# 

# **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

**Indice**

[Introduzione 4](#_Toc57922270)

[Azienda 4](#_Toc57922271)

[Problema affrontato (Auditing, notifiche di eventi, Publish/Subscribe) 4](#_Toc57922272)

[Specifiche funzionali 5](#_Toc57922273)

[Specifiche architetturali 6](#_Toc57922274)

[Altri sistemi esistenti (Librerie e tecnologie) 7](#_Toc57922275)

[I Trigger 8](#_Toc57922276)

[Cenni sulle tecniche di auditing 8](#_Toc57922277)

[RabbitMQ cenni 10](#_Toc57922278)

[Conclusioni sulle altre tecnologie 12](#_Toc57922279)

[Scelte architetturali 13](#_Toc57922280)

[Software 13](#_Toc57922281)

[Librerie 14](#_Toc57922282)

[Metodologia di Progettazione 14](#_Toc57922283)

[Implementazione Server 15](#_Toc57922284)

[Sharding e Replication 16](#_Toc57922285)

[Gestione con Interfaccia Web 17](#_Toc57922286)

[Gestione con ReQL 18](#_Toc57922287)

[Gestione nella libreria 19](#_Toc57922288)

[Comandi per la gestione del server 19](#_Toc57922289)

[Test e Performance 20](#_Toc57922290)

[Risultati Test Performance 21](#_Toc57922291)

[Server Rethink a singolo nodo 21](#_Toc57922292)

[Server MySQL a singolo nodo 22](#_Toc57922293)

[Server Rethink a cinque nodi 23](#_Toc57922294)

[Una shard con cinque repliche ciascuna 23](#_Toc57922295)

[Cinque shard con una sola repliche 23](#_Toc57922296)

[Due shard con tre repliche ciascuna 23](#_Toc57922297)

[Design libreria UtilityRethink 24](#_Toc57922298)

[Aspetti statici 24](#_Toc57922299)

[Base Dati 24](#_Toc57922300)

[Notifiche 24](#_Toc57922301)

[Notifiche di nuovo dato 26](#_Toc57922302)

[Notifiche di esecuzione 26](#_Toc57922303)

[Design UML applicativo 27](#_Toc57922304)

[Design della Struttura generale UML libreria UtilityRethink 27](#_Toc57922305)

[UtilityRethink 28](#_Toc57922306)

[DbManager 29](#_Toc57922307)

[NotificationsManager 30](#_Toc57922308)

[IQueryNotification 31](#_Toc57922309)

[IRXNotifier 32](#_Toc57922310)

[Esempio di OnNext, OnError e OnComplete 34](#_Toc57922311)

[Connection 35](#_Toc57922312)

[Aspetti dinamici 36](#_Toc57922313)

[Caso d’uso aziendale 36](#_Toc57922314)

[Simple Injector 37](#_Toc57922315)

[Diagrammi di attività 38](#_Toc57922316)

[Sviluppi futuri 39](#_Toc57922317)

[Conclusioni 40](#_Toc57922318)

[Bibliografia 41](#_Toc57922319)

**Capitolo 1**

# Introduzione

## Azienda

Il tirocinio è stato svolto 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.

Sommario progetto

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

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

La libreria è stata prototipata in C# su piattaforma .NET Core e il codice prodotto è stato organizzato in maniera efficiente e riutilizzabile nei prodotti della società.

Dal punto di vista del database, abbiamo studiato lo schema migliore di rappresentazione dei dati per ottenere le migliori prestazioni possibili a fronte delle necessarie caratteristiche di consistenza ed isolamento.

I risultati ottenuti sono stati oggetto di misurazione e presentazione.

## Problema affrontato (Auditing, notifiche di eventi, Publish/Subscribe)

Il problema affrontato dalla libreria sviluppata si avvicina, concettualmente, alla tecnica di database auditing tipica dei sistemi dbms relazionali.

(https://en.wikipedia.org/wiki/Database\_audit)

Il database auditing implica 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.

[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.

Oltre alla capacità di osservazione e salvataggio delle operazioni svolte su una tabella di un database il lavoro prevedeva un secondo requisito.

La libreria, infatti, è anche in grado di avvisare della modifica avvenuta ai vari servizi software, “in ascolto” sulla tabella del database.

In particolare il sistema permette anche ai vari servizi software di rimanere in ascolto solo su alcuni dati di interesse, disinteressandosi quindi di alcuni messaggi riguardanti altri dati.

(<https://it.wikipedia.org/wiki/Publish/subscribe>)

La libreria è quindi un caso implementativo del design pattern publish/subscribe.

