# 

# **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**

[Scelte architetturali per il caso d’uso aziendale 4](#_Toc59113065)

[Software utilizzato per il server 5](#_Toc59113066)

[RethinkDb 5](#_Toc59113067)

[Docker 8](#_Toc59113068)

[Software utilizzati dalla libreria di sistema implementata 9](#_Toc59113069)

[Metodologia di Progettazione 10](#_Toc59113070)

[Implementazione Server 11](#_Toc59113071)

[Sharding e Replication 12](#_Toc59113072)

[(questa parte è stata letta dalle slide di BDA , cosa mettere in biblio?) 12](#_Toc59113073)

[Gestione con Interfaccia Web 13](#_Toc59113074)

[Gestione con ReQL 14](#_Toc59113075)

[Gestione nella libreria 14](#_Toc59113076)

[Comandi per la gestione del server implementato 15](#_Toc59113077)

[Test e Performance 16](#_Toc59113078)

[Risultati Test Performance 17](#_Toc59113079)

[Server Rethink a singolo nodo 17](#_Toc59113080)

[Server MySQL a singolo nodo 17](#_Toc59113081)

[Server Rethink a cinque nodi 18](#_Toc59113082)

[Bibliografia 19](#_Toc59113083)

**Capitolo 4**

# Scelte architetturali per il caso d’uso aziendale

Inizialmente, è stato ricercato il miglior software utilizzabile per la realizzazione del sistema. Per il contesto di utilizzo aziendale, è stato deciso di affidare il ruolo di server del sistema al dbms RethinkDB [[22]](#RethinkDb) mentre il progetto di libreria UtilityRethink, presentato nel capitolo precedente, è stato implementato sull’ambiente di sviluppo Visual Studio e, come già anticipato, è al momento disponibile solo la versione per applicativi scritti in c#.

Le funzionalità offerte da Git, infine, sono state necessarie alla creazione e mantenimento del repository di progetto online.

## Software utilizzato per il server

RethinkDb

Tra tutti i sistemi, elencati e descritti nel secondo capitolo di questa relazione, che potevano essere utilizzati per svolgere il ruolo di server del sistema, è stato scelto di utilizzare RethinkDB.

RethinkDB è un dbms non relazionale di tipo documentale [[23]](#RethinkDb_domande_frequenti) che appartiene, inoltre, alla classe di sistemi database real-time discussi nel paragrafo introduttivo.

Come nella maggior parte dei sistemi dbms non relazionale anche i dati presenti in una tabella di un database RethinkDB non seguono uno schema fisso predefinito. I documenti all’interno di una stessa tabella, infatti, possono avere campi diversi e, per questo motivo, è stato possibile mantenere tutte le notifiche di sistema in un’unica tabella (*Notifications*).

RethinkDB è il primo dbms non relazionale scalabile e open source costruito appositamente per il Web in tempo reale. Essendo un sistema database real-time, 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 e tipica dei sistemi database real-time, riduce i tempi e gli sforzi necessari allo sviluppo di applicazioni scalabili e real-time.

RethinkDB è, quindi, una buona scelta per il proprio progetto quando l’applicazione necessita di sapere dei cambiamenti dei dati in tempo reale. Per questa motivazione, quindi, è risultata essere 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

Per il contesto aziendale, è stato scelto di utilizzare RethinkDB perché rispetto a tutti gli altri software database real-time forniva una migliore documentazione online e ha permesso un’ efficiente implementazione del principale requisito di sistema, ovvero la funzionalità di notificare, agli applicativi client, gli eventi avvenuti su una tabella di proprio interesse del database.

Essendo, inoltre, un sistema dbms, utilizzando RethinkDB è stato possibile archiviare e salvare le notifiche inviate ai servizi client. In questo modo la libreria UtilityRethink implementata garantisce all’utente anche la possibilità di poter controllare, in un secondo momento, le notifiche che sono state inviate perché sono salvate sul database RethinkDB. Ad esempio, vedi funzionalità di libreria *IQueryNotification*, l’utente potrà richiedere tutte le notifiche avvenute in una specifica data.

RethinkDB è, inoltre un sistema fondamentalmente diverso dalle API di sincronizzazione in tempo reale come FireBase, per tre diverse motivazioni [[24]](#RethinkDbvsAPI_sinc_in_temporeale) .

In primo luogo, le API di sincronizzazione in tempo reale sono servizi cloud mentre RethinkDB è un progetto open source. Sebbene RethinkDB sia disponibile anche nel cloud tramite Compose.io e Amazon AWS, può anche essere distribuito nelle proprie infrastrutture senza restrizioni.

In secondo luogo, i sistemi come Firebase, sono limitati alla sincronizzazione dei documenti, mentre RethinkDB è un sistema dbms. Utilizzando RethinkDB è, quindi, anche possibile eseguire query arbitrarie tra cui operazioni di join tra tabelle, sottoquery, query geospaziali e query di aggregazione attraverso il linguaggio di interrogazione fornito chiamato ReQL. I servizi di sincronizzazione in tempo reale, invece, hanno capacità di interrogazione molto più limitate.

Infine, le API di sincronizzazione in tempo reale sono progettate per essere accessibili direttamente dal browser semplificando l’istallazione e l’esecuzione di applicativi di base, limitando però la flessibilità man mano che l’applicazione si “espande”. RethinkDB è progettato per essere accessibile da un server di applicazioni, in modo molto simile a un sistema dbms tradizionale. Ciò ha richiesto, per lo sviluppo del server di sistema, un codice di istallazione leggermente più complicato (vedi in seguito paragrafo Docker) ma ha consentito una maggiore flessibilità nel caso in cui in futuro il sistema diventi più “sofisticato”.

