Caratteristiche principali del modello NoSQL *key-value*

Simone Richetti, mat. 129180, attività 3

Abstract

Con l'esplosione dei *Big Data*, diverse tipologie di applicazioni necessitano di performance e scalabilità più alte di quanto i tradizionali database relazionali possano garantire. In questo contesto si sono sviluppati i database *key-value*. Questi modelli sono caratterizzati dalla semplicità nella gestione e nell'accesso dei dati, la quale garantisce velocità nelle operazioni su essi, dalla scalabilità del sistema, la quale permette di ampliarlo con minimo impatto, e dall'alta disponibilità dei dati anche in condizioni critiche dei sistemi, necessaria per tutte quelle applicazioni in cui è importante la *user experience* dei fruitori. Queste caratteristiche sono ottenute al prezzo di limiti su dimensioni e tipologie di dati trattabili e sulle operazioni che si possono eseguire su essi, oltre che ad una complessa gestione della consistenza dei dati in caso di *failures* della rete o dei sistemi.

1 Introduzione

Per molti anni i database relazionali sono stati la principale soluzione utilizzata dalla maggior parte dei sistemi per il salvataggio persistente dei dati e per la loro gestione.

Con la diffusione di servizi *web-based* e le relativa gestione di grandi moli di dati, questo tipo di soluzioni ha iniziato ad evidenziare dei limiti strutturali. La difficoltà di questi sistemi a scalare con la quantità di dati in termini di performance nelle operazioni svolte hanno portato alla ricerca di soluzioni alternative per lo *storage* dei dati.

In questo contesto si sono diffusi i primi database NoSQL e in particolare quelli di tipologia *key-value*, che cercano di soddisfare questa richiesta di alte performance con la semplicità nell'accesso ai dati, un'alta scalabilità e con la distribuzione dei dati su diversi *data storage*.

In questa relazione vengono approfondite le caratteristiche principali di questo modello NoSQL, facendo riferimento ad alcune sue implementazioni, in particolare a DynamoDB di Amazon. Dynamo è stato il primo tra i sistemi che implementano questo modello e ha introdotto le caratteristiche e le tecniche che sono state poi riprese dalle successive implementazioni.

2 Caratteristiche

Approfondiamo ora le principali caratteristiche dei database che implementano il modello *key-value*: ci si concentrerà principalmente sulla rappresentazione e gestione dei dati e sulle tecniche utilizzate per garantire scalabilità e disponibilità dei dati, facendo riferimento alle caratteristiche di *consistency*, *availability* e *partition-tolerance*.

2.1 Gestione dei dati

Mentre i database relazionali salvano i dati sotto forma di tabella con uno schema ben preciso, il modello *key-value* salva i dati in una struttura simile ad un dizionario o ad una *hash map*:

- Il database è diviso in tabelle;
- Ogni tabella contiene una serie di *record* o *item*, che rappresentano i singoli oggetti;
- Ogni *record* è definito come un'insieme di *field*, ovvero coppie chiave-valore;
- Ogni *record* è identificato univocamente da una *partition key* che viene utilizzata per accedere a quell'elemento, ma anche per determinare in quale partizione esso verrà salvato, basandosi sul valore dell'*hash* della chiave.

Nei sistemi più semplici, come ad esempio in Aerospike, le chiavi sono stringhe, mentre i valori sono salvati come *blob*, ovvero come vettori di byte senza uno specifico tipo associato. Ci sono poi sistemi che invece supportano una tipizzazione dei dati, chi più semplice e chi invece, come ad esempio DynamoDB, permettendo anche strutture dati più complesse come dati geografici, collezioni di oggetti o campi innestati.

Molti sistemi che implementano il modello chiave-valore fanno utilizzo anche di una chiave aggiuntiva, detta *sort key*, la quale viene utilizzata per ordinare i *record* all'interno della partizione.

