Key-Value Stores e Document Databases

Nell'ultimo decennio i database relazionali sono stati particolarmente apprezzati grazie alla loro flessibilità; purtroppo non sono noti per le loro **performance**. Come accennato nell'ultima parte del capitolo precedente, gli importanti avanzamenti tecnologici degli ultimi anni hanno portato alla luce delle forti limitazioni legate alle caratteristiche dei sistemi relazionali.

L'idea è dunque quella di passare a dei sistemi che siano in generale meno flessibili rispetto a sistemi relazionali sotto alcuni punti di vista, ma che possano adattarsi meglio e con maggiore efficienza ai casi d'uso nei quali è necessaria la loro applicazione.

Key-Value Stores

L'idea alla base di questo approccio è molto semplice: viene costruito un **array associativo permanente**. Come il nome suggerisce, gli elementi chiave di un array associativo sono una **chiave** e **valori** associati alla chiave; volendo fare un paragone con i linguaggi di programmazione, possiamo associare questo concetto a quello di *dizionario in Python* e a quello di *hashMap in Java*. Di seguito andiamo ad elencare alcune delle proprietà fondamentali:

- È possibile accedere a valori (o cancellarli) tramite l'utilizzo delle *chiavi*
- È possibile inserire coppie chiave-valore arbitrarie senza che queste aderiscano necessariamente ad uno schema (**schemaless**)
- I valori possono avere tipi di dato molteplici (liste, stringhe, valori atomici, array, ...)
- È un sistema molto semplice ma veloce: grazie alla semplicità della struttura dati, non abbiamo bisogno di un query language molto avanzato per accedere ai dati. Si tratta di un'alternativa ottimale nel caso di applicazioni *data intensive*.

Proprio riguardo all'ultimo punto, è necessario specificare che il compito di combinare più coppie chiave-valore in oggetti complessi è tipicamente responsabilità dell'applicazione che si interfaccia con il sistema. Alcuni esempi molto comuni di questo approccio sono Amazon Dynamo, Riak e Redis

Map-Reduce

In generale, ci si riferisce a MapReduce come un approccio di programmazione che consente di processare enormi quantità di dati in parallelo sfruttando diversi cluster di calcolo dividendo grandi operazioni in piccoli passi di map e reduce. Nell'ambito dei key-value stores si può vedere la procedura divisa nei seguenti passaggi:

- **Splittare** l'input e iterare sulle coppie chiave-valore in sottinsiemi disgiunti
- Calcolare la funzione di **map** su ognuno dei sottoinsiemi splittati
- Raggruppare tutti i valori intermedi per chiave (**shuffle**)
- Iterare su tutti i gruppi applicare **reduce** in modo da riunire i vari gruppi

Figure 1 illustra chiaramente il funzionamento dei vari passaggi necessari al funzionamento dell'algoritmo MapReduce.

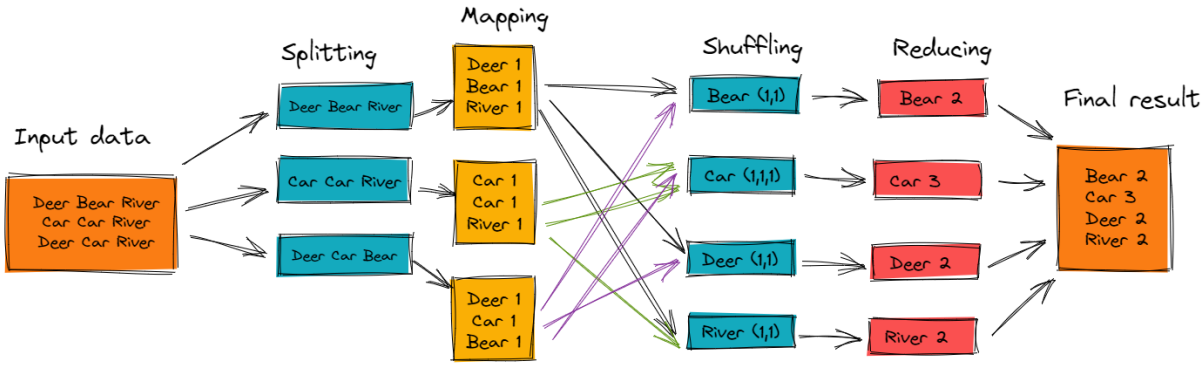

Figure 1: Esempio di applicazione dell'algoritmo MapReduce per contare le occorrenze di ogni parola all'interno di un input

