# 

# **ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA CAMPUS DI CESENA**

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

**TITOLO DELL’ELABORATO**

*Elaborato in*

BASI DI DATI AVANZATE

*Relatore* *Presentata da*

**Prof. Alessandra Lumini** **Francesco Foschini**

Anno Accademico 2018 - 2020

# Sommario

Il lavoro prevede il design e la realizzazione di un sistema che permetta di distribuire notifiche relative a eventi generati su software di beckend e di frontend per poter fornire informazioni costantemente aggiornate agli utenti delle applicazioni web aziendali e monitorare lo stato di avanzamento di task/procedure.

Per il design del sistema è stato necessario acquisire una solida conoscenza dei sistemi distribuiti, dei problemi di concorrenza sui dati condivisi e dell’interfacciamento con database realtime.

Il progetto ha previsto due attività principali ovvero l’implementazione del server e della libreria di sistema UtilityRethink.

In seguito ad una attenta ricerca iniziale è stato deciso di affidare il ruolo del server di sistema al dbms RethinkDB.

La libreria, invece, è stata scritta in linguaggio C# su piattaforma .NET Core allo scopo di garantire ai client l’interfacciamento al server di sistema.

Il codice prodotto è stato organizzato in maniera efficiente e riutilizzabile nei prodotti della società.

Dal punto di vista del database, è stato studiato lo schema migliore di rappresentazione delle notifiche allo scopo di ottenere le migliori prestazioni possibili a fronte delle necessarie caratteristiche di consistenza ed isolamento.

L’attività è stata svolta interamente presso l’azienda Energy Software a Faenza.

EnergySoftware s.r.l è un’azienda che nasce nel 2015 da un team di persone con esperienza decennale con l’obiettivo di fornire tutta una serie di servizi legati al mondo dell’informatica tra cui sviluppo di software per i mercati energetici, sviluppo di applicativi per specifiche esigenze del cliente, consulenza sistematica specializzata su server e networking.

I risultati ottenuti sono stati oggetto di misurazione e presentazione.

Il seguito della tesi è articolato in 5 capitoli:

* Capitolo 1: introduce nei dettagli il contesto di collocazione di questo progetto di tesi. Viene inizialmente trattato il design pattern Publish/Subscribe e il concetto di database-realtime.

Infine vengono definite le specifiche funzionali e architetturali del sistema.

* Capitolo 2: presenta altre tecniche e tecnologie utilizzabili per l’implementazione del sistema.

Viene definito il concetto di database auditing tipico dei sistemi dbms relazionali e, inoltre, saranno elencati altri “dbms realtime” utilizzabili come server di sistema.

Viene infine trattato il sistema di messaggistica RabbitMQ.

* Capitolo 3: descrive il design della libreria UtilityRethink implementata.

Vengono forniti i dettagli implementativi delle principali funzionalità fornite dall’applicazione.

Inoltre, sarà spiegato lo schema di rappresentazione scelto delle notifiche sia sul database che per la libreria.

* Capitolo 4: vengono spiegate le scelte architetturali effettuate per l’implementazione del server e della libreria di sistema UtilityRethink.

RethinkDB e Docker per l’implementazione del server di sistema.

BChavez/RethinkDb.driver, Reactive Extension(Rx) e Simple Injector per l’implementazione di Utility Rethink.

* Capitolo 5: descrive tutti i passaggi chiave necessari all’implementazione del server di sistema.

Elenca inizialmente le diverse possibilità di gestione delle politiche di Sharding e Replication sul dbms “a più nodi” RethinkDB.

Descrive inseguito i comandi di docker-compose da eseguire per poter gestire i tre tipi di server RethinkDB implementati (uno, due e cinque nodi).

Sarà infine presentato il risultato di un test eseguito per verificare le performance in scrittura e lettura del dbms RethinkDB.

**Indice**

[Sommario 2](#_Toc58490311)

[Specifiche e descrizione del problema 6](#_Toc58490312)

[Introduzione al problema 6](#_Toc58490313)

[Specifiche architetturali 7](#_Toc58490314)

[Specifiche funzionali 7](#_Toc58490315)

[Altri sistemi esistenti utilizzabili in futuro (Librerie e tecnologie) 8](#_Toc58490316)

[Cenni sulle tecniche di auditing 9](#_Toc58490317)

[RabbitMQ cenni 11](#_Toc58490318)

[Conclusioni sulle altre tecnologie utilizzabili 12](#_Toc58490319)

[Design libreria UtilityRethink 13](#_Toc58490320)

[Aspetti statici 13](#_Toc58490321)

[Base Dati 13](#_Toc58490322)

[Notifiche 13](#_Toc58490323)

[Notifiche di nuovo dato 15](#_Toc58490324)

[Notifiche di esecuzione 15](#_Toc58490325)

[Design UML applicativo 16](#_Toc58490326)

[Design della Struttura generale UML libreria UtilityRethink 16](#_Toc58490327)

[UtilityRethink 17](#_Toc58490328)

[DbManager 18](#_Toc58490329)

[NotificationsManager 19](#_Toc58490330)

[IQueryNotification 20](#_Toc58490331)

[IRXNotifier 21](#_Toc58490332)

[Esempio di OnNext, OnError e OnComplete 23](#_Toc58490333)

[Connection 24](#_Toc58490334)

[Aspetti dinamici 26](#_Toc58490335)

[Caso d’uso aziendale 26](#_Toc58490336)

[Simple Injector 27](#_Toc58490337)

[Diagrammi di attività 28](#_Toc58490338)

[Scelte architetturali per il caso d’uso aziendale 29](#_Toc58490339)

[Software utilizzato per il server 29](#_Toc58490340)

[RethinkDb 29](#_Toc58490341)

[Docker 32](#_Toc58490342)

[Software utilizzati dalla libreria di sistema implementata 33](#_Toc58490343)

[Metodologia di Progettazione 34](#_Toc58490344)

[Implementazione Server 35](#_Toc58490345)

[Sharding e Replication 36](#_Toc58490346)

[(questa parte è stata letta dalle slide di BDA , cosa mettere in biblio?) 36](#_Toc58490347)

[Gestione con Interfaccia Web 37](#_Toc58490348)

[Gestione con ReQL 38](#_Toc58490349)

[Gestione nella libreria 38](#_Toc58490350)

[Comandi per la gestione del server implementato 39](#_Toc58490351)

[Test e Performance 40](#_Toc58490352)

[Risultati Test Performance 41](#_Toc58490353)

[Server Rethink a singolo nodo 41](#_Toc58490354)

[Server MySQL a singolo nodo 41](#_Toc58490355)

[Server Rethink a cinque nodi 42](#_Toc58490356)

[Una shard con cinque repliche ciascuna 42](#_Toc58490357)

[Cinque shard con una sola repliche 42](#_Toc58490358)

[Due shard con tre repliche ciascuna 42](#_Toc58490359)

[Sviluppi futuri 43](#_Toc58490360)

[Conclusioni 44](#_Toc58490361)

[Bibliografia 46](#_Toc58490362)

**Capitolo 1**

# Specifiche e descrizione del problema

## Introduzione al problema

### Publish/Subscribe

L’implementazione del sistema richiedeva un unico importante e particolare requisito di partenza.

