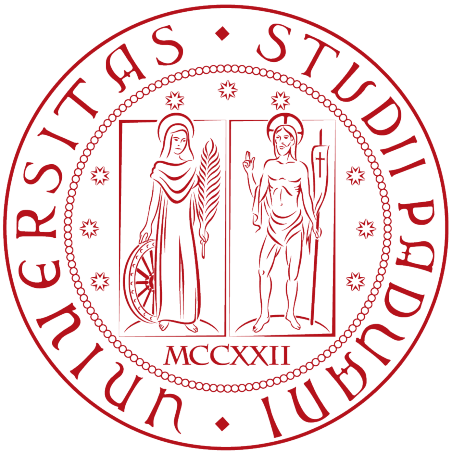
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Realizzazione e analisi prestazionale di un database NoSQL per la possibile migrazione da un database relazionale

Tesi di laurea

Relatore

Prof. Luigi De Giovanni

Laureando

Nicholas Sertori

Anno Accademico 2021-2022

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage svolto presso l'azienda Ifin Sistemi s.r.l. Lo stage è stato svolto alla conclusione del percorso di studi della laurea triennale in Informatica, occupando circa trecentoventi ore divise in otto settimane. Lo scopo del progetto svolto è stato di effettuare uno studio di fattibilità per l'integrazione di una soluzione di database NoSQL nei prodotti dell'azienda. Lo studio di fattibilità ha comportato una fase di analisi delle varie soluzioni NoSQL esistenti sul mercato, una fase di analisi delle soluzioni attualmente adottate all'interno dei prodotti Ifin, ed infine una fase di valutazione pratica delle soluzioni individuate, con relativi benchmark per il confronto delle prestazioni ed un approfondimento sulle differenze di progettazione tra database relazionali classici e database NoSQL.

"So long, and thanks for all the fish"

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Luigi De Giovanni, relatore della mia tesi, per l'aiuto fornitomi durante la stesura del lavoro.

Desidero ringraziare con affetto i miei genitori, mio fratello, i nonni e gli zii per il sostegno fornitomi in questi n+1 anni passati da quando ho iniziato questo percorso di laurea triennale, e per tutto quello che avete fatto da sempre per spronarmi a dare il massimo, senza costrizioni di sorta, senza mai giudicare le scelte che ho fatto, rimanendo sempre a disposizione per fornire il vostro supporto incondizionato.

Ringrazio amici ed amori che in questi anni mi sono sempre stati vicini come una grande famiglia, condividendo i miei alti ed i miei bassi. Siete troppi per nominarvi tutti, ma se state leggendo, sapete che sto parlando di voi. Grazie di cuore. Non sarei chi sono se non avessi incontrato ognuno di voi, vi devo molto e vi voglio bene.

Un ultimo doveroso ringraziamento va ad Ifin, che mi ha ospitato durante i due mesi di tirocinio, ma soprattutto alle persone che ho incontrato durante la mia permanenza. Grazie di aver condiviso le vostre conoscenze e la vostra pazienza, permettendomi di imparare qualcosa. Alla prossima.

Padova, Dicembre 2022 Nicholas Sertori

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 L'azienda

L'azienda proponente è Ifin Sistemi s.r.l., un'azienda di prodotto che si occupa principalmente di informatica finanziaria. Il suo *core business* è incentrato su piattaforme che facilitano l'archiviazione di pratiche e documenti legali in modo sicuro e affidabile, e mediano l'invio di fatture elettroniche tra aziende e Sistema di Interscambio, un sistema informatico gestito dall'Agenzia delle Entrate.

Figura 1.1: Logo di Ifin Sistemi s.r.l.

1.2 Situazione attuale

I due prodotti di punta dell'azienda, che compongono il *core business* sopra accennato, sono LegalArchive e InvoiceChannel. All'interno di Ifin coesistono vari team che mantengono la codebase di questi software, occupandosi della loro manutenzione e della modellazione di nuove funzionalità richieste dai clienti.
A livello pratico, il codice per i software di Ifin è scritto in *Java*, appoggiato quando serve al framework *Spring*.
All'interno di quello che è lo stack tecnologico aziendale, data la scelta del linguaggio di programmazione, troviamo quello che può essere considerato uno standard per lo sviluppo di applicativi che fanno largo uso di database. Tecnologie come JPA, JDBC, Tomcat, Hibernate ed EclipseLink risultano essere mattoni fondamentali alla base dei

software di Ifin.
Infine, per quanto riguarda la scelta dei database veri e propri, anche in questo caso l'azienda fa riferimento a quelli che sono gli standard dell'industria. Si parla quindi di database relazionali, e più nello specifico di *Oracle Database* e *Microsoft SQL Server*.

1.3 Esigenze da cui nasce l'idea del progetto

Sebbene attualmente, all'interno dell'azienda, siano implementate varie soluzioni intelligenti per garantire il funzionamento delle piattaforme anche in situazioni di stress dei sistemi (come per esempio il partizionamento delle tabelle più grandi), i database relazionali possono soffrire di problemi di scalabilità quando la mole di dati che devono gestire raggiunge determinate dimensioni.
L'utilizzo di una soluzione NoSQL è pertanto un'allettante alternativa, proprio perchè spesso scalabilità e affidabilità sono caratteristiche centrali di queste tecnologie. Occorre tuttavia effettuare uno studio più completo per determinare se l'utilizzo di questo tipo di database si presta realmente alle necessità e alle complessità dei sistemi di Ifin, per poter giustificare un investimento non indifferente di risorse nella conversione e migrazione che ne conseguirebbe. Il progetto di stage si inserisce in questo contesto, unendo le necessità dell'azienda alla possibilità di effettuare una ricerca consocitiva dei database NoSQL.

1.4 Organizzazione del testo

Il secondo capitolo descrive il progetto di stage e presenta un'iniziale pianificazione delle attività.

Il terzo capitolo descrive il contesto del progetto, presentando nel dettaglio le tecnologie studiate e quelle già implementate dall'azienda.

Il quarto capitolo approfondisce la fase di preparazione alla sperimentazione, con il settaggio degli strumenti necessari e l'organizzazione dell'ambiente di test.

Il quinto capitolo descrive la fase di sperimentazione e raccolta dei dati utili al confronto che sta al centro del progetto.

Il sesto capitolo contiene le conclusioni elaborate al termine del tirocinio.

Capitolo 2

Descrizione dello Stage

2.1 Introduzione al progetto

L'obiettivo dello stage è quello di effettuare uno studio preliminare delle tecnologie che gravitano attorno al concetto di NoSQL, per evidenziarne vantaggi e svantaggi, in modo da poter valutare in maniera concreta eventuali possibilità di integrazione nello stack aziendale.
Una volta portato a termine questo studio, si vuole mettere a confronto le soluzioni attualmente adottate con quelle analizzate, per valutare in modo concreto in che modo queste ultime possono portare ad un miglioramento nella gestione e nell'utilizzo degli applicativi aziendali.

2.2 Obiettivi formativi

Gli obiettivi formativi dell'attività di stage sono i seguenti:

* Approfondire conoscenze in ambito NoSQL;
* Apprendere come effettuare attività di test di carico per la valutazione prestazionale di un Database;
* Apprendere metodologie ed approcci propri dell'ambiente lavorativo, diversi da quelli universitari.

2.3 Attività preventivate

La durata complessiva dello stage è stata di 8 settimane di lavoro a tempo pieno per un totale di circa trecentoventi ore.
Secondo il piano di lavoro iniziale definito con l'azienda, le attività sono distribuite come riportato in Tabella 2.1.

Durata in ore	Settimana	Descrizione

40	1	∗ Formazione sullo stack operativo e di sviluppo aziendale; ∗ Formazione stack *Java EE*
80	2, 3	∗ Studio NoSQL in generale; ∗ Verifica soluzioni NoSQL specifiche; ∗ Identificazione di soluzioni NoSQL enterprise da analizzare e KPI aziendali.
80	4, 5	∗ Analisi di dettaglio delle soluzioni precedentemente identificate.
80	6, 7	∗ Creazione e codifica di test per lo stuio delle performance di carico e aderenza ai KPI individuati.
40	8	∗ Revisione test e stesura documentazione finale.
Totale ore:		**320**

Tabella 2.1: Pianificazione delle attività

Durante il periodo di stage in azienda il piano di lavoro ha subito modifiche e di conseguenza le attività svolte divergono leggermente dalla pianificazione qui presentata. Tali modifiche sono state effettuate in risposta al naturale evolversi del progetto di fronte ad imprevisti ed esigenze nate durante il percorso.
Durante tutta la durata del tirocinio si è lavorato a contatto con il relatore preposto all'interno dell'azienda e con varie altre figure di riferimento più esperte nei vari ambiti toccati dal progetto.

Capitolo 3

Contesto

Questo capitolo è dedicato all'introduzione delle nozioni emerse dalla fase di apprendimento svolta durante il tirocinio, riguardanti nello specifico le varie di tipologie di database NoSQL esistenti.

3.1 Cosa significa NoSQL

Quando si parla di database NoSQL si intendono tutte quelle tecnologie di persistenza di dati che non prevedono strettamente l'utilizzo del paradigma SQL. L'acronimo significa infatti Not-only Structured Query language.
Tuttavia, mentre le tecnologie classiche che rientrano sotto l'ombrello dei RDBMS (*relational database management systems*) sono in qualche modo uniformate dalla "lingua franca" che condividono (SQL), le implementazioni che ricadono nel paradigma NoSQL sono estremamente varie nell'effettivo modo in cui gestiscono i dati, e di conseguenza nei linguaggi che adottano.
Questo può rendere il processo di selezione più complesso, ma fornisce anche un'ampia gamma di opzioni che, se scelte con cognizione di causa, possono portare a soluzioni estremamente specializzate ed efficaci.

3.2 Idee ed implementazioni

Vengono di seguito elencati i vari sottogruppi che possiamo individuare all'interno del panorama NoSQL.

3.2.1 Key-Value Store

Questa categoria di database rappresenta in realtà un modo per immagazzinare informazioni, basato sulla strutturazione dei dati in coppie "chiave-valore".
Tutte le operazioni effettuate su un DB di questo tipo si basano quindi sulla chiave per recuperare il suo valore associato. Nella sua forma più semplcie, un *key-value* store funziona esattamente come un dizionario.

Come esempio abbiamo il caso di Redis.
Nasce inizialmente come database appartenente al paradigma *key-value*, composto quindi da un set di chiavi a cui sono legati dei valori, contenuto per intero nella memoria RAM.
L'idea era di avere una sorta di cache in cui salvare dei dati frequentemente utilizzati per poterli recuperare molto velocemente.
Se inizialmente veniva utilizzato a supporto di altri database, nel tempo Redis è stato ampliato fino a diventare una soluzione unica (*Redis Stack*), in grado grazie ai suoi moduli di effettuare ricerche *fulltext*, visualizzare le informazioni in grafi ed implementare il salvataggio di documenti in formato `JSON` su memoria persistente in un database distribuito.
Il potere di questa soluzione sta nell'avere tutte queste funzionalità all'interno dello stesso prodotto, garantendo una semplificazione non indifferente del processo di integrazione.

Figura 3.1: Logo di Redis

3.2.2 Wide Column Store

Questo tipo di database NoSQL estende il concetto di *key-value*, dove le informazioni sono raccolte in colonne, le colonne in righe, le righe in tabelle e quest'ultime sono raccolte sotto un cosiddetto *keyspace*. A differenza dei database relazionali classici, questo tipo di soluzione non necessita di uno schema, la struttura che definisce il contenuto di una tabella nei RDBMS. Questo permette ai *Wide Column database* di accettare dati non strutturati, diventando quindi molto più flessibile.
Un'altra importante differenza è che questo tipo di DB è decentralizzato e può essere scalato a dimensioni considerevoli senza troppi problemi, grazie al modo in cui è strutturato.

Prendiamo come esempio cardine il database Cassandra, che è un database NoSQL distribuito, decentralizzato, scalabile e ad "alta disponibilità". Questo significa che può essere fatto operare su macchine diverse per spezzare il carico, ma soprattutto che non segue il paradigma master-slave. In questo modo tutti i nodi (*client*) sono omogenei e hanno gli stessi privilegi. La decentralizzazione permette di garantire la disponibilità del sistema, ad esempio quando uno o più nodi dovessero essere irraggiungibili o non responsivi.
Si parla poi di scalabilità per gli stessi motivi, poiché aggiungere nodi alla rete e ridistribuire il carico è estremamente facile.
Cassandra sfrutta inoltre un linguaggio proprietario simile a SQL, denominato *CQL*, che risulta quindi facilmente comprensibile se si è abituati a lavorare con database relazionali.

Questo tipo di database viene utilizzato quando la costante disponibilità dei dati è una priorità, specialmente se tali dati sono in continua crescita e la loro struttura tende a

cambiare. Il costo da pagare come *tradeoff* per le potenzialità elencate è una minore *consistency* dei dati, ma l'argomento verrà approfondito nella sezione 3.3.

