale, che rappresenta i dati sottoforma di tabelle composte da righe (record) e colonne (attributi), tra cui una viene detta chiave primaria e consente di identificare una singola riga. Le tabelle sono collegate tra loro tramite chiavi esterne, che stabiliscono le relazioni tra i dati. I database relazionali offrono il supporto a transazioni ACID (*Atomicity*, *Consistency*, *Isolation*, *Durability*), che garantiscono le seguenti proprietà:

- Atomicità: una transazione è un'unità indivisibile di operazioni sui dati, per cui non è ammesso eseguirne una a metà, lasciando la base di dati in uno stato intermedio.
- Consistenza: una transazione deve rispettare i vincoli di integrità, ovvero, se lo stato iniziale è corretto, allora al termine della transazione lo stato (finale) dovrà essere corretto a sua volta.
- Isolamento (concorrenza): una transazione non influenza e non risente degli effetti di altre transazioni concorrenti.
- Durabilità (persistenza): gli effetti di una transazione sono permanenti anche in caso di guasti e/o errori.

I database relazionali sono tradizionalmente utilizzati per applicazioni che richiedono una rigorosa coerenza dei dati, come sistemi bancari o applicazioni di gestione aziendale. Essi presentano una struttura dei dati ben definita e normalizzata, la quale caratteristica contribuisce a evitare la ridondanza e l'inconsistenza dei dati, garantendo che le informazioni siano organizzate in modo coerente e affidabile. Inoltre, i database relazionali sono noti per il loro linguaggio di interrogazione standardizzato, ovvero SQL (Standard Query Language).

Questo linguaggio consente di eseguire query complesse e flessibili sui dati, rendendo più semplice e potente l'accesso alle informazioni del database. Tuttavia, i DBMS relazionali presentano anche degli svantaggi, soprattutto con applicazioni web su larga scala e ad alta concorrenza. Uno di questi svantaggi è dato dalla scalabilità orizzontale, ovvero la capacità di distribuire i dati su più nodi per aumentare le prestazioni e la disponibilità. Questa limitazione può diventare un ostacolo significativo quando si cerca di gestire grandi volumi di dati o un traffico molto elevato. Anche la struttura rigida rappresenta un problema in molti scenari odierni, in quanto lo schema viene stabilito in fase di progettazione e ciò potrebbe limitare la flessibilità delle applicazioni. Infatti, i database relazionali hanno una scarsa adattabilità a dati non strutturati o semi-strutturati, sempre più diffusi nelle applicazioni web moderne, nei social network e nell'Internet of Things (IoT).

I database NoSQL sono una categoria diversa rispetto ai sistemi SQL, più eterogenea e che non segue il modello relazionale. Essi offrono soluzioni alternative per la gestione dei dati, di solito su architetture cluster. I database NoSQL si distinguono per la struttura dei dati e si possono annoverare: modelli key-value, document store, modelli column-family, modelli a grafo. I database NoSQL rinunciano al supporto verso le transazioni ACID in favore di un approccio BASE (*Basically Available, Soft state, Eventual consistency*). In dettaglio, un sistema di questo tipo deve essere sempre disponibile (anche in caso di guasti), lo stato del sistema può cambiare nel tempo senza perdere la sua funzionalità e, a seguito dell'aggiornamento della replica di un dato, la modifica si deve propagare a tutte le altre repliche, portando il database ad assumere uno stato consistente entro un certo periodo di tempo. I vantaggi dei DBMS NoSQL sono: la scalabilità orizzontale, ovvero la capacità di distribuire i dati su più nodi mantenendo performance stabili; la flessibilità della struttura dei dati, che permette di aggiungere o modificare i dati senza impattare le prestazioni; l'adattabilità a dati non strutturati o semi-strutturati, che sono sempre più diffusi nelle applicazioni web, nell'e-commerce e nell'IoT. D'altro canto, essi presentano anche degli svantaggi, poiché manca uno standard di interrogazione, che rende difficile eseguire query complesse e flessibili sui dati. Inoltre, si rinuncia parzialmente alle proprietà transazionali, compromettendo l'affidabilità e l'integrità dei dati, e aumentando la complessità nella gestione della consistenza dei dati distribuiti su più nodi.

I database NoSQL assumono sempre più importanza dato che le attuali tendenze dei Big Data e del Cloud Computing stanno portando le aziende ad abbandonare i database relazionali verso database non relazionali. Questi paradigmi, infatti, richiedono, molto spesso, l'utilizzo di ampie reti distribuite di nodi. Questa specifica si integra molto bene con le caratteristiche peculiari dei sistemi NoSQL come la loro scalabilità e flessibilità.

Affianco ai database sopra descritti è nato anche il movimento NewSQL. Esso cerca di combinare le caratteristiche dei database relazionali e dei database non relazionali, garantendo scalabilità orizzontale e supporto a transazioni ACID. Varie soluzioni sono state applicate da un'ampia gamma di organizzazioni del settore, tra cui Amazon, Google, Yahoo e Meta, per processare enormi volumi di dati sui propri server.

1.1 Database benchmarking

Data la crescente variabilità e disponibilità di soluzioni DBMS sul mercato, è sempre più importante avere metodi validi per la misurazione delle qualità di queste soluzioni e delle caratteristiche peculiari di ognuna di esse, per poter individuare quella più adatta ad ogni specifico caso d'uso. In questo contesto, il benchmarking dei database diventa un'operazione spesso essenziale per effettuare delle scelte adeguate e basate su dati oggettivi. Esso è ormai un metodo consolidato per analizzare e confrontare le caratteristiche dei DBMS, con un focus particolare sulle prestazioni e sulla scalabilità. Questa operazione ha dunque lo scopo principale di individuare le opportunità di miglioramento, confrontando, in un determinato contesto applicativo, le soluzioni disponibili misurandone le prestazioni.

Il primo passo cruciale in questo processo è la definizione del carico di lavoro che il sistema di gestione di database dovrà affrontare. Il carico di lavoro rappresenta il tipo di attività che il database dovrà gestire e varia notevolmente in base alle esigenze dell'applicazione. In scenari più semplici, il carico di lavoro può essere suddiviso in operazioni CRUD. Le caratteristiche di queste operazioni hanno un impatto significativo sulle metriche misurate durante il benchmark.

Nel passato, il panorama dei database SQL era relativamente limitato, con poche opzioni principali, e il benchmarking era più agevole, poiché le tecnologie erano più uniformi e omogenee. Successivamente, con l'introduzione dei database NoSQL, si è verificata una significativa evoluzione nel settore dei database. Questo cambiamento ha reso impossibile l'applicazione dei tradizionali benchmarking a questi nuovi tipi di database.

Questo caso di studio si concentrerà proprio sull'analisi di database NoSQL. L'obiettivo è quello di studiare e confrontare database NoSQL differenti su un medesimo caso d'uso per analizzare le loro caratteristiche e peculiarità. Per far questo si è dunque reso utile e necessario l'utilizzo di un framework apposito per l'analisi delle prestazioni di database, che consentisse, con una soluzione unica, il benchmarking di database NoSQL differenti. Il framework che si è scelto di utilizzare è YCSB.

Il Yahoo Cloud Serving Benchmarking (YCSB) è una suite di benchmarking open source per database, che permette di misurare le prestazioni di numerosi

DBMS moderni con operazioni semplici su dati generati sinteticamente. Inoltre, YCSB si presta al confronto delle prestazioni di database su infrastrutture distribuite, come il cloud. Esso può essere usato per confrontare database diversi e misurare le prestazioni di varie configurazioni di database, sotto diversi carichi di lavoro. Questa suite di benchmark fornisce un framework che automatizza i compiti essenziali in un processo di benchmarking, come la definizione di un carico di lavoro con i parametri essenziali, la connessione al database tramite i driver specifici, l'esecuzione del workload sul database, la raccolta e l'archiviazione dei dati riguardo alle prestazioni. Dunque, YCSB ha l'obiettivo di facilitare il confronto delle performance dei sistemi, in particolare, per carichi di lavoro che si differenziano da quelli misurati dai benchmark più tradizionali.

Nel caso specifico, questo strumento è stato utilizzato per valutare le prestazioni in termini di inserimento, aggiornamento, scansione e lettura di tre diversi database su tre diversi carichi di lavoro.

1.2 Database testati

Nell'effettuare la scelta dei database da testare, per avere un confronto più interessante, si è utilizzato come criterio la variabilità. Infatti, i tre database scelti sono tutti non relazionali, ma hanno una struttura per la gestione dei dati differente. I database scelti sono: Redis, MongoDB e Cassandra; di seguito, questi vengono approfonditi nel dettaglio.

1.2.1 Cassandra

Apache Cassandra è un database NoSQL distribuito di tipo super column-family. Esso è stato studiato per scalare in maniera efficace e lineare all'aumentare dei nodi del cluster che lo costituisce, senza la perdita di prestazioni o di tolleranza al rischio anche su hardware commodity. Nel caso in questione, poiché non si ha a disposizione un cluster di macchine, si è deciso di testare le prestazioni di Cassandra sul singolo nodo. Questo permette, inoltre, una comparazione migliore con gli altri database, anch'essi testati con una configurazione non distribuita.

1.2.2 MongoDB

MongoDB è un database NoSQL di tipo document-store molto flessibile con un'ampia community di sviluppatori e utilizzatori alle spalle, che lo rende un progetto vivo e in continua evoluzione. Esso è disponibile in due versioni: una è la versione Server, installabile su una singola macchina in maniera standalone; l'altra è la versione Atlas, cioè una versione cloud pay-per-use che è installabile su molti cloud provider. Ovviamente, non avendo a disposizione una piattaforma cloud su cui fare il deploy di MongoDB, in questo lavoro si è testata la versione Server, installata su una singola macchina. Il motivo principale per cui si è scelto di analizzare e confrontare con altri, un database come MongoDB è la sua facilità d'uso e il formato in cui sono salvati i dati, ovvero il formato BSON.

1.2.3 Redis

Redis, rilasciato per la prima volta nel 2009, è un esempio di *key-value storage*. Il modello chiave-valore è stato uno dei primi modelli NoSQL ed è nato con la caratteristica di privilegiare l'*availability* e il *partition tolerance*. In dettaglio, ogni valore immagazzinato è abbinato ad una chiave univoca che ne permette il recupero. Dunque, questo modello è effettivamente *schemaless*, ovvero il dato non ha una struttura riconoscibile e dovrà essere correttamente interpretato dall'applicazione. Generalmente, dallo store sono offerte alcune funzioni elementari, come l'inserimento di una coppia chiave-valore (*Put*), la lettura di un dato sulla base della chiave (*Get*), la cancellazione di una chiave (*Delete*). Riguardo all'interrogazione, una query restituisce un solo elemento, lavorando istanza per istanza. I vantaggi sono sicuramente la semplicità, la flessibilità, la memoria e la scalabilità. In particolare, Redis conserva i dati in memoria RAM, salvandoli in maniera persistente solo in un secondo momento, e proprio questo permette di ottenere ottime prestazioni in scrittura e lettura.

2. Dominio applicativo

Al fine di effettuare dei test sui database che fossero il più possibile attinenti a un caso reale, si è deciso di scegliere un particolare dominio applicativo sul quale basare tutti i test. In particolare, si è scelto come caso di studio di simulare un "e-commerce".