È possibile notare i seguenti aspetti:

- L'approccio è estramamente adatto alla **parallelizzazione**, infatti i passaggi di mapping e di riduzione possono essere eseguiti in parallelo, ad esempio lanciando un processo di map per ogni "frase" o, nel caso dell'esempio, ogni n parole e un processo di *reduce* per ogni parola
- È possibile sfruttare la **località** dei dati in modo da processare i dati direttamente sulla macchina che li sta "ospitando" in modo da ridurre il più possibile il traffico sulla rete.
- È possibile migliorare ulteriormente quanto presente in Figure 1 applicando una procedura di **combinazione**, che consenta di combinare i risultati intermedi invece di mandarli alla procedura di riduzione in formato grezzo, tuttavia non è sempre garantito che questa operazione sia implementata sui sistemi che scegliamo di utilizzare

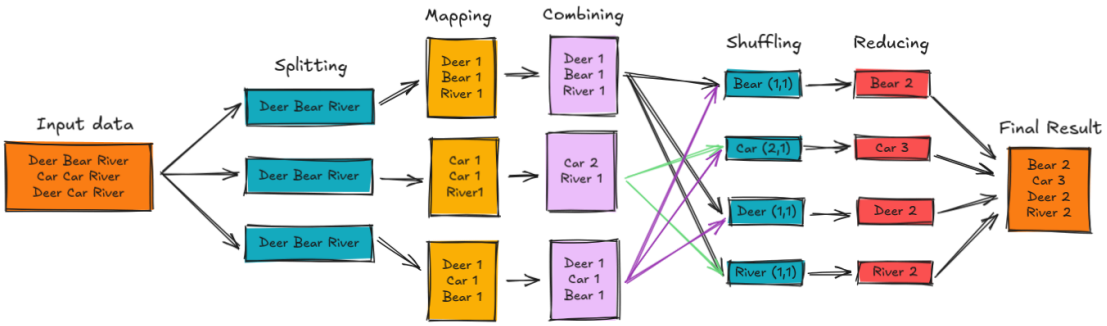

Figure 2: Applicazione della funzionalità di combine al metodo MapReduce

Document Database

Se i key-value store sono metodi molto semplici per memorizzare dati, questo approccio a volte può risultare fin troppo semplice: memorizzare soltanto dati primitivi a volte è fin troppo riduttivo per quelle che potrebbero essere le necessità di un'applicazione.

Per questo motivo nascono i **document database**, i quali permettono di memorizzare documenti in un formato di **testo strutturato** (JSON, XML, YAML, ...). Anche in questo caso ogni documento è identificato da una *chiave univoca*, mostrando quindi una forte correlazione al concetto di key-value store, ma i dati memorizzati hanno un requisito in più sulla loro struttura. Questo è spesso molto utile in quanto ci consente di effettuare validazione dei dati, per esempio tramite XML o JSON schema.

JavaScript Object Notation

Per lo scopo di questo corso non andremo a soffermarci su database di documenti che utilizzano XML per dare struttura ai propri documenti, per semplicità sceglieremo di concentrare la nostra

attenzione su quelli che memorizzano oggetti JSON. Per fare ciò è necessario andare a comprendere quali siano le peculiarità di questo linguaggio:

- Si tratta di un **formato testuale** di facile lettura per rappresentazione di strutture dati
- Ogni documento JSON è sostanzialmente un annidamento di coppie **chiave-valore** separate da un simbolo ":"
- Per dare struttura al documento vengono utilizzate le parentesi graffe "`{}`"
- La chiave è sempre definita tramite una **stringa**, mentre i valori possono essere vari:
  - floating point
  - stringhe unicode
  - booleani
  - array
  - oggetti