Figura 3.2: Logo di Cassandra

3.2.3 Document Store

Questo paradigma racchiude probabilmente la famiglia più ampia e utilizzata di database NoSQL, presentando comunque soluzioni diverse al suo interno.
L'idea principale è sempre quella di non fare uso di uno schema per strutturare i propri dati. Al posto di avere righe in una tabella, in questi database si usano *documents* raggruppati in *collections*. I documenti non seguono una struttura uniforme all'interno della stessa collezione, e sono quasi sempre salvati in formato `JSON` o simili.
In questo tipo di struttura la lettura dei dati è più rapida, a discapito di scrittura e update.
Generalmente sono di facile utilizzo e sono tra i database NoSQL più versatili.

Prenderemo come esempio MongoDB e CouchDB.
MongoDB è uno dei database NoSQL più popolari e diffusi, sfrutta documenti e collezioni, salva i propri dati in formato `BSON` (binary-json), e adotta il paradigma master-slave.
Funziona molto bene quando la funzione principale di cui si ha bisogno è il salvataggio di grosse moli di dati, mentre è meno performante se questi dati devono essere prelevati e rielaborati, specialmente quando è necessario unire dati provenienti da documenti o collezioni diverse.
CouchDB è piuttosto simile, sebbene sia un progetto meno popolare. Differisce da MongoDB per come salva i documenti (direttamente in formato `JSON`) e per l'architettura di base, che si distanzia dal paradigma master-slave e sfrutta nodi multipli per mantenere le informazioni sempre disponibili, a discapito della loro consistenza.

Figura 3.3: Logo di MongoDB a sinistra, CouchDB a destra

3.2.4 Graph Database

Esistono alcune categorie di database all'interno del panorama NoSQL che sono state sviluppate soddisfare necessità specifiche. Una di queste categorie è quella dei database

basati sui grafi.
Se nei RDBMS le relazioni sono sfruttate per collegare le tabelle, nel caso dei grafi le relazioni diventano vere e proprie entità, al pari dei nodi che esse collegano.
Questo consente di recuperare i dati con richieste (query) più concise e leggibili, oltre che ridurre i tempi di attesa.
Questo tipo di database funziona al meglio in situazioni dove le query si basano molto sulle relazioni tra i dati, dove un database relazionale dovrebbe operare molte operazioni di *join* che aumentano notevolmente il tempo di elaborazione.

Un esempio di database in questa categoria è Neo4j.
Come i database relazionali, Neo4j è *ACID compliant*, ovvero rispetta i parametri di atomicità, consistenza, isolazione e durabilità. Non è tuttavia un database distribuito e soffre quindi quando si parla di scalabilità.
Sfrutta un linguaggio molto intuitivo e simile a SQL per le query, chiamato *cypher*.

Come già detto, questo tipo di soluzione è utile in determinati campi e risulta molto interessante, ma può risultare inutile, o addirittura un ostacolo, se utilizzata in casi in cui non è necessaria.

Figura 3.4: Logo di Neo4j

3.2.5 Search Engine

Un'altra soluzione interessante, ma altrettanto specifica, è quella dei cosiddetti *full-text search engine*. Questo tipo di database è piuttosto simile, in superficie, a quelli basati sui documenti, con la differenza che il *search engine* analizza i contenuti del database e ne fornisce un indice. In questo modo la ricerca di dati, che ovviamente sfrutta tale indice, risulta estremamente rapida anche su dataset molto grandi, con la possibilità di implementare anche vari filtri per migliorare la *user experience*.

L'esempio più classico di implementazione di questo tipo di tecnologia è Elasticsearch, che è inizialmente nato come *search engine* e si è successivamente espanso per fornire le funzionalità classiche di un normale database.

Figura 3.5: Logo di Elasticsearch

3.3 CAP Theorem e semplicità di integrazione

Nel tentativo di categorizzare i database distribuiti, siano essi relazionali o meno, si sfrutta spesso il cosiddetto *CAP Theorem*, secondo cui un database distribuito può fornire soltanto due delle seguenti tre garanzie:

* **Consistency**: i dati sono sempre coerenti all'interno dei vari nodi distribuiti;
* **Availability**: i dati sono sempre disponibili in ogni momento;
* **Partition Tolerance**: i dati sono disponibili anche se partizionati, ovvero separati "orizzontalmente" su nodi diversi.

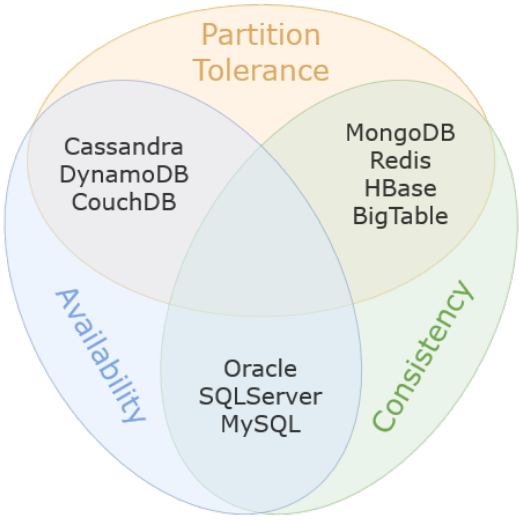


Figura 3.6: Rappresentazione del *CAP Theorem*, con posizionamento di alcuni database al suo interno in base alle loro caratteristiche

Mentre le soluzioni classiche (database relazionali) garantiscono Consistenza e Disponibilità, ma hanno grossi problemi con la scalabilità orizzontale e la separazione dei dati in cluster diversi, le soluzioni NoSQL tendenzialmente garantiscono il Partizionamento, e possono quindi soddisfare soltanto una delle altre due caratteristiche.

In base quindi alle necessità di Ifin Sistemi, sarà necessario valutare quale tipo di database adottare anche in base al *CAP Theorem*.

3.3.1 Integrazione

Tenendo in considerazione che lo scopo della ricerca che si è condotta rimane quello di studiare la possibilità di integrazione nello stack aziendale, è necessario valutare

quanto i database individuati si prestino a tale operazione.

Per ognuna delle implementazioni viste precedentemente è necessario seguire regole specifiche per mettere in comunicazione il proprio applicativo con il database.
A volte alcune soluzioni sembrano più funzionali al proprio caso d'uso, ma non hanno un sistema ben documentato, conosciuto e/o sviluppato per interfacciarsi con l'applicativo che si sta sviluppando, solitamente per motivi di compatibilità dei linguaggi e diffusione di librerie e driver utili.
A tale scopo si elencano i software individuati e le loro opzioni di integrazione, tenendo a mente che lo stack aziendale si basa sull'utilizzo di *Java* come linguaggio di programmazione.

Redis Stack

Per quanto riguarda Redis, Redis Stack e tutti i moduli ad esso collegati, il modo più semplice e "ad alto livello" per interagire con il database è sfruttare Redis OM, costruito sulla base di *Spring Data Redis*. Si tratta di un modulo del progetto *Spring Data*, che racchiude librerie specifiche per interagire con diversi database usando il framework *Spring*.

Cassandra

Cassandra può essere utilizzato in modo facile tramite Astra DB, un *multi-cloud DBaaS* costruito su Cassandra.
Per interagire con Astra si possono sfruttare le soluzioni più disparate, dall'utilizzo delle rest API all'implementazione di driver specifici per *Java* che ne facilitano l'utilizzo.

MongoDB

MongoDB è una delle implementazioni NoSQL più diffuse e versatili. Si può utilizzare Atlas come "*MongoDB as a Service*", e anche in questo caso viene fornita la rest API per effettuare operazioni sui dati. MongoDB fornisce driver per l'integrazione con *Java*, ma si può direttamente usare *Spring Data*, espressamente creato per fornire un modello familiare per datastores basato su *Spring*.

CouchDB

CouchDB non possiede librerie specifiche per l'integrazione e la comunicazione con applicativi scritti in *Java*. Quelle che esistono spesso non sono ufficiali e quindi utilizzabili "a proprio rischio e pericolo". Risulta possibile implementare la comunicazione direttamente tramite rest API, ma è dispendioso.
Esiste tuttavia una versione di CouchDB sviluppata da IBM, denominata *Cloudant*, che propone il servizio cloud e l'integrazione con *Java*.

Neo4j

Anche Neo4j ha recentemente fornito una propria piattaforma, lanciando il progetto Aura per fornire DBaaS, con supporto per integrazione con *Spring Boot* e *Spring Data*.

Elasticsearch

Elasticsearch sfrutta Elastic Cloud come soluzione distribuita.
Possiede varie funzionalità interessanti tra cui la possibilità di migrare i propri dati da un cloud all'altro, a prescindere dal provider.

3.4 Panoramica sui prodotti Ifin e sulle soluzioni RDBMS adottate in azienda

La seguente sezione è il risultato del dialogo avuto con i team leader di tre diversi gruppi di sviluppo interni all'azienda, necessario per comprendere più nel dettaglio il funzionamento dei software che essa produce e mantiene.
Nello specifico, si è ritenuto importante capire quali fossero le tecnologie coinvolte all'interno di questi prodotti, come essi comunichino con i database che sono al centro delle dissertazioni presenti in questa tesi, ed infine quali sono i colli di bottiglia che essi affrontano, per cominciare a farsi un'idea sui punti critici che una migrazione verso un sistema diverso potrebbe migliorare o risolvere.

3.4.1 LegalArchive

Il primo prodotto analizzato è LegalArchive, un applicativo che si occupa di archiviazione di documenti con importanza legale.
Questo software si occupa di prendere in carico i documenti che un'ente desidera conservare, e tradurli in informazioni che possano essere inserite in un database.
Questo è soltanto uno dei suoi scopi, ma è quello che più da vicino riguarda gli argomenti affrontati in tirocinio.
Per comunicare con i DB, che in questo caso sono distribuzioni enterprise di *Microsoft SQL Server* e *Oracle Server*, LegalArchive si serve di uno stack di strumenti che semplificano tale dialogo.
Il database viene montato su un server Tomcat, e per fare uso delle informazioni presenti all'interno di esso si sfruttano il connettore JDBC ed EclipseLink come implementazione della specifica JPA.
L'architettura si basa sul pattern Model-View-ViewModel, sfruttando DTO e DAO all'interno del modello per costruire oggetti *Java* derivanti dagli elementi del database.
Più semplicemente, ogni elemento di una tabella nel database ha una diretta corrispondenza con un oggetto di una classe all'interno del codice, in modo da poter manipolare tali elementi in base alle necessità.
I problemi più grossi che creano rallentamenti nel servizio sono legati alle query utilizzate per richiedere i dati al database.
Può infatti capitare che queste vengano testate su campioni troppo piccoli, rischiando poi di dare problemi o "rompersi" quando lavorano sulle grosse moli di dati (fino a circa 200.000 inserimenti di nuove righe ogni ora) con cui il software ha a che fare ogni giorno.
Un altro punto critico di questo sistema è il modo in cui si affronta l'aumentare dei dati di cui esso si deve occupare. Finora questo problema è stato affrontato implementando tre diversi approcci di partizionamento:

* Partizionamento lato Documenti, dividendo le tabelle più grosse in varie tabelle ordinate con un indice, per facilitarne la gestione;

* Partizionamento lato Application, operato sui database da parte dell'azienda;

* Partizionamento su diversi database paralleli.

Dalle informazioni raccolte emerge quindi come i problemi principali per questo prodotto siano la scalabilità (finora soddisfatta sfruttando vari sistemi di partizionamento delle tabelle) e la rapidità di scrittura e persistenza di nuovi dati.

Figura 3.7: Logo di LegalArchive

3.4.2 InvoiceChannel

InvoiceChannel è il secondo prodotto per importanza all'interno dell'azienda, ed è strettamente legato a LegalArchive.
Si tratta di un software che fa da ponte tra le aziende ed il Sistema di Interscambio, un servizio dell'Agenzia delle Entrate che gestisce la fatturazione elettronica.
Si presenta come una piattaforma tramite cui inserire dati e documenti perché questi vengano inoltrati al SdI ed elaborati come necessario. All'interno di InvoiceChannel è presente anche l'opzione di conservare le fatture inserite assieme con eventuali allegati all'interno di LegalArchive.
Dal lato tecnico, anche per quanto riguarda InvoiceChannel vengono sfruttati sia database Oracle che Microsoft.
Sia per quanto riguarda l'ambiente di produzione che negli ambienti dei clienti è presente un'unica istanza, non distribuita nel cloud.
Anche InvoiceChannel si basa su Tomcat e JDBC per la comunicazione tra applicativo e database. Come per Legal Archive, anche per questo applicativo le preoccupazioni maggiori ricadono su scalabilità e performance in condizioni di utilizzo massivo, date le grosse moli di dati in continuo aumento.
Anche l'ottimizzazione dei tempi di attesa per determinate operazioni *time-sensitive* è un tema caldo, su cui si sta ancora lavorando.

Figura 3.8: Logo di InvoiceChannel

3.4.3 Dialogo con il team di test

Dopo aver consultato i due team che si occupano in modo diretto dello sviluppo e del mantenimento del software, si è ritenuto opportuno avere un confronto ulteriore con il team che all'interno dell'azienda si occupa di effettuare test su tutte le piattaforme, in modo da avere un'idea più concreta sugli strumenti e sulle metodologie da adottare.
L'approccio proposto consisterà nel costruire due database, uno relazionale e uno secondo il paradigma NoSQL, per simulare l'ambiente dei prodotti di Ifin ed effettuare una serie di query volte al raccoglimento di dati che rappresentino le prestazioni di tali database.
In questo modo sarà possibile determinare, dati alla mano, quali possono essere vantaggi e svantaggi delle tecnologie coinvolte.