Il server di sistema stesso deve essere in grado di notificare eventi in real-time ai servizi client interessati .

Il sistema realizzato è quindi, per questa ragione, un caso implementativo del design pattern Publish/Subscribe [**[1]**](#Publish_Subscribe).

In questo schema, infatti, il mittente dei messaggi è il server mentre i destinatari sono i servizi client.

Nello schema del design publish-subscribe, [mittenti](https://it.wikipedia.org/wiki/Mittente) e [destinatari](https://it.wikipedia.org/wiki/Destinatario) di messaggi dialogano attraverso un tramite, chiamato *dispatcher* o *broker*.

Il server (*publisher*) non essendo consapevole dell'identità dei destinatari (detti *subscriber*) si limita a "pubblicare" i propri messaggi al dispatcher.

I client si rivolgono a loro volta al dispatcher "abbonandosi" alla ricezione di messaggi.

Il dispatcher quindi inoltra ogni messaggio inviato dal server solo ai servizi client interessati a quel messaggio.

Questo schema di Publish/Subscribe, implica che per qualsiasi tecnologia del server scelta non sia noto quanti e quali siano i servizi client che utilizzano la libreria contribuendo di conseguenza alla scalabilità del sistema.

Utilizzando questo schema di messaggistica i servizi client del sistema riescono quindi a registrarsi alla ricezione di notifiche utilizzando una funzionalità della libreria UtilityRethink implementata (vedi capitolo IRXNotifier).

Il meccanismo di sottoscrizione consente, inoltre, ai servizi client di precisare a quali messaggi sono interessati.

Ad esempio, un client potrebbe "abbonarsi" solo alla ricezione di notifiche aventi un determinato valore di un campo.

### Database-Realtime

### Negli ultimi anni, le persone si aspettano reattività dalle proprie applicazioni, ovvero si presume che le modifiche apportate da altri utenti si riflettano immediatamente nelle interfacce che stanno utilizzando.

### Tuttavia, la creazione di tali applicazioni con la tecnologia di database tradizionale è difficile, perché i i sistemi per l’archiviazione e il recupero dei dati sono stati sviluppati per decenni attorno a un modello di accesso richiesta-risposta puramente “pull-based”.

### Rispondendo alla necessità di reattività sul lato database, è emersa una nuova classe di sistemi di database “orientati nativamente al push delle modifiche”.

### Questi nuovi sistemi promettono quindi di facilitare lo sviluppo di applicazioni web reattive.

### Questi sistemi vengono spesso definiti database in tempo reale, poiché mantengono i dati sul client sincronizzati con lo stato corrente del database.

### La modifica dei dati viene intercettata e fatta notificare in tempo reale a tutti i servizi client.

## Specifiche architetturali

Il sistema coinvolge essenzialmente tre attori/entità principali ovvero un applicativo client, una libreria e un server.

Il servizio client riesce ad interagire con il server utilizzando la libreria.

I requisiti architetturali e tecnologici sono quindi:

• Architettura client - server;

• Accesso al server attraverso i protocolli http ed https;

• Dbms non relazionale real-time(opzionale nel caso si scelga di utilizzare RabbitMQ come tecnologia server).

## Specifiche funzionali

La libreria implementata consente ai servizi client che ne usufruiscono di registrarsi alla ricezione di messaggi a cui si è interessati dal server.

Nel contesto aziendale, in particolare, i servizi di frontend e beckend possono registrarsi alla ricezione di notifiche relative a eventi (tabella di sistema “Notifications”) con un certo valore di argomento (campo “arg”).

Se ad esempio viene inserita sulla tabella “Notifications” una nuova notifica con campo “arg” uguale ad “A” allora verranno avvisati della nuova notifica inserita tutti i servizi interessati a notifiche aventi “arg” uguale ad “A”.

In particolare, i servizi client riceveranno un oggetto che racchiude in se lo stato della notifica.

Se è stata scelta una tecnologia dbms per il server di sistema è possibile eseguire operazioni CRUD sulle tabelle che vengono considerate di sistema dalla libreria (es “Notifications”).

Inoltre, fornisce funzionalità gestionali per il database come la creazione e cancellazione di nuove tabelle e indici e la riconfigurazione delle replication e shards dei dati di una tabella.

**Capitolo 2**

# Altri sistemi esistenti utilizzabili in futuro (Librerie e tecnologie)

I sistemi utilizzabili, per i requisiti richiesti dall’applicativo, sono quindi tutte quelle tecnologie in grado di implementare il design pattern publish-subscibe.

Un’altra metodologia che si avvicina concettualmente al design pattern “Publish/Subscribe” sono le tecniche di auditing di database realizzate e tipiche dei dbms relazionali**.**

Il database auditing implica, infatti, l' [osservazione di](https://en.wikipedia.org/wiki/Audit) un [database in](https://en.wikipedia.org/wiki/Database) modo da essere consapevoli delle azioni svolte dagli [utenti](https://en.wikipedia.org/wiki/User_(computing)) del database [**[2]**](#Database_auditing).

[Gli amministratori](https://en.wikipedia.org/wiki/Database_administrators) e i consulenti di [database](https://en.wikipedia.org/wiki/Database_administrators) spesso impostano il controllo per motivi di sicurezza, ad esempio, per garantire a coloro che non hanno il permesso di accedere ad alcune informazioni sul db non vi accedano.

Attraverso le tecniche di database auditing sui dbms relazionali è quindi possibile monitorare e salvare le operazioni svolte su una tabella di un database [[3]](#Meccanismi_auditing).

Anche in questo caso quindi il database diviene in questo modo un entità “attiva”.

Una seconda tecnologia basata sul design “Publish/Subscribe” che distribuisce quindi i messaggi di un server ai vari client registrati è il protocollo RabbitMQ.

## Cenni sulle tecniche di auditing

Il concetto generale di database auditing riguarda il monitoraggio dell’utilizzo dei record di un database all’interno di un dbms relazionale [[4].](#SQLserver_audit)

Questa tecnica seppure si avvicini, non implementa completamente il design pattern Publish/Subscribe.

Quando si controlla un database, infatti, le operazione sui dati vengono monitorate e registrate in un registro di controllo interno del dbms, chiamato audit trail [[5]](#why_you_need_auditTrail).

Quindi, se si avesse scelto di utilizzare un dbms relazionale come server di sistema non si sarebbe in grado di notificare i servizi client.

Le modifiche sulla tabella infatti vengono salvate sul file di log audit trail ma non viene rispettato il requisito principale del sistema.

Sul registro è anche possibile salvare le informazioni sul dato modificato, quale account ha eseguito l’azione e quando si è verificata l’operazione stessa.

Le tecniche di auditing, pongono maggiormente il proprio accento sulla sicurezza dei dati.

In genere, l’auditing viene utilizzato infatti per:

* Assegnare ad ogni utente delle “responsabilità” per le azioni eseguite in un particolare schema, tabella di un database.
* Impedire agli utenti del database di azioni inappropriate basandosi sulle “responsabilità” dell’utente stesso.