In generale, come per quasi tutti i dbms non relazionali, 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.

ReQL

Per interrogare un server RethinkDb, come già anticipato, viene utilizzato il linguaggio ReQl [[25]](#ReQL) .

Il ReQL è un linguaggio che permette di manipolare i documenti JSON di una tabella presente in un database RethinkDB in maniera efficace riuscendo ad eseguire tutte le interrogazioni implementabili attraverso il linguaggio SQL classico. Permette, infatti, di eseguire operazioni di “join” tra tabelle come mostrato in figura e le funzioni di aggregazione.



Figura 4.1: esempio di query di join con linguaggio SQL e ReQL

SQL e RethinkDB condividono, inoltre, una terminologia molto simile. In figura 4.2 è presente una tabella dei termini e dei concetti dei due sistemi.

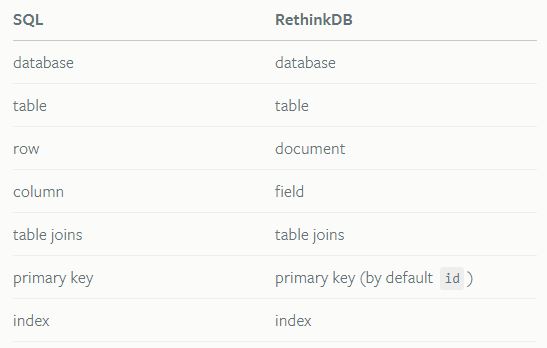


Figura 4.2: similitudine dei termini adottati da SQL e RethinkDB

Il linguaggio ReQL è diverso dagli altri linguaggi di query NoSQL. Le query ReQL vengono “costruite” attraverso chiamate di funzione nel linguaggio di programmazione scelto; nel caso della libreria di sistema UtilityRethink, in particolare, si è utilizzata l’API per applicativi 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 chiamato il comando *run* passandogli l’oggetto che rappresenta la connessione attiva al database. Nella libreria UtilityRethink la connessione al database è rappresentata dal campo *conn* di *ConnectionNodes*.

Docker

Docker Desktop è stato un secondo software fondamentale per la realizzazione e gestione del server di sistema [[26]](#Docker) .

Docker Desktop è un’applicazione per macchine MacOS e Windows per progettare e distribuire applicazioni e micro servizi containerizzati. Consente agli sviluppatori di utilizzare immagini e modelli certificati e i flussi di lavoro di sviluppo sfruttano la piattaforma Docker Hub per estendere ogni ambiente di sviluppo a un repository sicuro per la creazione automatica rapida, l’integrazione continua e la collaborazione sicura. Queste caratteristiche sono state sfruttate per l’implementazione del server che sarà descritta in dettaglio nel capitolo 5.

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 [[27]](#docker_container) .

Siccome per il contesto aziendale il sistema esegue sul sistema operativo Windows è stata utilizzata la versione Docker Desktop for windows [[28]](#docker_for_windows) .

I comandi offerti da Docker Desktop, come sarà analizzato nello specifico inseguito, sono stati fondamentali per l’implementazione e mantenimento del server Rethinkdb containerizzato.

## Software utilizzati per l’implementazione della libreria di sistema Utility Rethink

Utilizzando lo store NuGet di Visual Studio è possibile scaricare diverse librerie di supporto ai propri progetti. Per l’implementazione della libreria di sistema UtilityRethink, sono quindi state scaricate e utilizzate le librerie *Bchavez/ RethinkDb.Driver*, *Reactive Extensions (Rx)* e *Simple Injector*.

*Bchavez/RethinkDb.Driver* è un driver che fornisce, agli applicativi scritti in linguaggio c#, un’api utile per l’interfacciamento al server Rethinkdb [[29]](#Bchavez_RethinkDb_driver). Pur non essendo un driver “ufficiale”, è molto simile a quello ufficiale scritto in java, ed è stato fondamentale per lo sviluppo della libreria di sistema.

Per implementare il notificatore oltre alla funzionalità di *changefeed* offerta dal driver *Bchavez/ RethinkDb.Driver* è stata utilizzata anche la libreria *Reactive Extensions (Rx).* *Reactive Extensions (Rx)* è una libreria per lo sviluppo di programmi asincroni basati su eventi utilizzando il pattern observer tipico della programmazione a oggetti e operatori in stile LINQ [[30]](#Reactive_extensions).

*Simple Injector*, infine, è stata utilizzata, per il contesto aziendale, dagli applicativi sofware di frontend e beckend aziendali che utilizzeranno la libreria per gestire e richiamare le funzionalità dell’ applicativo stesso. 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 [[31]](#simple_injector). Attraverso questo meccanismo offerto dalla libreria *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 stata scelta Scrum come metodologia di progettazione [[32].](#Scrum) Scrum è una metodologia di progettazione agile che 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 riuscire a implementare il server RethinkDB di sistema, come anticipato nel capitolo precedente, 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, è stato scelto di utilizzare l’immagine ufficiale *rethinkdb* mantenuta dalla società RethinkDb [[33]](#immagine_rethinkdb). In seguito al download dell’immagine, scrivendo sul terminale i comandi di Docker-compose, offerti dall’applicativo Docker-Desktop, è stato possibile implementare tre diversi server RethinkDB.