> Example: Semplice Descrizione JSON di un oggetto Person

```JSON
{
  "firstName": "Alice",
  "lastName" : "Smith",
  "age"      : 31,
}
```

> Example: Oggetto complesso con figli composti e array di valori

```JSON
{
  "firstName" : "Alice",
  "lastName"  : "Smith",
  "age"       : 31,
  "address"   : {
    "street"    : "Main Street",
    "number"    : 12,
    "city"      : "Newtown",
    "zip"       : 31141,
  },
  "telephone" :  [123456, 908077, 2782783],
}
```

Il linguaggio JSON presenta tuttavia alcune limitazioni:

- Non supporta referenze da un documento all'altro, dunque non è possibile adottare un meccanismo di foreign key come in SQL
- Non supporta referenze all'interno di uno stesso documento JSON

Esistono tuttavia alcuni strumenti per il processing di file JSON che supportano referenze basate su ID: per esempio, è possibile aggiungere una chiave `id` per l'oggetto persona e impostare un valore univoco per questo, per fare in modo di utilizzare tale `id` per riferirci all'oggetto di tipo persona appena costruito in altri oggetti.

MongoDB

Uno degli esempi più noti di document database è sicuramente **MongoDB**. Se volessimo andare a fare un confronto con un DBMS relazionale avremmo le seguenti differenze:

- Capacità di **scalare orizzontalmente** su più macchine
- Rispetto a un DBMS relazionale abbiamo una **miglior località dei dati**; questa proprietà è garantita proprio dal fatto che ogni oggetti contiene tutti i dati di cui ha bisogno, senza necessità di dover effettuare "join" con altre tabelle
- Mancanza di possibilità di far rispettare ai dati memorizzati uno **schema** con conseguente mancata possibilità di validare i dati in ingresso
- Mancata possibilità di eseguire operazioni di **join** per unire risultati
- Mancata possibilità di supportare **transazioni**

Similmente ad un database relazionale, è invece possibile operare query per il recupero di dati o la costruzione di indici sia primari che secondari per migliorare l'efficienza. Per semplicità andiamo di seguito a stabilire un mapping tra un DBMS relazionale e MongoDB:

RDBMS		MongoDB
Database		Database
Table, View		Collection
Row		Document (JSON, BSON)
Column		Field
Index		Index
Join		Embedded Document
Foreign Key		Reference
Partition		Shard

De-normalizzazione

L'aspetto più rilevante nell'utilizzo di questo modello risiede nella **semplicità** con cui è possibile rappresentare e gestire diverse strutture dati. Il concetto chiave a cui si deve questa immediatezza è la **de-normalizzazione**. Come mostrato in Figure 3, la de-normalizzazione semplifica la struttura della base di dati accorpando le informazioni che altrimenti sarebbero distribuite su più tabelle o entità.

Questa scelta, tuttavia, introduce un importante svantaggio: la **gestione delle modifiche ai dati**. Se, ad esempio, un utente compare in più contesti e si rende necessario aggiornare i dati relativi agli ordini a lui associati, sarà necessario applicare la modifica in **ogni copia** presente nel sistema. In caso contrario, la base di dati rischierebbe di diventare incoerente.