In questo contesto, il mittente dei messaggi è il server del dbms mentre i destinatari sono i servizi di frontend e beckend aziendali.

I due servizi aziendali si interfacciano al server tramite la libreria.

In questo schema, [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 dbms (*publisher*) non è consapevole dell'identità dei destinatari (detti *subscriber*); esso si limita a "pubblicare" i propri messaggi al dispatcher.

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

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

Il meccanismo di sottoscrizione consente ai servizi di precisare nel modo più specifico possibile a quali messaggi sono interessati.

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

Questo schema, implica che al dbms scelto non sia noto quanti e quali siano i servizi che utilizzano la libreria contribuendo di conseguenza alla scalabilità del sistema.

## Specifiche funzionali

Descrizione delle funzionalità (funzionali) + tipo di architettura richiesta (client server, interfaccia web….)

Vedi ad esempio in questo capitolato cosa si intende per requisiti architetturali (<https://www.lilt.it/sites/default/files/allegati/amm-trasparente/2018-11/capitolato-speciale-all.1-specifiche-tecniche.pdf>)

La libreria consente ai servizi che ne usufruiscono di:

* Eseguire operazioni Crud sulle tabelle, considerate dalla libreria di, sistema (es “Notifications”);
* Registrarsi in ascolto sulle operazioni che avvengono in una tabella che la libreria considera “di sistema”.

Per il contesto aziendale, i servizi possono anche registrarsi solo per le notifiche (tabella di sistema “Notifications”) che hanno un certo valore di argomento (campo “arg”).

Se ad esempio viene inserita sulla tabella “Notification” una nuova notifica con campo “arg” uguale ad “A” allora verranno avvisati della nuova notifica inserita tutti i servizi registrati per aventi “arg” uguale ad “A”.

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

* Operazioni di gestione del database come la creazione e cancellazione di nuove tabelle e indici e la riconfigurazione delle replication e shards dei dati di una tabella.

## Specifiche architetturali

I requisiti architetturali e tecnologici sono:

• Architettura client - server;

• Accesso al dbms attraverso i protocolli http ed https;

• Data base non relazionale;

**Capitolo 2**

# Altri sistemi esistenti (Librerie e tecnologie)

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

Come precedentemente accennato nella sezione introduttiva, si può fare riferimento anche alle tecniche di auditing realizzate per i sistemi dbms relazionali.

Un’altra tecnologia/modello che distribuisce messaggi ai vari client RabbitMQ.

## I Trigger

## Cenni sulle tecniche di auditing

Vedi database auding wikipedia :

( <https://solutioncenter.apexsql.com/it/13274-2/> )

https://docs.microsoft.com/it-it/sql/relational-databases/security/auditing/sql-server-audit-database-engine?view=sql-server-ver15

(https://www.ugiss.org/2008/09/17/meccanismi-di-auditing/)

(non andare troppo nello specifico sulle altre tecnologie)

(<https://www.imperva.com/blog/why-you-need-a-database-audit-trail/>)

Il concetto generale di database auditing riguarda il monitoraggio dell’utilizzo dei record di un database all’interno di un dbms relazionale.

Quando si controlla un database, le operazione sui dati possono essere monitorate e registrate in un registro di controllo del dbms, chiamato audit trail,

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.

( <https://www.datasunrise.com/blog/professional-info/why-you-need-a-database-audit-trail/> )

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 si può utilizzare un software esterno dedicato.

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 e i dati di audit non vengono acquisiti nel formato richiesto dai revisori e dai team di sicurezza.

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

(https://it.wikipedia.org/wiki/RabbitMQ)

RabbitMQ è sicuramente un altro tipo di tecnologia che potrebbe essere utilizzata per gestire lo scambio dei messaggi tra le diverse entità dell’applicativo.

Rappresenta di fatto 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).

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

Ho cercato di sviluppare, fin da subito la libreria in .Net, il design della libreria indipendentemente dal tipo di tecnologia scelto.

Se, infatti, in futuro fosse necessario utilizzare un diverso sistema dbms o un’altra tecnologia rispetto alle scelte fatte inizialmente si potrà applicando poche modifiche di refactoring sulle classi di libreria implementate.

**Capitolo 3**

# Scelte architetturali

## Software

Inizialmente, ho ricercato il miglior software utilizzabile per l’applicativo.

Il progetto è stato implementato sull’ambiente di sviluppo Visual Studio.

Abbiamo pensato, per le esigenze del progetto, di utilizzare RethinkDb, un dbms non relazionale di tipo documentale.