Server a singolo nodo

Il server RethinkDB a singolo nodo, come rappresentato in figura 5.1, è un server che è stato costruito utilizzando un solo container Docker RethinkDB.

L’unico nodo disponibile di questo server, rimane in ascolto sulla porta 8081 e sarà raggiungibile da tutti gli applicativi che utilizzeranno la libreria di sistema UtilityRethink implementata.

Server1NodoTerminale

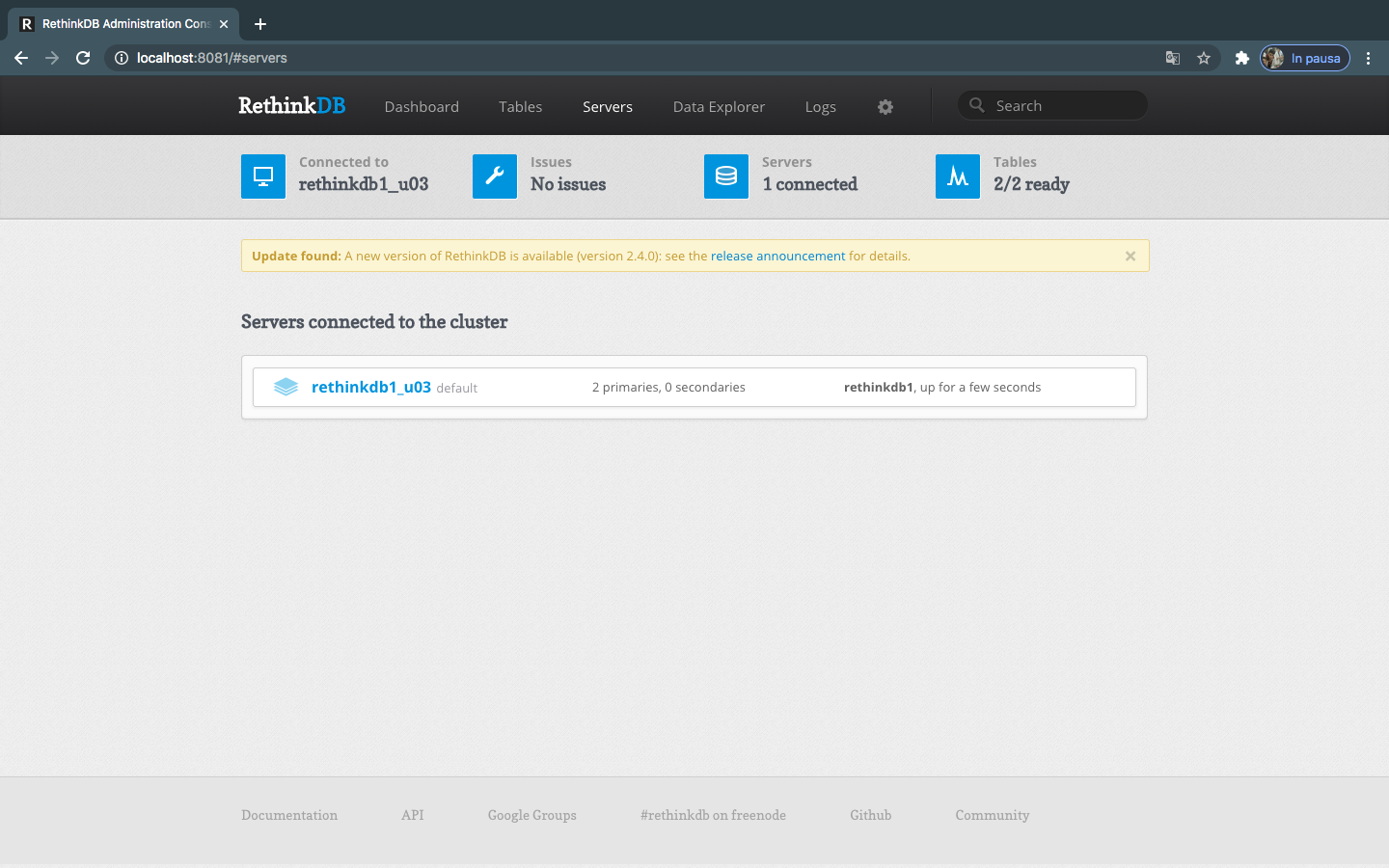


Figura 5.1

Server a due nodi

Il server a due nodi, invece, come mostrato in figura 5.2, è stato implementato attraverso due container Docker.

Utilizzando il sistema di tagging precedentemente discusso, è stato possibile implementare uno “script bash” , chiamato “*start-rethinkdb-cluster.sh*”, che, attraverso alcuni specifici comandi, ha garantito l’unione e la collaborazione dei due container Docker in un unico server RethinkDB.

Questo server, essendo formato da due nodi, è raggiungibile sia sul proprio indirizzo locale sulla porte 8081 e 8082.