Effettuata questa scelta è stato necessario definire una struttura del dato e dei workload in maniera tale che fossero congruenti e attinenti al caso di studio. Per prima cosa, si è definita la struttura del dato (*products*) che verrà utilizzata per testare i database. Essa rappresenta, in maniera semplificata, le informazioni che è possibile trovare più comunemente in un articolo di un sito di e-commerce. Ogni record contiene dei campi descrittivi di vario genere e una chiave primaria per l'identificazione del tipo "prod1234".

Campo	Dimensione
name	1-100 byte
price	1-4 byte
availability	1-4 byte
description	1-100 byte
category	1-100 byte

La dimensione del campo viene scelta automaticamente in base a una distribuzione di probabilità che è possibile indicare nella definizione dei workload.

2.1 Workload

Per workload, o carico di lavoro, si intende un insieme di operazioni di lettura/scrittura, a cui si aggiungono ulteriori informazioni come la dimensione dei dati, il numero delle operazioni, la distribuzione delle richieste. YCSB consente di utilizzare i carichi di lavoro di default, che sono di diverso genere, o di creare dei carichi di lavoro personalizzati, tramite l'impostazione manuale di vari parametri e caratteristiche che essi dovranno avere. Questo consente di costruire più workload, ognuno con proprietà peculiari, in grado di testare le basi di dati in situazioni e condizioni di utilizzo differenti. La definizione dei workload è sicuramente l'operazione più importante per effettuare un corretto ed efficace test dei database.

Ci sono vari tipi di operazioni eseguibili su un data store tra cui: *Insert*, cioè l'inserimento di un nuovo record, *Update*, ovvero l'aggiornamento di un dato (sostituzione del valore di un campo), *Read*, che corrisponde o alla lettura di un campo scelto in maniera casuale o alla lettura di un intero record o oggetto, *Scan*, cioè una scansione ordinata che inizia da una chiave estratta casualmente (anche

il numero dei record da scansionare è scelto randomicamente). In ogni workload è necessario specificare il numero totale di operazioni e la percentuale delle operazioni da eseguire per ogni tipologia.

Come risulta anche dalla modalità con cui avviene la *Scan*, nel testing dei data store vengono prese diverse decisioni casuali durante l'esecuzione di un workload. Ad esempio, si sceglie casualmente quale operazione bisogna eseguire, quale record bisogna scrivere o leggere, quanti record devono essere scansionati e così via. Quindi, essendo la decisione di natura randomica, si devono considerare delle distribuzioni, che sono già preimpostate in YCSB. Queste distribuzioni intervengono specialmente nella scelta di quale record leggere, aggiornare o leggere per poi scrivere. Le alternative possibili offerte dal framework sono:

- *Uniform*: distribuzione uniforme, per cui la probabilità di scegliere uno o l'altro elemento è la medesima;
- *Zipfian*: alcuni record (una minoranza) hanno una probabilità di estrazione maggiore rispetto ad altri (la maggioranza), in linea con la *Power Law*;
- *Latest*: gli elementi che sono stati inseriti più di recente hanno una probabilità maggiore di essere selezionati;
- *Multinomial*: la probabilità di estrazione di un record viene specificata.

Riguardo al funzionamento, YCSB prevede due fasi per la valutazione delle prestazioni dei database, con diverse peculiarità:

- *Load*: la fase di caricamento prevede il popolamento delle base di dati con un numero di record impostato tramite l'apposito parametro. Questa fase consente di effettuare dei test su un database già popolato ricreando, in maniera realistica, le condizioni di utilizzo dei database.
- *Run*: questa fase può essere considerata come quella di esecuzione effettiva del test. In essa, vengono effettuate delle operazioni di scrittura, lettura, scansione sul database e vengono registrate le corrispondenti latenze al fine di eseguire analisi ulteriori.

Queste fasi possono essere personalizzate tramite alcuni parametri, come:

Parametro	Descrizione
recordcount	Numero di record da inserire nella fase di load e numero di record già presenti nel database durante la fase di run
operationcount	Il numero di operazioni effettuate durante la fase di run
threads	Numero di thread utilizzati per fare le richieste al database. Simula l'utilizzo

	contemporaneo del database da più client
readproportion	La proporzione delle operazioni che sono read
updateproportion	La proporzione delle operazioni che sono update
insertproportion	La proporzione delle operazioni che sono insert
readmodifywriteproportion	La proporzione delle operazioni che leggono e poi modificano
scanproportion	La proporzione delle operazioni che sono scan, ovvero letture di più record insieme
fieldlengthdistribution	Il tipo di distribuzione usata per scegliere la lunghezza dei campi
requestdistribution	Indica la distribuzione usata per scegliere come vengono fatte le richieste ai record

Modificando questi parametri, si sono definiti tre diversi carichi di lavoro per la valutazione delle performance dei database:

- Workload Read: carico principalmente in lettura (*read mostly*)
- Workload Update: carico principalmente in modifica (*update heavy*)
- Workload Insert: carico principalmente in inserimento (*insert mostly*)

2.1.1 Workload Read

Questo carico di lavoro simula scenari in cui gli utenti consultano le informazioni sui prodotti, le recensioni dei clienti o la cronologia degli ordini. Ciò significa che il benchmark YCSB misura la velocità e la latenza con cui il database può recuperare i dati richiesti dal client. In particolare, il sistema deve gestire in modo efficiente un grande volume di richieste di lettura di un articolo, alcune scansioni per il recupero di una serie di prodotti e alcuni aggiornamenti occasionali. Il workload si compone di tre tipologie di operazioni, ripartite, rispetto al numero di totale di operazioni, secondo le percentuali riportate di seguito:

- *Read*: 65%
- *Scan:* 30%
- *Update:* 5%

2.1.2 Workload Update

Questo carico di lavoro riproduce attività quali l'elaborazione degli ordini dei clienti, l'aggiornamento dei livelli di inventario o la gestione dei dettagli dei prodotti. Il workload è composto principalmente da operazioni di scrittura, per

analizzare come il data store gestisce un numero significativo di richieste di modifica di un record esistente. D'altra parte, si ha anche una piccola percentuale di operazioni di lettura, per capire come il data store riesce a processare alcune richiesti dei clienti in un periodo di forte sollecitazione. Di seguito si riporta le percentuali delle tipologie di operazioni del workload:

- *Update*: 80%
- *Read-Modify-Write*: 10% (aggiornamenti preceduti da una lettura del dato da modificare)
- *Read*: 10%

2.1.3 Workload Insert

Questo carico di lavoro simula prevalentemente l'inserimento di nuove informazioni nel database, come le registrazioni di nuovi prodotti al catalogo, durante un periodo in cui i clienti sono comunque operativi, cioè, visionano articoli e fanno ordini. In questa maniera si testa la capacità del database di inserire velocemente nuove informazioni per renderle subito visibili ai clienti che stanno operando. Per questo le tipologie di operazioni sono ripartite secondo le percentuali indicate di seguito:

- *Insert*: 80%
- *Read*: 10%
- *Update*: 10%

3. Test e risultati

Per tutti gli esperimenti è stata utilizzata una macchina con la seguente configurazione:

HARDWARE	SOFTWARE
CPU: Intel i5, 7th gen, 2 core RAM: 12 GB Disco Fisso: 200 GB	Ubuntu 20.04 64 bit YCSB 0.18.0 Redis 7.2.2 MongoDB 6.0.10 Cassandra 3.0.1

Strutturati i tre carichi di lavoro, si è reso necessario impostare alcuni parametri o proprietà di esecuzione, cioè il numero dei record da processare, il numero delle operazioni da eseguire e il numero dei threads. Si è deciso di non assegnare un unico valore ad ogni singola proprietà di esecuzione, ma di determinare un intervallo, in maniera tale da testare più scenari e ottenere una visione di insieme, per poi scendere nel dettaglio su alcuni casi particolari. Nella determinazione di questi intervalli si è proceduto in maniera euristica, cercando di scegliere delle dimensioni il più realistiche possibile, tenendo conto, però, dei limiti dell'hardware a disposizione.

Infatti, in base al workload si hanno le seguenti configurazioni:

	Workload Read	**Workload Update**	**Workload Insert**
Operationcount	200.000 400.000 600.000	20.000 40.000 60.000	10.000 25.000 50.000
Recordcount	100.000, 200.000, 300.000		
Threads	1, 2, 4, 6		

Avendo deciso di salvare i risultati grezzi, in seguito al caricamento e all'esecuzione è stato ottenuto un file csv con una struttura simile per ogni configurazione adottata, come mostrato in *Figura 1*.

READ latency raw data: op, timestamp(ms), latency(us)	[OVERALL], RunTime(ms), 236159
READ,1695942876487,274	[OVERALL], Throughput(ops/sec), 846.8870549079222
READ,1695942876487,140	[TOTAL_GCS_G1_Young_Generation], Count, 64
READ,1695942876487,108	[TOTAL_GC_TIME_G1_Young_Generation], Time(ms), 338
READ,1695942876500,136	[TOTAL_GC_TIME_%_G1_Young_Generation], Time(%), 0.14312391227943885
READ,1695942876500,126	[TOTAL_GCS_G1_Old_Generation], Count, 0
READ,1695942876500,133	[TOTAL_GC_TIME_G1_Old_Generation], Time(ms), 0
READ,1695942876500,135	[TOTAL_GC_TIME_%_G1_Old_Generation], Time(%), 0.0
READ,1695942876501,170	[TOTAL_GCs], Count, 64
READ,1695942876501,142	[TOTAL_GC_TIME], Time(ms), 338
READ,1695942876501,134	[TOTAL_GC_TIME_%], Time(%), 0.14312391227943885
READ,1695942876501,121	[READ], Total Operations, 130176
READ,1695942876510,161	[READ], Below is a summary of latency in microseconds:, -1
READ,1695942876511,142	[READ], Average, 72.35532663470993
READ,1695942876511,133	[READ], Min, 57
READ,1695942876517,122	[READ], Max, 5004
READ,1695942876522,111	[READ], p1, 60
READ,1695942876523,101	[READ], p5, 63
READ,1695942876523,107	[READ], p50, 68
READ,1695942876523,105	[READ], p90, 87
READ,1695942876528,179	[READ], p95, 88
	[READ], p99, 93
	[READ], p99.9, 140

Figura 1: Esempio di output

Infatti, nella prima parte del file, vengono salvati il tipo di operazione eseguita, il timestamp (in millisecondi) e la latenza relativa (in microsecondi) in maniera ordinata. Successivamente, si trovano delle misure aggregate, come il tempo totale di esecuzione, il throughput, la latenza media, ecc. Tuttavia, i file csv presentavano alcune righe accessorie, per cui è stato necessario procedere con una prima fase di ETL minima.