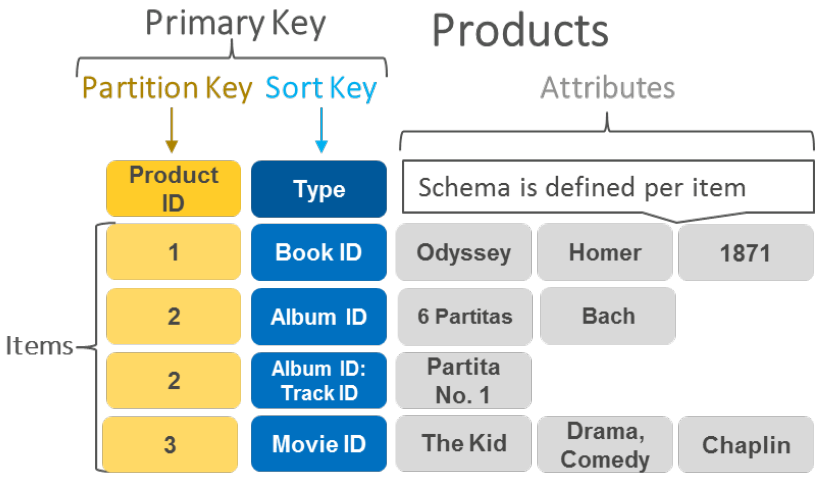

Figura 1: Struttura chiave-valore dei dati

È importante notare come, fatta eccezione per la *partition key* ed eventualmente la *sorting key*, non esista uno schema fisso per i *record*. Ogni oggetto ha uno schema a sé stante, ovvero diversi oggetti possono presentare diversi campi. Questo permette un grande risparmio di memoria, poiché non si devono salvare *placeholder* o occupare memoria per campi che un oggetto non utilizza, al contrario di ciò che avviene nei database relazionali, e permette anche una minore complessità dal punto di vista della gestione dei dati e della velocità delle operazioni su di essi.

Questa flessibilità nello schema, però, rende limitato il numero ed il tipo di operazioni che si possono svolgere su questi dati. Per maggiori dettagli riguardo a questo si veda la sezione 2.2.

L'hardware utilizzato per lo *storage* dei dati varia a seconda delle necessità: sistemi come DynamoDB che hanno bisogno di persistenza utilizzano *data storage* distribuiti basati su Dischi a Stato Solido (SSD), mentre ci sono altri sistemi chiamati *in-memory*, come Memcached o Redis, che utilizzano principalmente la memoria DRAM per un accesso ad altissime prestazioni ai dati ed effettuano un salvataggio asincrono e periodico dei dati su dischi persistenti. L'utilizzo di una memoria volatile, però, genera il rischio di una perdita di dati nel caso di guasti e *failures* del sistema o della rete, quindi è una soluzione utilizzata solo in sistemi in cui questa perdita è un rischio tollerabile.

2.2 Operazioni sui dati

In generale, i database chiave-valore non hanno un linguaggio di *query*, ma utilizzano delle semplici primitive per leggere e scrivere i dati: le funzioni `get(key)` e `put(key, object)`. La `get` permette di ottenere un intero oggetto indicando la chiave che lo identifica, la `put` permette di aggiungere e/o aggiornare un record. Alcuni database prevedono anche una primitiva aggiuntiva `delete(key)` per rimuovere un oggetto, identificandolo con la sua chiave.

Da un lato, è proprio la semplicità di questo modello che rende i database *key-value* veloci, scalabili, flessibili e semplici da utilizzare. I *record* sono acceduti interamente con una richiesta diretta alla memoria. Non ci sono operazioni che riguardano molteplici oggetti, non c'è uno schema da cui recuperare dati, non c'è un *query language* da tradurre.

D'altra parte, la semplicità del modello limita fortemente le tipologie di applicazioni che possono sfruttarlo. Un database di questo tipo permette solo l'accesso a singoli oggetti mediante la loro chiave: non ci sono *join*, aggregazioni o proiezioni come in un database relazionale. Se si vogliono implementare operazioni di *filtering* o di costruzione di una determinata *view* sui dati, questo deve essere fatto a livello applicativo, aggiungendo un *overhead* computazionale alle prestazioni del sistema.