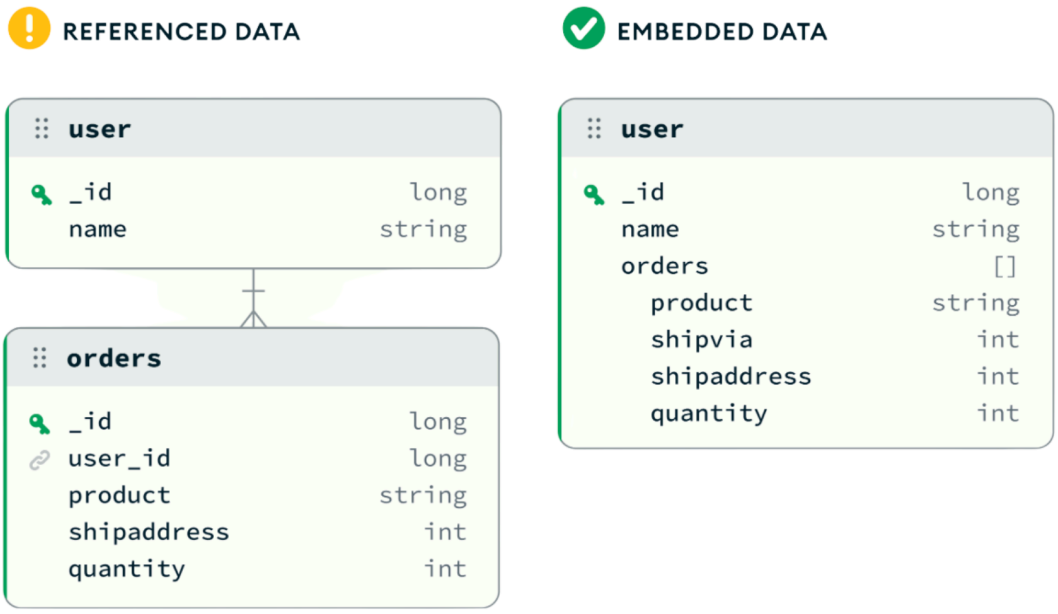

Figure 3: Esempio di de-normalizzazione: a sinistra una struttura normalizzata basata su due entità distinte, a destra una rappresentazione de-normalizzata della stessa relazione.

Salvataggio Atomico

Per quanto abbiamo citato che sia stato abbandonato il concetto di transazioni e delle loro proprietà "ACID", è comunque necessario anche in questo contesto andare a garantire consistenza, specialmente nel momento in cui più processi concorrenti vanno ad interrogare la base di dati. Per questo il paradigma adottato è quello del **salvataggio atomico**, che risulta comunque più semplice e rapido da effettuare rispetto all'esecuzione di una transazione.

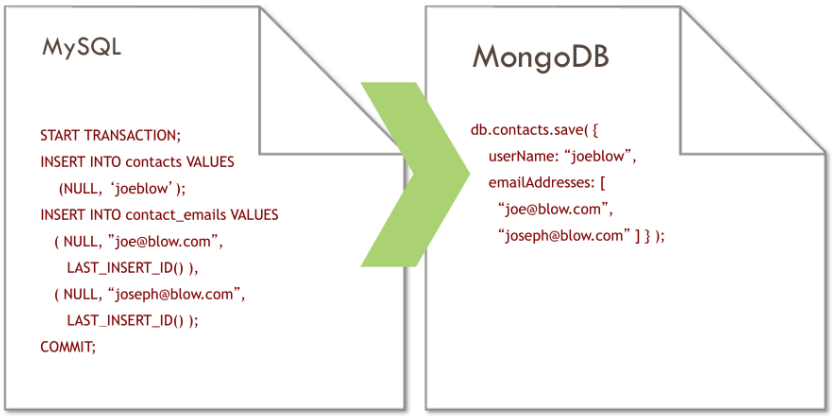

Figure 4: Esempio di Salvataggio atomico

Altre caratteristiche di MongoDB

Di seguito andiamo ad elencare altre caratteristiche peculiari di MongoDB che lo differenziano rispetto ad altri sistemi:

- Ogni documento JSON memorizzato deve essere provvisto di un campo identificativo con nome `_id`, se non fornito, questo campo viene creato in automatico dal sistema
- I dati vengono in realtà memorizzati in formato **BSON** che consiste in una rappresentazione binaria dei dati JSON per garantire maggiore efficienza e semplicità di manipolazione dei dati
- MongoDB è in grado di capire quali sono i dati ai quali sono richiesti accessi più frequenti e di **cachare** in memoria principale i loro valori, così da garantirne un accesso più rapido ed efficiente

CRUD

Questa sezione andrà ad illustrare i vari comandi che è possibile utilizzare per effettuare le operazioni CRUD su MongoDB:

Per quanto riguarda l'operazione di **create** abbiamo a disposizione i seguenti comandi:

- `db.collection.insert(<document>)`
- `db.collection.save(<document>)`
- `db.collection.update(<query>,<update>, {upsert: true})`: prova ad aggioranre un record ma se non esiste nulla che corrisponda alla `query` allora va ad inserire il valore che avrebbe dovuto aggiornare

