Actorbase

NoSQL database

Le capacità di calcolo richieste dalle applicazioni attuali rendono spesso obsoleto e inadeguato il modello di gestione dei dati relazionale. I database (DB) relazionali, infatti, devono – per essere tali -- garantire il concetto logico di transazione, ossia soddisfare le quattro proprietà note come ACID:

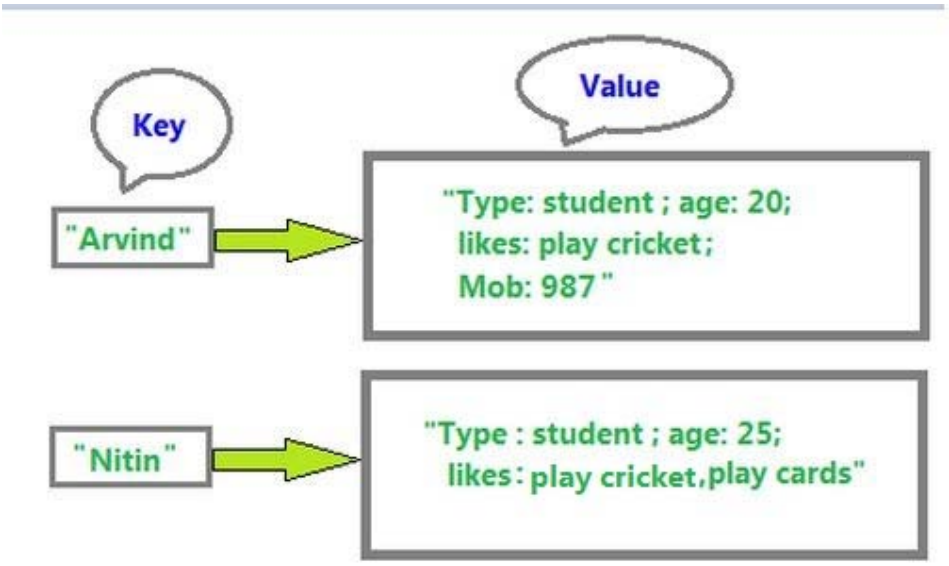
- **Atomicità**: la transazione è indivisibile nella sua esecuzione e la sua esecuzione deve essere o totale o nulla; non sono ammesse esecuzioni parziali;
- **Coerenza**: quando inizia una transazione il DB si trova in uno stato coerente e quando la transazione termina il DB deve essere in un altro stato coerente, che non violi eventuali vincoli di integrità, privo quindi di contraddizioni (*inconsistenze*) tra i dati;
- **Isolamento**: ogni transazione deve essere eseguita in modo isolato e indipendente dalle altre transazioni, l'eventuale fallimento di una transazione non deve interferire con le altre transazioni in esecuzione;
- **Durabilità**: detta anche **persistenza**, si riferisce al fatto che una volta che una transazione abbia richiesto un *commit work*, i cambiamenti apportati non dovranno essere più persi. Per evitare che nel lasso di tempo fra il momento in cui il DB si impegna a scrivere le modifiche e quello in cui le scriva effettivamente si verifichino perdite di dati dovuti a malfunzionamenti, vengono tenuti dei registri di log dove sono annotate tutte le operazioni sul DB.

Il soddisfacimento di queste proprietà, e in termini più generici, il concetto di transazione, è incompatibile con l'architettura distribuita richiesta per l'elaborazione di grandi moli di dati.

Dal 2009, sempre più insistentemente si stanno facendo strada una serie di modelli non relazionali, definiti NoSQL. L'acronimo NoSQL sta per "*not only SQL*", sottolineando il fatto che esistono diversi casi d'uso per i quali il modello relazionale rappresenta una forzatura, ma tanti altri per i quali tale modello è ancora la soluzione migliore. I DB NoSQL si possono suddividere sulla base dello schema dati utilizzato, per esempio:

- Document oriented, il cui rappresentante più illustre è MongoDB (https://www.mongodb.org/)
- Graph databases, il cui rappresentante più famoso è Neo4j (http://neo4j.com/)
- Column oriented, come ad esempio Apache HBase (http://hbase.apache.org/) e Apachae Cassandra (http://cassandra.apache.org/)
- Key-value map, come ad esempio Amazon Dynamo (http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html)

In particolare, nei database di tipo Key-Value (KV), i dati vengono gestiti utilizzando array associativi (mappe o dizionari). Ad ogni chiave differente possono essere associati dati strutturati o destrutturati. Sarà il sistema richiedente ad interpretare le informazioni fornite dal database.