2.3 *High-availability* e scalabilità

Per il tipo di servizi che devono supportare, i sistemi che implementano un modello *key-value* devono garantire le seguenti caratteristiche chiave:

- Scalabilità, per poter permettere una continua crescita;
- *High-availability*, ovvero le operazioni di *read* e *write* devono andare sempre a buon fine;
- Resistenza a fallimenti senza impattare *availability* e performance.

In sistemi i cui dati sono distribuiti su più nodi, il *CAP theorem* evidenza come in condizioni critiche non sia possibile garantire contemporaneamente una forte consistenza dei dati e la loro immediata disponibilità. Molti sistemi tradizionali sacrificano la disponibilità dei dati per la loro consistenza, ad esempio non fornendo la risposa ad una richiesta fino a quando non si è certi della sua correttezza. Al contrario, il modello chiave-valore ha come target sistemi che prediligono scalabilità e disponibilità dei dati, sacrificando la consistenza sotto alcuni scenari di fallimento, ovvero sistemi in cui è

fondamentale fornire una risposta ad ogni richiesta, a costo del fornire una risposta che non è garantita essere la più aggiornata e corretta possibile.

Oltre a queste caratteristiche fondamentali, possiamo trovare implementati in questi sistemi i seguenti principi chiave:

Simmetria ogni nodo ha le stesse responsabilità degli altri, il sistema è distribuito senza nodi centrali con funzioni di comando o controllo;

Decentralizzazione estensione della simmetria, il design del sistema deve favorire tecniche di comunicazione di tipo *peer-to-peer* piuttosto che uno schema di comando centralizzato. Questo permette di avere un sistema più scalabile e non crea *single points of failure*;

Eterogeneità il sistema deve sfruttare l'eterogeneità della struttura sottostante, ad esempio bilanciare il *workload* sui singoli nodi in base alle capacità del singolo hardware.

Vediamo ora alcuni soluzioni di design che garantiscono le caratteristiche di scalabilità e disponibilità dei dati. I dettagli di design riportati nelle sezioni seguenti si riferiscono ad Amazon DynamoDB: potrebbero esserci quindi altri database che differiscono in tecniche utilizzate o caratteristiche implementate, si può però affermare che i principi che stanno alla base di esse accomunano la grande maggioranza dei sistemi che implementano un modello *key-value*.

2.3.1 *Partitioning*: scalabilità

I database chiave-valore implementano soluzioni di *partitioning* per assicurarsi scalabilità. I dati sono salvati in diversi nodi: questo permette di aggiungere e rimuovere facilmente nodi alla rete senza impattare l'intero sistema. Per nodo intendiamo ogni singola unità di *storage* dei dati.

La distribuzione dei dati su diversi nodi si basa sull'*hash* della *primary key* di ogni record. Possiamo rappresentare l'architettura sottostante come un *ring* di nodi, ad ogni nodo è associato un valore casuale che rappresenta la sua posizione nell'anello e quel nodo è responsabile di tutte le posizioni che vanno dalla posizione del suo predecessore alla sua posizione. La funzione di *hashing* è creata in modo che il suo codominio equivalga allo spazio dei valori delle posizioni dei nodi, perciò il valore dell'*hash* della *primary key* di un *record* indica quale nodo sia "responsabile" di esso. Questo nodo è chiamato *coordinator node*.

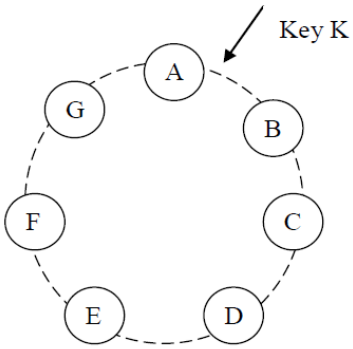

Figura 2: Data partitioning

Il vantaggio principale di questa tecnica, detta *consistent hashing*, è dato dal fatto che l'aggiunta o la rimozione di un nodo impatta solo i nodi circostanti e non l'intero sistema, rendendo facile modificare la dimensione della rete di nodi, risultando in un sistema molto scalabile.