> Example: Inserimento di un documento
>
> ```JavaScript
> > db.user.insert({
>   firstName : "John",
>   lastName  : "Doe",
>   age       : 39
> })
> ```

Per quanto riguarda l'operazione di **read**, in modo simile a come facciamo in SQL andiamo a leggere tutti i dati che soddisfano una certa condizione:

- `db.collection.find(<query>, <projection>)`
- `db.collection.findOne(<query>, <projection>)`

> Example: Lettura di tutti gli elementi
>
> ```JavaScript
> > db.user.find()
>
> > result : {
>     "_id"       : ObjectId("51.."),  // assegnato automaticamente
>     "firstName" : "John",
>     "lastName"  : "Doe",
>     "age"       : 39
>   }
> ```
>
> Si può notare come in questa occasione, dal momento che non sono stati passati parametri per `<query>`, il database abbia restituito tutti gli elementi della collection

Per quello che riguarda le operazioni di **update**:

- `db.collection.update(<query>, <update>, <options>)`

> Example: Aggiornamento di un documento

```JavaScript
> db.user.update(
  {"_id": ObjectId("51.")}, // query per modificare specifico objectId
  {
    $set: {   // aggiornamento che si intende fare
      age: 40, salary: 7000
    }
  }
)
```

Per ciò che invece riguarda le operazioni di **delete** abbiamo la seguente funzionalità:

- `db.collection.remove(<query>, <justOne>)`

> Example: Eliminazione di un documento in base ad una query

```JavaScript
> db.user.remove({
  "firstName": /^J/     // regexp per identificare tutti i documenti dove firstName inizia per "J"
})
```

Proprietà ACID vs. BASE
Se abbiamo visto che nei database relazionali vengono tenute in forze considerazione proprietà *ACID* (atomicità, consistenza, isolamento, durevolezza); in sistemi come MongoDB è stato preferito richiedere l'aderenza ad un nuovo tipo di paradigma, più lasco, ma che consente maggior efficienza e meglio si adatta ai casi d'uso:

- **Basically Available**: il sistema rimane operazionale anche durante possibili crash parziali di sistema. Anche in questi casi dovrebbe essere possibile l'accesso alla base di dati, assicurando che il servizio continui ad essere disponibile. Si tratta di una proprietà fondamentale in sistemi che richiedono "constant uptime", come ad esempio applicazioni di e-commerce o applicazioni social media
- **Soft State**: questo concetto si riferisce all'idea che lo stato del database potrebbe cambiare nel corso del tempo, anche se non dovessero essere stati aggiunti o modificati dati memorizzati. Queste modifiche sono tipicamente atte al raggiungimento di uno stato che sia consistente per tutti i nodi della base di dati
- **Eventually Consistent**: questo significa che dopo un aggiornamento non è necessario che le modifiche apportate alla base di dati siano rese note ad ogni istanza. Questo è particolarmente utile in un *contesto distribuito*, dove sarebbe altrimenti necessario aggiornare tutte le possibili istanze del dato aggiornato, che si trovano potenzialmente in più locazioni fisiche

Caso d'uso: Location-Based Application
Vogliamo costruire un'applicazione con le seguenti caratteristiche:

- Gli utenti deveono avere la possibilità di effettuare il *check-in*
- Gli utenti possono lasciare *note* o *commenti* riguardo una location

Per ogni **location** vogliamo le seguenti possibilità:

- Salvare il *nome*, l'*indirizzo* e dei *tag*
- Possibilità di memorizzare contenuti generati dagli utenti (*tips*, *note*)
- Possibilità di trovare altre location nei paraggi

Per quanto riguarda invece i **check-in** abbiamo i seguenti requisiti:

- Gli utenti dovrebbero essere in grado di effettuare il check-in
- Possibilità di generare *statistiche* sui check-in per ogni location

In primo luogo è necessario andare a definire le **collection** (tabelle) che andranno in qualche modo a rappresentare le entità coinvolte all'interno del sistema.