L'implementazione degli audit trail aiuta a rendere il comportamento dell'utente più appropriato perché sa che i suoi record utente possono essere ricondotti alla sua identità.

Pertanto, aiuta, inoltre, a prevenire fughe di dati importanti aziendali.

* Rilevamento dei problemi con un'autorizzazione o implementazione del controllo degli accessi e assistenza nella rivalutazione delle autorizzazioni degli utenti.

 Gli audit trail consentono di identificare gli abusi dei diritti di accesso da parte di utenti normali o privilegiati, quindi aiutano a valutare i diritti appropriati per questi utenti.

* Monitoraggio e raccolta di informazioni su attività specifiche del database. A volte gli audit trail possono essere utili per raccogliere informazioni statistiche.

Le tecniche di auditing sono quindi utilizzate soprattutto per prevenire la compromissione dei dati aziendali.

Alcune delle minacce più gravi, per un’azienda, provengono infatti dagli attuali dipendenti con accesso autorizzato.

Tutte queste caratteristiche possono essere soddisfatte implementando meccanismi di controllo nativi del dbms stesso oppure è possibile utilizzare software esterni dedicati.

Bisogna specificare, però, che l’audit nativo (di ciascun dbms) tende a causare un sovraccarico sul server del database.

Gli archivi di audit di grandi dimensioni richiedono, infatti, l’archiviazione del database.

I dati di audit, inoltre, non vengono acquisiti nel formato richiesto dai revisori e dai team di sicurezza [[6]](#DataSunrise).

Pertanto, la soluzione migliore per soddisfare queste esigenze è l’integrazione di un software autonomo dedicato (es DataSunrise, Imperva Data Activity Monitoring (DAM)).

## RabbitMQ cenni

RabbitMQ è sicuramente un altro diverso tipo di tecnologia che potrebbe essere utilizzata per implementare la gestione dello scambio dei messaggi tra le diverse entità del sistema [[7]](#RabbitMQ) .

Rappresenta, infatti, un [message-oriented middleware](https://it.wikipedia.org/wiki/Message-oriented_middleware" \o "Message-oriented middleware) detto anche [broker di messaggistica](https://it.wikipedia.org/w/index.php?title=Broker_di_messaggistica&action=edit&redlink=1) che andrebbe a svolgere il ruolo di server del sistema.

Un [broker di messaggi](https://en.wikipedia.org/wiki/Message_broker) è un programma intermedio che traduce un messaggio dal protocollo di messaggistica assunto dal mittente al protocollo di messaggistica del ricevitore.

Rappresenta, inoltre, un modello architetturale per il routing dei messaggi e implementa il pattern publish-subscribe.

Il suo scopo principale è di prendere i messaggi in arrivo dalle applicazioni publisher ed eseguire alcune azioni su di essi.

Ad esempio, il broker di messaggi può essere utilizzato per gestire una coda di messaggi per più ricevitori, fornendo memoria affidabile, garantendo, soprattutto, la consegna dei messaggi.

Nel tipico scenario dello scambio di messaggi tra le diverse entità coinvolte, RabbitMQ introduce, oltre la presenza del Publisher, del Consumer e della Queue, un nuovo elemento chiamato Exchange.

Il Publisher definito da RabbitMQ non invia più il messaggio direttamente alla coda, ma passa dall’Exchange, il quale crea a sua volta la comunicazione con la coda.

RabbitMQ offre la possibilità di utilizzare diversi tipi di Exchange in base al proprio contesto.

Si possono, tuttavia, delineare tre principali categorie:

* Fanout
* Direct
* Topic

Ognuna di queste categorie definisce un diverso comportamento rispetto a come viene indirizzato il messaggio dall’Exchange alle code.

Utilizzando l’Exchange viene a determinarsi il sistema definito dal pattern Publish-Subscriber.

Nella realtà, spesso, non sarà presente un solo consumatore, bensì migliaia.

Per questo motivo il messaggio pubblicato dal Publisher viene inviato a tutte le code sottoscritte ad uno specifico Exchange relativo al Publisher mittente.

## Conclusioni sulle altre tecnologie utilizzabili

Ho cercato di sviluppare, fin dal principio, il design del sistema indipendentemente dal tipo di tecnologia scelto per le varie entità coinvolte.

Se, infatti, in futuro fosse necessario utilizzare un diverso sistema dbms o un’altra tecnologia rispetto alle scelte fatte inizialmente si potranno applicare poche e semplici modifiche di refactoring sulle classi di libreria implementate e presentate nel capitolo 5.

**Capitolo 3**

# Design libreria UtilityRethink

Successivamente allo studio del dbms RethinkDb, il lavoro ha previsto l’implementazione della libreria di sistema chiamata UtilityRethink.

UtilityRethink è stata scritta in .Net per permettere ai software di backend e frontend aziendali di interrogare i server RethinkDb.

## Aspetti statici

### Base Dati

Siccome nel contesto aziendale fosse necessario mantenere le notifiche di eventi generati dai propri software di frontend e beckend è stato necessario considerare solamente l’utilizzo di una tabella.

Ho inserito quindi nei tre dbms dei server implementati una tabella “Notifications” che raccogliesse tutte le possibili notifiche aziendali.

Notifiche

Rethinkdb essendo un db di tipo non relazionale accetta la diversità di campi tra documenti di una stessa tabella.

Diversamente dai dbms relazionali, RethinkDb non segue uno schema fisso per le proprie tabelle

Ho sfruttato quindi questa caratteristica per raccogliere due diversi tipi di notifiche all’interno della stessa tabella “Notifications”.

L’unico vincolo impostato è che per inserire qualsiasi nuovo tipo di notifica in futuro su “Notifications” dovrà obbligatoriamente avere i campi della classe astratta “Notification” di UtilityRethink.

(Foto Notification)

Dal punto di vista dei servizi client della libreria, ogni classe che rappresenta un tipo di notifica deve ereditare dalla classe base Notification sopra elencata.

Grazie a questo design della libreria, se in futuro sarà necessario aggiungere qualche altro tipo di notifica sul database non sarà un problema.

La classe c# che “mapperà” il nuovo tipo di notifica dovrà semplicemente ereditare dalla classe astratta “Notification”.

Le funzionalità gestionali delle notifiche offerte dalla libreria UtilityRethink, (vedi IQueryNotification e IRXNotifier), inoltre, lavorano con qualsiasi tipo di notifica sempre a patto che, la classe della notifica, erediti da “Notification”.

Per implementare questa soluzione, ho utilizzato le proprietà di Reflection e dei generici. La reflection, in particolare è servita per riuscire a determinare a run-time il tipo di una notifica (campo “type” di “Notification).

Nel contesto aziendale esistono, in particolare, due diversi tipi di notifiche.

Le notifiche di nuovo nuovo dato, e le notifiche di esecuzione.

Notifiche di nuovo dato

(foto Notifiche di nuovo dato)

Le notifiche di nuovo dato del db sono “mappate” dalla libreria UtilityRethink come “NotificationNewData”.

Le notifiche di nuovo dato hanno tutti i campi “base” delle notifiche e hanno, in aggiunta, un campo “Table”.