Una caratteristica importante dei DB non relazionali è l'essere particolarmente portati allo *scaling* orizzontale, noto anche come *scale out*[1]. Esso si ottiene aggiungendo trasparentemente un nuovo nodo ad un sistema distribuito. L'elaborazione complessiva del sistema, quindi, viene suddivisa fra i nodi che lo compongono, permettendo migliore parallelismo ed migliore resistenza a malfunzionamenti.

Lo scaling orizzontale richiede capacità di gestione dei dati in modo distribuito fra più nodi. Uno dei modelli che risolve con eleganza alcuni dei problemi più comuni della programmazione concorrente distribuita è il modello ad attori.

Actor model

Il modello ad attori è un modello matematico di esecuzione concorrente di un programma nel quale le primitive di elaborazione concorrente sono individuate negli "attori". Dalla definizione fornita da John Mitchell:

"Each actor is a form of reactive object, executing some computation in response to a message and sending out a reply when the computation is done"

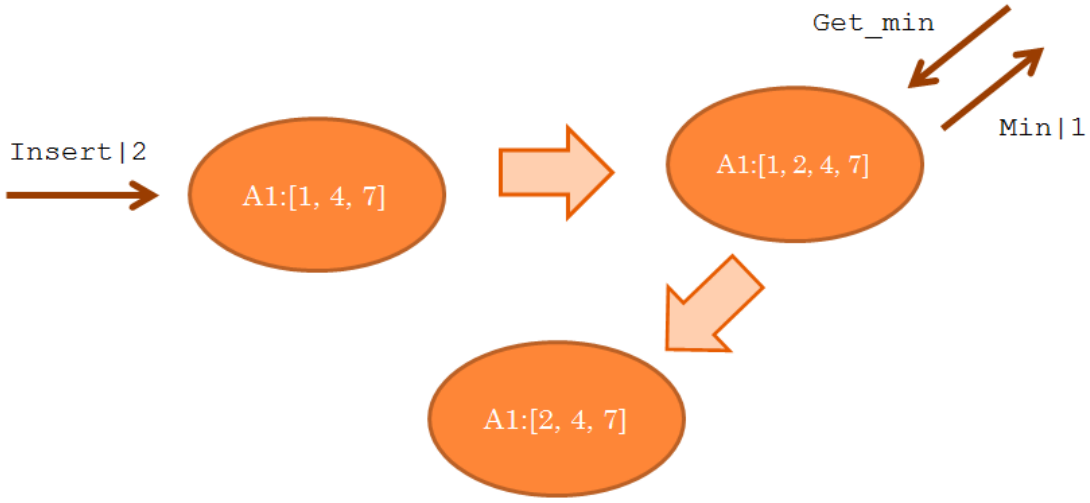
Questi oggetti possono ricevere messaggi (contenenti richieste), in risposta ai quali un attore può prendere decisioni, creare ulteriori attori, inviare messaggi ad altri attori e modificare il proprio stato per determinare il prossimo messaggio a cui rispondere.

I messaggi rappresentano quindi l'unica interfaccia di comunicazione fra gli attori, con ovvi benefici in termini di trasparenza e quindi di scalabilità. Ogni messaggio è composto di tre parti, ossia

1. Identificativo dell'operazione richiesta (*tag)*
2. Identificativo dell'attore al quale spedire il messaggio (*target)*
3. Informazioni del messaggio (*data*).

[1] Cf. p.es. http://goo.gl/VuzoTH

I messaggi ricevuti da un attore vengono depositati nella sua *inbox*. L'attore, elaborerà i messaggi in sequenza, uno ad uno, in modo tale da isolare la propria elaborazione ed evitare l'insorgere di *race condition*.



Ad ogni cambio di stato viene generato virtualmente un nuovo attore, che modifica la propria interfaccia esterna sulla base delle operazioni precedentemente elaborate.

Gli attori possono trovarsi fisicamente su nodi differenti della rete, permettendo in questo modo di distribuire l'elaborazione in modo nativo.