3.5 Conclusione della fase di indagine

Si evince dall'analisi eseguita e dai dialoghi avuti con i vari team leader che lo stack tecnologico utilizzato in Ifin è ben consolidato e strettamente legato ai processi. Per questo motivo, l'idea di sostituire parti di esso rappresenta una sfida non indifferente e necessita di una valutazione attenta di vantaggi e svantaggi, poiché apportare modifiche di questa taglia e importanza richiederebbe uno sforzo immane.
Per quanto riguarda i problemi che le tecnologie in questione sono costrette ad affrontare quotidianamente, emerge una comune preoccupazione per scalabilità dei sistemi e performance delle operazioni in rapidità ed affidabilità.
Il fatto che questi siano i punti forti di molte soluzioni NoSQL giustifica lo studio di fattibilità che si sta eseguendo, nonostante l'effettiva problematicità di una eventuale ristrutturazione dei principali prodotti di Ifin.

Dato quanto visto nei capitoli precedenti, si è scelto MongoDB come database NoSQL da adottare per effettuare il confronto con i database relazionali. Oltre a proporre soluzioni alternative ai problemi principali in cui tali database incorrono, MongoDB è estremamente popolare, ben documentato, ed è già stato approcciato all'interno di Ifin per altri progetti al di fuori dei suoi prodotti principali. Per questi motivi è sembrato una buona scelta ed un ottimo rappresentante del mondo NoSQL.

Capitolo 4

Metodologie

4.1 Strumenti e tecnologie utilizzate

Per poter eseguire il confronto tra le prestazioni di database relazionali e NoSQL è stato necessario preparare tutta una serie di strumenti utili a far funzionare i database stessi e al monitoraggio dei dati contenuti al loro interno.

4.1.1 Docker

Per poter effettuare test consistenti, evitare problemi di installazione e compartimentalizzare l'ambiente di lavoro, si è deciso di sfruttare Docker. Questo software permette di virtualizzare l'esecuzione di altri applicativi in ambienti chiusi e controllati, facilitando determinati processi di sviluppo. L'utilizzo di Docker in questo progetto lo rende più maneggevole e lineare.

Per poter utilizzare Docker è necessario prima comprendere alcune parole chiave e ciò che esse rappresentano:

* *Dockerfile*
* *Image*
* *Container*

Il dockerfile racchiude una serie di istruzioni che servono a Docker per sapere come costruire un'immagine.
Questa, a sua volta, è uno snapshot del software che vogliamo utilizzare, comprese tutte le sue dipendenze, fino al sistema operativo. Questa immagine viene messa all'interno di un container per poter utilizzare il suddetto software.
Le immagini di molti software sono disponibili online e possono essere utilizzate come template da cui partire all'interno del dockerfile.
Una volta creata un'immagine è possibile usarla in quanti container si desidera, e la si può mettere a disposizione di altri utenti perché possano utilizzare lo stesso ambiente su macchine diverse.
In questo senso Docker è di fatto molto simile ad una *Virtual Machine*, con la differenza (cruciale) che mentre all'interno di una *VM* viene fatto girare un sistema operativo "ospite", ed ogni macchina virtuale ne utilizza uno a sè stante, i container di docker condividono lo stesso *kernel*, il Docker Engine, rendendo questo sistema molto più

leggero e veloce.
Docker ci permette quindi di creare più container che funzionano parallelamente e possono comunicare tra loro, rimanendo indipendentementi dagli altri container e dal sistema operativo in cui eseguiamo il Docker Engine.

Nel caso di questo progetto, Docker risulta utile innanzitutto per gestire il funzionamento dei due database (che saranno PostgreSQL e MongoDB), che verranno quindi inseriti in appositi container.
Per facilitare poi i processi di comunicazione con i database verranno utilizzati altri due container contenenti delle interfacce grafiche per la gestione dei dati (rispettivamente pgAdmin e mongo-express).

Figura 4.1: Logo di Docker

4.1.2 PostgreSQL e tecnologie annesse

Dopo aver scelto Docker come ambiente in cui montare i database, alcune altre scelte legate alle tecnologie coinvolte in questa ricerca sono state prese di conseguenza.
Sebbene infatti sia possibile *dockerizzare* moltissimi software, per gli scopi di questa tesi è risultato più sensato partire da immagini pre-esistenti ed affidabili, disponibili sulla piattaforma ufficiale del software (hub-docker).
Questo ha permesso di non investire troppo tempo nella preparazione degli strumenti e di concentrare le risorse disponibili nell'effettivo confronto tra database.

Per tutti questi motivi si è quindi scelto di usare PostgreSQL come database relazionale da portare a confronto con MongoDB.
PostreSQL è un database relazionale *open source* molto robusto che si presta bene per rappresentare le prestazioni di un generico database di questa categoria. Esso è inoltre presente nell'hub di docker con un'immagine ufficiale, ed è quindi facile da far funzionare all'interno di un container a differenza di altri prodotti simili.
Sebbene PostgreSQL non sia utilizzato massivamente all'interno dell'azienda, risulta comunque molto simile alle sue controparti non open source, *MSSQL* e *Oracle Server*, che sono invece largamente usate all'interno di Ifin e sarebbero oggetto di una potenziale migrazione qualora i risultati di questa tesi si rivelassero convincenti.

Vale la pena di evidenziare che a differenza delle tecnologie che ricadono sotto la sigla NoSQL, quelle legate ai database relazionali sono molto più simili tra loro, poichè condividono appunto il paradigma relazionale che sta alla base di tutte le variazioni esistenti proposte da aziende diverse in contesti diversi.
In questo senso, usare PostgreSQL non è una scelta incoerente con i motivi che hanno dato alla luce questo progetto.

Oltre al container utilizzato per il database di PostgreSQL, è necessario poter utilizzare uno strumento per effettuare operazioni su di esso.
Come spiegato in fase di analisi dei software proprietari di Ifin, questi sfruttano diversi framework e paradigmi per poter comunicare con i database ed operare su di essi le operazioni CRUD necessarie al funzionamento del software stesso. Considerato l'obiettivo di questa ricerca non avrebbe tuttavia senso investire tempo nel setup di un software dedicato a questi scopi.
Esistono infatti delle interfacce che ci permettono di interagire "manualmente" con i database senza fare uso di codice ed altre infrastrutture nel mezzo.
Per quanto riguarda PostgreSQL, la scelta è ricaduta su pgAdmin, un'interfaccia molto popolare e ben mantenuta, disponibile come immagine nella hub di docker per essere utilizzata all'interno di un container. È quindi sufficiente mettere in comunicazione i due container, uno per PostgreSQL e uno per pgAdmin, per avere a disposizione un ambiente completo in cui effettuare test su database relazionali.

Figura 4.2: Logo di PostgreSQL

4.1.3 MongoDB e tecnologie annesse

Come già anticipato, per mettere a confronto le potenzialità dei database NoSQL rispetto a quelli di tipo relazionale è stato scelto MongoDB.
Anche in questo caso la scelta è stata guidata da più fattori che rendono MongoDB il candidato perfetto per questa ricerca.
Lo schema dinamico su cui esso si basa, assieme alla struttura dei dati che permette di immagazzinare, rappresentano fattori che potrebbero migliorare il modo in cui le informazioni vengono tutt'ora gestite ed archiviate da Ifin.
La popolarità di MongoDB determina inoltre un'abbondanza di guide e documentazione a cui fare riferimento per condurre un'analisi accurata delle sue potenzialità.
Infine, anche per questo database esiste un'immagine, mantenuta ufficialmente dal team di MongoDB, per poterlo utilizzare come container all'interno di Docker.

Per quanto riguarda l'interazione con il database e l'utilizzo di operazioni CRUD al suo interno, sono stati utilizzati principalmente due strumenti.
In una fase iniziale in cui la necessità principale era quella di imparare e testare la sintassi delle operazioni sul database, un'interfaccia molto semplificata in grado di effettuare solo operazioni di inserimento e cancellazione dei dati è stata sufficiente. Si è quindi ricorso all'uso di mongo-express, un'interfaccia grafica molto basilare, il cui più grande pregio è quello di essere un software *dockerizzabile*, eseguibile assieme al container dedicato a MongoDB e connesso ad esso seguendo le regole già apprese per connettere interfaccia e database di PostreSQL.
In un secondo momento, tuttavia, tale interfaccia non è più risultata sufficiente, dato il tipo di analisi che si è voluto condurre sui risultati delle query e sul loro tempo di

esecuzione.
Si è quindi deciso di passare all'utilizzo di un'altra interfaccia grafica, MongoDB Compass, questa volta esterna all'ambiente di Docker, installata direttamente sulla postazione fornita da Ifin durante il tirocinio.
Compass è un software sviluppato direttamente dal team di MongoDB che mette a disposizione sia un'interfaccia grafica di facile utilizzo, sia un terminale da cui eseguire le operazioni più a basso livello, per poter utilizzare tutte le funzionalità che MongoDB ha da offrire.

Figura 4.3: Logo di MongoDB

4.1.4 Altri strumenti

Oltre a quelli principali visti nelle sezioni precedenti, sono stati unilizzati altri strumenti di base per la stesura di appunti ed analisi, quali fogli di calcolo e documenti di testo, assieme ad altri software per la creazione dei dati di test da inserire nei database, che verrranno descritti in maggiore dettaglio nei capitoli successivi.

4.2 Selezione di un ambito di confronto

L'idea generale per effettuare il confronto tra le prestazioni di database relazionali e database NoSQL è di creare un Proof of Concept in grado di gestire i dati di uno dei software di punta dell'azienda (o comunque dati molto simili ad essi per quanto riguarda mole e contenuto), e di fare questo lavoro per entrambi i tipi di database. Una volta fatto ciò, si potranno elaborare delle query che mettano a confronto le prestazioni.

Viste le considerazioni fatte sui due software "fratelli", e considerato il risultato dei dialoghi avuti con i team leader di entrambi i progetti, la scelta è ricaduta su InvoiceChannel. Il motivo principale di questa scelta è che sebbene i due software siano piuttosto simili nella loro parte di *backend*, legata ai database, InvoiceChannel è un software leggermente più recente e per questo meno convoluto di LegalArchive.
Sebbene questi siano progetti commerciali, dotati di estensiva documentazione, è bene considerare che nel tempo allocato per effetturare lo stage non sarebbe stato possibile approfondire nel dettaglio tutti gli aspetti di questi software.
Per lo stesso motivo, anche una volta scelto InvoiceChannel è stato necessario condurre uno studio per identificare una piccola parte del database che esso utilizza, su cui basarsi per costruire i database necessari ai test di confronto.

4.3 Modellazione del database relazionale

Una volta definite queste premesse, è stato effettuato uno studio approfondito della struttura del database di InvoiceChannel, ricostruendo le relazioni tra tabelle a partire da un file in formato `SQL` fornito dal team di test.
Esso contiene più di ottantamila righe di codice e definisce la struttura, il contenuto e le relazioni tra le circa 150 tabelle che compongono un database utilizzato per effettuare test di vario tipo all'interno dell'azienda.

Risulta ovvio come InvoiceChannel sia troppo grande e complesso per poter essere rimodellato nella sua interezza in questo contesto. Anche pensare di utilizzarne soltanto una sezione risulta piuttosto complesso, poichè tutte le tabelle sono estremamente interconesse.
Un ulteriore livello di complessità viene posto dalle query, anch'esse molto convolute, che fanno spesso un uso massiccio di operazioni di *join*.
L'opzione migliore è quindi ricostruire effettivamente un pezzetto di database di InvoiceChannel "ad-hoc", per soddisfare le necessità del progetto, sia in formato RDBMS, sia in formato NoSQL.

4.3.1 Definizione di tabelle e relazioni

La sezione in questione è quella relativa alle fatture, che risulta piuttosto centrale allo scopo del software. Le tabelle coinvolte nel diagramma entità-relazione sono quelle che si occupano di immagazzinare i dati relativi alle fatture elettroniche, lo stato dei processi che le coinvolgono, i documenti ad essa collegati e l'ente che l'ha generata.