Prima di verificare le prestazioni per ogni workload e per ogni configurazione, si è posta l'attenzione sul tempo di esecuzione. Infatti, sono stati considerati sia il tempo medio sia il tempo mediano, in modo da visualizzare il comportamento dei tre database in presenza dello stesso carico di lavoro, aggregando le misure relative al tempo di esecuzione delle configurazioni testate. Come si può notare dalla *Figura 2*, con un workload read-intensive MongoDB riesce ad eseguire le operazioni più velocemente, mentre Redis risulta essere il più lento. Infatti, il primo DBMS riesce a terminare questo tipo di task in circa un quinto del tempo impiegato da Redis, mentre Cassandra si posiziona a metà. Per gli altri due carichi di lavoro, sia quello con più inserimenti sia quello con più aggiornamenti, Redis risulta essere quello con un tempo di esecuzione migliore, mentre Cassandra sembra essere quello più lento. In dettaglio, Redis impiega rispettivamente un quarto e un terzo del tempo necessario a Cassandra e a MongoDB per completare il task. Perciò, visti i diversi comportamenti tra letture e scritture, MongoDB sembra essere quello preferibile in presenza di sole letture, mentre Redis solo in presenza di inserimenti o aggiornamenti. Tuttavia, se si dovesse optare per un workload misto, MongoDB potrebbe essere la soluzione con un tradeoff migliore.

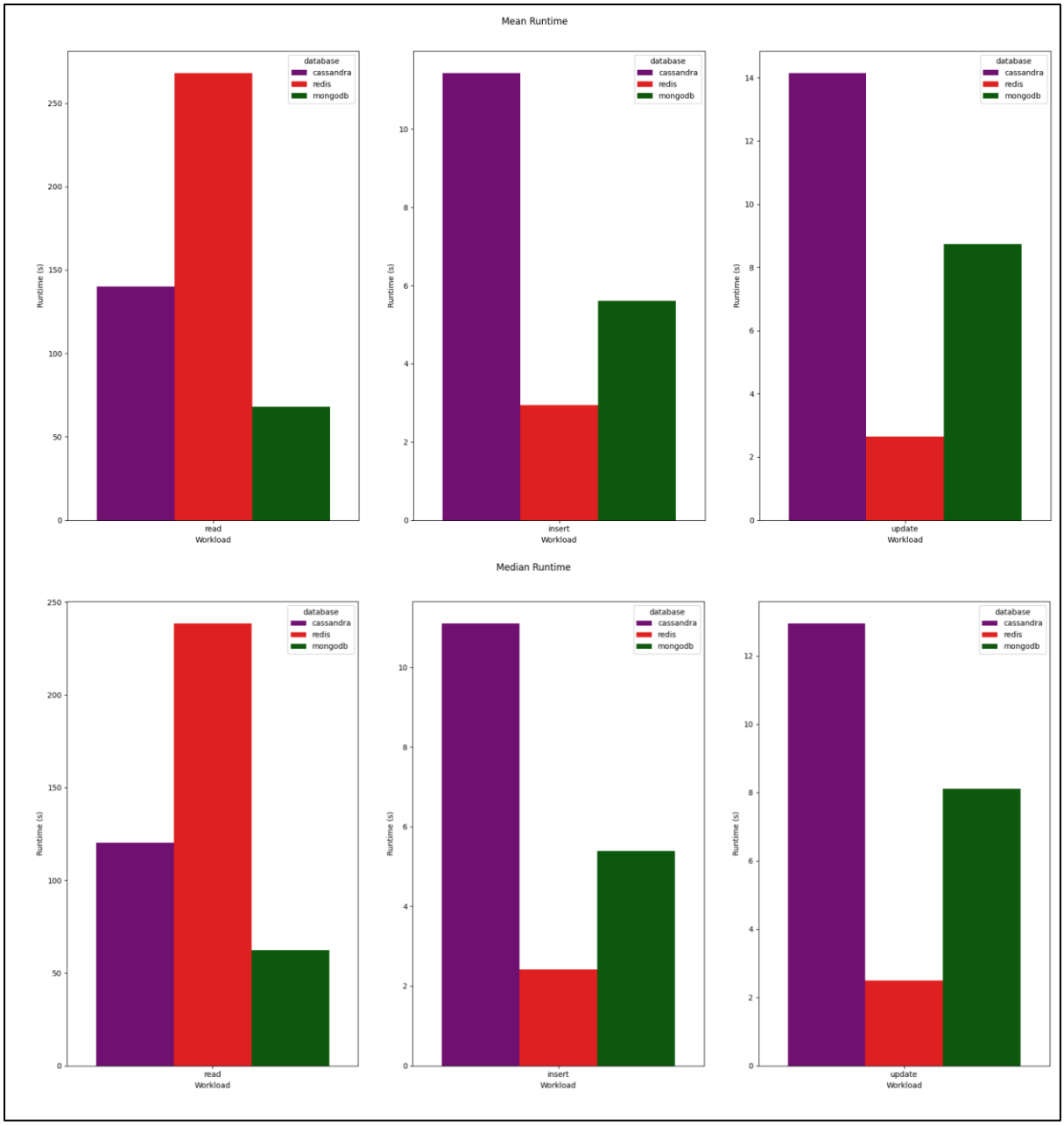

Figura 2: Tempo di esecuzione medio e mediano

Queste considerazioni iniziali sono state successivamente raffinate, analizzando in dettaglio i vari workload. Nei prossimi paragrafi, per ognuno di questi verranno confrontate le performance ottenute dai tre database, misurate in termini di troughput raggiunto e di latenza media delle operazioni. Prima si procederà con un'analisi aggregata, cercando di individuare delle tendenze comuni a tutti i database, e delle caratteristiche che li differenziano dal punto di vista delle performance. In seguito, si scenderà nel dettaglio con analisi più approfondite che permetteranno di mettere in luce aspetti interessanti riguardo alle prestazioni dei database nel portare a termine le operazioni del workload.

3.1 Workload Read

Nel workload in analisi, principalmente sono eseguite operazioni di *read* e di *scan*, con poche *update*, a simulare un'alta richiesta di visualizzazione di uno o più dati. A seguito dei vari test eseguiti relativamente a questo workload, si sono prodotte le seguenti mappe di calore finalizzate ad avere una visione di insieme sulle performance dei database:

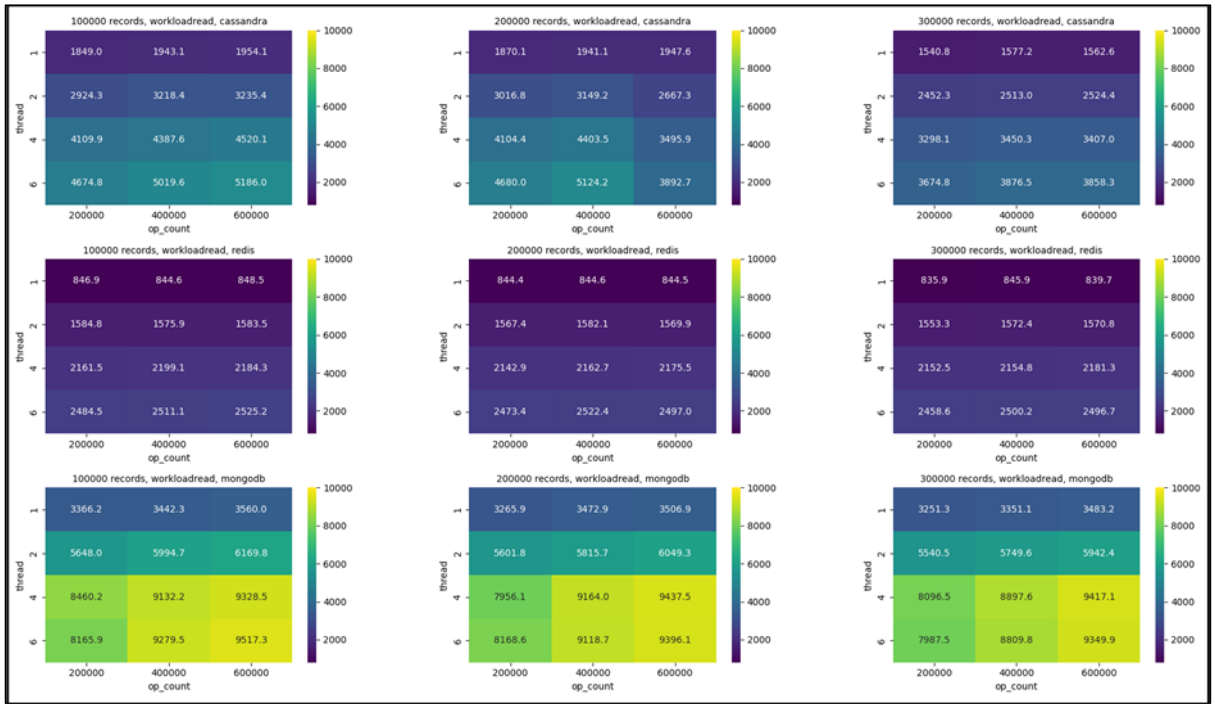

Figura 3: Heatmap Throughput Workload Read

Analizzando le mappe di calore nella *Figura 3* si possono individuare alcuni fenomeni interessanti. In tutti i database all'aumentare del numero di client che fanno richieste, aumenta il throughput, cioè nello stesso lasso di tempo i database riescono a servire più operazioni. Questo accade perché, essendoci più richieste, i database sono maggiormente stimolati e, nello stesso periodo di tempo, servono più operazioni. Confrontando le prestazioni al livello di throughput, si osserva che MongoDB fornisce prestazioni migliori per questo workload, con Cassandra che si posiziona in mezzo e Redis che offre prestazioni peggiori degli altri. Inoltre, mentre MongoDB sembra "saturare", ovvero raggiungere il throughput massimo con soli 4 o 6 client, gli altri due database sembrano avere un throughput che cresce in maniera meno rapida, senza arrivare a saturazione con pochi client. Al variare della taglia del database, ovvero del *record_count*, e del numero di operazioni eseguite, *op_count*, non sembrano esserci tendenze che indicano un aumento o una diminuzione del throughput.

Ai fini dell'analisi, è sembrato importante considerare, oltre al throughput, anche un altro indicatore delle prestazioni di un database, ovvero la latenza media. Di seguito si discutono le mappe di calore riportate in *Figura 4*:

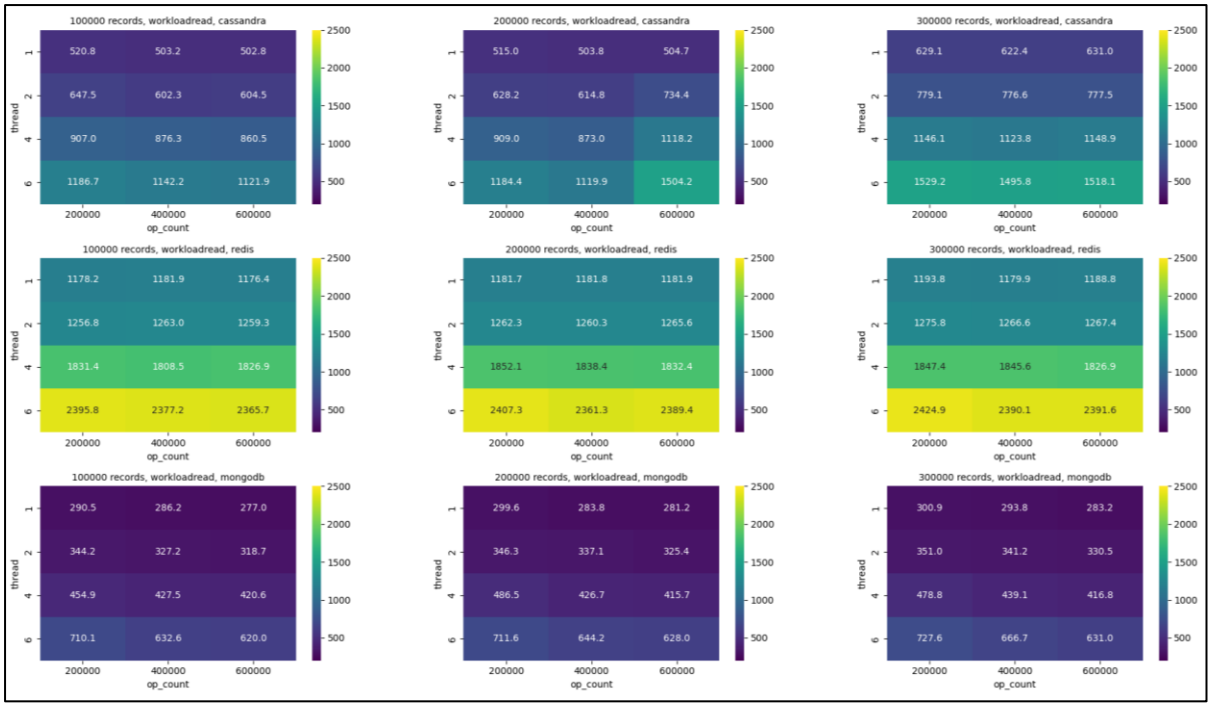

Figura 4: Heatmap Latenza Workload Read

In analogia con quanto riscontrato per il throughput, non si hanno delle tendenze chiare della latenza media, al variare della taglia del database e del numero di operazioni. Al contrario, si osserva che la latenza media aumenta all'aumentare del numero di client che richiedono operazioni, per tutti e tre i database considerati. Il fenomeno descritto potrebbe essere spiegato dal fatto che, anche se nello stesso lasso di tempo sono servite più operazioni, ottenendo un throughput maggiore, comunque le singole operazioni aspettano di più per essere eseguite, provocando un aumento della latenza. Per quanto riguarda il confronto tra i database, come prima, si osserva che MongoDB ha le prestazioni migliori servendo le operazioni in minor tempo, con Cassandra che rimane in mezzo e Redis che si comporta peggio degli altri due.

Dalle heatmap analizzate, si osserva un'incongruenza. Infatti, essendo Redis un key-value store, si presume che nelle letture sia molto più veloce di altri database, come MongoDB e Cassandra, che hanno un modello dati più strutturato. Al contrario, i test evidenziano che Redis ottiene prestazioni peggiori, nonostante abbia un modello dati più semplice e lavori in RAM. Per questo motivo, si è deciso di approfondire questo aspetto considerando le latenze delle singole tipologie di operazioni, come riportato nella *Figura 5*. Osservando l'immagine si deduce che la maggior parte delle letture puntuali e degli update fatti in Redis ha bassissima latenza, a differenza delle operazioni di *scan* che hanno una latenza nettamente maggiore. Quindi, Redis è molto veloce a fare letture e update di un singolo oggetto, mentre tende ad avere molte difficoltà nel completare in maniera rapida delle *scan*.

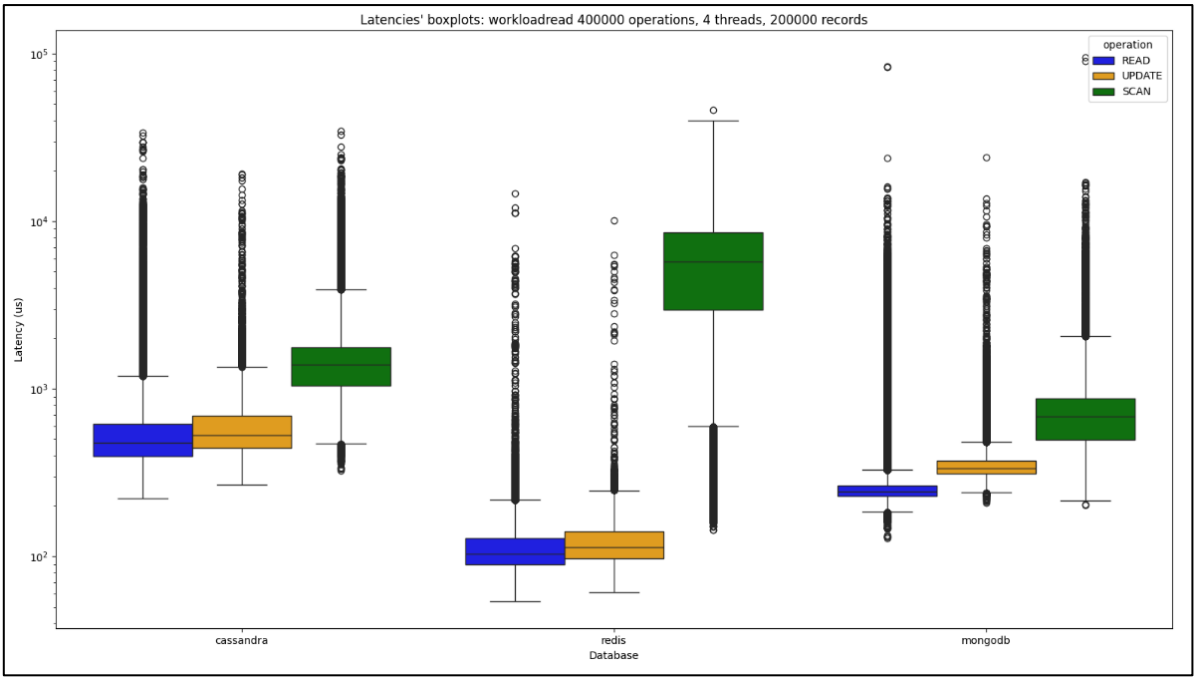

Figura 5: Boxplot Latenza Workload Read

Quindi, anche se il 65% delle operazioni di lettura viene completato in maniera molto rapida da Redis, comunque, la latenza media risente in maniera importante delle operazioni di *scan*, che hanno una latenza mediamente di uno o due ordini di grandezza superiore rispetto alle altre operazioni. Al contrario, osservando gli altri database con una strutturazione del dato maggiore, si deduce che le operazioni di *scan* sono leggermente più lente delle semplici *read*, probabilmente a causa di ottimizzazioni interne e della definizione di indici, specialmente nel caso di MongoDB. Invece, queste ottimizzazioni non sono presenti in Redis, probabilmente a causa della struttura di tipo key-value.

Un altro fenomeno interessante, osservato dalle mappe di calore precedenti, è la tendenza di MongoDB a "saturare" velocemente verso il valore massimo raggiungibile di throughput. Per investigare meglio questa tendenza, si sono eseguiti dei test ulteriori con un numero di client progressivamente superiore, in modo da studiare anche l'andamento del throughput degli altri due database. Si sono eseguiti i test anche con 10, 12, 14, 16, 18 e 20 thread, non solo per quanto riguarda il throughput, ma anche per le latenze, per capire anche quanto velocemente deteriorano le prestazioni dei singoli database all'aumentare dei client. Tali risultati sono visibili nella *Figura 6* e nella *Figura 7*.

Per quanto riguarda il throughput, si conferma la tendenza menzionata prima di MongoDB, che tende a raggiungere, anche con pochi client, il valore massimo, mentre successivamente si assesta su un valore costante all'aumentare dei client. Al contrario, Redis raggiunge il throughput massimo in maniera più lenta e graduale, arrivando a saturazione solo con un numero di client superiore a 12.

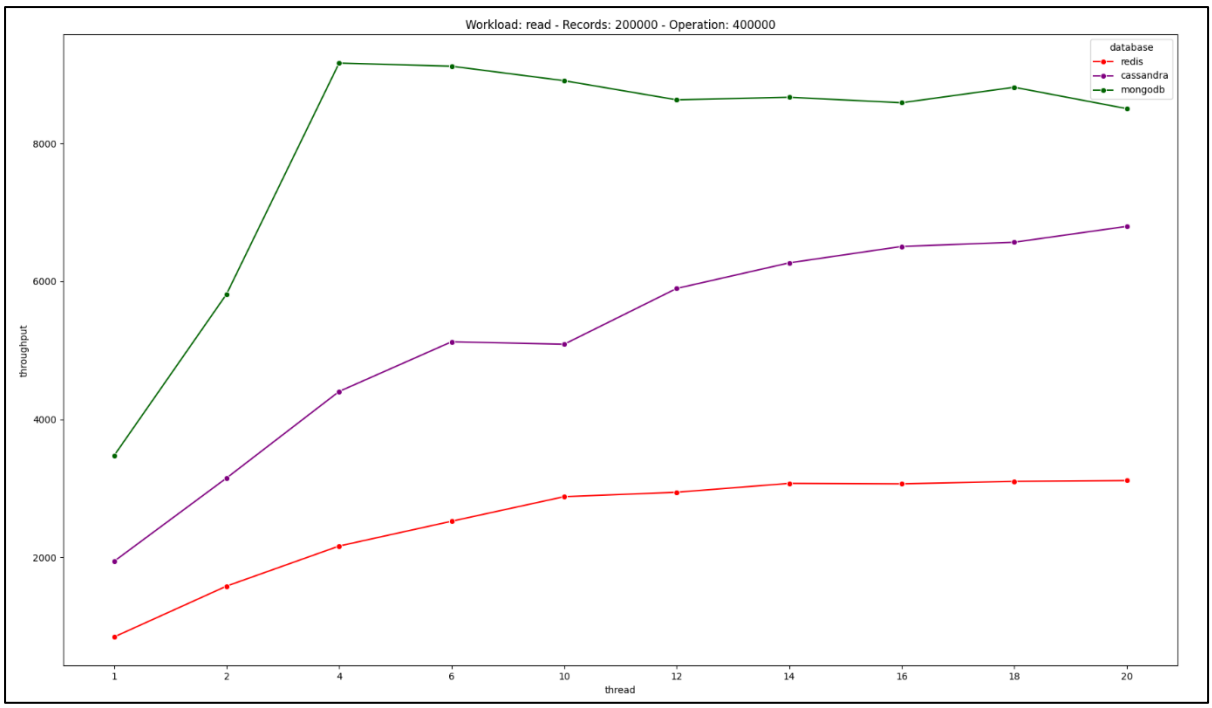

Figura 6: Throughput all'aumentare del numero di client

Più interessante è il caso di Cassandra, in quanto, nonostante l'aumentare dei client, si osserva una tendenza sempre crescente del throughput senza rilevare un punto di saturazione, nel caso di operazioni di lettura. Quindi, dai test condotti, si potrebbe pensare che Cassandra, tra i tre database, scali meglio all'aumentare del numero di client che fanno delle richieste, il che è ragionevole, dato che è pensato per architetture cluster.