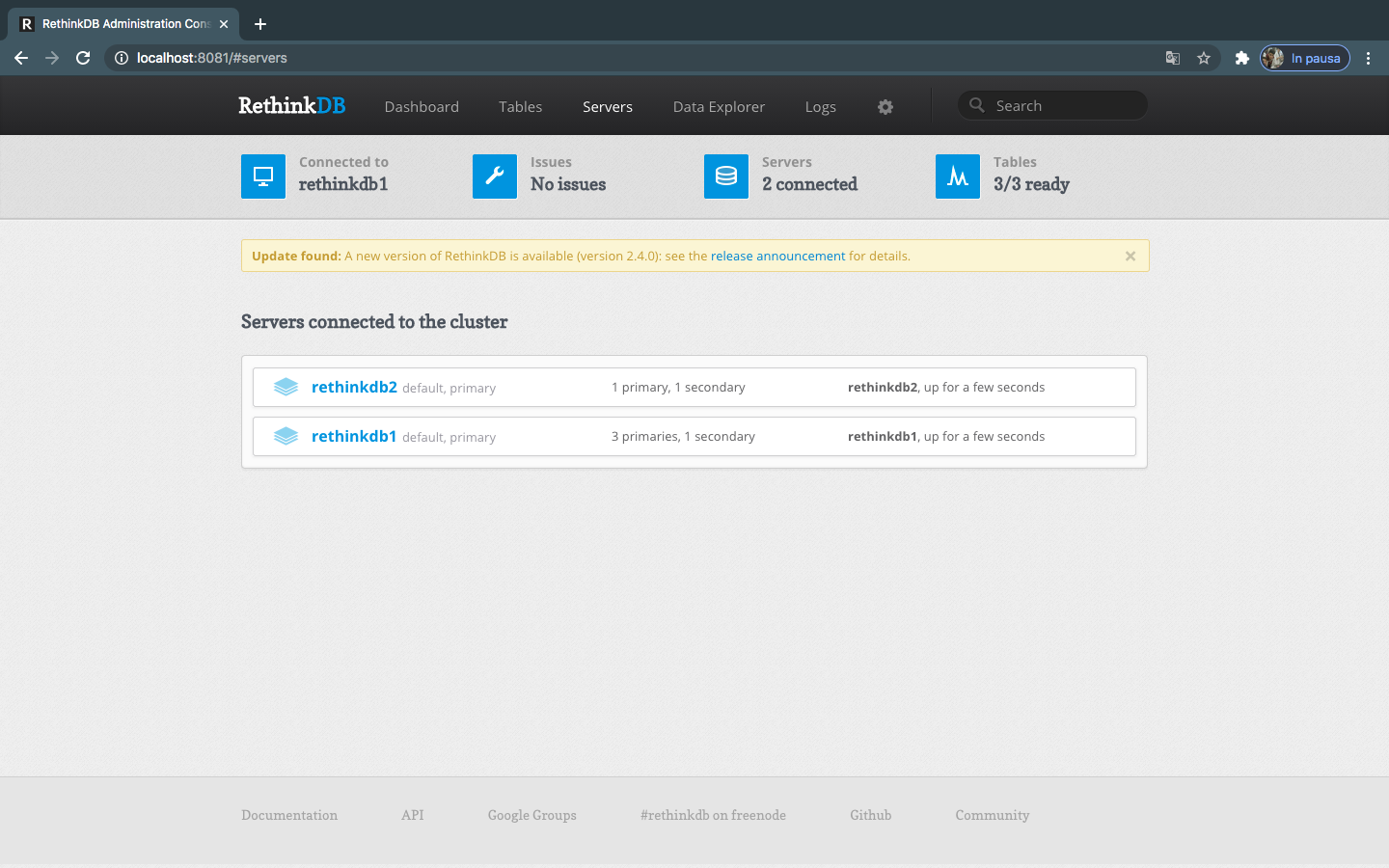


Figura 5.2

Server a cinque nodi

Il server a cinque nodi, infine come descritto in figura 5.3, è stato sviluppato attraverso cinque container Docker.

Utilizzando lo script bash “*start-rethinkdb-cluster.sh*”, come per il server a due nodi, è stato eseguito il join dei cinque nodi in un unico server RethinkDB. Questo server, essendo formato da cinque nodi RethinkDB, sarà raggiungibile sul proprio indirizzo Ip locale sulle porte 8081, 8082, *8083*, 8084 e 8085.

Saranno spiegati, in seguito, i comandi per costruire, eseguire e fermare i server RethinkDB implementati.