Actor model + NoSQL database = Actorbase

Si immagini quindi di implementare un modello di database NoSQL di tipo Key-value utilizzando il modello ad attori. Ogni mappa può contenere un numero arbitrario di coppie chiave / valore. Ad ogni mappa è possibile attribuire un nome che la contraddistingua. Ogni mappa è implementata da uno o più attori, che mantengono fisicamente al proprio interno le coppie chiave / valore. Il numero di attori utilizzato è dipendente da un parametro di configurazione della mappa, che specifica ogni quante coppie, la mappa debba essere suddivisa fra più attori. Definiamo questo tipo di attori come STOREKEEPER.

Ogni attore di questo tipo può avere uno o più attori "ombra", che definiamo NINJA. Questi, localizzati su nodi differenti dai rispettivi STOREKEEPER ed in continuo aggiornamento con essi, permettono di mantenere consistente il database anche in caso di caduta di uno STOREKEEPER. In questo caso, uno dei NINJA verrà eletto nuovo STOREKEEPER ed il sistema continuerà a funzionare (cf. p.es. https://en.wikipedia.org/wiki/Leader_election).

Un altro tipo di attori, detti STOREFINDER si occupano di ricevere le richieste dall'esterno e instradarle ai rispettivi STOREKEEPER, in modo da soddisfarle. Anche in questo caso, il numero di STOREFINDER è variabile e può essere configurato tramite una proprietà dedicata. Virtualmente, uno STOREFINDER definisce un indice sulla chiave della mappa: più STOREFINDER sono presenti e meno messaggi dovranno essere utilizzati per recuperare l'informazione richiesta. Si definisce uno STOREFINDER principale (MAIN), che funge da punto di accesso al database. Infine, è presente una quarta tipologia di attori, i WAREHOUSEMEN, che si interfacciano con gli STOREKEEPER e persistono su disco le rispettive mappe.

Esiste un ultimo tipo di attori, MANAGER, che ricevono le richieste di gestione degli attori di tipo STOREKEEPER. In particolare, questi attori sono responsabili della gestione del numero massimo di coppie chiave / valore contenute in un'istanza di STOREKEEPER.

Figura 1 mostra uno schema logico degli attori descritti e delle interazioni che sono definite tra di essi.