Collection `locations`
Per prima cosa andiamo a dare un rudimentale schema per la collection `locations`, che andrà a rappresentare le varie location del nostro sistema:

> Example: Locations v1 - Possibilità di filtrare per zipcode e per tags
>
> ```JSON
> location = {
>   name: "10gen East Coast",
>   address: "134 5th Avenue 3rd Floor",
>   city: "New York",
>   zip: "10011",
>
>   tags: ["business", "offices"]
> }
> ```
>
> Di seguito andiamo a mostrare alcune query che sarà possibile effettuare su questa collection per andarne a visualizzarne i valori:
>
> ```JavaScript
> // 1.trova le prime 10 location con zip code 10011
> db.locations.find({zip:"10011"}).limit(10)
>
> // 2. trova le prime 10 location con tag business
> db.locations.find({tag: "business"}).limit(10)
>
> // 3. trova le location con zip code 10011 e tag business
> db.locations.find({zip: "10011", tags: "business"})
> ```
>
> Si noti come nell'esempio sopra le query 2. e 3. differiscano dalla query 1. , infatti nella prima query andiamo ad effettuare un controllo di uguaglianza con un valore unico, mentre nelle altre due il tag "`business`" si trova inserito all'interno di una lista, per cui il controllo sarà effettuato sugli elementi della lista e basterà trovare un un elemento della lista che corrisponda al valore che stiamo cercando.

Ci piacerebbe andare a memorizzare anche le *coordinate* di una posizione, in modo tale da andare in seguito a ricercare locations vicine ad alcune coordinate.

> Example: Locations v2 - Implementazione di un semplice sistema di coordinate

```JSON
location = {
  name: "10gen East Coast",
  address: "134 5th Avenue 3rd Floor",
  city: "New York",
  zip: "10011",

  tags: ["business", "offices"],
  latlong: [40.0, 72.0]
}
```

Per quanto noi siamo consapevoli che il campo `latlong` corrisponda a delle coordinate, quel campo per MongoDB non è altro che una lista. Anche se MongoDB è nativamente un sistema **schema-less**, si rende a volte necessario aggiungere degli schemi parziali per garantire più efficienza. A questo scopo vengono creati degli **indici**:

```JavaScript
db.locations.ensureIndex({latlong: "2d"})
```

Questo comando ci permette non solo di rendere le nostre query più efficienti, ma anche di andare a forzare il fatto che i valori per il campo `latlong` siano bidimensionali (‘`2d`’).

Per andare ora ad effettuare query che ci permettano di ottenere location vicine a delle certe coordinate possiamo andare ad utilizzare gli **operatori spaziali**:

```JavaScript
db.locations.find({latlong:{$near[40,70]}})
```

È comunque importante menzionare il fatto che per quanto sia possibile andare a memorizzare informazioni spaziali, MongoDB non è sicuramente il sistema più consono a questo scopo. Esistono infatti soluzioni più efficienti e studiate proprio per questo caso d’uso.

Ipotizziamo ora di voler aggiungere la possibilità per gli utenti di aggiungere delle *note* e dei *commenti* su ogni location.

> Example: Locations v3 - Aggiunta la possibilità di lasciare commenti

```JSON
location = {
  name: "10gen East Coast",
  address: "134 5th Avenue 3rd Floor",
  city: "New York",
  zip: "10011",

  tags: ["business", "offices"],
  latlong: [40.0, 72.0],
  tips: [ // lista di oggetti complessi
    {
      user: "nosh", date: "6/26/2010",
      tip: "stop by for office hours on Thursdays",
    },
    {...},
  ]
}
```

Ipotizzando che la v3 sia la versione completa che ci serve per la collection **locations**, andiamo avedere quali sono gli indici che ci sarà necessario definire per avere più efficienza:

```JavaScript
db.locations.ensureIndex({tags:1})
db.locations.ensureIndex({name:1})
db.locations.ensureIndex({latlong:"2d"})
```

Quando andiamo a creare un indice su una lista, questo verrà creato su ogni elemento della lista. Assieme alle possibilità già viste per effettuare query è anche disponibile la funzionalità delle regular expression:

```JavaScript
db.locations.find({name:^typeaheadstring/})
```