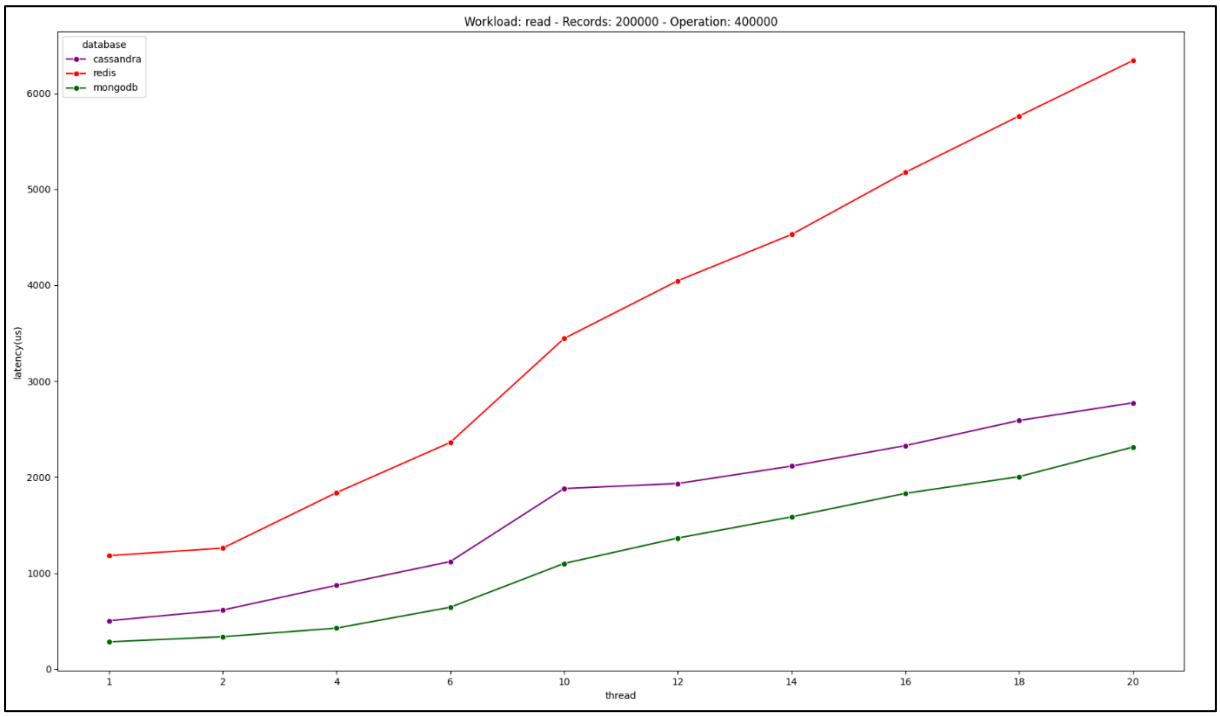

Figura 7: Latenza all'aumentare del numero di client

Osservando la *Figura 7*, si rileva che, all'aumentare del numero di client Redis ha una latenza che aumenta rapidamente superando anche i 5 millisecondi per un numero di client considerevole. Tale fenomeno è imputabile probabilmente alla lentezza delle *scan*, in quanto facendo eseguire delle *scan* da più client, il database deve servire più operazioni nella stessa unità di tempo, generando attese più lunghe. Al contrario, MongoDB e Cassandra hanno una latenza media che cresce lentamente all'aumentare dei thread, riuscendo a scalare bene con tanti client nel caso in cui queste richiedano per lo più letture puntuali e scansioni.

3.2 Workload Update

Questo workload, come descritto in precedenza, presenta una grande quantità di operazioni di modifica come *update* e *read-modify-write*, intervallate da un numero limitato di operazioni di *read.*

Come nel caso precedente si è deciso di studiare, attraverso delle heatmap, quali sono i throughput che i tre database riescono ad esprimere nei diversi scenari di test predisposti.

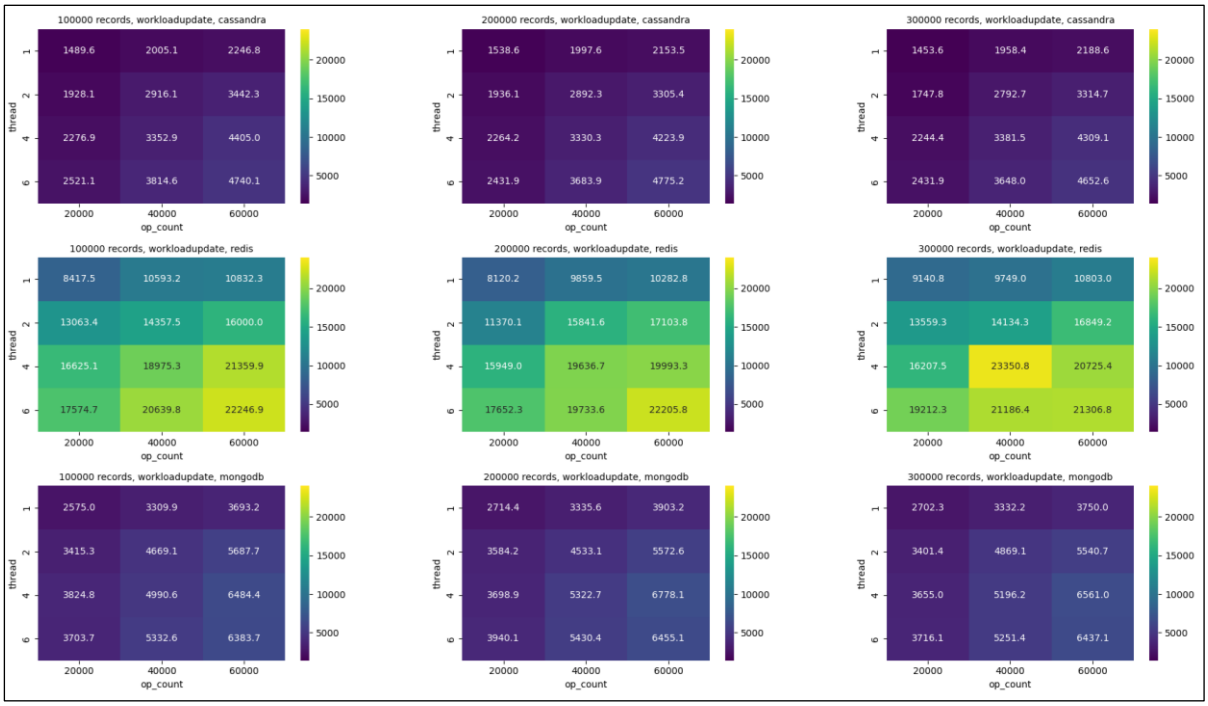

Figura 8: Heatmap Throughput Workload Update

Come si può notare dalla **Figura 9: Heatmap Latenza Workload Update***Figura 9*, la prima cosa evidente è la schiacciante superiorità, al livello di prestazioni, del database Redis rispetto agli altri due. Sicuramente, grazie alla sua capacità di lavorare in-memory, esso riesce ad effettuare le operazioni di update molto velocemente, in quanto può gestire più operazioni di modifica in memoria centrale, salvandole solo in un secondo momento in memoria secondaria. Entrando più nel dettaglio, si può anche qui riscontrare la stessa tendenza vista

per il workload precedente: all'aumentare del numero di client che fanno richieste contemporaneamente ai database, si assiste all'aumentare del throughput. Stessa tendenza si evidenzia all'aumentare del *op_count*. Quest'ultimo fenomeno potrebbe essere dovuto al fatto che, più sono le operazioni destinate ai database, più vengono attivati e sono efficaci meccanismi di batching, che raggruppano operazioni simili effettuandole tutte insieme nel database. Dalla figura si possono riscontrare anche alcuni test che sono in contrasto con le tendenze appena descritte, come quello di Redis a 300000 record e 40000 operazioni. Probabilmente questi casi sono outlier dovuti a una inevitabile varianza nei test.

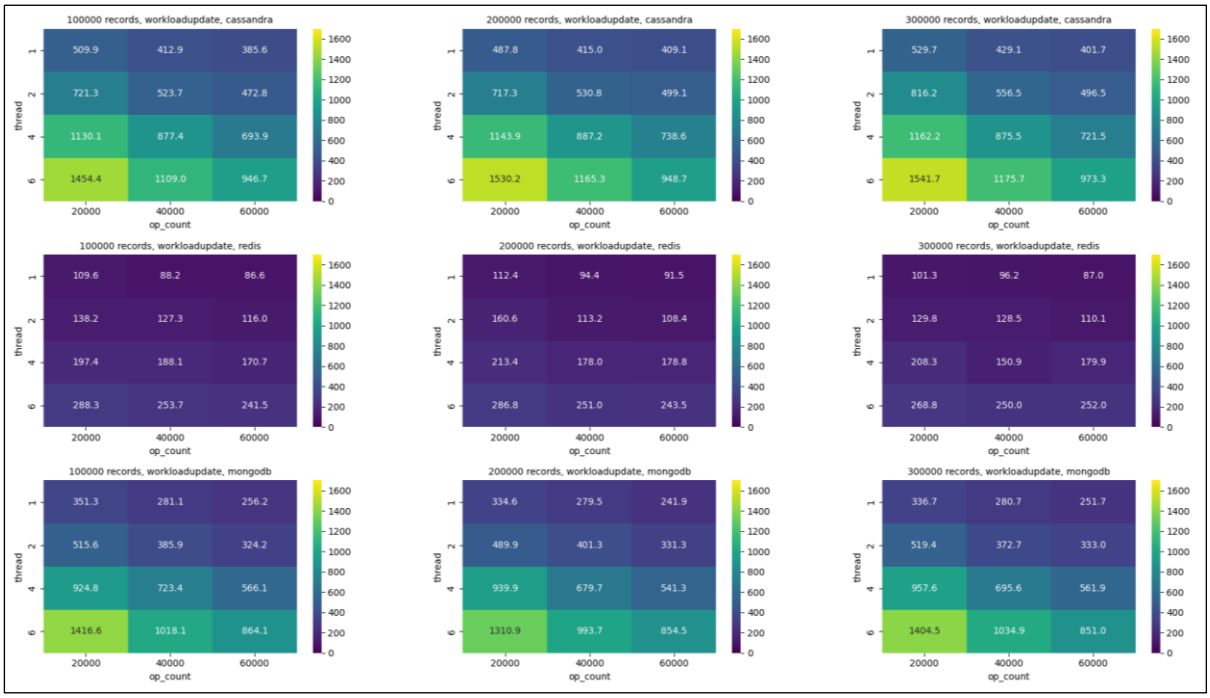

Figura 9: Heatmap Latenza Workload Update

Osservando, invece, la heatmap della latenza (*Figura 10*), si può riscontrare un aumento della latenza media all'aumentare del numero di client. Infatti, come nel workload precedente, all'aumentare del numero di thread, le singole operazioni richiedono più tempo per essere eseguite. Tuttavia, lavorando in parallelo, si riescono ad effettuare complessivamente più operazioni, determinando un aumento del throughput. Inoltre, i test in esame sono stati effettuati con una distribuzione uniforme delle operazioni di modifica su tutti i dati, comportando una minore sovrapposizione delle operazioni. Di conseguenza, gli effetti di questo fenomeno risultano essere ancora più evidenti. In generale, osservando i dati contenuti in questi grafici, si può osservare come Redis sia il migliore per le operazioni di update, al secondo posto si posiziona MongoDB e infine Cassandra.

Successivamente, si sono effettuate analisi più dettagliate delle prestazioni di questi database, al fine di investigare ulteriormente i fenomeni emersi dall'analisi aggregata. Come in tutti i workload, per queste analisi si sono fissate le tre

variabili dei test, ovvero numero di operazioni, numero di record e numero di thread.