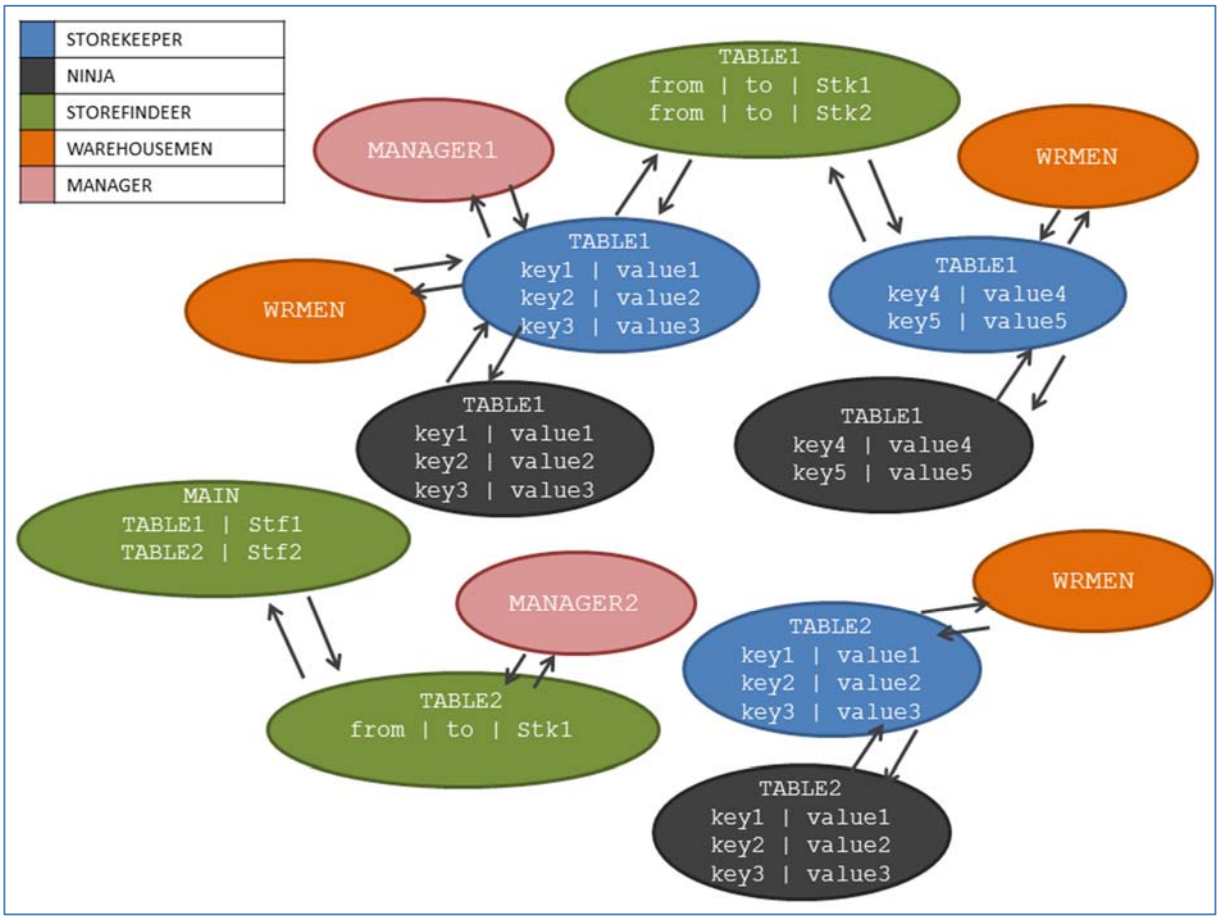


Figura 1: Schema logico interazione attori.

Requisiti obbligatori

Si richiede pertanto di implementare il suddetto sistema ad attori che prende il nome di ACTORDB. In particolare, per l'implementazione degli attori si richiede l'utilizzo della libreria Akka (http://akka.io/). Akka fornisce un'implementazione del modello ad attori su JVM. Il linguaggio da utilizzare può essere scelto fra Scala (http://www.scala-lang.org/, preferito) o Java. In particolare è richiesta lo sviluppo del seguente insieme di attori:

1. STOREKEEPER
2. STOREFINDER
3. WAREHOUSEMEN

Le operazioni da implementare sul database sono le seguenti:

1. Inserimento
2. Cancellazione
3. Aggiornamento (caso particolare di un inserimento con chiave già presente)

Si richiede inoltre la definizione di un *domain specific language* (DSL) da utilizzare da riga di comando per poter interagire con il database.

Si richiede inoltra la pubblicazione del progetto su GitHub e l'utilizzo delle *issue* per la segnalazione di eventuali *bug*.

Requisiti Opzionali

Tra i requisiti opzionali si richiede l'implementazione dei restanti tipi di attori. Inoltre, si richiede l'implementazione di un driver Java o Scala per l'interfacciamento al database direttamente da programmi scritti nei suddetti linguaggi.

Variazione dei requisiti

Non sono ammesse variazioni se non a evidente miglioramento di quanto qui specificato. Non è esclusa la comunicazione, da parte del committente, di variazioni ai requisiti sia precedentemente alla consegna delle offerte che durante la realizzazione del sistema.

Documentazione

La consegna del sistema dovrà essere accompagnata dai necessari manuali d'uso per gli utilizzatori del framework e da ogni altra documentazione tecnica necessaria per l'utilizzo del prodotto da parte del personale operatore del committente. È richiesta la versione inglese.

Garanzia e manutenzione

Il fornitore dovrà dimostrare in sede di collaudo (Revisione di Accettazione) il funzionamento corretto del sistema. L'eliminazione dei difetti e delle non conformità eventualmente emersi in sede di collaudo sono a totale carico del fornitore.

Rinvio

Il proponente Riccardo Cardin è interessato a questo progetto come dimostrazione della fattibilità dell'obiettivo utilizzando le tecnologie indicate. Il progetto verrà distribuito con licenza MIT con copyright condiviso tra gli studenti e il proponente. Per tutto quanto non previsto nel presente capitolato, sono applicabili le disposizioni contenute nelle legge e nei collegati per la gestione degli appalti pubblici.

Proponente

Il proponente Riccardo Cardin si impegna a seguire il fornitore durante tutte le fasi del progetto ed a proporre un piano successivo di sviluppo del prodotto nel caso in cui i risultati ottenuti e le prospettive lo consentano.