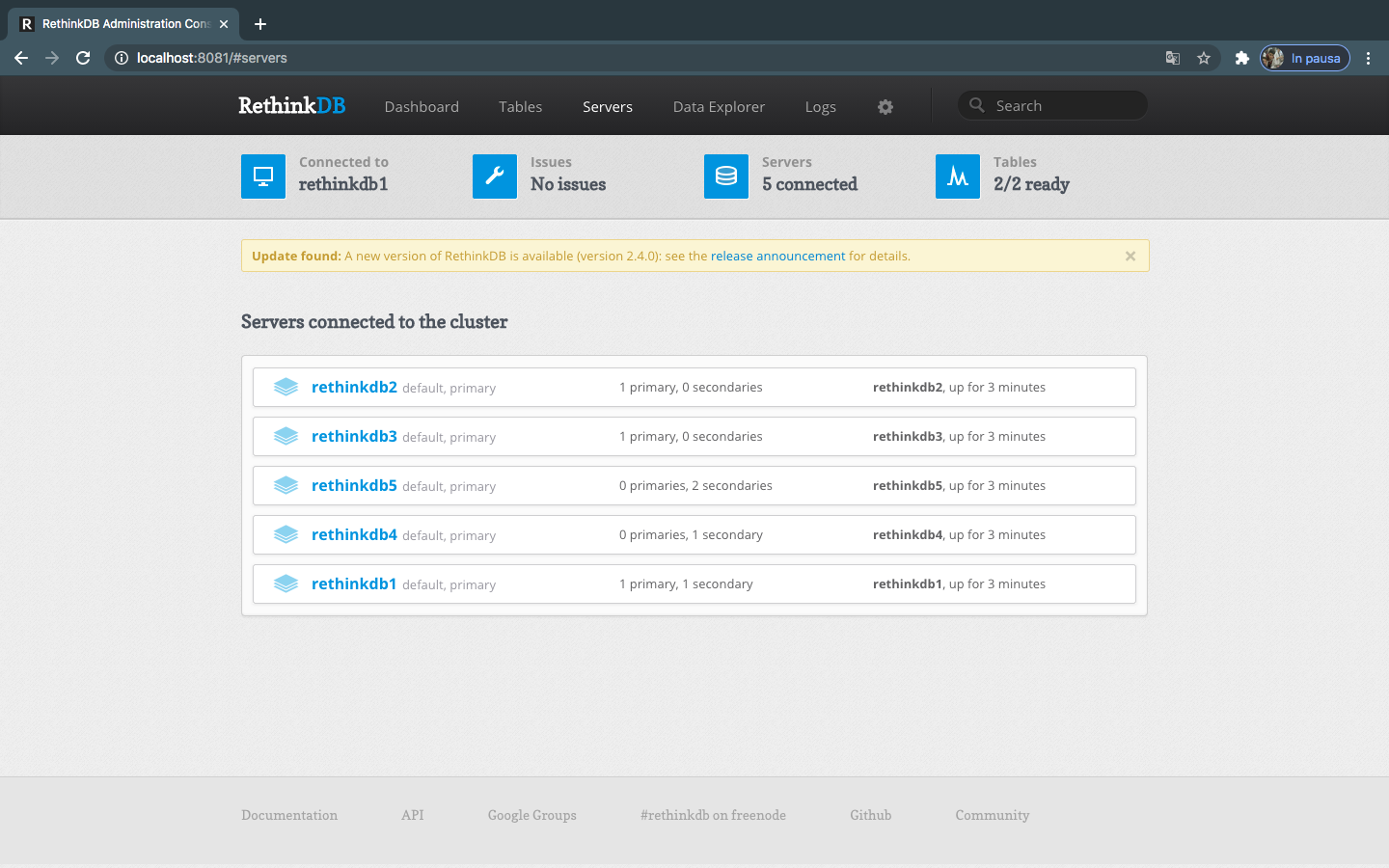
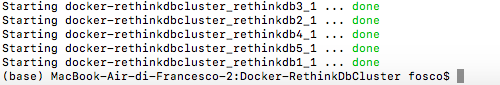


Figura 5.3

Conclusioni sui tre server implementati

Ogni nodo dei tre server RethinkDB, quindi, rappresenta di fatto un container. Nei due casi in cui il server è formato da più nodi, è stato utilizzato un particolare “sistema di tagging” tipico di RethinkDB[[37]](#RethinkDb_tag). Questi *tag* sono definiti dall’utente e servono a identificare i nodi. A ogni nodo in un cluster RethinkDB, come indicato in figura 5.4 è possibile assegnare zero o più *tag*.



Figura 5.4

Nei due server a più nodi, il sistema di tagging è stato fondamentale per identificare i nodi/container RethinkDB. In questo modo, è stato possibile nello script bash *start-rethinkdb-cluster.sh* i comandi necessari ad eseguire il join dei nodi in un unico server [[34]](#RethinkDb_join_cluster).

Grazie a questo particolare script, quindi, i nodi di uno stesso server RethinkDB collaborano tra di loro per riuscire a rispondere alle richieste in arrivo dai servizi client e per sfruttare, in maniera efficace, le politiche di Replication e di Sharding.

In questa applicazione del sistema, bisogna però specificare che il server risiede, al momento, completamente sulla macchina locale perché sono stati utilizzati i comandi di Docker-compose [[35]](#Docker_compose) . In un contesto più articolato è possibile utilizzare altre tecnologie per distribuire i diversi nodi del server su più macchine fisiche. Questa potrebbe essere una probabile modifica futura per l’applicativo e Docker Swarm o Kubernetes sarebbero, ad esempio, alcune tecnologie su cui fare riferimento.

## 

## 

## Comandi per la gestione del server di sistema

Per iniziare a utilizzare uno dei tre server RethinkDB implementati la procedura è la stessa in tutti i casi.

Aprire il terminale e dirigersi sulla cartella del progetto corrispondente all’opzione del server che si vuole scegliere (uno, due o cinque nodi). Digitare, in seguito, sempre sul terminale il comando “*docker-compose -f docker-compose.yml build*” che, come mostrato in figura 5.5, costruirà l’immagine docker del server sul proprio computer. Questo comando è, quindi, necessario eseguirlo solamente al primo utilizzo del tipo di server scelto perché l’immagine costruita viene anche salvata permanentemente sul proprio dispositivo.

(docker build screen terminale)



Figura 5.5: build dell’immagine del server RethinkDB a due nodi

Successivamente, è possibile controllare il numero di immagini salvate sul proprio dispositivo, controllandone anche lo stato (colonna “*status*”), attraverso il comando “*docker ps –a*”.

Dopo aver costruito l’immagine è possibile mettere in esecuzione il server RethinkDB scelto. Sempre attraverso il terminale, scrivere il comando “*docker-compose -f docker-compose.yml up -d*” per eseguire il server. Il server Rethink, come mostrato in figura 5.6, è ora online e i suoi nodi sono in attesa di rispondere alle richieste delle applicazioni client.