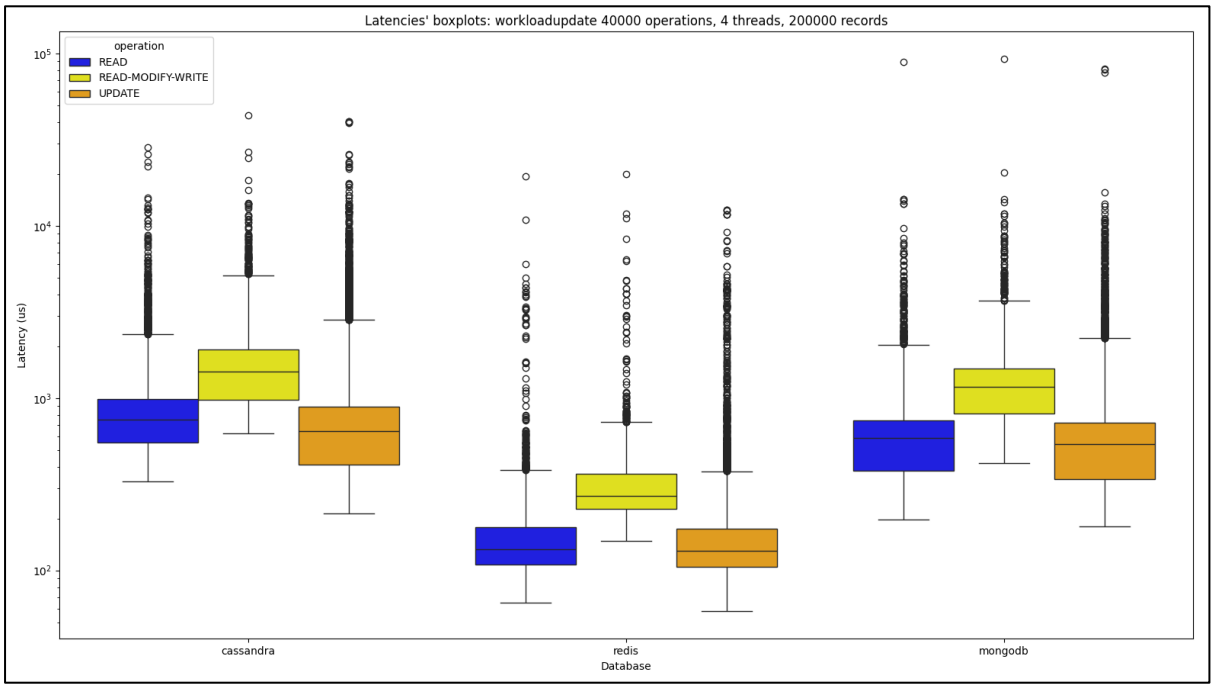

Figura 10: Boxplot Latenza Workload Update

Per prima cosa, si sono analizzati i boxplot dei tre database (*Figura 10*), tracciati sulla base dei valori delle latenze delle singole operazioni del test. Anche da questo grafico si evidenzia lo stesso comportamento che era possibile vedere dalle heatmap. Redis si conferma il database più performante, perché non ci sono in questo caso operazioni che comportano un rallentamento, come avveniva per la *scan* nel workload read. Inoltre, si può notare come Redis sia anche il database con meno outlier, cioè quello che garantisce prestazioni più costanti.

Successivamente si è deciso di analizzare come i database riescono a scalare all'aumentare del numero di client che fanno loro richieste. Già dalle heatmap, come negli altri workload, si era reso evidente come alcuni database tendessero a saturare il loro throughput già con pochi client. A confermare questo trend anche per questo workload è la *Figura 11*. In essa è molto evidente che da un certo numero di client in poi i database tendano a non migliorare ulteriormente le loro prestazioni. Il primo che sembra arrivare a saturazione è MongoDB che raggiunge il suo picco molto presto, già a quattro o sei thread. Cassandra e Redis sembrano invece saturare più o meno allo stesso momento, intorno ai 12 thread.

Infine, all'aumentare del numero di client, osservando la *Figura 12*, si può osservare un aumento di latenza simile e vertiginoso per MongoDB e Cassandra. Essi sembrano procedere con un aumento lineare considerevole, con Cassandra che sembra performare meglio con un numero alto di client. Redis, invece, sembra essere il database nettamente più performante per le operazioni di update.

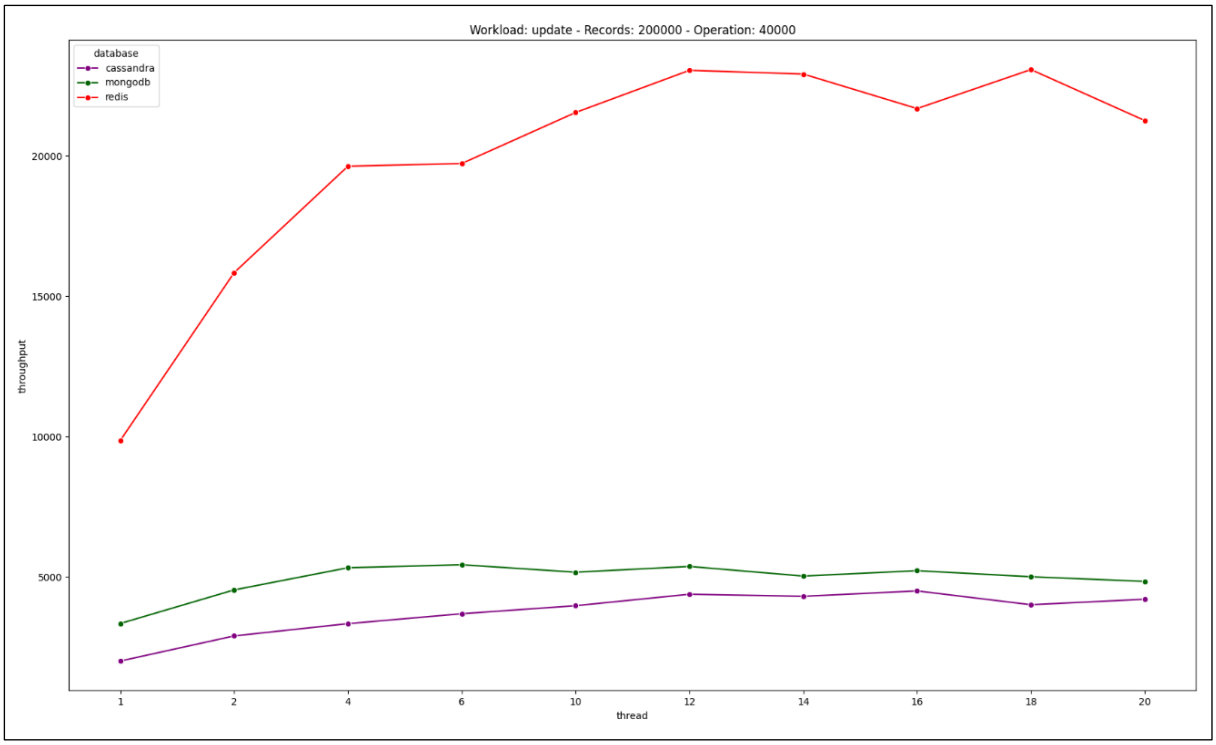

Figura 11: Throughput all'aumentare del numero di client

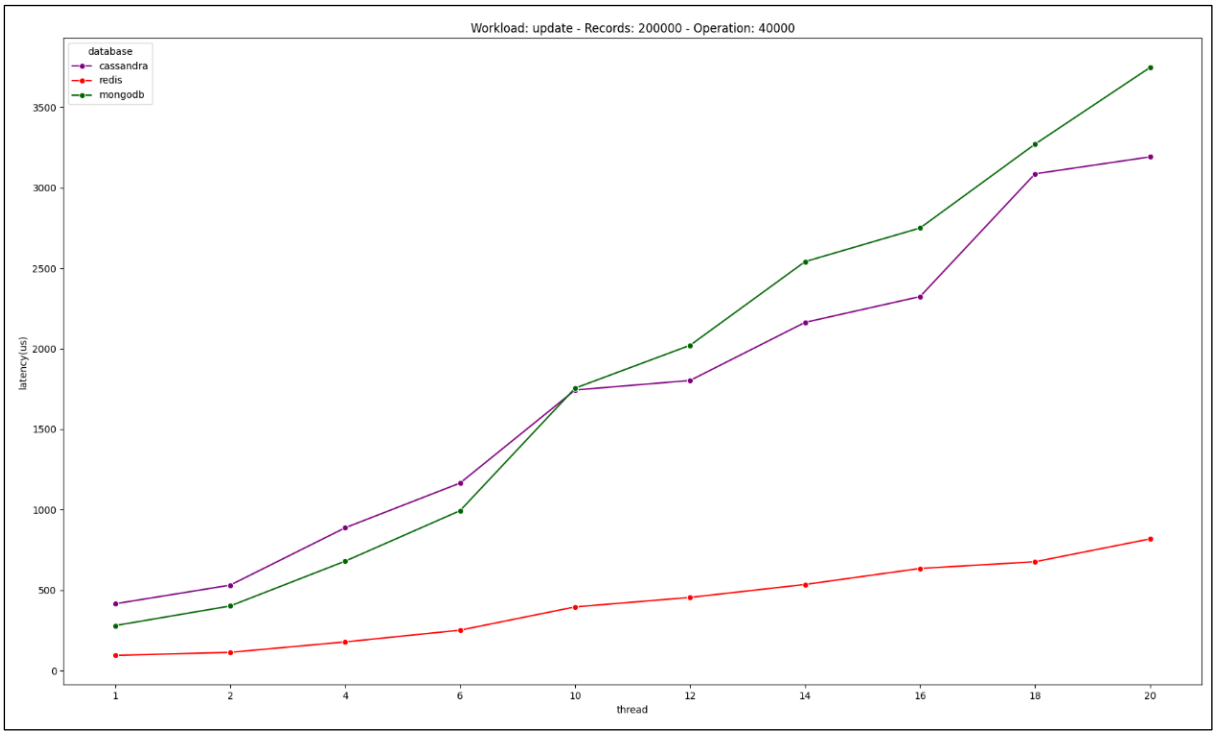

Figura 12: Latenza all'aumentare del numero di client

3.3 Workload Insert

Il terzo carico di lavoro testato si distingue dagli altri poiché ha la peculiarità di essere caratterizzato da molte operazioni di inserimento, oltre che da letture ed aggiornamenti sporadici. Ciò permette di simulare l'inserimento di nuovi dati nel database, mentre altri dati vengono visionati o modificati. In questo modo, è possibile verificare il comportamento dei tre sistemi in uno scenario totalmente differente da quelli precedenti. Innanzitutto, è stato analizzato il throughput medio per ogni configurazione testata, come mostrato in *Figura 13*.