2.3.2 *Data replication*: *availability*

Per ottenere un'alta *availability*, i sistemi chiave-valore replicano i dati su molteplici nodi: ogni record è affidato ad un *coordinator node* che lo salva nel proprio storage e che si occupa di salvarne una copia sugli N nodi successivi del *ring*. La lista di nodi che contiene una determinata chiave è chiamata *preference list*.

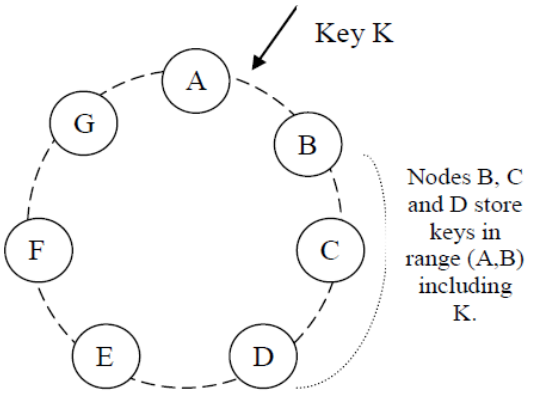

Figura 3: Data replication

Il vantaggio della *data replication* è una maggiore resistenza ai fallimenti della rete o dei singoli sistemi: se un nodo è irraggiungibile, posso recuperare i dati salvati su di esso da altri nodi. D'altra parte però, questo crea un problema dal punto di vista della consistenza: in certi istanti alcuni nodi che mantengono queste copie potrebbero essere irraggiungibili da eventuali aggiornamenti, rischiando così di avere delle copie conflittuali dello stesso dato. Un modo per gestire questi conflitti può essere l'utilizzo di *data versioning*.

2.3.3 *Data versioning*: consistenza

Abbiamo visto come, seguendo le logiche descritte dal *CAP theorem*, i sistemi *key-value* prioritizzino assolutamente disponibilità e scalabilità, accettando in situazioni critiche problemi di consistenza tra le diverse copie dei dati. Vediamo quindi come viene gestita la consistenza dei dati in situazioni ordinarie e come invece viene gestita la presenza di copie conflittuali dello stesso dato.

Database come DynamoDB e Cassandra garantiscono una *eventual consistency*: in condizioni operative normali, le modifiche ad una copia sono propagate in maniera asincrona a tutte le repliche entro un certo lasso di tempo. Tuttavia, in condizioni di sovraccarico o guasti nella rete, le modifiche potrebbero non raggiungere tutte le copie per un tempo prolungato, generando la presenza di copie diverse di uno stesso dato per tutto il lasso di tempo.

Per gestire questa situazione, ogni modifica di un oggetto viene trattata come una sua nuova *versione* e il sistema consente la convivenza di diverse versioni di uno stesso oggetto al suo interno. In condizioni normali, il sistema è in grado di riconoscere una versione come aggiornamento della precedente e quindi di portare tutte le copie alla versione più recente, ma in caso di *failures* della rete combinate con modifiche concorrenti dello stesso oggetto potrebbero crearsi diverse versioni con *histories* di modifiche diverse e non conciliabili tra loro. In questo caso, il sistema non può risolvere il conflitto e, in caso di *read* sul dato, ritorna tutte le versioni differenti al client in modo che questo performi il *merging* dei diversi *branch* di modifiche in uno unico.