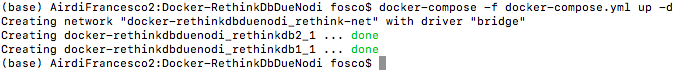


Figura 5.6: up del server RethinkDB a due nodi

Infine, come mostrato in figura 5.8, è possibile “stoppare” il server RethinkDb digitando, sempre sul terminale, il comando “*docker-compose -f docker-compose.yml stop*”.

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

Se, ad esempio, sono inseriti dei nuovi dati sul database e, inseguito, il server viene fatto fermare, al prossimi comando di “up” il server mantiene i dati che sono stati precedentemente caricati.

C:\Users\Utente\Desktop\francesco\Università\Tesi\rethinkdb\fotoserver\Docker-stop.png

Figura 5.7: stop del server RethinkDB a due nodi.

## Sharding e Replication

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

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 e garantisce la lettura dei dati in “parallelo” da diversi utenti siccome sono presenti repliche su più nodi del Cluster.

Utilizzando le tecniche di 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 [[36]](#rethinkdb_Sharding_replication) .

### Gestione con Interfaccia Web

Le impostazioni del server RethinkDB possono essere controllate facilmente dalla console di amministrazione web.  Quando si utilizza l'interfaccia utente Web, per gestire la Replication e lo Sharding dei dati, è sufficiente specificare il numero di frammenti desiderati e il numero di repliche dei dati richieste. In base alle proprie disponibilità, sarà poi RethinkDB stesso a determinare i propri migliori punti di divisione per mantenere i frammenti bilanciati.

Per replicare e dividere i propri dati:

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

Fare clic sul pulsante *Reconfigure* mostrato in figura 5.8 .

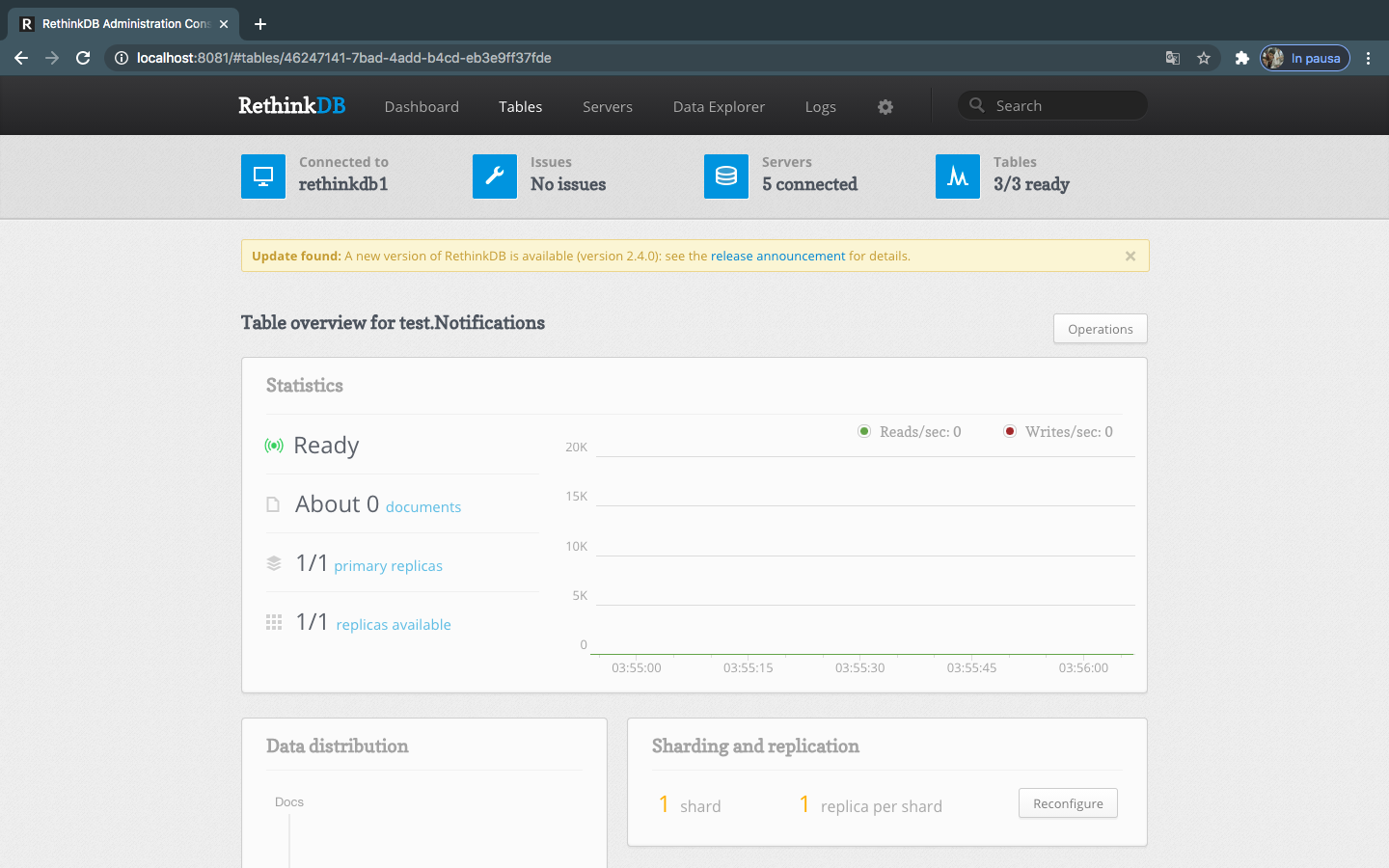


Figura 5.8: riconfigurazione dei dati della tabella di sistema Notifications

A questo punto, come mostrato in figura 5.9, è possibile impostare il numero di frammenti e repliche che si desiderano per i propri dati. Nell’esempio mostrato in figura, in particolare, si analizza il caso in cui si vogliano riconfigurare i parametri di Replication e Sharding sui dati della tabella di sistema *Notifications* presenti sul server a cinque nodi che

è stato implementato. Il limite massimo, come indicato, che è possibile impostare che numero di repliche e shard dei dati è cinque perché il server, in questo caso, è formato da cinque nodi.