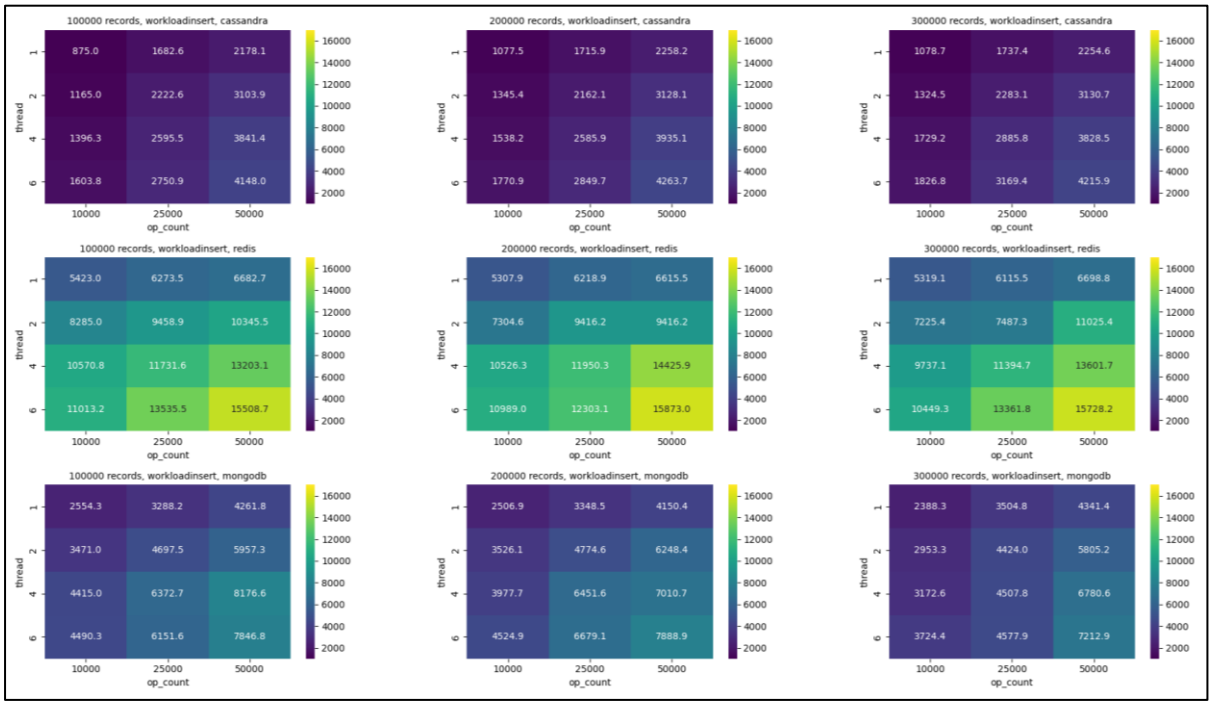

Figura 13: Heatmap Throughput Workload Insert

Come si può ben vedere dalla heatmap, si riscontra la stessa tendenza che si aveva negli altri workload. Infatti, per tutti i database, il throughput non dipende dal numero dei record, ma cresce all'aumentare del numero delle operazioni totali e del numero dei thread. Come già detto in precedenza, tale comportamento è dovuto al fatto che più richieste vengono elaborate contemporaneamente, sfruttando al massimo le risorse del sistema. Inoltre, con un maggior numero di client, si sfruttano meccanismi di caching o di *batch inserting* che portano ad un aumento del throughput globale. Tra i tre database confrontati, emerge una netta superiorità di Redis in termini di prestazioni, soprattutto al crescere del numero di client e del numero totale di operazioni eseguite. Questo vantaggio può essere attribuito a due fattori principali: il primo è che Redis è un key-value store, che non presenta una strutturazione definita del dato, il secondo è che lavora in memoria, ottimizzando le operazioni e differendo il trasferimento dei dati su disco. Al contrario, le performance di MongoDB e di Cassandra sono inferiori, con quest'ultimo che risulta essere il peggiore dei tre.

Similmente a quanto fatto per gli altri carichi di lavoro, anche in un workload *insert-mostly* è stato considerato il tempo necessario per eseguire una query sul database e ottenere il risultato desiderato; infatti, la latenza potrebbe essere cruciale nella scelta del database.

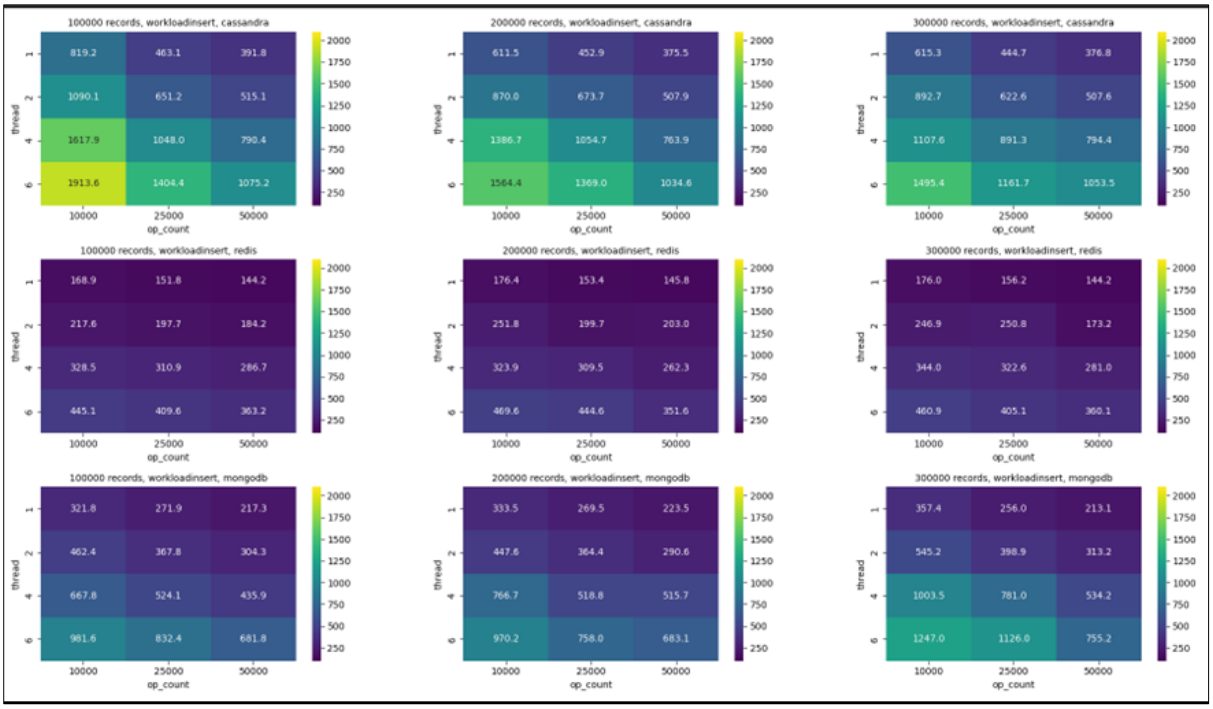

Figura 14: Heatmap Latenza Workload Insert

Come si evince dalla *Figura 14*, le tendenze della latenza media non sono chiaramente evidenti quando si varia la taglia del database e il numero di operazioni. In accordo con quanto notato nei casi precedenti, anche per questo workload, in tutti e tre i database, si nota un incremento della latenza media in relazione al numero di client che richiedono accesso al database. Tale aspetto è giustificato dal fatto che, nonostante un aumento del throughput a causa dell'esecuzione di un maggior numero di operazioni nello stesso intervallo temporale, i singoli processi subiscono un ritardo nell'esecuzione, determinando un conseguente aumento della latenza. Confrontando la *latency* dei tre database, Redis risulta essere il migliore, mentre la latenza media di Cassandra è molto alta; quella di MongoDB, invece, si stabilizza a metà tra gli altri due casi.

A tal punto, è stata eseguita l'analisi della latenza delle operazioni, fissando il numero delle operazioni, la taglia del database e il numero dei thread. In particolare, si pone l'attenzione sulle *insert*, che sono preponderanti nel workload considerato. Dal boxplot in *Figura 15*, è interessante osservare la distribuzione dei valori della latenza e il valore mediano, in linea con quello medio espresso dalla heatmap precedente. Perciò, è possibile confermare che Redis è il database caratterizzato dalla latenza più bassa, che MongoDB ha una latenza un po' più alta e che Cassandra, da questo punto di vista, è il peggiore.

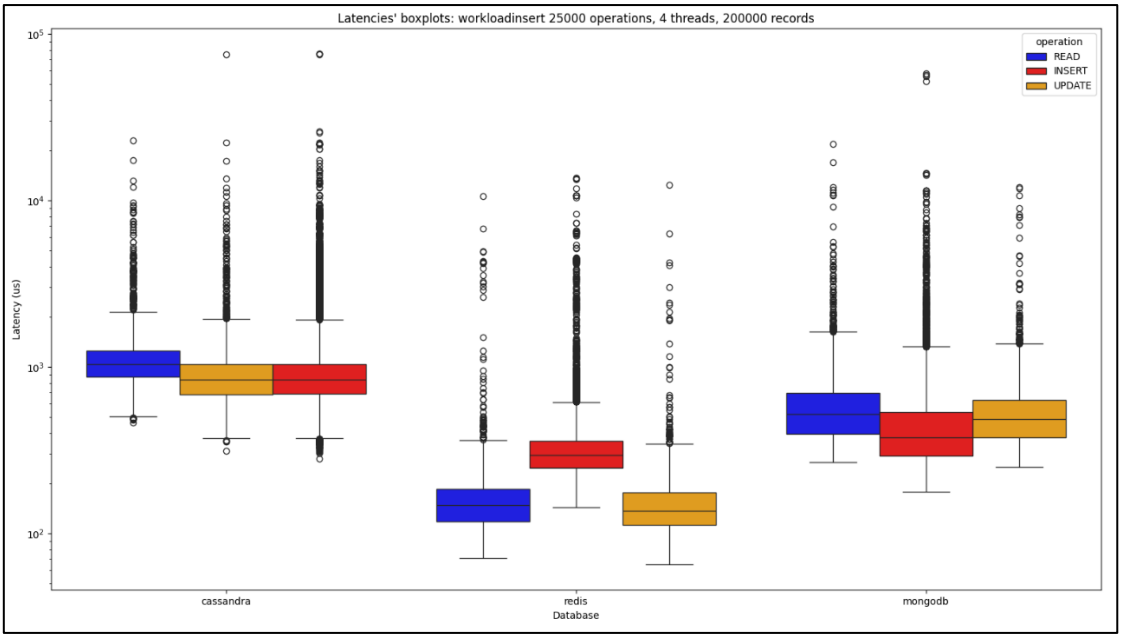

Figura 15: Boxplot Latenza Workload Insert

Riguardo al restante 20% delle operazioni (equamente divise tra letture ed aggiornamenti), la latenza mediana è simile a quella riferita alle *insert* solo nel caso di MongoDB e di Cassandra; invece, per queste operazioni quella registrata con Redis è nettamente inferiore, in linea con quanto analizzato precedentemente.

Nelle analisi precedenti, si poteva notare il fenomeno della saturazione da parte dei database già dalle heatmap, come avveniva nel workload read. Nel caso del workload *insert-mostly*, invece, non è stato possibile rilevare un comportamento simile, per cui si è voluto procedere con uno studio più dettagliato. Dunque, per capire sia questo fenomeno sia l'andamento generale del throughput, sono stati eseguiti altri test con un numero di client variabile. In maniera simile a quanto fatto in precedenza, sono stati considerati anche le casistiche con 10, 12, 14, 16, 18, 20 thread, ed è stato posto il focus sia sul numero di operazioni al secondo sia sulla latenza media, così da visualizzare come peggiorano le prestazioni dei database.