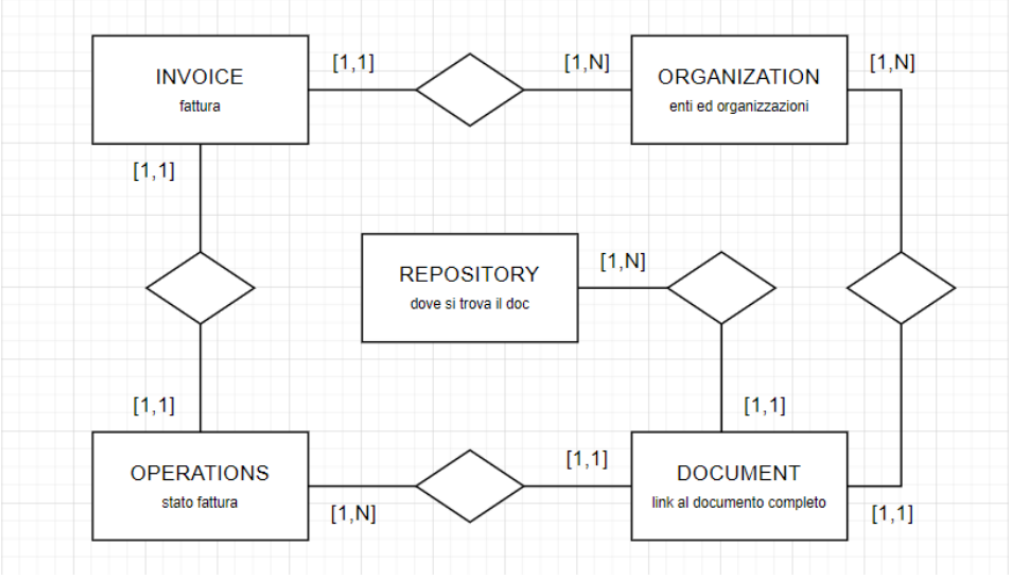


Figura 4.4: Diagramma ER per la sezione semplificata di InvoiceChannel

Dopo aver identificato le tabelle su cui concentrarsi per condurre lo studio e le relazioni che intercorrono fra di esse, si è passati alla definizione dei campi che compongono le suddette tabelle. Esse infatti devono essere ridimensionate per rispecchiare la sezione di database ridotto che è stata ricostruita.

4.3.2 Script ridotti per la creazione delle tabelle

Viene di seguito riportato il contenuto delle tabelle ristrutturate.

```
CREATE TABLE INVOICE(
    ID numeric(22, 0) NOT NULL,
    ID_OPERATION numeric(22, 0) NOT NULL,
    INVOICE_NUMBER Varchar(60) NULL,
    INVOICE_DATE Timestamp(3) NULL,
    TOTAL_AMOUNT numeric(25, 9) NULL,
    RECIPIENT_NAME varchar(255) NULL,
    DATA_INVIO Timestamp(3) NULL,
    SOURCE_FILE_NAME Varchar(255) NULL,
    TIPO_FATTURA Varchar(6) NOT NULL,
    DATA_RICEZIONE Timestamp(3) NULL,
    CAUSALE Varchar(200) NULL,
    MITTENTE Varchar(200) NULL,
    ID_ORGANIZATION numeric(22, 0) NULL,
    IS_B2B numeric(1, 0) NULL,
    TIPO_DOCUMENTO char(4) NULL,
    RIEPILOGO_IVA Text NULL,
 CONSTRAINT PK_INVOICE PRIMARY KEY ( ID )
);
```

La tabella `INVOICE` rappresenta la fattura, con tutti i dati e i codici che essa contiene. Nella sua versione ridotta sono stati eliminati molti campi che facevano riferimento a tabelle non più collegate o dati specifici appartenenti all'ambito della fatturazione elettronica non ritenuti necessari agli scopi di questa tesi.

```
CREATE TABLE DOCUMENT(
    ID numeric(22, 0) NOT NULL,
    ID_OPERATION numeric(22, 0) NOT NULL,
    CREATION_DATE Timestamp(3) NOT NULL,
    COD_DOCUMENT Varchar(255) NOT NULL,
    PATH_FILE varchar(1024) NULL,
    ID_REPOSITORY numeric(5, 0) NULL,
    DOCUMENT_SIZE numeric(22, 0) NULL,
    ID_ORGANIZATION numeric(22, 0) NULL,
 CONSTRAINT PK_DOCUMENT PRIMARY KEY ( ID )
);
```

La tabella `DOCUMENT` contiene una serie di dati relativi al contenuto di un documento. Ogni elemento della tabella `INVOICE` fa riferimento infatti ad uno o più documenti, tra cui la fatura vera e propria e potenzialmente altri allegati

```
CREATE TABLE OPERATIONS(
    ID numeric(22, 0) NOT NULL,
    CREATION_DATE Timestamp(3) NOT NULL,
    MESSAGE Text NULL,
 CONSTRAINT PK_OPERATIONS PRIMARY KEY ( ID )
);
```

La tabella `OPERATIONS` rappresenta lo stato di processo in cui si trova fattura all'interno del sistema. Man mano che il software processa la fattura, il suo stato viene aggiornato all'interno di questa tabella.

```
CREATE TABLE ORGANIZATION(
    ID numeric(22, 0) NOT NULL,
    NAME Varchar(255) NOT NULL,
    ALIAS varchar(100) NULL,
    PARTITA_IVA Varchar(255) NULL,
    COD_ORGANIZATION_TYPE Varchar(10) NULL,
    TIPO_SOCIETA Varchar(10) NULL,
    CREATION_DATE Timestamp(3) NULL,
 CONSTRAINT PK_ORGANIZATION PRIMARY KEY ( ID )
);
```

La tabella `ORGANIZATION` contiene informazioni riguardanti gli enti e le aziende che fanno uso del software, per poter tracciare la provenienza di una fattura e compiere moltissime altre operazioni all'interno dell'applicativo.

```
CREATE TABLE REPOSITORY(
    ID numeric(5, 0) NOT NULL,
    DEFINITION Varchar(255) NULL,
    COD_REPOSITORY_TYPE varchar(10) NULL,
 CONSTRAINT PK_REPOSITORY PRIMARY KEY ( ID )
);
```

La tabella `REPOSITORY` contiene informazioni riguardanti la localizzazione degli archivi che contengono i documenti.

4.4 Modellazione del database in MongoDB

In seguito alla costruzione del database relazionale che semplifica InvoiceChannel, si può iniziare a costruirne la controparte NoSQL seguendo il modello documentale per implementarla con MongoDB.

4.4.1 Relazioni tra tabelle

Sebbene sia un modello documentale, Mongo mantiene comunque la capacità di mettere in relazione documenti provenienti da collezioni diverse. Bisogna tuttavia farlo nel modo corretto perché questo porti dei vantaggi.

Le possibili relazioni esistenti sono tre:

* One to One
* One to Many
* Many to Many

Esiste inoltre un altro concetto proposto da mongo per rappresentare moli di dati enormi: si tratta dello "zillion", per indicare un "many" ancora più grande. Le vediamo con ordine.

One to One

Sono le relazioni che intercorrono tra campi della stessa tabella, ma vengono usate anche per connettere tabelle diverse, per separare i dati secondo una determinata logica.
In MongoDB è possibile mantenere separati documenti che sono in relazione *one to one*, usando quindi degli identificatori, oppure è possibile incorporare i documenti uno nell'altro (*embedding*).
Riassumendo:

* Quando possibile è meglio incorporare, per non introdurre complessità inutile.

One to Many

Sono relazioni in cui un'entità fa riferimento a molteplici altre entità, mentre queste altre possono riferirne soltanto una.
Anche in questo caso ci sono due possibili approcci, o si fa *embedding*, o si mantengono le relazioni tramite indici.
Nel primo caso si può inserire il documento singolo in ogni elemento del gruppo di documenti riferiti, o in alternativa si può inserire nel singolo documento tutto il gruppo con cui era relazionato. Solitamente si fa *embedding* nell'entità che subisce più query.
Se si decide invece di gestire la relazione con un indice, si può inserire una reference a tale indice, nel lato "one" o nel lato "many". Solitamente si sceglie il lato "many".
Riassumendo:

* Quando possibile è meglio preferire l'operazione di *embedding*, per favorire la semplicità, soprattutto se si può fare sulla collection maggiormente colpita da query.

* Va bene usare reference quando si mettono in relazione documenti che non hanno lo stesso grado di query associate.

Many to Many

Le relazioni "molti a molti" sono solitamente più complesse, ma possono/devono essere semplificate nel modello relazionale introducendo una tabella di congiunzione in modo da spezzare la relazione in due relazioni "molti a uno"; Nel modello documentale questo non è strettamente necessario, ma è sempre giusto chiedersi se la relazione sia fondamentale così com'è o se si può semplificare. Riassumendo:

* Preferire *embedding* nel lato che subisce più query.
* Preferire *embedding* per informazioni che sono primariamente statiche e che possono funzionare bene anche in caso di duplicazione;
* Preferire invece l'uso di reference per evitare di dover avere a che fare con la duplicazione, quando questo è un effetto indesiderato.

Zillions

Con questo termine si vuole semplicemente identificare un subset delle relazioni *one to many*, in cui "many" è esponenzialmente grande. Serve un occhio di riguardo per evitare operazioni che recuperano tutti gli elementi coinvolti nella relazione, poiché questo andrebbe ovviamente a creare dei grossi rallentamenti. Per questo motivo, in caso di relazioni fatte così non è possibile pensare a soluzioni incorporate, e si può solo mantenere la relazione inserendo la reference nel lato "zillions".

4.4.2 Pattern

Un po come i design pattern nella programmazione ad oggetti, i pattern di modellazione dei dati sono utili per rendere questi modelli più comprensibili, basandosi su principi provati.
Per quanto riguarda questo progetto, andremo inizialmente a verificare quali sono i pattern consolidati e utili per lavorare con MongoDB, e successivamente determineremo se sarà consono applicarne qualcuno. Non avrebbe senso infatti applicarli a prescindere, se il progetto si rivelasse essere già abbastanza semplice così com'è.

Prima di tutto, è necessario quindi specificare che l'utilizzo di pattern può causare l'introduzione di *duplication*, *staleness* e *integrity issues* nei dati interni al modello.
Questi possono essere un deterrente, specialmente nel caso di progetti semplici, ma se affrontati nel modo giusto rendono l'utilizzo dei pattern una scelta proficua.

Primo Problema: Duplication

Può essere il risultato dell'inserimento di documenti all'interno di altri documenti, per velocizzare le query. Spesso è una cosa negativa, se fatta senza badare alle conseguenze, ma non sempre è impossibile da gestire, anzi.

Ci sono casi in cui la duplicazione può essere desiderabile. Per esempio se si fa *embedding* dell'indirizzo di spedizione nell'ordine di un prodotto, questo sarà una copia

dell'indirizzo dell'utente. Tuttavia l'informazione è statica e anche se l'utente cambiasse indirizzo in futuro questo non influenzerebbe gli ordini passati e già ricevuti.
Questo porta ad una soluzione migliore rispetto al dover referenziare l'indirizzo, cosa che diventerebbe lenta e non necessaria per lo scopo dell'applicativo.

Altro caso è quello della duplicazione di informazioni che hanno sì un peso, ma questo è talmente piccolo che non provoca problemi. Per esempio incorporare gli attori all'interno dei film in cui recitano è una soluzione accettabile, poiché anche in questo caso si tratta di informazioni statiche che non dovranno essere cambiate, e poco importa se lo stesso attore dovrà essere reinserito in molteplici film. Rimane una soluzione migliore rispetto al dover gestire una relazione *many to many* tra attori e film in cui compaiono.

Ultimo caso, molto più importante, è quello in cui la duplicazione c'è ed è fastidiosa. Questo è spesso il caso di dati che devono essere aggiornati nel tempo. Avere più copie di questi dati porta ad un *overhead* di lavoro per mantenerli coerenti. In questi casi è necessario valutare tale *overhead* a confronto con una soluzione che non usa il pattern in questione (che sta provocando la necessità di duplicazione), per capire quale direzione prendere.

Secondo Problema: Staleness

Con questo termine si intende "l'andare a male" dei dati, che diventano obsoleti se aggiornati troppo raramente. Fornire dei dati "stantii" non è certo una cosa desiderabile. Si tratta di un problema che affligge i sistemi moderni a causa della grande velocità con cui i dati vengono recuperati e aggiornati. A causa di ciò non si può avere la certezza di avere l'ultimo dato aggiornato.
La domanda da porsi quindi è: "quanta staleness è accettabile?". Ovviamente dipende dal tipo di dato.
La soluzione è usare *batch updates* in modo da avere la sicurezza di fare update in un tempo fissato, e per avere la possibilità di visionare gli update tramite uno stream.

Terzo Problema: Referential Integrity

Solitamente si hanno problemi con questo aspetto quando si eliminano parti di documenti o tabelle, senza eliminare le loro references. Non c'è quindi la possibilità di applicare un "on cascade - delete", che daremmo per scontata nei database relazionali. MongoDB non supporta questa funzione quindi è necessario che sia l'applicazione ad occuparsene attivamente.

Riassumendo quindi quanto visto, per ogni dato che intendiamo maneggiare dobbiamo chiederci quanto segue:

* È necessario/utile duplicare quest'informazione?
  - Risolvere usando *batch updates*.
* Qual'è il livello accettabile di *staleness*?
  - Risolvere con update basati sul *change stream*.

* Quali parti del dato necessitano maggiormente di *referential integrity*?
  - Risolvere o prevenire implementando transazioni o *change stream*.

Andiamo ora a vedere nello specifico quali sono i pattern che si possono applicare ad uno schema documentale, e quali sono i vantaggi che possono apportare.