(cosa significa db documentale -> le tabelle non seguono uno schema, i documenti di una stessa tabella possono avere campi diversi)

Per interrogare un server RethinkDb viene utilizzato il linguaggio ReQl.

Il Reql è un linguaggio di interrogazione molto simile a SQL classico.

Permette, infatti, di eseguire operazioni di “join” tra tabelle di uno stesso database.

E’ inoltre possibile generare una Pipeline di operazioni concatenandole una dopo l’altra separate da un punto.

(Mettere es Query Reql: )

Tra i software largamente utilizzati troviamo, inoltre, Git e Docker Desktop.

Git è stato fondamentale alla gestione del repository di progetto.

Docker Desktop, invece, è stato utile alla creazione e mantenimento del server Rethinkdb.

(Docker container, cosa sono ?)

## Librerie

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.

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.

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.

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, è stato scelto Scrum come metodologia di progettazione agile.

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 più cose nuove 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.

## Implementazione Server

Per poter testare le capacità del dbms RethinkDb abbiamo pensato di affidarci alla 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.

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 un join dei nodi-container in unico server.

Attraverso questa tecnica, i nodi del server collaborano tra loro per rispondere alle richieste in arrivo e anche per poter sfruttare 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.

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 per questo contesto sarebbero ad esempio Docker Swarm o Kubernetes.

### Sharding e Replication

Due aspetti che bisogna considerare, durante lo sviluppo di un cluster, 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.

(Strategia di sharding mio caso:

vedi paragrafo **Sharding e replica tramite ReQL:**

<https://rethinkdb.com/docs/sharding-and-replication/> li dice che usa politica di hash per lo sharding -> non ho controllo sulla partizione)

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.

(Strategia di Replication da me usata)

#### Gestione con Interfaccia Web

RethinkDB consente di suddividere e replicare il cluster in base ad una tabella.

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 che determinerà i migliori punti di divisione per mantenere i frammenti bilanciati.

Per dividere i propri dati:

Vai alla vista 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 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

Aprire il terminale e dirigersi sulla cartella di progetto corrispondente all’opzione del server scelta (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 Rethink 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 Rethink 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 (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.

Ricordo che il server, in questo test, 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 Rethink ho creato due tabelle: “Author” e “Post” all’interno di un database “test”.

(Foto db con tabelle Author e Post)

Ho inserito 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” .

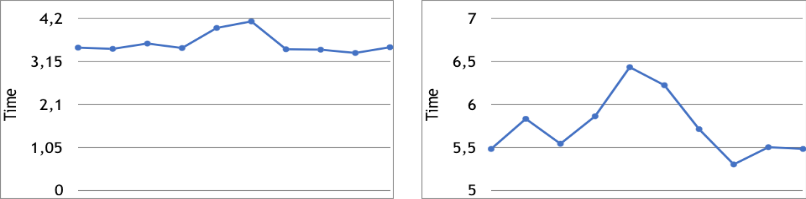
Ho costruito quindi un indice secondario sul campo “author\_id” di “Post” per ottenere le migliori performance possibili.

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 sfruttando quindi l’indice sul campo “author\_id” su “Post” precedentemente costruito.

### Risultati Test Performance

Server Rethink a singolo nodo

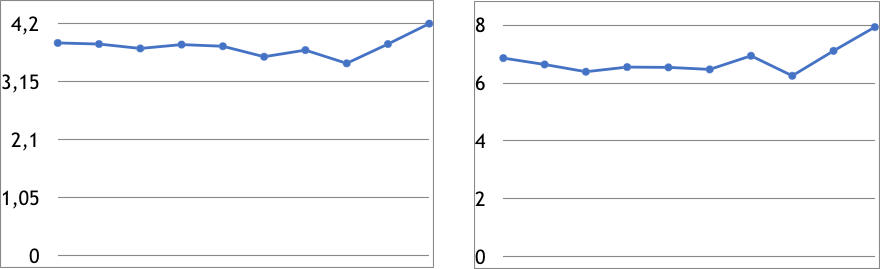
Singolo nodo (non è possibile applicare politiche di sharding e replication):