Andiamo ora a vedere come sfruttare le operazioni CRUD viste in precedenza applicate a questo contesto. Per andare ad inserire gli elementi nella collection utilizziamo il comando `insert`:

```JavaScript
db.locations.insert(location) // per definizione di location di
vedano gli esempi (v3 in particolare)
```

Per andare a modificare una specifica location andiamo ad utilizzare il comando `update`, specificando una query e come andremo a modificare tale documento; in questo caso andremo ad aggiungere un tip, ipotizzando che questo non fosse già presente:

```JavaScript
db.locations.update(
  {name: "10gen HQ"}, // query
  // push  è usato per aggiungere elementi ad una lista (tips)
  {$push: {tips:
      {
        user: "nosh", date: "2/26/2010",
        tip: "stop by for office hours on Thursdays",
```

```
      }
    }}
  }
)
```

Rappresentazione dei check-in

Per andare a rappresentare i vari check-in degli utenti abbiamo a disposizione varie scelte:

- Possiamo scegliere come con i vari tips, di associarli alla collection delle locations,memorizzando per ogni location una lista di checkin
- Possiamo andare a creare una collection `user` all'interno della quale memorizzeremo per ogni utente i check-in da questo effettuati
- Possiamo utilizzare una nuova collection specifica per i check-in che verrà gestita allo stesso modo di come trattiamo una relazione **many-to-many**

La scelta dell'approccio da utilizzare dipende più che altro dall'utilizzo che andremo a fare dei dati: in particolare, in questo caso, dal tipo di statistiche che vogliamo estrarre (o che vogliamo estrarre più frequentemente rispetto alle altre):

- Ci potrebbe interessare capire per ogni utente quale luogo è stato più frequentato, in questo caso forse è meglio salvare i checkin nella collection degli utenti
- Ci potrebbe interessare capire quale è location più frequentata tra tutte, e in questo caso sarebbe utile avere i check-in come attributo delle locations
- Nel caso in cui abbiamo bisogno di entrambe le statistiche, probabilmente sarebbe il caso di utilizzare una collection separata per i soli check-in

Utenti con check-in

Andiamo a mostrare come sia possibile mostrare i check-in come proprietà di un utente. La modalità non dovrebbe soprendere dal momento che il meccanismo è analogoa quello per i *tips* nella collection delle *location*.

JSON
```
user = {
  name: "nosh",
  email: "nosh@10gen.com",
  ... // altre proprietà dell'utente
  checkins: [
    {
      location: "10gen HQ",
      timestamp: "9/20/2010, 10:12:00",
      ... // altre proprietà
    },
    ... // altri check-in dell'utente
  ]
}
```

Per andare ad estrarre delle statistiche è possibile utilizzare le seguenti query:

JavaScript
```
// estrazione di tutti gli utenti che hanno effettuato un check-in in una location
db.users.find({"checkins.location": "10gen HQ"})
```

```
3
4  // estrae i 10 utenti che hanno effettuato più check-in in una location
5  db.users.find({"checkins.location": "10gen HQ"}).sort({ts:-1}).limit(10)
6
7  // estrae quanti utenti hanno effettuato un check-in in una location dopo un
   certo timestamp
8  db.users.find({
9    "checkins.location": "10gen HQ",
10   timestamp: {$gt: ...$}}
11 ).count()
```

Evidentemente è ancora possibile calcolare statistiche riguardo alle specifiche location, ma è necessario in questo caso andare a scorrere tutti i record della collection `users`, risultando potenzialmente inefficiente nel caso in cui ogni utente abbia effettuato molti check-in.

Rappresentazione separata dei check-in

Possiamo scegliere di gestire separatamente la collection dei vari check-in, per fare ciò andremo a salvare un campo `checkins` all'interno dei record `user` che sarà costituito da una serie di references ai record della collection `checkins`

```JSON
1 user = {
2   name: "nosh",
3   email: "nosh@10gen.com",
4   ... // altre proprietà
5   checkins = [e4af242f, cfeb950a, a542e63e]
6 }
```