Attribute Pattern

In alcuni casi, nel costruire un documento, può sorgere la necessità di creare molti campi simili ma non identici (e quindi non appartenenti ad uno stesso array, per esempio). Se poi si volesse poter cercare informazioni all'interno di questi campi in modo simultaneo, questo diventa complesso.
Per risolvere questo problema si usa il pattern degli attributi. L'idea è di raggruppare in qualche modo questi campi che di fatto sono simili e contengono informazioni che ha senso mantenere raggruppate.

```
{
  "title": "Dunkirk",
  ...
  "release_USA": "2017/07/23",
  "release_Mexico": "2017/08/01",
  "release_France": "2017/08/01",
  "release_Festival_San_Jose":
   "2017/07/22"
}
```

```
{
  "title:" "Dunkirk",
  ...
  "releases": [ {
    { "k": "release_USA",
      "v": "2017/07/23" },
    { "k": "release_Mexico",
      "v": "2017/08/01" },
    { "k": "release_France",
      "v": "2017/08/01" },
    { "k": "release_Festival_San_Jose",
      "v": "2017/07/22" }
  ]
}
```

Figura 4.5: Esempio di utilizzo dell'attribute pattern

In questo modo si possono inserire indici ed effettuare ricerche su `releases.v` (per esempio).
I casi d'uso più comuni sono il raggruppamento di caratteristiche di un prodotto, o di un set di campi che hanno lo stesso tipo (come visto sopra per le date).

Extended Reference Pattern

Questo pattern è tra quelli che generano duplicazione nei dati, e va quindi usato con cura. L'idea è di migliorare le operazioni che coinvolgono relazione *One to Many*.
Immaginiamo di avere una relazione di quel tipo in cui, per collegare due documenti, è presente un id nell'entita "One" e una reference a tale id in ogni entità nel lato "Many". Se le operazioni che coinvolgono questi due documenti sono molto frequenti e richiedono delle informazioni che esistono solo nel lato "Many", saranno necessari molti *join* che rallentano per forza di cose l'utilizzo del database.
Per ovviare a questo problema si decide quindi di rendere la reference non soltanto un campo del documento "Many", ma espanderla in un sottodocumento che contenga il riferimento all'id e alcuni dei campi del documento "One", quelli più frequentemente

utilizzati nelle query, in modo da evitare il grosso dei *join*.
Tutto questo, come dicevamo, produce duplicazione di dati. Capiamo ora come gestirla.
Prima di tutto va minimizzata:

* scegliendo di duplicare quei dati che non cambiano frequentemente;
* duplicando solo i dati necessari ad evitare le operazioni di *join*.

Quando avviene un aggiornamento dell'originale:

* capire quali sono le reference estese da aggiornare a loro volta;
* capire quando queste reference dovrebbero essere aggiornate.

È bene inoltre tenere a mente che a volte la duplicazione è desiderabile, rispetto all'utilizzo di una normale "singola" reference.

Subset Pattern

Anche questo pattern si occupa di migliorare le prestazioni del database, questa volta concentrandosi su problemi di *working set*.
Quando l'insieme di dati su cui si deve lavorare è più grande dello spazio disponibile ad accesso veloce (per esempio in RAM), il continuo scambio di informazioni con la memoria diventa un collo di bottiglia.
Per ovviare a questo problema, notiamo innanzitutto come in determinati casi il *working set* è sì troppo grande, ma è anche utilizzato solo parzialmente. Nei campi di un documento "film" troviamo la lista di tutto il cast, ma raramente vorremo visualizzarlo per intero. Ci basterà una lista dei 10 o 20 attori più importanti, nella maggior parte dei casi. Stessa cosa vale per le recensioni, e altri campi di questo tipo.
In questi casi risulta quindi utile creare un *subset* di questi campi, in modo da includere solo la parte importante di queste grandi liste all'interno del *working set*, e lasciare il resto in memoria per accedervi solo quelle rare volte che servirà, lasciando spazio a informazioni più importanti.
Si tende quindi a dividere le *collections* in due segmenti, quello con i dati più importanti e quello con i dati che ricevono meno accessi.

Computed Pattern

Questo pattern si basa sulla comprensione del peso che le operazioni di lettura e scrittura hanno sul sistema che si sta utilizzando.
In particolare, quando vengono effettuate delle operazioni (somme o altri calcoli) in fase di lettura su un set di informazioni molto ampio, questo può rendere tale lettura estremamente lenta. L'idea è quindi di tenere traccia di tali operazioni (salvandone il risultato) per poi modificarle solo in fase di aggiunta di nuove informazioni, in modo che durante la lettura il dato sia già accessibile senza nuove operazioni necessarie.

Bucket Pattern

Il Bucket Pattern offre una via di mezzo tra dover contenere tutte le informazioni in un solo documento, e avere un documento per ogni informazione granulare.
Solitamente viene utilizzato per l'IoT, dove grosse moli di dati vengono prodotte da

sensori e rilevatori. Spesso si usa la data come discriminante per creare il *bucket* in cui raggruppare le informazioni.
Ci sono anche degli svantaggi nell'introdurre questo pattern, legati al fatto che ora i dati sono separati in gruppetti, ed effettuare operazioni su tutti i dati diventa più complesso.

Schema Versioning Pattern

Questo pattern entra in gioco quando inevitabilmente arriviamo al punto di dover fare un upgrade alla struttura (*schema*) dei documenti. Questo processo può essere dispendioso e complesso, soprattutto per migrare i dati già esistenti nel nuovo schema. Il pattern propone di procedere come segue:

* Ogni documento riceve un campo *schema-version* che identifica la versione dello schema su cui si basa, con cui è stato costruito;
* L'applicazione deve poter gestire tutte le versioni di schema presenti a database, per poter gestire il cambiamento in-itinere senza dover bloccare tutto durante la transizione ed effettuarla in un unico momento;
* Si sceglie una strategia di migrazione dei dati, che sia essa aspettare l'update dei singoli documenti per altri motivi e sfruttare l'occasione per mutare anche lo schema, oppure fare dei *batch update* appositamente dedicati a questa modifica.

Questo pattern è particolarmente utile quindi quando non ci si può permettere di avere *downtime* sul sistema, che deve continuare a funzionare in modo costante.

Tree Patterns

Il modello documentale si presta bene per rappresentare modelli gerarchici di dati.
All'interno di questo tipo di rappresentazione, esistono alcuni pattern utili per migliorarne l'utilizzo. Come per gli altri pattern, anche in questo caso è importante capire quali siano le necessità del sistema che stiamo modellando, in modo da fare delle scelte accorte.
I pattern disponibili sono:

* Parent References
* Child References
* Array of Ancestors
* Materialized Path

Ognuno ha vantaggi e svantaggi.
Nel caso di Parent References, si inserisce un campo "genitore" in ogni nodo, che indica appunto da quale nodo discende.
Nel caso di Child References è invece presente un array contenente tutti i "figli immediati", i nodi che sottostanno a quello corrente.
Array of Ancestors consiste in un campo (un array appunto) che contiene il padre del nodo corrente, il padre del padre, e via dicendo fino alla radice della struttura.
Materialized Path è leggermente diversa dagli altri come soluzione, ma consiste nel salvare come stringa un campo contenente gli antenati del nodo corrente separati

con un *value separator*, quindi per esempio il punto. In questo modo si facilitano le operazioni, che possono sfruttare quel campo in una *regular expression*.
Ovviamente si possono applicare più pattern contemporaneamente, in base alle operazioni da svolgere, perchè ogni pattern funziona bene in determinati casi, ma può avere difficoltà in altri.

Polymorphic Pattern

È un pattern semplice che sta alla base di altri pattern visti in precedenza. Affronta il problema di accorpare oggetti simili ma diversi nella stessa collezione, tenendo traccia del tipo specifico di oggetto tramite un campo apposito.
Ci saranno quindi una serie di campi di base condivisi tra gli oggetti e altri campi specifici, a volte contenuti in subdocuments.

Riepilogo

È chiaro come in base al caso d'uso, l'utilizzo di un pattern può migliorare l'architettura che sta alla base del database.

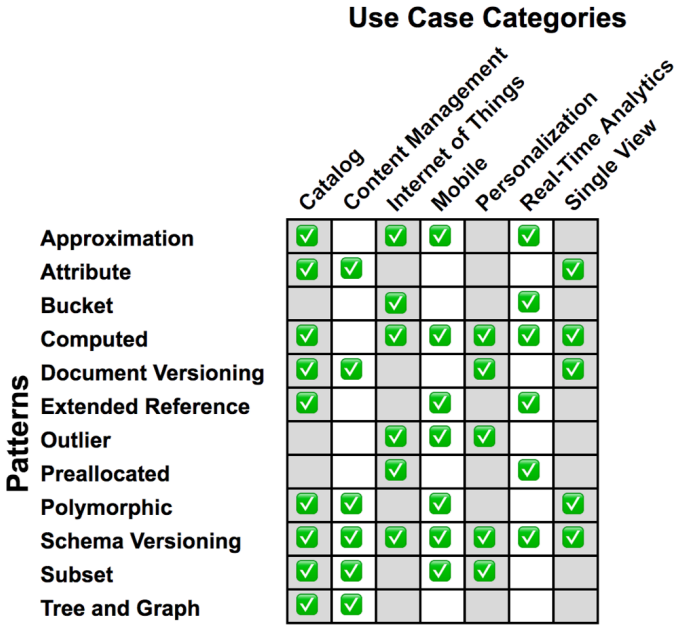


Figura 4.6: Applicabilità dei pattern in base al caso d'uso

4.4.3 Migrazione del database relazionale nel modello NoSQL

Il database relazionale costruito ed inserito all'interno di PostgreSQL è descritto dalla Figura 4.4. Per costruire l'architettura dei documenti che risiederanno all'interno del

database NoSQL, vengono riassunti i contenuti di tale figura:

* `INVOICE` contiene dei dati relativi alla fattura inserita dall'utente;
* `OPERATIONS` contiene un messaggio relativo allo stato di tale fattura;
* `DOCUMENT` contiene il path ai vari documenti legati alla fattura, come un file di notifica del SdI, la fattura stessa, allegati, ecc;
* `REPOSITORY` contiene i riferimenti ai repository utilizzati per il salvataggio dei documenti;
* `ORGANIZATION` contiene le informazioni relative agli enti e alle aziende che usano IC. Ogni fattura o documento "appartiene" ad una organizzazione.

Inoltre, per quanto riguarda le relazioni che intercorrono fra le tabelle, si ha quanto segue:

* `INVOICE - OPERATIONS` (*One to One*)
* `INVOICE - ORGANIZATION` (*One to Many*)
* `DOCUMENT - OPERATIONS` (*One to Many*)
* `DOCUMENT - ORGANIZATION` (*One to Many*)
* `DOCUMENT - REPOSITORY` (*One to Many*)

Dallo studio svolto sui metodi di costruzione di database documentali, si evince che l'idea generale è di accorpare, laddove possibile, più tabelle in un unico documento. Questo vale per tabelle in relazione *One to Many*, ma soprattutto *One to One*. Sarebbe quindi il caso di accorpare `INVOICE` e `OPERATIONS`, ovvero le fatture assieme con le operazioni ad esse associate (sostanzialmente un messaggio sullo stato della fattura).

In base alle best practices proposte dal team di MongoDB, la soluzione più coerente per costruire il database scelto per il confronto prestazionale individua tre collection separate per `INVOICE`, `DOCUMENT` e `ORGANIZATION`, strutturate come segue.

A seguito dello studio condotto sui pattern esistenti per la costruzione di database di questo tipo, risulta chiaro come l'aggiunta di uno di essi sarebbe una forzatura. Alcuni sarebbero indubbiamente utili qualora questo tipo di migrazione dovesse essere effettuato per componenti più corpose di InvoiceChannel, ma per quanto riguarda la parte selezionata per questa tesi, essa risulta già adeguata senza l'utilizzo di pattern.

Schema del database NoSQL

Come già detto in fase di presentazione di MongoDB, questo tipo di database documentale è spesso denominato *schema-less*. Ciò significa che a differenza dei database relazionali, un documento all'interno di MongoDB non deve rispettare alcuno schema per strutturare i dati al proprio interno.
I dati sono salvati in formato `JSON`, sfruttando coppie di chiave valore che possono a loro volta contenere array e sotto-documenti.
Riportiamo quindi un esempio di documento per ognuna delle *collection* individuate

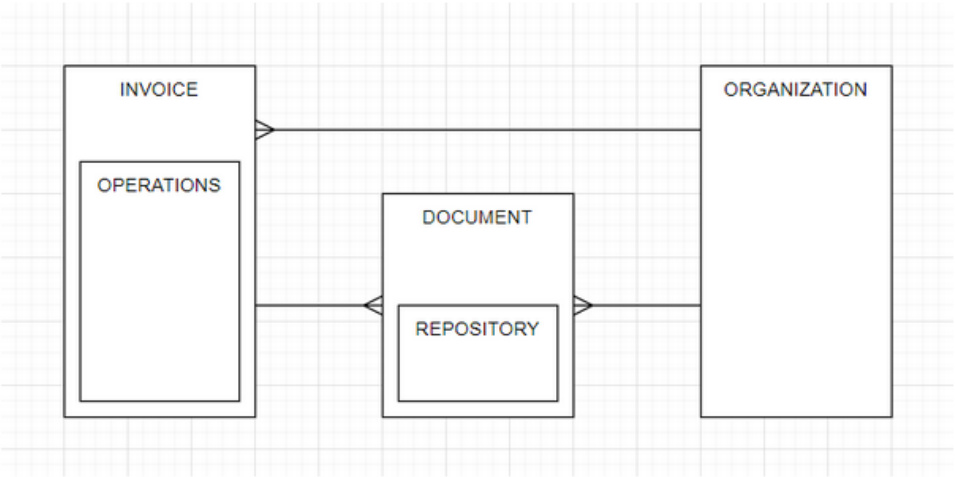


Figura 4.7: Diagramma per l'architettura del database documentale

precedentemente, per capire meglio quanto visto nel diagramma dell'architettura.