“Table” servirà a identificare la tabella del db reale aziendale modificata, ad esempio in seguito ad un’operazione di insert o update di un dato.

In questo modo, in seguito, sarà possibile aggiornare anche la “cache del sistema aziendale” su cui vengono effettuate le ricerche.

Il sistema di caching aziendale viene molto utilizzato dall’azienda per richiedere i dati in maniera più performante rispetto a interrogare direttamente il “db vero”.

Se per ottenere i dati, tutti interrogassero il db reale sarebbe infatti molto dispendioso.

Notifiche di esecuzione

(Foto notifiche di esec)

Le notifiche di esecuzione, invece, sono “mappate” da UtilityRethink come “NotificationExec”.

Esse serviranno a notificare i servizi software di frontend e beckend aziendali sullo stato di avanzamento di un Task/processo.

Hanno tutti i campi base delle notifiche e in aggiunta un campo idExec che rappresenta l’id del Task aziendale

### Design UML applicativo

#### Design della Struttura generale UML libreria UtilityRethink

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\StrutturaGeneraleUml.jpg

(cambiare mettendo le interfacce? A mio parere no perché sono le implementazioni a “comporsi” di altri oggetti)

#### UtilityRethink

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\IUtilityRethink.JPG

E’ la classe fondamentale della libreria.

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\FotoCodice\UtilityRethinkCodice.JPG

Successivamente all’istanziazione di UtilityRethink viene creata una nuova connessione verso i nodi del server aventi indirizzi precedentemente specificati (“indirizzoIp:numero Porta”).

In seguito alla connessione al server, UtilityRethink “connette” l’applicativo al db specificato.

Ad esempio, nell’immagine sopra, il db si chiama “test”.

Se sul server rethink non esiste il database allora ne viene creato e su di esso vengono inserite tutte le tabelle di “Sistema”.

Nel contesto aziendale, l’unica tabella di sistema da considerare è “Notifications”

UtilityRethink consente all’utente di interagire con il db richiesto sul cluster attraverso i manager dell’applicativo: DbManager e INotificationsManager.

In particolare, INotificationsManager è il manager della tabella di sistema “Notification”.

Se in futuro sarà necessario avere altre tabelle di sistema basterà aggiungere a UtilityRethink il suo manager chiamato “INomeTabellaManager”

La stessa istanza di “connessione” al db del server creata, viene inseguito “passata” a tutti i sotto-componenti di UtilityRethink per poter essere utilizzata.

#### DbManager

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\DbManager.JPG

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\FotoCodice\DbManagerCodice.JPG

Il DbManager è a tutti gli effetti il gestore del Db specificato inizialmente.

Tra le funzionalità offerte troviamo la possibilità di creare e eliminare tabelle e indici.

Attenzione: Non è tuttavia possibile eliminare le tabelle di sistema del db (es “Notification”).

Il DbManager offre, infine, la possibilità, di riconfigurare le repliche e le shard di una tabella del db.

#### NotificationsManager

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\NotificationManager.JPG

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\FotoCodice\NotificationsManager.JPG

Il NotificationsManager rappresenta, invece, il manager della tabella di sistema ”Notifications”.

Come sottolineato precedentemente, se in futuro si avesse la necessità di avere altre “tabelle di sistema” si potrà creare semplicemente il suo nuovo manager per poi essere gestito da UtilityRethink.

INotificationsManager ha due sotto componenti fondamentali alla gestione delle notifiche: IQueryNotification e IRXNotifier.

#### IQueryNotification

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\QueryNotifications.JPG

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\FotoCodice\QueryNotificationCodice.JPG

IQueryNotification è una funzionalità dell’INotificationsManager che permette all’utilizzatore della libreria di effettuare le “interrogazioni” alla tabella di sistema “Notifications”.

E’ un componente che esegue solamente operazioni CRUD

Offre le funzionalità di inserimento di nuove notifiche, cancellazione di notifiche con un determinato id e richiedere Notifiche in base a certi valori dei suoi campi.

(Utilizzo dei generici)

(Foto queryNotification per le Notifiche di nuovo dato)

(Foto queryNotification per le Notifiche di exec)

#### IRXNotifier

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\RXNotifier.JPG

(Foto getter IRXNotifier)

IRXNotifier è il notificatore di eventi.

Offre le proprie funzionalità sempre per la tabella di sistema “Notifications”.

Grazie agli approfondimenti svolti durante lo sviluppo dell’applicativo sul pattern observer e ad una attento utilizzo delle librerie “bchavez/RethinkDb.Driver” e Reactive Extensions (Rx) è stato possibile implementare il notificatore [[8]](#Consuming_changefeed) .

Il notificatore rimane in ascolto ad eventi sulla tabella “Notifications”.

In particolare, può intercettare eventi che riguardano solamente notifiche che abbiano certi valori di argomento (campo “arg”).

Gli eventi possono essere inserimento, cancellazione, modifica di una notifica che abbia come argomento uno tra quelli presi in considerazione.

In seguito alla “richiesta di registrazione” viene restituito un NotificationSubscription, oggetto che raccoglie l’observable con un suo id.

Ora, passaggio fondamentale, chi utilizza la libreria dovrà sottoscrivere l’observable a dei metodi implementati in base alle proprie esigenze.

Per esempio come mostrato in figura, in caso di intercettazione corretta di un evento verrà chiamato il metodo onNext su cui è possibile manipolare la nuova Notifica oppure la vecchia (in caso di update).

In caso di errore verrà chiamato il metodo OnError mentre al termine dell’ascolto (se verrà chiamato il metodo StopListening sull’observable) verrà invocato il metodo OnComplete.

I tre metodi, OnNext, OnError e OnComplete verrano in questo modo implementati lato cliente ognuno in base alle proprie esigenze [[9]](#notifier_es) .

Esempio di utilizzo notificatore sulle notifiche di esecuzione:

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\ObservableCodice.JPG

(x ha in se lo stato dell’evento catturato: nuovo Dato, vecchio Dato, ecc …)

Per ottenere un notificatore di qualsiasi altro tipo di notifica, sempre a patto che erediti dalla classe base “Notification.cs”, come ad esempio le notifiche di nuovo dato (“NotificationNewData”), è sufficiente sostuire dentro le “<>” “NotificationNewData” al posto di “NotificationExec”.

##### Esempio di OnNext, OnError e OnComplete

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\FotoCodice\OnNextOnErrorOnCompleteCodice.JPG

#### Connection

# C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\Uml\Connection.JPG

L’ oggetto ConnectionNodes viene istanziato inizialmente da UtilityRethink per poi essere utilizzato da tutti i sotto-componenti (IQueryNotifications e IRXNotifier).

I sotto-componenti, infatti, per interfacciarsi al server RethinkDb utilizzano quindi la stessa istanza di ConnectionNodes.

ConnectionNodes tiene traccia dei nodi del server a cui è connesso tramite i DbOptions.

In questo caso, in particolare, possono essere 1, 2 o 5, in base al numero dei nodi del server su cui ci si è connessi.

Il campo “conn” di ConnectionNodes rappresenta il vero oggetto di connessione al database.

Esso viene istanziato solamente alla prima chiamata del metodo GetConnection(), ovvero alla prima interazione col server RethinkDB.