Per gestire la risoluzione di conflitti di versione, DynamoDB utilizza la tecnica dei *vector clocks*. Un *vector clock* è una lista di coppie $(nodo, contatore)$: ad ogni versione di un oggetto è associato un *vector clock* che contiene quali nodi hanno modificato quel record e quante volte. Quando un *vector clock* è incluso in un altro, ovvero è stato modificato dagli stessi nodi con tutti i contatori minori o uguali a quelli dell'altro, si può supporre che il secondo sia la versione più aggiornata del primo e di conseguenza si aggiornano le copie che riportano una versione precedente. Altrimenti, se i due *vector clocks* non

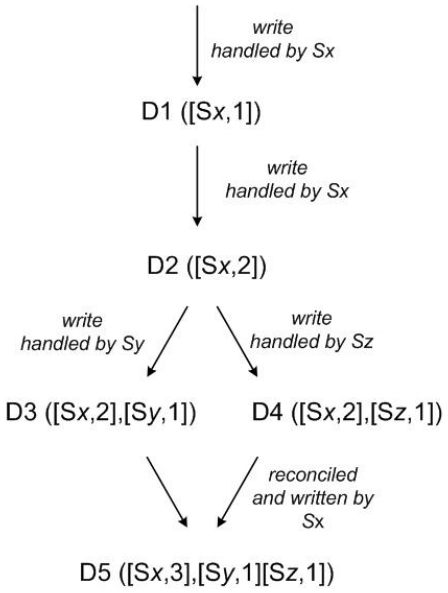

Figura 4: Evoluzione della versione e del vector clock di un oggetto

sono associabili con nessun rapporto di parentela, allora le copie sono in conflitto: se il client effettua una *read* in presenza di copie conflittuali, Dynamo ritorna tutte le copie presenti con relative versioni e *clocks*. Un successivo *update* di quel *record* da parte del client viene visto come una riconciliazione del conflitto e tutte le copie vengono collassate nell'unica nuova versione dell'oggetto.

3 Conclusioni

Il punto di forza dei database *key-value* risiede nelle alte performance dovute alla loro semplicità. L'accesso in memoria è diretto, senza un *query language* e senza proiezioni o *filtering* dei dati, e i dati possono essere distribuiti su più storage con un design resistente a eventuali guasti della rete, risultando in un'immediata disponibilità dei dati e in un sistema scalabile. Queste caratteristiche rendono il modello chiave-valore ideale per sistemi *always-on*, con forti requisiti di performance e che devono gestire grandi moli di dati. Per questo motivo, i database *key-value* sono ampiamente utilizzati in casi d'uso come la memorizzazione di sessioni utente in applicazioni web, la gestione del carrello in grandi sistemi *e-commerce* tra cui Amazon, gestione di profili utente e delle loro preferenze o sistemi di *recommendation* di prodotti.

Riferimenti bibliografici

[1] DeCandia G., Hastorun D., Jampani M., Kakulapati G., Lakshman A., Pilchin A., Sivasubramanian S., Vosshal P., Vogels W.; *Dynamo: Amazon's highly available key-value store.*; ACM SIGOPS symposium on Operating systems principles (2007), ACM Press New York, NY, USA, pp. 205–220.

[2] Si Liu, Nguyen S., Ganhotra J., Rahman M. R., Gupta I., Meseguer J., *Quantitative Analysis of Consistency in NoSQL Key-value Stores*, al link `https://assured-cloud-computing.illinois.edu/files/2014/03/Quantitative-Analysis-of-Consistency-in-NoSQL-Key-values-Stores-1.pdf`

[3] Pagina AWS su db *key-value*: `https://aws.amazon.com/it/nosql/key-value/`

[4] Articolo su Medium sul confrontro tra RDB e *key-value*: `https://medium.com/@wishmithasmendis/from-rdbms-to-key-value-store-data-modeling-techniques-a2874906bc46`

[5] Introduzione di Aerospike ai db chiave-valore: `https://www.aerospike.com/what-is-a-key-value-store/`

[6] Pagina Wikipedia sul modello *key-value*: `https://en.wikipedia.org/wiki/Key-value_database`

[7] Introduzione a Redis: `https://redis.io/topics/introductions`