Esempio di documento all'interno della collection `INVOICE`:

```
{
    "ID": "1",
    "INVOICE_NUMBER": "2",
    "INVOICE_DATE": "1986-05-16T17:36:51.514Z",
    "TOTAL_AMOUNT": "32006.83",
    "RECIPIENT_NAME": "Floor Holdings Inc",
    "DATA_INVIO": "2012-10-10T18:18:33.919Z",
    "SOURCE_FILE_NAME": "edt",
    "TIPO_FATTURA": "CA",
    "DATA_RICEZIONE": "1994-05-09T10:37:37.325Z",
    "CAUSALE": "blvd yoga bg contractor",
    "MITTENTE": "Distance Software GmbH",
    "ID_ORGANIZATION": "72",
    "IS_B2B": "false",
    "TIPO_DOCUMENTO": "TD04",
    "OPERATIONS": {
      "CREATION_DATE": "1985-03-05T20:05:37.400Z",
      "MESSAGE": "Stato fattura = Primo_Stato"
    }
}
```

Come si può notare, il documento `OPERATIONS` è inserito nel documento della invoice come sottodocumento. Ciò significa che alla chiave `OPERATIONS` corrisponde un documento innestato, al posto di un semplice valore.

Esempio di documento all'interno della collection `DOCUMENT`:

```
{
    "ID": "1",
    "ID_INVOICE": "462",
    "CREATION_DATE": "1970-09-18 04:42:03.000",
    "COD_DOCUMENT": "1",
    "PATH_FILE": "2",
    "DOCUMENT_SIZE": "19375",
    "ID_ORGANIZATION": "1",
    "REPOSITORY":{
        "ID": 1,
        "DEFINITION": "/home/invoicechannel/ic3/repository/
           repository70",
        "COD_REPOSITORY_TYPE": "HDD"
    }
}
```

Anche in questo caso esiste per ogni documento di `DOCUMENT` un documento innestato che fa riferimento al *repository* di archiviazione.

Esempio di documento all'interno della *collection* `ORGANIZATION`.

```
{
    "ID": "1",
    "NAME": "Kuphal, Rath and O'Keefe",
    "ALIAS": "1",
    "PARTITA_IVA": "4",
    "COD_ORGANIZATION_TYPE": " Azienda",
    "TIPO_SOCIETA": " Altro",
    "CREATION_DATE": "1974-01-14 08:44:54.000"
}
```

I documenti di `ORGANIZATION` sono piuttosto semplici e contengono, come nella versione relazionale, le informazioni riguardanti enti e aziende che interagiscono con il sistema.

Capitolo 5

Sperimentazione

5.1 Popolamento dei database

Portata a termine la predisposizione di strumenti ed ambienti di lavoro, e determinata la struttura dei database, è ora possibile andare a caricare al loro interno i dati che li popolano.

5.1.1 Creazione dei dati

Per effettuare test di carico sui database è necessario che questi contengano una quantità di dati elevata. Si è scelto di generare dei dati "fantoccio" per automatizzare il processo.
Scegliendo lo strumento giusto, si possono così ottenere dati realistici in quanto a contenuto e dimensione, per fare in modo che le misurazioni effettuate sui tempi di esecuzione delle query abbiano un significato anche al di fuori dell'ambiente di test.
Sebbene inizialmente i dati siano stati generati tramite siti online che permettevano di esportare fino a un migliaio di tuple, è presto diventato chiaro come questo avrebbe rallentato di molto il processo di popolamento, rendendolo anche più complesso.

Si è quindi deciso di ricorrere ad uno strumento diverso, disponibile come pacchetto di *Node.js* e installabile tramite npm in modo da poter essere utilizzato da linea di comando.
Questo strumento, *datamaker*, è in grado di generare un numero arbitrario di records basandosi su template forniti dall'utente. Senza contare che è molto più rapido dei sistemi online.
Grazie a *datamaker* è stato possibile generare dei dati fittizzi secondo specifiche personalizzate, in quantità elevate.
Questo sistema garantisce anche un buon grado di consistenza, poiché rimane la traccia dello schema utilizzato in forma di template, cosa che invece andava spesso persa nei processi online. Anche grazie a questo è stato possibile generare dati affidabili su cui condurre i test, variando la quantità di dati presente in un file di import senza alterare in alcun modo lo schema che definisce come questi dati vengono costruiti.

5.1.2 Formato dei dati

Quando si parla di formato dei dati è bene specificare la differenza tra formato di importazione dei dati e formato dei dati all'interno dei database.
MongoDB salva i propri dati in formato `BSON`, una versione estesa del più comune `JSON`. Per la costruzione di tali dati è sufficiente creare dei file in formato `JSON`, e il database si occupa del resto.
Utilizzando l'interfaccia di Compass è tuttavia possibile importare i propri dati da un file in formato `CSV`. Questo può facilitare il processo, specialmente quando si è più abituati a lavorare con questo tipo di formato.
Per quanto riguarda PostgreSQL, anche l'interfaccia di pgAdmin permette di importare i propri dati da file, che devono essere in formato `CSV`.
Tale file verrà elaborato e il suo contenuto trasferito automaticamente nelle tabelle su cui si esegue questa operazione.

Nel caso di questo progetto, si è scelto di utilizzare file in formato `CSV` per l'importazione all'interno del database relazionale e file in formato `JSON` per l'importazione nel database NoSQL.
Nonostante il contenuto di documenti e tabelle sia pressochè lo stesso, proprio perchè il confronto abbia senso, la differenza sta proprio nella struttura in cui sono organizzati questi dati. Per questo motivo non sarebbe stato possibile usare lo stesso file `CSV` per popolare entrambi i database, nonostante gli strumenti utilizzati lo avrebbero permesso.

5.2 Metodi di monitoraggio dei risultati e creazione delle query

Per confrontare le performance delle query nei due database è necessario raccogliere dei dati sulla loro esecuzione. Nello specifico, serve sapere quanto tempo impiegano le query per essere eseguite.
Il tempo impiegato, tuttavia, può essere fuorviante. La notifica che per esempio riceviamo nel software pgAdmin si riferisce al tempo totale di elaborazione della richiesta, che per esempio comprende anche la latenza di connessione al network, e altri dati non relativi al tempo strettamente necessario all'esecuzione della query.

Per eseguire un confronto più preciso è necessario capire come estrapolare i dati che cerchiamo in entrambe le basi di dati.

5.2.1 Estrapolare i tempi di esecuzione in PostgreSQL usando pgAdmin

PostgreSQL, come molti database relazionali e non, mette a disposizione il metodo `EXPLAIN`. Quando questo viene utilizzato in testa ad una query, il risultato che si ottiene è una serie di informazioni riguardanti la sua esecuzione. Tra i vari parametri di questo metodo, quelli più interessanti per questo caso d'uso sono `ANALYZE` e `TIMING`, che possono essere specificati per ottenere informazioni specifiche riguardanti il tempo di esecuzione delle varie sezioni di query.
Le query che vengono eseguite in questo modo vengono comunque portate a termine dal database, anche se il risultato di eventuali operazioni di `SELECT` non viene mostrato

a schermo.
Per impedire che questo accada è bene seguire la query con un'operazione di *rollback*.

pgAdmin permette di utilizzare il metodo `EXPLAIN` eseguendo la query che si vuole analizzare in uno spazio apposito dell'interfaccia. Il risultato dell'operazione viene riportato in una tabella per semplificare la lettura, mentre vengono messi a disposizione anche una vista a grafo ed una tabella più complessa per il confronto tra tempi stimati dal metodo e tempi reali di esecuzione.
Agli scopi di questa tesi verranno prese in considerazioni le informazioni riportate nella tabella principale, dove i tempi di completamento delle operazioni sono all'occorrenza separati nei vari pezzi che compongono la query.

5.2.2 Estrapolare i tempi di esecuzione in MongoDB usando Compass

All'interno di MongoDB si utilizza un linguaggio specifico per estrapolare informazioni dai database. Se per i database relazionali si parla di Structured Query Language, in questo caso invece si usa *MQL*, ovvero MongoDB Query Language, basato sulla sintassi di *JavaScript*.

Attraverso un'ampia selezione di metodi questo linguaggio offre la possibilità di effettuare tutte le operazioni CRUD, oltre ad alcune altre funzionalità che possono tornare utili in varie situazioni, dalla gestione dei database alla misurazione delle prestazioni delle query.
Utilizzando Compass, è poi possibile sfruttare una finestra apposita dell'interfaccia grafica per eseguire l'*explain plan* di alcune query. Questa funzionalità è limitata tuttavia all'analisi del metodo `find()`, che corrisponde ad una select in SQL.
Per gli altri metodi è necessario approfondire il funzionamento del linguaggio di MongoDB per effettuare manualmente una ricerca sui tempi di esecuzione delle varie operazioni.
Una volta determinato quali metodi concatenare per ottenere i risultati desiderati, è poi necessario eseguire tali comandi nella shell di MongoDB. Fortunatamente questa è accessibile sempre dall'interfaccia di Compass, rendendo il processo più lineare.

5.3 Statistiche sulle query eseguite sui due database

Il confronto tra i due database è stato effettuato creando sette copie di query volte ad ottenere lo stesso risultato sia sul database relazionale che su quello documentale. Il risultato di questo tipo di analisi evidenzia come le diverse strutture di archiviazione dei dati e le diverse architetture all'interno dei database possono garantire risultati migliori o peggiori in base ai casi d'uso.
La creazione delle query è stata basata su una esemplificazione dei casi d'uso più comuni individuati per InvoicheChannel.

5.3.1 Query 1

La prima query è piuttosto semplice. Effettua una ricerca generica per elencare tutte le invoice e le operations ad esse collegate.

PostgreSQL	`SELECT *`
	`FROM invoice i JOIN operations o ON i.ID_OPERATION = o.id`
	`ORDER BY i.ID;`

MongoDB	`db.invoice.find().sort({ID:1})`

Figura 5.1: Codice della query numero 1, scritto in entrambi i linguaggi

Nota: PostgreSQL crea automaticamente gli indici sulle primary keys, quindi sono stati indicizzati anche i campi ID dei documenti in MongoDB. Questo è stato fatto per effettuare un confronto alla pari, ma è anche dovuto al fatto che intorno ai 200.000 documenti MongoDB non permette più di effettuare operazioni di `sort()` se non viene integrato l'uso di indici.

Nella seguente tabella possiamo vedere i risultati del confronto sui tempi ottenuti ripetendo la stessa query su database popolati con un numero sempre maggiore di elementi.

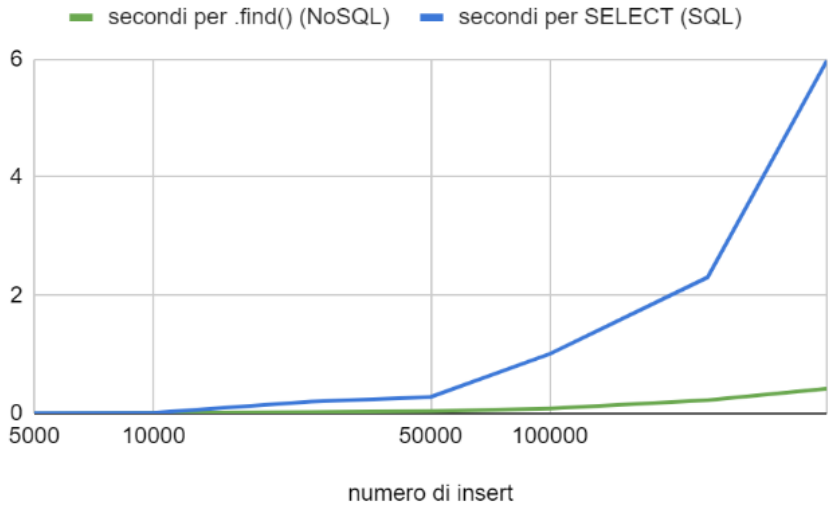


Figura 5.2: Risultato del confronto, query numero 1

Dal grafico si evince quanto già scoperto durante lo studio delle tecnologie coinvolte nel confronto. L'utilizzo di documenti incorporati (in questo caso i dati dell'operation che riguarda l'invoice sono contenuti nel documento dell'invoice stessa) permette a MongoDB di evitare le operazioni di `join`, che rallentano di molto l'esecuzione delle query in PostgreSQL.

Possiamo notare come la differenza sia ancora più evidente se cerchiamo gli stessi dati applicando una condizione di ricerca.

5.3.2 Query 2

Cerchiamo tutte le invoice e le operations associate, questa volta limitandoci alle sole fatture che soddisfano una specifica condizione sul proprio numero identificativo.

PostgreSQL	`SELECT *`
	`FROM invoice i JOIN operations o ON i.ID_OPERATION = o.id`
	`WHERE i.ID > 250000;`

MongoDB	`db.invoice.find({ID:{$gt: 250000}})`

Figura 5.3: Codice della query numero 2, scritto in entrambi i linguaggi

In questo caso PostgreSQL farà uso di indici per andare a scorrere invoice e verificare le condizioni che validano i dati. Introduciamo quindi l'indice su `ID` anche per il database di MongoDB.