Alle successive chiamate di GetConnection, da parte dei sotto componenti dei vari Manager non viene più istanziato ma viene sempre restituito lo stesso oggetto.

Riinizializzare più volte “conn” risulterebbe infatti un’operazione pesante.

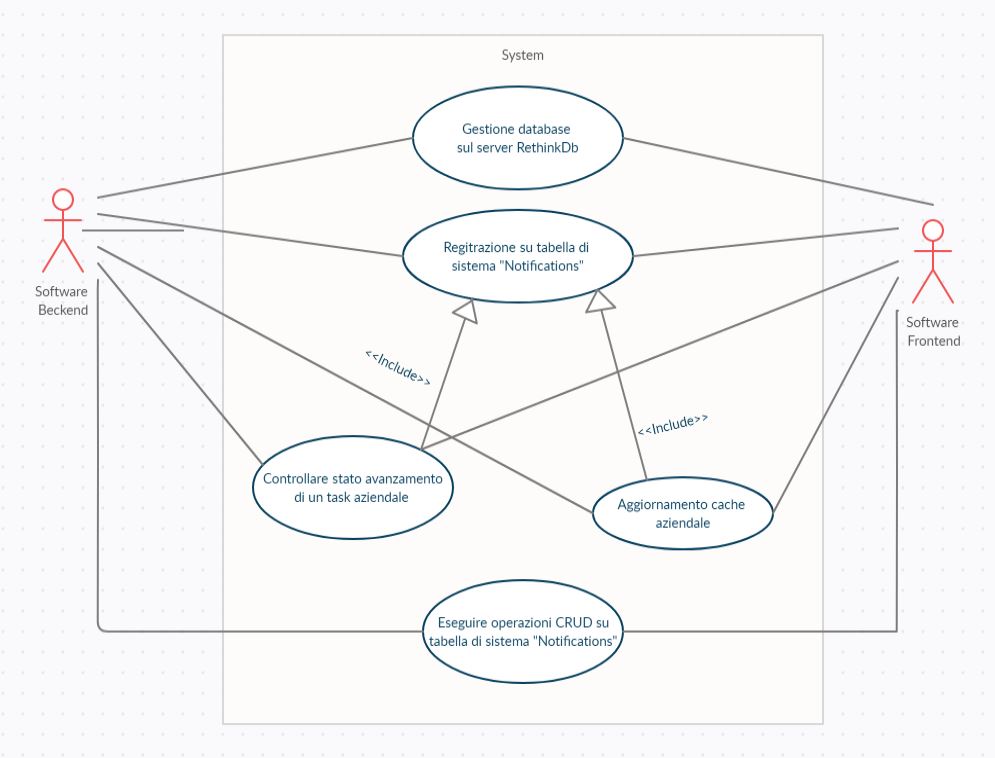
La connessione al server Rethink viene eseguita grazie ad un’algoritmo di RoundRobin, implementato dalla libreria “Bchavez/Rethinkdb”, che sceglie a run-time un nodo disponibile al momento.

ConnectionNodes gestisce, infine, anche i tentativi di connessione su indirizzi Ip non raggiungibili [[10]](#connessione_roundRobin) .

Se dopo 20 secondi non si riesce a connettere al server viene fatto scattare il Timeout per la mancata connessione.

## Aspetti dinamici

### Caso d’uso aziendale



#### Simple Injector

Nel codice dei servizi software di frontend e beckend aziendali serviva la possibilità di richiamare la stessa istanza della libreria UtilityRethink in diversi punti del codice.

Era inoltre necessario fornire un punto di accesso globale a tale istanza.

A tal proposito, ho approfondito i design patterns Singleton e Factory method per poter utilizzare correttamente la libreria Simple Injector in .Net.

Attraverso la classe container, di Simple Injector, è infatti possibile registrare un’istanza di una classe come se fosse un “singleton”.

Successivamente alla registrazione l’istanza registrata può essere richiamata in diversi punti del codice.

Ho sfruttato, quindi, questa caratteristica della libreria Simple Injector per simulare il pattern Singleton sulla classe principale della libreria “UtilityRethink.cs”.

# 

### Diagrammi di attività

(foto diagrammi attivita)

**Capitolo 4**

# Scelte architetturali per il caso d’uso aziendale

Inizialmente, ho ricercato il miglior software utilizzabile per la realizzazione del sistema.

Il progetto è stato implementato sull’ambiente di sviluppo Visual Studio e le funzionalità offerte da Git sono state necessarie alla creazione e mantenimento del repository di progetto online.

## Software utilizzato per il server

Per il caso d’uso aziendale è stato deciso di affidare il ruolo del server di sistema al dbms RethinkDb [[8]](#RethinkDb).

RethinkDb

RethinkDB è un dbms non relazionale di tipo documentale [[9]](#RethinkDb_domande_frequenti).

Come la maggior parte dei sistemi dbms non relazionale anche i dati di RethinkDB presenti in una tabella non seguono uno schema fisso predefinito.

I vari record infatti all’interno di una stessa tabella possono presentare campi diversi.

RethinkDB è il primo dbms non relazionale scalabile e open source costruito appositamente per il Web in tempo reale.

Si basa su un’architettura fondamentalmente diversa dalla maggior parte dei dbms tradizionali (es MongoDB) esponendo una nuova modalità di accesso.

Invece di eseguire il polling per le modifiche avvenute sui dati di un database, sovraccaricando il sistema, lo sviluppatore può richiedere a RethinkDb di inviare continuamente i risultati di query alle applicazioni client in tempo reale.

L’architettura push in tempo reale di RethinkDB riduce drasticamente i tempi e gli sforzi necessari alla creazione di applicazioni scalabili e realtime.

RethinkDB è, in generale, un ottima scelta quando le applicazioni necessitano di sapere dei cambiamenti dei dati in tempo reale.

Per questa motivazione il dbms è risultato essere quindi, una scelta ottimale per la realizzazione del server di sistema.

Altri casi d’uso in cui gli applicativi hanno beneficiato dell’architettura push in tempo reale offerta da RethinkDB sono:

* Web collaborativo e app mobile
* Streaming di applicazioni di analisi
* Giochi multiplayer
* Mercati in tempo reale
* Dispositivi collegati

Ad esempio, quando un utente cambia la posizione di un pulsante in un’applicazione di progettazione collaborativa, il server deve informare gli altri utenti che stanno lavorando contemporaneamente allo stesso progetto.

L’adattamento dei sistemi di gestionali di database alle esigenze in tempo reale rappresenta ancora un’enorme sfida ingegneristica.

In generale, RethinkDB non è invece una buona scelta quando:

* Si ha la necessità di rispettare le proprietà ACID ovvero atomicità, coerenza, isolamento e durabilità dei dati.
* Bisogna rispettare necessariamente lo schema di una tabella (record con gli stessi campi)
* Si eseguono analisi ad alta intensità di calcolo. In questo caso affidarsi ad un sistema Big Data come Hadoop o ad un archivio orientato alle colonne come Vertica.
* Se si ha la necessità di avere una forte disponibilità in scrittura dei dati. In alcuni casi RethinkDB, infatti, scambia la disponibilità in scrittura a favore della coerenza dei dati.