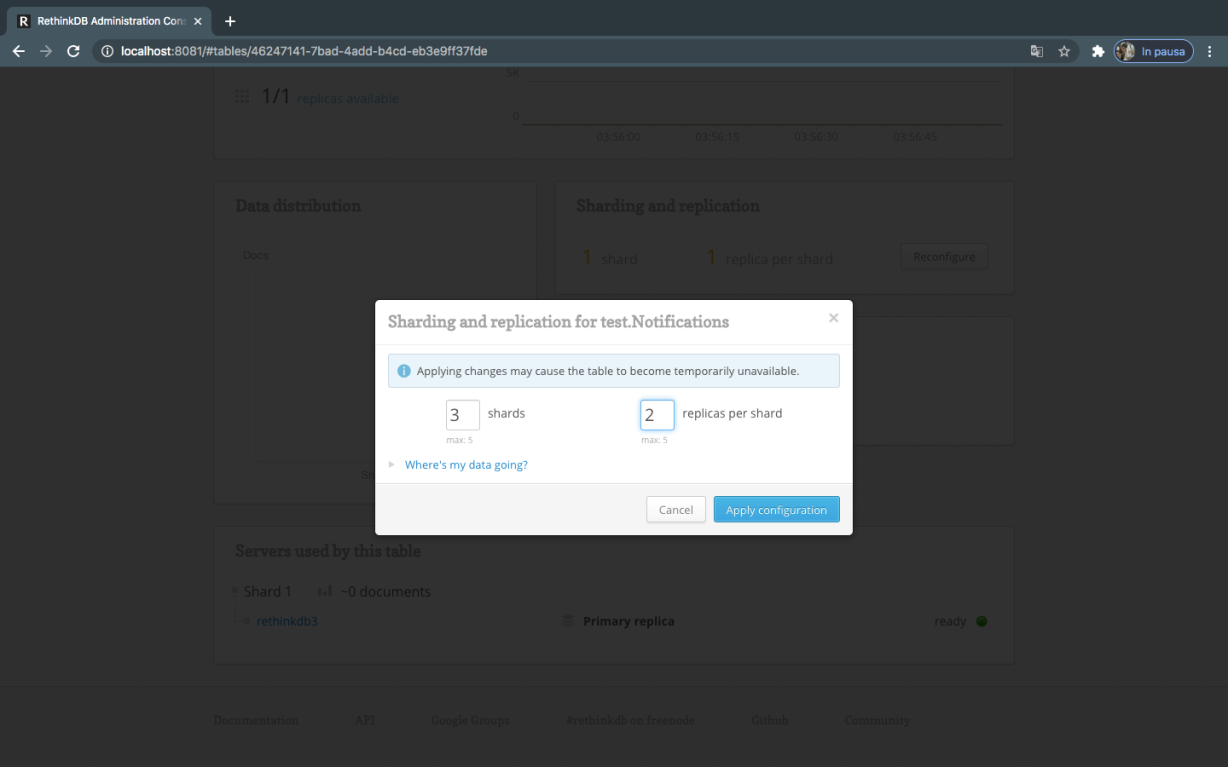


Figura 5.9

In seguito al “clic” sul pulsante “*Apply configuration”* i dati della tabella di sistema Notifications sono ridistribuiti come indicato nella figura 5.11