Riguardo al throughput (*Figura 16*), ancora una volta MongoDB satura prima, cioè tende a raggiungere il valore massimo in corrispondenza di 6 thread, per poi assestarsi su valori più bassi. Invece, Cassandra ha un valore di throughput più basso degli altri, ma ha un andamento crescente più lineare all'aumentare dei client, per cui non si può affermare di aver raggiunto il punto di saturazione. Invece, Redis ha un comportamento più peculiare, in quanto il throughput aumenta vertiginosamente all'inizio, si stabilizza con 10 client e torna ad aumentare, fino a stabilizzarsi, suggerendo saturazione.

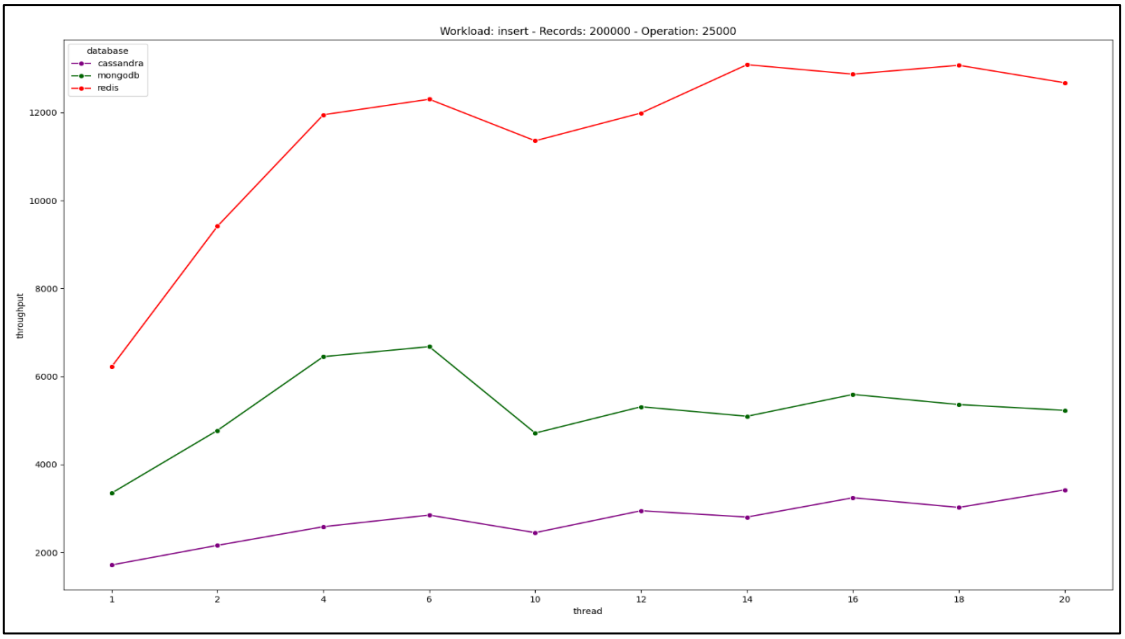

Figura 16: Throughput all'aumentare del numero di client

Per quanto riguarda la latenza, dalla *Figura 17* si può confermare quanto detto in precedenza; si può osservare come Redis scali nettamente meglio, avendo una crescita della *latency* molto più bassa rispetto agli altri due database. Il comportamento di MongoDB e di Cassandra è più particolare, poiché sembra che la differenza di latenza si assottigli nel tempo, suggerendo che, con più client, forse si potrebbe arrivare a valori di latenza uguali, o addirittura superiori per MongoDB. Tuttavia, dagli esperimenti eseguiti, Cassandra offre prestazioni peggiori sotto questo punto di vista, probabilmente perché le ottimizzazioni per esso definite sono evidenti con molti più client.

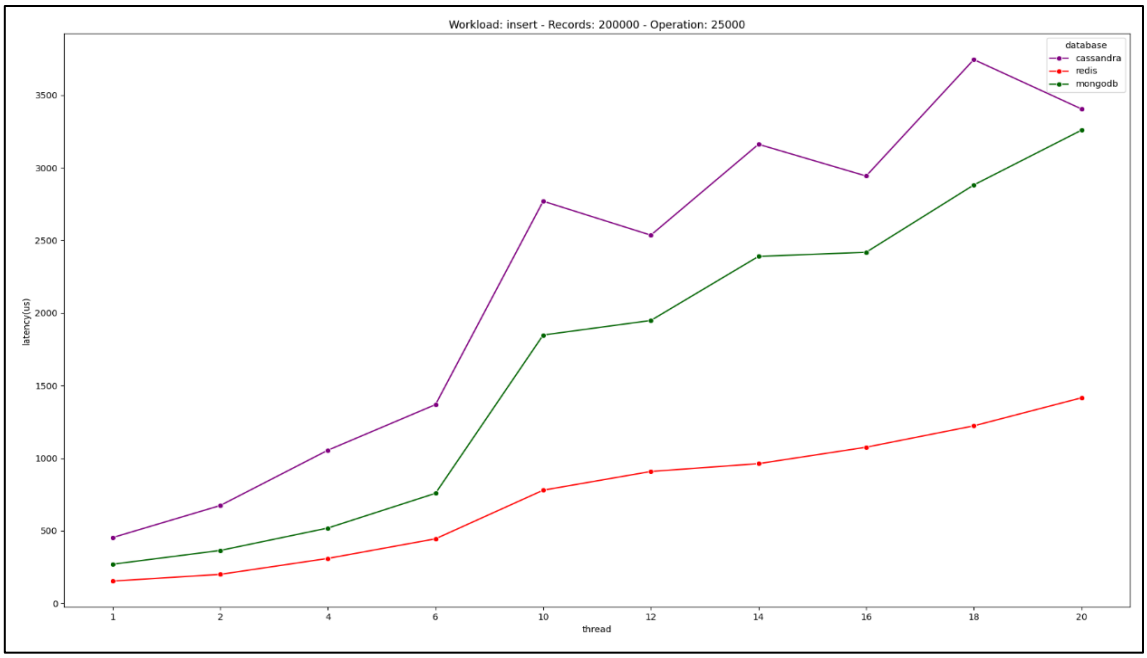

Figura 17: Latenza all'aumentare del numero di client

4. Conclusioni

In questa analisi, sono state confrontate le prestazioni di tre popolari sistemi NoSQL, ovvero Redis, MongoDB e Cassandra, mediante l'utilizzo del framework di benchmarking YCSB (Yahoo! Cloud Serving Benchmark). L'obiettivo principale era valutare le prestazioni di questi DBMS in diversi carichi di lavoro e in diverse configurazioni, al fine di determinare i loro punti di forza e le relative debolezze.

Un fenomeno che vale per tutti i workload, a prescindere dalla taglia del database o del numero delle operazioni, riguarda il numero dei thread. Infatti, all'aumentare dei client aumenta anche il throughput poiché, essendoci più richieste, vengono eseguite più operazioni, nello stesso lasso di tempo. D'altro canto, per lo stesso motivo, aumenta la latenza. Considerando il numero delle operazioni totali, si può notare che la latenza diminuisce all'aumentare dell'*opcount*, probabilmente a causa di meccanismi di caching o di batch inserting/updating. Questa tendenza non si riscontra solo nel workload read, che differisce dagli altri workload per il numero di operazioni effettuate, di un ordine di grandezza più grande. Inoltre, un'altra differenza dal workload update è data anche dal tipo di distribuzione scelta per selezionare i dati su cui effettuare le operazioni, in quanto nel caso dell'update si usa la distribuzione uniforme e nel caso del read si usa la distribuzione zipfian.

Perciò, alla luce di tutti questi elementi, si può ipotizzare che la diminuzione della latenza e l'aumento del throughput, osservati nel workload update e nel workload insert, siano dovuti a meccanismi di ottimizzazione interni o ai database o al framework stesso. Tale ipotesi sembra avvalorata dal fatto che, a parità di database e setting sperimentale, la latenza media di un test nel workload read è inferiore a quella degli altri due. Viceversa, si osserva che il throughput è maggiore per il workload read e minore per gli altri due.

Infatti, è ragionevole supporre che, dato l'alto numero di operazioni del workload read e dato il tipo di distribuzione, i record che hanno una probabilità maggiore di essere scelti entrano quasi tutti in cache a un certo punto, accelerando le operazioni. Al contrario nel workload update, in cui sono eseguite meno operazioni e la distribuzione è di tipo uniforme, è probabile che siano più facilmente richiesti record fuori dalla cache, provocando un rallentamento delle operazioni. Infine, nel caso del workload insert, potrebbero intervenire meccanismi di batching o raggruppamento delle operazioni che sono più efficienti all'aumentare del numero di operazioni complessive da svolgere.

I risultati di questa analisi hanno fornito una panoramica completa sulle peculiarità, sui vantaggi e sui difetti caratteristici dei tre DBMS. Dunque, si

possono fare alcune considerazioni che siano anche in grado di guidare nella scelta dell'uno o dell'altro sistema.

Redis ha dimostrato di essere la migliore soluzione per i carichi di lavoro in cui sono eseguite operazioni puntuali di lettura, di aggiornamento e di inserimento. Invece, in presenza di scansioni, questo database ha mostrato i propri limiti, il che non lo rende preferibile per i task di *analytics*, poiché quest'ultimi spesso richiedono l'aggregazione di dati in seguito a query su intervalli. In aggiunta, essendo un database che lavora principalmente in memoria centrale, è ottimo per tutte quelle applicazioni in cui si richiede un accesso veloce ai dati. A conferma di quanto detto, il tempo di esecuzione offerto da Redis nel workload update e in quello insert è nettamente più basso.

MongoDB ha offerto una buona flessibilità ed è risultato il più equilibrato, essendo il database più congruo ai test eseguiti. Esso ha avuto le prestazioni migliori nel workload read, data la maggiore capacità di gestire le operazioni di scansione, e delle prestazioni accettabili negli altri due. Dunque, MongoDB rappresenta il tradeoff migliore tra i vari workload, rendendolo così la soluzione più opportuna per un carico misto. Al contempo, dai test eseguiti, si è notato che è il database che satura prima il throughput all'aumentare del numero di client. Inoltre, MongoDB è un sistema molto utilizzato, user-friendly e ormai consolidato, risultando più intuitivo e semplice da utilizzare.

Cassandra non si è distinto in nessun workload e le sue prestazioni sono state generalmente inferiori a quelle degli altri sistemi. Infatti, poiché questo database è pensato per un utilizzo distribuito su cluster, non si è riusciti a valutare nel migliore dei modi le sue performance. Tuttavia, all'aumentare dei thread, in alcuni casi sembra scalare meglio, confermando che, probabilmente, in architetture distribuite è una buona soluzione.

Per queste motivazioni, si può affermare che MongoDB sia il database più adeguato nel caso d'uso scelto. In futuro, però, ulteriori approfondimenti potrebbero riguardare le prestazioni di MongoDB in coordinazione con Redis nello stesso scenario, utilizzando quest'ultimo come cache di MongoDB, per vedere se c'è un miglioramento delle prestazioni. Allo stesso modo potrebbe essere interessante confrontare su un'architettura distribuita le prestazioni di MongoDB, nella versione cloud, e di Cassandra così da poter effettuare test più adeguati alla struttura e alle caratteristiche di quest'ultimo.