Per interrogare un server RethinkDb viene utilizzato il linguaggio ReQl [[10]](#ReQL) .

ReQL è un linguaggio che permette la manipolazione di documenti JSON in maniera conveniente ed efficace.

Le query vengono “costruite” attraverso chiamate di funzione nel linguaggio di programmazione scelto, nel nostro caso sfruttando l’api in c# RethinkDB/BChavez.

Tutte le query ReQL, inoltre, sono concatenabili attraverso l’operatore “.” e sebbene siano costruite sull’applicativo client vengono eseguite interamente sul server del database una volta che si chiama il comando run.

Il Reql è, infine, un linguaggio di query, che riesce ad eseguire tutte le operazioni implementabili attraverso il linguaggio SQL classico.

Permette, infatti, l’esecuzione di operazioni di “join” tra tabelle e le funzioni di aggregazione.

(Mettere foto es Query Reql: )

Docker

Docker Desktop è stato un secondo software largamente utilizzato per la realizzazione e gestione del server di sistema [[11]](#Docker) .

(Docker container e immagini, cosa sono ?)

Un container è un’unità software che impacchetta il codice e tutte le sue dipendenze in modo che l’applicazione sia eseguita affidabilmente da un ambiente di elaborazione a un altro.

Un’immagine del container Docker rappresenta, invece, un pacchetto software leggero ed eseguibile che include il necessario per l’esecuzione dell’applicazione.

Le immagini dei container diventano contenitori in fase di esecuzione e, nel caso dei Docker container, le immagini diventano contenitori quando vengono eseguite su Docker Engine.

Il software containerizzato sarà infine eseguibile sempre allo stesso modo, indipendentemente dall’infrastruttura/sistema operativo [12].

Docker Desktop è un’applicazione per macchine MacOS e Windows per la creazione e condivisione di applicazioni e micro servizi containerizzati.

Siccome nel contesto aziendale il sistema esegue sul sistema operativo Windows è stata utilizzata la versione Docker Desktop for windows [[13]](#docker_for_windows) .

I comandi offerti da Docker Desktop sono stati fondamentali per l’implementazione e mantenimento del server Rethinkdb.

Consente, inoltre, di sfruttare immagini e modelli certificati.

E’ inoltre inoltre possibile estendendere il proprio ambiente di lavoro ad un repository sicuro su Docker Hub per l’integrazione continua del server RethinkDB containerizzato.

## Software utilizzati dalla libreria di sistema implementata

Tramite lo store NuGet di Visual Studio è possibile scaricare diverse librerie di supporto ai propri progetti.

Ho utilizzato le Librerie bchavez/ RethinkDb.Driver, Reactive Extensions (Rx) e Simple Injector.

“bchavez/ RethinkDb.Driver” è un driver che fornisce un’api in .Net, utile per l’interfacciamento al server Rethinkdb [[14]](#Bchavez_RethinkDb_driver) .

Pur non essendo un driver “ufficiale”, è molto simile a quello ufficiale scritto in java.

Per implementare il manager delle notifiche è stata utilizzata la libreria Reactive Extensions (Rx).

Reactive Extensions (Rx) è una libreria per la composizione di programmi asincroni e basati su eventi utilizzando sequenze osservabili e operatori di query in stile LINQ [[15]](#Reactive_extensions).

Simple Injector, infine, sarà utilizzata dai sofware di frontend e beckend aziendali per gestire e richiamare le funzionalità dell’ applicativo.

Grazie ad essa, è infatti possibile registrare un’istanza di una classe e fare in modo che quella restituita sia sempre la stessa al momento della registrazione [[16]](#simple_injector) .

Attraverso questo meccanismo di Simple Injector è stato possibile implementare automaticamente il Pattern Singleton sulla classe principale della libreria (UtilityRethink).

## Metodologia di Progettazione

In seguito alla scelta del software da utilizzare per la realizzazione del sistema, è stato scelto Scrum come metodologia di progettazione agile [[17].](#Scrum)

Esso permette la divisione del progetto in Sprint, ovvero intervalli di tempo di durata fissa generalmente da uno a quattro settimane.

Gli Sprint sono l’unità di base dello sviluppo in Scrum.

Ognuno di essi è preceduto da una riunione di pianificazione in cui vengono identificati gli obiettivi e vengono stimati i tempi.

Durante uno sprint non è permesso cambiare gli obiettivi, quindi le modifiche sono sospese fino alla successiva riunione di pianificazione, e potranno essere prese in considerazione nel successivo Sprint.

Durante lo sviluppo dell’applicazione ogni attività svolta ha assunto i seguenti stati: “ToDo”, “Doing” e “Done”.

Scrum è una metodologia di progettazione che si utilizza quando i requisiti sono ben chiari e una volta iniziato il progetto non ci sono nuove attività che entrano nella lista “ToDo”.

A tal proposito, per la gestione del progetto ho utilizzato Jira Software attraverso credenziali datemi dall’azienda.

Quando iniziavo una nuova attività (Sprint) essa passava dallo stato “ToDo” al “Doing” per poi andare in “Done” quando veniva completato.

**Capitolo 5**

# Implementazione Server

Per poter testare le capacità del dbms RethinkDb, come anticipato nel capitolo sulle scelte architetturali, è stata utilizzata la tecnologia offerta dai container docker.

Attraverso il sito Docker-hub, sito web su cui è possibile cercare e scaricare immagini docker di applicativi, abbiamo scelto di utilizzare l’immagine ufficiale “rethinkdb” mantenuta dalla società RethinkDb [[18]](#immagine_rethinkdb).

In seguito al download dell’immagine, utilizzando sul terminale i comandi di Docker-compose sono riuscito a costruire tre diversi tipi di server Rethink.

Server a singolo nodo

(foto terminale e interfaccia rethink)

Server a due nodi

(foto terminale e interfaccia rethink)

Server a cinque nodi

(foto terminale e interfaccia rethink)

Ogni nodo del server rappresenta di fatto un container.

Nei due casi del server a più nodi, attraverso lo script bash “start-rethinkdb-cluster.sh” è stato possibile eseguire i comandi necessari al join dei nodi-container Rethink in unico server [[19]](#RethinkDb_join_cluster) .

Grazie all’implementazione dello script, i nodi del server collaborano tra di loro per rispondere alle richieste in arrivo dai servizi client e anche per poter sfruttare efficacemente le politiche di Replication e di Sharding.

Bisogna specificare che, in questo caso, il server risiede completamente sulla macchina locale avendo utilizzato i comandi di Docker-compose [[20]](#Docker_compose) .

In un contesto più articolato, se necessario, è possibile utilizzare altre tecnologie per distribuire i diversi nodi del server su più macchine fisiche.

Alcune tecnologie di riferimento sarebbero ad esempio Docker Swarm o Kubernetes.

## Sharding e Replication

### (questa parte è stata letta dalle slide di BDA , cosa mettere in biblio?)

Due aspetti che bisogna considerare, durante lo sviluppo di un cluster di un dbms non relazionale, sono le politiche di Sharding e Replication sui dati.