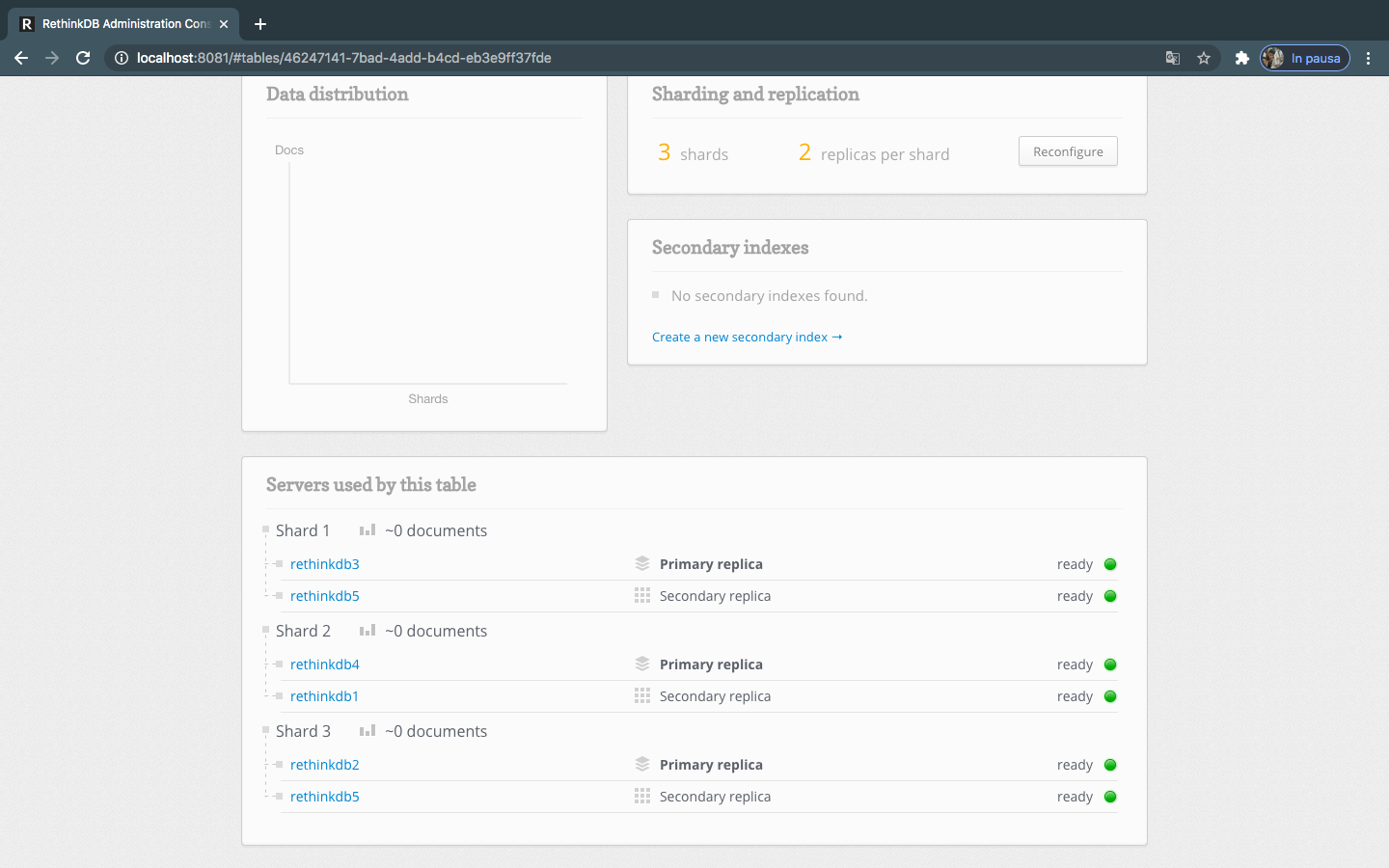


Figura 5.10

### 

### Gestione con ReQL

Il linguaggio di interrogazione ReQL, tipico di RethinkDB, garantisce, oltre alle funzionalità di eseguire operazioni CRUD (*insert*, *delete* e *update*) sui dati e la funzionalità *changefeed* precedentemente discussa, un controllo dettagliato sul numero di repliche e la politica di Sharding adottata.

Se per l’implementazione del sistema fosse stato richiesto di gestire delle configurazioni più avanzate come, ad esempio, saper decidere su quale specifico nodo del cluster inserire una delle repliche di un dato, allora, sarebbe stato necessario rivolgersi a comandi specifici offerti dal linguaggio ReQL. Quando si utilizza l’interfaccia grafica, infatti, i migliori punti di divisione dei dati sono autodeterminati da RethinkDB stesso.

Il sistema di tagging precedentemente discusso, può anche essere utilizzato, oltre per la costruzione di un server RethinkDB a più nodi, per configurare le opzioni di Sharding e Replication. Nelle configurazioni di una tabella, in particolare, i tag permettono di archiviare su uno specifico nodo di interesse del cluster le repliche di un dato. Grazie a questo “sistema di tag”, quindi, è possibile avere un migliore controllo su dove saranno salvate le repliche dei dati di una tabella. Il comando ReQL a cui si fa riferimento è reconfigure.

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



Figura 5.11

### Gestione nella libreria UtilityRethink

Anche attraverso la libreria di sistema UtilityRethink implementata è possibile gestire la Replication e lo Sharding dei dati di una tabella.

Come indicato in figura , attraverso una funzionalità offerta dal *DbManager*, vedi capitolo 3 sul design della libreria, l’utente è in grado di aggiornare lo sharding e il numero di repliche dei dati di una tabella appartenente a un server formato da più nodi.

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



Figura 5.12

**Capitolo 6**

# Bibliografia

RethinkDb, the open source database for the realtime web. URL <https://rethinkdb.com/>.

RethinkDb domande frequenti. URL <https://rethinkdb.com/faq>.

RethinkDB vs API di sincronizzazione in tempo reale (es Firebase) <https://rethinkdb.com/faq>

Linguaggio ReQL. URL <https://rethinkdb.com/docs/introduction-to-reql/>

Docker Desktop. URL <https://www.docker.com/products/docker-desktop>

I Docker container e le immagini. URL <https://www.docker.com/resources/what-container>

Docker Desktop for windows. URL <https://docs.docker.com/docker-for-windows/>

Bchavez/RethinkDb.driver. URL <https://github.com/bchavez/RethinkDb.Driver>

Reactive extensions. URL <http://reactivex.io/>

Simple injector. URL <https://simpleinjector.org/>

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>.

Immagine RethinkDB utilizzata. URL <https://hub.docker.com/_/rethinkdb>

How to start a RethinkDB server. URL <https://rethinkdb.com/docs/start-a-server/>

Docker compose. URL <https://docs.docker.com/compose/>

Sharding e Replication su RethinkDB. URL <https://rethinkdb.com/docs/sharding-and-replication/>

Server tag di RethinkDB. URL <https://rethinkdb.com/docs/sharding-and-replication/#server-tags>