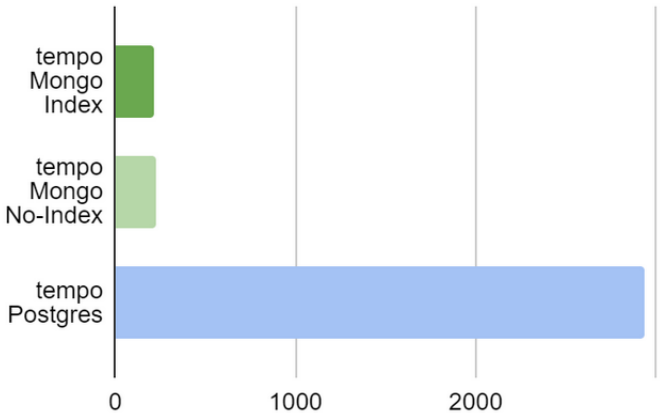


Figura 5.4: Risultato del confronto, query numero 2

Questa query è stata testata su database contenenti 500.000 elementi, chiedendo loro di restituirne soltanto la metà.
Si può vedere come, a prescindere dalla presenza dell'indice, MongoDB sia comunque più veloce di un ordine di grandezza.
Inoltre, grazie agli strumenti messi a disposizione da PostgreSQL, possiamo anche vedere come è stato speso il tempo all'interno di questa query:
Quasi il 70% del tempo è dedicato all'operazione di `join`, di cui invece MongoDB non si deve preoccupare (in questo caso).
La sola operazione di Index Scan occupa PostgreSQL per circa 90 ms, un tempo che sarebbe estremamente competitivo.

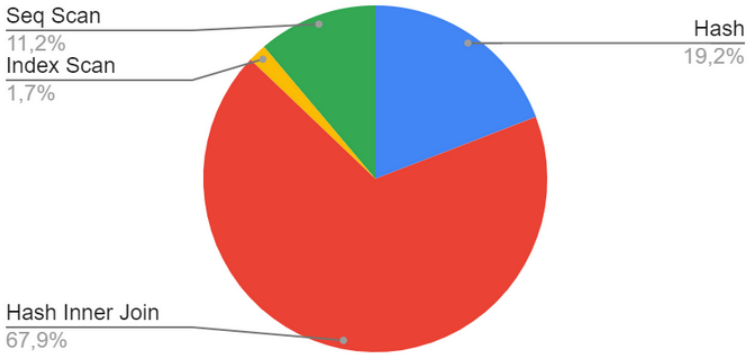


Figura 5.5: Utilizzo del tempo di esecuzione, query numero 2

5.3.3 Query 3

A confermare la tesi portata con la seconda query, effettuare una ricerca su un campo non indicizzato, in cui non è richiesto alcuna operazione di `join`, produce risultati più o meno simili per entrambi i sistemi: 240ms per MongoDB contro i 300ms per Postgres.
Anche in questo caso i test sono stati effettuati su database contenenti 500.000 elementi. Non sono riportati grafici a causa della minima differenza di prestazioni.

PostgreSQL	`SELECT * FROM invoice i WHERE i.mittente = 'Leeds SIA';`
MongoDB	`db.invoice.find({MITTENTE:"Leeds SIA"})`

Figura 5.6: Codice della query numero 3, scritto in entrambi i linguaggi

5.3.4 Query 4

Questa query cerca di evidenziare un'altra differenza importante tra i due database, dovuta al diverso metodo di archiviazione dei dati (documenti `JSON` o tabelle).

PostgreSQL	`SELECT * FROM invoice i WHERE i.total_amount > 50000 ORDER BY i.total_amount DESC LIMIT 100;`
MongoDB	`db.invoice.find({TOTAL_AMOUNT:{$gt:"55000"}}).sort({TOTAL_AMOUNT:-1}).limit(100)`

Figura 5.7: Codice della query numero 4, scritto in entrambi i linguaggi

In questo caso vengono testati entrambi i database con 100.000 elementi inseriti.
Si vogliono trovare i documenti con data di invio più recente. Dover accedere al contenuto dei campi sembrerebbe essere un lavoro più dispendioso quando si tratta di un campo di un documento, piuttosto che un campo di una tabella. Si nota infatti che il risultato ottenuto conferma questa tesi.

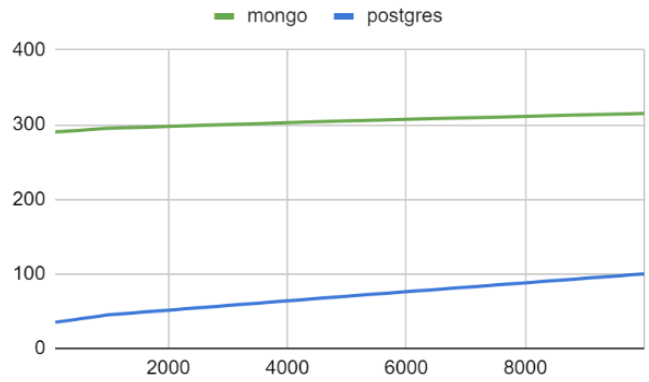


Figura 5.8: Risultato del confronto, query numero 4

5.3.5 Query 5

Viene testata ora una query più complessa, utilizzata per effettuare l'update di determinati campi all'interno del database. Nello specifico, si vogliono simulare l'update della causale della fattura, con delle condizioni che comprendono dati dell'operation ad essa collegata.

Per eseguire questa query in mongo la concatenazione dei metodi è più complessa, come si può notare dall'immagine che riporta il codice.
Purtroppo Compass mette a disposizione l'interfaccia explain solo per il metodo `find()`. Come detto in precedenza dobbiamo quindi passare tramite *mongosh*, il terminale di MongoDB su cui si possono eseguire tutti i comandi nella sua sintassi specifica.
Inoltre, il sistema utilizzato precedentemente (ovvero concatenare il metodo `explain()` a fine query) non è disponibile per l'operazione di update, e bisogna quindi usare una sintassi più completa che prevede l'uso del metodo `runCommand()`, visibile in tabella.

Nuovamente, il grafico non viene riportato poiché lineare. L'unico dato utile che traspare è la costante differenza nei tempi di esecuzione, che anche in questo caso favoriscono MongoDB, aggirandosi intorno ai 105 ms. Per postgres abbiamo tempi all'incirca doppi.

5.3.6 Query 6

Questa query è simile alla precedente. In questo caso tuttavia l'aggiornamento viene fatto sulla tabella `OPERATIONS`, e la condizione risiede in tabella `INVOICE`.

PostgreSQL

```
UPDATE invoice AS i SET causale = 'causale1'
WHERE i.causale = 'causale2' AND
(
SELECT o.creation_date FROM operations AS o WHERE
i.id_operation = o.id
) > '2004-01-01 00:00:00';
```

MongoDB

```
db.runCommand(
      {
      explain: {
            update: "invoice_100000_2",
            updates: [ {
                  q: {$and :[{"OPERATIONS.CREATION_DATE": {$gt:
                  "2004-01-01 00:00:00"}},{"CAUSALE": "causale1"}]},
                  u: {$set: {"CAUSALE": "causale"}},
                  multi: true
            } ]
      },
      verbosity: "executionStats"
   }
)
```

Figura 5.9: Codice della query numero 5, scritto in entrambi i linguaggi

PostgreSQL

```
UPDATE operations AS o SET message = 'Stato fattura =
Terzo_Stato'
WHERE o.message = 'Stato fattura = Secondo_Stato' AND (
      SELECT i.data_invio FROM invoice AS i WHERE o.id =
i.ID_OPERATION
) > '2004-01-01 00:00:00';
```

MongoDB

```
db.runCommand(
   {
     explain: {
            update: "invoice_100000_2",
            updates: [ {
                  q: {$and :[{"DATA_INVIO": {$gt: "2004-01-01
                  00:00:00"}},{"OPERATIONS.MESSAGE": "Stato fattura =
                  Secondo_Stato"}]},
                  u: {$set: {"OPERATIONS.MESSAGE": "Stato fattura
                  = Terzo_Stato"}},
                  multi: true
            } ]
      },
      verbosity: "executionStats"
   }
)
```

Figura 5.10: Codice della query numero 6, scritto in entrambi i linguaggi

La query su PostgreSQL impiega dai 2 ai 5 minuti per essere portata a termine su un database contenente 100.000 elementi.
Possiamo notare come quasi tutto questo tempo sia utilizzato per effettuare uno scan della tabella `OPERATIONS`, che viene poi aggiornata.

Node type	Count	Time spent
Seq Scan	2	141140.886 ms
Update	1	322.887 ms

Figura 5.11: Suddivisione dei tempi di esecuzione per la query numero 6 in PostgreSQL

MongoDB impiega soltanto 65 millisecondi per effettuare tutte le operazioni.
Ripetendo la query su database più ampi il risultato non cambia. Per 250.000 elementi PostgreSQL impiega più di trenta minuti, MongoDB si mantiene sotto i 200 millisecondi. Il tempo raddoppia quando all'interno del database sono contenuti 500.000 elementi. Per questa quantità di dati, PostgreSQL non restituisce un risultato in tempi utili.

5.3.7 Query 7

Quest'ultima query rappresenta un altro caso d'uso importante.

PostgreSQL

```
SELECT org.name, count(*) AS number_of_invoices
FROM organization org JOIN invoice i ON i.id_organization
= org.id
GROUP BY org.name;
```

MongoDB

```
db.org.aggregate([
      { $lookup:
            {
                  from: "invoice_100000_2",
                  localField: "ID",
                  foreignField: "ID_ORGANIZATION",
                  let: { id_org : "$ID_ORGANIZATION"},
                  pipeline: [ {
                              $group:
                              {
                                    _id: "$$id_org",
                                    countINVOICES: {$count:{}}
                              }
                  } ],
                  as: "INVOICES"
            }
      },
      { $project: {"NAME":1, "INVOICES":1}}
]).explain("executionStats")
```

Figura 5.12: Codice della query numero 7, scritto in entrambi i linguaggi

Come già menzionato, MongoDB è sì un database NoSQL, ma tecnicamente è considerabile anche "relazionale", perchè consente di effettuare operazioni di *lookup* tra *collection* diverse per unire informazioni altrimenti separate. Questa, tuttavia, non è sicuramente la feature di punta di un database documentale, e si traduce quindi in

un'operazione piuttosto lenta rispetto ad altre operazioni per cui un un database di questo tipo è stato espressamente studiato.

Tutto questo è dimostrato dai dati statistici raccolti durante l'esecuzione delle query. Per PostgreSQL si tratta di una join piuttosto semplice, accoppiata ad un raggruppamento. Il tempo di esecuzione si aggira intorno ai 65 ms se le tabelle `INVOICE` e `ORGANIZATION` contengono 100.000 elementi ciascuna.
MongoDB impiega tra i 5 e i 6 secondi per la stessa operazione, con la stessa quantità di dati.

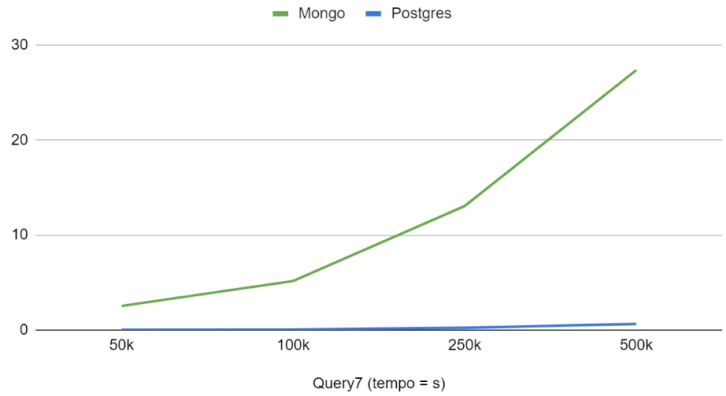


Figura 5.13: Risultato del confronto, query numero 7

Con questa query si conclude la fase di confronto diretto tra i tempi di esecuzione delle operazioni sui due database.Vale la pena di evidenziare un fattore che sicuramente salta all'occhio leggendo le precedenti pagine: la complessità del linguaggio utilizzato per le query di MongoDB non è trascurabile e rispetto all'immediatezza di SQL risulta piuttosto impegnativo da imparare, utilizzare e comprendere.

5.4 Inserimento della fattura integrale

Uno dei casi d'uso di interesse per l'analisi condotta su InvoiceChannel è quello che riguarda il salvataggio delle fatture.
Sebbene parte dei dati che le compongono sia salvata nella tabella invoice, per velocizzare l'accesso alle informazioni più importanti la fattura viene anche salvata per intero in formato `XML`.

Questo ovviamente non può essere fatto all'interno del database, principalmente perchè salvare dei dati così grandi all'interno di un unico campo non è desiderabile.
All'estremo opposto c'è la possibilità di elaborare il contenuto del file per scomporlo in campi salvabili e indicizzabili, potendo così ricomporre la fattura quando necessario. Anche questa soluzione ha dei problemi, perchè effettuare operazioni di questo tipo su ogni fattura introdotta nel sistema creerebbe probabilmente un collo di bottiglia, e anche ricomporla quando necessario potrebbe rivelarsi costoso o non pratico.