In particolare, le politiche di Sharding servono a distribuire i dati al meglio sul Cluster.

I dati vengono inizialmente suddivisi in piccole parti, dette shards, per essere successivamente inserite al meglio nei diversi nodi del server.

Una buona strategia di sharding è fondamentale per ottimizzare le performance.

Le diverse partizioni dei dati, inseguito ad una richiesta, potranno essere lette in parallelo dai diversi nodi.

La Replication, invece, serve alla creazione e gestione di copie dei dati sui nodi del Cluster.

In caso di “node failure” è una tecnica che previene la perdita dei dati nel caso siano stati precedentemente salvati anche su un secondo nodo.

Attraverso la Replication, inoltre, un dato può essere letto in parallelo da diversi utenti siccome è presente in più nodi del Cluster.

Tramite la replicazione, è inoltre possibile, che la lettura di un dato avvenga da un nodo del server vicino geograficamente a dove è stato richiesto.

Se esistesse una sola copia di un dato ci sarebbe, infatti, altà probabilità che risieda su un nodo particolarmente lontano rispetto a dove è stato richiesto.

Ci sarebbero, di conseguenza, alti costi di rete.

E’ possibile gestire le politiche di sharding e replication su un cluster RethinkDB tramite interfaccia web o attraverso comandi ReQL [[21]](#rethinkdb_Sharding_replication) .

### Gestione con Interfaccia Web

Le impostazioni possono essere controllate facilmente dalla console di amministrazione web.

Quando si utilizza l'interfaccia utente Web, è sufficiente specificare il numero di frammenti desiderati e, in base ai dati disponibili, è RethinkDB stesso che determinerà i migliori punti di divisione per mantenere i frammenti bilanciati.

Per dividere i propri dati:

Dirigersi sulla “vista” della tabella ( *Tabelle* → *nome tabella* ).

Fare clic sul pulsante *Riconfigura* .

Impostare il numero di frammenti e repliche che si desidera.

Fare clic sul pulsante *Applica configurazione*

(Foto Reconfigure tabella Notifications table overview)

### Gestione con ReQL

Il linguaggio ReQL consente anche un controllo più dettagliato sulla replica di un dato.

E’ possibile, quindi, oltre all’interfaccia grafica, gestire le repliche utilizzando comandi forniti dal linguaggio ReQL.

Se si avesse la necessità di ottenere configurazioni più avanzate, (es decidere su quale nodo di un Cluster inserire una replica di una tabella), sarebbe necessario rivolgersi a comandi del linguaggio ReQL.

Se si utilizza l’interfaccia grafica vengono infatti autodeterminati i migliori punti di divisione.

Per raggruppare i nodi nei cluster, RethinkDb utilizza i tag di ciascun nodo definiti dall’utente.

A tutti i nodi in un cluster RethinkDB possono essere assegnati zero o più tag:

rethinkdb --server-tag us\_west

I tag possono anche essere utilizzati nelle configurazioni di tabella per mappare le repliche ai server specificati dal tag utilizzando il comando ReQL reconfigure.

Per assegnare 3 repliche della tabella users al nodo us\_west e 2 al nodo us\_east:

r**.**table('users')**.**reconfigure(shards**=**2, replicas**=**{'us\_west':3,

'us\_east':2}, primary\_replica\_tag**=**'us\_east')**.**run(conn)

### Gestione nella libreria

Nella libreria del sistema (UtilityRethink) implementata, un suo sottocomponente (il DbManager) ha anch’esso una funzionalità di reconfigure che permetterà di assegnare il numero di shard e repliche dei dati di una tabella.

Non avendo al momento la necessità di dover assegnare ad uno specifico nodo una determinata replica ho lasciato a RethinkDb il compito di autodeterminarsi la migliore suddivisione dei dati.

## Comandi per la gestione del server implementato

Aprire il terminale e dirigersi sulla cartella di progetto corrispondente all’opzione del server scelto (uno, due o cinque nodi).

In seguito, scrivere sul terminale: “docker-compose -f docker-compose.yml build”.

Questo comando è necessario eseguirlo solamente al primo utilizzo del tipo di server scelto.

In seguito all’esecuzione, infatti, viene costruita e salvata permanentemente sul proprio dispositivo l’immagine docker del server RethinkDB (se necessario è possibile successivamente cancellare l’immagine, vedi documentazione Docker).

Successivamente eseguendo, sempre su terminale, il comando “docker-compose -f docker-compose.yml up -d” il server RethinkDB è online e i suoi nodi sono in attesa di ricevere richieste.

Per stoppare il server digitare “docker-compose -f docker-compose.yml stop”.

E’ importante sottolineare che in seguito allo stop lo stato dell’immagine docker del server RethinkDB viene salvato.

Se, ad esempio, inserissi dei nuovi dati sul db e stoppassi successivamente il server, al prossimo comando di up il server mantiene i dati precedentemente caricati.

I tre comandi elencati valgono per tutti e tre i tipi di server implementati (1, 2, 5 nodi).

## Test e Performance

Il test sulle performance riguarda uno studio del comportamento del dbms RethinkDB in termini di velocità di lettura e scrittura.

Siccome attraverso l'interfaccia utente Web, è possibile gestire semplicemente le politiche di Sharding e Replication, nei due casi in cui il server è composto da più nodi ho testato le performance e il comportamento del dbms tenendo anche in considerazione questi due aspetti.

In questo test, il server RethinkDB risiede interamente sulla macchina locale avendo utilizzato i comandi di Docker-compose.

Per uno studio completo del comportamento e delle performance bisognerebbe oltre ai casi analizzati distribuire anche i vari nodi del server Rethink (nei casi a più nodi) su diverse macchine.

Ho progettato un piccolo applicativo scritto in .Net finalizzato alla fase di test.

In particolare sui tre server RethinkDB sono state create due tabelle: “Author” e “Post” all’interno di un database “test”.

(Foto db con tabelle Author e Post)

Sono stati inseriti, inseguito, circa 200 documenti su Author e 20 mila su Post.

Gli autori sono caratterizzati da: id, nome, età, hobby.

Mentre i post da: id, author\_id, titolo, contenuto.

Da ogni post è possibile ricavare il suo autore attraverso il campo “author\_id” .

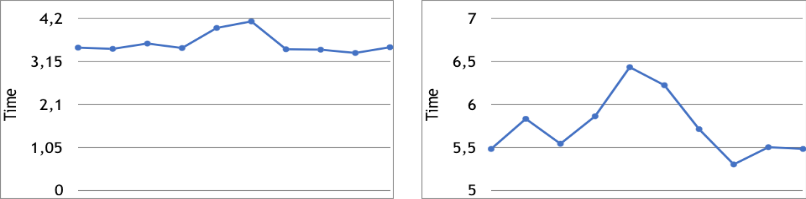
Nei casi a 1 e 5 nodi del server ho misurato in secondi quanto tempo impiega Rethink ad effettuare 50 inserimenti su Post e una query di selezione basata tra un join tra Author e Post per riuscire a determinare quanti post ha scritto ogni autore.