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

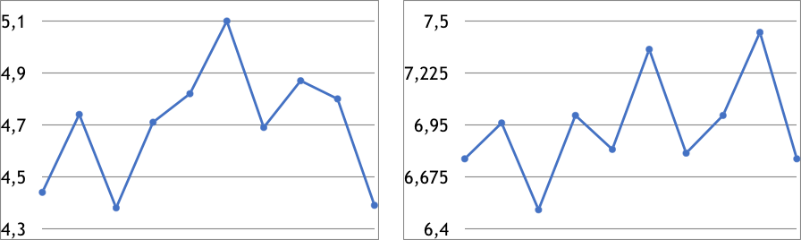
#### Server MySQL a singolo nodo

Server Rethink a cinque nodi

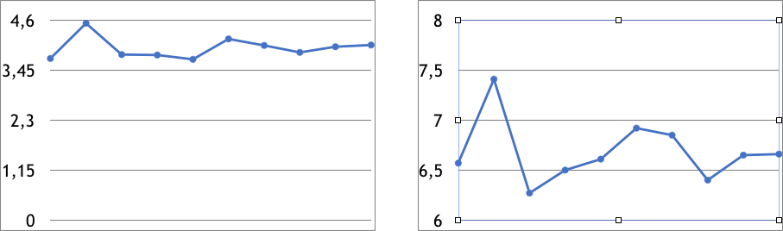
Una shard con cinque repliche ciascuna

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 4**

# Design libreria UtilityRethink

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

UtilityRethink è stata scritta in .Net per permettere ai software di beckend 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 server implementati una tabella Notification 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”.

Qualsiasi tipo di notifica presente sulla “Notification”, per mia imposizione dovrà avere i campi della classe astratta “Notifications”

(Foto Notifications)

Dal punto di vista di UtilityRethink (lato applicativo), 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 non sarà un problema.

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

Le funzionalità gestionali delle notifiche offerte da 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 la proprietà della 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 il tirocinio sul pattern observer e ad una attento utilizzo delle librerie “bchavez/RethinkDb.Driver” e Reactive Extensions (Rx) è stato possibile implementare il notificatore.

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.

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

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

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.

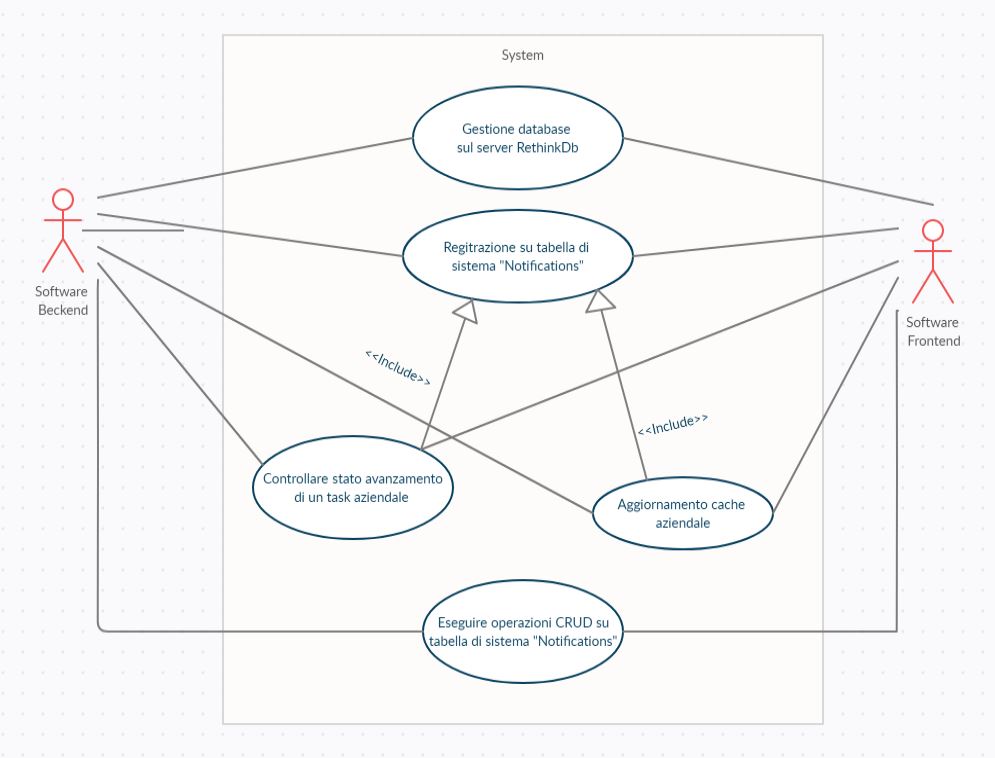
La connessione al server Rethink viene eseguita grazie ad un’algoritmo di RoundRobin, implementato dalla libreria “Bchavez/Rethinkdb”, che sceglie a run-time un nodo disponibile al momento.

ConnectionNodes gestisce, infine, anche i tentativi di connessione su indirizzi Ip non raggiungibili.

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 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