La soluzione, come spesso succede, sta nel mezzo, e se ne trova un esempio all'interno dell'architettura di MongoDB.

Come è stato visto, MongoDB salva i propri documenti in formato `JSON`, introducendo degli "effetti collaterali" che tornano utili al nostro caso d'uso.
L'idea è di convertire le fatture da `XML` a `JSON` per poi salvarle integralmente all'interno di una collection apposita.
In questo modo la fattura sarebbe sempre a disposizione per intero all'interno del database, come se fosse stato inserito il file `XML` in un campo di tabella. Allo stesso tempo sarebbe facile effettuare ricerche sui "campi" che compongono la fattura, per natura stessa dei file `JSON`, senza dover fare alcuna operazione di scomposizione e ricostruzione del file, se non per l'iniziale traduzione da un formato all'altro.

Tale trasformazione è bidirezionale, quindi così come si è passati dall'`XML` al `JSON` si può fare il contrario senza alcuna perdita di dati. Questo è importante perchè a livello legale `XML` rimane il formato ufficiale per le fatture elettroniche, quindi in caso di necessità si può ricostruire l'originale.

Tutto questo ovviamente ha un costo, e sta a chi crea un sistema di questo tipo il compito di soppesare pro e contro. MongoDB supporta l'inserimento di file `BSON` fino a 16 MB e una fattura tradotta in formato `JSON` occupa circa 5 KB, quindi sebbene appaia come un file "grande" all'occhio umano, si tratta di una dimensione bel al di sotto dei limiti strutturali del database. Le prestazioni non sarebbero intaccate da questo tipo di soluzione, anche quando si considera l'archiviazione di centinaia di migliaia di fatture, specialmente se si fa uso di indici.
Anche il costo della traduzione da `XML` a `JSON` va sicuramente rendicontato, ma dai test condotti non sembra avere un grosso impatto. Utilizzando uno strumento esterno basato su npm (come *datamaker*) è possibile automatizzare questo processo ad un bassissimo costo computazionale.

Capitolo 6

Conclusioni

6.1 Conclusioni riguardanti il confronto in generale

Prima di avviare questo progetto di tesi è stata effettuata una breve ricerca per comprendere se l'argomento fosse stato già esplorato in precedenza in altri contesti.
Sono state individuate altre tesi dai temi simili che propongono il confronto all'interno di database "da laboratorio", e alcune ricerche condotte su larga scala da aziende.
Vista la natura di questo progetto, legato ai prodotti software sviluppati da Ifin Sistemi, la prospettiva di effettuare una ricerca approfondita risultava comunque interessante.

Le conclusioni raggiunte al termine dei due mesi di tirocinio non sono dissimili da quelle raggiunte da altre ricerche, quando queste erano condotte senza lo scopo di pubblicizzare l'una o l'altra tecnologia.
Come in molti altri ambiti, anche in questo caso il contesto di utilizzo di uno strumento ne determina efficacia ed efficienza.
L'introduzione dei database NoSQL ha permesso di raggiungere risultati estremamente vantaggiosi in molti ambiti applicativi, ma questo non li rende uno strumento universale per potenziare qualsiasi database in ogni contesto.
Tale fatto è valido per qualsiasi tecnologia, ma lo è ancora di più quando si parla di database NoSQL, data la grande varietà di soluzioni che questo gruppo comprende.
Nel caso specifico di MongoDB è facile essere tentati di farne largo uso in molti contesti diversi, grazie alla sua versatilità, semplicità e alla grande disponibilità di documentazione e risorse, data la sua natura *open source*.
Si è visto tuttavia come in determinati casi questa potrebbe non essere la scelta migliore.
Allo stesso tempo, fare affidamento solo alle tecnologie già consolidate, con alle spalle anni di sviluppo e di esperienza, può ancorare il proprio software a delle soluzioni vecchie o inadeguate.

Anche in questo caso la soluzione migliore sta nel mezzo. L'utilizzo di database ibridi che permettono di sfruttare il meglio di entrambe le tecnologie può garantire migliori prestazioni senza sacrificare solidità delle strutture e affidabilità del servizio.

6.2 Conclusioni specifiche per il contesto di Ifin Sistemi

Per quanto riguarda l'azienda per cui è stato condotto lo studio e il software su cui è stato basato, i risultati sono evidenti. L'implementazione di una soluzione ibrida potrebbe giovare di molto al sistema, alleggerendo il carico di lavoro per i database tradizionali e permettendo di implementare nuove funzionalità.

Non è tuttavia prevista, nell'immediato futuro, un'analisi più approfondita volta a provare quanto postulato con questa tesi, per rendere più realistica l'idea di una migrazione di database. Come già evidenziato dal dialogo avuto con i team dedicati allo sviluppo dei software analizzati, le soluzioni adottate fin'ora per potenziare i database in uso sono state al contempo dispendiose da implementare (a livello di tempo ed energia) e proficue nel modo in cui sono riuscite a tamponare i problemi di disponibilità a cui tali tecnologie sono andate incontro con l'aumentare del carico di lavoro imposto. Finchè i sistemi adottati continuano a funzionare stabilmente, non ha senso per l'azienda dedicare forza lavoro allo studio di nuove tecnologie, soprattutto in luce di un secondo fattore estremamente centrale: i prodotti dell'azienda non sono nuovi. Questo ha un peso per quel che riguarda la necessità di far "collaborare" tecnologie nuove con altre meno recenti, ma significa soprattutto che negli anni in cui LegalArchive e InvoiceChannel (i prodotti in questione) sono stati in commercio, il naturale ciclo di manutenzione e upgrade del software ha avuto corso, rendendoli dei progetti estremamente grandi e complessi, la cui migrazione richiederebbe probabilmente più energie di quelle che si andrebbero a risparmiare una volta terminata.

Ciò che questa ricerca può aspirare ad essere è uno spunto per evidenziare quanto l'implementazione ibrida di soluzioni relazionali miste a quelle NoSQL possa essere proficua. La soluzione che viene proposta nella sezione 5.4 è un esempio valido di questa convivenza di tecnologie, e rappresenta un buon punto di partenza qualora l'azienda decidesse di percorrere questa strada.

6.3 Conoscenze acquisite e valutazione personale

Il progetto di tirocinio mi aveva inizialmente attirato per la prospettiva di riprendere in mano le consocenze acquisite nell'ambito delle basi di dati ed approfondirle entrando a contatto con tecnologie più nuove, in grado di mettere in discussione i paradigmi su cui ci si basa quando si approccia per la prima volta questo contesto.

Le mie aspettative sono state soddisfatte ampiamente, sia per quanto riguarda la possibilità di eseguire degli studi su quello che è lo stato dell'arte ad oggi nell'ambito NoSQL, ma anche per le possibilità che ho avuto di sviluppare qualcosa di più concreto e potenzialmente utile, sia per me che per l'azienda.

È stato inoltre estremamente interessante ed importante per me avere la possibilità di approcciarmi all'ambiente lavorativo, in cui ho potuto imparare a muovermi seguendo ritmi e direttive diverse da quelle a cui ci si abitua all'interno dell'Università.

La concomitanza di tutte queste condizioni ha reso questo stage un'esperienza proficua e nel complesso positiva.

Glossario

change stream Con riferimento a MongoDB, i *change stream* consentono alle applicazioni di accedere alla modifica dei dati in tempo reale senza la complessità e il rischio di seguire l'*operation log*. Le applicazioni possono utilizzare i *change stream* per sottoscrivere tutte le modifiche ai dati su una singola raccolta, un database o un'intera distribuzione e reagire immediatamente.. 24, 25

DAO I *Data Access Object* sono un pattern utilizzato per separare il servizio business di alto livello dalle API di basso livello che hanno accesso ai dati.. 11

database distribuito Un database distribuito è un database che si trova sotto il controllo di un database management system (DBMS) nel quale gli archivi di dati non sono memorizzati sullo stesso computer bensì su più elaboratori o nodi.. 6, 8

DBaaS Un DBaaS (noto anche come servizio di database gestito) è un servizio di cloud computing che consente agli utenti di accedere ad un sistema di database cloud senza acquistare e configurare hardware proprio, installare software di database o gestire il database personalmente.. 10

design pattern Un design pattern è una soluzione progettuale generale ad un problema ricorrente. Si tratta di una descrizione o modello logico da applicare per la risoluzione di un problema che può presentarsi in diverse situazioni durante le fasi di progettazione e sviluppo del software.. 23

DTO Un *Data Transfer Object* è un oggetto che trasporta dei dati tra i processi. Si può utilizzare questo pattern per facilitare la comunicazione tra due sistemi senza esporre informazioni potenzialmente riservate.. 11

EclipseLink EclipseLink è un progetto open source di Eclipse Foundation. Il software fornisce un framework estensibile che consente agli sviluppatori Java di interagire con vari servizi di dati, inclusi database, servizi web e sistemi informativi aziendali. EclipseLink supporta una serie di standard di persistenza, tra cui Jakarta Persistenza (JPA).. 1, 11

Hibernate Hibernate è una piattaforma middleware open source per lo sviluppo di applicazioni Java, attraverso l'appoggio al relativo framework, che fornisce un servizio di Object-relational mapping (ORM) ovvero gestisce la persistenza dei dati sul database attraverso la rappresentazione e il mantenimento su database relazionale di un sistema di oggetti Java.. 1

IoT L'*Internet of Things* (IoT), è un neologismo utilizzato nel mondo delle telecomunicazioni e dell'informatica che fa riferimento all'estensione di internet al mondo degli oggetti e dei luoghi concreti, che acquisiscono una propria identità digitale in modo da poter comunicare con altri oggetti nella rete e poter fornire servizi agli utenti.. 26

JDBC *Java DataBase Connectivity* è un connettore e un driver per database che consente l'accesso e la gestione della persistenza dei dati sulle basi di dati da qualsiasi programma scritto con il linguaggio di programmazione *Java*, indipendentemente dal tipo di DBMS utilizzato.. 1, 11, 12

JPA Le Java Persistence API sono un framework per il linguaggio di programmazione *Java* che si occupa della gestione della persistenza dei dati di un DBMS relazionale nelle applicazioni che usano le piattaforme *Java*.. 1, 11

KPI L'indicatore chiave di prestazione (o *key performance indicator*) è una metrica che indica il livello di raggiungimento di un dato obiettivo da parte di un individuo, di un reparto o di un'azienda.. 4

master-slave Il paradigma "master-slave" è un modello di comunicazione o controllo asimmetrico in cui un dispositivo o processo (il "master") controlla uno o più altri dispositivi o processi (gli "slave") e funge da hub di comunicazione.. 6, 7

Model-View-ViewModel Il Model-view-viewmodel (MVVM) è un modello architetturale che facilita la separazione dello sviluppo dell'interfaccia utente (GUI, la vista), dallo sviluppo della *business logic* o logica di back-end (il modello) in modo tale che la vista non dipenda da alcuna piattaforma specifica del modello.. 11

npm npm (originariamente abbreviazione di *Node Package Manager*) è un gestore di pacchetti per il linguaggio di programmazione *JavaScript*. npm è il gestore di pacchetti predefinito per l'ambiente di runtime *JavaScript Node.js*.. 33, 43

operazioni CRUD *Create*, *read*, *update*, e *delete* (CRUD) sono le quattro operazioni basilari della gestione persistente dei dati.. 17, 35

Proof of Concept Realizzazione incompleta o abbozzata di un determinato progetto o metodo, allo scopo di provarne la fattibilità o dimostrare la fondatezza di alcuni principi o concetti costituenti.. 18

rest API Un'API, o *application programming interface*, è un insieme di regole che definiscono il modo in cui le applicazioni o i dispositivi possono connettersi e comunicare tra loro. Un'API REST è un'API conforme ai principi di progettazione REST, o *representational state transfer architectural style*. Per questo motivo, le API REST sono talvolta chiamate *RESTful APIs*.. 10

SQL SQL è un linguaggio standardizzato per database basati sul modello relazionale (RDBMS), progettato per eseguire operazioni di creazione e modifica di schemi di database, inserimento, modifica e gestione di dati memorizzati, creazione di strumenti di controllo ed accesso ai dati.. 5, 6, 8, 35, 42

test di carico Un test di carico è mirato a misurare la propensione di una funzionalità a sopportare un numero definito e via via crescente di dati semplici o aggregati che dovranno essere visualizzati oppure elaborati.. 3

Tomcat Apache Tomcat è un server web (nella forma di contenitore servlet) *open source* sviluppato dalla Apache Software Foundation.. 1, 11, 12

Bibliografia

Siti web consultati

Corsi udemy. URL: https://www.udemy.com/.

Corso di Data Modeling, MongoDB University. URL: https://university.mongodb.com/courses/M320/about.

Documentazione datamaker. URL: https://www.npmjs.com/package/datamaker/v/0.1.5?activeTab=readme.

Documentazione Docker. URL: https://docs.docker.com/.

Documentazione MongoDB. URL: https://www.mongodb.com/docs/.

Documentazione MongoDB Compass. URL: https://www.mongodb.com/docs/compass/current/.

Documentazione pgAdmin. URL: https://www.pgadmin.org/docs/.

Documentazione PostgreSQL. URL: https://www.postgresql.org/docs/.

Documentazione xml-js. URL: https://www.npmjs.com/package/xml-js.