(Foto query)

### Risultati Test Performance

Server Rethink a singolo nodo

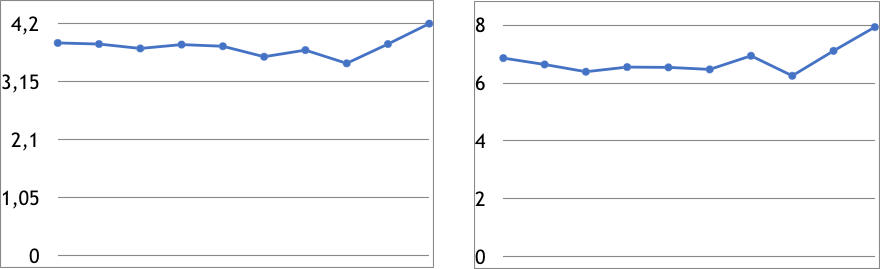
Singolo nodo (non è possibile applicare politiche di sharding e replication):

Select (media: 3,577 s) Insert (media: 5,735 s)

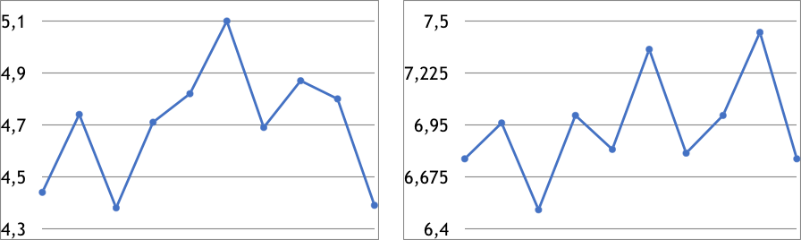
#### Server MySQL a singolo nodo

Server Rethink a cinque nodi

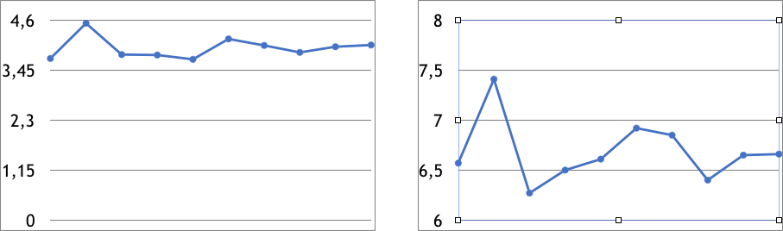
Una shard con cinque repliche ciascuna

Select (media: 3.787 s) Insert (media: 6,768 s)

Cinque shard con una sola repliche

Select (media: 4,694 s) Insert (media: 6,941 s)

Due shard con tre repliche ciascuna

Select (media: 3,963 s) Insert (media: 6,684 s)

**Capitolo 5**

# Sviluppi futuri

-Getter di notifiche sul db Rethink per ora non sfruttano gli indici secondari.

In caso ci fosse l’indice su un campo aggiungere la possibilità di sfruttarlo.

-Dare la possibilità all’utente di poter impostare il timeout della connessione

-Rendere la libreria non dipendente dalla tecnologia utilizzata.

Esempio: se un giorno si volesse sostituire RethinkDb con RabbitMq.

Se un giorno RethinkDb diventasse a pagamento.

Interfaccia principale: INotificationProvider (non avrà il DbManager -> RabbitMq non è un db) viene implementata da UtilityRethink (classe specifica per Utility Rethink)

-IQueryNotifications essendo una classe che esegue solamente operazioni CRUD rinominarla in INotificationRepository. (oggetto repository pattern di progettazione).

-Tutto troppo “Java like”, togliere i metodi getter() e mettere i campi con “{ get; }” seguendo le convenzioni di .Net

-Sostituire nel nome dei metodi “with” con “By” o “On”

-L’idExec delle Notifiche di esecuzione essendo l’id di un Task/processo esterno dalla libreria non deve essere di tipo Guid.

Scegliere tra: generico, string o object.

**Capitolo 6**

# Conclusioni

Questa esperienza l’ho trovata molto formativa.

Stando a stretto contatto con i dipendenti dell’azienda sono riuscito ad apprendere nuovi concetti anche pratici dell’informatica.

Ringraziamenti

Ringrazio infine l’azienda Energy Software per la sua disponibilità e accoglienza che mi ha riservato.

**Capitolo 6**

# Bibliografia

1. Design Publish/Subscribe. URL <https://it.wikipedia.org/wiki/Publish/subscribe>.
2. Database auditing. URL <https://en.wikipedia.org/wiki/Database_audit>.
3. Meccanismi di auditing. URL [https://www.ugiss.org/2008/09/17/meccanismi-di- auditing/](https://www.ugiss.org/2008/09/17/meccanismi-di-   auditing/).
4. SQL Server Audit (Database Engine). URL <https://docs.microsoft.com/it-it/sql/relational-databases/security/auditing/sql-server-audit-database-engine?view=sql-server-ver15>.
5. Why you need a Database audit trail. URL <https://www.imperva.com/blog/why-you-need-a-database-audit-trail/>.
6. Esempio di software per il data-auditing. URL <https://www.datasunrise.com/blog/professional-info/why-you-need-a-database-audit-trail/>
7. RabbitMQ. URL <https://it.wikipedia.org/wiki/RabbitMQ>
8. Consuming changefeed. URL <https://github.com/bchavez/RethinkDb.Driver/wiki/Extra-C%23-Driver-Features#consuming-changefeeds>
9. Esempio di utilizzo combinato delle libreria bchavez/RethinkDb.driver e Reactive Extension. URL <https://github.com/bchavez/RethinkDb.Driver/blob/master/Source/RethinkDb.Driver.Tests/ReQL/ChangeFeedTests.cs>
10. Strategia di Round Robin per la connessione al server RethinkDB. URL <https://github.com/bchavez/RethinkDb.Driver/wiki/Connections-&-Pooling#round-robin-strategy>
11. RethinkDb, the open source database for the realtime web. URL <https://rethinkdb.com/>.
12. RethinkDb domande frequenti. URL <https://rethinkdb.com/faq>.
13. Linguaggio ReQL. URL <https://rethinkdb.com/docs/introduction-to-reql/>
14. Docker Desktop. URL <https://www.docker.com/products/docker-desktop>
15. I Docker container e le immagini. URL https://www.docker.com/resources/what-container
16. Docker Desktop for windows. URL <https://docs.docker.com/docker-for-windows/>
17. Bchavez/RethinkDb.driver. URL <https://github.com/bchavez/RethinkDb.Driver>
18. Reactive extensions. URL <http://reactivex.io/>
19. Simple injector. URL <https://simpleinjector.org/>
20. Metodologia di progettazione Scrum. URL <https://it.wikipedia.org/wiki/Scrum_(informatica)#:~:text=Scrum%20%C3%A8%20un%20framework%20agile,Ken%20Schwaber%20e%20Jeff%20Sutherland>.
21. Immagine RethinkDB utilizzata. URL <https://hub.docker.com/_/rethinkdb>
22. How to start a RethinkDB server. URL <https://rethinkdb.com/docs/start-a-server/>
23. Docker compose. URL <https://docs.docker.com/compose/>
24. Sharding e Replication su RethinkDB. URL <https://rethinkdb.com/docs/sharding-and-replication/>